A DYNAMIC AND EFFICIENT PARALLEL INDEX STRUCTURE FOR TEMPORAL DATABASES

By

HANI HAFEZ

Bachelor of Science in Engineering

Alexandria University

Alexandria, Egypt

1979

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1996

A DYNAMIC AND EFFICIENT PARALLEL INDEX STRUCTURE FOR TEMPORAL DATABASES

Thesis Approved:

H. Lu

Thesis Adviser

Jacques E. LaFrance

N. E. Hel

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere gratitude to my major adviser, Dr. Huizhu Lu for all her effort, invaluable guidance, support, patience, and friendship. I also wish to express my deep appreciation to my committee members, Dr. G. E. Hedrich, and to Dr. Jacques LaFrance for their experienced advice, support, and encouragement.

I would also like to express my sincere appreciation to Miss Elaine Burgess, coordinator of the international services office for her enormous support, precious friendship, and for offering me an assistantship.

Very special appreciation goes to my parents, Mr. and Mrs. Hafez, and to my wife Manal for their support, sacrifice, and understanding during my study for this degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A conventional database can store only one version of information, because the new data always replace the old one in an update operation. Despite the importance of past data, it was discarded, because it was not economically possible to keep it. However, with the development of newer computer systems, and specifically storage devices, emerged the development of temporal databases. According to Ling (1988), a temporal database maintains past data, while new data are appended to it.

Ling (1988) defined a temporal model by three main features: (1) A conceptual theoretical representation. This is mainly concerned with the time attributes supported by the model, and how these time attributes are represented. (2) A temporal query language. This is the communication tool between a user and the model. (3) A physical representation. This addresses the implementation aspect of the model. Thus, this feature mainly considers the storage structures required for implementing the temporal model.

There can be two main approaches to the implementation of a temporal model. The first approach is designing a complete new model. The second approach is based on extending an existing model. Most proposed temporal models, e.g. Gadia (1993), Dayal (1992), and including our research, belong to the second approach. The reason behind this is that it is not practical nor economical to replace existing models that handle nontemporal data with new data models that handle temporal and nontemporal data.

The storage structure is not only the core of the physical representation of a temporal model, but also the most difficult part. Conventional database indexes, like the B-tree

presented in Comer (1979), cannot be directly applied without modification to temporal databases for three major reasons: (1) Temporal databases tend to be very large. (2) The domains of temporal queries' semantics are large. (3) Temporal data are mostly represented as intervals, which cannot be ordered like linear dimensional data. Consequently, efficient handling of temporal data, implies the use of indexes designed specifically to meet the requirements of temporal databases.

Since the choice of indexing techniques is of paramount importance for temporal databases, much research has been focused on the design of new access methods for temporal databases, Lu (1993), Elmasri (1990), Gunadhi (1993), and Kolovson (1993). According to literature, one area where research was very few, is parallel processing support for temporal databases. Although multiprocessor architectures have been widely used in practical applications, very few researchers have discussed temporal databases in a multiprocessor environment, e.g. Bassiouni (1990), and Leung (1992). Although in general these few researchers targeted toward improving the performance of temporal query processing, their research was focused in one of two main areas. The first area was concerned with improving query translators in order to be able to handle temporal and nontemporal queries. The second area dealt with improving response times by applying data fragmentation strategies.

The objective of this research is to develop a new index structure that is capable of efficiently handling a valid time temporal database in a parallel processing environment. In particular, this new mechanism aims at improving the response times for processing temporal queries through the utilization of several identical shared memory parallel processors. Since the size of data in temporal databases is very large, improving the space

complexity is also a vital issue in our research.

In this research, a new dynamic index structure called the C-tree is proposed to handle temporal data in valid time databases. It is based on the relational data model with tuple timestamping. The authors propose a new coding system which is targeted toward improving search performance in large temporal databases. The new coding system efficiently handles simple queries. A new parallel processing scheme is also proposed to improve response time for simple and complex range queries. The scheme is based on the decomposition of complex range queries into three different types of simple range queries. The resulting simple queries are handled by parallel C-trees, such that each type of simple query is handled by the appropriate type of C-trees. For optimized performance, each C-tree occupies a processor. The parallelization is enhanced by vertical, and horizontal data partitioning. An important advantage of the new parallel processing scheme is that any complex range query can be processed in a time close to the time of processing a simple range query. Another advantage is that only C-trees of one type required for processing a query can be loaded to main memory, while C-trees of the other two types may be left on the secondary storage device.

In the next chapter, several index structures for temporal databases are reviewed from the literature. The C-tree concept, its directory organization, and the parallel processing scheme are presented in chapter III. Algorithms for update operations are presented in chapter IV. Evaluation of the performance of the new indexing mechanism, comparison with other temporal indexes, and conclusion are given in chapter V.

CHAPTER II

LITERATURE REVIEW

Recently computer science research related to temporal databases have been focused on introducing new indexing structures that are specially designed to handle temporal data. Several indexes have been currently proposed, e.g. Lu (1993), Elmasri (1990), Gunadhi (1993), and Kolovson (1993). These temporal indexes have four common features : (1) They adopt the view of time as equidistant discrete points in the set {0, 1, ..., now}, where now denotes the changing value of the current time. This implies the selection of some granularity, e.g. day, year, ..etc. (2) They represent a temporal relation by a set of intervals in the time dimension, where each interval is specified by a starting and an ending point. The value of an object is assumed to be constant during an interval. (3) Updates occur in append mode, keeping past versions in the database, thus deletions do not generally occur. Insertions of new object versions occur mostly in increasing time.

Currently proposed temporal indexes can be classified by the kind of time they support. For example, the time split B-tree (TSB-tree) presented in Lomet (1993), supports transaction time, while the time polygon index (TP-index), in Lu (1993), supports valid time. However, some indexes support both valid and transaction times, e.g. the append-only tree (AP-tree) in Gunadhi (1993). In the following we discuss some examples of temporal indexes that support at least valid time, since this is the area closely related to our thesis. Also, since many temporal indexes were designed using spatial indexing concepts, some related spatial indexes are also reviewed.

The time index, introduced by Elmasri (1990), is one of the basic indexing techniques

for temporal data. It was designed to deal with queries relating to starting or ending time, e.g. "Retrieve all tuples whose starting time is between time a and time b." In addition to the regular attributes, each record has an additional interval attribute called valid time. The time index supports object versioning, thus records are used to store more than one version for one object. The versions of an object are linked to the current version. Since older object versions are never deleted, the index supports append-only databases. The main idea of the time index is to maintain a set of linearly ordered indexing points on the time dimension. An indexing point is created where a new interval starts or terminates. A regular B+-tree, in Comer (1979), is used to index the totally ordered indexed points. Each entry in a leaf node at point T has the form [T, bucket] where bucket is a pointer to a bucket containing pointers to object versions. In temporal databases it is highly likely to have a large number of object versions , and many of these will be repeated from the previous indexing point. To reduce this duplication only the leading entry of a leaf node includes all object versions, while a nonleading entry in a leaf node keeps only the incremental changes. According to Lu (1993), these duplications may still decrease space utilization and query efficiency.

Elmasri (1991) introduced three variations of the time index. One proposed variation called the separate variation separates the incremental and decremental pointers into separate buckets rather than being stored in one bucket called the incremental bucket in the original time index. This improves the search time for an interval query. Additional storage space is needed but not substantial. Other two variations were proposed called variation 1, and variation 2. These two variations were based on the idea that for a given particular time interval, object versions can be classified into six groups according to the relation between the starting, and ending points of each object version, and the particular interval. In the two

variations three buckets -representing three of the classified groups- are associated with each leaf node. In this way duplications are reduced. The difference between the two variations is in the choice of the class of object versions to be stored in each of the three buckets. For the time index variations, the search time measured in number of total blocks accessed, while changing the average number of versions per object was almost the same for the three variations, and in general better than the original time index. Under the same conditions variation 2 required the least storage cost.

In the time polygon index (TP-index), presented by Lu (1993), temporal data are viewed as spatial objects in a multidimensional space. Each temporal interval is mapped into a data point in a two-dimensional temporal space. Queries are translated to regions in the space. Data points are grouped such that each group is stored in a page, and search time is reduced. Analysis of temporal queries showed that partitioning the space into rectangles is not ideal for temporal queries. The partitioning strategy uses polygons with similar shapes to the query regions in order to improve the hit rate. A set of five well-formed shapes were defined. A restricted partitioning policy that maintains the polygon shapes to the well-formed ones was used. The structure of the used index is similar to a B+ tree. An entry in a nonleaf node has the form [child-pointer, polygon], where child-pointer points to a child node and polygon describes the space of the child node. An entry in a leaf node has the form [bucket-pointer, polygon] where bucket-pointer points to a data bucket where data points are stored. Polygon refers to the polygon containing the data points. The root of a TP-index has at least two children, unless it is a leaf node. Each node has between m and M entries, unless it is the root. The polygons included in one node do not overlap. To avoid poor storage efficiency, the index allows more than one entry to point to the same bucket. Although the TP-index is

not a height balanced tree, only one path is traversed to search for one data point. When the current time increases from now1 to now2 the temporal space expands.

The append only tree (AP-tree), presented by Gunadhi (1993), uses time-interval representation. It assumes that all relations are in first temporal normal form (1TNF), which means that for each combination of surrogate, and time attributes, there is at most one temporal attribute value. The structure of the tree is such that the leaves contain the starting times. The leaf pointers point to tuples with the same starting time. Access to the tree is either through the root or through the right-most leaf. Insertions are done rapidly by accessing the rightmost leaf. No splitting of nodes is needed. If the node is full, a new leaf is created to its right. In the worst case, a new node is created from the root to the leaf node, and may be a new root is created. In case of queries based on the time attributes, search is done using the index, i.e. starting from the root. In case of queries based on the surrogate, forward scan may be used, i.e. accessing the tree through the rightmost leaf. Deletions from the AP-tree require reconstruction of the tree to maintain balance, but this drawback does not affect the index performance because deletions rarely occur in an append-only database.

The nested surrogate & time (ST-tree), explained by Gunadhi (1993), is a variation of the AP-tree, designed to mainly answer queries where the primary qualification is on the surrogate. The structure consists of nested trees, in which the index in the first level is a modified B+ tree, while the index in the second level is an AP-tree. The insertions are easily carried out, and balancing of one level can be separated from the other.

Two other variations of the AP-tree were proposed by Gunadhi (1993). The first is the composite index, in which the key is formed by the concatenation of the surrogate and the time attribute. The structure used a B+ tree. The other variation is called the sparse tree,

which indexed only the surrogate values. The time attributes were stored in a list that could be accessed from leaf nodes. This list consisted of a time value, and a tuple pointer.

For indexes based on the AP-tree, a comparison between the three variations based on the AP-tree was conducted. The results showed that current query processing is cheaper using the nested and the sparse index, than the composite index. For arbitrary queries involving historical data, the performance changed with the relation size. For storage cost, the composite approach on the average was the most expensive. The sparse index showed the best storage efficiency.

According to Leung (1992), the first published paper related to using multiprocessors with temporal databases was by Bassiouni (1990). It introduced the TRRDS (temporal RRDS) model, which was built on top of the RRDS (relational replicated database system) model. Bassiouni used a separate B+ tree for each attribute in the database. The temporal attributes, Start and End were also handled by two B+ trees. The focus of his research was on translating temporal queries to nontemporal queries that can be handled by the RRDS.

Among the very few recent research supporting multiprocessors in temporal databases is by Leung (1992), who proposed a parallel processing strategy for temporal query processing. Their proposed strategy used a data model in which time points were regarded as natural numbers {0, 1, ...,now} and are monotonically increasing. A temporal relation was denoted as X (S, V, TS, TE) where S is the surrogate, V is a time varying attribute. The lifespan of a tuple is denoted by the interval [TS, TE]. To facilitate query processing of complex temporal joins in a multiprocessor environment, they suggested a data fragmentation strategy based on range partitioning the timestamped values along the time dimension. The number of partitions must be at least as the number of processors. The

partitioning may be based on TS or TE value. Each partition is assigned to a processor. The strategy included three phases. (1) Replication Phase: The state information based on stream processing is constructed for every partition. The state information is a summary of the history of a computation on the portion of data streams that have been read, e.g. copies of tuples for join operations, or partial sum for aggregate functions. (2) Join Phase: Each processor executes the query using its local relation fragments and the constructed state information. (3) Merge Phase: All the results are merged and duplicates are eliminated.

Chaudhry and Elmasri (1994) proposed declustering a temporal index in a single processor, but multiple independent disk architecture. In their research, they employed the time index. They suggested declustering the temporal data, such that it can be accessed simultaneously on multiple disks and thus parallelizing the I/O operation, and therefore improve query processing efficiency.

Kolovson and Stonebraker (1991) introduced a family of indexes called the segment indexes. Segment indexes combined the memory resident segment tree, explained in Kolovson (1993), with multiway trees based on paging. The segment index was designed to index interval, or point data. The main features of the design were as follows. (1) The index records could be stored in leaf or nonleaf nodes. (2) The index node size may vary. These features meant that by estimating the input data distribution, the index may be initially built then dynamically adopted to the input date. This last feature was called skeleton indexing. An example of a segment index is the SR-tree which combines the memory resident segment tree with the R-tree introduced by Guttman (1984). An example of the skeleton index is the skeleton SR-tree presented in Kolovson (1991). The skeleton segment indexes showed substantial improvement in search operations. However, they are only

suitable for uniform, or known data distributions.

Among the efficient attempts to parallelize spatial indexes, is the multiplexed R-tree (MX-R-tree) presented in Faloutsos (1992). It was proposed to maximize parallelism for large range queries, and minimize the number of accessed disks for point queries. The design is based on an R-tree, with several disks attached using cross-disk pointers, in a uni-processor environment. A strategy called the proximity index is used to select a disk for a new node. The proximity index checks for similarity between the new node and other R-tree nodes already on a disk. The disk with the least similarity is chosen. The main two reasons for using multiple disks are: the average disk access time is much bigger than the processing time of a page in memory, and the size of the data may be very large to fit on one disk. In the MX-R-tree, the root node is kept in main memory, while other nodes are on disks. Results showed that it improved response time for large queries. The uni-processor architecture was chosen to avoid communication costs of multiprocessors.

The literature review indicates that:

* The full potential of a temporal database can be achieved by employing indexing structures that are specially designed to provide efficient storage space, and query processing for temporal data.

* Although there has been increasing interest in general in multiprocessor machines, very few researchers discussed parallel processing support for temporal databases, despite the fact that temporal databases are I/O intensive, and the search operations are mostly range queries on the time dimension.

* Temporal databases can benefit from multiple disks, because of the quick growth in size of temporal data, and to parallelize the I/O operations, and thus improve query processing.

CHAPTER III

C-TREE CONCEPT AND DIRECTORY ORGANIZATION

In the following the concept of the C-tree, its structure, and the space decomposition technique are presented.

C-TREE CONCEPT

The concept of interval-spatial transformation was successfully applied in the TP-index presented Lu, Ooi, and Shen (1993). In this thesis, the interval-spatial transformation concept presented by Lu, Ooi, and Shen is extended to support parallel C-trees.

According to the basic interval-spatial transformation, each tuple in the temporal data can be mapped into a point in a two dimension isosceles triangular space. The X dimension of the space represents the event time, while the Y dimension represents the interval of the event. The hypotenuse of the triangle intersects with the X dimension at point (now, 0), and with the Y dimension at point (0, now), where now is the current time. In this thesis, the hypotenuse of the triangular space is denoted by the front line. As the current time progresses, the front line moves along, expanding the temporal space. Consider the students relation in table 1. This relation keeps record of students' beginning of study, and finishing dates at a school. Each tuple in the database is mapped into a point in the temporal space using the interval-spatial transformation as shown in figure 1.

The main advantage of the interval-spatial transformation concept, is that it allows temporal queries to be represented by spatial regions in the temporal space. For instance, the answer to the query that asks for all students who began studying at the school on or before

Student Id.	Student Name	Study Period [From - To]
t1	Kung, T.	[0 - 3]
t2	Fred, J.	[0 - 4]
t3	Bently, G.	[1 - 2]
t4	Carey, L.	[2 - 4]
t5	Smith, C.	[3 - 7]
t6	Miller, O.	[3 - 5]
t7	Stout, K.	[4 - 7]

Table 1 Students Relation

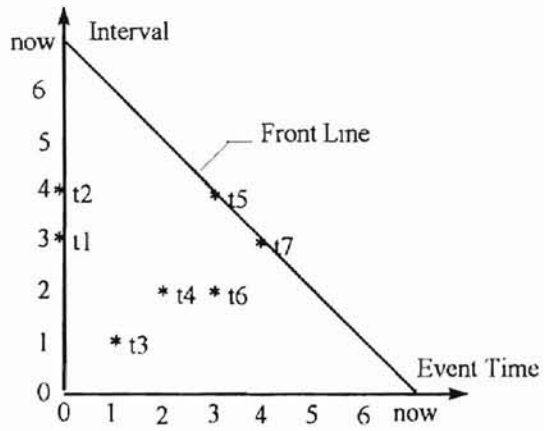
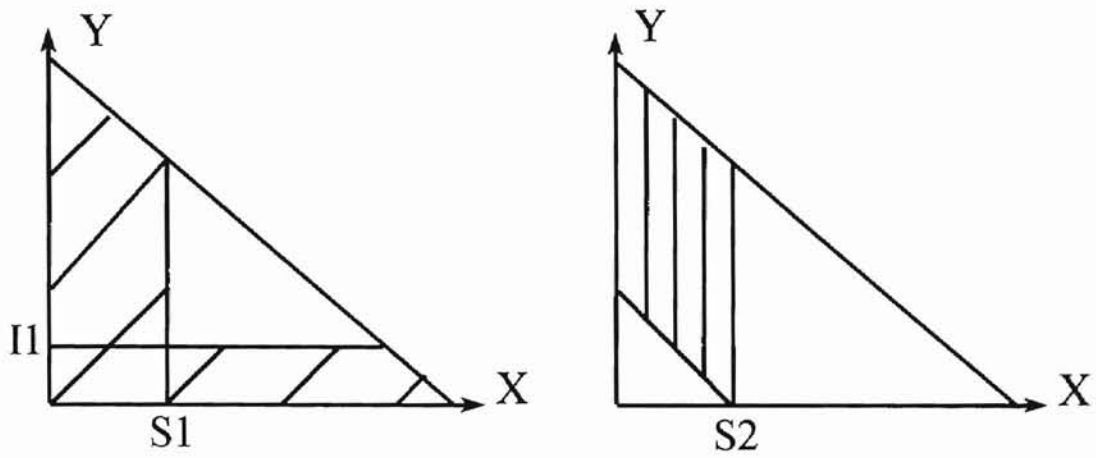


Figure 1 Interval-spatial Transformation of the Students Relation.

day S1, or have studied at the school for I1 days or less, corresponds to the shaded regions in figure 2a. Another query that asks for all the school students as of day S2, is answered by the shaded region in figure 2b.

For parallel processing of temporal queries, this thesis extends the basic interval-spatial transformation. The extension is based on the following important observation. A range query is represented in the temporal space by the use of at most three types of

decomposition lines:



(a) Regions Corresponding to the First Query.

(b) Region Corresponding to the Second Query.

Figure 2 Examples of Mapping Queries to Regions in the Temporal Space

-Type "one" lines: include lines parallel to the X-axis.

-Type "two" lines: include lines parallel to the front line.

-Type "three" lines: include lines parallel to the Y-axis.

Based on the above classification, a complex range query is defined as a query that is represented in the temporal space by using more than one type of decomposition lines. Meanwhile, a simple range query is defined as a query that is represented by only one type of decomposition lines. Accordingly, three types of C-trees are defined, namely types "A", "B", and "C." A C-tree of type "A" is a C-tree that decomposes the space by type "one" lines. A C-tree of type "B" is a C-tree that decomposes the space by type "two" lines. A C-tree of type "C" is a C-tree that decomposes the space by type "three" lines. Consequently, each type of C-trees is appropriate for handling a special type of simple range queries. The combination of types "A", "B", and "C" can efficiently handle complex range queries.

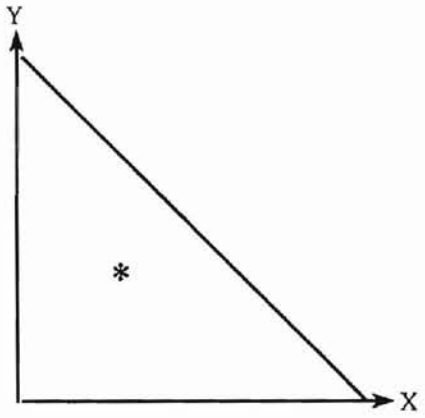
C-TREE STRUCTURE AND SPACE DECOMPOSITION STRATEGY

A C-tree is defined as a height balanced tree with the following properties. All leaf nodes are at the same level. The first node in the tree is the root. Any node that is not the root points to its parent. A node in the C-tree has a maximum capacity of M entries. Each entry stores a string of symbols called the address expression, a pointer, and an integer or a floating number called the range field.

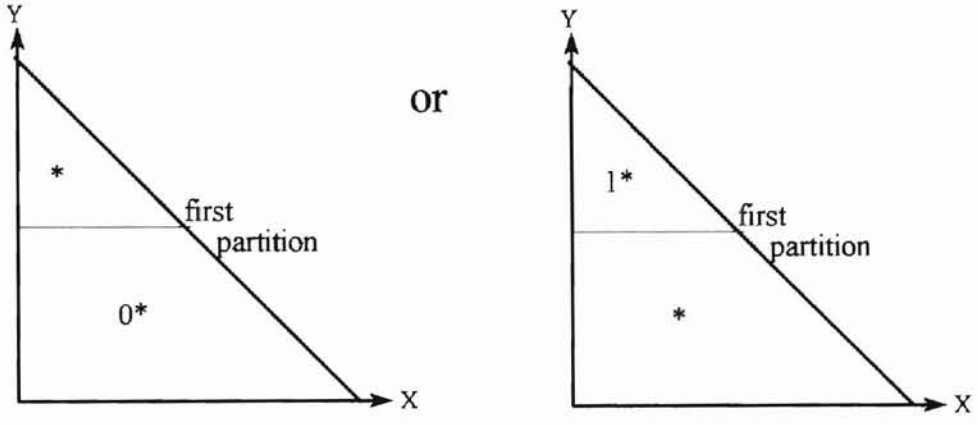
An address expression is a string of symbols. The set of symbols from which an address expression is formed is {0, 1, D, + *}. An address expression describes a region in the space. It gives the path to the region, and the lines bounding the region. The " 0 " symbol refers to a region below, or on the left of a decomposition line. The " 1 " symbol refers to a region above, or on the right of a decomposition line. The " * " symbol with no preceding symbols, refers to the whole space. The " * " symbol at the end of a sequence of symbols indicates the end of an address expression. Figure 3 illustrates examples of address expressions It shows the address expressions of the regions resulting from a single space decomposition in a C-tree of type "A." It can be observed from figure 3 that the address expression of a partitioned region dictates the address expressions of the resulting two subregions. One of the two subregions -usually the one with more data points- is given a new address expression according to its position from the decomposition line, while the other region keeps the same address expression of the overflowing original region. For instance, the region " * " can be decomposed into " 1* " and " * ", or it can be decomposed into " 0* " and " * ". Similarly, the region " 0* " can be decomposed into " 00* " and " 0* ", or into " 01* " and " 0* ."

The other symbols, which are the " + " symbol, and the " D " symbol, are considered special purpose symbols. The " + " symbol is used to concatenate two address expressions,

e.g. 10* + 01*. The " D " symbol is the only symbol that can exist after the " * " symbol. It indicates a special case of space decomposition. This special case takes place if the regular space decomposition cannot be performed because all the points in the region are on one straight line parallel to the direction of partitioning. Further elaboration on the use of the special purpose symbols is included in the update algorithms in chapter IV.



(a) Empty Space



(b) The Space after the First Partition

Figure 3 Examples of Regions Address Expressions

The region described by the address expression of a leaf node entry is the region that includes the data points in the bucket of that entry. The region described by the address expression of a nonleaf node entry includes the regions stored in all the occupied entries of the child node. The address expression stored in a nonleaf node entry, is the same address expression of the rightmost entry of the child node. The address expression of the rightmost entry of a node has to be either an asterisk " * ", or a prefix (not including the " * " symbol) in all the address expressions in this node. If the address expression of a region is a prefix in the address expression of another region, then the first region includes the second region For instance, let region R1 have the address expression " 00* ", and let region R2 have the address expression " 001* ." Then region R1 includes region R2, because the address expression " 00* " is a prefix in the address expression " 001* ."

The length of an address expression is the number of symbols contained in it, up to and including the " * " symbol. The length of the longest address expression in the tree is dynamically stored in the root node. Thus, it is updated with every insertion of a new data point in the temporal space. The use of an address expression, rather than corner points of a region, eliminates complicated computations during search operations.

The pointer stored in an entry of a nonleaf node points to a child node, while the pointer of an entry in a leaf node points to a bucket that includes data points. The range field in each entry is used to store an integer or a floating number. It is associated with the use of the special purpose symbol " D ." In other words, if the address expression of an entry contains the " D " symbol, then the number stored in the range field has to be examined to determine whether the point can be present in this entry or not. More elaboration on the use of the range field can be found in the update algorithms presented in chapter IV.

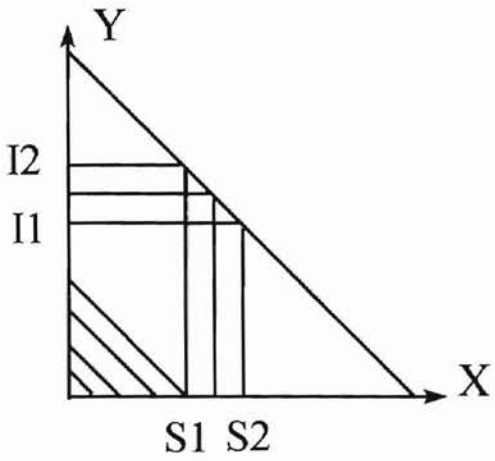
THE PARALLEL PROCESSING SCHEME

The parallel processing scheme is based on the decomposition of complex range queries into simple range queries. Simple range queries are then handled by parallel C-trees, that operate on vertically and horizontally partitioned temporal data. Finally the results are merged to provide the answer to the complex query.

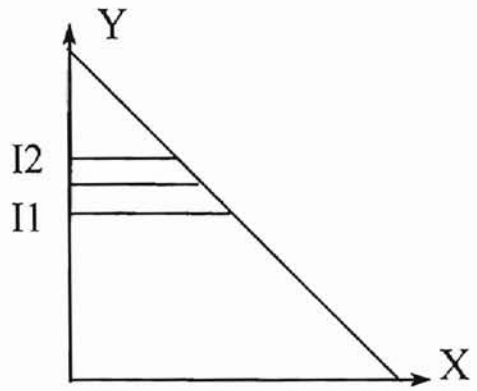
Data partitioning or fragmentation is a common feature among distributed databases, in which data reside at different sites of a network. According to Meghini and Thanos (1993), horizontal data partitioning or fragmentation is based on selecting subsets of tuples from a global relation. This selection may be done according to one or more criteria. In vertical data fragmentation, the fragments are attributes of tuples. In our proposed parallel processing scheme, data fragmentation is applied to a very large centralized temporal database. A centralized database does not incur the overhead cost of communication between sites.

Figure 4 illustrates an example of decomposing a complex range query into simple range queries. The complex query represented by the shaded areas in figure 4a asks for the students who joined the school in the period between day S1 and day S2, *or* who studied at the school for a period greater than I1 days and less than I2 days, *or* who left the school on or before day S1. The complex query is decomposed into the simple queries represented by the shaded regions in figures 4b, 4c, and 4d. Thus, any point in the space can be assigned three values x, y, and j, as shown in figure 5. The x and y values of a point are equal to its X and Y coordinates in the space, respectively. The j value is calculated from the formula: (x^2 + y^2) ^ 1/2. Based on these three values, the temporal data set is partitioned into vertical subsets, such that each subset contains one value for each data point. Accordingly, each subset is handled by one type of C-trees, where the C-trees of type "A" store the y values of the

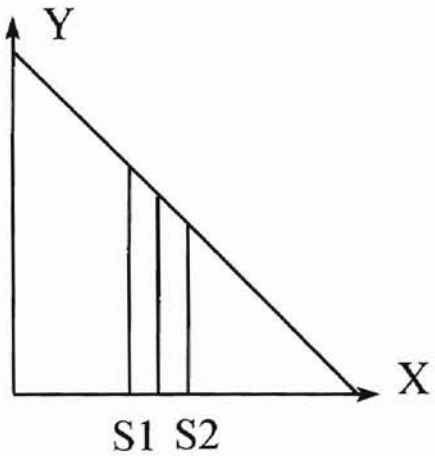
temporal data, while C-trees of type "B" store the j values, and C-trees of type "C" store the x values. For each data point, each of the three assigned values, is linked with the other two values via pointers. The three types of C-trees are co-operated in parallel, to answer complex range queries. For a simple query, one appropriate C-tree type is selected.



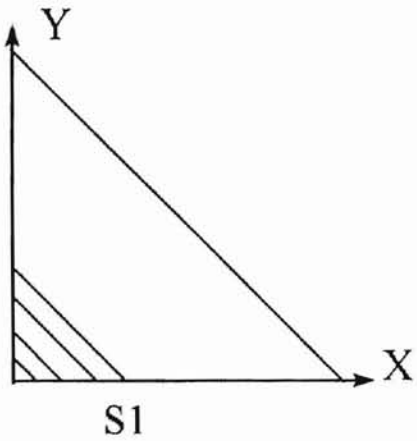
(a) Example of a Complex Range Query.



(b) A Simple Range Query that Requires only the Y-value of Each Data Point.



(c) A Simple Range Query that Requires only the X-value of Each Data point.



(d) A Simple Range Query that Requires only the J-value of Each Data Point.

Figure 4 Example Illustrating the Decomposition of a Complex Range Query into Simple Range Queries.

In order to increase the number of parallel operations as much as possible, while keeping even loading balance, horizontal data partitioning is also employed as follows. Since the temporal space expands with time in a well defined direction, a level granularity is defined as the fixed progressive step of the temporal space. Also, a time level is defined as the portion of the temporal space associated with each level granularity, as illustrated in figure 6. This means that the whole temporal space consists of time levels that can be numbered starting with zero, and ending with the number that corresponds to the current time level, as illustrated in figure 6. Accordingly, the records in the temporal data set are partitioned into horizontal subsets, such that each subset contains the records of one time level. Each subset, or each time level, is handled by three C-trees of types "A", "B", and "C."

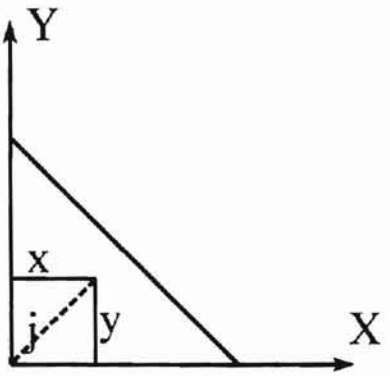


Figure 5 The Three Values Assigned to Each Data Point..

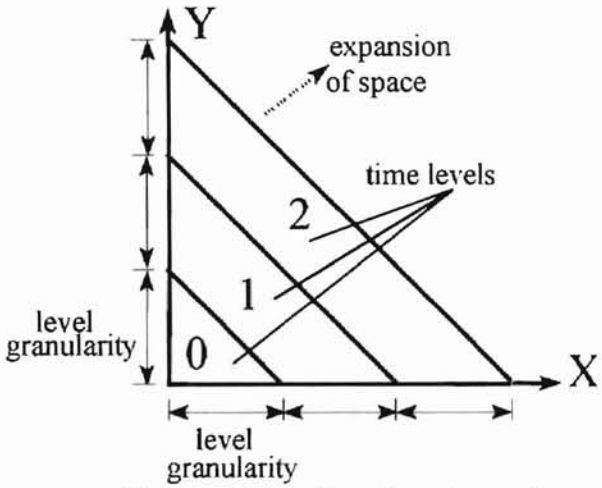


Figure 6 Time Levels and Level Granularities.

The organization of the parallel processing scheme is illustrated in figure 7. It consists of mainly two parts:

1. A sorted array representing the dynamic time levels.

2. Parallel C-trees, such that each time level points to the roots of three different types of C-trees.

The above organization has the following advantages:

* The sorted array of dynamic N time levels can be searched using the concept of binary search in O(log N) .

* The search in the C-trees can be performed in parallel, where the search time in a C-tree, which is a height balanced tree of n nodes, is O(log n) for one access.

* Individual C-trees corresponding to different time levels can be easily isolated for maintenance from the index without affecting the rest of the C-trees.

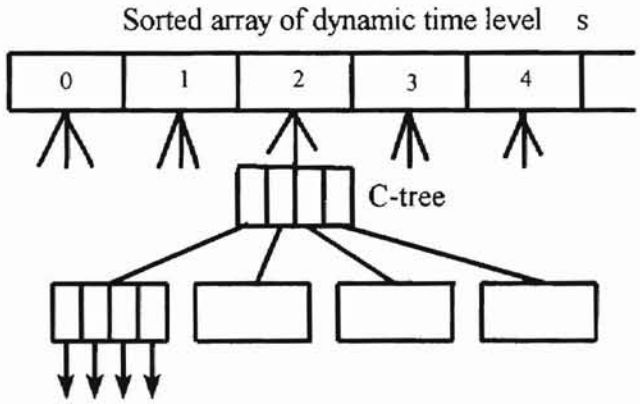


Figure 7 Organization of the Parallel Indexing Scheme

CHAPTER IV

UPDATE OPERATIONS

In the following, the main algorithms for search, insertion, deletion are presented. An illustrative example is also given at the end of this chapter.

SEARCH

The search operations are classified into point query search, and range query search.

Algorithm: Point_search

This algorithm searches for a single point in a C-tree.

Input: A pointer to the root of a C-tree, the search point coordinates (x, y, j) in the temporal space.

Output: [A pointer to a node, the entry number where the point was found] or [NULL].

1. Invoke procedure *Find_time_level*, which returns the time level L of the search point.

2. Invoke procedure *Point_add_exp*, which returns the address expression AE of the region in the space that may include the search point.

3. Let pointer S point to the root of the tree.

4. Search the node pointed to by S, starting from the leftmost entry and toward the right, to find the first entry E whose address expression is a prefix in AE. If the address expression of entry E includes the "D" symbol, then examine the number stored in the range field of E. If the search point lies in the stored range, then goto step 5. Else if not on the range, continue the search toward the right to find the appropriate entry E.

5. If pointer S points to a nonleaf node, let S point to the child of E. Goto step 4.

6. Otherwise, search the bucket of E. If the search point is found, return the entry number E, and a pointer to the node of E. Otherwise, return NULL.

Algorithm: Region_search

This algorithm handles a simple range query by a single type of C-trees as follows. As explained earlier, a simple range query is represented in the temporal space using one type of decomposition lines. This means that a simple query can have a triangular shape with three corner points or a polygon with four corner points. Figure 8 illustrates an example of a simple query region whose corner points in time level "0" are (a, b, f, e) and its corner points in level "1" are (e, f , c, d. The corner points of the whole simple query region are (a, b, c, d).

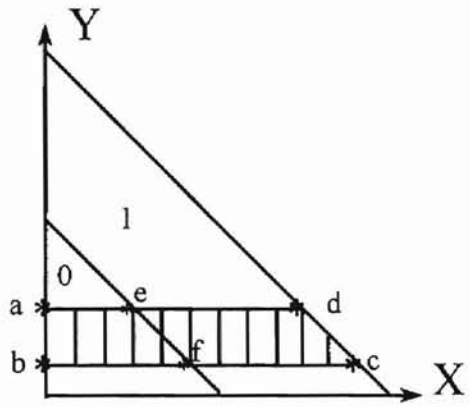


Figure 8 Example of Simple Range Query with Four Corner Points in Each Time Level

Input: A pointer to the root of a C-tree, the corner points of the whole simple query region R.

Output: The data points in the simple query region R.

1. Given the corner points of the whole simple query region R, invoke procedure *Find_time_level* for each corner point. Sort the returned time levels to find the minimum, and maximum time levels, Lmin and Lmax, respectively. Do the following steps for each selected time level L between and including Lmin and Lmax.

(a) If the first arbitrary decomposition line would intersect R decomposing it into two regions R1 and R2, then do (b) for each of R1 and R2, then concatenate the resulting address expressions with a " + " symbol, e.g. " 10* + 01* ."

(b) Invoke procedure *Region_add_exp* to return the address expression AE[L] of the simple query region in each time level L.

2. For each selected time level L do the following :

(a) Let S point to the root.

(b) Search node S, starting from the leftmost entry and toward the right, to find all entries whose address expressions are prefixes in AE[L], or AE[L] is a prefix in each of them.

(c) If the searched node(s) are nonleaf node(s), then descend the tree to the children of the selected entries. Repeat (b) for each of the child nodes.

(d) Otherwise, retrieve the points in the buckets of the searched nodes.

3. Return all the retrieved data points.

Procedure: Region_add_exp

Input: A time level L, the corner points of the simple query region R in time level L, the length G of the longest address expression so far in time level L.

Output: AE[L] which is the address expression of the simple query region in time level L.

1. Let AE[L] be an empty string.

2. Select the region W, that is the whole space of time level L.

3. If the length of AE[L] is equal to (G-1), then goto step 7.

4. Otherwise, divide the region W, using the type of line associated with the C-tree type.

5. If all the corner points of region R lie below or left of the division line, then let region

W be the whole region below or left of the division line. Append " 0 " to AE[L], then goto step 3.

6. Else if the corner points of region R lie above the division line, then let region W be the whole region above or right of the division line. Append " 1 " to AE[L], then goto step 3.

7. Otherwise (i.e. the division line crosses region R), append " * " to AE[L].

8. Return AE[L].

INSERTION

Inserting a new point in the database includes inserting three values for each point. These values are the x value, the y value, and the j value. This is performed in parallel using the three C-tree types "A", "B", and "C." The algorithm for insertion in each C-tree type is as follows :

Algorithm: Insert

Input: A pointer to the root of a C-tree, the new data point values (x, y, j), pointer to the array of dynamic time levels.

Output: A pointer to the array of dynamic time levels.

1. Invoke procedure *find_time_level*, which returns the time level L of the new data point.

2. If no node in the dynamic array has time level=L, then create one.

3. Invoke procedure *Point_add_exp*, which returns the address expression of the region that can include the new data point.

4. If there is no C-tree in time level L, then create one. Let S point to the root of the C-tree in time level L.

5. Search the node pointed to by S starting from the leftmost entry and moving toward the right. Select the first entry whose address expression is a prefix in the address expression of the new point.

6. If pointer S points to a nonleaf node, then let S point to the child of the selected entry. Goto step 5.

7. Otherwise, place the new point in the bucket of the selected entry.

8. If the bucket overflows, then invoke procedure *Split_bucket*.

9. Return a pointer to the array of dynamic time levels.

Procedure: Find_time_level

Input: The level granularity GRAN, the new data point values (x, y, j).

Output: The time level L of the new data point.

1. Calculate the time level L of the new point from the formula: L = (x / GRAN)

2. Return L.

Procedure: Point_add_exp

Input: The time level L of the new data point, the level granularity GRAN, the length G of the longest address expression so far in L.

Output: The address expression AD of a region in time level L that can include the new data point.

1. Let AD be an empty string.

2. If G=1, this means that the space in time level L has only one region, which is the whole space L. Thus, AD is " * ." Return AD.

3. Otherwise, calculate NOW_L of time level L from the formula: NOW_L = L x GRAN.

4. Invoke procedure *Space*, which returns the corner points of the space of time level L.

5. Let region R be the space of time level L.

6. If the C-tree is of type "A", divide region R with a line parallel to the X-axis. Else if the C-tree is of type "B", then divide the region R with a line parallel to the front line. Else if it is of type "C", then divide the region R using a line parallel to the Y-axis.

7. If the new point lies below or left of the division line, append " 0 " to AD. Else if the point lies above or to the right of the division line, append " 1 " to AD.

8. If the length of AD is less than G-1, then let region R be the one in which the new data point lies. Goto step 6.

9. Otherwise, append " * " to AD. Return AD.

Procedure: Space

Input: A time level L, the time level granularity GRAN.

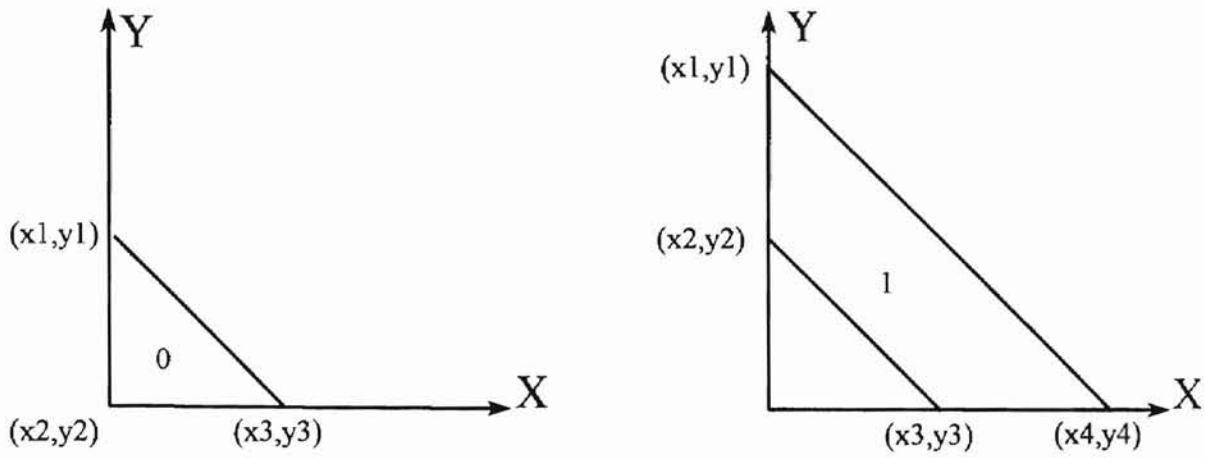
Output: The corner points of the whole space of time level L.

1. Define NOW_L for the current time level L by: NOW_L = L x GRAN.

2. Let the corner points of the space of L be (x1, y1), (x2, y2), (x3, y3), (x4, y4), respectively, starting from the top left corner of the space, and moving anti-clockwise.

3. As shown in figure 9a, time level L=0 has three corner points as follows: x1=0, y1=NOW_L, x2=0, y2=0, x3=NOW_L, y3=0, x4, and y4 do not exist. Figure 9b shows a time level L<>0 with four corner points as follows: x1=0, y1=NOW_L, x2=0, y2=NOW_L-GRAN, x3=NOW_L-GRAN, y3=0, x4=NOW_L, y4=0.

4. Return x1, y1, x2, y2, x3, y3, and x4, y4 (if exist).



(a) The Three Corner Points of the Space of Level " 0 ."

(b) The Four Corner Points of the Space of Level " 1 ."

Figure 9 Illustration of Time Levels Corner Points.

Procedure: Split_bucket

This algorithm performs a bucket split when the number of data points exceeds the threshold value.

Input: A pointer to the root of a C-tree, a time level L, the corner points (x1, y1), (x2, y2), (x3, y3), and (x4, y4) of the space of time level L, a pointer P to the node where the bucket is overflowing, the entry E whose bucket is overflowing.

Output: A pointer to the root of the C-tree.

1. Assume that the only points in the space of L are the points in the overflowing bucket.

2. Let the region R be the whole space of time level L. Let its address expression in L be AD. Let AD be an empty string.

3. If the C-tree is of type "A", then divide region R using a line parallel to the X-axis. If the C-tree is of type "B", then divide the space using a line parallel to the front line. If it is of

type "C", then use a line parallel to the Y-axis.

4. If all the points in region R do not lie on one straight line parallel to the decomposition line, then goto step 6. Otherwise, let AD have the address expression of the overflowing entry with the symbol "D" appended to it. For example, if the original address expression of the overflowing entry is " 0* ", then AD will be "0*D ."

5. Find the point in region R, that has the greatest value (y value for type "A", x value for type "B", j value for type "C"). Also, find the point that has the smallest value. Calculate the integer F form the formula: F=(The greatest value + The smallest value) / 2

Store F in the range field of the entry. Goto step 11.

6. If the number of points below or left of the division line is greater than the number of points above or right of the division line, then let R be the region below or left of the division line, and append " 0 " to AD.

7. Else if the number of points below or left of the division line is less than the number of points above or right of the division line, then let R be the region above or right of the division line, and append " 1 " to AD.

8. Else if the number of points is equal, then let R be anyone of the two regions on the sides of the decomposition line, and append to AD the corresponding symbol (1 or 0).

9. If the overflow still exists, then let region R be the region that is still overflowing. Goto step 3.

10. Else if there is no more overflow in any region, then append " * " to AD.

11. If there are entries on the right of E, then shift them one entry to the right, creating an empty entry on the right of E.

12. Let the empty entry on the right of E have the address expression of E. Let AD replace

the address expression of E.

13. Redistribute the points of the split bucket, such that the points of region R remain in the split bucket, while the rest of the points are transferred to the new bucket.

14. If node P overflows, invoke procedure *Split_node*.

15. Return a pointer to the root of the C-tree.

Procedure: Split_node

Input: A pointer to the root of a C-tree, a pointer to the overflowing node.

Output: A pointer to the root of the C-tree.

1. Let the splitting node be node1. Create a new empty node labeled node2. Let the numbering of entries of node1, starting from the leftmost entry and moving toward the right, be 0, 1,..., M+1.

2. Search node1 starting from entry number M and moving toward the left, to find the first entry that is *not* a prefix in each of the address expressions of the entries on its left. If not found then select entry number M+1. Transfer the selected entry, and all the entries on its right (if exist) to node2.

3. If node1 is a nonleaf node, update its children nodes with their new parent entries.

4. Invoke procedure *Adjust_tree*. Return a pointer to the root of the C-tree.

Procedure: Adjust_tree

Input: A pointer to a C-tree, a pointer to the split node labeled node1, a pointer to the new node labeled node2.

Output: A pointer to the root of the C-tree.

1. If node1 is the root, then create a new root, whose children are node1, and node2. Let the parent entry of node1 have the same address expression of the rightmost entry of node1, and let the parent entry of node2 have the same address expression of the rightmost entry of node2. Return a pointer to the root of the C-tree.

2. Else if node1 is not the root, then let pointer S point to the parent of node1. Let the parent entry of node1 be E1. Let the parent entry of node2 be E2. E1 is already in node S, but E2 is not. E2 must have the same address expression of the rightmost entry in node2.

3. Shift all the entries on the right of E1 one entry to the right. Install E2 on the right of E1.

4. If node S has overflown, then invoke procedure *Split_node*. Then invoke *Adjust_tree*.

5. Otherwise, return pointer to the root of the C-tree.

DELETION

The importance of deletion in temporal databases is minor, because of the append-only nature of temporal databases. However, deletion may still be needed for corrections.

Algorithm: Delete

Input: A pointer to the root of a C-tree., the values (x, y, j) of the data point to delete.

Output: A pointer to the root of the C-tree.

1. Invoke procedure *Point_search*.

2. If Point_search returns NULL, then return NULL.

3. Else if Point_search returns an entry number E, and a pointer P to the node of E, then remove the point from the bucket of E.

4. If the remaining number of points in the bucket of E is greater than zero, then return a

pointer to the root of the tree

5. Else if the remaining number of points in the bucket of E is zero, then consider the following cases:

(a) The entry E is not the only occupied entry in the node, but it is the rightmost entry, therefore let the entry on its left have the address expression of E. Remove E from the node.

(b) The entry E is not the only occupied entry in the node and is not the rightmost entry nor the leftmost, therefore shift all the entries on the left of E one entry to the left, such that E is replaced by the one on its left.

(c) The entry E is the only occupied entry in the node and it is the leftmost entry, then if the parent entry Ep of the node of entry E is not the leftmost entry, then remove the node of E, remove the parent entry Ep and shift the entries on the left of Ep one entry to the left. Else if Ep is the leftmost entry, then leave empty node as it is to preserve the balance of the tree.

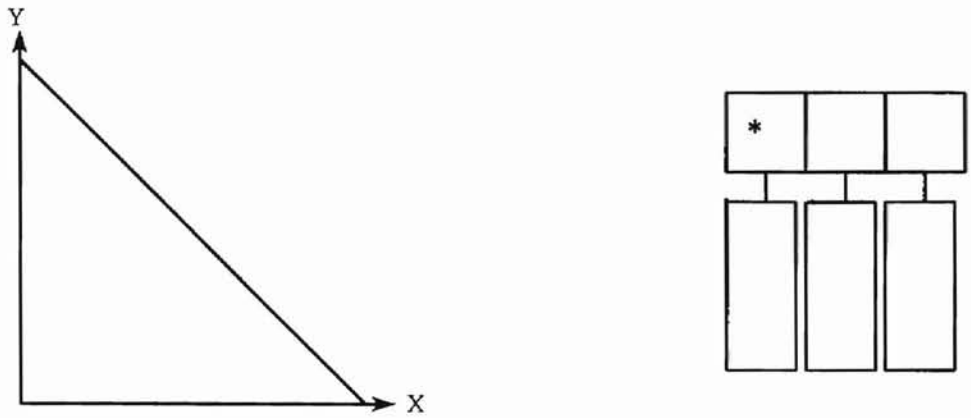
6. Return a pointer to the root of the C-tree.

ILLUSTRATIVE EXAMPLE ON MAIN OPERATIONS

The following is a step by step example to elaborate on the space decomposition technique utilized within insertion and search operations.

Step 1: figure 10 shows the temporal space and the corresponding empty C-tree before any insertions. The C-tree is arbitrarily assumed to be of type "A." For the sake of illustration, we assumed that the maximum bucket capacity is equal to three, and the maximum node capacity is also three. As shown in figure 10b, the C-tree has only one occupied entry, which is the leftmost entry in the root. The address expression stored in this entry is " * ", which corresponds to the whole space in figure 10a. The tree buckets have no data points, since the

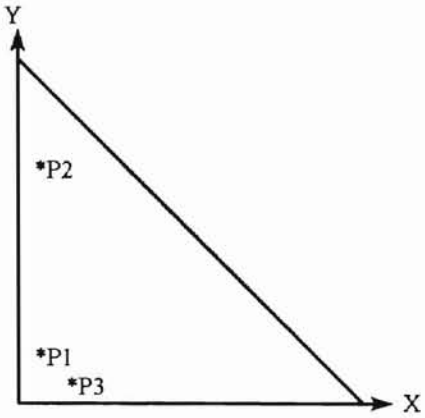
tree is empty. Thus, an empty C-tree is defined as a C-tree that has one occupied entry with the address expression " * " in the root node, and zero data points.



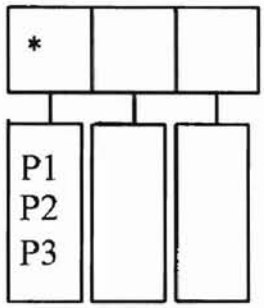
(a) Temporal Space before Insertion. **(b)** Empty C-tree.

Figure 10 The Temporal Space and the Corresponding C-tree before any Insertions

Step 2: figure 11 shows the location of a new point P1 in the space. To insert a new point into the C-tree, an address expression is assigned to the new point as explained earlier in this section in procedure Point_add_exp. According to this procedure, the length of the assigned address expression to a new point is equal to the length of the longest address expression stored in the root at the time of inserting the new point. In this example, the C-tree in figure 10b, has only one address expression, containing only one symbol. Thus, the length of the longest address expression , at the time of inserting P1, is equal to one. Consequently, the address expression of point P1 is " * ", which represents the whole space and is the only possible address expression with length equal to one. After P1 is assigned an address expression, it is ready to be inserted into the C-tree. In general, a search operation for an insertion starts at the root of the C-tree, and ends by locating the appropriate leaf node for inserting the new point. In each node, the search starts at the leftmost entry. If the address



(a) The Space after Insertion of P1, P2, and P3.

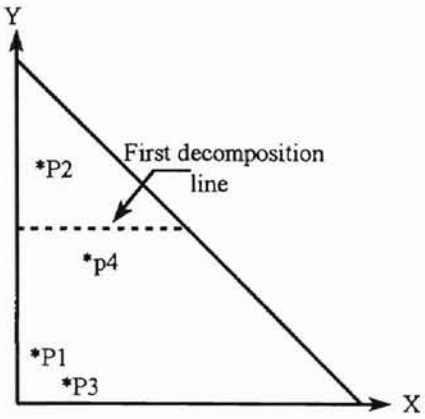


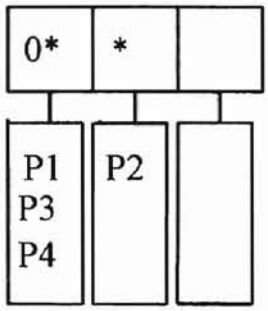
(b) The C-tree after Insertion of P1, P2, and P3.

Figure 11 Insertion of the New Points P1, P2, P3.

expression of the searched entry is " * " , or is a prefix in the address expression of the new point, then the search moves to the child node in case of nonleaf node, and to the data bucket in case of leaf node. Otherwise, the search moves to the next entry on the right. In this example, the address expression of P1 is compared with the leftmost entry in the root, and thus P1 is inserted in the data bucket of this entry, as shown in figure 11b. Similarly, points P2, and P3 are inserted in the same data bucket.

Step 3: Since the C-tree in this example has a maximum bucket capacity equal to three, then the insertion of point P4 in the same bucket causes the bucket to overflow. To overcome this problem, a space decomposition is performed. Since the C-tree is of type "A", then the decomposition lines are of type "one", i.e. parallel to the X-axis. As shown in figure 12a, the first decomposition line divides the entire space of the C-tree along the Y-axis into two regions. The address expression of the region below the decomposition line is " 0* ", while that of the region above the decomposition line is " 1* ." No further decomposition is needed, since none of the resulting regions has overflow. Thus, the original entry is split into





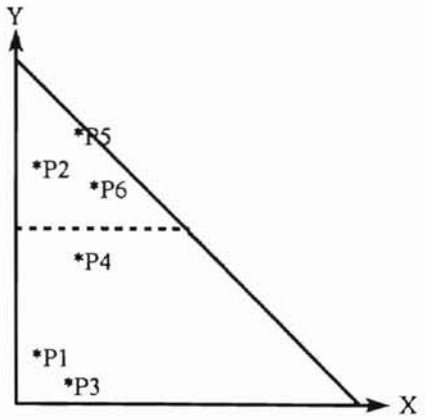
(a) The Space after Insertion of P4.

(b) The C-tree after Insertion of P4.

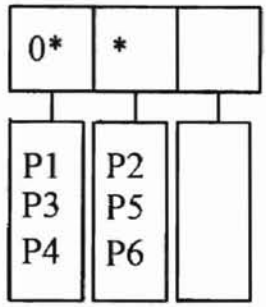
Figure 12 Insertion of Point P4.

two new adjacent entries. In general, the left entry always represents the region with greater number of points. Special cases in choosing the region to be stored in the left entry, are explained in procedure Split_bucket . The right entry is always assigned the same address expression as the original split entry. Thus, " * " is assigned to the right entry instead of " 1* ." All the points in the bucket of the original entry, and not in the bucket of the left entry, are placed in the bucket of the right entry. Figure 12b shows the C-tree after the leftmost entry is split.

Step 4: Similarly, to insert P5 and P6, they have to be assigned address expressions. By following procedure Point_add_exp, we find that the address expression of P5 is " 11* ", and the address expression of P6 is " 10* ." By searching the tree in figure 12b, we find that the first entry that can include P5 or P6 is the root's second entry, whose address expression is " * ." Figure 13 shows that points P5 and P6 have been inserted without further space decomposition.



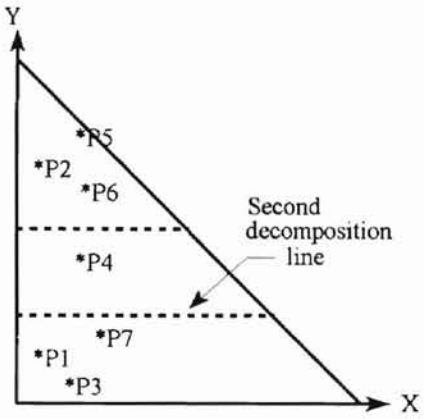
(a) The Space after Insertion of P5, P6.



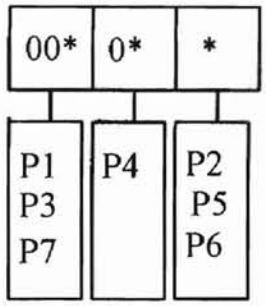
(b) The C-tree after Insertion of P5, P6.

Figure 13 Insertion of Points P5, P6.

Step 5: Figure 14a shows the location of the new point P7 in the space. The address



(a) The Space after Insertion of P7.



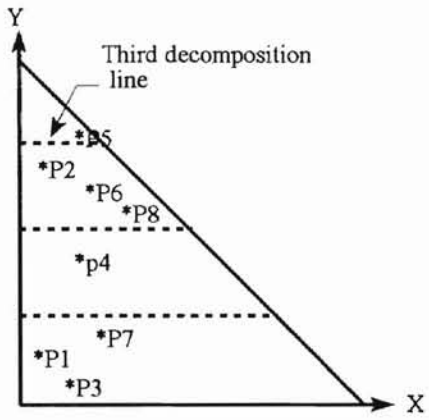
(b) The C-tree after Insertion of P7.

Figure 14 Insertion of Point P7.

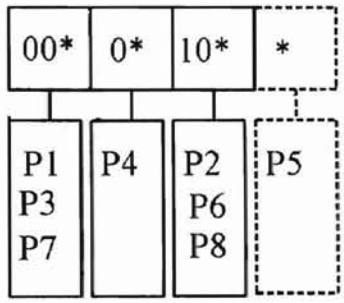
expression of P7 is " 00* ." Thus, P7 can be inserted in the bucket of the leftmost entry in figure 13b. Since the assumed maximum bucket capacity of the tree is three, therefore the selected bucket overflows. Thus, a second decomposition is performed as shown in figure

14a. Note that the second decomposition line divides the overflowing region, located below the first decomposition line. Thus, the entry of the overflowing bucket is split, as shown in figure 14b. The split entry is replaced by two adjacent entries. The left entry of these two entries is assigned the region that contains a bigger number of points. Thus, the address expression of the left entry is " 0* ." The right entry is always assigned the same address expression as the original split entry. Thus, its address expression is " 0* ."

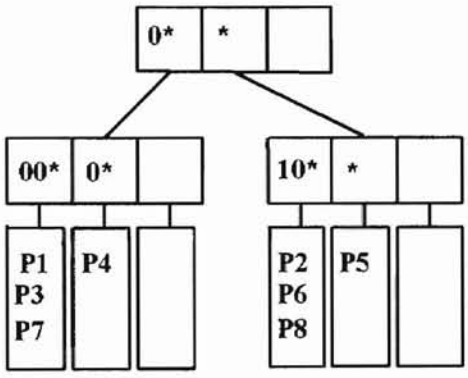
Step 6: Figure 15a shows the location of P8 in the space. The address expression of P8 is



(a) The Space after Insertion of P8.



(b) The C-tree after Insertion of P8.



(c) The C-tree after Node Split.

Figure 15 Insertion of Point P8.

" 10* ." Thus, it can be inserted in the bucket of the rightmost entry in figure 14b. Since this causes the bucket to overflow, another space decomposition is performed, as shown in figure 15a. Consequently, the bucket of the rightmost entry is split, as shown in figure 15b.

Since the C-tree in this example has a maximum node capacity equal to three, then a node split is performed on the root of figure 15b. As presented in procedure Split_node, the overflowing node is split into two nodes. To select which entries are installed in each node, a search is performed in the overflowing node of figure 15b. The direction of the search is from right to left. The objective of the search is to find the first entry E, whose address expression is not a prefix in the address expressions of at least one entry on its left. In the overflowing node of figure 15b, E is the entry whose address expression is " 1* ." Thus, entry E, and all the entries on its left, are transferred to a new node, while all the entries on the right of E remain in the original node, as shown in figure 15c. Since the split node is the root, a new root is created. The address expression of a parent entry is always the same as the address expression of the rightmost entry of the child node as shown in figure 15c.

CHAPTER IV

PERFORMANCE ANALYSIS & CONCLUSION

In this chapter, we evaluate the performance of each C-tree type and the combination of the three types. The evaluation focuses on the storage cost and query processing performance. Our simulation study was implemented on Sequent Symmetry S-81 multiprocessor, using the C programming language. Our study consisted of three experiments. The first experiment employed our local system parameters. The goal of this experiment was to make a comparison between the three C-tree types. The second experiment aimed at comparing the empirical results from our study on the parallel C-trees with the study of Lu, Ooi, and Shen (1993) on the time index and the TP-index. Accordingly, the second experiment of our study used the same system parameters of the study of Lu, Ooi, and Shen. The third experiment focused on studying the influence of the level granularity as a tuning factor. The selected system parameters in the third phase were the same as the second experiment.

The system parameters of our local system, and which were used in the first experiment of the simulation study were as follows: page size = 1k, character = 1 byte, integer = 4 bytes, pointer = 4 bytes, float = 4 bytes. The maximum number of entries M in a C-tree node can be calculated from the formula:

*M * (address expression + pointer to child + range field number) + pointer to parent = page size*

The address expression is assumed to consist of at most 25 characters, and the range field number is assumed to be an integer or a float. Thus, from the above formula, M = 30. The

parameters used in the study of Shen, Ooi, and Lu, and in the second and third experiments of our simulation study were as follows: page size = 4k, character = 1 byte, integer = 8 bytes, pointer = 32 bytes, float = 8 bytes. Thus, from the above formula M = 62, which is also the same value used for maximum node capacity in the simulation study of Lu, Ooi, and Shen.

STORAGE EFFICIENCY

In studying the storage cost of the parallel C-trees, our simulation study used the same test data parameters used in the simulation study of Lu, Ooi, and Shen. Thus, we used a constant number of 100,000 data points, while varying the mean interarrival time (mean time between two subsequent arrivals), and the mean duration. The space cost was measured in number of pages (tree nodes).

The study of Lu, Ooi, and Shen shows that in general the space cost of the TP-index is very low and is independent on the mean interarrival time, or the mean duration. As for the time index, the space cost changes with the change of the mean interarrival time, and the mean duration. The space cost of the time index is low, only if both the mean interarrival time, and the mean duration are small. It increases substantially, if the mean duration is large compared to the mean interarrival time. Their explanation for this was as follows. If the mean interarrival time was small, multiple tuples arriving at the same time was more likely, and thus the number of indexing points was reduced leading to a smaller tree. If the interval duration was large compared to the interarrival time, the same tuple was duplicated in many leading buckets, and thus creating a larger storage space. The space cost of the time index was five to eight times higher than the TP-index.

The empirical results produced from the first experiment of our simulation study on the

parallel C-trees are shown in figures 16, 17, 18. It can be observed from the charts that the space cost of C-trees of type "C" is higher than the other two types "A" and "B ." This can be explained as follows. Temporal databases tend to have a high degree of overlapping between time intervals. Thus, data points tend to form close clusters along the X-axis. Recall that a C-tree of type "C" uses decomposition lines of type "three", i.e. parallel to the Y-axis. Thus, a C-tree of type "C" uses more decomposition lines to decompose the closely clustered points along the X-axis, and consequently acquires more space than the other two indexes in which it is less likely to have the same high degree of clustering.

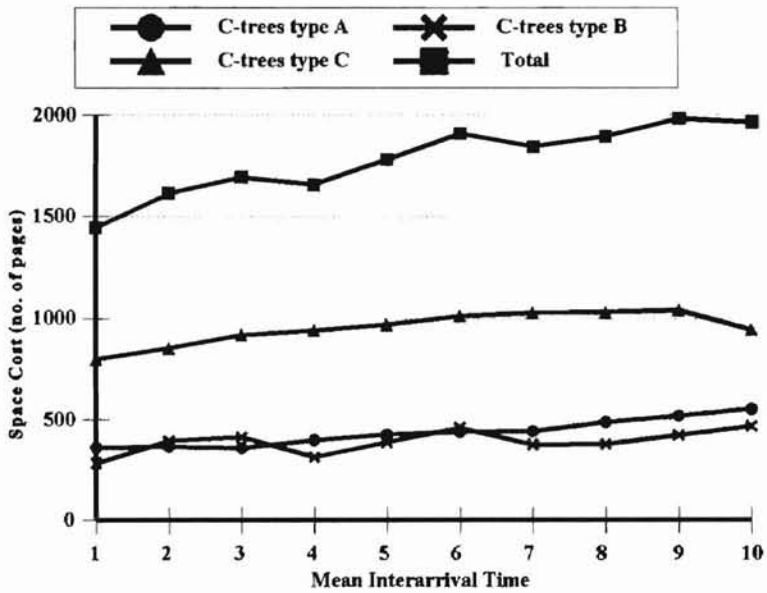


Figure 16 Space Cost of Each C-tree Type, where Maximum Node Capacity (M)=30, Level Granularity=365, and Mean Duration / Mean Interarrival time= 100.

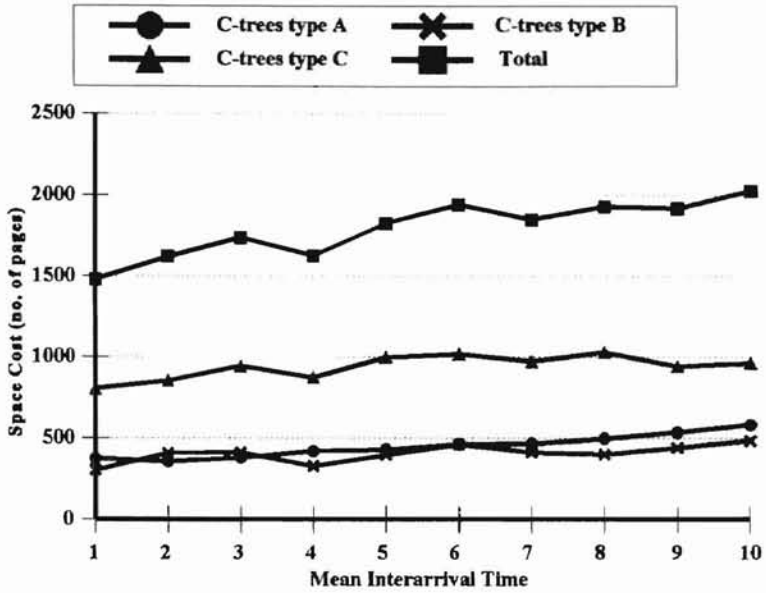


Figure 17 Space Cost of Each C-tree Type, where Maximum Node Capacity (M)=30, Level Granularity=365, and Mean Duration / Mean Interarrival time = 300.

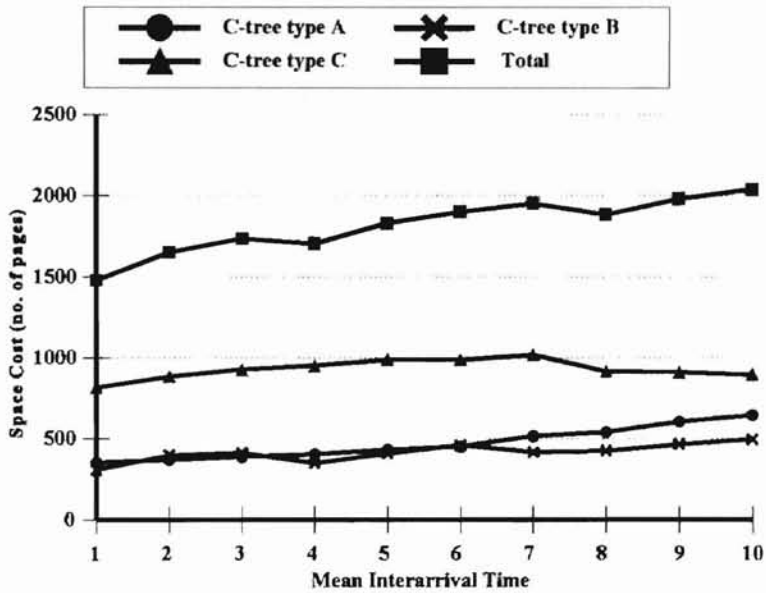


Figure 18 Space Cost of the Three C-tree types, where Maximum Node Capacity (M)=30, Level Granularity=365, and Mean Duration / Mean Interarrival Time = 500.

The charts also show that the individual space cost of each C-tree type slightly increases by the increase in mean interarrival time, or in mean duration. This is explained due to the increase in number of required time levels with the increase in mean interarrival time, or in mean duration. Since each time level is indexed by one C-tree of each type, then the increase in mean interarrival time, or in mean duration, implies increase in number of required C-trees. Figures 19, 20, 21 show the change in number of time levels as the mean interarrival time and the mean duration are varied. The charts show that the number of time levels increases with the increase in mean interarrival time, or in mean duration.

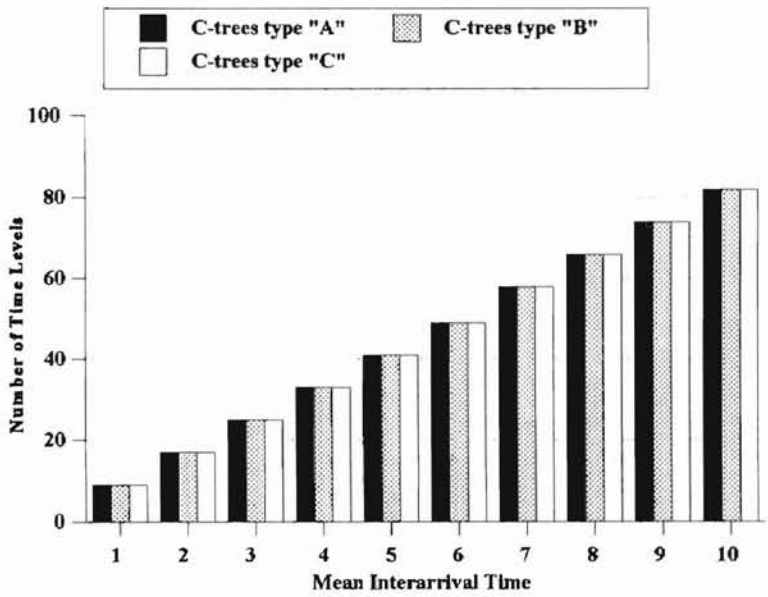


Figure 19 Number of Time Levels of Each C-tree Type, where Maximum Node Capacity (M)=30, Level Granularity=365, and Mean Duration / Mean Interarrival Time = 100.

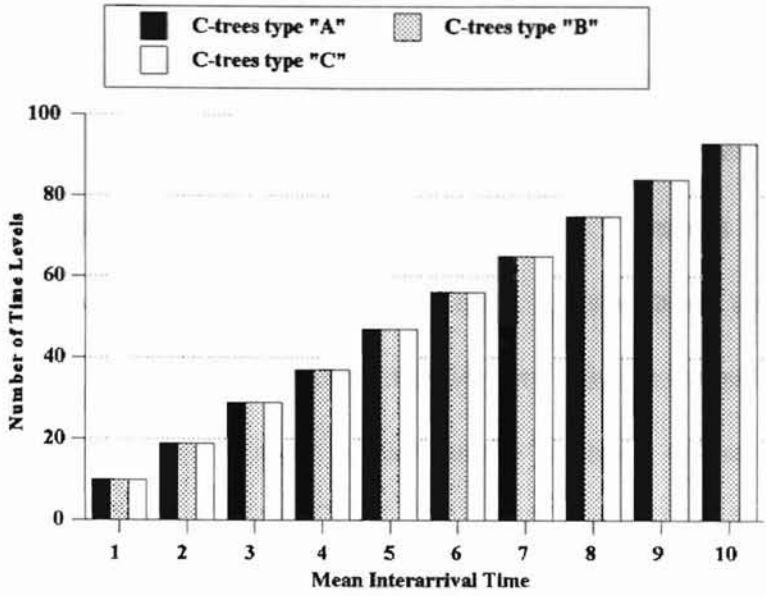


Figure 20 Number of Time Levels of Each C-tree Type, where Maximum Node Capacity (M)=30, Level Granularity=365, and Mean Duration / Mean Interarrival Time = 300.

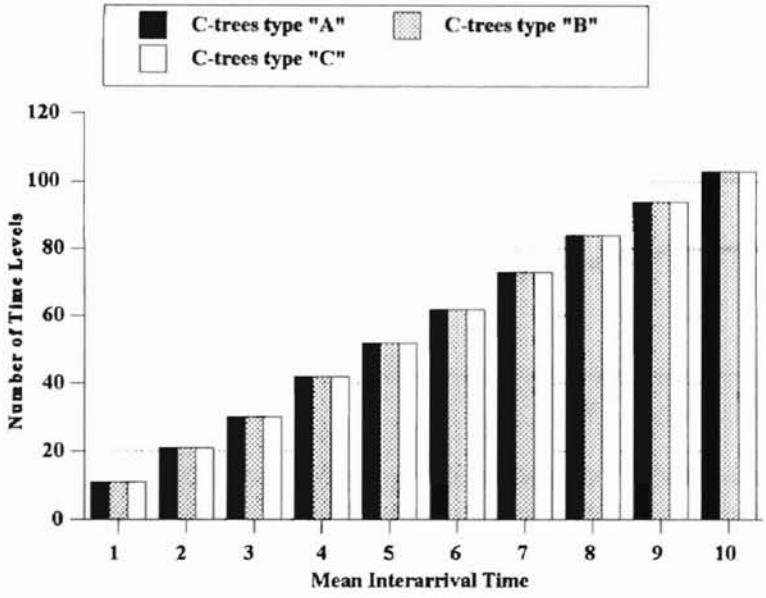


Figure 21 Number of Time Levels of Each C-tree Type where Maximum Node Capacity=30, Level Granularity=365, and Mean Duration / Mean Interarrival Time = 500.

In the second experiment of our simulation study, a comparison was conducted between the performance of each C-tree type based on our empirical results, the time index, and the TP-index studied by Lu, Ooi, and Shen. The results of the second experiment of our simulation study are displayed in figures 22, 23, and 24. Table 2 presents a summary of the comparison between the space costs of the parallel C-trees, the time index, and the TP-index. It can be seen from table 2 that the parallel C-trees are very efficient in space cost, under all test conditions. Its performance is much better than the TP-index, or the time index in all cases.

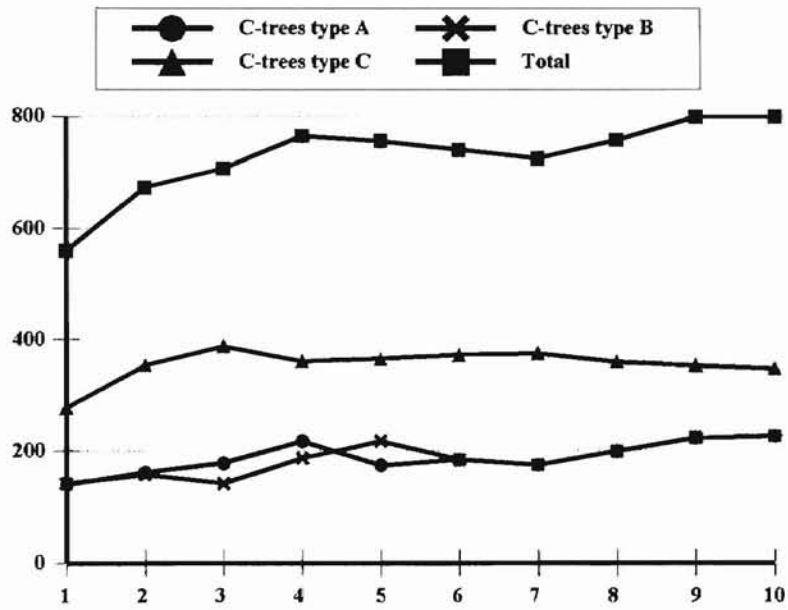


Figure 22 Space Cost of Each C-tree type, where Maximum Node Capacity (M)=62, Level Granularity=365, and Mean Duration / Mean Interarrival Time= 100.

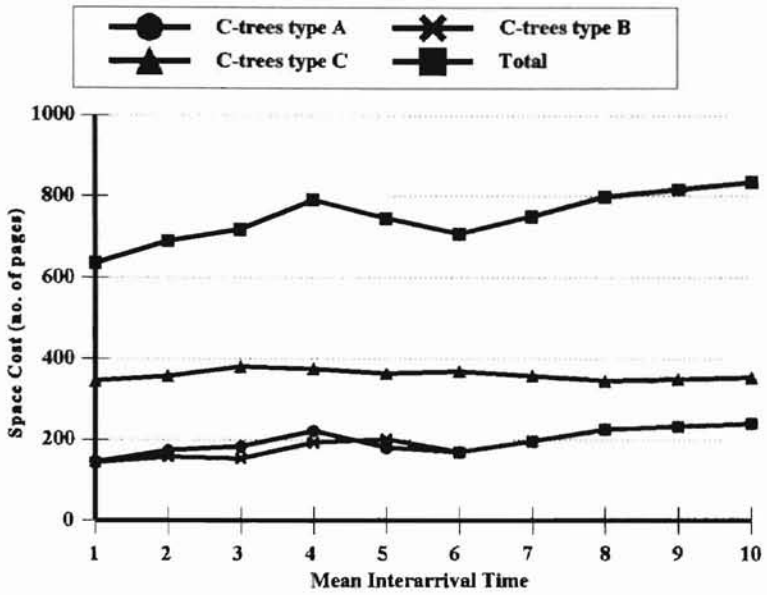


Figure 23 Space Cost of Each C-tree type, where Maximum Node Capacity (M)=62, Level Granularity=365, and Mean Duration / Mean Interarrival Time= 300.

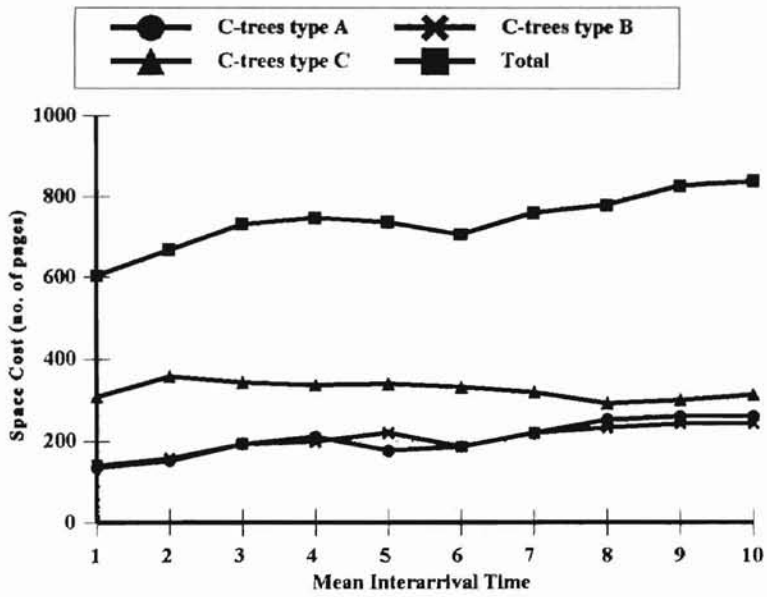


Figure 24 Space Cost of Each C-tree type, where Maximum Node Capacity (M)=62, Level Granularity=365, and Mean Duration / Mean Interarrival Time=500.

Test Condition	Space Cost Comparison Result
mean interarrival time < 4	-Parallel C-trees have best space cost. -TP-index is 2.9 times C-trees.. -Time Index is 17 times C-trees.
4 < mean interarrival time < 6	-Parallel C-trees have best space cost. -TP-index is 2.7 times C-trees. -Time Index is 22.7 times C-trees.
mean interarrival time > 6	-Parallel C-trees have best space cost. -TP-index is 2.5 times C-trees. -Time Index is 37.5 times C-trees.

Table 2 Summary of Space Cost Comparison between Parallel C-trees, TP-index, and Time Index.

The last experiment in our simulation study deals with tuning the space cost by changing the value of the level granularity. In order to know the impact of varying the level granularity on the space cost of the parallel C-trees, similar tests were conducted on the individual C-tree types. The ratio of mean duration to mean interarrival time was assumed constant and made arbitrarily equal to 100. Figures 25, 26, 27 show the effect of varying the level granularity on C-trees type "A", "B", and "C", respectively. The charts show that for different values of interarrival time, the space cost of each C-tree type deceases as the level granularity increases. This is due to the decrease in number of time levels as the level granularity increases.

An important advantage of the parallel C-trees is that not all of the three types of C-trees have to be loaded to main memory for every update operation. For instance, a query may be processed by one type of C-trees leaving the remaining types on the secondary storage

device. Accordingly, in worst case of loading the three types to main memory, the space cost would still be very efficient, as shown from table 2.

Our study shows that the parallel C-trees are efficient in storage cost for short and long duration temporal data. The space cost is more efficient for small interarrival times. Small interarrival times is the common case among temporal databases. Nevertheless, for larger interarrival times, the increase in the space cost is minor. Although a choice of a bigger value for level granularity means more decompositions, the space cost decreases due to the decrease in number of time levels or C-trees. An immediate conclusion is that the used decomposition technique is efficient and is not responsible for a substantial increase in the space cost compared with the number of time levels.

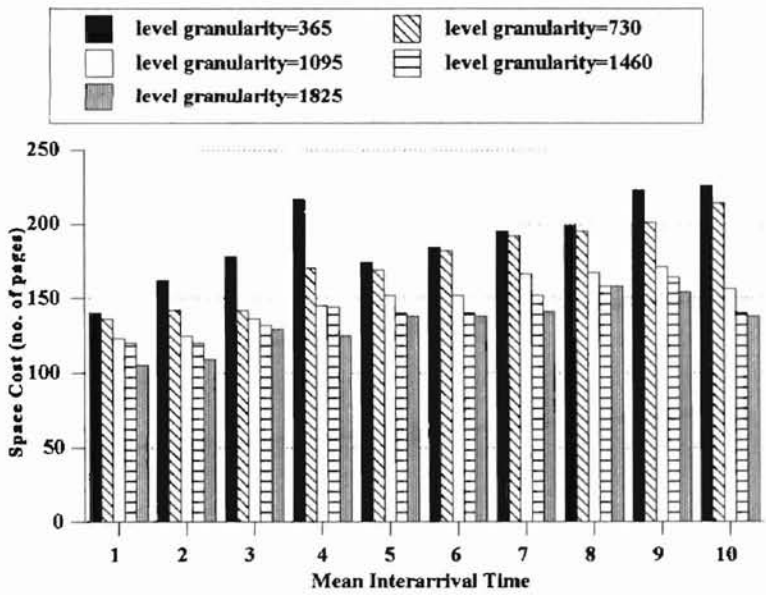


Figure 25 Effect of Changing the Level Granularity on the Space Cost of C-trees Type "A", where Maximum Node Capacity (M)=62, and Mean Duration / Mean Interarrival Time= 100.

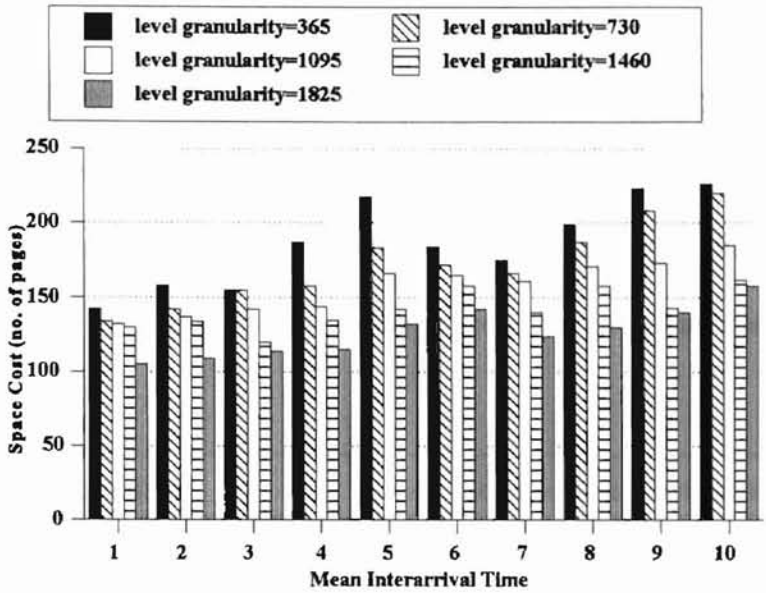


Figure 26 Effect of Changing the Level Granularity on the Space Cost of C-trees Type "B", where Maximum Node Capacity (M)=62, and Mean Duration / Mean Interarrival time= 100.

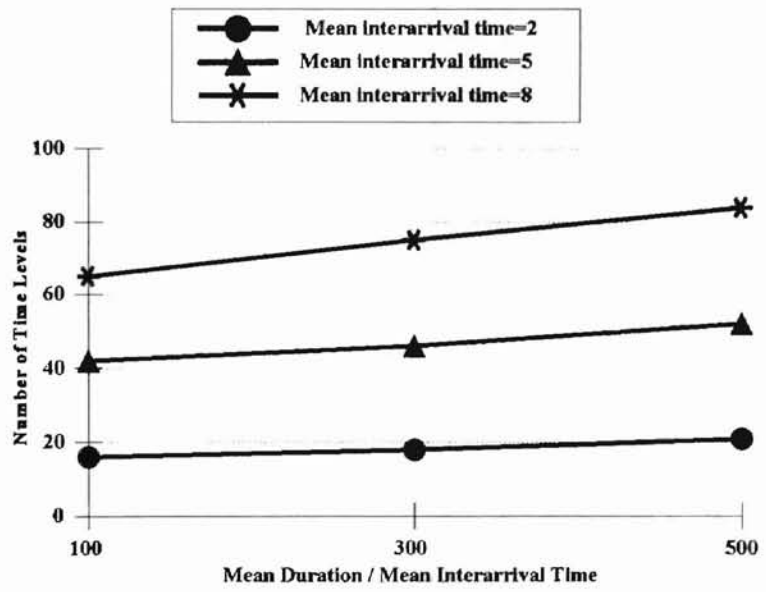


Figure 27 Number of Time Levels for any C-tree type, where Maximum Node Capacity (M)=62, Level Granularity=365, while Varying Mean Duration/Mean Interarrival Time.

QUERY PROCESSING EFFICIENCY

Lu, Ooi, and Shen (1993), conducted the analysis on the query: "Retrieve all the temporal tuples whose starting time is between a and b." The mean length of the interval (b-a) was assumed 1000. The mean duration was set to 600 times larger than the interarrival time. Figure 28 shows the region corresponding to the given query under analysis. Table 3

Test Condition	Search Time Comparison Result
mean arrival rate < =1	Time Index is 5.9 times TP-index
1 < mean arrival rate < 3	Time Index is 6.3 times TP-index
mean arrival rate >= 3	Time Index is 2.5 times TP-index

Table 3 Summary of Search Performance Comparison between TP-index and Time Index from the Study of Lu, Ooi, and Shen (1993).

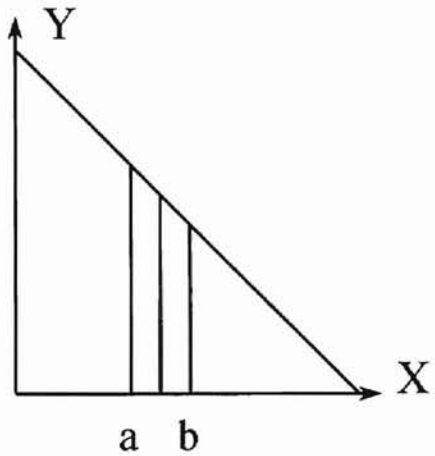


Figure 28 The Region of the First Query under Analysis

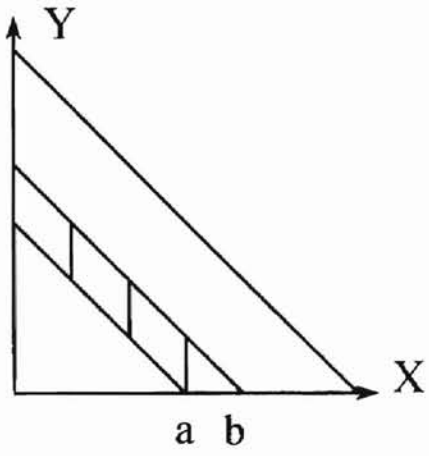


Figure 29 The Region of the Second Query under Analysis

summarizes some of the results presented in the study of Lu, Ooi, and Shen in comparing the search performances of the TP-index, and the time index. It can be seen from table 3 that

the search time of the TP-index is much less than the time index under all test conditions.

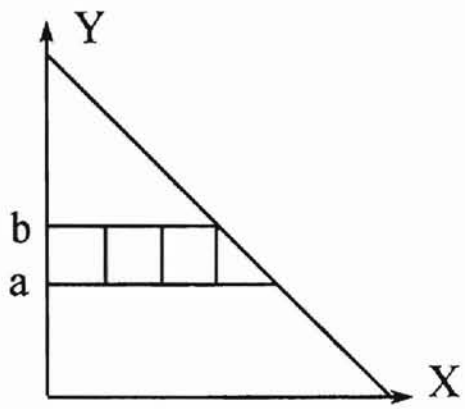


Figure 30 The Region of the Third Query under Analysis

In the first experiment of our simulation study, we used our local system parameters. Our analysis was conducted on the queries of figure 28, 29, and 30. Since the given query in figure 28 is a simple query that utilizes decomposition lines of type "three", parallel C-trees of type "C" were employed to answer the query. The simple queries shown in figures 29 and 30 were analyzed using parallel C-trees of types "B", and "A", respectively.

The empirical results produced from the first experiment, showing the effect of varying the mean arrival rate (1 / mean interarrival time) on the query processing performance of the three C-tree types "A", "B", and "C" are shown in figures 31, 32, and 33. Since the dominant factor in measuring the response time in our scheme is the time required to search the C-trees, other overhead times for decomposing a query or merging the results of the parallel C-trees are ignored. Thus, the response time or search time is measured by number of accessed leaf and nonleaf nodes. It can be observed from the charts that for the three types of C-trees, the search time slightly increases, as the mean arrival rate increases. This can be explained as follows. As the mean arrival rate increases, or in other words as the mean interarrival time decreases, the number of time levels decreases. Thus, the potential of parallelization decreases, and more search time is needed. This result is consistent with the fact that the more parallelization is established, the less search time is required.

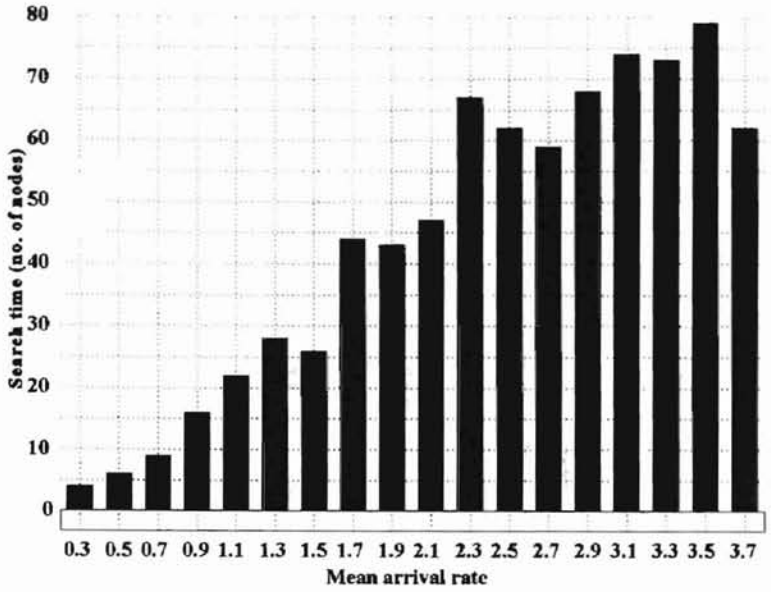


Figure 31 Search Performance of C-trees type "A" for a Simple Range Query, where Maximum Node Capacity (M)=30, and Level Granularity=365.

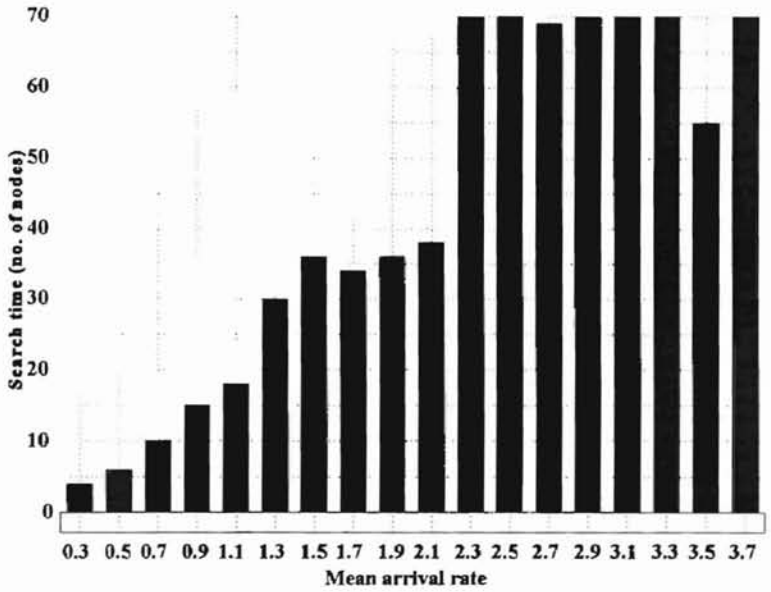


Figure 32 Search Performance of C-trees Type "B" for a Simple Range Query, where Maximum Node Capacity=30, and Level Granularity=365.

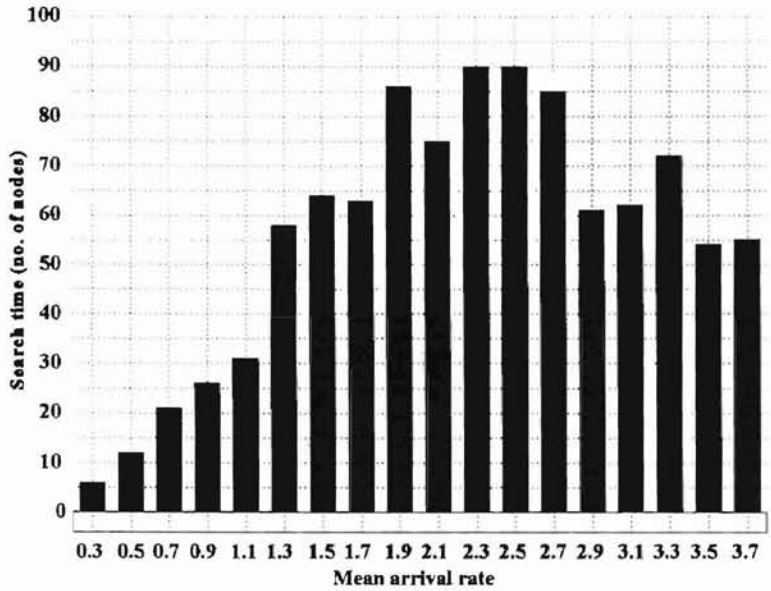


Figure 33 Search Performance of C-trees type "C" for a Simple Range Query, where Maximum Node Capacity (M)=30, and Level Granularity=365.

Figures 34, 35, and 36 display the results of the second experiment of our simulation study, in which we used the same system parameters of the study of Lu, Ooi, and Shen. The

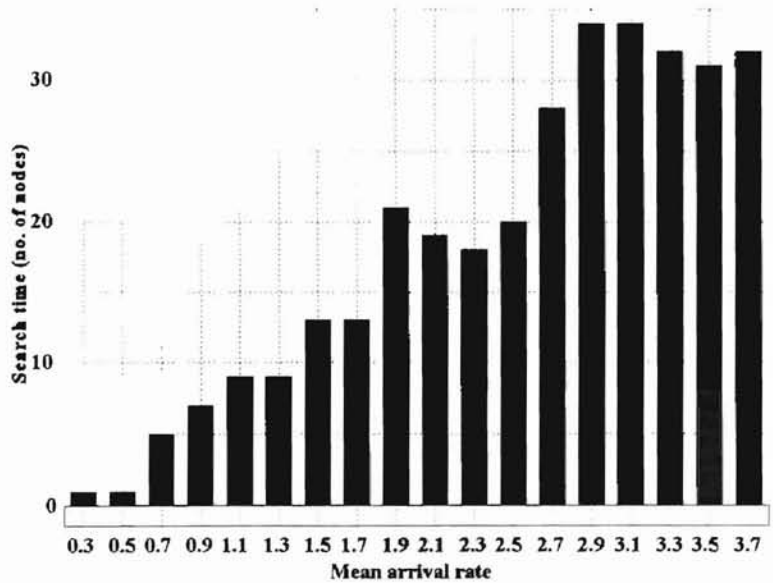


Figure 34 Search Performance of C-trees Type "A" for a Simple Range Query, where Maximum Node Capacity (M)=62, and Level Granularity=365.

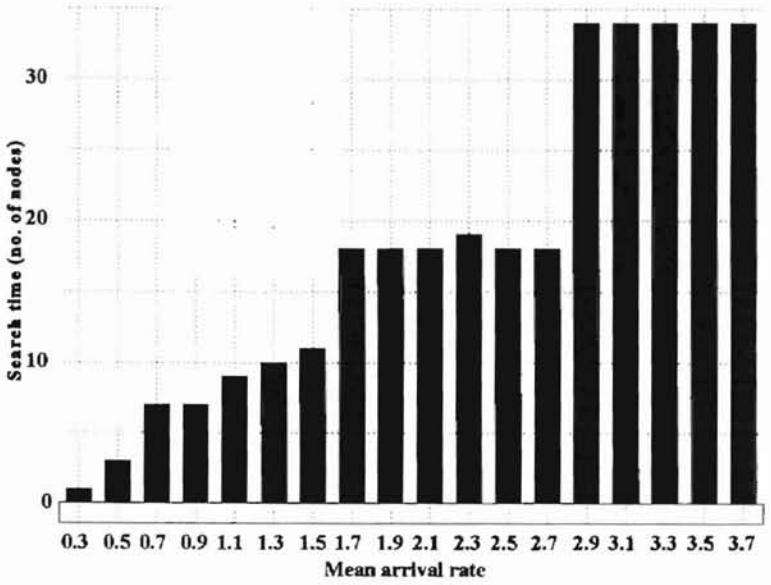


Figure 35 Search Performance of C-trees Type "B" for a Simple Range Query, where Maximum Node Capacity (M)=62, and Level Granularity=365.

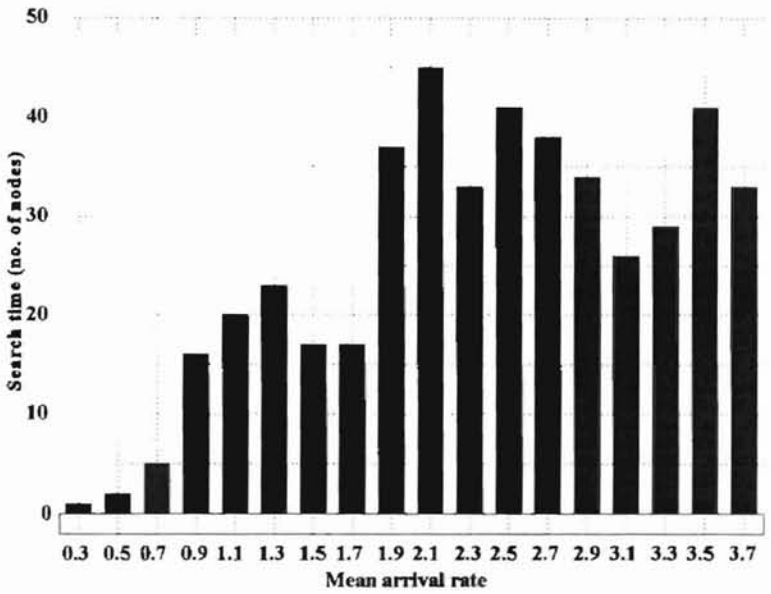


Figure 36 Search performance of C-trees type "C" for a simple range query, where maximum node capacity (M)=62, and level granularity=365.

performance results of each C-tree type was compared individually with the results of the

TP-index, and the time index as presented in the study of Lu, Ooi, and Shen. Table 4 summarizes the results of the comparison. It can be seen that in all cases each C-tree type individually showed better search performance than the TP-index, or the time index.

Test Condition	Search Time Comparison Result
mean arrival rate < =1	-C-trees type "A" gave best search time -C-trees type "B" are 1.3 times type "A" -C-trees type "C" are 1.7 times type "A" -TP-index is 2.9 times C-trees type "A" -Time Index is 17 times C-trees type "A"
1 < mean arrival rate < 3	-C-trees type "B" gave best search time -C-trees type "A" is 1.1 times type "B" -C-trees type "C" is 1.81 times type "B" -TP-index is 8 times C-trees type "B" -Time Index is 50 times C-trees type "B"
mean arrival rate >= 3	-C-trees type "C" gave best search time -C-trees type "A" are 1.1 times type "C" -C-trees type "B" are 1.3 times type "C" -TP-index is 10 times C-trees type "C" -Time Index is 25 times C-trees type "C"

Table 4 Summary of Search Performance Comparison between Parallel C-trees, TP-index, and Time Index.

In the third experiment, tests were conducted on each C-tree type to study the effect of varying the level granularity on the search time. Figures 37, 38, and 39 show that for each

C-tree type, less level granularities rendered less search times. This result can be explained as follows. From the definition of the level granularity, it is obvious that as the level granularity decreases, the number of time levels increase. Since in our tests, adequate number of processors was always available, the increase in number of time levels, implied more parallelization. Thus, less search time is needed.

Although we have shown that each individual C-tree type is efficient in query processing, the real power of the parallel C-trees is in handling complex range queries. As explained earlier, a complex range query is decomposed into simple range queries. Accordingly, the complex query processing efficiency is a direct reflection of the simple query processing efficiency, which has been proven by our empirical results in this chapter.

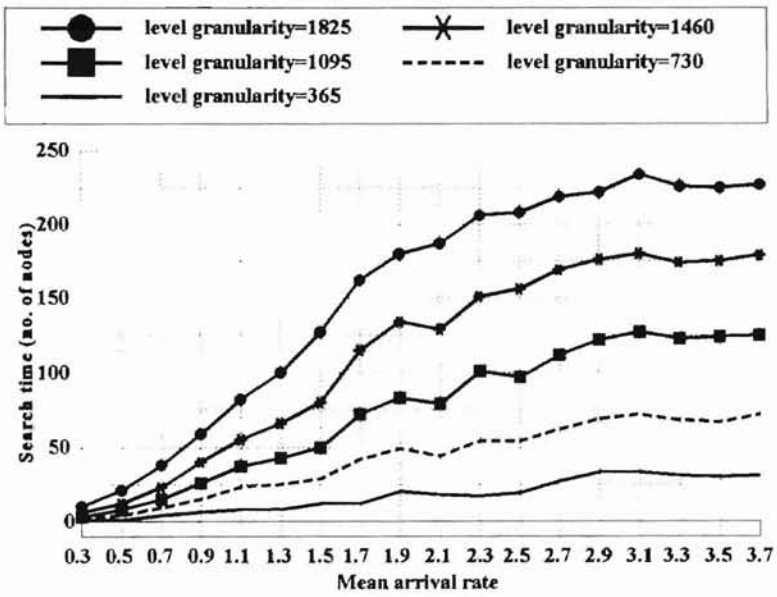


Figure 37 Effect of Changing the Level Granularity on the Search Performance of C-trees Type "A", where Maximum Node Capacity (M)=62.

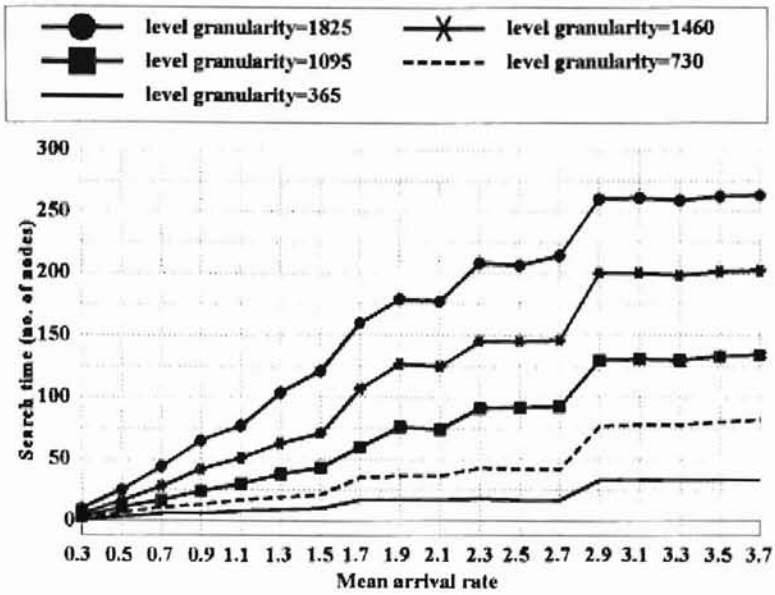


Figure 38 Effect of Changing the Level Granularity on the Search Performance of C-trees Type "B", where Maximum Node Capacity (M)=62.

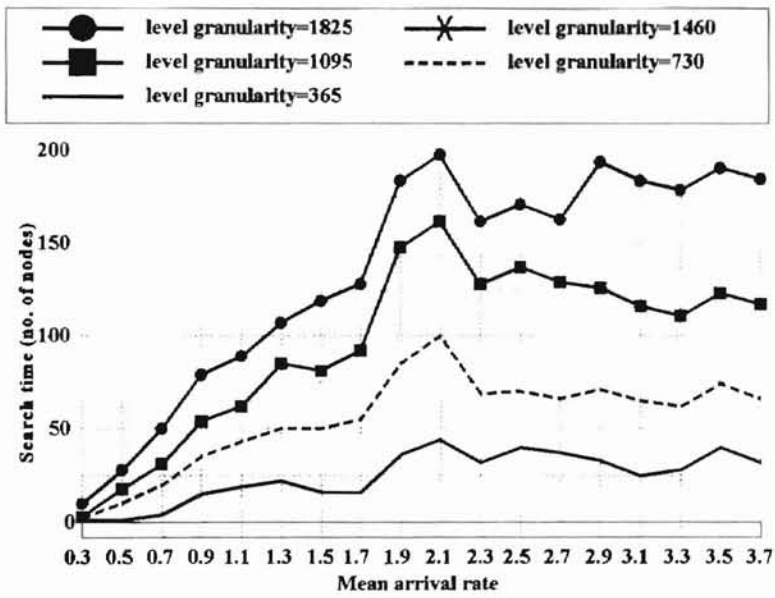


Figure 39 Effect of Changing the Level Granularity on the Search Performance of C-trees Type "C", where Maximum Node Capacity (M)=62.

SUMMARY, CONCLUSIONS, AND SUGGESTED FUTURE WORK

This thesis introduced the C-tree, which is a dynamic index for valid time temporal databases. Indexing in the C-tree is mainly based on a new powerful coding system. A query analysis have showed that there are three types of lines used in constructing query regions in a temporal space. On this basis, a new parallel processing scheme was presented. The scheme employed three types of C-trees, such that each type of C-trees handles a special type of queries. Parallelization is enhanced by vertically and horizontally partitioned data. Algorithms for update operations were explained.

Empirical results showed that the space cost of the new indexing mechanism slightly increases by the increase in mean interarrival time, or in mean duration. This is mainly due to the increase in number of time levels. It has also been shown that the space cost is much less than the TP-index, or the time index. The level granularity has been found to be suitable for tuning the space cost, based on an inverse proportional relationship.

Empirical results also showed that the search time in the parallel C-trees slightly increases with the decrease in interarrival time, due to the decrease in number of time levels. A comparison with the TP-index, and the time index showed that the parallel C-trees showed much better performance in query processing. The search time can also be tuned by varying the level granularity, based on a directly proportional relationship.

Since the ultimate goal of our research is producing a complete temporal model, further research is recommended in the area of developing a suitable temporal query language. Another important area of enhancement is the design of a query optemizer. The optemizer topic should deal with the following. (1) Query translation, and decomposition. (2) Selecting the appropriate type of C-trees. (3) Merging the results according to the query semantics.

Finally, we recommend further research in the area of establishing concurrency for multiusers. This topic should be closely studied with the optemizer to provide an integral solution.

REFERENCES

Bassiouni, M., Karimi, S., Orooji, A. (1990). Supporting temporal capabilities in a multi-computer database system. Proceedings of the International Conference on databases parallel architecture and their applications (pp. 20-26). Miami, FL: IEEE.

Chaudhry, A., Elmasri, R., Kouramajian, V. (1994). Declustering techniques for parallelizing temporal access structures. Proceedings of the Tenth International Conference on Data Engineering (pp. 232-242). Houston, TX: IEEE.

Christian, S. (1994). A consensus glossary of temporal database concepts. Sigmod Record, 23(1), 52-61.

Comer, D. (1979). The ubiquitous B-tree. ACM Computing Surveys, 11(2), 121-138.

Dayal, U., Wuu, G. (1992). A uniform approach to processing temporal queries. Proceedings of the Eighteenth VLDB Conference (pp. 407-418). Vancouver, Canada: IEEE.

Elmasri, R., Kim, Y., Wuu, G. (1991). Efficient implementation techniques for the time index. Proceedings of the seventh International Conference on Data Engineering (pp. 102-109). Kobe, Japan: IEEE.

Elmasri, R., Kim, Y., Wuu, G. (1990). The time index: an access structure for temporal data. Sixteenth International Conference on Very Large Databases (pp. 1-12). Brisbane, Australia: IEEE.

Faloutsos, C., Kamel, I. (1992). Parallel R-trees. Proceedings of ACM SIGMOD International Conference on Management of Data (pp. 195-204). San Diego, CA: ACM Press.

Gadia, S. (1993). Ben-Zvi's pioneering work in relational temporal databases. Temporal Databases: Theory, Design, and Implementation (pp. 202-207). Redwood City, CA: Benjamin/Cummings.

Gunadhi, H., Segev, A. (1993). Efficient indexing methods for temporal relations. IEEE Transactions on Knowledge and Data Engineering, 5(3), 496-509.

Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. Proceedings of ACM SIGMOD International Conference on Management of Data (pp. 47-57). Boston, MA: ACM Press.

Kolovson, C. (1993). Indexing techniques for historical databases. Temporal Databases: Theory, Design, and Implementation (pp. 418-432). Redwood City, CA: Benjamin/Cummings..

Kolovson, C., Stonebraker, M. (1991). Segment indexes: dynamic indexing techniques for multi-dimensional interval data. Proceedings of ACM SIGMOD International Conference on Management of Data (pp. 138-147). Denver, CO: ACM Press.

Leung, T., Muntz, R. (1992). Temporal query processing and optimization in multiprocessor database machines. Proceedings of the Conference on Very Large Databases (pp. 383-394). Vancouver, Canada: IEEE.

Ling, D. (1988). Query execution and temporal support in a distributed database system. Northern Ireland, UK: University of Ulster at Jordanstown.

Lomet, D., Salzberg, B. (1993). Transaction-time databases. Temporal Databases: Theory, Design, and Implementation. (pp. 388-417). Redwood City, CA: Benjamin/Cummings.

Lu, H., Ooi, B., Shen, H. (1993). The TP-index: a dynamic and efficient indexing mechanism for temporal databases. Technical Report No. TRC6/93 Department of Information Systems and Computer Science (pp. 1-28). Singapore: National University of Singapore.

Meghini, C., Thanos, C. (1991). The complexity of operations on a fragmented relation. ACM Transactions on Database Systems, 16 (1), 56-87.

Weiss, M. (1993). Data structures and algorithm analysis in C. Redwood City, CA: Benjamin/Cummings.

APPENDIX A

DEFINITIONS AND ACRONYMS

AP-Tree

An index for temporal databases, based on indexing either the starting times or the ending times in a relation, Gunadhi (1993).

B+ Tree

An index structure, that is a variation of the B tree, suitable for indexing data points that can be totally ordered, Comer (1979).

Binary Search Tree

A binary tree, in which for every node X in the tree, the values of all the keys in the left subtree are smaller than the key value in X, and the values of all the keys in the right subtree are larger than the key value in X, Weiss (1993).

Binary Tree

A tree, in which no node can have more than two children, Weiss (1993).

Composite Index

A temporal index, that is a variation of the AP-tree. The key in a node is a concatenation of the surrogate and the time attribute, Gunadhi (1993).

Data Model

A Data model on a relation has a query language, and supports the specification of constraints on the relation, Christian (1994).

First Temporal Normal Form Relation (1TNF)

A relation in which for each combination of surrogate and time attributes, there is at most

one temporal attribute value, Gunadhi (1993).

Granularity

The size of a minimal, fixed, non-decomposable time interval, Christian (1994).

Instant

A time point on an underlying time axis, Christian (1994).

Lifespan

Is the time over which an object is defined, Christian (1994).

MBR (Minimal Bounding Rectangle)

A concept used in several spatial data structures. The n-dimensional rectangle stored in a nonleaf node entry is the smallest n-dimensional rectangle that encloses the rectangles in the child node of this entry, Guttman (1984).

Multiplexed R Tree (MXR-Tree)

A spatial index, based on parallelizing the R-tree, Faloutsos (1992).

Nested Surrogate & Time Tree (ST-Tree)

Temporal database index, consists of a B+-tree, and several AP-trees, Gunadhi (1993).

Proximity Index

A strategy to select a disk for inserting a new node, Faloutsos (1992).

R-Tree

A height balanced dynamic index for spatial databases, Guttman (1984).

Segment Indexes

A family of indexes for handling interval data. It is based on combining a segment tree, with multiway balanced trees, Kolovson (1991).

Segment Tree

A binary search tree that stores one dimensional line segments, Kolovson (1993).

Segment Tree and R-Tree (SR-Tree)

A segment index, based on combining the segment tree, and the R-tree, Kolovson (1991).

Skeleton SR-Tree

An SR-tree, based on estimating the input data distribution, and initially building the index, Kolovson (1991).

Sparse Tree

A temporal index, that is a variation of the AP-tree, based on indexing only the surrogate value. The time attribute were stored in an accessible list, Gunadhi (1993).

Spatial Database

A database of tuples representing spatial objects, Guttman (1984).

Spatial Index

Index structure that retrieves objects according to their spatial locations, Guttman (1984)

Spatial Object

Object with non-zero size, located in a multi-dimensional space, Guttman (1984)

Spatial Search

Search for spatial objects in a multi-dimensional space, Guttman (1984).

Surrogate

The identity key in a relation, Gunadhi (1993).

Temporal Attribute

A time varying attribute in a relation, Gunadhi (1993).

Temporal Database

A database that supports some aspect of time, Christian (1994).

Time Attribute

A timestamp associated with some object. It can be a time point or a time interval, Christian (1994).

Time Index

An index for temporal databases, based on using indexing points. An indexing point is the starting or ending time of an interval, Elmasri (1990).

Time Interval

The time between two instants (events), Christian (1994).

Time Polygon Index (TP-index)

An index for temporal databases. It is based on decomposing the space into polygons, Lu, Ooi, and Shen (1993).

Time Split B Tree (TSB-Tree)

A two dimensional search structure, which indexes records, each of which has a key, attribute and a time interval attribute (its transaction time), Lomet (1993).

Timestamp

Is a time value associated with some object, Christian (1994).

Transaction Time

The time a fact is stored in the database, Christian (1994).

Valid Time

The time a fact is true in reality, Christian (1994).

APPENDIX B

Implementation Source Code

```
/*****************************************************************************/
/*                                  Parallelizer                             */
/*                                                                           */
/* This program invokes the update programs of the three C-tree types, "A", "B", and "C" in parallel. */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

main()
{
        int p1,p2,p3;
        p1=fork();
        if(p1==0)
    {
          execl("g1","1",0);
    }
    else
        {
          p2=fork();
          if(p2==0)
       {
                execl("g2","1",0);
       }
          else
          {
            p3=fork();
            if(p3==0)
        {
                  execl("g3","1",0);
        }
       }
     }
     exit(0);
}
```

```
/*****************************************************************************/
/*                                C-tree type "A"                            */
/*                                                                           */
/* This program includes the code for insertion, search, and deletion in     */
/* C-trees type "A". This program is called from the "Parallelizer"          */
/*****************************************************************************/


#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>

#define M 62 /*Max number of entries in a node*/
#define GRAN 365 /*space advances every 8 time units*/

struct entry        {
                    char w[30];     /*entry binary expressions*/
                    void *p_to_down; /*may point to tre_node or buc_node*/
                    int no_of_ps;    /*is no. of points in bucket of leaf*/
                                     /*is zero for nonleaf*/
          float f;
                    };

struct tre_node     {
                    struct entry E[M+1], /*array of structures*/
                    int flag;          /*is 0 for nonleaf,1 for leaf*/
          int nonempty;      /*no. of nonempty slots in the node*/
          int from;          /*no. of entry in parent*/
          struct tre_node *up;/*pointer to parent*/
          struct tre_node *join;/*pointer to right node*/
                    };

struct buc_node     {
                    float x;
                    float y;
                    float j;
                    struct buc_node *p_t_same; /*pointer to bucket node*/
                                               /*where same point is*/
          struct buc_node *next;
                    };

struct v_root {
          int level;
          struct tre_node *tree;
          struct v_root *link;
          };

struct v_root *A;
struct v_root *hop_A;
struct v_root *dummy_A;
struct tre_node *S_A;
```

```
struct tre_node *S_A2;
struct tre_node *hold_A[150];
struct tre_node *parent_A;
struct buc_node *p_buck_A ;
struct buc_node *temp;
struct buc_node *jmp_A[M+3];
struct buc_node *mark_A;
struct buc_node *trace_A[M+1];
struct buc_node *trace_A2[M+1];
struct buc_node *del_A;

int nodes1_A;
int nodes2_A;
int nleaf1_A;
int nleaf2_A;
int t_level_A;
int ln_GBE_A[150]; /*length of greatest GBA*/
float X1,Y1,X2,Y2,X3,Y3,X4,Y4; /*space corner points*/
int above_A[M+3];
int below_A[M+3];
int above_n_A=0;
int below_n_A=0;
int points_A=0;

char GBE_A[30];    /*The greatest binary expression*/
char buffer1[30];
char buffer[30];

void insert_A(float,float);
void create_A();
void ins_en_A(int);  /*prepare entry in field node*/
void f_level_A(float,float);     /*find time level of point*/
void poi_BE_A(int,float,float);  /*find BE of point*/
int s_n_in_A(char *,float,float); /*search node*/
int a_buck_A(int,float,float,float); /*add point to bucket*/
void in_buck_A(int,float,float,float);/*insert point in buck list*/
void sp_buc_A(int,float,float,float); /*split bucket*/
void space_A(int);
void dec_A(int,int,int,float);
void phy_sp_A(int,char *,char *);
void sp_nod_A();
void tran_en_A(int,int);
void shift_A(int);
void adjust_A();
void union_A(struct tre_node *);
void print_A();
int search_A(float,float);
void delete_A(float,float);
void cond_A(int);
void deshift_A(int);
void pol_A(float,float,float,float,float,float,float,float);
void pol_ex_A(float,float);
int max_t_A(int,int);
int min_t_A(int,int);
void find_A(struct tre_node *,char *,float,float);
int pref_A(char *,char *);
```

```
/*_____________________________________________________________________*/
void in_buck_A(int i,float a,float b,float c)
{
  temp=(struct buc_node *)malloc(sizeof(struct buc_node));
  temp->next=NULL;
  if(S_A->nonempty==0)
  {
     p_buck_A=(struct buc_node *)malloc(sizeof(struct buc_node));
     S_A->E[i].p_to_down=p_buck_A;
     p_buck_A->next=NULL;
  }
  p_buck_A=S_A->E[i].p_to_down;
  trace_A[i]=S_A->E[i].p_to_down,
  while(p_buck_A->next!=NULL)
  {
    if(a<=p_buck_A->next->x)
    {
     temp->next=p_buck_A->next;
     p_buck_A->next=temp;
     p_buck_A->next->x=a;
     p_buck_A->next->y=b;
     p_buck_A->next->j=c;
     mark_A=p_buck_A; /*mark points before the point node*/
     temp=NULL;
     if(a>S_A->E[i].f)
         S_A->E[i].f=a;
     return;
    }
    else if(a>p_buck_A->next->x)
    {
     p_buck_A=p_buck_A->next;
    }
  } /*end of while loop*/
  temp->next=p_buck_A->next;
  p_buck_A->next=temp;
  p_buck_A->next->x=a;
  p_buck_A->next->y=b;
  p_buck_A->next->j=c;
  mark_A=p_buck_A;
  temp=NULL;
  if(a>S_A->E[i].f)
    S_A->E[i].f=a;
  return;
}
/*_____________________________________________________________________*/
void ins_en_A(int i)
{
 strncpy(S_A->E[i].w,"",30);
 strcpy(S_A->E[i].w,buffer1);
 if(strlen(S_A->E[i].w)>ln_GBE_A[t_level_A])
   ln_GBE_A[t_level_A]=strlen(S_A->E[i].w);
}
/*_____________________________________________________________________*/
void create_A()
{
 int i;
```

```
GBE_A[0]='*';
ln_GBE_A[t_level_A]=strlen(GBE_A);
hop_A->tree=(struct tre_node *)malloc(sizeof(struct tre_node));
hop_A->tree->flag=1;
S_A=hop_A->tree;
i=0;
while(i<=M)
{
 strncpy(hop_A->tree->E[i].w,"",30);
 hop_A->tree->E[i].no_of_ps=0;
 hop_A->tree->E[i].p_to_down=NULL;
 hop_A->tree->E[i].f=0;
 ++i;
}
hop_A->tree->up=NULL;
hop_A->tree->join=NULL;
hop_A->tree->nonempty=0;
}
/*_____________________________________________________________________*/
void f_level_A(float p,float q)  /*This function is independent on index used*/
{
int number=-1;
int NOW_L;
t_level_A=(p+q)/GRAN;
hop_A=A;
while((hop_A->link!=NULL)&&(hop_A->link->level<=t_level_A))
{
 hop_A=hop_A->link;
 if(hop_A->level==t_level_A)
 {
   number=t_level_A;
   break;
 }
}/*end of while*/
if(number==-1)
{
  dummy_A=(struct v_root *)malloc(sizeof(struct v_root));
  dummy_A->link=NULL;
  dummy_A->tree=NULL;
  dummy_A->link=hop_A->link;
  hop_A->link=dummy_A;
  hop_A=dummy_A;
  dummy_A=NULL;
  hop_A->level=t_level_A;
}/*end of if*/
}
/*_____________________________________________________________________*/
void space_A(int NOW_L)
{
if(t_level_A==0)
{
 X1=0;
 Y1=NOW_L;
 X2=0;
 Y2=0;
 X3=NOW_L;
```

```
 Y3=0;     /*anticlockwise order from top*/
}
else
{
 X1=0;
 Y1=NOW_L;
 X2=0;
 Y2=NOW_L-GRAN;
 X3=NOW_L-GRAN;
 Y3=0;
 X4=NOW_L;
 Y4=0;
}
}
/*_____________________________________________________________________*/
void poi_BE_A(int NOW_L,float a, float b)
{
char bin_exp[30];
float value;
int i=0;
strncpy(bin_exp,"",30);
space_A(NOW_L);
if(t_level_A==0)
    value=Y2+((Y1-Y2)/2);
else
    value=Y3+((Y1-Y3)/2);
while((strlen(bin_exp)<(ln_GBE_A[t_level_A]-1))&&(ln_GBE_A[t_level_A]!=0))
{
 if(b<=value)
 {
  bin_exp[i]='0';
  Y1=value;
  if(t_level_A==0)
     value=Y2+((Y1-Y2)/2);
  else
     value=Y3+((Y1-Y3)/2);
 }
 else
 {
  bin_exp[i]='1';
  if(t_level_A==0)
  {
    Y2=value;
    value=Y2+((Y1-Y2)/2);
  }
  else
  {
    Y3=value;
    value=Y3+((Y1-Y3)/2);
  }
 }
 i=i+1;
} /*end of while loop*/
bin_exp[i]='*';
strncpy(buffer1,"",30);
strcpy(buffer1,bin_exp);
```

```
}
/*_____*/
int s_n_in_A(char BE_A[],float a,float b)
{
 int differ=0;
 int l=0;
 int i=0;
 int k=0;
 char *D_CAP=NULL;
 if(S_A->nonempty==0)
  return(i); /*the first entry i.e no. zero*/
 else
  {
   while(i<S_A->nonempty)  /*to search nonempty entries of a node*/
   {
   if((S_A->E[i].w[0]=='*')&&((strstr(S_A->E[i].w,"D")==NULL)))
       return(i);
   l=0;
   while(S_A->E[i].w[l]!='*')
   {
      if(S_A->E[i].w[l]!=BE_A[l])
      {
         differ=1;
         break;
      }
      l=l+1;
   }/*end of while*/
   D_CAP=NULL;
   D_CAP=strstr(S_A->E[i].w,"D");
   if(differ==0) /*the entry is qualified*/
   {
    if(D_CAP==NULL) /*no D character in BE*/
          return(i);
    else if((D_CAP!=NULL)&&(a<=S_A->E[i].f))
          /*point lies inside D character range f*/
          return(i);
    else
          differ=1;
   }/*end of if differ..*/
   i=i+1, /*to search next entry*/
   differ=0;
  }/*end of outer while*/
  if((differ==1)&&(S_A->flag==1)&&(S_A->nonempty<M))
    return(i+1);
  /*else if((differ==1)&&(S_A->flag==1)&&(S_A->nonempty==M))*/
    /*return(M+1); */
 }/*end of else*/
}
/*_____*/
void phys_sp_A(int i,char *BE1,char *BE2)
{
   int l=0;
   int v;
   int u;
   int array[M+3];
   array[0]=0; /*not used*/
```

```
if(points_A==below_n_A)
{
 S_A->E[i].f=jmp_A[below_A[points_A]]->x;
 S_A->E[i+1].f=jmp_A[above_A[M+1-points_A]]->x;
}
else
{
 S_A->E[i].f=jmp_A[above_A[points_A]]->x;
 S_A->E[i+1].f=jmp_A[below_A[M+1-points_A]]->x;
}
l=l+1;
strncpy(S_A->E[i].w,"",30);
strcpy(S_A->E[i].w,BE1);
S_A->E[i].no_of_ps=points_A;
strncpy(buffer1,"",30);
strcpy(buffer1,BE2);
ins_en_A(i+1);
p_buck_A=(struct buc_node *)malloc(sizeof(struct buc_node));
S_A->E[i+1].p_to_down=p_buck_A;
p_buck_A=S_A->E[i+1].p_to_down;
trace_A[i+1]=S_A->E[i+1].p_to_down;
if(BE1[strlen(BE1)-2]=='1') /*points_A is no. of points in ist entry*/
{
 while(l<=(M+1-points_A))
 {
    array[l]=below_A[l];
    l=l+1;
 }
}
else
{
 while(l<=(M+1-points_A))
 {
    array[l]=above_A[l];
    l=l+1;
 }
}
l=l-1;
v=1;
u=1;
while(v<=l)
{
 p_buck_A->next=jmp_A[array[v]];
 while(1)
 {
  if(jmp_A[array[v]-u]!=NULL)
  {
    jmp_A[array[v]-u]->next=jmp_A[array[v]]->next;
    p_buck_A=p_buck_A->next;
    p_buck_A->next=NULL;
    jmp_A[array[v]]=NULL;
    break;
  }
  else
    u=u+1;
 }/* end of while(1)*/
```

```
    v=v+1;
    u=1;
   } /*end of while*/
   S_A->E[i+1].no_of_ps=l;
   ++S_A->nonempty;
}
/*_____*/
void dec_A(int left,int right,int sub,float value)
{
  int la=1, /*the subscript the array above starts with*/
  int lb=1; /*the subscript the array below starts with*/
  above_n_A=0;  /*reset*/
  below_n_A=0;  /*reset*/
  while(left<=right)
  {
    if(jmp_A[sub]->y>value)
    {
      above_A[la]=left;
      above_n_A=above_n_A+1;
      la=la+1;
    }
    else
    {
      below_A[lb]=left;
      below_n_A=below_n_A+1;
      lb=lb+1;
    }
    left=left+1;
    sub=sub+1;
  }
}
/*_____*/
void tran_en_A(int count,int n)
{
    struct tre_node *helper;
    /*copy entry to node 2*/
    strncpy(S_A2->E[n].w,"",30);
    strcpy(S_A2->E[n].w,S_A->E[count].w);
/*count is the entry no. in the original node*/
/*n is the entry no. in the 2nd node to transfer to*/
    S_A2->E[n].p_to_down=S_A->E[count].p_to_down;
    trace_A2[n]=S_A2->E[n].p_to_down;
    S_A2->E[n].no_of_ps=S_A->E[count].no_of_ps;
    S_A2->E[n].f=S_A->E[count].f;
    S_A2->flag=S_A->flag;
    ++S_A2->nonempty;
    /*remove copied entry from node 1*/
    strncpy(S_A->E[count].w,"",30);
    S_A->E[count].p_to_down=NULL;
    S_A->E[count].no_of_ps=0;
    S_A->E[count].f=0;
    --S_A->nonempty;
    if(S_A2->flag==0)
    {
      helper=S_A2->E[n].p_to_down;
      helper->from=n;
```

```
      helper->up=S_A2;
      helper=NULL;
    }
}
/*_____________________________________________________________________*/
void sp_nod_A()
{
  int i=0;
  int k=0;
  int o=0;
  int differ=0;
  S_A2=(struct tre_node *)malloc(sizeof(struct tre_node));/*points to node 2*/
  S_A2->nonempty=0;
  S_A2->join=S_A->join;
  S_A->join=S_A2;
  i=M-1; /*start with the entry before the rightmost i.e before no.M*/
  while((S_A->E[i].w[0]=='*')&&(S_A->E[i].w[1]=='D'))
   i=i-1;
  while(i>0) /*to change the picked entry to compare with all its left entr*/
  {
   k=0;/*start with the first digit in binary expression*/
   while(S_A->E[i].w[k]!='*') /*change the digit*/
   {
    o=i-1;
    while(o>=0)/*to move along the same digit on all entries left of i*/
    {
      if(S_A->E[i].w[k]!=S_A->E[o].w[k])
      {
        differ=i;
        break;
      }
      o=o-1;/*jump to the left entry*/
    }/*end of while*/
    if(differ==0)
      k=k+1;/*take the next digit*/
    else
      break;
   }/*end of inner while*/
   if(differ!=0)
   {
     i=i-1; /*pick the entry to the left and start again*/
     differ=0;
   }
   else
     break;
  }/*end of outer while*/
  while(i<M)
  {
      tran_en_A(i+1,S_A2->nonempty);
      i=i+1;
  }
}
/*_____________________________________________________________________*/
void union_A(struct tre_node *p)
{
 strncpy(buffer1,"",30);
```

```
 strcpy(buffer1,p->E[p->nonempty-1].w);
}
/*_____________________________________________________________________*/
void adjust_A()
{
  int i=0;
  struct tre_node *aux;
  parent_A=S_A;
  if(parent_A!=hop_A->tree)
    parent_A=S_A->up;
  while((parent_A!=S_A)&&(parent_A!=NULL)&&(S_A!=hop_A->tree))
  {
      i=S_A->from;
      strncpy(parent_A->E[i].w,"",30);
      strcpy(parent_A->E[i].w,S_A->E[(S_A->nonempty)-1].w);
      parent_A->E[i].f=S_A->E[(S_A->nonempty)-1].f;
      if(S_A2!=NULL)
      {
        S_A=parent_A;
        shift_A(i+1);
        S_A=parent_A->E[i].p_to_down;
        strncpy(parent_A->E[i+1].w,"",30);
        strcpy(parent_A->E[i+1].w,S_A2->E[(S_A2->nonempty)-1].w);
        parent_A->E[i+1].f=S_A2->E[(S_A2->nonempty)-1].f;
        ++(parent_A->nonempty);
        parent_A->E[i+1].p_to_down=S_A2;
        S_A2->up=parent_A;
        S_A2->from=i+1;
        S_A2=NULL;
      }
      S_A=parent_A;
      parent_A=S_A->up;
      if(S_A->nonempty==M+1)
        sp_nod_A();
  }
 if((S_A->nonempty==M+1)&&(S_A==hop_A->tree))
        sp_nod_A();

 i=0;
 if((S_A==hop_A->tree)&&(S_A2!=NULL))  /*the split node is the root*/
 {
    aux=(struct tre_node *)malloc(sizeof(struct tre_node));
    hop_A->tree=aux;
    aux=NULL;
    parent_A=hop_A->tree;
    S_A->up=parent_A;
    S_A2->up=parent_A,
    S_A->from=i;
    S_A2->from=i+1;
    hop_A->tree->E[i].p_to_down=S_A,
    hop_A->tree->E[i+1].p_to_down=S_A2;
    hop_A->tree->flag=0;
    hop_A->tree->nonempty=2;
    union_A(S_A);
    strncpy(hop_A->tree->E[i].w,"",30);
    strcpy(hop_A->tree->E[i].w,buffer1);
```

```
    strncpy(hop_A->tree->E[i+1].w,"",30);
    strcpy(hop_A->tree->E[i+1].w,"*");
    hop_A->tree->E[i].f=S_A->E[S_A->nonempty-1].f;
    hop_A->tree->E[i+1].f=S_A2->E[S_A2->nonempty-1].f;
  } /*end of if*/
}
/*_____________________________________________________________________*/
void sp_buc_A(int i,float a,float b,float c)
{
 char BE1[30]; /*of new 1st entry*/
 char BE2[30]; /*of new 2nd entry*/
 char ent_exp[30]; /*of original entry*/
 int dif=0;
 int t=0;
 int k=0;
 int count=0;
 int counter=0;
 int ans;
 int tt;
 int total;
 jmp_A[k]=S_A->E[i].p_to_down;
 below_n_A=0;
 above_n_A=0;
 k=k+1;
 counter=counter+1;
 total=M+1,
 while(k<=M+1)
 {
    jmp_A[k]=jmp_A[k-1]->next;
    if((jmp_A[k]->y!=jmp_A[k-1]->y)&&(k>1))
       dif=1;
    k=k+1;
 }
 if((dif==0)&&((k-1)>M)) /*all y-coord are equal*/
 {
    strncpy(BE1,"",30);
    strcpy(BE1,S_A->E[i].w);
    if(strstr(BE1,"D")==NULL)
         BE1[strlen(BE1)]='D';
    strncpy(BE2,"",30);
    strcpy(BE2,S_A->E[i].w);
    while(counter<=(total/2))
    {

     below_A[counter]=counter;
     ++below_n_A;
     ++counter;
    }
    counter=1;
    while(counter<=total-below_n_A)
    {
     above_A[counter]=counter+below_n_A,
     ++above_n_A,
     ++counter;
    }
    points_A=below_n_A;
```

```
}/*end of if*/
else
{
    strncpy(BE1,"",30);
        k=1;
        if(t_level_A==0)
                dec_A(k,M+1,k,(((Y1-Y2)/2)+Y2));
        else
                dec_A(k,M+1,k,(((Y1-Y3)/2)+Y3));
        while(1)
        {
                if(above_n_A==total)
                {
                        BE1[t]='1';
                        count=1;
                        if(t_level_A==0)
                        {
                        Y2=Y2+((Y1-Y2)/2);
                        dec_A(count,above_n_A,above_A[count],(((Y1-Y2)/2)+Y2));
                        }
                        else
                        {
                        Y3=Y3+((Y1-Y3)/2);
                        dec_A(count,above_n_A,above_A[count],(((Y1-Y3)/2)+Y3));
                        }
                }
                else if(below_n_A==total)
                {
                        BE1[t]='0';
                        count=1;
                        if(t_level_A==0)
                        {
                        Y1=Y2+((Y1-Y2)/2);
                        dec_A(count,below_n_A,below_A[count],(((Y1-Y2)/2)+Y2));
                        }
                        else
                        {
                        Y1=Y3+((Y1-Y3)/2);
                        dec_A(count,below_n_A,below_A[count],(((Y1-Y3)/2)+Y3));
                        }
                }
                else
                {
                if(below_n_A>above_n_A)
                        BE1[t]='0';
                else if(above_n_A>below_n_A)
                        BE1[t]='1';
                else
                {
                        if(t!=0)
                                BE1[t]=BE1[t-1];
                        else
                                BE1[t]='0';
                }
                break;
                }/*end of else*/
```

```
                t=t+1;
        }/*end of while*/
        BE1[t+1]='*';
    strncpy(ent_exp,"",30);
        strcpy(ent_exp,S_A->E[i].w);
        if(strlen(BE1)>ln_GBE_A[t_level_A])
            ln_GBE_A[t_level_A]=strlen(BE1);
    strncpy(BE2,"",30);
    strcpy(BE2,ent_exp);
        if(BE1[t]=='1')
                points_A=above_n_A;  /*points_A is no. of points in 1st entry*/
        else
                points_A=below_n_A;
 }/*end of else*/
 phys_sp_A(i,BE1,BE2);
 if(S_A->nonempty==M+1)
 {
                sp_nod_A();
                adjust_A(); /*adjust tree*/
 }
}
/*_____________________________________________________________________*/
int a_buck_A(int i,float a,float b,float c)
{
 int split=0; /*no split*/
 if(S_A->E[i].no_of_ps<M)
 {
  in_buck_A(i,a,b,c); /*to insert in bucket list*/
  return(0);
 }
 else
 {
  in_buck_A(i,a,b,c);
  sp_buc_A(i,a,b,c); /*split bucket*/
  return(1); /*split occured*/
 }
}
/*_____________________________________________________________________*/
void shift_A(int i)
{
 /*i is the entry no. to be shifted to the right*/
 /*count is the no. of the rightmost non empty entry*/
 int count=S_A->nonempty-1;
 while(count>=i)
 {
    strncpy(S_A->E[count+1].w,"",30);
    strcpy(S_A->E[count+1].w,S_A->E[count].w);
    S_A->E[count+1].p_to_down=S_A->E[count].p_to_down;
    S_A->E[count+1].no_of_ps=S_A->E[count].no_of_ps;
    S_A->E[count+1].f=S_A->E[count].f;
    if(S_A->flag==0)
    {
      S_A=S_A->E[count+1].p_to_down;
      S_A->from=count+1,
      S_A=S_A->up;
    }
```

```
    count=count-1;
 }/*end of while*/
 strncpy(S_A->E[i].w,"",30);
 S_A->E[i].p_to_down=NULL;
 S_A->E[i].no_of_ps=0;
 S_A->E[i].f=0;
}
/*_____________________________________________________________________*/
void insert_A(float a,float b)
{
char BE_A[30]; /*BE of piont in index A*/
float c;
int NOW_L;
int split;
int i1;          /*entry numbers returned by search node function*/
if(A==NULL)
{
      A=(struct v_root *)malloc(sizeof(struct v_root));
      A->link=NULL;
      hop_A=A,
}
f_level_A(a,b);  /*Find time level where point is*/
if(hop_A->tree==NULL)
 create_A();
c=(a+b)/1.414;
NOW_L=(t_level_A+1)*GRAN;   /*Calculate NOW_L of the time level*/
space_A(NOW_L);
poi_BE_A(NOW_L,a,b);
space_A(NOW_L);
strncpy(BE_A,"",30);
strcpy(BE_A,buffer1);
S_A=hop_A->tree;
hold_A[t_level_A]=hop_A->tree;
while(1) /*to repeat in case of nonleaf node but with descending to child*/
{
i1=s_n_in_A(BE_A,a,b);
if(i1<=M)
{
 if(S_A->flag==0)  /*in case of nonleaf node */
   S_A=S_A->E[i1].p_to_down; /*descend the tree*/
 else if((S_A->flag==1)&&(S_A->nonempty==0))
 {
   ins_en_A(i1);
   split=a_buck_A(i1,a,b,c);
   if(split==0) /*no split occured*/
   {
    ++(S_A->E[i1].no_of_ps);
    ++(S_A->nonempty);
   }
   break;
 }
 else if((S_A->flag==1)&&(i1>S_A->nonempty))
 {
   ins_en_A(i1);
   split=a_buck_A(i1,a,b,c);
   if(split==0)
```

```
    {
     ++(S_A->E[i1].no_of_ps);
     ++(S_A->nonempty);
    }
    break;
  }
  else if((S_A->flag==1)&&(i1<=S_A->nonempty)&&(S_A->nonempty<=M))
  {
    if((i1!=M-1)&&(S_A->E[i1].no_of_ps==M))
       shift_A(i1+1);
    split=a_buck_A(i1,a,b,c);
    if(split==0)
     ++(S_A->E[i1].no_of_ps);
    break;
  }

 }/*end of if i1<=M*/
 } /*end of while*/
 hold_A[t_level_A]=hop_A->tree;
}
/*_____________________________________________________________________*/
void print_A()
{
  int qty=0;
  struct tre_node *pin;
  struct tre_node *pin2;
  struct v_root *pin3;
  struct buc_node *pin4;
  int i=0;
  int turn=0;
  int round=0;
  FILE *stream1;
  stream1=fopen("put1","w");
  pin3=A;
  if(pin3==NULL)
   fprintf(stream1,"\n Index A does not exist ");
  else
  {
   pin3=A->link;
   ++qty;
  /* printf("\n Index . A ");*/
  /* printf("\n ----------");*/
   while(pin3!=NULL)
   {
   /*printf("\ntime level  %d",pin3->level);*/
   pin=hold_A[pin3->level];
   pin2=pin;
  /* printf("\n              ");*/
   while(1) /*vertical move*/
   {
   /* printf("\n.....new tree level.....\n");*/
    while(pin!=NULL)/*horizontal move*/
    {
     ++qty;
     /*printf("\n-------new node----------\n");*/
     i=0;
```

```
  if(pin->flag==1)
  {
    /*  printf("field node\n");*/
     ++qty;
  }
  while(i<pin->nonempty)/*inside node*/
  {
    if((round==4)&&(pin->flag==0))
    {
  /*    printf("\n");*/
      round=0;
    }
    /*printf("| %s |",pin->E[i].w);*/
    /*printf("%.1f",pin->E[i].f);*/
    ++round;
    if(pin->flag==1)
    {
     /* printf("\n");*/
      pin4=pin->E[i].p_to_down;
      pin4=pin4->next;
      while(pin4!=NULL)
      {
       if(turn==5)
       {
          /*printf("\n");*/
          turn=0;
       }
       /*printf("(%.1f",pin4->x);*/
       /*printf(",%.1f)",pin4->y);*/
       pin4=pin4->next;
       ++turn;
      }
      /*printf("\n");*/
    }
    i=i+1;
    turn=0;
   }/*end of while*/
   pin=pin->join;
   if(pin->flag==0)
     ; /* printf("\n");*/
  }/*end of while*/
  pin=pin2;
  if(pin->flag==1)
    break;
  else
  {
    pin=pin2->E[0].p_to_down;
    pin2=pin;
  }
 }/*end of while*/
 fprintf(stream1,"\ntime level : %d",pin3->level);
 pin3=pin3->link;
 ++qty;
 }/*end of while*/
}/*end of else*/
fprintf(stream1,"\ntotal no. of nodes= %d\n",qty);
```

```
  fclose(stream1);
}
/*_____________________________________________________________________*/
int search_A(float a,float b)
{
  int NOW_L;
  int nodes=0;
  char string[30];
  int e;
  struct buc_node *look;
  f_level_A(a,b);
  NOW_L=(t_level_A+1)*8;
  space_A(NOW_L);
  poi_BE_A(NOW_L,a,b);
  strncpy(string,"",30);
  strcpy(string,buffer1);
  S_A=hold_A[t_level_A];
  while(S_A->flag==0)
  {
      e=s_n_in_A(string,a,b);
      ++nodes;
      S_A=S_A->E[e].p_to_down;
  }/*end of while*/
  e=s_n_in_A(string,a,b);
  ++nodes;
  if(e<S_A->nonempty)
  {
     look=S_A->E[e].p_to_down;
     trace_A[e]=look;
     look=look->next; /*by pass the dummy head*/
     while(look!=NULL)
     {
      if((look->x==a)&&(look->y==b))
      {
        printf("\n  The point (%.1f,%.1f) is in index A",a,b);
        printf("\n  at time level=%d",t_level_A);
        printf("\n  number of nodes searched=%d",nodes);
        del_A=look;
        break;
      }
      else
      {
        trace_A[e]=look;
        look=look->next;
      }
     }/*end of while*/
  }
  else
     printf("\n  The point (%.1f,%.1f) is not in index A",a,b);
  return(e);
}
/*_____________________________________________________________________*/
void deshift_A(int n)
{
   while(n<=S_A->nonempty-1)
   {
```

```
    strncpy(S_A->E[n-1].w,"",30);
    strcpy(S_A->E[n-1].w,S_A->E[n].w);
    S_A->E[n-1].p_to_down=S_A->E[n].p_to_down;
    S_A->E[n-1].no_of_ps=S_A->E[n].no_of_ps;
    S_A->E[n-1].f=S_A->E[n].f;
    if((S_A->flag==0)&&(S_A->E[n-1].p_to_down!=NULL))
    {
      S_A=S_A->E[n-1].p_to_down;
      S_A->from=n-1;
      S_A=S_A->up;
    }
    n=n+1;
   }/*end of while*/
   strncpy(S_A->E[n-1].w,"",30);
   S_A->E[n-1].p_to_down=NULL;
   S_A->E[n-1].no_of_ps=0;
   S_A->E[n-1].f=0;
   --S_A->nonempty;
}
/*_____________________________________________________________________*/
void cond_A(int e)
{
 struct tre_node *help;
 if((S_A->E[e].no_of_ps!=0)&&(S_A->flag==1))
    return;
 else if((e==S_A->nonempty-1)&&(e!=0)&&(S_A->flag==1))
     /*the rightmost but not no. 0 */
 {
  strncpy(S_A->E[e-1].w,"",30);
  strcpy(S_A->E[e-1].w,S_A->E[e].w);
  strncpy(S_A->E[e].w,"",30);
  S_A->E[e].p_to_down=NULL;
  --S_A->nonempty;
  return;
 }
 else if((e<S_A->nonempty-1)&&(S_A->flag==1))
 {
  deshift_A(e+1);/*start shift left from no. e+1 */
  return;
 }
 else if((e==0)&&(e==S_A->nonempty-1)&&(S_A->flag==1)&&(S_A->E[e].w[0]!='*'))
 {
    e=S_A->from;
    if(e>0)
    {
       help=S_A;
       S_A=S_A->up;
       S_A=S_A->E[e-1].p_to_down;
       S_A->join=help->join;
       help=NULL;
       S_A=S_A->up;
       S_A->E[e].p_to_down=NULL;
       deshift_A(e+1);
    }/*end of if*/
    else /*e==0*/
        ; /*leave node as it is to preserve tree balance*/
```

```
 } /*end of else*/
}
/*_____*/
void delete_A(float a,float b)
{
  int e;
  del_A=NULL;
  e=search_A(a,b);
  if(del_A==NULL)
    printf("\n Point(%.1f,%.1f) was not found in index A",a,b);
  else
  {
    trace_A[e]->next=del_A->next;
    del_A=NULL;
    --S_A->E[e].no_of_ps;
    cond_A(e);/*e is the empty entry*/
    printf("\n Point(%.1f,%.1f) deleted from index A",a,b);
  }/*end of else*/
}
/*_____*/
int max_t_A(int t,int max)
{
 if(t>max)
   max=t;
 return(max);
}
/*_____*/
int min_t_A(int t,int min)
{
 if(t<min)
   min=t;
 return(min);
}
/*_____*/
void pol_ex_A(float v1,float v2)
{
 int NOW_L;
 float value;
 char bin_exp[30];
 int i=0;
 NOW_L=(t_level_A+1)*GRAN;
 space_A(NOW_L);
 strncpy(bin_exp,"",30);
 if(t_level_A==0)
 {
    value=Y2+((Y1-Y2)/2);
 }
 else
 {
    value=Y3+((Y1-Y3)/2);
 }
 while(strlen(bin_exp)<(ln_GBE_A[t_level_A]-1))
 {
  if(v1<=value) /*all the polygon below the division line*/
  {
    bin_exp[i]='0';
```

```
    Y1=value;
    if(t_level_A==0)
        value=Y2+((Y1-Y2)/2);
    else
        value=Y3+((Y1-Y3)/2);
   }
   else if(v2>=value) /*all the polygon above the division line*/
   {
    bin_exp[i]='1';
    if(t_level_A==0)
    {
      Y2=value;
      value=Y2+((Y1-Y2)/2);
    }
    else
    {
      Y3=value;
      value=Y3+((Y1-Y3)/2);
    }
   }
   else
     break;
   i=i+1;
 }/*end of while*/
 bin_exp[i]='*';
 strncpy(buffer1,"",30);
 strcpy(buffer1,bin_exp);
}
/*_____________________________________________________________________*/
int pref_A(char *sh,char *lo)
{
 if(strstr(lo,sh)==lo)
   return(1);
 else
   return(0);
}
/*_____________________________________________________________________*/
void find_A(struct tre_node *s,char *string,float b1,float b2)
{
  struct buc_node *p;
  static int turn;
  char shorter[30];
  char longer[30];
  static int number;
  int i=0;
  int boolean=0;
  while(i<s->nonempty)
  {
    strncpy(shorter,"",30);
    strncpy(longer,"",30);
    if(s->flag==0)
      number=0;
    else if(s->flag==1)
      ++number;
    if(strlen(s->E[i].w)<=strlen(string))
    {
```

```
        strncpy(shorter,s->E[i].w,strlen(s->E[i].w)-1);
        strncpy(longer,string,strlen(string)-1);
    }
    else
    {
        strncpy(longer,s->E[i].w,strlen(s->E[i].w)-1);
        strncpy(shorter,string,strlen(string)-1);
    }
    boolean=pref_A(shorter,longer);
    if((boolean==1)&&(number==0))
    {
      if(nodes2_A>0)
         ++nodes2_A;
      else
         ++nodes1_A;
     if((((struct tre_node *)(s->E[i].p_to_down))->flag==0)&&(s->flag!=1))
     {
       if(nleaf2_A>0)
         ++nleaf2_A;
       else
         ++nleaf1_A;
     }

     find_A(s->E[i].p_to_down,string,b1,b2);
    }
    else if((boolean==1)&&(number>0))
    {
        p=s->E[i].p_to_down;
        p=p->next;
        while(p!=NULL)
        {
         if(turn==5)
         {
           /*printf("\n");*/
           turn=0;
         }
         if((p->y<=b1)&&(p->y>=b2))
         {
           /*printf("(%.1f",p->x);*/
           /*printf(",%.1f)",p->y);*/
           ++turn;
         }
         p=p->next;
        }
    }
    i=i+1;
  } /*end of while*/
}
/*_____________________________________________________________________*/
void pol_A(float a1,float b1,float a2,float b2,float a3,float b3,float a4,float b4)
{
 int NOW_L;
 int n=0;
 int max=0;
 int min=0;
 int value=0;
```

```
int count=0;
int first;
char string[30];
int i=0;
char *array[30];
FILE *out1;
out1=fopen("put1","w");
fprintf(out1,"\n----------search for polygon----------\n");
fprintf(out1,"The polygon that is between the 2 horizontal line y=%.1f,y=%.1f has :",b1,b2);
f_level_A(a1,b1);
max=max_t_A(t_level_A,max);
min=min_t_A(t_level_A,min);
f_level_A(a2,b2);
max=max_t_A(t_level_A,max);
min=min_t_A(t_level_A,min);
f_level_A(a3,b3);
max=max_t_A(t_level_A,max);
min=min_t_A(t_level_A,min);
if(a4!=-0.0)
{
 f_level_A(a4,b4);
 max=max_t_A(t_level_A,max);
 min=min_t_A(t_level_A,min);
}
t_level_A=min;
while(t_level_A<=max)
{
 NOW_L=(t_level_A+1)*GRAN;
 space_A(NOW_L);
 if(t_level_A==0)
 {
  value=Y2+((Y1-Y2)/2);
 }
 else
 {
  value=Y3+((Y1-Y3)/2);
 }
 if((value<b1)&&(value>b2))
 {
  pol_ex_A(b1,value);
  array[count]=calloc(30,sizeof(char));
  strncpy(array[count],"",30);
  strcpy(array[count],buffer1);
  pol_ex_A(value,b2);
  strcat(array[count],"+");
  strcat(array[count],buffer1);
  ++t_level_A;
  ++count;
 }
 else
 {
  pol_ex_A(b1,b2);
  array[count]=calloc(30,sizeof(char));
  strncpy(array[count],"",30);
  strcpy(array[count],buffer1);
  ++t_level_A;
```

```
    ++count;
   }
  }/*end of while*/
  t_level_A=min;
  count=0;
  while(t_level_A<=max)
  {
    fprintf(out1,"\nIn time level %d - - - -\n",t_level_A);
    S_A=hold_A[t_level_A];
    nodes1_A=0;
    nodes2_A=0;
    nleaf1_A=0;
    nleaf2_A=0;
    if(S_A!=NULL)
    {
     if(strstr(array[count],"+")==NULL)
     {
      ++nodes1_A;
      if(S_A->flag==0)
         ++nleaf1_A;
      find_A(S_A,array[count],b1,b2);
     }
     else
     {
      first=0;
      strncpy(string,"",30);
      strcpy(string,strstr(array[count],"+")+1);
      ++nodes1_A;
      if(S_A->flag==0)
         ++nleaf1_A;
      find_A(S_A,string,b1,b2);
      first=strcspn(array[count],"+");
      strncpy(string,"",30);
      strncpy(string,array[count],first);
      ++nodes2_A;
      if(S_A->flag==0)
         ++nleaf2_A;
      find_A(S_A,string,b1,b2);
     }
    }
    if(nodes1_A>nodes2_A)
          fprintf(out1,"\nthe total number of searched nodes=%d\n",nodes1_A);
    else
          fprintf(out1,"\nthe total number of searched nodes=%d\n",nodes2_A);
    if(nleaf1_A>nleaf2_A)
          fprintf(out1,"\nthe number of nonleaf nodes=%d\n",nleaf1_A);
    else
          fprintf(out1,"\nthe number of nonleaf nodes=%d\n",nleaf2_A);

    ++t_level_A;
    ++count;
  }/*end of while*/
}
/*_____________________________________________________________________*/
main(int argc,char **argv)
{
```

```
FILE *in;
int uu1;
char ch;
char file_name[10];
float xx,yy;
float u1,v1,u2,v2,u3,v3,u4,v4;
/*printf("enter file name : ");*/
/*scanf("%s",file_name);*/
/*printf("\n");*/
in=fopen("large1","r");
if (!in)
{
 printf("error open input file\n");
}
while(1)
{
 fscanf(in,"%c",&ch);
 if(ch=='+')
 {
  fscanf(in,"(%f,%f)",&xx,&yy);
  insert_A(xx,yy);
  /*printf("%.1f,%.1f\n",xx,yy);*/
 }
 else if(ch=='#')
 {
  fscanf(in,"(%f,%f)",&xx,&yy);
  uu1=search_A(xx,yy);
 }
 else if(ch=='$')
 {
   print_A();
 }
 else if(ch=='-')
 {
   fscanf(in,"(%f,%f)",&xx,&yy);
   delete_A(xx,yy);
 }
 else if(ch=='?')
 {
   fscanf(in,"(%f,%f,%f,%f,%f,%f,%f,%f)",&u1,&v1,&u2,&v2,&u3,&v3,&u4,&v4);
   pol_A(u1,v1,u2,v2,u3,v3,u4,v4);
 }
 if (feof(in))
   break;
}
free(A);
free(S_A);
free(S_A2);
free(temp);
exit(0);
}
/*_____________________________________________________________________*/
```

```
/*****************************************************************************/
/*                          C-tree type "B"                                  */
/*                                                                           */
/* This program includes the code of insertion, search, and delation in      */
/* C-trees type "B". This program is called from the "Parallelizer"          */
/*****************************************************************************/

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>

#define M 62 /*Max number of entries in a node*/
#define GRAN 365 /*space advances every 8 time units*/

struct entry        {
                    char w[30];     /*entry binary expressions*/
                    void *p_to_down;  /*may point to tre_node or buc_node*/
                    int no_of_ps;    /*is no. of points in bucket of leaf*/
                                     /*is zero for nonleaf*/
          float f;
                    };

struct tre_node     {
                    struct entry E[M+1];  /*array of structures*/
                    int flag;         /*is 0 for nonleaf,1 for leaf*/
          int nonempty;      /*no. of nonempty slots in the node*/
          int from;         /*no. of entry in parent*/
          struct tre_node *up;/*pointer to parent*/
          struct tre_node *join;/*pointer to right node*/
                    };

struct buc_node     {
                    float x;
                    float y;
                    float j;
                    struct buc_node *p_t_same; /*pointer to bucket node*/
                                               /*where same point is*/
          struct buc_node *next;
                    };

struct v_root {
          int level;
          struct tre_node *tree;
          struct v_root *link;
          };

struct v_root *B;
struct v_root *hop_B;
struct v_root *dummy_B;
struct tre_node *S_B;
struct tre_node *S_B2,
struct tre_node *hold_B[150];
```

```
struct tre_node *parent_B;
struct buc_node *p_buck_B;
struct buc_node *temp;
struct buc_node *jmp_B[M+3];
struct buc_node *mark_B;
struct buc_node *trace_B[M+1];
struct buc_node *trace_B2[M+1];
struct buc_node *del_B;

int NOW;      /*normalized most rescent current time*/
int nodes1_B;
int nodes2_B;
int nleaf1_B;
int nleaf2_B;
int t_level_B;
int start=0;  /*normalized start time of database*/
int ln_GBE_B[150];
float X1,Y1,J1,X2,Y2,J2,X3,Y3,J3,X4,Y4,J4; /*space corner points*/
int above_B[M+3];
int below_B[M+3];
int above_n_B=0;
int below_n_B=0;
int points_B=0;

char GBE_B[30];
char buffer2[30];
char buffer[30];

void insert_B(float,float);
void create_B();
void ins_en_B(int);
void f_level_B(float,float);     /*find time level of point*/
void poi_BE_B(int,float,float);
int s_n_in_B(char *,float,float);
int a_buck_B(int,float,float,float);
void in_buck_B(int,float,float,float);
void sp_buc_B(int,float,float,float);
void space_B(int);
void dec_B(int,int,int,float);
void phy_sp_B(int,char *,char *);
void sp_nod_B();
void tran_en_B(int,int);
void shift_B(int);
void adjust_B();
void union_B(struct tre_node *);
void print_B();
int search_B(float,float);
void delete_B(float,float);
void cond_B(int);
void deshift_B(int);
void pol_B(float,float,float,float,float,float,float,float);
void pol_ex_B(float,float);
int max_t_B(int,int);
int min_t_B(int,int);
void find_B(struct tre_node *,char *,float,float);
int pref_B(char *,char *);
```

```
/*_____________________________________________________________________*/
void in_buck_B(int i,float a,float b,float c)
{
  temp=(struct buc_node *)malloc(sizeof(struct buc_node));
  temp->next=NULL;
  if(S_B->nonempty==0)
  {
     p_buck_B=(struct buc_node *)malloc(sizeof(struct buc_node));
     S_B->E[i].p_to_down=p_buck_B;
     p_buck_B->next=NULL;
  }
  p_buck_B=S_B->E[i].p_to_down;
  trace_B[i]=S_B->E[i].p_to_down;
  while(p_buck_B->next!=NULL)
  {
    if(a<=p_buck_B->next->x)
    {
     temp->next=p_buck_B->next;
     p_buck_B->next=temp;
     p_buck_B->next->x=a;
     p_buck_B->next->y=b;
     p_buck_B->next->j=c;
     mark_B=p_buck_B;
     temp=NULL,
     if(a>S_B->E[i].f)
        S_B->E[i].f=a;
     return;
    }
    else if(a>p_buck_B->next->x)
    {
     p_buck_B=p_buck_B->next;
    }
  } /*end of while loop*/
  temp->next=p_buck_B->next;
  p_buck_B->next=temp;
  p_buck_B->next->x=a;
  p_buck_B->next->y=b;
  p_buck_B->next->j=c;
  mark_B=p_buck_B;
  temp=NULL;
  if(a>S_B->E[i].f)
        S_B->E[i].f=a;
  return;
}
/*_____________________________________________________________________*/
void ins_en_B(int i)
{
 strncpy(S_B->E[i].w,"",30);
 strcpy(S_B->E[i].w,buffer2);
 if(strlen(S_B->E[i].w)>ln_GBE_B[t_level_B])
   ln_GBE_B[t_level_B]=strlen(S_B->E[i].w);
}
/*_____________________________________________________________________*/
void create_B()
{
 int i;
```

```
GBE_B[0]='*';
ln_GBE_B[t_level_B]=strlen(GBE_B);
hop_B->tree=(struct tre_node *)malloc(sizeof(struct tre_node));
hop_B->tree->flag=1;
S_B=hop_B->tree;
i=0;
while(i<=M)
{
 strncpy(hop_B->tree->E[i].w,"",30);
 hop_B->tree->E[i].no_of_ps=0;
 hop_B->tree->E[i].p_to_down=NULL;
 hop_B->tree->E[i].f=0;
 ++i;
}
hop_B->tree->up=NULL;
hop_B->tree->join=NULL;
hop_B->tree->nonempty=0;

}
/*_____________________________________________________________________*/
void f_level_B(float p,float q)  /*This function is independent on index used*/
{
int number=-1;
int NOW_L;
t_level_B=(p+q)/GRAN;
hop_B=B;
while((hop_B->link!=NULL)&&(hop_B->link->level<=t_level_B))
{
 hop_B=hop_B->link;
 if(hop_B->level==t_level_B)
 {
   number=t_level_B;
   break;
 }
}/*end of while*/
if(number==-1)
{
  dummy_B=(struct v_root *)malloc(sizeof(struct v_root));
  dummy_B->link=NULL;
  dummy_B->tree=NULL;
  dummy_B->link=hop_B->link;
  hop_B->link=dummy_B;
  hop_B=dummy_B;
  dummy_B=NULL;
  hop_B->level=t_level_B;
}/*end of if*/
}
/*_____________________________________________________________________*/
void space_B(int NOW_L)
{
if(t_level_B==0)
{
 X1=0;
 Y1=NOW_L;
 X2=0;
 Y2=0;
```

```
 X3=NOW_L;
 Y3=0;     /*anticlockwise order from top*/
}
else
{
 X1=0;
 Y1=NOW_L;
 X2=0;
 Y2=NOW_L-GRAN;
 X3=NOW_L-GRAN;
 Y3=0;
 X4=NOW_L;
 Y4=0;
 J4=(X4+Y4)/1.414;
}
 J1=(X1+Y1)/1.414;
 J2=(X2+Y2)/1.414;
 J3=(X3+Y3)/1.414;
}
/*_____________________________________________________________________*/
void poi_BE_B(int NOW_L,float a, float b)
{
char bin_exp[30];
float value;
int i=0;
float c;
space_B(NOW_L);
value=J2+((J1-J2)/2);
c=(a+b)/1.414;
strncpy(bin_exp,"",30);
while(strlen(bin_exp)<(ln_GBE_B[t_level_B]-1))
{
 if(c<=value)
 {
  bin_exp[i]='0';
  J1=value;
  value=J2+((J1-J2)/2);
 }
 else
 {
  bin_exp[i]='1';
  J2=value;
  value=J2+((J1-J2)/2);
 }
 i=i+1;
} /*end of while loop*/
bin_exp[i]='*';
strncpy(buffer,"",30);
strcpy(buffer2,bin_exp);
}
/*_____________________________________________________________________*/
int s_n_in_B(char BE_B[],float a,float b)
{
int differ=0;
int l=0;
int i=0;
```

```
char *D_CAP=NULL;
if(S_B->nonempty==0)
 return(i); /*the first entry i.e no. zero*/
else
 {
  while(i<S_B->nonempty)  /*to search nonempty entries of a node*/
  {
   if((S_B->E[i].w[0]=='*')&&(strstr(S_B->E[i].w,"D")==NULL))
     return(i);
   l=0;
   while(S_B->E[i].w[l]!='*')
   {
      if(S_B->E[i].w[l]!=BE_B[l])
      {
         differ=1;
         break;
      }
      l=l+1;
   }/*end of while*/
   D_CAP=NULL;
   D_CAP=strstr(S_B->E[i].w,"D");
   if(differ==0) /*the entry is qualified*/
   {
    if(D_CAP==NULL) /*no D character in BE*/
          return(i);
    else if((D_CAP!=NULL)&&(a<=S_B->E[i].f))
          /*point lies inside D character range f*/
          return(i);
    else
          differ=1;
   }/*end of if differ..*/
   i=i+1; /*to search next entry*/
   differ=0;
  }/*end of outer while*/
  if((differ==1)&&(S_B->flag==1)&&(S_B->nonempty<M))
    return(i+1);
  /*else if((differ==1)&&(S_B->flag==1)&&(S_B->nonempty==M))*/
    /*return(M+1); */
 }/*end of else*/
}
/*_____________________________________________________________________*/
void phys_sp_B(int i,char *BE1,char *BE2)
{
   int l=0;
   int v;
   int u;
   int array[M+3];
   array[0]=0; /*not used*/
   if(points_B==below_n_B)
   {
    S_B->E[i].f=jmp_B[below_B[points_B]]->x;
    S_B->E[i+1].f=jmp_B[above_B[M+1-points_B]]->x;
   }
   else
   {
    S_B->E[i].f=jmp_B[above_B[points_B]]->x;
```

```
 S_B->E[i+1].f=jmp_B[below_B[M+1-points_B]]->x;
}
l=l+1;
strncpy(S_B->E[i].w,"",30);
strcpy(S_B->E[i].w,BE1);
S_B->E[i].no_of_ps=points_B;
strncpy(buffer2,"",30);
strcpy(buffer2,BE2);
ins_en_B(i+1);
p_buck_B=(struct buc_node *)malloc(sizeof(struct buc_node));
S_B->E[i+1].p_to_down=p_buck_B;
p_buck_B=S_B->E[i+1].p_to_down;
trace_B[i+1]=S_B->E[i+1].p_to_down;
if(BE1[strlen(BE1)-2]=='1')
{
 while(l<=(M+1-points_B))
 {
   array[l]=below_B[l];
   l=l+1;
 }
}
else
{
 while(l<=(M+1-points_B))
 {
   array[l]=above_B[l];
   l=l+1;
 }
}
l=l-1;
v=1;
u=1;
while(v<=l)
{
 p_buck_B->next=jmp_B[array[v]];
 while(1)
 {
  if(jmp_B[array[v]-u]!=NULL)
  {
    jmp_B[array[v]-u]->next=jmp_B[array[v]]->next;
    p_buck_B=p_buck_B->next;
    p_buck_B->next=NULL;
    jmp_B[array[v]]=NULL;
    break;
  }
  else
    u=u+1;
 }/* end of while(1)*/
 v=v+1,
 u=1;
} /*end of while*/
S_B->E[i+1].no_of_ps=l;
++S_B->nonempty;
}
/*_____________________________________________________________________*/
void dec_B(int left,int right,int sub,float value)
```

```
{
  int la=1; /*the subscript the array starts with 1*/
  int lb=1;
  above_n_B=0;  /*reset*/
  below_n_B=0;  /*reset*/
  while(left<=right)
  {
    if(jmp_B[sub]->j>value)
    {
      above_B[la]=left;
      above_n_B=above_n_B+1;
      la=la+1;
    }
    else
    {
      below_B[lb]=left;
      below_n_B=below_n_B+1;
      lb=lb+1;
    }
    left=left+1;
    sub=sub+1;
  }
}
/*_____________________________________________________________________*/
void tran_en_B(int count,int n)
{
     struct tre_node *helper;
     /*copy entry to node 2*/
     strncpy(S_B2->E[n].w,"",30);
     strcpy(S_B2->E[n].w,S_B->E[count].w);
     S_B2->E[n].p_to_down=S_B->E[count].p_to_down;
     trace_B2[n]=S_B2->E[n].p_to_down;
     S_B2->E[n].no_of_ps=S_B->E[count].no_of_ps;
     S_B2->E[n].f=S_B->E[count].f;
     S_B2->flag=S_B->flag;
     ++S_B2->nonempty;
     /*remove copied entry from node 1*/
     strncpy(S_B->E[count].w,"",30);
     S_B->E[count].p_to_down=NULL;
     S_B->E[count].no_of_ps=0;
     S_B->E[count].f=0;
     --S_B->nonempty;
     if(S_B2->flag==0)
     {
       helper=S_B2->E[n].p_to_down;
       helper->from=n;
       helper->up=S_B2;
       helper=NULL;
     }
}
/*_____________________________________________________________________*/
void sp_nod_B()
{
  int i=0;
  int k=0;
  int o=0;
```

```
int differ=0;
S_B2=(struct tre_node *)malloc(sizeof(struct tre_node));/*points to node 2*/
S_B2->nonempty=0;
S_B2->join=S_B->join;
S_B->join=S_B2;
i=M-1; /*start with entry no. 1*/
while((S_B->E[i].w[0]=='*')&&(S_B->E[i].w[1]=='D'))
  i=i-1;
while(i>0)
{
 k=0;
 while(S_B->E[i].w[k]!='*')
 {
  o=i-1;
  while(o>=0)
  {
    if(S_B->E[i].w[k]!=S_B->E[o].w[k])
    {
      differ=i;
      break;
    }
    o=o-1;
  }
  if(differ==0)
     k=k+1;
  else
     break;
 }/*end of inner while*/
 if(differ!=0)
 {
   i=i-1;
   differ=0;
 }
 else
   break;
}/*end of outer while*/
while(i<M)
{
    tran_en_B(i+1,S_B2->nonempty);
    i=i+1;
}
}
/*_____________________________________________________________________*/
void union_B(struct tre_node *p)
{
  strncpy(buffer2,"",30);
  strcpy(buffer2,p->E[p->nonempty-1].w);
}
/*_____________________________________________________________________*/
void adjust_B()
{
  int i=0;
  struct tre_node *carry;
  parent_B=S_B;
  if(parent_B!=hop_B->tree)
    parent_B=S_B->up;
```

```
 while((parent_B!=S_B)&&(parent_B!=NULL)&&(S_B!=hop_B->tree))
 {
     i=S_B->from;
     strncpy(parent_B->E[i].w,"",30);
     strcpy(parent_B->E[i].w,S_B->E[(S_B->nonempty)-1].w);
     parent_B->E[i].f=S_B->E[(S_B->nonempty)-1].f;
     if(S_B2!=NULL)
     {
       S_B=parent_B;
       shift_B(i+1);
       S_B=parent_B->E[i].p_to_down;
       strncpy(parent_B->E[i+1].w,"",30);
       strcpy(parent_B->E[i+1].w,S_B2->E[(S_B2->nonempty)-1].w);
       parent_B->E[i+1].f=S_B2->E[(S_B2->nonempty)-1].f;
       ++(parent_B->nonempty);
       parent_B->E[i+1].p_to_down=S_B2;
       S_B2->up=parent_B;
       S_B2->from=i+1,
       S_B2=NULL;
     }
     S_B=parent_B;
     parent_B=S_B->up;
     if(S_B->nonempty==M+1)
       sp_nod_B();
 }
 if((S_B->nonempty==M+1)&&(S_B==hop_B->tree))
       sp_nod_B();

 i=0;
 if((S_B==hop_B->tree)&&(S_B2!=NULL)) /*the split node is the root*/
 {
   carry=(struct tre_node *)malloc(sizeof(struct tre_node));
   hop_B->tree=carry;
   carry=NULL;
   parent_B=hop_B->tree;
   S_B->up=parent_B;
   S_B2->up=parent_B;
   S_B->from=i;
   S_B2->from=i+1;
   hop_B->tree->E[i].p_to_down=S_B;
   hop_B->tree->E[i+1].p_to_down=S_B2;
   hop_B->tree->flag=0;
   hop_B->tree->nonempty=2;
   union_B(S_B);
   strncpy(hop_B->tree->E[i].w,"",30);
   strcpy(hop_B->tree->E[i].w,buffer2);
   strncpy(hop_B->tree->E[i+1].w,"",30);
   strcpy(hop_B->tree->E[i+1].w,"*");
   hop_B->tree->E[i].f=S_B->E[S_B->nonempty-1].f;
   hop_B->tree->E[i+1].f=S_B2->E[S_B2->nonempty-1].f;
 } /*end of if*/
}
/*_____________________________________________________________________*/
void sp_buc_B(int i,float a,float b,float c)
{
 char BE1[30]; /*of new 1st entry*/
```

```
char BE2[30], /*of new 2nd entry*/
char ent_exp[30]; /*of original entry*/
int dif=0;
int t=0;
int k=0;
int count=0;
int counter=0;
int ans;
int tt;
int total;
jmp_B[k]=S_B->E[i].p_to_down;
strncpy(BE1,"",30);
strncpy(BE2,"",30);
BE1[0]='*';
below_n_B=0;
above_n_B=0;
k=k+1;
counter=counter+1;
total=M+1;
while(k<=M+1)
{
   jmp_B[k]=jmp_B[k-1]->next;
   if((jmp_B[k]->j!=jmp_B[k-1]->j)&&(k>1))
       dif=1;
   k=k+1;
}
if((dif==0)&&((k-1)>M)) /*all j-coord. are equal*/
{
   strcpy(BE1,S_B->E[i].w);
   if(strstr(BE1,"D")==NULL)
      BE1[strlen(BE1)]='D';
   strcpy(BE2,S_B->E[i].w);
   while(counter<=(total/2))
   {
       below_B[counter]=counter;
       ++below_n_B;
       ++counter;
   }
   counter=1;
   while(counter<=total-below_n_B)
   {
       above_B[counter]=counter+below_n_B;
       ++above_n_B;
       ++counter;
   }
   points_B=below_n_B;
}/*end of if*/
else
{
 k=1;
 dec_B(k,M+1,k,(((J1-J2)/2)+J2));
 while(1)
 {
  if(above_n_B==total)
  {
   BE1[t]='1';
```

```
    count=1;
    J2=J2+((J1-J2)/2);
    dec_B(count,above_n_B,above_B[count],(((J1-J2)/2)+J2));
   }
   else if(below_n_B==total)
   {
    BE1[t]='0';
    count=1;
    J1=J2+((J1-J2)/2);
    dec_B(count,below_n_B,below_B[count],(((J1-J2)/2)+J2));
   }
   else
   {
    if(below_n_B>above_n_B)
        BE1[t]='0';
    else if(above_n_B>below_n_B)
        BE1[t]='1';
    else
    {
      if(t!=0)
        BE1[t]=BE1[t-1];
      else
        BE1[t]='0';
    }
    break;
   }/*end of else*/
   t=t+1;
  }/*end of while*/
  BE1[t+1]='*';
  strncpy(ent_exp,"",30);
  strcpy(ent_exp,S_B->E[i].w);
  if(strlen(BE1)>ln_GBE_B[t_level_B])
     ln_GBE_B[t_level_B]=strlen(BE1);
  if(BE1[t]=='1')
    points_B=above_n_B; /*points_B is no. of points in ist entry*/
  else
    points_B=below_n_B;
  strncpy(BE2,"",30);
  strcpy(BE2,ent_exp);
 }/*end of else*/
 phys_sp_B(i,BE1,BE2);
 if(S_B->nonempty==M+1)
 {
    sp_nod_B();
    adjust_B();/*adjust tree*/
 }
}
/*_____________________________________________________________________*/
int a_buck_B(int i,float a,float b,float c)
{
 int split=0;
 if(S_B->E[i].no_of_ps<M)
 {
  in_buck_B(i,a,b,c); /*to insert in bucket list*/
  return(0);
 }
```

```
 else
 {
  in_buck_B(i,a,b,c);
  sp_buc_B(i,a,b,c); /*split bucket*/
  return(1);
 }
}
/*_____________________________________________________________________*/
void shift_B(int i)
{
 int count=S_B->nonempty-1;
 while(count>=i)
 {
    strncpy(S_B->E[count+1].w,"",30);
    strcpy(S_B->E[count+1].w,S_B->E[count].w);
    S_B->E[count+1].p_to_down=S_B->E[count].p_to_down;
    S_B->E[count+1].no_of_ps=S_B->E[count].no_of_ps;
    S_B->E[count+1].f=S_B->E[count].f;
    if(S_B->flag==0)
    {
      S_B=S_B->E[count+1].p_to_down;
      S_B->from=count+1;
      S_B=S_B->up;
    }
    count=count-1;
 }/*end of while*/
 strncpy(S_B->E[i].w,"",30);
 S_B->E[i].p_to_down=NULL;
 S_B->E[i].no_of_ps=0;
 S_B->E[i].f=0;
}
/*_____________________________________________________________________*/
void insert_B(float a,float b)
{
 char BE_B[30]; /*BE of point in index B*/
 float c;
 int NOW_L;
 int split;
 int i2,          /*entry numbers returned by search node function*/
 if(B==NULL)
 {
      B=(struct v_root *)malloc(sizeof(struct v_root));
      B->link=NULL;
      hop_B=B;
 }
 f_level_B(a,b);  /*Find time level where point is*/
 if(hop_B->tree==NULL)
  create_B();
 c=(a+b)/1.414;
 NOW_L=(t_level_B+1)*GRAN;   /*Calculate NOW_L of the time level*/
 space_B(NOW_L);
 poi_BE_B(NOW_L,a,b);
 space_B(NOW_L);
 strncpy(BE_B,"",30);
 strcpy(BE_B,buffer2);
 S_B=hop_B->tree;
```

```
hold_B[t_level_B]=hop_B->tree;
while(1) /*to repeat in case of nonleaf node but with descending to child*/
{
 i2=s_n_in_B(BE_B,a,b);
 if(i2<=M)
 {
  if(S_B->flag==0)  /*in case of nonleaf node */
    S_B=S_B->E[i2].p_to_down; /*descend the tree*/
  else if((S_B->flag==1)&&(S_B->nonempty==0))
  {
    ins_en_B(i2);
    split=a_buck_B(i2,a,b,c);
    if(split==0)
    {
     ++(S_B->E[i2].no_of_ps);
     ++(S_B->nonempty);
    }
    break;
  }
  else if((S_B->flag==1)&&(i2>S_B->nonempty))
  {
    ins_en_B(i2);
    split=a_buck_B(i2,a,b,c);
    if(split==0)
    {
     ++(S_B->E[i2].no_of_ps);
     ++(S_B->nonempty);
    }
    break;
  }
  else if((S_B->flag==1)&&(i2<=S_B->nonempty)&&(S_B->nonempty<=M))
  {
    if((i2!=M-1)&&(S_B->E[i2].no_of_ps==M))
      shift_B(i2+1);
    split=a_buck_B(i2,a,b,c);
    if(split==0)
     ++(S_B->E[i2].no_of_ps);
    break;
  }
 }/*end of if i2<=M*/
} /*end of while*/
hold_B[t_level_B]=hop_B->tree;
}

/*_____________________________________________________________________*/
void print_B()
{
  int qty=0;
  struct tre_node *pin;
  struct tre_node *pin2;
  struct v_root *pin3;
  struct buc_node *pin4;
  int i=0;
  int turn=0;
  int round=0;
  FILE *stream2;
```

```
stream2=fopen("put2","w");
pin3=B;
if(pin3==NULL)
 fprintf(stream2,"\n Index B does not exist ");
else
{
 pin3=B->link;
 ++qty;
 /*printf("\n Index . B ");*/
 /*printf("\n ----------");*/
 while(pin3!=NULL)
 {
  /*printf("\ntime level : %d",pin3->level);*/
  pin=hold_B[pin3->level];
  pin2=pin;
  /*printf("\n              ");*/
  while(1) /*vertical move*/
  {
   /*printf("\n.....new tree level.....\n");*/
   while(pin!=NULL)/*horizontal move*/
   {
    ++qty;
    /*printf("\n--------new node------\n");*/
    i=0;
    if(pin->flag==1)
    {
       /*printf("field node\n");*/
       ++qty;
    }
    while(i<pin->nonempty)/*inside node*/
    {
      if((round==4)&&(pin->flag==0))
      {
       /*printf("\n");*/
       round=0;
      }
      /*printf("| %s |",pin->E[i].w);*/
      /*printf("%.1f",pin->E[i].f);*/
      ++round;
      if(pin->flag==1)
      {
       /* printf("\n");*/
        pin4=pin->E[i].p_to_down;
        pin4=pin4->next;
        while(pin4!=NULL)
        {
         if(turn==5)
         {
           /*printf("\n");*/
           turn=0;
         }
         /*printf("(%.1f",pin4->x);*/
         /*printf(",%.1f",pin4->y);*/
         /*printf(",%.1f)",pin4->j);*/
         pin4=pin4->next;
         ++turn;
```

```
          }
          /*printf("\n");*/
         }
         i=i+1;
         turn=0;
       }/*end of while*/
       pin=pin->join;
       if(pin->flag==0)
          ;/*printf("    ");*/
     }/*end of while*/
     pin=pin2;
     if(pin->flag==1)
       break;
     else
     {
       pin=pin2->E[0].p_to_down;
       pin2=pin;
     }
    }/*end of while*/
    fprintf(stream2,"\ntime level : %d",pin3->level);
    pin3=pin3->link;
    ++qty;
   }/*end of while*/
  }/*end of else*/
  fprintf(stream2,"\ntotal no of nodes= %d\n",qty);
}
/*_____________________________________________________________________*/
int search_B(float a,float b)
{
  int NOW_L;
  int nodes=0;
  char string[30];
  int e;
  struct buc_node *look;
  f_level_B(a,b);
  NOW_L=(t_level_B+1)*8;
  space_B(NOW_L);
  poi_BE_B(NOW_L,a,b);
  strncpy(string,"",30);
  strcpy(string,buffer2);
  S_B=hold_B[t_level_B];
  while(S_B->flag==0)
  {
      e=s_n_in_B(string,a,b);
      ++nodes;
      S_B=S_B->E[e].p_to_down;
  }/*end of while*/
  e=s_n_in_B(string,a,b);
  ++nodes;
  if(e<S_B->nonempty)
  {
     look=S_B->E[e].p_to_down;
     trace_B[e]=look;
     look=look->next; /*by pass the dummy head*/
     while(look!=NULL)
     {
```

```
    if((look->x==a)&&(look->y==b))
    {
      printf("\n  The point (%.1f,%.1f) is in index B",a,b);
      printf("\n  at time level=%d",t_level_B);
      printf("\n  number of nodes searched=%d",nodes);
      del_B=look;
      break;
    }
    else
    {
      trace_B[e]=look;
      look=look->next;
    }
   }/*end of while*/
  }
  else
    printf("\n  The point (%.1f,%.1f) is not in index B",a,b);
  return(e);
}
/*_____________________________________________________________________*/
void deshift_B(int n)
{
  while(n<=S_B->nonempty-1)
  {
   strncpy(S_B->E[n-1].w,"",30);
   strcpy(S_B->E[n-1].w,S_B->E[n].w);
   S_B->E[n-1].p_to_down=S_B->E[n].p_to_down;
   S_B->E[n-1].no_of_ps=S_B->E[n].no_of_ps;
   S_B->E[n-1].f=S_B->E[n].f;
   if((S_B->flag==0)&&(S_B->E[n-1].p_to_down!=NULL))
   {
     S_B=S_B->E[n-1].p_to_down;
     S_B->from=n-1;
     S_B=S_B->up;
   }
   n=n+1;
  }/*end of while*/
  strncpy(S_B->E[n-1].w,"",30);
  S_B->E[n-1].p_to_down=NULL;
  S_B->E[n-1].no_of_ps=0;
  S_B->E[n-1].f=0;
  --S_B->nonempty;
}
/*_____________________________________________________________________*/
void cond_B(int e)
{
 struct tre_node *help;
 if((S_B->E[e].no_of_ps!=0)&&(S_B->flag==1))
   return;
 else if((e==S_B->nonempty-1)&&(e!=0)&&(S_B->flag==1))
     /*the rightmost but not no. 0 */
 {
  strncpy(S_B->E[e-1].w,"",30);
  strcpy(S_B->E[e-1].w,S_B->E[e].w);
  strncpy(S_B->E[e].w,"",30);
  S_B->E[e].p_to_down=NULL;
```

```
  --S_B->nonempty;
  return;
 }
 else if((e<S_B->nonempty-1)&&(S_B->flag==1))
 {
  deshift_B(e+1);/*start shift left from no. e+1 */
  return;
 }
 else if((e==0)&&(e==S_B->nonempty-1)&&(S_B->flag==1)&&(S_B->E[e].w[0]!='*'))
 {
    e=S_B->from;
    if(e>0)
    {
       help=S_B;
       S_B=S_B->up;
       S_B=S_B->E[e-1].p_to_down;
       S_B->join=help->join;
       help=NULL;
       S_B=S_B->up;
       S_B->E[e].p_to_down=NULL;
       deshift_B(e+1);
    }/*end of if*/
    else /*e==0*/
        ; /*leave node as it is to preserve tree balance*/
 } /*end of else*/
}
/*_____________________________________________________________________*/
void delete_B(float a,float b)
{
  int e;
  del_B=NULL;
  e=search_B(a,b);
  if(del_B==NULL)
    printf("\n Point(%.1f,%.1f) was not found in index B",a,b);
  else
  {
    trace_B[e]->next=del_B->next;
    del_B=NULL;
    --S_B->E[e].no_of_ps;
    cond_B(e);/*e is the empty entry*/
    printf("\n Point(%.1f,%.1f) deleted from index B",a,b);
  }/*end of else*/
}
/*_____________________________________________________________________*/
int max_t_B(int t,int max)
{
  if(t>max)
     max=t;
  return(max);
}
/*_____________________________________________________________________*/
int min_t_B(int t,int min)
{
  if(t<min)
      min=t;
  return(min);
```

```
}
/*_____________________________________________________________________*/
void pol_ex_B(float c1,float c2)
{
  int NOW_L;
  float value;
  char bin_exp[30];
  int i=0;
  NOW_L=(t_level_B+1)*GRAN;
  space_B(NOW_L);
  value=J2+((J1-J2)/2);
  strncpy(bin_exp,"",30);
  while(strlen(bin_exp)<(ln_GBE_B[t_level_B]-1))
  {
   if(c1<=value)
   {
    bin_exp[i]='0';
    J1=value;
    value=J2+((J1-J2)/2);
   }
   else if(c2>=value)
   {
     bin_exp[i]='1';
     J2=value;
     value=J2+((J1-J2)/2);
   }
   else
   {
      break;
   }
   i=i+1;
  } /*end of while*/
  bin_exp[i]='*';
  strncpy(buffer2,"",30);
  strcpy(buffer2,bin_exp);
}
/*_____________________________________________________________________*/
int pref_B(char *sh,char *lo)
{
 if(strstr(lo,sh)==lo)
    return(1);
 else
    return(0);
}
/*_____________________________________________________________________*/
void find_B(struct tre_node *s,char *string,float c1,float c2)
{
 struct buc_node *p;
 static int turn;
 char shorter[30];
 char longer[30];
 static int number;
 int i=0;
 int boolean=0;
 while(i<s->nonempty)
 {
```

```
strncpy(shorter,"",30);
strncpy(longer,"",30);
if(s->flag==0)
      number=0;
else if(s->flag==1)
      ++number;
if(strlen(s->E[i].w)<=strlen(string))
{
     strncpy(shorter,s->E[i].w,strlen(s->E[i].w)-1);
     strncpy(longer,string,strlen(string)-1);
}
else
{
     strncpy(longer,s->E[i].w,strlen(s->E[i].w)-1);
     strncpy(shorter,string,strlen(string)-1);
}
boolean=pref_B(shorter,longer);
if((boolean==1)&&(number==0))
{
 if(nodes2_B>0)
       ++nodes2_B;
    else
       ++nodes1_B;
  if((((struct tre_node *)(s->E[i].p_to_down))->flag==0)&&(s->flag!=1))
  {
     if(nleaf2_B>0)
       ++nleaf2_B;
     else
       ++nleaf1_B;
  }

  find_B(s->E[i].p_to_down,string,c1,c2);
 }
 else if((boolean==1)&&(number>0))
 {
    p=s->E[i].p_to_down;
    p=p->next;
    while(p!=NULL)
    {
      if(turn==5)
      {
        /* printf("\n");*/
        turn=0;
      }
      if((p->j<=c1)&&(p->j>=c2))
      {
       /*printf("(%.1f",p->x);*/
       /*printf(",%.1f",p->y);*/
       /*printf(",%.1f)",p->j);*/
       ++turn;
      }
      p=p->next;
    }
  }
  i=i+1;
} /*end of while*/
```

```
}
/*_____________________________________________________________________*/
void pol_B(float a1,float b1,float a2,float b2,float a3,float b3,float a4,float b4)
{
 int NOW_L;
 int n=0;
 int max=0;
 int min=0;
 int value=0;
 int count=0;
 int first;
 char string[30];
 int i=0;
 float c1,c2;
 FILE *out2;
 char *array[30];
 out2=fopen("put2","w");
 c1=(a1+b1)/1.414;
 c2=(a2+b2)/1.414;
 fprintf(out2,"\n----------search for polygon----------\n");
 fprintf(out2,"The polygon that is between the e parallel lines j=%.1f,j=%.1f has :",c1,c2);
 f_level_B(a1,b1);
 max=max_t_B(t_level_B,max);
 min=min_t_B(t_level_B,min);
 f_level_B(a2,b2);
 max=max_t_B(t_level_B,max);
 min=min_t_B(t_level_B,min);
 f_level_B(a3,b3);
 max=max_t_B(t_level_B,max);
 min=min_t_B(t_level_B,min);
 if(a4!=-0.0)
 {
   f_level_B(a4,b4);
   max=max_t_B(t_level_B,max);
   min=min_t_B(t_level_B,min);
 }
 t_level_B=min;
 while(t_level_B<=max)
 {
  NOW_L=(t_level_B+1)*GRAN;
  space_B(NOW_L);
  value=J2+((J1-J2)/2);
  if((value<c1)&&(value>c2))
  {
     pol_ex_B(c1,value);
     array[count]=calloc(30,sizeof(char));
     strncpy(array[count],"",30);
     strcpy(array[count],buffer2);
     pol_ex_B(value,c2);
     strcat(array[count],"+");
     strcat(array[count],buffer2);
     ++t_level_B;
     ++count;
  }
  else
  {
```

```
    pol_ex_B(c1,c2);
    array[count]=calloc(30,sizeof(char));
    strncpy(array[count],"",30);
    strcpy(array[count],buffer2);
    ++t_level_B;
    ++count;
  }
} /*end of while*/
t_level_B=min;
count=0;
while(t_level_B<=max)
{
 fprintf(out2,"\nIn time level %d - - - -\n",t_level_B);
 S_B=hold_B[t_level_B];
 nodes1_B=0;
 nodes2_B=0;
 nleaf1_B=0;
 nleaf2_B=0;
 if(S_B!=NULL)
 {
   if(strstr(array[count],"+")==NULL)
   {
    ++nodes1_B;
    if(S_B->flag==0)
        ++nleaf1_B;
    find_B(S_B,array[count],c1,c2);
   }
   else
   {
    first=0;
    strncpy(string,"",30);
    strcpy(string,strstr(array[count],"+")+1);
    ++nodes1_B;
    if(S_B->flag==0)
         ++nleaf1_B;
    find_B(S_B,string,c1,c2);
    first=strcspn(array[count],"+");
    strncpy(string,"",30);
    strncpy(string,array[count],first);
    ++nodes2_B;
    if(S_B->flag==0)
        ++nleaf2_B;
    find_B(S_B,string,c1,c2);
   }
   }
   if(nodes1_B>nodes2_B)
         fprintf(out2,"\nthe total number of searched nodes=%d\n",nodes1_B);
   else
         fprintf(out2,"\nthe total number of searched nodes=%d\n",nodes2_B);
   if(nleaf1_B>nleaf2_B)
         fprintf(out2,"\nthe number of nonleaf nodes=%d\n",nleaf1_B);
   else
         fprintf(out2,"\nthe number of nonleaf nodes=%d\n",nleaf2_B);

   ++t_level_B;
   ++count;
```

```
 }/*end of while*/
}
/*_____________________________________________________________________*/
main(int argc,char **argv)
{
FILE *in;
int uu2;
char ch;
float u1,v1,u2,v2,u3,v3,u4,v4;
char file_name[10];
float xx,yy;
/*printf("enter file name : ");*/
/*scanf("%s",file_name);*/
/*printf("\n");*/
in=fopen("large2","r");
if (!in)
{
 printf("error open input file\n");
}
while(1)
{
 fscanf(in,"%c",&ch);
 if(ch=='+')
 {
  fscanf(in,"(%f,%f)",&xx,&yy);
  insert_B(xx,yy);
 }
 else if(ch=='#')
 {
  fscanf(in,"(%f,%f)",&xx,&yy);
  uu2=search_B(xx,yy);
 }
 else if(ch=='$')
 {
   print_B();
 }
 else if(ch=='-')
 {
   fscanf(in,"(%f,%f)",&xx,&yy);
   delete_B(xx,yy);
 }
 else if(ch=='?')
 {
   fscanf(in,"(%f,%f,%f,%f,%f,%f,%f,%f)",&u1,&v1,&u2,&v2,&u3,&v3,&u4,&v4);
   pol_B(u1,v1,u2,v2,u3,v3,u4,v4);
 }
 if (feof(in))
   break;
}
free(B);
free(S_B);
free(S_B2);
free(temp);
return(0);
}
/*_____________________________________________________________________*/
```

```
/*************************************************************************/
/*                          C-tree type "C"                              */
/*                                                                       */
/* This program includes the code for insertion, search, and deletion in */
/* C-trees type "C". This program is called from the "Parallelizer"      */
/*************************************************************************/


#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <parallel/microtask.h>
#include <parallel/parallel.h>

#define M 62 /*Max number of entries in a node*/
#define GRAN 365 /*space advances every 8 time units*/

struct entry        {
                    char w[30];     /*entry binary expressions*/
                    void *p_to_down; /*may point to tre_node or buc_node*/
                    int no_of_ps;    /*is no. of points in bucket of leaf*/
                                     /*is zero for nonleaf*/
          float f;
                    };

struct tre_node     {
                    struct entry E[M+1]; /*array of structures*/
                    int flag;        /*is 0 for nonleaf,1 for leaf*/
          int nonempty;      /*no. of nonempty slots in the node*/
          int from;          /*no. of entry in parent*/
          struct tre_node *up;/*pointer to parent*/
          struct tre_node *join;/*pointer to right node*/
                    };

struct buc_node     {
                    float x;
                    float y;
                    float j;
                    struct buc_node *p_t_same; /*pointer to bucket node*/
                                               /*where same point is*/
          struct buc_node *next;
                    };

struct v_root {
          int level;
          struct tre_node *tree;
          struct v_root *link;
          };

struct v_root *C;
struct v_root *hop_C;
struct v_root *dummy_C;
struct tre_node *S_C;
struct tre_node *S_C2;
```

```
struct tre_node *hold_C[150];
struct tre_node *parent_C;
struct buc_node *p_buck_C;
struct buc_node *temp;
struct buc_node *jmp_C[M+3];
struct buc_node *mark_C;
struct buc_node *trace_C[M+1];
struct buc_node *trace_C2[M+1];
struct buc_node *del_C;

int NOW;      /*normalized most rescent current time*/
int nodes_no1;
int nodes_no2;
int nonleafs1;
int nonleafs2;
int t_level_C;
int start=0;   /*normalized start time of database*/
int ln_GBE_C[150]; /*length of greatest GBA*/
float X1,Y1,X2,Y2,X3,Y3,X4,Y4; /*space corner points*/
int above_C[M+3];
int below_C[M+3];
int above_n_C=0;
int below_n_C=0;
int points_C=0;

char GBE_C[30];
char buffer3[30];
char buffer[30];

void insert_C(float,float);
void create_C();
void ins_en_C(int);
void f_level_C(float,float);     /*find time level of point*/
void poi_BE_C(int,float,float);
int s_n_in_C(char *,float,float);
int a_buck_C(int,float,float,float);
void in_buck_C(int,float,float,float);
void sp_buc_C(int,float,float,float);
void space_C(int);
void dec_C(int,int,int,float);
void phy_sp_C(int,char *,char *);
void sp_nod_C();
void tran_en_C(int,int);
void shift_C(int);
void adjust_C();
void union_C(struct tre_node *);
void print_C();
int search_C(float,float);
void delete_C(float,float);
void cond_C(int);
void deshift_C(int);
void pol_C(float,float,float,float,float,float,float,float);
void pol_ex_C(float,float);
int max_t_C(int,int);
int min_t_C(int,int);
void find_C(struct tre_node *,char *,float,float);
```

```
int pref_C(char *,char *);

/*_____________________________________________________________________*/
void in_buck_C(int i,float a,float b,float c)
{
  temp=(struct buc_node *)malloc(sizeof(struct buc_node));
  temp->next=NULL;
  if(S_C->nonempty==0)
  {
     p_buck_C=(struct buc_node *)malloc(sizeof(struct buc_node));
     S_C->E[i].p_to_down=p_buck_C;
     p_buck_C->next=NULL;
  }
  p_buck_C=S_C->E[i].p_to_down;
  trace_C[i]=S_C->E[i].p_to_down;
  while(p_buck_C->next!=NULL)
  {
    if(b<=p_buck_C->next->y)
    {
          temp->next=p_buck_C->next;
          p_buck_C->next=temp;
          p_buck_C->next->x=a;
          p_buck_C->next->y=b;
          p_buck_C->next->j=c;
          mark_C=p_buck_C;
          temp=NULL;
          if(b>S_C->E[i].f)
            S_C->E[i].f=b;
          return;
  }
    else if(b>p_buck_C->next->y)
    {
          p_buck_C=p_buck_C->next;
    }
  } /*end of while loop*/
  temp->next=p_buck_C->next;
  p_buck_C->next=temp;
  p_buck_C->next->x=a;
  p_buck_C->next->y=b;
  p_buck_C->next->j=c;
  mark_C=p_buck_C;
  temp=NULL;
  if(b>S_C->E[i].f)
            S_C->E[i].f=b;
   return;
}

/*_____________________________________________________________________*/
void ins_en_C(int i)
{
 strncpy(S_C->E[i].w,"",30);
 strcpy(S_C->E[i].w,buffer3);
 if(strlen(S_C->E[i].w)>ln_GBE_C[t_level_C])
   ln_GBE_C[t_level_C]=strlen(S_C->E[i].w);
}
/*_____________________________________________________________________*/
```

```
void create_C()
{
 int i;
 GBE_C[0]='*';
 ln_GBE_C[t_level_C]=strlen(GBE_C);
 hop_C->tree=(struct tre_node *)malloc(sizeof(struct tre_node));
 hop_C->tree->flag=1;
 S_C=hop_C->tree;
 i=0;
 while(i<=M)
 {
  strncpy(hop_C->tree->E[i].w,"",30);
  hop_C->tree->E[i].no_of_ps=0;
  hop_C->tree->E[i].p_to_down=NULL;
  hop_C->tree->E[i].f=0;
  ++i;
 }
 hop_C->tree->up=NULL;
 hop_C->tree->join=NULL;
 hop_C->tree->nonempty=0;
}
/*_____________________________________________________________________*/
void f_level_C(float p,float q)  /*This function is independent on index used*/
{
 int number=-1;
 int NOW_L;
 t_level_C=(p+q)/GRAN;
 hop_C=C;
 while((hop_C->link!=NULL)&&(hop_C->link->level<=t_level_C))
 {
  hop_C=hop_C->link;
  if(hop_C->level==t_level_C)
  {
    number=t_level_C;
    break;
  }
 }/*end of while*/
 if(number==-1)
 {
   dummy_C=(struct v_root *)malloc(sizeof(struct v_root));
   dummy_C->link=NULL;
   dummy_C->tree=NULL;
   dummy_C->link=hop_C->link;
   hop_C->link=dummy_C;
   hop_C=dummy_C;
   dummy_C=NULL;
   hop_C->level=t_level_C;
 }/*end of if*/
}

/*_____________________________________________________________________*/
void space_C(int NOW_L)
{
 if(t_level_C==0)
 {
  X1=0;
```

```
 Y1=NOW_L;
 X2=0;
 Y2=0;
 X3=NOW_L;
 Y3=0;    /*anticlockwise order from top*/
}
else
{
 X1=0;
 Y1=NOW_L;
 X2=0;
 Y2=NOW_L-GRAN;
 X3=NOW_L-GRAN;
 Y3=0;
 X4=NOW_L;
 Y4=0;
}
}

/*_____________________________________________________________________*/
void poi_BE_C(int NOW_L,float a, float b)
{
char bin_exp[30];
float value;
int i=0;
int X_X; /*is the higher point*/
strncpy(bin_exp,"",30);
space_C(NOW_L);
if(t_level_C==0)
 X_X=X3;
else
 X_X=X4,
value=X1+(X_X-X1)/2;
while(strlen(bin_exp)<(ln_GBE_C[t_level_C]-1))
{
 if(a<=value)
 {
  bin_exp[i]='0';
  X_X=value;
  value=X1+(X_X-X1)/2,
 }
 else
 {
  bin_exp[i]='1';
  X1=value;
  value=X1+((X_X-X1)/2);
 }
 i=i+1;
} /*end of while loop*/
bin_exp[i]='*';
strncpy(buffer3,"",30);
strcpy(buffer3,bin_exp);
}
/*_____________________________________________________________________*/
int s_n_in_C(char BE_C[],float a,float b)
{
```

```
int differ=0;
int l=0;
int i=0;
int k=0;
char *D_CAP=NULL;
if(S_C->nonempty==0)
 return(i); /*the first entry i.e no. zero*/
else
{
   while(i<S_C->nonempty)  /*to search nonempty entries of a node*/
   {
    if((S_C->E[i].w[0]=='*')&&((strstr(S_C->E[i].w,"D")==NULL)))
          return(i);
    l=0;
    while(S_C->E[i].w[l]!='*')
    {
      if(S_C->E[i].w[l]!=BE_C[l])
           {
              differ=1;
              break;
           }
           l=l+1;
    }/*end of while*/
    D_CAP=NULL;
    D_CAP=strstr(S_C->E[i].w,"D");
    if(differ==0) /*the entry is qualified*/
    {
          if(D_CAP==NULL) /*no D character in BE*/
                    return(i);
          else if((D_CAP!=NULL)&&(b<=S_C->E[i].f))
                    /*point lies inside D character range f*/
                    return(i);
          else
                    differ=1;
    }/*end of if differ..*/
    i=i+1; /*to search next entry*/
    differ=0;
   }/*end of outer while*/
   if((differ==1)&&(S_C->flag==1)&&(S_C->nonempty<M))
          return(i+1);
   /*else if((differ==1)&&(S_C->flag==1)&&(S_C->nonempty==M))*/
          /*return(M+1); */
  }/*end of else*/
}
/*_____________________________________________________________________________*/
void phys_sp_C(int i,char *BE1,char *BE2)
{
    int l=0;
    int v;
    int u;
    int array[M+3];
    array[0]=0; /*not used*/
    if(points_C==below_n_C)
    {
     S_C->E[i].f=jmp_C[below_C[points_C]]->y;
     S_C->E[i+1].f=jmp_C[above_C[M+1-points_C]]->y;
```

```
}
else
{
 S_C->E[i].f=jmp_C[above_C[points_C]]->y;
 S_C->E[i+1].f=jmp_C[below_C[M+1-points_C]]->y;
}
l=l+1;
strncpy(S_C->E[i].w,"",30);
strcpy(S_C->E[i].w,BE1);
S_C->E[i].no_of_ps=points_C;
strncpy(buffer3,"",30);
strcpy(buffer3,BE2);
ins_en_C(i+1);
p_buck_C=(struct buc_node *)malloc(sizeof(struct buc_node));
S_C->E[i+1].p_to_down=p_buck_C;
p_buck_C=S_C->E[i+1].p_to_down;
trace_C[i+1]=S_C->E[i+1].p_to_down;
if(BE1[strlen(BE1)-2]=='1')
{
 while(l<=(M+1-points_C))
 {
        array[l]=below_C[l];
        l=l+1;
 }
}
else
{
 while(l<=(M+1-points_C))
 {
        array[l]=above_C[l];
        l=l+1;
 }
}
l=l-1;
v=1;
u=1;
while(v<=l)
{
 p_buck_C->next=jmp_C[array[v]];
 while(1)
 {
       if(jmp_C[array[v]-u]!=NULL)
       {
         jmp_C[array[v]-u]->next=jmp_C[array[v]]->next;
         p_buck_C=p_buck_C->next;
         p_buck_C->next=NULL;
         jmp_C[array[v]]=NULL;
         break;
       }
       else
         u=u+1;
 }/* end of while(1)*/
 v=v+1;
 u=1;
} /*end of while*/
S_C->E[i+1].no_of_ps=l;
```

```
    ++S_C->nonempty;
}
/*_____________________________________________________________________*/
void dec_C(int left,int right,int sub,float value)
{
  int la=1; /*the subscript the array starts with 1*/
  int lb=1;
  above_n_C=0;  /*reset*/
  below_n_C=0;  /*reset*/
  while(left<=right)
  {
    if(jmp_C[sub]->x>value)
    {
          above_C[la]=left;
          above_n_C=above_n_C+1;
          la=la+1;
    }
    else
    {
          below_C[lb]=left;
          below_n_C=below_n_C+1;
          lb=lb+1;
    }
    left=left+1;
    sub=sub+1;
  }
}
/*_____________________________________________________________________*/
void tran_en_C(int count,int n)
{
     struct tre_node *helper;
     /*copy entry to node 2*/
     strncpy(S_C2->E[n].w,"",30);
     strcpy(S_C2->E[n].w,S_C->E[count].w);
     S_C2->E[n].p_to_down=S_C->E[count].p_to_down;
     trace_C2[n]=S_C2->E[n].p_to_down;
     S_C2->E[n].no_of_ps=S_C->E[count].no_of_ps;
     S_C2->E[n].f=S_C->E[count].f;
     S_C2->flag=S_C->flag;
     ++S_C2->nonempty;
     /*remove copied entry from node 1*/
     strncpy(S_C->E[count].w,"",30);
     S_C->E[count].p_to_down=NULL;
     S_C->E[count].no_of_ps=0;
     S_C->E[count].f=0;
     --S_C->nonempty;
     if(S_C2->flag==0)
     {
          helper=S_C2->E[n].p_to_down;
          helper->from=n,
          helper->up=S_C2;
          helper=NULL;
     }
}
/*_____________________________________________________________________*/
void sp_nod_C()
```

```
{
  int i=0;
  int k=0;
  int o=0;
  int differ=0;
  S_C2=(struct tre_node *)malloc(sizeof(struct tre_node));
  S_C2->nonempty=0;
  S_C2->join=S_C->join;
  S_C->join=S_C2;
  i=M-1;
  while((S_C->E[i].w[0]=='*')&&(S_C->E[i].w[1]=='D'))
     i=i-1;
  while(i>0)
  {
   k=0;
   while(S_C->E[i].w[k]!='*')
   {
    o=i-1;
    while(o>=0)
    {
           if(S_C->E[i].w[k]!=S_C->E[o].w[k])
           {
              differ=i;
              break;
           }
           o=o-1;
    }
    if(differ==0)
            k=k+1;
    else
            break;
   }/*end of inner while*/
   if(differ!=0)
   {
          i=i-1;
          differ=0;
   }
   else
          break;
  }/*end of outer while*/
  while(i<M)
  {
          tran_en_C(i+1,S_C2->nonempty);
          i=i+1;
  }
}
/*_____________________________________________________________________*/
void union_C(struct tre_node *p)
{
  strncpy(buffer3,"",30);
  strcpy(buffer3,p->E[p->nonempty-1].w);
}
/*_____________________________________________________________________*/
void adjust_C()
{
  int i=0;
```

```
struct tre_node *inter;
parent_C=S_C;
if(parent_C!=hop_C->tree)
  parent_C=S_C->up;
while((parent_C!=S_C)&&(parent_C!=NULL)&&(S_C!=hop_C->tree))
{
        i=S_C->from;
        strncpy(parent_C->E[i].w,"",30);
        strcpy(parent_C->E[i].w,S_C->E[(S_C->nonempty)-1].w);
        parent_C->E[i].f=S_C->E[(S_C->nonempty)-1].f;
        if(S_C2!=NULL)
        {
          S_C=parent_C;
          shift_C(i+1);
          S_C=parent_C->E[i].p_to_down;
          strncpy(parent_C->E[i+1].w,"",30);
          strcpy(parent_C->E[i+1].w,S_C2->E[(S_C2->nonempty)-1].w);
          parent_C->E[i+1].f=S_C2->E[(S_C2->nonempty)-1].f;
          ++(parent_C->nonempty);
          parent_C->E[i+1].p_to_down=S_C2;
          S_C2->up=parent_C;
          S_C2->from=i+1;
          S_C2=NULL;
        }
        S_C=parent_C;
        parent_C=S_C->up;
        if(S_C->nonempty==M+1)
          sp_nod_C();
}
if((S_C->nonempty==M+1)&&(S_C==hop_C->tree))
          sp_nod_C();
i=0;
if((S_C==hop_C->tree)&&(S_C2!=NULL))  /*the split node is the root*/
{
  inter=(struct tre_node *)malloc(sizeof(struct tre_node));
  hop_C->tree=inter;
  inter=NULL;
  parent_C=hop_C->tree;
  S_C->up=parent_C;
  S_C2->up=parent_C;
  S_C->from=i;
  S_C2->from=i+1;
  hop_C->tree->E[i].p_to_down=S_C;
  hop_C->tree->E[i+1].p_to_down=S_C2;
  hop_C->tree->flag=0;
  hop_C->tree->nonempty=2;
  union_C(S_C);
  strncpy(hop_C->tree->E[i].w,"",30);
  strcpy(hop_C->tree->E[i].w,buffer3);
  strncpy(hop_C->tree->E[i+1].w,"",30);
  strcpy(hop_C->tree->E[i+1].w,"*");
  hop_C->tree->E[i].f=S_C->E[S_C->nonempty-1].f;
  hop_C->tree->E[i+1].f=S_C2->E[S_C2->nonempty-1].f;
 } /*end of if*/
}
/*_____________________________________________________________________*/
```

```
void sp_buc_C(int i,float a,float b,float c)
{
 char BE1[30]; /*of new 1st entry*/
 char BE2[30]; /*of new 2nd entry*/
 char ent_exp[30]; /*of original entry*/
 int dif=0;
 int t=0;
 int k=0;
 int count;
 int counter=0;
 int ans;
 int tt;
 float X_X;
 int total;
 below_n_C=0;
 above_n_C=0;
 if(t_level_C==0)
   X_X=X3;
 else
   X_X=X4;
 jmp_C[k]=S_C->E[i].p_to_down;
 k=k+1;
 counter=counter+1;
 total=M+1;
 while(k<=M+1)
 {
    jmp_C[k]=jmp_C[k-1]->next;
    if((jmp_C[k]->x!=jmp_C[k-1]->x)&&(k>1))
           dif=1;
    k=k+1;
 }
 if((dif==0)&&((k-1)>M))
 {
   strncpy(BE1,"",30);
   strcpy(BE1,S_C->E[i].w);
   if(strstr(BE1,"D")==NULL)
          BE1[strlen(BE1)]='D';
   strncpy(BE2,"",30);
   strcpy(BE2,S_C->E[i].w);
   while(counter<=(total/2))
   {
    below_C[counter]=counter;
    ++below_n_C;
    ++counter;
   }
   counter=1;
   while(counter<=total-below_n_C)
   {
        above_C[counter]=counter+below_n_C;
        ++above_n_C;
        ++counter;
   }
   points_C=below_n_C;
 }
 else
 {
```

```
 strncpy(BE1,"",30);
 k=1;
 dec_C(k,M+1,k,(((X_X-X1)/2)+X1));
 while(1)
 {
  if(above_n_C==total)
  {
        BE1[t]='1';
        count=1;
        X1=X1+((X_X-X1)/2);
        dec_C(count,above_n_C,above_C[count],(((X_X-X1)/2)+X1));
  }
  else if(below_n_C==total)
  {
        BE1[t]='0';
        count=1;
        X_X=X1+((X_X-X1)/2);
        dec_C(count,below_n_C,below_C[count],(((X_X-X1)/2)+X1));
  }
  else
  {
        if(below_n_C>above_n_C)
            BE1[t]='0';
        else if(above_n_C>below_n_C)
            BE1[t]='1';
        else
        {
          if(t!=0)
            BE1[t]=BE1[t-1];
          else
            BE1[t]='0';
        }
        break;
  }/*end of else*/
  t=t+1;
 }/*end of while*/
 BE1[t+1]='*';
 strncpy(ent_exp,"",30);
 strcpy(ent_exp,S_C->E[i].w);
 if(strlen(BE1)>ln_GBE_C[t_level_C])
   ln_GBE_C[t_level_C]=strlen(BE1);
 strncpy(BE2,"",30);
 strcpy(BE2,ent_exp);
 if(BE1[t]=='1')
   points_C=above_n_C; /*points_C is no of points in 1st entry*/
 else
   points_C=below_n_C;
}
phys_sp_C(i,BE1,BE2);
if(S_C->nonempty==M+1)
{
   sp_nod_C();
   adjust_C();/*adjust tree*/
}

}
```

```
/*_____________________________________________________________________*/
int a_buck_C(int i,float a,float b,float c)
{
 int split=0;
 if(S_C->E[i].no_of_ps<M)
 {
  in_buck_C(i,a,b,c); /*to insert in bucket list*/
  return(0);
 }
 else
 {
  in_buck_C(i,a,b,c);
  sp_buc_C(i,a,b,c); /*split bucket*/
  return(1);
 }
}
/*_____________________________________________________________________*/
void shift_C(int i)
{
 int count=S_C->nonempty-1;
 while(count>=i)
 {
    strncpy(S_C->E[count+1].w,"",30);
    strcpy(S_C->E[count+1].w,S_C->E[count].w);
    S_C->E[count+1].p_to_down=S_C->E[count].p_to_down;
    S_C->E[count+1].no_of_ps=S_C->E[count].no_of_ps;
    S_C->E[count+1].f=S_C->E[count].f;
    if(S_C->flag==0)
    {
          S_C=S_C->E[count+1].p_to_down;
          S_C->from=count+1;
          S_C=S_C->up;
    }
    count=count-1;
 }/*end of while*/
 strncpy(S_C->E[i].w,"",30);
 S_C->E[i].p_to_down=NULL;
 S_C->E[i].no_of_ps=0;
 S_C->E[i].f=0;
}
/*_____________________________________________________________________*/
void insert_C(float a,float b)
{
char BE_C[30]; /*BE of point in index C*/
float c;
int NOW_L;
int split;
int i3;          /*entry numbers returned by search node function*/
if(C==NULL)
{
          C=(struct v_root *)malloc(sizeof(struct v_root));
          C->link=NULL;
          hop_C=C;
}
f_level_C(a,b);  /*Find time level where point is*/
if(hop_C->tree==NULL)
```

```
 create_C();
c=(a+b)/1.414;
NOW_L=(t_level_C+1)*GRAN;   /*Calculate NOW_L of the time level*/
space_C(NOW_L);
poi_BE_C(NOW_L,a,b);
space_C(NOW_L);
strncpy(BE_C,"",30);
strcpy(BE_C,buffer3);
S_C=hop_C->tree;
hold_C[t_level_C]=hop_C->tree;
while(1) /*to repeat in case of nonleaf node but with descending to child*/
{
i3=s_n_in_C(BE_C,a,b);
if(i3<=M)
{
 if(S_C->flag==0)  /*in case of nonleaf node */
   S_C=S_C->E[i3].p_to_down; /*descend the tree*/
 else if((S_C->flag==1)&&(S_C->nonempty==0))
 {
   ins_en_C(i3);
   split=a_buck_C(i3,a,b,c);
   if(split==0)
   {
        ++(S_C->E[i3].no_of_ps);
        ++(S_C->nonempty);
   }
   break;
 }
 else if((S_C->flag==1)&&(i3>S_C->nonempty))
 {
   ins_en_C(i3);
   split=a_buck_C(i3,a,b,c);
   if(split==0)
   {
        ++(S_C->E[i3].no_of_ps);
        ++(S_C->nonempty);
   }
   break;
 }
 else if((S_C->flag==1)&&(i3<=S_C->nonempty)&&(S_C->nonempty<=M))
 {
   if((i3!=M-1)&&(S_C->E[i3].no_of_ps==M))
        shift_C(i3+1);
   split=a_buck_C(i3,a,b,c);
   if(split==0)
        ++(S_C->E[i3].no_of_ps);
   break;
 }
}/*end of if i3<=M*/
} /*end of while*/
hold_C[t_level_C]=hop_C->tree;
}
/*_____________________________________________________________________*/
void print_C()
{
 int qty=0;
```

```
struct tre_node *pin;
struct tre_node *pin2;
struct v_root *pin3;
struct buc_node *pin4;
int i=0;
int turn=0;
int round=0;
FILE *stream3;
stream3=fopen("put3","w");
pin3=C;
if(pin3==NULL)
 fprintf(stream3,"\n Index C does not exist ");
else
{
 pin3=C->link;
 ++qty;
 /*printf("\n Index : C ");*/
 /*printf("\n ----------");*/
 while(pin3!=NULL) /*levels*/
 {
 /*printf("\ntime level : %d",pin3->level);*/
 pin=hold_C[pin3->level];
 pin2=pin;
 /*printf("\n              ");*/
 while(1) /*vertical move*/
 {
  /*printf("\n.....new tree level.....");*/
  while(pin!=NULL)/*horizontal move*/
  {
   ++qty;
        /*printf("\n----------new node-------\n");*/
        i=0;
        if(pin->flag==1)
   {
            /*printf("field node\n");*/
       ++qty;
   }
        while(i<pin->nonempty)/*inside node*/
        {
          if((round==4)&&(pin->flag==0))
          {
            /*printf("\n");*/
            round=0;
          }
          /*printf("| %s |",pin->E[i].w);*/
          /*printf("%.1f",pin->E[i].f);*/
          ++round;
          if(pin->flag==1)
          {
           /*printf("\n");*/
           pin4=pin->E[i].p_to_down;
           pin4=pin4->next;
           while(pin4!=NULL)
           {
                 if(turn==5)
                 {
```

```
                    /*printf("\n");*/
                    turn=0;
                  }
                  /*printf("(%.1f",pin4->x);*/
                  /*printf(",%.1f)",pin4->y);*/
                  pin4=pin4->next;
                  ++turn;
                }
                /*printf("\n");*/
              }
              i=i+1;
              turn=0;
            }/*end of while inside node*/
            pin=pin->join;
            if(pin->flag==0)
              ;/*printf("\n");*/
      }/*end of while horizontal move*/
      pin=pin2;
      if(pin->flag==1)
            break;
      else
      {
            pin=pin2->E[0].p_to_down;
            pin2=pin;
      }
     }/*end of while vertical move*/
     fprintf(stream3,"\ntime level : %d",pin3->level);
     pin3=pin3->link;
     ++qty;
    }/*end of while levels*/
   }/*end of else*/
   fprintf(stream3,"\ntotal no. of nodes= %d\n",qty);
}
/*_____________________________________________________________________________*/
int search_C(float a,float b)
{
  int NOW_L;
  int nodes=0;
  char string[30];
  int e;
  struct buc_node *look;
  f_level_C(a,b);
  NOW_L=(t_level_C+1)*8;
  space_C(NOW_L);
  poi_BE_C(NOW_L,a,b);
  strncpy(string,"",30);
  strcpy(string,buffer3);
  S_C=hold_C[t_level_C];
  while(S_C->flag==0)
  {
        e=s_n_in_C(string,a,b);
        ++nodes;
        S_C=S_C->E[e].p_to_down;
  }/*end of while*/
  e=s_n_in_C(string,a,b);
  ++nodes;
```

```
  if(e<S_C->nonempty)
  {
     look=S_C->E[e].p_to_down;
     trace_C[e]=look;
     look=look->next; /*by pass the dummy head*/
     while(look!=NULL)
     {
          if((look->x==a)&&(look->y==b))
          {
            printf("\n  The point (%.1f,%.1f) is in index C",a,b);
            printf("\n  at time level=%d",t_level_C);
            printf("\n  number of nodes searched=%d",nodes);
            del_C=look;
            break;
          }
          else
          {
            trace_C[e]=look;
            look=look->next;
          }
     }/*end of while*/
  }
  else
     printf("\n  The point (%.1f,%.1f) is not in index C",a,b);
  return(e);
}
/*_____________________________________________________________________*/
void deshift_C(int n)
{
   while(n<=S_C->nonempty-1)
   {
    strncpy(S_C->E[n-1].w,"",30);
    strcpy(S_C->E[n-1].w,S_C->E[n].w);
    S_C->E[n-1].p_to_down=S_C->E[n].p_to_down;
    S_C->E[n-1].no_of_ps=S_C->E[n].no_of_ps;
    S_C->E[n-1].f=S_C->E[n].f;
    if((S_C->flag==0)&&(S_C->E[n-1].p_to_down!=NULL))
    {
          S_C=S_C->E[n-1].p_to_down;
          S_C->from=n-1;
          S_C=S_C->up;
    }
    n=n+1;
   }/*end of while*/
   strncpy(S_C->E[n-1].w,"",30);
   S_C->E[n-1].p_to_down=NULL,
   S_C->E[n-1].no_of_ps=0;
   S_C->E[n-1].f=0;
   --S_C->nonempty;
}
/*_____________________________________________________________________*/
void cond_C(int e)
{
 struct tre_node *help;
 if((S_C->E[e].no_of_ps!=0)&&(S_C->flag==1))
    return;
```

```
else if((e==S_C->nonempty-1)&&(e!=0)&&(S_C->flag==1))
        /*the rightmost but not no. 0 */
{
 strncpy(S_C->E[e-1].w,"",30);
 strcpy(S_C->E[e-1].w,S_C->E[e].w);
 strncpy(S_C->E[e].w,"",30);
 S_C->E[e].p_to_down=NULL;
 --S_C->nonempty;
 return;
}
else if((e<S_C->nonempty-1)&&(S_C->flag==1))
{
 deshift_C(e+1);/*start shift left from no. e+1 */
 return;
}
else if((e==0)&&(e==S_C->nonempty-1)&&(S_C->flag==1)&&(S_C->E[e].w[0]!='*'))
{
   e=S_C->from;
   if(e>0)
   {
        help=S_C;
        S_C=S_C->up;
        S_C=S_C->E[e-1].p_to_down;
        S_C->join=help->join;
        help=NULL;
        S_C=S_C->up;
        S_C->E[e].p_to_down=NULL;
        deshift_C(e+1);
   }/*end of if*/
   else /*e==0*/
        ; /*leave node as it is to preserve tree balance*/
 } /*end of else*/
}
/*_____________________________________________________________________*/
void delete_C(float a,float b)
{
 int e;
 del_C=NULL;
 e=search_C(a,b);
 if(del_C==NULL)
  printf("\n Point(%.1f,%.1f) was not found in index C",a,b);
 else
 {
  trace_C[e]->next=del_C->next;
  del_C=NULL;
  --S_C->E[e].no_of_ps;
  cond_C(e);/*e is the empty entry*/
  printf("\n Point(%.1f,%.1f) deleted from index C",a,b);
 }/*end of else*/
}
/*_____________________________________________________________________*/
int max_t_C(int t,int max)
{
 if(t>max)
  max=t;
 return(max);
```

```
}
/*_____________________________________________________________________*/
int min_t_C(int t,int min)
{
 if(t<min)
   min=t;
 return(min);
}
/*_____________________________________________________________________*/
void pol_ex_C(float v1,float v2)
{
 int NOW_L;
 float value;
 float X_X;
 char bin_exp[30];
 int i=0;
 NOW_L=(t_level_C+1)*GRAN;
 space_C(NOW_L);
 if(t_level_C==0)
  X_X=X3;
 else
  X_X=X4,
 value=X1+(X_X-X1)/2;
 strncpy(bin_exp,"",30);
 while(strlen(bin_exp)<(ln_GBE_C[t_level_C]-1))
 {
  if(v1<=value)
  {
   bin_exp[i]='0';
   X_X=value;
   value=X1+(X_X-X1)/2;
  }
  else if(v2>=value)
  {
   bin_exp[i]='1';
   X1=value;
   value=X1+((X_X-X1)/2);
  }
  else
          break;
  i=i+1;
 } /*end of while loop*/
 bin_exp[i]='*';
 strncpy(buffer3,"",30);
 strcpy(buffer3,bin_exp);
}
/*_____________________________________________________________________*/
int pref_C(char *sh,char *lo)
{
 if(strstr(lo,sh)==lo)
   return(1);
 else
   return(0);
}
/*_____________________________________________________________________*/
void find_C(struct tre_node *s,char *string,float a3,float a2)
```

```
{
  struct buc_node *p;
  static int turn;
  char shorter[30];
  char longer[30];
  static int number;
  int i=0;
  int boolean=0;
  while(i<s->nonempty)
  {
    strncpy(shorter,"",30);
    strncpy(longer,"",30);
    if(s->flag==0)
    {
          number=0;
    }
    else if(s->flag==1)
          ++number;
    if(strlen(s->E[i].w)<=strlen(string))
    {
              strncpy(shorter,s->E[i].w,strlen(s->E[i].w)-1);
              strncpy(longer,string,strlen(string)-1);
    }
    else
    {
              strncpy(longer,s->E[i].w,strlen(s->E[i].w)-1);
              strncpy(shorter,string,strlen(string)-1);
    }
    boolean=pref_C(shorter,longer);
    if((boolean==1)&&(number==0))
    {
     if(nodes_no2>0)
          ++nodes_no2;
     else
          ++nodes_no1;
     if((((struct tre_node *)(s->E[i].p_to_down))->flag==0)&&(s->flag!=1))
     {
        if(nonleafs2>0)
          ++nonleafs2;
        else
          ++nonleafs1;
     }
     find_C(s->E[i].p_to_down,string,a3,a2);
    }
    else if((boolean==1)&&(number>0))
    {
              p=s->E[i].p_to_down;
              p=p->next;
              while(p!=NULL)
              {
                    if(turn==5)
                    {
                      /*printf("\n");*/
                      turn=0;
                    }
                    if((p->x<=a3)&&(p->x>=a2))
```

```
                    {
                      /*printf("(%.1f",p->x);*/
                      /*printf(",%.1f)",p->y);*/
                      ++turn;
                    }
            p=p->next;
           }
     }
     i=i+1;
  } /*end of while*/
}
/*_____________________________________________________________________*/
void pol_C(float a1,float b1,float a2,float b2,float a3,float b3,float a4,float b4)
{
 int NOW_L;
 float X_X;
 int n=0;
 int max=0;
 int min=0;
 int value=0;
 int count=0;
 int first;
 char string[30];
 int i=0;
 char *array[30];
 FILE *out3;
 out3=fopen("put3","w");
 fprintf(out3,"\n----------search for polygon----------\n");
 fprintf(out3,"The polygon that is between the 2 vertical line x=%.1f,x=%.1f has "
,a3,a2);
 f_level_C(a1,b1);
 max=max_t_C(t_level_C,max);
 min=min_t_C(t_level_C,min);
 f_level_C(a2,b2);
 max=max_t_C(t_level_C,max);
 min=min_t_C(t_level_C,min);
 f_level_C(a3,b3);
 max=max_t_C(t_level_C,max);
 min=min_t_C(t_level_C,min);
 if(a4!=-0.0)
 {
  f_level_C(a4,b4);
  max=max_t_C(t_level_C,max);
  min=min_t_C(t_level_C,min);
 }
 t_level_C=min;
 while(t_level_C<=max)
 {
  NOW_L=(t_level_C+1)*GRAN;
  space_C(NOW_L);
  if(t_level_C==0)
    X_X=X3;
  else
    X_X=X4;
  value=X1+(X_X-X1)/2;
  if((value<a3)&&(value>a2))
```

```
{
 pol_ex_C(a3,value);
 array[count]=calloc(30,sizeof(char));
 strncpy(array[count],"",30);
 strcpy(array[count],buffer3);
 pol_ex_C(value,a2);
 strcat(array[count],"+");
 strcat(array[count],buffer3);
 ++t_level_C;
 ++count;
}
else
{
 pol_ex_C(a3,a2);
 array[count]=calloc(30,sizeof(char));
 strncpy(array[count],"",30);
 strcpy(array[count],buffer3);
 ++t_level_C;
 ++count;
}
}/*end of while*/
t_level_C=min;
count=0;
while(t_level_C<=max)
{
  fprintf(out3,"\nIn time level %d - - - -\n",t_level_C);
  S_C=hold_C[t_level_C];
  nodes_no1=0;
  nodes_no2=0;
  nonleafs1=0;
  nonleafs2=0;
  if(S_C!=NULL)
  {
   if(strstr(array[count],"+")==NULL)
   {
    ++nodes_no1;
    if(S_C->flag==0)
      ++nonleafs1;
    find_C(S_C,array[count],a3,a2);
   }
   else
   {
    first=0;
    strncpy(string,"",30);
    strcpy(string,strstr(array[count],"+")+1);
    ++nodes_no1;
    if(S_C->flag==0)
      ++nonleafs1;
    find_C(S_C,string,a3,a2);
    first=strcspn(array[count],"+");
    strncpy(string,"",30);
    strncpy(string,array[count],first);
    ++nodes_no2;
    if(S_C->flag==0)
      ++nonleafs2,
    find_C(S_C,string,a3,a2);
```

```
    }
    }
    if(nodes_no1>nodes_no2)
        fprintf(out3,"\nthe total number of searched nodes=%d\n",nodes_no1);
    else
        fprintf(out3,"\nthe total number of searched nodes=%d\n",nodes_no2);
    if(nonleafs1>nonleafs2)
        fprintf(out3,"\nthe number of nonleaf nodes=%d\n",nonleafs1);
    else
        fprintf(out3,"\nthe number of nonleaf nodes=%d\n",nonleafs2);
    ++t_level_C;
    ++count;
 }/*end of while*/
}
/*_____________________________________________________________________*/
main(int argc,char **argv)
{
 FILE *in;
 int uu3;
 char ch;
 char file_name[10];
 float xx,yy;
 float u1,v1,u2,v2,u3,v3,u4,v4,
 /*printf("enter file name : ");*/
 /*scanf("%s",file_name);*/
 /*printf("\n");*/
 in=fopen("large3","r");
 if(!in)
 {
  printf("error open input file\n");
 }
 while(1)
 {
  fscanf(in,"%c",&ch);
  if(ch=='+')
  {
   fscanf(in,"(%f,%f)",&xx,&yy);
   /*printf("%.1f,%.1f\n",xx,yy);*/
   insert_C(xx,yy);
  }
  else if(ch=='#')
  {
   fscanf(in,"(%f,%f)",&xx,&yy);
   uu3=search_C(xx,yy);
  }
  else if(ch=='$')
  {
    print_C();
  }
  else if(ch=='-')
  {
    fscanf(in,"(%f,%f)",&xx,&yy);
    delete_C(xx,yy);
  }
  else if(ch=='?')
  {
```

```
    }
    }
    if(nodes_no1>nodes_no2)
        fprintf(out3,"\nthe total number of searched nodes=%d\n",nodes_no1);
    else
        fprintf(out3,"\nthe total number of searched nodes=%d\n",nodes_no2);
    if(nonleafs1>nonleafs2)
        fprintf(out3,"\nthe number of nonleaf nodes=%d\n",nonleafs1);
    else
        fprintf(out3,"\nthe number of nonleaf nodes=%d\n",nonleafs2);
    ++t_level_C;
    ++count;
 }/*end of while*/
}
/*_____________________________________________________________________*/
main(int argc,char **argv)
{
 FILE *in;
 int uu3;
 char ch;
 char file_name[10];
 float xx,yy;
 float u1,v1,u2,v2,u3,v3,u4,v4,
 /*printf("enter file name : ");*/
 /*scanf("%s",file_name);*/
 /*printf("\n");*/
 in=fopen("large3","r");
 if(!in)
 {
  printf("error open input file\n");
 }
 while(1)
 {
  fscanf(in,"%c",&ch);
  if(ch=='+')
  {
   fscanf(in,"(%f,%f)",&xx,&yy);
   /*printf("%.1f,%.1f\n",xx,yy);*/
   insert_C(xx,yy);
  }
  else if(ch=='#')
  {
   fscanf(in,"(%f,%f)",&xx,&yy);
   uu3=search_C(xx,yy);
  }
  else if(ch=='$')
  {
    print_C();
  }
  else if(ch=='-')
  {
    fscanf(in,"(%f,%f)",&xx,&yy);
    delete_C(xx,yy);
  }
  else if(ch=='?')
  {
```

```
   fscanf(in,"(%f,%f,%f,%f,%f,%f,%f,%f)",&u1,&v1,&u2,&v2,&u3,&v3,&u4,&v4);
   pol_C(u1,v1,u2,v2,u3,v3,u4,v4);
 }
 if (feof(in))
   break;
}
free(C);
free(S_C);
free(S_C2);
free(temp);
return(0);
}
/*_____________________________________________________________________*/
```

VITA

Hani A. Hafez

Candidate for the Degree of

Master of Science

Thesis: A DYNAMIC AND EFFICIENT PARALLEL INDEX STRUCTURE FOR TEMPORAL DATABASES

Major Field: Computer Science

Biographical:

Personal Data: Born in Alexandria, Egypt, On July 7, 1957, the son of Ahmed Hafez and Nawal Saleh.

Education: Graduated from Raml High School, Alexandria, Egypt in June 1974; received Bachelor of Science degree in Electrical Engineering from Alexandria University, Alexandria, Egypt in June 1979.
Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in July 1996.

Experience: System Analyst, Dayemtex Co., Cairo, Egypt, 1985 to 1992; Graduate Assistant, Oklahoma State University, Department of Counseling Services, July 1994 to May 1996.