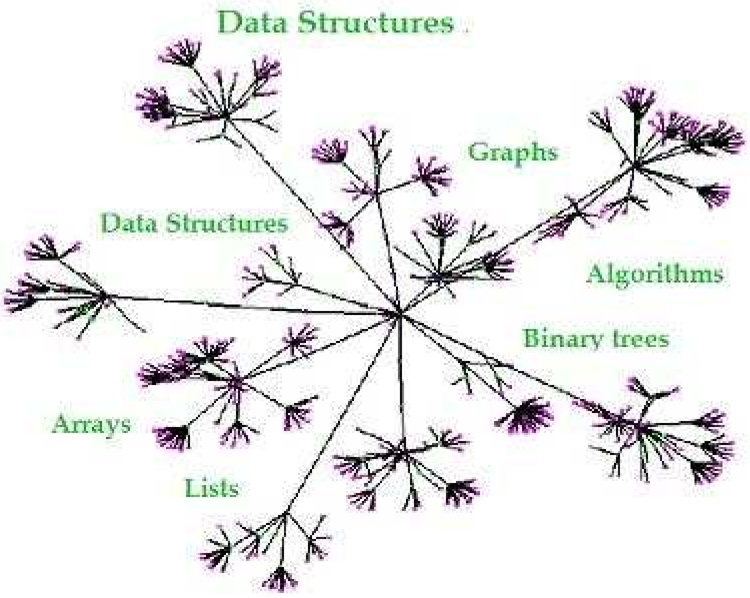
MUFFAKHAM JAH COLLEGE OF ENGINEERING AND TECHNOLOGY

Banjara Hills, Hyderabad, Telangana

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



Data Structures using C++ Laboratory Manual

Academic Year 2016-2017

Table of Contents

Part I

Contents

1. Vision of the Institution

To be part of universal human quest for development and progress by contributing high calibre, ethical and socially responsible engineers who meet the global challenge of building modern society in harmony with nature.

2. Mission of the Institution

- To attain excellence in imparting technical education from undergraduate through doctorate levels by adopting coherent and judiciously coordinated curricular and co-curricular programs.
- To foster partnership with industry and government agencies through collaborative research and consultancy.
- To nurture and strengthen auxiliary soft skills for overall development and improved employability in a multi-cultural work space.
- To develop scientific temper and spirit of enquiry in order to harness the latent innovative talents.
- To develop constructive attitude in students towards the task of nation building and empower them to become future leaders
- To nourish the entrepreneurial instincts of the students and hone their business acumen.
- To involve the students and the faculty in solving local community problems through economical and sustainable solutions.

3. Department Vision

To contribute competent computer science professionals to the global talent pool to meet the constantly evolving societal needs.

4. Department Mission

Mentoring students towards a successful professional career in a global environment through quality education and soft skills in order to meet the evolving societal needs.

5. Programme Education Objectives

1. Graduates will demonstrate technical skills and leadership in their chosen fields of employment by solving real time problems using current techniques and tools.

2. Graduates will communicate effectively as individuals or team members and be successful in the local and global cross cultural working environment.

3. Graduates will demonstrate lifelong learning through continuing education and professional development.

4. Graduates will be successful in providing viable and sustainable solutions within societal, professional, environmental and ethical contexts

6. Programme Outcomes

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Lifelong learning:** Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

7. Programme Specific Outcomes

The graduates will be able to:

PSO1: Demonstrate understanding of the principles and working of the hardware and software aspects of computer systems.

PSO2: Use professional engineering practices, strategies and tactics for the development, operation and maintenance of software

PSO3: Provide effective and efficient real time solutions using acquired knowledge in various domains.

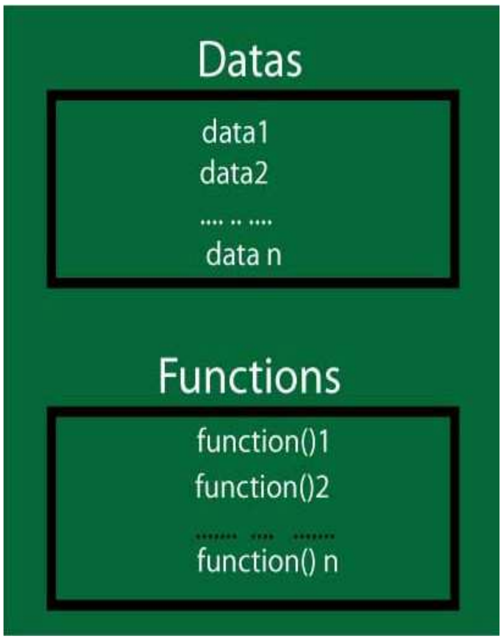
8. Introduction to Data Structures using C++ Laboratory

Class:

A class in C++ is a user defined type declared with keyword class that has data and functions (also called methods) as its members whose access is governed by the three access specifiers private, protected or public (by default access to members of a class is private).

A class is the collection of related data and function under a single name. A C++ program can have any number of classes. When related data and functions are kept under a class, it helps to visualize the complex problem efficiently and effectively.

Class:



A Class is a blueprint for objects. When a class is defined, no memory is allocated. You can imagine like a data type.

```
int var;
```

The above code specifies var is a variable of type integer; int is used for specifying variable var is of integer type. Similarly, class are also just the specification for objects.

Defining the Class in C++

Class is defined in C++ programming using keyword class followed by identifier(name of class). Body of class is defined inside curly brackets and terminated by semicolon at the end in similar way as structure.

```
class class_name{

    // some data

    // some functions
};
```

Example of Class in C++

```
class temp
{
    private:
        int data1;
        float data2;

    public:
        void func1()
        {
            data1=2;
        }
    float func2()
    {
        data2=3.5;
        return data;
    }
};
```

Keywords: `private` and `public`

Keyword `private` makes data and functions private and keyword `public` makes data and functions public. Private data and functions are accessible inside that class only whereas, public data and functions are accessible both inside and outside the class. This feature in OOP is known as data hiding. If programmer mistakenly tries to access private data outside the class, compiler shows error which prevents the misuse of data. Generally, data are private and functions are public.

C++ Objects

When class is defined, only specification for the object is defined. Object has same relationship to class as variable has with the data type. Objects can be defined in similar way as structure is defined.

Syntax to Define Object in C++

class_name variable name;
For the above defined class temp, objects for that class can be defined as:
temp obj1,obj2;
Here, two objects (obj1 and obj2) of temp class are defined.

Data member and Member functions

The data within the class is known as data member. The function defined within the class is known as member function. These two technical terms are frequently used in explaining OOP. In the above class temp, data1 and data2 are data members and func1() and func2() are member functions

Defining Member functions outside the class:

The member functions of the class are generally defined outside the class using following syntax:

```
returntype classname::functionname(list of parameters)
{
        //function body
}
```

Accessing Data Members and Member functions

Data members and member functions can be accessed in similar way the member of structure is accessed using member operator(.). For the class and object defined above, func1() for object obj2 can be called using code:

```
obj2.func1();
```

Similarly, the data member can be accessed as:

```
object_name.data_memeber;
```

Note: You cannot access the data member of the above class temp because both data members are private so it cannot be accessed outside that class.

Example to Explain Working of Object and Class in C++ Programming

```
#include <iostream.h>

class Rectangle
{
    private:

        int length;
        int breadth;
    public:

        void setData(int l,int b);
        int findArea();
        void display();
};

void Rectangle::setData(int l,int b)
{
        length=l;
        breadth=b;
}

int Rectangle::findArea()
{
        return (length*breadth);
}

void Rectangle::display()
{
    cout<<"Rectangle Length: "<<length<<endl;
    cout<<"Rectangle Breadth: "<<breadth<<endl;
}



main()
{

        cout<<"Rectangle Length: "<<length<<endl;
        cout<<"Rectangle Breadth: "<<breadth<<endl;
        Rectangle A1,A2;
        A1.setData(5,6);
        A2.setData(6,7);
        A1.display();
        A2.display();
        cout<<"Area of first rectangle is "<<A1.findArea();
        cout<<"Area of second rectangle is"<<A2.findArea();

}
```

Constructors and Destructors

A **Constructor** is a special type of member function that is used to initialize the object automatically when it gets created.

1. Constructor has same name as that of class and it does not have any return type.
2. It is automatically called whenever an object of that class is created.
3. It is used to initialize data members of the object and to create dynamic memory.

Syntax: ClassName(datatype var1, datatype var2, ...)

```
{
    //initialize the data variables of the object
}
```

A **Destructor** is a special member function of a class that is automatically executed whenever an object of its class goes out of scope

1. Destructor has same name as that of class preceded by   symbol and it does not have any return type or any parameters.
2. It is used to free dynamic memory used by the object.

Syntax:

```
~ClassName()
{

        //free the dynamic memory of the object
}
```

Class Templates

Class template is a generic class that is written so that it can be used with any data type.
Syntax:

```
template<class T>

class NameOfTheClass

{
    private:
        //declare variables of generic type T
    public:
        //declare functions
   };
```

```
The object of a class template is declared as :

classname<datatype> objectname;
```

The data type specified in the object declaration is used to replace the generic datatype T in the class template definition.
The member functions of a class template are defined outside of the class as:

```
template<class T>
returntype classname<T>::functionname(list of parameters)
{
}
```

Example:

```
#include <iostream>
template<class T> class Rectangle
{
    private:
        T length;
        T breadth;

    public:
        void setData(T l,T b); T findArea();
        void display();
};

template<class T>
void Rectangle<T>::setData(T l,T b)
{
    length=l;
    breadth=b;
}

template<class T>
T Rectangle<T>::findArea()
{
    return (length*breadth);
}

template<class T>
void Rectangle<T>::display()
{
    cout<<"Rectangle Length: "<<length<<endl;
    cout<<"Rectangle Breadth: "<<breadth<<endl;
}
```

```
main()
{

    Rectangle<int> A1;
    Rectangle<float> A1;
    A1.setData(5,6);
    A2.setData(6.4,7.2);
    A1.display();
    A2.display();

    cout<<"Area of first rectangle is"<<A1.findArea();
    cout<<"Area of second rectangle is"<<A2.findArea();
}
```

Part II

Programs

Program 1

INSERTION SORT

Problem Definition

Write a C++ program to implement Insertion Sort

Problem Description

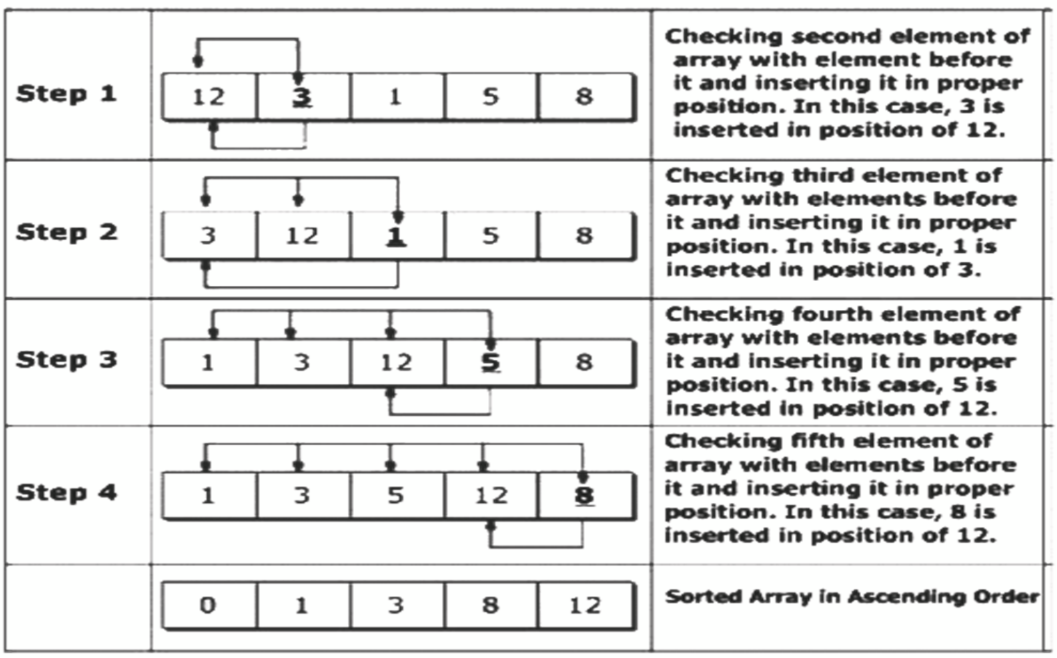

Step 1
12 3 1 5 8
Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2
3 12 1 5 8
Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3
1 3 12 5 8
Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4
1 3 5 12 8
Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
0 1 3 8 12
Sorted Array in Ascending Order

Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Step 1: The second element of an array is compared with the elements that appears before it (only first element in this case). If the second element is smaller than first element, second element is inserted in the position of first element. After first step, first two elements of an array will be sorted.

Step 2: The third element of an array is compared with the elements that appears before it (first and second element). If third element is smaller than first element, it is inserted in the position of first element. If third element is larger than first element but, smaller than second element, it is inserted in the position of second element. If third element is larger than both the elements, it is kept in the position as it is. After second step, first three elements of an array will be sorted.

Step 3: Similary, the fourth element of an array is compared with the elements that appears before it (first, second and third element) and the same procedure is applied and that element is inserted in the proper position. After third step, first four elements of an array will be sorted.

If there are n elements to be sorted. Then, this procedure is repeated n-1 times to get sorted list of array.

Pseudocode

```
for i = 2 to n-1
    x = a[i]
    //Insert A[j] into the sorted sequence A[1  j - 1].
    for j = i - 1 to 0 and a[j] > x
        do A[j + 1] = A[j]
    end for
    a[j + 1] = x

end for
```

Problem Validation

Input:

```
(A)  enter size: 5

     enter elements: 50    9   4    0    3

(B)  enter size: 8

     enter elements: 51    92  34  80   33    78    55   23
```

Output:

```
(A) Sorted List is : 0   3    4    9    50

(B) Sorted List is : 23    33    34    51   55   78    80   92
```

Program 2

MERGE SORT

Problem Definition

Write a C++ program to implement Merge Sort

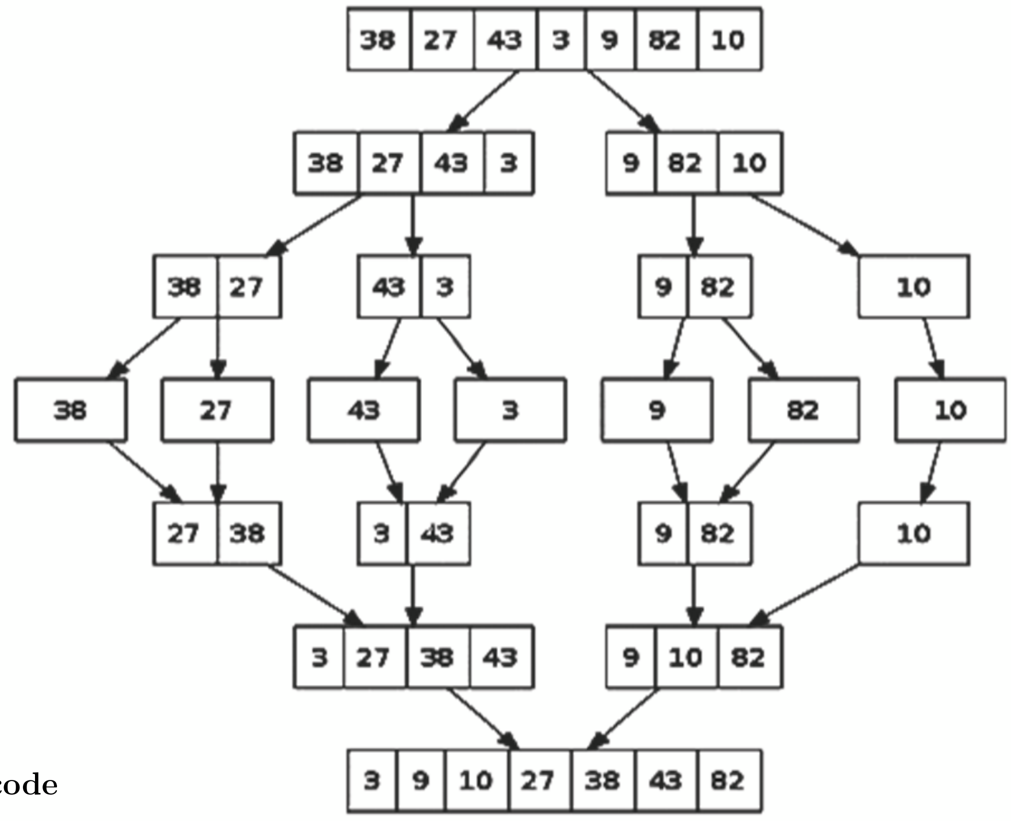
Problem Description

MergeSort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merg() function is used for merging two halves. The merge(arr, low, mid, high) is key process that assumes that arr[low..mid] and arr[mid+1..high] are sorted and merges the two sorted sub-arrays into one.

```
MergeSort(arr[], low,  high)

if high > low
     1. Find the middle point to divide the array into two halves:
          mid = (low+high)/2
     2. Call mergeSort for first half:
          Call mergeSort(a, low, mid)
     3. Call mergeSort for second half:
          Call mergeSort(a, mid+1,high)
     4. Merge the two halves sorted in step 2 and 3:
          Call merge(a, low, mid, high)
```

The following diagram shows the complete merge sort process for an example array {38, 27,43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



Pseudocode

```
MERGESORT(A, low, high)
If low < high
     mid =  ( low + high ) /2;
MERGESORT(A, low, mid);
MERGESORT(A, mid+1, high);
MERGE(A, low, mid, high);
End If
End MERGESORT

MERGE(A,low,mid,high)

Take temporary array B[20]

i = low , j = mid+1 , k = low

While i <= mid and j <= high
     If A[i] < A[j]
          B[k] = A[i]
          Increment k and i by 1
     Else
          B[k] = A[j]
          Increment k and j by 1
End while

While i <= mid
     B[k] = A[i]
     Increment k and i by 1
End while

While j <= mid
     B[k] = A[j]
     Increment k and j by 1
End while
Copy B[] into A[]
End MERGE
```

Problem Validation
Input:

```
        (A)  enter size: 5

             enter elements: 5   4   10    13    6

(B)  enter size: 10

     enter elements: 9  8  7   6   4   0  3  2  1  5  10
```

Output:

```
        (A) Sorted List is : 4  5  6  10  13

        (B) Sorted List is : 1  2  3  4  5  6  7  8  9  10
```

Program 3

QUICK SORT

Problem Definition

Write a C++ program to implement Quick Sort

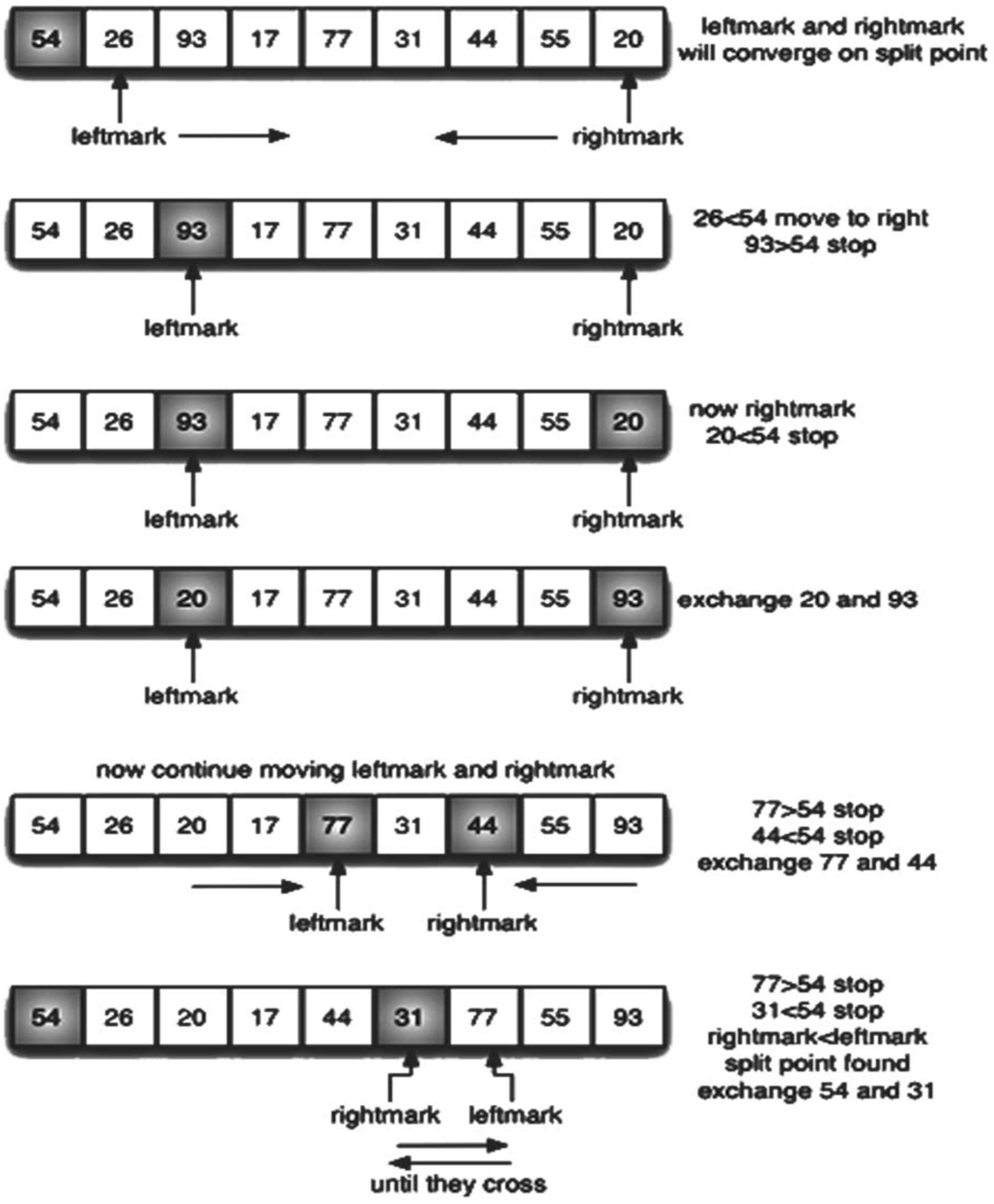
Problem Description

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. The basic idea of Quick sort is described below:

1. Choose a pivot value. We take the value of the first element as pivot value, but it can be any value.
2. Partition. Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
3. Sort both parts. Apply quicksort algorithm recursively to the left and the right parts.

Partition algorithm in detail There are two indices i and j and at the very beginning

of the partition algorithm i points to the first element in the array and j points to the last one. Then algorithm moves i forward, until an element with value greater or equal to the pivot is found. Index j is moved backward, until an element with value lesser or equal to the pivot is found. If i $\leq$ j then they are swapped and i steps to the next position (i + 1), j steps to the previous one (j - 1). Algorithm stops, when i becomes greater than j. After partition, all values before i-th element are less or equal than the pivot and all values after j-th element are greater or equal to the pivot.



Pseudocode

```
QuickSort(a, low, high)
      If low < high
            i = low,j = high
            pivot = a[low]
            //partition algorithm
            while i < j
                    while i <= high and a[i] <= pivot
                          Inc i by 1
            while j >= low and a[j] > pivot
                  Inc j by 1
            if i < j
                  swap a[i] and a[j]
            End while

    swap  a[low] and a[j]

    Recursive Call - QuickSort(a,low,j-1)
    Recursive Call - QuickSort(a,j+1,high)
```

Problem Validation

Input:

```
(A)  enter size: 5

     enter elements: 5   4   10    13    6

(B)  enter size: 10

     enter elements: 9  8  7   6   4   0  3  2  1  5  10
```

Output:

```
(A) Sorted List is : 4  5  6  10  13

(B) Sorted List is : 1  2  3  4  5  6  7  8  9  10
```

Program 4

STACK DATA STRUCTURE

Problem Definition

Write a C++ program to implement Stack data structure using Linear List

Problem Description

A stack is an abstract data type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example - deck of cards or pile of plates etc. A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, We can only access the top element of a stack. This feature makes it LIFO data structure. LIFO stands for

Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Like an array, a linear list stores a collection of objects of a certain type, usually denoted as the elements of the list. The elements are ordered within the linear list in a linear sequence. Linear lists are usually simply denoted as lists. Stack can be implemented using various data structures like Linear list, linked list etc.

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.

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

- **topelement()** - get the top data element of the stack, without removing it.
- **isempty()** - check if stack is empty.

Stack uses a variable ”top” which always stores the index of the top element of the stack. It is initialized with -1 at the start.

Push Operation Steps:

1. If stack is Full, display that stack is full and can't push
2. Else
   a. Increment top by 1
   b. Place element to be pushed in arr[top] position

Pop Operation Steps:

1. If stack is empty, display that stack is empty and cannot pop
2. Else
   a. Copy the top element into x as x = arr[top]
   b. Decrement top by 1
   c. Return x as the popped element

Pseudocode

Use variable capacity to store the max. capacity of the stack Use variable arr to store the stack

```
Constructor

Stack(int c)
       capacity = c
       arr = new T[capacity]
       top = -1

Destructor
       ~Stack()
       delete [] arr

size()
       return top+1

isempty()
       If top==-1
              Return true
       Else
              Return false

push(element)
       if top < capacity-1
              arr[++top] = element
       else
              display "Stack is Full\n"
```

```
pop()
       if Stack is Empty
              Display "Stack is Empty. Cannot Pop\n"
       else
              return arr[top--]

topelement()
       if Stack is Empty
              Display "Stack is Empty. No top element\n"
       else
              return arr[top]

display()
       display "Stack contents are\n"
       if Stack is not empty
              for i = top to 0
                     display arr[i]
       else
              display "Nil\n"
```

Problem Validation

Input: None

Output:

```
 Stack Contents are:
       90        80        70        60        50        40

 Size of the stack is 6

 Popped element is 90

 Popped element is 80

 Stack Contents are:
       70        60        50        40

 Top element is 70

 Size of the Stack is 4
```

Program 5

QUEUE DATA STRUCTURE

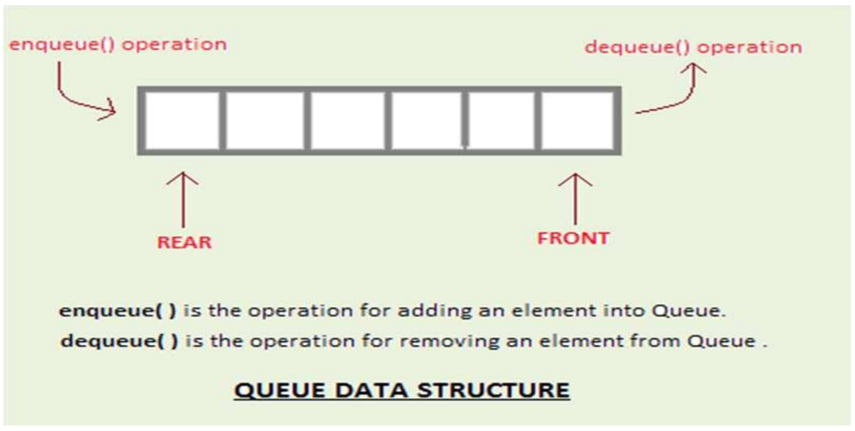
Problem Definition

Write a C++ program to implement Queue data structure using Linear List

Problem Description

Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called REAR(also called tail), and the deletion of existing element takes place from the other end called as FRONT(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.


enqueue() operation
dequeue() operation
REAR
FRONT
enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .
QUEUE DATA STRUCTURE

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 for queue are -

- **frontelement()** - get the element at front of the queue without removing it.
- **rearelement()** - get the element at rear end of the queue without removing it.
- **isfull()** - checks if queue is full. [condition rear > capacity then Q is full]
- **isempty()** - checks if queue is empty. [condition rear = = front = = -1]
- **size()** - returns the no. of elements in the queue.

In queue, we use variable ”front” to store the index just before the first element and variable ”rear” to store the index of last element in the queue. These variables are initialized to -1 at the start.

Enqueue Operation Steps:

1. If Queue is full then
   a. Display Queue is full, cannot insert element

2. Else
   a. Increment rear by 1
   b. Place new element at arr[front] position

Deqeue Operation Steps:

1. If Queue is empty then a. Display Queue is empty, cannot remove element

2. Else
   a. Increment front by 1
   b. Return arr[front] as the removed element

Pseudocode Use variable capacity to store the max. capacity of the queue

Use variable arr to store the queue

```
Constructor

Queue(int c)
       capacity = c
       arr = new T[capacity]
       front = -1
       rear = -1

Destructor
~Queue()
       delete [] arr;

size()
       if rear > front
              return rear-front
       else
               return front-rear

isempty()
       if front==-1 and rear==-1
              return true
       else
               return false
```

```
enqueu(element)
       f rear <= capacity
              inc rear by 1
       else
              arr[rear] = element

       display "Queue is Full"

dequeue()
       if Queue is empty
              display " Queue is Empty. Cannot remove"
       else
               inc front by 1
              return arr[front]

frontelement()
       if Queue is empty
              display "Queue is Empty. No front element"
       else
               return arr[front+1]

rearelement()
       if Queue is empty
              display "Queue is Empty. No rear element"
       else
               return arr[rear]

display()
       if Queue is not empty
              for i = front+1 to rear
                     display arr[i]
       else
              display "Nil"
```

Use variable capacity to store the max. capacity of the queue Use variable arr to store the queue

Problem Validation

Input: Execute the above pseudo code

Output:

```
Queue Contents are:
      90        80        70        60        50        40

Size of the Queue is 6

Removed element is 90

Removed element is 80

Queue Contents are:
        70        60        50        40

Front element is 70

Rear element is 40

Size of the Queue is 4
```

Program 6

CIRCULAR QUEUE DATA STRUCTURE

Problem Definition

Write a C++ program to implement Circular Queue data structure using Linear List

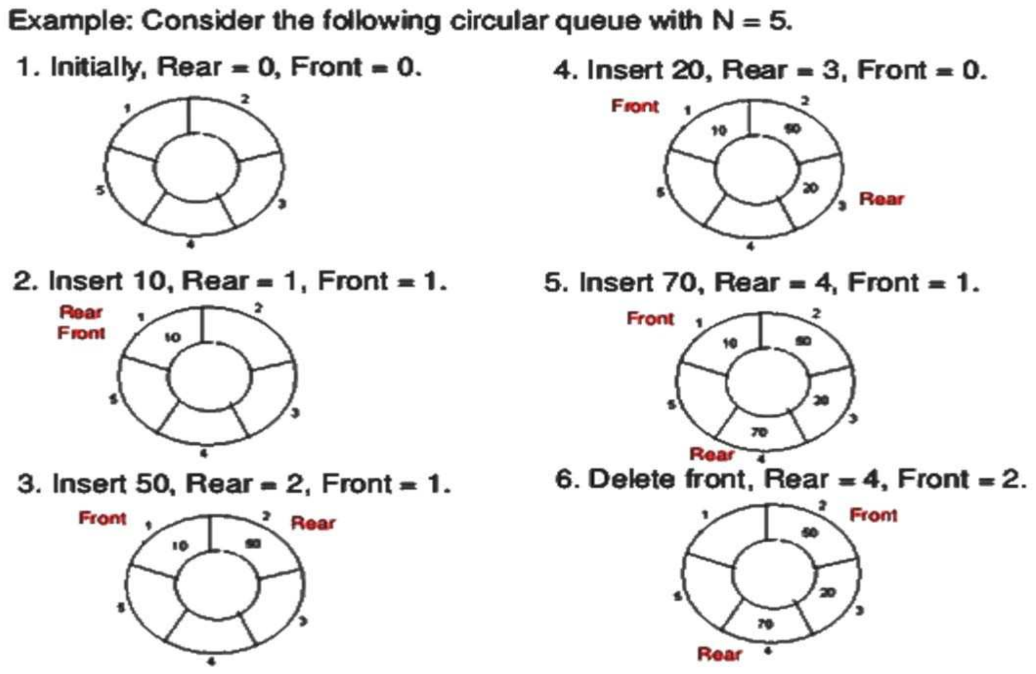
Problem Description

The disadvantage of queue is that when rear variable value reaches end of the queue i.e capacity-1, then even if there is space in queue, we will not be able to insert new elements as Queue full condition will be satisfying. To rectify this problem, we use circular queue.

In circular queue, last element is connected to first element. When rear reaches the end of Queue, then it can move in circular motion to front of the queue.

Initially, front and rear variables are assigned to 0.
To increment front and rear variables in circular motion, following formulae are used:

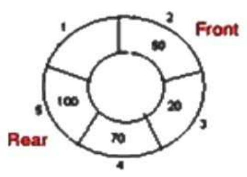
front=(front+1) % capacity
rear = (rear+1) % capacity



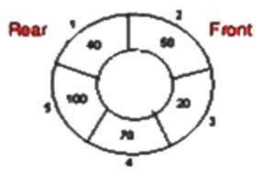
Example: Consider the following circular queue with N = 5.

1. Initially, Rear = 0, Front = 0.
2. Insert 10, Rear = 1, Front = 1.
3. Insert 50, Rear = 2, Front = 1.
4. Insert 20, Rear = 3, Front = 0.
5. Insert 70, Rear = 4, Front = 1.
6. Delete front, Rear = 4, Front = 2.

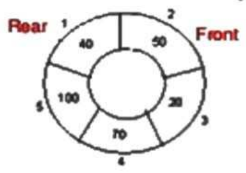
7. Insert 100, Rear = 5, Front = 2.



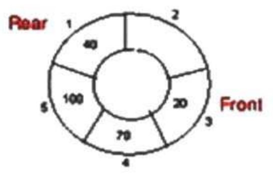
8. Insert 40, Rear = 1, Front = 2.



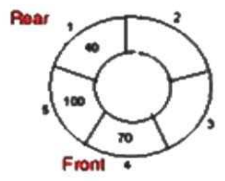
9. Insert 140, Rear = 1, Front = 2.
As Front = Rear + 1, so Queue overflow.



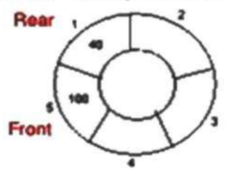
10. Delete front, Rear = 1, Front = 3.



11. Delete front, Rear = 1, Front = 4.



12. Delete front, Rear = 1, Front = 5.



Enqueue Operation Steps:

1. If Queue is full then
   a. Display Queue is full, cannot insert element
2. Else
   a. Increment rear by 1 circularly using rear=(rear+1) % capacity
   b. Place new element at arr[front] position

Dequeue Operation Steps:

1. If Queue is empty then
   a. Display Queue is empty, cannot remove element
2. Else
   a. Increment front by 1 circularly using front=(front+1) % capacity
   b. Return arr[front] as the removed element

Pseudocode

Use variable i to store the size of the circular queue Use variable capacity to store the max. capacity of the circular queue Use variable arr to store the circular queue

```
Constructor
Queue(c)
        capacity = c
        arr = new T[capacity]
        front = 0; rear = 0; i = 0;

Destructor
~Queue()
        delete [] arr

size()
        return i

isempty()
      if front==rear
            return true
      else
            return false

push(element)
      if circular queue is full then
            display " circular queue is Full"
      else
            rear = (rear+1) % capacity
      arr[rear] = elem
      inc i by 1

pop()
      if circular queue is empty then
            display "Circular Queue is Empty. Cannot remove"
      else
            front = (front+1) % capacity
      return arr[front]

frontelement()
      if circular queue is empty then
            display "Circular Queue is Empty. No front element"
      else
            return arr[(front+1) % capacity]

rearelement()
      if circular queue is empty then
            display "Circular Queue is Empty. No rear element"
      else
             return arr[rear]
```

```
display()
      if circular queue is empty then
            display "Circular Queue is Empty"
      else
            j = (front+1) % capacity
            while j!=rear
                  display arr[j]
                  j = (j+1) % capacity
            display arr[rear]
```

Problem Validation

Input: None

Output:

```
Queue Contents are:
90        80        70        60        50        40

Size of the Queue is 6

Removed element is 90

Removed element is 80

Queue Contents are:
70        60        50        40

Front element is 70

Rear element is 40

Size of the Queue is 4
```

Program 7

INFIX TO POSTFIX CONVERSION

Problem Definition

Write a C++ program to implement Infix expression to Postfix Conversion using Stacks

Problem Description

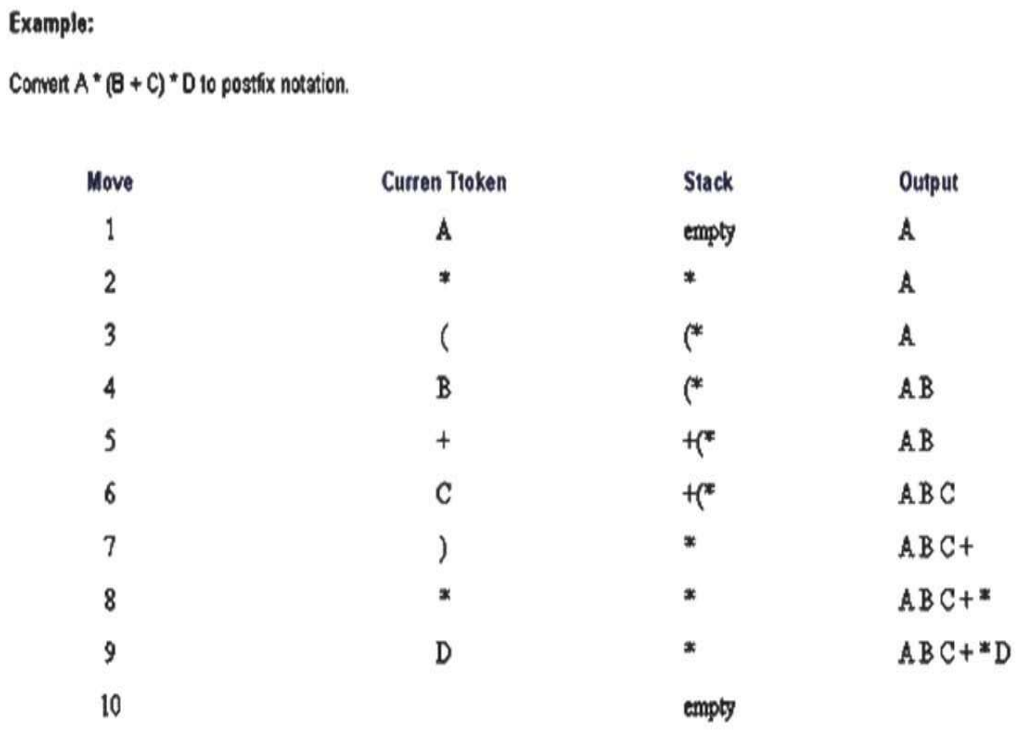
Infix expression is the expression of the form a op b. When an operator is in-between every pair of operands. The Postfix expression is the expression of the form a b op. When an operator is written after its operands, then it is called as postfix expression.

Why postfix representation of the expression? The compiler scans the expression either from left to right or from right to left.

Consider the below expression: a + b * c - d
The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then subtracted from d after another scan. The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation. The corresponding expression in postfix form is: abc*d++. The postfix expressions can be evaluated easily using a stack.

Infix expression to Postfix Conversion Algorithm

1. Read the infix expression from the user.
2. Scan the infix expression from left to right.
3. Let the current infix character is x
4. If x is an operand(i.e digit), display it.
5. Else If x is an '(', push it to the stack.
6. Else If x is an ')', pop and output from the stack until an '(' is encountered.
7. Else If the precedence of x is less than or equal to the precedence of stack top,then
    a. Repeatedly pop and display stack elements as long as precedence of x is less than or equal to the precedence of stack top
    b. Then, Push x onto the stack.
8. Repeat steps 4-7 until infix expression is scanned.
9. Pop and output from the stack until it is not empty.



Example:

Convert A * (B + C) * D to postfix notation.

Move	Curren Ttoken	Stack	Output
1	A	empty	A
2	*	*	A
3	(	(*	A
4	B	(*	AB
5	+	+(*	AB
6	C	+(*	ABC
7	)	*	ABC+
8	*	*	ABC+*
9	D	*	ABC+*D
10		empty	

Pseudocode

Read infix expression from user and store it in char array infix[]
Declare a stack st of size 50

```
st.push('#')

len =  strlen(infix)

display "Postfix expression is"

for i = 0 to len-1
       x = infix[i]
       if x=='('
              st.push(x)
       else if x==')'
              while stack top != '('
                     display popped element st.pop()
              st.pop()
```

```
    else if x is a digit
             display x
         else
             while precendence(x) <= precendence(stack top)
                    display popped element st.pop()
             st.push(x)
while stack is not empty
     display popped element st.pop()
precendence(x)
   '*' , '/' := (2)
   '+',  '-' := (1)
   '(', '#'  := (0)
```

Problem Validation

```
Input:
        Enter the infix expression: (a+b) * (c-d) / e

Output:

        Postfix expression is : ab + cd-*e/
```

Program 8

POSTFIX EVALUATION

Problem Definition

Write a C++ program to implement Postfix Evaluation using stacks

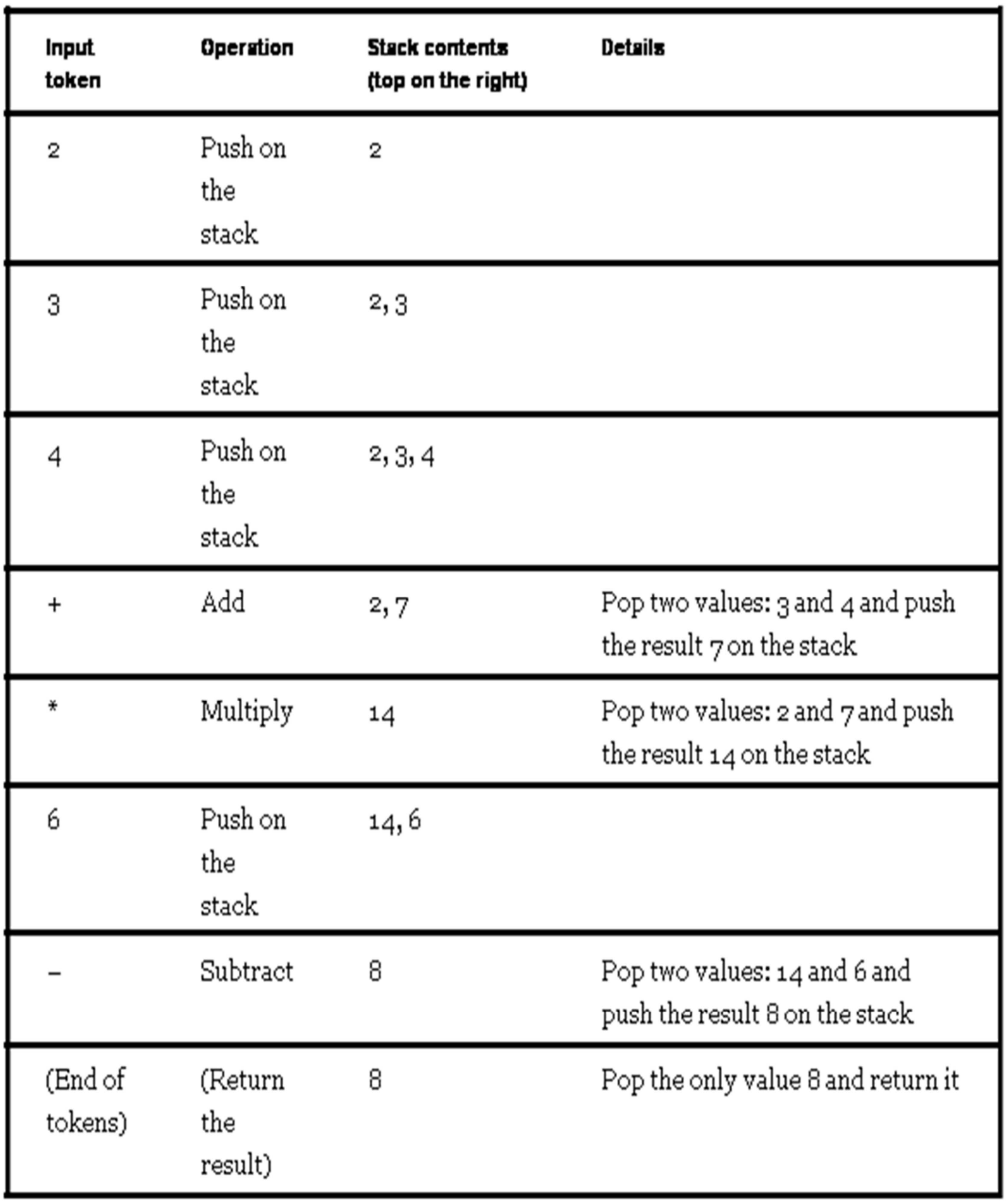
Problem Description

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

Following is algorithm for evaluation postfix expressions.

1. Create a stack to store operands (or values).
2. Read the postfix expression from the user
3. Scan the given postfix expression from left to right
4. Let x be the current postfix character
   a. If the element is a number, push it into the stack
   b. If the element is a operator, pop 2 operands from stack. Evaluate the operator and push the result back to the stack
5. When the postfix expression is ended, the number in the stack is the final answer

```
Example:
Evaluate the postfix expression  2  3  4  +  *  6  -
```



Input token	Operation	Stack contents (top on the right)	Details
2	Push on the stack	2	
3	Push on the stack	2, 3	
4	Push on the stack	2, 3, 4	
+	Add	2, 7	Pop two values: 3 and 4 and push the result 7 on the stack
*	Multiply	14	Pop two values: 2 and 7 and push the result 14 on the stack
6	Push on the stack	14, 6	
-	Subtract	8	Pop two values: 14 and 6 and push the result 8 on the stack
(End of tokens)	(Return the result)	8	Pop the only value 8 and return it

Result is 8

Pseudocode

```
Read postfix expression from the user

Declare a stack st of some size
len = strlen(postfix)
```

```
for  i = 0 to len-1
       x = postfix[i]
       if x is a digit
              st.push(x='0')
       else
              a = st.pop()
              b = st.pop()

       switch(x)
              case '+':
                     c = b+a;
                     st.push(c);
              case '-':
                     c = b-a;
                     st.push(c);
              case '*':
                     c = b*a;
                     st.push(c);
              case '/':
                     c = b/a;
                     st.push(c);
       display "result is" st.pop()
```

Problem Validation

Input: Enter the postfix expression: 2 4 + 3 - 2 * 3 /

Output: result is 2

Program 9

LINKED LIST

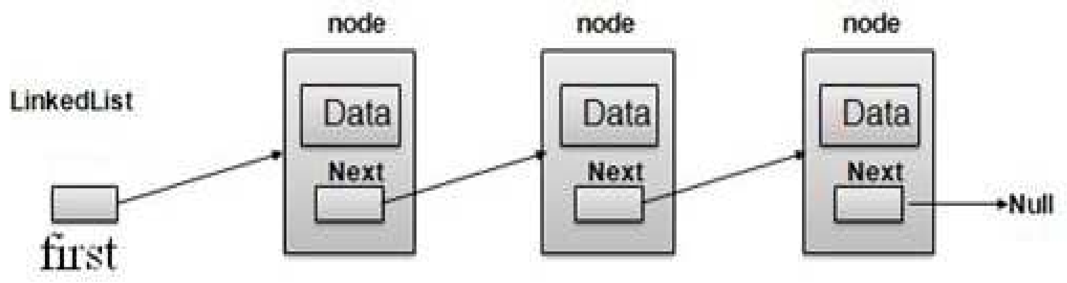
Problem Definition

Write a C++ program to implement Linked List

Problem Description

Linked List is a sequence of nodes which contains date elements. Each node has two parts:

- **Data** - it stores data i.e an element.
- **Next** - it contains a pointer/link to next node.



The first node is pointed to by a pointer called ”first”.

Basic Operations

Following are the basic operations supported by a linked list:-

- **Constructor** - initializes pointer first as NULL to indicate linked list is initially empty
- **Insertion** - add an element at the given position in the linked list.
- **Deletion** - delete an element from the given position in the linked list.
- **Display** - displaying complete linked list.
- **isempty** - checks whether the linked list is empty or not. [condition: first==NULL]
- **Length** - finds the number of nodes in the linked list.
- **Destructor** - deletes/frees memory used by all the nodes of the linked list

Insertion Operation: Inserts the given element at given position pos

1. Create a new node ”temp” to store the new element to be inserted.
2. Initialize data part of temp to element value and next part to NULL.
3. If element is to be inserted at position 1 i.e beginning of the list then,
   - Make temp– >next point to first
   - Make first point to temp
4. Else, i.e element is to be inserted in middle or end of the list,
   - Take another pointer p and using a for loop make it point to (pos-1)th node
   - Make temp– >next point to p– >next
   - Make p– >next point to temp

Removal Operation: Removes and returns the element from given position pos=> remove(pos)

1. Declare node pointer ”temp” and some variable x. temp will always point to the node to be removed and x stores the data of the removed element.
2. If element is to be removed from position 1 i.e first node, then
   - Make temp point to first node
   - Make first point to next node i.e second node
   - Store the data of temp node in x
   - Delete temp node
3. Else, i.e if the element is to be removed from middle or end, then,
   - Take another pointer p and using a for loop make it point to (pos-1)th node
   - Make temp point to p– >next
   - Make p– >next point to temp– >next
   - Store the data of temp node in x
   - Delete temp node
4. Return the x value as the removed element

Pseudocode

```
Constructor

Chain()
        First = NULL

destructor
        ~Chain()
        Node<T> *temp = first
        While temp!=NULL
                first = first->next
                delete temp
                temp = first

isempty()
       if first == NULL
              return true
       else
              return false

insert(element,pos)
       Node<T> *temp = new Node<T>
       temp->data = element
       temp->next = NULL
       if pos == 1
              temp->next = first
              first = temp
       else
              Node<T> *p = first
              for i=1 to pos-1
                     p = p->next
                     temp->next = p->next
                     p->next = temp

remove(pos)
       Node<T> *temp
       T x

       If pos == 1
              temp = first
              first = first->next
              x = temp->data
              delete temp
       else
              Node<T> *p = first
              for i=1 to pos-1
```

```
                    p = p->next
                    temp = p->next
                    p->next = temp->next
                    x = temp->data
                    delete temp
             return x

display()
       Node<T> *temp = first
       While temp!=NULL
             Display temp->data
             temp = temp->next

length()
       Node<T> *temp = first
       i = 0
       While temp!=NULL
             temp = temp->next
             inc i by 1
       return i
```

Problem Validation

Input: None

Output:

```
Queue Contents are:
        90        80        70        60        50        40

Size of the Queue is 6

Removed element is 90

Removed element is 80

Queue Contents are:
70        60        50        40

Front element is 70

Rear element is 40

Size of the Queue is 4
```

Program 10

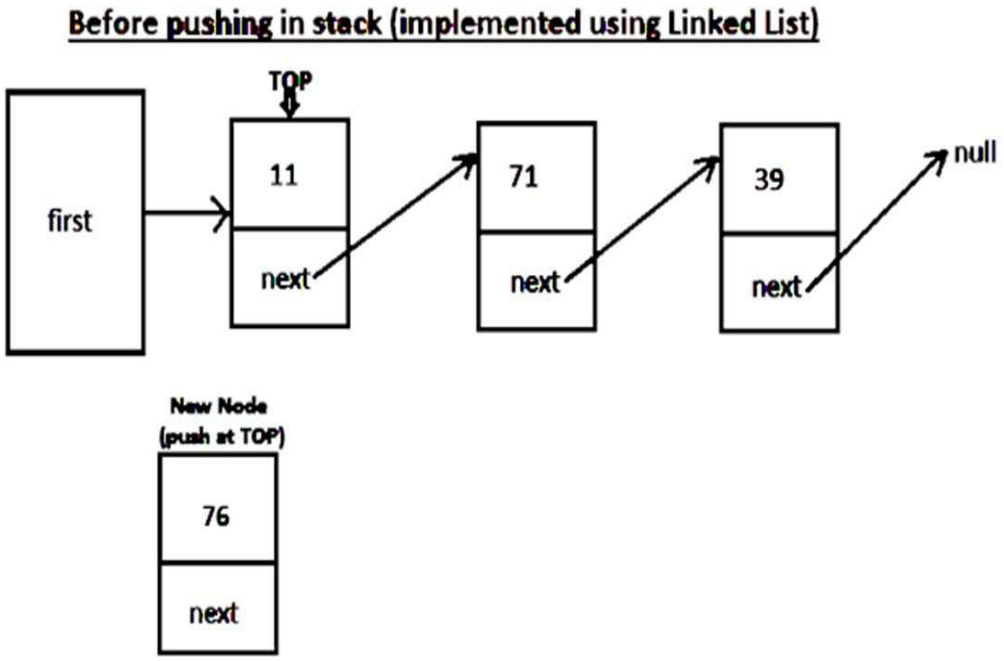
LINKED STACK

Problem Definition

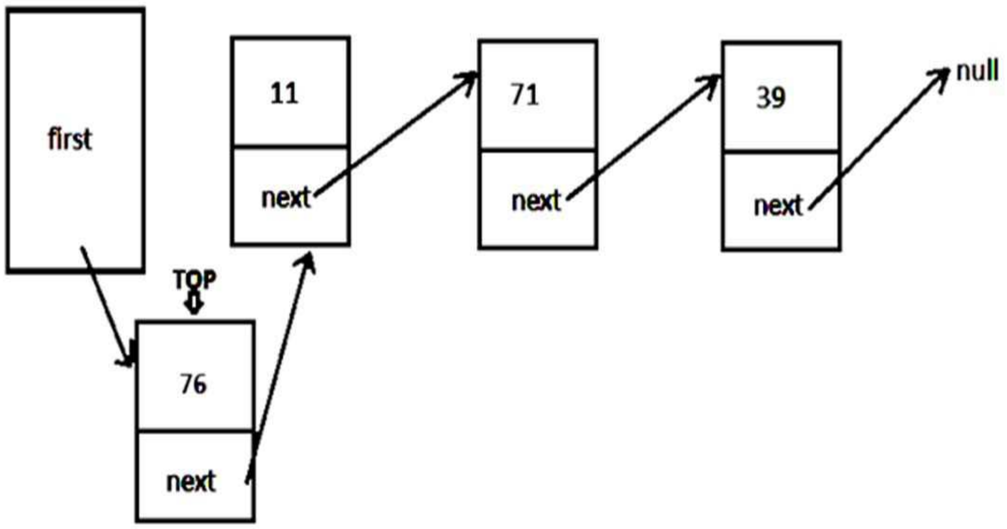
Write a C++ program to implement Linked Stack

Problem Description

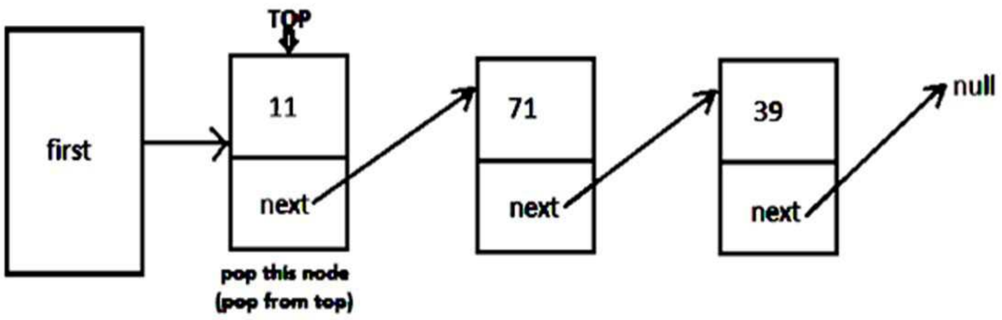
Stack data structure can also be implemented using linked list. The "top" pointer is used to point to the top of the stack. The first node is always considered as top of the stack. In stack, we can only insert and delete elements from top of the stack. It means that in Linked Stack, we can insert and delete nodes only from the beginning of the linked stack.



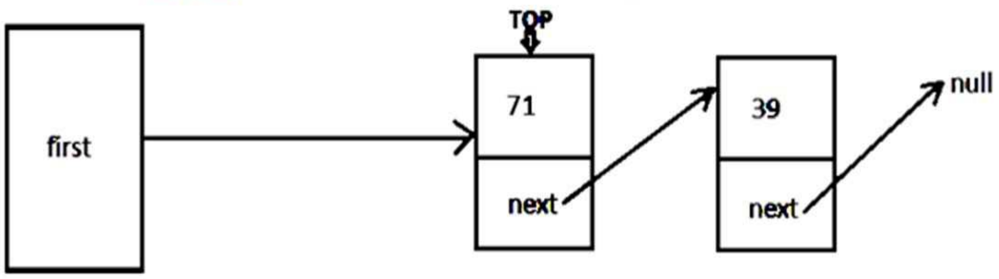
After pushing in stack (implemented using Linked List)



Before popping in stack (implemented using Linked List)



After popping in stack (implemented using Linked List)



Pseudocode

```
Constructor

LinkedStack()
        top = NULL

destructor
~LinkedStack()
        Node<T> *temp = top
        While temp!=NULL
                top = top->next
                delete temp
                temp = top


isempty()
if top == NULL
                return true
        else
                return false


push(element)
Node<T> *temp = new Node<T>
        temp->data = element
        temp->next = top
        top = temp


topelement()
if Stack is empty then
                display "Stack is empty"
        else
                 return top->data


 pop()
if Stack is empty then
                display "Stack is empty"
         else
                Node<T> *temp = top
                top = top->next
                x = temp->data
                delete temp
                return x
```

```
display()
        Node<T> *temp = top
        While temp!=NULL
                Display temp->data
                temp = temp->next

length()
        Node<T> *temp = top
        i = 0
        While temp!=NULL
                temp = temp->next
                inc i by 1
        return i
```

Problem Validation

Input: None

Output:

```
Linked Stack Contents are:
     90     80     70     60     50     40

Size of the stack is 6

Popped element is 90

Popped element is 80

Linked Stack Contents are:
     70     60     50     40

Top element is 70

Size of the Linked Stack is 4
```

Program 11

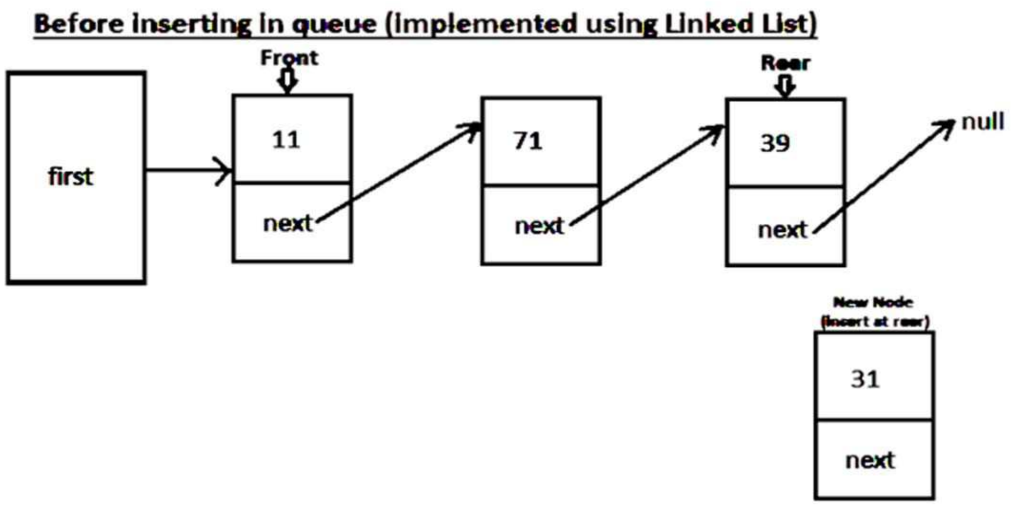
LINKED QUEUE

Problem Definition

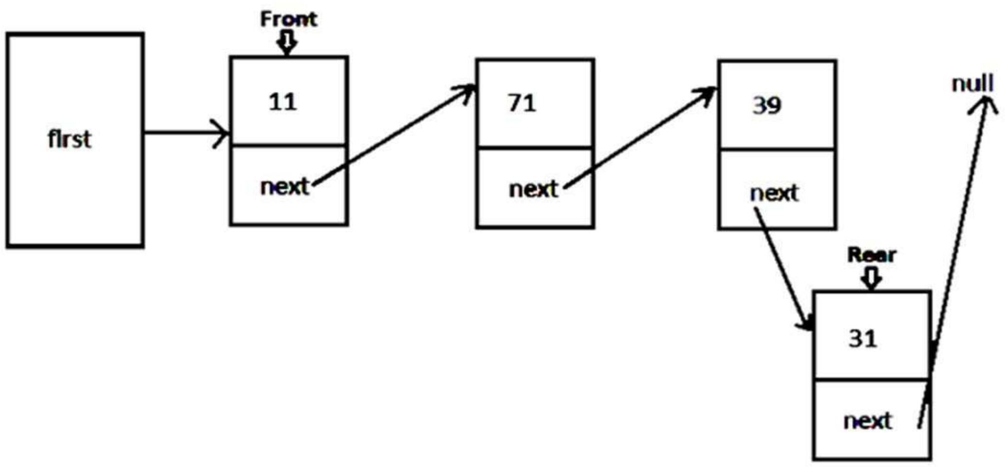
Write a C++ program to implement Linked Queue

Problem Description

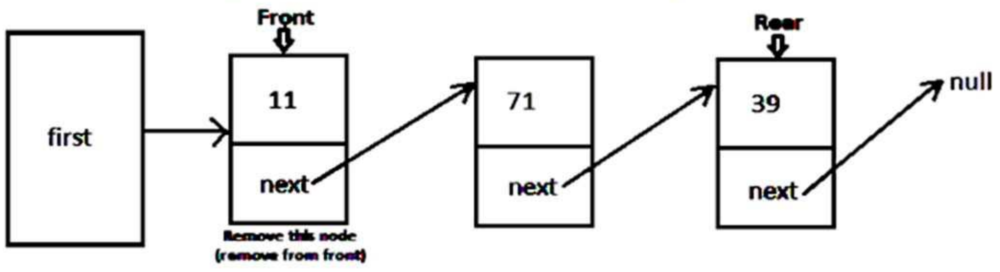
Queue data structure can also be implemented using a linked list. The “front” pointer points to the first node of the linked queue which is considered as the front end of the queue. The deletions can be done only at the front end of the linked queue. The “rear” pointer points to the last node of the linked queue which is considered as the rear end of the queue. The insertion of new elements can be done only at rear end of linked queue.



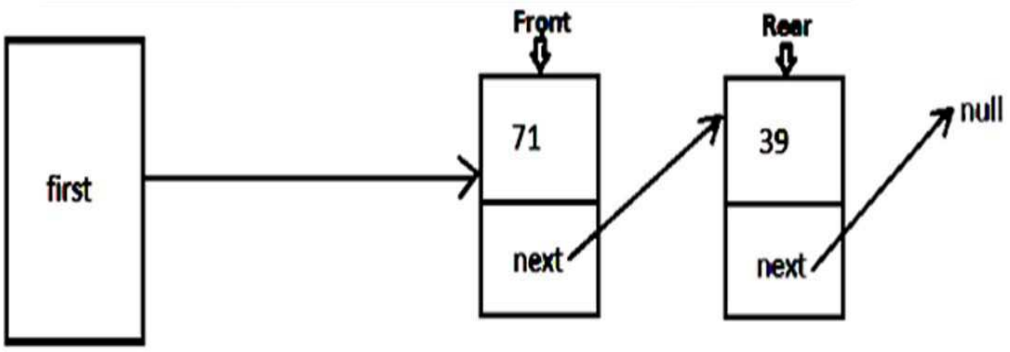
After inserting in queue (implemented using Linked List)



Before removing from queue (implemented using Linked List)



After removing from queue (implemented using Linked List)



Pseudocode

```
Constructor
LinkedQueue()
        Front = NULL
        Rear = NULL

destructor
~LinkedQueue()
        Node<T> *temp=front
        While temp!=NULL
                front = front->next
delete temp
                temp = front

isempty()

        if front == NULL
                return true
        else
                return false

display()
        Node<T> *temp = front
        While temp!=NULL
                Display temp->data
                temp = temp->next

length()
        Node<T> *temp = front
        i = 0
        while temp!=NULL
                temp = temp->next
                inc i by 1
        return i

insert(element)
        Node<T> *temp = new Node<T>
        temp->data = elem
        temp->next = NULL
        if rear == NULL
                front = temp
                rear = temp
        else
                rear->next = temp
                rear = temp
```

```
remove()
        if Queue is empty then
                display "queue is empty"
        else
                Node<T> *temp = front
                front = front->next
                x = temp->data
                delete temp
                return x

frontelement()
        if Queue is empty then
                display "queue is empty"
        else
                return front->data

rearelement()
        if Queue is empty then
                display "queue is empty"
        else
                return rear->data
```

Problem Validation

Input: None

Output:

```
Queue Contents are:
     90     80     70     60     50     40

Size of the Queue is 6

Removed element is 90

Removed element is 80

Queue Contents are:
     70     60     50     40

Front element is 70

Rear element is 40

Length of the Queue is 4
```

Program 12

CIRCULAR LINKED LIST

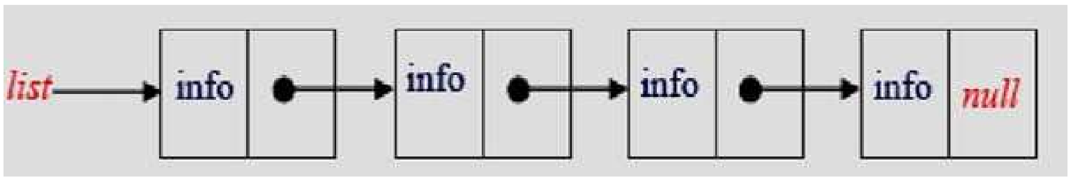
Problem Definition

Write a C++ program to implement Circular Linked List

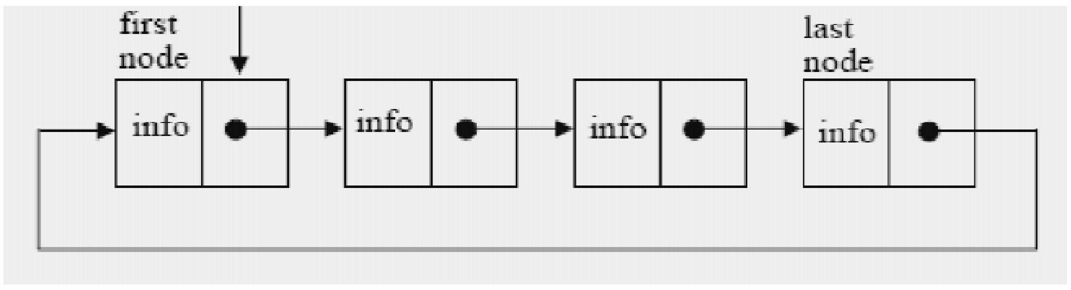
Problem Description

Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.

Linked List:



Circular Linked List:



Each node has 2 fields:

- **Data field** :=> stores the element value
- **Next pointer** :=> points to the next node in the list

The first node is pointed to by a pointer called "first".

Basic Operations

Following are the basic operations supported by a linked list:-

- **Constructor** - initializes pointer first as NULL to indicate circular linked list is initially empty
- **Insertion** - add an element at the given position in the circular linked list.
- **Deletion** - delete an element from the given position in the circular linked list.
- **Display** - displaying complete circular linked list.
- **isempty** - checks whether the linked list is empty or not. [condition: first==NULL]
- **Length** - finds the number of nodes in the circular linked list.
- **Destructor** - deletes/frees memory used by all the nodes of the circular linked list

Insertion Operation: Inserts the given element at given position pos

1. Create a new node "temp" to store the new element to be inserted.
2. Initialize data part of temp to element value and next part to NULL.
3. If circular linked list is empty i.e len is 0 and we need to insert the first node in the list then
   a. Make first point to newly created node temp
   b. Make temp− >next point back to first
4. If element is to be inserted at position 1 i.e beginning of the list then,
   a. Make temp− >next point to first
   b. Take a pointer p and using for loop make it point to last node in the list
   c. Make first point to temp
   d. Make p− >next(i.e last node) point to first
5. Else, i.e element is to be inserted in middle or end of the list,
   a. Take a pointer p and using a for loop make it point to (pos-1)th node
   b. Make temp− >next point to p− >next
   c. Make p− >next point to temp
6. Increment "len" by 1

Removal Operation: Removes and returns the element from given position pos=> remove(pos)

1. Declare node pointer "temp" and some variable x. temp will always point to the node to be removed and x stores the data of the removed element.
2. If the circular linked list has only one node (i.e len is 1) and we need to delete that node, i.e after deletion, list will become empty, then
   a. Assign first to NULL
   b. Make temp point to first pointer
   c. Store the data of temp node in x
   d. Delete temp node
3. If element is to be removed from position 1 i.e first node , then
   a. Take a pointer p and using for loop make it point to last node in the list
   b. Make temp point to first pointer
   c. Make first point to next node i.e second node
   d. Make last node i.e (p− >next) point to first node
   e. Store the data of temp node in x
   f. Delete temp node
4. Else, i.e if the element is to be removed from middle or end, then
   a. Take a pointer p and using a for loop make it point to (pos-1)th node
   b. Make temp point to p− >next
   c. Make p− >next point to temp− >next
   d. Store the data of temp node in x
   e. Delete temp node

Pseudocode

```
Constructor
CircularChain()
        first =NULL
        len = 0

destructor
~CircularChain()
        Node<T> *temp = first
        for i=1 to len
                first = first->next
                delete temp
                temp = first
```

```
isempty()
        if first == NULL
                return true
        else
                return false

display()
        Node<T> *temp = first
        For  i=1 to
                Display temp->data
                temp = temp->next

length()
        return len

insert(element,pos)
        Node<T> *temp = new Node<T>
        temp->data = elem
        temp->next = NULL
        if len == 0
                first = temp
                temp->next = first
        else if pos == 1
                temp->next = first
                Node<T> *p = first
                for i=1 to len-1
                        p = p->next
                        first = temp
             else
                        len++
                        p->next = first
                        Node<T> *p = first
                        for i=1 to pos-2
                                p = p->next
                                temp->next  = p->next
                                p->next = temp

remove(int pos)
        Node<T> *temp = first
        if len == 1
                first = NULL
                x = temp->data
                delete temp
        else if pos == 1
                Node<T> *p = first
                for i=1 to len-1
                        p = p->next
                        first = first->next
                        p->next = first
                        x = temp->data
                        delete temp
                else
                        len--
                        Node<T> *p = first
                        for i=1 to pos-2
                                p = p->next
                                temp = p->next
                                p->next = temp->next
                                x = temp->data
                                delete temp
                                return x
```

Problem Validation

```
Input:   None

Output:

        Circular Linked List Contents :
                10  20  30  40  50

        Removed element from position 2 is:20

        Length of the Circular Linked List is 4

        Circular Linked List Contents : 10   30  40  50
```

Program 13

DOUBLY LINKED LIST

Problem Definition

Write a C++ program to implement Doubly Linked List

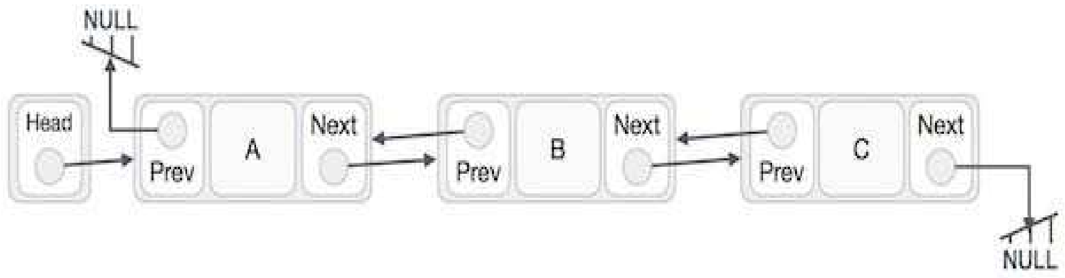
Problem Description

A doubly linked list is a list that contains pointer links to next and previous nodes. Unlike singly linked lists where traversal is only one way, doubly linked lists allow traversals in both ways.

Each node has 3 fields:

- **Prev pointer** :=> points to the previous node in the list
- **Data field** :=> stores the element value
- **Next pointer** :=> points to the next node in the list

TThe “first” pointer points to the first node in the list and the “last” pointer points to the last node in the list.



Basic Operations

Following are the basic operations supported by a linked list:-

- **Constructor** - initializes pointer first as NULL, last as NULL and len as 0 to indicate doubly linked list is initially empty
- **Insertion** - add an element at the given position in the doubly linked list.
- **Deletion** - delete an element from the given position in the doubly linked list.
- **Display** - displaying complete doubly linked list.

- **isempty** - checks whether the doubly linked list is empty or not. [condition => first==NULL and last == NULL]
- **Length** - finds the number of nodes in the doubly linked list.
- **Destructor** - deletes/frees memory used by all the nodes of the doubly linked list

Insertion Operation: Inserts the given element at given position pos

1. Create a new node "temp" to store the new element to be inserted
2. Initialize data part of temp to element value and prev part and next part to NULL.
3. If doubly linked list is empty i.e len is 0 and we need to insert the first node in the list then
   a. Make both "first" and "last" pointers point to newly created node "temp"
4. If element is to be inserted at position 1 i.e beginning of the list then,
   a. Make temp− >next point to first
   b. Make first− >prev point to temp
   c. Make first point to temp
5. If element is to be inserted at last position i.e len+1 then,
   a. Make temp− >next point to last
   b. Make last− >next point to temp
   c. Make last point to temp
6. Else, i.e element is to be inserted in middle of the list,
   a. Take a pointer p and using a for loop make it point to (pos-1)th node
   b. Take a pointer q and make it point to (pos)th node i.e p− >next
   c. Make p− >next point to temp
   d. Make p− >next point to temp
   e. Make temp− >prev point to p
   f. Make temp− >next point to q
7. Increment "len" by 1

Removal Operation: Removes and returns the element from given position pos=> remove(pos)

1. Declare node pointer "temp" and some variable x. temp will always point to the node to be removed and x stores the data of the removed element.
2. If the doubly linked list has only one node (i.e len is 1) and we need to delete that node, i.e after deletion, list will become empty, then

   a. Assign first,last to NULL
   b. Make temp point to first pointer
   c. Store the data of temp node in x
   d. Delete temp node

3. If element is to be removed from position 1 i.e first node , then
   a. Make temp point to first pointer
   b. Make first point to next node i.e second node
   c. Make first− >prev point to NULL
   d. Store the data of temp node in x
   e. Delete temp node

4. If element is to be removed from last position i.e len , then
   a. Make temp point to last pointer
   b. Make last point to prev node i.e last− >prev
   c. Make last− >next point to NULL
   d. Store the data of temp node in x
   e. Delete temp node

5. Else, i.e if the element is to be removed from middle, then
   a. Take a pointer p and using a for loop make it point to (pos-1)th node
   b. Make temp point to p− >next
   c. Make q point to temp− >next
   d. Make p− >next point to q
   e. Make q− >prev point to p
   f. Store the data of temp node in x
   g. Delete temp node

6. Return the x value as the removed element

7. Decrement “len” by 1

Pseudocode

```
Constructor
DoublyChain()
        first = NULL
        last = NULL
        len = 0

~DoublyChain()
        Node<T> *temp = first

        For i=1 to len
                first = first->next
                delete temp
                temp = first

isempty()
        if first is NULL and last is NULL
                return true
        else
                 return false

display()
        Node<T> *temp = first
        For i=1 to len
                Display temp->data
                temp = temp->next

length()
        return len

insert(element,pos)
        Node<T> *temp=new Node<T>
        temp->data = element
        temp->prev = NULL
        temp->next = NULL
        if len is 0
                first = temp
                last = temp
        else if pos is 1
                temp->next = first
                first->prev = temp
                first = temp
             else if pos is len+1
                        temp->prev = last
                        last->next = temp
                        last = temp
```

```
                  else
                        Node<T> *p = first
                        For i=1to pos-2
                                p = p->next
                                Node<T> *q = p->next
                                p->next = temp
                                q->prev = temp
                                temp->prev = p
                                len++
                                temp->next = q

remove(pos)
        Node<T> *temp
        T x

        If len is 1
                temp = first
                first = NULL
                last = NULL
                x = temp->data
                delete temp
        else if pos is 1
                temp = first
                first = first->next
                first->prev = NULL
                x = temp->data
                delete temp
             else if pos is len
                        temp = last
                        last = last->prev
                        last->next = NULL
                        x = temp->data
                        delete temp
                  else
                        Node<T> *p = first
                        Node<T> *q
                        For i=1 to pos-2
                                p = p->next
                        temp = p->next
                        q = temp->next
                        p->next = q
                        q->prev = p
                        x = temp->data
                        delete temp
                return x
                len--
```

Problem Validation

```
Input:   None

Output:

        Doubly Linked List Contents :
                10  20  30  40  50

        Removed element from position 2 is:20

        Length of the Doubly Linked List is 4

        Doubly Linked List Contents : 10   30  40  50
```

Program 14

POLYNOMIAL REPRESENTATION USING LINKED LIST

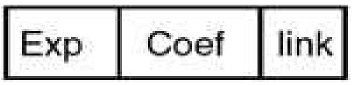
Problem Definition

Write a C++ program to implement Polynomial Addition using Linked List

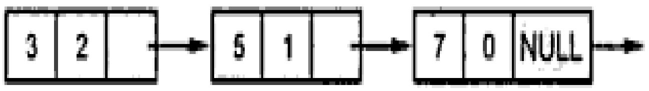
Problem Description

A Polynomial has mainly two fields. exponent and coefficient.

Node of a Polynomial:



For example $3x^2$ + 5x + 7 will represent as follows.



In each node the exponent field will store the corresponding exponent and the coefficient field will store the corresponding coefficient. Link field points to the next item in the polynomial.

Algorithm to add two polynomials using Linked Lists:

1. Take two pointers ax and bx. Make ax point to first term in Polynomial A and make bx point to first term in Polynomial B
2. Declare polynomial C to store the added polynomial A+B
3. While ax and bx both are not NULL i.e didn't reach the end of A and B, then
   a. If Exponent of ax is equal to Exponent of bx, then
      i. Find the Sum of coefficients of ax and bx

  ii. If sum > 0,then insert the new term (exp of ax or bx, sum) into polynomial C
  iii. Make ax and bx point to next term in polynomials A,B

b. Else if Exponent of ax ¿ Exponent of bx, then
  i. Insert the term from polynomial A into polynomial C
  ii. Make ax point to next term in A

c. Else, i.e Exponent of ax ¡ Exponent of bx, then
  i. Insert the term from polynomial B into polynomial C
  ii. Make bx point to next term in B

4. The above while loop ends if either we reached the end of polynomial A and few terms are remaining in polynomial B (OR) we reached the end of polynomial B and few terms are remaining in polynomial A
5. Add the remaining terms from polynomial A, if any
6. Add the remaining terms from polynomial B, if any
7. Return the resultant polynomial C

Pseudocode

```
Constructor
Poly()
        first = NULL
        last = NULL

insertback(e, c):inserts new term (exponent, coefficient) at the end
of polynomial
        term<T> *temp = new term<T>
        temp->coeff = c temp->exp = e
        temp->next = NULL
        if last is NULL
                first = temp
                last = temp
        else
                last->next = temp
                last = temp

operator+(Poly<T> &b)
        term<T> *ax = first
        term<T> *bx = b.first
        Poly<T> c
        While ax!=NULL and bx!=NULL
                If ax->exp is equal to bx->exp
                Sum = ax->coeff + bx->coeff
                If sum > 0
                        c.insertback(ax->exp,sum)
                        ax = ax->next
                        bx = bx->next
                else if ax->exp < bx->exp
                        c.insertback(bx->exp,bx->coeff)
                        bx = bx->next
                else
                        c.insertback(ax->exp,ax->coeff)
                        ax = ax->next
        while ax != NULL
                c.insertback(ax->exp,ax->coeff)
                ax = ax->next

        while bx!=NULL
                c.insertback(bx->exp,bx->coeff)
                bx = bx->next
        return c

display()
        for term<T> *temp = first;temp!=NULL;temp=temp->next
                display temp->exp and temp->coeff
```

Problem Validation

```
Input:

        Inserting terms (5,1), (3,2), (1,2), (0,7) in polynomial A
Inserting terms (4,2), (3,5), (2,4), (1,3) in polynomial B

Output:
        Polynomial after addition is
        Exp.      Coeff

        5           1

        4           2

        3           7

        2           4

        1           5

        0           7
```

Program 15

SPARSE MATRIX REPRESENTATION USING LINKED LIST

Problem Definition

Write a C++ program to implement Sparse Matrix using Linked Representation

Problem Description

A matrix is a two-dimensional data object made of m rows and n columns, therefore having m X n values. The most natural representation is to use two-dimensional array A[m][n] and access the element of ith row and jth column as A[i][j]. If a large number of elements of the matrix are zero elements, then it is called a **sparse matrix**.
Representing a sparse matrix by using a two-dimensional array leads to the wastage of a substantial amount of space. Therefore, an alternative representation must be used for sparse matrices. One such representation is to store only non- zero elements along with their row positions and column positions. That means representing every non-zero element by using triples (i, j, value), where i is a row position and j is a column position.

To represent a sparse matrix using linked lists, we use 4 types of nodes

1. **Head node :**

No. of rows in sparse matrix (**row**)	No. of Columns in sparse matrix (**col**)	No. of Non Zero values in sparse matrix (**value**)
Pointer to first Row node (**down**)		Pointer to first column node (**right**)

2. **Row Node :**

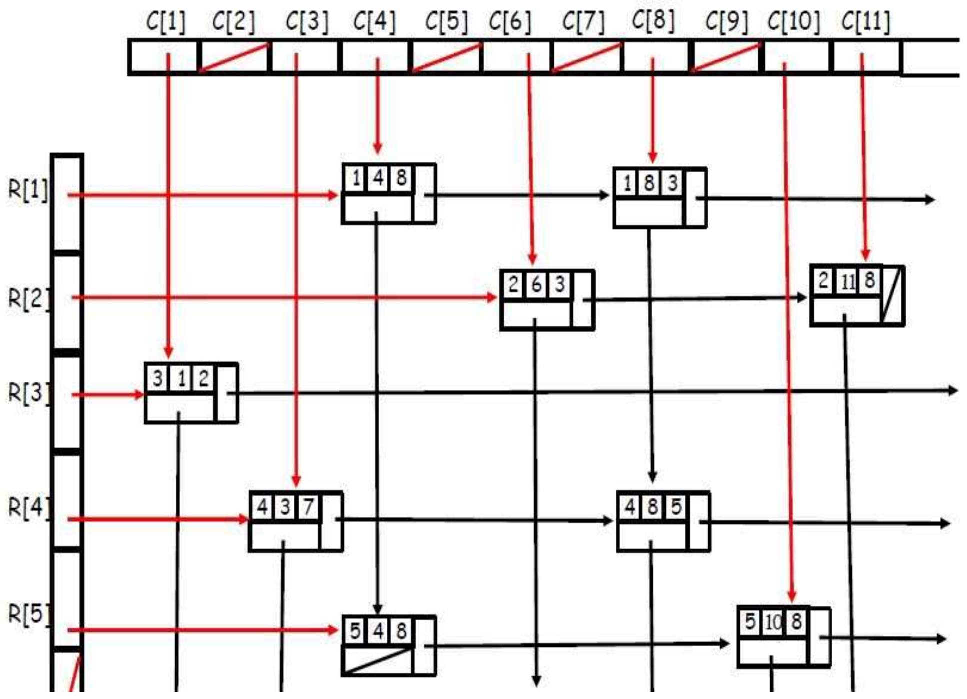
Pointer to next row node(**next**)	
Not used (**down**)	Pointer to data node in this row (**right**)

3. **Column node :**

Pointer to next column node(**next**)	
Pointer to data node in this column (**down**)	Not used (**right**)

4. **Data node :**

Row no. of non zero value (**row**)	Column no. of non zero value(**col**)	Non zero value (**value**)
Pointer to next data node in this column (**down**)		Pointer to next data node in this Row (**right**)



Pseudocode

```
MatrixNode(bool h,int r,int c,int v)
        head=h
        if head is false
                row=r
                col=c
                value=v
                down=NULL
                right=NULL
        else
                down=NULL
                right=NULL
                next=NULL

Matrix()
        headnode=NULL

readmatrix()
        declare variables i,rs,cs,nz
        display "Enter the rowsize, column size and no. of non-zero
        elements in the sparse matrix"
        read rs,cs,nz
        headnode=new MatrixNode(false,rs,cs,nz)
        MatrixNode **rownode=new MatrixNode * [rs]
        MatrixNode **colnode=new MatrixNode * [cs]
        For i=0 to rs-1
                rownode[i]=new MatrixNode(true,0,0,0)
        MatrixNode *lastrow=rownode[0]
        initialize currentrow=0
        MatrixNode *lastcol[cs]
        For i=0 to cs-1
                colnode[i]=new MatrixNode(true,0,0,0)
                lastcol[i]=colnode[i]
        headnode->right=colnode[0]
        headnode->down=rownode[0]
        For i=0 to nz-1
                declare r,c,v
                display "enter the non zero term"
                read r,c,v
                MatrixNode *datanode=new MatrixNode(false,r,c,v)
                If r > currentrow
                        currentrow=r
                        lastrow=rownode[currentrow]
                lastrow->right=datanode
                lastrow=datanode
                lastcol[c]->down=datanode
                lastcol[c]=datanode

        for i=0 to rs-2
                rownode[i]->next=rownode[i+1]
        for i=0 to cs-2
                colnode[i]->next=colnode[i+1]


display()
        display "Matrix is RowNo.\tColNo.\tValue\n"
        MatrixNode *row=headnode->down
        for i=0 to headnode->row
                MatrixNode *temp=row->right
                While temp!=NULL
                        Display temp->row,temp->col,temp->value
                        temp=temp->right
                row=row->next;

~Matrix()
        MatrixNode *temp,*p
        declare i,j=0
        MatrixNode *row=headnode->down
        For i=0 to headnode->row
```

```
                temp=row
                row=row->next
                while temp != NULL
                        p=temp->right
                        display j++
                delete temp
                temp=p
        temp=headnode->right
        for i=0 to headnode->col
                p=temp->next
                display j++
                delete temp
                temp=p
        delete headnode
        display j++
```

Problem Validation

```
        Input:
                Enter the rowsize, column size and no. of non-zero

elements in the sparse matrix: 5 4 4

enter the rowno, colon and non zero value:0 0 50

enter the rowno, colon and non zero value:1 2 30

enter the rowno, colon and non zero value:2 3 25

enter the rowno, colon and non zero value:3 1 15

        Output:
                Sparse Matrix is

                RowNo           ColNo           Value


                  0               0               50

                  1               2               30

                  2               3               25

                  3               1               15
```

Program 16

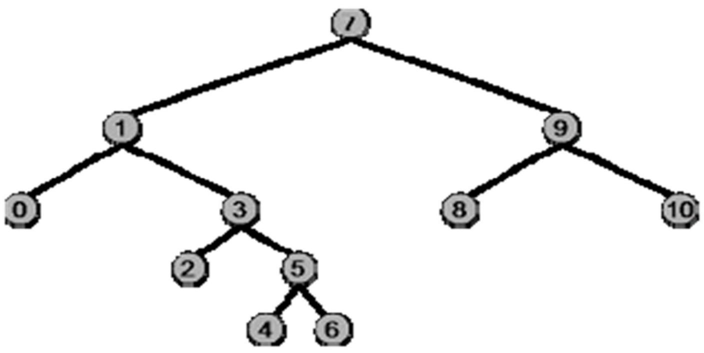
BINARY TREE TRAVERSALS

Problem Definition

Write a C++ program to implement Binary Tree Traversal

Problem Description

A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.



Preorder traversal: To traverse a binary tree in Preorder, following operations are carried- out (i) Visit the root, (ii) Traverse the left subtree, and (iii) Traverse the right subtree. Therefore, the Preorder traversal of the above tree will outputs: 7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10

Inorder traversal: To traverse a binary tree in Inorder, following operations are carried-out (i) Traverse the left most subtree starting at the left external node, (ii) Visit the root, and (iii) Traverse the right subtree starting at the left external node. Therefore, the Inorder traversal of the above tree will outputs: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Postorder traversal: To traverse a binary tree in Postorder, following operations are carried- out (i) Traverse all the left external nodes starting with the left most subtree which is then followed by bubble-up all the internal nodes, (ii) Traverse the right subtree starting at the left external node which is then followed by bubble-up all the internal nodes, and (iii) Visit the root. Therefore, the Postorder traversal of the above tree will outputs: 0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7

Pseudocode

```
constructor
binarytree()
      root=NULL

createtree(element,binarytreenode<T>  *p,binarytreenode<T> *q)
      root=new binarytreenode<T>
      root->data=element root->left=p
      root->right=q

inorder(binarytreenode<T> *temp)
      if temp != NULL
            inorder(temp->left)
            display temp->data
            inorder(temp->right)

preorder(binarytreenode<T> *temp)
      if temp != NULL
            display temp->data
            preorder(temp->left)
            preorder(temp->right)

postorder(binarytreenode<T> *temp)
      if temp != NULL
            postorder(temp->left)
            postorder(temp->right)
            display temp->data

levelorder(binarytreenode<T> *temp)
      LinkedQueue<binarytreenode<T> *> q
While temp != NULL
            Display temp->data
      If temp->left != NULL
            q.push(temp->left)
      if temp->right != NULL
            q.push(temp->right)
      if q is empty
            return
      else
            temp=q.pop()
```

Problem Validation

```
\noindent{\bf Input:} None

\noindent{\bf Output :}

Preorder Traversal :   7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10

Inorder Traversal: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Postorder Traversal: 0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7
```

Program 17

HASHING

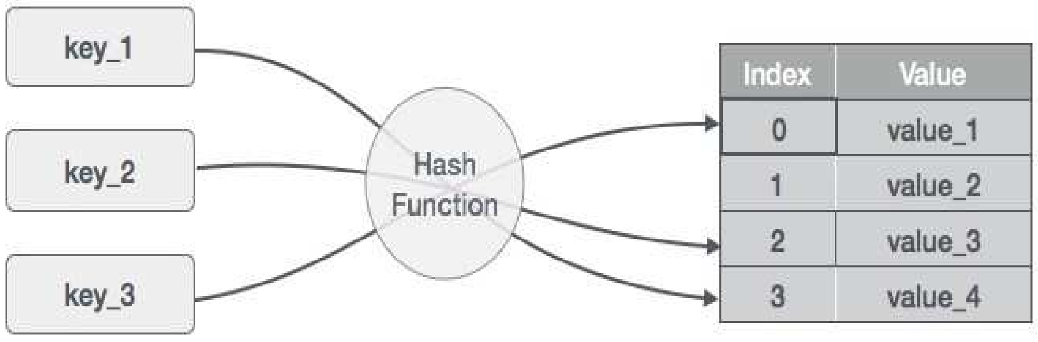
Problem Definition

Write a C++ program to implement Hashing

Problem Description

Hash Table is a data structure which store data in a hash table. The insertion and search operations are very fast irrespective of size of data. Hash Table is an array and a hash function is used to determine at which index position each data record is to be stored. Each data record has a unique key associated with it. Hashing is a technique to convert a range of key values into a range of indexes of an array.

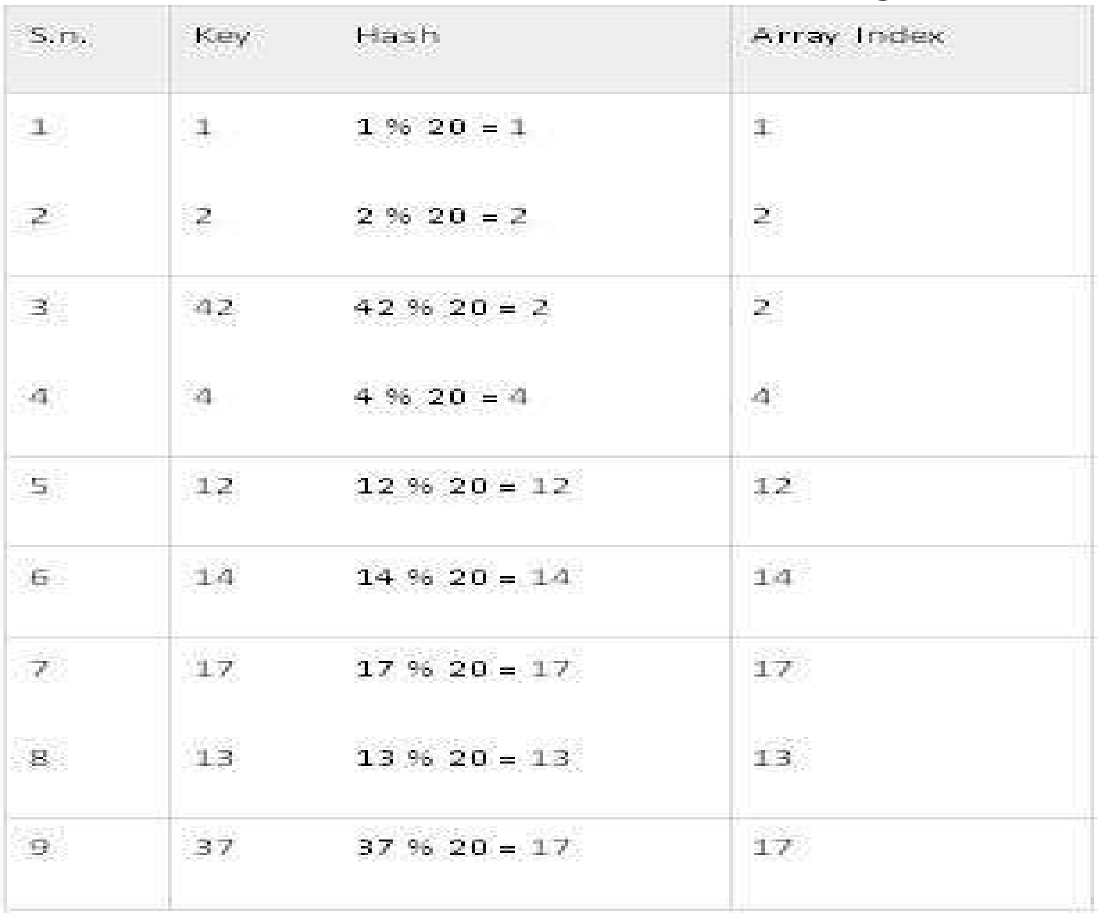
Hash(k)=index position where record with key “k” is stored in hash table



We’re going to use modulo operator to get a range of key values. Consider an example of hashtable of size 20, and following items are to be stored. Item are in (key,value) format.

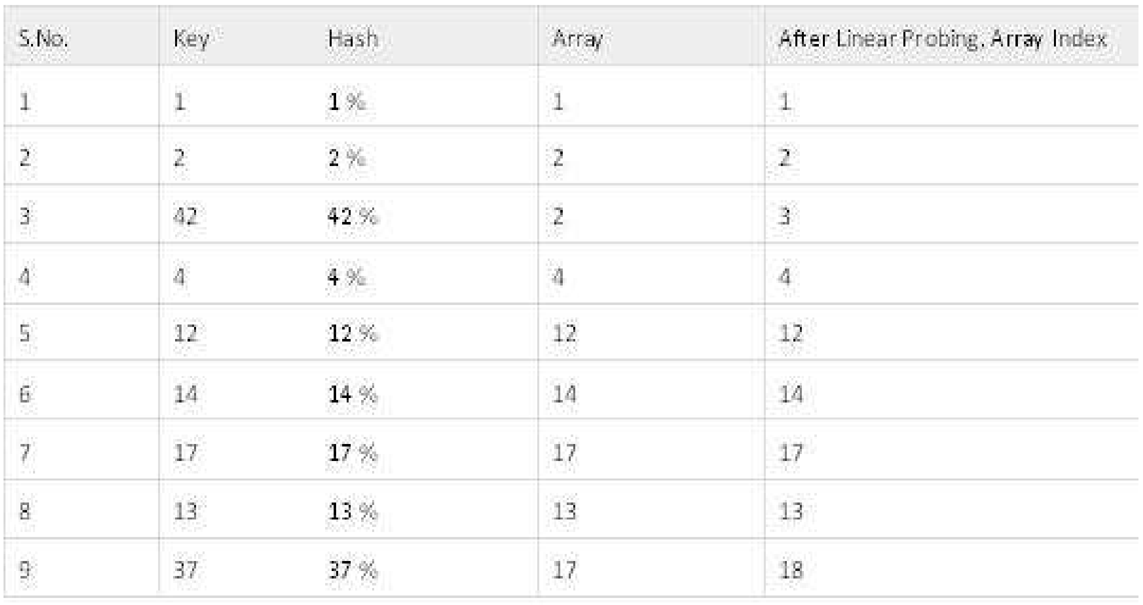
Example: hash table size=20. Hash function used is **hash(key)=key%size=index position**

Data records to be inserted in (key,value) format :(1,20), (2,70), (42,80), (4,25), (12,44), (14,32), (17,11), (13,78), (37,98)



S.n.	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

It may happen that the hashing technique used may map a record key to already used index of the array. This is called as collision. There are many collision resolution techniques. The simplest one is linear probing. In linear probing, when a record key is mapped to already filled array position, then we can search the next empty location in the array linearly until we found an empty cell. This technique is called **linear probing**.



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

Basic Operations

Following are basic primary operations of a hashtable :

Insert Algorithm

1. Declare variables hash and home. Variable home is used to store index position generated by hash function and variable hash is used to save a copy of home value.

2. Apply hash function to convert given "key" to index position i.e home=key

3. Assign hash = home, save the starting index position

4. Repeatedly search the hash table by doing following steps as long as empty cell is not found and key is not found in hash table

    a. Apply linear probing. That is go to next index position in hash table circularly.

    b. If we reach back the index position from which we started i.e "hash", then display "Table is full. Cant insert"

5. If empty cell is found during linear probing, then

    a. Save the key at that index position

Search Algorithm

1. Declare variables hash and home. Variable home is used to store index position generated by hash function and variable hash is used to save a copy of home value.
2. Apply hash function to convert given “key” to index position i.e home=key
3. Assign hash = home, save the starting index position.
4. Repeatedly search the hash table by doing following steps as long as empty cell is not found and key is not found in hash table.
   a. Apply linear probing. That is go to next index position in hash table circularly.
   b. If we reach back the index position from which we started i.e “hash”, then return -1 to indicate “Not Found”.
5. If empty cell is found during linear probing, then
   a. return -1 to indicate “Not Found”
6. Else,
   a. Return index value “hash” where key is found

Pseudocode

```
constructor - creates dynamic memory for hashtable of size 10

Hashing()
      table=new int[10]
      for i=0 to 9
            table[i]=0

destructor-deletes/frees memory used by hash table
~Hashing()
      delete [] table

insert(key)
      declare hash,home
      home=key%10
      hash=home
      while table[hash]!=0 and table[hash]!=key
            hash=(hash+1)%10
            if hash == home
                  display "Table is full. Cant insert"
                  return
      if table[hash] == 0
            table[hash]=key
```

```
search(key)
      declare hash,home
      home=key%10
      hash=home
      while table[hash!=0 and table[hash]!=key
            hash=(hash+1)%10
            if hash is equal to home
                  return -1
      if table[hash] is 0
            return -1
      else
            return hash
```

Problem Validation

```
\noindent{\bf Input: }
                Enter the key to be inserted: 12
                Enter the key to be inserted: 21
                Enter the key to be inserted: 34
                Enter the key to be inserted: 56
                Enter the key to be inserted: 78
                Enter the key to be inserted: 92

                (A) Enter the key to be searched:34
                (B) Enter the key to be searched:92
                (C) Enter the key to be searched:38

\noindent{\bf Output:}
                (A) Found at position 4
                (B) Found at position 3
                (C) Not Found
```

Program 18

HEAP SORT

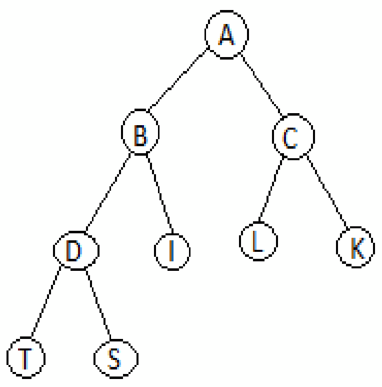
Problem Definition

Write a C++ program to implement Heap sort

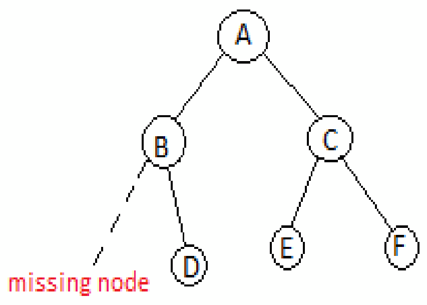
Problem Description

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

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

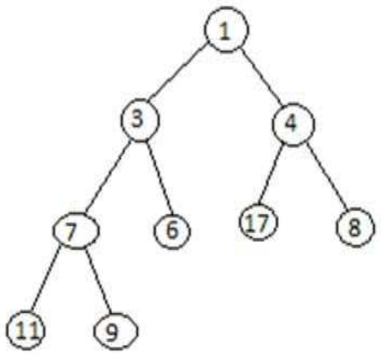


Complete Binary Tree



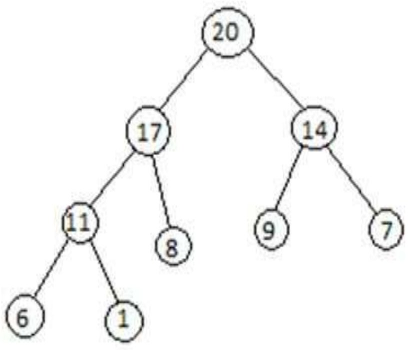
In-Complete Binary Tree

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



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.

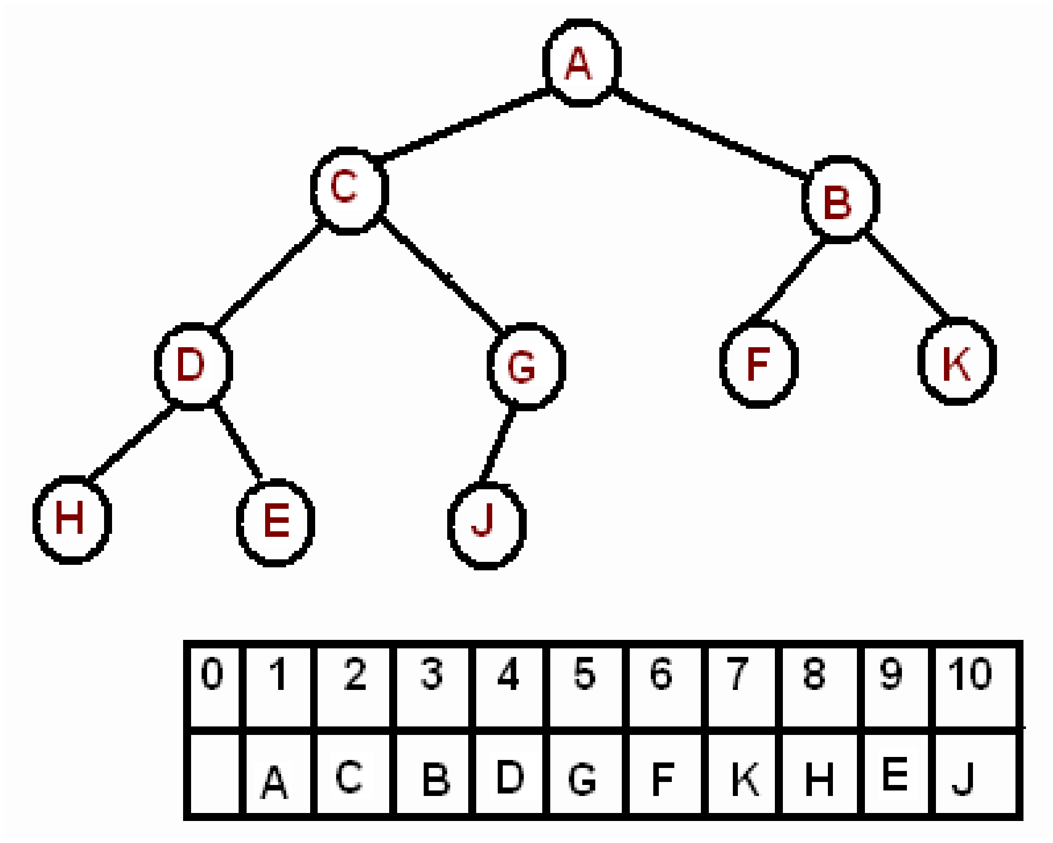


Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

Array Implementation A Heap can be represented by storing its level order traversal in an array. The root is the second item in the array. We skip the index zero cell of the array for the convenience of implementation. Consider i-th element of the array, the

```
its left child is located at 2*i index
its right child is located at 2*i+1.
index its parent is located at i/2 index
```



Heap Sort Algorithm

1. Store the given numbers in an array starting from position 0. This creates a complete binary tree.

2. The complete binary tree is then converted to a minheap as follows:

   a. For i=n/2 to 1 (n/2 is the index of first non-leaf node)

   a. Heapify node i in a heap of size n

3. To sort the numbers in ascending order:

   a. For i =n to 2

      i. Swap root node a[1] with last leaf node a[i]

      ii. Heapify root node 1 in heap of size i-1

(Root node is the smallest number. By putting it in last place and reducing the heap size by 1, we are removing it from heap)

Heapify Algorithm: It converts the non heap to min-heap form again. It starts from the given node position and creates a heap of given size.

1. Declare variables left,right,maxchild. They store the node positions of left,right and max child of a node.
2. Initialize left=2i, right=2i+1
3. While left child exists in heap
   a. If right child exists and eight child value is greater than left child value, then
      i. Assign maxchild=right
   b. Else
      i. Assign maxchild=left
   c. If parent is greated than maxchild,then (min heap condition is not satisfied)
      i. Break out of the loop
4. Swap parent and maxchild node values- (to make it a min heap)
5. Move one level down the heap by assigning parent i=maxchild, left=2i,right=2i+1

Pseudocode

```
heapsort(a[],n):array a and size n
        for i=n/2 to 1
                heapify(a,i,n)
        for i=n to 2
                swap a[1] and a[i]
                heapify(a,1,i-1)


heapify(a[],i,size)
        declare variables left,right,maxchild
        left=2*i
        right=2*i+1
        while left <= size
                if right <= size and a[right] > a[left]
                        maxchild=right
                else
                        maxchild=left
                if a[i] >= a[maxchild]
                        break
        swap a[i] and a[maxchild]
        i=maxchild
        left=2*i
        right=2*i+1
```

Problem Validation

\noindent{\bf Input: }

```
                Enter the size:10

                Enter the elements to be sorted using heapsort:
                12  89  9  78  45  34  22  11  90  55
```

\noindent{\bf Output:}

```
                Sorted elements are:
                9  11  12  22  34  45  55  78  89  90
```

Program 19

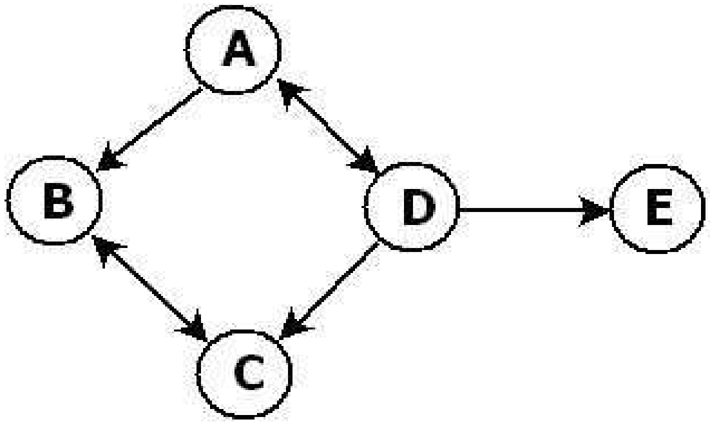
GRAPH : DFS & BFS

Problem Definition

Write a C++ program to implement Graph Traversal Techniques – DFS and BFS

Problem Description

A graph is a collection of nodes called vertices, and the connections between them, called edges. When the edges in a graph have a direction, the graph is called a directed graph or digraph, and the edges are called directed edges or arcs. More formally, a graph is an ordered pair, G = <V, E>, where V is the set of vertices, and E, the set of edges, is itself a set of ordered pairs of vertices. For example, the following expressions describe the graph shown below in set-theoretic language: V = {A, B, C, D, E}
E = {<A, B>, <A, D>, <B, C>, <C, B>, <D, A>, <D, C>, <D, E>}

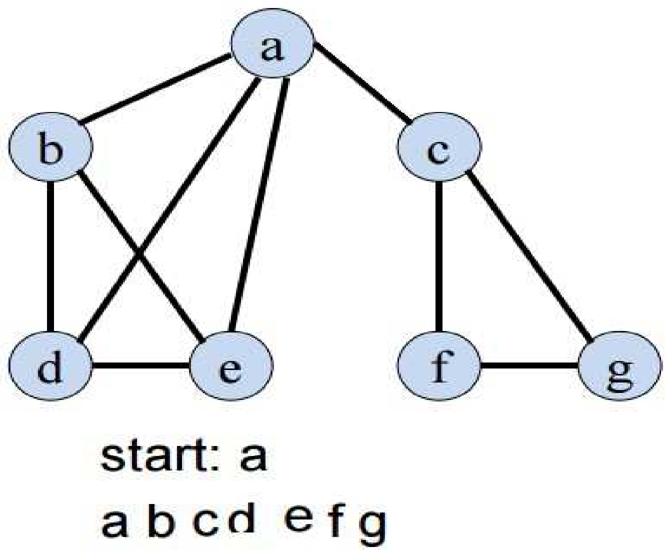


To traverse means to visit the vertices in some systematic order. Two popular graph traversal techniques: breadth first search (BFS) and depth first search (DFS) Graph is stored using adjacency matrix a[][] and n is no. of vertices. The array visited is used to keep track of which vertices are visited during graph traversals- BFS and DFS.

BFS Algorithm In a breadth first search, you start at the root node, and then scan each node in the first level starting from the leftmost node, moving towards the right. Then you continue scanning the second level (starting from the left) and the third level, and so on until you've scanned all the nodes. BFS uses Queue data structure

```
Bfs(vertex v)
       mark v as visited
       display v as output
       insert v into queue Q
       while Q is nonempty
              remove an element v from Q
              for each unmarked neighbor w of v(i,e w is adjacent to v)
                     mark w as visited
                     display w as output
                     insert w into queue Q
```

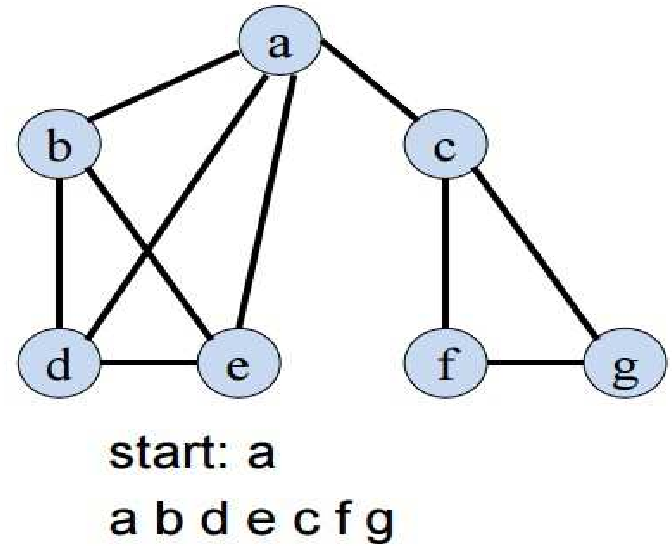
BFS Example



DFS Algorithm In a depth first search, you start at the root, and follow one of the branches of the tree as far as possible until either the node you are looking for is found or you hit a leaf node (a node with no children). If you hit a leaf node, then you continue the search at the nearest ancestor with unexplored children.DFS uses recursion, so indirectly it uses stack data structure.

```
dfs(vertex v)
       mark v as visited
       display v as output
       for each neighbor w of v
              if w is unvisited
                     dfs(w)
```

DFS Example



Pseudocode

```
Constructor-creates memory to store graph adjacency matrix and
visited array
graph(x):x is no. of vertices
       n=x
       e=0
       visited=new int[n+1]
       a=new int *[n+1]
       for i=1 to n
              a[i]=new int [n+1]
       for i=1 to n
              for j=1 to n
                     a[i][j]=0

destructor-deletes/frees memory used to store graph and visited matrix
~graph()
       delete []visited
       for i=1 to n
              delete a[i]
       delete[]a

addedge(v,u):adding edge(v,u) to graph
       a[v][u]=1
       a[u][v]=1
       e++
clearvisited():used to initialize visited array to zeroes
       for i=1 to n
              visited[i]=0

dfs(v)
       visited[v]=1 display v
       for i=1 to n
              if a[v][i]==1 and visited[i]!=1
                     dfs(i)

bfs(v)
      queue<int> q(50)
       visited[v]=1 display v
       q.push(v)
       while q not empty
              v=q.pop()
       for w=1 to n
              if a[v][w]==1 and visited[w]!=1
                     q.push(w)
                     visited[w]=1
                     display w
```

Problem Validation

```
 \noindent{\bf Input:}

              No. of vertices:7

              Edges added: (a,b),(a,c),(b,d),(a,d),(a,e),(c,f),(c,g)

\noindent{\bf Output:}

              BFS : a,b,c,d,e,f

              DFS: a,b,d,e,c,f,g
```

Annexure – I

List of programs according to O.U. curriculum

Code: CS231	**Data Structures using C++ LAB**	
Instruction	**3**	**Periods per week**
Duration of University Examination	**3**	**Hours**
University Examination	**50**	**Marks**
Sessional	**25**	**Marks**

1. Implementation of Stacks, Queues.
2. Infix to Postfix Conversion, evaluation of postfix expression.
3. Polynomial arithmetic using linked list.
4. Implementation of Binary Search and Hashing.
5. Implementation of Selection, Shell, Merge and Quick sorts.
6. Implementation of Tree Traversals on Binary Trees.
7. Implementation of Heap Sort.
8. Implementation of operations on AVL Trees.
9. Implementation of Traversal on Graphs.
10. Implementation of Splay Trees.

Suggested Reading:

1. Ellis Horowitz, Dinesh Mehta, S. Sahani. *Fundamentals of Data Stuctures in C++,* Universities Press. 2007.
2. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms,* Prentice Hall of India 1996.
3. Mark Allen Weiss, *Data Structures and Algorithm Analysis in C++,* Pearson Education 2006.