1.1 What is an algorithm?

An algorithm is a finite set of step by step instructions to solve a problem. In normal language, algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows:

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented as a program on a computer.

Properties

Every algorithm must satisfy the following properties:

1. **Definiteness** - Every step in an algorithm must be clear and unambiguous
2. **Finiteness –** Every algorithm must produce result within a finite number of steps.
3. **Effectiveness -** Every instruction must be executed in a finite amount of time.
4. **Input & Output** - Every algorithm must take zero or more number of inputs and must produce at least one output as result.

1.2 Performance Analysis

In computer science there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyses them and pick the one which is best suitable for our requirements. Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyze an algorithm, we consider only the space and time required by that particular algorithm and we ignore all remaining elements.

Performance analysis of an algorithm is performed by using the following measures:

1. Space Complexity
2. Time Complexity

1.2.1 What is Space complexity?

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
3. **Data Space:** It is the amount of memory used to store all the variables and constants.

NOTE: When we want to perform analysis of an algorithm based on its Space complexity, we consider only Data Space and ignore Instruction Space as well as Environmental Stack. That means we calculate only the memory required to store Variables, Constants, Structures, etc.,

Consider the following piece of code...

```
int  square (int  a)
{
        return  a*a;
}
```

In above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value. That means, totally it requires 4 bytes of memory to complete its execution.

1.2.2 What is Time complexity?

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution. Generally, running time of an algorithm depends upon the following:

1. Whether it is running on Single processor machine or Multi processor machine.
2. Whether it is a 32 bit machine or 64 bit machine
3. Read and Write speed of the machine.
4. The time it takes to perform Arithmetic operations, logical operations, return value and assignment operations etc.,

NOTE: When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent.

Consider the following piece of code...

```
Algorithm Search (A, n, x)
{
   // where A is an array, n is the size of an array and x is the item to be searched.
   For  i := 1 to n do
   {
      If (x==A[i]) then
        {
        Write (Item found at location i)
        }
    }
    Write (item not found)
}
```

For the above code, time complexity can be calculated as follows:

Cost is the amount of computer time required for a single operation in each line. Repetition is the amount of computer time required by each operation for all its repetitions, so above code requires '*n*' units of computer time to complete the task.

1.2.3 Asymptotic Notation

Asymptotic notation of an algorithm is a mathematical representation of its complexity. Majorly, we use THREE types of Asymptotic Notations and those are:

1. Big - Oh (O)
2. Omega (Ω)
3. Theta (Θ)

Big - Oh Notation (O)

- ✓ Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.
- ✓ Big - Oh notation always indicates the maximum time required by an algorithm for all input values.
- ✓ Big - Oh notation describes the worst case of an algorithm time complexity.
- ✓ It is represented as O(T)

Omega Notation (Ω)

- ✓ Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.
- ✓ Omega notation always indicates the minimum time required by an algorithm for all input values.
- ✓ Omega notation describes the best case of an algorithm time complexity.
- ✓ It is represented as **Ω** (T)

Theta Notation (Θ)

- ✓ Theta notation is used to define the average bound of an algorithm in terms of Time Complexity.
- ✓ Theta notation always indicates the average time required by an algorithm for all input values.
- ✓ Theta notation describes the average case of an algorithm time complexity.
- ✓ It is represented as **Θ** (T)

Example

Consider the following piece of code...

```
Algorithm Search (A, n, x)
{
   // where A is an array, n is the size of an array and x is the item to be searched.
   for i := 1 to n do
   {
      if(x==A[i]) then
        {
            Write (Item found at location i)
        }
    }
    Write (item not found)
}
```

The time complexity for the above algorithm

1. Best case is **Ω** (1)
2. Average case is **Θ** (n/2)
3. Worst case is **O** (n)

1.3 What is Data Structure?

The logical or mathematical model of data in a particular organization is called *data structure*. Data structures are generally classified into primitive and non- primitive data structures.

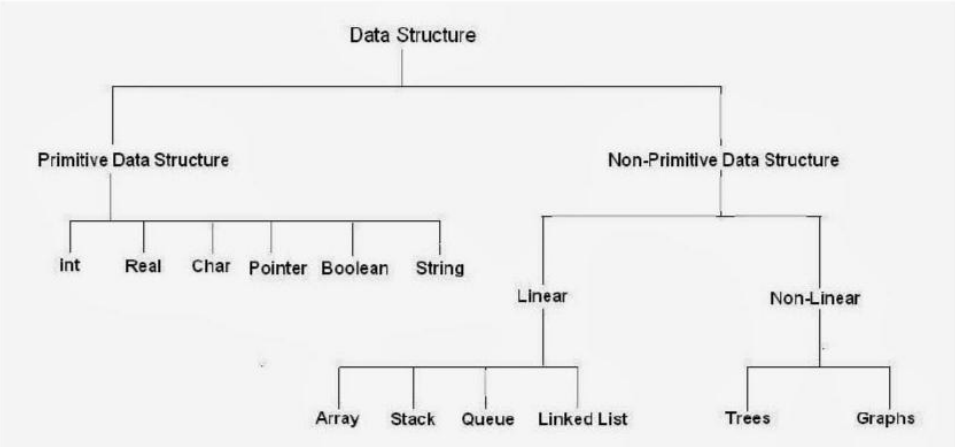


Fig. Classification of Data Structures

Based on the organizing method of a data structure, data structures are divided into two types.

1. Linear Data Structures
2. Non - Linear Data Structures

Linear Data Structures

If a data structure is organizing the data in sequential order, then that data structure is called as Linear Data Structure. For example:

1. Arrays
2. Linked List
3. Stacks
4. Queues

Non - Linear Data Structures

If a data structure is organizing the data in random order, then that data structure is called as Non-Linear Data Structure. For example:

1. Trees
2. Graphs
3. Dictionaries
4. Heaps , etc.,

Operations on Data Structures

The basic operations that are performed on data structures are as follows:

1. **Traversal:** Traversal of a data structure means processing all the data elements present in it exactly once.
2. **Insertion:** Insertion means addition of a new data element in a data structure.
3. **Deletion:** Deletion means removal of a data element from a data structure if it is found.
4. **Searching:** Searching involves searching for the specified data element in a data structure.
5. **Sorting:** Arranging data elements of a data structure in a specified order is called sorting.
6. **Merging:** Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

1.4 Abstract Data Type (ADT)

An Abstract Data type refers to set of data values and associated operations that are specified accurately, independent of any particular implementation.

(Or)

ADT is a user defined data type which encapsulates a range of data values and their functions.

(Or)

An **Abstract Data Type** is a mathematical model of a **data structure**. It describes a container which holds a finite number of objects where the objects may be associated through a given binary relationship.

Advantages:

- ✓ Code is easier to understand.
- ✓ Implementations of ADTs can be changed without requiring changes to the program that uses the ADTs.
- ✓ ADTs can be reused in future programs.

ADT Model

The ADT model is shown in below figure. It consists of two different parts:

1. Public part
2. Private part

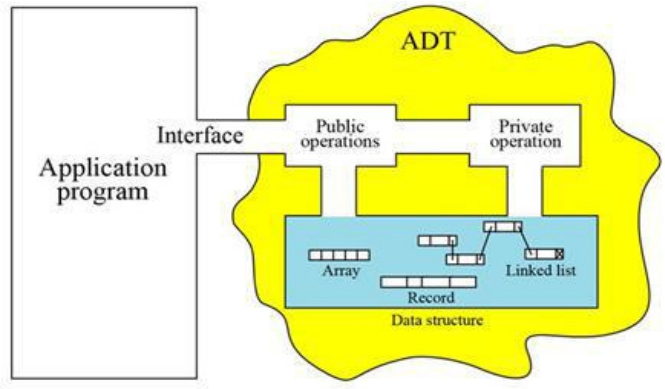


Figure : The model for an ADT

The **public** or **external** part, which consists of:

- ✓ The conceptual picture (the user's view of how the object looks like, how the structure is organized)
- ✓ The conceptual operations (what the user can do to the ADT)

- ✓ The representation (how the structure is actually stored).
- ✓ The implementation of the operations (the actual code)

1.5 Array as an ADT

The **array** is a basic abstract data type that holds an ordered collection of items accessible by an integer index. Since it's an ADT, it doesn't specify an implementation, but is almost always implemented by an array data structure or dynamic array.

1.5.1 Linear Array

An array is collection of homogeneous elements that are represented under a single variable name.

(Or)

A linear array is a list of a finite number of *n* homogeneous data elements (that is data elements of the same type) such that

- ✓ The elements are referenced respectively by an index set consisting of *n* consecutive numbers
- ✓ The elements are stored respectively in successive memory locations
- ✓ The number *n* of elements is called the *length* or *size* of the array.
- ✓ The index set consists of the integer 0,1, 2, … n-1.
- ✓ Length or the number of data elements of the array can be obtained from the index set by

  Length = UB – LB + 1

  Where UB is the largest index called the upper bound and LB is the smallest index called the lower bound of the arrays

- ✓ Element of an array A may be denoted by
  - – Subscript notation $A_1, A_2, , \ldots , A_n$
  - – Parenthesis notation A(1), A(2), …. , A(n)
  - – Bracket notation A[1], A[2], ….. , A[n]
  - – The number K in A[K] is called subscript or an index and A[K] is called a Subscripted variable

1.5.2 Representation of linear array in memory

- ✓ Let LA be a linear array in the memory of the computer.
- ✓ LOC(LA[K]) = address of the element LA[K] of the array LA
- ✓ The elements of LA are stored in the successive memory locations.

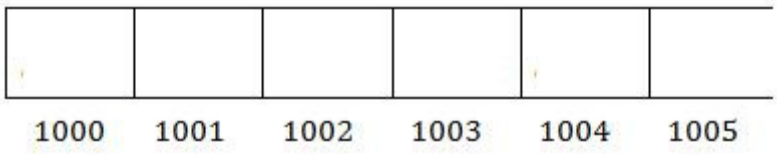


Fig. memory representation of an array of elements

Computer does not need to keep track of the address of every element of LA, but need to track only the address of the first element of the array denoted by

Base (LA)

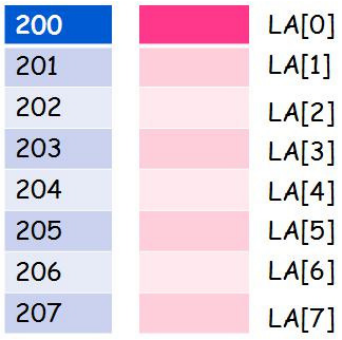
and called the *base address* of LA. Using this address, the computer calculates the address of any element of LA by the following formula:

LOC (LA [K]) = Base (LA) + w (K – LB)

Where **w** is the number of words per memory cell of the array LA [**w** is the size of the **data type**].

Example: Find the address for LA [6]. Each element of the array occupy 1 byte LOC (LA [K]) = Base (LA) + w (K – lower bound)

LOC (LA [6]) = 200 + 1(6 – 0) = 206



1.6 Operations on Arrays

The operation performed on any linear structure, where it can be an array or linked list, include the following

1. **Traversal:** processing each element in the list exactly once.
2. **Insertion:** adding new element to the list.
3. **Deletion:** removing an element from the list.
4. **Searching:** finding location of the element with a given value or key.
5. **Sorting:** Arranging elements in some type of order.
6. **Merging:** Combining two lists into a single list.

1.6.1 Traversing

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the content of each element of A or count the number of elements of A with a given property. This can be accomplished by *traversing* A, that is, by accessing and processing (frequently called *visiting*) each element of A exactly once.

The following algorithm traverses a linear array:

Algorithm: (Traversing a linear array.)

Here, A is a linear array with lower bound LB and upper bound UB. This algorithm traverses A applying an operation PROCESS to each element of A.

Step 1: [Initialize counter] Set I: = LB.
Step 2: Repeat steps 3 and 4 while I<= UB:
Step 3:[Visit element.] Apply PROCESS to A.
Step 4:[Increase counter.] Set I: = I+1.
[End of step 2 loop.]
Step 5: Exit.

Implementation of array traversing using C.

```
#include <stdio.h>
#include <conio.h>
void main()
{
        int i, a[5]={10,20,30,40,50};
        clrscr();
        printf("\n The array elements are ");
        for (i=0;i<5;i++)
        printf("\t %d", a[i]);
        getch();
}
```

Output

The array elements are 10 20 30 40 50

1.6.2 Insertion

Let A be a collection of data elements in the memory of the computer. “Inserting” refers to the operation of adding another element to the collection A,

Inserting an element at the ‘end’ of a linear array can be easily done. On the other hand, suppose we need to insert an element in the middle of the array. Then on the average, half of the elements must be moved downward to new locations to accommodate the new element and keep an order of the order of the other elements.

Algorithm: (Inserting into a linear array.)
INSERT (A, N, K, ITEM).
Here, A is a linear array with N elements and K is a positive integer such that K <= N.
This algorithm inserts an element ITEM into the K_{th} position in A.

Step 1: [Initialize counter.] Set I: = N.
Step 2: Repeat steps 3 and 4 while I >= K:
Step 3:[Move element downward.] Set A[I+1] := A[I].
Step 4:[Decrease counter.] I: = I-1.
[End of step 2 loop.]
Step 5: [Insert element.] Set A [K]:= ITEM.
Step 6: [Reset N.] Set N: = N+1.
Step 7: Exit.

Implementing array insertion algorithm using C.

```
#include<stdio.h>
#include<conio.h>

void  main()
 {
  Int  a[100], n, element, i, pos;
  clrscr();
  printf("\nEnter size of an array:");
  scanf("%d", &n);
  printf("\nEnter elements :");
  for (i = 0; i < n; i++)
         scanf("%d", &a[i]);
  printf("\nEnter the element to be inserted :");
  scanf("%d", &element);

  printf("\nEnter the position :");
  scanf("%d", &pos);

  //Create space at the specified location
  for (i = n; i >= pos; i--)
  {
    a[i] = a[i - 1];
  }

  n++;
  a[pos - 1] = element;

  //Print  the result of insertion
printf("\nResultant Array: ");
 for (i = 0; i < n; i++)
    printf(" %d", a[i]);
  getch();
}
```

Output
Enter size of an array : 5
Enter elements: 1 2 3 4 5
Enter the element to be inserted : 6
Enter the location : 2
Resultant Array: 1 6 2 3 4 5

1.6.3 Deletion

Let A be a collection of data elements in the memory of the computer. "Deleting" refers to the operation of removing one of the elements from A.

Deleting an element at the 'end' of an array presents no difficulties, but deleting an element somewhere in the middle of the array would require that each subsequent element is moved one location upward in order to 'fill up' the array.

Algorithm: (Deletion from a linear array.)
DELETE (A, K, N, ITEM).
Here, A is a linear array with N elements and K is a positive integer such that K<= N.
This algorithm deletes the K_{th} element from A.

Step 1: [Initialize counter.] Set ITEM: = A [K] and I: = K.
Step 2: Repeat steps 3 and 4 while I>N:
Step 3:[Move element upward.] A [I]:= A [I+1].
Step 4:[Increase counter.] Set I: = I+1.
[End of step 2 loop.]
Step 5: [Reset N.] Set N: = N-1.
Step 6: Exit.

Implementing array deletion algorithm using C

```
#include<stdio.h>
#include<conio.h>
void main()
{
  Int  a[100],  n,  i, pos;
  clrscr();
  printf("\nEnter size of an array:");
  scanf("%d", &n);
  printf("\nEnter elements :");
  for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
  printf("\n Enter position of the element to be deleted :");
  scanf("%d", &pos);
  while(pos<n)
  {
        a[pos] = a[pos+1];
        pos++;
  }
  n--;
  printf("\nResultant Array: ");
  for (i = 0; i < n; i++)
    printf(" %d", a[i]);
  getch();
}
```

Output
Enter size of an array : 5
Enter elements: 4 8 16 12 5
Enter position of element to be deleted : 2
Resultant Array: 4 8 12 5

1.6.4 Sorting

Sorting means arranging the elements of an array in specific order may be either ascending or descending. There are different types of sorting techniques are available:

1. Bubble sort
2. Selection sort
3. Insert sort
4. Merge sort
5. Quick sort etc.

Bubble sort, sometimes referred to as *sinking sort*, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order.

Let A be a list of 'n' numbers. Sorting A refers to the operation of re-arranging the elements of A, so they are in increasing order, i.e. so that

A [1] < A [2] < A [3] < …..... <A [N]

For example, Suppose A originally is the list 8, 4, 19, 2, 7, 13, 5, 16. After sorting, A is the list 2, 4, 5, 7, 8, 13, 16, 19.

Algorithm: (Bubble Sort.)
BUBBLE (A, N).
Here, A is an array with N elements. This algorithm sorts the elements in A.

Step 1: Repeat steps 2 to 4 for I=1 to N-1.
Step 2: Set J: = 1. [Initializes pass pointer J.]
Step 3: Repeat step 4 while J <= N-I: [Execute pass.]
Step 4: If A [J] > A [J+1], then:
[Swap the elements.]
TEMP: = A [J]
A [J]:= A [J+1]
A [J+1]:= TEMP
[End of If structure.]
Step 5: Set PTR: = PTR+1. [Increase pointer.]
[End of step 3 (inner) loop.]
[End of step 1 (outer) loop.]
Step 6: Exit.

/ * C Program to implement Bubble Sort Technique */

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int  a[100], n, i, j, temp;
   printf("Enter number of elements:");
   scanf("%d", &n);
   printf("Enter elements\n");

   for(i=0;i<n;i++)
   {
         printf("Enter a[%d]=",i);
         scanf("%d", &a[i]);


   }
```

Output

```
Enter number of elements: 5
Enter elements
Enter a[0] = 14
Enter a[1] = 5
Enter a[2] = 23
Enter a[3] = 9
Enter a[4] = 15
Sorted elements....
5        9        14       15       23
```

```
   for(i=0;i<n-1;i++)
   {
    for(j=i+1;j<n;j++)
          if(a[i]>a[j])
          {
            temp=a[i];
            a[i] = a[j];
            a[j] = temp;
          }
   }
   printf("Sorted elements ....\n");
   for(i=0;i<n;i++)
     printf("%3d",a[i]);
   printf("\n");
   getch();
}
```

1.6.5 Searching

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be *successful* and the searching process gives the location of that value in the array. If the value is not present in the array, the searching is said to be *unsuccessful*. There are two popular methods for searching the array elements:

1. Linear search
2. Binary search.

1.6.5.1 Linear Search

Linear search, also called as *sequential search*, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. It is mostly used to search an unordered list of elements. For example, if an array *a[]* is declared and initialized as,

Int a[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};

and the value to be searched is *VAL = 7*, then searching means to find whether the value '7' is present in the array or not. If yes, then it returns the position of its occurrence. Here, POS = 3 (index starting from 0).

Algorithm

LINEAR SEARCH (A, N, ITEM, LOC).

Here, A is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in A, or sets LOC: = -1 if the search is unsuccessful.

Step 1: [Initialize Location.] Set LOC: = -1.

Step 2: [Initialize Counter.] Set I: = 0.

Step 3: Repeat steps 3 and 4 while I<N [Search for ITEM.] IF

A [I] = ITEM

Set: = LOC: = I.

Write (Search successful or ITEM found)

[End of IF.]

Step 4: Set I: = I+1.
[End of Step 2 Loop]
Step 5: Write (Search unsuccessful or ITEM not found)
Step 6: Exit.

```
/* C Program to search an element in an array using the linear search */
#include <stdio.h>
#include <conio.h>
void main()
{
   int a[100],n,i,key,flag=0;
   printf("Enter number of
   elements:"); scanf("%d", &n);
   printf("Enter
   elements\n"); /* Read
   array elements */
   for(i=0;i<n;i++)
   {
         printf("Enter a[%d]=", i);
         scanf("%d", &a[i]);
   }
   printf("Enter an element to be earched:");
   scanf("%d", &key);

    for(i=0;i<n;i++)
   {

      if(key==a[i])
      {
         printf("%d is found at position %d\n", key,i);
             flag=1;
             break;
      }
   }
   if (flag==0)
   printf("%d is not found\n",key);
getch();

}
```

Output
Enter number of elements:
5 Enter elements Enter a[0]
= 14
Enter a[1] = 5
Enter a[2] = 23
Enter a[3] = 9
Enter a[4] = 15
Enter an element to be searched:
9 9 is found at position 3

1.6.5.2 Binary Search

Suppose A is an array which is sorted in increasing numerical order or, equivalently alphabetically. Then, there is an extremely efficient searching algorithm, called 'binary search', which can be used to find the location LOC of a given ITEM of information in A. It works efficiently with a sorted list.

Algorithm: BINARY SEARCH (A, LB, UB, ITEM, LOC).

Here, A is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variable BEG, END, and MID denote respectively, the beginning, end and middle locations of a segment of elements of A. This algorithm finds the location LOC of ITEM in A or sets LOC = -1.

```
Step 1: [Initialize Location.] SET LOC = -1
Step 2: [Initialize Bounds.]
        Set BEG := LB, END := UB and MID := (BEG+END)/2.
Step 3: Repeat steps 4 and 5 while BEG <= END and A[MID] != ITEM
Step 4: Set MID := (BEG+END)/2).
Step 5:         If A[MID] = ITEM then
                    Set LOC := MID
                Else If A[MID] > ITEM then
                    Set END := MID-1.
                Else
                        Set BEG := MID+1.
                [End of If structure.]
        [End of step 2 loop.]
Step 6: If LOC := -1 then
                Write (Search Unsuccessful or element not found).
         Else
                Write (Search Successful or element found at LOC).
Step 7: Exit.
```

```
/* C Program to search an element in an array using binary search. */
#include <stdio.h>
#include <conio.h>
void main()
{
        int a[25],i, n, key, high, low, mid, flag=0;
        printf("\n Enter the number of elements in the array: "); scanf("%d",
        &n);

        printf("\n Enter the elements\n");

        for(i=0;i<n;i++)

        {

                printf("Enter a[%d]=",i);

                scanf("%d", &a[i]);

        }

        printf("Enter the element to be searched: ");

        scanf("%d", &key);

        low = 0, high = n-1;

        while(high>=low)

        {

                mid = (low + high)/2;

                if (a[mid] == key)
                {
                        printf("\n %d is found at position %d", key, mid);
                        flag = 1;
                        break;
                }
                 else if (a[mid]>key)
                        high = mid-1;
                else
                        low = mid+1;

        }
```

Output

```
Enter the number of elements in the

array: 5

Enter the elements

Enter a[0] = 11

Enter a[1] = 26

Enter a[2] = 32

Enter a[3] = 49

Enter a[4] = 68

Enter the element to be searched: 49

49 is found at position 3
```

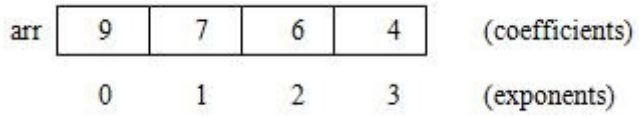
```
        if (flag == 0)
        printf("\n %d does not found in the array", key);
        getch();
}
```

1.7 Representation of polynomials using arrays

A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is

$P(x) = 4x^3+6x^2+7x+9$

A polynomial thus may be represented using arrays. A single dimensional array is used for representing a single variable polynomial. The index of such array can be considered as an exponent and coefficient can be stored at that particular index. The array representation for the above polynomial expression is given below:



Advantages:

1. Easy to handle

Drawbacks

1. It is time consuming
2. Wastage of memory space
3. We can change the size of an array

Algorithm for Addition of two polynomials

Assume that there are two polynomials P and Q. The polynomial result is stored in third array SUM.

Step 1: While P and Q are not null, repeat step 2.

Step 2: If powers of the two terms are equal then

```
        If the terms do not cancel then
            insert the sum of the terms into the SUM Polynomial
            Advance P
            Advance Q
        Else if the power of the first polynomial> power of second Then
            insert the term from first polynomial into SUM polynomial
            Advance P
        Else insert the term from second polynomial into SUM
            polynomial Advance Q
```

Step 3: Copy the remaining terms from the non empty polynomial into the SUM Polynomial.

Step 4: Print Sum

Step 5: Exit

/* Implementation of polynomial addition algorithm using C */

```
include <stdio.h>
#include <conio.h>
void main()
{
   int  a[6],b[6],c[6], i, flag=0;
```

```
   clrscr();
   for (i=0;i<6;i++)
          a[i]=b[i]=0;

  a[1]= 1; a[2]=4; a[5]=-7;

  b[0]= -14; b[1]=10;
  b[2]=6; b[3]= 5; b[4]=3;

printf ("Polynomial one is : ");          //Printing first polynomial
for (i=5;i>=0;i--)
{
    if(a[i]!=0 && i>0)
   {
          printf ("%d(x)%d ",a[i],i);
          flag=1;
   }
   if(i>0 && a[i-1]>0 && flag==1)
         printf ("+ ");
   if(a[i]!=0 && i==0)
          printf ("%d",a[i]);
}
flag=0;
printf ("\nPolynomial two is : ");     //Printing second polynomial
for (i=5;i>=0;i--)
{
     if(b[i]!=0 && i>0)
     {
          printf ("%d(x)%d ",b[i],i);
          flag=1;
     }
     if(i>0 && b[i-1]>0 && flag==1)
          printf ("+ ");
     if(b[i]!=0 && i==0)
          printf ("%d", b[i]);
}
for (i=0;i<6;i++)        //polynomial addition
          c[i] = a[i]+b[i];

printf ("\n Resultant Polynomial is : ");        //Printing Resultant polynomial
for (i=5;i>=0;i--)
{
    if(c[i]!=0 && i>0)
    {
          printf ("%d(x)%d ",c[i],i);
          lag=1;
     }
     if(i>0 && c[i-1]>0 && flag==1)
          printf ("+ ");
     if(c[i]!=0 && i==0)
```

```
        printf ("%d",c[i]);
}
getch();
}
```

Output

Polynomial one is : -7(x)5 + 4(x)2 +1(x)1
Polynomial two is : -3(x)4 + 5(x)3 + 6(x)2 + 10(x)1 – 14
Resultant polynomial is : -7(x)5 + 3(x)4 + 5(x)3 + 10(x)2 + 11(x)1 – 14

1.8 Multi-Dimensional Arrays

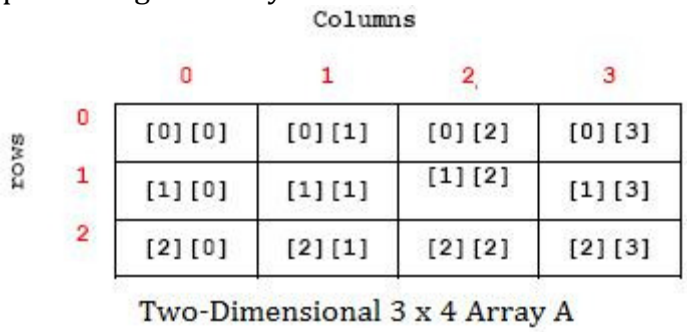
1.8.1 Two-Dimensional Array

A Two-Dimensional **m x n** array **A** is a collection of **m . n** data elements such that each element is specified by a pair of integers (such as J, K) called subscripts with property that

1 ≤ J ≤ m and **1 ≤ K ≤ n**

The element of **A** with first subscript **J** and second subscript **K** will be denoted by $A_{J,K}$ or **A[J][K]**

The two-dimensional arrays are also called matrices in mathematics and tables business applications. Hence 2d arrays are sometimes called matrix arrays. The standard way of representing 2-d arrays:



Two-Dimensional 3 x 4 Array A

Storage Representations

Let *A* be a two-dimensional m x n array. The array A will be represented in the memory by a block of m x n sequential memory locations. Programming language will store array A either

1. Column-Major Order
2. Row-Major Order

If a two-dimensional array can be represented as a single row with many columns and mapped sequentially is known as column-major representation.

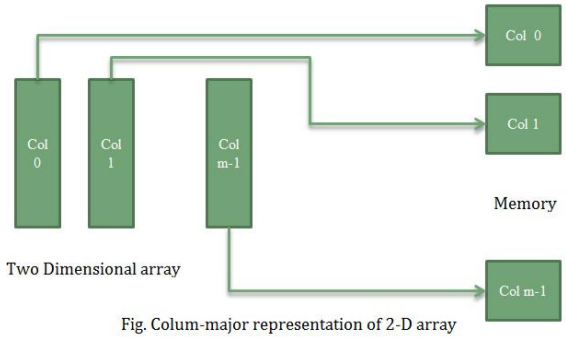


Fig. Colum-major representation of 2-D array

If a two-dimensional array can be represented as a single column with many rows and mapped sequentially is known as row-major representation.

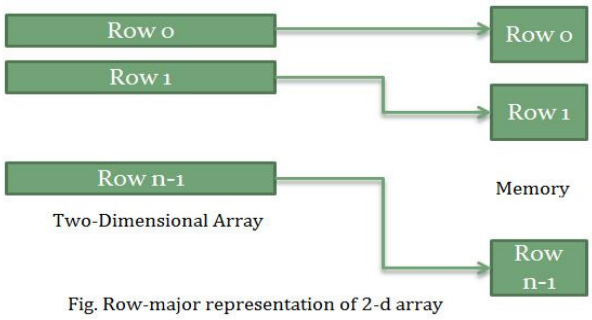
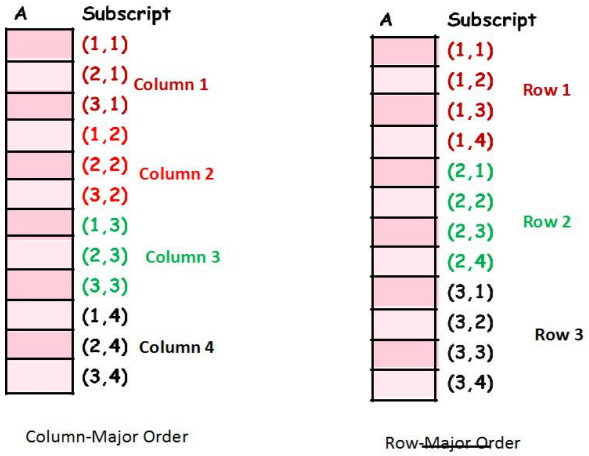


Fig. Row-major representation of 2-d array

Representation of 2-D Arrays in memory

Let A be a two-dimensional m x n array. The array A will be represented in the memory by a block of m x n sequential memory location



1.8.2 Multi dimensional arrays

An **n**-dimensional m_1 x m_2 x …. x m_n array **B** is a collection of $m_1.m_2...m_n$ data elements in which each element is specified by a list of **n** integers indices – such as K_1, K_2, …., K_n – called subscript with the property that

$1 \le K_1 \le m_1$, $1 \le K_2 \le m_2$, …. $1 \le K_n \le m_n$

The Element **B** with subscript K_1, K_2, …,K_n will be denoted by

$B_{K1,K2,\ldots,Kn}$ or $B[K_1,K_2,\ldots,K_n]$

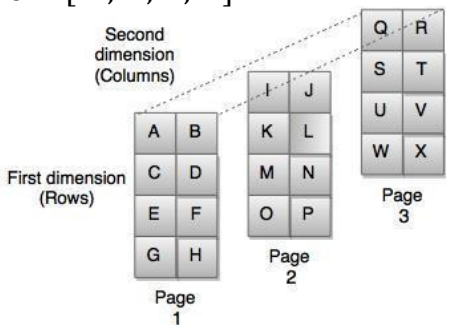


Fig: Multidimensional Array

1.8.3 Matrix multiplication

The following algorithm finds the product AB of matrices A and B. which are stored in two – dimensional arrays.

Algorithm: MATMUL(AB,C,M,P,N)

Let A be an MxP matrix array , and Let B be an PxN matrix array. This algorithm stores the product of A and B in MxN matrix array C.

```
Step 1: Repeat steps 2 to 4 for I = 1 to M:
Step 2:         Repeat steps 3 to 4 for J = 1 to N:
Step 3:                 Set C[I,J]:= 0 [Initialize C[I,J]
Step 4:                 Repeat for K = 1 to P
                                C[I,J]:= C[I,J] + A[I,K] * B[K,J]
                        [End of inner loop]
                [End of step 2 middle loop]
        [End of step 1 outer loop]
Step 5: Exit.
```

Example: Write a C program to perform multiplication of two matrices

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int a[10][10],b[10][10],c[10][10];
   int m,n,p,q,i,j,k;
   clrscr();
   printf("Enter number of rows and columns of first matrix(between 1 and
   10):"); scanf("%d%d",&m,&n);
   printf("Enter number of rows and columns of second matrix(between 1 and 10):");
   scanf("%d%d",&p,&q);
   if(n==p)
   {
         /* Read the elements of first matrix */
         printf("Enter elements of first matrix\n");
         for(i=0;i<m;i++)
         {
              for(j=0;j<n;j++)
                   scanf("%d",&a[i][j]);
          }
         /* Read the elements of second matrix */
         Print f("Enter elements of second matrix\n");
         for(i=0;i<p;i++)
         {
              for(j=0;j<q;j++)
                   scanf("%d",&b[i][j]);
          }
     /* Display A and B matrices */
     printf("The Matrix A\n");
     for(i=0;i<m;i++)
     {
           for(j=0;j<n;j++)
              printf("%3d",a[i][j]);
           printf("\n");
     }
     printf("The Matrix B\n");
     for(i=0;i<p;i++)
     {
```

```
            for(j=0;j<q;j++)
              printf("%3d",b[i][j]);
            printf("\n");
      }
      /*multiply two matrices and print resultant matrix */
      printf("The resultant matrix is\n");
      for(i=0;i<m;i++)
      {
            for(j=0;j<q;j++)
            {
              c[i][j]=0;              /* Initializing resultant matrix elements to zero */
              for(k=0;k<p;k++)
                    c[i][j] += a[i][k]*b[k][j];
              printf("%3d",c[i][j]);
            }

            printf("\n");
      }
    }
    else
     printf("Multiplication is not possible\n");
    getch();
}
```

Output

```
Enter number of rows and columns of first matrix (between 1 and 10):2 3
Enter number of rows and columns of second matrix (between 1 and 10):3 2
Enter elements of first matrix: 1 2 3 4 5 6
Enter elements of second matrix: 1 2 3 4 5 6
The Matrix A
1  2  3
4  5  6
The Matrix B
1   2
3   4
5   6
The resultant matrix is
22   28
49   64
```

1.9 Sparse Matrix

Matrices with a relatively high proportion of zero entries are called sparse matrices. Two general types of n-square sparse matrices occur in various applications. They are:

1. Triangular matrix
2. Tridiagonal matrix

In *triangular matrix*, all entries above the main diagonal are zero or equivalently, where nonzero entries can only occur on or below the main diagonal.

In *tridiagonal matrix*, where nonzero entries can only occur on the diagonal or on elements immediately above or below the diagonal.

$$\begin{pmatrix} 1 & & & & \\ 1 & 1 & & & \\ 1 & 2 & 1 & & \\ 1 & 3 & 3 & 1 & \\ 1 & 4 & 6 & 4 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 4 & & \\ 3 & 4 & 1 & \\ & 2 & 3 & 4 \\ & & 1 & 3 \end{pmatrix}.$$

(a) Triangular matrix (b) Tridiagonal matrix

Sparse Matrix Representations

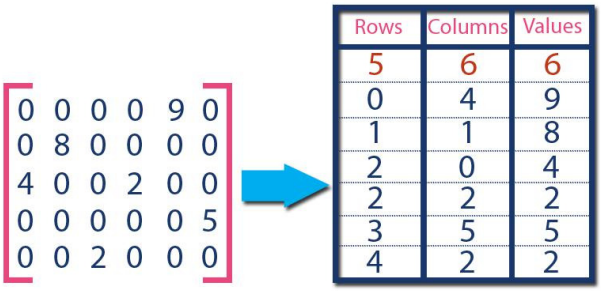
A sparse matrix can be represented by using two representations, those are:

1. Triplet Representation
2. Linked Representation

Triplet Representation

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the table:



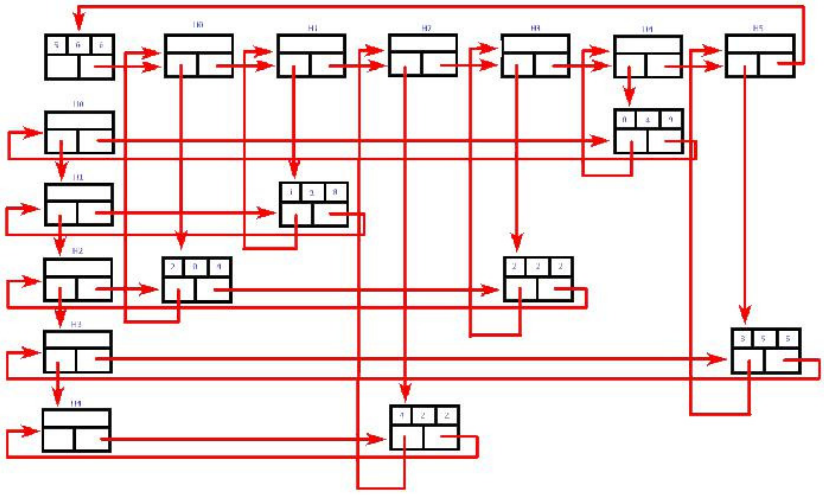
Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above table. Here the first row in the right side table is filled with values 5, 6 & 6 which indicate that it is a sparse matrix with 5 rows, 6 columns and 6 non-zero values. Second row is filled with 0, 4, and 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

Linked Representation

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely header node and element node. Header node consists of three fields and element node consists of five fields as shown in the fig.

Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below figure.



In above representation, H0, H1,...,H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to it's respective header node.

Example : C Program for transposing a sparse matrix

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int s[9][3], t[9][3];
   int  r, c, nzvs, i;
   clrscr();
   printf("\nEnter no. of rows, cols and non-zero
   elements:"); scanf("%d%d%d",&r,&c,&nzvs); s[0][0]=r;

   s[0][1]=c;
   s[0][2]=nzvs;

   for(i=1;i<=nzvs;i++)                    //reading sparse matrix
   {
         printf("Enter the next triplet(row,column,value):");
         scanf("%d%d%d",&s[i][0],&s[i][1],&s[i][2]);
   }

  printf("\n******Sparse matrix*********\n");          //printing sparse matrix
  printf("\nrow\t\tcolumn\t\tvalue\n");
  for(i=0;i<=nzvs;i++)
         printf("%d\t\t%d\t\t%d\n",s[i][0],s[i][1],s[i][2]);
```

```
  for(i=0;i<=nzvs;i++)                    //Transposing sparse matrix
    {
         t[i][0]=s[i][1];
         t[i][1]=s[i][0];
         t[i][2]=s[i][2];
    }
         //Printing transposed sparse matrix
   printf("\n*********After Transpose***********\n");
   printf("\nrow\t\tcolumn\t\tvalue\n"); for(i=0; i<=nzvs;
   i++)
         printf("%d\t\t%d\t\t%d\n",t[i][0],t[i][1],t[i][2]);
   getch();
}
```

Output

```
Enter no. of rows, cols and non-zero elements:6 6 8
Enter the next triplet(row, column, value): 0 0 15
Enter the next triplet(row, column, value): 0 3 22
Enter the next triplet(row, column, value): 0 5 -15
Enter the next triplet(row, column, value): 1 1 11
Enter the next triplet(row, column, value): 1 2 3
Enter the next triplet(row, column, value): 2 3 -6
Enter the next triplet(row, column, value): 4 0 91
Enter the next triplet(row, column, value): 5 2 28

******Sparse matrix*********
  row    column        value
   6         6            8
   0         0           15
   0         3           22
   0         5           -15
   1         1           11
   1         2            3
   2         3           -6
   4         0           91
   5         2           28

******After Transpose*********
  row    column        value
   6         6            8
   0         0           15
   3         0           22
   5         0           -15
   1         1           11
   2         1            3
   3         2           -6
   0         4           91
   2         5           28
```


2.1 STACK

Stack is a linear data structure and very much useful in various applications of computer science.

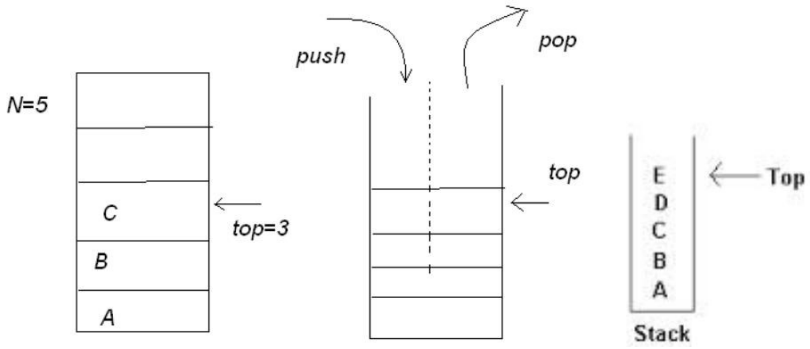
Definition

A stack is an ordered collection of homogeneous data elements, where the insertion and deletion operation takes place at one end only, called the top of the stack.

Like array and linked list, stack is also a linear data structure, but the only difference is that in case of former two, insertion and deletion operations can take place at any position.

In a stack the last inserted element is always processed first. Accordingly, stacks are called as Last-in-First-out (**LIFO**) lists. Other names used for stacks are **"piles"** and "**push-down lists**".

Stack is most commonly used to store local variables, parameters and return addresses when a function is called.



Stack Model of Stack

The above figure is a pictorial representation of a stack. N=5 is the maximum m capacity of the stack. Currently there are three elements in the stack, so the variable top value is 3.The variable top always keeps track of the top most element or position.

STACK OPERATIONS

5. PUSH: "**Push**" is the term used to insert an element into a stack.
6. POP: "**Pop**" is the term used to delete an element from a stack.

Two additional terms used with stacks are "overflow" & "underflow". Overflow occurs when we try to push more information on a stack that it can hold. Underflow occurs when we try to pop an item from a stack which is empty.

2.2 REPRESENTATION OF STACKS

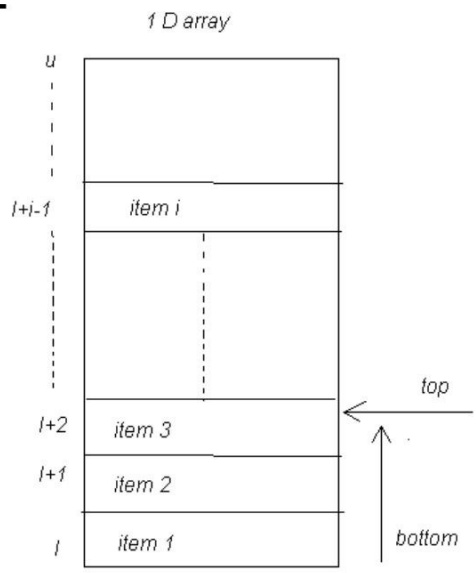
A stack may be represented in the memory in various ways. Mainly there are two ways. They are:

6. Using one dimensional arrays(Static Implementation)
7. Using linked lists(Dynamic Implementation)

2.2.1 Representation of Stack using Array

First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack. Then, starting from the first location of the memory block, items of the stack can be stored in sequential fashion.

In the figure *Item* denotes the i^{th} item in the stack; l & u denote the index range of array in use; usually these values are 1 and size respectively. Top is a pointer to point the position of array up to which it is filled with the items of stack. With this representation following two statuses can be stated:



EMPTY: Top<l

FULL: Top>=u +l-1

Algorithms for Stack Operations

Algorithm PUSH (STACK, TOP, MAXSTK, ITEM)

This algorithm pushes an ITEM onto a stack.

Step 1: If TOP = MAXST K, then: \\ Check Overflow?
Print: OVERFLOW, and Exit.
Step 2: Set TOP: = TOP + 1. \\ increases TOP by 1
Step 3: Set STACK[TOP] := ITEM. \\ Inserts ITEM in new TOP position.
Step 4: Exit.

Here we have assumed that array index varies from 1 to SIZE and Top points the location of the current top most item in the stack.

Algorithm POP (STACK, TOP, ITEM)

This algorithm deletes the top element of STACK and assigns it to the variable ITEM.

Step 1: If TOP = NULL, then: \\ Check Underflow.
Print: UNDERFLOW, and Exit.
Step 2: Set ITEM := STACK[TOP]. \\ Assigns TOP element to ITEM.
Step 3: Set TOP := TOP – 1. \\ Decreases TOP by 1.
Step 4: Exit.

/* C implementation of Stack using Arrays

***/** #include<stdio.h>

```
#include<conio.h>
#include<stdlib.h>
#define size 5
struct stack
{
        int s[size];
        int top;
} st;

int isfull() {
  if (st.top >= size - 1)
     return 1;
  else
     return 0;
}
void push(int item) {
```

```
   st.top++;
   st.s[st.top] = item;
}

int isempty() {
   if (st.top == -1)
      return 1;
   else
      return 0;
}

 int pop() {
   int item;
   item = st.s[st.top];
   st.top--;
   return (item);
}

void display() {
   int i;
   if (isempty())
      printf("\nStack Is Empty!");
   else {
      for (i = st.top; i >= 0; i--)
        printf("\n%d", st.s[i]);
   }
}

int main() {
   int item, choice;
   char ans;
   st.top = -1;
   clrscr();
   printf("\nImplementation Of Stack");
printf("\n----------------------------
   "); do {
      printf("\nMain Menu");
      printf("\n1.Push \n2.Pop \n3.Display
\n4.exit ");
      printf("\nEnter your
      choice:"); scanf("%d",
      &choice); switch (choice) {
      case 1:
        printf("\nEnter the item to be
pushed:");
        scanf("%d", &item);
        if (isfull())
          printf("\nStack is Full!");
        else
          push(item);
        break;
      case 2:
        if (isempty())
          printf("\nEmpty   stack…!   Underflow
!!");
        else {
          item = pop();
          printf("\nThe popped element is %d",
item);
        }
        break;
      case 3:
        display();
        break;
      case 4:
        exit(0);
      }
      printf("\nDo you want to
      continue?"); ans = getche();
   } while (ans == 'Y' || ans ==
   'y'); return 0;
}
```

2.2.2 Linked representation of Stacks

The linked representation of a stack, commonly termed linked stack is a stack that is implemented using a singly linked list. The INFO fields of the nodes hold the elements of the stack and the LINK fields hold pointers to the neighboring element in the stack. The START pointer of the linked list behaves as the TOP pointer variable of the stack and the null pointer of the last node in the list signals the bottom of the stack.

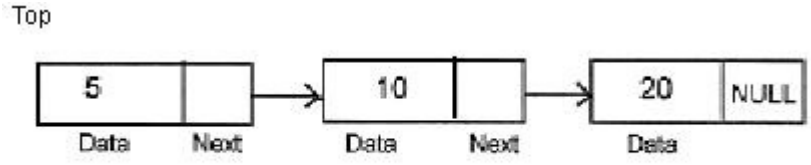


Fig. Linked list representation of a stack

Algorithms for Stack Operations

Algorithm: PUSH_LINKSTACK (INFO, LINK, TOP, AVAIL, ITEM).
This algorithm pushes an ITEM into a linked stack.

Step 1: If AVAIL = NULL, then: \\ Check Overflow?
Write: OVERFLOW, and Exit.
Step 2: [Take node from AVAIL list.]
Set NEW: = AVAIL, and AVAIL: = LINK [AVAIL].
Step 3: [Copies ITEM into new node.]
Set INFO [NEW]:= ITEM.
Step 4: [Add this node to the STACK.]
Set LINK [NEW]:= TOP, and TOP: = NEW.
Step 5: Exit.

Algorithm: POP_LINKSTACK (INFO, LINK, TOP, AVAIL, ITEM).

This algorithm deletes the top element of a linked stack and assigns it to the variable ITEM.

Step 1: If TOP = NULL, then: \\ Check Underflow?
Write: UNDERFLOW and Exit.
Step 2: Set ITEM: = STACK [TOP]. \\ Copies the TOP element of STACK into ITEM.
Step 3: Set TEMP := TOP, and
TOP: = LINK [TOP]. \\ Reset the position of TOP.
Step 4: [Add node to the AVAIL list.]
Set LINK[TEMP] := AVAIL, AVAIL: = TEMP.
Step 5: Exit.

2.3 APPLICATIONS OF STACKS

4. Reversing elements in a list or string.
5. Recursion Implementation.
6. Memory management in operating systems.
7. Evaluation of postfix or prefix expressions.
8. Infix to postfix expression conversion.
9. Tree and Graph traversals.
10. Checking for parenthesis balancing in arithmetic expressions.
11. Used in parsing.

2.4 Stack as ADT

The Stack ADT implementation in C is quite simple. Instead of storing data in each node, we store pointer to the data.

ADT implementation of Stacks

The node structure consists only of a data pointer and link pointer. The Stack head structure also contains only two elements – a pointer to the top of the stack and a count of the number of entries in the stack

2.5 Arithmetic Expressions

An expression is a collection of operators and operands that represents a specific value.
In above definition,

5. ***Operator*** is a symbol (+,-,>,<,..) which performs a particular task like arithmetic or logical or relational operation etc.,
6. ***Operands*** are the values on which the operators can perform the task.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows :

\{ Infix Expression

\{ Postfix Expression

\{ Prefix Expression

Infix Expression

In infix expression, operator is used in between operands. This notation is also called as ***polish notation***. The general structure of an Infix expression is:

<Operand1> <Operator> <Operand2>	Ex : A + B

Postfix Expression

In postfix expression, operator is used after operands. We can say that "Operator follows the Operands". This notation is also called as ***reverse - polish notation***. The general structure of Postfix expression is:

<Operand1> <Operand2> <Operator>	Ex : A B +

Prefix Expression

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator". The general structure of Prefix expression is:

<Operator><Operand1> <Operand2>	Ex : + A B

In 2nd and 3rdnotations, parentheses are not needed to determine the order evaluation of the operations in any arithmetic expression.

E.g.:	**Infix**	**Prefix**	**Postfix**	
1.	A+B*C	A+{*BC}	A+{BC*}	
		+A* BC	ABC* +	{ } indicate partial translation
2.	(A+B)*C	{+AB}*C	{AB+}*C	
		*+ABC	AB+C*	

The computer usually evaluates an arithmetic expression written in infix notation in two steps.

4. It converts the expression to postfix notation and
5. It evaluates the postfix expression.

In each step, the stack is the main tool that is used to accomplish the given task.

2.5.1 Transforming Infix Expression into Postfix Expression

In the infix expression we assume only exponentiation(^ or **), multiplication(*), division(/), addition(+), subtraction(-) operations. In addition to the above operators and operands they may contain left or right parenthesis. The operators three levels of priorities are:

Priority	Operators
HIGHEST	^ or **
NEXT HIGHEST	* , /
LOWEST	+ , –

The following algorithm transforms the infix expression Q into the equivalent postfix expression P. The algorithm uses a stack to temporarily ho ld operators and left parenthesis.

ALGORITHM INFIX-TO-POSTFIX (Q)

Step1: Push „(„ onto stack and add „)“ to the end of Q.

Step 2: Scan Q from left to right and repeat steps 3 to 6 for each element of Q until STACK is empty.

Step 3: If an operand is encountered, add it to P.

Step 4: If a left parentheses is encountered, push it onto STACK.

Step 5: If an operator Θ is encountered, then:

3. Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than Θ.

4. Add Θ to the STACK

[End of If Structure]

Step 6: If a right parentheses is encountered, then:

☐ Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parentheses is encountered.

☐ Remove the left parenthesis. [Do not add this to P]

Step 7: Exit.

EXAMPLE 1: Consider the following infix expression Q: A + (B * C - (D / E ^ F) * G) * H

Symbol Scanned	Stack Contents	Expression P
Initial	(	
(1) A	(	A
(2) +	(+	A
(3) (	(+ (	A
(4) B	(+ (	AB
(5) *	(+ (*	AB
(6) C	(+ (*	ABC
(7) -	(+ (-	ABC *
(8) (	(+ (- (	ABC*
(9) D	(+ (- (	ABC*D
(10) /	(+ (- (/	ABC*D
(11) E	(+ (- (/	ABC*DE
(12) ^	(+ (- (/ ^	ABC*DE
(13) F	(+ (- (/ ^	ABC*DE

(14))	(+ (-	ABC*DE^/
(15) *	(+ (- *	ABC*DE^/
(16) G	(+ (- *	ABC*DE^/G
(17))	(+	ABC*DE^/G*-
(18) *	(+ *	ABC*DE^/G*-
(19) H	(+ *	ABC*DE^/G*-H
(20))		ABC*DE^/G*-H * +

OUTPUT P: ABC*DE^/G*-H * + (POSTFIX FORM)

EXAMPLE 2: Consider the infix expression Q: (A+B)*C + (D-E)/F)

Symbol Scanned	Stack Contents	Expression P
Initial	(	
(1) (	((	
(2) A	((	A
(3) +	((+	A
(4) B	((+	AB
(5))	(	AB+
(6) *	(*	AB+
(7) C	(*	AB+C
(8) +	(+	AB+C*
(9) (	(+(	AB+C*
(10) D	(+(	AB+C* D
(11) -	(+(-	AB+C* D
(12) E	(+(-	AB+C* DE
(13))	(+	AB+C* DE-
(14) /	(+/	AB+C* DE-
(15) F	(+/	AB+C* DE-F
(16))		AB+C* DE-F/+

OUTPUT P: AB+C* DE-F/+ (POSTFIX FORM)

2.5.2 EVALUATION OF A POSTFIX EXPRESSION

Suppose P is an arithmetic expression written in postfix. The following algorithm uses a STACK to hold operands during the evaluation of P.

Algorithm EVALPOSTFIX (p)

Step 1: Add a right parentheses ") " at the end of P. //this acts as a sentinel

Step 2: Scan P from left to right and repeat steps 3 and 4 for each element of P until the Sentinel ") " is encountered.

Step 3: If an operand is encountered, push on STACK.

Step 4: If an operator Θ is encountered, then:

☐ Remove the two top elements of STACK, where A is the top element and B is the next- to-top element.

Evaluate B Θ A.

Push the result of (b) back on STACK

[End of If structure]

[End of step2 loop]

Step 5: Set VALUE equal to top element on STACK.

Step6: Return (VALUE)

Step 7: Exit.

EXAMPLE: Consider the following postfix expression:

P: 5, 6, 2, +, *, 12, 4, /, -

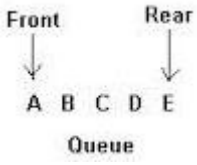
Symbols Scanned	STACK
(1) 5	5
(2) 6	5, 6
(3) 2	5, 6, 2
(4) +	5, 8
(5) *	40
(6) 12	40, 12
(7) 4	40, 12, 4
(8) /	40, 3
(9) -	37
(10))	

The result of this postfix expression is 37.

2.6 QUEUES

Queue is a linear data structure used in various applications of computer science. Like people stand in a queue to get a particular service, various processes will wait in a queue for their turn to avail a service.

Queue is a linear list in which insertions takes place at one end called the *rear* or *tail* of the queue and deletions at the other end called as *front* or *head* of the queue.

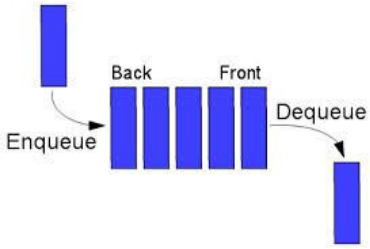


When an element is to join the queue, it is inserted at the rear end of the queue and when an element is to be deleted, the one at the front end of the queue is deleted automatically.

In queues always the first inserted element will be the first to be deleted. That"s why it is also called as *FIFO – First-in First-Out* data structure (or FCFS – First Come First Serve data structure).

APPLICATIONS of QUEUE

- ✓ CPU Scheduling (Round Robin Algorithm)
- ✓ Printer Spooling
- ✓ Tree & Graph Traversals
- ✓ Palindrome Checker
- ✓ Undo & Redo Operations in some Software"s



OPERATIONS ON QUEUE

The queue data structure supports two operations:

Enqueue: Inserting an element into the queue is called *enqueuing* the queue. After the data have been inserted into the queue, the new element becomes the rear.

Dequeue: Deleting an element from the queue is called *de queuing* the queue. The data at the front of the queue are returned to the user and removed from the queue.

TYPES OF QUEUES

- \{ **Simple or Single Ended Queue**: In this queue insertions take place at one end and deletions take place at other end.
- \{ **Circular Queue**: It is similar to single ended queue, but the front is connected back to rear. Here the memory can be utilized effectively.
- \{ **Double Ended Queue**: In double ended queue both the insertions and deletions can take place at both the ends.
- \{ **Priority Queue**: In priority queue the elements are not deleted according to the order they entered into the queue, but according to the priorities associated with the elements.

2.6.1 IMPLEMENTATION OF QUEUES

Queues can be implemented or represented in memory in two ways:

4. Using Arrays (Static Implementation).
5. Using Linked Lists (Dynamic Implementation).

2.6.1.1 Implementation of Queue Using Arrays

A common method of implementing a queue data structure is to use another sequential data structure, viz, arrays. With this representation, two pointers namely, Front and Rear are used to indicate two ends of the queue. For insertion of next element, pointer Rear will be adjusted and for deletion pointer Front will be adjusted.

However, the array implementation puts a limitation on the capacity of the queue. The number of elements in the queue cannot exceed the maximu m dimension of the one dimensional array. Thus a queue that is accommodated in an array Q[1:n], cannot hold more than *n* elements. Hence every insertion of an element into the queue has to necessarily test for a *QUEUE FULL* condition before executing the insertion operation. Again, each deletion has to ensure that it is not attempted on a queue which is already empty calling for the need to test for a *QUEUE EMPTY* condition before executing the deletion operation.

Algorithm of insert operation on a queue

```
Procedure INSERTQ (Q, n, ITEM, REAR)
// this procedure inserts an element ITEM into Queue with capacity n

Step 1:  if(REAR=n ) then
           Write:"QUEUE_FULL" and Exit
Step 2: REAR=REAR + 1              //Increment REAR
Step 3: Q[REAR]= ITEM              //Insert ITEM as the rear element
Step 4: Exit
```

It can be observed that addition of every new element int o the queue increments the REAR variable. However, before insertion, the condition whether the queue is full is checked.

Algorithm of delete operation on a queue

```
Procedure DELETEQ (Q, FRONT, REAR, ITEM)
Step 1:  If (FRONT >REAR) then:
                Write: "QUEUE EMPTY" and Exit.
Step 2:  ITEM = Q[FRONT]
```

Step 3: FRONT =FRONT + 1
Step 4: Exit.

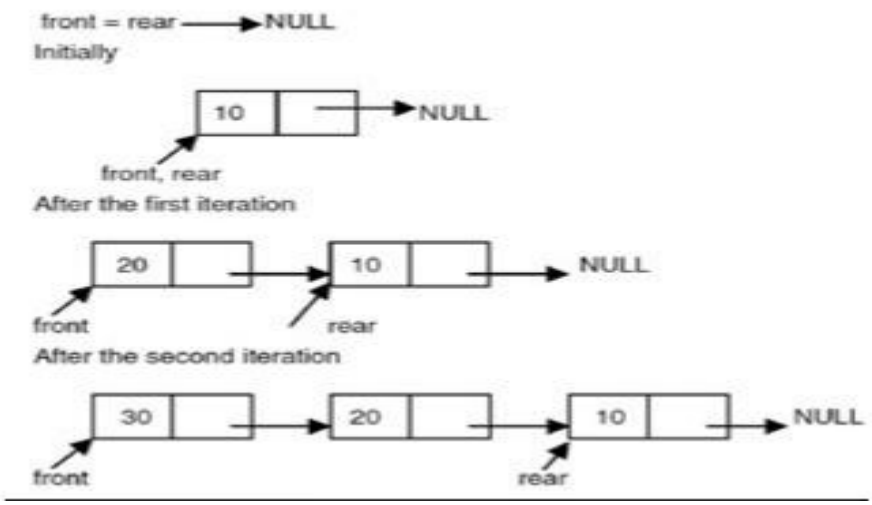
To perform delete operation, the participation of both the variables FRONT and REAR is essential. Before deletion, the condition FRONT =REAR checks for the emptiness of the queue. If the queue is not empty, the element is removed through ITEM and subsequently FRONT is incremented by 1.

/*C implementation of Queue using Arrays*/

```
#include <stdio.h>
#include <conio.h>
#define MAX 5

int insert(); /* function prototypes */
int delete();
void display();

int queue[MAX], rear = -1, front = - 1;

main()
{
   int choice;
   clrscr();
   printf("Implmentation of Queue\n");
   printf("-------------------- \n");
   while (1)
   {
        printf("1.Insert\n2.Delete\n3.Displa
y\n4.Quit\n");
        printf("Enter your choice :
        "); scanf("%d", &choice);
        switch (choice) {

           case 1:
           insert();
           break;
           case 2:
           delete();
           break;
           case 3:
           display();
           break;
           case 4:
           exit(1);
        } /*End of switch*/
   } /*End of while*/
} /*End of main()*/

int insert()
{
   int add_item;
   if (rear == MAX - 1)
        printf("Queue Overflow \n");
   else
   {
        if (front == - 1)
        /*If queue is initially empty */
        front = 0;
         printf("Inset the element in queue :
");
        scanf("%d", &add_item);
        rear = rear + 1;
        queue[rear] = add_item;
   }
   return;
} /*End of insert()*/

int delete()
{
   if (front == - 1 || front > rear)
   {
        printf("Queue Underflow
        \n"); return ;
   }
   else
   {
        printf("Element deleted from queue
is : %d\n", queue[front]);
        front = front + 1;
   }
   return;
} /*End of delete() */

void display()
{
   int i;
   if (front == - 1)
        printf("Queue is empty \n");
   else
   {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", queue[i]);
        printf("\n");
   }
} /*End of display() */
```

2.6.1.2 Linked Representation

A linked queue is a queue implemented as a linked list with two pointer variables FRONT and REAR pointing to the nodes which is in the FRONT and REAR of the queue.



Algorithm: LINKQ_INSERT (INFO, LINK, FRONT, REAR, ITEM, AVAIL).
This algorithm inserts an element ITEM into a linked queue.

Step 1: [OVERFLOW?]
If AVAIL: = NULL, then:
Write: OVERFLOW and Exit.
Step 2: [Remove first node from AVAIL list.]
Set NEW: = AVAIL, and
AVAIL: = LINK [AVAIL].
Step 3: Set INFO [NEW]:= ITEM, and
LINK [NEW]:= NULL. [Copies ITEM into new node.]
Step 4: If FRONT = NULL, then:
Set FRONT: = NEW and REAR: = NEW.
[If Q is empty then ITEM is the first element in the queue Q.]
Else:
Set LINK [REAR] := NEW, and
REAR: = NEW.
[REAR points to the new node appended to the end of the list.]
Step 5: Exit.

Algorithm: LINKQ_DELETE (INFO, LINK, FRONT, REAR, ITEM, AVAIL).
This algorithm deletes the front element of the linked queue and stores it in ITEM.

Step 1: [UNDERFLOW?]
If FRONT = NULL, then:
Write: UNDERFLOW and Exit.
Step 2: Set ITEM: = INFO [FRONT]. [Save the data value of FRONT.]
Step 3: Set NEW := FRONT, and [Reset FRONT to the next position.]
Set FRONT: = LINK [FRONT].
Step 4: Set LINK[NEW] := AVAIL, and [Add node to the AVAIL list.]
AVAIL := NEW
Step 5: Exit.

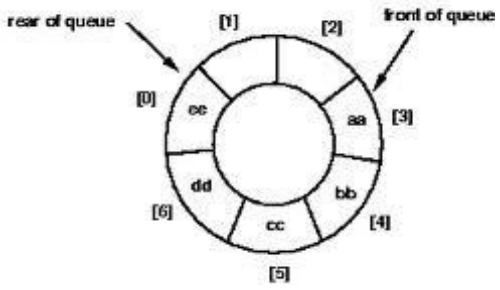
2.7 CIRCULAR QUEUE

One of the major problems with the linear queue is the lack of proper utilization of space. Suppose that the queue can store 10 elements and the entire queue is full. So, it means that the queue is holding 10 elements. In case, some of the elements at the front are deleted, the element at the last position in the queue continues to be at the same position and there is no efficient way to find out that the queue is not full.

In this way, space utilization in the case of linear queues is not efficient. This problem is arising due to the representation of the queue.

The alternative representation is to depict the queue as circular. In case, we are representing the queue using arrays, then, a queue with *n* elements starts from index 0 and ends at n-1. So, clearly, the first element in the queue will be at index 0 and the last element will be at n-1 when all the positions between index 0 and n-1 (both inclusive) are filled. Under such circumstances, front will point to 0 and rear will point to n-1. However, when a new element is to be added and if the rear is pointing to n-1, then, it needs to be checked if the position at index 0 is free. If yes, then the element can be added to that position and rear can be adjusted accordingly. In this way, the utilization of space is increased in the case of a circular queue.

In a circular queue, front will point to one position less to the first element. So, if the first element is at position 4 in the array, then the front will point to position 3. When the circular queue is created, then both front and rear point t o index 1. Also, we can conclude that the circular queue is empty in case both front and rear point to the same index. Figure depicts a circular queue.



Enqueue(value) - Inserting value into the Circular Queue

Step 1: Check whether queue is FULL.

If ((REAR == SIZE-1 && FRONT == 0) || (FRONT == REAR+1))

Write "Queue is full " and Exit

Step 2: If (rear == SIZE - 1 && front != 0) then:

SET REAR := -1.

Step 4: SET REAR = REAR +1

Step 5: SET QUEUE[REAR] = VALUE

Step 6: Exit and check 'front == -1' if it is TRUE, then set front = 0.

deQueue() - Deleting a value from the Circular Queue

Step 1: Check whether queue is EMPTY? If

(FRONT == -1 && REAR == -1)

Write "Queue is empty" and

Exit. Step 2: DISPLAY QUEUE [FRONT]

Step 3: SET FRONT: = FRONT +1.

Step 4: If (FRONT = SIZE, then:

SET FRONT: = 0.

Step 5: Exit

/* Program to implement Circular Queue using Array */

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5

int cinsert(int);
int cdelete();
void display();

int cq[SIZE], front = -1, rear = -1;
void main()
{
```

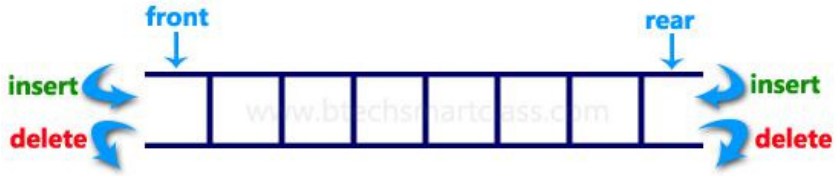
```
   int choice, ele;
   clrscr();
   while(1){
      printf("CIRCULAR QUEUE
IMPLEMENTATION\n");
      printf("---------------------------- \n");
      printf("****** MENU ******\n");
      printf("1. Insert\n2. Delete\n3.
Display\n4. Exit\n");
      printf("Enter your choice:
      "); scanf("%d",&choice);
      switch(choice){
          case 1: printf("\nEnter the value to
be insert:  ");
                  scanf("%d",&ele);
                  cinsert(ele);
                  break;
          case 2: cdelete();
                  break;
          case 3: display();
                  break;
          case 4: exit(1);
      } /*End of switch */
   } /*End of while */
}  /*End of main */

int cinsert(int value)
{
   if((front == 0 && rear == SIZE - 1)
|| (front == rear+1))
      printf("\nCircular Queue is Full!
Insertion not possible!!!\n");
   else{
      if(rear == SIZE-1 && front != 0)
          rear = -1;
      cq[++rear] = value;
      printf("\nInsertion
      Success!!!\n"); if(front == -1)
          front = 0;
   }
   return;
}  /*End of cinsert() */

int cdelete()
{
   if(front == -1 && rear == -1)
      printf("\nCircular Queue is Empty!
Deletion is not possible!!!\n");
   else{
      printf("\nDeleted element :
%d\n",cq[front ++]);
      if(front == SIZE)
          front = 0;
      if(front-1 == rear)
          front = rear = -1;
   }
   return;
} /*End of cdelete() */

void display()
{
   if(front == -1)
      printf("\nCircular Queue
is Empty!!!\n");
   else{
      int i = front;
      printf("\nCircular Queue Elements are :
\n");
      if(front <= rear){
          while(i <= rear)
             printf("%d\t",cq[i++]);
      }
      else{
          while(i <= SIZE - 1)
             printf("%d\t", cq[i++]);
          i = 0;
          while(i <= rear)
             printf("%d\t",cq[i++]);
      }
   }
} /*End of display() */
```

2.8 DOUBLE ENDED QUEUE (DEQUEUE)

A Dequeue is a homogeneous list of elements in which insertions and deletion operations are performed on both the ends.Because of this property it is known as double ended queue i.e. Dequeue or deck. Deque has two types:

5. Input restricted queue: It allows insertion at only one end
6. Output restricted queue: It allows deletion at only one end

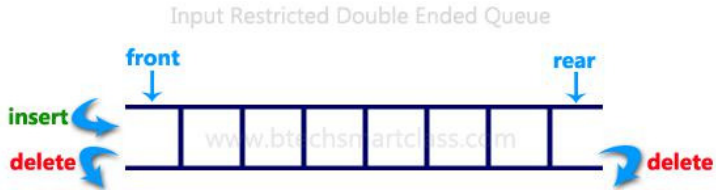
In dequeue four pointers are used. They are left front(lf), left rear(lr), right front(rf) and right rear(rr).



7. *If (lf==lr) and (rf==rr) then deque is empty.*
8. *If lr>rr then dequeue is full*
9. For inserting we have to modify rear pointer. For deleting we have to modify front pointer.
10. Always rear pointer is 1 position ahead of last element.
11. After insertion on left side, left rear should be incremented.
12. After insertion on right side, right rear should be decremented.

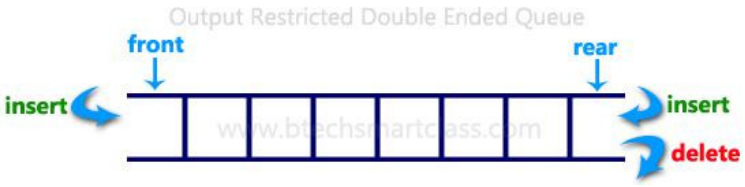
Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



2.9 PRIORITY QUEUE

Def: *Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority*.

In priority queue every element is associated with some priority. Normally the priorities are specified using numerical values. In some cases lower values indicate high priority and in some cases higher values indicate high priority

In priority queues elements are processed according to their priority but not according to the order they are entered into the queue.

For example, let P be a priority queue with three elements *a, b, c* whose priority factors are 2,1,1 respectively. Here, larger the number, higher is the priority accorded to that element. When a new element *d* with higher priority 4 is inserted, *d* joins at the head of the queue superseding the remaining elements. When elements in the queue have the same priority, then the priority queue behaves as an ordinary queue following the principle of FIFO amongst such elements.

The working of a priority queue may be likened to a situation when a file of patients waits for their turn in a queue to have an appoint ment with a doctor. All patients are accorded equal priority and follow an FCFS scheme by appoint ments. However, when a patient with bleeding injuries is brought in, he/she is accorded higher priority and is immediately moved to the head of the queue for immediate attention by the doctor. This is priority queue at work.

There are two types of priority queues they are as follows...

- ☐ Max Priority Queue
- ☐ Min Priority Queue

Max Priority Queue

In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2.

There are two representations of max priority queue.

3. Using One-Way List Representation
4. Using an Array

One-Way List Representation

One way to maintain a priority queue in memory is by means of a one -way list, as follows:

- ☐ Each node in the list will contain three items of information:
  - an information Feld INFO
  - a priority number PRN, and
  - a link number LINK
- ☐ A node X precedes a node Y in the list
  - ☐ when X has higher priority than Y or
  - ☐ when both have same priority but X was added to the list before Y

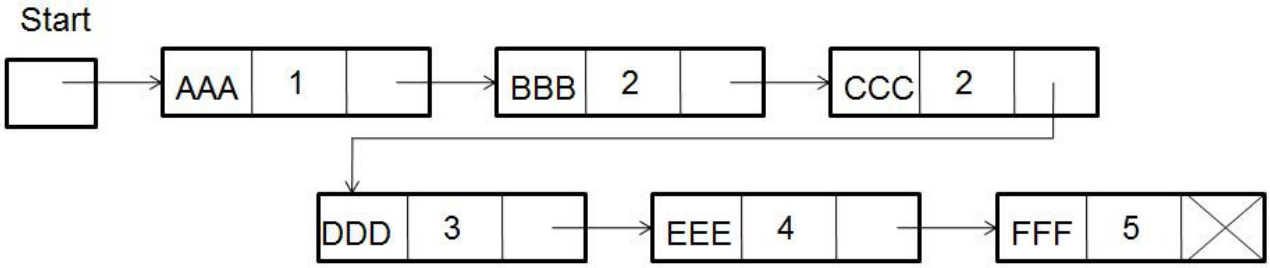


Fig.: Representation of Linked

list Algorithms for insertion and deletion

Insertion: Find the location of Insertion
Step 1: Add an ITEM with priority number N
Step 2: Traverse the list until finding a node X whose priority exceeds N. Insert ITEM in front of node X.
Step 3: If no such node is found, insert ITEM as the last element of the list.

Deletion: Delete the first node in the list.

Array representation

- ✓ Separate queue for each level of priority.
- ✓ Each queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR.
- ✓ If each queue is given the same amount space then a 2D queue can be used

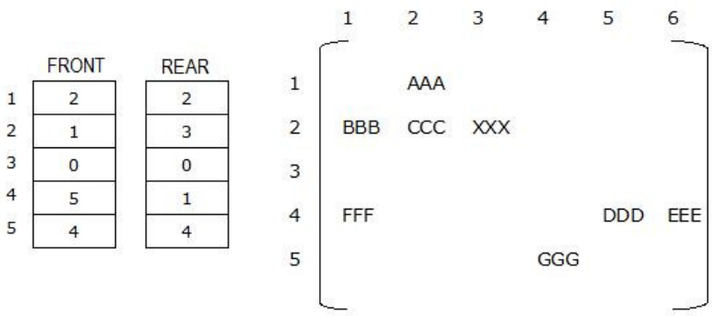


Fig. Array Representation with multiple queues

Deletion Algorithm

Step 1: Find the smallest K such that FRONT[K] ≠ NULL
Step 2: Delete and process the front element in row K of QUEUE
Step 3: Exit

Insertion Algorithm

Step 1: Insert ITEM as the rear element in row M of QUEUE
Step 2: Exit

Min Priority Queue

In min priority queue, elements are inserted in the order in which they arrive the queue and always minimu m value is removed first from the queue. For example assume that we insert in order 8, 3, 2, 5 and they are removed in the order 2, 3, 5, 8.

UNIT-VI
SORTING TECHNIQUES

Sorting is a technique to rearrange the list of elements either in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order. Sorting can be classified in two types:

a) Internal Sorting

If the data to be sorted remains in main memory and also the sorting is carried out in main memory it is called internal sorting. Internal Sorting takes place in the main memory of a computer. The internal sorting methods are applied to small collection of data. It means that, the entire collection of data to be sorted in small enough that the sorting can take place within main memory.

The following are some internal sorting techniques:

1. Insertion sort
2. Selection sort
3. Merge Sort
4. Radix Sort or Bucket Sort (Both or Same)
5. Quick Sort
6. Heap Sort
7. Bubble Sort

b) External Sorting

If the data resides in secondary memory and is brought into main memory in blocks for sorting and then result is returned back to secondary memory is called external sorting.

External sorting is required when the data being sorted do not fit into the main memory (usually RAM) and instead they must reside in the slower external memory (usually a hard drive).

The following are some external sorting techniques:

1. Two-Way External Merge Sort
2. K-way External Merge Sort

Insertion Sort

Insertion Sort iterates through a list of data items. Each data item is inserted at the correct position within a sorted list composed of those data items already processed.

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

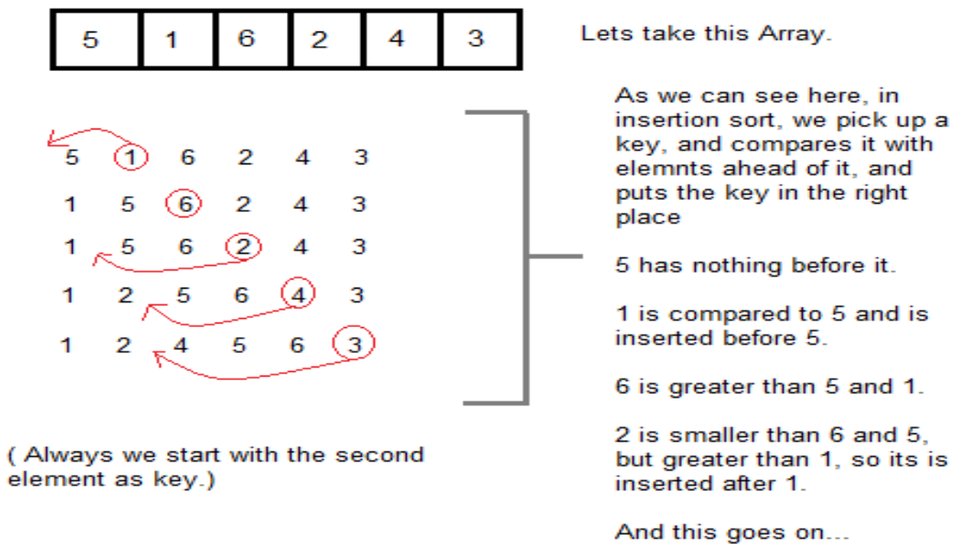
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large d a t a sets as its average and worst case complexity are of O(n2), where n is the number of items.

Insertion sort is a faster and more improved sorting algorithm than selection sort. In selection sort the algorithm iterates through all of the data through every pass whether it is already sorted or not. However, insertion sort works differently, instead of iterating through all of the data after every pass the algorithm only traverses the data it needs to until the segment that is being sorted is sorted.

Insertion Sort Algorithm

Step 1 – If it is the first element, it is already sorted. return 1;
Step 2 – Pick next element
Step 3 – Compare with all elements in the sorted sub-list
Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 – Insert the value
Step 6 – Repeat until list is sorted

Working and Examples of Insertion Sort



Time complexity of Insertion Sort

Time complexity, T (n) =1+2+3+4+5+.....+n-1
=n (n-1)/2
=O (n^2)

```
#include <stdio.h>

int main()
{
  int n, array[1000], c, d, t;
   printf("Enter number of elements\n");
  scanf("%d", &n);
   printf("Enter %d integers\n", n);
   for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
   for (c = 1 ; c <= n - 1; c++) {
    d = c;
     while ( d > 0 && array[d-1] > array[d]) {
      t = array[d];
      array[d]   = array[d-1];
      array[d-1] = t;
       d--;
    }
  }
   printf("Sorted list in ascending order:\n");
   for (c = 0; c <= n - 1; c++) {
    printf("%d\n", array[c]);
  }
   return 0;
}
```

Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

In this method, the elements are divided into partitions until each partition has sorted elements. Then, these partitions are merged and the elements are properly positioned to get a fully sorted list.

Working of Merge Sort

To understand merge sort, we take an unsorted array as the following –

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

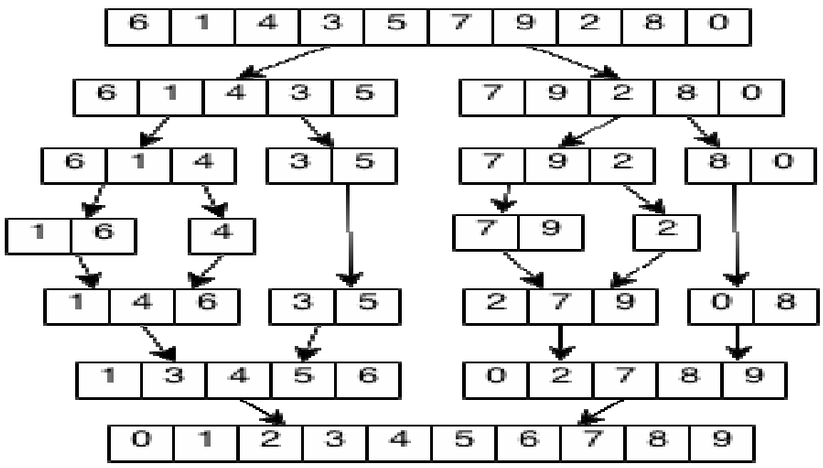
After the final merging, the list should look like this –

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 – if it is only one element in the list it is already sorted, return.
Step 2 – divide the list recursively into two halves until it can no more be divided.
Step 3 – merge the smaller lists into new list in sorted order.

Example on Merge




```
#include <stdio.h>
void merge(int [], int, int [], int, int[]);
 int main() {
  int a[100], b[100], m, n, c,
sorted[200];
 printf("Input number of elements in first array\n");
  scanf("%d", &m);
 printf("Input %d integers\n", m);
  for (c = 0; c < m; c++) {
    scanf("%d", &a[c]);
  }
 printf("Input number of elements in second array\n");
  scanf("%d", &n);
printf("Input %d integers\n", n);
  for (c = 0; c < n; c++) {
    scanf("%d", &b[c]);
  }
 merge(a, m, b, n, sorted)
printf("Sorted array:\n");
 for (c = 0; c < m + n; c++) {
    printf("%d\n", sorted[c]);
  }
 return 0;
}
void merge(int a[], int m, int b[], int
n, int sorted[]) {
  int i, j, k;
   j = k = 0;
  for (i = 0; i < m + n;) {
   if (j < m && k < n) {
     if (a[j] < b[k]) {
       sorted[i] = a[j];
```

```
      j++;
     }
     else {
      sorted[i] = b[k];
      k++;
     }
     i++;
    }
    else if (j == m) {
     for (; i < m + n;) {
      sorted[i] = b[k];
      k++;
      i++;
     }
    }
    else {
     for (; i < m + n;) {
      sorted[i] = a[j];
      j++;
      i++;
     }
    }
  }
}
```

Quick Sort

In Quick sort, an element called pivot is identified and that element is fixed in its place by moving all the elements less than that to its left and all the elements greater than that to its right.

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

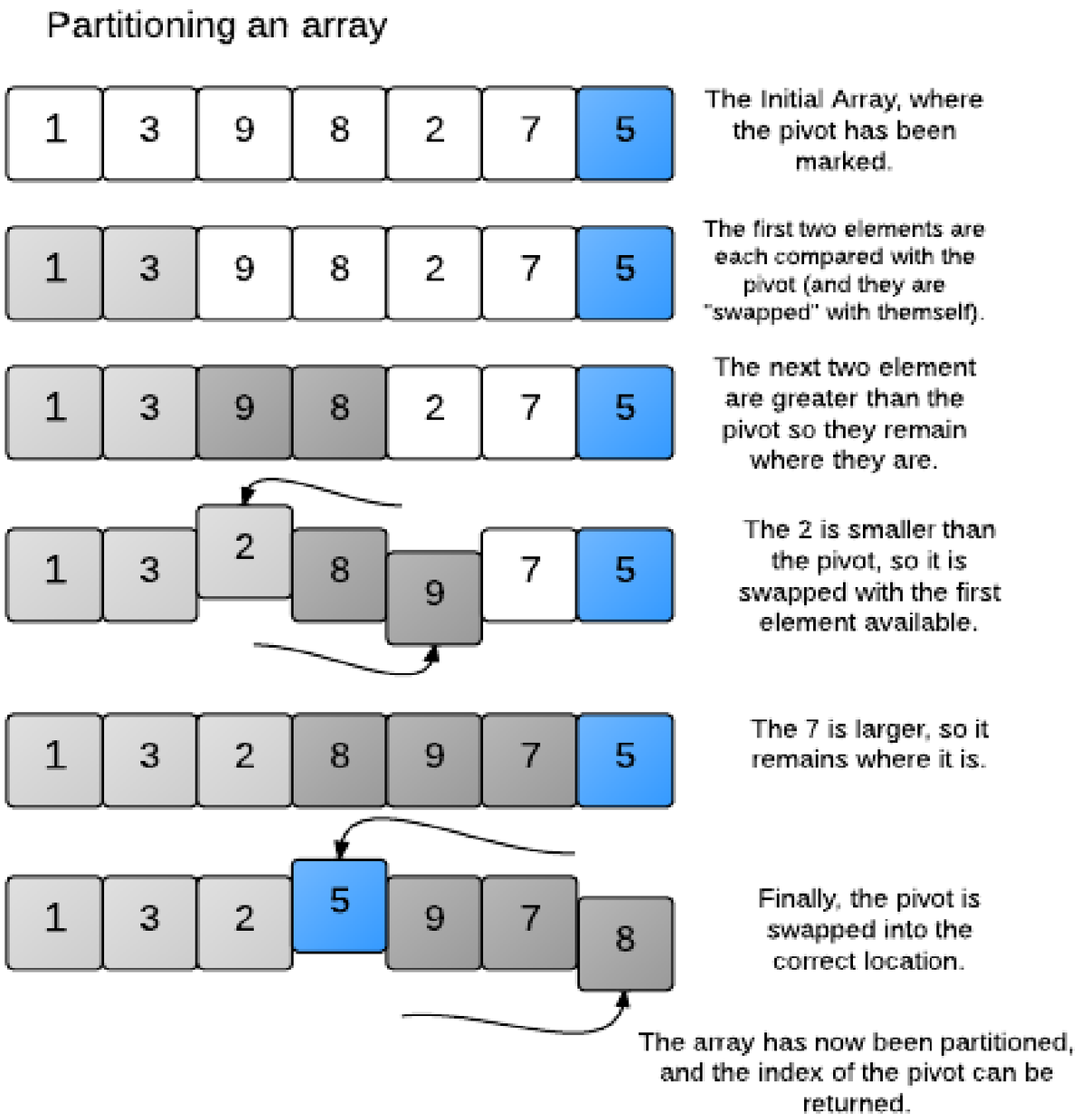
Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of O(nlogn), where n is the number of items.

Quick Sort Partition Algorithm

Step 1 – Choose the lowest index value has pivot
Step 2 – Take two variables i and j to point left and right of the list respectively
Step 3 – 'i' points to the low index
Step 4 – 'j' points to the high index
Step 5 – while value at a[i] is less than pivot move 'i' right
Step 6 – while value at a[j] is greater than pivot move 'j' left
Step 7 – if both step 5 and step 6 does not match swap a[i] and a[j]
Step 8 – if left ≥ right, swap pivot and a[j], where partition of the list occurs in such a way that all the elements in the left partition are less than pivot and all the elements of in the right partition are greater than pivot.

Step 1 – Make the left-most index value pivot **Step 2** – partition the array using pivot value **Step 3** – quicksort left partition recursively **Step 4** – quicksort right partition recursively

Working of Quick Sort




```
#include <stdio.h>
int partition(int a[], int beg, int end);
void quickSort(int a[], int beg, int end);
void main()
{
   int i;
   int arr[10]={90,23,101,45,65,23,67,89,34,23};
   quickSort(arr, 0, 9);
   printf("\n The sorted array is: \n");
   for(i=0;i<10;i++)
   printf(" %d\t", arr[i]);
}
int partition(int a[], int beg, int end)
{

   int left, right, temp, loc, flag;
   loc = left = beg;
   right = end;
   flag = 0;
   while(flag != 1)
   {
      while((a[loc] <= a[right]) && (loc!=right))
      right--;
```

```
    if(loc==right)
    flag =1;
    else if(a[loc]>a[right])
    {
       temp = a[loc];
       a[loc] = a[right];
       a[right] = temp;
       loc = right;
    }
    if(flag!=1)
    {
       while((a[loc] >= a[left]) && (loc!=left))
       left++;
       if(loc==left)
       flag =1;
       else if(a[loc] <a[left])
       {
          temp = a[loc];
          a[loc] = a[left];
          a[left] = temp;
          loc = left;
       }
    }
  }
  return loc;
}
void quickSort(int a[], int beg, int end)
{

  int loc;
  if(beg<end)
  {
    loc = partition(a, beg, end);
    quickSort(a, beg, loc-1);
    quickSort(a, loc+1, end);
  }
}
```

Time Complexity of Quick Sort:

Average Case: Let the average case value be $T_A(n)$.
Under the assumptions, the partitioning element v has an equal probability of being the ith smallest element, $1 \le i \le p-m$ in a[m:p-1]. Hence the two subarrays remaining to be sorted are a[m:j] and a[j+1:p-1] with probability 1/(p-m), $m \le j < p$.
From this recurrence obtained is

$$T_A(n) = (n+1) + 1/n \sum_{1 \le k \le n} [T_A(k-1) + T_A(n-k)]$$

$T_A(n) = O(n \log n)$

Best Case: Let the best case value be $T_B(n)$.
$T_B(n) = O(n \log n)$

Worst Case: Let the worst case value be $T_W(n)$. If all the elements are sorted, then worst case occurs, $T_w(n) = O(n^2)$

Heap Sort

Heap Sort is one of the best sorting methods being in-place and with O(nlogn) as worst-case complexity.
Heap sort algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Heap sort is an in -place sorting algorithm: only a constant number of array elements are stored outside the input array at any time. `
Worst case running time of Heap sort is O(nlogn).

Heap
Heap is a special tree-based data structure that satisfies the following special heap properties:

1. **Shape Property:** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

2. **Heap Property:** All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their children, heap is called a **Max-Heap**, and if the parent nodes are smalled than their child nodes, heap is called **Min-Heap**.

Algorithm

1. Build a max heap from the input data.

2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

Heap Sort Working and Psuedocode

Initially on receiving an unsorted list,

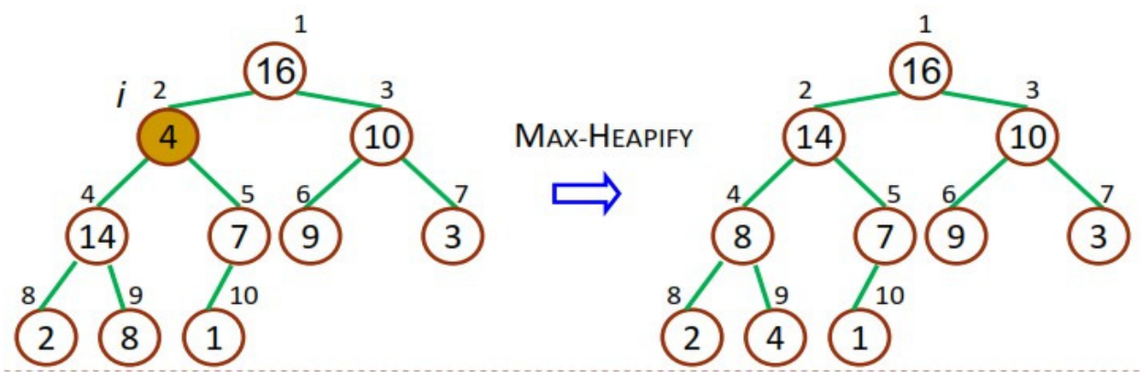
1. First step in heap sort is to build Max-Heap.
2. Repeat Second, Third and Fourth steps, until we have the complete sorted list in our array.
3. Second step- Once heap is built, the first element of the Heap is largest, so we exchange first and last element of a heap.
4. Third step- We delete and put last element(largest) of the heap in our array.
5. Fourth-Then we again make heap using the remaining elements, to again get the largest element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

Basic Procedures

1. The **MAX-HEAPIFY** Procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
2. The **BUILD-MAX-HEAP** procedure, which runs in $O(n)$ time, produces a max-heap from an unordered input array.
3. The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.

The MAX-HEAPIFY procedure

MAX-HEAPIFY is an important subroutine for manipulating max heaps.
Input: an array A and an index *i*
Output: the subtree rooted at index *i* becomes a max heap **Assume**:
the binary trees rooted at LEFT(*i*) and RIGHT(*i*) are max- heaps, but
A[i] may be smaller than its children
Method: let the value at A[i] "float down" in the max-heap



The MAX-HEAPIFY pseudocode

```
Procedure MAX-HEAPIFY(A, i)
{
    l =LEFT(i)
    r =RIGHT(i)
     if l <=heap-size[A] and A[l] > A[i]
     then largest =l
     else largest =i
     if r<=heap-size[A] and a[r] > A[largest]
     then largest =r
     if largest ≠ i
     then exchange[ A[i], A[largest] ] MAX-
     HEAPIFY (A, largest)
}
```

An example of MAX-HEAPIFY procedure

Building a Heap

We can use the MAX-HEAPIFY procedure to convert an array A= [1...n] into a max-heap in a **bottom-up** manner.

The elements in the subarray A [(⌊*n*/2⌋+1)…*n]* are all leaves of the tree, and so each is a 1-element heap.

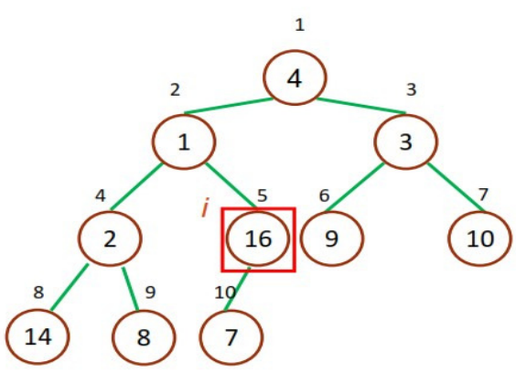
The procedure BUILD-MAX-HEAP goes through the remaining nodes of the tree and runs MAX-HEAPIFY on each one.

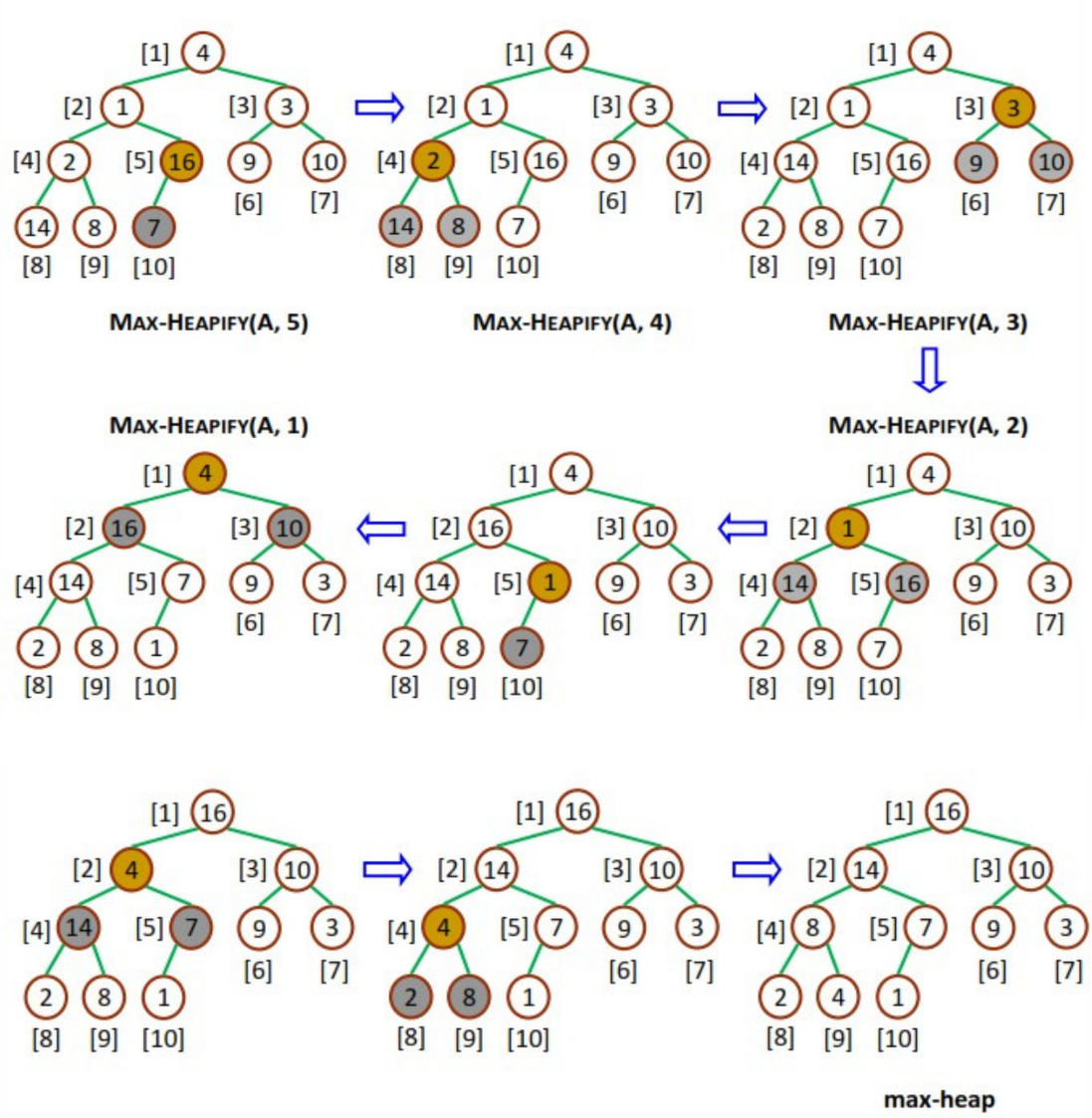
BUILD_MAX_HEAP Psuedocode

```
Procedure BUILD-MAX-HEAP(A)
{
    heap-size[A] =length[A]
    for i = ⌊length[A]/2⌋ downto 1
    do MAX-HEAPIFY(A,i)
}
```

An example

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7





The heap sort algorithm

Since the maximum element of the array is stored at the root, A[1] we can **exchange** it with $A[n]$.

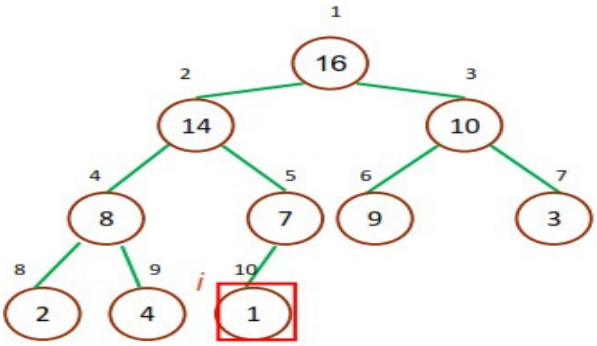
If we now "**discard**" $A[n]$, we observe that $A[1...(n-1)]$ can easily be made into a max-heap.

The children of the root $A[1]$ remain maxheaps, but the new root $A[1]$ element may vio late the max-heap property, so we need to **readjust** the maxheap. That is to call MAX-HEAPIFY(A, 1).

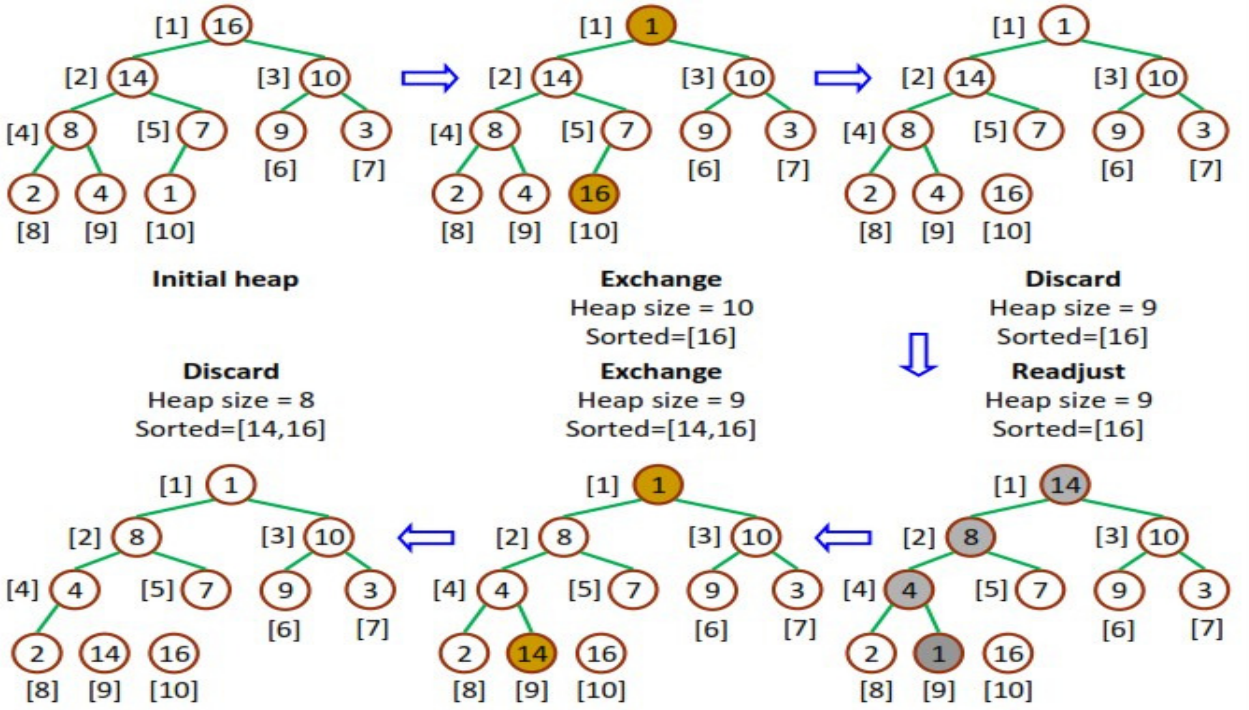
HeapSort Psuedocode

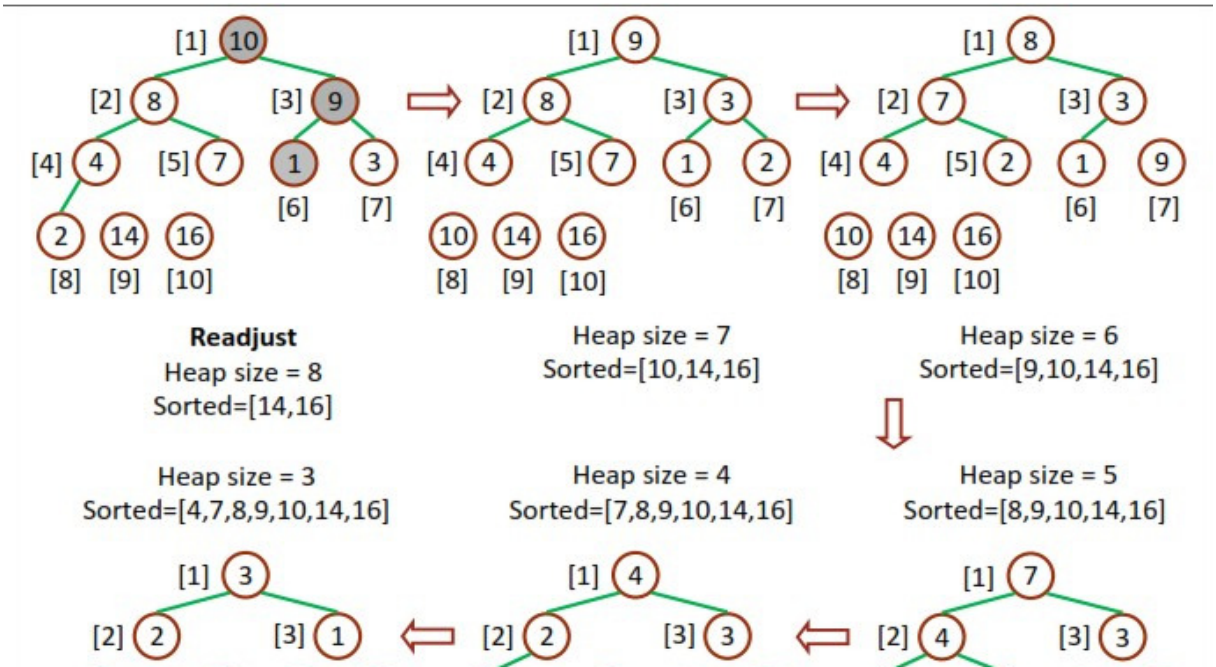
```
Procedure HEAPSORT(A)
{
    BUILD-MAX-HEAP(A)
    for i =length[A] downto 2 do
        exchange (A[1], A[i])
        heap-size[A] =heap-size[A] -1
        MAX-HEAPIFY(A, 1)
}
```

An example



	1	2	3	4	5	6	7	8	9	10
A	1	2	3	4	7	8	9	10	14	16





Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.

First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array.

The array with n elements is sorted by using n-1 pass of selection sort algorithm.

- In 1st pass, smallest element of the array is to be found along with its index **pos**. then, swap A[0] and A[pos]. Thus A[0] is sorted, we now have n -1 elements which are to be sorted.
- In 2nd pas, position pos of the smallest element present in the sub-array A[n-1] is found. Then, swap, A[1] and A[pos]. Thus A[0] and A[1] are sorted, we now left with n-2 unsorted elements.
- In n-1th pass, position pos of the smaller element between A[n-1] and A[n-2] is to be found. Then, swap, A[pos] and A[n-1].

Therefore, by following the above explained process, the elements A[0], A[1], A[2],...., A[n-1] are sorted.

Example

Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

A = {10, 2, 3, 90, 43, 56}.

Pass	Pos	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
1	1	2	10	3	90	43	56
2	2	2	3	10	90	43	56
3	2	2	3	10	90	43	56
4	4	2	3	10	43	90	56
5	5	**2**	**3**	**10**	**43**	**56**	**90**

Sorted A = {2, 3, 10, 43, 56, 90}

Complexity

Complexity	Best Case	Average Case	Worst Case
Time	Ω(n)	θ(n^2)	o(n^2)
Space			o(1)

Algorithm

SELECTION SORT (ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1 to N-1

Step 2: CALL SMALLEST(ARR, K, N, POS)

Step 3: SWAP A[K] with ARR[POS]
[END OF LOOP]

Step 4: EXIT

```
#include<stdio.h>
int smallest(int[],int,int);
void main ()
{
   int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
   int i,j,k,pos,temp;
   for(i=0;i<10;i++)
   {
      pos = smallest(a,10,i);
      temp = a[i];
      a[i]=a[pos];
      a[pos] = temp;
   }
   printf("\nprinting sorted elements...\n");
   for(i=0;i<10;i++)
   {
      printf("%d\n",a[i]);
   }
}
int smallest(int a[], int n, int i)
{
   int small,pos,j;
   small = a[i];
   pos = i;
   for(j=i+1;j<10;j++)
   {
      if(a[j]<small)
      {
         small = a[j];
         pos=j;
      }
   }
   return pos;
```

}

Bubble Sort:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

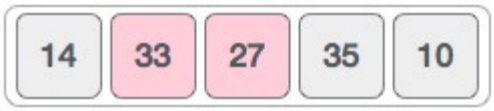
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –

Next we compare 33 and 35. We find that both are in already sorted positions.

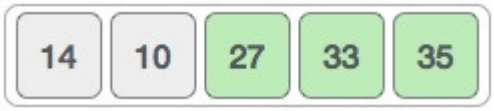
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

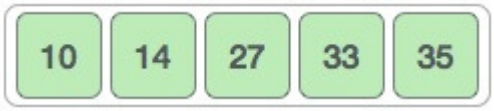
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap**function swaps the values of the given array elements.

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```