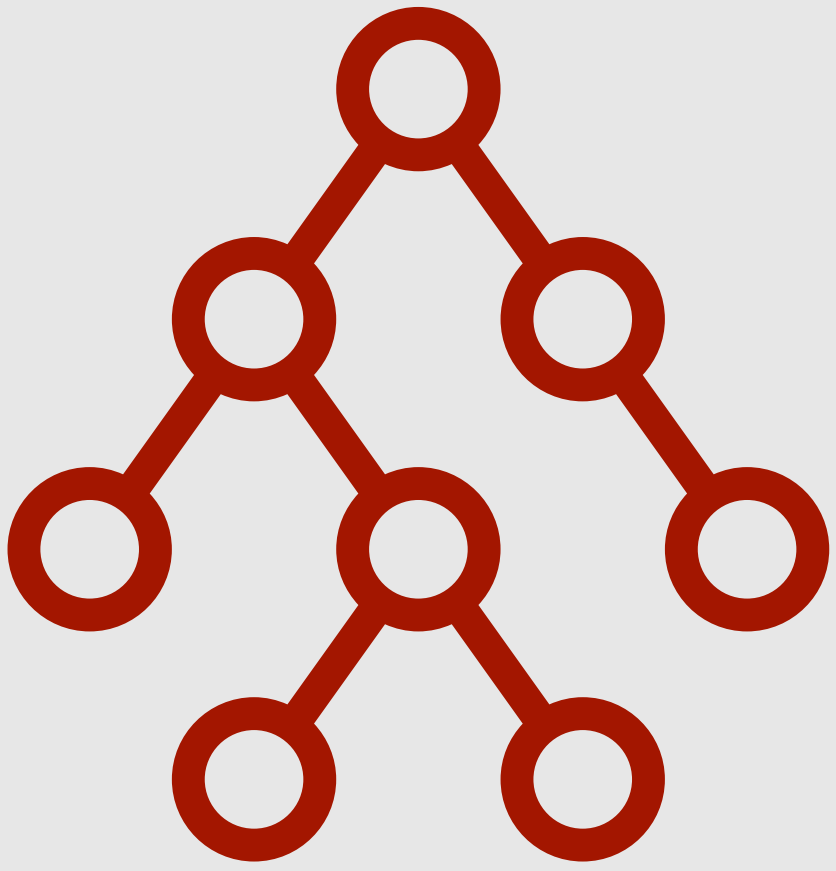
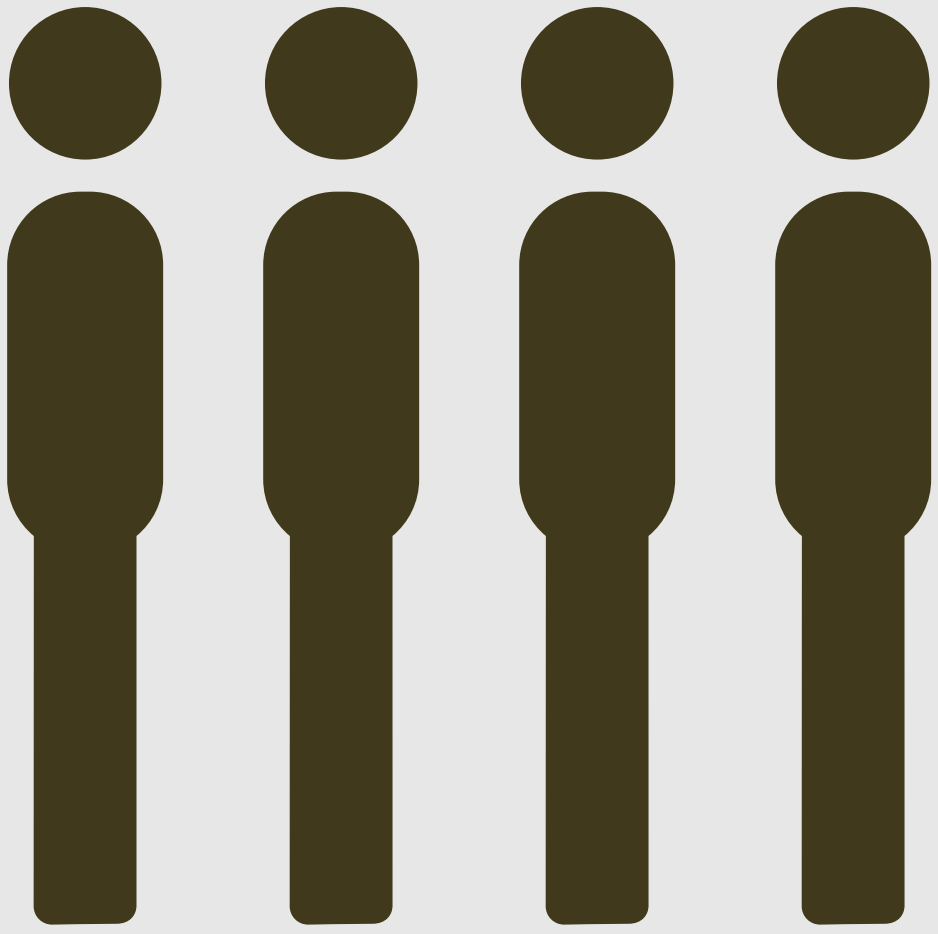
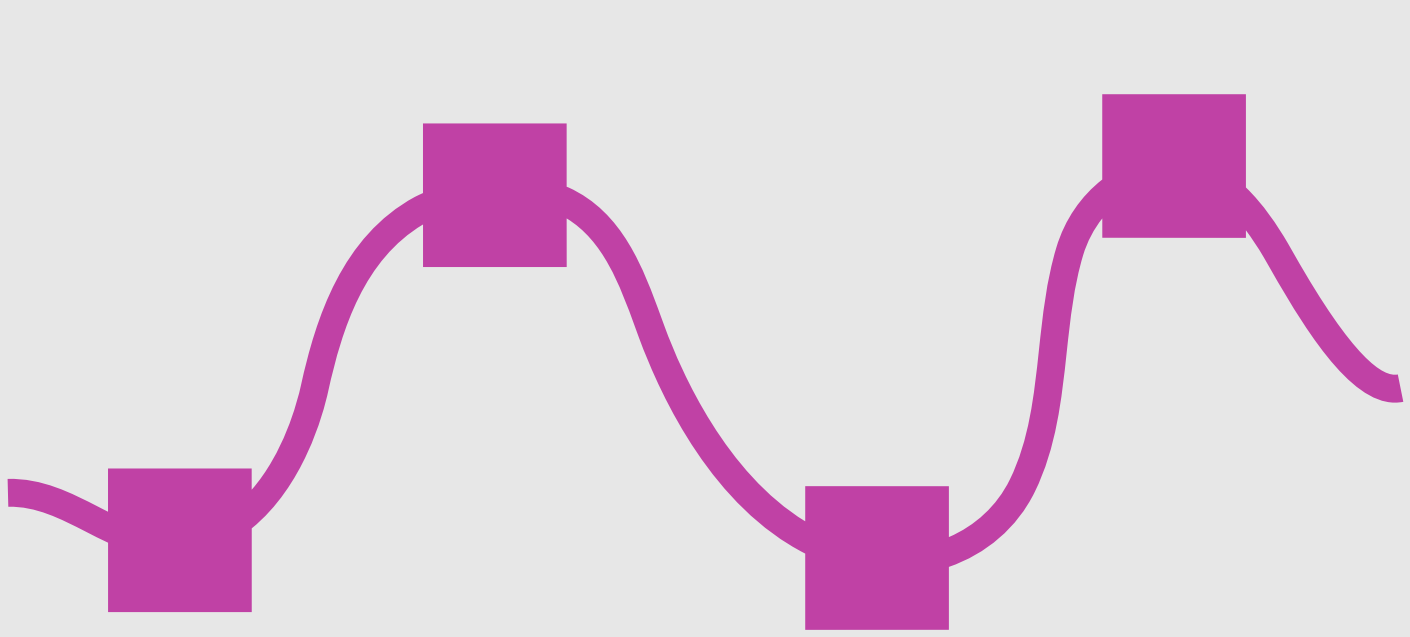
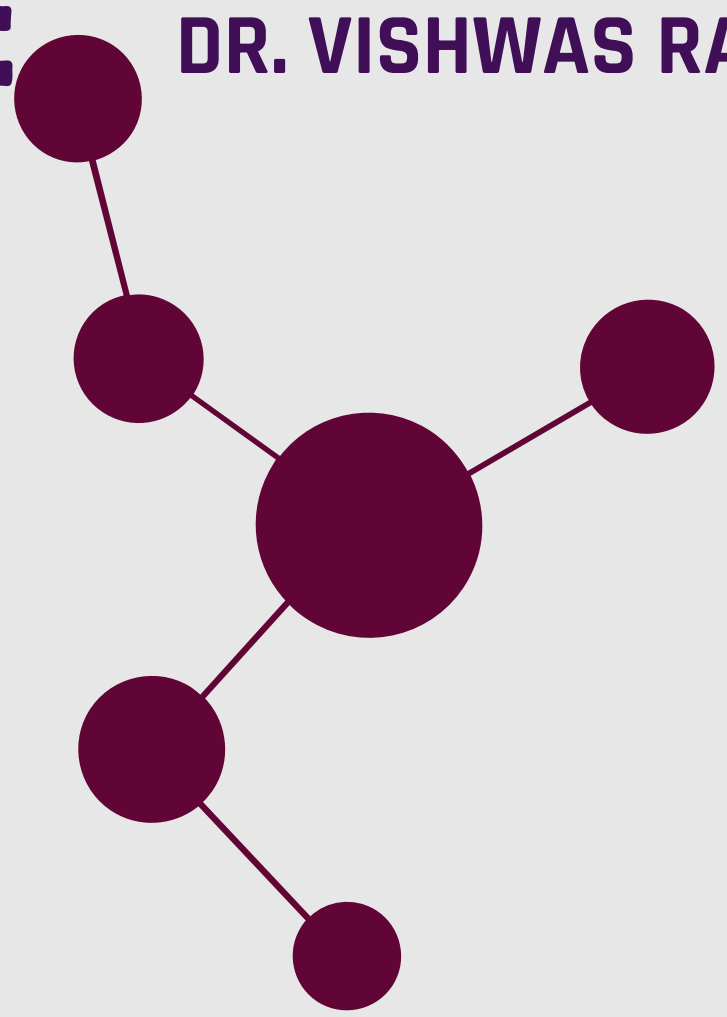


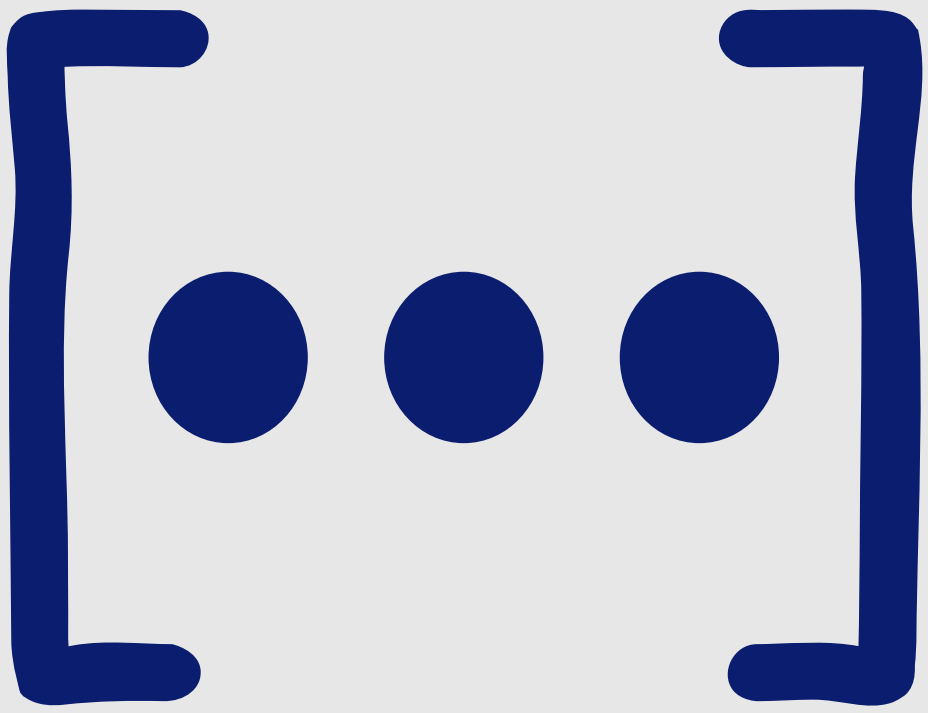




DATA STRUCTURE HANDBOOK

DR. VISHWAS RAVAL





978-93-5906-359-1

Data Structure Handbook

Dr. Vishwas Raval
Assistant Professor, CSE
Faculty of Technology & Engineering
The M S University of Baroda

Data Structure Handbook
By - Dr. Vishwas Raval

Publisher:
Dr. Vishwas Raval

Address:
Assistant Professor, CSE
Faculty of Technology & Engineering,
The M S University of Baroda

ISBN : 978-93-5906-359-1

First Edition

Price: Free

1. INTRODUCTION TO DATA STRUCTURES & ARRAYS

Data structures, is one of the most important subject, which is required by all the software programmers. As a programmer, we would be handling the data in huge quantity. The data we are given requires to be stored in the memory. The memory should be used efficiently. The data will be handled on the basis of its type. It could be an integer, real, character etc. There could be even the combination of the data, resulting in some new type, like structures, unions etc.

1.1 DATA OBJECTS & DATA STRUCTURES

DATA OBJECT is a set of elements, say D. This D may or may not be finite, for example, when data object is set of real numbers is infinite and when it is a set of numeric digits it is finite.

DATA STRUCTURE contains the data object along with the set of operations, which will be performed on them or data structure contains the information about the manner in which these data objects are related. The data structures deal with the study of how data is organized in the memory, how efficiently it can be retrieved and manipulated.

They can be classified into

1. Primitive Data structures.
2. Non Primitive Data structures.

1.1.1 Primitive Data structures

These are the data structures that can be manipulated directly by machine instructions. The integer, real, character etc., are some of primitive data structures. In C, the different primitive data structures are int, float, char and double.

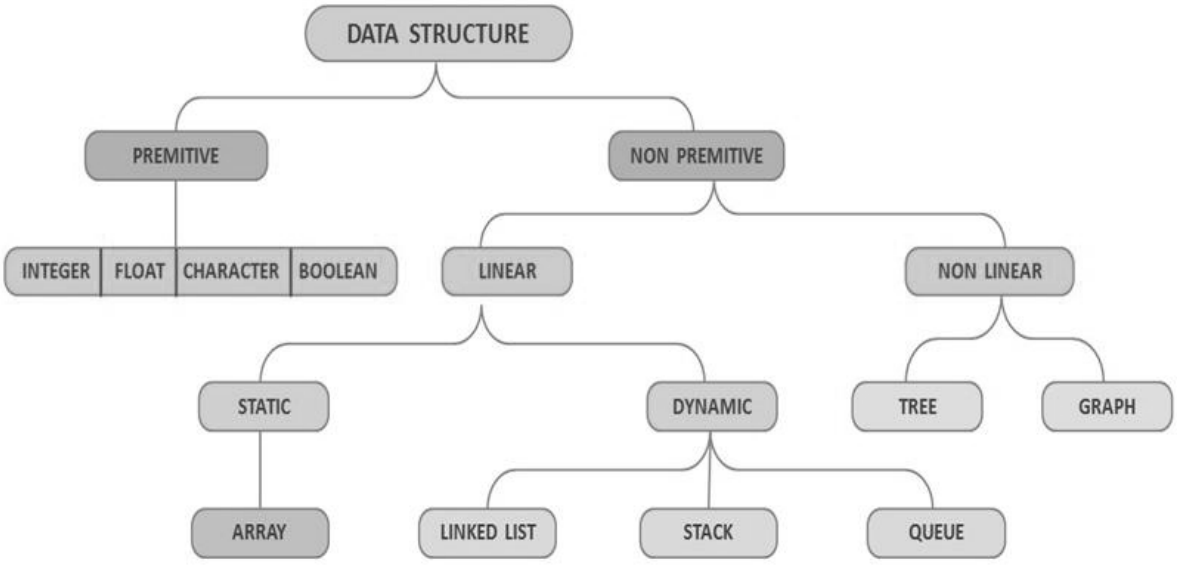
1.1.2 Non Primitive Data structures

These data structures cannot be manipulated directly by machine instructions. Arrays, linked list, trees etc., are some non-primitive data structures. These data structures can be further classified into ‘linear’ and ‘non-linear’ data structures.

The data structures that show the relationship of logical adjacency between the elements are called linear data structures. Otherwise they are called non-linear data structures.

There are two types of non primitive data structures: Linear and non-linear.

Different linear data structures are stack, queue, linear linked lists such as singly linked list, doubly linked list etc. Trees, graphs etc are non-linear data structures. Following figure shows the classification of data structures:



1.1.3 Implementation of The Data Structures

When we define the data structure, we also give the functions or the rules used for handling of the data and its logical and/or physical relation. This can be considered as conceptual handling of data for effective working. But very often we face limitations of a particular language or by the available data i.e. its type and size. When these come into picture along with the defined data structure, we may chose some other available form for handling the data along with all the restrictions imposed on it.

Consider the common example of the QUEUE. It will be every day experience that whenever we are in queue – we follow all the rules of the queue. We are aware that the first person in the queue will be the first one to leave it. We will not allow anyone to enter the queue in between the first and the last person. You cannot leave the queue and cannot search for someone.

When we think of processing the data the very first thing that comes to our mind is that, we should process the data in same sequence in which it arrives and hence for storing the data we define the data structure called QUEUE.

The rules to be followed by the queue are:

1. The data can be removed from one end, called as the front.
2. The new data should be always added at the other end called rear.
3. It is possible that there is no data in the queue, which indicates queue empty condition.
4. It is possible that there is no space in the queue for data to be stored which indicates queue full condition.

Now if we think of actually using the concept in our program then it is necessary to store

these data items. We should remember which the first is and which the last data item is. If we use different variable names for each item they will not look as if they are related.

The only method to store related items, which we all know by this time, is using an Array. The array can be used to implement queue.

In case of arrays, deletion and addition can be made at any position. For queues, we have to impose some restrictions. We will have to remember two positions indicating the first and the last positions of the queue. Whenever an item is removed the first position will change to its next.

If our first position is beyond the last, the queue will not contain any data items. The deletion of an element from the queue will be logical deletion. If we observe the array, then all the elements are physically available all the time.

The same data structure can also be implemented by another data structure known as LINKED LIST. In short we say that implementing data structure d1 using another data structure d2, is mapping from d1 to d2.

1.1.4 Abstract Data Type

A data structure is a set of domains Δ, a designated domain P, a set of functions Φ and set of axioms Γ. A triplet (Δ, Φ, Γ) denotes the data structure d.

The triplet is referred to as abstract data type (ADT). It is abstract because the axioms in the triple do not give any idea about the representation. Defining the data structure is a continuous process because at the initial stage we can design a data structure, and also we indicate as what it should do. In the later stages of the refinement we try to find the ways in which it can be done or how it can be achieved. Thus it is the total process of specification and implementation.

The idea for representing of data, relation in the data objects and the tasks to be performed will be the specification of the data structure. When we actually try to use all the concepts then we decide as how to achieve each goal, which set by each function. This will be the implementation phase.

1.1.5 Algorithm and Pseudo Code

Whenever we need to solve a problem it is a better approach to first write down the solution in algorithm or pseudo codes. Once the logic and data structures to be used are decided we can write algorithm or a pseudo code. Later we can implement them into a program of a particular language.

Algorithm is a set by step solution to a problem written in English alone or with some programming language constructs.

Pseudo code is algorithm written in a particular programming language that will be used to implement the algorithm.

1.1.6 Complexity of Algorithms

When a program is written, it is evaluated on many criteria, like satisfactory results,

minimum code, optimum logic etc. The complexity of the algorithm, which is used, will depend on the number of statements executed. Though the execution for each statement is different, we can roughly check as how many times each statement is executed. Whenever we execute a conditional statement, we will be skipping some statements or we might repeat some statements. Hence the total number of statements executed will depend on conditional statements.

At this stage we can roughly estimate that the complexity is the number of times the condition is executed.

e.g.

```
for(i=0; i<n; i++)
{
        printf("%d",i);
}
```

The output is from 0 to n-1. We can easily say that it has been executed n times. The condition which is checked here is i<n. It is executed n+1 times - n times when it is true and once when it is false. Hence the total number of statements executes is 2n+1. Also the statement i=0 is executed once and i++ is executed n times.

The total = 1+(n+1)+(n)+(n) = 3n + 2

If we ignore the constants, we say that the complexity is of the order of n. The notation used is big-O. i.e. O(n).

1.2 DATA TYPES

As we know that the data, which will be given to us, should be stored and again referred back. These are done with the help of variables. A particular variable's memory requirement depends on which type it belongs to. The different types in C are integers, float (Real numbers), characters, double, long, short etc. These are the available built in types.

Many a times we may come across many data members of same type that are related. Giving them different variable names and later remembering them is a tedious process. It would be easier for us if we could give a single name as a parent name to refer to all the identifiers of the same type. A particular value is referred using an index, which indicates whether the value is first, second or tenth in that parents name.

We have decided to use the index for reference as the values occupy successive memory locations. We will simply remember one name (starting address) and then can refer to any value, using index. Such a facility is known as ARRAYS.

1.3 ARRAYS

An array can be defined as the collection of the sequential memory locations, which can be referred to by a single name along with a number, known as the index, to access a particular

field or data. When we declare an array, we will have to assign a type as well as size.

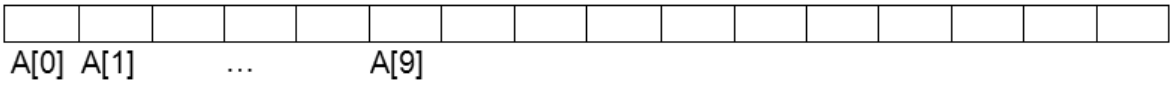
e.g. When we want to store 10 integer values, then we can use the following declaration.

```
int A[10];
```

By this declaration we are declaring A to be an array, which is supposed to contain in all 10 integer values.

When the allocation is done for array a then in all 10 locations of size 2 bytes for each integer i.e. 20 bytes will be allocated and the starting address is stored in A.

When we say A[0] we are referring to the first integer value in A.



Array representation

Hence if we refer to the i^{th} value in array we should write A[i-1]. When we declare the array of SIZE elements, where, SIZE is given by the user, the index value changes from 0 to SIZE-1.

Here it should be remembered that the size of the array is 'always a constant' and not a variable. This is because the fixed amount of memory will be allocated to the array before execution of the program. This method of allocating the memory is known as 'static allocation' of memory.

1.3.1 Handling Arrays

Normally following procedure is followed for programming so that, by changing only one

#define statement we can run the program for arrays of different sizes.

```
#define SIZE 10
int a[SIZE], b[SIZE];
```

Now if we want this program to run for an array of 200 elements we need to change just the

define statement.

1.3.2 Initializing the Arrays.

One method of initializing the array members is by using the 'for' loop. The following for loop initializes 10 elements with the value of their index.

```
# define SIZE 10
main()
{
  int arr[SIZE], i;
```

```
    for(i = 0; i < SIZE ; i++ )
        {
            arr[i] = i;
        }
}
```

An array can also be initialized directly as follows.

```
int arr[3] = {0,1,2};
```

An explicitly initialized array need not specify size but if specified the number of elements provided must not exceed the size. If the size is given and some elements are not explicitly initialized they are set to zero.

e.g.

int arr[] = {0,1,2};

int arr1[5] = {0,1,2}; / Initialized as {0,1,2,0,0}*/*

const char a_arr3[6] = "Daniel"; / ERROR; Daniel has 7 elements 6 in Daniel and a \0*/*

To copy one array to another each element has to be copied using for structure.

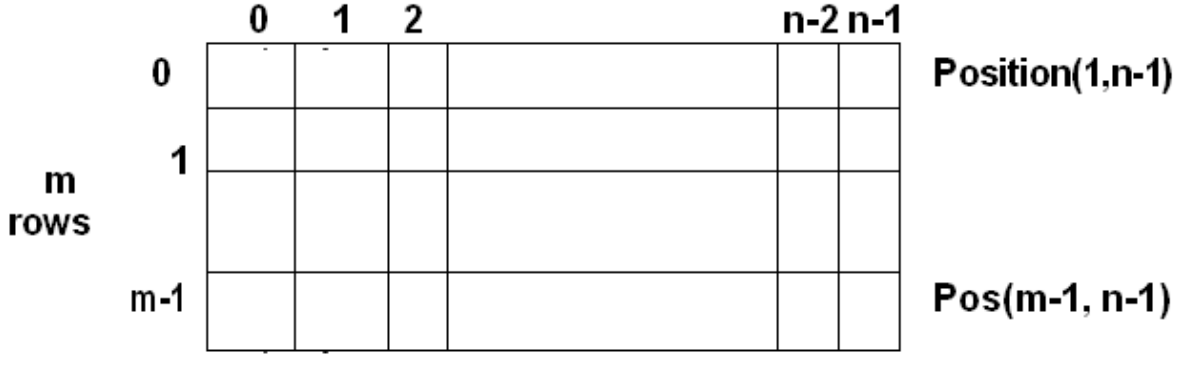
Any expression that evaluates into an integral value can be used as an index into array.

e.g.

arr[get_value()] = somevalue;

1.3.3 Multidimensional Arrays (Two Dimensional Array)

Like we have a linear array with single index, we can also have multidimensional arrays with **n** indexes. At present we will discuss about two-dimensional array, **Matrix.** They are identified with two indices, one is row index and another is the column index. If there are **m** rows and **n** columns in the matrix we say the matrix is of the order (**m** X **n**)**.** The pictorial representation is:



While indicating the position (i,j), i will always be row and j will always be the column number. The names i and j are not important but their position is always very important. We should always put the condition on the number of variables i.e. row number and column number up to the maximum row and column number. This is very essential to avoid undesirable errors. As we have already seen that the address arithmetic row number will start from **0** up to (**m-1**) and column numbers will start from **0** to (**n-1).** Just like one-dimensional arrays, members of matrices can also be initialized in two ways – using 'for' loop and directly. Initialization using nested loops is shown below.

e.g.

```
int arr[10][10];

for(int i = 1;i< = 10;i++) for(int j
  = 1; j< = 10;j++)
        {
          arr[i][j] = i+j;
        }
```

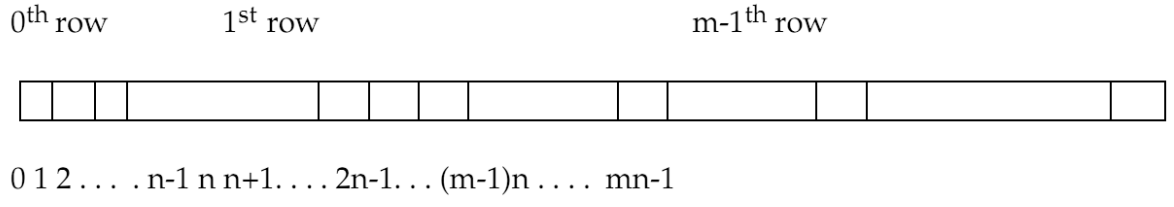
Now let us see how members of matrices are initialized directly.

e.g.

```
int arr[4][3] = {{0,1,2},{3,4,5},{6,7,8},{9,10,11}};
```

The nested brackets are optional.

Let us see how they are stored in the memory. They may be stored row-wise(row major) or column-wise(column major). Most of the languages use row major method. This is required because memory of the computer is always one-dimensional. Therefore for a matrix a linear memory of size (**m** x **n)** will be allocated. Hence in the first n locations we will have 0^{th} row, in the next n location we will have 1^{st} row and so on.



Now we can even find from the position (i,j) , the actual location of in the one-dimensional memory. When we are in the i^{th} row, we know that (i-1) rows are before it and each having **n** memory locations. Therefore (i-1)***n** memory locations should be skipped to reach i^{th} row. It may appear to be confusing as when we refer to i^{th} row, the row number is actually (i-1) and (as our counting begins at 0 rather than 1) hence 1^{st} row will be row number 0 which means that the row number itself will indicate as how many rows are complete prior to it.

e.g. Row number 3 i.e. 0,1,2,3, is actually the 4^{th} row and therefore 3 rows are completed before we reach it. Hence to reach position (i,j), we should skip (i * n) locations of complete rows and j locations for filled locations in that row. Hence the relative address will be (i*n+j).

(0,0)-> (0*n+0) -> 0

(1,4)-> (1*n+4) -> n+4

(5,2)-> (5*n+2) -> 5n+2 … etc.

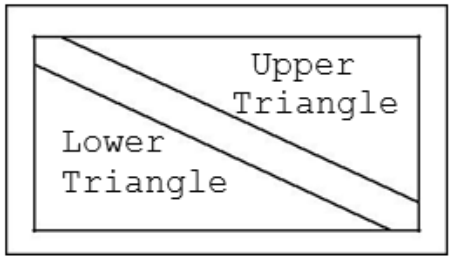
(m-1,n-1)-> ((m-1)*n + (n-1))

-> mn – n + n +1

Now we may also be required to reverse the job, i.e. if we are given a position in one-dimensional memory, we should be able to find row number and column number. If p is the position, then the row number will be (p / n+0) and the column number will be (p % n).

Consider some basic definitions related to matrices.

1. Matrix is said to be **square** if number of rows is equal to the number of columns . i.e. **m=n.**
2. When only the diagonal elements of the square matrix are 1, and every other element is zero, it is called as **Identity matrix.**
3. In a square matrix, position wise the elements are divided as :
   - Diagonal elements
   - Lower triangle, i.e. elements below the diagonal.
   - Upper triangle, i.e. elements above the diagonal.



two dimensional matrix with classifications

4. Reading or referring to the elements row wise. In this we will refer the elements in a particular row one by one, i.e. row number remains same in the process but the column numbers will vary. Similarly we can also refer to the elements column-wise.
5. Transpose of the matrix is changing the row elements to the column elements. i.e. the element in the (i,j) position will occupy (j,i) position in the transpose. The transpose matrix will have the size n x m.

6. Symmetric matrix is a square matrix whose transpose is identical to the original matrix.

Some functions you may use with respect to matrices :

1. Read a matrix row-wise/ column-wise.

   a. When the number of rows and columns are read outside the function .

```
void readmat(int a[][SIZE], int m, int n)
{
  int i,j;
  for(i=0;i<m;i++)
   for(j=0; j<n; j++)
   {
        printf(" Enter elements for a[%d][%d]=",i,j);
        scanf("%d",&a[i][j]);
   }
}
```

   b. When the number of rows and columns are read inside the function. Since these changes have to be reflected back to calling function we use pointer variables as arguments.

```
void readmat(int a[][SIZE], int *m, int *n)
{
  int i,j;
  printf(" Enter the order of the matrix " );
  scanf("%d %d",m,n);
  for(i=0;i< *m;i++) for(j=0;
        j< *n;j++)
   {
        printf(" Enter elements for a[%d][%d]=",i,j);
        scanf("%d",&a[i][j]);
   }
}
```

2. Print the matrix.

```
void printmat(int a[][SIZE], int m, int n)
{
```

```
int i,j;
for(i=0;i<m;i++)
 {
      printf(" Enter elements for a[%d][%d]=",i,j);
      printf("%d",a[i][j]);
 }
}
```

3. Find the sum of the upper triangle , diagonal and lower triangle matrix elements.

```
int lowmat(int a[][SIZE],  int n)
{
 int i,j,sum1=0;
 for(i=0;i<n;i++)
  for(j=0;j<i;j++)
  {
      sum1=sum1+a[i][j];
  }
 return(sum1);
}
```

The above function considers a square matrix of size n . To find the sum of the upper triangle elements just interchange i and j in the statement sum1=sum1+a[i][j] and make it sum1=sum1+a[j][i]. To find the sum of the diagonal elements put a check condition of whether it is equal to j before finding the sum1. These two functions are left to the students.

4. Checking whether a matrix is symmetric or not.

For this, the matrix has to be a square matrix and the element at every position (i,j) must be same as that of the element at the position (j,i).

```
int sym_mat(int a[][SIZE],  int n)
{
      int i,j;
      for(i=0;i<n;i++)
       for(j=0;j<i;j++)
       {
            if(a[i][j] != a[j][i])
```

```
                return(0);
        else
                return(1);
    }
}
```

5. Printing the transpose/ storing the transpose.

   To interchange i^{th} row with j^{th} row, we should swap(exchange) the elements in the columns:

```
void swap_row(int a[][SIZE],int i, int j, int n)
{
  int k,temp;
  for(k=0;k<n;k++)
   {
        temp = a[i][k];
        a[i][k]= a[j][k];
        a[j][k]=temp;
   }
}
```

1.3.4 Sparse Matrix

A matrix in which majority of the elements are zeroes they are known as Sparse matrices. Generally in scientific calculations, a matrix with hundreds of rows and columns may be required. The elements required could be very few and remaining positions are filled with zeroes. For example, in a 10 x 10 matrix, only 15 elements are non-zero and the remaining 85 locations contain zeroes. In such cases we can store the matrix in a different form. The representation we will be using is a one-dimensional array where each data item is required to store three things, Row number, Column number and Value.

Hence, we will use a structure for this case :

```
struct sparse
{
        int row, col, val;
}sp[SIZE];          /* SIZE is declared as required */
```

Here, we are declaring sp as one-dimensional array of structures. Now this array can represent a matrix having any number of rows and columns but the only restriction is that

the number of nonzero elements should not exceed SIZE.

e.g.

matrix A Row/Col 0 1 2 3 4

0	2 0 0 -3 0
1	0 0 11 0 0
2	0 -7 0 0 1
3	–4 0 0 0 9

matrix of order 4 X 5

Sparse Representation:

	0	1	2	3	4	5	6	7
row	4	0	0	1	2	2	3	3
col	5	0	3	2	1	4	0	4
val	7	2	-3	11	-7	1	-4	9

Here the position 0 in the sparse matrix representation actually informs about the total number of rows , total number of columns and number of non-zero elements.

Figure 5 shows as to how a particular matrix has been stored in rows, columns and val form. Here we should preferably take the row number in ascending manner and for each row we must take the columns in ascending manner. This is not a rule but it helps in a number of applications.

The next question is how to take the input from the user. One way is to accept the full matrix and then converting it to the sparse form. The other way is to ask the user to input the row number, col number and val.

Now for the first method it is a simple matrix reading.

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

```
        if(a[i][j] != 0)
        {
                sp[k].row=i;
                sp[k].col=j;
                sp[k].val=a[i][j];
                k++;
        }
 }
        [OR]
ans = 1;
do
 {
         scanf("%d %d%d",&sp[k].row,&sp[k].col,&sp[k].val);
         k++;
         printf(" \n Any more values ? ( 1= Yes / 0=No)");
         scanf("%d",&ans);
 }while(ans==1);
```

Both these operations will work in an identical way and will generate sp array correctly. Now once the matrix is read properly, the next job is to perform operations like addition, multiplication etc.

Algorithm:

1. *To add two sparse matrices, every time we will check whether row numbers are same.*
2. *If not then copy the smaller row value element into the resultant and increment in that array.*
3. *Otherwise check the column number, if they are not same, then again copy the smallest element into the resultant and increment that array.*
4. *When the row numbers and the column numbers are same , we will add the elements and store them into the resultant.*

The advantage of the addition of the two sparse matrices is that irrespective of their size we can add two matrices.

1.3.5 Multiplication of the sparse matrices:

Now let us consider a problem of multiplying two sparse matrices. It probable could be very difficult as we do not even know the size of the matrices. Secondly, the other matrix, to which the first matrix is going to be multiplied should satisfy the condition, i.e. number of columns of the first matrix must be equal to the number of rows of the second matrix. In sparse matrix we can always add a row of zeroes or a column of zeroes so that the size barrier doesn't exist. Another thing is that we should take transpose of the second matrix or arrange it column-wise so that during multiplication, instead of row of the first into column of the second, we may have row-to-row multiplication and the result will be the same.

First we should write a function of taking the transpose of the sparse matrix. If you look at the definition of the transpose, which says that (i,j) element in the original matrix will have the position(j,i) in transpose, we get the idea that we should interchange the row number and the column number. This can be very easily done as given below:

```
for(i=0; i<k; i++)
 {
        temp = sp[i].row;
        sp[i].row = sp[i].col;
        sp[i].col = temp;
 }
```

But then it won't follow the sorted nature of the matrix, i.e. row wise arrangement of elements. hence, we will have to sort sp. Now, the two matrices , sp1 and sp2, are ready for multiplication and the result is to be stored in the matrix sp3.

The pseudo code is given below. You may write the program and implement it.

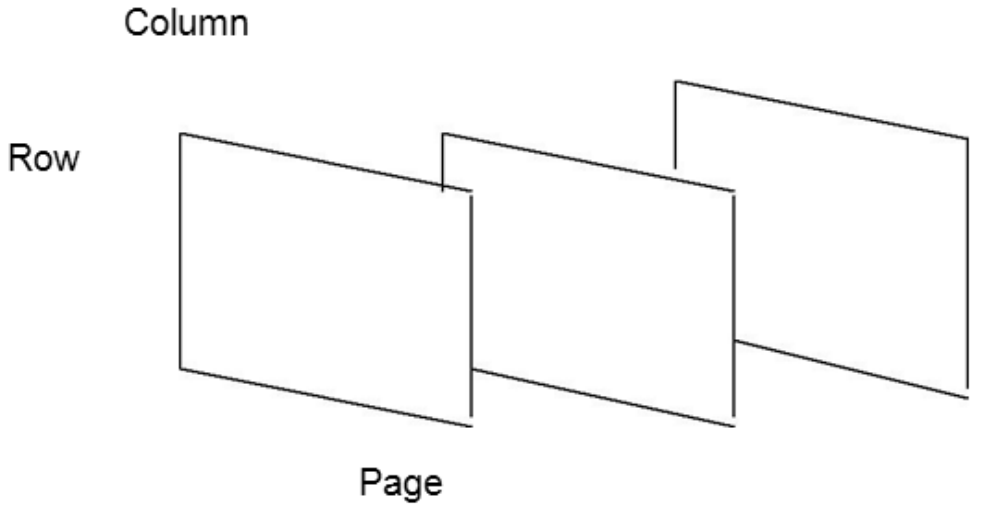
1. sp1 is first matrix and sp2 is the second matrix in the transpose form.
2. Let M1 and M2 be number of elements in the matrices sp1 and sp2.
3. i and j are positions of sp1 and sp2 and k for sp3.
4. i=0, j=0, k=0
5. sp3[k].row = i
   sp3[k].col = j
   sp3[k].val = 0
6. if(sp1[i].col = sp2[j].col)
   {
        sp3[k].val += sp1[i].bval * sp2[j].val;
        i++;

```
            j++;
        }
7.  if(sp1[i].col < sp2[j].col)
            i++;
    else
            j++;
8. if((sp1[i].row = sp3[k].row) && ( sp2[j].row=sp3[k].row))
        goto step 6;
```

Using the above steps, the user is advised to write a program for the multiplication of the sparse matrices.

1.3.6 Multidimensional Arrays:

We can have three dimensional array or even more. The three dimensional arrays will have as usual two indexes for row number and column number and the third index for page number.



Multidimensional array

We normally read data page wise. This will be used only in case of specific applications where all the data under consideration is related to each other by the position specification. By three dimensional or more dimensional array, the physical interpretation is rather difficult to imagine. When we learn about other data structures and after considering the memory requirement, compared to the utilization of the memory in multidimensional arrays, we may choose some other data structure (very effective) related to the specific application.

1.4 AN APPLICATION OF ARRAY: EVALUATION OF POLYNOMIALS

Polynomial is a expression of type :

P(x) = P0 + P1 X^1 + P2 X^2 + ... + Pn X^n.

Where, n is called the degree of that polynomial.

One thing is very clear from the expression, which contain only one variable x, is that we will require storing for each item its coefficient and its power. The following structure would serve the purpose:

```
struct polyno
{
        float coef;
        int pw;
}
```

Then we may proceed with the declaration of an array of structures.

```
struct polyno P[SIZE];
```

Array as a data structure will have many applications and also very important facilities. But size limitations will appear as a setback for this choice. The degree of the polynomial will decide the size of the array.

(size of the polynomial) = (degree of polynomial) + 1

But as we are going to input any polynomial, the size i.e. the degree is not known. Hence let us decide the array size bigger than the normal value polynomial say 10 or 100. We have to keep this limitation in mind when using the program.

Now also observe that the degree of each term is an integer and so is the index of the array. Now we may say that the position of the coefficient in the array itself is the degree of x with it. Then we may not even require the array of structures, but even a float array of **coef** will work.

e.g.

coef[0] is coefficient of x^0
coef[1] is coefficient of x^1
:
coef[i] is coefficient of x^i

The input may be taken in two ways:

1. Ask for coefficient, every time display the power of x i.e.

   Please input the coefficient of [x^0]:

Please input the coefficient of [x^1]:

Here it will be totally the user's responsibility to give the coefficient as zero if a particular power is absent.

2. Ask the user to input the power and coefficient. Here the user should carefully input all values without repeating any power because we will be directly storing the coefficient at the power location. Here user doesn't have to input all the zero coefficients. It is the programmer's responsibility now to initialize the array with zeroes so that if a particular power is not entered then its coefficient is automatically zero.

Now, we can declare the array as:

```
float coef[SIZE];
```

Now if we are given a polynomial having two variables per term, i.e. for example :

$$P(x, y) = 3x^3y - 4x^2y^4 + 6x - 7y^3 + 18xy$$

Here we cannot employ the previous method and so we have to go back for declaring a structure as follows:

```
struct poly2
 {
  float coef;
  int pwx, pwy;
  };
```

The following is the program, which deals with the polynomials in one variable. It is implemented using array of structures. The terms are stored in the sequence given by the user. Hence we have to scan the second polynomial to match the power when we perform addition. Also, we are required to remember whether a particular term is added or not. We can use a flag for this purpose. You may write programs to perform the polynomial operations with two variables.

1.5 POINTERS

The computer memory is a collection of storage cells. These locations are numbered sequentially and are called addresses.

Pointers are addresses of memory location. Any variable, which contains an address is a pointer variable.

Pointer variables store the address of an object, allowing for the indirect manipulation of that object. They are used in creation and management of objects that are dynamically created during program execution.

1.5.1 Advantages and disadvantages of pointers

Pointers are very effective when

- The data in one function can be modified by other function by passing the address.
- Memory has to be allocated while running the program and released back if it is not required thereafter.
- Data can be accessed faster because of direct addressing.

The only disadvantage of pointers is, if not understood and not used properly can introduce bugs in the program.

1.5.2 Declaring and initializing pointers

Pointers are declared using the (*) operator. The general format is:

```
data_type  *ptrname;
```

where type can be any of the basic data type such as integer, float etc., or any of the user-defined data type. Pointer name becomes the pointer of that data type.

e.g.

```
int *iptr; char
*cptr; float
*fptr;
```

The pointer iptr stores the address of an integer. In other words it points to an integer, cptr to a character and fptr to a float value.

Once the pointer variable is declared it can be made to point to a variable with the help of an address (reference) operator (&).

e.g.

```
int num = 1024;
int *iptr;

iptr =  &num;  // iptr points to the variable num.
```

The pointer can hold the value of 0(NULL), indicating that it points to no object at present.

Pointers can never store a non-address value.

e.g.

```
iptr1=ival;   // invalid, ival is not address.
```

A pointer of one type cannot be assigned the address value of the object of another type.

e.g.

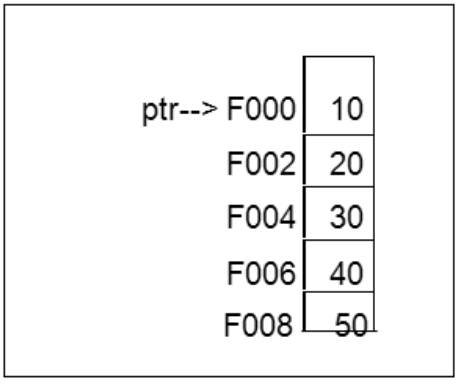
```
double dval,  *dptr = &dval; // allowed

iptr = &dval ;          //not allowed
```

1.5.3 Pointer Arithmetic

The pointer values stored in a pointer variable can be altered using arithmetic operators. You can increment or decrement pointers, you can subtract one pointer from another, you can add or subtract integers to pointers but two pointers cannot be added as it may lead to an address that is not present in the memory. No other arithmetic operations are allowed on pointers than the ones discussed here. Consider a program to demonstrate the pointer arithmetic.

e.g.

```
# include<stdio.h>
main()
{
        int a[]={10,20,30,40,50};
        int *ptr;
        int i;
        ptr=a;
        for(i=0; i<5; i++)
           {
                printf("%d",*ptr++);
           }
 }
```



Output:

```
10 20 30 40 50
```

The addresses of each memory location for the array 'a' are shown starting from F002 to F008. Initial address of F000 is assigned to 'ptr'. Then by incrementing the pointer value next values are obtained. Here each increment statement increments the pointer variable by 2 bytes because the size of the integer is 2 bytes. The size of the various data types is shown below for a 16-bit machine. It may vary from system to system.

char	1 byte
int	2 bytes
float	4 bytes
long int	4 bytes
double	8 bytes
short int	2 bytes

1.5.4 Array of pointers

Consider the declaration shown below:

*char *A[3]={"a", "b", "Text Book"};*

The example declares 'A' as an array of character pointers. Each location in the array points to string of characters of varying length. Here A[0] points to the first character of the first string and A[1] points to the first character of the second string, both of which contain only one character. However, A[2] points to the first character of the third string, which contains 9 characters.

1.5.5 Passing parameters to the functions

The different ways of passing parameters into the function are:

- Pass by value(call by value)
- Pass by address/pointer(call by address)

In pass by value we copy the actual argument into the formal argument declared in the function definition. Therefore any changes made to the formal arguments are not returned back to the calling program.

In pass by address we use pointer variables as arguments. Pointer variables are particularly useful when passing to functions. The changes made in the called functions are reflected back to the calling function. The program uses the classic problem in programming, swapping the values of two variables.

```
void val_swap(int x, int y)            // Call by Value
 {
  int t;
        t = x;
        x = y;
        y = t;
}
void add_swap(int *x, int *y) // Call by Address
 {
  int t;
        t = *x; *x
        = *y; *y
        = t;

}
```

```
void main()
{
 int n1 = 25, n2 = 50;
        printf("\n Before call by Value : ");
        printf("\n n1 = %d n2 = %d",n1,n2);
        val_swap( n1, n2 );
        printf("\n After call by value : ");
        printf("\n n1 = %d n2 = %d",n1,n2);
        printf("\n Before call by Address : ");
        printf("\n n1 = %d n2 = %d",n1,n2);
        val_swap( &n1, &n2 );
        printf("\n After call by value : ");
        printf("\n n1 = %d n2 = %d",n1,n2);
}
```

Output:

Before call by value : n1 = 25 n2 = 50

After call by value : n1 = 25 n2 = 50 // x = 50, y = 25

Before call by address : n1 = 25 n2 = 50

After call by address : n1 = 50 n2 = 25 //x = 50, y = 25

1.5.6 Relation between Pointers and Arrays

Pointers and Arrays are related to each other. All programs written with arrays can also be written with the pointers. Consider the following:

```
int arr[] = {0,1,2,3,4,5,6,7,8,9};
```

To access the value we can write,

arr[0] or *arr;
arr[1] or *(arr+1);

Since '*' is used as pointer operator and is also used to dereference the pointer variables, you have to know the difference between them thoroughly.

*(arr+1) means the address of arr is increased by 1 and then the contents are fetched.
*arr+1 means the contents are fetched from address arr and one is added to the content.

Now we have understood the relation between an array and pointer. The traversal of an array can be made either through subscripting or by direct pointer manipulation.

e.g.

```
void print(int *arr_beg, int *arr_end)
{
        while(arr_beg ! = arr_end)
          {
        printf("%i",*arr);
                ++arr_beg;
        }
}
void main()
{
       int arr[] =
      {0,1,2,3,4,5,6,7,8,9}
      print(arr,arr+9);
}
```

arr_end initializes element past the end of the array so that we can iterate through all the elements of the array. This however works only with pointers to array containing integers.

1.6 SCOPE RULES AND STORAGE CLASSES

Since we explained that the values in formal variables are not reflected back to the calling program, it becomes important to understand the scope and lifetime of the variables.

The storage class determines the life of a variable in terms of its duration or its scope. There are four storage classes:

- automatic
- static
- external
- register

1.6.1 Automatic Variables

Automatic variables are defined within the functions. They lose their value when the function terminates. It can be accessed only in that function. All variables when declared within the function are, by default, 'automatic'. However, we can explicitly declare them by using the keyword 'automatic'.

e.g.

```
void print()
 {
```

```
  auto int i =0;
        printf("\n Value of i before incrementing is %d", i);
        i= i + 10;
        printf("\n Value of i after incrementing is %d", i);
 }
main()
 {
        print();
        print();
        print();
 }
```

Output:

Value of i before incrementing is : 0

Value of i after incrementing is : 10

Value of i before incrementing is : 0

Value of i after incrementing is : 10

Value of i before incrementing is : 0

Value of i after incrementing is : 10

1.6.2 Static Variables

Static variables have the same scope s automatic variables, but, unlike automatic variables, static variables retain their values over number of function calls. The life of a static variable starts, when the first time the function in which it is declared, is executed and it remains in existence, till the program terminates. They are declared with the keyword static.

e.g.

```
void print()
 {
  static int i =0;
        printf("\n Value of i before incrementing is %d", i);
        i= i + 10;
        printf("\n Value of i after incrementing is %d", i);
 }
main()
```

```
{
        print();
        print();
        print();
}
```

Output:

```
Value of i before incrementing is : 0
Value of i after incrementing  is : 10
Value of i before incrementing is : 10
Value of i after incrementing  is : 20
Value of i before incrementing is : 20
Value of i after incrementing  is : 30
```

It can be seen from the above example that the value of the variable is retained when the function is called again. It is allocated memory and is initialized only for the first time.

1.6.3 External Variables

Different functions of the same program can be written in different source files and can be compiled together. The scope of a global variable is not limited to any one function, but is extended to all the functions that are defined after it is declared. However, the scope of a global variable is limited to only those functions, which are in the same file scope. If we want to use a variable defined in another file, we can use extern to declare them.

e.g.

```
// FILE 1 – g is global and can be used only in main() and // // fn1();
int g = 0;
void main()
{
        :
}
void fn1()
{
        :
}
// FILE 2 If the variable declared in file 1 is required to be used in file 2 then it is to
```

be declared as an extern.

```
extern int g = 0;
void fn2()
{
        :
}
void fn3()
{
        :
}
```

1.6.4 Register Variable

Computers have internal registers, which are used to store data temporarily, before any operation can be performed. Intermediate results of the calculations are also stored in registers. Operations can be performed on the data stored in registers more quickly than on the data stored in memory. This is because the registers are a part of the processor itself. If a particular variable is used often – for instance, the control variable in a loop, can be assigned a register, rather than a variable. This is done using the keyword register. However, a register is assigned by the compiler only if it is free, otherwise it is taken as automatic. Also, global variables cannot be register variables.

e.g.

```
void loopfn()
{

 register int i;
        for(i=0; i< 100; i++)
        {
                printf("%d", i);
        }
}
```

1.7 DYNAMIC ALLOCATION AND DE-ALLOCATION OF MEMORY

Memory for system defined variables and arrays are allocated at compilation time. The size of these variables cannot be varied during run time. These are called 'static data structures'.

The disadvantage of these data structures is that they require fixed amount of storage. Once the storage is fixed if the program uses small memory out of it remaining locations are wasted. If we try to use more memory than declared overflow occurs.

If there is an unpredictable storage requirement, sequential allocation is not recommended. The process of allocating memory at run time is called 'dynamic allocation'. Here, the required amount of memory can be obtained from free memory called 'Heap', available for the user. This free memory is stored as a list called 'Availability List'. Getting a block of memory and returning it to the availability list, can be done by using functions like:

- malloc()
- calloc()
- free()

1.7.1 Function malloc(size)

This function is defined in the header file <stdlib.h> and <alloc.h>. This function allocates a block of 'size' bytes from the heap or availability list. On success it returns a pointer of type 'void' to the allocated memory. We must typecast it to the type we require like int, float etc. If required space does not exist it returns NULL.

Syntax:

ptr = (data_type*) malloc(size);

where

- ptr is a pointer variable of type data_type.
- data_type can be any of the basic data type, user defined or derived data type.
- size is the number of bytes required.

e.g.

ptr =(int)malloc(sizeof(int)*n);*

allocates memory depending on the value of variable n.

```
# include<stdio.h>
# include<string.h>
# include<alloc.h>
# include<process.h>
main()
{
  char *str;
       if((str=(char*)malloc(10))==NULL) /* allocate memory for string */
```

```
    {
        printf("\n OUT OF MEMORY");
        exit(1);                    /* terminate the program */
    }
    strcpy(str,"Hello");            /* copy hello into str */
    printf("\n str= %s ",str);      /* display str */
    free(str);                      /* free memory */
}
```

In the above program if memory is allocated to the str, a string hello is copied into it. Then str is displayed. When it is no longer needed, the memory occupied by it is released back to the memory heap.

1.7.2 Function calloc(n,size)

This function is defined in the header file <stdlib.h> and <alloc.h>. This function allocates memory from the heap or availability list. If required space does not exist for the new block or n, or size is zero it returns NULL.

Syntax:

ptr = (data_type*) malloc(n,size);

where

- ptr is a pointer variable of type data_type.
- data_type can be any of the basic data type, user defined or derived data type.
- size is the number of bytes required.
- n is the number of blocks to be allocated of size bytes.

and a pointer to the first byte of the allocated region is returned.

e.g.

```
# include<stdio.h>
# include<string.h>
# include<alloc.h>
# include<process.h>
main()
{
  char *str = NULL;
        str=(char*)calloc(10,sizeof(char)); /* allocate memory for string */
```

```
    if(str == NULL);
     {
          printf("\n OUT OF MEMORY");
          exit(1);                    /* terminate the program */
     }
    strcpy(str,"Hello");              /* copy hello into str */
    printf("\n str= %s ",str);        /* display str */
    free(str);                        /* free memory */
}
```

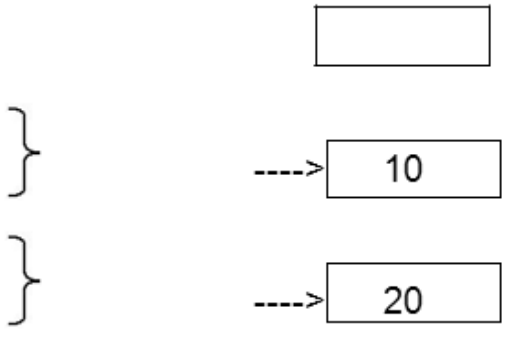
1.7.3 Function free(block)

This function frees allocated block of memory using malloc() or calloc(). The programmer can use this function and de-allocate the memory that is not required any more by the variable. It does not return any value.

1.7.4 Dangling pointer problem.

We can allocate memory to the same variable more than once. The compiler will not raise any error. But it could lead to bugs in the program. We can understand this problem with the following example.

```
# include<stdio.h>
# include<alloc.h>
main()
{
  int *a;

  a= (int*)malloc(sizeof(int));
   *a = 10;

  a= (int*)malloc(sizeof(int));
   *a = 20;

 }
```

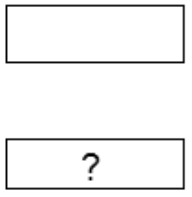


In this program segment memory allocation for variable 'a' is done twice. In this case the variable contains the address of the most recently allocated memory, thereby making the earlier allocated memory inaccessible. So, memory location where the value 10 is stored, is

inaccessible to any of the application and is not possible to free it so that it can be reused.

To see another problem, consider the next program segment:

```
{
  int *a;
  a= (int*)malloc(sizeof(int));
   *a = 10;                                    ---->10
  free(a);
                                               ---->
}
```



Here, if we de-allocate the memory for the variable 'a' using free(a), the memory location pointed by 'a' is returned to the memory pool. Now since pointer 'a' does not contain any valid address we call it as 'Dangling Pointer'. If we want to reuse this pointer we can allocate memory for it again.

1.8 STRUCTURES & UNIONS

1.8.1 Structures

A structure is a derived data type. It is a combination of logically related data items. Unlike arrays, which are a collection of similar data types, structures can contain members of different data type. The data items in the structures generally belong to the same entity, like information of an employee, players etc.

The general format of structure declaration is:

```
struct tag
 {
        type    member1;
        type    member2;
        type member3;
        :
        :
}variables;
```

We can omit the variable declaration in the structure declaration and define it separately as follows:

```
struct tag variable;
```

e.g.

```
struct Account
{
        int accnum; char
        acctype; char
        name[25]; float
        balance;
};
```

We can declare structure variables as :

```
struct Account oldcust;
```

We can refer to the member variables of the structures by using a dot operator (.).

e.g.

```
newcust.balance = 100.0
printf("%s", oldcust.name);
```

We can initialize the members as follows :

e.g.

```
Account customer = {100, 'w', 'David', 6500.00};
```

We cannot copy one structure variable into another. If this has to be done then we have to do member-wise assignment.

We can also have nested structures as shown in the following example:

```
struct  Date
{
        int dd, mm, yy;
};
struct Account
{
        int accnum; char
        acctype; char
        name[25]; float
        balance; struct
        Date d1;
};
```

Now if we have to access the members of date then we have to use the following method.

Account c1;

c1.d1.dd=21;

We can pass and return structures into functions. The whole structure will get copied into formal variable.

We can also have array of structures. If we declare array to account structure it will look like,

Account a[10];

Everything is same as that of a single element except that it requires subscript in order to know which structure we are referring to.

We can also declare pointers to structures and to access member variables we have to use the pointer operator -> instead of a dot operator.

*Account *aptr;*

printf("%s",aptr->name);

A structure can contain pointer to itself as one of the variables, also called self-referential structures.

e.g.

```
struct info
{
        int i, j, k;
        info *next;
};
```

In short we can list the uses of the structure as:

- Related data items of dissimilar data types can be logically grouped under a common name.
- Can be used to pass parameters so as to minimize the number of function arguments.
- When more than one data has to be returned from the function these are useful.
- Makes the program more readable.

1.8.2 Unions

A union is also like a structure, except that only one variable in the union is stored in the allocated memory at a time. It is a collection of mutually exclusive variables, which means all of its member variables share the same physical storage and only one variable is defined at a time. The size of the union is equal to the largest member variables. A union is defined as follows:

```
union tag
{
        type    memvar1;
        type    memvar2;
        type memvar3;
        :
};
```

A union variable of this data type can be declared as follows,

```
union tag variable_name;
```

e.g.

```
union utag
{
        int num;
        char ch;
};
union utag ff;
```

The above union will have two bytes of storage allocated to it. The variable num can be accessed as ff.sum and ch is accessed as ff.ch. At any time, only one of these two variables can be referred to. Any change made to one variable affects another.

Thus unions use memory efficiently by using the same memory to store all the variables, which may be of different types, which exist at mutually exclusive times and are to be used in the program only once.

1.9 ENUMERATED CONSTANTS

Enumerated constants enable the creation of new types and then define variables of these types so that their values are restricted to a set of possible values. There syntax is:

```
enum identifier {c1,c2,...}[var_list];
```

where

- enum is the keyword.
- identifier is the user defined enumerated data type, which can be used to declare the variables in the program.
- {c1,c2,...} are the names of constants and are called enumeration constants.
- var_list is an optional list of variables.

e.g.

enum Colour{RED, BLUE, GREEN, WHITE, BLACK};

Colour is the name of an enumerated data type. It makes RED a symbolic constant with the value 0, BLUE a symbolic constant with the value 1 and so on.

Every enumerated constant has an integer value. If the program doesn't specify otherwise, the first constant will have the value 0, the remaining constants will count up by 1 as compared to their predecessors.

Any of the enumerated constant can be initialised to have a particular value, however, those that are not initialised will count upwards from the value of previous variables.

e.g.

enum Colour{RED = 100, BLUE, GREEN = 500, WHITE, BLACK = 1000};

The values assigned will be RED = 100,BLUE = 101,GREEEN = 500,WHITE = 501,BLACK = 1000

You can define variables of type Colour, but they can hold only one of the enumerated values. In our case RED, BLUE, GREEEN, WHITE, BLACK.

You can declare objects of enum types.

e.g.

enum Days{SUN, MON, TUE, WED, THU, FRI SAT};

Days day;

Day = SUN;

Day = 3; // error int and day are of different types
Day = hello; // hello is not a member of Days.

Even though enum symbolic constants are internally considered to be of type unsigned int we cannot use them for iterations.

e.g.

enum Days{SUN, MON, TUE, WED, THU, FRI, SAT};

for(enum i = SUN; i<SAT; i++) //not allowed.

There is no support for moving backward or forward from one enumerator to another.
However whenever necessary, an enumeration is automatically promoted to arithmetic type.

e.g.

```
if( MON > 0)
{
        printf(" Monday is greater");
}
```

```
int num = 2*MON;
```

1.10 STRINGS

A string is an array of characters. They can contain any ASCII character and are useful in many operations. A character occupies a single byte. Therefore a string of length N characters requires N bytes of memory. Since strings do not use bounding indexes it is important to mark their end. Whenever enter key is pressed by the user the compiler treats it as the end of string. It puts a special character '\0' (NULL) at the end and uses it as the end of the string marker there onwards.

When the function scanf() is used for reading the string, it puts a '\0' character when it receives space. Hence if a string must contain a space in it we should use the function gets().

1.10.1 String Functions

Let us first consider the functions, which are required for general string operations. The string functions are available in the header file "string.h". We can also write these ourselves to understand their working. We can write these functions using

1. Array of Characters
2. Pointers

1.10.2 String Length

The length of the string is the number of characters in the string, which includes spaces, and all ASCII characters. As the array index starts at zero, we can say the position occupied by '\0' indicates the length of that string. Let us write these functions in two different ways mentioned earlier.

Using Arrays

```
int strlen1(char s[])
{
 int i=0;
        while(s[i] != '\0')
                i++;
 return(i);
}
```

Here we increment the positions till we reach the end of the string. The counter contains the size of the string.

Using Pointers

```
int strlen1(char *s)
{
```

```
char *p;
        p=s;

        while(*s != '\0')
                s++;
return(s-p);
};
```

The function is called in the same manner as earlier but in the function we accept the start address in s. This address is copied to p. The variable s is incremented till we get end of string. The difference in the last and first address will be the length of the string.

1.10.3 String Copy : Copy s2 to s1

In this function we have to copy the contents of one string into another string.

Using Arrays

```
void strcopy(char s1[], char s2[])
{
 int i=0;
        while( s2[i] != '\0')
                s1[i] = s2[i++];
        s1[i]='\0';
}
```

Till i^{th} character is not '\0' copy the character s and put a '\0' as the end of the new string.

Using Pointers

```
void  strcopy( char *s1, char *s2)
{
        while( *s2)
        {
                *s1 = *s2;
                s1 ++;
                s2 ++;
         }
        *s1 = *s2;
}
```

1.10.4 String Compare

Using Arrays

```
void strcomp(char s1[], char s2[])
{
  int i=0;
        while( s1[i] != '\0' && s2[i] != '\0')
         {
                if(s1[i] != s2[i])
                        break;
                else
                        i++;
          }
        return( s1[i] – s2[i]);
}
```

The function returns zero , if the two strings are equal. When the first string is less compared to second, it returns a negative value, otherwise a positive value.

1.10.5 Concatenation of S2 to the end of S1

At the end of string one add the string two. Go till the end of the first string. From the next position copy the characters from the second string as long as there are characters in the second string and at the end close it with a '\0' character. This is left as an exercise for the student.

2. STACKS & QUEUES

2.1 INTRODUCTION

While solving a problem we have to represent relation between their data items, for this we use data structures. Its implementation will be our main interest. The functions should be discussed in detail and a proper way of representing the data should be decided. In this session we will mainly discuss two important structures, STACKS & QUEUES.

Stack is a data structure in which the latest data will be processed first. The data is coming in a sequence and we want to decide the sequence in which it must be processed. It is many times necessary that we accept a data item, which in turn depends on some other data item and so we accept that data item. The processing of the earlier data item will be temporarily suspended till we process the new data item completely. This process should be considered very seriously as the data items will have dependence between themselves and it should be preserved.

At one time it will so happen that the processing of the current data item is over and some new data item has come. The decision will be dependent as in which situation current data has been processed. If it has been due to the previous data then the whole processing is suspended and will be continued at the completion of the current data, then we will have to continue with the processing of that item and not the new one. It has been independent then immediately new data item can be processed.

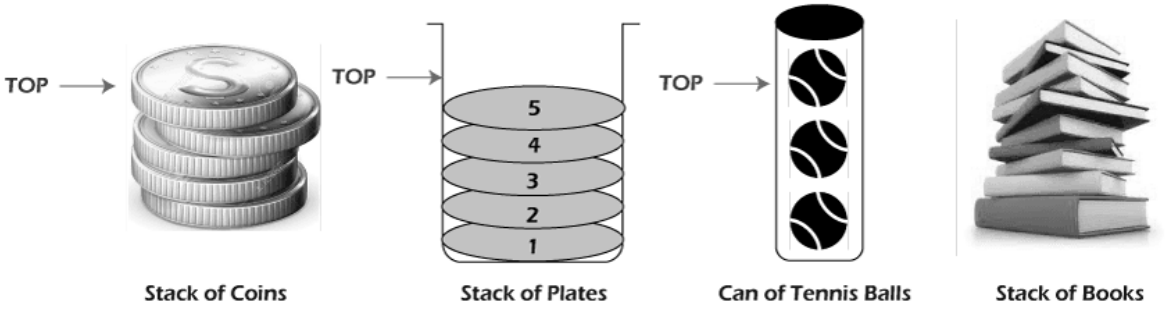
Thus we will have to think about the storage of the suspended data items. In many other cases it so happens that we will be processing the data in a sequence in which it arrived. But normally before the completion of the processing of a particular data, next data arrives. We have to store these arriving data in an order so that after the current processing is over, we can service the arrived data.

2.2 STACKS

The first problem discussed above can be solved, by using a stack. In stacks one data item will be placed on the other as they arrive.

STACK is a special type of data structure where insertion is done from one end called ‘top’ of stack and deletion will be done from the same end.

Here, the last element inserted will be on the top of the stack. Since deletion is done from the same end, last element inserted is the first element to be deleted. So, stack is also called Last In First Out (LIFO) data structure. Following figure shows some real world examples of stack:



As we find that there must be facility to push the items on the stack, or to remove them (pop) from the top of the stack. Also we should have a way by which we will be able to know about the status of the stack as whether it is full or empty.

There are several applications where a stack can be used. For example, recursion, keeping stack of function calls, evaluation of expression etc.

2.2.1 Implementation of Stacks using Arrays

Stacks are one of the most important linear data structures of variable size. This is an important subclass of the lists(arrays) that permit the insertion and deletion from only one end TOP.

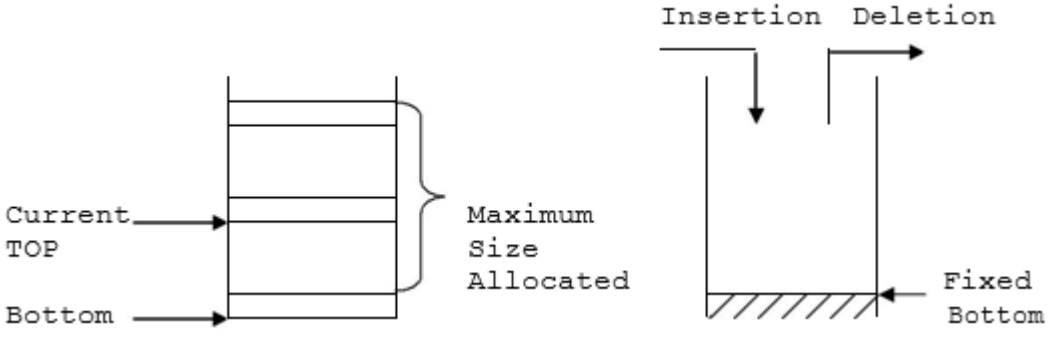
Some terminologies

Push : The Insertion operation is called push

Pop : The operation is Deletion called pop

Top : A pointer, which keeps track of the top element in the Stack. If an array of size N is declared to be a stack, then TOP will be –1 when the stack is empty and is N-1 when the stack is full.

Following is the pictorial representation of stack



Stack Representation

2.2.2 Function To Insert an element into the stack

Before inserting any element into the stack we must check whether the stack is full. In such case we cannot enter the element into the stack. If the stack is not full, we must increment the position of the TOP and insert the element. So, first we will write a function that checks whether the stack is full.

```
#define   SIZE  5

int stack[SIZE];
```

The function returns 1, if, the stack is full. Since we place TOP at –1 to denote stack empty condition, the top varies from 0 to SIZE –1 when it stores elements. Therefore TOP at SIZE-1 denotes that the stack is full.

```
/* function to insert an element into the stack */

void push(element){
        if(Stack is full)
        {
            printf("FULL!!!");
        }
        else
        {
            Top++;
            stack[Top] = element;
        }
}
```

Note that we are treating the TOP as a pointer. This is because the changes must be reflected to the calling function. If we simply pass it by value the changes will not be reflected. Otherwise we have to explicitly return it back to the calling function.

2.2.3 Function to delete an element from the stack

Before deleting any element from the stack we must check whether the stack is empty. In such case we cannot delete the element from the stack. If the stack is not empty, we must delete the element by decrementing the position of the TOP. So, first we will write a function that checks whether the stack is empty.

```
int pop( ){
        if(Stack is Empty)
        {
            printf("EMPTY!!!");
```

```
        return  Top;
    }
    else
    {
        deleted = stack[Top];
        Top--;
        return  deleted;
    }
}
```

This function return 1 if the stack is empty. Since the elements are stored from positions 0 to SIZE-1, the empty condition is considered when the TOP has –1 in it.

Since the TOP points to the current top item, first we store this value in a temporary variable *deleted* and then decrements the TOP. Now return *deleted* to the calling function.

We can also see functions to display the stack, either in the same way as they arrived or the reverse (the way in which they are waiting to be processed).

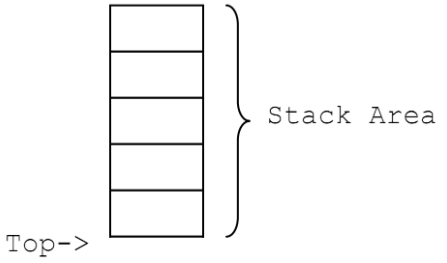
```
void display( ){
if(Stack is Empty)
{
    printf("EMPTY!!!");
}
else
{
    for(i=Top; i>-1; i--)
        printf("%d\n",stack[i]);
}
}
```

We can use these functions in a program and see how they look in the stack.

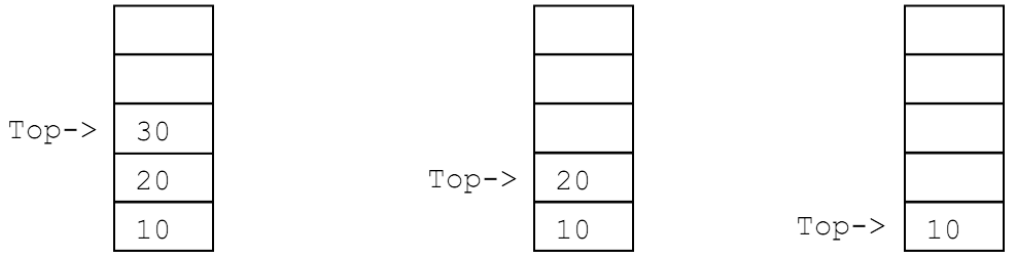
```
# define SIZE 10
main()
{
 int stack[SIZE], val;
 int top = -1,ele;
```

```
        push(10);
        push(20);
        push(30);
        ele = pop();
        printf("%d", ele);
        ele = pop();
        printf("%d", ele);
        push(40);
        ele = pop();
        printf("%d", ele);
}
```

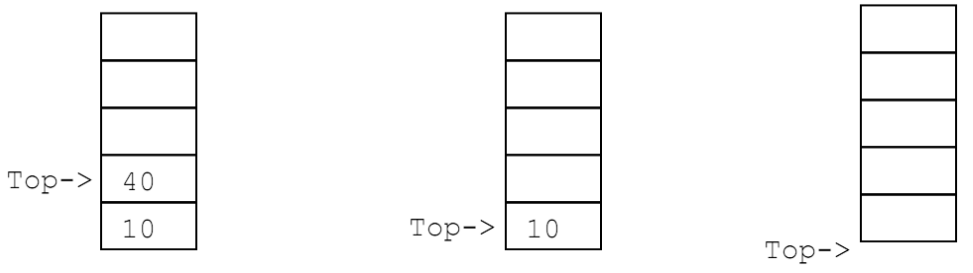
Now we will see the working of the stack with diagrams.



Empty Stack | Push 10 | PUSH 20



PUSH 30 | POP | POP



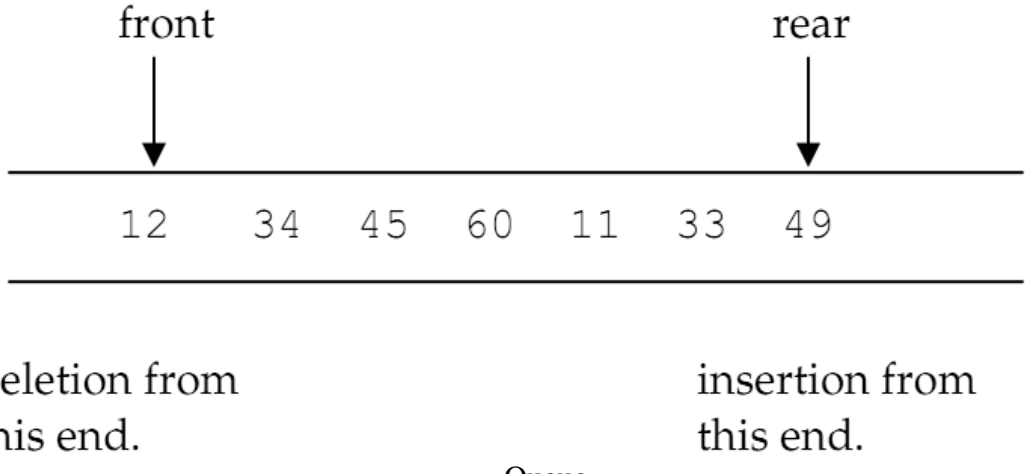
PUSH 40 | POP | POP, STACK EMPTY

2.3 QUEUES

The data structure queue can be considered as the processing by FIRST IN FIRST OUT technique, commonly known as FIFO. As we find there must be facility to put the items at the rear end of the queue, or to remove them from the front end. Also we should have a way by which we will be able to know about the status of the queues as whether it is full or empty. Queues like stacks, also arise quite naturally in the computer solution of many problems. Perhaps the most common occurrence of a queue in computer applications is for the scheduling of the jobs. In batch processing the jobs are "queued-up" as they are read-in and executed, one after another in the order they were received. Following figure shows real world example of queue:

2.3.1 Implementing Queues using Arrays

As mentioned earlier, when we talk of queues we talk about two distinct ends; the front and the rear. Additions to the queue take place at the rear. Deletions are made from the front. So, if the job is submitted for execution, it joins at the rear of the job queue. The job at the front of the queue is the next one to be executed.



Queue

The element, which will be deleted first is 12, addition will take place after 49. We will see some important functions with respect to queues. As we will be using array to represent the queue, the SIZE will be the limitation on the functions. The front and the rear are the two positions of the array, both filled, indicating that the queue exists from front position to rear position. Initially front is 0 and rear is –1.

Function to insert an element into the Queue

Before inserting any element into the queue we must check whether the queue is full. This function returns 1, if the queue is full. Since we place rear at –1 to denote queue empty condition, the rear varies from 0 to SIZE –1 when it stores elements. Therefore rear at SIZE-1 denotes that the queue is full.

```
/* function to insert an element into the queue */
enQueue( int element )
{
        if( rear != size-1)
        {
        rear ++;
        Q[rear] = element;
        }
        else
        printf("Queue is full !!!");
}
```

Function to delete an element from the queue

Before deleting any element from the queue we must check whether the queue is empty. In such case we cannot delete the element from the queue.

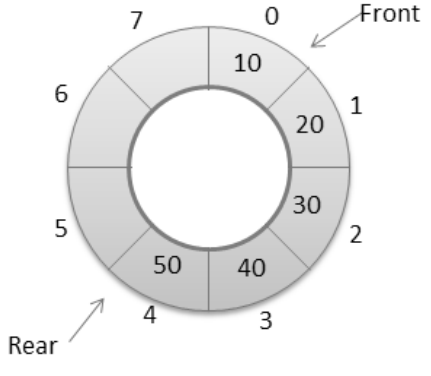
```
deQueue(  )
{
        if( front != rear )
        {
        Q[front++]=-1;
        printf("\nElement removed : %d",Q[front]);
        }
        else
        printf("Queue is empty !!!)";
```

```
}
display(  )
{
        if( front != rear )
        {
        for(int  i=front+1; i<=rear, i++)
              printf("%dt",Q[i]);
        }
        else
        printf("Queue is empty !!!");
}
```

The major problem in the above implementation is that whenever we remove an element from the queue, front will increment and the location used by the element cannot be used again. This problem can be solved if we shift all the elements to the left by one location on every delete operation. This will be very time consuming and is not the effective way of solving the problem.

2.3.2 Circular Queue

The above problem can be solved only when the first position in the array will be logically the next position of the last position of the array. By this way we can say that the array is circular in nature because every position in the array will have logical next position in the array. The queue, which we are going to handle, using this approach is called the circular queue. Remember that it is not the infinite queue but we reuse the empty locations effectively. Now all the functions, which we have written previously will change. We will have a very fundamental function for such case, which will find the logical next position for any given position in the array.



Implementing Circular Queues using Arrays

Insert CircularQueue ( )

1. If (FRONT == 0 && REAR == N-1) or (FRONT == REAR + 1) Then

Print: Queue is Full

Else

If (REAR == -1) Then [Check if QUEUE is empty]

(a) Set FRONT = 0

(b) Set REAR = 0

Else If (REAR == N-1 && FRONT>0) Then [If REAR reaches end of QUEUE]

Set REAR = 0

Else

Set REAR = REAR + 1 [Increment REAR by 1]

[End of Step 4]

2. Set QUEUE[REAR] = ITEM

3. Print: ITEM inserted

4. Exit

Function to delete an element to the circular queue

Delete CircularQueue ( )

1. If (FRONT == -1) Then [Check for Underflow]

Print: Queue is Empty

Else

QUEUE[FRONT]=-1 (Set any specific value indicating NULL)

If (FRONT == REAR) Then [If only element is left]

(a) Set FRONT = -1

(b) Set REAR = -1

Else If (FRONT == N-1) Then [If FRONT reaches end of QUEUE]

Set FRONT = -1

Else

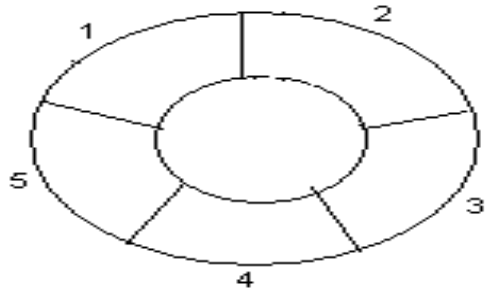
Set FRONT = FRONT + 1 [Increment FRONT by 1]

2. Print: ITEM deleted

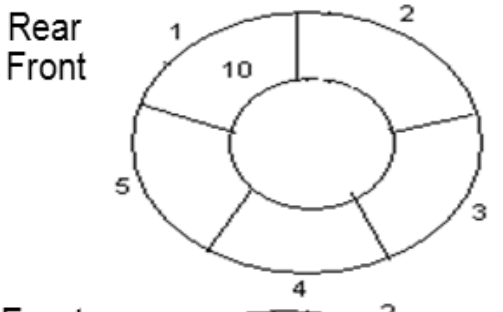
3. Exit

Example 1:

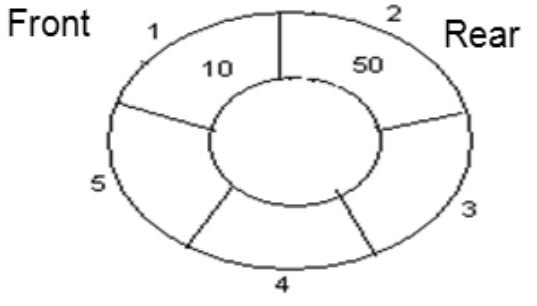
1. Initially, Rear = -1, Front = -1.



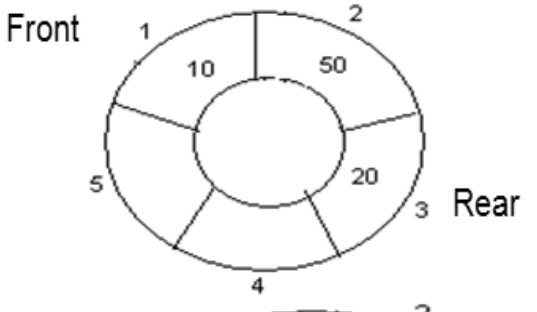
2. Insert 10, Rear = 0, Front = 0.



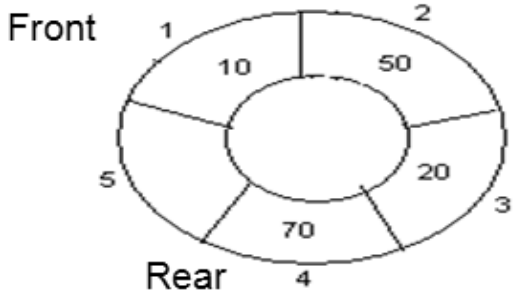
3. Insert 50, Rear = 1, Front = 0.



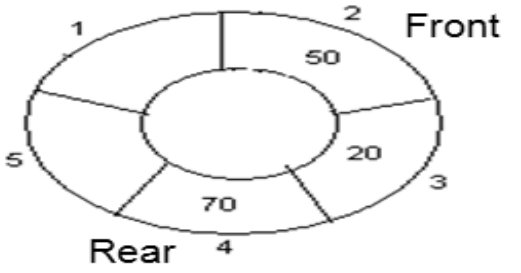
4. Insert 20, Rear = 2, Front = 0.



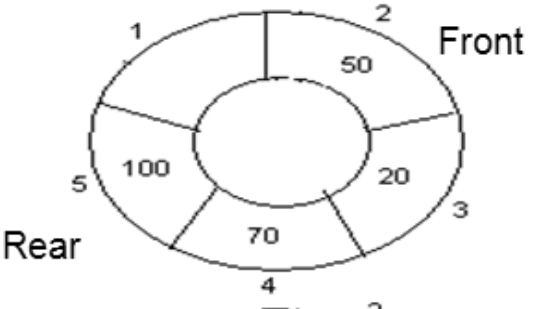
5. Insert 70, Rear = 3, Front = 0.



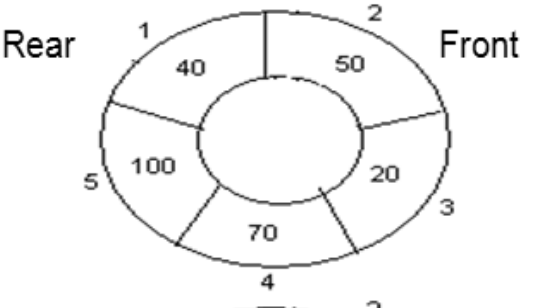
6. Delete front, Rear = 3, Front = 1.



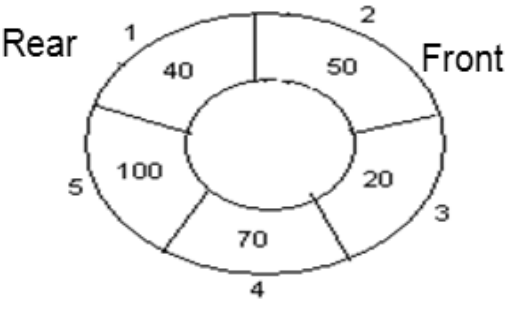
7. Insert 100, Rear = 4, Front = 1.



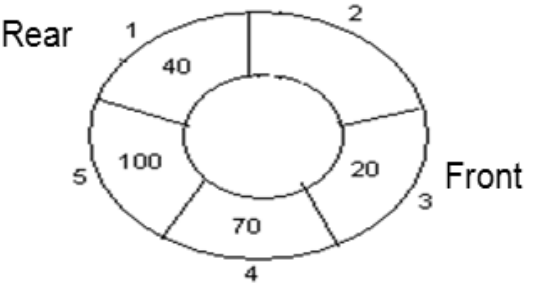
8. Insert 40, Rear = 0, Front = 1.



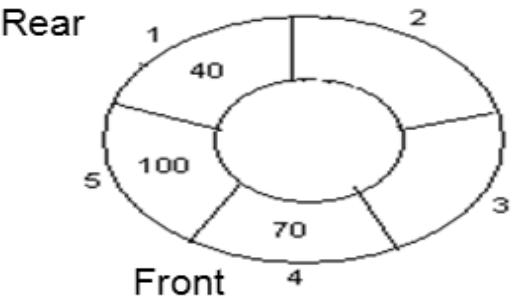
9. Insert 140, Rear = 0, Front = 1. As Front = Rear + 1, so Queue overflow.



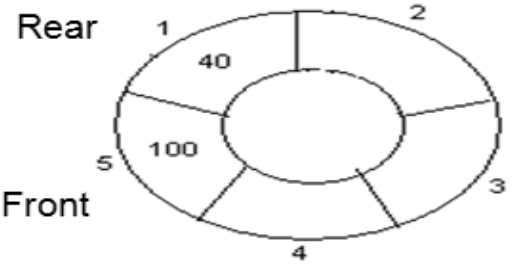
10. Delete front, Rear = 0, Front = 2.



11. Delete front, Rear = 0, Front = 3.



12. Delete front, Rear = 0, Front = 4.



2.3.3 Double Ended Queue (DeQueue)

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.

Algorithm for Insertion at rear end

```
if(rear==MAX)
        Print("Queue is Full");
Else
        if rear=-1
                rear=0;
        else if front=-1
                front=0;
        else
                rear=rear+1;
```

q[rear]=no;`

Algorithm for Insertion at front end

if(front==-1)

Print ("Cannot add item at front end");

else

front=front-1;

q[front]=no;

Algorithm for Deletion at front end

if front==-1

print("Queue is Empty");

else

q[front]=-1;

if front==rear

front=rear=-1;

else

front=front+1;

Algorithm for Deletion at rear end

if rear=-1

print("Cannot delete value at rear end");

else

q[rear]=-1;

if front==rear

front=-1;

rear=-1;

else

rear=rear-1;

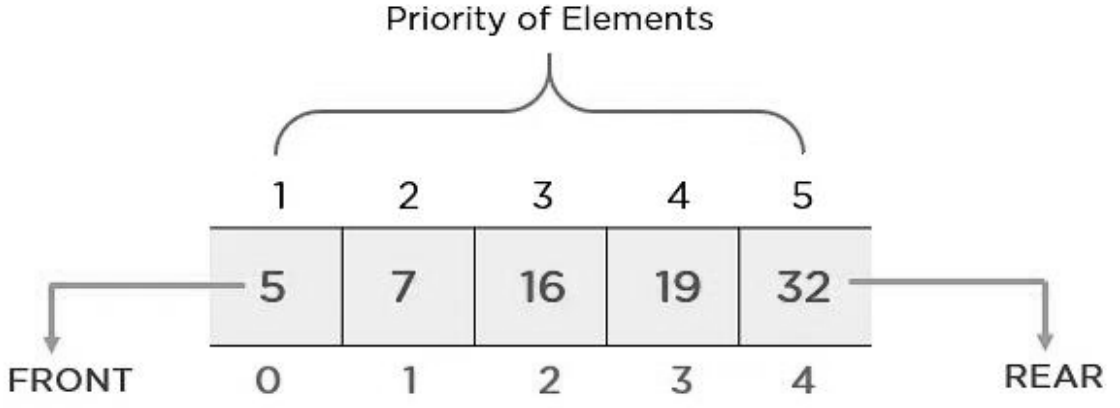
2.3.4 Priority Queue

A priority queue is *least-first-out.* The "smallest" element is the first one removed i.e. the element with highest priority (You could also define a *largest-first-out* priority queue). The definition of priority is up to the programmer. for example, you might define the implementation with value and its relevant priority. If there are several "smallest" elements, the implementer must decide which to remove first

- Remove any “smallest” element (don’t care which)
- Remove the first one added

Priority Queue is an abstract data type that performs operations on data elements per their priority. To understand it better, first analyze the real-life scenario of a priority queue.

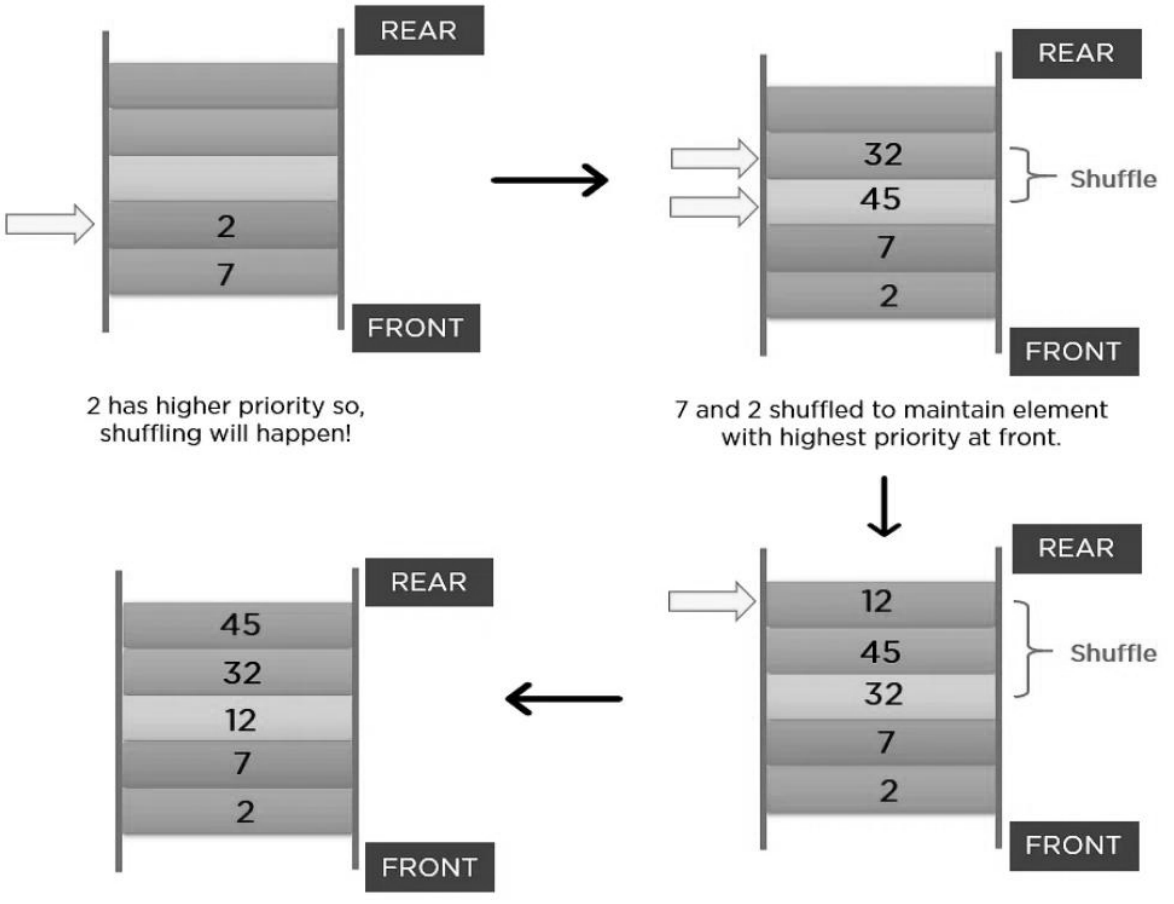
It behaves similar to a linear queue except for the fact that each element has some priority assigned to it. The priority of elements determines the order of removal in a queue, i.e., the element with higher priority will leave the queue first, whereas the element with the lowest priority at last.



Priority queue in a data structure is an extension of a linear queue that possesses the following properties:

- Every element has a certain priority assigned to it.
- Every element of this queue must be comparable.
- It will delete the element with higher priority before the element with lower priority.
- If multiple elements have the same priority, it does their removal from the queue according to the FIFO principle.

Now, understand these properties with the help of an example. Consider you have to insert 7, 2, 45, 32, and 12 in a priority queue. The element with the least value has the highest property. Thus, you should maintain the lowest element at the front node.



2.4 APPLICATIONS OF STACKS

Recursion is a powerful tool in C. It is an important facility that is available in most of the programming languages such as Pascal, C, C++ etc. Now we would understand the meaning of recursion, how a recursive definition for a problem can be obtained, the designing aspects and how they are implemented in C language. Finally we compare the iterative technique with recursion, their merits and demerits.

2.4.1 Recursion

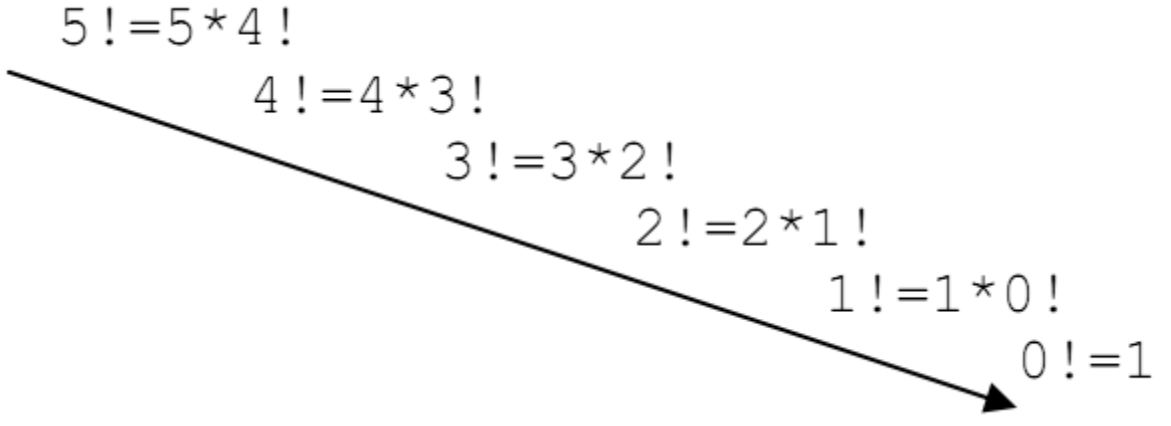
Recursion is a process of expressing a function in terms of itself. A function, which contains a call to the same function or a call to another function (direct recursion), which eventually call the first function (indirect recursion) is also termed as recursion. We can also define recursion as a process in which a function calls itself with reduced input and has a base condition to stop the process. i.e. any recursive function must satisfy two conditions:

1. It must have a terminal condition
2. After each recursive call it should reach a value nearing the terminal condition.

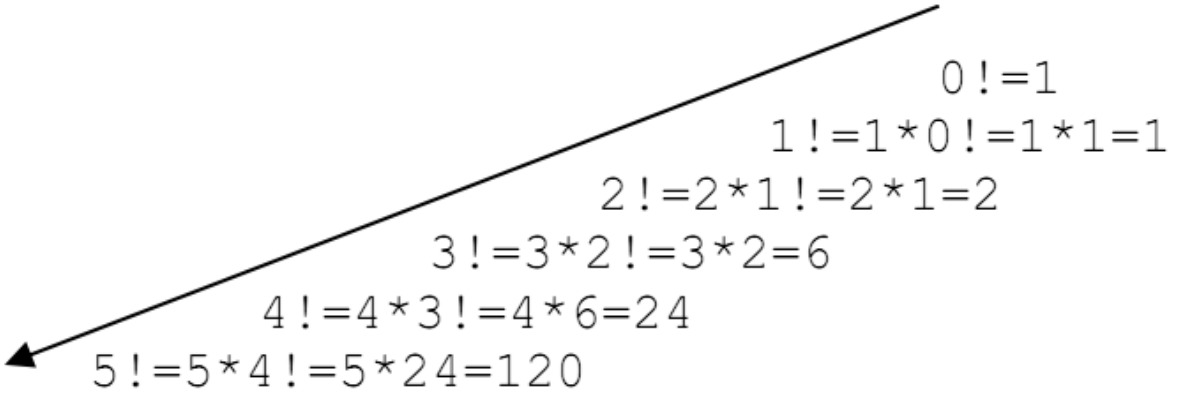
An expression, a language construct or a solution to a problem can be expressed using recursion. We will understand the process with the help of a classic example ; to find the factorial of a number. The recursive definition to obtain factorial of n is shown below

$$\text{Fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{fact}(n-1) & \text{otherwise} \end{cases}$$

We can compute 5! as shown below



By definition 0! is 1. So, 0! Will not be expressed in terms of itself. Now, the computations will be carried out in reverse order as shown.



The C program for finding the factorial of a number is as shown:

```
# include<stdio.h>
int fact(int n)
 {
        if(n==0)
                return 1;
        return n*fact(n-1);
}
main()
{
        int n;
        printf("Enter the number \n");
        scanf("%d",&n);
        printf("The factorial of %d = %d\n",n,fact(n));
}
```

In fact() the terminal condition is fact(0) which is 1. if we don't write terminal condition, the function ends up in calling itself forever, i.e. in an infinite loop. Every time the function is entered in a recursion a separate memory is allocated for the local variables and formal variables. Once the control comes out the function the memory is de-allocated.

When a function is called, the return address, the values of local and formal variables are pushed onto the stack, a block of memory of contiguous locations, set aside for this purpose. After this the control enters into the function. Once the return statement is encountered, control comes back to the previous call, by using the return value present in the stack, and it substitutes the return value to the call. If the function does not return any value, control goes to the statement that follows the function call.

To explain how the factorial program actually works, we will write it using indirect recursion:

```
# include<stdio.h>
int fact(int n)
 {
        int x, y, res;
        if(n==0)
                return 1;
        x=n-1;
        y=fact(x);
```

```
        res= n*y;
        return res;
}
main()
{
        int n;
        printf("Enter the number \n");
        scanf("%d",&n);
        printf("The factorial of %d = %d\n",n,fact(n));
}
```

Suppose we have the statement

A= fact(n);

In the function main(), where the value of n is 4. When the function is called first time, the value of n in the function and the return address say XX00 is pushed on to the stack. Now the value of formal parameter is 4. since we have not reached the base condition of n=0 the value of n is reduced to 3 and the function is called again with 3 as parameter. Now again the new return address say XX20 and parameter 3 are stored into the stack. This process continues till n takes up the value 0, every time pushing the return address and the parameter, Finally control returns after folding back each time from one call to another with the result value 24. The number of times the function is called recursively is called the Depth of recursion. Now let see the working of this program to find the factorial of 4 using the pictorial representation of stack and understand how the recursion takes place in the system.

4	--	--	--	XX00
n	x	y	res	pc

fact(4) in main() function

4	3	--	--	XX20
4	--	--	--	XX00
N	x	y	res	pc

fact(4)

3	2	--	--	XX20
4	3	--	--	XX20
4	--	--	--	XX00
n	x	y	Res	pc

fact(3)

2	1	--	--	XX20
3	2	--	--	XX20
4	3	--	--	XX20
4	--	--	--	XX00
n	x	y	res	pc

fact(2)

1	0	--	--	XX20
2	1	--	--	XX20
3	2	--	--	XX20
4	3	--	--	XX20
4	--	--	--	XX00
n	x	y	Res	pc

fact(1)

n	x	y	res=n*y
1	0	1	1
2	1	1	2
3	2	2	6
4	3	6	24

The trace of stack to find fact(4)

Iteration Vs Recursion

In recursion, every time a function is called, all the local variables , formal variables and return address will be pushed on the stack. So, it occupies more stack and most of the time is spent in pushing and popping. On the other hand, the non-recursive functions execute much faster and are easy to design.

There are many situations where recursion is best suited for solving problems. In such cases this method is more efficient and can be understood easily. If we try to write such functions using iterations we will have to use stacks explicitly.

2.4.2 Evaluation of expressions using stacks

All the arithmetic expressions contain variables or constants, operators and parenthesis. These expressions are normally in the infix form, where the operators separate the operands. Also, there will be rules for the evaluation of the expressions and for assigning the priorities to the operators. The expression after evaluation will result in a single value. We can evaluate an expression using the stacks.

An expression consists of operators and operands. In an expression if the operator, which performs an operation , is written in between the operands it is called an **Infix** expression. If the operator is written before the operands , it is called **Prefix** expression. If the operator is written after the operands, it is called **Postfix** expression.

Consider an infix expression:

2 + 3 * (4 – 6 / 2 + 7) / (2 + 3) – ((4 –1) * (2 – 10 / 2))

When it comes to the evaluation of the expression, following rules are used.

1. brackets should be evaluated first.
2. * and / have equal priority, which is higher than + and -.
3. All operators are left associative, or when it comes to equal operators, the evaluation is from left to right.

In the above case the bracket (4-1) is evaluated first. Then (2-10/2) will be evaluated in

which /, being higher priority, 10/2 will be evaluated first. The above sentence is questionable, because as we move from left to right, the first bracket, which will be evaluated, will be (4-6/2+7).

The evaluation is as follows:-

Step 1: Division has higher priority. Therefore 6/2 will result in 3. The expression now will be (4-3+7).

Step 2: As – and + have same priority, (4-3) will b evaluated first.
Step 3: 1+7 will result in 8.

The total evaluation is as follows.

2 + 3 * (4 – 6 / 2 + 7) / (2 + 3)-((4 – 1) * (2 – 10 /2))

=2 + 3 * 8 / (2 + 3) - ((4 - 1) * (2 – 10 / 2))

=2 + 3 * 8 / 5 -((4 - 1) * (2 – 10 / 2))

=2 + 3 * 8 /5 -(3 * (2 – 10 / 2))

=2 + 3 * 8 /5 -(3 * (2 - 5))

=2 + 3 * 8 / 5 - (3 * (-3))

=2 + 3 * 8 / 5 + 9

=2 + 24 / 5 + 9

=2 + 4.8 + 9 =6.8 + 9 =15.8

2.4.3 Postfix Expressions (Reverse Polish Notations)

In the postfix expression, every operator is preceded by two operands on which it operates. The postfix expression is also known as reverse polish notation. If we want to operate the postfix expression from the given infix, then consider the evaluation sequence and apply it from bottom to top, every time converting infix to postfix.

e7	e7
= e6 + a	= e6 a +
But e6 = e5 - e4	= e5 e4 – a +
But e5 = a - b	= ab – e4 – a +
But e4 = e3 * b	= ab – e3 b * - a +
But e3 = e2 + a	= ab – e2 a + b * -a +
But e2 = c * e1	= ab – ce1 * a + b * -a +

But e1 = d / a = ab – cda / * a + b * -a +

The postfix expression does not require brackets. The above method will not be useful for programming. For programming, we use a stack, which will contain operators and opening brackets. The priorities are assigned using numerical values. Priority of + and – is equal to 1. Priority of * and / is 2. The incoming priority of the opening bracket is highest and the outgoing priority of the closing bracket is lowest. An operator will be pushed into the stack, provided the priority of the stack top operator is less than the current operator. Opening bracket will always be pushed into the stack. Any operator can be pushed on the opening bracket. Whenever operand is received, it will directly be printed. Whenever closing bracket is received, elements will be popped till opening bracket and printed, the execution is as shown.

Algorithm to Convert Infix to Postfix

1. Scan the Infix string from left to right.

2. Initialize an empty stack.

3. If the scanned character is an operand, add it to the Postfix string.

4. If the scanned character is an operator and if the stack is empty push the character to stack.

5. If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack.

6. If top of Stack has higher precedence over the scanned character pop the stack else push the scanned character to stack. Repeat this step until the stack is not empty and top Stack has precedence over the character.

7. Repeat 4 and 5 steps till all the characters are scanned.

8. After all characters are scanned, we have to add any character that the stack may have to the Postfix string.

9. If stack is not empty add top Stack to Postfix string and Pop the stack.

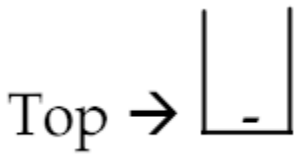
10. Repeat this step as long as stack is not empty.

Example 1:

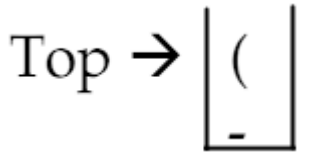
b - (a + b) * ((c - d) / a + a)

1. Symbol is b, hence print b b

2. On -, Stack being empty, push -



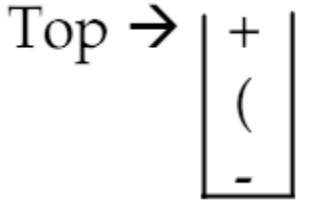
3. On (, push (



4. On a, operand, Hence Print a

b a

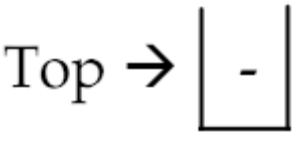
5. On +, Stack being push +



6. On b, operand, Hence Print b

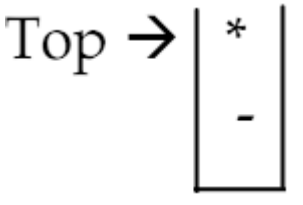
b a b

7. On), pop till (and then print +.
Therefore, pop +, print +, pop (

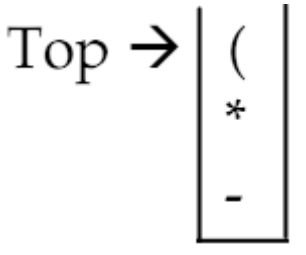


b a b +

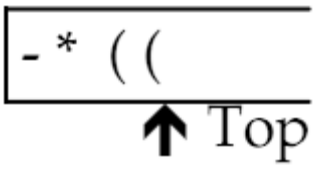
8. On *, Stack being push *



9. On (, push (

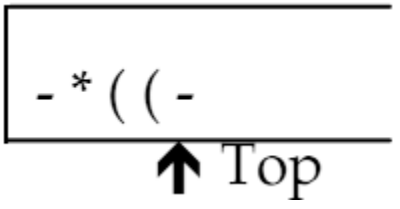


10. On (, push (



11. On c, operand hence prints c — b a b + c

12. On-, push -

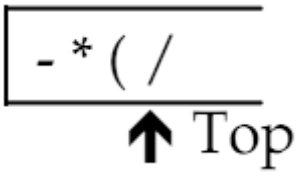


13. On d, operand hence prints d — b a b + c d

14. On), pop till (and then print.
Therefore pop -, print -, pop(

b a b + c d -

15. On /, push /



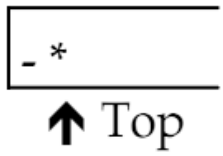
16. On a, operand, Hence print a — b a b + c d - a

17. On +, Stack top is /. So, pop till (and then print /. Therefore, pop /, print /, pop (, push +

b a b + c d - a /

18. On a, operand, Hence print a — b a b + c d - a / a

19. On), pop till (and then print +.
Therefore, pop +, print +, pop (



b a b + c d - a / a +

20. End of the Infix expression

21. pop all and print, Hence print *. print - b a b + c d - a / a + * -

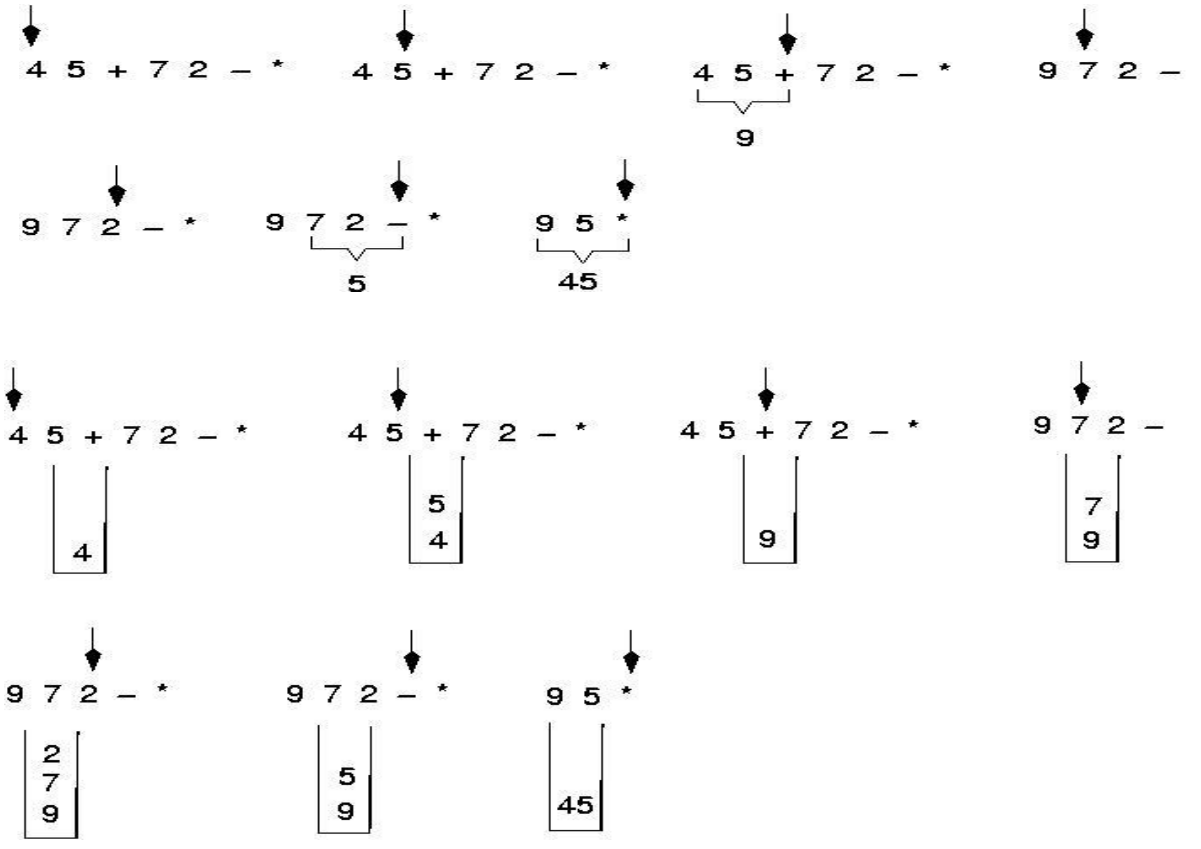
Therefore the generated postfix expression is

b a b + c d - a / a + * -

Evaluating A Postfix Expression

Following are the steps to evaluate a postfix expression:

1. *Each operator in a postfix string refers to the previous two operands in the string.*
2. *Suppose that each time we read an operand we push it into a stack. When we reach an operator, its operands will then be top two elements on the stack*
3. *We can then pop these two elements, perform the indicated operation on them, and push the result on the stack.*
4. *So that it will be available for use as an operand of the next operator.*
5. *Use a stack to evaluate an expression in postfix notation.*
6. *The postfix expression to be evaluated is scanned from left to right.*
7. *Variables or constants are pushed onto the stack.*
8. *When an operator is encountered, the indicated action is performed using the top elements of the stack, and the result replaces the operands on the stack.*



Postfix to Prefix Conversion

1. *Read the Postfix expression from left to right*
2. *If the symbol is an operand, then push it onto the Stack*
3. *If the symbol is an operator, then pop two operands from the Stack*
   Create a string by concatenating the two operands and the operator before them.
   string = operator + operand2 + operand1
   And push the resultant string back to Stack
4. *Repeat the above steps until end of Prefix expression.*

2.4.4 Prefix Expression (Polish Notation)

Prefix expression contains an operator followed by 2 operands. It is called polish notation named after polish scientist hence a.k.a. Polish notation. e.g. +AB

Following are the characteristics of prefix expression:

1. *Operator precedes operands*

a. *infix expression: A + B*

b. *prefix expression: +AB*

2. *Parentheses become unnecessary*

   a. *infix expression: (A + B) * C*

   b. *prefix expression: * + A B C*

3. *Write out operands in original order .*
4. *Place operators in front of their operands.*
5. *If there's a compound expression, the prefix expression may have two or more operators in a row.*
6. *If parentheses are not present, pay attention to precedence.*

Prefix evaluation Example

1. scan left to right until we find the first operator immediately followed by pair of operands
2. evaluate expression, and replace the "used" operator & operands with the result
3. continue until a single value remains

e.g. + * / 4 2 3 9

+ * 2 3 9 // 4/2 evaluated

+ 6 9 // 2*3 evaluated

15 // 6+9 evaluated

Prefix to Infix Conversion

1. *Read the Prefix expression in reverse order (from right to left)*
2. *If the symbol is an operand, then push it onto the Stack*
3. *If the symbol is an operator, then pop two operands from the Stack*
   Create a string by concatenating the two operands and the operator between them.
   string = (operand1 + operator + operand2)
   And push the resultant string back to Stack
4. *Repeat the above steps until end of Prefix expression.*

Prefix to Postfix Conversion

1. *Read the Prefix expression in reverse order (from right to left)*
2. *If the symbol is an operand, then push it onto the Stack*
3. *If the symbol is an operator, then pop two operands from the Stack*

Create a string by concatenating the two operands and the operator after them.
string = operand1 + operand2 + operator
And push the resultant string back to Stack

4. *Repeat the above steps until end of Prefix expression.*

Infix to Prefix Algorithm

1. *Reverse the infix expression.*
2. *Make Every '(' as ')' and every ')' as '('*
3. *Convert expression to postfix form.*
4. *Reverse the expression again.*

e.g. (A+B^C)*D+E^5

1. Reverse the infix expression.
   5^E+D*)C^B+A(
2. Make Every '(' as ')' and every ')' as '('
   5^E+D*(C^B+A)
3. Convert expression to postfix form.
4. Reverse the expression.
   +*+A^BCD^E5

Try it yourself: Postfix to Infix Conversion

3. LINKED LISTS

3.1 INTRODUCTION

We have seen representation of linear data structures by using sequential allocation method of storage, as in, arrays. But this is unacceptable in cases like:

1. Unpredictable storage requirements:
    - The exact amount of data storage required by the program varies with the amount of data being processed. This may not be available at the time we write programs but are to be determined later.
    - For example, linked allocations are very beneficial in case of polynomials. When we add two polynomials, and none of their degrees match, the resulting polynomial has the size equal to the sum of the two polynomials to be added. In such cases we can generate nodes (allocate memory to the data member) whenever required, if we use linked representation (dynamic memory allocation).
2. Extensive data manipulation takes place.
    - Frequently many operations like insertion, deletion etc, are to be performed on the linked list.

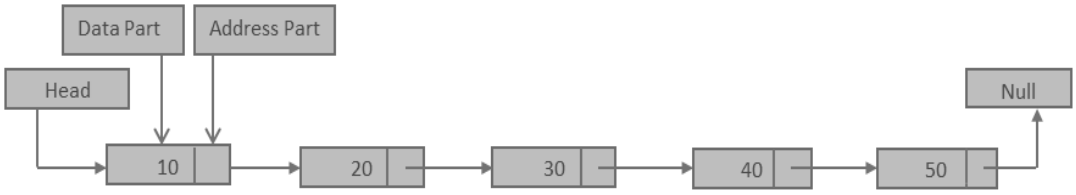
Pointers are used for the dynamic memory allocation. These pointers are always of same length regardless of which data element it is pointing to(int, float, struct etc,). This enables the manipulation of pointers to be performed in a uniform manner using simple techniques. These make us capable of representing a much more complex relationship between the elements of a data structure than a linear order method.

The use of pointers or links to refer to elements of a data structure implies that elements, which are logically adjacent, need not be physically adjacent in the memory. Just like family members dispersed, but still bound together.

3.2 SINGLY LINKED LIST

A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list.

This is a list, which may consist of an ordered set of elements that may vary in number. Each element in this linked list is called as *node*. A node in a singly linked list consists of two parts, a *information part* where the actual data is stored and a *link part*, which stores the address of the successor(next) node in the list. The order of the elements is maintained by this explicit link between them. The typical structure of linked list is shown below:



Advantages of Linked Lists over Arrays and vectors

1. A linked list can easily grow or shrink in size.
2. Insertion and deletion of nodes is quicker with linked lists than with vectors.

In above figure, the arrows represent the links. The **data** part of each node consists of the marks obtained by a student and the **next** part is a pointer to the next node. The NULL in the last node indicates that this node is the last node in the list and has no successors at present. In the above the example the data part has a single element marks but you can have as many elements as you require, like his name, class etc.

3.2.1 Creating a List

There are several operations that we can perform on linked lists. We can see some of them now. To begin with we must define a structure for the node containing a data part and a link part. We will write a program to show how to build a linked list by adding new nodes in the beginning. We will also see the function to add the node at the end or in the middle of the linked list. A function display() is used to display the contents of the nodes present in the linked list and a function delete(), which can delete any node in the linked list .

```
include <stdio.h>
include <alloc.h>  /* required for dynamic memory allocation */
typedef struct ListNode
 {
        int data;
       struct node *ListNode;
 }NODE;
```

To declare a variable of type ListNode, following declaration is done:

```
ListNode *newNode;
```

To allocate the memory to the node either of the following can be used:

```
newNode = malloc(sizeof(NODE));
```

or

newNode = new ListNode;

The next step is to declare a pointer to serve as the list head (First node), as shown below.

*ListNode *head;*

Once you have declared a node data structure and have created a NULL head pointer, you have an empty linked list. The next step is to implement operations with the list.

To append a node to a linked list means to add the node to the end of the list. The pseudo code is shown below.

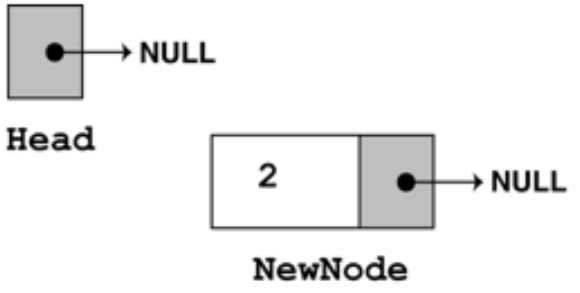
```
Create a new node.
        Store data in the new node.
        If there are no nodes in the list
                Make the new node the first node.
        Else
                Traverse the List to Find the last node.
                Add the new node to the end of the list.
        End If.
```

```
void appendNode(int num)
{
        ListNode *newNode, *nodePtr;

        // Allocate a new node & store num
        newNode = new ListNode;
        newNode->value = num;
        newNode->next = NULL;
        // If there are no nodes in the list
        // make newNode the first node
        if (!head)
                head = newNode;
        else    // Otherwise, insert newNode at end
        {
                // Initialize nodePtr to head of list
                nodePtr = head;
                // Find the last node in the list
                while (nodePtr->next)
                        nodePtr = nodePtr->next;
                // Insert newNode as the last node
                nodePtr->next = newNode;

                newNode->next = NULL;
        }
}
```

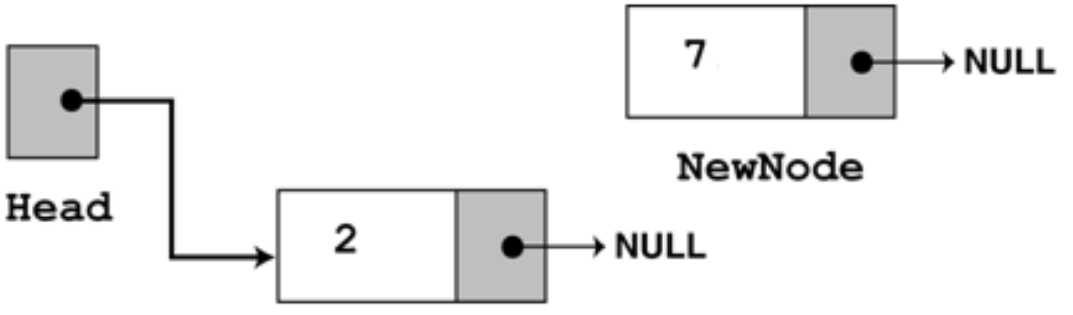
Understanding the Program

1. The head pointer is declared as a global variable. head is automatically initialized to 0 (NULL), which indicates that the list is empty.
2. The first call to appendNode passes 2 as the argument. In the following statements, a new node is allocated in memory, 2 is copied into its value member, and NULL is assigned to the node's next pointer.

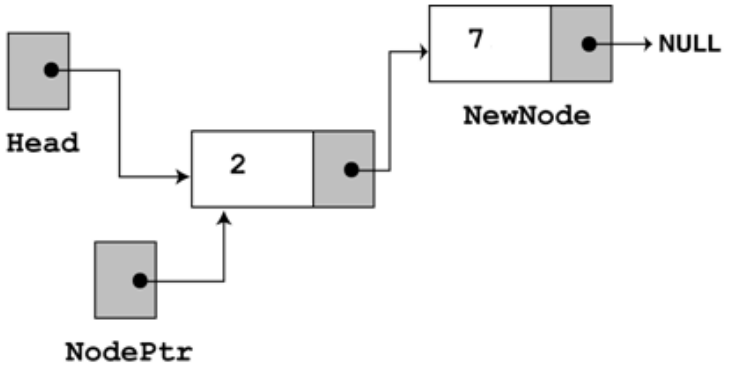
```
newNode = new ListNode;
newNode->value = num;
newNode->next = nULL;
```



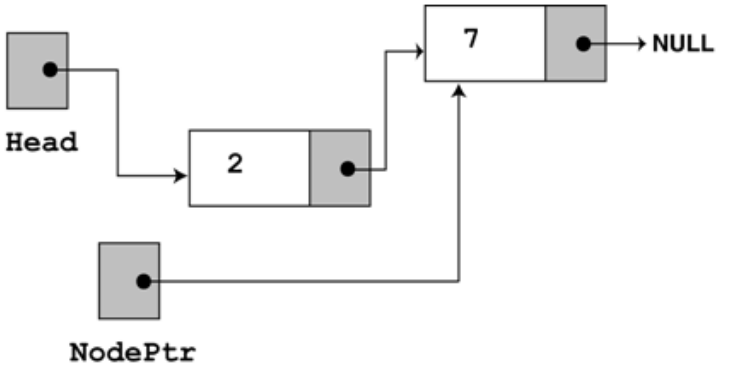
In the second call to appendNode, 7 is passed as the argument. Once again, the first three statements in the function create a new node, store the argument in the node's value member, and assign its next pointer to NULL.



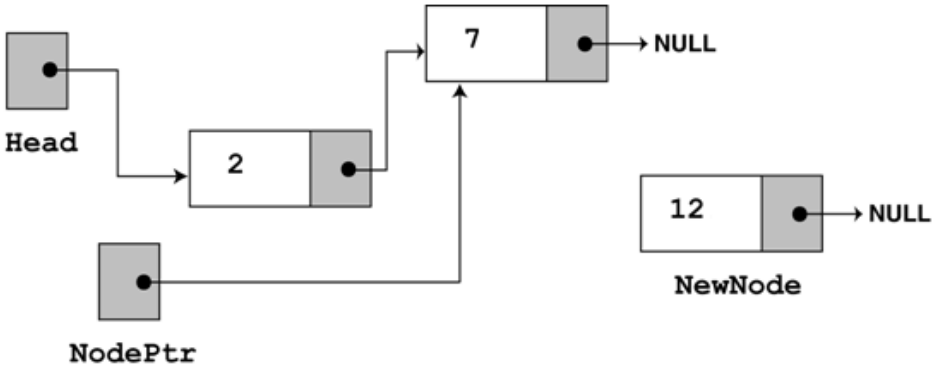

```
// Initialize nodePtr to head of list
nodePtr = head;
// Find the last node in the list
while (nodePtr->next)
        nodePtr = nodePtr->next;
// Insert newNode as the last node
nodePtr->next = newNode;

//Point the nodePtr to the newNode

nodePtr = newNode;
```



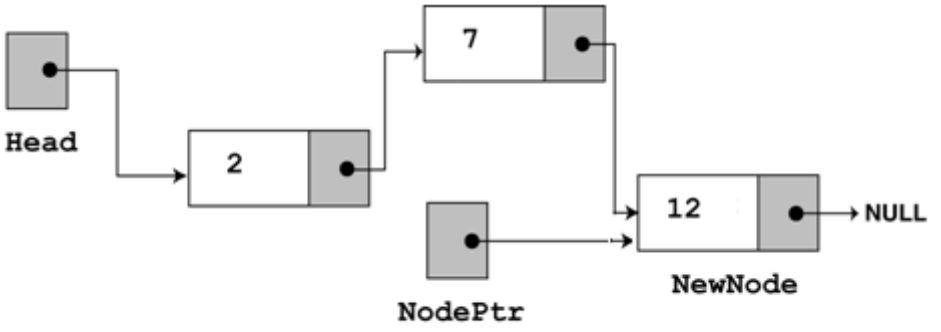
nodePtr is already at the end of the list, so the while loop immediately terminates. The last statement, nodePtr->next = newNode; causes nodePtr->next to point to the new node. This inserts newNode at the end of the list. Also, nodePtr is moved to the last node of the list i.e. newNode. So, the list will look as follows.



The third time appendNode is called, 12 is passed as the argument. Once again, the first three statements create a node with the argument stored in the value member.



The same process is repeated as in the case of inserting value 7. The structure becomes as under:



This process is repeated till the user wants to add nodes into the list.

3.2.2 Traversing the list

The displayList member function traverses the list, displaying the value member of each node. The following pseudo code represents the algorithm.

Assign List head to node pointer.
While node pointer is not NULL
Display the value member of the node pointed to by node pointer.
Assign node pointer to its own next member.
End While.

```
void displayList(void)
{
        ListNode *nodePtr;

        nodePtr = head;
        while (nodePtr)
        {
                printf("%d\n",nodePtr->value);
                nodePtr = nodePtr->next;
        }
}
```

3.2.3 To add a node at the beginning and at the end of the linked list

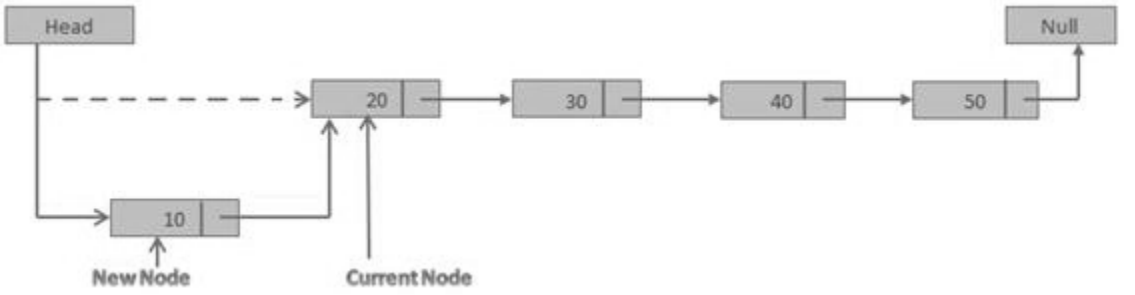
newnode = malloc(sizeof(NODE));

Insertion at the beginning

newnode->data = num;
newnode->link = NULL;

newnode->next=head;

head=newnode;



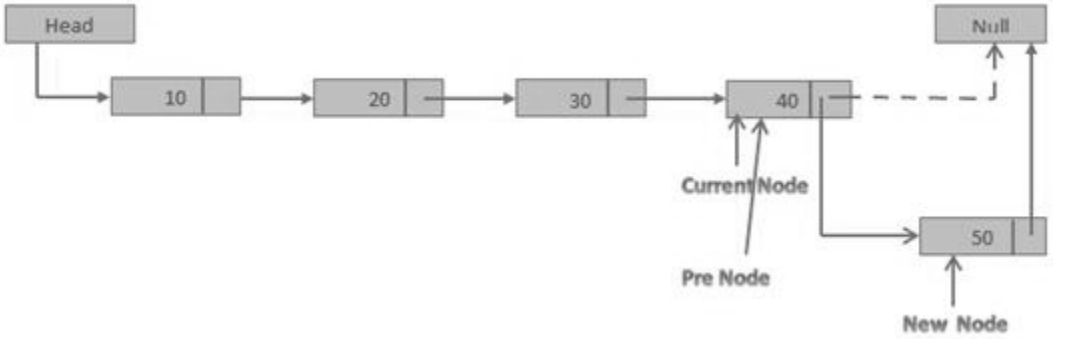
Insertion at the end

while(ptr->next!=NULL)

ptr=ptr->next;

ptr->next=newnode;

newnode->next=NULL;



3.2.4 To add a node before or after a given key node

Insertion before a key node

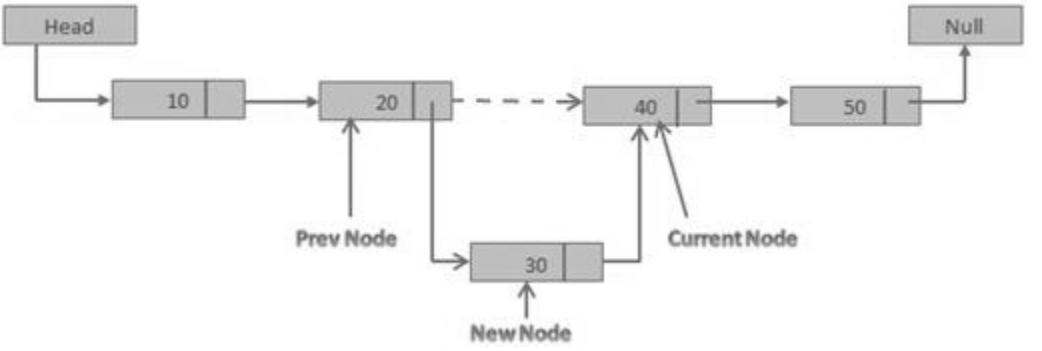
key=40

while(ptr->next->value!=key)

ptr=ptr->next;

newnode->next=ptr->next;

ptr->next=newnode;



Insertion after a key node

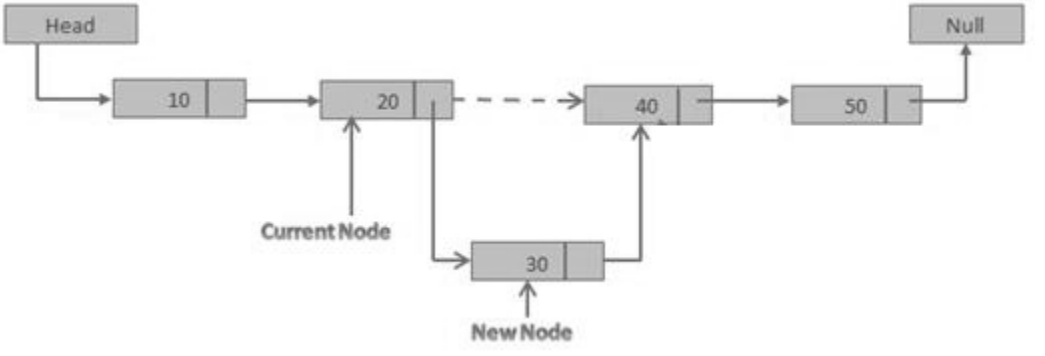
key=20

while(ptr->value!=key)

ptr=ptr->next;

newnode->next=ptr->next;

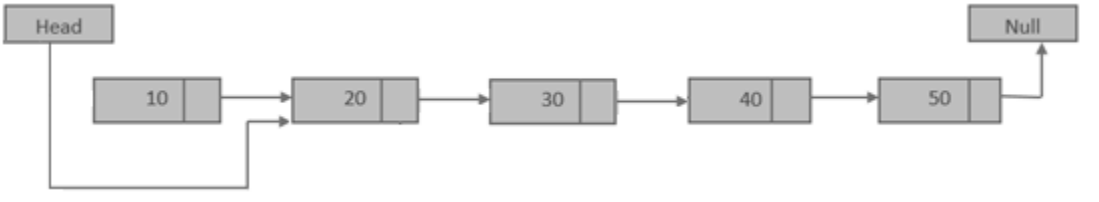
ptr->next=newnode;



3.2.5 To delete the first and last node

Deleting the first node

head=head->next



Delete the last node

while(ptr->next->next!=NULL)

ptr=ptr->next;

ptr->next=NULL;

3.2.6 To delete the specified node from the list

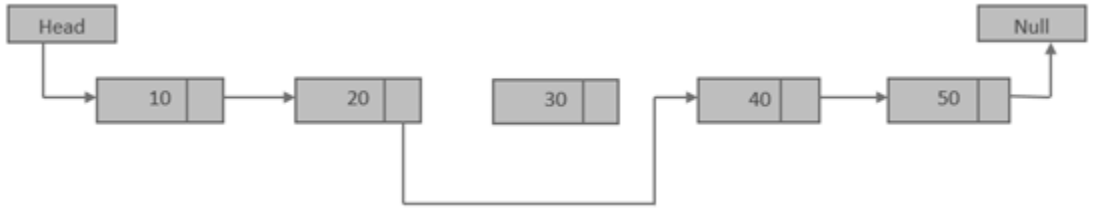
Deleting a node from a linked list requires to make sure that the node from the list must be removed without breaking the links created by the next pointers

key=30

while(ptr->next->value!=key)

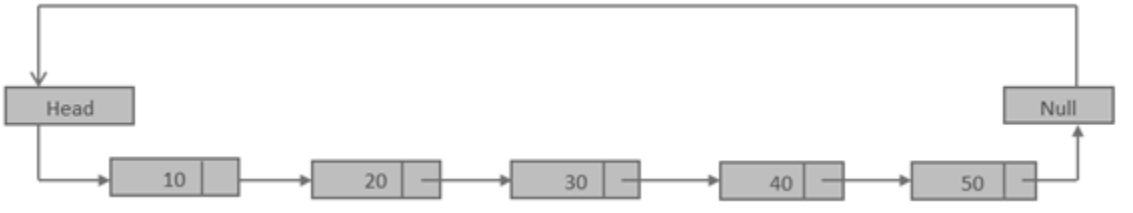
ptr=ptr->next;

ptr->next=ptr->next->next;



3.3 CIRCULAR LINKED LIST

A circular linked list is a linked list in which the last node's next pointer points to the head element. A circularly linked list node looks exactly the same as a linear singly linked list.



3.3.1 Creating a Circular Linked Lists

To create a node to a circular linked list means to add the node to the end of the list and next pointer of the node points to head. The pseudo code is shown below.

Create a new node.

Store data in the new node.

If there are no nodes in the list

Make the new node the first node.

```
        Else
                Traverse the List to Find the last node.
                Add the new node to the end of the list.
                Set the next part of new node to first.
        End If.
```

```
void appendNode(int num)
{
        ListNode *newNode, *nodePtr;

        // Allocate a new node & store num
        newNode = new ListNode;
        newNode->value = num;
        newNode->next = NULL;
        // If there are no nodes in the list
        // make newNode the first node
        if (!head)
                head = newNode;
        else    // Otherwise, insert newNode at end
        {
                // Initialize nodePtr to head of list
                nodePtr = head;
                // Find the last node in the list
                while (nodePtr->next)
                        nodePtr = nodePtr->next;
                // Insert newNode as the last node
                nodePtr->next = newNode;
                newNode->next=head;
        }
}
```

3.3.2 Traversing the list

The displayList member function traverses the list, displaying the value member of each node. The following pseudo code represents the algorithm.

```
Assign List head to node pointer.
While node pointer is not Head
        Display the value member of the node pointed to by node pointer.
        Assign node pointer to its own next member.
End While.
```

```
void displayList(void)
{
        ListNode *nodePtr;
```

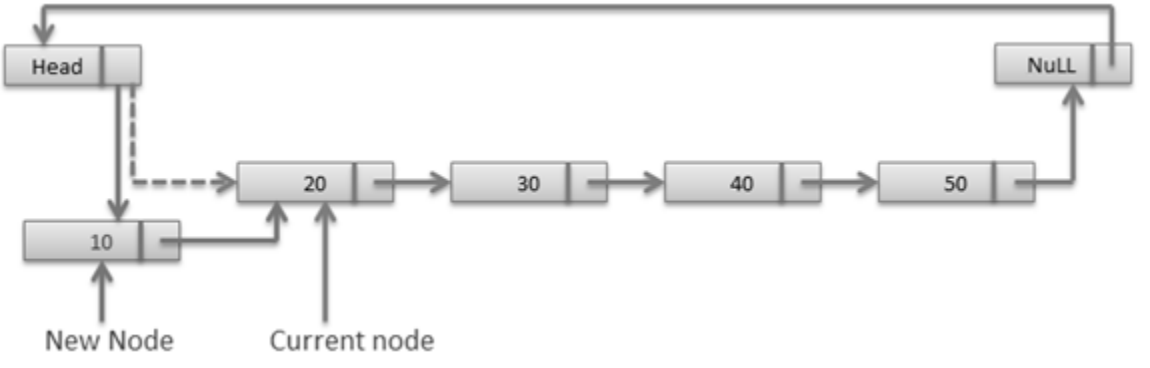
```
        nodePtr = head;
        while (nodePtr-next!=head)
        {
                printf("%d\n",nodePtr->value);
                nodePtr = nodePtr->next;
        }
}
```

3.3.3 To add a node at the beginning or end of the circular linked list

As contrast to the linear linked list, due to the last node pointing to the first node, we will have to traverse till end of the list to set the new node as last and pointing to the first node. The same will be used for adding the node at the end of the list.

Insertion at the beginning

```
while(ptr->next!=head)
        ptr=ptr->next;
ptr->next=newnode;
newnode->next=head;
head=newnode;
```



Insertion at the end

```
while(ptr->next!=head)
        ptr=ptr->next;
ptr->next=newnode;
newnode->next=head;
```

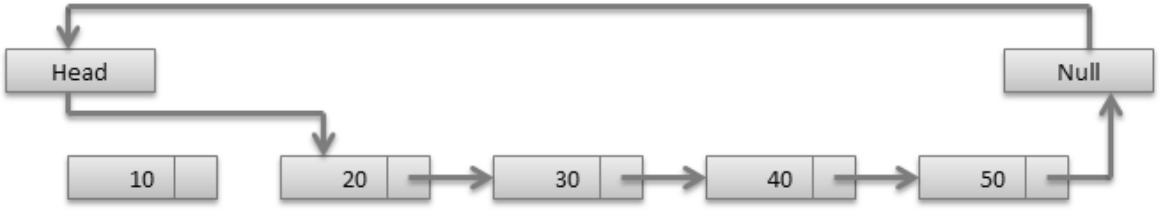
Try it yourself: Insertion at a specific position (Before or after a key node)

3.3.4 To delete a node at the beginning or end of the circular linked list

Similar to insertion at beginning or end, due to the last node pointing to the first node, we will have to traverse till end of the list to set the previous node pointing to the first node. The same will be used for removing the node at the end of the list.

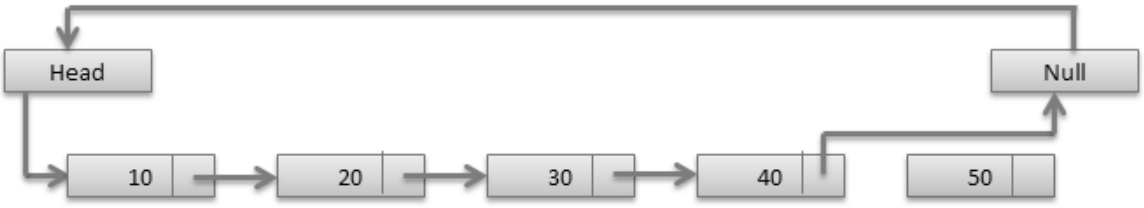
Deletion at the beginning

```
while(ptr->next!=head)
        ptr=ptr->next;
ptr->next=head->next;
head=head->next
```



Deletion at the end

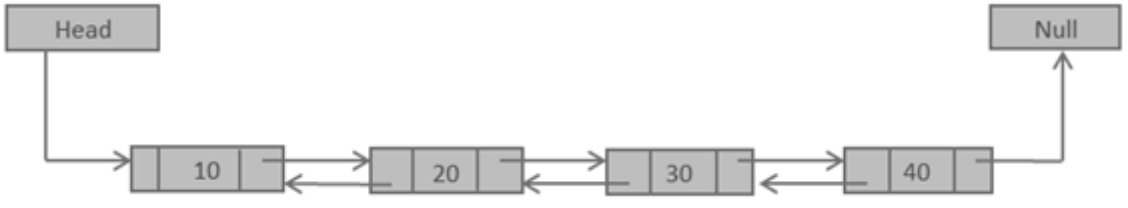
```
while(ptr->next->next!=head)
        ptr=ptr->next;
ptr->next=head;
```



Try it yourself: Deletion a specific key node

3.4 DOUBLY LINKED LIST OR TWO-WAY LINKED LIST

In a singly linked list we can traverse in only one direction (forward), i.e. each node stores the address of the next node in the linked list. It has no knowledge about where the previous node lies in the memory. If we are at the 12^{th} node(say) and if we want to reach 11^{th} node in the linked list, we have to traverse right from the first node. This is a cumbersome process. Some applications require us to traverse in both forward and backward directions. Here we can store in each node not only the address of the next node but also the address of the previous node in the linked list. NEXT holds the memory location of the Previous Node in the List. PREV holds the memory location of the Next Node in the List. This arrangement is often known as a Doubly linked list . The node and the arrangement is shown below:



The left pointer of the leftmost node and the right pointer of the rightmost node are NULL indicating the end in each direction.

The following program implements the doubly linked list.

```
struct Doubly {
  int data;
  sturct Doubly* next;
  struct Doubly* prev;
};
```

Doubly Linked List are more convenient than Singly Linked List since we maintain links for bi-directional traversing. We can traverse in both directions and display the contents in the whole List. In Doubly Linked List, we can traverse from Head to Tail as well as Tail to

Head. Each Node contains two fields, called Links , that are references to the previous and to the Next Node in the sequence of Nodes. The previous link of the first node and the next link of the last node points to NULL.

3.4.1 Creating a Doubly Linked Lists

To create a node to a circular linked list means to add the node to the end of the list and next pointer of the node points to head. The pseudo code is shown below.

```
Create a new node.
        Store data in the new node.
        If there are no nodes in the list
                Make the new node the first node.
        Else
                Traverse the List to Find the last node.
                Add the new node to the end of the list.
                Set the next and previous pointers accordingly.
        End If.
```

```
void appendNode(int num)
{
        ListNode *newNode, *nodePtr;

        // Allocate a new node & store num
        newNode = new ListNode;
        newNode->value = num;
        newNode->next = NULL;
        newNode->prev = NULL;
        // If there are no nodes in the list
        // make newNode the first node
        if (!head)
                head = newNode;
        else    // Otherwise, insert newNode at end
        {
                // Initialize nodePtr to head of list
                nodePtr = head;
                // Find the last node in the list
                while (nodePtr->next)
                        nodePtr = nodePtr->next;
                // Insert newNode as the last node
                nodePtr->next = newNode;
                newNode->prev=nodePtr;
                newNode->next=NULL;
```

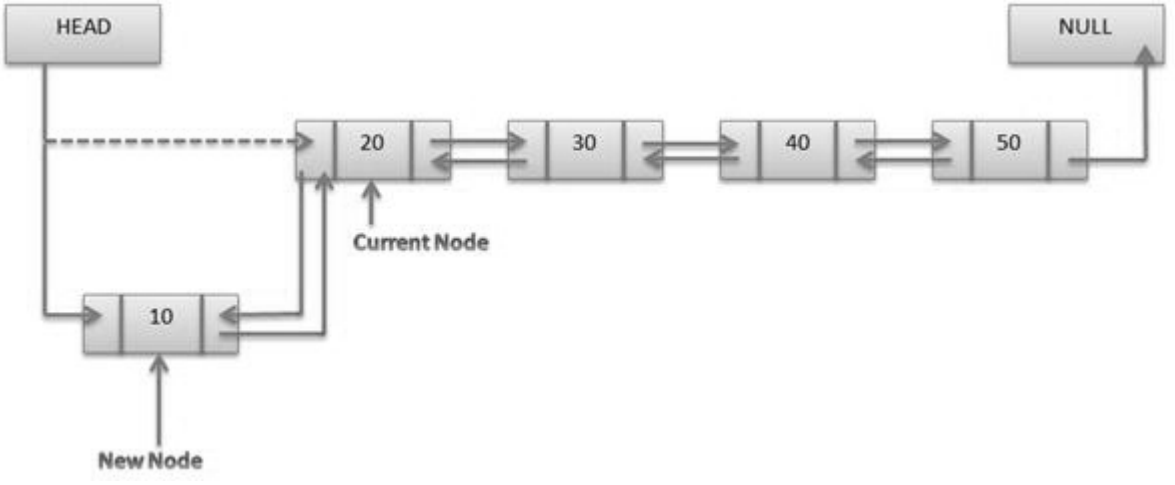
```
        }
}
```

Traversal of the list is same as linear linked list.

3.4.2 Adding a node at beginning or end in a Doubly Linked Lists

Insertion at the beginning

```
newnode->prev=NULL;
newnode->next=head;
head->prev=newnode;
head=newnode;
```



Insertion at the end

```
while(ptr->next!=NULL)
        ptr=ptr->next;
newnode->prev=ptr;
newnode->next=NULL;
ptr->next=newnode;
```

3.4.3 To add a node before or after a given key node

Insertion before a key node

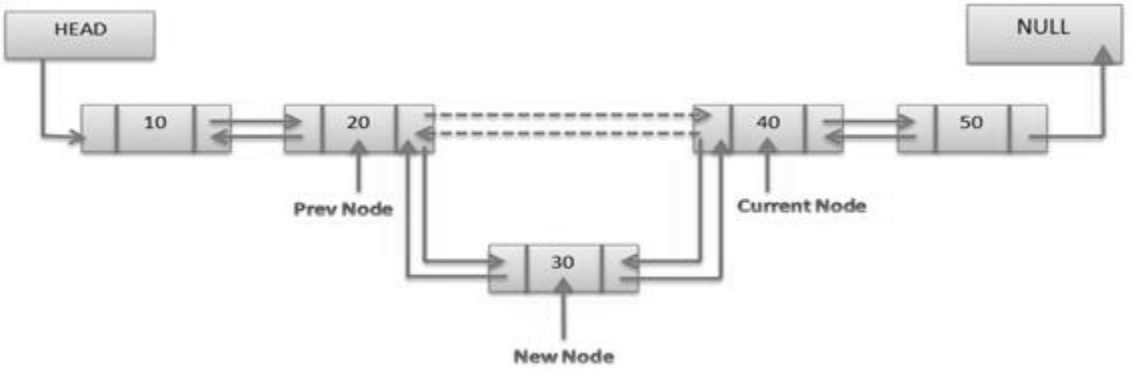
```
key=40;
while(ptr->data!=key)
        ptr=ptr->next;
```

newnode->prev=ptr->prev;

newnode->next=ptr;

ptr->prev->next=newnode;

ptr->prev=newnode;



Insertion after a key node

key=20;

while(ptr->data!=key)

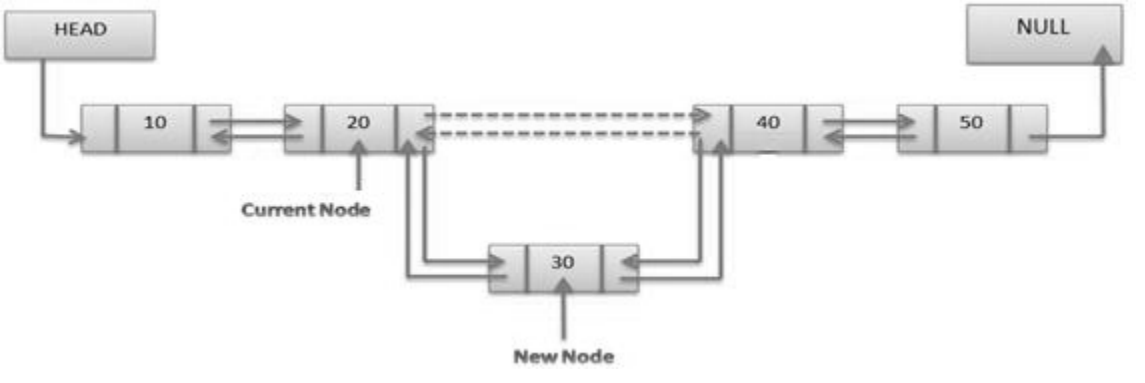
ptr=ptr->next;

newnode->prev=ptr;

newnode->next=ptr->next;

ptr->next->prev=newnode;

ptr->next=newnode;

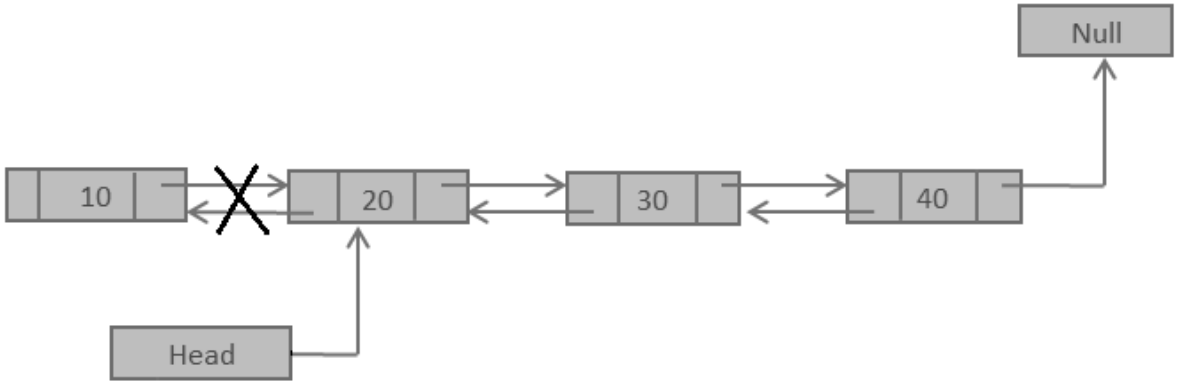


3.4.4 To delete first node, last node or a given key node

Deletion at the beginning

head=head->next;

head->prev=NULL;

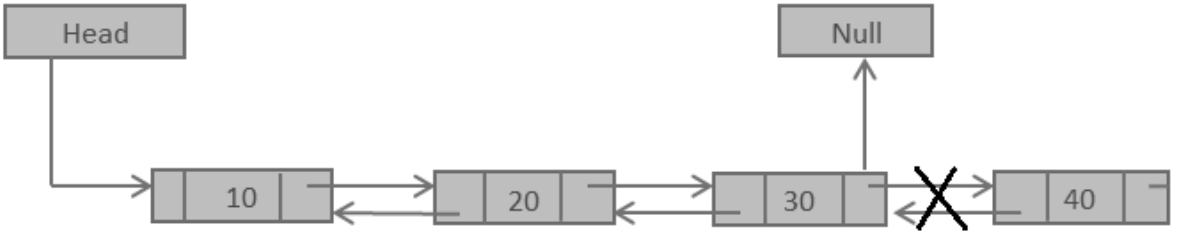


Deletion at the end

while(ptr->next!=NULL)

ptr=ptr->next;

ptr->prev->next=NULL;



Deletion of a key node

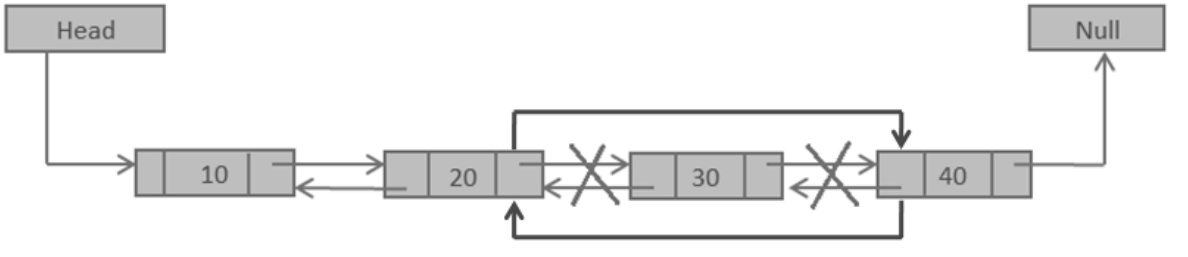
Key=30;

while(ptr->data!=key)

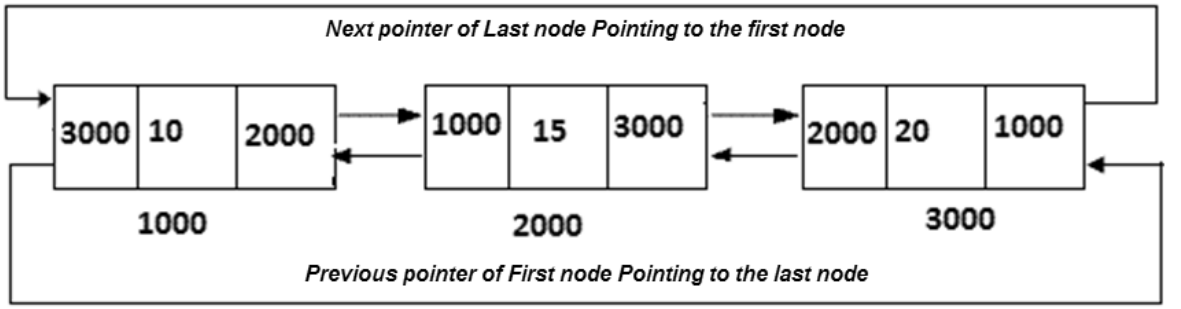
ptr=ptr->next;

ptr->prev->next=ptr->next;

ptr->next->prev=ptr->prev;



Try it yourself: Circular Doubly Linked List



3.5 APPLICATIONS OF THE LINKED LISTS AND POLYNOMIALS

In computer science linked lists are extensively used in Data Base Management Systems Process Management, Operating Systems, Editors etc. Earlier we saw that how singly linked list and doubly linked list can be implemented using the pointers. We also saw that while using arrays vary often the list of items to be stored in an array is either too short or too big as compared to the declared size of the array. Moreover, during program execution the list cannot grow beyond the size of the declared array. Also, operations like insertions and deletions at a specified location in a list require a lot of movement of data, thereby leading to an inefficient and time-consuming algorithm.

The primary advantage of linked list over an array is that the linked list can grow or shrink in size during its lifetime. In particular, the linked list ‘s maximum size need not be known in advance. In practical applications this often makes it possible to have several data structures share the same space, without paying particular attention to their relative size at any time. The second advantage of providing flexibility in allowing the items to be rearranged efficiently is gained at the expense of quick access to any arbitrary item in the list. In arrays we can access any item at the same time as no traversing is required.

We are not suggesting that you should not use arrays at all. There are several applications where using arrays is more beneficial than using linked lists. We must select a particular data structure depending on the requirements.

One of the application of the linked list is polynomial representations.

Polynomials can be maintained using a linked list. To have a polynomial like 5x4 _ 2x3 + 7x2 + 10x –8 , each node should consist of three elements, namely coefficient, exponent and a link to the next item. While maintaining the polynomial it is assumed that the exponent of each successive term is less than that of the previous term. If this is not the case you can also use a function to build a list, which maintains this order. Once we build a linked list to represent the polynomial we can perform operations like addition and multiplication.

4. SEARCHING, SORTING & HASHING

4.1 INTRODUCTION

In many cases we require the data to be presented in the form where it follows certain sequence of the records. If we have the data of the students in the class, then we will prefer to have them arranged in the alphabetical manner. For preparing the result sheet, we would like to arranger the data as per the examination numbers. When we prepare the merit list we would like to have the same data so that it is arranged in the decreasing order of the total marks obtained by the students.

Thus arranging the data in either ascending or descending manner based on certain key in the record is known as SORTING. As we do not receive the data in the sorted form, we are required to arrange the data in the particular from. For ascending order we will require the smallest key value first. Thus till we do not get all the data items, we cannot start arranging them. Arranging the data as we receive it is done using linked lists. But in all other cases, we need to have all the data, which is to be sorted, and it will be present in the form of an ARRAY. Sometimes it is very important to search for a particular record, may be, depending on some value. The process of finding a particular record is known as SEARCHING.

4.2 SEARCHING TECHNIQUES

We will discuss two searching methods – the sequential search and the binary search.

4.2.1 Sequential Search

This is a natural searching method. Here we search for a record by traversing through the entire list from beginning until we get the record. For this searching technique the list need not be ordered. The algorithm is presented below:

1. Set flag =0.
2. Set index = 0.
3. Begin from index at first record to the end of the list and if the required record is found make flag = 1.
4. At the end if the flag is 1, the record is found, otherwise the search is failure.

```
int Lsearch(int L[SIZE], int ele )
{
  int it;
        for(it = 1; it<=SIZE; it++)
         {
                if( L[it] == ele)
                 {
                  return 1; break;
```

```
            }
        }
  return 0;
}
```

Analysis

Whether the search takes place in an array or a linked list, the critical part in performance is the comparison in the loop. If the comparisons are less the loop terminates faster.

The least number of iterations that could be required is 1 if the element that was searched is first one in the list. The maximum comparison is N (N is the total size of the list), when the element is the last in the list. Thus if the required item is in position 'i' in the list, 'i' comparisons are required. Hence the average number of comparisons is

$$= \frac{1 + 2 + 3 + ... + I + ... + N}{N}$$

$$= \frac{N(N+1)}{2 * N}$$

$$= (N + 1) / 2$$

Sequential search is easy to write and efficient for short lists. It does not require the list to be sorted. However if the list is long, this searching method becomes inefficient, as it has to travel through the whole list. We can overcome this shortcoming by using Binary Search Method.

4.2.2 Binary Search

Binary search method employs the process of searching for a record only in half of the list, depending on the comparison between the element to be searched and the central element in the list. It requires the list to be sorted to perform such a comparison. It reduces the size of the portion to be searched by half after each iteration.

Let us consider an example to see how this works.

The numbers in the list are 10 20 30 40 50 60 70 80 90.

The element to be searched is 30.

First it is compared with 50(central element). Since it is smaller than the central element we consider only the left part of the list(i.e. from 10 to 40) for further searching. Next the comparison is made with 30 and returns as search is successful.

Let us see the function, which performs this on the list.

```
int Bsearch(int list[SIZE],int ele)
 {
        int top, bottom, middle;
        top= SIZE –1; bottom = 0;
        while(top > = bottom)
        {
                middle = (top + bottom)/2;
                if(list[middle]==ele)
                        return middle;
                else if(list[middle] < ele)
                        bottom = middle + 1;
                else
                        top = middle –1;
        }
        return –1;
}
```

Analysis

In this case, after each comparison either the search terminates successfully or the list remaining to be searched, is reduced by half. So after k comparisons the list remaining to be searched is $N/(2^k)$where N is the number of elements. Hence even at the worst case this method needs no more than $\log_2(N+1)$ comparisons.

4.3 SORTING

Arranging the data in either ascending or descending manner based on certain key in the record is known as SORTING. We will learn selection sort, Bubble sort, Quick sort, Merge sort, Radix sort and Heap sort. We will give the algorithms/pseudo codes for these sorts. Students can write the program for these and test them.

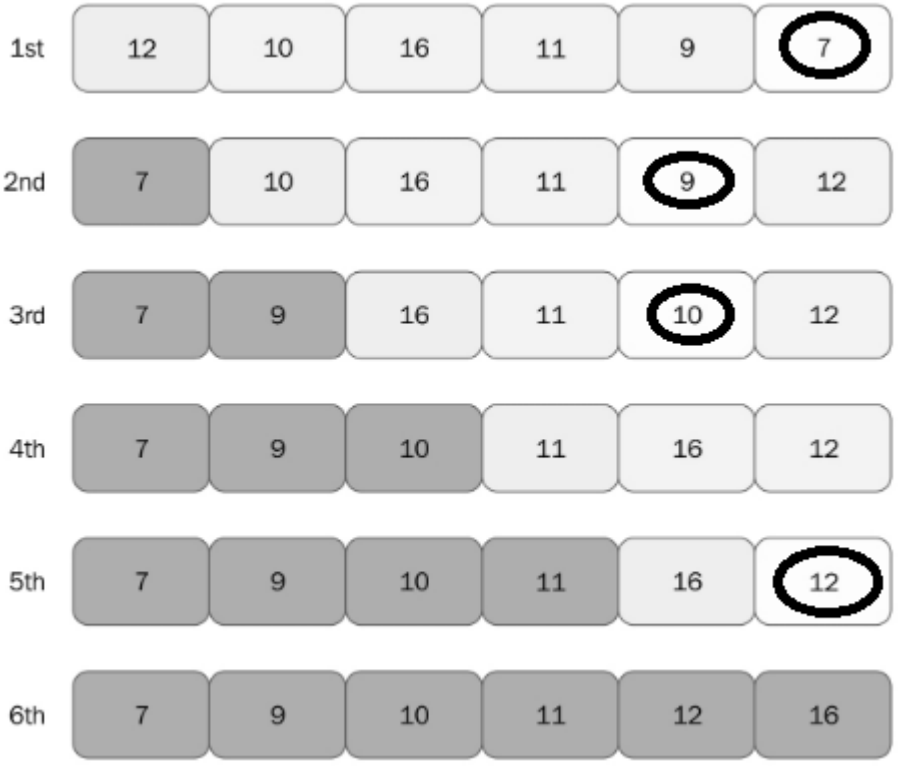
4.3.1 Selection Sort

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

1. *Start by finding the **smallest** entry.*
2. *Swap the smallest entry with the **first entry**.*
3. *Part of the array is now sorted.*

4. *Find the smallest element in the remaining unsorted side.*
5. *Swap with the front of the unsorted side i.e. actually, this will become the next element in the sorted side.*
6. *Now, the sorted side size increased by one element.*
7. *Repeat step 4 to 6 till the unsorted side has just one number, since that number must be the largest number.*
8. *The sorted side has the smallest numbers, arranged from small to large.*

Following is the trace of the selection sort.



Assume we have an array 'K' with 'N' number of elements. This algorithm arranges elements in ascending order. 'Pass' is an index variable, which indicates the number of passes. The variable 'min_index' denotes the position of the smallest element encountered in that pass. 'i' is another index variable. The array and the size are passed to the function.

1. Using Pass index variable repeat steps from first record to last – 1 records and perform all the steps up to 4.
2. Initialize the minimum index as min_index = pass.
3. Obtain the element with the smallest value.

   for(i= pass+1; i < N; i++)

```
        if( K[i] < K[min_index])
            min_index = i;
```

4. exchange the elements

```
if(min_index != pass)
        {
                temp = K[pass];
                K[pass] = K[min_index];
                K[min_index] = temp;
        }
```

Analysis:

In the first pass N-1 records are examined. In the second pass N-2 records are examined.

In the third pass N-3 records are examined. Therefore in ith pass – (N-i) records are examined. Therefore the efficiency is given by $N(N-1)/2 = O(N^2)$

1. Worst Case Time Complexity [Big-O]: $O(N^2)$
2. Best Case Time Complexity [Big-omega]: $O(N^2)$
3. Average Time Complexity [Big-theta]: $O(N^2)$

4.3.2 Bubble sort

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on. This will continued till the largest element is placed at the last position. Now, the process is repeated for n-1 elements by skipping the last element. This will go on till all the elements are sorted. If we have total n elements, then we need to repeat this process for n-1 times.

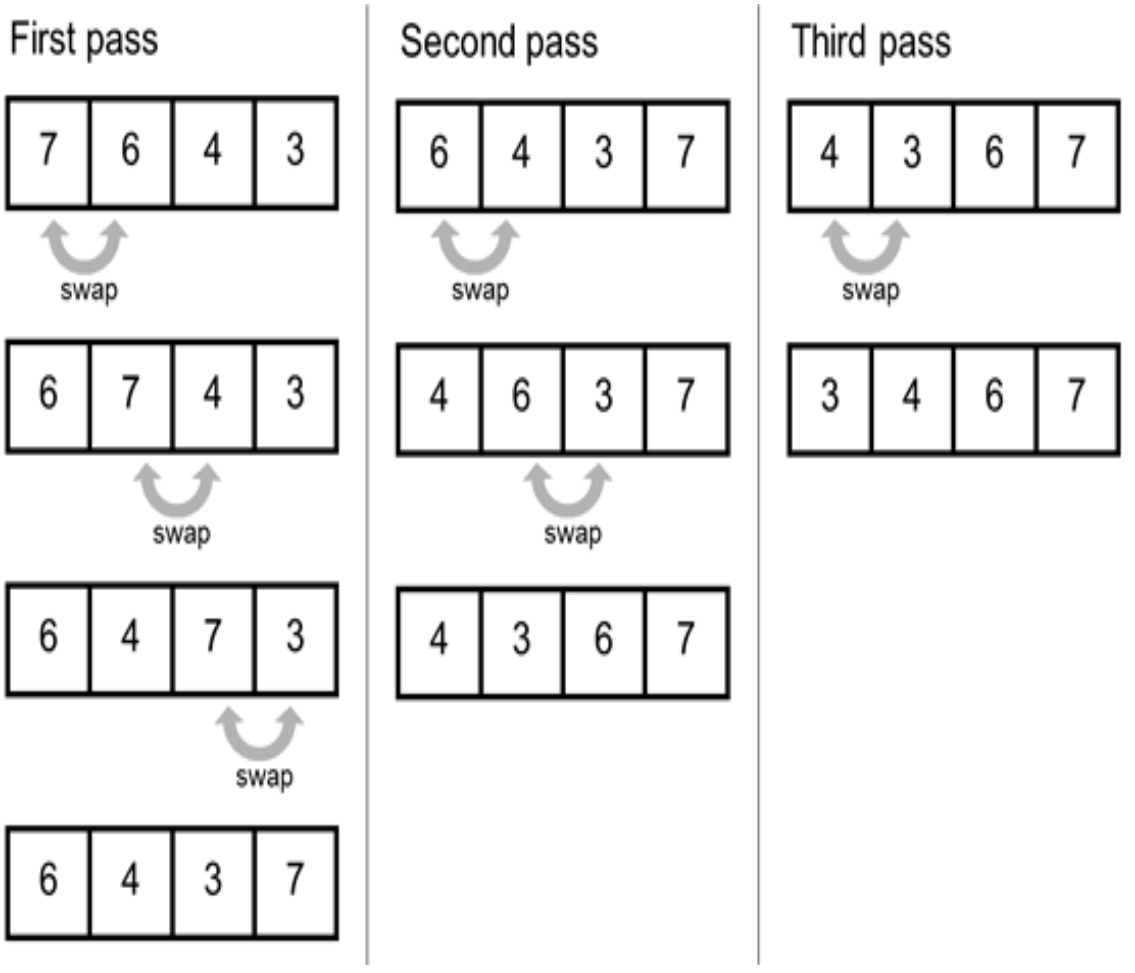
It is known as bubble sort, because after every complete iteration, the largest element in the given list is bubbles up towards the last place just like a water bubble rises up to the water surface.

Let's say, arr is the given array with n elements to be sorted. Pseudo Code for bubble sort is given below:

```
for(i = 0; i < n; i++)
{
        for(j = 0; j < n-i-1; j++)
        {
```

```
        if( arr[j] > arr[j+1])
                {
                        // swap the elements
                        temp = arr[j];
                        arr[j] = arr[j+1];
                        arr[j+1] = temp;
                }
        }
}
```

Following is the trace of Bubble Sort



Analysis

Similar to Selection sort, the total number of comparisons in Bubble sort is

=(N –1) + (N-2) + ... + 2 + 1

=(N-1)*N/2

= $O(N^2)$

1. Worst Case Time Complexity [Big-O]: $O(N^2)$
2. Best Case Time Complexity [Big-omega]: O(N)
3. Average Time Complexity [Big-theta]: $O(N^2)$

4.3.3 Insertion sort

Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers.

If I give you another card, and ask you to insert the card in just the right position, so that the cards in your hand are still sorted. What will you do? You will have to go through each card from the starting or the back and find the right position for the new card, comparing its value with each card. Once you find the right position, you will insert the card there. Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too.

This is exactly how insertion sort works. It starts from the index 1(not 0), and each index starting from index 1 is like a new card, that you have to place at the right position in the sorted subarray on the left.

Following are some of the important characteristics of Insertion Sort:

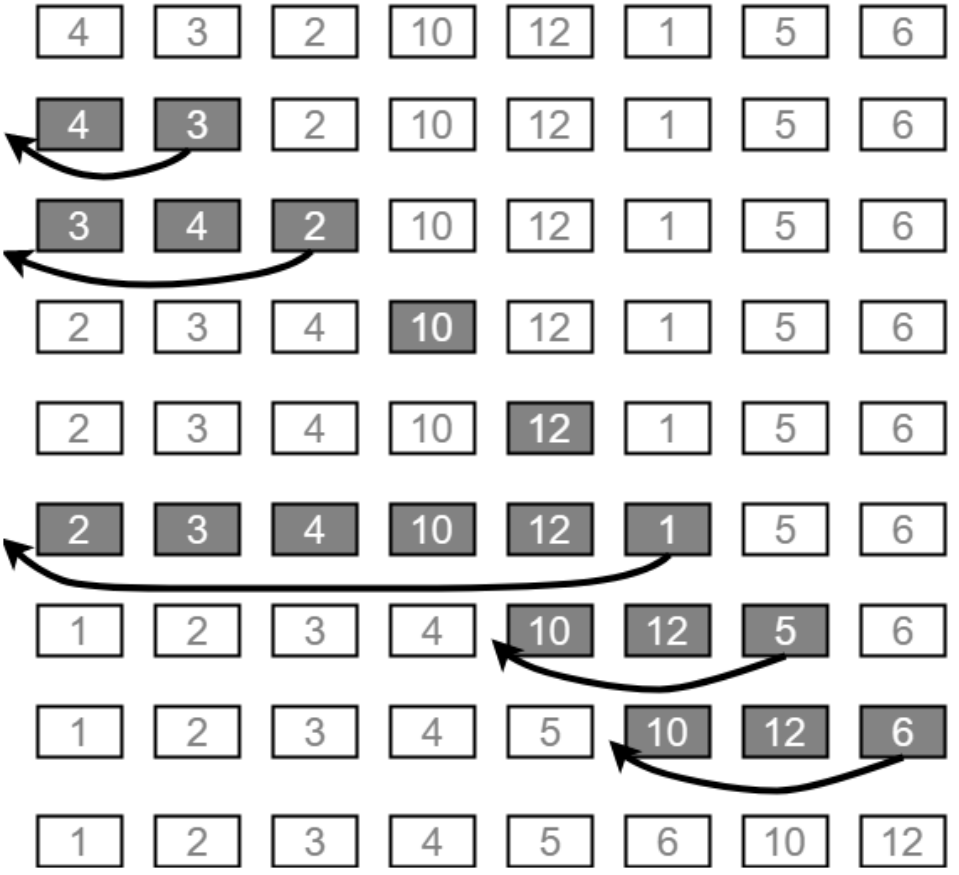
1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Selection Sort and Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
5. It is a stable sorting technique, as it does not change the relative order of elements which are equal.

Pseudo Code for insertion sort is given below:

```
for (i = 1; i < length; i++)
{
        j = i;
        while (j > 0 && arr[j - 1] > arr[j])
        {
                key = arr[j];
```

```
                arr[j] = arr[j - 1];
                arr[j - 1] = key;
                j--;
        }
}
```

Following is the trace of Insertion Sort.



Analysis

Bubble sort and Insertion sort have the same complexities in all cases.

$=(N -1) + (N-2) + ... + 2 + 1$

$=(N-1)*N/2$

$= O(N^2)$

1. Worst Case Time Complexity [Big-O]: $O(N^2)$
2. Best Case Time Complexity [Big-omega]: O(N)

---

3. Average Time Complexity [Big-theta]: $O(N^2)$

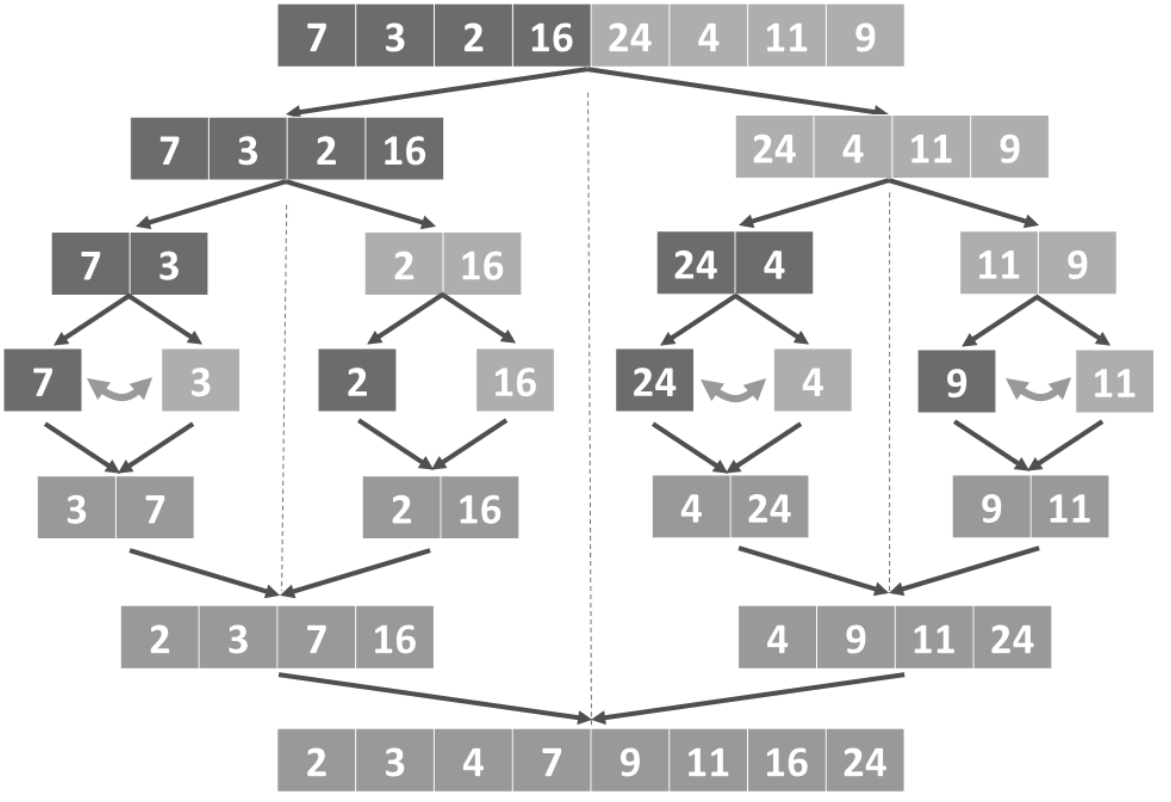
4.3.4 Merge Sort

Merge sort and is based on a **divide and conquer approach**. We know that merging of the two sorted arrays can be done so that resulting array will also be in the sorted form. If we decide to use the same technique for creating the sorted array, then the requirement will be the two arrays, which we are going to merge, must be sorted.

To sort the array, we will use the merging technique and that merging will again require the arrays to be sorted. But we can say that a single element will be always in the sorted form. Thus to implement merge sort we will first merge first element with second, generating a list of sorted element which contain two elements. The process is repeated for third & fourth, fifth & sixth etc.

A single element is always in the sorted form. Thus we will have sorted lists most of which contain two elements. If we merge first two, we will get a sorted list of four elements. Thus we will go on building the sorted array. At pass i, each vector contains 2^i elements except may be in the last vector.

Consider an example with Original list: 7, 3, 2, 16, 24, 4, 11, 9



Analysis

As we have already learned in Binary Search that whenever we divide a number into half in every step, it can be represented using a logarithmic function, which is log N and the number of steps can be represented by $\log_2 N + 1$(at most). Also, we perform a single step operation to find out the middle of any subarray, i.e. O(1). And to merge the subarrays, made by dividing the original array of n elements, a running time of O(n) will be required. Hence the total time for mergeSort function will become $N(\log_2 N + 1)$, which gives us a time complexity of $O(N*\log_2 N)$.

1. Worst Case Time Complexity [Big-O]: $O(N*\log_2 N)$
2. Best Case Time Complexity [Big-omega]: $O(N*\log_2 N)$
3. Average Time Complexity [Big-theta]: $O(N*\log_2 N)$

4.3.5 Quick Sort

Quick Sort is another sorting technique which is based on Divide and Conquer, just like merge sort. But in quick sort all the heavy lifting(major work) is done while dividing the array into subarrays, while in case of merge sort, all the real work happens during merging the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called partition-exchange sort. This algorithm divides the list into three main parts:

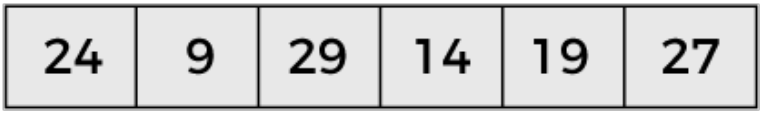
- Elements less than the Pivot element
- Pivot element(Central element)
- Elements greater than the pivot element

The steps of the quick sort are given below:

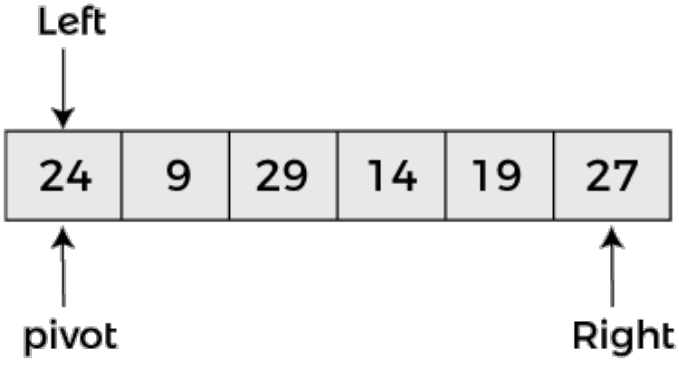
1. *Select the pivot element. (Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this, we will take the first element or the left most element as pivot).*
2. *After selecting an element as pivot, which is the first element of the array in our case, traverse from right to left up to pivot element position and swap pivot with whichever first element is found smaller than the pivot.*
3. *Now, after swapping, change the direction of traversing from left to right up to pivot element position and swap pivot element with whichever first element is found greater than the pivot.*
4. *This changing of direction continued till we do not get any element for swapping in left to right or right to left traverse.*
5. *When this situation arises, it means we have to partition the array into two subarrays. This partitioning is done in such a way so that all the elements smaller than the pivot will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of pivot.*
6. *The pivot element will be at its final sorted position. The elements to the left and right, may not be sorted.*
7. *Now, pick the subarrays i.e. elements on the left of pivot and elements on the right of pivot and repeat the steps 2 to 6 till all the subarrays are with single element.*

8. *Merge all the sub arrays and the sorted array is generated.*

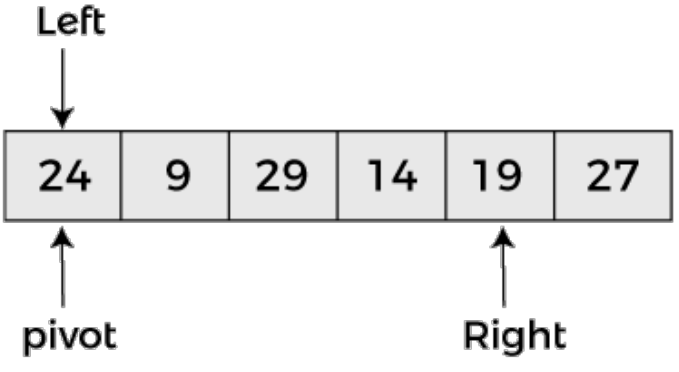
Now, let's see the working of the Quicksort Algorithm with an example. Let the elements of array are:



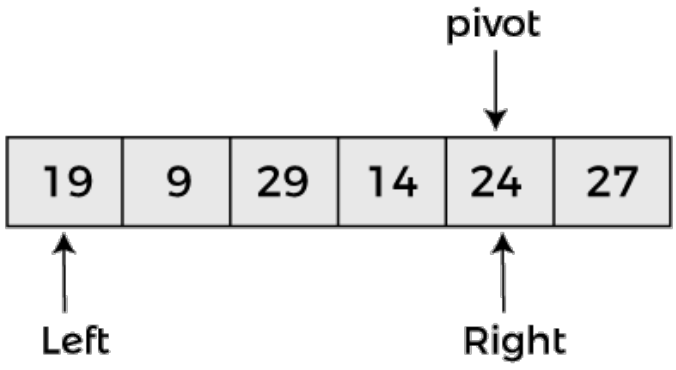
In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24. Since, pivot is at left, so algorithm starts from right and move towards left.



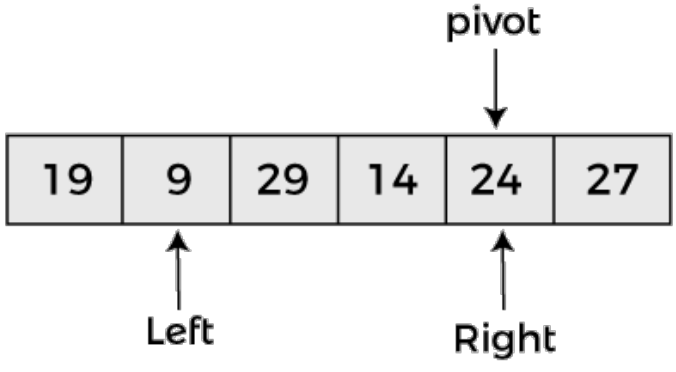
Now, a[pivot] < a[right], so algorithm moves forward one position towards left as below:



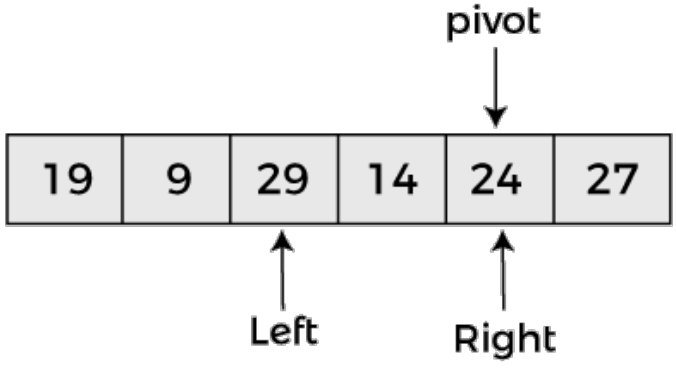
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24. Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as below:



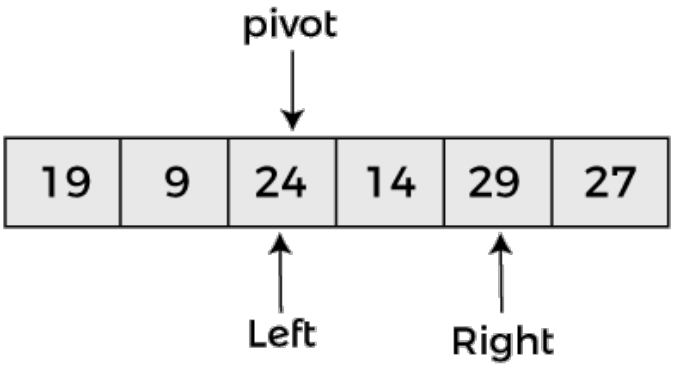
Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right. As a[pivot] > a[left], so algorithm moves one position to right as below:



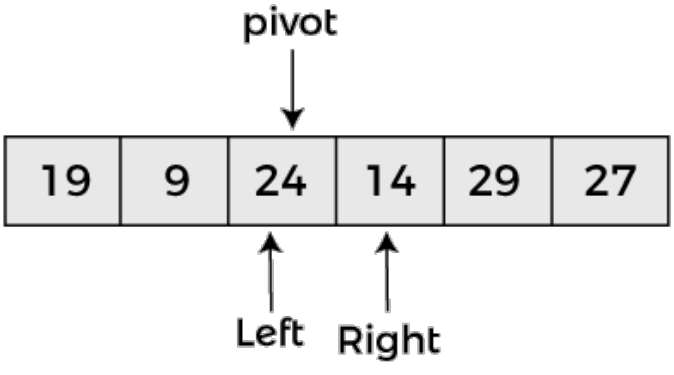
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as below:



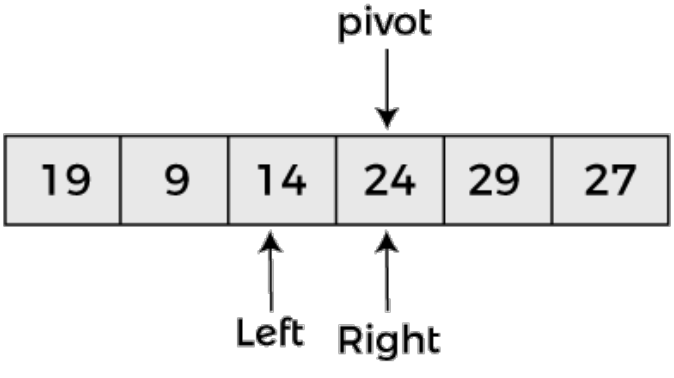
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left as given below:



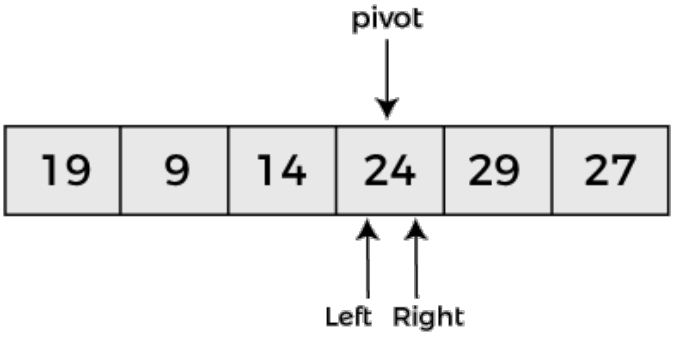
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as below:



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right as given below:

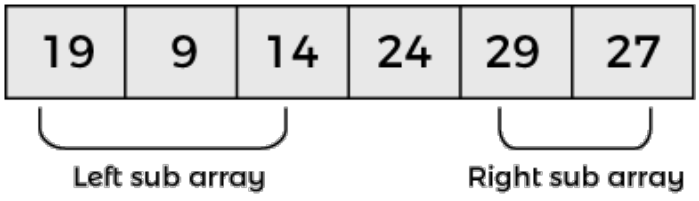


Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.

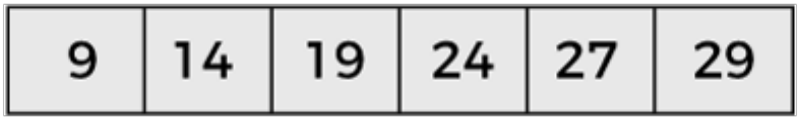


Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure. Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After the repeated processes the sorted array will be



Analysis

For an array, in which partitioning leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the pivot, hence on the right side. If we keep on getting unbalanced subarrays, then the running time is the worst case, which is $O(N^2)$. Where as if partitioning leads to almost equal subarrays, then the running time is the best, with time complexity as $O(N*\log_2 N)$.

1. Worst Case Time Complexity [Big-O]: $O(N^2)$
2. Best Case Time Complexity [Big-omega]: $O(N*\log_2 N)$
3. Average Time Complexity [Big-theta]: $O(N*\log_2 N)$

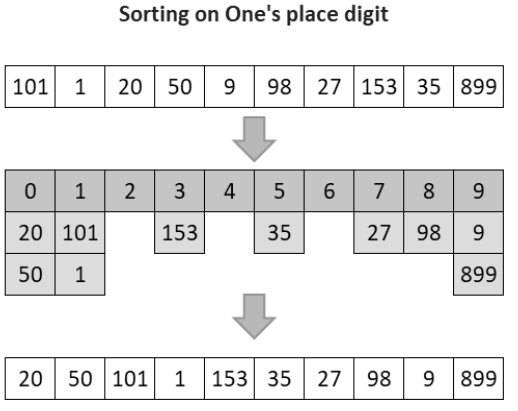
4.3.6 Radix Sort

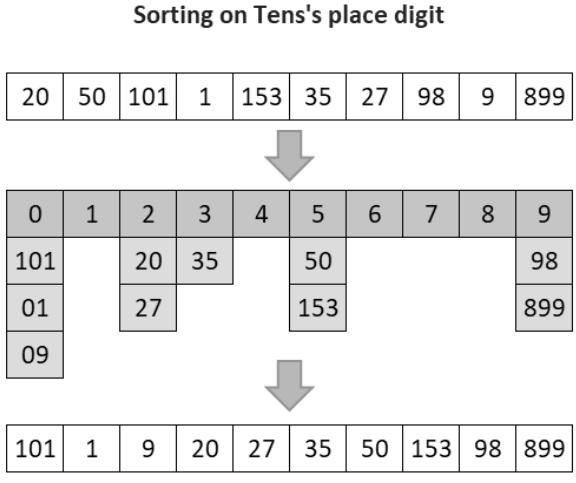
This is a formal algorithm predating computers. This was first devised for punched cards but is very efficient for sorting linked lists. The sorter used to place all cards containing a given digit in an appropriate pocket. There are 10 pockets for 10 digits. If character strings are to be used we need 26 pockets. Let us explore this method with an example. Radix sort is based on the idea that the sorting of the input data is done digit by digit from least significant digit to most significant digit and it uses counting sort as a subroutine to perform sorting.

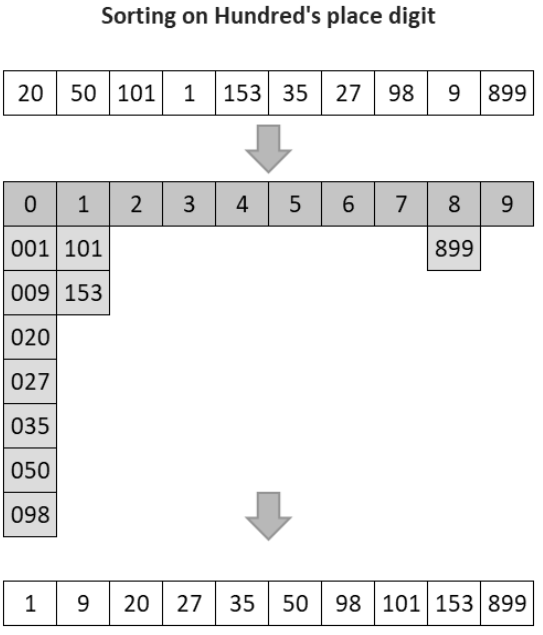
Counting sort is a linear sorting algorithm with overall time complexity O*(N+K)* in all cases, where *N* is the number of elements in the unsorted array and *K* is the range of input data. The idea of radix sort is to extend the counting sort algorithm to get a better time complexity when *K* goes up.

Consider numbers: 101, 1, 20, 50, 9, 98, 27, 153, 35, 899

The following is the trace of Radix sort for the given set of numbers:

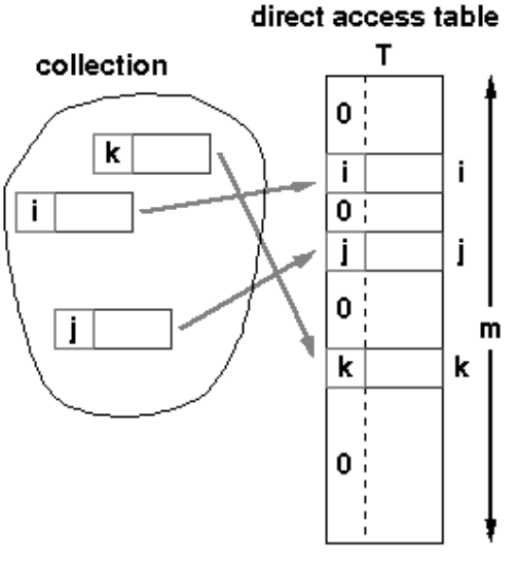






4.4 HASHING (DIRECT ADDRESS TABLES)

Hashing is the process of transforming any given key or a string of characters into another value. This is usually represented by a shorter, fixed-length value or key that represents and makes it easier to find or employ the original string. The most popular use for hashing is the implementation of hash tables.



If we have a collection of **n** elements whose keys are unique integers in (1,**m**), where **m** >=

n, then we can store the items in a *direct address* table, **T[m]**, where $\mathbf{T_i}$ is either empty or contains one of the elements of our collection.

Searching a direct address table is clearly an **O(1)** operation:

for a key, **k**, we access $\mathbf{T_k}$,

- if it contains an element, return it,
- if it doesn't then return a NULL.

There are two constraints here:

1. the keys must be unique, and
2. the range of the key must be severely bounded.

If the keys are not unique, then we can simply construct a set of **m** lists and store the heads of these lists in the direct address table. The time to find an element matching an input key will still be **O(1)**.

However, if each element of the collection has some other distinguishing feature (other than its key), and if the maximum number of duplicates is $\mathbf{n_{dup}^{max}}$, then searching for a specific element is $\mathbf{O(n_{dup}^{max})}$. If duplicates are the exception rather than the rule, then $\mathbf{n_{dup}^{max}}$ is much smaller than **n** and a direct address table will provide good performance. But if $\mathbf{n_{dup}^{max}}$ approaches **n**, then the time to find a specific element is **O(n)** and a tree structure will be more efficient.

The range of the key determines the size of the direct address table and may be too large to be practical. For instance it's not likely that you'll be able to use a direct address table to store elements which have arbitrary 32-bit integers as their keys for a few years yet!

Direct addressing is easily generalised to the case where there is a function, **h(k)** => (1,**m**) which maps each value of the key, **k**, to the range (1,**m**). In this case, we place the element in **T[h(k)]** rather than **T[k]** and we can search in **O(1)** time as before.

4.4.1 Mapping functions

The direct address approach requires that the function, h(k), is a one-to-one mapping from each k to integers in (1,m). Such a function is known as a perfect hashing function: it maps each key to a distinct integer within some manageable range and enables us to trivially build an O(1) search time table.

Unfortunately, finding a perfect hashing function is not always possible. Let's say that we can find a hash function, h(k), which maps most of the keys onto unique integers, but maps a small number of keys on to the same integer. If the number of collisions (cases where multiple keys map onto the same integer), is sufficiently small, then hash tables work quite well and give O(1) search times.

Division Method

The hash function depends upon the remainder of division. Typically the divisor is table length. For e.g. If the record 54, 72, 89, 37 is placed in the hash table and if the table size is 10 then,

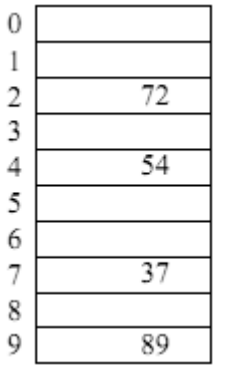
H(key) = record % table size

54%10=4

72%10=2

89%10=9

37%10=7



Mid Square

In the mid square method, the key is squared and the middle or mid part of the result is used as the index. If the key is a string, it has to be preprocessed to produce a number. Consider that if we want to place a record 3111 then,

$$3111^2 = 9678321$$

for the hash table of size 1000, H(3111) = 783 (the middle 3 digits)

Multiplicative hash function

The given record is multiplied by some constant value. The formula for computing the hash key is,

H(key) = floor(p*(fractional part of key*A))

where p is integer constant and A is constant real number.

Donald Knuth suggested to use constant A = 0.61803398987

So, if key 107 and p=50 then,

H(key) = floor(50*(107*0.61803398987))

= floor(3306.4818458045)

= 3306

At 3306 location in the hash table the record 107 will be placed.

Digit Folding

The key is divided into separate parts and using some simple operation these parts are combined to produce the hash key. For e.g. consider a record 12365412 then it is divided into separate parts as 123 654 12 and these are added together.

H(key) = 123+654+12

= 789

The record will be placed at location 789

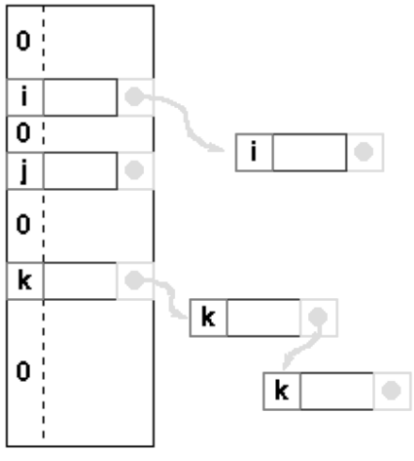
Digit Analysis

The digit analysis is used in a situation when all the identifiers are known in advance. We first transform the identifiers into numbers using some radix, r. Then examine the digits of each identifier. Some digits having most skewed distributions are deleted. This deleting of digits is continued until the number of remaining digits is small enough to give an address in the range of the hash table. Then these digits are used to calculate the hash address.

4.4.2 Handling the collisions

In the small number of cases, where multiple keys map to the same integer, then elements with different keys may be stored in the same "slot" of the hash table. It is clear that when the hash function is used to locate a potential match, it will be necessary to compare the key of that element with the search key. But there may be more than one element which should be stored in a single slot of the table. Various techniques are used to manage this problem are explained below:

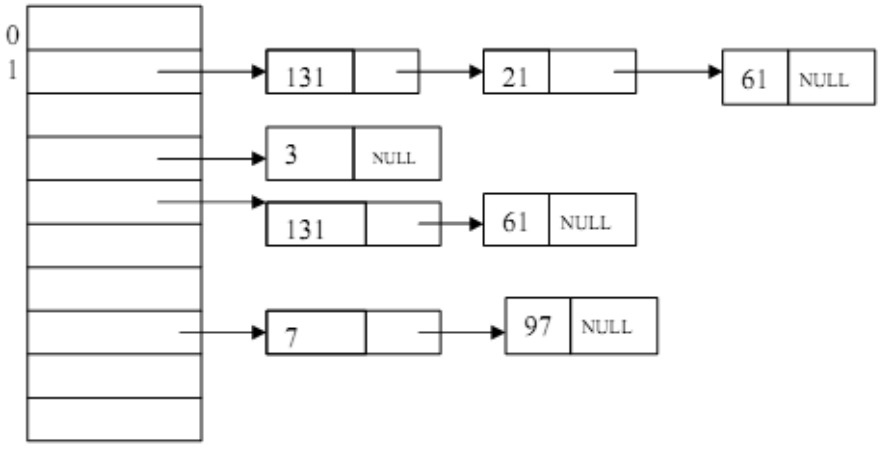
Chaining



One simple scheme is to chain all collisions in lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and doesn't require a priori knowledge of how many elements are contained in the collection. The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.

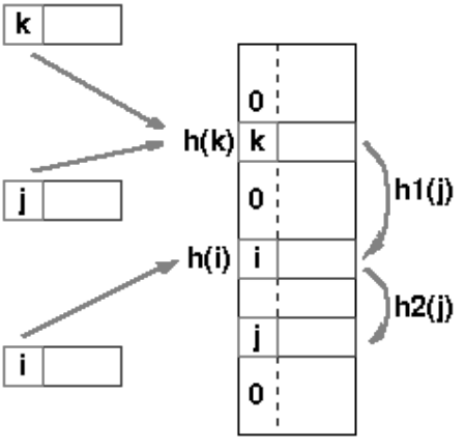
For e.g. Consider the keys to be placed in their home buckets are 131, 3, 4, 21, 61, 7, 97, 8, 9.

Then we will apply a hash function as H(key) = key % D where D is the size of table. Here, D = 10



A chain is maintained for colliding elements. for instance 131 has a home bucket (key) 1. similarly key 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1.

Re-hashing



Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable is the total size of table is a prime number. There are situations in which the rehashing is required.

- When table is completely full
- With quadratic probing when the table is f
- When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by re computing their positions using hash functions. Consider we have to insert the elements

37, 90, 55, 22, 17, 49, and 87. the table size is 10 and will use hash function.,

H(key) = key % tablesize

37 % 10 = 7

90 % 10= 0

55 % 10 = 5

22 % 10 = 2

17 % 10 = 7 Collision solved by linear probing

49 % 10 = 9

Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size. The old table size is 10 then we should double this size for new table, that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. So, new hash function will be

H(key) = key % 23

37 % 23 = 14

90 % 23 = 21

55 % 23 = 9

22 % 23 = 22

17 % 23 = 17

49 % 23 = 3

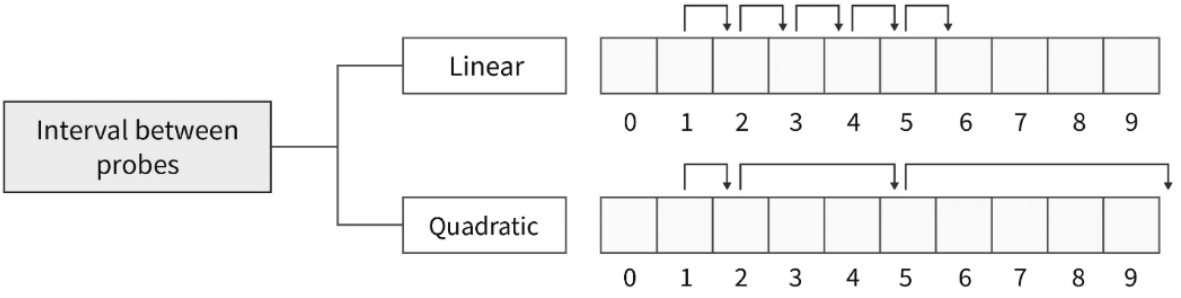
87 % 23 = 18

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	

Now the hash table is sufficiently large to accommodate new insertions.

Linear probing (using neighbouring slots)

One of the simplest re-hashing functions is +1 (or -1), ie on a collision, look in the neighbouring slot in the table. It calculates the new address extremely quickly and may be extremely efficient on a modern RISC processor due to efficient cache utilisation (cf. the discussion of linked list efficiency).



Linear probing is subject to a clustering phenomenon. Re-hashes from one location occupy a block of slots in the table which "grows" towards slots to which other keys hash. This exacerbates the collision problem and the number of re-hashed can become large.

Consider that following keys are to be inserted in the hash table:

131, 4, 8, 7, 21, 5, 31, 61, 9, 29

Initially, we will put the following keys in the hash table. We will use Division hash function. That means the keys are placed using the formula,

H(key) = key % tablesize

H(key) = key % 10

For instance the element 131 can be placed at,

H(key) = 131 % 10

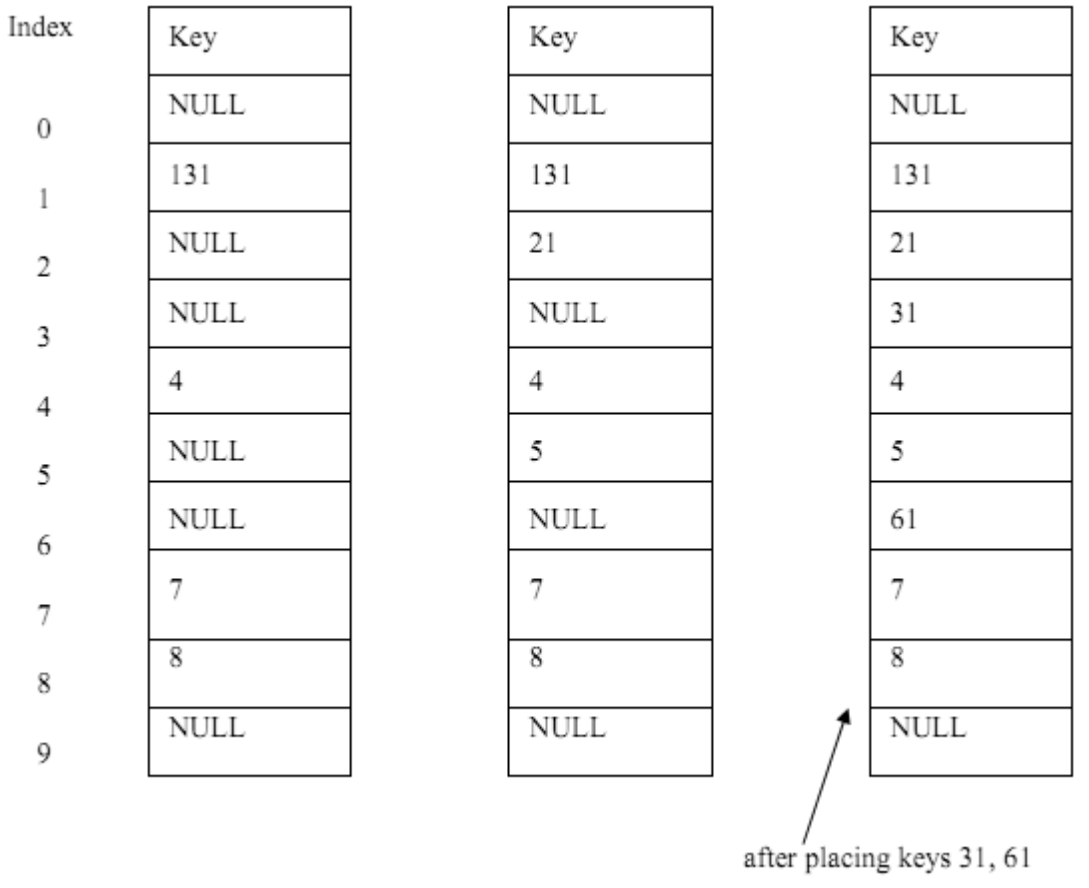
= 1

Index 1 will be the home bucket for 131. Continuing in this fashion we will place 4, 8, 7. Now the next key to be inserted is 21. According to the hash function

H(key)=21%10

H(key) = 1

But the index 1 location is already occupied by 131 i.e. collision occurs. To resolve this collision, we will linearly move down and at the next empty location we will probe the element. Therefore, 21 will be placed at the index 2. If the next element is 5 then we get the home bucket for 5 as index 5 and this bucket is empty so we will put the element 5 at index 5.



The next record key is 9. According to decision hash function it demands for the home bucket 9. Hence we will place 9 at index 9. Now the next final record key 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs. To handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0^{th} index. One major problem with linear probing is primary clustering. Primary clustering is a process in which a block of data is formed in the hash table when collision is resolved.

Quadratic Probing

Better behaviour is usually obtained with quadratic probing, where the secondary hash function depends on the re-hash index:

address = (Hash(key) + i^2) % m) where m can be table size or any prime number.

for eg; If we have to insert following elements in the hash table with table size 10:

37, 90, 55, 22, 17, 49, 87

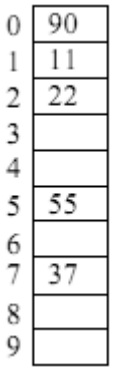
37 % 10 = 7

90 % 10 = 0

55 % 10 = 5

22 % 10 = 2

11 % 10 = 1



Now if we want to place 17 a collision will occur as 17%10 = 7 and bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

Hi (key) = (Hash(key) + i^2) % m

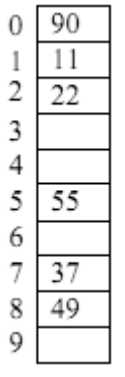
Consider i = 0 then

(17 + 0^2) % 10 = 7

(17 + 1^2) % 10 = 8, when i =1

The bucket 8 is empty hence we will place the element at index 8. Then comes 49 which will be placed at index 9.

49 % 10 = 9



Now to place 87 we will use quadratic probing.

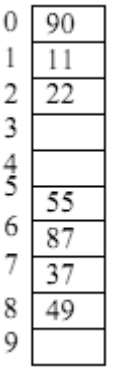
(87 + 0) % 10 = 7

(87 + 1) % 10 = 8… but already occupied

(87 + 2^2) % 10 = 1... already occupied

(87 + 3^2) % 10 = 6

It is observed that if we want place all the necessary elements in the hash table the size of divisor (m) should be twice as large as total number of elements.



However, the collision elements are stored in slots to which other key values map directly, thus the potential for multiple collisions increases as the table becomes full.

Double Hashing

Double hashing is technique in which a second hash function is applied to the key when a collision occurs. By applying the second hash function we will get the number of positions from the point of collision to insert. There are two important rules to be followed for the second function:

- It must never evaluate to zero.
- It must make sure that all cells can be probed.

The formula to be used for double hashing is

H1(key) = key % tableSize

H2(key) = M – (key mod M)

where M is a prime number smaller than the size of the table.

Consider the following elements to be placed in the hash table of size 10.

37, 90, 45, 22, 17, 49, 55

Initially insert the elements using the formula for H1(key).

Insert 37, 90, 45, 22

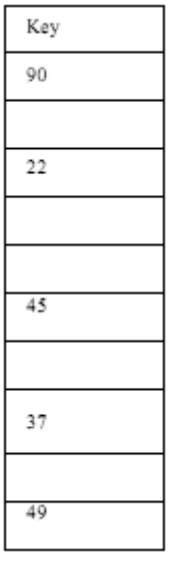
H1(37) = 37 % 10 = 7

H1(90) = 90 % 10 = 0

H1(45) = 45 % 10 = 5

H1(22) = 22 % 10 = 2

H1(49) = 49 % 10 = 9



Now if 17 to be inserted then

H1(17) = 17 % 10 = 7

H2(key) = M – (key % M)

Here, M is prime number smaller than the size of the table. Prime number smaller than table size 10 is 7.

Hence, M = 7

H2(17) = 7-(17 % 7)

= 7 – 3 = 4

That means we have to insert the element 17 at 4 places from 37. In short we have jumps. Therefore the 17 will be placed at index 1.

Key
90
17
22
45
37
49

Now to insert number 55

H1(55) = 55 % 10 =5 Collision

H2(55) = 7-(55 % 7)

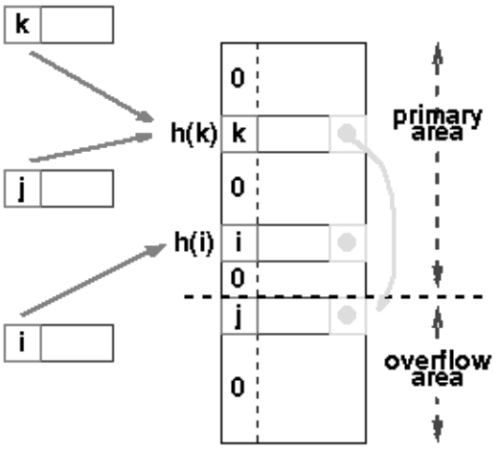
= 7 – 6 = 1

That means we have to take one jump from index 5 to place 55. Finally the hash table will be

Key
90
17
22
45
55
37
49

Overflow area

Another scheme will divide the pre-allocated table into two sections: the primary area to which keys are mapped and an area for collisions, normally termed the overflow area.



When a collision occurs, a slot in the overflow area is used for the new element and a link from the primary slot established as in a chained system. This is essentially the same as chaining, except that the overflow area is pre-allocated and thus possibly faster to access. As with re-hashing, the maximum number of elements must be known in advance, but in this case, two parameters must be estimated: the optimum size of the primary and overflow areas.

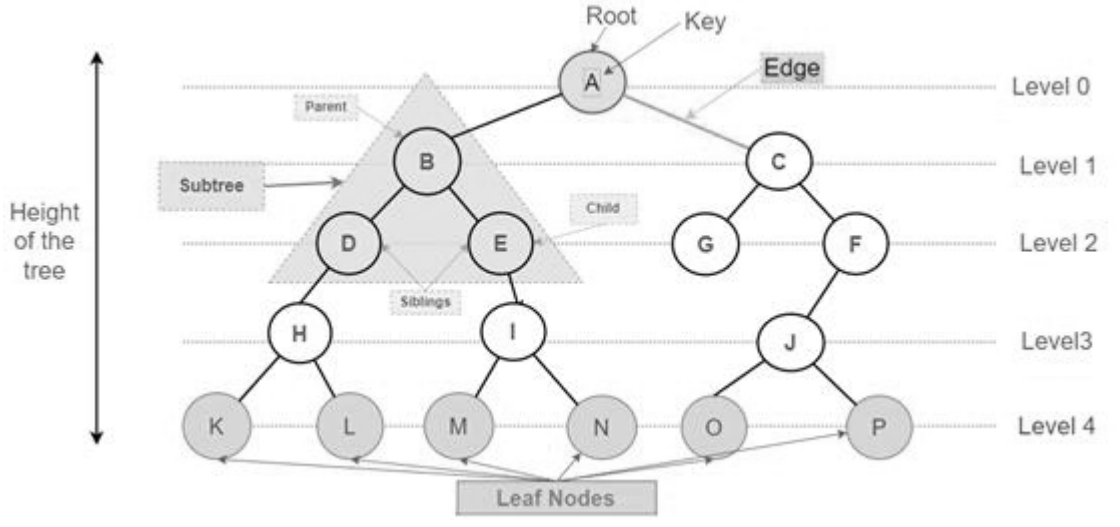
Of course, it is possible to design systems with multiple overflow tables, or with a mechanism for handling overflow out of the overflow area, which provide flexibility without losing the advantages of the overflow scheme.

5. TREES

5.1 INTRODUCTION

Until now we have seen the data structures, which were basically connected linearly. But many times we are required to have two more paths from the correct 'object'. It may not be always possible for a linear relationship between values stored. There may be multiple connections among the values (objects) stored. This introduced the development of non-linear data structures. Tree and graph are the non-linear data structures. These data structures are the most widely used in computer science.

A Tree is a data structure in which each element is attached to one or more elements directly beneath it. In other words, Tree is collection of nodes (or) vertices and their edges (or) links. In tree data structure, every individual element is called as Node. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.

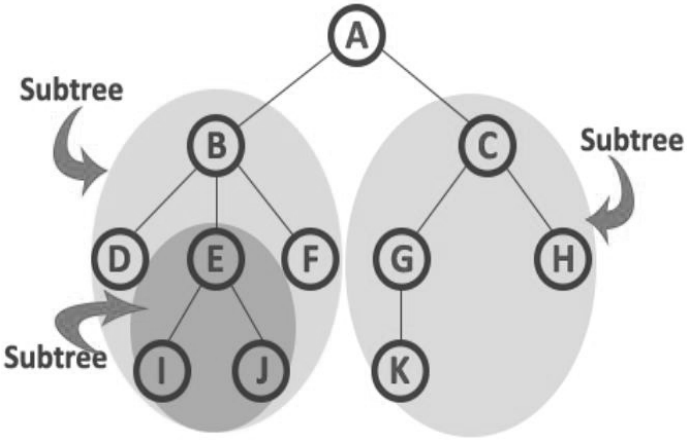


From the figure given above, let us understand tree and its terminologies.

1. Root: This is the unique node in the tree to which further sub trees are attached. e.g. A
2. Edge: In a Tree, the connecting link between any two nodes is called as Edge. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges. It is also called branch.
3. Degree of the node: The Degree of a node is total number of children that node has. The highest degree of a node among all the nodes in a tree is called as Degree of Tree. e.g. For node A degree is 2, for node K degree is 0, for node F degree is 1.
4. Leaves: These are the terminal nodes of the tree. A node which does not have any child called leaf or terminal node. In other words, the nodes with degree 0 are always the leaf nodes. Eg: K, L, M, N, G, O, P
5. Internal nodes: The nodes other than the root and the leaf nodes are called the internal nodes. e.g. B, C, D, E, F, H, I, J
6. Parent nodes: The node which is having further suB trees(branches) is called the parent

node of those suB trees. A, B, C, D, E, F, H, I, J are parent nodes.

7. Child nodes: In a Tree data structure, the node which is descendant of any node is called as Child Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.
8. Predecessor: While displaying the tree, if some particular node occurs previous to some other node then that node is called the predecessor of the other node. e.g. B is the predecessor of the node D and E, C is the predecessor of the node G and F, A is the predecessor of the node B and C, H is predecessor of K and L and so on.
9. Successor: The node which occurs next to some other node is a successor node. e.g. D and E are the successor of B, G and F are the successor of C, O and P are the successor of J and so on.
10. Level of the tree: In a Tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on. In simple words, in a tree each edge from top to bottom is called as a Level. The Level count starts with '0' and incremented by one at each level. e.g. A is at level 0. B and C are at level 1. D, E, F, G are at level 2. H, I, J are at level 3 and K, L, M, N, O, P are at level 4.
11. Height of the tree: The maximum level is the height of the tree. Here, height of the tree is 4. In a tree, height of the root node is said to be height of the tree. The height of all leaf nodes is '0'. Sometimes, the height is also referred as depth also.
12. Path: The sequence of Nodes and Edges from one node to another node is called as Path between that two Nodes. Length of a Path is total number of nodes in that path. In our tree the path A-B-E-I-M has length 5, the path A-C-G has length 3.
13. Sub Tree: Each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



However, the tree that is shown above is a general tree which is, implementation point of view, difficult to create and maintain. Hence, the binary tree was defined and created. General tree has been discussed in 5.2.3.

5.2 BINARY TREE

Binary tree is a tree in which each node has at most two children, a left child and a right child. Thus the order of binary tree is 2. A binary tree is either empty or consists of

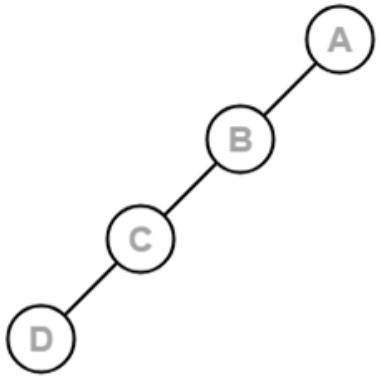
1. a node called the root
2. left and right sub trees are themselves binary trees.

To be more specific, A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint trees called left suB tree and right suB tree.

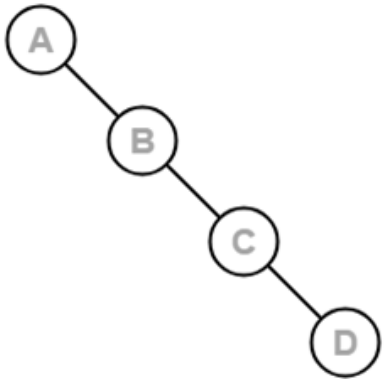
5.2.1 Types of Binary Trees

A full binary tree, or proper binary tree, is a tree in which every node has zero or two children.

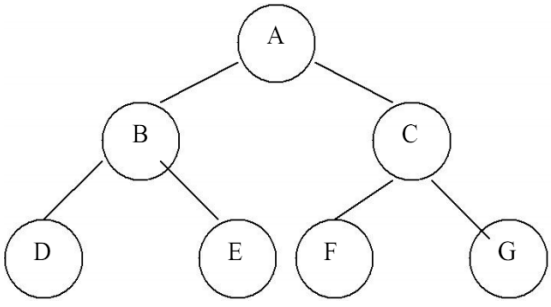
1. Left skewed binary tree: If the right suB tree is missing in every node of a tree we call it as left skewed tree.



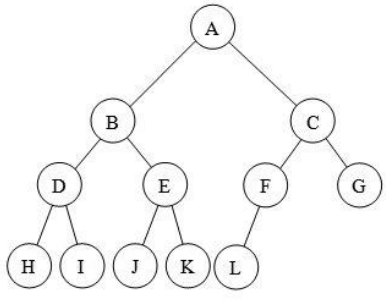
2. Right skewed binary tree: If the left suB tree is missing in every node of a tree we call it is right suB tree.



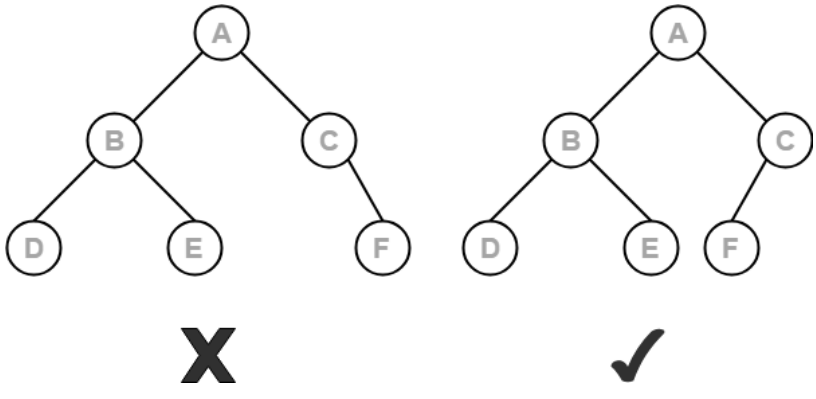
3. A perfect binary tree (sometimes complete binary tree) is a full binary tree in which all leaves are at the same depth.



4. A complete binary tree may also be defined as a full binary tree in which all leaves are at depth n or n-1 for some n. In order for a tree to be the latter kind of complete binary tree, all the children on the last level must occupy the leftmost spots consecutively, with no spot left unoccupied in between any two. For example, if two nodes on the bottommost level each occupy a spot with an empty spot between the two of them, but the rest of the children nodes are tightly wedged together with no spots in between, then the tree cannot be a complete binary tree due to the empty spot.

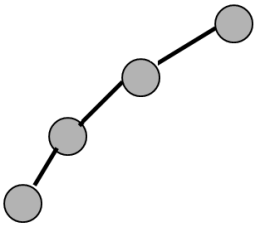


5. An almost complete binary tree is a tree in which each node that has a right child also has a left child. Having a left child does not require a node to have a right child. Stated alternately, an almost complete binary tree is a tree where for a right child, there is always a left child, but for a left child there may not be a right child.

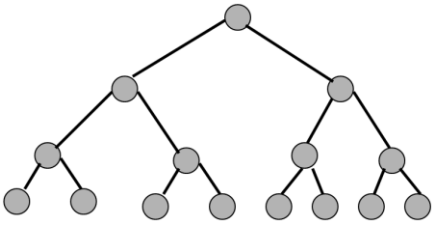


5.2.2 Binary Trees Properties and Representation

1. Minimum number of nodes in a binary tree whose height is h are h+1.



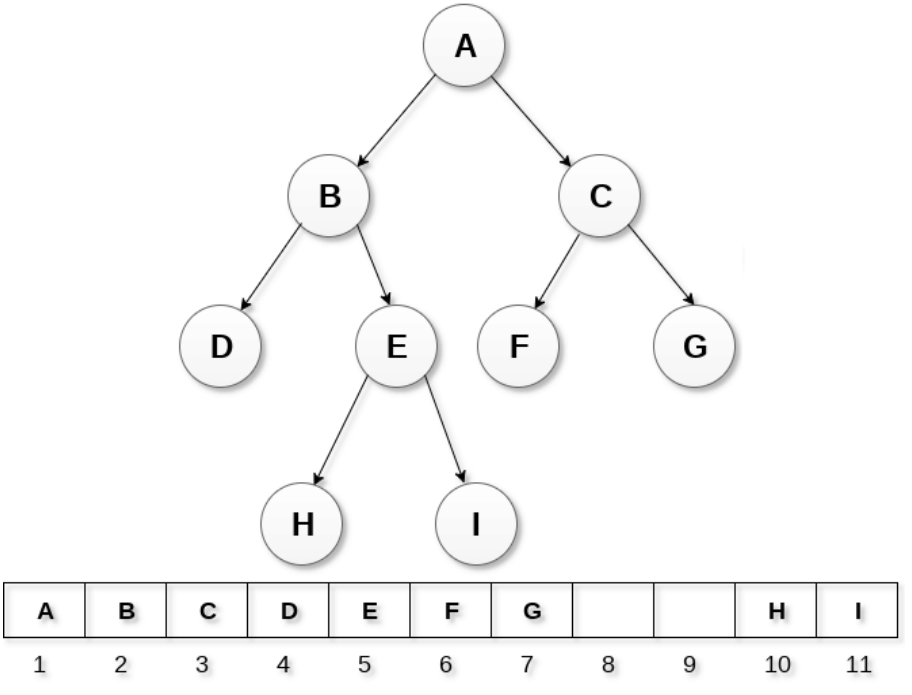
2. Maximum number of nodes in a binary tree whose height is h are $n = 2^{h+1} - 1$ where h is the height of the tree.



Maximum number of nodes $= 1 + 2 + 4 + 8 + \ldots + 2^{h-1}$

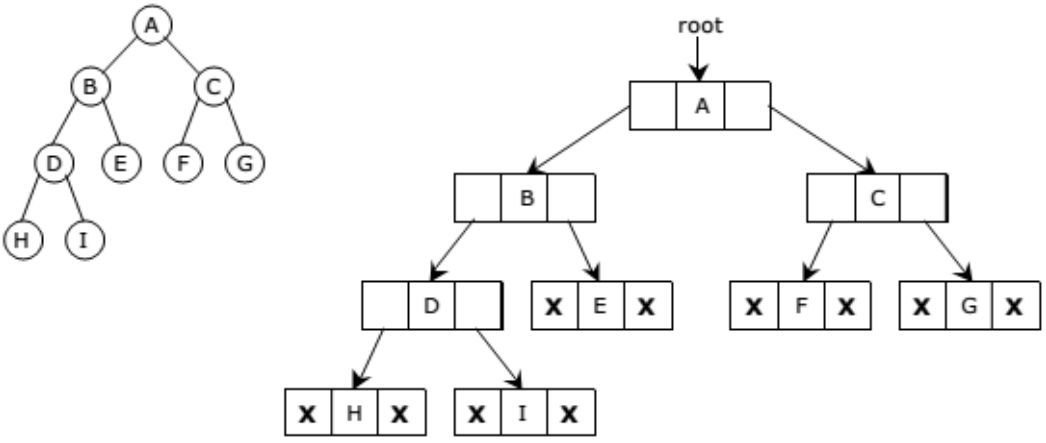
$= 2^{h+1} - 1$

Binary trees are represented in the form of linear array or linked list.



The above figure shows array representation of the tree. To represent a binary tree of height h using array representation, we need one dimensional array with a maximum size of 2^{h+1}.

We use a doubly linked list to represent a binary tree. Recall that, in a doubly linked list, every node consists of three fields, two address pointers and one data. The following node structure shows each node of a tree. The left pointer contains the address of left child and the right pointer contains the address of right child. The middle data part contains the data value. The following figure shows linked list representation of the given tree.



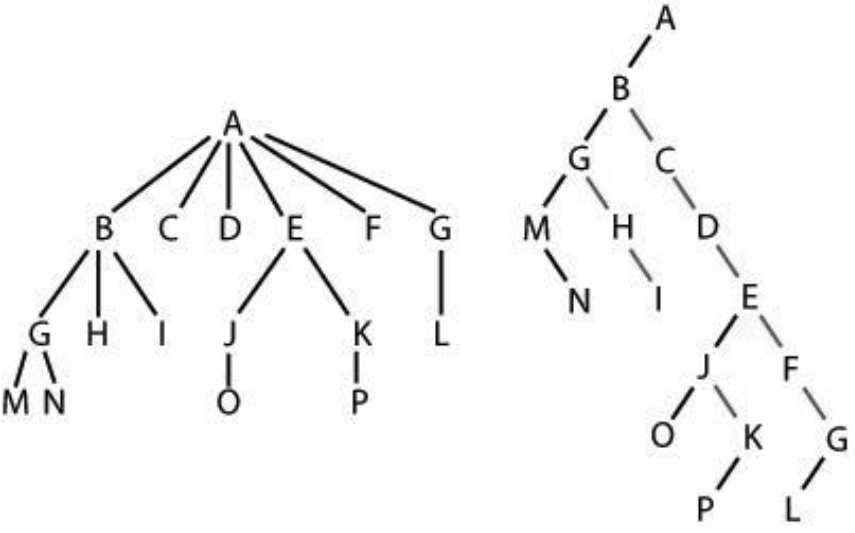
5.2.3 General Tree to Binary Tree Conversion

General trees are those in which the number of subtrees for any node is not required to be 0, 1, or 2. The tree may be highly structured and therefore have 3 subtrees per node in which case it is called a ternary tree. However, it is often the case that the number of subtrees for any node may be variable. Some nodes may have 1 or no subtrees, others may have 3, some 4, or any other combination. The ternary tree is just a special case of a general tree.

General trees can be represented as linked representation in whatever form they exist. However, there are some problems which are to be addressed. First, the number of references for each node are not certain so maximum size and number of pointers cannot be decided. It is also obvious that most of the algorithms for searching, traversing, adding and deleting nodes become much more complex in that they must now cope with situations where there are not just two possibilities for any node but multiple possibilities.

Fortunately, general trees can be converted to binary trees. Following algorithm shows steps to convert general tree into binary tree. General to binary tree will result into left skewed tree (left rooted tree)

1. *Use the root of the general tree as the root of the binary tree.*
2. *Determine the first child of the root. This is the leftmost node in the general tree at the next level.*
3. *Continue finding the first child of each parent node and insert it as left child below the parent node with the child reference of the parent to this node.*
4. *Put the sibling of the node as a right child of the node in binary tree*



5.2.4 Binary Tree Traversal

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, binary trees can be traversed in different ways. Following are the generally used ways for traversing binary trees. When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method. Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal. There are three types of binary tree traversals:

Pre-Order Traversal

In-Order Traversal

Post-Order Traversal

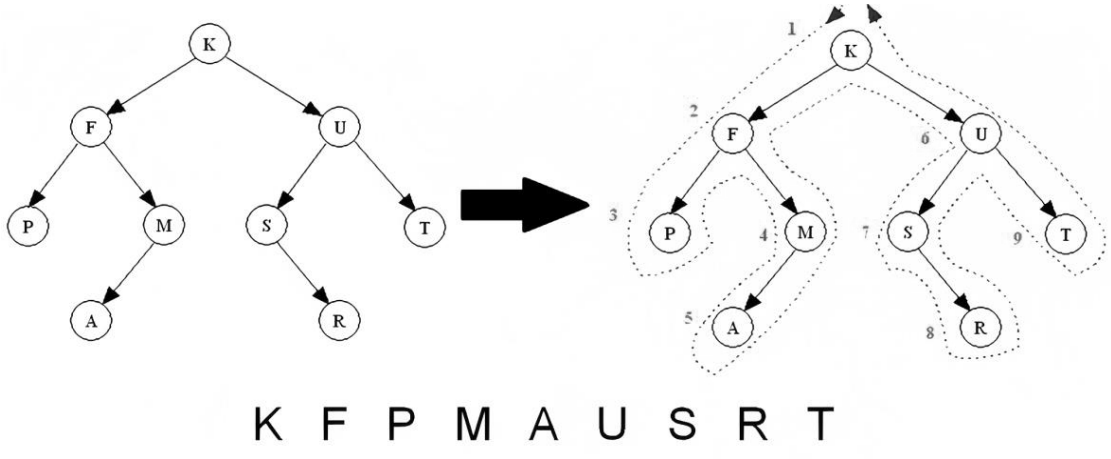
These traversals are recursive in working as for each subtree the process of visiting the node is repeated.

Pre-Order Traversal

In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree. Preorder search is also called backtracking.

Algorithm:

1. *Visit the root.*
2. *Visit the left subtree, using preorder.*
3. *Visit the right subtree, using preorder.*

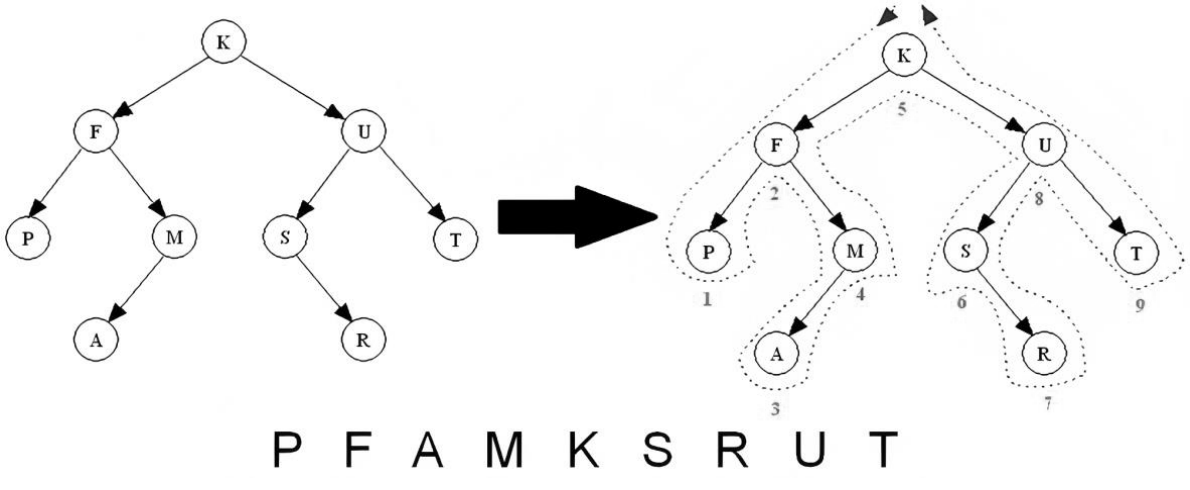


In-Order Traversal

In In-Order traversal, the root node is visited between the left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all sub trees in the tree. This is performed recursively for all nodes in the tree.

Algorithm:

1. *Visit the left subtree, using inorder.*
2. *Visit the root.*
3. *Visit the right subtree, using inorder.*



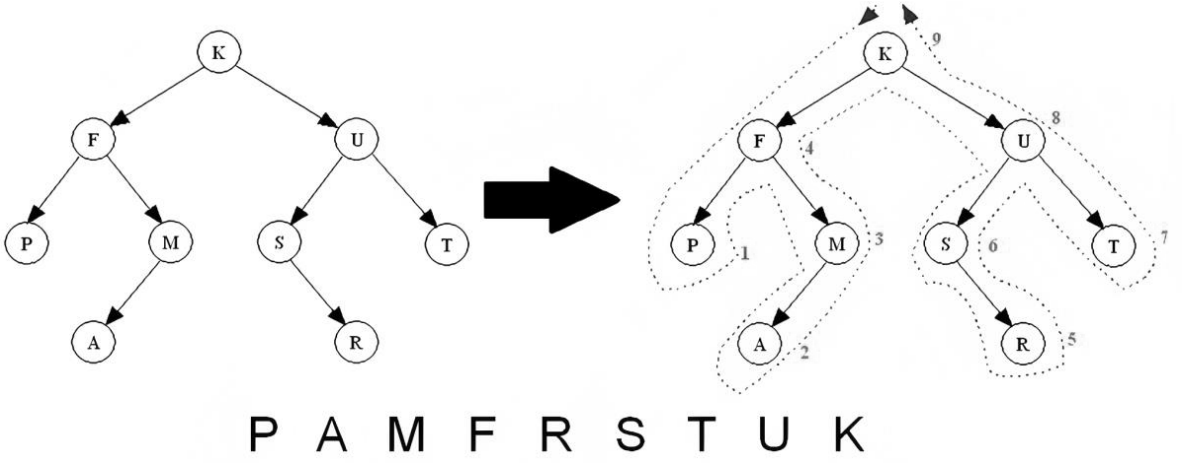
Post-Order Traversal

In Post-Order traversal, the root node is visited after left child and right child. In this

traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most nodes are visited.

Algorithm:

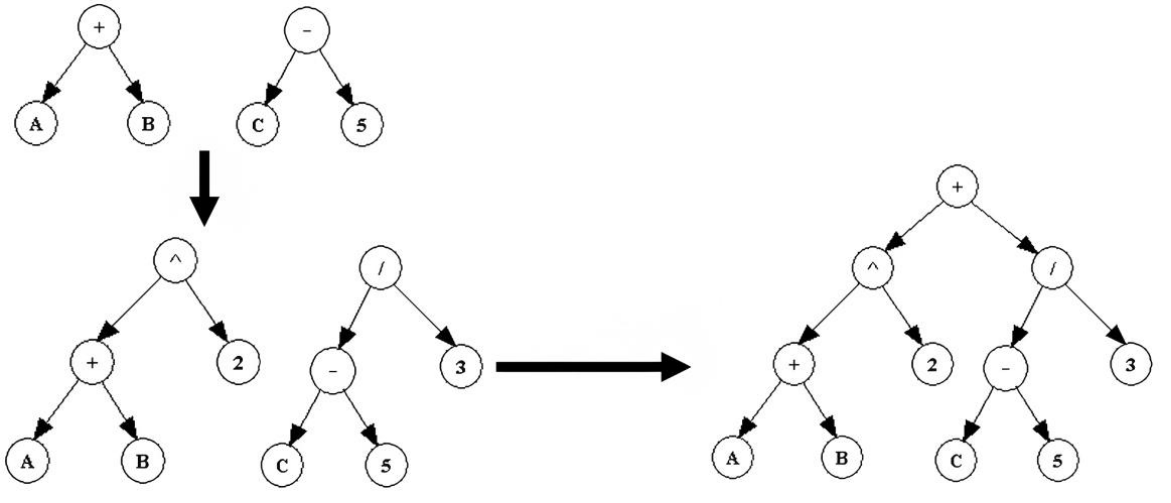
1. *Visit the left subtree, using postorder.*
2. *Visit the right subtree, using postorder*
3. *Visit the root.*



5.2.5 Application of Tree Traversal - Expression Trees

An arithmetic expression or a logic proposition can be represented by a Binary tree. In this, internal vertices represent operators, leaves represent operands and sub trees are sub expressions. A Binary tree representing an expression is called an expression tree.

Build the expression tree bottom-up for $(A + B)^2 + (C - 5) / 3$.



All traversals of expression tree will convert the expression into respective prefix, infix and

postfix expressions. Consider the above tree and performing pre, in and post order traversals will create the following expressions:

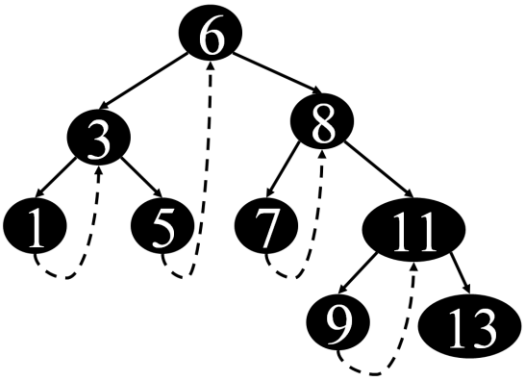
Infix: A + B ^ 2 + C – 5 / 3

Prefix: + ^ + A B 2 / - C 5 3

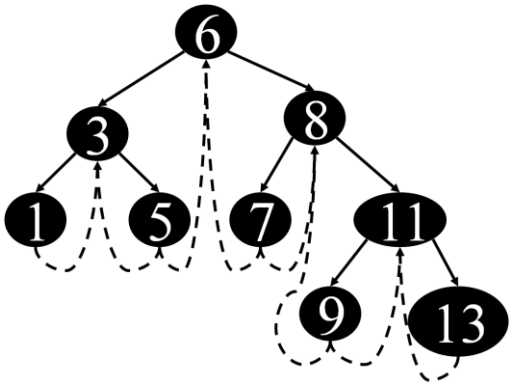
Postfix: A B + 2 ^ C 5 - 3 / +

5.2.6 Threaded Binary Trees

Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals. We can have the pointers reference the next node in an inorder traversal, called threads. We need to know if a pointer is an actual link or a thread, so we can keep a boolean for each pointer.



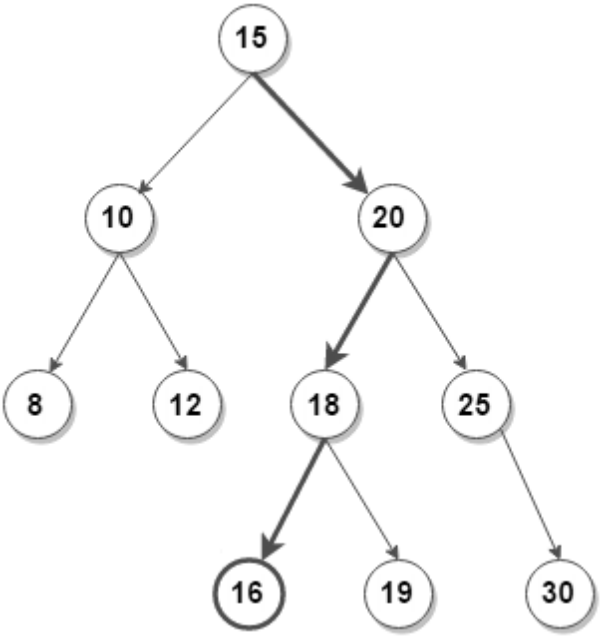
In some of the cases, as half of the leaves' pointers would be null, we can add threads to the previous node in an inorder traversal as well. These can be used to traverse the tree backwards or even to do postorder traversals.



5.3 BINARY SEARCH TREE (BST)

In the simple binary tree the nodes are arranged in any fashion. Depending on user's desire the new nodes can be attached as a left or right child of any desired node. In such a case finding for any node is a long cut procedure, because in that case we have to search the entire tree. Thus the searching time complexity will get increased unnecessarily. So to

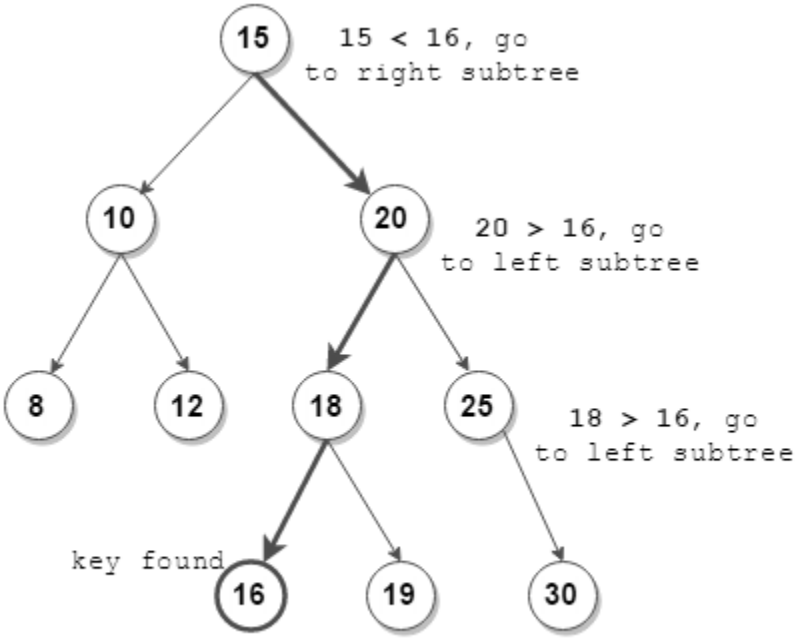
make the searching algorithm faster in a binary tree, the binary search tree (BST) was developed. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged in such a way so that values at left suB tree < root node value < right suB tree values. This data organization leads to O(log n) complexity for searches, insertions and deletions in certain types of the BST (balanced trees).



5.3.1 Searching in BST

In searching, the node which we want to search is called a key node. The key node will be compared with each node starting from root node if value of key node is greater than current node then we search for it on right sub branch otherwise on left sub branch. If we reach to leaf node and still we do not get the value of key node then we declare "node is not present in the tree". Following is the algorithm for Searching in BST.

```
if the tree is empty
        return NULL
else if  the item in the node equals the target
        return the node value
else if  the item in the node is greater than the target
        return the result of searching the left subtree
else if  the item in the node is smaller than the target
        return the result of searching the right subtree
```



Search (root, 16)

5.3.2 Insertion in BST

While inserting any node in binary search tree, look for its appropriate position in the binary search tree. We start comparing this new node with each node of the tree. If the value of the node which is to be inserted is greater than the value of the current node we move on to the right sub-branch otherwise we move on to the left sub-branch. As soon as the appropriate position is found we attach this new node as left or right child appropriately.

if tree is empty

create a root node with the new key

else

compare key with the top node

if ***key = node key***

replace the node with the new value

else if ***key > node key***

compare key with the right subtree:

if subtree is empty create a leaf node

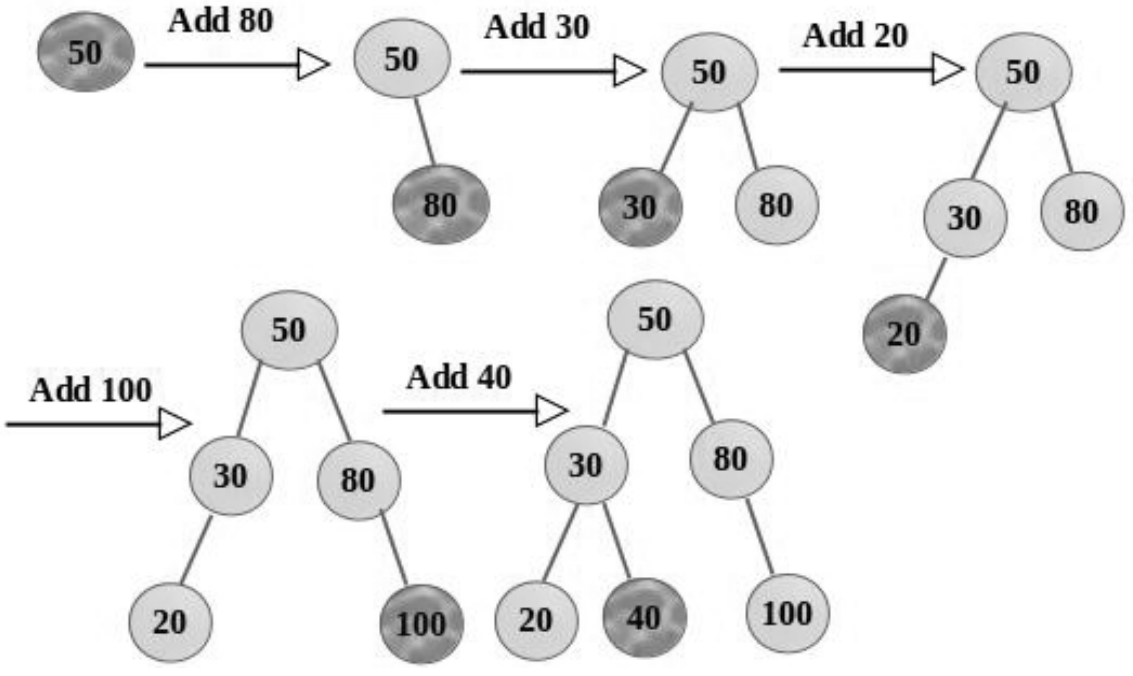
else add key in right subtree

else ***key < node key***

compare key with the left subtree:

if the subtree is empty create a leaf node

else add key to the left subtree



5.3.3 Deletion in BST

For deletion of any node from binary search tree there are three which are possible.

1. *Deletion of leaf node.*
2. *Deletion of a node having one child.*
3. *Deletion of a node having two children.*

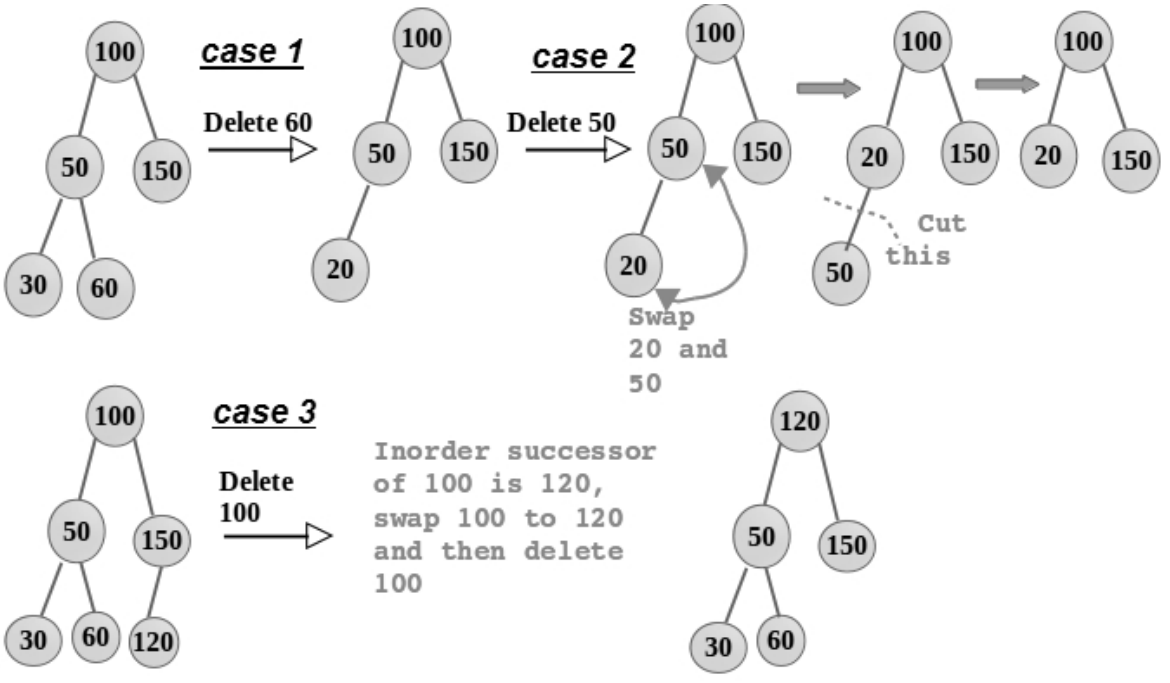
Case 1: If the node has 2 empty subtrees i.e. it is a lead node, replace the link in the parent with null

Case 2: If the node has single child:

1. *If the node has no left child, link the parent of the node to the right (non-empty) subtree.*
2. *if the node has no right child link the parent of the target to the left (non-empty)*

subtree.

Case 3: If the node has a left and a right subtree replace the node's value with the max value in the left subtree and delete the max node in the left subtree.



However, the BST can be degenerate, if the elements are added on either left or right side of the root node. A BST can be set up to maintain balance during updating operations (insertions and removals). Such trees are known as AVL trees, Red-Black trees, B trees, B+ Trees. The balanced trees ensure the depth so that the searching, insertion and deletion complexity remains O(log N). It also makes sure that every node must have left & right suB trees of the same height.

5.4 AVL TREE (HEIGHT BALANCED TREE)

AVL trees are Binary Search Trees generated using height balance factor. They are named after Adelson, Velskii and Landis who developed it. It was the first dynamically balanced Binary Search tree developed. It is also called height balanced tree as the height of suB trees of each node can differ by at most 1.

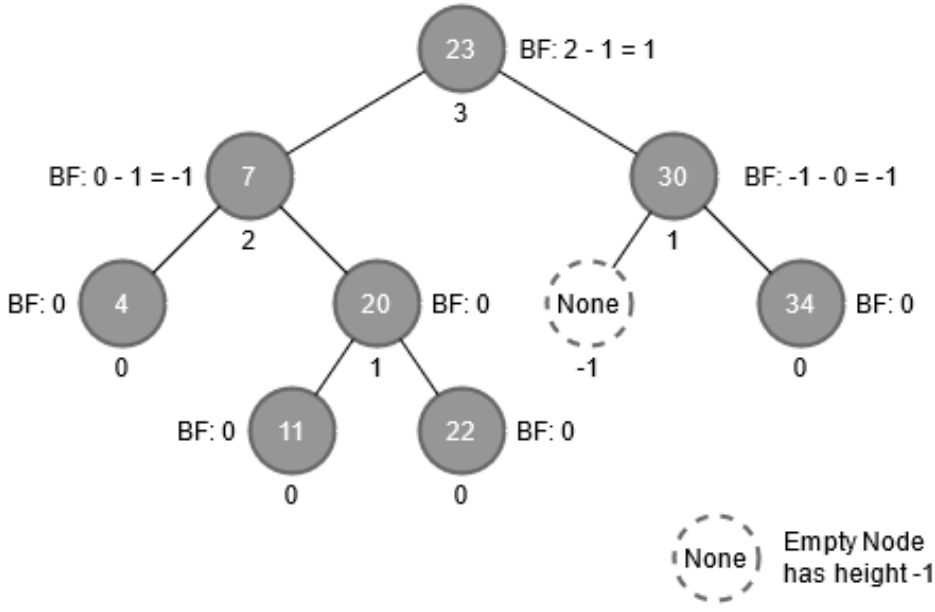
Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Following are the propoerties of AVL tree.

1. Sub trees of each node can differ by at most 1 in their height i.e. difference of height in left and right for each node must be -1 or 0 or 1

2. Every sub trees is an AVL tree

Following figure shows AVL tree.



Searching in AVL tree is same as normal BST. However, insertion and deletion has a different way to do it.

5.4.1 Insertion in AVL Tree

When a node is inserted into the tree, the tree may become unbalanced. In that situation, we will have to rebalance the tree through rotation at the deepest affected node whose balance is violated due to insertion. Here, four different cases can be there of violation at node k (deepest affected node). It is possible that there may be multiple nodes imbalanced due to insertion. So, the deepest affected node means the nearest node to the leaf in the direction in which the new node has been inserted.

Outside Case of Single Rotation:

1. An insertion into left subtree of left child of x
2. An insertion into right subtree of right child of x

Inside Case of Double Rotations:

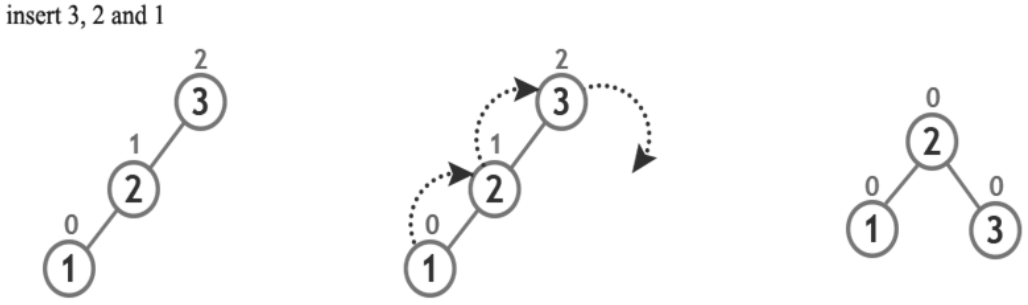
1. An insertion into right subtree of left child of x
2. An insertion into left subtree of right child of x

The cases 1 and 4 equivalent and requires single rotation to rebalance whereas the cases 2 and 3 equivalent and require double rotation to rebalance.

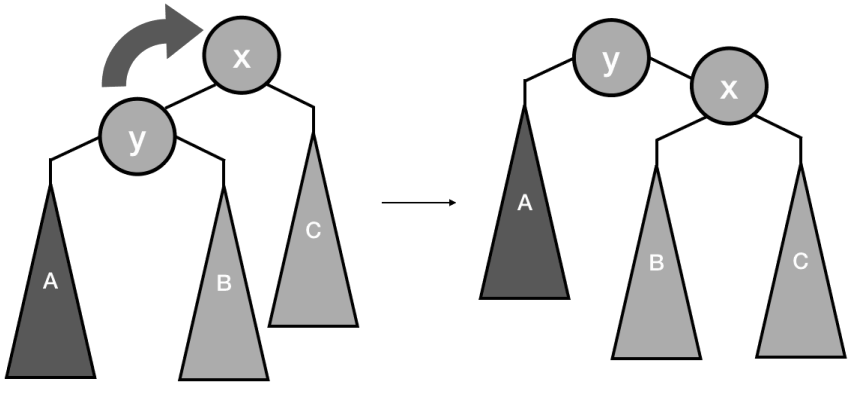
Outside Case-1: Right Rotation (RR Rotation)

When a node is inserted into left subtree of left (pivot) of the deepest affected node, LL rotation is performed. Here, new node is A is inserted into left node of y which is left child of x.

Example:



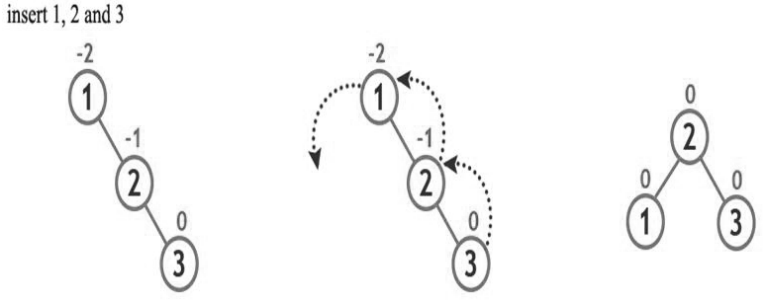
However, in this rotation, if there is right child (here B) of left child (here y) of deepest affected node (here x), that node becomes left child of the deepest affected node after RR rotation. Here, B was right child of y has now become left child of x after RR rotation. The following figure shows that transformation.



Outside Case-2: Left Rotation (LL Rotation)

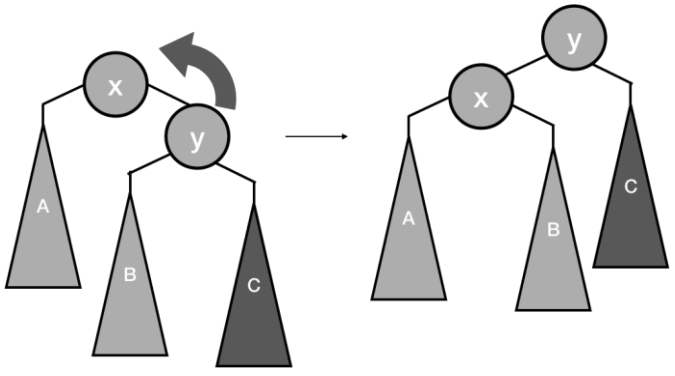
When a node is inserted into right subtree of right (pivot) of the deepest affected node, LL rotation is performed. Here, new node is C is inserted into right node of y which is right child of x.

Example:



However, in this rotation, if there is left child (here B) of right child (here y) of deepest affected node (here x), that node becomes right child of the deepest affected node after LL rotation. Here, B was left child of y has now become right child of x after LL rotation. The

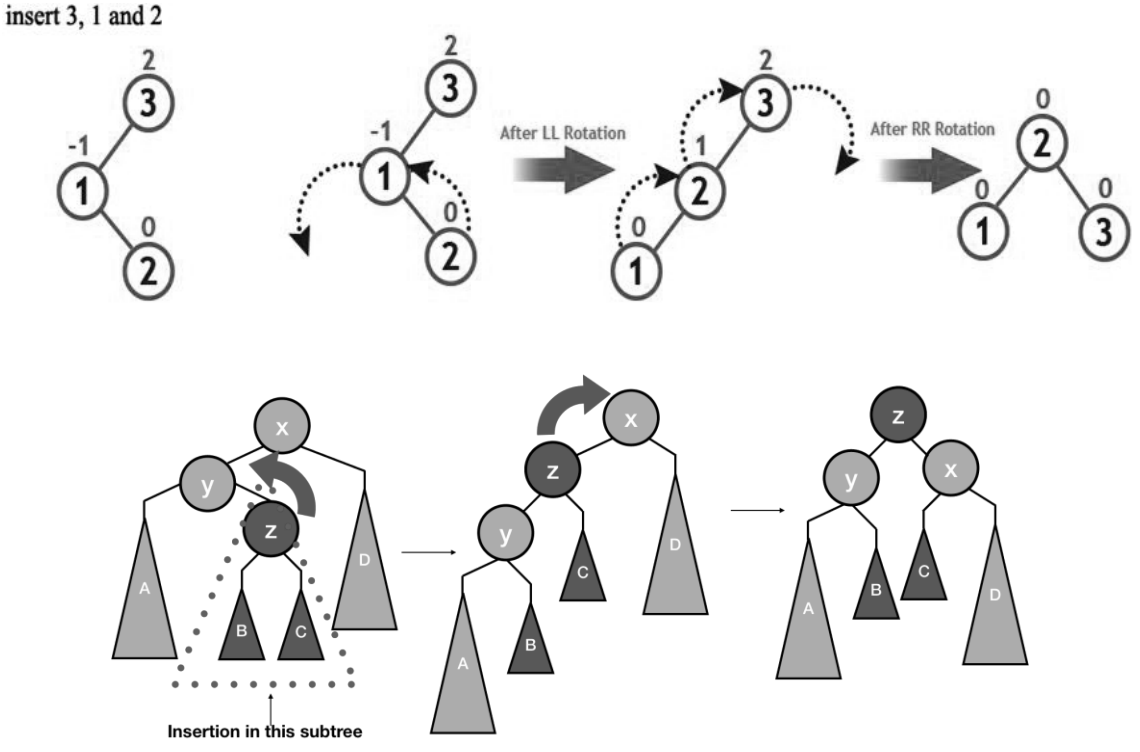
following figure shows that transformation.



Inside Case-1: Left-Right Rotation (LR Rotation)

When a node is inserted into right subtree of left child of the deepest affected node, LR Rotation is performed. We first do left rotation (LL) through the left child (pivot) of deepest affected node and then perform right rotation (RR) through the deepest affected node. Here, new node is B or C is inserted into right subtree of y which is left child of x. So LR rotation is performed.

Example:

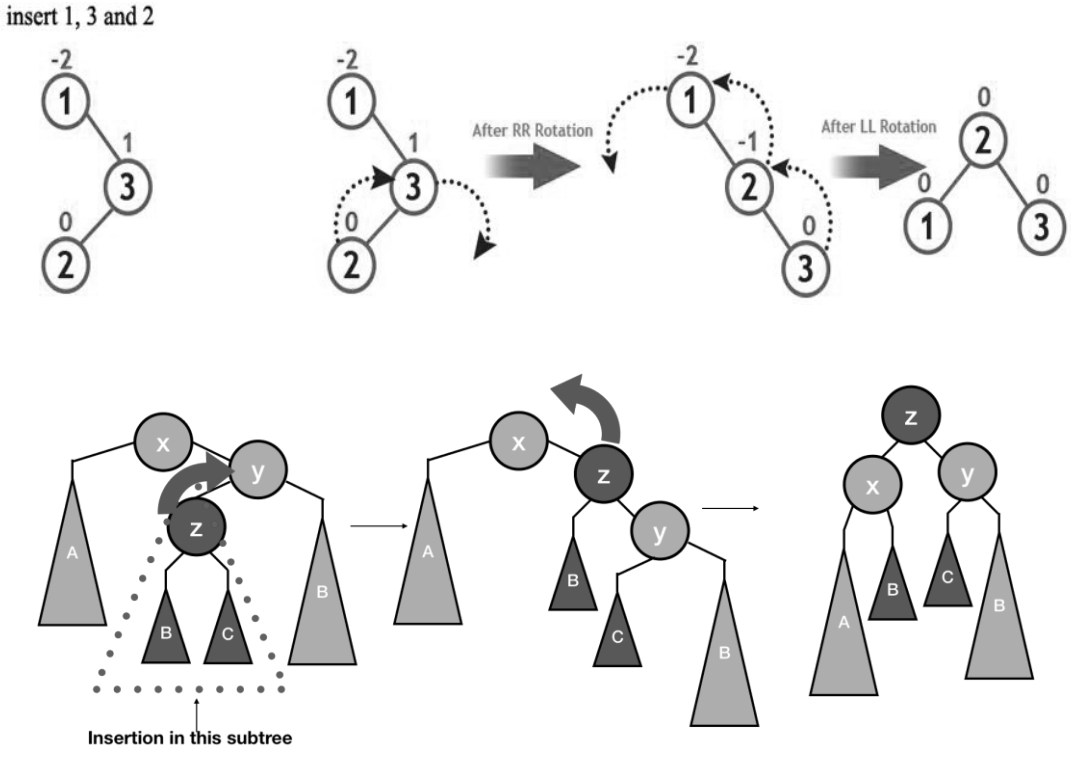


Inside Case-2: Right-Left Rotation (RL Rotation)

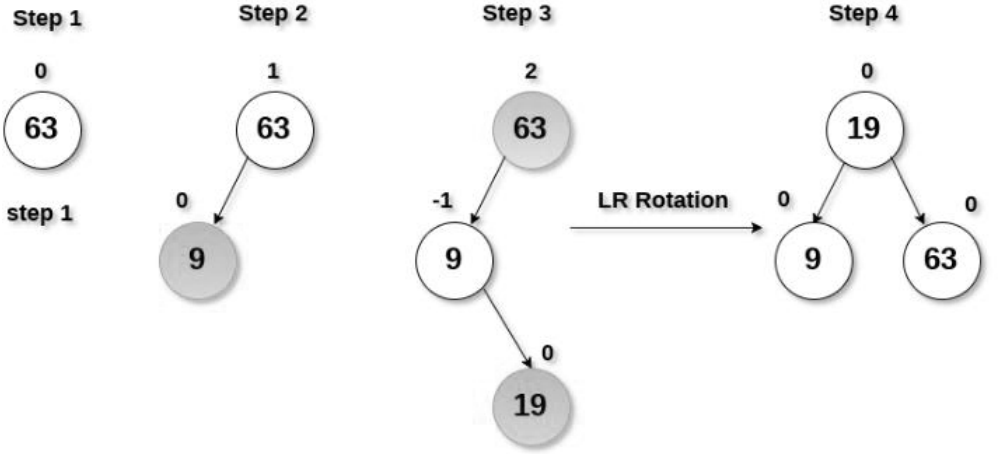
When a node is inserted into left subtree of right child of the deepest affected node, RL Rotation is performed. We first do right rotation (RR) through the right child (pivot)

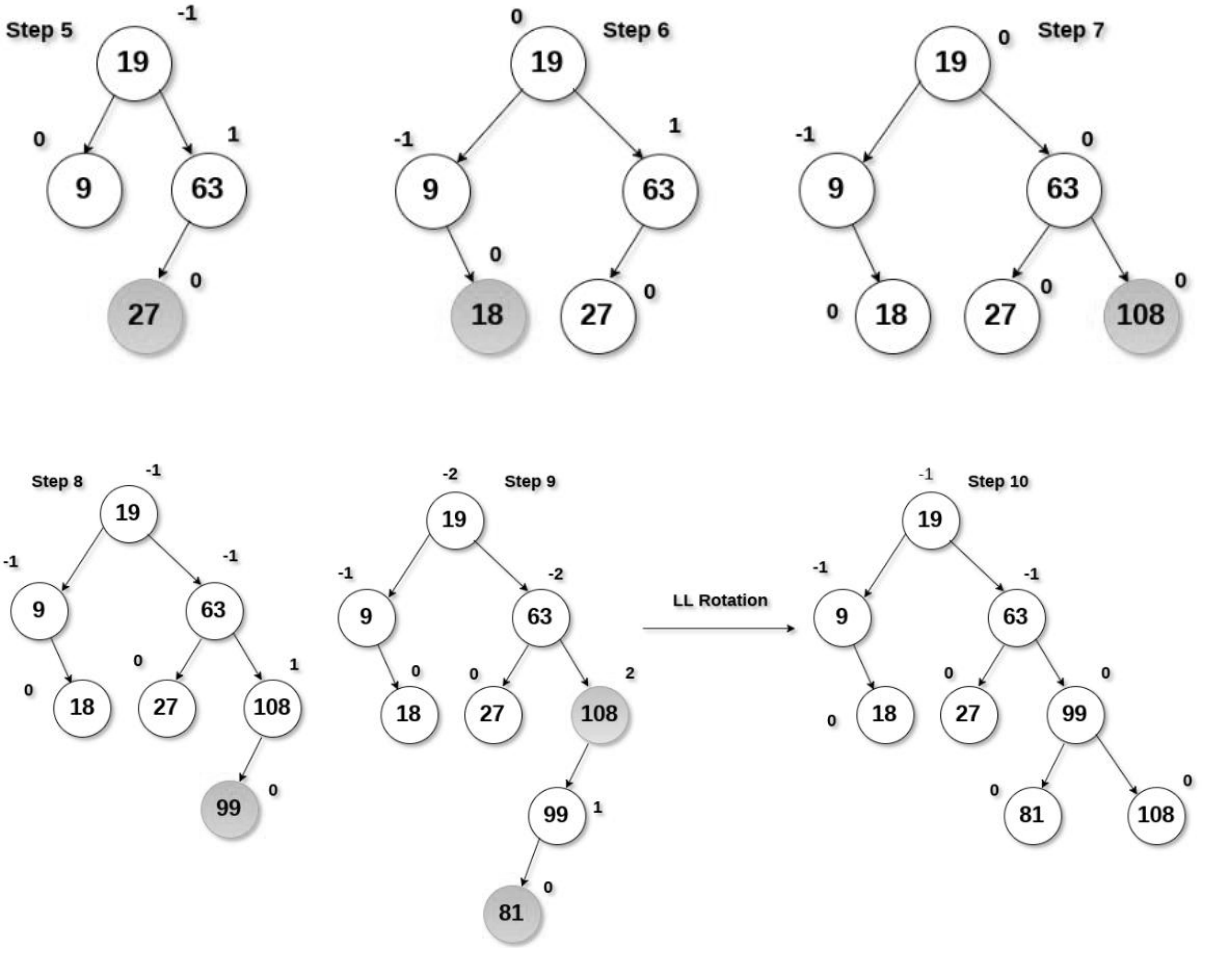
of deepest affected node and then perform left rotation (LL) through the deepest affected node. Here, new node is B or C is inserted into right subtree of y which is left child of x. So LR rotation is performed.

Example:



Example 1: 63, 9, 19, 27, 18, 108, 99, 81

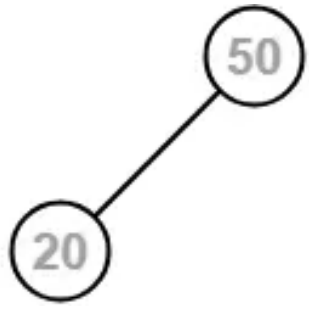




Example 2: 50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48

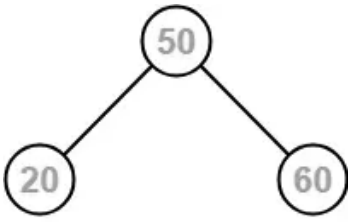
Step-1: Create node with 50

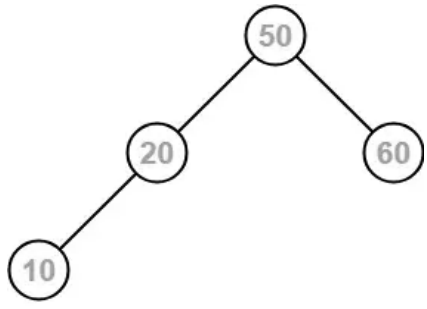
Step-2: Insert 20. As 20 < 50, so insert 20 in 50's left sub tree.



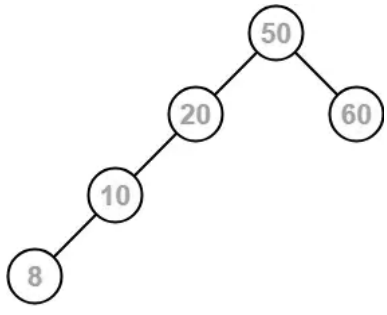
Step-3: Insert 60. As 60 > 50, so insert 60 in 50's right sub tree.

Step-4: Insert 10. As 10 < 50, so insert 10 in 50's left sub tree. As 10 < 20, so insert 10 in 20's left sub tree.

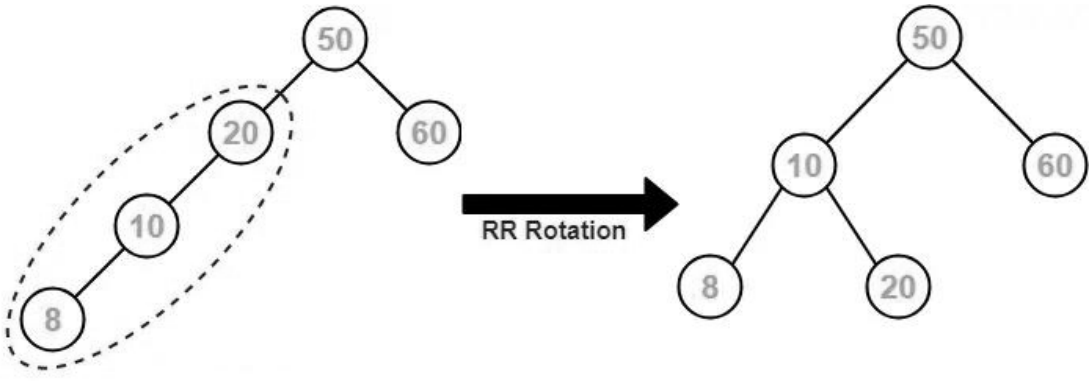




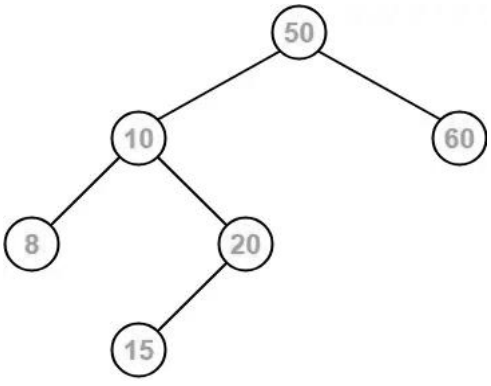
Step-5: Insert 6. As 8 < 50, so insert 8 in 50's left sub tree. As 8 < 20, so insert 8 in 20's left sub tree. As 8 < 10, so insert 8 in 10's left sub tree.



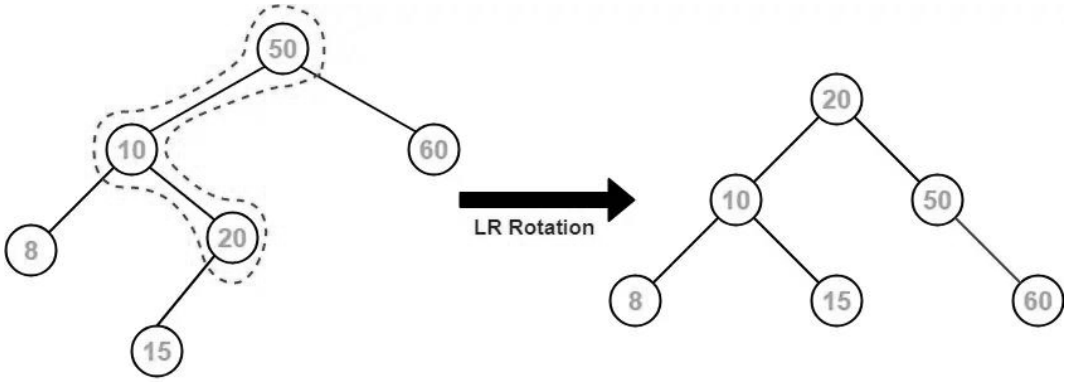
The tree is imbalanced. To balance the tree, find the first imbalanced node on the path from the newly inserted node (node 8) to the root node. The first imbalanced node is node 20. Now, count three nodes from node 20 in the direction of leaf node. Then, perform RR Rotation to balance the tree.



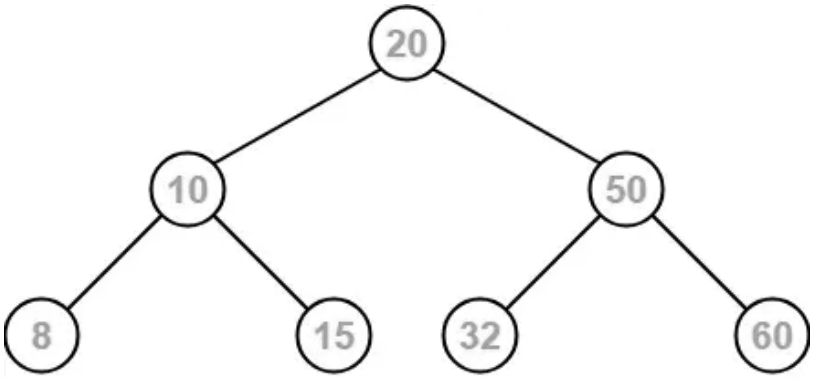
Step-6: Insert 15. As 15 < 50, so insert 15 in 50's left sub tree. As 15 > 10, so insert 15 in 10's right sub tree. As 15 < 20, so insert 15 in 20's left sub tree.



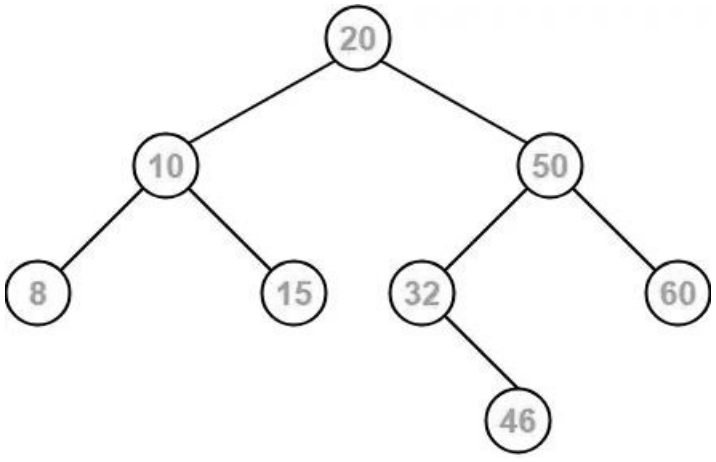
The tree is imbalanced. To balance the tree, find the first imbalanced node on the path from the newly inserted node (node 15) to the root node. The first imbalanced node is node 50. Now, count three nodes from node 50 in the direction of leaf node. Then, use LR Rotation to balance the tree.



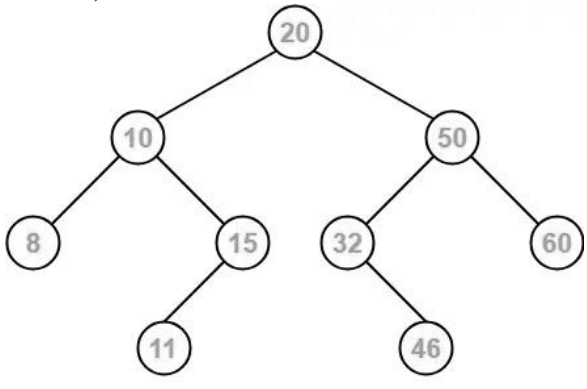
Step-07: Insert 32. As 32 > 20, so insert 32 in 20's right sub tree. As 32 < 50, so insert 32 in 50's left sub tree.



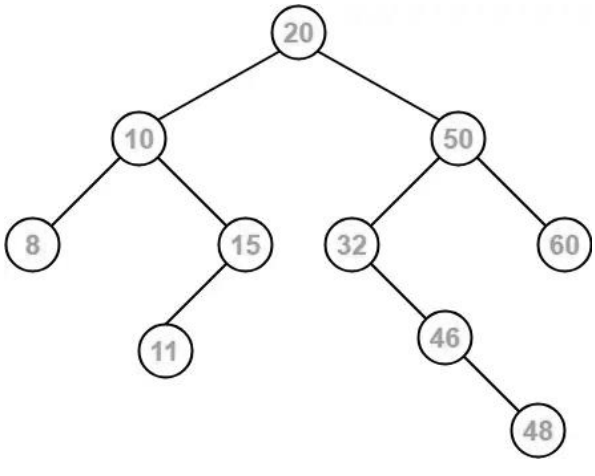
Step-08: Insert 46. As 46 > 20, so insert 46 in 20's right sub tree. As 46 < 50, so insert 46 in 50's left sub tree. As 46 > 32, so insert 46 in 32's right sub tree.



Step-09: Insert 11. As 11 < 20, so insert 11 in 20's left sub tree. As 11 > 10, so insert 11 in 10's right sub tree. As 11 < 15, so insert 11 in 15's left sub tree.

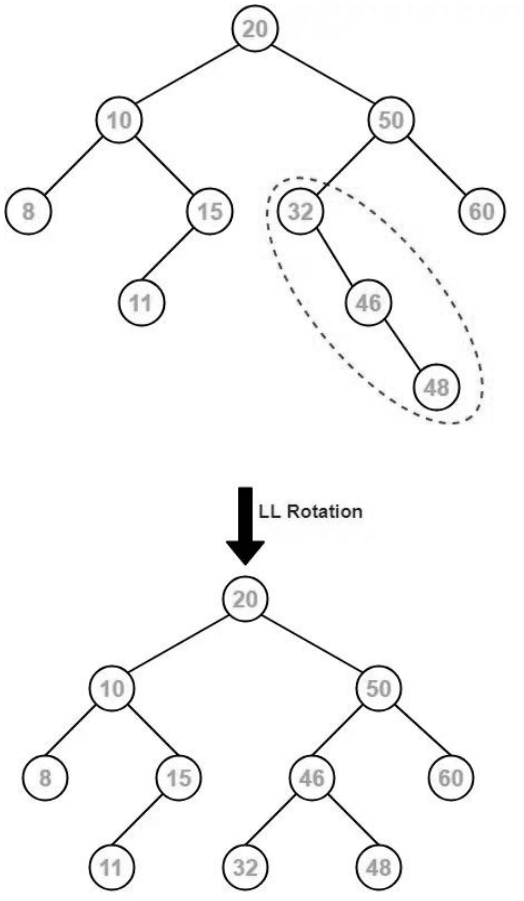


Step-10: Insert 48. As 48 > 20, so insert 48 in 20's right sub tree. As 48 < 50, so insert 48 in 50's left sub tree. As 48 > 32, so insert 48 in 32's right sub tree. As 48 > 46, so insert 48 in 46's right sub tree.

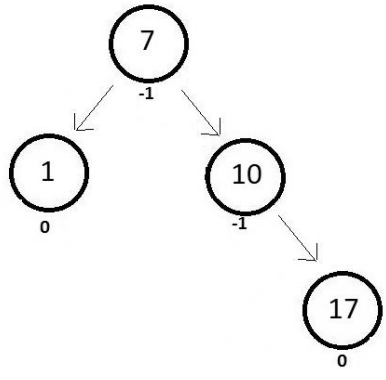


The tree is imbalanced. To balance the tree, find the first imbalanced node on the path from the newly inserted node (node 48) to the root node. The first imbalanced node is node 32.

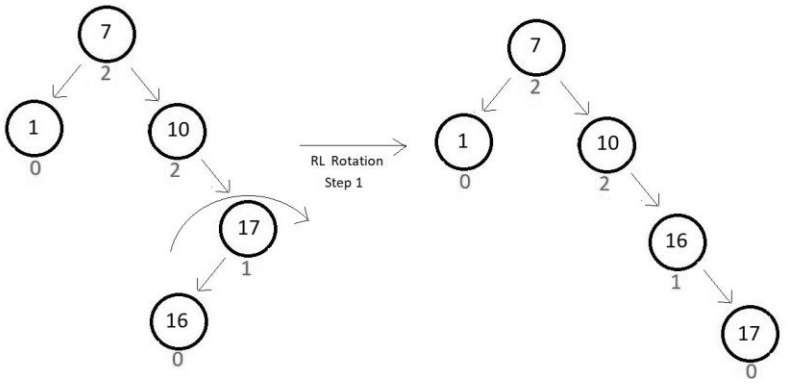
Now, count three nodes from node 32 in the direction of leaf node. Then, use LL rotation to balance the tree.



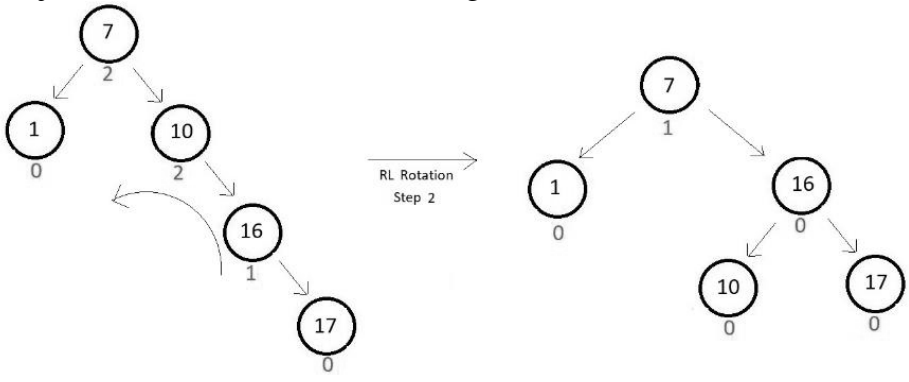
Example 3: Perform RL Rotation for the following tree after inserting 16.



Since this is a case of right-left insertion with respect to the first imbalanced node which is node 10, we would first rotate right once with respect to the child of the first imbalanced node which comes into the path of the insertion node. Follow the figure below.



Now, we rotate left with respect to the node we found first imbalanced, here 10. And this would do our job. Our tree has been balanced again.



5.4.2 Deletion in AVL Tree

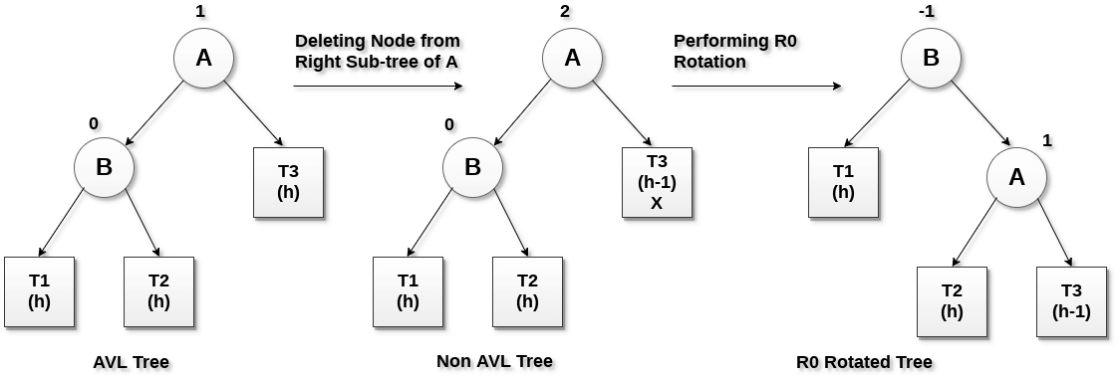
Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the height. For this purpose, we need to perform rotations. The two types of rotations are L rotation and R rotation.

Here, we will discuss R rotations. L rotations are the mirror images of them. If the node which is to be deleted is present in the left suB tree of the critical node, then L rotation needs to be applied else if, the node which is to be deleted is present in the right suB tree of the critical node, the R rotation will be applied.

Let us consider that, A is the critical node and B is the root node (pivot) of its left suB tree. If node X, present in the right suB tree of A, is to be deleted, then there can be three different situations:

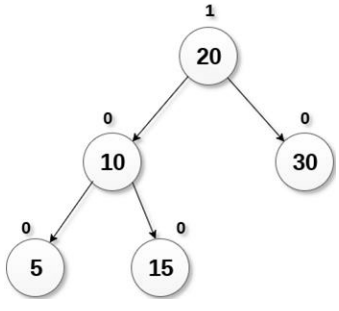
R0 Rotation (Pivot B has balance factor 0)

If the pivot node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation.

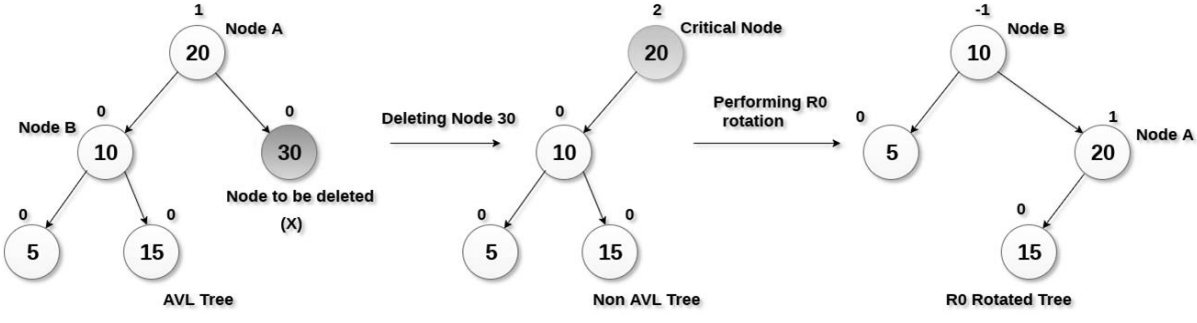


The critical node A is moved to its right and the pivot node B becomes the root of the tree with T1 as its left suB tree. The suB trees T2 and T3 becomes the left and right suB tree of the node A. The process involved in R0 rotation is shown in the above image.

Example 1: Delete the node 30 from the following AVL tree.

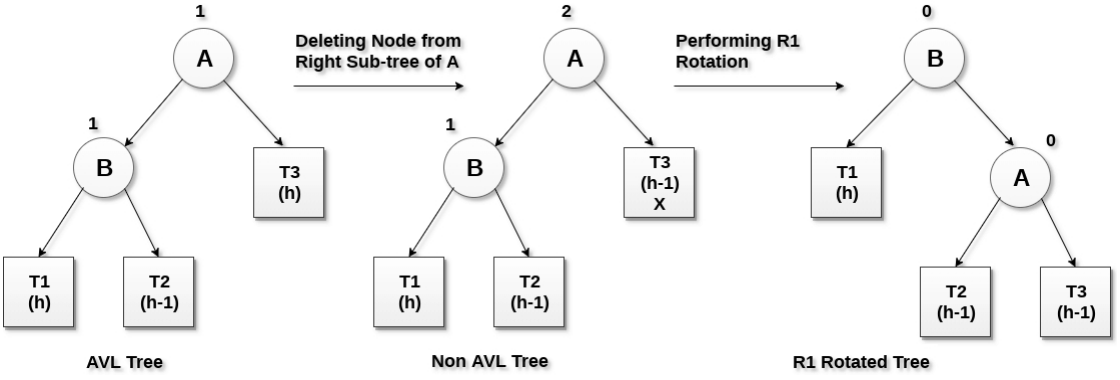


In this case, the pivot node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The pivot node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.

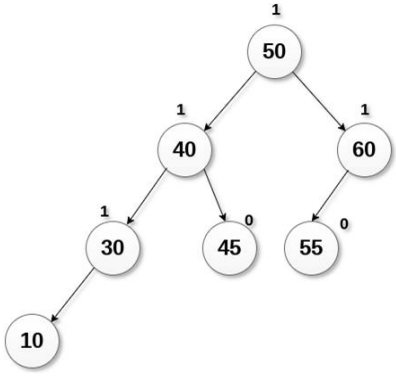


R1 Rotation (Pivot B has balance factor 1)

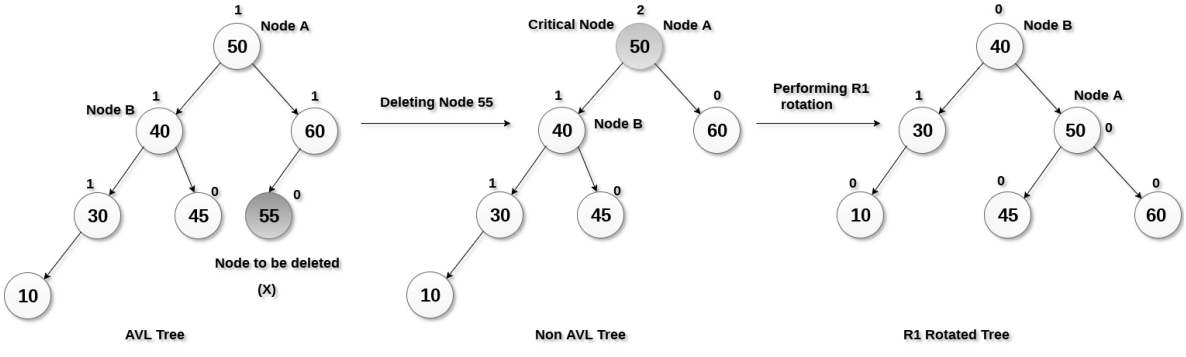
R1 Rotation is to be performed if the balance factor of pivot node B is 1. In R1 rotation, the critical node A is moved to its right having suB trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left suB tree of the pivot node B.



Example 2: Delete the node 55 from the following AVL tree.

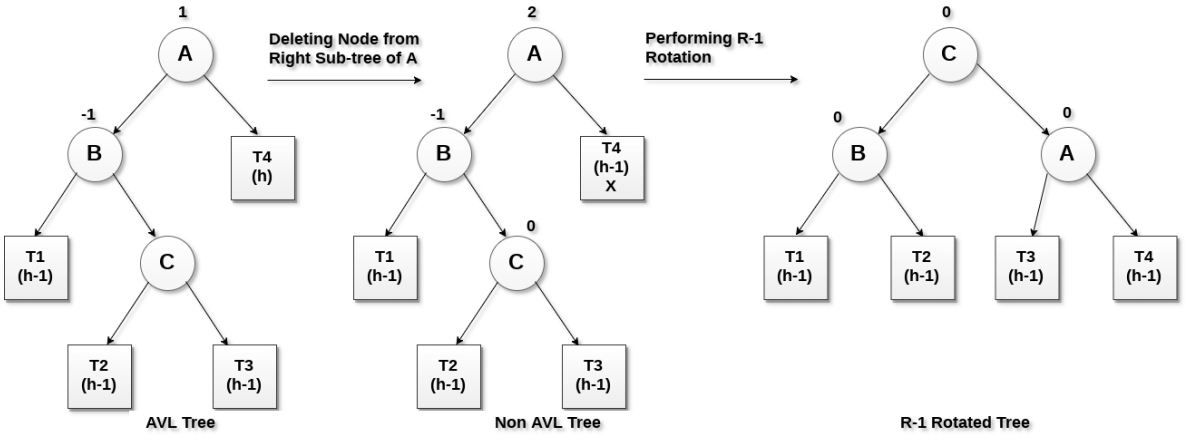


Deleting 55 from the AVL Tree disturbs the balance factor of the node 50 i.e. node A which becomes the critical node. This is the condition of R1 rotation in which, the node A will be moved to its right (shown in the following figure). The right of pivot B is now become the left of A (i.e. 45).

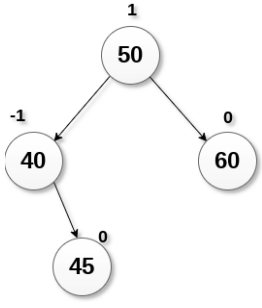


R-1 Rotation (Pivot B has balance factor -1)

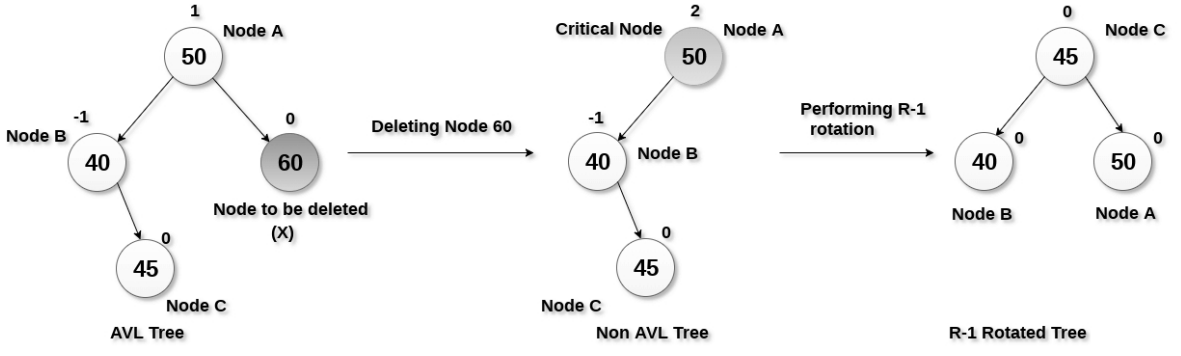
R-1 rotation is to be performed if the pivot node B has balance factor -1. This case is treated in the same way as LR rotation. In this case, the node C, which is the right child of pivot node B, becomes the root node of the tree with pivot B and A as its left and right children respectively. The suB trees T1, T2 becomes the left and right suB trees of pivot B whereas, T3, T4 become the left and right suB trees of A.



Example 3: Delete the node 60 from the following AVL tree.



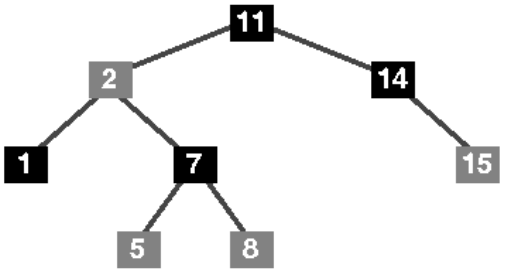
In this case, pivot node B has balance factor -1. Deleting the node 60, disturbs the balance factor of the node 50 therefore, it needs to be R-1 rotated. The node C i.e. 45 becomes the root of the tree with the pivot node B(40) and A(50) as its left and right child.



5.5 RED-BLACK TREE

The red-Black tree is a binary search tree. The prerequisite of the red-black tree is that we should know about the binary search tree. In a binary search tree, the values of the nodes in the left subtree should be less than the value of the root node, and the values of the nodes in the right subtree should be greater than the value of the root node. Each node in the Red-black tree contains an extra bit that represents a color to ensure that the tree is balanced

during any operations performed on the tree like insertion, deletion, etc. In a binary search tree, the searching, insertion and deletion take **O(log$_2$n)** time in the average case, **O(1)** in the best case and **O(n)** in the worst case.



5.5.1 Why do we require a Red-Black tree?

If AVL is also a height-balanced tree then why do we need Red-Black Tree? The Red-Black tree is used because the AVL tree requires many rotations when the tree is large, whereas the Red-Black tree requires a maximum of two rotations to balance the tree. The main difference between the AVL tree and the Red-Black tree is that the AVL tree is strictly balanced, while the Red-Black tree is not completely height-balanced. So, the AVL tree is more balanced than the Red-Black tree, but the Red-Black tree guarantees O(log$_2$n) time for all operations like insertion, deletion, and searching. Insertion is easier in the AVL tree as the AVL tree is strictly balanced, whereas deletion and searching are easier in the Red-Black tree as the Red-Black tree requires fewer rotations. As the name suggests that the node is either colored in **Red** or **Black** color. Sometimes no rotation is required, and only recoloring is needed to balance the tree.

5.5.2 Properties of Red-Black tree

1. It is a self-balancing Binary Search tree. Here, self-balancing means that it balances the tree itself by either doing the rotations or recoloring the nodes.
2. This tree data structure is named as a Red-Black tree as each node is either Red or Black in color. Every node stores one extra information known as a bit that represents the color of the node. For example, 0 bit denotes the black color while 1 bit denotes the red color of the node. Other information stored by the node is similar to the binary tree, i.e., data part, left pointer and right pointer.
3. In the Red-Black tree, the root node is always black in color.
4. In a binary tree, we consider those nodes as the leaf which have no child. In contrast, in the Red-Black tree, the nodes that have no child are considered the internal nodes and these nodes are connected to the NIL nodes that are always black in color. The NIL nodes are the leaf nodes in the Red-Black tree.
5. If the node is Red, then its children should be in Black color. In other words, we can say that there should be no red-red parent-child relationship.
6. Every path from a node to any of its descendant's NIL node should have same number of black nodes.

Every AVL tree can be a Red-Black tree if we color each node either by Red or Black

color. But every Red-Black tree is not an AVL because the AVL tree is strictly height-balanced while the Red-Black tree is not completely height-balanced.

5.6 B TREES

B trees were originally invented for storing data structures on disk, where locality is even more crucial than with memory. Accessing a disk location takes about 5ms = 5,000,000ns. Therefore, if you are storing a tree on disk, you want to make sure that a given disk read is as effective as possible. B trees have a high branching factor, much larger than 2, which ensures that few disk reads are needed to navigate to the place where data is stored. B trees may also useful for in-memory data structures because these days main memory is almost as slow relative to the processor as disk drives were to main memory when B trees were first introduced!

A B tree of order m is an m-way tree (i.e., a tree where each node may have up to m children/links). It has the following properties:

1. The number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
2. All leaves are on the same level
3. All non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children/links and maximum m and hence at least $\lceil m / 2 \rceil$ -1 keys and max m-1
4. The root is either a leaf node, or it has from two to m children/links
5. A leaf node contains no more than m – 1 keys

Here, the number m should always be odd. A B tree of order m of height h will have the maximum number of keys when all nodes are completely filled. So, the B tree will have

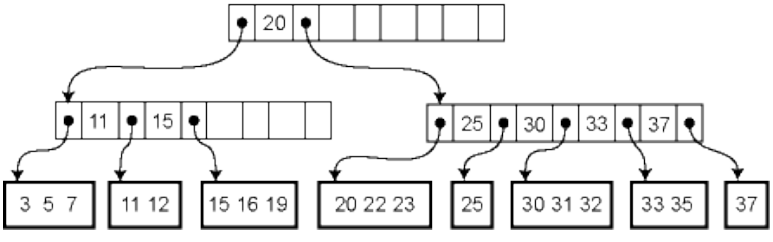
$n = (m^{h+1}–1)$ keys in this situation. So, for example, a B tree of order 5 with height 2 will have maximum keys = 5^{2+1} - 1 = 125 – 1 = 124.

The following is a B tree of order-5 (also called 4-5 Tree). It has the following properties as per the characteristics of B tree.

A 4-5 B tree means 4 key values and 5 links/children. To be precise,

1. Other than the root node, all internal nodes have at least, ceil(5 / 2) = ceil(2.5) = 3 children and maximum 5.
2. At least ceil(5/2)-1 = ceil(2.5)-1 = 3 -1 = 2 keys and maximum 4.

In practice B trees are usually having a very large orders in terms of thousands. See the following figure. Note that all the leaves are at the same level.

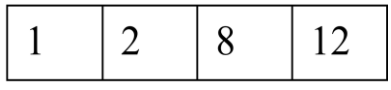


5.6.1 Construction of (Insertion in) B tree

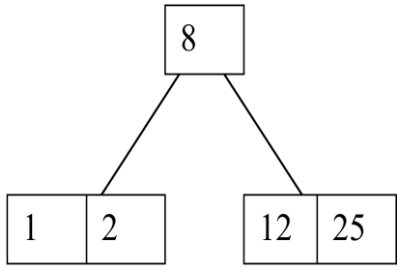
1. *Attempt to insert the new key into a leaf.*
2. *If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent.*
3. *If this would result in the parent becoming too big, split the parent into two, promoting the middle key.*
4. *This strategy might have to be repeated all the way to the top.*
5. *If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher.*

Example 1: Suppose we want to create a B tree of order-5 with the following values:1, 12, 8, 2, 25, 5, 14, 28, 17, 7, 52, 16, 48, 68, 3, 26, 29, 53, 55, 45.

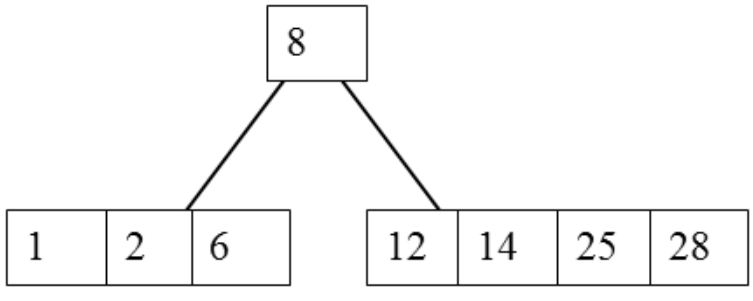
The first four items go into the root:



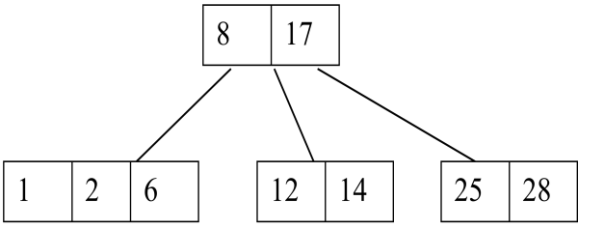
To put the fifth item in the root would violate condition 5. Therefore, when 25 arrives, pick the middle key to make a new root.



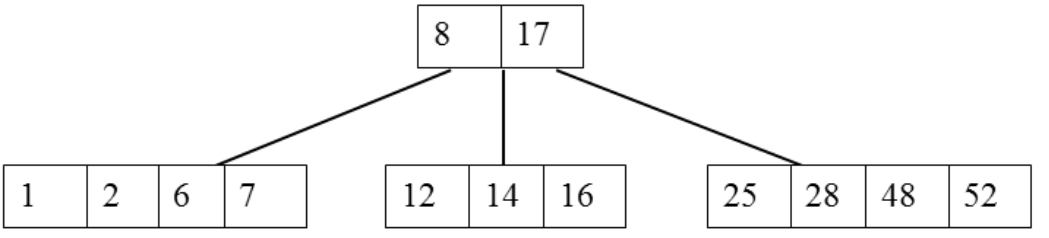
6, 14, 28 get added to the leaf nodes:



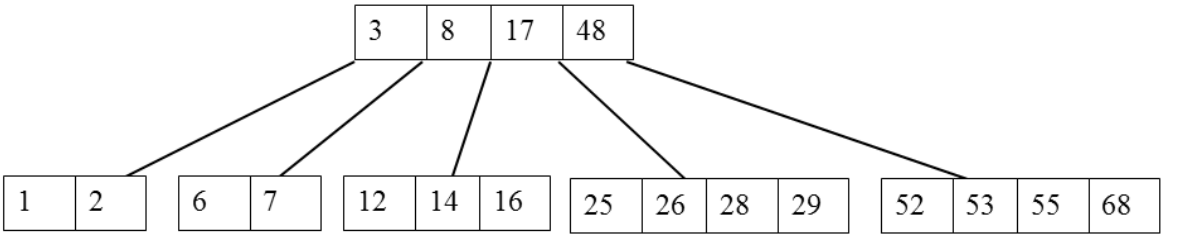
Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf.



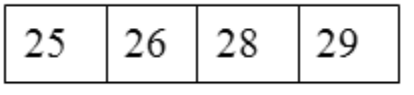
7, 52, 16, 48 get added to the leaf nodes



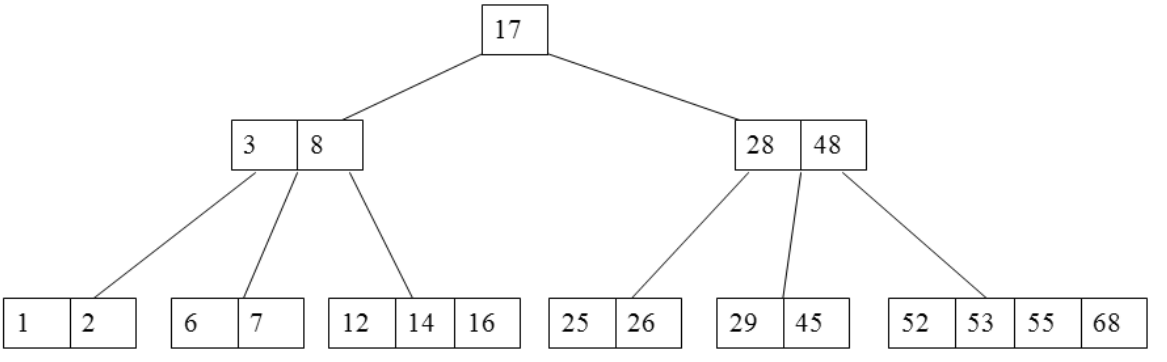
Adding 68 causes us to split the right most leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves.



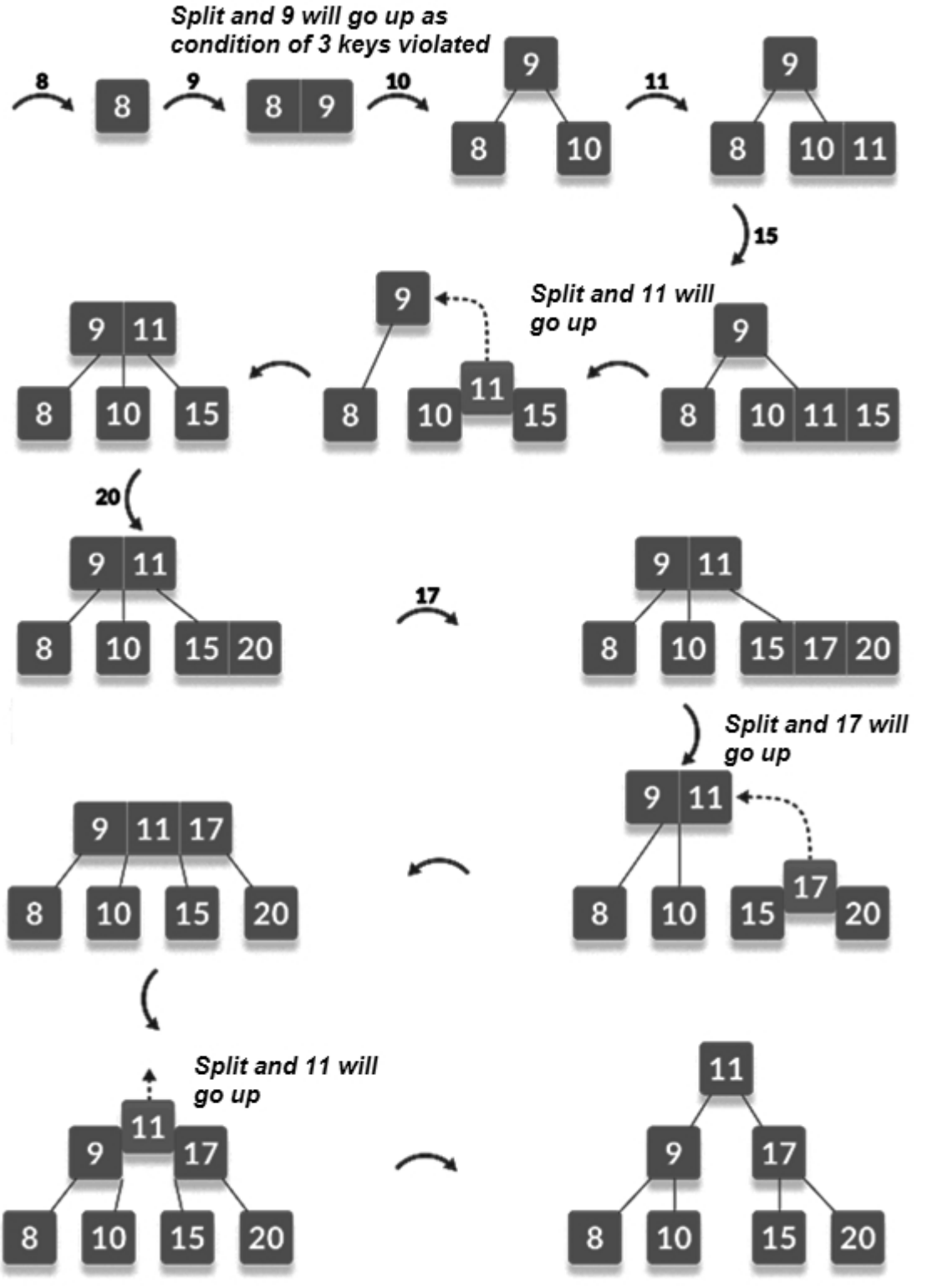
Adding 45 causes a split of



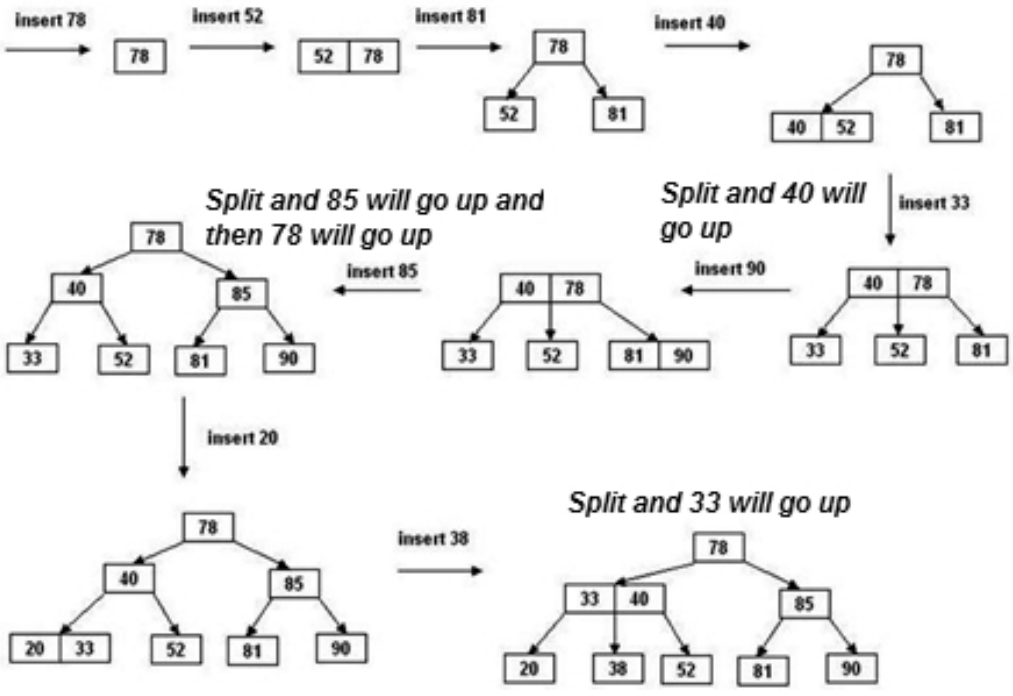
and promoting 28 to the root then causes the root to split .



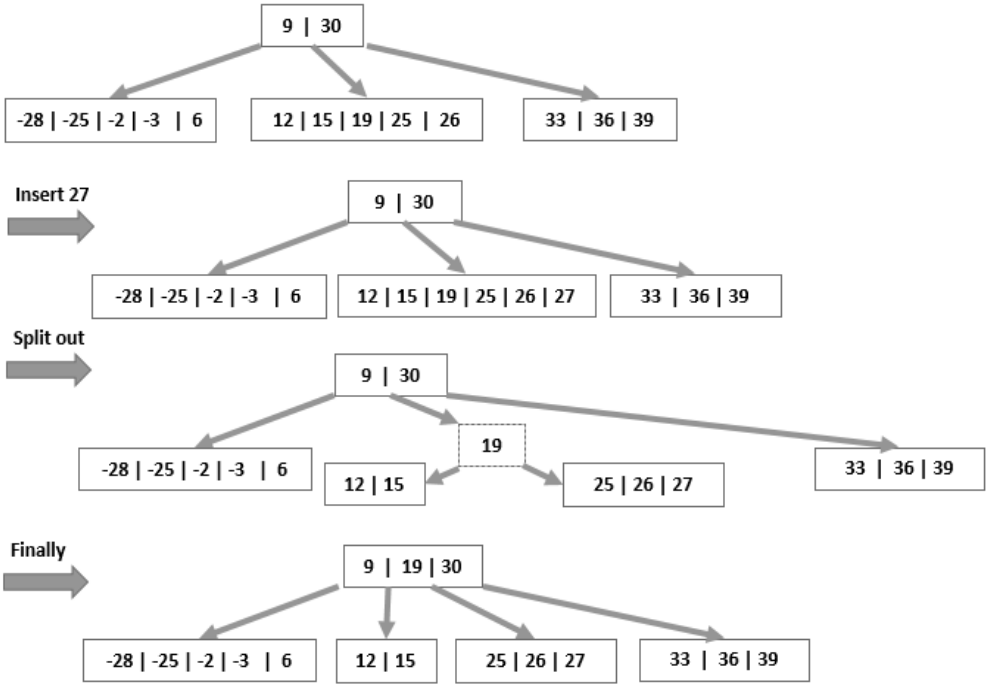
Example 2: Create a B tree of order-3 with following elements 8, 9, 10, 11, 15, 16, 17, 18, 20, 23


Split and 9 will go up as condition of 3 keys violated
8
9
10
11
15
Split and 11 will go up
20
17
Split and 17 will go up
Split and 11 will go up

Example 3: Create a B tree of order-3 with following elements 78, 52, 81, 40, 33, 90, 85, 20 38



Example 4: Create a B tree of order-6 with following elements 9, 19, 30, -28, -25, -2, -3, 6, 12, 15, 25, 26, 27, 33, 36, 39

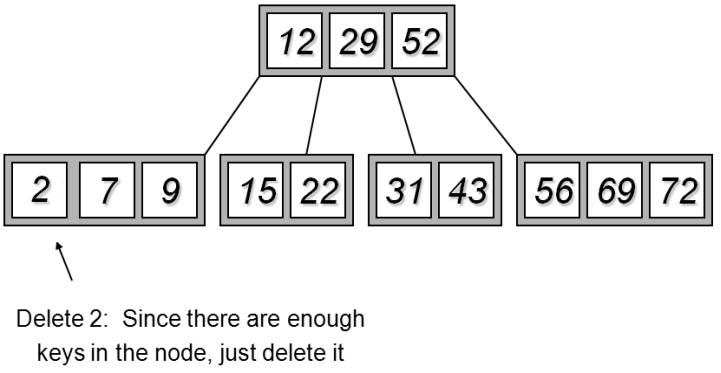


5.6.2 Deletion in B tree

While removing the keys from B tree, the key conditions must not be violated. Let us assume a tree of order-5 and we are removing some specific nodes to consider the different cases of violation of the key and children conditions.

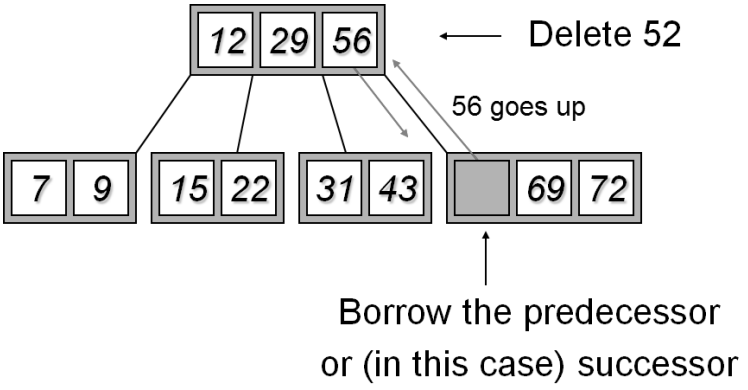
Type #1: Simple leaf deletion

If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.



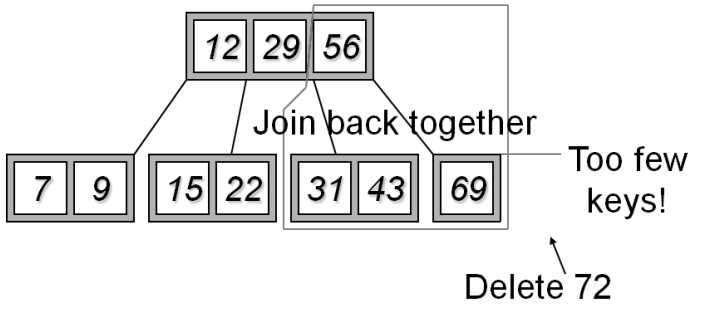
Type #2: Simple non-leaf deletion

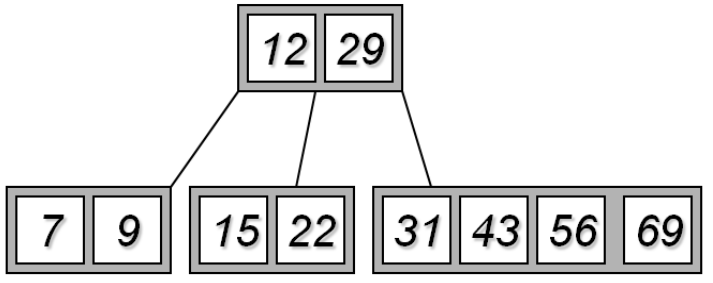
If the key is not in a leaf then it is guaranteed (by the nature of a B tree) that its predecessor or successor will be in a leaf -- in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position. Min. keys required at non-leaf level are 2.



Type #3: Too few keys in node and its siblings

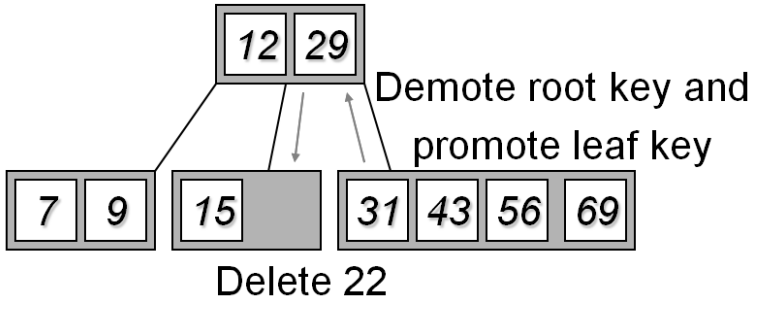
If neither of them has more than the min. number of keys (here, 2 in this case) then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required.

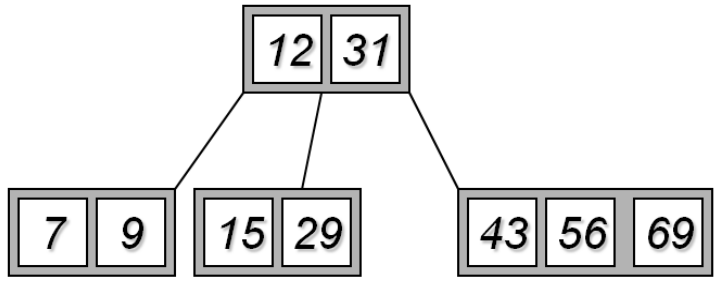




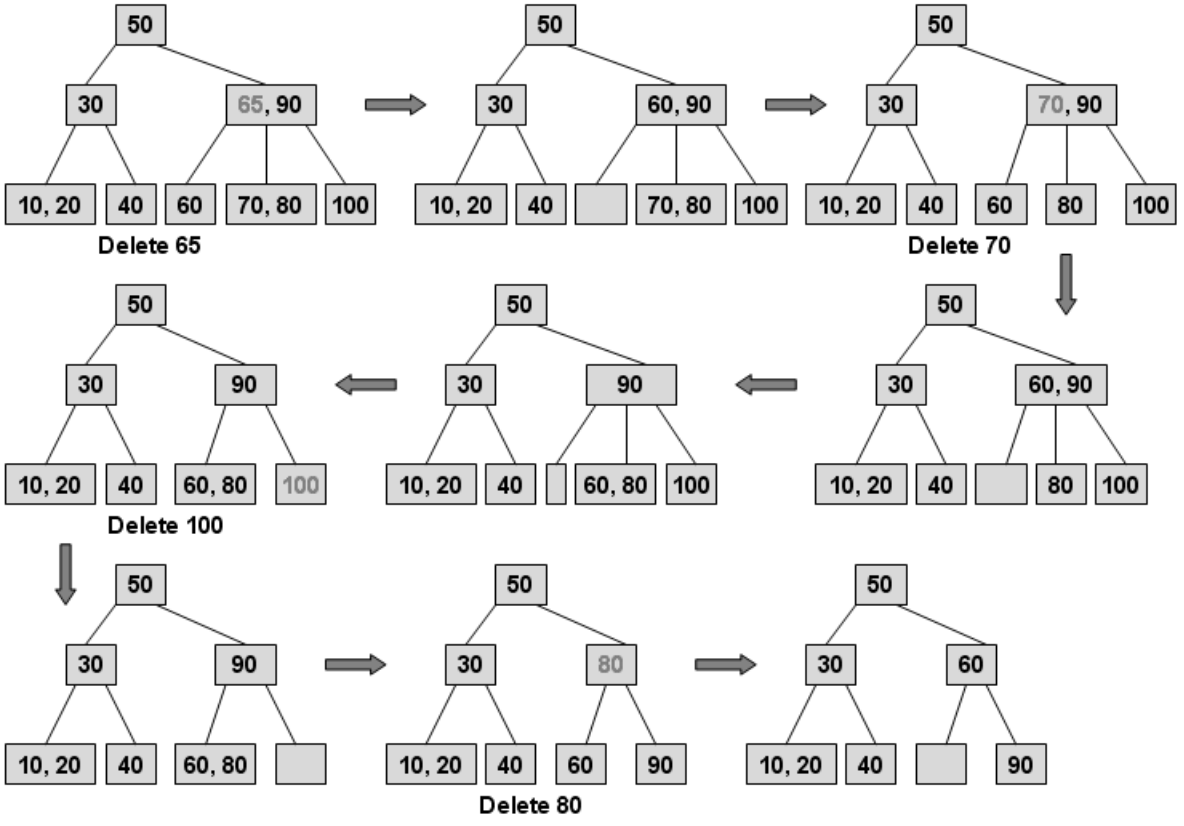
Type #4: Enough siblings

If one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf.





Example 1: Delete the 65, 70, 100 and 80 from the given B tree.



5.7 B+TREES

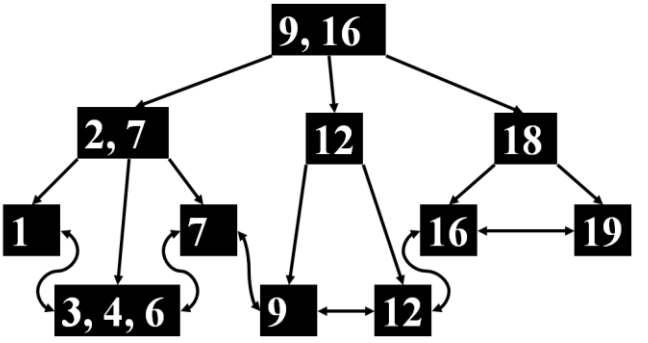
Similar to B trees, with a few slight differences. All data is stored at the leaf nodes (leaf pages); all other nodes (index pages) only store keys. Leaf pages are linked to each other which leads to duplication of keys. Every key to the right of a particular key is >= to that key.

5.7.1 B+ Tree Insertion

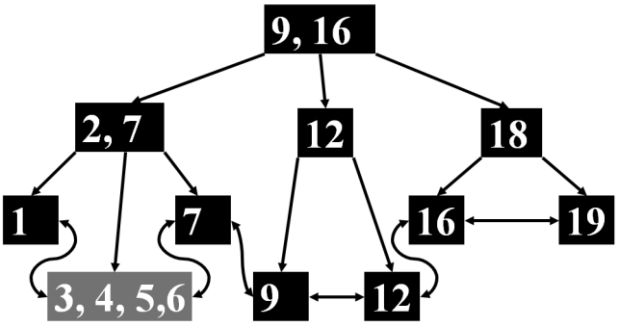
The tree will be constructed in the same way B tree is constructed. However, there are following things which will have to be considered.

1. *Insert at bottom level.*
2. *If leaf page overflows, split page and copy middle element to next index page.*
3. *If index page overflows, split page and move middle element to next index page*

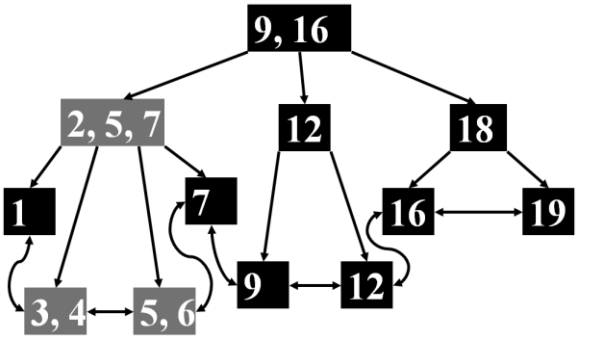
Example 1: Consider the following B+ tree and insert 5 into it.



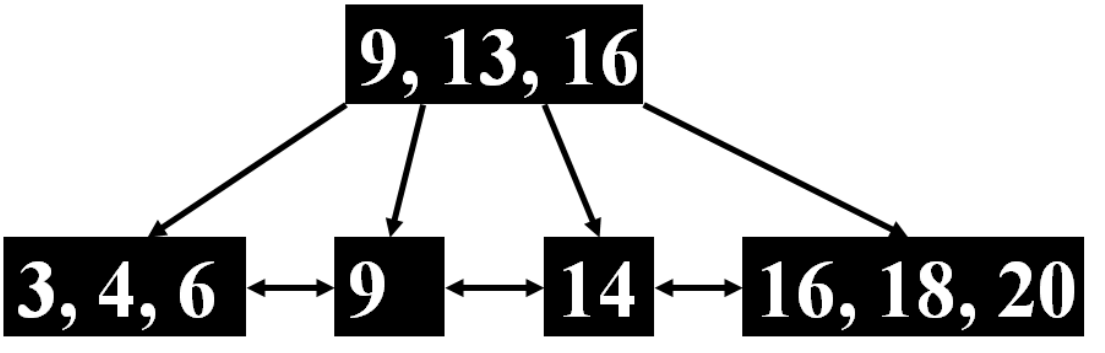
Insert 5.



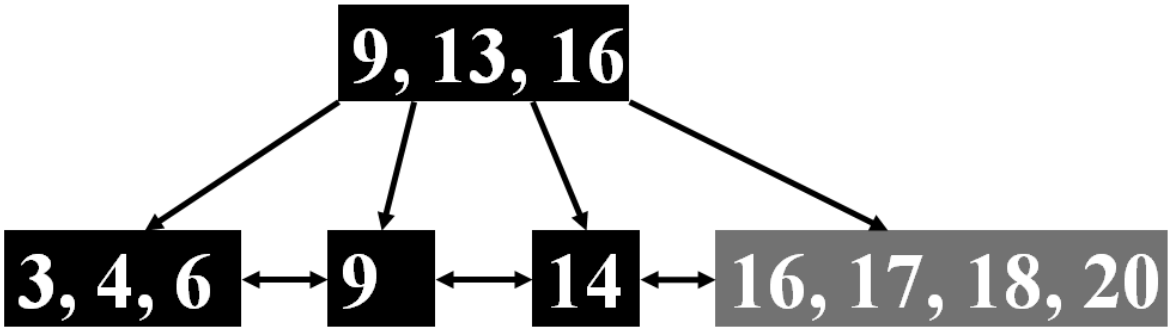
Split page, copy 5



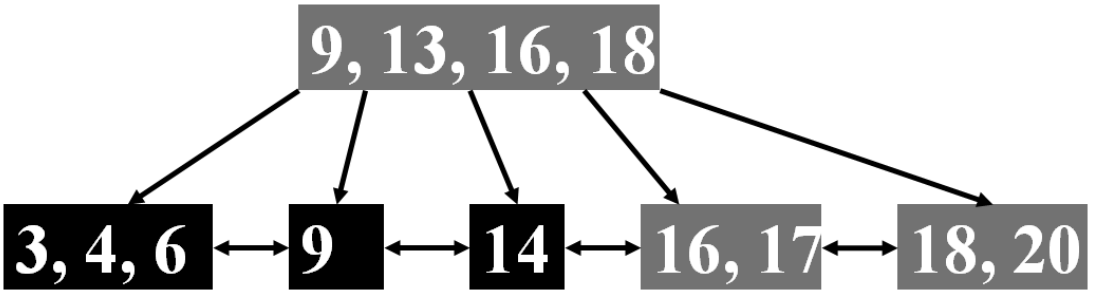
Example 2: Consider the following B+ tree and insert 17 into it.



Insert 17



Split leaf page and copy 18

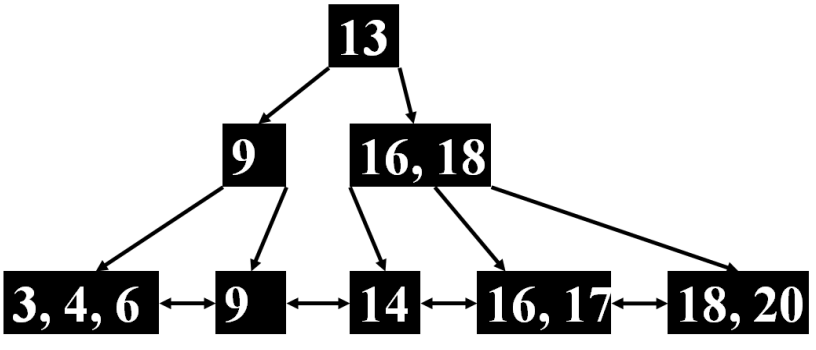


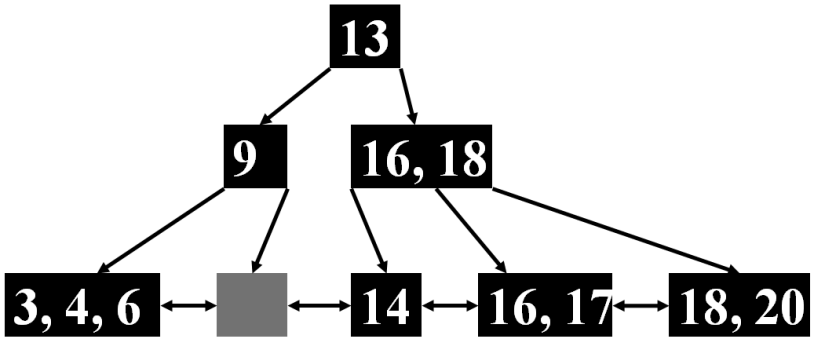
5.7.2 B+ Tree Deletion

While removing the keys from B+ tree, the following things to be taken into consideration.

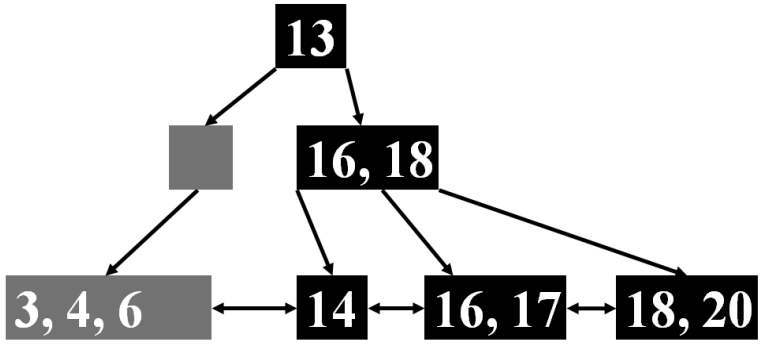
1. *Delete key and data from leaf page.*
2. *If leaf page underflows, merge with sibling and delete key in between them.*
3. *If index page underflows, merge with sibling and move down key in between them.*

Remove 9

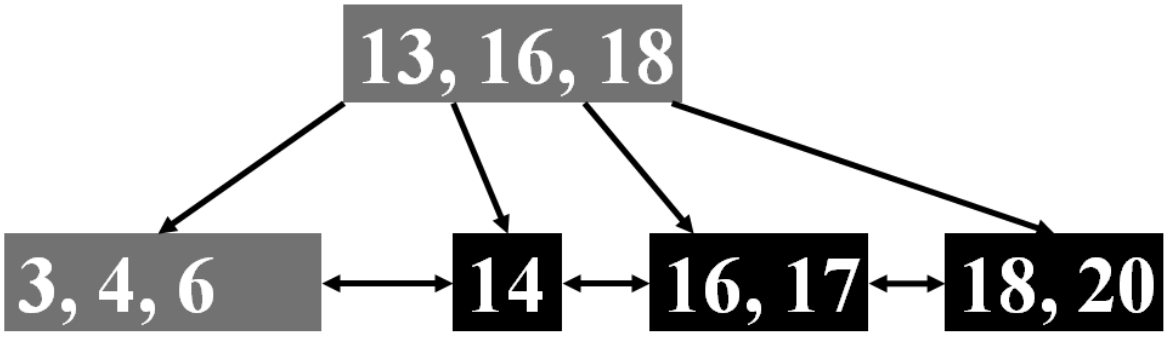




Leaf page underflow, so merge with sibling and remove 9



Index page underflow, so merge with sibling and demote 13



5.8 HEAP SORT

A heap is a data structure that stores a collection of objects (with keys), and has the following properties:

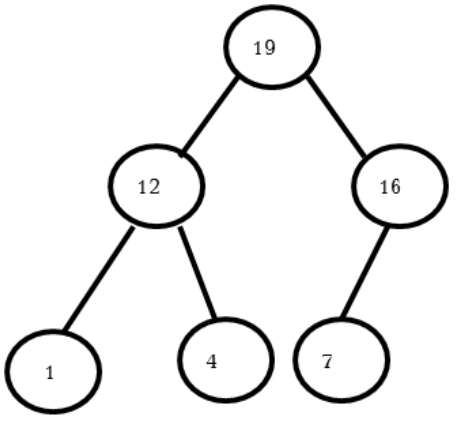
1. Complete Binary tree
2. Heap Order

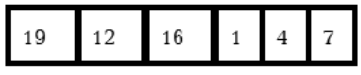
It is implemented as an array where each node in the tree corresponds to an element of the array. The binary heap data structures is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is completely filled on all levels except possibly lowest.

For every node v, other than the root, the key stored in v is greater or equal (smaller or

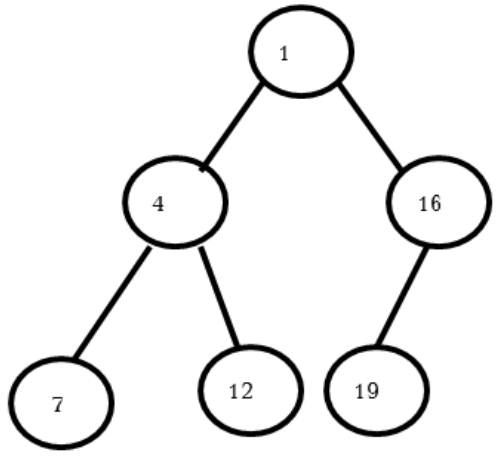
equal for max heap) than the key stored in the parent of v. In the above case the maximum value is stored in the root. Based on value stored at root, there are two types of heap.

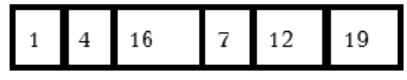
1. Max Heap: In this, the parent node is always greater than its children.
2. Min Heap: In this, the parent node is always smaller than its children.





Max Heap





Min Heap

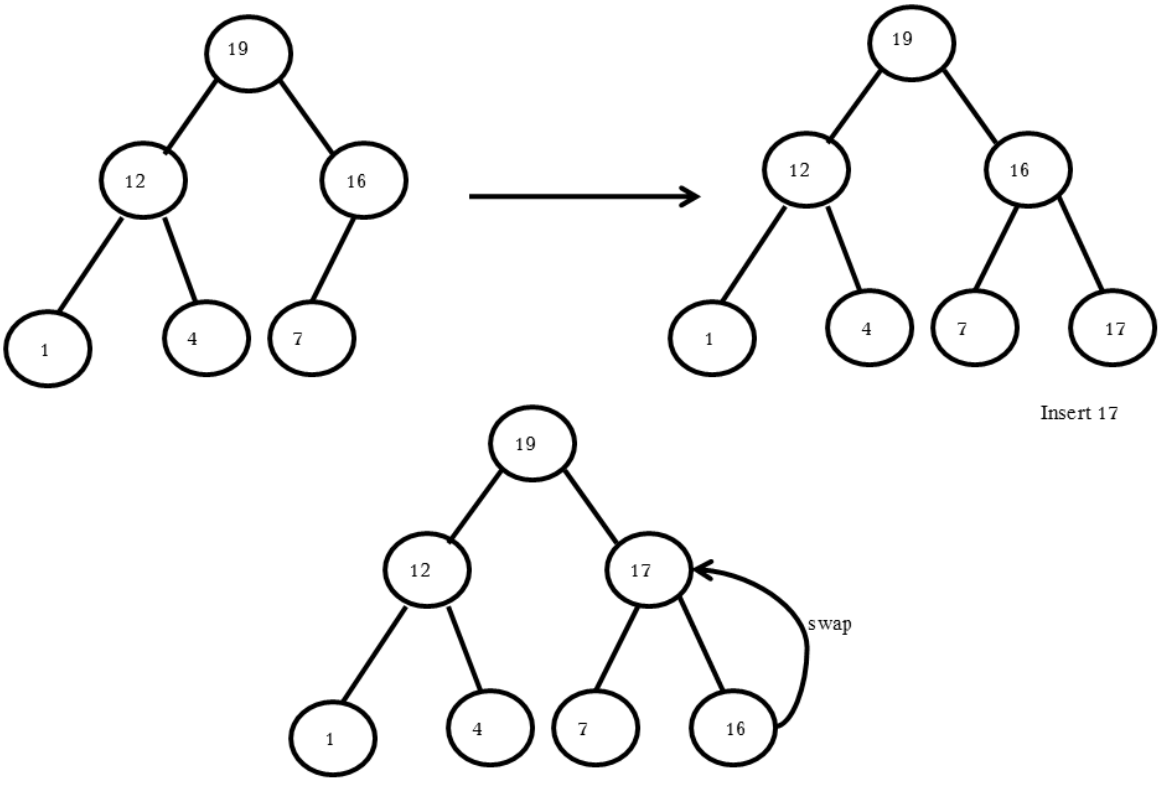
5.8.1 Insertion in Heap

The following steps are to be followed while creating or inserting into a heap.

1. Add the new element to the next available position at the lowest level
2. Restore the max-heap property if violated: General strategy is percolate up (or bubble up): if the parent of the element is smaller than the element, then interchange the parent and child.

   OR

   Restore the min-heap property if violated: General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent and child.



5.8.2 Deletion in Heap

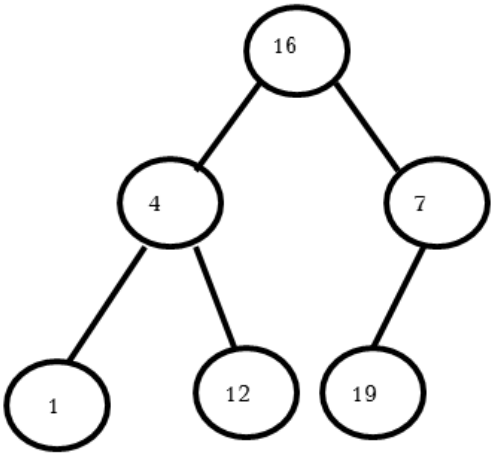
The following steps are to be followed while deleting a node from a heap.

1. Delete max
   - Copy the last number to the root (overwrite the maximum element stored there).
   - Restore the max heap property by percolate down.
2. Delete min
   - Copy the last number to the root (overwrite the minimum element stored there).
   - Restore the min heap property by percolate down.

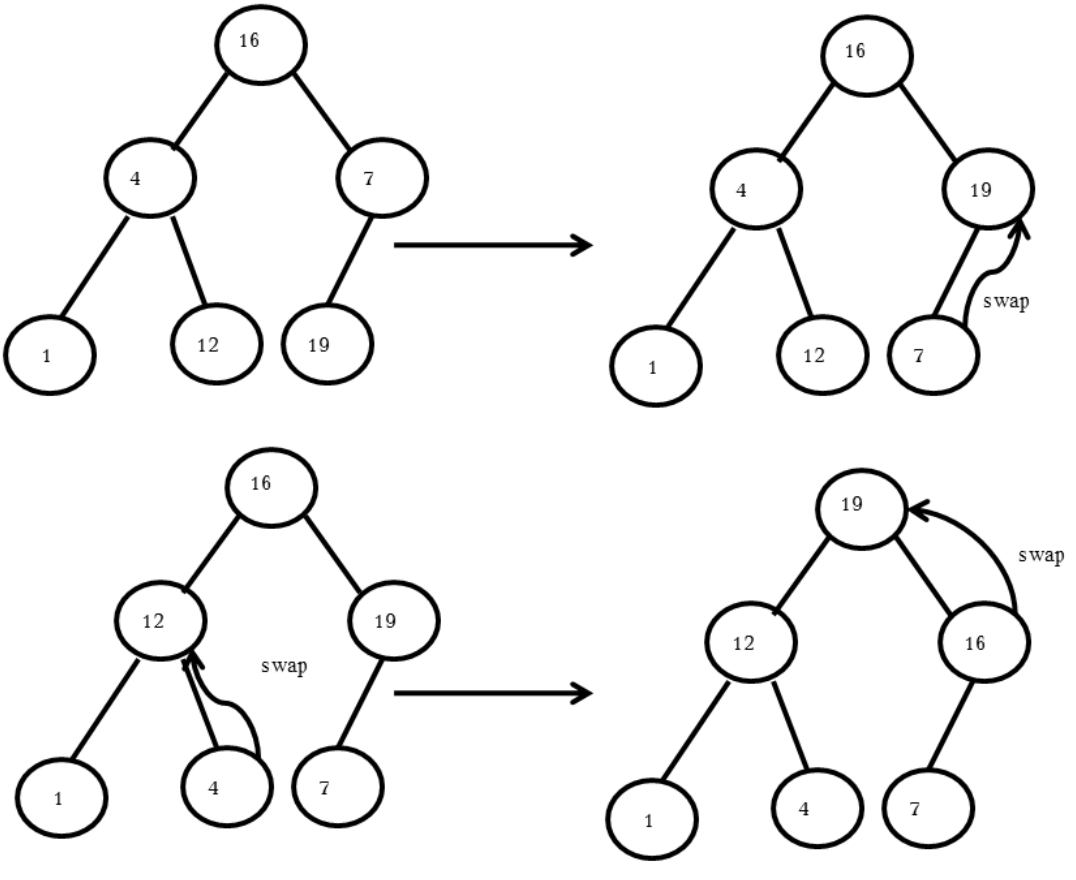
5.8.3 Creating a Heap for Sorting - Heapify

Consider the following numbers: 16, 4, 7, 1, 12, 19

Using the numbers, the following complete binary tree is created.



Now, convert the binary tree into heap. To sort the elements in the decreasing order, create a min heap and to sort the elements in the increasing order, create a max heap. This process of creating min or max heap in order to maintain the property of heap is called Heapify. Here, we are creating a Max Heap.



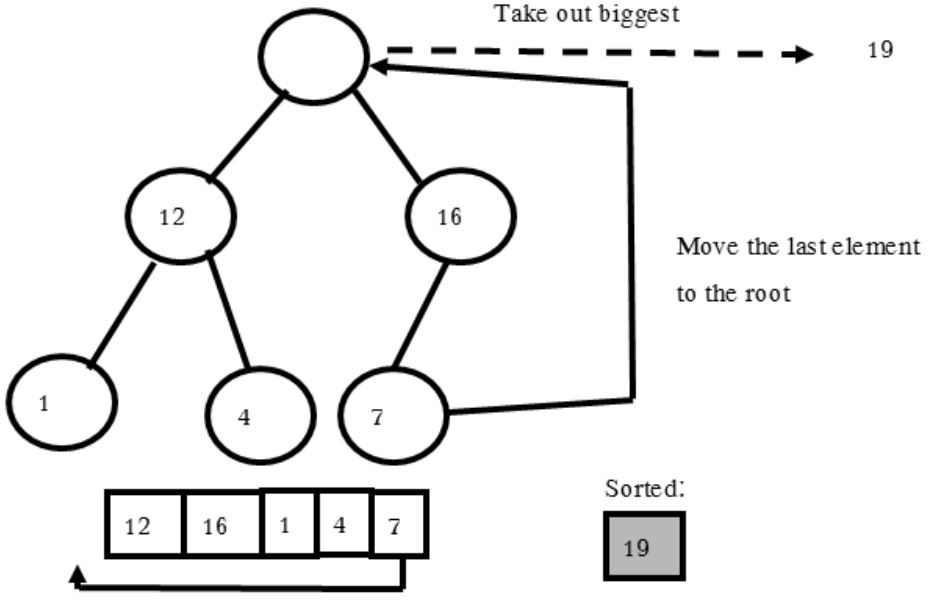
5.8.4 Sorting a Heap

Following is the algorithm to perform Heap sort.

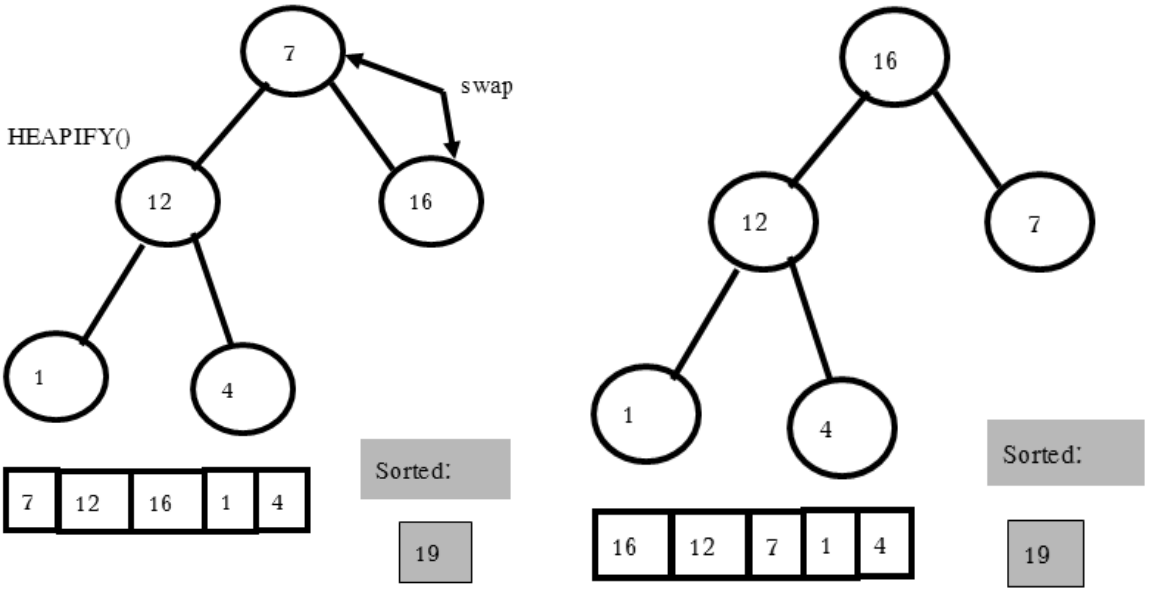
1. *Construct a Complete Binary Tree with given list of Elements.*
2. *Transform the Binary Tree into Min/Max Heap.*
3. *Delete the root element from Min/Max Heap and Put the deleted element into the Sorted list.*
4. *Remove the last leaf element and make it as root.*
5. *Heapify the Min/Max heap, if required, by selecting lowest or greatest child of root ans swap them.*
6. *Repeat steps 3 to 5 until Min/Max Heap becomes empty.*
7. *Display the sorted list.*

Example 1: Consider the following array with elements: 16, 4, 7, 1, 12, 19

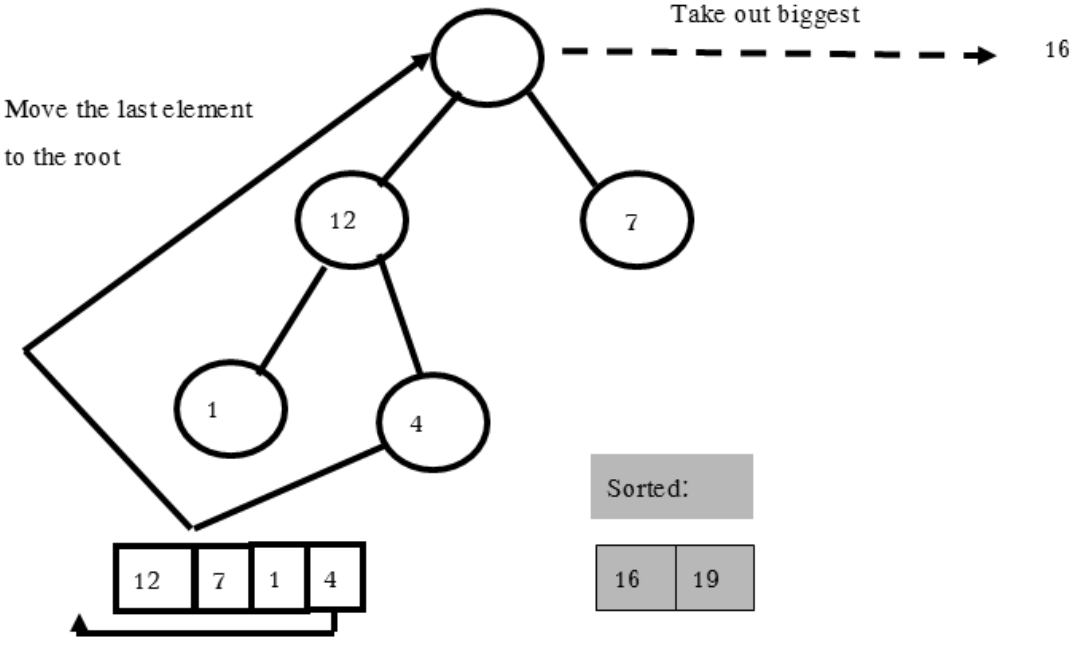
Step 1: Remove 19 and add it to bottom of the array.



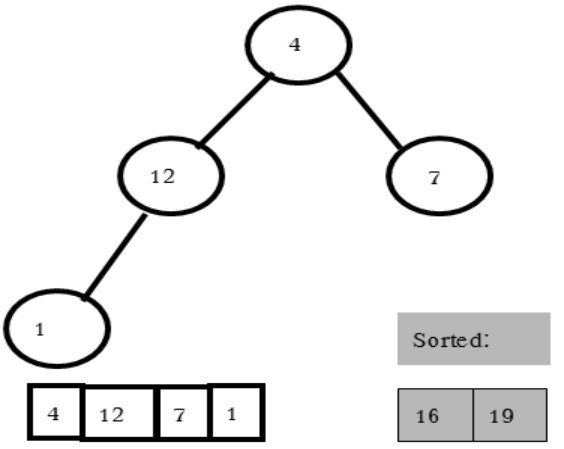
Heapify it so that it remains max heap. For this, swap largest child with root. Here, 7 and 16 are swapped.

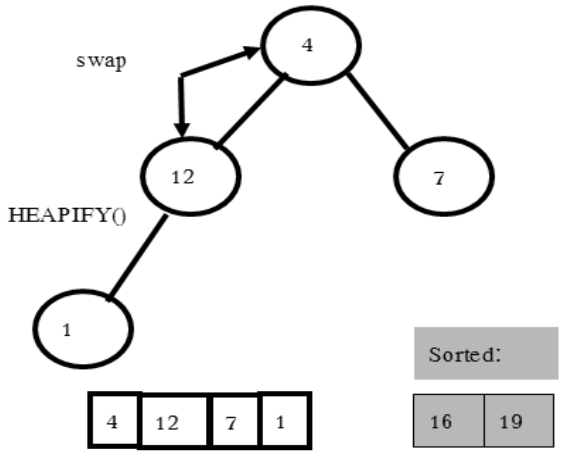


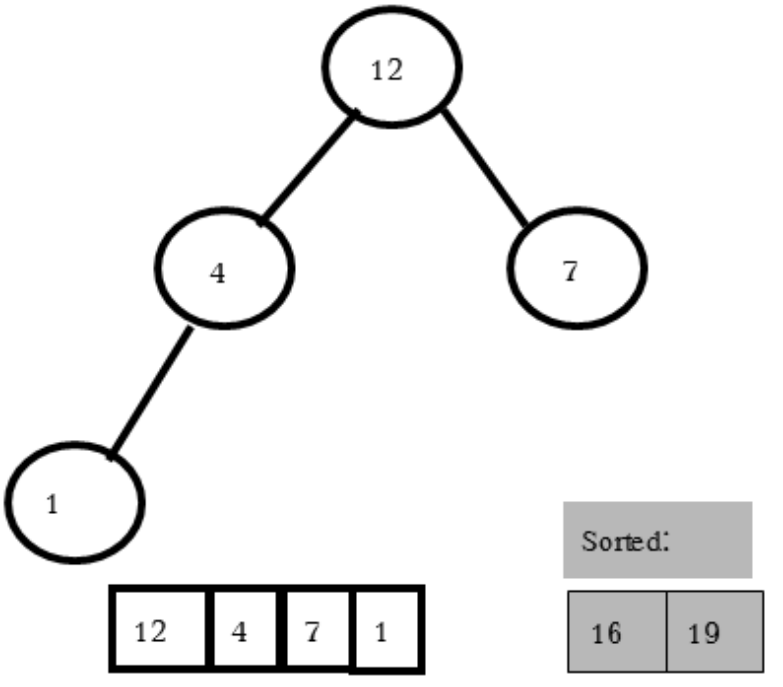
Step 2: Remove 16 and add it second last position of the array.



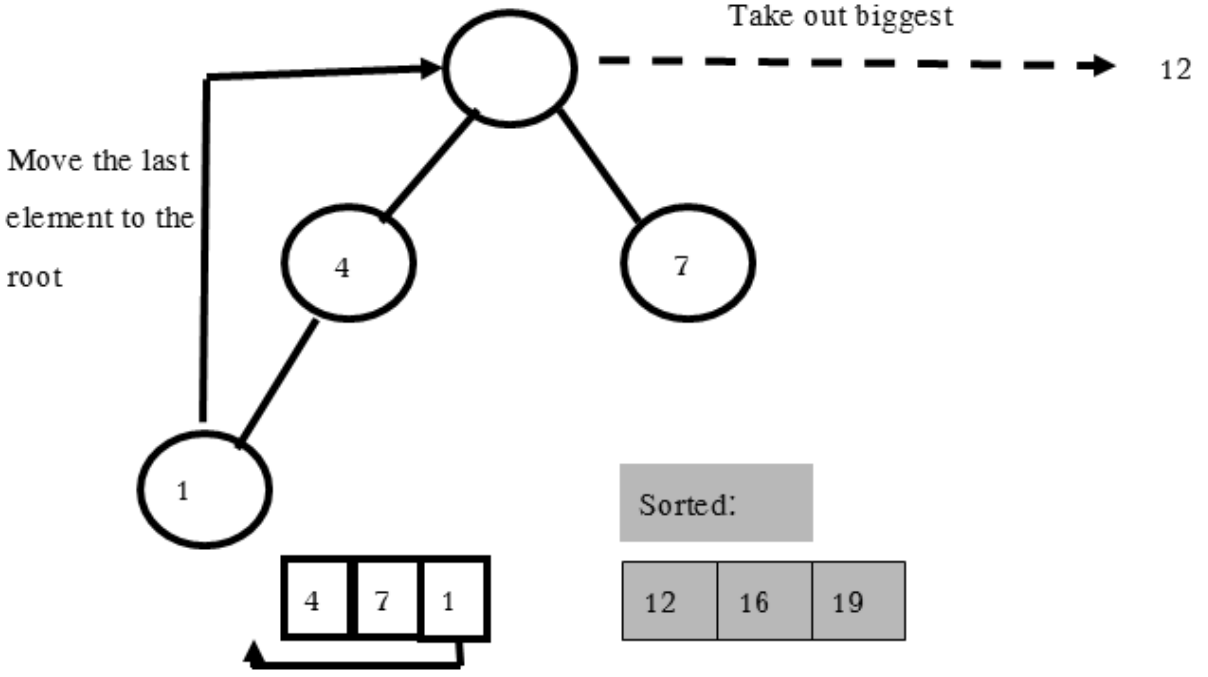
Heapify it so that it remains max heap. For this, swap largest child with root. Here, 4 and 12 are swapped.


4
12
7
1
4
12
7
1
Sorted:
16
19

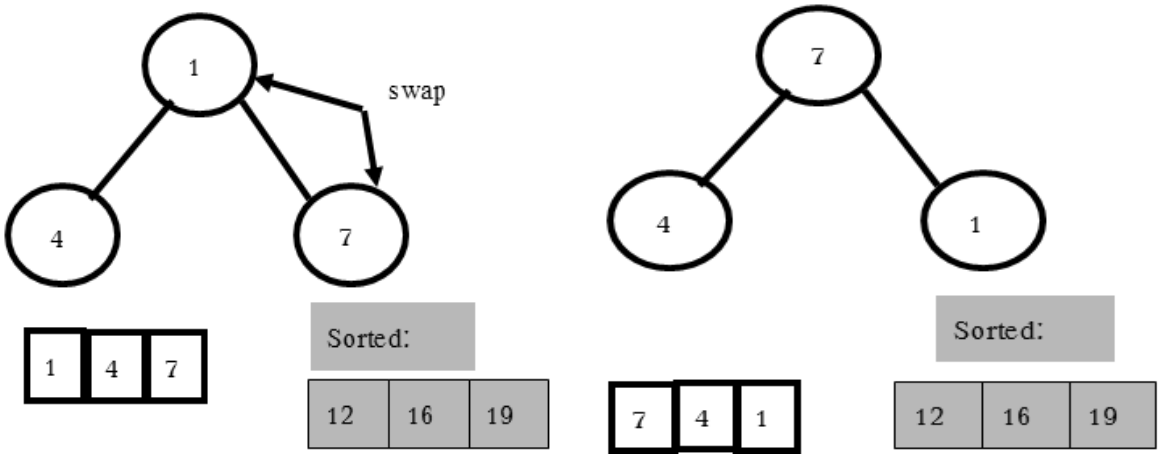

swap
4
12
7
HEAPIFY()
1
4
12
7
1
Sorted:
16
19


12
4
7
1
12
4
7
1
Sorted:
16
19

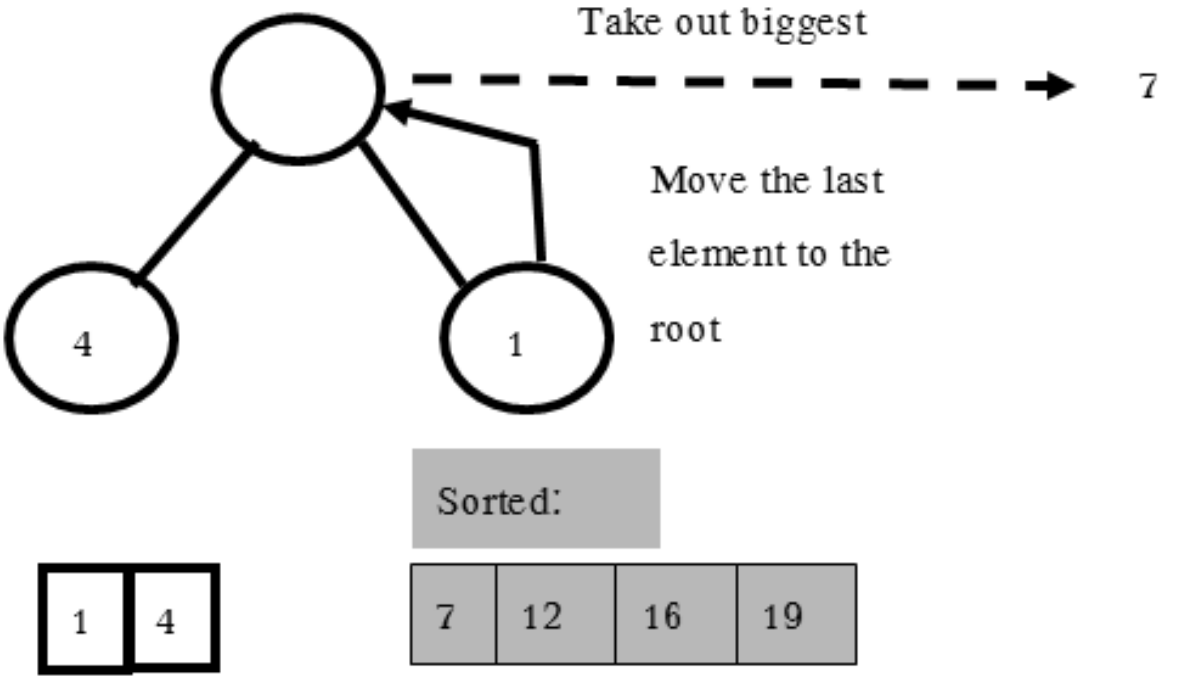
Step 3: Remove 12 and add it third last position of the array.



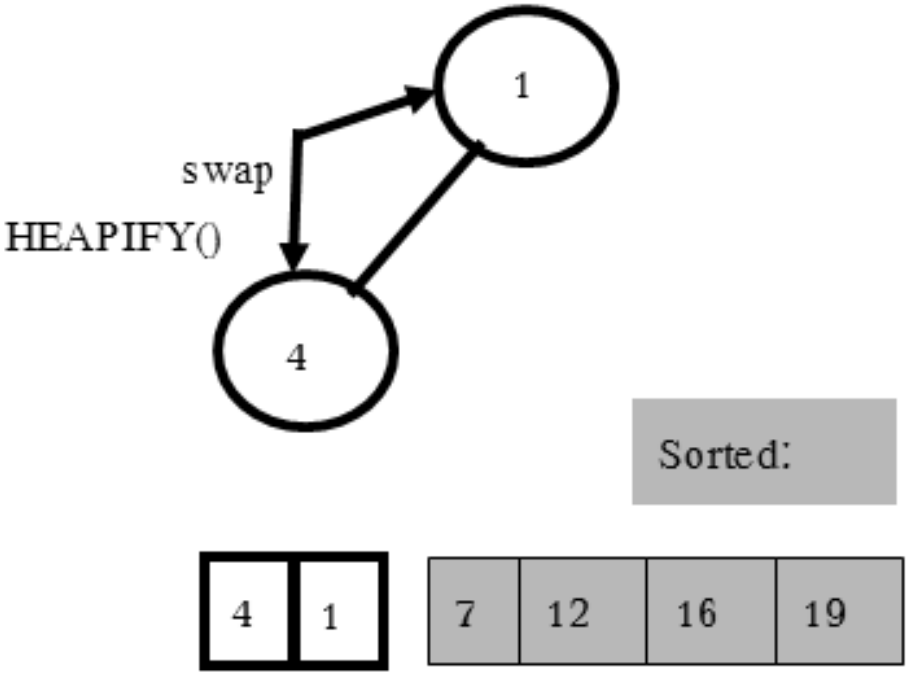
Heapify it so that it remains max heap. For this, swap largest child with root. Here, 1 and 7 are swapped.



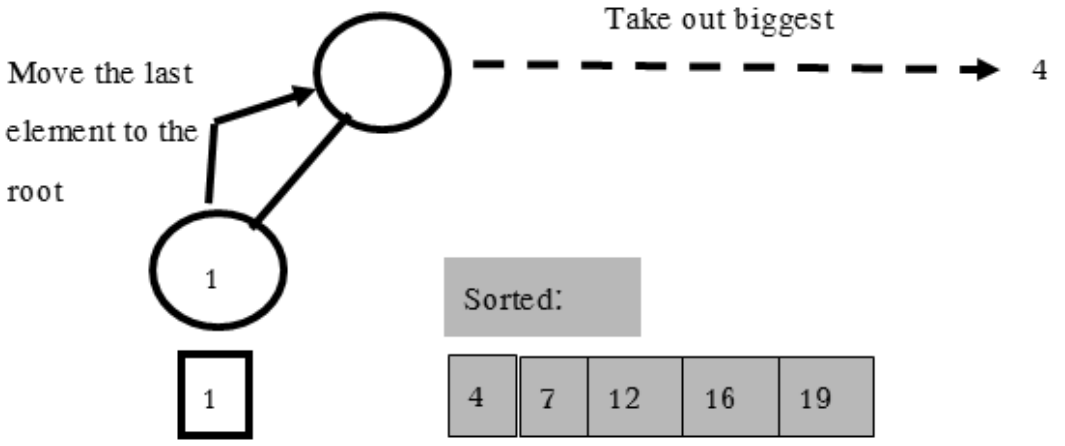
Step 4: Remove 7 and add it fourth last position of the array.



Heapify it so that it remains max heap. For this, swap largest child with root. Here, 1 and 4 are swapped.



Step 5: Remove 4 and add it fifth last position of the array.



Step 6: Remove 1 and add it into array.

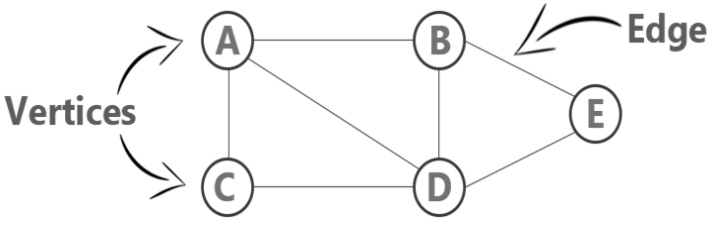
Finally, we have the sorted array as shown above.

6. GRAPHS

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. To be specific, a graph G is a discrete structure consisting of nodes (called vertices) and lines joining the nodes (called edges). Two vertices are adjacent to each other if they are joint by an edge. The edge joining the two vertices is said to be an edge incident with them. We use V (G) and E(G) to denote the set of vertices and edges of G respectively.

6.1 INTRODUCTION TO GRAPH

The following is a graph with 5 vertices and 6 edges. This graph G can be defined as G=(V,E), where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



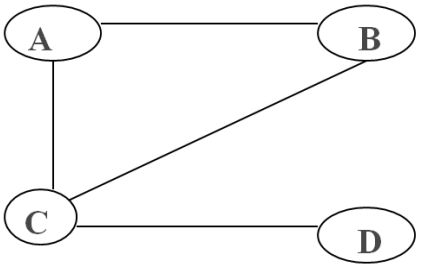
Vertex: Individual data element of a graph is called as Vertex. Vertex is also known as node**.** In above example graph, A, B, C, D & E are known as vertices.

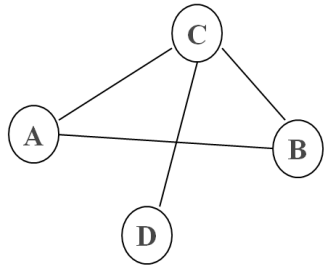
Edge: An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

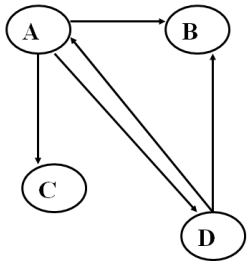
1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge -** A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted egde is a edge with value (cost) on it.

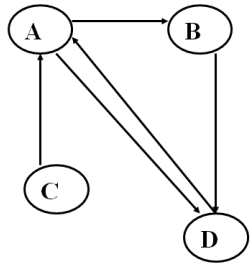
Undirected Graph: A graph with only undirected edges is said to be undirected graph. It can be represented in any way. For e.g. consider the following two graphs representing the same set of vertices A,B,C,D and set of edges: AB, AC, BC, CD





Directed Graph: A graph with only directed edges is said to be directed graph.





Adjacent: If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

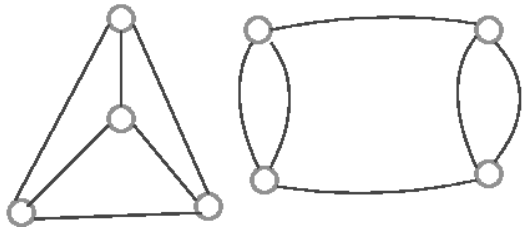
Incident: Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge: A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge: A directed edge is said to be incoming edge on its destination vertex.

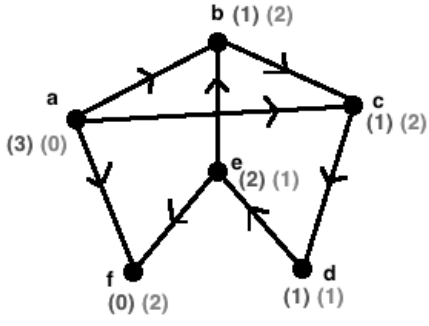
Degree: Total number of edges connected to a vertex is said to be degree of that vertex.

Graph with degree 2



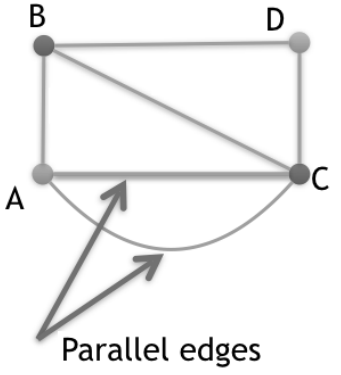
Graph with degree 3

Indegree and Outdegree: Total number of incoming edges connected to a vertex is said to be indegree of that vertex. Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

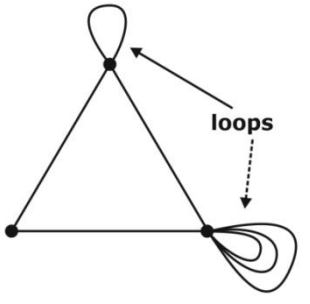


The first value in bracket shows out degree and second value shows in degree.

Parallel edges or Multiple edges: If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.



Self-loop or Loop: Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.



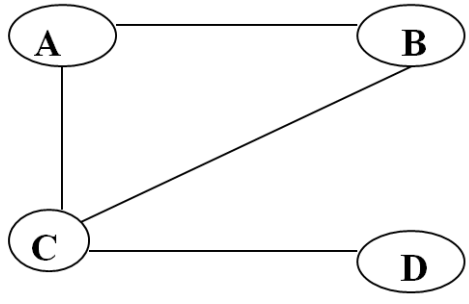
Path: A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

A B C

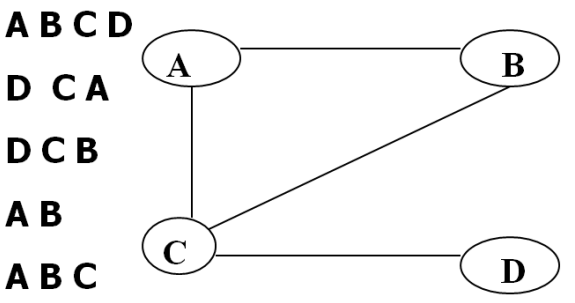
B A C D

A B C A B C A B C D

B A C B C

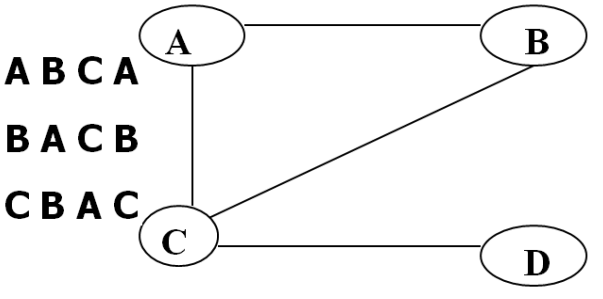


Simple Path: A path in which no vertices are repeated.



Closed Path: A path will be called as closed path if the initial node is same as terminal node.

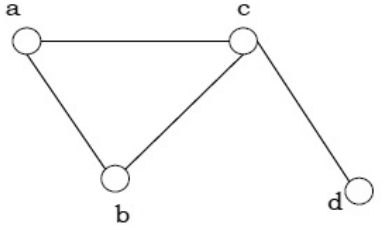
Cycle: A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.



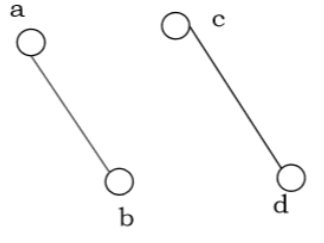
Simple Graph: A graph is said to be simple if there are no parallel and self-loop edges.

Connected Graph: A connected graph is the one in which some path exists between every two vertices. There are no isolated nodes in connected graph.

Disconnected Graph: A connected graph is the one in which at least two vertices are not connected.

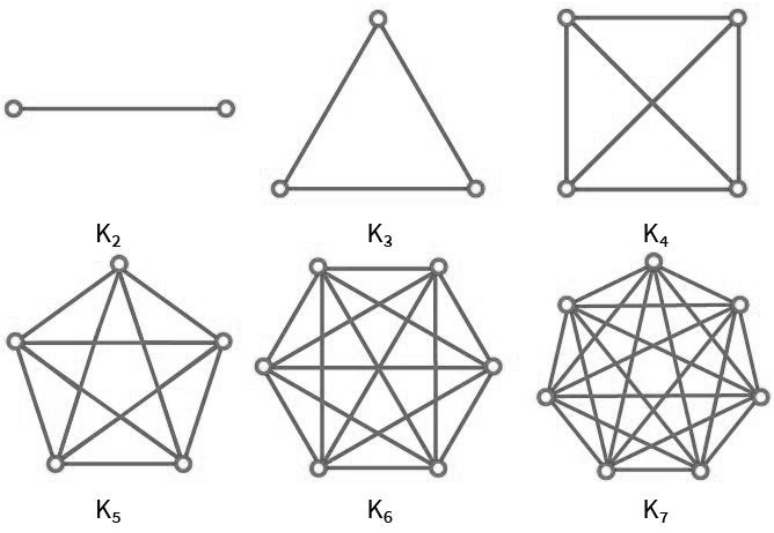


Connected Graph



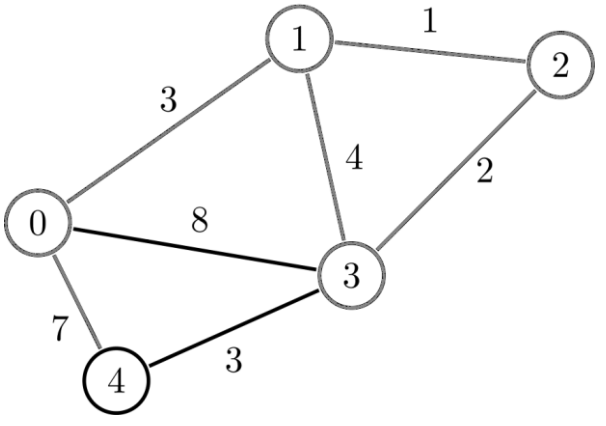
Disconnected Graph

Complete Graph: A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

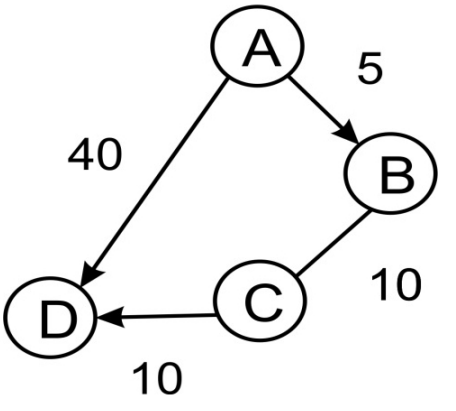


Weighted Graph: In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Networks: directed weighted graphs are called networks.



Weighted Graph



Network

6.2 GRAPH REPRESENTATION

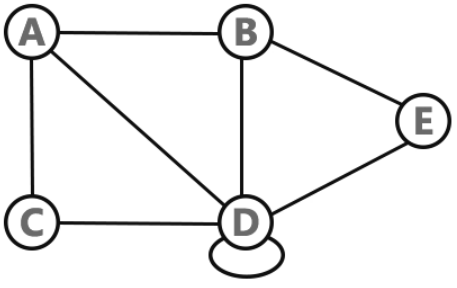
Graph data structure is represented using following representations.

1. Adjacency Matrix
2. Incidence Matrix
3. Adjacency List

6.2.1 Adjacency Matrix

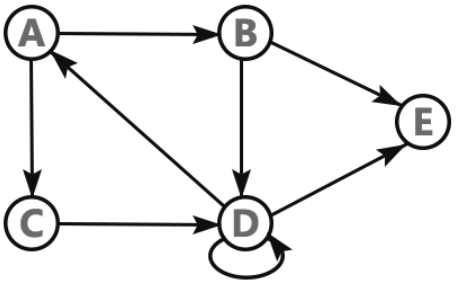
In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation.



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

For the directed graph, the representation becomes little different.



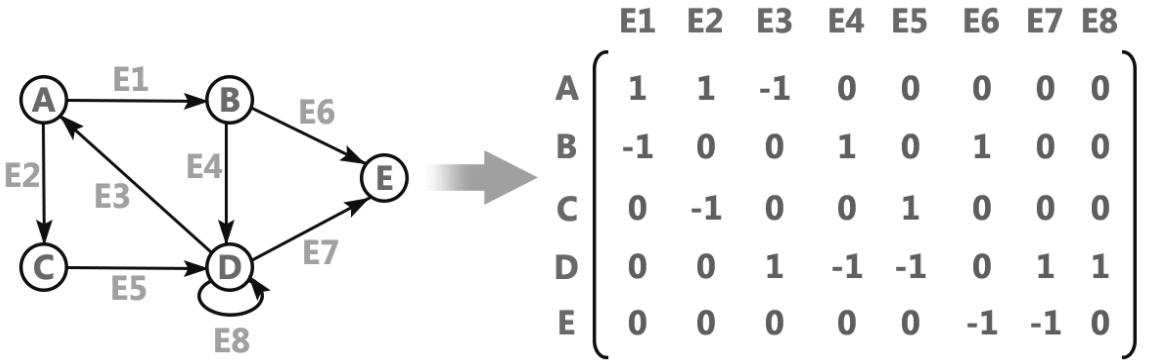
	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

6.2.2 Incidence Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns

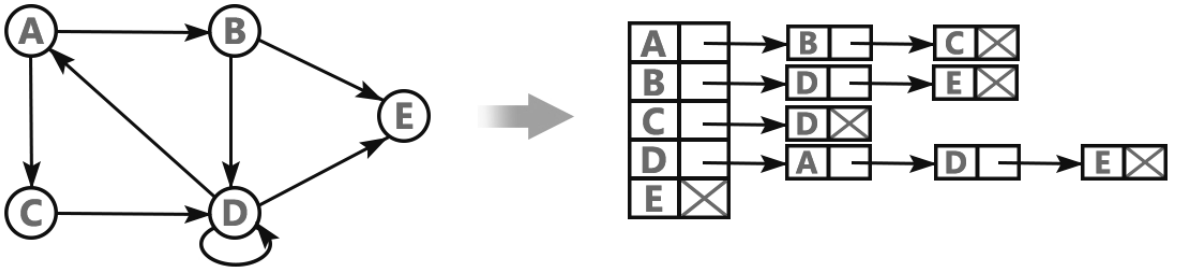
represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

For example, consider the following directed graph representation.

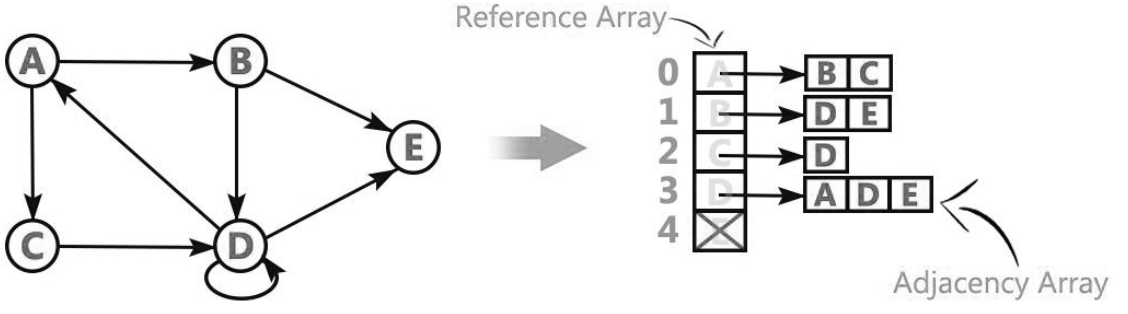


6.2.3 Adjacency List

In this representation, every vertex of a graph contains list of its adjacent vertices. For example, consider the following directed graph representation implemented using linked list.



This representation can also be implemented using an array as follows..



6.3 GRAPH TRAVERSALS

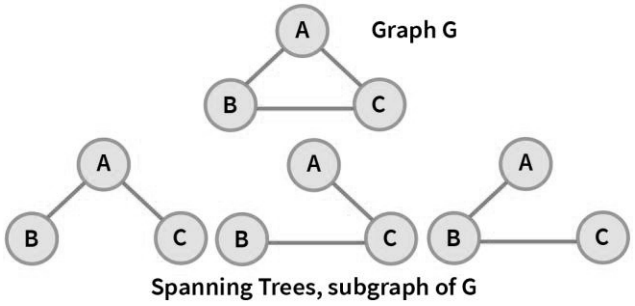
Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path. There are two graph traversal techniques and they are as follows.

1. DFS (Depth First Search)
2. BFS (Breadth First Search)

6.3.1 Depth First Traversal

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. So, let us understand what is a spanning tree. A spanning tree is any tree that consists solely of edges in G and that includes all the vertices. E(G): T(tree edges) + N(nontree edges) where T: set of edges used during search and N: set of remaining edges

A sub-graph that contains all the vertices, and no cycles. If we add any edge to the spanning tree, it forms a cycle, and the tree becomes a graph.



Spanning Trees, subgraph of G

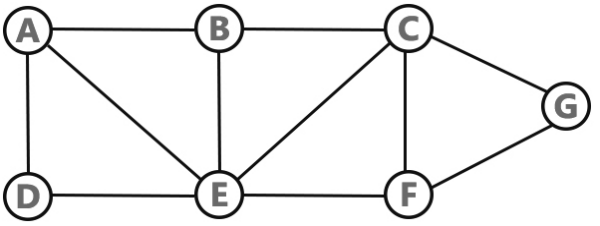
Now let us go for DFS traversal. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

Following are steps to for DFS traversal.

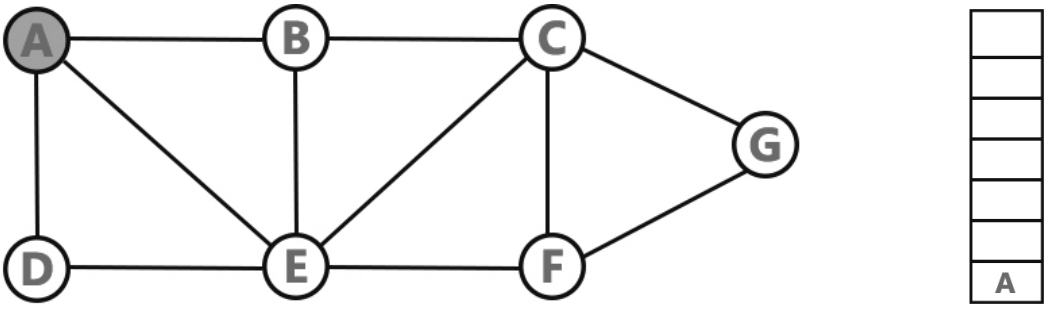
1. *Define a Stack of size total number of vertices in the graph.*
2. *Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.*
3. *Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.*
4. *Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.*
5. *When there is no new vertex to visit then use back tracking and pop one vertex from the stack.* ***Back tracking*** *is coming back to the vertex from which we reached the current vertex.*
6. *Repeat steps 3, 4 and 5 until stack becomes Empty.*
7. *When stack becomes Empty, then produce final spanning tree by removing unused*

edges from the graph.

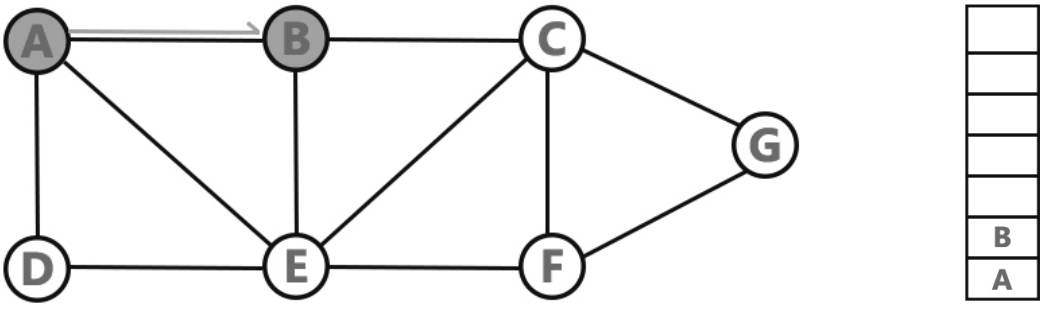
Example 1: Consider the following graph for DFS.



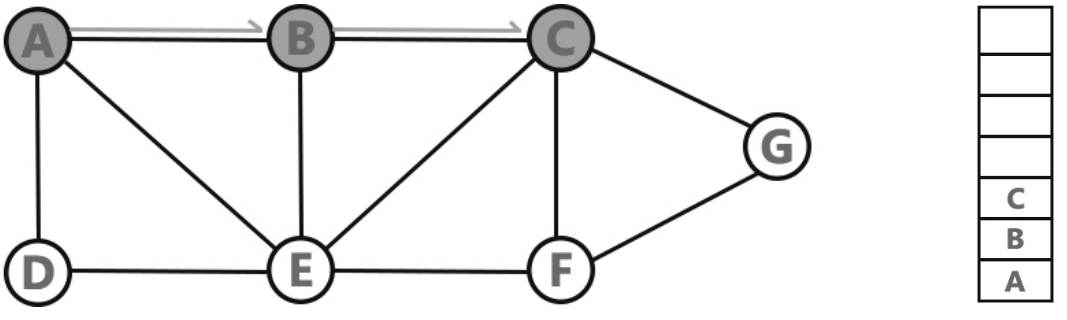
Step 1: Select vertex A as a starting point (Visit A). Push A on to the stack.



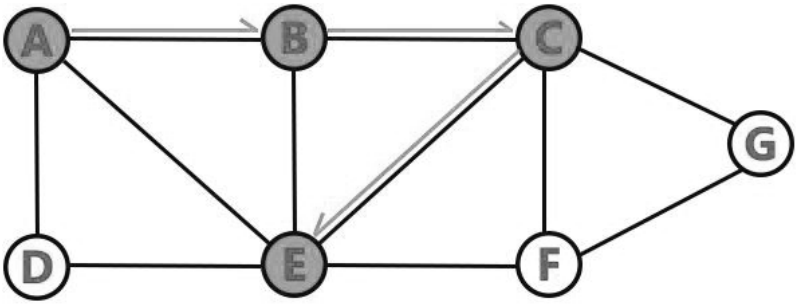
Step 2: Visit any adjacent vertex of A which is not visited (B). Push B on to the stack.

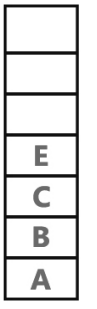


Step 3: Visit any adjacent vertex of B which is not visited (C). Push C on to the stack.

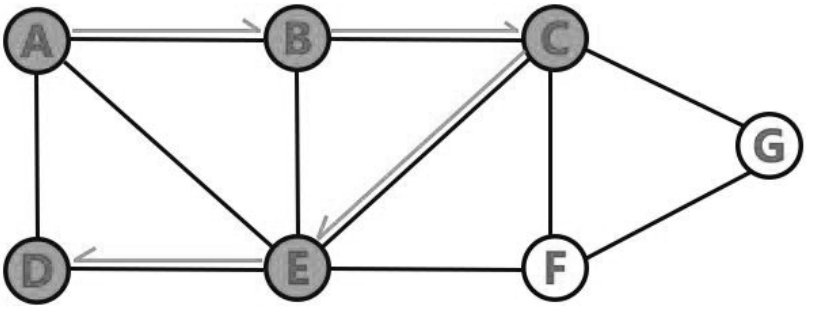


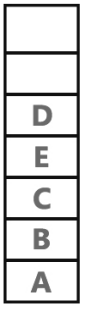
Step 4: Visit any adjacent vertex of C which is not visited (E). Push E on to the stack.



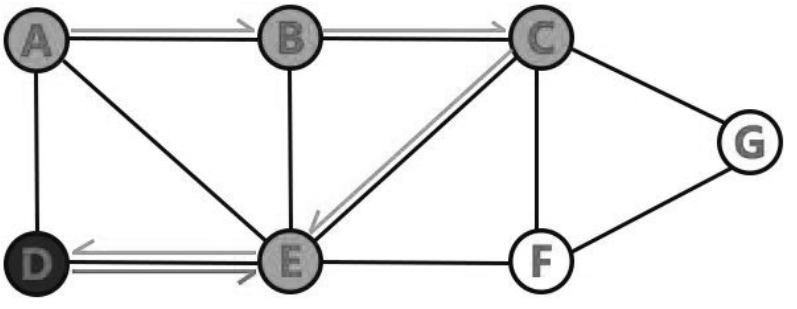


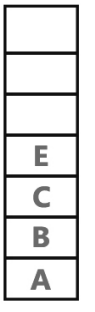
Step 5: Visit any adjacent vertex of E which is not visited (D). Push D on to the stack.



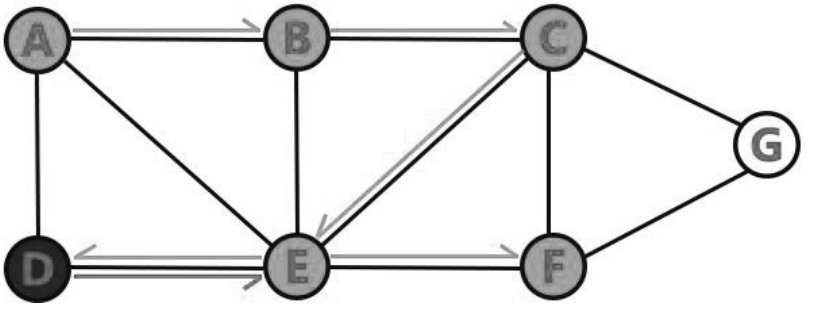


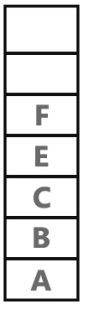
Step 6: There is no new vertex to be visited from D. So use back tracking and pop D from stack.



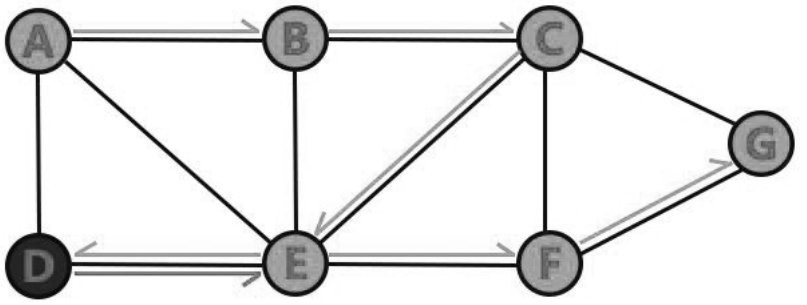


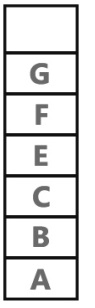
Step 7: Visit any adjacent vertex of E which is not visited (F). Push F on to the stack.



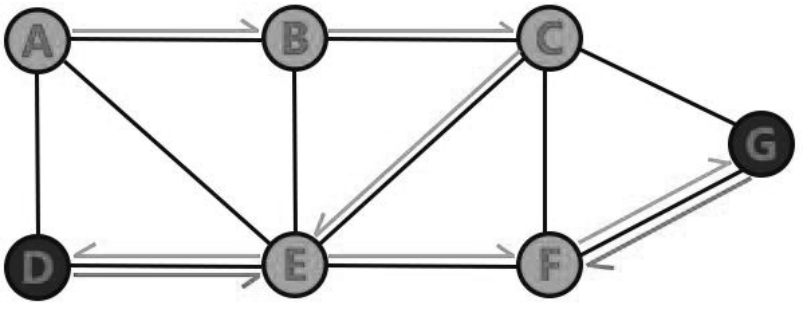


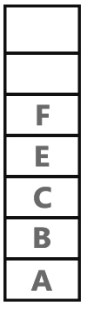
Step 8: Visit any adjacent vertex of F which is not visited (G). Push G on to the stack.



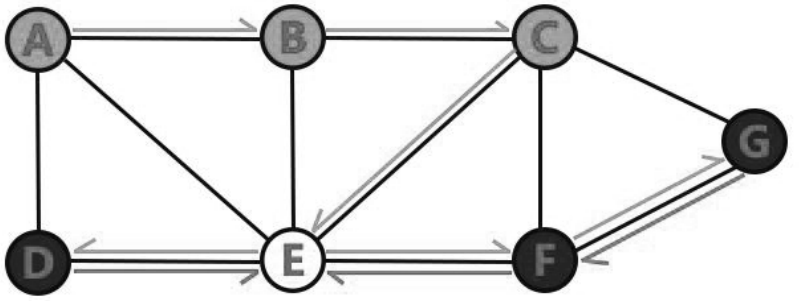


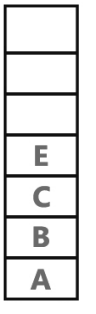
Step 9: There is no new vertex to be visited from G. So use back tracking and pop G from stack.



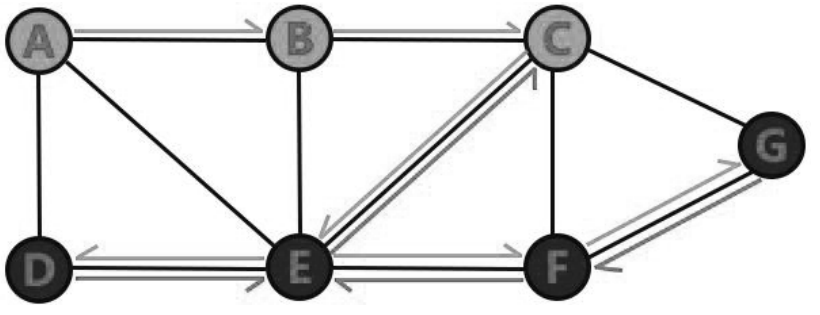


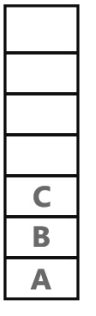
Step 10: There is no new vertex to be visited from F. So use back tracking and pop F from stack.



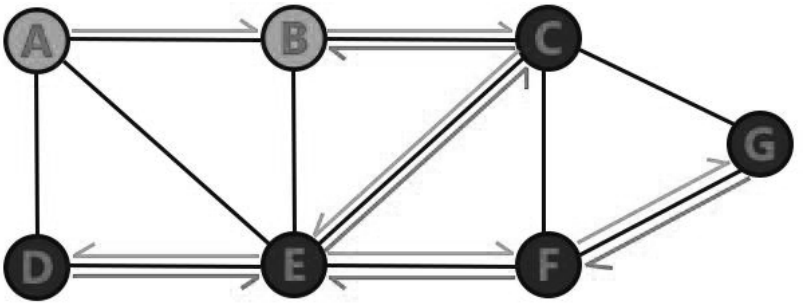


Step 11: There is no new vertex to be visited from E. So use back tracking and pop E from stack.

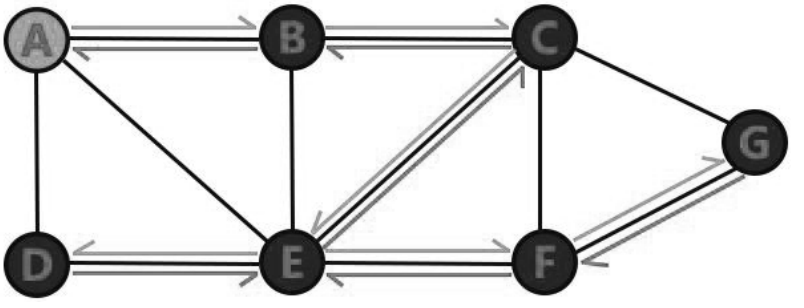




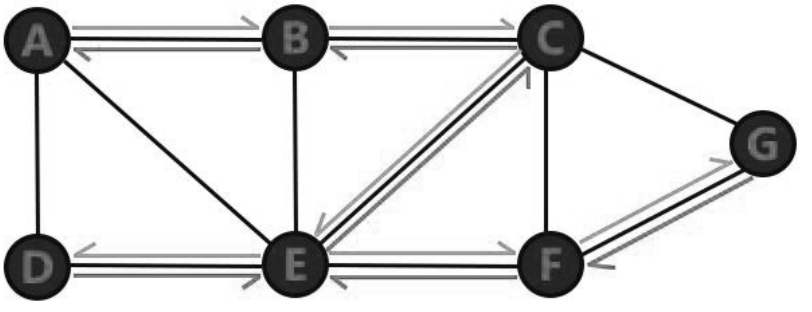
Step 12: There is no new vertex to be visited from C. So use back tracking and pop C from stack.



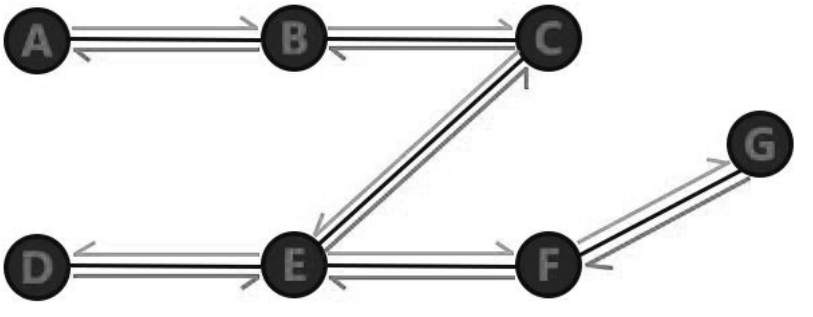
Step 13: There is no new vertex to be visited from B. So use back tracking and pop B from stack.



Step 14: There is no new vertex to be visited from A. So use back tracking and pop A from stack.



Stack is now empty. So DFS traversal is over. The final spanning tree of DFS traversal is shown below:

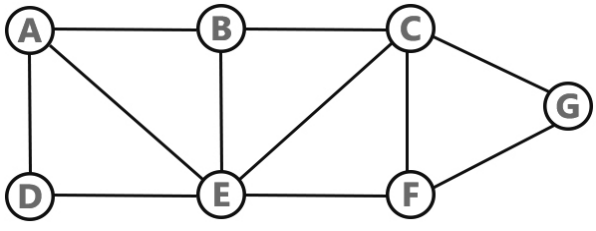


6.3.2 Breadth First Traversal

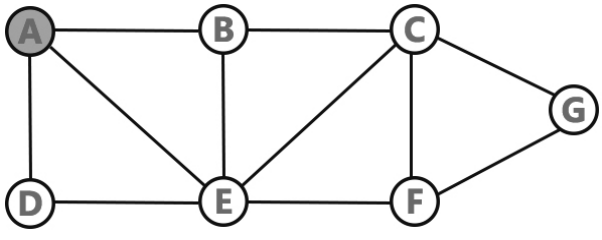
BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal. We use the following steps to implement BFS traversal.

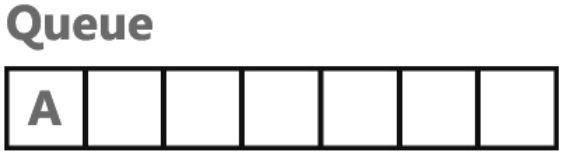
1. Define a Queue of size total number of vertices in the graph.
2. Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
3. Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.
4. When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
5. Repeat steps 3 and 4 until queue becomes empty.
6. When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Example 2: Consider the same graph we took for DFS.

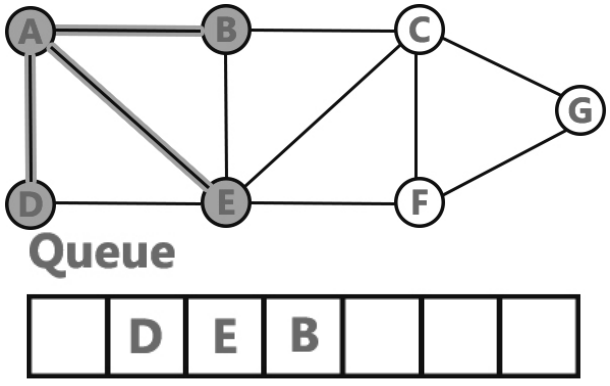


Step 1: Select vertex A as a starting point (Visit A). Insert A into queue.

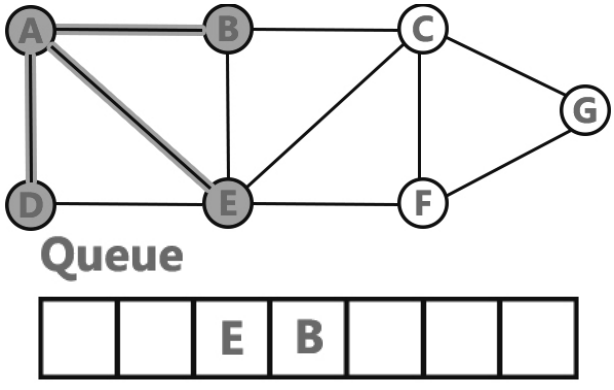




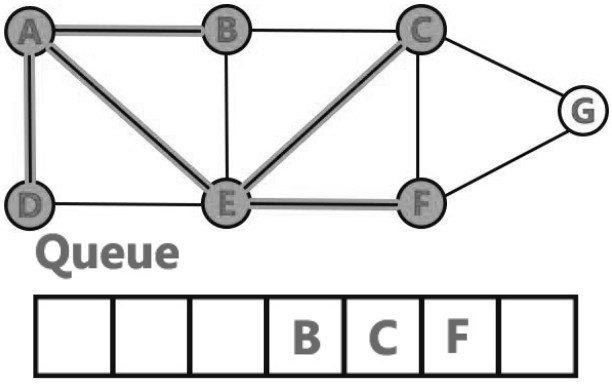
Step 2: Visit all adjacent vertices of A which are not visited (D, E, B). Insert newly visited vertices into queue and delete A from queue.



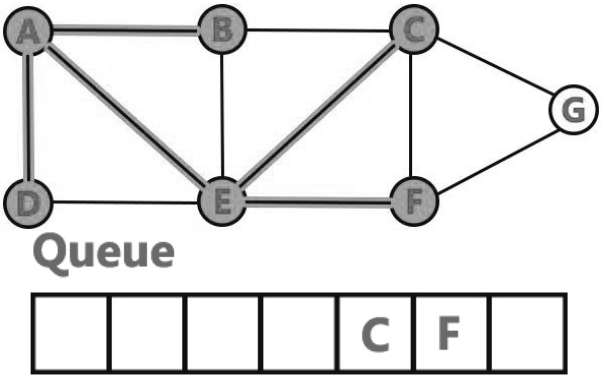
Step 3: Visit all adjacent vertices of D which are not visited (No vertex is left out for visit). Delete D from queue.



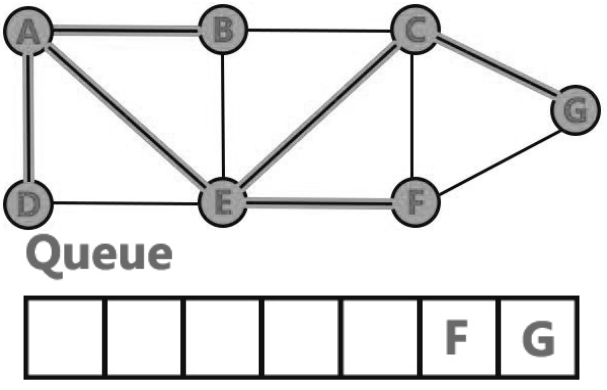
Step 4: Visit all adjacent vertices of E which are not visited (C, F). Insert newly visited vertices into queue and delete E from queue.



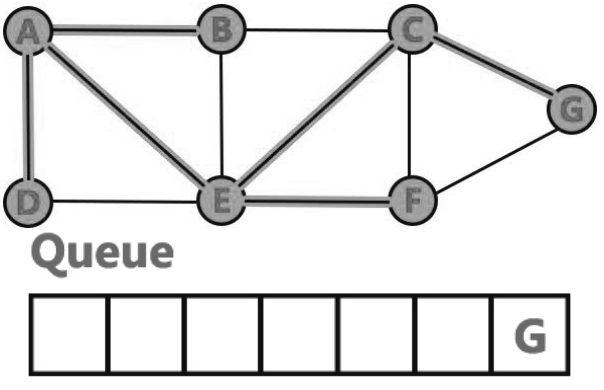
Step 5: Visit all adjacent vertices of B which are not visited (No vertex is left out for visit). Delete B from queue.



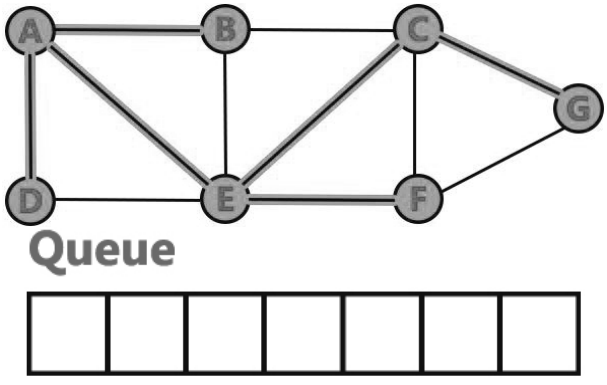
Step 6: Visit all adjacent vertices of C which are not visited (G). Insert newly visited vertices into queue and delete C from queue.



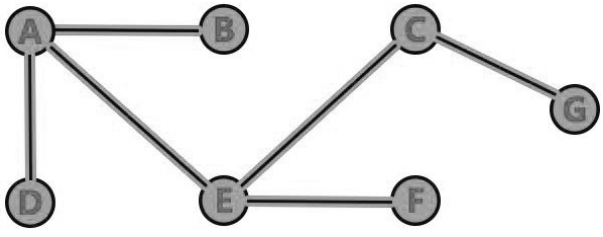
Step 7: Visit all adjacent vertices of F which are not visited (No vertex is left out for visit). Delete F from queue.



Step 8: Visit all adjacent vertices of G which are not visited (No vertex is left out for visit). Delete G from queue.



Queue becomes empty now. So DFS traversal is over. The final spanning tree of BFS traversal is shown below:

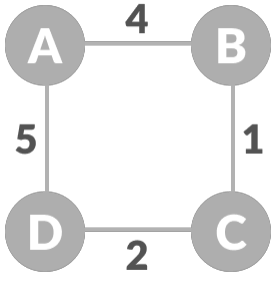


6.3.3 Minimum Spanning Tree

A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight. Spanning trees are useful in,

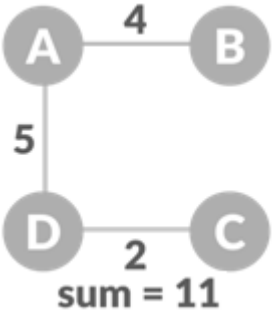
- Computer Network Routing Protocol
- Cluster Analysis
- Civil Network Planning

Example 3: Consider the weighted graph given below:

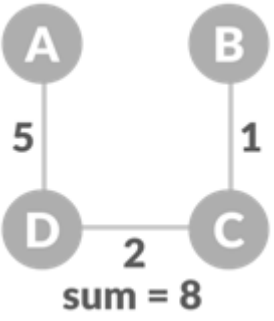


Weighted graph

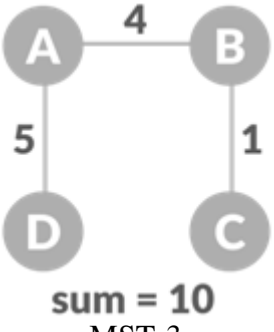
The possible spanning trees from the above graph are:



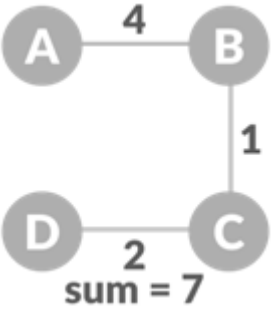
MST-1



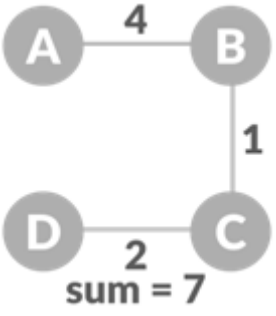
MST-2



MST-3



MST-4



Minimum Spanning Tree

The minimum spanning tree from a graph is found using the following algorithms:

1. Prim's Algorithm
2. Kruskal's Algorithm

Prim's Algorithm

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which form a tree that includes every vertex has the minimum sum of weights among all the trees that can be formed from the graph.

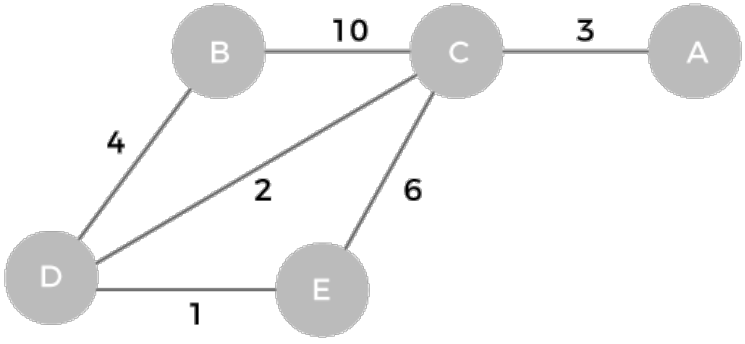
It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from one vertex and keep adding edges with the lowest weight until we reach our goal. The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

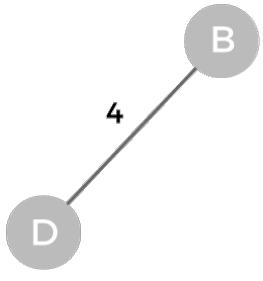
Prim's algorithm can be simply implemented by using the adjacency matrix or adjacency list graph representation, and to add the edge with the minimum weight requires the linearly searching of an array of weights. It requires O(|V|2) running time. It can be improved further by using the implementation of heap to find the minimum weight edges in the inner loop of the algorithm.

Example 4: Consider the following weighted graph.

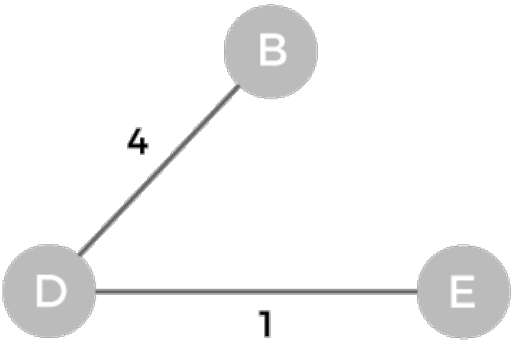


Step 1: First, we have to choose a vertex from the above graph. Let's choose B.

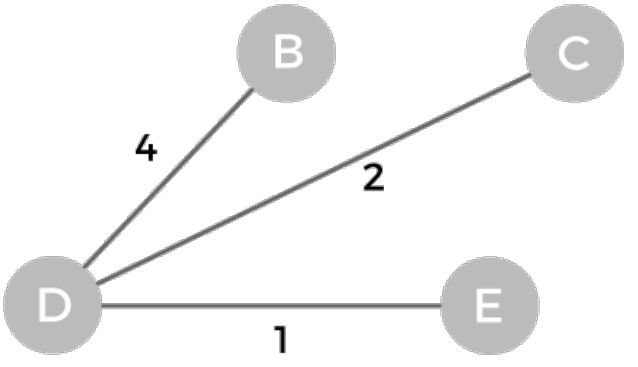
Step 2: Now, we have to choose and add the shortest edge from vertex B. There are two edges from vertex B that are B to C with weight 10 and edge B to D with weight 4. Among the edges, the edge BD has the minimum weight. So, add it to the MST.



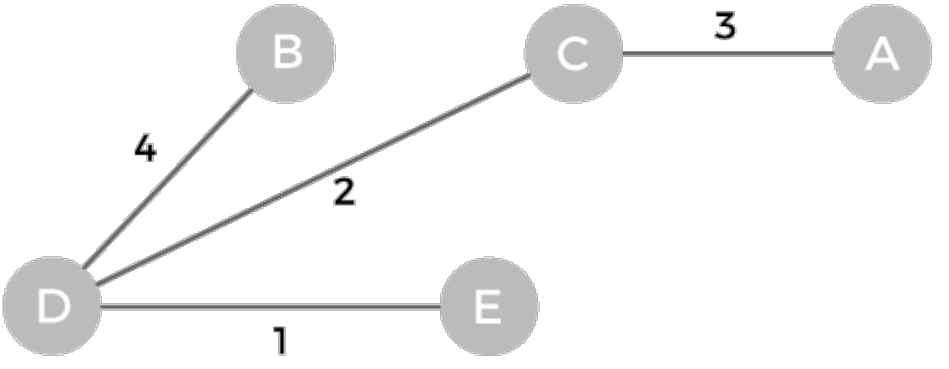
Step 3: Now, again, choose the edge with the minimum weight among all the other edges. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C, i.e., E and A. So, select the edge DE and add it to the MST.



Step 4: Now, select the edge CD, and add it to the MST.



Step 5: Now, choose the edge CA. Here, we cannot select the edge CE as it would create a cycle to the graph. So, choose the edge CA and add it to the MST.



So, the graph produced in step 5 is the minimum spanning tree of the given graph. The cost of the MST is given below -

Cost of MST = 4 + 2 + 1 + 3 = 10 units.

Kruskal's Algorithm

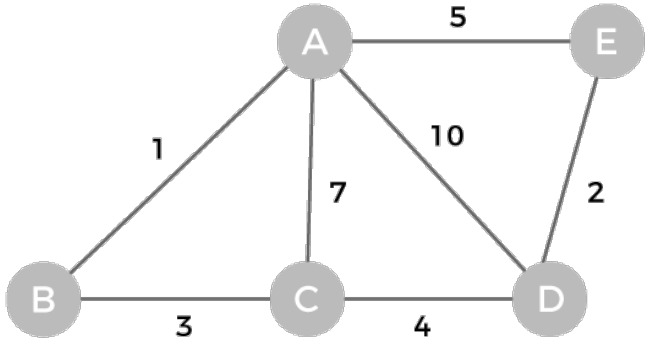
Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which we can traverse every vertex of the graph. It follows the greedy approach that finds an optimum solution at every stage instead of focusing on a global optimum.

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows:

1. First, sort all the edges from low weight to high.
2. Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
3. Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

The time complexity of Kruskal's algorithm is O(E logE) or O(V logV), where E is the no. of edges, and V is the no. of vertices.

Example 5: Consider the following weighted graph.



The weight of the edges of the above graph is given in the below table -

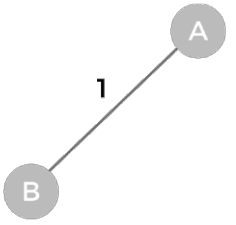
Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

Now, sort the edges given above in the ascending order of their weights.

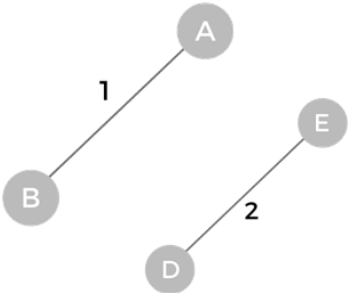
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Now, let's start constructing the minimum spanning tree.

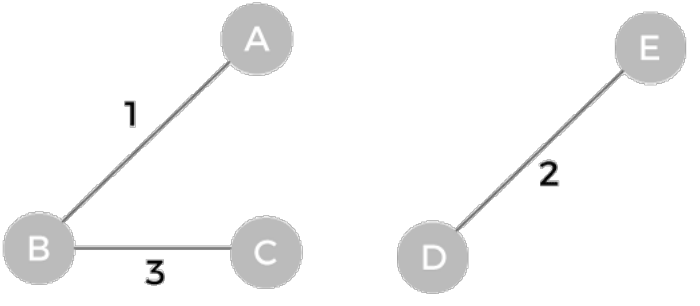
Step 1: First, add the edge AB with weight 1 to the MST.



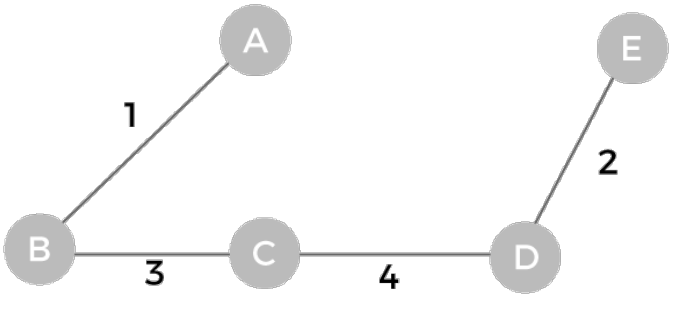
Step 2: Add the edge DE with weight 2 to the MST as it is not creating the cycle.



Step 3: Add the edge BC with weight 3 to the MST, as it is not creating any cycle or loop.



Step 4: Now, pick the edge CD with weight 4 to the MST, as it is not forming the cycle.

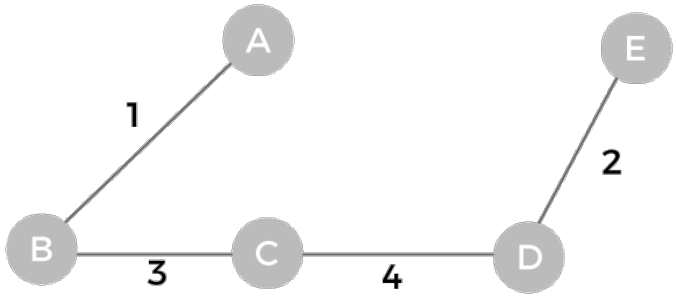


Step 5: After that, pick the edge AE with weight 5. Including this edge will create the cycle, so discard it.

Step 6: Pick the edge AC with weight 7. Including this edge will create the cycle, so discard it.

Step 7: Pick the edge AD with weight 10. Including this edge will also create the cycle, so discard it.

So, the final minimum spanning tree obtained from the given weighted graph by using Kruskal's algorithm is



The cost of the MST is = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

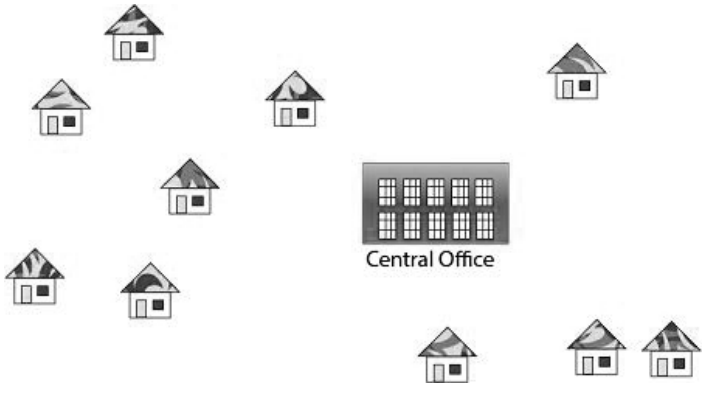
Now, the number of edges in the above tree equals the number of vertices minus 1. So, the algorithm stops here.

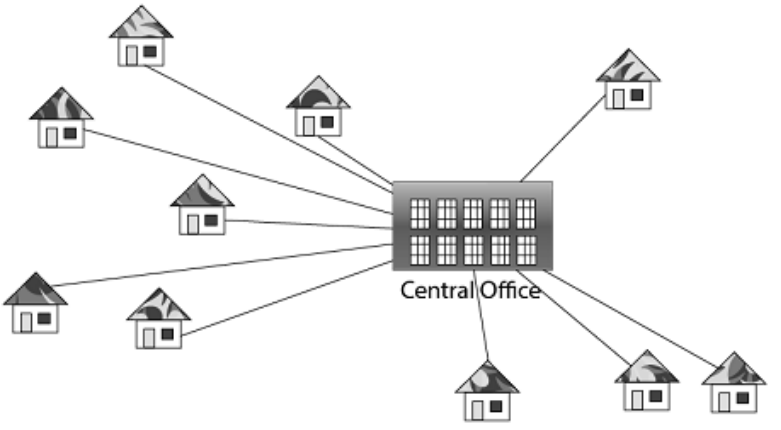
Applications of Minimum Spanning Tree

1. Consider n stations are to be linked using a communication network & laying of communication links between any two stations involves a cost. The ideal solution would be to extract a subgraph termed as minimum cost spanning tree.
2. Suppose you want to construct highways or railroads spanning several cities then we can use the concept of minimum spanning trees.
3. Designing Local Area Networks.
4. Laying pipelines connecting offshore drilling sites, refineries and consumer markets.
5. Suppose you want to apply a set of houses with
   - Electric Power
   - Water
   - Telephone lines
   - Sewage lines

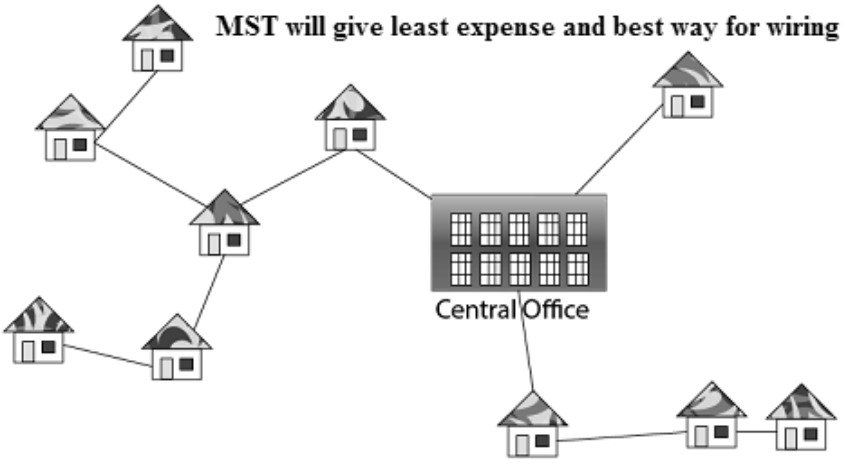
To reduce cost, you can connect houses with minimum cost spanning trees.

For Example, Problem laying Telephone Wire.

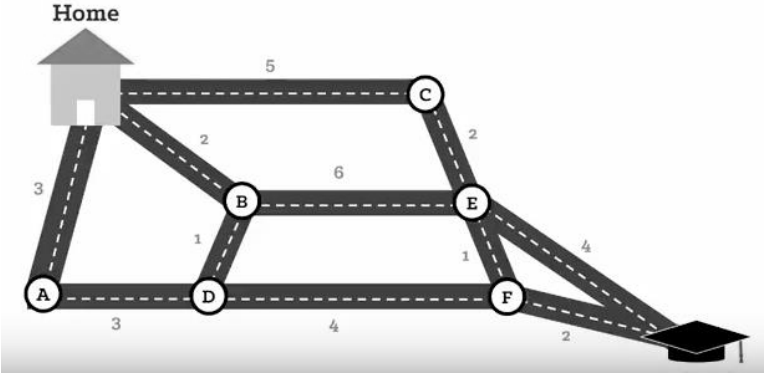




Wiring: Naive Approach will be expensive



6.3.4 Dijkstra's Algorithm for Shortest Path

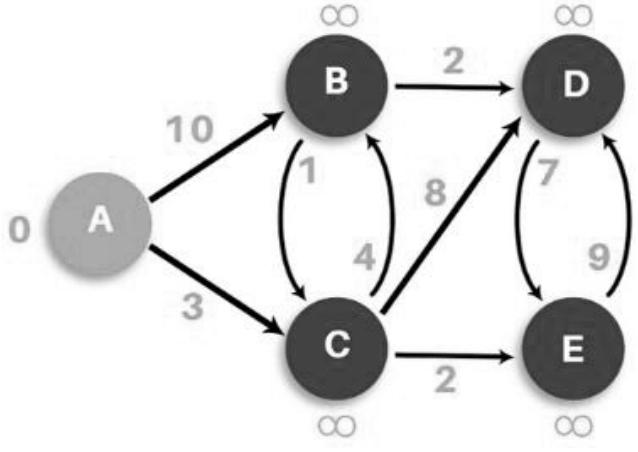


Dijkstra’s algorithm is also known as the shortest path algorithm. It is an algorithm used to find the shortest path between nodes of the graph. The algorithm creates the tree of the shortest paths from the starting source vertex from all other points in the graph. It differs from the minimum spanning tree as the shortest distance between two vertices may not be included in all the vertices of the graph. The algorithm works by building a set of nodes that have a minimum distance from the source. Here, Dijkstra's algorithm uses a greedy approach to solve the problem and find the best solution.

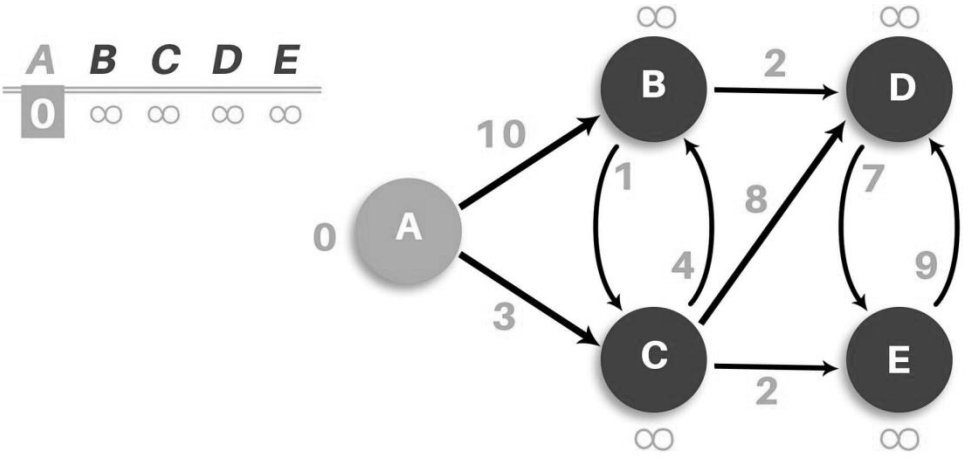
Following are the steps of the algorithm:

1. First of all, we will mark all vertex as unvisited vertex
2. Then, we will mark the source vertex as 0 and all other vertices as infinity
3. Consider source vertex as current vertex
4. Calculate the path length of all the neighboring vertex from the current vertex by adding the weight of the edge in the current vertex
5. Now, if the new path length is smaller than the previous path length then replace it otherwise ignore it
6. Mark the current vertex as visited after visiting the neighbor vertex of the current vertex
7. Select the vertex with the smallest path length as the new current vertex and go back to step 4.
8. Repeat this process until all the vertex are marked as visited.

Example 6: Consider the following weighted graph.



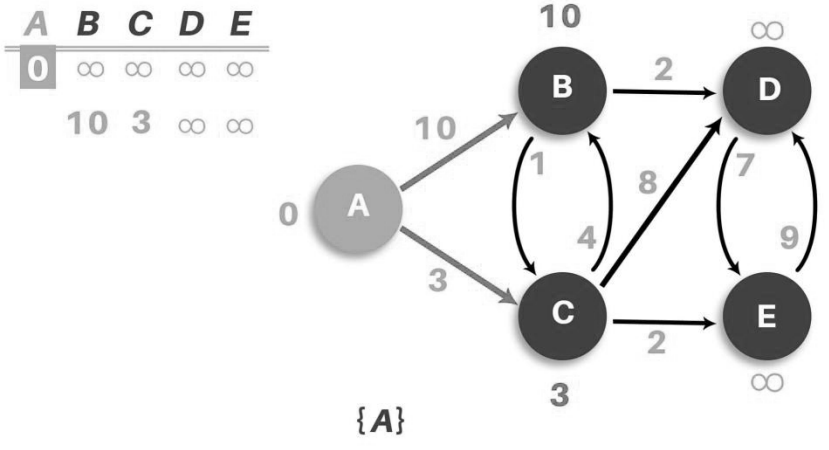
Let us consider the below example to understand the algorithm. Here we are given a weighted graph, and we will choose vertex 'A' as the source vertex of the graph. As the algorithm generates the shortest path from the source vertex to every other vertex, we will set the distance of the source vertex to itself as '0'. The distance from the source vertex to all other vertex is not determined yet, and therefore, we will represent it using infinity. For now, the list of unvisited nodes will be: {B, C, D, E}



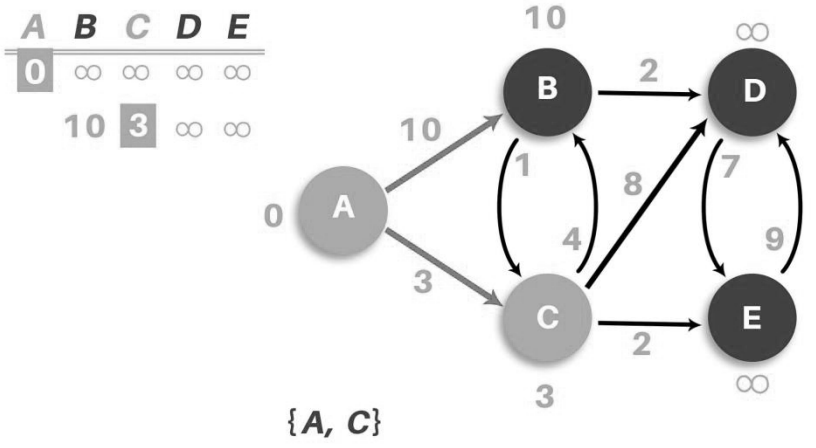
Here, we will start checking the distance from node 'A' to its adjacent vertex. You can see the adjacent vertexes are 'B' and 'C' with weights '10' and '3', respectively. Remember that you don't have to add the two vertexes to the shortest path immediately.

Firstly, we will update the distance from infinity to given weights. Then we have to select the node closest to the source node depending on the updated weights. Mark it as visited and add it to the path.

As shown below, we update vertex B from infinity to 10 and vertex C from infinity to 3.

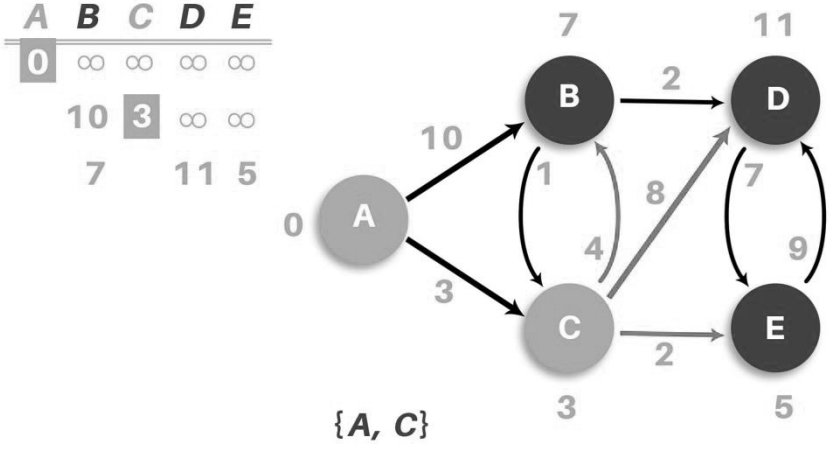


Now select the vertex with the smaller path length as visited vertex and put it in the answer. Therefore, the list of unvisited nodes is {B, D, E}.

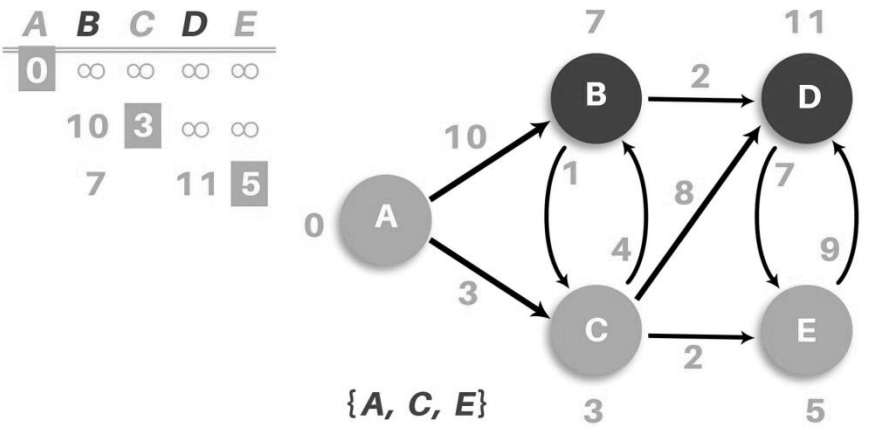


Now, we have to analyze the new adjacent vertex to find the shortest path. So we will visit the neighboring nodes of the visited vertex and update the path lengths as required. Here, we have B, D, and E as adjacent vertex to node 'A' and node 'C.' Therefore, we will update the path of all three vertexes from infinity to their respective weights as shown in the image below.

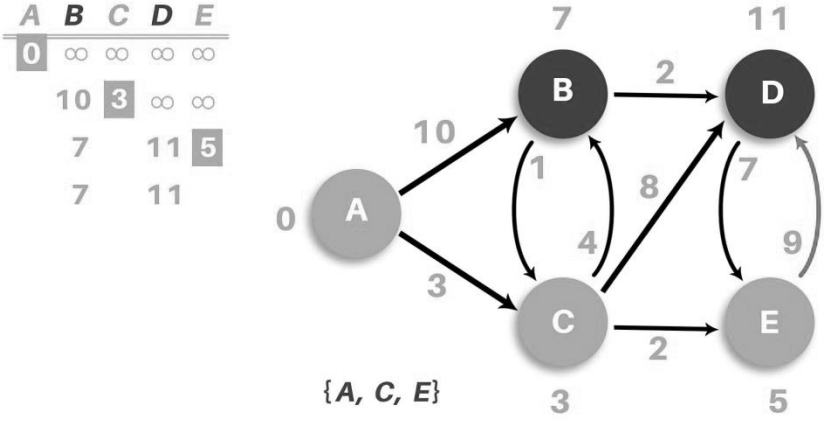
Note that node 'B' is directly connected adjacent to node 'A,' hence, node 'B' weight will be the same as displayed. But for Node 'D' and node 'E,' the path is calculated via node 'C,' and hence the weight of that vertex will be 11 and 5 because we add the weight of the edges from path A->C->D and A->C->E respectively.



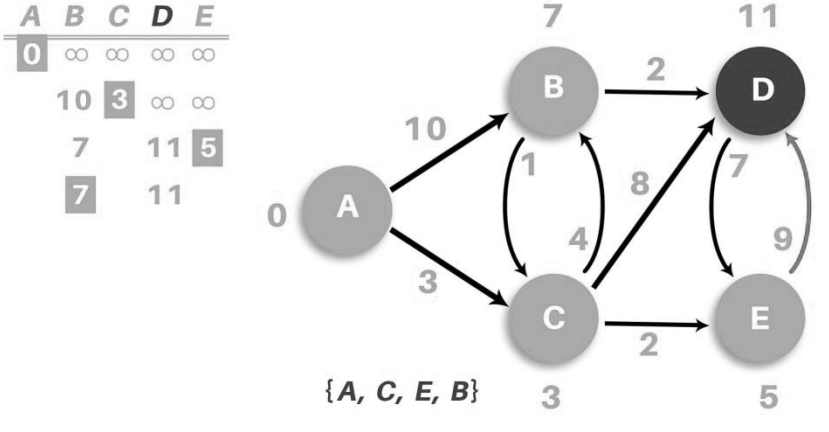
Now, choosing the shortest path from the above table results in choosing the node 'E' with the shortest distance of 5 from the source vertex. And hence the list of unvisited nodes is {B, D}



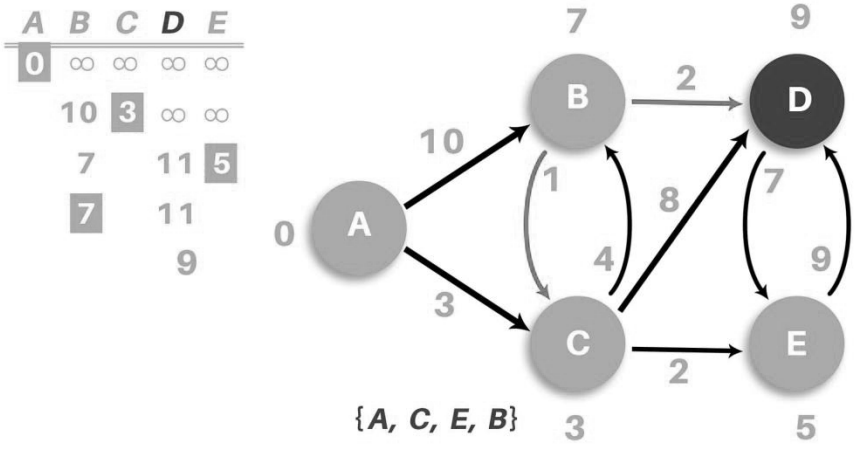
Repeat the process until all vertices have been visited. Here, vertex 'B' and vertex 'D' are both considered adjacent vertex, and the shortest distance of both the vertex from the source node does not change, as shown in the figure below.



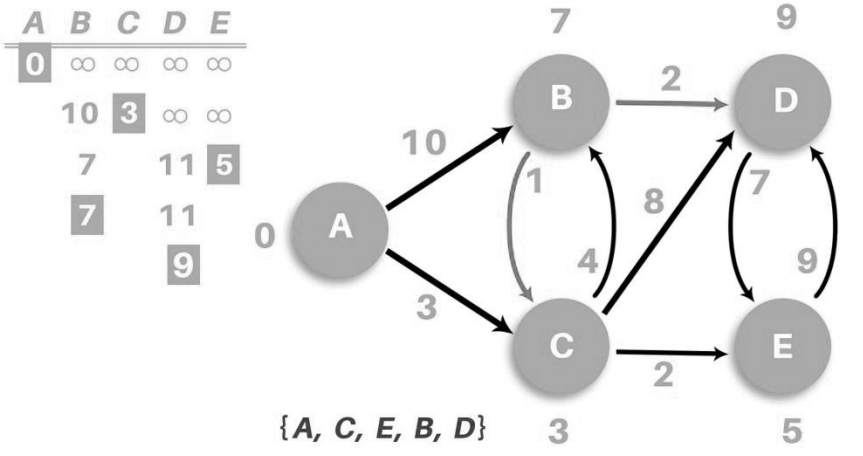
Therefore, the weight of vertex 'B' is minimum compared to vertex 'D,' so we will mark it as a visited node and add it to the path. The list of the unvisited nodes will be {D}.



After visiting vertex 'B,' we are left with visiting vertex 'D.' If you carefully notice, the distance from source vertex to vertex 'D' can be modified from the previous one, i.e., instead of visiting vertex 'D' directly via vertex 'C,' we can visit it via vertex 'B' with the total distance of 9. That is because we add the weight of the edges like A->C->B->D (3+4+2=9) as shown below.



Therefore, the final output of the algorithm will be {A, C, E, B, D}.



7. REFERENCES

Following are the references that have been used to create this book. I am thankful to these contributors for providing these resources.

1. https://www.codingeek.com/category/data-structure/
2. https://www.javatpoint.com/data-structure-tutorial
3. https://www.programiz.com/dsa
4. http://www.btechsmartclass.com/data_structures/
5. https://www.geeksforgeeks.org/data-structures/
6. https://www.tutorialspoint.com/data_structures_algorithms/index.htm
7. https://www.scaler.com/topics/data-structures/