**Unit-4:-**

**Data Structure:-**

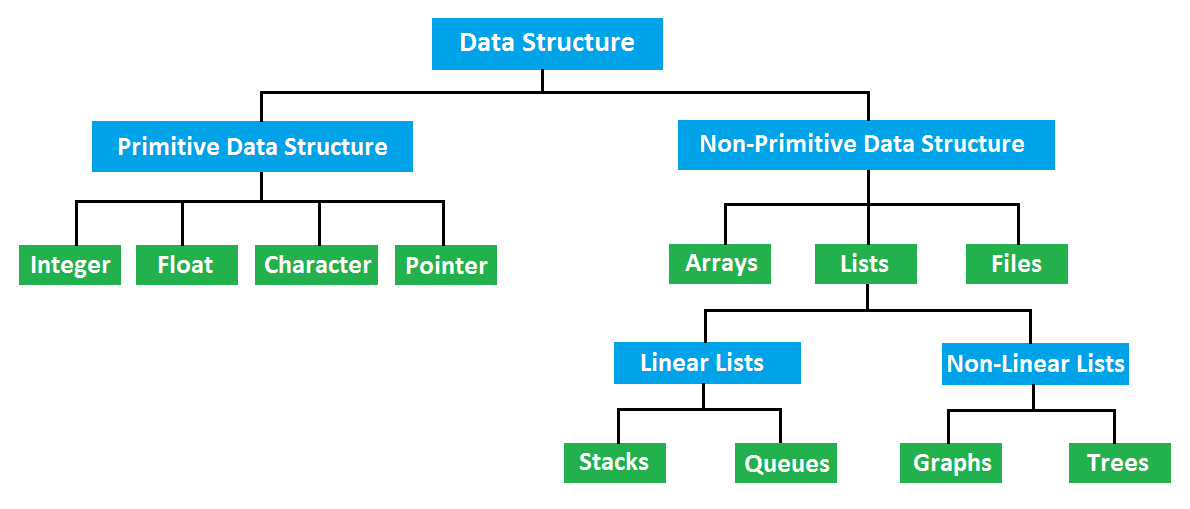
* We can say that data structure is special way to organization and management of the data. Data structure provides the functionality to manage the data and information.

**Data structures:-**Data structures provide the way to manage data efficiently as per our requirement.

**Classification Of Data Structure:-**

The classification of data structure mainly consists of :

1. **Primitive data structure**
2. **Non-primitive data structure**



**Primitive data structure :-**

The primitive data structures are known as basic data structures. These data structures are directly operated upon by the machine instructions. Normally, primitive data structures have different representation on different computers.

Example of primitive data structure :

* **Integer**
* **Float**
* **Character**
* **Pointer**

**Non-Primitive data structure :-**

The non-primitive data structures are highly developed complex data structures. Basically, these are developed from the primitive data structure. The non-primitive data structure is responsible for organizing the group of homogeneous and heterogeneous data elements.

Example of Non-primitive data structure :

* **Arrays**
* **Lists**
* **Files**

**Integer :**

The integers are signed or unsigned whole numbers with the specified range such as 5, 39, -1917, 0 etc. They have no fractional parts. Integers can be positive or negative but whether or not they can have negative values, it depends upon the integer types.

**Float :**

Float refers floating point or real number. It can hold a real number or a number having a fractional part like 3.112 or 588.001 etc. The decimal point signals that it is a floating point number, not an integer. The number 15 is an integer but 15.0 is a floating point number.

**Character :**

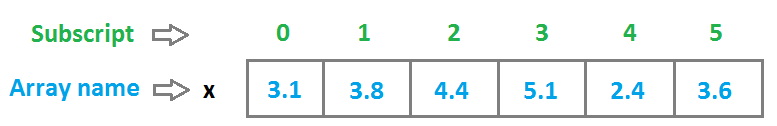
It can store any member of the basic character set. If a character from this set is stored in a character variable, its value is equivalent to the integer code of that character basically known as [ASCII](https://en.wikipedia.org/wiki/ASCII) code. It can hold one letter/symbol like a, B, d etc. Characters can also be of different types.

**Pointer :**

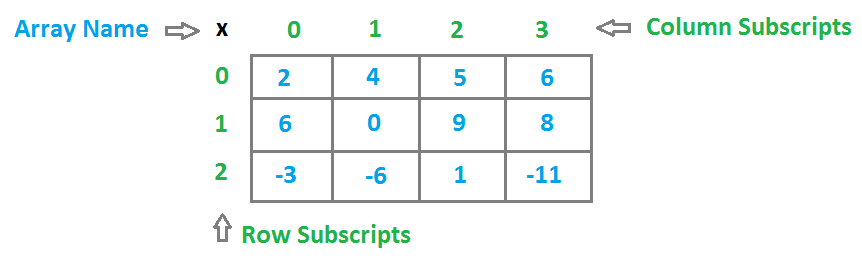
A pointer is but a variable-like name points or represents a storage location in memory (RAM). RAM contains many cells to store values. Each cell in memory is 1 byte and has a unique address to identify it. The memory address is always an unsigned integer.

**Arrays :**

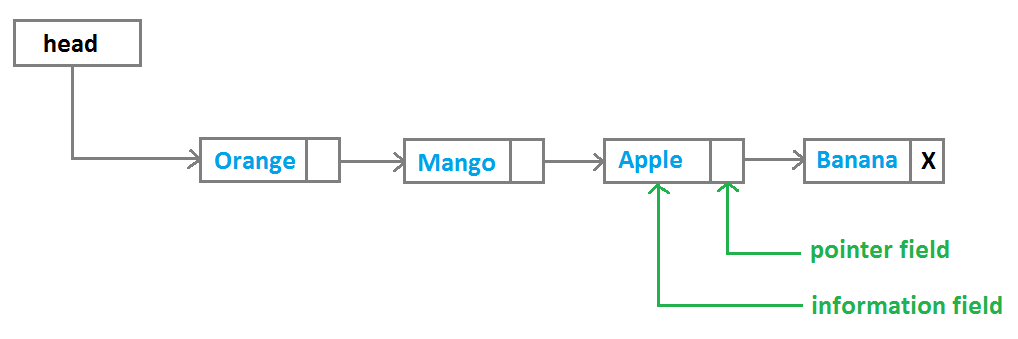
Arrays are the set of homogeneous data elements stored in RAM. So, they can hold only one type of data. The data may be all integers, all floating numbers or all characters. Values in an array are identified using array name with subscripts. Single sub-scripted variables are known as a one-dimensional array or linear array; two sub-scripted variables are referred as a two-dimensional array.



One Dimensional Array

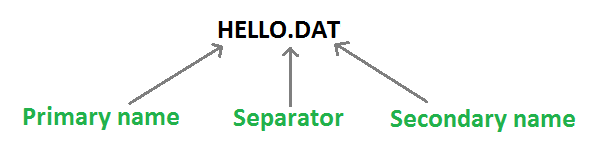
Two Dimensional Array

**Lists :**

A list is a collection of a variable number of data items. Lists fall in the non-primitive type of data structure in the classification of data structure. Every element on a list contains at least two fields, one is used to store data and the other one is used for storing the address of next elementLinear linked lists

**Files :**

Files contain data or information, stored permanently in the secondary storage device such as Hard Disk and Floppy Disk. It is useful when we have to store and process a large amount of data. A file stored in a storage device is always identified using a file name like **HELLO.DAT** or **TEXTNAME.TXT** and so on. A file name normally contains a primary and a secondary name which is separated by a **dot(.)**.

Files

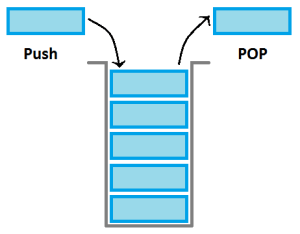
**Stack :**

Like arrays, a stack is also defined as an ordered collection of elements. A stack is a non-primitive linear data structure having a special feature that we can delete and insert elements from only one end, referred as **TOP** of the stack. The stack is also known as **Last In First Out (LIFO)** type of data structure for this behaviour.

When we perform insertion or deletion operation on a stack, its base remains unchanged but the top of the stack changes. Insertion in a stack is called **Push** and deletion of elements from the stack is known as **Pop.**

We can implement a stack using 2 ways:

* **Static implementation (using arrays)**
* **Dynamic implementation (using pointers)**

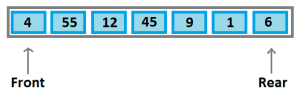
**Stack**

**Queues :**

Queues are also non-primitive linear data structure. But unlike stacks, queues are the First In First Out (FIFO) type of data structures. We can insert an element in a queue from the REAR end but we have to remove an element from the only FRONT end.

We can also implement queues using 2 ways :

* **Using arrays**
* **Using pointers**

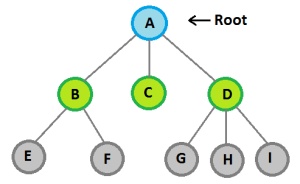
Queue

**Trees :**

Trees fall into the category of non-primitive non-linear data structures in the classification of data structure. They contain a finite set of data items referred as nodes. We can represent a hierarchical relationship between the data elements using trees.

A Tree has the following characteristics :

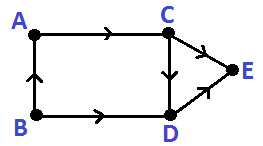
* The top item in a hierarchy of a tree is referred as the root of the tree.
* The remaining data elements are partitioned into a number of mutually exclusive subsets and they itself a tree and are known as the subtree.
* Unlike natural trees, trees in the data structure always grow in length towards the bottom.

Trees

**Graph :**

Graph falls in the non-primitive non-linear type of data structure in the classification of data structure. Graphs are capable of representing different types of physical structures. Apart from computer science, they are used broadly in the fields of Geography, Chemistry & Engineering Sciences.

A graph normally a combination of the set of vertices **V** and set of edges **E**.

Graph

The different types of Graphs are :

1. **Directed Graph**
2. **Non-directed Graph**
3. **Connected Graph**
4. **Non-connected Graph**
5. **Simple Graph**
6. **Multi-Graph**

**Stack:-**

Stacks are dynamic data structures that follow the **Last In First Out (LIFO)** principle. The last item to be inserted into a stack is the first one to be deleted from it.

For example, you have a stack of trays on a table. The tray at the top of the stack is the first item to be moved if you require a tray from that stack.

**Inserting and deleting elements**

Stacks have restrictions on the insertion and deletion of elements. Elements can be inserted or deleted only from one end of the stack i.e. from the top. The element at the top is called the top element. The operations of inserting and deleting elements are called push() and pop() respectively.

When the top element of a stack is deleted, if the stack remains non-empty, then the element just below the previous top element becomes the new top element of the stack.

For example, in the stack of trays, if you take the tray on the top and do not replace it, then the second tray automatically becomes the top element (tray) of that stack.

**Features of stacks**

* Dynamic data structures
* Do not have a fixed size
* Do not consume a fixed amount of memory
* Size of stack changes with each push() and pop() operation. Each push() and pop() operation increases and decreases the size of the stack by 1, respectively.

A stack can be visualized as follows:



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

**Basic Operation:-**

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

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

When data is PUSHed onto stack.

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

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

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

First we should learn about procedures to support stack functions −

**peek()**

Algorithm of peek() function −

begin procedure peek

return stack[top]

end procedure

Implementation of peek() function in C programming language −

**Example**

int peek() {

return stack[top];

}

**isfull()**

Algorithm of isfull() function −

begin procedure isfull

if top equals to MAXSIZE

return true

else

return false

endif

end procedure

Implementation of isfull() function in C programming language −

**Example**

bool isfull() {

if(top == MAXSIZE)

return true;

else

return false;

}

**isempty()**

Algorithm of isempty() function −

begin procedure isempty

if top less than 1

return true

else

return false

endif

end procedure

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

**Example**

bool isempty() {

if(top == -1)

return true;

else

return false;

}

**Push Operation**

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

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



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

Algorithm for PUSH Operation

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

begin procedure push: stack, data

if stack is full

return null

endif

top ← top + 1

stack[top] ← data

end procedure

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

**Example**

void push(int data) {

if(!isFull()) {

top = top + 1;

stack[top] = data;

} else {

printf("Could not insert data, Stack is full.\n");

}

}

**Queue:-**

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between [stacks](https://www.geeksforgeeks.org/stack-data-structure/)and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Simple Queue



**Queue Operations**

* **enqueue** - adds an item to the end of the queue
* **dequeue** - remove an item from front of the queue
* **initialize** - create an empty queue
* **isEmpty** - tests for whether or not queue is empty
* **isFull** - tests to see if queue is full (not needed if data structure grows automatically)
* **front** - looks at value of the first item but do not remove it

# Queue Implementations

This section will look at how to efficiently implement a queue using both an array and a linked list. Like a stack, a queue is also a special type of list. While a queue is a "line up" the ideas of front and back are not meant to be taken literally. The key is to understand that a Queue is a FIFO (first in first out) structure. That is the item to be removed is the oldest item in the list. as long as this is true, it doesn't matter where exactly the items get put into the queue or where it is removed.

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

## Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

## Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

* **enqueue()** − add (store) an item to the queue.
* **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

* **peek()** − Gets the element at the front of the queue without removing it.
* **isfull()** − Checks if the queue is full.
* **isempty()** − Checks if the queue is empty.

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

Let's first learn about supportive functions of a queue −

### **peek()**

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

**Algorithm**

begin procedure peek

return queue[front]

end procedure

Implementation of peek() function

**Example**

int peek() {

return queue[front];

}

### **isfull()**

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

**Algorithm**

begin procedure isfull

if rear equals to MAXSIZE

return true

else

return false

endif

end procedure

Implementation of isfull() function in C programming language −

**Example**

bool isfull() {

if(rear == MAXSIZE - 1)

return true;

else

return false;

}

### **isempty()**

Algorithm of isempty() function −

**Algorithm**

begin procedure isempty

if front is less than MIN OR front is greater than rear

return true

else

return false

endif

end procedure

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

**Example**

bool isempty() {

if(front < 0 || front > rear)

return true;

else

return false;

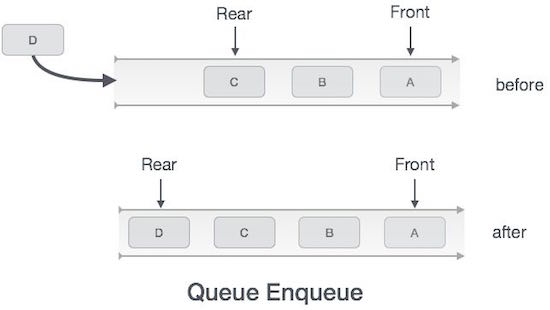
}

## **Enqueue Operation**

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

* **Step 1** − Check if the queue is full.
* **Step 2** − If the queue is full, produce overflow error and exit.
* **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
* **Step 4** − Add data element to the queue location, where the rear is pointing.
* **Step 5** − return success.



Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

### Algorithm for enqueue operation

procedure enqueue(data)

if queue is full

return overflow

endif

rear ← rear + 1

queue[rear] ← data

return true

end procedure

Implementation of enqueue() in C programming language −

**Example**

int enqueue(int data)

if(isfull())

return 0;

rear = rear + 1;

queue[rear] = data;

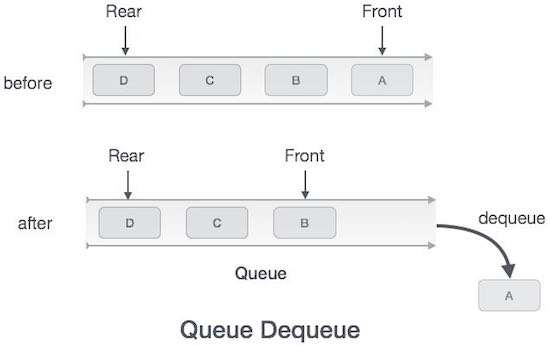
return 1;

end procedure

## **Dequeue Operation**

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

* **Step 1** − Check if the queue is empty.
* **Step 2** − If the queue is empty, produce underflow error and exit.
* **Step 3** − If the queue is not empty, access the data where **front** is pointing.
* **Step 4** − Increment **front** pointer to point to the next available data element.
* **Step 5** − Return success.



### Algorithm for dequeue operation

procedure dequeue

if queue is empty

return underflow

end if

data = queue[front]

front ← front + 1

return true

end procedure

Implementation of dequeue() in C programming language −

**Example**

int dequeue() {

if(isempty())

return 0;

int data = queue[front];

front = front + 1;

return data;

}

**Circular Queue:-**

Once the queue is full, even though few elements from the front are deleted and some occupied space is relieved, it is not possible to add anymore new elements, as the rear has already reached the Queue’s rear most position.

**Circular Queue**

* This queue is not linear but circular.
* Its structure can be like the following figure:

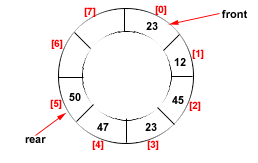
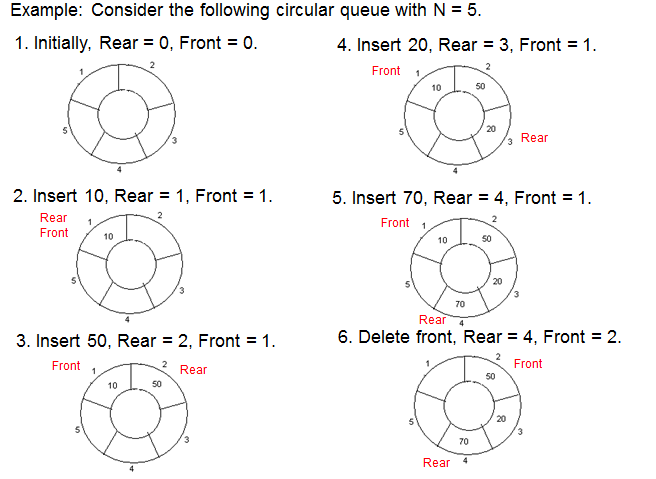
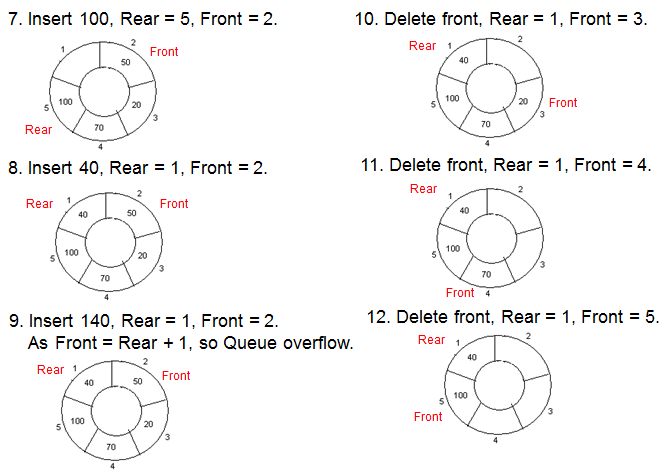


Figure: Circular Queue having

Rear = 5 and Front = 0

* In circular queue, once the Queue is full the "First" element of the Queue becomes the "Rear" most element, if and only if the "Front" has moved forward. otherwise it will again be a "Queue overflow" state.

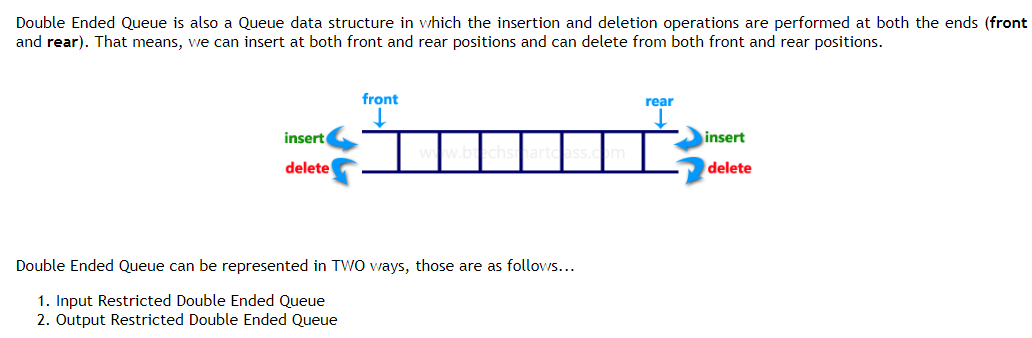


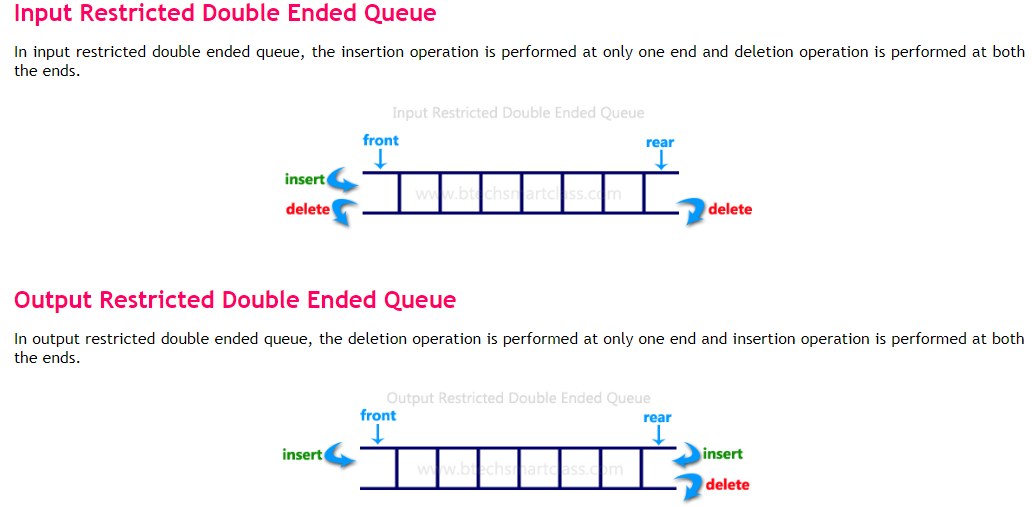


**Algorithms for Insert and Delete Operations in Circular Queue**

* **For Insert Operation**
* Insert-Circular-Q(CQueue, Rear, Front, N, Item)
* Here, CQueue is a circular queue where to store data. Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed. Here N is the maximum size of CQueue and finally, Item is the new item to be added. Initailly Rear = 0 and Front = 0.
* 1. If Front = 0 and Rear = 0 then Set Front := 1 and go to step 4.
* 2. If Front =1 and Rear = N or Front = Rear + 1
* then Print: “Circular Queue Overflow” and Return.
* 3. If Rear = N then Set Rear := 1 and go to step 5.
* 4. Set Rear := Rear + 1
* 5. Set CQueue [Rear] := Item.
* 6. Return
* **For Delete Operation**
* Delete-Circular-Q(CQueue, Front, Rear, Item)
* Here, CQueue is the place where data are stored. Rear represents the location in which the data element is to be inserted and Front represents the location from which the data element is to be removed. Front element is assigned to Item. Initially, Front = 1.
* 1. If Front = 0 then
* Print: “Circular Queue Underflow” and Return. /\*..Delete without Insertion
* 2. Set Item := CQueue [Front]
* 3. If Front = N then Set Front = 1 and Return.
* 4. If Front = Rear then Set Front = 0 and Rear = 0 and Return.
* 5. Set Front := Front + 1
* 6. Return.

**Double Ended Queue(Dqueue):-**





**DOUBLE ENDED QUEUE: (Deque)**

As we have been seen earlier queue is an ordered list of elements in which we can add the element only at one end called rear of queue and delete the element only at the other end called front of queue. but in deque, also called double ended queue, as the name implies we can add or delete the element from both sides. Deque can be of two types.

1. Input Restricted
2. Output Restricted

Input Restricted deque, element can added at only one end but we can deleted from both side.

Output Restricted deque, element can add both side but we can deleted from one-side.

**Figure:**

A Deque or double-ended queue is linear list in which insertion and deletion are made to or from either end of the structure. Such a structure can be represented by below figure.

Insertion

Deletion

Insertion

Deletion

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

**Operation of deque:**

This is the Q in which insertion & removal operation can be performed at either end of the Q. Consider the following empty deque and see how it represents as we performs insertion & removal operation.

STEP 1 Initial state of deque (i.e. empty queue F=0,R=0)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  | | F=0;R=0; |

STEP 2 Insert 10 into queue from right end

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 10 |  |  |  |  |  |  | | F=1;R=1; |

STEP 3 Insert 20 into queue from right end

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 10 | 20 |  |  |  |  |  | | F=1;R=2; |

STEP 4 Remove from left end. Removal element is 10

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | \* | 20 |  |  |  |  |  | | F=2;R=2; |

STEP 5 Insert 30 into queue from left end

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 30 | 20 |  |  |  |  |  | | F=1;R=2; |

STEP 6 Remove from right end, So Removed element is 20

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 30 | \*\* |  |  |  |  |  | | F=1;R=1; |

STEP 7 Insert 40 from right end.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 30 | 40 |  |  |  |  |  | | F=1;R=2; |

STEP 8 Insert 50 from left end: This is a special situation. There is no space to insert new element from left end (since front = 1).This situation must be declared as an overflow of a deque at the left end. So this insertion is not possible and deque is remained unchanged.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | 30 | 40 |  |  |  |  |  | |  |

STEP 9 Remove from left end

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | \*\* | 40 |  |  |  |  |  | | F=2;R=2; |

STEP 10 Remove from right end : removed element is 40 and after removal deque becomes empty because there is no elements, after removal and so front & rear must be zero (0)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | \*\* | \*\* |  |  |  |  |  | | F=0;R=0; |

Consider that after several insertion & removal at the both end (left & right), the value of the f & r are 6 & 7 and deque is as below:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  | 60 | 70 | | F=6;R=7;(max Q) |

STEP 11 Insert 88 from the left end.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  | 88 | 60 | 70 | | F=5;R=7; |

STEP 12 Insert 90 from the right end.: This is a special situation. Here R. = 7 (maxq). So Q is full at right end. We can't insert element from the right end and this is know as overflow of a deque at right end and so deque remains unchanged.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  | 88 | 60 | 70 | | F=5;R=7; |

**RDQInsert(Q,F,N,Y)**

Given F and R pointer to front and rear of double ended queue.respectively Q consisting of N element, Element Y this procedure want to insert at right side. Initially F and R set to 0.

STEP 1: [Queue is Overflow or not?]

IF R>=N then

Write ("Dqueue is full right side")

return;

STEP 2 : [Increment rear value]

R🡨R+1

STEP 3 : [Insert the value]

Q[R]🡨Y

STEP 4: [Is Front pointer Set]

IF F=0 THEN

F🡨1

STEP 4: RETURN

**LDQinsert (Q,F,,R,N,Y)**

Given F and R pointer to front and rear of double ended queue, respectively and a vector. Q consisting of N elements, Element Y this procedure want to insert at lefts side. Initially F and R set to 0

STEP 1: [Q UEUE IS FULL AT LEFT END SIDE ]

If F=1 then

Write("Queue is full at left side")

RETURN

STEP 2: [Is front zero ?]

if F=0 then

F 🡨 N

ELSE

F 🡨F-1

STEP 3: [Insert Element]

Q[F]🡨 Y

STEP 4: [Is rear zero?]

If R=0 then

R🡨N

STEP 6: [RETURN]

**R.DQREMOVE(Q,F,R)**

Given F and R pointer to front and rear of double ended queue,respectively and a vector Q consisting of N elements, this delete the right side last element's is a temporary variable.

STEP 1:[Is queue is empty at right side?]

IF R=0

WRITE(" QUEUE IS EMPTY AT RIGHT SIDE")

RETURN

STEP 2:[remove the element]

Y🡨Q[R]

STEP 3:[is dqueue is empty then initialize again?]

IF F==R THEN

F🡨0

R🡨0

Else

R🡨 R- 1

STEP 4:[Return]

**Procedure: LDQREMOVE (Q,F,R)**

Given F and R, pointers to the front and rear elements of queue.

The queue Q to which they correspond, this function deleted and returns the last element of queue.Y is a temporary variable

**Steps: =**

1. [Check underflow condition]

If F = 0

WRITE( “QUEUE IS EMPTY AT RIGHT SIDE”)

return

1. [Remove an element]

Y🡨 Q [F]

1. [Check for empty queue]

If F = R

F🡨 R🡨 0

Else

F🡨F +1

1. Return (Y)

**Difference B/W Stack and Queue**

|  |  |  |
| --- | --- | --- |
|  | **Stack** | **Queue** |
| **1 1** | Exhibits only FIFO type Algorithm | E Exhibits a FIFO type Algorithm. |
| **2 2** | S Since insertion and deletion occurs at the same end stack cam be maintained with the help of one pointer only. | Since insertion and deletion occurs at two different ends Queue requires at least two pointers for its execution. |
| **3 3** | Modification of Stack to perform the task of queue is not possible. | Q Queue can be modified as Double Ended Queue that exhibits the properties of both Stack and Queue. |
| **4 4** | An Example of Stack is  the railway system for shunting cars. | An Example of Queue is a line of cars waiting to proceed inside Drive-in |
| **5 5** | Memory representation of stack is as under:  10  20  30  40    Top | Memory representation of queue is as under:  10  20  30  40    Front Rear |

**Difference B/W Simple Queue and Circular Queue**

|  |  |  |
| --- | --- | --- |
|  | **Simple Queue** | **Circular Queue** |
| **1 1** | When Queue is full insertion of more elements is not possible even after deletion of elements until and unless all elements have been deleted and queue has been made empty. | Once Queue is full, insertion of more elements is possible after deleting one or more elements without completely emptying the Queue. |
| **2 2** | Memory representation of Simple Queue is as under:  10  20  30  40  Front Rear | Memory representation of Circular Queue is as under:  CQ |

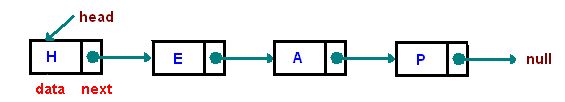
**Difference B/W Simple Queue and Double Queue**

|  |  |  |
| --- | --- | --- |
|  | **Simple Queue** | **Double Ended Queue** |
| **1 1** | Exhibits only FIFO type Algorithm. | Exhibits both FIFO as well as LIFO type Algorithm |
| **2 2** | I Insertion occurs at one end and deletion at the other. | I Insertion and Deletion can occur at either of the ends. |
| **3 3** | S Simple and Easy to maintain. | Complex algorithm hence difficult to maintain |
| **4 4** | Memory representation of Simple Queue is as under:  10  20  30  40  Front Rear | Memory representation of Double Ended Queue is as under:  10  20  30  40  Insert Insert  Delete Delete |

**Link List:-**

A linked list is a data structure that is basically a chain of nodes in which each node is connected to each other by means of pointers or references. It is dynamic in nature; that means that the size of the linked list can vary depending on the requirements of the users. It is a collection of similar kinds of items. The basic building block of linked list is a node.

**Node**Node is a basic building block of linked list that is comprised of two parts, a data and a next pointer. The data part basically contains the data whereas the next pointer contains the reference to the next node. The last node in the linked list points to null. The first node in the linked list is called the Head node and last node is called the Tail node.

  
  
A linked list must provide the following operations:

1. Add nodes
2. Remove nodes
3. Iterate among nodes
4. Find nodes

**Node Chaining**For creating a linked list we must understand what a node is and how node chaining is done. Node is a basic building block of a linked list that must consist of two things, one is data and the other is a pointer to the next node.

**Advantages of Linked Lists**

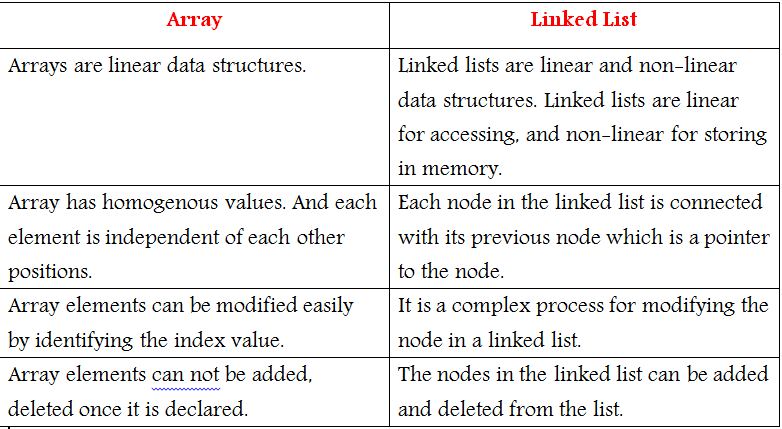
* They are a dynamic in nature which allocates the memory when required.
* Insertion and deletion operations can be easily implemented.
* Stacks and queues can be easily executed.
* Linked List reduces the access time.

**Disadvantages of Linked Lists**

* The memory is wasted as pointers require extra memory for storage.
* No element can be accessed randomly; it has to access each node sequentially.
* Reverse Traversing is difficult in linked list.

**Applications of Linked Lists**

* Linked lists are used to implement stacks, queues, graphs, etc.
* Linked lists let you insert elements at the beginning and end of the list.
* In Linked Lists we don't need to know the size in advance.



**Types of Linked Lists**

There are 3 different implementations of Linked List available, they are:

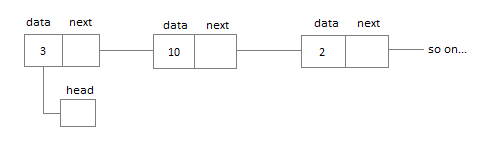
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Let's know more about them and how they are different from each other.

**Singly Linked List**

Singly linked lists contain nodes which have a **data** part as well as an **address part** i.e. next, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are **insertion**, **deletion** and **traversal**.



**Node Creation:-**

struct node

{

**int** data;

    struct node \*next;

};

struct node \*head, \*ptr;

ptr = (struct node \*)malloc(sizeof(struct node \*));

**Ex:-**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

//display the list

void printList() {

struct node \*ptr = head;

printf("\n[head] =>");

//start from the beginning

while(ptr != NULL) {

printf(" %d =>",ptr->data);

ptr = ptr->next;

}

printf(" [null]\n");}

//insert link at the first location

void insert(int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

//link->key = key;

link->data = data;

//point it to old first node

link->next = head;

//point first to new first node

head = link;

}

int main() {

insert(10);

insert(20);

insert(30);

insert(1);

insert(40);

insert(56);

printList();

return 0;

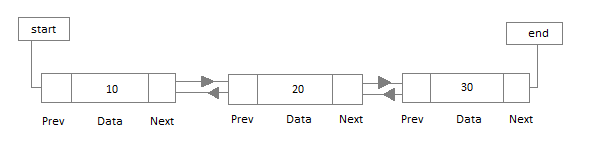
}

**Output:-**

[head] => 56 => 40 => 1 => 30 => 20 => 10 => [null]

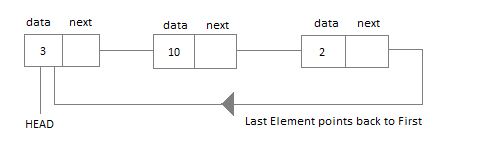
**Doubly Linked List**

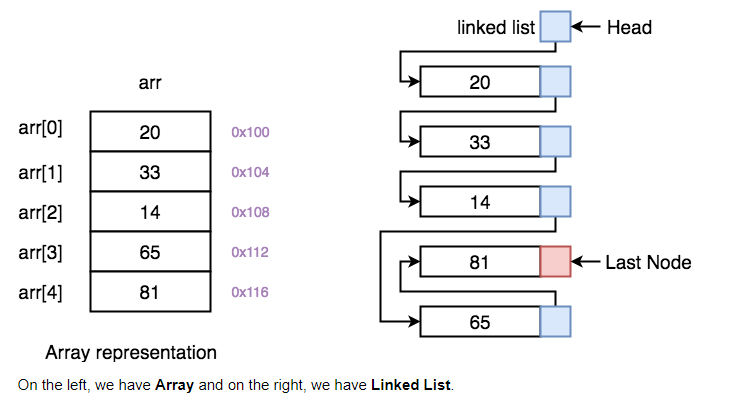
In a doubly linked list, each node contains a **data** part and two addresses, one for the **previous** node and one for the **next** node.



**Circular Linked List**

In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.





## Ex:-

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

struct node {

int data;

int key;

struct node \*next;

struct node \*prev;

};

//this link always point to first Link

struct node \*head = NULL;

//this link always point to last Link

struct node \*last = NULL;

struct node \*current = NULL;

//is list empty

Bool isEmpty() {

return head == NULL;

}

int length() {

int length = 0;

struct node \*current;

for(current = head; current != NULL; current = current->next){

length++;

}

return length;

}

//display the list in from first to last

void displayForward() {

//start from the beginning

struct node \*ptr = head;

//navigate till the end of the list

printf("\n[ ");

while(ptr != NULL) {

printf("(%d,%d) ",ptr->key,ptr->data);

ptr = ptr->next;

}

printf(" ]");}

//display the list from last to first

void displayBackward() {

//start from the last

struct node \*ptr = last;

//navigate till the start of the list

printf("\n[ ");

while(ptr != NULL) {

//print data

printf("(%d,%d) ",ptr->key,ptr->data);

//move to next item

ptr = ptr ->prev;

}

}

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

if(isEmpty()) {

//make it the last link

last = link;

} else {

//update first prev link

head->prev = link;

}

//point it to old first link

link->next = head;

//point first to new first link

head = link;

}

//insert link at the last location

void insertLast(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

if(isEmpty()) {

//make it the last link

last = link;

} else {

//make link a new last link

last->next = link;

//mark old last node as prev of new link

link->prev = last;

}

//point last to new last node

last = link;

}

//delete first item

struct node\* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

//if only one link

if(head->next == NULL){

last = NULL;

} else {

head->next->prev = NULL;

}

head = head->next;

//return the deleted link

return tempLink;

}

//delete link at the last location

struct node\* deleteLast() {

//save reference to last link

struct node \*tempLink = last;

//if only one link

if(head->next == NULL) {

head = NULL;

} else {

last->prev->next = NULL;

}

last = last->prev;

//return the deleted link

return tempLink;}

//delete a link with given key

struct node\* delete(int key) {

//start from the first link

struct node\* current = head;

struct node\* previous = NULL;

//if list is empty

if(head == NULL) {

return NULL;

}

//navigate through list

while(current->key != key) {

//if it is last node

if(current->next == NULL) {

return NULL;

} else {

//store reference to current link

previous = current;

//move to next link

current = current->next;

}

}

//found a match, update the link

if(current == head) {

//change first to point to next link

head = head->next;

} else {

//bypass the current link

current->prev->next = current->next;

}

if(current == last) {

//change last to point to prev link

last = current->prev;

} else {

current->next->prev = current->prev;

}

return current;

}

bool insertAfter(int key, int newKey, int data) {

//start from the first link

struct node \*current = head;

//if list is empty

if(head == NULL) {

return false;

}

//navigate through list

while(current->key != key) {

//if it is last node

if(current->next == NULL) {

return false;

} else {

//move to next link

current = current->next;

}

}

//create a link

struct node \*newLink = (struct node\*) malloc(sizeof(struct node));

newLink->key = newKey;

newLink->data = data;

if(current == last) {

newLink->next = NULL;

last = newLink;

} else {

newLink->next = current->next;

current->next->prev = newLink;

}

newLink->prev = current;

current->next = newLink;

return true; }

void main() {

insertFirst(1,10);

insertFirst(2,20);

insertFirst(3,30);

insertFirst(4,1);

insertFirst(5,40);

insertFirst(6,56);

printf("\nList (First to Last): ");

displayForward();

printf("\n");

printf("\nList (Last to first): ");

displayBackward();

printf("\nList , after deleting first record: ");

deleteFirst();

displayForward();

printf("\nList , after deleting last record: ");

deleteLast();

displayForward();

printf("\nList , insert after key(4) : ");

insertAfter(4,7, 13);

displayForward();

printf("\nList , after delete key(4) : ");

delete(4);

displayForward();

}

## Output:-

List (First to Last):

[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]

List (Last to first):

[ (1,10) (2,20) (3,30) (4,1) (5,40) (6,56) ]

List , after deleting first record:

[ (5,40) (4,1) (3,30) (2,20) (1,10) ]

List , after deleting last record:

[ (5,40) (4,1) (3,30) (2,20) ]

List , insert after key(4) :

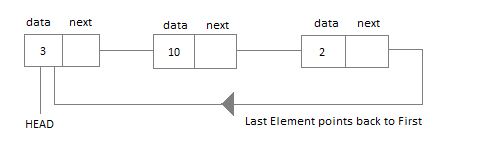
[ (5,40) (4,1) (7,13) (3,30) (2,20) ]

List , after delete key(4) :

[ (5,40) (4,13) (3,30) (2,20) ]

## Circular Linked List

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



#### Application of Circular Linked List

* The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
* Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
* Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

**Ex:-**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

struct node {

int data;

int key;

struct node \*next;

};

struct node \*head = NULL;

struct node \*current = NULL;

bool isEmpty() {

return head == NULL;

}

int length() {

int length = 0;

//if list is empty

if(head == NULL) {

return 0;

}

current = head->next;

while(current != head) {

length++;

current = current->next;

}

return length;}

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

if (isEmpty()) {

head = link;

head->next = head;

} else {

//point it to old first node

link->next = head;

//point first to new first node

head = link;

} }

//delete first

itemstruct node \* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

if(head->next == head) {

head = NULL;

return tempLink;

}

//mark next to first link as first

head = head->next;

//return the deleted link

return tempLink;

}

//display the list

void printList() {

struct node \*ptr = head;

printf("\n[ ");

//start from the beginning

if(head != NULL) {

while(ptr->next != ptr) {

printf("(%d,%d) ",ptr->key,ptr->data);

ptr = ptr->next;

}

}

printf("[ ]");}

void main() {

insertFirst(1,10);

insertFirst(2,20);

insertFirst(3,30);

insertFirst(4,1);

insertFirst(5,40);

insertFirst(6,56);

printf("Original List: ");

//print list

printList();

while(!isEmpty()) {

struct node \*temp = deleteFirst();

printf("\nDeleted value:");

printf("(%d,%d) ",temp->key,temp->data);

}

printf("\nList after deleting all items: ");

printList();

}

**Output:-**

Original List:

[ (6,56) (5,40) (4,1) (3,30) (2,20) ]

Deleted value:(6,56)

Deleted value:(5,40)

Deleted value:(4,1)

Deleted value:(3,30)

Deleted value:(2,20)

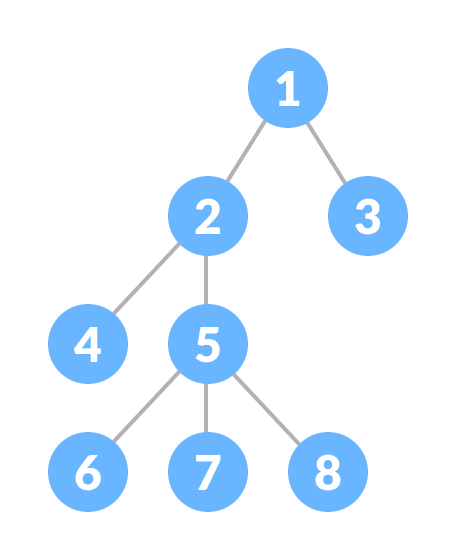
Deleted value:(1,10)

List after deleting all items:

[ ]

**Tree:-**

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



A Tree

## Why Tree Data Structure?

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

## Tree Terminologies

### **Node**

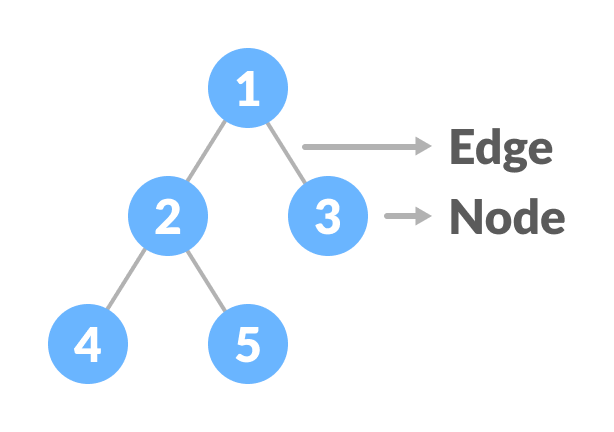
A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called ****leaf nodes or external nodes**** that do not contain a link/pointer to child nodes.

The node having at least a child node is called an ****internal node****.

### **Edge**

It is the link between any two nodes.



Nodes and edges of a tree

### **Root**

It is the topmost node of a tree.

### **Height of a Node**

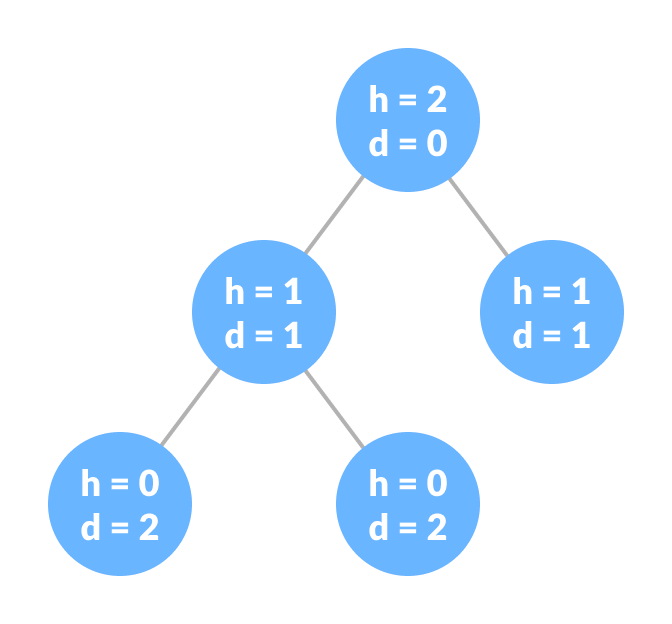
The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

### **Depth of a Node**

The depth of a node is the number of edges from the root to the node.

### **Height of a Tree**

The height of a Tree is the height of the root node or the depth of the deepest node.



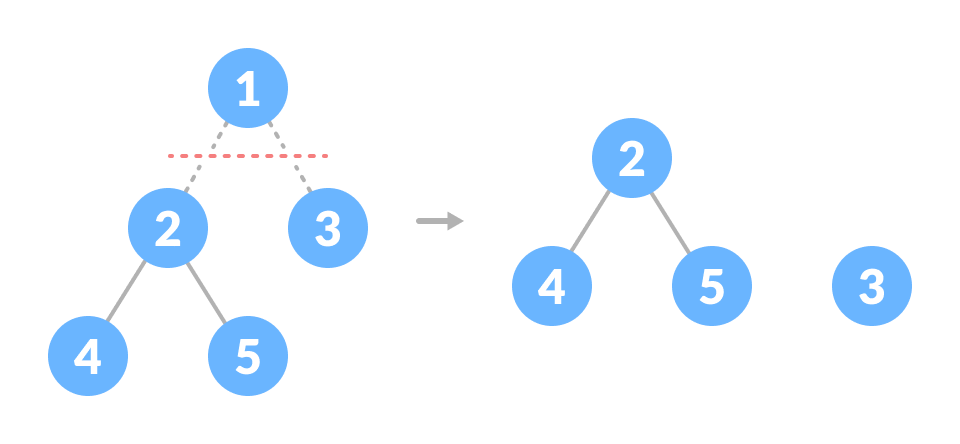
Height and depth of each node in a tree

### **Degree of a Node**

The degree of a node is the total number of branches of that node.

### **Forest**

A collection of disjoint trees is called a forest.



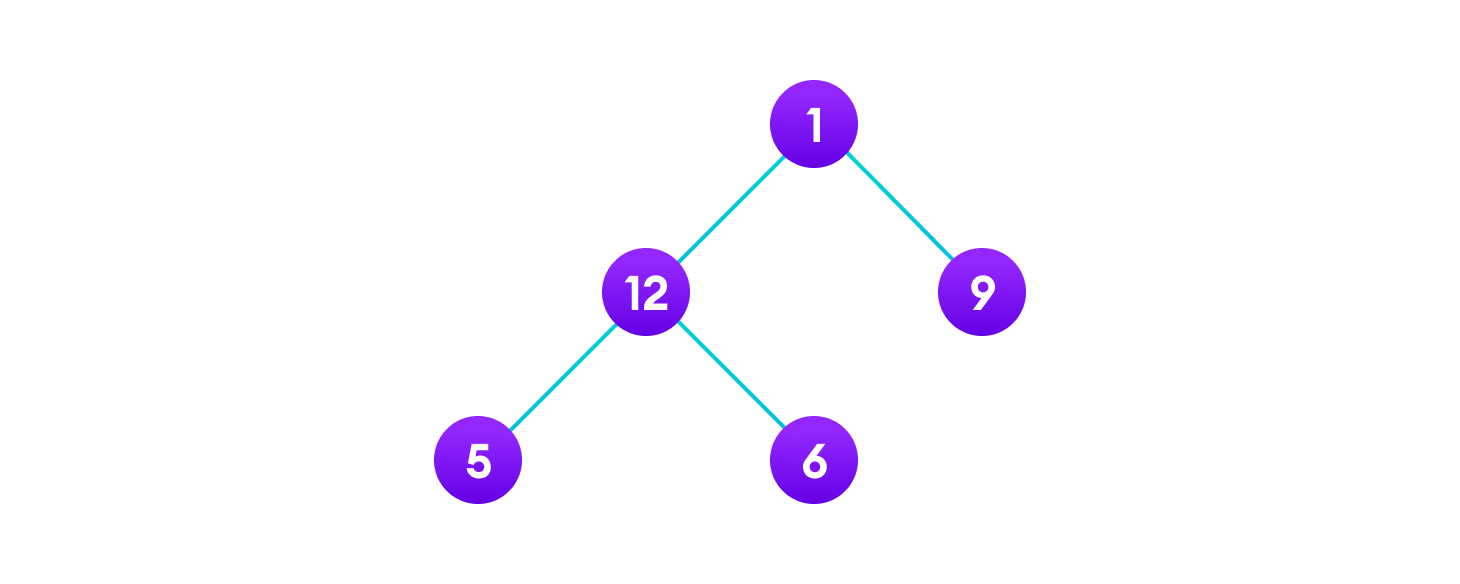
Creating forest from a tree

You can create a forest by cutting the root of a tree.

# Tree Traversal - inorder, preorder and postorder:-

Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.

Linear data structures like arrays, [stacks](https://www.programiz.com/data-structures/stack), [queues](https://www.programiz.com/data-structures/queue), and [linked list](https://www.programiz.com/data-structures/linked-list) have only one way to read the data. But a hierarchical data structure like a [tree](https://www.programiz.com/data-structures/trees) can be traversed in different ways.



Tree traversal

 There are three ways which we use to traverse a tree −

* In-order Traversal
* Pre-order Traversal
* Post-order Traversal

## In-order Traversal:-

## In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



***D → B → E → A → F → C → G***

**Algorithm:-**

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree

inorder(root->left)

display(root->data)

inorder(root->right)

## Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



***A → B → D → E → C → F → G***

**Algorithm:-**

1. Visit root node
2. Visit all the nodes in the left subtree
3. Visit all the nodes in the right subtree

display(root->data)

preorder(root->left)

preorder(root->right)

## Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



***D → E → B → F → G → C → A***

**Algorithm:-**

1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node

postorder(root->left)

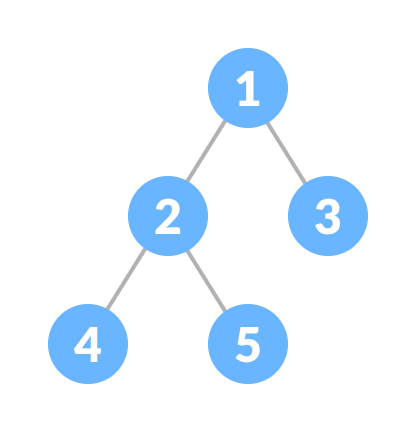
postorder(root->right)

display(root->data)

**Binary Tree:-**

A binary tree is a tree data structure in which each parent node can have at most two children.

For example: In the image below, each element has at most two children.



Binary Tree

## Types of Binary Tree

### **Full Binary Tree**

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



Full Binary Tree

### **Perfect Binary Tree**

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

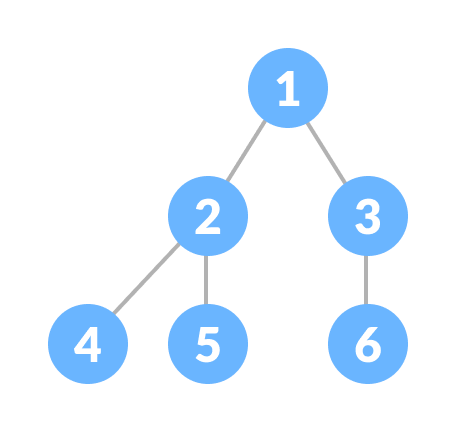


Perfect Binary Tree

### **Complete Binary Tree**

A complete binary tree is just like a full binary tree, but with two major differences

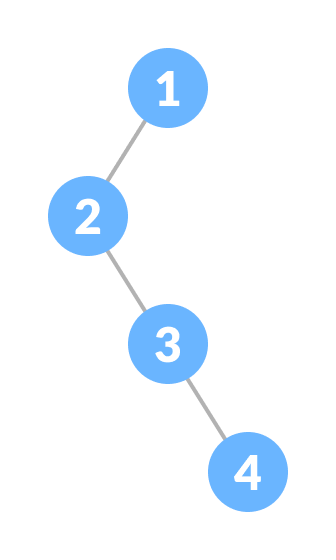
1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



Complete Binary Tree

### **Degenerate or Pathological Tree**

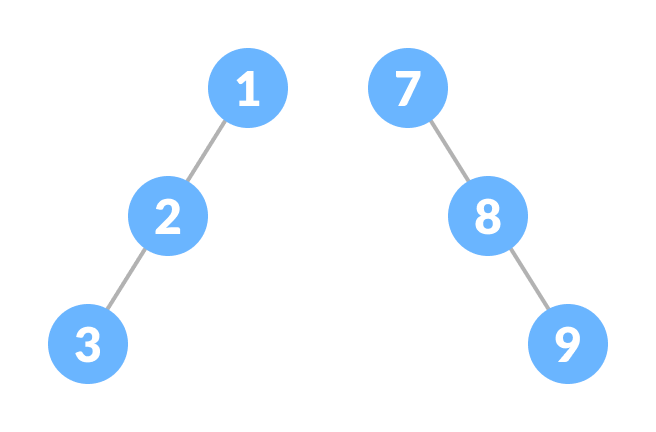
A degenerate or pathological tree is the tree having a single child either left or right.



Degenerate Binary Tree

### **Skewed Binary Tree**

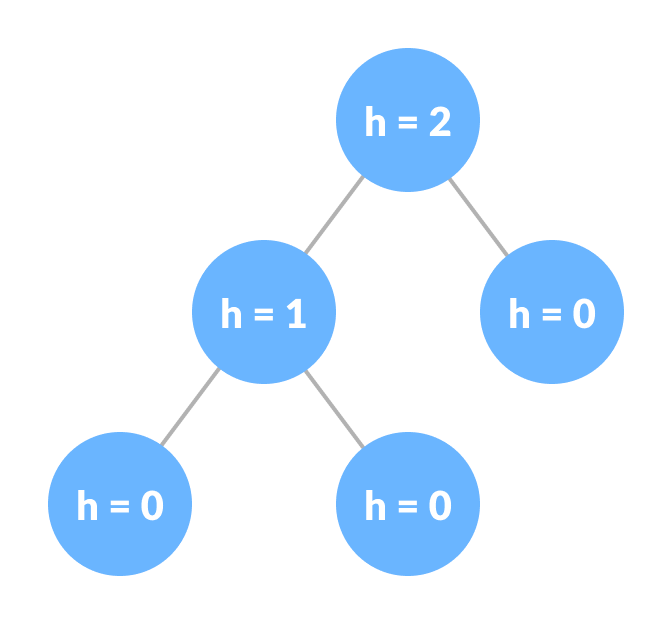
A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: ****left-skewed binary tree**** and ****right-skewed binary tree****.



Skewed Binary Tree

### **Balanced Binary Tree**

It is a type of binary tree in which the difference between the left and the right subtree for each node is either 0 or 1.



Balanced Binary Tree

## Binary Tree Representation

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

struct node

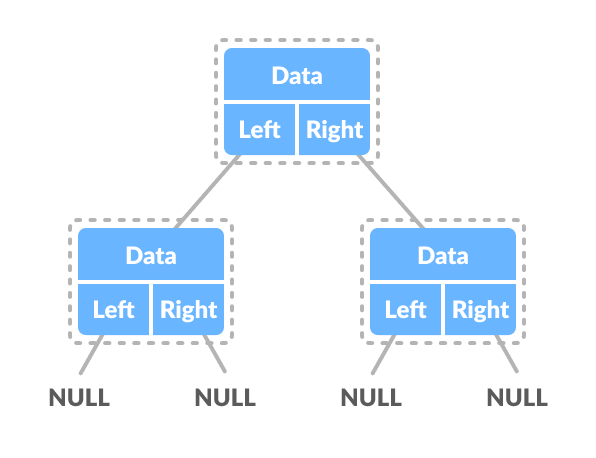
{

int data;

struct node \*left;

struct node \*right;

};



Binary Tree Representation

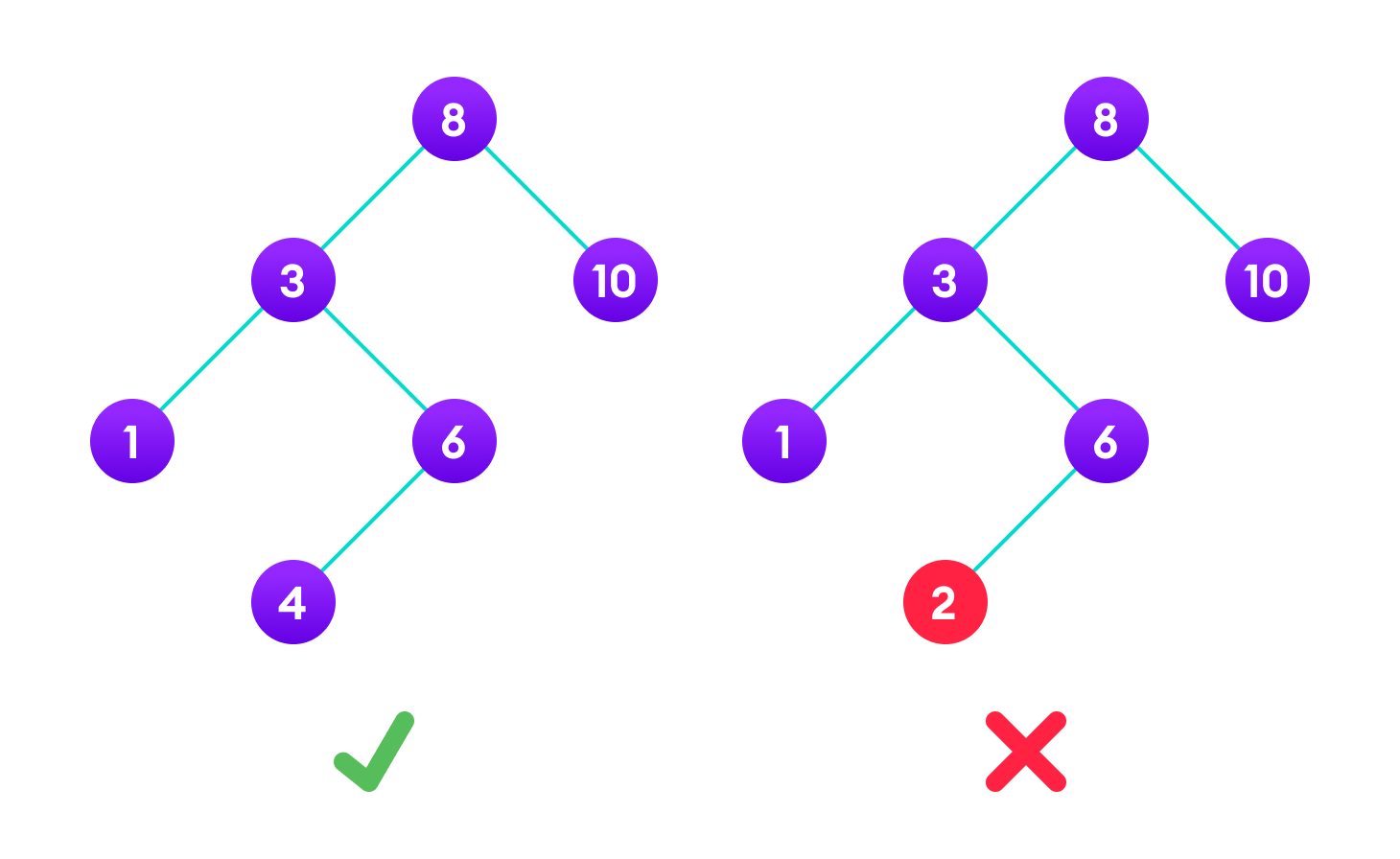
**Binary Search Tree(BST):-**

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

* It is called a binary tree because each tree node has a maximum of two children.
* It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular [binary tree](https://www.programiz.com/data-structures/trees) is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties

A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree.

The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

There are two basic operations that you can perform on a binary search tree:

## Search Operation

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

****Algorithm:****

If root == NULL

return NULL;

If number == root->data

return root->data;

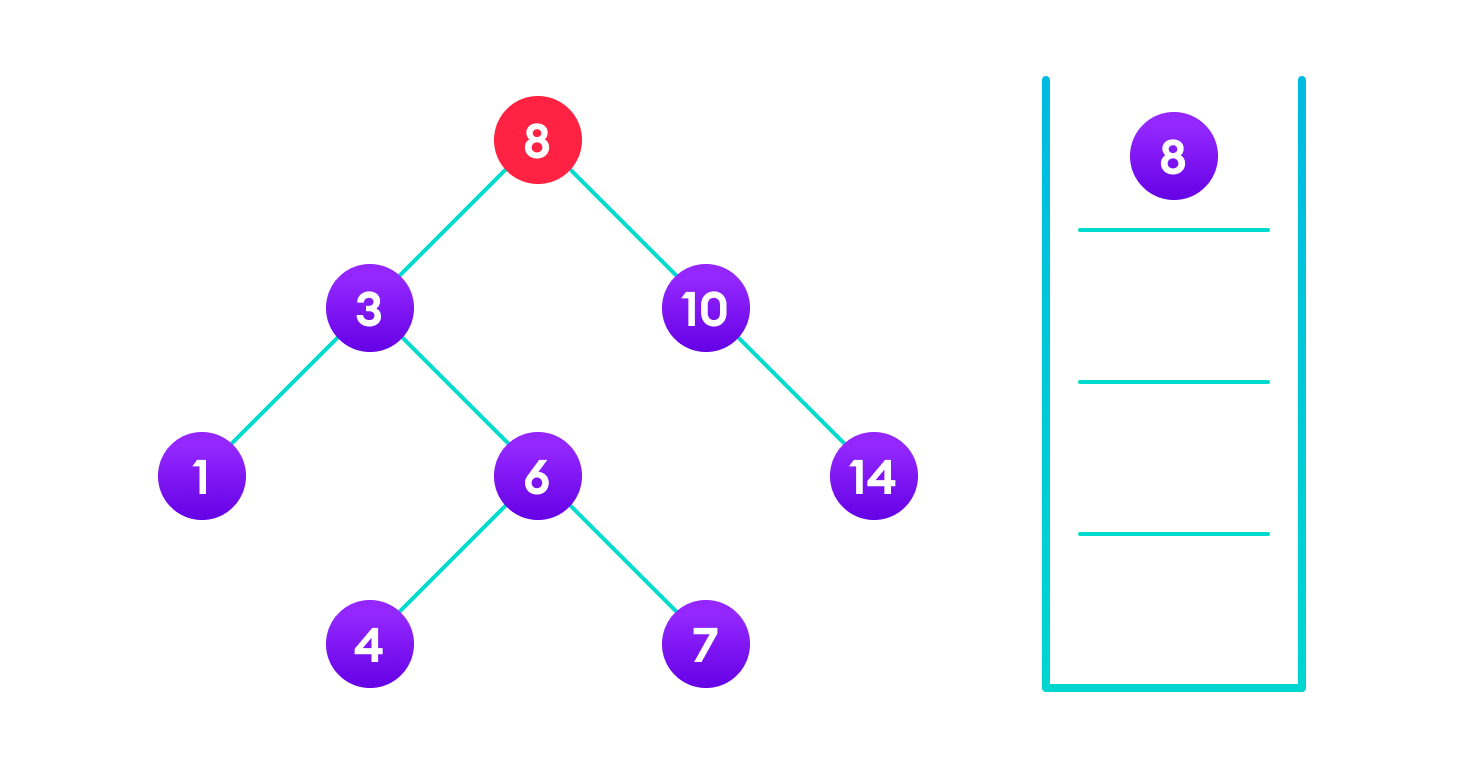
If number < root->data

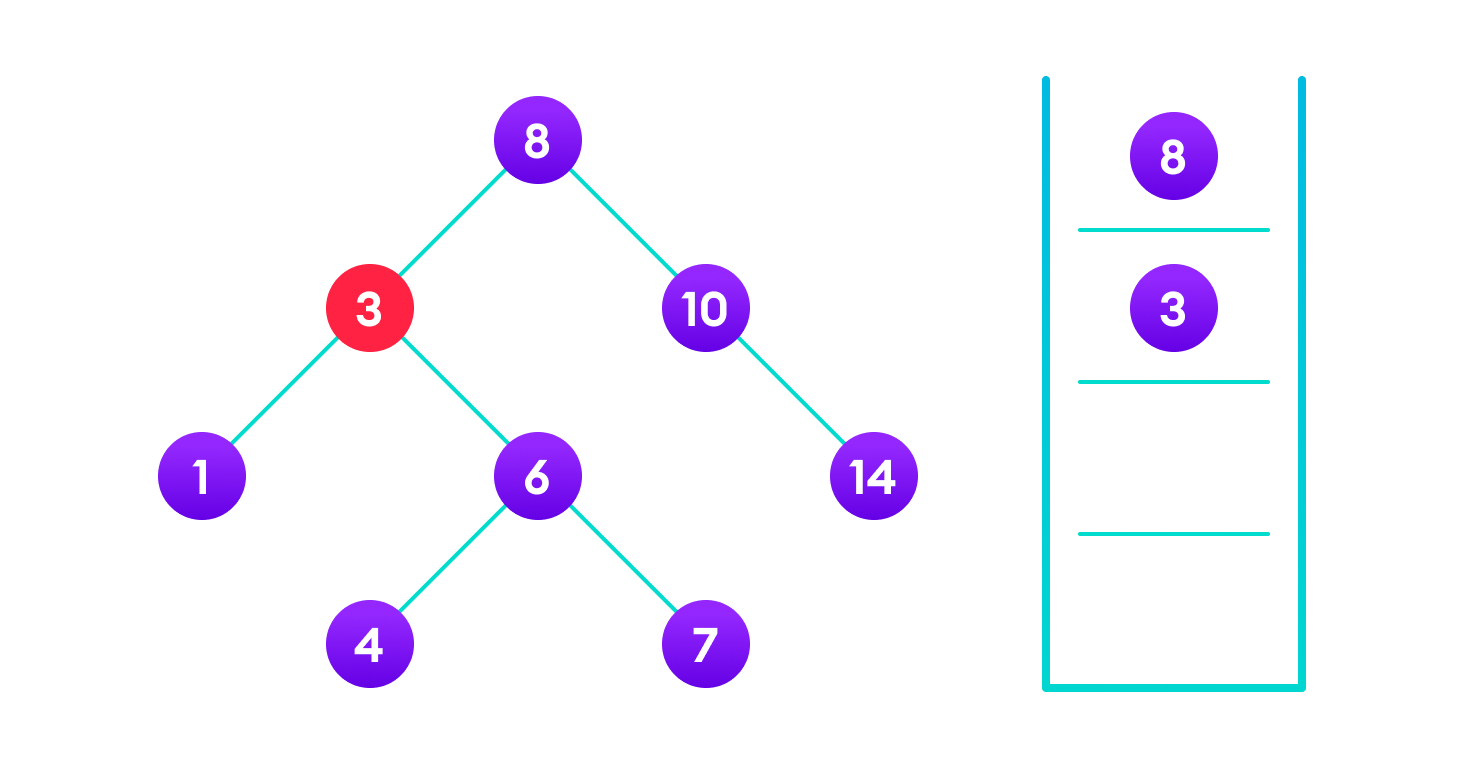
return search(root->left)

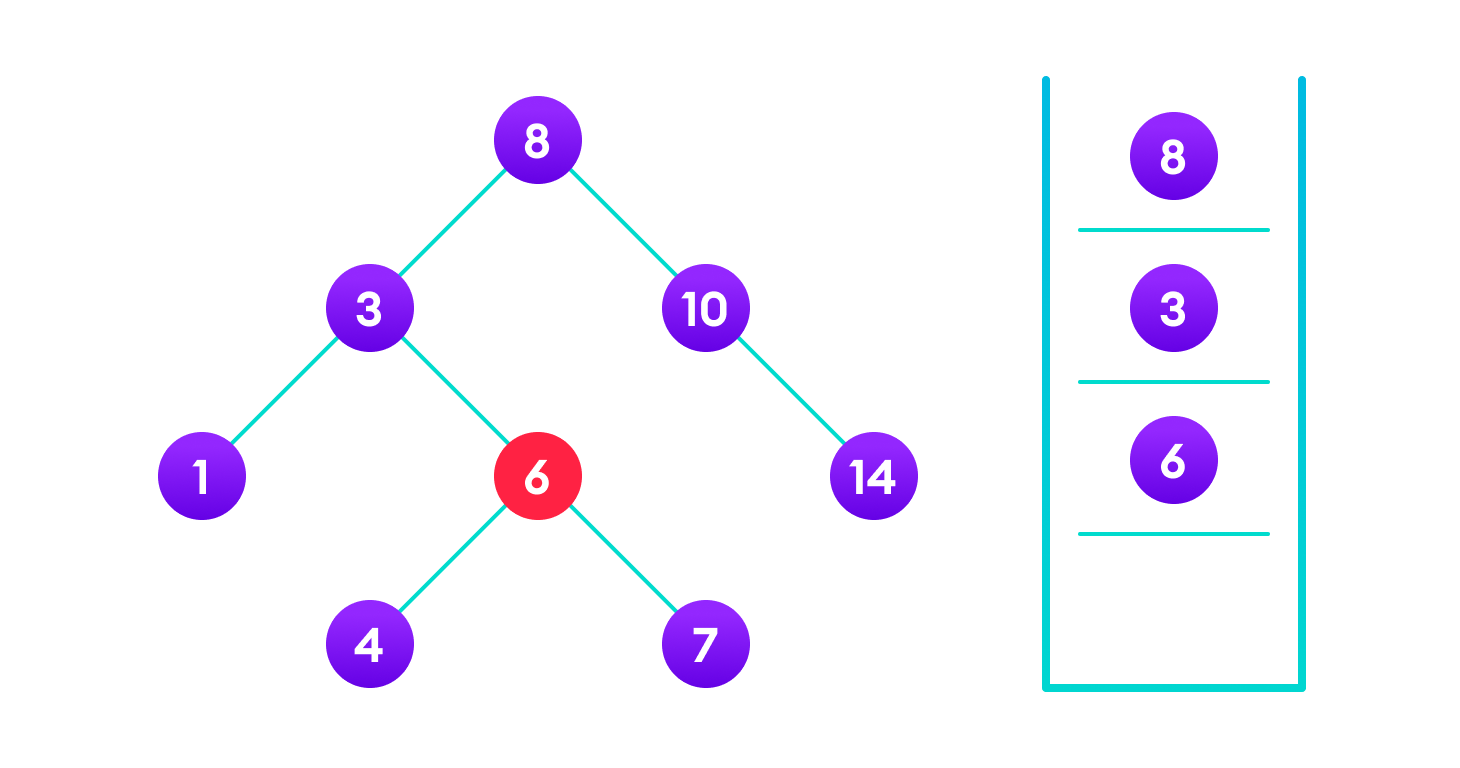
If number > root->data

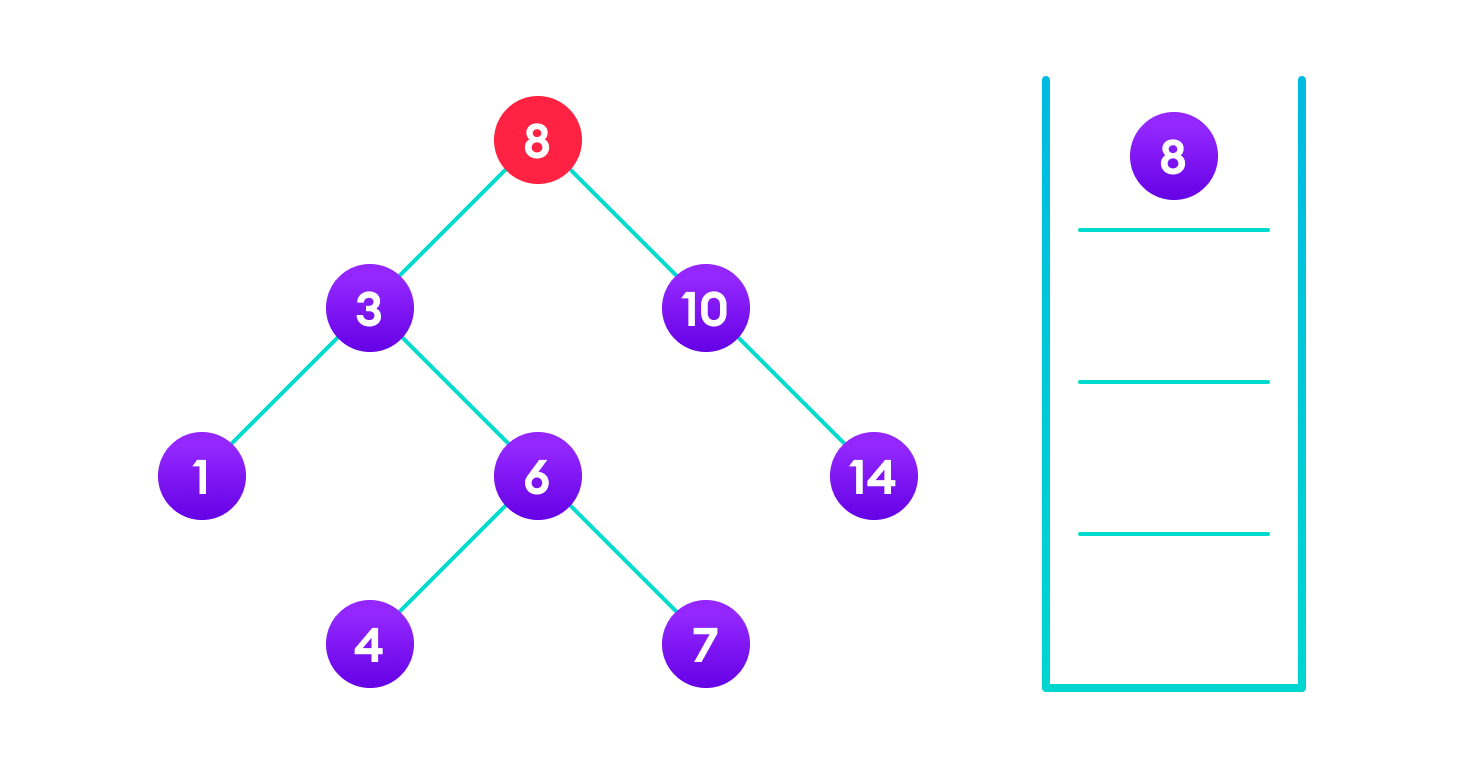
return search(root->right)

Let us try to visualize this with a diagram.

**4 is not found so, traverse through the left subtree of 8**

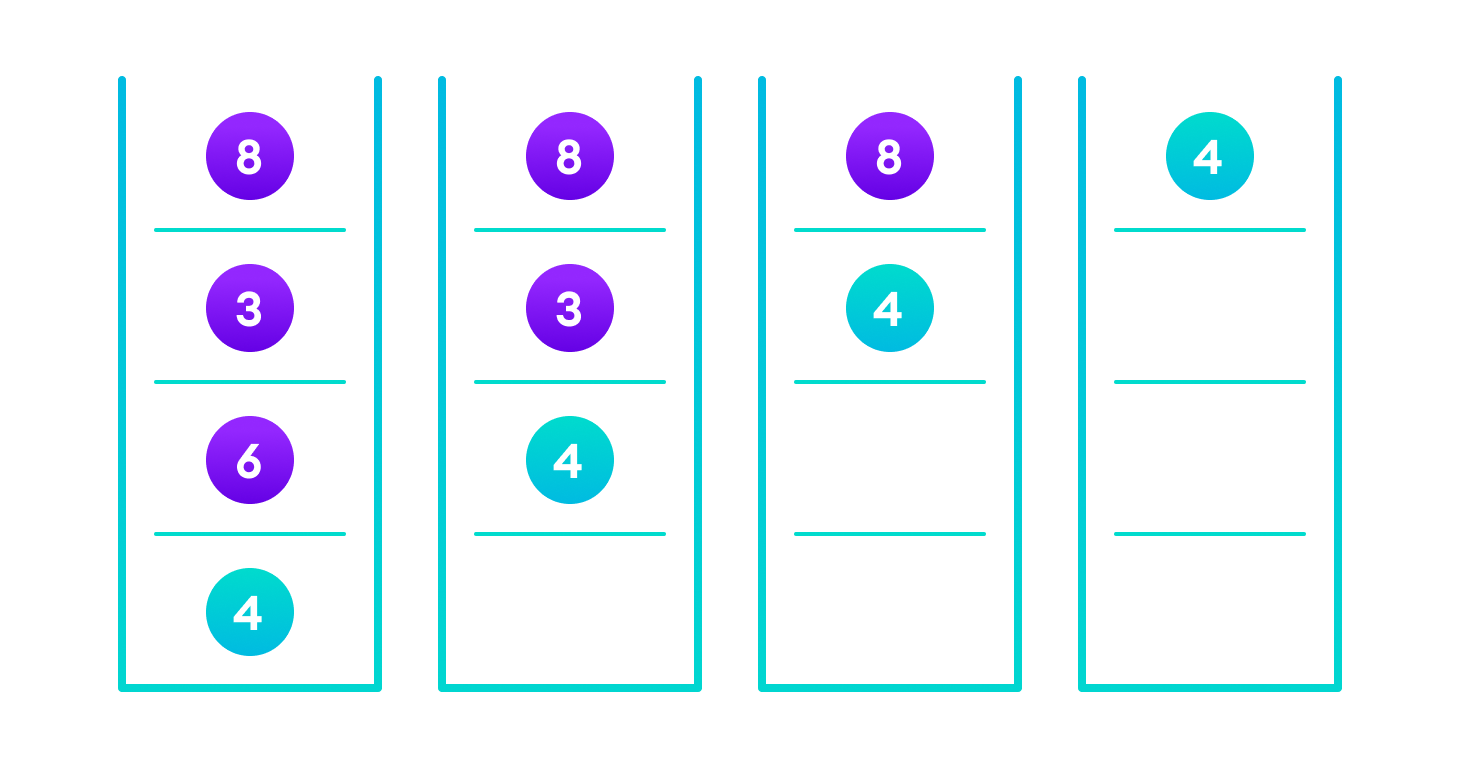
**4 is not found so, traverse through the right subtree of 3**

**4 is not found so, traverse through the left subtree of 6**

**4 is found**

If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.

If you might have noticed, we have called return search(struct node\*) four times. When we return either the new node or NULL, the value gets returned again and again until search(root) returns the final result.

If the value is found in any of the subtrees, it is propagated up so that in the end it is returned, otherwise null is returned

If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

## Insert Operation

Inserting a value in the correct position is similar to searching because we try to maintain the rule that the **left subtree is lesser than root and the right subtree is larger than root.**

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

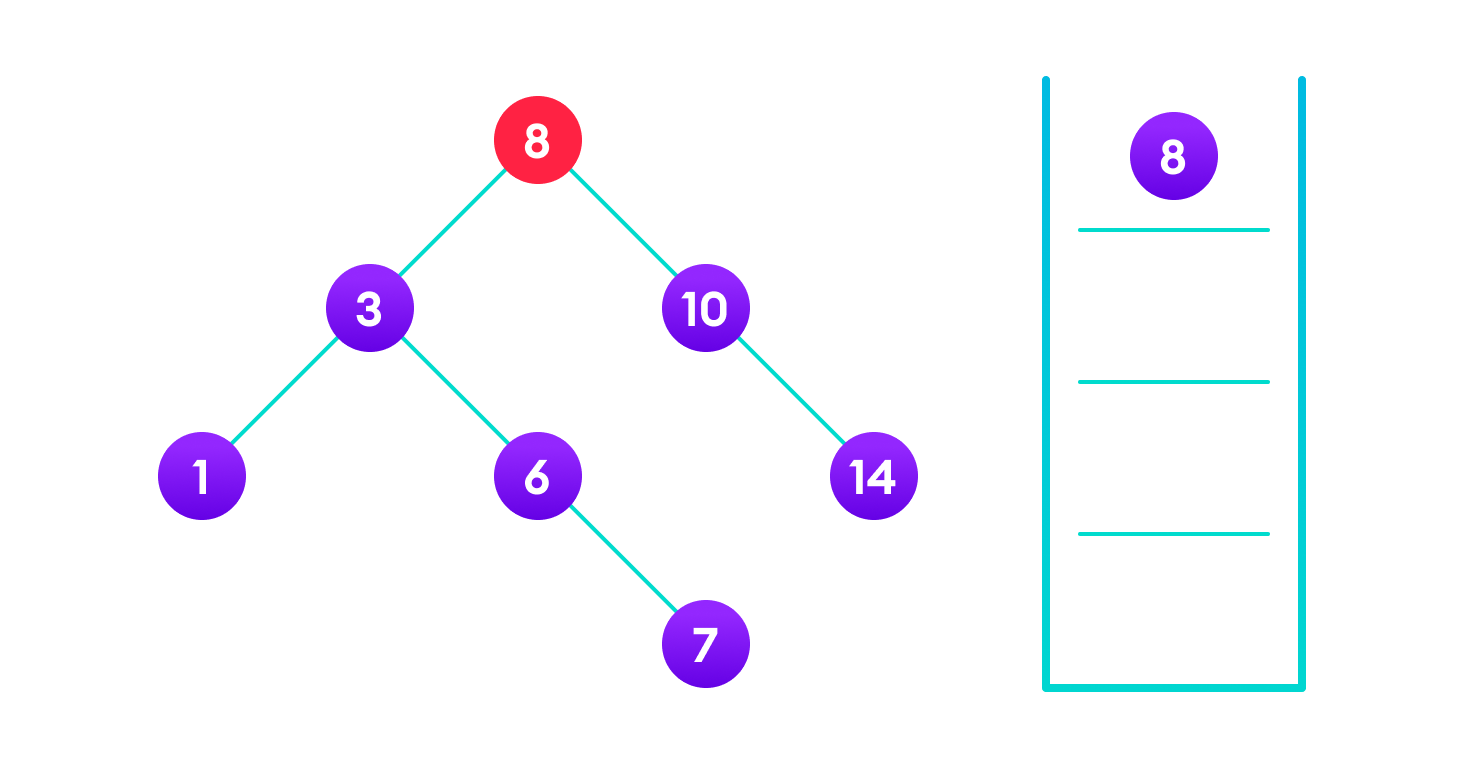
****Algorithm:****

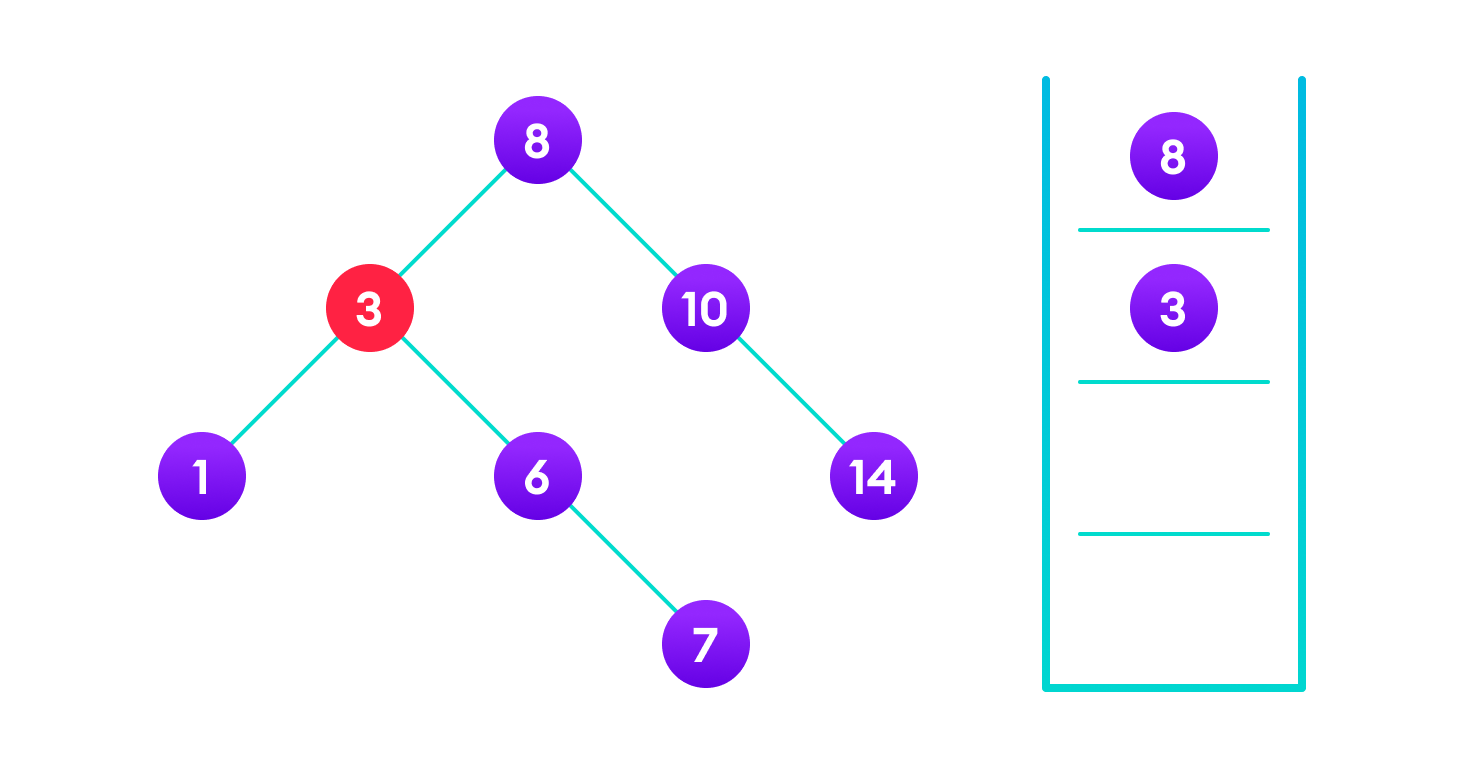
If node == NULL

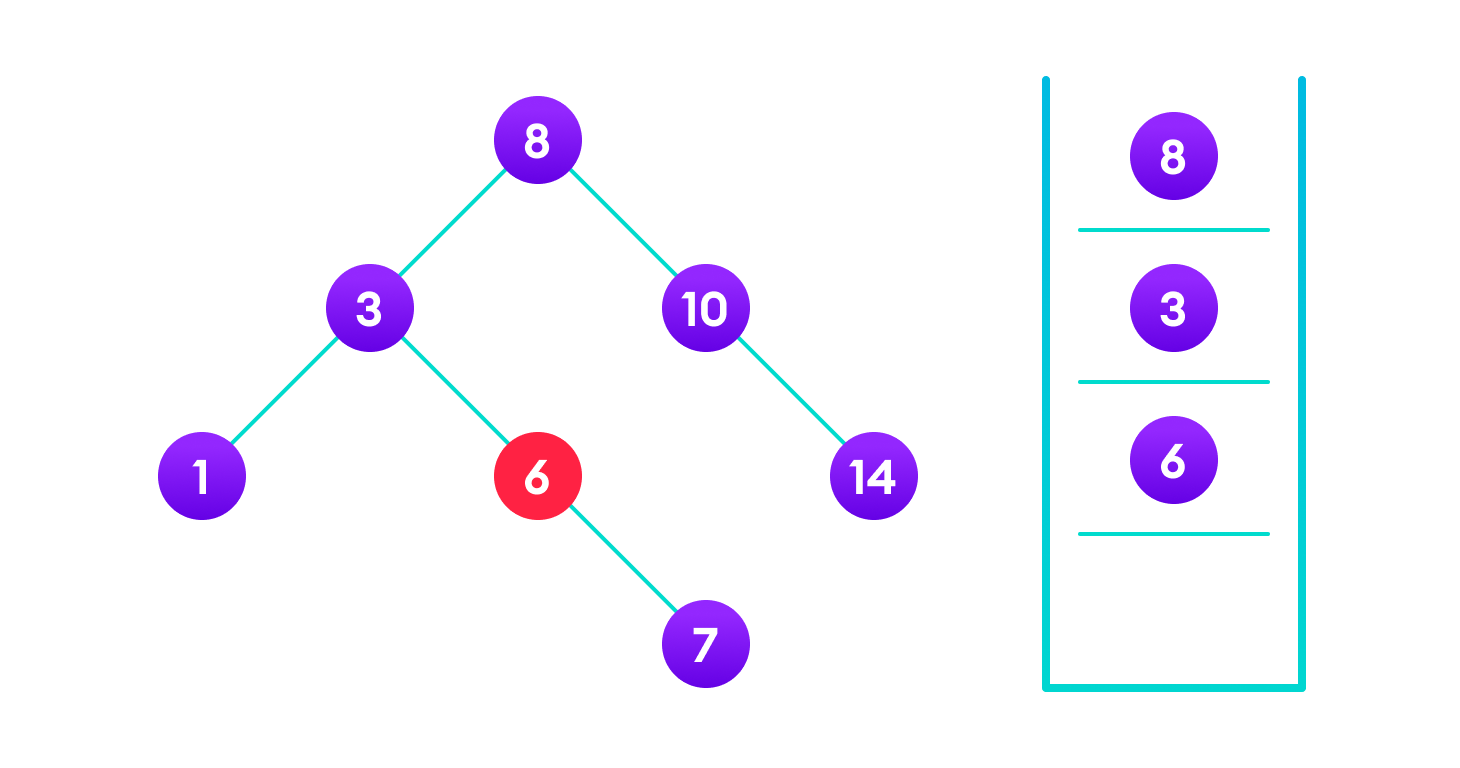
return createNode(data)if (data < node->data)

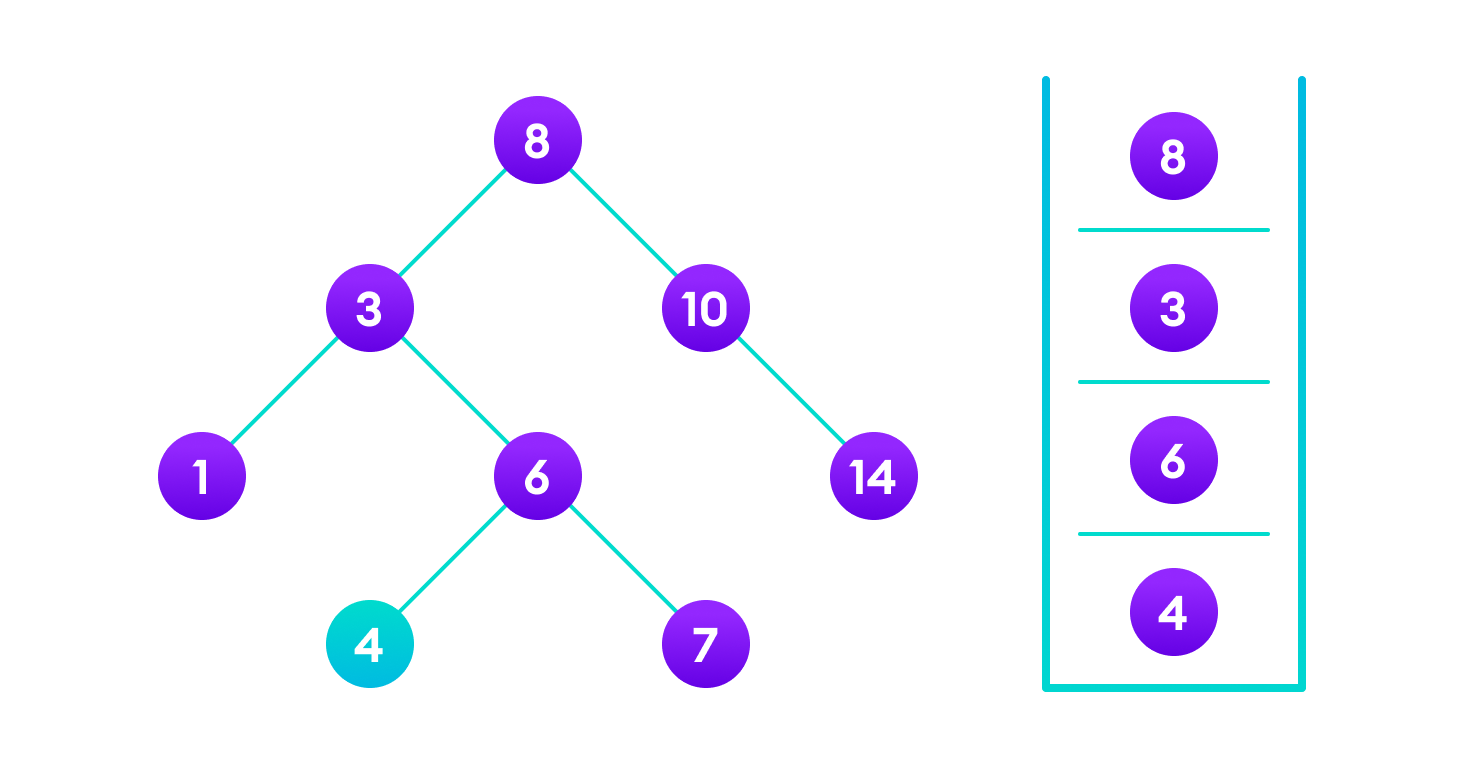
node->left = insert(node->left, data);else if (data > node->data)

node->right = insert(node->right, data); return node;

**4<8 so, transverse through the left child of 8**

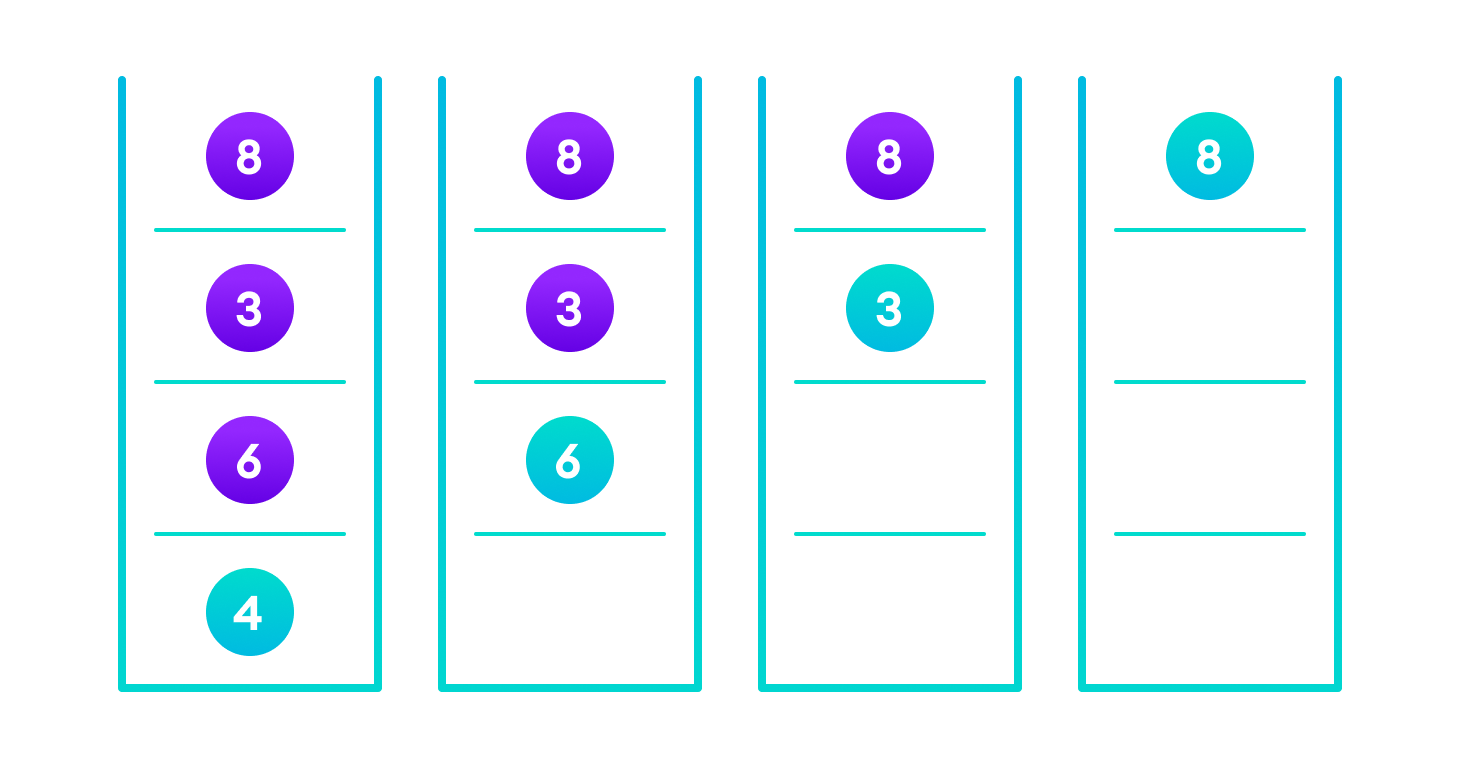
**4>3 so, transverse through the right child of 8**

**4<6 so, transverse through the left child of 6**

**Insert 4 as a left child of 6**

We have attached the node but we still have to exit from the function without doing any damage to the rest of the tree. This is where the return node; at the end comes in handy. In the case of NULL, the newly created node is returned and attached to the parent node, otherwise the same node is returned without any change as we go up until we return to the root.

This makes sure that as we move back up the tree, the other node connections aren't changed.

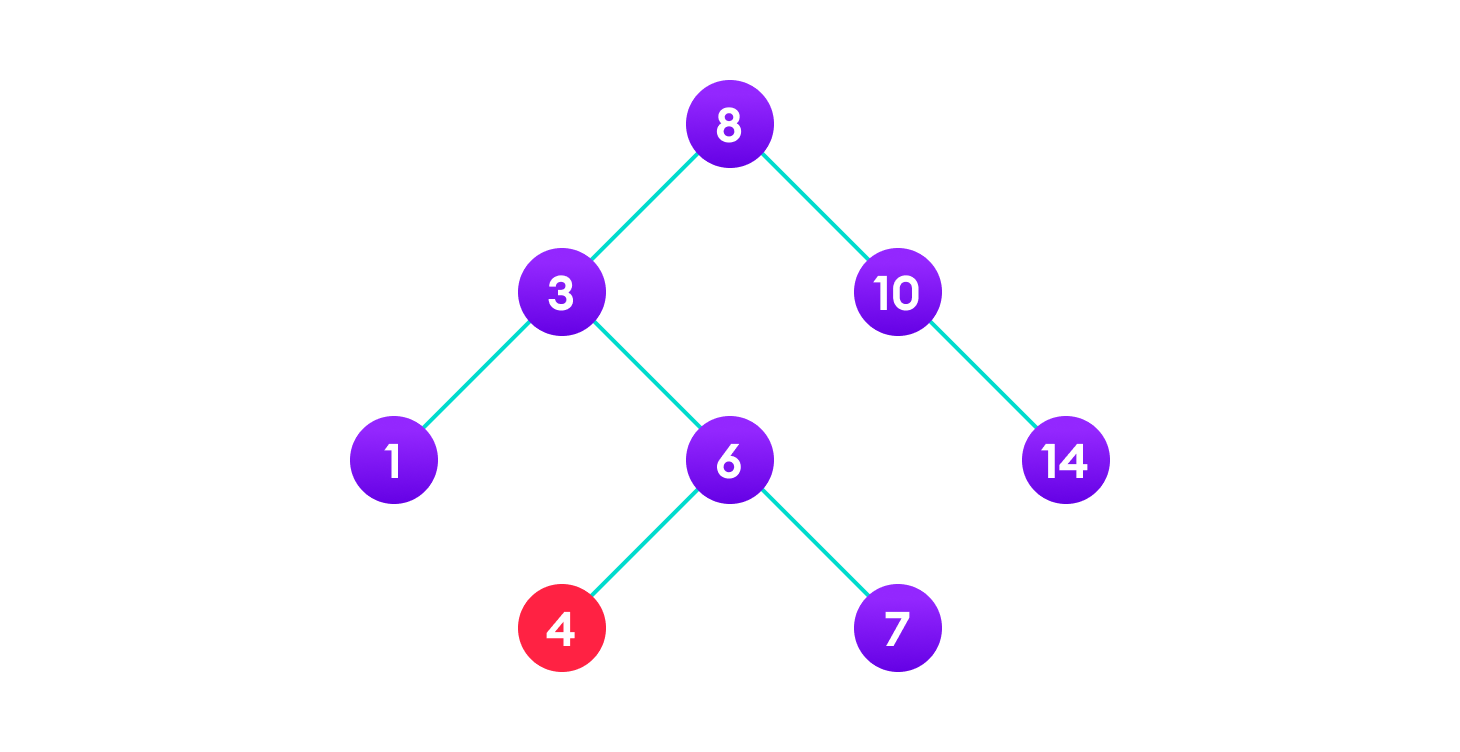
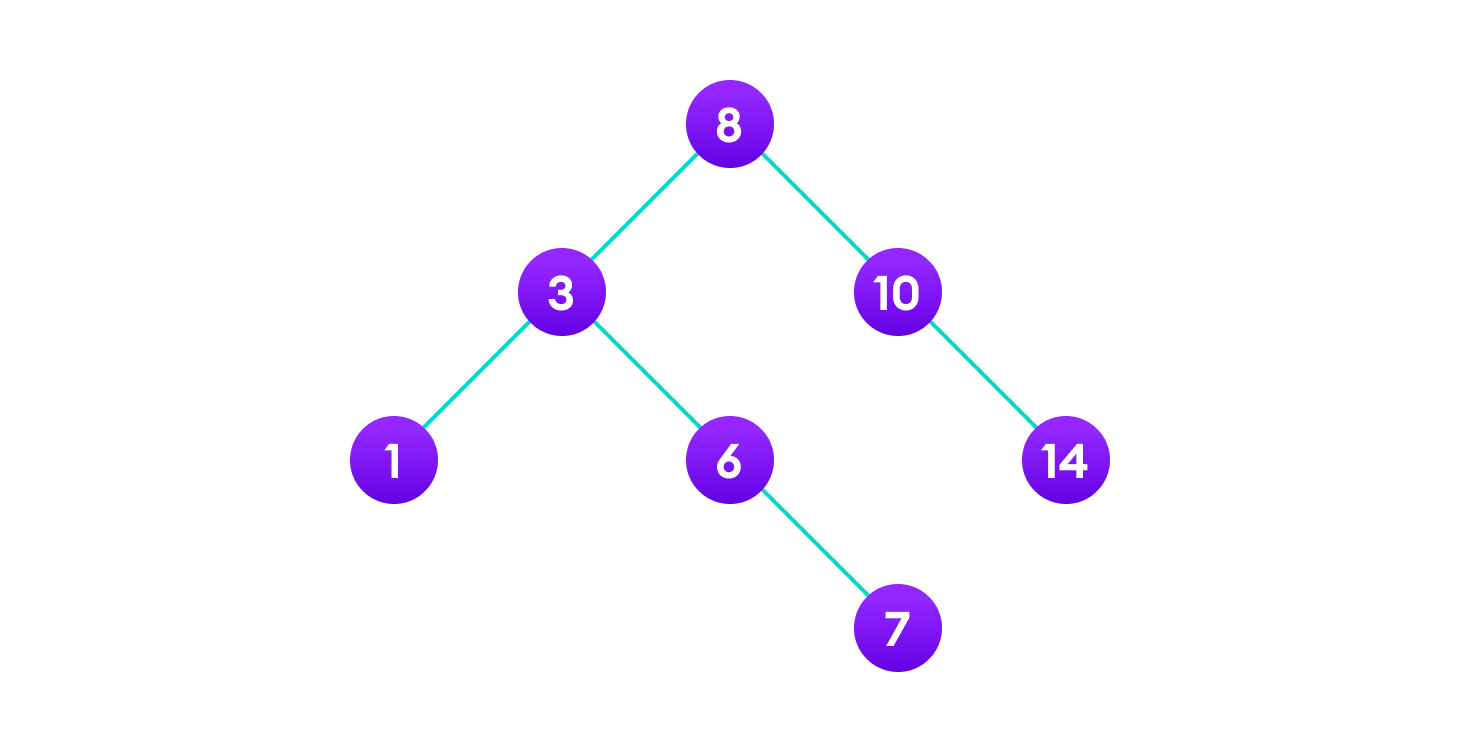


## Deletion Operation

There are three cases for deleting a node from a binary search tree.

### **Case I**

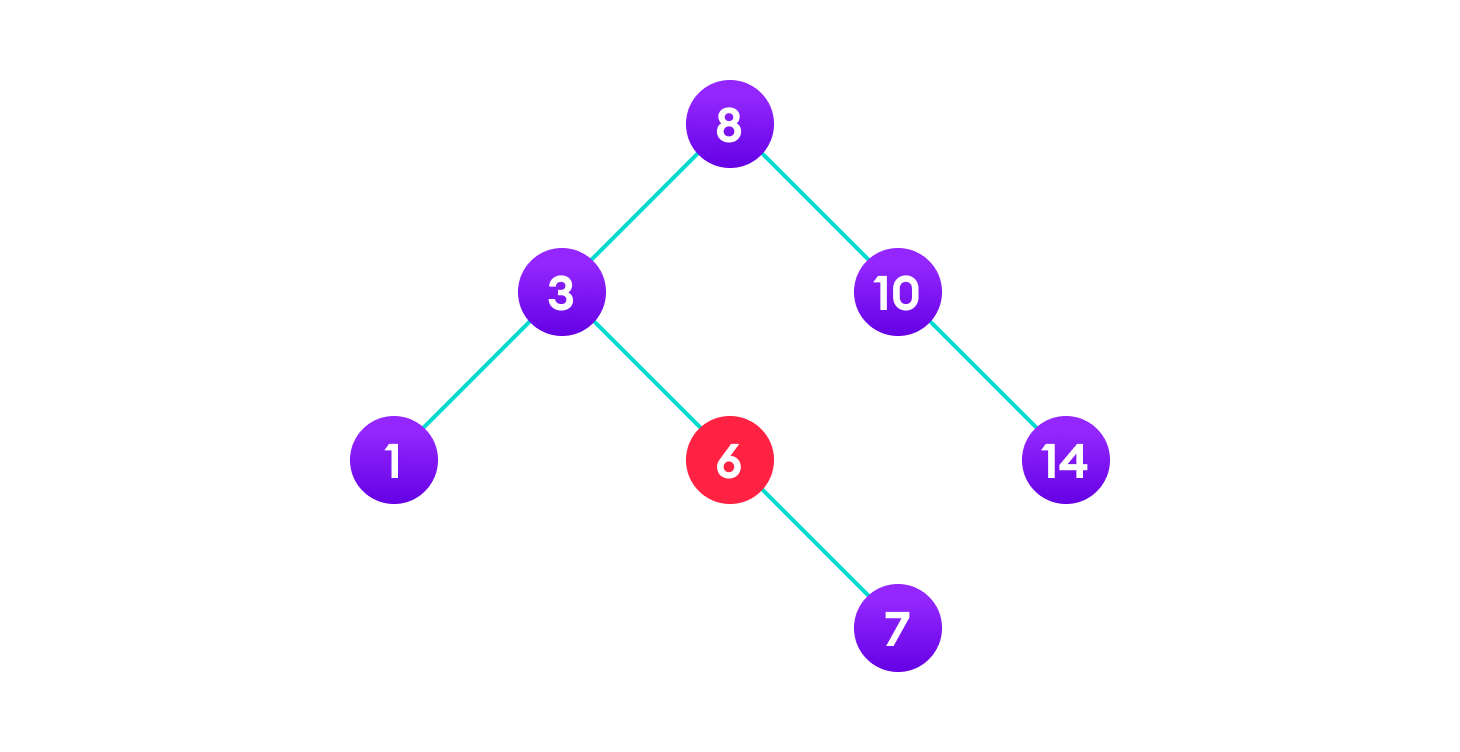
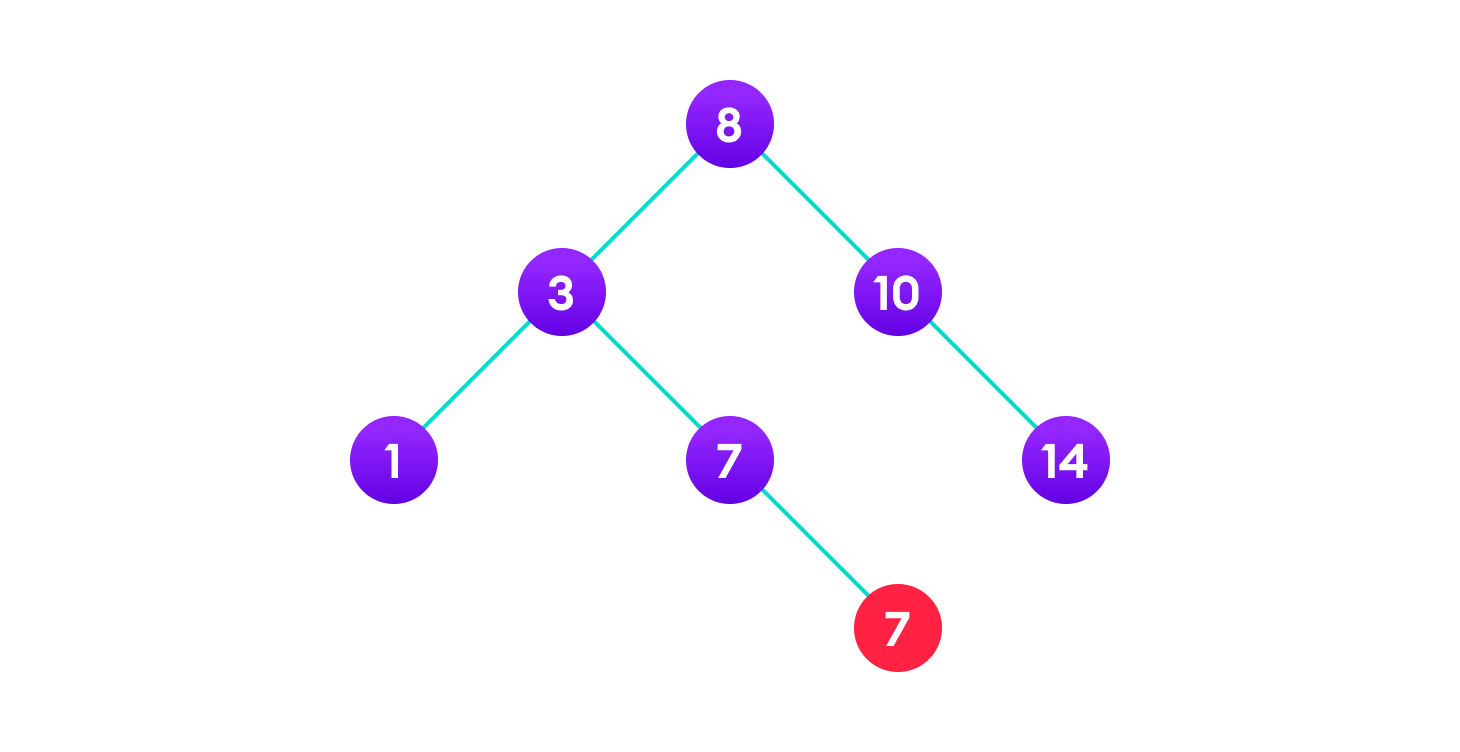
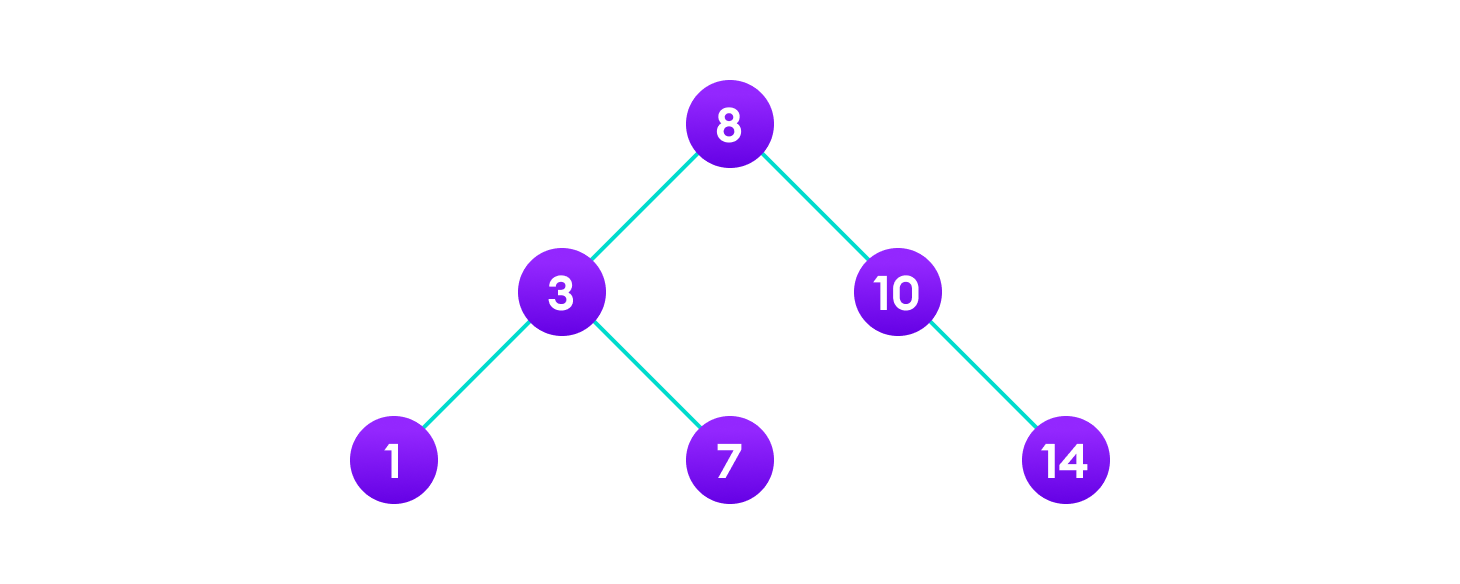
In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

**4 is to be deleted****Delete the node**

### **Case II**

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

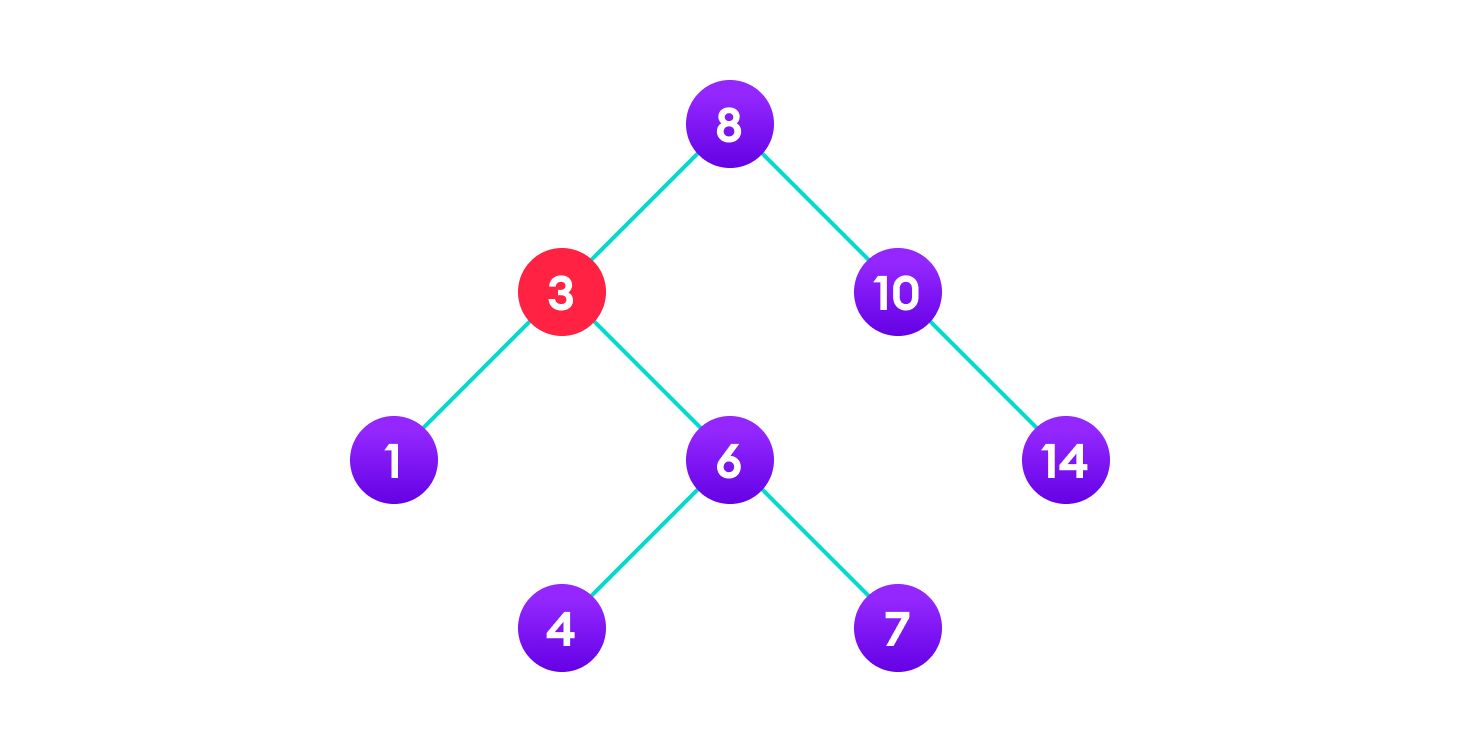
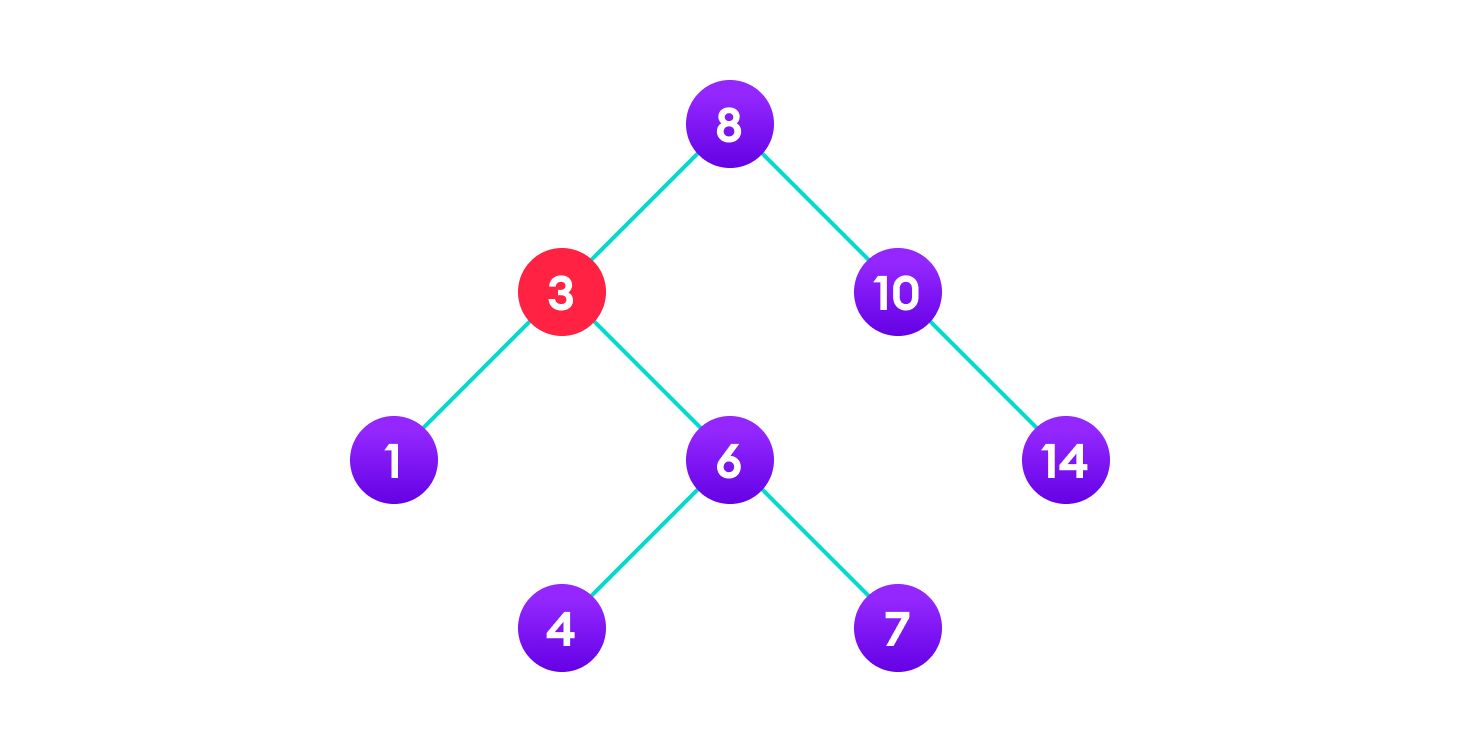
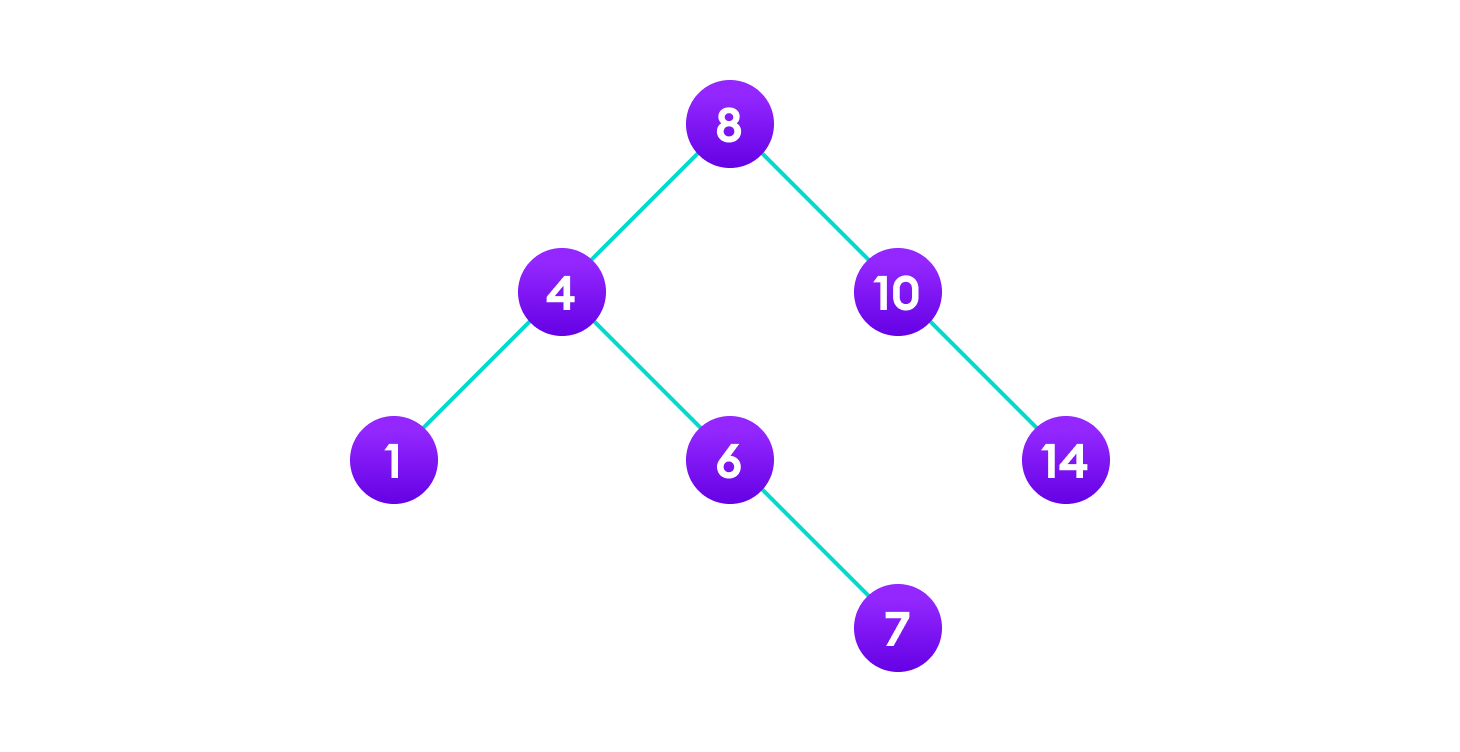
1. Replace that node with its child node.
2. Remove the child node from its original position.

**6 is to be deleted****copy the value of its child to the node and delete the child****Final tree**

### **Case III**

In the third case, the node to be deleted has two children. In such a case follow the steps below:

1. Get the inorder successor of that node.
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.

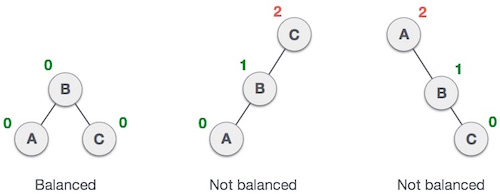
**3 is to be deleted****Copy the value of the inorder successor (4) to the node****Delete the inorder successor**

**AVL Tree:-**

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

Here we see that the first tree is balanced and the next two trees are not balanced −



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

***BalanceFactor***  = height(left-sutree) − height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

## AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations −

* Left rotation
* Right rotation
* Left-Right rotation
* Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

### Left Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

## Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

### Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

|  |  |
| --- | --- |
| **State** | **Action** |
| IMG_259 | A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation. |
| IMG_260 | We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**. |
| IMG_261 | Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree. |
| IMG_262 | We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree. |
| IMG_263 | The tree is now balanced. |

### Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

|  |  |
| --- | --- |
| **State** | **Action** |
| IMG_264 | A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2. |
| IMG_265 | First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**. |
| IMG_266 | Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation. |
| IMG_267 | A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**. |
| IMG_268 | The tree is now balanced. |

**Graph:-**

“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 connects the vertices. A graph is defined as follows... Generally, a graph G is represented as G = ( V , E ), where V is set of vertices and E is set of edges.”

**Basic Terminologies**

**Graph:-** A graph G consists of a nonempty set V called the set of nodes of the graph, a set E which is the set of edges of the graph and a mapping from the set of edges E to a set of pairs of element V. Figure I is the example of graph.

1

X3

X1

3

2 X2

X5

X4

4

[Figure I]

Here 1,2,3,4 are called nodes of the graph and X1, X2, X3, X4, X5 are called the edges of the graph.

**Adjacent Node:-** Any two nodes which are connected by an edge in a graph are called adjacent nodes. In figure I node 1 and 2, 1 and 3, 3 and 4, 4 and 2, 2 and 3 are called adjacent nodes.

**Directed & Undirected Edge:-** In a graph G=(V,E) an edge which is directed from one node to another is called Directed Edges, while an edge which has no specific directions is called Undirected Edge.

**Directed Graph:-** A graph in which every edge is directed is called Directed Graph or a digraph. Figure II is the example of directed graph.

1

X3

X1

3

2 X2

X5

X4

4

[Figure II]

**Undirected Graph:-** A graph in which every edge is undirected is called Undirected Graph. Figure III is the example of undirected graph.

1

X3

X1

3

2 X2

X5

X4

4

[Figure III]

**Mixed Graph:-** If some of the edges are directed and some are undirected in a graph then the graph is called Mixed Graph. Figure I is the example of Mixed graph.

**Initial & Terminal Nodes:-** Let (V,E) be a graph and let x € E be a directed edge associated with the ordered pair of nodes (u.v).Then the edge x is said to be initiating or originating in the node u and terminating or ending in the node v. The nodes u and v are also called the initial and terminal nodes of the edge x.

**Incident:-** An edge x € E which joins the nodes u and v, whether it be directed or undirected is said to be incident to the nodes u and v.

**Loop:-** An edge of a graph which joins a node to itself is called a loop (sling). (in Fig IV node 1 contains loop which is represented by edge X6)

X6

1

1

X3

X1

X7

3

2 X2

X5

X4

4

[Figure IV]

**Parallel:-** In some directed as well as undirected graph we may have certain pairs of nodes joined by more than one edge, Such edges are called Parallel. In figure IV there are parallel edges between nodes 1 and 3.

**Multigraph:-** Any graph which contains some parallel edges is called Multigraph. Figure IV represents multigraph.

**Simple Graph:-** On the other hand, if there is no more than one edge between a pair of nodes (no more than one directed edge in the case of a directed graph),than such a graph is called A Simple Graph. Figure I is the example of simple graph.

**Weighted Graph:-** A graph in which weights are assigned to every edge is called a Weighted graph.

**Isolated Node:-** In a graph a node which is not adjacent to any other nodes is called Isolated node. In figure V node 5 is isolated node.

1

X3

X1

3

2 X2

X5

X4

4

5

[Figure V]

**Null Graph:-** A graph containing only isolated nodes is called null graph.

Following is the example of null graph.

1

**Degree:-** In a directed graph, for any nodes v the number of edges which have v as their initial node is called the **out degree** of the node v. The number of edges which have v as their terminal node is called **in degree** of v, and some of the out degree and in degree of a node v is called its **total degree**.

In the case of an undirected graph, the total degree or the degree of a node v is equal to the number of edges incident with v. The total degree of a loop is 2 and that of an isolated node is 0.

**Path Of Graph:-** Any sequence of a digraph such that the terminal nodes of any edge in the sequence is the initial node of the edge, If any , appearing next in the sequence defines a path of graph.

**Simple Path:-** A path in digraph in which the edges are distinct(fixed) is called simple path.(edge simple)

**Elementary path:-** A path in which all the nodes through which it traverses are distinct is called an elementary path.(node simple)

**Cycle:-** A path which originates and ends in the same node is called a cycle (circuit).

**Acyclic:-** A simple digraph which does not have any cycles is called acyclic.

**Representation of Graph**

To represent the graph a matrix is used called an **adjacency matrix**, in which the rows and columns of two dimensional array represent source and destination vertices and entries in graph indicate whether an edges exists between the vertices.

Let G=(V,E) be a simple digraph in which V={v1,v2,…,vn} and the nodes are assumed to be ordered from v1 to vn. An n\*n matrix A whose elements Aij are given by

1 if (vi,Vj) € E

Aij= {0 Otherwise

Is called adjancy matrix of the graph G.

Example

V1 V4

V2 V3

For the above graph adjacency matrix is as below

V1 V2 V3 V4

V1 0 1 0 1

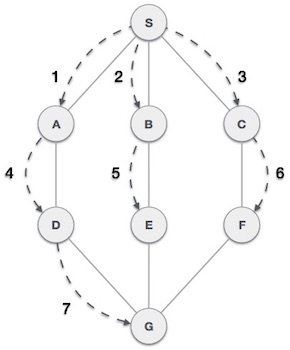
V2 1 0 0 0

V3 1 1 0 1

V4 0 1 0 0

**BFS:-**

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



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

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

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

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

|  |  |  |
| --- | --- | --- |
| **Step** | **Traversal** | **Description** |
| 1 | IMG_257 | Initialize the queue. |
| 2 | IMG_258 | We start from visiting **S** (starting node), and mark it as visited. |
| 3 | IMG_259 | We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it. |
| 4 | IMG_260 | Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it. |
| 5 | IMG_261 | Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it. |
| 6 | IMG_262 | Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**. |
| 7 | IMG_263 | From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it. |

**DFS:-**

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



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

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

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

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

|  |  |  |
| --- | --- | --- |
| **Step** | **Traversal** | **Description** |
| 1 | IMG_257 | Initialize the stack. |
| 2 | IMG_258 | Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order. |
| 3 | IMG_259 | Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only. |
| 4 | IMG_260 | Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order. |
| 5 | IMG_261 | We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack. |
| 6 | IMG_262 | We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack. |
| 7 | IMG_263 | Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack. |

**Sorting and Searching Technique:-**

The operation of sorting is most often performed in business data processing application. Sorting is the operation of arranging the records of a table into some sequential order according to an ordering criterion. The sort is performed on records according to the key value of each record. In numerical sorting, the records are arranged in ascending or descending order according to the numerical value of the key.

**SORTING.**

* Retrieval information is made easier when it is stored in some predefined order.
* Sorting is very important to the computer activity.
* Many sorting algorithms are available different environment required Retrieval information is made easier. When it is stored in some different sorting method.
* Algorithms are characterized in following two ways.

1. Simple sorting Algorithm which required order of n2 (written as O (n2)) comparison to sort n item.
2. Sophisticated algorithm that required. (N log1n) comparisons to sort n items.

* In the first method moves data only over small distances in process of sorting .so it will take time more.
* In second method move data over large. Distances. So that items settle in to proper order sooner. Thus result in fewer comparisons.
* **There are two basic categories for sorting**.

1. Internal sorting.
2. External sorting.

**1. Internal sorting.**

* Internal sorting are applied when the entire collection of data to sort in small enough.
* Sorting is take place with in main memory.
* The time required to read or write is not consider to be significant in evaluating the performance of internal sorting.
* Internal sorting algorithm. .
* Bubble - Binary tree sorting.
* Insertion - Binary insertion sort.
* Shell - Address calculation sort.
* Quick. - Help sort.

**2. External sorting**

* External sorting method applied on large collection of data which reside on auxiliary memory device such as magnetic tabs or disks.
* In external sorting read and write access times are major concerns in determining sort performance.
* External sorting algorithm…
* Merge sort.
* Radix sort.
* Polyphase sort.

**Bubble Sort:-**

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where **n** is the number of items.

## How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes Ο(n2) time so we're keeping it short and precise.

IMG_256

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

IMG_257

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

IMG_258

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

IMG_259

The new array should look like this −

IMG_260

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

IMG_261

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

IMG_262

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

IMG_263

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

IMG_264

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

IMG_265

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

IMG_266

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

IMG_267

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

## Algorithm

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

begin BubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

return list

end BubbleSort

**Insertion Sort:-**

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

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

## How Insertion Sort Works?

We take an unsorted array for our example.

IMG_256

Insertion sort compares the first two elements.

IMG_257

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

IMG_258

Insertion sort moves ahead and compares 33 with 27.

IMG_259

And finds that 33 is not in the correct position.

IMG_260

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

IMG_261

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

IMG_262

These values are not in a sorted order.

IMG_263

So we swap them.

IMG_264

However, swapping makes 27 and 10 unsorted.

IMG_265

Hence, we swap them too.

IMG_266

Again we find 14 and 10 in an unsorted order.

IMG_267

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

IMG_268

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

### Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

**Step 1** − If it is the first element, it is already sorted. return 1;

**Step 2** − Pick next element

**Step 3** − Compare with all elements in the sorted sub-list

**Step 4** − Shift all the elements in the sorted sub-list that is greater than the

value to be sorted

**Step 5** − Insert the value

**Step 6** − Repeat until list is sorted

**Selection Sort:-**

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of Ο(n2), where **n** is the number of items.

## How Selection Sort Works?

Consider the following depicted array as an example.

IMG_256

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

IMG_257

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

IMG_258

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

IMG_259

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

IMG_260

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

IMG_261

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



Now, let us learn some programming aspects of selection sort.

### Algorithm

**Step 1** − Set MIN to location 0

**Step 2** − Search the minimum element in the list

**Step 3** − Swap with value at location MIN

**Step 4** − Increment MIN to point to next element

**Step 5** − Repeat until list is sorted

**Merge Sort:-**

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

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

## How Merge Sort Works?

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

IMG_256

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

IMG_257

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

IMG_258

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

IMG_259

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

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

IMG_260

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

IMG_261

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

IMG_262

Now we should learn some programming aspects of merge sorting.

### 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.

**Shell Sort:-**

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**. This interval is calculated based on Knuth's formula as −

### Knuth's Formula

h = h \* 3 + 1

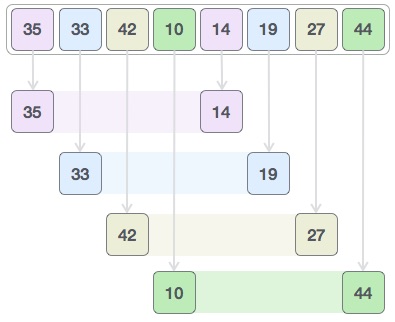
where −

h is interval with initial value 1

This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is Ο(n), where n is the number of items. And the worst case space complexity is O(n).

## How Shell Sort Works?

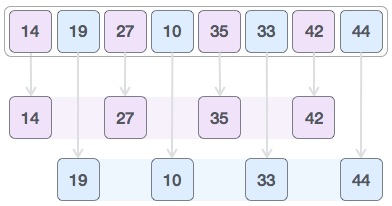
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values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



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

IMG_257

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



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

IMG_259

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

Following is the step-by-step depiction −



We see that it required only four swaps to sort the rest of the array.

### Algorithm

Following is the algorithm for shell sort.

**Step 1** − Initialize the value of *h*

**Step 2** − Divide the list into smaller sub-list of equal interval *h*

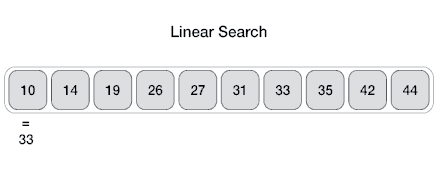
**Step 3** − Sort these sub-lists using **insertion sort**

**Step 3** − Repeat until complete list is sorted

**Searching Technique:-**

**Linear Search:-**

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



## Algorithm

Linear Search ( Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

**Binary Search:-**

Binary search is a fast search algorithm with run-time complexity of Ο(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

## How Binary Search Works?

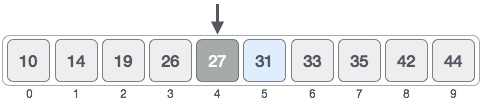
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0 ) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



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



We change our low to mid + 1 and find the new mid value again.

low = mid + 1

mid = low + (high - low) / 2

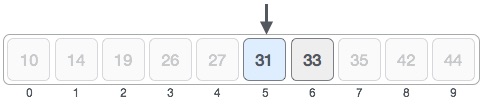
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



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



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.