Data structures and VLSI

Bink, J.M.

Award date:
1991

EB321
5864

Eindhoven University of Technology
Department of Electrical Engeneering
Digital Systems Group (EB)

DATA STRUCTURES and VLSI

by J.M. Bink

Master Thesis Report
Supervision: Prof. Ir. M.P.J. Stevens.
Ir. L.P.M. Benders.

Eindhoven, August 1991

Summary

For the description of problems, many high level description languages are known. Most of these languages are software languages. An easy way to describe problems is to use abstract data types, but then the translation to hardware is difficult. If we could find a hardware implementation for often used data types, it would become much easier to translate the problem description to hardware implementation.

I have searched in literature for data structures, which are often used for the description of problems. These data structures are described just like abstract data types. Research is done to software structures, used for the implementation of these given data structures. I have also studied algorithms which are used with the data structures (e.g. a search algorithm for trees). If we implement algorithms along with the data structures, faster implementations can be found.

Then, two hardware memories (RAM and CAM) are described, which can be used for the implementation of the data structures in hardware. The CAM can besides storing information, also compare applied information with stored information. For some structures, like the table, the set and some graphs, the CAM gives a faster implementation than the RAM. There are also many CAM algorithms known for searching, sorting etc. The CAM however, is much more complex (size CAM is 3 to 4 times the size of the RAM).

Finally I have started with the hardware description of the data structures. This is done with the hardware description language VHDL. Because of the complexity and variety of the subject, I have not yet finished this.

For each data structure, several implementations are possible. Which implementation we should use depends on the use of the structure. Adding algorithms to the data structures improves the performance.

Table of contents

1. Introduction

At the Digital Systems Group, Department of Electrical Engineering, Eindhoven University of Technology, research is done to the traject of developing digital systems. This is the traject of high level problem description to hardware implementation. If possible, this traject has to be automated. One of the topics is the high level description language, used for describing problems.

There are many ways to describe a problem, and there are many ways to implement the solution. An easy way to describe problems is to use software data structures. The main disadvantage of this solution is that these data structures are usually not known in hardware implementations. So if we could find a hardware implementation of often used data structures, it would become much easier to translate the problem description to hardware implementation.

My task was, under supervision of prof.ir. M.P.J. Stevens and ir. L.P.M. Benders, to inventorise the most commonly used data structures, and then I should try to find a good hardware implementation.

First, I have studied the most commonly used data structures, and described these like abstract data types.

Secondly I have studied software structures, used for the implementation of these data structures. Using these software implementations, we can construct abstract data types, which can be used for the description of problems. I have also looked for each data structure for algorithms, which are often used with that data structure.

I have studied some hardware memories, like a RAM and a CAM, which can be used for the implementation of the data structures in hardware.

Finally I have started with the hardware description of the data structures. Because of the complexity and variety of the subject, I haven't finished this. There is for every data structure a behavioral description. A structural description still have to be made. This should be done by someone else.

2. Data Structures

When we want to solute a problem, it is useful to have some predefined data structures, which makes it easy to describe the problem. This chapter is an overview of the most fundamental data structures, with a set of operations, used for the description of problems.

We study the elementary data structures array, record and linked list. We consider lists, which are sequences of elements, and two special cases of lists: stacks, where elements are inserted and deleted at one end only, and queues, where elements are inserted at on end and deleted at the other. Then we study tables, which is often used for the conversion of types. We study trees, which is a collection of elements with a hierarchical structure. We also study sets, which is closely related to the mathematical notion of a set. Finally we study graphs, which are often used for problems arising in computer science, mathematics, engineering, etcetera. We can use graphs to represent arbitrary relationships among data objects. We consider the basic data types (boolean, char, integer and real) known.

2.1. Arrays

An array is a collection of objects, all of the same type - called *elementtype* - and indexed by a *linear ordered* set of values - called the *index set*. The members of the index set are all of the same type, called the *index type*. There is a one-to-one correspondence between the value of the index type and the object of each array element. An array is often denoted as

A: **array**[i_1, . . ,i_n]**of** elementtype;.

An array is a linear store, which has a finite number (n) of places with indexes *i*, with in each array element *A*[*i*] an object of the type elementtype. Elementtype can be of any type. The following operations are defined:

RETRIEVE(*A*,*i*,*c*): This operation reads element *i* of array *A* and assigns it to value *c*. This is often denoted as *c* := *A*[*i*]. This operation is not defined if $i \le i_1$ or if $i \ge i_n$.

UPDATE(*A*,*i*,*c*): This operation assigns the value *c* to element *i* of array *A*. This often denoted as *A*[*i*] := *c*. This operation is not defined if $i \le i_1$ or if $i \ge i_n$.

2.2. Records

A record is a collection of objects, which may be of a different type, with for each object an identifier. There is a one-to-one correspondence between the record identifiers and the record elements. A record is often denote as

```
R = record
      id_1:   type_1;
      . . .   . . .
      . . .   . . .
      id_n:   type_n;
    end;.
```

Each record has a finite number of identifiers id_n, with each identifier connected to an object of any type. We can define the following operations on the data structure record.

RETRIEVE(*R*,*id*,*v*): This operation assigns to *v* the value of the component of *R*, which is identified by *id*. This is often denoted as *v* := *R.id*. This operation is not defined if the type of *v* is not equal to the type of *R.id*, or if *id* is not in *R*.

UPDATE(*R*,*id*,*v*): This operation assigns the value *v* to component *id* of record *R*. This is often denoted as *R.id* := *v*. This operation is not defined if the type of *v* is not equal to the type of *R.id*, or if *id* is not in *R*.

2.3. Linked lists

In computer science, there are many applications in which the number of objects is changing dynamically. If we use for these applications static data structures, such as arrays or records, it would lead to inefficient use of memory space, and bad operations. So, it is useful to have a dynamic data structure.

Linked lists are the simplest form of dynamic data structures. A linked list is a list of pairs, each consisting of an element and a pointer, such that each pointer contains the address of the next pair. A pointer is simply a variable that holds as its value an address of another element. Each such pair is represented by a record. The following operations are defined:

CLEAR(*LL*): This operation makes the linked list *LL* to be an empty list.

INSERT(*LL*,*p*,*x*): This operation creates a new record, fills it with data *x*, and links it on position *p* in linked list *LL*. If position *p* does not exist, this operation is not defined.

DELETE(*LL*,*p*): This operation relinks linked list *LL* in such a way, that the record on position *p* in the list is no longer a member of list *LL*. If position *p* does not exist, this operation is not defined.

RETRIEVE(*LL*,*p*,*x*): Gives variable *x* the value of the data stored on position *p* of linked list *LL*. If position *p* does not exist, this operation is not defined.

NEXT(*LL*,*p*): This function returns the boolean true if there exist an element $p + 1$ of linked list *LL*, otherwise it returns false.

In figure 2.1. we see that the operations INSERT and DELETE are just a relinking of the pointers.

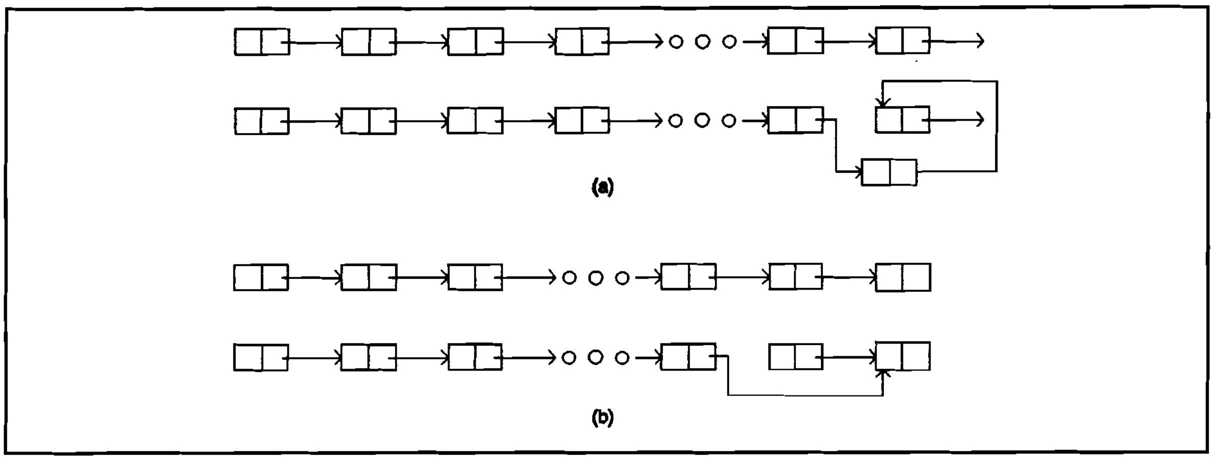


Figure 2.1. *(a)Insert and (b) delete operation on linked list.*

2.4. Lists

Mathematically, a list is a sequence of zero or more elements of a given type (which we generally call the elementtype). We often represent such a list by a comma-separated sequence of elements

$$a_1,a_2, \ldots ,a_n$$

where $n \geq 0$, and each a_i is of type elementtype. The number n of elements is said to be the length of the list. Assuming $n \geq 1$, we say that a_1 is the first element and a_n is the last element. If $n = 0$, we have an empty list, one that has no elements. An important property of a list is that its elements have an ordering according to their position on the list. We say a_i precedes a_{i+1} for $i = 1,2, \ldots ,n\text{-}1$, and a_i follows $a_{i\text{-}1}$ for $i = 2,3, \ldots ,n$. We say that the element a_i is at position i.

There is another ordering, which is often used. That is an ordering on the 'magnitude' of the elements. Then, $|a_i| \leq |a_{i+1}|$. A list ordered by position we call 'unordered' and a list ordered by magnitude we call 'ordered'. On the data structure 'unordered' list we can define a set of operations. This is not a unique set of operations, but we try to give a complete set.

CLEAR(*L*):	This function causes *L* to become an empty list.
END(*L*,*p*):	The function END(*L*) will assign to *p* the position following position *n* in a *n*-element list *L*. Note that the position END(*L*) has a distance from the beginning of the list that varies as the list grows or shrinks, while other positions have a fixed distance from the beginning of the list.
INSERT(*L*,*p*,*x*):	Insert *x* at position *p*, moving elements at *p* and following positions to the next higher position. That is, if *L* is $a_1,a_2, \ldots ,a_n$ then *L* becomes $a_1,a_2, \ldots,a_{p-1},x,a_p, \ldots ,a_n$. If *p* is END(*L*), then *L* becomes $a_1,a_2, \ldots ,a_n,x$. If list *L* has no position *p*, the result is undefined.
LOCATE(*L*,*p*,*x*):	This functions returns the position of *x* on list *L*. If *x* appears more than once, then the position of the first occurrence is returned. If *x* does not appear at all, then END(*L*) is returned.
RETRIEVE(*L*,*p*,*x*):	This function returns to *x* the element at position *p* on list *L*. The result is undefined if *p* = END(*L*) or if *L* has no position *p*.
DELETE(*L*,*p*):	Delete the element at position *p* of list *L*. If *L* is $a_1,a_2, \ldots ,a_n$, then *L* becomes $a_1,a_2, \ldots ,a_{p-1},a_{p+1}, \ldots ,a_n$. The result is undefined if *L* has no position *p* or if *p* = END(*L*).
FIRST(*L*,*p*):	This function returns to *p* the first position on list *L*. If *L* is empty, the position returned is END(*L*).
EMPTY(*L*):	Return true if list *L* is empty, return false otherwise.
FULL(*L*):	Return true if list *L* is full, return false otherwise.

With this set of operations it is possible to describe all possible manipulations on 'unordered' lists. If we want to define an 'ordered' list, we have to replace INSERT(*L*,*p*,*x*) by INSERT(*L*,*x*). Element *x* is inserted 'in order' and not on position *p*.

2.4.1. Stacks

A stack is a special list in which all insertions and deletions take place at one end, called the *top*. Other names for a stack are "pushdown list", and "LIFO" or "last-in-first-out" list. The intuitive model of a stack is a pile of poker chips on a table, books on a floor, or dishes on a shelf, where it is only convenient to remove the top object on the pile or add a new one above the top. We can define here a set of basic operations.

CLEAR(*S*):	This function causes *S* to become an empty stack. This operation is the same as for general lists.
TOP(*S*,*x*):	Return to *x* the element at the top of stack *S*. If, as is normal, we identify the top of a stack with position 1, then TOP(*S*,*x*) can be written in terms of list operations as RETRIEVE(*S*,FIRST(*S*),*x*).
POP(*S*,*x*):	Delete the top element of the stack, that is, DELETE(*S*,FIRST(*S*)). Sometimes it is convenient to implement POP as a function that returns the element it has just popped. Then, we don't need the function TOP(*S*).

PUSH(*S*,*x*): Insert the element *x* at the top of stack *S*. The old top element becomes next-to-top, and so on. In terms of list operations this operation is INSERT(*S*,FIRST(*S*),*x*).
EMPTY(*S*): Return true if *S* is an empty stack; return false otherwise.
FULL(*S*): Return true if *S* is a full stack; return false otherwise.

2.4.2. Queues

A queue is an other special list, where items are inserted at one end (the rear) and deleted at the other end (the front). Another name for a queue is a "FIFO" or "first-in-first-out" list. The operations for a queue are similar to those for a stack, the substantial differences being that insertions go at the end of the list, rather than the beginning, and that the traditional terminology for stacks and queues is different. We shall use the following basic operations on queues.

CLEAR(*Q*): Makes queue *Q* an empty list.
FRONT(*Q*,*x*): This is a function that returns the first element on queue *Q*. FRONT(*Q*,*x*) can be written in terms of list operations as RETRIEVE(*Q*,FIRST(*Q*),*x*).
ENQUEUE(*Q*,*x*): This function inserts element *x* at the end of queue *Q*. In terms of list operations, ENQUEUE(*Q*,*x*) is INSERT(*Q*,END(*Q*),*x*).
DEQUEUE(*Q*,*x*): This function deletes the first element of *Q*; that is, DEQUEUE(*Q*,*x*) is DELETE(*Q*,FIRST(*Q*)). It is also possible to combine FRONT and DEQUEUE in a single operation. Then, we don't need the operation FRONT.
EMPTY(*Q*): Returns true if and only if *Q* is an empty queue.
FULL(*Q*): Returns true if and only if *Q* is a full queue.

2.5. Tables

A table is a collection of *keyed* records, with for every record a *unique* key. When we want to retrieve a record, we look in the table for the key. If the key is found, that record is retrieved. Sometimes there is used a translation mechanism on the key for computing the address of the record. The following operations are defined:

CLEAR(*T*): Makes table *T* to be an empty table.
INSERT(*T*,*id*,*x*): Inserts data *x* with identifier *id* in table *T*. If *id* already exist in table *T*, the data connected with *id* is replaced by *x*.
DELETE(*T*,*id*): Deletes the record with key *id* from table *T*. If *id* is not a MEMBER of table *T*, nothing happens.
RETRIEVE(*T*,*id*,*x*): Returns the data to *x* from the record with key *id*, from table *T*. If *id* is not a MEMBER of table *T*, nothing happens.
MEMBER(*T*,*id*): Returns TRUE if the record with key *id* is in hash table *T*.
FULL(*T*): Returns true if table *T* is full; returns false otherwise.
EMPTY(*T*): Returns true if table *T* is empty; returns false otherwise.

2.6. Trees

A tree imposes a hierarchical structure on a collection of items. A tree is a collection of elements called *nodes*, one of which is distinguished as a *root*, along with a relation ("parenthood") that places a hierarchical structure on the nodes. A node, like an element of a list, can be of whatever type we wish. We often depict a node as a character, a string or a number with a circle around it. Formally a *tree* can be defined recursively in the following manner.

1. A single node by itself is a tree. This node is also the root of the tree.

2. Suppose n is a node and $T_1,T_2, \ldots ,T_k$ are trees with roots $n_1,n_2, \ldots ,n_k$, respectively. We can construct a new tree by making n be the parent of nodes $n_1,n_2, \ldots ,n_k$. In this tree n is the root and $T_1,T_2, \ldots ,T_k$ are *sub-trees* of the root. Nodes $n_1,n_2, \ldots ,n_k$ are called the *children* of node n.

Sometimes it is convenient to include among trees the *null tree*, a "tree" with no nodes, which we shall represent by •. It is possible to define an order on the nodes of a tree. Then, all the children of a node are assigned a different order. This depends on the implementation of a tree. In a graphic representation of a tree, the children of a node are usually ordered from left to right (see figure 2.2.).

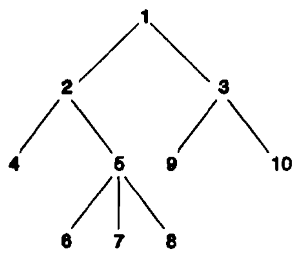


Figure 2.2. *Graphic representation of a tree.*

The *ancestors* of node n are defined as following: If node n is the root of a tree, then it has no ancestors. Else, the parent of n and all the ancestors of the parent of n are ancestors of n. The *descendants* of node n are: If node n is a leaf of a tree, it has no descendants. Else, the children of node n and all the descendants of the children of n are descendants of n.

We can associate a *label*, or value, with each node of a tree, in the same way as we associated a value with a list element. That is, the label of a node is not the name of the node, but an object that is "stored" at the node. Now we shall present several useful operations on trees. As with lists, there are a great variety of operations that can be performed on trees. Here, we shall consider the following operations:

CLEAR(*T*): Makes *T* the null tree.

PARENT(*T*,*n*,*x*): This function returns to *x* the parent of node *n* in tree *T*. If *n* is the root, which has no parent, ● is returned. In this context, ● is a "null node", which is used as a sign that we have navigated off the tree.

LEFT_CHILD(*T*,*n*,*x*): Returns to *x* the child of node *n* in tree *T* with the lowest order, and returns ● if *n* is a leaf, which has no children.

RIGHT_SIBLING(*T*,*n*,*x*): Returns to *x* the right sibling of node *n* in tree *T*, defined to be that node *m* with the same parent *p* as *n* such that *m* has the lowest ordering, following *n* in the ordering of the children of *p*.

LABEL(*T*,*n*,*l*): This function returns to *l* the label of node *n* in tree *T*. We do not, however, require labels to be defined for every tree.

CREATE($T,i,T_1,T_2,\ldots,T_i,l$): This is an infinite family of functions, one for each value of $i = 0,1,2,\ldots$. CREATE makes a new node *r* with label *l* and give it *i* children, which are the roots of trees $T_1,T_2,\ldots,T_i$, in order from the left. The tree *T* with root *r* is returned. Note that if $i = 0$, then *r* is both a leaf and the root.

ROOT(*T*,*x*): This function returns to *x* the node that is the root of tree *T*, or ● if *T* is the null tree.

There are special trees, such as binary trees, AVL trees, B-trees and tries, which often have a special set of operations. See also [AHO83], [COR90], [MAN89] and [WAR90]. These operations are usually combinations of operations described above. However, it can be useful to define for each data structure its own operation set.

2.7. Sets

As we said before, the set is the basic structure underlying all mathematics. A set is a collection of *members* (or *elements*); each member of a set either is itself a set or is a primitive element called an *atom*. All members of a set are different, which means no set can contain two copies of the same subset. We can denote a set of atoms by putting curly brackets around its members. A set with only the members 1, 3 and 4 is denoted as:

$\{1,3,4\}$ or $\{1,4,3\}$ or $\{4,3,1\}$ etc.

Sometimes, atoms are linearly ordered by a relation, usually denoted by "<". In that case, a *linear order* < on a set *S* satisfies two properties:

1. For any *a* and *b* in *S*, exactly one of $a < b$, or $b < a$ is true.
2. For all *a*, *b*, and *c* in *S*, if $a < b$ and $b < c$, then $a < c$ (transitivity).

We can give a set of operations on the data structure *set*, but this is not a unique set.

UNION(*A*,*B*,*C*): This operation takes the set-valued arguments *A* and *B*, and assign the result $A \cup B$ to the set variable *C*.

INTERSECTION(*A*,*B*,*C*): This operation takes the set-valued arguments *A* and *B*, and assign the result $A \cap B$ to the set variable *C*.

DIFFERENCE(*A*,*B*,*C*): This operation takes the set-valued arguments *A* and *B*, and assign the result $A - B$ to the set variable *C*.

CLEAR(*A*): Makes the null set to be the value for set variable *A*.

MEMBER(*A*,*x*): Takes set *A* and object *x*, whose type is the type of elements of *A*, and returns a boolean value — true if $x \in A$ and false if $x \notin A$.

INSERT(*A*,*x*): Makes element *x* a member of set *A*. That is, the new value of *A* is $A \cup \{x\}$. Note that if *x* is already a member of *A*, INSERT(*x*,*A*) does not change *A*.

DELETE(*A*,*x*): Removes element *x* from set *A*, i.e., *A* is replaced by $A - \{x\}$. If *x* is not in *A* originally, DELETE(*x*,*A*) does not change *A*.

ASSIGN(*A*,*B*): Sets the set value of set variable *A* to be equal to the value of set variable *B*.

EQUAL(*A*,*B*): This function returns the value true if and only if sets *A* and *B* consist of the same elements.

MIN(*A*): This function returns the smallest element in set *A*. This operation may only be applied when the members of the parameter set are linearly ordered.

MAX(*A*): This function returns the greatest element in set *A*. This operation may only be applied when the members of the parameter set are linearly ordered.

2.8. Graphs

Directed and undirected graphs are models for representing arbitrary relationships among data objects. A *directed graph* (*digraph* for short) *G* consists of a set of nodes *N* and a set of edges *E*. The nodes are also called *vertices* or *points*; the directed edges could be called *arcs* or *directed lines*. A directed edge is an ordered pair of nodes (*n*,*m*); *n* is called the *tail* and *m* the *head* of the edge. The directed edge (*n*,*m*) is expressed by $n \to m$ (figure 2.3).

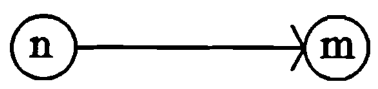


Figure 2.3. *A directed graph.*

The nodes of graph *G* are used to represent objects, and the arcs or directed edges are used to represent relationships between objects. Common operations on this data structure are:

INSERT_NODE(*G*,*n*,*l*): Makes a new node *n* with label *l* in graph *G*. This node is not connected to graph *G*.

INSERT_EDGE(*G*,*n*,*m*,*l*): Creates in graph *G* a directed edge from node *n* to node *m* which has the label *l*. This operation is undefined if *n* or *m* are no nodes of graph *G*.

DELETE_NODE(*G*,*n*): Deletes node *n* from graph *G*. There are no edges between node *n* and the rest of graph *G*.

DELETE_EDGE(*G*,*n*,*m*): Deletes in graph *G* the directed edge between node *n* and node *m*. Both *n* and *m* are elements of the set nodes of graph G, else this operation has no effect.

RETRIEVE_EDGE(*G*,*n*,*m*,*l*): This function returns to *l* the label of the directed edge between node *n* and node *m* in graph *G*. If there is no edge between *n* and *m* or these nodes are not in G, this function returns the null label •.

RETRIEVE_NODE(*G*,*n*,*l*): This function returns to *l* the label of node *n* in graph *G*. If node *n* is not in G, then the null label • is returned.

CLEAR(*G*): Makes G to be a graph with no nodes and no edges (null graph).

We represented only the directed graphs, because the undirected graphs can be implemented as directed graphs. An undirected edge can be represented as two directed edges in the opposite direction. See figure 2.4.

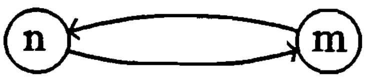


Figure 2.4. *Undirected graph.*

In this chapter we gave a definition of the data structure, and an overview of the (basic) operations on that data structure. It isn't always possible to define a 'best' set of operations for a data structure, but we tried to give a complete set of operations. It is possible that an other set of operations gives an easier or faster implementation. It is also known that a set of operations can be optimal for one problem, but not for another problem. For more information about data structures and algorithms see [AHO83], [COR90], [MAN89], [SED88], [STU85], [WAR90] and [WUL81]. In the next chapter, we will give some commonly used software implementation, and some algorithms used on these data structures.

3. Software implementations and their complexity

If we look at the most commonly used software languages, we see that only a few data structures are available (e.g. array, record, pointer). The other structures have to be implemented with these available structures. If all discussed data structures would be available for the programmer, it would be much easier to implement solution for a problem. For that reason, many languages support *abstract data types.* This are modules, which contain a data structure with its predefined operations. Then, if they use these modules, they have all data structures available.

This chapter gives an overview of the software implementation of the data structures. With every data structure we have given its characteristics and possible mappings to elementary structures (arrays, records and linked-lists). For every mapping, we discuss the complexity of the predefined operations, and we examine algorithms, which are often used with the data structure.

3.1. Arrays

An array is a row of elements of the same type. The size of the array is the number of elements in the array. This size must be fixed. The array is a storage type that is direct accessible, i.e. we can retrieve or update every single element of the array in a fixed time. In almost every software-language the array is a predefined data type. For the implementation of the array no extra memory space is necessary, but the size and type of the array should be defined before use. Because the array is a direct accessible storage type, the predefined operations 'RETRIEVE' and 'UPDATE' have a complexity O(c) (table 1).

Table 1: Array characteristics

ARRAY	
Operations	**Complexity**
UPDATE	O(c)
RETRIEVE	O(c)
Access	**Direct**
Size	**Fixed**

c = constant

Arrays cannot be used to store elements of different types (or sizes), and the size of an array cannot be changed dynamically.

3.2. Records

A record is a list of elements of different types. The exact combination of types has to be defined before use. The storage size of a record is known in advance. Records are a predefined type in many programming languages. Each element of a record can be accessed in a fixed time. This is often done using an indirection. The names of the record elements are stored in a table with the same number of elements, such that for each element the table contains its starting location. This table is automatically created by the compiler. The characteristics of an array are listed in table 2.

Table 2: Record characteristics

Record	
Operations	**Complexity**
UPDATE	O(c)
RETRIEVE	O(c)
Access	**Direct**
Size	**Fixed**

c = constant

Like an array, the size of a record cannot be changed dynamically, but we can store elements of a different type in it.

3.3. Linked lists

Linked lists are the simplest form of dynamic data structures. Each element is represented separately, and all elements are connected by pointers (see figure 2.1). A pointer is a variable that holds as its value the address of another element.

There are two major drawbacks to the linked list representation. First it requires more memory space. There is one additional pointer per element. Secondly, if we want to look at the, for example, 30th element, we start at the beginning and look at 29 pointers, one at a time. With arrays, we make a simple calculation and find the 30th element quickly. The characteristics of the linked list are listed in table 3.

Table 3. Linked list characteristics

Linked list	
Operations	**Complexity**
CLEAR	O(c)
INSERT	O(n)
DELETE	O(n)
RETRIEVE	O(n)
NEXT	O(c)
Access	**Sequential**
Size	**Dynamic**

c = constant, n = number of elements in linked-list.

The data structures array, record and linked list and pointer are the basic data structures, and are predefined in many programming languages. They are used in a wide range of applications. The following data structures are more complex can be created by a mapping to the basic data structures.

3.4. Lists

A list is a sequence of elements with a certain ordering. Lists are a particularly flexible structure because they can grow and shrink on demand, and elements can be accessed, inserted, or deleted at any position within a list. Lists arise in applications such as information retrieval, programming language translation, and simulation.

Lists have a certain ordering. The ordering defined in chapter 2 was an ordering by position. In algorithms however, there is often an ordering by 'magnitude'. Then, the insertion of an element is not on a fixed position, but on a place where it fits by its magnitude. The first, we will call an 'unordered' list (The list is ordered by position, but the 'magnitude' of the elements is unordered), and the second an 'ordered' list.

3.4.1. Implementations

For the implementation of lists, there are two methods commonly used. That is, the list is mapped on an array, or the list is mapped on a linked list (see figure 3.1). The advantage of the first method is that all elements are direct accessible, but the size of the list has a fixed upper bound. The advantage of the second method is that the size of the list has no upper-bound, but the access to the list is sequential.

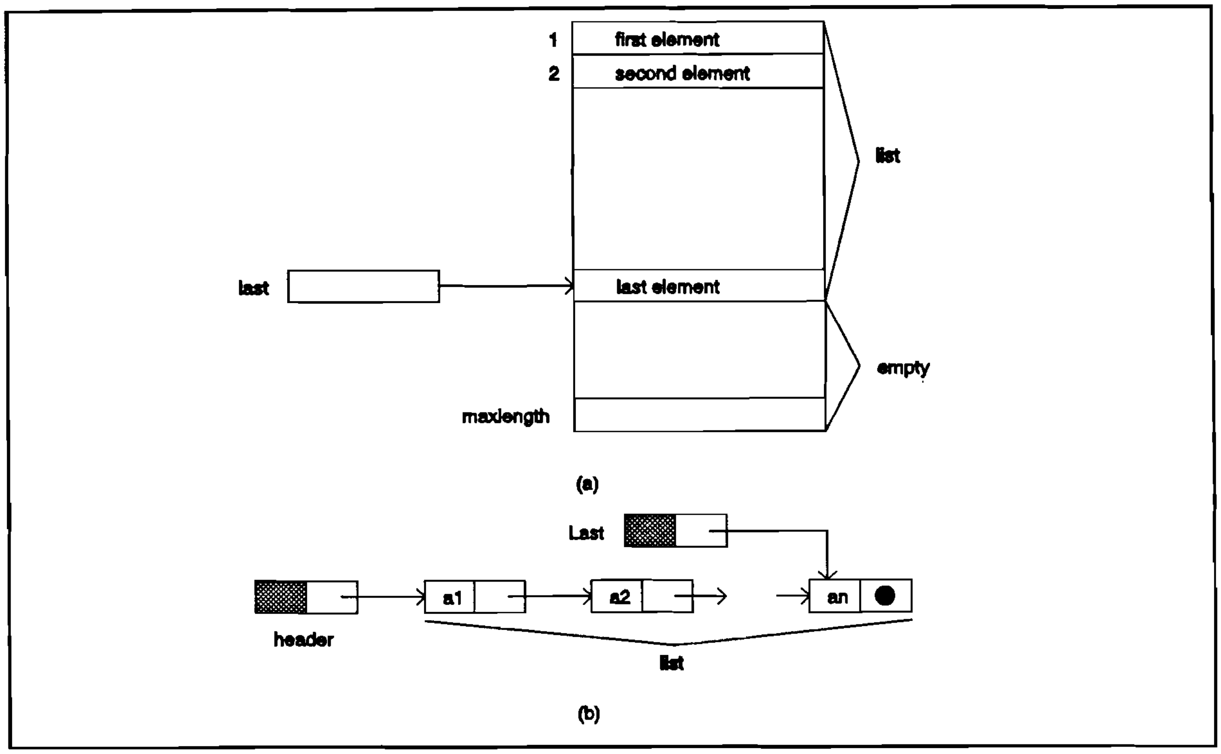


Figure 3.1. *(a) Array and (b) linked list implementation of a list.*

A third mapping, which is used very little, is interesting for a mapping to hardware. This method is called a cursor implementation, and simulates the linked list implementation (see figure 3.2). The list is mapped on an array, which contains record elements. The first record entry contains the data, while the second entry contains an index (cursor) to the next array entry of the list. The head of the list is stored in a separate variable, while a second list contains the available array elements.

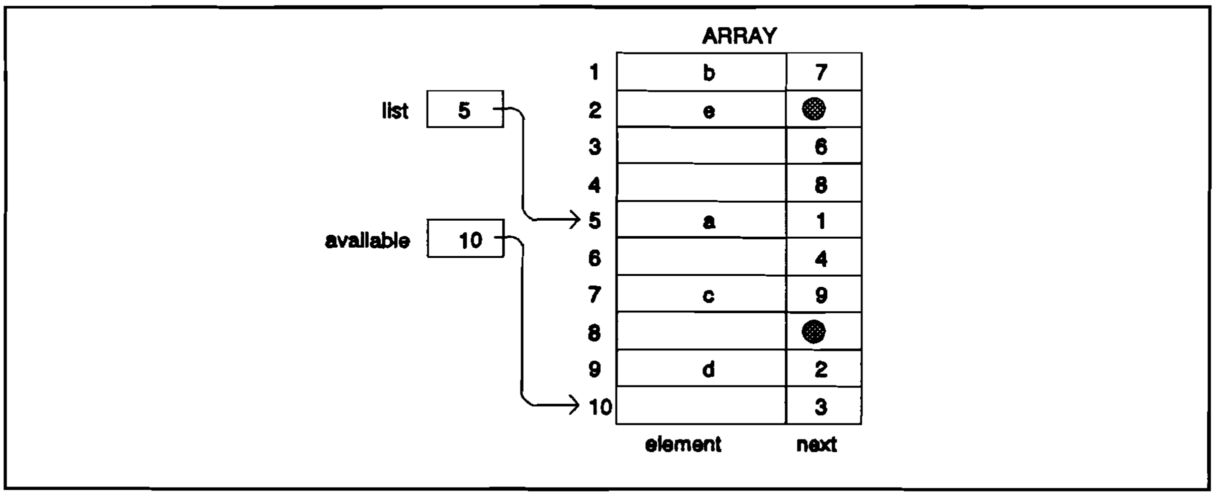


Figure 3.2. *Cursor implementation of a list.*

All three representations make it possible to implement the operations. The characteristics are given in table 4.

Table 4: List characteristics

	List			
	Complexity			
Implement.	Array		Linked list	
Operation	Ordered	Unordered	Ordered	Unordered
CLEAR	O(c)	O(c)	O(c)	O(c)
INSERT	O(n)	O(n)	O(n)	O(n)
LOCATE	O(log n)	O(n)	O(n)	O(n)
RETRIEVE	O(log n)	O(c)	O(n)	O(n)
DELETE	O(n)	O(n)	O(n)	O(n)
FIRST	O(c)	O(c)	O(c)	O(c)
END	O(c)	O(c)	O(c)	O(c)
EMPTY	O(c)	O(c)	O(c)	O(c)
FULL	O(c)	O(c)	—	—
Access	Direct/Seq.	Direct/Seq.	Sequential	Sequential
Size	Fixed	Fixed	Dynamic	Dynamic

c = constant, n = number of elements in list.

For the LOCATE and RETRIEVE operation of the sorted array implementation, a search algorithm (binary search) is used.

3.4.2 Algorithms

The two major problems for lists are -- search an element and -- sort the list. Several algorithms have been developed for these problems, which will not be discussed, but the use and complexity will be given.

For 'unsorted' lists or lists implemented by a linked list, searching for an element is only possible by a linear search.

For an 'ordered' list implemented by an array, there are several algorithms known for searching. Often used algorithms are:

- Linear search: The algorithm inspects all elements of the list, one by one, for a searched element and decides whether the element is found. The complexity is O(n). This algorithm is used when the list contains few elements (n is small).

- Binary search: The algorithm looks at the element in the middle position of the list. If the searched element is smaller than the middle element, than it is positioned in the first half of the list, else in the second half. Then we repeat this search with the half (sub)list that contains the element etc. The complexity of this algorithm is O(log n). This algorithm is used when the list contains much elements (n is big).

- Interpolation search: This algorithm guesses by interpolation the position of the searched element. If the searched element is smaller than the element on the guessed position (pivot), the operation is repeated on the first part of the list, else on the second part. This operation is repeated until the search is successful, or the element is not in the list. The complexity of this algorithm is O(log log n). This algorithm is only this good if the elements are uniformly distributed (i.e. equally distributed). Otherwise, the binary search is just as good or even better than interpolation search.

Several algorithms are known for the sorting of an 'unordered' list, which is implemented by an array. Often used algorithms are:

- Radix sort: (Bucket sort) The elements of a list are retrieved and then sorted in a number of buckets, depending on a part of the element. For example, if we want to sort numbers from 100 to 999, we can sort first on the first digit and put the resulting lists in buckets. Then we know that the first bucket contains a list with the elements form 100 to 199. We continue the sorting by looking to the next digit, etc. The complexity of this method is O(nk), where n is the number of elements and k is the number of search iterations. This method is not 'in place' i.e., there is extra storage necessary.

- Insertion sort: This algorithm divides the list in two parts, a sorted part and an unsorted part. Then it takes the first element of the unsorted and inserts it on it's correct position in the sorted part. This algorithm has the complexity O(n^2) and is 'in place'.

- Selection sort: This algorithm divides the list also in two parts. It takes the maximum unsorted element and adds it to the beginning of the sorted list. The algorithm is 'in place' and the complexity is O(n^2).

- Merge sort: This algorithm splits the list into two equal or close-to-equal parts. Then, each part is sorted recursively. Finally, the two sorted parts are merged (sorting by picking each time the smallest element of both lists, i.e. the first element of one of the lists) into one sorted list. The complexity of this 'divide and conquer' algorithm is O(n log n). This is good, but the algorithm is not 'in place', and it is difficult to implement.

- Quicksort: This algorithm is also a 'divide and conquer' algorithm. We do not divide the list in two equal parts, but we choose element value (pivot). The elements of the list are swapped, so that all elements smaller than the pivot are in the first part of the list and elements larger than the pivot are in the second part of the list. Then the algorithm is applied to both subparts. Finally all element will be sorted. If we choose the pivot well (the parts should be equal) than we have an algorithm of complexity O(n log n). If we choose the pivot bad, it goes to O(n^2).

We should keep in mind, that we can use these algorithms for all data structures thatwhich use a list, implemented by an array. For an exact description of these algorithms, you should see [AHO83], [SED83], [MAN89] and [WAR90].

3.4.3. Stacks

The stack is a special list in which all insertions and deletions take place at one end, called the top. Because the stack is a list, any list implementation can be used for a stack. The restrictions of the stack make it much easier to implement a stack in software. The stack is usually mapped on an array, with an extra variable to hold the end of the list. It is also possible to map a stack on a linked list, with a cursor to the top element, but this encounters less frequently (see figure 3.3.).

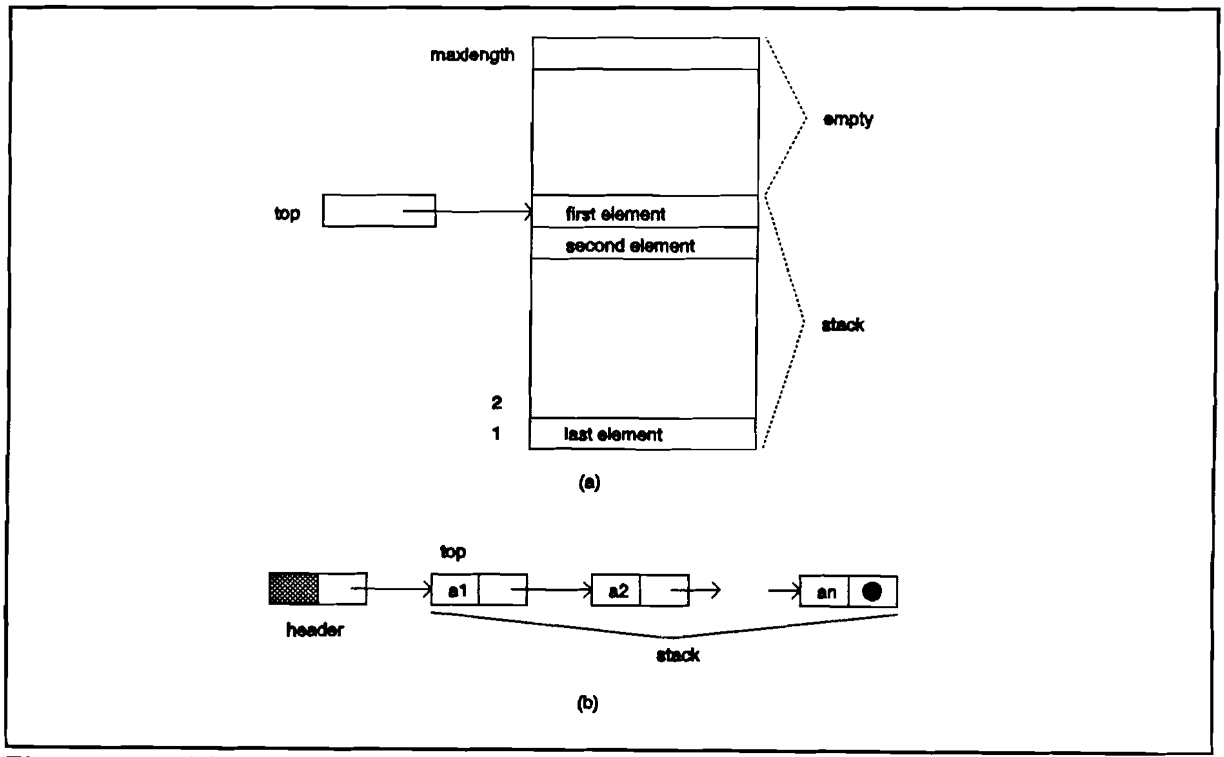


Figure 3.3. *(a) Array and (b) linked list implementation of a stack.*

The characteristics of the stack are presented in table 5. Normally spoken, the access to a linked list structure is sequential, but we only insert and retrieve elements at the front of the list, and so we need only one access to insert or retrieve an element. That's why the access to the linked list in this case is called direct. The linked list implementation is more difficult to implement, and it uses per element additional storage space.

Table 5: Stack characteristics

	Stack	
	Complexity	
Operation	Array	Linked list
CLEAR	O(c)	O(c)
TOP	O(c)	O(c)
POP	O(c)	O(c)
PUSH	O(c)	O(c)
EMPTY	O(c)	O(c)
FULL	O(c)	—
Access	Direct	Direct
Size	Fixed	Dynamic

c = constant.

Stacks are used for controlling the recursive procedures in programming languages. Stacks are automatically created and destroyed by the host of the process, store state data and other data temporarily, while another (or the same) process is called. After finishing the called process, the stopped process can continue in its old state by retrieving it's state data from the stack. Stacks are also often used in language compilers implementations.

3.4.4. Queues

A queue is another special list, where the items are inserted at one end of the list(the rear) and deleted at the other end of the list (the front). As for stacks, any list implementation is possible for queues, but we can take advantage of the fact that insertions are only done at the rear and deletions are only done at the front. We can keep a pointer (or cursor) to the first and last element of the list. This makes it possible to implement the operations on the queue much more effective.

We can implement the queue with an array and a linked list. We use a circular array, to make optimal use of the memory space. Both implementations are shown in figure 3.4. The characteristics of the queue are found in table 6.

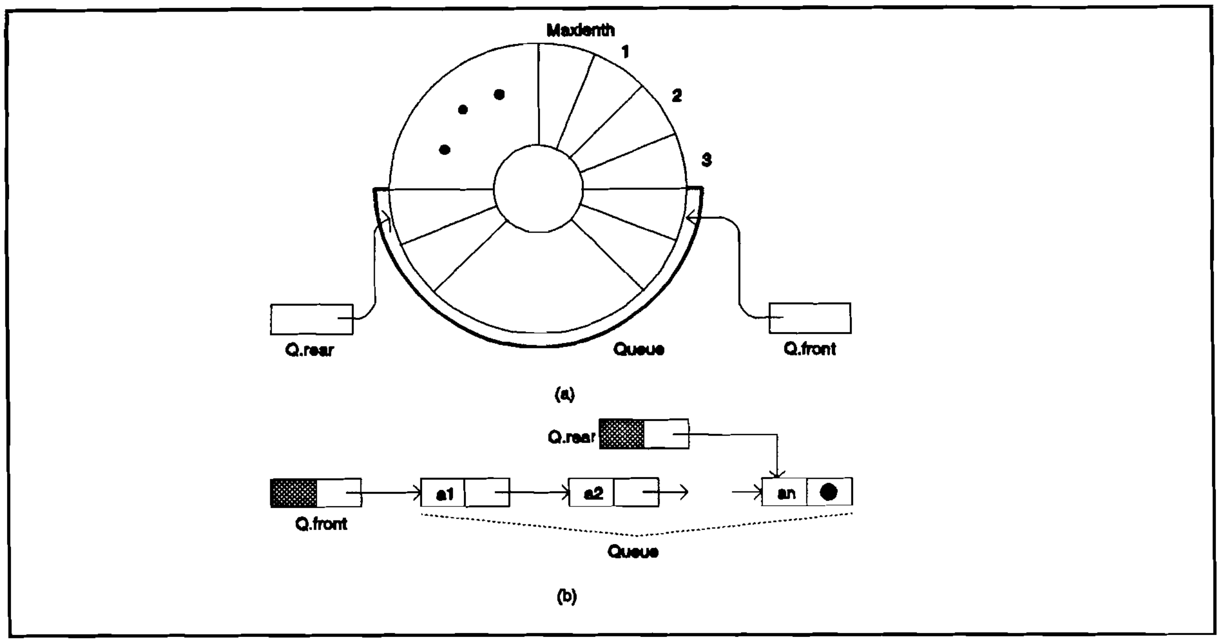


Figure 3.4. *(a) Circular array and (b) linked list implementation of a queue.*

Table 6: Queue characteristics

	Queue	
	Complexity	
Operations	Array	Linked list
CLEAR	O(c)	O(c)
FRONT	O(c)	O(c)
ENQUEUE	O(c)	O(c)
DEQUEUE	O(c)	O(c)
EMPTY	O(c)	O(c)
FULL	O(c)	—
Access	Direct	Direct
Size	Fixed	Dynamic

c = constant.

Queues are mostly used as buffer between two processes, which communicate with each other.

3.5. Tables

A table is a list of records with two different fields, i.e. the key field and the data field. Tables are mostly used for converting data of one type to another type. For example, we can think of a video memory, with for every pixel (=position) an intensity and a colour. The key is the pixel position and the data is the colour and intensity. The key of every record is unique, which makes it possible to identify every record of the table. The table is addressed by the key of a record, so if we want to retrieve a record with key *K*, then we have to look in the table for the record with key *K*.

3.5.1 Implementations

There are several software implementations for a table. The first one is a mapping to an array of records. It is also possible to map the table to a linked list of records. Both implementations are shown in figure 3.5.

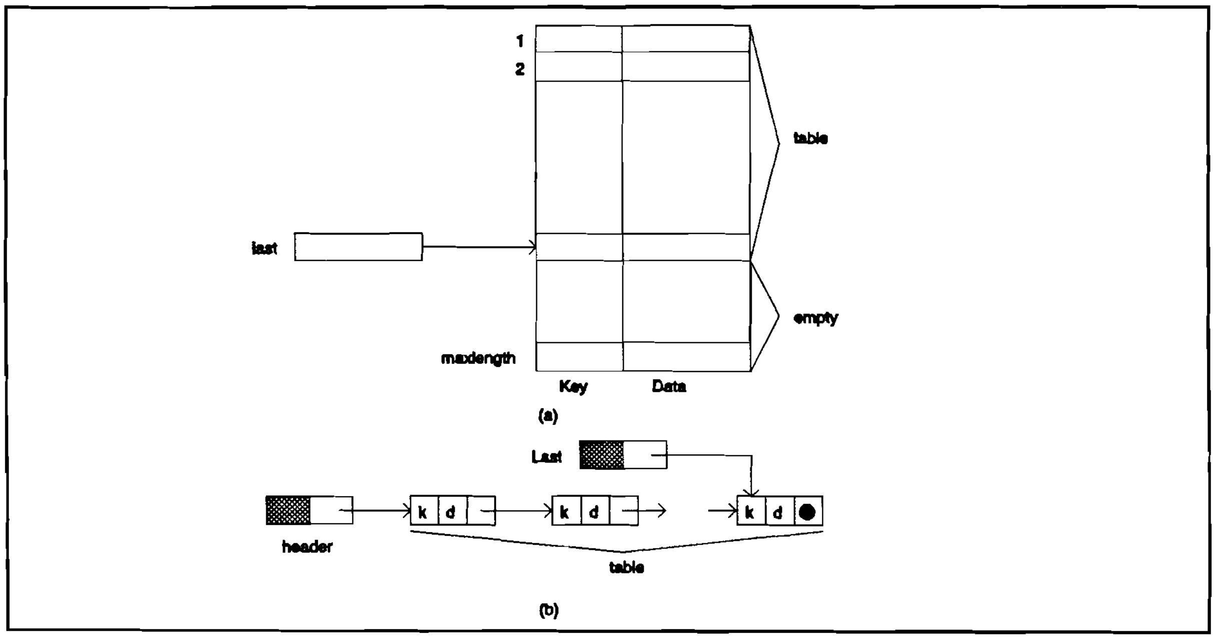


Figure 3.5. *(a) Array and (b) linked list implementation table.*

For every INSERT, DELETE, RETRIEVE and MEMBER operation we have to search the whole list, so the complexity of most of the operations is O(n), which is not good for a table (tables are often used for fast reference). But there is an other implementation technique, called *hashing*. Then, the key is translated by a translation function to an address. We call such a translation function a *hash function*. It can occur that the computed address does not contain the record being sought, but an other record with a different key. When this happens, a search of other addresses is required, and this is known as *rehashing*. It is important that the hash function satisfies the following properties:

1. They compute rapidly.
2. They produce a random distribution of index values.

It is difficult to find a *perfect hashing function*. To find a good hashing function, it is necessary to know all possible keys and the available memory space in advance. With a hash table it is possible to retrieve and insert records in a fast way.

There are two implementations of hashing, open or external hashing and closed or internal hashing. In figure 3.6.(a). we see the basic structure for open hashing.

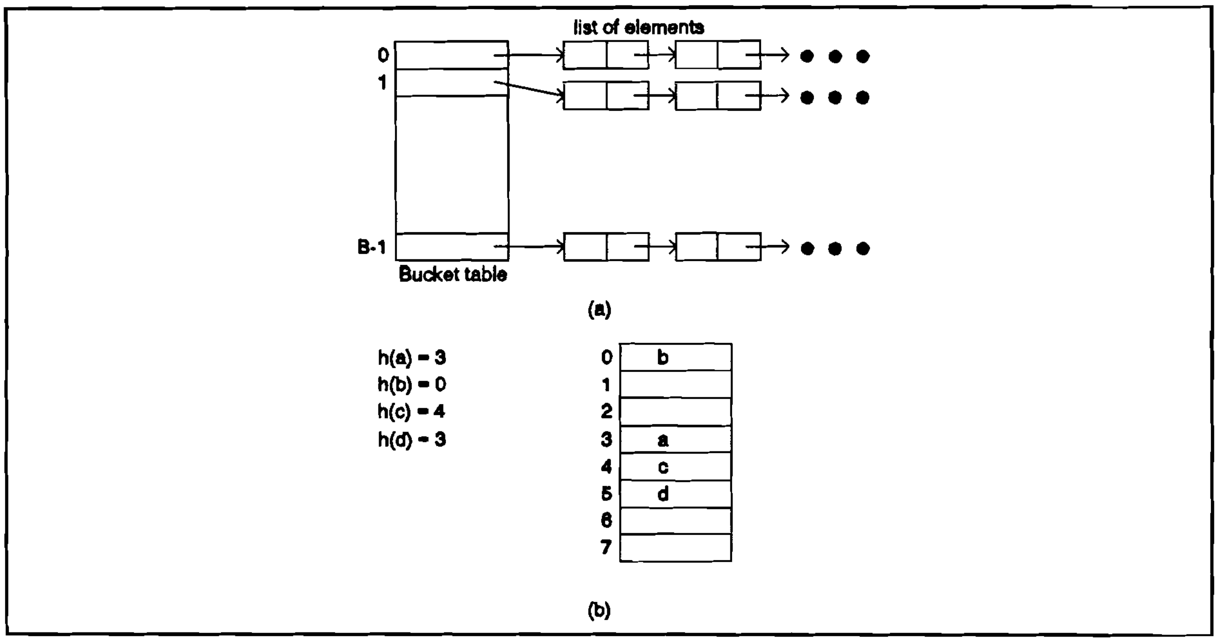


Figure 3.6. *(a) Open hashing and (b) closed hashing.*

The input set (keys) is divided in a finite number of classes. The partitioning is done by a hash function *h(x)*, which computes the bucket number *0..B-1* where the record is stored. Each bucket contains a pointer to a linked list, containing records. We need a constant amount of time to find the bucket, and a small amount of time to find the record in the linked list. For a good performance, we have to choose a hash function that gives a random distribution of elements over the buckets, and the number of buckets *B* has to be greater than n/2.

The closed hashing structure is simpler (figure 3.6.(b)), but the hashing strategy is more complicated. Now if we want to insert an element and we have computed the address of a record, it is possible that this address is occupied by an other record.

Then we have to *rehash* and find an other place where we can store the record. We can use a *rehash function* or use a simple strategy by picking the first free place. This last strategy is used in the example of figure 3.6.(b).

If next an element is not on its computed address, we have to look at the addresses, until an empty place is reached. If in the meanwhile an element is deleted, it is possible that we reach an empty place, before we reach the searched element. That is why we have to mark places of deleted elements. For a good performance, we have to choose a good hash function, and the size of the table should be greater than 1.1*B. See also [AHO83], [KOH87]. The characteristics of each implementation, and the complexity of the operations is given in table 7.

Table 7: Table characteristics

	Table			
	Complexity / representation			
Implementation	**Array**	**Linked list**	**Open hashing***	**Closed hashing***
CLEAR	O(c)	O(c)	O(B)	O(B)
INSERT	O(n)	O(n)	O(c+n/B)	O(c+1/(1-n/B)
DELETE	O(n)	O(n)	O(c+n/B)	O(c+1/(1-n/B)
RETRIEVE	O(n)	O(n)	O(c+n/B)	O(c+1/(1-n/B)
MEMBER	O(n)	O(n)	O(c+n/B)	O(c+1/(1-n/B)
FULL	O(c)	—	—	O(c)
EMPTY	O(c)	O(c)	O(c)	O(c)
Access	Sequential	Sequential	Direct/Seq.	Direct/Seq.
Size	Fixed	Dynamic	Dynamic	Fixed

c = constant, n = number of elements in table.
* B = number of buckets (open) c.q. size of hash table (closed).

3.6. Trees

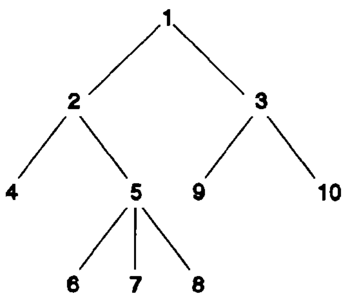
A tree imposes a hierarchical structure on a collection of items. Familiar examples of trees are organization charts. Trees are used to analyze electrical circuits and to represent the structure of mathematical formulas. Trees also arise naturally in many different areas of computer science. For example, trees are used to organize information in database systems and to represent the syntactic structure of source programs for compilers.

3.6.1 Implementations

There are many different variants of trees, and each tree has its own optimized set of operations. But first of all, we'll discuss some implementations of the tree defined in chapter 2. These implementations are mostly a mix of an array and a linked list implementation, to make the operations as effective as possible.

We suppose that the nodes are named *1,2,3,..,n*. This numbering can be random, but we suppose that we start numbering with number *1* and stop with number *n* for a tree with *n* nodes.

The array implementation of a tree uses the property that each node has a unique parent. The parent of every node is stored in an array. The root, which lacks a parent, will point to a null node. With this representation the operation PARENT, which looks for the parent of a node, can be executed in fixed time. A path going up the tree, that is, from node to parent to parent, and so on, can be traversed in a time proportional to the number of nodes on the path. We support the LABEL operator by adding another array *L*, such that *L[i]* is the label of node *i*, or by defining the elements of array A to be records consisting of an integer (cursor) and a label. An example of such an implementation is given in figure 3.7.



(a)

	1	2	3	4	5	6	7	8	9	10
A	0	1	1	2	2	5	5	5	3	3

(b)

Figure 3.7. *(a) A tree and its (b) parent pointer representation.*

Such a parent representation does not facilitate operations that require child-of information. Given a node *n*, it is expensive to determine the children of *n*, and the implementation does not specify the order of the children of the node. Thus, operations like LEFT_CHILD and RIGHT_SIBLING are not well defined. We could impose an artificial order, for example, by numbering the children of each node after numbering the parent, and numbering the children in increasing order from left to right. Then the operations LEFT_CHILD and RIGHT_SIBLING are defined. The operations PARENT, LABEL, ROOT and CLEAR are also defined, but the operation CREATE can not be implemented with this structure. This implementation can't represent multiple trees, which is necessary for the operation CREATE.

An important and useful representation of trees is to form for each node a list of its children. The list can be represented by any of the methods suggested in paragraph 3.4., but because the number of children of each node can vary, the linked-list representations are often more appropriate. Figure 3.8. suggests how the tree of figure 3.7.(a) might be represented.

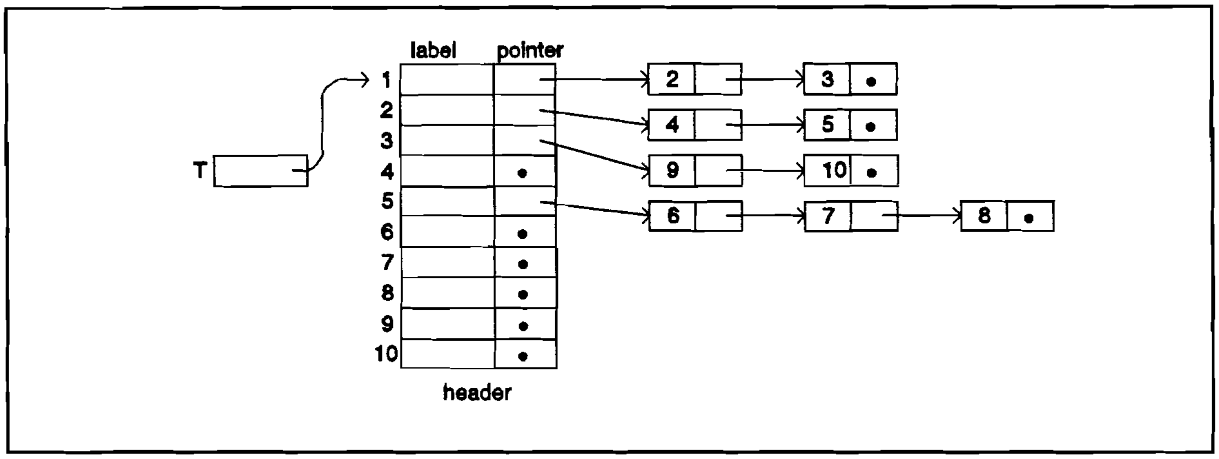


Figure 3.8. *Linked-list representation of a tree.*

An array of header cells is indexed by the nodes, and each cell contains the label of that node and a pointer to a linked list of child nodes. If a node has no children, then a nil pointer is used. The root is stored in a separate root field. With this representation it is easy to implement the operations LEFT_CHILD, LABEL, ROOT and CLEAR. The operations RIGHT_SIBLING and PARENT are more difficult to implement, and the operation CREATE can not be implemented. An extra operation should be defined to build the tree.

The data structure described above can't create large trees from smaller ones, using the CREATE operator. The reason is that all trees have a separate array of headers for their nodes. For example, to implement CREATE($T,2,l,T_2,T_2$) we would have to copy T_1 and T_2 into a third tree T and add a new node with label l and two children - the roots of T_1 and T_2.

If we wish to build trees from smaller ones, the representation of nodes from all trees should share one area. This can be done by adding an array with pointers to the root of every tree. If we want to implement the RIGHT_SIBLING operation effectively, we have to add pointers to the right_sibling element in the tree. This representation is given in figure 3.9.

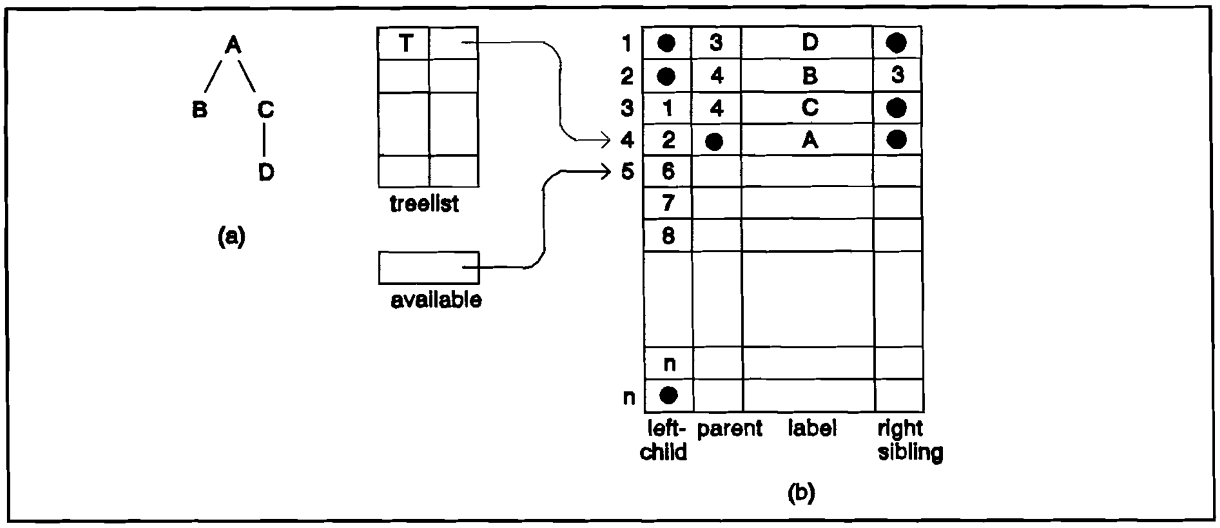


Figure 3.9. *(a) A tree and its (b) left-child, right sibling implementation.*

With this representation it is easy to implement all defined operations. There are many special trees, each for a special purpose, and all with its own operation set. Usually, trees are used to store information, which has to be searched for certain elements. They are also used for the implementation of other data structures, like sets. The characteristics of the implementations and the complexity of the operations are given in table 8.

Table 8: Tree characteristics

	Tree		
	Complexity / representation		
Operations	Parent pointer	Linked list	Left_child-Right_sibling
CLEAR	O(n)	O(n)	O(c)
PARENT	O(c)	O(n^2)	O(c)
LEFT_CHILD	—	O(c)	O(c)
RIGHT_SIBLING	—	—	O(c)
LABEL	O(c)	O(c)	O(c)
CREATE	—	—	O(c)
ROOT	O(log n)	O(c)	O(c)
Access	Direct	Direct/seq.	Direct
Size	Fixed	Dynamic	Fixed/Dynamic

c = constant, n = number of nodes in tree.

There are some well known algorithms for trees. The depth-first search (dfs) and the breadth-first search (bfs) are some often used search algorithms. These algorithms are also used with graphs (a tree is a very special kind of graph), and are discussed in section 3.8. See also [AHO83], [COR90], [MAN89] and [SED88].

3.7. Sets

The set is a very often used data structure, often with only a subset of operations. Using this knowledge, we can define several representations for a set.

3.7.1. Simple implementations

The best implementation of a set structure depends on the operations to be performed and on the size of the set. When the set consists of a small 'universal set' whose elements are the integers *1,..,N* for some fixed *N*, then we can use a *bit-vector* (boolean array) implementation. The set is represented by a bit vector in which the i^{th} bit is true if *i* is an element of the set. The major advantage is that MEMBER, INSERT, and DELETE operations are performed in fixed time by directly addressing the appropriate bit. UNION, INTERSECTION, and DIFFERENCE can be performed in a time proportional to the size of the universal set. Also the other operations can be executed in a constant amount of time. The main disadvantage of this implementation is that all elements must be in a small, predefined universal set.

It should be evident that sets can be represented by lists, where the items in the list are the elements of the set. If we use a linked list, it uses, unlike the bit-vector representation, space proportional to the size of the set represented, not the size of a universal set. Moreover, the list representation is more general since it can handle sets that need not be subsets of some finite universal set.

When we have operations like INTERSECTION on sets L_1 and L_2 represented by lists, we have several options. If the universal set is linearly ordered, then we can represent a set by a sorted list. With unsorted lists we must match each element on L_1 with each element on L_2, a process that takes O(n^2) steps on lists of length *n*. But if we use sorted lists, this operation takes only O(n) steps. On the other hand an insertion in an unsorted list takes only O(c) steps (if we insert at end of list), while an insertion in a sorted list takes O(n) steps.

Another representation is the hash table. The set-element is stored in the key field, and the data field will be empty. The operations INSERT, DELETE, and MEMBER will be executed in a constant amount of time, but other operations can not be implemented. The characteristics of the implementations discussed above are given in table 9.

Table 9. Set characteristics (1)

Implement.	Set						
	Complexity / representation						
	Bit-vector[*]	List - Array		List - Linked list		Hash table[**]	
Operation		Ordered	Unordered	Ordered	Unordered	Open	Closed
UNION	O(N)	O(n)	$O(n^2)$	O(n)	$O(n^2)$	—	—
INTERSECTION	O(N)	O(n)	$O(n^2)$	O(n)	$O(n^2)$	—	—
DIFFERENCE	O(N)	O(n)	$O(n^2)$	O(n)	$O(n^2)$	—	—
CLEAR	O(n)	O(c)	O(c)	O(c)	O(c)	O(B)	O(B)
MEMBER	O(c)	O(log n)	O(n)	O(n)	O(n)	O(c+n/B)	O(c+1/(1-n/B))
INSERT	O(c)	O(n)	O(n)	O(n)	O(n)	O(c+n/B)	O(c+1/(1-n/B))
DELETE	O(c)	O(n)	O(n)	O(n)	O(n)	O(c+n/B)	O(c+1/(1-n/B))
ASSIGN	O(N)	O(n)	O(n)	O(n)	O(n)	—	—
EQUAL	O(N)	O(n)	$O(n^2)$	O(n)	$O(n^2)$	—	—
MIN	O(N/n)	O(c)	O(n)	O(c)	O(n)	—	—
MAX	O(N/n)	O(c)	O(n)	O(c)	O(n)	—	—
Access	Direct	Dir./Seq.	Dir./Seq.	Sequent.	Sequent.	Dir./Seq.	Dir./Seq.
Size	Fixed	Fixed	Fixed	Dynamic	Dynamic	Fixed	Dynamic

[*] N = maximal set size, n = number of set elements
[**] see note table 7.

3.7.2 Advanced implementations

We will now introduce some data structures for sets that permit a more efficient implementation of common collections of set operations. These structures however, are often more complex and are often only appropriate for large sets. All are based on various kinds of trees, such as binary search trees and balanced trees.

Binary search tree implementation of sets

A representation of a set can be given by a binary search tree, a basic data structure for representing sets whose elements are ordered by some linear order (denoted by '<'). This structure is useful when we have a set of elements from a universe so large that it is impractical to use the elements of the set themselves as indices into arrays. A binary search tree can support the set of operations INSERT, DELETE, MEMBER and MIN, taking on the average O(log n) steps per operation for a set of n elements.

A *binary search tree* is a tree in which the nodes are labelled with elements of a set. The important property is that all elements stored in the left subtree of any node *x* are all less than the element stored at *x*, and all elements stored in the right subtree of *x* are greater than the element stored at *x*. This condition, called the *binary search tree* *property*, holds for every node of a binary search tree, including the root. Figure 3.10 shows two binary search trees representing the same set of integers.

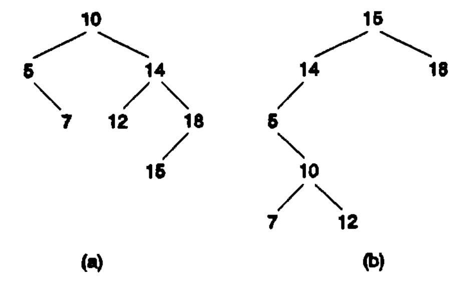


Figure 3.10. *Two binary search trees.*

Suppose a binary search tree is used to represent a set. The tree property makes testing for membership in the set simple. To determine whether *x* is a member of the set, first compare *x* with element *r* at the root of the tree. If $x = r$, we are done, the element is found. If $x < r$, then *x* can only be in the left subtree of *r*, if $x > r$, then *x* can only be in the right subtree of *r*. Then we repeat this operation on the left, c.q. right subtree.

The operation INSERT is easy. We have to compare the element *x* with a node element. If the node element equals *x*, then *x* is already in the set. If *x* is smaller than the node element, then we have to repeat this comparison with the element of the left child. If there is no left child, then that is the position, where *x* has to be inserted. This is also true for the right child.

MIN is also easy to implement. We only have to search in the tree for the first left child node that has no left child. Using this, the operation DELETEMIN is also easy. We replace the element found with MIN, by its right child.

The operation DELETE is implemented by using DELETEMIN. When we want to delete element *x*, we have to locate this element. If it is found, we delete it and replace it by the minimal element of its right sub tree (if exist). In figure 3.11 we see an insertion of 6 and a deletion of 10 in the tree of figure 3.10.(a).

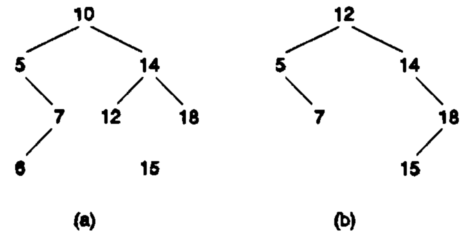


Figure 3.11. *Tree 3.10(a) after insertion of 6 (a) and after deletion of 10 (b).*

A binary search tree is mostly implemented by linked records. The implementation of the tree of figure 3.10.(a) is given in figure 3.12.

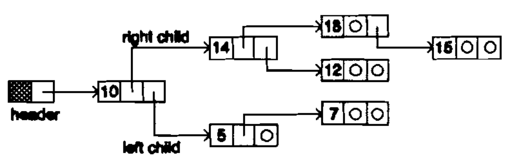


Figure 3.12. *Binary search tree implementation.*

If we insert a sorted list into a set, we don't get a balanced binary tree, but a chain of nodes, so the complexity of the operations would be O(n). If we assume a random input sequence, we have a complexity O(log n).

Balanced tree implementation of sets

Another implementation is the balanced tree. This representation has as advantage that the execution of the operations has a time of O(log n) worst case. There are many implementations of balanced trees, such as the 2-3 tree, the AVL-tree, the B-tree etc. One of the balanced tree implementations is the '2-3 tree'. A 2-3 tree is a tree with the following two properties.

1. Each interior node has two or three children.
2. Each path from the root to a leaf has the same length

We shall consider a tree with zero nodes or one node as special cases of a 2-3 tree.

We represent sets of elements that are ordered by some linear order <, as follows. Elements are placed at the leaves; if element *a* is to the left of element *b*, then *a* < *b* must hold. We shall assume that the ordering of elements is based on one field of a record that forms the element type; this field is called the *key*. For example, elements might represent information about students, and the key field might be 'id-number'.

At each interior node we record the key of the smallest element that is a descendant of the second child. If there is a third child, we record the key of the smallest element descending from that child as well. Figure 3.13. is an example of a 2-3 tree.

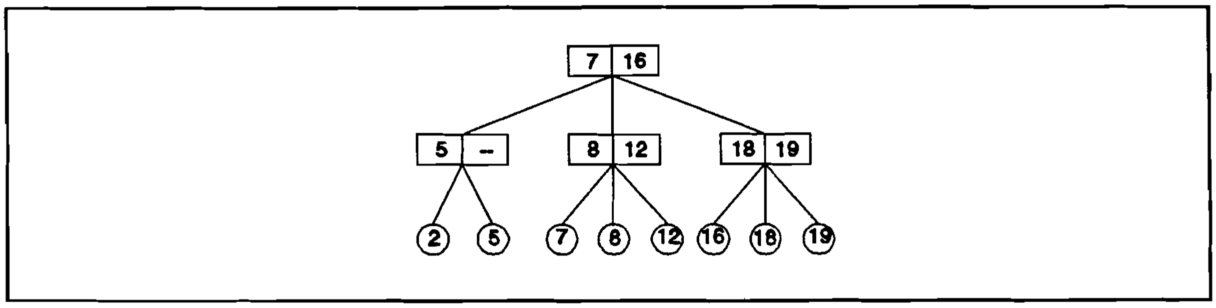


Figure 3.13. *A 2-3 tree.*

Observe that a 2-3 tree representing a set of *n* elements requires at least 1 + $\log_3 n$ levels and no more than 1 + $\log_2 n$ levels. Thus, path lengths in the tree are O(log n).

To test membership of a record with key *x*, we just have to move down the tree and compare *x* with the keys stored at the internal nodes. If $x <$ first key, then we go to the first sub tree. If there is a second node key and if first key $\leq x <$ second key, then we search the second subtree. If second key $\leq x$, then we search the third tree. When x equals one of the node keys, we know that *x* is a member of the set, and we don't have to search further. So testing membership takes O(log n).

To insert elements, we proceed at first if we were testing membership of *x* in the set. However, at the level just above the leaves, node's children do not include *x*. If that node has only two children, we simply make *x* the third child of that node, placing the children in proper order, and adjusting the keys of the node.

Suppose, however, that *x* is the fourth child of the node. We cannot have a node with four children, so we split the node in two nodes, giving the first new node the two smallest elements and the second new node the other two. Now we have to insert an extra node among the parent of the node, just as the insertion at the leaf level. One special case occurs when we wind up splitting the root. Then we create a new root, whose children are the two nodes into which the old root was split. This is how the number of levels increase. In figure 3.14. an example is given of an insertion.

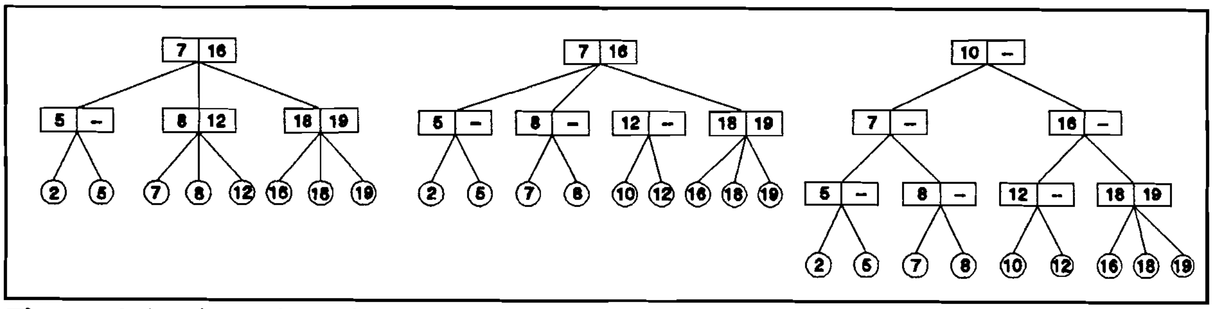


Figure 3.14. *Insertion of 10 in 2-3 tree.*

When we delete a leaf, we may leave its parent node *n* with only one child. If that node *n* is the root, delete the node and lets its lone child be the new root. Otherwise, if the parent *p* of node *n* has another child, adjacent to *n* on either the right or left, and that child of *p* has three children, we can transfer the proper one of those three to *n*. Then, *n* has two children and we are done. If that adjacent node has only two children, transfer the lone child of *n* to that adjacent node and delete node *n*. Should parent *p* now have only one child, repeat all the above recursively. An example of a deletion is given in figure 3.15.

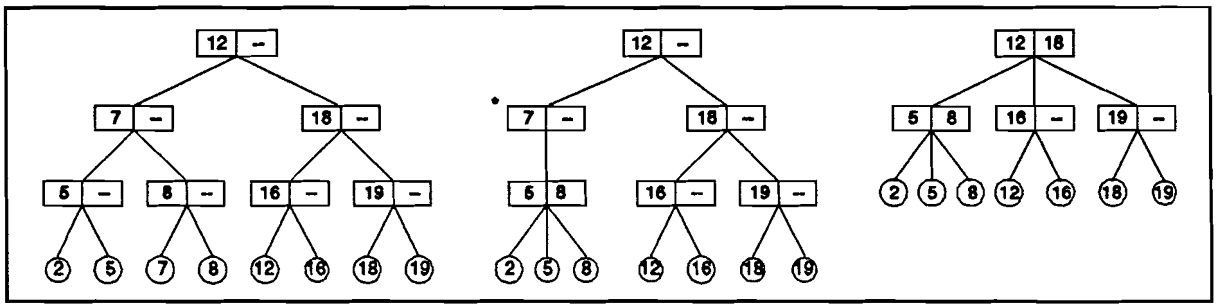


Figure 3.15. *Deletion of 7 in a 2-3 tree.*

This 2-3- tree is one of the many implementations of balanced trees. This tree can easily be extended to, for example, 2-3-4 trees. They all have in common that the maximal time needed for execution of the operations is O(log n). In table 10 an overview of the characteristics is given.

Table 10. Set characteristics (2)

	Set	
	Complexity / representation	
Operation	Binary search tree	Balanced tree
CLEAR	O(c)	O(c)
INSERT	O(log n) - O(n)	O(log n)
DELETE	O(log n) - O(n)	O(log n)
MEMBER	O(log n) - O(n)	O(log n)
DELETEMIN	O(log n) - O(n)	O(log n)
MIN	O(log n) - O(n)	O(log n)
Access	Sequential	Sequential
Size	Dynamic	Dynamic

c = constant, n = number of elements in set.

For more information about set representations, trees etc. see [AHO83], [COR90], [MAN 89], [SED88] and [WAR90].

3.8. Graphs

Graphs, representing relationships among data objects, are used for problems arising in computer science, mathematics, engineering etc. This paragraph presents the basic structures that can be used to represent graphs. Also some algorithms are given, which are often used on graphs.

3.8.1 Implementations

When we look at a graph, we should see a difference between a directed and an undirected graph. However, an undirected graph can be represented by a directed graph (figure 3.16), so the representation can be the same. This makes the graph structure more general.

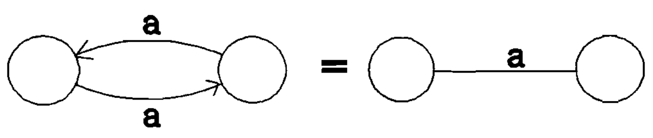
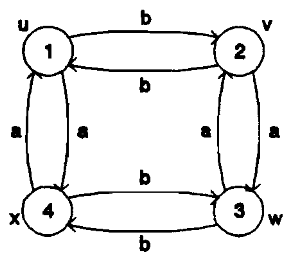


Figure 3.16. *Directed and undirected graph.*

Adjacency matrix implementation of graphs

The appropriate choice of data structure depends on the operations that will be applied to the edges and nodes of the digraph. One common representation for a digraph is the *adjacency matrix*. Suppose the nodes are in {1,2,..,*n*}. The adjacency matrix is an *n* × *n* matrix of edge labels, where A[*i*,*j*] is the label of the edge from node *i* to node *j*. If there is no edge between two nodes, there should be a special nil-value in the matrix. There should also be a list with the nodes that are present at that moment (with their node labels). Figure 3.17 shows such an implementation.



	node label
1	u
2	v
3	w
4	x

	1	2	3	4
1		b		a
2	b		a	
3		a		b
4	a		b	

adjacency matrix

Figure 3.17. *Digraph with adjacency matrix representation.*

In the adjacency matrix representation the time required to access an element of an adjacency matrix is independent of the number of nodes and the number of edges. This representation is useful in those graph algorithms in which we frequently need to know whether a given edge is present and what's its label is. The main disadvantage of using an adjacency matrix is that the matrix requires n^2 storage even if the digraph has less than n^2 edges. So if we have a sparse adjacency matrix, we can better use another representation.

Adjacency list implementation of graphs

Another common representation for a digraph is called the *adjacency list* representation. The adjacency list for node *i* is a list, in some order, of all edges adjacent to *i*. We can represent the graph by an array with all nodes and their labels (if the node exists), with a pointer from every node to the adjacency list for that node *i*. The adjacency list representation of a digraph requires a storage proportional to the sum of the number of nodes and the number of edges ($n + e$). This representation is often used when the number of edges is less than n^2. However, a potential disadvantage of the adjacency list representation is that it may take O(n) time to determine whether there is an edge from node *i* to node *j*, since there can be O(n) edges on the adjacency list for nodes *i*. Figure 3.18 shows such an implementation.

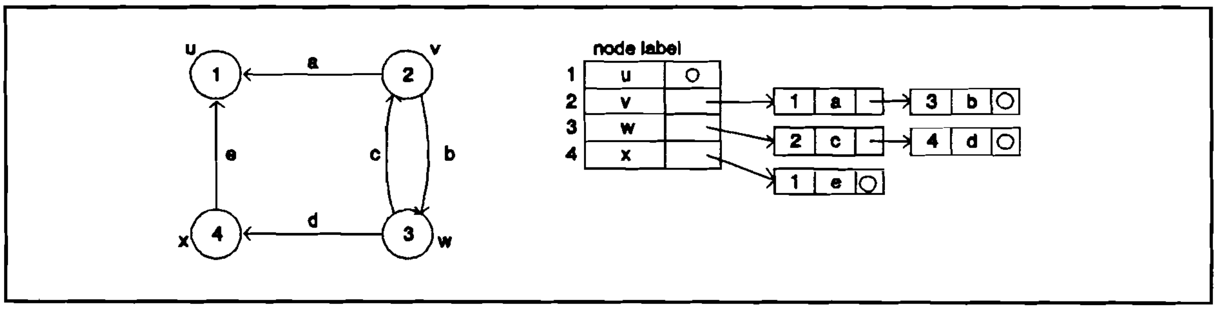


Figure 3.18. *Graph with adjacency list representation.*

In table 11 are given some characteristics of the two implementations.

Table 11. Graph characteristics

	Graph	
	Complexity / representation	
Operations	**Adjacency matrix**	**Adjacency list**
CLEAR	$O(n^2)$	O(c)
INSERT_NODE	O(c)	O(c)
INSERT_EDGE	O(c)	O(n)
DELETE_NODE	O(n)	$O(n^2)$
DELETE_EDGE	O(c)	O(n)
RETRIEVE_NODE	O(c)	O(c)
RETRIEVE_EDGE	O(c)	O(n)
Access	Direct	Direct/Seq.
Size	Fixed	Dynamic

c = constant, n = number of nodes in graph.

3.8.2 Algorithms

There are many algorithms that are used on graphs. We make a difference between algorithms for directed and undirected graphs, because each of these two types of graphs has its own use. First we give some algorithms for directed graphs. We won't explain the algorithms in detail, we will just explain the problem, give the name of an often use algorithm, and give the complexity of that algorithm.

- Single source shortest path problem: Consider a directed graph in which each edge has a non negative label, and one node is specified as the *source*. When there is no edge between two nodes, the label is ∞. Our problem is to determine the cost of the shortest path from the source to every other node, where the *length* of the path is the sum of the costs of the arcs on the graph. To solve this problem we use a greedy technique, often known as *Dijkstra's algorithm* [AHO83], [MAN89], [SED88]. The algorithm works by maintaining a set S of nodes whose shortest distance from the source is known. At each step we add to S a remaining node *n* whose distance to *S* is as short as possible. When S includes all nodes, then the solution is known. If we use an adjacency matrix representation for the graph, the running time is $O(n^2)$, but if the number of edges e is much smaller than n^2, we might better use an adjacency list representation for the digraph, and a priority queue representation for the nodes not yet selected in S. This gives a running time of O(e log n).

- All-pairs shortest paths problem: Find the shortest distance from any node to every other node. We could solve this with *Dijkstra's* algorithm, taking every node once as the source. This would take a running time of $O(n^3)$, or if e is much smaller than n^2, then it would take $O(ne \log n)$ time. There is for this problem an other algorithm known, i.e. *Floyd's algorithm* [AHO83], [MAN89], [SED88]. This takes also $O(n^3)$ time.

- Transitive closure: Given a graph $G = (V,E)$, the transitive closure $C = (V,F)$ of G is a directed graph such that there is an edge (v,w) in C if and only if there is a directed path from v to w in G. For this problem is developed an algorithm by Warshall, called *Warshall's algorithm* [AHO83], [MAN89], [SED88] that is based on Floyd's algorithm. This algorithm has also complexity $O(n^3)$.

- Depth-first search: (dfs) To solve many problems dealing with directed graphs efficiently we need to visit the nodes and edges of a digraph in a systematic fashion. Depth-first search is one important technique for doing so. Depth-first search can be used for many other algorithms. Suppose we have a digraph G in which all nodes are initially marked *unvisited*. Depth first search works by selecting on node n of G as start node, n is marked *visited*. Then each unvisited node adjacent to n is searched in turn, using depth-first search recursively. If some nodes remain unvisited, we select an unvisited node as a new start node. We repeat this process until all nodes of G have been visited. This technique is called depth_first search because it continues searching in forward direction as long as possible. We can use an adjacency list representation for the graph and an array to mark nodes visited or unvisited. The complexity is O(e), i.e the time to look to all edges adjacent to all nodes. There are several algorithms based on the depth-first search. Some of them are:

  - Depth-first spanning forest: During a dept_first search traversal of a digraph, certain edges, when traversed, lead to unvisited nodes. The edges leading to new nodes are called tree edges and they form a depth-first spanning forest.
  - Test for acyclicity: testing whether a directed graph is cyclic or not.
  - Topological sort: a process of assigning a linear order to the nodes of a directed acyclic graph so that if there is a directed edge from node i to node j, then i appears before j in the linear ordering.
  - Finding strong components: finding a maximal set of nodes in which there is a path from any node in the set to any other node in the set (cyclic).

See also [AHO83], [MAN89] and [SED83].

Now, we give some algorithms for undirected graphs.

- Minimum cost spanning tree: A *spanning tree* is a free tree (i.e. a tree with no directions) that connect all nodes of the graph. The *cost* of a spanning tree is the sum of costs of the edges in the tree. There are two popular methods for finding the minimum cost spanning tree. The first is *Prim's algorithm* [AHO83], [SED88], which has a complexity of $O(n^2)$, and *Kruskal's algorithm* [AHO83], [SED88], which has a complexity of $O(e \log e)$. If e is much less than n^2, Kruskal's algorithm is superior, although if e is about n^2, we would prefer Prim's algorithm.

- Depth-first search: (dfs) This algorithm is the same as the depth-first search for digraphs, but the implementation is simpler. The complexity is also $O(e)$.

- Breadth-first search: (bfs) From each node *n* that we visit, we search as broadly as possible by next visiting all the vertices adjacent to *n*. We repeat this on all adjacent nodes of node *n*. This strategy can also be applied to digraphs. The complexity of this algorithm is just as dfs $O(e)$ [AHO83], [SED88].

- Articulation points and biconnected components: An *articulation point* of a graph is a node *n* such that when we remove *n* and all edges incident upon *n*, we break a connected component of the graph into two or more pieces. A connected graph with no articulation points is said to be *biconnected.* Depth-first search is particularly useful in finding the biconnected components of a graph. The solution can be found in $O(n + e)$ time, i.e. $O(e)$ time if $n << e$.

- Graph matching: If a graph is divided in two sets of nodes, with only edges between nodes of the two sets, we can use this for 'matching problem'. For example, we have a set of teachers, who give each one or more courses, and we have a set of courses. Then, find the best match of teachers with courses. The solution of this problem can be given in $O(ne)$ time.

For more information about data structures, their software implementation and algorithms, see [AHO83], [COR90], [MAN89], [SED88], and [WAR90].
In the next chapter we will look to basic hardware building blocks, in special to physical memories, and their behaviour. These memories can be used to map the data structures to hardware.

4. Basic hardware building blocks

We need some basic building blocks that we can use for the implementation of data structures in hardware. Most building blocks, like adders, multipliers, registers, counters, busses, etc are well described in the manuals of many hardware description tools. We will not discuss those common building blocks, but only some very important blocks, namely the memory structures.

Data structures store data in an easy way, so if we want to map these data structures to hardware, it is necessary to know what kind of hardware memory structures there are. For that reason, we'll discuss here two memory structures, namely the random access memory (RAM) and the content addressable memory (CAM).

4.1 RAM

The random access memory is a memory what can be accessed by pointing to the position (address) of the data word. It is possible to address every single data word in the memory. That is why it is called a random access memory. There are two versions of a random access memory, which are a synchronous RAM and an asynchronous RAM. Looking at the internal structure of a RAM, we can split it up in two parts, i.e. the memory matrix and the control part (figure 4.1.). The control part controls all the possible operations, the memory matrix stores the data.

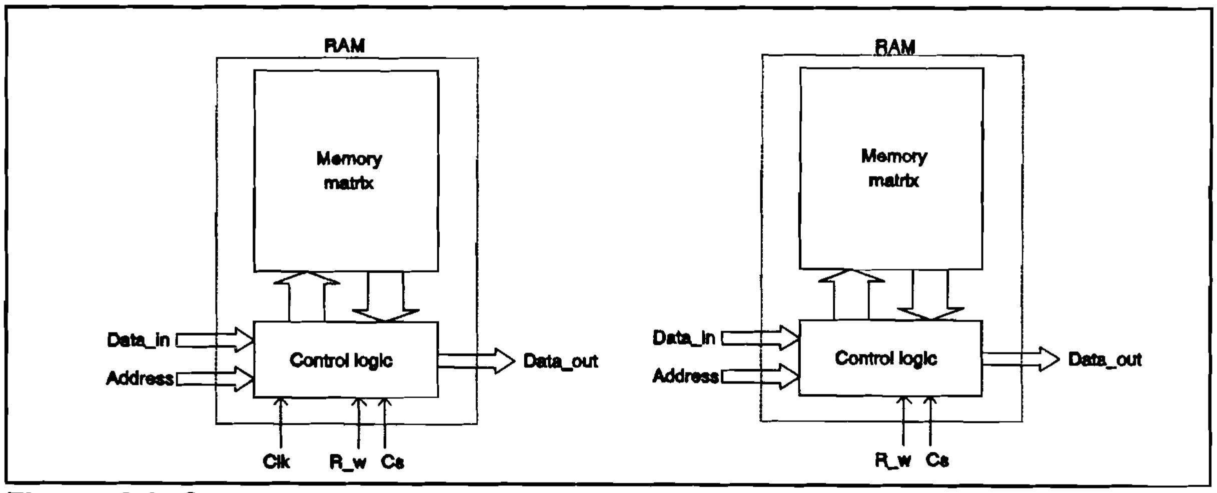


Figure 4.1. *Synchronous and asynchronous RAM.*

The first RAM shown above is a synchronous RAM, which executes operations on the first clock transition from '0' to '1'. There are no operations executed as long as the line 'chip select' (CS) is low. If the line CS is high, then the following commands can be executed:

read:

This operation is executed as the line R_W is high. The data word at the position of the address at address input port ADDRESS is read from the memory matrix and sent out to the output port DATA_OUT.

write:

This operation is executed as the line R_W is low. The data word at the input port DATA_IN is written to the memory matrix at address given by the address input port ADDRESS.

For an asynchronous RAM we have the same operations, but they are now executed after a low to high transition of CS (not on a clock, because it's not available). For more information on RAMs, see the data sheets of some RAMs.

4.2 CAM

An associative memory (content addressable memory) is a memory, in which the access to the memory element information is based on a part of or all the information stored in the memory element. We get access to the CAM by matching a part of the information with a reference data (the key). If a memory word in the CAM has the same contents as the key, this memory word is selected. In a conventional memory, the access to the memory is based on the location (address) and not on its contents. See also [LOO75] and [KOH87].

We see that the only real difference between a RAM and a CAM is that for a RAM, every element can be read and be written, and that a CAM memory also can compare. An associative memory is much more complex than a conventional memory (size CAM is about 3 to 4 times the size of a RAM), and for that reason more expensive.

To find a certain data word in the associative memory, a reference data word (key) is stored in the match register of the memory. Then, this key is compared in parallel with all data words in the memory, and just those words that 'match' with the reference word, give a match sign in the association register. This match info can be converted to an address of the first matching element. It is possible to compare just a part of the key with the memory elements. A mask is placed (a bit pattern just as long as the key) on the key, with 'cares'('1') and 'don't cares'('0'). Just those bits that 'care' (maskbit = '1'), are compared with the bits of the memory elements. It is possible that more than one memory element match with the 'masked' key, then we have a 'multiple response'. When a multiple response occurs, it is possible to read those addresses consecutive. The basic structure of a CAM is given in figure 4.2.

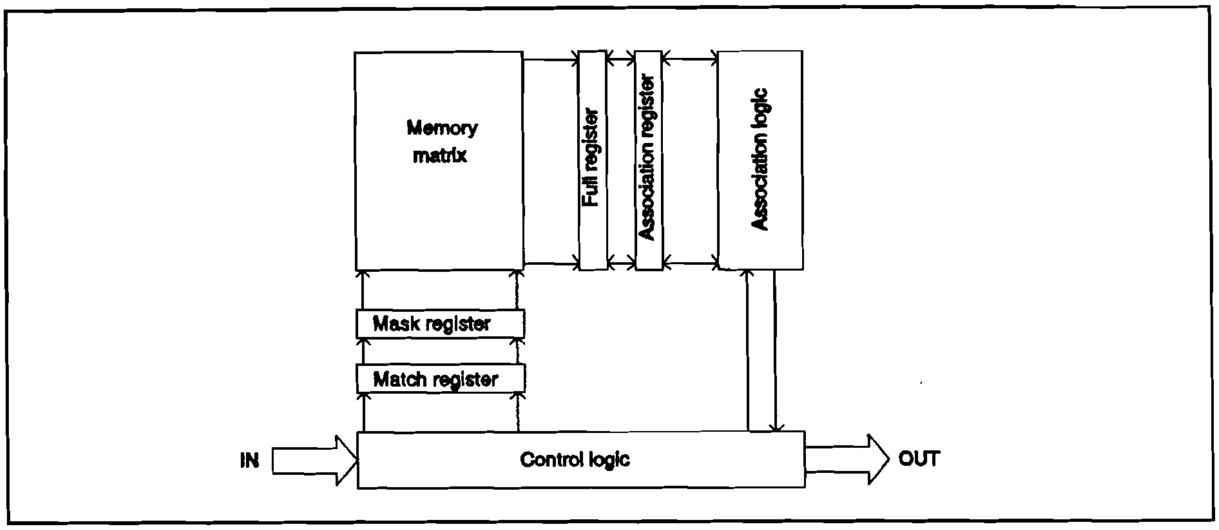


Figure 4.2. *The basic CAM structure.*

An elementary CAM is build from the following blocks:

- Memory matrix: In this bit matrix the data is stored.
- Match register: This register contains the reference data, which has to be compared with the data in the memory matrix.
- Mask register: This register contains data, which specifies what bit of the match register has to be compared and which not.
- Full register: This register, with as many places as memory words, gives a 1 on position *i* if memory word *i* is written. This register indicates whether a memory position contains useful data.
- Association register: This register, with as many places as memory words, gives a 1 on position *j* if the memory word on position *j* associates with the key in the match register.
- Association logic: This block computes the addresses of the associative memory words.
- Control logic: This block controls the all operations.

We can make a difference between several associative memory matrices, by the way the handle the matching. There are 5 types of associative memory (see also figure 4.3.), i.e:

- Identifying memory (catalog memory): All bit places of a memory must be compared to associate (no mask).
- Fixed tap memory: The memory has an associative part and a data part. The associative part can be compared with the key, the data part can only store data. All bit places of the associative part must be compared to associate (no mask).

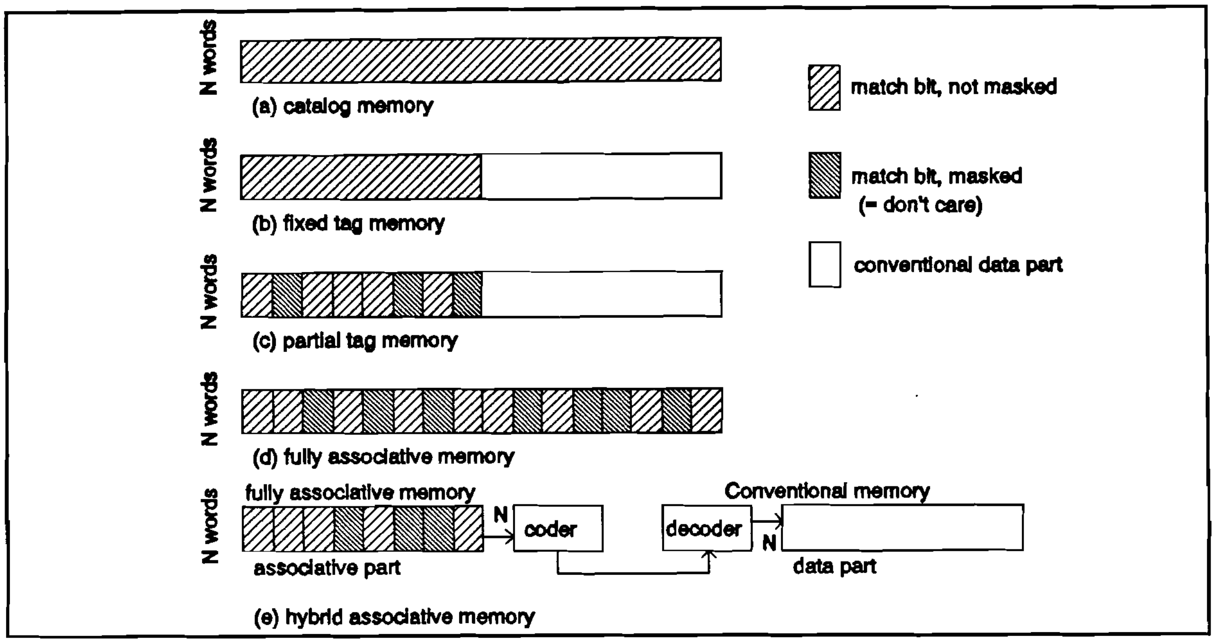


Figure 4.3. *Types of associative memory.*

- Partial tag memory: The memory has an associative part and a data part. For association can be taken any subpart of the association part (mask).
- Fully associative memory: All bits of the memory can be used for association, and every subset of bits can be used for association (mask).
- Hybrid associative memory: A fully associative memory, linked together with a data part. It is possible to link the CAM with the data part directly, but it is better to code the association word to an address, and use a data part a conventional RAM. The advantages of this method are:
  - We have the address of the word that associates.
  - We can read a data word out of the data part with an address, without using the associative part.

The model that we will use as model for the CAM, is based on the software model developed by ir. A.C. Verschueren, and is given in [VER90]. The used model has a few minor modifications, but the principle is the same.

The Contents Addressable Memory models a memory type where data words can be addressed by comparing bits in these words with a given reference word. The memory is fully synchronous, all commands are executed at the clock edge following their issuing.

If a memory word matches the given 'match data' word in the bits that are ONE in the 'match mask' word, that memory word is said to be matching. Both 'match data' and 'match mask' are input ports. An output provides us information whether there are no matches, one match or more matches. An address output gives the first matching address. A data output will provide the contents of the first matched memory word (all ZEROes if no match is found). These outputs are synchronous to the clock (figure 4.4.).

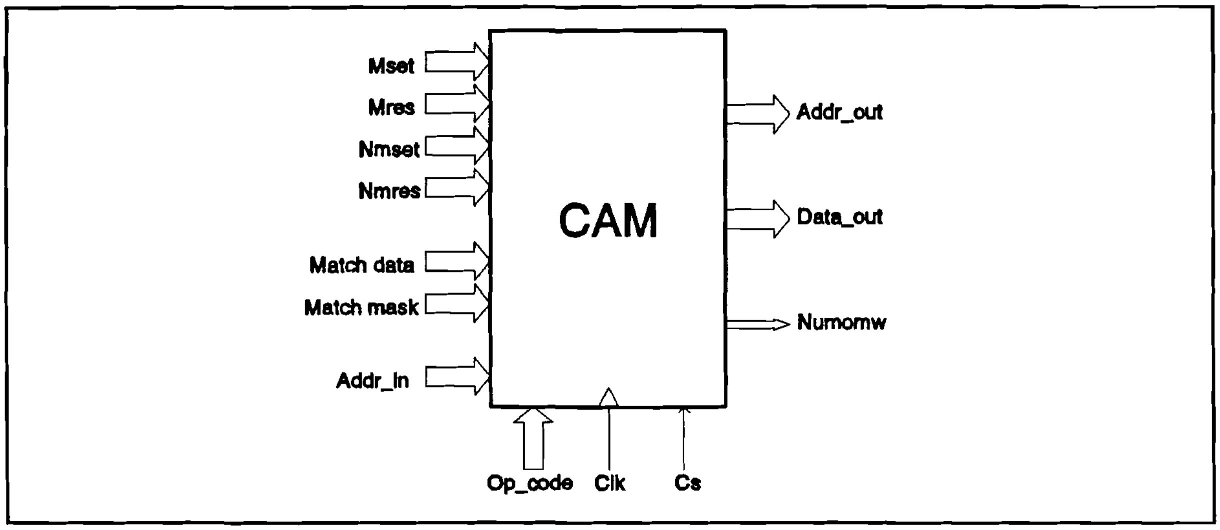


Figure 4.4. *Block structure of the CAM.*

Writing to the memory can be done at the first matching address, or to all matching addresses at the same time. Writing is done with date on the input ports 'mset' and 'mres'. If we want to write a data word 'data' we just have to set mset equal to 'data' and mres equal to non-'data'. The 'write all' function can also modify the non-matching addresses with data from the input ports 'nmset' and 'nmres'.

We use set and reset ports, so that we can modify every memory word by bit. With this mechanism we will be able to change every single bit in the memory, without changing the other data.

Besides this, a standard address input can be added to read and write at addressed locations (synchronous).

It should be mentioned, that the registration of which cells are filled and which not should be done by the user him/herself, because there is no 'full register' in this CAM. This can be done by reserving one bit for each cell as flag, which have to be set as a cell is being filled and reset as a cell is being deleted.

The CAM recognises the following commands (op_code):

reset:

Fills the CAM with the user defined 'synchronous reset' contents at the next clock, then executes the function **'match'** to initialize the output ports. This function overrules all other commands.

match:

Looks for the first memory word that matches with the 'match data' and 'match mask' word.

Input ports: match_data, match_mask.
Output ports: numomw, addr_out, data_out.

This function uses the match mask and match data to check the contents of the CAM. The contents of a cell 'match' when the bits in the cell that correspond with ONE bits in the match mask word equal the corresponding bits in the match data word (all other bits in the cell and the match data word are don't care). The following events occur at the next clock:

- The number of matches is counted and made visible at output port numomw (00 = no matches, 01 = one match, 11 = multiple matches).
- The data-out output will output contents of the first matched cell (with lowest address). If no match is made, this output will carry the value 0.
- The address output will output the address of the first matched cell (The size of the memory if no match is found).

wrfirst:

Writes the first matching word.

Input ports: match_data, match_mask, mset, mres.
Output ports: numomw, addr_out, data_out.

This function behaves like **'match'** and adds the following:

- The contents of the <u>first</u> matched cell are replaced at the next clock using the formula:

  contents := (contents AND (NOT mres)) OR mset

  Thus, setting bit has higher priority than resetting bits. To set an element to a certain data word data_in, we have to make mset equal to data_in and mres equal to all ONEs.

wrall:

Writes all the matching words. This command can change also the non-matching words.

Input ports: match_data, match_mask, mset, mres, nmset, nmres.
Output ports: numomw, addr_out, data_out.

This function behaves like **'match'**, and changes <u>all</u> matched cells like **'wrfirst'** using the formula:

contents := (contents AND (NOT mres)) OR mset

In addition, all cells that do not match are replaced at the next clock using the formula:

contents := (contents AND (NOT nmres)) OR nmset

If we don't want to change the cells that do not match, we have to make nmset and nmres equal to all ZEROes.

rdaddr:

Synchronous read command.

Input ports: addr_in.
Output ports: numomw, data_out, addr_out.

This function lets the following events occur at the next clock:

- The 'number of matches found' numomw is set to '01' if the address is in range. Numomw is set to '00' if the address is out of range.
- The output data_out will output the cell contents for the addressed cell. If the address lies outside the CAM addressing range, this output will carry the value 0.
- The address output will output the input address.

wraddr:

Synchronous write command.

Input ports: addr_in, mset, mres.
Output ports: numomw, data_out, addr_out.

- The 'number of matches found' numomw is set to '01' if the address is in range. Numomw is set to '00' if the address is out of range.
- The contents of the addressed cell will be replaced by the data from the 'mset' and 'mres' input ports using the formula:

  contents := (contents AND (NOT mres)) OR mset

- The output data_out will output the cell contents for the addressed cell. If the address lies outside the CAM addressing range, this output will carry the value 0.
- The address output will output the input address.

When we execute a **'match'** operation, it is possible that there is a multiple match. If we want all addresses and/or data of these matching cells, we have to mark the cells which match, but are not read yet. Therefore we also need to use one bit per cell. Marking the cells that match can be done by executing a **'wrall'** command, with for all bits of nmset and mres all ZEROes, and for nmres and mset all ZEROes, except the match mark bit, which should be ONE.

4.2.1 CAM applications

A lot of research is done on possible application of a CAM. We will give some possible applications, but don't discuss them in detail. A survey of these applications and a detailed description is given by [LOO75].

We can divide the CAM applications in a few classes. That are:

Sorting
- Sorting a CAM (ascending or descending).

Searching
- Searching for a particular word
- Searching for the smallest word.
- Searching for the biggest word.
- Searching for a word that precedes to the key word.
- Searching for a word that follows on the key word.
- Searching for all words smaller than the key word.
- Searching for all words greater than the key word.
- Searching for all words between two key words.
- Searching for the word with the least Hamming distance to the key.

Matrix computations (only useful for sparse matrices)
- Adding matrices.
- Multiplying matrices.
- etc.

Graph algorithms
- Single source shortest path problem.
- Minimal spanning tree.
- etc.

Dynamic stack allocation (only useful with multiple stacks)

It is possible that some applications need extra storage, for example a RAM.

5. Hardware implementations in VHDL

In this chapter we give a short overview of possible hardware implementations of the data structures. We will not give a detailed description, but only a few possibilities. First, we give a short description of VHDL, a hardware description language, which is used for the description of the hardware models.

5.1. VHDL

Quoting [LIP90], VHDL is a hardware description language which is standardized by the IEEE in 1977 and is known as IEEE-1076. VHDL is technology independent, is not tied to a particular simulator or value set, and does not enforce a design methodology on a designer. It allows the designer to choose technologies and methodologies while remaining in a single language.

VHDL supports behavioral description of hardware from the digital system level to the gate level. One of the primary advantages of VHDL lies in the ability to capture the operations of a digital system on a number of these descriptive levels at once, using a coherent syntax and semantics across these levels, and simulate that system using any mixture of those levels of description. It is therefore possible to simulate designs that mix high-level behavioral descriptions of some subsystems with detailed implementations of other subsystems in the model.

With VHDL, it is possible to have one behavioral description, with many structural (low level) descriptions. This makes it possible to define several hardware implementations with only one behavioral description. It is also possible to use 'generics', which makes it possible to create 'generic cells', i.e. building blocks of which the size, delay etc. at can be defined at the moment we want to use a block.

These generic cells, together with the possibility to use several implementations of a certain behavioral description, makes VHDL a good language to describe the data structures. For more information of VHDL see [LIP90].

In appendix A are given the behavioral descriptions of the data structures at the digital system level. The lower level implementations of these behavioral descriptions still have to be described. This can also be done with VHDL, using self-defined building blocks and connections between them.

The VHDL compiler of Mentor Graphics does not support generics, so the declaration of these values is done in a package. Because it was not possible to assign binary values to bit-vectors, we have used integers to test the behaviour.

For every data structure, there are several possibilities for the implementation in hardware. All structures can be implemented using the following block schematic.

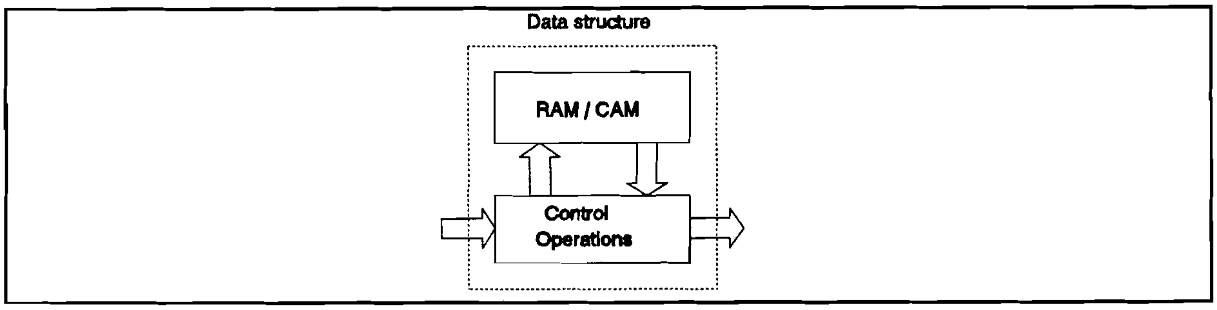


Figure 5.1. *Basic block structure.*

It is possible to optimize some operations of certain data structures for execution speed, using algorithms. For example, it is possible to optimize the search operations in a set by using some search algorithms. This can be implemented in two ways. First, we can add the algorithm to the data structure (figure 5.2.), but then the data structure looses its generality. We can also create an algorithm, that uses the data structure (figure 5.2.(b)). This solution doesn't change the data structure, but it is not that fast as the first solution.

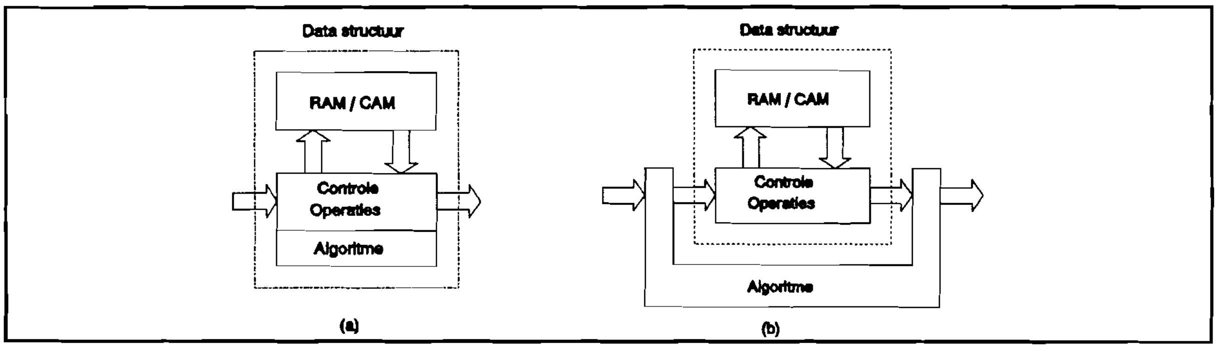


Figure 5.2. *Data structures with algorithms.*

We will give for every data structure the behavioral descriptions (appendix A), with possibilities for the lower level implementations. It is possible that the definitions of the signals of some blocks must change, for an easy implementation on lower levels, or for an easy use on higher levels.

All data structures are described using generics, so it is possible to change the size and the delay of the data structure.

5.2. Array

The block schematic of an array is given in figure 5.3.

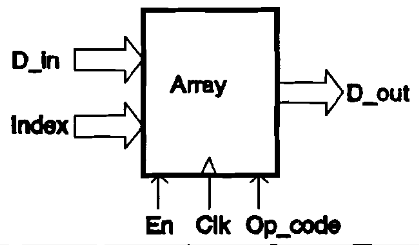


Figure 5.3. *Block schematic of an array.*

The array is a one dimensional array with simple elements. The defined operations are: retrieve and update. The array is a synchronous array, i.e. the operations are executed on the first clock transition from '0' to '1' and only if the enable input EN is '1'. The operations are executed as defined in section 2.1. The VHDL behavioral description is given in appendix A.

An array can be mapped on a RAM, by converting the indices to addresses (subtracting 1 from the index).

5.3. Record

The block schematic of the record structure is given in figure 5.4.

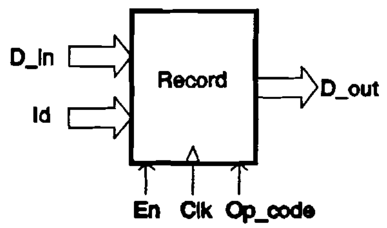


Figure 5.4. *Block schematic of the record.*

The record is a one dimensional record with simple elements. The defined operations are: retrieve and update. The record is synchronous, i.e. the operations are executed on the first clock transition from '0' to '1' and only if the enable input EN is '1'. The operations defined on a record are given in section 2.2. The VHDL description is given in appendix A.

If the identifiers of a record are integers from 1 to n, we can map a record simply on a RAM, but if the identifiers are random, we will have to use a conversion table with the addresses of the elements. This table can be implemented with a RAM and a CAM. It is also possible to implement the record totally with a CAM. The CAM implementations are faster, but the complexity is much higher.

5.4. Linked list

The block schematic of the linked list structure is given in figure 5.5.

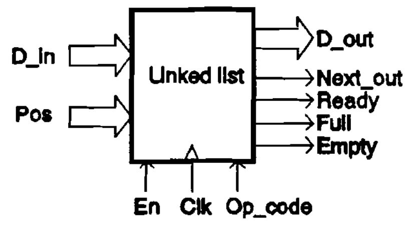


Figure 5.5. *Block schematic of the linked list.*

This is a linked list with simple elements. The elements are identified by their position. The operations which are defined are: clear, insert, delete, retrieve and next. The linked list is synchronous, i.e. the operations are executed on the first clock transition from '0' to '1' and only if the enable input EN is '1'. At the beginning of an operation, the output line READY goes to '0', and becomes '1' again after finishing the operation. The output line NEXT_OUT gives a '1' if there is a next position in the linked list. The lines FULL and EMPTY give the status of the linked list. The exact definitions of the commands are given in section 2.3. The VHDL description of this data structure is given in appendix A.

The linked list can be mapped on a ram, using the cursor representation of figure 3.2. It is also possible to map the linked list on a CAM, using per element a position number. Locating the position in the CAM list is done in one clock cycle, but if we insert an element in or delete an element from the list, we have to update the position numbers of the elements following the inserted/deleted element. So, if we have random insertions and deletions, the CAM is not much faster, only more complex.

5.5. List

The block schematic of a list structure is given in figure 5.6.

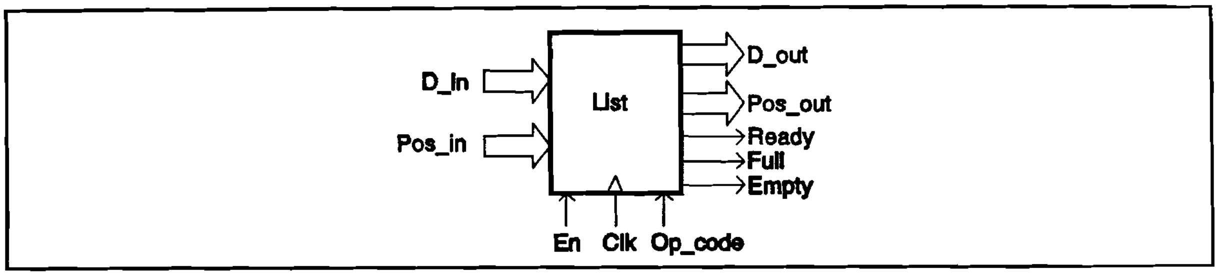


Figure 5.6. *Block schematic of the list.*

This is a list with simple elements, and the elements are identified by their position. The defined operations are: clear, insert, delete, retrieve, locate and end. The list is synchronous, i.e. the operations are executed on the first clock transition from '0' to '1' and only if the enable input EN is '1'. At the beginning of an operation, the output line READY goes to '0', and becomes '1' again after finishing its operation. The lines FULL and EMPTY give the status of the list. The exact definitions of the commands are given in section 2.4. The VHDL description of this data structure is given in appendix A.

The array and the linked list representations of the list can be mapped on a RAM. As mentioned, it is also possible to use a CAM implementation for the linked list. This might be recommendable if we have to sort or search for elements. The CAM implementation is much faster than the usual RAM implementations for sort and search operations (if we use some sort or search algorithms for a CAM).

5.5.1. Stack

The block schematic for a stack is given in figure 5.7.

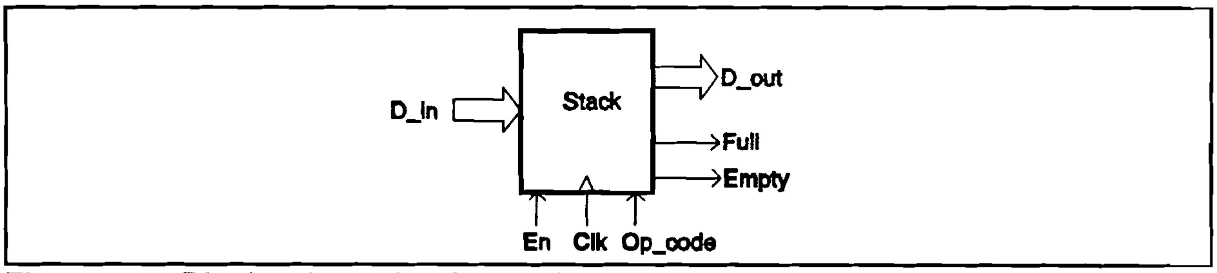


Figure 5.7. *Block schematic of a stack.*

The stack is a stack with simple elements. The defined operations are: clear, push, pop and top. The stack is synchronous, i.e. the operations are executed on the first clock transition from '0' to '1' and only if the enable input EN is '1'. All operations are executed in one clock cycle. The lines FULL and EMPTY give the status of the stack.

The exact definitions of the commands are given in section 2.4.1. The VHDL description of this data structure is given in appendix A.

The best implementation for a stack is a mapping to a RAM.

5.5.2. Queue

The block schematic of a queue is given in figure 5.8.

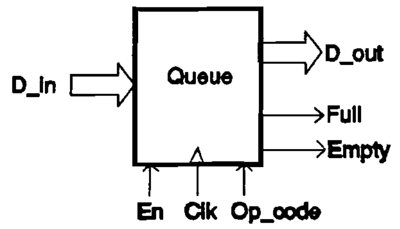


Figure 5.8. *Block schematic of a queue.*

The queue is a queue with simple elements. The operations which are defined are: clear, enqueue, dequeue and front. The queue is synchronous, i.e. the operations are executed on the first clock transition from '0' to '1' and only if the enable input EN is '1'. All operations are executed in one clock cycle. The lines FULL and EMPTY give the status of the queue. The exact definitions of the commands are given in section 2.4.2. The VHDL description of this data structure is given in appendix A.

The best implementation of a queue is a mapping to a RAM.

5.6. Table

The block schematic of a table is given in figure 5.9.

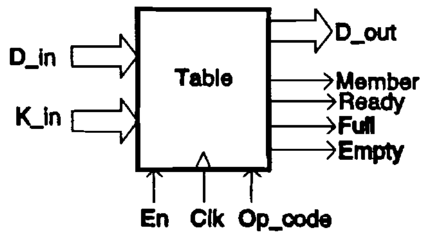


Figure 5.9. *Block schematic of a table.*

The table is a table with simple elements. The defined operations are: clear, insert, delete, retrieve and member. For the behavioral description, an unordered array is used to represent the table. The table is synchronous, i.e. the operations are executed on the first clock transition from '0' to '1' and only if the enable input EN is '1'. At the beginning of an operation, the output line READY goes to '0', and becomes '1' again after finishing the operation. The line MEMBER becomes high if the key K_IN is an element of the table. The lines FULL and EMPTY give the status of the table. The exact definitions of the commands are given in section 2.5. The VHDL description of this data structure is given in appendix A.

We can map the table structure to a RAM, but this gives us a slow implementation, The hash table is easy to map to a CAM, because they have about the same operations. This is a very fast implementation, but the complexity is much higher.

5.7. Tree

The block schematic of the tree structure is given in figure 5.10.

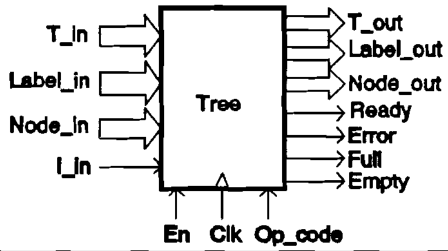


Figure 5.10. *Block schematic of the tree structure.*

The tree structure is a **set** of trees with simple elements. The defined operations are: clear, parent, left_child, right_sibling, label, create and root. For the VHDL description, the left_child-right_sibling representation is used with for every node an entry for the parent, the left_child, the right_sibling, the node_label and the node_number. This is stored in separate arrays, linked together by cursors.

The tree is synchronous, i.e. the operations are executed on the first clock transition from '0' to '1' and only if the enable input EN is '1'. At the beginning of an operation, the output line READY goes to '0', and becomes '1' again after finishing the operation. The lines FULL and EMPTY give the status of the tree. With the operation CREATE, the tree names must be given serially on the tree input port. The exact definitions of the commands are given in section 2.6. The VHDL description of this data structure is given in appendix A.

It is possible to map a tree on a RAM, using the cursor implementation for the linked list. It is also possible to map the tree to a CAM. In each CAM word we have to store the node number, its label, its parent and its right sibling. This implementation is not much faster than a good ram implementation.

5.8. Set

The block schematic of the set structure is given in figure 5.11.

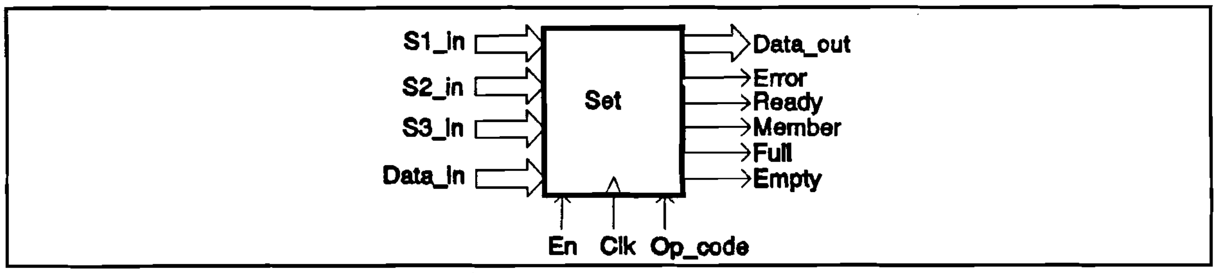


Figure 5.11. *Block schematic of the set structure.*

The set structure is a set of sets, stored in an array with for each set a reserved part. The sets are stored in order, so it will be easy to find the minimum and maximum element of the set. The defined operations are: clear, union, intersection, difference, member, insert, delete, assign, equal, min and max.

The set structure is synchronous, i.e. the operations are executed on the first clock transition from '0' to '1' and only if the enable input EN is '1'. At the beginning of an operation, the output line READY goes to '0', and becomes '1' again after finishing its operation. The lines FULL and EMPTY give the status of the set. The exact definitions of the commands are given in section 2.7. The VHDL description of this data structure is given in appendix A.

All the software implementations mentioned in chapter 3.7. can be mapped to a RAM. If we use a CAM (as a table), we can also implement all operations. The CAM has to store the set name and the set element. This implementation is much faster as the RAM implementation, but it is also more complex.

5.9. Graph

The block schematic of the graph structure is given in figure 5.12.

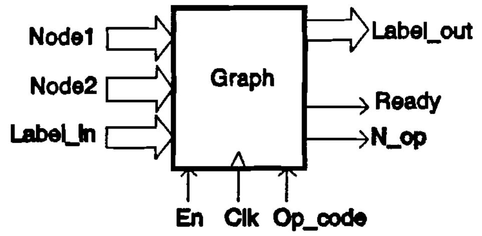


Figure 5.12. *Block schematic of the graph structure.*

The graph structure is a graph with simple elements. The adjacency representation is used for the VHDL description of the graph. The defined operations are: clear, insert_node, insert_edge, delete_node, delete_edge, retrieve_node, retrieve_edge and clear.

The graph structure is synchronous, i.e. the operations are executed on the first clock transition from '0' to '1' and only if the enable input EN is '1'. At the beginning of an operation, the output line READY goes to '0', and becomes '1' again after finishing its operation. The line N_OP gives a '1' if an operation can't be executed (out of range, existence of edges etc.). The exact definitions of the commands are given in section 2.8. The VHDL description of this data structure is given in appendix A.

The adjacency matrix and the adjacency list representations are easy to map on a RAM. It is also possible to map the graph on a CAM, by storing in each CAM word the elements node1, node2 and label. The node-label can be stored by using a null node for node2. With this representation, we only use space proportional to the number of edges. Most operations are just as fast as the adjacency matrix representation, only the operation DELETE_NODE is faster. If the matrix is a sparse matrix, we can best use the adjacency list representation, or the CAM representation. For dense matrices we can best use the adjacency matrix representation.

6. Conclusions and recommendations

I have described several data structures, with their software and possible hardware implementations. If we create abstract data types (software) or generic building blocks (hardware) for these data structures, we can use these data structures for the description and solution of problems.

One data structure, the heap structure, is not covered. This structure for dynamic memory allocation is so complex that it was not possible to look at it during my graduation period.

It is possible to map all described data structures to a RAM. These implementations are often not very fast. For the table, the set and some graphs, a faster implementation is a mapping to a CAM, but this memory is more complex than the RAM (size is 3 to 4 times the size of a RAM).

It is possible to add algorithms (like search algorithms) to the data structures, which improves the performance (execution speed), but which increases the size of the hardware implementation.

The description of the hardware implementation of the data structures can be done with VHDL. However, the VHDL compiler of Mentor Graphics does not support generics, so it is not possible to create generic cells. The simulator of Mentor Graphics is also not perfect. An other VHDL tool is welcome.

The behaviour of the data structure is described in VHDL, The structure of the data structure still has to be described. This should be done by someone else.

A good example of the use of data structures would be welcome.

References

[AHO83] Aho A.V., J.E. Hopcroft, and J.D. Ullmann
DATA STRUCTURES AND ALGORITHMS
Addison-Wesley, Reading, MA, 1983.

[COR90] Cormen T.H., C.E. Leiserson, and R.L. Rivest
INTRODUCTION TO ALGORITHMS
MIT Press, Cambridge, 1990

[KOH87] Kohonen T.
CONTENT-ADDRESSABLE MEMORIES, 2nd ed.
Springer-Verlag, Berlin, 1987.

[KRU87] Kruse R.L.
DATA STRUCTURES AND PROGRAM DESIGN, 2nd ed.
Prentice-Hall, Englewood Cliffs, N.J., 1987.

[LIP90] Lipsett R., C. Scheafer, and C. Ussery
VHDL, HARDWARE DESCRIPTION AND DESIGN, 2nd ed.
Kluwer Academic Publishers, Boston, 1990.

[LOO75] Loon J.M. van, G.J.W. van Nunen, and Th.P.C. Stoffele
TOEPASSINGEN VOOR EN EEN MODEL VAN EEN ASSOCIATIEF GEHEUGEN
Eindhoven, Eindhoven University of Technology, Department of Electrical Engineering, Digital Systems Group, 1975.

[MAN89] Manber U.
INTRODUCTION TO ALGORITHMS, A CREATIVE APPROACH
Addison-Wesley, Reading, MA, 1989.

[SED88] Sedgewick R.
ALGORITHMS, 2nd ed.
Addison-Wesley, Reading, MA, 1988.

[STU85] Stubbs D.F., and N. Webre
DATA STRUCTURES WITH ABSTRACT DATA TYPES AND PASCAL
Brook/Cole Publishing Company, Monterey, Cal., 1985.

[WAR90] Ward S.A., and R.H. Halstead Jr.
COMPUTATION STRUCTURES
McGraw-Hill, New York, 1990.

[VER90] Verschueren A.C.
IDASS FOR ULSI, V0.08d
Eindhoven, Eindhoven University of Technology, Department of Electrical Engineering, Digital Systems Group, 1990.

Appendix A

Appendix A contains the VHDL descriptions of the data structures. For the simulation of these structures, buses are replaced by 'integer' lines, because I couldn't assign values to binary buses with the simulator, 'Quicksim'. I have also used packages for the definition of 'generic' signals, which are not supported by the VHDL compiler 'HDL'. Packages are modules with the declaration of types, functions, procedures, blocks etc.

For every data structure is given the VHDL description of the entity, with its behavioral description, and the description of the package. These models are simulated with the simulator 'Quicksim'. The input files have the extension 'do'. The output of the simulations is not given, because the simulator could not write it to a file (if I tried, the simulator went 'down').

In the models of the data structures, all structures are mapped to arrays, because pointers were not supported. Sometimes, the naming may be confusing, because some arrays are named as ...ram.

```
USE std.standard.ALL;
USE std.mentor_base.ALL;      -- This is a mentor graphics pakage.
USE work.array_pkg.ALL;

-- Type declarations and constant declarations are in arrray_pkg. This is a one dimensional array
-- with simple elements. The operations RETRIEVE and UPDATE are executed on the first clock
-- transition from 0 to 1 and only when EN is high. The results are available on the D_OUT port after at
-- most one clock period.
--
-- RETRIEVE:
--      Input:  index
--      Output: d_out
--
--      - The element with index 'index' is send to the output port 'd_out'.
--
-- UPDATE:
--      Input:  index, d_in
--
--      - The element in the array with index 'index' is updated with data from the port 'd_in'.

ENTITY array_1d IS

--   GENERIC (
--             delay             : time;
--             num_of_elements   : positive;
--             element_length    : positive
--           );

   PORT (
          clk     : IN    bit;
          en      : IN    bit;
          op_code : IN    array_mnemonic;
          index   : IN    address;
          d_in    : IN    data;
          d_out   : OUT   data
        );

END array_1d;


ARCHITECTURE behavior1 OF array_1d IS

   TYPE ram_type IS ARRAY (address) of data;

BEGIN

   array_operation:PROCESS(clk)

   VARIABLE ram            : ram_type;

      BEGIN
         IF clk = '1' THEN
            IF en = '1' THEN
               CASE op_code IS
                  WHEN update   => ram(index) := d_in;
                  WHEN retrieve => d_out <= ram(index) AFTER delay;
               END CASE;    -- opcode
            END IF;    -- en = '1'
         END IF;    -- clk = '1'
      END PROCESS array_operation;

   init:PPOCESS

   VARIABLE ram            : ram_type;

      BEGIN
         d_out <= ram(1) AFTER delay;
         wait;
      END PROCESS init;

END behavior1;
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;   -- This is a mentor graphics package.

PACKAGE array_pkg IS

CONSTANT delay            : time    := 2ns;                 -- Generics
CONSTANT num_of_elements  : integer := 4;                   -- Generics
CONSTANT element_length   : integer := 4;                   -- Generics

TYPE array_mnemonic        IS (update, retrieve);
TYPE address               IS RANGE 1 TO num_of_elements;
TYPE data                  IS RANGE 1 TO 2**element_length;
--TYPE data                  IS ARRAY (element_length-1 DOWNTO 0) OF bit;

END array_pkg;

-- This file contains the information, which should be implemented with
-- the GENERIC-clause in the queue entity. A generic has to be initialized
-- when a other entity uses this entity (module).

-- The type DATA should be a bit_vector(element_length-1 DOWNTO 0) of bit.

-- This is an array with elements of the simple type (types which need only
-- one element of memory to store it).
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;                -- This is a mentor graphics pakage.
USE work.record_pkg.ALL;

-- Type declarations and constant declarations are in record_pkg. This is a one dimensional record
-- with simple elements.The operations RETRIEVE and UPDATE are executed on the first clock transition
-- from 0 to 1 and only when EN is high. The operations are ready after at most one clock period.
-- The data (of different types = different lengths) will be stored in the least significant bits.
--
--- RETRIEVE:
--      Input:  id
--      Output: d_out
--
--      - The element with index 'id' is send to the output port 'd_out'.
--
-- UPDATE:
--      Input:  id, d_in
--
--      - The element in the record with identifier 'id' is updated with data from the port 'd_in'


ENTITY record1 IS

--   GENERIC (
--             delay                  : time;
--             num_of_elements        : positive;
--             max_element_length     : positive
--           );

   PORT (
          clk     : IN    bit;
          en      : IN    bit;
          op_code : IN    array_mnemonic;
          id      : IN    address;
          d_in    : IN    data;
          d_out   : OUT   data
        );

END record1;


ARCHITECTURE behavior1 OF record1 IS

   TYPE ram_type IS ARRAY (address) of data;

BEGIN

   record_operation:PROCESS(clk)

   VARIABLE ram             : ram_type;

      BEGIN
         IF clk = '1' THEN
            IF en = '1' THEN
               CASE op_code IS
                  WHEN update   => ram(id) := d_in;
                  WHEN retrieve => d_out <= ram(id) AFTER delay;
               END CASE;    -- opcode
            END IF;    -- en = '1'
         END IF;    -- clk = '1'
      END PROCESS record_operation;

   init:PROCESS

   VARIABLE ram             : ram_type;

      BEGIN
         d_out <= ram(1) AFTER delay;
         wait;
      END PROCESS init;


END behavior1;
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;    -- This is a mentor graphics pakage.

PACKAGE record_pkg IS

CONSTANT delay                : time    := 2ns;              -- Generics
CONSTANT num_of_elements      : integer := 4;                -- Generics
CONSTANT max_element_length   : integer := 4;                -- Generics

TYPE array_mnemonic        IS (update, retrieve);
TYPE address               IS RANGE 1 TO num_of_elements;
TYPE data                  IS RANGE 1 to 2**max_element_length;
--TYPE data                  IS ARRAY (max_element_length-1 DOWNTO 0) OF bit;

END record_pkg;

-- This file contains the information, which should be implemented with
-- the GENERIC-clause in the queue entity. A generic has to be initialized
-- when an other entity uses this entity (module).

-- The type DATA should be a bit_vector(element_length-1 DOWNTO 0) of bit.

-- This is an record with elements of the simple type (types which need only
-- one element of memory to store it). The size of the record elements is
-- maximal length of the types of the record. We store the elements
-- beginning at the least significant bit of the record memory.
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;          -- This is a Mentor Graphics package.
USE work.linked_list_pkg.ALL;

-- Type declarations and constant declarations are in linked_list_pkg. This is a linked list,
-- where the elements are identified by their position. The operations are started at the first clock
-- transition from '0' to '1', and only if the enable line EN is high. An operation is finished after
-- the READY line becomes high. The line EMPTY is '1' if the list is empty, the line FULL is '1' when
-- the list is full. The line NEXT_OUT is high, when there is a position following the current position
-- (by a retrieve and nxt operation).
--
--
-- CLEAR:
--
--      - This operation makes the linked-list an empty linked-list.
--
-- INSERT:
--      Input: pos, d_in
--
--      - This operation inserst an element on position 'pos' in the list. If position 'pos' does
--        not exist, then nothing will happen.
--
-- DELETE:
--      Input: pos
--      Output: d_out
--
--      - This operation will output the data on position 'pos' to output port 'd_out', and then
--        delete the element on position 'pos'. If position 'pos' does not exist, then nothing
--        will happen.
--
-- RETRIEVE:
--      Input: pos
--      Output: d_out
--
--      - This operation will output the data on position 'pos' to output port 'd_out'. If position
--        'pos' does not exist, then nothing will happen.
--
-- NEXT:
--      Input : pos
--      Output : next_out
--
--      - This operation will output 'true' to port 'next_out' if there is a next position to position
--        'pos', else 'false'.
--


ENTITY linked_list IS

--   GENERIC (
--             delay            : time;
--             num_of_elements  : positive;
--             elementlength    : positive
--           );

   PORT (
          clk      : IN  bit;                  -- Clock
          en       : IN  bit;                  -- Enable input
          op_code  : IN  linked_list_mnemonic; -- Operation code
          d_in     : IN  data;                 -- Data input
          d_out    : OUT data;                 -- Data output
          pos      : IN  address;              -- Position input
          next_out : OUT bit;                  -- Is there a next position?
          ready    : OUT bit;                  -- Is operation ready?
          full     : OUT bit;                  -- Is list full?
          empty    : OUT bit                   -- Is list empty?
        );

END linked_list;


ARCHITECTURE behavior1 OF linked_list IS

  TYPE ram_type IS ARRAY (address) of data;
```

```
BEGIN

  list_operation:PROCESS(clk)

    VARIABLE full_var       : boolean := false;
    VARIABLE empty_var      : boolean := true;
    VARIABLE eol_pos        : address := 0;      -- Last position in list
    VARIABLE eol_pos_var    : integer := 0;
    VARIABLE ram            : ram_type;
    VARIABLE i              : address := 0;

    BEGIN

      IF clk = '1' THEN
        IF en = '1' THEN

          CASE op_code IS

          WHEN clear    => eol_pos := 0;            -- This operation makes the list
                           eol_pos_var := 0;        -- an empty list.
                           empty_var := true;
                           empty <= '1' AFTER delay;
                           full_var := false;
                           full <= '0' AFTER delay;
                           next_out <= '0' AFTER delay;

          WHEN insert   => IF full_var = false THEN                -- List not full?
                             IF pos <= eol_pos THEN                 -- 'Pos'<last pos.
                               FOR i IN eol_pos DOWNTO pos LOOP     -- Move elements
                                 ram(i + 1) := ram(i);               -- up.
                               END LOOP;
                               ram(pos) := d_in;                       -- Insert data.
                               eol_pos := eol_pos + 1;
                               eol_pos_var := eol_pos_var + 1;

                             ELSIF pos = eol_pos + 1 THEN            -- Pos = last pos.
                               ram(pos) := d_in;                        -- Insert data.
                               eol_pos := eol_pos + 1;
                               eol_pos_var := eol_pos_var + 1;

                             END IF;
                             IF pos <= eol_pos THEN                 -- Element inserted.
                               empty <= '0' AFTER delay;
                               empty_var := false;
                               IF eol_pos_var = num_of_elements THEN -- List full?
                                 full_var := true;                    -- Yes, change
                                 full <= '1' AFTER delay;             -- full.
                               END IF;
                             END IF;
                           END IF;

          WHEN delete   => IF empty_var = false THEN              -- List empty?
                             IF pos <= eol_pos THEN             -- Pos. in list.
                               d_out <= ram(pos) AFTER delay;     -- Retrieve data.
                               FOR i IN pos+1  TO eol_pos LOOP    -- Reposition elem.
                                 ram(i - 1) := ram(i);
                               END LOOP;
                               eol_pos := eol_pos - 1;            -- Decrease last pos.
                               eol_pos_var := eol_pos_var -1;
                               full_var := false;
                               full <= '0' AFTER delay;
                               IF eol_pos = 0 THEN               -- List empty?
                                 empty_var := true;
                                 empty <= '1' AFTER delay;
                               END IF;
                             END IF;
                           END IF;

          WHEN retrieve => IF empty_var = false THEN              -- List not empty.
                             IF full_var = true THEN               -- List full
                               d_out <= ram(pos) AFTER delay;
                             ELSIF pos <= eol_pos THEN             -- or pos < last pos.
                               d_out <= ram(pos) AFTER delay;
```

```
                              END IF;
                              IF pos < eol_pos THEN
                                 next_out <= '1' AFTER delay;
                              END IF;
                            END IF;

          WHEN nxt      => IF pos < eol_pos THEN           -- Is there a next position?
                             next_out <= '1' AFTER delay;
                           END IF;

        END CASE;    -- opcode
        ready <= TRANSPORT '1' AFTER delay;

      END IF;    -- en = '1'
    END IF;    -- clk = '1'

  END PROCESS list_operation;

  reset_signals:PROCESS(clk)

    BEGIN
      IF clk = '1' THEN
        ready <= '0';
        next_out <= '0';
      END IF;
    END PROCESS reset_signals;



  init:PROCESS

    BEGIN
      full <= '0' AFTER delay;
      empty <= '1' AFTER delay;
      next_out <= '0' AFTER delay;
      wait;
    END PROCESS init;

END behavior1;
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;      -- This is a mentor graphics package.

PACKAGE linked_list_pkg IS

CONSTANT delay            : time    := 2ns;
CONSTANT num_of_elements  : integer :=  4;
CONSTANT elementlength    : integer :=  4;

TYPE linked_list_mnemonic IS (clear,insert,retrieve,delete,nxt);
TYPE data                 IS RANGE 0 to 2**elementlength - 1;
-- TYPE data                 IS ARRAY (elementlength -1 DOWNTO 0) of bit;
TYPE address              IS RANGE 0 TO num_of_elements;

END linked_list_pkg;

-- This file contains the information, which should be implemented with the
-- GENERIC-clause in the list entity. Because we implement the linked-list
-- in hardware, have have to give an upperbound for the length of the list.

-- nxt stands for the operation next, which is a reserved word.

-- The type DATA should be a bit_vector(wordlength-1 DOWNTO 0).

-- This is a linked-list with elements of the simple type (types which need
-- only one element of memory to store it).
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;          -- This is a Mentor Graphics package.
USE work.list_pkg.ALL;

-- Type declarations and constant declarations are in list_pkg. This is a list, where the elements are
-- identified by their position. The operations are started at the first clock transition from '0' to '1',
-- and only if the enable line EN is high. An operation is finished after  the READY line becomes high.
-- The line EMPTY is '1' if the list is empty, the line FULL is '1' when the list is full.
--
--
-- CLEAR:
--
--      - This operation makes the list an empty list.
--
-- INSERT:
--      Input: pos_in, d_in
--
--      - This operation inserts an element on position 'pos' in the list. If position 'pos' does
--        not exist, then nothing will happen.
--
-- DELETE:
--      Input: pos_in
--      Output: d_out
--
--      - This operation will output the data on position 'pos' to output port 'd_out', and then
--        delete the element on position 'pos'. If position 'pos' does not exist, then nothing
--        will happen.
--
-- RETRIEVE:
--      Input: pos_in
--      Output: d_out
--
--      - This operation will output the data on position 'pos' to output port 'd_out'. If position
--        'pos' does not exist, then nothing will happen.
--
-- LOCATE:
--      Input : d_in
--      Output : pos_out
--
--      - This operation will output the first position of element 'd_in' to port 'pos_out'. If there
--         is no element 'd_in' in the list. The first empty position is given.
--
-- END:
--      Output: pos_out
--
--      -This operation will give the position of the last element + 1 to port 'pos_out'.
--
--



ENTITY list IS

--   GENERIC (
--            delay             : time;
--            num_of_elements   : positive;
--            elementlength     : positive
--          );

   PORT (
        clk       : IN    bit;
        en        : IN    bit;
        op_code   : IN    list_mnemonic;
        d_in      : IN    data;
        d_out     : OUT   data;
        pos_in    : IN    address;
        pos_out   : OUT   address;
        ready     : OUT   bit;
        full      : OUT   bit;
        empty     : OUT   bit
      );

END list;
```

```
ARCHITECTURE behavior1 OF list IS

  TYPE ram_type IS ARRAY (address) of data;

BEGIN

  list_operation:PROCESS(clk)

    VARIABLE full_var        : boolean := false;
    VARIABLE empty_var       : boolean := true;
    VARIABLE located         : boolean := false; -- Indicates whether an element is found
    VARIABLE eol_addr        : address := 1;     -- First empty position
    VARIABLE eol_addr_var    : integer := 1;     -- Same, but then an internal variable.
    VARIABLE ram             : ram_type;
    VARIABLE i               : integer := 0;

    BEGIN

      IF clk = '1' THEN
        IF en = '1' THEN
          ready <= '0';
          CASE op_code IS

          WHEN clear     => eol_addr := 1;                     -- This operation makes the list
                            eol_addr_var := 1;                 -- an empty list, and returns the
                            empty_var := true;                 -- first empty position (eol_addr).
                            empty <= '1' AFTER delay;
                            full_var := false;
                            full <= '0' AFTER delay;
                            pos_out <= eol_addr AFTER delay;

          WHEN insert    => IF full_var = false THEN                 -- List not full?
                              IF pos_in < eol_addr THEN              -- 'Pos'<last pos.
                                FOR i IN eol_addr-1 DOWNTO pos_in LOOP  -- Move elements up
                                  ram(i + 1) := ram(i);
                                END LOOP;
                                ram(pos_in) := d_in;                 -- Insert data.
                              ELSIF pos_in = eol_addr THEN           -- Pos = last pos.
                                ram(pos_in) := d_in;                 -- Insert data.
                              END IF;
                              IF pos_in <= eol_addr THEN             -- Element inserted.
                                empty <= '0' AFTER delay;
                                empty_var := false;
                                IF eol_addr_var = num_of_elements THEN  -- List full?
                                  full_var := true;                  -- Yes, change
                                  full <= '1' AFTER delay;           -- full.
                                ELSE
                                  eol_addr := eol_addr + 1;          -- No, incr.
                                  eol_addr_var := eol_addr_var + 1;  -- eol_addr.
                                END IF;
                              END IF;
                            END IF;

          WHEN delete    => IF empty_var = false THEN              -- List empty?
                              IF full_var = true THEN              -- List full.
                                d_out <= ram(pos_in) AFTER delay;  -- Retrieve data.
                                FOR i IN pos_in + 1 TO eol_addr LOOP  -- Reposition elem.
                                  ram(i - 1) := ram(i);
                                END LOOP;
                                full_var := false;                 -- List not full.
                                full <= '0' AFTER delay;
                              ELSIF pos_in < eol_addr THEN         -- Pos. in list.
                                d_out <= ram(pos_in) AFTER delay;  -- Retrieve data.
                                FOR i IN pos_in+1  TO eol_addr LOOP  -- Reposition elem.
                                  ram(i - 1) := ram(i);
                                END LOOP;
                                eol_addr := eol_addr - 1;          -- Decrease last pos.
                                eol_addr_var := eol_addr_var - 1;
                                IF eol_addr = 1 THEN               -- List empty?
                                  empty_var := true;
                                  empty <= '1' AFTER delay;
                                END IF;
                              END IF;
```

```
                        END IF;

        WHEN retrieve => IF empty_var = false THEN                -- List not empty.
                           IF full_var = true THEN                -- List full
                             d_out <= ram(pos_in) AFTER delay;
                           ELSIF pos_in < eol_addr THEN           -- or pos < last pos.
                             d_out <= ram(pos_in) AFTER delay;
                           END IF;
                         END IF;

        WHEN locate   => IF empty_var = false THEN                -- List not empty.
                           located := false;
                           IF full_var = true THEN                -- List full.
                             FOR i IN 1 TO eol_addr LOOP          -- Search element.
                               IF located = false THEN            -- Element not found.
                                 IF ram(i) = d_in THEN            -- Compare elements.
                                   located := true;
                                   pos_out <= i AFTER delay;      -- First located pos.
                                 END IF;
                               END IF;
                             END LOOP;
                           ELSE                                  -- List not full.
                             FOR i IN 0 TO eol_addr-1 LOOP       -- Search element.
                               IF located = false THEN
                                 IF ram(i) = d_in THEN
                                   located := true;
                                   pos_out <= i AFTER delay;
                                 END IF;
                               END IF;
                             END LOOP;
                           END IF;
                           IF located = false THEN
                             pos_out <= eol_addr AFTER delay;
                           END if;
                         ELSE
                           pos_out <= eol_addr AFTER delay;
                         END IF;

        WHEN f_e_p    => pos_out <= eol_addr;                    -- Give size of list.

      END CASE;    -- opcode
      ready <= TRANSPORT '1' AFTER delay;

    END IF;    -- en = '1'

  END IF;    -- clk = '1'

END PROCESS list_operation;



 init:PROCESS

   BEGIN
     full <= '0';
     empty <= '1';
     ready <= '0';
     pos_out <= 1;
     d_out <= 0;
     wait;
   END PROCESS init;

END behavior1;
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;      -- This is a Mentor Graphics pakage;

PACKAGE list_pkg IS

CONSTANT delay            : time    := 2ns;           -- Generic
CONSTANT num_of_elements  : integer :=  4;            -- Generic
CONSTANT element_length   : integer :=  4;            -- Generic

TYPE list_mnemonic        IS (clear, insert, retrieve, delete, locate, f_e_p);
TYPE data                 IS RANGE 0 TO (2**element_length)-1;
-- TYPE data                 IS ARRAY (element_length - 1 DOWNTO 0);
TYPE address              IS RANGE 1 TO num_of_elements;

END list_pkg;

-- This file contains the information, which should be implemented with the
-- GENERIC-clause in the list entity.

-- The type DATA should be a bit_vector(element_length-1 DOWNTO 0).

-- f_e_p (first empty position) has the same meaning as END (which is a reserved
--   word).
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;   -- This is a Mentor Graphics package.
USE work.stack_pkg.ALL;

-- Type declarations and constant declarations are in stack_pkg. This a LIFO. The operations CLEAR,
-- PUSH , POP nad TOP are executed on the first clock transition from 0 to 1, and only when EN
-- is high. The operations are executed in one clock cycle.
--
--
--
-- CLEAR:
--
--      - This operation makes the stack an empty stack.
--
-- PUSH:
--      Input: d_in
--
--      - This operation inserts an element 'd_in' on the top of the stack.
--
-- POP:
--      Output: d_out
--
--      - This operation will output the data on the top of the stack to port 'd_out' and then deletes
--        the element from the stack. The top of the stack is the element last inserted.
--
-- TOP:
--      Output: d_out
--
--      - This operation will output the data on the top of the stack to output port 'd_out'.
--




ENTITY stack IS

--   GENERIC (
--            delay      : time;
--            wordcount  : positive;
--            wordlength : positive
--           );

   PORT (
          clk     : IN    bit;
          en      : IN    bit;
          op_code : IN    stack_mnemonic;
          d_in    : IN    data;
          d_out   : OUT   data;
          full    : OUT   bit;
          empty   : OUT   bit
        );

END stack;



ARCHITECTURE behavior1 OF stack IS

   TYPE ram_type IS ARRAY (0 TO wordcount - 1) of data;

BEGIN

   stack_operation:PROCESS(clk)

      VARIABLE full_var       : boolean := false;
      VARIABLE empty_var      : boolean := true;
      VARIABLE stack_addr     : integer := 0;
      VARIABLE ram            : ram_type;

      BEGIN
         IF clk = '1' THEN
            IF en = '1' THEN
            CASE op_code IS

              WHEN clear => stack_addr := 0;
                            empty_var := true;
                            empty <= '1' AFTER delay;
                            full_var := false;
                            full <= '0';

               WHEN push  => IF full_var = false THEN
                               ram(stack_addr) := d_in;
                               empty_var := false;
                               empty <= '0' AFTER delay;
                               IF stack_addr = wordcount-1 THEN
                                  full_var := true;
                                  full <= '1' AFTER delay;
                               ELSE
                                  stack_addr := stack_addr + 1;
                               END IF;
                            END IF;

               WHEN pop   => IF empty_var = false THEN
                               IF full_var = true THEN
                                  d_out <= ram(stack_addr) AFTER delay;
                                  full_var := false;
                                  full<= '0' AFTER delay;
                               ELSE
                                  stack_addr := stack_addr - 1;
                                  d_out <= ram(stack_addr) AFTER delay;
                                  full_var := false;
                                  full <= '0' AFTER delay;
                               END IF;
                               IF stack_addr = 0 THEN
                                  empty_var := true;
                                  empty <= '1';
                               END IF;
                            END IF;

               WHEN top   => IF empty_var = false THEN
                               IF full_var = false THEN
                                  d_out <= ram(stack_addr - 1) AFTER delay;
                               ELSE
                                  d_out <= ram(stack_addr) AFTER delay;
                               END IF;
                            END IF;

            END CASE;    -- opcode

         END IF;     -- en = '1'

      END IF;     -- clk = '1'
   END PROCESS stack_operation;


  init:PROCESS

   BEGIN
      full <= '0';
      empty <= '1';
      wait;
   END PROCESS init;


END behavior1;
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;

PACKAGE stack_pkg IS

CONSTANT delay          : time    := 2ns;        -- Generic
CONSTANT  wordcount     : integer :=  4;         -- Generic
CONSTANT wordlength     : integer :=  4;         -- Generic

TYPE stack_mnemonic     IS (clear, push, pop, top);
TYPE data               IS RANGE 0 TO (2**wordlength)-1;
-- TYPE data               IS ARRAY (wordlength - 1 DOWNTO 0);

END stack_pkg;

-- This file contains the information, which should be implemented with the
-- GENERIC-clause in the stack entity.

-- The type DATA should be a bit_vector(wordlength-1 DOWNTO 0).
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;        -- This is a Mentor Graphics package.
USE work.queue_pkg.ALL;

-- Type declarations and constant declarations are in queue_pkg. This is a FIFO. The operations CLEAR,
-- ENQUEUE, DEQUEUE, and FRONT are executed on the first clock transition from 0 to 1, and only
-- when EN is high. The operations are executed in one clock cycle.
--
--
--
-- CLEAR:
--
--      - This operation makes the queue an empty queue.
--
-- ENQUEUE:
--      Input: d_in
--
--      - This operation inserts an element 'd_in' at the end of the queue.
--
-- DEQUEUE:
--      Output: d_out
--
--      - This operation will output the data at the front of the queue to output port 'd_out'. This is
--        the element that was for the longest time in the queue. (FIFO)
--
-- FRONT:
--      Output: d_out
--
--      - This operation will output the data at the front of the queue to output port 'd_out'.
--




ENTITY queue IS

--   GENERIC (
--            delay      : time;
--            wordcount  : positive;
--            wordlength : positive
--           );

   PORT (
          clk     : IN   bit;
          en      : IN   bit;
          op_code : IN   queue_mnemonic;
          d_in    : IN   data;
          d_out   : OUT  data;
          full    : OUT  bit;
          empty   : OUT  bit
        );

END queue;


ARCHITECTURE behavior1 OF queue IS

   TYPE ram_type IS ARRAY (0 TO wordcount-1) of data;

BEGIN

   queue_operation:PROCESS(clk)

      VARIABLE full_var        : boolean := false;
      VARIABLE empty_var       : boolean := true;
      VARIABLE enque_addr      : integer := 0;
      VARIABLE deque_addr      : integer := 0;
      VARIABLE ram             : ram_type;

      BEGIN
         IF clk = '1' THEN
            IF en = '1' THEN
            CASE op_code IS

               WHEN clear =>   deque_addr := 0;
                               enque_addr := 0;
                               empty_var := true;
                               empty <= '1' AFTER delay;
                               full_var := false;
                               full <= '0' AFTER delay;

               WHEN enqueue => IF full_var = false THEN
                                  ram(enque_addr) := d_in;
                                  enque_addr := (enque_addr + 1) MOD wordcount;
                                  IF enque_addr = deque_addr THEN
                                     full_var := true;
                                     full <= '1' AFTER delay;
                                  END IF;
                                  empty_var := false;
                                  empty <= '0' AFTER delay;
                               END IF;

               WHEN dequeue => IF empty_var = false THEN
                                  d_out <= ram(deque_addr) AFTER delay;
                                  deque_addr := (deque_addr + 1) MOD wordcount;
                                  IF deque_addr = enque_addr THEN
                                     empty_var := true;
                                     empty <= '1' AFTER delay;
                                  END IF;
                                  full_var := false;
                                  full <= '0' AFTER delay;
                               END IF;

               WHEN front   => IF empty_var = false THEN
                                  d_out <= ram(deque_addr) AFTER delay;
                               END IF;

            END CASE;    -- opcode

         END IF;    -- en = '1'
      END IF;    -- clk = '1'
   END PROCESS queue_operation;


  init:PROCESS

   BEGIN
      full <= '0';
      empty <= '1';
      wait;
   END PROCESS init;

END behavior1;
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;

PACKAGE queue_pkg IS

CONSTANT delay          : time    := 2ns;           -- Generic
CONSTANT wordcount      : integer :=  4;            -- Generic
CONSTANT wordlength     : integer :=  4;            -- Generic

TYPE queue_mnemonic     IS (clear, enqueue, dequeue, front);
TYPE data               IS RANGE 0 TO (2**wordlength)-1;
-- TYPE data               IS ARRAY (wordlength -1 DOWNTO 0);

END queue_pkg;

-- This file contains the information, which should be implemented with the
-- GENERIC-clause in the queue entity.

-- The type DATA should be a bit_vector(wordlength-1 DOWNTO 0).
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;    -- This is a Mentor Graphics package.
USE work.table_pkg.ALL;

-- Type declarations and constant declarations are in table_pkg. The operations CLEAR, INSERT, DELETE,
-- RETRIEVE and MEMBER are executed on the first clock transition from 0 to 1, and only when EN is high.
-- The output line READY is high at the next clock transition after finishing an operation. The output
-- line MEMBER is high when an element is in the table (after an INSERT, you can see whether an element
-- is inserted or not)


-- CLEAR:
--
--      - This operation makes the table an empty table.
--
-- INSERT:
--      Input: k_in, d_in
--
--      - This operation inserts an element with key 'k_in' and data 'd_in' in the table.
--
-- DELETE:
--      Input: k_in
--      Output: d_out
--
--      - This operation will output the data with key 'k_in' to output port 'd_out', and then
--        delete the element with key 'k_in'. If the key 'k_in' is not in the table, then nothing
--        will happen.
--
-- RETRIEVE:
--      Input: k_in
--      Output: d_out
--
--      - This operation will output the data with key 'k_in' to output port 'd_out'. If key
--        'k_in' does not exist, then nothing will happen.
--
-- MEMBER:
--      Input : k_in
--      Output : member
--
--      - This operation will output 'true' to output port 'member' if the element with key 'k_in'
--        is in the table, else the output will be 'false'.
--



ENTITY table IS

--   GENERIC (
--             delay                 : time;
--             max_num_of_elements   : positive;
--             key_length            : positive;
--             data_length           : positive
--           );

   PORT (
          clk      : IN    bit;                    -- Clock
          en       : IN    bit;                    -- Enable input
          op_code  : IN    table_mnemonic;         -- Operation code
          k_in     : IN    key;                    -- Key input
          k_out    : OUT   key;                    -- Key output
          d_in     : IN    data;                   -- Data input
          d_out    : OUT   data;                   -- Data output
          member   : OUT   bit;                    -- Is element in table?
          ready    : OUT   bit;                    -- Is operation ready?
          full     : OUT   bit;                    -- Is list full?
          empty    : OUT   bit                     -- Is list empty?
        );

END table;


ARCHITECTURE behavior1 OF table IS

  TYPE key_ram_type  IS ARRAY (0 TO max_num_of_elements - 1) of key;
  TYPE data_ram_type IS ARRAY (0 TO max_num_of_elements - 1) of data;

BEGIN

  list_operation:PROCESS(clk)

    VARIABLE full_var       : boolean := false;
    VARIABLE empty_var      : boolean := true;
    VARIABLE located        : boolean := false;
    VARIABLE eot_pos        : integer := 0;      -- Last position in table
    VARIABLE k_ram          : key_ram_type;      -- Array with keys
    VARIABLE d_ram          : data_ram_type;     -- Array with data
    VARIABLE i              : integer := 0;

    BEGIN

      IF clk = '1' THEN
        member <= '0';

        IF en = '1' THEN
          ready <= '0';

          CASE op_code IS


          WHEN clear    => eot_pos := 0;                      -- This operation makes the
                           empty_var := true;                 -- table empty.
                           empty <= '1' AFTER delay;
                           full_var := false;
                           full <= '0' AFTER delay;
                           member <= '0' AFTER delay;

          WHEN insert   => IF empty_var = true THEN           -- Table empty?
                             k_ram(eot_pos) := k_in;          --  Yes, insert element.
                             d_ram(eot_pos) := d_in;
                             empty_var := false;
                             empty <= '0' AFTER delay;
                             member <= TRANSPORT '1' AFTER delay;
                           ELSE
                             located := false;                -- No, table not empty
                             FOR i IN 0 TO eot_pos LOOP       -- Key already in table?
                               IF located = false THEN
                                 IF k_in = k_ram(i) THEN
                                   located := true;
                                   d_ram(i) := d_in;
                                   member <= TRANSPORT '1' AFTER delay;
                                 END IF;
                               END IF;
                             END LOOP;
                             IF located = false THEN          -- Key not yet in table!
                               IF full_var = false THEN       -- Table not full?
                                 eot_pos := eot_pos + 1;
                                 k_ram(eot_pos) := k_in;
                                 d_ram(eot_pos) := d_in;
                                 member <= TRANSPORT '1' AFTER delay;
                                 IF eot_pos = max_num_of_elements - 1 THEN
                                   full_var := true;
                                   full <= '1' AFTER delay;
                                 END IF;
                               END IF;
                             END IF;
                           END IF;

          WHEN delete   => located := false;
                           IF empty_var = false THEN          -- Table empty?
                             FOR i IN 0 TO eot_pos LOOP
                               IF located = false THEN
                                 IF k_in = k_ram(i) THEN      -- Search for element
                                   k_out <= k_ram(i) AFTER delay;
                                   d_out <= d_ram(i) AFTER delay;
                                   located := true;           -- Move last element to position.
                                   k_ram(i) := k_ram(eot_pos);  --  of deleted element
                                   d_ram(i) := d_ram(eot_pos);
                                   member <= TRANSPORT '1' AFTER delay;
```

```
                                END IF;
                              END IF;
                            END LOOP;
                          END IF;
                          IF located = true THEN                  -- Element deleted?
                            IF eot_pos /= 0 THEN
                              eot_pos := eot_pos - 1;
                              full_var := false;
                              full <= '0' AFTER delay;
                            ELSE
                              empty_var := true;
                              empty <= '1' AFTER delay;
                            END IF;
                          END IF;

        WHEN retrieve => IF empty_var = false THEN               -- Table not empty.
                            located := false;
                            FOR i IN 0 TO eot_pos LOOP            -- Search for element
                              IF located = false THEN
                                IF k_in = k_ram(i) THEN
                                  k_out <= k_ram(i) AFTER delay;
                                  d_out <= d_ram(i) AFTER delay;
                                  located := true;
                                  Member <= TRANSPORT '1' AFTER delay;
                                END IF;
                              END IF;
                            END LOOP;
                          END IF;

        WHEN memb     => IF empty_var = false THEN               -- Element in table?
                            located := false;                     -- Same as retrieve.
                            FOR i IN 0 TO eot_pos LOOP
                              IF located = false THEN
                                IF k_in = k_ram(i) THEN
                                  k_out <= k_ram(i) AFTER delay;
                                  d_out <= d_ram(i) AFTER delay;
                                  located := true;
                                  member <= TRANSPORT '1' AFTER delay;
                                END IF;
                              END IF;
                            END LOOP;
                          END IF;

      END CASE;    -- opcode
      ready <= TRANSPORT '1' AFTER delay;

    END IF;    -- en = '1'
  END IF;    -- clk = '1'

END PROCESS list_operation;


init:PROCESS

  BEGIN
    full <= '0';
    empty <= '1';
    member <= '0';
    ready <= '0';
    wait;
  END PROCESS init;

END behavior1;
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;      -- This is a mentor graphics package.


PACKAGE table_pkg IS

CONSTANT delay                  : time    := 2ns;       -- Generic
CONSTANT max_num_of_elements    : integer :=  4;        -- Generic
CONSTANT key_length             : integer :=  4;        -- Generic
CONSTANT data_length            : integer :=  4;        -- Generic

TYPE table_mnemonic         IS (clear, insert, delete, retrieve, memb);
TYPE key                    IS RANGE 0 TO 2**key_length - 1;
-- TYPE key                    IS ARRAY (key_length - 1 DOWNTO 0) of bit;
TYPE data                   IS RANGE 0 TO 2**data_length - 1;
-- TYPE data                   IS ARRAY (data_length - 1 DOWNTO 0) of bit;

END table_pkg;


-- This file contains the information, which should be implemented with the
-- GENERIC-clause in the table entity.

-- The type KEY  should be a bit_vector(key_length - 1 DOWNTO 0).
-- The type DATA should be a bit_vector(data_length - 1 DOWNTO 0).
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;       -- This is a Mentor Graphics package.
USE work.tree_pkg.ALL;

-- Type declarations and constant declarations are in tree_pkg.
-- The trees are stored in an array, using the left-child, right-sibling representation.
-- This is a tree with the operations CLEAR, PARENT, LEFT_CHILD, RIGHT_SIBLING, RETR_LABEL, CREATE, ROOT
-- and RESET. The operations are started at the first clock transition from 0 to 1, and only when
-- EN is high. An operation is finished when the READY signal becomes high.
-- If an operation is executed incomplete, an ERROR is reported. When we execute
-- the operation CREATE, all tree names have to be different.

-- CLEAR:
--
--      - This operation makes the tree an empty tree.
--
-- PARENT:
--      Input: node_in
--      Output: node_out
--
--      - This operation gives the parent of node 'node_in' to output port 'node_out'. If 'node_in'
--        has no parent, the null node is sent out.
--
-- LEFT_CHILD:
--      Input: node_in
--      Output: node_out
--
--      - This operation gives the left-child of node 'node_in' to output port 'node_out'. If 'node_in'
--        has no left-child, the null node is sent out.
--
-- RIGHT_SIBLING:
--      Input: node_in
--      Output: node_out
--
--      - This operation gives the right-sibling of node 'node_in' to output port 'node_out'. If 'node_in'
--        has no right-sibling, the null node is sent out.
--
-- LABEL:
--      Input : d_in
--      Output : pos_out
--
--      - This operation will output the label of node 'node_in' to port 'label_out'.
--
-- CREATE:
--      Input: i_in, label_in, t_in
--      Output: t_out, node_out, label_out
--
--      - This operation creates trees with root 'node_out', which has label 'label_in' (='label_out'), and
--        which has 'i_in' children. The children are the roots of the trees, 't_in' which are given serially
--        to input ports 't_in'. If 'i_in' is 0, then a single node is created. If 'i_in' is 1, a tree with
--        one child is created, etc. The name of the new tree is given on output port 't_out'.
--
-- ROOT:
--      Input: t_in
--      Output: node_out
--
--      - This operation will give the root of tree 't_in' to output port 'node_out'. If there is no tree
--        't_in' then the null node is sent out.
--
--

ENTITY tree IS

--   GENERIC (
--             delay                      : time;
--             num_of_sets                : positive;
--             num_of_elements_per_set    : positive;
--             elementlength              : positive;
--             nul_node                   : ?
--             nul_label                  : ?
--           );

   PORT (
        clk       : IN    bit;
```

```
          en         : IN    bit;
          op_code    : IN    tree_mnemonic;
          t_in       : IN    tree_name;
          t_out      : OUT   tree_name;
          label_in   : IN    data;
          label_out  : OUT   data;
          node_in    : IN    node_name;
          node_out   : OUT   node_name;
          i_in       : IN    num_of_subtrees;
          error      : OUT   bit;
          ready      : OUT   bit;
          full       : OUT   bit;
          empty      : OUT   bit
        );

END tree;


ARCHITECTURE behavior1 OF tree IS

  TYPE tree_ram            IS ARRAY (1 TO max_num_of_trees) OF node_name;
  TYPE left_child_ram      IS ARRAY (1 TO max_num_of_nodes) OF node_name;
  TYPE parent_ram          IS ARRAY (1 TO max_num_of_nodes) OF node_name;
  TYPE label_ram           IS ARRAY (1 TO max_num_of_nodes) OF data;
  TYPE right_sibling_ram   IS ARRAY (1 TO max_num_of_nodes) OF node_name;
  TYPE used_ram            IS ARRAY (0 TO max_num_of_nodes) OF bit;

BEGIN

tree_operation:PROCESS

  VARIABLE tree_list             : tree_ram;                    -- Array with root of trees
  VARIABLE left_ch               : left_child_ram;              -- Array with index of left_child
  VARIABLE parent_n              : parent_ram;                  -- Array with parent of node
  VARIABLE node_label            : label_ram;                   -- Array with label of node
  VARIABLE right_s               : right_sibling_ram;           -- Array with right-sibling of node
  VARIABLE node_used             : used_ram;                    -- Array with bit which indicates whether
                                                                --  node is used or not.

  VARIABLE available_nodes       : node_name;                   -- First node of list of empty nodes, linked
                                                                --  by cursors in the left_child array.
  VARIABLE node                  : node_name;                   --
  VARIABLE temp_root             : node_name;                   --
  VARIABLE num_of_avail_nodes    : integer := max_num_of_nodes; -- Number of available nodes
  VARIABLE num_of_avail_trees    : integer := max_num_of_trees; -- Number of available tree entries.
  VARIABLE i                     : integer := 0;                --
  VARIABLE j                     : integer := 0;
  VARIABLE ready_var             : boolean := false;
  VARIABLE located               : boolean := false;

  BEGIN

    WAIT ON clk;
    IF clk = '1' THEN
      error <= '0';

      IF en = '1' THEN
        ready <= '0';

        CASE op_code IS

        WHEN clear_tree    => t_out <= t_in AFTER delay;
                              node_out <= null_node AFTER delay;
                              label_out <= null_label AFTER delay;
                              IF tree_list(t_in) /= null_node THEN          -- Is  tree empty?
                                node := tree_list(t_in);
                                ready_var := false;
                                WHILE ready_var = false LOOP
                                  WHILE left_ch(node) /= null_node LOOP     -- Go to leaf
                                    node := left_ch(node);
                                  END LOOP;
                                  left_ch(node) := available_nodes;         -- Add deleted leaf_node to
                                  node_used(node) := '0';                   --  empty_list. Node not used
                                  available_nodes := node;                  --  anymore.
                              num_of_avail_nodes := num_of_avail_nodes + 1;
                              IF right_s(node) /= null_node THEN          -- Is there a right neighbour?
                                node := right_s(node) ;
                              ELSIF parent_n(node) /= null_node THEN      -- Is there a parent?
                                node := parent_n(node);
                                left_ch(node) := null_node;
                              ELSE                                        --Ready?
                                node := null_node;
                                ready_var := true;
                              END IF;
                            END LOOP;
                            tree_list(t_in) := null_node;
                            num_of_avail_trees := num_of_avail_trees + 1;  -- Increase num of free trees.
                          END IF;

      WHEN parent        => IF node_used(node_in) = '1' THEN              -- Node used?
                            node_out <= parent_n(node_in) AFTER delay;    -- Give parent or
                          ELSE
                            node_out <= null_node AFTER delay;
                            error <= TRANSPORT '1' AFTER delay;                     -- error
                          END IF;

      WHEN left_child    => IF node_used(node_in) = '1' THEN              -- Node used?
                            node_out <= left_ch(node_in) AFTER delay;     -- Give left child or
                          ELSE
                            node_out <= null_node AFTER delay;
                            error <= TRANSPORT '1' AFTER delay;                     -- error
                          END IF;

      WHEN right_sibling => IF node_used(node_in) = '1' THEN              -- Node used?
                            node_out <= right_s(node_in) AFTER delay;     -- Give right_sibling or
                          ELSE
                            node_out <= null_node AFTER delay;
                            error <= TRANSPORT '1' AFTER delay;                     -- error
                          END IF;

      WHEN retr_label    => IF node_used(node_in) = '1' THEN              -- Node used?
                            label_out <= node_label(node_in) AFTER delay; -- Give label or
                          ELSE
                            label_out <= null_label AFTER delay;
                            error <= TRANSPORT '1' AFTER delay;                     -- error
                          END IF;

      WHEN create_tree   => IF num_of_avail_nodes /= 0 THEN               -- num of available nodes > 0.
                            j := 1;
                            i := i_in;
                            IF i = 0 THEN                                 -- Create root/leaf tree.
                              IF num_of_avail_trees /= 0 THEN             -- Is there an empty tree?
                                node := available_nodes;                  -- Create node.
                                available_nodes := left_ch(available_nodes);
                                num_of_avail_nodes := num_of_avail_nodes - 1;
                                node_label(node) := label_in;
                                node_used(node) := '1';
                                parent_n(node) := null_node;
                                left_ch(node) := null_node;
                                right_s(node) := null_node;
                                located := false;
                                FOR j IN 1 TO max_num_of_trees LOOP       -- Search for empty tree.
                                  IF located = false THEN
                                    IF tree_list(j) = null_node THEN
                                      located := true;
                                      tree_list(j) := node;               -- Put single node in root of
                                      t_out <= j AFTER delay;             --  tree
                                      node_out <= node AFTER delay;
                                      label_out <= label_in AFTER delay;
                                      num_of_avail_trees := num_of_avail_trees - 1;
                                    END IF;
                                  END IF;
                                END LOOP;
                              ELSE
                                error <= TRANSPORT '1' AFTER delay;
                              END IF;
                            ELSE                                         -- Create tree with one or more
                              node := available_nodes;                   --  children.
```

```
                                      available_nodes := left_ch(available_nodes);
                                      num_of_avail_nodes := num_of_avail_nodes - 1;
                                      t_out <= t_in AFTER delay;
                                      label_out <= label_in AFTER delay;
                                      node_out <= node AFTER delay;
                                      parent_n(node) := null_node;
                                      node_used(node) := '1';
                                      node_label(node) := label_in;
                                      left_ch(node) := tree_list(t_in);
                                      tree_list(t_in) := node;
                                      temp_root := node;
                                      node := left_ch(node);
                                      WHILE j < i LOOP
                                        WAIT ON clk;
                                        IF clk = '1' THEN
                                          IF en = '1' THEN
                                            right_s(node) := tree_list(t_in);
                                            tree_list(t_in) := null_node;
                                            num_of_avail_trees := num_of_avail_trees + 1;
                                            parent_n(node) := temp_root;
                                            node := right_s(node);
                                            j := j + 1;
                                          END IF;
                                        END IF;
                                      END LOOP;
                                      parent_n(node) := temp_root;
                                    END IF;
                                  ELSE
                                    error <= TRANSPORT '1' AFTER delay;
                                  END IF;

      WHEN root          => node_out <= tree_list(t_in) AFTER delay;

      WHEN reset_tree    => available_nodes := null_node;
                            FOR i IN max_num_of_nodes DOWNTO 1 LOOP
                              node_used(i) := '0';
                              left_ch(i) := available_nodes;
                              available_nodes := i;
                            END LOOP;
                            FOR i IN 1 TO max_num_of_trees LOOP
                              tree_list(i) := null_node;
                            END LOOP;
                            num_of_avail_nodes := max_num_of_nodes;
                            num_of_avail_trees := max_num_of_trees;
                            t_out <= t_in AFTER delay;
                            node_out <= null_node AFTER delay;
                            label_out <= null_label AFTER delay;


      END CASE;    -- op_code
      ready <= TRANSPORT '1' AFTER delay;
      IF num_of_avail_nodes = max_num_of_nodes THEN
        empty <= '1' AFTER delay;
        full <= '0' AFTER delay;
      ELSIF num_of_avail_nodes = 0 THEN
        empty <= '0' AFTER delay;
        full <= '1' AFTER delay;
      ELSE
        empty <= '0' AFTER delay;
        full <= '0' AFTER delay;
      END IF;

    END IF;    -- en = '1'

  END IF;    -- clk = '1'

END PROCESS tree_operation;


init:PROCESS

  BEGIN
    label_out <= null_label;
    node_out <= null_node;
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;      -- This is a mentor graphics package.

PACKAGE tree_pkg IS


CONSTANT delay                  : time     := 2ns;
CONSTANT max_num_of_trees       : integer :=  4;
CONSTANT max_num_of_nodes       : integer :=  8;
CONSTANT elementlength          : integer :=  4;



TYPE tree_mnemonic IS (clear_tree,parent,left_child,right_sibling,retr_label,create_tree,root,reset_tree);

TYPE data                       IS RANGE 0 to (2**elementlength) - 1;
-- TYPE data                       IS ARRAY (elementlength -1 DOWNTO 0) of bit;
CONSTANT null_label             : data    :=  0;

SUBTYPE tree_name               IS integer RANGE 1 TO max_num_of_trees;
-- TYPE tree_name                  IS ARRAY (2log max_num_of_trees -1 DOWNTO 0) of bit;

SUBTYPE node_name               IS integer RANGE 0 TO max_num_of_nodes;
CONSTANT null_node              : node_name :=  0;


SUBTYPE num_of_subtrees     IS integer RANGE 0 TO max_num_of_trees;

END tree_pkg;

-- This file contains the information, which should be implemented with the
-- GENERIC-clause in the set entity.

-- The type DATA should be a bit_vector(elementlength-1 DOWNTO 0).
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;
USE work.set_pkg.ALL;

-- Type declarations and constant declarations are in set_pkg. The sets are stored in an array,
-- with for each set a reserved part. The sets are stored in order, so the operations MIN and MAX
-- may be applied. We assume that the sets are numbered from 1 TO num_of_sets. If the naming of
-- the sets is random, we have to use a table to map the names on the base address of the set.

-- This is a set with the normal set operations. The operations are started at the first clock
-- transition from 0 to 1, and only when EN is high. An operation is finished when the READY signal
-- becomes high. If an operation is executed incomplete, an ERROR is reported. When we execute
-- the operations UNION, INTERSECTION and DIFFERENCE, all three set names have to be different.

-- CLEAR:
--
--      - This operation makes the set an empty set.
--
-- UNION:
--      Input: s1_in, s2_in, s3_in
--
--      - This operation takes the sets 's1_in' and 's2_in' and assigns the result of the union
--        of 's1_in' and 's2_in' to the set 's3'in'. 's1_in', 's2_in' and 's3_in' have to be different.
--
-- INTERSECTION:
--      Input: s1_in, s2_in, s3_in
--
--      - This operation takes the sets 's1_in' and 's2_in' and assigns the result of the intersection
--        of 's1_in' and 's2_in' to the set 's3'in'. 's1_in', 's2_in' and 's3_in' have to be different.
--
-- DIFFERENCE:
--      Input: s1_in, s2_in, s3_in
--
--      - This operation takes the sets 's1_in' and 's2_in' and assigns the result of the difference
--        of 's1_in' and 's2_in' to the set 's3'in'. 's1_in', 's2_in' and 's3_in' have to be different.
--
-- MEMBER:
--      Input: s1_in, d_in
--      Output: member
--
--      - This operations will output 'true' to output port 'member' if element 'd_in' is in set 's1_in',
--        else the output becomes 'false'.
--
-- INSERT:
--      Input: s1_in, d_in
--
--      - This operation inserts an element 'd_in' in set 's1_in'.
--
-- DELETE:
--      Input: s_in, d_in
--
--      - This operation deletes element 'd_in' from set 's1_in'. If 'd_in' is not in set 's1_in',
--        nothing will happen.
--
-- ASSIGN:
--      Input: s1_in, s2_in
--
--      - This operation will make set 's1_in' equal to set 's2_in'.
--
-- EQUAL:
--      Input : s1_in, s2_in
--      Output : error
--
--      - This operation will output 'false' to output port 'error' if set 's1_in' is equal to set 's2_in',
--        else 'error' becomes 'true'.
--
-- MIN:
--      Input: s1_in
--      Output: d_out
--
--      - This operation will give the minimum element of set 's1_in' to output port 'd_out'.
--
-- MAX:
--      Input: s1_in
--      Output: d_out
--
--      - This operation will give the maximum element of set 's1_in' to output port 'd_out'.



ENTITY set IS

--   GENERIC (
--            delay                      : time;
--            num_of_sets                : positive;
--            num_of_elements_per_set    : positive;
--            elementlength              : positive
--          );

   PORT (
         clk       : IN    bit;
         en        : IN    bit;
         op_code   : IN    set_mnemonic;
         s1_in     : IN    set_name;
         s2_in     : IN    set_name;
         s3_in     : IN    set_name;
         d_in      : IN    data;
         d_out     : OUT   data;
         error     : OUT   bit;
         member    : OUT   bit;
         ready     : OUT   bit;
         full      : OUT   bit;
         empty     : OUT   bit
        );

END set;


ARCHITECTURE behavior1 OF set IS

  TYPE ram_type IS ARRAY (0 TO ((num_of_el_per_set * num_of_sets)-1) ) of data;
  TYPE ram_elements_per_set IS ARRAY (set_name) OF integer;

BEGIN

  set_operation:PROCESS(clk)

    VARIABLE num_of_el_set       : ram_elements_per_set;          -- Array with elements per set
    VARIABLE located             : boolean := false;              -- Is element in set?
    VARIABLE equal_var           : boolean := false;              -- Are elements equal?
    VARIABLE base_addr_s1        : integer := 0;                  -- base address of set 1
    VARIABLE base_addr_s2        : integer := 0;                  -- base address of set 2
    VARIABLE base_addr_s3        : integer := 0;                  -- base address of set 3
    VARIABLE ram                 : ram_type;                      -- Array for simulating a RAM
    VARIABLE i                   : integer := 0;
    VARIABLE j                   : integer := 0;
    VARIABLE k                   : integer := 0;
    VARIABLE max_num_of_el_p_s   : integer := num_of_el_per_set;

    BEGIN

      IF clk = '1' THEN
        member <= '0';
        error <= '0';

        IF en = '1' THEN
          ready <= '0';

          CASE op_code IS

          WHEN clear      => num_of_el_set(s1_in) := 0;                  -- This operation makes the
                             empty <= '1' AFTER delay;                   -- set an empty set
                             full <= '0' AFTER delay;

          WHEN union      => IF s1_in = s2_in
                             OR s1_in = s3_in
                             OR s2_in = s3_in
                             THEN
```

```
                                  error <= '1' AFTER delay;
                                ELSE
                                  base_addr_s1 := (s1_in - 1) * max_num_of_el_p_s;  -- Set3 is created by merging
                                  base_addr_s2 := (s2_in - 1) * max_num_of_el_p_s;  -- set1 and set2. (picking
                                  base_addr_s3 := (s3_in - 1) * max_num_of_el_p_s;  -- each time the smallest el.
                                  i := 0;                                           -- of set1 or 2, when we walk
                                  j := 0;                                           -- trough both sets). This is
                                  k := 0;                                           -- easy because the sets are
                                  WHILE (i < num_of_el_set(s1_in)) AND              -- ordered.
                                        (j < num_of_el_set(s2_in)) AND
                                        (k < max_num_of_el_p_s) LOOP
                                    IF ram(base_addr_s1 + i) < ram(base_addr_s2 + j) THEN    -- el.s1 < el.s2
                                      ram(base_addr_s3 + k) := ram(base_addr_s1 + i);        -- el.s3 := el.s1
                                      i := i + 1;
                                    ELSIF ram(base_addr_s1 + i) > ram(base_addr_s2 + j) THEN -- el.s1 > el.s2
                                      ram(base_addr_s3 + k) := ram(base_addr_s2 + j);        -- el.s3 := el.s2
                                      j := j + 1;
                                    ELSE                                                     -- el.s1 = el.s2
                                      ram(base_addr_s3 + k) := ram(base_addr_s1 + i);        -- el.s3 := el.s1
                                      i := i + 1;
                                      j := j + 1;
                                    END IF;
                                    k := k + 1;
                                  END LOOP;
                                  WHILE (i < num_of_el_set(s1_in)) AND (k < max_num_of_el_p_s) LOOP
                                    ram(base_addr_s3 + k) := ram(base_addr_s1 + i);          -- add rest of set1
                                    i := i + 1;                                              --  to set3
                                    k := k + 1;
                                  END LOOP;                                                  -- or
                                  WHILE (j < num_of_el_set(s2_in)) AND (k < max_num_of_el_p_s) LOOP
                                    ram(base_addr_s3 + k) := ram(base_addr_s2 + j);          -- add rest of set2
                                    j := j + 1;                                              --  to set3
                                    k := k + 1;
                                  END LOOP;                                                  -- or
                                  IF k = max_num_of_el_p_s THEN                              -- set3 is full?
                                    IF (i < num_of_el_set(s1_in)) OR (j < num_of_el_set(s2_in)) THEN
                                      error <= '1' AFTER delay;
                                    END IF;
                                  END IF;
                                  num_of_el_set(s3_in) := k;
                                  IF num_of_el_set(s3_in) = max_num_of_el_p_s THEN           -- give status set3
                                    full <= '1' AFTER delay;                                 -- set3 full?
                                  ELSE
                                    full <= '0' AFTER delay;
                                  END IF;
                                  IF num_of_el_set(s3_in) = 0 THEN                           -- set3 empty?
                                    empty <= '1' AFTER delay;
                                  ELSE
                                    empty <= '0' AFTER delay;
                                  END IF;
                                END IF;

              WHEN intersection => IF s1_in = s2_in
                                OR s1_in = s3_in
                                OR s2_in = s3_in
                                THEN
                                  error <= '1' AFTER delay;
                                ELSE
                                  base_addr_s1 := (s1_in - 1) * max_num_of_el_p_s;  -- Set3 is created by merging
                                  base_addr_s2 := (s2_in - 1) * max_num_of_el_p_s;  -- set1 and set2. (picking
                                  base_addr_s3 := (s3_in - 1) * max_num_of_el_p_s;  -- each time the smallest el.
                                  i := 0;                                           -- of set1 or 2, when we walk
                                  j := 0;                                           -- trough both sets). This is
                                  k := 0;                                           -- easy because the sets are
                                  WHILE (i < num_of_el_set(s1_in)) AND (j < num_of_el_set(s2_in)) LOOP --ordered.
                                    IF ram(base_addr_s1 + i) < ram(base_addr_s2 + j) THEN    -- el.s1 < el.s2
                                      i := i + 1;
                                    ELSIF ram(base_addr_s1 + i) > ram(base_addr_s2 + j) THEN -- el.s1 > el.s2
                                      j := j + 1;
                                    ELSE                                                     -- el.s1 = el.s2
                                      ram(base_addr_s3 + k) := ram(base_addr_s1 + i);        -- el.s3 := el.s1
                                      i := i + 1;
                                      j := j + 1;
                                      k := k + 1;
                                  END IF;
                                END LOOP;
                                num_of_el_set(s3_in) := k;
                                IF num_of_el_set(s3_in) = max_num_of_el_p_s THEN             -- give status set3
                                  full <= '1' AFTER delay;                                   -- set 3 full?
                                ELSE
                                  full <= '0' AFTER delay;
                                END IF;
                                IF num_of_el_set(s3_in) = 0 THEN                             -- se 3 empty?
                                  empty <= '1' AFTER delay;
                                ELSE
                                  empty <= '0' AFTER delay;
                                END IF;
                              END IF;

        WHEN difference => IF s1_in = s2_in
                              OR s1_in = s3_in
                              OR s2_in = s3_in
                              THEN
                                error <= '1' AFTER delay;
                              ELSE
                                base_addr_s1 := (s1_in - 1) * max_num_of_el_p_s;  -- Set3 is created by merging
                                base_addr_s2 := (s2_in - 1) * max_num_of_el_p_s;  -- set1 and set2. (picking
                                base_addr_s3 := (s3_in - 1) * max_num_of_el_p_s;  -- each time the smallest el.
                                i := 0;                                           -- of set1 or 2, when we walk
                                j := 0;                                           -- trough both sets). This is
                                k := 0;                                           -- easy because the sets are
                                WHILE (i < num_of_el_set(s1_in)) AND              -- ordered.
                                      (j < num_of_el_set(s2_in)) AND
                                      (k < max_num_of_el_p_s) LOOP
                                  IF ram(base_addr_s1 + i) < ram(base_addr_s2 + j) THEN     -- el.s1 < el.s2
                                    ram(base_addr_s3 + k) := ram(base_addr_s1 + i);         -- el.s3 := el.s1
                                    i := i + 1;
                                    k := k + 1;
                                  ELSIF ram(base_addr_s1 + i) > ram(base_addr_s2 + j) THEN -- el.s1 > el.s2
                                    ram(base_addr_s3 + k) := ram(base_addr_s2 + j);         -- el.s3 := el.s2
                                    j := j + 1;
                                    k := k + 1;
                                  ELSE                                                      -- el.s1 = el.s2
                                    i := i + 1;
                                    j := j + 1;
                                  END IF;
                                END LOOP;
                                WHILE (i < num_of_el_set(s1_in)) AND (k < max_num_of_el_p_s) LOOP
                                  ram(base_addr_s3 + k) := ram(base_addr_s1 + i);           -- add rest of set1
                                  i := i + 1;                                               --  to set3
                                  k := k + 1;
                                END LOOP;                                                   -- or
                                WHILE (j < num_of_el_set(s2_in)) AND (k < max_num_of_el_p_s) LOOP
                                  ram(base_addr_s3 + k) := ram(base_addr_s2 + j);           -- add rest of set2
                                  j := j + 1;                                               --  to set3
                                  k := k + 1;
                                END LOOP;                                                   -- or
                                IF k = max_num_of_el_p_s THEN                               -- set3 is full?
                                  IF (i < num_of_el_set(s1_in)) OR (j < num_of_el_set(s2_in)) THEN
                                    error <= '1' AFTER delay;
                                  END IF;
                                END IF;
                                num_of_el_set(s3_in) := k;
                                IF num_of_el_set(s3_in) = max_num_of_el_p_s THEN            -- give status set 3
                                  full <= '1' AFTER delay;                                  -- set 3 full?
                                ELSE
                                  full <= '0' AFTER delay;
                                END IF;
                                IF num_of_el_set(s3_in) = 0 THEN                            -- set 3 empty?
                                  empty <= '1' AFTER delay;
                                ELSE
                                  empty <= '0' AFTER delay;
                                END IF;
                              END IF;

        WHEN memb        => IF num_of_el_set(s1_in) = 0 THEN                               -- set empty.
                                empty <= '1' AFTER delay;                                  -- give status set.
                                full <= '0' AFTER delay;
```

```
                   member <= TRANSPORT '0' AFTER delay;
                 ELSE
                   IF num_of_el_set(s1_in) = max_num_of_el_p_s THEN            -- set not empty
                      full <= '1' AFTER delay;                                  -- give status set
                      empty <= '0' AFTER delay;
                   ELSE
                      full <= '0' AFTER delay;
                      empty <= '0' AFTER delay;
                   END IF;
                   located := false;
                   base_addr_s1 := (s1_in - 1) * max_num_of_el_p_s;             -- Search element.
                   FOR i IN base_addr_s1 TO
                           base_addr_s1 + num_of_el_set(s1_in) - 1
                   LOOP
                     IF located = false THEN                                    -- Element not found
                       IF ram(i) = d_in THEN                                    -- Compare elements.
                         located := true;                                       -- Element found.
                         member <= TRANSPORT '1' AFTER delay;
                       END IF;
                     END IF;
                   END LOOP;
                 END IF;

WHEN insert    => located := false;
                 base_addr_s1 := (s1_in - 1) * max_num_of_el_p_s;              -- Search element.
                 IF num_of_el_set(s1_in) /= 0 THEN
                   FOR i IN base_addr_s1 TO
                     base_addr_s1 + num_of_el_set(s1_in) - 1
                   LOOP
                     IF located = false THEN                                    -- Element not found
                       IF ram(i) = d_in THEN                                    -- Compare elements.
                         located := true;                                       -- El. in set.
                         member <= TRANSPORT '1' AFTER delay;
                       ELSIF ram(i) > d_in THEN                                 -- El. not in set
                         IF num_of_el_set(s1_in) < max_num_of_el_p_s THEN       -- set not full.
                           FOR j IN base_addr_s1 + num_of_el_set(s1_in) - 1 DOWNTO
                                    base_addr_s1 + i
                           LOOP
                             ram(j+1) := ram(j);
                           END LOOP;
                           ram(base_addr_s1 + i) := d_in;
                           member <= '1' AFTER delay;
                           located := true;
                           num_of_el_set(s1_in) := num_of_el_set(s1_in) + 1;
                         ELSE
                           located := true;
                           error <= '1' AFTER delay;                            -- Unable to insert
                         END IF;                                                --  el. (set full).
                       END IF;
                     END IF;
                   END LOOP;
                   IF located = false THEN                                      -- no el. inserted.
                     IF num_of_el_set(s1_in) < max_num_of_el_p_s THEN           -- set not full.
                       ram(base_addr_s1 + num_of_el_set(s1_in)) := d_in;
                       num_of_el_set(s1_in) := num_of_el_set(s1_in) + 1;
                     ELSE
                       error <= '1' AFTER delay;
                     END IF;
                   END IF;
                 ELSE
                   ram(base_addr_s1) := d_in;
                   num_of_el_set(s1_in) := 1;
                 END IF;
                 IF num_of_el_set(s1_in) = max_num_of_el_p_s THEN              -- set full?
                   full <= '1' AFTER delay;                                     -- give status set
                 ELSE
                   full <= '0' AFTER delay;
                 END IF;
                 empty <= '0' AFTER delay;

WHEN delete    => IF num_of_el_set(s1_in) /= 0 THEN                            -- set not empty
                   located := false;
                   base_addr_s1 := (s1_in - 1) * max_num_of_el_p_s;             -- Search element.
                   FOR i IN base_addr_s1 TO
```

```
                          base_addr_s1 + num_of_el_set(s1_in) - 1
                 LOOP
                   IF located = false THEN                              -- Elem. not found.
                     IF ram(i) > d_in THEN                              -- Compare elements.
                       located := true;                                 -- Elem. not in set.
                     ELSIF ram(i) = d_in THEN                           -- Element in set
                       FOR j IN i TO                                    -- Delete element
                                base_addr_s1 + num_of_el_set(s1_in) - 1 --   (ordered).
                       LOOP
                         ram(j) := ram(j+1);
                       END LOOP;
                       located := true;
                       num_of_el_set(s1_in) := num_of_el_set(s1_in) - 1;
                     END IF;
                   END IF;
                 END LOOP;
               END IF;
               IF num_of_el_set(s1_in) = max_num_of_el_p_s THEN         -- set full?
                 full <= '1' AFTER delay;                               -- give status set
               ELSE
                 full <= '0' AFTER delay;
               END IF;
               IF num_of_el_set(s1_in) = 0 THEN                         -- set empty?
                 empty <= '1' AFTER delay;
               ELSE
                 empty <= '0' AFTER delay;
               END IF;

WHEN assign   => IF num_of_el_set(s2_in) /= 0 THEN
                 base_addr_s1 := (s1_in - 1) * max_num_of_el_p_s;       -- Give set1 the same
                 base_addr_s2 := (s2_in - 1) * max_num_of_el_p_s;       --  values as set2.
                 FOR i IN 0 TO num_of_el_set(s2_in) - 1 LOOP
                   ram(base_addr_s1 + i) := ram(base_addr_s2 +i);
                 END LOOP;
               END IF;
               num_of_el_set(s1_in) := num_of_el_set(s2_in);
               IF num_of_el_set(s1_in) = max_num_of_el_p_s THEN         -- set full?
                 full <= '1' AFTER delay;                               -- give status set
               ELSE
                 full <= '0' AFTER delay;
               END IF;
               IF num_of_el_set(s1_in) = 0 THEN                         -- set empty?
                 empty <= '1' AFTER delay;
               ELSE
                 empty <= '0' AFTER delay;
               END IF;

WHEN equal    => base_addr_s1 := (s1_in - 1) * max_num_of_el_p_s;       -- Decides whether
               base_addr_s2 := (s2_in - 1) * max_num_of_el_p_s;         --  set1 and set2 are
               equal_var := true;                                       --  equal. Error:='1'
               IF num_of_el_set(s1_in) = num_of_el_set(s2_in) THEN      --  if not equal.
                 IF num_of_el_set(s1_in) /= 0 THEN
                   FOR i  IN 0 TO num_of_el_set(s1_in) - 1 LOOP
                     IF equal_var = true THEN
                       IF ram(base_addr_s1 + i) /= ram(base_addr_s2 + i) THEN
                         equal_var := false;
                       END IF;
                     END IF;
                   END LOOP;
                 END IF;
               ELSE
                 equal_var := false;
               END IF;
               IF equal_var = false THEN
                 error <= '1' AFTER delay;
               END IF;
               IF num_of_el_set(s1_in) = max_num_of_el_p_s THEN         -- set full?
                 full <= '1' AFTER delay;                               -- give status set
               ELSE
                 full <= '0' AFTER delay;
               END IF;
               IF num_of_el_set(s1_in) = 0 THEN                         -- set empty?
                 empty <= '1' AFTER delay;
               ELSE
```

```
                                  empty <= '0' AFTER delay;
                                END IF;

          WHEN min_el        => base_addr_s1 := (s1_in - 1) * max_num_of_el_p_s;          -- Give to d_out the
                                IF num_of_el_set(s1_in) /= 0 THEN                          --  min-element.
                                  d_out <= ram(base_addr_s1) AFTER delay;
                                ELSE
                                  error <= '1' AFTER delay;
                                END IF;
                                IF num_of_el_set(s1_in) = max_num_of_el_p_s THEN           -- set full?
                                  full <= '1' AFTER delay;                                 -- give status set
                                ELSE
                                  full <= '0' AFTER delay;
                                END IF;
                                IF num_of_el_set(s1_in) = 0 THEN                           -- set empty?
                                  empty <= '1' AFTER delay;
                                ELSE
                                  empty <= '0' AFTER delay;
                                END IF;

          WHEN max_el        => base_addr_s1 := (s1_in - 1) * max_num_of_el_p_s;          -- Give to d_out the
                                IF num_of_el_set(s1_in) /= 0 THEN                          --  max element.
                                  d_out <= ram(base_addr_s1 + num_of_el_set(s1_in) - 1) AFTER delay;
                                ELSE
                                  error <= '1' AFTER delay;
                                END IF;
                                IF num_of_el_set(s1_in) = max_num_of_el_p_s THEN           -- set full?
                                  full <= '1' AFTER delay;                                 -- give status set
                                ELSE
                                  full <= '0' AFTER delay;
                                END IF;
                                IF num_of_el_set(s1_in) = 0 THEN                           -- set empty?
                                  empty <= '1' AFTER delay;
                                ELSE
                                  empty <= '0' AFTER delay;
                                END IF;

          END CASE;     -- op_code
          ready <= TRANSPORT '1' AFTER delay;

        END IF;     -- en = '1'

      END IF;     -- clk = '1'

  END PROCESS set_operation;

  init:PROCESS

    BEGIN
      full <= '0';
      empty <= '1';
      member <= '0';
      ready <= '1';
      error <= '0';
      wait;
    END PROCESS init;

END behavior1;
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;      -- This is a mentor graphics package.

PACKAGE set_pkg IS


CONSTANT delay              : time     := 2ns;
CONSTANT num_of_sets        : integer :=  4;
CONSTANT num_of_el_per_set  : integer :=  4;
CONSTANT elementlength      : integer :=  4;

TYPE set_mnemonic IS ( clear, union, intersection, difference, memb, insert, delete, assign,
                                                                      equal, min_el, max_el);

SUBTYPE data                IS integer RANGE 0 to (2**elementlength) - 1;
-- TYPE data                   IS ARRAY (elementlength -1 DOWNTO 0) of bit;

SUBTYPE set_name            IS integer RANGE 1 TO num_of_sets;
-- TYPE set_name               IS ARRAY (2log num_of_sets -1 DOWNTO 0) of bit;

END set_pkg;

-- This file contains the information, which should be implemented with the
-- GENERIC-clause in the set entity.

-- The type DATA should be a bit_vector(elementlength-1 DOWNTO 0).
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;
USE work.graph_pkg.ALL;

-- Type declarations and constant declarations are in graph_pkg. This is a graph with operations INSERT,
-- DELETE and RETRIEVE. The operations are started at the first clock transition from 0 to 1, and only
-- when EN is high. An operation is finished when the READY signal becomes high. If an operation can not
-- execute, a N_OP is reported. If there is no edge between two nodes, the null-label (0) is assigned to
-- that position. If an node is deleted, also the null label is assigned to that node. I have restricted
-- the model, by assymming that the nodes are always numbered from 1 to num_of_nodes. If we want to name
-- the nodes arbitrarily, we have to use a table to find the correct addresses.
--
-- CLEAR:
--
--      - This operation makes the graph an empty graph.
--
-- INSERT_NODE:
--      Input: node1, label_in
--
--      - This operation inserts a node 'node1' in the graph and adds a label 'lable_in' to it. The
--        node is not connected with the rest of the graph.
--
-- INSERT_EDGE:
--      Input: node1, node2, label_in
--
--      - This operation inserts a edge between node 'node1' and node 'node2' in the graph and adds a
--        label 'lable_in' to it. The nodes 'node1' and 'node2' are both in the graph.
--
-- DELETE_NODE:
--      Input: node1
--      Output: label_out
--
--      - This operation will output the label of node 'node1' to output port 'label_out', and then deletes
--        the node 'node1' from the graph. node 'node1' is not connected to the graph.
--
-- DELETE_EDGE:
--      Input: node1, node2
--      Output: label_out
--
--      - This operation will output the label of the edge between node 'node1' and node 'noide2' to output
--        port 'label_out', and then deletes the edge from the graph.
--
-- RETRIEVE_NODE:
--      Input: node1
--      Output: label_out
--
--      - This operation will output the label from node 'node1' to output port 'label_out'. If no label is de
fined,
--        the nul label is send out.
--
-- RETRIEVE_EDGE:
--      Input: node1, node2
--      Output: label_out
--
--      - This operation will output the label from the edge between node 'node1' and node 'node2' to output p
ort
--        'label_out'. If no label is defined, the nul label is send out.
--


ENTITY graph IS

--   GENERIC (
--            delay            : time;
--            num_of_nodes     : positive;
--            max_label_length : positive
--            nul_label        : ?
--          );

   PORT (
         clk        : IN    bit;
         en         : IN    bit;
         op_code    : IN    graph_mnemonic;
         node1      : IN    node_id;
```

```
        node2     : IN    node_id;
        label_in  : IN    label_data;
        label_out : OUT   label_data;
        n_op      : OUT   bit;
        ready     : OUT   bit
      );

END graph;


ARCHITECTURE behavior1 OF graph IS

  TYPE node_label_ram  IS ARRAY (node_id) of label_data;
  TYPE matrix          IS ARRAY (node_id, node_id) of label_data;

BEGIN

  graph_operation:PROCESS(clk)

    VARIABLE node_ram        : node_label_ram;  -- Array with node lables.
    VARIABLE adj_matrix      : matrix;          -- Matrix with edge lables.
    VARIABLE i               : node_id;         -- i, j , k and l are variables
    VARIABLE j               : node_id;         --    used with loops. During a
    VARIABLE k               : node_id;         --    loop : i = k and j = l
    VARIABLE l               : node_id;
    VARIABLE exist_edge      : boolean;         -- Control variable for testing edge existence

    BEGIN

      IF clk = '1' THEN
        n_op <= '0';

        IF en = '1' THEN
          ready <= '0';

          CASE op_code IS

          WHEN clear    =>  k:= node_id'low;
                            FOR i IN node_id'low TO node_id'high LOOP
                              node_ram(k) := nul_label;
                              l:= node_id'low;
                              FOR j IN node_id'low TO node_id'high LOOP
                                adj_matrix(k,l) := nul_label;
                                IF j < node_id'high THEN
                                  l:= l + 1;
                                END IF;
                              END LOOP;
                              IF i < node_id'high THEN
                                k := k + 1;
                              END IF;
                            END LOOP;
                            label_out <= nul_label AFTER delay;

          WHEN ins_node =>  IF node_ram(node1) = nul_label THEN
                              node_ram(node1) := label_in;
                            ELSE
                              n_op <= '1' AFTER delay;
                            END IF;

          WHEN ins_edge =>  IF (node_ram(node1) /= nul_label AND node_ram(node2) /= nul_label) THEN
                              adj_matrix(node1, node2) := label_in;
                            ELSE
                              n_op <= '1' AFTER delay;
                            END IF;

          WHEN del_node =>  IF node_ram(node1) /= nul_label THEN
                              exist_edge := false;
                              k := node_id'low;
                              FOR i IN node_id'low TO node_id'high LOOP
                                IF adj_matrix(node1, k) /= nul_label THEN
                                  exist_edge := true;
                                END IF;
                                IF adj_matrix(k, node1) /= nul_label THEN
                                  exist_edge := true;
                              END IF;
                              IF exist_edge = true THEN
                                n_op <= '1' AFTER delay;
                              END IF;
                              IF i /= num_of_nodes THEN
                                k := k + 1;
                              END IF;
                            END LOOP;
                            IF exist_edge = false THEN
                              label_out <= node_ram(node1) AFTER delay;
                              node_ram(node1) := nul_label;
                            END IF;
                          ELSE
                            label_out <= nul_label AFTER delay;
                          END IF;

        WHEN del_edge =>  IF (node_ram(node1) /= nul_label AND node_ram(node2) /= nul_label) THEN
                            label_out <= adj_matrix(node1,node2) AFTER delay;
                            adj_matrix(node1, node2) := nul_label;
                          ELSE
                            n_op <= '1' AFTER delay;
                          END IF;

        WHEN retr_node => label_out <= node_ram(node1);
                          IF node_ram(node1) = nul_label THEN
                            n_op <= '1' AFTER delay;
                          END IF;

        WHEN retr_edge => IF (node_ram(node1) /= nul_label AND node_ram(node2) /= nul_label) THEN
                            label_out <= adj_matrix(node1, node2) AFTER delay;
                          ELSE
                            label_out <= nul_label AFTER delay;
                            n_op <= '1' AFTER delay;
                          END IF;

        END CASE;    -- opcode
        ready <= TRANSPORT '1' AFTER delay;

      END IF;    -- en = '1'

    END IF;    -- clk = '1'

  END PROCESS graph_operation;



 init:PROCESS

  BEGIN
    ready <= '0';
    n_op <= '0';
    label_out <= nul_label;
    wait;
  END PROCESS init;

END behavior1;
```

```
USE std.standard.ALL;
USE std.mentor_base.ALL;     -- This is a Mentor Graphics pakage;

PACKAGE graph_pkg IS

CONSTANT delay              : time    := 2ns;          -- Generic
CONSTANT num_of_nodes       : integer :=  4;           -- Generic
CONSTANT max_label_length   : integer :=  4;           -- Generic

TYPE graph_mnemonic         IS (clear,ins_node,ins_edge,del_node,del_edge,retr_node,retr_edge);
TYPE label_data             IS RANGE 0 TO (2**max_label_length)-1;
-- TYPE label_data             IS ARRAY (element_length - 1 DOWNTO 0);
CONSTANT nul_label          : label_data := 0;
TYPE node_id                IS RANGE 1 TO num_of_nodes;
-- TYPE node_id                IS ARRAY ( ... ? DOWNTO 0);


END graph_pkg;

-- This file contains the information, which should be implemented with the
-- GENERIC-clause in the list entity.

-- The type DATA should be a bit_vector(element_length-1 DOWNTO 0).
```