DATA STRUCTURE & ALGORITHM

A BRIEF NOTE BY: UZAIR SALMAN

MSC-IT TERM-III
Superior Group of colleges Jauharabad

Table of Contents

1. INTRODUCTION TO DATA STRUCTURES

Whenever we want to work with large amount of data, then organizing that data is very important. If that data is not organized effectively, it is very difficult to perform any task on that data. If it is organized effectively then any operation can be performed easily on that data.

A data structure can be defined as follows.

"Data structure is a method of organizing large amount of data more efficiently so that any operation on that data becomes easy"

- ☀ Every data structure is used to organize the large amount of data
- ☀ Every data structure follows a particular principle
- ☀ The operations in a data structure should not violate the basic principle of that data structure.

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

a. Linear Data Structures
b. Non - Linear Data Structures

1.1 Linear Data Structures

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

Example

a. Arrays
b. List (Linked List)
c. Stack
d. Queue

1.2 Non - Linear Data Structures

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

Example

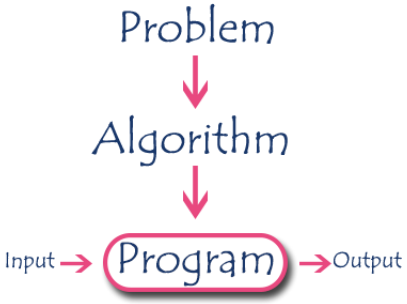
a. Tree
b. Graph
c. Dictionaries
d. Heaps
e. Tries

2. INTRODUCTION TO ALGORITHM

2.1 What is an algorithm?

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

"An algorithm is a sequence of simple instructions used for solving a problem, which can be implemented (as a program) on a computer"



Algorithms are used to convert our problem solution into step by step statements. These statements can be converted into computer programming instructions which form a program. This program is executed by computer to produce solution.

2.1.1 Algorithm Specifications

Every algorithm must satisfy the following specifications...

Input - Every algorithm must take zero or more number of input values from external.
Output - Every algorithm must produce an output as result.
Definiteness - Every statement/instruction in an algorithm must be clear and simple.
Finiteness - For all different cases, the algorithm must produce result within a finite number of steps.
Effectiveness - Every instruction must be basic enough to be carried out and it also must be feasible.

Example of Algorithm

Let us consider the following problem for finding the largest value in a given list of values.

Problem Statement: Find the largest number in the given list of numbers?
Input: A list of positive integer numbers. (List must contain at least one number).
Output: The largest number in the given list of positive integer numbers.
Consider the given list of numbers as 'L' (input), and the largest number as 'max' (Output).

Algorithm

```
Step 1: Define a variable 'max' and initialize with '0'.
Step 2: Compare first number (say 'x') in the list 'L' with 'max', if 'x' is larger than
        'max', set 'max' to 'x'.
Step 3: Repeat step 2 for all numbers in the list 'L'.
Step 4: Display the value of 'max' as a result.
```

Code in Java Programming

```
intfindMax(L)
{
int max = 0,i;
for(i=0; i<listSize; i++)
   {
      if(L[i] > max)
         max = L[i];
   }
   return max;
}
```

2.2 Recursive Algorithm

In computer science, all algorithms are implemented with programming language functions. We can view a function as something that is invoked (called) by another function. It executes its code and then returns control to the calling function. Here, a function can call themselves (by itself) or it may call another function which again call same function inside it.

“The function which calls by itself is called as Direct Recursive function (or Recursive function)”

“The function which calls a function and that function calls it’s called function is called Indirect Recursive function (or Recursive function)”

3. PERFORMANCE ANALYSIS

3.1 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.

"Performance of an algorithm is a process of making evaluative judgment about algorithms."
"Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task."

We compare all algorithms with each other which are solving same problem, to select best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, 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 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 required by that algorithm and time required by that algorithm."

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

1. Space required completing 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)**

3.2 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.

- Memory required to store program instructions
- Memory required to store constant values
- Memory required to store variable values

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...

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

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...

I. 2 bytes to store Integer value,
II. 4 bytes to store Floating Point value,
III. 1 byte to store Character value,
IV. 6 (OR) 8 bytes to store double value

Example:

Consider the following piece of code...

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

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

That means, totally it requires 4 bytes of memory to complete its execution. 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**"

Example:

Consider the following piece of code...

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

In 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 amount of memory depends on the input value of 'n'. This 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"**

3.3 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.

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

I. Whether it is running on Single processor machine or Multi processor machine.
II. Whether it is a 32 bit machine or 64 bit machine
III. Read and Write speed of the machine.
IV. The time it takes to perform Arithmetic operations, logical operations, return value and assignment operations etc.,
V. 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.

Consider the following piece of code...

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

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

	Cost (Time require for line (Units))	Repeatation (No. of Times Executed)	Total (Total Time required in worst case)
int sumOfList(int A[ ], int n)			
{			
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.
Repetition is the amount of computer time required by each operation for all its repetitions.
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.

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

3.4 Big O notation

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

3.4.1 O(1)

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
bool IsFirstElementNull(IList<string> elements)
{
 return elements[0] == null;
}
```

3.4.2 O(N)

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. The example below also demonstrates how Big O favours the worst-case performance scenario; a matching string could be found during any iteration of the for loop and the function would return early, but Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

```
bool ContainsValue(IList<string> elements, string value)
{
foreach (var element in elements)
 {
 if (element == value) return true;
 }
 return false;
}
```

3.4.3 O(N2)

O(N2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N3), O(N4) etc.

```
bool ContainsDuplicates(IList<string> elements)
{
 for (var outer = 0; outer <elements.Count; outer++)
 {
 for (var inner = 0; inner <elements.Count; inner++)
 {
 // Don't compare with self
 if (outer == inner) continue;
 if (elements[outer] == elements[inner]) return true;
 }
 }
 return false;
}
```

3.4.4 O(2N)

O(2N) denotes an algorithm whose growth doubles with each addition to the input data set. The growth curve of an O(2N) function is exponential - starting off very shallow, then rising meteorically. An example of an O(2N) function is the recursive calculation of Fibonacci numbers:

```
int Fibonacci(int number)
{
 if (number <= 1) return number;
 return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

3.4.5 Logarithms

To understand logarithm we discuss a common example: Binary search is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it. Likewise, if the target value is lower than the value of the probe element it will perform the operation against the lower half. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set.
This type of algorithm is described as O(log N). The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase e.g. an input data set containing 10 items takes one second to complete, a data set containing 100 items takes two seconds, and a data set containing 1000 items will take three seconds. Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved and therefore on a par with an input data set half the size. This makes algorithms like binary search extremely efficient when dealing with large data sets.

4. ARRAYS

Whenever we want to work with large number of data values, we need to use that much number of different variables. As the numbers 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.”

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

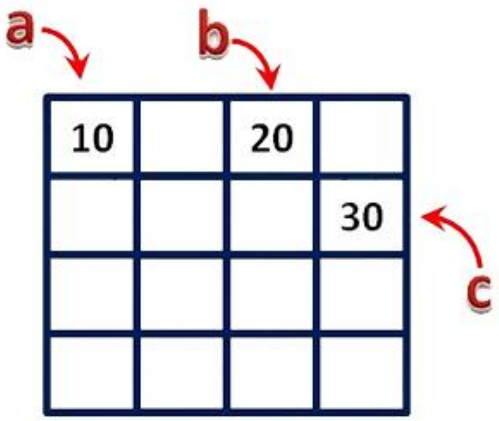
int a, b, c;

In computer memory is organized as shown in figure. Here assume that each box is of 2 bytes of memory.

2 byte for 'a', another 2 bytes for 'b' and 2 more bytes for 'c'.

If we assign following values they will inserted into that memory locations.

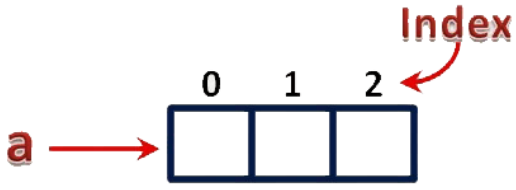
a = 10;
b = 20;
c = 30;

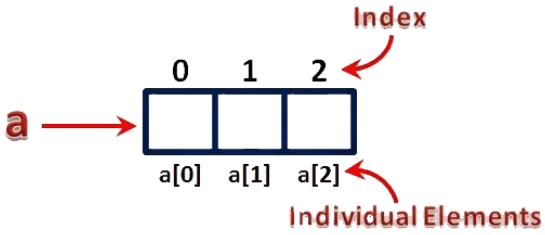


Now consider the following declaration.

int a[3];

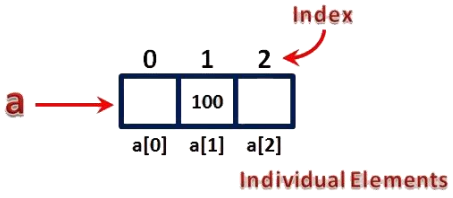
That means all these three memory locations are named as 'a'. But "how can we refer individual elements?" is the big question. Answer for this question is, compiler not only allocates memory, but also assigns a numerical value to each individual element of an array. This numerical value is called as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



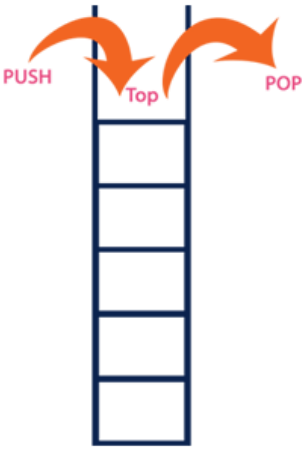


If I want to assign a value to any of these memory locations (array elements), we can assign as follows.

a [1] = 100;



5. STACK ADT



Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at single position which is known as "top". That means, new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on LIFO (Last In First Out) principle.

In a stack, the insertion operation is performed using a function called "push" and deletion operation is performed using a function called "pop".

In the figure, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

A stack data structure can be defined as follows.

“Stack is a linear data structure in which the operations are performed based on LIFO principle.”

"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle"

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. Top is at 50 as shown in the image

5.1 Operations on a Stack

The following operations are performed on the stack...

- Push (To insert an element on to the stack)
- Pop (To delete an element from the stack)
- Display (To display elements of the stack)

Stack data structure can be implementing in two ways. They are as follows...

- Using Array
- Using Linked List

5.2 Stack Using Array

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable 'top'. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

5.2.1 Stack Operations using Array

Before implementing actual operations, first follow the below steps to create an empty stack.

```
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 functions used in stack implementation.
Step 3: Create a one dimensional array with fixed size (int stack[SIZE])
Step 4: Define a integer variable 'top' and initialize with '-1'. (int top = -1)
Step 5: In main method display menu with list of operations and make suitable function
        calls to perform operation selected by the user on the stack.
```

5.2.1.1 push(value)

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

```
Step 1: Check whether stack is FULL. (top == SIZE-1)
Step 2: If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and
        terminate the function.
Step 3: If it is NOT FULL, then increment top value by one (top++) and set stack[top] to
        value (stack[top] = value).
```

5.2.1.2 pop()

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

```
Step 1: Check whether stack is EMPTY. (top == -1)
Step 2: If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and
        terminate the function.
Step 3: If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).
```

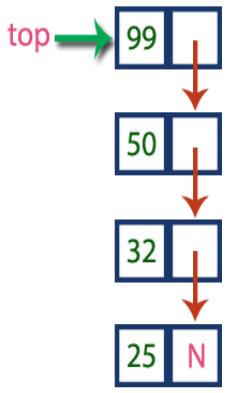
5.2.1.3 display()

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

```
Step 1: Check whether stack is EMPTY. (top == -1)
Step 2: If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.
Step 3: If it is NOT EMPTY, then define a variable 'i' and initialize with top. Display
        stack[i] value and decrement i value by one (i--).
Step 3: Repeat above step until i value becomes '0'.
```

5.3 Stack using Linked List

The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means stack implemented using linked list works for variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.



In linked list implementation of a stack, every new element is inserted as 'top' element. That means every newly inserted element is pointed by 'top'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by 'top' by moving 'top' to its next node in the list. The next field of the first element must be always NULL.

5.3.1 Operations

To implement stack using 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 a Node pointer 'top' and set it to NULL.
Step 4: Implement the main method by displaying Menu with list of operations and make
        suitable function calls in the main method.
```

5.3.1.1 push(value)

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

```
Step 1: Create a newNode with given value.
Step 2: Check whether stack is Empty (top == NULL)
Step 3: If it is Empty, then set newNode → next = NULL.
Step 4: If it is Not Empty, then set newNode → next = top.
Step 5: Finally, set top = newNode.
```

5.3.1.2 pop()

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

```
Step 1: Check whether stack is Empty (top == NULL).
Step 2: If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!" and
        terminate the function
Step 3: If it is Not Empty, then define a Node pointer 'temp' and set it to 'top'.
Step 4: Then set 'top = top → next'.
Step 5: Finally, delete 'temp' (free(temp)).
```

5.3.1.3 display()

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

```
Step 1: Check whether stack is Empty (top == NULL).
Step 2: If it is Empty, then display 'Stack is Empty!!!' and terminate the function.
Step 3: If it is Not Empty, then define a Node pointer 'temp' and initialize with top.
Step 4: Display 'temp → data --->' and move it to the next node. Repeat the same until temp
        reaches to the first node in the stack (temp → next != NULL).
Step 5: Finally! Display 'temp → data ---> NULL'.
```

6. RECURSION

Recursion is a programming technique by which the function calls to itself. In other words, to determine the solution of any problem recursion depends on solutions to smaller instances of the same problem. For example

```
Function recursive(paramenters)
{
        recursive(paramenter);
}
```

The function calls to itself. But is it sufficient for any function to be recursive. The above function will make an infinite call to itself. Hence for any function to be recursive it should follow following rules:

- It should have base case. Base case will determine when to terminate the function.
- After every call the function tends closer to the base case.

Fibonacci series Using Recursion

```
private static int fibonacci(int number) {
          if (number == ZERO) {
              return ZERO;
          } else if (number == ONE) {
              return ONE;
          } else {
              return fibonacci(number - ONE)
                       + fibonacci(number - TWO);
          }
```

7. EXPRESSIONS

in any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression. An expression can be defined as follows.

"An expression is a collection of operators and operands that represents a specific value."

Operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

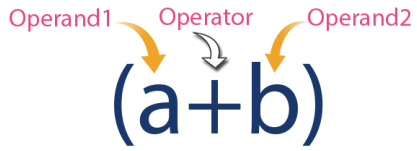
7.1 Expression Types

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

- Infix Expression
- Postfix Expression
- Prefix Expression

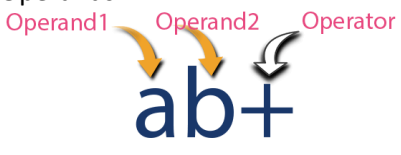
7.1.1 Infix Expression

In infix expression, operator is used in between operands.



7.1.2 Postfix Expression

In postfix expression, operator is used after operands. We can say that "Operator follows the Operands".



7.1.3 Prefix Expression

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".

7.2 Expression Conversion

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

1. Find all the operators in the given Infix Expression.
2. Find the order of operators evaluated according to their Operator precedence.
3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Example
Consider the following Infix Expression to be converted into Postfix Expression...

D = A + B * C

```
Step 1: The Operators in the given Infix Expression: = , + , *
Step 2: The Order of Operators according to their preference: * , + , =
Step 3: Now, convert the first operator * ----- D = A + B C *
Step 4: Convert the next operator + ----- D = A BC* +
Step 5: Convert the next operator = ----- D ABC*+ =
```

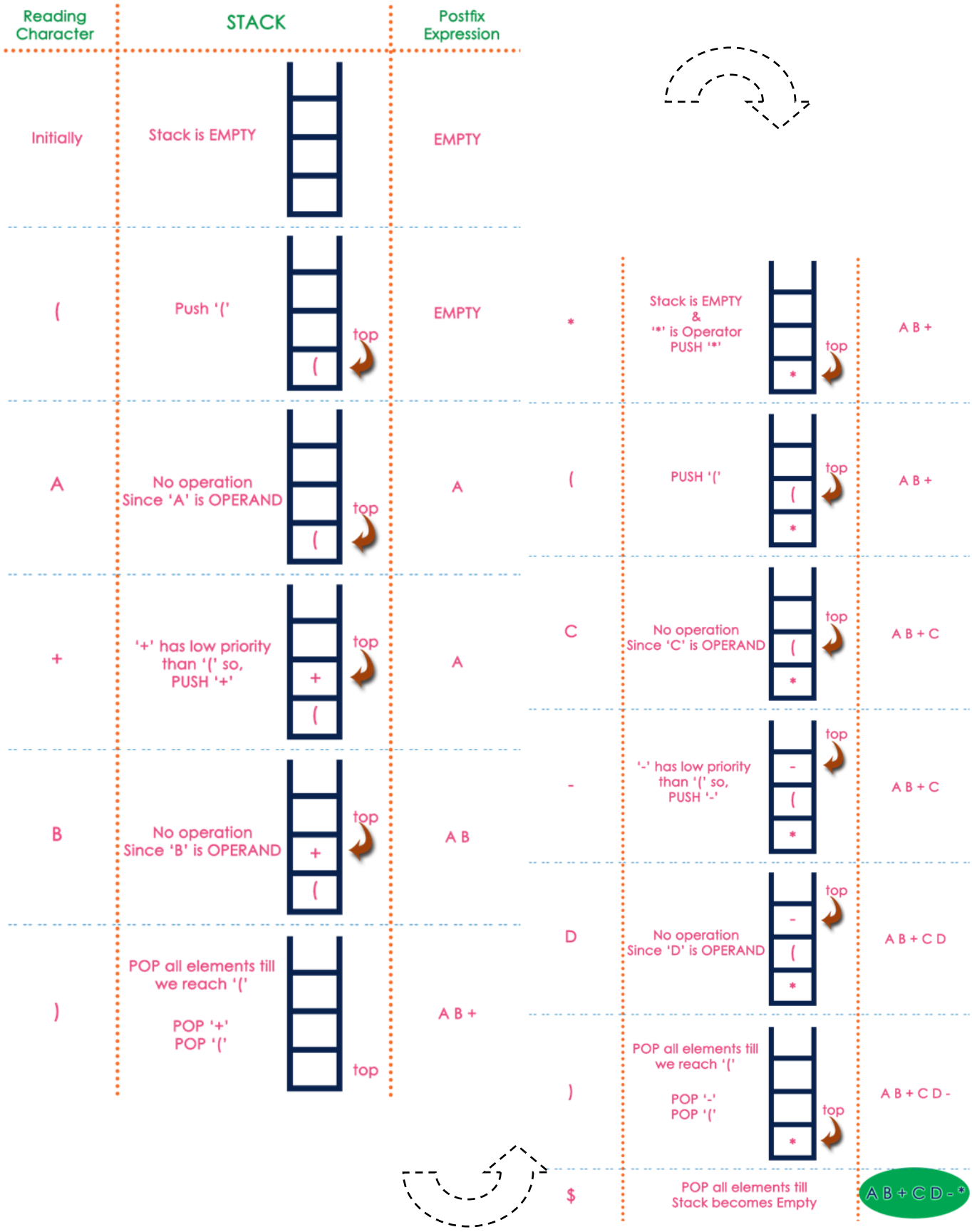
Finally, given Infix Expression is converted into Postfix Expression as follows...

D A B C * + =

7.2.1 Conversion using Stack

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is **operand**, then directly print it to the result (Output).
3. If the reading symbol is **left parenthesis '('**, then Push it on to the Stack.
4. If the reading symbol is **right parenthesis ')'**, then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
5. If the reading symbol is **operator (+ , - , * , / etc.,),** then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.


Reading Character
STACK
Postfix Expression
Initially
Stack is EMPTY
EMPTY
(
Push '('
top
EMPTY
A
No operation Since 'A' is OPERAND
top
A
+
'+' has low priority than '(' so, PUSH '+'
top
A
B
No operation Since 'B' is OPERAND
top
A B
)
POP all elements till we reach '('
POP '+'
POP '('
top
A B +
*
Stack is EMPTY & '*' is Operator PUSH '*'
top
A B +
(
PUSH '('
top
A B +
C
No operation Since 'C' is OPERAND
top
A B + C
-
'-' has low priority than '(' so, PUSH '-'
top
A B + C
D
No operation Since 'D' is OPERAND
top
A B + C D
)
POP all elements till we reach '('
POP '-'
POP '('
top
A B + C D -
$
POP all elements till Stack becomes Empty
A B + C D - *

7.3 Postfix Expression Evaluation

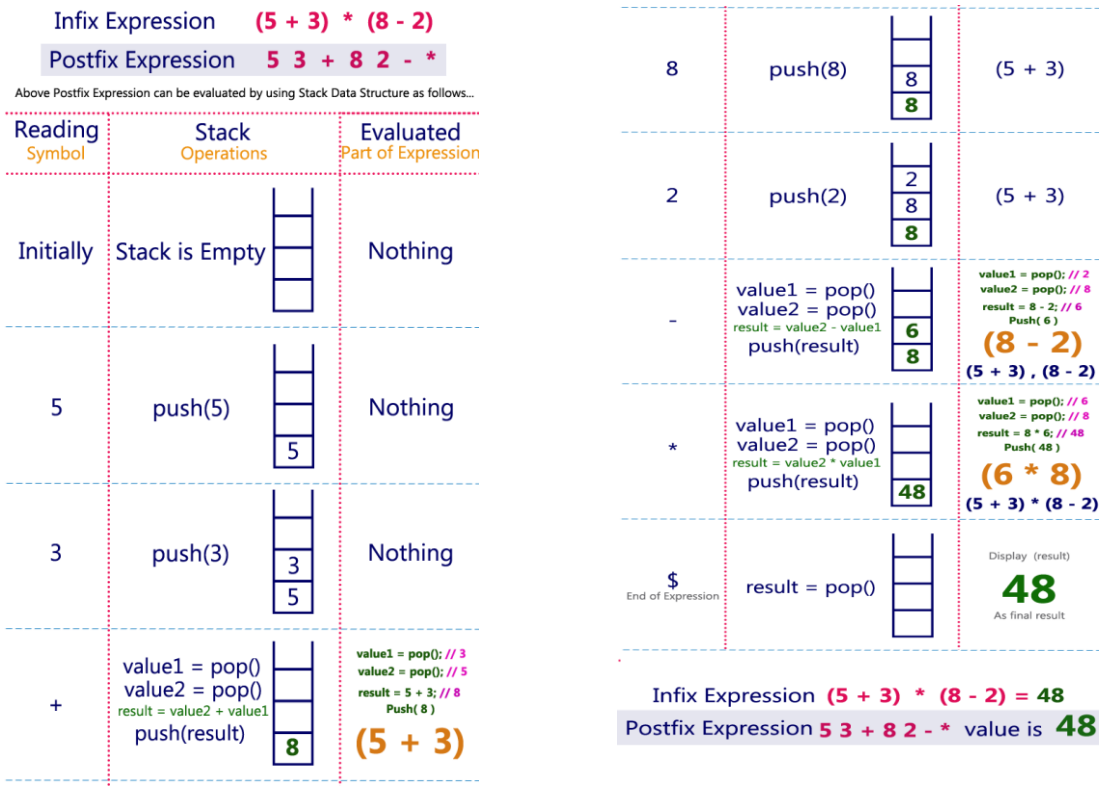
A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure.

7.3.1 Expression Evaluation using Stack

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

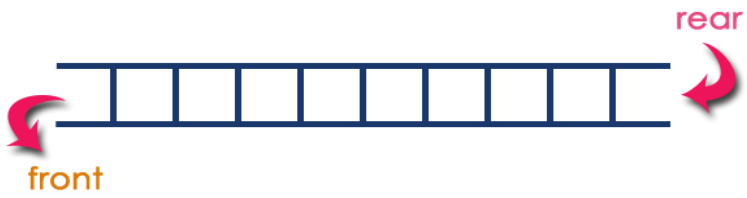
```
Step 1: Read all the symbols one by one from left to right in the given Postfix Expression
Step 2: If the reading symbol is operand, then push it on to the Stack.
Step 3: If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop
        operations and store the two popped oparands in two different variables (operand1
        and operand2). Then perform reading symbol operation using operand1 and operand2
        and push result back on to the Stack.
Step 4: Finally! perform a pop operation and display the popped value as final result.
```



8. QUEUE ADT

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 of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at other 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.



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.”

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

Figure 1: Queue after inserting 25, 30, 51, 60 and 85.

8.1 Operations on a Queue

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

- enQueue(value) - (To insert an element into the queue)
- deQueue() - (To delete an element from the queue)
- display() - (To display the elements of the queue)

Queue Implementation

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

- Using Array
- Using Linked List

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

8.2 Queue Using Array

Data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values. The implementation of queue data structure using array 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 increment 'front' value by one and then display the value at 'front' position as deleted element.

8.2.1 Queue Operations using Array

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 main method by displaying menu of operations list and make suitable
        function calls to perform operation selected by the user on queue.
```

8.2.1.1 enQueue(value)

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 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.
```

8.2.1.2 deQueue()

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 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).
```

8.2.1.3 display()

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

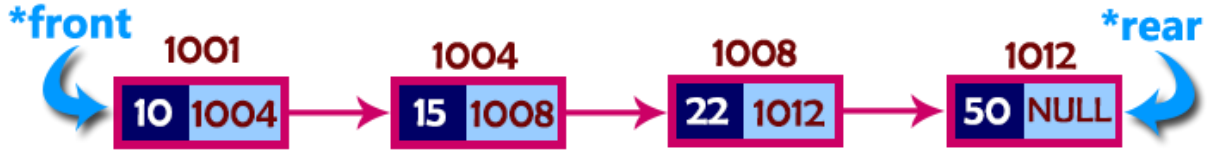
```
Step 1: Check whether queue is EMPTY. (front == rear)
Step 2: If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
Step 3: If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.
Step 4: Display 'queue[i]' value and increment 'i' value by one (i++).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

8.3 Queue using Linked List

The major problem with the queue implemented using array is, It will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using 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 linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list 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'.



8.3.1 Operations

To implement queue using 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.
```

8.3.1.1 enQueue(value)

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.
```

8.3.1.2 deQueue()

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)).
```

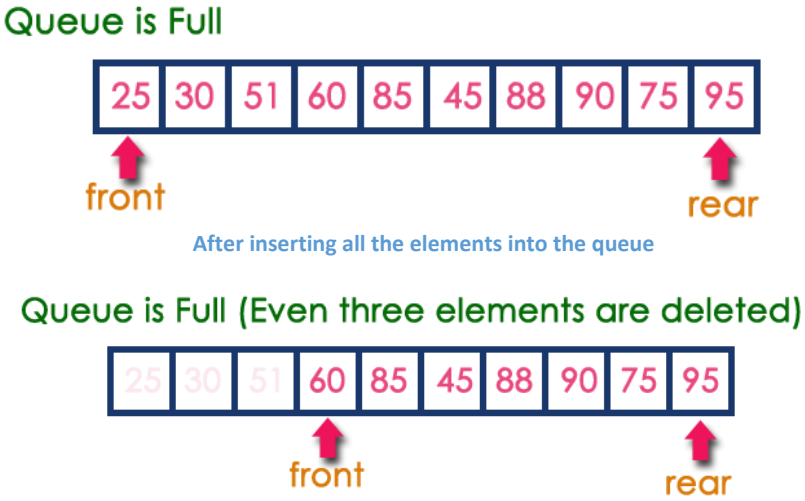
8.3.1.3 display()

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'.
```

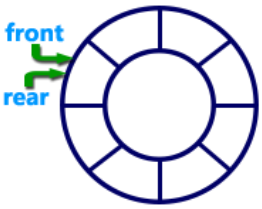
9. CIRCULAR QUEUE

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once if queue becomes full, we cannot insert the new element until queue is rest.



This situation also says that Queue is full and we cannot insert the new element because, 'rear' is still at last position.

What is Circular Queue?



"Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle."

9.1 Implementation of Circular Queue

To implement a circular queue data structure using array, we first perform the following steps before we implement actual operations.

```
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 user defined functions used in circular queue implementation.
Step 3: Create a one dimensional array with above defined SIZE (int cQueue[SIZE])
Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int
        front = -1, rear = -1)
Step 5: Implement main method by displaying menu of operations list and make suitable
        function calls to perform operation selected by the user on circular queue.
```

9.1.1 enQueue(value)

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

```
Step 1: Check whether queue is FULL. ((rear == SIZE-1 && front == 0) || (front == rear+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 check rear == SIZE - 1 &&front != 0 if it is TRUE, then set
        rear = -1.
```

```
Step 4: Increment rear value by one (rear++), set queue[rear] = value and check 'front == -
        1' if it is TRUE, then set front = 0.
```

9.1.2 deQueue()

deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position. The deQueue() function doesn't take any value as parameter. We can use the following steps to delete an element from the circular queue...

```
Step 1: Check whether queue is EMPTY. (front == -1 && rear == -1)
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 display queue[front] as deleted element and increment the
        front value by one (front ++). Then check whether front == SIZE, if it is TRUE,
        then set front = 0. Then check whether both front - 1 and rear are equal (front -1
        == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).
```

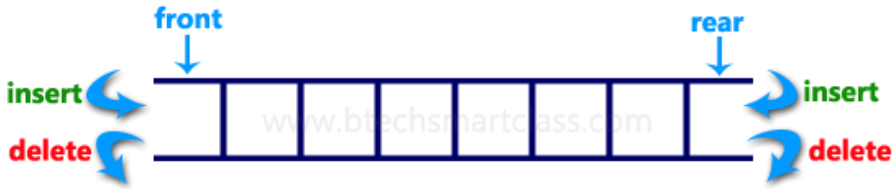
9.1.3 display()

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

```
Step 1: Check whether queue is EMPTY. (front == -1)
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'.
Step 4: Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value and
        increment 'i' value by one (i++). Repeat the same until 'i<= rear' becomes FALSE.
Step 5: If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value
        by one (i++). Repeat the same until'i<= SIZE - 1' becomes FALSE.
Step 6: Set i to 0.
Step 7: Again display 'cQueue[i]' value and increment i value by one (i++). Repeat the same
        until 'i<= rear' becomes FALSE.
```

10. DOUBLE ENDED QUEUE (DEQUEUE)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

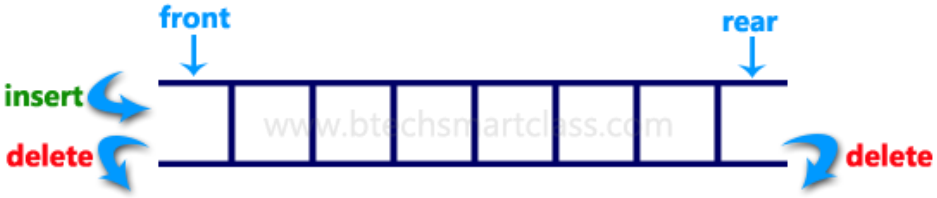


Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

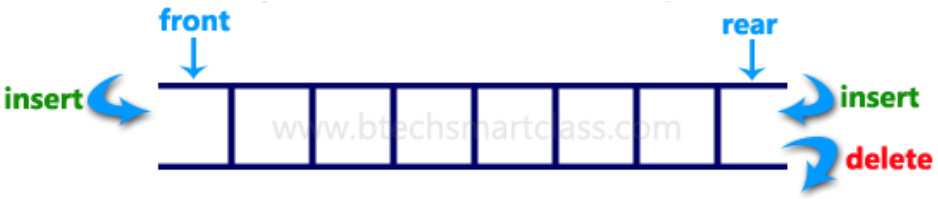
10.1 Input Restricted Dequeue

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



10.2 Output Restricted Double Ended Queue

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



11. LINKED LIST

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

There are three types of Linked list.

I. Single linked list
II. Circular linked list
III. Double linked list

11.1 Linked List

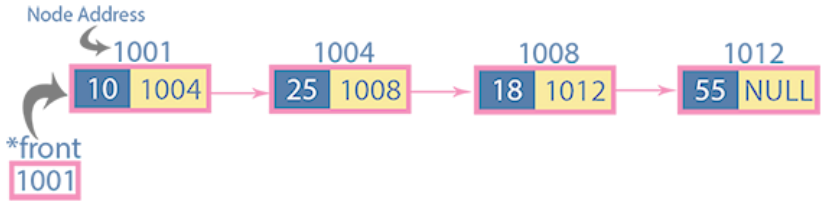
"Single linked list is a sequence of elements in which every element has link to its next element in the sequence."

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

NOTE

☀ In a single linked list, the address of the first node is always stored in a reference node known as "head".

☀ Always next part of the last node must be NULL.



Example of a single linked list.

11.1.1 Operations

In a single 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.
```

11.1.1.1 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**

```
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**

```
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)**

```
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'
```

11.1.1.2 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**

```
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**

```
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**

```
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)).
```

11.1.1.3 Displaying a Single Linked List

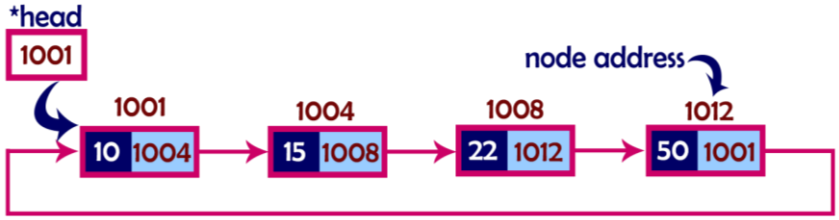
```
Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!!' and terminate the function.
Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
```

```
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).
```

11.2 Circular Linked List

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

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



11.2.1 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.
```

11.2.1.1 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

```
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

```
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)

```
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.
```

11.2.1.2 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

```
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

```
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

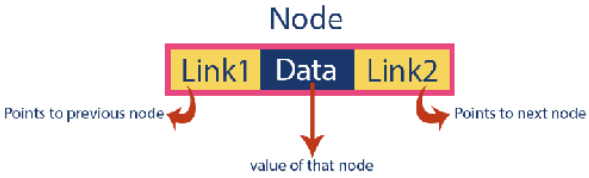
```
Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and
        terminate the function.
Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and
        initialize 'temp1' with head.
Step 4: Keep 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 11: 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

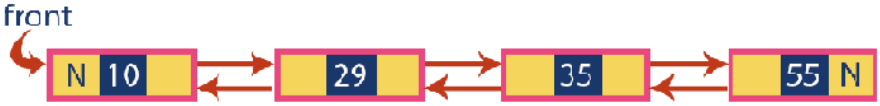
```
Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty, then display 'List is Empty!!!' and terminate the function.
Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4: Keep displaying temp → data with an arrow (--->) until temp reaches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

11.3 Double Linked List

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



NOTE

- 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.

11.3.1 Operations

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

1. Insertion
2. Deletion
3. Display

11.3.1.1 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**

```
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**

```
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)**

```
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 newNode → previous &newNode → next and 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.
```

11.3.1.2 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**

```
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**

```
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**

```
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)).
```

11.3.1.3 Displaying 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).
```

12. Tree

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

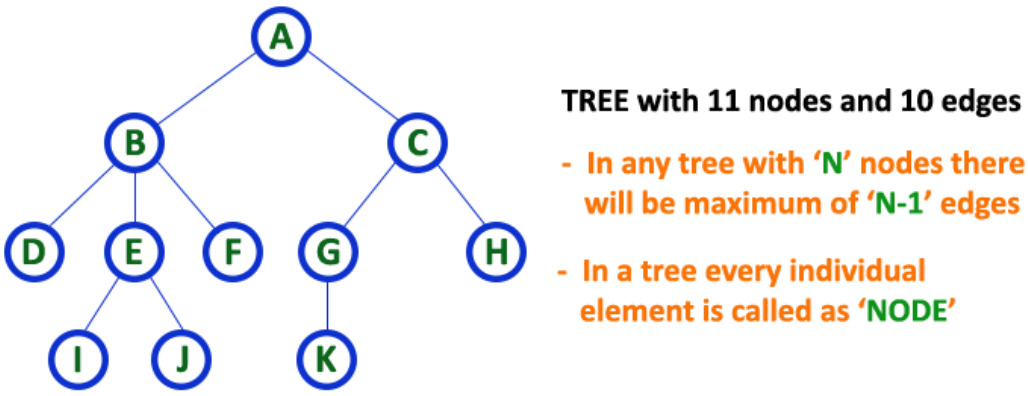
"Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition."

"Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition"

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.

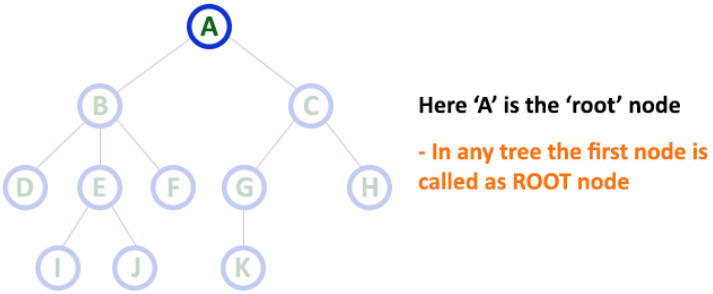
Example:



12.1 Terminology

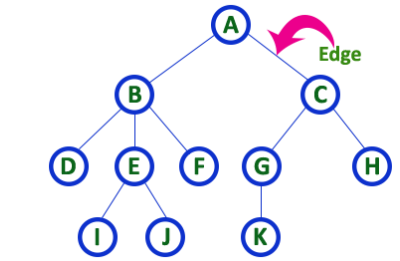
1. Root

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



2. Edge

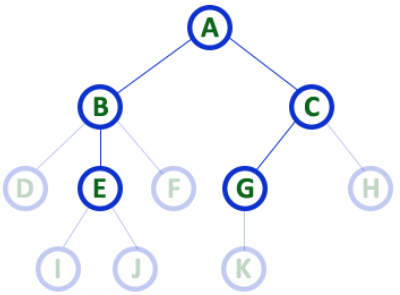
In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

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

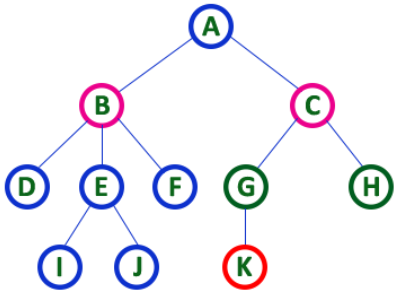


Here A, B, C, E & G are Parent nodes

- 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.

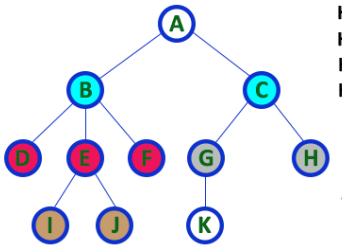


Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

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



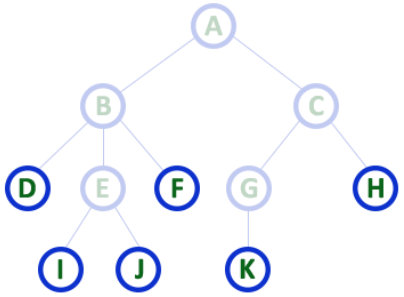
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- 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 as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.



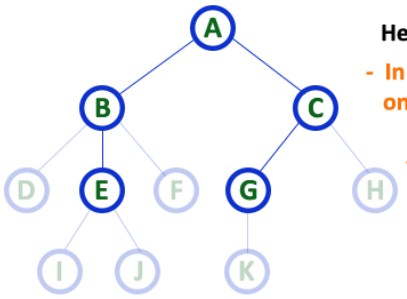
Here D, I, J, F, K & H are Leaf nodes

- 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 atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child.

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

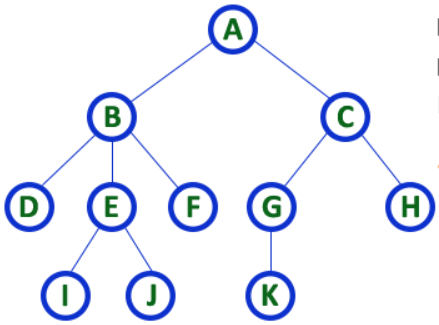


Here A, B, C, E & G are Internal nodes

- 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 as DEGREE of that Node. In simple words, the Degree of a node is 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'

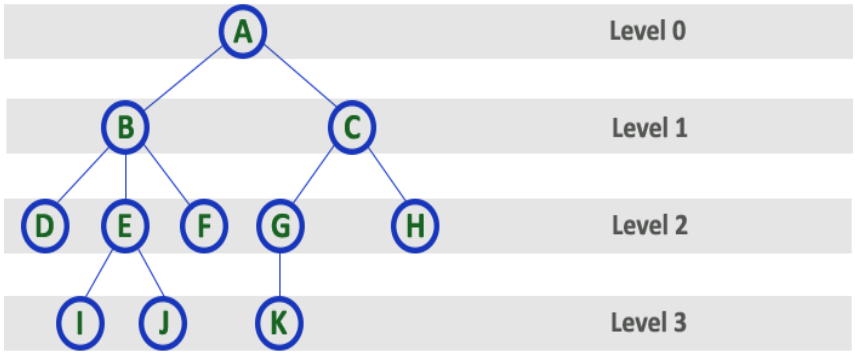


Here Degree of B is 3
Here Degree of A is 2
Here Degree of F is 0

- In any tree, 'Degree' a node is total number of children it has.

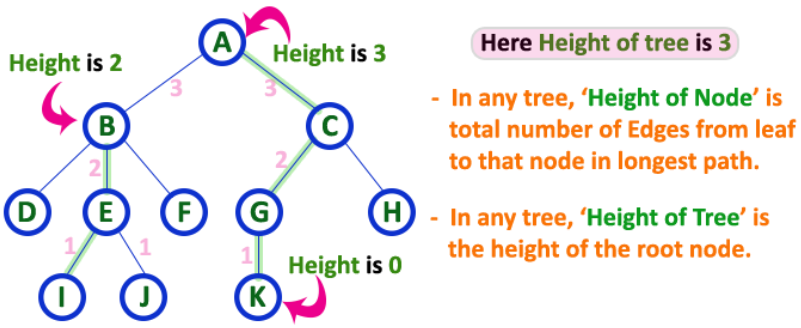
9. Level

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 as a Level and the Level count starts with '0' and incremented by one at each level (Step).



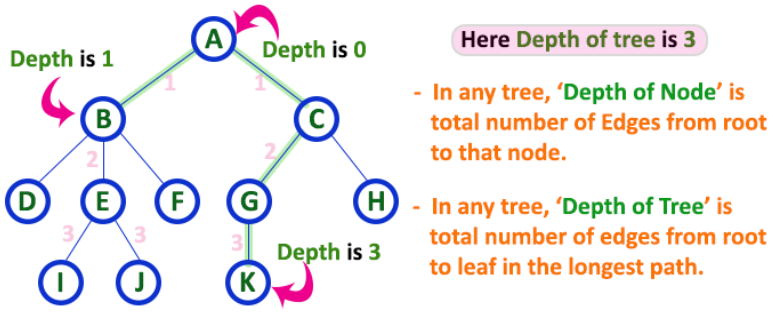
10. Height

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'.



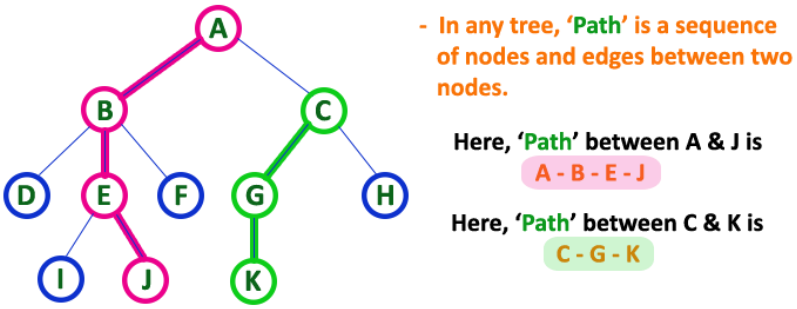
11. Depth

In a tree data structure, the total number of edges from root node to a particular node is called as 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 depth of that tree. In a tree, depth of the root node is '0'.



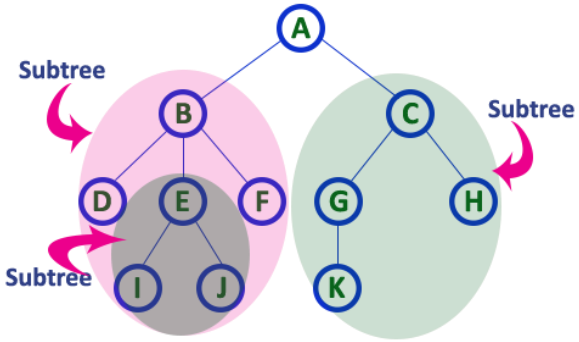
12. Path

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



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.

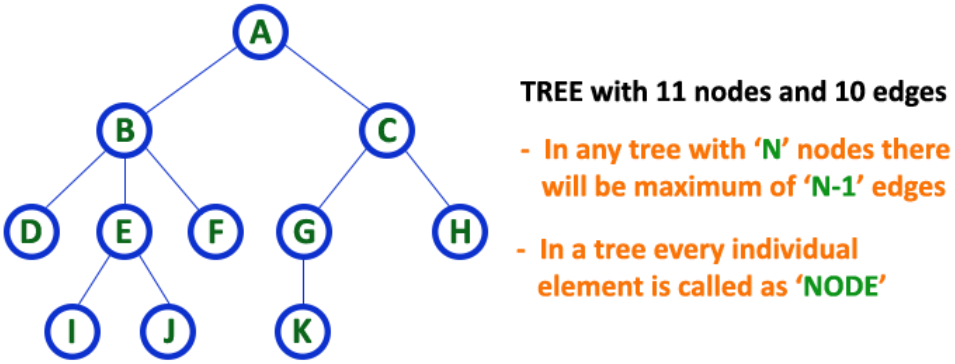


12.2 Tree Representations

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

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

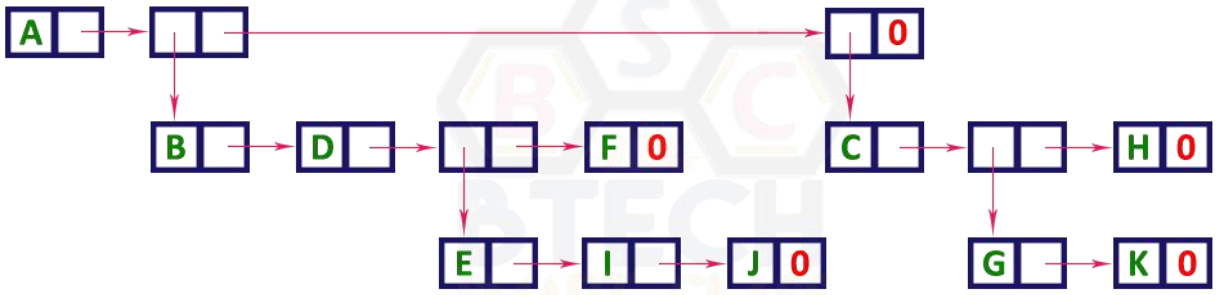
Consider the following tree...



12.2.1 List Representation

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

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

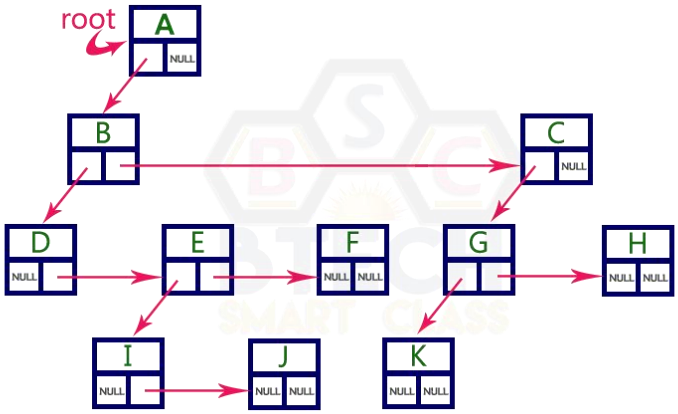


12.2.2 Left Child - Right Sibling Representation

In this representation, we use 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.

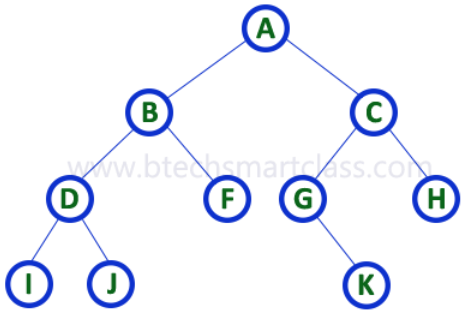
In this representation, every node's data field stores the actual value of that node. If that node has left child, then left reference field stores the address of that left child node otherwise that field stores NULL. If that node has right sibling then right reference field stores the address of right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

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



12.3 Binary Tree

In a normal tree, every node can have any number of children. 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 left child and the other is known as right child.



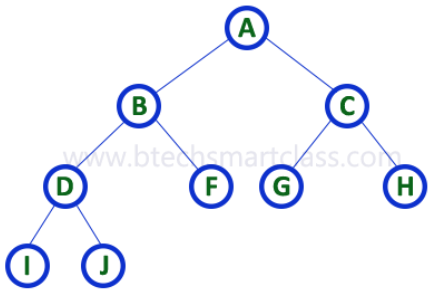
"A tree in which every node can have a maximum of two children is called as Binary Tree."

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

There are different types of binary trees and they are.

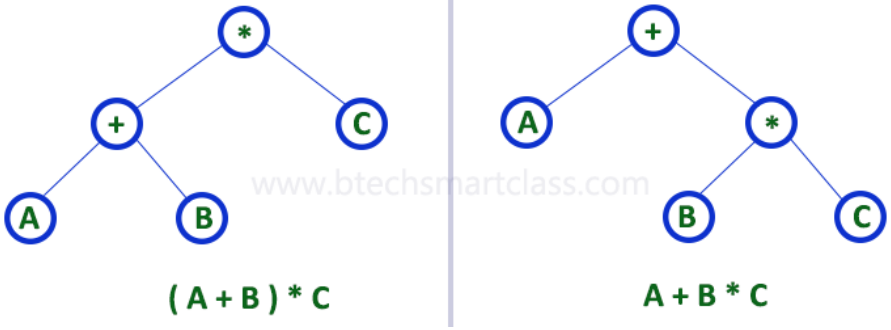
12.3.1 Strictly Binary Tree

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. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows.



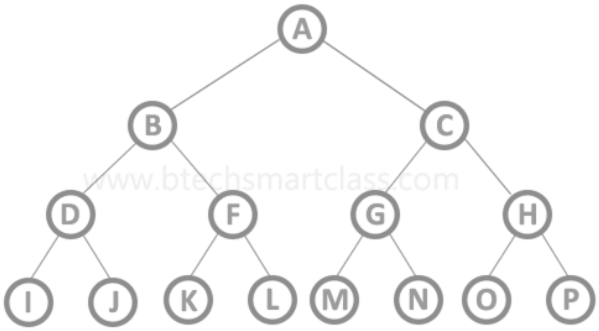
"A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree"

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



12.3.2 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 2level number of nodes. For example at level 2 there must be 22 = 4 nodes and at level 3 there must be 23 = 8 nodes.

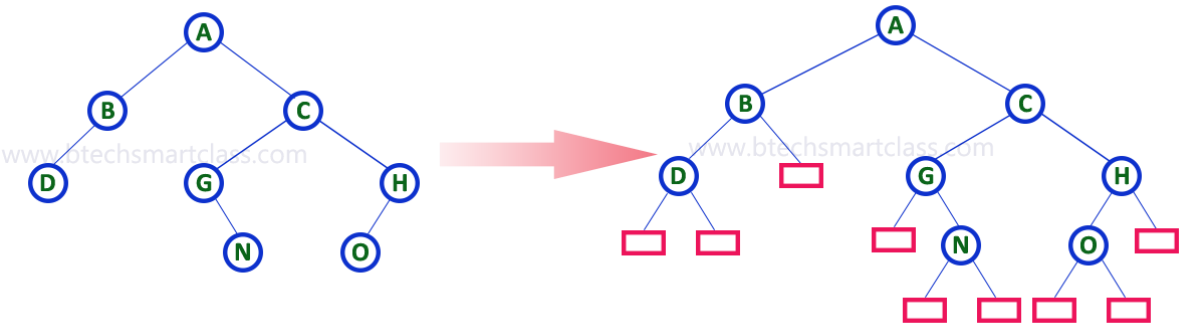


"A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree."

Complete binary tree is also called as Perfect Binary Tree

12.3.3 Extended Binary Tree

A binary tree can be converted into 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 as Extended Binary Tree."



a normal binary tree is converted into full binary tree by adding dummy nodes

12.4 Binary Tree Representations

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

I. Array Representation
II. Linked List Representation

12.4.1 Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.

Consider the above example of binary tree and it is represented as follows.

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

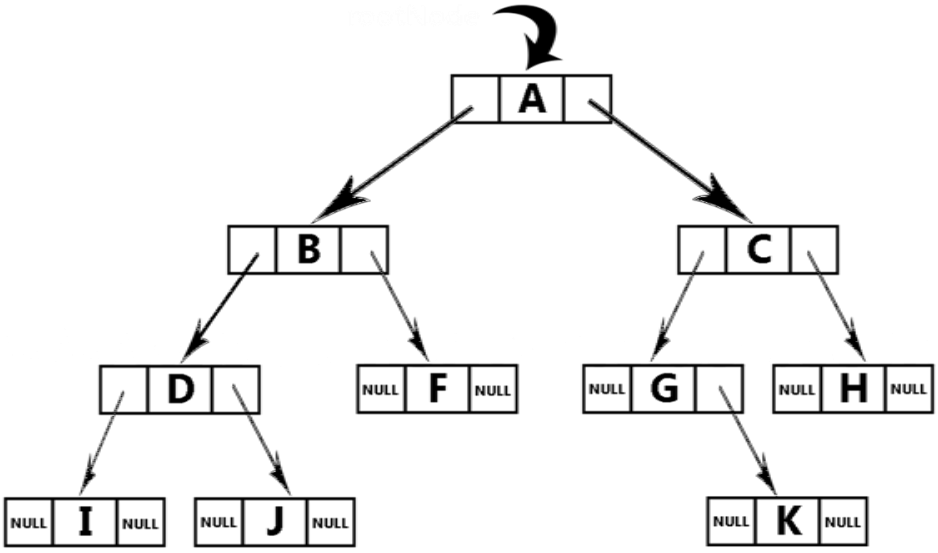
To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n+1 - 1.

12.4.2 Linked List Representation

We use 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



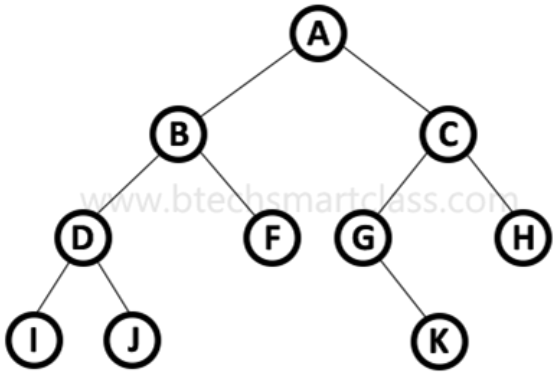
12.5 Binary Tree Traversals

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

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

There are three types of binary tree traversals.

I. In - Order Traversal
II. Pre - Order Traversal
III. Post - Order Traversal



Consider the following binary tree

12.5.1 In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between 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 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 binary tree, first we try to visit left child of root node 'A', but A's left child 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 left most 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 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 right most child in the tree so we stop the process. In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C – H

Algorithm

```
Until all nodes are traversed –
Step 1 – Recursively traverse left subtree.
Step 2 – Visit root node.
Step 3 – Recursively traverse right subtree.
```

12.5.2 Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before 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 left most 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 right most child in the tree. So we stop the process.Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K – H

Algorithm

```
Until all nodes are traversed –
Step 1 – Visit root node.
Step 2 – Recursively traverse left subtree.
Step 3 – Recursively traverse right subtree.
```

12.5.3 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.Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C – A

Algorithm

```
Until all nodes are traversed –
Step 1 – Recursively traverse left subtree.
Step 2 – Recursively traverse right subtree.
Step 3 – Visit root node.
```

Code in Java Programming

```
/ Java program for different tree traversals

class Node
{
    int key;
    Node left, right;
```

```
    public Node(int item)
    {
        key = item;
        left = right = null;
    }
}

class BinaryTree
{
    Node root;

    BinaryTree()
    {
        root = null;
    }

    void printPostorder(Node node)
    {
        if (node == null)
            return;

        printPostorder(node.left); // first recur on left subtree
        printPostorder(node.right); // then recur on right subtree
        System.out.print(node.key + " ");// now deal with the node
    }

    void printInorder(Node node)
    {
        if (node == null)
            return;

         printInorder(node.left);   /* first recur on left child */
         System.out.print(node.key + " ");/* then print the data of node */
         printInorder(node.right);     /* now recur on right child */

    }

    void printPreorder(Node node)
    {
        if (node == null)
            return;


        System.out.print(node.key + " "); /* first print data of node */
        printPreorder(node.left);        /* then recur on left sutree */
        printPreorder(node.right);        /* now recur on right subtree */

    }

    // Wrappers over above recursive functions
    void printPostorder()  {     printPostorder(root);  }
    void printInorder()    {     printInorder(root);   }
    void printPreorder()   {     printPreorder(root);  }

    // Driver method
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
```

```
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        System.out.println("Preorder traversal of binary tree is ");
        tree.printPreorder();

        System.out.println("\nInorder traversal of binary tree is ");
        tree.printInorder();

        System.out.println("\nPostorder traversal of binary tree is ");
        tree.printPostorder();
    }
}
```

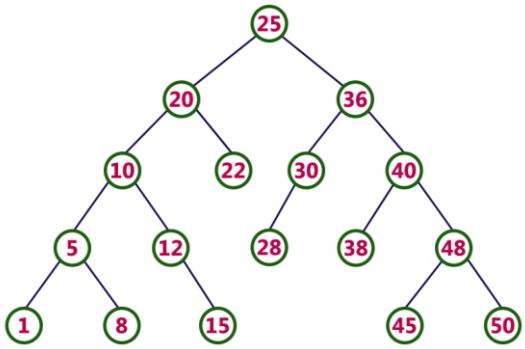
12.6 Binary Search Tree

To enhance the performance of binary tree, we use special type of binary tree known as Binary Search Tree. Binary search tree mainly focus on the search operation in 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."

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.



12.6.1 Operations on a Binary Search Tree

The following operations are performed on a binary search tree.

1. Search
2. Insertion
3. Deletion

12.6.1.1 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 matching, then display "Given node found!!!" and terminate the function
Step 4: If both are not matching, 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 found exact element or we completed with a leaf node
Step 8: If we reach to the node with search value, then display "Element is found" and
        terminate the function.
Step 9: If we reach to a leaf node and it is also not matching, then display "Element not
        found" and terminate the function.
```

12.6.1.2 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 set root to newNode.
Step 4: If the tree is Not Empty, then check whether 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 step until we reach to a leaf node (e.i., reach to NULL).
Step 7: After reaching a leaf node, then isert the newNode as left child if newNode is
        smaller or equal to that leaf else insert it as right child.
```

12.6.1.3 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 has follwing 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

```
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

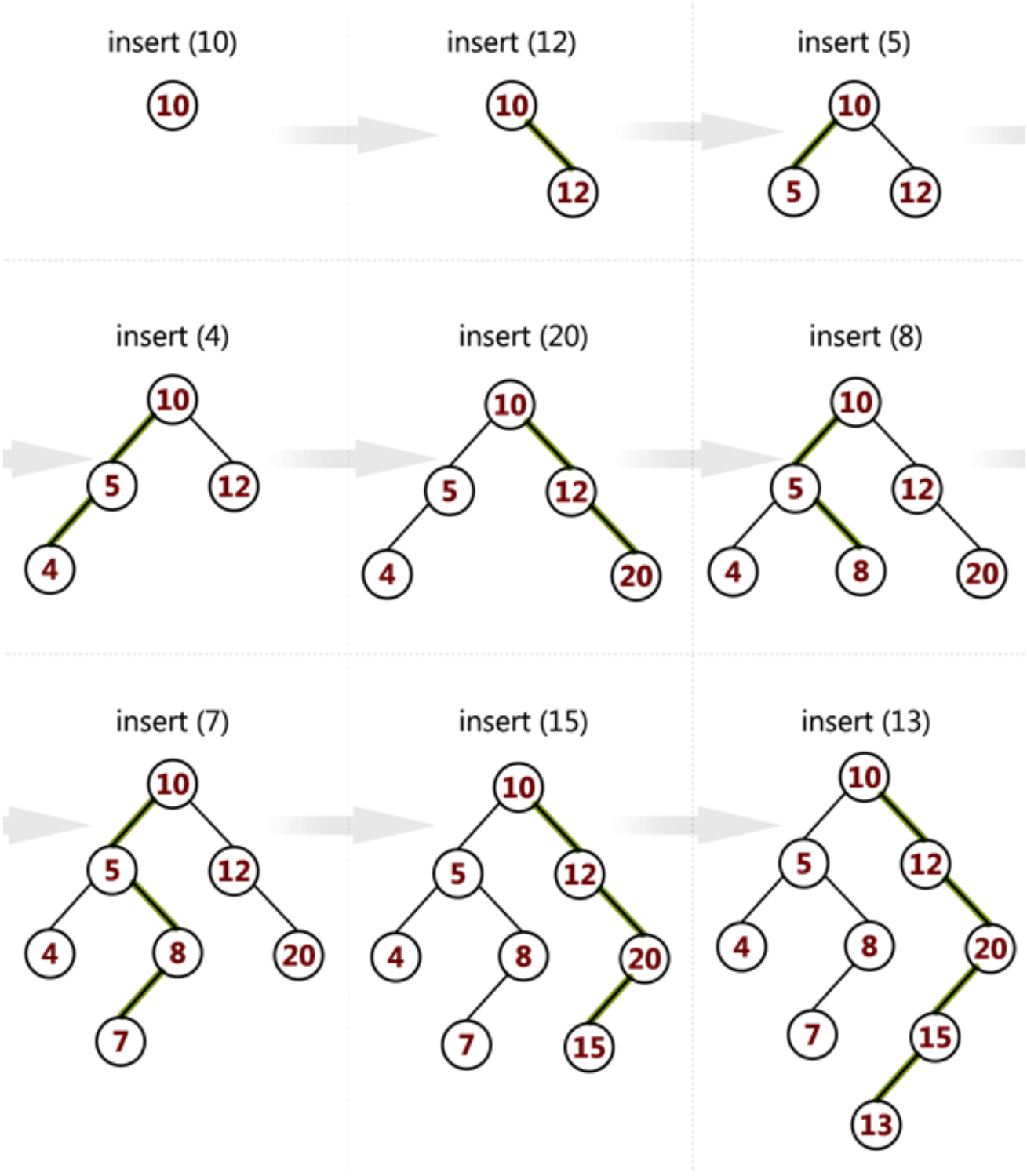
```
Step 1: Find the node to be deleted using search operation
Step 2: If it has only one child, then create a link between its parent and child nodes.
Step 3: Delete the node using free function and terminate the function.
```

Case 3: Deleting a node with two children

```
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 found in above step.
Step 4: Then, check whether deleting node came to case 1 or case 2 else gotosteps 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 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.

12.7 AVL Tree

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the hieghts of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree,

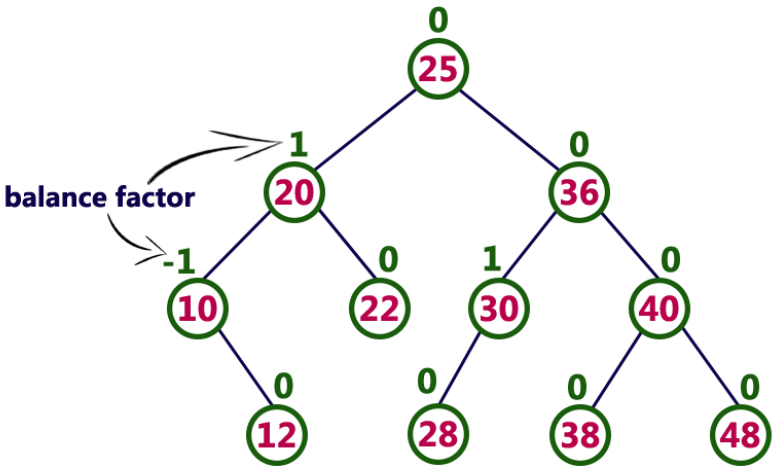
every node maintains a extra information known as balance factor. The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

"An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1."

Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we are calculating as follows.

Balance factor = heightOfLeftSubtree–heightOfRightSubtree

Example



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

"Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees."

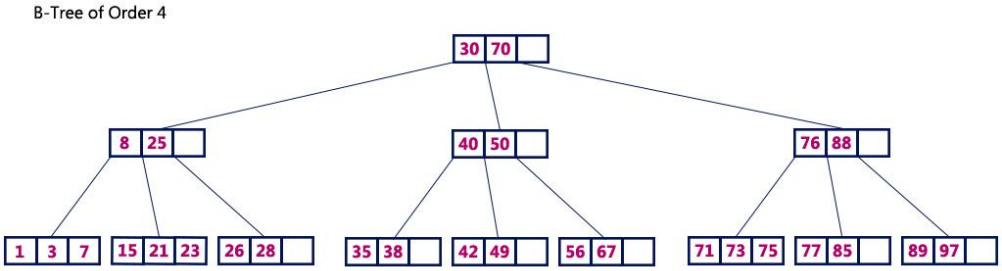
12.8 B – Trees

In a binary search tree, AVL Tree, Red-Black tree etc., every node can have only one value (key) and maximum of two children but there is another type of search tree called B-Tree in which a node can store more than one value (key) and it can have more than two children. B-Tree was developed in the year of 1972 by Bayer and McCreight with the name Height Balanced m-way Search Tree. Later it was named as B-Tree. B-Tree can be defined as follows.

"B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node."

B-Tree of Order m has the following properties.

Property #1 - All the leaf nodes must be at same level.
Property #2 - All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.
Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.
Property #4 - If the root node is a non leaf node, then it must have at least 2 children.
Property #5 - A non leaf node with n-1 keys must have n number of children.
Property #6 - All the key values within a node must be in Ascending Order.



- **Search Operation in B-Tree**

In a B-Ttree, the search operation is similar to that of Binary Search Tree. In a Binary search tree, the search process starts from the root node and every time we make a 2-way decision (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but every time we make n-way decision where n is the total number of children that node has. In a B-Ttree, 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 first key value of root node in the tree.
Step 3: If both are matching, then display "Given node found!!!" and terminate the function
Step 4: If both are not matching, then check whether search element is smaller or larger
        than that key value.
Step 5: If search element is smaller, then continue the search process in left subtree.
Step 6: If search element is larger, then compare with next key value in the same node and
        repeate step 3, 4, 5 and 6 until we found exact match or comparision completed with
        last key value in a leaf node.
Step 7: If we completed with last key value in a leaf node, then display "Element is not
        found" and terminate the function.
```

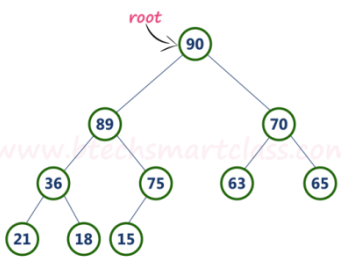
- **Insertion Operation in B-Tree**

In a B-Tree, the new element must be added only at leaf node. That means, always the new keyValue is attached to leaf node only. The insertion operation is performed as follows...

```
Step 1: Check whether tree is Empty.
Step 2: If tree is Empty, then create a new node with new key value and insert into the
        tree as a root node.
Step 3: If tree is Not Empty, then find a leaf node to which the new key value cab be added
        using Binary Search Tree logic.
Step 4: If that leaf node has an empty position, then add the new key value to that leaf
        node by maintaining ascending order of key value within the node.
Step 5: If that leaf node is already full, then split that leaf node by sending middle
        value to its parent node. Repeat tha same until sending value is fixed into a node.
Step 6: If the spilting is occuring to the root node, then the middle value becomes new
        root node for the tree and the height of the tree is increased by one.
```

13. Heap

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 value. A heap data structure, some time called as 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 a order according to values based on Max heap or Min heap.
- Property #2 (Structural): All levels in a heap must full, except last level and nodes must be filled from left to right strictly.

13.1 Max Heap

"Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes. And last leaf node can be alone."

13.1.1 Operations on Max Heap

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

I. Finding Maximum
II. Insertion
III. Deletion

I. Finding Maximum

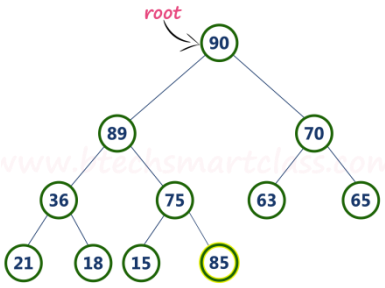
Finding the node which has maximum value in a max heap is very simple. In max heap, the root node has the maximum value than all other nodes in the max heap. So, directly we can display root node value as maximum value in max heap.

Insertion Operation in Max Heap

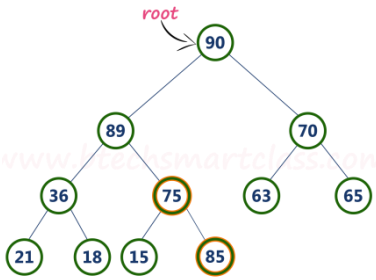
II. Insertion

```
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 nede (or)
        newNode reached to root.
```

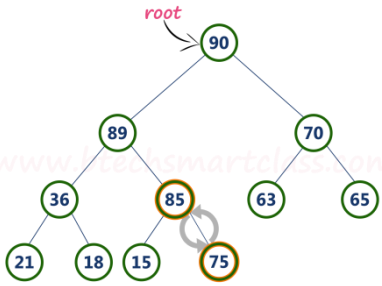
Example:



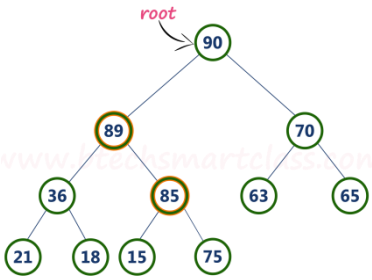
Step-1



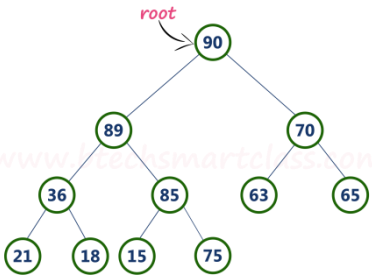
Step-2



Step-3



Step-4



Step-5

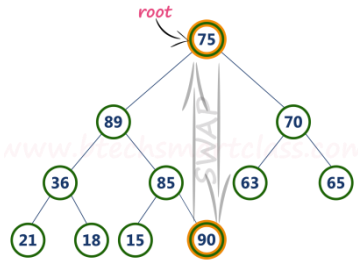
III. Deletion

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

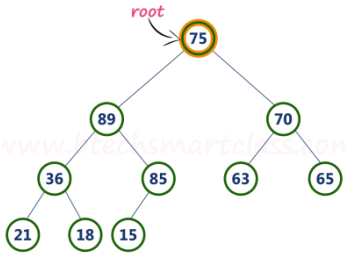
Deleting root node from a max heap is title difficult as it disturbing the max heap properties. We use the following steps to delete root node from a max heap...

```
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 root value is smaller than its left child, then compare 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 rith child.
        otherwise stop the process.
Step 8: Repeat the same until root node is fixed at its exact position.
```

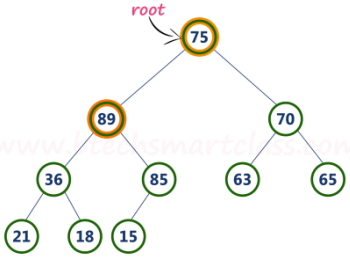
Example



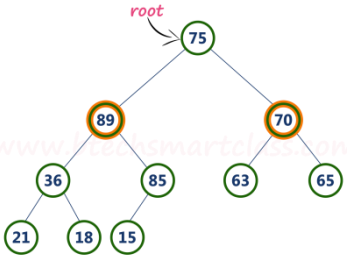
Step-1



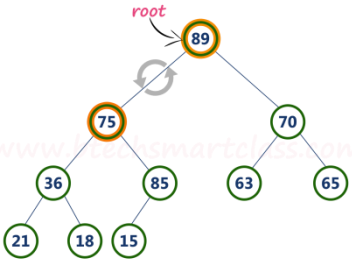
Step-2



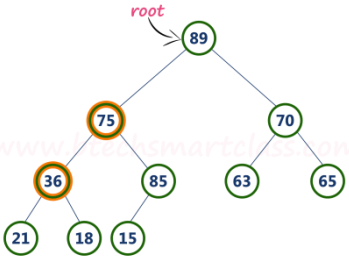
Step-3



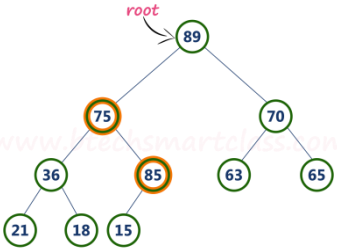
Step-4



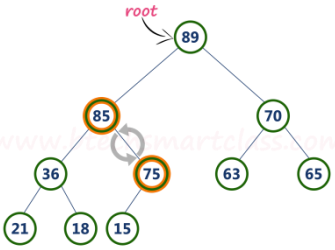
Step-5



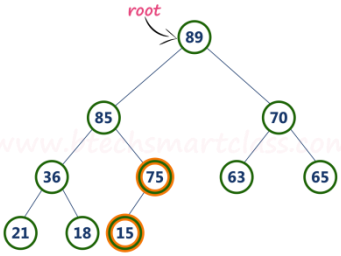
Step-6



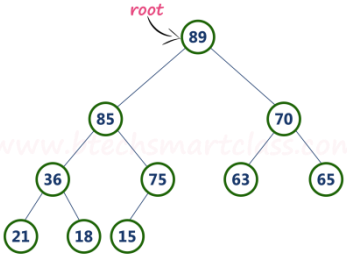
Step-7



Step-8



Step-9



Step-4

14. GRAPHS

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of linkes known as edges (or Arcs) which connets the vertices. A graph is defined as:

"Graph is a collection of vertices and arcs which connects vertices in the graph"

"Graph is a collection of nodes and edges which connects nodes in the graph"

A graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

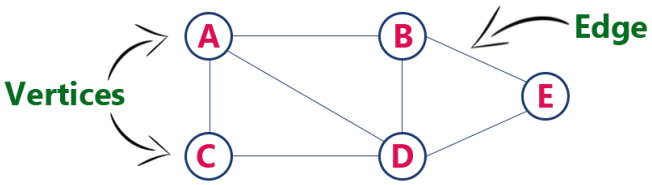
Example

The following is a graph with 5 vertices and 6 edges.

This graph G can be defined as G = (V , E)

Where V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



14.1 Graph Terminology

I. Vertex

A individual data element of a graph is called as **Vertex**. Vertex is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

II. Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

a) **Undirected Edge -** An undirected egde is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

b) **Directed Edge** - A directed egde is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

c) **Weighted Edge -** A weighted egde is an edge with cost on it.

III. Undirected Graph

A graph with only undirected edges is said to be undirected graph.

IV. Directed Graph

A graph with only directed edges is said to be directed graph.

V. Mixed Graph

A graph with undirected and directed edges is said to be mixed graph.

VI. End vertices or Endpoints

The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

VII. Origin

If an edge is directed, its first endpoint is said to be origin of it.

VIII. Destination

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

IX. Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

X. **Incident**
An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

XI. **Outgoing Edge**
A directed edge is said to be outgoing edge on its orign vertex.

XII. **Incoming Edge**
A directed edge is said to be incoming edge on its destination vertex.

XIII. **Degree**
Total number of edges connected to a vertex is said to be degree of that vertex.

XIV. **Indegree**
Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

XV. **Outdegree**
Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

XVI. **Parallel edges or Multiple edges**
If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

XVII. **Self-loop**
An edge (undirected or directed) is a self-loop if its two endpoints coincide.

XVIII. **Simple Graph**
A graph is said to be simple if there are no parallel and self-loop edges.

XIX. **Path**
A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

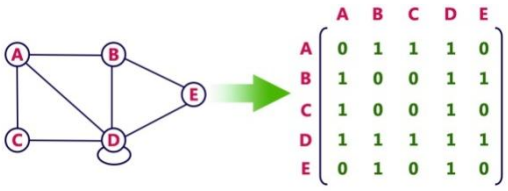
14.2 Graph Representations

Graph data structure is represented using following representations...

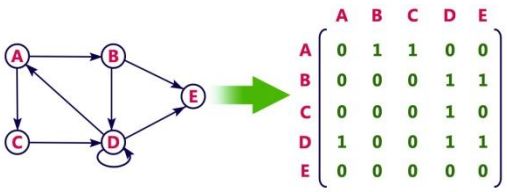
1. Adjacency Matrix
2. Incidence Matrix
3. Adjacency List

14.2.1 Adjacency Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.



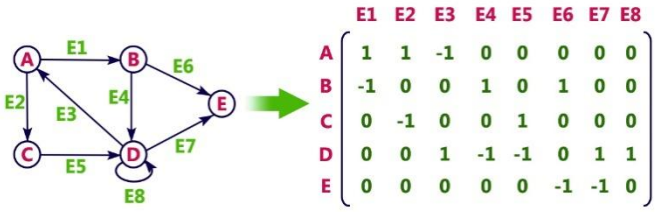
undirected graph representation



directional Graph Representation

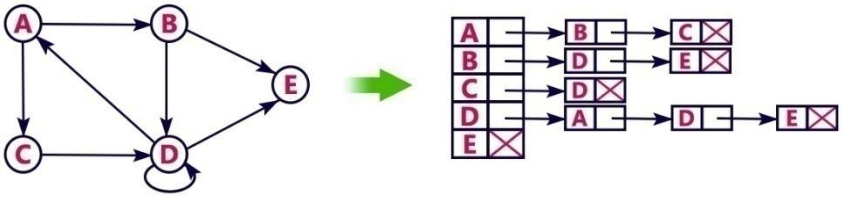
14.2.2 Incidence Matrix

in this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represent vertices and columns represent edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

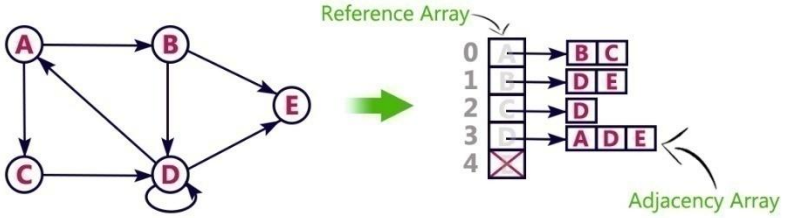


14.2.3 Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. For example, consider the following directed graph representation implemented using linked list.



This representation can also be implemented using array as follows.



14.3 Graph Traversals

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.

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

1. DFS (Depth First Search)
2. BFS (Breadth First Search)

14.3.1 DFS (Depth First Search)

DFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

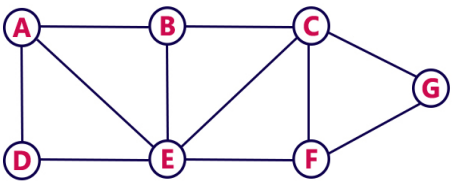
We use the following steps to implement DFS traversal.

Algorithm:

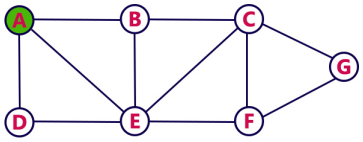
```
Step 1: Define a Stack of size total number of vertices in the graph.
Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on
        to the Stack.
Step 3: Visit any one of the adjacent vertex of the verex which is at top of the stack
        which is not visited and push it on to the stack.
Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of
        the stack.
Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex
        from the stack.
Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.
Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges
        from the graph
```

"Back tracking is coming back to the vertex from which we came to current vertex."

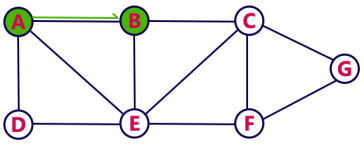
Consider the following example graph to perform DFS traversal



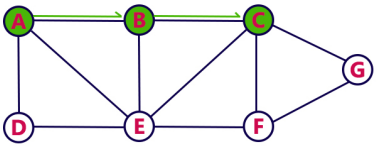
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

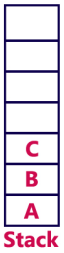


Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

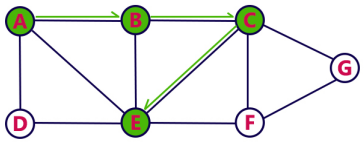


tep 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

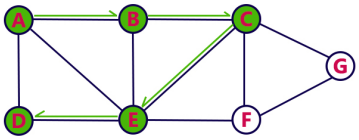




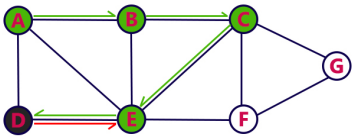
Step 4:
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



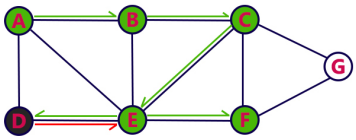
Step 5:
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



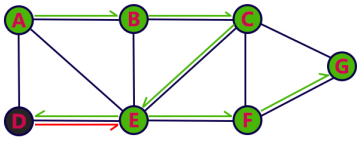
Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



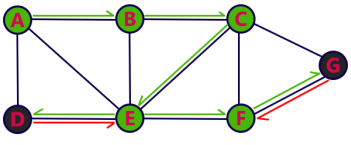
Step 7:
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

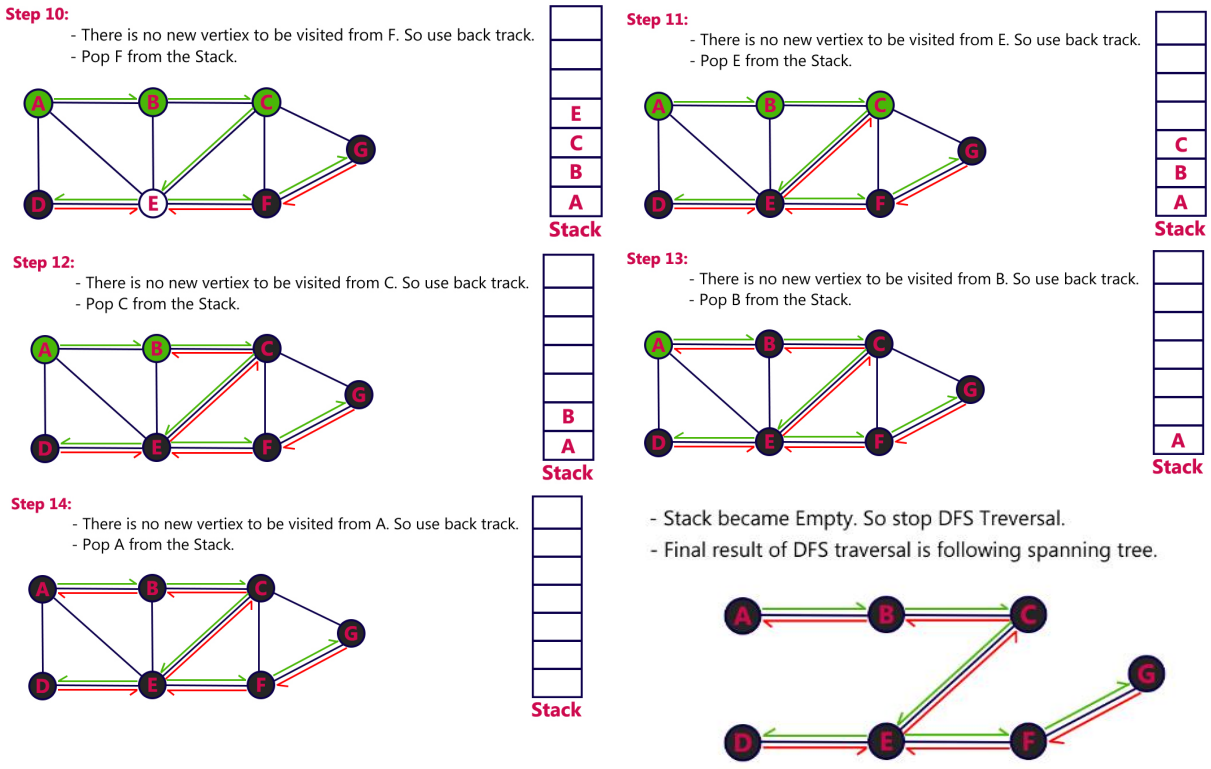


Step 8:
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



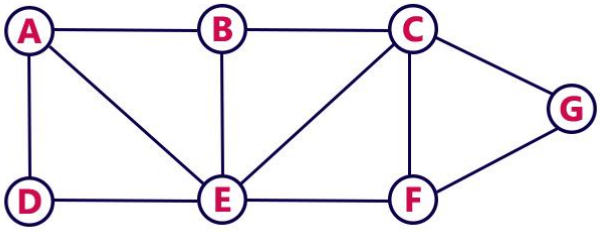


14.3.2 BFS (Breadth First Search)

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph. We use the following steps to implement BFS traversal.

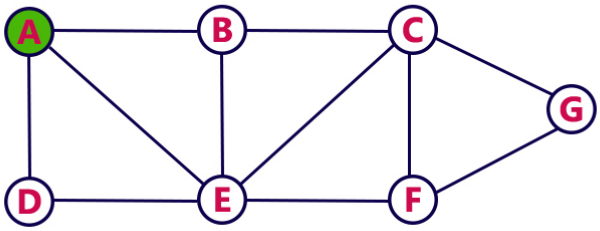
```
Step 1: Define a Queue of size total number of vertices in the graph.
Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it
        into the Queue.
Step 3: Visit all the adjacent vertices of the verex which is at front of the Queue which
        is not visited and insert them into the Queue.
Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then
        delete that vertex from the Queue.
Step 5: Repeat step 3 and 4 until queue becomes empty.
Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges
        from the graph
```

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

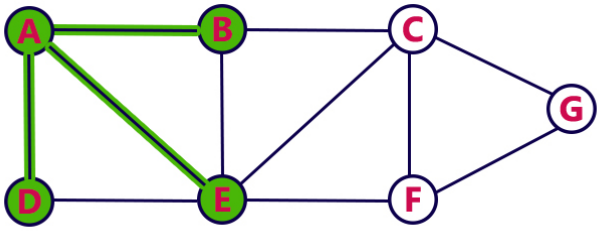


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

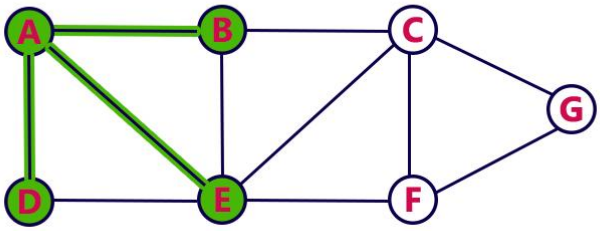


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

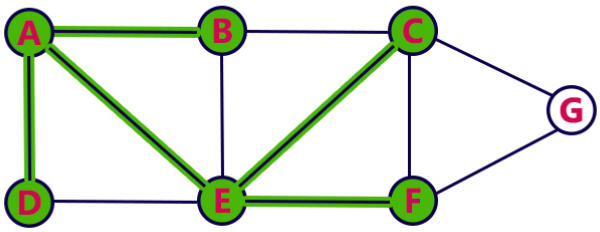


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

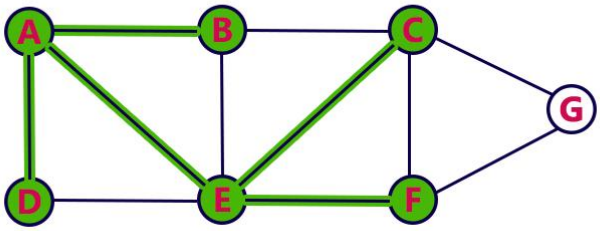


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

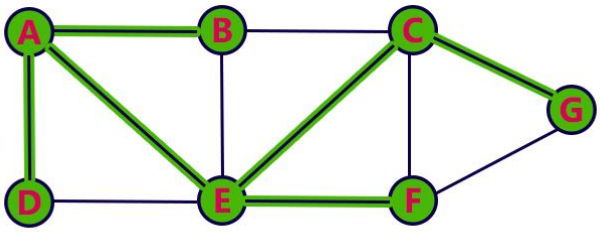


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

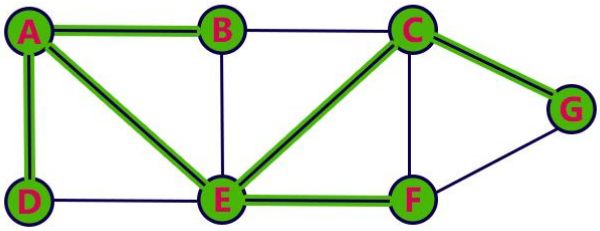


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

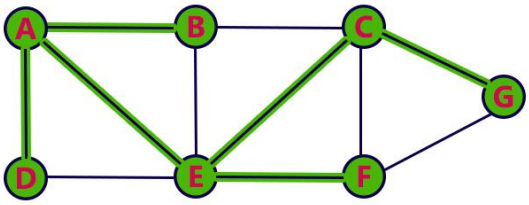


Queue

						G

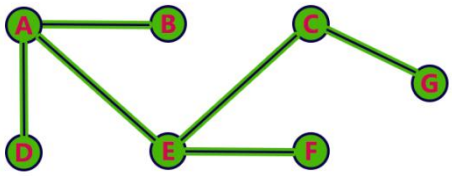
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



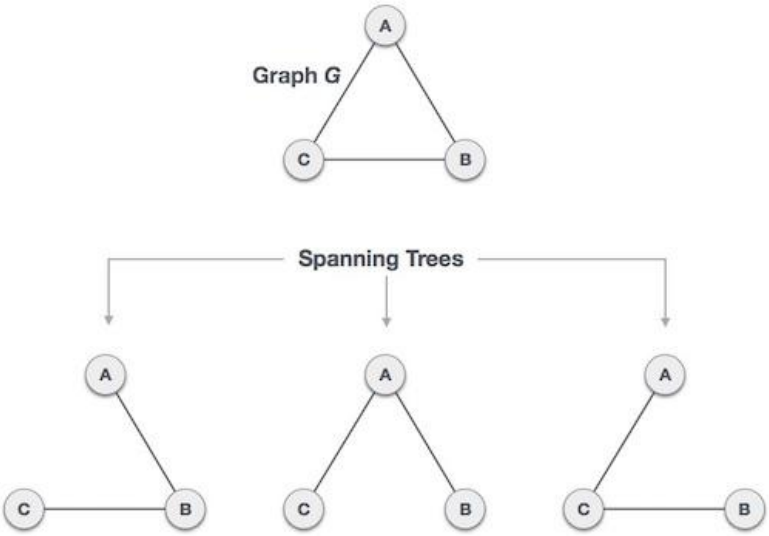
Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



14.4 Spanning Tree

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected. every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



A complete undirected graph can have maximum nn-2 number of spanning trees, where n is the number of nodes.

14.4.1 Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

14.4.2 Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

15. SEARCH ALGORITHM

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

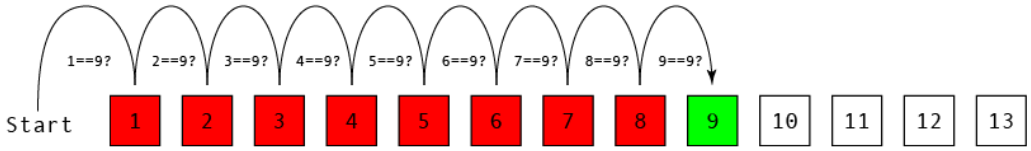
15.1 Linear Search Algorithm

Linear search algorithm finds given element in a list of elements with O(n) time complexity where n is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching, then results with element found otherwise search element is compared with next element in the list. If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is

"Element not found in the list". That means, the search element is compared with element by element in the list.

Linear search is implemented using following steps.

```
Step 1: Read the search element from the user
Step 2: Compare, the search element with the first element in the list.
Step 3: If both are matching, then display "Given element found!!!" and terminate the
        function
Step 4: If both are not matching, then compare search element with the next element in the
        list.
Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in
        the list.
Step 6: If the last element in the list is also doesn't match, then display "Element not
        found!!!" and terminate the function.
```

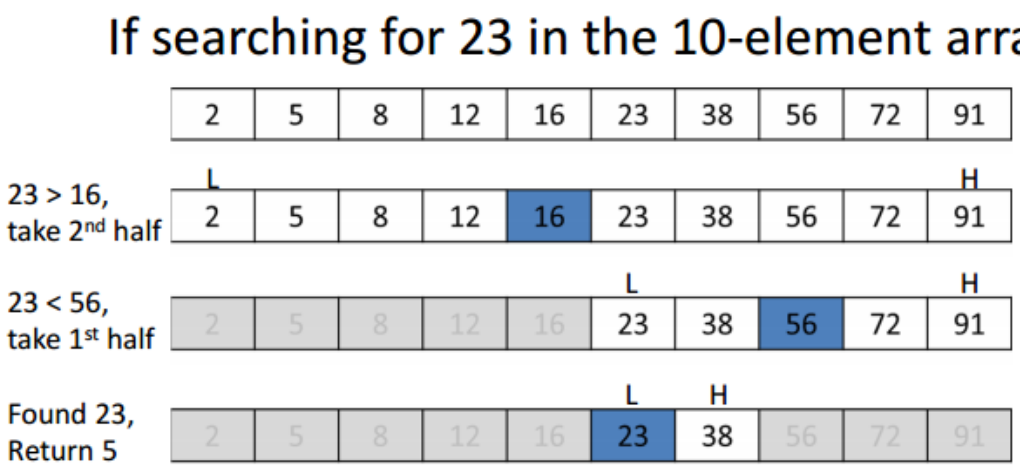


15.2 Binary Search Algorithm

Binary search algorithm finds given element in a list of elements with O(log n) time complexity where n is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arranged in a order. The binary search cannot be used for list of element which are in random order. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps.

```
Step 1: Read the search element from the user
Step 2: Find the middle element in the sorted list
Step 3: Compare, the search element with the middle element in the sorted list.
Step 4: If both are matching, then display "Given element found!!!" and terminate the
        function
Step 5: If both are not matching, then check whether the search element is smaller or
        larger than middle element.
Step 6: If the search element is smaller than middle element, then repeat steps 2, 3, 4 and
        5 for the left sublist of the middle element.
Step 7: If the search element is larger than middle element, then repeat steps 2, 3, 4 and
        5 for the right sublist of the middle element.
Step 8: Repeat the same process until we find the search element in the list or until
        sublist contains only one element.
Step 9: If that element also doesn't match with the search element, then display "Element
        not found in the list!!!" and terminate the function.
```



15.3 Hashing

In all search techniques like linear search, binary search and search trees, the time required to search an element is depending on the total number of element in that data structure. In all these search techniques, as the number of element are increased the time required to search an element also increased linearly.

Hashing is another approach in which time required to search an element doesn't depend on the number of element. Using hashing data structure, an element is searched with constant time complexity. Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

"Hashing is the process of indexing and retrieving element (data) in a datastructure to provide faster way of finding the element using the hash key."

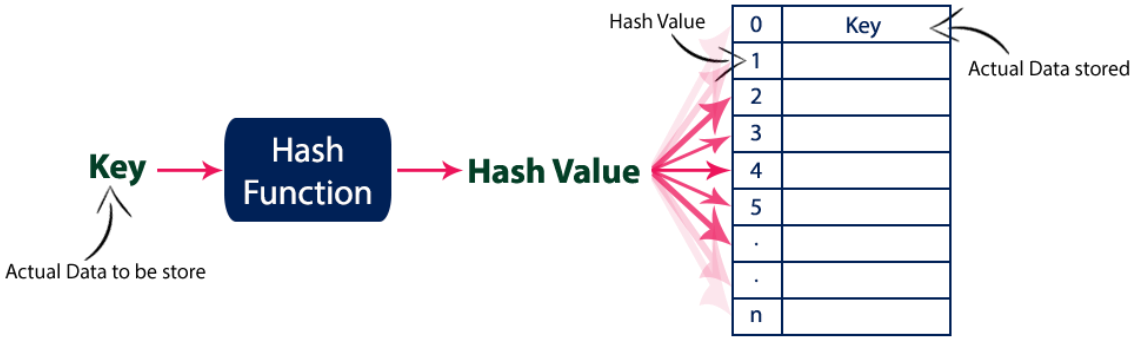
Here, hash key is a value which provides the index value where tha actual data is likely to store in the data structure.

In this data structure, we use a concept called Hash table to store data. All the data values are inserted into the hash table based on the hash key value. Hash key value is used to map the data with index in the hash table. And the hash key is generated for every data using a hash function. That means every entry in the hash table is based on the key value generated using a hash function. Hash Table is defined as follows...

"Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity (i.e. O(1))."

Hash tables are used to perform the operations like insertion, deletion and search very quickly in a data structure. Using hash table concept insertion, deletion and search operations are accomplished in constant time. Generally, every hash table make use of a function, which we'll call the hash function to map the data into the hash table. A hash function is defined as follows.

"Hash function is a function which takes a piece of data (i.e. key) as input and outputs an integer (i.e. hash value) which maps the data to a particular index in the hash table."



15.3.1 Hashing function

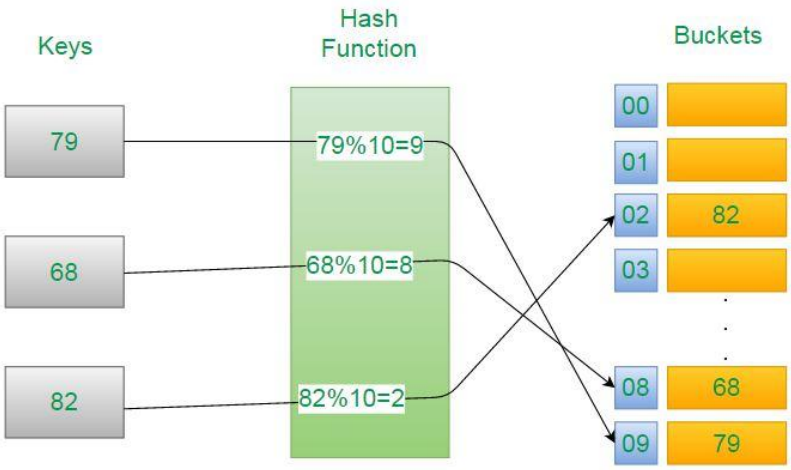
There are many ways to create an hash function. In this chapter, we will understand division/modulo method to create hash function.

Division Method

If there are m slots available, then the key is mapped to given m slots using following formula: -

h(k) = k mod m

For example: - Let us say apply division approach to find hash value for some values considering number of buckets be 10 as shown below.

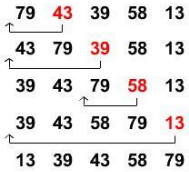


The division method is generally a good choice, unless the key happens to have some undesirable properties. For example, if the table size is 10 and all of the keys end in zero. In this case, the choice of hash function and table size needs to be carefully considered. The best table sizes are prime numbers.

16. SORTING

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

16.1 Insertion Sort



Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

The insertion sort algorithm is performed using following steps...

```
Step 1: Assume that first element in the list is in sorted portion of the list and
        remaining all elements are in unsorted portion.
Step 2: Consider first element from the unsorted list and insert that element into the
        sorted list in order specified.
Step 3: Repeat the above process until all the elements from theunsorted list are moved
        into the sorted list.
```

16.2 Selection Sort

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped. Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.The selection sort algorithm is performed using following steps.

```
Step 1: Select the first element of the list (i.e., Element at first position in the list).
Step 2: Compare the selected element with all other elements in the list.
Step 3: For every comparison, if any element is smaller than selected element (for
        Ascending order), then these two are swapped.
Step 4: Repeat the same procedure with next position in the list till the entire list is
        sorted.
```

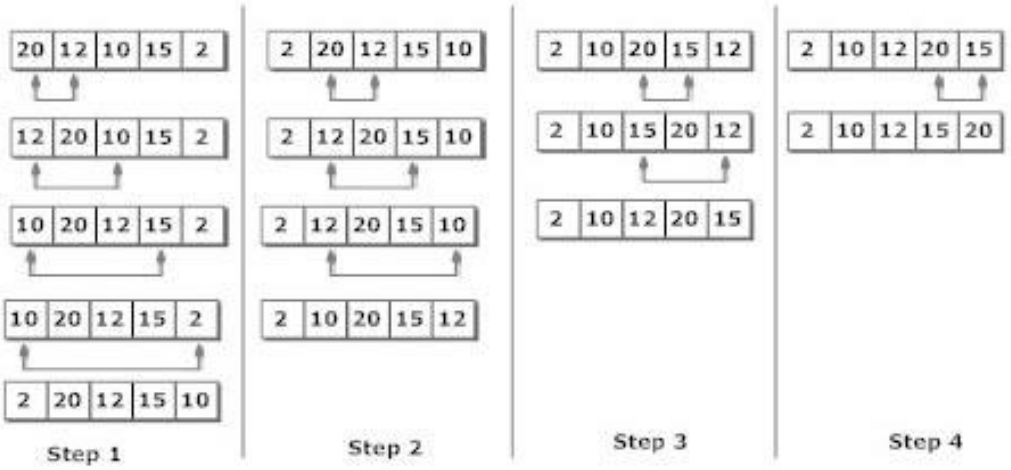


Figure: Selection Sort

16.3 Bubble Sort Algorithm

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you mustn't swap the element. Then, again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.

If there are n elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result.

Algorithm:

```
Step 1: Start
Step 2: go through all elements in array
Step 3: check if list[i] > list[i+1]
Step 4: swap(list[i], list[i+1])
Step 5: Repeat step 3,4 until all element are sorted
```

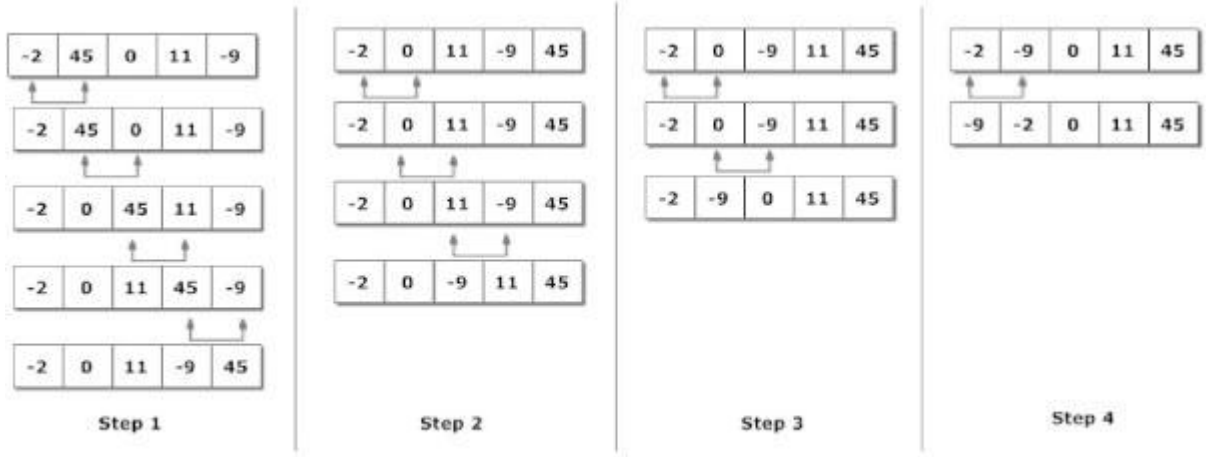


Figure: Working of Bubble sort algorithm

16.4 Quick Sort

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

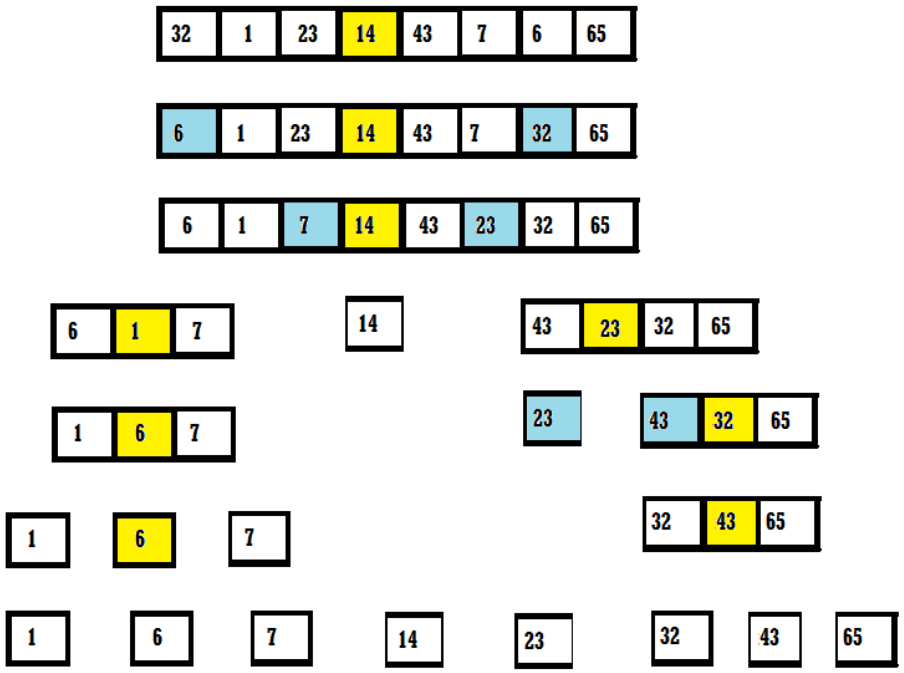
Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of O(n2), where n is the number of items.

Quick Sort Pivot Algorithm:

```
Step 1 – Choose the highest index value has pivot
Step 2 – Take two variables to point left and right of the list excluding pivot
Step 3 – left points to the low index
Step 4 – right points to the high
Step 5 – while value at left is less than pivot move right
Step 6 – while value at right is greater than pivot move left
Step 7 – if both step 5 and step 6 does not match swap left and right
Step 8 – if left ≥ right, the point where they met is new pivot
```

Quick Sort Algorithm

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



16.5 Heap Sort

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts :

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

We can define heap as :

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

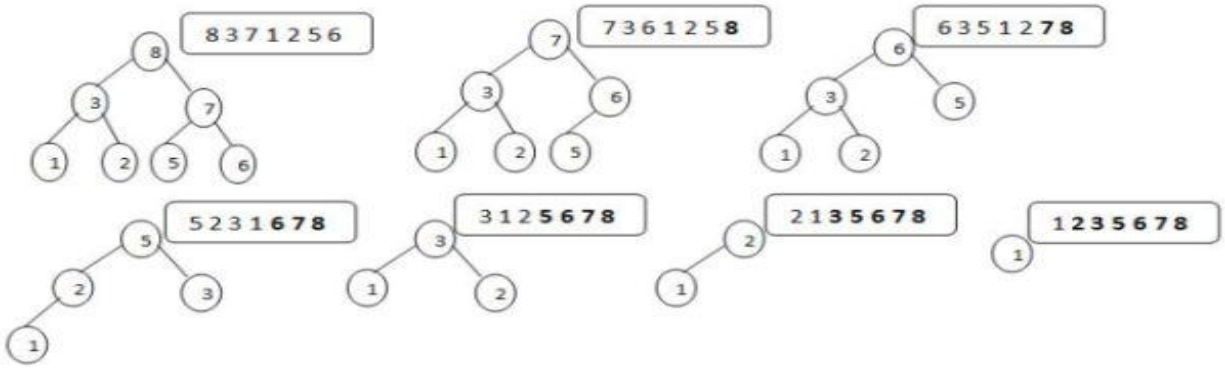
Shape Property : Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

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

Algorithm:

```
Step 1: create a Heap data structure(Max-Heap or Min-Heap).
Step 2: put the first element of the heap in array.
Step 3: make heap using the remaining elements.
Step 4: repeat step 2 and 3 untill we have the complete sorted list in our array.
```

Example:- The fig. shows steps of heap-sort for list (2 3 7 1 8 5 6)



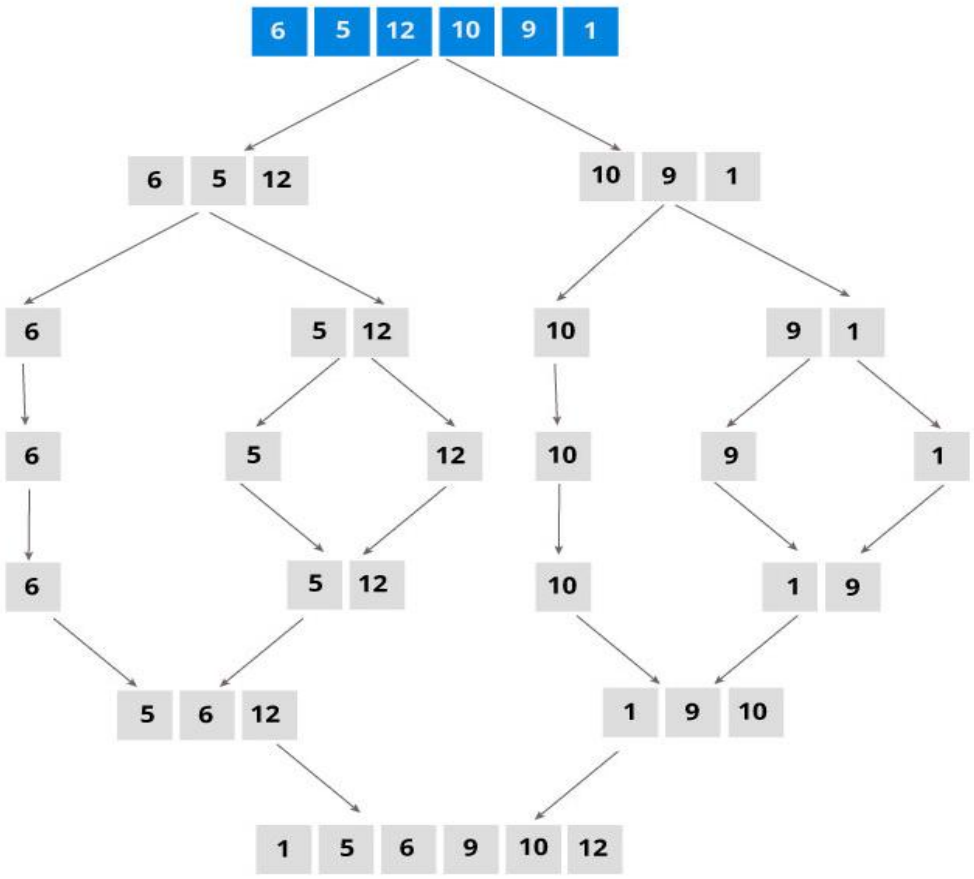
16.6 Merge Sort Algorithm

Merge Sort is a kind of Divide and Conquer algorithm in computer programming. We divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Algorithm

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

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



17. References

I. B, %. (n.d.). Data Structure . Retrieved from http://btechsmartclass.com/DS/U1_T1.html
II. Data Structures and. Algorithms in Java™. Sixth Edition. Michael T. Goodrich. Department of Computer Science. University of California,
III. Data Structures and Algorithms in Java .Michael T. Goodrich and Roberto Tamassia
IV. Data Structure and Algorithms. Retrieved from https://www.tutorialspoint.com/data_structures_algorithms/
V. Data Structure and Algorithms. Retrieved from www.studytonight.com/data-structures/introduction-to-data-structures
VI. Data Structure and Algorithms. Retrieved from http://www.geeksforgeeks.org/data-structures/