Unit 1 : Topic : 1: Algorithm analysis

What is Algorithm :

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics

- **Unambiguous** − Algorithms should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility (possibility)** − Should be feasible with the available resources.
- **Language independent:** An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

Analysis of Algorithm :

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following –

- **Priori Analysis:** Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm.
  Various factors can be considered before implementing the algorithm like processor speed, which has no effect on the implementation part.
- **Posterior Analysis:** Here, posterior analysis is a practical analysis of an algorithm.

The practical analysis is achieved by implementing the algorithm using any programming language.
This analysis basically evaluates how much running time and space taken by the algorithm.

generally we can bifurcate this in following ways :

Best Case : The minimum number of steps taken to perform the program.

Average Case : Average Number of steps taken to perform the program.

Worst case : The maximum number of steps taken to perform the program.

Complexity of algoritham :

- Algorithmic complexity is a measure of how long an algorithm would take to complete given an input of size n.

- If an algorithm has to scale, it should compute the result within a finite and practical time bound even for large values of n.
- For this reason, complexity is calculated asymptotically as n approaches infinity.
- While complexity is usually in terms of time, sometimes complexity is also analyzed in terms of space, which translates to the algorithm's memory requirements.
- Complexity of an algorithm is a measure of analysis of an algorithm. It is also known as computational complexity.
- Algorithm complexity is a rough approximation of the number of steps, which will be executed depending on the size of the input data.
- Complexity gives the order of steps count, not their exact count. The performance of the algorithm can be measured in two factors:

Time complexity:

- The time complexity of an algorithm is the amount of time required to complete the execution.
- Time complexity is mainly calculated by counting the number of steps to finish the execution in a specific amount of time.
- The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity.

Let's understand the time complexity through an example.

```
int i,sum = 0;
(for i = 0; i<5; i++)
{
        sum+= i;
}
```

- In the above code, the time complexity of the loop statement will be at least n, and if the value of n increases, then the time complexity also increases.
- We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

Space complexity:

- An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.
- For an algorithm, the space is required for the following purposes:
    1. To store program instructions
    2. To store constant values
    3. To store variable values
    4. To track the function calls, jumping statements, etc.

Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

So, **Space complexity = Auxiliary space + Input size.**

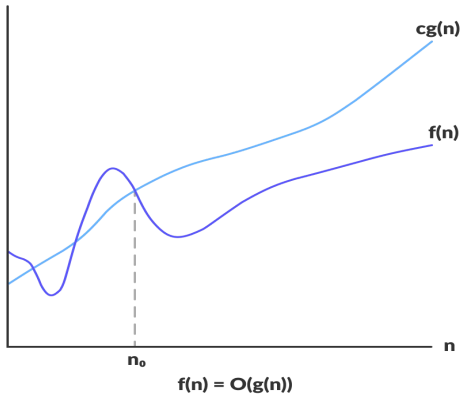
Asymptotic Notation :

- Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n.
- An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.
- The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.
- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.
- For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.
- But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.
- When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.
- The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.
- There are mainly three asymptotic notations:
    - Big-O notation - worst
    - Omega notation - best
    - Theta notation - avg

<u>Big-O notation : - worst - upper</u>

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

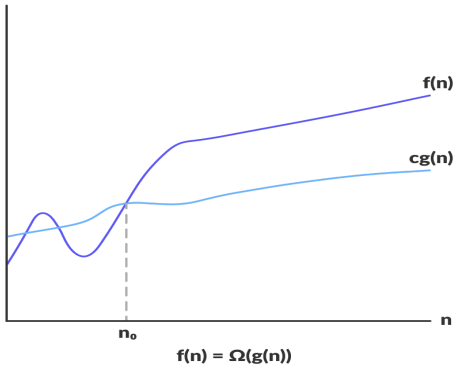
For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).



Omega Notation, Ω - best

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

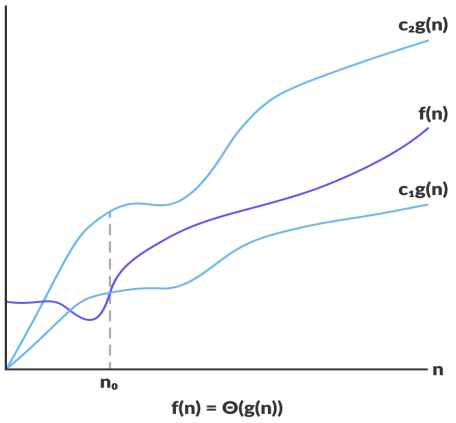
For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).



Theta Notation, θ - avg -

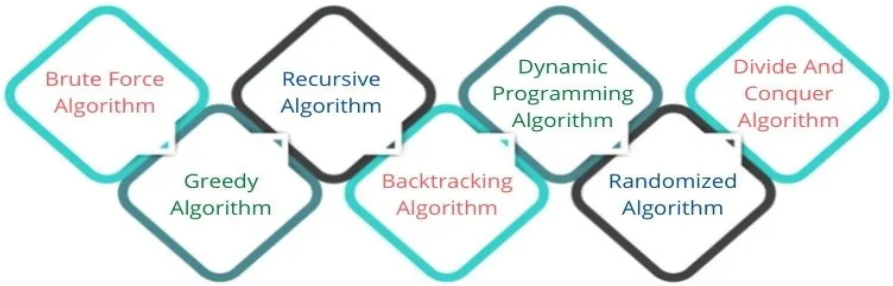
The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.



Classes of an algorithm :

- Brute Force Algorithm
- Recursive Algorithm
- Dynamic Programming Algorithm
- Divide and Conquer Algorithm
- Greedy Algorithm
- Backtracking Algorithm
- Randomized Algorithm



Brute Force Algorithm

- This is the most basic and simplest type of algorithm.
- A Brute Force Algorithm is the straightforward approach to a problem i.e., the first approach that comes to our mind on seeing the problem.
- More technically it is just like iterating every possibility available to solve that problem.

Recursive Algorithm

- A problem is resolved by breaking it down into subproblems of a similar nature and repeating the process over and over until it is resolved with the help of a base condition.

Dynamic Programming Algorithm

- Using Dynamic Programming, you can break up the unpredictable issue into smaller, more manageable subproblems and put the outcome aside for later. We can say that it remembers previous results and uses them to find new ones. It is the third main type of algorithm.
- This algorithm has two version:
    - **Bottom-Up Approach:** Starts solving from the bottom of the problems i.e. solving the last possible subproblems first and using the result of those solving the above subproblems.
    - **Top-Down Approach:** Starts solving the problems from the very beginning to arrive at the required subproblem and solve it using previously solved subproblems.

Divide And Conquer Algorithm

- This algorithm divides the problems into subproblems and then solve each of them and then combines them to form the solution of the given problems.
- Again, it is not possible to solve all problems using it. As the name suggests it has two parts: Divide the problem into subproblems and solve them.

Greedy Algorithm

- In this algorithm, a decision is made that is good at that point without considering the future. This means that some local best is chosen and considers it as the global optimal.

Dynamic Programming approach

- This type of algorithm is also known as the memoization technique because in this the idea is to store the previously calculated result to avoid calculating it again and again.
- In Dynamic Programming, divide the complex problem into smaller overlapping subproblems and store the result for future use.

Backtracking Algorithm

- It is an improvement to the brute force approach.
- Here we start with one possible option out of many available and try to solve the problem if we are able to solve the problem with the selected move then we will print the solution else we will backtrack and select some other and try to solve it.
- It is a form of recursion, it’s just that when a given option cannot give a solution, we backtrack to the previous option which can give a solution, and proceed with other options.

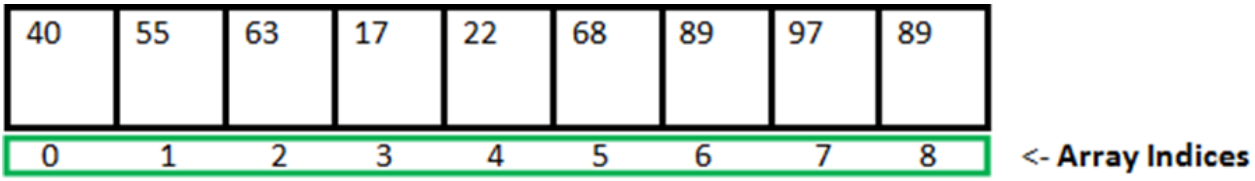
Randomized Algorithm

- In the randomized algorithm, we use a random number.it helps to decide the expected outcome.
- The decision to choose the random number so it gives the immediate benefit
- Some common problems that can be solved through the Randomized Algorithm are Quicksort: In Quicksort we use the random number for selecting the pivot.

Unit : 1 : Topic : 2: Advanced Concepts of C

Memory Allocation :

Since C is a structured language, it has some fixed rules for programming. One of them includes changing the size of an array. An array is a collection of items stored at contiguous memory locations.



Array Length = 9
First Index = 0
Last Index = 8

- As it can be seen that the length (size) of the array is 9. But what if there is a requirement to change this length (size).
- For Example, if there is a situation where only 5 elements are needed to be entered in this array. In this case the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12. This procedure is referred to as Dynamic Memory Allocation.

Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

malloc() calloc() free() realloc()

1. **Malloc() :**

“Malloc” or “memory allocation” method is used to dynamically allocate a single large block of memory with the specified size.
It returns a pointer of type void which can be cast into a pointer of any form.
Syntax: *ptr = (cast-type*) malloc(byte-size)
Example : *ptr = (int*) malloc(100 * sizeof(int));
Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

If the space is insufficient, allocation fails and returns a **NULL pointer.**

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
void main()
{
	int* ptr;
	int n, i;
	clrscr();
	printf("Enter number of elements:");
	scanf("%d",&n);

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

	// Check if the memory has been successfully allocated by malloc or not
	if (ptr == NULL)
	{
		printf("Memory not allocated.\n");
		exit(0);
	}
	else
	{
		printf("Memory successfully allocated using malloc.\n");
		for (i = 0; i < n; ++i) {
			ptr[i] = i + 1;
		}
		printf("The elements of the array are: ");
		for (i = 0; i < n; ++i) {
			printf("%d, ", ptr[i]);
		}
	}
	getch();
}
```

2. **Calloc() :**

- “calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type.
- The calloc() function offers a couple of advantages over malloc().
- It allocates memory as a number of elements of a given size.
- It initializes the memory that is allocated so that all bytes are zero.
- calloc() function requires two argument values:
  - The number of data items for which space is required.
  - Size of each data item.
- It is very similar to using malloc() but the big plus is that you know the memory area will be initialized to zero.

Syntax: ptr = (cast-type*)calloc(n, element-size);

For Example: ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of float.

Example :

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
void main()
{
        int* ptr;
        int n, i;
        clrscr();
        printf("Enter number of elements: ");
        scanf("%d",&n);

        ptr = (int*)calloc(n, sizeof(int));

        // Check if the memory has been successfully  allocated by calloc or not
        if (ptr == NULL) {
                printf("Memory not allocated.\n");
                exit(0);
        }
        else
        {
                printf("Memory successfully allocated using calloc.\n");
                for (i = 0; i < n; ++i) {
                        ptr[i] = i + 1;
                }

                printf("The elements of the array are: ");
```

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

3. realloc():

- The realloc() function enables you to reuse or extend the memory that you previously allocated using malloc() or calloc().
- A pointer containing an address that was previously returned by a call to malloc(), calloc().
- The size in bytes of the new memory that needs to be allocated.
- The realloc function allocates a block of memory (which be can make it larger or smaller in size than the original) and copies the contents of the old block to the new block of memory, if necessary.
- realloc deallocates the old object pointed to by ptr and returns a pointer to a new object that has the size specified by size.
- The point to note is that realloc() should only be used for dynamically allocated memory. If the memory is not dynamically allocated, then behavior is undefined.

Syntax:

```
void *realloc(void *ptr, size_t size)
```

Example :

```
#include <stdio.h>
#include <stdlib.h>
#include<conio.h>
void main ()
{
   char *str;
   clrscr();
   str = (char *) malloc(15);
   strcpy(str, "Geetanjali");
   printf("String = %s,  Address = %u\n", str, str);

   /* Reallocating memory */
   str = (char *) realloc(str, 25);
   strcat(str, "College ");
   printf("String = %s,  Address = %u\n", str, str);
   free(str);
   getch();
}
```

Output :

String - Geetanjali, Address - 1512
String - GeetanjaliCollege, Address - 1532

4. free():

- When memory is allocated dynamically it should always be released when it is no longer required.
- Memory allocated dynamically will be automatically released when the program ends but is always better to explicitly release the memory when done with it, even if it's just before exiting from the program.

Syntax: free(pointer);

What is the difference between malloc calloc and realloc?:

malloc	calloc	realloc
malloc stands for memory allocations.	calloc stands for contiguous allocation	realloc stands for re-allocation.
malloc() is a function which is used to allocate a block of memory dynamically.	Calloc() is a function which is used to allocate multiple blocks of memory.	realloc() is a function which is used to resize the memory block which was allocated by malloc or calloc before.
It is used to dynamically allocate a single large block of memory with the required size	calloc() specifies the number of blocks of memory to be allocated.	When memory allocated previously is not sufficient, then more memory is required, the realloc method is used to re-allocate memory dynamically.
Syntax: mp = (cast_type*) malloc(byte_size);	Syntax: cp= (int*) calloc(100, sizeof(int));	Syntax: rp = realloc(rp, newSize);
malloc() just declare block of memory. (does not initialize it)	calloc() assign a 0 to each memory block.	realloc() reassign a new memory block which was previously assigned with malloc() or calloc().

Below is the C program to illustrate the Dynamic Memory Allocation:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
void main()
{
        int size, resize;
        char* str = NULL;
```

```
        printf("Enter limit of the text: \n");
        scanf("%d", &size);

        str = (char*)malloc(size * sizeof(char));
        if (str != NULL)
        {
                printf("Enter some text: \n");
                //scanf(" ");
                //gets(str);
                printf("Inputted text by allocating memory using malloc() is: %s\n", str);
        }
        free(str);

        str = (char*)calloc(50, sizeof(char));
        if (str != NULL)
        {
        printf("Enter some text: \n");
        scanf(" ");
        gets(str);
        printf("Inputted text by allocating memory using calloc() is: %s\n",     str);
        }

        printf("Enter the new size: \n");
        scanf("%d", &resize);
        str = (char*)realloc(str, resize * sizeof(char));
        printf("Memory is successfully reallocated by using realloc() \n");
        if (str != NULL) {
        printf("Enter some text: \n");
        scanf(" ");
        gets(str);
        printf("Inputted text by reallocating  memory using realloc()is: %s\n",str);
        }
        free(str);
        getch();
}
```

Difference between Static Memory Allocation v/s Dynamic Memory Allocation:

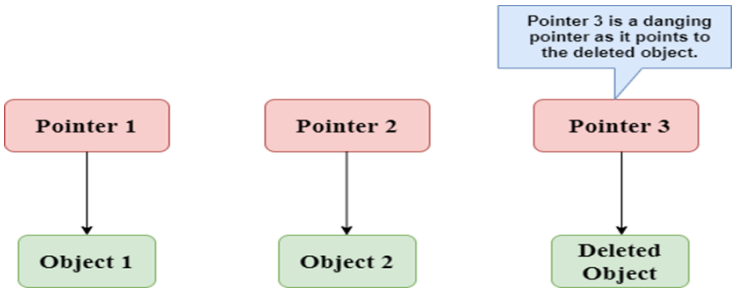
Static Memory Allocation	Dynamic Memory Allocation
In static memory allocation, size is fixed which can not be changed during runtime.	In Dynamic memory allocation, size can be changed during runtime.
In static memory allocation, memory is allocated at compile time.	In dynamic memory allocation, memory is allocated at run time.

In static memory allocation, memory wastage takes place.	In dynamic memory allocation, memory is saved.
In static memory allocation, there is no need to free the memory.	In dynamic memory allocation, memory must be freed using free().

Dangling Pointer Problem

- Dangling pointers arise when an object is deleted or de-allocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the de-allocated memory.
- In this case, the pointer is pointing to the memory, which is de-allocated.
- In short, pointers pointing to non-existing memory locations are called dangling pointers.
- If any pointer is pointing to the memory address of any variable but after some variable has deleted from that memory location while the pointer is still pointing to such memory location. Such a pointer is known as a dangling pointer and this problem is known as the dangling pointer problem.

Let's observe the following examples :



- In the above figure, we can observe that the Pointer 3 is a dangling pointer. Pointer
- 1 and Pointer 2 are the pointers that point to the allocated objects, i.e., Object 1 and Object 2, respectively. Pointer 3 is a dangling pointer as it points to the de-allocated object.
- Let's understand the dangling pointer through some C programs using free() function to de- allocate the memory.

Example :

```
#include <stdio.h>
int main()
{
  int *ptr=(int *)malloc(sizeof(int));
  int a=560;
  ptr=&a;
  free(ptr);
  return 0;
```

```
}
```

The dangling pointer errors can be avoided by initializing the pointer to the **NULL** value. If we assign the **NULL** value to the pointer, then the pointer will not point to the de-allocated memory. Assigning **NULL** value to the pointer means that the pointer is not pointing to any memory location.

Enumerated Constants :

- Enumerated constants enable you to create new types and then to define variables of those types whose values are restricted to a set of possible values.
- For example, you can declare COLOR to be an enumeration, and you can define five values for COLOR: RED, BLUE, GREEN, WHITE, and BLACK.
- The syntax for enumerated constants is to write the keyword enum, followed by the type name, an open brace, each of the legal values separated by a comma, and finally, a closing brace and a semicolon. Here's an example:

```
enum COLOR {RED,BLUE , GREEN,WHITE,BLACK};
```

This statement performs two tasks:

- It makes COLOR the name of an enumeration; that is, a new type.
- It makes RED a symbolic constant with the value 0, BLUE a symbolic constant with the value 1, GREEN a symbolic constant with the value 2, and so forth.

By default, *value1* will be equal to 0, *value2* will be 1 and so on but, the programmer can change the default value as below:

```
enum suit
{
        club=0;
        diamonds=10;
        hearts=20;
        spades=3;

};
```

Example :

```
#include<stdio.h>
enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
int main()
{
        enum week day;
        day = Wed;
        printf("%d",day);
        return 0;
```

```
}
```

// Another example program to demonstrate working of enum in C

```
#include<stdio.h>
enum year{Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
int main()
{
        int i;
        for (i=Jan; i<=Dec; i++)
        printf("%d ", i);
        return 0;
}
```

Unit : 2 Sorting and Searching

Sorting :

- Sorting is the process of arranging data information in some logical order.
- Sorting algorithm specifies the way to arrange data in a particular order.
- Most common orders are in numerical or lexicographical(શબ્દકોશ) order.
- This logical order may be ascending or descending in case of numeric values.
- Various techniques are available to sort data depending on length of data, speed of sorting, number of swapping done during procedure of sorting etc.
- Sorting is also used to represent data in more readable formats.
- Following are some of the examples of sorting in real-life scenarios

Telephone Directory − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

Dictionary − The dictionary stores words in an alphabetical order so that searching for any word becomes easy.

following are most common sorting techniques :

1. Bubble sorting
2. Insertion sorting
3. Quick sorting
4. Bucket sorting
5. Merge sorting
6. Selection sorting
7. Shell sorting

Bubble Sort : (Bubble Sort)

- Bubble sort is a simple sorting algorithm
- In this technique, we continually compare two adjacent items (elements) from the list. If the first element is larger than the second one, then the position of the elements are interchanged (swap) otherwise not changed and then the after sorting is completed. This process is used frequently until no swaps are needed.
- It is also known as **"comparison sort"** because it continually compares two adjacent elements from the list.
- This algorithm is not suitable for large data sets.

Advantages	Disadvantages
It's a simple algorithm that can be implemented on a computer.	It's not an efficient way to sort a list.
Efficient way to check if a list is already in order.	The bubble sort algorithm is pretty slow for very large lists of items.

Don't use too much memory.	Takes more time to sort data.

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order.

5	1	4	2	8

For example :
First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), the algorithm does not swap them.

Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Algorithm for Bubble sort :
Let “a” be an array of n numbers. “temp” is a temporary variable for swapping the position of the numbers.
Step 1: Input n numbers for an array “a”
Step 2: Initialize i=0 and repeat through step 4 if(i<n)
Step 3 : initialize j = 0 and repeat through step 4 if (j<n-1)
Step 4 : if (a[j] >a[j+1])

```
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
```

Step 5: Display the sorted numbers of array a

Step 6: Exit

Example :

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int a[100],temp,i,j,n,changes,k;
	clrscr();
	printf("\nEnter array size:");
	scanf("%d",&n);
	for(i=0;i<n;i++)
	{
		printf("\nEnter array elements a[%d]:",i);
		scanf("%d",&a[i]);
	}
	for(i=0;i<n;i++)
	{
		//changes=0;
		for(j=0;j<n-1;j++)
		{
			if(a[j]>a[j+1])
			{
				temp=a[j];
				a[j]=a[j+1];
				a[j+1]=temp;
				changes++;
			}
		}
		if(changes==0)
		break;
		printf("\nAfter pass %d elements are:",i+1);
		for(k=0;k<n;k++)
		{
			printf("\t%d",a[k]); printf("\n");
		}
	}
	printf("\nAfter sorting array elements are:");
	for(i=0;i<n;i++)
	{
			printf("\n%d",a[i]);
```

```
	}
	getch();
}
```

Example : Bubble sort for 2d Array :

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int i, j, k,x,y, temp, A[2][2],ch;
	clrscr();
	printf("Enter array elements in A:\n");
	for (i = 0; i <2; i++)
	{
		for (j = 0; j < 2; j++)
		{
			scanf("%d", &A[i][j]);
		}
	}
	printf("\nUnsorted array is : \n");
	for (i = 0; i <2; i++)
	{
		for (j = 0; j < 2; j++)
		{
			printf("%d\t", A[i][j]);
		}
	}
	for (i=0; i<4; i++)
	{
		ch=0;
		for (k=0; k<2; k++)
		{
			for (j=0; j<2; j++)
			{
			do  {
				temp = A[k][j];
				A[k][j] = A[k][j+1];
				A[k][j+1] = temp;
				//ch++;
			} while (A[k][j]>A[k][j+1]);
			}
```

```
                }
                /*if(ch ==0)
                        break;
                printf("\nAfter PAss %d value is :\n",i);
                for(x=0; x<2; x++)
                {
                        for(y=0; y<2; y++)
                        {
                                printf("%d  ", A[x][y]);
                        }
                }   */
        }
        printf("\nSorted array elements are:\n");
        for (i = 0; i < 2; i++)
        {
                for (j = 0; j < 2; j++)
                {
                        printf("%d\t", A[i][j]);
                }
        }
        getch();
}
```

Insertion Sort : (Insertion Sort)

- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- It is a very simple and efficient algorithm for the smallest lists.
- The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are pic
- ked and placed at the correct position in the sorted part.
- Its mechanism is very simple: just take elements from the list one by one and insert them in their correct position into a new sorted list.
- The name inserting sorting means that sorting occurs by inserting a particular element at proper position.

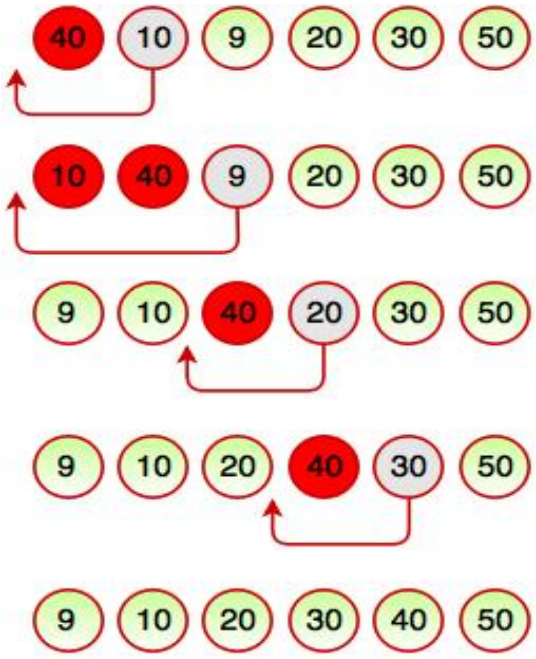


Fig. Working of Insertion Sort

Another Example:

12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13

11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13

Algorithm for Insertion sort :

Let “a” be an array of n numbers. “temp” is a temporary variable for swapping the position of the numbers.

Step 1: Input n numbers for an array “a”

Step 2: Initialize i=0 and repeat through step 4 if(i<n-1)

temp=a[i]

j=i-1

Step 3: Repeat the step 3 if(temp<a[j] and (j>=0))

a[j+1]=a[j]

j=j-1

Step 4: a[j]=temp

Step 5: Exit

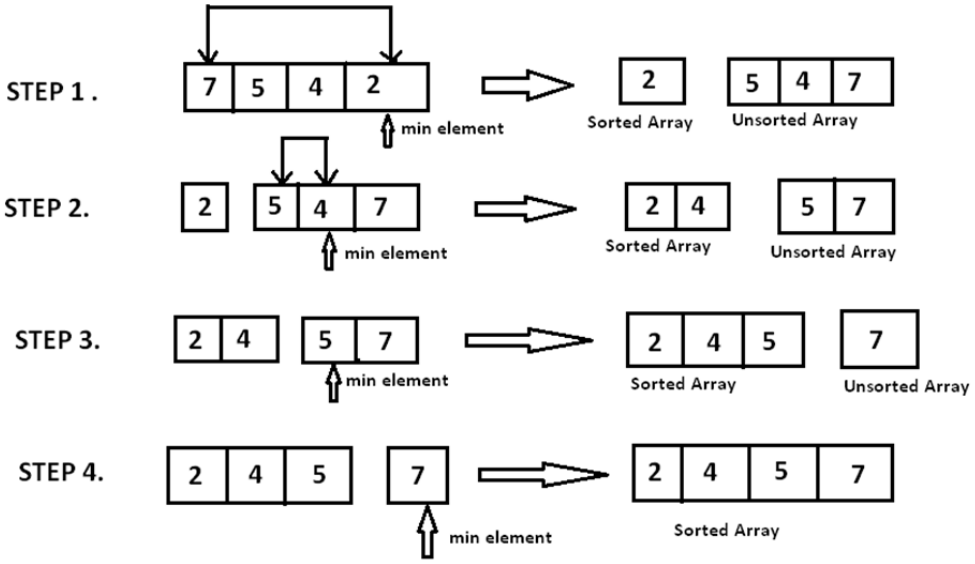
Example :

```
#include<stdio.h>
#include<conio.h>
 void main()
{
        int a[100],n,i,j,k,tmp,changes;
        clrscr();
        printf("\nEnter array size:");
```

```
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter array element a[%d]:",i);
                scanf("%d",&a[i]);
        }
        printf("\n\n Unsorted array");
        for(i=0;i<n;i++)
        {
                printf("\t%d",a[i]);
        }
        for(i=1;i<n;i++)
        {
                tmp=a[i];
                changes=0;
                for(j=i-1;j>=0;j--)
                {
                        if(tmp<a[j])
                        {
                                a[j+1]=a[j];
                                a[j]=tmp;
                        }
                        changes++;
                  }
                if(changes==0)
                break;
                printf("\nPass %d,element inserted at proper place:%d",i,tmp);
                for(k=0;k<n;k++)
                {
                          printf("\t%d",a[k]);
                          printf("\n");
                }
        }
        printf("\n\nSorted array");
         for(i=0;i<n;i++)
        {
                          printf("\t%d",a[i]);
        }
        getch();
}
```

Selection Sort :(Selection sort)

- Selection Sort algorithm is a simple sorting algorithm which specially is an in-place comparison sort. It is a technique to arrange the data in proper order.
- This type of sorting is called “Selection sort” because it works by repeatedly selecting the smallest element.
- This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
- If we want to sort an array in increasing order(i.e smallest element at the beginning of the array and the largest element at the end.) then find the minimum element and place it in the first position (recursion).
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving the unsorted array boundary by one element to the right.
- This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.



How Selection Sort Works?

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.
Following is a pictorial depiction of the entire sorting process −

Algorithm for Selection sort

Step 1: Input n number of elements in array a.
Step 2: Initialize i=0
Step 3: Repeat through step 8 while i<n-1(i=0,1,2….,n-1)
Step 4: Initialize min=i
Step 5: Initialize j=i+1
Step 6: Repeat through j=j+1 while j<n(j=i+1,i+2,….)
    If(a[j]<a[min])
    min=j
Step 7: if(min!=i)
    temp =a[i]
    a[i]=a[min]
    a[min]=temp
Step 8: i=i+1
Step 9: Exit

Program for Selection sort :

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a[100],n,i,j,temp,min;
        clrscr();
        printf("\nEnter array size:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter array elements a[%d]:",i);
                scanf("%d",&a[i]);
        }
        for(i=0;i<n-1;i++)
        {
                min=i;
                for(j=i+1;j<n;j++)
                {
                        if(a[j]<a[min])
                        min=j;
                }
                if(min!=i)
                {
                        temp=a[i];
```

```
                    a[i]=a[min];
                    a[min]=temp;
                }
        }

        printf("\n\nSorted array:");
         for(i=0;i<n;i++)
        {
                printf("\t%d",a[i]);
        }
        getch();
}
```

Quick Sort : (Quick sort)

- Quick sort is a highly efficient sorting algorithm and is based on partitioning an array of data into smaller arrays.
- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
- It is a widely used sorting technique which uses divide and conquer (also known as partition exchange sort)mechanism. The quicksort algorithm works by partitioning the array to be sorted. And each partition is internally sorted recursively.
- It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.
    - Always pick the first element as pivot.
    - Always pick last element as pivot (implemented below)
    - Pick a random element as pivot.
    - Pick median as pivot.
- There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.
- Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

Algorithm for Quick sort :

Based on our understanding of partitioning in quicksort, we will now try to write an algorithm for it, which is as follows.

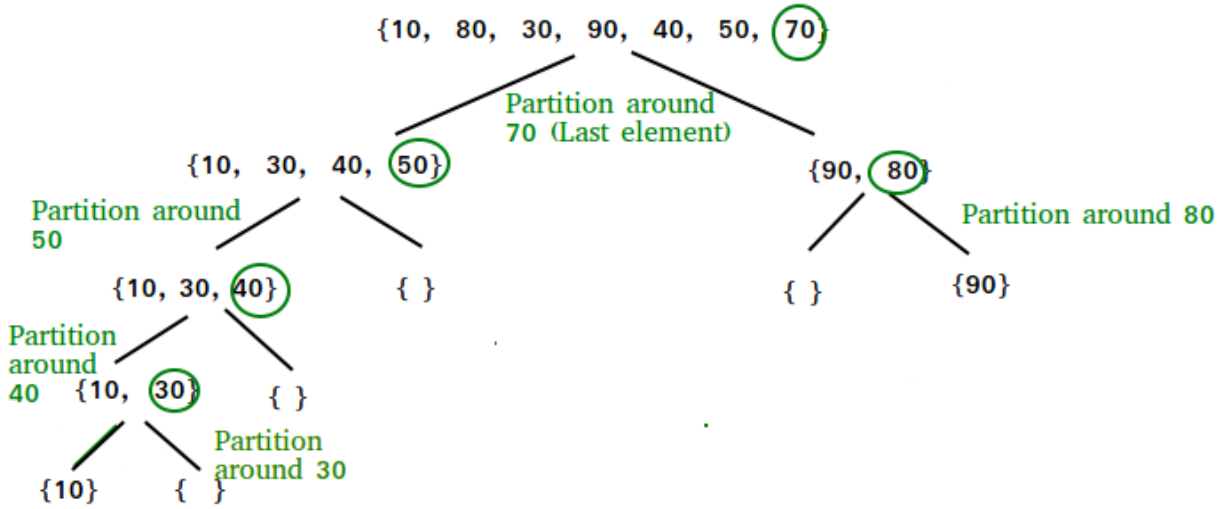
Step 1 − Choose the highest index value has pivot

Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot



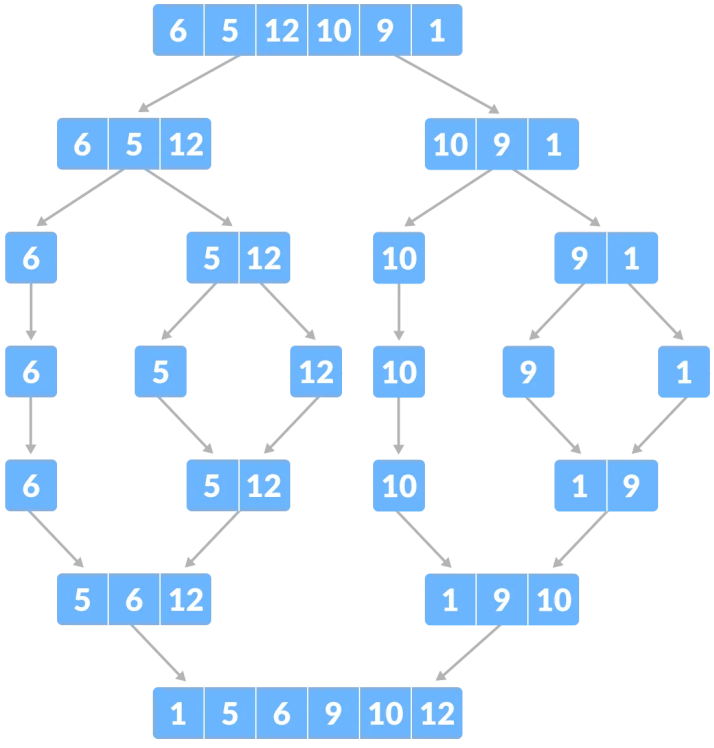
Example :

```
#include<stdio.h>
#include<conio.h>
void quicksort(int [],int,int);
void main()
{
        int a[100],n,i;
        clrscr();
        printf("\nenter array size:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\n enter array elements a[%d]",i);
                scanf("%d",&a[i]);
        }
        quicksort(a,0,n-1);
        for(i=0;i<n;i++)
        {
                printf("\n sorted array elements %d",a[i]);
        }
        getch();
}
void quicksort(int a[], int first, int last)      //function definition
{
```

```
        int low, high, temp, pivot, i;
        low=first;
        high=last;
        pivot=(low+high)/2;
        while(low<=high)
        {
                while(a[low]<a[pivot])
                low++;
                while(a[high]>a[pivot])
                high--;
                if(low<=high)
                {
                        temp=a[low];
                        a[low]=a[high];
                        a[high]=temp;
                        low++;
                        high--;
                }
          }
        if(first<high)
        quicksort(a,first,high);
        if(low<last)
        quicksort(a,low,last);
    }
```

Merge Sort : (Merge Sort)

- Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.
- Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each sub problem is ready, we 'combine' the results from the sub problems to solve the main problem.
- It is the process of combining two or more sorted arrays into a third sorted array.
- Consider an array A of n number of elements. The algorithm processes the elements in 3 steps.



- If A Contains 0 or 1 elements then it is already sorted, otherwise, Divide A into two sub-array of equal number of elements.
- Conquer means sort the two sub-arrays recursively using the merge sort.
- Combine the sub-arrays to form a single final sorted array maintaining the ordering of the array.

- The main idea behind merge sort is that the short list takes less time to be sorted.

How Merge Sort Works?

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

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

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

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

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the elements for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

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

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

Algorithm for Merge sort :

Mergesort(n,list1,m,list2);

n:- represent number of elements in first list
list1:- represent list of elements (first list)
m:- represent number of elements in second list
list2: represent list of elements (second list)

Step 1: First, the array is divided into two parts. i.e. mid is determined between low index and high index.

```
midsort(low, high);
low: low index
high: high index
mid=(low+high)/2
```

Step 2: then first part(from low to mid) and second part (mid+1 to high) are sorted by calling the function midsort

```
midsort(low,mid);
midsort(mid+1,high)
```

Step 3: Then, above two sorted parts are merged by calling mergesort function

Step 4: initialize i =low, j =mid+1, k=high

Step 5: repeat this step till i<=mid and j<=high

```
if(a[i]>=a[j])
        temp[k]=a[j];
        k++;
        j++;
else
        temp[k]=a[i];
        k++;
        i++;
```

Step 6: repeat this step till i<=mid.

```
        temp[k]=a[i];
        k++;
        i++;
```

Step 7: repeat this step till j<=high

temp[k]=a[j];

k++;

j++;

Step 8: repeat this step till i<=high

for(i=low;i<=high;i++)

copying final array to a

a[i]=temp[i]

Step 9: STOP

Program for Merge sort :

```
#include<stdio.h>
#include<conio.h>
int a[100];
void m_sort(int,int);
void merge_sort(int,int,int);
void main()
{
        int n,i;
        clrscr();
        printf("\nEnter array size:");
         scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter array elements a[%d]:",i);
                 scanf("%d",&a[i]);
        }

        printf("\n\nUnsorted array");
        for(i=0;i<n;i++)
        {
                printf("\t%d",a[i]);
        }

        m_sort(0,n-1);
        printf("\n\nSorted array");
        for(i=0;i<n;i++)
        {
                printf("\t%d",a[i]);
        }
        getch();
```

```
}
void m_sort(int low,int high)
{
        int mid;
        if(low!=high)
        {
                mid=(low+high)/2;
                m_sort(low,mid);
                m_sort(mid+1,high);
                merge_sort(low,mid,high);
        }
}
void merge_sort(int low,int mid,int high)
{
        int i,j,k,temp[100];
        i=low;
        j=mid+1;
         k=low;
        do
        {
                if(a[i]>=a[j])
                        temp[k++]=a[j++];
                else
                        temp[k++]=a[i++];
        }while((i<=mid)&&(j<=high));

         while(i<=mid)
        temp[k++]=a[i++];

         while(j<=high)
        temp[k++]=a[j++];

         for(i=low;i<=high;i++)
        {
                a[i]=temp[i];
        }
}
```

Bucket sort : (Bucket Sort)

- Bucket sort or **bin sort** is a sorting method that can be used to sort a list of numbers by its base.

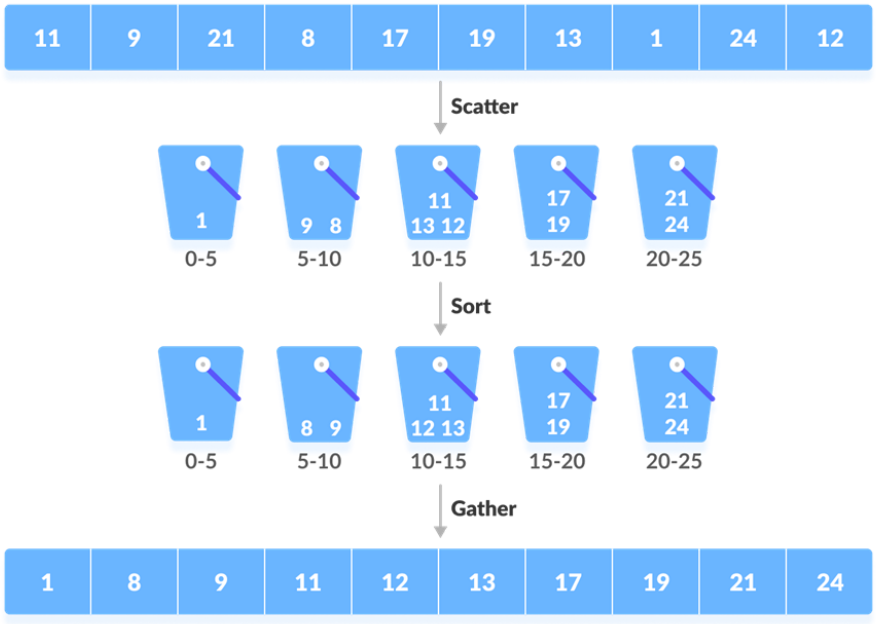
- Bucket Sort is a sorting technique that sorts the elements by first dividing the elements into several groups called buckets.
- The elements inside each bucket are sorted using any of the suitable sorting algorithms or recursively calling the same algorithm.
- A number of buckets are created.
- Each bucket is filled with a specific range of elements.
- The elements inside the bucket are sorted using any other algorithm.
- Finally, the elements of the bucket are gathered to get the sorted array.
- The process of bucket sort can be understood as a scatter-gather approach.
- The elements are first scattered into buckets then the elements of buckets are sorted. Finally, the elements are gathered in order.

The advantages of bucket sort are -

- Bucket sort reduces the no. of comparisons.
- It is asymptotically fast because of the uniform distribution of elements.

The limitations of bucket sort are -

- It may or may not be a stable sorting algorithm.
- It is not useful if we have a large array because it increases the cost.
- It is not an in-place sorting algorithm, because some extra space is required to sort the buckets.



Algorithm for Bucket sort :

Step 1: initialize array and its elements
Step 2: Find out the largest element and the digit of the largest element
Step 3: Initialize the buckets j=0 and repeat the steps (a) until (bucket[i] > 0)

Step 4: Compare the position of each element of the array with the bucket number and place it into the corresponding bucket.
Step 5: Read the elements of the bucket from 0^{th} bucket to 9^{th} bucket and from first position to higher one to generate a new array a.
Step 6: Display the sorted array a
Step 7: Exit

Program for Bucket sort

```
#include <stdio.h>
#include<conio.h>
int getMax(int array[], int size)
{
        int max = array[0];
        int i;
        for (i = 1; i < size; i++)
        if (array[i] > max)
        max = array[i];
        return max;
}
void bucketSort(int array[], int size)
{
        /* The size of the bucket must be at least the (max+1) but we cannot assign it as int bucket(max+1) in C as it does not support dynamic memory allocation. So, its size is provided statically. */

        int bucket[10];
        int i,j;
        const int max = getMax(array, size);
        for (i = 0; i <= max; i++)
        {
                bucket[i] = 0;
        }
        for (i = 0; i < size; i++)
        {
                bucket[array[i]]++;
        }
        for (i = 0, j = 0; i <= max; i++)
        {
                while (bucket[i] > 0)
                {
                        array[j++] = i;
```

```
                        bucket[i]--;
                }
        }
}
void main()
{
  int data[] = {4, 3, 7, 8, 6, 0, 9, 5};
  int i;
  int size = sizeof(data) / sizeof(data[0]);
  clrscr();
  bucketSort(data, size);
  printf("Sorted array in ascending order: \n");
  for (i = 0; i < size; ++i)
  {
                printf("\t%d",data[i]);
  }
  getch();
}
```

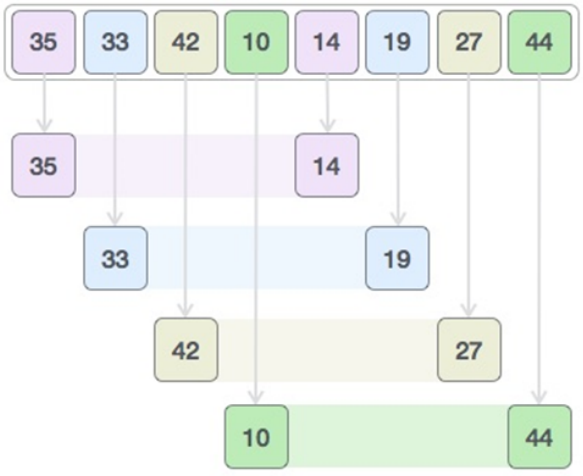
Shell Sort : (Shell Sort)

- Shell sort is also called **diminishing increment sort.**
- It can either be seen as a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort).
- Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm.
- This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.
- This algorithm uses insertion sort on widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval.

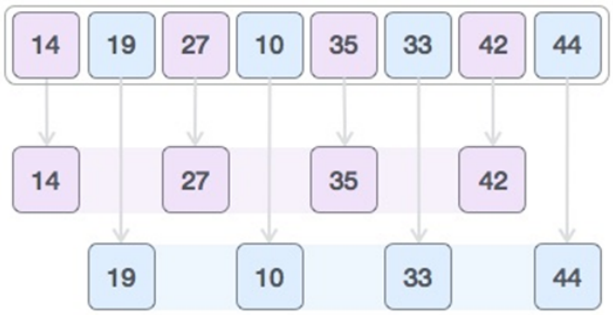
How Shell Sort Works?

Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}

We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −



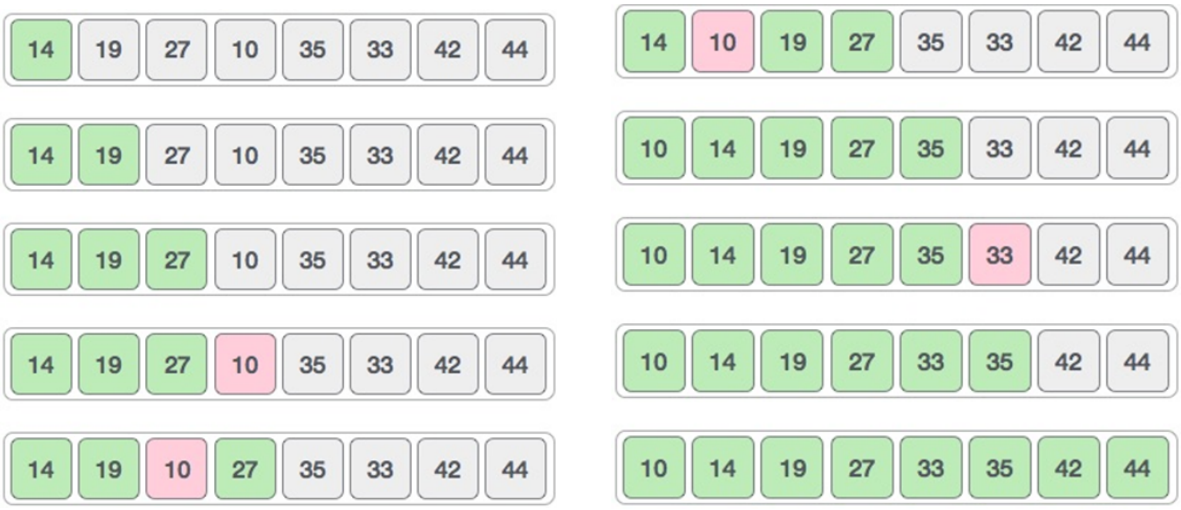
Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this −

Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −



Algorithm for Shell sort

Step 1: Input n number of elements in array a.
Step 2: Initialize i= n/2 and repeat through step 5 if(i>0)
Step 3: Initialize j=i and repeat through step 5 if (j<=n)
Step 4: Initialize k=j-i and repeat through step 5 if(k>=0)
Step 5: Check whether a[k+i] is greater than a[k], break the loop.

```
else
tmp = arr[k];
arr[k] = arr[k+i];
arr[k+i] = tmp;
```

Step 6: Exit

Program for Shell sort :

```
#include <stdio.h>
#include <conio.h>
void shellsort(int arr[], int num)
{
```

```
    int i, j, k, tmp;
    for (i = num / 2; i > 0; i = i / 2)
    {
        for (j = i; j < num; j++)
        {
          for(k = j - i; k >= 0; k = k - i)
          {
                    if (arr[k+i] >= arr[k])
                              break;
                    else
                    {
                              tmp = arr[k];
                              arr[k] = arr[k+i];
                              arr[k+i] = tmp;
                    }
          }
        }
    }
}
void main()
{
    int arr[30];
    int k,  num;
    clrscr();
    printf("Enter total no. of elements : ");
    scanf("%d", &num);
    for (k =  0 ; k < num; k++)
    {
          printf("\nEnter element [%d]: ", k);
          scanf("%d", &arr[k]);
    }
    shellsort(arr, num);
    printf("\n Sorted array is: ");
    for (k = 0; k < num; k++)
        printf("%d ", arr[k]);
    getch();
 }
```

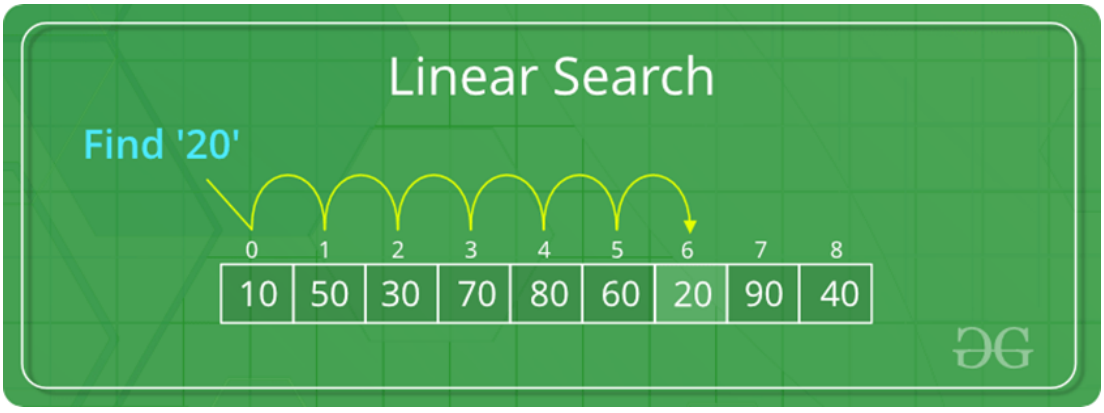
Searching :

- Searching in C Language has to check for an element or retrieve an element from any data structure where the data is stored.
- Searching in data-structure refers to the process of finding a desired element in a set of items. The desired element is called "target". The set of items to be searched in, can be any data-structure like − list, array, linked-list, tree or graph.
- Search refers to locating a desired element of specified properties in a collection of items. We are going to start our discussion using the following commonly used and simple search algorithms.
- Based on the type of search operation, these algorithms are generally classified into two categories:
    - **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.
    - **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

Linear Search : (Linear Search)

- Linear search is a very simple search algorithm.
- In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.
- To search (locate, find) an element from the unsorted array list we are using this simplest technique.
- A simple approach is to do linear search : Start from the leftmost element of arr[] and one by one compare x with each element of arr[].
- If x matches with an element, return the index.
- If x doesn't match with any of the elements, return -1.
- It is also known as a sequential searching method.

Linear Search to find the element "20" in a given list of numbers



Algorithm for sequential searching

Here, a represents array
n represent number of elements in array
ele - represents element to be searched

Step 1: Initialize flag=0

Step 2: Initialize i=0 and repeat till if(i<n)
Get array elements (&a[i])

Step 3: Repeat step 4 for i=0,1,2….n-1

Step 4:if(a[i]==ele)
Output “successful searching”
flag=1

Step 5: if flag==0
Output “unsuccessful search”

Step 6: STOP

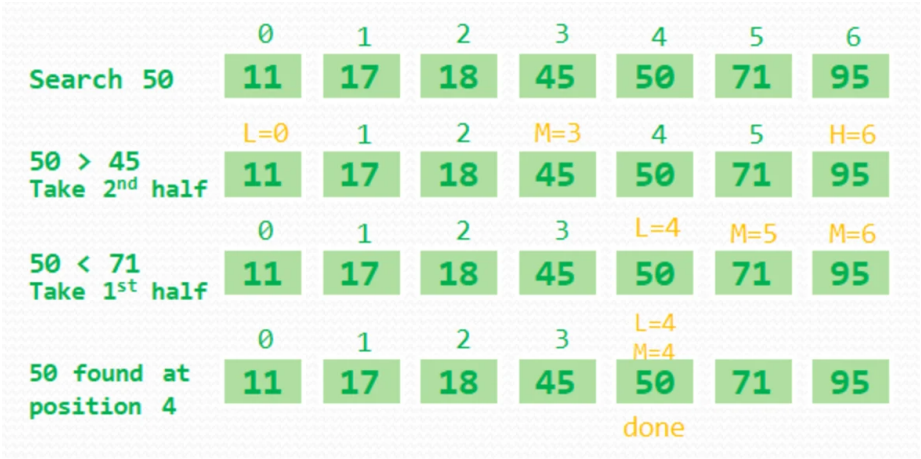
Example :

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int list[20],size,i,ele,flag=0;
        clrscr();
        printf("Enter size of the list: ");
         scanf("%d",&size);
        printf("Enter Values for array");
         for(i = 0; i < size; i++)
        scanf("%d",&list[i]);

        printf("Enter the element to be Search: ");
        scanf("%d",&ele);

        // Linear Search Logic
        for(i = 0; i < size; i++)
        {
                if(ele == list[i])
                        flag++;
        }
        if(flag!=0)
                printf("Given element is found %d times",flag);
        else
```

```
            printf("element is not found in the list");
        getch();
}
```

Binary Search : (Binary Search)

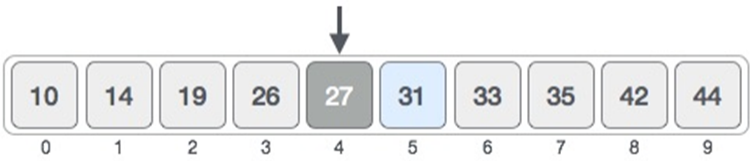
- A **binary search**, also known as a **half-interval search**.
- This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.
- It is an algorithm used in computer science to locate a specified value (key) within an array.
- Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found.
- If the search ends with the remaining half being empty, the target is not in the array.
- Binary search is a fast search algorithm.
- Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of the item is returned.
- If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item.
- This process continues on the sub-array as well until the size of the subarray reduces to zero.



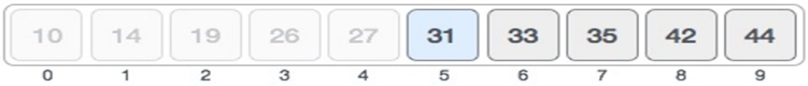
How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −
mid = low + (high - low) / 2
Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



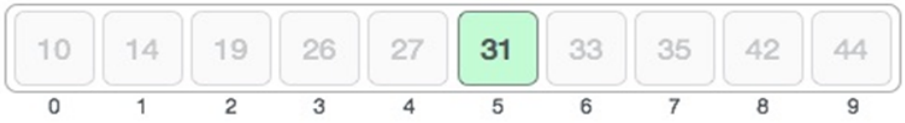
Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, we also know that the target value must be in the upper portion of the array.



We change our low to mid + 1 and find the new mid value again.
low = mid + 1
mid = low + (high - low) / 2
Our new mid is 7. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.
We compare the value stored at location 5 with our target value. We find that it is a match. We conclude that the target value 31 is stored at location 5.
Binary search halves the searchable items and thus reduces the count of comparisons to be made to very few numbers.



Algorithm :

arr - array size[50]
n = number of elements in array
search - element to be search

Step - 1 : Initialize i=0 and repeat till **n**
Get array elements (&a[i])

Step 2: Get index value to be searched on from user(&ind)

Step-3 : first = 0;
last = n-1;
middle = (first+last)/2;

Step - 4 : while (first <= last)
{
if(arr[middle] < search)
{
first = middle + 1;
}
else if(arr[middle] == search)
{
printf("%d found at location %d\n", search, middle+1);
break;
}
else
{
last = middle - 1;
}
middle = (first + last)/2;
}

Step - 5 : if(first > last)
printf(“element not found.”);

Step – 6 : Stop

Example :

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int n, i, arr[50], search, first, last, middle;
        clrscr();
        printf("Enter total number of elements :");
        scanf("%d",&n);
        printf("Enter %d number :", n);
```

```
for (i=0; i<n; i++)
{
        scanf("%d",&arr[i]);
}
printf("Enter a number to find :");
scanf("%d", &search);
first = 0;
last = n-1;
middle = (first+last)/2;
while (first <= last)
{
        if(arr[middle] < search)
        {
                first = middle + 1;
        }
        else if(arr[middle] == search)
        {
                printf("Element %d is found at location %d\n", search,
                middle+1);
                  break;
        }
        else
        {
                 last = middle - 1;
        }
        middle = (first + last)/2;
}
if(first > last)
{
        printf("Not found! %d is not present in the list.",search);
}
getch();
}
```

Difference between Linear Search and Binary Search :

Linear Search	Binary Search
Sorted list is not required.	Sorted list is required.
It can be used in linked list implementation.	It cannot be used in linked list implementation.

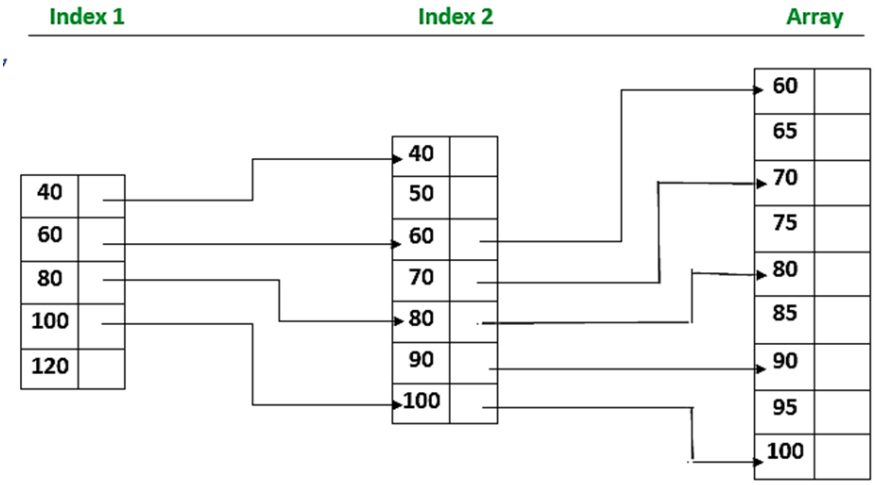
It is suitable for a list changing very frequently.	It is only suitable for static lists, because any change requires resorting to the list.
The average number of comparisons is very high.	The average number of comparisons is relatively slow.
Time taken to search elements keeps increasing as the number of elements are increased.	A binary search however, cut down your search to half as soon as you find the middle of a sorted list.
Linear search does the sequential access	whereas Binary search access data randomly.
Linear search performs equality comparisons	Binary search performs ordering comparisons

Index Searching :

- Index search is special search. This search method is used to search a record in a file. Searching a record refers to the searching of location loc in memory where the file is stored. Indexed search searches the record with a given key value relative to a primary key field. This search method is accomplished by the use of pointers.
- Index helps to locate a particular record with less time. Indexed sequential files use the principle of index creation.
- For Example, Directory, book etc. In this type of file organization, the records are organized in a sequence and an index table is used to speed up the access to records without searching the entire file. The records of the file may be sorted in random sequence but the index table is in the start sequence on the key field. Thus the file can be processed randomly as well as sequentially.

Advantage :

- More efficient.
- Time required is less.



Example :

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

void indexedSequentialSearch(int arr[], int n, int k)
{
	int GN = 3; // GN is group number that is number of elements in a group
	int elements[3], indices[3], i, set = 0;
	int j = 0, ind = 0, start, end;
	for (i = 0; i < n; i += 3) {

		// Storing element
		elements[ind] = arr[i];

		// Storing the index
		indices[ind] = i;
		ind++;
	}
	if (k < elements[0]) {
		printf("Not found");
		exit(0);
	}
	else {
		for (i = 1; i <= ind; i++)
			if (k <= elements[i]) {
				start = indices[i - 1];
				end = indices[i];
				set = 1;
				break;
			}
	}
	if (set == 0) {
		start = indices[GN - 1];
		end = GN;
	}
	for (i = start; i <= end; i++) {
		if (k == arr[i]) {
			j = 1;
			break;
		}
	}
```

```
        if (j == 1)
                printf("Found at index %d", i);
        else
                printf("Not found");
}

// Driver code
void main()
{

        int arr[] = { 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
        int n = sizeof(arr) / sizeof(arr[0]);

        // Element to search
        int k = 8;
        indexedSequentialSearch(arr, n, k);
}
```

Output:

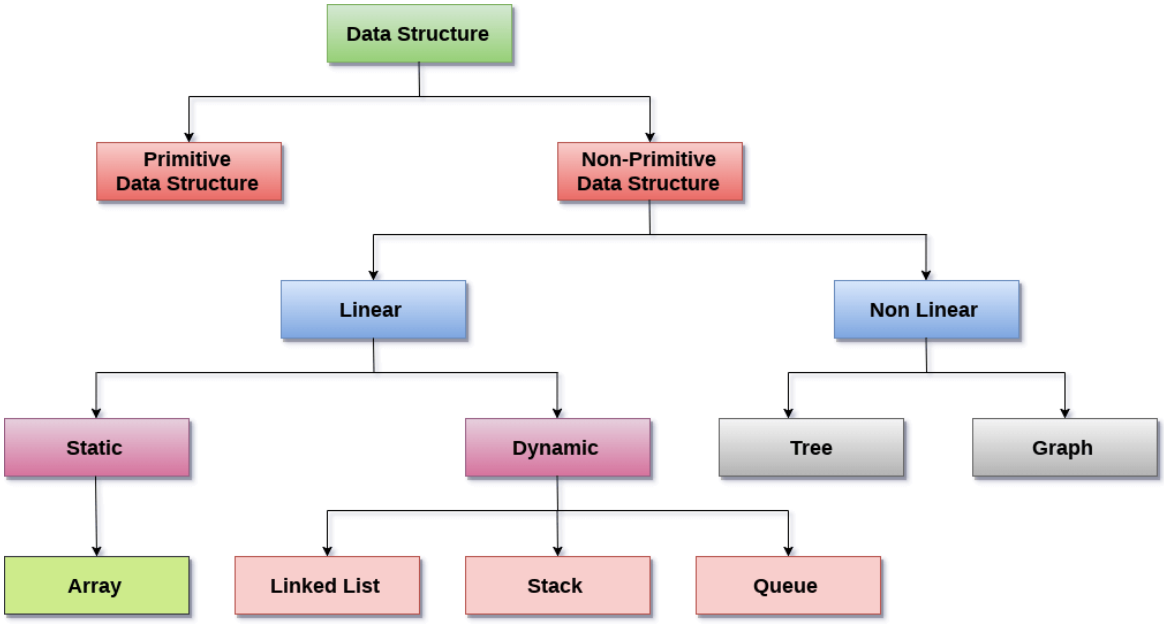
Found at index 2

Unit : 3 : Topic 1 : Introduction in Data Structure

Data Structure :

- Data Structure is a way to store and organize data so that it can be used efficiently.
- Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently.
- The data structure name indicates itself by organizing the data in memory.
- There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. Array is a collection of memory elements in which data is stored sequentially.

Types of Data Structures :



There are two types of data structures:

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

Linear Data Structure :

The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

Non-linear data structure :

When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in a random manner.

Data structures can also be classified as:

Static data structure:

It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.

Dynamic data structure:

It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Advantages of Data structures

The following are the advantages of a data structure:

Efficiency: If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.

Reusability: The data structures provide reusability means that multiple client programs can use the data structure.

Abstraction: The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

Operations on data structures :

Traversing: Every data structure contains a set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Insertion: Insertion can be defined as the process of adding the elements to the data structure at any location.If the size of data structure is **n** then we can only insert **n-1** data elements into it.

Deletion: The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.
If we try to delete an element from an empty data structure then underflow occurs.

Searching: The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

Sorting: The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

Merging: When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

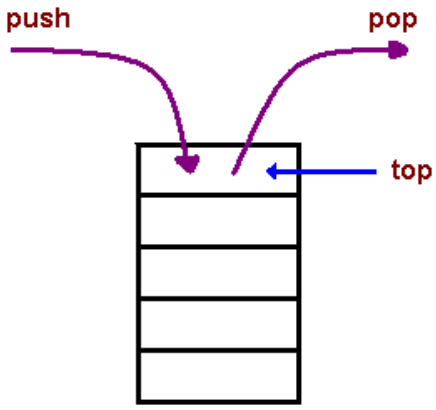
Unit : 3 : Topic : 2 : Elementary data structure :

Stack : (Stack)

- A stack is a container of objects that are inserted and removed according to the **Last-In First-Out (LIFO)** principle [**or FILO - First In Last Out**]
- In the stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack.
- A stack is a limited access data structure - elements can be added and removed from the stack only at the top.
- **Push** adds an item to the top of the stack, **pop** removes the item from the top.
- Stack is a linear data structure in which insertions and deletions of an element are done at one end which is known as **TOS (Top Of Stack).**
- Consider a stack of plates placed on the counter in a restaurant. During dinner time, plates are taken from the top of place(stack) and the waiter puts the washed plates on the top.
- The most important application of stack is recursion.
- It is also used in memory management and in operating systems

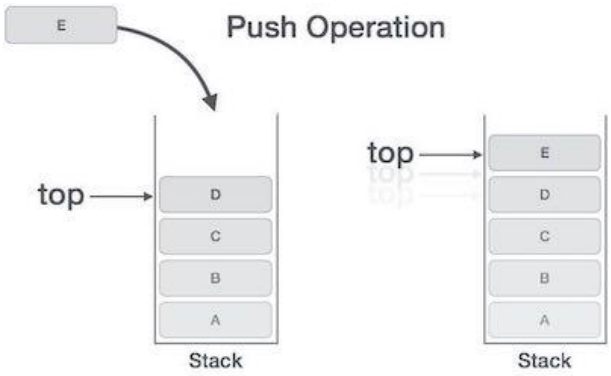
[**What is Recursion?**
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.]

Mainly the following three basic operations are performed in the stack:
Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
Peek or Top: Returns top element of stack.
isEmpty: Returns true if stack is empty, else false.
isFull : returns true if the stack is full else false.



Push Operation :
The process of putting a new data element onto a stack is known as a Push Operation. Push operation involves a series of steps −

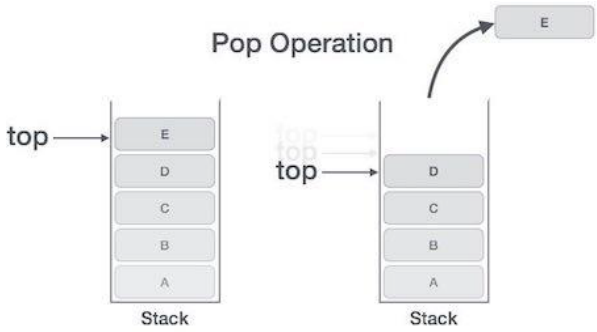
Step 1 − Check if the stack is full.
Step 2 − If the stack is full, produce an error and exit.
Step 3 − If the stack is not full, increment the top to point to the next empty space.
Step 4 − Adds data element to the stack location, where the top is pointing.
Step 5 − Returns success.



Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead the top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data elements and deallocates memory space.A Pop operation may involve the following steps −

Step 1 − Check if the stack is empty.
Step 2 − If the stack is empty, produce an error and exit.
Step 3 − If the stack is not empty, access the data element at which top is pointing.
Step 4 − Decreases the value of top by 1.
Step 5 − Returns success.



Peep operation

If we want to access some information stored at some location in the stack then peep operation is required. The index value is subtracted from tos

Step-1: If tos<0---stack is empty
Step-2: Input the position of the element that you want to read
Step-3: If(element<0 || element>tos+1)
Out of range
Step-4: Else Print peeped element
Stack[tos-pos+1]
Step-5: End

Display operation

Step-1: If(tos<0)—stack is empty
Step-2: else
for(i=tos;i>=0;i--)

print element that is stack[i]

Step-3: End

Example :

```
#include<stdio.h>
#include<conio.h>
#define size 100
int tos=-1;
int stack[size];
void push(int);
void display();
void peep();
void pop();
void main()
{
        clrscr();
        push(10);
        push(11);
        push(12);
        display();
        peep();
        pop();
        display();
        push(100);
        display();
        getch();
}
void push(int ele)
{
        if(tos>=size-1)
                printf("\nStack is full");
        else
                tos++;
                stack[tos]=ele;
}
void display()
{
        int i, count = 0;
        if(tos<0)
                printf("\nstack is empty");
        else
```

```
        {
                for(i=tos;i>=0;i--)
                {
                        printf("\nElement at position %d is %d",i+1,stack[i]);
                        count++;
                }
        }
        printf("\nNo of elements in this stack is : %d", count);
}
void peep()
{
        int pos;
        if(tos<0)
                printf("\nstack is empty");
        else
                printf("\nEnter value of pos:");
                scanf("%d",&pos);
                if(pos<0 || pos>tos+1)        //if(pos>=0 && pos<=tos)
                        printf("\nout of range");
                else
                        printf("\nPeeped element is %d",stack[tos-pos+1]);
}
void pop()
{
        if(tos<=-1 )
                printf("\nStack is empty");
        else
                printf("\nelement : %d deleted", stack[tos]);
                tos--;
}
```

Recursion and Stack :

- "Recursion" is the technique of solving any problem by calling the same function again and again until some breaking (base) condition where recursion stops and it starts calculating the solution from there on. For eg. calculating factorial of a given number
- The High level Programming languages, such as Pascal , C etc. that provide support for recursion use stack for bookkeeping.
- In each recursive call, there is need to save the
    - current values of parameters,
    - local variables and

    - the return address (the address where the control has to return from the call).
- Also, as a function calls another function, first its arguments, then the return address and finally space for local variables is pushed onto the stack.
- Recursion is extremely useful and extensively used because many problems are elegantly specified or solved in a recursive way.
- The example of recursion as an application of stack is keeping books inside the drawer and then removing each book recursively.

Example :

- Print Stack Elements from Bottom to Top using recursion :
- Given a stack s, the task is to print the elements of the stack from bottom to top, such that the elements are still present in the stack without their order being changed in the stack.
- The idea is to pop the element of the stack and call the recursive function PrintStack. Once the stack becomes empty start printing the element which was popped last and the last element that was popped was the bottom-most element. Thus, elements will be printed from bottom to top. Now push back the element that was printed, this will preserve the order of the elements in the stack.

Example :

```
#include <stdio.h>
#include<conio.h>
int top = -1;
int array[5];
//Checks if Stack is Full or not Adds an element to stack and then increment top index
void push(int num)
{
        if (top >= 4)
                printf("Stack is Full...\n");
        else
        {
                top++;
                array[top] = num;
        }
}
//Removes top element from stack and decrement top index
int pop()
{
        if (top <= -1)
                printf("Stack is Empty...\n");
        else
```

```
	{
		top = top - 1;
	}
	return array[top+1];
}
//Prints elements of stack using recursion
void printStack()
{
	int i;
	if(top <= -1)
		printf("\nStack is Empty");
	else
	{
		for(i=0; i<=top; i++)
		printf("%d  ", array[i]);
	}

}
void insertAtBottom(int item)
{
	if (top <= -1)
	{
		push(item);
	}
	else
	{
	/* Store the top most element of stack in top variable and recursively call
insertAtBottom for rest of the stack */
		int top = pop();
		insertAtBottom(item);

	/* Once the item is inserted at the bottom, push the  top element back to stack */
		push(top);
	}
}
void reverse()
{
	if (top != -1)
	{
```

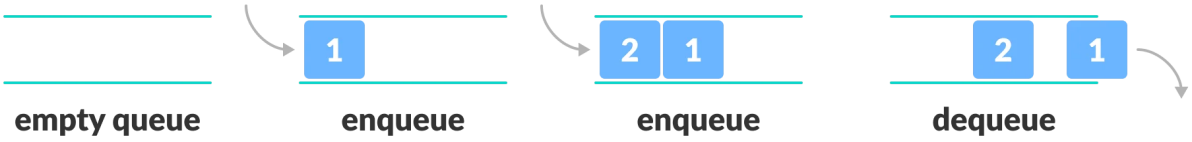
```
        /* keep on popping top element of stack in every recursive call till stack is empty
*/
                int top = pop();
                reverse();

        /* Now, instead of inserting element back on top of stack, we will insert it at the
bottom of stack */
                insertAtBottom(top);
        }
}
int main()
{
        clrscr();
        push(1);
        push(2);
        push(3);
        push(4);
        push(5);
        printf("Original Stack\n");
        printStack();
        reverse();
        printf("\nReversed Stack\n");
        printStack();
        getch();
        return 0;
}
```

Evaluation of expressions using stack :
[Prefix , Postfix , Infix - from COA material]

Queue : (Queue)

- Queue is the linear data structure in which information is based on FIFO(First In First Out) or FCFS(First Come First Server)
- In queue insertion of an element is performed at one end known as back or rear end and deletion is performed at another end known as front end.
- Insertion operation is known as enqueue and deletion operation is known as dequeue.
- A queue is open at both its ends; one end is always used to insert data (enqueue) and the other is used to remove data (dequeue).
- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

- Queue follows the FIFO rule - the element added first is the element removed first.



Basic Operations of Queue

- **Enqueue:** Add an element to the end of the queue
- **Dequeue:** Remove an element from the front of the queue
- **IsEmpty:** Check if the queue is empty
- **IsFull:** Check if the queue is full
- **Peek:** Get the value of the front of the queue without removing it

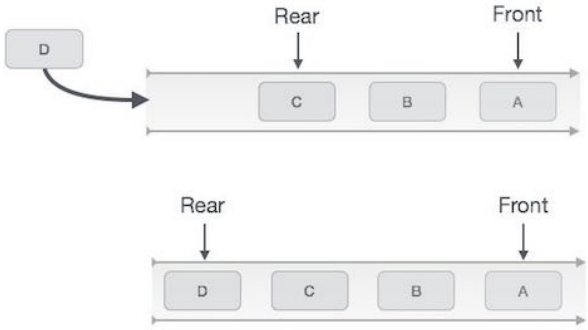
[Front: Pointer element responsible for fetching the first element from the queue
Rear: Pointer element responsible for fetching the last element from the queue]

Algorithm :

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueuing (or storing) data in the queue we take help of rear pointer

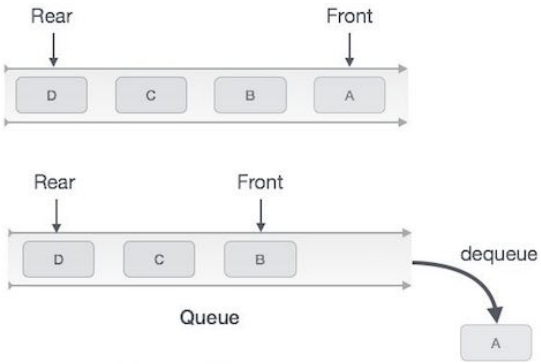
1. **Insert operation (Enqueue) :**

(we have to define size first)
Step 1 : if(rear=>size)
Queue is full
Step 2 : rear++;
queue[rear]=ele;
Step 3 : if(rear==0)
front=0;
Step 4 : End



Delete operation (Dequeue) :

Step 1 : if(front<0)
Output : Queue is empty
Step 2 :if(front==rear) //Single element
front=-1;
rear=-1;
Step 3 : front++;
Step 4 : Stop



Display operation :

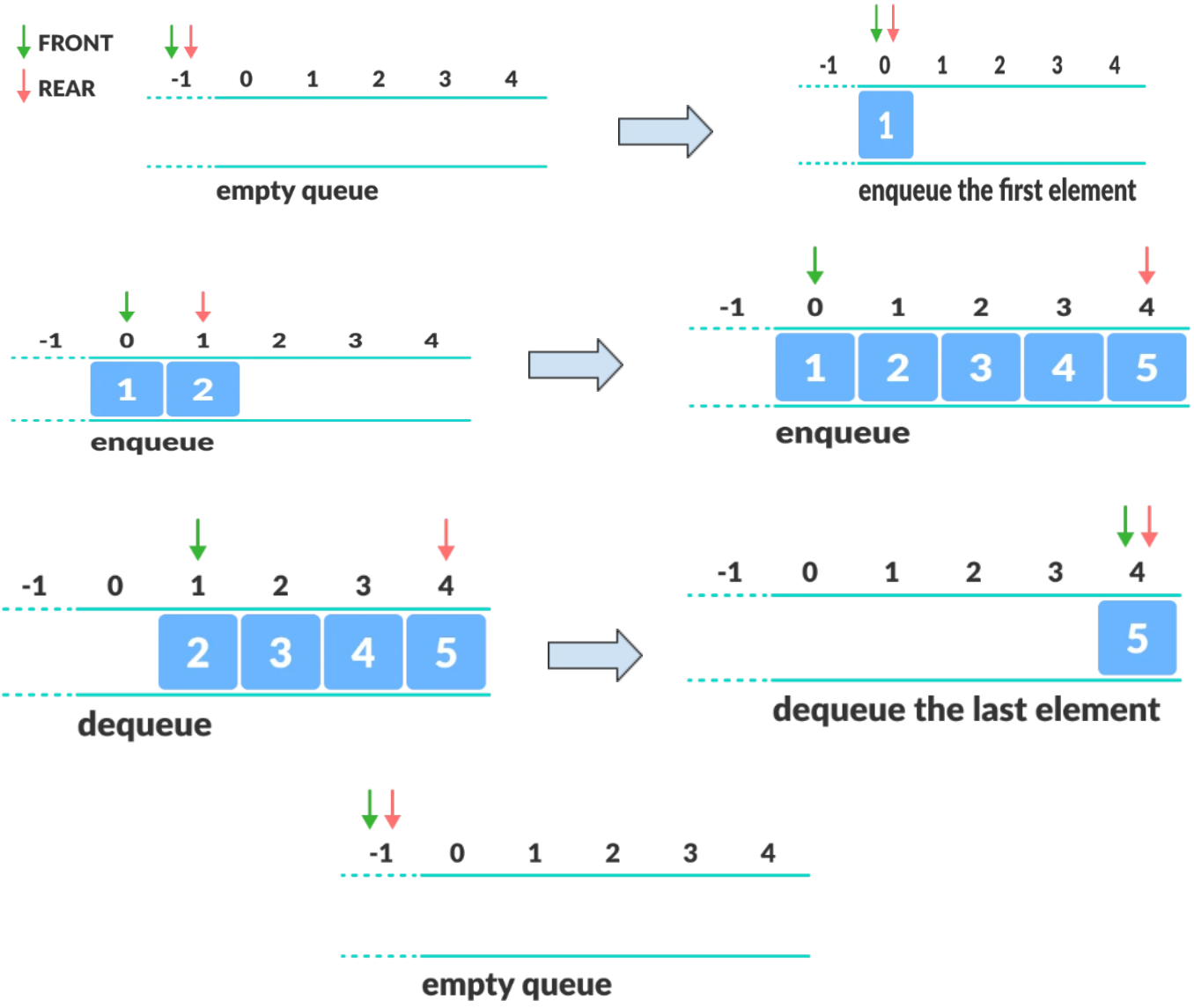
Step-1: If front<0

Output : queue is Empty

Step-2:for(i=front;i<=rear;i++)

Print : queue[i]

Step-3:End



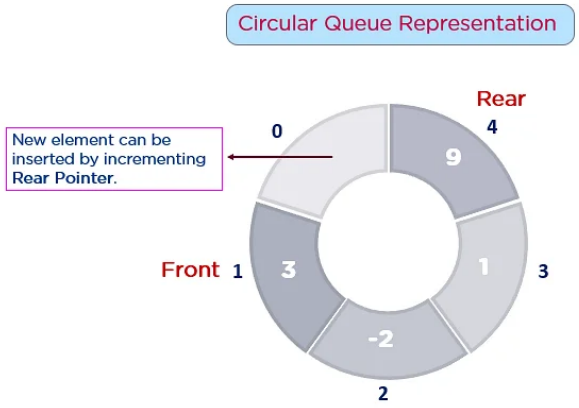
Example :

```
#include<stdio.h>
#include<conio.h>
#define size 100
int front=-1,rear=-1;
int queue[size];
void enqueue(int);
void dequeue();
```

```
void display();
void main()
{
	clrscr();
	enqueue(10);
	enqueue(11);
	enqueue(12);
	enqueue(13);
	enqueue(14);
	display();
	printf("\n\n");
	dequeue();
	display();
	getch();
}
void enqueue(int ele)
{
	if(rear>=size)
		printf("\nQueue is full");
	else
	{
		rear++;
		queue[rear]=ele;
		if(rear==0)
		front=0;
	}
}
void dequeue()
{
	if(front<0)
		printf("\nQueue is empty");
	else
	{
		if(front==rear)
		{
			front=-1;
			rear=-1;
		}
		else
			front++;
```

```
        }
}
void display()
{
        int i;
        if(front<0)
                printf("\nQueue empty");
        else
        {
                for(i=front;i<=rear;i++)
                {
                        printf("\nElement at position %d is %d",i,queue[i]);
                }
        }
}
```

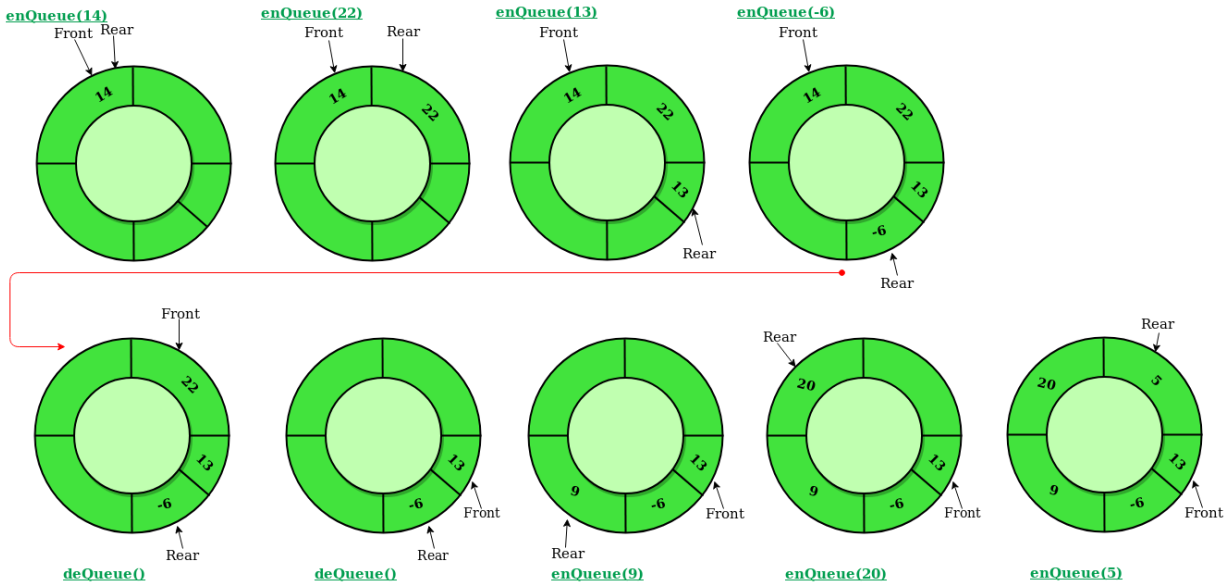
Circular Queue : (circular queue)



- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
- There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be the possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.
- A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a **Ring Buffer.**

Operations on Circular Queue

- **Front:** It is used to get the front element from the Queue.
- **Rear:** It is used to get the rear element from the Queue.
- **enqueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **dequeue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.



Applications of Circular Queue :

The circular Queue can be used in the following scenarios:

- **Memory management:** The circular queue provides memory management. As we have already seen in the linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- **Traffic system:** In a computer-control traffic system, traffic lights are one of the best examples of the circular queue. Each traffic light gets ON one by one after every interval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After the green light, the red light gets ON.

Algorithm :

1. **Insert operation :**

Step-1: If front==1 and rear==size (output queue overflow)
Step-2:else if (front==0)
Front=rear=1
q[rear]=element
Step-3:else if (rear==size)
rear=1
q[rear]=element
Step-4:else
rear=rear+1
q[rear]=element
Step-5: End

2. **Delete operation :**

Step-1: If front==0 (output queue is empty)
Step-2: If front==rear
front=rear=0
Step-3: else if (front==size)
front=1
Step-4:else
front=front+1
Step-5: End

3. **Display operation :**

Step-1: If front==0 (output queue is empty)
Step-2: else if (front >rear)
for(i=1;i<=rear;i++)
printf("%d",queue[i])
for(i=front;i<=size;i++)
printf("%d",queue[i])
Step-3: else
for(i=front;i<=rear;i++)
printf("%d",queue[i])
Step-4:END

Example :

```
#include<stdio.h>
#include<conio.h>
#define size 5
int queue[size];
int front=-1,rear=-1;
void insert(int);
void deleted();
void display();
void main()
{
        clrscr();
        insert(10);
        insert(20);
        insert(30);
        insert(40);
        insert(50);
        display();
        printf("\n\n");
```

```
	deleted();
	display();
	printf("\n\n");
	insert(60);
	display();
	getch();
}
void insert(int ele)
{
	if(front==0 && rear==size-1)
		printf("\nQueue is full");
	else if(front==-1)
	{
		front=rear=0;
		queue[rear]=ele;
	}
	else if(rear==size-1)
	{
		rear=0;
		queue[rear]=ele;
	}
	else
	{
		rear++;
		queue[rear]=ele;
}
}
void deleted()
{
	if(front==-1)
		printf("\nQueue is empty");
	else if(front==rear)
		front=rear=-1;
	else if(front==size)
		front=0;
	else
		front++;
}
void display()
{
```

```
        int i;
        if(front==-1)
                printf("\nQueue is empty");
        else if(front>rear)
        {
                for(i=0;i<=rear;i++)
                {
                        printf("\nElement at %d is %d",i,queue[i]);
                }
                for(i=front;i<=size-1;i++)
                {
                        printf("\nElement at %d is %d",i,queue[i]);
                }
        }
        else
        {
                for(i=front;i<=rear;i++)
                {
                        printf("\nElement at %d is %d",i,queue[i]);
                }
        }
}
```

Deque : (Dequeue)

- A deque, also known as a **double-ended queue**, is an ordered collection of items similar to the queue.
- It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items.
- New items can be added at either the front or the rear. Likewise, existing items can be removed from either end.
- It is also known as a **head-tail linked list** because elements can be added to or removed from either the front (head) or the back (tail) end. However, no element can be added and deleted from the middle.
- In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.
- It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures.

Types of deque :

There are two types of deque -

- Input restricted queue
- Output restricted queue

Input restricted Queue

In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.

Output restricted Queue

In the output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.

Operations performed on deque

There are the following operations that can be applied on a deque -

- **insertFront()**: Adds an item at the front of Deque.
- **insertLast()**: Adds an item at the rear of Deque.
- **deleteFront()**: Deletes an item from front of Deque.
- **deleteLast()**: Deletes an item from the rear of Deque.

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements. So, in addition to the above operations, following operations are also supported in deque -

- **getFront()**: Gets the front item from the queue.
- **getRear()**: Gets the last item from the queue.
- **isEmpty()**: Checks whether Deque is empty or not.
- **isFull()**: Checks whether Deque is full or not.

Priority Queue :

- A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue.
- In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.
- The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.
- For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

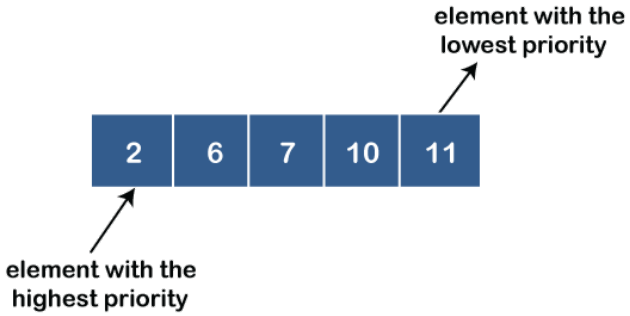
Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

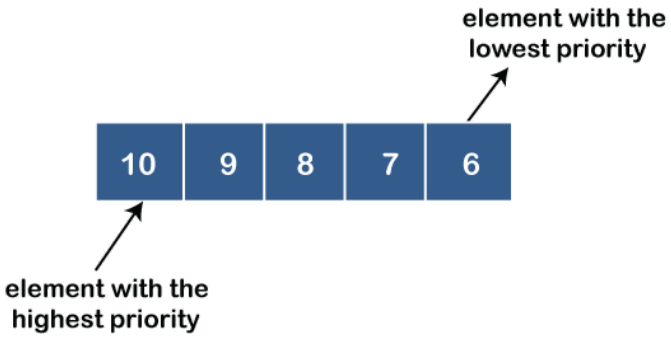
- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Types of Priority Queue

1. **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



2. **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Extra material for chapter no 3 :

Linked list implementation for stack :

a. Push operation

Step-1:Allocate the memory to the node (node *t)

```
t=(node *)malloc(sizeof(node));
```

Step-2:Assign data part and next part of node

```
t->info=ele;
t->next=tos;
top=t;
```

Step-3: End

b. Pop operation

Step-1:Check whether stack is empty or not.

```
If(tos==NULL)
Stack is empty
```

Step-2: If the stack is not empty then

```
t=tos
tos=t->next
free(t)
```

Step-3: End

c. Peep operation

Step-1: Enter the position from user

Step-2:Initialize i=0,j=1;

Step-3:node *temp

```
for(temp=tos; temp!=NULL;temp=temp->next)
i++;
```

Step-4:if(pos>i)—stack is full

Step-5:else

```
temp=tos;
for(j=1;j<pos;j++)
```

temp=temp->next;
printf(“%d”,temp->info)

Step-5: End

d. Display operation

Step-1: while(temp!=NULL)
printf(“%d”,temp->info)
temp=temp->next;

Step-2: End

e. Update operation

Step-1: Enter the position from user
Step-2:Initialize i=0,j=1;
Step-3:node *temp
for(temp=tos; temp!=NULL;temp=temp->next)
i++;
Step-4:if(pos>i)—stack is full
Step-5:else
temp=tos;
new1=(node *)malloc(sizeof(node)
scanf(“%d”,new1->info)
for(j=1;j<pos;j++)
temp=temp->next;
temp->info=new1->info
Step-5: End

Program :

```
#include<stdio.h>
#include<conio.h>
struct list
{
        int info;
        struct list *next;
};
typedef struct list node;
struct list *new1;
node *tos;
void push(int);
void pop();
void peep();
void display();
void update();
void main()
```

```
{
        tos=NULL;
        clrscr();
        push(10);
        push(11);
        push(12);
        push(13);
        push(14);
        display();
        pop();
        display();
        peep();
        update();
        display();
        getch();
}
void push(int ele)
{
        node *t;
        t=(node *)malloc(sizeof(node));
        t->info=ele;
        t->next=tos;
        tos=t;
}
void display()
{
        node *temp;
        temp=tos;
        while(temp!=NULL)
        {
                printf("\nStack Element is:%d",temp->info);
                temp=temp->next;
        }
}
void pop()
{
        node *temp;
        if(tos==NULL)
                printf("\nStack is empty");
        else
```

```
                temp=tos;
        tos=temp->next;
        free(temp);
}
void peep()
{
        int pos,i,j;
        node *temp;
        printf("\nEnter position:");
        scanf("%d",&pos);
        for(temp=pos;temp!=NULL;temp=temp->next)
        {
                i++;
        }
        if(pos>i)
        printf("\nStack is empty");
        else
        {
                temp=tos;
                for(j=1;j<pos;j++)
                {
                        temp=temp->next;
                }
                printf("\nValue at %d is %d",pos,temp->info);
        }
}
void update()
{
        int pos,i=0,j;
        node *temp;
        printf("\nEnter position:");
        scanf("%d",&pos);
        for(temp=pos;temp!=NULL;temp=temp->next)
        {
                i++;
        }
        if(pos>i)
                printf("\nStack is full");
        else
        {
```

```
                new1=(node *)malloc(sizeof(node));
                printf("\nenter new value");
                scanf("%d",&new1->info);
                temp=tos;
                for(j=1;j<pos;j++)
                {
                        temp=temp->next;
                }
                temp->info=new1->info;
        }
}
```

Linked list representation on queue :

a. Insert operation

Step-1:Allocate the memory to the node (node *q)

```
        q=(node *)malloc(sizeof(node));
```

Step-2:Assign data part and next part of node

```
        q->info=ele;
        q->next=NULL;
```

Step-3:If queue is empty then:

```
        if(front==NULL)
        front=rear=q
```

Step-4:If queue is not empty

```
        rear-next=q;
        rear=q;
```

Step-5: End

b. Delete operation

Step-1:Check whether the queue is empty or not.

Step-2: If the queue is empty then

```
        If(front==NULL)---queue empty
```

Step-3: If queue is not empty then

```
        node *q
        q=front
        front=q->next
```

Step-4:free(q)

Step-5:End

c. Display operation

Step-1: node *q

```
        q=front
```

Step-2:Check if queue is empty or not

If(q==NULL)---queue empty
Step-3: If queue is not empty then
While(q!=NULL)
Printf("%d",q->info);
q=q->next
Step-4:END

Program

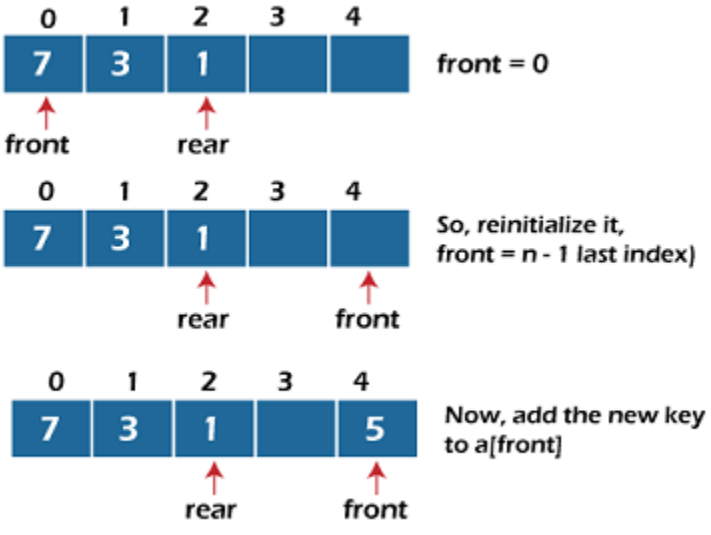
```
#include<stdio.h>
#include<conio.h>
struct list
{
        int info;
        struct list *next;
};
typedef struct list node;
node *front,*rear;
void insert(int);
void deleted();
void display();
void main()
{
        front=rear=NULL;
        clrscr();
        insert(10);
        insert(11);
        insert(12);
        display();
        deleted();
        display();
        getch();
}
void insert(int ele)
{
        node *t;
        t=(node *)malloc(sizeof(node));
        t->info=ele;
        t->next=NULL;
        if(front==NULL)
        {
                front=rear=t;
```

```
        }
        else
        {
                rear->next=t;
                rear=t;
        }
}
void deleted()
{
        node *temp;
        temp=front;
        if(temp==NULL)
                printf("\nQueue is empty");
        else
                front=temp->next;
                free(temp);
}
void display()
{
        node *temp;
        int i=0;
        temp=front;
        if(temp==NULL)
                printf("\nQueue is empty");
        else
        {
                while(temp!=NULL)
                {
                        printf("\nElement is %d",temp->info);
                        temp=temp->next;
                }
        }
}
```

Operation performed on deque using an example.

Insertion at the front end

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check

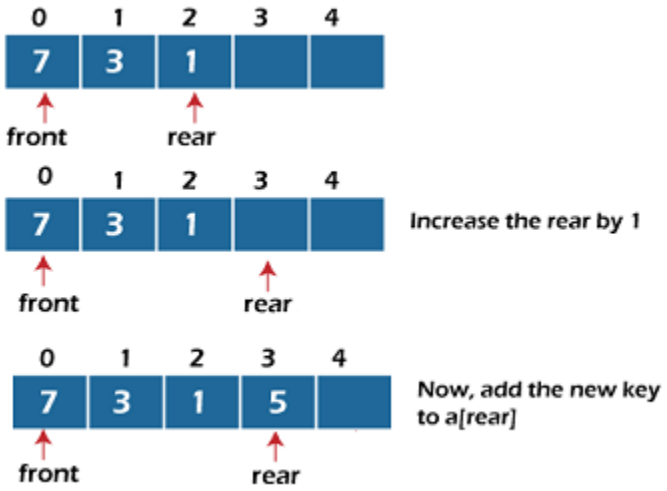


whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by front = n - 1, i.e., the last index of the array.

Insertion at the rear end

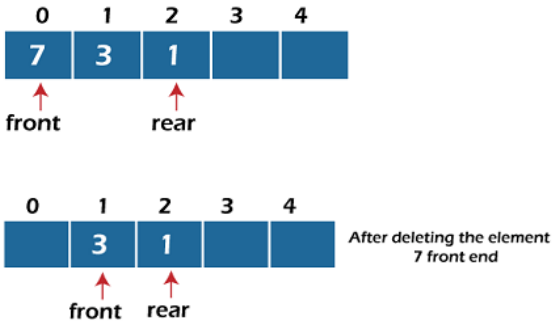
In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions -



- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.

Deletion at the front end

In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.



If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

If the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).

Deletion at the rear end

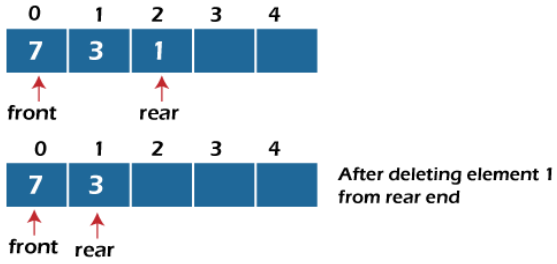
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).



Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

Deque program :

```
#include <stdio.h>
#define size 5
int deque[size];
int f = -1, r = -1;
//  insert_front function will insert the value from the front
void insert_front(int x)
{
   if((f==0 && r==size-1) || (f==r+1))
   {
      printf("Overflow");
   }
   else if((f==-1) && (r==-1))
   {
      f=r=0;
      deque[f]=x;
   }
   else if(f==0)
   {
      f=size-1;
      deque[f]=x;
```

```
    }
    else
    {
        f=f-1;
        deque[f]=x;
    }
}

// insert_rear function will insert the value from the rear
void insert_rear(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("Overflow");
    }
    else if((f==-1) && (r==-1))
    {
        r=0;
        deque[r]=x;
    }
    else if(r==size-1)
    {
        r=0;
        deque[r]=x;
    }
    else
    {
        r++;
        deque[r]=x;
    }

}

// display function prints all the value of deque.
void display()
{
    int i=f;
    printf("\nElements in a deque are: ");

    while(i!=r)
```

```
    {
        printf("%d ",deque[i]);
        i=(i+1)%size;
    }
     printf("%d",deque[r]);
}

// getfront function retrieves the first value of the deque.
void getfront()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the element at front is: %d", deque[f]);
    }

}

// getrear function retrieves the last value of the deque.
void getrear()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the element at rear is %d", deque[r]);
    }

}

// delete_front() function deletes the element from the front
void delete_front()
{
    if((f==-1) && (r==-1))
    {
```

```
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=-1;
        r=-1;

    }
     else if(f==(size-1))
     {
         printf("\nThe deleted element is %d", deque[f]);
         f=0;
     }
     else
     {
          printf("\nThe deleted element is %d", deque[f]);
          f=f+1;
     }
}
// delete_rear() function deletes the element from the rear
void delete_rear()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[r]);
        f=-1;
        r=-1;

    }
     else if(r==0)
     {
         printf("\nThe deleted element is %d", deque[r]);
         r=size-1;
     }
     else
```

```
    {
        printf("\nThe deleted element is %d", deque[r]);
        r=r-1;
    }
}
int main()
{
   insert_front(20);
   insert_front(10);
   insert_rear(30);
   insert_rear(50);
   insert_rear(80);
   display();  // Calling the display function to retrieve the values of deque
   getfront();  // Retrieve the value at front-end
   getrear();  // Retrieve the value at rear-end
   delete_front();
   delete_rear();
   display(); // calling display function to retrieve values after deletion
   return 0;
}
```

Unit 4 : Linked List and implementation

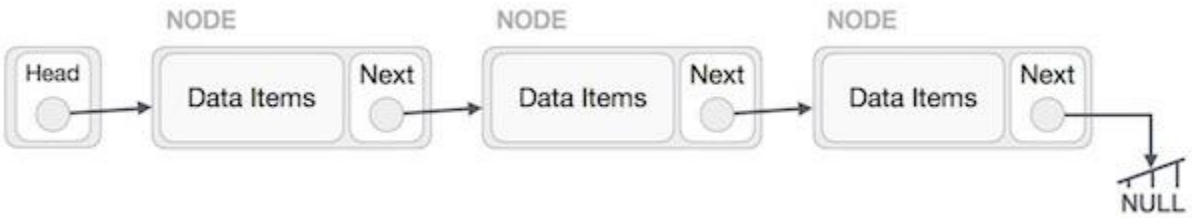
- Linked list is the second most-used data structure after array.
- Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.
- In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Following are the important terms to understand the concept of Linked List.

- Link − Each link of a linked list can store a data called an element.
- Next − Each link of a linked list contains a link to the next link called Next.
- LinkedList − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Things to Remember about Linked List

- head points to the first node of the linked list
- the next pointer of the last node is NULL, so if the next current node is NULL, we have reached the end of the linked list.

Types of Linked List

Following are the various types of linked lists.

- **Simple Linked List** − Item navigation is forward only. (singly linked list)
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains a link of the first element as next and the first element has a link to the last element as previous.
- **Header linked list** - In Header linked list, we have a special node present at the beginning of the linked list. This special node is used to store the number of nodes present in the linked list.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.
- **Traversal** - access each element of the linked list

Representation of Linked List

Let's see how each node of the linked list is represented. Each node consists:

- A data item
- An address of another node

We wrap both the data item and the next node reference in a struct as:

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

Singly Linked List:

- Singly linked list is the most basic of all the linked data structure
- A singly linked list is a collection of nodes and each node contains the pointer to next element
- In the singly link list we can move from left to right that is only in one dimension but we cannot return back.
- Each node represents structure, containing variables for information and a structure pointer to itself.

Create operation :

Step-1 Check whether any node exists in the list or not

Step-2: If list is Empty then create node

```
if(q==NULL)
{
        q=(node *)malloc(sizeof(node));
        q->data=info;
        q->next=NULL;
}
```

Step-3: If the list is not Empty then create the node after the last node

```
while(q->next!=NULL)
{
        q=q->next;
```

```
    }
    q->next=(node *)malloc(sizeof(node));
    q->next->data=info;
    q->next->next=NULL;
    Step-4: End
```

Display operation :

Step-1: while(q!=NULL)

```
    {
        printf("\nElement is %d",q->info);
        q=q->next;
    }
```

Step-2: End

Add node at beginning operation :

Step-1: Allocate the memory to the node P that is to be inserted in the beginning

```
    p=(node *)malloc(sizeof(node))
```

Step-2: Allocate the data and pointer to next node part

```
    p->data=info
    p->next=q;
```

Step-3: End

Add node at after operation :

Step-1: Take a temporary node which is to be inserted (node *temp)

Step-2: for(i=1;i<pos;i++)

```
    {
        q=q->next;
        if(q==NULL)
        printf("\nout of range");
    }
```

Step-3: Allocate memory to temporary node and data and next part

```
    temp=(node *)malloc(sizeof(node))
    temp->info=ele
    temp->next=q->next;
    q->next=temp;
```

Step-4: End

Count operation :

Step-1: Initialize one counter variable for counting total no of nodes (c=0)

Step-2:while(q!=NULL)

```
    {
        q=q->next;
        c++;
```

```
        }
        return c;
```
Step-3: End

Example :

```
#include<stdio.h>
#include<conio.h>
struct list
{
        int info;
        struct list *next;
};
typedef struct list node;
node *p;
void create(int,node *);
void display(node *);
void addbeg(int,node *);
void addafter(int,int,node *);
void deleted(int,node *);
int count(node *);
void sort(node *);
void search(int,node *);
void main()
{
        p=NULL;
        clrscr();
        create(10,p);
        create(20,p);
        create(5,p);
        display(p);
        printf("\n\n");
        addbeg(0,p);
        printf("After adding element at beginning of the Linked List :\n");
        display(p);
        printf("\n\n");
        addafter(2,500,p);
        printf("After adding element in specific position:\n");
        display(p);
        printf("\n\n");
        deleted(20,p);
        printf("After Deleted elements are :\n");
```

```
	display(p);
	getch();
}
void create(int ele,node *q)
{
	if(q==NULL)
	{
		p=(node *)malloc(sizeof(node));
		p->info=ele;
		p->next=NULL;
	}
	else
	{
		while(q->next!=NULL)
		{
			q=q->next;
		}
		q->next=(node *)malloc(sizeof(node));
		q->next->info=ele;
		q->next->next=NULL;
	}
}
void display(node *q)
{
	while(q!=NULL)
	{
		printf("\t%d",q->info);
		q=q->next;
	}
}
int count(node *q)
{
	int c=0;
	while(q!=NULL)
	{
		q=q->next;
		c++;
	}
	return (c);
}
```

```
void addbeg(int ele,node *q)
{
	p=(node *)malloc(sizeof(node));
	p->info=ele;
	p->next=q;
}
void addafter(int c,int ele,node *q)
{
	node *temp;
	int i;
	for(i=1;i<c;i++)
	{
		q=q->next;
		if(q==NULL)
		{
			printf("\nposition is out of range");
			return;
		}
	}
	temp=(node *)malloc(sizeof(node));
	temp->info=ele;
	temp->next=q->next;
	q->next=temp;
}
void deleted(int ele,node *q)
{
	node *temp;
	if(q->info==ele)
	{
		p=q->next;
		free(q);
		return;
	}
	while(q->next->next!=NULL)
	{
		if(q->next->info==ele)
		{
			temp=q->next;
			 q->next=q->next->next;
			free(temp);
```

```
                return;
            }
            q=q->next;
        }
}
```

[What is typedef : The typedef is a keyword used in C programming to provide some meaningful names to the already existing variable in the C program. It behaves similarly as we define the alias for the commands. In short, we can say that this keyword is used to redefine the name of an already existing variable.**]**

Merge of Singly Linked List :

Merging is the process of combining two sorted lists into a single sorted list.
Here we have two linked lists and we will combine these two into a single sorted list.
We just need to follow some very simple steps and the steps to join two lists (say ‘a’ and ‘b’) are as follows:

1. Traverse over the linked list ‘a’ until the element next to the node is not NULL.
2. If the element next to the current element is NULL (a->next == NULL) then change the element next to it to ‘b’ (a->next = b).

Example :

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
void display(struct node *head)
{
   if(head != NULL)
   {
        printf("\nElements after Merging is :");
        printf("%d\n", head -> data);
        display(head->next);
   }
}
void concatenate(struct node *a,struct node *b)
{
   if( a != NULL && b!= NULL )
   {
        if (a->next == NULL)
```

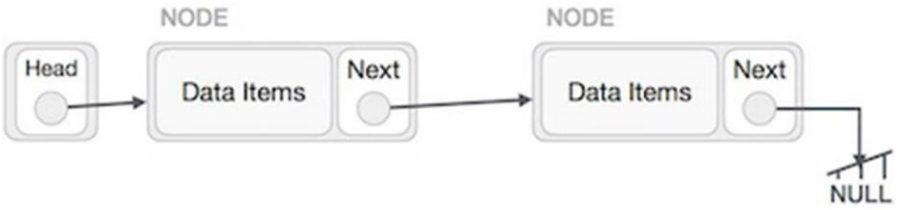
```
            a->next = b;
        else
            concatenate(a->next,b);
    }
    else
    {
        printf("Either a or b is NULL\n");
    }
}
void main()
{
    struct node *prev,*a, *b, *p;
    int n,i;
    clrscr();
    printf ("\n Enter number of elements in a:");
    scanf("%d",&n);
    printf("\nEnter %d elements : ", n);
    a=NULL;
    for(i=0;i<n;i++)
    {
        p=malloc(sizeof(struct node));
        scanf("%d",&p->data);
        p->next=NULL;
        if(a==NULL)
            a=p;
        else
            prev->next=p;
                prev=p;
    }
    printf ("\nEnter number of elements in b:");
    scanf("%d",&n);
    printf("\nEnter %d elements", n);
    b=NULL;
    for(i=0;i<n;i++)
    {
        p=malloc(sizeof(struct node));
        scanf("%d",&p->data);
        p->next=NULL;
        if(b==NULL)
            b=p;
```

```
        else
            prev->next=p;
        prev=p;
    }
    concatenate(a,b);
    display(a);
    getch();
}
```

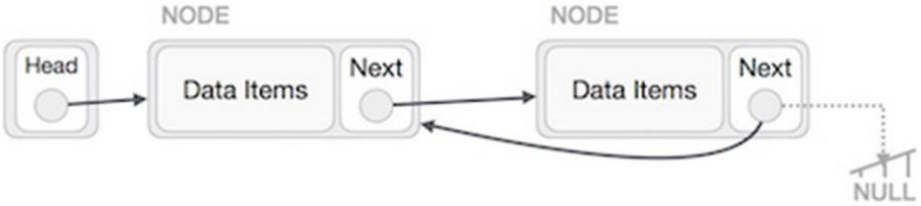
Reverse singly linked list :

In order to reverse a LinkedList in place, we need to reverse the pointers such that the next element now points to the previous element.

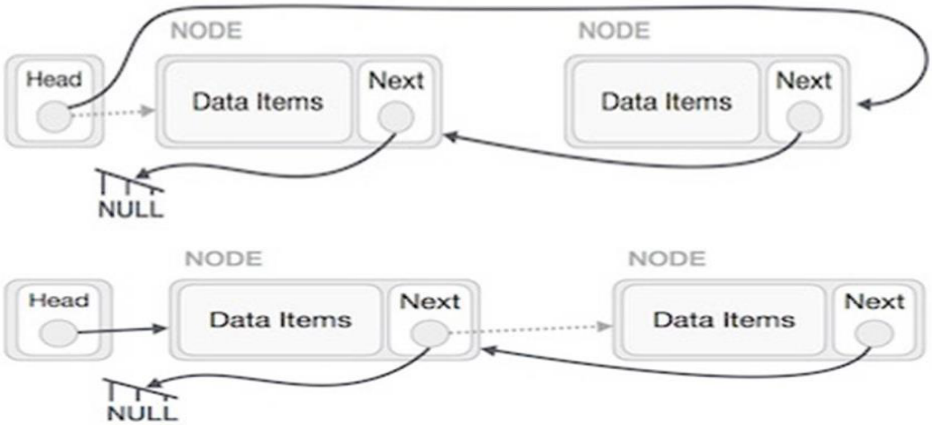
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.
We'll make the head node point to the new first node by using the temp node.

Program to reverse a Singly Linked List

```
#include <stdio.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
}*head;
void createList();
void reverseList();
void displayList();

void main()
{
   createList();
   printf("\nData in the list \n");
   displayList();
   reverseList();
   printf("\nData in the list\n");
   displayList();
}
void createList()
{
   struct node *newNode, *temp;
   int data, i;
   head = (struct node *)malloc(sizeof(struct node));
   printf("Enter the data of node 1: ");
   scanf("%d", &data);

   head->data = data;
   head->next = NULL;
   temp = head;
   for(i=2; i<=4; i++)
   {
        newNode = (struct node *)malloc(sizeof(struct node));
        printf("Enter the data of node %d: ", i);
        scanf("%d", &data);
```

```
            newNode->data = data;
            newNode->next = NULL;
            temp->next = newNode;
            temp = temp->next;
        }
printf("SINGLY LINKED LIST CREATED SUCCESSFULLY\n");
}
void reverseList()
{
    struct node *prevNode, *curNode;
    if(head != NULL)
    {
        prevNode = head;
        curNode = head->next;
        head = head->next;
        prevNode->next = NULL;

        while(head != NULL)
        {
            head = head->next;
            curNode->next = prevNode;
            prevNode = curNode;
            curNode = head;
        }
     head = prevNode; // Make last node as head
     printf("SUCCESSFULLY REVERSED LIST\n");
    }
}
void displayList()
{
    struct node *temp;
    if(head == NULL)
    {
        printf("List is empty.");
    }
    else
    {
        temp = head;
        while(temp != NULL)
        {
```

```
            printf("Data = %d\n", temp->data);
            temp = temp->next;
        }
    }
}
```

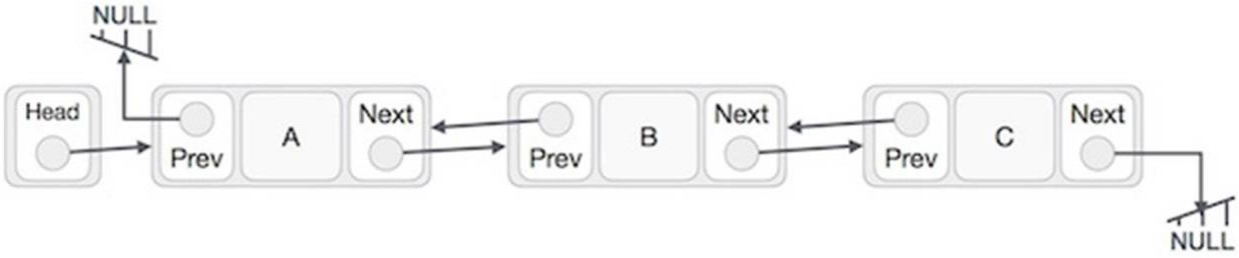
Doubly Linked List :

- Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.
- A Doubly Linked List (DLL) contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.
- In a singly linked list, we could traverse only in one direction, because each node contains the address of the next node and it doesn't have any record of its previous nodes.
- However, doubly linked lists overcome this limitation of singly linked lists.
- Due to the fact that each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Following are the important terms to understand the concept of doubly linked list.

- **Info** − Each link of a linked list can store a data called an element or information.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **Prev** − Each link of a linked list contains a link to the previous link called Prev.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

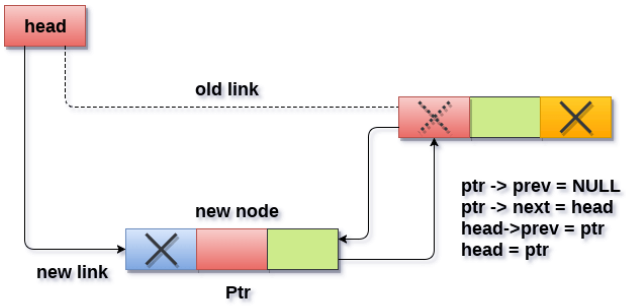
- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic Operations

1. Adding the node into the linked list at the beginning.

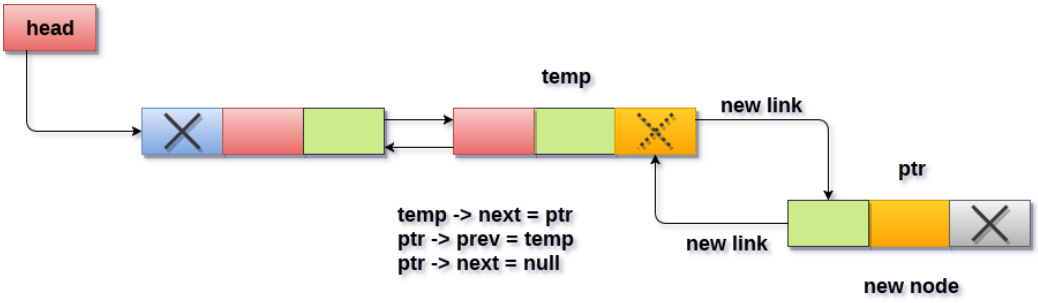
Step 1: IF ptr = NULL

Write OVERFLOW
Go to Step 9 [END OF IF]
Step 2: SET NEW_NODE = ptr
Step 3: SET ptr = ptr -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET head -> PREV = NEW_NODE
Step 8: SET head = NEW_NODE
Step 9: EXIT



2. Adding the node into the linked list to the end.

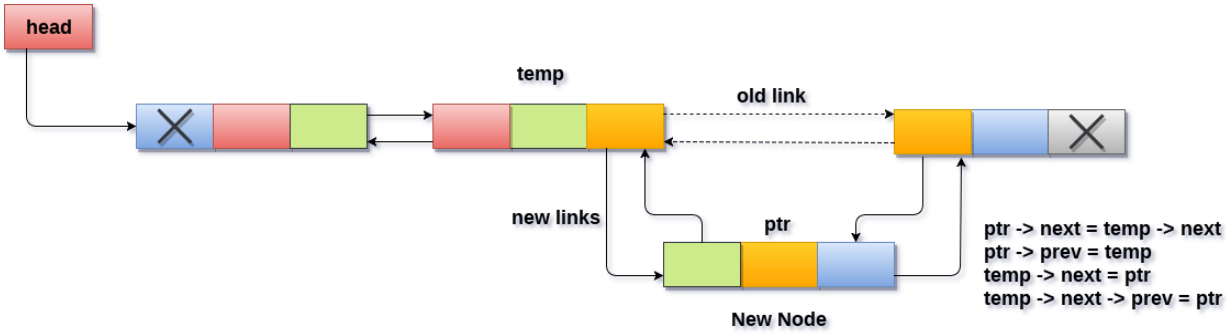
Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 11 [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET TEMP = START
Step 7: Repeat Step 8 while TEMP -> NEXT != NULL
Step 8: SET TEMP = TEMP -> NEXT
[END OF LOOP]
Step 9: SET TEMP -> NEXT = NEW_NODE
Step 10C: SET NEW_NODE -> PREV = TEMP
Step 11: EXIT



3. Adding the node into the linked list after the specified node :

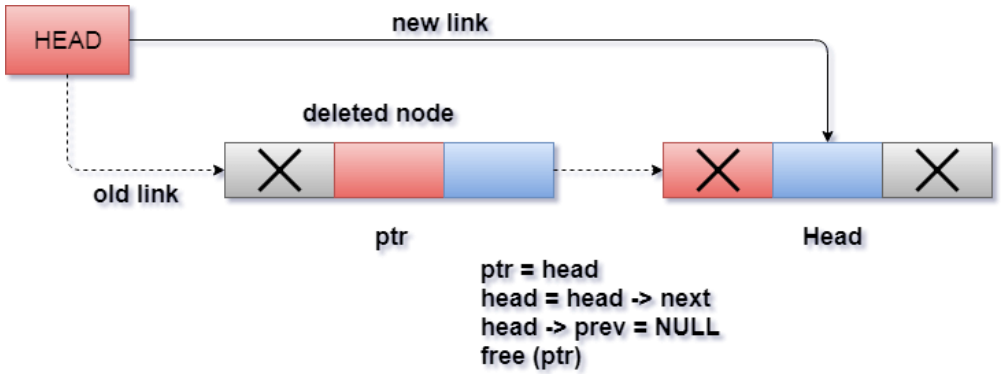
Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 15 [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET TEMP = START
Step 6: SET I = 0
Step 7: REPEAT 8 to 10 until I
Step 8: SET TEMP = TEMP -> NEXT
STEP 9: IF TEMP = NULL
STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
GOTO STEP 15 [END OF IF] [END OF LOOP]
Step 11: SET NEW_NODE -> NEXT = TEMP -> NEXT
Step 12: SET NEW_NODE -> PREV = TEMP
Step 13 : SET TEMP -> NEXT = NEW_NODE
Step 14: SET TEMP -> NEXT -> PREV = NEW_NODE
Step 15: EXIT



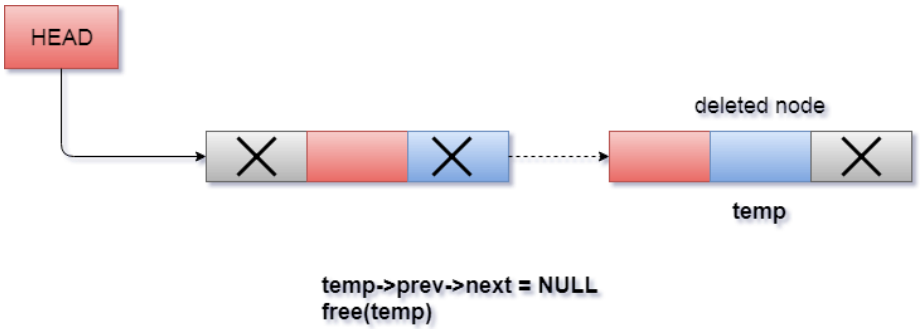
4. Removing the node from beginning of the list

STEP 1: IF HEAD = NULL
WRITE UNDERFLOW
GOTO STEP 6
STEP 2: SET PTR = HEAD
STEP 3: SET HEAD = HEAD → NEXT
STEP 4: SET HEAD → PREV = NULL
STEP 5: FREE PTR
STEP 6: EXIT



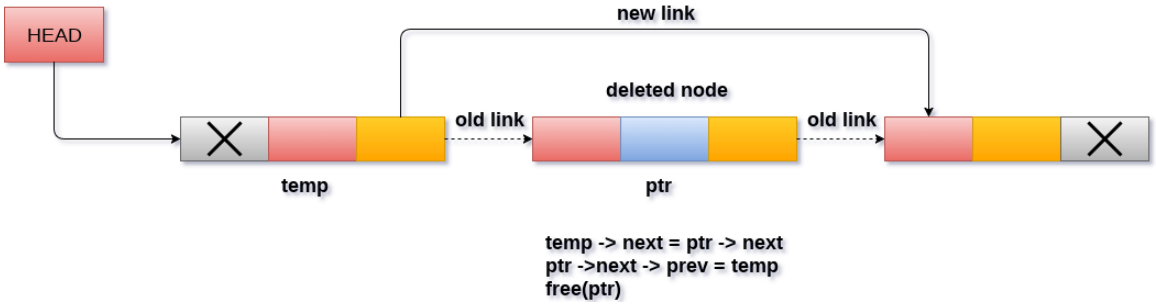
5. Removing the node from end of the list :

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 7 [END OF IF]
Step 2: SET TEMP = HEAD
Step 3: REPEAT STEP 4 WHILE TEMP->NEXT != NULL
Step 4: SET TEMP = TEMP->NEXT [END OF LOOP]
Step 5: SET TEMP ->PREV-> NEXT = NULL
Step 6: FREE TEMP
Step 7: EXIT



6. Deletion of the node having given data

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 9 [END OF IF]
Step 2: SET TEMP = HEAD
Step 3: Repeat Step 4 while TEMP -> DATA != ITEM
Step 4: SET TEMP = TEMP -> NEXT [END OF LOOP]
Step 5: SET PTR = TEMP -> NEXT
Step 6: SET TEMP -> NEXT = PTR -> NEXT
Step 7: SET PTR -> NEXT -> PREV = TEMP
Step 8: FREE PTR
Step 9: EXIT



7. Display All the nodes :

Step 1: IF HEAD == NULL

WRITE "UNDERFLOW"
GOTO STEP 6
[END OF IF]Step 2: Set PTR = HEAD
Step 3: Repeat step 4 and 5 while PTR != NULL
Step 4: Write PTR → data
Step 5: PTR = PTR → next
Step 6: Exit

Example :

```
#include<stdio.h>
#include<conio.h>
#include<stdlib>
struct list
{
        int info;
        struct list *next,*prev;
};
typedef struct list node;
node *p;
void create(int,node *);
void display(node *);
void addbeg(int,node *);
void addafter(int,int,node *);
void deleted(int,node *);
void main()
{
        p=NULL;
        clrscr();
        create(10,p);
        create(20,p);
        create(30,p);
        display(p);
        printf("\n\n");
        addbeg(0,p);
        printf("After Adding element at beginning :");
        display(p);
        printf("\n\n");
        addafter(2,500,p);
        printf("After adding element at specific position :");
        display(p);
        printf("\n\n");
```

```
	deleted(20,p);
	printf("After deletion elements are :");
	display(p);
	getch();
}
void create(int ele,node *q)
{
	node *temp;
	if(q==NULL)
	{
		p=(node *)malloc(sizeof(node));
		p->prev=NULL;
		p->info=ele;
		p->next = NULL;
	}
	else
	{
		while(q->next!=NULL)
		{
			q=q->next;
		}
		temp=(node *)malloc(sizeof(node));
		temp->next=NULL;
		temp->info=ele;
		temp->prev=q;
		q->next=temp;
	}
}
void display(node *q)
{
	while(q!=NULL)
	{
		printf("\t%d",q->info);
		q=q->next;
	}
}
void addbeg(int ele,node *q)
{
	p=(node *)malloc(sizeof(node));
	p->prev=NULL;
```

```
        p->info=ele;
        p->next=q;
        q->prev=p;
}
void addafter(int c,int ele,node *q)
{
        node *temp;
        int i;
        for(i=1;i<c;i++)
        {
                q=q->next;
                if(q==NULL)
                {
                        printf("\nposition is out of range");
                        return;
                }
        }
        temp=(node *)malloc(sizeof(node));
        temp->prev=q;
        temp->next=q->next;
        temp->info=ele;
        temp->next->prev=temp;
        q->next=temp;
        return;
}
void deleted(int ele,node *q)
{
        node *temp;
        if(q->info==ele)
        {
                p=q->next;
                q->next->prev=NULL;
                free(q);
                return;
        }
        while(q->next->next!=NULL)
        {
                if(q->next->info==ele)
                {
                        temp=q->next;
```

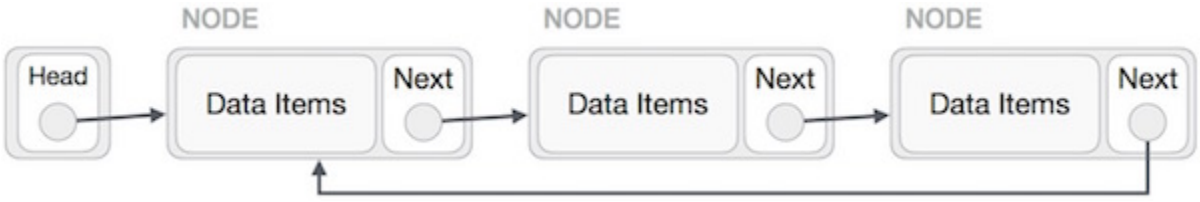
```
                    q->next=q->next->next;
                    q->next->prev=temp->prev;
                    free(temp);
                    return;
                }
                q=q->next;
        }
}
```

Circular Linked list :

- A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.
- We traverse a circular singly linked list until we reach the same node where we started.
- The circular singly linked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.
- Circular linked lists are mostly used in task maintenance in operating systems.
- There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.
- There are basically two types of circular linked list:

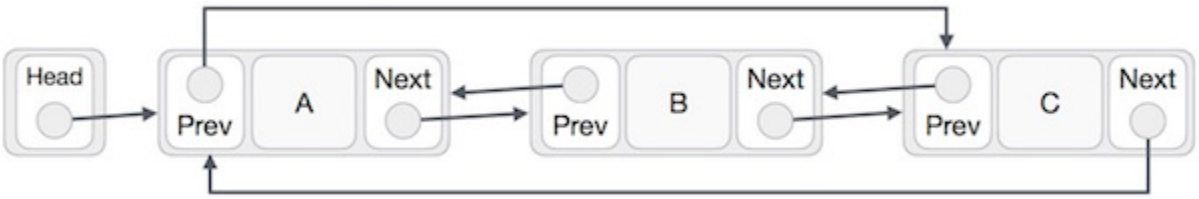
  Circular Singly Linked List

  Here, the address of the last node consists of the address of the first node.



  Circular Doubly Linked List

  Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.

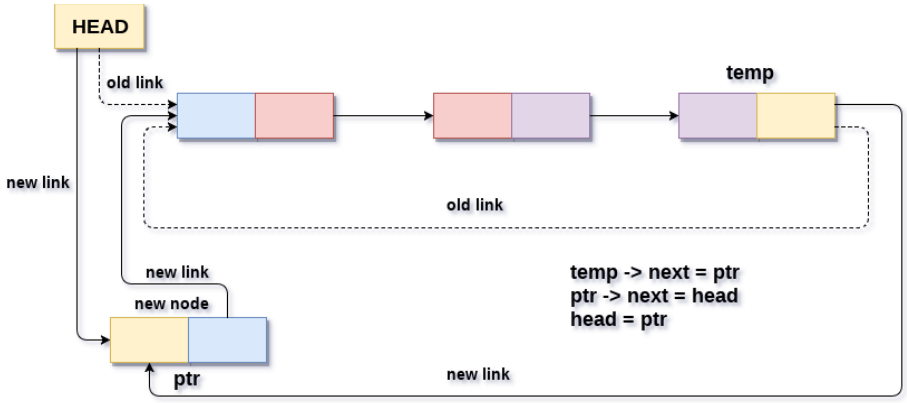


Operations on Circular Singly linked list:

Insertion at beginning : Adding a node into circular singly linked list at the beginning
Insertion at the end : Adding a node into circular singly linked list at the end.
Deletion at beginning : Removing the node from circular singly linked list at the beginning.
Deletion at the end : Removing the node from circular singly linked list at the end.

1. **Insertion into circular singly linked list at beginning**

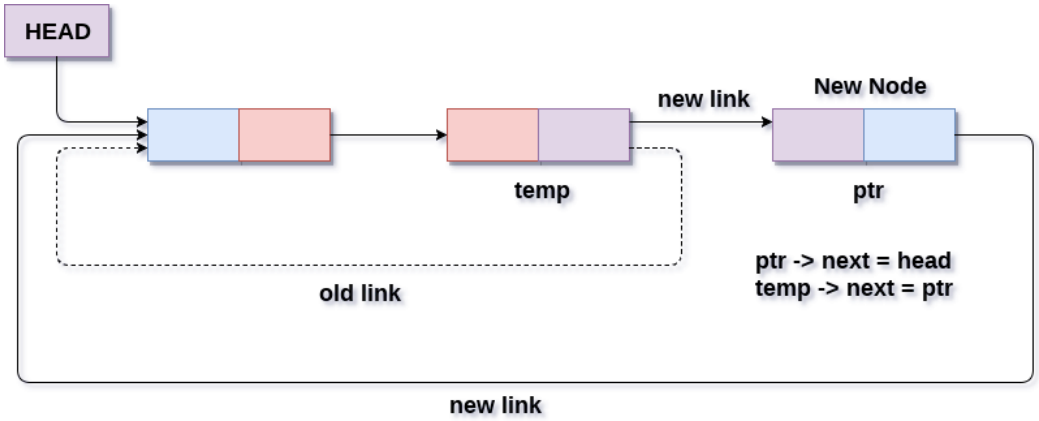
Step 1: IF PTR = NULL
  Write OVERFLOW
 Go to Step 11 [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET TEMP = HEAD
Step 6: Repeat Step 8 while TEMP -> NEXT != HEAD
Step 7: SET TEMP = TEMP -> NEXT [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = HEAD
Step 9: SET TEMP → NEXT = NEW_NODE
Step 10: SET HEAD = NEW_NODE
Step 11: EXIT



2. **Insertion into circular singly linked list at the end**

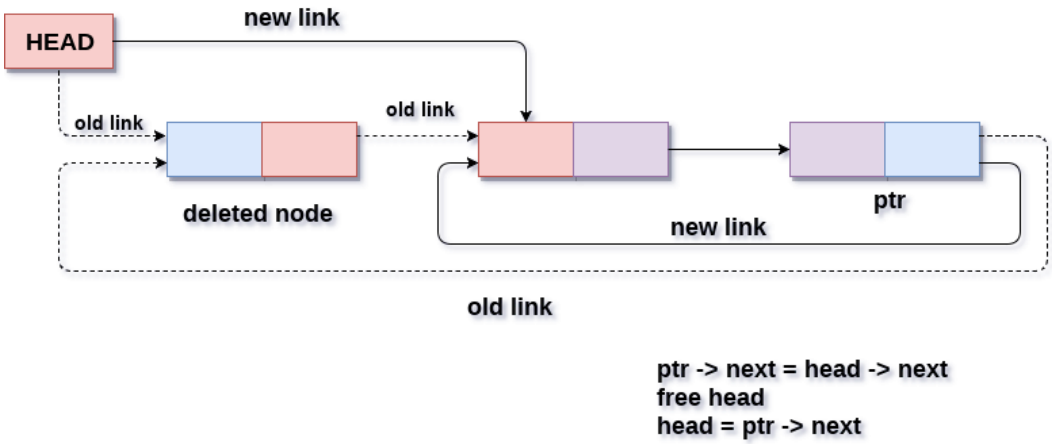
Step 1: IF PTR = NULL
  Write OVERFLOW
   Go to Step 1 [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = HEAD
Step 6: SET TEMP = HEAD
Step 7: Repeat Step 8 while TEMP -> NEXT != HEAD

Step 8: SET TEMP = TEMP -> NEXT [END OF LOOP]
Step 9: SET TEMP -> NEXT = NEW_NODE
Step 10: EXIT



3. Deletion in circular singly linked list at beginning

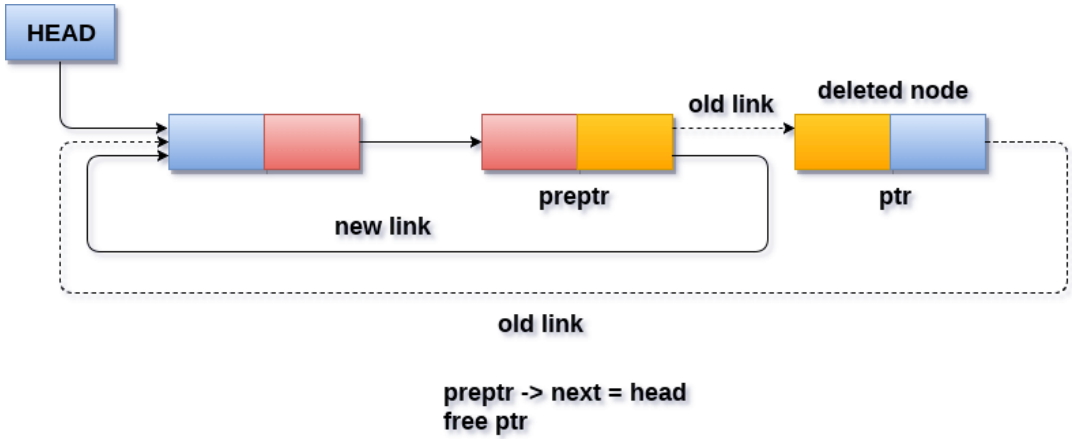
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8 [END OF IF]
Step 2: SET PTR = HEAD
Step 3: Repeat Step 4 while PTR → NEXT != HEAD
Step 4: SET PTR = PTR → next [END OF LOOP]
Step 5: SET PTR → NEXT = HEAD → NEXT
Step 6: FREE HEAD
Step 7: SET HEAD = PTR → NEXT
Step 8: EXIT



4. Deletion in Circular singly linked list at the end

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]Step 2: SET PTR = HEAD

Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != HEAD
Step 4: SET PREPTR = PTR
Step 5: SET PTR = PTR -> NEXT [END OF LOOP]
Step 6: SET PREPTR -> NEXT = HEAD
Step 7: FREE PTR
Step 8: EXIT



5. <u>Display all node in Circular Linked list :</u>

STEP 1: SET PTR = HEAD
STEP 2: IF PTR = NULL
WRITE "EMPTY LIST"
GOTO STEP 8
END OF IFSTEP 4: REPEAT STEP 5 AND 6 UNTIL PTR → NEXT != HEAD
STEP 5: PRINT PTR → DATA
STEP 6: PTR = PTR → NEXT [END OF LOOP]
STEP 7: PRINT PTR→ DATA
STEP 8: EXIT

Program for Circular Linked List in C:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib>
struct list
{
        int info;
        struct list *next;
};
typedef struct list node;
node *p;
void create(int,node *);
void display(node *);
```

```
void addbeg(int,node *);
void addafter(int,int,node *);
void deleted(int,node *);
void main()
{
	p=NULL;
	clrscr();
	create(10,p);
	create(11,p);
	create(12,p);
	display(p);
	printf("\n\n");
	addbeg(0,p);
	addafter(3,12,p);
	deleted(11,p);
	getch();
}
void create(int ele,node *q)
{
	if(q==NULL)
	{
		p=(node *)malloc(sizeof(node));
		p->info=ele;
		p->next=p;
	}
	else
	{
		while(q->next!=p)
		{
			q=q->next;
		}
		q->next=(node *)malloc(sizeof(node));
		q->next->info=ele;
		q->next->next=p;
	}
}
void display(node *q)
{
	do
	{
```

```
                printf("\nElement is %d",q->info);
                q=q->next;
                }while(q!=p);
        }
void addbeg(int ele,node *q)
{
        node *temp;
        temp=(node *)malloc(sizeof(node));
        temp->info=ele;
        temp->next=q;
        while(q->next!=p)
        {
                q=q->next;
        }
        q->next=temp;
        p=temp;
}
void addafter(int c,int ele,node *q)//pos is same as c var.
{
        node *temp;
        int i;
        for(i=1;i<c;i++)
        {
                q=q->next;
                if(q==p)
                {
                        printf("\nposition is out of range");
                        return;
                }
        }
        temp=(node *)malloc(sizeof(node));
        temp->info=ele;
        temp->next=q->next;
        q->next=temp;
}
void deleted(int ele,node *q)
{
        node *temp;
        if(q->info==ele)
        {
```

```
        do
                {
                        q=q->next;
                }while(q->next!=p);
                q->next=p;
                p=p->next;
                return;
        }
        while(q->next->next!=p)
        {
                if(q->next->info==ele)
                {
                        temp=q->next;
                        q->next=temp->next;
                        free(temp);
                        return;
                }
                q=q->next;
        }
}
```

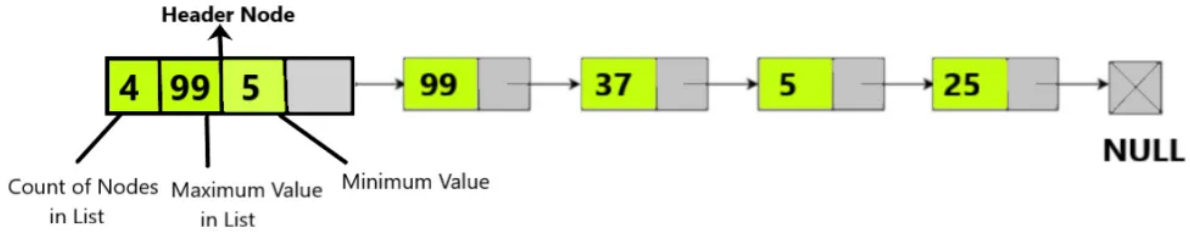
Header Linked list :

- Header Linked List is a modified version of Singly Linked List.
- In the Header linked list, we have a special node, the **Header Node** present at the beginning of the linked list.
- The Header Node is an extra node at the front of the list storing meaningful information about the list.
- Such a node is not similar in structure to the other nodes in the list. It does not represent any items of the list like other nodes, rather the information present in the Header node is global for all nodes such as **Count of Nodes in a List, Maximum among all Items, Minimum value among all Items etc.**

Types of Header Linked List :

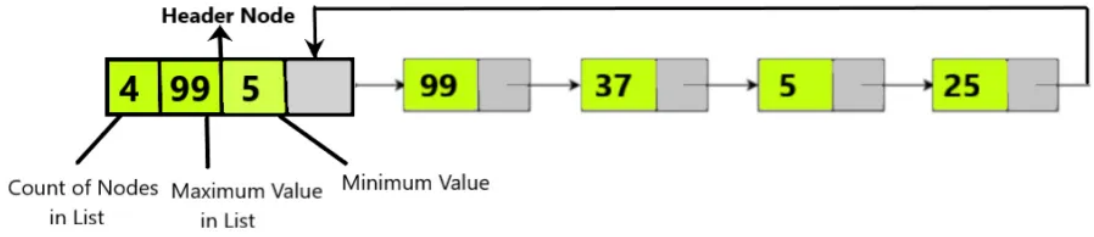
1. **Grounded Header Linked List**

- In this type of Header Linked List, the last node of the list points to NULL or holds the reference to NULL Pointer.
- The head pointer points to the Header node of the list. If there is no node to the next of head pointer or head.next equals NULL then we know that the Linked List is empty.
- The operations performed on the Header Linked List are the same as Singly Linked List such as Insertion, Deletion, and Traversal of nodes. Let us understand this with an example:



2. Circular Header Linked List :

- A Linked List whose last node points back to the First node or the Head Node of the list is called a Circular Linked List.
- Similarly, if the last node of the Header Linked List points back to Header Node then it is a Circular Header Linked List.
- The last node of the list does not hold NULL reference.
- In this case, We have to use external pointers to maintain the last node.The end of the list is not known while traversing.
- It can perform the same operations as its counterpart such as Insert, Delete and Search. Let us see an example.



Applications of linked list are:

- Implementation of stacks and queues
- Implementation of graphs: Adjacency list representation of graphs is the most popular which uses a linked list to store adjacent vertices.
- Dynamic memory allocation: We use a linked list of free blocks.
- Maintaining a directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of the linked list
- representing sparse matrices
- In Doubly linked list :
  - Redo and undo functionality.
  - Use of the Back and forward button in a browser.
  - The most recently used section is represented by the Doubly Linked list.
  - Other Data structures like Stack, HashTable, and BinaryTree can also be applied by Doubly Linked List.

Applications of linked list in the real world:

1. **Image viewer** – Previous and next images are linked and can be accessed by the next and previous buttons.
2. **Previous and next page in a web browser** – We can access the previous and next URL searched in a web browser by pressing the back and next buttons since they are linked as a linked list.
3. **Music Player** – Songs in the music player are linked to the previous and next songs. So you can play songs either from the start or ending of the list.

Extra Study Material for Unit 4:

Singly linked list menu driven program :

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
   int choice =0;
   while(choice != 9)
   {
      printf("\n\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n===============================================\n");
      printf("\n1.Insert in begining \n2.Insert at last \n3.Insert at any random location
\n4.Delete from Beginning \n5.Delete from last \n6.Delete node after specified location
\n7.Search for an element \n8.Show \n9.Exit \n");
      printf("\nEnter your choice?\n");
```

```
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
            beginsert();
            break;
            case 2:
            lastinsert();
            break;
            case 3:
            randominsert();
            break;
            case 4:
            begin_delete();
            break;
            case 5:
            last_delete();
            break;
            case 6:
            random_delete();
            break;
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void beginsert()
{
    struct node *ptr;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node *));
```

```
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value\n");
        scanf("%d",&item);
        ptr->data = item;
        ptr->next = head;
        head = ptr;
        printf("\nNode inserted");
    }

}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            ptr -> next = NULL;
            head = ptr;
            printf("\nNode inserted");
        }
        else
        {
            temp = head;
            while (temp -> next != NULL)
            {
```

```
            temp = temp -> next;
        }
        temp->next = ptr;
        ptr->next = NULL;
        printf("\nNode inserted");

    }
  }
}
void randominsert()
{
  int i,loc,item;
  struct node *ptr, *temp;
  ptr = (struct node *) malloc (sizeof(struct node));
  if(ptr == NULL)
  {
    printf("\nOVERFLOW");
  }
  else
  {
    printf("\nEnter element value");
    scanf("%d",&item);
    ptr->data = item;
    printf("\nEnter the location after which you want to insert ");
    scanf("\n%d",&loc);
    temp=head;
    for(i=0;i<loc;i++)
    {
      temp = temp->next;
      if(temp == NULL)
      {
        printf("\ncan't insert\n");
        return;
      }

    }
    ptr ->next = temp ->next;
    temp ->next = ptr;
    printf("\nNode inserted");
  }
```

```
}
void begin_delete()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\nList is empty\n");
   }
   else
   {
      ptr = head;
      head = ptr->next;
      free(ptr);
      printf("\nNode deleted from the begining ...\n");
   }
}
void last_delete()
{
   struct node *ptr,*ptr1;
   if(head == NULL)
   {
      printf("\nlist is empty");
   }
   else if(head -> next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nOnly node of the list deleted ...\n");
   }

   else
   {
      ptr = head;
      while(ptr->next != NULL)
      {
         ptr1 = ptr;
         ptr = ptr ->next;
      }
      ptr1->next = NULL;
      free(ptr);
```

```
        printf("\nDeleted Node from the last ...\n");
    }
}
void random_delete()
{
    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after which you want to perform deletion \n");
    scanf("%d",&loc);
    ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nCan't delete");
            return;
        }
    }
    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted node %d ",loc+1);
}
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
```

```
            if(ptr->data == item)
            {
               printf("item found at location %d ",i+1);
               flag=0;
            }
            else
            {
               flag=1;
            }
            i++;
            ptr = ptr -> next;
         }
         if(flag==1)
         {
            printf("Item not found\n");
         }
      }

}

void display()
{
   struct node *ptr;
   ptr = head;
   if(ptr == NULL)
   {
      printf("Nothing to print");
   }
   else
   {
      printf("\nprinting values . . . . .\n");
      while (ptr!=NULL)
      {
         printf("\n%d",ptr->data);
         ptr = ptr -> next;
      }
   }
}
```

Menu Driven Program in C to implement all the operations of doubly linked list

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

```
#include<stdlib.h>
struct node
{
    struct node *prev;
    struct node *next;
    int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void search();
void main ()
{
int choice =0;
    while(choice != 9)
    {
        printf("\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================================\n");
        printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random
location\n4.Delete from Beginning\n5.Delete from last\n6.Delete the node after the
given data\n7.Search\n8.Show\n9.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
            insertion_beginning();
            break;
            case 2:
                  insertion_last();
            break;
            case 3:
            insertion_specified();
            break;
```

```
            case 4:
            deletion_beginning();
            break;
            case 5:
            deletion_last();
            break;
            case 6:
            deletion_specified();
            break;
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void insertion_beginning()
{
   struct node *ptr;
   int item;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
    printf("\nEnter Item value");
    scanf("%d",&item);

   if(head==NULL)
   {
      ptr->next = NULL;
```

```
        ptr->prev=NULL;
        ptr->data=item;
        head=ptr;
    }
    else
    {
        ptr->data=item;
        ptr->prev=NULL;
        ptr->next = head;
        head->prev=ptr;
        head=ptr;
    }
    printf("\nNode inserted\n");
}

}
void insertion_last()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value");
        scanf("%d",&item);
         ptr->data=item;
        if(head == NULL)
        {
            ptr->next = NULL;
            ptr->prev = NULL;
            head = ptr;
        }
        else
        {
          temp = head;
          while(temp->next!=NULL)
```

```
        {
            temp = temp->next;
        }
        temp->next = ptr;
        ptr ->prev=temp;
        ptr->next = NULL;
        }

      }
     printf("\nnode inserted\n");
    }
void insertion_specified()
{
   struct node *ptr,*temp;
   int item,loc,i;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\n OVERFLOW");
   }
   else
   {
      temp=head;
      printf("Enter the location");
      scanf("%d",&loc);
      for(i=0;i<loc;i++)
      {
         temp = temp->next;
         if(temp == NULL)
         {
            printf("\n There are less than %d elements", loc);
            return;
         }
      }
      printf("Enter value");
      scanf("%d",&item);
      ptr->data = item;
      ptr->next = temp->next;
      ptr -> prev = temp;
      temp->next = ptr;
```

```
      temp->next->prev=ptr;
      printf("\nnode inserted\n");
   }
}
void deletion_beginning()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nnode deleted\n");
   }
   else
   {
      ptr = head;
      head = head -> next;
      head -> prev = NULL;
      free(ptr);
      printf("\nnode deleted\n");
   }

}
void deletion_last()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nnode deleted\n");
   }
```

```
    else
    {
        ptr = head;
        if(ptr->next != NULL)
        {
            ptr = ptr -> next;
        }
        ptr -> prev -> next = NULL;
        free(ptr);
        printf("\nnode deleted\n");
    }
}
void deletion_specified()
{
    struct node *ptr, *temp;
    int val;
    printf("\n Enter the data after which the node is to be deleted : ");
    scanf("%d", &val);
    ptr = head;
    while(ptr -> data != val)
    ptr = ptr -> next;
    if(ptr -> next == NULL)
    {
        printf("\nCan't delete\n");
    }
    else if(ptr -> next -> next == NULL)
    {
        ptr ->next = NULL;
    }
    else
    {
        temp = ptr -> next;
        ptr -> next = temp -> next;
        temp -> next -> prev = ptr;
        free(temp);
        printf("\nnode deleted\n");
    }
}
void display()
{
```

```
    struct node *ptr;
    printf("\n printing values...\n");
    ptr = head;
    while(ptr != NULL)
    {
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
}
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("\nitem found at location %d ",i+1);
                flag=0;
                break;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("\nItem not found\n");
```

```
        }
    }

}
```

Circular Doubly Linked List:

```
#include<stdio.h>
#include<conio.h>
struct list
{
        int info;
        struct list *next,*prev;
};
typedef struct list node;
node *p;
void create(int,node *);
void display(node *);
void addbeg(int,node *);
void addafter(int,int,node *);
void deleted(int,node *);
void main()
{
        p=NULL;
        clrscr();
        create(10,p);
        create(5,p);
        create(15,p);
        addbeg(0,p);
        deleted(15,p);
        display(p);
        getch();
}
void create(int ele,node *q)
{
        node *temp;
        if(q==NULL)
        {
                p=(node *)malloc(sizeof(node));
                p->prev=p;
                p->info=ele;
                p->next=p;
```

```
        }
        else
        {
                while(q->next!=p)
                {
                        q=q->next;
                }
                temp=(node *)malloc(sizeof(node));
                temp->next=p;
                temp->info=ele;
                temp->prev=q;
                q->next=temp;
        }
}
void display(node *q)
{
        do
        {
                printf("\nElement is %d",q->info);
                q=q->next;
        }while(q!=p);
}
void addbeg(int ele,node *q)
{
        node *temp;
        while(q->next!=p)
        {
                q=q->next;
        }
        temp=(node *)malloc(sizeof(node));
        temp->prev=q;
        temp->info=ele;
        temp->next=p;
        p->prev=temp;
        q->next=temp;
        p=temp;
}
void addafter(int c,int ele,node *q)
{
        node *temp;
```

```
        int i;
        for(i=1;i<c;i++)
        {
                q=q->next;
                if(q==p)
                {
                        printf("\nposition is out of range");
                        return;
                }
        }
        temp=(node *)malloc(sizeof(node));
        temp->prev=q;
        temp->next=q->next;
        temp->info=ele;
        temp->next->prev=temp;
        q->next=temp;
        return;
}
void deleted(int ele,node *q)
{
        node *temp;
        if(q->info==ele)
        {
                temp=q;
                p=temp->next;
                while(q->next!=p)
                {
                        q=q->next;
                }
                p->prev=q;
                return;
        }
        do
        {
                if(q->next->info==ele)
                {
                        temp=q->next;
                        q->next=temp->next;
                        temp->next->prev=q;
                        // free(temp);
```

```
                return;
            }
            q=q->next;
        }while(q!=p);
}
```

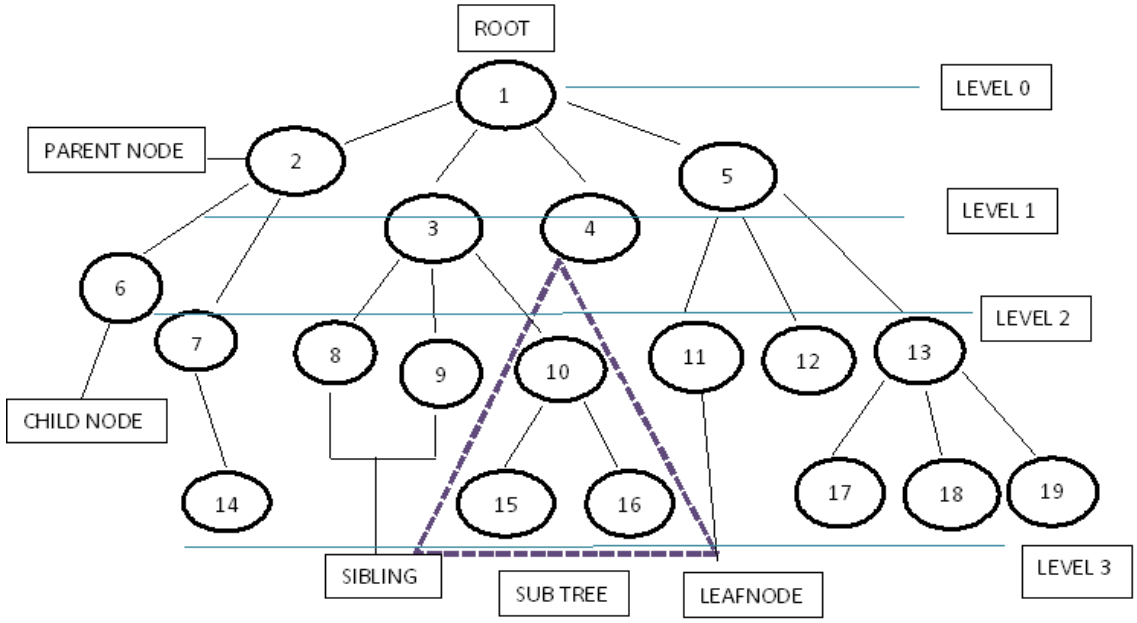
Header Linked List Menu Driven code :

```
#include <stdio.h>
#include <malloc.h>
struct node
{
        int data;
        struct node *next;
};
struct node *start = NULL;
struct node *create_hll(struct node *);
struct node *display(struct node *);
int main()
{
        int option;
        do
        {
            printf("\n\n -----Main Menu-----");
            printf("\n 1. Create a list");
            printf("\n 2. Display the list");
            printf("\n 3. Exit");
            printf("\n Enter your choice : ");
            scanf("%d", &option);
            switch(option)
            {
                case 1:
                start = create_hll(start);
                printf("\n Header Linked List Created Successfully");
                break;
                case 2:
                start = display(start);
                break;
            }
        }while(option != 3);
        return 0;
}
```

```
struct node *create_hll(struct node *start)
{
	struct node *new_node, *ptr;
	int num;
	printf("\n Enter -1 to end");
	printf("\n Enter the data :");
	scanf("%d", &num);
	while(num != -1)
	{
		new_node = (struct node*)malloc(sizeof(struct node));
		new_node -> data = num;
		new_node -> next = NULL;
		if(start == NULL)
		{
			start = (struct node*)malloc(sizeof(struct node));
			start -> next = new_node;
		}
		else
		{
			ptr = start;
			while(ptr -> next != NULL)
			ptr = ptr -> next;
			ptr -> next = new_node;
		}
		printf("\n Enter the data :");
		scanf("%d", &num);
	}
	return start;
}
struct node *display(struct node *start)
{
	struct node *ptr;
	ptr = start;
	ptr = ptr -> next;
	while(ptr != NULL)
	{
		printf("\t %d",ptr -> data);
		ptr = ptr -> next;
	}
	return start;   }
```

Unit 5 : Tree Data Structure

- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.
- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially.
- As the size of data increases, the time to access each data element increases linearly.
- The tree data structure allows quicker and easier access to the data as it is a non-linear data structure.



Basic Terminology In Tree Data Structure:

Path − Path refers to the sequence of nodes along the edges of a tree.

Parent Node: The node which is a predecessor of a node is called the parent node of that node. {2} is the parent node of {6, 7}.

Child Node: The node which is the immediate successor of a node is called the child node of that node. Examples: {6, 7} are the child nodes of {2}.

Root Node: The topmost node of a tree or the node which does not have any parent node is called the root node. {1} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.

Leaf Node or External Node: The nodes which do not have any child nodes are called leaf nodes. {6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19} are the leaf nodes of the tree.

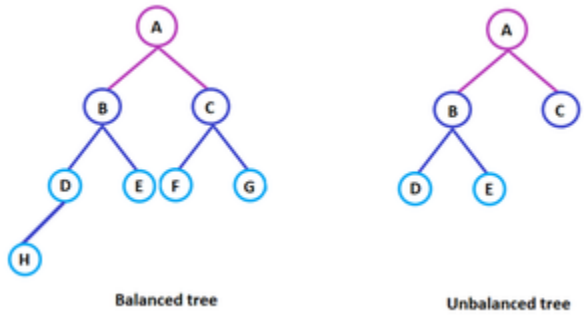
Ancestor of a Node: Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {1, 2} are the parent nodes of the node {7}

Descendant: Any successor node on the path from the leaf node to that node. {7, 14} are the descendants of the node. {2}.

Sibling: Children of the same parent node are called siblings. {8, 9, 10} are called siblings.
Depth of a node: The count of edges from the root to the node. Depth of node {14} is 3.
Height of a node: The number of edges on the longest path from that node to a leaf. Height of node {3} is 2.
Height of a tree: The height of a tree is the height of the root node i.e the count of edges from the root to the deepest node. The height of the above tree is 3.
Level of a node: The count of edges on the path from the root node to that node. The root node has level 0.
Internal node: A node with at least one child is called Internal Node.
Neighbor of a Node: Parent or child nodes of that node are called neighbors of that node.
Subtree: Subtree represents the descendants of a node.
Visiting node − Visiting refers to checking the value of a node when control is on the node.
keys − Key represents a value of a node based on which a search operation is to be carried out for a node.
Traversing − Traversing means passing through nodes in a specific order.
Levels − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.

Types of Tree data structures :

The different types of tree data structures are as follows:
General tree : A general tree data structure has no restriction on the number of nodes. It means that a parent node can have any number of child nodes.
Binary tree : A node of a binary tree can have a maximum number of two child nodes. In the given tree diagram, nodes B, D, and F are left children, while E, C, and G are the right children.
Balanced tree : If the height of the left subtree and the right subtree are equal or differ at most by 1, the tree is known as a balanced tree data structure.



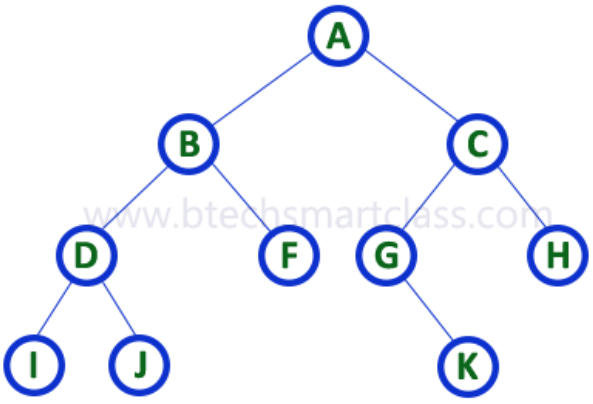
Binary search tree :
As the name implies, binary search trees are used for various searching and sorting algorithms. The examples include AVL tree and red-black tree. It is a non-linear data structure. It shows that the value of the left node is less than its parent, while the value of the right node is greater than its parent.

AVL Tree :
On behalf of the inventors Adelson-Velshi and Landis, the name AVL is given. An AVL tree is a self-balancing binary search tree. This is the first tree introduced which automatically balances its height.
Red-black tree : A red-black tree is a self-balancing binary search tree, where each node has a color; red or black. The colors of the nodes are used to make sure that the tree remains approximately balanced during insertions and deletions.
Splay tree : A splay tree is a self-balancing binary search tree.
Treap : A treap (the name derived from tree + heap) is a binary search tree.Each node has two values; a key and a priority. The keys follow the binary-search-tree property.
B-tree : B tree is a self-balancing search tree and contains multiple nodes which keep data in sorted order. Each node has 2 or more children and consists of multiple keys.

Binary tree :

- A binary tree is a tree-type non-linear data structure with a maximum of two children for each parent.
- A binary tree is an important class of a tree data structure in which a node can have at most two children. Every node in a binary tree has a left and right reference along with the data element.
- Child node in a binary tree on the left is termed as the 'left child node' and the node on the right is termed as the 'right child node.'
- The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes.
- A parent node has two child nodes: the left child and right child. Hashing, routing data for network traffic, data compression, preparing binary heaps, and binary search trees are some of the applications that use a binary tree.
- A binary tree is either an empty tree Or a binary tree consists of a node called the root node, a left subtree and a right subtree, both of which will act as a binary tree once again

A binary tree data structure is represented using two methods. Those methods are as follows…
1) Sequential representation (Linear - Array representation)
2) Linked List representation (Non-linear)



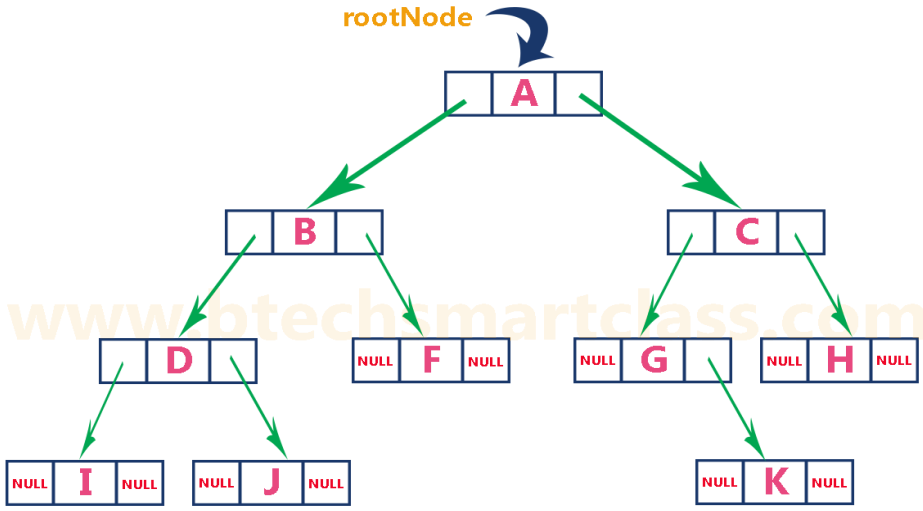
1. Array Representation of Binary Tree :
In array representation of a binary tree, we use a one-dimensional array (1-D Array) to represent a binary tree.
Consider the above example of a binary tree and it is represented as follows...

2. Linked List Representation of Binary Tree :

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...

The above example of the binary tree represented using Linked list representation is shown as follows...



Example :

```
#include <stdio.h>
#include <stdlib.h>
struct node {
        int data;
        struct node* left;
        struct node* right;
};
/* newNode() allocates a new node with the given data and NULL left and right pointers.
*/
struct node* newNode(int data)
{
        // Allocate memory for new node
        struct node* node =(struct node*)malloc(sizeof(struct node));

        // Assign data to this node
```

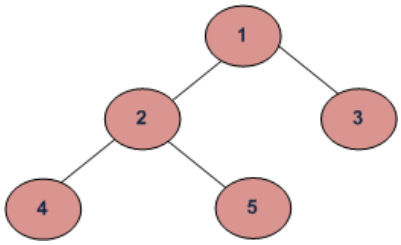
```
        node->data = data;

        // Initialize left and right children as NULL
        node->left = NULL;
        node->right = NULL;
        return (node);
}
void main()
{
        /*create root*/
        struct node* root = newNode(1);
        /* following is the tree after above statement
              1
            /   \
        NULL    NULL
        */
        root->left = newNode(2);
        root->right = newNode(3);
        /* 2 and 3 become left and right children of 1
                    1
                  /   \
                 2     3
               /  \    /  \
          NULL NULL NULL NULL
        */

        root->left->left = newNode(4);
        /* 4 becomes left child of 2
                      1
                    /   \
                   2     3
                 /  \    /  \
               4   NULL NULL NULL
             /  \
        NULL NULL
        */
        getchar();
}
```

What is tree traversal? Explain tree traversal techniques with examples.

- Tree traversal is the method or technique to display data in a binary tree.

- Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.
- In the traversing method, the tree is processed in such a way that each node is visited only once.

Following are the generally used ways for traversing trees.



1) Depth First Traversals:
   - o Inorder (Left, Root, Right) : 4 2 5 1 3
   - o Preorder (Root, Left, Right) : 1 2 4 5 3
   - o Postorder (Left, Right, Root) : 4 5 2 3 1
2) Breadth First or Level Order Traversal : 1 2 3 4 5

Inorder Traversal(left subtree, root, right subtree):

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Preorder Traversal (root, left subtree, right subtree):

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal (left subtree, right subtree, root):

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

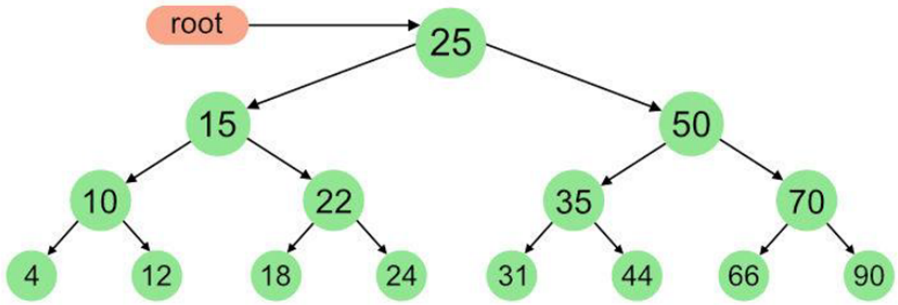
Example: Postorder traversal for the above given figure is 4 5 2 3 1.

One more example :

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Binary Search Tree

- Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called an **ordered binary tree.**
- In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
- Similarly, the value of all the nodes in the right subtree is greater than or equal to the value of the root.
- This rule will be recursively applied to all the left and right subtrees of the root.

Advantages of using binary search tree

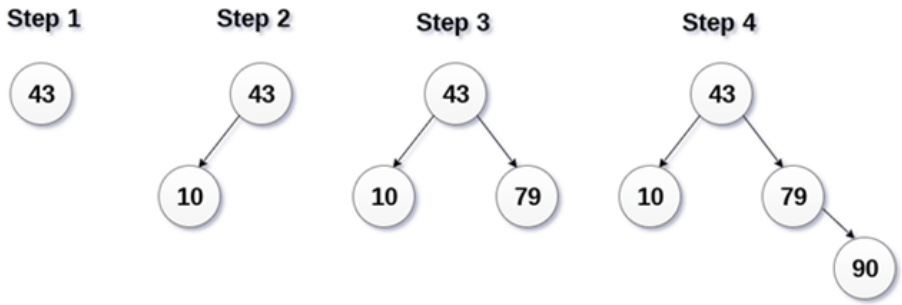
1. Searching becomes very efficient in a binary search tree since we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as an efficient data structure in comparison to arrays and linked lists. In the searching process, it removes half a sub-tree at every step.
3. It also speeds up the insertion and deletion operations as compared to that in array and linked list.

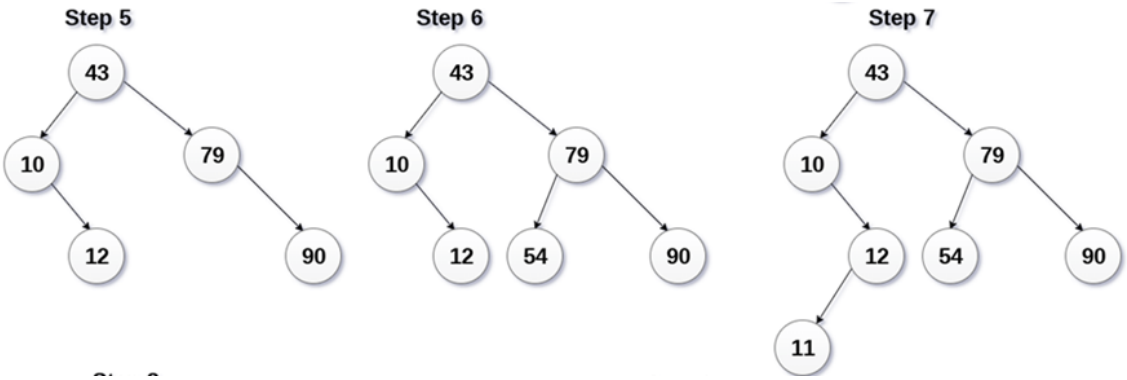
Create the binary search tree using the following data elements.

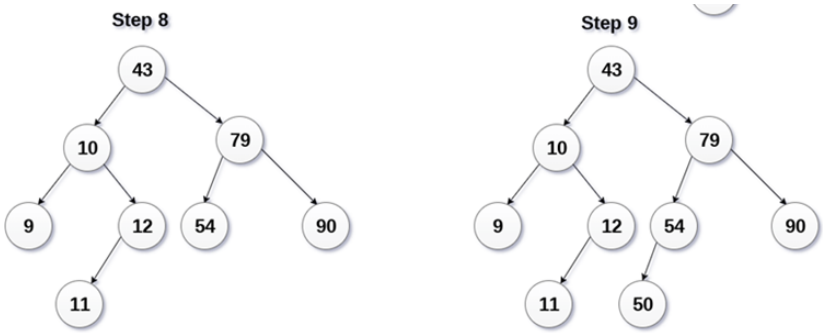
43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is less than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right subtree.

The process of creating BST by using the given elements, is shown in the image below.





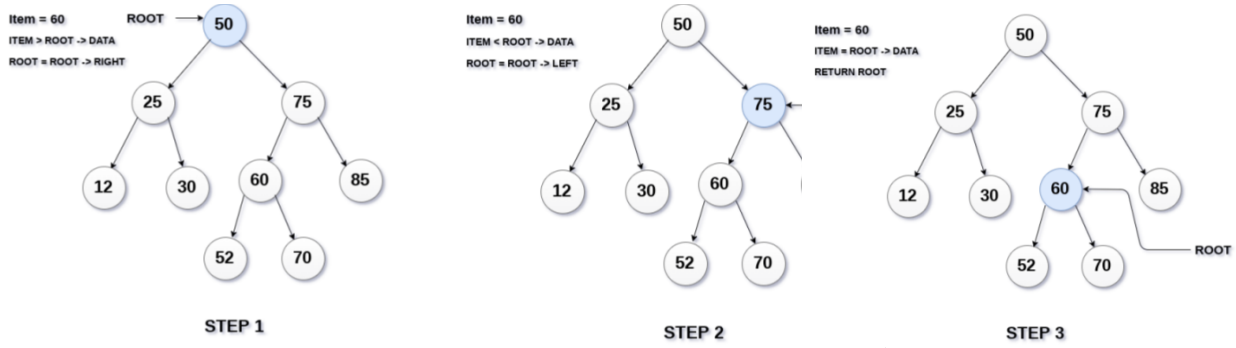


Operations on Binary Search Tree

1. **Searching :** Searching means finding or locating some specific element or node within a data structure.

However, searching for some specific node in a binary search tree is pretty easy due to the fact that elements in BST are stored in a particular order.

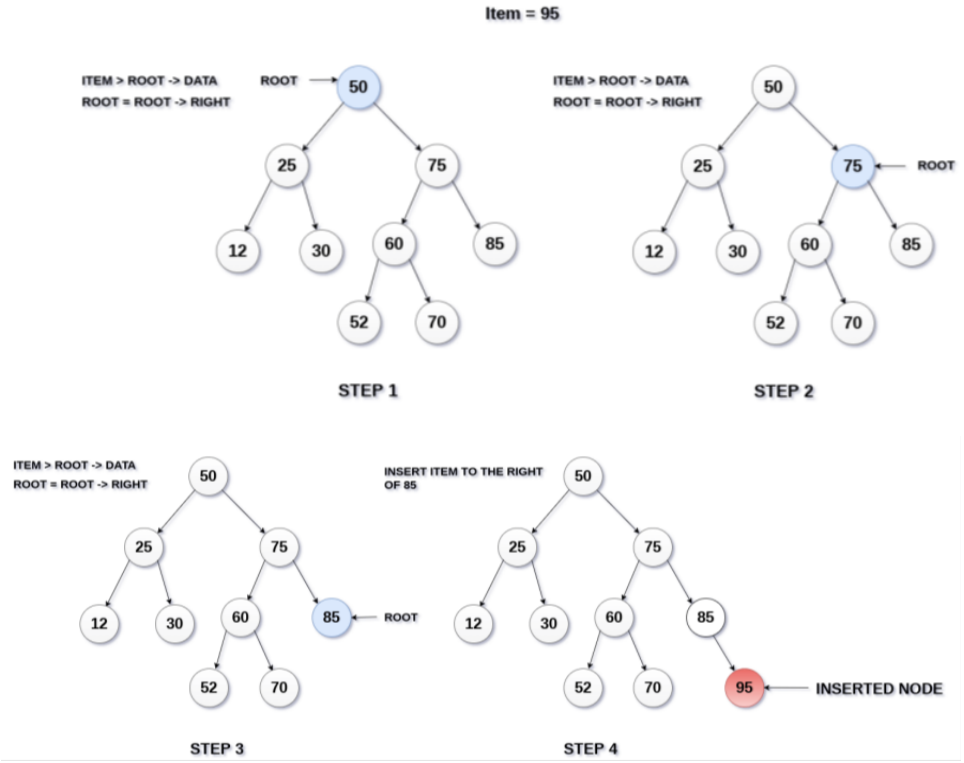
1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if the item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right subtree.
5. Repeat this procedure recursively until a match is found.
6. If an element is not found then return NULL.



2. Insertion: Insert function is used to add a new element in a binary search tree at appropriate location.

Insert function is to be designed in such a way that it must violate the property of binary search tree at each value.

1. Allocate the memory for trees.
2. Set the data part to the value and set the left and right pointer of the tree, pointing to NULL.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
5. If this is false, then perform this operation recursively with the right subtree of the root.

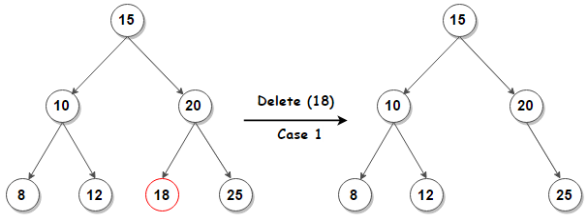


3. Deletion : Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way that the property of the binary search tree doesn't violate. There are three situations of deleting a node from a binary search tree. The node to be deleted is a leaf node

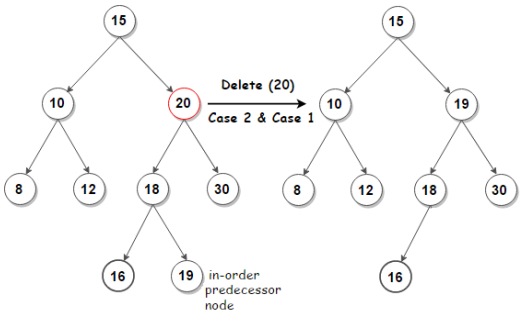
It is the simplest case, in this case, to replace the leaf node with the NULL and simply free the allocated space.

In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.

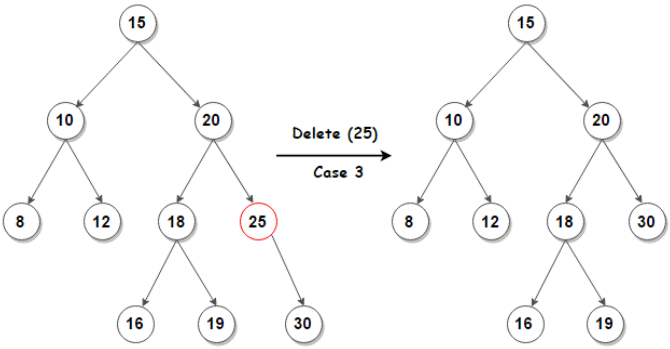
Case 1: Deleting a node with no children: remove the child node from the tree.
It is the simplest case, in this case, to replace the leaf node with the NULL and simply free the allocated space.



Case 2: Deleting a node with two children: call the node to be deleted N. Do not delete N. Instead, choose either its inorder successor node or its inorder predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.



Case 3 : Deleting a node with one child: remove the node and replace it with its child.

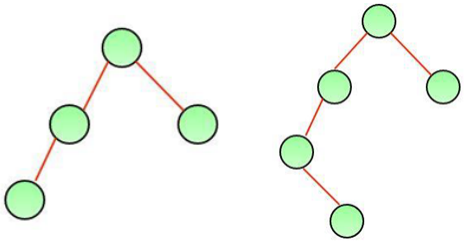


Height Balanced Tree :

- AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree.
- AVL Tree was invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honor of its inventors.
- AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.
- Tree is said to be balanced if the balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.
- If the balance factor of any node is 1, it means that the left sub-tree is one level higher than the right subtree.If balance factor of any node is 0, it means that the left subtree and right subtree contain equal height.
- If the balance factor of any node is -1, it means that the left sub-tree is one level lower than the right subtree.
- In an AVL tree, every node maintains an extra information known as **balance factor.**

- A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.
- An empty tree is height-balanced.
- Non-empty binary tree T is balanced if: 1) Left subtree of T is balanced 2) Right subtree of T is balanced and 3) The difference between heights of left subtree and right subtree is not more than 1.
- The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we calculate as follows…

  Balance factor = heightOfLeftSubtree - heightOfRightSubtree



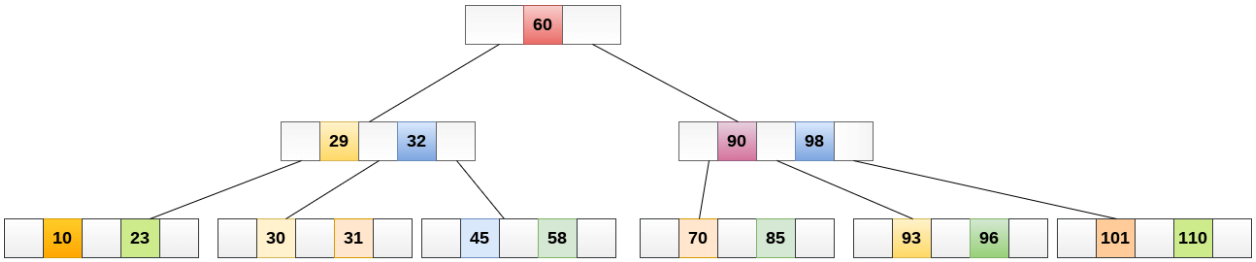
  - The above height-balancing scheme is used in AVL trees. The diagram below shows two trees, one of them is height-balanced and other is not. The second tree is not height-balanced because the height of the left subtree is 2 more than the height of the right subtree.
  - To check if a tree is height-balanced, get the height of left and right subtrees. Return true if the difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.
- **Every AVL Tree is a binary search tree but every Binary Search Tree need not be an AVL tree.**

<u>B - Tree</u> :

- A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions.
- Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data. It is most commonly used in database and file systems.
- One of the main reasons for using B tree is its capability to store a large number of keys in a single node and large key values by keeping the height of the tree relatively small.
- In the B-tree data is sorted in a specific order, with the lowest value on the left and the highest value on the right.
- Unlike a binary tree, in B-tree, a node can have more than two children.
- B-tree has a height of logM N (Where ‘M’ is the order of tree and N is the number of nodes). And the height is adjusted automatically at each update. To insert the data or key in B-tree is more complicated than binary tree.
- In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the

number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time.

- The main idea of using B-Trees is to reduce the number of disk accesses.
- Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree.
- The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.
- While performing some operations on B Tree, any property of B Tree may violate such as the number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.



A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

- The root nodes must have at least 2 nodes.
- All leaf nodes must be at the same level.
- Above the leaf nodes of the B-tree, there should be no empty sub-trees.
- B- tree's height should lie as low as possible.

Traversal in B-Tree:

Traversal is also similar to Inorder traversal of Binary Trees. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

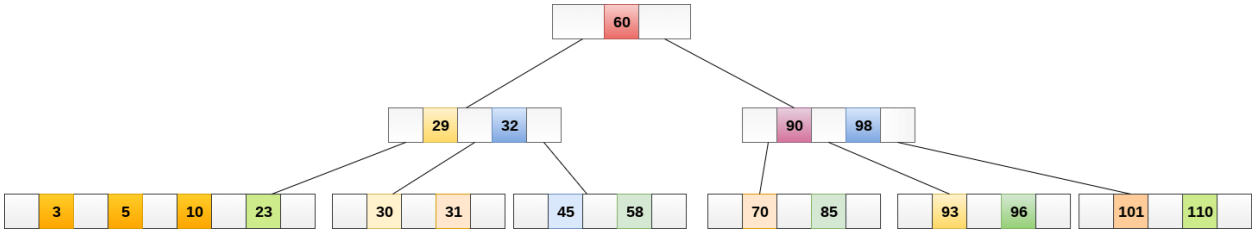
Inserting in B-Tree :

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

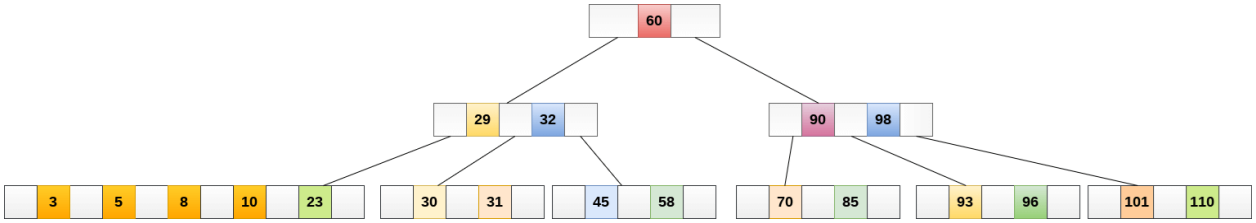
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contains less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.

- Insert the new element in the increasing order of elements.
- Split the node into the two nodes at the median.
- Push the median element up to its parent node.
- If the parent node also contains m-1 number of keys, then split it too by following the same steps.

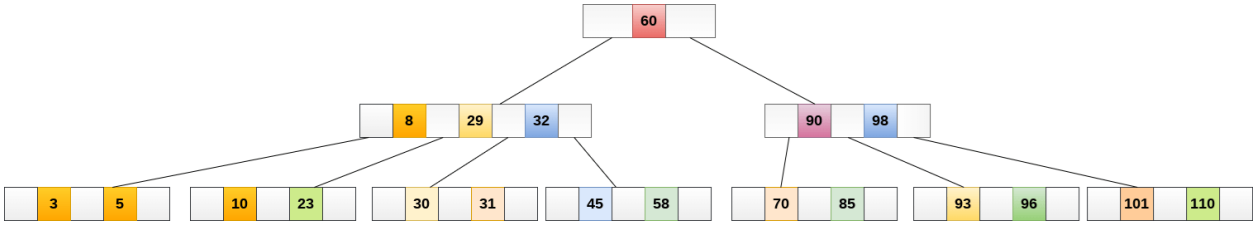
Example: Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node now contains 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



Deletion in B-Tree :

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node.

Following algorithm needs to be followed in order to delete a node from a B tree.

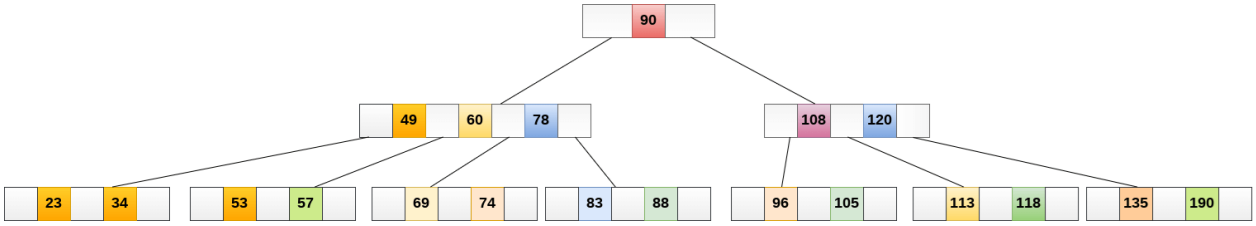
1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from the eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.

- If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move the intervening element down to the node where the key is deleted.

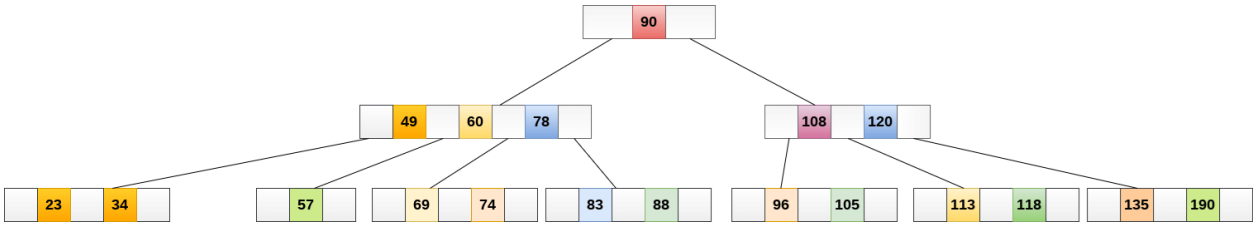
4. If neither of the siblings contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If the parent is left with less than m/2 nodes then, apply the above process on the parent too.

If the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, the successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

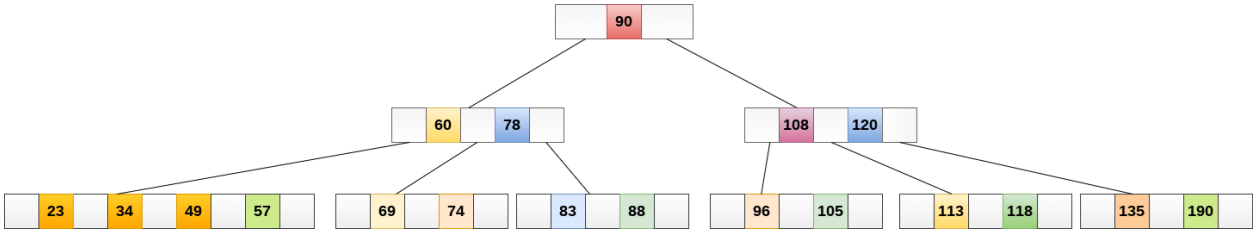
Example 1: Delete the node 53 from the B Tree of order 5 shown in the following figure.



53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right subtree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49. The final B tree is shown as follows.



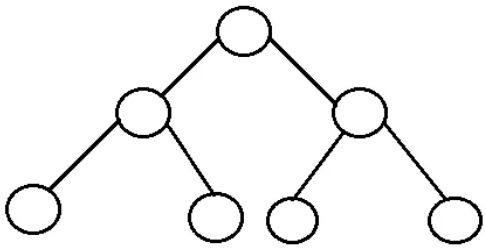
Application of B tree :

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

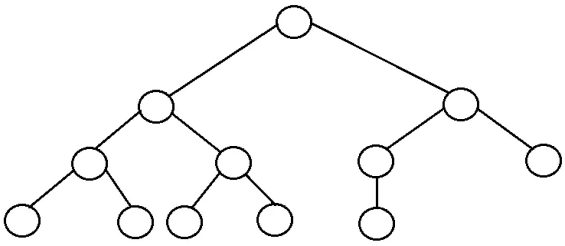
Extra material :

Types of Binary Trees

Full Binary Tree A Binary Tree is a full binary tree if every node has 0 or 2 children. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.



Perfect binary tree: It is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.

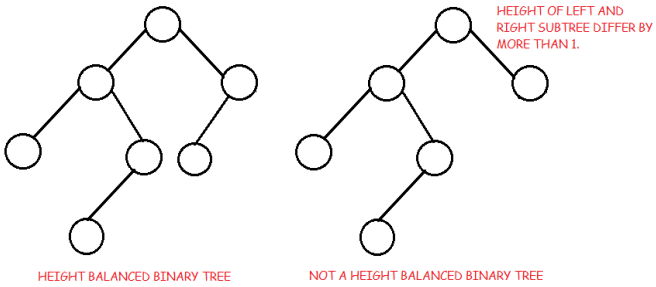


Complete binary tree: It is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

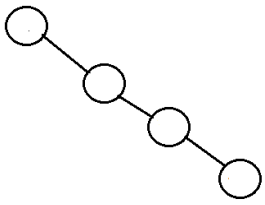
Balanced binary tree: A binary tree is height balanced if it satisfies the following constraints:

1. The left and right subtrees' heights differ by at most one, AND
2. The left subtree is balanced, AND
3. The right subtree is balanced

An empty tree is height balanced.



Degenerate tree: It is a tree where each parent node has only one child node. It behaves like a linked list.



Example to demonstrate insert in binary search tree :(inorder traversal)

// operation in binary search tree.

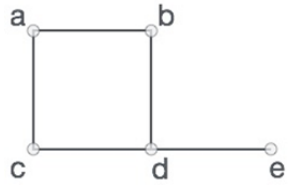
```
#include <stdio.h>
#include <stdlib.h>
struct node {
        int key;
        struct node *left, *right;
```

```
};
struct node* newNode(int item)
{
	struct node* temp = (struct node*)malloc(sizeof(struct node));
	temp->key = item;
	temp->left = temp->right = NULL;
	return temp;
}

// A utility function to do inorder traversal of BST
void inorder(struct node* root)
{
	if (root != NULL) {
		inorder(root->left);
		printf("%d \n", root->key);
		inorder(root->right);
	}
}

/* A utility function to insert a new node with given key in BST */
struct node* insert(struct node* node, int key)
{
	/* If the tree is empty, return a new node */
	if (node == NULL)
		return newNode(key);

	/* Otherwise, recur down the tree */
	if (key < node->key)
		node->left = insert(node->left, key);
	else if (key > node->key)
		node->right = insert(node->right, key);

	/* return the (unchanged) node pointer */
	return node;
}

int main()
{
	/* Let us create following BST
```

```
                  /     \
               30       70
               / \       / \
             20 40    60 80 */
        struct node* root = NULL;
        root = insert(root, 50);
        insert(root, 30);
        insert(root, 20);
        insert(root, 40);
        insert(root, 70);
        insert(root, 60);
        insert(root, 80);

        // print inoder traversal of the BST
        inorder(root);
        return 0;
}
```

Unit 5 : topic 2 : Graph

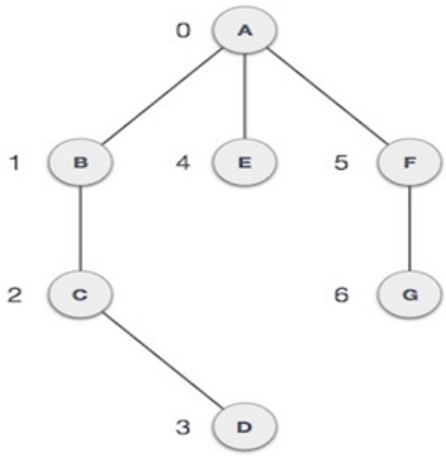
- A Graph is a non-linear data structure consisting of nodes and edges.
- The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.
- **A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes**
- Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



In the above graph,
V = {a, b, c, d, e}
E = {ab, ac, bd, cd, de}

- Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.
- We can represent a graph using an array of vertices and a two-dimensional array of edges.

Before we proceed further, let's familiarize ourselves with some important terms



Vertex − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

Edge − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represent edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

Adjacency (જોડેનું, પાસેનું) − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

Path − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

Basic Operations on Graph :

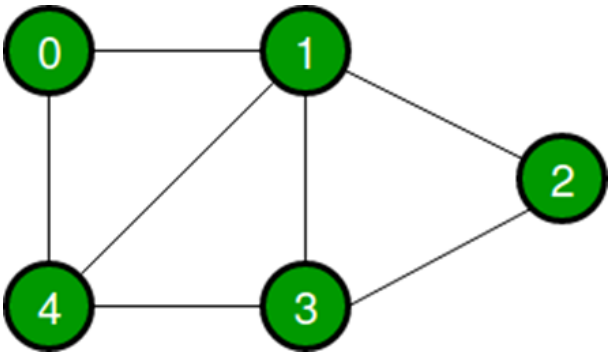
- Add Vertex : Adds a vertex to the graph.
- Remove Vertex : Removes a vertex to the graph.
- Add Edge: Adds an edge between the two vertices of the graph.
- Remove Edge : Removes an edge between the two vertices of the graph.
- Display Vertex : Displays a vertex of the graph.
- Breadth-First Search (BFS)
- Depth First Search (DFS)

Adjacency Matrix and Adjacent Lists :

- A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.
- Graph is a mathematical structure and finds its application in many areas, where the problem is to be solved by computers.
- The problem related to graph G must be represented in computer memory using any suitable data structure to solve the same.
- There are two standard ways of maintaining a graph G in the memory of a computer.

Adjacency Matrix:

- Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for an undirected graph is always symmetric. (સપ્રમાણ)
- The adjacency matrix for the above example graph is:

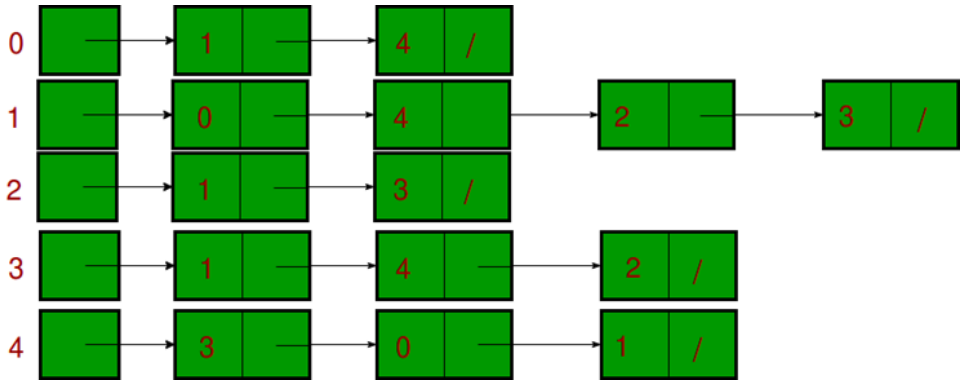


	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Linked representation(i.e. Adjacency list representation) :

- An Adjacency list is an array consisting of the address of all the linked lists.
- The first node of the linked list represents the vertex and the remaining lists connected to this node represent the vertices to which this node is connected.
- This representation can also be used to represent a weighted graph. The linked list can slightly be changed to even store the weight of the edge.

- An array of lists is used. The size of the array is equal to the number of vertices.
- Let the array be an array[].
- The index of the array represents a vertex and each element in its linked list represents the vertices that form an edge with the vertex.
- An entry array[i] represents the list of vertices adjacent to the ith vertex.
- The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.



Graph Traversal :

- 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 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 a looping path.

There are two graph traversal techniques and they are as follows...

1. DFS (Depth First Search)
2. BFS (Breadth First Search/level order)

1. Depth(ತಿರುಳು) First Traversal:

We use the following steps to implement DFS traversal...

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it onto the Stack.

Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of the stack and push it onto the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use backtracking and pop one vertex from the stack.

Step 6 - Repeat steps 3, 4 and 5 until the stack becomes Empty.

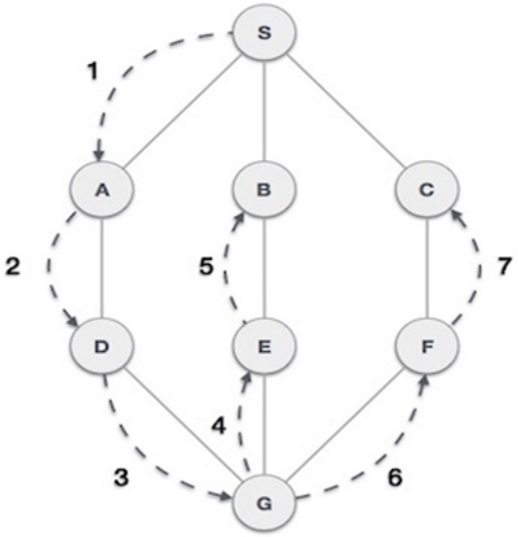
Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

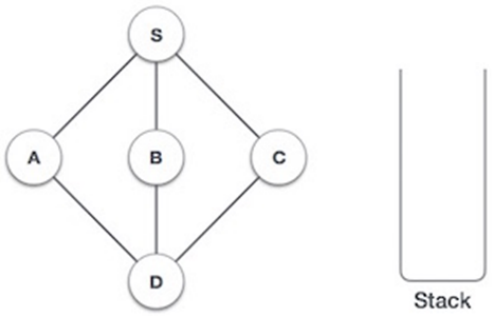
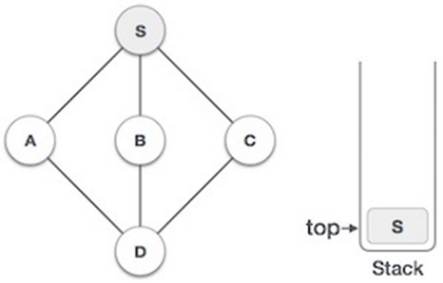
As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

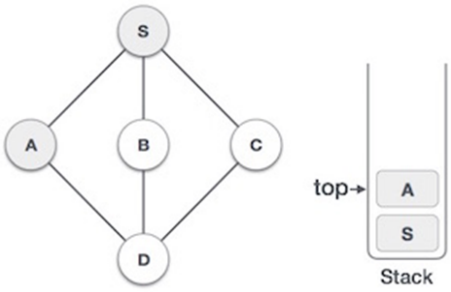
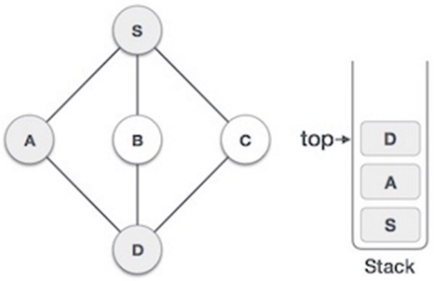
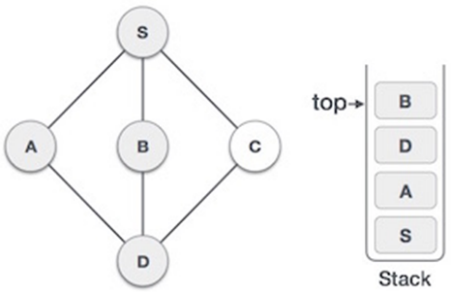
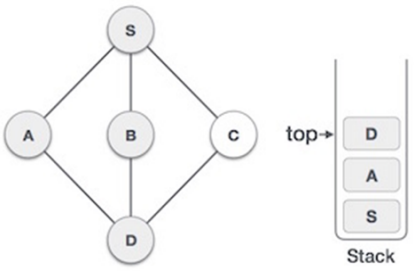
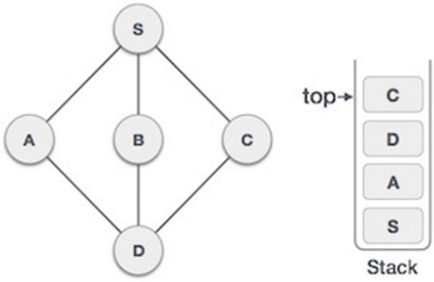
Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 2 − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 − Repeat Rule 1 and Rule 2 until the stack is empty.

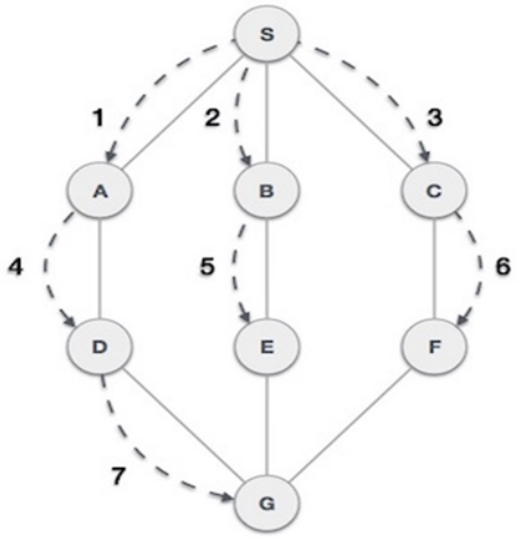


Step	Traversal	Description
1		Initialize the stack.
2		Mark S visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

3		Mark A visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.
4		Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent nodes. So, we pop B from the stack.
6		We check the stack top to return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.
7		Only unvisited adjacent node from D is C now. So we visit C, mark it as visited and put it onto the stack.

As C does not have any unvisited adjacent nodes, we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Breadth(પહોળાઈ) First Traversal :



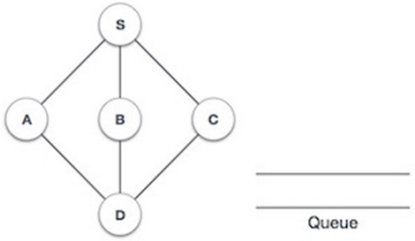
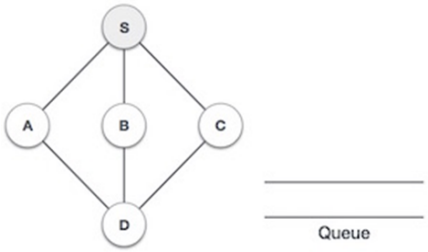
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

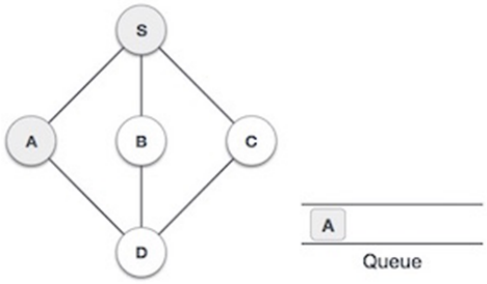
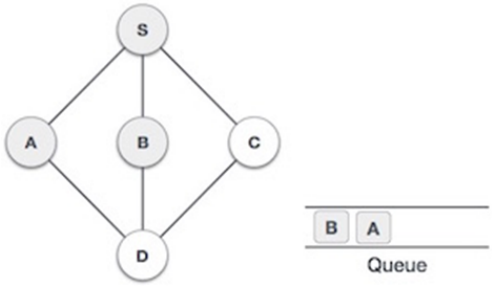
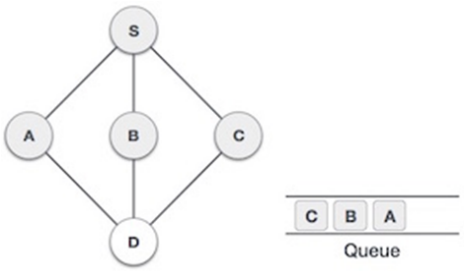
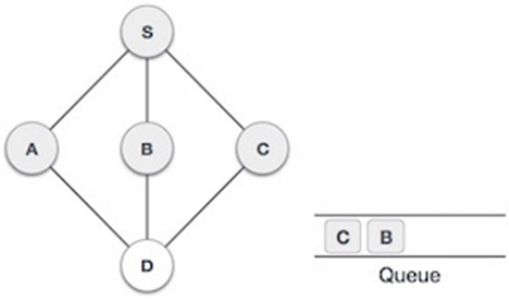
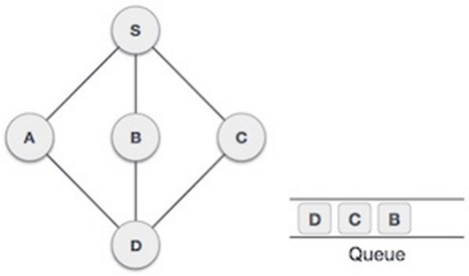
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 − If no adjacent vertex is found, remove the first vertex from the queue.

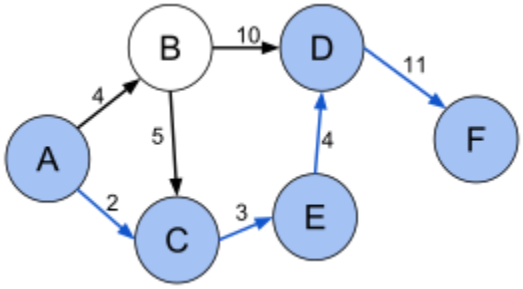
Rule 3 − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting S (starting node), and mark it as visited.

3		We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.
6		Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.
7		From A we have D as unvisited adjacent node. We mark it a

Shortest Path Problem :

- As we always do when we want to reach from one station to another, at that time the driver takes the shortest possible route to reach the destination.
- There are many instances to find the shortest path for traveling from one place to another.
- That is to find which route can reach as quick as possible of a route for which the traveling cost is minimum.
- In a graph, finding the shortest path is the most important problem.
- In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.



- Shortest path algorithms are applied to automatically find directions between physical locations, such as driving directions on web mapping websites like MapQuest or Google Maps. For this application fast specialized algorithms are available.

[Shortest path (A, C, E, D, F) between vertices A and F in the weighted directed graph The problem is also sometimes called the **single-pair shortest path problem**, to distinguish it from the following variations: **]**

- The **single-source shortest path problem**, in which we have to find shortest paths from a source vertex *v* to all other vertices in the graph.
- The **single-destination shortest path problem**, in which we have to find shortest paths from all vertices in the directed graph to a single destination vertex *v*. This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.
- The **all-pairs shortest path problem**, in which we have to find shortest paths between every pair of vertices *v*, *v′* in the graph.

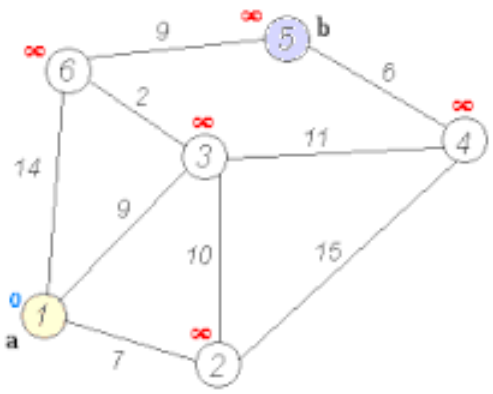
The most important algorithms for solving this problem are:

- **Dijkstra's algorithm** solves the single-source shortest path problem with non-negative edge weight.
- **Bellman–Ford algorithm** solves the single-source problem if edge weights may be negative.
- **A* search algorithm** solves for single-pair shortest path using heuristics to try to speed up the search.
- **Floyd–Warshall algorithm** solves all pairs shortest paths.
- **Johnson's algorithm** solves all pairs' shortest paths, and may be faster than Floyd–Warshall on sparse graphs.

- **The Viterbi algorithm** solves the shortest stochastic path problem with an additional probabilistic weight on each node.

DIJKSTRA'S ALGORITHM

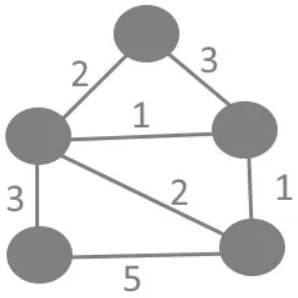
Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist **Edsger W. Dijkstra in 1956** and published three years later.



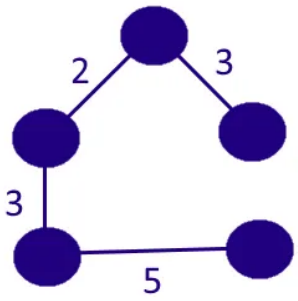
[Dijkstra's algorithm to find the shortest path between a and b. It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited (set to red) when done with neighbors.]

Minimal Spanning Tree :

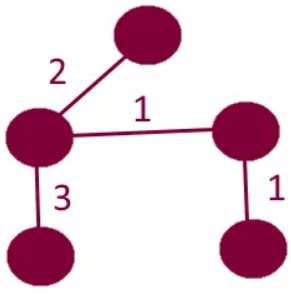
- A spanning tree of a graph is a collection of connected edges that include every vertex in the graph, but **that do not form a cycle**.
- A spanning tree of a graph is just a sub graph that contains all the vertices and is a tree(with no cycle).
- A graph may have many spanning trees.
- A minimum spanning tree(MST) for a graph is a subgraph of a graph that contains all the vertices of G(graph).
- If a graph G is not a connected graph, then it cannot have any spanning tree.



Graph



Spanning Tree
Cost = 13



Minimum Spanning
Tree, Cost = 7

- To obtain minimum spanning tree of connected weighted and undirected graph, different algorithms are use which are listed as under:
    i. Krukal's Algorithm
    ii. Prim's Algorithm

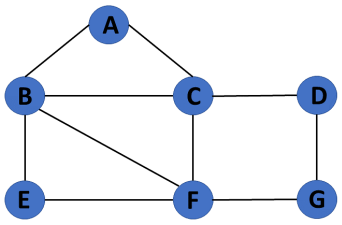
iii. Sollin's Algorithm

Extra Material for this unit :

Types of Graphs in Data Structures

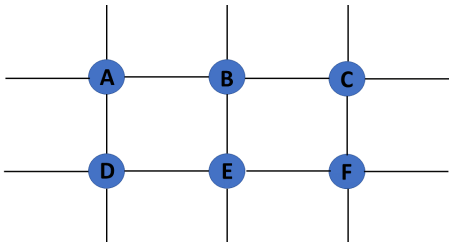
1. Finite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is limited in number



2. Infinite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is interminable.

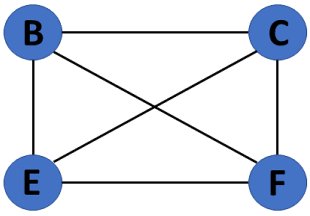


3. Trivial Graph

A graph G= (V, E) is trivial if it contains only a single vertex and no edges.

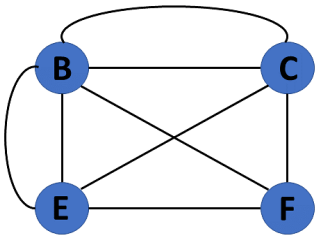
4. Simple Graph

If each pair of nodes or vertices in a graph G=(V, E) has only one edge, it is a simple graph. As a result, there is just one edge linking two vertices, depicting one-to-one interactions between two elements.



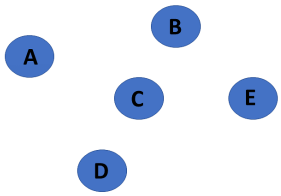
5. Multi Graph

If there are numerous edges between a pair of vertices in a graph G= (V, E), the graph is referred to as a multigraph. There are no self-loops in a Multigraph.



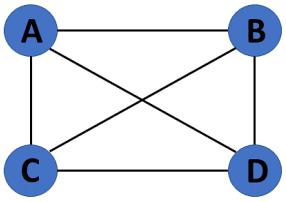
6. Null Graph

It's a reworked version of a trivial graph. If several vertices but no edges connect them, a graph G= (V, E) is a null graph.



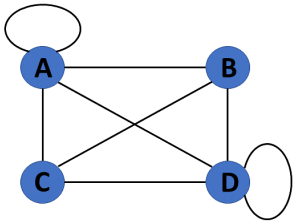
7. Complete Graph

If a graph G= (V, E) is also a simple graph, it is complete. Using the edges, with n number of vertices must be connected. It's also known as a full graph because each vertex's degree must be n-1.



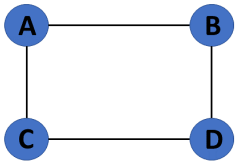
8. Pseudo Graph

If a graph G= (V, E) contains a self-loop besides other edges, it is a pseudograph.



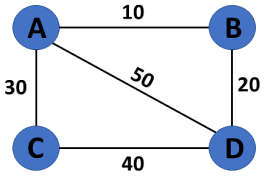
9. Regular Graph

If a graph G= (V, E) is a simple graph with the same degree at each vertex, it is a regular graph. As a result, every whole graph is a regular graph.



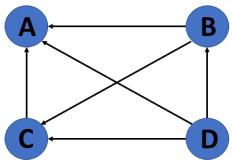
10. Weighted Graph

A graph G= (V, E) is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



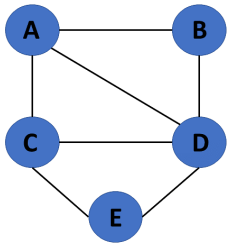
11. Directed Graph

A directed graph, also referred to as a digraph, is a set of nodes connected by edges, each with a direction.



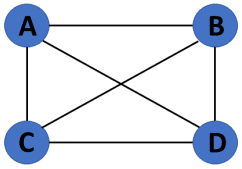
12. Undirected Graph

An undirected graph comprises a set of nodes and links connecting them. The order of the two connected vertices is irrelevant and has no direction. You can form an undirected graph with a finite number of vertices and edges.



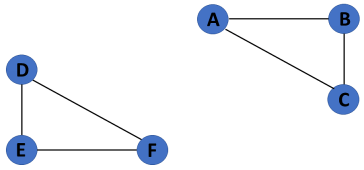
13. Connected Graph

If there is a path between one vertex of a graph data structure and any other vertex, the graph is connected.



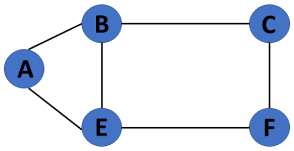
14. Disconnected Graph

When there is no edge linking the vertices, you refer to the null graph as a disconnected graph.



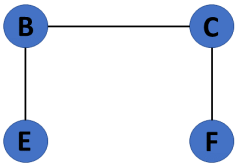
15. Cyclic Graph

If a graph contains at least one graph cycle, it is considered to be cyclic.



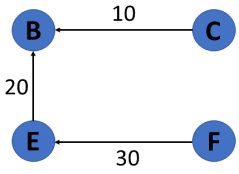
16. Acyclic Graph

When there are no cycles in a graph, it is called an acyclic graph.



17. Directed Acyclic Graph

It's also known as a directed acyclic graph (DAG), and it's a graph with directed edges but no cycle. It represents the edges using an ordered pair of vertices since it directs the vertices and stores some data.



18. Subgraph

The vertices and edges of a graph that are subsets of another graph are known as a subgraph.