PEDECIBA INFORMATICA
Departamento de Investigación Operativa, Instituto de Computación
Facultad de Ingeniería
Universidad de la República

Tesis de Maestría

El Calendario en la Simulación a Eventos Discretos

Leslie Murray

Supervisora: M Sc. María E. Urquhart
Orientadores de Tesis: Dr. Gerardo Rubino y
M Sc. María E. Urquhart

Montevideo, Uruguay
Abril 2003

Dedico este humilde trabajo a Patricia, mi esposa, quien con su actitud siempre altruista y desinteresada me regaló la posibilidad de terminar la Maestría.

Agradecimientos

Seguramente es inevitable que el alcance de los agradecimientos exceda el trabajo de Tesis y se extienda a todo el desarrollo de la Maestría. En tal sentido quiero expresar mi gratitud y mi más sincero reconocimiento a la Prof. María E. Urquhart y a los Dres. Gerardo Rubino y Héctor Cancela, quienes desde sus respectivos ámbitos, y en diferentes etapas, cumplieron conmigo una labor que excede holgadamente sus obligaciones profesionales. Siento que en todo momento atendieron los asuntos de pleno agrado, con claro espíritu solidario y haciendo gala de un elevado sentido de la responsabilidad. Su actitud constituye un extraordinario favor, no sólo a mi persona en calidad de estudiante, no sólo al InCo y a todo su ámbito académico y científico por haberme brindado una formación de excelencia, sino también a su país, el que está en deuda con ellos porque han contribuido a su integración con el mío, haciendo que nuestra frontera no sea más que el camino que nos une.

Estoy orgulloso de haber pasado por los claustros de la Facultad de Ingeniería de la UDELAR y seguro de que llevaré siempre conmigo el recuerdo de estos años de ardua y fructífera labor.

Resumen

Los *Simuladores a Eventos Discretos* cuentan con un bloque, usualmente llamado *calendario*, cuyo propósito es gestionar los *eventos* durante la simulación. A tal efecto el simulador establece un mecanismo en el cual, por un lado va notificando al *calendario* la necesidad de planificar futuros *eventos* y por otro lado va recibiendo del *calendario* la indicación de ejecutar las tareas correspondientes, a medida que los *eventos* van ocurriendo. En la implementación más elemental de *calendario*, los *eventos* cuya ocurrencia no ha sucedido aún, son colocados en una lista simple, ordenada por el valor de tiempo de planificación de los mismos. Esta variante, en la que el orden de ocurrencia de los *eventos* depende su posición dentro de la lista, es eficiente mientras hay muy pocos *eventos* planificados (no más de unas decenas), pero cuando ese número crece, la implementación se torna muy lenta y es preciso recurrir a mecanismos más eficientes.

La gestión de los *eventos* es una de las tareas con mayor peso en la determinación del tiempo de ejecución de la simulación, por lo que la eficiencia del simulador depende fuertemente de la eficiencia del *calendario*. Para la mayoría de las implementaciones, el tiempo de ejecución de sus operaciones depende del número de *eventos* planificados (cuyo orden varía en la práctica desde *10* hasta 10^6) y en menor medida de su distribución temporal. Este contexto compromete la elección del tipo de *calendario* y obliga a elegir una variante adecuada a las condiciones que plantea el modelo a simular.

El propósito de esta Tesis es explorar, al estilo de un estado del arte, las alternativas que existen para implementar el *calendario*, introducir propuestas y variantes, programar un conjunto de ellas y ensayarlas bajo diversas condiciones. Se comienza definiendo formalmente las operaciones del *calendario* y comentando los problemas que el mismo debe resolver. Luego se presentan veintitrés implementaciones, se describen sus características constructivas básicas, se destacan sus principales cualidades y se reportan algunas pautas de carácter comparativo. Dos de las veintitrés implementaciones presentadas constituyen propuestas propias de este trabajo de Tesis. Una de ellas es el mecanismo BSS (*Bounded Sequential Searching*), el cual sirve de base a siete posibles implementaciones. La otra es *Short-Cut List*. En ambos casos se trata de estructuras que alojan los *eventos* en una lista ordenada y que procuran, por distintos medios, reducir el costo de las inserciones apelando a mecanismos para acelerar las *búsquedas secuenciales*.

Una vez presentadas las veintitrés implementaciones se describen las dos propuestas propias con mayor detalle, luego de lo cual se aborda la faz experimental del trabajo. Esta etapa se apoya en un conjunto de implementaciones que fueron programadas para su ensayo, son estas: siete que tienen como base el mecanismo BSS, dos variantes de *Short-Cut List*, tres variantes de lista simple y seis de las implementaciones más conocidas (la estructura de *Henriksen*, *Pairing Heap*, *Skip List*, *Implicit Heap*, *Skew-Heap (Top-Down)* y *Splay Tree*). La metodología empleada en los análisis comparativos consiste en: 1) Elegir un modelo a simular, 2) Realizar la simulación del modelo elegido dotando al simulador del *calendario* a evaluar, 3) Medir el tiempo de ejecución de la simulación, 4) Repetir los puntos 2) y 3) con distintos *calendarios* y cotejar los resultados. El modelo más difundido y aceptado como test de *calendario*, y sobre el cual se realiza la mayoría de los ensayos de esta Tesis, es *The Hold Model*. Se trata de un sencillo mecanismo que consiste en planificar sólo un *evento* por cada *evento* ocurrido. Su principal limitación consiste en mantener constante el número de *eventos*. Este hecho motiva la búsqueda de modelos alternativos y en tal sentido se ensayan algunos *Modelos de Cola*, uno de los cuales resulta apropiado para cubrir aspectos con los que *The Hold Model* no permite experimentar (la variabilidad del número de *eventos* planificados y la remoción de *eventos* distintos del próximo a ocurrir).

A pesar de que el importante desarrollo de los equipos de cómputo ha contribuido a estrechar las diferencias entre los tiempos de ejecución de las distintas implementaciones, una elección adecuada de *calendario* no es un problema trivial, y cuando en una simulación el número de *eventos* planificados alcanza valores del orden de *100* o superior, es una tarea inevitable y crucial en la eficiencia de todo el proceso. Esta Tesis constituye una contribución a presentar formalmente este problema, ofrece un extenso panorama de las soluciones existentes, introduce propuestas y variantes, muestra resultados de ensayos realizados sobre algunas de ellas, señala las diferencias y sugiere algunas pautas de elección. Su resultado más relevante es el destacado desempeño de tres de las implementaciones sustentadas por el mecanismo BSS (*Linked Exact Bottom Up*, *Array Bottom Up* y *Array Top Down*) y el problema abierto más interesante es el avance en la búsqueda de variantes más eficientes para la implementación *Short-Cut List*.

Palabras Clave: Evento, Nodo, Implementación, Simulación a Eventos Discretos

Abstract

Discrete Event Simulators include a block, usually named *calendar*, aimed to manage *events* throughout the simulation. In order to do it, the simulator settles a mechanism through which it warns the *calendar* every time an *event* needs to be scheduled, and gets from the *calendar* the indication to execute the corresponding task as *events* occurs. The simplest *calendar* implementation keeps all generated, but not yet evaluated, *events* in a simple list. Anytime new *events* need to be scheduled they are inserted into the list, and when the tasks due to the lastly occurred *event* are over, the simulator seeks the next one, removes it from the list and executes its corresponding tasks. This option, in which the *event* occurrence order depends upon their position into the list, performs quite good as long as the number of scheduled *events* keeps low all the time (no more than a few tens), but as this number grows higher, the implementation becomes lower, and more efficient solutions must be attempted.

The time consumed by *events* management is critical beside the whole simulation time, therefore the simulator efficiency strongly depends upon the *calendar* efficiency. The execution time of most of the implementations mostly depends on the number of scheduled *events* (whose order may vary from *10* up to 10^6) and also on their temporal distribution. This environment makes the *calendar* election be so dependent on the conditions imposed by the model to simulate.

The purpose of this Thesis is to explore, in a state of the art style, the huge number of existing solutions to work out the *calendar* implementation, to introduce modifications and proposals and to test a set of them under varied conditions. At the beginning the *calendar* operations are formally defined and the problems that it is committed to solve are commented. Afterwards twenty three implementations are introduced, their main characteristics are described, their features are pointed out and some comparative guidelines are reported. Two, out of these implementations are proposals of this Thesis. One of them, called BSS (*Bounded Sequential Searching*) is a mechanism that performs as the basis for seven possible implementations. The other one is called *Short-Cut List*. Both of them allocate *events* in a sorted list, and attempt to reach efficiency by hastening sequential searching through the list.

Once the twenty three implementations are introduced, both proposals are described in more detail and afterwards the experimental side of the Thesis is approached. This part of the work is supported by a set of *calendars* that has been programmed. They are: seven implementations of the BSS mechanism, two of *Short-Cut List*, three of the simple list and six out of the best known ones (the *Henriksen´s* structure, *Pairing Heap*, *Skip List*, *Implicit Heap*, *Skew Heap (Top-Down)* and *Splay Tree*). The procedure to undertake comparative analysis requires: 1) To select a model to simulate, 2) To simulate the selected model being the *calendar* of the simulator the one under test, 3) To measure the simulation execution time, 4) To repeat 2) and 3) with different *calendar* implementations and to compare the results. The best known and most accepted model to test a *calendar*, and over which most of the experiments are performed is *The Hold Model*. It consists of a mechanism that schedules only one *event* at every occurred *event*. Its main lack concerns the number of *events*, that is kept fixed throughout the whole trial. This fact encourages the research for alternative models, and in this sense some *Queuing Models* are tested. One of them results to properly cover the features that the *The Hold Model* does not let to experiment with (variability of the number of scheduled *events* and removal of *events* different than the next to occur).

Even though computers development has contributed to reduce the differences of execution time among all implementations, an adequate election of the *calendar* is still not a trivial problem, and whenever the number of scheduled *events* of a simulation reaches a number of *100* or higher, it is an unavoidable task, and crucial in the whole simulation efficiency. This Thesis is a contribution to formally introduce this problem, it shows many of the existing solutions, it introduces modifications and proposals, it shows the results of experimental trials done over some of them, it points out the differences among them and suggests some guidelines for a proper election. Its most relevant result is the remarkable performance of three of the implementations supported by the BSS mechanism (*Linked Exact Bottom Up*, *Array Bottom Up* and *Array Top Down*) and the most interesting open problem is the research for more efficient implementations of the *Short-Cut List*.

Key Words: Event, Node, Implementation, Discrete Event Simulation

Contenido

1. Introducción

La Simulación a Eventos Discretos (SED) es una técnica para modelar y evaluar sistemas en los que los cambios de estado se producen en un conjunto discreto de instantes de tiempo correspondientes a *eventos* de interés que acontecen a intervalos aleatorios.

Para cada modelo a simular es preciso determinar el conjunto de todos los *eventos* capaces de modificar su estado y establecer las acciones a ejecutar como consecuencia de la ocurrencia de los mismos. El mecanismo de la SED cobra sentido a partir de que las acciones asociadas a ciertos *eventos* contemplan la planificación de nuevos *eventos*. Como consecuencia de ello, a medida que el conjunto de *eventos* planificados crece, sus tiempos de ocurrencia van quedando intercalados de un modo impredecible. Sin embargo e independientemente del número de *eventos* planificados, el simulador debe estar, en todo momento, en condiciones de reconocer el *evento* más pronto a ocurrir y de ejecutar las acciones asociadas al mismo. La tarea de administrar los *eventos*, que es la que en definitiva garantiza la correcta evolución de todo el mecanismo, está a cargo de un bloque usualmente llamado *calendario*.

Debido a que el simulador recurre a la generación de números aleatorios para imitar el comportamiento de las variables del modelo que tienen ese carácter, los resultados de una serie de simulaciones realizadas sobre el mismo modelo presentan, en general, valores estadísticamente dispersos. Para acotar esa dispersión y mantenerla dentro de valores admisibles es necesario, entre otras cosas, promediar los resultados de varias ejecuciones, tomando la precaución de elegir, cada vez, juegos de números aleatorios nuevos e independientes de los anteriores. Esta necesidad de repetir la simulación (eventualmente gran cantidad de veces) torna críticos los tiempos de ejecución y obliga a redoblar esfuerzos para mantenerlos bajos. Una de las tareas con mayor peso en la determinación del tiempo total de ejecución de la SED es precisamente la gestión de los *eventos*.

Se puede decir entonces que el *calendario* debe resolver dos tipos de problema. Uno de carácter eminentemente lógico y operativo cual es el de garantizar el orden correcto de ocurrencia de todos los *eventos*, y otro de carácter puramente cualitativo, cual es el de operar en el menor tiempo posible.

En la práctica las tareas del *calendario* se implementan asociando a cada *evento* un objeto (registro) capaz de recordar información relativa al mismo, por ejemplo el instante para el que está prevista su ocurrencia (*t*) y las modificaciones que la misma ha de provocar en el modelo [15]. El problema se reduce entonces a gestionar adecuadamente los registros, para lo cual el simulador debe contar con una estructura de datos y su correspondiente algoritmo capaces de:

1) Almacenar los registros en el momento de su generación (*insertar*).

2) Extraer, en cualquier momento, el registro asociado al *evento* más próximo a ocurrir, o sea el de más bajo valor de *t* (*remover*).

La estructura apropiada para administrar los registros del modo indicado es la *cola de prioridad* [23, 27]. A diferencia de las *pilas* y *colas* que como tipo de dato abstracto aceptan *inserciones* sin restricción y permiten las extracciones de acuerdo al orden cronológico de las *inserciones*, en las *colas de prioridad* cada registro aloja un valor indicador de su prioridad, de manera que las *inserciones* también proceden sin restricción, pero las *remociones* extraen de la estructura el registro con el valor de prioridad más alto. El *calendario* es entonces una *cola de prioridad* (entendiendo por tal a la estructura de datos más el conjunto de funciones que permiten operar sobre ella) encargada de gestionar los registros en la SED. En el *calendario* la prioridad más alta se corresponde con el valor más bajo de *t* y viceversa y su eficiencia depende de qué tan rápido sea capaz de ejecutar las operaciones 1) y 2) indicadas más arriba, sujeto a una diversidad de condiciones vinculadas fundamentalmente al número de registros contenidos al momento de su ejecución y a la distribución temporal de los mismos.

En su formulación más elemental el *calendario* se implementa colocando los registros en una lista simple. Se puede prescindir de mantenerla ordenada, haciendo las *inserciones* mediante el agregado por uno cualquiera de sus extremos (el mismo por razones de simplicidad, pero no necesariamente) y recorriéndola íntegra en busca del registro con el valor más bajo de *t* en cada *remoción*. O bien se puede mantener la lista ordenada según el valor de *t* de los registros, resultando así las *remociones* operaciones de muy bajo costo (tanto como las *inserciones* en el caso anterior) ya que el registro del próximo *evento* estará siempre en el mismo extremo de la lista. En este último caso las *inserciones* deben recorrerla pero no íntegramente, sino sólo hasta llegar a la posición en la que corresponda alojar el nuevo registro. En un *calendario* implementado mediante este último mecanismo la complejidad, medida a través del número de comparaciones necesarias para llevar a cabo las *inserciones* es ***O****(n)*, donde n es el número de registros en la lista al momento de *insertar*, mientras que el costo de las *remociones* es ***O****(1)*, es decir independiente de *n*.

Si los registros insertados en el *calendario* con el mismo valor de *t* son removidos respetando el orden FIFO, el *calendario* se dice estable. Para algunas implementaciones la estabilidad resulta una propiedad imposible de ser cumplida. Para otras depende de la forma en que se planteen en el código ciertas comparaciones. Por ejemplo, si en una implementación mediante lista simple, con *inserciones* asistidas por *búsqueda secuencial*, cada comparación pregunta si el valor de *t* del registro a *insertar* es "mayor" que el valor de *t* del registro siguiente en el recorrido secuencial y en caso afirmativo se desplaza hacia él, la implementación no resulta estable. Pero si la comparación decide según la condición "mayor o igual", entonces el *calendario* sí resulta estable. La estabilidad suele ser un requisito impuesto por el modelo, para resolver necesidades puntuales, pero estadísticamente hablando no es relevante por lo que una larga serie de operaciones difícilmente presentará diferencias sustanciales en los resultados sea que se trate de una implementación estable o no.

El objetivo principal de esta Tesis es el estudio de las distintas implementaciones de *calendario* y su impacto sobre el tiempo de ejecución de la SED. El trabajo se inicia explorando, al estilo de un estado del arte, las alternativas que existen para implementar el *calendario* se destacan sus principales cualidades y se reportan algunas pautas de carácter comparativo. Dos de las veintitrés implementaciones presentadas constituyen propuestas propias de este trabajo de Tesis. Una de ellas, llamada BSS (*Bounded Sequential Searching*), es un mecanismo que opera sobre una lista ordenada de *eventos*, procurando acelerar las búsquedas mediante la asistencia de una estructura montada sobre la lista. Para ello se define un límite B al tamaño de la lista y al de cualquiera de los segmentos resultantes de su partición. Cada vez que, producto de las *inserciones*, el tamaño de la lista o el de cualquier segmento supera el valor de B, se realiza su partición, lo cual consiste en dar acceso a alguno de sus elementos intermedios. El acceso se da desde un nodo insertado en una lista auxiliar, creada en un nivel superior. El mismo mecanismo de partición vale para las listas auxiliares, por lo que el incremento del número de *eventos* va acompañado del crecimiento del número de nodos auxiliares y por ende de listas auxiliares. La ventaja del mecanismo está en que el ingreso a la estructura se hace a través de la lista auxiliar del nivel más alto, lo que permite el acceso inmediato a posiciones adelantadas de la lista de *eventos* y consecuentemente rápido acceso al punto de *inserción*. La estructura que resulta de la aplicación de este mecanismo guarda estrecho parecido con la versión determinística de *Skip List* (DSL), tanto en su arquitectura como en sus mecanismos de operación, por lo que puede considerarse una variante de implementación de la misma. La otra propuesta, llamada *Short-Cut List* también basa su estructura en una lista ordenada de *eventos*. Al igual que en BSS, su objetivo es acelerar los recorridos secuenciales mediante búsquedas eficientes logrando por lo tanto rápido acceso al punto de *inserción*. Para ello cada elemento de la lista establece dos enlaces, uno hacia el elemento siguiente y otro hacia alguno más adelantado en el recorrido secuencial. Toda búsqueda procede chequeando el elemento adelantado antes que el siguiente. Entonces, según resulte la comparación contra el elemento adelantado es probable que la búsqueda prosiga por él, lo que equivale a tomar un "atajo" (short-cut) en el recorrido secuencial de la lista.

Además de los enfoques de tipo analítico presentados para describir las implementaciones a través de las propiedades de su estructura y de los algoritmos que operan sobre ella (al estilo del análisis realizado unos párrafos más arriba sobre la lista simple), también es pertinente contar con evaluaciones de tipo experimental y a ese efecto se apela en esta Tesis a una metodología que consiste en: 1) Elegir un modelo a simular, 2) Realizar la simulación del modelo elegido dotando al simulador del *calendario* a evaluar, 3) Medir el tiempo de ejecución de la simulación, 4) Repetir los puntos 2) y 3) con distintos *calendarios* y cotejar los resultados. En la mayoría de los experimentos se utilizó un modelo de prueba conocido y aceptado para la evaluación de *calendarios*. Se denomina *The Hold Model* y su evolución se reduce a planificar uno y sólo un *evento* por cada *evento* ocurrido. Una de las variables para ajustar su comportamiento tiene que ver con el tiempo de planificación de cada nuevo *evento*, el cual se determina sumando un valor aleatorio al tiempo del *evento* cuya ocurrencia se está tratando. La otra variable importante de *The Hold Model* es el número de *eventos* planificados, el cual se mantiene constante e igual al valor inicial durante todo el ensayo. Si bien la mayoría de los ensayos de esta Tesis se realiza sobre *The Hold Model*, también se procura extender el alcance de los experimentos hacia algunos Modelos de Cola. La búsqueda de modelos alternativos tiene por objetivo salvar la limitación que impone *The Hold Model* manteniendo fijo el número de *eventos* planificados. Luego de una serie de pruebas, se arriba a un modelo que además de presentar gran variabilidad en el número de *eventos* permite ensayar la *remoción* del *calendario* de *eventos* distintos del próximo a ocurrir lo cual, si bien es poco frecuente en la SED, puede cobrar importancia en el tratamiento de modelos especiales. Se trata de una sucesión o Tandem de colas M/M/1 en el que la primera de ellas incluye determinado tipo de falla como parte de su comportamiento.

Las implementaciones de *calendario* han sido objeto de análisis durante largo tiempo [8, 13, 14, 20,21]. El desarrollo de las *colas de prioridad* y de muchas de las implementaciones de *calendario* tuvo un auge (puesto en evidencia por la fecha de la mayoría de los artículos disponibles) alrededor del año 1980. El principal propósito de su estudio es diseñar algoritmos capaces de administrar los registros de un modo eficiente, contribuyendo así a incrementar la velocidad de la SED. Existe actualmente una enorme variedad de estructuras capaces de

comportarse como *calendario*, algunas de ellas diseñadas específicamente, otras bajo la forma de estructuras clásicas. La clasificación más habitual es aquella que las considera divididas en tipo lista y tipo árbol, pero dado que esta clasificación no guarda una relación estricta con el rendimiento, a los fines del presente análisis resulta tan arbitraria como cualquier otra. Así también se pueden distinguir aquellas que realizan el ordenamiento de los registros por lotes (*en batch*) mediante la aplicación de un algoritmo específico en un instante determinado, de aquellas que realizan el ordenamiento en forma continua. O por ejemplo aquellas que necesitan montar parte importante de su estructura antes de iniciar la simulación, es decir estando vacías, de las que mientras no contienen registros no existen como estructura. Pueden distinguirse también las implementaciones que para operar necesitan que se les suministre el valor de un determinado número de parámetros (típicamente el máximo número de registros a alojar, pero a veces muchos más) de aquellas que no tienen parámetros de operación. También puede resultar práctico considerar una división entre aquellas cuya programación demanda un esfuerzo importante de aquellas de programación sencilla. Por último, una clasificación inevitable es aquella que las considera divididas de acuerdo al costo de sus operaciones como función del número de registros, típicamente evaluado mediante la función ***O( )***.

El mecanismo de la SED es tan simple que permite ser abordado sin más elementos que las herramientas básicas de programación y una cierta cuota de sentido común. No es imposible desarrollar simuladores cuyos mecanismos se aparten de la ortodoxia y de las normas recomendadas para la determinación de los *eventos*, su clasificación en clases, etc., y si el trabajo se realiza en forma cuidadosa es muy factible incluso obtener resultados correctos. Sin embargo es muy difícil obviar el análisis y la consecuente elección de la implementación del *calendario* cuando la magnitud del problema, medida en número de *eventos* planificados, excede cierto límite (unos cientos de *eventos*). En una implementación elemental la elección recae casi invariablemente en la lista simple y en ese caso, por muy sencillas que sean las tareas asociadas a cada *evento*, cuando su número crece razonablemente los tiempos de ejecución se tornan inmanejables. Una adecuada elección de la implementación del *calendario* es una de las claves para llevar los tiempos a valores admisibles.

En la totalidad de los casos abordados en esta Tesis los registros se materializan mediante unidades de memoria capaces de alojar un conjunto de datos entre los cuales destacan, además de los mencionados más arriba, punteros hacia otras unidades similares para permitir establecer enlaces entre las mismas y consecuentemente la construcción de las estructuras correspondientes. Se designa a esas unidades como nodo (casos típicos son los record en el lenguaje Pascal o las structure en el lenguaje C). En toda implementación de lista se llama *Header* al nodo cabecera, o sea al que precede al primer nodo asociado a un registro, y *Rear* al indicador de final o sea el que está a continuación del último nodo asociado a un registro. Para toda estructura se entiende por desborde (*overflow*) a la situación en la que un nodo a *insertar* no encuentra sitio entre los espacios disponibles, ya sea porque están completamente ocupados o porque su valor de *t* rebasa algún límite o umbral fijado para aceptar *inserciones* en el lugar previsto.

La Sección **2** presenta veintitrés de implementaciones de *calendario*, entre las que se encuentran la dos propuestas propias de esta Tesis, el mecanismo *Bounded Sequential Searching* (BSS) y *Short-Cut List*. Estas dos últimas se describen con mayor detalle en las Secciones **3** y **4** respectivamente. La Sección **5** cubre la faz experimental del trabajo; se presentan allí tanto los modelos ensayados como los resultados experimentales obtenidos. Las conclusiones y los problemas abiertos se presentan en la Sección **6**.

2. Implementaciones de Calendario

En esta Sección se comentan veintitrés implementaciones de *calendario*. Dos de ellas constituyen propuestas de esta Tesis, se trata del mecanismo BSS (*Bounded Sequential Searching*) y de *Short-Cut List*. Ambas se desarrollan en detalle en las Secciones **3** y **4** respectivamente. Las restantes son las implementaciones de *calendario* más frecuentemente utilizadas. Se comentan sus características constructivas básicas y se destacan las cualidades más importantes de cada una. Al final se presenta una tabla en la que se resumen los puntos más relevantes citados en el desarrollo de la Sección.

2.1. Linear [15, 23]

Es la más sencilla de todas de las implementaciones. Los *nodos* forman una lista simplemente enlazada, ordenada por valor de *t*. Las *inserciones* se inician con una *búsqueda secuencial* a partir de uno de los extremos y culminan estableciendo los enlaces necesarios para colocar el nuevo *nodo* en una posición tal que se conserve el orden una vez realizada. Las *remociones* se practican en forma directa, retirando del extremo correspondiente el *nodo* que aloja el menor valor de *t*.

2.2. Linear Median Value [21]

Es una mejora a *Linear* lograda mediante dos agregados: los enlaces simples se convierten en dobles para permitir *búsquedas secuenciales* en ambas direcciones y se mantiene una variable con el promedio entre los valores máximo y mínimo de *t* (o sea los *t* de los *nodos* extremos). Según sea t_i (el valor de *t* del *nodo* a *insertar*) mayor, menor o igual al promedio, se elige el extremo por el cual comenzar la búsqueda en cada *inserción*. No necesariamente el promedio temporal se ha de corresponder con el punto medio geométrico de la lista, razón por la cual el método es fuertemente afectado por la distribución de *t* de los *eventos pendientes*. De todas maneras la modificación reporta una mejora sobre *Linear*. Las *remociones* retiran el *nodo* de más bajo valor de *t* cuya posición es siempre conocida, tal como sucede en *Linear*.

2.3. Linear Median Pointer [20]

La idea es la misma que en *Linear Median Value* sólo que en lugar del promedio entre los *t* de los *nodos* extremos se mantiene un puntero al centro geométrico real de la lista. En función de la comparación entre el valor de *t* del *nodo* central y t_i se elige el extremo por el cual comenzar la *inserción*. Ahora existe la certeza de que en todos los casos se elegirá el extremo a partir del cual cantidad de comparaciones necesarias sea la mínima. El criterio de hecho es más efectivo que el de *Linear Median Value*, sólo que el incremento de eficiencia se debe balancear con el costo adicionado para mantener el puntero central. Las *remociones* no difieren de las dos implementaciones anteriores. Una variante (que no introduce mayores cambios de rendimiento) consiste en iniciar las búsquedas desde el *nodo* central eligiendo la dirección de avance de acuerdo a la comparación de t_i contra el valor de *t* del *nodo* central.

2.4. Short-Cut List

Se trata de una propuesta de esta Tesis cuyo objetivo es acelerar las *búsquedas secuenciales* sobre *nodos* que se alojan en una lista enlazada. Independientemente del punto de inicio y de la dirección de avance, las búsquedas del punto de *inserción* en cualesquiera de las tres implementaciones anteriores se basan en el principio de desplazarse hacia el *nodo* contiguo todas las veces. Según resulte la comparación entre t_i y el *t* del *nodo* siguiente, o bien finaliza la búsqueda o bien se practica el desplazamiento, pero el hecho a destacar es que nunca se "saltean" *nodos*. *Short-Cut List* (ver Sección **4**) permite, en cambio, intentar todas la veces desplazarse hacia un *nodo* distinto del inmediato siguiente. Para ello cada *nodo* cuenta con dos punteros que le permiten establecer otros tantos enlaces, uno hacia el siguiente y otro hacia otro *nodo* más adelantado en el recorrido secuencial. Los enlaces que conducen al *nodo* inmediato siguiente constituyen lo que se denomina "camino común" u ordinario, de ahí su nombre: *OPL* (Ordinary Path Link) en tanto los demás constituyen una suerte de "atajo" o salto por sobre una cierta cantidad de *nodos*, por lo que se los denomina *SCL* (Short-Cut Link).

Antes de comparar el valor de t_i con el *t* del *nodo* visto a través del enlace *OPL*, o sea el inmediato siguiente, se lo compara con el *t* del *nodo* ligado mediante el enlace *SCL*. Según resulte esa comparación o se avanza a través del enlace *SCL* o se realiza la comparación con el *nodo* visto por el enlace *OPL*. En definitiva, reconociendo que los enlaces *SCL* permiten "saltos largos" y los enlaces *OPL* "saltos cortos", el algoritmo de

búsqueda procura permanentemente avanzar dando "saltos largos"; cuando uno de ellos no es posible, intenta un "salto corto", y cuando ninguno de los dos es posible se está frente al punto de *inserción*. La dificultad que la geometría de la estructura pretende salvar es justamente la de establecer los enlaces *SCL*, es decir los "saltos largos". En una lista enlazada, de la que sólo se conoce la cantidad de *nodos* alojados, la posición de la cabecera (*Header*) y en todo caso la del final (*Rear*), no resulta sencillo ir en procura de un *nodo* adelantado en el recorrido secuencial a fin de establecer un enlace hacia él. A diferencia de una lista montada sobre arreglo, en la que existen posiciones "absolutas" dadas por el índice, en las listas enlazadas sólo hay forma de reconocer posiciones "relativas", y si únicamente se cuenta con enlaces hacia el *nodo* siguiente, las determinaciones de posición llevan implícito el tránsito a través de todos los *nodos* existentes entre la posición actual y la posición de interés. Esto complica el establecimiento de los enlaces *SCL* a punto tal que no sólo resulta difícil hacerlo en forma determinística o exacta, sino que también se tropieza con dificultades al intentar darles carácter aleatorio, es decir intentando apuntar hacia "algún" *nodo* adelantado en el recorrido.

La forma de resolver el establecimiento de los enlaces *SCL* en *Short-Cut List* consiste en promover los *nodos* insertados hacia diferentes niveles, existiendo un recorrido secuencial exclusivo para cada nivel. Resulta entonces que los enlaces entre *nodos* de un mismo nivel son justamente enlaces *SCL*. Los detalles y mecanismos para promover *nodos* hacia distintos niveles pueden verse en la Sección **4**. A modo ilustrativo se presenta en **FIGURA 2.1** una estructura completa ajustada a una de las versiones de *Short-Cut List*.

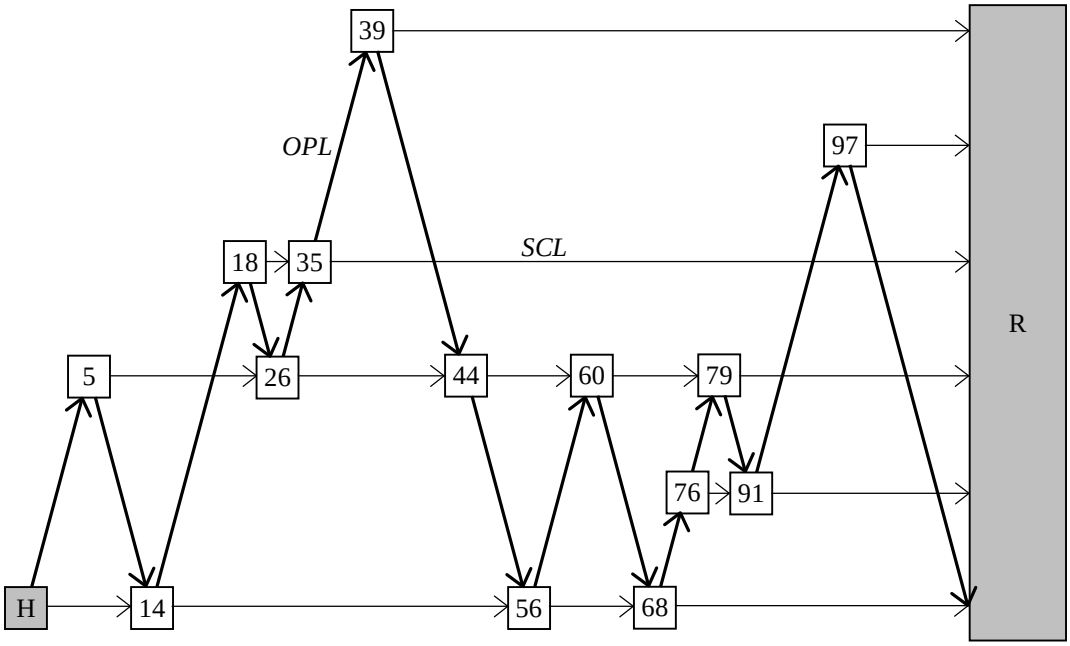


FIGURA 2.1 Short-Cut List

Las *remociones* no ofrecen mayores dificultades pues se trata de la extracción de un *nodo* cuya posición es siempre conocida (apuntado por *Header*) y la restitución de enlaces es bastante inmediata.

El rendimiento de esta implementación depende de que el incremento de costo introducido para mantener la estructura y todos sus algoritmos de operación se cancele con el aumento de velocidad en las búsquedas, producto de los caminos alternativos a través de los enlaces *SCL*.

2.5. Henriksen's [9]

La estructura de *Henriksen* pone en marcha un mecanismo tendiente a evitar el recorrido de la totalidad de los *nodos* intermedios al desplazarse entre dos puntos de una lista enlazada. Para ello apela a dos recursos:

1) Divide la lista en sublistas.

2) Mantiene un arreglo, cada uno de cuyos elementos aloja un valor de t y un puntero a *nodo* de la lista.

El arreglo se declara de una extensión determinada, pero en todo momento sólo se mantiene activo el tramo necesario (de longitud 2^k, con k natural) con los elementos de índice más alto. Cada puntero del arreglo apunta al *nodo* de mayor *t* de cada sublista, y además guarda el valor de *t* del *nodo* apuntado (ver **FIGURA 2.2**).

Las *inserciones* se inician con una búsqueda binaria sobre el arreglo, la cual muy rápidamente conduce al extremo superior de la sublista a la que pertenece el punto de *inserción*. Desde ese extremo superior se inicia una *búsqueda secuencial* por los enlaces de derecha a izquierda, la cual conduce hasta el punto de *inserción* del nuevo *nodo*. El último paso, es decir la *búsqueda secuencial* pura sobre la sublista seleccionada pone de manifiesto la necesidad de que las sublistas se mantengan "razonablemente" cortas. Se fija entonces una cota a la longitud de las mismas en un número *m* de *nodos* y en todo recorrido secuencial sobre un sublista se lleva la cuenta de la cantidad de *nodos* que se van visitando. Si en algún momento esa cuenta alcanza el valor de *m*, inmediatamente se redirecciona el puntero anterior a aquel por el cual se descendió a la lista, hacia el *nodo* que llevó la cuenta a *m*. Si no hubiese un puntero disponible para direccionar o redireccionar, la longitud del arreglo se duplica. Es útil que la longitud activa del arreglo sea siempre 2^k porque ello simplifica la aplicación del algoritmo inicial de búsqueda binaria sobre el mismo.

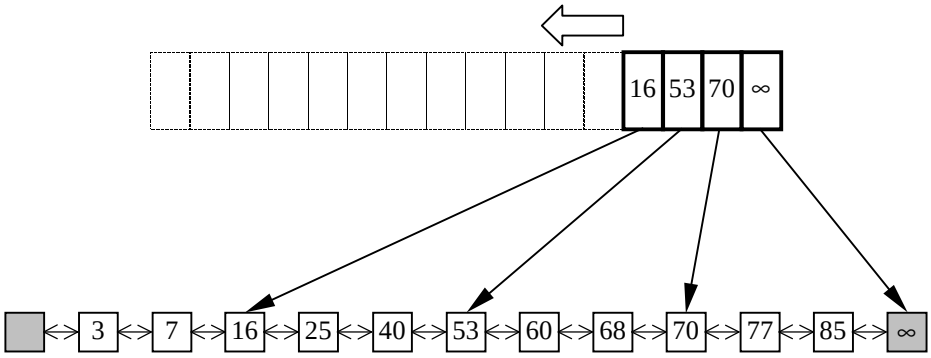


FIGURA 2.2 Estructura de Henriksen

Supóngase que en la estructura de **FIGURA 2.2** se inicia una búsqueda tendiente a *insertar* un *nodo* en la sublista 16-53 y supóngase que mientras se está visitando el *nodo* con *t=25* la cuenta alcanza el valor de *m*, entonces inmediatamente se redirecciona el puntero 16 hacia 25. Si una futura búsqueda requiriese colocar un puntero sobre alguno de los elementos de la sublista 3-25, entonces se le apuntará el anterior a 16, previo a lo cual habrá que duplicar la longitud activa del arreglo.

Las *remociones* operan sin mayor inconveniente por el extremo izquierdo de la lista y lo único que tienen de particular es que podrían dejar sin *nodo* para apuntar a algún elemento del arreglo. En ese caso ese puntero debe tomar un valor *indefinido* y quedar a la espera de ser solicitado nuevamente.

El por qué del doble enlazado es el siguiente:

1) Es preciso contar con enlaces de izquierda a derecha para que exista siempre acceso (desde la cabecera) al primer *nodo*, o sea el de más bajo valor de *t*. Es la única forma de permitir las *remociones* en forma directa.
2) Las *búsquedas secuenciales* sobre las sublistas bien podrían realizarse de izquierda a derecha si se ingresara (desde el arreglo) por el *nodo* con más bajo valor de *t* de la sublista. En ese caso, para que se satisfaga la condición de estabilidad, las comparaciones deberían ser por "mayor o igual", sobrepasando los *nodos* con valor de *t* igual a t_i, ocasionando ello tantas comparaciones adicionales como *nodos* con igual valor de t_i haya. En tanto si se ataca por el *nodo* con el valor más alto de *t* de la sublista y se avanza hacia la izquierda, comparar por "menor" resulta suficiente para garantizar la estabilidad y la cantidad de comparaciones es la mínima.

2.6. Skip List [18]

Skip List surge como alternativa a los árboles binarios, en la expectativa de ofrecer similares prestaciones mediante algoritmos más simples. Es bien conocido que las búsquedas en árboles son tanto más eficientes cuanto más balanceado está el árbol. De hecho hay implementaciones que dedican parte importante de su algoritmo a operaciones de rebalanceo. Algunas lo hacen en forma determinística, forzando balanceos estrictos, y otras lo hacen de un modo probabilístico, procurando mantener la estructura "lejos" de configuraciones nocivas (desbalanceadas). Aún cuando *Skip List* no es una estructura de *tipo árbol* sino *lista*, logra un balanceo probabilístico en forma continua, es decir sin necesidad de ejecutar procesos específicos en instantes

determinados de su operación. Para ello necesita consultar permanentemente un generador de números aleatorios.

Una búsqueda sobre lista ordenada simplemente enlazada podría tener que chequear la totalidad de sus *nodos* (*n*) para realizar una *inserción*. Pero si desde cada *nodo* es posible examinar no sólo el siguiente sino también el que está continuación del siguiente, o sea dos hacia adelante, el máximo número de *nodos* a chequear se reduce a *(n/2)+1*. Para ello es preciso que uno de cada dos *nodos* del recorrido (alternados simétricamente) cuente con dos punteros. Si además se provee a uno de cada cuatro *nodos* (también alternados simétricamente) de un tercer puntero que apunte cuatro *nodos* hacia adelante en el recorrido, la búsqueda demandará no más de *(n/4)+2* comparaciones. Luego, si uno de cada 2^i *nodos* cuenta con un puntero 2^i *nodos* hacia adelante, el número de *nodos* a chequear se reduce a $log_2\ n$. La **FIGURA 2.3** muestra una configuración implementada siguiendo este criterio. Claro que una estructura construida de este modo resulta eficiente para búsquedas y en todo caso para *remociones*, pero resulta imposible mantener su geometría ante *inserciones*.

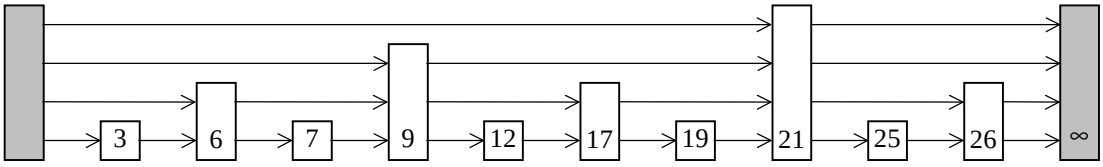


FIGURA 2.3 Distribución Predeterminada de Nodos

Se dice que un *nodo* con *k* punteros es un *nodo* de nivel *k*. Entonces si uno de cada 2^i *nodos* cuenta con un puntero 2^i *nodos* hacia adelante, los niveles se distribuyen según el siguiente esquema: *50%* serán nivel *1*, *25%* nivel *2*, *12.5%* nivel *3*, etc. Una estructura que puede aspirar a un comportamiento similar al anterior es aquella que no resulta de distribuir los *nodos* según el patrón de simetría indicado más arriba sino en forma aleatoria, pero ajustada a una distribución tal que guarde la misma proporción indicada en los porcentajes. Es decir, cuidando que, en valores medios, la mitad de los *nodos* insertados sea de nivel *1*, la cuarta parte nivel *2*, etc. En este caso el *i*-ésimo puntero de cada *nodo*, no deberá apuntar exactamente 2^{i-1} *nodos* hacia adelante sino al próximo *nodo* de nivel *i* o superior, tal como se indica en **FIGURA 2.4**.

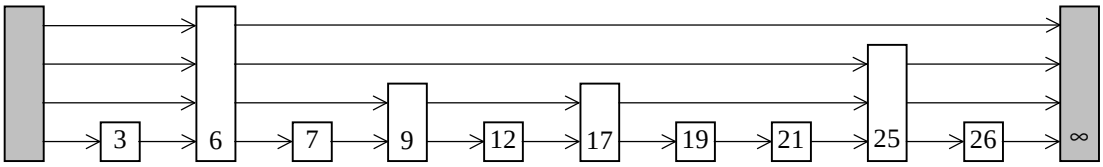


FIGURA 2.4 Skip List

El algoritmo de *inserción* debe iniciar una *búsqueda secuencial* en el nivel más alto de la cabecera de la lista. A medida que el valor del próximo *nodo* a visitar excede al valor de t_i se desciende un nivel. Cuando ya no hay nivel hacia el cual descender se está ante el punto de *inserción*. Llega entonces el momento de elegir el nivel del *nodo* a *insertar* para lo cual se consulta una variable aleatoria *V* con dos valores posibles (por ejemplo *más* y *fin*) y *50%* de probabilidad para cada valor. Mientras el valor leído en *V* es *más*, se van haciendo nuevas lecturas. Cuando se detecta el valor *fin* por primera vez se asigna al nuevo *nodo* un nivel igual a la cantidad de lecturas que se hubieran hecho de *V*.

En una estructura construida de acuerdo a las reglas presentadas hasta el momento, en promedio, la mitad de los *nodos* que tienen puntero en el nivel *i* tiene también puntero en el nivel *i+1*. Pero este es sólo un caso particular de un esquema en el que una fracción *p* de los *nodos* con puntero en el nivel *i* cuenta además con puntero en el nivel *i+1* (en el caso descrito hasta ahora *p=1/2*). El algoritmo para elegir el nivel de un nuevo *nodo*, para cualquier valor de *p*, es el mismo que se comenta más arriba sólo que se deben ajustar de acuerdo a *p* los valores de probabilidad de *más* y *fin*. Existe el peligro potencial, aunque remoto, de que en algún momento la determinación de nivel arroje un valor "extremadamente" alto. La solución más práctica para este problema consiste en guardar, en una variable, el nivel más alto existente en la lista (*L*). Toda vez que alguna determinación supera el valor de *L* se elige *L+1*.

Por último se debe considerar la necesidad de conocer el máximo nivel que la lista puede soportar (entre otras cosas porque ese será el nivel del *nodo* cabecera). Siendo *N* el mayor número de *nodos* que la lista va a alojar, el máximo nivel se deberá elegir como $log_{1/p}\ N$.

Las *remociones* no ofrecen dificultad y se llevan a cabo en tiempo constante.

Debido al mecanismo empleado en la determinación de altura de los *nodos*, la geometría de *Skip List* tiene carácter aleatorio, razón por la cual su rendimiento tiene el mismo carácter. Esto impone la existencia de un peor caso muy desfavorable, a pesar del buen rendimiento medio. La inquietud por acotar el peor caso en las operaciones de *Skip List* motivó la investigación y dio por resultado algunas variantes de carácter determinístico: DSL (*Deterministic Skip Lists*) [17]. La idea de estas implementaciones es ajustar la configuración a un esquema preestablecido, no tan restrictivo cómo el de **FIGURA 2.3**, sino permitiendo ligeras desviaciones que deben mantenerse dentro de límites preestablecidos. Entonces, por un lado la altura de cada nuevo *nodo* ya no es aleatoria sino que depende de la posición en la que ha de ser insertado, y por otro lado los *nodos* adyacentes deben tener cierta capacidad para asimilar la *inserción*, lo cual se traduce en posibilidad de variar su altura. Para salvar esta dificultad algunas variantes de DSL proponen soluciones como por ejemplo la implementación de *nodos* mediante arreglos de punteros o a través de *nodos* especiales con capacidad de enlazarse unos sobre otros.

2.7. Indexed List [15]

La idea de indexar una lista consiste en establecer, según algún criterio, una división de la misma en sublistas, de modo que los extremos de cada una de ellas resulten de "rápido" acceso. La forma más elemental de llevar a la práctica esta idea es colocando sobre la lista cuyos *nodos* se relacionan biunívocamente con los *registros* (enlazada simple o doblemente, según las necesidades del caso), otra lista formada por elementos (*claves* o *índices*) con capacidad de apuntar a los *nodos* (ver **FIGURA 2.5**). Al estar los *nodos* enlazados según el orden creciente de *t*, recorriendo los *índices* en el mismo sentido, se podrán leer (a través de sus punteros) valores de *t* también crecientes. Es posible también implementar estas estructuras de modo que cada *índice* contenga un valor propio de *t* (no necesariamente existente en la lista de *nodos*), que la lista de *índices* siga también un orden creciente de sus tiempos, y que cada *nodo* apuntado contenga el máximo valor de *t* (de la lista de *nodos*) que no supere al del *índice* que lo apunta.

Sea con *índices* que tienen su propio valor de *t* o no, las *búsquedas secuenciales* no se inician por la cabecera de la lista de *nodos* sino por el primero de los *índices* (aquel con más bajo valor de *t*, ya sea propio o apuntado) e independientemente del criterio de búsqueda, la idea es que ese primer barrido secuencial permita determinar la sublista a la que pertenece el punto de *inserción* para poder iniciar luego una *búsqueda secuencial* a través de los *nodos* de esa sublista, finalizando con la *inserción* misma. Según sean los enlaces dobles o simples, los recorridos podrán realizarse en una u otra dirección, no obstante el siguiente análisis es válido en cualquier caso.

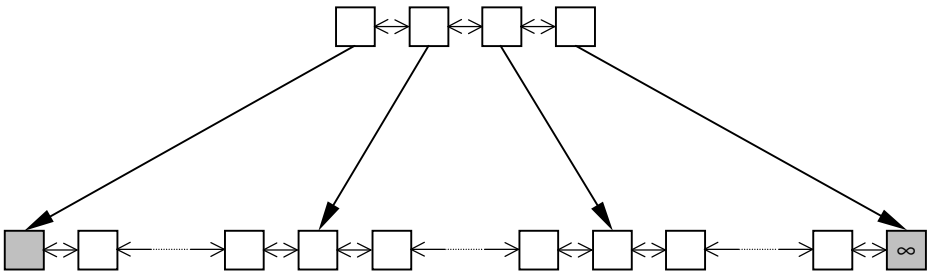


FIGURA 2.5 Indexed List

Siendo *n* la cantidad total de *nodos*, *m* el máximo número de *nodos* por sublista y *k* el número de *índices*, las *inserciones* apoyadas mediante *búsqueda secuencial* pura, es decir sin ayuda de los *índices*, tienen una complejidad (medida en cantidad de comparaciones),

$C = \boldsymbol{O}(n)$

Pero si se inicia el proceso a través de la lista de *índices*, ese valor se convierte en

$C = \boldsymbol{O}(k+m)$

Por distintos medios se busca dar flexibilidad a estas estructuras de modo que puedan adaptarse, en tiempo de ejecución, a las fluctuaciones de *n*, conservando a la vez una relación adecuada entre los demás parámetros de modo de no perder eficiencia.

Mantener constante el valor de *k* implica una gran rigidez puesto que:

1) Es obvia la necesidad de que *k* y *n* guarden alguna relación (piénsese en casos extremos). Si *k* es fijo, la estructura resultará eficiente sólo para una gama de valores de *n*.

2) Según como se distribuyan temporalmente los *eventos*, si los valores de *t* de los índice son fijos, podrían darse acumulaciones de *nodos* a punto tal que se concentre la totalidad en una única sublista, conduciendo esta situación a una peor caso de tipo ***O****(n)*.

Otra alternativa de implementación (en lugar de fijar el valor de *k*) es poner una cota al valor de *m* o incluso asignarle un valor fijo, permitiendo la variación de *k*. Ante cada *inserción* podría ser preciso entonces reapuntar uno o más *índices* o eventualmente agregar uno nuevo. Ahora,

$C = \boldsymbol{O}(k+m) = \boldsymbol{O}(n/m+m)$

Expresión que encuentra su valor mínimo cuando $m = n^{0.5}$, en cuyo caso

$C = \boldsymbol{O}(n^{0.5})$

En este tipo de estructura se plantea también la posibilidad de que los índices no se alojen en una lista enlazada sino en un arreglo. En ese caso la complejidad se puede reducir notablemente e incluso, lo que es mucho más importante, puede llegar a (¡casi!) independizarse del valor de *n*. Esta variante, con arreglo, permite introducir el principio de las implementaciones que exhiben complejidad ***O****(1)*, es decir independencia de *n*. La clave de todas ellas está en la presencia de un arreglo que a través de una operación de *hashing* permite ir de t_i al índice del arreglo cuyo elemento conduce a un segmento, sector de lista o sublista de longitud acotada. La resultante es entonces: "un cálculo matemático más una *búsqueda secuencial* sobre una lista no mayor que un determinado *m*", lo cual resulta bastante independiente del número *n* de *nodos*. Ahora bien, poner los *índices* en un arreglo en lugar de hacerlo en una lista enlazada introduce una serie de complicaciones:

1) La operación de *hashing* no ha de ir más allá de la extracción de la parte entera al resultado de una función lineal (producto del corrimiento de tiempos). Ello hace que los valores de *t* de los *nodos* apuntados desde los *índices* deban estar equiespaciados (pues estarán a la larga en relación directa a los índices del arreglo); luego satisfacer la pretensión de mantener *m* acotado o fijo en un determinado valor resultará imposible ya que la longitud de las sublistas quedará supeditada íntegramente a la forma en que se distribuyan temporalmente los *eventos*. Entonces lo que siga al "cálculo matemático" no será una búsqueda sobre lista de longitud acotada, sino variable.

2) El *calendario* es una estructura con evolución dinámica permanente, y lo que es más, evoluciona al compás de una variable monótonamente creciente (el *tiempo actual*) razón por la cual, aún no mediando cambios en el valor de *n*, es preciso que los elementos apuntados por los índices vayan adecuándose mediante una suerte de "corrimiento". Gráficamente, si se remueven *nodos* permanentemente y a la vez se van insertando otros nuevos, frecuentemente el primero de los *índices* se irá quedando sin *nodo* al que apuntar y a la vez irán haciendo falta nuevos *índices* tanto al final de la lista como en posiciones intermedias. Luego, la función de *hashing* deberá evolucionar "acompañando" el avance de *t* en la simulación. La "independencia respecto a *n*" introducirá una "dependencia respecto a *t*".

3) Que el número de elementos del arreglo sea fijo, sumado a que el intervalo temporal que separa sus elementos también lo sea, hacen que para poder aceptar la *inserción* de *nodos* con cualquier valor de t_i, sea menester contar con un espacio (por ejemplo la última de las sublistas) destinado al desborde (*overflow*), esto es, a posiciones que sobrepasen el alcance del arreglo y que según la distribución temporal de los *evento*s pueda llenarse considerablemente.

4) El arreglo deberá dimensionarse al inicio y luego deberá redimensionarse toda vez que se necesiten más *índices* o cuando sobre un número importante de ellos. Esto implica establecer un criterio, en forma un tanto arbitraria, tanto para la elección de valores umbrales como para los factores de multiplicación y división o cualquier otro criterio que se considere adecuado para hacer crecer y decrecer el arreglo. Estas operaciones introducen un costo adicional tanto mayor cuanto más frecuentes e importantes sean las variaciones *n*, además de la necesidad de conocer, desde el comienzo, la máxima dimensión alcanzable por el arreglo.

En todos los casos las *remociones* operan en tiempo constante y sin mayores diferencias respecto a sus equivalentes sobre implementaciones de *tipo lista*.

2.8. Two Level Structure [5, 6]

Con base en los principios enunciados para *Indexed List* se propone una estructura indexada que mantiene dos niveles de índice, uno sobre lista enlazada y otro sobre arreglo (ver **FIGURA 2.6**).

Los elementos del arreglo determinan intervalos para los cuales se establece una cota tanto en número como en longitud. Esos intervalos se manifiestan como sublistas en la lista enlazada de *índices*, desde la cual se determinan a la vez sublistas sobre la lista de *nodos*. También se establece una cota para la cantidad de *nodos* por sublista e incluso se prevé la *inserción* de *falsos nodos* e *índices* con los valores exactos de los correspondientes en el arreglo.

Una vez determinado el intervalo que contiene el punto de *inserción*, los recorridos secuenciales, tanto de *índices* como de *nodos* se realizan de derecha a izquierda.

Las *remociones* se realizan en tiempo constante.

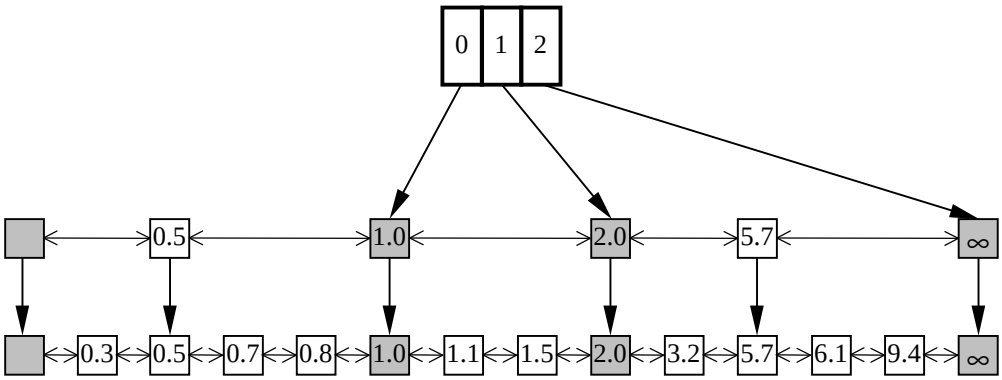


FIGURA 2.6 Two Level Structure

2.9. Two List [1]

Esta opción surge como alternativa para acelerar búsquedas sobre listas enlazadas. Consiste en fijar un valor umbral T_B para decidir en cual de dos estructuras diferentes se debe alojar cada *nodo* insertado. Las opciones son:

1) *Lista 1*, una lista ordenada simplemente enlazada sobre la que se inserta según el procedimiento de rutina, es decir mediante *búsqueda secuencial* del punto de *inserción* y de la que es posible *remover* en tiempo constante.

2) *Lista 2*, una lista que establece una cola simple entre los *nodos*, es decir una estructura tipo *FIFO* en la que, desde luego, no se mantiene el orden según el valor de *t* de los *nodos*.

Si t_i es menor que T_B el *nodo* se inserta en *Lista 1*, caso contrario en *Lista 2*. Al cabo de un cierto número de *remociones Lista 1* se podría vaciar, haciendo que las *remociones* ya no sean posibles. Entonces se inicia un proceso de *reacomodamiento* de *nodos*. Se incrementa primeramente el umbral llevándolo a un valor $T_B+\Delta T_B$, se recorre luego *Lista 2* en busca de *nodos* cuyo valor de *t* sea menor al nuevo umbral y a medida que se van encontrando se los coloca en *Lista 1*, tal como si se tratara de una *inserción*.

Cabe preguntarse si, algorítmicamente, este mecanismo implica o no una ventaja sobre modelos como *Linear*, por ejemplo, ya que la simple observación no permite dar respuesta a este interrogante. Se plantea entonces el siguiente análisis.

Llegado el momento del *reordenamiento* los *n nodos* presentes en la estructura se encuentran en *Lista 2*, pero sólo una cantidad αn va a ser transferida a *Lista 1*. Durante el *reordenamiento* el crecimiento de *Lista 1* es lineal, dado que los *nodos* son transferidos uno a uno. Su longitud pasa de *0* a αn, por lo que su valor medio

durante el proceso es $\alpha n/2$. Una estimación para el valor medio de la cantidad de comparaciones necesarias para alojar en *Lista 1* los αn *nodos* traídos de *Lista 2* es $\alpha n/4$[1].

Por otro lado, cada vez que se lanza un *reordenamiento* hay que barrer completa la *Lista 2* en busca de *nodos* cuyo valor de *t* sea inferior al nuevo T_B. En ese momento *Lista 2* aloja todos los *nodos* presentes en la estructura o sea *n*.

Luego, sumando la cantidad de comparaciones que es necesario hacer en *Lista 2* para detectar *nodos* a transferir y las comparaciones necesarias en el armado de *Lista 1* a medida que ésta va creciendo, el valor medio de la cantidad de comparaciones (*C*) en cada *reordenamiento* podrá ser estimado por

$$C = \alpha n/4 + n$$

Sea ahora una simulación completa, implementada para correr hasta que ocurran *E eventos*, es decir hasta que se hayan practicado *E remociones*. Por simplicidad supóngase que el modelo a simular es *The Hold Model* (ver Sección **5.1**), razón por la cual *n* se mantiene constante y además acéptese que el modelo ha alcanzado ya su estado estable. La cantidad total de comparaciones (C_E) a lo largo de la simulación será

$$C_E = N\,C + C_I$$

Donde *N* es la cantidad total de *reordenamientos* necesarios, *C* la cantidad de comparaciones de cada *reordenamiento* y C_I la cantidad de comparaciones necesarias para practicar las *inserciones*.

Luego de cada *reordenamiento*, *Lista 1* se llenará con αn *nodos*. Esos αn *nodos* comenzarán a ser removidos uno a uno mediante *remociones*, por lo que en todo momento *Lista 1* contará con un número de *nodos* variable entre *1* y αn, pudiendo aceptarse entonces $\alpha n/2$ como su valor medio. Llegado el momento de las *inserciones* estas podrán tener lugar tanto en *Lista 1* como en *Lista 2*, siendo las más costosas aquellas que opten por *Lista 1*. Una cota razonable entonces para el valor medio de las comparaciones necesarias durante las *inserciones* es $\alpha n/4$, o sea la mitad de la lista a recorrer. Se está ensayando *The Hold Model*, por lo que a las *E remociones* le corresponderán *E inserciones*; entonces en valor medio la cantidad de comparaciones realizadas para *insertar* podrá estimarse por

$$C_I = E\,(\alpha n/4)$$

En cuanto a *N*, admitiendo que la cantidad *E* de *eventos* a ocurrir sea suficientemente más grande que αn, la mayor cantidad de *reordenamientos* tendrá lugar si todas las *inserciones* optan por *Lista 2* debido a que los vaciamientos de *Lista 1* serán más frecuentes. En ese caso *Lista 1* se vaciará tantas veces como quepa αn dentro de *E*.

$$N = E/\alpha n$$

Finalmente,

$$C_E = E/\alpha n\,[\alpha n/4 + n] + E\,(\alpha n/4) \qquad (*)$$

La variable cuyo valor es posible elegir es α, e interesa ver para que valor hace mínima la expresión anterior,

$$dC_E/d\alpha = 0 \;\Rightarrow\; \alpha' = 2/n^{0.5}$$

Luego el número óptimo de *evento*s a transferir desde *Lista 2* hacia *Lista 1* durante los *reordenamientos* es

$$\alpha' n = 2\,n^{0.5}$$

Reemplazando αn *por* $\alpha' n$ en *(*)* , dividiendo por *E* y admitiendo que $n^{0.5} >> 1/4$, el costo promedio de cada acceso al *calendario*, medido en cantidad de comparaciones, es

$$C_E = n^{0.5}$$

Si bien este resultado no permite responder al interrogante planteado respecto a la comparación de rendimiento entre *Two list* y *Linear*, sí permite asegurar que hay una diferencia de complejidad a favor de *Two list* que al crecer *n* tornará mucho más favorable esta última: $\boldsymbol{O(n^{0.5})}$ contra $\boldsymbol{O(n)}$. Además queda demostrado que, más allá de cuestiones de implementación, *Two list* es una alternativa a la que no es necesario suministrar parámetros para adecuar su rendimiento a las condiciones de la simulación.

[1] La mitad de la cantidad de *nodos* de una lista es una estimación de costo, medido en número de comparaciones, para *búsquedas secuenciales* sobre la misma. Es un medida fuertemente influida por la distribución temporal y únicamente válida cuando los tiempos se distribuyen uniformemente.

2.10. *Bounded Sequential Searching* - **BSS** [16]

El mecanismo conocido como *Búsqueda Binaria* o *Dicotómica* es capaz de realizar búsquedas sobre listas simples, ordenadas, realizando el menor número de comparaciones posible. Se basa en el principio de practicar sucesivas particiones, en las que cada una da origen a dos sublistas de igual tamaño. La sola comparación entre el elemento buscado y el que determina la partición (elemento central) permite saber, cada vez, a cual de las dos sublistas generadas pertenece el punto de búsqueda[2]. Comenzando por partir la lista original, de tamaño *n*, el proceso finaliza cuando ya no son posibles más particiones, lo cual sucede en no más de $log_2\ n$ particiones, o lo que es lo mismo, luego de no más de $log_2\ n$ comparaciones.

Si bien la *Búsqueda Binaria* es el método que demanda el menor número de comparaciones, desde el punto de vista algorítmico es un caso particular de otro, menos efectivo, que va partiendo las listas en *B* sublistas, en lugar de hacerlo en dos. Claro que luego de cada partición ya no alcanza con una comparación para determinar a cual de las *B* sublistas resultantes pertenece el punto buscado, sino que son necesarias *B-1* comparaciones (contra el extremo superior de cada sublista, por ejemplo). El proceso finaliza cuando ya no son posibles más particiones, previo a lo cual es necesario realizar no más de $log_B\ n^{B-1}$ comparaciones (ver Sección **3**).

BSS es un mecanismo propuesto en esta Tesis que, mediante el apoyo de una estructura *auxiliar* que se va creando a medida que ocurren *inserciones*, hace que las búsquedas guarden estrecho parecido con esta última modalidad de particiones sucesivas. Para ello se establece un valor límite *B*, entero, y contra él se compara el número de *nodos* en la lista luego de cada *inserción*. Mientras ese valor se mantiene no mayor que *B* no se hace nada. Ni bien *B* es superado se crea una segunda lista (*lista auxiliar*), de un sólo elemento (*nodo auxiliar*) con capacidad para:

1) Enlazarse dentro de una lista.
2) Apuntar a un *nodo*.
3) Alojar un valor de *t* igual al del *nodo* apuntado.

La *cabecera* y el *final* de la nueva lista apuntarán a la *cabecera* y el *final* de la lista de *nodos* respectivamente, y el *nodo auxiliar* a algún *nodo* de la lista de *nodos*[3]. De aquí en más las *búsquedas secuenciales* no se inician en la cabecera de la lista de *nodos* sino en la cabecera de la *lista auxiliar*. Según resulte la comparación de t_i con el *t* del *nodo auxiliar* se decide en cual de los dos segmentos de la lista de *nodos* proseguir la *búsqueda secuencial*.

Producto de futuras *inserciones*, los segmentos determinados pueden crecer hasta superar el valor de *B*. Ello conduce a la necesidad de instalar un nuevo *nodo auxiliar* apuntando a algún *nodo* del segmento que hubiera superado el valor de *B*. Sucesivas *inserciones* podrían llevar incluso el número de *nodos auxiliares* a superar *B*. En ese caso una segunda *lista auxiliar* deberá instalarse sobre la existente, con el mismo criterio que fue instalada la primera sobre la lista de *nodos*.

La estructura alcanza una conformación tal que nunca es necesario recorrer más de *B nodos* en ninguna de las listas. Las *búsquedas secuenciales* resultan entonces acotadas por el valor de *B*, de ahí el nombre dado a este mecanismo: *Bounded Sequential Searching - BSS* (Búsqueda Secuencial Acotada).

Si bien pareciera existir una dependencia entre el costo de la *remociones* y el tamaño *n* de la cola, debido a que existe una probabilidad, tanto más alta cuanto mayor sea *n*, de que la *remoción* de un *nodo* deba ser acompañada de la(s) *remoción*(*es*) de algún(os) *nodo*(*s*) *auxiliar*(*es*), esa dependencia se manifiesta únicamente en el costo de peor caso, pero no en el promedio que de hecho resulta ***O****(1)*, ya que si bien la altura de las "columnas" a *remover* es *n*-dependiente, vale el criterio de que "cuanto más altas, menos frecuentes", y ese criterio rige de un modo tal que cancela la relación entre la altura promedio y *n*.

El valor de *B* es el que determina en última instancia la velocidad de las búsquedas. De acuerdo a lo expresado en párrafos anteriores *B=2* pareciera ser la mejor elección (*B=2* aproxima *BSS* lo más posible al modelo de *Búsqueda Binaria*), pero también se nota que es la opción que da lugar a la creación del mayor número de *listas auxiliares* y a la mayor densidad de *nodos auxiliares* en cada una de ellas. Por lo tanto *B=2* es el valor más exigente para el programa en cuanto al manejo de *nodos*, genera la mayor cantidad de solicitudes de "alocación" y "desalocación", el establecimiento de la mayor cantidad de enlaces y la realización de la mayor cantidad de cálculos y operaciones vinculadas a la *inserción* y *remoción* de *nodos* de todo tipo.

[2] Sea este un elemento existente o un sitio en el que se procura *insertar* un nuevo elemento.
[3] Los distintos criterios para determinar el *nodo* apuntado en el nivel más bajo dan origen a distintas implementaciones del mecanismo *BSS*

En otro extremo se encuentra la elección de un valor inmensamente grande para *B*. Desde el punto de vista del esfuerzo computacional asociado al manejo de *nodos* cuanto mayor sea *B* tanto más rápido responderá el programa, pero no menos cierto es que la velocidad de las búsquedas (medida en cantidad de comparaciones necesarias) irá creciendo junto con el valor de *B*, pudiendo llegar al extremo (en que *B* tenga un valor siempre superior a *n*) en el que no se genere ninguna *lista auxiliar* y por ende la búsqueda se transforme en secuencial pura sin mecanismo *auxiliar* para acelerarla (***O****(n)*).

Este tipo de análisis conduce a la necesidad de elegir el valor o la gama de valores de *B* que llevan a cada implementación de *BSS* a su mejor rendimiento.

2.11. Calendar Queue [2]

Dentro de las soluciones catalogadas como de *tipo lista*, *Calendar Queue* forma parte de una clasificación conocida como *multilista*, habida cuenta que no consiste en particionar una lista mediante marcas o punteros hacia determinados elementos, sino en distribuir los *nodos* entre un cierto número de listas no conectadas entre sí.

Su principio de operación es el de un calendario o agenda de uso personal, de ahí su nombre. Básicamente se trata de establecer un intervalo de tiempo (*año*) y subdividirlo en un número entero de intervalos (*días*). Cada lista del esquema *multilista* representa un *día* y como tal aloja *nodos* cuyo valor de *t* le corresponden. El *año* se materializa mediante la secuencia ordenada de *días* y la forma de recorrerlo es desplazándose a través de un arreglo de punteros a la cabecera de cada una de las listas (ver **FIGURA 2.7**).

En principio el funcionamiento es el siguiente: las *inserciones* se resuelven determinando, mediante un sencillo proceso de *hashing*, a cuál de los *días* pertenece el *nodo* a *insertar* y estableciendo luego, mediante *búsqueda secuencial*, el punto de *inserción* en la lista correspondiente. Las *remociones* deberían encontrar el *nodo* con valor más bajo de *t* en la primer posición del primero de los *días* del *año* todavía ocupado. Ahora bien, el *año* es una intervalo que desde luego va quedando "desactualizado" a medida que la simulación progresa (tal vez no en su dimensión pero sí en el valor de sus extremos). El valor de *t* de los *eventos* que se planifican tiene una tendencia permanente a crecer, por lo que desde el mismo comienzo es probable que se generen *nodos* cuyo valor de *t* exceda el *tiempo actual* en un valor superior al tiempo restante del presente *año*. Esto ocasionaría una desborde (*overflow*) del *calendario* o eventualmente la necesidad de modificarlo. La forma de resolver este problema es colocando el *nodo* a *insertar* en el *año* en curso, pero alojándolo como si su valor de *t* fuese igual al resto de la división entera entre su valor real y el largo del *año*, sin modificar el valor de *t* registrado en el *nodo*. Si por ejemplo el año actual es *a* y se necesita planificar un *evento* para el *día d* del año *a+k* (o sea *k años* más adelante), se lo planifica para el *día d* del *año* en curso. Luego, al barrer el *calendario* se encontrará que ciertos *nodos* alojados en determinados *días* exceden el límite superior del mismo. Se deberá concluir entonces que pertenecen a algún *año* futuro y no se los *removerá* hasta llegado el mismo.

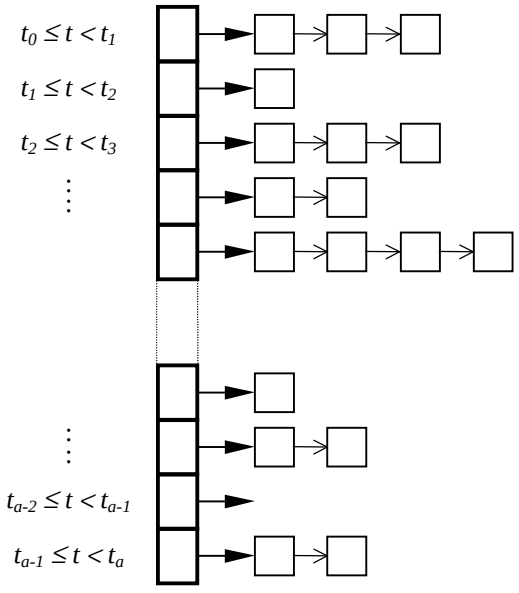


FIGURA 2.7 Calendar Queue

Este ingenioso mecanismo permite prescindir de sectores destinados al desborde (*overflow*) y de sus consecuentes peligros, para el caso que crezcan exageradamente. Como contrapartida esto hace que, a diferencia de casi todas las implementaciones de *tipo lista*, el *nodo* asociado con el próximo *evento* a ocurrir no esté siempre en el mismo sitio de la estructura (por ejemplo en alguna posición extrema). La forma más efectiva de llevar a cabo las *remociones* consiste entonces en mantener conocida la posición (*día*) del último *nodo* removido. En esa lista se inicia una *búsqueda secuencial* entre los *nodos* restantes. Si quedan uno o más pertenecientes al *año* en curso (y por lo tanto al *día* en curso) esos serán los próximos a *remover*, pero si todos correspondieran a otro año se pasa al *día* siguiente. Es necesario entonces conservar el valor de *t* del primer *nodo* encontrado que corresponda a otro *año* así como su ubicación dentro de la estructura, porque podría darse el caso de que ya no queden *eventos* planificados para el presente *año* y la elección deba resolverse entre *nodos* de *años* futuros. La elección recaerá siempre sobre el *nodo* de mínimo valor de *t* entre los primeros de cada lista.

Existe una limitación evidente al proceso de *inserción* (no así al de *remoción* ya que los *nodos* a *remover* estarán siempre en el primer lugar de su lista). Cada *día* es una lista a ser barrida en forma secuencial, razón por la cual es preciso que se mantengan "suficientemente" cortas. A la vez es necesario que no "muchos" *días* estén vacíos ya que esto forzaría una cantidad de comparaciones inútiles. La forma de resolver este inconveniente es *redimensionando* el *calendario* de manera tal que toda vez que una *inserción* haga que *n* supere en uno al doble de la cantidad de *días*, la cantidad de *días* disponibles se duplique y, toda vez que una *remoción* lleve el número *n* a uno por debajo del número de *días* el número de *días* disponibles se divida en dos.

Estas operaciones tienden a hacer que el valor medio de *nodos* por lista no se aleje de uno, pero resultan, de un costo muy elevado ya que dada la coexistencia de *nodos* de distintos años en la misma lista, es indispensable copiar uno a uno todos los elementos del *calendario* hacia una estructura nueva en cada *redimensionamiento*.

Se recomienda aprovechar los *redimensionamientos* no sólo para dividir o multiplicar por dos la cantidad de *días* del *año* sino también para modificar sus longitudes, por supuesto conservando una relación adecuada entre las mismas y la cantidad total de *nodos*.

Este último punto pone en evidencia una cuestión fundamental de esta implementación: la elección de sus parámetros es crucial para su rendimiento. Todo indica que debería exhibir una marcada tendencia hacia una complejidad ***O(1)***, pero para ello es fundamental que la cantidad de *redimensionamientos* sea mínima y que valores como la longitud del *año* y la del *día* (y consecuentemente su cantidad), tanto al inicio como durante la simulación sean adecuados al modelo que se esté simulando. Esta es la razón por la que, aun cuando es una de las implementaciones que exhibe mejor rendimiento, *Calendar Queue* no puede ser considerada como alternativa de propósito general y debe, en todo caso, quedar reservada para problemas específicos cuyos parámetros de operación se conozcan con suficiente detalle.

2.12. Lazy Queue [19]

Se trata de una opción de *tipo multilista* en la que el espacio de tiempo para planificar *eventos* se divide en tres zonas, conocidas como *NF: Near Future* (Futuro Cercano), *FF: Far Future* (Futuro Lejano) y *VFF: Very Far Future* (Futuro Muy Lejano). Para ello se establecen valores límite o de frontera que permiten determinar, por simple comparación, en cual de los tres sectores corresponde hacer cada *inserción*.

NF aloja los *nodos* con más bajo valor de *t*, es decir aquellos asociados a los *eventos* más próximos a ocurrir. Es el sector previsto para hacer las *remociones*. Se implementa mediante la combinación de dos estructuras, un arreglo de *nodos* ordenados por valor de *t* y una lista de *nodos* enlazados, también ordenada por valor de *t*. El arreglo es una estructura apta para practicar *remociones* pero no *inserciones*, por eso es necesario contar también con una estructura adecuada para recibir *nodos* (la lista enlazada). De hecho las *inserciones* se harán exclusivamente en la lista mientras que las *remociones* extraerán el *nodo* con más bajo valor de *t*, pudiendo ser éste el primero de la lista o el primero del arreglo. El por qué de la coexistencia de dos estructuras diferentes en este sector está en que el arreglo le es pasado directamente desde *FF* en determinados instantes, y así como es recibido es conservado hasta su vaciamiento, mientras que la lista enlazada tiene el único propósito de mantener ordenados los *nodos* que arriban a *NF* desde el exterior, o sea como producto de *inserciones*.

La zona *FF* cubre un extenso intervalo de tiempo, de hecho es el sector en el que se espera se produzca la mayoría de las *inserciones*. Se implementa mediante un conjunto de arreglos desordenados, cada uno de los cuales aloja *nodos* con valores de *t* pertenecientes a intervalos consecutivos. La unión de los intervalos determina una partición en el segmento de tiempo cubierto por el sector (si *FF* cubriera el espacio de un año, los arreglos podrían representar, por ejemplo, cada uno de los meses). Toda vez que, producto de sucesivas *remociones*, se vacía el arreglo ordenado de *NF*, se toma el primer arreglo ocupado de *FF*, se lo ordena mediante un algoritmo rápido y se lo convierte en el arreglo de *NF*.

Por último, *VFF* es el sector encargado del desborde (*overflow*). En él se alojan *nodos* cuyo valor de *t* excede el límite del más alto de los arreglos de *FF*. Se trata de una estructura con capacidad de acomodar ordenadamente las *inserciones*, por ejemplo una lista enlazada como la de *NF*. Esta necesidad obedece al hecho siguiente: en determinados reacomodamientos de los valores de frontera entre los tres sectores puede ser necesario transferir *nodos* desde *VFF* hacia *FF*. Esa transferencia será más eficiente (requiriendo un único barrido) si los elementos de *VFF* están ordenados.

La mayor parte de las *inserciones* se produce como un agregado en la primer posición vacía de un arreglo (*FF*) y la mayoría de las *remociones* mediante la extracción del primer elemento de un arreglo que se va vaciando (*NF*). En ambos casos las posiciones son de acceso inmediato e independiente de la cantidad total de *nodos n* alojados en la estructura. Aceptando que las operaciones para mantener los límites y la conformación estructural de *Lazy Queue* en valores adecuados (*redimensionamientos*) son poco frecuentes, su comportamiento se puede considerar razonablemente cercano a *O(1)*, tanto para *inserciones* como para *remociones*. Ahora bien, el costo a pagar está en lo engorroso de la determinación de los principales parámetros de operación, tanto en lo que hace a sus valores iniciales como a los criterios de su evolución a durante la simulación. A tal efecto se establecen como relevantes y capaces de reflejar el estado de la estructura los siguientes parámetros:

1) El número total de *nodos* en la lista de *NF*.

2) El número total de *nodos* en *VFF*.

3) El promedio de *nodos* por arreglo de *FF*.

4) El número total de *nodos* en la mitad (correspondiente a los valores más altos de *t*) de los arreglos de *FF*.

5) La altura y el ancho de los picos detectados en la distribución de *t* de los *nodos*.

Sobre todos estos valores pesan condiciones que se chequean inmediatamente luego de *inserciones* y *remociones*. La comparación contra valores umbrales así como la confrontación de algunos parámetros entre sí dan origen a reglas que deberán ser satisfechas en todo momento. A su vez no todas las reglas son compatibles en cuanto a su cumplimiento, por lo que también es preciso establecer prioridades y criterios de "desempate" entre las mismas.

A partir de las reglas se decide en que momento es necesario *redimensionar* la estructura lo cual consiste únicamente en:

1) Dividir en dos o duplicar el número total de arreglos de *FF*.

2) Dividir en dos o duplicar el tamaño del intervalo de *t* asociado con cada arreglo de *FF*.

Es habitual que 1) y 2) se combinen de manera que, por ejemplo, al duplicar el número total de arreglos se divida en dos el intervalo y viceversa. No obstante aplicaciones independientes también son frecuentes.

Entre los motivos para admitir que *Lazy Queue* exhibe un buen rendimiento se mencionan:

1) La mayor parte de las operaciones se materializa mediante acceso a posiciones de arreglos con conocimiento del valor de índice de la posición a acceder.

2) El proceso de ordenamiento es demorado tanto como es posible y realizado a demanda, en contraposición a casi la mayoría de las implementaciones que lo ejecutan, implícitamente, en cada *inserción*.

3) El hecho de ordenar mediante algoritmos específicos, reduce la sensibilidad a asimetrías o picos en la distribución temporal de los *eventos*.

2.13. SPEEDES-Q [34]

Synchronous Parallel Environment for Emulation and Discrete-Event Simulation Q_{heap} (*SPEEDES-Q*) es la *cola de prioridad* encargada de la gestión de *evento*s en el entorno *SPEEDES*. Se basa en el principio de distribuir los *nodos* entre dos sectores *Q* y Q_{temp}. Las *inserciones* se practican siempre por el mismo extremo de Q_{temp}, que es una lista simple, razón por la cual se hacen en tiempo constante. La única característica de Q_{temp} está en la necesidad de mantener siempre conocido el mínimo valor de *t* de los *nodos* en ella alojados ($mín_{temp}$). Ese mantenimiento consiste en su eventual actualización luego de cada *inserción*.

Las *remociones* son un poco más complicadas. *Q* es una estructura ordenada que permite conocer siempre el *nodo* con más bajo valor de *t*, pero no hay garantía de que ese sea el *nodo* a *remover*, primeramente hay que comparar su valor de *t* con $mín_{temp}$ para saber si el *nodo* a *remover* está en *Q* o en Q_{temp}. A partir de esa comparación surgen dos casos posibles:

Si el mínimo valor de *t* de la estructura es $mín_{temp}$, primeramente se ordena la lista Q_{temp} mediante aplicación de un algoritmo rápido. Una vez obtenida la versión ordenada de Q_{temp} se le desconecta el primer *nodo* (el de valor de $t = mín_{temp}$), se lo devuelve como resultado de la *remoción* y se setea el valor de $mín_{temp}$ a ∞. A la lista resultante se la reconoce como un *metanodo* de valor de *t* igual al del primero de sus *nodos* componentes (el de más bajo valor de *t*) y como tal se lo inserta en el lugar apropiado de *Q*. Producto de estas operaciones *Q* resulta ser una lista de *metanodos* sobre la que pesa además una limitación en su tamaño, esto es, si el número de elementos de *Q* supera un determinado valor umbral *S*, todos sus componentes son transformados en uno único. De hecho el *metanodo* resultante de ordenar Q_{temp} y extraerle el primer *nodo* podría hacer superar el umbral *S*, en cuyo caso todos los *nodos* y *metanodos* de *Q* deberán "fundirse" en uno único, quedando convertida *Q* en una lista ordenada de *nodos*.

Si el mínimo valor de *t* de la estructura corresponde al primer elemento de *Q* pueden suceder dos cosas, que se trate de un *nodo* o que se trate de un *metanodo*. Si es un *nodo* el caso se resuelve inmediatamente como la *remoción* del *nodo* cabecera de una lista simple. Si se trata de un *metanodo* primeramente se le debe desconectar el cuerpo (esto es: dejar el *nodo* cabecera del *metandodo* enlazado a *Q* y constituir en nueva cabecera del *metanodo* al siguiente) dando origen esta desconexión a un nuevo *metanodo* el cual se debe *insertar* en *Q*. Hay que cuidar que está última *inserción* no lleve a superar el valor umbral *S*; si así fuera todos los elementos de *Q* se deberán fundir en uno. Finalmente se remueve de la cabecera de *Q* el *nodo* con menor valor de *t*.

Cabe aclarar que como producto de sucesivas *inserciones* de *metanodos*, *Q* va tomando la forma de una lista de cuyos elementos se desprenden nuevas listas, de cuyos elementos se desprenden nuevas listas, ... Es decir una suerte de "lista enlazada recursiva" que desde luego requiere de una algoritmo específico tanto para las *inserciones* como para la refundición de todos sus elementos en una única lista, siendo esta última la operación más costosa de esta implementación. Dada su estructura *Q* verifica la propiedad de *Heap* (ver Sección **2.15**).

2.14. Cascade [12]

Es una nueva variante de *tipo multilista* que consigue, mediante un ingenioso algoritmo, responder en tiempo amortizado $\boldsymbol{O}(log_2\ n)$ a las dos operaciones básicas del *calendario*. El principio en el que se sustenta es la operación de fusión o *merge* entre listas simples, combinado con una adecuada elección de sus dimensiones y con una administración de las mismas que permita un rápido recorrido de sus cabeceras. La operación de *merge* entre dos listas consiste en la unión de sus elementos en una única lista, conservando el orden dado por el valor de *t* de todos los *nodos*.

La base de la estructura es un conjunto ilimitado de listas L_i ($i \geq 0$), ordenada cada una de ellas por el valor de *t* de sus *nodos*. Existe un límite, de la forma $k_i = k \cdot 2^i$, para la dimensión de cada lista (dónde *k* es una constante, habitualmente en el rango de *5* a *20*), de modo que el límite para L_i es el doble del de su predecesora, L_{i-1}. Cada vez que en alguna lista se rebasa ese límite, se dice que en ella ocurre un desborde (*overflow*).

Las *inserciones* se practican agregando el *nodo* a *insertar* en L_0, cuidando mantener el orden de la misma (sería un *merge* entre una lista cuyo único elemento es el *nodo* a *insertar* y la lista L_0). Cuando una *inserción* provoca el desborde (*overflow*) de L_0 se fusionan, mediante la operación *merge*, L_0 y L_1. El resultado de la fusión se convierte en la nueva L_1, quedando L_0 vacía y disponible para recibir futuras *inserciones*. Según el número de *nodos* resultantes en L_1 luego de la fusión, podría producirse en ella un nuevo desborde (*overflow*). En ese caso se procede de un modo análogo entre L_1 y L_2. El proceso iterativo se detiene cuando en un determinado momento resulta una L_x con menos de k_x *nodos* y, por ende, L_{x-1} vacía.

Las *remociones* resultan bastante simples por cuanto implican barrer todas las listas ocupadas en busca del *nodo* con menor valor de *t*, chequeando sólo los primeros de cada una. La *remoción* se practica entonces en forma directa, retirando el primer *nodo* de la lista correspondiente y sin necesidad de proceso ulterior alguno.

La eficiencia de esta implementación está directamente relacionada a dos parámetros:

1) La cantidad de listas ocupadas.

2) El máximo número de *nodos* para cada una.

El mecanismo de evolución de *Cascade* es tal que en todo momento sólo un pequeño número de listas no están vacías. Siendo *j* el índice de la última de las listas ocupadas, en algún momento debe haber ocurrido un desborde (*overflow*) en L_{j-1}, habiéndose concentrado en ella, a lo sumo, los *n nodos* presentes en la estructura, $n \geq k_{j-1} = k \cdot 2^{j-1}$, luego $j \leq 1 + log_2\ n - log_2\ k$, de donde el número de listas ocupadas es $\boldsymbol{O}(log_2\ n)$, siendo esa también la complejidad para las *remociones*. Las *inserciones* presentan su peor caso cuando se genera un encadenamiento de desbordes (*overflows*) en todas las listas, entonces el comportamiento es $\boldsymbol{O}(n)$, no obstante el valor medio para una secuencia suficientemente larga es, al igual que para las *remociones* $\boldsymbol{O}(log_2\ n)$.

Se ha intentado (mediante ensayos con implementaciones en lenguaje *MODULA-2*) una mejora sustituyendo las listas por otro tipo de estructura fusionable, como por ejemplo *Heaps* o *Leftist Trees*, pero los resultados son fuertemente afectados por la pérdida de posibilidad de *remover* con ***O(1)*** de cada estructura, como es el caso de las listas, pesando este hecho más que el beneficio que reporta *insertar* en ***O(*log_2 *n)*** cuando en las listas se lo hace con ***O(n)***. Es por eso que la opción presentada, en la que cada L_i es una lista simple, es aceptada como la más eficiente.

2.15. Implicit Heap [15, 23, 27]

Dentro de los árboles binarios existe una variedad, llamada *Heap*, que verifica las siguientes propiedades:

1) Cada *nodo* aloja al menos un valor escalar (*t* en este caso).

2) Ningún *nodo* tiene hijos con valor de *t* menor al propio.

Heap es una estructura que ofrece ciertas facilidades para la implementación del *calendario*, la más evidente de las cuales es que el *nodo* vinculado al *evento* más próximo a ocurrir, es decir el de más bajo valor de *t*, es el *nodo* raíz del árbol, por ende el más fácil de acceder.

Existe un gran número de variantes en las que los *nodos* se mantienen ligados según una estructura tipo *Heap*. Lo que las hace diferentes es:

1) La forma en que se realiza una *inserción*, habida cuenta que es necesario que luego de la misma la estructura mantenga la condición de *Heap*.

2) La forma en que a partir de los dos árboles resultantes al extraer el *nodo* raíz se construye otro, con los *nodos* restantes, que satisfaga la condición de *Heap*.

Implicit Heap resuelve estos dos problemas tal como lo hace el clásico algoritmo de ordenamiento *Heapsort*. Ante una *inserción* aloja el nuevo *nodo* en el primer lugar desocupado del nivel más bajo (el de los *nodos* más alejados del raíz), comenzando a revisar de izquierda a derecha. Si la presencia del último *nodo* agregado hace que el árbol resultante no satisfaga la condición de *Heap*, se procede a intercambiarlo con su padre. Y así se prosigue hasta restablecer la condición de *Heap* (ver **FIGURA 2.8**).

El mecanismo para las *remociones* es similar. Una vez extraído el *nodo* raíz, se retira de la estructura el *nodo* ubicado más hacia la derecha del nivel más alejado del raíz y se lo coloca en su reemplazo. Este movimiento hará que el árbol resultante indefectiblemente viole la condición de *Heap*.

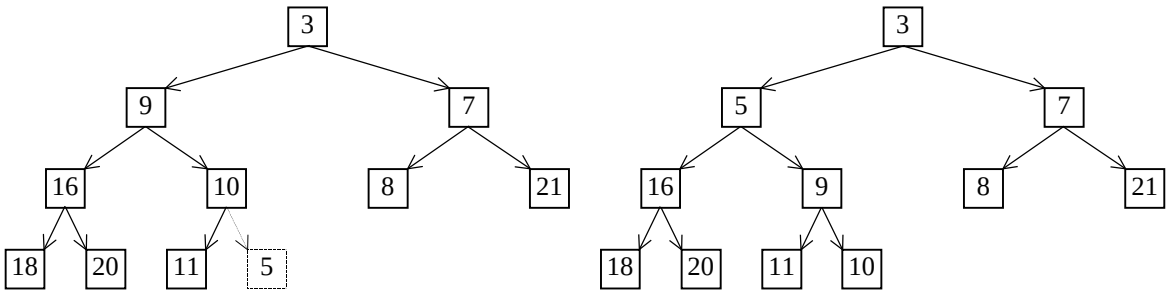


FIGURA 2.8 Implicit Heap

El algoritmo comienza entonces a "mover hacia abajo" el nuevo *nodo* raíz en procura de reconstituirla. Para ello va comparando su valor de *t* con el de sus hijos y ni bien encuentra uno de ellos con un valor menor, realiza el intercambio. El proceso prosigue hasta que la condición de *Heap* se restablece.

Tanto en las *inserciones* (ascenso) como en las *remociones* (descenso) el máximo número posible de intercambios es igual al número total de niveles, expresión que en todo árbol binario está acotada por log_2 *n*, donde *n* es el número total de *nodos* contenidos al momento de la operación.

Existe un modo de implementar árboles binarios que de hecho es el más frecuentemente utilizado en el caso de *Implicit Heap*. En esta opción el soporte del árbol no es una estructura de *nodos* enlazados sino un arreglo en cada una de cuyas posiciones se aloja un *registro*. La regla de formación de *Implicit Heap* sobre arreglo es entonces:

1) La raíz del árbol se aloja en la primer posición del arreglo (la de índice más bajo).

2) Para todo sub-árbol, si el padre se encuentra en la posición *p*, el hijo izquierdo se encuentra en la posición *2p* y el hijo derecho en la posición *2p+1*.

Esta regla de construcción da origen a una configuración en la cual los movimientos (equivalentes a "ascenso" y "descenso" a través del árbol) para las operaciones de intercambio, tanto en *inserciones* como en *remociones* son muy sencillos.

1) El pasaje de padre a hijo se hace mediante cálculo directo aplicando la regla 2) de más arriba.

2) El pasaje de hijo a padre se hace dividiendo por dos el número de posición.

FIGURA 2.9 muestra el Heap de **FIGURA 2.8** montado sobre arreglo (luego de la *inserción*).

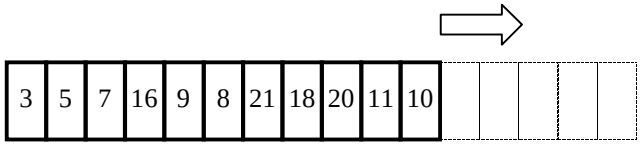


FIGURA 2.9 Implicit Heap sobre Arreglo

Otras facilidades ofrecidas por esta modalidad de implementación son,

1) El primer lugar desocupado, en el nivel más bajo (el de los *nodos* más alejados del raíz), revisando de izquierda a derecha, es la primer celda desocupada del arreglo.

2) El *nodo* ubicado más hacia la derecha del nivel más alejado del raíz es el que ocupa la celda de índice más alto del arreglo.

2.16. Skew Heap (Top-Down) [25]

He aquí una de las variantes para restablecer la condición de *Heap* inmediatamente luego de *inserciones* y *remociones*. Trabaja sobre la base de una única operación llamada *meld*, encargada de convertir dos *Heaps* en uno solo que contenga la totalidad de los elementos alojados en los dos originales. Ante esta facilidad, una *inserción* consiste en ejecutar *meld* entre el *Heap* en su estado actual y un *Heap* formado sólo por el *nodo* a *insertar*. Una *remoción* se resuelve haciendo *meld* entre los dos *Heaps* resultantes al extraer el *nodo* raíz. Queda claro entonces que toda la interpretación se reduce al algoritmo *meld*. Su mecánica es la siguiente:

Sean H_1 y H_2 los dos *Heaps* a combinar[4] y H el *Heap* resultante de la combinación. Si alguno de ambos es nulo, H es el otro y se termina. Si eso no sucede entonces,

1) Si es necesario, reasignar los nombres de los dos *Heaps* para que H_1 sea el de menor valor de *t* en su *nodo* raíz.

2) Designar como H a H_1.

3) Tomar H_1, "desconectarle" el hijo derecho (con todo el sub-árbol que cuelga él) y colocar el hijo izquierdo en su reemplazo. Llamar H_1 al árbol "desconectado".

4) Si H_1 es nulo ir a 7).

5) Comparar el valor de *t* de H_1 con el *t* del *nodo* raíz de H_2. Si (*t* de H_1) > (*t* de H_2) ir a 6), caso contrario ir a 3).

6) Designar a H_1 como H_2 y dejarlo aparte. Colocar el que hasta este momento era H_2 en el lugar que ocupaba H_1, designarlo como H_1 y volver a 3).

7) Colocar a H_2 en la posición desde la que se intentó desconectar H_1 (y resultó nulo) y terminar.

[4] Los *Heaps* son referidos mediante un puntero a su *nodo* raíz. Así H_i es un *Heap* tal que el puntero H_i apunta a su *nodo* raíz.

2.17. Binomial Queue [29]

Así como las implementaciones de tipo *multilista* se caracterizan por no *alojar* los *nodos* en una única lista sino por distribuirlos en un conjunto, *Binomial Queue* reparte los *nodos* en un conjunto de árboles, no directamente conectados entre sí, dando origen a un *bosque*. Los árboles son del tipo conocido como *árbol binomial*. No son binarios y se ajustan a ciertas restricciones que facilitan las operaciones:

1) Verifican la propiedad de *Heap*.

2) Cada uno de ellos aloja, un número de *nodos* igual a 2^k, con $k=0,1,2,...$

Designando a cada árbol de 2^k *nodos* como B_k, B_0 resulta ser un árbol de un solo *nodo*, mientras que todo árbol B_j se puede obtener "agregando" un árbol B_{j-1} a otro árbol B_{j-1}. **FIGURA 2.10** ilustra algunos casos.

El principio fundamental en el cual se apoya Binomial Queue es aquel según el cual "todo número entero *N* puede representarse, de manera única, mediante la suma de no más de $log_2 N$ sumandos de la forma 2^x, para $x=0,1,2,...$". Si el entero a descomponer es *n*, o sea la cantidad total de *nodos*, existe entonces un único *bosque* de *árboles binomiales* que permite alojarlos. Luego, sin perjuicio de que los *nodos* puedan ocupar distintas posiciones, dando así origen a "diferentes" *Binomial Queues*, la cantidad y dimensión de *árboles binomiales* es única para cada valor de *n*. Así por ejemplo $n=13$ deberá implementarse mediante el *bosque* $\{B_3,B_2,B_0\}$. Por extensión de esta última notación se puede representar la estructura de la *Binomial Queue* para $n=13$ mediante el número *1101*, atendiendo al hecho de que existe un árbol de $k=3$, uno de $k=2$, ninguno de $k=1$ y uno de $k=0$. El hecho de que *1101* es la notación binaria del decimal *13* confirma lo anticipado en cuanto a que la descomposición es única.

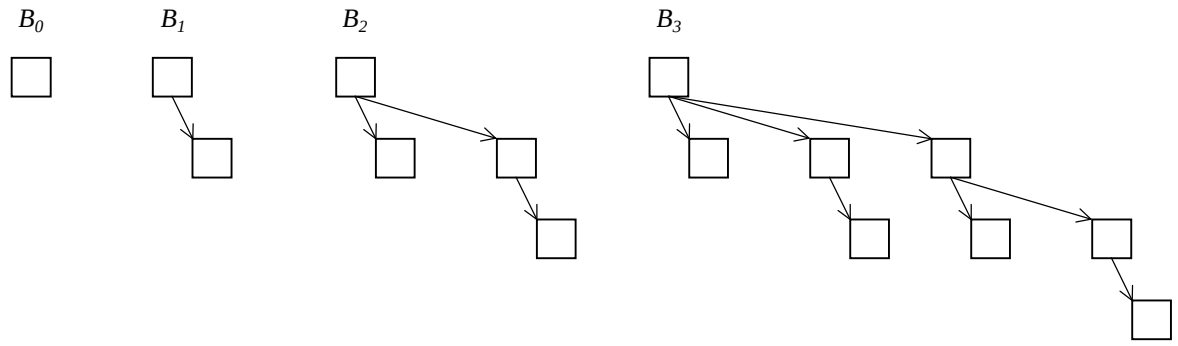


FIGURA 2.10 Árboles Binomiales

Antes de entrar a analizar *inserciones* y *remociones* cabe destacar un hecho importante: se dijo que todos los árboles verifican la propiedad de *Heap*, razón por la cual el *nodo* de más bajo valor de *t* de la *Binomial Queue* será el *nodo* raíz de alguno de sus árboles, y dado que hay un máximo de $log_2 n$ árboles, esa será la cota (medida en número de comparaciones) para determinar el *nodo* a *remover*. Si sólo se trata de conocer la ubicación del *nodo* que ostenta el menor valor de *t* pero sin intención de *removerlo*, hay un modo todavía más rápido y simple que barrer los *nodos* raíz de los árboles. Consiste sencillamente en mantener un puntero al mismo, actualizándolo luego de *inserciones* y *remociones*. Sólo la actualización por *remoción* puede adicionar un pequeño costo, pero fuera de ello se puede conseguir prácticamente una determinación ***O(1)***.

Al igual que en otras tantas implementaciones relacionadas al *Heap*, las dos operaciones fundamentales se resuelven a través de una tercera, de fusión o *merge* de dos *Binomial Queues*, siendo ésta la que se comenta seguidamente.

El *merge* entre dos *árboles binomiales* se realiza en tiempo constante puesto que la única dificultad se reduce a determinar cual de ellos tiene menor valor de *t* en el *nodo* raíz. Luego se le "conecta" el otro como sub-árbol, "colgado" del raíz mediante un nuevo enlace, tal como se muestra en **FIGURA 2.11** (este criterio responde al hecho de que el árbol resultante debe verificar la condición de *Heap*).

La heurística de la operación *merge* debe contemplar que, aún cuando sean factibles "geométricamente" fusiones de árboles cualesquiera, no todas las combinaciones son posibles sino sólo aquellas que dan por resultado un árbol con 2^k *nodos* (por ejemplo, no tiene sentido el *merge* entre los árboles B_1 y B_2).

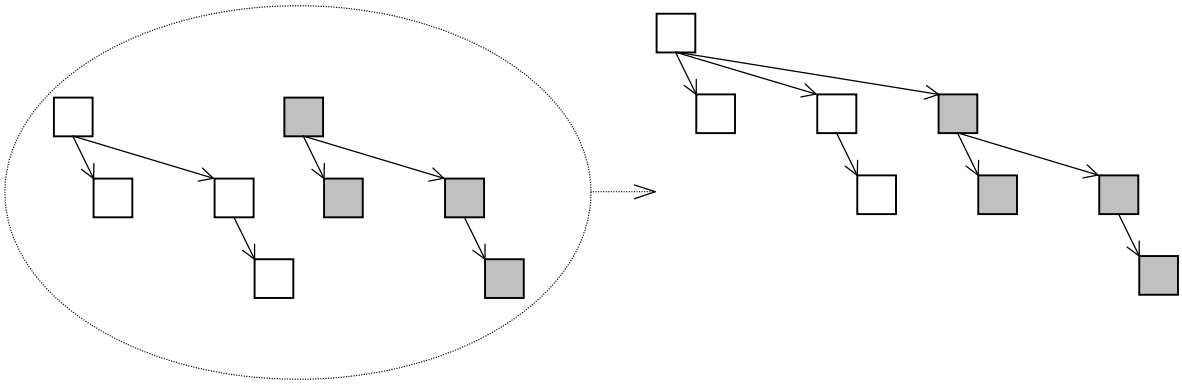


FIGURA 2.11 Merge de dos Árboles Binomiales

El algoritmo de *merge* entre *Binomial Queues* debe reconocer el conjunto de todos los árboles de que dispone, determinar los árboles que deberá contener la *Binomial Queue* resultante e ir haciendo sucesivos *merges* entre los árboles disponibles en todo momento, esto es entre los originales de las dos *Binomial Queues* a fusionar y/o los que se vayan generando durante el proceso. Supóngase la fusión de $BQ_1=\{B_2,B_1\}$ con $BQ_2=\{B_2,B_1,B_0\}$, o sea dos *Binomial Queues* de *6* y *7 nodos* respectivamente. Independientemente de los valores de *t* de sus *nodos* es seguro que el *merge* tendrá una composición $BQ_3=\{B_3,B_2,B_0\}$, o lo que es lo mismo, *13 nodos*. El proceso puede resumirse en los siguientes pasos, teniendo en cuenta que el Objetivo se debe armar comenzando por los árboles más pequeños, pudiendo resultar que estos existan entre los Disponibles o que se vayan obteniendo mediante la fusión de dos de ellos.

1) Disponibles: $1\times B_0$, $2\times B_1$ y $2\times B_2$. Objetivo B_0.

   Inmediato.

2) Disponibles: $2\times B_1$ y $2\times B_2$. Objetivo B_2.

   Ya no harán falta árboles B_1 en el Objetivo, por lo que corresponde primeramente convertir a los dos existentes en único B_2, quedando la situación modificada a,

3) Disponibles: $3\times B_2$. Objetivo B_2.

   Se elige el de menor valor de *t* en el *nodo* raíz.

4) Disponibles: $2\times B_2$. Objetivo B_3.

   Inmediato.

Una *inserción* se resuelve mediante *merge* entre la *Binomial Queue* existente y otra formada únicamente por el *nodo* a *insertar* como B_0. Una *remoción* debe primeramente localizar el árbol con menor valor de *t* en su raíz, supóngase que sea de tipo B_k. Al *removerlo*, para ser devuelto por la función, se generarán los árboles B_0, B_1, B_2,... , B_{k-1}, que serán reconocidos como una *Binomial Queue* aparte. Entre ella y lo que quedara de la original se hace *merge* para restituir una única.

2.18. Leftist Tree [33]

Se trata de una variante de *Heap*, cuya distribución de *nodos* se ajusta a una regla que afecta el largo de los caminos descendentes que parten desde cada uno.

Llamando *nodos* externos a todos aquellos que tengan menos de dos hijos y llamando distancia entre dos *nodos* a la cantidad de enlaces que hay que atravesar para ir de uno a otro (siempre y cuando pertenezcan a distintos niveles y el "más alto" pertenezca al camino que lleva desde el "más bajo" hasta el raíz), *d(x)* es la distancia desde un *nodo x* hasta el *nodo* externo más cercano[5]. *Leftist Trees* se caracteriza porque para todo *nodo x*, $d(hi(x)) \geq d(hd(x))$, donde *hi(x)* e *hd(x)* son hijo izquierdo y derecho del *nodo x* respectivamente. Esta propiedad hace que el camino más corto desde el raíz hasta un *nodo* externo sea el que resulta de avanzar a través del hijo derecho todas las veces, o sea el "más hacia la derecha" de todos los caminos descendentes. Al

[5] "más cercano" teniendo en cuenta todos los *nodos* externos alcanzables, iniciando el descenso tanto por el hijo izquierdo como por el derecho.

igual que en *Skew Heap*, *inserción* y *remoción* se implementan mediante una operación auxiliar, llamada *merge*, que dados dos *Leftist Trees* devuelve un tercero formado por la unión de los *nodos* de los dos originales, destruyéndolos. Es *merge* entonces la operación que merece análisis, ya que una *inserción* consiste en ejecutar *merge* entre el *Leftist Tree* en su estado actual y otro formado sólo por el *nodo* a *insertar*, y una *remoción* en *merge* entre los dos *Leftist Trees* resultantes de extraer el *nodo* raíz. El algoritmo recursivo que permite ejecutar *merge* entre dos *Leftist Trees* LT_1 y LT_2 se ilustra en **FIGURA 2.12**. Consiste en:

1) Llamar LT_1 al Leftist Tree con valor más bajo de *t* en el *nodo* raíz y LT_2 al otro.

2) Desconectar el hijo derecho de LT_1 y hacer *merge* entre él y LT_2.

3) Convertir el *Leftist Tree* resultante del *merge* realizado en 2) en hijo derecho del LT_1 determinado en 1).

En el punto 3) se debe corregir, mediante el eventual intercambio entre hijos izquierdo y derecho, la condición de *Leftist Tree* sobre distancia al *nodo* externo más cercano. Esto marca la necesidad de que cada *nodo* aloje el valor de *d(x)*.

La idea de mantener "cortos" los caminos descendentes del costado derecho es precisamente aprovecharlos para hacer sobre ellos la operación *merge*, consiguiendo así realizarla con un costo ***O****(log n)*, el menor posible.

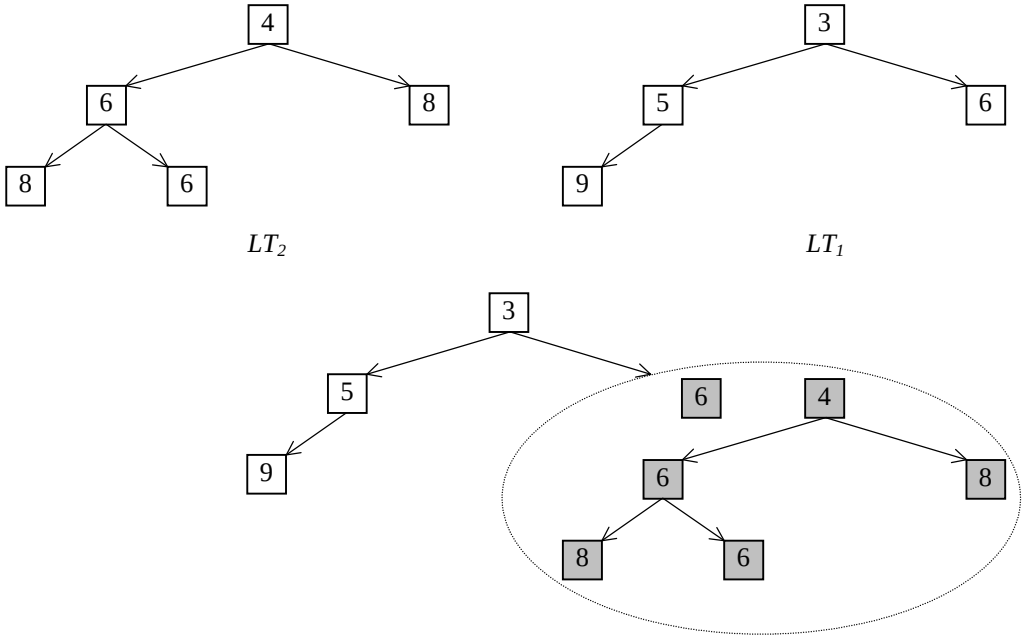


FIGURA 2.12 Merge de dos Leftist Trees

2.19. Pagoda [4]

Tal vez la forma más elemental de representar el *calendario* sea mediante una lista de *nodos*, mantenida por su orden de arribo. Las *inserciones* se reducen entonces a la operación *append* del *nodo* más recientemente arribado y tienen un costo ***O****(1)*, en tanto las *remociones*, asistidas por *búsqueda secuencial*, tienen un costo ***O****(n)*. En el análisis que sigue se va a considerar a cada *evento* representado, en la lista, sólo por su tiempo de planificación. Así una secuencia como $s=(t_1, t_2, t_3, \ldots t_n)$ sirve para analizar las operaciones básicas sobre un *calendario* cuyos *registros* exhiben los tiempos t_i, $i=1, 2, \ldots, n$. Si $t_{mín}$ es el mínimo valor de *t* en *s*, se puede decir que $s=g \cdot t_{mín} \cdot d$, donde $\cdot$ indica la concatenación entre dos secuencias y donde *g* y *d* son las secuencias que están respectivamente a izquierda y derecha de $t_{mín}$. A cada lista o secuencia, *s*, se puede asociar un árbol binario, $\delta(s)$, en el que $t_{mín}$ es la raíz y los subárboles $\delta(g)$ y $\delta(d)$ los hijos izquierdo y derecho respectivamente. (Por convención $\delta(\Lambda)=\phi$, esto es, a la secuencia vacía le corresponde el árbol vacío). La correspondencia es biunívoca, en el sentido que existe un árbol para cada secuencia y viceversa. **FIGURA 2.13** muestra $\delta(s)$ para $s=(5, 7, 3, 9, 6, 1, 4, 2, 8)$.

$\delta(s)$ verifica la propiedad de Heap para toda secuencia *s*. En principio una representación mediante árboles de tipo $\delta(s)$ permitiría implementar las operaciones básicas del *calendario*. Como en todo *Heap*, la *remoción* del *nodo* con $t_{mín}$ es simple, sólo que va seguida de la unión de lo dos árboles resultantes.

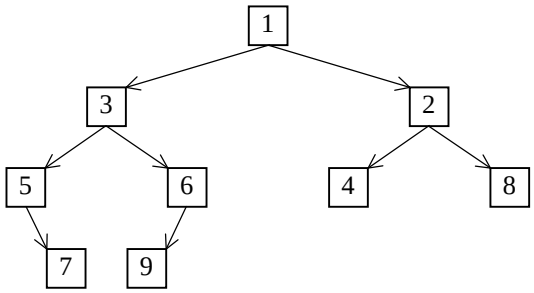


FIGURA 2.13 Árbol $\delta(s)$

También existe la posibilidad de implementar la unión de dos árboles de un modo sencillo, considerando la concatenación de sus secuencias asociadas. Así una posible unión entre $\delta(y)$ y $\delta(z)$ puede obtenerse como $\delta(y \cdot z)$. El hecho es que no resulta éste un método muy eficiente. Dada la importancia de la operación, es imprescindible encontrar una forma alternativa, más efectiva. Una forma posible es la que utiliza *Pagoda* para llevar a cabo las operaciones del *calendario*, tanto que incluso realiza la *inserción* mediante la unión del árbol existente con otro cuyo único *nodo* es el *evento* a *insertar*.

Se define la *rama derecha* de un árbol *T*, *rd(T)*, como la secuencia creciente de *nodos* que resulta de seguir el camino que se inicia en el raíz y desciende, mientras existe, hacia el hijo derecho. De un modo análogo se define la *rama izquierda*, *ri(T)*. El algoritmo que propone *Pagoda* considera la unión ordenada de las *ramas izquierda* y *derecha* de los árboles *y* y *z* a unir como base del proceso, esto es

$$(x_1, x_2, x_3, \ldots, x_k) = sorted[rd(y) \cup ri(z)]$$

y define la unión entre $\delta(y)$ y $\delta(z)$ según las siguientes reglas:

1) El *nodo* raíz es x_1.
2) Para todo *i* tal que $1 \le i < k$, si $x_i \in y$, su hijo derecho es x_{i+1} y su hijo izquierdo el subárbol que tiene por hijo izquierdo en $\delta(y)$.
3) Para todo *i* tal que $1 \le i < k$, si $x_i \in z$, su hijo izquierdo es x_{i+1} y su hijo derecho el subárbol que tiene por hijo derecho en $\delta(z)$.

Sean por ejemplo las secuencias $y=(5,7,3,9)$ y $z=(6,1,4,2,8)$. **FIGURA 2.14** muestra los árboles $\delta(y)$ y $\delta(z)$.

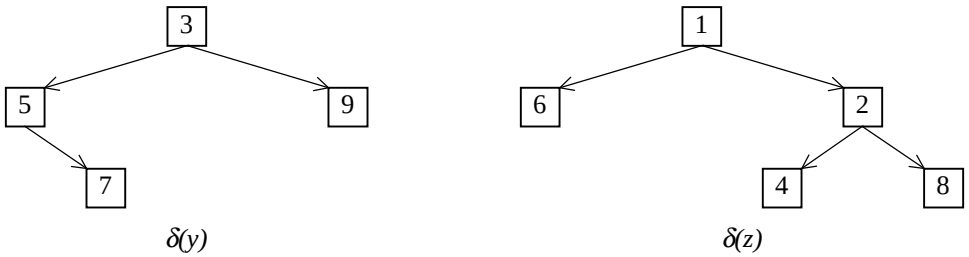


FIGURA 2.14 Árboles $\delta(y)$ y $\delta(z)$ a unir

$rd(y)=(3,9)$ y $ri(z)=(1,6)$, luego $sorted[rd(y) \cup ri(z)]=(1,3,6,9)$. Con lo que la unión entre $\delta(y)$ y $\delta(z)$ resulta el árbol de **FIGURA 2.15**.

Existen dos formas de realizar la unión o merge entre *rd* y *ri*, según se comience a recorrerlas desde abajo, o sea desde los *nodos* con valor de *t* más alto, o desde arriba, es decir desde los *nodos* con valor de *t* más bajo. Estas alternativas se conocen como *bottom-up* y *top-down* y siendo la primera de ambas la que implementa el algoritmo de merge de *Pagoda*.

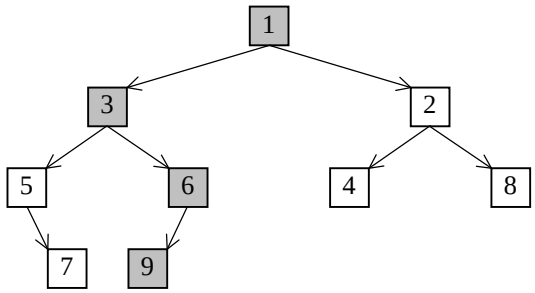


FIGURA 2.15 Unión de Árboles *δ(y) y δ(z)*

A fin de llevar a la práctica esta implementación se propone la inclusión, en cada *nodo*, de dos punteros *g* y *d* según las siguientes reglas:

1) Raíz: si *r* es el *nodo* raíz de *T*, *g(r)* apunta al *nodo* más alejado de la *ri* y *d(r)* al *nodo* más alejado de la *rd*.
2) Hijo Izquierdo: si *k* es un hijo izquierdo en *T*, *g(k)* apunta al padre de *k* y *d(k)* al *nodo* más alejado de la *rd* que comienza en *k*.
3) Hijo Derecho: si *k* es un hijo derecho en *T*, *d(k)* apunta al padre de *k* y *g(k)* al *nodo* más alejado de la *ri* que comienza en *k*.

Estos punteros (mostrados en **FIGURA 2.16**) enlazan todas las ramas en forma circular y permiten a los algoritmos realizar la unión de dos árboles mediante unión de sus *rd* y *ri* de acuerdo a lo expresado más arriba. Ningún proceso tendiente a mantener el balance de los árboles es realizado, razón por la cual configuraciones desbalanceadas pueden conducir a situaciones de peor caso ***O****(n)*. De todas maneras en promedio *inserciones* y *remociones*, apoyadas en este forma de unión o merge entre dos árboles, tienen un costo ***O****(log n)*.

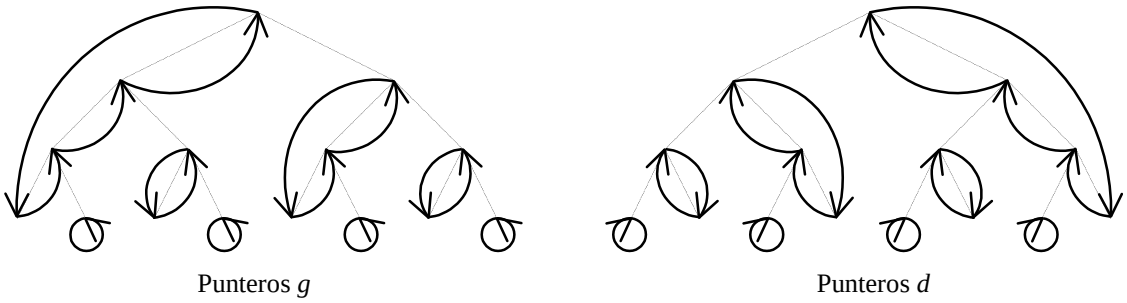


FIGURA 2.16 Distribución de Punteros en Pagoda

2.20. Priority Tree [31]

Priority Tree, también conocido como *p-tree*, es una variante de árbol binario de búsqueda que se reconoce, históricamente, como al antecesor del *Heap* (de hecho un *p-tree* verifica la condición de *Heap*). Se trata de una implementación en la que los *nodos* se alojan de modo que al atravesarlos según el recorrido *post-order*, se los visita según el orden creciente de sus valores de *t*. Eso hace que la estructura adquiera una configuración particular, cuya interpretación puede ayudar a describir los algoritmos de *inserción* y *remoción*.

El *nodo* con más alto valor de *t* es el raíz (el último en ser visitado en el recorrido *post-order*). Si desde él se comienza a descender siguiendo la cadena de enlaces que llevan sucesivamente al hijo izquierdo, y el proceso continúa hasta encontrar por primera vez un puntero nulo, se cubre un tramo denominado *camino izquierdo*. Dados dos *nodos* consecutivos del *camino izquierdo*, por ejemplo *x* e *y*, tales que *x* es el padre de *y*, con tiempos t_x y t_y respectivamente, todos los *nodos* del sub-árbol que se enlaza como hijo derecho de *x* tienen valores de *t* comprendidos entre t_x y t_y., y lo que es más importante, el sub-árbol enlazado como hijo derecho es también un *p-tree*. **FIGURA 2.17** ilustra esta propiedad.

Una versión recursiva para el algoritmo de *inserción* en un *p-tree T* es:

1) Si *T* es un árbol vacío o el valor de *t* de su *nodo* raíz es menor o igual que t_i, convertir el *nodo* a *insertar* en raíz de *T*, a *T* en su hijo izquierdo y terminar.

2) Descender por el *camino izquierdo* hasta encontrar un *nodo y* tal que t_y sea menor o igual que t_i,

   - Si *y* no existe, agregar el nuevo *nodo* como último del *camino izquierdo*, o sea como hoja izquierda y terminar.
   - Ir hasta el padre de *y*, llamar *T* a su hijo derecho y volver a 1).

Una forma rápida de implementar la *remoción* es manteniendo, bajo al forma de una variable global (*min*), un puntero al *nodo* con menor valor de *t*. Esa variable se debe actualizar, en las *inserciones* sólo en caso de arribo de un *nodo* con valor un de *t* inferior, y luego de las *remociones* con el fin de apuntar al *nodo* con menor valor de *t* entre todos los restantes. Esta opción se simplifica si cada *nodo* cuenta también con puntero a su padre. Entonces,

1) Una vez realizada la *remoción* (lo cual es inmediato dado que *min* es siempre una hoja y además se cuenta con puntero al padre) apuntar *min* a su padre (*p*).

2) Transformar lo que *p* tenga por hijo derecho en su hijo izquierdo.

3) Comenzar a descender desde *p* siguiendo la cadena de enlaces que llevan sucesivamente al hijo izquierdo. Continuar hasta encontrar por primera vez un puntero nulo. Apuntar *min* al *nodo* que tiene como hijo izquierdo a ese puntero nulo y terminar.

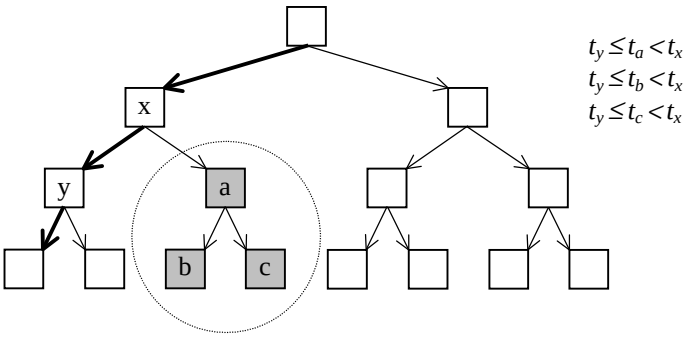


FIGURA 2.17 Priority Tree

2.21. Splay Tree [24]

Existe una gran variedad de árboles binarios de búsqueda, muchos de los cuales (por ejemplo los *balanceados en altura*, los *balanceados en peso*, los *B-trees*) exhiben un comportamiento de peor caso ***O****(log n)* para su velocidad de acceso. No obstante ello, no son estructuras de buen tiempo de respuesta y, más aún, *inserciones* y *remociones* son operaciones de un costo importante. Además, casi todas esas implementaciones necesitan almacenar en sus *nodos* información relativa al estado de balanceo, lo cual implica un perjuicio al rendimiento.

Todas estas estructuras ponen énfasis en la reducción del tiempo de respuesta de peor caso, lo cual puede resultar de interés por ejemplo en sistemas de tiempo real, pero si la aplicación sólo exige la ejecución de una gran número de operaciones en un bajo tiempo, como puede ser el caso de la SED, entonces el parámetro a minimizar es el tiempo amortizado, entendiendo por tal a la duración media de cada operación de una serie ejecutada como una secuencia de peor caso. O sea no se analiza la duración de la operación que más tiempo demanda, sino el promedio de tiempo de las operaciones correspondientes a una secuencia tal que en conjunto representan la sucesión más exigente de todas las posibles.

Una alternativa para reducir el tiempo amortizado son las estructuras *autoajustables*, entendiendo por tales a aquellas que estando en un estado determinado realizan, ante la ejecución de cualquier operación sobre ellas, tareas de reestructuración tendientes a favorecer la ejecución de operaciones futuras. Este tipo de estructura prescinde de toda información relativa al balanceo, es más adaptable a distintas distribuciones temporales y tiene por lo general algoritmos más sencillos que las estructuras de balanceo estricto.

Splay Tree es un tipo de árbol binario de búsqueda *autoajustable*. Por lo tanto,

1) En cuanto árbol binario de búsqueda, su configuración se ajusta al siguiente principio: para todo *nodo* con valor de tiempo *t*, el valor de tiempo de su hijo izquierdo es tal que $t_{izq} < t$, y el valor de tiempo de su hijo derecho es tal que $t_{der} \geq t$.

2) En cuanto estructura *autoajustable*, su heurística de reestructuración consiste en una operación (*splaying*) encargada de convertir el *nodo* accedido en el *nodo* raíz del árbol, cualquiera sea la operación realizada sobre el mismo.

La operación de *splaying* consiste en una serie de rotaciones realizadas a lo largo del camino que lleva del *nodo* accedido al *nodo* raíz. Las rotaciones son tomadas, en todos los casos, de un conjunto de seis posibles, dependiendo la elección de la posición del *nodo* accedido dentro del árbol, en cada momento. El análisis de las rotaciones presentado en las **FIGURAS 2.18**, **2.19** y **2.20** se ajusta a las siguientes convenciones:

1) Las celdas cuadradas, identificadas mediante letras minúsculas, representan *nodos*.

2) Los puntos denotados con letras mayúsculas representan sub-árboles.

3) El *nodo* accedido es, en todos los casos, *x*.

4) El *nodo* cuya posición ha de ocupar *x* luego de la rotación (el *nodo* raíz del árbol sólo luego de la última rotación) es, en todos los casos, *z*.

5) Los movimientos de *x* hacia *z* son siempre ascendentes. Una forma de identificar la posibles ubicaciones relativas entre ambos *nodos* es siguiendo el recorrido que lleva desde *z* hacia *x*. Todo descenso hacia la izquierda (por el enlace que conduce al hijo izquierdo) se denomina *zig* y todo descenso hacia la derecha *zag*.

El punto 5) establece entonces las bases para la clasificación de las rotaciones, las cuales (sólo una de las dos opciones simétricas en cada Caso) se presentan seguidamente.

Caso 1 (*zig*): Constituye la más elemental de las rotaciones y es sólo aplicable como última de la serie, es decir cuando *z* es el *nodo* raíz del árbol.

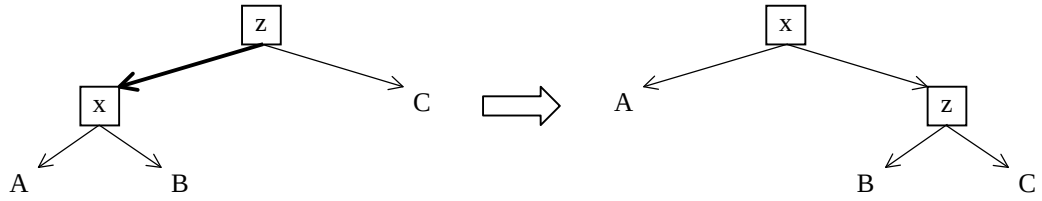


FIGURA 2.18 Splay Tree, caso *zig*

Caso 2 (*zig-zig*):

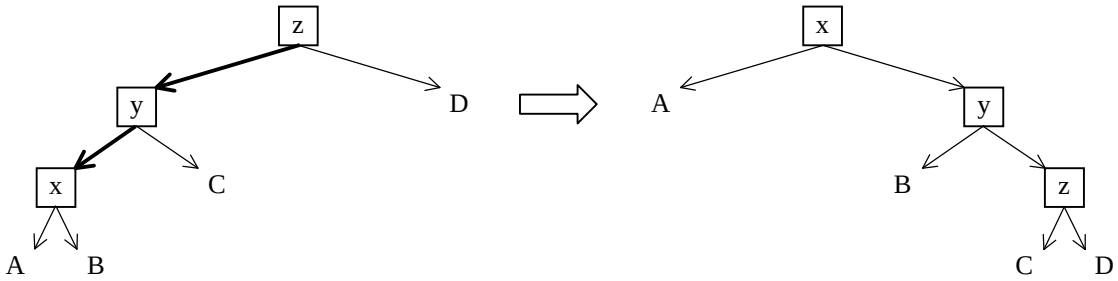


FIGURA 2.19 Splay Tree, caso *zig-zig*

No debe confundirse *splaying* con la aplicación sucesiva de rotaciones *zig* y *zag* (respecto al padre) hasta convertir el *nodo* accedido en raíz del árbol. Esa es otra heurística, conocida como Movimiento Hacia la Raíz (*Move to Root*), de hecho menos eficiente y que conduce a distintas configuraciones que *splaying*.

Case 3 (*zig-zag*):

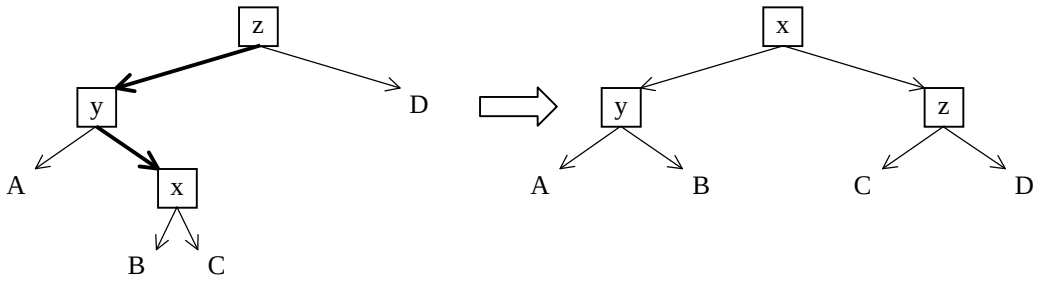


FIGURA 2.20 Splay Tree, caso *zig-zag*

En *Splay Trees* toda vez que un *nodo* es accedido, inmediatamente luego de la operación de acceso se ejecuta *splaying* sobre él. Se prevé incluso que si se solicita el acceso a un *nodo* que no existe en el árbol, se hace *splaying* sobre el inmediato anterior en el recorrido, es decir aquel que tiene un valor "nulo" en el puntero al hijo en el que debería estar el *nodo* buscado. Se demuestra tanto teórica como empíricamente que este mecanismo no sólo mantiene cerca del raíz a los *nodos* frecuentemente accedidos, sino que fuerza una tendencia a mantener el árbol balanceado y consecuentemente mejora el rendimiento de operaciones asociadas a búsquedas.

Resta analizar la forma de ejecutar en *Splay Trees* las operaciones propias de la planificación de *eventos*. Para ello resultan útiles las siguientes operaciones adicionales:

1) *join(A_1,A_2)*: Siendo A_1 y A_2 dos árboles binarios tales que el valor de *t* de todos los *nodos* de A_1 es menor que el *t* de cada uno de los elementos de A_2, *join(A_1,A_2)* devuelve un puntero al *nodo* raíz de un único árbol que contiene todos los *nodos* de A_1 y de A_2.

2) *split(t,A)*: construye y devuelve dos árboles binarios tales que uno de ellos se compone de todos los *nodos* de *A* cuyo valor de tiempo en menor o igual que *t* y el otro de *nodos* de *A* con tiempos mayores que *t*.

La implementación de *join(A_1,A_2)* requiere primeramente ubicar el *nodo* con el valor más alto de *t*. Luego de accederlo quedará convertido en raíz de su árbol. Por no haber *nodos* con valor de *t* superior carecerá de hijo derecho. Se coloca entonces el otro árbol en esa posición.

split(t,A) sólo requiere un acceso al *nodo* cuyo valor de tiempo es *t*. Luego se rompe uno de los dos enlaces (izquierdo o derecho) según sea *t* del *nodo* raíz mayor que el argumento de *split(t,A)* o no.

Ahora una *inserción* sobre el árbol *A* procede de la siguiente manera: se ejecuta *split(t_i,A)* y se arma un nuevo árbol conteniendo el *nodo* a *insertar* como raíz y los dos árboles devueltos por *split(t,A)* como hijos izquierdo y derecho respectivamente. En tanto una *remoción* se ejecuta ubicando y accediendo el *nodo* con más bajo valor de *t*, desconectando y devolviendo ese *nodo* y haciendo finalmente *join* de los dos árboles resultantes.

Existen, no obstante, formas alternativas de implementación de estas operaciones. Para las *inserciones*, simplemente llevando a cabo la *inserción* tradicional sobre un árbol binario de búsqueda y luego haciendo *splaying* sobre ese *nodo*. La *remociones* pueden resolverse del siguiente modo: dado que el *nodo* a *remover* contiene el menor valor de *t* del árbol, no puede tener hijo izquierdo. Se desconecta entonces el *nodo* a *remover*, se le desconecta el subárbol enlazado como su hijo derecho, se lo coloca en reemplazo del *nodo* removido y se hace *splaying* sobre el que era su padre.

2.22. Fibonacci Heap [7]

Los *nodos* se alojan en un bosque de árboles, no binarios, cada uno de los cuales verifica la propiedad de *Heap*. Tanto las *inserciones* como las *remociones* se ejecutan con apoyo de una tercer operación llamada *enlazado*, cuya ejecución consiste en lo siguiente: dados dos árboles se compara el valor de *t* de sus *nodos* raíz y se convierte al de mayor valor en hijo del *nodo* raíz del otro. No pesa ninguna restricción sobre el número de árboles ni sobre la configuración de los mismos. Para todo *nodo x* ubicado en cualquier lugar de cualesquiera de los árboles se define el *grado(x)* como el número de hijos de *x*[6].

[6] Se utiliza también un bit de *marcado* en cada *nodo*, pero su utilidad está reservada a operaciones que no se comentan en el presente análisis.

En materia de implementación los requisitos son que cada *nodo* cuente con:

1) Un puntero al padre (con valor nulo en caso de ser raíz de algún árbol).
2) Un puntero a uno de sus hijos.
3) Un campo indicador de su *grado*.

Además,

1) Todos los hijos de un determinado *nodo* están ligados formando una lista circular doblemente enlazada.
2) Los *nodos* raíz de todos los árboles están ligados formando una lista circular doblemente enlazada.

Se llama *nodo mínimo* a aquel que ostenta el menor valor de *t* en la lista de *nodos* raíz, siendo ese un punto siempre conocido de la estructura. **FIGURA 2.21** muestra un bosquejo de una posible configuración de *Fibonacci Heap*.

Las *inserciones* se resuelven creando un árbol cuyo único *nodo* sea el *nodo* a *insertar* y fusionándolo con el árbol actual, es decir el que contiene la totalidad de los *nodos*. En general fusionar dos *Fibonacci Heaps* implica combinar sus listas de *nodos* raíz formando una única, cuidando de elegir como *nodo mínimo* de la fusión al *nodo* de menor valor de *t* entre los dos *nodos mínimos* de los *Fibonacci Heaps* fusionados. Se trata de una operación $\boldsymbol{O(1)}$.

Las *remociones* son más complicadas, de hecho, aunque se ejecutan en tiempo amortizado $\boldsymbol{O(\log_2 n)}$, son las operaciones más lentas de *Fibonacci Heap*. Su ejecución consiste en *remover* el *nodo mínimo*, luego concatenar la lista de sus hijos con la de los *nodos* raíz restantes y ejecutar, finalmente, el siguiente proceso iterativo: *enlazar* todo par de árboles con *nodos* raíz del mismo *grado* hasta que ya no sea posible continuar (cada enlazado produce un árbol cuyo *nodo* raíz tiene *grado* uno mayor que los *nodos* raíz de los árboles *enlazados*). Luego se conforma una lista con los *nodos* raíz resultantes, buscando simultáneamente el *nodo* con menor valor de *t* para designarlo como nuevo *nodo mínimo*.

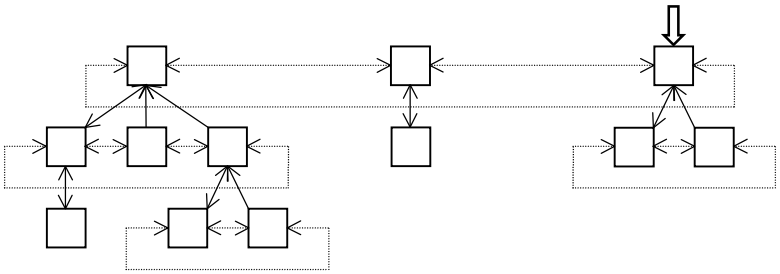


FIGURA 2.21 Fibonacci Heap

En la práctica para implementar el proceso iterativo de *enlazados* se utiliza un arreglo cada uno de cuyos elementos es un puntero a *nodo*. Eso establece una relación entre los valores del índice del arreglo y los valores de *grado*, porque a medida que se va barriendo la lista se van apuntando los punteros del arreglo con el *siguiente* criterio: si el *nodo* raíz tiene *grado k* se le apunta el *k*-ésimo puntero del arreglo; si en algún momento aparece un segundo *nodo* de grado *k*, dado que la posición *k* del arreglo estará ocupada se realiza el *enlazado* entre ambos *nodos* y se intenta colocar el árbol resultante en la posición *(k+1)*-ésima. El proceso finaliza cuando ya no queda ningún *nodo* raíz sin ser apuntado por algún puntero del arreglo, luego de lo cual se los pasa a una nueva lista y se determina el *nodo mínimo*. El tiempo total de la *remoción* resultará proporcional al máximo *grado* encontrado y al número de *enlazados* realizados.

Se debe tener en cuenta que si se inicia un *Fibonacci Heap* vacío, es decir sin *nodos*, y se lo hace crecer mediante una secuencia de *inserciones* y *remociones*, los árboles resultantes serán todos *árboles binomiales*, los que, en este caso, puede definirse como:

1) Un *árbol binomial* de *grado 0* es un árbol de un solo *nodo*.
2) Un árbol binomial de grado *k* se forma mediante el enlazado de dos árboles binomiales de grado *k-1*.

Un *árbol binomial* de *grado k* contiene exactamente 2^k *nodos* y su *nodo* raíz *k* hijos, por lo que en un *Fibonacci Heap* de *n nodos*, el *grado* de cada uno será, a lo sumo, $\log_2 n$.

Fibonacci Heap fue concebida para otro tipo de problema, distinto de la planificación de *eventos* en la SED dado que exhibe buen rendimiento no sólo para *inserciones* y *remociones* sino también en otro tipo de operaciones de las *colas de prioridad*. Su dificultad de implementación hace, por lo tanto, que no sea una opción frecuentemente considerada si las operaciones a ejecutar son casi exclusivamente *inserción* y *remoción*.

2.23. Pairing Heap [26]

Desde cierto punto de vista *Pairing Heap* es considerada una estructura *autoajustable* (versión *autoajustable* de *Fibonacci Heap*) ya que sus operaciones son independientes de todo valor de las estructuras intervinientes (su dimensión por ejemplo) y además ante la ejecución de determinadas operaciones se realizan, en forma complementaria y automática, procesos que favorecen la ejecución de operaciones futuras. Su estructura base es un único árbol, no binario, que verifica la propiedad de *Heap*. Al igual que en *Fibonacci Heaps*, las operaciones fundamentales del *calendario* se realizan con el apoyo de una sencilla operación auxiliar de *enlazado*, consistente en lo siguiente. Dados dos *nodos*, con o sin descendientes, su *enlazado* convierte al de mayor valor de *t* en el primer hijo izquierdo del otro. El resultado es un *Heap* conteniendo la totalidad de los *nodos* involucrados (los dos cuyo valor de *t* fue primeramente comparado y, eventualmente, todos sus descendientes).

Una *inserción* se resuelve entonces simple y rápidamente mediante el *enlazado* entre el *nodo* a *insertar* y el *nodo* raíz del árbol que aloja al totalidad de los *nodos* en ese momento. Resulta así una operación ***O(1)***.

Las *remociones* son un poco más complicadas. Obviamente el *nodo* a *remover* es el raíz del árbol que aloja todos los *nodos*. Su extracción generará entonces tantos árboles como hijos haya tenido el *nodo* a *remover*, previo a la *remoción*. El conjunto de árboles resultante, sea éste $\{x_1,x_2,...,x_k\}$, es barrido de izquierda a derecha y sus elementos *enlazados* de a dos (de a pares, de ahí el nombre *Pairing*), es decir x_1 con x_2, x_3 con x_4, ... Esta primer pasada de *enlazados* da origen a un nuevo conjunto de *nodos* raíz, $\{y_1,y_2,...,y_h\}$ donde $h=[k/2]$. Si k es par, los h árboles generados serán la resultante de un *enlazado*, pero si k es impar sólo se enlaza hasta x_{k-1} y se designa a x_k como y_h. Luego se inicia una segunda pasada de *enlazados*, ahora de derecha a izquierda pero con carácter acumulativo, es decir se *enlaza* y_h con y_{h-1} generándose z_1, luego z_1 con y_{h-2} y así sucesivamente, hasta que el último *enlazado* de $z_{h/2}$ con y_1 da por resultado un único árbol. Esta variante se conoce como *dos-pasadas*, pero es posible también resolver la *remoción* mediante un mecanismo conocido como *multi-pasada*, en el cual la primer etapa es idéntica a la primera del caso *dos-pasadas* e idéntica a todas las subsiguientes, es decir no hay proceso de *enlazados* acumulativos sino que todas las pasadas van tomando pares de *nodos* nuevos cada vez. Como resultado, cada pasada va encontrando el número do *nodos* a *enlazar* dividido en dos, de manera que si el *nodo* raíz del árbol original tenía *r* hijos, el proceso finaliza luego de $log_2\ r$ pasadas. A pesar de que cualquiera de las dos variantes realiza, en definitiva, la misma cantidad de *enlazados*, dada la configuración resultante luego de una *remoción*, la opción *dos-pasadas* ofrece mejor rendimiento.

Las peores secuencias a que se puede someter la estructura *Pairing Heap* son aquellas compuestas exclusivamente de *inserciones* en las que

1) el valor de *t* del *nodo* a *insertar* es siempre mayor que el *t* del *nodo* raíz o

2) el valor de *t* del *nodo* a *insertar* es siempre menor que el *t* del *nodo* raíz.

La opción 1) produce un árbol formado por un *nodo* raíz con una "enorme" cantidad de hijos, mientras que la opción 2) conduce a un árbol con gran cantidad de *nodos* encadenados formando un "muy largo" *camino izquierdo*. En cualquier caso la operación *remoción* se ejecuta en tiempo amortizado $\boldsymbol{O(log_2\ n)}$.

2.24. Resumen de características

La **TABLA 2.1** resume las principales características de las implementaciones citadas en la presente Sección [8, 20]. El detalle de la información consignada en cada columna es el siguiente:

TIPO: L indica que la estructura está basada en una o más Listas, en tanto A indica lo propio respecto a Arboles.

ORD: C indica que el ordenamiento se realiza en forma Continua, es decir, que ante cada *inserción* o *remoción* se procura que las posiciones ocupadas dentro de la estructura guarden una relación de orden tal que nunca sea necesario apelar a algoritmos de ordenamiento. B indica que en un determinado momento los *registros* se ordenan por lotes (*en batch*) mediante la aplicación de un algoritmo específico.

ESTR: S indica que es preciso montar parte importante de la estructura antes de iniciar la simulación. En general esto corresponde a la declaración de arreglos y el establecimiento de valores umbrales. N indica que mientras la estructura no contiene *registros* no existe como tal.

PAR: S indica que el mecanismo necesita que se le suministre el valor de un determinado número de parámetros para operar, en tanto N corresponde implementaciones que no tienen parámetros de operación.

PROG: corresponde a una indicación, estimativa, de la dificultad que implica la tarea de programar la implementación. En tal sentido S, I y D significan Simple, Intermedia y Difícil.

INS: es el costo medio de la *inserción*.

REM: es el costo medio de la *remoción*.

IMPLEMENTACION	*TIPO*	*ORD*	*ESTR*	*PAR*	*PROG*	*INS*	*REM*
Linear	L	C	N	N	S	$\boldsymbol{O}(n)$	$\boldsymbol{O}(1)$
Short-Cut List	L	C	S	S/N [a]	S	- [b]	$\boldsymbol{O}(1)$
Henriksen's	L	C	S	S	S	$\boldsymbol{O}(n^{0.5})$	$\boldsymbol{O}(1)$
Skip List	L	C	N	S	I	$\boldsymbol{O}(\log n)$	$\boldsymbol{O}(1)$
Indexed List	L	C	- [a]	S	- [a]	- [a]	$\boldsymbol{O}(1)$
Two Level Structure	L	C	S	S	I	- [c]	$\boldsymbol{O}()$
Two List	L	B	N	N	I	$\boldsymbol{O}(n^{0.5})$	$\boldsymbol{O}(1)$
Linked Exact Bottom Up - BSS [d]	L	C	N	S	I	$\boldsymbol{O}(\log n)$	$\boldsymbol{O}(1)$
Calendar Queue	L	C	S	S	D	$\boldsymbol{O}(1)$	$\boldsymbol{O}(1)$
Lazy Queue	L	C/B [e]	S	S	D	$\boldsymbol{O}(1)$	$\boldsymbol{O}(1)$
SPEEDES-Q	L	B	N	S	I	$\boldsymbol{O}(1)$	$\boldsymbol{O}(n \log n)$
Cascade	L	C	S	S	I	$\boldsymbol{O}(\log n)$	$\boldsymbol{O}(\log n)$
Implicit Heap	A	C	N	N	S	$\boldsymbol{O}(\log n)$	$\boldsymbol{O}(\log n)$
Skew Heap (Top Down)	A	C	N	N	S	$\boldsymbol{O}(\log n)$	$\boldsymbol{O}(\log n)$
Binomial Queue	A	C	N	N	D	$\boldsymbol{O}(\log n)$	$\boldsymbol{O}(\log n)$
Leftist Tree	A	C	N	N	I	$\boldsymbol{O}(\log n)$	$\boldsymbol{O}(\log n)$
Pagoda	A	C	N	N	D	$\boldsymbol{O}(\log n)$	$\boldsymbol{O}(\log n)$
Priority Tree	A	C	N	N	I	$\boldsymbol{O}((\log n)^2)$	$\boldsymbol{O}((\log n)^2)$
Splay Tree	A	C	N	N	D	$\boldsymbol{O}(\log n)$	$\boldsymbol{O}(\log n)$
Fibonacci Heap	A	C	N	N	D	$\boldsymbol{O}(1)$	$\boldsymbol{O}(\log n)$
Pairing Heap	A	C	N	N	S	$\boldsymbol{O}(1)$	$\boldsymbol{O}(\log n)$

a Depende de cual de las variantes de trate.
b No está determinada.
c No encontrado en la literatura. Originalmente sus creadores [5, 6] le atribuyeron ***O(1)***, pero estudios posteriores [2, 14] demostraron que se trataba de un error.
d Una de las Implementaciones BSS (Ver Sección **3**).
e Recurre a ambos mecanismos.

3. *Bounded Sequential Searching* - BSS [16]

En esta Sección se aborda una de las propuestas de esta Tesis para el *calendario* de la SED. No se trata de una implementación concreta sino de un mecanismo que sirve de base a siete implementaciones cada una de las cuales lo lleva a la práctica con ligeras variantes. La estructura que da sustento a la implementación es la lista y el objetivo es, mediante la aplicación del mecanismo BSS, evitar recorridos secuenciales largos, acotando así el costo de la búsqueda en la lista y consecuentemente el costo de las operaciones sobre el *calendario*. De las siete propuestas de implementación presentadas, una de ellas se describe en forma detallada. Al final se hace un análisis comparativo experimental entre todas las implementaciones del mecanismo BSS y se comentan algunas conclusiones.

3.1. Introducción

Se dijo que una lista ordenada de *nodos* simplemente enlazados se desempeña suficientemente bien como *cola de prioridad* a pesar de su simplicidad de implementación (en todo caso "gracias" a esa simplicidad). En tanto el número de *eventos* no exceda unas pocas decenas, si las *remociones* ocurren por uno de los extremos y las *inserciones* son asistidas por *búsqueda secuencial*, esta variante es aceptada como la más rápida. Pero cuando el tamaño de la lista crece, lo propio sucede con los tiempos de búsqueda, y una vez que el número de *eventos* asciende a unas centenas este sencillo mecanismo se torna "pesado" frente a algoritmos más sofisticados, su simplicidad de implementación pierde importancia y soluciones algorítmicamente más ingeniosas resultan más efectivas, a pesar de las complicaciones de su implementación.

No obstante, si las búsquedas son aceleradas mediante algún artificio o mecanismo complementario, los *eventos* pueden ser administrados eficientemente sobre una lista simplemente enlazada. Una de las soluciones para lograr esta mejora es la Búsqueda Binaria o Dicotómica [11, 27]. De hecho su aplicación no es inmediata en listas simplemente enlazadas de las que sólo se conoce su tamaño y a lo sumo las posiciones extremas. En este caso el único modo de desplazarse desde un *nodo* hacia otro es atravesando todos los intermedios, y esta pesada tarea convierte el método en impracticable. La clave para tornarlo factible es encontrar la forma de localizar *nodos* dentro de la estructura, no en forma absoluta como elementos de un arreglo, sino al menos en forma relativa a alguna posición conocida como por ejemplo la cabecera de la lista. El conjunto de posiciones que es necesario conocer sigue una suerte de patrón iterativo tal como: la posición central, el centro de cada uno de los segmentos determinados por la posición central, y así sucesivamente. En una implementación común (mediante lenguajes como C o Pascal) no es complicado mantener un puntero al centro geométrico de una lista. Ese puntero deberá actualizarse luego de *inserciones* y *remociones*, siendo esta una tarea bastante simple. Un tanto más complicado es mantener, además, un puntero al centro de cada uno de los segmentos determinados en la lista por el puntero central. Pero el intento de multiplicar el número de punteros para particionar la lista en 2^i segmentos, siendo i un entero, merece dos observaciones. La primera es que la complejidad del algoritmo para manejar los punteros crece dramáticamente con su número, y la segunda es que todavía estamos hablando de un número fijo de punteros, razón por al cual la utilidad que los mismos han de prestar está inversamente relacionada al tamaño de la lista, tendiendo a extinguirse a medida que ésta crece. El desafío para llevar a la práctica la Búsqueda Binaria en listas ordenadas, simplemente enlazadas, pasa entonces por hallar la forma de moverse a través de sus elementos sin necesidad de atravesar todos los que conforman la cadena que liga origen y destino.

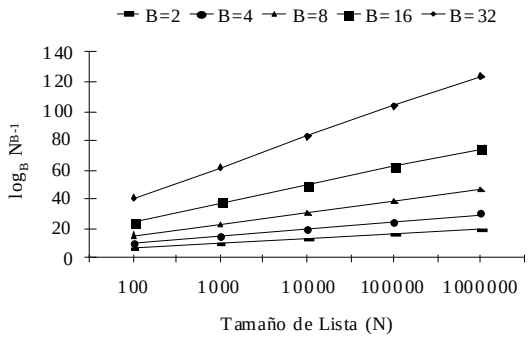


FIGURA 3.1 Costo de Búsqueda

Por otro lado, la Búsqueda Binaria es un caso particular de un método más general en el cual la lista es primeramente dividida en varios segmentos del mismo tamaño (dos o más). Luego, el segmento al que pertenece el punto buscado es dividido nuevamente en tantos segmentos iguales como los generados en la primera partición. Procediendo de este modo el éxito de la búsqueda está garantizado luego de chequear no más de $log_B N^{B-1}$ elementos ($log_2 N$ para la Búsqueda Binaria) siendo N el tamaño de la lista y B el número de segmentos en los que la lista o cualquiera de las subdivisiones es particionada cada vez.

Para demostrar este resultado supóngase que la lista aloja *N registros*, y que está ordenada según el valor de *t* de los mismos. Supóngase entonces que se divide la lista en *B* segmentos de igual tamaño, conteniendo cada uno de ellos *N/B* elementos. La determinación del segmento dentro del cual se halla un punto buscado (sea éste un *registro* existente o un lugar en el que agregar un nuevo elemento) puede resolverse chequeando el valor de *t* del *registro* que ostenta el valor más alto (el más hacia la derecha para la representación lineal) de los primeros *B-1* segmentos. Si el segmento al que pertenece el punto buscado es dividido nuevamente en *B* segmentos de igual tamaño, conteniendo cada uno de ellos N/B^2 elementos, nuevamente *B-1* segmentos deberán chequearse para decidir en cual de ellos se encuentra el punto buscado. Luego de *K* iteraciones del método, el tamaño de los *B* segmentos resultantes podría ser *1*, indicando esto que alguna de las últimas *B-1* comparaciones hubo de coincidir con el punto buscado, y en ese momento el proceso termina. Dado que el tamaño de los últimos segmentos generados es $1=N/B^K$, entonces $K=log_B N$. Y dado que en cada una de las *K* iteraciones fueron necesarias *B-1* comparaciones, el total de comparaciones realizadas será: $K(B-1) = log_B N^{B-1}$.

La **FIGURA 3.1** presenta el costo de búsqueda para este mecanismo y muestra que, algorítmicamente, la opción más rápida se corresponde con el valor *B=2*.

Las propuestas que se abordan seguidamente apuntan a lograr un buen nivel de eficiencia en implementaciones basadas en lista, para colas de gran tamaño. En cierto sentido las ideas se sustentan en este último método de particiones sucesivas, pero los mecanismos para llevar a cabo las divisiones se ajustan a patrones diferentes. En todos los casos una estructura *auxiliar* es creada y enlazada por sobre la lista, y es a partir de esa estructura *auxiliar* que se consigue la posibilidad de desplazarse entre *nodos* no consecutivos, cercando así el punto buscado.

3.2. Búsqueda Secuencial Acotada (*Bounded Sequential Searching* - BSS)

Sea una implementación de *calendario* en la cual los *eventos* son representados por *nodos* comunes, de los que sólo se exige capacidad para establecer un enlace a otro *nodo* hacia el cual es posible "ir" y además estar enlazado con otro *nodo* desde el cual es posible "venir". Si los *nodos* arribados son encadenados mediante el uso de estos enlaces, cuidando de mantenerlos ordenados por su tiempo de ocurrencia, la estructura resultante es una *cola de prioridad* sobre lista simplemente enlazada, ordenada, en la que el *nodo* asociado al *evento* más pronto a ocurrir (es decir el más prioritario) está ubicado en uno de los extremos, resultando por lo tanto muy sencillo de *remover*. El trabajo más pesado debe realizarse para planificar nuevos *eventos*, o sea para *insertar* nuevos *nodos*. Al momento de contar el *calendario* con *N nodos*, el éxito de la búsqueda del punto de *inserción* únicamente puede garantizarse luego de chequearlos todos. Se propone la existencia de un límite para ese proceso estableciendo que **"ninguna búsqueda debería atravesar más de *B* nodos consecutivos"** siendo *B* un valor fijo. Entonces ante la solicitud de una nueva *inserción* luego de que la lista ha alcanzado el tamaño *B*, se agrega una sub-estructura de soporte para evitar el recorrido de los *B+1 nodos* resultantes en futuras *inserciones*. La construcción tal sub-estructura comienza con la *inserción* de un *nodo* en un cola *auxiliar* paralela, enlazada por sobre la lista de *eventos*, tal como se muestra en **FIGURA 3.2**.

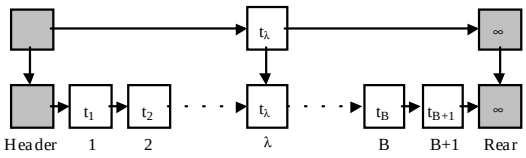


FIGURA 3.2 Inserción del 1er Nodo Auxiliar

Además de estar adecuadamente insertado en la cola a la que pertenece, el *nodo auxiliar* apunta a un *nodo* alojado en alguna posición λ de la lista de *eventos*, tal que $1 < \lambda \leq (B+1)$, y copia sobre sí el valor de tiempo del *nodo* apuntado, t_λ. De ahí en más el punto de ingreso a la estructura para iniciar búsquedas ya no es el *Header* de la lista de *eventos* sino el *Header* de la cola *auxiliar* que se acaba de agregar. La próxima *inserción* encuentra el punto para colocar el nuevo *nodo* buscando primeramente el *nodo* inmediato anterior al punto en el que

correspondería la *inserción* en la cola *auxiliar* recientemente creada. Para ello procede a comparar el valor de *t* del nuevo *nodo* contra t_λ, y eventualmente también contra el valor de *t* del *Rear* de la cola *auxiliar*, que es ∞. Luego la búsqueda continúa en la lista de *eventos*, ya sea desde el *Header* o desde el *nodo* apuntado por el *nodo auxiliar*, de acuerdo con las comparaciones anteriores. Si ninguna *remoción* tiene lugar, el proceso de *inserciones* puede continuar, a través del mismo mecanismo, hasta hacer que uno de los segmentos determinados por el *nodo auxiliar* alcance el tamaño (span) *B+1*. Entonces un nuevo *nodo* debe *insertarse* en el lugar apropiado de la cola *auxiliar*, siguiendo el mismo mecanismo. De ahí en más las búsquedas continúan iniciándose en el *Header* de la cola *auxiliar* pero ahora hasta tres comparaciones pueden ser necesarias para decidir el *nodo* por el cual continuar en la lista de *eventos*. Si la cola *auxiliar* sigue creciendo podría alcanzar un tamaño de *B+1 nodos*. Si esto sucede una nueva cola *auxiliar* se enlaza sobre la existente y consecuentemente el punto de inicio de búsquedas se lleva al *Header* de la última cola agregada. La estructura *auxiliar* resulta ser entonces un conjunto de colas paralelas, cada una en un nivel diferente, tal que todos los *nodos* de una determinada cola (incluyendo *Header* y *Rear*) apuntan a un *nodo* de la cola del nivel inmediato inferior, pero no a la inversa. A partir de la observación del tipo de estructura que se está construyendo (ver **FIGURA 3.3**) la proposición anterior pude reescribirse como **"ninguna búsqueda deberá atravesar más de *B* nodos consecutivos en ninguna de las colas"**. Por lo tanto el costo de la *búsqueda secuencial* estará acotado por *B+1* en cada nivel de la estructura resultante ya que de hecho *B+1* es el máximo número de *nodos* a chequear en cada uno, para toda búsqueda. De ahí el nombre de "Búsqueda Secuencial Acotada" (*Bounded Sequential Searching* - BSS) dado al mecanismo a través del cual se realizan las búsquedas, las *inserciones* y eventualmente las modificaciones de la estructura resultante. Cabe acotar que el modo en que la lista o cualquiera de los segmentos de la misma es subdividido obliga a *k+1* comparaciones toda vez que un segmento de tamaño (span) *k* es atravesado.

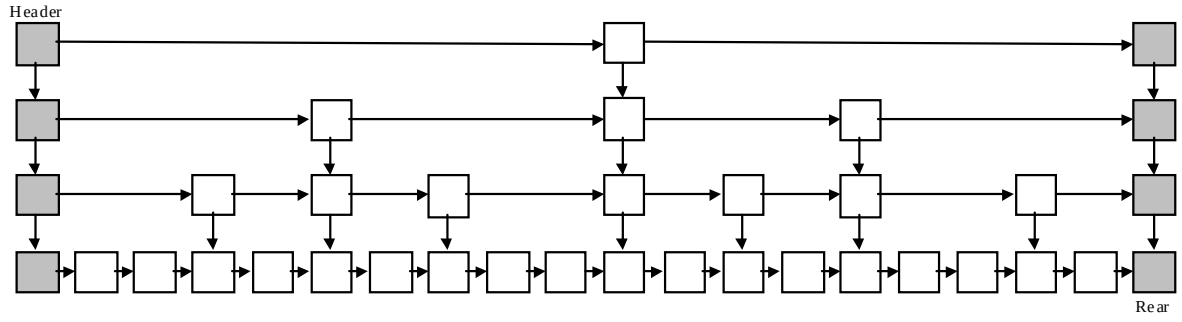


FIGURA 3.3 Una estructura resultante de la aplicación de BSS (B=2)

Aún sin mediar una equivalencia explícita entre *nodos* y enlaces, las estructuras resultantes de la aplicación del mecanismo BSS sugieren la existencia de un Árbol m-ario o Multivías subyacente [10, 27]. El número de *nodos* en BSS es más alto y algunos de los enlaces del árbol se corresponden con caminos de BSS. No obstante ambas estructuras distribuyen los *nodos* de acuerdo a un esquema similar y el mecanismo de búsqueda es parecido. La estructura resultante también guarda estrecha similitud con algunas de las formas del Modelo Determinístico de Skip List - DSL [17] sólo que de momento no se plantea en el mecanismo BSS una cota inferior para el tamaño de los segmentos (span), no se propone todavía ningún criterio para la determinación de λ ni se establecen los pasos a seguir para mantener la configuración ante *remociones* o borrado de *nodos* intermedios.

Llamaremos entonces BSS al mecanismo de búsqueda-inserción (y en cierto sentido de construcción), de cuya aplicación resulta el tipo de estructura presentada en esta Sección. Para que la misma pueda desempeñarse eficientemente como *calendario* en la SED es de esperar también que el costo de la *remoción* sea bajo (al menos que resulte bajo el costo conjunto *inserción + remoción*). Salvo mención en contrario y más allá de consideraciones particulares de cada variante, se aceptará que para *remover* basta con separar de la estructura el *nodo* con el valor más bajo de tiempo, eventualmente junto con toda la columna de *nodos auxiliares* que tuviese enlazada sobre sí. Si el valor de *B* es pequeño frente al número total de *nodos*, el sector que se ve afectado en su configuración como producto de este tipo de *remoción* no ejerce una influencia importante en futuras operaciones sobre la estructura resultante.

Se propone seguidamente una variedad de alternativas para la implementación de estas ideas, es decir una serie de estructuras cada una de las cuales procede de un modo diferente para garantizar el cumplimiento de las pautas constructivas BSS.

3.2.1. *Linked Exact Bottom Up* - BSS

Una posibilidad consiste en tomar *λ=ceiling((B+1)/2)* en el esquema de **FIGURA 3.2**, esto es partir por el medio, o por la posición inmediata siguiente, cada segmento cuyo span ha alcanzado el valor *B+1*. Esta propuesta se desarrollará en detalle más adelante en la presente Sección, de momento baste decir que el nombre *Linked* indica que se trata de una estructura formada únicamente por *nodos* enlazados, *Exact* que las particiones se realizan exactamente por el punto medio de los segmentos correspondientes en tanto *Bottom Up* es una alusión al modo en que la estructura se modifica ante cada *inserción*. Cuando se inicia una búsqueda se procede, tal lo anticipado, en descenso desde el *Header* de la más alta de las colas hasta arribar al punto de *inserción* en el nivel de los *eventos*, es decir el más bajo. Si la *inserción* del nuevo *evento* desencadena una *inserción auxiliar* esta se realiza. Producto de ella podría requerirse una nueva *inserción auxiliar* en el nivel inmediato superior. Así avanza el proceso de recomposición de "Abajo hacia Arriba" (*Bottom Up*) y ni bien se inserta en un nivel que no genera una nueva *inserción*, el proceso termina. Otra opción, conocida como de "Arriba hacia Abajo", consiste en realizar la verificación de span durante el descenso mismo. La corrección ya no se realiza en el instante en el que es estrictamente necesaria sino anticipándose a la superación del valor *B* por parte de los segmentos en cuestión. Lo que se hace es chequear el valor de span de todo segmento hacia el cual se va a descender y si es *B* se realiza su partición (sin que ello afecte el camino de descenso). Esto garantiza que luego de la *inserción* final en el nivel de los *eventos* ningún segmento de la estructura va a ostentar un valor de span superior a *B*, no siendo necesario ningún proceso ulterior.

3.2.2. *Linked Top Down* - BSS

Esta opción apela al mecanismo de recomposición de "Arriba hacia Abajo" sobre *nodos* enlazados, pero no tiene el carácter de *Exact*. En su lugar la condición para realizar una *inserción auxiliar* se relaja un tanto y no se recurre a la medición de span de los segmentos por los que progresa el descenso sino al recuento de *nodos* consecutivos atravesados en cada nivel. Así toda vez que se atraviesan *B nodos* pertenecientes a un mismo nivel se inserta un *nodo auxiliar* en el nivel superior, enlazado exactamente sobre el *B*-ésimo *nodo* recorrido. Esto evita la necesidad de registrar el valor de span en los *nodos auxiliares* así como también realizar la división en dos y el desplazamiento hasta el centro de los segmentos a particionar, pero el costo a pagar por ese ahorro es un peor caso bastante perjudicial que se da para secuencias de *inserciones* tales que el valor de tiempo sea siempre menor que el tiempo del *B*-ésimo *nodo* actualmente en la lista de *eventos*. Para esa secuencia no se genera estructura *auxiliar* alguna. Además la configuración permite la *inserción* de *nodos auxiliares* sobre *nodos* consecutivos en cualquier nivel. Los dos inconvenientes citados son salvados (desde luego pagando un cierto precio) por las implementaciones presentadas en los dos los puntos siguientes.

3.2.3. *Linked Check-Right Top Down* - BSS

Los principios son los mismos que los del punto anterior sólo que se busca evitar la generación de segmentos "demasiado" cortos. Con tal propósito, antes de realizar una *inserción auxiliar* se verifica que el *nodo auxiliar* que quedará a continuación del *nodo auxiliar* a *insertar* (eventualmente el *Rear* del nivel correspondiente) esté sobre un *nodo* desplazado una distancia no menor a *B* del *nodo* sobre el que se realizará la nueva *inserción*. La configuración resultante se beneficia al precio de una comparación más por *inserción*.

3.2.4. *Linked Bidirectional Top Down* - BSS

En este caso se procura evitar la generación de segmentos "demasiado" largos en la implementación 3.2.2 ocasionadas por *inserciones* dentro de segmentos existentes tales que demanden recorridos siempre menores a *B*. La propuesta para solucionar este problema consiste en transformar toda la estructura en doblemente enlazada. De esta manera las *inserciones* se pueden alternar, invirtiéndose los roles de *Header* y *Rear* cada vez, realizando el proceso una vez de "izquierda a derecha" y otra de "derecha a izquierda". En este caso el beneficio debe contrarrestar el aumento de costo introducido por el doble enlazado.

3.2.5. *Linked Move Top Down* - BSS

La propiedad *Exact*, hace que el tamaño de los segmentos generados por las particiones se mantenga dentro de ciertos límites (ver Sección **3.3.2.1**) y también garantiza, en alguna medida, que la cantidad de *nodos auxiliares* creados es la más adecuada para el valor de *B* elegido. El objetivo de esta propuesta es buscar eficiencia reduciendo al mínimo la creación de *nodos auxiliares*. Para ello, sobre la estructura simplemente enlazada, se resuelven las *inserciones auxiliares* de manera tal que en lugar de crear un *nodo auxiliar* cada vez que se han recorrido *B nodos* de un cierto nivel, se verifica si existe alguno en una posición más adelantada (hacia la derecha) en el nivel que se requiere la *inserción*. En caso afirmativo se trae (hacia la izquierda) hasta la posición de interés el *nodo* existente, junto con toda la columna que pudiera existir sobre él. De esta manera sólo

se crean *nodos auxiliares* cuando han de ocupar la última posición en el nivel que son requeridos. El precio a pagar es una comparación adicional tendiente a detectar intentos de desplazar el *Rear* de cualquier nivel, y también la generación de segmentos más largos que *B*, a continuación de *nodos* que acaban de sufrir un desplazamiento hacia la izquierda. La recomposición se logra entonces sobre el "sector izquierdo" (el de tiempos más bajos) de la estructura resultando ésta muy influida por la distribución temporal de los *eventos*.

3.2.6. *Array Bottom Up* - BSS

Esta variante también se basa en el principio de particiones sucesivas de lista para evitar recorridos secuenciales largos, sólo que las listas no se implementan mediante el enlazado de *nodos* simples sino a través de *celdas* de longitud fija, cada una de las cuales equivale a una lista de hasta *B nodos*. Las *celdas* se componen internamente de un arreglo (de ahí el nombre de *Array* de la implementación) cada uno de cuyos de *B* elementos aloja un puntero y un valor de tiempo.

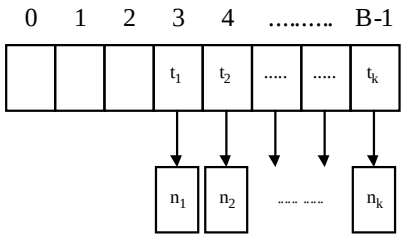


FIGURA 3.4 Enlazado de *nodos* en *Array Bottom Up* - BSS

Los elementos son accesibles dentro de la *celda* respectiva mediante un índice *i∈{0,1,2,...,B-1}*. Además cada *celda* mantiene dos parámetros: el valor de índice del primer elemento ocupado (*k*) y un indicador para identificar las *celdas* pertenecientes al nivel más bajo (*bottom*) de modo que al comienzo la estructura se compone de una única *celda* vacía (*k=B* y *bottom=1*). La *inserción* de *nodos* se realiza enlazándolos a la *celda* tal como se muestra en **FIGURA 3.4**, llenándola de derecha a izquierda.

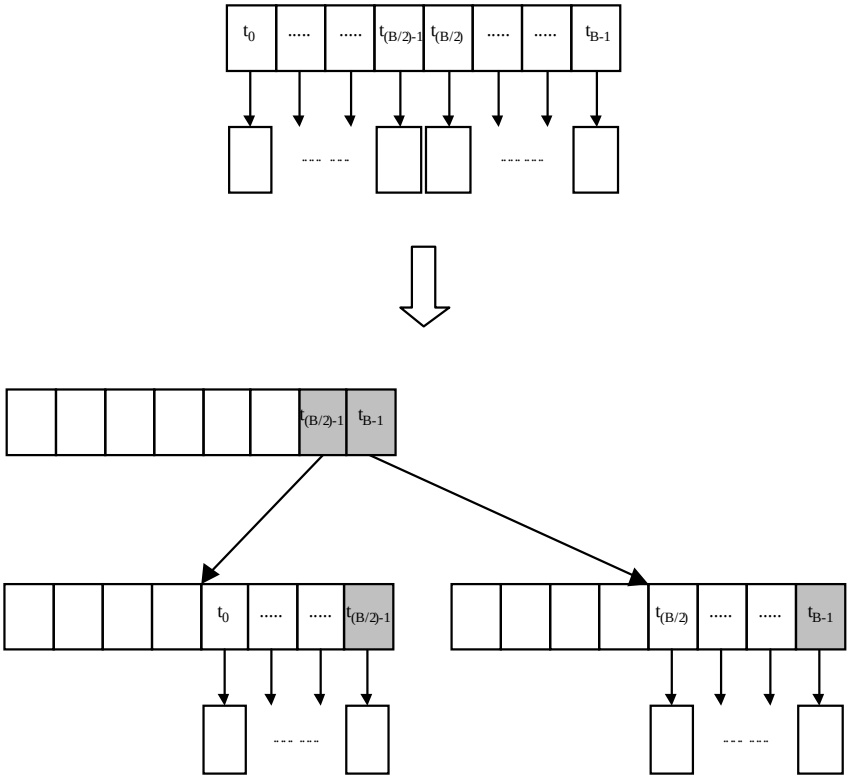


FIGURA 3.5 Operación *Split_Create* en *Array Bottom Up* - BSS

Los valores de tiempo de los *nodos* se van copiando sobre los elementos del arreglo desde el que son apuntados y el valor del parámetro *k* de la *celda* va descendiendo una unidad por cada inserción. Se conserva el orden creciente de los valores de tiempo, por lo que además de ser precedidas de una búsqueda dentro de la *celda* misma (sea Secuencial o Binaria) es probable que sea necesario desplazar (*shift*) los elementos existentes, o parte de ellos, para hacer lugar. El crecimiento de la estructura prosigue con la asistencia de dos operaciones: *Split* y *Extend*. La primera realiza la partición de *celdas* que se llenan completamente y la segunda extiende el alcance de la estructura cuando se quiere *insertar* un *nodo* cuyo valor de tiempo no encuentra sitio entre los espacios disponibles de las *celdas* existentes. La primer aplicación de *Split*, que se muestra en **FIGURA 3.5**, se produce cuando la *celda* por la que comienzan las *inserciones* se completa. Entonces se crean dos *celdas* más; sobre una de ellas se copian los primeros *(B/2)* elementos de la *celda* existente, es decir la mitad correspondiente a los tiempos más bajos. Luego ambas *celdas* (aquella desde la cual se copió y aquella hacia la cual se copió) copian el valor de tiempo de su *(B-1)*-ésimo elemento (el más alto) en cada una de las dos posiciones más altas de la tercer *celda*, respetando el orden creciente. Finalmente se apuntan los punteros de los dos elementos ocupados de esa tercer *celda* hacia cada una de las otras dos de manera que cada elemento apunte a la *celda* desde la que fue copiado su valor de tiempo. Desde la *celda* más alta ya no se apunta a *nodos* sino a *celdas*. Las más bajas conservan el valor *1* para *bottom* y llevan *k* a *(B/2)* en tanto la más alta fija *k* en *(B-2)* y *bottom* en *0*.

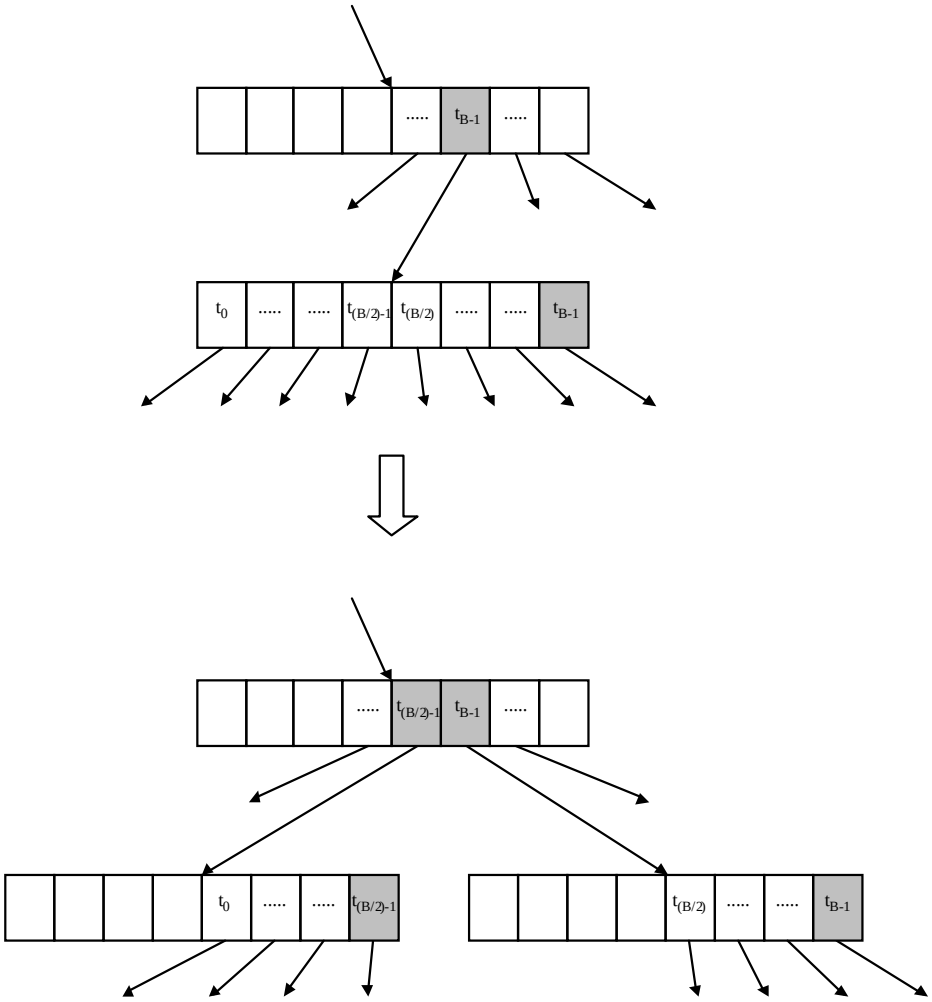


FIGURA 3.6 Operación *Split* en *Array Bottom Up* - BSS

De aquí en más las *inserciones* se inician recorriendo la *celda* más alta (sea en forma Secuencial o Binaria) hasta encontrar el elemento cuyo valor de tiempo sea el menor de los que superan el tiempo del *nodo* a *insertar*. Una vez encontrado se desciende a través de su puntero y se repite el proceso en la *celda* siguiente. En general se debe llegar hasta una *celda* con *bottom=1*, es decir una *celda* cuyos punteros ya no apunten a nuevas *celdas* sino a *nodos*, y allí se practica la *inserción*. Cuando alguna *celda* distinta de la más alta se llena, se realiza su partición mediante un mecanismo similar al anterior (ver **FIGURA 3.6**) sólo que ahora no es necesario crear dos nuevas *celdas* sino sólo una. Desde la *celda* que se acaba de completar se copia la mitad más baja sobre la mitad más alta de la nueva, se copia el elemento más alto de la nueva *celda* sobre la *celda* más alta y finalmente se

apunta el puntero correspondiente a la nueva *celda*. En la implementación se distingue esta última operación llamándola *Split*, de la anterior (la que crea dos nuevas *celdas*) a la que se denomina *Split_Create*.

En esta variante el proceso de particionar toda *celda* que se llene por completo se realiza de "Abajo hacia Arriba" (*Bottom Up*). Se comienza verificando el nivel más bajo, o sea aquel en el que se ha insertado un *nodo* (*bottom=1*), y eventualmente se realiza su partición. Producto de la misma podría producirse el llenado de la *celda* que le apunta desde el nivel inmediato superior. Se realiza entonces su verificación y eventualmente su partición. Las iteraciones terminan ni bien se practica una *inserción* que no llena por completo la *celda* superior. El proceso podría incluso culminar con una operación *Split_Create* en caso que se llegue a afectar el nivel más alto de la estructura.

El camino que se inicia en el elemento *(B-1)*-ésimo de la *celda* más alta y continúa hasta que encuentra una *celda* con *bottom=1*, descendiendo siempre por el elemento *(B-1)*-ésimo de cada *celda* visitada tiene dos propiedades:

1) El tiempo leído en todos los elementos visitados es el mismo y además es el más alto de todos los tiempos registrados en la estructura: $t_{máx}$.

2) $t_{máx}$ constituye una suerte de barrera o tope dado que, mediante las operaciones comentadas hasta el momento, únicamente pueden *insertarse* en la estructura *nodos* con valor de tiempo menor que $t_{máx}$.

La operación *Extend* se encarga de resolver la *inserción* de *nodos* con $t \geq t_{máx}$. El mecanismo propuesto para la misma consiste en reemplazar el valor de tiempo de todos los elementos que constituyen la barrera $t_{máx}$ por el tiempo del *nodo* a *insertar*, salvo el último. En la *celda* más baja del camino $t_{máx}$ no se realiza el reemplazo sino la *inserción* del nuevo *nodo* en la posición *(B-1)*-ésima para lo cual es preciso hacer lugar mediante un desplazamiento (*shift*) de todas las posiciones ocupadas. Esta última *inserción* podría completar la *celda* correspondiente por lo que también es preciso lanzar el mecanismo de recomposición de "Abajo hacia Arriba".

Los Árboles m-arios o Multivías [10, 27] se implementan interconectando *nodos* que guardan cierto parecido con las *celdas* de esta implementación. Una diferencia importante está en el número de punteros de los *nodos* que componen los citados Árboles que es, en todos los casos, uno más que el número de elementos del arreglo, tal como se muestra en **FIGURA 3.7**. Dado un elemento e_i de una *celda* en una implementación *Array Bottom Up* - BSS, el correspondiente puntero p_i conduce a una celda, en el nivel inferior, con tiempos menores o iguales que t_i. En un Árbol Multivías para todo par de elementos consecutivos $\{e_i,e_j\}$ existe un puntero p_{ij} que conduce a una *nodo* con valores de tiempo comprendidos entre t_i y t_j. El puntero que está más a la izquierda conduce a un *nodo* con valores de tiempo menores y el que está más a la derecha a un *nodo* con valores de tiempo mayores que el elemento del *nodo* desde el que se apunta.

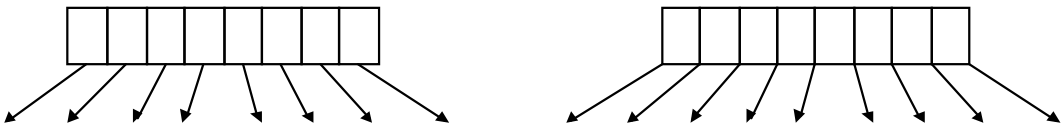


FIGURA 3.7 Distribución de Punteros en *Array Bottom Up* - BSS y en Árboles Multivías

Existe un ventaja para cada implementación en esta diferencia. Por un lado los Árboles no necesitan de la operación *Extend* (o similar) y pueden manejar todas las *inserciones* mediante operaciones de tipo *Split*. Es evidente también, y aquí está la ventaja a su favor, que cada *nodo* de Árbol permite un *nodo* más en el nivel inmediato inferior, diferencia ésta que contribuye a que la estructura desarrolle una altura menor que si contara con tantos punteros como elementos, y una altura menor reporta búsquedas más rápidas. En la práctica esta ventaja es tanto menos relevante cuanto mayor es *B*. Contrariamente puede reconocerse una ventaja en favor del esquema *Array Bottom Up* - BSS, con un puntero por elemento y operación *Extend*. Cuando en un Árbol Multivías se inserta un *nodo* con valor de tiempo más alto que el más alto de todos los existentes se realiza una búsqueda en descenso hasta localizar el punto de *inserción* para lo cual es necesaria una comparación en cada nivel visitado durante el descenso. En la estructura que se está presentando, una vez que se decide la aplicación de *Extend* ya no es necesaria ninguna comparación durante su aplicación. Cabe acotar además que a diferencia de otras aplicaciones, en las que en un determinado momento de la ejecución puede suceder la *inserción* de un *nodo* con valor de tiempo tal que ninguna *inserción* futura lo supere, en la SED son de esperar frecuentes superaciones de ese límite por lo que contar con una operación de bajo costo para esos casos es una ventaja.

Otro punto a considerar con relación a los Árboles Multivías es que estos no contemplan la repetición de valores de tiempo en los elementos de sus *nodos*. Se podría decir que "mueven" en lugar de "copiar" al ejecutar

operaciones de tipo *Split* o similares. Si la no repetición también contribuye a reducir la altura de la estructura, no menos cierto es que los algoritmos resultan más complicados. Desde este punto de vista en *Array Bottom Up* - BSS se paga entonces un precio en consumo de memoria para lograr mecanismos más simples. Existe también una relación entre el número de punteros (uno más por *nodo* en los Árboles) y la posibilidad de evitar repeticiones. El ejemplo mostrado en **FIGURA 3.8** pone de manifiesto esta dificultad. Allí se aprecia la evolución de una parte de la estructura ante la misma *inserción*, en un caso con punteros al estilo de los Árboles y no repetición de elementos y en el otro con los punteros del modo propuesto, quedando en evidencia la imposibilidad de evitar repeticiones en este último. Cuando la *celda* compuesta por 12-18-26 se completa con el agregado del 22, la *celda* compuesta por 15-16-17, que antes de la *inserción* "cuelga" del 18, no encuentra luego elemento al cual quedar enlazada en el modelo tentativo de *Array Bottom Up* - BSS sin repeticiones.

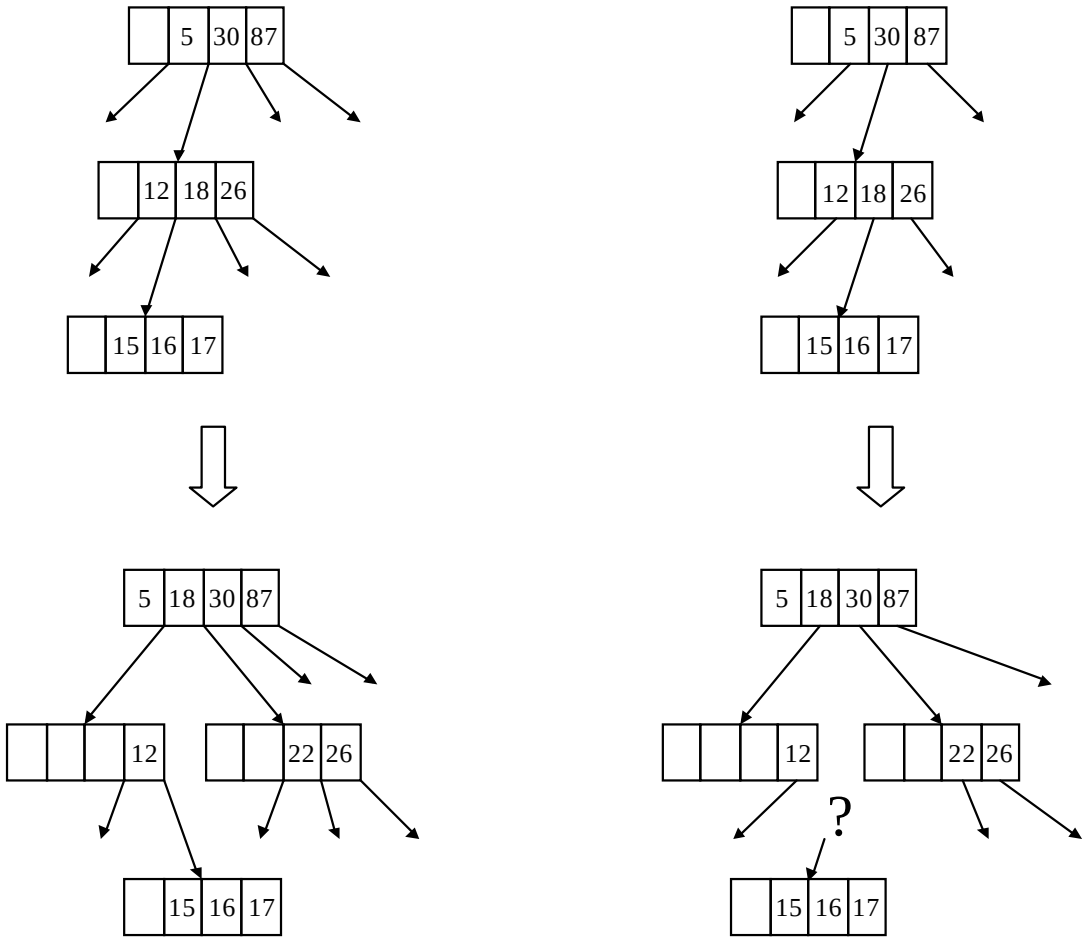


FIGURA 3.8 *Inserción* del 22 "sin repeticiones y punteros tipo Árbol"
y "sin repeticiones y punteros tipo *Array Bottom Up* - BSS"

Por otro lado, la forma en que ha sido planteada la *inserción* de *nodos* ("colgados" de las *celdas* del nivel más bajo, como lo hacen los B^+ Tree [27]) hace que en la programación de *Array Bottom Up* - BSS los punteros puedan desempeñar un doble papel (el de puntero a *nodo* o el de puntero a *celda*) según el punto del algoritmo desde el que son referidos. Si en cambio se alojara toda la información de los *nodos* dentro de cada elemento de *celda*, en procura de evitar repeticiones, además de convertirse las *celdas* en estructuras mucho más "pesadas", las de nivel más bajo desperdiciarían la totalidad de sus punteros.

3.2.7. *Array Top Down* - BSS

Esta última variante resulta de realizar el proceso de recomposición de "Arriba hacia Abajo" (*Top Down*) en el modelo presentado en el punto anterior. Al igual que en los casos *Linked*, la idea es ir realizando la verificación a medida que se produce la búsqueda en descenso del punto de *inserción*. Se realiza entonces la partición de toda *celda* que al momento de ser visitada aloja una cantidad *B* de elementos.

Un análisis comparativo entre los mecanismos *Top Down* y *Bottom Up* (y esto vale también para las implementaciones de tipo *Linked*) permite establecer en favor del primero una simplificación en los algoritmos y su correspondiente programación y también la posibilidad de evitar recordar el camino de descenso dado que si se realiza la verificación del nivel hacia el que se va a descender (o sea antes del descenso) se puede prescindir de la necesidad de enlaces hacia arriba. En favor de *Bottom Up* la ventaja consiste en que ni bien se produce una *inserción* que no requiere una futura partición, el proceso se detiene. De ello resulta que la cantidad de niveles verificados (y por ende de comparaciones) es, en general, menor que el número total de niveles mientras que en *Top Down* se verifican todos los niveles en todas las *inserciones*.

3.3 Análisis de la Implementación *Linked Exact Bottom Up* - BSS

A continuación se describen las principales características de *Linked Exact Bottom Up* - BSS, una de las siete implementaciones presentadas en esta Sección. Se trata de la variante que más se parece a la versión Determinística de Skip List, DSL [17]. Se comenta su mecanismo en forma detallada, se hace una evaluación experimental de sus operaciones, se determina su complejidad y se reportan algunas pautas de diseño.

3.3.1 Comparación con el Mecanismo de Particiones Sucesivas

La similitud y, en cierto sentido, la equivalencia entre *Linked Exact Bottom Up* - BSS y el mecanismo de particiones o subdivisiones sucesivas es abordada ahora con mayor detalle. El criterio mediante el cual *Linked Exact Bottom Up* - BSS inserta *nodos auxiliares* y determina por ende el valor de span de los segmentos generados depende de los tiempos de los *eventos* arribados para su *inserción*. Por esa razón, aún cuando de tamaño acotado, los segmentos son en general diferentes. A pesar de tratarse de un caso muy particular, la vista de **FIGURA 3.9**(a) captura la evolución de una cierta *Linked Exact Bottom Up* - BSS con *B=4*, y constituye un ejemplo útil para nuestro propósito. Nivel I corresponde a la lista de *eventos*, cuya creación ocurre en primer término. Sucesivamente los niveles II y III completan la estructura a medida que el número de *eventos* crece.

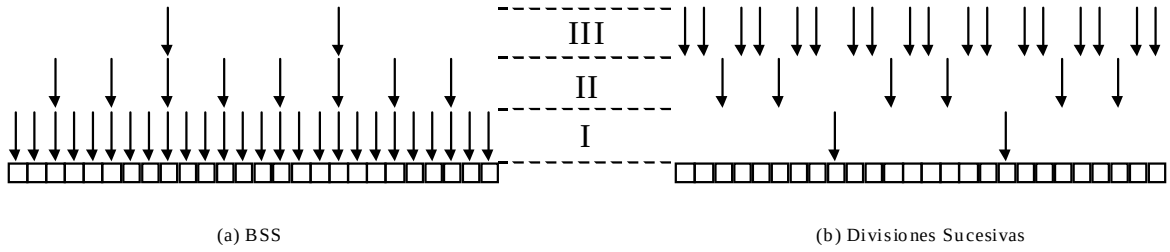


FIGURA 3.9 Comparación entre algoritmos

Por otro lado, la **FIGURA 3.9**(b) esquematiza la distribución de punteros destinada a permitir *inserciones* por subdivisión sucesiva (por *3*) de la misma lista de *eventos*. Nivel I muestra una primer tanda de punteros encargada de dividir la lista en tres segmentos del mismo tamaño. El *nodo* de más alto valor de *t* de los primeros dos segmentos resultantes debe ser chequeado a fin de determinar en cual de los tres se encuentra el punto de *inserción* para un nuevo *nodo*. (En **FIGURA 3.9**(b) todos los segmentos se muestran divididos, pero en realidad sólo el que contiene el punto de *inserción* debería particionarse). Los segmentos son iterativamente subdivididos hasta que ya no es posible proseguir lo que en el presente ejemplo ocurre cuando los último tres segmentos están compuestos por un único *nodo* (Nivel III).

Ambos mecanismos acometen el proceso de subdivisión en dirección opuesta. *Linked Exact Bottom Up* - BSS comienza por los segmentos de menor tamaño y evoluciona hacia los de mayor tamaño (sobre la lista original) mientras que el otro método lo hace al revés. No obstante hay una relación entre ambos mecanismos, incluso numérica. La configuración de *Linked Exact Bottom Up* - BSS mostrada en el ejemplo, con *B=4*, provee el mismo potencial de búsqueda que el método que va practicando divisiones por *3*. Extendiendo la idea, el método que va dividiendo la lista por *k*, encuentra un equivalente en una configuración de *Linked Exact Bottom Up* - BSS con *B=2(k-1)*. Pero el resultado más interesante proviene de una inspección visual de las **FIGURAS 3.9** (a) y (b). En definitiva la construcción de *Linked Exact Bottom Up* - BSS procede, en cualquier nivel, colocando *"1 nodo auxiliar por cada B-1 nodos en el nivel inferior"*. En el ejemplo mostrado lo hace a razón de *"1 nodo auxiliar por cada 3 en el nivel inferior"*. Como consecuencia termina finalmente partiendo la lista en 3 partes iguales (en Nivel III). Existe entonces una estrecha relación entre la "cuenta" de *nodos* para generar

segmentos en el nivel inferior y el número por el cual la lista es finalmente dividida en el nivel más alto. Y este hecho hace explícito el modo en que ambos mecanismos están relacionados.

El tamaño de la lista bajo análisis fue seleccionado deliberadamente, y la estructura de *Linked Exact Bottom Up* - BSS sólo se ajusta exactamente al patrón indicado como caso particular. De todos modos el ejemplo ilustra el hecho de que, en cierto sentido, ambos métodos se comportan en forma similar y que, en alguna medida, el mecanismo de *Linked Exact Bottom Up* - BSS tiende al de divisiones sucesivas.

3.3.2 Características de *Linked Exact Bottom Up* - BSS

Habiendo sido *Linked Exact Bottom Up* - BSS creada para administrar la lista de *eventos* en SED, salvo casos especiales (poco frecuentes), las *remociones* tendrán lugar siempre sobre el primer *nodo*, o sea el de tiempo más bajo. Sólo la presencia de *nodos auxiliares* enlazados por sobre el *nodo* a *remover* constituye una diferencia respecto a la *remoción* en una lista simplemente enlazada. El borrado de una "columna" completa de *nodos* sólo implica esfuerzo computacional, y dado que nunca genera segmentos de tamaño mayor que *B*, en principio ningún proceso debe ejecutarse inmediatamente luego. Llamando *s* al valor de span de los segmentos, sólo para el primero de cada nivel (el más hacia la izquierda) vale la relación $0 \leq s \leq B$, para el resto, $(B/2) \leq s \leq B$.

Si se ejecuta una larga secuencia de *inserciones*, en las que el tiempo para cada *evento* es leído de una variable aleatoria, con distribución uniforme de probabilidad, el span de los segmentos generados se va a distribuir de manera que la mayoría tendrá el valor *(B/2)*. Los siguientes, en proporción decreciente, valdrán *(B/2)+1*, *(B/2)+2*, ..., *B* (ver **FIGURA 3.10** obtenida experimentalmente para *B=16*).

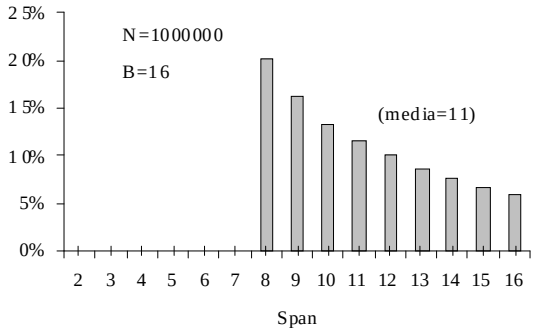


FIGURA 3.10 Distribución de Span en *Linked Exact Bottom Up* - BSS

Pero si en lugar de ser tomados de una variable aleatoria, los tiempos de los *eventos* a *insertar* son generados mediante un algoritmo como $t=t_{li}+r$, siendo t_{li} el valor de tiempo del último *nodo* insertado y *r* un valor positivo, fijo, entonces el valor de span de todos los segmentos convergerá al valor *(B/2)*

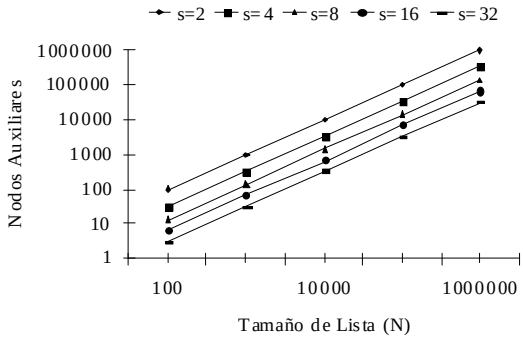


FIGURA 3.11 Estructura Auxiliar de *Linked Exact Bottom Up* - BSS

Ninguna de estas dos evoluciones constituye de por sí una aplicación real, pero ambas dan una clara idea sobre cómo evoluciona la construcción de *Linked Exact Bottom Up* - BSS así cómo de la relación entre el valor promedio de span de los segmentos y la distribución temporal de los *eventos* insertados. De hecho el span medio es el valor que, más que ningún otro, determina la velocidad de búsqueda sobre *Linked Exact Bottom Up* - BSS.

Bajo el supuesto de que el span de todos los segmentos esté fijado al mismo valor, la **FIGURA 3.11** muestra una característica interesante. Allí se puede apreciar una estimación del tamaño de la estructura *auxiliar*, medida en el número de *nodos auxiliares*.

De ser utilizada para distintos propósitos, como por ejemplo la implementación de diccionarios, donde los borrados ocurren en cualquier lugar de la lista frecuentemente, un proceso de reestructuración de *Linked Exact Bottom Up* - BSS deberá seguir a cada *remoción* de un *nodo* intermedio. La idea es mantener en niveles adecuados la relación entre el span de los segmentos y el valor de *B*. (Ver Sección **3.3.7.4**).

3.3.2.1. Relación Fundamental

Para que tanto los mecanismos básicos de operación de *Linked Exact Bottom Up* - BSS como las determinaciones de complejidad sean válidas en todo momento, es preciso que ninguna de las operaciones altere sustancialmente su "forma". La configuración deseada se corresponde con las pautas de construcción de la operación *insertar*, esto es: todo segmento cuyo span alcanza el valor (B+1) se parte en dos (Izquierdo y Derecho) de modo que,

Si B es par:	span(segmento Izquierdo)	= (B+1)/2 = B/2
	span(segmento Derecho)	= (B+1)/2 = B/2
Si B es impar:	span(segmento Izquierdo)	= (B+1)/2
	span(segmento Derecho)	= B/2

La clasificación de Izquierdo y Derecho únicamente tiene sentido al momento de la partición, pero no de ahí en más. No obstante, sus valores permiten conocer los límites por debajo de los cuales no estará el span de ningún segmento. (B/2) será entonces el mínimo valor de span de la estructura. Por otro lado ningún segmento puede superar el valor de B, caso contrario se realiza su partición. Luego, la relación $\Re$ que rige el valor de span de todos los segmentos de *Linked Exact Bottom Up* - BSS es:

$$\forall s \in \textit{Linked Exact Bottom Up} \text{ - BSS} \qquad (B/2) \leq \text{span}(s) \leq B$$

Es importante que las funciones de borrado aseguren que luego de su aplicación se siga cumpliendo $\Re$, por lo que, en general, ni bien realizada una *remoción* será preciso ejecutar un proceso de reestructuración. En cierto sentido este proceso haría las veces de las rotaciones en los árboles binarios de búsqueda, practicadas para restituir condiciones de balance luego de una actualización. El no cumplimiento de $\Re$ puede dar lugar a secuencias de borrado particularmente nocivas, capaces de producir configuraciones como por ejemplo las que se muestran en la **FIGURA 3.12**, para las cuales el costo de búsqueda es ***O(n)***, y para las que los mecanismos pensados para *Linked Exact Bottom Up* - BSS no reportan beneficio alguno.

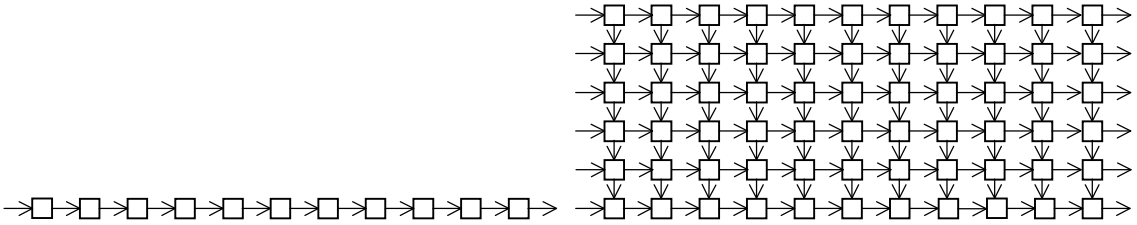


FIGURA 3.12 Dos casos de *Linked Exact Bottom Up* - BSS que no cumplen $\Re$

Dado que el proceso de reestructuración es un tanto costoso, a los fines prácticos puede admitirse una violación parcial de $\Re$ por parte de las funciones que remueven el primero y el último *registro*. Para una *Linked Exact Bottom Up* - BSS que aloje gran cantidad de *nodos* se puede aceptar que en el primer segmento de cada nivel (el más hacia la izquierda) o en el último, el span pueda caer por debajo de B/2. (En rigor, el mecanismo de reestructuración que se va a presentar en Sección **3.3.7.4** no es aplicable para reestructurar el último segmento de ningún nivel).

3.3.3 Llenado de *Linked Exact Bottom Up* - BSS

En simulaciones reales nunca ocurren separadamente *inserciones* o *remociones* en forma sostenida durante espacios de tiempo muy largos. Hay períodos en los que una prevalece sobre la otra y períodos en los que una suerte de compensación mantiene el número de *eventos* más o menos estable. No obstante ensayar un *calendario* con tests de "sólo *inserciones*" o "sólo *remociones*" pero no ambas, puede resultar esclarecedor y constituir una importante ayuda para futuras conclusiones sobre su comportamiento.

Si sólo *inserciones* ocurren, para un valor fijo de *B*, el tiempo de *inserción* crecerá logarítmicamente con el tamaño de la lista. Este hecho fue delineado mediante el análisis de un método similar en **FIGURA 3.1**, y será abordado en mayor detalle en Sección **3.3.6**. Algunas *inserciones* podrán desencadenar *inserciones auxiliares* (desde una hasta el número de colas *auxiliares* más uno). El número de *inserciones auxiliares* depende del valor de span de los segmentos a través de los cuales la *inserción* progresa, pero no está sujeto a ninguna condición respecto al tamaño de la lista de *eventos* ni al cruce por ningún valor umbral en dicho tamaño, razón por la cual el comportamiento de las *inserciones* no presenta "saltos" en los que hace a su costo, y esta es una propiedad importante de *Linked Exact Bottom Up* - BSS.

Si bien las configuraciones de *Linked Exact Bottom Up* - BSS con valores grandes de *B* resultan computacionalmente poco costosas de mantener, dado el bajo número de *nodos* y colas *auxiliares*, la estructura tiende a convertirse en una lista simplemente enlazada carente del mecanismo provisto para acelerar las búsquedas. Por esa razón a partir de determinado rango de valores de B los tiempos de *inserción* resultan crecientes. Por otro lado, al acercarse *B* a los mínimos valores posibles (esto es valores cercanos a *2* ya que aún cuando *1* también es posible se probará más adelante que conduce a una configuración muy ineficiente) el mecanismo de búsqueda tiende a parecerse al de la Búsqueda Binaria [11], es decir a la opción algorítmicamente más eficaz, pero en este caso el tamaño de la estructura *auxiliar* tiende a crecer y con ella el costo de su mantenimiento (ver **FIGURA 3.13**). Estas son las razones por las cuales los mejores rendimientos se verifican (experimentalmente) para valores de *B* entre *4* y *16* en un extenso rango de tamaños de la lista de *eventos*.

Para lograr ensayos con resultados estadísticamente confiables, el número de operaciones ejecutadas deberá ser suficientemente grande como para permitir la ocurrencia de *inserciones* de todo tipo, es decir cubriendo toda la gama posible de acuerdo al número de *inserciones auxiliares*. Los detalles técnicos y de implementación de todos los ensayos realizados sobre *Linked Exact Bottom Up* - BSS se comentan en la Sección **5.3**.

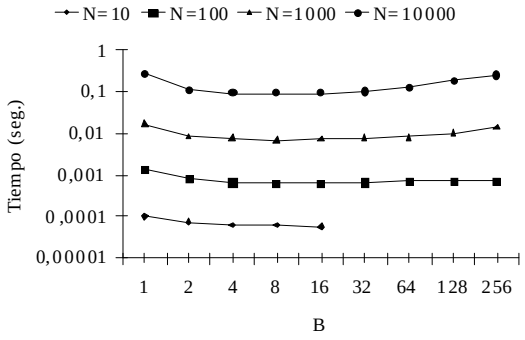


FIGURA 3.13 Estructura Auxiliar de *Linked Exact Bottom Up* - BSS

3.3.4 Vaciado de *Linked Exact Bottom Up* - BSS

Con relación al número de operaciones a practicar en un ensayo de "sólo *remociones*", el criterio es garantizar que columnas de todas las alturas han de ser borradas.

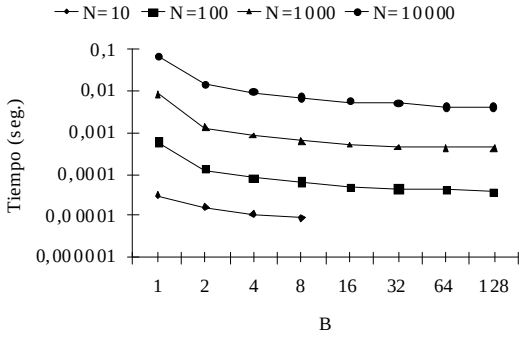


FIGURA 3.14 Tiempo de Vaciado de *Linked Exact Bottom Up* - BSS

Para un valor fijo de *N*, las *remociones* son más rápidas a medida que *B* crece porque el número de colas *auxiliares* y la densidad de *nodos auxiliares* son máximos para *B=1*, comenzando a decrecer a medida que se incrementa *B*. Al alcanzar *B* valores suficientemente grandes la estructura se convierte en una lista simplemente enlazada y el tiempo de las *remociones* en el mínimo posible (ver **FIGURA 3.14**).

3.3.5 Operaciones de Hold sobre *Linked Exact Bottom Up* - BSS

La interpretación de resultados de un ensayo sobre *The Hold Model* (ver Sección **5**) puede apoyarse en los experimentos previos de "sólo *inserciones*" y "sólo *remociones*". En cierta medida los efectos de ambos tests interactúan ahora haciendo crecer los tiempos de Hold para valores bajos de *B*, debido al costo de la administración de un número elevado de *nodos auxiliares* y también para valores altos de B debido a que una reducida estructura *auxiliar* produce elevados tiempos de búsqueda. **FIGURA 3.15** muestra estos dos efectos para distintos valores de *N*.

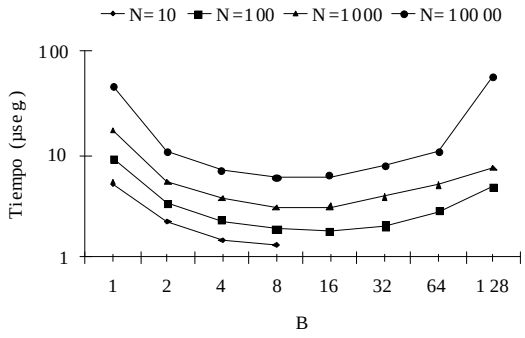


FIGURA 3.15 Tiempo de Hold de *Linked Exact Bottom Up* - BSS

3.3.6 Determinaciones de Complejidad sobre *Linked Exact Bottom Up* - BSS

Sea una estructura *Linked Exact Bottom Up* - BSS formada por un total de *h* colas (la lista de *eventos* más *h-1* colas *auxiliares*), numeradas en forma ascendente de *1* a *h*, cada una de las cuales conteniendo N_i *nodos*, *i=1,2,3,...,h*, de manera que $N_1=N$ es el número total de *eventos*. Aceptando que todos los *nodos*, excepto los de la lista de *eventos* y todos los *Headers*, alojan un valor de span s_{ij}, donde *i* denota la cola y *j* la posición dentro la misma, el span medio está dado por la expresión:

$$s = \left(\sum_{i=2}^{h} \sum_{j=1}^{N_i+1} s_{ij} \right) / \sum_{i=2}^{h} (N_i + 1)$$

La **FIGURA 3.16** indica el modo en que la determinación de span se realiza. De todas maneras, para abordar la determinación de la altura se supondrá que todos los segmentos de *Linked Exact Bottom Up* - BSS tienen el valor de span medio, *s*.

Sea n_i el número de *nodos* cuyo valor de *t* es potencial candidato a ser comparado contra el valor de *t* del *nodo* a *insertar*, en cada una de las colas. Entonces,

$n_1 = N+1$
$n_2 = n_1/(s+1) = (N+1)/(s+1)$
$n_3 = n_2/(s+1) = n_1/(s+1)^2 = (N+1)/(s+1)^2$
:
:
$n_h = n_{h-1}/(s+1) = n_1/(s+1)^{h-1} = (N+1)/(s+1)^{h-1}$

La cola del nivel más alto se aceptará formada por un único segmento, o sea *s nodos*. De contener más no sería la última. Podría aceptarse que contenga menos que *s*, pero por un lado se propuso que todos los segmentos fueran iguales y por otro (como se verá) *s* es el valor que conducirá a una valor más costoso para las búsquedas.

A partir de la última igualdad y dado que el *Rear* de la cola más alta también es susceptible de ser chequeado resulta ser *(s+1)* el valor de n_h. Sigue entonces que $(N+1) = (s+1)^h$ y por lo tanto $h = \log_{s+1}(N+1)$ (que para valores grandes de *N* se convierte en $h = \log_{s+1} N$).

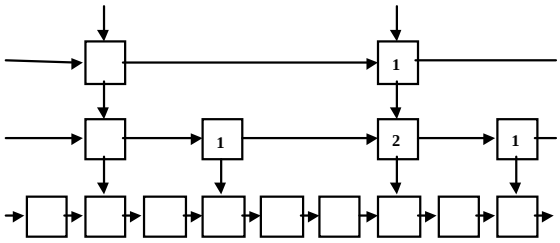
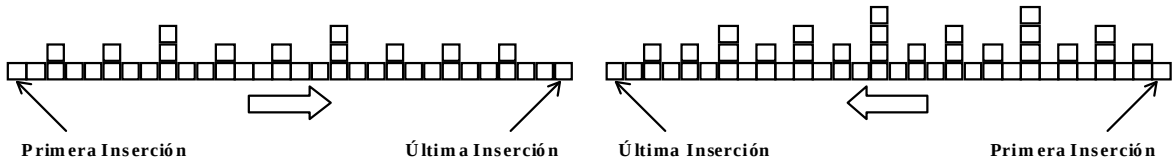


FIGURA 3.16 Determinación del Span en *Linked Exact Bottom Up* - BSS

Dado que no hay límite explícito para *h*, conviene extender su análisis, fundamentalmente para evaluar la factibilidad del caso $s \rightarrow 0$ en que la estructura podría crecer sin límite. Para cualquier valor de *N*, el máximo posible de *h* se alcanza sí y sólo sí el arribo de los *nodos* a *insertar* se produce en una secuencia determinada. Para que tal secuencia sea efectiva, una vez que han arribado los *B+1* primeros *nodos*, y que la primer subdivisión ha ocurrido, los *nodos* siguientes deberán "caer" dentro del mismo segmento, hasta que lo llenen completamente. Los próximos *nodos* deberán caer también dentro de un mismo segmento, hasta llenarlo, y así sucesivamente. La clave del proceso es que no cualquier segmento es igualmente útil para alojar conjuntos de *nodos* arribados. Es preciso que los *nodos auxiliares* promovidos hacia niveles más altos también caigan dentro de un mismo segmento, mientras hay lugar para ello. Y el mismo criterio vale para todos los niveles siguientes.



(a) Secuencia con valor creciente de *t*. B=3 (b) Secuencia con valor decreciente de *t*. B=3

FIGURA 3.17 Secuencias más Efectivas

Si los *eventos* arribados se "desparramaran" todo a lo largo de la lista, la secuencia no sería efectiva en el sentido de hacer crecer al estructura hasta su máxima altura. Dado que las *remociones* únicamente acortan el valor de span de algún segmento, únicamente se considerarán secuencias de *inserciones* sin que ello quite generalidad al análisis. En particular se mostrarán dos secuencias efectivas, una de las cuales será reconocida como la mejor.

La primera de ellas está compuesta por *eventos* con valor de *t* siempre creciente, por lo que cada *nodo* es enlazado a la lista mediante la operación "append" y mediante idéntica operación son agregados a la cola correspondiente todos los *nodos auxiliares* generados. **FIGURA 3.17**(a) muestra el caso para *B=3*. Recíprocamente la otra posibilidad está dada por una secuencia de *eventos* con valor de *t* siempre decreciente, resultando los *nodos* permanentemente enlazados por la cabecera de la lista, y lo mismo para todos los *nodos auxiliares* generados. El caso *B=3* evoluciona de acuerdo a lo mostrado en la **FIGURA 3.17**(b). El resultado es claro en favor de al opción (b) y la razón de ello es que, al ser divididos los segmentos mediante la función *ceiling((B+1)/2)*, si B es impar, luego de cada división el segmento izquierdo es mayor que el derecho. *Insertar* siempre en el segmento de mayor tamaño hace que el límite de *B* sea alcanzado más rápidamente. Cuando *B* es par, agregar los *nodos* nuevos a derecha o izquierda no implica diferencia. Desde luego enlazar *nodos* permanentemente por la cabecera de la lista constituye una suerte de idealización ya que en sistemas reales el tiempo raramente será decreciente en todo momento (eso sólo será factible en ciertos períodos). De cualquier manera, el propósito de este análisis es únicamente la determinación de un límite para el valor de *h*, por lo que la consideración de ese tipo de secuencia, aunque ideal, es válida. Si en lugar de hacerlo mediante la función *ceiling((B+1)/2)* los segmentos fuesen divididos usando la función *floor((B+1)/2)*, la secuencia más efectiva hubiese sido la de *eventos* con valor de *t* siempre creciente.

Considerando entonces *eventos* generados según la secuencia más efectiva (valor de *t* siempre decreciente) es posible inferir el número de *nodos* en un nivel cualquiera, como función del número en el nivel inmediato inferior.

$N_{i+1} = (N_i - ((B/2)+1)) / ((B/2)+1)$ $\quad i=1,2,...,h-1$ si B es impar.
$N_{i+1} = (N_i - (B/2)) / ((B/2)+1)$ $\quad i=1,2,...,h-1$ si B es par.

Donde / denota la división entera. Véanse los resultados para algunos valores de B:

$N_{i+1} = (N_i - 1)$ $\quad B=1$
$N_{i+1} = (N_i - 1) / 2$ $\quad B=2$
$N_{i+1} = (N_i - 2) / 2$ $\quad B=3$
$N_{i+1} = (N_i - 2) / 3$ $\quad B=4$

Estas funciones, en particular la primera de ellas, ponen en evidencia un caso crítico. Si $B=1$ la estructura crece como se muestra en la **FIGURA 3.18**, el número de *nodos* en cada nivel es obtenido mediante sustracción de *1* a la cantidad del nivel inmediato inferior y la consecuencia es un costo $\boldsymbol{O}(N)$ para las búsquedas. Para lograr que el costo se mantenga en el orden de $\boldsymbol{O}(\log N)$ es preciso que el número de *nodos* de cada nivel resulte de la división de la cantidad del nivel inferior por un factor mayor que *1*, lo cual para la secuencia en cuestión sucede cuando $B>1$. Luego una importante recomendación surge, en cuanto a descartar el valor $B=1$ toda vez que el costo de búsqueda (peor caso) quiera mantenerse mejor que $\boldsymbol{O}(N)$. Por otro lado, la ineficiencia del caso $B=1$ fue comprobada experimentalmente.

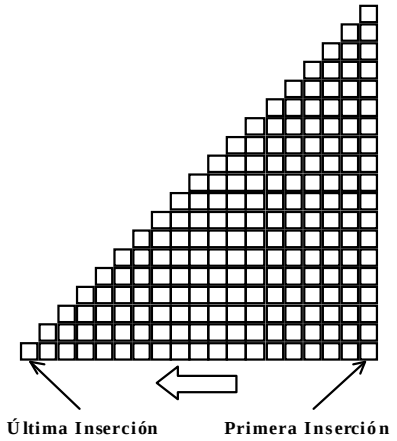


FIGURA 3.18 Secuencia con valor decreciente de *t* y B=1

A medida que N crece,

$(N_i - ((B/2)+1)) \approx (N_i - (B/2)) \approx N_i$

La simplificación sólo tiene sentido para $B>1$, porque $B=1$ convertiría la expresión $N_{i+1}=N_i-1$ en $N_{i+1}=N_i$, o sea en una función de crecimiento ilimitado. La simplificación no parece ser válida en niveles altos tampoco ya que en ellos N_i es del orden de B. No obstante su aplicación lleva a considerar un número de *nodos* mayor que el real por lo que, siendo el objetivo la obtención de un límite para el valor de h, la simplificación puede aceptarse. En cualquier caso, de aquí en más B será considerado siempre mayor que *1*.

Entonces, a partir de la simplificación,

$N_{i+1} = N_i / ((B/2)+1)$ $\quad i=1,2,...,h-1$ $\quad$ para $B>1$ (ya sea par o impar)

Luego la estructura crece según,

$N_1 = N$
$N_2 = N_1 / ((B/2)+1) = N / ((B/2)+1)$
$N_3 = N_2 / ((B/2)+1) = N / ((B/2)+1)^2$
:
:
$N_H = N_{H-1} / ((B/2)+1) = N / ((B/2)+1)^{H-1}$

Aceptar a H como el nivel ocupado más alto es como aceptar que no hay elementos alojados en $H+1$. Luego, el valor de H para el cual $H+1$ alojara un único *nodo* es de hecho un límite a la altura de *Linked Exact Bottom Up* - BSS, construida a partir de la "secuencia más efectiva", con N *eventos* y $B>1$.

$N_{H+1} = N_H / ((B/2)+1) = N / ((B/2)+1)^H = 1 \quad \rightarrow \quad H = \log_{((B/2)+1)} N$

De la comparación entre la altura media h y el límite H recientemente determinado (es decir $h=\log_{s+1} N$ y $H=\log_{((B/2)+1)} N$) surge una interesante observación: toda vez que $B>1$ el valor de s nunca será inferior a $(B/2)$.

Pero lo más relevante de esa comparación está en el hecho de que para $B>1$ tanto el valor medio como la cota superior de la altura de *Linked Exact Bottom Up* - BSS evolucionan de acuerdo al logaritmo del número de *eventos* alojados. La **TABLA 3.1** compara resultados experimentales contra determinaciones analíticas de altura. Para cada par de N y B mostrados, h y s se determinan mediante la corrida de un programa. El valor real de h es un entero por lo que su evolución es discreta y en saltos de una unidad. Ello torna complicado el seguimiento del mismo mediante las funciones continuas $log_{s+1} N$ y $log_{((B/2)+1)} N$. De todos modos los resultados obtenidos se ajustan razonablemente a los valores reales.

A través del análisis realizado hasta ahora la condición $B<<N$ (B mucho menor que N) fue implícitamente aceptada ya que cuando $B \rightarrow N$ entonces $h \rightarrow 0$ y la estructura se convierte en una lista simplemente enlazada, sin el mecanismo propio de *Linked Exact Bottom Up* - BSS sobre sí.

N	B	s	h	$log_{(s+1)} (N+1)$	$log_{((B/2)+1)} N$
1000	2	1.338	8	8.135	9.966
1000	4	2.678	6	5.305	6.288
1000	8	5.424	4	3.714	4.292
1000	16	10.564	3	2.822	3.144
10000	2	1.338	11	10.845	13.288
10000	4	2.677	7	7.074	8.384
10000	8	5.532	5	4.908	5.723
10000	16	11.217	4	3.680	4.192

TABLA 3.1 Determinación de altura en *Linked Exact Bottom Up* - BSS

Luego del análisis de la altura de *Linked Exact Bottom Up* - BSS, se aborda la evaluación de complejidad de la operación de *inserción*. El proceso siempre se inicia con una Búsqueda en Descenso (ver Sección **3.3.7.2**), prosigue con el Enlazado del Nodo a Insertar y culmina con las Inserciones Auxiliares (ver Sección **3.3.7.3**), esto es:

Búsqueda en Descenso + Enlazado del Nodo a Insertar + Inserciones Auxiliares

Dado que sólo el primero y el último de estos tres procesos realizan comparaciones, la complejidad de la *inserción* (C_I) se determina como:

$$C_I = C_{BD} + C_{IA}$$

Donde C_{BD} es la complejidad de la Búsqueda en Descenso y C_{IA} la complejidad de las Inserciones Auxiliares.

De acuerdo a la definición de *Linked Exact Bottom Up* - BSS las Búsquedas en Descenso habrán de chequear un número de *nodos* comprendido entre 1 y $B+1$, en cada cola dado que el span de todos los segmentos está entre 0 y B. Para un determinado número de búsquedas el promedio de comparaciones, por nivel tomará un valor s'. Entonces:

$$C_{BD} = s' \cdot h = s' \cdot log_{s+1} (N+1) \qquad 1 \leq s' \leq B+1$$

Lo que indica que el costo medio para la Búsqueda en Descenso es $\boldsymbol{O}(log\ N)$. El mejor caso ocurre cuando el camino de búsqueda es tal que debe hacer sólo una comparación por nivel, mientras que el peor caso corresponde a un camino por el cual sea preciso hacer $B+1$ comparaciones en cada cola. La altura de la estructura seguirá siendo h, independientemente del camino tomado por la búsqueda. El mejor y el peor caso de la Búsqueda en Descenso serán entonces $log_{s+1} (N+1)$ y $(B+1) \cdot log_{s+1} (N+1)$ respectivamente, lo que significa que ambos son también $\boldsymbol{O}(log\ N)$.

Resta entonces establecer C_{IA} para completar la determinación de complejidad de todo el proceso de *inserción*. A tal efecto hay un resultado interesante en la determinación de complejidad de la *remoción*, el cual se aborda seguidamente. Se deja entonces para luego la determinación de C_{IA} y consecuentemente de C_I.

La clave para determinar la complejidad de las *remociones* es la altura de la columna de *nodos* a *remover* en cada operación. Estas columnas incluyen un *nodo* de la lista de *eventos* más un número adicional de *nodos auxiliares* que podrá variar entre 0 y $(h-1)$. Aún cuando los *Rears* no son *nodos* removibles, en el análisis que nos aprestamos a realizar atribuirles ese carácter reportará una importante simplificación. Las columnas de *Rears* se ven tal como el producto de una *inserción*, y hacer de cuenta que son pasibles de ser removidas no hace

más que agregar una de las más costosas *remociones* pero no ocasiona una diferencia global importante en el cálculo. Bajo esta suposición un máximo de $n_1=(N+1)$ *remociones* consecutivas son posibles en todo momento. De todas ellas sólo (n_1-n_2) corresponden a columnas de altura *1*, (n_2-n_3) a columnas de altura 2, y así sucesivamente. El valor medio de la altura columna removida será *R*:

$$R = [1\cdot(n_1-n_2)+2\cdot(n_2-n_3)+ \dots +(h-1)\cdot(n_{h-1}-n_h)+h\cdot(n_h)] / n_1$$
$$R = [n_1+n_2+n_3+ \dots + n_{h-1}+n_h] / n_1$$
$$R = [1+(s+1)^{-1}+(s+1)^{-2}+\dots+(s+1)^{-(h-1)}+(s+1)^{-h}]$$

El valor de *h* crece junto con el de *N* de manera que:

$$lim_{N\to\infty} R = lim_{h\to\infty} R = (s+1)/s$$

Sigue entonces que el costo promedio de las *remociones* es ***O(1)***. En una larga secuencia de *remociones* la altura media de la columna eliminada no depende del tamaño de la lista de *eventos*. Este hecho se muestra en **FIGURA 3.19** para un extenso rango de tamaños (un rango todavía más extenso que el considerado en el resto de los experimentos de esta Tesis, precisamente para resaltar la citada independencia).

Queda claro también que $R_{max} = h = log_{s+1}(N+1)$, significando ello que el peor caso para las *remociones* tiene un costo ***O(log N)***.

En implementaciones reales la cantidad de trabajo requerida para *remover nodos* de la lista de *eventos* es diferente de la cantidad de trabajo que demanda la *remoción* de *nodos auxiliares* debido a que:

1) Los *nodos auxiliares* a *remover* deben ser detectados mediante comparaciones inevitables en recorridos de abajo hacia arriba.

2) Los *nodos auxiliares* deben desalocarse (devolverse a memoria) mientras que los *nodos* de la lista de *eventos* son simplemente devueltos al *simulador*.

Por estas razones el trabajo necesario para borrar una columna completa no será directamente proporcional a su altura. Supóngase que el costo para *remover* un *nodo auxiliar* sigue siendo *1* (una unidad) mientras que el costo para extraer un *nodo* de la lista de *eventos* es δ (aceptando $\delta<1$). Entonces la expresión inicial de R se transformará en:

$$R = [\delta\cdot(n_1-n_2)+(\delta+1)\cdot(n_2-n_3)+ \dots +(\delta+h-2)\cdot(n_{h-1}-n_h)+(\delta+h-1)\cdot(n_h)] / n_1$$

Y consecuentemente la expresión final $(s+1)/s$ en $(\delta s+1)/s$, pero el carácter ***O(1)*** no se ve afectado.

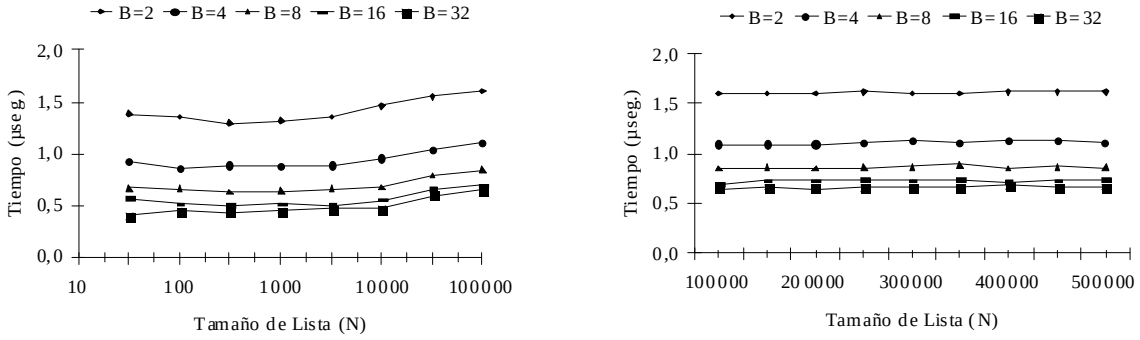


FIGURA 3.19 Tiempo de Remoción de *Linked Exact Bottom Up* - BSS

Ahora es posible retomar y completar la determinación de complejidad de la *inserción*. Las Inserciones Auxiliares son un proceso iterativo que consiste en una suerte de "ascenso" a través de la estructura, de modo que cada paso hacia arriba va precedido de una comparación entre el valor de span leído y el máximo valor de span permitido. Según sea el resultado de esa comparación caben dos posibilidades,

1) Se enlaza un nuevo *nodo auxiliar* y se asciende al nivel superior para realizar una nueva comparación.

2) Termina el proceso.

Luego, la cantidad máxima de comparaciones realizadas en las Inserciones Auxiliares será *h* (la cantidad total de niveles). Pero en general ese número estará entre *1* y *h*, dependiendo del "camino de ascenso" y por ende de los valores de span leídos. Para la determinación de la cantidad media de comparaciones realizadas durante el ascenso se propone lo siguiente: asumiendo que la estructura se construye, es decir "crece hacia arriba",

exclusivamente a partir de las Inserciones Auxiliares, cabe esperar que para una estructura suficientemente llena (*N* grande), en valores medios habrá tantas Inserciones Auxiliares que suban hasta una altura *z* ($1 \leq z \leq h$) como "columnas" de altura *z* existan al momento de ejecutarse el proceso. En rigor las Inserciones Auxiliares no construyen "columnas" ya que no necesariamente cada nuevo *nodo auxiliar* se enlaza por sobre el último insertado, pero desde el punto de vista de la cantidad de *nodos auxiliares* agregados no tiene importancia el punto de *inserción* sino el número de niveles recorridos en el ascenso.

La altura promedio de las columnas fue determinada en el análisis de complejidad de la *remoción*, y para la misma se determinó su carácter independiente de *N*. Finalmente resulta que el proceso de Inserciones Auxiliares tiene un costo medio $\boldsymbol{O}(1)$ y no ejerce influencia en el valor medio de complejidad de la *inserción*, el cual se corresponde únicamente con el de la Búsqueda en Descenso, es decir $\boldsymbol{O}(\log N)$. El mejor caso se corresponde con Inserciones Auxiliares que no agregan ningún *nodo auxiliar*, completándose por lo tanto al cabo de la primer comparación. Eso les da un carácter independiente de *N*, no alterando el carácter $\boldsymbol{O}(\log N)$ para el mejor caso de la Búsqueda en Descenso. En cuanto al peor caso es claro que las Inserciones Auxiliares realizarán *h* comparaciones. Atendiendo al hecho que el peor caso de Búsqueda en Descenso fue determinado como $(B+1)\cdot h$, el peor caso para el proceso completo de *inserción* resultará $(B+2)\cdot h$, es decir $\boldsymbol{O}(\log N)$.

3.3.7 Pautas de Implementación de *Linked Exact Bottom Up* - BSS

Linked Exact Bottom Up - BSS fue concebida como *cola de prioridad* para alojar al lista de *eventos* pendientes en SED y como tal fue desarrollada en el lenguaje C de programación. Seguidamente se describen las características principales del programa que sirvió de soporte al presente trabajo.

3.3.7.1. Tipos de Nodo

La principal dificultad del diseño tiene que ver con la elección de tipo para los *nodos*. A fin de decidir el más adecuado para su estructura considérese primeramente la información que los mismos deben registrar. Por un lado están los *nodos* del nivel más bajo de la estructura, que deben alojar información vinculada al *evento* al que están asociados (por ejemplo una referencia a la rutina que debe ejecutarse cuando llega el tiempo que tienen consignados, un valor de clase, etc.) y por lado otro están los *nodos* de carácter estrictamente *auxiliar*, es decir los que a partir de la lista de *eventos* hacen crecer la estructura hacia arriba y que sólo sirven para acelerar las búsquedas. Designando "*e*" a los primeros y "*a*" a los últimos, los de tipo *e* deberán tener una estructura como la siguiente:

$$e: \{time(f), next(p_e), event\}$$

donde *time* es un campo de tipo punto flotante (o aquel que el usuario elija) destinado a alojar el tiempo de ocurrencia o planificación del *evento* asociado, *next* es un puntero al *nodo* siguiente, o sea un campo de tipo puntero a *e*, y *event* es un campo o conjunto de campos vinculados estrictamente al proceso a ejecutarse al momento de ocurrir el *evento* asociado al *nodo*. Los *nodos* tipo *a* deberán estructurarse alojando la siguiente información:

$$a: \{time(f), next(p_a), down(?), span(i)\}$$

Siendo *time* y *next* los equivalentes a los mismos campos en los *nodos* tipo *e*, *down* un puntero al *nodo* inmediatamente debajo y *span* un valor entero para registrar la longitud del segmento del nivel inferior del cual el *nodo a* es su extremo derecho.

A partir de estas estructuras resultan evidentes las cuestiones a resolver para poder programar algoritmos que operen sobre *Linked Exact Bottom Up* - BSS. La principal dificultad está en la elección de tipo para el puntero *down* y en el diseño de los algoritmos consecuentes con dicha elección. Tres alternativas fueron implementadas y ensayadas.

3.3.7.1.1. Nodos de un único tipo

Si no existe inconveniente en cuanto a cierto desperdicio de memoria es posible que los *nodos e* y *a* sean del mismo tipo. La solución más sencilla para esta alternativa consiste en que el tipo único (*u*) permita alojar los siguientes campos:

$$u: \{time(f), next(p_u), down(p_u), span(i), event\}$$

Podría suceder también que *event* corresponda a un entero (por ejemplo un valor codificado), en cuyo caso los *nodos* se convertirían en:

$$u: \{time(f), next(p_u), down(p_u), value(i)\}$$

Donde *value* hace las veces *span* o de *event*, según *u* esté actuando como *a* o como *e*. El único campo desperdiciado es *down* cuando los *nodos* hacen el papel de *e*. Este problema admite una solución interesante en el lenguaje C. Asumiendo que los *nodos* son estructuras y los campos miembros de las mismas, la idea es

declarar un arreglo de punteros cuando se necesita un número variable de ellos. Ese arreglo puede figurar en la declaración como de dimensión *0* (cero) y llegado el momento de alocar cada *nodo*, de acuerdo al número de punteros que se necesiten se elige el índice del arreglo del nuevo *nodo*. En ese caso, una precaución a tener es que el arreglo deberá ser el último de la lista de miembros de la estructura. Los *nodos* quedan finalmente.

$$u: \{time(f)\ ,\ value(i),\ link[n](p_u)\}$$

Esta solución cubre además el problema del *Header*, que en el caso de las colas *auxiliares* debe alojar también la dimensión de la misma, dato que puede ser absorbido por *value*, y también los punteros *trail* y *up* que pueden obtenerse como elementos de *link[n]*.

3.3.7.1.2. Nodos tipo *a* sin campo time

Considerando dos tipos de *nodo* diferente, o sea los mencionados *e* y *a*, existe una solución que consiste en eliminar del tipo *a* el campo *time* (como valor flotante) y en su reemplazo colocar un puntero hacia el tipo *e*.

$$e: \{time(f),\ next(p_e),\ event\}$$

$$a: \{time(p_e),\ link[n](p_a),\ span(i)\}$$

El puntero *time* de cada *nodo* tipo *a* debe apuntar al más bajo de los *nodos* de la columna a la que pertenece, que desde luego es de tipo *e*. La lectura de tiempo en los *nodos auxiliares* ya no es directa sino indirecta. Por otro lado es necesario que todos los *nodos auxiliares* tengan acceso al que se encuentra inmediatamente debajo, sea éste otro *auxiliar* (tipo *a*) o uno asociado a *evento* (tipo *e*). Ese acceso se puede lograr mediante un elemento de *link[n]* en todos los *nodos* que tengan un *auxiliar* debajo de sí, sólo que en el caso de la primer cola de *auxiliares*, es decir en la que está por encima de la lista de *eventos*, ese elemento de *link[n]* no es útil ya que se trata de un puntero de tipo p_a y lo que hace falta es un puntero de tipo p_e. Únicamente en ese caso el acceso se logra a través del puntero *time*. Es decir, en la cola *auxiliar* más baja, el puntero *time* cobra una utilidad doble. El resultado logrado es que ningún *nodo* carga con campos innecesarios. Si se dejara el campo *time* como valor de tiempo en *a*, cada *nodo auxiliar* debería contar con dos punteros, por ejemplo *event*(p_e) y *auxiliary*(p_a), pero lógicamente los que utilizaran uno no harían lo propio con el otro y viceversa.

3.3.7.1.3. Punteros void

La última de las variantes propuestas y ensayadas consiste en implementar los dos tipos de *nodo*, *e* y *a*, tal como fueron presentados al comienzo, sólo que dejando indefinido (*void*) el tipo del puntero *down*:

$$e: \{time(f),\ next(p_e),\ event\}$$

$$a: \{time(f),\ next(p_a),\ down(p_{void}),\ span(i)\}$$

Eventualmente el puntero *next* en *a* puede ser un arreglo de punteros que permita obtener, a partir de él, el enlace hacia el *nodo* siguiente y en el caso de los *Headers* los punteros *trail* y *up*. En esta opción las rutinas iterativas deben encargarse de asignar el tipo que corresponda cuando se realizan descensos a través de los enlaces descendentes. El puntero *down* asumirá el tipo p_a en los primeros pasos del descenso y el tipo p_e en el último. Esto complica los algoritmos haciendo que, en general, lazos de la forma:

```
while(...)
{
  ...
  ...
}
```

se deban transformar en otros de la forma:

```
while(...)
{
  ...
  ...
}
if(...)
{
  ...
  ...
}
```

de modo de poder aislar la última iteración para hacer una asignación de tipo distinta a la del resto del lazo. (Otros algoritmos presentan la necesidad de aislar la primera iteración, cosa que se resuelve de la forma opuesta).

Además de ofrecer esta opción (al igual que la de Nodos tipo *a* sin campo *time*) una total independencia entre los *nodos auxiliares* y los asociados a *eventos*, se comprueba experimentalmente que es la que exhibe mejor rendimiento en cuanto a tiempo de ejecución.

3.3.7.2. Búsqueda en descenso

El *Header* del nivel más alto es un punto siempre conocido. Desde él se inician siempre las búsquedas apelando al mecanismo que da el nombre a la estructura. Sea que se busque un sitio para *insertar* un *nodo* nuevo o que se busque un *nodo* existente, hay un valor de *t* contra el cual las *búsquedas secuenciales* realizan comparaciones. Debe existir coherencia entre el código escrito para *inserciones* (*Insert*), *remociones* (*DeleteMin*) y búsquedas (*Find*). Por ejemplo, si se pretende que la estructura resulte *estable* será preciso que todo nuevo *nodo* se aloje a continuación de él o los que pudieran contener su mismo valor de *t*. Las *inserciones* deberán iniciarse entonces con búsquedas que, en todos los niveles, prosigan hacia adelante mientras el valor de *t* del *nodo* a *insertar* sea "mayor o igual" que el de los *nodos* que se van visitando. Luego, a la hora de buscar o *remover nodos* se deberá tener en cuenta la forma en que fueron codificadas las *inserciones* para saber, en caso de *nodos* con valor repetido de *t*, cuál de ellos es el que interesa ubicar y en función de ese interés decidir si las comparaciones deben ser por "mayor" o "mayor o igual".

En todos los casos las búsquedas avanzan mientras les es posible, luego de lo cual descienden al nivel inferior para proseguir. Tanto para *inserciones* como para *remociones* es necesario que se conserve la posición del *nodo* por el cual se descendió hasta el nivel inferior. Un puntero desde el *Header* de cada nivel, llamado *trail*, se encarga de esa tarea. La utilidad de ese puntero se ve en los siguientes procesos.

3.3.7.3 Inserciones Auxiliares

Ante una *inserción*, en cualquier nivel, la relación $\Re$ puede verse afectada si el span del segmento modificado supera el valor de *B*. En ese caso es necesario practicar una *inserción* en el lugar adecuado del nivel superior. Luego de esa segunda *inserción* podría ser necesario una tercera, por idéntica razón, y así sucesivamente. El apuntamiento de los punteros *trail* permite conocer rápidamente el segmento de cada nivel de la estructura en el que es potencialmente necesario *insertar* un *nodo auxiliar*.

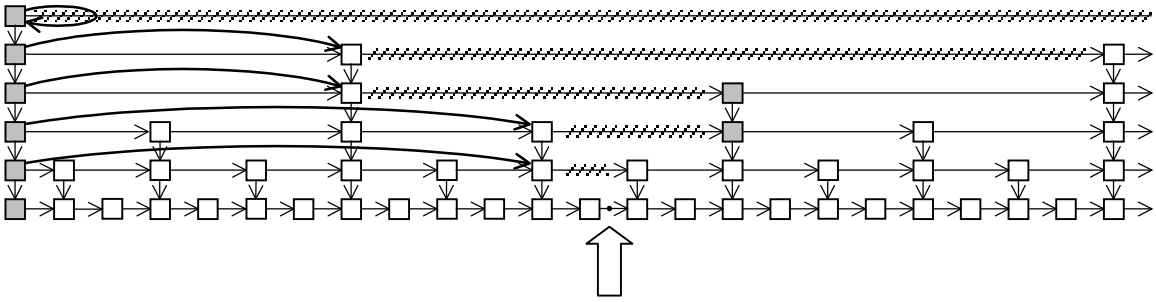


FIGURA 3.20 Distribución de punteros *trail*

FIGURA 3.20 muestra la distribución de los *trail* para una *inserción* en el punto indicado con una flecha grande vertical. El proceso de Inserciones Auxiliares se inicia en caso que el segmento sombreado del nivel que sigue al más bajo necesite alojar un nuevo *nodo*. Continúa del mismo modo en forma ascendente y ni bien arriba a un nivel que no necesita *inserción*, el proceso termina. Los candidatos a alojar nuevos *nodos* son los segmentos sombreados, fácilmente determinables a partir de los *trails* apuntados durante la búsqueda en descenso.

Cada *inserción* involucra tres niveles consecutivos, denominados de abajo hacia arriba: *X*, *Y* y *Z*. En el caso más general, luego de que una *inserción* lleva la longitud de algún segmento de *X* a superar el valor de *B*, un *nodo auxiliar* debe *insertarse* en el lugar apropiado de *Y*, luego de lo cual la longitud del segmento de *Y* que se acaba de modificar (que está guardada en un *nodo* de *Z*) se debe actualizar. Sólo *X* es el nivel que siempre participa del proceso. Puede suceder que únicamente exista *Y* por encima pero no *Z*, o que no existan ni *Y* ni *Z*. Se habla entonces de Inserciones Auxiliares de tres tipos distintos, con su significado obvio: *XYZ*, *XY* y *X*.

Las dos primeras son bastante parecidas, sólo difieren en la actualización de un valor en tanto la última es diferente por cuanto implica la creación de un nuevo nivel en la estructura (*Y*), conteniendo el *nodo auxiliar* a

insertar. Por razones de implementación es útil programar las tres por separado. El algoritmo que comanda el proceso sigue un camino ascendente y va llamando entonces, en función del nivel que se encuentra visitando, a la función que corresponde. **FIGURA 3.21** presenta un algoritmo que implementa el mecanismo comentado.

Los punteros *up* permiten ascender por la columna de *Header*s y los *next* desplazarse hacia el *nodo* consecutivo en cualquiera de los niveles. Las funciones *add_XYZ*, *add_XY* y *add_X* implementan las *inserciones* propiamente dichas de acuerdo a lo explicado más arriba (por simplicidad se han omitido sus argumentos).

```
Y = X→up;
if(Y){
  ++(Y→trail→next→span);
  Z = Y→up;
  while(Z){
    if(Y→trail→next→span > B) add_XYZ;
    else return;
    X = X→up;
    Y = Y→up;
    Z = Z→up;
  }
  if(Y→trail→next→span > B) add_XY;
  else return;
  X = X→up;
}
if(X→SIZE > B) add_X;
return;
```

FIGURA 3.21 Algoritmo de Inserciones Auxiliares

Se ingresa a éste código una vez que algún segmento del nivel *X* ha superado el valor de *B*, y se sale de él cuando se arriba a un nivel en el que ya no es necesario *insertar*. El caso más simple de aplicación de Inserciones Auxiliares es la función *Insert()*, en la cual el algoritmo de la **FIGURA 3.21** se ejecuta ni bien se ha alojado un nuevo *nodo* en el nivel más bajo de la estructura.

3.3.7.4. Reestructuración post-remoción

Sea la función *Delete()* concebida para retirar de la estructura, y devolver al *simulador*, el *nodo* con valor de tiempo t_r, tal que si no existe devuelva NULL y si existe más de uno opere respetando el orden FIFO. El proceso se inicia con una Búsqueda en Descenso hasta localizar el *nodo* a *remover*. Durante el mismo quedará determinado un conjunto de "columnas" sujetas a *remoción*. Se trata de las columnas constituidas por los *nodos* apuntados desde los *nodos* apuntados por cada puntero *trail*.

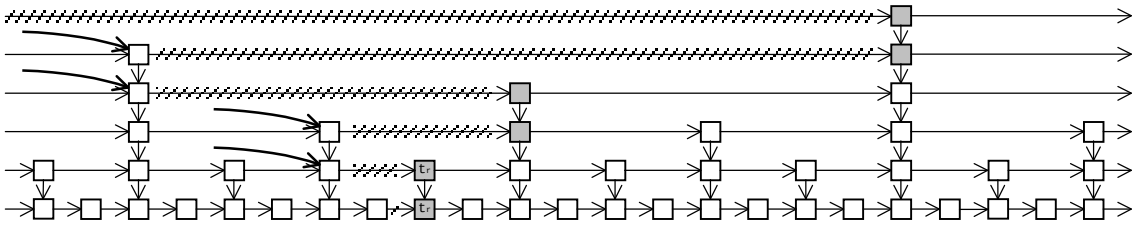


FIGURA 3.22 Determinación de Columnas a Remover

En la **FIGURA 3.22** las columnas sujetas a *remoción* se muestran en gris y los enlaces desde los *trails* correspondientes se muestran sombreados. En caso de existir, el mecanismo comienza removiendo la columna de *nodos* con t_r. Producto de esa *remoción* se deben realizar las siguientes correcciones de valor de span:

1) Un incremento en los *nodos* que están a continuación de cada *nodo auxiliar* removido.
2) Un decremento, en uno, en el *nodo* más bajo de la columna siguiente a la que se acaba de *remover* (la que le sigue hacia la derecha).

Allí termina el proceso de *remoción* propiamente dicho y podría terminar la función si no existiera interés en mantener la relación ℜ de la estructura. Pero de existir ese interés, el proceso debe continuar por:

1) Verificar si el valor de span de alguno de los segmentos que percibió un incremento superó el límite *B*.

2) Verificar si el valor de span del segmento que percibió un decremento cayó por debajo del límite *(B/2)*. Pero sólo después de haber aplicado 1).

Si a partir de 1) resulta la necesidad de nuevas *inserciones*, éstas se resuelven mediante la rutina de Inserciones Auxiliares. Esto ofrece la posibilidad de restituir una columna (o parte de ella) en reemplazo de la que se acaba de *remover*, sólo que en una posición ligeramente desplazada. Es simple ver que basta con la restitución de no más de un *nodo* por nivel. Para ello considérese el peor caso, en el que los dos segmentos fusionados hayan tenido un span de *B*. En ese caso la primer reinserción genera dos nuevos segmentos de longitud *B* y *(B-1)* respectivamente, y con eso basta. Si la nueva columna (es decir la columna reinsertada) tuviese la misma altura que la removida, se cancelaría el decremento practicado en la columna siguiente, el punto 2) quedaría sin efecto y el algoritmo terminaría. Pero si la altura de la nueva columna fuese menor que la de la columna original se deberá resolver el problema de la columna siguiente, cuyo *nodo* más bajo redujo su valor de span en una unidad. Si no obstante esa disminución, esa columna sigue teniendo un valor de span admisible según ℜ el algoritmo termina, pero si su valor de span hubiese caído por debajo de *(B/2)* la columna completa se remueve e inmediatamente se vuelve a aplicar el algoritmo de Inserciones Auxiliares, ofreciendo la posibilidad de reemplazarla, total o parcialmente, pero desde luego en una nueva posición. Lamentablemente es inevitable la necesidad de asegurarse que la columna a *remover* no es la de *Rear*. Si el proceso avanza hasta ese punto, la reestructuración se debe abortar quedando algún(os) segmento(s) sin ajustar. También es simple ver que para reestructurar en este caso basta con una única *inserción*, ya que si se resolvió la eliminación de una columna es porque el span de su *nodo* más bajo era estrictamente menor que *(B/2)*, y el siguiente, con el que se va a fusionar, podrá ser a lo sumo *B*. La partición *de (B/2)+B* se logra mediante un único *nodo*.

En definitiva la Reestructuración post-*remoción* es un proceso iterativo que consiste en: *"remover una columna, reemplazarla total o parcialmente, en una posición cercana, chequear el nodo más bajo de la columna siguiente y eventualmente removerla, ..."*. Este mecanismo reacomoda los *nodos auxiliares* y redistribuye las columnas procediendo desde abajo hacia arriba y de izquierda a derecha, a partir de los puntos señalados por el puntero *trail* de cada nivel. Cuanto más cerca del *Rear* se encuentre el *nodo* a *remover*, tanto menos espacio disponible habrá para la ejecución del proceso.

La reestructuración post-*remoción* es indispensable luego de la eliminación de *nodos* interiores, pero su aplicación también es factible inmediatamente luego de la función *DeleteMin()*, que no es sino un caso particular de *Delete()*. A priori resulta evidente que la reestructuración implica un costo adicional para la *remoción* del primer *nodo*, pero no menos evidente resulta que la configuración resultante de su aplicación deja como saldo una distribución más adecuada de la estructura *auxiliar*, favoreciendo el rendimiento de futuras *inserciones*. Experimentalmente se comprueba que la operación *DeleteMin()* tiene un rendimiento prácticamente idéntico sea que se ejecute seguida del proceso de reestructuración o no. De esto se desprenden dos consideraciones. La primera es que dado que el código del proceso de reestructuración es extremadamente largo, es preferible obviarlo en la función *DeleteMin()*. Y la segunda es que esta determinación experimental garantiza la efectividad del proceso de reestructuración por cuanto el deterioro de la estructura es todavía mayor si se remueve, sin reestructurar, todo a lo largo de la cola que si se lo hace sólo sobre el primer elemento. Y dado que reestructurando al *remover* el primer *nodo* el rendimiento no se altera, tanto menos habrá de alterarse ante *remociones* intermedias.

3.3.7.5. Eliminación del nivel más alto

En *Linked Exact Bottom Up* - BSS el número de *nodos* de las colas *auxiliares* es tanto menor cuanto más alto es el nivel de la cola. Es probable que a causa de *remociones* en cualquier sector, en algún momento se vacíe la cola del nivel más alto. Entonces se pueden tomar dos caminos, uno es mantener niveles vacíos a la espera de que futuras *inserciones* comiencen a ocuparlos nuevamente y otro es *removerlos* de la estructura. Mantenerlos vacíos tiene la ventaja de que pueden ahorrarse algunas aplicaciones de la función *add_X*, ejecutando en su lugar *add_XY* llegado el momento de futuras *inserciones*, pero tiene la desventaja de que hay que mantener un puntero al *Header* del último nivel ocupado. En materia de comparaciones las dos opciones son prácticamente equivalentes, y en materia de rendimiento también. Se optó entonces por la alternativa de quitar todo nivel que quede vacío lo que ofrece la posibilidad, llegado el caso, de destruir la estructura completa en menor tiempo.

3.3.7.6. Determinación del máximo

La estructura *auxiliar* debe ser lo más versátil posible para no incrementar los tiempos de las operaciones sobre *Linked Exact Bottom Up* - BSS. Para ello conviene que:

1) Los *nodos auxiliares* alojen la menor cantidad posible de información.

2) Los enlaces entre *nodos* de un mismo nivel sean simples y no dobles.

Las operaciones que requieren encontrar el *nodo* con mayor valor de *t* (*FindMax()* y *DeleteMax()*) se ven considerablemente afectadas por la regla 2). Si los enlaces fuesen dobles, el *nodo* de mayor valor de *t* sería inmediatamente alcanzable desde el *Rear* del nivel más bajo (que es una posición siempre conocida de la estructura) y su desconexión también sería inmediata. Pero si los enlaces son simples hay que apelar a un mecanismo alternativo. Se puede comenzar realizando la búsqueda en descenso de un *nodo* con valor "infinito" de *t* (el *Rear*), ya que se vio que la búsqueda en descenso conduce hasta el *nodo* inmediato anterior al buscado. En esta búsqueda se debe avanzar en función de comparaciones contra el valor de *t* del *nodo* visitado (sea éste *x*), y se debe decidir el paso hacia el siguiente *nodo*, es decir $x \rightarrow next$, de acuerdo al valor de *t* del mismo y su relación con el t_{Rear}. Un lazo como el siguiente podría comandar la búsqueda en cada nivel:

$$while(x \rightarrow next \rightarrow time < t_{Rear})\ \ x = x \rightarrow next;$$

Cuando la misma ha concluido en el nivel más bajo, *x* está sobre el *nodo* de más alto valor de *t*. Ahora bien, pretender además desconectarlo para devolverlo es más complicado porque para ello es preciso ubicar su antecesor. El lazo anterior debe modificarse entonces, de modo que en todos los niveles la comparación no intente determinar si el *nodo* siguiente es el *Rear*, sino si el siguiente del siguiente lo es.

$$while(x \rightarrow next \rightarrow next \rightarrow time < t_{Rear})\ \ x = x \rightarrow next;$$

Esta opción contempla también el caso en que el *nodo* de más alto valor de *t* sea una columna de *nodos*.

3.3.7.7. Administración de Nodos Auxiliares

La gestión de *nodos auxiliares* hace que *Linked Exact Bottom Up* - BSS deba operar permanentemente en acceso dinámico a memoria [3]. La SED, en general, trabaja en esta modalidad pero *Linked Exact Bottom Up* - BSS lo hace mucho más, por lo que se implementó un mecanismo tendiente a acelerar esas operaciones. En lugar de solicitar y devolver los *nodos auxiliares* directamente al sistema mediante las funciones que para tal fin provee el lenguaje (*malloc()* y *free()* del lenguaje C en nuestro caso) se reemplazaron esas funciones por otras que:

1) Toman los *nodos auxiliares* de una pila, en lugar de hacerlo del sistema. Si la pila está vacía, entonces los toman del sistema.

2) Devuelven siempre los *nodos auxiliares* a la pila.

A partir del momento en que *Linked Exact Bottom Up* - BSS ha alcanzado su máximo tamaño (independientemente del modelo simulado) ya no se solicitarán *nodos auxiliares* al sistema. Esta facilidad constituye un importante aporte al rendimiento, principalmente en sistemas en los que la memoria es compartida por distintos usuarios.

3.4. Evaluación Experimental de las Implementaciones BSS

En esta Sección se vuelve sobre las siete variantes de implementación del mecanismo BSS a fin de establecer entre ellas una valuación de carácter comparativo.

Las **FIGURAS 3.23** y **3.24** muestran los resultados de una serie de ensayos correspondientes a la medición de rendimiento, evaluado como el tiempo de ejecución, de una simulación en la que cada implementación desempeña el papel del *calendario* y el modelo simulado es *The Hold Model* con distribución Exponencial (ver Sección **5**). El primer ensayo (I) corresponde a un rango de número de *eventos* "bajo" (hasta unas centenas) y el segundo (I) a un número "alto" (hasta unos cientos de miles). A pesar del cambio de rango existe una marcada tendencia a favor de tres de las siete implementaciones, son ellas: *Linked Exact Bottom Up* en (I) y (II), y las dos que operan sobre *Array* en (II).

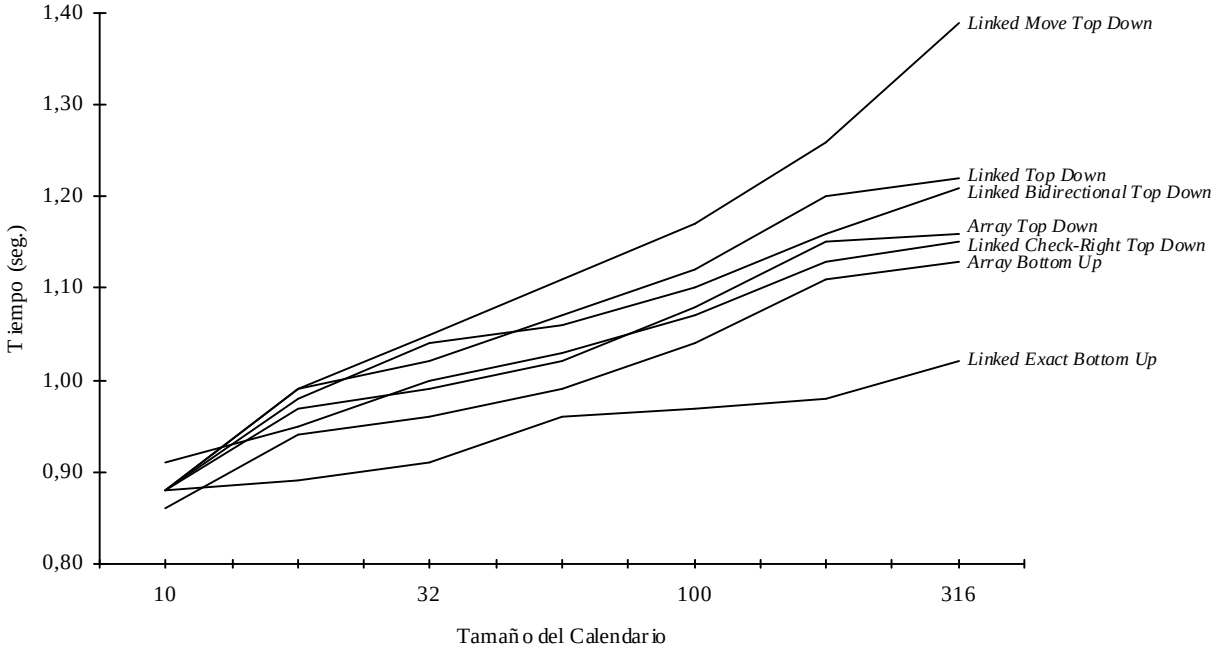


FIGURA 3.23 Implementaciones de tipo BSS (I)

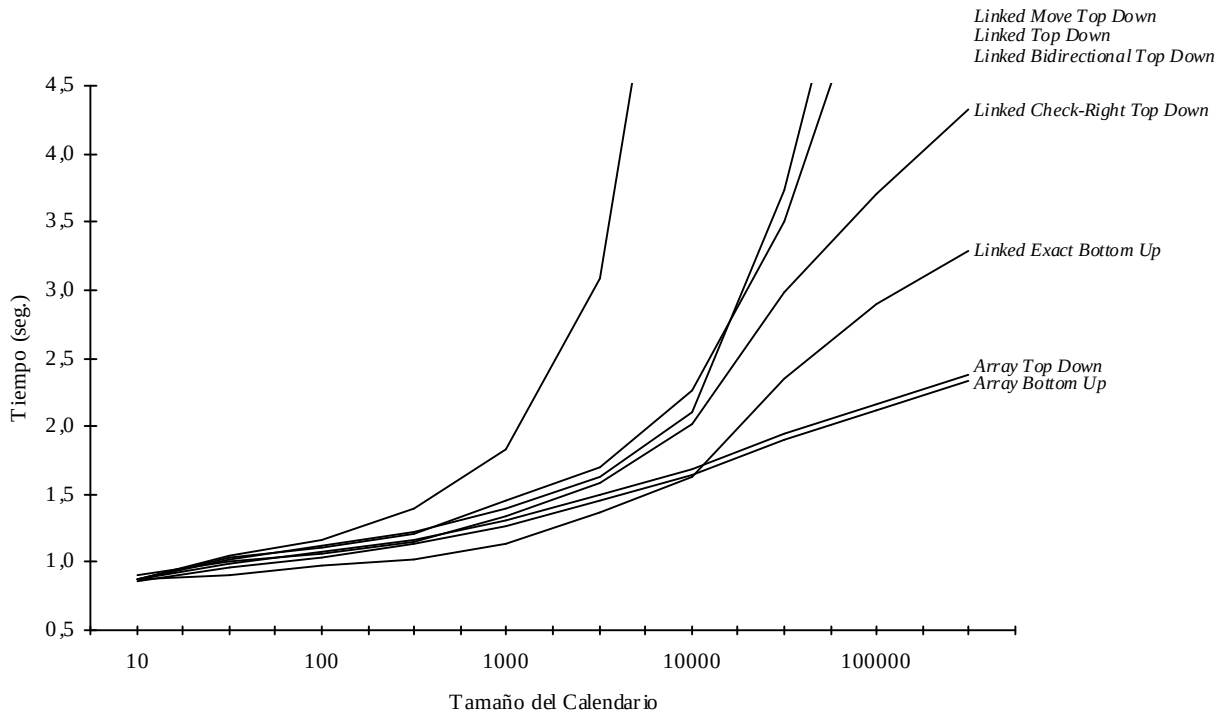


FIGURA 3.24 Implementaciones de tipo BSS (II)

3.5. Conclusiones

En la Sección **3** se introdujo el mecanismo BSS para acelerar búsquedas secuenciales sobre listas y a partir de esa idea se propusieron siete implementaciones de *calendario*. De todas ellas *Linked Exact Bottom Up* fue analizada en detalle; se determinó la complejidad de sus operaciones, se evaluó experimentalmente su comportamiento y se reportaron pautas de implementación. Por último se realizó un ensayo comparativo entre las siete implementaciones del cual se deduce que *Linked Exact Bottom Up* y las dos variantes sobre *Array* son las más eficientes para todo rango de número de *eventos*.

Dado el buen compromiso entre eficiencia y simplicidad algunas de las implementaciones de tipo BSS resuelven muy bien las necesidades de un *calendario* de propósito general en la SED. Si las comparaciones son adecuadamente programadas las implementaciones resultan *estables*. Aquellas que cuentan con un peor caso bien acotado tanto para *inserciones* como para *remociones* (mejor que ***O****(N)*) resultan apropiadas para la planificación de *eventos* en sistemas de tiempo real, son estas *Linked Exact Bottom Up* y las dos variantes sobre *Array*.

Aún cuando todas las implementaciones necesitan que se les fije el valor de *B* como parámetro de operación (más allá de que no tenga el mismo significado en todas), los resultados experimentales demuestran que puede ser fijado al mismo valor para un extenso rango de tamaños de lista, sin perder eficiencia. En cierto sentido, esta propiedad recuerda a la famosa implementación de *Henriksen* [9] y su parámetro "tamaño máximo de sub-lista" habitualmente fijado en *4*.

En general los algoritmos de BSS son simples de programar, no son necesarias operaciones de redimensionamiento, no hay necesidad de división en sectores y menos aún de tratamiento especial para desborde (*overflow*). Los mecanismos son tan claros y simples que admiten el intento de variaciones, con mínimo esfuerzo.

Los resultados experimentales mostrados revelan que hasta 10.000 *eventos* la variante *Linked Exact Bottom Up* es la mas eficiente pero a partir de ese valor las diferencias se vuelcan notablemente a favor de *Array Bottom Up* y *Array Top Down*.

Una línea de trabajo futuro consiste en intentar ajustar el valor de B en tiempo de ejecución en *Linked Exact Bottom Up* - BSS, de modo de adaptar el span medio a la distribución temporal de los *eventos*. Se comprueba, experimentalmente, que para un valor de *B* fijo el span medio es función de la distribución temporal de los *eventos*. Entonces, sin que se modifique el mecanismo de las operaciones, si en lugar de fijar *B* como parámetro de operación, se fija un valor *S*, algo así como el valor pretendido de span medio y en función de una determinación, realizada en tiempo de ejecución del span medio se fuera corrigiendo el valor de *B* de manera de hacer que el span medio tienda a S, podría lograrse un incremento en la independencia respecto de la distribución temporal de *eventos*. La dos dificultades a resolver son:

1) La determinación del span medio sin que la medición afecte sustancialmente el rendimiento.

2) La política de variación de *B*.

4. Short-Cut List

En esta Sección se presenta una propuesta de implementación de *calendario* que opera sobre lista enlazada. La idea es agilizar las operaciones propias del *calendario* acotando los tiempos de *búsqueda secuencial* sobre lista. Para ello se procura establecer enlaces alternativos de manera que cada *nodo* esté ligado no sólo al *nodo* contiguo sino también a otro más adelantado en el recorrido secuencial. Se introduce una serie de ideas para su puesta en marcha, se comentan las dificultades de su implementación y se sugieren posibles variantes.

4.1. Introducción

Dado el problema de implementar el *calendario* en la SED, se propone un modelo de lista ordenada cuyo sistema de búsqueda presenta ventajas sobre el método secuencial directo. Se trata de una estructura que se implementa, básicamente, sobre una lista ordenada, doblemente enlazada, en la que los *nodos*, además de tener una posición dada por el orden creciente del ítem que representan, pertenecen a distintos niveles. Existen caminos para desplazase exclusivamente a través de *nodos* de un mismo nivel y caminos que permiten transiciones entre niveles.

4.2. Nodos

Además de contar con él o los campos de dato necesarios (*D*) y con un campo indicador del nivel de prioridad dentro de la lista, el modelo de *nodo* a utilizar debe contar con tres punteros para ligarlo a la estructura mediante dos enlaces, uno simple y uno doble (ver **FIGURA 4.1**). A través de los enlaces dobles se ligan *nodos* pertenecientes a distinto nivel y es a través de los cuales es posible recorrerlos todos. Dan origen al camino común (ordinary) para una lista doblemente enlazada, es decir permiten el paso desde cada *nodo*, tanto hacia el siguiente como hacia el anterior, de ahí el nombre de estos enlaces: *OPL - Ordinary Path Link* (Enlace del Camino Común). Los enlaces simples ligan exclusivamente *nodos* de un mismo nivel, permitiendo recorrerlos también en forma ordenada. Se mostrará que los enlaces entre *nodos* de un mismo nivel constituyen atajos (short-cuts) en el recorrido común, de ahí el nombre de los mismos: *SCL - Short-Cut Link* (Enlace del Atajo). En cuanto a la indicación del nivel de prioridad, salvo mención en contrario, de aquí en más se utilizará una unidad de tiempo (*t*). De manera que los *nodos* estarán en relación directa a *eventos* tales que él o los campos *D* indicarán los procesos representativos de su ocurrencia y *t* el instante de la misma.

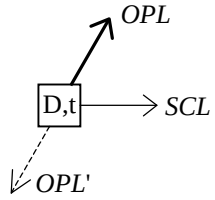


FIGURA 4.1 Tipo de Nodo de Short-Cut List

Aceptando que *OPL* apunta al *nodo* siguiente y *OPL'* al anterior, la utilidad del puntero *OPL'* es menos relevante que la de su contrapartida *OPL* ya que no tiene intervención en el recorrido en sí, sino sólo en un proceso auxiliar, posterior a cada *inserción* (se verá luego). De ahí que se lo haya indicado en un trazo diferente y que, por razones de claridad de los diagramas, no se lo muestre de aquí en más, no obstante lo cual se reconoce siempre el doble enlazado a través de los enlaces *OPL*.

4.3. Construcción

La propuesta consiste en mantener los *nodos* integrados mediante una red de enlaces que permita múltiples recorridos. Como se anticipó, de todos ellos el principal es el asociado a los enlaces *OPL*, y es a través del cual se puede transitar por todos los *nodos*, ordenados según el valor creciente de *t*. Los demás recorridos (en general más de uno) están en correspondencia con los enlaces *SCL*, y permiten desplazarse, respetando el mismo orden, pero sólo a través de *nodos* pertenecientes al mismo nivel. La mecánica de *inserción* de *nodos* se verá en detalle en la Sección **4.7**, de momento sólo se anticipará, a través de algunos ejemplos, la configuración que la lista debe adoptar a medida que crece.

En principio, para alojar un nuevo *nodo*, dos procesos de *inserción* simultáneos deben llevarse a cabo, uno en el recorrido que, en ese momento, constituyan los enlaces *OPL* y otro en el nivel hacia el cual se vaya a promover el nuevo *nodo*, sea éste un nivel existente o uno a crearse.

Ejemplo1: Sea la estructura de **FIGURA 4.2**, formada por *nodos* cuyo valor de *t* es un entero, tal que 18, 26, 56 y 60 son los valores de cuatro *nodos* existentes, ya enlazados y distribuidos en tres niveles. Estos *nodos* están ligados a la estructura mediante enlaces de los cuales sólo se muestran algunos. La **FIGURA 4.2** presenta entonces la *inserción* de un nuevo *nodo* con *t*=35.

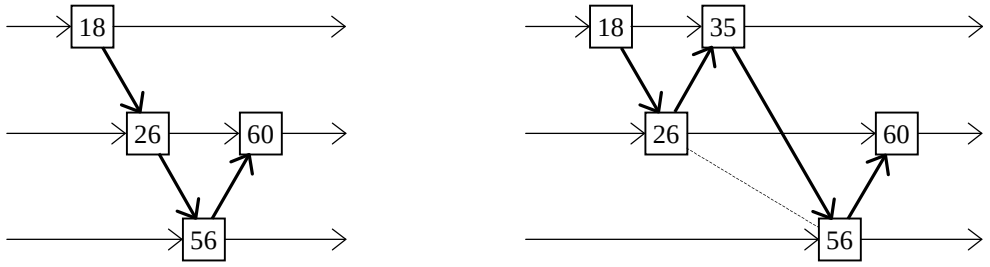


FIGURA 4.2 Inserción de Nodo con *t*=35

Ejemplo 2: Siendo 18, 26, 35 y 44 los valores de *t* de cuatro *nodos* (del mismo tipo que los del ejemplo anterior) existentes y ya enlazados, en la estructura de **FIGURA 4.3** se muestra la *inserción* de un nuevo *nodo* con *t*=39.

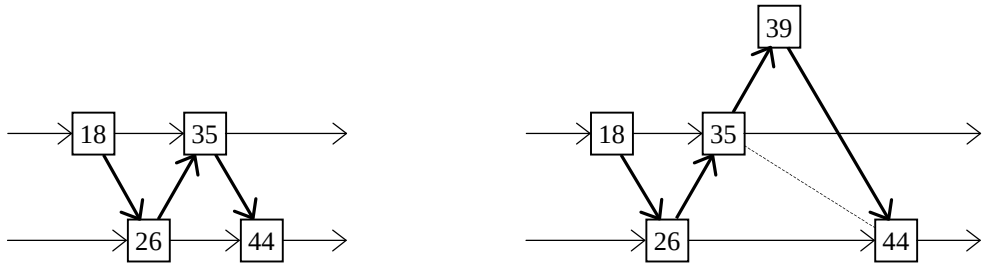


FIGURA 4.3 Inserción de Nodo con *t*=39

El *Ejemplo 1* muestra la *inserción* en un nivel existente, en tanto en el *Ejemplo 2* el *nodo* insertado se promueve hacia un nivel nuevo, creado exclusivamente para alojarlo. Lo interesante es que en ambos casos, luego de la *inserción*, se conserva el orden en todos los recorridos existentes. En el Ejemplo 1 el camino a través de los enlaces *OPL* es -18-26-56-60- antes, y -18-26-35-56-60- después de la *inserción*; mientras que a través de los enlaces *SCL* los recorridos en cada uno de los distintos niveles son -18-, -26-60- y -56- antes, y -18-35-, -26-60- y -56- después. Lo mismo se verifica en el *Ejemplo 2*.

Ejemplo 3: Aún cuando todavía no ha sido presentado el método de construcción propiamente dicho, es interesante, antes de hacerlo, visualizar una estructura (de *nodos* del mismo tipo que el de los ejemplos anteriores) que aloje una lista completa. La configuración, que se muestra en **FIGURA 4.4**, es la resultante de *insertar nodos* recibidos en el siguiente orden (según su valor de *t*): 76, 05, 56, 18, 26, 35, 97, 14, 44, 60, 79, 91, 39 y 68. Lo único a destacar es la presencia de los *nodos* especiales, *H* y *R*, que actúan como cabecera (*Header*) y final (*Rear*) de la estructura. Aunque por razones gráficas *R* se presenta diferente, y no se muestran sus enlaces *OPL* y *SCL*, en realidad tanto *R* como *H* tienen la misma estructura que el resto de los *nodos*.

4.4. Recorrido

Sea para nuevas *inserciones*, para *remociones* o simplemente para búsquedas, el proceso de recorrido de la estructura, hasta dar con un lugar de interés, es el mismo. Se comienza por el *Header*, que deberá contener un

valor de *t* más bajo que el de cualquiera de los *nodos* a *insertar* (por ejemplo un valor negativo, $t<0$, si la prioridad está representada por enteros positivos) y contra él se compara el valor de prioridad del *nodo* a *insertar*, eliminar o buscar, por ejemplo t_x. Mientras t_x es mayor que el *t* de los *nodos* que se van visitando, se continúa transitando hacia el *nodo* siguiente, por el mismo nivel, a través de los enlaces *SCL*. El proceso se detiene cuando t_x resulta menor que el *t* de alguno de los *nodos* visitados, entonces desde el *nodo* anterior se cambia de nivel a través de su enlace *OPL* y se comienza a avanzar por el nuevo nivel hasta encontrar nuevamente un *nodo* con *t* menor a t_x. Dada la estructura de los *nodos*, el cambio de nivel es siempre único y posible, por lo que la transición entre niveles no presenta ambigüedades.

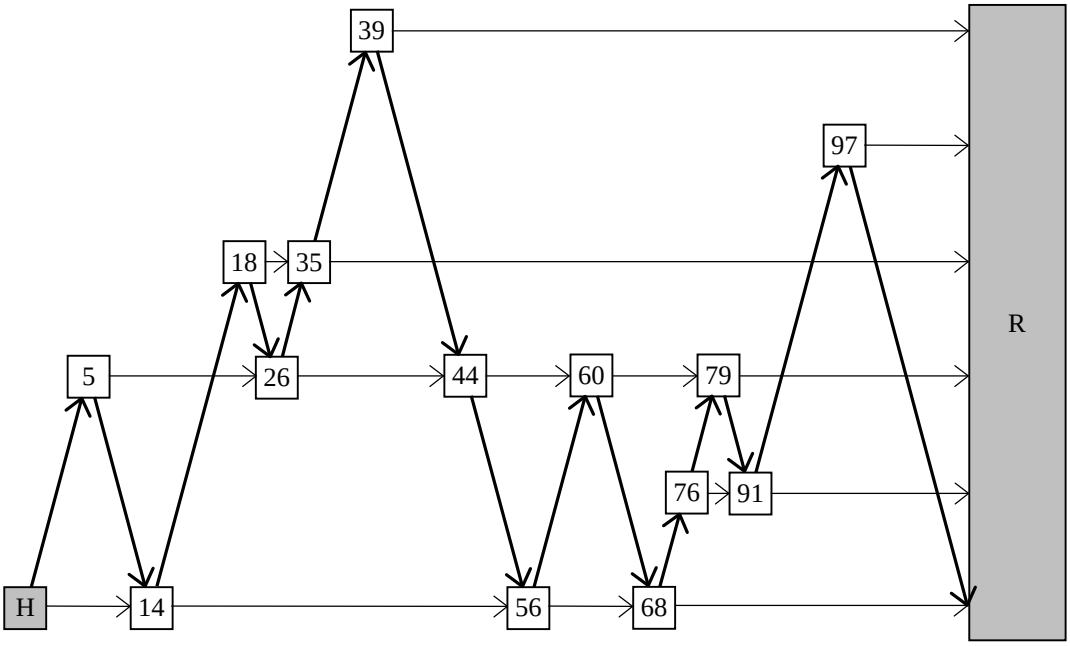


FIGURA 4.4 Short-Cut List Estructura Completa

El *nodo R* deberá alojar un valor de *t* mayor que el de cualquiera de los *nodos* a *insertar* o sea la prioridad más baja posible (por ejemplo el mayor valor que permita el tipo de dato elegido para representar *t*), de este modo los recorridos a través de cualquier nivel tienen siempre un límite más allá del cual no pueden continuar. ¿Cuando se detiene el proceso?, si se trata de una búsqueda o una *remoción* (que primeramente requiere la búsqueda) el mecanismo se mantiene en marcha hasta que en algún momento se hace match entre t_x y el *t* de alguno de los *nodos* visitados, lo cual puede suceder en cualquier punto del recorrido. Pero si se trata de una *inserción* hay que tener más cuidado ya que es preciso encontrar dos *nodos* consecutivos tales que sus valores de *t* sean uno menor y el otro mayor que t_x. Decir "consecutivos" implica que la búsqueda del punto de *inserción* (luego quedará esto demostrado más rigurosamente) tendrá éxito únicamente entre dos *nodos* de distinto nivel, es decir que los *nodos* entre los cuales se ha de *insertar* uno nuevo pertenecerán, necesariamente, a distintos niveles y estarán ligados, por lo tanto, a través de un enlace *OPL*.

Resumiendo, el procedimiento para dar con un punto de interés consiste en intentar avanzar por los enlaces *SCL* mientras resulte posible, luego dar un salto hacia el nivel que indique el enlace *OPL* del *nodo* anterior al último visitado y en el nuevo nivel comenzar a avanzar nuevamente por los enlaces *SCL* mientras resulte posible...

4.5. Remoción

En principio la *remoción* de *nodos* "interiores" y su correspondiente restitución de enlaces no resulta una operación muy directa. Hay casos, por ejemplo, las *remociones* de los *nodos* con valor de *t* 14, 39 y 44, en la **FIGURA 4.4**, que no presentan dificultad. Allí tanto los enlaces *OPL* como los *SCL* se pueden reconstituir manteniendo la continuidad y el orden de los caminos. La sola *remoción* de los *nodos* a eliminar y la posterior reconexión de enlaces para garantizar la continuidad entre los *nodos* que estaban antes y después del eliminado, es todo lo que hay que hacer en ambos recorridos. Pero en cambio, en la misma **FIGURA 4.4**, *remociones* como las de los *nodos* con valor de *t* 26, 56 y 79 presentan inconvenientes. En esos casos el enlace *SCL* se reconstituye

sin problemas, pero no sucede lo mismo con el *OPL* debido a que las *remociones* generarían, como consecuencia, un enlace *OPL* entre *nodos* del mismo nivel, lo que no se corresponde con la idea de construcción de la estructura. Luego, si resultara de interés la *remoción* de *nodos* interiores, deberá implementarse un mecanismo destinado a salvar esa dificultad. De todas maneras, esa situación es poco frecuente en la planificación de *eventos* de la SED, donde las *remociones* tienen lugar, prioritariamente, en un extremo de la lista.

El primer elemento de la lista, es decir el de más alta prioridad, es siempre el *nodo* apuntado por el puntero *OPL* del *Header* de la estructura. Su *remoción* corresponde, directamente, a su reemplazo por el *Header*. Volviendo a la **FIGURA 4.4**, el mecanismo para *remover* el *nodo* con *t*=5 equivaldría a su reemplazo por *H*, es decir a su desconexión completa, y posterior colocación de *H* en su lugar. Una vez concluida la operación, *H* quedará apuntando al *nodo* con valor de *t*=26 a través de su puntero *SCL* y al de *t*=14 a través de *OPL*, pasando por lo tanto a ser 14 el próximo elemento a eliminar. Lo expuesto se muestra en **FIGURA 4.5**.

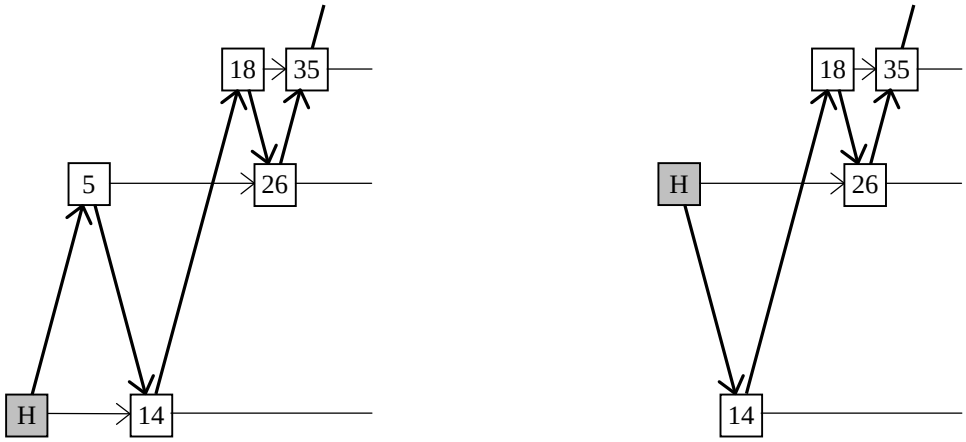


FIGURA 4.5 Remoción en Short-Cut List

No es necesario contar con un registro de niveles, ni con punteros al primer elemento de cada uno. Tampoco con la información del nivel al que pertenece cada *nodo* dentro mismo de su estructura. El único elemento cuya ubicación es menester conocer, es el primero de todos, o sea el de más alta prioridad, y eso es posible a través del *Header* cuya posición es, por supuesto, siempre conocida.

4.6. Interpretación del mecanismo

¿Dónde está la ventaja de este modelo? La estructura debe verse como una lista ordenada doblemente enlazada con un agregado. La lista queda materializada por los enlaces *OPL* y puede visualizarse siguiendo la "línea quebrada" que conforman dichos enlaces (por ejemplo en la estructura de **FIGURA 4.4** la secuencia de *nodos* con valor de *t* 5 14 18 26 35 39...). El agregado consiste en los caminos horizontales (*atajos*) producidos por los enlaces *SCL*. Esos atajos permiten saltear "algunos" de los *nodos* que habría que visitar en las *búsquedas secuenciales* y de hecho dan el nombre a la estructura: *"Short-Cut List "* (Lista con Atajos). Supóngase que en la estructura del **FIGURA 4.4** se quisiera *insertar* un *nodo* con ítem t_x >56. Procediendo sobre el recorrido secuencial completo, es decir del modo tradicional, sería necesario comparar a t_x con 5, 14, 18, 26, 35, 39, 44 y recién entonces con 56. Pero si se aprovecha la facilidad ofrecida por los atajos habría que comparar t_x con 14 e inmediatamente después con 56. Cada salto horizontal a través de un enlace *SCL* (*salto largo*) cubre un número mayor que uno de saltos en el recorrido secuencial clásico (*salto corto*). Dicho en palabras simples, el proceso se compone de secuencias como la siguiente: se da un *salto largo*, si resulta demasiado largo se cancela, se da un *salto corto* y luego se intenta un nuevo *salto largo*, ... Cada vez que un *salto largo* "queda", es decir no se cancela, es muy probable que con él se hayan cubierto varios enlaces al *nodo* siguiente. El principal desafío está en determinar si la estructura se puede poner en marcha mediante una implementación suficientemente simple como para que la ventaja ofrecida por el método prevalezca sobre el incremento en líneas de código respecto a la programación del método de *búsqueda secuencial* tradicional.

4.7. Implementación

Existe una importante dificultad de implementación que a esta altura debería resultar evidente Se dijo que para llevar a cabo una *inserción* es necesario encontrar un par de *nodos* de distinto nivel, ligados por su enlace *OPL*, tales que uno de ellos tenga un valor de *t* más bajo y el otro más alto que el del *nodo* a *insertar*. Una vez llegado a ese punto, la *inserción* sobre el camino *OPL* no presenta dificultad. También se dijo que es preciso *insertar* el nuevo *nodo* en algún nivel, ahora diremos, en algún nivel distinto al de los dos *nodos* (de distinto nivel) entre los que el nuevo hubiere sido colocado. Pero, he aquí la dificultad: en caso de alojarlo en un nivel existente, se deberá conservar el orden en ese nivel inmediatamente luego de la *inserción*. Prescindiendo todavía del criterio para elegir el nuevo nivel, cabe pensar que la única manera de conservar su orden, es mediante un barrido secuencial directo sobre sus enlaces *SCL*, comenzando por el primero, pero ¿no resultará esto computacionalmente muy costoso?. La respuesta es sí, debido a que no existe límite para el crecimiento de la lista y por ende tampoco, en principio, para el crecimiento de cada uno de los niveles. Surge entonces la necesidad de encontrar un método alternativo que evite ese recorrido secuencial en el nivel elegido. La metodología propuesta consiste en proceder al revés, es decir, no en elegir uno de los niveles existentes para luego realizar la *inserción* en él, sino en buscar, mediante algún procedimiento sencillo, un punto de *inserción* fácilmente accesible, adoptando entonces el nivel al que pertenezca ese punto de *inserción* fácilmente accesible. Y si el procedimiento no tuviera éxito, recién entonces proceder a crear un nuevo nivel para colocar el nuevo *nodo*.

A fin de introducir el procedimiento para dar con un punto de *inserción* fácilmente accesible, en algún nivel existente, llámese N_a y N_p a los *nodos anterior* y *posterior* al punto de *inserción*, es decir aquellos entre los cuales la *inserción OPL* ha tenido lugar. El procedimiento consiste en un retroceso (*Back Track*) por el enlace *OPL* hasta el *nodo* anterior a N_a , llámese N_{aa} (retroceder significa seguir la dirección del puntero *OPL'* de N_a). Una vez allí se chequea el valor de *t* del *nodo* apuntado por el enlace *SCL* y se compara su valor de *t* con el de N_p; si resulta más alto, se realiza la *inserción* en ese mismo nivel y a continuación del *nodo* N_{aa}; pero si resulta igual o más bajo, se promueve el *nodo* insertado hacia un nuevo nivel específicamente creado para alojarlo.

Ahora se puede volver sobre las **FIGURAS 4.2** y **4.3** e interpretar las operaciones realizadas. En la **FIGURA 4.2**, el 35 se debe colocar entre 26 y 56, se retrocede entonces desde 26 hasta 18 y se mira desde 18 a través de su enlace *SCL*. Aunque no se muestra en la **FIGURA 4.2**, el valor que se ve desde 18 es mayor que 56, entonces se inserta 35 inmediatamente a continuación de 18, lo cual habiendo "llegado" hasta el 18 es extremadamente simple. En cuanto a la **FIGURA 4.3**, corresponde *insertar* el 39 entre 35 y 44. Al retroceder desde 35 hasta 26 y mirar por su enlace *SCL*, no se ve un valor mayor que 44 sino precisamente el 44, entonces se aloja a 39 en un nivel nuevo.

Otra faceta interesante de la implementación está en resolver el inicio de la lista, fundamentalmente en procura de iniciar la *inserción* de *nodos* mediante el mismo algoritmo que opera cuando la lista ya está ocupada, evitando así la necesidad de preguntar cada vez si el elemento a *insertar* es o no el primero. Para ello es necesario que los enlaces que vinculan *Header* y *Rear* se inicialicen de el modo mostrado en **FIGURA 4.6**.

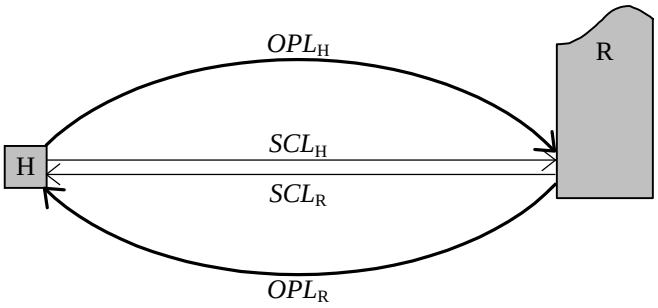


FIGURA 4.6 Short-Cut List Configuración Inicial

La aplicación del algoritmo de *inserción*, en el momento de colocar el primer *nodo*, comenzará avanzando por el enlace SCL_H y deteniéndose inmediatamente al llegar a *R* que es el *nodo* con el valor de *t* más alto posible. Luego se desplazará por el enlace OPL_H (como buscando cambiar de nivel) y concluirá que ése es el enlace a romper para alojar el nuevo *nodo*, ya que es seguro que el valor de *t* del mismo estará entre los de *H* y *R*. Luego retrocederá por OPL_R en procura de encontrar un nivel capaz de recibir al *nodo* a *insertar* y desde *R* chequeará el valor de *t* del *nodo* que vea a través del enlace SCL_R, que no es ni más ni menos que *H*. Concluirá

que no es más alto que el de *R* resolviendo por lo tanto crear un nivel nuevo para alojar el primer *nodo* (**FIGURA 4.7**).

De ahí en más las *inserciones* proceden sin inconvenientes. Los enlaces SCL_R y OPL_R sólo vuelven a ser utilizados cada vez que se deba *insertar* un *nodo* con valor de *t* más bajo que el primero, o sea entre el primero y *H*.

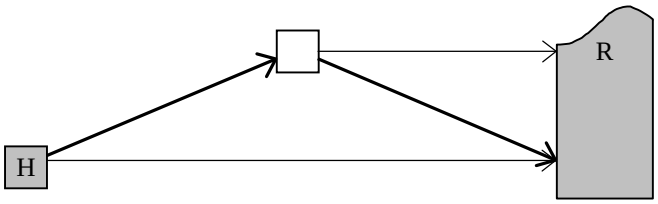


FIGURA 4.7 Inserción del Primer Nodo

Cabe acotar que el *nodo R* no debe considerarse como perteneciente a ningún nivel. Es utilizado como final de lista por todos, pero los algoritmos que operan sobre la estructura pueden funcionar sin necesidad de asociarlo a ninguno.

La forma de resolver la *inserción* de *nodos* con valores de *t* que ya existan en la lista, es decir repeticiones, resulta una consecuencia del algoritmo de *inserción* y determina la estabilidad de la implementación. De acuerdo al mecanismo propuesto los *nodos* repetidos se van a colocar antes del existente, es decir como si tuviesen una prioridad "mayor" (con la misma, en realidad, pero listos para ser removidos antes) resultando entonces una implementación no estable. Hacerlo así es más rápido ya que si hubiera más de una repetición, *insertar* en último lugar implicaría hacer tantas comparaciones como ítems repetidos haya, en tanto hacerlo antes, sólo una. De todas formas la elección de una u otra modalidad implica una modificación mínima en el algoritmo correspondiente y puede dejarse, en última instancia, a criterio del implementador, fundamentalmente si existe interés en lograr una implementación estable.

4.8. Observaciones

Bajo cierto punto de vista los *nodos* ligados a esta estructura pueden considerarse distribuidos en dos dimensiones. Un de ellas dada por su distribución a lo largo de cada nivel (*ancho*) y la otra directamente por el número de niveles (*alto*). Lejos de pretender anticipar, antes de ensayar la estructura propuesta o de estudiarla en mayor profundidad, y en particular discutir cuál será la "forma" ideal para la estructura, una cosa resulta evidente a partir del algoritmo de *inserción*: la elección de un nivel para alojar el nuevo *nodo* se inicia con un retroceso (*Back Track*) a través un enlace *OPL'*. Si ese retroceso no conduce a un nivel adecuado, se crea uno especialmente. Ahora bien, cuando el primer retroceso falla, se podría intentar otro antes de crear un nuevo nivel. Desde luego esta última opción haría descender la altura (y crecer el ancho) de la lista, puesto que aumentaría la probabilidad de éxito para hallar un nivel apropiado entre los existentes y descendería, consecuentemente, la probabilidad de creación de niveles nuevos. Lo mismo podría decirse si en lugar de dos retrocesos se intentaran tres, y así sucesivamente. Este podría ser entonces un parámetro cuya elección permita ensayar una misma lista en distintas conformaciones. El único cuidado a tener está en el hecho de que si se permite más de un retroceso, la *inserción* de un *nodo* antes del primero, o sea con la prioridad más alta, podría conducir la búsqueda de nivel hacia el final de la lista (a través del enlace OPL_R), es decir a la zona de *nodos* con la prioridad más baja, siendo esto absolutamente incorrecto. Luego, si el número de retrocesos permitidos es mayor que uno, es preciso agregar una comparación que permita saber si se ha saltado desde el comienzo hacia el final de la lista. Resultados experimentales demuestran que la respuesta más eficiente se corresponde con una única operación de retroceso (*Back Track*) por *inserción*.

Otra pauta que merece consideración tiene que ver con el algoritmo de búsqueda. Para poder abordar el problema con mayor claridad conviene visualizar la lista de un modo diferente, colocando los *nodos* en forma linealizada, uno detrás del otro, manteniendo el orden dado por su valor de prioridad, pero prescindiendo de su clasificación en niveles (ver **FIGURA 4.8**). Los enlaces *OPL* (no se muestran todos) son aquellos que llevan desde cada *nodo* hacia su consecutivo y los *SCL* los que dan saltos que cubren uno o más *nodos* (tampoco se muestran todos).

El problema en cuestión es el siguiente: ¿cómo convendrá continuar la búsqueda una vez que el punto de *inserción* (INS) ha sido "acorralado" por un salto *SCL*?. Hasta el momento se había sugerido continuar intentando saltos *SCL* y ante cada fracaso dar un salto *OPL*. De esta manera, algún salto *OPL* determinará el

punto exacto de *inserción*. Ahora bien, cada par de saltos *SCL-OPL*, cuando el *SCL* ha fallado, implica dos comparaciones. En algún momento un salto *SCL* podría ser exitoso y "quedar", habiéndose cubierto entonces varios *nodos*, pero también se puede ver que si muchos de esos saltos fallan, es decir "se pasan", tal vez resulte más económico limitarse a dar sólo saltos *OPL* luego de que el primer salto *SCL* haya cercado el punto buscado. Resumiendo, una vez acorralado el punto de *inserción*, ¿convendrá continuar intentando saltos *SCL* o sólo limitarse a dar sólo saltos *OPL*?. Experimentalmente resulta que , si bien el problema está bastante influido por la distribución temporal de *eventos* dentro de la lista, la diferencia de rendimiento es bastante marcada a favor de proceder siempre en secuencias tales que luego de cada salto *OPL*, mientras se pueda, se prosiga por el camino *SCL*.

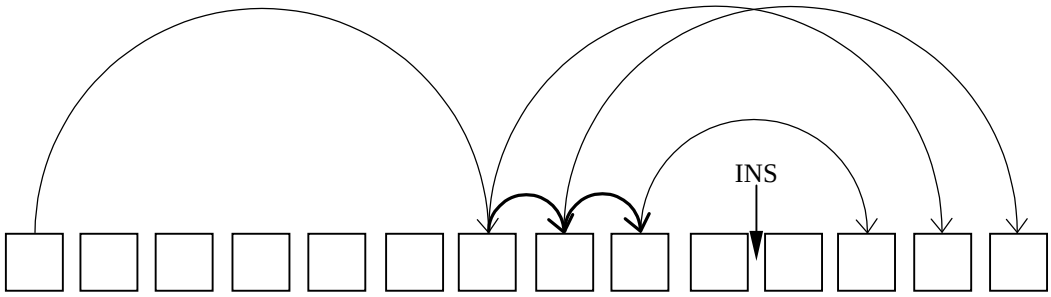


FIGURA 4.8 Short-Cut List Vista Linealizada

La principal dificultad de implementación está en la elección de nivel para los *nodos* insertados. Sin perjuicio de los principios que rigen, fundamentalmente, las búsquedas en el tipo de estructura propuesta, una variante de la misma fue ensayada. La misma consiste en fijar el número máximo de niveles y también el número máximo de *nodos* por nivel. Entonces es posible crear un arreglo, cada uno de cuyos elementos sea un puntero al primer *nodo* de cada nivel. Es simple llevar la cuenta del número de *nodos* por nivel, a partir del cual el criterio para resolver una nueva *inserción* es el siguiente: 1) Se ubica el punto de *inserción* del mismo modo que en la implementación original, 2) Se detecta el más bajo de los niveles con lugar disponible y 3) Se elige el lugar correspondiente en el nivel seleccionado en 2) mediante *búsqueda secuencial* simple. Estas *búsquedas secuenciales* son factibles gracias al arreglo que da acceso al primer *nodo* de cada cola. Las *remociones* operan tal cual la versión original por lo que permanentemente se va generando lugar disponible en todas los niveles. Esta variante hace que el "llenado" de la estructura se haga de abajo hacia arriba. Debido al uso de la *búsqueda secuencial* tradicional, es preciso que las colas se mantengan razonablemente cortas. La puesta en marcha requiere entonces la elección de ese valor como parámetro de operación (la cantidad de niveles ha de elegirse suficientemente alta como para que nunca falten lugares disponibles). Si bien persiste al influencia de la distribución temporal de los *eventos* sobre el rendimiento, esta alternativa ofrece mejores resultados que la original.

4.9. Conclusiones

En la Sección **4** se introdujeron los principios del mecanismo que sustenta la implementación *Short-Cut List*. Se analizaron sus principales operaciones, se sugirieron pautas de implementación y se propusieron algunas variantes.

La idea central de esta propuesta es la introducción de los atajos como una alternativa para acelerar las *búsquedas secuenciales* sobre listas ordenadas. Existe un variedad de algoritmos cuyas ventajas de búsqueda están, entre otras, en saltearse un número de *nodos* en cada salto, pero en general lo hacen siguiendo un esquema determinado por particiones o algún criterio que, en alguna medida, les da un carácter determinístico. En tanto en *Short-Cut List* los atajos tiene un carácter aleatorio directamente relacionado con la aleatoriedad de los valores de *t* de los *nodos* arribados para su *inserción*. La estratificación o clasificación en niveles tiene por única finalidad el establecimiento de los caminos alternativos, siendo ésta una tarea no del todo sencilla en listas implementadas mediante *nodos* enlazados.

5. Análisis Experimental

En esta Sección se cubre la faz experimental de esta Tesis. Las implementaciones *Linked Exact Bottom Up* - BSS, *Array Bottom Up* - BSS, *Short-Cut List* (versión con Arreglo de Cabeceras), *Linear Median Value*, la Estructura de *Henriksen*, *Pairing Heap*, *Skip List*, *Implicit Heap*, *Skew-Heap (Top-Down)* y *Splay Tree* se cotejan experimentalmente entre sí. El modelo utilizado en la mayoría de los ensayos es *The Hold Model*. Se comentan algunos modelos alternativos y se realizan ensayos mediante la simulación de *Modelos de Cola*.

5.1. The Hold Model

La SED se lleva a cabo mediante la ejecución de un programa en un sistema de cómputo, razón por la cual además de las determinaciones de complejidad y análisis de carácter teórico de sus partes componentes (principalmente el *calendario*) es de sumo interés evaluar su rendimiento en forma experimental. El principal inconveniente es que los sistemas pasibles de ser simulados en esta modalidad presentan una diversidad muy grande de características y propiedades por lo que someten al *calendario* de los *simuladores* a exigencias de muy diverso tipo. Es imposible elegir un modelo que resulte plenamente representativo de la totalidad. No obstante hay algunos diseñados específicamente para evaluar las prestaciones del *calendario* bajo distintas condiciones de operación. Es preciso entonces caracterizar las condiciones de operación de alguna forma y para ello hay dos parámetros que resultan particularmente adecuados: el número medio de *registros* presente en el *calendario* (*n*) y su distribución temporal. *The Hold Model* es un modelo que permite ajustar esos dos parámetros a voluntad, posibilitando ensayos de una gran generalidad. Tiene su origen en la sentencia *hold(X)* del lenguaje *SIMULA*, de cuya aplicación resulta la ejecución consecutiva de las siguientes acciones:

1) Se practica una *remoción* sobre el *calendario*.
2) Se le suma *X* al valor de *t* del *registro* removido.
3) Se *inserta* nuevamente en el *calendario* el *registro* removido y modificado.

The Hold Model consiste en ejecutar una secuencia suficientemente larga de operaciones *hold(X)* en las que *X* es tomado de un generador de valores aleatorios ajustado a una distribución de probabilidad preestablecida. Así la cantidad de *registros* dentro del *calendario* se mantiene constante y conocida, y sus valores de *t* se distribuyen según un esquema determinado. De este modo es posible operar el *calendario* ante una variada gama de exigencias. *The Hold Model* tiene además la interesante propiedad de que una aplicación suficientemente prolongada lleva el *calendario* a un estado estable; esto significa que para cada valor de *n* y cada distribución de la variable *X*, habrá un umbral en el número de operaciones *hold(X)*, más allá del cual ni la distribución de los *eventos pendientes* ni la duración de las operaciones se modifique sustancialmente. Esto permite, en general, detectar sin mayores ambigüedades el arribo al resultado del experimento.

Sea entonces un *calendario* con *n registros* y sea la secuencia de operaciones $hold(X_1)$, $hold(X_2)$, $hold(X_3)$, ... sobre el mismo, donde X_1, X_2, X_3 son variables aleatorias idénticamente distribuidas, con función de densidad de probabilidad *f(x)* y función de distribución acumulativa *F(x)*:

$$F(x) = \int_{x=0}^{x} f(t)dt$$

Si inmediatamente luego de cada *remoción* se descuenta el valor de *t* del *registro* recientemente removido (*tiempo actual*) al *t* de los *n-1 registros* restantes, se podrá asociar los valores de *t* resultantes a una variable aleatoria, con su propia distribución. Dado que el tiempo *X* que adiciona cada operación *hold(X)* a lo largo de todo el ensayo es generado a partir de la misma distribución de probabilidad *f(x)*, una vez alcanzado el estado estable los tiempos de cada uno de los *registros* del *calendario* seguirán también una misma distribución, cuya densidad notaremos *g(x)*. Resultan entonces dos distribuciones perfectamente distinguibles en el modelo, la de los tiempos que cada *registro* insertado ha de permanecer en el *calendario*, *f(x)*, y la del conjunto de *eventos pendientes*, es decir la del *t* de los *registros* residentes en el *calendario* en espera de ser removidos una vez descontado el valor de *t* del primer *registro*, *g(x)*. Es útil determinar la relación entre ambas funciones [9].

Inmediatamente luego de practicada una *remoción* en un *calendario* con *n registros* habrá *(n-1)g(x)dx registros* en un intervalo *(x,x+dx)* del eje de tiempo.

Mientras el tiempo avanza desde *0* hasta *dt*, se suceden operaciones *hold(X)* con dos consecuencias:

1) El intervalo *(x,x+dx)* se desplaza hacia la posición *(x-dt,x+dx-dt)* debido a las *remociones*.

2) El número de *registros* dentro del intervalo *(x,x+dx)* crece en un número Δ producto de las *inserciones*, donde,

$$\Delta = (n-1)g(x-dt)dx - (n-1)g(x)dx$$

El total de *registros* removidos corresponde precisamente a la cantidad alojada en el primer intervalo de ancho *dt*, es decir *(n-1)g(0)dt*. Pero dado que las operaciones *hold(X)* no *modifican* el número total de *registros* en el *calendario*, la cantidad removida será la misma que la cantidad insertada o, lo que es lo mismo, que la cantidad de *registros* generados para ser insertados. Ahora bien, no todos los *registros* generados en el lapso *dt* caerán dentro del intervalo *(x-dt,x+dx-dt)*, sino sólo la proporción correspondiente, *(n-1)g(0)dt f(x)dx*, siendo este número precisamente el Δ determinado más arriba. Igualando se obtiene

$$\Delta = (n-1)g(x-dt)dx - (n-1)g(x)dx = (n-1)g(0)dt\, f(x)dx,$$

lo que implica que

$$(g(x-dt) - g(x))/ dt = g(0)\, f(x).$$

Tomando límites cuando dt tiende a cero se obtiene

$$- g'(x) = g(0)\, f(x),$$

de donde

$$g(x) = g(0)[1- F(x)].$$

Integrando entre *0* e ∞, y utilizando el conocido resultado del cálculo de probabilidades

$$\int_{x=0}^{\infty} [1- F(x)]\, dx = E(x) = \mu$$

obtenemos $g(0) = 1/\mu$. Finalmente,

$$g(x) = [1-F(x)] / \mu$$

Esta última expresión permite conocer la distribución temporal de los *registros* residentes en el *calendario* como función de la distribución de los tiempos *X* que se adiciona en cada operación *hold(X)*. En general interesará conocer *G(x)*, o sea la distribución de probabilidad acumulativa de *g(x)*, ya que para cualquier valor positivo de *t*, *G(t)* es la probabilidad de que un determinado *registro* permanezca alojado en el *calendario* un tiempo no mayor que *t*, a la espera de ser removido

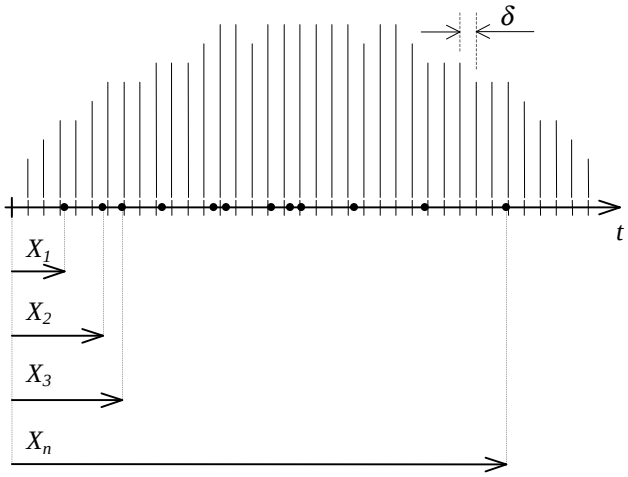


FIGURA 5.1 Trazado de *f(x)* a partir de lecturas de *X*

La relación entre las distribuciones *f(x)* y *g(x)* puede visualizarse mediante un análisis gráfico. Para ello es preciso establecer, primeramente, una metodología que permita trazar la función densidad de probabilidad (*fdp*) de una variable aleatoria, o al menos una aproximación, a partir de lecturas de la misma. Para esto, si la distribución bajo análisis es *f(x)*, se traza primeramente un eje en las unidades de *X*, por ejemplo *t*. Se releva entonces una serie de valores X_1, X_2, X_3, ... X_n, a partir de sucesivas lecturas de la variable aleatoria *X*. Se colocan esas lecturas sobre el eje, para lo cual se mide la distancia desde el origen hasta el valor leído cada vez y se marca allí un punto. El proceso se muestra en la **FIGURA 5.1**.

Luego se divide el eje en segmentos iguales, de longitud δ y en cada marca divisoria se traza una ordenada proporcional a la cantidad de puntos que aloja el segmento del cual la marca es uno de los extremos. La envolvente de esas ordenadas se acercará entonces a la *fdp f(x)*, y si $\delta \to 0$ y $n \to \infty$ la envolvente se convertirá en la función misma.

Supóngase ahora que el eje es una representación temporal del *calendario*, de manera que los puntos corresponden al tiempo de los *registros* planificados. Supóngase que la función *f(x)* y por ende las sucesivas lecturas de *X* son las mismas que antes y que además *f(x)* es la *fdp* de la variable aleatoria cuyos valores han de sumarse en cada operación de Hold. Supóngase también que el *calendario* ya aloja una cierta cantidad de *registros*. Si cada vez que le llega el turno de ser removido a un *registro*, se le resta a todos los planificados (incluido el que se va a *remover*) el tiempo del que se va a *remover*, eso equivale a colocar el nuevo origen del eje sobre el *nodo* removido y a realizar desde allí la nueva medición de *X*. Entonces al ir removiendo *registros*, para ejecutar operaciones de Hold, se irá desplazando el origen desde el que se hacen las mediciones para dibujar nuevos puntos y se irá descartando la parte del eje que va quedando a la izquierda del origen desplazado.

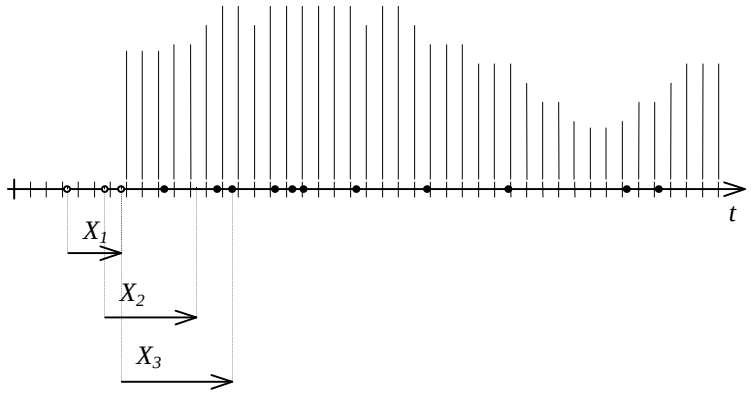


FIGURA 5.2 Trazado de *g(x)* a partir de lecturas de *X*

Las ordenadas que están sobre las marcas de zonas que ya no alojan *registros* pierden vigencia y no deben considerarse. Resulta como si ante el agregado de cada nuevo punto, todo el gráfico se fuera desplazando ("enrollando") hacia la derecha. La relación entre la distribución resultante, *g(x)*, y la que rige las distancias desde el origen (ahora móvil) hasta cada nuevo punto, o sea *f(x)*, no es directamente visualizable en el gráfico, pero sí queda claro que las funciones resultantes son diferentes y que existe una relación entre ambas, a priori no muy fácil de determinar. En el trazado de *f(x)* todas las distancias correspondientes a lecturas de *X* se hacían desde un punto fijo. En el caso de *g(x)*, ante cada punto agregado, el origen va avanzando un tramo igual a la distancia entre dos puntos consecutivos existentes y esto origina una deformación respecto *f(x)*. La nueva curva y un bosquejo de su proceso de construcción se pueden ver en la **FIGURA 5.2**.

Volviendo ahora a la relación matemática que vincula ambas funciones, supóngase que se realiza un ensayo de *Hold* sumando cada vez un tiempo tomado de la distribución *uniforme* sobre el intervalo *(0,1)* al *t* de cada *registro* removido; es decir,

$$f(x) = \begin{cases} 1 & 0 \le x \le 1 \\ 0 & x > 1 \end{cases}$$

Entonces

$$G(x) = \begin{cases} 2x-x^2 & 0 \le x \le 1 \\ 1 & x > 1 \end{cases}$$

Algunas observaciones:

1) *G(0.5) = 0.75* significa que, una vez alcanzado el estado estable, el 75% de los *eventos pendientes* estará planificado para ocurrir en un lapso de *0.5* unidades de tiempo, o dicho de otro modo la tres cuartas partes de los *eventos* planificados estará a menos de *0.5* unidades de tiempo de ser removida y el resto en las *0.5* unidades de tiempo siguientes.

2) Si el *calendario* se implementa mediante una lista simple ordenada, la *búsqueda secuencial* realizada para ejecutar una *inserción* será más efectiva si se inicia en el último elemento de la lista (el de más alto valor de *t*) que si se inicia en el primero.

El ejemplo pone en evidencia la influencia de la distribución de probabilidad de *X* y su relación con el modelo de *calendario* implementado.

Un valor aceptado como indicador de la distribución de los *eventos* en el *calendario* es el valor esperado de la distribución *g(x)*. En el ejemplo anterior dicha media vale *1/3*. Eso significa que, aún cuando la totalidad de *eventos pendientes* se distribuya en un intervalo comprendido entre el *tiempo actual* (caso *X=0*) y una unidad más que ese tiempo (caso *X=1*), el promedio estará a *1/3* de unidades del *tiempo actual*, ratificando esto lo dicho en cuanto a la concentración de *eventos* en tiempos bajos.

Se presentan seguidamente cinco distribuciones capaces, en conjunto, de cubrir de un modo bastante representativo las demandas requeridas de un *calendario* en materia de distribución temporal de *eventos*. El valor esperado de *f(x)* es *1* en todos los casos. Los valores de la esperanza o media según la distribución *g(x)* varían entre *0* y *1*, indicando *0* un comportamiento tipo *LIFO* (Last In - First Out) y *1* un comportamiento *FIFO* (First In - First Out). Estas distribuciones serán utilizadas en los próximos ensayos.

Distribución	***f(x)***		***g(x)***		***media***
Exponential	e^{-x}	$x \ge 0$	e^{-x}	$x \ge 0$	0.50
Uniform	0.5	$0 \le x \le 2$	1-x/2	$0 \le x \le 2$	0.67
	0	$x > 2$	0	$x > 2$	
Biased	0	$0 \le x \le 0.9$	1	$0 \le x \le 0.9$	0.97
	5	$0.9 < x \le 1.1$	5(1.1-x)	$0.9 < x \le 1.1$	
	0	$x > 1.1$	0	$x > 1.1$	
Bimodal (q=2/21)	0.9/q	$x \le q$	1-(0.9x/q)	$x \le q$	0.13
	0	$q < x \le 100q$	0.1	$q < x \le 100q$	
	0.1/q	$100q < x \le 101q$	10.1-(0.1x/q)	$100q < x \le 101q$	
	0	$x > 101q$	0	$x > 101q$	
Triangular	8x/9	$x \le 1.5$	$1-4x^2/9$	$x \le 1.5$	0.80
	0	$x > 1.5$	0	$x > 1.5$	

TABLA 5.1 Distribuciones a utilizar en los ensayos

Apelando al criterio presentado más arriba para trazar una aproximación de la *fdp* de una variable aleatoria se procedió determinar, en forma experimental, las curvas de las funciones *f(x)* y *g(x)* indicadas en la **TABLA 5.1**. Para el caso de *f(x)* se realizaron 10^6 lecturas de cada variable con la distribución correspondiente. Esos valores se *insertaron* en una implementación Linear de *calendario*, luego se le practicaron 10^6 operaciones de Hold para alcanzar el estado estable y luego otras 10^6 operaciones de Hold. Finalmente se leyeron uno a uno los valores de *t* de los *registros* residentes en el *calendario* y se procedió, tal como con *f(x)*, al trazado de *g(x)*. Las determinaciones se presentan en **FIGURAS 5.3**, **5.4**, **5.5**, **5.6** y **5.7**.

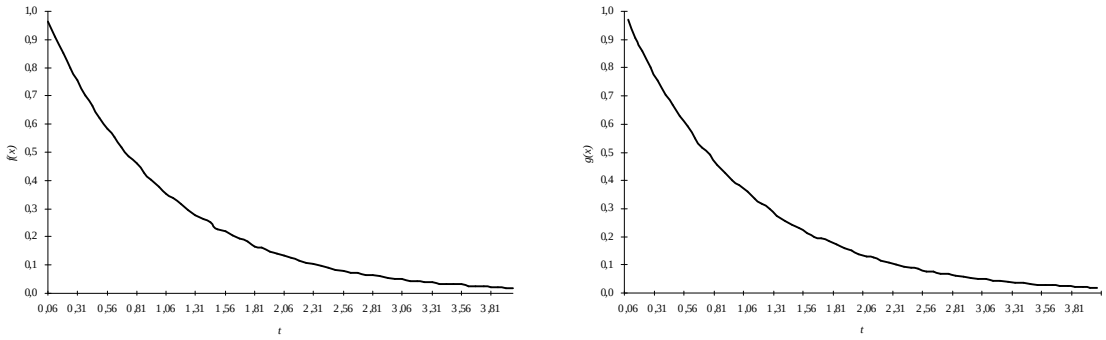


FIGURA 5.3 *f(x)* y *g(x)* - Exponential

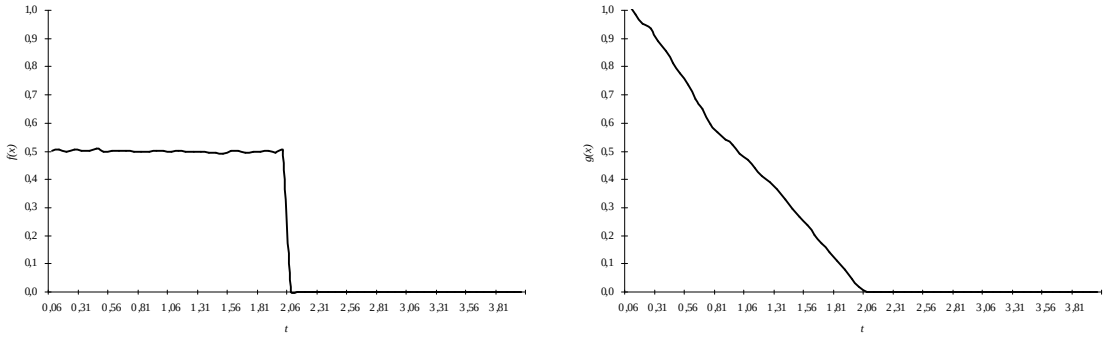


FIGURA 5.4 *f(x)* y *g(x)* - Uniform

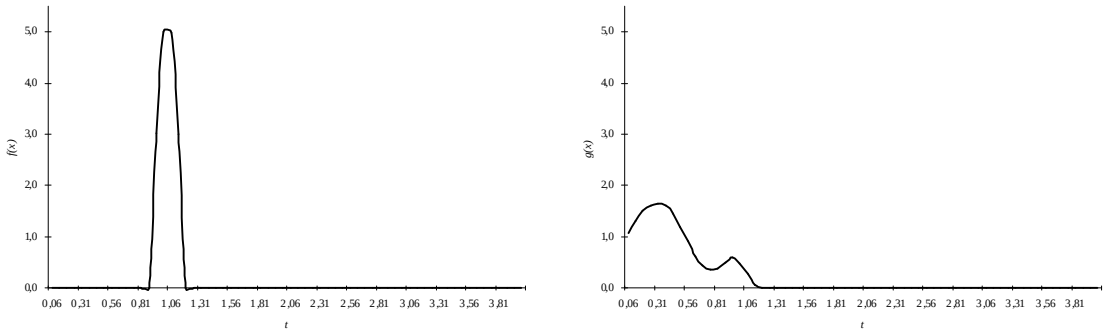


FIGURA 5.5 *f(x)* y *g(x)* - Biased

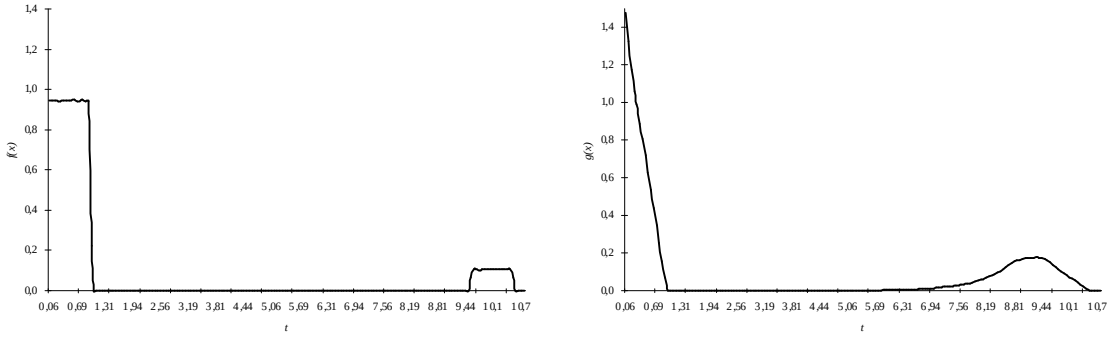


FIGURA 5.6 *f(x)* y *g(x)* - Bimodal

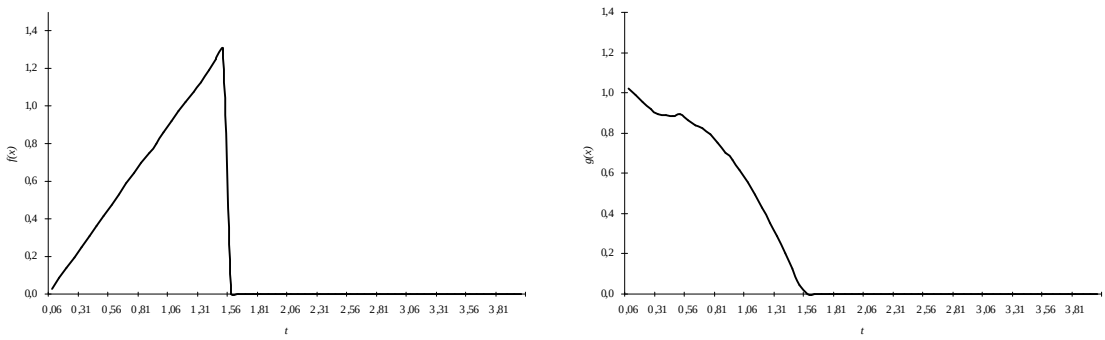


FIGURA 5.7 *f(x)* y *g(x)* - Triangular

5.1.1. Resultados

Se presenta seguidamente una serie de ensayos comparativos sobre *The Hold Model*, realizados de acuerdo a las pautas descritas en la presente Sección. Los *calendarios* utilizados pertenecen a un conjunto de implementaciones programadas específicamente para estos experimentos y se indican a continuación del último punto de cada curva. Se trata de: *Linked Exact Bottom Up* - BSS, *Array Bottom Up* - BSS, *Short-Cut List* (versión con Arreglo de Cabeceras), *Linear Median Value*, la estructura de *Henriksen*, *Pairing Heap*, *Skip List*, *Implicit Heap*, *Skew-Heap (Top-Down)* y *Splay Tree*. Para cada una de las distribuciones presentadas en la **TABLA 5.1** se realizaron dos juegos de ensayos, uno cubriendo una gama de "grandes cantidades de *registros*" (hasta poco más de 300.000, **FIGURAS 5.8**, **5.9**, **5.10**, **5.11** y **5.12**) y otro para tamaños más reducidos (hasta poco más de 300 *registros*, **FIGURAS 5.13**, **5.14**, **5.15**, **5.16** y **5.17**).

Los detalles técnicos y la metodología correspondiente a la realización de los ensayos pueden consultarse en la Sección **5.3**.

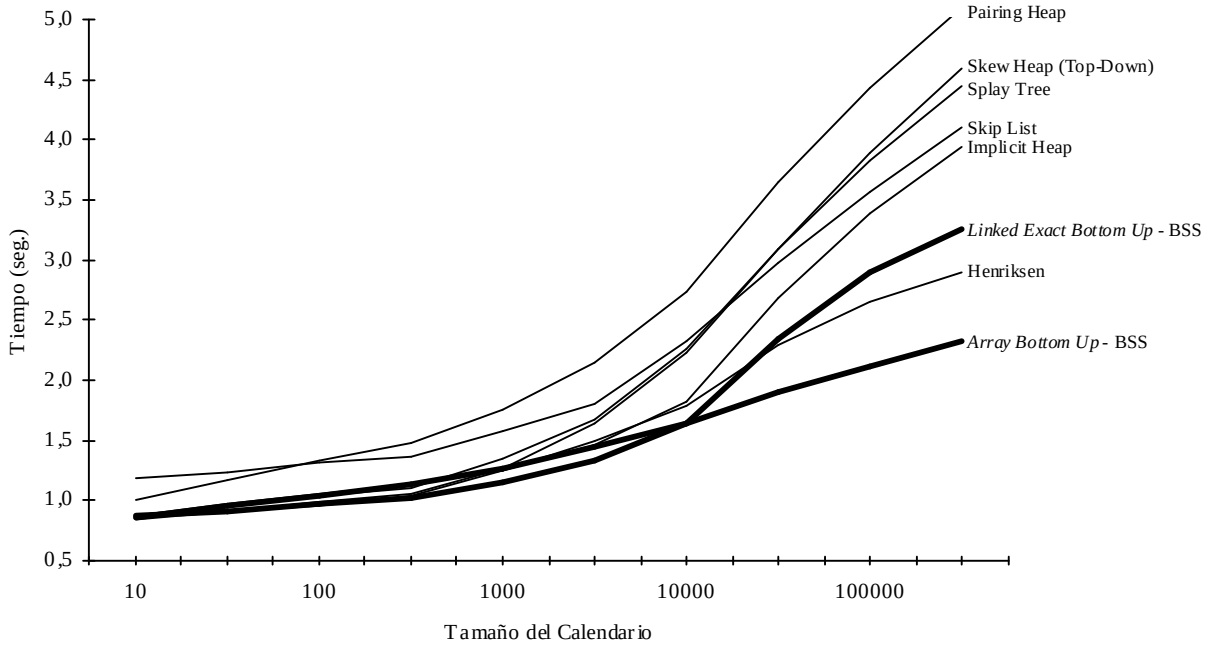


FIGURA 5.8 The Hold Model - Exponential

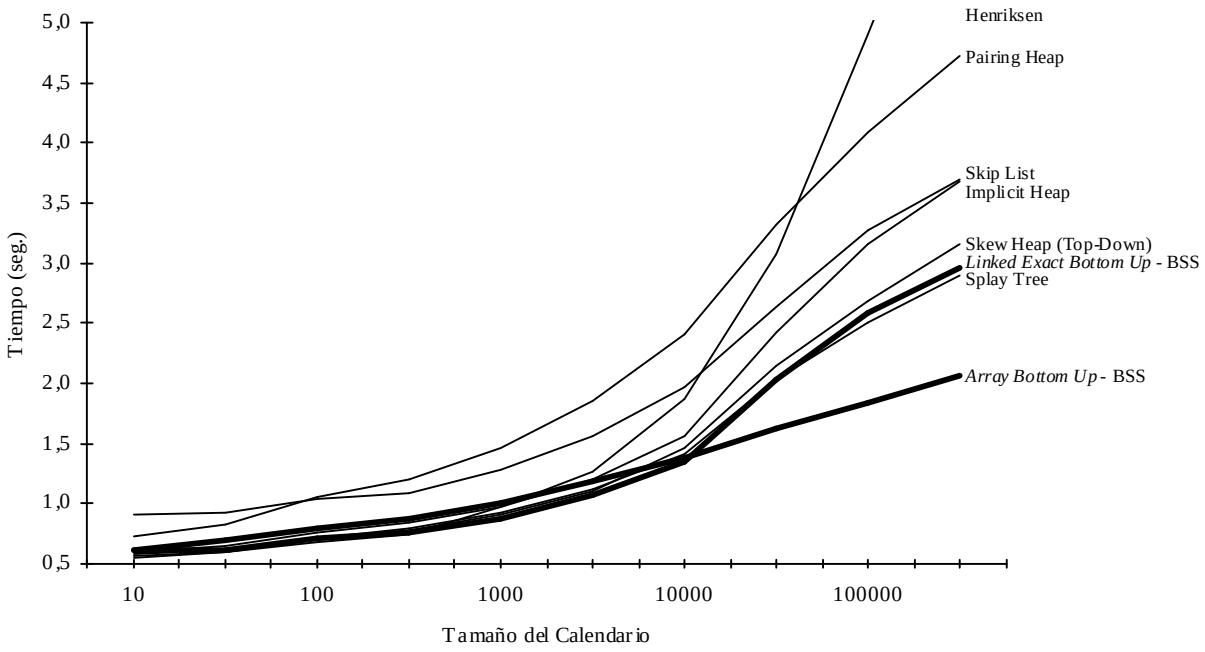


FIGURA 5.9 The Hold Model - Uniform

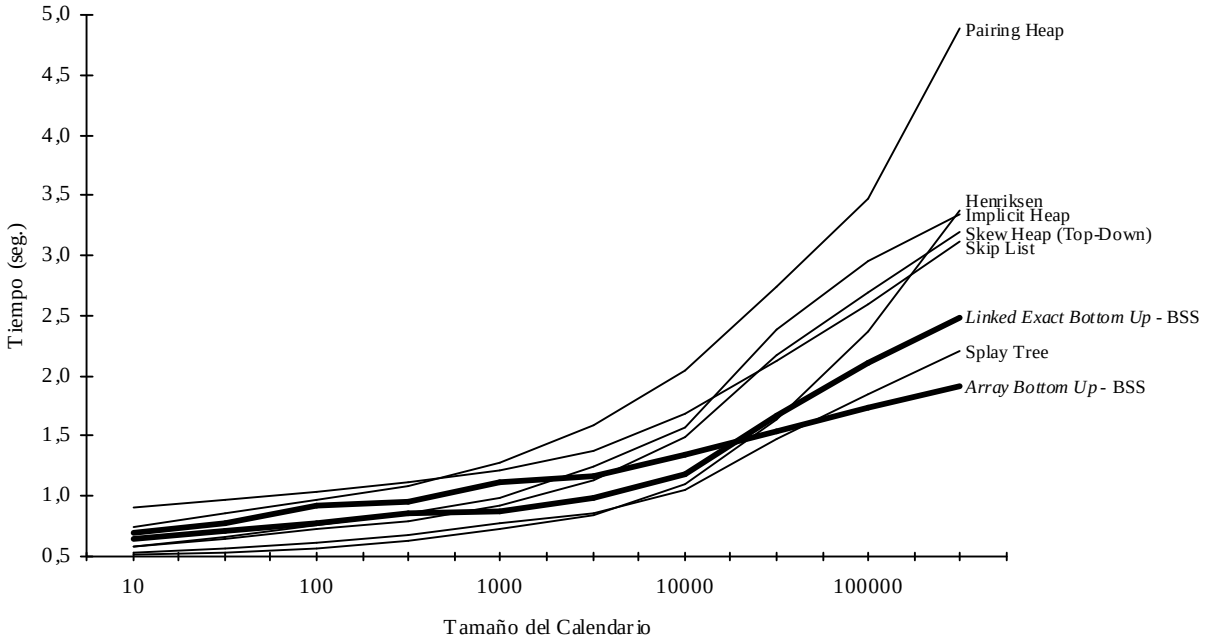


FIGURA 5.10 The Hold Model - Biased

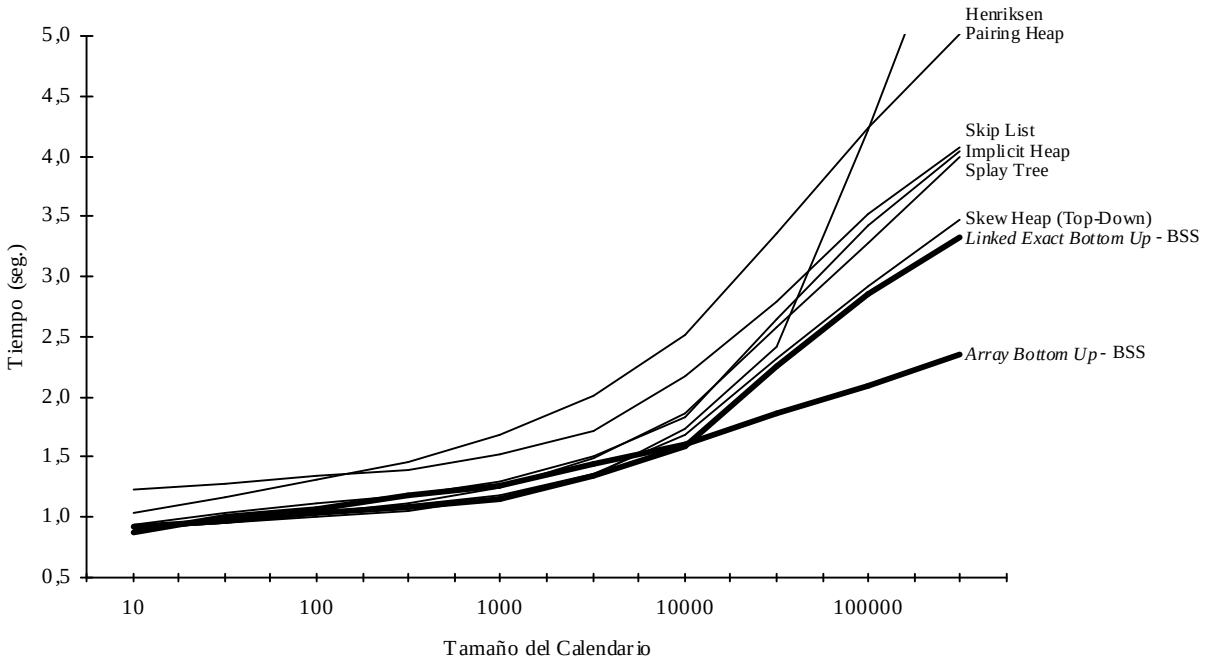


FIGURA 5.11 The Hold Model - Bimodal

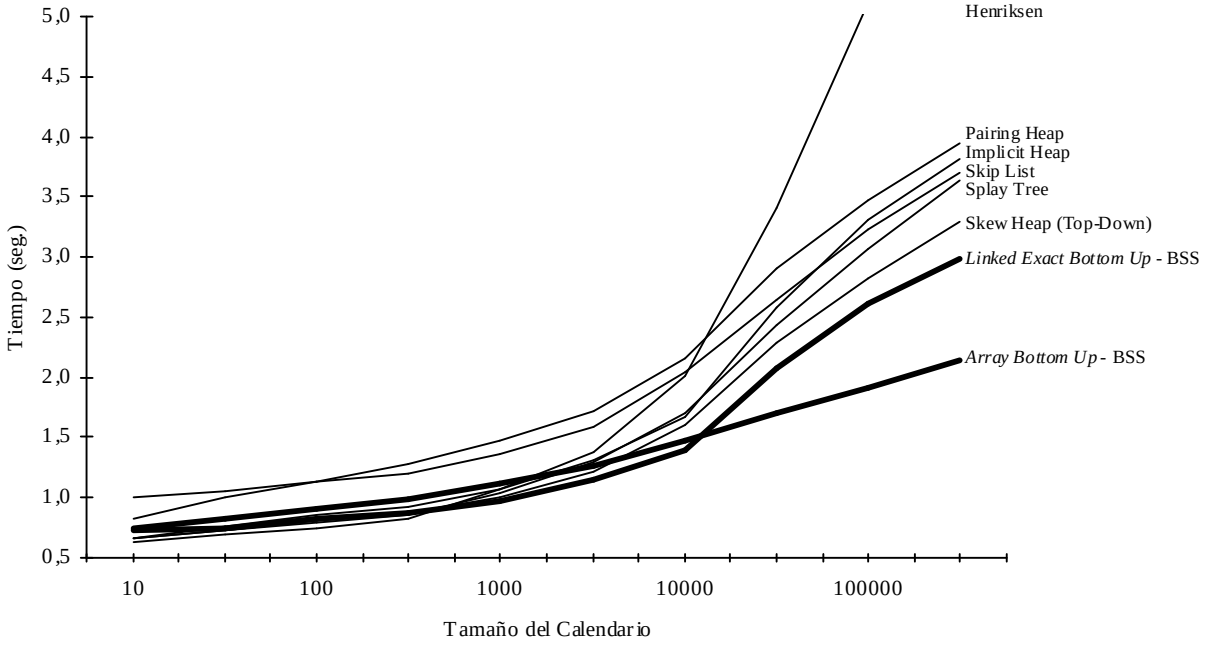


FIGURA 5.12 The Hold Model - Triangular

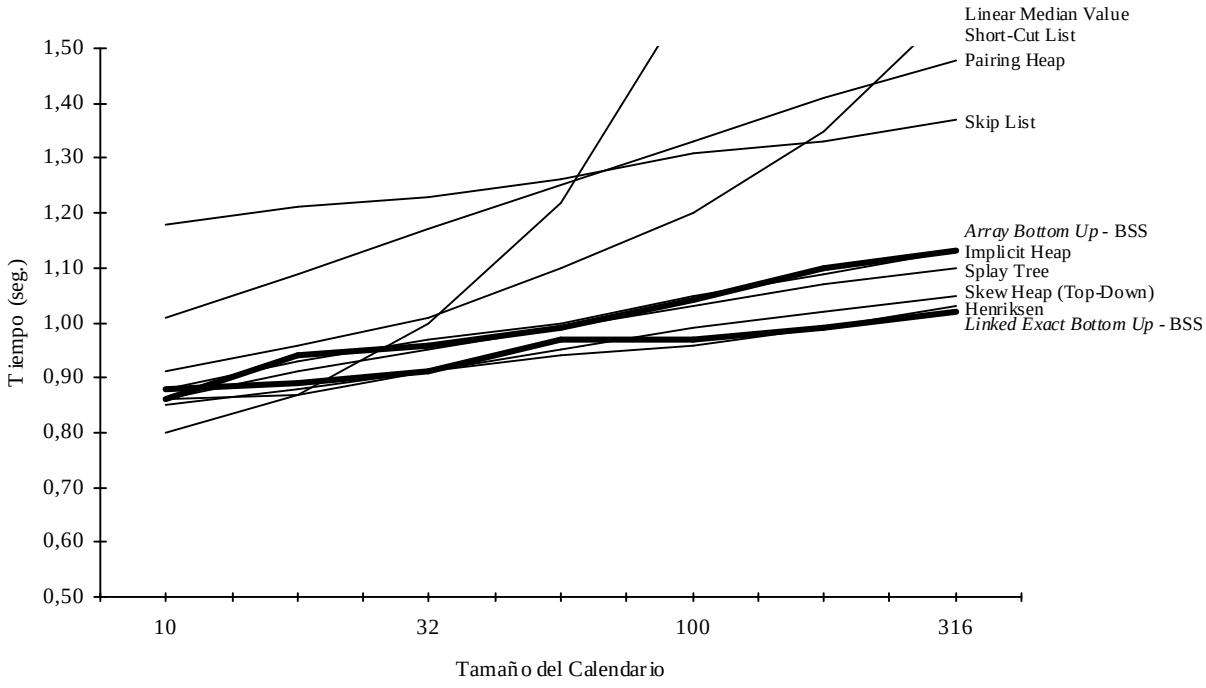


FIGURA 5.13 The Hold Model - Exponential

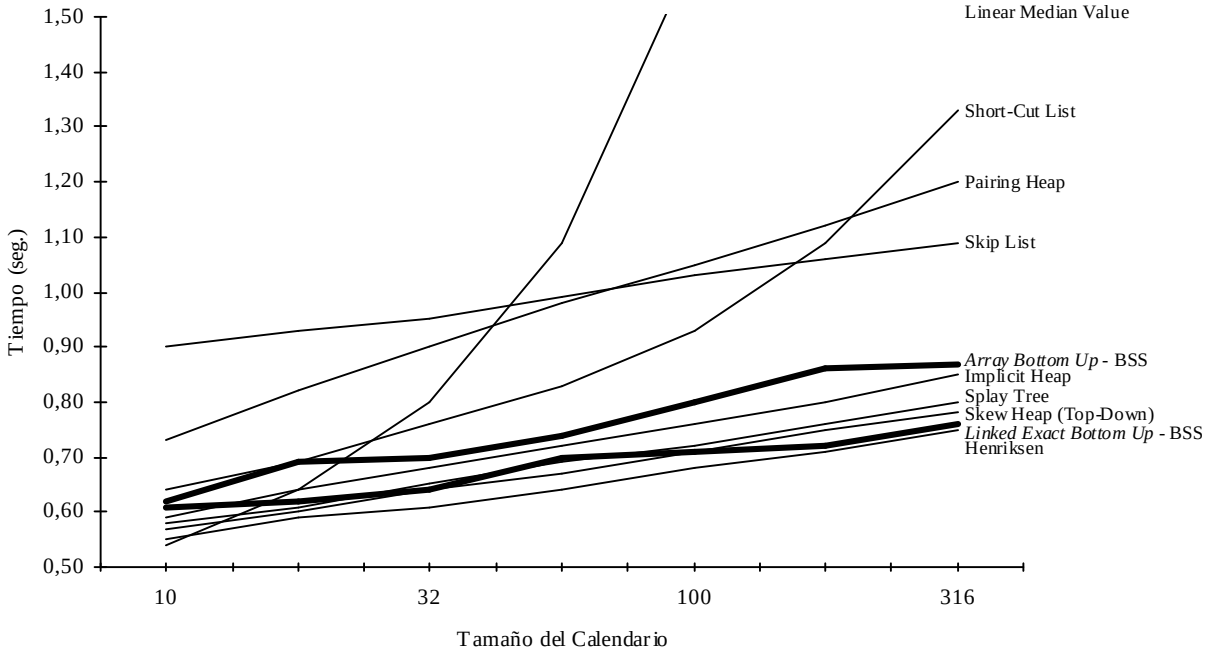


FIGURA 5.14 The Hold Model - Uniform

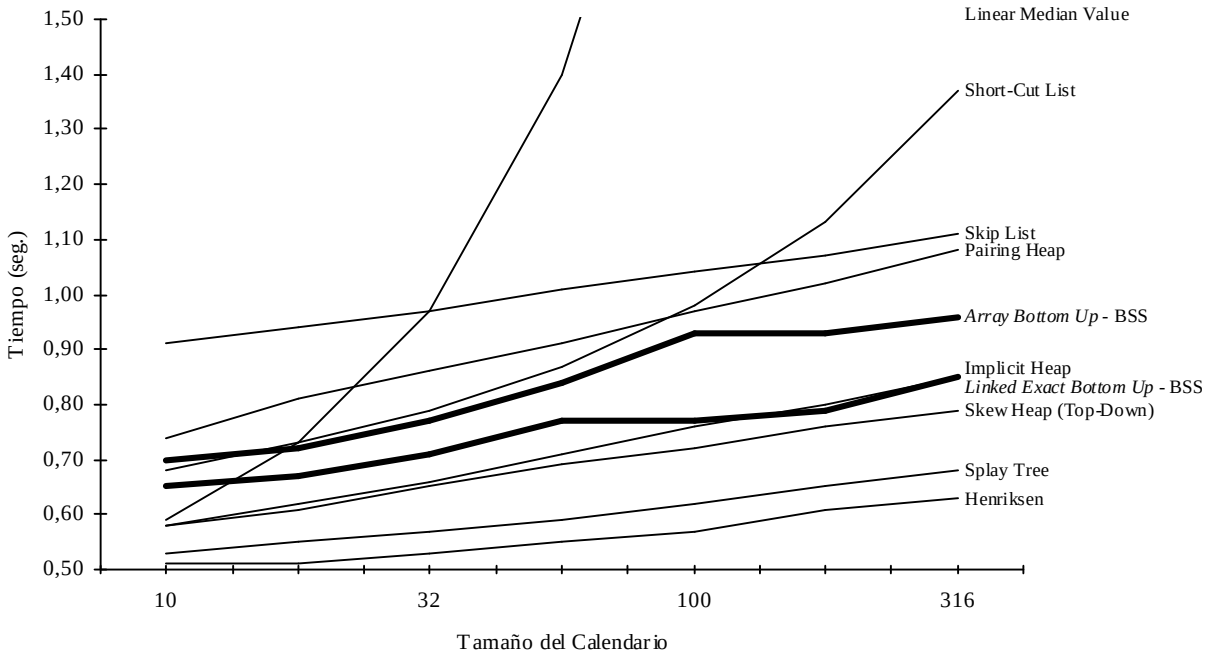


FIGURA 5.15 The Hold Model - Biased

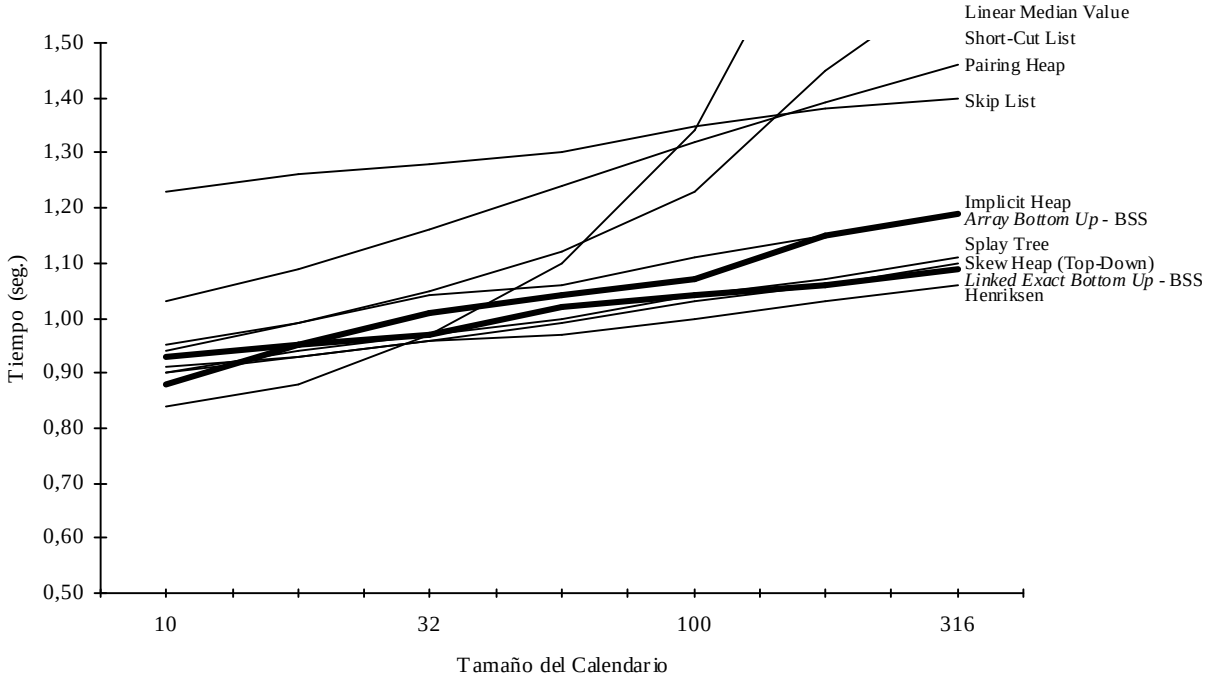


FIGURA 5.16 The Hold Model - Bimodal

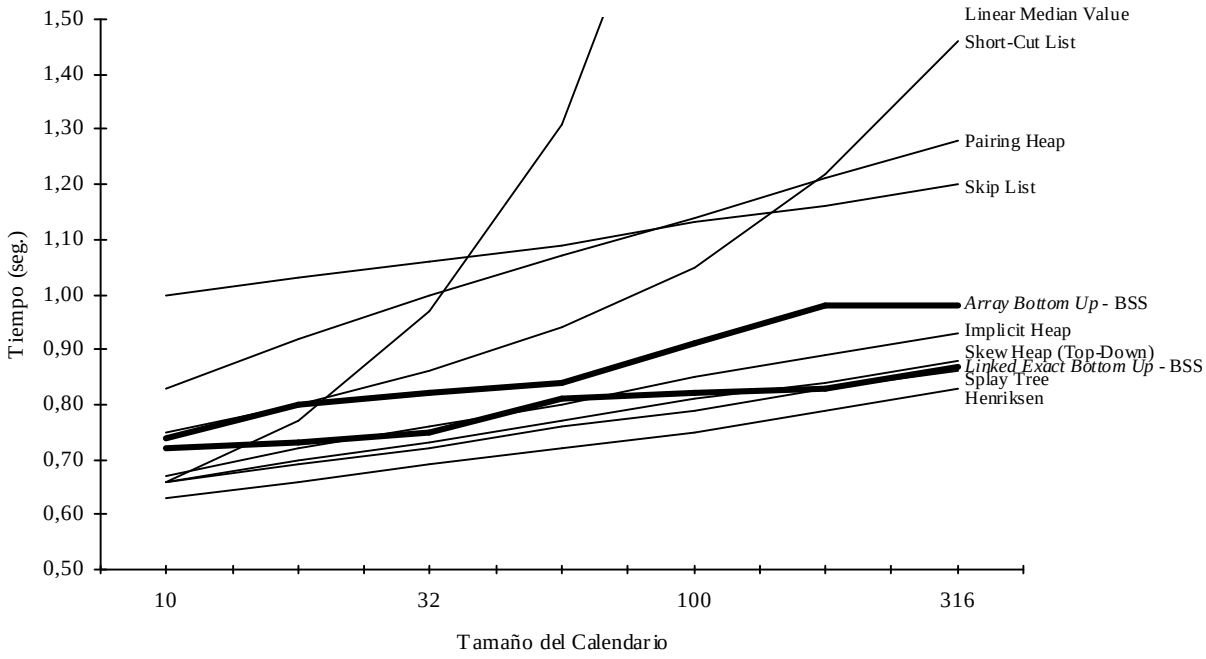


FIGURA 5.17 The Hold Model - Triangular

5.2. Modelos de Cola

The Hold Model nunca remueve *eventos* intermedios sino, únicamente, el de más bajo valor de *t*. Si bien ese es el caso más frecuente en SED, en la evaluación de implementaciones de *calendario* vale la pena considerar también tests que remuevan *eventos* intermedios. Otra limitación importante de *The Hold Model* está en el hecho de que el número de *eventos* planificados se mantiene constante a lo largo del ensayo. Algunas implementaciones de *calendario* ejecutan procesos de redimensionamiento al atravesar umbrales en el valor del su tamaño. Otras presentan un rendimiento fuertemente dependiente del número de *eventos*. Un ensayo en el que el *calendario* varíe su tamaño plantearía exigencias que *The Hold Model* no ofrece y que sólo permite aproximar mediante múltiples ensayos. De hecho no es lo mismo ensayar un *calendario* tantas veces como tamaños se desee testear, que hacerlo en un único ensayo en el que el tamaño evolucione a través de los valores que constituyen la gama de interés, ya que las transiciones por valores umbrales y sus esfuerzos computacionales asociados sólo pueden ser evaluados si la variación se produce en el mismo ensayo.

Para salvar esas limitaciones de *The Hold Model* la literatura menciona, entre otras, dos alternativas. Una es el modelo *Up/Down* [20], en que el *calendario* es llenado mediante *inserciones* consecutivas y luego, una vez alcanzado un determinado número de *eventos*, es vaciado mediante *remociones* sucesivas. El proceso puede repetirse cíclicamente para facilitar la medición de tiempo de ejecución, pero la esencia del mismo está en el proceso de "llenar" y luego "vaciar" completamente el *calendario*, ya sea una o varias veces pero sin intercalar *inserciones* y *remociones*. Este modelo constituye una variante extrema, absolutamente opuesta a *The Hold Model* y debe asociarse a simulaciones en estado transitorio, mientras que *The Hold Model* aproxima su comportamiento al de simulaciones que han alcanzado un estado estable. La otra alternativa citada en la literatura es un proceso de Markov con dos estados, *"insertar"* y *"remover"*, en el que el arribo a cada estado implica la ejecución de la operación indicada por su nombre, y el tránsito hacia el estado siguiente se realiza de acuerdo a una matriz de probabilidad de transición, con parámetros α y β., como la de **FIGURA 5.18**.

Este modelo, conocido como *Markov Hold Model* [20], se comporta de un modo fuertemente dependiente de la relación entre los valores de las probabilidades de transición α (de *"remover"* a *"remover"*) y β (de *"insertar"* a *"insertar"*). Asumiendo que comienza a operar con el *calendario* ocupado, si $\alpha > \beta$, en un tiempo razonablemente prolongado se va a vaciar; si $\alpha < \beta$ la cantidad de *eventos* crecerá en forma proporcional al

tiempo transcurrido; en tanto si $\alpha = \beta$ la cantidad de *eventos* tenderá a mantenerse próxima al valor inicial. El hecho a destacar es que si existe interés en ensayar el *calendario* dentro de una gama de tamaños diferentes, en un mismo ensayo, esa posibilidad se transforma en definitiva en variantes de los modos "llenado" y "vaciado", sólo que con velocidades fijadas por la relación entre α y β, a diferencia del modelo *Up/Down* que llena o vacía a la máxima velocidad.

	remover	*insertar*
remover	α	$1-\alpha$
insertar	$1-\beta$	β

FIGURA 5.18 Matriz de Probabilidad de Transición del Markov Hold Model

En definitiva, es difícil hacer que la distribución del número de *eventos* planificados presente una dispersión importante, haciendo uso únicamente de las funciones *insertar* y *remover*. Una posibilidad para lograr mayor variabilidad en el tamaño del *calendario* es recurrir a modelos que, además, realicen borrado de *nodos* intermedios (*Delete*). En procura de este objetivo se realizaron pruebas sobre algunos *Modelos de Cola* [22]. En particular modelos con fallas ya que, como se verá inmediatamente, algunos presentan variaciones importantes en el número de *eventos* planificados sin que ello implique un crecimiento o decrecimiento sostenido de ese número.

Primeramente, a modo de introducción y como para contar con un código base a utilizar en los demás modelos, se planteó una única cola M/M/1 (sin fallas) mediante el algoritmo de **FIGURA 5.19**. Este modelo genera muy pocos *eventos*, concretamente un número que está entre uno y tres. Uno está garantizado por el *evento* de fin de simulación (END_SIM), en tanto tres corresponde al caso en que coexisten además un arribo (ARR) y un fin de servicio o partida (DEP). Eso hace que las *remociones* encuentren el *calendario* con no más de tres *eventos* planificados y las *inserciones* con no más de dos.

```
do
{
  current = FIRST(C);
  task    = current->class;
  time    = current->time;

  switch(task)
  {
    case ARR:
      q++;
      INSERT(C, ARR, time+expo(1/λ));
      break;

    case DEP:
      busy = 0;
      break;
  }

  if(!busy && q>0)
  {
    q--;
    busy = 1;
    INSERT(C, DEP, time+expo(1/μ));
  }

  DESTROY(current);

}while(task != END_SIM);
```

FIGURA 5.19 Algoritmo de Cola M/M/1 sin Fallas

FIRST e *INSERT* son las funciones que ejecutan las operaciones *remover* e *insertar* respectivamente, *q* es la cantidad de clientes en espera y *busy* un indicador de servidor ocupado. El argumento *C* indica el *calendario* sobre el que se hacen las operaciones. Un modo de validar la programación, tanto del *Modelo de Cola* como del *calendario* encargado de la gestión de los *eventos*, es cotejar resultados conocidos del modelo contra los mismos valores, obtenidos a partir de corridas de la simulación. En ese sentido un valor característico de las colas M/M/1 estables es la cantidad media de clientes luego de operar durante un tiempo infinito (*N*). Siendo λ la tasa media de arribos por unidad de tiempo (de modo que el tiempo entre arribos sigue la distribución *expo(1/*λ*)*) y μ la tasa media de servicios por unidad de tiempo (de modo que el tiempo de servicio sigue la distribución *expo(1/*μ*)*), entonces, cuando $\lambda < \mu$, condición necesaria y suficiente de estabilidad, se tiene $N = \lambda/(\mu - \lambda)$ [22]. En la **TABLA 5.1** se muestran los resultados de *N* para tres juegos distintos de valores de λ y μ y también el valor del mismo parámetro determinado a partir de tres corridas de la simulación (N_m). t_{max} es el tiempo de planificación del *evento* END_SIM, valor éste estrechamente relacionado a la cantidad de iteraciones del lazo.

[redacted]	[redacted]	$N = [redacted]$	N_m	t_{max}
0.9	1.0	9.0000	8.8475	10^6
0.5	1.0	1.0000	1.0029	10^6
0.2	1.0	0.2500	0.2504	10^6

TABLA 5.2 Resultados de Cola M/M/1 sin Fallas

Si bien un modelo de esta magnitud no reviste importancia como test de *calendario*, constituye la base para los siguientes.

Consideremos ahora que el servidor pueda fallar (y ser reparado). Corresponde decidir, primeramente, la política de falla e implementar los algoritmos correspondientes según el caso. Se implementaron tres modalidades:

1) Simulación de cola M/M/1 con fallas modo RESTART.

   Al producirse una falla:

   - Si el Servidor está ocupado, el cliente que está siendo servido queda en espera de la reparación, luego de lo cual es el primero en recibir servicio.
   - Los clientes en espera no son afectados.
   - La cola sigue aceptando clientes mientras dura la falla.

   El algoritmo se muestra en **FIGURA 5.20**. Este modelo incorpora dos nuevos *eventos* (FAIL y REPAIR) que son planificados en forma alternativa, y además la operación de borrado (*Delete*) para el caso en que el servidor se encuentre ocupado al momento fallar. *f* y *r* son las tasas medias de falla y reparación respectivamente y la variable *Next_Dep* conserva en todo momento la ubicación del último *evento* DEP planificado en el *calendario* de modo de poder borrarlo en caso de fallar el servidor estando ocupado.

2) Simulación de cola M/M/1 con fallas modo STOP.

   Al producirse una falla:

   - Si el Servidor está ocupado el cliente que está siendo servido queda en espera de la reparación, luego de lo cual es el primero en recibir servicio.
   - Los clientes en espera no son afectados.
   - La cola mata a todos los clientes que llegan durante la falla.

   El algoritmo se muestra en **FIGURA 5.21**.

3) Simulación de cola M/M/1 con fallas modo KILL.

   Al producirse una falla:

   - Si el Servidor está ocupado, mata al cliente que está siendo servido.
   - La cola mata a todos los clientes que están en espera.
   - La cola mata a todos los clientes que llegan durante la falla.

   El algoritmo se muestra en **FIGURA 5.22**.

```
do
{
  current = FIRST(C);
  task    = current->class;
  time    = current->time;

  switch(task)
  {
    case ARR:
      q++;
      INSERT(C, ARR, time+expo(1/λ));
      break;

    case DEP:
      busy = 0;
      break;

    case FAIL:
      out = 1;
      if(busy)
      {
        busy = 0;
        q++;
        Delete(C, Next_Dep);
      }
      INSERT(C, REPAIR, time+expo(1/r));
      break;

    case REPAIR:
      out = 0;
      INSERT(C, FAIL, time+expo(1/f));
      break;
  }

  if(!out && !busy && q>0)
  {
    q--;
    busy = 1;
    Next_Dep = INSERT(C, DEP, time+expo(1/μ));
  }

  DESTROY(current);

}while(task != END_SIM);
```

FIGURA 5.20 Algoritmo de Cola M/M/1 modo RESTART

La programación de estos algoritmos implica un esfuerzo adicional respecto al anterior y permite además una nueva validación de todo el *simulador*. En tal sentido es útil considerar ahora no sólo la cantidad media de clientes luego de operar durante un tiempo infinito (N), sino también el porcentaje de tiempo que el servidor permanece "sano" y ocupado (U).Las expresiones analíticas para estos parámetros [22], son:

1) Modo RESTART

   Condición de estabilidad: $\lambda/\mu < r/(f + r)$

   $N = \lambda\,[(f + r)^2+\mu\, f] / (f + r)[r\,(\mu - \lambda)-\lambda\, f]$

   $U = \lambda / \mu$

2) Modo STOP

   Condición de estabilidad: $\lambda < \mu$

   $N = \lambda / (\mu - \lambda)$

   $U = \lambda\, r / \mu\, (f + r)$

```
do
{
  current = FIRST(C);
  task    = current->class;
  time    = current->time;

  switch(task)
  {
    case ARR:
      if(!out) q++;
      INSERT(C, ARR, time+expo(1/λ));
      break;

    case DEP:
      busy = 0;
      break;

    case FAIL:
      out = 1;
      if(busy)
      {
        busy = 0;
        q++;
        Delete(C, Next_Dep);
      }
      INSERT(C, REPAIR, time+expo(1/r));
      break;

    case REPAIR:
      out = 0;
      INSERT(C, FAIL, time+expo(1/f));
      break;
  }

  if(!out && !busy && q>0)
  {
    q--;
    busy = 1;
    Next_Dep = INSERT(C, DEP, time+expo(1/μ));
  }

    DESTROY(current);

}while(task != END_SIM);
```

FIGURA 5.21 Algoritmo de cola M/M/1 modo STOP

3) Modo KILL

Condición de Estabilidad: Sistema Siempre Estable

$$N = [\lambda (1 - p) - \mu U] / f$$

$$U = \mu - z_1 [\lambda p(1 - z_1) + \mu + f] / \mu (1 - z_1)$$

Donde:

$$z_1 = [\lambda + \mu + f - sqrt(\Delta)] / 2 \lambda$$

y

$$\Delta = (\lambda + \mu + f)^2 - 4 \lambda \mu = (\lambda - \mu)^2 + f^2 + 2 f (\lambda + \mu) > 0$$

Los cálculos para distintos juegos de λ, μ, f y r, y los correspondientes resultados, obtenidos a partir de dos corridas de la simulación para cada modelo (N_m y U_m), se muestran el la **TABLA 5.3**.

```
do
{
  current = FIRST(C);
  task    = current->class;
  time    = current->time;

  switch(task)
  {
    case ARR:
      if(!out) q++;
      INSERT(C, ARR, time+expo(1/λ));
      break;

    case DEP:
      busy = 0;
      break;

    case FAIL:
      out = 1;
      q   = 0;
      if(busy)
      {
        busy = 0;
        Delete(C, Next_Dep);
      }
      INSERT(C, REPAIR, time+expo(1/r));
      break;

    case REPAIR:
      out = 0;
      INSERT(C, FAIL, time+expo(1/f));
      break;
  }

  if(!out && !busy && q>0)
  {
    q--;
    busy = 1;
    Next_Dep = INSERT(C, DEP, time+expo(1/μ));
  }

    DESTROY(current);

  }while(task != END_SIM);
```

FIGURA 5.22 Algoritmo de cola M/M/1 modo KILL

El número de *eventos* generados por estos modelos sigue siendo muy bajo, tanto que las simulaciones correspondientes no presentan exigencias importantes sobre el *calendario*. Una posibilidad para elevar número de *eventos* generados es apelar a modelos en los que las colas presentadas hasta el momento se repitan un número considerable de veces. Por ejemplo, *Q* colas M/M/1 independientes en paralelo, como se muestra en **FIGURA 5.23**.

Mod	[illegible]	[illegible]	*f*	*r*	*N*	N_m	*U*	U_m	t_{max}
R	0.5	1.0	0.1	0.2	6.3333	6.3666	0.5000	0.5002	10×10^6
R	0.9	1.0	0.2	5.0	14.7332	14.6145	0.9000	0.8996	10×10^6
S	0.5	1.0	0.1	0.2	1.0000	0.9979	0.3333	0.3328	10×10^6
S	0.9	1.0	0.2	5.0	9.0000	8.8861	0.8653	0.8646	10×10^6
K	0.5	1.0	0.1	0.2	0.4944	0.4957	0.2839	0.2842	10×10^6
K	0.9	1.0	0.2	5.0	1.4423	1.4432	0.5769	0.5769	10×10^6

TABLA 5.3 Resultados de Cola M/M/1 con Fallas

En este caso habrá un único *evento* de fin de simulación para todas las colas (END_SIM), más un arribo (ARR), un probable fin de servicio (DEP) y una falla (FAIL) o reparación (REPAIR) por cola, lo que hace un máximo de *3*Q+1 eventos*. Se ensayaron distintas modalidades de falla, tanto en el modo (RESTART, STOP y KILL) como en el hecho de admitir fallas múltiples o de no permitir fallar simultáneamente a más de un servidor, sino restringir las fallas a sólo uno por vez (Cabe acotar que si sólo se admite una falla por vez, el número máximo de *eventos* ya no será *3*Q+1* sino *2*Q+2*). Si bien el número medio de *eventos* puede ser variado a discreción (eligiendo el valor de *Q*), no se aprecia una dispersión que indique una variabilidad importante de ese número. A través de histogramas mostrando la distribución del número de *eventos* planificados, se comprueba que, en cualquiera de los casos, el tamaño del *calendario* se mantiene estrechamente cercano a un determinado valor, dependiente de la elección de valores medios de tiempos entre arribos, de servicio, entre fallas y de reparación.

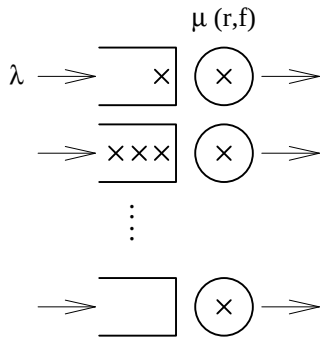


FIGURA 5.23 Q colas M/M/1 en Paralelo

Otra posibilidad es considerar un Tandem de *Q* colas M/M/1 como el de **FIGURA 5.24**, de manera que luego de recibir servicio en un determinado servidor, los clientes pasen consecutivamente por todos los que siguen a continuación. Este modelo admite más variantes que el anterior para el ingreso de tráfico desde el exterior, puesto que bien puede considerarse el ingreso exclusivamente por la primera de las colas del Tandem, o por todas ellas. Asimismo es posible admitir fallas simultáneas o individuales, con la lógica consecuencia en el número máximo de *eventos* generados.

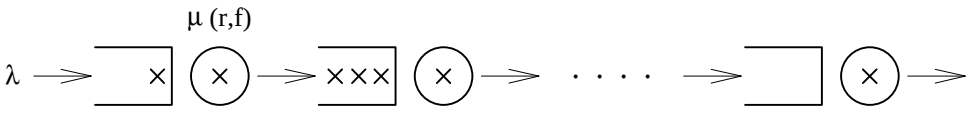


FIGURA 5.24 Tandem de Q colas M/M/1

Existe una variante de este modelo en la que intuitivamente se aprecia que el número de *eventos* ha de sufrir variaciones importantes. Se trata de aquella en la que únicamente arriban clientes desde el exterior en la primer cola del Tandem, siendo además la primer cola la única en condiciones potenciales de fallar. El máximo número de *eventos* en este caso es *Q+3* y está dado por, un fin de simulación (END_SIM), un arribo (ARR) y una falla (FAIL) o reparación (REPAIR) en la primer cola, más un fin de servicio (DEP) por cada una de las colas. Si el tiempo medio de servicios es bajo en relación al tiempo medio de reparaciones, es altamente probable que mientras se encuentre fuera de servicio el primer servidor se desocupen uno tras otro los servidores que están a continuación. Cada servidor que se desocupa es un *evento* DEP que se remueve del *calendario*. Si los valores de los parámetros así lo permiten, podrían incluso desocuparse todos los servidores quedando sólo tres *eventos* en el *calendario*. Luego si el tiempo entre fallas lo permite, podrían volver a ocuparse todos. Resulta evidente que para una adecuada elección de parámetros el número de *eventos* planificados sufre variaciones importantes. Este modelo fue elegido entonces por presentar esta cualidad y además por hacer uso de la función de borrado de *nodos* intermedios, caso que se da, únicamente, cuando ocurre el *evento* FAIL estando el servidor de la primer cola prestando servicio.

Las **FIGURAS 5.25**, **5.26** y **5.27** muestran una serie de histogramas correspondientes al número de *eventos* generados, para *Q=40* y distintos valores de los parámetros λ, μ, *f* y *r*. La indicación porcentual corresponde a la fracción de lecturas en que es detectado el número de *eventos* correspondientes, siendo que se realiza una lectura por cada acceso al *calendario*. Estos gráficos son independientes del tipo de *calendario* ya que sólo registran información relativa a la cantidad de *registros* presentes en el mismo, no obstante el hecho de que distintas implementaciones produzcan el mismo histograma es un elemento de validación muy importante ya que se trata del único ensayo presentado del que participa en forma activa la función de borrado de *nodos* intermedios (*Delete*). Se realizaron también lecturas de tiempo de ejecución para el mismo ensayo, corriendo el programa con *Linked Exact Bottom Up* - BSS y *Splay Tree* como *calendario* respectivamente Los resultados se muestran en la **TABLA 5.4**.

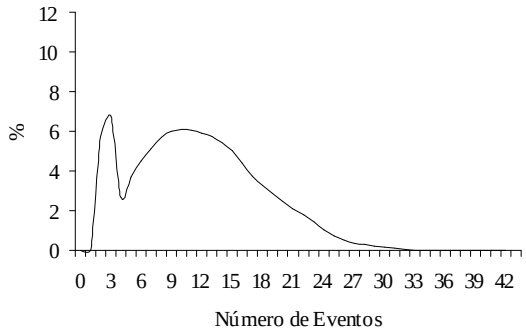


FIGURA 5.25 Distribución de *eventos* λ*=1.0,* μ*=1.0, f=0.1* y *r=0.01*

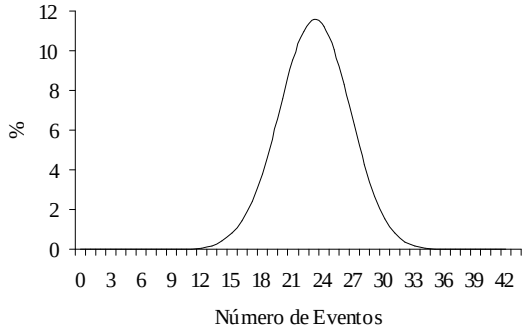


FIGURA 5.26 Distribución de *eventos* λ*=1.0,* μ*=1.0, f=1.0* y *r=1.0*

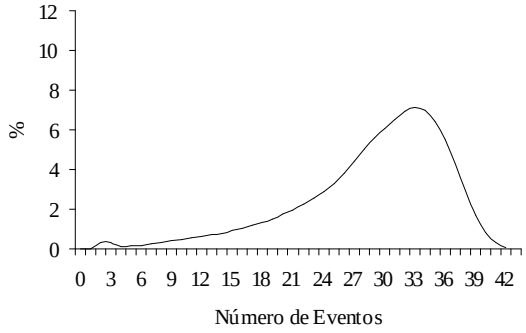


FIGURA 5.27 Distribución de *eventos* λ*=1.0,* μ*=1.0, f=0.01* y *r=0.01*

[illegible]	[illegible]	f	r	t_{max}	***Linked Exact Bottom Up* - BSS**	***Splay Tree***
1.00	1.00	0.10	0.01	100000	2.81 seg.	2.81 seg.
1.00	1.00	1.00	1.00	100000	14.59 seg.	15.00 seg.
1.00	1.00	0.01	0.01	100000	13.82 seg.	14.20 seg.

TABLA 5.4 Tandem con Falla. Análisis Comparativo

Cabe recordar que las dos operaciones fundamentales del *calendario* son *insertar* y *remover*, por lo que, salvo en la simulación de modelos muy particulares la cantidad de borrados de *nodos* intermedios es poco relevante. Esto no quiere decir que no los haya, sino que, en general, su número es muy bajo frente al número de *inserciones* y *remociones*. Tanto es así, que en las evaluaciones de *calendario* únicamente se toma en cuenta el rendimiento de estas dos operaciones. Luego, un *calendario* puede desempeñarse eficientemente en la SED aún si su operación de borrado de *nodos* intermedios es costosa. Por eso en esta Tesis el énfasis fue puesto en las operaciones *insertar* y *remover* y sólo dos de las implementaciones ensayadas cuentan con borrado de nodos intermedios, son estas: *Linked Exact Bottom Up* - BSS y *Splay Tree*. En el caso de *Linked Exact Bottom Up* la programación se realizó de acuerdo a la operación de Reestructuración post-*remoción* explicada en la Sección **3.3.7.4**, mientras que la operación correspondiente para *Splay Tree* es parte del código de [32].

5.3. Metodología de Trabajo

Todos los ensayos de esta Tesis se realizaron en un equipo Pentium-MMX PC, con 32 Mb de memoria RAM y 166 MHz de frecuencia de reloj, bajo Sistema Operativo Linux (Kernel v.2.0.36). El lenguaje de programación fue C y el compilador gcc v.2.7.

En todos los casos se trabajó sobre una interfase común, armada con del siguiente criterio: existe un único programa del modelo a simular en el cual las operaciones de *calendario* son provistas mediante la inclusión del archivo de cabecera "*nombre*.h", donde *nombre* varía de acuerdo a la implementación de *calendario* con que se desee realizar la simulación. A tal efecto las funciones implementadas en el correspondiente archivo "*nombre*.c" son, entre otras, las operaciones del *calendario*. De manera que la única precaución a tener para la operación de la interfase es que las funciones provistas por los distintos archivos de cabecera para la misma operación tengan el mismo nombre, respetándose además tipo y número de todos los argumentos y variables en juego.

Las implementaciones de *calendario* que fueron programadas para ser sometidas a ensayos son:

1) Las siete de tipo BSS presentadas en la Sección **3**.
2) Las dos de *Short-Cut List* presentadas en la Sección **4**.
3) *Linear*, *Linear Median Value* y *Linear Median Pointer* (con sus variantes) presentadas en la Sección **2**.
4) La estructura de *Henriksen*, como adaptación del código en Pascal de [9].
5) *Pairing Heap*, como adaptación del código en C de [30].
6) *Skip List*, como adaptación del código en C citado en [18].
7) *Implicit Heap*.
8) *Skew-Heap (Top-Down)*.
9) *Splay Tree*, como adaptación del código en C de [32].

Ni en los ensayos de "sólo *inserciones*" ni en los de "sólo *remociones*" de la Secciones **3.3.3** y **3.3.4** se consignan los tiempos de operaciones individuales, sino el total que demanda llenar o vaciar completamente el *calendario*. En tanto en el ensayo de "sólo *remociones*" de la Sección **3.3.6** se reporta el tiempo medio de cada *remoción* para cada tamaño de *calendario*, esto es, el tiempo total de vaciado dividido por el número inicial de *registros*. En estos casos la distribución utilizada fue *Exponential* y en cada ensayo se practicaron secuencias de 10^5 *inserciones* y 10^5 *remociones* respectivamente.

Para los tests sobre *The Hold Model* se procedió a llenar el *calendario* hasta el tamaño deseado haciendo uso de la distribución elegida. Luego se practicaron 2×10^5 operaciones de Hold para llevar el modelo al estado estable y finalmente las 2×10^5 operaciones registradas. El tiempo invertido por los algoritmos en tareas no directamente vinculadas a las operaciones de Hold fue descontado.

5.4. Criterios de Validación

Con el objeto de validar la programación de las implementaciones de *calendario* se realizó un agregado de código para que, inmediatamente luego de la ejecución de *The Hold Model*, se vacíe completamente el calendario [28]. Ese proceso de vaciado permite entonces:

1) Contar el número de *registros* retirados de modo de asegurarse de extraer exactamente la cantidad inicial designada para el ensayo. Esta verificación garantiza la no existencia de pérdidas ni repeticiones de *registros*.

2) Comparar los valores de *t* de los *registros* retirados en forma consecutiva, asegurándose de que respetan el orden creciente de *t*.

3) Registrar el valor de *t* del último *registro* removido. El hecho de que para distintas implementaciones de *calendario*, luego de una enorme cantidad de operaciones de *inserción* y *remoción*, el último *registro* removido tenga el mismo valor de *t*, independientemente del *calendario* utilizado, garantiza, al menos, que todas las implementaciones ejecutan las operaciones de un modo idéntico.

El depurado de la programación de algunas implementaciones de *calendario* tuvo como soporte un agregado para mostrar gráficamente (de un modo elemental pero muy útil) la evolución de la estructura ante *inserciones* y *remociones* para números muy reducidos de *registros*.

También se registraron trazas, colocando puntos de lectura en los accesos al *calendario* para relevar, en esos instantes, tanto el tipo de operación realizada como algún parámetro importante (por ejemplo el valor de *t*). Las trazas se conservaron en archivos de texto para realizar luego la comparación entre ellos (de a pares) verificando mediante esta prueba que todas las implementaciones operan en forma idéntica para toda la gama de exigencias de cada ensayo.

5.5. Conclusiones

En la Sección **5** se cubrieron los aspectos comparativos, desde el punto de vista experimental. Para ello primeramente se presentó *The Hold Model*, un modelo ampliamente difundido y aceptado para simulaciones cuya finalidad es evaluar el rendimiento del *calendario*. Dado que *The Hold Model* mantiene constante el número de *eventos* planificados durante toda la simulación y que no realiza operaciones distintas de *insertar* y *remover*, se avanzó también en la búsqueda de un modelo alternativo que permita salvar estas limitaciones. Con ese propósito se desarrollaron algunos *Modelos de Cola*, uno de los cuales resuelve satisfactoriamente el objetivo propuesto.

Los ensayos de las Secciones 5.1.2. (*The Hold Model*) y 5.2 (*Modelos de Cola*) permiten, por un lado, complementar la información de carácter comparativo relevada de la literatura existente y por otro evaluar las implementaciones propuestas frente a las más conocidas y utilizadas. En cuanto a las pruebas sobre *The Hold Model*, la evaluación toma en cuenta el comportamiento de las respectivas implementaciones en el papel más frecuentemente desempeñado, cual es el de *calendario* de SED que únicamente realiza *inserciones* y *remociones*. En este terreno dos de las implementaciones BSS (cuyas curvas fueron resaltadas con trazo más grueso) presentan un rendimiento destacable en rangos intermedio y alto de número de *eventos* para todas las distribuciones, son estas *Linked Exact Bottom Up* - BSS y *Array Bottom Up* - BSS. En el rango de bajo número de *eventos* también se percibe un buen comportamiento para estas implementaciones, pero sólo para algunas distribuciones y allí se verifica también que, cuando el número de *eventos* es extremadamente bajo, para casi todas las distribuciones, la implementación sobre lista simple (*Linear Median Value* en esta caso) es la más eficiente. Las dos implementaciones ensayadas sobre *Modelos de Cola* arrojaron resultados muy parecidos, con una ligera ventaja a favor de *Linked Exact Bottom Up* -BSS por *sobre Splay Tree* (ver **TABLA 5.4**).

Establecer una clasificación de las distintas implementaciones de *calendario* de acuerdo al rendimiento es una tarea dificultosa. Tomando en cuenta las observaciones experimentales así como el análisis de la literatura que reporta pautas de carácter comparativo [8, 13, 14, 20, 21] es posible establecer tres elementos clave en los que radica esa dificultad.

1) *Programa*. El programa que ejecuta una implementación de *calendario* es, quizá, la más crítica de las razones que hacen difícil mantener un ranking estricto entre las distintas variantes. Hay implementaciones cuya programación es extremadamente dificultosa por lo que se presta para ser abordada de muy diversos modos, siendo difícil establecer a priori el más adecuado (casi la totalidad de las implementaciones están desarrolladas en los lenguajes C y Pascal). También hay detalles de los ensayos sobre los que no hay acuerdo ni forma común de plantearlos, por ejemplo el tipo de *nodo* a utilizar. Una opción se limita a considerar únicamente lo mínimo indispensable, esto es, el campo *t* más él o los punteros necesarios. Otras

toman en cuenta la reserva de espacio para información relativa a la simulación, por ejemplo la clase de *evento*, un puntero a la rutina a ejecutar, etc. En definitiva el tamaño de *nodo* elegido tiene una importancia relevante en el tiempo de ejecución y ejerce una influencia desigual sobre las distintas variantes dado que no todas someten los *nodos* al mismo "movimiento" dentro de la estructura. Detalles como este, así como el estilo y la calidad de la programación son bastante determinantes y complican las evaluaciones de propósito comparativo.

2) *Equipo*. Las características y prestaciones del equipo de cómputo en el que se realizan los ensayos también tienen una importancia gravitante, capaz incluso de invertir los resultados. El tipo de procesador, la cantidad y arquitectura de la memoria, el sistema operativo y el compilador utilizados pueden llegar a favorecer algunas implementaciones en perjuicio de otras. El hecho de que sólo sean plenamente confiables los análisis comparativos realizados sobre el mismo equipo torna difícil establecer pautas de elevado grado de generalidad.

3) *Modelo*. Tanto por razones teóricas como experimentales *The Hold Model* es ampliamente aceptado como test de *calendario*. Este modelo varía su comportamiento de acuerdo a la distribución de probabilidad utilizada. Y los resultados experimentales son fuertemente influidos por la distribución elegida, e incluso cambiantes. Más crítico todavía es considerar modelos menos generales que *The Hold Model*, los que podrían incluso ejecutar frecuentemente operaciones como el borrado de *nodos* intermedios, el corrimiento de *nodos* dentro del *calendario*, etc. Para estos modelos especiales los resultados, desde el punto de vista comparativo, son todavía más dispersos y menos predecibles.

En definitiva el parámetro principal a considerar para evaluar las implementaciones según su rendimiento es el número medio de *eventos* planificados al momento de ejecutar las operaciones de interés. Tomando en cuenta este valor hay algunas implementaciones cuya eficiencia les permite escapar a las consideraciones establecidas en los tres puntos anteriores y perfilarse como "las mejores". Se trata de las que se destacan en rangos de tamaño extremo, es decir para números "muy bajos" o "muy altos" de *eventos*. En ese sentido se acepta que cualquier implementación que opere sobre lista simple asistida por *búsqueda secuencial* (por ejemplo, *Linear*, *Linear Median Value* o *Linear Median Pointer*) es la más efectiva si el número de *eventos* es bajo, entendiendo por tal, inferior a *50*. En el otro extremo, si la cantidad de *eventos* es elevada, típicamente más de *10.000*, destacan, tal como es previsible, las que exhiben complejidad cercana a ***O****(1)*, a saber *Calendar Queue* y *Lazy Queue*. La zona más crítica es la comprendida entre estos dos rangos o sea valores intermedios de tamaño. Allí no hay acuerdo generalizado y la influencia de los tres puntos citados más arriba es mucho más marcada. No obstante es posible detectar algunas implementaciones que ante cierta diversidad de ensayos, aparecen repetidamente entre las de mejor performance dentro de ese rango, por ejemplo la estructura de *Henriksen*, *Binomial Queue*, *Splay Tree*, *Linked Exact Bottom Up* y *Array Bottom Up* - BSS, por citar algunas.

De las citadas como más efectivas, todas las implementaciones sobre lista simple (*Linear*, *Linear Median Value* y *Linear Median Pointer*), *Calendar Queue*, *Henriksen*, *Splay Tree* y las dos de tipo *BSS* pueden programarse de modo que resulten *estables*.

En muchos casos el precio a pagar por un buen rendimiento es el de suministrar uno o más parámetros de operación. Esto hace que algunas implementaciones deban reservarse para problemas específicos cuyas propiedades deben conocerse con sumo detalle (casos típicos son *Calendar Queue* y *Lazy Queue*).

6. Conclusiones

El presente trabajo de Tesis tuvo como principal motivación el interés por explorar una componente fundamental de los *Simuladores a Eventos Discretos*, como es el *calendario*. Para ello, por un lado se procedió al análisis de la literatura existente en la cual se reportan tanto evaluaciones analíticas como resultados experimentales que cotejan entre sí diversas implementaciones bajo distintas condiciones de operación, y por otro lado se llevó a cabo un plan de experimentos sobre un conjunto de implementaciones programadas a ese efecto. Dentro de este contexto de trabajo se presentaron dos propuestas propias de implementación de *calendario* y se sugirió un modelo que permite evaluar algunas propiedades interesantes del mismo.

En la Sección **1** de la Tesis se definieron formalmente las operaciones del calendario, se comentaron los problemas que el mismo debe resolver y se anticiparon las principales pautas de trabajo, sobre todo en lo concerniente a la faz experimental.

En la Sección **2** se pasó revista a las principales implementaciones de *calendario*. Se comenzó describiendo veintitrés de las soluciones más difundidas por la literatura y consecuentemente más empleadas en la práctica. Conjuntamente con las soluciones más conocidas se presentaron dos propuestas de implementación propias de este trabajo de Tesis. Una de ellas llamada BSS (*Bounded Sequential Searching*), consiste en un mecanismo de asistencia a las búsquedas secuenciales sobre listas que permite rápido acceso al punto de inserción. De este mecanismo se desprenden siete posibles implementaciones de *calendario*, tres de la cuales exhiben un rendimiento destacable. La otra propuesta propia de la Tesis, llamada *Short-Cut List*, tiene el mismo propósito, en el sentido de acelerar recorridos secuenciales sobre listas enlazadas, pero se basa en un principio totalmente diferente al de BSS.

En la Sección **3** se describió en detalle el mecanismo BSS, su principio de operación y sus principales características, se evaluó la complejidad de alguna de sus operaciones y se sugirieron pautas para su diseño. Se cotejaron experimentalmente entre sí todas las posibles implementaciones y se emitieron algunas conclusiones entre las que destacan: el buen compromiso entre eficiencia y simplicidad del mecanismo BSS y la indicación de las variantes más eficientes (*Linked Exact Bottom Up* para menos de *10.000 eventos* y *Array Bottom Up* y *Array Top Down* para más de *10.000 eventos*).

La Sección **4** se dedicó a *Short-Cut List*. Allí se presentaron las ideas fundamentales de su funcionamiento, se sugirieron algunas pautas tendientes a sustentar su diseño y introdujo una variante (basada en un Arreglo de Cabeceras de lista) que en los ensayos se reveló como más eficiente que la propuesta original.

En la Sección **5** se concentraron los aspectos experimentales de la Tesis. Se comentaron los modelos utilizados en los ensayos, destacándose las propiedades de cada uno y se realizaron pruebas de carácter comparativo. Los experimentos más relevantes se realizaron con *The Hold Model* como modelo, mientras que para salvar algunas limitaciones impuestas por el mismo, se recurrió a *Modelos de Cola*, uno de los cuales (un Tandem de colas M/M/1 con probables fallas en la primera de las colas) resultó muy útil por presentar gran variabilidad en el número de eventos durante el ensayo y por requerir del *calendario* una operación adicional (la *remoción* de nodos intermedios).

Los ensayos realizados muestran que algunas de las variantes de tipo BSS están a la altura de las implementaciones más efectivas. En el caso *Linked Exact Bottom Up* su utilidad se circunscribe, principalmente al rango intermedio de número de *eventos*, no obstante es de destacar su excelente comportamiento para tamaños reducidos y la razón de esta cualidad es clara: mientras el tamaño del *calendario* se mantiene por debajo del límite *B*, la estructura de BSS no es sino una lista simple (la más eficiente para muy pocos *eventos*), con el agregado de una comparación entre el número de *eventos* y el valor de *B* para decidir la primer *inserción auxiliar*. Mientras el tamaño del *calendario* se mantiene en el orden de *B* (unas "pocas veces" *B*), la estructura *auxiliar* es mínima y por ende el costo de las operaciones se mantiene cercano al de las correspondientes sobre lista simple. En otro extremo, *Array Bottom Up* - BSS se torna marcadamente eficiente para número de *eventos* superior a *10.000*, tanto que en ese rango ninguna de las implementaciones ensayadas exhibe tiempos de ejecución más bajos.

A partir de la experiencia acumulada en el desarrollo de esta Tesis es posible emitir una recomendación tendiente a sustentar la elección del tipo de *calendario* para simulaciones de número medio de *eventos* (entre *100* y *10.000*). Si se tata de encontrar una variante que satisfaga necesidades específicas, por ejemplo desempeñarse como *calendario* de un lenguaje de simulación o resolver un problema de características muy particulares, la elección deberá tomar en cuenta esas necesidades, que bien podrían ser la eficiencia en operaciones distintas de *insertar* y *remover*, la adaptabilidad a determinados lenguajes de programación, el buen rendimiento ante distribuciones temporales atípicas, etc. Pero si se trata una aplicación simple, típicamente un

programa de SED desarrollado por un estudiante, un investigador o un programador ocasional, sobre modelos que no generen más que unos miles de *eventos* y el único objetivo es acotar "razonablemente" los tiempos de ejecución, una buena recomendación es optar por la variante que resulte más simple de programar (descartando por supuesto la lista simple que, siendo la más sencilla, no es eficiente en ese rango). La alternativa que se implementa cuando no se tiene en cuenta ningún criterio es casi invariablemente la lista simple en cualquiera de sus variantes, pero para número medio de *eventos* el rendimiento de esta solución está a una distancia tan grande de la siguiente, que a los fines del presente objetivo todas las demás pueden considerarse próximas entre sí. Entonces, implementaciones como *Skew Heap* o *Implicit Heap*, que son las de programación más sencilla y cuyos algoritmos pueden copiarse y/o transcribirse a cualquier lenguaje con mínimo esfuerzo, pueden proveer soluciones aceptables.

Short-Cut List constituye una idea, y debe ser vista más como una línea de trabajo que como una propuesta concreta de implementación de *calendario*. El objetivo perseguido por su mecanismo es simple y claro y el desafío para llevarla a buenos niveles de eficiencia pasa por simplificar su implementación más aún de lo hecho en el presente trabajo. Invertir esfuerzo en acelerar implementaciones eficientes sólo en bajo número de *eventos* pude no tener sentido en aplicaciones sobre equipos mono-procesador pero es un problema crítico en simulación paralela, donde el total de *eventos* planificados se distribuye en tantos *calendarios* como procesadores tenga el equipo.

Otras líneas de trabajo futuro contemplan: un desarrollo completo de ambas implementaciones BSS sobre *Array*, un análisis comparativo entre las distintas implementaciones de *calendario* tomando como medida de eficiencia el espacio de memoria consumido en lugar del costo de las operaciones (ya sea medido a través del número de operaciones elementales o del tiempo de ejecución) y un avance en el terreno de los *Modelos de Cola* en pos de conseguir buenos niveles de variabilidad en el número de *eventos* sobre modelos con menor cantidad de variables para ajustar.

La gran cantidad de trabajo de investigación dedicado al problema de implementar el *calendario* en la SED ha dado origen a una inmenso número de soluciones. Por un lado esto ayuda a percibir lo crítico del problema y por otro lado hace que, llegado el momento de diseñar un *simulador* de elevado rendimiento, se torne inevitable realizar un análisis previo del modelo a simular a fin de elegir una implementación de *calendario* adecuada. El conjunto de opciones entre las cuales es posible elegir presenta alternativas de muy diverso tipo, y con un nivel de complejidad de implementación muy vasto. Las hay desde una simplicidad extrema hasta de una complejidad muy grande, no obstante el nivel de dificultad de las implementaciones no siempre está en relación directa a la dimensión del problema que solucionan ni al rendimiento con que se desempeñan. El tiempo total de ejecución de la simulación es la medida más importante sobre la que una elección apropiada de *calendario* permite influir, y en tal sentido el parámetro más importante para realizar la elección es el número de *eventos* que la simulación puede llegar a generar.

El desarrollo de las *colas de prioridad* y de muchas de las implementaciones de *calendario* tuvo un auge (puesto en evidencia por la fecha de la mayoría de los artículos disponibles) alrededor del año 1980. Con el correr del tiempo, a la dificultad para encontrar soluciones nuevas, radicalmente distintas a las existentes, se sumó el espectacular desarrollo de los equipos de cómputo y de su capacidad de cálculo. A muy bajo costo, prestaciones impensables en otros tiempos, tanto en cantidad de memoria como en velocidad de procesador, hicieron bajar los tiempos de ejecución de todas las aplicaciones, estrechándose por lo tanto sus diferencias relativas. Esta evolución permitió abordar mediante implementaciones "sencillas" problemas que para alcanzar tiempos de ejecución admisibles exigían soluciones complicadas, tornándose entonces menos dramática la necesidad de encontrar algoritmos más rápidos. No obstante, la investigación en este terreno es siempre fértil y provechosa, y por mínimo que resulte el aporte de cada nueva propuesta al momento de su desarrollo, bien podría sentar las bases para futuras variantes, más eficientes o tal vez más adecuadas para la solucionar nuevos problemas.

Referencias

[1] Blackstone J.H., Hogg G.L. y Phillips D.T., "A two-list synchronisation procedure for discrete event simulation". *Comm. ACM,* 24 (12): 825-829 [Diciembre 1981].

[2] Brown, R., "Calendar Queues: A fast O(1) priority Queue implementation for the simulation event set problem". *Comm. ACM,* 31 (10): 1220-1227 [Octubre 1988].

[3] Fishman, G. S., "Discrete-Event Simulation. Modeling, Programming and Analysis". *Springer-Verlag New York, Incorporated* [Marzo 2001].

[4] Francon J., Viennot G. y Vuillemin, "Description and Analysis of an efficient priority queue representation". *Proceedings of the 19th Annual Symposium on Foundations of Computer Science, IEEE Piscataway, NJ,* pp. 1-7 [Octubre 1978].

[5] Franta W.R. y Maly K., "A comparison of Heaps and TL structure for the simulation event set". *Comm. ACM,* 21 (10): 873-875 [Octubre 1978].

[6] Franta W.R. y Maly K., "An efficient data structure for the simulation event set". *Comm. ACM,* 20 (8): 596-602 [Agosto 1977].

[7] Fredman M. y Tarjan R., "Fibonacci Heaps and their uses in Improved Network Optimization Algorithms". *Journal of the ACM,* 34 (3): 596-615 [Julio 1987].

[8] Jones Douglas W., "An Empirical Comparison of Priority-Queue and Event-Set Implementations". *Comm. ACM,* 29 (4): 300-311 [Abril 1986].

[9] Kingston J.H., "Analysis of algorithms for the simulation event list", PhD Thesis. *Basser Dept. of Computer Science, University of Sidney, Australia* [Julio 1984].

[10] Knuth, D.E., "The Art of Computer Programming. Vol 1". *Addison-Wesley, Reading, Mass* [1973].

[11] Knuth, D.E., "The Art of Computer Programming. Vol 3". *Addison-Wesley, Reading, Mass* [1973].

[12] Luckow M. Y Müller N., "Cascade: A simple and Efficient Algorithm for Priority Queues". *Technical Report D-54286, Trier University* [1994].

[13] Marín M., "An Empirical Comparison of Priority Queue Algorithms", *Technical Report PRG-TR-10-97, Oxford University* [1997].

[14] Mc Cormack W y Sargent R., "Analysis of Future Event Set Algorithms for Discrete Event Simulation". *Comm. ACM,* 24 (12): 801-811 [Diciembre 1981].

[15] Mitrani I., "Simulation Techniques for Discrete Event Systems". *Cambridge University Press* [1982].

[16] Murray L., Rubino G. and Urquhart M., "BSS Priority Queue", *Technical Report INCO 02-13, Facultad de Ingeniería, UDELAR, Montevideo, Uruguay* (ISSN: 0797-6410) [2002].

[17] Papadakis Thomas, "Skip Lists and Probabilistic Analysis of Algorithms", PhD Thesis. *University of Waterloo. Waterloo, Ontario, Canada* [1993].

[18] Pugh W., "Skip Lists: A probabilistic alternative to balanced trees". *Comm. ACM,* 33 (6): 668-676 [Junio 1990].

[19] Röngren R., Riboe J. y Ayani R., "Lazy Queue: A new approach to implementing the pending event set". *Int. J. Comput. Simul.* (3): 303-332 [1993].

[20] Röngrenn R. y Ayani R., "A Comparative Study of Parallel and Sequential Priority Queue Algorithms". *ACM Transactions on Modelling and Computer Simulation (TOMACS),* 7 (2): 157-209 [Abril 1997].

[21] Röngrenn R., Riboe J. y Ayani R., "A Comparative Study of some Priority Queues Suitable for Implementation of the Pending Event Set", *Proceedings of the 1993 Western Multi Conference on Computer Simulation, San Diego, CA* [Enero 1993].

[22] Rubino G., "Modelos Avanzados de Colas para Redes de Comunicaciones". *Curso Escuela de Ciencias Informáticas (ECI), Universidad de Buenos Aires* [Julio 2001].

[23] Sedgewick R, "Algoritmos en C++". *Addison-Wesley / Diaz de Santos, Wilmington, Delaware.*[1995].

[24] Sleator D.D. y Tarjan R.E., "Self-Adjusting binary search trees". *Journal of ACM*, 32 (3):652-686 [Julio 1985].

[25] Sleator D.D. y Tarjan R.E., "Self-Adjusting heaps". *SIAM J. Comput.*, 15 (1): 52-69 [Febrero 1986].

[26] Stasko J.T. y Vitter J.S., "Paring heaps: Experiments and Analysis". *Comm ACM*, 30 (3): 234-249 [1987].

[27] Tenenbaum A., Langsam Y. y Augenstein M., "Estructuras de Datos en C". *Pearson - Prentice Hall* [1993].

[28] ftp://ftp.cs.uiowa.edu/pub/jones/event/cacm.txt.Z (Varias implementaciones, código en Pascal) [18/02/03].

[29] http://lcm.csa.iisc.ernet.in/dsa/node140.html (Binomial Queue) [18/02/03].

[30] http://www.cs.fiu.edu/~weiss/dsaa_c2e/files.html (Pairing Heap, código en C) [18/02/03].

[31] http://www.cs.mcgill.ca/~sperez/doc4.html (Priority Tree) [18/02/03].

[32] http://www.cs.uiowa.edu/~jones/compress/ (Splay Tree, código en C) [18/02/03].

[33] http://www.dgp.toronto.edu/people/JamesStewart/378notes/10leftist/ (Leftist Tree) [18/02/03].

[34] http://www.nasatech.com/Briefs/Jan99/NPO20095.html (SPEEDES) [18/02/03].