

Chapter 2

□ BASIC ABSTRACT DATA TYPES

2.1 Introduction

We now study the elementary data structures which are used in the construction of more complicated data structures. We then use some of these to implement some fundamental algorithms. These algorithms, in turn, will be used to implement more complicated algorithms.

As a general rule we will study each abstract data type by

1. Defining the ADT,
2. Indicate some applications of it,
3. Discuss various implementations of it,
4. Consider further applications.

Virtual Machine Model. We assume that the machine on which the algorithms will be implemented has a single processor and that memory is an array of contiguous cells indexed by a single number (address). We further assume that address arithmetic can be performed on these addresses and that a cell can be accessed in $O(1)$ (constant) time, given its address. This is a *Random Access Machine*

2.2 Elementary Data Structures

Apart from scalars, there are two fundamental ways to store structures in a machine : *contiguous* or array storage, and *linked* or pointer storage.

Contiguous Storage. Each element of the structure is stored beside the next as shown in Figure 1.1. Once the address of the first element is known then the address of all other elements can be calculated. This is because each element is in a fixed position relative to the first. Of necessity, the size of such a structure is *static* (fixed). The address of any element is calculated as follows :

$$Addr(a[i]) = Addr(a[1]) + i - 1$$

Access to any element requires $O(1)$ time.

Linked Storage. Each element of the structure may be stored anywhere in memory. For this reason we must explicitly store in each element the address of the next element. This is shown in Figure 1.2. The size of such a structure is *dynamic* (variable) because elements can be added and deleted. The address of any element is calculated as follows :

$$Addr(a_i) = NextAddr(a_i)$$

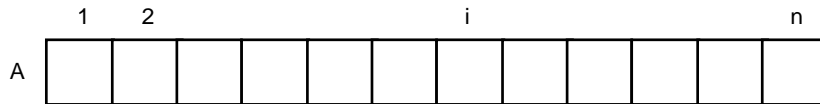


Figure 2.1. Contiguous Storage.

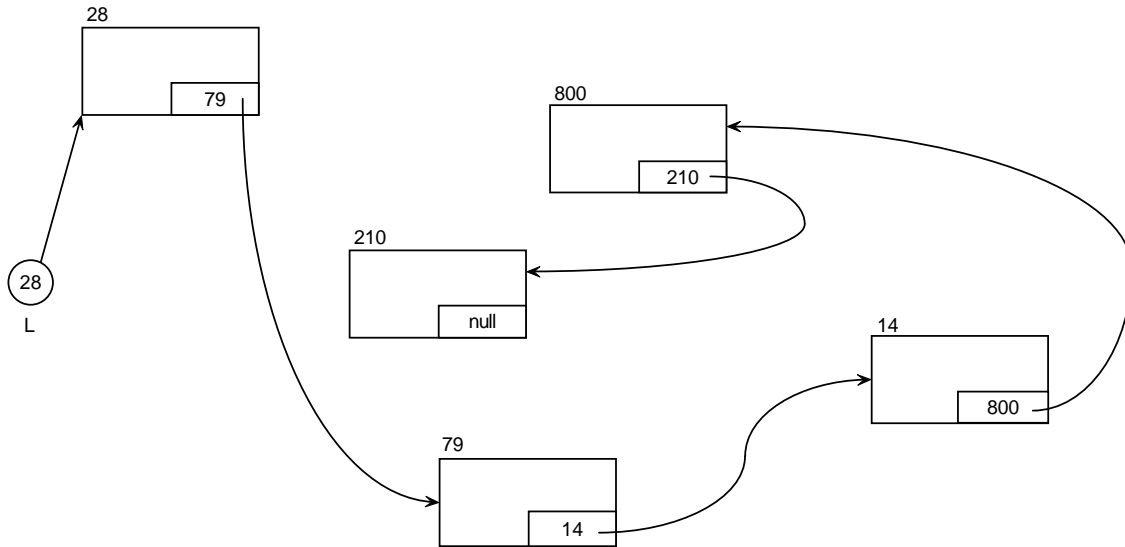


Figure 2.2. Linked Storage.

Access to any element requires $O(n)$ time, where n is the number of elements in the structure.

Each element of a linked structure must contain the address (link) of the next element. To avoid confusion we call this combined element a *node*, i.e., a node contains the element and the link. A node can be easily implemented as a 2-field record as follows :

```

TYPE
  nodeptrtype = ^ NodeType;
  NodeType = RECORD
    element : elemtype;
    link    : nodeptrtype;
  END;
    
```

The basic abstract data types we consider in this chapter are

1. List
2. Stack
3. Queue
4. Deque
5. Table

2.3 The ADT List

The ADT List is used by many algorithms alone or in combination with other ADTs. We will see that the (linked) list and the array are used in combination to implement all sorts of complicated data structures.

DEFINITION (*List*).

- Set : elements of any type
- Structure : The elements form a sequence, i.e., there is a 1st, 2nd, . . . , last element.
- Operations : $Create(L)$, $Empty(L)$, $Insert(x, p, L)$, $Delete(p, L)$, etc. (see below)
- Implementation : Array, Pointer, or Cursor.

□

We will represent a list in typeface as

$$L = (e_1, e_2, \dots, e_i, \dots, e_n)$$

Graphically, a list is represented as shown in Figure 2.3

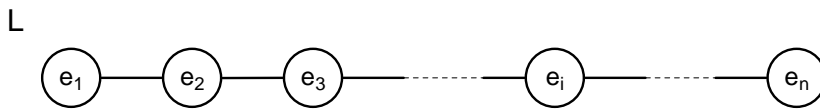


Figure 2.3. A List.

Abstract Operations on Lists. We now define a set of operations to be performed on lists. In what follows L is a *list* of elements of type *elemtype*, x is an *element*, and p denotes the position of some element in the list and has type *postype*. In general p does not denote the *order* position of an element, i.e., it is not necessarily one of the values $\{1, 2, \dots, n\}$. This is because the type of p will depend on the implementation of the list. This point will become clear after we have discussed the various implementations of lists.

The operations on lists are as follows, where L' denotes the list *before* an operation is performed on it and L'' denotes the list *after* the operation :

1. $Create(L') \rightarrow$ an empty list $L'' = ()$
2. $Empty(L') \rightarrow$ true or false, $L'' = L'$.
3. $InsertA(x, p, L') \rightarrow L''$ where

$$L' = (e_1, e_2, \dots, e_p, \dots, e_n)$$

$$L'' = (e_1, e_2, \dots, e_p, x, \dots, e_n)$$

4. $DeleteA(p, L') \rightarrow L''$ where

$$L' = (e_1, e_2, \dots, e_p, e_{p+1}, \dots, e_n)$$

$$L'' = (e_1, e_2, \dots, e_p, e_{p+2}, \dots, e_n)$$

5. $Locate(x, L) \rightarrow p$ position of x in $L, L'' = L'$.
6. $Retrieve(p, L) \rightarrow x$ the element at position p in $L, L'' = L'$.
7. $Prev(p, L)$ and $Next(p, L) \rightarrow q$ the positions of elements before and after the element at position $p, L'' = L'$.

Examples of List Operations.

```

L1 = ( dog, cat, rat, bat)
InsertA(mouse, Locate(rat, L1), L1)
L1 = (dog, cat, rat, mouse, bat)
Delete(First(L1), L1)
L1 = (cat, rat, mouse, bat)
L2 = (apple, orange, pear, banana)
Concatenate(L1, L2)
L1 = (cat, rat, mouse, bat, apple, orange, pear, banana)
    
```

2.3.1 Implementations of the ADT List

We now discuss three data structures for representing the ADT list and implementing its operations.

Array Implementation. In this implementation elements of the list are stored in contiguous cells of an array, as shown in Figure 4.

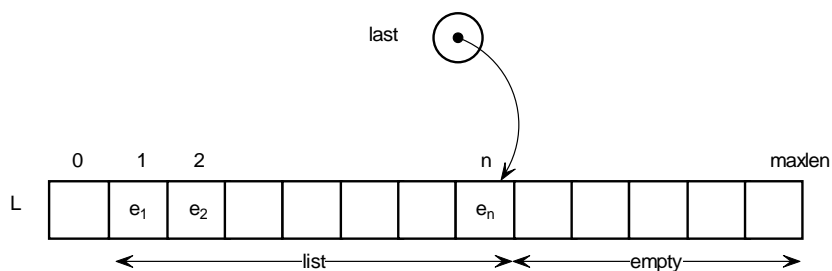


Figure 2.4. Array Implementation of ADT List.

The variable `last` contains the array location of the last element of the list. The list is empty when `last = 0`. We will wrap all the pieces of this data structure in a record.

We now define a Turbo Pascal unit that implements this data structure and the operations performed on it. We do this to gain modularity and information hiding.

```

UNIT ListArrayADT;


---


INTERFACE
USES ElemDefU; {supplied by the user to define the element type}
CONST
  null = 0;
  maxlen = 1000;
TYPE
  postype = null..maxlen;
  listtype = RECORD
    elem : ARRAY[postype] OF elemtype;
    last : postype;
  END;
{ NOTE: elemtype comes from ElemDefU }


---


PROCEDURE Create( VAR List:ListType );
PROCEDURE InsertA ( x:elemtype; p:postype; L:ListType);
PROCEDURE DeleteA ( p:postype; L:ListType);
FUNCTION Empty ( List:Listtype ):BOOLEAN;
FUNCTION Locate ( x:elemtype; List:ListType):postype;
FUNCTION First ( List:ListType):postype;


---


IMPLEMENTATION
PROCEDURE Create( VAR List:ListType );
BEGIN
  L.last := null;
END; {PROC Create }

FUNCTION Empty ( List:Listtype ):BOOLEAN;
BEGIN
  Empty := L.last=null;
END; {FUNC Empty}

PROCEDURE InsertA ( x:elemtype; p:postype; L:ListType);
VAR
  q:postype;
BEGIN
  IF Full(L) THEN Error('InsertA')
  ELSE BEGIN
    FOR q := L.last DOWNTO p+1 DO BEGIN
      L.elem[q+1] := L.elem[q];
    END; {FOR q}
    L.elem[p+1] := x;
    Inc(L.last);
  END; {IF}
END; {PROC InsertA}

PROCEDURE DeleteA ( p:postype; L:ListType);
VAR
  q:postype;
BEGIN
  IF Empty(L) THEN Error('DeleteA')
  ELSE BEGIN
    Dec(L.last);
    FOR q := p+1 TO L.last DO BEGIN
      L.elem[q] := L.elem[q+1];
    END; {FOR q}
  END; {IF}
END; {PROC DeleteA}

FUNCTION Locate ( x:elemtype; List:ListType):postype;
{ Do as an EXERCISE }

FUNCTION First ( List:ListType):postype;
BEGIN
  IF Empty(L) THEN First := null
  ELSE First := 1;
END; {FUNC First}

```

Exercise 2.3.1 : Write the remaining procedures and functions.

□

Analysis of the Array Implementation of ADT List. This implementation is poor for two reasons

1. *InsertA* and *DeleteA* require $O(n)$ time.
2. The array structure is static and does not use memory efficiently. For example, we could have two lists L_1 and L_2 , with L_1 full and L_2 empty. If we try to insert one more element in L_1 then we get an overflow error and cannot continue. Yet, at the same time, L_2 is empty and its space is unused. This problem becomes more acute as more lists are created and in use simultaneously.

The array implementation is good for some special cases of lists, such as *stacks* and *queues*. We will see these later.

Pointer Implementation. In this implementation we store each element in a separate *node* which also contains the address (pointer to) of the next node on the list. We use a *header node* at the start of each list to avoid some of the problems of checking for special cases. This structure is shown in Figure 5.

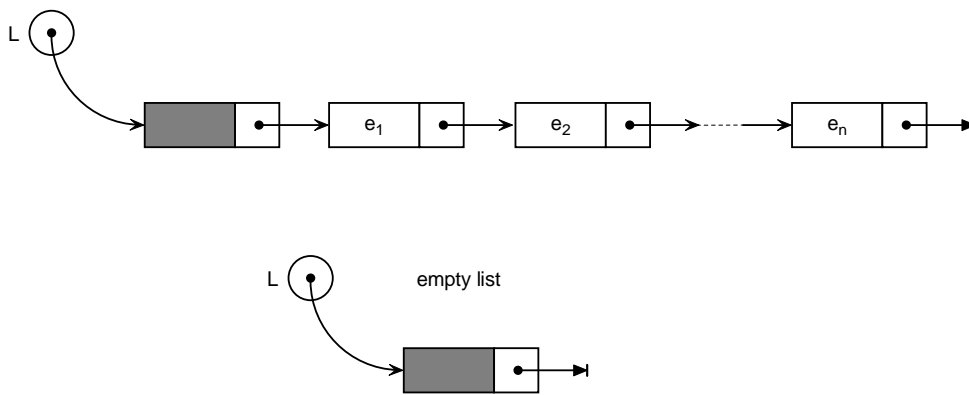


Figure 2.5. Pointer Implementation of ADT List.

```

UNIT SLHLists; {Pointer implementation of singly-linked lists with header nodes}
INTERFACE
USES
    ElemDefU;
CONST
    NULL = NIL;
TYPE
    NodePtrType = ^ Nodetype;
    Nodetype = RECORD
        elem : elemtype;
        link : NodePtrType;
    END;
    ListType = NodePtrType;
PROCEDURE Create( VAR List:ListType );
PROCEDURE InsertA ( x:elemtype; p:NodePtrType; L:ListType);
FUNCTION DeleteA ( p:NodePtrType; L:ListType);
FUNCTION Empty ( List:Listtype ):BOOLEAN;
FUNCTION Locate ( x:elemtype; List:ListType):NodePtrType;
{End of INTERFACE}
    
```

The code above is the unit's *interface* and is visible to any user of the unit. As such, it is the *public* part of the unit. Notice that only the procedure and function *headers* are given in the interface and the bodies are

hidden from the user. In practice each procedure or function header would have comments to describe the purpose and operation of each. These have been omitted to save space.

We must now write the implementation part of this unit. In general we will draw a diagram of what we wish each procedure to accomplish. If this is done correctly then it is often a simple matter to translate the diagram into code. Usually we will start with the general case and then consider the special cases. These tend to occur at the boundaries of the data structure. Hence we say that the procedures must take account of the *boundary conditions*.

We start, as always, with the creation of the empty data structure, the empty list, in this case. The diagram is shown in Figure 6.

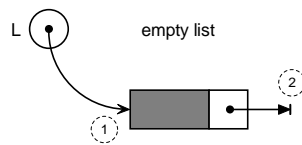


Figure 2.6. The Empty List.

The $InsertA(x, p, L)$ and $DeleteA(p, L)$ operations are shown in Figures 7 and 8. Note the correspondence between the numbered circles and the code below. The dotted lines denote things that have changed.

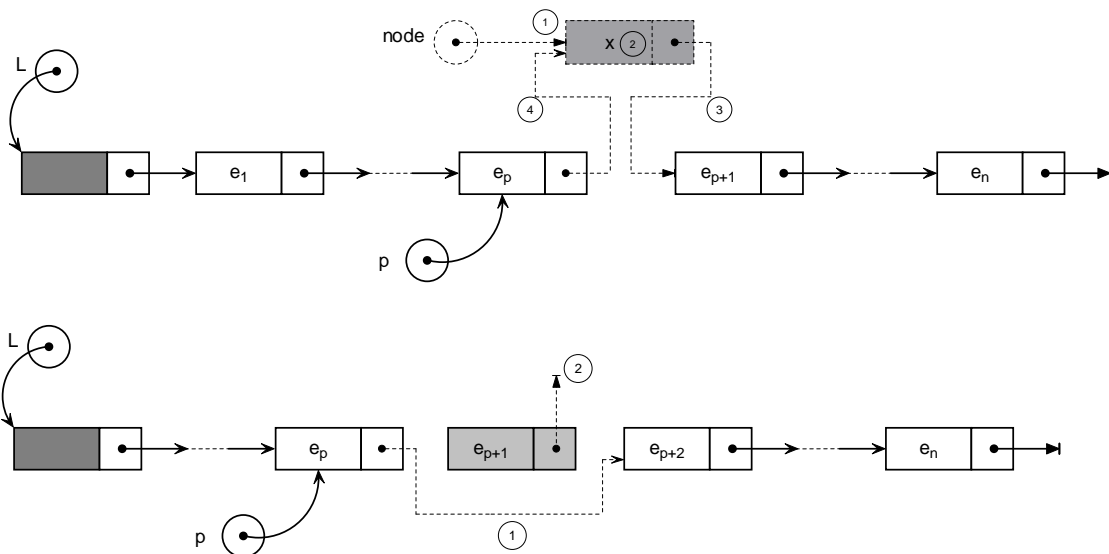


Figure 2.7. The $Insert$ and $DeleteA$ Operations.

```

IMPLEMENTATION { Singly-Linked Lists }
-----
PROCEDURE Create( VAR List:ListType );
BEGIN
    New(List);                { 1 }
    List^.link := NULL;      { 2 }
END; { PROC Create }
-----
FUNCTION Empty ( List:ListType ):BOOLEAN;
BEGIN
    Empty := (List^.link = NULL);
END; { FUNC Empty }
-----
PROCEDURE InsertA(x:elemtype; p:NodePtrType; L:ListType);
VAR
    node:NodePtrType;
BEGIN
    New(node);                { 1 }
    node^.elem := x;          { 2 }
    node^.link := p^.link;    { 3 }
    p^.link := node;          { 4 }
END; { PROC Insert }
-----
FUNCTION DeleteA(p:NodePtrType;L:ListType):elemtype;
BEGIN
    IF p^.link = NULL THEN { Error ('Delete') }
    ELSE BEGIN
        DeleteA := p^.link^.elem;
        p^.link := p^.link^.link;    { 1 }
        p^.link := null;              { 2 }
    END; { FUNC Delete }
-----
FUNCTION Locate ( x:elemtype; List:ListType):NodePtrType;
VAR
    p      : NodePtrType;
    found  : BOOLEAN;
BEGIN
    p      := L^.link;
    found  := FALSE;
    WHILE (p <> NULL) AND (NOT found) DO BEGIN
        found := (p^.elem = x) ;
        IF NOT found THEN p := p^.link;
    END; { WHILE }
    Locate := p;
END; { FUNC Locate }
-----
BEGIN
{ No initialization code}
END. {UNIT SLHLists}

```

Exercise 2.3.1 : Complete the unit above by adding the missing procedures and functions. In particular, add the procedures *InsertFirst(x, L)* and *DeleteLast(L)*. These operations are not possible with the procedures defined in the unit above. Note also that these new procedures are necessary for inserting an element into an empty list and deleting the only element in a list. □

Analysis of the Pointer Implementation of ADT List.

• Advantages:

1. *InsertA* and *DeleteA* require $O(1)$ time.
2. Memory is efficiently used. There is no wasted space.
3. It is a very flexible data structure and can be used in many applications, either by itself or as part of a more complicated data structure.

• Disadvantages :

1. Random access is impossible because we must follow the links to get to a particular node.
2. Movement along the list is one-way. This means that $Pred(p, L)$ requires $O(n)$ time.

Exercise 2.3.1 : There is a ‘node-copying’ trick that allows deletion of the node being pointed by p , without finding out the location of the predecessor node. What is the trick? [It is not really a trick in that it breaks no rules of good programming practice]. □

Cursor Implementation. We will do this implementation at the end of this chapter.

Circular Lists. These are useful when we wish to repeatedly process the elements of a list as it expands and contracts. For example, a telephone help line has customers joining a queue (*Insert*) and waiting for service. They leave when their query is answered (*Delete*). The help-line operator processes the elements of this queue in a circular fashion.

A linked can be made circular as shown in Figure 9.

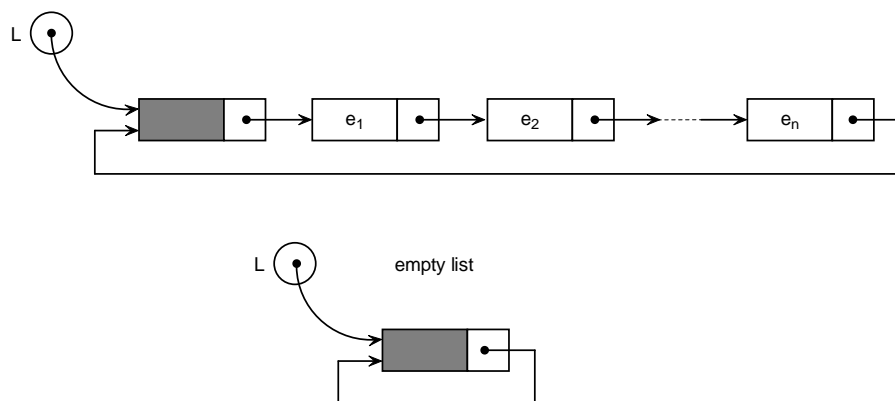


Figure 2.8. A Circular List.

The following procedure is a prototype for processing circular lists. Notice that it has an infinite **while** - loop, and as such it is very similar to an operating system which runs while the computer is turned on and processes each task in the order in which it occurred.

```

procedure ProcessCircList(VAR CList : CListType);
VAR
    p : nodeptrtype;
BEGIN
    WHILE True DO BEGIN
        IF p <> Clist THEN Process(p);
        p := Next(p);
    END;
END; { PROC ProcessCircList }
    
```

Doubly-Linked Lists. Doubly-linked lists allow movement in both directions because we maintain links in both directions. The advantage of this two-way movement and the flexibility of these lists far outweigh the extra memory and programming needed to implement them.

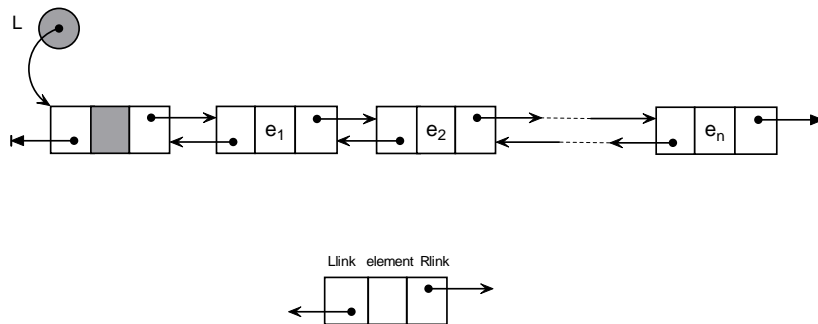


Figure 2.9. A Doubly-Linked List.

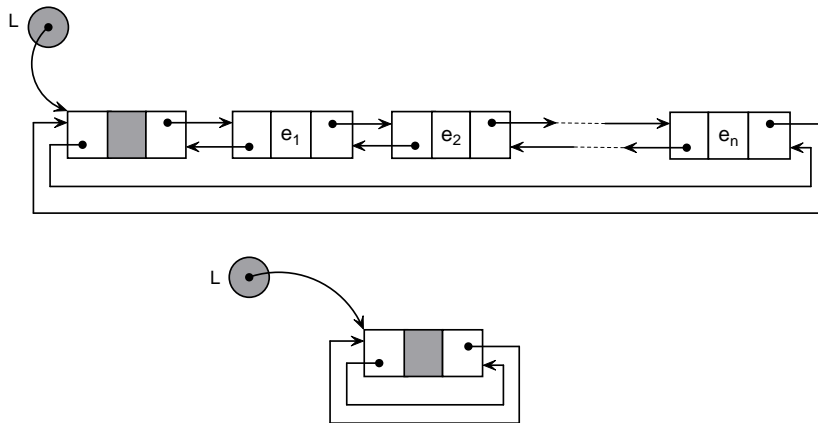


Figure 2.10. A Doubly-Linked Circular List.

We now implement the *List* ADT using doubly-linked circular lists. The INTERFACE or export section of the list ADT is shown below. Notice that we can insert before or after a given node p and that we can delete the node p rather than the node after p

```

UNIT DLHCLists;
{Implementation of ADT List with doubly-linked circular lists
with header nodes}

INTERFACE

USES
  ElemDefU;
CONST
  NULL = NIL;
TYPE
  NodePtrType = ^ Nodetype;
  Nodetype = RECORD
    Rlink : NodePtrType;
    elem  : elemtype;
    Llink : NodePtrType;
  END;
  ListType = NodePtrType;

PROCEDURE Create( VAR List:ListType );
PROCEDURE InsertB ( x:elemtype; p:NodePtrType;VAR L:ListType);
PROCEDURE InsertA ( x:elemtype; p:NodePtrType;VAR L:ListType);
FUNCTION  Delete ( p:NodePtrType;VAR L:ListType);
FUNCTION  Empty (List:Listtype ):BOOLEAN;
FUNCTION  Locate ( x:elemtype;List:ListType):NodePtrType;
{ ETC. }
{End of INTERFACE}
    
```

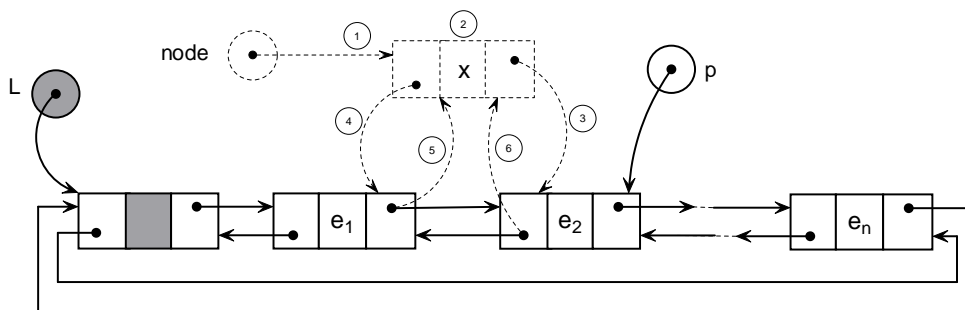


Figure 2.11. InsertB in a Doubly-Linked Circular List.

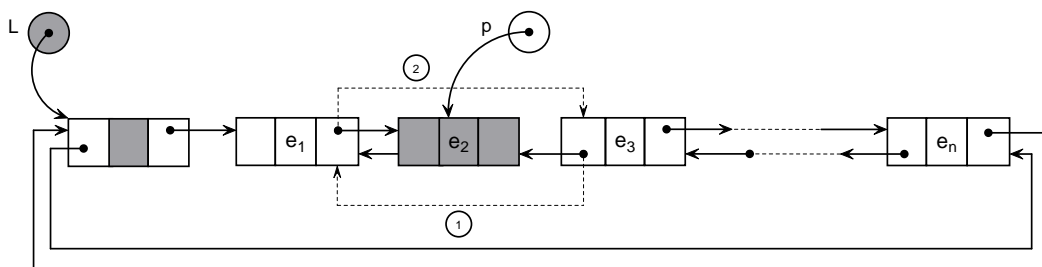


Figure 2.12. Delete in a Doubly-Linked Circular List.

```

IMPLEMENTATION { Doubly-Linked Circular Lists }
-----
PROCEDURE Create( VAR List:ListType );
BEGIN
    New(List);                { 1 }
    List^.Llink := List;     { 2 }
    List^.Rlink := List;     { 3 }
END; { PROC Create }
-----
FUNCTION Empty ( List:ListType ):BOOLEAN;
BEGIN
    Empty := (List^.Llink = List^.Rlink);
END; { FUNC Empty }
-----
PROCEDURE InsertB(x:elemtype; p:NodePtrType; L:ListType);
VAR
    node:NodePtrType;
BEGIN
    New(node);                { 1 }
    node^.elem := x;          { 2 }
    node^.Rlink := p;         { 3 }
    node^.Llink := p^.Llink; { 4 }
    p^.Llink^.Rlink := node; { 5 }
    p^.Llink := node;        { 6 }
END; { PROC InsertB }
-----
FUNCTION Delete(p:NodePtrType;L:ListType):elemtype;
BEGIN
    IF p^.link = NULL THEN { Error ('Delete') }
    ELSE BEGIN
        DeleteA := p^.link^.elem;
        p^.link := p^.link^.link; { 1 }
        p^.link := null;          { 2 }
    END; { FUNC Delete }
-----
FUNCTION Locate ( x:elemtype; List:ListType):NodePtrType;
VAR
    p : NodePtrType;
    found : BOOLEAN;
BEGIN
    p := L^.link;
    found := FALSE;
    WHILE (p <> NULL) AND (NOT found) DO BEGIN
        found := (p^.elem = x);
        IF NOT found THEN p := p^.link;
    END; { WHILE }
    Locate := p;
END; { FUNC Locate }
-----
BEGIN
{ No initialization code}
END. {UNIT SLHLists}

```

2.4 The ADT Stack

DEFINITION (*Stack*).

- Set and Structure : ADT List with all insertions and deletions a one end of the list, called the *Top*.
- Operations :
 1. *Create*(S') \rightarrow an empty list, $S'' = ()$
 2. *Empty*(L') \rightarrow true or false, $S'' = S'$.
 3. *Push*(x, S') $\rightarrow S''$, where $S' = (e_1, e_2, \dots, e_n)$ and $S'' = (x, e_1, e_2, \dots, e_n)$
 4. *Pop*(S') $\rightarrow e_1$, where $S' = (e_1, e_2, \dots, e_n)$ and $S'' = (e_2, \dots, e_n)$
 5. *Top*(S') $\rightarrow e_1, S'' = S'$.
- Implementation : Array or linked.

□

2.4.1 Applications of Stacks

Stacks are used in many areas of computing including the following :

- Tracing procedure calls in programming languages.
- Implementing recursive procedures.
- Traversing graph structures.

The following algorithm traverses a graph in *Depth First Order*.

algorithm *DFS*($s:node, G:graph$)

```

Mark all nodes  $u \in G$  'unvisited'
Create(S)
Push(s,S)
WHILE NOT Empty(S) DO
  u := Pop(S)
  Visit(u)
  FOR each  $v \in Adj(u)$  DO
    IF NOT visited(v) AND  $v \notin S$  THEN
      p[v] := u
      Push(v,S)
    ENDFOR
  ENDWHILE
endalg BFS

```

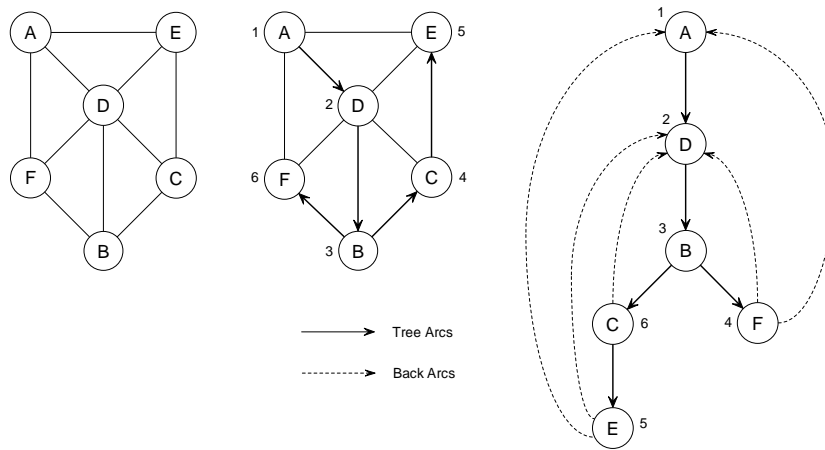


Figure 2.13. Depth First Search and its Tree.

2.4.2 Implementations of Stacks

Array Implementation.

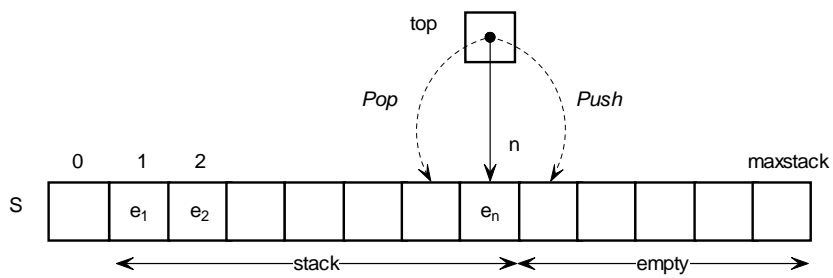


Figure 2.14. Array implementation of Stack.

```
UNIT StackArrayADT; {Array implementation of Stack ADT}
```

```
INTERFACE
USES ElemDefU;
{supplied by the user to define the element type}

CONST
  null = 0;
  maxstack = 1000;
TYPE
  postype = null..maxstack;
  stacktype = RECORD
    elem : ARRAY[postype] OF elemtype;
    top : postype;
  END;
{ NOTE: elemtype comes from ElemDefU }
```

```
PROCEDURE CreateS( VAR S:stacktype );
FUNCTION EmptyS(VAR S:stacktype ):BOOLEAN;
PROCEDURE Push(x:elemtype;VAR S:stacktype);
PROCEDURE Pop(VAR S:stacktype):elemtype;
PROCEDURE Top(VAR S:stacktype):elemtype;
```

```
IMPLEMENTATION
```

```
PROCEDURE CreateS( VAR S:stacktype ); BEGIN
  S.top := null;
END; {PROC Create }

FUNCTION EmptyS(VAR S:stacktype ):BOOLEAN;
BEGIN
  Empty := S.top=null;
END; {FUNC Empty}

PROCEDURE Push(x:elemtype;VAR S stacktype);
BEGIN
  IF Full(S) THEN Error('InsertA')
  ELSE BEGIN
    Inc(S.top);
    S[S.top] := x;
  END; {IF}
END; {PROC Push}

FUNCTION Pop(x:elemtype;VAR S stacktype):elemtype;
BEGIN
  IF Empty(S) THEN Error('Pop')
  ELSE BEGIN
    Pop := S[S.top];
    Dec(S.top);
  END; {IF}
END; {FUNC Pop}
```

Analysis of the Array Implementation.

to be expanded later

Pointer Implementation.

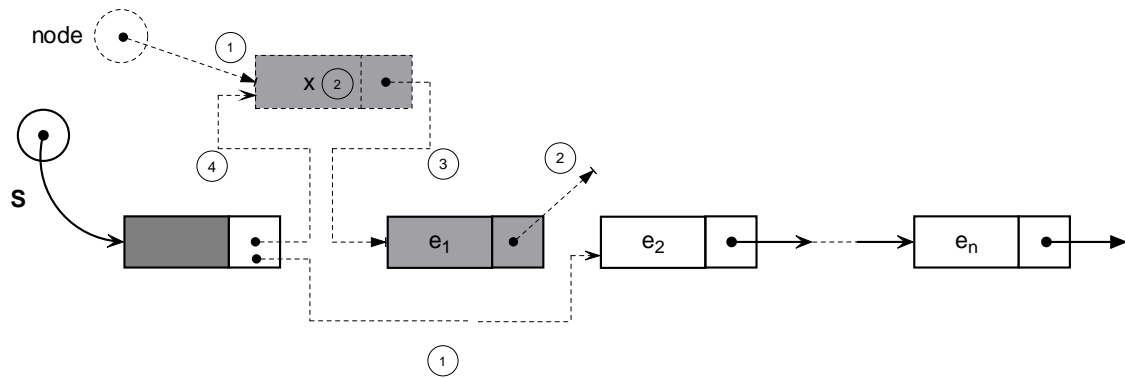


Figure 2.15. Pointer implementation of Stack.

Analysis of the Pointer Implementation.

to be expanded later

2.5 The ADT Queue

DEFINITION (*Queue*).

- Set and Structure : ADT List with insertions at one end and deletions at the other end of the list, called the *Back* and *Front*, respectively.
- Operations :
 1. $Create(Q') \rightarrow$ an empty list, $Q'' = ()$
 2. $Empty(Q') \rightarrow$ true or false, $Q'' = Q'$.
 3. $EnQueue(x, Q') \rightarrow Q''$, where $Q' = (e_1, e_2, \dots, e_n)$ and $Q'' = (e_1, e_2, \dots, e_n, x)$
 4. $DeQueue(Q') \rightarrow e_1$, where $Q' = (e_1, e_2, \dots, e_n)$ and $Q'' = (e_2, \dots, e_n)$
 5. $Front(Q') \rightarrow e_1, Q'' = Q'$.
 6. $Back(Q') \rightarrow e_n, Q'' = Q'$.
- Implementation : Array or linked.

□

2.5.1 Applications of Queues

The following algorithm uses a queue to traverse a graph. Note that this is the same as the Depth First Search algorithm except that a queue replaces the stack. This results in a different traversal order.

algorithm $BFS(s:node, G:graph)$

```

Mark all nodes  $u \in G$  'unvisited'
Create(Q)
EnQueue(s,Q)
WHILE NOT Empty(Q) DO
  u := DeQueue(Q)
  Visit(u)
  FOR each  $v \in Adj(u)$  DO
    IF NOT visited(v) AND  $v \notin Q$  THEN
      p[v] := u
      EnQueue(v,Q)
    ENDFOR
  ENDWHILE
endalg BFS

```

2.5.2 Implementations of Queues

Array Implementation.

Analysis of the Array Implementation.

Pointer Implementation.

Analysis of the Pointer Implementation.

to be expanded later

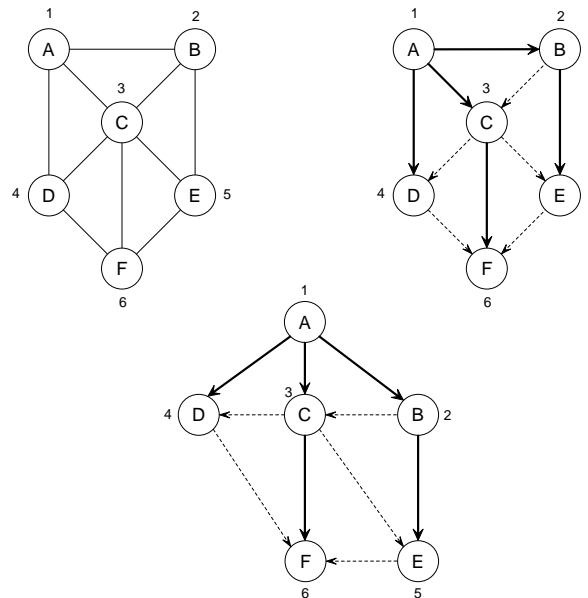


Figure 2.16. Breadth First Search and its Tree.

2.6 The ADT Deque

DEFINITION (*Deque*).

- Set and Structure : ADT List with insertions and deletions at the both ends of the list, called the *Back* and *Front*, respectively.
- Operations :
 1. $Create(Q') \rightarrow$ an empty list, $Q'' = ()$
 2. $Empty(Q') \rightarrow$ true or false, $Q'' = Q'$.
 3. $EnQueueB(x, Q') \rightarrow Q''$, where $Q' = (e_1, e_2, \dots, e_n)$ and $Q'' = (e_1, e_2, \dots, e_n, x)$
 4. $EnQueueF(x, Q') \rightarrow Q''$, where $Q' = (e_1, e_2, \dots, e_n)$ and $Q'' = (x, e_1, e_2, \dots, e_n)$
 5. $DeQueueF(Q') \rightarrow e_1$, where $Q' = (e_1, e_2, \dots, e_n)$ and $Q'' = (e_2, \dots, e_n)$
 6. $DeQueueB(Q') \rightarrow e_n$, where $Q' = (e_1, e_2, \dots, e_n)$ and $Q'' = (e_1, e_2, \dots, e_{n-1})$
 7. $Front(Q') \rightarrow e_1, Q'' = Q'$.
 8. $Back(Q') \rightarrow e_n, Q'' = Q'$.
- Implementation : Array or linked.

□

2.6.1 Applications of Deques

algorithm *Pape-SPathTree* (r, G, p, D)

```

{S is represented as an output-restricted Deque}
Initialize ( $r, G, p, D$ )
 $S := \text{empty}; D[r] := 0; p[r] := \text{null}$ 
 $u := r$ 
while  $u \neq \text{null}$  do
  for each  $v \in \text{Adj}(u)$  do
    if  $D[v] > D[u] + d_{uv}$  then
      if  $v \notin S$  then
        if  $D[v] = \infty$  then  $\text{EnQueueB}(v, S)$ 
        else  $\text{EnQueueF}(v, S)$ 
         $D[v] := D[u] + d_{uv}$ 
         $p[v] := u$ 
      endif  $\{v\}$ 
    endif  $\{D[v]\}$ 
  endfor
   $u := \text{DeQueue}(S)$ 
endwhile
endalg Pape-SPathTree.

```

2.6.2 Implementations of Deques

Array Implementation.

Pointer Implementation.

to be expanded later

2.7 Cursor Implementation of Linked Lists

Some older languages such as Fortran 77 and Basic do not provide dynamic memory allocation and pointers and so we cannot directly implement linked lists in them. However, we can *emulate* dynamic memory allocation and hence implement linked lists using this emulation.

There are two ways to emulate dynamic memory allocation

1. An array of records with k fields, or
2. k parallel arrays that emulate an array of records.

We will use k parallel arrays because some languages do not have records. The array-of-records emulation can be obtained by an almost-mechanical translation of the two-array emulation.

Dynamic Memory Space or Heap. For simplicity we assume that only nodes of one type need to be dynamically allocated. We choose the two-field node with an element and a link field. We emulate the dynamic memory of 2-field nodes by two parallel arrays, where one array stores elements and the other stores links.

Initially we set up the arrays as shown in Figure 17. We may view this dynamic memory space as a *List of Available Space* as shown in Figure 18.

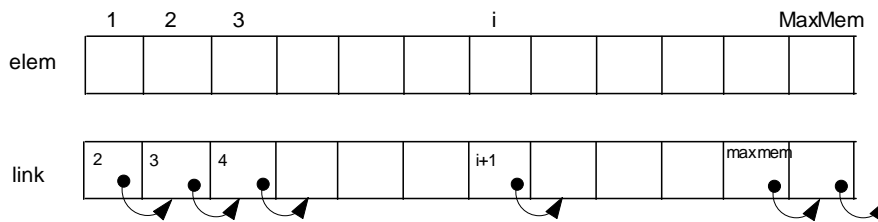


Figure 2.17. Initial Dynamic Memory.

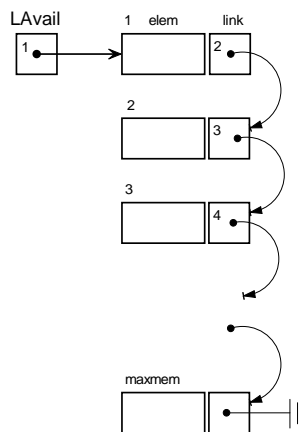


Figure 2.18. Initial List of Available Space.

```

UNIT DynaMem;{Uses two parallel arrays to emulate dynamic memory allocation}


---


INTERFACE


---


USES ElemDefU;           {supplied by the user to define the element type}
CONST
  null    = 0;
  maxmem  = 10000;
TYPE
  nodeptrtype = null..maxmem;


---


PROCEDURE NewNode (VAR p:nodeptrtype);
PROCEDURE DisposeNode (VAR p:nodeptrtype);
PROCEDURE SetElem (x:elementype; p:nodeptrtype);
PROCEDURE SetLink (val:nodeptrtype; p:nodeptrtype);
FUNCTION  GetElem (p:nodeptrtype):elementype;
FUNCTION  GetLink (p:nodeptrtype):nodeptrtype;


---


IMPLEMENTATION


---


VAR
  LAvail : nodeptrtype;
  elem   : ARRAY[nodeptrtype] OF elementype;
  link   : ARRAY[nodeptrtype] OF nodeptrtype;


---


PROCEDURE NewNode( VAR p:nodeptrtype );
BEGIN
  p := LAvail;                               { 1 }
  IF LAvail <> null THEN BEGIN
    LAvail := link[LAvail];                 { 2 }
    link[p] := null;                        { 3 }
  END; { IF }
END; {PROC NewNode }


---


PROCEDURE DisposeNode( VAR p:nodeptrtype );
BEGIN
  link[p] := LAvail;                         { 1 }
  LAvail := p;                               { 2 }
END; { PROC DisposeNode }


---


PROCEDURE SetElem (x:elementype; p:nodeptrtype);
BEGIN
  elem[p] := x;
END; { PROC SetElem }


---


PROCEDURE SetLink (val:nodeptrtype; p:nodeptrtype);
BEGIN
  link[p] := val;
END; { PROC SetLink }


---


FUNCTION  GetElem (p:nodeptrtype):elementype;
BEGIN
  GetElem := elem[p];
END; { PROC GetElem }


---


FUNCTION  GetLink (p:nodeptrtype):nodeptrtype;
BEGIN
  GetLink := link[p];
END; { PROC GetLink }


---


VAR
  i : nodeptrtype;
BEGIN
  {Initialization Code}
  FOR i := 1 TO MaxMem DO BEGIN
    link[i] := i+1;
  END;
  link[MaxMem] := null;
  LAvail := 1;
END. { UNIT DynaMem }

```

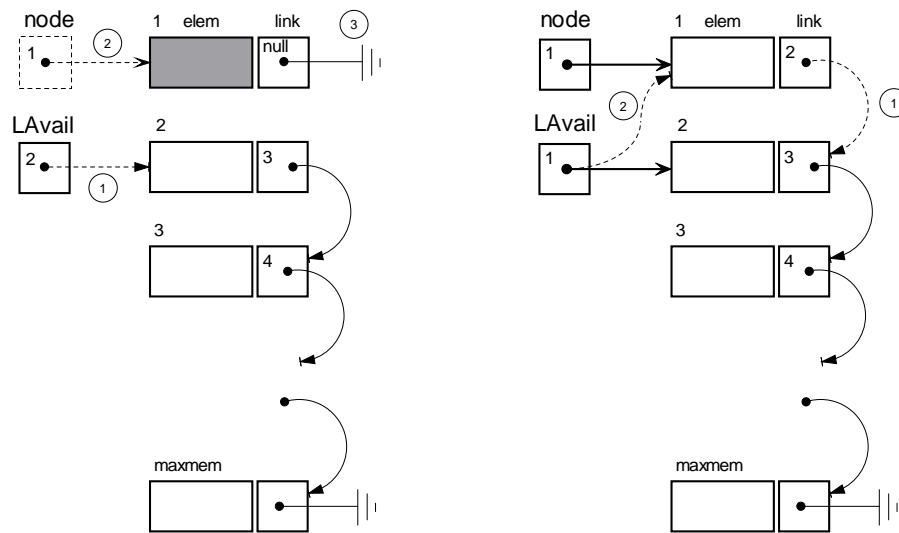


Figure 2.19. NewNode and DisposeNode.

The code above is a translation of the diagrams in Figures 17, 18, and 19. Although this code is mainly self-explanatory a few comments are in order.

1. The *NewNode* procedure returns null if the available space is exhausted. Hence the user must check for this condition each time the *NewNode* procedure is used. This is a somewhat arbitrary design decision : we could have called an error procedure and let this handle the condition.
2. Notice that *maxmem* is visible to the user in the interface. This tells the user how many nodes can be allocated. Also, this constant allows us to define `nodeptr` type which is safer than declaring it as `INTEGER`.