SECTION II

DATA STRUCTURES

Bad programmers worry about the code.
Good programmers worry about data structures and their relationships.

— Linus Torvalds

CHAPTER 5

Basic Data Structures

OBJECTIVES

After studying this chapter, the reader will be able to

- Understand the concept of data structures
- Appreciate the importance of abstract data structures
- Understand and use arrays and implement various operations such as search and traverse
- Understand the linked list and various operations on a singly linked list
- Differentiate between a doubly linked list and a circular linked list
- Understand the concept of stacks
- Implement stacks using arrays and linked list
- Understand the concept of queues
- Implement queues using arrays and linked list
- Understand the importance of data structures vis-à-vis algorithms

5.1 INTRODUCTION

For most of the readers, the primary aim of reading this book is to become an accomplished programmer or an accomplished algorithm designer. It is important to understand the concept of data structures in order to do that. It is essential to be able to develop algorithms, analyse them, and make them as efficient as possible. This efficiency can be attained in two ways, either by the techniques explained in the following sections or by using efficient data structures. This chapter concentrates on the latter. In order to make things happen efficiently, it is essential to organize the data so that its retrieval becomes easy. This branch of computer science deals with not just the organization of data and its retrieval, but also the processing alternatives. Finally, in order to become a good programmer, one must understand both data structures and algorithms.

Even the basic data types, provided by a language, such as int, float, and char, are considered as data structures. They are called *primal data structures*. The more intricate ones such as arrays, stacks, and linked lists described in the following sections are called *non-primal data structures*. The latter can be linear as in the case of a stack or a queue or can be non-linear like a graph or a tree. Trees and graphs have been discussed in Chapters 6 and 7, respectively, of this book.

Each data structure comes with standard operations such as create, delete, update, and traverse. Let us now explore the tremendous world of data structures. In the journey, one must try to empower oneself with the immense powers of the data structures so that the war of algorithms can be fought and won.

5.2 ABSTRACT DATA TYPES

Abstract data types (ADTs) refer to the abstraction of certain class of types that have similar behaviour. Such data structures are defined by the operations that can be performed on the above said class. At times, the restrictions on such operation(s) are also stated along with the ADT. One of the classical examples of an ADT is stack. Stack is a linear data structure that follows the principle of last in first out. The behaviour of a stack may be defined by push, pop, underflow, and overflow operations. The push operation inserts an element into the stack at the position indicated by the value of TOP. The pop operation takes out an element from the stack and decrements the value of TOP. The overflow operation checks if the value of TOP increments MAX −1, where MAX is the maximum number of elements that can be stored in the stack. The underflow operation, on the other hand, checks if the pop operation is running when the value of TOP is −1, that is, there is no element in the array and still pop is being invoked. The stack ADT has been implemented using both arrays and linked list in the following sections. In the above discussion, the various operations that can be performed on a stack and the constraints on the operations have been discussed. The above discussion also brings forth the point that an ADT can be implemented using various data types.

Though there is no regular convention for defining them, the ADTs find their application even in the fascinating field of Artificial Intelligence wherein groups, rings, etc. are described using ADT. ADTs are also an important ingredient of Object Oriented Programming (OOP), wherein they are widely used in design by contract.

The above approach not only provides flexibility and facilitates change as needed but also gives a way to understand the concept of abstraction in OOP.

5.3 ARRAYS

An array is a linear data structure, wherein homogeneous elements are located at consecutive locations. An array can be of any standard data type; for example, an integer array cannot store a character and a string array would not store a user-defined structure. Hence, all arrays are said to be homogeneous. Moreover, if an integer type array starts from a location, say 2048, then the first element stored at 2048 would have its neighbour stored at 2050 (assuming that an integer takes 2 bytes of memory). The *i*th element, in this case, would be stored at the location, which can be found by the following formula:

$$\text{Address of } i\text{th element} = \text{Base Address} + (i-1) \times \text{size of (int)}$$

The following discussion explains some of the basic operations on arrays.

5.3.1 Linear Search

An element stored in an array (ITEM), having *n* elements, can be easily found by a simple traversal. This procedure would be henceforth referred to as linear search. Although the algorithm has also been discussed earlier, the following explanation will revisit the concept and will help the user to implement the procedure. The variable FLAG, in Algorithm 5.1, is initially 0, indicating that the value has not been found. If the element is found, then the value of FLAG becomes 1. At the end of the procedure, if FLAG remains 0, it means that the element has not been found. In case the element to be found occurs in the array more than once, the value of FLAG remains 1.

Algorithm 5.1 Algorithm for linear search (Array [] a, int n, int ITEM)

```
{
    FLAG = 0;
    int i = 0;
    while (i<n)
        {
        if (a[i] == ITEM)
            {
            FLAG = 1;
            print: "FOUND";
            }
            i++;
        }
    if (FLAG == 0)
        {
        Print: 'Not Found';
        }
}
```

Complexity: If the element to be found is present at the first position, then the complexity would be O(1). In the other extreme case, the element may not be present in the given array, or even be present at the last position. In such case, the complexity would be $O(n)$. What matters the most is the average complexity which, in the case of linear search, is $O(n)$.

There is another, more efficient, search technique referred to as binary search, which requires the use of Divide and Conquer. The algorithm has been discussed in Chapter 9.

5.3.2 Reversing the Order of Elements of a Given Array

In order to reverse the order of elements of an array, the following procedure is employed. A loop takes the *i*th element from the beginning and the *i*th element from the end (or the $(n - i - 1)$th element from the beginning) and swaps them.

Algorithm 5.2 Reorder (int[] a)

```
//temp is a temporary variable
//i is the counter
for(i=0; i<n; i++)
     {
     temp = a[i];
     a [i] = a [n-i-1];
     a [n-i-1] = temp;
     }
```

Complexity: Every statement inside the block runs $(n/2)$ times, so the complexity of the above algorithm becomes $O(n)$.

5.3.3 Sorting

The sorting of a given array $[a_1, a_2, a_3, \ldots,]$ produces an array $[b_1, b_2, b_3, \ldots,]$, such that $b_i > b_j$, if $i > j$ (or for that matter $b_i < b_j$, if $i > j$). The concept has been dealt with in Chapter 8 and has been carried forward in Chapter 9.

5.3.4 2D Array

A 2D array contains rows and columns. The base index of rows is 0 and so is that of the first column. There are two ways of storing the elements of a 2D array, the row major or the column major. In the row major technique, the first row elements are stored, followed by those in the second row and so on. A 3×2, 2D array has been depicted as follows:

$$\begin{matrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{matrix}$$

A 2D array has many applications. One of the most important is matrix operations. A matrix is a 2D array. These operations are used in graphics and animations along with many other things. The basic operations like that of addition and subtraction are $O(n^2)$ algorithms. For example, a matrix A of order $m \times n$, when added to another matrix B of the same order, would require the execution of $O(n^2)$ instructions, if $m = n$, otherwise $O(mn)$ (refer to the code that follows). This is because there is a loop within a loop.

```
for (i=0; i<n; i++)
     {
     for(j=0; j<n; j++)
             {
             c[i][j]=0;
             for(k=0; k<n; k++)
                     {
                     c[i][j]+=a[i][k]*b[k][j];
                     }
             }
     }
```

Complexity: The multiplication of two matrices requires three levels of nesting, thus making the complexity as $O(n^3)$. However, the complexity can be greatly reduced by a method explained in Chapter 9 of this book.

5.3.5 Sparse Matrix

A sparse matrix is a special type of matrix in which most of the elements are zero. In such cases, there is hardly any need to store each and every element of the matrix. Only the non-zero elements of the matrix can be stored along with their corresponding row and column numbers. For example, consider the following 5×5 sparse matrix. There are only four elements. The rest of the elements are zero and hence are shown in the following array:

$$\begin{pmatrix} 3 & \cdots & 2 \\ \vdots & \ddots & \vdots \\ 21 & \cdots & 11 \end{pmatrix}$$

The above elements can therefore be stored as follows. Here, the first column depicts the element, the second column depicts the row number, and the third depicts the column number of the element. This method requires only 12 memory locations, as against 25 required to store the whole array.

3	0	0
2	0	4
21	4	0
11	4	4

5.4 LINKED LIST

A linked list is a data structure whose basic unit is a node. Each node has two parts namely: data and link. The data part contains the value, whereas the link part has the address of the next node. This helps to connect various nodes of a list. The last node of the list has NULL in the link part, thus indicating that nothing is attached after the last node. In the discussion that follows the first node would be denoted by FIRST. In C, a linked list can be created using structures, the code of which is written as follows.

```
struct node
    {
    int data;
    struct node * LINK;
    };
```

In the above code, node is a structure having data part, which is of any standard data type and the link, which is a pointer to the node itself. Figure 5.1 depicts the node.

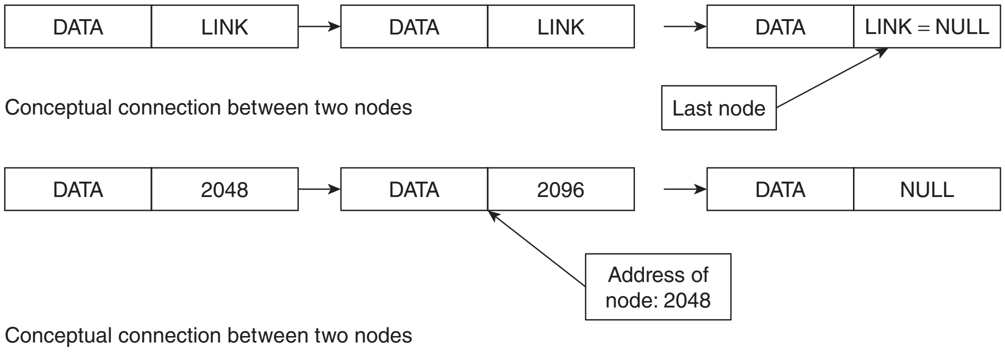


Figure 5.1 Nodes in a linked list

5.4.1 Advantages of a Linked List

Insertion and deletion in a linked list are easier as compared to an array. Moreover, a linked list does not suffer from the problem of limited placeholders as in the case with arrays. Linked lists are flexible, efficient, and provide more functionality as compared to an array. However, the linked list makes use of pointers, which make the implementation of a linked list a bit difficult. Moreover, the linked list algorithms are more involved as compared to that of arrays. However, the advantages of using a linked list are far more than its disadvantages.

5.4.2 Creation of a Linked List

In order to create a linked list, allocate memory to the node (in the case of C, it can be done via malloc()). This is followed by setting the value in the DATA part of the node. If one wants to insert a new node, then the address of the next node is set in the LINK field. If there is just a single node in the LIST, then the LINK of the first node becomes NULL (Fig. 5.2).

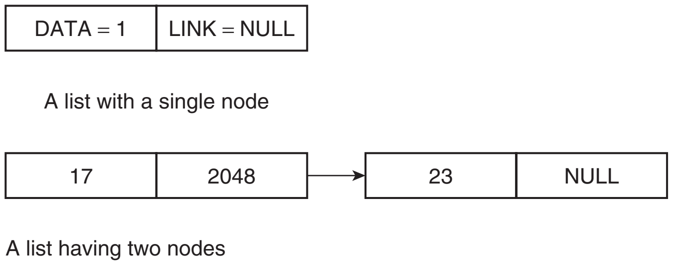


Figure 5.2 A list with a single node and with two nodes

5.4.3 Insertion at the Beginning

In order to insert a node at the beginning of a linked list, the following steps are required:
INPUT: VALUE and LIST

Algorithm 5.3 Insert_beg()

```
{
//Create a node called temp.
     struct node * temp;
Allocate memory to temp.
//Now put the given value (VALUE) in the data part of temp.
     temp->DATA = VALUE;
//Set the LINK part of temp to FIRST.
     temp->LINK = FIRST
Rename temp to FIRST
}
```

Complexity: *O*(1).

The process is depicted in Fig. 5.3. In this figure, a linked list having elements such as 11, 43, and 28 is taken. An element is to be inserted at the beginning having a value = 23.

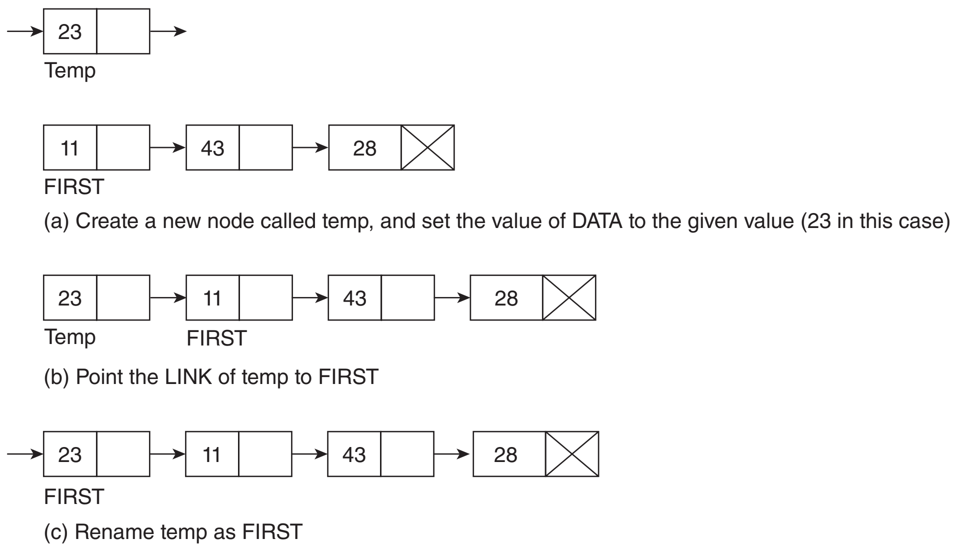
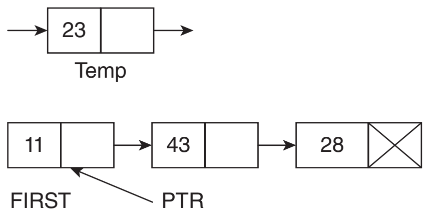

23
Temp
11
43
28
FIRST
(a) Create a new node called temp, and set the value of DATA to the given value (23 in this case)
23
11
43
28
Temp
FIRST
(b) Point the LINK of temp to FIRST
23
11
43
28
FIRST
(c) Rename temp as FIRST

Figure 5.3 Insertion at the beginning of a linked list

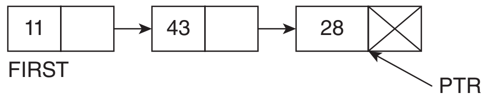
5.4.4 Insertion at End

In order to insert an element at the end, first of all, a pointer, PTR, is set at the beginning. The list is traversed till the LINK of the present node becomes NULL.

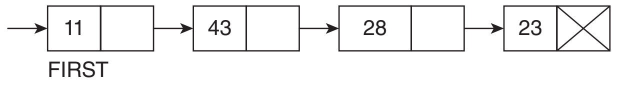
A new node called TEMP is created, the DATA of which is set to the given value. The LINK of PTR is then set to TEMP. The LINK of temp is then set to NULL, indicating that it has become the last node. The process is depicted in Fig. 5.4.



(a) Create a new node called temp, and set the value DATA to the given value (23 in this case), set PTR to FIRST



(b) Traverse till PTR reaches the last node



(c) Set LINK of PTR to temp and LINK of temp to NULL

Figure 5.4 Adding node at the end

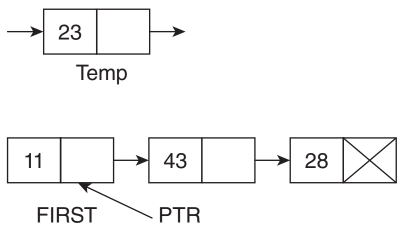
Algorithm 5.4 Insert_end (VALUE)

```
{
PTR = FIRST;
while (PTR -> LINK! = NULL)
     {
     PTR = PTR-> LINK;
     }
Create a new node called TEMP;
SET TEMP->DATA = VALUE;
SET PTR ->LINK = TEMP;
TEMP->LINK = NULL;
}
```

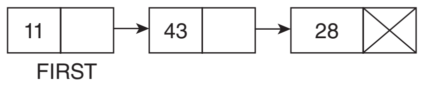
5.4.5 Inserting an Element in the Middle

In order to insert an element after the node having DATA = ‘x’, follow the below steps.

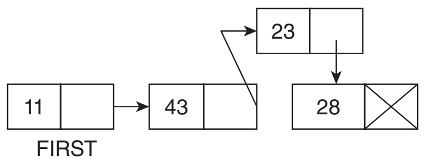
Set PTR to FIRST. Traverse till the DATA of PTR becomes ‘x’. Create a new node called TEMP and set the DATA of Temp to the given value. Now set the LINK of Temp to the next of PTR, and the LINK of PTR to TEMP. The algorithm is given as follows. Figure 5.5 depicts the process.



(a) Create a new node called Temp, and set the value of DATA to the given value (23 in this case), set PTR to FIRST. The value of ‘x’ is 43



(b) When PTR reaches the node having value 43, stop



(c) Set LINK of PTR to Temp and LINK of Temp to PTR

Figure 5.5 Inserting node after a given value

Algorithm 5.5 Insert_middle (VALUE)

```
{
PTR = FIRST;
while (PTR -> DATA! = VALUE)
     {
     PTR = PTR-> LINK;
     }
Create a new node called TEMP;
SET TEMP -> DATA = VALUE;
SET TEMP -> LINK = PTR -> LINK;
SET PTR ->LINK = TEMP;
}
```

Complexity: Since this is a linear algorithm, so the complexity is $O(n)$.

5.4.6 Deleting a Node from the Beginning

In order to delete a node from the beginning, the DATA of FIRST is first stored in a backup. This is followed by renaming the FIRST → LINK (node to which the pointer of FIRST points) to FIRST.

Algorithm 5.6 Delete_beg()

```
    {
    int backup = FIRST->DATA
    Rename (FIRST->LINK) as FIRST;
    }
```

5.4.7 Deleting a Node from the End

In order to delete a node from the end, first of all we traverse till the last but one node. This can be done with the help of a pointer, PTR.

```
PTR = FIRST;
while ((PTR -> LINK)->LINK! = NULL)
    {
    PTR = PTR->LINK;
    }
```

After the above code has been executed, PTR reaches the last but one node.
Now save the value of PTR->LINK in the backup

```
int backup = (PTR->LINK)->DATA;
```

In the final step, the pointer of PTR is set to NULL.

```
PTR->LINK = NULL;
```

The complete algorithm is as follows.

Algorithm 5.7 Delete_end()

```
{
PTR = FIRST;
while ((PTR-> LINK)->LINK != NULL)
    {
    PTR = PTR->LINK;
    }
int backup = (PTR->LINK)->DATA;
PTR->LINK = NULL;
}
```

5.4.8 Deletion from a Particular Point

In order to accomplish the task, the pointer PTR is first set to FIRST. This is followed by traversal till the requisite value ‘x’ is found. Now, the LINK of PTR is set to the (PTR -> LINK -> LINK).

The algorithm is left as an exercise for the reader.

The web resources of this book contain the programs of all the aforementioned operations.

Tip: A stack can be implemented using a linked list by combining two algorithms: insert_end() and delete_end() of a singly linked list. In this case, the element can be inserted only at the end and can be taken out from the end only.

Tip: A queue can be implemented using a linked list by combining two algorithms: insert_end() and end_beg() of a singly linked list. In this case, the element can be inserted only at the end and can be deleted only from the beginning.

5.4.9 Doubly Linked List

The node of a linked list may also have two links namely: PREV and NEXT. Such a linked list is called a doubly linked list.

The insertion and deletion in such a linked list are a bit involved. However, the extra effort pays as this can be used to represent a number of useful data structures like a binary tree. The node of a doubly linked list is shown in Fig. 5.6.

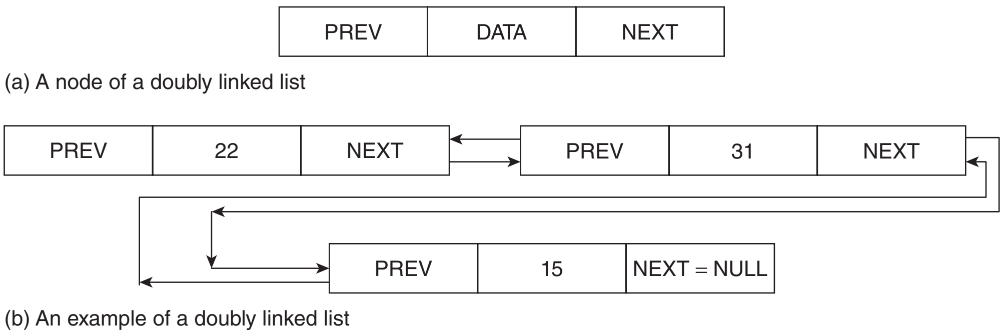

PREV
DATA
NEXT
(a) A node of a doubly linked list
PREV
22
NEXT
PREV
31
NEXT
PREV
15
NEXT = NULL
(b) An example of a doubly linked list

Figure 5.6 An example of a doubly linked list

The premise of insertion of a node at the beginning, middle, and end in a doubly linked list is fundamentally the same as that of a singly linked list. However, in the case of a doubly linked list, both the pointers, PREV and NEXT are to be taken care of. For example, in inserting a node at the beginning, a node called TEMP should be created. The value of the DATA part of TEMP should be set to VALUE (the given value). After this, the LINK of TEMP should be set to the FIRST of the given list. The PREV of the FIRST should be set to TEMP. Since TEMP has to become the FIRST of the list, the PREV of TEMP should be set to NULL. Finally, TEMP should be renamed as FIRST. The algorithm of the above process is given as follows.

Algorithm 5.8 Insert_beg_doubly

```
{
    Set PTR = FIRST;
    Create a new node called TEMP;
    Set TEMP->DATA = VALUE;
    Set TEMP->LINK = PTR;
    Set PTR->PREV = TEMP;
    TEMP-> PREV = NULL;
    FIRST = TEMP;
}
```

In inserting a node at the end, PTR has to traverse till the last node. A new node called TEMP is to be created whose DATA is set to VALUE.

Now, the LINK of PTR is set to TEMP, the PREV of TEMP to PTR, and the LINK of TEMP to NULL. The process is summarized in the following algorithm.

Algorithm 5.9 Insert_end_doubly

```
{
SET PTR = FIRST;
While(PTR -> LINK ! = NULL)
  {
  PTR = PTR->LINK;
  }
Create a new node called TEMP. TEMP-> DATA = VALUE;
Set PTR-> LINK = TEMP;
Set TEMP-> PREV = PTR;
Set TEMP-> LINK = NULL;
}
```

In inserting a node in the middle, PTR is made to traverse till ‘x’ is found. The LINK TEMP is then set LINK PTR. The PREV of the (PTR->LINK) becomes TEMP. The LINK of PTR is then set to TEMP. This is followed by setting the PREV of TEMP to PTR.

While deleting a node from the beginning, the FIRST is simply set to the (FIRST-> LINK) and the second node is renamed as FIRST.

Deletion from the end requires PTR to traverse till the last but one node and then make it LINK as NULL.

Deletion from the middle also follows the same approach as was done in a singly linked list, keeping in mind that PREV pointers are to be catered with.

5.4.10 Circular Linked List

A circular linked list is one in which the LINK of the last node points to the FIRST node. Here, the circular list can be implemented both by a singly linked list and by a doubly linked list. The algorithms pertaining to the circular linked list are left as an exercise for the reader. However, the web resources of this book contain programs pertaining to a doubly linked list.

5.5 STACK

When we pile up our books on our study table, we pick up the book which we had kept at the end, that is, the book which was at the last index is popped first, the initial index being that of the book which was kept initially. The same thing happens when we open the ‘open file’ dialog in MS Word and *via* the ‘open file’ dialog we open the ‘browse’ dialog. Till the browse dialog is not closed, we cannot close the ‘open file’ dialog, and until the ‘open file’ dialog is closed, we cannot close the Word application.

Such data structures that follow the principle of Last In First Out are referred to as a stack. The data structure can be implemented in a static fashion with the help of arrays or in a dynamic fashion with the help of a linked list.

Definition Stack is a linear data structure, which follows the principle of last in first out.

The variable TOP keeps track of the last element of the stack. Initially, the value of TOP in the static implementation of stack is –1. As we add the elements, the value of the variable increases. However, this increase is possible only if the value of TOP is less than a maximum threshold, henceforth be referred as MAX. If an element is added onto a stack, wherein the value of TOP is (MAX –1), then a condition known as ‘Overflow’ occurs.

The deletion of the element, from a stack, is possible if the value of TOP is not –1. If the value of TOP is –1 and the pop operation is invoked, then a condition known as ‘Underflow’ occurs.

Terminology:

MAX:	Maximum number of elements in a stack
TOP:	Initially: –1 Maximum value: MAX –1
push():	A method which increments the value of TOP and inserts value ‘item’ in the stack.
pop():	A method which pops value from the stack and decrements the value of TOP.
Underflow:	Condition wherein pop is invoked and the value of TOP is –1
Overflow:	Condition wherein push is invoked and the value of TOP is (MAX –1)
Starting index:	–1

5.5.1 Static Implementation of Stack

The static implementation of a stack is done with the help of an array. The initial value of the variable TOP is set to –1. The push (int item) function increments the value of TOP and inserts the value onto the stack. Algorithm 5.10 depicts the push function.

push()

Algorithm 5.10 push(int item)

Input: A variable of the type int, array a[ ].
Output: none
Strategy: Increment the value of TOP and insert the item at the index denoted by the new value of TOP.

```
{
if(TOP == MAX-1)
     {
          Print “Overflow”;
     }
else
     {
     TOP++;
     a[TOP]=item;
     }
}
```

pop()

The pop algorithm pops or takes out an element from the top of the stack. The algorithm basically decreases the value of the variable TOP. However, this is possible only if the value of TOP ! = –1.

Algorithm 5.11 pop( )

Input: none
Output: none
Strategy: If the value of TOP is not –1, then pop an item from the top of the stack and decrement the value of TOP by –1.

```
POP ()
if (TOP !=-1)
     {
          TOP--;
     }
else
     {
          print ‘Underflow’;
          }
}
```

5.5.2 Dynamic Implementation of Stack

The dynamic implementation of a stack requires a linked list. The push operation traverses to the end of the list and joins the new node at the end of the list. In order to do so, a pointer PTR, initially at the FIRST of the linked list, traverses till the NEXT of the node is NULL. At the end of this step, PTR would be at the last node of the list.

A new node TEMP is created. The DATA part of the new node would contain the value to be inserted and the NEXT of PTR would be set to NULL. The NEXT of PTR would then point to the new node TEMP. Algorithm 5.12 depicts the dynamic implementation of the push operation.

Algorithm 5.12 push(int item)

```
// A node has two parts DATA and NEXT.
//PTR is a pointer to a node.
//FIRST is the first node of the list
//TEMP is a temporary node.
{
PTR=FIRST;
while( PTR-> NEXT !=NULL)
     {
     PTR = PTR -> NEXT;
     }
//Create a new node called TEMP.
TEMP-> DATA = item;
TEMP->NEXT = NULL;
PTR->NEXT = TEMP;
}
```

The pop operation removes an element from the top of the stack in question. In order to carry out the operation, a pointer to node, PTR, is set to FIRST. The PTR traverses till the NEXT of PTR is NULL. That is, go to the second last node in order to remove the last node. It is like cutting a branch of a tree. In order to cut a branch, we will not sit on the branch, which we intend to cut but on the branch just before that. In order to remove an element from the linked list, we need not to physically separate it, just make the NEXT pointer of the previous node NULL. The following Algorithm 5.13 depicts the implementation of the pop operation.

Algorithm 5.13 pop ( )

Input: none
Output: the element at the TOP of the stack.

```
// A node has two parts DATA and NEXT.
//PTR is a pointer to a node.
//FIRST is the first node of the list
//TEMP is a temporary node.

{
     PTR=FIRST;
     while((PTR-> NEXT)->NEXT !=NULL)
     {
     PTR = PTR -> NEXT;
     }
PTR->NEXT = NULL;
}
```

5.5.3 Applications of Stack

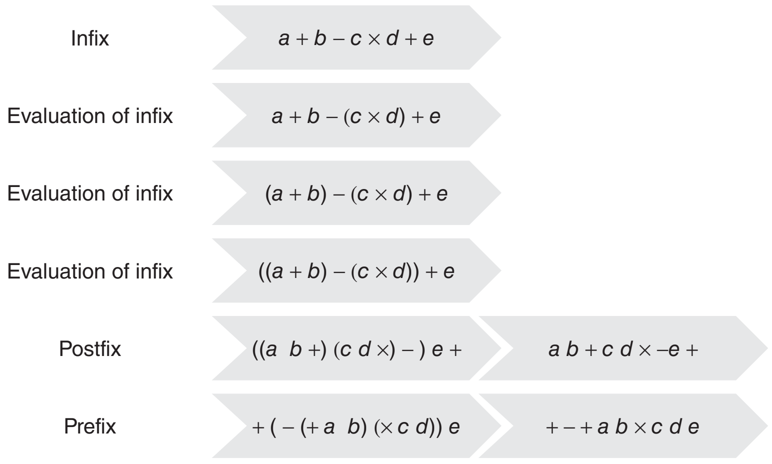
One of the most important applications of stack is conversion of one type of expression into another. An expression can be written in three forms: infix, prefix, and postfix. In the infix form, the operator is written in between the two operands; in the prefix form, the operator is written before two operands, whereas in the postfix form, the operator is written after the two operands. For example, if two numbers stored in variables 'a' and 'b' are to be added and the result is to be stored in a variable called 'c', then the infix expression would be $c = a + b$, the postfix would be $c = ab+$, and the prefix would be $c = +ab$.

A more complex expression would make the things more clear. The second example is as follows:

$$c = a + b - c \times d + e$$

Figure 5.7 shows the evaluation of the expression.

Stacks help us to evaluate postfix expressions as explained in the next section.



Infix
$a + b - c \times d + e$
Evaluation of infix
$a + b - (c \times d) + e$
Evaluation of infix
$(a + b) - (c \times d) + e$
Evaluation of infix
$((a + b) - (c \times d)) + e$
Postfix
$((a\ b +)\ (c\ d \times) -)\ e +$
$a\ b + c\ d \times - e +$
Prefix
$+ (- (+ a\ b)\ (\times c\ d))\ e$
$+ - + a\ b \times c\ d\ e$

Figure 5.7 Evaluation of an expression

5.5.4 Evaluation of a Postfix Expression

Evaluation of a postfix expression refers to finding its value. Stacks can be used to evaluate a postfix expression. The procedure to accomplish the above task is pretty simple. The given expression is scanned from left to right. The symbols scanned are processed as follows. The operands are put into the stack. (The string representing the infix expression is initially set to NULL). When an operator 'op' is encountered, then two symbols 'x' and 'y' are popped from the stack. The expression 'y', 'op', and 'x' is evaluated and put into stack. At the end, whatever is left in the stack is the final answer. In order to understand the above procedure, let us consider the postfix expression in Fig. 5.8.

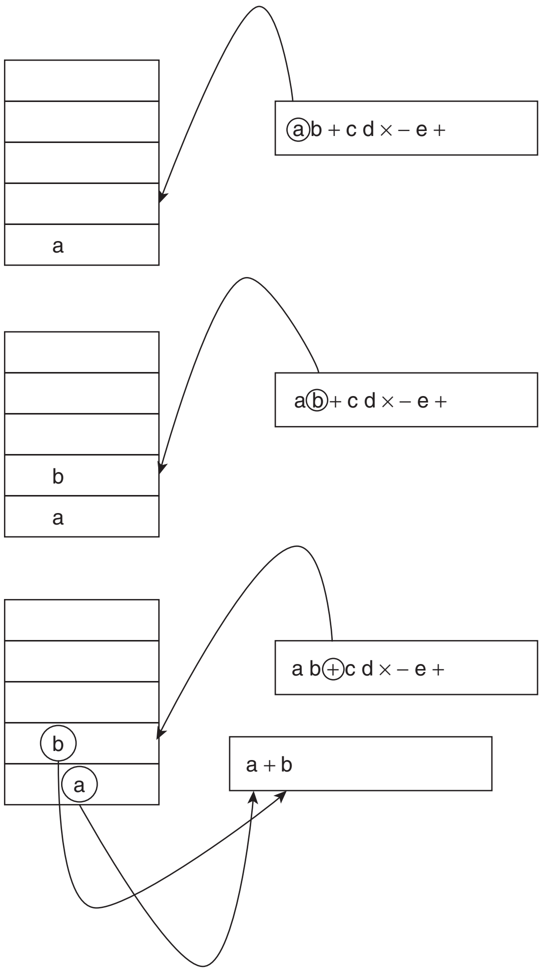


Figure 5.8 Evaluation of a postfix expression (*Contd*)

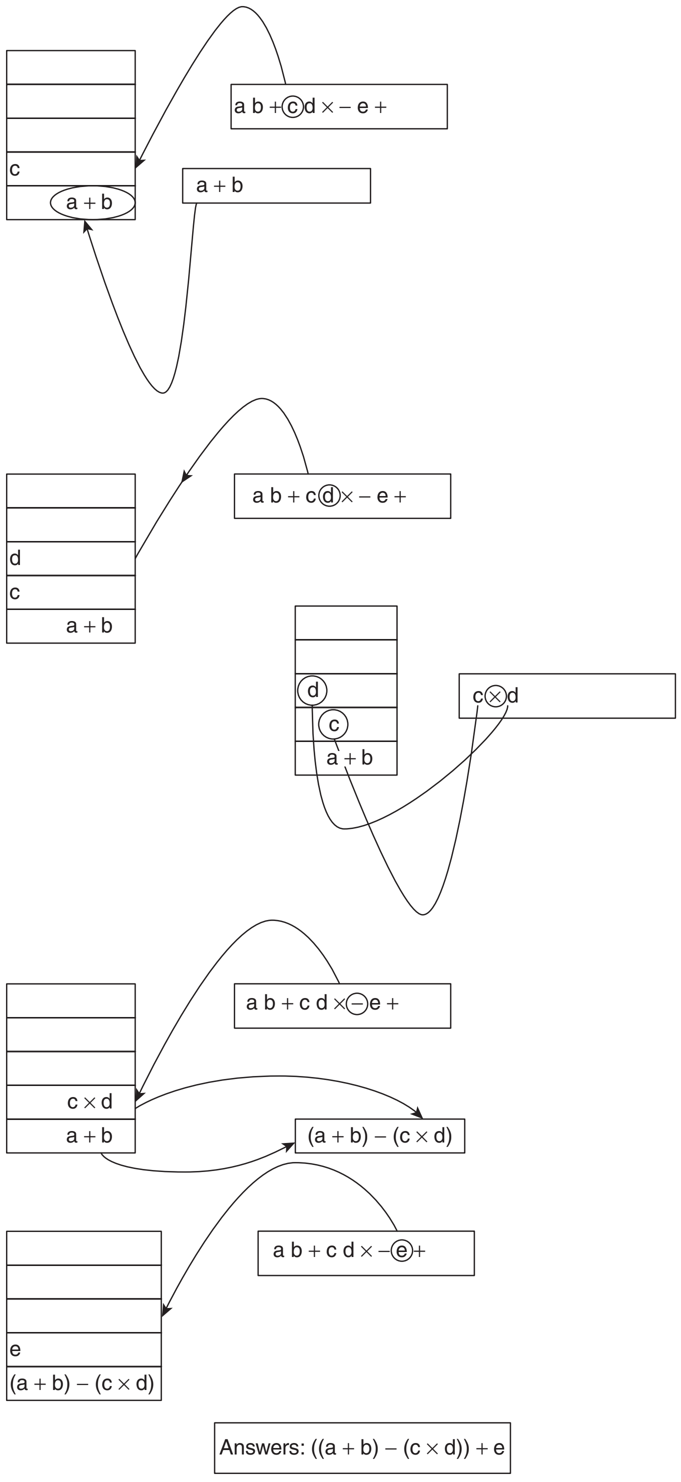


Figure 5.8 (Contd) Evaluation of a postfix expression

5.5.5 Infix to Postfix

Another application of stack is the conversion of an infix expression into postfix. The procedure for converting the expression is as follows.

Algorithm 5.14 InfixToPostfix (E) returns P

Input: Expression E, containing operators, operands, opening parentheses, and closing parentheses
Output: The expression P, which is the postfix of E.

```
{
$P = \varphi$;
Add ‘)’ at the end of E and ‘(‘at the top of the stack;
for each $x \in P$ do
    {
    if (x is an operand)
        {
        add it to P;
        }
    else if (x is ‘(’)
        {
        push it to the stack;
        }
    else if (x is ‘)’)
        {
        y=pop(S);
        while (y!=’(’)
            {
            add y to P;
            y=pop(S);
            }
    else if (x is an operator)
            {
        add it to the stack, S, if the operator on the top of the stack has a
        lower priority;
        otherwise pop the existing operator from stack and add the incoming
        operator at the top of the stack
            }
    }// end of for
return P;
}
```

Complexity: Each symbol is checked and processed. The complexity of the procedure discussed earlier is therefore $O(n)$.

Illustration 5.1 Convert the following expression to postfix:

$$((a + b) - c * d)$$

Solution

$$\text{E: } ((a + b) - c * d)$$
$$P = \varphi$$

Add ‘)’ at the end of E and ‘(’ at the top of the stack;
The processing of the given expression has been shown in the following table.

Symbol Scanned	Stack	P
(	((	NULL
(	(((	NULL
a	(((	a
+	(((+	a
b	(((+	ab
)	((	ab+
–	((–	ab+
c	((–	ab+c
*	((–*	ab+c
d	((–*	ab+cd
)	(	ab+cd*–
)	NULL	ab+cd*–

5.5.6 Infix to Prefix

Another application of stack is the conversion of an infix expression into the prefix. The procedure is the same as that of converting an infix expression to the postfix except for the fact that the given expression is first reversed and then the procedure is applied. Moreover, the expression obtained after applying the procedure is also reversed to get the final answer. The procedure for converting the expression is as follows.

Algorithm 5.15 InfixToPrefix (E) returns P

Input: Expression E containing operators, operands, opening parentheses, and closing parentheses.
Output: The expression P, which is the prefix of E.

```
{
$P = \varphi$;
E = Expression obtained by reversing the order of symbols in E;
Add ‘)’ at the end of E and ‘(‘ at the top of the stack;
for each $x \in P$ do
    {
    if (x is an operand)
        {
        add it to P;
        }
    else if (x is ‘(’)
        {
        push it to the stack;
        }
    else if (x is ‘)’)
        {
        y=pop(S);
        while (y!=’(’)
            {
            add y to P;
            y=pop(S);
            }
    else if (x is an operator)
            {
        add it to the stack, S, if the operator on the top of the stack has a
        lower priority;
        otherwise pop the existing operator from stack and add the incoming
        operator at the top of the stack
            }
        }
    }// end of for
P = expression obtained by reversing the order of symbols of P;
return P;
}
```

Complexity: Each symbol is checked and processed. The complexity of the above procedure is therefore $O(n)$.

Illustration 5.2 Convert the following expression to prefix:

$$(a + b) - c*d$$

Solution

E: $(a + b) - c * d$

E obtained by reversing the order of symbols: $d * c -)b + a($

$P = \varphi$;

Add ‘)’ at the end of E and ‘(‘ at the top of the stack;

The processing of the given expression has been shown in the following table:

Symbol Scanned	Stack	P
None	(	NULL
d	(*	d
c	(*	dc
–	(–	dc*
)		dc*–
b		dc*–b
+	+	dc*–b
a	+	dc*–ba
(	+(	dc*–ba
)	+	dc*–ba
		dc*–ba+

P: *dc*–ba+*
P obtained by reversing the order of symbols: +ab–*cd
Answer: +ab–*cd

5.6 QUEUE

Queue is a linear data structure that follows the principle of First In First Out. A queue can be implemented using an array or a linked list. A queue implemented using an array is referred to as *a static* queue whereas those implemented using a linked list is called a *dynamic queue*. Figure 5.9 summarizes the classification.

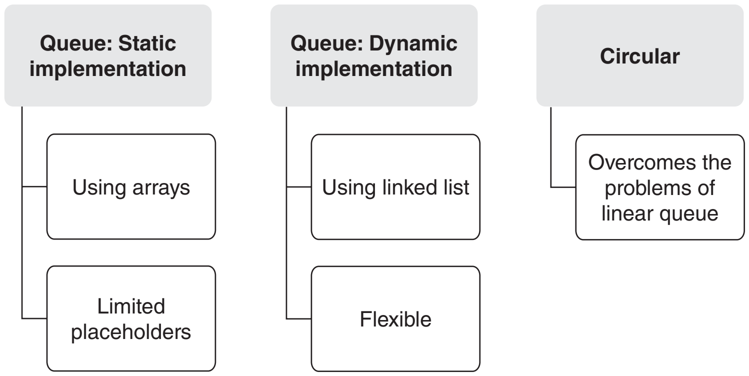

Queue: Static implementation
Using arrays
Limited placeholders
Queue: Dynamic implementation
Using linked list
Flexible
Circular
Overcomes the problems of linear queue

Figure 5.9 Queues and their types

5.6.1 Static Implementation

A queue implemented using an array has two indicators FRONT and REAR. In an empty queue, both the FRONT and REAR are –1. When the first element is inserted in the queue, both FRONT and REAR become 0. On further insertion, the value of REAR increments and the new values are added to the new position. However, this is not possible if the value of REAR is MAX −1, where MAX is the maximum number of elements, an array can have. If one tries to enter an element in a queue whose REAR is MAX −1, then a condition referred to as Overflow is raised, indicating that no more elements can be added to the queue.

On deleting an element from a queue, the deletion must be from FRONT, hence the value of FRONT increments by 1. However, this is not possible if the queue is empty, that is FRONT=REAR= −1 (Underflow Condition).

The algorithms for the insertion and deletion from a queue are as follows.

Algorithm 5.16 Insert_Queue(int VALUE)

```
    {
    if (REAR = MAX-1)
        {
        print 'Overflow';
        }
    else if(FRONT = REAR = -1)
        {
        FRONT = REAR = 0;
        Queue[REAR] = VALUE;
        }
    else
    {
    REAR ++;
    Queue[REAR ] = VALUE;
    }
}
```

Algorithm 5.17 Delete_Queue()

```
    {
    if(FRONT = REAR = –1)
        {
        print: 'Underflow';
        }
    else if (FRONT = REAR)
        {
        FRONT = REAR = –1;
        }
    else
        {
        FRONT ++;
        }
    }
```

Tip: The dynamic implementation of a queue can be done by insert_end() and delete_begin() algorithms of linked lists (refer to Section 5.4).

5.6.2 Problems with the Above Implementation

The problem with the above implementation is that the Overflow condition may be raised even when the queue has some empty cells. For example, consider the situation depicted in Fig. 5.10.

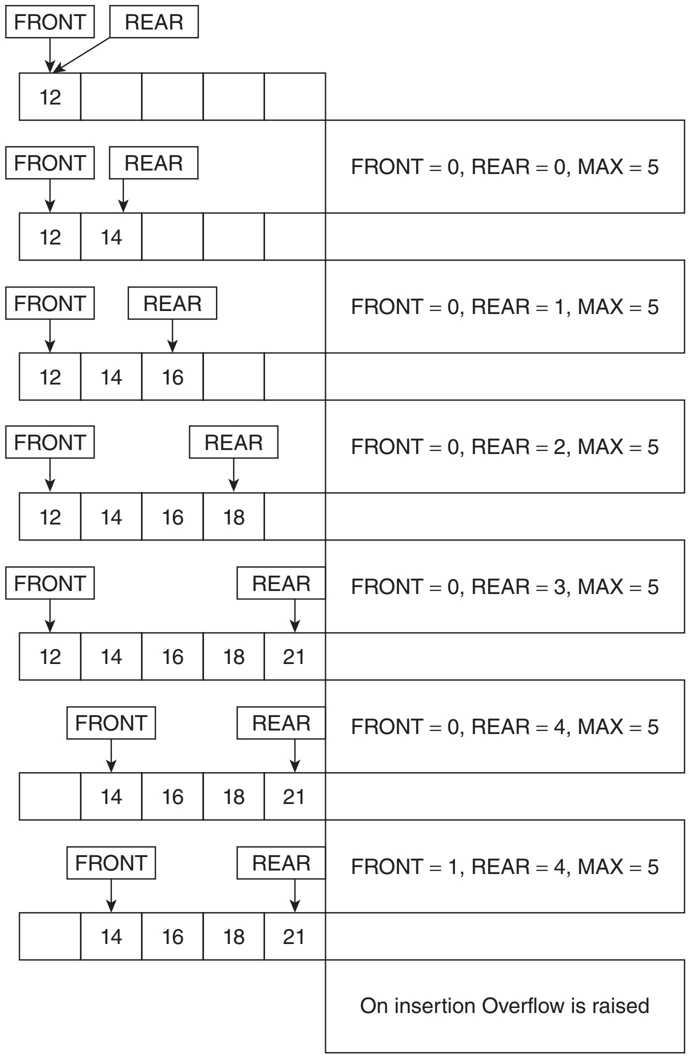

FRONT
REAR
12
FRONT = 0, REAR = 0, MAX = 5
12 14
FRONT = 0, REAR = 1, MAX = 5
12 14 16
FRONT = 0, REAR = 2, MAX = 5
12 14 16 18
FRONT = 0, REAR = 3, MAX = 5
12 14 16 18 21
FRONT = 0, REAR = 4, MAX = 5
14 16 18 21
FRONT = 1, REAR = 4, MAX = 5
14 16 18 21
On insertion Overflow is raised

Figure 5.10 Example of insertion and deletion in a queue

In the figure, 12, 14, 16, 18, and 21 are added to a queue having MAX = 5. After which, 12 is deleted. The queue has empty place. However, on insertion, the Overflow condition is raised as the value of REAR is MAX – 1.

5.6.3 Circular Queue

The above problem can be handled by what is called a circular queue. In a circular queue, the REAR is connected to the 0th index of the array. This makes way for the element entering the array if the value of REAR is MAX – 1 and FRONT is not 0. However, the following points must be observed in the reference to a circular queue as against a linear queue.

In a circular queue, if there is just one element then FRONT = REAR. This condition is the same as that of a linear queue. In a circular queue, the algorithm for insertion increments the value of REAR but also takes its Mod with (MAX), so that if the value of REAR is MAX – 1, it becomes 0. The algorithm for insertion of an element in a circular queue is as follows.

Algorithm 5.18 Insert_Circular_Queue ()

```
{
  if (REAR = FRONT =-1)
         {
         REAR = FRONT = 0;
         Circular_Queue[REAR] = VALUE;
         }
  else if (FRONT = (REAR +1) % MAX)
         {
         Print ‘Overflow’;
         }
  else
         {
         REAR = (REAR + 1 )% MAX;
         Circular_Queue[REAR] = VALUE;
         }
}
```

The algorithm for deletion is given as follows.

Algorithm 5.19 Delete_Circular_Queue()

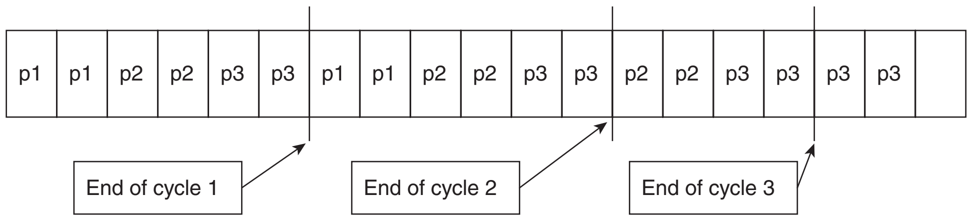
```
{
if (REAR = FRONT =-1)
```

```
     {
     Print 'Underflow';
     }
else if (REAR = FRONT)
     {
     REAR = FRONT = -1;
     }
else
     {
     FRONT = (FRONT + 1)% MAX;
     }
}
```

5.6.4 Applications of a Queue

The queue data structure finds applications in many fields. A few of them are listed here.

1. In a printer, the print commands are queued and hence the command, which was given first, is printed first followed by the next one. This is also referred to as *spooling*.
2. The operating system assigns the CPU to different processes. There can be many ways of doing this. One of the most common ways is a technique called Round Robin technique. In this technique, the CPU is allotted to different processes for a fixed time slot. For example, the slice (time) is two units and the four processes take six units, four units, and eight units, respectively. Then, the CPU executes the first process for the first two units, then it goes to the second process for two units, and then to the third process. The second cycle also proceeds like this. In the third cycle, the processor goes to the first, second, and then the third process. In the next cycle, only third process remains. The process is depicted in Fig. 5.11.
3. Queues are also used in various customer services.



p1 is the first process, p2 is the second, and p3 is the third process. A block represents a unit of time.

Figure 5.11 Round Robin scheduling

5.7 CONCLUSION

The chapter introduced the concept of data structures, which is the soul of algorithm design. Without the knowledge of data structures, the study of algorithms is like an attempt to attain *Nirvana* without understanding the pains of living beings. The chapter also gave a short description of ADTs.

The chapter explored the types of stacks, their static and dynamic implementations, and their applications. It also introduced the concept of queues and provided an insight into their implementation. Finally, all the algorithms given in this chapter have been implemented in C. We can find their implementation in the web resources of this book. It is advised that before having a look at those implementations, the reader must at least try to implement them. The above concepts as well as the chapters that follow extensively use the concepts explained earlier.

Points to Remember

- Abstract data types (ADTs) refer to the abstraction of certain class of types that have similar behaviour.
- An array is a linear data structure, wherein homogeneous elements are located at consecutive locations.
- The complexity of linear search is $O(n)$.
- The complexity of addition, subtraction of two matrices of order $n \times n$ is $O(n^2)$.
- The complexity of conventional matrix multiplication is $O(n^3)$.
- Most of the elements in a sparse matrix are 0.
- The insertion and deletion in a linked list are easy as compared to an array.
- A stack is used in conversion of expressions (like from infix to postfix), in recursion, etc.
- Round Robin algorithm of CPU scheduling uses a queue.
- The static implementation of a stack is done using an array and the dynamic implementation is done using a linked list.
- Problem of having empty spaces and still showing 'overflow' in a linear queue can be solved by using a circular queue.
- If a stack has n elements then the space complexity is $O(n)$ and the time complexity of each operation is $O(1)$.
- The complexity of insertion or deletion at the beginning in the linked list is $O(1)$.
- The complexity of insertion at the end in a linked list is $O(n)$.
- The complexity of deletion from the end in a linked list is $O(n)$.
- The complexity of insertion or deletion in the middle of a linked list is $O(n)$.

KEY TERMS

Circular linked list A circular linked list is one in which the NEXT of the last node points to the FIRST node.
Circular queue In a circular queue, the REAR is connected to the 0th index of the array.
Doubly linked list Each node in a doubly linked list has two links namely: PREV and NEXT.
Linked list A linked list is a data structure whose basic unit is a node. Each node has two parts namely: DATA and LINK. The DATA part contains the value whereas the LINK part has the address of the next node.
Queue A queue is a linear data structure that follows the principle of First In First Out.
Stack It is a linear data structure that follows the principle of Last In First Out.

EXERCISES

I. Multiple Choice Questions

1. Which of the following follows the principle of First In First Out (FIFO)?
   (a) Stack (c) Tree
   (b) Queue (d) Graph
2. Which of the following follows the principle of Last In First Out (LIFO)?
   (a) Stack (c) Tree
   (b) Queue (d) Graph
3. Which of the following is a linear data structure?
   (a) Stack (c) Tree
   (b) Graph (d) None of the above
4. Which of the following is a non-linear data structure?
   (a) Stack (c) Tree
   (b) Queue (d) None of the above
5. Which of the following are facilitated by a data structure?
   (a) Access of data (c) Manipulation of data
   (b) Organization of data (d) All of the above
6. Which of the following must be defined by an abstract data structure?
   (a) Operations on the data
   (b) Constraints on operations
   (c) Both
   (d) None
7. Which of the following uses a queue?
   (a) Round Robin (c) Martin Robin
   (b) Round Martin (d) Robin Robin
8. Which of the following does not use a stack?
   (a) Round Robin (c) Evaluation of postfix
   (b) Conversion to postfix (d) Conversion to prefix

9. Which of the following is used in the dynamic implementation of a queue?
   (a) Stack (c) Array
   (b) Linked List (d) None of the above
10. Which of the following is the best in terms of flexibility?
    (a) Linked List (c) Array
    (b) Both (d) All of the above

II. Review Questions

1. What is an abstract data type?
2. What is meant by data structures? Classify them.
3. What are the applications of a queue?

III. Application-based Questions

1. Write an algorithm to insert an element in a queue. In addition, write the steps to delete an element.
2. Implement a queue using a linked list.
3. Write an algorithm to implement a stack using array.
4. Write an algorithm to implement a stack using a linked list.
5. Write an algorithm for
   (a) Inserting a node at the beginning of a singly linked list
   (b) Inserting a node at the end of a singly linked list
   (c) Inserting a node after a specific position in a singly linked list
   (d) Deleting a node at the beginning of a singly linked list
   (e) Deleting a node from the end of a singly linked list
   (f) Deleting a specific node from a singly linked list
6. Write an algorithm for
   (a) Inserting a node at the beginning of a doubly linked list
   (b) Inserting a node at the end of a doubly linked list
   (c) Inserting a node after a specific position in a doubly linked list
   (d) Deleting a node at the beginning of a doubly linked list
   (e) Deleting a node from the end of a doubly linked list
   (f) Deleting a specific node from a doubly linked list
7. Write an algorithm for inserting a node in a circular linked list.
8. Write an algorithm for deleting a node from a circular linked list.
9. Write an algorithm to reverse a linked list.
10. Write an algorithm to find the maximum element from a linked list.
11. Write an algorithm to find the minimum element from a linked list.
12. Write an algorithm to sort a linked list.
13. How do you find whether a given list has a cycle or a NULL terminated node?

14. Solve the above problem in $O(n)$ time. (HINT: Explore Floyd's Cycle finding algorithm).
15. Write an algorithm to find the *n*th node from the end in a linked list.

Answers to MCQs

1. (b)	3. (a)	5. (d)	7. (a)	9. (b)
2. (a)	4. (c)	6. (c)	8. (a)	10. (a)