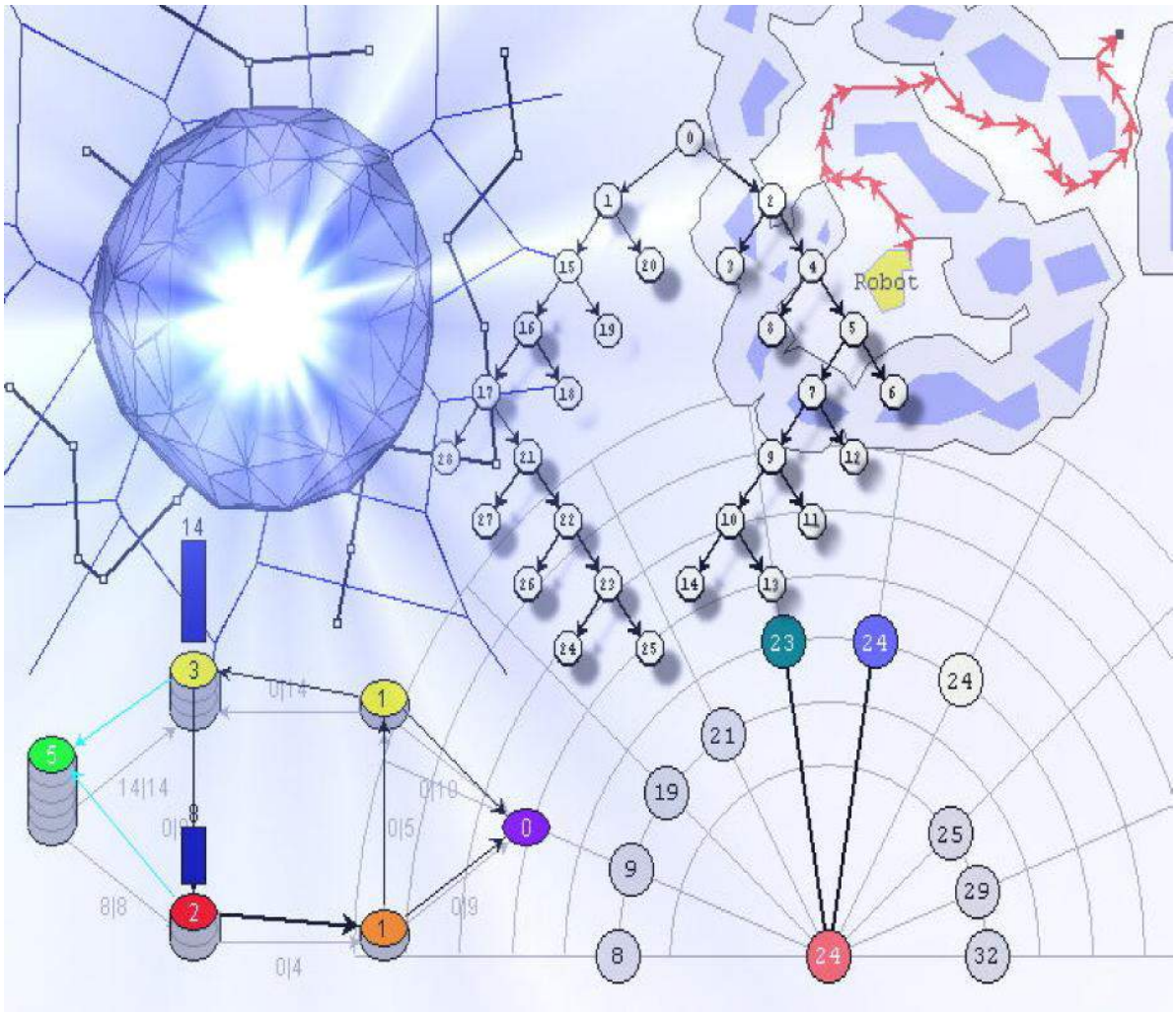
Chatpter 4
Data Structures

Objectives:

- **To understand the concept of data structures**
- **To understand the types of data structures**
- **To understand the operations on different data structures**
- **The implementation of different data structures**



4.1 Introduction

Data is a collection of raw facts that are processed by computers. Data may contain single value or set of values. For example students name may contain three sub items like first name, middle name, and last name, whereas students regno can be treated as a single item. These data have to be processed to form information.

In order to process the data the data should be organized in a particular way. This leads to structuring of data. Utilization of space by data in memory and optimization of time taken for processing, plays an important role in data structures. Data structures are efficient way to organize the data.

Data structures provide a means to manage large amount of data efficiently. Data structures are also key factor in designing efficient algorithms. In this chapter we will discuss about various types of data structures and operations performed on them.

Data structure means organization or structuring a collection of data items in appropriate form so as to improve the efficiency of storage and processing.

> A data structure is a specialized format for organizing and storing data.

4.2 Data representation

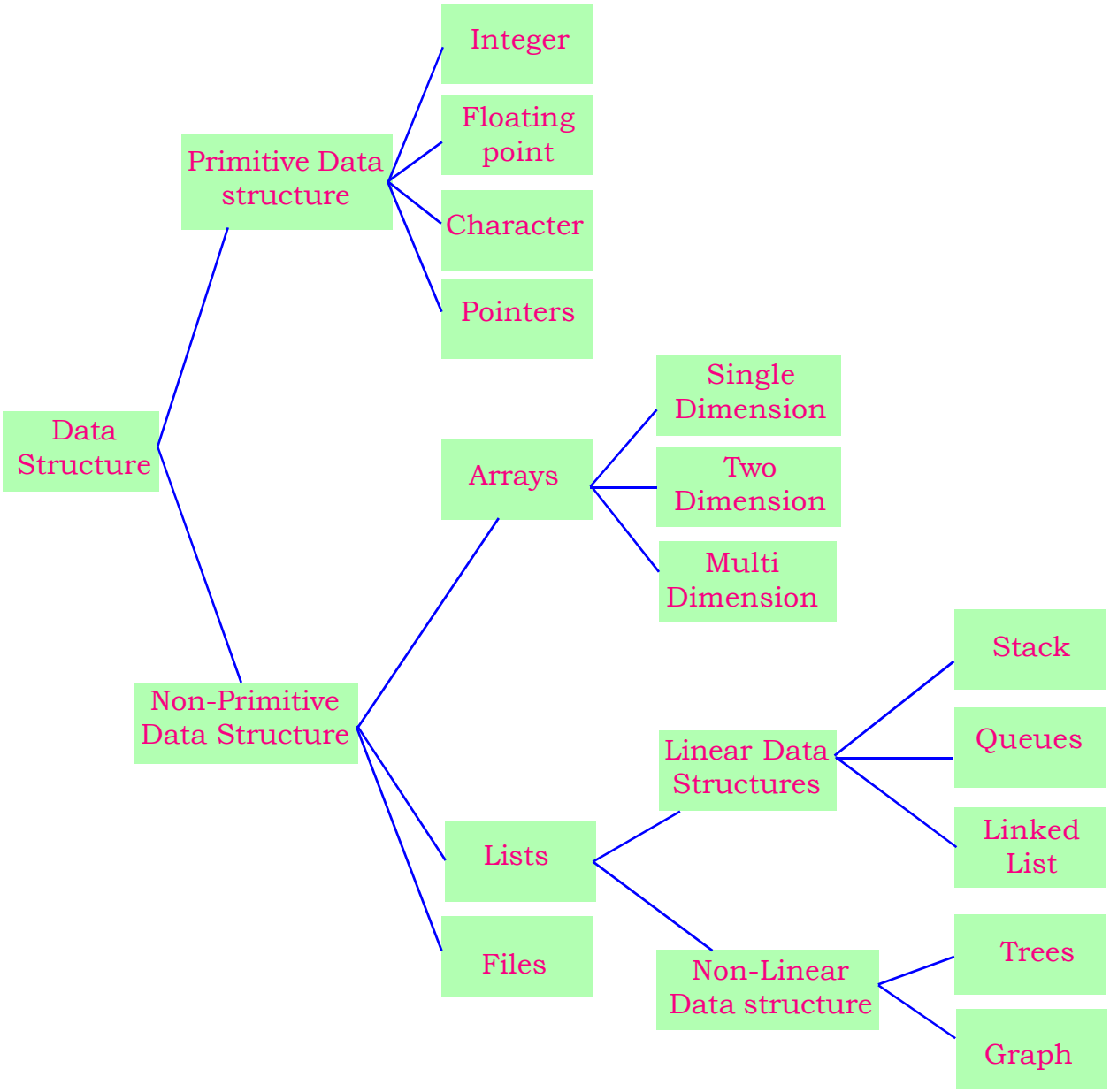
Computers memory is used to store the data which is required for processing. This process is known as data representation. Data may be of same type or different types. A need may arise to group these items as single unit. Data structures provide efficient way of combining these data types and process data.

Data structure is a method of storing data in memory so that it can be used efficiently. The study of data structure mainly deals with:

- Logical or mathematical description of structure.
- Implementation of structure on a computer.
- Analysis of structure which includes determining amount of space in memory and optimization of time taken for processing.

We consider only the first and second cases are considered in this chapter.

4.3 classification of data structures



4.3.1 Primitive data structures:

Data structures that are directly operated upon by machine-level instructions are known as primitive data structures.

The integer, real (float), logical data, character data, pointer and reference are primitive data structures.

4.3.2 Operations on primitive data structures

The various operations that can be performed on primitive data structures are:

- **Create:** Create operation is used to create a new data structure. . This operation reserves memory space for the program elements. It can be carried out at compile time and run-time.

  For example, int x;

- **Destroy:** Destroy operation is used to destroy or remove the data structures from the memory space.

  When the program execution ends, the data structure is automatically destroyed and the memory allocated is eventually de-allocated. C++ allows the destructor member function destroy the object.

- **Select:** Select operation is used by programmers to access the data within data structure. This operation updates or alters data.

- **Update:** Update operation is used to change data of data structures. An assignment operation is a good example of update operation.

  For example, int x = 2; Here, 2 is assigned to x.

  Again, x = 4; 4 is reassigned to *x*. The value of *x* now is 4 because 2 is automatically replaced by 4, i.e. updated.

4.3.3 Non primitive data structures:

Non-primitive data structures are more complex data structures. These data structures are derived from the primitive data structures. They stress on formation of groups of homogeneous and heterogeneous data elements.

Non-primitive data structures are classified as arrays, lists and files.

Array is the collection of homogenous elements under the same name. The different types of arrays are one-dimensional, two-dimensional and multi-dimensional.

Data structures under lists are classified as linear and non-linear data structures.

4.3.4 Linear data structure:

Linear data structures are a kind of data structure that has homogenous elements. Each element is referred to by an index. The linear data structures are Stack, Queues and Linked Lists.

Data elements of linear data structures will have linear relationship between data elements. One of the methods is to have linear relationship between elements by means of sequential memory locations. The other method is to have linear relationship between data elements by means of pointers or links.

4.3.5 Non-linear data structure:

A non-linear data structure is a data structure in which a data item is connected to several other data items. The data item has the possibility to reach one or more data items. Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure.

Trees and Graphs are the examples of non-linear data structures.

4.4 Operations on linear data structure

The basic operations on non-linear data structures are as follows:

- **Traversal**: The process of accessing each data item exactly once to perform some operation is called traversing.
- **Insertion**: The process of adding a new data item into the given collection of data items is called insertion.
- **Deletion**: The process of removing an existing data item from the given collection of data items is called deletion.
- **Searching**: The process of finding the location of a data item in the given collection of data items is called as searching.
- **Sorting**: The process of arrangement of data items in ascending or descending order is called sorting.
- **Merging**: The process of combining the data items of two structures to form a single structure is called merging.

4.5 Arrays

An array is a collection of homogeneous elements with unique name and the elements are arranged one after another in adjacent memory location.

The data items in an array are called as elements. These elements are accessed by numbers called as subscripts or indices. Since the elements are accessed using subscripts, arrays are also called as subscripted variables.

4.5.1 Types of Arrays:

There are three types of array:

- One-dimensional Array
- Two-dimensional Array
- Multi-dimensional array

4.5.2 One-dimension Array

An array with only one row or column is called one-dimensional array. It is finite collection of n number of elements of same type such that

- Elements are stored in contiguous locations
- Elements can be referred by indexing

Array can be denoted as: data type Arrayname[size];

Here, size specifies the number of elements in the array and the index (subscript) values ranges from 0 to n-1.

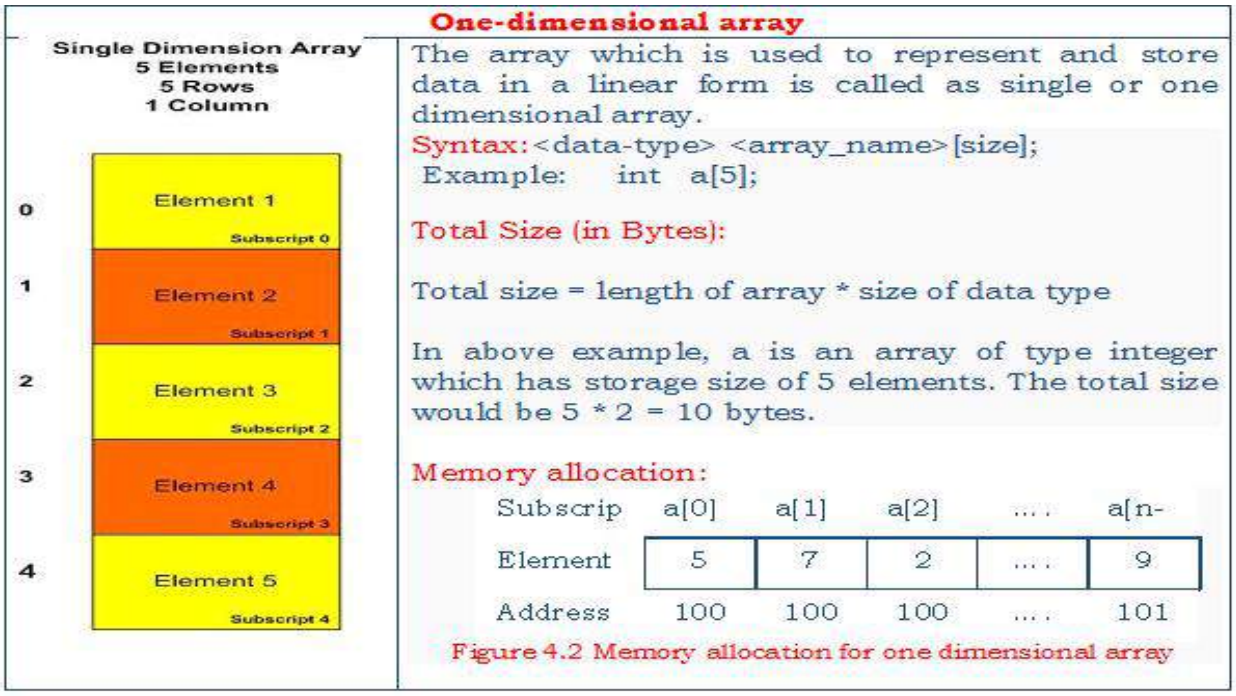


Figure 4.2 Memory allocation for one dimensional array

Features:

- Array size should be positive number only.
- String array always terminates with null character ('\0').
- Array elements are counted from 0 to n-1.
- Useful for multiple reading of elements.

Calculating the length of Array

Arrays can store a list of finite number (n) of data items of same data type in consecutive locations. The number n is called size or length of the array.

The length of an array can be calculated by L = UB – LB + 1

Here, UB is the largest index and LB is the smallest index of an array.

Example: If an array A has values 10, 20, 30, 40, 50, 60 stored in locations 0, 1, 2, 3, 4, 5 then UB = 5 and LB= 0

Size of the array L = 5 – 0 + 1 = 6

4.5.3 Representation of one-dimensional arrays in memory

Elements of linear array are stored in consecutive memory locations. Let P be the location of the element. Address of first element of linear array A is given by Base(A) called the **Base address** of A. Using this we can calculate the address of any element of A by the formula

LOC(A[P]) = Base(A) + W(P – LB)

Here W is the number of words per memory cell.

Example: Suppose if a string S is used to store a string ABCDE in it with starting address at 1000, one can find the address of fourth element as follows:

Address	content	location
1000	A	S[0]
1001	B	S[1]
1002	C	S[2]
1003	D	S[3]
1004	E	S[4]

Now the address of element S[3] can be calculated as follows:

Address(S[3]) = Base(S) + W(P - LB) Here W = 1 for characters

= 1000 + 1(3 - 0)

= 1003

4.5.4 Basic operations on one-dimensional arrays

The following operations are performed on arrays:

1. **Traversing:** Accessing each element of the array exactly once to do some operation.
2. **Searching:** Finding the location of an element in the array.
3. **Sorting:** Arranging the elements of the array in some order.
4. **Insertion:** Inserting an element into the array.
5. **Deletion:** Removing an element from the array.
6. **Merging:** Combining one or more arrays to form a single array.

4.5.5 Traversing a Linear Array

Traversing is the process of visiting each subscript at least once from the beginning to last element.

For example, to find the maximum element of the array we need to access each element of the array.

Algorithm: Let A be a linear array with LB and UB as lower bound and upper bound. This algorithm traverses the array A by applying the operation PROCESS to each element of A.

```
1. for LOC = LB to UB
                PROCESS A[LOC]
   End of for
2. Exit
```

Program: To input and output the elements of the array.

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
void main()
{
        int     a[10], i, n;
        clrscr();
        cout<<"How many elements? ";
        cin>>n;
        cout<<"Enter the elements: ";
        for(i=0; i<n; i++)
                cin>>a[i];
        cout<<"The elements are ";
        for(i=0; i<n; i++)
                cout<<setw(4)<<a[i];
        getch();
}
```

```
How many elements? 5
Enter the elements   5 10 20 15 10
The elements are  5  10  20  15  10
```

4.5.6 Searching an element

It refers to finding the location of the element in a linear array. There are many different algorithms. But the most common methods are linear search and binary search.

Linear Search

This is the simplest method in which the element to be searched is compared with each element of the array one by one from the beginning till end of the array. Since searching is one after the other it is also called as sequential search or linear search.

Algorithm: A is the name of the array with N elements. ELE is the element to be searched. This algorithm finds the location loc where the search ELE element is stored.

```
Step 1:     LOC = -1
Step 2:     for P = 0 to N-1
                if( A[P] = ELE)
                    LOC = P
                    GOTO 3
                End of if
            End of for
Step 3:     if(LOC >= 0)
                    PRINT    LOC
            else
                    PRINT "Search is unsuccessful"
Step 4:     Exit
```

Program: To find the location of an element

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main( )
{
        int    a[50], i, pos, ele, n;
        clrscr( );
        cout<<"Enter the number of elements: ";
```

```
        cin>>n;
        cout<<"Enter the elements: ";
        for(i=0; i<n; i++)
                cin>>a[i];
        cout<<"Enter the search element: ";
        cin>>ele;
        pos=-1;
        for(i=0; i<n ;i++)
        if(ele == a[i])
        {
                pos = i;
                break;
        }
        if(pos>= 0)
             cout<<ele<<" is present at position "<<pos<<endl;
        else
             cout<<ele<<" is not present"<<endl;
        getch( );
}
```

```
Enter the number of elements: 5
Enter the elements: 10 20 50 40 30
Enter the search element: 40
40 is present at position 3
```

Binary Search:

When the elements of the array are in sorted order, the best method of searching is binary search. This method compares the element to be searched with the middle element of the array. If the comparison does not match the element is searched either at the right-half of the array or at the left-half of the array.

Let B and E denote the beginning and end locations of the array. The middle element A[M] can be obtained by first finding the middle location M by M = int(B+E)/2, where int is the integer value of the expression.

If A[M] = ELE, then search is successful. Otherwise a new segment is found as follows:

1. If ELE<A[M] , searching is continued at the left-half of the segment. Reset E = M – 1.
2. If ELE>A[M], searching is continued at the right-half of the segment. Reset B = M + 1.

3. If ELE not found then we get a condition B>E. This results in unsuccessful search.

Algorithm: A is the sorted array with LB as lower bound and UB as the upper bound respectively. Let B, E, M denote beginning, end and middle locations of the segments of A.

```
Step 1:     set B = 0, E = n-1   LOC=-1
Step 2:     while (B <= E)
                M= int(B+E)/2
                if(ELE = A[M]
                        loc = M
                        GOTO 4
                else
                        if(ELE <A[M])
                                E = M-1
                        else
                                B = M+1
            End of while
Step 3:     if(LOC >= 0)
                PRINT   LOC
            else
                PRINT "Search is unsuccessful"
Step 4:     Exit
```

Program: To find the location of an element.

```
#include<iostream.h>
void main()
{
        int a[10], i, n, m, loc, b, e, pos, ele;

        cout<<"How many elements? ";
        cin>>n;

        cout<<"Enter the elements: ";
        for(i=0; i<n; i++)
                cin>>a[i];

        cout<<"Enter the search element ";
        cin>>ele;

        pos=-1;
        b=0;
        e=n-1;

        while(b<=e)
        {
```

```
                m=(b+e)/2;
                if(ele==a[m])
                {
                        pos=m;
                        break;
                }
                else
                        if(ele<a[m])
                                e=m-1;
                        else
                                b=m+1;
        }
        if(pos>=0)
                cout<<"Position= "<<pos;
        else
                cout<<"Search is unsuccessful";
}
```

```
How many elements? 7
Enter the elements: 10 20 30 40 50 60 70
Enter the search element 60
Position= 5
```

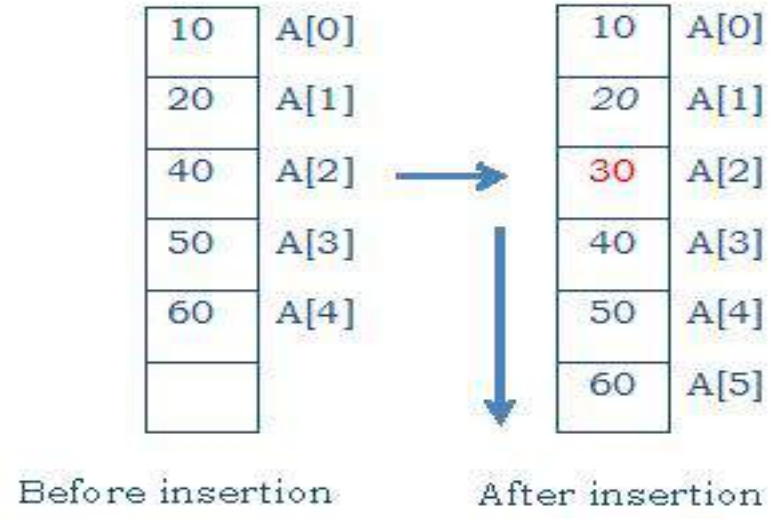
4.5.7 Insertion an element

Insertion refers to inserting an element into the array. A new element can be done provided the array should be large enough to accommodate the new element.

When an element is to be inserted into a particular position, all the elements from the asked position to the last element should be shifted into the higher order positions.

Example: Let A be an array with items 10, 20, 40, 50 and 60 stored at consecutive locations. Suppose item=30 has to be inserted at position 2. The following procedure is applied.

- Move number 60 to position 5.
- Move number 50 to position 4.
- Move number 40 to position 3
- Position 2 is blank. Insert 30 into the position 2. i.e., A[2] = 30.



Algorithm: A is the array with N elements. ITEM is the element to be inserted in the position P.

```
Step 1:    for I = N-1 downto P
                A[I+1] = A[I]
           End of for
Step 2:    A[P] = ITEM
Step 3:    N = N+1
Step 4:    Exit
```

Program: To insert an element into the array.

```
#include<iostream.h>
#include<iomanip.h>
void main()
{
     int a[10], i, n, ele, p;

     cout<<"How many elements? ";
     cin>>n;

     cout<<"Enter the elements: ";
     for(i=0; i<n; i++)
          cin>>a[i];

     cout<<"Enter the element to be inserted: ";
     cin>>ele;

     cout<<"Enter the position ( 0 to "<<n<<"): ";
     cin>>p;

     if(p > n)
```

```
            cout<<"Invalid position";
        else
        {
            for(i=n-1; i>=p; i--)
                a[i+1] = a[i];

            a[p] = ele;
            n = n+1;

            cout<<"The elements after the insertion are ";
            for(i=0; i<n; i++)
                cout<<setw(4)<<a[i];
        }
    }
```

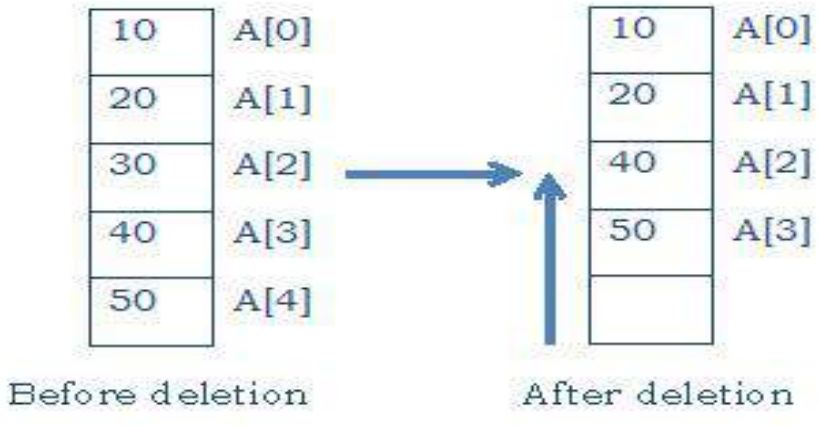
```
How many elements? 5
Enter the elements: 1 2 4 5 6
Enter the element to be inserted: 3
Enter the position ( 0 to 5): 2
The elements after the insertion are 1   2   3   4   5   6
```

4.5.8 Deleting an element from the array

Deletion refers to removing an element into the array. When an element is to be deleted from a particular position, all the subsequent shifted into the lower order positions.

Example: Let A be an array with items 10, 20, 30, 40 and 50 stored at consecutive locations. Suppose item=30 has to be deleted at position 2. The following procedure is applied.

- Copy 30 to Item. i.e., Item = 30
- Move number 40 to position 2.
- Move number 50 to position 3.



Algorithm: A is the array with N elements. ITEM is the element to be deleted in the position P and it is stored into the variable Item.

```
Step 1:     Item = A[P]
Step 2:     for I = P to N-1
                  A[I] = A[I+1]
            End of for
Step 2:     N = N-1
Step 4:     Exit
```

Program: To delete an element from the array.

```
#include<iostream.h>
#include<iomanip.h>
void main()
{
     int a[10], i, n, ele, p;

     cout<<"How many elements? ";
     cin>>n;

     cout<<"Enter the elements: ";
     for(i=0; i<n; i++)
          cin>>a[i];

     cout<<"Enter the position (0 to "<<n-1<<"): ";
     cin>>p;

     if(p > n-1)
          cout<<"Invalid position";
     else
     {
          ele = a[p];
          for(i=p; i<n; i++)
               a[i] = a[i+1];
          n = n-1;
          cout<<"The elements after the deletion is ";
          for(i=0; i<n; i++)
               cout<<setw(4)<<a[i];
     }
}
```

```
How many elements? 5
Enter the elements: 10 20 30 40 50
Enter the position (0 to 4): 2
The elements after the deletion is   10   20   40   50
```

4.5.9 Sorting the elements

Sorting is the arrangement of elements of the array in some order. There are various methods like bubble sort, shell sort, selection sort, quick sort, heap sort, insertion sort etc. But only insertion sort is discussed in this chapter.

Insertion Sort:

The first element of the array is assumed to be in the correct position. The next element is considered as the key element and compared with the elements before the key element and is inserted in its correct position.

Example: Consider the following list of numbers 70 30 40 10 80 stored in the consecutive locations.

Step 1: Assuming 30 in correct position 70 is compared with 30 . Since 30 is less the list now is 30 70 40 10 8 0

Step 2: Now 40 is the key element. First it is compared with 70. Since 40 is less than 70 it is inserted before 70. The list now is 30 40 70 10 80

Step 3: Now 10 is the key element. First it is compared with 70. Since it is less it is exchanged. Next it is compared with 40 and it is exchanged. Finally it is compared with 30 and placed in the first position. The list now is

10 30 40 70 80

Step 4: Now 80 is the key element and compared with the sorted elements and placed in the position. Since 80 is greater than 70 it retains its position.

Algorithm: Let A be an array with N unsorted elements. The following algorithm sorts the elements in order.

```
Step 1:     for I = 1 to N-1
Step 2:         J =  I
                While ( J >= 1 )
                    If( A[J] < A[J-1])
                        temp = A[J]
                        A[J] = A[J-1]
                        A[J-1] = temp
                    If end
                    J = J-1
                While end
            for end
Step 3:     Exit
```

4.5.10 Two-dimensional arrays

A two dimensional array is a collection of elements and each element is identified by a pair of indices called as subscripts. The elements are stored in contiguous memory locations.

We logically represent the elements of two-dimensional array as rows and columns. If a two-dimensional array has m rows and n columns then the elements are accessed using two indices I and J, where 0<=I<=m-1 and 0<=J<=n-1. Thus the expression A[I][J] represent an element present at I^{th}-row and J^{th}-column of the array A. The number of rows and columns in a matrix is called as the order of the matrix and denoted as m x n.

The number of elements can be obtained by multiplying number of rows and number of columns.

	[0]	[1]	[2]
A[0]	2	-2	3
A[1]	1	0	-3
A[2]	2	2	5

In the above figure, the total number of elements in the array would be, rows x columns = 3 x 3 =9-elements.

Representation of 2-dimensional array in memory

Suppose A is the array of order m x n. To store m*n number of elements, we need m*n memory locations. The elements should be in contiguous memory locations.

There are two methods:

- Row-major order
- Column-major order

Row-major order

Let A be the array of order m x n. In row-major order, all the first-row elements are stored in sequential memory locations and then all the second-row elements are stored and so on.

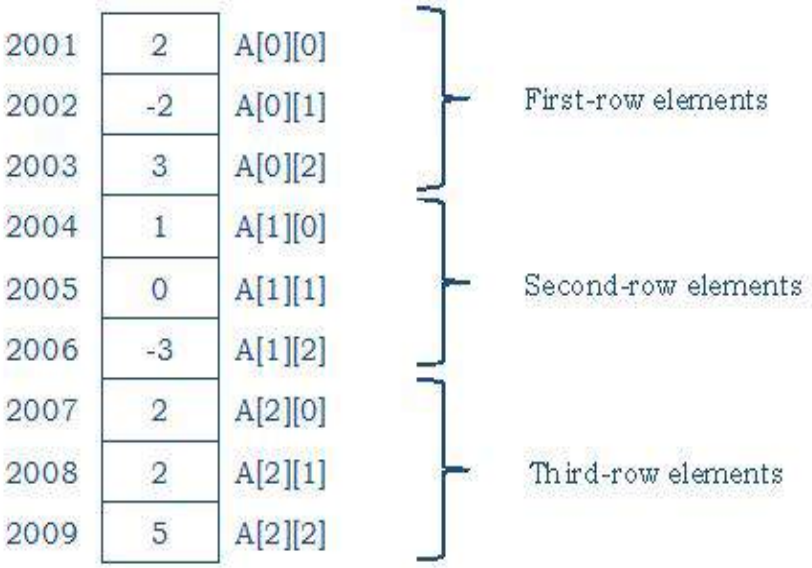
Base(A) is the address of the first element. The memory address of any element A[I][J] can be obtained by the formula

LOC(A[I][J]) = Base(A) + W[n(I-LB) + (J-LB)]

where W is the number of words per memory location.

Example: Consider the array of order 3 x 3.

	[0]	[1]	[2]
A[0]	2	-2	3
A[1]	1	0	-3
A[2]	2	2	5



Column-major order

Let A be the array of order m x n. In column-major order, all the first-column elements are stored in sequential memory locations and then all the second-column elements are stored and so on.

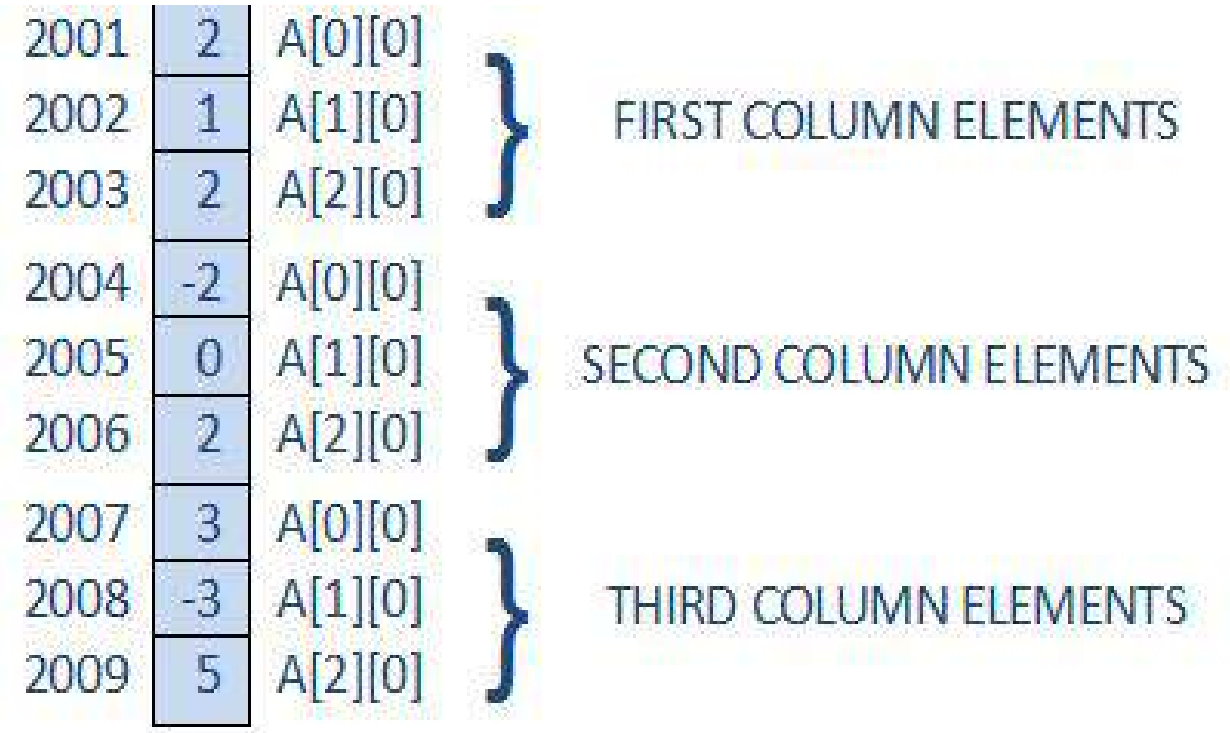
Base(A) is the address of the first element. The memory address of any element A[I][J] can be obtained by the formula

LOC(A[I][J]) = Base(A) + W[(I-LB) + m(J-LB)]

where W is the number of words per memory location.

Example: Consider the array of order 3 x 3.

	[0]	[1]	[2]
A[0]	2	-2	3
A[1]	1	0	-3
A[2]	2	2	5



Program: To read and print the elements in column-major order.

```
#include<iostream.h>
#include<iomanip.h>
void main()
{
     int a[5][5], i, j, r, c;
     cout<<"Enter the order: ";
     cin>>r>>c;
     cout<<"Enter the elements: "<<endl;
     for(i=0; i<c; i++)
          for(j=0; j<r; j++)
               cin>>a[j][i];
     cout<<"The matrix in column-major order is: "<<endl;
     for(i=0; i<r; i++)
     {
          for(j=0; j<c; j++)
               cout<<setw(4)<<a[i][j];
          cout<<endl;
     }
}
```

```
Enter the order: 2 3
Enter the elements:
1 2 3
4 5 6
The matrix in column-major order is:
   1   2   3
   4   5   6
```

Example: Consider the array A of order 25 x 4 with base value 2000 and one word per memory location. Find the address of A[12][3] in row-major order and column-major order.

Solution:

Given Base(A) = 2000, m = 25, n = 4 LB = 0

W = 1, I = 12, J = 3

Row-major order: LOC(A[I][J]) = Base(A) + W[n(I-LB) + (J-LB)]

LOC(A[12][3]) = 2000 + 1[4(12-0)+(3-0)]

= 2000 + 4(12) + 3

= 2000 + 48 + 3

= 2051

Column-major order: LOC(A[I][J]) = Base(A) + W[(I-LB) + m(J-LB)]

LOC(A[12][3]) = 2000 + 1[(12-0)+25(3-0)]

= 2000 + 1(12 + 75)

= 2000 + 87

= 2087

Applications of arrays

1. Arrays are used to implement other data structures such as heaps, hash tables, queues, stacks and strings etc.
2. Arrays are used to implement mathematical vectors and matrices.
3. Many databases include one-dimensional arrays whose elements are records.

Advantages of arrays

1. It is used to represent multiple data items of same type by using only single name.

2. It can be used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.

3. Two-dimensional arrays are used to represent matrices.

Disadvantages of arrays

1. We must know in advance that how many elements are to be stored in array.

2. Array is static structure. It means that array is of fixed size. The memory which is allocated to array cannot be increased or reduced.

3. Since array is of fixed size, if we allocate more memory than requirement then the memory space will be wasted. If we allocate less memory than requirement, then it will create problem.

4. The elements of array are stored in consecutive memory locations. So insertions and deletions are very difficult and time consuming.

STACKS

4.6.1 Introduction

A **stack** is an ordered collection of items where the addition of new items and the removal of existing items always take place at the same end. This end is commonly referred to as the "top". The end opposite to top is known as the **base**.

The base of the stack is significant since items stored in the stack that are closer to the base represent those that have been in the stack the longest. The most recently added item is the one that is in position to be removed first. This ordering principle is sometimes called **LIFO**, **last-in first-out**. Newer items are near the top, while older items are near the base.

Many examples of stacks occur in everyday situations. Almost any cafeteria has a stack of trays or plates where you take the one at the top, uncovering a new tray or plate for the next customer in line. Imagine a stack of books on a desk. The only book whose cover is visible is the one on top. To access others in the stack, we need to remove the ones that are placed on top of them. Another Figure shows another stack. This one contains a number of primitive data objects.

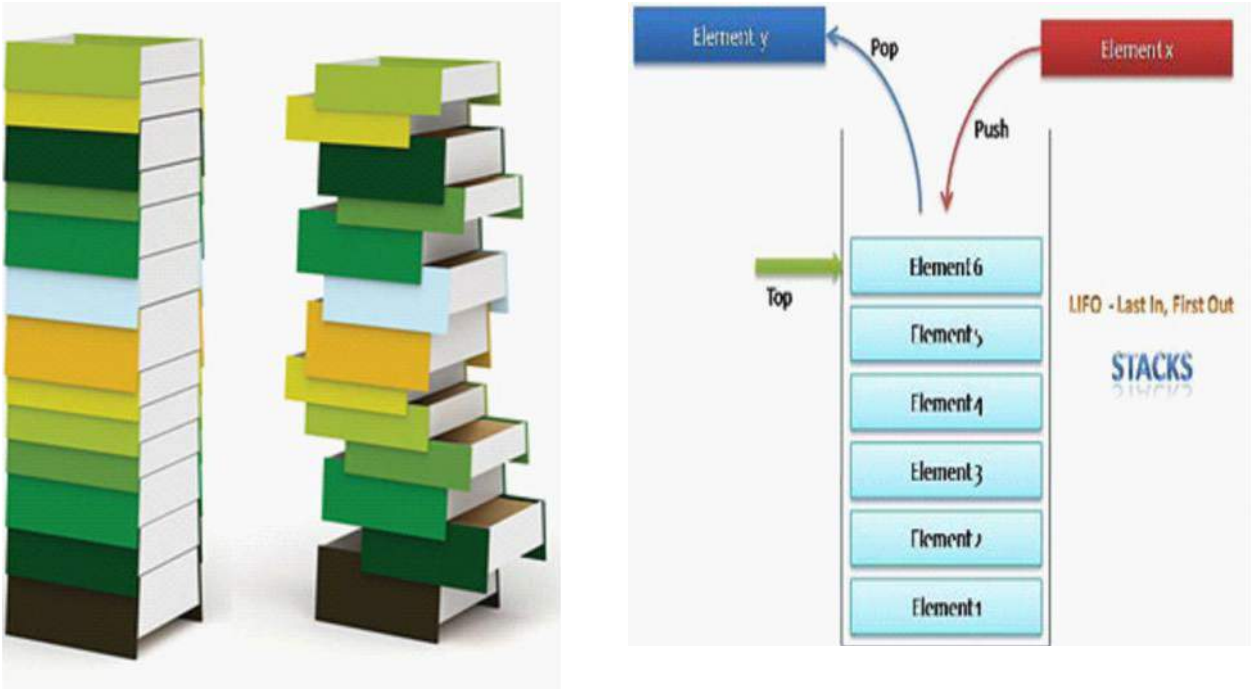


Figure A Stack of Books

One of the most useful ideas related to stacks comes from the simple observation of items as they are added and then removed. Assume you start out with a clean desktop. Now, place books one at a time on top of each other. You are constructing a stack. Consider what happens when you begin removing books. The order that they are removed is exactly the reverse of the order that they were placed. Stacks are fundamentally important, as they can be used to reverse the order of items. The order of insertion is the reverse of the order of removal.

Considering this reversal property, you can perhaps think of examples of stacks that occur as you use your computer. For example, every web browser has a Back button. As you navigate from web page to web page, those pages are placed on a stack (actually it is the URLs that are going on the stack). The current page that you are viewing is on the top and the first page you looked at is at the base. If you click on the Back button, you begin to move in reverse order through the pages.

4.6.2 Representation of stacks in memory

The representation of a stack in the memory can be done in two ways.

- Static representation using arrays
- Dynamic representation using linked lists

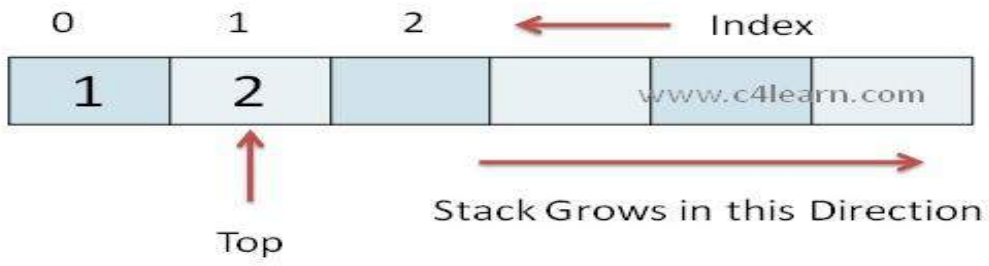
Array Representation of a Stack

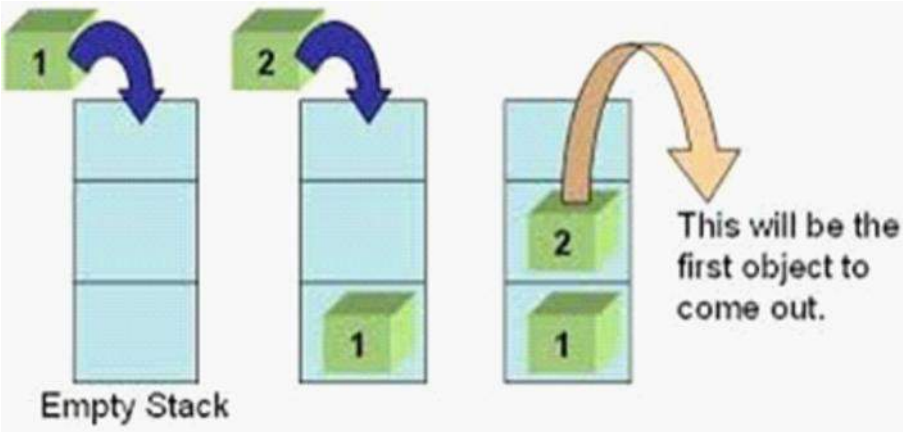
Stack can be represented using a one-dimensional array. A block of memory is allocated which is required to accommodate the items to the full capacity of the stack. The items into the stack are stored in a sequential order from the first location of the memory block.

A pointer TOP contains the location of the top element of the stack. A variable MAXSTK contains the maximum number of elements that can be stored in the stack.

The condition TOP = MAXSTK indicates that the stack is full and TOP = NULL indicates that the stack empty.

Representing a stack using arrays is easy and convenient. However, it is useful for fixed sized stacks. Sometimes in a program, the size of a stack may be required to increase during execution, i.e. dynamic creation of a stack. Dynamic creation of a stack is not possible using arrays. This requires linked lists.

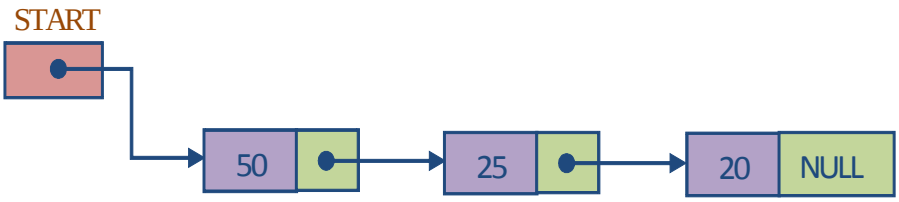




Linked list representation of a Stack

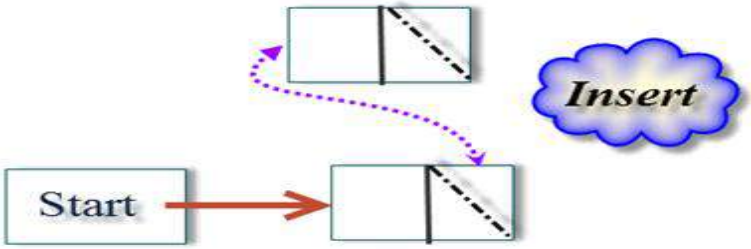
The size of the array needs to be fixed to store the items into the stack. If the stack is full we cannot insert an additional item into the array. It gives an overflow exception. But in linked list we can increase the size at runtime by creating a new node. So it is better to implement stack data structure using

Linked list data structure. The linked list structure will be studied in detail later.



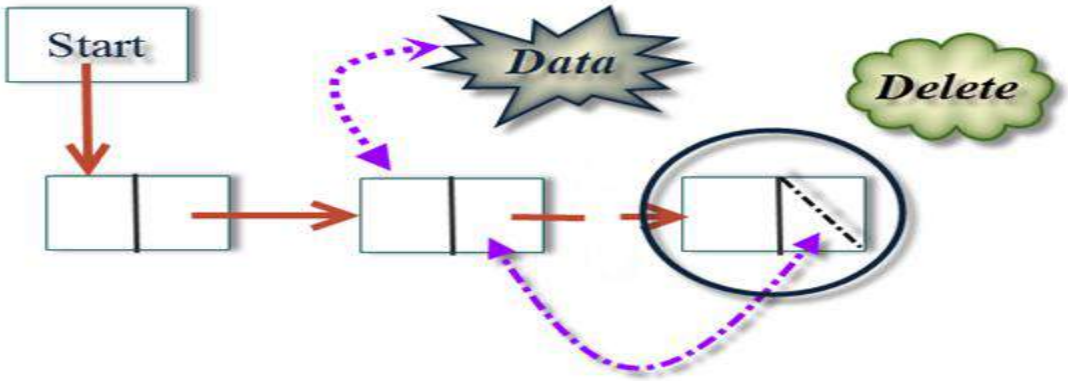
Insertion

Insertion operation refers to Inserting an element into stack. We create a new node and insert an element into the stack. To follow the stack principle "**Last-in-first-out**", a node need to be created from the end of the Linked list and element need to be inserted into the node from the end of the linked list.



Deletion

Deletion operation is to delete an element or node from the Linked list. Deletion can be done by deleting the top-most item from the stack as the last item inserted is the first item that needs to be deleted as per stack principle. So the recently inserted item i.e, top item must be deletes from the linked list to perform as stack deletion.



4.6.3 Operation Stack

The stack abstract data type is defined by the following structure and operations. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the "top." Stacks are ordered LIFO. The **stack operations** are given below.

- **stack()** creates a new stack that is empty. It needs no parameters and returns an empty stack.
- **push(item)** adds a new item to the top of the stack. It needs the item and returns nothing.
- **pop()** removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- **peek()** returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- **isEmpty()** tests whether the stack is empty. It needs no parameters and returns a Boolean value.
- **size()** returns the number of items on the stack. It needs no parameters and returns an integer.

Algorithm for PUSH Operation: PUSH(STACK , TOP, SIZE, ITEM)

STACK is the array that contains N elements and TOP is the pointer to the top element of the array. ITEM the element to be inserted. This procedure inserts ITEM into the STACK.

```
Step 1:   If TOP = N -1then                [check overflow]
              PRINT "Stack is full"
              Exit
          End of If
Step 2:   TOP = TOP + 1                    [Increment the TOP]
Step 3:   STACK[TOP] = ITEM                [Insert the ITEM]
Step 4:   Return
```

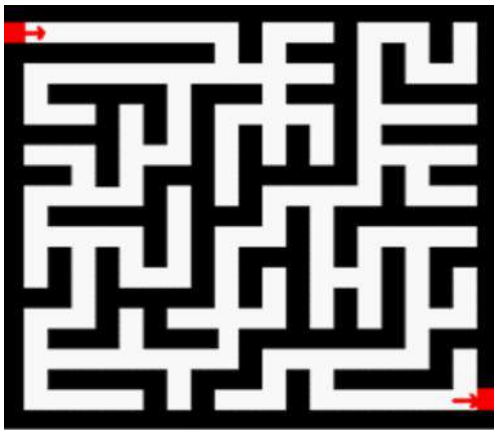
Algorithm for POP Operation: POP(STACK , TOP, ITEM)

STACK is the array that store N items. TOP is the pointer to the top element of the array. This procedure deleted top element from STACK.

```
Step 1:   If TOP = NULL then                    [check underflow]
              PRINT "Stack is empty"
          Exit
          End of If
Step 2:   ITEM = STACK[TOP]                     [Copy the top element]
Step 3:   TOP = TOP – 1                         [Decrement the top]
Step 4:   Return
```

4.6.4 Application of Stacks

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.
- Backtrackin



Backtracking: This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?

Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

- Language processing:
  - Space for parameters and local variables is created internally using a stack.
  - Compiler's syntax check for matching braces is implemented by using stack.
  - support for recursion
- Conversion of decimal number into binary
- To solve tower of Hanoi
- Expression evaluation and syntax parsing
- Conversion of infix expression into prefix and postfix.
- Rearranging railroad cars

- Quick sort
- Stock span problem
- Runtime memory management

Arithmetic expression: An expression is a combination of operands and operators that after evaluation results in a single value. Operands consist of constants and variables. Operators consists of {, +, - , * , / …….. etc.,) [Expressions can be

- Infix expression
- Post fix expression
- Prefix expression

Infix expression: If an operator is in between two operands it is called infix expression.

Example: a + b, where a and b are the operands and + is an operator.

Postfix expression: If an operator follows the two operands it is called post fix expression.

Example: ab +

Prefix expression: If an operator precedes two operands, it is called prefix expression.

Example: +ab

Algorithm for Infix to Postfix

1. Examine the next element in the input.
2. If it is operand, output it.
3. If it is opening parenthesis, push it on stack.
4. If it is an operator, then
   - If stack is empty, push operator on stack.
   - If the top of stack is opening parenthesis, push operator on stack
   - If it has higher priority than the top of stack, push operator on stack.
   - Else pop the operator from the stack and output it, repeat step 4.
5. If it is a closing parenthesis, pop operators from stack and output them until an opening parenthesis is encountered. Pop and discard the opening parenthesis.
6. If there is more input go to step 1.

7. If there is no more input, pop the remaining operators to output.

Example: Suppose we want to convert 2*3/(2-1)+5*3 into Postfix form.

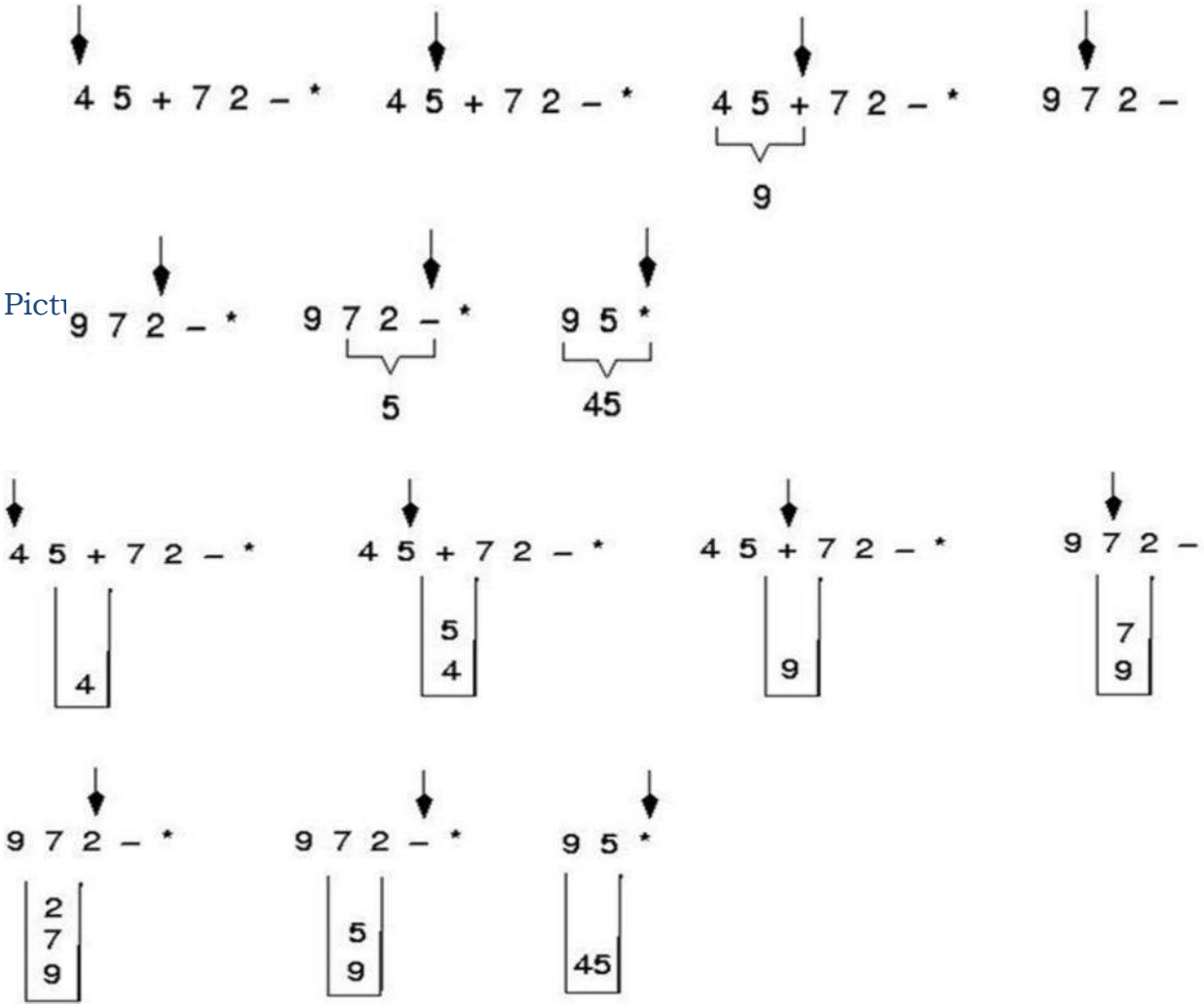
Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

So, the Postfix Expression is 23*21-/53*+

Evaluating a postfix expression using stack

- The postfix expression to be evaluated is scanned from left to right.
- Each operator in a postfix string refers to the previous two operands in the string.
- Each time we read an operand we push it into a stack. When we reach an operator, its operands will then be top two elements on the stack.
- We can then pop these two elements, perform the indicated operation on them, and push the result on the stack.
- So that it will be available for use as an operand of the next operator.
- Initialize an empty stack.
- While character remain in the input stream
  - Read next character.
  - If character is an operand, push it into the stack.

- Else, if character is an operator, pop top two characters off the stack, apply the operator, and push the answer back into the stack.
- Pop the answer off the stack.



Algorithm for evaluating a postfix expression

```
WHILE more input items exist
{
        If symb is an operand then
                push (operand_stk, symb)
        else                                    //symbol is an operator
        {
                Opnd1 = pop(operand_stk);
                Opnd2 = pop(operand_stk);
                Value  = opnd2 symb opnd1
                Push(operand_stk, value);
        }
}       Result = pop (operand_stk);
```

Consider an expression S having operators, operands, left parenthesis and right parenthesis. Operators includes +, - , /, *. Evaluations are performed from left to right otherwise indicated by parenthesis. Stack is used to hold operators and left parenthesis. The postfix expression can be obtained from left to right using operands from } the operators which are removed from STACK. Left parentheses are pushed onto stack and right parenthesis at the end of S. Algorithm is completed when STACK is empty.

Example 1: Convert A + (B * C – (D/E^ F) into postfix notation.

Symbol scanned		stack	expression
1.	A	(	A
2.	+	(+	A
3.	(	(+(	A
4.	B	(+(	A B
5.	*	(+(*	A B
6.	C	(+(*	A B C
7.	-	(+(-	A B C *
8.	(	(+(-(	A B C *
9.	D	(+(-(	A B C * D
10.	/	(+(-(/	A B C * D
11.	E	(+(-(/	A B C * D E
12.	^	(+(-(/^	A B C * D E
13.	F	(+(-(/^	A B C * D E F
14.	)	(+(-	A B C * D E F ^ /

Examples 2: Convert the following infix expressions postfix notation.

+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
(	(*(	AB+
C	(*(	AB+C
-	(*(-	AB+C
D	(*(-	AB+CD
)	(*	AB+CD-
/	(/	AB+CD-*
E	(/	AB+CD-*E
)		AB+CD-*E/

Answer: Postfix expression of **(A+B)*(C-D)/E** is **AB+CD-*E/**

Infix Expression	Prefix Expression
A +B-C	-+ABC
(A+B) *(C+D)	*+AB+CD
A/B*C-D+E/F/(G+H)	+-*/ABCD//EF+GH
((A + B) * C - (D-E))*(F+G)	*-*+ABC-DE +FG
A-B/(C *D/E)	-A/B/*CDE

Queues

4.7.1 Introduction

We now turn our attention to another linear data structure. This one is called **queue**. Like stacks, queues are relatively simple and yet can be used to solve a wide range of important problems.

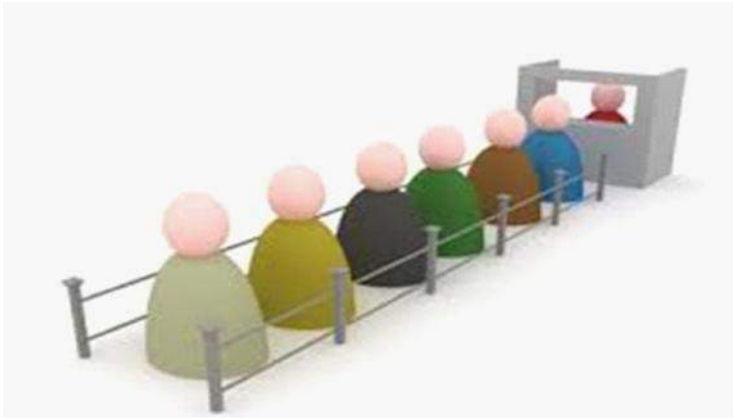
A **queue** is an ordered collection of items where the addition of new items and the removal of existing items always take place at different ends.

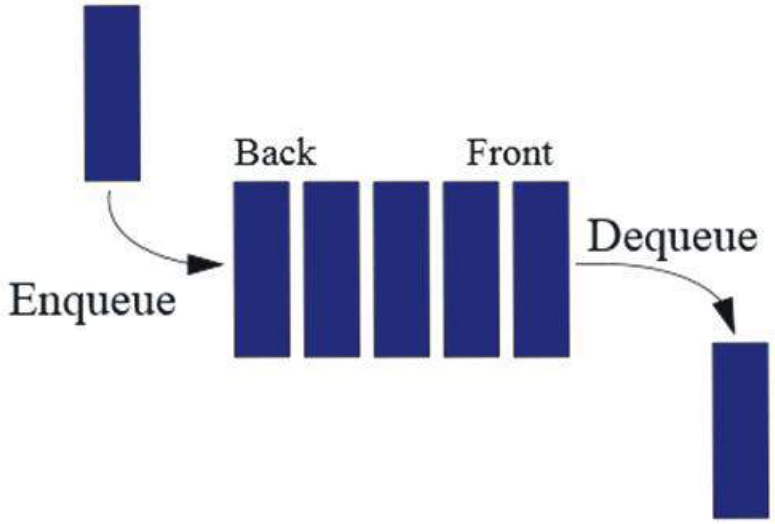
A queue is an ordered collection of items where an item is inserted at one end called the "rear," and an existing item is removed at the other end, called the "front." Queues maintain a FIFO ordering property.

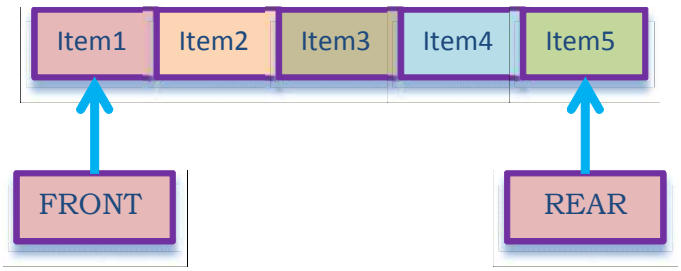
Insertion and deletion is performed according to the first-in first-out (FIFO) principle. An excellent example of a queue is a line of students in the food court of a canteen. New additions to the line are made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed **enqueue** and **dequeue**. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Queue is also called as FIFO list, i.e. **First-In-First-Out**. Here insertions are limited to one end of the list called the **rear**, whereas deletions are limited to other end called as **front**.





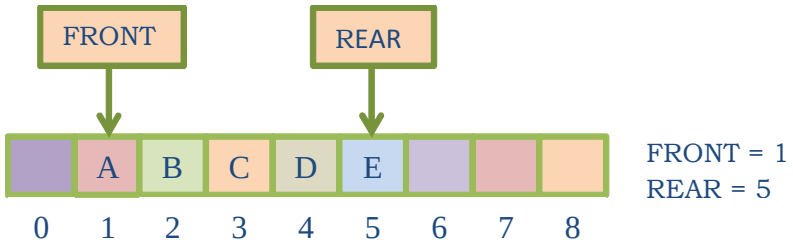


4.7.2 Types of queues

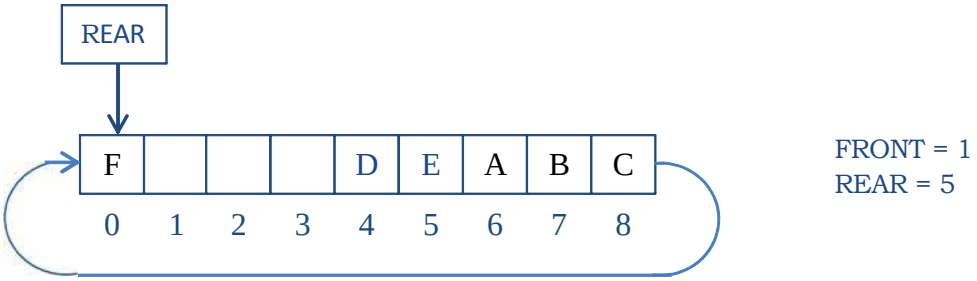
Queue can be of four types:

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Dequeue (Double Ended queue)

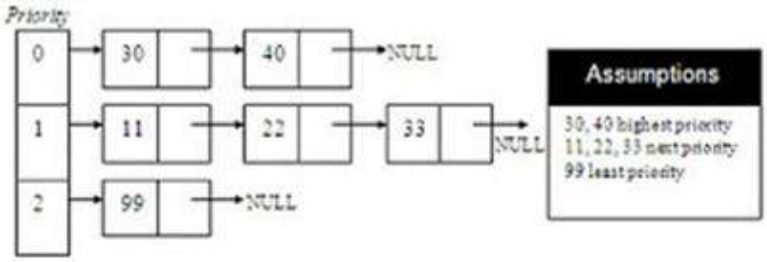
1. Simple Queue: In Simple queue insertion occurs at the rear end of the list, and deletion occurs at the front end of the list.



2. Circular Queue: A circular queue is a queue in which all nodes are treated as circular such that the last node follows the first node.

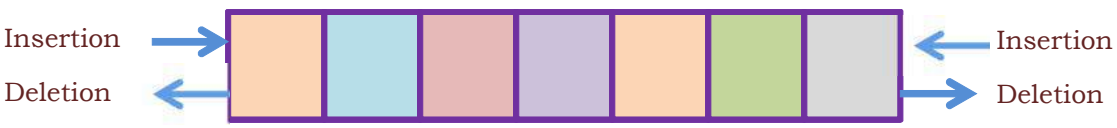


3. Priority Queue: A priority queue is a queue that contains items that have some preset priority. An element can be inserted or removed from any position depending on some priority.



4. Dequeue (Double Ended queue):

It is a queue in which insertion and deletion takes place at both the ends.



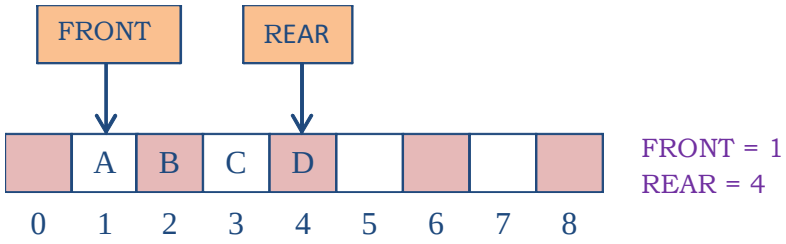
4.7.3 Operations on queue

The queue abstract data type is defined by the following structure and operations. The queue operations are given below.

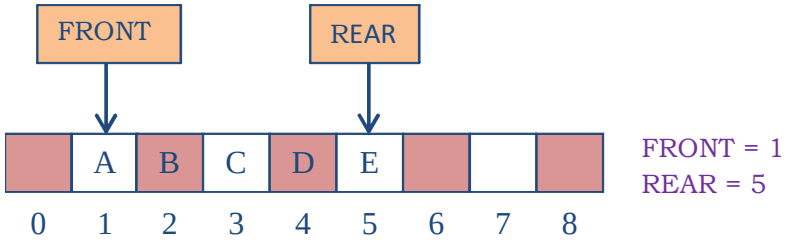
- **Queue()** creates a new queue that is empty. It needs no parameters and returns an empty queue.
- **enqueue(item)** adds a new item to the rear of the queue. It needs the item and returns nothing. This operation is generally called as **push**.
- **dequeue()** removes the front item from the queue. It needs no parameters and returns the item. The queue is modified. This operation is generally called as **pop**.
- **isEmpty()** tests to see whether the queue is empty. It needs no parameters and returns a Boolean value.
- **size()** returns the number of items in the queue. It needs no parameters and returns an integer.

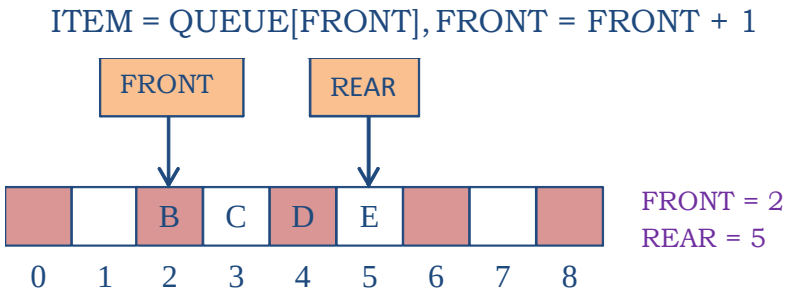
4.7.4 Memory representation of a queue using arrays

Queue is represented in memory linear array. Let QUEUE be a linear queue. Two pointer variables called FRONT and REAR are maintained. The pointer variable FRONT contains the location of the element to be removed and the pointer variable REAR contains location of the last element inserted. The condition FRONT = NULL indicates that the queue is empty and the condition REAR = N indicates that the queue is full.



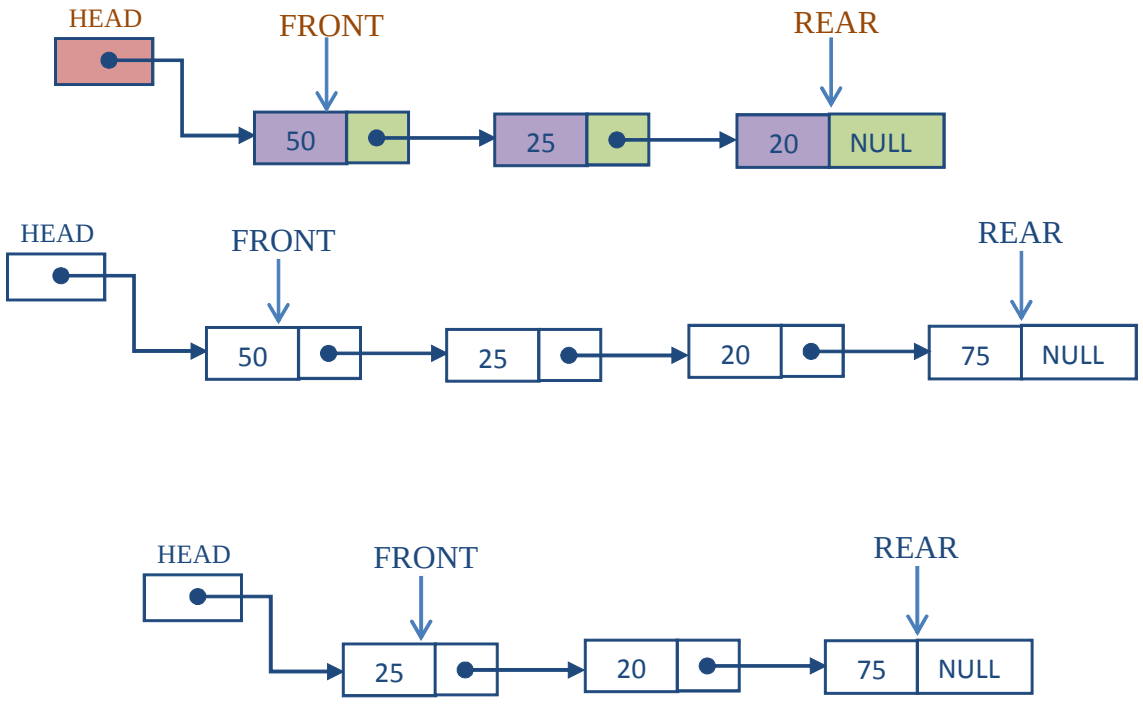
REAR = REAR +1, QUEUE[REAR] =' E'





Memory representation of a queue using linked list

Queues are also represented using linked lists. In queues an item should be inserted from rear end and an item should be removed from the front end. The pointer front contains location of the first node of the linked list and another pointer rear contains location of the last node.



Algorithm for insertion

Algorithm: Let QUEUE be the linear array consisting of N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. ITEM is the element to be inserted.

```
Step 1:   If REAR = N-1 Then      [Check for overflow]
              PRINT "Overflow"
              Exit
Step 2:   If FRONT = NULL Then    [Check whether QUEUE is empty]
              FRONT = 0
              REAR = 0
          Else
              REAR = REAR + 1     [Increment REAR Pointer]
Step 3:   QUEUE[REAR] = ITEM      [Copy ITEM to REAR position]
Step 4:   Return
```

Algorithm for deletion

Algorithm: Let QUEUE is the linear array consisting of N elements. FRONT is the pointer that contains the location of the element to be deleted and REAR contains the location of the inserted element. This algorithm deletes the element at FRONT position.

```
Step 1:   If FRONT = NULL Then    [Check whether QUEUE is empty]
              PRINT "Underflow"
              Exit
Step 2 :  ITEM = QUEUE[FRONT]
Step 3:   If FRONT = REAR Then    [If QUEUE has only one element]
              FRONT = NULL
              REAR = NULL
          Else
              FRONT = FRONT + 1   [Increment FRONT pointer]
Step 3:   Return
```

4.7.5 Applications of queues

- Simulation
- Various features of operating system.
  [Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.]
- Multi-programming platform systems
- Different type of scheduling algorithm
- Round robin technique or Algorithm
- Printer server routines
- Various applications software is also based on queue data structure
- Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

LINKED LISTS

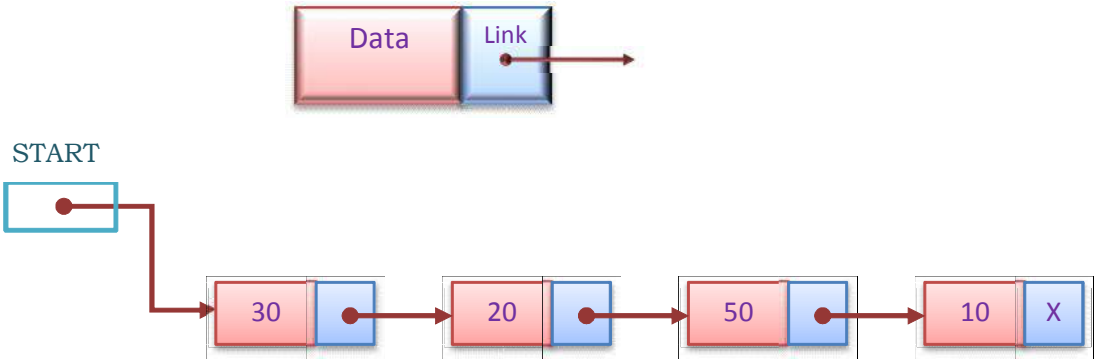
4.8.1 Introduction

One disadvantage of using arrays is that arrays are **static** structures and therefore cannot be easily extended or reduced to fit the data set. During insertion, the array elements are shifted into higher order memory locations and during deletion, elements ate shifted into lower order memory locations. In this chapter we study another data structure called Linked Lists that addresses these limitations of arrays. The linked list uses **dynamic memory allocations.**

Linked list

A linked list is a linear collection of data elements called nodes and the linear order is given by means of pointers.

Each node contains two parts fields: the data and a reference to the next node. The first part contains the information and the second part contains the address of the next node in the list. This is also called the link field.



The above picture is linked list with 4 with nodes, each node is contains two parts. The left part of the node represents **the information part** of the node and the right part represents the **next pointer field** that contains the address of the next node. A pointer START gives the location of the first node. This pointer is also represented as HEAD. Note that the link field of the last node contains NULL.

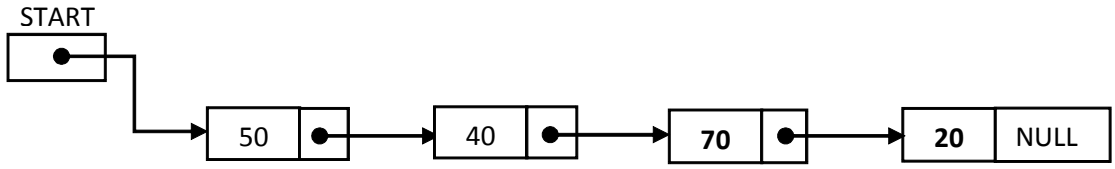
4.8.2 Types of linked lists

There are three types of linked lists.

1. Singly linked list (SLL)
2. Doubly linked list (DLL)
3. Circular linked list (CLL)

Single linked list

A singly linked list contains two fields in each node – the data field and link field. The data field contains the data of that node while the link field contains address of the next node. Since there is only one link field in each node, the linked list is called as singly linked list.



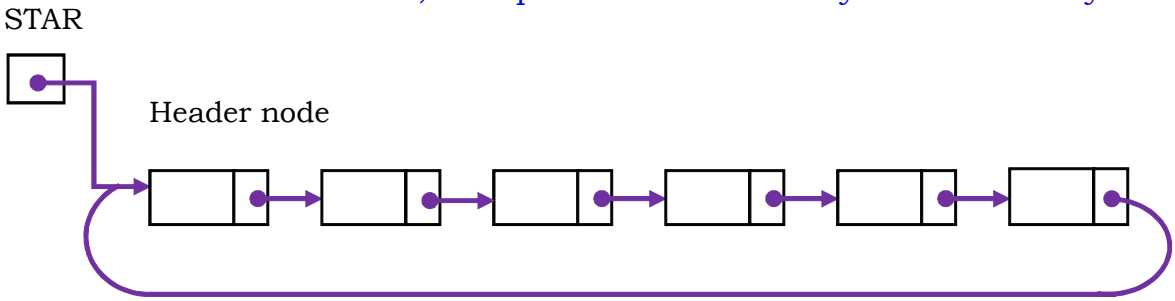
In any linked list the nodes need not necessarily represent a set of consecutive memory locations (or contiguous memory locations).

Circular linked lists:

In a singly linked list, a pointer is available to access all the succeeding nodes, but not preceding nodes. In the singly linked lists, the link field of the last node contains NULL.

In circular lists, if the link field of the last node contains the address of the first node first node, such a linked list is called as circular linked list.

In a circular linked list, it is possible to reach any node from any other nc



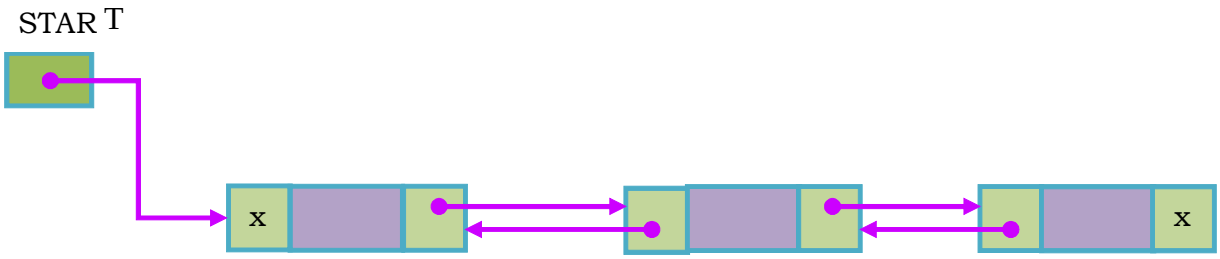
Doubly linked lists:

It is a linked list in which each node is points both to the next node and also to the previous node.

In doubly linked list each node contains three parts – FORW, BACK and INFO.

BACK: It is a pointer field containing the address of the previous node.
FORW: It is a pointer field that contains the address of the next node.
INFO: It contains the actual data.

In the first node, if BACK contains NULL, it indicates that it is the first node in the list. The node in which FORW contains NULL indicates that the node is the last node in the linked list.



In our discussion, we will only study the singly linked list.

4.8.3 Operations on linked lists:

The operations that are performed on linked lists are

1. Creating a linked list
2. Traversing a linked list
3. Inserting an item into a linked list
4. Deleting an item from the linked list
5. Searching an item in the linked list
6. Merging two or more linked lists

Creating a linked list

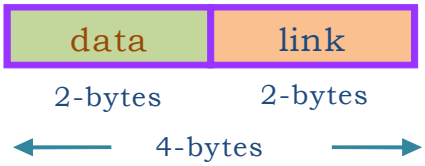
Linked list is linear data structure which contains a group of nodes and the nodes are sequentially arranged. Nodes are composed of data and address of the next node or reference of the next node. These nodes are sequentially or linearly arrayed that is why the Linked list is a linear data structure. In linked list we start with a node and create nodes and link to the starting node in order and sequentially. The pointer START contains the location of the first node. Also the next pointer field of the last node must be assigned to NULL. In this topic we will be discussing about Single Linked List. Elements can be inserted anywhere in the linked list and any node can be deleted.

The nodes of a linked list can be created by the following structure declaration.

```
struct Node
{
      int data;
Node* link;
}
Node   *node1, *node2;
```

OR

```
struct Node
{
      int    data;
      Node* link;
} *node1, node2;
```



Here data is the information field and link is the link field. The link field contains a pointer variable that refers the same node structure. Such a reference is called as self-addressing pointer.

The above declaration creates two variable structures. Each structure will be taken as node. Thus it is linked list that contains two nodes. Another node cannot be inserted if it not already declared. Also, there may be cases where the memory needs of a program can only be determined during runtime. On these cases, programs need to dynamically allocate memory, for which the C++ language use the operators **new** and **delete**.

- Operator new allocates space.
- Operator new[] allocates memory space for array.
- Operator delete deallocate storage space.
- Operator delete[] deallocate memory space for array.

Operator new and new[]

Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated.

Syntax: pointer = new type
pointer = new type [number_of_elements]

The first expression is used to allocate memory to contain one single element of the required data type. The second one is used to allocate a block (an array) of elements of data type type, where number_of_elements is an integer value representing the amount of these. For example:

For example, int *tmp;
tmp = new int [5]

Operator delete and delete[]

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. For this purpose the operator delete is used.

Syntax: delete pointer;
delete [] pointer;

The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new and a size in brackets [].

The value passed as argument to delete shall be either a pointer to a memory block previously allocated with new, or a *null pointer* (in the case of a *null pointer*, delete produces no effect).

Dynamically the memory space is allocated for the linked list using new operator as follows:

```
Node *newNode;

p = new Node;
data(p) = num;
link(p) = NULL;
```

Program: Creating a linked list and appending nodes into the linked list.

```
#include <iostream.h>
class LinkList
{
     private:
          struct Node
          {
               int data;
               Node* link;
          }*START;
     public:
          LinkList();
          void Print(); //Prints the contents of linked list
          void Append(int num);   //Adds a new node at the end
of the linked list
          void Count(); //Counts number of nodes in the linked
list
};
LinkList::LinkList()              //Constructor
{
     START = NULL;
}

// Prints the contents of linked list
void LinkList::Print()
{
     if (START == NULL)
     {
          cout<< "Linked list is empty"<<endl;
```

```
          return;
     }
     //Traverse
     cout<<"Linked list contains";
     Node* p = START;
     while(p != NULL)
     {
          cout<<"   "<<p->data;;
          p = p->link ;
     }
}
// Adds a new node at the end of the linked list
void LinkList::Append(int num)
{
     Node *newNode;
     newNode = new Node;
     newNode->data = num;
     newNode->link = NULL;

     if(START == NULL)
     {
          //create first node
          START = newNode;
          cout<<endl<<num<<" is inserted as the first node"<<endl;
     }
     else
          {
               //Traverse
               Node *p = START;
               while(p->link != NULL)
               p = p->link;
               //add newnode to the end of the linked list
               p->link = newNode;
               cout<<endl<<num<<" is inserted"<<endl;
          }
}
// Counts number of nodes present in the linked list
void LinkList::Count()
{
     Node *p;
     int c = 0;
          //Traverse the entire Linked List
     for (p = START; p != NULL; p = p->link)
          c++;
```

```
    cout<<endl<<"No. of elements in the linked list= "<<c<<endl;
}
void main()
{
    LinkList* obj = new LinkList();

    obj->Print();
    obj->Append(100);
    obj->Print();
    obj->Count();

    obj->Append(200);
    obj->Print();
    obj->Count();

    obj->Append(300);
    obj->Print();
    obj->Count();
}
```

```
    Linked list is empty

    100 is inserted as the first node
    Linked list contains  100
    No. of elements in the linked list= 1

    200 is inserted
    Linked list contains  100  200
    No. of elements in the linked list= 2

    300 is inserted
    Linked list contains  100  200  300
    No. of elements in the linked list= 3
```

Traversing a linked list

Traversing is the process of accessing each node of the linked list exactly once to perform some operation.

To traverse a linked list, steps to be followed are given here.

1. To begin with, move to the first node.
2. Fetch the data from the node and perform the required operation depending on the data type.
3. Advance the pointer to the next node.
4. Step 2 and step 3 is repeated until all the nodes are visited.

Algorithm: This algorithm traverses the linked list. Here START contains the address of the first node. Another pointer p is temporarily used to visit all the nodes from the beginning to the end of the linked list.

```
Step 1:    p = START                        [Initialize p to first node]
Step 2:    while p != NULL
Step 3:          PROCESS   data(p)          [Fetch the data]
Step 4:          p = link(p)                [Advance p to next node]
Step 5:    End of while
Step 6:    Return
```

Memory allocation to a linked list

Computer memory is a limited resource. Therefore some mechanism is required where the memory space of the deleted node becomes available for future use. Also, to insert a node to the linked list, the deleted node should be inserted into the linked list. The operating system of the computer maintains a special list called AVAIL list that contains only the unused deleted nodes.

AVAIL list is linked list that contains only the unused nodes. Whenever a node is deleted from the linked list, its memory is added to the AVAIL list and whenever a node is to be inserted into the linked list, a node is deleted from the AVAIL list and is inserted into the linked list. AVAIL list is also called as the free-storage list or free-pool.

Inserting a node into the linked list

Sometimes we need insert a node into the linked list. A node can be inserted into the linked list only if there are free nodes available in the AVAIL list. Otherwise, a node cannot be inserted. Accordingly there sre three types of insertions.

1. Inserting a node at the beginning of the linked list
2. Inserting a node at the given position.
3. Inserting a node at the end of the linked list

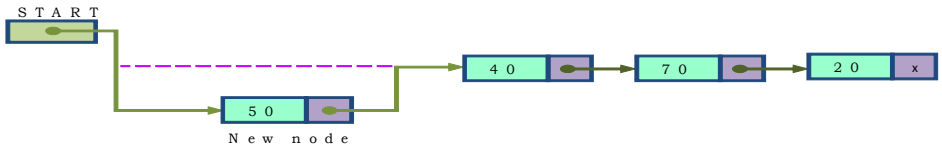
Inserting a node at beginning of the linked list

START is pointer that contains the memory address of the first node. To insert an ITEM into the linked list, the following procedure is used.

1. Create a new node.
2. Fill data into the data field of the new node.
3. Mark its next pointer field as NULL.
4. Attach this newly created node to START.
5. Make the new node as the STARTing node.

Algorithm (inst-beg): This algorithm inserts a node at the beginning of the linked list.

1. p ← new Node;
2. data(p) ← num;
3. link(p) ← START
4. START ← p
5. Return



Program: Inserting node at the beginning of the linked list.

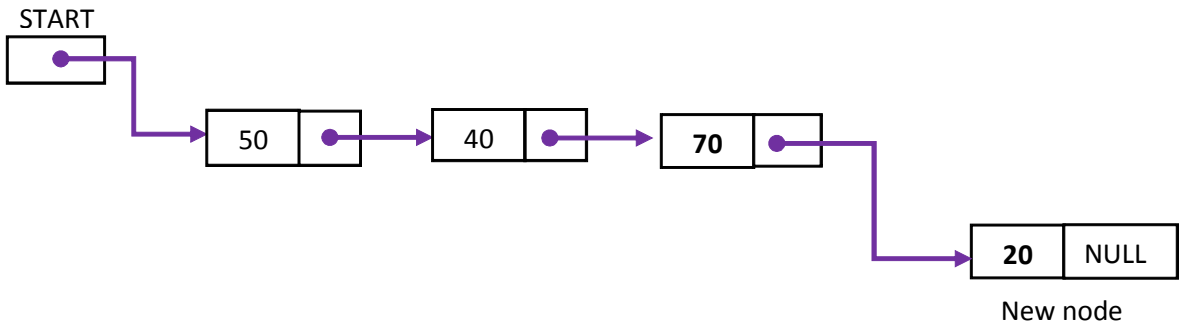
```
#include <iostream.h>
#include<ctype.h>
class LinkList
{
        private:
                struct Node
                {
                        int     data;
                        Node* link;
                }*START;
        public:
                LinkList();
                void    Print();
                void    Count();
                void    insert(int num);
};
LinkList::LinkList()
{
        START = NULL;
}
void LinkList::Print()
{
        if (START == NULL)
        {
                cout<< "Linked list is empty"<<endl;
                return;
        }
        cout<<endl<<"Linked list contains";
```

```
        Node* p = START;
        while(p != NULL)
        {
                cout<<“  “<<p->data;;
                p = p->link ;
        }
}
void LinkList::insert(int num)
{
        Node *newNode;
        newNode = new Node;
        newNode->data = num;
        newNode->link = START;
        START = newNode;
        cout<<num<<“ is inserted at the beginning”<<endl;
}
void LinkList::Count()
{
        Node *p;
        int c = 0;
                //Traverse the entire Linked List
        for (p = START; p != NULL; p = p->link)
                c++;
        cout<<endl<<“No. of elements in the linked list= “<<c<<endl;
}
void main()
{
        LinkList* obj = new LinkList;
        int item;
        char ch;

        cout<<“Do you want to insert a node at the beginning (Y/N): “;
        cin>>ch;
        while(toupper(ch) == ‘Y’)
        {
                cout<<“Enter the item to be inserted at the beginning: “;
                cin>>item;
                obj->insert(item);
                cout<<“Do you want to insert a node at the beginning (Y/N): “;
                cin>>ch;
        }
        obj->Print();
        obj->Count();
        cout<<“T H A N K   Y O U”;
}
```

```
Do you want to insert a node at the beginning (Y/N): y
Enter the item to be inserted at the beginning: 111
111 is inserted at the beginning
Do you want to insert a node at the beginning (Y/N): y
Enter the item to be inserted at the beginning: 222
222 is inserted at the beginning
Do you want to insert a node at the beginning (Y/N): y
Enter the item to be inserted at the beginning: 333
333 is inserted at the beginning
Do you want to insert a node at the beginning (Y/N): n

Linked list contains  333  222  111
No. of elements in the linked list= 3
T H A N K   Y O U
```

Inserting a node at the end of the linked list

We can insert a node at the end of the linked list. To insert at the end, we have to traverse the liked list to know the address of the last node.

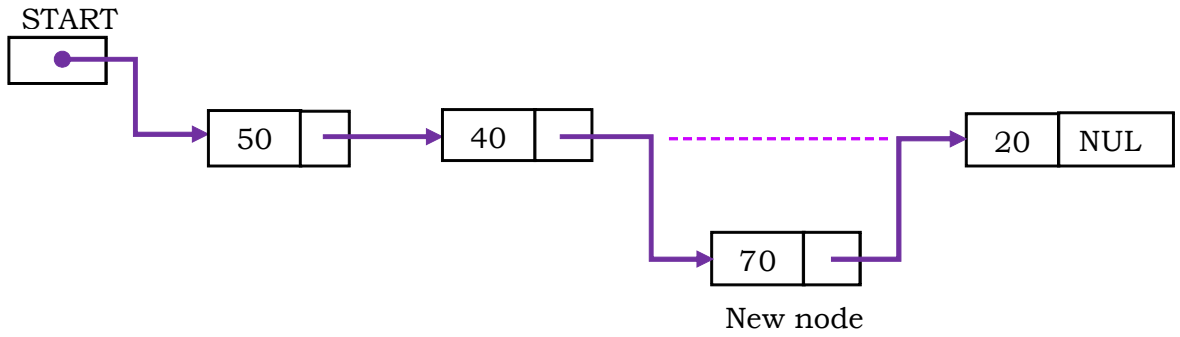


Algorithm (inst-end): Inserting a node at the end of the linked list.

1. START
2. [Identify the last node in the list]
   a. p ← START
   b. while p != NULL
      p ← next(p)
      while end
3. N ← new Node [Create new node copy the address to the pointer N]
4. data(N) ← item
5. link(N) ← NULL
6. link(p) ← N
7. RETURN

Inserting a node at a given position

We can insert a node at a given position. The following procedure inserts a node at the given position.



Algorithm (INST-POS): This algorithm inserts item into the linked list at the given position pos.

```
1. START
2. [[Initialize] count ← 0
                 p1 ← START
3. while (p1 != NULL)
          count ← count +1
          p1 ← link(p1)
   while end
4. a.  if (pos = 1)
          call function INST-BEG()
       else
   b.      if (pos = count +1)
                call function INST-END()
          else
   c.      if( pos <= count)
                        p1 ← START
                        for (i = 1; i <= pos; i++ )
                              p1 ← next(p1)
                        for end
   d.                   [create] p2 ← new node
                        data(p2) ← item
                        link(p2) ← link(p1)
                        link(p1) ←  p2
                else
   e.                   PRINT " Invalid position "
   5. RETURN
```

Garbage collection:

If a node is deleted from the linked list or if a linked list is deleted, we require the space to be available for future use. The memory space of the deleted nodes is immediately reinserted into the free-storage list. The operating system of the computer periodically collects all the deleted space into the free-storage

list. This technique of collecting deleted space into free-storage list is called as garbage collection.

Deleting an item from the linked list:

The deletion operation is classified into three types:

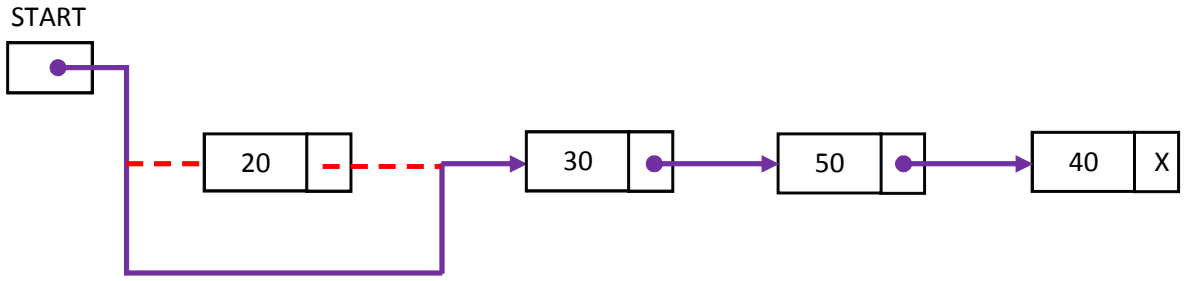
1. Deletion of the first node
2. Deletion of the last node
3. Deletion of the node at the given position

Underflow

If we try to delete a node from the linked list which is empty, the condition is called as underflow.

Deletion of the first node:

We can easily delete the first node of the linked list by changing the START to point to the second node of the linked list. If there is only one node in the liked list, the START can be stored with NULL.

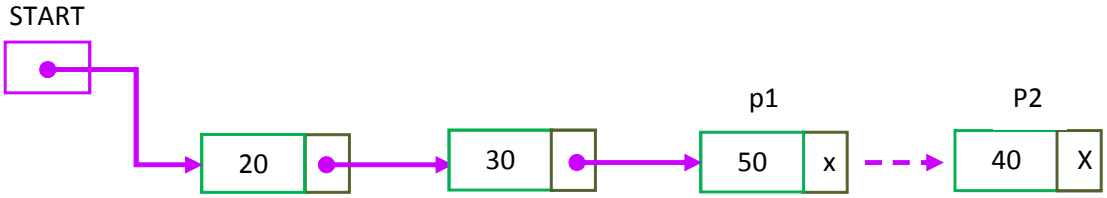


Algorithm (DELE-BEG): This algorithm first copy data in the first node to a variable and deletes the first node of the linked list.

Step1. START
Step2. p ← START
Step3. PRINT data(p)
Step4. START← link(p)
Step5. free(p)
Step6. RETURN

Deleting a node at the end (DELE-END)

To delete the last node, we should find location of the second last node. By assigning NULL to link field of the second last node, last node of the linked list can be deleted.



Algorithm (DELE-END): This used two pointers p1 and p2. Pointer p1 is used to traverse the linked list and pointer p2 keeps the location of the previous node of p1.

```
1.  START
2.  p2 ← START
3.  while ( link(p2) != NULL)
        p1 ← p2
        p2 ← link(p2)
    while end
4.  PRINT data(p2)
5.  link(p1) ← NULL
    free(p1)
6.  STOP
```

Deleting a node at the given position

To delete a node at the given position, the following procedure is used.

1. Find location of the node to be deleted and the previous node.
2. Delete the node to be deleted by changing the location of the previous node of the node to be deleted.

Algorithm (DELE-POS): Here p1 and p2 are the pointers. Pointer p2 is used to traverse the linked list. If pointer p2 has the location of the node to be deleted, p1 keeps the location of previous node of p2.

```
1. START
2. Count ← 0
   p1 ← START
3. while(p1 != NULL)
       count = count +1
       p1 ←link(p1)
   while end
4. if( pos = 1)
       CALL DELE-BEG()
   else
       if(pos = count)
           CALL DELE-END()
       else
           if(pos < count)
               p2 ← START
               for i = 1 to pos
                   p1 ← p2
                   p2 ← link(p2)
               for end
```

```
                    PRINT data(p2)
                    link(p1) ←link(p2)
                    free(p2)
               else      PRINT "Invalid position"
5. RETURN
```

Non-linear data structure

4.9.1 Introduction

A non-linear data structure is a data structure in which a data item is connected to several other data items, so that a given data item has the possibility to reach one-or-more data items.

The data items in a non-linear data structure represent hierarchical relationship. Each data item is called a node. Examples of non-linear data-structures are Graphs and Trees.

Pros

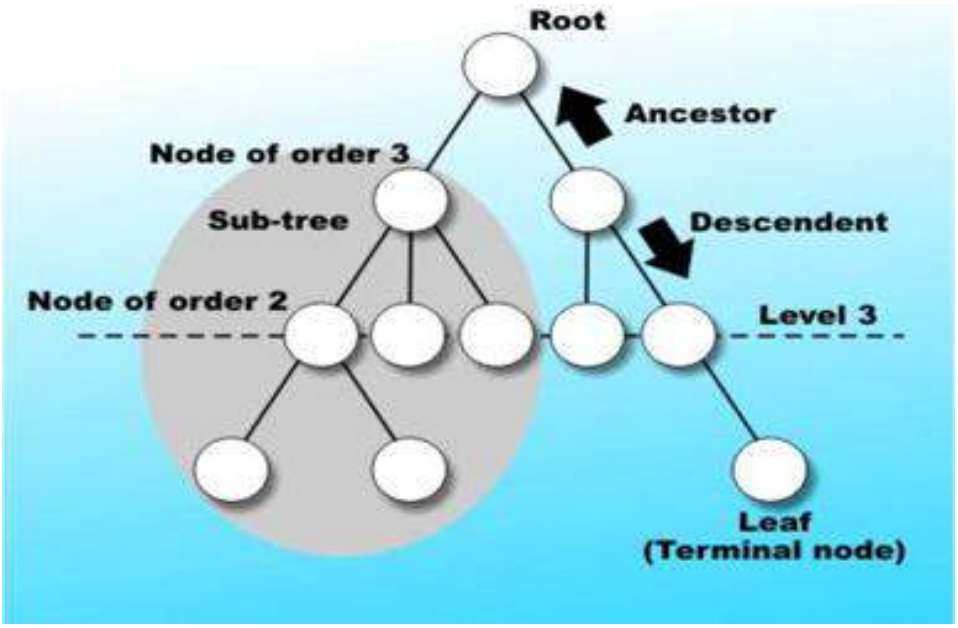
- Uses memory efficiently as contiguous memory is not required for allocating data items.
- The length of the data items is not necessary to be known prior to allocation.

Cons

- Overhead of the link to the next data item.

4.9.2 TREES

A tree is a data structure consisting of nodes organized as a hierarchy - see figure.



Terminology

A node is a structure which may contain a value, a condition, or represent a separate data structure (which could be a tree of its own). Each node in a tree has zero or more **child nodes**, which are below it in the tree (by convention, trees are drawn growing downwards). A node that has a child is called the **parent node** (or *ancestor node*, or superior). A node has at most one parent.

Nodes that do not have any children are called **leaf nodes**. They are also referred to as terminal nodes.

The **height** of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The **depth** of a node is the length of the path to its root (i.e., its root path).

The topmost node in a tree is called the **root node**. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin. All other nodes can be reached from it by following **edges** or **links**.

An **internal node** or **inner node** is any node of a tree that has child nodes and is thus not a leaf node.

A **subtree** of a tree *T* is a tree consisting of a node in *T* and all of its descendants in *T*.

Binary trees

The simplest form of tree is a binary tree. A binary tree is a tree in which each node has at most two descendants - a node can have just one but it can't have more than two.

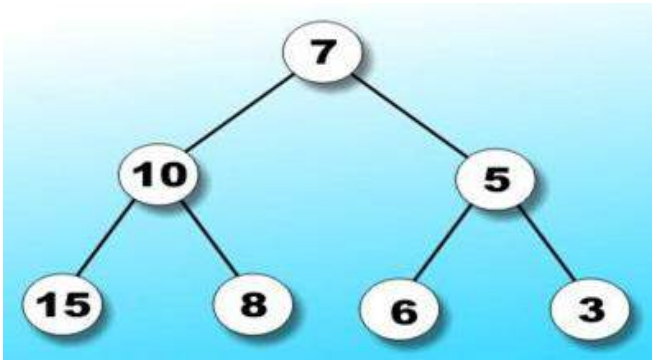
A binary tree consists of

a. a *node* (called the **root** node) and

b. left and right sub-trees.

Both the sub-trees are themselves binary trees.

The importance of a binary tree is that it can create a data structure that mimics a "yes/no" decision making process.



Root Node: Node at the "top" of a tree - the one from which all operations on the tree commence. The root node may not exist (a NULL tree with no nodes in it) or have 0, 1 or 2 children in a binary tree.

Leaf Node: Node at the "bottom" of a tree - farthest from the root. Leaf nodes have no children.

Complete Tree: Tree in which each leaf is at the same distance from the root. i.e. all the nodes have maximum two subtrees.

Height: Number of nodes which must be traversed from the root to reach a leaf of a tree.

4.9.3 GRAPHS

A graph is a set of *vertices* and *edges* which connect them. A *graph* is a collection of nodes called *vertices*, and the connections between them, called *edges*.

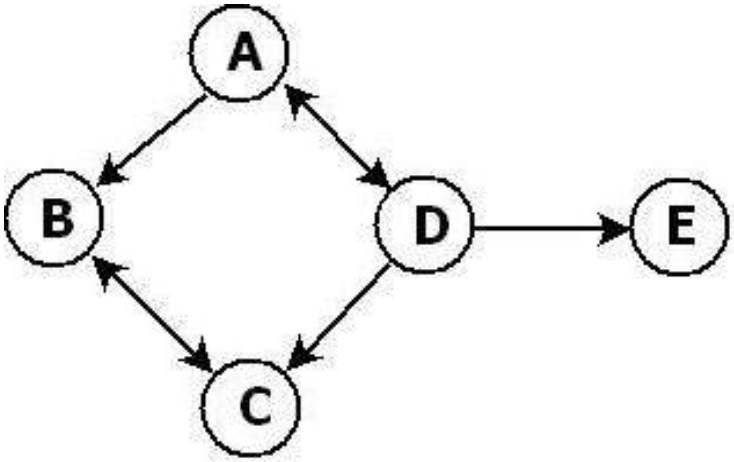
Undirected and directed graphs

When the edges in a graph have a direction, the graph is called a *directed graph* or *digraph*, and the edges are called *directed edges* or *arcs*.

Neighbors and adjacency

A vertex that is the end point of an edge is called a *neighbor* of the vertex, that is its starting-point. The first vertex is said to be *adjacent* to the second.

The following diagram shows a graph with 5 vertices and 7 edges. The edges between A and D and B and C are pairs that make a bidirectional connection, represented here by a double-headed arrow.



One mark questions

1. What are data structures?
2. Differentiate between one-dimensional and two-dimensional array.
3. Give any two examples for primitive data structures.
4. Mention any two examples for non-primitive data structures.
5. What are primitive data structures ?
6. What are non-primitive data structures ?
7. Name the data structure which is called LIFO list.
8. What is the other name of queue.
9. Define an array.
10. What are lists ?
11. What is meant by linear data structures ?
12. What are non-linear data structures ?
13. What is a stack ?
14. What is a queue ?
15. Name the data structure whose relationship between data elements is by means of links.
16. What is a linked list ?
17 .Mention any one application of stack .
18. What do you mean by traversal in data structure.
19. Define searching in one-dimensional array.
20. What is meant by sorting in an array.
21. Mention the types of searching in the array.
22. What is a binary tree.
23. What do you mean by depth of a tree.
24. What are the operations that can be performed on stacks
25. What are the operations that can be performed on queues ?
26. Define the term PUSH and POP operation in stack.
27. What is FIFO list ?
28. What is LIFO list ?
29. Mention the different types of queues.

Two marks questions:

1. How are data structure classified ?
2. Justify the need for using arrays.
3. How are arrays classified ?
4. Mention the various operations performed on arrays .
5. How do you find the length of the array ?
6. Mention the types of linked lists.
7. What is a stack ? Mention the types of operations performed on the stacks.
8. What is a queue ? Mention the various operations performed on the queue.

Three marks questions:

1. Mention the various operations performed on data structures.
2. Explain the memory representation of a one-dimensional array.
3. Explain the memory representation of a stack using one-dimensional array.
4. Explain the memory representation of queue using one-dimensional array.
5. Explain the memory representation of single linked list .
6. Define the following with respect to binary tree
   a. root b. subtree c. depth
7. Write an algorithm for traversal in a linear array.
8. Give the memory representation of two-dimensional array.

Five marks questions:

1. Write an algorithm to insert an element in an array.
2. Write an algorithm to delete an element in an array.
3. Write an algorithm to search an element in an array using binary search.
4. Write an algorithm to sort an array using insertion sort.
5. Write an algorithm for push and pop operation in stack using array.
6. Write an algorithm to insert a data element at the rear end of the queue.
7. Write an algorithm to delete a data element from the front end of the queue.
8. Write an algorithm to insert a data element at the beginning of a linked list.
9. Write an algorithm to delete a data element at the end of a linked list.
10. Apply binary search for the following sequence of numbers.
    10, 20, 30, 35, 40, 45, 50, 55, 60 search for item = 35

******