Data Structures

- Data Structures are the programmatic way of storing data so that data can be used efficiently.
- Data Structure is a systematic way to organize data in order to use it efficiently.

Characteristics of a Data Structure

- **Correctness** – Data structure implementation should implement its interface correctly.

- **Time Complexity** – Running time or the execution time of operations of data structure must be as small as possible.

- **Space Complexity** – Memory usage of a data structure operation should be as little as possible.

Execution Time Cases

There are three cases which are usually used to compare various data structure's execution time in a relative manner.

- **Worst Case** – This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is ƒ(n) then this operation will not take more than ƒ(n) time where ƒ(n) represents function of n.

- **Average Case** – This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes f(n) time in execution, then m operations will take mƒ(n) time.

- **Best Case** – This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution, then the actual operation may take time as the random number which would be maximum as ƒ(n).

Data Type

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

- Built-in Data Type
- Derived Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built−in data types and associated operations on them. For example −

- List
- Array
- Stack
- Queue

Basic Operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

Arrays

Array is a container which can hold a fixed number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- Element – Each item stored in an array is called an element.
- Index – Each location of an element in an array has a numerical index, which is used to identify the element.

Linked List

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

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

Linked List Representation

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

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

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

Types of Linked List

Following are the various types of linked lists.

- Simple Linked List − Item navigation is forward only.
- Doubly Linked List − Items can be navigated forward and backward.
- Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations

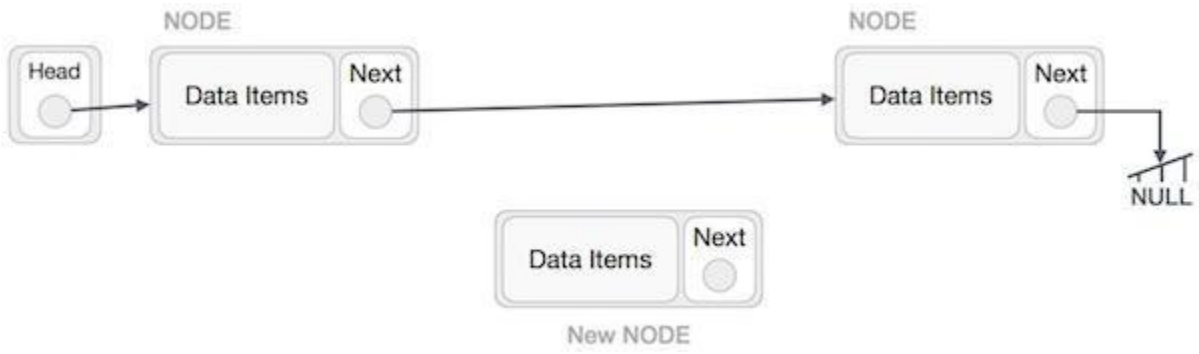
Following are the basic operations supported by a list.

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

- **Delete** – Deletes an element using the given key.

Insertion Operation

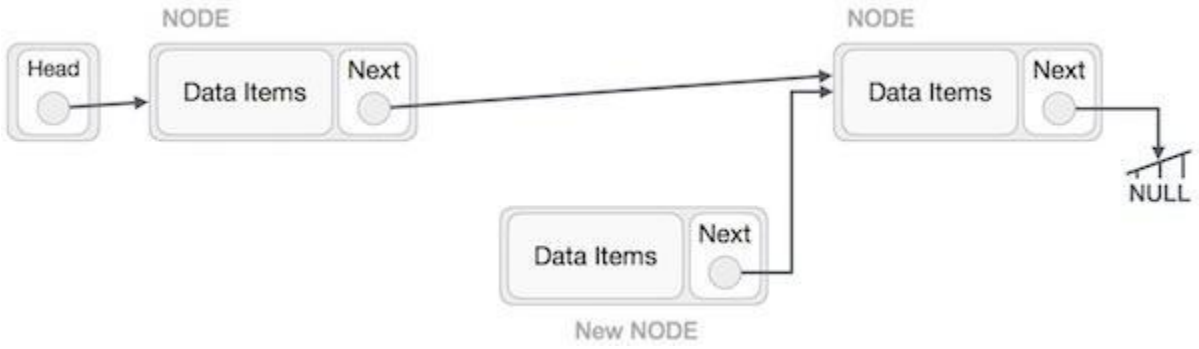
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C –

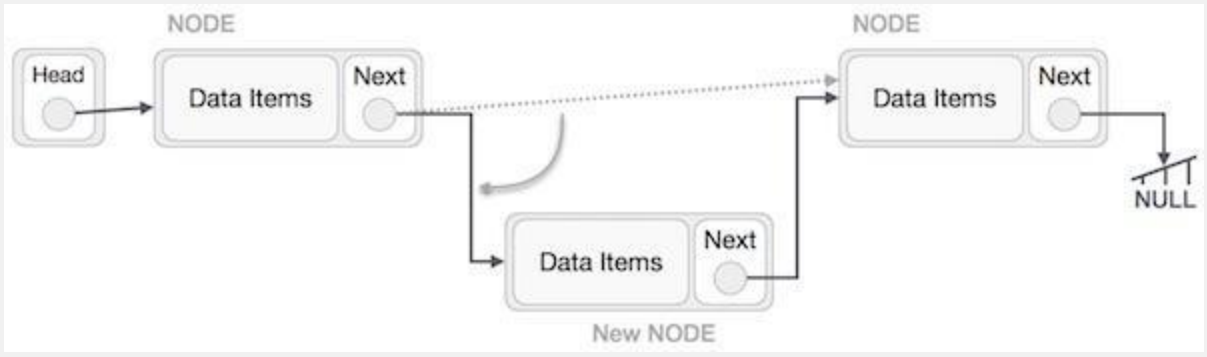
```
NewNode.next -> RightNode;
```

It should look like this –



Now, the next node at the left should point to the new node.

```
LeftNode.next -> NewNode;
```

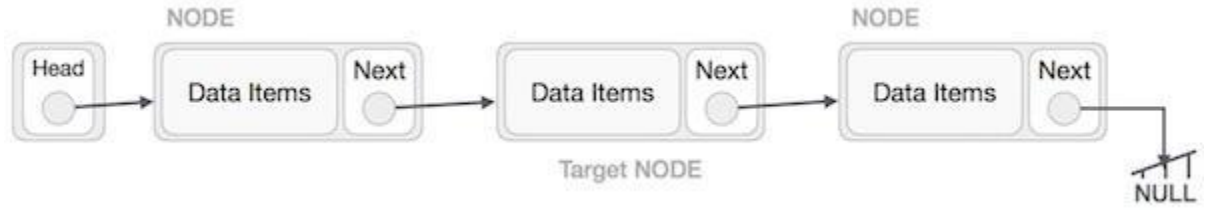


This will put the new node in the middle of the two. The new list should look like this –

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



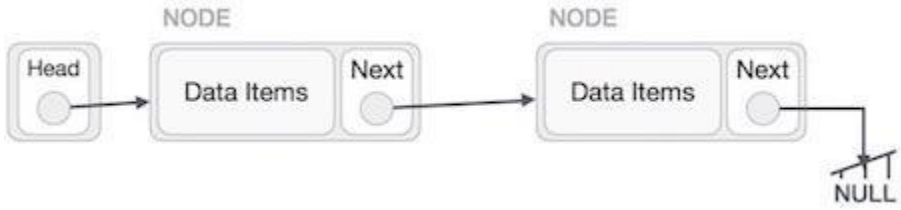
The left (previous) node of the target node now should point to the next node of the target node –

```
LeftNode.next -> TargetNode.next;
```

This will remove the link that was pointing to the target node. Now, using the following code, we will remove what what the target node is pointing at.

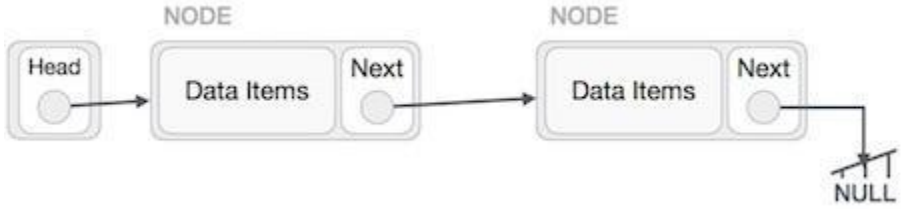
```
TargetNode.next -> NULL;
```

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

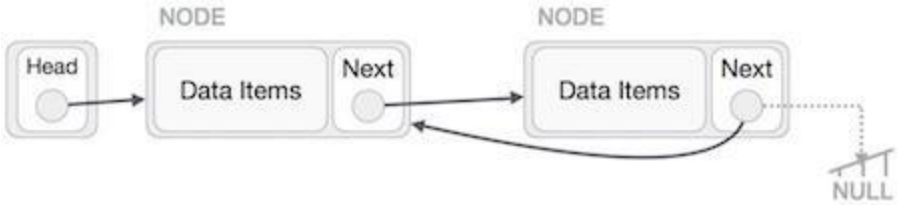


Reverse Operation

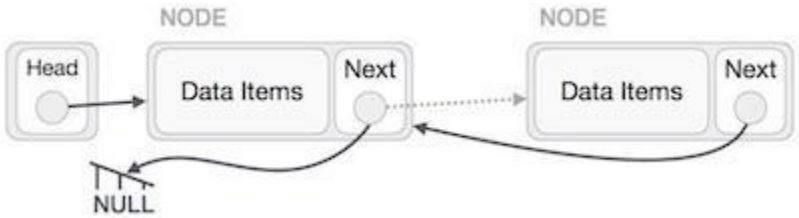
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



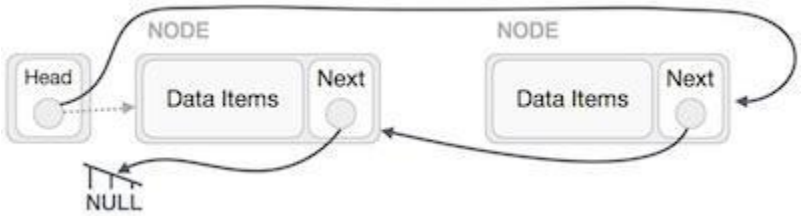
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



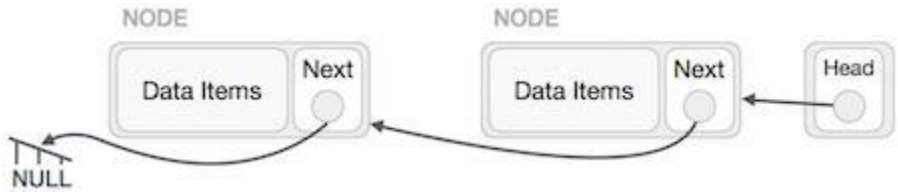
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



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



We'll make the head node point to the new first node by using the temp node.



Program:

C

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

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

struct node {
  int value;
  struct node *next;
};

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

typedef struct node DATA_NODE;

DATA_NODE *head_node, *first_node, *temp_node = 0, *prev_node, next_node;
int data;

int main() {
  int option = 0;

  printf("Singly Linked List Example - All Operations\n");

  while (option < 5) {

    printf("\nOptions\n");
    printf("1 : Insert into Linked List \n");
    printf("2 : Delete from Linked List \n");
    printf("3 : Display Linked List\n");
    printf("4 : Count Linked List\n");
    printf("Others : Exit()\n");
    printf("Enter your option:");
    scanf("%d", &option);
    switch (option) {
      case 1:
        insert();
        break;
      case 2:
        delete();
        break;
```

```
      case 3:
        display();
        break;
      case 4:
        count();
        break;
      default:
        break;
    }
  }

  return 0;
}

void insert() {
  printf("\nEnter Element for Insert Linked List : \n");
  scanf("%d", &data);

  temp_node = (DATA_NODE *) malloc(sizeof (DATA_NODE));

  temp_node->value = data;

  if (first_node == 0) {
    first_node = temp_node;
  } else {
    head_node->next = temp_node;
  }
  temp_node->next = 0;
  head_node = temp_node;
  fflush(stdin);
}

void delete() {
  int countvalue, pos, i = 0;
  countvalue = count();
  temp_node = first_node;
  printf("\nDisplay Linked List : \n");

  printf("\nEnter Position for Delete Element : \n");
  scanf("%d", &pos);
```

```
  if (pos > 0 && pos <= countvalue) {
    if (pos == 1) {
      temp_node = temp_node -> next;
      first_node = temp_node;
      printf("\nDeleted Successfully \n\n");
    } else {
      while (temp_node != 0) {
        if (i == (pos - 1)) {
          prev_node->next = temp_node->next;
          if(i == (countvalue - 1))
          {
                                    head_node = prev_node;
                        }
          printf("\nDeleted Successfully \n\n");
          break;
        } else {
          i++;
          prev_node = temp_node;
          temp_node = temp_node -> next;
        }
      }
    }
  } else
    printf("\nInvalid Position \n\n");
}
void display() {
  int count = 0;
  temp_node = first_node;
  printf("\nDisplay Linked List : \n");
  while (temp_node != 0) {
    printf("# %d # ", temp_node->value);
    count++;
    temp_node = temp_node -> next;
  }
  printf("\nNo Of Items In Linked List : %d\n", count);
}
int count() {
  int count = 0;
  temp_node = first_node;
```

```
  while (temp_node != 0) {
    count++;
    temp_node = temp_node -> next;
  }
  printf("\nNo Of Items In Linked List : %d\n", count);
  return count;
}
```

Java

```
import java.io.*;

// Java program to implement
// a Singly Linked List
public class LinkedList {

	Node head; // head of list

	// Linked list Node.
	// Node is a static nested class
	// so main() can access it
	static class Node {

		int data;
		Node next;

		// Constructor
		Node(int d)
		{
			data = d;
			next = null;
		}
	}

	// Method to insert a new node
	public static LinkedList insert(LinkedList list,
                                        int data)
	{
		// Create a new node with given data
		Node new_node = new Node(data);
		new_node.next = null;
```

```
                // If the Linked List is empty,
                // then make the new node as head
                if (list.head == null) {
                        list.head = new_node;
                }
                else {
                        // Else traverse till the last node
                        // and insert the new_node there
                        Node last = list.head;
                        while (last.next != null) {
                                last = last.next;
                        }

                        // Insert the new_node at last node
                        last.next = new_node;
                }

                // Return the list by head
                return list;
        }

        // Method to print the LinkedList.
        public static void printList(LinkedList list)
        {
                Node currNode = list.head;

                System.out.print("LinkedList: ");

                // Traverse through the LinkedList
                while (currNode != null) {
                        // Print the data at current node
                        System.out.print(currNode.data + " ");

                        // Go to next node
                        currNode = currNode.next;
                }

                System.out.println();
        }
```

```
	// Method to delete a node in the LinkedList by POSITION
	public static LinkedList
	deleteAtPosition(LinkedList list, int index)
	{
		// Store head node
		Node currNode = list.head, prev = null;

		//
		// CASE 1:
		// If index is 0, then head node itself is to be
		// deleted

		if (index == 0 && currNode != null) {
			list.head = currNode.next; // Changed head

			// Display the message
			System.out.println(
				index + " position element deleted");

			// Return the updated List
			return list;
		}

		//
		// CASE 2:
		// If the index is greater than 0 but less than the
		// size of LinkedList
		//
		// The counter
		int counter = 0;

		// Count for the index to be deleted,
		// keep track of the previous node
		// as it is needed to change currNode.next
		while (currNode != null) {

			if (counter == index) {
				// Since the currNode is the required
				// position Unlink currNode from linked list
```

```
                                prev.next = currNode.next;

                                // Display the message
                                System.out.println(
                                        index + " position element deleted");
                                break;
                        }
                        else {
                                // If current position is not the index
                                // continue to next node
                                prev = currNode;
                                currNode = currNode.next;
                                counter++;
                        }
                }

                // If the position element was found, it should be
                // at currNode Therefore the currNode shall not be
                // null
                //
                // CASE 3: The index is greater than the size of the
                // LinkedList
                //
                // In this case, the currNode should be null
                if (currNode == null) {
                        // Display the message
                        System.out.println(
                                index + " position element not found");
                }

                // return the List
                return list;
        }

        // **************MAIN METHOD**************

        // method to create a Singly linked list with n nodes
        public static void main(String[] args)
        {
                /* Start with the empty list. */
```

```
        LinkedList list = new LinkedList();

        //
        // ******INSERTION******
        //

        // Insert the values
        list = insert(list, 1);
        list = insert(list, 2);
        list = insert(list, 3);
        list = insert(list, 4);
        list = insert(list, 5);
        list = insert(list, 6);
        list = insert(list, 7);
        list = insert(list, 8);

        // Print the LinkedList
        printList(list);

        //
        // ******DELETION AT POSITION******
        //

        // Delete node at position 0
        // In this case the key is ***at head***
        deleteAtPosition(list, 0);

        // Print the LinkedList
        printList(list);

        // Delete node at position 2
        // In this case the key is present ***in the
        // middle***
        deleteAtPosition(list, 2);

        // Print the LinkedList
        printList(list);

        // Delete node at position 10
        // In this case the key is ***not present***
```

```
                deleteAtPosition(list, 10);

                // Print the LinkedList
                printList(list);
        }
}
```

Python

```
# class node to store data and next
class Node:

  def __init__(self,data, next=None):
    self.data = data
    self.next = next

  # defining getter and setter for data and next
  def getData(self):
    return self.data

  def setData(self, data):
    self.data = data

  def getNextNode(self):
    return self.next

  def setNextNode(self, node):
    self.next = node

# class Linked List
class LinkedList:

  def __init__(self, head=None):
    self.head = head
    self.size = 0

  def getSize(self):
    return self.size

  def addNode(self, data):
    node = Node(data, self.head)
```

```
    self.head = node
    # incrementing the size of the linked list
    self.size += 1
    return True

  # delete a node from linked list
  def removeNode(self, value):
    prev = None
    curr = self.head
    while curr:
      if curr.getData() == value:
        if prev:
            prev.setNextNode(curr.getNextNode())
        else:
            self.head = curr.getNextNode()
        return True

      prev = curr
      curr = curr.getNextNode()

    return False

  # find a node in the linked list
  def findNode(self,value):
    curr = self.head
    while curr:
      if curr.getData() == value:
        return True
      else:
        curr = curr.getNextNode()
    return False

  # print the linked list
  def printLL(self):
    curr = self.head
    while curr:
      print(curr.data)
      curr = curr.getNextNode()
```

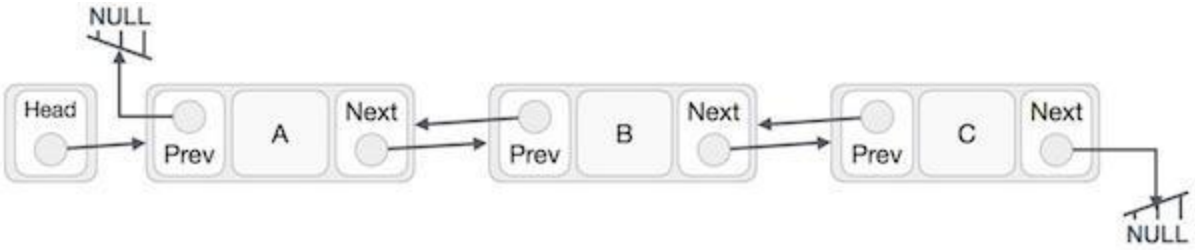
```
myList = LinkedList()
print("Inserting")
print(myList.addNode(5))
print(myList.addNode(15))
print(myList.addNode(25))

myList.printLL()

print(myList.getSize())

print(myList.findNode(25))

print(myList.removeNode(25))

myList.printLL()
```

Doubly Linked List

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked lists.

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

Doubly Linked List Representation



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

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

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Insert Last** – Adds an element at the end of the list.
- **Delete Last** – Deletes an element from the end of the list.
- **Insert After** – Adds an element after an item of the list.
- **Delete** – Deletes an element from the list using the key.
- **Display forward** – Displays the complete list in a forward manner.
- **Display backward** – Displays the complete list in a backward manner.

Program:

C

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

```
#include<stdlib.h>
struct node
{
   struct node *prev;
   struct node *next;
   int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void search();
void main ()
{
int choice =0;
   while(choice != 9)
   {
        printf("\n1.Insert in begining\n
                    2.Insert at last\n
                    3.Insert at any random location\n
                    4.Delete from Beginning\n
                    5.Delete from last\n
                    6.Delete the node after the given data\n
                    7.Search\n
                    8.Show\n
                    9.Exit\n");

       printf("\nEnter your choice?\n");
       scanf("\n%d",&choice);
       switch(choice)
       {
          case 1:
          insertion_beginning();
          break;
          case 2:
                insertion_last();
          break;
          case 3:
          insertion_specified();
          break;
```

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

   if(head==NULL)
   {
      ptr->next = NULL;
      ptr->prev=NULL;
      ptr->data=item;
      head=ptr;
   }
```

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

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

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

```
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        ptr = head;
        head = head -> next;
        head -> prev = NULL;
        free(ptr);
        printf("\nnode deleted\n");
    }

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

```
    printf("\n Enter the data after which the node is to be deleted : ");
    scanf("%d", &val);
    ptr = head;
    while(ptr -> data != val)
    ptr = ptr -> next;
    if(ptr -> next == NULL)
    {
        printf("\nCan't delete\n");
    }
    else if(ptr -> next -> next == NULL)
    {
        ptr ->next = NULL;
    }
    else
    {
        temp = ptr -> next;
        ptr -> next = temp -> next;
        temp -> next -> prev = ptr;
        free(temp);
        printf("\nnode deleted\n");
    }
}
void display()
{
    struct node *ptr;
    printf("\n printing values...\n");
    ptr = head;
    while(ptr != NULL)
    {
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
}
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
```

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

}
```

Java

```
package org.arpit.java2blog.algo;

class Node {
    public int data;
    public Node next;
    public Node prev;

    public void displayNodeData() {
        System.out.println("{ " + data + " } ");
    }
}

public class DoublyLinkedList {

    private Node head;
    private Node tail;
    int size;
```

```
    public boolean isEmpty() {
        return (head == null);
    }

    // used to insert a node at the start of linked list
    public void insertFirst(int data) {
        Node newNode = new Node();
        newNode.data = data;
        newNode.next = head;
        newNode.prev=null;
        if(head!=null)
            head.prev=newNode;
        head = newNode;
        if(tail==null)
            tail=newNode;
        size++;
    }

    // used to insert a node at the start of linked list
    public void insertLast(int data) {
        Node newNode = new Node();
        newNode.data = data;
        newNode.next = null;
        newNode.prev=tail;
        if(tail!=null)
            tail.next=newNode;
        tail = newNode;
        if(head==null)
            head=newNode;
        size++;
    }
    // used to delete node from start of Doubly linked list
    public Node deleteFirst() {

        if (size == 0)
            throw new RuntimeException("Doubly linked list is already empty");
        Node temp = head;
        head = head.next;
        head.prev = null;
        size--;
        return temp;
    }

    // used to delete node from last of Doubly linked list
```

```
    public Node deleteLast() {

        Node temp = tail;
        tail = tail.prev;
        tail.next=null;
        size--;
        return temp;
    }

    // Use to delete node after particular node
    public void deleteAfter(Node after) {
        Node temp = head;
        while (temp.next != null && temp.data != after.data) {
            temp = temp.next;
        }
        if (temp.next != null)
            temp.next.next.prev=temp;
        temp.next = temp.next.next;

    }

    // For printing Doubly Linked List forward
    public void printLinkedListForward() {
        System.out.println("Printing Doubly LinkedList (head --> tail) ");
        Node current = head;
        while (current != null) {
            current.displayNodeData();
            current = current.next;
        }
        System.out.println();
    }

    // For printing Doubly Linked List forward
    public void printLinkedListBackward() {
        System.out.println("Printing Doubly LinkedList (tail --> head) ");
        Node current = tail;
        while (current != null) {
            current.displayNodeData();
            current = current.prev;
        }
        System.out.println();
    }
}
```

```
public class DoubleLinkedListMain {

 public static void main(String args[])
 {
   DoublyLinkedList myLinkedlist = new DoublyLinkedList();
   myLinkedlist.insertFirst(5);
   myLinkedlist.insertFirst(6);
   myLinkedlist.insertFirst(7);
   myLinkedlist.insertFirst(1);
   myLinkedlist.insertLast(2);
   myLinkedlist.printLinkedListForward();
   myLinkedlist.printLinkedListBackward();

   System.out.println("================");
   // Doubly Linked list will be
   // 1 ->  7 -> 6 -> 5 -> 2

   Node node=new Node();
   node.data=1;
   myLinkedlist.deleteAfter(node);
   myLinkedlist.printLinkedListForward();
   myLinkedlist.printLinkedListBackward();
   // After deleting node after 1,doubly Linked list will be
   // 2 -> 1 -> 6 -> 5
   System.out.println("================");
   myLinkedlist.deleteFirst();
   myLinkedlist.deleteLast();

   // After performing above operation, doubly Linked list will be
   //  6 -> 5
   myLinkedlist.printLinkedListForward();
   myLinkedlist.printLinkedListBackward();
 }
}
```

Python

```
class Node(object):
    def __init__(self, data, Next = None, Previous = None):
        self.data = data
        self.next = Next
        self.previous = Previous

    def getNext(self):
```

```
        return self.next

    def getPrevious(self):
        return self.previous

    def getData(self):
        return self.data

    def setData(self, newData):
        self.data = newData

    def setNext(self, newNext):
        self.next = newNext

    def setPrevious(self, newPrevious):
        self.previous = newPrevious


class LinkedList(object):
    def __init__(self):
        self.head = None

    def isEmpty(self):
        return self.head == None

    def insertFirst(self, data):
        newNode = Node(data)
        if self.head:
            self.head.setPrevious(newNode)
        newNode.setNext(self.head)
        self.head = newNode

    def insertLast(self, data):
        newNode = Node(data)
        current = self.head
        while current.getNext() != None:
            current = current.getNext()
        current.setNext(newNode)
        newNode.setPrevious(current)
```

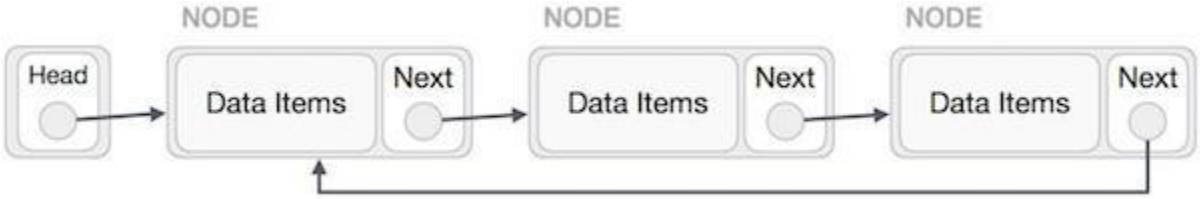
```
    def getAllData(self):
        ''' This function displays the data elements of the Linked List '''
        current = self.head
        elements = []
        while current:
            elements.append(current.getData())
            current = current.getNext()

        return elements

    def remove(self,item):
        current = self.head
        previous = None
        found = False
        while not found:
            if current.getData() == item:
                found = True
            else:
                previous = current
                current = current.getNext()

        if previous == None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())

if __name__ == '__main__':
    myList = LinkedList()
    myList.insertFirst(1)
    myList.insertFirst(12)
    myList.insertFirst(32)
    myList.insertFirst(22)
    myList.insertLast(2)
    myList.remove(12)
    print(myList.getAllData())
```

Circular Linked List

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

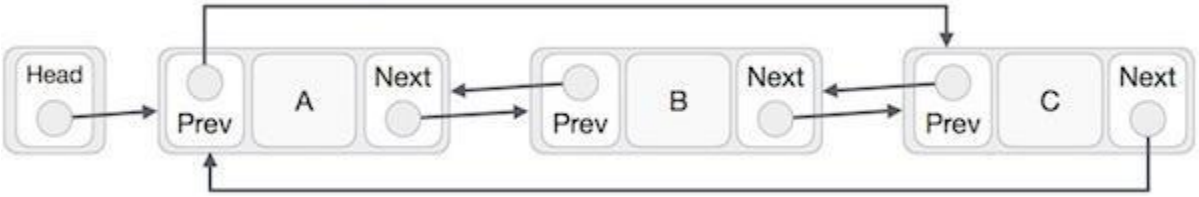
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



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

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

- **insert** – Inserts an element at the start of the list.
- **delete** – Deletes an element from the start of the list.
- **display** – Displays the list.

Program :

C

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

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
   int choice =0;
   while(choice != 7)
   {
      printf("\n1.Insert in begining\n
                  2.Insert at last\n
                  3.Delete from Beginning\n
                  4.Delete from last\n
                   5.Search for an element\n
                   6.Show\n
                   7.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:
         beginsert();
         break;
         case 2:
```

```
            lastinsert();
            break;
            case 3:
            begin_delete();
            break;
            case 4:
            last_delete();
            break;
            case 5:
            search();
            break;
            case 6:
            display();
            break;
            case 7:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void beginsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter the node data?");
        scanf("%d",&item);
        ptr -> data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
```

```
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
        printf("\nnode inserted\n");
    }

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

```
            temp -> next = ptr;
            ptr -> next = head;
        }

        printf("\nnode inserted\n");
    }

}

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

    else
    {   ptr = head;
        while(ptr -> next != head)
            ptr = ptr -> next;
        ptr->next = head->next;
        free(head);
        head = ptr->next;
        printf("\nnode deleted\n");

    }
}
void last_delete()
{
    struct node *ptr, *preptr;
    if(head==NULL)
    {
        printf("\nUNDERFLOW");
```

```
    }
    else if (head ->next == head)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");

    }
    else
    {
        ptr = head;
        while(ptr ->next != head)
        {
            preptr=ptr;
            ptr = ptr->next;
        }
        preptr->next = ptr -> next;
        free(ptr);
        printf("\nnode deleted\n");

    }
}

void search()
{
    struct node *ptr;
    int item,i=0,flag=1;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        if(head ->data == item)
        {
        printf("item found at location %d",i+1);
        flag=0;
```

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

}

void display()
{
    struct node *ptr;
    ptr=head;
    if(head == NULL)
    {
        printf("\nnothing to print");
    }
    else
    {
        printf("\n printing values ... \n");

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

```
        {

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

}
```

Java

```
import java.util.Scanner;

/*  Class Node  */
class Node
{
    protected int data;
    protected Node link;

    /*  Constructor  */
    public Node()
    {
        link = null;
        data = 0;
    }
    /*  Constructor  */
    public Node(int d,Node n)
    {
        data = d;
        link = n;
    }
    /*  Function to set link to next Node  */
    public void setLink(Node n)
    {
        link = n;
    }
    /*  Function to set data to current Node  */
    public void setData(int d)
    {
        data = d;
    }
    /*  Function to get link to next node  */
    public Node getLink()
```

```
    {
        return link;
    }
    /*  Function to get data from current Node  */
    public int getData()
    {
        return data;
    }
}

/* Class linkedList */
class linkedList
{
    protected Node start ;
    protected Node end ;
    public int size ;

    /* Constructor */
    public linkedList()
    {
        start = null;
        end = null;
        size = 0;
    }
    /* Function to check if list is empty */
    public boolean isEmpty()
    {
        return start == null;
    }
    /* Function to get size of the list */
    public int getSize()
    {
        return size;
    }
    /* Function to insert element at the begining */
    public void insertAtStart(int val)
    {
        Node nptr = new Node(val,null);
        nptr.setLink(start);
        if(start == null)
        {
            start = nptr;
            nptr.setLink(start);
            end = start;
```

```
        }
        else
        {
            end.setLink(nptr);
            start = nptr;
        }
        size++ ;
    }
    /* Function to insert element at end */
    public void insertAtEnd(int val)
    {
        Node nptr = new Node(val,null);
        nptr.setLink(start);
        if(start == null)
        {
            start = nptr;
            nptr.setLink(start);
            end = start;
        }
        else
        {
            end.setLink(nptr);
            end = nptr;
        }
        size++ ;
    }
    /* Function to insert element at position */
    public void insertAtPos(int val , int pos)
    {
        Node nptr = new Node(val,null);
        Node ptr = start;
        pos = pos - 1 ;
        for (int i = 1; i < size - 1; i++)
        {
            if (i == pos)
            {
                Node tmp = ptr.getLink() ;
                ptr.setLink( nptr );
                nptr.setLink(tmp);
                break;
            }
            ptr = ptr.getLink();
        }
        size++ ;
```

```
    }
    /* Function to delete element at position */
    public void deleteAtPos(int pos)
    {
        if (size == 1 && pos == 1)
        {
            start = null;
            end = null;
            size = 0;
            return ;
        }
        if (pos == 1)
        {
            start = start.getLink();
            end.setLink(start);
            size--;
            return ;
        }
        if (pos == size)
        {
            Node s = start;
            Node t = start;
            while (s != end)
            {
                t = s;
                s = s.getLink();
            }
            end = t;
            end.setLink(start);
            size --;
            return;
        }
        Node ptr = start;
        pos = pos - 1 ;
        for (int i = 1; i < size - 1; i++)
        {
            if (i == pos)
            {
                Node tmp = ptr.getLink();
                tmp = tmp.getLink();
                ptr.setLink(tmp);
                break;
            }
            ptr = ptr.getLink();
```

```
        }
        size-- ;
    }
    /* Function to display contents */
    public void display()
    {
        System.out.print("\nCircular Singly Linked List = ");
        Node ptr = start;
        if (size == 0)
        {
            System.out.print("empty\n");
            return;
        }
        if (start.getLink() == start)
        {
            System.out.print(start.getData()+ "->"+ptr.getData()+ "\n");
            return;
        }
        System.out.print(start.getData()+ "->");
        ptr = start.getLink();
        while (ptr.getLink() != start)
        {
            System.out.print(ptr.getData()+ "->");
            ptr = ptr.getLink();
        }
        System.out.print(ptr.getData()+ "->");
        ptr = ptr.getLink();
        System.out.print(ptr.getData()+ "\n");
    }
}

/* Class CircularSinglyLinkedList */
public class CircularSinglyLinkedList
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in);
        /* Creating object of linkedList */
        linkedList list = new linkedList();
        System.out.println("Circular Singly Linked List Test\n");
        char ch;
        /*  Perform list operations  */
        do
        {
```

```
        System.out.println("\nCircular Singly Linked List Operations\n");
        System.out.println("1. insert at begining");
        System.out.println("2. insert at end");
        System.out.println("3. insert at position");
        System.out.println("4. delete at position");
        System.out.println("5. check empty");
        System.out.println("6. get size");
        int choice = scan.nextInt();
        switch (choice)
        {
        case 1 :
            System.out.println("Enter integer element to insert");
            list.insertAtStart( scan.nextInt() );
            break;
        case 2 :
            System.out.println("Enter integer element to insert");
            list.insertAtEnd( scan.nextInt() );
            break;
        case 3 :
            System.out.println("Enter integer element to insert");
            int num = scan.nextInt() ;
            System.out.println("Enter position");
            int pos = scan.nextInt() ;
            if (pos <= 1 || pos > list.getSize() )
                System.out.println("Invalid position\n");
            else
                list.insertAtPos(num, pos);
            break;
        case 4 :
            System.out.println("Enter position");
            int p = scan.nextInt() ;
            if (p < 1 || p > list.getSize() )
                System.out.println("Invalid position\n");
            else
                list.deleteAtPos(p);
            break;
        case 5 :
            System.out.println("Empty status = "+ list.isEmpty());
            break;
        case 6 :
            System.out.println("Size = "+ list.getSize() +" \n");
            break;
         default :
            System.out.println("Wrong Entry \n ");
```

```
                break;
            }
            /*  Display List  */
            list.display();
            System.out.println("\nDo you want to continue (Type y or n) \n");
            ch = scan.next().charAt(0);
        } while (ch == 'Y'|| ch == 'y');
    }
}
```

Python

```
class Node:
    def __init__(self, val):
        self.value = val
        self.next = None

class CSLL:
    def __init__(self):
        self.head = None

    def add(self, val):
        if self.empty() is True:
            self.head = Node(val)
            self.head.next = self.head
            print("Head value created")
            return
        opt = int(input(
            "Enter 1 to add a Node at the beginning\nEnter 2 to add a Node at the end\nEnter 3 to add a Node in the "
            "middle::"))
        if opt == 1:
            a = Node(val)
            a.next = self.head
            n = self.head
            while n.next is not self.head:
                n = n.next
            n.next = a
            self.head = a
        elif opt == 2:
            i = self.head
            while i.next is not self.head:
                i = i.next
            i.next = Node(val)
            i.next.next = self.head
```

```
        elif opt == 3:
            pos = int(input("Enter the position ::"))
            i = 1
            n = self.head
            f = n.next
            flag = 0
            while f is not self.head:
                if i == pos:
                    flag = 1
                    break
                f = f.next
                n = n.next
                i = i + 1
            if flag == 1:
                n.next = Node(val)
                n.next.next = f
            else:
                print("Position not found")

    def delete(self):
        if self.empty() is True:
            print("Linked List empty")
            return
        elif self.head.next is self.head:
            self.head = None
            return
        opt = int(input("Enter 1 to delete the beginning element\nEnter 2 to delete the last element\nEnter 3
to "
                        "delete elements in between ::"))
        if opt == 1:
            n = self.head
            while n.next is not self.head:
                n = n.next
            n.next = self.head.next
            self.head = self.head.next
        elif opt == 2:
            n = self.head
            while n.next.next is not self.head:
                n = n.next
            n.next = self.head
        else:
            op = int(input("Enter 1 to delete by position\nEnter 2 to delete by value ::"))
            if op == 1:
                pos = int(input("Enter the position ::"))
```

```
            i = 1
            n = self.head
            f = self.head.next
            flag = 0
            while f.next is not self.head:
                if i == pos:
                    flag = 1
                    break
                i = i + 1
                n = n.next
                f = f.next
            if flag == 1:
                n.next = f.next
            else:
                print("Position not found")
                return
        else:
            val = int(input("Enter the value you want to delete ::"))
            n = self.head
            f = self.head.next
            flag = 0
            while f.next is not self.head:
                if f.value == val:
                    flag = 1
                    break
                f = f.next
                n = n.next
            if flag == 1:
                n.next = f.next
            else:
                print("Value not found")
                return

    def clear(self):
        self.head = None
        print("Linked List cleared")

    def empty(self):
        if self.head is None:
            return True
        else:
            return False

    def display(self):
```

```
        if self.empty() is True:
            print("Linked List empty")
            return
        print("THE LINKED LIST")
        print(self.head.value, " <== HEAD")
        n = self.head.next
        while n is not self.head:
            print(n.value)
            n = n.next
        print("Linked List ends")

LL = CSLL()
while True:
    option = int(input("Enter 1 to add an element\nEnter 2 to delete an element\nEnter 3 to clear the Linked
"
                       "List\nEnter 4 to display the Linked List\nEnter 5 to exit ::"))
    if option == 1:
        value = int(input("Enter the value you want to add ::"))
        LL.add(value)
        continue
    elif option == 2:
        LL.delete()
        continue
    elif option == 3:
        LL.clear()
        continue
    elif option == 4:
        LL.display()
        continue
    elif option == 5:
        print("Goodbye")
        break
    elif option == 6:
        print(LL.empty())
    else:
        print("Wrong option")
```

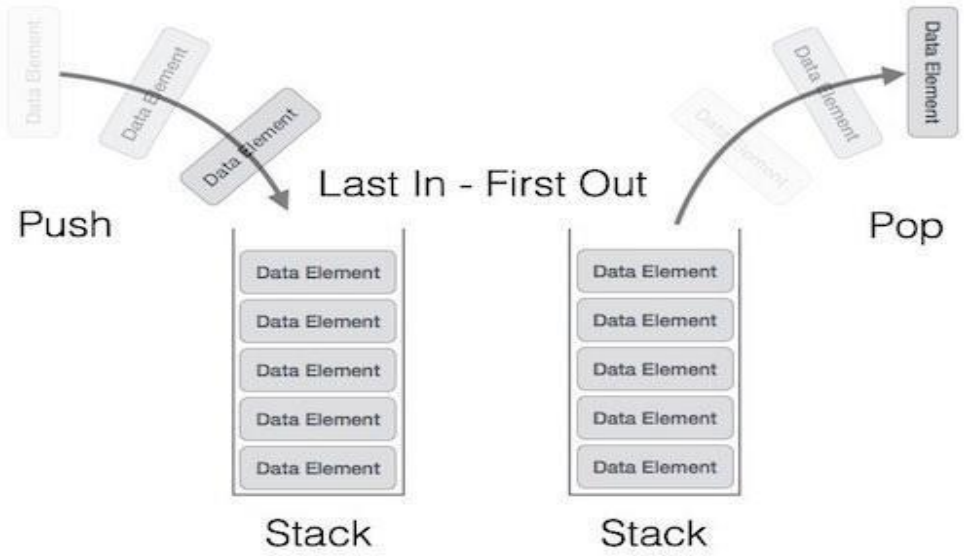
Stack

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack.

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

Stack Representation

The following diagram depicts a stack and its operations −



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

Basic Operations

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

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

When data is PUSHed onto the stack.

To use a stack efficiently, we need to check the status of the 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 toppointer provides the top value of the stack without actually removing it.

First we should learn about procedures to support stack functions –

peek()

Algorithm of peek() function –

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

Implementation of peek() function in C programming language –

Example

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

}
```

isfull()

Algorithm of isfull() function –

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif


end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
```

```
  if(top == MAXSIZE)
     return true;
  else
     return false;

}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty

  if top less than 1
     return true
  else
     return false
  endif


end procedure
```

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

Example

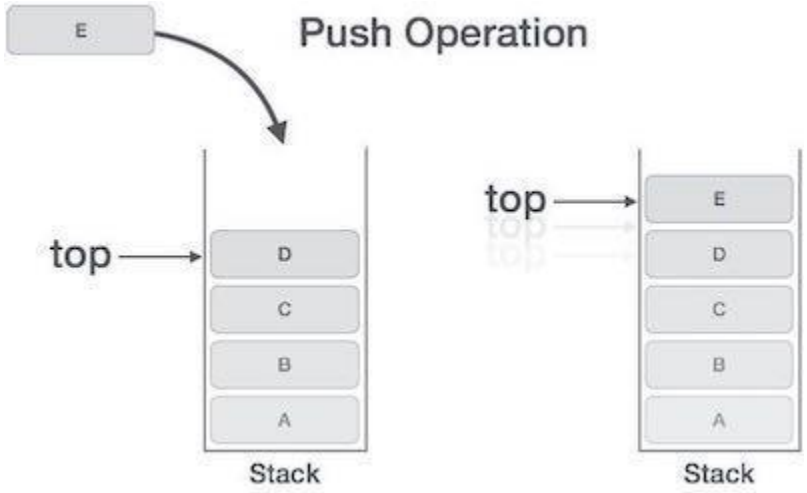
```
bool isempty() {
  if(top == -1)
     return true;
  else
     return false;

}
```

Push Operation

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

- Step 1 − Checks if the stack is full.

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



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

Algorithm for PUSH Operation

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

```
begin procedure push: stack, data

  if stack is full
    return null
  endif

  top ← top + 1
  stack[top] ← data

end procedure
```

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

Example

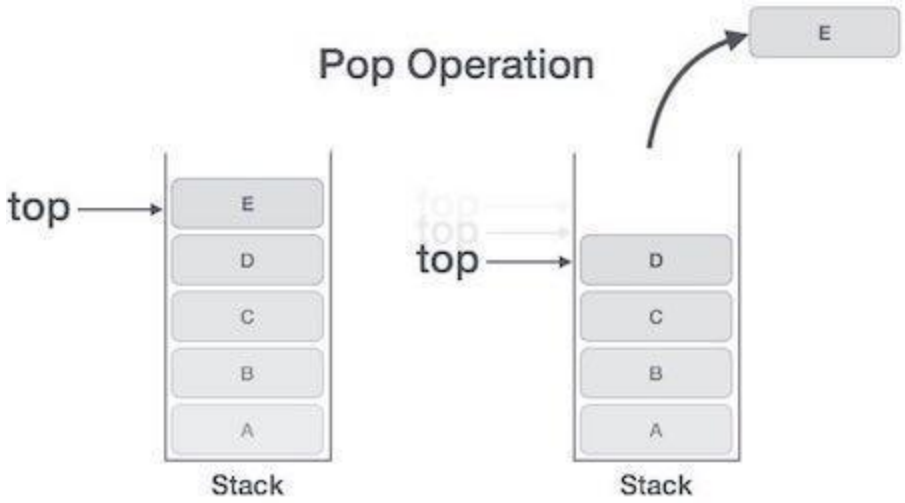
```
void push(int data) {
  if(!isFull()) {
     top = top + 1;
     stack[top] = data;
  } else {
     printf("Could not insert data, Stack is full.\n");
  }

}
```

Pop Operation

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

A Pop operation may involve the following steps –

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



Algorithm for Pop Operation

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

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

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


end procedure
```

Implementation of this algorithm in C, is as follows –

Example

```
int pop(int data) {

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

```
    }

}
```

Program:

Java

```
// Java code for stack implementation

import java.io.*;
import java.util.*;

class Test
{
        // Pushing element on the top of the stack
        static void stack_push(Stack<Integer> stack)
        {
                for(int i = 0; i < 5; i++)
                {
                        stack.push(i);
                }
        }

        // Popping element from the top of the stack
        static void stack_pop(Stack<Integer> stack)
        {
                System.out.println("Pop Operation:");

                for(int i = 0; i < 5; i++)
                {
                        Integer y = (Integer) stack.pop();
                        System.out.println(y);
                }
        }

        // Displaying element on the top of the stack
        static void stack_peek(Stack<Integer> stack)
        {
                Integer element = (Integer) stack.peek();
                System.out.println("Element on stack top: " + element);
        }

        // Searching element in the stack
        static void stack_search(Stack<Integer> stack, int element)
```

```
        {
                Integer pos = (Integer) stack.search(element);

                if(pos == -1)
                        System.out.println("Element not found");
                else
                        System.out.println("Element is found at position: " + pos);
        }


        public static void main (String[] args)
        {
                Stack<Integer> stack = new Stack<Integer>();

                stack_push(stack);
                stack_pop(stack);
                stack_push(stack);
                stack_peek(stack);
                stack_search(stack, 2);
                stack_search(stack, 6);
        }
}
```

Python

```
# Python program to
# demonstrate stack implementation
# using list


stack = []

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)

# pop() fucntion to pop
# element from stack in
# LIFO order
print('\nElements poped from stack:')
```

```
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are poped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

Expression Parsing

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

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

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

Infix Notation

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

Prefix Notation

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

Postfix Notation

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

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

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

Parsing Expressions

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

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

Precedence

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

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

Associativity

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

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

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

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

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

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation –

Step 1 – scan the expression from left to right
Step 2 – if it is an operand push it to stack
Step 3 – if it is an operator pull operand from stack and perform operation
Step 4 – store the output of step 3, back to stack
Step 5 – scan the expression until all operands are consumed
Step 6 – pop the stack and perform operation

Queue

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 in C programming language –

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

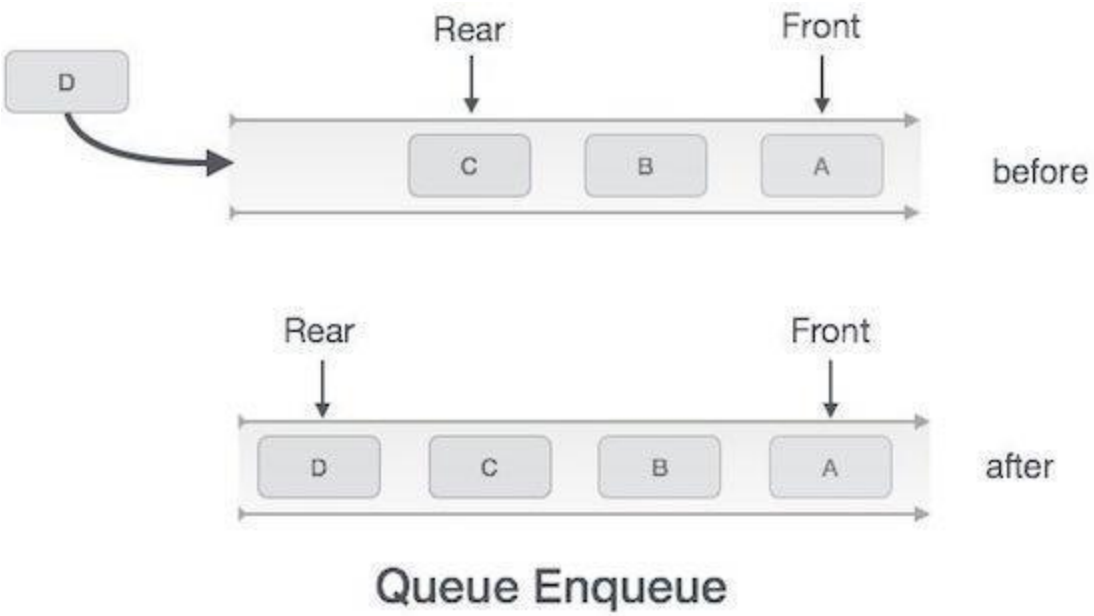
```
bool isempty() {
  if(front < 0 || front > rear)
     return true;
  else
     return false;

}
```

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.



Queue Enqueue

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

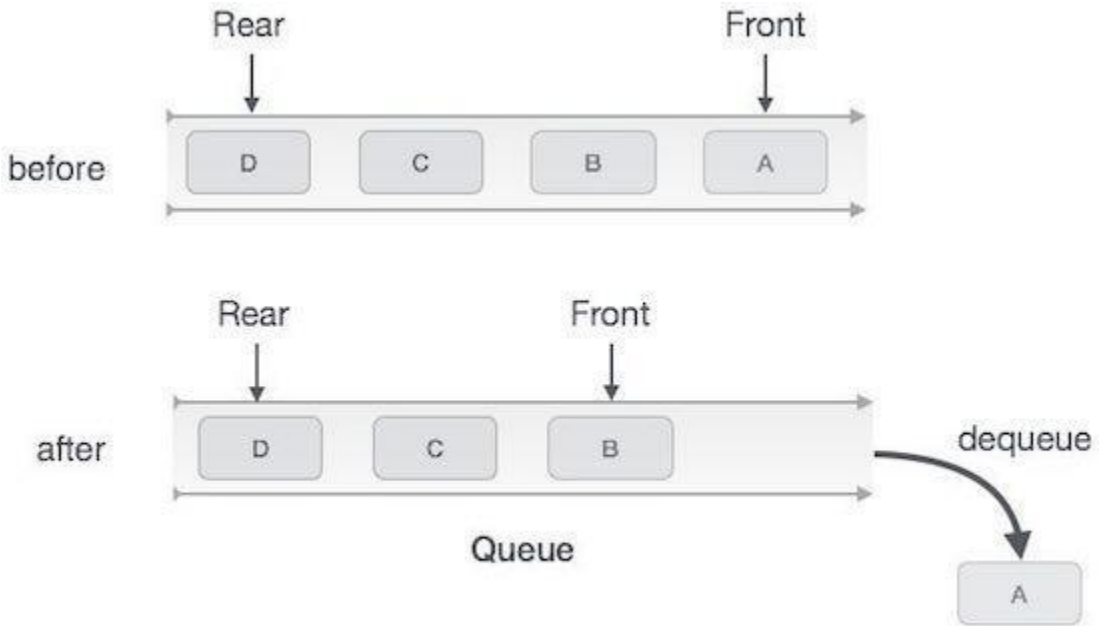
```
int enqueue(int data)
  if(isfull())
```

```
    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.



Queue Dequeue

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;

}
```

Programs:

Java

```
public class Queue {
  int SIZE = 5;
  int items[] = new int[SIZE];
  int front, rear;

  Queue() {
    front = -1;
    rear = -1;
  }

  // check if the queue is full
  boolean isFull() {
    if (front == 0 && rear == SIZE - 1) {
      return true;
    }
    return false;
  }
```

```
  // check if the queue is empty
  boolean isEmpty() {
    if (front == -1)
      return true;
    else
      return false;
  }

  // insert elements to the queue
  void enQueue(int element) {

    // if queue is full
    if (isFull()) {
      System.out.println("Queue is full");
    }
    else {
      if (front == -1) {
        // mark front denote first element of queue
        front = 0;
      }

      rear++;
      // insert element at the rear
      items[rear] = element;
      System.out.println("Insert " + element);
    }
  }

  // delete element from the queue
  int deQueue() {
    int element;

    // if queue is empty
    if (isEmpty()) {
      System.out.println("Queue is empty");
      return (-1);
    }
    else {
      // remove element from the front of queue
      element = items[front];

      // if the queue has only one element
      if (front >= rear) {
        front = -1;
```

```
      rear = -1;
    }
    else {
      // mark next element as the front
      front++;
    }
    System.out.println( element + " Deleted");
    return (element);
  }
 }

 // display element of the queue
 void display() {
  int i;
  if (isEmpty()) {
    System.out.println("Empty Queue");
  }
  else {
    // display the front of the queue
    System.out.println("\nFront index-> " + front);

    // display element of the queue
    System.out.println("Items -> ");
    for (i = front; i <= rear; i++)
      System.out.print(items[i] + "  ");

    // display the rear of the queue
    System.out.println("\nRear index-> " + rear);
  }
 }

 public static void main(String[] args) {

  // create an object of Queue class
  Queue q = new Queue();

  // try to delete element from the queue
  // currently queue is empty
  // so deletion is not possible
  q.deQueue();

  // insert elements to the queue
  for(int i = 1; i < 6; i ++) {
    q.enQueue(i);
```

```
    }

    // 6th element can't be added to queue because queue is full
    q.enQueue(6);

    q.display();

    // deQueue removes element entered first i.e. 1
    q.deQueue();

    // Now we have just 4 elements
    q.display();

  }
}
```

Python

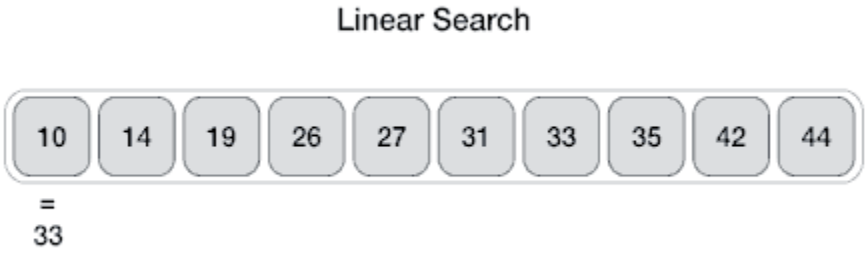
```
# Queue implementation in Python

class Queue:

    def __init__(self):
        self.queue = []

    # Add an element
    def enqueue(self, item):
        self.queue.append(item)

    # Remove an element
    def dequeue(self):
        if len(self.queue) < 1:
            return None
        return self.queue.pop(0)

    # Display  the queue
    def display(self):
        print(self.queue)

    def size(self):
        return len(self.queue)


q = Queue()
q.enqueue(1)
```

```
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)

q.display()

q.dequeue()

print("After removing an element")
q.display()
```

Searching Techniques

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

Pseudocode

```
procedure linear_search (list, value)
   for each item in the list
      if match item == value
         return the item's location
      end if
   end for
end procedure
```

Program:
C

```
#include <stdio.h>

#define MAX 20

// array of items on which linear search will be conducted.
int intArray[MAX] = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};

void printline(int count) {
   int i;

   for(i = 0;i <count-1;i++) {
      printf("=");
   }

   printf("=\n");
}

// this method makes a linear search.
int find(int data) {
```

```
   int comparisons = 0;
   int index = -1;
   int i;

   // navigate through all items
   for(i = 0;i<MAX;i++) {

      // count the comparisons made
      comparisons++;

      // if data found, break the loop
      if(data == intArray[i]) {
         index = i;
         break;
      }
   }

   printf("Total comparisons made: %d", comparisons);
   return index;
}

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i<MAX;i++) {
      printf("%d ",intArray[i]);
   }

   printf("]\n");
}

void main() {
   printf("Input Array: ");
   display();
   printline(50);

   //find location of 1
```

```
  int location = find(55);

  // if element was found
  if(location != -1)
     printf("\nElement found at location: %d" ,(location+1));
  else
     printf("Element not found.");

}
```

Java

```
public class LinearSearchExample{
public static int linearSearch(int[] arr, int key){
       for(int i=0;i<arr.length;i++){
           if(arr[i] == key){
               return i;
           }
       }
       return -1;
   }
   public static void main(String a[]){
       int[] a1= {10,20,30,50,70,90};
       int key = 50;
       System.out.println(key+" is found at index: "+linearSearch(a1, key));
   }
}
```

Python

```
def linearsearch(arr, x):
   for i in range(len(arr)):
      if arr[i] == x:
         return i
   return -1
arr = ['t','u','t','o','r','i','a','l']
x = 'a'
print("element found at index "+str(linearsearch(arr,x)))
```

Binary Search

Binary search is a fast search algorithm with run-time complexity of O(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.

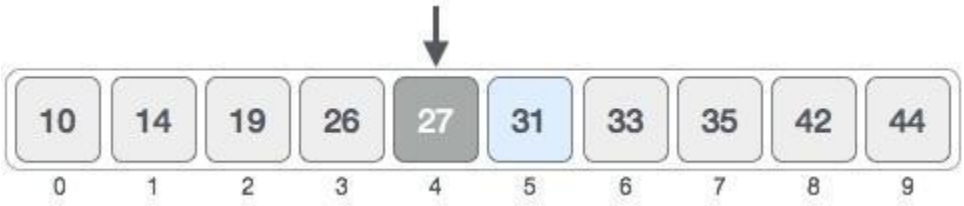
How Binary Search Works?

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.

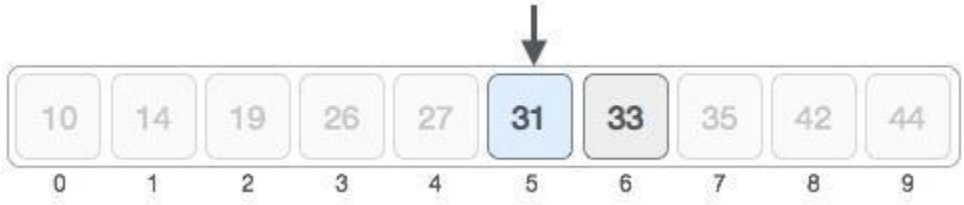
We change our low to mid + 1 and find the new mid value again.

```
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.

Pseudocode

The pseudocode of binary search algorithms should look like this –

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched

   Set lowerBound = 1
   Set upperBound = n
```

```
  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.

    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while


end procedure
```

Program:

C

```
#include <stdio.h>

#define MAX 20

// array of items on which linear search will be conducted.
int intArray[MAX] = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};

void printline(int count) {
  int i;

  for(i = 0;i <count-1;i++) {
    printf("=");
  }

  printf("=\n");
}

int find(int data) {
  int lowerBound = 0;
  int upperBound = MAX -1;
```

```
   int midPoint = -1;
   int comparisons = 0;
   int index = -1;

   while(lowerBound <= upperBound) {
      printf("Comparison %d\n" , (comparisons +1) );
      printf("lowerBound : %d, intArray[%d] = %d\n",lowerBound,lowerBound,
         intArray[lowerBound]);
      printf("upperBound : %d, intArray[%d] = %d\n",upperBound,upperBound,
         intArray[upperBound]);
      comparisons++;

      // compute the mid point
      // midPoint = (lowerBound + upperBound) / 2;
      midPoint = lowerBound + (upperBound - lowerBound) / 2;

      // data found
      if(intArray[midPoint] == data) {
         index = midPoint;
         break;
      } else {
         // if data is larger
         if(intArray[midPoint] < data) {
            // data is in upper half
            lowerBound = midPoint + 1;
         }
         // data is smaller
         else {
            // data is in lower half
            upperBound = midPoint -1;
         }
      }
   }
   printf("Total comparisons made: %d" , comparisons);
   return index;
}

void display() {
   int i;
   printf("[");
```

```
   // navigate through all items
   for(i = 0;i<MAX;i++) {
      printf("%d ",intArray[i]);
   }

   printf("]\n");
}

void main() {
   printf("Input Array: ");
   display();
   printline(50);

   //find location of 1
   int location = find(55);

   // if element was found
   if(location != -1)
      printf("\nElement found at location: %d" ,(location+1));
   else
      printf("\nElement not found.");

}
```

Java

```
class BinarySearchExample{
 public static void binarySearch(int arr[], int first, int last, int key){
   int mid = (first + last)/2;
   while( first <= last ){
      if ( arr[mid] < key ){
        first = mid + 1;
      }else if ( arr[mid] == key ){
        System.out.println("Element is found at index: " + mid);
        break;
      }else{
         last = mid - 1;
      }
      mid = (first + last)/2;
   }
   if ( first > last ){
      System.out.println("Element is not found!");
```

```
  }
 }
 public static void main(String args[]){
       int arr[] = {10,20,30,40,50};
       int key = 30;
       int last=arr.length-1;
       binarySearch(arr,0,last,key);
 }
}
```

Python

```
# Returns index of x in arr if present, else -1
def binary_search(arr, low, high, x):

        # Check base case
        if high >= low:

                mid = (high + low) // 2

                # If element is present at the middle itself
                if arr[mid] == x:
                        return mid

                # If element is smaller than mid, then it can only
                # be present in left subarray
                elif arr[mid] > x:
                        return binary_search(arr, low, mid - 1, x)

                # Else the element can only be present in right subarray
                else:
                        return binary_search(arr, mid + 1, high, x)

        else:
                # Element is not present in the array
                return -1

# Test array
arr = [ 2, 3, 4, 10, 40 ]
x = 10

# Function call
result = binary_search(arr, 0, len(arr)-1, x)
```

```
if result != -1:
        print("Element is present at index", str(result))
else:
        print("Element is not present in array")
```

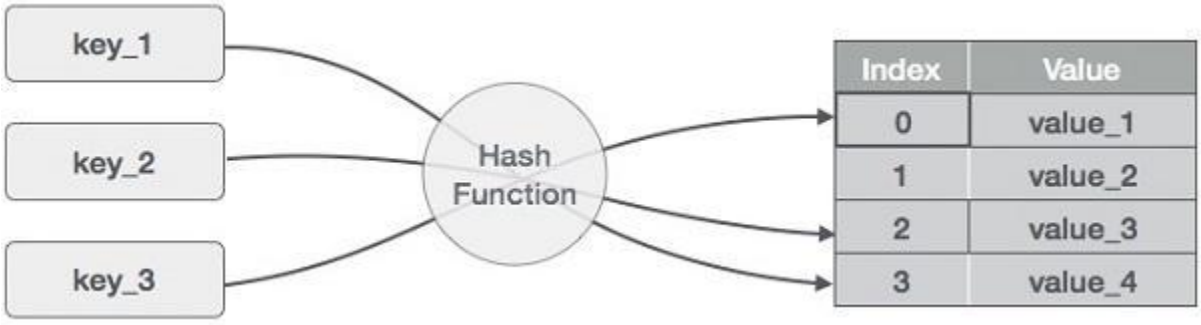
Hash Table

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

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

Hashing

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



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

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

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

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are the basic primary operations of a hash table.

- `Search – Searches an element in a hash table.`
- `Insert – inserts an element in a hash table.`
- `delete – Deletes an element from a hash table.`

DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {
   int data;
   int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
   return key % SIZE;

}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Example

```
struct DataItem *search(int key) {
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL) {

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];
      //go to next cell
      ++hashIndex;
      //wrap around the table
      hashIndex %= SIZE;
   }
   return NULL;

}
```

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Example

```
void insert(int key,int data) {
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;

}
```

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Example

```
struct DataItem* delete(struct DataItem* item) {
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] !=NULL) {

```

```
    if(hashArray[hashIndex]->key == key) {
        struct DataItem* temp = hashArray[hashIndex];

        //assign a dummy item at deleted position
        hashArray[hashIndex] = dummyItem;
        return temp;
    }

    //go to next cell
    ++hashIndex;

    //wrap around the table
    hashIndex %= SIZE;
  }

  return NULL;

}
```

Program:

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

#define SIZE 20

struct DataItem {
  int data;
  int key;
};

struct DataItem* hashArray[SIZE];
struct DataItem* dummyItem;
struct DataItem* item;

int hashCode(int key) {
  return key % SIZE;
}
```

```
struct DataItem *search(int key) {
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL) {

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}

void insert(int key,int data) {

   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;
}
```

```
struct DataItem* delete(struct DataItem* item) {
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL) {

      if(hashArray[hashIndex]->key == key) {
         struct DataItem* temp = hashArray[hashIndex];

         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}

void display() {
   int i = 0;

   for(i = 0; i<SIZE; i++) {

      if(hashArray[i] != NULL)
         printf(" (%d,%d)",hashArray[i]->key,hashArray[i]->data);
      else
         printf(" ~~ ");
   }

   printf("\n");
```

```
}

int main() {
   dummyItem = (struct DataItem*) malloc(sizeof(struct DataItem));
   dummyItem->data = -1;
   dummyItem->key = -1;

   insert(1, 20);
   insert(2, 70);
   insert(42, 80);
   insert(4, 25);
   insert(12, 44);
   insert(14, 32);
   insert(17, 11);
   insert(13, 78);
   insert(37, 97);

   display();
   item = search(37);
   if(item != NULL) {
      printf("Element found: %d\n", item->data);
   } else {
      printf("Element not found\n");
   }
   delete(item);
   item = search(37);
   if(item != NULL) {
      printf("Element found: %d\n", item->data);
   } else {
      printf("Element not found\n");
   }

}
```

Sorting Techniques

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 $O(n^2)$ where n is the number of items.

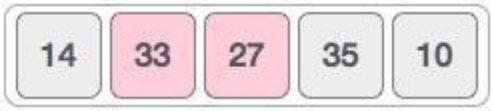
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

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

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

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



The new array should look like this –

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

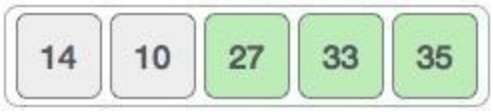
Then we move to the next two values, 35 and 10.

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

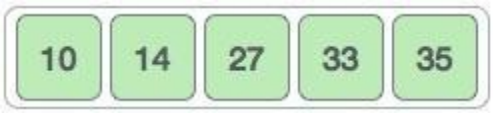
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

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 −

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



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



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 swapfunction 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
```

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable swapped which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

`Pseudocode of BubbleSort algorithm can be written as follows –`

```
procedure bubbleSort( list : array of items )

   loop = list.count;

   for i = 0 to loop-1 do:
      swapped = false

      for j = 0 to loop-1 do:

         /* compare the adjacent elements */
         if list[j] > list[j+1] then
            /* swap them */
            swap( list[j], list[j+1] )
            swapped = true
         end if

      end for

      /*if no number was swapped that means
      array is sorted now, break the loop.*/

      if(not swapped) then
         break
      end if

   end for
```

```
end procedure return list
```

Implementation

One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

Program:

C

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

#define MAX 10

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

void display() {
  int i;
  printf("[");

  // navigate through all items
  for(i = 0; i < MAX; i++) {
    printf("%d ",list[i]);
  }

  printf("]\n");
}

void bubbleSort() {
  int temp;
  int i,j;

  bool swapped = false;

  // loop through all numbers
  for(i = 0; i < MAX-1; i++) {
    swapped = false;

    // loop through numbers falling ahead
    for(j = 0; j < MAX-1-i; j++) {
      printf("    Items compared: [ %d, %d ] ", list[j],list[j+1]);
```

```
      // check if next number is lesser than current no
      //   swap the numbers.
      //  (Bubble up the highest number)

      if(list[j] > list[j+1]) {
         temp = list[j];
         list[j] = list[j+1];
         list[j+1] = temp;

         swapped = true;
         printf(" => swapped [%d, %d]\n",list[j],list[j+1]);
      } else {
         printf(" => not swapped\n");
      }

   }

   // if no number was swapped that means
   //   array is sorted now, break the loop.
   if(!swapped) {
      break;
   }

   printf("Iteration %d#: ",(i+1));
   display();
  }

}

void main() {
  printf("Input Array: ");
  display();
  printf("\n");

  bubbleSort();
  printf("\nOutput Array: ");
  display();

}
```

Java

```
public class BubbleSortExample {
   static void bubbleSort(int[] arr) {
      int n = arr.length;
      int temp = 0;
       for(int i=0; i < n; i++){
               for(int j=1; j < (n-i); j++){
                       if(arr[j-1] > arr[j]){
```

```
                        //swap elements
                        temp = arr[j-1];
                        arr[j-1] = arr[j];
                        arr[j] = temp;
                    }

                }
        }

    }
    public static void main(String[] args) {
            int arr[] ={3,60,35,2,45,320,5};

            System.out.println("Array Before Bubble Sort");
            for(int i=0; i < arr.length; i++){
                    System.out.print(arr[i] + " ");
            }
            System.out.println();

            bubbleSort(arr);//sorting array elements using bubble sort

            System.out.println("Array After Bubble Sort");
            for(int i=0; i < arr.length; i++){
                    System.out.print(arr[i] + " ");
            }

        }
}
```

Python

```
# Python program for implementation of Bubble Sort

def bubbleSort(arr):
        n = len(arr)

        # Traverse through all array elements
        for i in range(n-1):
        # range(n) also work but outer loop will repeat one time more than needed.

                # Last i elements are already in place
                for j in range(0, n-i-1):

                        # traverse the array from 0 to n-i-1
                        # Swap if the element found is greater
```

```
                    # than the next element
                    if arr[j] > arr[j + 1] :
                            arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print ("Sorted array is:")
for i in range(len(arr)):
        print ("% d" % arr[i]),
```

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 $O(n^2)$, where n is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

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

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

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.

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

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

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

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

Pseudocode

```
procedure insertionSort( A : array of items )
  int holePosition
  int valueToInsert

  for i = 1 to length(A) inclusive do:

    /* select value to be inserted */
    valueToInsert = A[i]
    holePosition = i

    /*locate hole position for the element to be inserted */

    while holePosition > 0 and A[holePosition-1] > valueToInsert do:
      A[holePosition] = A[holePosition-1]
      holePosition = holePosition -1
    end while
```

```
      /* insert the number at hole position */
      A[holePosition] = valueToInsert

   end for

end procedure
```

Program:

C

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

#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count) {
   int i;

   for(i = 0;i < count-1;i++) {
      printf("=");
   }

   printf("=\n");
}

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i < MAX;i++) {
      printf("%d ",intArray[i]);
   }

   printf("]\n");
}
```

```
void insertionSort() {

   int valueToInsert;
   int holePosition;
   int i;

   // loop through all numbers
   for(i = 1; i < MAX; i++) {

      // select a value to be inserted.
      valueToInsert = intArray[i];

      // select the hole position where number is to be inserted
      holePosition = i;

      // check if previous no. is larger than value to be inserted
      while (holePosition > 0 && intArray[holePosition-1] > valueToInsert) {
         intArray[holePosition] = intArray[holePosition-1];
         holePosition--;
         printf(" item moved : %d\n" , intArray[holePosition]);
      }

      if(holePosition != i) {
         printf(" item inserted : %d, at position : %d\n" , valueToInsert,holePosition);
         // insert the number at hole position
         intArray[holePosition] = valueToInsert;
      }

      printf("Iteration %d#:",i);
      display();

   }
}

void main() {
   printf("Input Array: ");
   display();
   printline(50);
   insertionSort();
```

```
   printf("Output Array: ");
   display();
   printline(50);

}
```

Java

```
public class InsertionSortExample {
   public static void insertionSort(int array[]) {
      int n = array.length;
      for (int j = 1; j < n; j++) {
         int key = array[j];
         int i = j-1;
         while ( (i > -1) && ( array [i] > key ) ) {
            array [i+1] = array [i];
            i--;
         }
         array[i+1] = key;
      }
   }

   public static void main(String a[]){
      int[] arr1 = {9,14,3,2,43,11,58,22};
      System.out.println("Before Insertion Sort");
      for(int i:arr1){
         System.out.print(i+" ");
      }
      System.out.println();

      insertionSort(arr1);//sorting array using insertion sort

      System.out.println("After Insertion Sort");
      for(int i:arr1){
         System.out.print(i+" ");
      }
   }
}
```

Python

```
# Python program for implementation of Insertion Sort

# Function to do insertion sort
def insertionSort(arr):
```

```
        # Traverse through 1 to len(arr)
        for i in range(1, len(arr)):

                key = arr[i]

                # Move elements of arr[0..i-1], that are
                # greater than key, to one position ahead
                # of their current position
                j = i-1
                while j >=0 and key < arr[j] :
                                arr[j+1] = arr[j]
                                j -= 1
                arr[j+1] = key


# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
        print ("%d" %arr[i])
```

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 $O(n^2)$, where n is the number of items.

How Selection Sort Works?

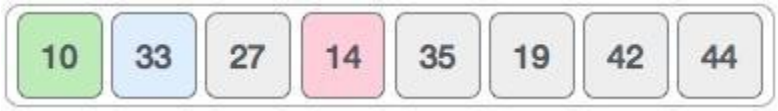
Consider the following depicted array as an example.

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

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

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

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



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

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

Following is a pictorial depiction of the entire sorting process −

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

Pseudocode

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end procedure
```

Program:

C

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

#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count) {
   int i;

   for(i = 0;i < count-1;i++) {
      printf("=");
   }
```

```
   printf("=\n");
}

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i < MAX;i++) {
      printf("%d ", intArray[i]);
   }

   printf("]\n");
}

void selectionSort() {
   int indexMin,i,j;

   // loop through all numbers
   for(i = 0; i < MAX-1; i++) {

      // set current element as minimum
      indexMin = i;

      // check the element to be minimum
      for(j = i+1;j < MAX;j++) {
         if(intArray[j] < intArray[indexMin]) {
            indexMin = j;
         }
      }

      if(indexMin != i) {
         printf("Items swapped: [ %d, %d ]\n" , intArray[i], intArray[indexMin]);

         // swap the numbers
         int temp = intArray[indexMin];
         intArray[indexMin] = intArray[i];
         intArray[i] = temp;
      }
```

```
    printf("Iteration %d#:",(i+1));
    display();
  }
}

void main() {
  printf("Input Array: ");
  display();
  printline(50);
  selectionSort();
  printf("Output Array: ");
  display();
  printline(50);

}
```

Java

```
public class SelectionSortExample {
   public static void selectionSort(int[] arr){
      for (int i = 0; i < arr.length - 1; i++)
      {
         int index = i;
         for (int j = i + 1; j < arr.length; j++){
            if (arr[j] < arr[index]){
               index = j;//searching for lowest index
            }
         }
         int smallerNumber = arr[index];
         arr[index] = arr[i];
         arr[i] = smallerNumber;
      }
   }

   public static void main(String a[]){
      int[] arr1 = {9,14,3,2,43,11,58,22};
      System.out.println("Before Selection Sort");
      for(int i:arr1){
         System.out.print(i+" ");
      }
      System.out.println();
```

```
        selectionSort(arr1);//sorting array using selection sort

        System.out.println("After Selection Sort");
        for(int i:arr1){
            System.out.print(i+" ");
        }
    }
}
```

Python

```
# Python program for implementation of Selection
# Sort
import sys
A = [64, 25, 12, 22, 11]
# Traverse through all array elements
for i in range(len(A)):
        # Find the minimum element in remaining
        # unsorted array
        min_idx = i
        for j in range(i+1, len(A)):
                if A[min_idx] > A[j]:
                        min_idx = j
        # Swap the found minimum element with
        # the first element
        A[i], A[min_idx] = A[min_idx], A[i]
# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
        print("%d" %A[i]),
```

Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(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 –

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

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

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

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

We first compare the 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.

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

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

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.

Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
```

```
    end if
  end while

  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while

  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while

  return c

end procedure
```

Program:

C

```
#include <stdio.h>

#define max 10

int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];

void merging(int low, int mid, int high) {
  int l1, l2, i;

  for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
    if(a[l1] <= a[l2])
      b[i] = a[l1++];
    else
      b[i] = a[l2++];
  }

  while(l1 <= mid)
    b[i++] = a[l1++];
```

```
  while(l2 <= high)
    b[i++] = a[l2++];

  for(i = low; i <= high; i++)
    a[i] = b[i];
}

void sort(int low, int high) {
  int mid;

  if(low < high) {
    mid = (low + high) / 2;
    sort(low, mid);
    sort(mid+1, high);
    merging(low, mid, high);
  } else {
    return;
  }
}

int main() {
  int i;

  printf("List before sorting\n");

  for(i = 0; i <= max; i++)
    printf("%d ", a[i]);

  sort(0, max);

  printf("\nList after sorting\n");

  for(i = 0; i <= max; i++)
    printf("%d ", a[i]);

}
```

Java

```
/* Java program for Merge Sort */
class MergeSort
{
```

```
	// Merges two subarrays of arr[].
	// First subarray is arr[l..m]
	// Second subarray is arr[m+1..r]
	void merge(int arr[], int l, int m, int r)
	{
		// Find sizes of two subarrays to be merged
		int n1 = m - l + 1;
		int n2 = r - m;

		/* Create temp arrays */
		int L[] = new int [n1];
		int R[] = new int [n2];

		/*Copy data to temp arrays*/
		for (int i=0; i<n1; ++i)
			L[i] = arr[l + i];
		for (int j=0; j<n2; ++j)
			R[j] = arr[m + 1+ j];


		/* Merge the temp arrays */

		// Initial indexes of first and second subarrays
		int i = 0, j = 0;

		// Initial index of merged subarray array
		int k = l;
		while (i < n1 && j < n2)
		{
			if (L[i] <= R[j])
			{
				arr[k] = L[i];
				i++;
			}
			else
			{
				arr[k] = R[j];
				j++;
			}
			k++;
		}
```

```
        /* Copy remaining elements of L[] if any */
        while (i < n1)
        {
            arr[k] = L[i];
            i++;
            k++;
        }

        /* Copy remaining elements of R[] if any */
        while (j < n2)
        {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

    // Main function that sorts arr[l..r] using
    // merge()
    void sort(int arr[], int l, int r)
    {
        if (l < r)
        {
            // Find the middle point
            int m = (l+r)/2;

            // Sort first and second halves
            sort(arr, l, m);
            sort(arr , m+1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
```

```
        }

        // Driver method
        public static void main(String args[])
        {
                int arr[] = {12, 11, 13, 5, 6, 7};

                System.out.println("Given Array");
                printArray(arr);

                MergeSort ob = new MergeSort();
                ob.sort(arr, 0, arr.length-1);

                System.out.println("\nSorted array");
                printArray(arr);
        }
}
```

Python

```
# Python program for implementation of MergeSort

# Merges two subarrays of arr[].
# First subarray is arr[l..m]
# Second subarray is arr[m+1..r]
def merge(arr, l, m, r):
        n1 = m - l + 1
        n2 = r- m

        # create temp arrays
        L = [0] * (n1)
        R = [0] * (n2)

        # Copy data to temp arrays L[] and R[]
        for i in range(0 , n1):
                L[i] = arr[l + i]

        for j in range(0 , n2):
                R[j] = arr[m + 1 + j]

        # Merge the temp arrays back into arr[l..r]
        i = 0     # Initial index of first subarray
        j = 0     # Initial index of second subarray
```

```
        k = l      # Initial index of merged subarray

        while i < n1 and j < n2 :
                if L[i] <= R[j]:
                        arr[k] = L[i]
                        i += 1
                else:
                        arr[k] = R[j]
                        j += 1
                k += 1

        # Copy the remaining elements of L[], if there
        # are any
        while i < n1:
                arr[k] = L[i]
                i += 1
                k += 1

        # Copy the remaining elements of R[], if there
        # are any
        while j < n2:
                arr[k] = R[j]
                j += 1
                k += 1

# l is for left index and r is right index of the
# sub-array of arr to be sorted
def mergeSort(arr,l,r):
        if l < r:

                # Same as (l+r)//2, but avoids overflow for
                # large l and h
                m = (l+(r-1))//2

                # Sort first and second halves
                mergeSort(arr, l, m)
                mergeSort(arr, m+1, r)
                merge(arr, l, m, r)


# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
```

```
n = len(arr)
print ("Given array is")
for i in range(n):
        print ("%d" %arr[i]),

mergeSort(arr,0,n-1)
print ("\n\nSorted array is")
for i in range(n):
        print ("%d" %arr[i]),
```

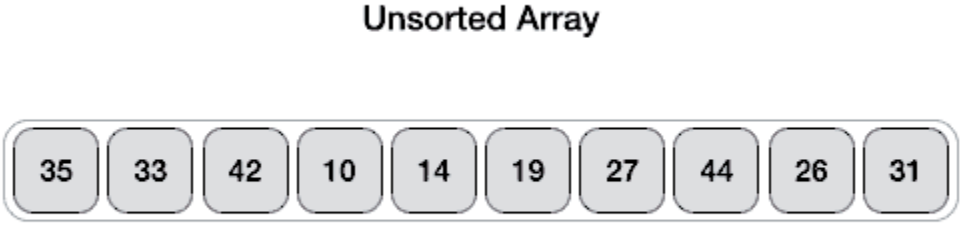
Quick Sort

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 $O(n^2)$, where n is the number of items.

Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

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

Step 1 – Choose the highest index value has pivot

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

Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as –

```
function partitionFunc(left, right, pivot)
  leftPointer = left
  rightPointer = right - 1

  while True do
    while A[++leftPointer] < pivot do
      //do-nothing
    end while

    while rightPointer > 0 && A[--rightPointer] > pivot do
      //do-nothing
    end while

    if leftPointer >= rightPointer
      break
    else
      swap leftPointer,rightPointer
    end if

  end while

  swap leftPointer,right
  return leftPointer

end function
```

Program:

C

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 7
int intArray[MAX] = {4,6,3,2,1,9,7};
void printline(int count) {
   int i;
   for(i = 0;i < count-1;i++) {
      printf("=");
   }
   printf("=\n");
}
void display() {
   int i;
   printf("[");
   // navigate through all items
   for(i = 0;i < MAX;i++) {
      printf("%d ",intArray[i]);
   }
   printf("]\n");
}
void swap(int num1, int num2) {
   int temp = intArray[num1];
   intArray[num1] = intArray[num2];
   intArray[num2] = temp;
}
int partition(int left, int right, int pivot) {
   int leftPointer = left -1;
   int rightPointer = right;
   while(true) {
      while(intArray[++leftPointer] < pivot) {
         //do nothing
      }
      while(rightPointer > 0 && intArray[--rightPointer] > pivot) {
         //do nothing
      }
      if(leftPointer >= rightPointer) {
         break;
      } else {
         printf(" item swapped :%d,%d\n", intArray[leftPointer],intArray[rightPointer]);
```

```
      swap(leftPointer,rightPointer);
    }
  }
  printf(" pivot swapped :%d,%d\n", intArray[leftPointer],intArray[right]);
  swap(leftPointer,right);
  printf("Updated Array: ");
  display();
  return leftPointer;
}
void quickSort(int left, int right) {
  if(right-left <= 0) {
    return;
  } else {
    int pivot = intArray[right];
    int partitionPoint = partition(left, right, pivot);
    quickSort(left,partitionPoint-1);
    quickSort(partitionPoint+1,right);
  }
}
int main() {
  printf("Input Array: ");
  display();
  printline(50);
  quickSort(0,MAX-1);
  printf("Output Array: ");
  display();
  printline(50);
}
```

Java

```
// Java program for implementation of QuickSort
class QuickSort
{
        /* This function takes last element as pivot,
        places the pivot element at its correct
        position in sorted array, and places all
        smaller (smaller than pivot) to left of
        pivot and all greater elements to right
        of pivot */
        int partition(int arr[], int low, int high)
        {
                int pivot = arr[high];
```

```
                int i = (low-1); // index of smaller element
                for (int j=low; j<high; j++)
                {
                        // If current element is smaller than or
                        // equal to pivot
                        if (arr[j] <= pivot)
                        {
                                i++;

                                // swap arr[i] and arr[j]
                                int temp = arr[i];
                                arr[i] = arr[j];
                                arr[j] = temp;
                        }
                }

                // swap arr[i+1] and arr[high] (or pivot)
                int temp = arr[i+1];
                arr[i+1] = arr[high];
                arr[high] = temp;

                return i+1;
        }


        /* The main function that implements QuickSort()
        arr[] --> Array to be sorted,
        low --> Starting index,
        high --> Ending index */
        void sort(int arr[], int low, int high)
        {
                if (low < high)
                {
                        /* pi is partitioning index, arr[pi] is
                        now at right place */
                        int pi = partition(arr, low, high);

                        // Recursively sort elements before
                        // partition and after partition
                        sort(arr, low, pi-1);
                        sort(arr, pi+1, high);
                }
        }

        /* A utility function to print array of size n */
        static void printArray(int arr[])
        {
```

```
                int n = arr.length;
                for (int i=0; i<n; ++i)
                        System.out.print(arr[i]+" ");
                System.out.println();
        }

        // Driver program
        public static void main(String args[])
        {
                int arr[] = {10, 7, 8, 9, 1, 5};
                int n = arr.length;

                QuickSort ob = new QuickSort();
                ob.sort(arr, 0, n-1);

                System.out.println("sorted array");
                printArray(arr);
        }
}
```

Python

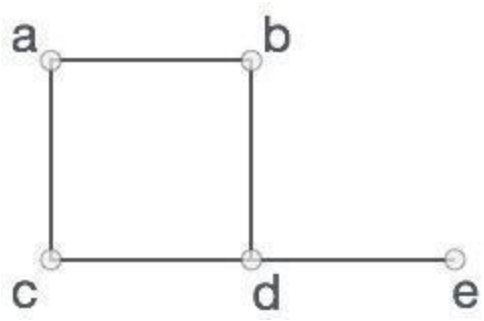
```
# This function takes last element as pivot, places
# the pivot element at its correct position in sorted
# array, and places all smaller (smaller than pivot)
# to left of pivot and all greater elements to right
# of pivot

def partition(arr, low, high):
        i = (low-1)             # index of smaller element
        pivot = arr[high]       # pivot

        for j in range(low, high):

                # If current element is smaller than or
                # equal to pivot
                if arr[j] <= pivot:

                        # increment index of smaller element
                        i = i+1
                        arr[i], arr[j] = arr[j], arr[i]

        arr[i+1], arr[high] = arr[high], arr[i+1]
        return (i+1)
```

```
# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low --> Starting index,
# high --> Ending index

# Function to do Quick sort


def quickSort(arr, low, high):
	if len(arr) == 1:
		return arr
	if low < high:

		# pi is partitioning index, arr[p] is now
		# at right place
		pi = partition(arr, low, high)

		# Separately sort elements before
		# partition and after partition
		quickSort(arr, low, pi-1)
		quickSort(arr, pi+1, high)


# Driver code to test above
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
quickSort(arr, 0, n-1)
print("Sorted array is:")
for i in range(n):
	print("%d" % arr[i]),
```

Graph

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

Graph Data Structure

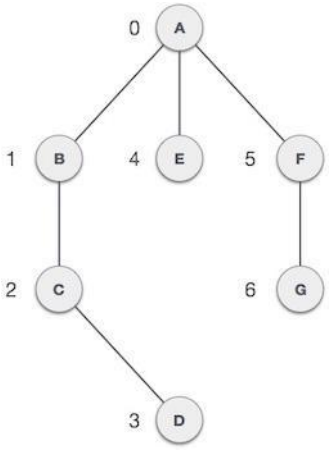
Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

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

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

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

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



Basic Operations

Following are basic primary operations of a Graph −

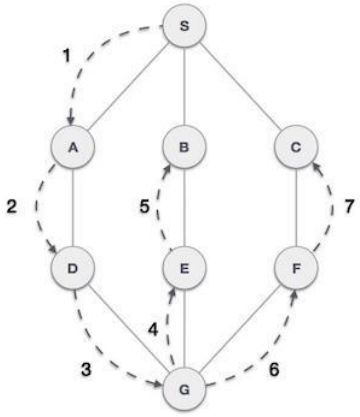
Add Vertex − Adds a vertex to the graph.

Add Edge − Adds an edge between the two vertices of the graph.

Display Vertex − Displays a vertex of the graph.

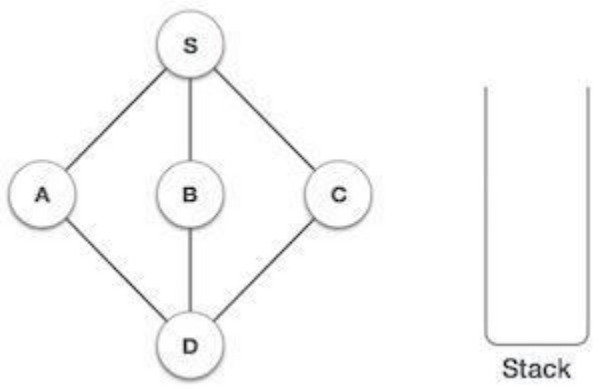
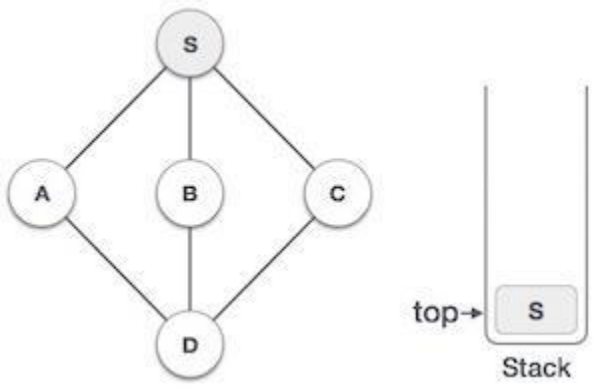
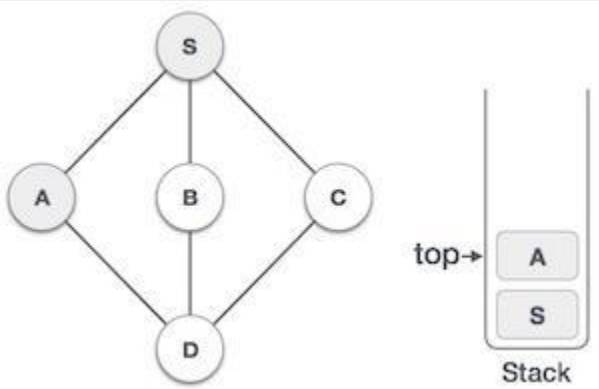
Depth First Traversal

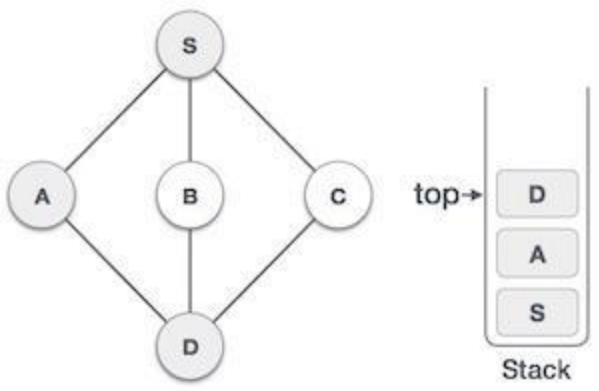
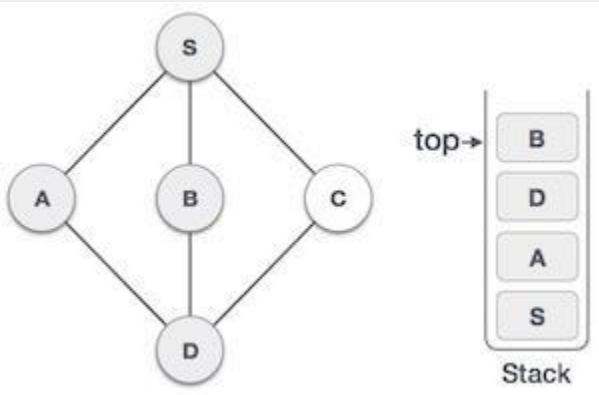
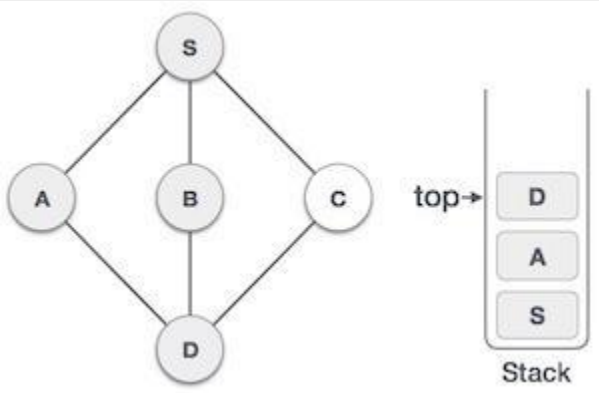
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

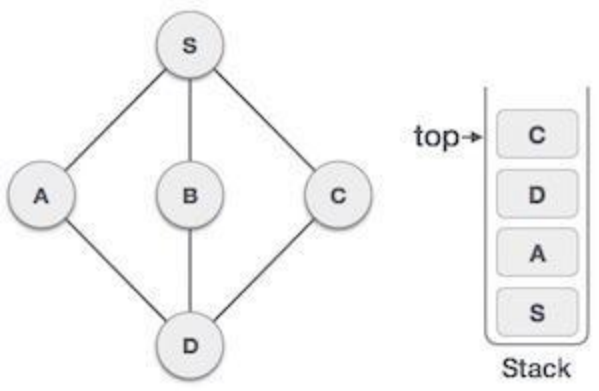


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- 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.

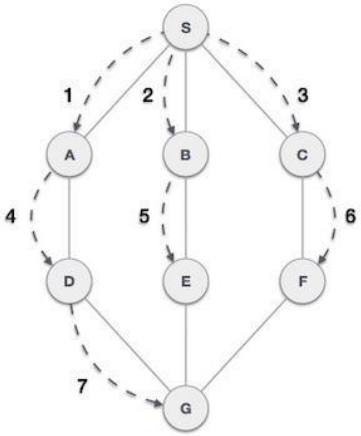
Step	Traversal	Description
1		Initialize the stack.
2		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark A as visited and put it onto the 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		Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.
6		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		Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.

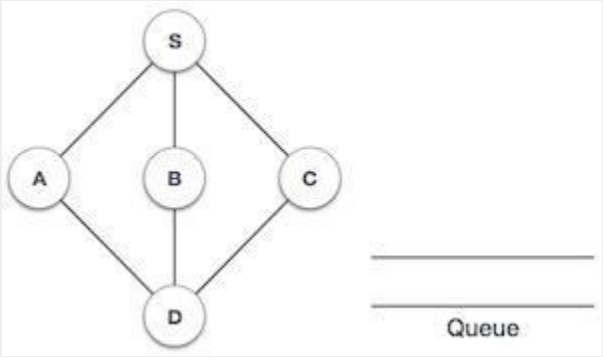
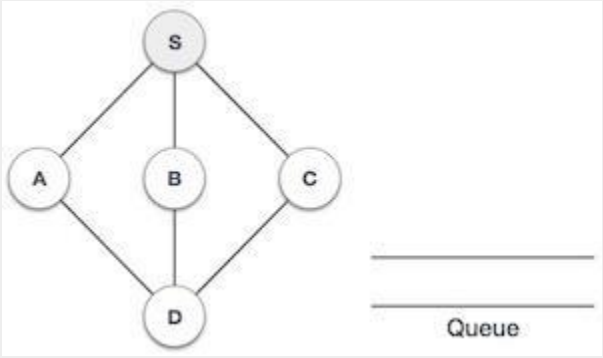
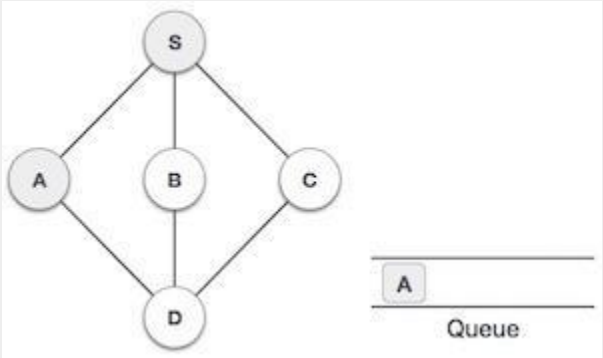
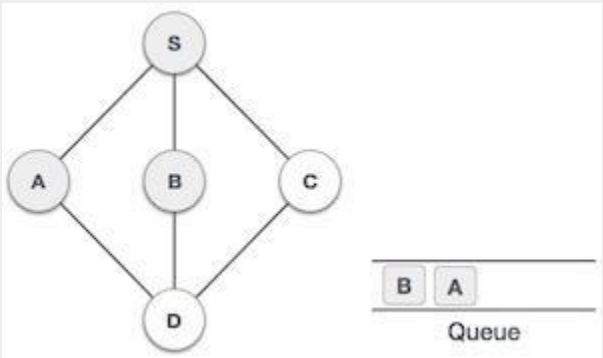
Breadth First Traversal

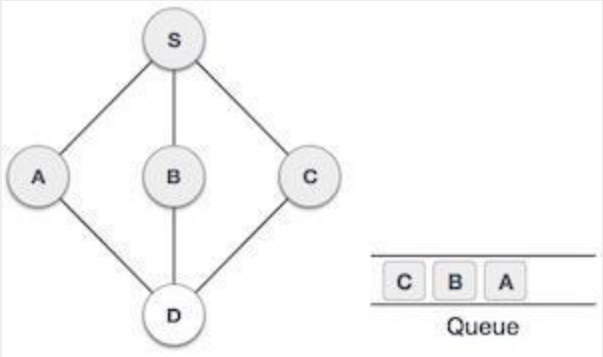
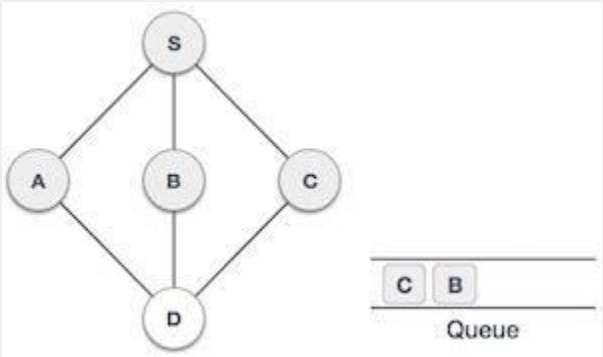
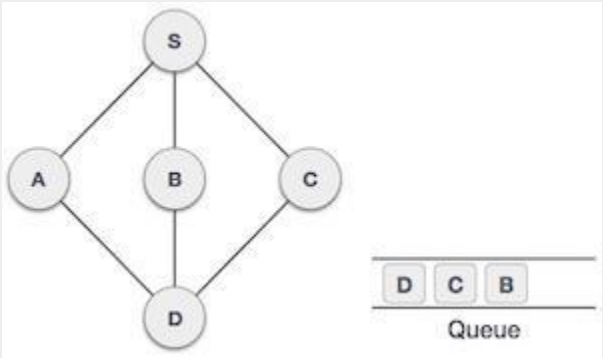
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



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

- Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- Rule 2 − If no adjacent vertex is found, remove the first vertex from the queue.
- Rule 3 − Repeat Rule 1 and Rule 2 until the queue is empty.

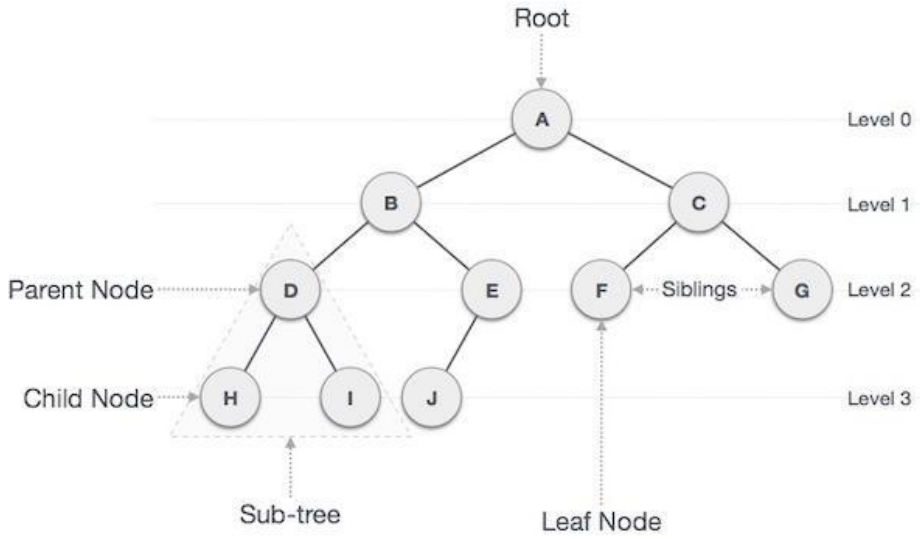
Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting S (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.

5		Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.
6		Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.
7		From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.

Tree

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Important Terms

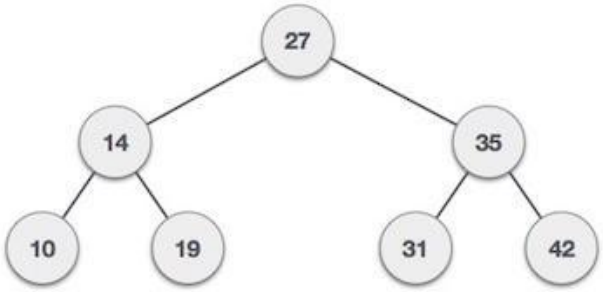
Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.

- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following –

- **Insert** – Inserts an element in a tree/create a tree.
- **Search** – Searches an element in a tree.
- **Preorder Traversal** – Traverses a tree in a pre-order manner.
- **Inorder Traversal** – Traverses a tree in an in-order manner.

- **Postorder Traversal** – Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
If root is NULL
  then create root node
return

If root exists then
  compare the data with node.data

  while until insertion position is located

    If data is greater than node.data
       goto right subtree
    else
       goto left subtree

  endwhile

  insert data

end If
```

Implementation

The implementation of insert function should look like this –

```
void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;

  tempNode->data = data;
```

```
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty, create root node
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent  = NULL;

      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }

         //go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
  return root
else
  while data not found

    If data is greater than node.data
      goto right subtree
    else
      goto left subtree

    If data found
      return node
  endwhile

  return data not found

end if
```

The implementation of this algorithm should look like this.

```
struct node* search(int data) {
  struct node *current = root;
  printf("Visiting elements: ");

  while(current->data != data) {
    if(current != NULL)
    printf("%d ",current->data);

    //go to left tree

    if(current->data > data) {
      current = current->leftChild;
    }
    //else go to right tree
```

```
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
      }

      return current;
   }
}
```

Program

Java

```
public class BinarySearchTree {

   //Represent a node of binary tree
   public static class Node{
      int data;
      Node left;
      Node right;

      public Node(int data){
         //Assign data to the new node, set left and right children to null
         this.data = data;
         this.left = null;
         this.right = null;
      }
   }

   //Represent the root of binary tree
   public Node root;

   public BinarySearchTree(){
      root = null;
   }

   //insert() will add new node to the binary search tree
   public void insert(int data) {
```

```
        //Create a new node
        Node newNode = new Node(data);

        //Check whether tree is empty
        if(root == null){
            root = newNode;
            return;
          }
        else {
            //current node point to root of the tree
            Node current = root, parent = null;

            while(true) {
                //parent keep track of the parent node of current node.
                parent = current;

                //If data is less than current's data, node will be inserted to the left of tree
                if(data < current.data) {
                    current = current.left;
                    if(current == null) {
                        parent.left = newNode;
                        return;
                    }
                }
                //If data is greater than current's data, node will be inserted to the right of tree
                else {
                    current = current.right;
                    if(current == null) {
                        parent.right = newNode;
                        return;
                    }
                }
            }
        }
    }

    //minNode() will find out the minimum node
    public Node minNode(Node root) {
        if (root.left != null)
            return minNode(root.left);
```

```
        else
            return root;
    }

    //deleteNode() will delete the given node from the binary search tree
    public Node deleteNode(Node node, int value) {
        if(node == null){
            return null;
         }
        else {
            //value is less than node's data then, search the value in left subtree
            if(value < node.data)
                node.left = deleteNode(node.left, value);

            //value is greater than node's data then, search the value in right subtree
            else if(value > node.data)
                node.right = deleteNode(node.right, value);

            //If value is equal to node's data that is, we have found the node to be deleted
            else {
                //If node to be deleted has no child then, set the node to null
                if(node.left == null && node.right == null)
                    node = null;

                //If node to be deleted has only one right child
                else if(node.left == null) {
                    node = node.right;
                }

                //If node to be deleted has only one left child
                else if(node.right == null) {
                    node = node.left;
                }

                //If node to be deleted has two children node
                else {
                    //then find the minimum node from right subtree
                    Node temp = minNode(node.right);
                    //Exchange the data between node and temp
                    node.data = temp.data;
```

```
                //Delete the node duplicate node from right subtree
                node.right = deleteNode(node.right, temp.data);
            }
        }
        return node;
    }
}

//inorder() will perform inorder traversal on binary search tree
public void inorderTraversal(Node node) {

    //Check whether tree is empty
    if(root == null){
        System.out.println("Tree is empty");
        return;
     }
    else {

        if(node.left!= null)
            inorderTraversal(node.left);
        System.out.print(node.data + " ");
        if(node.right!= null)
            inorderTraversal(node.right);

    }
}

public static void main(String[] args) {

    BinarySearchTree bt = new BinarySearchTree();
    //Add nodes to the binary tree
    bt.insert(50);
    bt.insert(30);
    bt.insert(70);
    bt.insert(60);
    bt.insert(10);
    bt.insert(90);

    System.out.println("Binary search tree after insertion:");
    //Displays the binary tree
```

```
        bt.inorderTraversal(bt.root);

        Node deletedNode = null;
        //Deletes node 90 which has no child
        deletedNode = bt.deleteNode(bt.root, 90);
        System.out.println("\nBinary search tree after deleting node 90:");
        bt.inorderTraversal(bt.root);

        //Deletes node 30 which has one child
        deletedNode = bt.deleteNode(bt.root, 30);
        System.out.println("\nBinary search tree after deleting node 30:");
        bt.inorderTraversal(bt.root);

        //Deletes node 50 which has two children
        deletedNode = bt.deleteNode(bt.root, 50);
        System.out.println("\nBinary search tree after deleting node 50:");
        bt.inorderTraversal(bt.root);
    }
}
```

Python

```
class Binary_Search_Tree:

    """
    Constructor with vaue we are going
    to insert in tree with assigning
    left and right child with default None
    """

    def __init__(self, data):
        self.data = data
        self.Left_child = None
        self.Right_child = None

    """
    If the data we are inserting already
    present in tree it will not add it
    to avoid the duplicate values
    """
```

```
    def Add_Node(self, data):
        if data == self.data:
            return # node already exist

        """
        If the data we are inserting is Less
        than the value of the current node, then
        data will insert in Left node
        """

        if data < self.data:
            if self.Left_child:
                self.Left_child.Add_Node(data)
            else:
                self.Left_child = Binary_Search_Tree(data)

            """
            If the data we are inserting is Greater
            than the value of the current node, then
            data will insert in Right node
            """

        else:
            if self.Right_child:
                self.Right_child.Add_Node(data)
            else:
                self.Right_child = Binary_Search_Tree(data)


    def Find_Node(self, val):

        """
        If current node is equal to
        data we are finding return true
        """

        if self.data == val:
            return True

        """
```

```
        If current node is lesser than
        data we are finding we have search
        in Left child node
        """

        if val < self.data:
            if self.Left_child:
                return self.Left_child.Find_Node(val)
            else:
                return False

        """
        If current node is Greater than
        data we are finding we have search
        in Right child node
        """

        if val > self.data:
            if self.Right_child:
                return self.Right_child.Find_Node(val)
            else:
                return False

    """
    First it will visit Left node then
    it will visit Root node and finally
    it will visit Right and display a
    list in specific order
    """

    def In_Order_Traversal(self):
        elements = []
        if self.Left_child:
            elements += self.Left_child.In_Order_Traversal()

        elements.append(self.data)

        if self.Right_child:
            elements += self.Right_child.In_Order_Traversal()
```

```
        return elements

    """
    First it will visit Left node then
    it will visit Right node and finally
    it will visit Root node  and display a
    list in specific order
    """

    def Post_Order_Traversal(self):
        elements = []
        if self.Left_child:
            elements += self.Left_child.Post_Order_Traversal()
        if self.Right_child:
            elements += self.Right_child.Post_Order_Traversal()

        elements.append(self.data)

        return elements

    """
    First it will visit Root node then
    it will visit Left node and finally
    it will visit Right node  and display a
    list in specific order
    """

    def Pre_Order_Traversal(self):
        elements = [self.data]
        if self.Left_child:
            elements += self.Left_child.Pre_Order_Traversal()
        if self.Right_child:
            elements += self.Right_child.Pre_Order_Traversal()

        return elements

    """
    This method will give
    the Max value of tree
    """
```

```
    def Find_Maximum_Node(self):
        if self.Right_child is None:
            return self.data
        return self.Right_child.Find_Maximum_Node()

    """
    This method will give
    the Min value of tree
    """

    def Find_Minimum_Node(self):
        if self.Left_child is None:
            return self.data
        return self.Left_child.Find_Minimum_Node()

    """
    This method will give
    the Total Sum value of tree
    """

    def calculate_Sum_Of_Nodes(self):
        left_sum = self.Left_child.calculate_Sum_Of_Nodes() if self.Left_child else 0
        right_sum = self.Right_child.calculate_Sum_Of_Nodes() if self.Right_child else 0
        return self.data + left_sum + right_sum

"""
This method helps to build the tree
whith the element we inserted in it
"""

def Build_Tree(elements):
    root = Binary_Search_Tree(elements[0])

    for i in range(1,len(elements)):
        root.Add_Node(elements[i])

    return root
```

Tree Traversal

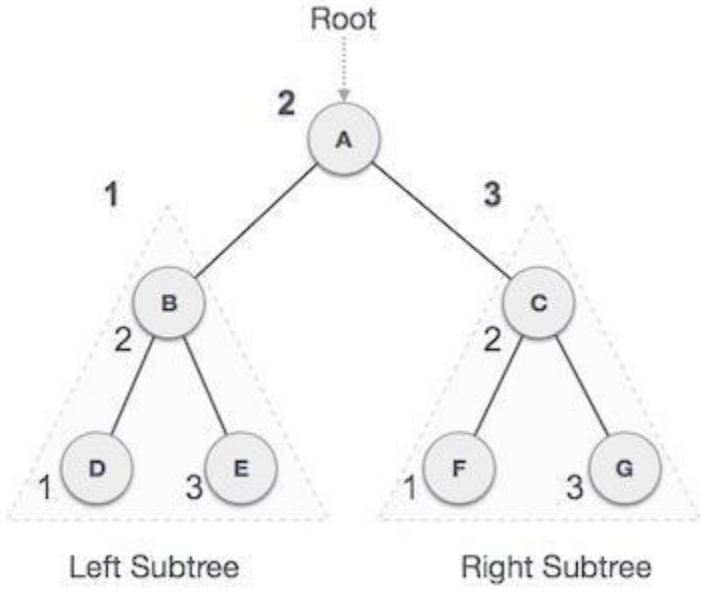
Traversal is a process to visit all the nodes of a tree and may print their values too. Because all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

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

Generally, we traverse the tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right subtree. 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 ascending order.



We start from A, and following in−order traversal, we move to its left subtree B. B is also traversed in−order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

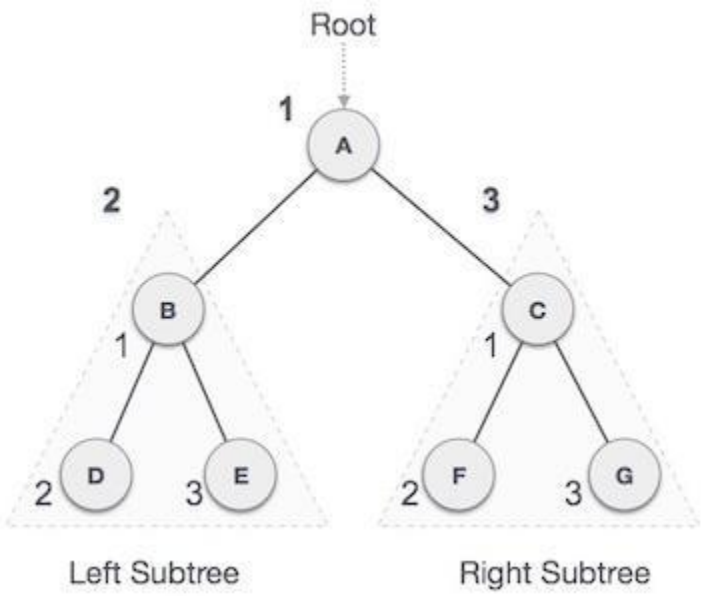
D → B → E → A → F → C → G

Algorithm

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

Pre-order Traversal

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



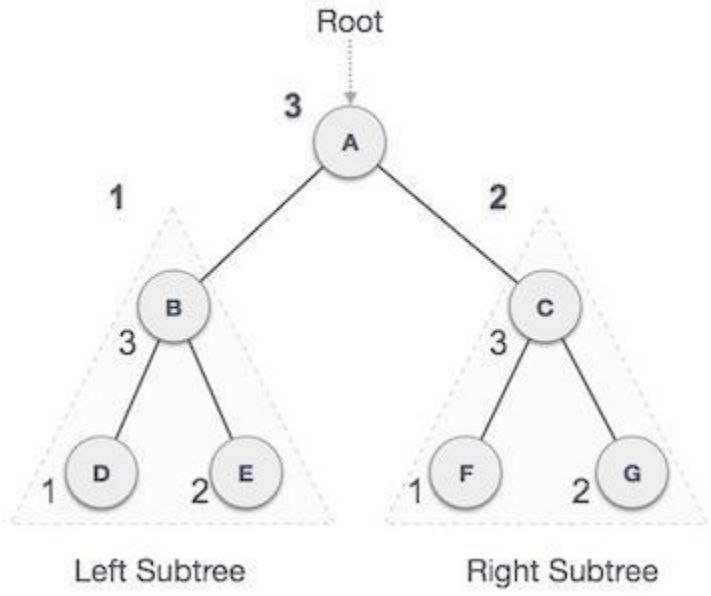
We start from A, and following pre−order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre−order. The process goes on until all the nodes are visited. The output of pre−order traversal of this tree will be −
A → B → D → E → C → F → G

Algorithm

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

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.



We start from A, and following Post−order traversal, we first visit the left subtree B. B is also traversed post−order. The process goes on until all the nodes are visited. The output of post−order traversal of this tree will be −

D → E → B → F → G → 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.

Binary Search Tree

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

- The left subtree of a node has a key less than or equal to its parent node's key.
- The right subtree of a node has a key greater than to its parent node's key.

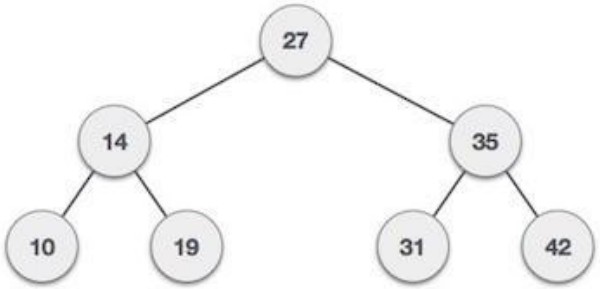
Thus, BST divides all its sub−trees into two segments; the left subtree and the right subtree and can be defined as −

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

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −



We observe that the root node key (27) has all less-valued keys on the left subtree and the higher valued keys on the right subtree.

Basic Operations

Following are the basic operations of a tree −

- Search − Searches an element in a tree.
- Insert − Inserts an element in a tree.
- Pre-order Traversal − Traverses a tree in a pre-order manner.
- In-order Traversal − Traverses a tree in an in-order manner.
- Post-order Traversal − Traverses a tree in a post-order manner.

Node

Define a node having some data, references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
struct node* search(int data){
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data){

      if(current != NULL) {
         printf("%d ",current->data);

         //go to left tree
         if(current->data > data){
            current = current->leftChild;
         }  //else go to right tree
         else {
            current = current->rightChild;
         }

         //not found
         if(current == NULL){
            return NULL;
         }
      }
   }

   return current;
}
```

Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

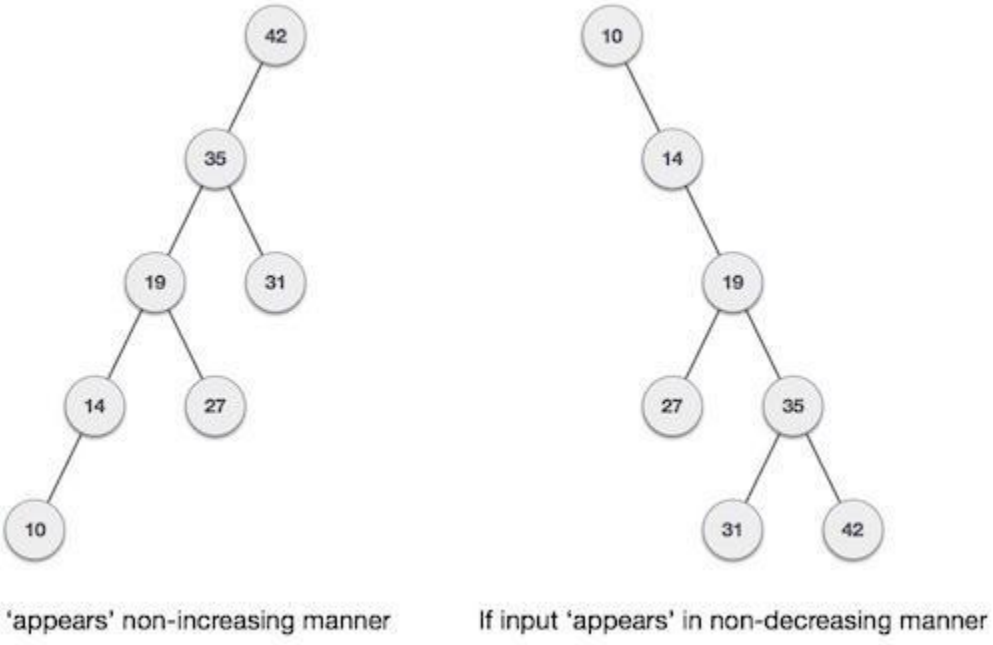
Algorithm

```
void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;

  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;

  //if tree is empty
  if(root == NULL) {
    root = tempNode;
  } else {
    current = root;
    parent = NULL;

    while(1) {
      parent = current;

      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;
        //insert to the left

        if(current == NULL) {
          parent->leftChild = tempNode;
          return;
        }
      }  //go to right of the tree
      else {
        current = current->rightChild;

        //insert to the right
        if(current == NULL) {
          parent->rightChild = tempNode;
          return;
        }
      }
    }
```

```
  }
}
```

AVL Tree

What if the input to the binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −



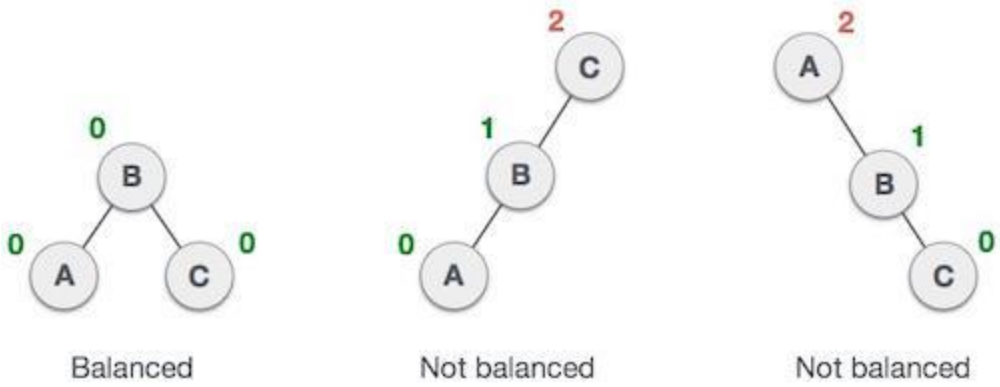
If input 'appears' non-increasing manner

If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data patterns and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor Adelson, Velski & Landis, AVL trees are height balancing binary search trees. AVL tree checks the height of the left and the right subtrees and assures that the difference is not more than 1. This difference is called the Balance Factor.

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

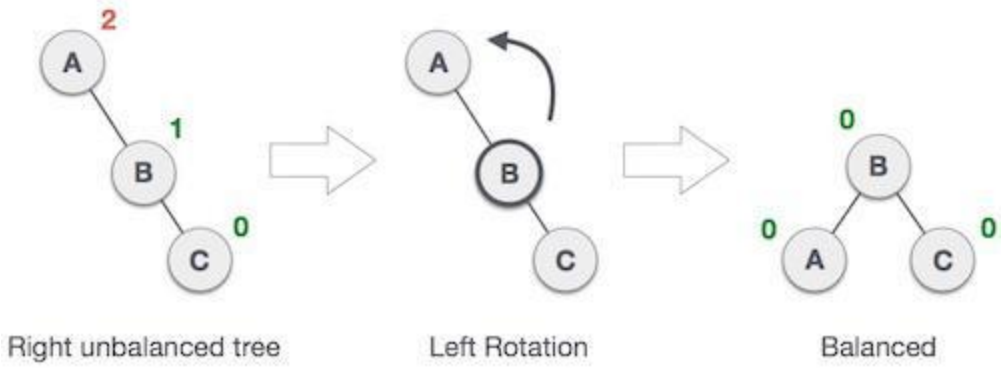
To balance itself, an AVL tree may perform the following four kinds of rotations −

- 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

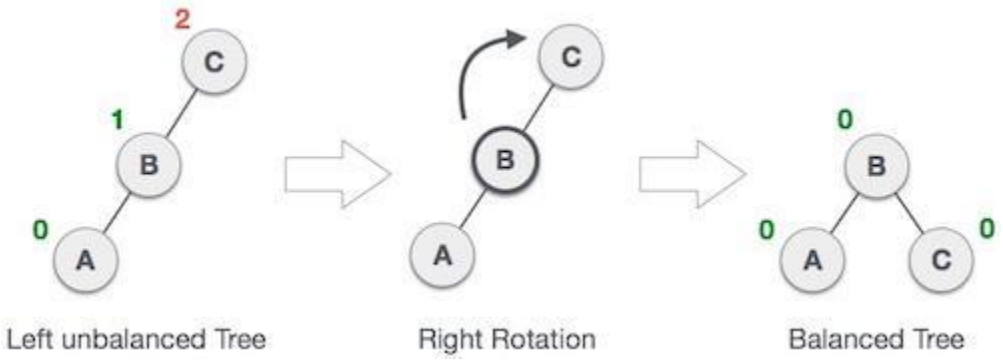
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



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.

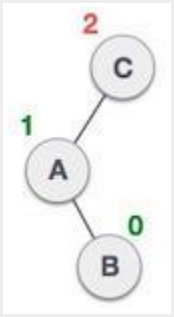
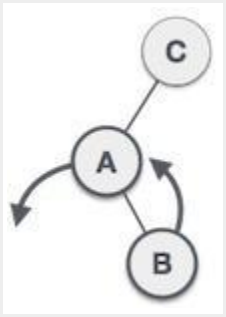
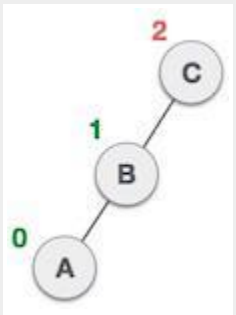
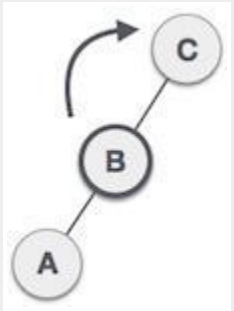


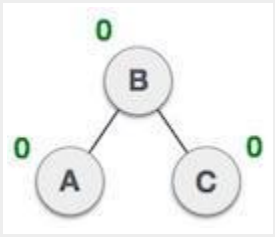
As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

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.

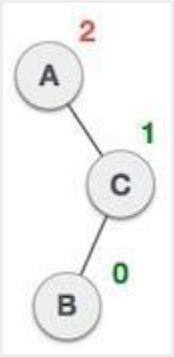
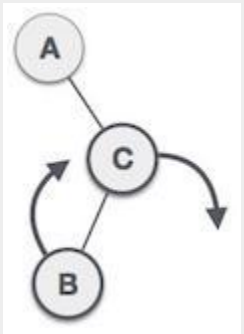
State	Action

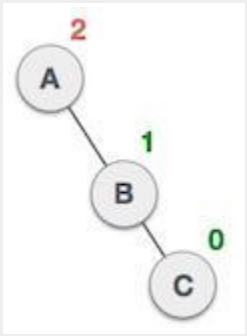
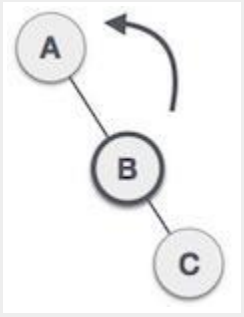
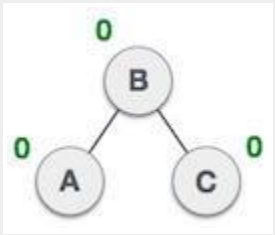
	A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.
	Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	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.

	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.

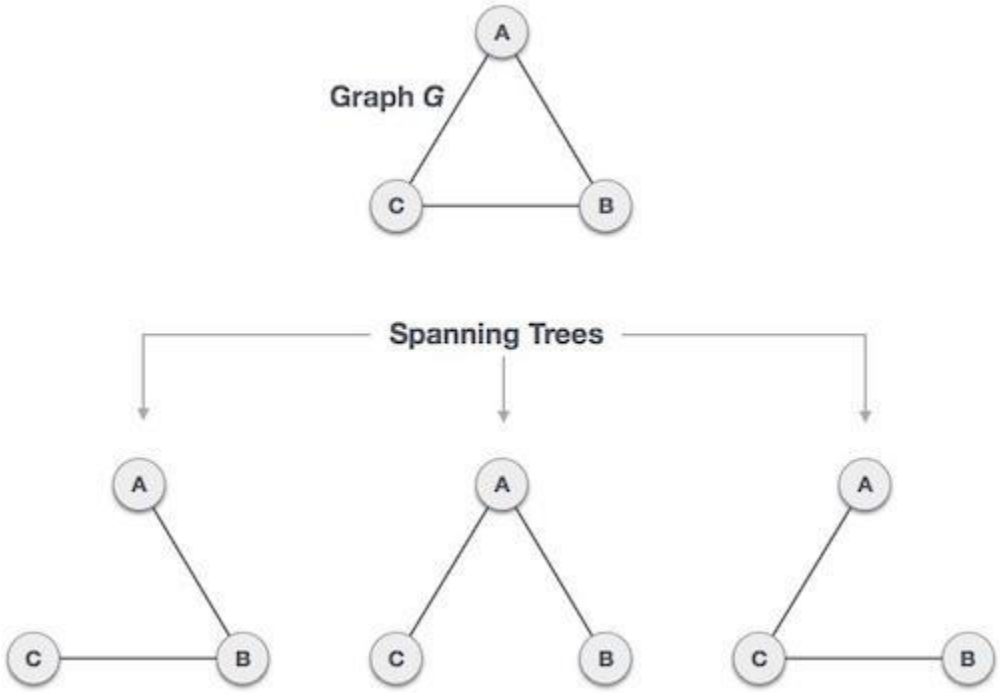
State	Action
	A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.
	First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.

	Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	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.
	The tree is now balanced.

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..

By this definition, we can draw a conclusion that 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.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n_{n-2} number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3, hence $3^{3-2} = 3$ spanning trees are possible.

General 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.

Mathematical Properties of Spanning Tree

- Spanning tree has n-1 edges, where n is the number of nodes (vertices).

- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
- A complete graph can have maximum nn-2 number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

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.

Algorithms

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

- Search − Algorithm to search an item in a data structure.
- Sort − Algorithm to sort items in a certain order.
- Insert − Algorithm to insert item in a data structure.
- Update − Algorithm to update an existing item in a data structure.

- Delete − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

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

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well−defined inputs.
- **Output** − An algorithm should have 1 or more well−defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step−by−step directions, which should be independent of any programming code.

How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

Step 1 − START

Step 2 − declare three integers **a**, **b** & **c**

Step 3 − define values of **a** & **b**

Step 4 − add values of **a** & **b**

Step 5 − store output of step 4 to **c**

Step 6 − print **c**

Step 7 − STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as −

Step 1 − START ADD

Step 2 − get values of **a** & **b**

Step 3 − c ← a + b

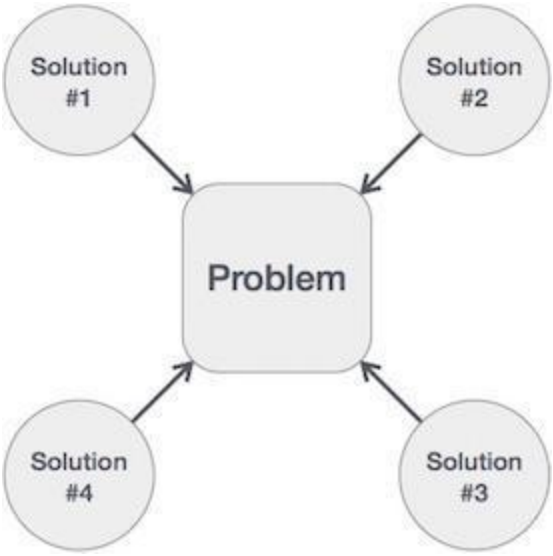
Step 4 − display c

Step 5 − STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing step numbers, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

Algorithm Analysis

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

- *A Priori* Analysis − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- *A Posterior* Analysis − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about *a priori* algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c * n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

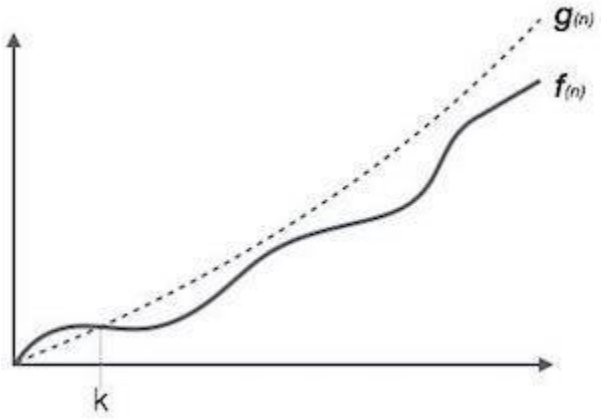
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

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

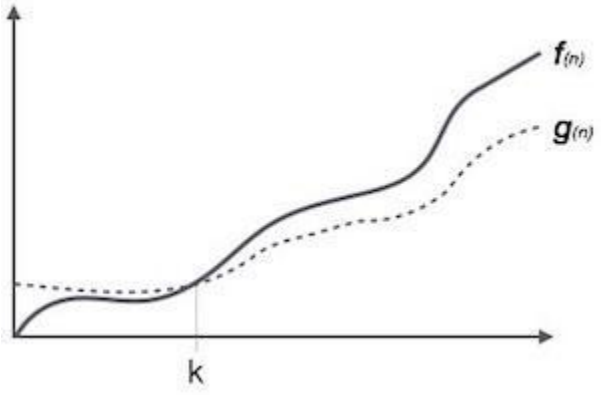


For example, for a function *f*(n)

O(*f*(n)) = { *g*(n) : there exists c > 0 and n_0 such that *f*(n) ≤ c.*g*(n) for all n > n_0. }

Omega Notation, Ω

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

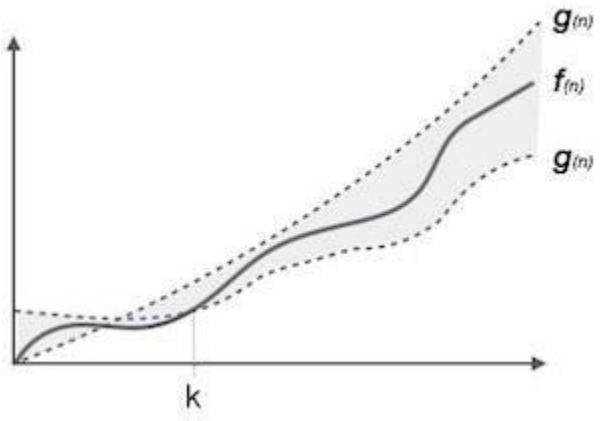


For example, for a function *f*(n)

Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n_0 such that *g*(n) ≤ c.*f*(n) for all n > n_0. }

Theta Notation, θ

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



θ(*f*(n)) = { *g*(n) if and only if *g*(n) = O(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n_0. }

Common Asymptotic Notations

Following is a list of some common asymptotic notations −

constant	–	O(1)
logarithmic	–	O(log n)
linear	–	O(n)
n log n	–	O(n log n)

quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$

Greedy Algorithm

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Counting Coins

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be −

- 1 − Select one ₹ 10 coin, the remaining count is 8
- 2 − Then select one ₹ 5 coin, the remaining count is 3
- 3 − Then select one ₹ 2 coin, the remaining count is 1
- 4 − And finally, the selection of one ₹ 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)

Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

Examples

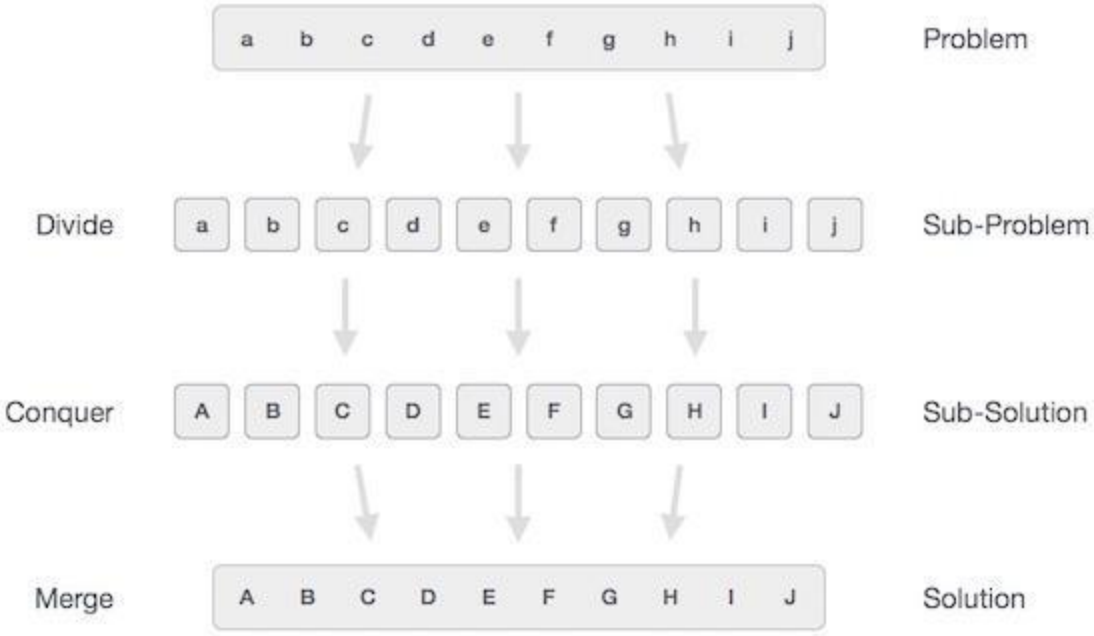
`Most networking algorithms use the greedy approach. Here is a list of few of them –`

- Travelling Salesman Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Graph - Map Coloring
- Graph - Vertex Cover
- Knapsack Problem
- Job Scheduling Problem

There are lots of similar problems that uses the greedy approach to find an optimum solution.

Divide and Conquer

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.



Broadly, we can understand divide-and-conquer approach in a three-step process.

Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve

This step receives a lot of smaller subproblems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine

When the smaller subproblems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Examples

`The following computer algorithms are based on divide-and-conquer programming approach –`

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication

- Closest pair (points)

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

Dynamic Programming

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that –

- The problem should be able to be divided into smaller overlapping sub-problem.
- An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
- Dynamic algorithms use Memoization.

Comparison

In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.

In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use Memoization to remember the output of already solved sub-problems.

Example

The following computer problems can be solved using dynamic programming approach –

- Fibonacci number series
- Knapsack problem
- Tower of Hanoi
- All pair shortest path by Floyd-Warshall
- Shortest path by Dijkstra
- Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

Recursions

Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function α either calls itself directly or calls a function β that in turn calls the original function α. The function α is called recursive function.

`Example – a function calling itself.`

```
int function(int value) {
  if(value < 1)
    return;
  function(value - 1);

  printf("%d ",value);
}
```

`Example – a function that calls another function which in turn calls it again.`

```
int function1(int value1) {
  if(value1 < 1)
    return;
  function2(value1 - 1);
  printf("%d ",value1);
}
int function2(int value2) {
  function1(value2);     }
```

Properties

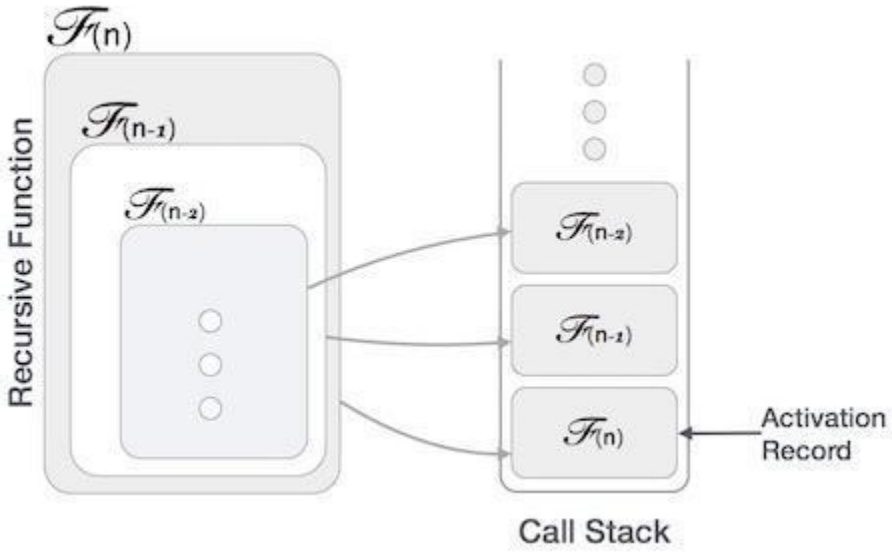
`A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have –`

- Base criteria – There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- Progressive approach – The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Implementation

Many programming languages implement recursion by means of stacks. Generally, whenever a function (caller) calls another function (callee) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.

This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.



This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.

Analysis of Recursion

One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

Time Complexity

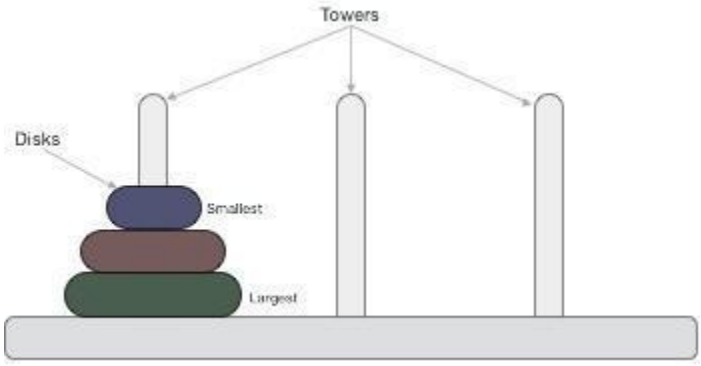
In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is O(1), hence the (n) number of times a recursive call is made makes the recursive function O(n).

Space Complexity

Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.

Tower of Hanoi

`Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted –`



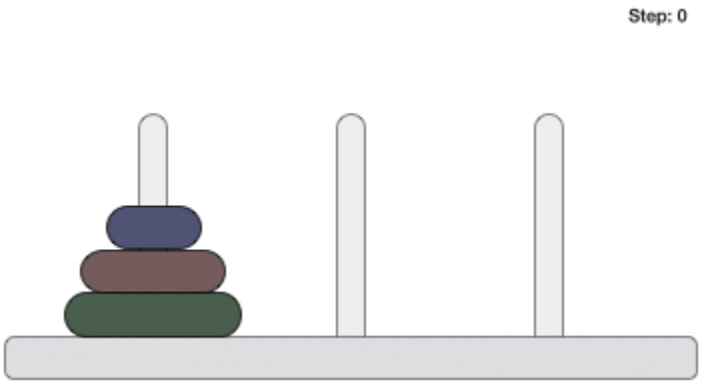
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are –

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



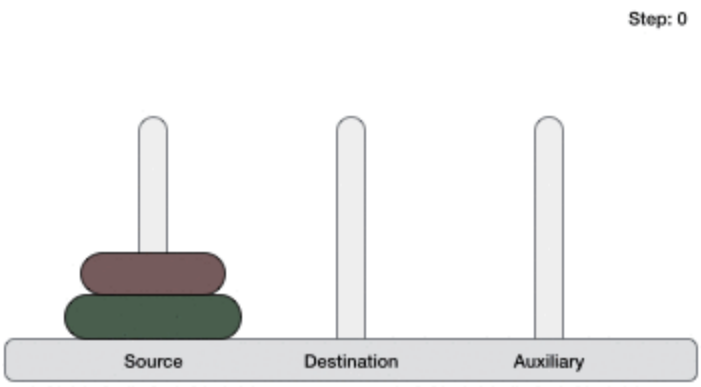
Tower of Hanoi puzzle with n disks can be solved in minimum $2^n - 1$ steps. This presentation shows that a puzzle with 3 disks has taken $2^3 - 1 = 7$ steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks –

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk n from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are –

Step 1 – Move n–1 disks from source to aux
Step 2 – Move n_{th} disk from source to dest
Step 3 – Move n–1 disks from aux to dest

A recursive algorithm for Tower of Hanoi can be driven as follows –

```
START
Procedure Hanoi(disk, source, dest, aux)

  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)    // Step 1
    move disk from source to dest         // Step 2
    Hanoi(disk - 1, aux, dest, source)    // Step 3
  END IF

END Procedure
STOP
```

Fibonacci Series

Fibonacci series generates the subsequent number by adding two previous numbers. Fibonacci series starts from two numbers – F_0 & F_1. The initial values of F_0 & F_1 can be taken 0, 1 or 1, 1 respectively.

Fibonacci series satisfies the following conditions –

$F_n = F_{n-1} + F_{n-2}$

Hence, a Fibonacci series can look like this –

F_8 = 0 1 1 2 3 5 8 13

or, this –

F_8 = 1 1 2 3 5 8 13 21

For illustration purpose, Fibonacci of F_8 is displayed as –

Fibonacci Iterative Algorithm

First we try to draft the iterative algorithm for Fibonacci series.

```
Procedure Fibonacci(n)
  declare f0, f1, fib, loop

  set f0 to 0
  set f1 to 1

  display f0, f1

  for loop ← 1 to n

    fib ← f0 + f1
    f0 ← f1
    f1 ← fib

    display fib
  end for

end procedure
```

Fibonacci Recursive Algorithm

Let us learn how to create a recursive algorithm Fibonacci series. The base criteria of recursion.

```
START
Procedure Fibonacci(n)
  declare f0, f1, fib, loop

  set f0 to 0
  set f1 to 1

  display f0, f1

  for loop ← 1 to n

    fib ← f0 + f1
    f0 ← f1
    f1 ← fib

    display fib
  end for

END
```