ADVANCED DATA STRUCTURES

MSC COMPUTER SCIENCE - GUIDE

DHANYA ANTO
Guest Lecturer, Department of Computer Science
Prajyoti Niketan College, Pudukad

Data Structure

Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vitle role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible

Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

Record: Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File: A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field: Field is a single elementary unit of information representing the attribute of an entity.

Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arrise the following problems:

Processor speed: To handle very large amout of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process

In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

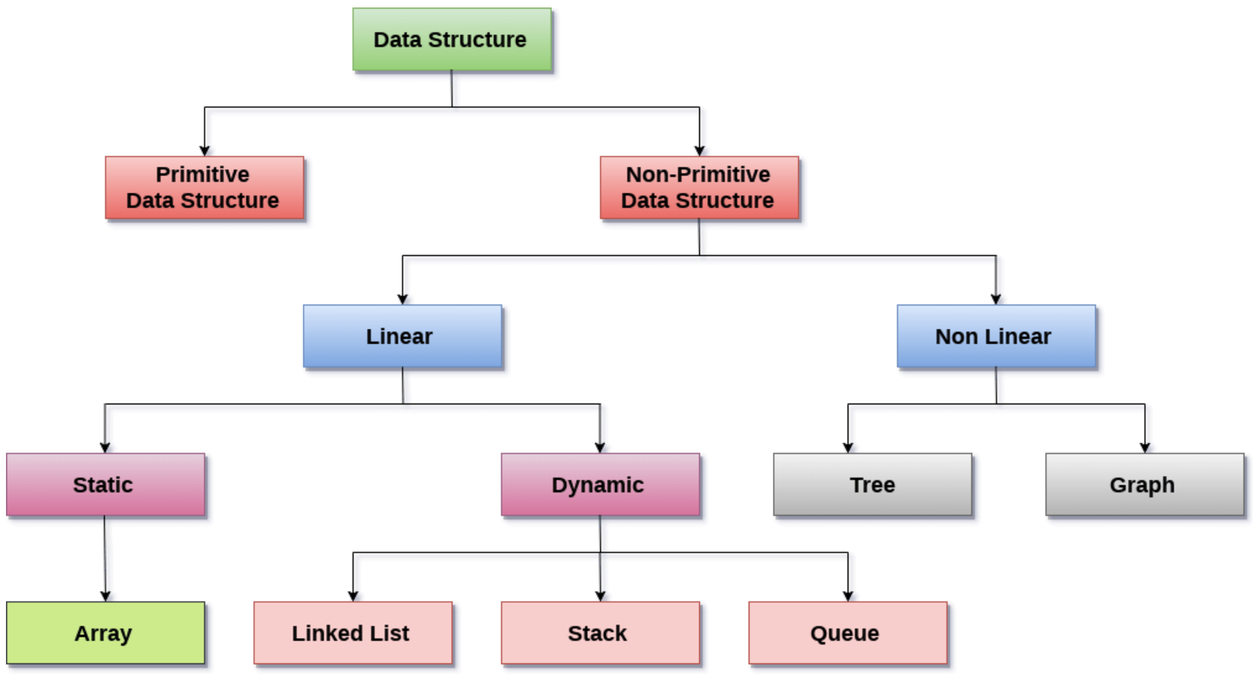
Advantages of Data Structures

Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a perticular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Data Structure Classification



Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in a non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of the array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],......... age[98], age[99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows the First-In-First-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures: This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the herierchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classfied into many categories which will be discussed later in this tutorial.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

Operations on data structure

1) **Traversing:** Every data structure contains the 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.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **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.

3) **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.

4) **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.

5) **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.

6) **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

DS Algorithm

What is an Algorithm?

An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. 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 or Pseudocode.

Characteristics of an Algorithm

The following are the characteristics of an algorithm:

- **Input:** An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- **Output:** We will get 1 or more output at the end of an algorithm.
- **Unambiguity:** An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- **Finiteness:** An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- **Effectiveness:** An algorithm should be effective as each instruction in an algorithm affects the overall process.
- **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.

Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem which is a step by step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

Why do we need Algorithms?

We need algorithms because of the following reasons:

- **Scalability:** It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.
- **Performance:** The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

Let's understand the algorithm through a real-world example. Suppose we want to make a lemon juice, so following are the steps required to make a lemon juice:

Step 1: First, we will cut the lemon into half.

Step 2: Squeeze the lemon as much you can and take out its juice in a container.

Step 3: Add two tablespoon sugar in it.

Step 4: Stir the container until the sugar gets dissolved.

Step 5: When sugar gets dissolved, add some water and ice in it.

Step 6: Store the juice in a fridge for 5 to minutes.

Step 7: Now, it's ready to drink.

The above real-world can be directly compared to the definition of the algorithm. We cannot perform the step 3 before the step 2, we need to follow the specific order to make lemon juice. An algorithm also says that each and every instruction should be followed in a specific order to perform a specific task.

Now we will look an example of an algorithm in programming.

We will write an algorithm to add two numbers entered by the user.

The following are the steps required to add two numbers entered by the user:

Step 1: Start

Step 2: Declare three variables a, b, and sum.

Step 3: Enter the values of a and b.

Step 4: Add the values of a and b and store the result in the sum variable, i.e., sum=a+b.

Step 5: Print sum

Step 6: Stop

Factors of an Algorithm

The following are the factors that we need to consider for designing an algorithm:

- **Modularity:** If any problem is given and we can break that problem into small-small modules or small-small steps, which is a basic definition of an algorithm, it means that this feature has been perfectly designed for the algorithm.
- **Correctness:** The correctness of an algorithm is defined as when the given inputs produce the desired output, which means that the algorithm has been designed algorithm. The analysis of an algorithm has been done correctly.
- **Maintainability:** Here, maintainability means that the algorithm should be designed in a very simple structured way so that when we redefine the algorithm, no major change will be done in the algorithm.
- **Functionality:** It considers various logical steps to solve the real-world problem.
- **Robustness:** Robustness means that how an algorithm can clearly define our problem.
- **User-friendly:** If the algorithm is not user-friendly, then the designer will not be able to explain it to the programmer.

- **Simplicity:** If the algorithm is simple then it is easy to understand.
- **Extensibility:** If any other algorithm designer or programmer wants to use your algorithm then it should be extensible.

Importance of Algorithms

1. **Theoretical importance:** When any real-world problem is given to us and we break the problem into small-small modules. To break down the problem, we should know all the theoretical aspects.
2. **Practical importance:** As we know that theory cannot be completed without the practical implementation. So, the importance of algorithm can be considered as both theoretical and practical.

Issues of Algorithms

The following are the issues that come while designing an algorithm:

- **How to design algorithms:** As we know that an algorithm is a step-by-step procedure so we must follow some steps to design an algorithm.
- **How to analyze algorithm efficiency**

Approaches of Algorithm

The following are the approaches used after considering both the theoretical and practical importance of designing an algorithm:

- **Brute force algorithm:** The general logic structure is applied to design an algorithm. It is also known as an exhaustive search algorithm that searches all the possibilities to provide the required solution. Such algorithms are of two types:
    1. **Optimizing:** Finding all the solutions of a problem and then take out the best solution or if the value of the best solution is known then it will terminate if the best solution is known.

2. **Sacrificing:** As soon as the best solution is found, then it will stop.

- **Divide and conquer:** It is a very implementation of an algorithm. It allows you to design an algorithm in a step-by-step variation. It breaks down the algorithm to solve the problem in different methods. It allows you to break down the problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.
- **Greedy algorithm:** It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting the best solution. It is easy to implement and has a faster execution time. But, there are very rare cases in which it provides the optimal solution.
- **Dynamic programming:** It makes the algorithm more efficient by storing the intermediate results. It follows five different steps to find the optimal solution for the problem:
    1. It breaks down the problem into a subproblem to find the optimal solution.
    2. After breaking down the problem, it finds the optimal solution out of these subproblems.
    3. Stores the result of the subproblems is known as memorization.
    4. Reuse the result so that it cannot be recomputed for the same subproblems.
    5. Finally, it computes the result of the complex program.
- **Branch and Bound Algorithm:** The branch and bound algorithm can be applied to only integer programming problems. This approach divides all the sets of feasible solutions into smaller subsets. These subsets are further evaluated to find the best solution.
- **Randomized Algorithm:** As we have seen in a regular algorithm, we have predefined input and required output. Those algorithms that have some defined set of inputs and required output, and follow some described steps are known as deterministic algorithms. What happens that when the random variable is introduced in the randomized algorithm?. In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output, which is random in nature. Randomized algorithms are simpler and efficient than the deterministic algorithm.
- **Backtracking:** Backtracking is an algorithmic technique that solves the problem recursively and removes the solution if it does not satisfy the constraints of a problem.

The major categories of algorithms are given below:

- **Sort:** Algorithm developed for sorting the items in a certain order.
- **Search:** Algorithm developed for searching the items inside a data structure.
- **Delete:** Algorithm developed for deleting the existing element from the data structure.
- **Insert:** Algorithm developed for inserting an item inside a data structure.
- **Update:** Algorithm developed for updating the existing element inside a data structure.

Algorithm Analysis

The algorithm can be analyzed in two levels, i.e., first is before creating the algorithm, and second is after creating the algorithm. The following are the two analysis of an algorithm:

- 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 evaluate that how much running time and space taken by the algorithm.

What is Performance Analysis of an algorithm?

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements. The formal definition is as follows...

Performance of an algorithm is a process of making evaluative judgement about algorithms.

It can also be defined as follows...

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.

We compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.,

Generally, the performance of an algorithm depends on the following elements...

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

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

Based on this information, performance analysis of an algorithm can also be defined as follows...

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

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

1. Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
2. Time required to complete the task of that algorithm (**Time Complexity**)

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. 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. The time complexity is mainly calculated by counting the number of steps to finish the execution. Let's understand the time complexity through an example.

```
sum=0;
// Suppose we have to calculate the sum of n numbers.
for i=1 to n
sum=sum+i;
// when the loop ends then sum holds the sum of the n numbers
return sum;
```

In the above code, the time complexity of the loop statement will be atleast n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. 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.

What is Space complexity?

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

1. To store program instructions.
2. To store constant values.
3. To store variable values.
4. And for few other things like funcion calls, jumping statements etc,.

Space complexity of an algorithm can be defined as follows...

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

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

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

Note - When we want to perform analysis of an algorithm based on its Space complexity, we consider only Data Space and ignore Instruction Space as well as Environmental Stack.

That means we calculate only the memory required to store Variables, Constants, Structures, etc.,

To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following...

1. 2 bytes to store Integer value.
2. 4 bytes to store Floating Point value.
3. 1 byte to store Character value.
4. 6 (OR) 8 bytes to store double value.

Consider the following piece of code...

Example 1

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

In the above piece of code, it requires 2 bytes of memory to store variable **'a'** and another 2 bytes of memory is used for **return value**.

That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be *Constant Space Complexity*.

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity.

Consider the following piece of code...

Example 2

```
int sum(int A[ ], int n)
{
  int sum = 0, i;
  for(i = 0; i < n; i++)
    sum = sum + A[i];
```

```
   return sum;
}
```

In the above piece of code it requires

'n*2' bytes of memory to store array variable **'a[ ]'**

2 bytes of memory for integer parameter **'n'**

4 bytes of memory for local integer variables **'sum'** and **'i'** (2 bytes each)

2 bytes of memory for **return value**.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the total amount of memory required depends on the value of 'n'. As 'n' value increases the space required also increases proportionately. This type of space complexity is said to be *Linear Space Complexity*.

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity.

What is Time complexity?

Every algorithm requires some amount of computer time to execute its instruction to perform the task.

This computer time required is called time complexity.

The time complexity of an algorithm can be defined as follows...

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, the running time of an algorithm depends upon the following...

1. Whether it is running on **Single** processor machine or **Multi** processor machine.
2. Whether it is a **32 bit** machine or **64 bit** machine.
3. **Read** and **Write** speed of the machine.
4. The amount of time required by an algorithm to perform **Arithmetic** operations, **logical** operations, **return** value and **assignment** operations etc.,
5. **Input** data

Note - When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent. We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc.,

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because the configuration changes from one system to another system. To solve this problem, we must assume a model machine with a specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.

To calculate the time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. It is a Single processor machine
2. It is a 32 bit Operating System machine
3. It performs sequential execution
4. It requires 1 unit of time for Arithmetic and Logical operations
5. It requires 1 unit of time for Assignment and Return value
6. It requires 1 unit of time for Read and Write operations

Now, we calculate the time complexity of following example code by using the above-defined model machine...

Consider the following piece of code...

Example 1

```
int sum(int a, int b)
{
   return a+b;
}
```

In the above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires the same amount of time i.e. 2 units.

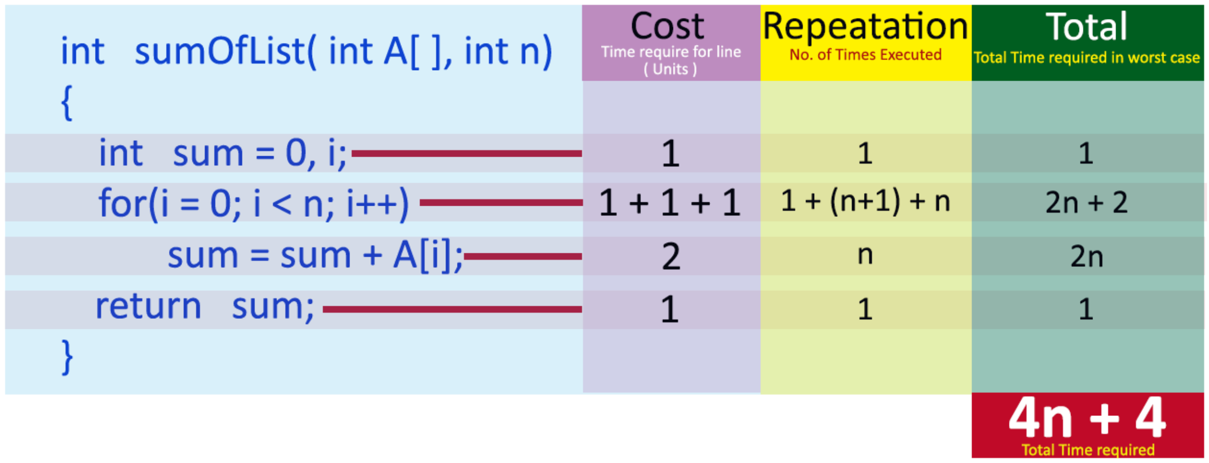
If any program requires a fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

Consider the following piece of code...

Example 2

```
int sum(int A[], int n)
{
   int sum = 0, i;
   for(i = 0; i < n; i++)
      sum = sum + A[i];
   return sum;
}
```

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



int sumOfList(int A[ ], int n) {	Cost Time require for line (Units)	Repeatation No. of Times Executed	Total Total Time required in worst case
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	1 + 1 + 1	1 + (n+1) + n	2n + 2
sum = sum + A[i];	2	n	2n
return sum;	1	1	1
}			
			4n + 4 Total Time required

In above calculation **Cost** is the amount of computer time required for a single operation in each line.

Repeatation is the amount of computer time required by each operation for all its repeatations.

Total is the amount of computer time required by each operation to execute.

So above code requires **'4n+4' Units** of computer time to complete the task. Here the exact time is not fixed. And it changes based on the **n** value. If we increase the **n** value then the time required also increases linearly.

Totally it takes '4n+4' units of time to complete its execution and it is *Linear Time Complexity*.

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity.

Types of Algorithms

The following are the types of algorithm:

Search Algorithm

Sort Algorithm

Search Algorithms

On each day, we search for something in our day to day life. Similarly, with the case of computer, huge data is stored in a computer that whenever the user asks for any data then the computer searches for that data in the memory and provides that data to the user. There are mainly two techniques available to search the data in an array:

Linear search

Binary search

Linear Search

Linear search is a very simple algorithm that starts searching for an element or a value from the beginning of an array until the required element is not found. It compares the element to be searched with all the elements in an array, if the match is found, then it returns the index of the element else it returns -1. This algorithm can be implemented on the unsorted list.

Binary Search

A Binary algorithm is the simplest algorithm that searches the element very quickly. It is used to search the element from the sorted list. The elements must be stored in sequential order or the sorted manner to implement the binary algorithm. Binary search cannot be implemented if the elements are stored in a random manner. It is used to find the middle element of the list.

Sorting Algorithms

Sorting algorithms are used to rearrange the elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements.

Why do we need a sorting algorithm?

- An efficient sorting algorithm is required for optimizing the efficiency of other algorithms like binary search algorithm as a binary search algorithm requires an array to be sorted in a particular order, mainly in ascending order.
- It produces information in a sorted order, which is a human-readable format.
- Searching a particular element in a sorted list is faster than the unsorted list.

Arrays

What is an Array?

Whenever we want to work with large number of data values, we need to use that much number of different variables. As the number of variables are increasing, complexity of the program also increases and programmers get confused with the variable names. There may be situations in which we need to work with large number of similar data values. To make this work more easy, C programming language provides a concept called "Array".

An array is a variable which can store multiple values of same data type at a time.

An array can also be defined as follows...

"Collection of similar data items stored in continuous memory locations with single name".

To understand the concept of arrays, consider the following example declaration.

int a, b, c;

Here, the compiler allocates 2 bytes of memory with name 'a', another 2 bytes of memory with name 'b' and more 2 bytes with name 'c'. These three memory locations are may be in sequence or may not be in sequence. Here these individual variables store only one value at a time.

Now consider the following declaration...

int a[3];

Here, the compiler allocates total 6 bytes of continuous memory locations with single name 'a'. But allows to store three different integer values (each in 2 bytes of memory) at a time. And memory is organized as follows...

That means all these three memory locations are named as 'a'. But "how can we refer individual elements?" is the big question. Answer for this question is, compiler not only allocates memory, but also assigns a numerical value to each individual element of an array. This numerical value is called as "Index". Index values for the above example are as follows...

The individual elements of an array are identified using the combination of 'name' and 'index' as follows...

arrayName[indexValue]

For the above example, the individual elements can be referred as follows...

If I want to assign a value to any of these memory locations (array elements), we can assign as follows...

a[1] = 100;

The result will be as follows...

There are different types of arrays in c programming language. Click on the below link to read TYPES OF ARRAYS

Data Structure and Algorithms - Stack

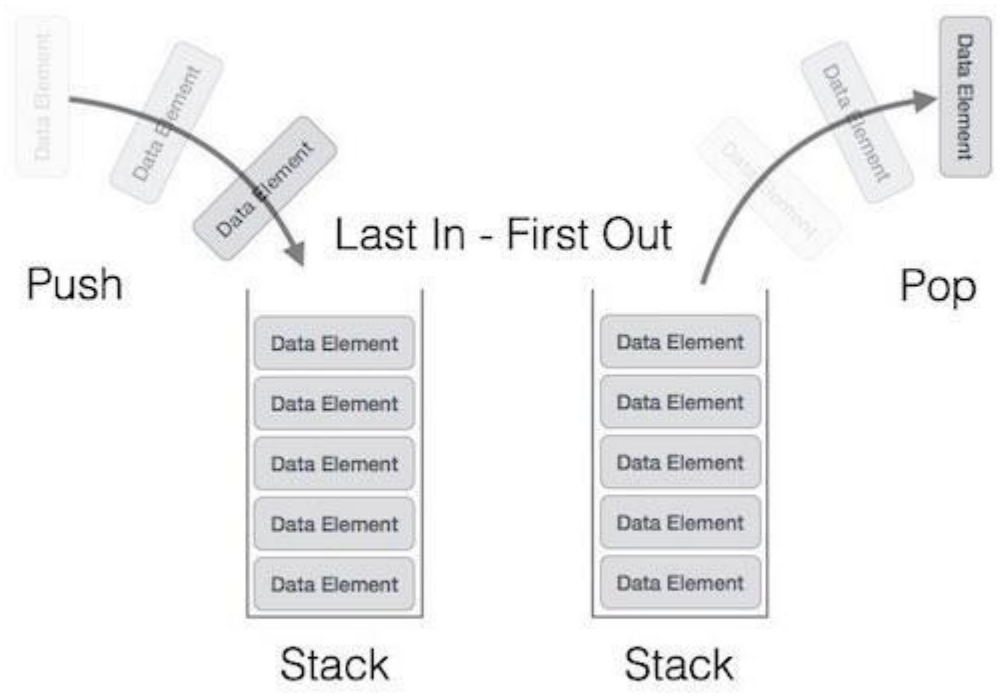
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates , etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- push() − Pushing (storing) an element on the stack.
- pop() − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- peek() − get the top data element of the stack, without removing it.
- isFull() − check if stack is full.
- isEmpty() − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek
   return stack[top]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return stack[top];   }
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

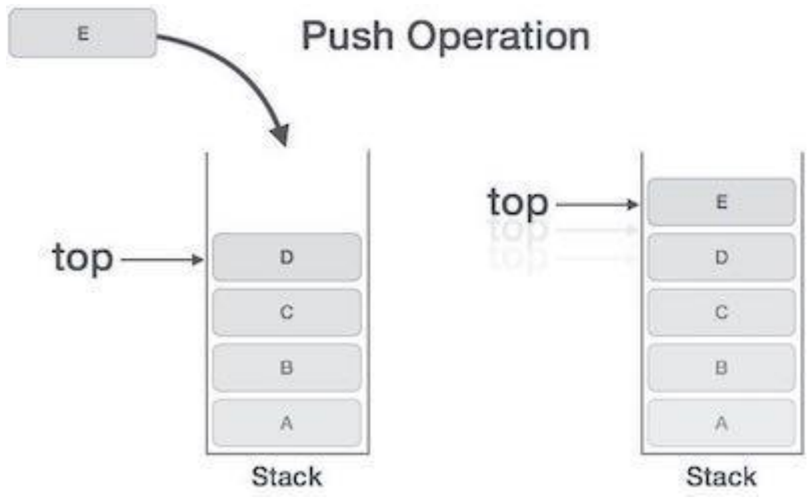
Example

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

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

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



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1

   stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

Example

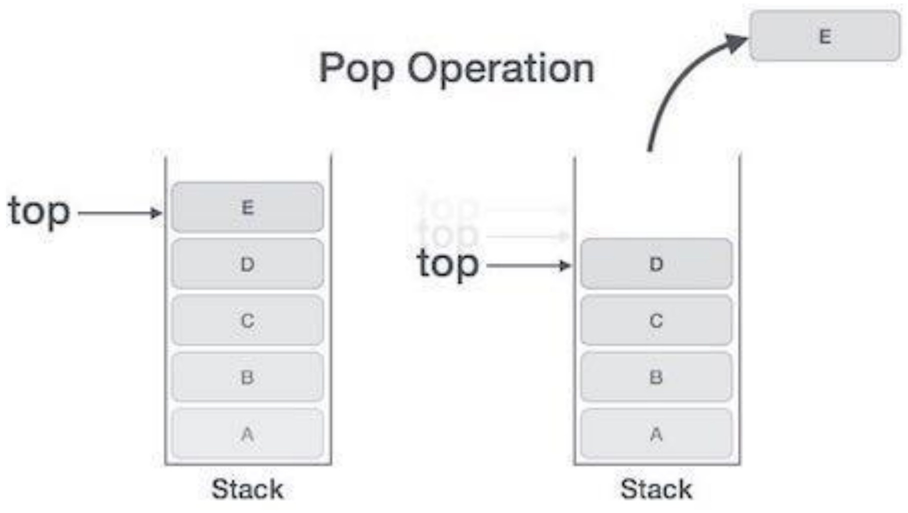
```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

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 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 element and deallocates memory space.

A Pop operation may involve the following steps −

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



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

  if stack is empty
    return null
  endif
```

```
   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Stack Applications- Expression Parsing

The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in infix notation, e.g. a - b + c, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as Polish Notation.

Postfix Notation

This notation style is known as **Reversed** Polish Notation. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+. This is equivalent to its infix notation a + b.

The following table briefly tries to show the difference in all three notations −

Sr.No.	**Infix Notation**	**Prefix Notation**	**Postfix Notation**
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *

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

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

$$a + b * c \Rightarrow a + (b * c)$$

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as (a + b) − c.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Sr.No.	Operator	Precedence	Associativity

1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a + b*c**, the expression part b*c will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for a + b to be evaluated first, like

1. Postfix Expression Evaluation Algorithm

Evaluation rule of a Postfix Expression states:

1. While reading the expression from left to right, push the element in the stack if it is an operand.
2. Pop the two operands from the stack, if the element is an operator and then evaluate it.
3. Push back the result of the evaluation. Repeat it till the end of the expression.

Algorithm

1) Add) to postfix expression.

2) Read postfix expression Left to Right until) encountered

3) If operand is encountered, push it onto Stack

[End If]

4) If operator is encountered, Pop two elements

i) A -> Top element

ii) B-> Next to Top element

iii) Evaluate B operator A

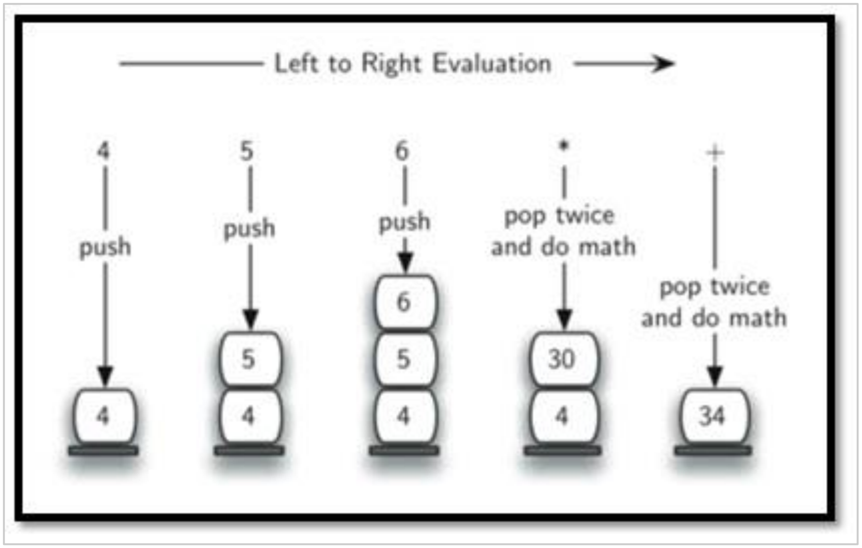
push B operator A onto Stack

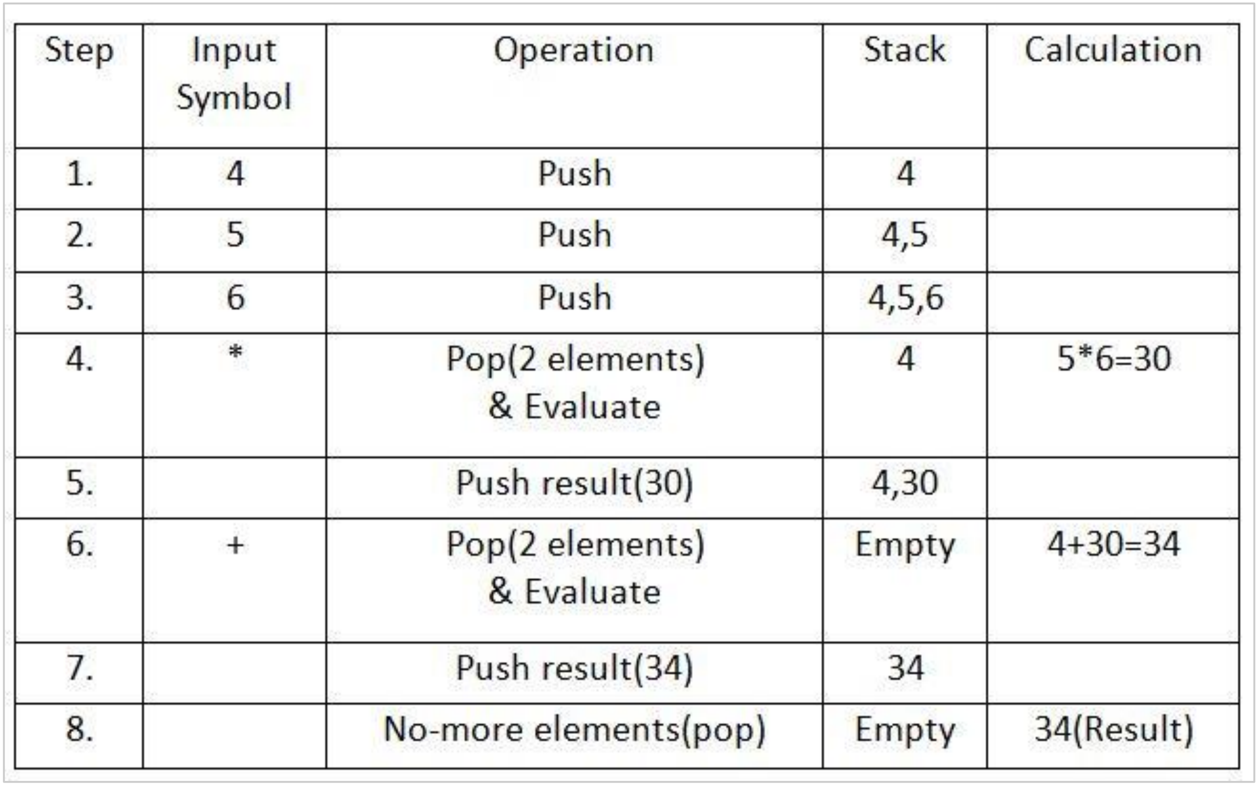
5) Set result = pop

6) END

Let's see an example to better understand the algorithm:

Expression: 456*+





Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

2 . Infix To Postfix Conversion

- **(C*B)+A)**

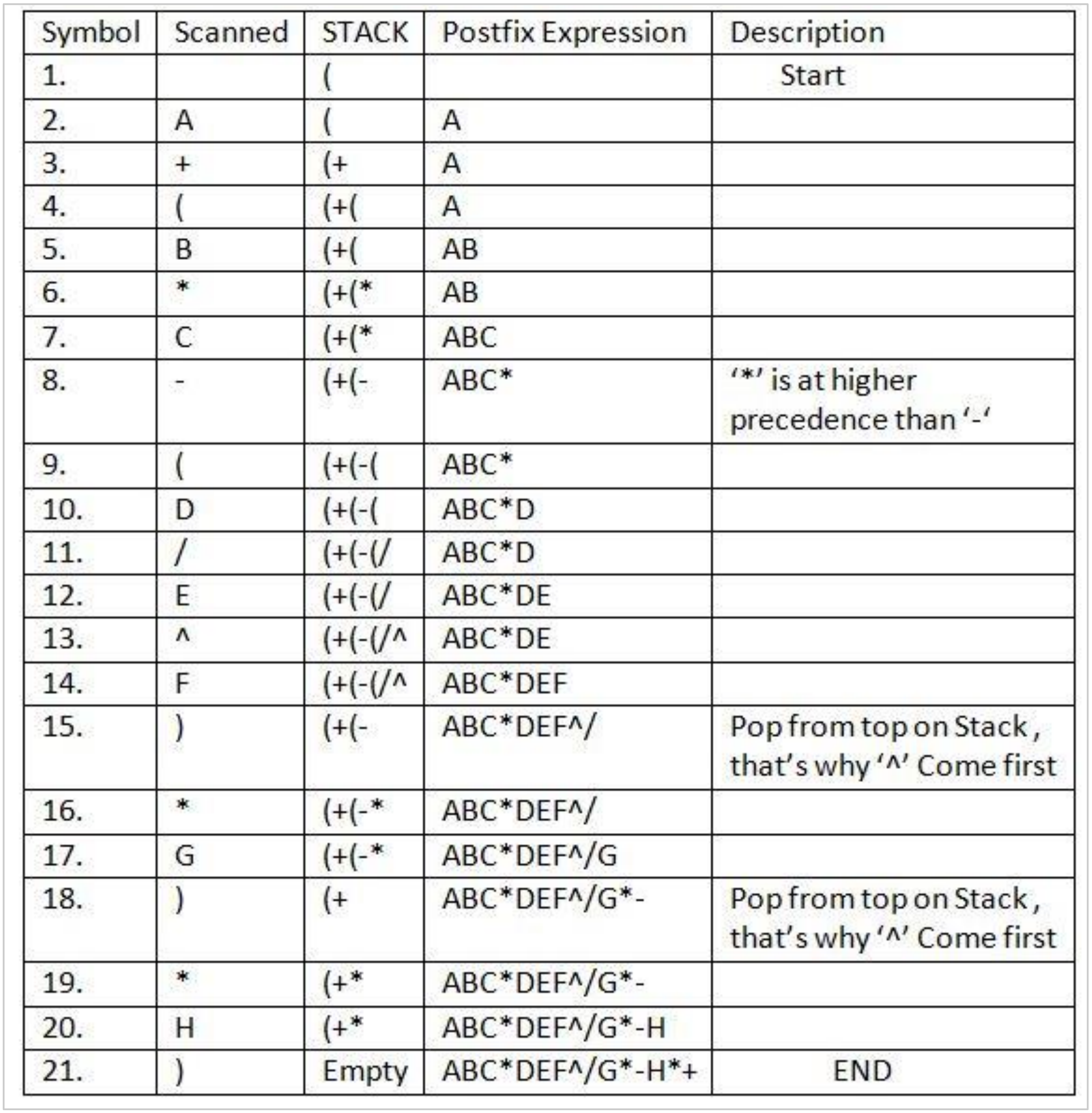
Algorithm

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push “(“onto Stack, and add “)” to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
       [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.
       [End of If]
       [End of If]
7. END.

Let’s take an example to better understand the algorithm

Infix Expression: A+ (B*C-(D/E^F)*G)*H, where ^ is an exponential operator.



Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: ABC*DEF^/G*-H*+

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

3. Infix to Prefix Conversion

Method 1

Algorithm of Infix to Prefix

```
Step 1. Push ")" onto STACK, and add "(" to end of the A
Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
Step 3. If an operand is encountered add it to B
Step 4. If a right parenthesis is encountered push it onto STACK
Step 5. If an operator is encountered then:
        a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator.
        b. Add operator to STACK
Step 6. If left parenthesis is encontered then
        a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encounterd)
        b. Remove the left parenthesis
Step 7. Exit
```

Method 2

We use the Infix to Postfix conversion to convert Infix to Prefix.

- **Step 1: Reverse the infix expression i.e A+(B*C) will become)C*B(+A. Note while reversing each '(' will become ')' and each ')' becomes '('.   (C*B)+A**
- **Step 2: Obtain the "nearly" postfix expression of the modified expression i.e CB*A+**
- **Step 3: Reverse the postfix expression. Hence in our example prefix is   +A*BC**

Program

Infix notation is easier for humans to read and understand whereas for electronic machines like computers, postfix is the best form of expression to parse. We shall see here a program to convert and evaluate infix notation to postfix notation −

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

```
#include<string.h>

//char stack
char stack[25];
int top = -1;

void push(char item) {
   stack[++top] = item;
}

char pop() {
   return stack[top--];
}

//returns precedence of operators
int precedence(char symbol) {

   switch(symbol) {
      case '+':
      case '-':
         return 2;
         break;
      case '*':
      case '/':
         return 3;
         break;
      case '^':
         return 4;
         break;
      case '(':
      case ')':
      case '#':
         return 1;
         break;
   }
}

//check whether the symbol is operator?
int isOperator(char symbol) {

   switch(symbol) {
      case '+':
      case '-':
```

```
    case '*':
    case '/':
    case '^':
    case '(':
    case ')':
      return 1;
    break;
      default:
      return 0;
  }
}

//converts infix expression to postfix
void convert(char infix[],char postfix[]) {
  int i,symbol,j = 0;
  stack[++top] = '#';
	
  for(i = 0;i<strlen(infix);i++) {
    symbol = infix[i];
		
    if(isOperator(symbol) == 0) {
      postfix[j] = symbol;
      j++;
    } else {
      if(symbol == '(') {
        push(symbol);
      } else {
        if(symbol == ')') {
				
          while(stack[top] != '(') {
            postfix[j] = pop();
            j++;
          }
					
          pop();   //pop out (.
        } else {
          if(precedence(symbol)>precedence(stack[top])) {
            push(symbol);
          } else {
					
            while(precedence(symbol)<=precedence(stack[top])) {
              postfix[j] = pop();
              j++;
```

```
                        }

                        push(symbol);
                    }
                }
            }
        }
    }

    while(stack[top] != '#') {
        postfix[j] = pop();
        j++;
    }

    postfix[j]='\0';  //null terminate string.
}

//int stack
int stack_int[25];
int top_int = -1;

void push_int(int item) {
    stack_int[++top_int] = item;
}

char pop_int() {
    return stack_int[top_int--];
}

//evaluates postfix expression
int evaluate(char *postfix){

    char ch;
    int i = 0,operand1,operand2;

    while( (ch = postfix[i++]) != '\0') {

        if(isdigit(ch)) {
                    push_int(ch-'0');  // Push the operand
        } else {
            //Operator,pop two  operands
            operand2 = pop_int();
            operand1 = pop_int();
```

```
      
      switch(ch) {
         case '+':
            push_int(operand1+operand2);
            break;
         case '-':
            push_int(operand1-operand2);
            break;
         case '*':
            push_int(operand1*operand2);
            break;
         case '/':
            push_int(operand1/operand2);
            break;
      }
    }
  }
      
  return stack_int[top_int];
}

void main() {
  char infix[25] = "1*(2+3)",postfix[25];
  convert(infix,postfix);
      
  printf("Infix expression is: %s\n" , infix);
  printf("Postfix expression is: %s\n" , postfix);
  printf("Evaluated expression is: %d\n" , evaluate(postfix));
}
```

If we compile and run the above program, it will produce the following result −

Output

```
Infix expression is: 1*(2+3)
Postfix expression is: 123+*
Result is: 5
```

4. Recursion

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. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

What is base condition in recursion?
In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)

{

    if (n < = 1) // base case

        return 1;

    else

        return n*fact(n-1);

}
```

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

How a particular problem is solved using recursion?
The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.
Why Stack Overflow error occurs in recursion?
If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)

{

    // wrong base case (it may cause

    // stack overflow).

    if (n == 100)

        return 1;


    else

        return n*fact(n-1);

}
```

If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

What is the difference between direct and indirect recursion?
A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly. Difference between direct and indirect recursion has been illustrated in Table 1.

```
// An example of direct recursion
void directRecFun()
{
   // Some code....

   directRecFun();

   // Some code...
}

// An example of indirect recursion
void indirectRecFun1()
{
   // Some code...

   indirectRecFun2();

   // Some code...
}
void indirectRecFun2()
{
   // Some code...

   indirectRecFun1();

   // Some code...
}
```

What is difference between tailed and non-tailed recursion?
A recursive function is tail recursive when recursive call is the last thing executed by the function. Please refer tail recursion article for details.

How memory is allocated to different function calls in recursion?
When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is

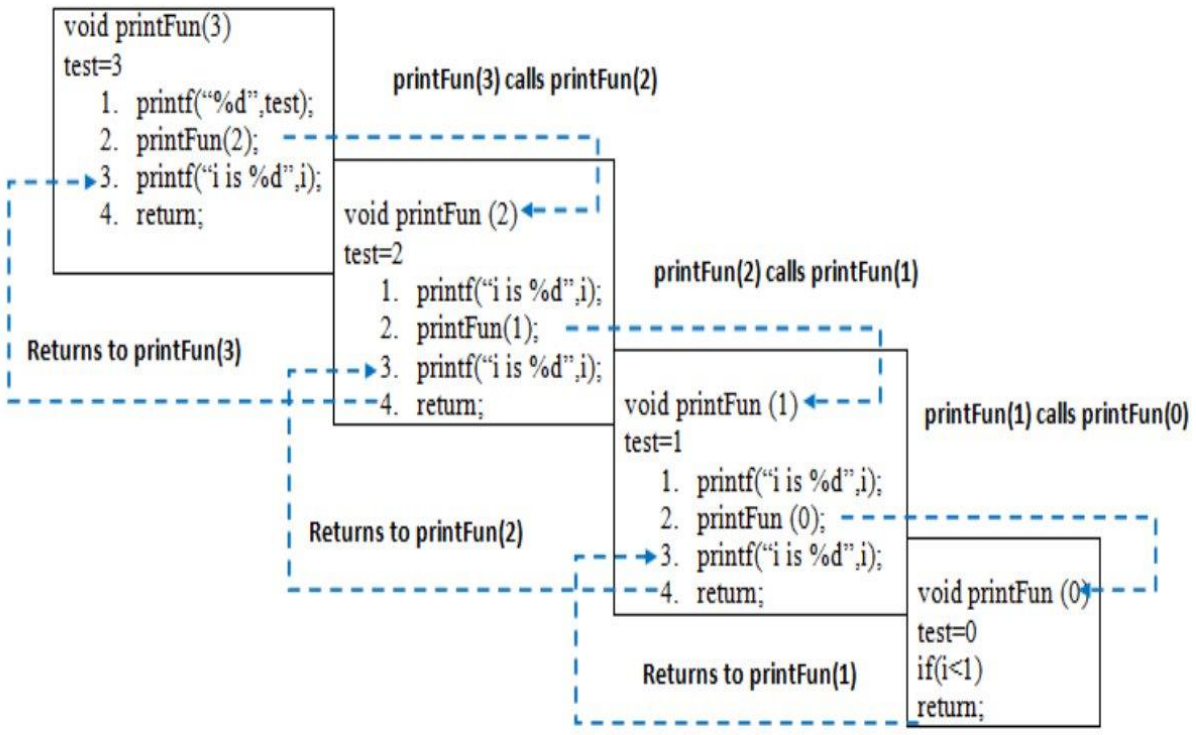
allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.
Let us take the example how recursion works by taking a simple function.

```
// A C++ program to demonstrate working of

// recursion

#include <bits/stdc++.h>

using namespace std;

void printFun(int test)

{

    if (test < 1)

        return;

    else {

        cout << test << " ";

        printFun(test - 1); // statement 2

        cout << test << " ";

        return;

    }

}

// Driver Code

int main()
```

```
{

    int test = 3;

    printFun(test);

}
```

Output :
3 2 1 1 2 3

When **printFun(3)** is called from main(), memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints '3'. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statement 1 to 4 are pushed in the stack.
Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun(0)** goes to if statement and it return to **printFun(1)**. Remaining statements of **printFun(1)** are executed and it returns to **printFun(2)** and so on. In the output, value from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.



Now, let's discuss a few practical problems which can be solved by using recursion and understand its basic working. For basic understanding please read the following articles.

Basic understanding of Recursion.
Problem 1: Write a program and recurrence relation to find the Fibonacci series of n where n>2 .
Mathematical Equation:
n if n == 0, n == 1;

fib(n) = fib(n-1) + fib(n-2) otherwise;

Recurrence Relation:

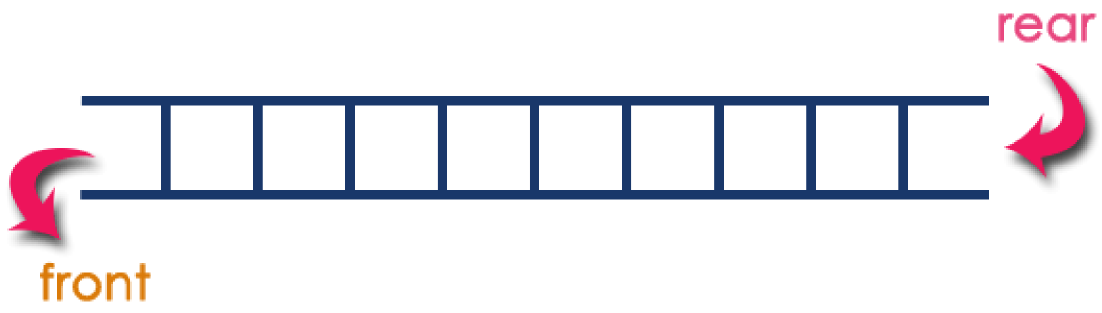
T(n) = T(n-1) + T(n-2) + O(1)

Recursive program:
Input: n = 5
Output:
Fibonacci series of 5 numbers is : 0 1 1 2 3

Queue ADT

What is a Queue?

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. The insertion is performed at one end and deletion is performed at another end. In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'. In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.



In a queue data structure, the insertion operation is performed using a function called "**enQueue()**" and deletion operation is performed using a function called "**deQueue()**".

Queue data structure can be defined as follows...

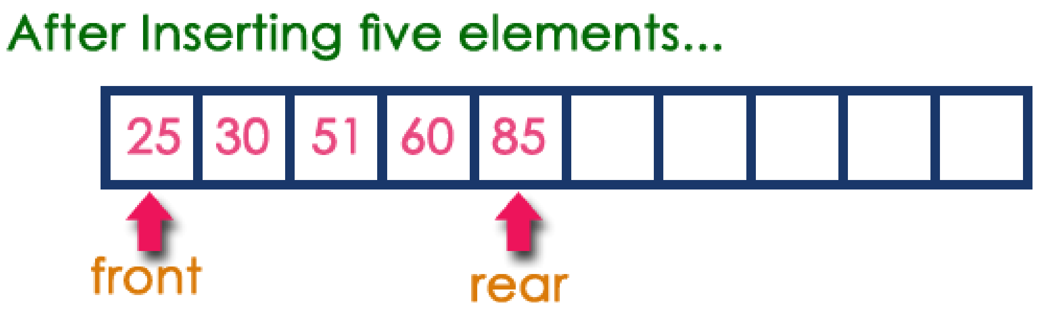
Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.

A queue data structure can also be defined as

"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle".

Example

Queue after inserting 25, 30, 51, 60 and 85.



Operations on a Queue

The following operations are performed on a queue data structure...

1. **enQueue(value) - (To insert an element into the queue)**
2. **deQueue() - (To delete an element from the queue)**
3. **display() - (To display the elements of the queue)**

Queue data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When a queue is implemented using an array, that queue can organize an only limited number of elements. When a queue is implemented using a linked list, that queue can organize an unlimited number of elements.

Queue Data Structure Using Array

A queue data structure can be implemented using a one dimensional array. The queue implemented using array stores only a fixed number of data values. The implementation of queue data structure using arrays is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear**'. Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Queue Operations using Array

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to create an empty queue.

- Step 1 - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- Step 2 - Declare all the **user defined functions** which are used in queue implementation.
- Step 3 - Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- Step 4 - Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
- Step 5 - Then implement the main method by displaying a menu of operations list and make suitable function calls to perform operations selected by the user on queue.

enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- Step 1 - Check whether **queue** is **FULL**. (**rear == SIZE-1**)
- Step 2 - If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- Step 3 - If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear]** = **value**.

deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

- Step 1 - Check whether **queue** is **EMPTY**. (**front == rear**)
- Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- Step 3 - If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as a deleted element. Then check whether both **front** and **rear** are equal (**front** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

- Step 1 - Check whether the **queue** is **EMPTY**. (**front == rear**)
- Step 2 - If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- Step 3 - If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
- Step 4 - Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

Implementation of Queue Data Structure using Array - C Programming

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10

void enQueue(int);
void deQueue();
void display();

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

void main()
{
  int value, choice;
  clrscr();
  while(1){
    printf("\n\n***** MENU *****\n");
    printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
         case 1: printf("Enter the value to be insert: ");
                 scanf("%d",&value);
                 enQueue(value);
                 break;
         case 2: deQueue();
                 break;
         case 3: display();
                 break;
         case 4: exit(0);
         default: printf("\nWrong selection!!! Try again!!!");
    }
  }
}
void enQueue(int value){
  if(rear == SIZE-1)
    printf("\nQueue is Full!!! Insertion is not possible!!!");
```

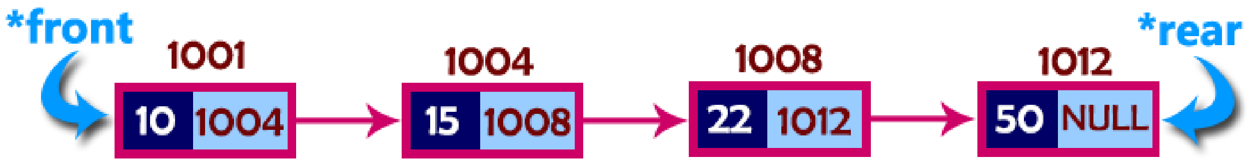
```
   else{
      if(front == -1)
              front = 0;
      rear++;
      queue[rear] = value;
      printf("\nInsertion success!!!");
   }
}
void deQueue(){
   if(front == rear)
      printf("\nQueue is Empty!!! Deletion is not possible!!!");
   else{
      printf("\nDeleted : %d", queue[front]);
      front++;
      if(front == rear)
              front = rear = -1;
   }
}
void display(){
   if(rear == -1)
      printf("\nQueue is Empty!!!");
   else{
      int i;
      printf("\nQueue elements are:\n");
      for(i=front; i<=rear; i++)
               printf("%d\t",queue[i]);
   }

   }
```

Queue Using Linked List

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, a queue using a linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked lists can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement a queue using a linked list, we need to set the following things before implementing actual operations.

- Step 1 - Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- Step 2 - Define a '**Node**' structure with two members **data** and **next**.
- Step 3 - Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- Step 4 - Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- Step 1 - Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- Step 2 - Check whether queue is **Empty** (**rear** == **NULL**)
- Step 3 - If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
- Step 4 - If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode**.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- Step 1 - Check whether **queue** is **Empty** (**front == NULL**).

- Step 2 - If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
- Step 3 - If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- Step 4 - Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- Step 1 - Check whether queue is **Empty** (**front** == **NULL**).
- Step 2 - If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- Step 4 - Display '**temp → data** --->' and move it to the next node. Repeat the same until

  '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).
- Step 5 - Finally! Display '**temp → data** ---> **NULL**'.

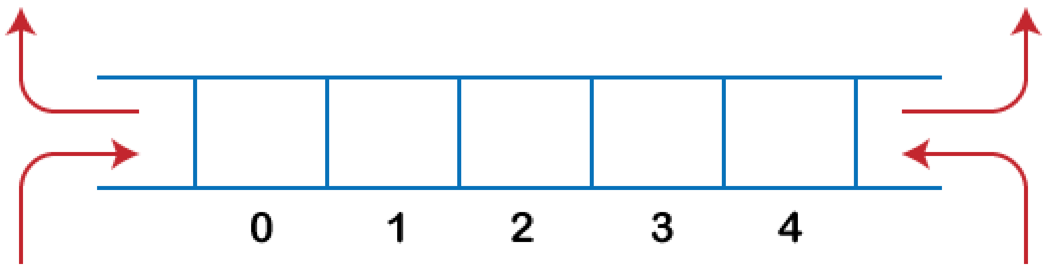
Implementation of Queue Data Structure using Linked List - C Programming

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

struct Node
{
   int data;
   struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
void delete();
void display();

void main()
{
   int choice, value;
   clrscr();
   printf("\n:: Queue Implementation using Linked List ::\n");
   while(1){
      printf("\n****** MENU ******\n");
      printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice){
          case 1: printf("Enter the value to be insert: ");
                  scanf("%d", &value);
```

```
                    insert(value);
                    break;
            case 2: delete(); break;
            case 3: display(); break;
            case 4: exit(0);
            default: printf("\nWrong selection!!! Please try again!!!\n");
      }
   }
}
void insert(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode -> next = NULL;
   if(front == NULL)
      front = rear = newNode;
   else{
      rear -> next = newNode;
      rear = newNode;
   }
   printf("\nInsertion is Success!!!\n");
}
void delete()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      front = front -> next;
      printf("\nDeleted element: %d\n", temp->data);
      free(temp);
   }
}
void display()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
      }
      printf("%d--->NULL\n",temp->data);
   }

   }
```

Deque

The dequeue stands for **Double Ended Queue**. In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the **rear end** whereas the end at which the deletion occurs is known as **front end**.

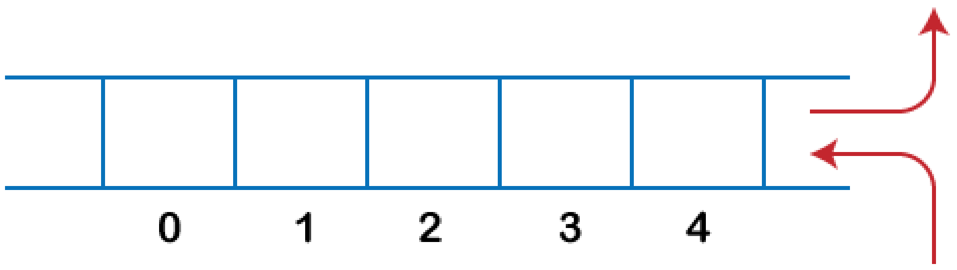


Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

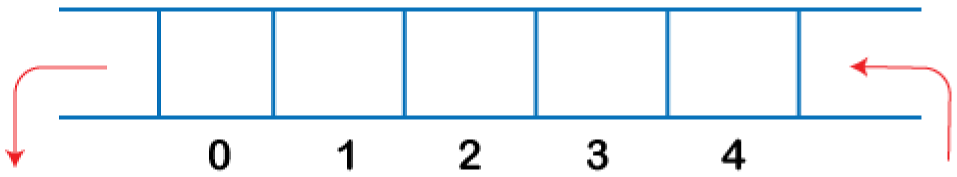
Let's look at some properties of deque.

- Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

In deque, the insertion and deletion operation can be performed from one side. The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that deque can be considered as a stack.

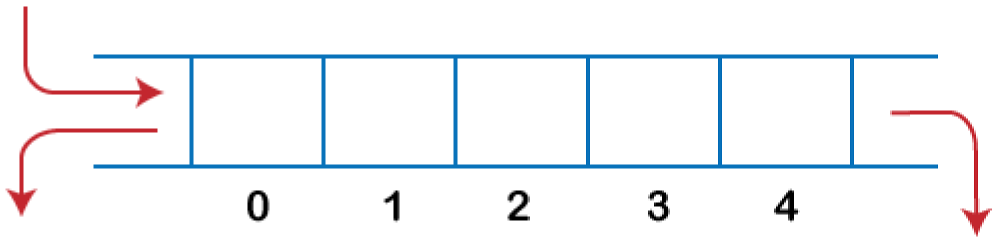


In deque, the insertion can be performed on one end, and the deletion can be done on another end. The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end. Therefore, we conclude that the deque can also be considered as the queue.

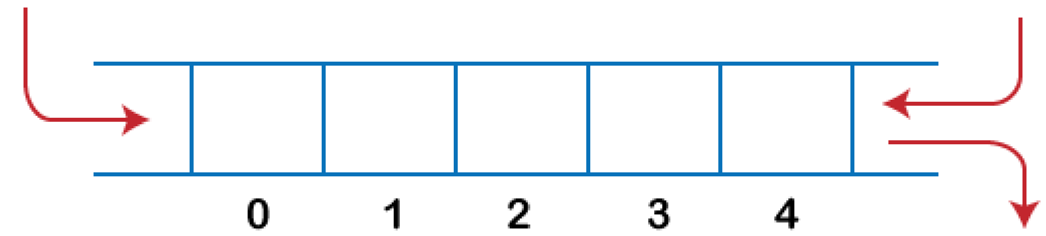


There are two types of Queues, **Input-restricted queue**, and **output-restricted queue**.

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.



2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



Operations on Deque

The following are the operations applied on deque:

- **Insert at front**

- **Delete from end**
- **insert at rear**
- **delete from rear**

Other than insertion and deletion, we can also perform **peek** operation in deque. Through **peek** operation, we can get the **front** and the **rear** element of the dequeue.

We can perform two more operations on dequeue:

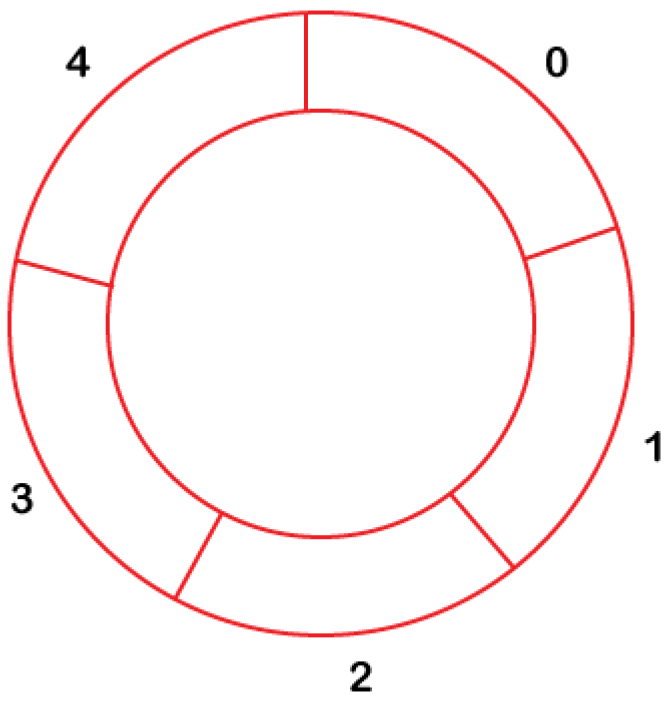
- **isFull():** This function returns a true value if the stack is full; otherwise, it returns a false value.
- **isEmpty():** This function returns a true value if the stack is empty; otherwise it returns a false value.

Memory Representation

The deque can be implemented using two data structures, i.e., **circular array**, and **doubly linked list**. To implement the deque using circular array, we first should know **what is circular array**.

What is a circular array?

An array is said to be **circular** if the last element of the array is connected to the first element of the array. Suppose the size of the array is 4, and the array is full but the first location of the array is empty. If we want to insert the array element, it will not show any overflow condition as the last element is connected to the first element. The value which we want to insert will be added in the first location of the array.



Applications of Deque

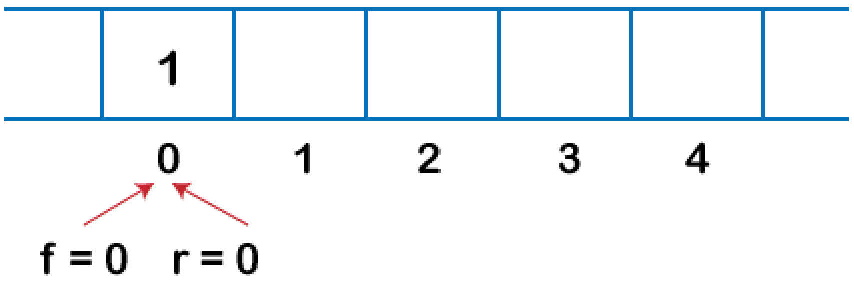
- The deque can be used as a **stack** and **queue**; therefore, it can perform both redo and undo operations.
- It can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.
- It can be used for multiprocessor scheduling. Suppose we have two processors, and each processor has one process to execute. Each processor is assigned with a process or a job, and each process contains multiple threads. Each processor maintains a deque that contains threads that are ready to execute. The processor executes a process, and if a process creates a child process then that process will be inserted at the front of the deque of the parent process. Suppose the processor P_2 has completed the execution of all its threads then it steals the thread from the rear end of the processor P_1 and adds to the front end of the processor P_2. The processor P_2 will take the thread from the front end; therefore, the deletion takes from both the ends, i.e., front and rear end. This is known as the **A-steal algorithm** for scheduling.

Implementation of Deque using a circular array

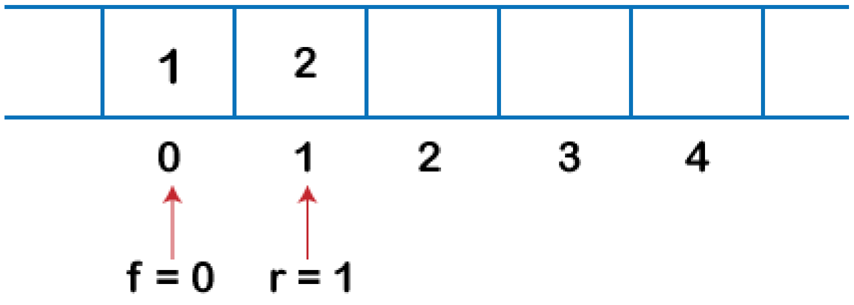
The following are the steps to perform the operations on the Deque:

Enqueue operation

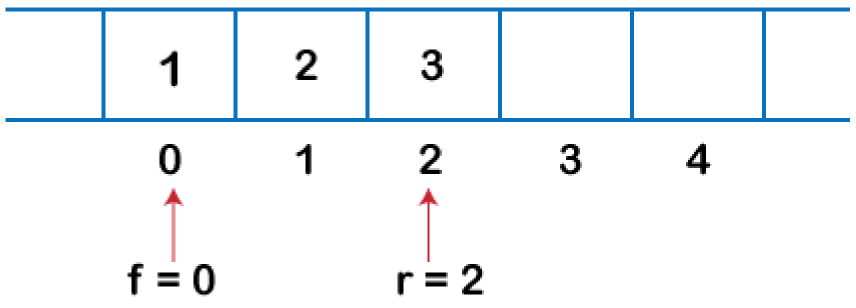
1. Initially, we are considering that the deque is empty, so both front and rear are set to -1, i.e., **f = -1** and **r = -1**.
2. As the deque is empty, so inserting an element either from the front or rear end would be the same thing. Suppose we have inserted element 1, then **front is equal to 0,** and the **rear is also equal to 0.**



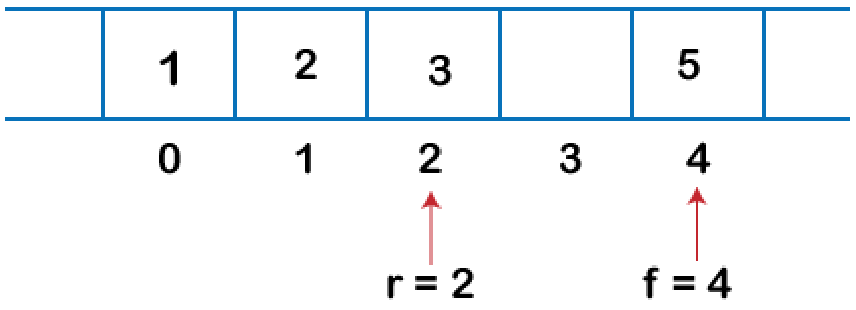
3. Suppose we want to insert the next element from the rear. To insert the element from the rear end, we first need to increment the rear, i.e., **rear=rear+1**. Now, the rear is pointing to the second element, and the front is pointing to the first element.



4. Suppose we are again inserting the element from the rear end. To insert the element, we will first increment the rear, and now rear points to the third element.



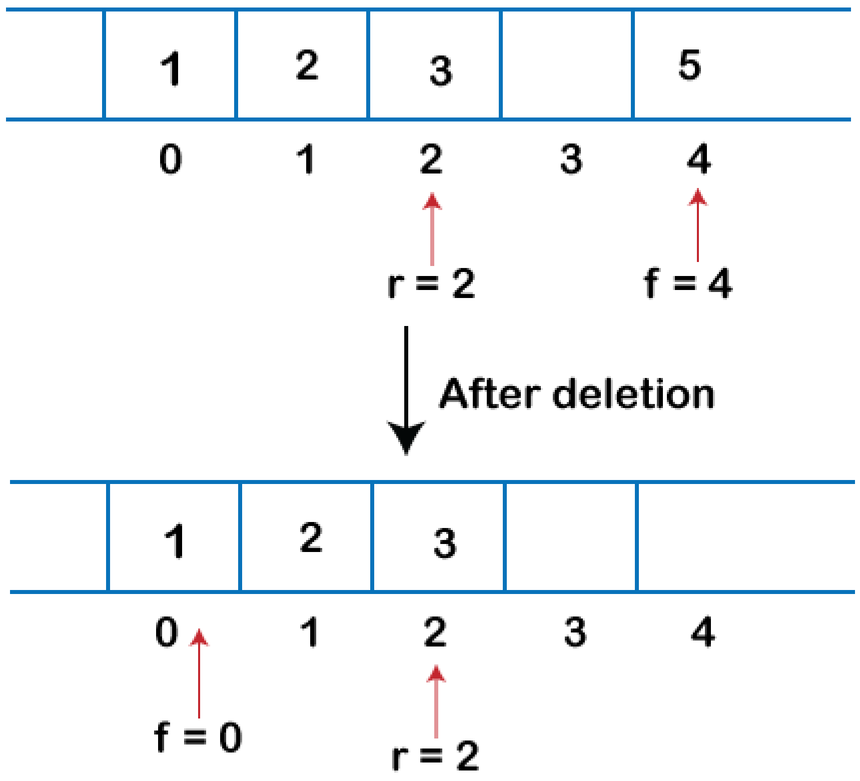
5. If we want to insert the element from the front end, and insert an element from the front, we have to decrement the value of front by 1. If we decrement the front by 1, then the front points to -1 location, which is not any valid location in an array. So, we set the front as **(n -1),** which is equal to 4 as n is 5. Once the front is set, we will insert the value as shown in the below figure:



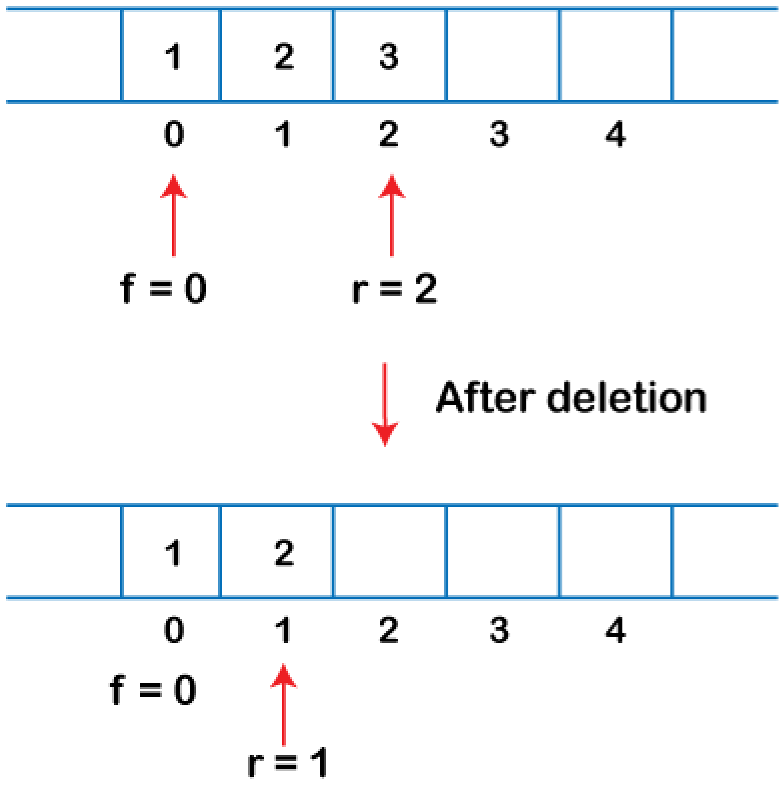
Dequeue Operation

1. If the front is pointing to the last element of the array, and we want to perform the delete operation from the front. To delete any element from the front, we need to set **front=front+1**. Currently, the value of the front is equal to 4, and if we increment the value of front, it becomes 5 which is not a valid index. Therefore, we conclude that if front points to the last

element, then front is set to 0 in case of delete operation.

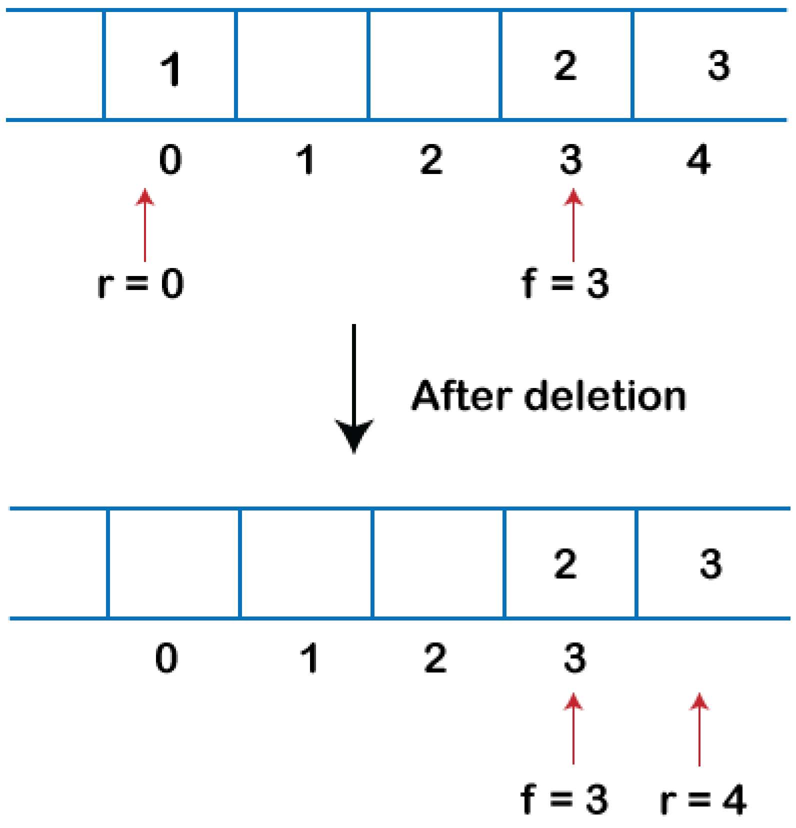


2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., **rear=rear-1** as shown in the below figure:



3. If the rear is pointing to the first element, and we want to delete the element from the rear end then we have to set **rear=n-1**

4. where **n** is the size of the array as shown in the below figure:



program of deque.

The following are the six functions that we have used in the below program:

- **enqueue_front():** It is used to insert the element from the front end.
- **enqueue_rear():** It is used to insert the element from the rear end.
- **dequeue_front():** It is used to delete the element from the front end.
- **dequeue_rear():** It is used to delete the element from the rear end.
- **getfront():** It is used to return the front element of the deque.
- **getrear():** It is used to return the rear element of the deque.

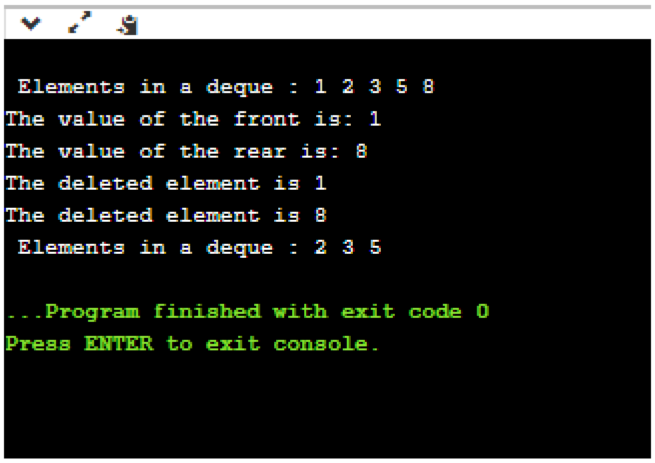
```
#define size 5
#include <stdio.h>
int deque[size];
int f=-1, r=-1;
//  enqueue_front function will insert the value from the front
void enqueue_front(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("deque is full");
    }
    else if((f==-1) && (r==-1))
    {
        f=r=0;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

```

```c
// enqueue_rear function will insert the value from the rear
void enqueue_rear(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("deque is full");
    }
    else if((f==-1) && (r==-1))
    {
        f=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("\n Elements in a deque : ");
    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 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 rear is: %d", deque[r]);
    }

}

// dequeue_front() function deletes the element from the front
void dequeue_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;
        }
    }

    // dequeue_rear() function deletes the element from the rear
    void dequeue_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()
{
   // inserting a value from the front.
    enqueue_front(2);
  // inserting a value from the front.
    enqueue_front(1);
  // inserting a value from the rear.
    enqueue_rear(3);
  // inserting a value from the rear.
    enqueue_rear(5);
  // inserting a value from the rear.
    enqueue_rear(8);
  // Calling the display function to retrieve the values of deque
    display();
 // Retrieve the front value
    getfront();
// Retrieve the rear value.
    getrear();
// deleting a value from the front
dequeue_front();
//deleting a value from the rear
dequeue_rear();
 // Calling the display function to retrieve the values of deque
 display();
    return 0;
}
```

Output:



```
 Elements in a deque : 1 2 3 5 8
The value of the front is: 1
The value of the rear is: 8
The deleted element is 1
The deleted element is 8
 Elements in a deque : 2 3 5

...Program finished with exit code 0
Press ENTER to exit console.
```

What is a 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. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

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 is 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.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

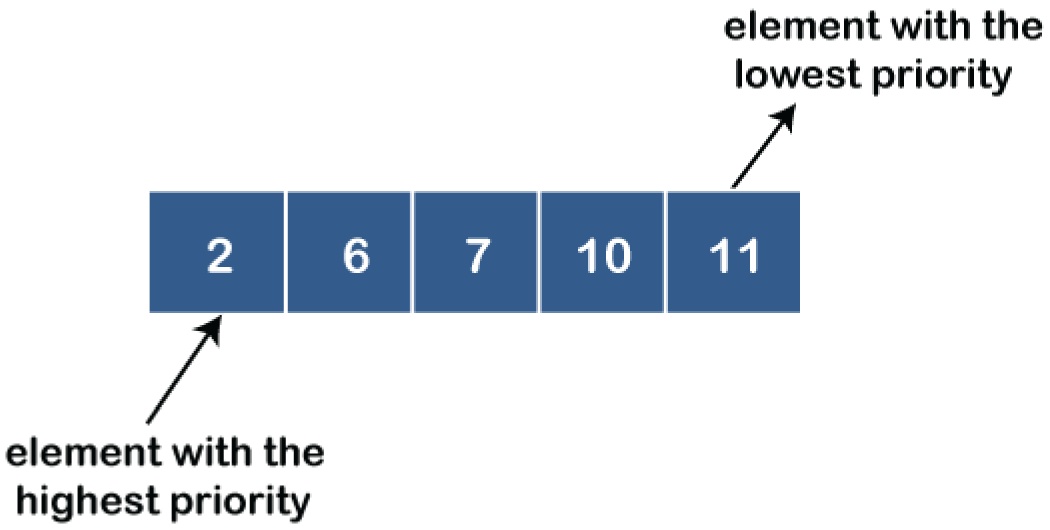
All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

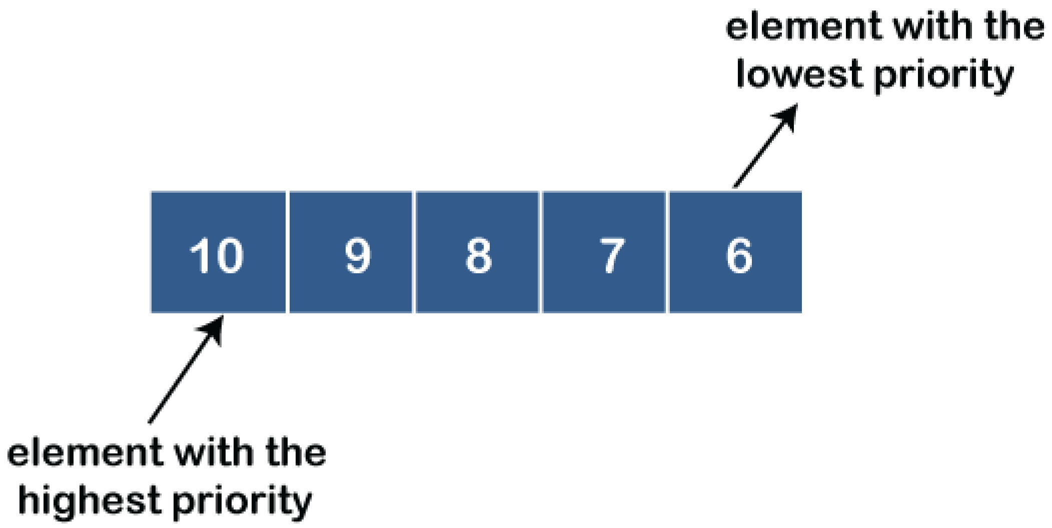
Types of Priority Queue

There are two types of priority queue:

- **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.


element with the lowest priority
2
6
7
10
11
element with the highest priority

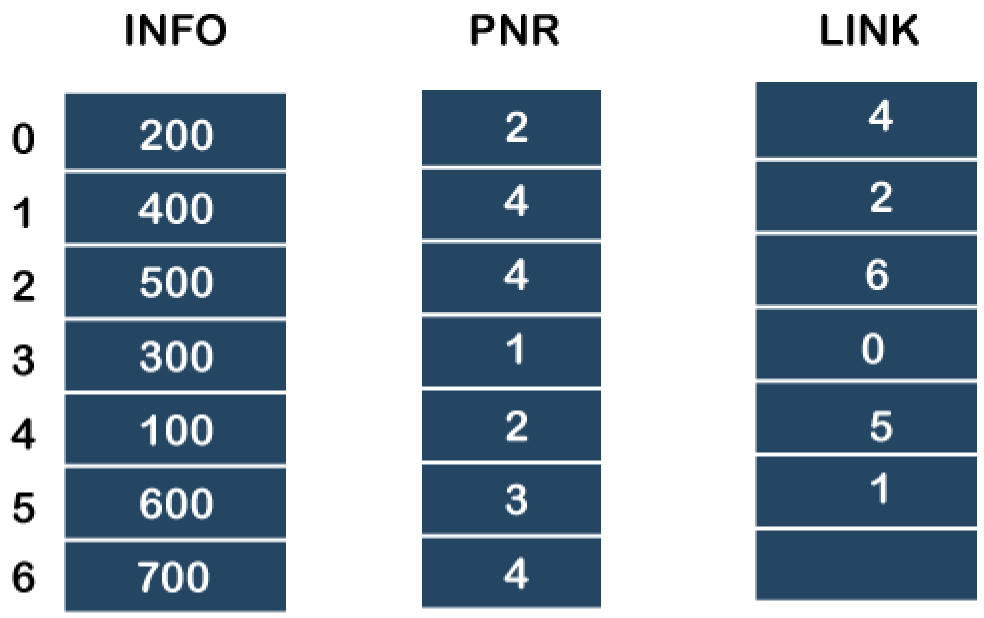
- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest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


element with the lowest priority
10
9
8
7
6
element with the highest priority

Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and LINK basically contains the address of the next node.



	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

Let's create the priority queue step by step.

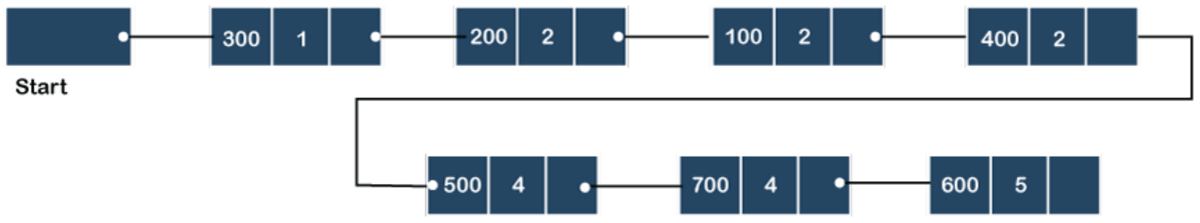
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

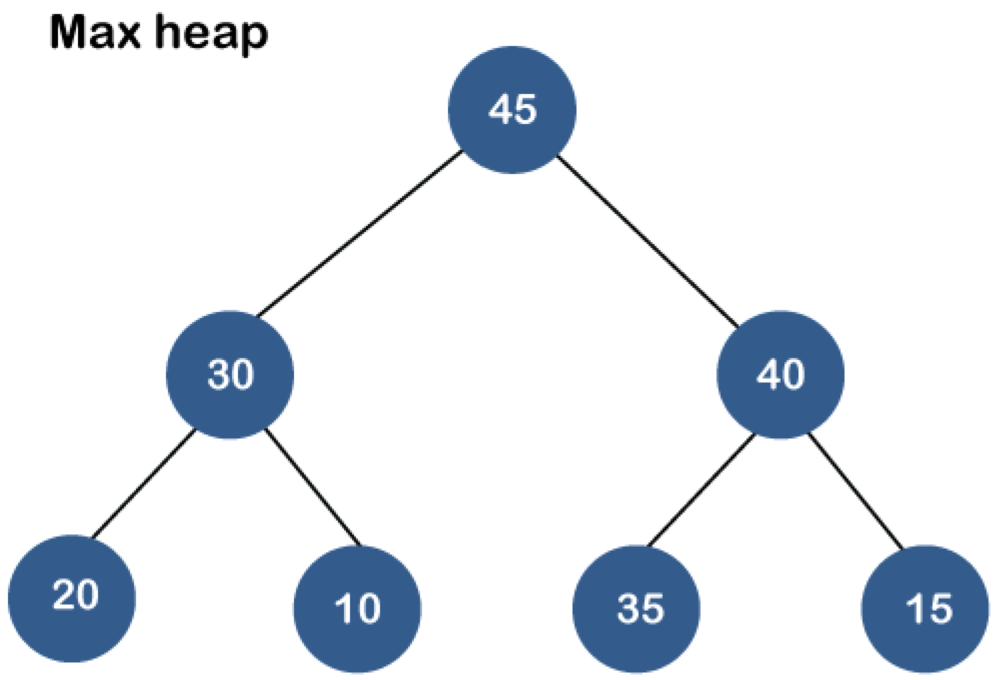
Analysis of complexities using different implementations

Implementation	add	Remove	peek
Linked list	O(1)	O(n)	O(n)
Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

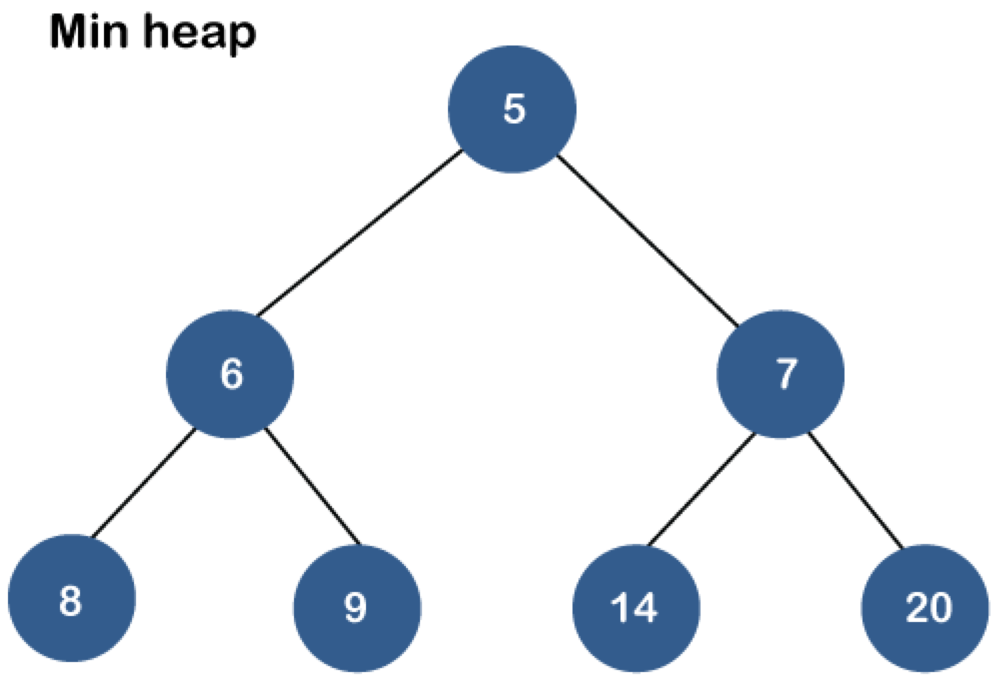
What is Heap?

A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.



- **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.



Both the heaps are the binary heap, as each has exactly two child nodes.

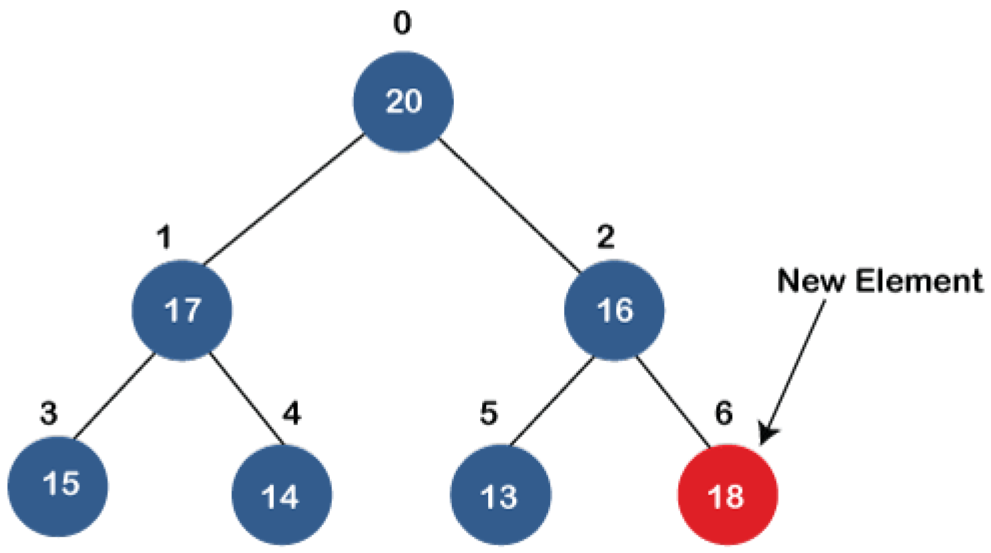
Priority Queue Operations

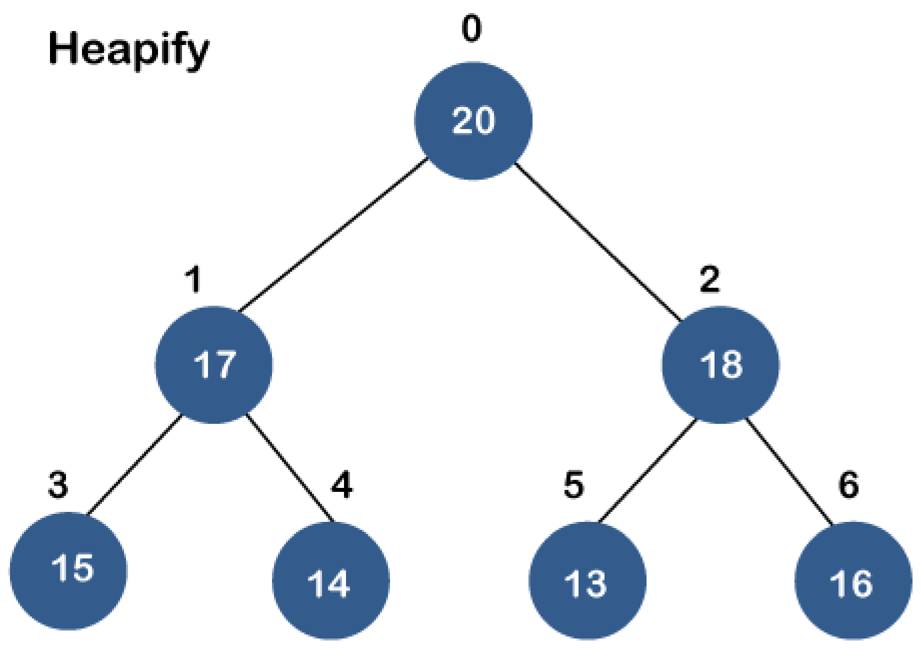
The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

- **Inserting the element in a priority queue (max heap)**

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.





- **Removing the minimum element from the priority queue**

As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

Program to create the priority queue using the binary max heap.

```c
#include <stdio.h>
#include <stdio.h>
int heap[40];
int size=-1;

// retrieving the parent node of the child node
int parent(int i)
{

   return (i - 1) / 2;
}

// retrieving the left child of the parent node.
int left_child(int i)
{
 return i+1;
}
// retrieving the right child of the parent
int right_child(int i)
{
 return i+2;
}
// Returning the element having the highest priority
int get_Max()
{
   return heap[0];
}
//Returning the element having the minimum priority
int get_Min()
```

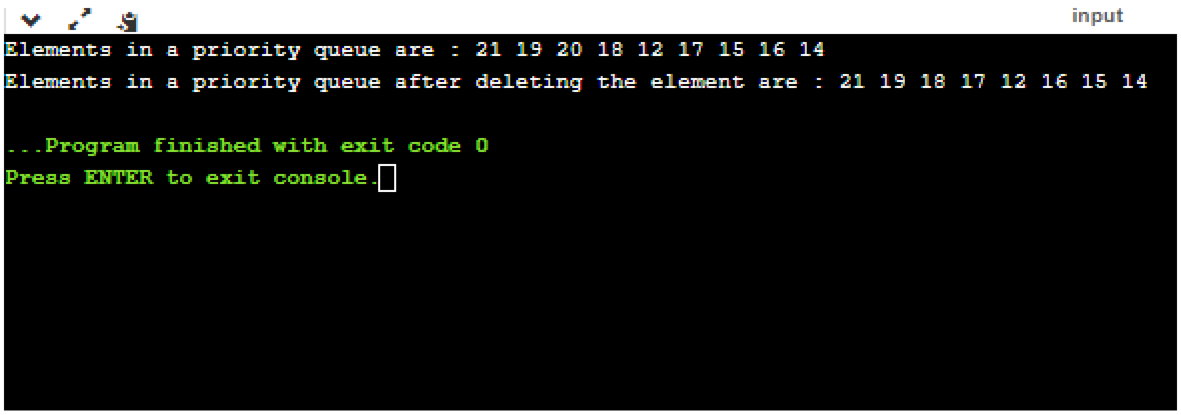
```
{
    return heap[size];
}
// function to move the node up the tree in order to restore the heap property.
void moveUp(int i)
{
    while (i > 0)
    {
        // swapping parent node with a child node
        if(heap[parent(i)] < heap[i]) {

        int temp;
        temp=heap[parent(i)];
        heap[parent(i)]=heap[i];
        heap[i]=temp;


    }
       // updating the value of i to i/2
       i=i/2;
    }
}

//function to move the node down the tree in order to restore the heap property.
void moveDown(int k)
{
    int index = k;

    // getting the location of the Left Child
    int left = left_child(k);

    if (left <= size && heap[left] > heap[index]) {
        index = left;
    }

    // getting the location of the Right Child
    int right = right_child(k);

    if (right <= size && heap[right] > heap[index]) {
        index = right;
    }

    // If k is not equal to index
    if (k != index) {
      int temp;
      temp=heap[index];
      heap[index]=heap[k];
      heap[k]=temp;
        moveDown(index);
    }
}

// Removing the element of maximum priority
void removeMax()
{
    int r= heap[0];
    heap[0]=heap[size];
    size=size-1;
    moveDown(0);
}
//inserting the element in a priority queue
void insert(int p)
{
    size = size + 1;
    heap[size] = p;

    // move Up to maintain heap property
    moveUp(size);
}

        //Removing the element from the priority queue at a given index i.
        void delete(int i)
        {
            heap[i] = heap[0] + 1;

           // move the node stored at ith location is shifted to the root node
            moveUp(i);

            // Removing the node having maximum priority
            removeMax();
        }
        int main()
        {
            // Inserting the elements in a priority queue

            insert(20);
            insert(19);
```

```
      insert(21);
      insert(18);
      insert(12);
      insert(17);
      insert(15);
      insert(16);
      insert(14);
      int i=0;

    printf("Elements in a priority queue are : ");
    for(int i=0;i<=size;i++)
      {
        printf("%d ",heap[i]);
      }
      delete(2); // deleting the element whose index is 2.
      printf("\nElements in a priority queue after deleting the element are : ");
      for(int i=0;i<=size;i++)
      {
        printf("%d ",heap[i]);
      }
  int max=get_Max();
      printf("\nThe element which is having the highest priority is %d: ",max);


      int min=get_Min();
         printf("\nThe element which is having the minimum priority is : %d",min);
      return 0;
  }
```

In the above program, we have created the following functions:

- **int parent(int i):** This function returns the index of the parent node of a child node, i.e., i.
- **int left_child(int i):** This function returns the index of the left child of a given index, i.e., i.
- **int right_child(int i):** This function returns the index of the right child of a given index, i.e., i.
- **void moveUp(int i):** This function will keep moving the node up the tree until the heap property is restored.
- **void moveDown(int i):** This function will keep moving the node down the tree until the heap property is restored.
- **void removeMax():** This function removes the element which is having the highest priority.
- **void insert(int p):** It inserts the element in a priority queue which is passed as an argument in a function**.**
- **void delete(int i):** It deletes the element from a priority queue at a given index.
- **int get_Max():** It returns the element which is having the highest priority, and we know that in max heap, the root node contains the element which has the largest value, and highest priority.
- **int get_Min():** It returns the element which is having the minimum priority, and we know that in max heap, the last node contains the element which has the smallest value, and lowest priority.

Output



```
Elements in a priority queue are : 21 19 20 18 12 17 15 16 14
Elements in a priority queue after deleting the element are : 21 19 18 17 12 16 15 14

...Program finished with exit code 0
Press ENTER to exit console.
```

Single Linked List

What is Linked List?

When we want to work with an unknown number of data values, we use a linked list data structure to organize that data. The linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node".

What is Single Linked List?

Simply a list is a sequence of data, and the linked list is a sequence of data linked with each other.

The formal definition of a single linked list is as follows...

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.

The graphical representation of a node in a single linked list is as follows...

Importent Points to be Remembered

In a single linked list, the address of the first node is always stored in a reference node known as "front" (Some times it is also known as "head").

Always next part (reference part) of the last node must be NULL.

Example

Operations on Single Linked List

The following operations are performed on a Single Linked List

- **Insertion**
- **Deletion**
- **Display**

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program.
- **Step 2 -** Declare all the **user defined functions**.

- **Step 3 -** Define a **Node** structure with two members **data** and **next**
- **Step 4 -** Define a Node pointer **'head'** and set it to **NULL**.
- **Step 5 -** Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Set **temp → next** = **newNode**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp → next** == **NULL**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)

- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1 → next** == **NULL**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)
- **Step 7 -** Finally, Set **temp2 → next** = **NULL** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next** == NULL).
- **Step 11 -** If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
- **Step 5 -** Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Circular Linked List

What is Circular Linked List?

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list

Example

Operations

In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program.
- **Step 2 -** Declare all the **user defined** functions.

- **Step 3 -** Define a **Node** structure with two members **data** and **next**
- **Step 4 -** Define a Node pointer '**head**' and set it to **NULL**.
- **Step 5 -** Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .
- **Step 4 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
- **Step 5 -** Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** == **head**').
- **Step 6 -** Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.

Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** == **head**).
- **Step 6 -** Set **temp → next** = **newNode** and **newNode → next** = **head**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8 -** If **temp** is last node then set **temp → next** = **newNode** and **newNode → next** = **head**.
- **Step 8 -** If **temp** is not last node then set **newNode → next** = **temp → next** and **temp → next** = **newNode**.

Deletion

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
- **Step 4 -** Check whether list is having only one node (**temp1 → next** == **head**)

- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** == **head**)
- **Step 7 -** Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

Deleting from End of the list

We can use the following steps to delete a node from end of the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1 → next** == **head**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next** == **head**)
- **Step 7 -** Set **temp2 → next** = **head** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)
- **Step 7 -** If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).

- **Step 8 -** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → next** = **head** and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- **Step 1 1-** If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
- **Step 5 -** Finally display **temp → data** with arrow pointing to **head → data**.

Double Linked List

What is Double Linked List?

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we can not traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure...

Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example

Importent Points to be Remembered

In double linked list, the first node must be always pointed by head.

Always the previous field of the first node must be NULL.

Always the next field of the last node must be NULL.

Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → previous** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.

- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to both **newNode → previous** & **newNode → next** and set **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5 -** Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7 -** Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp → previous** is equal to **temp → next**)
- **Step 5 -** If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)
- **Step 5 -** If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list. (until **temp → next** is equal to **NULL**)
- **Step 7 -** Assign **NULL** to **temp → previous → next** and delete **temp**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.
- **Step 5 -** If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the fuction.

- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node which is to be deleted then set **head** to **NULL** and delete **temp** (**free(temp)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp** is the first node in the list (**temp == head**).
- **Step 9 -** If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.
- **Step 10 -** If **temp** is not the first node, then check whether it is the last node in the list (**temp → next == NULL**).
- **Step 11 -** If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp)**).
- **Step 12 -** If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next** (**temp → previous → next = temp → next**), **temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).

Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Display **'NULL <--- '**.
- **Step 5 -** Keep displaying **temp → data** with an arrow (**<===>**) until **temp** reaches to the last node
- **Step 6 -** Finally, display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Sparse Matrix

What is Sparse Matrix?

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represent a m X n matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

Sparse matrix is a matrix which contains very few non-zero elements.

When a sparse matrix is represented with a 2-dimensional array, we waste a lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of the matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times. To make it simple we use the following sparse matrix representation.

Sparse Matrix Representations

A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation (Array Representation)
2. Linked Representation

Triplet Representation (Array Representation)

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

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

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

Implementation of Array Representation of Sparse Matrix using C++

```
#include<iostream>

using namespace std;

int main()
{
        // sparse matrix of class 5x6 with 6 non-zero values
        int sparseMatrix[5][6] =
        {
        {0 , 0 , 0 , 0 , 9, 0 },
        {0 , 8 , 0 , 0 , 0, 0 },
        {4 , 0 , 0 , 2 , 0, 0 },
        {0 , 0 , 0 , 0 , 0, 5 },
        {0 , 0 , 2 , 0 , 0, 0 }
        };

        // Finding total non-zero values in the sparse matrix
        int size = 0;
        for (int row = 0; row < 5; row++)
        for (int column = 0; column < 6; column++)
        if (sparseMatrix[row][column] != 0)
                size++;

        // Defining result Matrix
        int resultMatrix[3][size];

        // Generating result matrix
        int k = 0;
        for (int row = 0; row < 5; row++)
        for (int column = 0; column < 6; column++)
        if (sparseMatrix[row][column] != 0)
        {
                resultMatrix[0][k] = row;
                resultMatrix[1][k] = column;
                resultMatrix[2][k] = sparseMatrix[row][column];
                k++;
        }

        // Displaying result matrix
        cout<<"Triplet Representation : "<<endl;
        for (int row=0; row<3; row++)
        {
        for (int column = 0; column<size; column++)
        cout<<resultMatrix[row][column]<<" ";

        cout<<endl;
        }
```

```
        return 0;
}
```

Linked Representation

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

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

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

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

Tree - Terminology

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications. A tree data structure can be defined as follows...

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

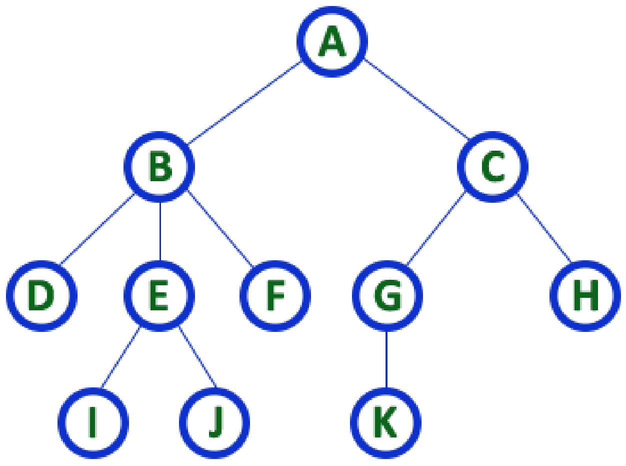
A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively

In tree data structure, every individual element is called as Node. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.

Example



TREE with 11 nodes and 10 edges

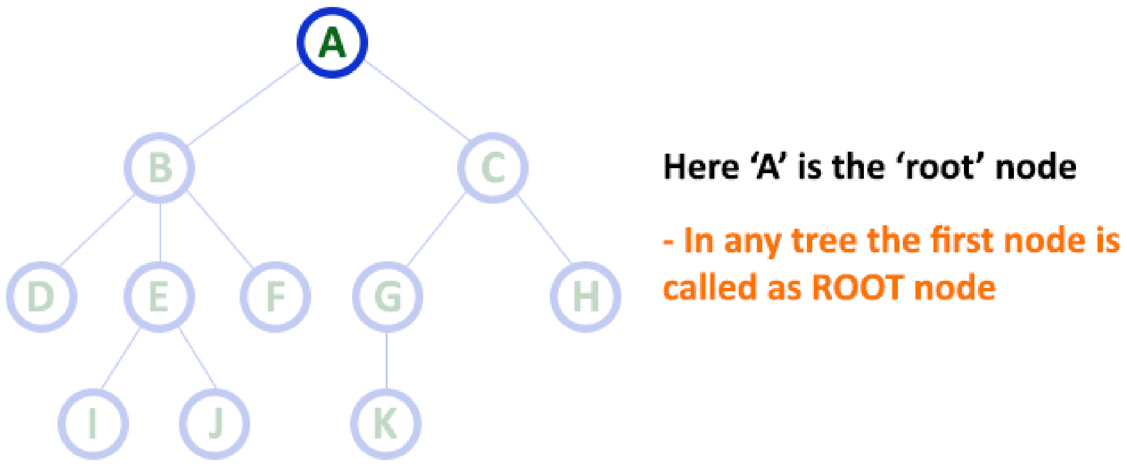
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Terminology

In a tree data structure, we use the following terminology...

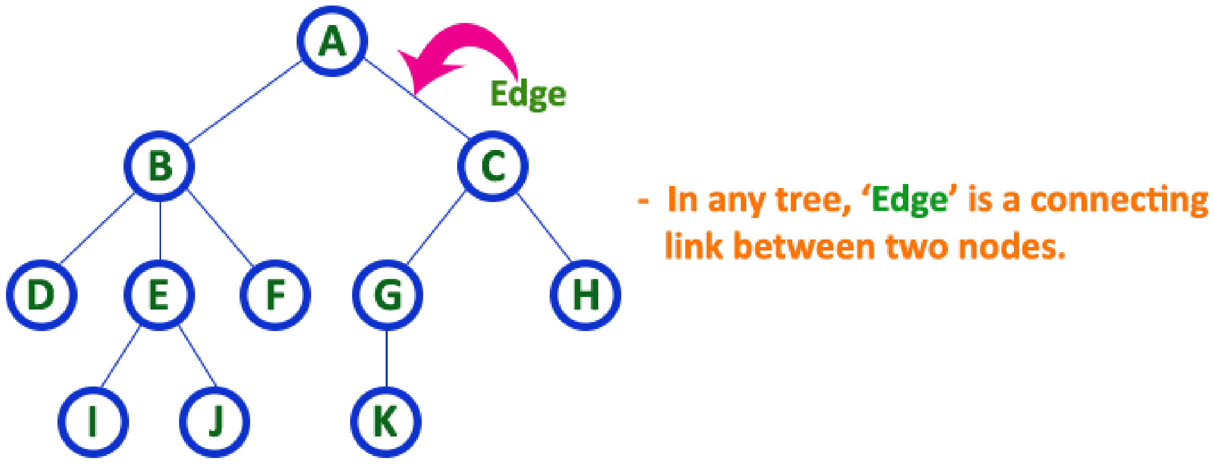
1. Root

In a tree data structure, the first node is called as Root Node. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.


A
B
C
D
E
F
G
H
I
J
K
Here 'A' is the 'root' node
- In any tree the first node is called as ROOT node

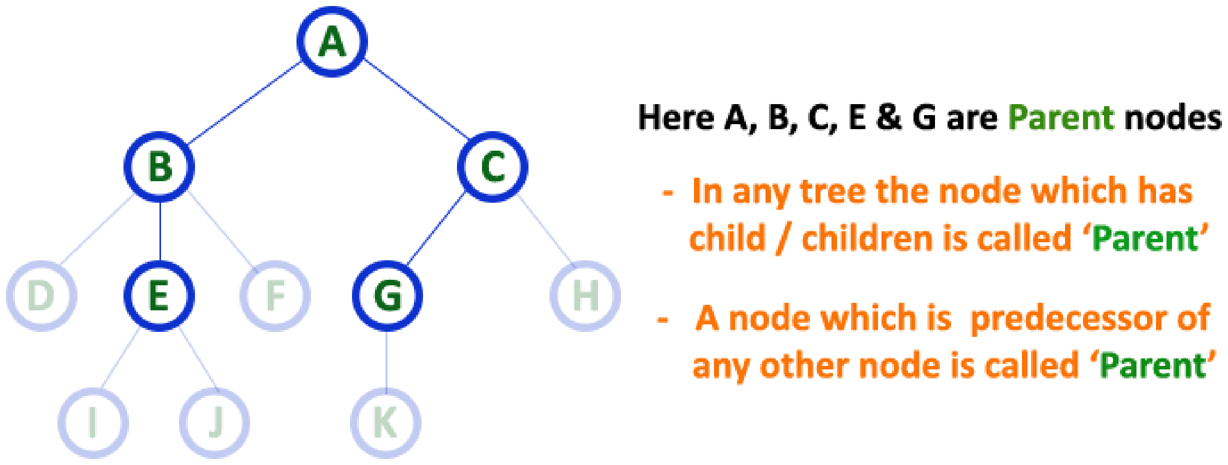
2. Edge

In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.


A
Edge
B
C
D
E
F
G
H
I
J
K
- In any tree, 'Edge' is a connecting link between two nodes.

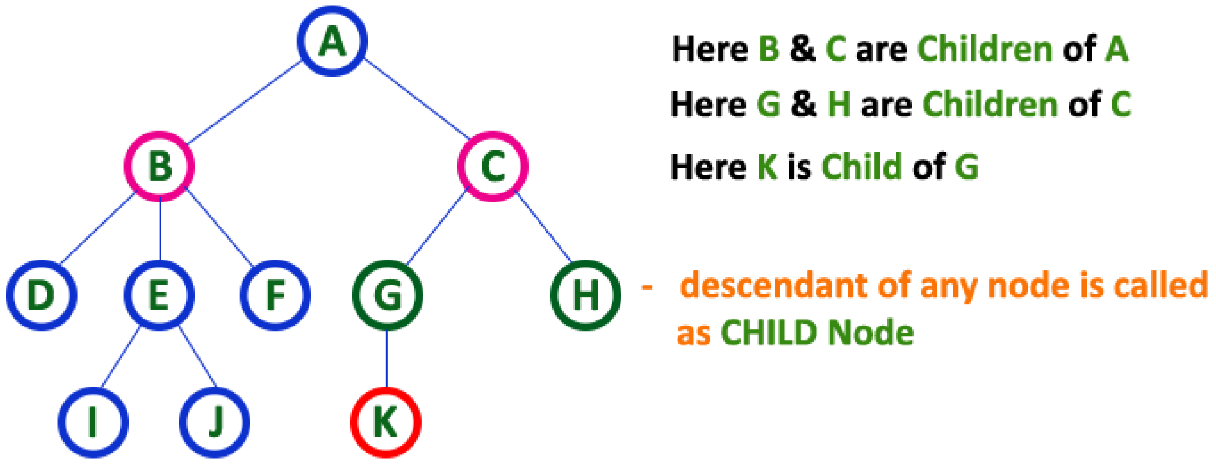
3. Parent

In a tree data structure, the node which is a predecessor of any node is called as PARENT NODE. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "The node which has child / children".


A
B
C
D
E
F
G
H
I
J
K
Here A, B, C, E & G are Parent nodes
- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

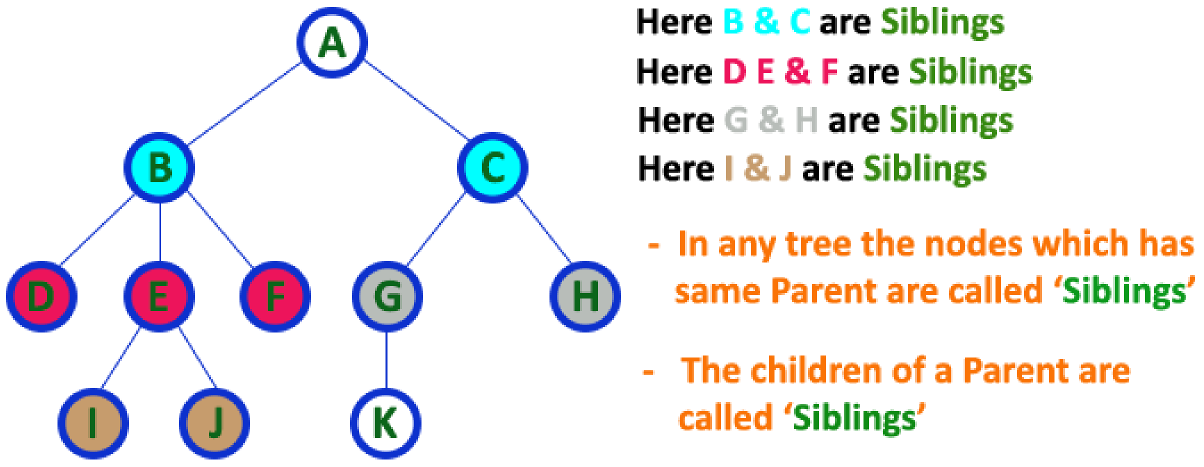
4. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.


A
B
C
D
E
F
G
H
I
J
K
Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G
- descendant of any node is called as CHILD Node

5. Siblings

In a tree data structure, nodes which belong to the same Parent are called SIBLINGS. In simple words, the nodes with the same parent are called Sibling nodes.



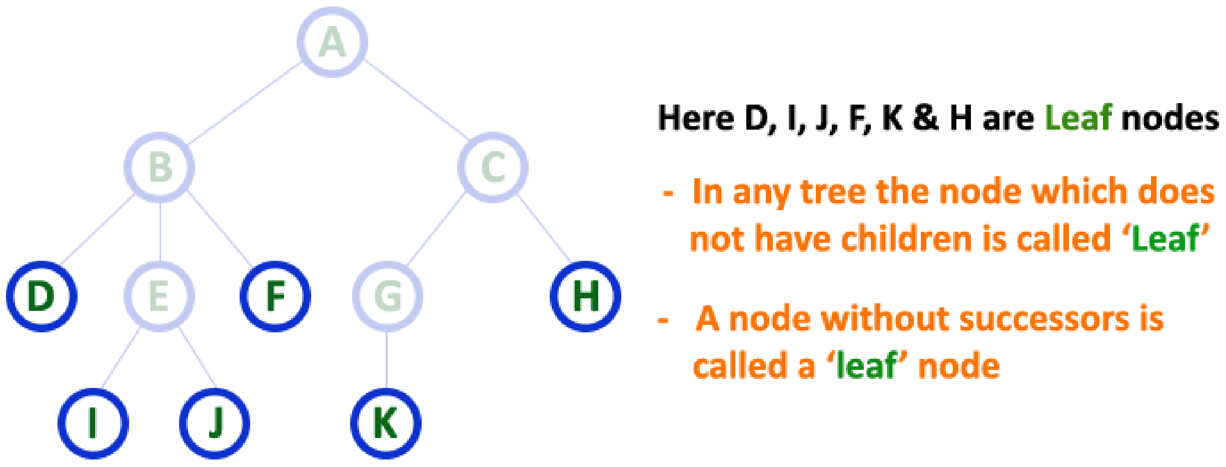
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called External Nodes. External node is also a node with no child. In a tree, <u>a leaf node is also called a 'Terminal' node.</u>



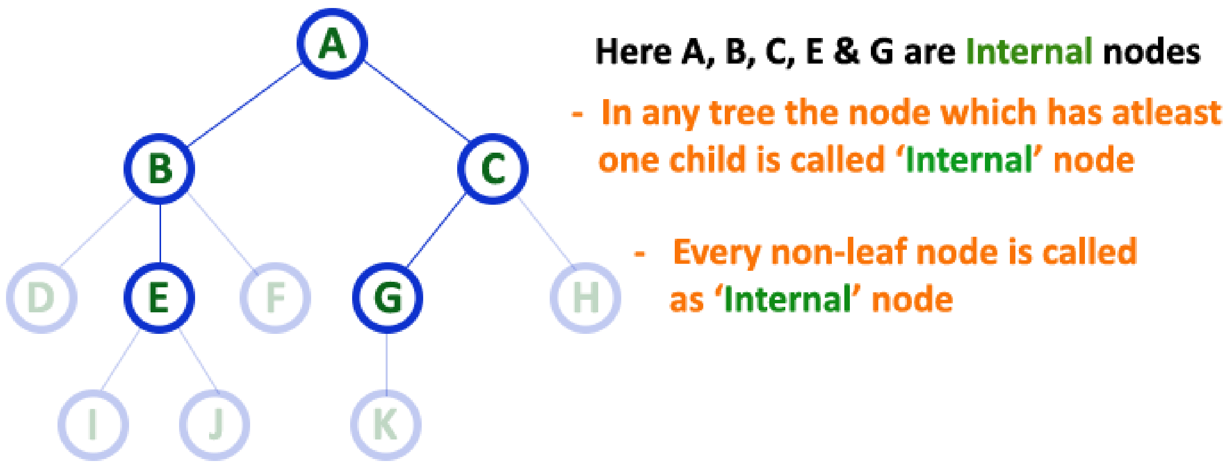
Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes

In a tree data structure, the node which has at least one child is called an INTERNAL Node. In simple words, an internal node is a node with at least one child.

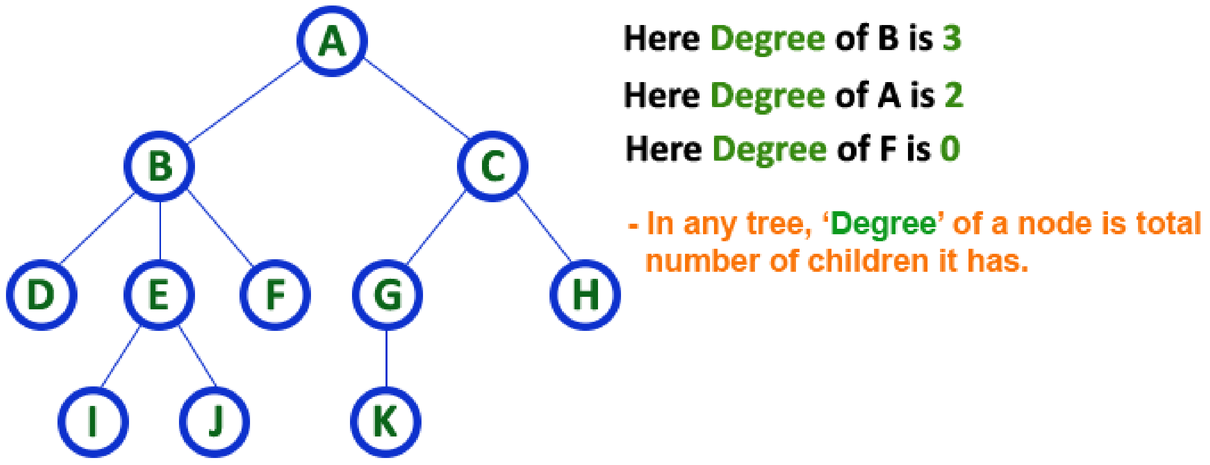
In a tree data structure, nodes other than leaf nodes are called Internal Nodes. The root node is also said to be an Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.


A
B
C
D
E
F
G
H
I
J
K
Here A, B, C, E & G are Internal nodes
- In any tree the node which has atleast one child is called 'Internal' node
- Every non-leaf node is called as 'Internal' node

---

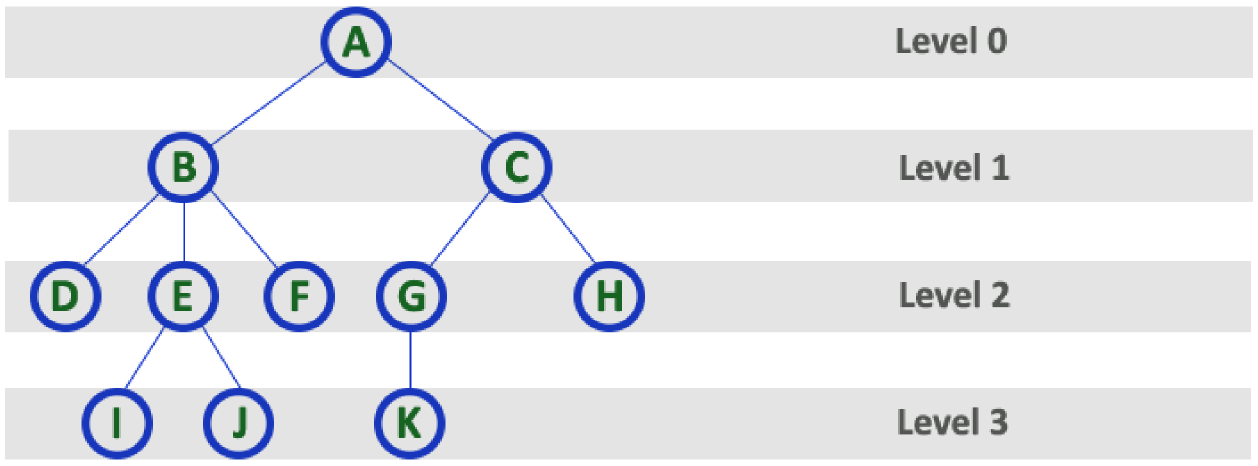
8. Degree

In a tree data structure, the total number of children of a node is called the DEGREE of that Node. In simple words, the Degree of a node is the total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'



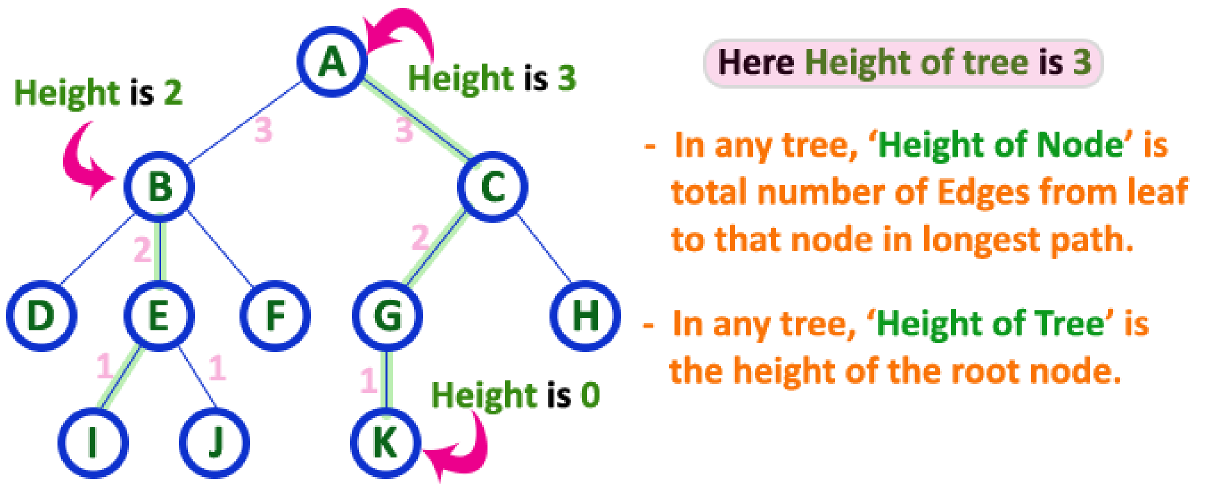
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called a Level and the Level count starts with '0' and incremented by one at each level (Step).



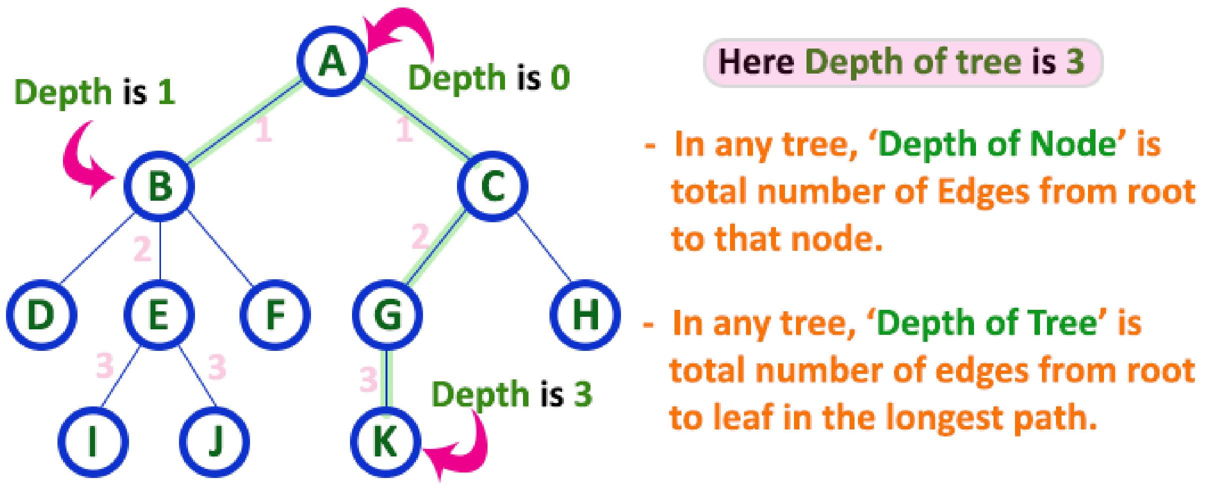
10. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.


Height is 2
Height is 3
Height is 0
A
B
C
D
E
F
G
H
I
J
K
3
3
2
2
1
1
1
Here Height of tree is 3
- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

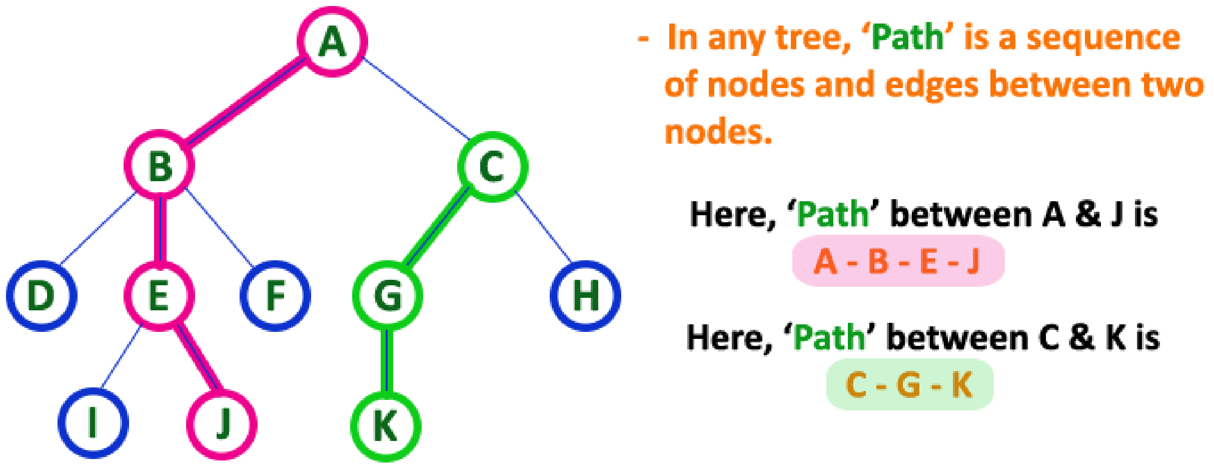
11. Depth

In a tree data structure, the total number of edges from root node to a particular node is called DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be the depth of that tree. In a tree, the depth of the root node is '0'.


Depth is 1
Depth is 0
Depth is 3
A
B
C
D
E
F
G
H
I
J
K
1
1
2
2
3
3
3
Here Depth of tree is 3
- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

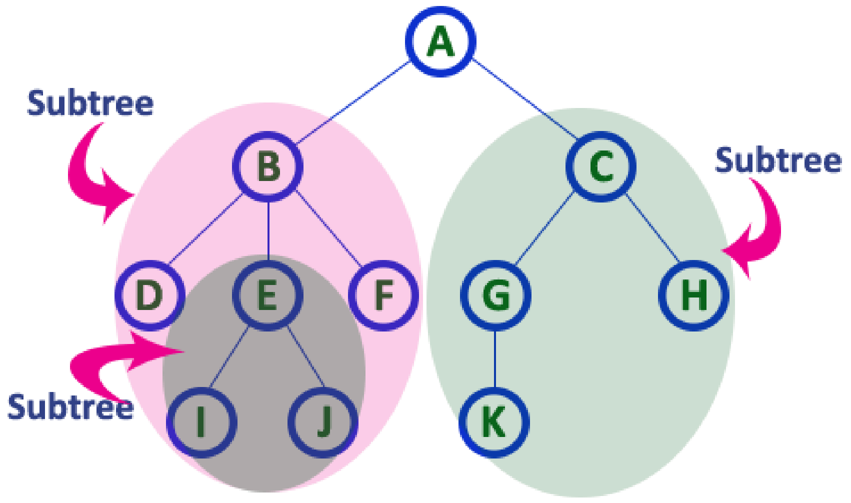
12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called PATH between those two Nodes. Length of a Path is the total number of nodes in that path. In below example the path A - B - E - J has length 4.


A
B
C
D
E
F
G
H
I
J
K
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.
Here, 'Path' between A & J is
A - B - E - J
Here, 'Path' between C & K is
C - G - K

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

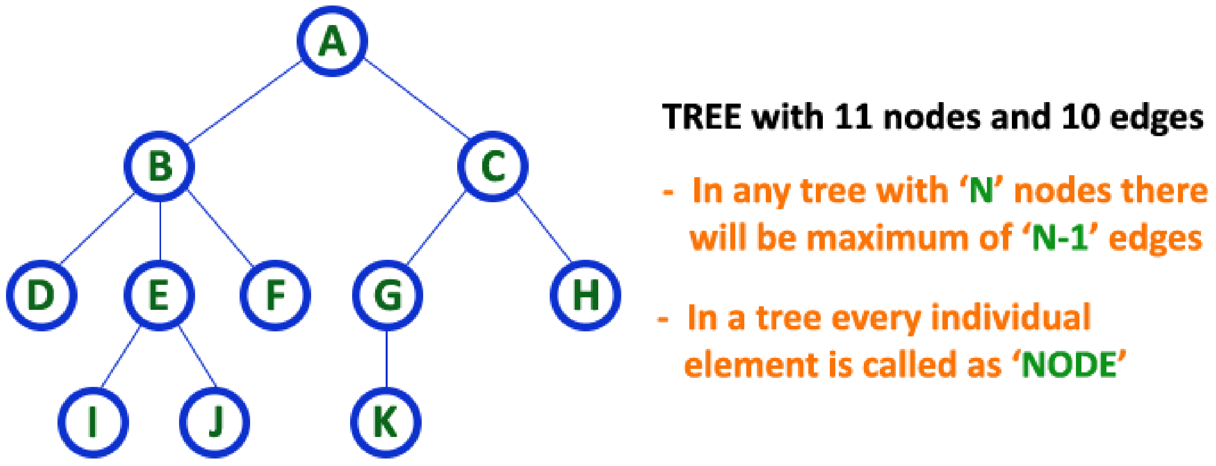


Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows...

1. **List Representation**
2. **Left Child - Right Sibling Representation**

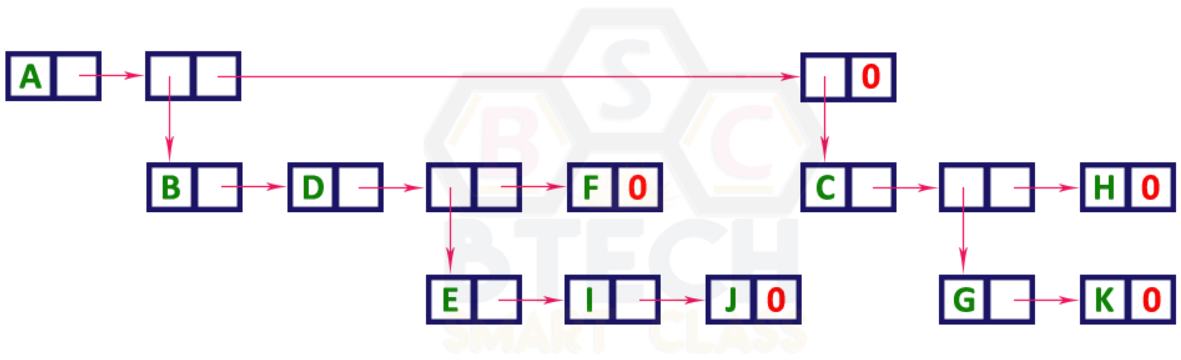
Consider the following tree...


A
B
C
D
E
F
G
H
I
J
K
TREE with 11 nodes and 10 edges
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

1. List Representation

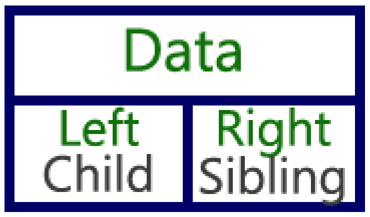
In this representation, we use two types of nodes one for representing the node with data called 'data node' and another for representing only references called 'reference node'. We start with a 'data node' from the root node in the tree. Then it is linked to an internal node through a 'reference node' which is further linked to any other node directly. This process repeats for all the nodes in the tree.

The above example tree can be represented using List representation as follows...



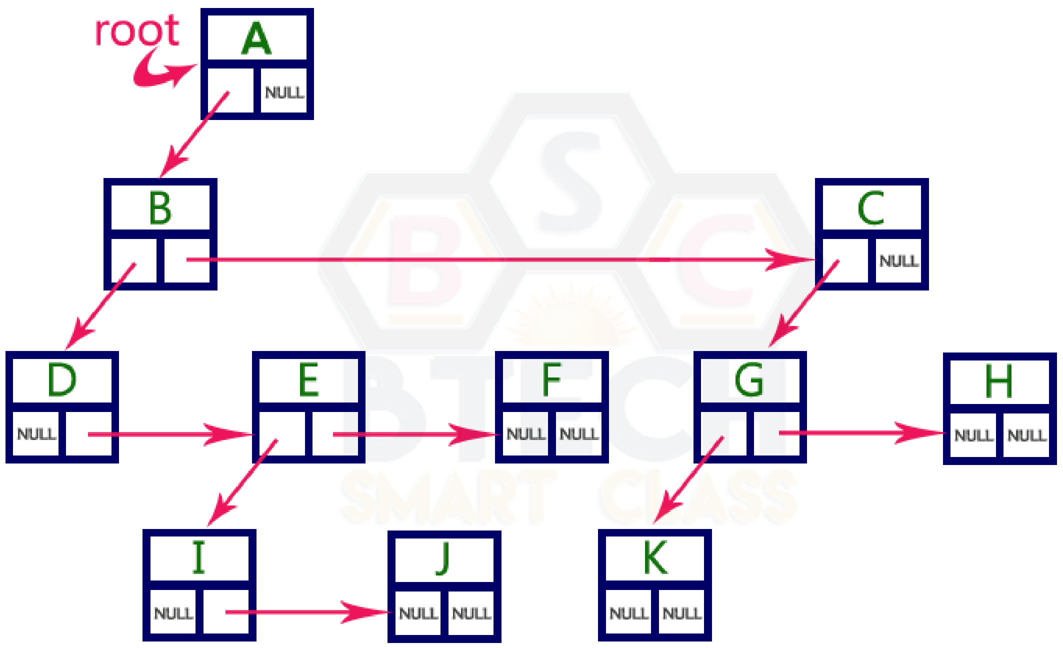
2. Left Child - Right Sibling Representation

In this representation, we use a list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...



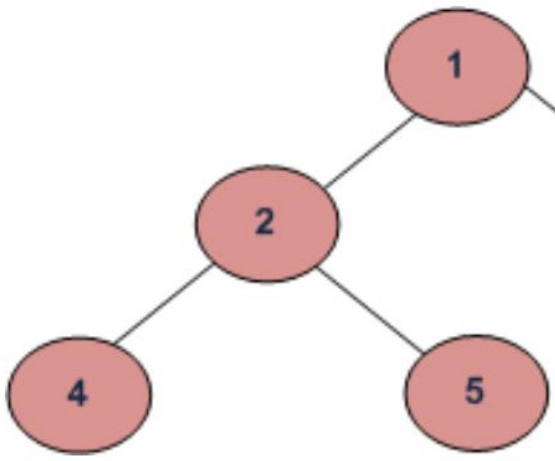
In this representation, every node's data field stores the actual value of that node. If that node has left a child, then left reference field stores the address of that left child node otherwise stores NULL. If that node has the right sibling, then right reference field stores the address of right sibling node otherwise stores NULL.

The above example tree can be represented using Left Child - Right Sibling representation as follows...



Tree Traversals (Inorder, Preorder and Postorder)

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. Following are the generally used ways for traversing trees.



Depth First Traversals:

(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5

Please see this post for Breadth First Traversal.

Inorder Traversal (Practice):

Algorithm Inorder(tree)

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)

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.

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

Preorder Traversal (Practice):

Algorithm Preorder(tree)

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)

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree. Please see http://en.wikipedia.org/wiki/Polish_notation to know why prefix expressions are useful.

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

Postorder Traversal (Practice):

Algorithm Postorder(tree)

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.

Uses of Postorder

Postorder traversal is used to delete the tree. Please see the question for deletion of tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see http://en.wikipedia.org/wiki/Reverse_Polish_notation to for the usage of postfix expression.

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

```
// C program for different tree traversals

#include <stdio.h>

#include <stdlib.h>
/* A binary tree node has data, pointer to left child

  and a pointer to right child */


struct node
{

    int data;


    struct node* left;


    struct node* right;
```

```
};
/* Helper function that allocates a new node with the

   given data and NULL left and right pointers. */

struct node* newNode(int data)
{

     struct node* node = (struct node*)

                                       malloc(sizeof(struct node));

     node->data = data;

     node->left = NULL;

     node->right = NULL;


     return(node);
}
/* Given a binary tree, print its nodes according to the

  "bottom-up" postorder traversal. */

void printPostorder(struct node* node)
{

     if (node == NULL)

        return;


     // first recur on left subtree

     printPostorder(node->left);


     // then recur on right subtree
```

```
    printPostorder(node->right);

    // now deal with the node

    printf("%d ", node->data);
}
/* Given a binary tree, print its nodes in inorder*/

void printInorder(struct node* node)
{

    if (node == NULL)

        return;

    /* first recur on left child */

    printInorder(node->left);

    /* then print the data of node */

    printf("%d ", node->data);

    /* now recur on right child */

    printInorder(node->right);
}
/* Given a binary tree, print its nodes in preorder*/

void printPreorder(struct node* node)
{

    if (node == NULL)
```

```
        return;

     /* first print data of node */

     printf("%d ", node->data);

     /* then recur on left sutree */

     printPreorder(node->left);

     /* now recur on right subtree */

     printPreorder(node->right);
}
/* Driver program to test above functions*/

int main()
{
     struct node *root  = newNode(1);

     root->left           = newNode(2);

     root->right          = newNode(3);

     root->left->left     = newNode(4);

     root->left->right   = newNode(5);

     printf("\nPreorder traversal of binary tree is \n");

     printPreorder(root);

     printf("\nInorder traversal of binary tree is \n");
```

```
    printInorder(root);

    printf("\nPostorder traversal of binary tree is \n");

    printPostorder(root);

    getchar();

    return 0;
}
```

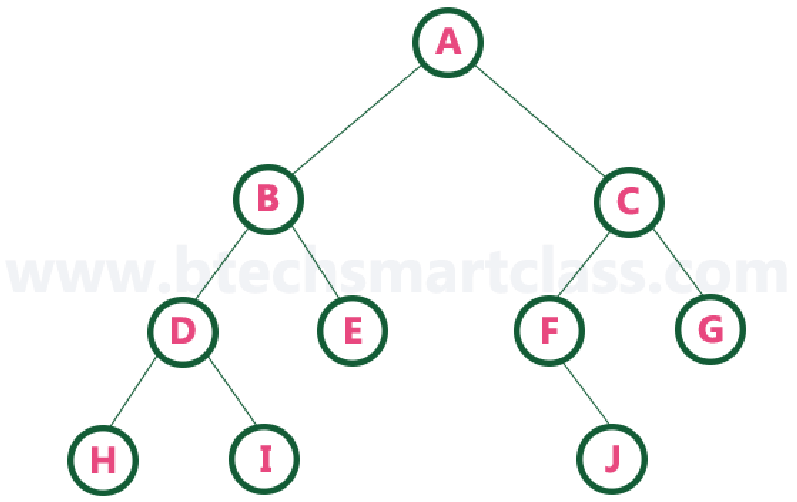
Binary Tree Datastructure

In a normal tree, every node can have any number of children. A binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as a left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



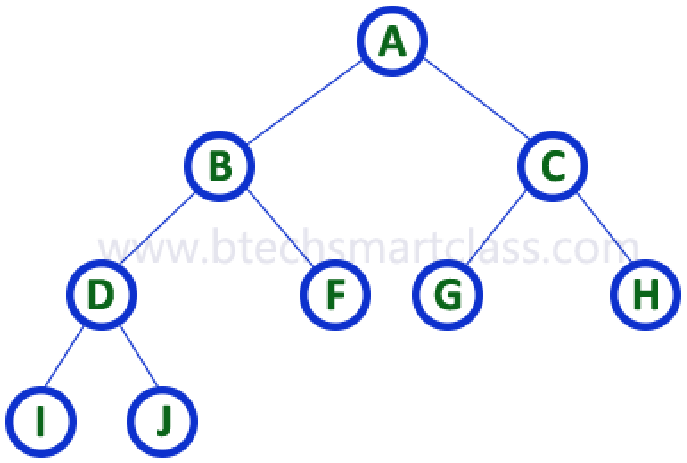
There are different types of binary trees and they are...

1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in a strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows**...**

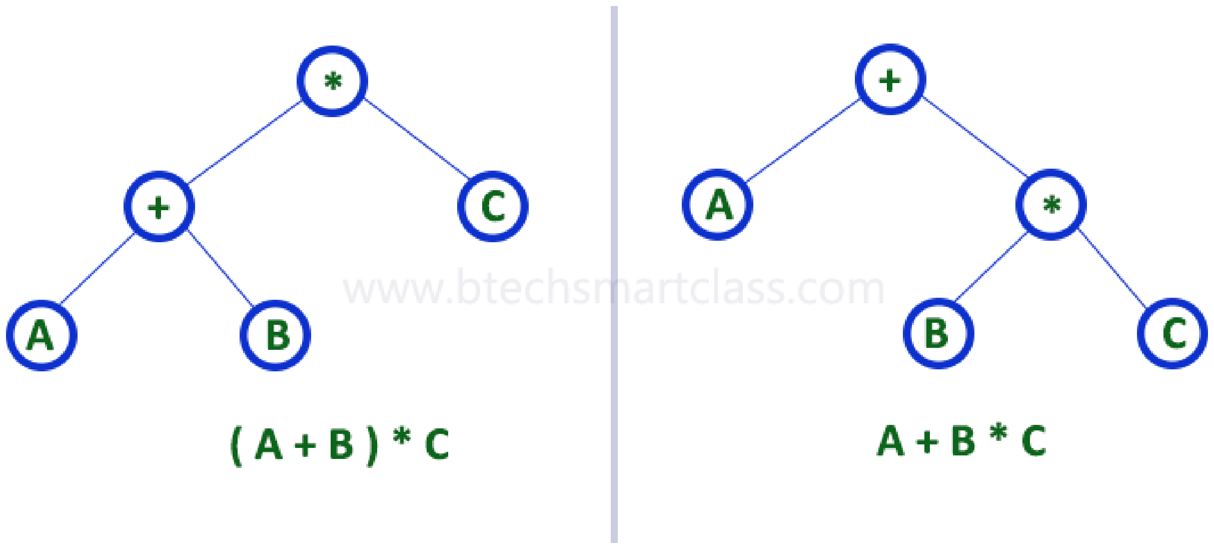
A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree



Strictly binary tree data structure is used to represent mathematical expressions.

Example

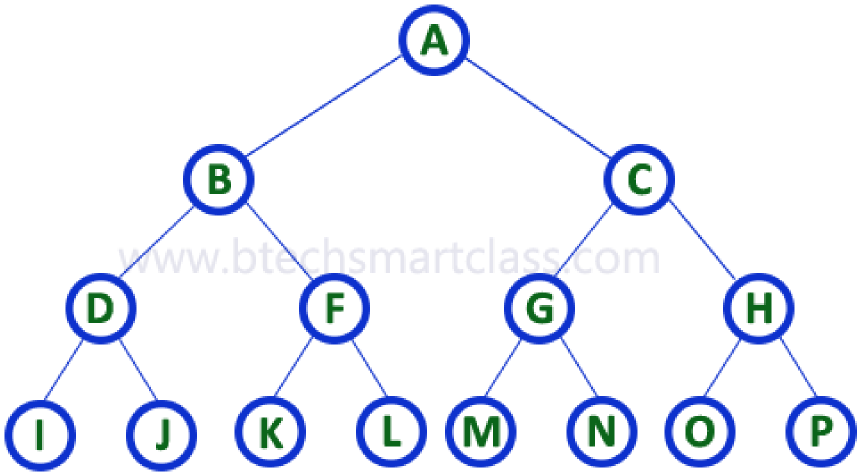


2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^level number of nodes. For example at level 2 there must be 2^2 = 4 nodes and at level 3 there must be 2^3 = 8 nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

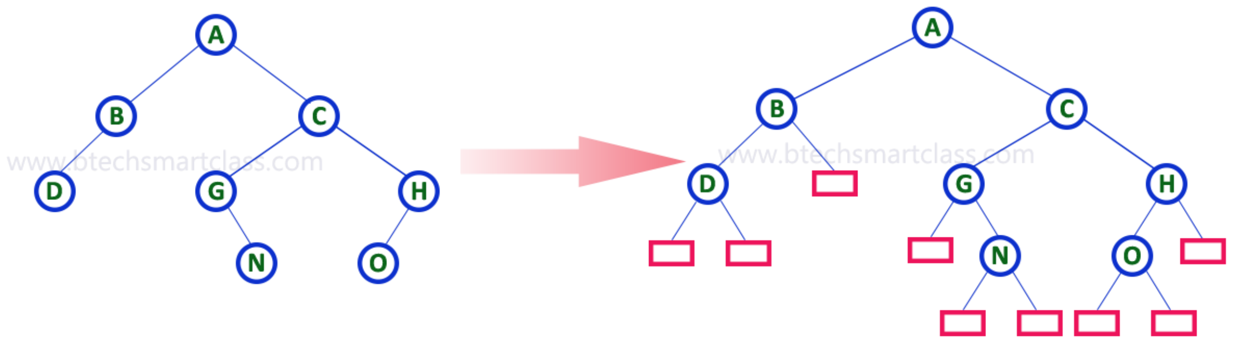
Complete binary tree is also called as Perfect Binary Tree



3. Extended Binary Tree

A binary tree can be converted into a Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called an Extended Binary Tree.



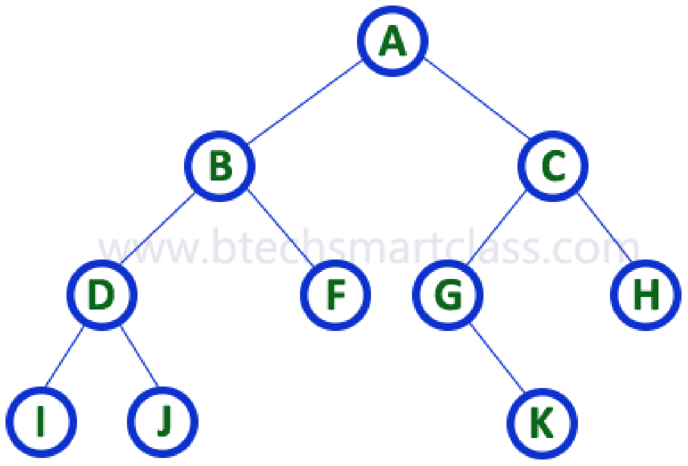
In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink colour).

Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Array Representation
2. Linked List Representation

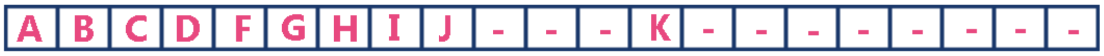
Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use 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...



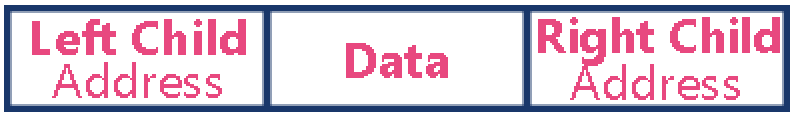
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n + 1.

2. Linked List Representation of Binary Tree

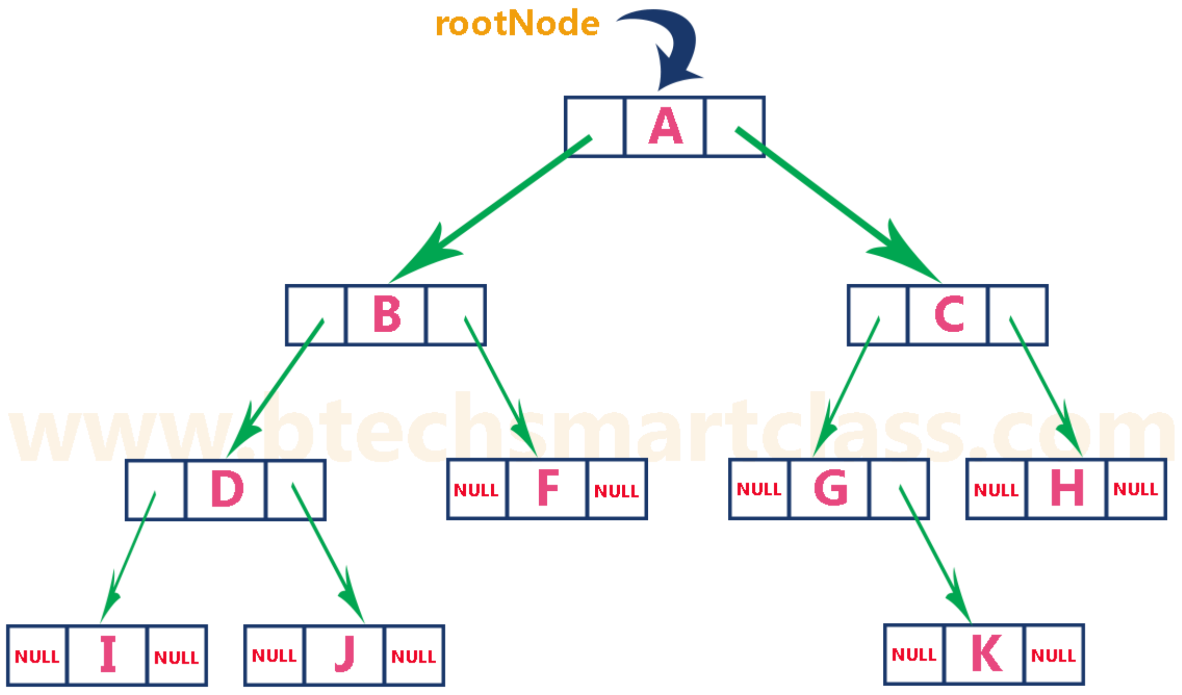
We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

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



Left Child Address	Data	Right Child Address

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



Binary Tree Traversals

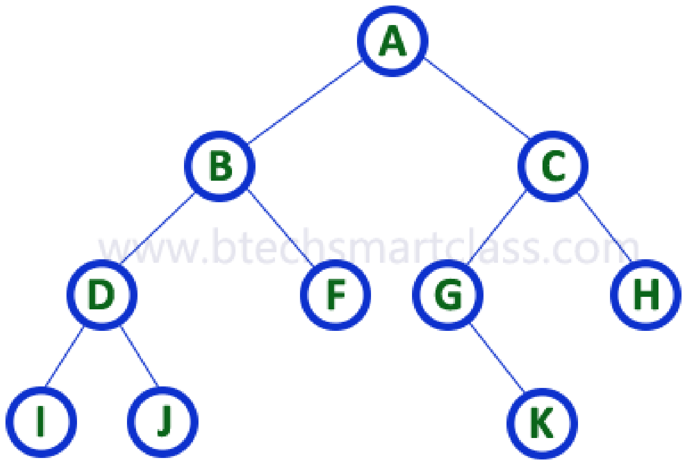
When we want to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree, displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called a Binary Tree Traversal.

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

Consider the following binary tree...



1. In - Order Traversal (leftChild - root - rightChild)

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

In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the leftmost child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for the right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node 'C' and next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of I - D - J - B - F - A - G - K - C - H using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. Pre - Order Traversal (root - leftChild - rightChild)

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

In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the leftmost child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this, we have completed node C's root and left parts. Next visit C's right child 'H' which is the rightmost child in the tree. So we stop the process.

That means here we have visited in the order of A-B-D-I-J-F-C-G-K-H using Pre-Order Traversal**.**

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

3. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of I - J - D - F - B - K - G - H - C - A using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

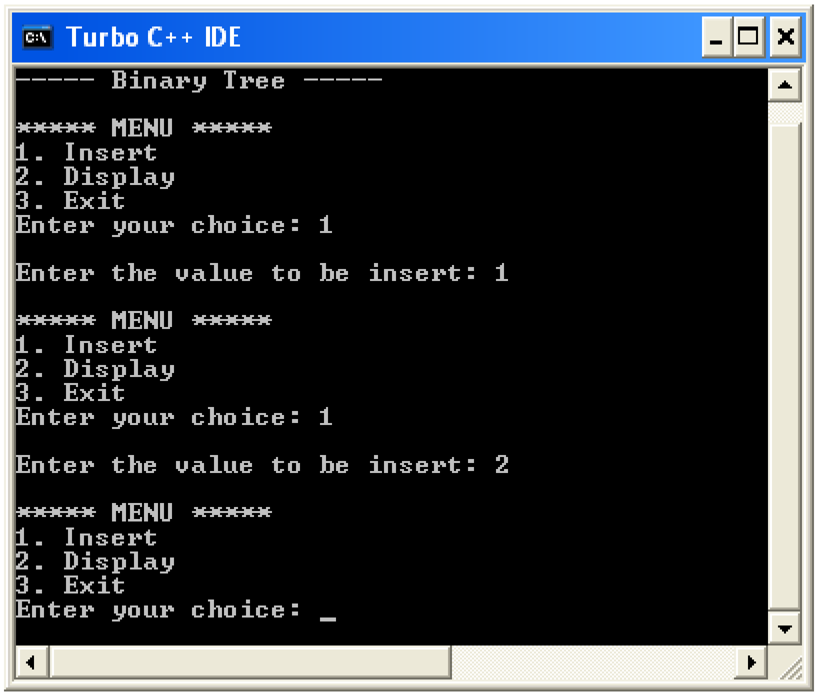
I - J - D - F - B - K - G - H - C - A

Program to Create Binary Tree and display using In-Order Traversal - C Programming

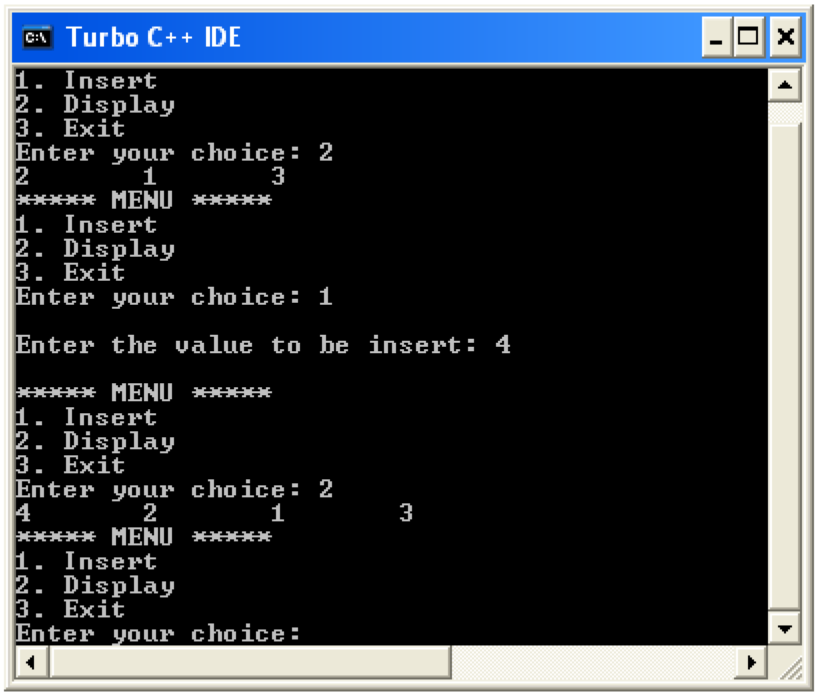
```
#include<stdio.h>
#include<conio.h>

struct Node{
  int data;
  struct Node *left;
  struct Node *right;
};

struct Node *root = NULL;
int count = 0;

struct Node* insert(struct Node*, int);
void display(struct Node*);

void main(){
  int choice, value;
  clrscr();
  printf("\n----- Binary Tree -----\n");
  while(1){
    printf("\n***** MENU *****\n");
    printf("1. Insert\n2. Display\n3. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
         case 1: printf("\nEnter the value to be insert: ");
                 scanf("%d", &value);
                 root = insert(root,value);
                 break;
         case 2: display(root); break;
```

```
            case 3: exit(0);
            default: printf("\nPlease select correct operations!!!\n");
      }
   }
}

struct Node* insert(struct Node *root,int value){
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(root == NULL){
      newNode->left = newNode->right = NULL;
      root = newNode;
      count++;
   }
   else{
      if(count%2 != 0)
            root->left = insert(root->left,value);
      else
            root->right = insert(root->right,value);
   }
   return root;
}
// display is performed by using Inorder Traversal
void display(struct Node *root)
{
   if(root != NULL){
      display(root->left);
      printf("%d\t",root->data);
      display(root->right);
   }
}
```

Output



```
------ Binary Tree ------

***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice: 1

Enter the value to be insert: 1

***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice: 1

Enter the value to be insert: 2

***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice: _
```



```
1. Insert
2. Display
3. Exit
Enter your choice: 2
2       1       3
***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice: 1

Enter the value to be insert: 4

***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice: 2
4       2       1       3
***** MENU *****
1. Insert
2. Display
3. Exit
Enter your choice:
```

Threaded Binary Trees

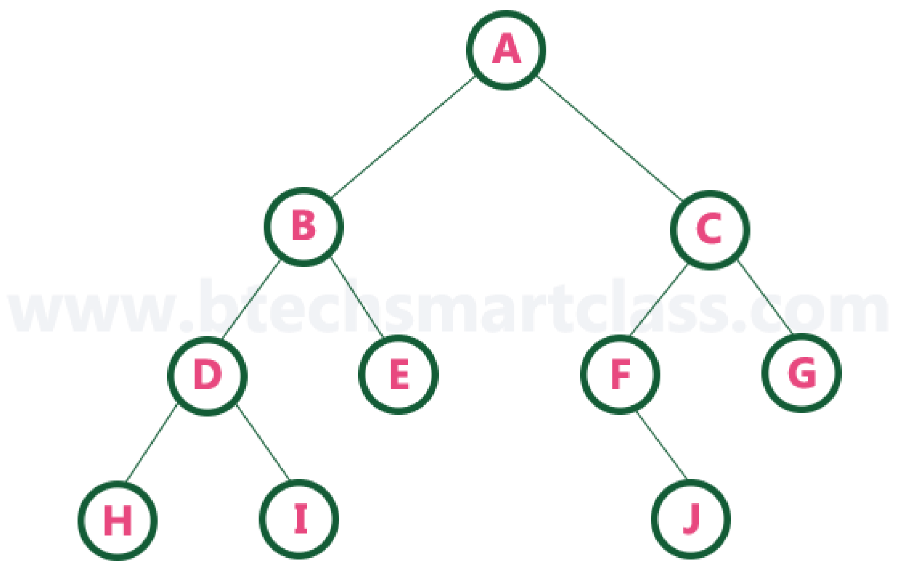
A binary tree can be represented using array representation or linked list representation. When a binary tree is represented using linked list representation, the reference part of the node which doesn't have a child is filled with a NULL pointer. In any binary tree linked list representation, there is a number of NULL pointers than actual pointers. Generally, in any binary tree linked list representation, if there are 2N number of reference fields, then N+1 number of reference fields are filled with NULL (N+1 are NULL out of 2N). This NULL pointer does not play any role except indicating that there is no link (no child).

A. J. Perlis and C. Thornton have proposed a new binary tree called "*Threaded Binary Tree*", which makes use of NULL pointers to improve its traversal process. In a threaded binary tree, NULL pointers are replaced by references of other nodes in the tree. These extra references are called threads.

Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.

If there is no in-order predecessor or in-order successor, then it points to the root node.

Consider the following binary tree...



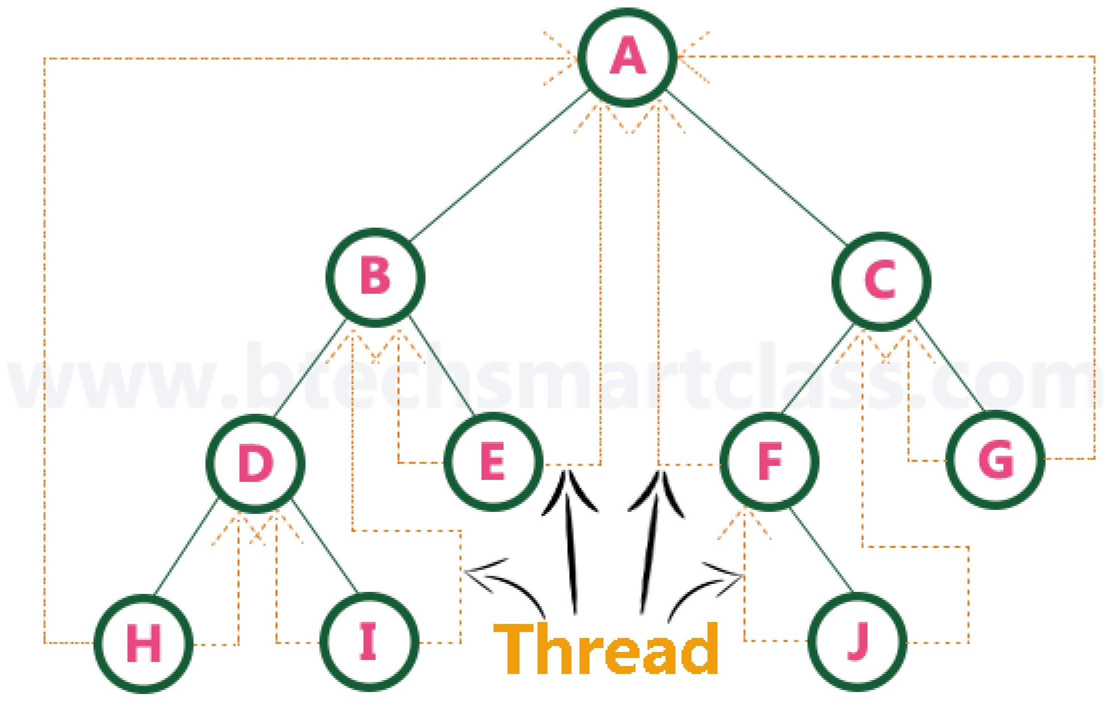
To convert the above example binary tree into a threaded binary tree, first find the in-order traversal of that tree...

In-order traversal of above binary tree...

H - D - I - B - E - A - F - J - C - G

When we represent the above binary tree using linked list representation, nodes H, I, E, F, J and G left child pointers are NULL. This NULL is replaced by address of its in-order predecessor respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A. And nodes H, I, E, J and G right child pointers are NULL. These NULL pointers are replaced by address of its in-order successor respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

Above example binary tree is converted into threaded binary tree as follows.



In the above figure, threads are indicated with dotted links.

Binary Search Tree

In a binary tree, every node can have a maximum of two children but there is no need to maintain the order of nodes basing on their values. In a binary tree, the elements are arranged in the order they arrive at the tree from top to bottom and left to right.

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left_subtree (keys) < node (key) ≤ right_subtree (keys)

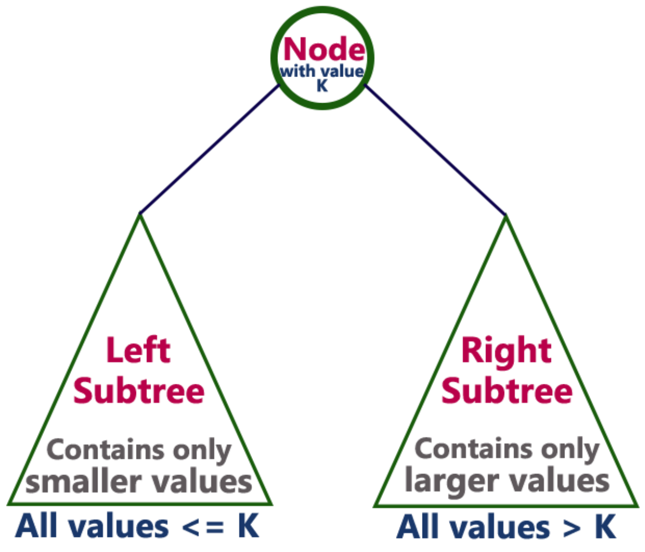
A binary tree has the following time complexities...

1. Search Operation - O(n)
2. Insertion Operation - O(1)
3. Deletion Operation - O(n)

To enhance the performance of binary tree, we use a special type of binary tree known as Binary Search Tree. Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows...

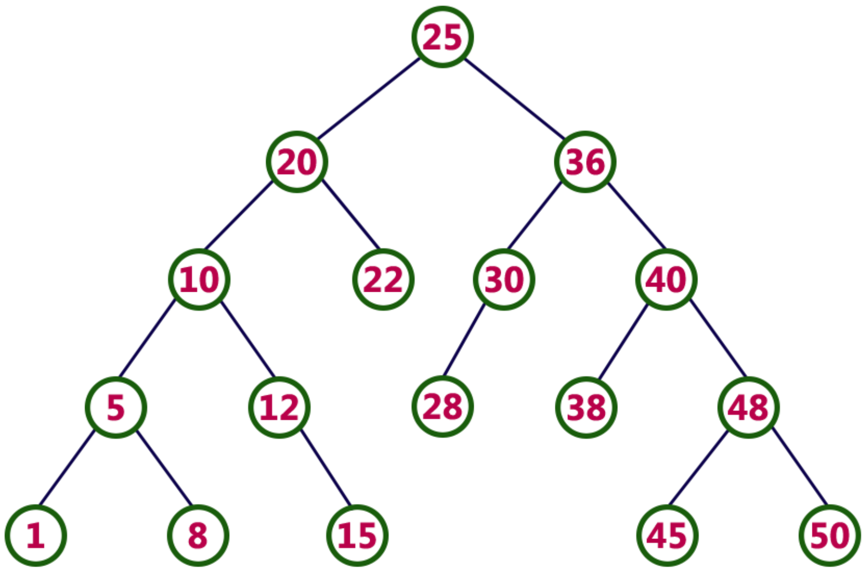
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as shown in the following figure...



Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Every binary search tree is a binary tree but every binary tree need not to be binary search tree.

Operations on a Binary Search Tree

The following operations are performed on a binary search tree...

1. Search
2. Insertion
3. Deletion

Search Operation in BST

In a binary search tree, the search operation is performed with O(log n) time complexity. The search operation is performed as follows...

- Step 1 - Read the search element from the user.
- Step 2 - Compare the search element with the value of root node in the tree.
- Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function
- Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.
- Step 5 - If search element is smaller, then continue the search process in left subtree.
- Step 6- If search element is larger, then continue the search process in right subtree.
- Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

- Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.
- Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with O(log n) time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- Step 1 - Create a newNode with given value and set its left and right to NULL.
- Step 2 - Check whether tree is Empty.
- Step 3 - If the tree is Empty, then set root to newNode.
- Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).
- Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.
- Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).
- Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with O(log n) time complexity. Deleting a node from Binary search tree includes following three cases...

- Case 1: Deleting a Leaf node (A node with no children)
- Case 2: Deleting a node with one child
- Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

- Step 1 - Find the node to be deleted using search operation
- Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

- Step 1 - Find the node to be deleted using search operation
- Step 2 - If it has only one child then create a link between its parent node and child node.

- Step 3 - Delete the node using free function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

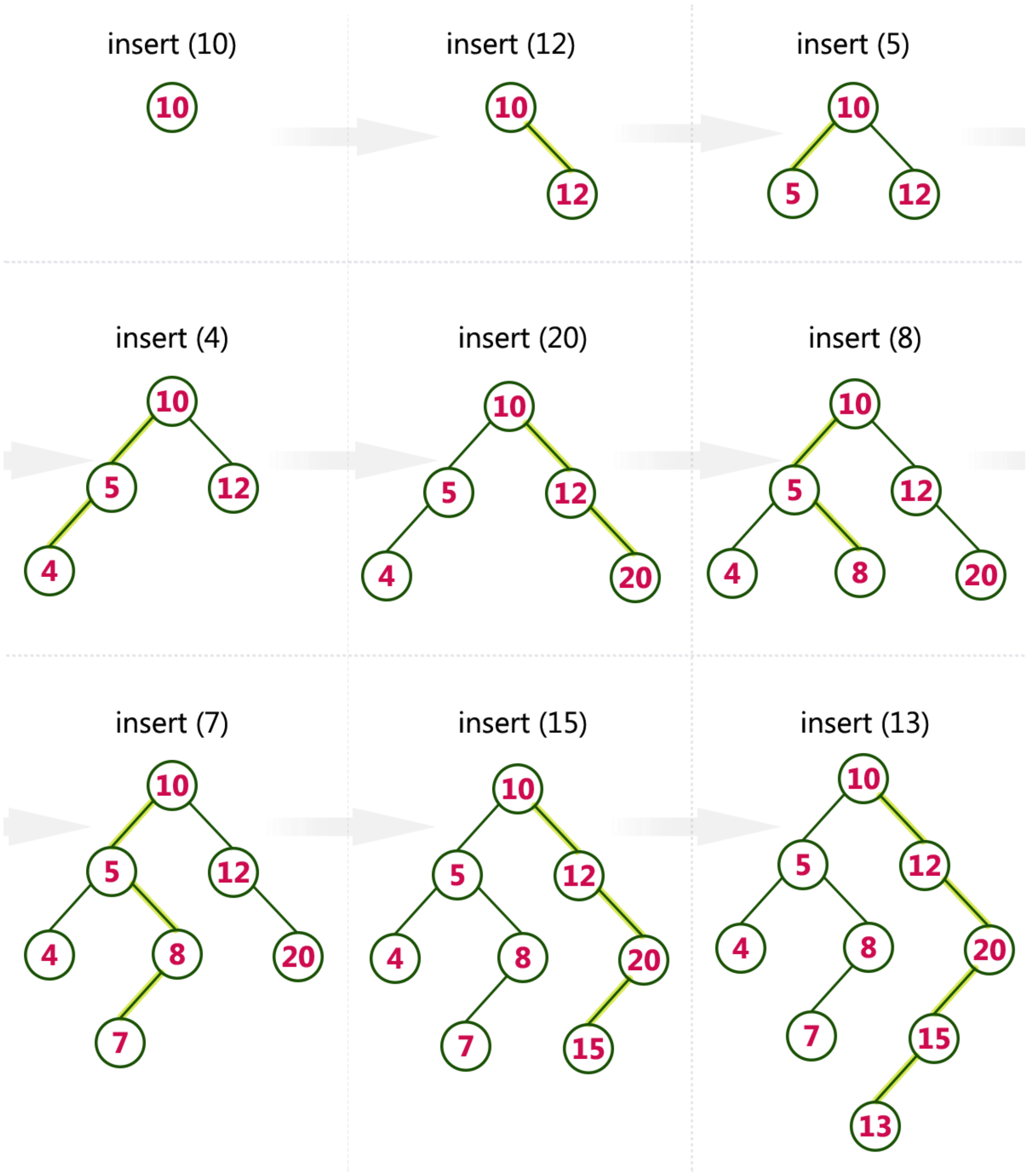
- Step 1 - Find the node to be deleted using search operation
- Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.
- Step 3 - Swap both deleting node and node which is found in the above step.
- Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2
- Step 5 - If it comes to case 1, then delete using case 1 logic.
- Step 6- If it comes to case 2, then delete using case 2 logic.
- Step 7 - Repeat the same process until the node is deleted from the tree.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

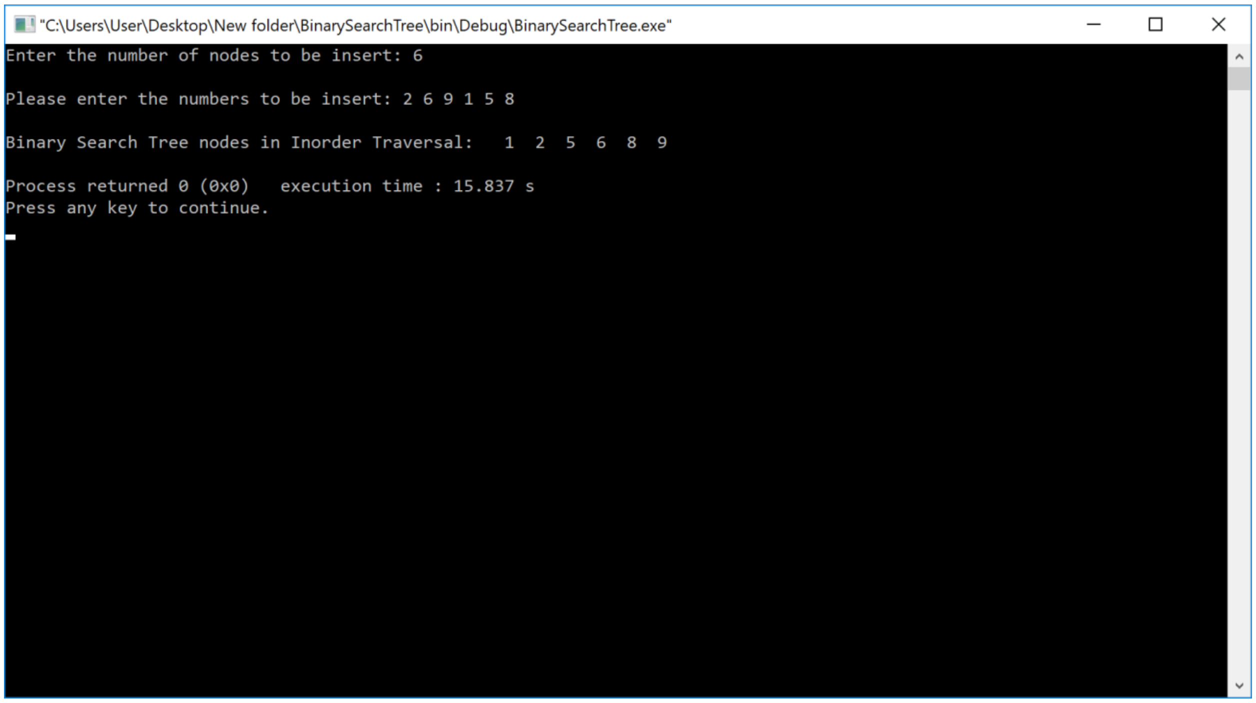
Above elements are inserted into a Binary Search Tree as follows...



Implementaion of Binary Search Tree using C Programming Language

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

struct node
{
        int data;
        struct node *left;
        struct node *right;
};


void inorder(struct node *root)
{
        if(root)
        {
                inorder(root->left);
                printf("  %d",root->data);
                inorder(root->right);
        }
}

int main()
{
        int n , i;
        struct node *p , *q , *root;
        printf("Enter the number of nodes to be insert: ");
        scanf("%d",&n);

        printf("\nPlease enter the numbers to be insert: ");

        for(i=0;i<i++)
        {
                p = (struct node*)malloc(sizeof(struct node));
                scanf("%d",&p->data);
                p->left = NULL;
                p->right = NULL;
                if(i == 0)
                {
                        root = p; // root always point to the root node
                }
                else
                {
                        q = root;   // q is used to traverse the tree
                        while(1)
                        {
                                if(p->data > q->data)
                                {
                                        if(q->right == NULL)
                                                {
                                                q->right = p;
                                                break;
                                                }
                                        else
                                                q = q->right;
                                }
                                else
                                {
                                        if(q->left == NULL)
                                              {
                                              q->left = p;
                                              break;
                                              }
                                        else
                                              q = q->left;
                                  }
                            }

                      }

                }

printf("\nBinary Search Tree nodes in Inorder Traversal: ");
inorder(root);
printf("\n");

return 0;
}
```

Output



"C:\Users\User\Desktop\New folder\BinarySearchTree\bin\Debug\BinarySearchTree.exe"

```
Enter the number of nodes to be insert: 6

Please enter the numbers to be insert: 2 6 9 1 5 8

Binary Search Tree nodes in Inorder Traversal:   1  2  5  6  8  9

Process returned 0 (0x0)   execution time : 15.837 s
Press any key to continue.
```

Heap Data Structure

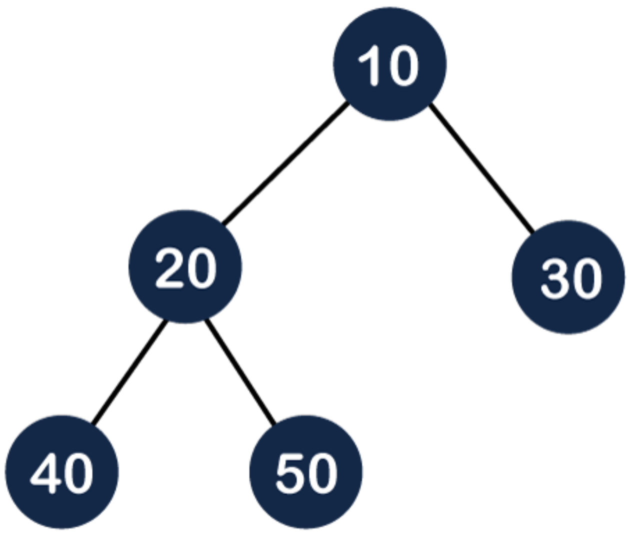
What is Heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have at most two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

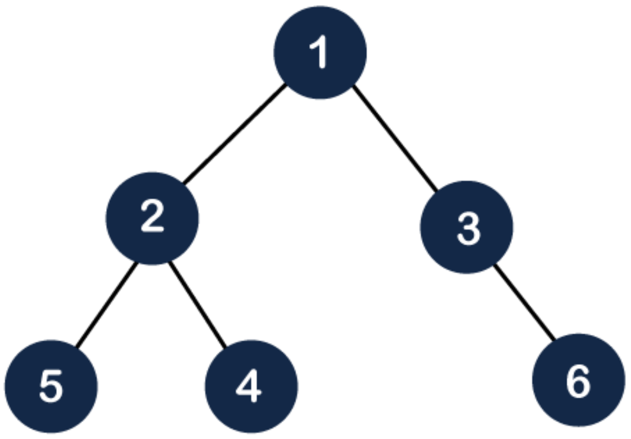
What is a complete binary tree?

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.

Let's understand through an example.



In the above figure, we can observe that all the internal nodes are completely filled except the leaf node; therefore, we can say that the above tree is a complete binary tree.



The above figure shows that all the internal nodes are completely filled except the leaf node, but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.

Note: The heap tree is a special balanced binary tree data structure where the root node is compared with its children and arrange accordingly.

How can we arrange the nodes in the Tree?

There are two types of the heap:

- Min Heap
- Max heap

Min Heap: The value of the parent node should be less than or equal to either of its children.

Or

In other words, the min-heap can be defined as, for every node i, the value of node i is greater than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] <= A[i]

Max Heap: The value of the parent node is greater than or equal to its children.

Or

In other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] >= A[i]

Time complexity in Max Heap

The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).

Max Heap Datastructure

Heap data structure is a specialized binary tree-based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on their values. A heap data structure sometimes also called a Binary Heap.

There are two types of heap data structures and they are as follows...

1. **Max Heap**
2. **Min Heap**

Every heap data structure has the following properties...

Property #1 (Ordering): Nodes must be arranged in an order according to their values based on Max heap or Min heap.

Property #2 (Structural): All levels in a heap must be full except the last level and all nodes must be filled from left to right strictly.

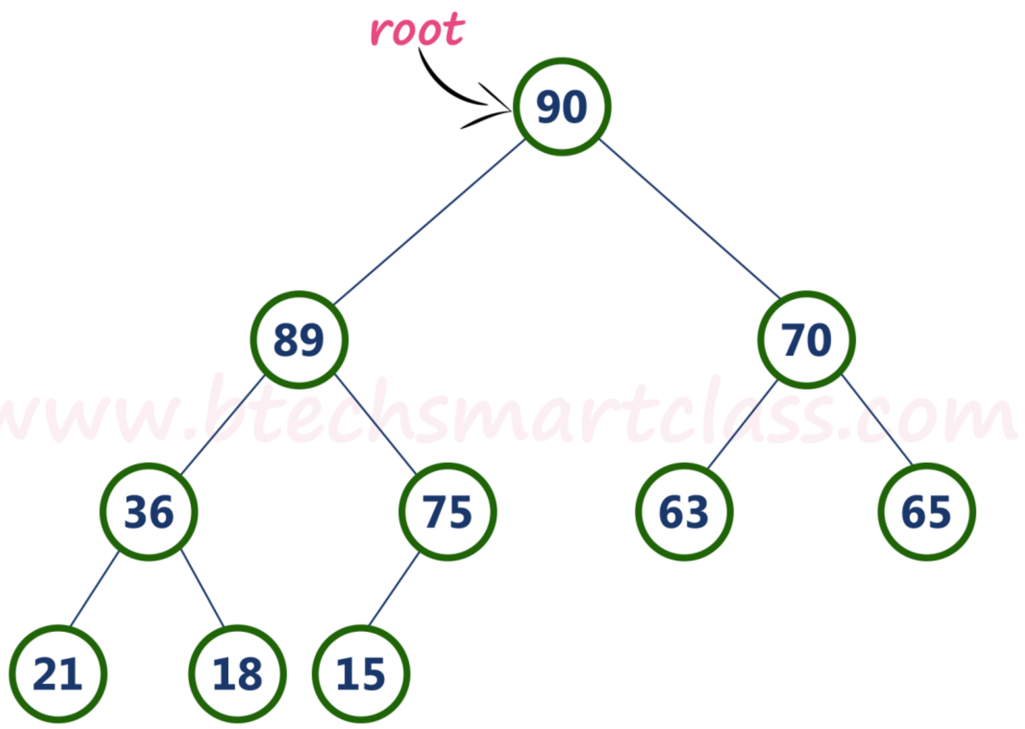
Max Heap

Max heap data structure is a specialized full binary tree data structure. In a max heap nodes are arranged based on node value.

Max heap is defined as follows...

Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes.

Example



Above tree is satisfying both Ordering property and Structural property according to the Max Heap data structure.

Operations on Max Heap

The following operations are performed on a Max heap data structure...

1. **Finding Maximum**
2. **Insertion**
3. **Deletion**

Finding Maximum Value Operation in Max Heap

Finding the node which has maximum value in a max heap is very simple. In a max heap, the root node has the maximum value than all other nodes. So, directly we can display root node value as the maximum value in max heap.

Insertion Operation in Max Heap

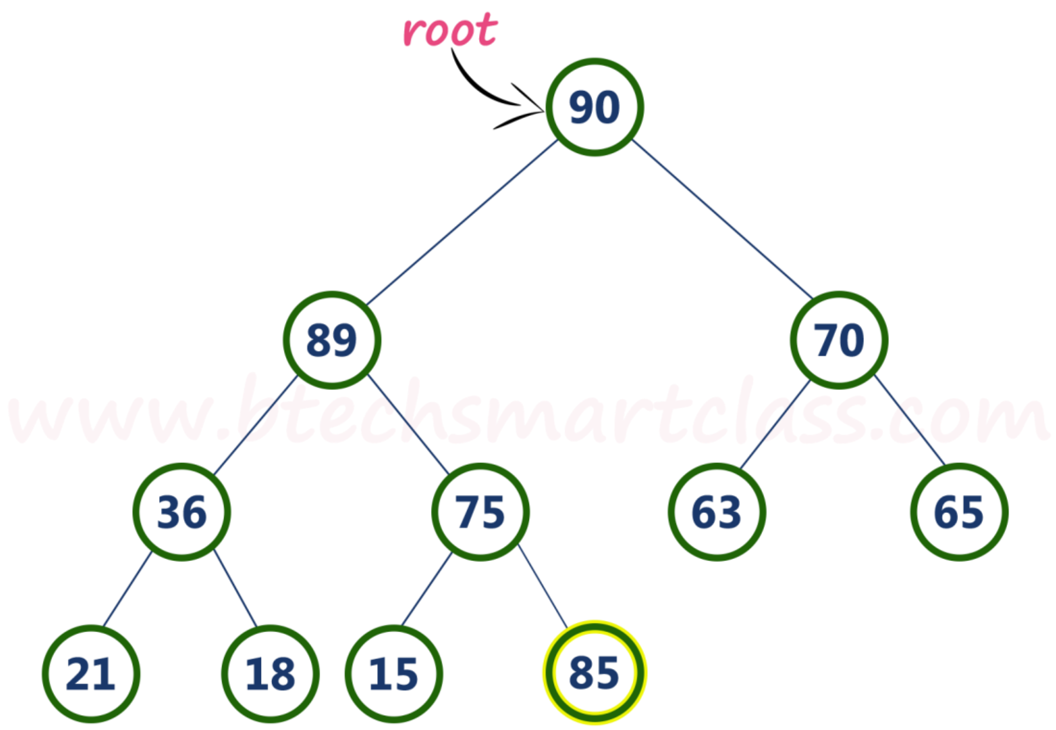
Insertion Operation in max heap is performed as follows...

- Step 1 - Insert the **newNode** as **last leaf** from left to right.
- Step 2 - Compare **newNode value** with its **Parent node**.
- Step 3 - If **newNode value is greater** than its parent, then **swap** both of them.
- Step 4 - Repeat step 2 and step 3 until newNode value is less than its parent node (or) newNode reaches to root.

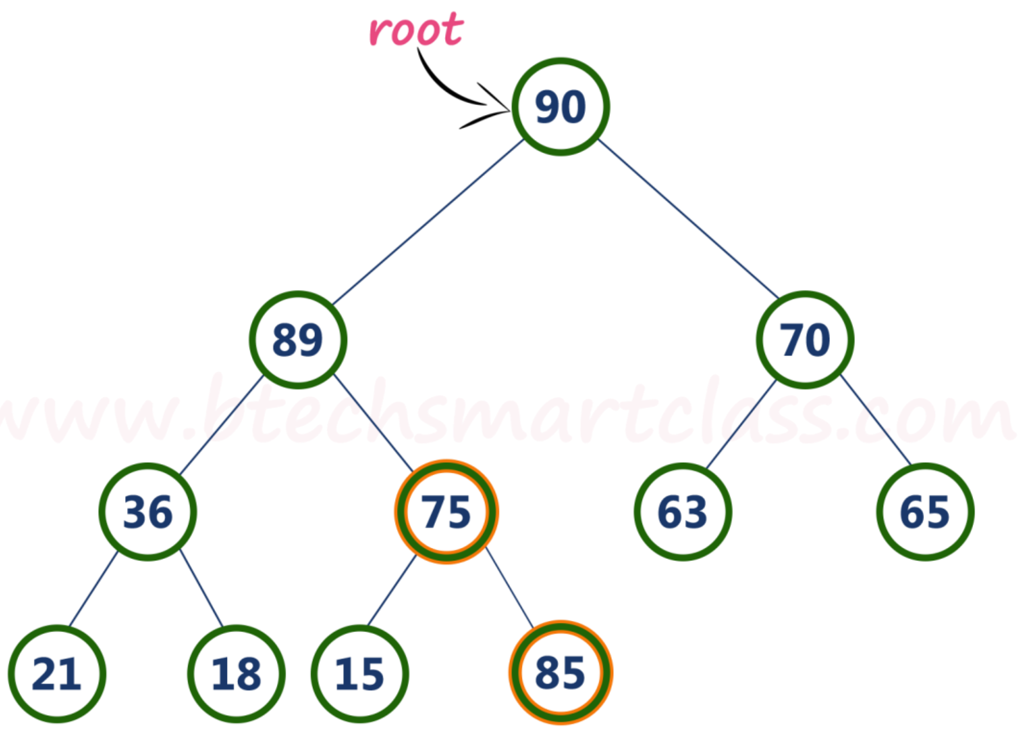
Example

Consider the above max heap. **Insert a new node with value 85.**

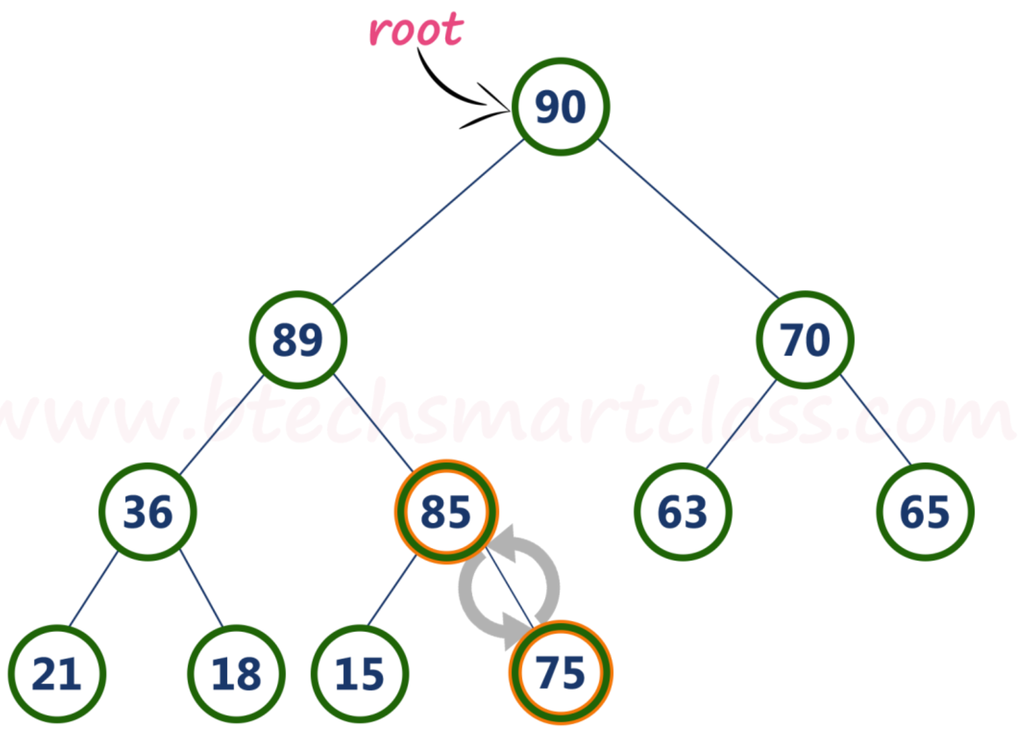
- Step 1 - Insert the **newNode** with value 85 as **last leaf** from left to right. That means newNode is added as a right child of node with value 75. After adding max heap is as follows...



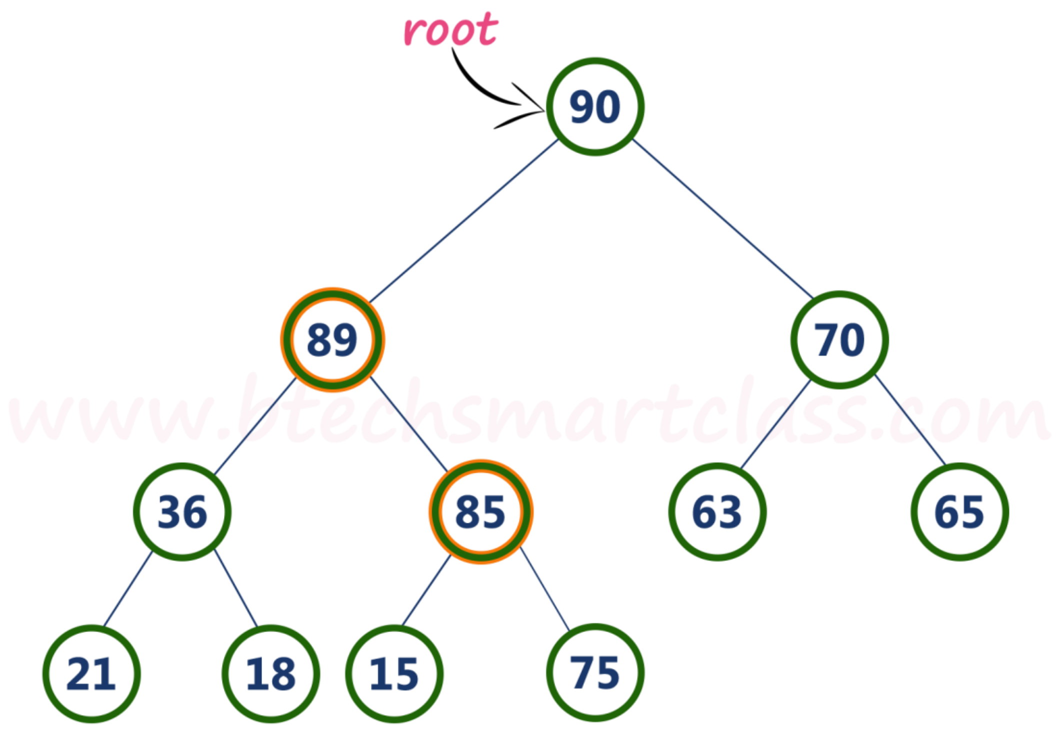
- Step 2 - Compare **newNode value (85)** with its **Parent node value (75)**. That means **85 > 75**



- Step 3 - Here **newNode value (85) is greater** than its **parent value (75)**, then **swap** both of them. After swapping, max heap is as follows...

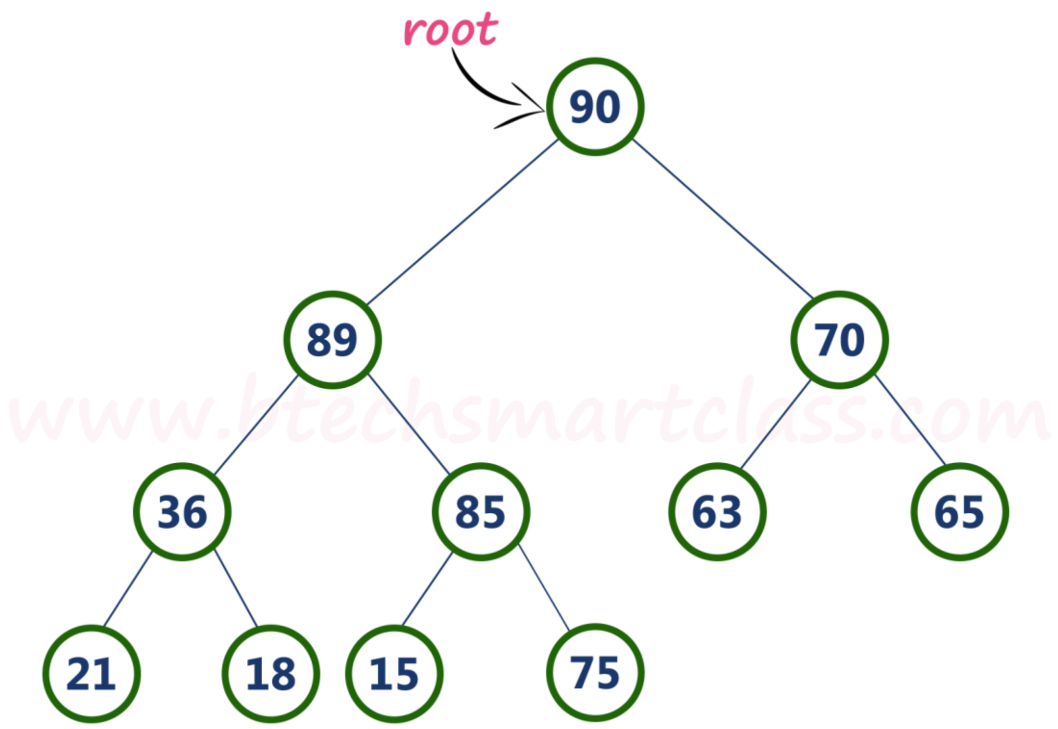


- Step 4 - Now, again compare newNode value (85) with its parent node value (89).



Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process.

Finally, max heap after insertion of a new node with value 85 is as follows...



Deletion Operation in Max Heap

In a max heap, deleting the last node is very simple as it does not disturb max heap properties.

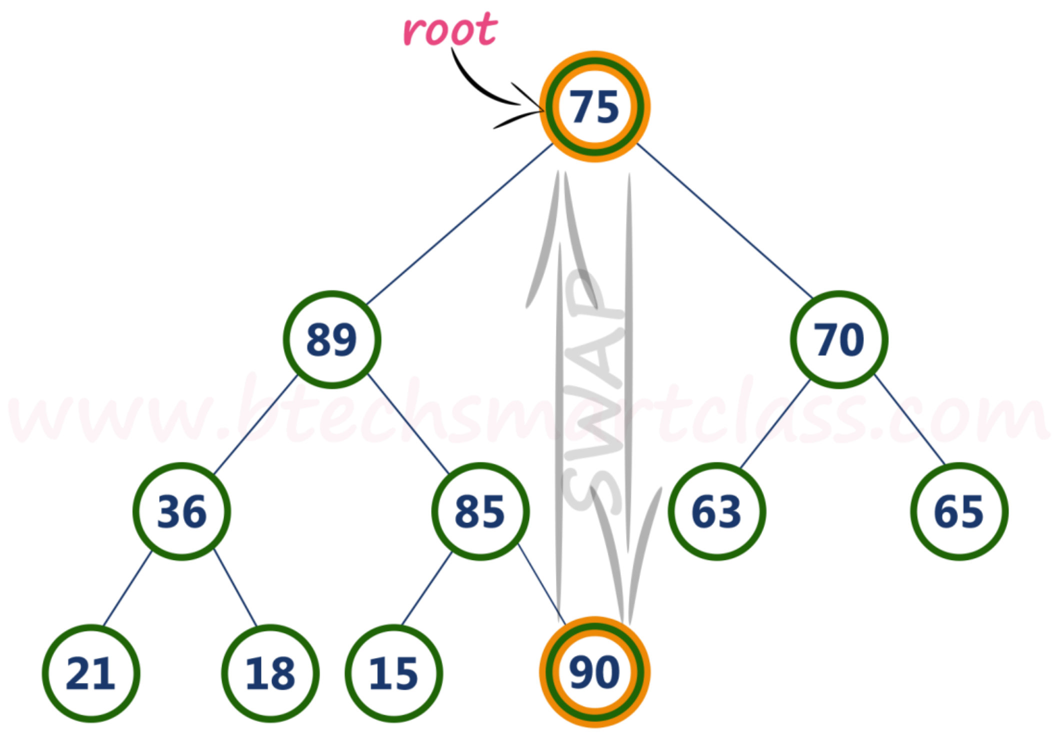
Deleting root node from a max heap is little difficult as it disturbs the max heap properties. We use the following steps to delete the root node from a max heap...

- Step 1 - **Swap** the **root** node with **last** node in max heap
- Step 2 - **Delete** last node.
- Step 3 - Now, compare **root value** with its **left child value**.
- Step 4 - If the root **value is smaller** than its left child, then compare the left **child** with its **right sibling**. Else goto **Step 6**
- Step 5 - If **left child value is larger** than its **right sibling**, then **swap root** with **left child** otherwise **swap root** with its **right child**.
- Step 6 - If **root value is larger** than its left child, then compare **root value** with its **right child** value.
- Step 7 - If **root value is smaller** than its **right child**, then **swap root** with the right **child** otherwise **stop the process**.
- Step 8 - Repeat the same until the root node fixes at its exact position.

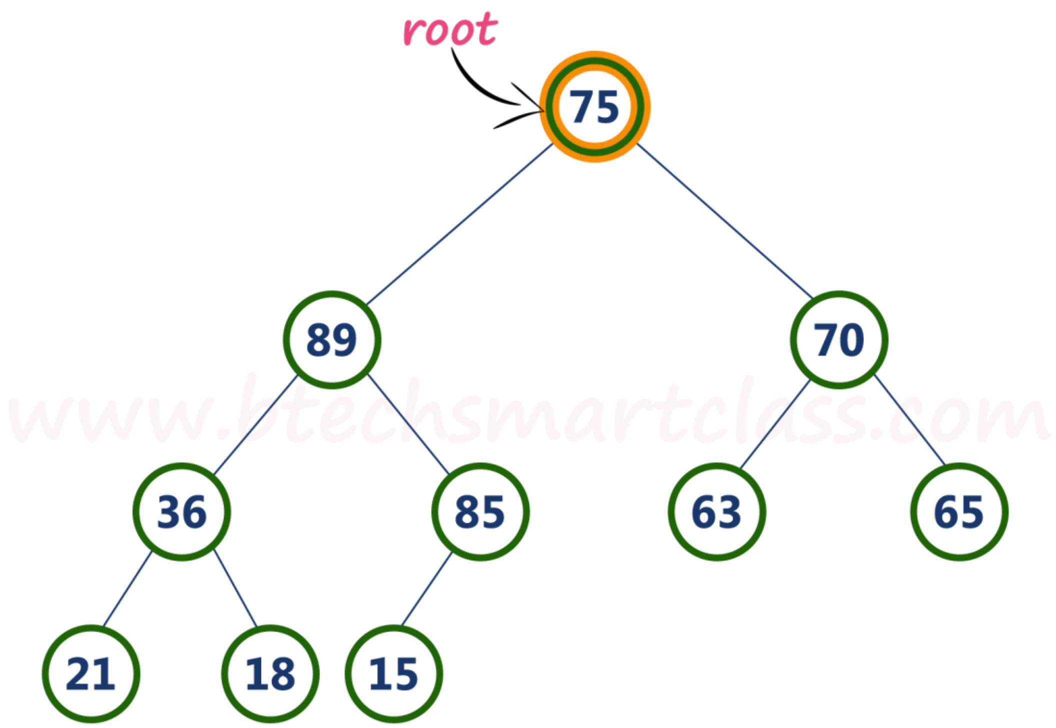
Example

Consider the above max heap. **Delete root node (90) from the max heap.**

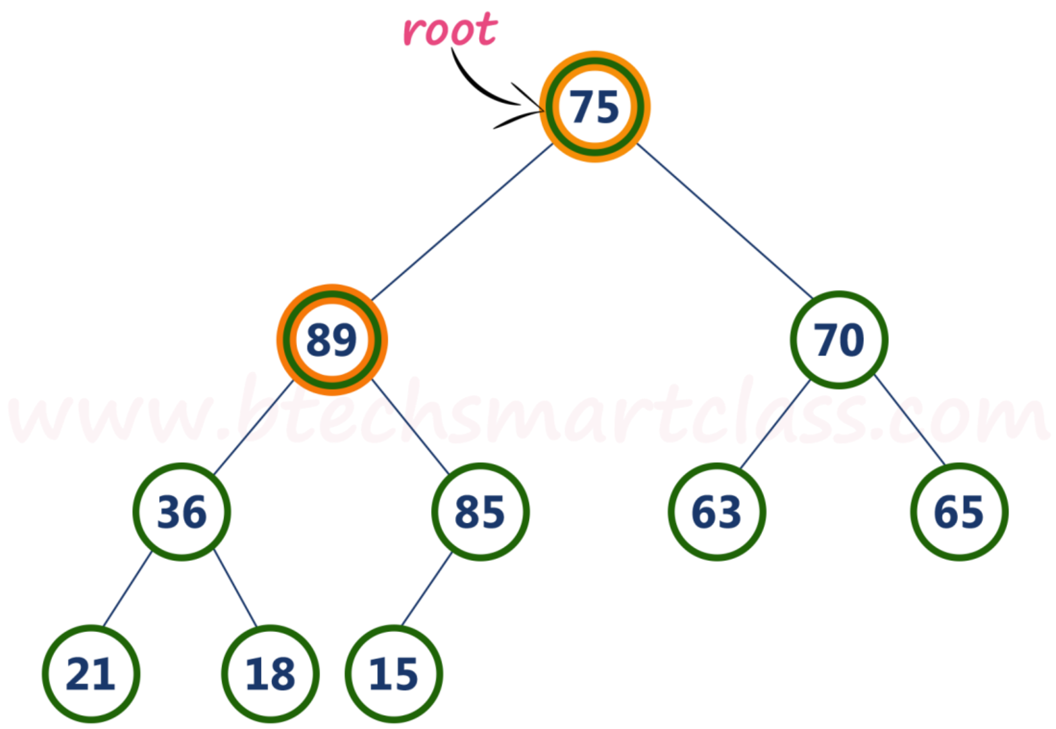
- Step 1 - **Swap** the **root node (90)** with **last node 75** in max heap. After swapping max heap is as follows...



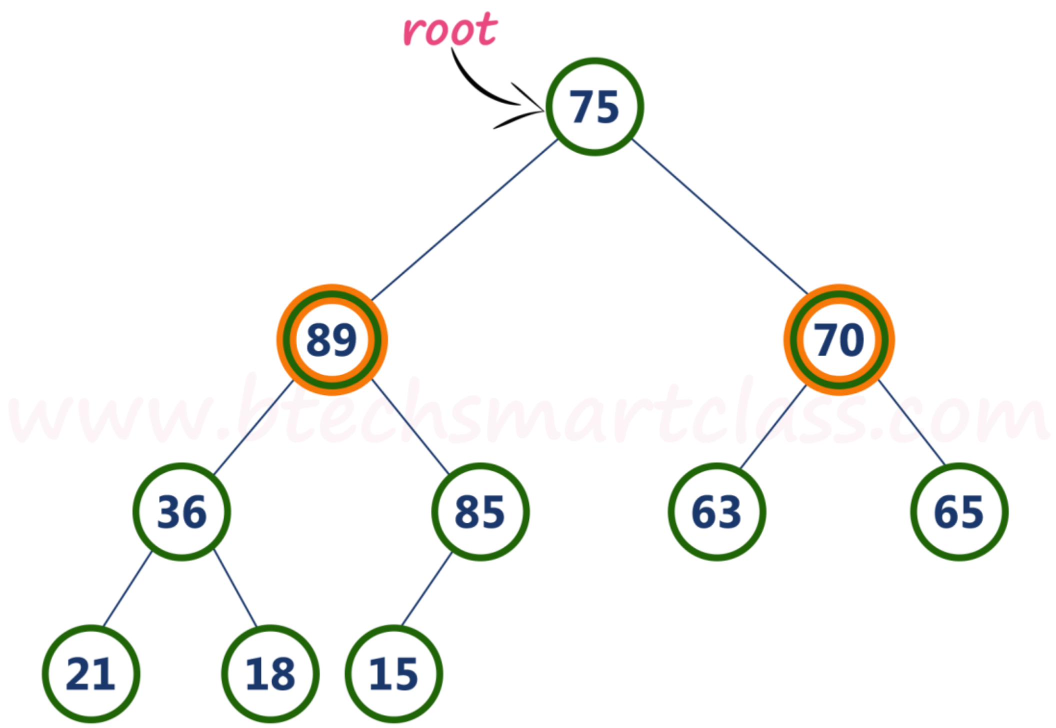
- Step 2 - **Delete** last node. Here the last node is 90. After deleting node with value 90 from heap, max heap is as follows...



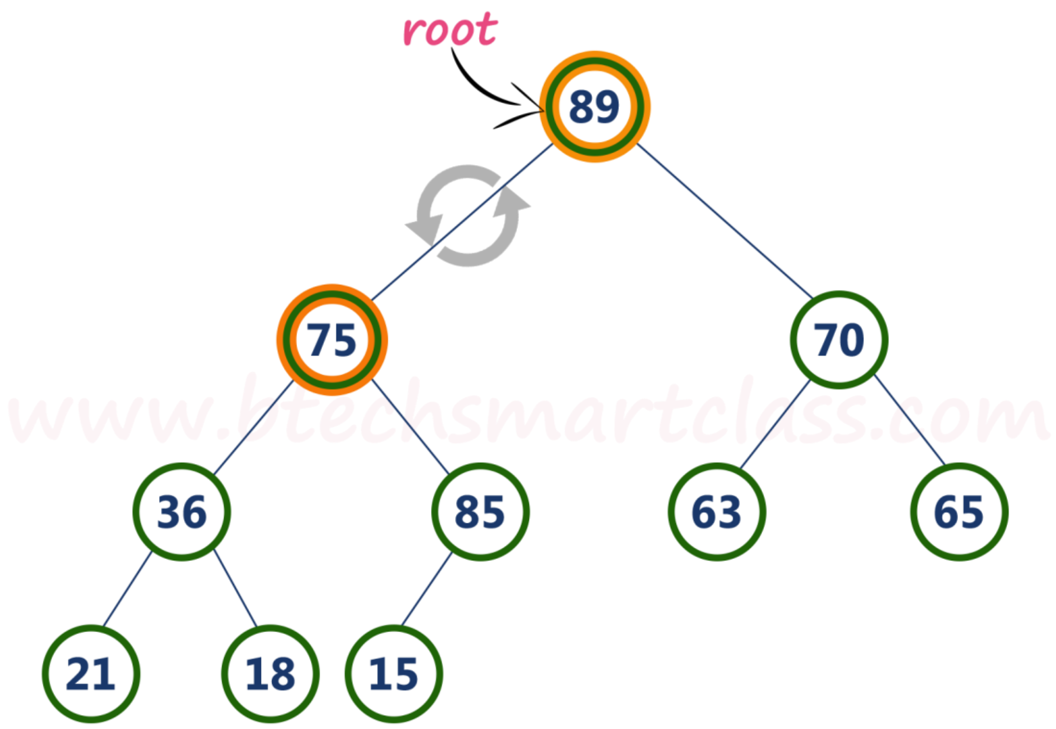
- Step 3 - Compare **root node (75)** with its **left child (89)**.



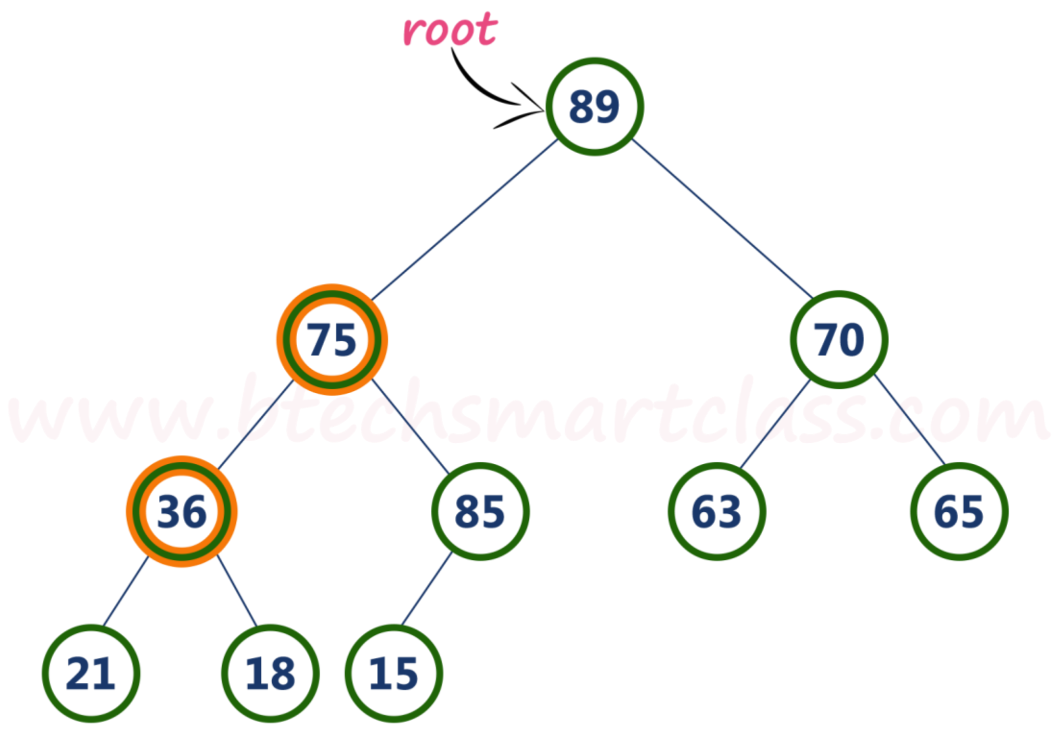
- Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).



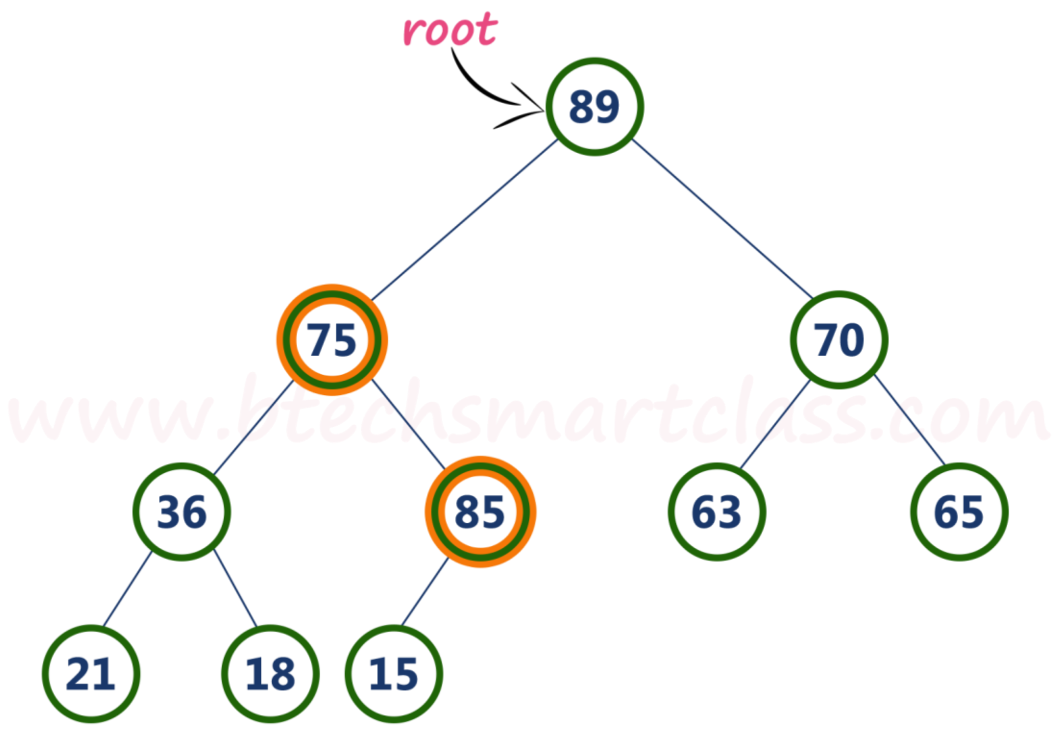
Step 4 - Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.



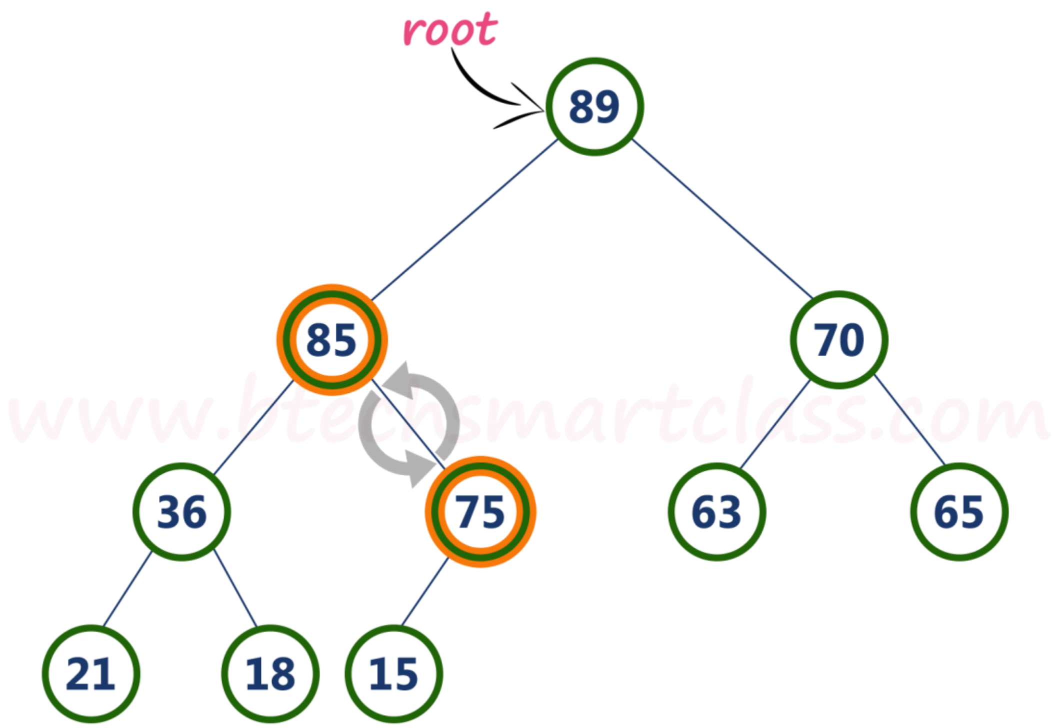
- Step 5 - Now, again compare **75** with its **left child (36)**.



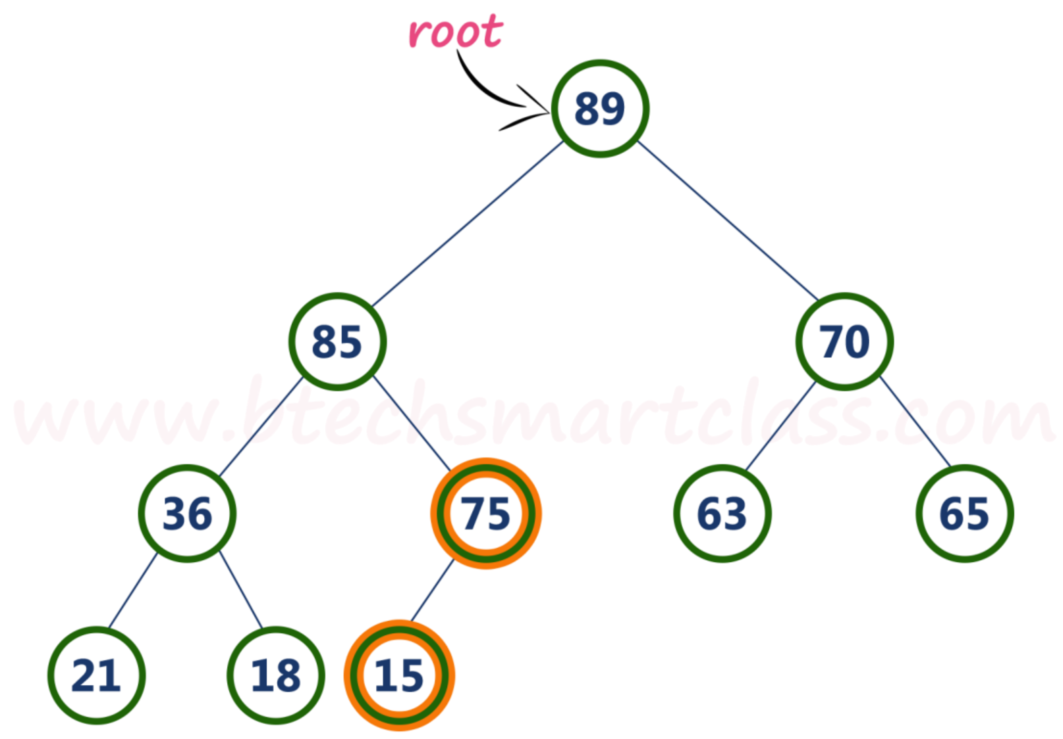
- Here, node with value **75** is larger than its left child. So, we compare node **75** with its right child **85**.



Step 6 - Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them. After swapping max heap is as follows...

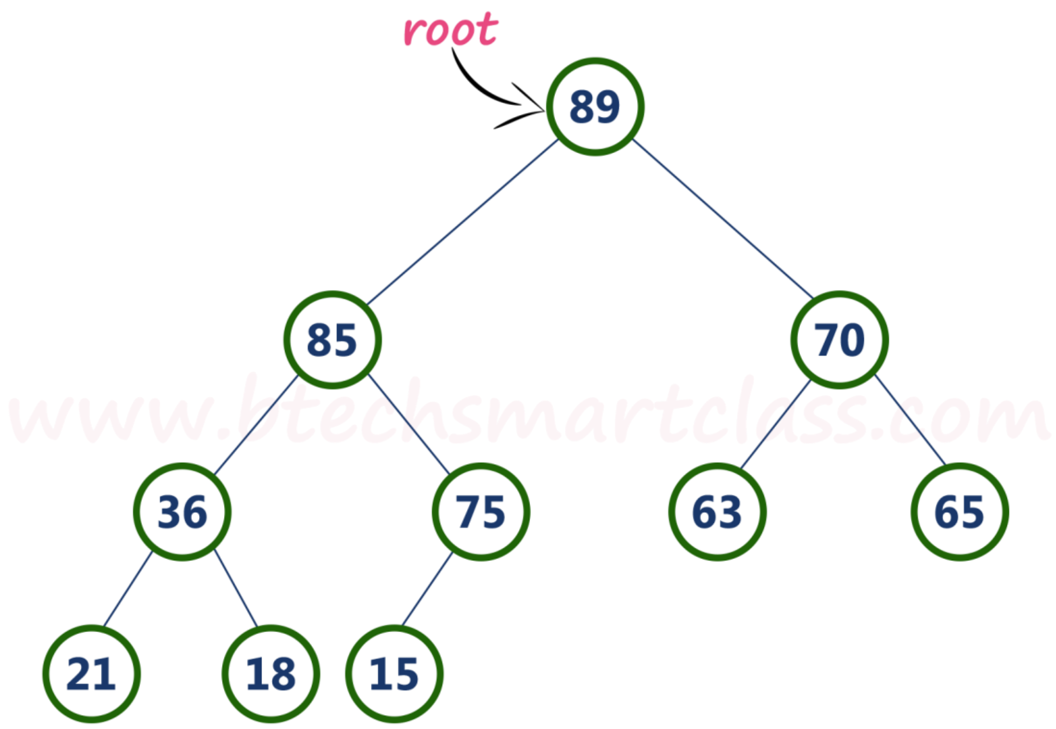


- Step 7 - Now, compare node with value **75** with its left child (**15**).



Here, node with value **75** is larger than its left child (**15**) and it does not have right child. So we stop the process.

Finally, max heap after deleting root node (**90**) is as follows...

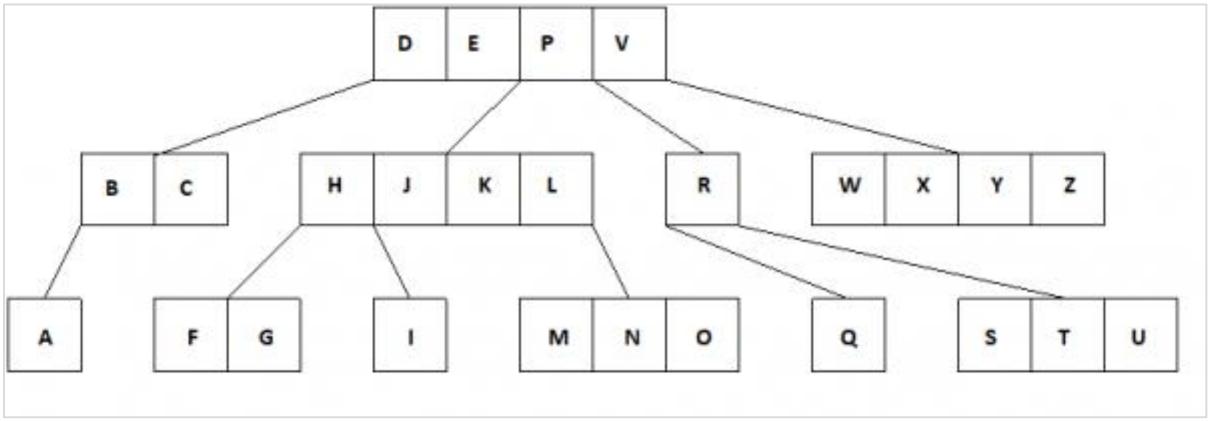


Multi-Way Trees

A multiway tree is defined as a tree that can have more than two children. If a multiway tree can have maximum m children, then this tree is called as multiway tree of order m (or an m-way tree).

As with the other trees that have been studied, the nodes in an m-way tree will be made up of m-1 key fields and pointers to children.

multiway tree of order 5



To make the processing of m-way trees easier some type of constraint or order will be imposed on the keys within each node, resulting in a multiway search tree of order m (or an m-way search tree). By definition an m-way search tree is a m-way tree in which following condition should be satisfied −

- Each node is associated with m children and m-1 key fields
- The keys in each node are arranged in ascending order.
- The keys in the first j children are less than the j-th key.
- The keys in the last m-j children are higher than the j-th key.

What is the B tree?

https://www.javatpoint.com/b-tree

B tree is a self-balancing tree, and it is a m-way search tree where m defines the order of the tree. **Btree** is a generalization of the Binary Search tree in which a node can have more than one key and more than two children depending upon the value of **m**. In the B tree, the data is specified in a sorted order having lower values on the left subtree and higher values in the right subtree.

Use

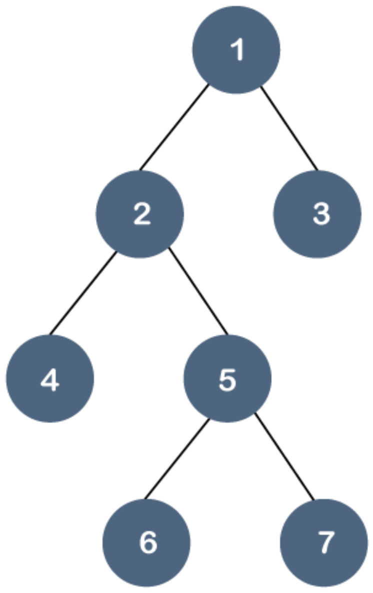
This type of tree will be used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node.

A B-tree of order m is a multiway search tree in which:

- The root has at least two subtrees unless it is the only node in the tree.
- Each nonroot and each nonleaf node have at most m nonempty children and at least ⌈m/2⌉ nonempty children. (Every node in a B-Tree except the root node and the leaf node contain at least ⌈m/2⌉ children.)
- The number of keys in each nonroot and each nonleaf node is one less than the number of its nonempty children.
- In the B tree, all the leaf nodes must be at the same level, whereas, in the case of a binary tree, the leaf nodes can be at different levels

These restrictions make B-trees always at least half full, have few levels, and remain perfectly balanced.The nodes in a B-tree are usually implemented as a class that contains an array of m-l cells for keys, an array of m pointers to other nodes, and whatever other information is required in order to facilitate tree maintenance.

Let's understand this property through an example.



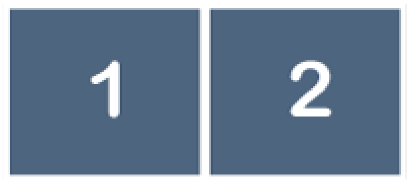
In the above tree, all the leaf nodes are not at the same level, but they have the utmost two children. Therefore, we can say that the above tree is a binary tree but not a B tree.

- If the Btree has an order of m, then each node can have a maximum of **m** In the case of minimum children, the leaf nodes have zero children, the root node has two children, and the internal nodes have a ceiling of m/2.
- Each node can have maximum (m-1) keys. For example, if the value of m is 5 then the maximum value of keys is 4.
- The root node has minimum one key, whereas all the other nodes except the root node have (ceiling of m/2 minus - 1) minimum keys.
- If we perform insertion in the B tree, then the node is always inserted in the leaf node.

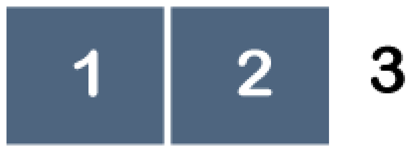
Suppose we want to create a B tree of order 3 by inserting values from 1 to 10.

Step 1: First, we create a node with 1 value as shown below:

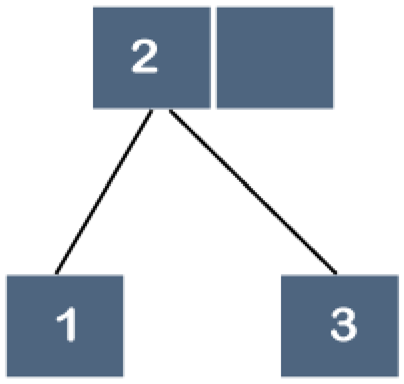
Step 2: The next element is 2, which comes after 1 as shown below:



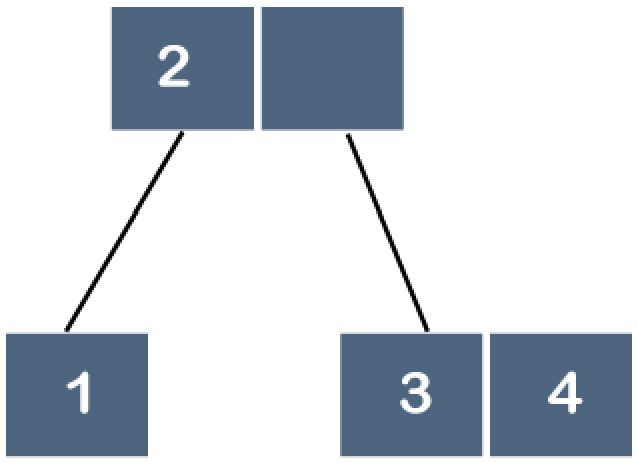
Step 3: The next element is 3, and it is inserted after 2 as shown below:



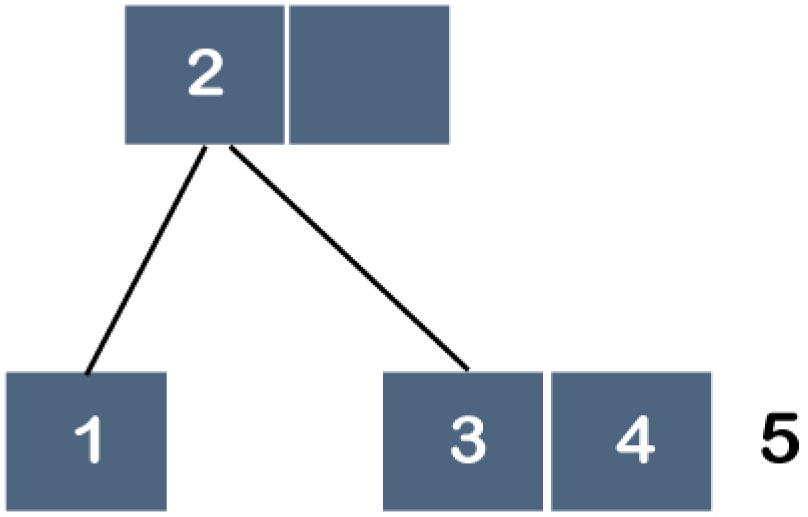
As we know that each node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 2, so it moves to its parent. The node 2 does not have any parent, so it will become the root node as shown below:



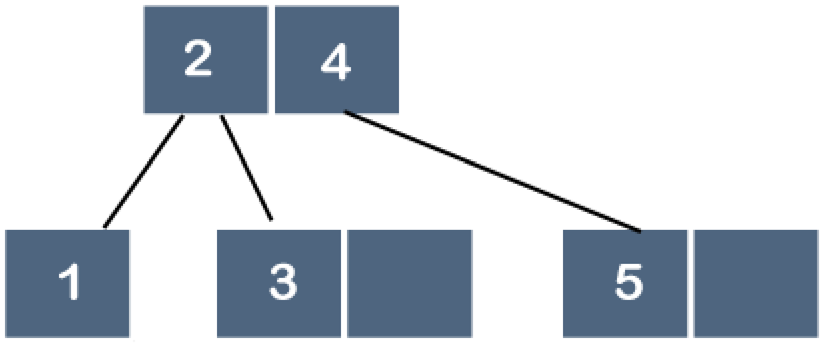
Step 4: The next element is 4. Since 4 is greater than 2 and 3, so it will be added after the 3 as shown below:



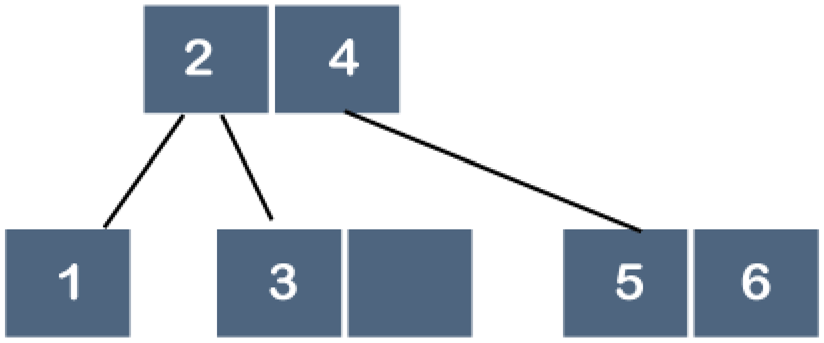
Step 5: The next element is 5. Since 5 is greater than 2, 3 and 4 so it will be added after 4 as shown below:



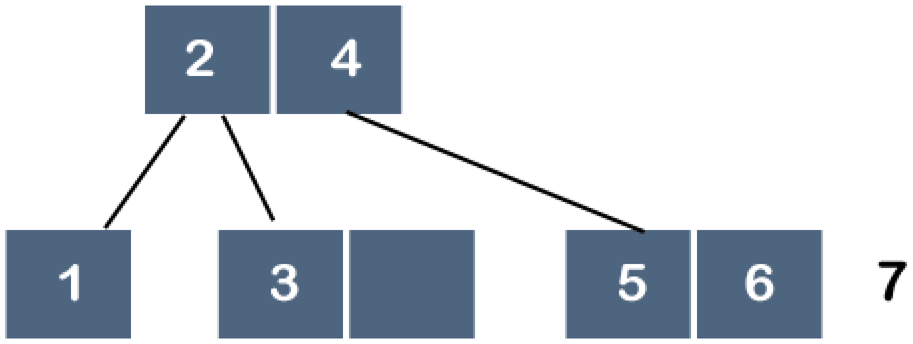
As we know that each node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 4, so it moves to its parent. The parent is node 2; therefore, 4 will be added after 2 as shown below:



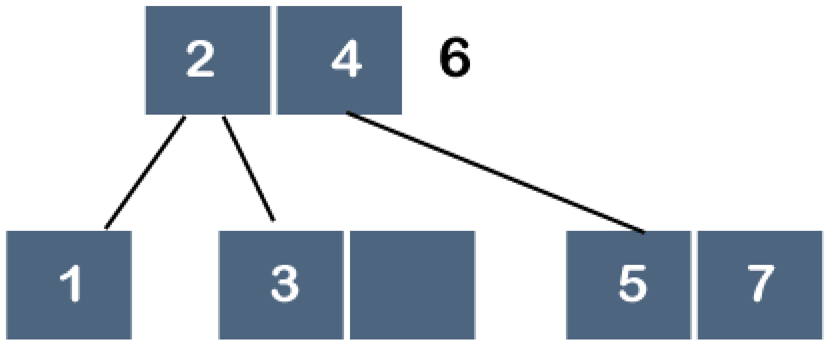
Step 6: The next element is 6. Since 6 is greater than 2, 4 and 5, so 6 will come after 5 as shown below:



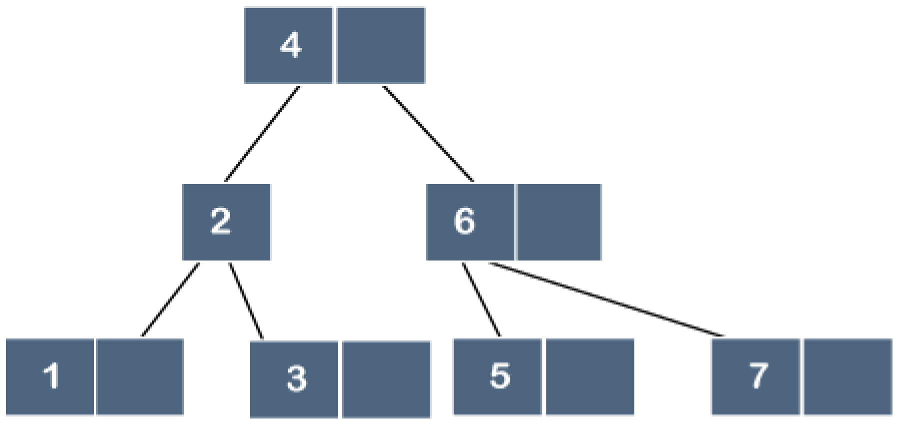
Step 7: The next element is 7. Since 7 is greater than 2, 4, 5 and 6, so 7 will come after 6 as shown below:



As we know that each node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 6, so it moves to its parent as shown below:



But, 6 cannot be added after 4 because the node can have 2 maximum keys, so we will split this node through the middle element. The middle element is 4, so it moves to its parent. As node 4 does not have any parent, node 4 will become a root node as shown below:



Insertion into a B-tree

The condition that all leaves must be on the same level forces a characteristic behavior of B-trees, namely that B-trees are not allowed to grow at the their leaves; instead they are forced to grow at the root.

When inserting into a B-tree, a value is inserted directly into a leaf. This leads to three common situations that can occur:

1. A key is placed into a leaf that still has room.
2. The leaf in which a key is to be placed is full.
3. The root of the B-tree is full.

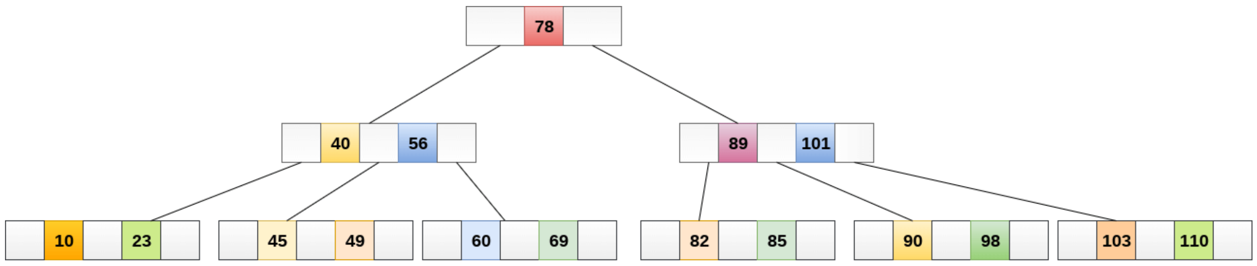
Operations

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.



Inserting

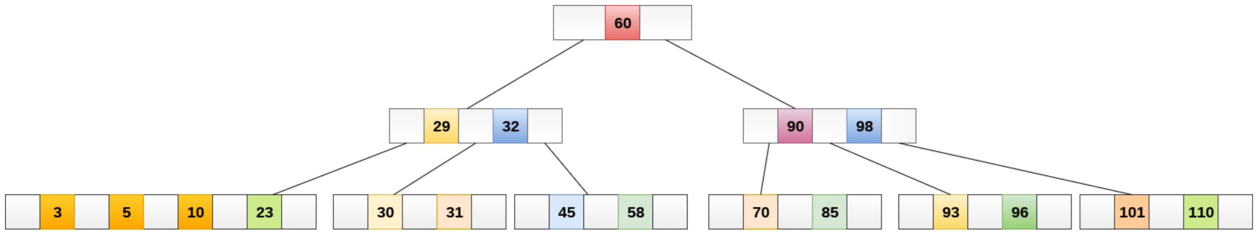
Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.

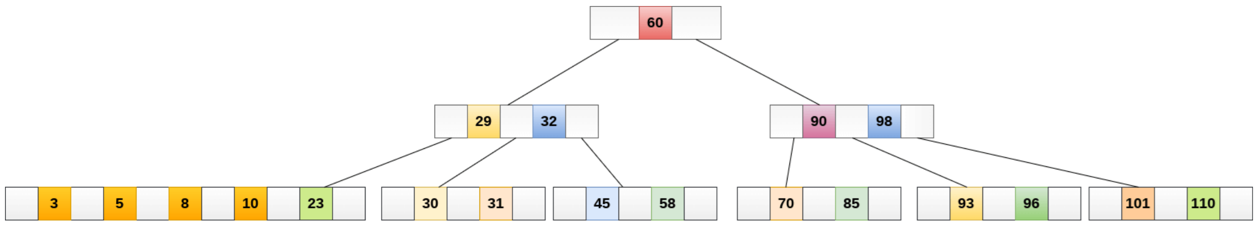
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
   - Insert the new element in the increasing order of elements.
   - Split the node into the two nodes at the median.
   - Push the median element upto its parent node.
   - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

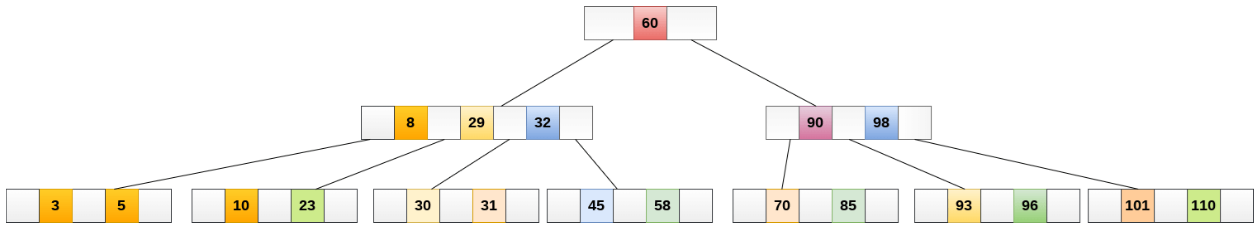
Insert the node 8 into the B Tree of order 5 shown in the following image.



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



The node, now contain 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

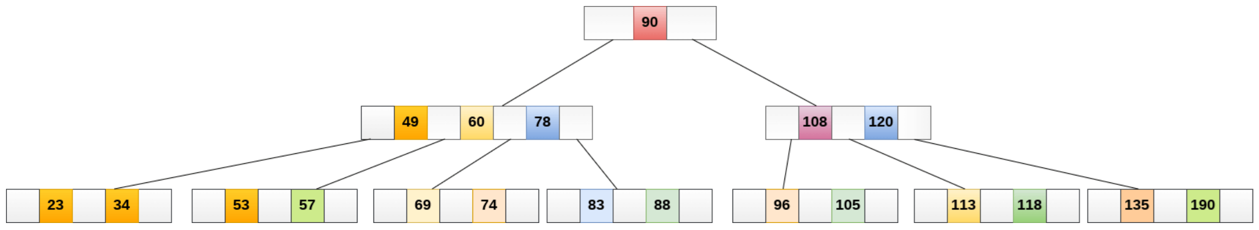
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

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 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 intervening element down to the node where the key is deleted.
4. If neither of the sibling 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 parent is left with less than m/2 nodes then, apply the above process on the parent too.

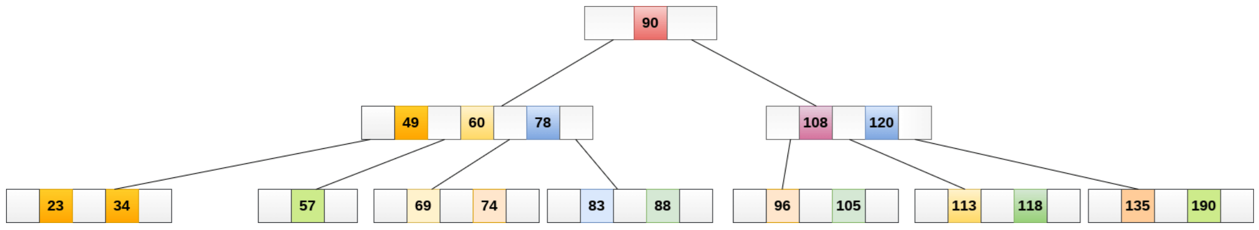
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

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

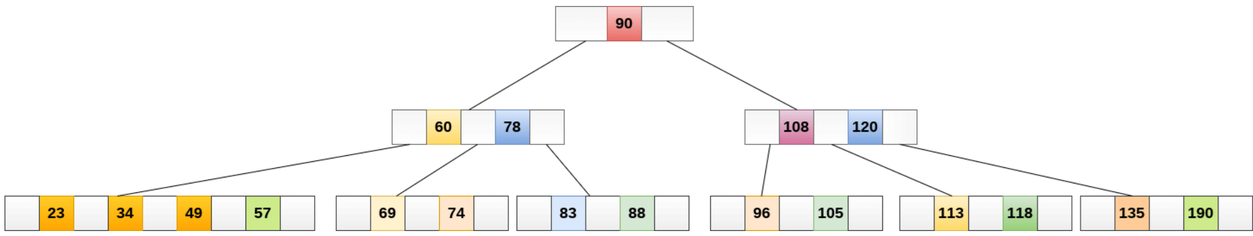


53 is present in the right child of element 49. Delete it.



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 sub-tree 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.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

B+ Tree

B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes.

Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compared to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.

B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.

5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Hashing

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.
2. Linked List of phone numbers and records.
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With **balanced binary search tree**, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(Logn) time.

Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in O(1) time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.

This solution has many practical limitations. First problem with this solution is extra space required is huge. For example if phone number is n digits, we need $O(m * 10^n)$ space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.

Due to above limitations Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

Hash Function**:** A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.

A good hash function should have following properties

1) Efficiently computable.

2) Should uniformly distribute the keys (Each table position equally likely for each key)

For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table**:** An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- **Chaining:**The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
- **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

What is Collision?

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

What are the chances of collisions with large table?

Collisions are very likely even if we have big table to store keys. An important observation is Birthday Paradox. With only 23 persons, the probability that two people have the same birthday is 50%.

How to handle Collisions?

There are mainly two methods to handle collision:

1) Separate Chaining

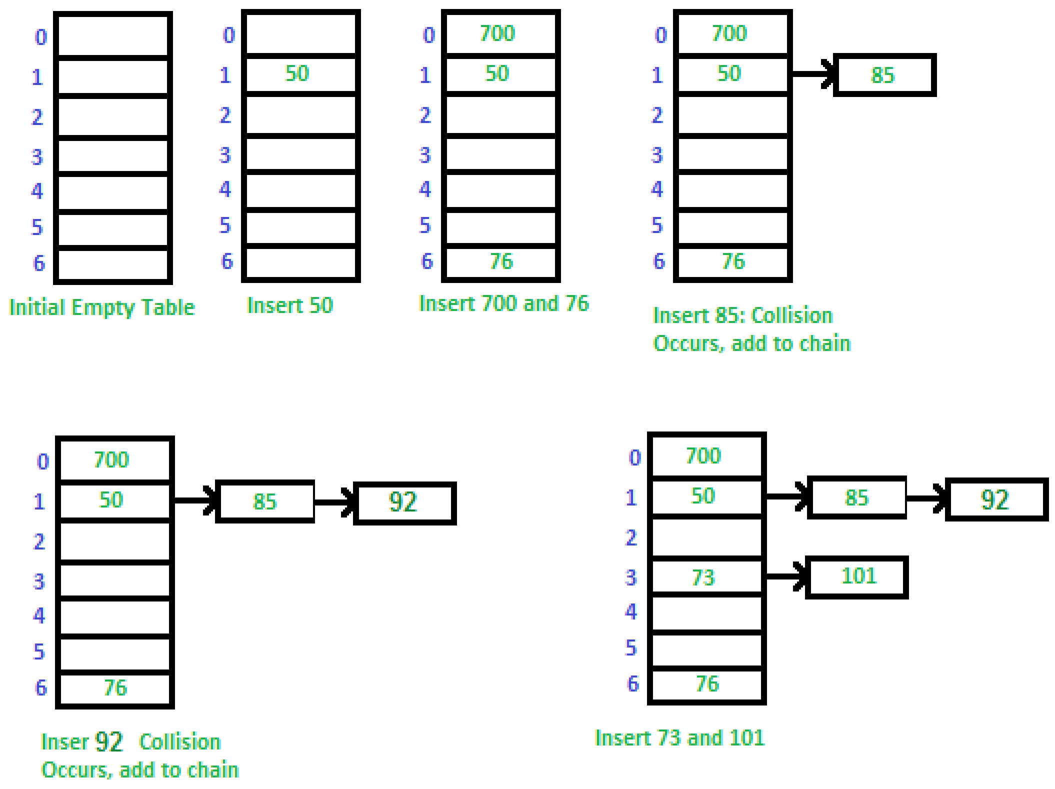
2) Open Addressing

In this article, only separate chaining is discussed. We will be discussing Open addressing in the next post.

Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as “**key mod 7**” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Advantages:

1) Simple to implement.

2) Hash table never fills up, we can always add more elements to the chain.

3) Less sensitive to the hash function or load factors.

4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.

2) Wastage of Space (Some Parts of hash table are never used)

3) If the chain becomes long, then search time can become O(n) in the worst case.

4) Uses extra space for links.

Performance of Chaining:

Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing).

```
m = Number of slots in hash table
 n = Number of keys to be inserted in hash table

 Load factor α = n/m

 Expected time to search = O(1 + α)

 Expected time to delete = O(1 + α)

Time to insert = O(1)

 Time complexity of search insert and delete is
 O(1) if  α is O(1)
```

Data Structures For Storing Chains:

- Linked lists
  - Search: O(l) where l = length of linked list
  - Delete: O(l)
  - Insert: O(l)
  - Not cache friendly
- Dynamic Sized Arrays (Vectors in C++, ArrayList in Java, list in Python)
  - Search: O(l) where l = length of array
  - Delete: O(l)
  - Insert: O(l)
  - Cache friendly
- Self Balancing BST (AVL Trees, Red Black Trees)
  - Search: O(log(l))

    - Delete: O(log(l))
    - Insert: O(l)
    - Not cache friendly
    - Java 8 onwards use this for HashMap

Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete(k): ***Delete operation is interesting***. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted".

The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

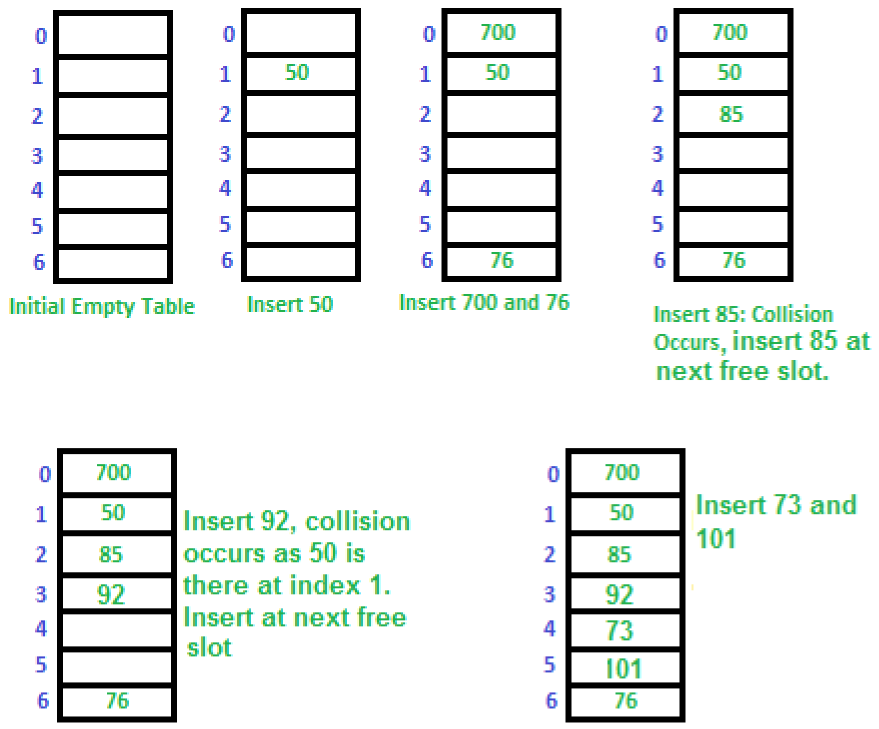
Open Addressing is done in the following ways:

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, the typical gap between two probes is 1 as seen in the example below.

Let **hash(x)** be the slot index computed using a hash function and **S** be the table size

```
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
..................................................
..................................................
```

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.



0
1
2
3
4
5
6
Initial Empty Table
50
Insert 50
700
76
Insert 700 and 76
85
Insert 85: Collision Occurs, insert 85 at next free slot.
92
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
73
101
Insert 73 and 101

Challenges in Linear Probing :

1. **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
2. **Secondary Clustering*:*** Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

b) Quadratic Probing We look for i2‘th slot in i’th iteration.

```
let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2)
% S
If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3)
% S
..................................................
..................................................
```

c) Double Hashing We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

```
let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x))
% S
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x)
+ 2*hash2(x)) % S
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x)
+ 3*hash2(x)) % S
..................................................
..................................................
```

See this for step by step diagrams.

Comparison of above three:

Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

S.No.	*Separate Chaining*	*Open Addressing*
1.	*Chaining is Simpler to implement.*	*Open Addressing requires more computation.*
2.	*In chaining, Hash table never fills up, we can always add more elements to chain.*	*In open addressing, table may become full.*
3.	*Chaining is Less sensitive to the hash function or load factors.*	*Open addressing requires extra care to avoid clustering and load factor.*

4.	*Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.*	*Open addressing is used when the frequency and number of keys is known.*
5.	*Cache performance of chaining is not good as keys are stored using linked list.*	*Open addressing provides better cache performance as everything is stored in the same table.*
6.	*Wastage of Space (Some Parts of hash table in chaining are never used).*	*In Open addressing, a slot can be used even if an input doesn't map to it.*
7.	*Chaining uses extra space for links.*	*No links in Open addressing*

Performance of Open Addressing:

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

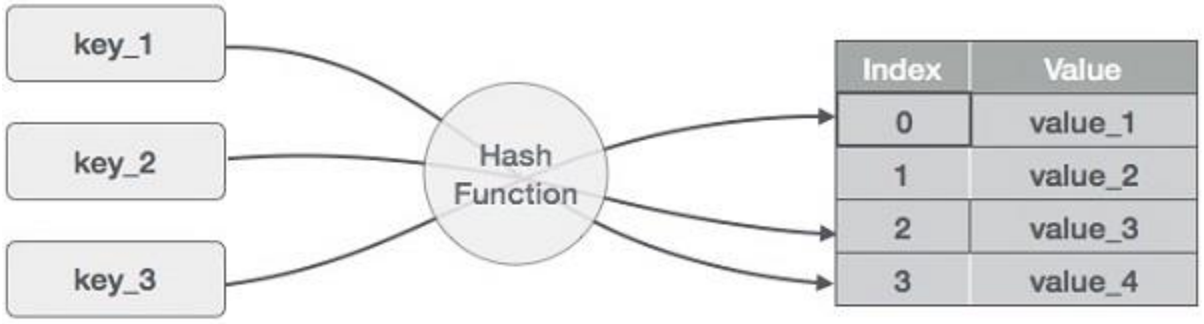
```
m = Number of slots in the hash table
n = Number of keys to be inserted in the hash table

Load factor α = n/m  ( < 1 )

Expected time to search/insert/delete < 1/(1 - α)

So Search, Insert and Delete take (1/(1 - α)) time
```

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12

6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12

6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are the basic primary operations of a hash table.

- Search − Searches an element in a hash table.
- Insert − inserts an element in a hash table.
- delete − Deletes an element from a hash table.

DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {
   int data;
   int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
   return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Example

```
struct DataItem *search(int key) {
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL) {

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}
```

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Example

```
void insert(int key,int data) {
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct
DataItem));
   item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key
!= -1) {
      //go to next cell
      ++hashIndex;
		
      //wrap around the table
      hashIndex %= SIZE;
   }
	
   hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Example

```
struct DataItem* delete(struct DataItem* item) {
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] !=NULL) {
	
      if(hashArray[hashIndex]->key == key) {
         struct DataItem* temp = hashArray[hashIndex];
			
         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }
		
      //go to next cell
      ++hashIndex;
		
      //wrap around the table
      hashIndex %= SIZE;
   }
```

```

    return NULL;
}
```

Load Factor and Rehashing

How hashing works:

For **insertion** of a key(K) – value(V) pair into a hash map, 2 steps are required:

1. K is converted into a small integer (called its hash code) using a hash function.
2. The hash code is used to find an index (hashCode % arrSize) and the entire linked list at that index(Separate chaining) is first searched for the presence of the K already.
3. If found, it's value is updated and if not, the K-V pair is stored as a new node in the list.

Complexity and Load Factor

- For the **first step**, time taken depends on the K and the hash function. For example, if the key is a string "abcd", then it's hash function may depend on the length of the string. But for very large values of n, the number of entries into the map, length of the keys is almost negligible in comparison to n so hash computation can be considered to take place in constant time, i.e, **O(1)**.

- For the **second step**, traversal of the list of K-V pairs present at that index needs to be done. For this, the worst case may be that all the n entries are at the same index. So, time complexity would be **O(n)**. But, enough research has been done to make hash functions uniformly distribute the keys in the array so this almost never happens.
- So, **on an average**, if there are n entries and b is the size of the array there would be n/b entries on each index. This value n/b is called the **load factor** that represents the load that is there on our map.
- This Load Factor needs to be kept low, so that number of entries at one index is less and so is the complexity almost constant, i.e., O(1).

Rehashing: As the name suggests, **rehashing means hashing again**. Basically, when the load factor increases to more than its pre-defined value (default value of load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity.

Why rehashing?

Rehashing is done because whenever key value pairs are inserted into the map, the load factor increases, which implies that the time complexity also increases as explained above. This might not give the required time complexity of O(1).Hence, rehash must be done, increasing the size of the bucketArray so as to reduce the load factor and the time complexity.

How Rehashing is done?

Rehashing can be done as follows:

- For each addition of a new entry to the map, check the load factor.
- If it’s greater than its pre-defined value (or default value of 0.75 if not given), then Rehash.
- For Rehash, make a new array of double the previous size and make it the new bucketarray.
- Then traverse to each element in the old bucketArray and call the insert() for each so as to insert it into the new larger bucket array.

Why Hashing ?

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.
2. Linked List of phone numbers and records.
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With **balanced binary search tree**, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(Logn) time.

Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in O(1) time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.

This solution has many practical limitations. First problem with this solution is extra space required is huge. For example if phone number is n digits, we need O(m * 10^n) space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.

Due to above limitations Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

Hash Function**:** A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.

A good hash function should have following properties

1) Efficiently computable.

2) Should uniformly distribute the keys (Each table position equally likely for each key)

For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table**:** An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Heap Data Structure

A Heap is a special Tree-based data structure in which the tree is a complete binary tree.

Note: The heap tree is a special balanced binary tree data structure where the root node is compared with its children and arrange accordingly.

Heap data structure is a complete binary tree that satisfies the heap property. It is also called as a binary heap. A complete binary tree is a special binary tree in which

Ø every level, except possibly the last, is filled

Ø all the nodes are as far left as possible

Complete Binary Tree

Generally, Heaps can be of two types:

1. **Max-Heap**: In a Max-Heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min-Heap**: In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If **α** has child node **β** then −

$$\mathbf{key(\alpha) \geq key(\beta)}$$

As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types −

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap − Where the value of the root node is less than or equal to either of its children.

Max-Heap − Where the value of the root node is greater than or equal to either of its children.

Both trees are constructed using the same input and order of arrival.

Array Representation

A binary heap is represented as an array. The representation follows some property.

The root of the tree will be at Arr[0].

For any node at Arr[i], its left and right children will be at Arr[2*i + 1] and Arr[2*i+2] respectively.

For any node at Arr[i], its parent node will be at Arr[(i-1)/2].

Why Array?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and array-based representation is space-efficient.

Level Order Traversal of the heap will give the order in which elements are filled in the array.

Applications of Heap

1. **Heap Sort**: Heap Sort uses Binary Heap to sort an array in O(nlogn).
2. **Priority Queue:** It uses binary heap to efficient implement operations insert(), delete(), extract_max(), update() in O(logn) time.
3. **Graph Algorithms:** Some of the Graph Algorithms also use heaps to reduce its time complexity like Dijkstra's Algorithm and Minimum Spanning Tree.
4. **K-way merge:** A heap data structure is useful to merge many already-sorted input streams into a single sorted output stream.
5. **Order statistics**: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.

Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

Step 1 − Remove root node.

Step 2 − Move the last element of last level to root.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Time complexity in Max Heap

The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete binary tree is always logn; therefore, the time complexity would also be O(logn).

Algorithm of insert operation in the max heap.

```
// algorithm to insert an element in the max heap.
insertHeap(A, n, value)
{
        n=n+1; // n is incremented to insert the new element
        A[n]=value; // assign new value at the nth position
        i = n; // assign the value of n to i
        // loop will be executed until i becomes 1.
        while(i>1)
        {  parent= floor value of i/2; // Calculating the floor value of i/2
        // Condition to check whether the value of parent is less than the given node or not
        if(A[parent]<A[i])
        {
        swap(A[parent], A[i]);
        i = parent;
        }
        else
        {
        return;
        }
        }
    }
```

Heapify

Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

Algorithm to heapify the tree

```
    MaxHeapify(A, n, i)
    {
    int largest =i;
    int l= 2i;
    int r= 2i+1;
    while(l<=n && A[l]>A[largest])
    {
    largest=l;
    }
```

```
while(r<=n && A[r]>A[largest])
{
largest=r;
}
if(largest!=i)
{
swap(A[largest], A[i]);
heapify(A, n, largest);
}}
```

Peek (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

```
return rootNode
```

Extract-Max/Min

Extract-Max returns the node with maximum value after removing it from a Max Heap whereas Extract-Min returns the node with minimum after removing it from Min Heap.

Time complexity Analysis

1. Complexity Analysis of Insert operation in Max Heap

When a node is supposed to add into the heap, the element is added at the next vacant index of the array. Then it is checked whether the inserted child node is in accordance with the parent node or not. If the child has a higher value than the parent, the swapping of the nodes is done. This swapping process goes on until the properties of Max Heap are fulfilled.

Complexity of adding a node is: **O(1)**
Complexity of swapping the nodes(heapify): **O(H) [where H is height of the tree]**
(swapping will be done H times in the worst case scenario)

Total complexity**: O(1) + O(H) = O(H)**
For a Complete Binary tree, its height H = **O(log N)**, where N represents total no. of nodes.

Therefore, Overall Complexity of insert operation is O(log N).

2. Complexity Analysis of Delete operation in min heap

Deletion of a node cannot be done randomly. The element with the highest priority (i.e. parent) will be deleted first followed by the next node in order of priority. This is why heap is called a priority queue.

First, swap the positions of the parent node and leaf node, and then remove the newly formed leaf node (which was originally the parent) from the queue. Next, start the swapping process so that the new parent node is placed in the right position in accordance with the properties of Min Heap.

If a node is to be deleted from a heap with height **H**:

Complexity of swapping parent node and leaf node is: **O(1)**
Complexity of swapping the nodes(heapify): **O(H)**
(swapping will be done H times in the worst case scenario)
Total complexity: **O(1) + O(H) = O(H)**
For a Complete Binary tree, its height H = O(log N), where N represents total no. of nodes.

Therefore, Overall Complexity of delete operation is O(log N).

3. Complexity of getting the Maximum value from max heap

In order to obtain the maximum value just return the value of the root node (which is the greatest element in Max Heap), so simply return the element at index 0 of the array.

Hence, Complexity of getting minimum value is: **O(1)**

Deaps (double-ended-priority-queues: deaps)

Deap is defined as a data structure which has no element or key value at the root node. It is formed by implementing the following rules −

- There is no element at root node that indicates root node is empty.
- Left subtree of the deap shall indicate min heap.
- Right subtree of deap shall indicate max heap.

A deap is a complete binary tree that is either empty or satisfies the following conditions:

1. The root is empty.

2. The left subtree is a min heap and the right subtree is a max heap.

3. Correspondence property. Suppose that the right subtree is not empty. For every node x in the left subtree, define its corresponding node y in the right subtree to be the node in the same position as x. In case such a y doesn't exist, let y be the corresponding node for the parent of x. The element in x is ≤ the element in y.

Thus, correctness to the following statement can be provided mathematically by a deap structure −

If the left sub tree and right sub tree of certain nodes are non-empty, and their corresponding nodes can be represented by 'a' and 'b' respectively, then −

a.KeyValue <= b.KeyValue

Correspondence property

The complete binary tree above , the corresponding nodes for the nodes with 3,7,5,9,15,11,12, respectively have 20,18,16,10,18,16, and 16.

Inserting an element in a deap

When an element is inserted into an n-element deap , we go from a complete binary tree that has n+1 nodes to one that has n+2 nodes. So the shape of the new deap is well defined.

The new node is j and its corresponding node is i.

To insert new element temporarily place new element into the new node j and check the correspondence property for node j. If the property is not satisfied, swap new element and the element in its corresponding node ; use a trickle up process to correctly position new element in the heap for the corresponding node i. If the correspondence property is satisfied, do not swap new element ; instead use a trickle up process to correctly place new element in the heap that contains node j.

That is;

1. Consider the insertion of new element ‘2’ into our deap.

2. The element in the corresponding node ‘i’ is 15

3. Since the correspondence property is not satisfied, we swap 2 and 15

4. Node j now contains 15 and this swap is guaranteed to preserve the max heap properties of the right subtree of the complete binary tree

5. To correctly position the 2 in the left subtree , we use the standard min heap trickle process beginning at node ‘i’ .

This is how the result will look like,

Deletion of the min element

Assume that n>0 .The min element is in the root of the min heap. Following its removal , the deap size reduces to n-1 and the element in position n+1 of the deap array is dropped from the deap.

In our example, the min element 3 is removed and 10 is dropped. To reinsert the dropped element we first trickle the vacancy in the root of the min heap down to a leaf of the min heap. The trickle down causes 5 and 11 to, respectively, move to their present nodes. Then the dropped element 10 is inserted using a trickle up process beginning at the vacant leaf of the min heap. Since a remove min requires a trickle – down pass followed by a trickle – up pass and since the height of a deap is O(log n), the time for a remove min is O(log n).

Splay Tree Data structure

Splay tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree. A splay tree is defined as follows...

Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called **"Splaying"**.

Splaying an element, is the process of bringing it to the root position by performing suitable rotation operations.

Ø In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.

Ø By splaying elements we bring more frequently used elements closer to the root of the tree so that any operation on those elements is performed quickly. That means the splaying operation automatically brings more frequently used elements closer to the root of the tree.

Every operation on splay tree performs the splaying operation. For example, the insertion operation first inserts the new element using the binary search tree insertion process, then the newly inserted element is splayed so that it is placed at the root of the tree.

Ø The search operation in a splay tree is nothing but searching the element using binary search process and then splaying that searched element so that it is placed at the root of the tree.

In splay tree, to splay any element we use the following rotation operations...

Rotations in Splay Tree

- **1. Zig Rotation**
- **2. Zag Rotation**
- **3. Zig - Zig Rotation**
- **4. Zag - Zag Rotation**

- **5. Zig - Zag Rotation**
- **6. Zag - Zig Rotation**

Factors required for selecting a type of rotation

The following are the factors used for selecting a type of rotation:

- Does the node which we are trying to rotate have a grandparent?
- Is the node left or right child of the parent?
- Is the node left or right child of the grandparent?

Cases for the Rotations

Case 1: If the node does not have a grand-parent, and if it is the right child of the parent, then we carry out the left rotation; otherwise, the right rotation is performed.

Case 2: If the node has a grandparent, then based on the following scenarios; the rotation would be performed:

Scenario 1: If the node is the right of the parent and the parent is also right of its parent, then ***zag zag left left rotation*** is performed.

Scenario 2: If the node is left of a parent, but the parent is right of its parent, then ***zig zag right left rotation*** is performed.

Scenario 3: If the node is left of the parent and the parent is left of its parent, then ***zig zig right right rotation*** is performed.

Scenario 4: If the node is right of a parent, but the parent is left of its parent, then ***zag zig left right rotation*** is performed.

Zig Rotation

The **Zig Rotation** in splay tree is similar to the single right rotation in AVL Tree rotations. In zig rotation, every node moves one position to the right from its current position. Consider the following example...

Zag Rotation

The **Zag Rotation** in splay tree is similar to the single left rotation in AVL Tree rotations. In zag rotation, every node moves one position to the left from its current position. Consider the following example...

Zig-Zig Rotation

The **Zig-Zig Rotation** in splay tree is a double zig rotation. In zig-zig rotation, every node moves two positions to the right from its current position. Consider the following example...

Zag-Zag Rotation

The **Zag-Zag Rotation** in splay tree is a double zag rotation. In zag-zag rotation, every node moves two positions to the left from its current position. Consider the following example...

Zig-Zag Rotation

The **Zig-Zag Rotation** in splay tree is a sequence of zig rotation followed by zag rotation. In zig-zag rotation, every node moves one position to the right followed by one position to the left from its current position. Consider the following example...

Zag-Zig Rotation

The **Zag-Zig Rotation** in splay tree is a sequence of zag rotation followed by zig rotation. In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position. Consider the following example...

Every Splay tree must be a binary search tree but it is need not to be balanced tree.

Insertion Operation in Splay Tree

The insertion operation in Splay tree is performed using following steps...

- **Step 1 -** Check whether tree is Empty.
- **Step 2 -** If tree is Empty then insert the **newNode** as Root node and exit from the operation.
- **Step 3 -** If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.
- **Step 4 -** After insertion, **Splay** the **newNode**

Deletion Operation in Splay Tree

The deletion operation in splay tree is similar to deletion operation in Binary Search Tree. But before deleting the element, we first need to **splay** that element and then delete it from the root position. Finally join the remaining tree using binary search tree logic.

Complexity

As we already know, the time complexity of a binary search tree in every case. The time complexity of a binary search tree in the average case is **O(logn)** and the time complexity in the worst case is O(n). In a binary search tree, the value of the left subtree is smaller than the root node, and the value of the right subtree is greater than the root node; in such case, the time complexity would be **O(logn)**. If the binary tree is left-skewed or right-skewed, then the time complexity would be O(n). To limit the skewness, the AVL and Red-Black tree came into the picture, having **O(logn**) time complexity for all the operations in all the cases. We can also improve this time complexity by doing more practical implementations, so the new Tree data structure was designed, known as a Splay tree.

Advantages of Splay tree

- In the splay tree, we do not need to store the extra information. In contrast, in AVL trees, we need to store the balance factor of each node that requires extra space, and Red-Black trees also require to store one extra bit of information that denotes the color of the node, either Red or Black.

- It is the fastest type of Binary Search tree for various practical applications. It is used in **Windows NT and GCC compilers**.
- It provides better performance as the frequently accessed nodes will move nearer to the root node, due to which the elements can be accessed quickly in splay trees. It is used in the cache implementation as the recently accessed data is stored in the cache so that we do not need to go to the memory for accessing the data, and it takes less time.

Drawback of Splay tree

The major drawback of the splay tree would be that trees are not strictly balanced, i.e., they are roughly balanced. Sometimes the splay trees are linear, so it will take O(n) time complexity.

Example - Insertion operation in Splay tree

In the ***insertion*** operation, we first insert the element in the tree and then perform the splaying operation on the inserted element.

15, 10, 17, 7

Step 1: First, we insert node 15 in the tree. After insertion, we need to perform splaying. As 15 is a root node, so we do not need to perform splaying.

Step 2: The next element is 10. As 10 is less than 15, so node 10 will be the left child of node 15, as shown below:

Now, we perform ***splaying***. To make 10 as a root node, we will perform the right rotation, as shown below:

Step 3: The next element is 17. As 17 is greater than 10 and 15 so it will become the right child of node 15.

Now, we will perform splaying. As 17 is having a parent as well as a grandparent so we will perform zag zag rotations.

In the above figure, we can observe that 17 becomes the root node of the tree; therefore, the insertion is completed.

Step 4: The next element is 7. As 7 is less than 17, 15, and 10, so node 7 will be left child of 10.

Now, we have to splay the tree. As 7 is having a parent as well as a grandparent so we will perform two right rotations as shown below:

Still the node 7 is not a root node, it is a left child of the root node, i.e., 17. So, we need to perform one more right rotation to make node 7 as a root node as shown below:

Algorithm for Insertion operation

```
1.  Insert(T, n)
2.  temp= T_root
3.  y=NULL
4.  while(temp!=NULL)
5.  y=temp
6.  if(n->data <temp->data)
7.  temp=temp->left
8.  else
9.  temp=temp->right
10. n.parent= y
11. if(y==NULL)
12. T_root = n
13. else if (n->data < y->data)
14. y->left = n
15. else
16. y->right = n
17. Splay(T, n)
```

In the above algorithm, T is the tree and n is the node which we want to insert. We have created a temp variable that contains the address of the root node. We will run the while loop until the value of temp becomes NULL.

Once the insertion is completed, splaying would be performed

Algorithm for Splaying operation

```
1.  Splay(T, N)
```

```
while(n->parent !=Null)
if(n->parent==T->root)
if(n==n->parent->left)
right_rotation(T, n->parent)
else
left_rotation(T, n->parent)
else
p= n->parent
g = p->parent
if(n=n->parent->left && p=p->parent->left)
right.rotation(T, g), right.rotation(T, p)
else if(n=n->parent->right && p=p->parent->right)
left.rotation(T, g), left.rotation(T, p)
else if(n=n->parent->left && p=p->parent->right)
right.rotation(T, p), left.rotation(T, g)
else
left.rotation(T, p), right.rotation(T, g)

Implementation of right.rotation(T, x)
right.rotation(T, x)
y= x->left
x->left=y->right
y->right=x
return y
```

In the above implementation, x is the node on which the rotation is performed, whereas y is the left child of the node x.

Implementation of left.rotation(T, x)

```
left.rotation(T, x)
```

```
2.  y=x->right
3.  x->right = y->left
4.  y->left = x
5.  return y
```

In the above implementation, x is the node on which the rotation is performed and y is the right child of the node x.

Deletion in Splay tree

As we know that splay trees are the variants of the Binary search tree, so deletion operation in the splay tree would be similar to the BST, but the only difference is that the delete operation is followed in splay trees by the splaying operation.

Types of Deletions:

There are two types of deletions in the splay trees:

1. Bottom-up splaying
2. Top-down splaying

Bottom-up splaying

In bottom-up splaying, first we delete the element from the tree and then we perform the splaying on the deleted node.

Let's understand the deletion in the Splay tree.

Suppose we want to delete 12, 14 from the tree shown below:

- First, we simply perform the standard BST deletion operation to delete 12 element. As 12 is a leaf node, so we simply delete the node from the tree.

The deletion is still not completed. We need to splay the parent of the deleted node, i.e., 10. We have to perform ***Splay(10)*** on the tree. As we can observe in the above tree that 10 is at the right of node 7, and node 7 is at the left of node 13. So, first, we perform the left rotation on node 7 and then we perform the right rotation on node 13, as shown below:

Still, node 10 is not a root node; node 10 is the left child of the root node. So, we need to perform the right rotation on the root node, i.e., 14 to make node 10 a root node as shown below:

- Now, we have to delete the 14 element from the tree, which is shown below:

As we know that we cannot simply delete the internal node. We will replace the value of the node either using ***inorder predecessor*** or ***inorder successor***. Suppose we use inorder successor in which we replace the value with the lowest value that exist in the right subtree. The lowest value in the right subtree of node 14 is 15, so we replace the value 14 with 15. Since node 14 becomes the leaf node, so we can simply delete it as shown below:

Still, the deletion is not completed. We need to perform one more operation, i.e., splaying in which we need to make the parent of the deleted node as the root node. Before deletion, the parent of node 14 was the root node, i.e., 10, so we do need to perform any splaying in this case.

Top-down splaying

In top-down splaying, we first perform the splaying on which the deletion is to be performed and then delete the node from the tree. Once the element is deleted, we will perform the join operation.

Let's understand the top-down splaying through an example.

Suppose we want to delete 16 from the tree which is shown below:

Step 1: In top-down splaying, first we perform splaying on the node 16. The node 16 has both parent as well as grandparent. The node 16 is at the right of its parent and the parent node is also at the right of its parent, so this is a zag zag situation. In this case, first, we will perform the left rotation on node 13 and then 14 as shown below:

The node 16 is still not a root node, and it is a right child of the root node, so we need to perform left rotation on the node 12 to make node 16 as a root node.

Once the node 16 becomes a root node, we will delete the node 16 and we will get two different trees, i.e., left subtree and right subtree as shown below:

As we know that the values of the left subtree are always lesser than the values of the right subtree. The root of the left subtree is 12 and the root of the right subtree is 17. The first step is to find the maximum element in the left subtree. In the left subtree, the maximum element is 15, and then we need to perform splaying operation on 15.

As we can observe in the above tree that the element 15 is having a parent as well as a grandparent. A node is right of its parent, and the parent node is also right of its parent, so we need to perform two left rotations to make node 15 a root node as shown below:

After performing two rotations on the tree, node 15 becomes the root node. As we can see, the right child of the 15 is NULL, so we attach node 17 at the right part of the 15 as shown below, and this operation is known as a ***join*** operation.

Note: If the element is not present in the splay tree, which is to be deleted, then splaying would be performed. The splaying would be performed on the last accessed element before reaching the NULL.

Algorithm of Delete operation

```
1.  If(root==NULL)
2.  return NULL
3.  Splay(root, data)
4.  If data!= root->data
5.  Element is not present
6.  If root->left==NULL
7.  root=root->right
8.  else
9.  temp=root
10. Splay(root->left, data)
11. root1->right=root->right
12. free(temp)
13. return root
```

In the above algorithm, we first check whether the root is Null or not; if the root is NULL means that the tree is empty. If the tree is not empty, we will perform the splaying operation on the element which is to be deleted. Once the splaying operation is completed, we will compare the root data with the element which is to be deleted; if both are not equal means that the element is not present in the tree. If they are equal, then the following cases can occur:

Case 1: The left of the root is NULL, the right of the root becomes the root node.

Case 2: If both left and right exist, then we splay the maximum element in the left subtree. When the splaying is completed, the maximum element becomes the root of the left subtree. The right subtree would become the right child of the root of the left subtree.

https://youtu.be/qMmqOHr75b8

Leftist Tree / Leftist Heap

A leftist tree or leftist heap is a priority queue implemented with a variant of a binary heap. Every node has an **s-value (or rank or distance)** which is the distance to the nearest leaf. In contrast to a binary heap (Which is always a complete binary tree), a leftist tree may be very unbalanced.

Below are time complexities of **Leftist Tree / Heap**.

Function	**Complexity**	**Comparison**
1) Get Min:	**O(1)**	[same as both Binary and Binomial]
2) Delete Min:	**O(Log n)**	[same as both Binary and Binomial]
3) Insert:	**O(Log n)**	[O(Log n) in Binary and O(1) in Binomial and O(Log n) for worst case]
4) Merge:	**O(Log n)**	[O(Log n) in Binomial]

A leftist tree is a binary tree with properties:

1. **Normal Min Heap Property :** key(i) >= key(parent(i))
2. **Heavier on left side :** dist(right(i)) <= dist(left(i)). Here, dist(i) is the number of edges on the shortest path from node i to a leaf node in extended binary tree representation (In this representation, a null child is considered as external or leaf node). The shortest path to a descendant external node is through the right child. Every subtree is also a leftist tree and dist(i) = 1 + dist(right(i)).

Example: The below leftist tree is presented with its distance calculated for each node with the procedure mentioned above. The rightmost node has a rank of 0 as the right sub tree of this node is null and its parent has a distance of 1 by dist(i) = 1 + dist(right(i)). The same is followed for each node and their s-value(or rank) is calculated.

From above second property, we can draw two conclusions :

1. The path from root to rightmost leaf is the shortest path from root to a leaf.
2. If the path to rightmost leaf has x nodes, then leftist heap has atleast 2^x – 1 nodes. This means the length of path to rightmost leaf is O(log n) for a leftist heap with n nodes.

Operations :

1. The main operation is merge().
2. deleteMin() (or extractMin() can be done by removing root and calling merge() for left and right subtrees.
3. insert() can be done be create a leftist tree with single key (key to be inserted) and calling merge() for given tree and tree with single node.

Idea behind Merging :

Since right subtree is smaller, the idea is to merge right subtree of a tree with other tree. Below are abstract steps.

1. Put the root with smaller value as the new root.
2. Hang its left subtree on the left.
3. Recursively merge its right subtree and the other tree.
4. Before returning from recursion:

– Update dist() of merged root.
– Swap left and right subtrees just below root, if needed, to keep leftist property of merged result

Source : http://courses.cs.washington.edu/courses/cse326/08sp/lectures/05-leftist-heaps.pdf

Detailed Steps for Merge:

1. Compare the roots of two heaps.
2. Push the smaller key into an empty stack, and move to the right child of smaller key.
3. Recursively compare two keys and go on pushing the smaller key onto the stack and move to its right child.
4. Repeat until a null node is reached.
5. Take the last node processed and make it the right child of the node at top of the stack, and convert it to leftist heap if the properties of leftist heap are violated.(swap left & right child)
6. Recursively go on popping the elements from the stack and making them the right child of new stack top.

Example:

Consider two leftist heaps given below:

Merge them into a single leftist heap

The subtree at node 7 violates the property of leftist heap so we swap it with the left child and retain the property of leftist heap.

Convert to leftist heap. Repeat the process

The worst case time complexity of this algorithm is O(log n) in the worst case, where n is the number of nodes in the leftist heap.

Another example of merging two leftist heap:

To insert a node

To insert a node into a leftist heap, merge the leftist heap with the node After deleting root X from a leftist heap, merge its left and right sub heaps In summary, there is only one operation, a merge

Height of a leftist heap ≈ O(log n) Maximum number of values stored in Stack ≈ 2 ∗ O(log n) ≈ O(log n) Total cost of merge ≈ O(log n)

Comparison of Search Trees

The comparison of search trees is performed based on the **Time complexity** of search, insertion and deletion operations in search trees. The following table provides the Time complexities of search trees. These Time complexities are defined for **'n'** number of elements.

Search Tree	Average Case			Worst Case		
	Insert	Delete	Search	Insert	Delete	Search
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)
AVL Tree	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
B - Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)

Red - Black Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Splay Tree	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$

Skew Heap

A **skew heap** (or self – adjusting heap) is a heap data structure implemented as a **binary tree**. Skew heaps are advantageous because of their ability to **merge more quickly** than binary heaps.

In contrast with binary heaps, there are no structural constraints, so there is no guarantee that the height of the tree is logarithmic.

Only two conditions must be satisfied :

1. The general heap order must be there (root is minimum and same is recursively true for subtrees), but balanced property (all levels must be full except the last) is not required.
2. Main operation in Skew Heaps is Merge. We can implement other operations like insert, extractMin(), etc using Merge only.

Ø Skew Simplifies leftist heap by not maintaining null path lengths
Ø Skew Heaps are a self-adjusting form of Leftist Heap by unconditionally swapping all nodes in the right merge path Skew Heaps maintain a short right path from the root.
Ø Swapping children at every step.
Ø A Skew (min)Heap is a binary tree that satisfies the following conditions.
If X is a node and L and R are its left and right children, then:

· 1 X.value ≤ L.value

· 2 X.value ≤ R.value

Ø A Skew (max)Heap is a binary tree that satisfies the following conditions.

If X is a node and L and R are its left and right children, then:

- 1 X.value ≥ L.value
- 2 X.value ≥ R.value

Ø Skews are binarys trees with heap property

Ø skew merging avoids right-heavy trees, gives O(log n) amortised complexity

Ø other operations are based on merge

Example :

1. Consider the skew heap 1 to be

2. The second heap to be considered

3. And we obtain the final merged tree as

Recursive Merge Process :

merge(h1, h2)

1. Compare roots of two heaps
2. Recursively merge the heap that has the larger root with the right subtree of the other heap.
3. Make the resulting merge the right subtree of the heap that has smaller root.
4. Swap the children of the new heap

Alternatively, there is a **non-recursive approach** which tends to be a little clearer, but does require some sorting at the outset.

- Split each heap into subtrees by cutting every rightmost path. (From the root node, sever the right node and make the right child its own subtree.) This will result in a set of trees in which the root either only has a left child or no children at all.
- Sort the subtrees in ascending order based on the value of the root node of each subtree.
- While there are still multiple subtrees, iteratively recombine the last two (from right to left).
- If the root of the second-to-last subtree has a left child, swap it to be the right child.
- Link the root of the last subtree as the left child of the second-to-last subtree.

Adding Values

Adding a value to a skew heap is like merging a tree with one node together with the original tree. The root node is linked as the left child of the new value, which itself becomes the new root.

Removing Values

Removing the first value in a heap can be accomplished by removing the root and (a) merging the child subtrees if the root has two children, (b) replacing the root with the non-nil child, or (c) ending with an empty heap.

Binomial Heap

As we have already discussed about the heap data structure which is of two types, i.e., ***min heap and max heap.*** A binomial heap can be defined as the collection of binomial tree that satisfies the heap properties, i.e., min heap. The min heap is a heap in which each node has a value lesser than the value of its child nodes.

To understand the binomial heap, we first understand about the binomial tree.

What is Binomial tree?

A Binomial tree is a tree in which B_k is an ordered tree defined recursively, where k is defined as the order of the binomial tree.

- If the binomial tree is represented as **B_0** then the tree consists of a single node.
- In general terms, **B_k** consists of two binomial trees, i.e., B_{k-1} and B_{k-1} are linked together in which one tree becomes the left subtree of another binomial tree. It can be represented as:

Let's understand through examples.

If B_0, where k is 0, means that there would exist only one node in the tree shown as below:

If B_1, where k is 1, means k-1 equal to 0. Therefore, there would be two binomial trees of B_0 in which one B_0 becomes the left subtree of another B_0.

If B_2, where k is 2, means k-1 equal to 1. Therefore, there would be two binomial trees of B_1 in which one B_1 becomes the left subtree of another B_1.

If B_3 , where k is 3, means k-1 equal to 2. Therefore, there would be two binomial trees of B_3 in which one B_3 becomes the left subtree of another B_3.]

Properties of Binomial tree

- There are 2^k

  If k=1 then $2^0 = 1$. The number of nodes is 1.

  If k = 2 then $2^1 = 2$. The number of nodes is 2.

- The height of the tree is k.

  If k=0 then the height of the tree would be 0.

  If k=1 then the height of the tree would be 1.

- There are exactly k_{ci} or k/i nodes at depth i = 0, 1..k.

  For example, if k is equal to 4 and at depth i=2, we have to determine the number of nodes.

Binomial Heap

A binomial heap is a collection of binomial trees that satisfies the properties of a min- heap.

The following are the two properties of the binomial heap:

- Each binomial heap obeys the min-heap properties.
- For any non-negative integer k, there should be atleast one binomial tree in a heap where root has degree k.

Let's understand the above two properties through an example.

The above figure has three binomial trees, i.e., B_0, B_2, B_3. The above all three binomial trees satisfy the min heap's property as all the nodes have a smaller value than the child nodes.

The above figure also satisfies the second property of the binomial heap. For example, if we consider the value of k as 3, then we can observe in the above figure that the binomial tree of degree 3 exists in the heap.

Representation of Binomial heap node

The above figure shows the representation of the binomial heap node in the memory. The first block in the memory contains the pointer that stores the address of the parent of the node.

The second block stores the key value of the node.

The third block contains the degree of the node.

The fourth block is divided into two parts, i.e., left child and sibling. The left child contains the address of the left child of a node, whereas the sibling contains the address of the sibling.

Important point

- If we want to create the binomial heap of 'n' nodes, that can be simply defined by the binary number of 'n'. For example: if we want to create the binomial heap of 13 nodes; the binary form of 13 is 1101, so if we start the numbering from the rightmost digit, then we can observe that 1 is available at the 0, 2, and 3 positions; therefore, the binomial heap with 13 nodes will have B_0, B_2, and B_3 binomial trees.

Let's consider another example.

If we want to create the binomial heap of 9 nodes. The binary form of 9 is 1001. As we can observe in the binary number that 1 digit is available at 0 and 3 position; therefore, the binomial heap will contain the binomial trees of 0 and 3 degree.

Operations on Binomial Heap

- **Creating a new binomial heap:** When we create a new binomial heap then it simply takes O(1) time because creating a heap will create the head of the heap in which no elements are attached.
- **Finding the minimum key:** As we know that binomial heap is a collection of binomial trees and each binomial tree satisfies the property of min heap means root node contains the minimum value. Therefore, we need to compare only root node of all the binomial trees to find the minimum key. The time complexity for finding a minimum key is O(logn).
- **Union of two binomial heap:** If we want to combine two binomial heaps, then we can simply find the union of two binomial heaps. The time complexity for finding a union is O(logn).
- **Inserting a node:** The time complexity for inserting a node is **O(logn)**.
- **Extracting minimum key:** The time complexity for removing a minimum key is O(logn).
- **Decreasing a key:** When the key value of any node is changed, then the binomial tree does not satisfy the min-heap. We need to perform some rearrangements in order to satisfy the min-heap property. The time complexity would be **O(logn)**.
- **Deleting a node:** The time complexity for deleting a node is O(logn).

Union of two Binomial Heap

To perform the union to two binomial heap, we will use the following cases:

Case 1: If degree[x] is not equal to degree[next x] then move pointer ahead.

Case 2: if degree[x] = degree[next x] = degree[sibling(next x)] then

Move pointer ahead.

Case 3: If degree[x] = degree[next x] but not equal to degree[sibling[next x]]

and key[x] < key[next x] then remove [next x] from root and attached to x.

Case 4: If degree[x] = degree[next x] but not equal to degree[sibling[next x]]

and key[x] > key[next x] then remove x from root and attached to [next x].

Let's understand the union of two binomial heaps through an example.

As we can observe in the above figure that there are two binomial heaps. First, we combine these two binomial heaps. To combine these two binomial heaps, we need to arrange them in the increasing order of binomial trees shown as below:

- Initially, x points to the B_0 having value 12, and next[x] points to B_0 having value 18. The B_1 is the sibling of B_0 having node value 18. Therefore, the sibling B_1 is represented as sibling[next x]. Now, we will apply case 1. Case 1 says that '**if degree[x] is not equal to degree[next x] then move pointer ahead**' but in the above example, the degree of x is the same as the degree of next x, so this case is not valid.

Now we will apply case 2. The case 2 says that '**if degree[x] = degree[next x] = degree[sibling(next x)] then Move pointer ahead**'. In the above example, the degree of x is same as the degree of the next x but not equal to the degree of a sibling of next x; therefore, this case is not valid.

Now we will apply case 3. The case 3 says that '**If degree[x] = degree[next x] but not equal to degree[sibling[next x] and key[x] < key[next x] then remove [next x] from root and attached to x'**. In the above example, the degree of x is equal to the degree of next x but equal to the degree of a sibling of next x, and the key value of x, i.e., 12, is less than the value of next x, i.e., 18; therefore, this case is valid. So, we have to remove the next x, i.e., 18, from the root and attach it to the x, i.e., 12, shown as below:

As we can observe in the above binomial heap that node 18 is attached to the node 12.

- Now we will reapply the cases in the above binomial heap. First, we will apply case 1. Since x is pointing to node 12 and next x is pointing to node 7, the degree of x is equal to the degree of next x; therefore, case 1 is not valid.

Now we will apply case 2. Since the degree of x is equal to the degree of next x and also equal to the degree of sibling of next x; therefore, this case is valid. We will move the pointer ahead shown as below:

As we can observe in the above figure that 'x' points to the binomial tree having root node 7, next(x) points to the binomial tree having root node 3 while prev(x) points to the binomial tree having root node 12. The sibling(next(x)) points to the binomial tree having root node 15.

- Now we will apply case 3 in the above tree. Since the degree of x is equal to the degree of next x, i.e., 1 but not equal to the degree of a sibling of next x, i.e., 2. Either case 3 or case 4 is valid based on the second condition. The key value of x, i.e., 7 is greater than the key value of next(x), i.e., 3; therefore, we can say that case 4 is valid. Here, we need to remove the x and attach it to the next(x) shown as below:

As we can observe in the above figure that x becomes the left child of next(x). The pointer also gets updated. Now, x will point to the binomial tree having root node 3, and degree is also changed to 2. The next(x) points to the binomial tree having root node as 15, and the sibling(next(x)) will point to the binomial tree having root node as 6.

- In the above tree, the degree of x, i.e., B_2, is the same as the degree of next(x), i.e., B_2, but not equal to the degree of sibling(next(x)), i.e., B_4. Therefore, either case 3 or case 4 is valid based on the second condition. Since the key value of x is less than the value of next(x) so we need to remove next(x) and attach it to the x shown as below:

As we can observe in the above figure that next(x) becomes the left child of x, and the degree of x also gets changed to B_3. The pointer next(x) also gets updated, and it now points to the binomial tree having root node 6. The degree of x is 3, and the degree of next(x) is 4. Since the degrees of x and next(x) are not equal, so case 1 is valid. We will move the pointer ahead, and now x points to node 6.

The B_4 is the last binomial tree in a heap, so it leads to the termination of the loop. The above tree is the final tree after the union of two binomial heaps.

Inserting a new node in a Binomial heap

Now we will see how to insert a new node in a heap. Consider the below heap, and we want to insert a node 15 in a heap.

-

In the above heap, there are three binomial trees of degree B_0, B_1, and B_2, where B_0 is attached to the head of the heap.

Let's assume that node 15 is attached to the head of the heap shown as below:

First, we will combine these two heaps. As the degree of node 12 and node 15 is B_0 so node 15 is attached to the node 12 shown in the above figure.

Now we assign 'x' to the B_0 having key value 12, next(x) to the B_0 having key value 15, and sibling(next(x)) to the B_1 having key value 7.

Since the degree of x is equal to the degree of next(x) but not equal to the degree of sibling of next(x) so either case 3 or case 4 will be applied to the heap. The key value of x is greater than the key value of next(x); therefore, next(x) would be removed and attached it to the x.

Now, we will reapply the cases in the above heap. Now, x points to the node 12 having degree B_1, next(x) points to the node 7 having degree B_1 and sibling(next(x)) points to the node 15 having degree B_2. Since the degree of x is same as the degree of next(x) but not equal to the degree of sibling of next(x), so either case 3 or case 4 is applied. The key value of x is greater than the key value of next(x); therefore, the x is removed and attached to the x shown as below:

As we can observe in the above figure that when 'x' is attached to the next(x) then the degree of node 7 is changed to B_2. The pointers are also get updated. Now 'x' points to the node 7 and next(x) points to the node 15. Now we will reapply the cases in the above heap. Since the degree of x is equal to the degree of next(x) and the key value of x is less than the key value of next(x); therefore, next(x) would be removed from the heap and attached it to the x shown as below:

As we can observe in the above heap that the degree of x gets changed to 3, and this is the final binomial heap after inserting the node 15.

Extracting a minimum key

Here extracting a minimum key means that we need to remove the element which has minimum key value. We already know that in min heap, the root element contains the minimum key value. Therefore, we have to compare the key value of root node of all the binomial trees.

Consider the binomial heap which is given below:

In the above heap, the key values of root node of all the binomial trees are 12, 7 and 15. The key value 7 is the minimum value so we will remove the node 7 from the tree shown as below:

As we can observe in the above binomial heap that the degree of node 12 is B_0, degree of node 25 is B_0, and degree of node 15 is B_2. The pointer x points to the node 12, next(x) points to the node 25, and sibling(next(x)) points to the node 15. Since the degree of x is equal to the degree of next(x) but not equal to the degree of sibling(next(x)); therefore, either case 3 or case 4 will be applied to the above heap. The key value of x is less than the key value of next(x), so node 25 will be removed and attached to the node 12 shown as below:

The degree of node 12 is also changed to 1.

Decreasing a key

Now we will see how to decrease a key with the help of an example. Consider the below heap, and we will decrease the key 45 by 7:

After decreasing the key by 7, the heap would look like:

Since the above heap does not satisfy the min-heap property, element 7 will be compared with an element 30; since the element 7 is less than the element 30 so 7 will be swapped with 30 element shown as below:

Now we will again compare element 7 with its root element, i.e., 8. Since element 7 is less than element 8 so element 7 will be swapped with an element 8 shown as below:

Now, the above heap satisfies the property of the min-heap.

Deleting a node

Now we will see how to delete a node with the help of an example. Consider the below heap, and we want to delete node 41:

First, we replace node 41 with the smallest value, and the smallest value is -infinity, shown as below:

Now we will swap the -infinity with the root node of the tree shown as below:

The next step is to extract the minimum key. Since the minimum key in a heap is -infinity so we will extract this key, and the heap would be:

Fibonacci Heaps

Like Binomial heaps, Fibonacci heaps are collection of trees. They are loosely based on binomial heaps. Unlike trees with in binomial heaps are ordered trees within Fibonacci heaps are rooted but unordered.

Each node x in Fibonacci heaps contains a pointer p[x] to its parent, and a pointer child[x] to any one of its children. The children of x are linked together in a circular doubly linked list known as child list of x. Each child y in a child list has pointers left[y] and right[y] to point left and right siblings of y respectively. If node y is only child then left[y] = right[y] = y. The order in which sibling appears in a child list is arbitrary.

Example of Fibonacci Heap

This Fibonacci Heap H consists of five Fibonacci Heaps and 16 nodes. The line with arrow head indicates the root list. Minimum node in the list is denoted by min[H] which is holding 4.

The asymptotically fast algorithms for problems such as computing minimum spanning trees, finding single source of shortest paths etc. makes essential use of Fibonacci heaps.

In terms of Time Complexity, Fibonacci Heap beats both Binary and Binomial Heaps.

Below are amortized time complexities of **Fibonacci Heap**.

1) Find Min: **Θ(1)** [Same as both Binary and Binomial]
2) Delete Min: **O(Log n)** [Θ(Log n) in both Binary and Binomial]
3) Insert: **Θ(1)** [Θ(Log n) in Binary and Θ(1) in Binomial]
4) Decrease-Key: **Θ(1)** [Θ(Log n) in both Binary and Binomial]
5) Merge: **Θ(1)** [Θ(m Log n) or Θ(m+n) in Binary and Θ(Log n) in Binomial]

Like Binomial Heap, Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be Binomial Tree).
Below is an example Fibonacci Heap taken from here.
Fibonacci Heap maintains a pointer to minimum value (which is root of a tree). All tree roots are connected using circular doubly linked list, so all of them can be accessed using single 'min' pointer.

The main idea is to execute operations in "lazy" way. For example merge operation simply links two heaps, insert operation simply adds a new tree with single node. The operation extract minimum is the most complicated operation. It does delayed work of consolidating trees. This makes delete also complicated as delete first decreases key to minus infinite, then calls extract minimum.

Below are some interesting facts about Fibonacci Heap

1. The reduced time complexity of Decrease-Key has importance in Dijkstra and Prim algorithms. With Binary Heap, time complexity of these algorithms is O(VLogV + ELogV). If Fibonacci Heap is used, then time complexity is improved to O(VLogV + E)

2. Although Fibonacci Heap looks promising time complexity wise, it has been found slow in practice as hidden constants are high
3. Fibonacci heap are mainly called so because Fibonacci numbers are used in the running time analysis. Also, every node in Fibonacci Heap has degree at most O(log n) and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the kth Fibonacci number.

Fibonacci Heap – Insertion and Union

Fibonacci Heap is a collection of trees with min-heap or max-heap property. In Fibonacci Heap, trees can have any shape even all trees can be single nodes (This is unlike Binomial Heap where every tree has to be Binomial Tree).

In this article, we will discuss Insertion and Union operation on Fibonacci Heap.

Insertion: To insert a node in a Fibonacci heap H, the following algorithm is followed:

1. *Create a new node ‘x’.*
2. *Check whether heap H is empty or not.*
3. *If H is empty then:*
   - *Make x as the only node in the root list.*
   - *Set H(min) pointer to x.*
4. *Else:*
   - *Insert x into root list and update H(min).*

Union: Union of two Fibonacci heaps H1 and H2 can be accomplished as follows:

1. *Join root lists of Fibonacci heaps H1 and H2 and make a single Fibonacci heap H.*
2. *If H1(min) < H2(min) then:*
   - *H(min) = H1(min).*
3. *Else:*
   - *H(min) = H2(min).*