CL 210 - Data Structures and Algorithms

LAB MANUAL

DEPARTMENT OF ELECTRICAL ENGINEERING,

FAST-NU, LAHORE

Created by: Ms. Shazia Haque, Mr. Syed M Ammar Sohail, Ms. Sana Zahid

Date: Dec 26, 2016

Last Updated by: Ms. Shazia Haque & Mr. Umar Bashir

Date: April 8, 2019

Approved by the HoD: Dr. S.M Sajid

Date: April 8, 2019

Table of Contents

List of Equipment

Sr. No.	Description
1	Workstations (PCs)
2	Visual Studio 2010 C++ (software)

Lab 1

Performance analysis

Objective:
The objective of this experiment is to study the analytical and experimental ways of evaluating the performance of different programs in terms of time and space complexity.

Introduction:
Performance Analysis of Algorithms and Data Structures is carried out in terms of:

1) Space complexity.
2) Time complexity.

Space Complexity of a program is the amount of memory it needs to run to completion.

Time Complexity of a program is the amount of computer time it needs to run to completion.

There are two methods to determine the performance of a program, one is analytical, and the other is experimental.

Asymptotic Notation:
Asymptotic analysis of an algorithm or data structure refers to defining the mathematical boundaries of its performance. Using asymptotic analysis, we can conclude the best case, average case, and worst-case scenario of an algorithm. The **BIG-Oh** is the formal way to express the upper bound or worst-case of a program's time and space complexity.

Analytical Analysis:
Analytically finding time complexity of the following code is calculated in Table 1.1:

```
int sum, i, j;
sum = 0;
for (i=0;i<n;++i)
{       for(j=0;j<n;++j)
        {       sum++;
        }
}
```

Result:

Statement	Number of times executed
sum=0	1
i=0	1
i<n	n+1
++i	N
j=0	N
j<n	$n(n+1) = n^2 + n$
++j	n^2
sum++	n^2
Total	**$3n^2+4n+3$**
	$T(n) = 3n^2+4n+3$, $T(n) = O(n^2)$

Table 1.1

Exercise 1: For the following codes carry out the analytical analysis to evaluate time complexity T(n) and then express it in terms of Big Oh:

int sum,i;
sum = 0;
for (i=0;i<n;++i)
  sum++;
 	
T(n) =
Big O =
	int sum,i,j;
	sum = 0;
	for (i=1;i<n; i=i*2)
	  sum++;
	
	T(n) =
	Big O =
		int sum,i,j;
		sum = 0;
		for (i=1;i<n; ++i)
		{
		for (j=0;j<i;++j)
		  { sum++;
		  }
		}
		
		T(n) =
		Big O =

int sum,i,j;
sum = 0;
for (i=0;i<n;++i)
{
for (j=0;j<n;++j)
  { sum++;
  }
}
 	
T(n) =
Big O =
	int sum,i,j;
	sum = 0;
	for (i=1;i<n; i=i*2)
	{
	for (j=0;j<n;++j)
	  { sum++;
	  }
	}
	
	T(n) =
	Big O =
		int sum,i,j;
		sum = 0;
		for (i=1;i<n; i=i*2)
		{
		for (j=0;j<i; j++)
		  { sum++;
		  }
		}
		
		T(n) =
		Big O =

Experimental Analysis:

To find time complexity of a code experimentally we need to calculate the time difference between starting and ending time of execution of code. So "**time.h"** library is needed for this purpose.

```
#include<iostream>
#include<time.h>
using namespace std;
void main()
{
clock_t start, end;
double cpu_time_used;

start = clock();
                {     ... Code  ...     }
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
          cout<<cpu_time_used<<   " seconds";
}
```

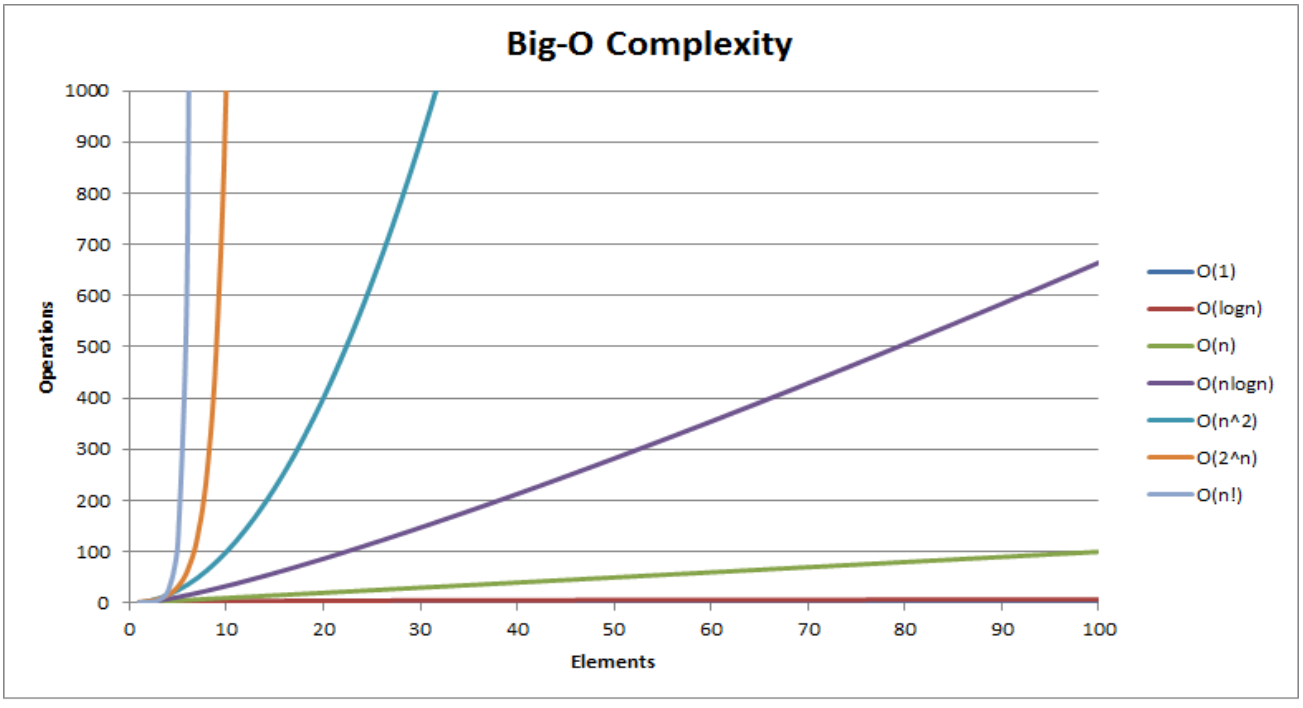


Figure 1.1: Asymptotic notation

Exercise2: Experimental Analysis

Your task is to write a program that takes the following array of 5 numbers:

Input_Numbers [5] = [10000, 50000, 100000, 500000, 1000000]

Each number present in the Input_Numbers array above represents a value for "n".

- For the codes given in table 1 find the execution time for each value of n given in the array above.
- Then plot separate graphs for time of execution vs. "n", for all the values of n given in the **Input_numbers** array.
- Use excel for plotting graph
- Verify that the graphs you plotted matches the asymptotic notation shown in figure 1.1.
- Sketch or paste the graphs in table 1.2.

Table 1.2

Exercise 3:

Write a program to find the factorial of a given number

i. Iteratively
ii. Recursively.

Describe the difference in terms of time and space complexity, on top of the code in comments.

POST LAB:

Following is a sorting algorithm. Your task is to run the code for various values of n (you may use the same values of n that you did your inLab for) and note the execution time taken by the sort method. By comparing this execution time graph with the ones in the inLab evaluate the Big Oh. Is this code Iterative or recursive?

```
#include<iostream>
#include<time.h>
using namespace std;
void print(int *a, int n)
{
	int i =0;
	while(i<n)
	{
		cout<<a[i]<<",";
		i++;
	}
}
void swap(int i,int j, int *a)
{
	int temp = a[i];
	a[i] = a[j];
	a[j] = temp;
}

void sort(int *arr, int left, int right)
{
	int min = (left+right)/2;

	int i = left;
	int j = right;
	int pivot = arr[min];
	while(left<j || i<right)
	{
		while(arr[i]<pivot)
			i++;
		while(arr[j]>pivot)
			j--;
	if(i<=j)
		{
			swap(i,j,arr);
			i++;
			j--;
		}
		else
		{
			if(left<j)
				sort(arr, left, j);
			if(i<right)
				sort(arr,i,right);
			return;
		}
	} // while ends
} // sort ends

int main()
{
	int n=10000;
	int *arrA=new int[n];

	//generate n random numbers
	for (int i=0; i<n; i++)
		arrA[i]=rand()%100;

	clock_t start, end;
	double cpu_time_used;
	start = clock();
	sort(arrA, 0, n-1);
	end = clock();
	cpu_time_used = ((double) (end - start)) /
CLOCKS_PER_SEC;
	cout<<cpu_time_used<<  " seconds";

	if(n<=100)
		print(arrA, n);
	return 0;
}
```

Lab 2

Arrays Implementation

Objective:

The objective of this experiment to get familiar with

- Arrays and its role in data structure
- Storage of data in Row major order and column major order.

Introduction:

An array is a series of elements of the same data type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

For example, five values of type **int** can be declared as an array without having to declare 5 different variables (each with its own identifier). These values can be accessed using the same identifier, with the proper index.

```
int array[5];
```

Multidimensional arrays can be described as "arrays of arrays". For example, a bi-dimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type.

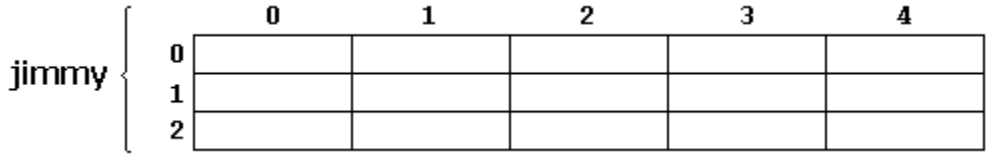


Figure 2.1

Jimmy represents a bi-dimensional array of 3 per 5 elements of type int as shown in above Figure 2.1. The C++ syntax for this is:

```
int jimmy [3][5];
```

In addition, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
jimmy[1][3]
```

2D row-major:

In row-major layout, the first row of the matrix is placed in first contiguous block of memory, then the second row in second contiguous block of memory, and so on as depicted in Figure 2.2:

Figure 2.2: Row Major

Offset required for accessing element of row index **row** and column index **col** can be represented by following equation:

$$Offset=row * NumCols + col$$

Where, NumCols is the number of columns per row in the matrix. It is easy to see this equation fits the linear layout in the diagram shown above.

2D column-major:

Column-major order same as row-major replacing every appearance of "row" by "column" and vice versa. The first column of the matrix is placed in first contiguous block of memory, then the second column in second contiguous block of memory, and so on as depicted in Figure 2.3:

Figure 2.3: Column Major

The offset of an element in column-major layout can be found using this equation:

$$Offset=col * NumRows + row$$

Where, NumRows is the number of Rows in the matrix.

Exercise 1:
Write a C++ code to copy data of a 2D array in a 1D array using Column Major Order.

Exercise 2:
Using the abstract data Type of a Matrix given below, write a program that

1. Input a 4*3 matrix from user in 2D array
2. Map this array in 1D array using Row major order
3. Input second matrix of 3*4 in 2D array
4. Map this array in 1D array using Row major order.
5. Now perform matrix multiplication **on these 1D arrays**
6. Save the result back in a 2D array.

Implement this question for any number of rows and columns using class "matrix".

```
#include<iostream>
using namespace std;
class matrix
{
    int **p;
    int r;
    int c;
    int *rowmajor;
    int *multiply1D;
public:

  matrix(int row, int col);
  // Constructor

  void disp2D();
  // displays the elements of **p


  void dispRowMajor();
  // converts 2D into 1D using row major
//and displays the elements Row Major Order Matrix

  void Multiply_rowMajor(matrix & x);
  // Multiplies Matrices in row major order and
save the result in a 1D dynamic array

 void rowMajor_2D();
  // Maps the elements stored in row major order to
  // the 2D array and print the results
  ~matrix();
  // Destructor
}
```

```
void main()
{
       matrix a(4,3);
       matrix b(3,4);
       a.disp2D();
       a.dispRowMajor();
       b.disp2D();
       b.dispRowMajor();
       a.Multiply_rowMajor(b);
       a.rowMajor_2D();

}

matrix::matrix(int row,int col)
{
       r=row;
       c=col;
       p = new int*[r];
       for(int  i=0;i<r;i++)
       {
              p[i]=new int[c];
              for(int j=0;j<c;j++)
                     p[i][j]=(i+j);
       }
// CODE FOR STORING DATA FROM
// **P TO *rowmajor ROW MAJOR
}
```

POST LAB:

Write down the node voltage equations of Circuit in Figure 2.4. Then write a code to input the data from user and display the data in the form of Matrix on Screen using Row Major Order, then Solve the matrix to get the value of V1, V2 and V3.

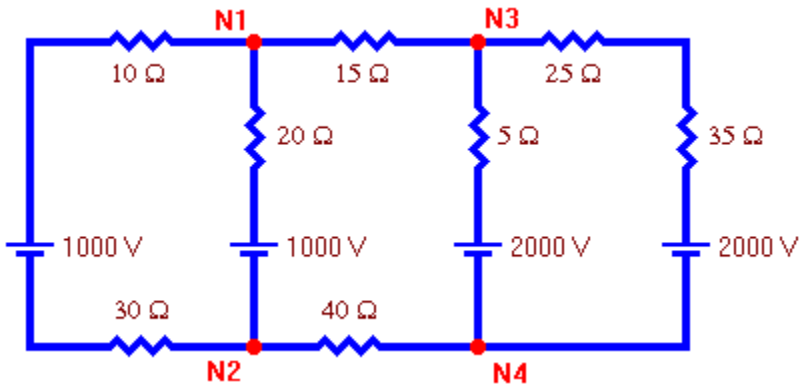


Figure 2.4: Electric Circuit

We will choose node 4 as the reference node and assign it a voltage of zero.

Lab 3

Singly Linked List

Objective:
The objective of this lab is to implement Linear Singly Linked List.

Introduction:
A singly linked list is a collection of components, called **nodes**. Every node (except the last node) contains the address of the next node. Thus, every node in a linked list has two components: one to store the relevant information (that is, data) and one to store the address, called the **link**, of the next node in the list. The address of the first node in the list is stored in a separate location, called the **head** or **first**. Last node to list points to NULL.

Properties:

- Singly linked list can store data in non-contiguous locations.
- Insertion and deletion of values is efficient with respect to Time and Space Complexity.
- Dynamic structure (Memory Allocated at run-time).
- Nodes can only be accessed sequentially. That means, we cannot jump to a particular node directly.
- There is no way to go back from one node to previous one. Only forward traversal is possible.

Basic Functionality:

1) Insertion of new node in Singly Linked List is depicted in Figure 3.1

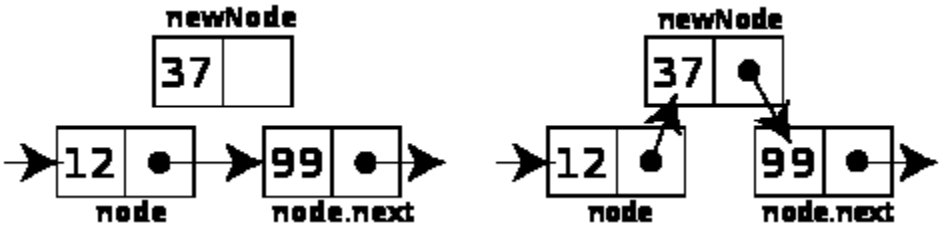


Figure 3.1: Insertion

1) Deletion of node from Singly Linked List is depicted in Figure 3.2

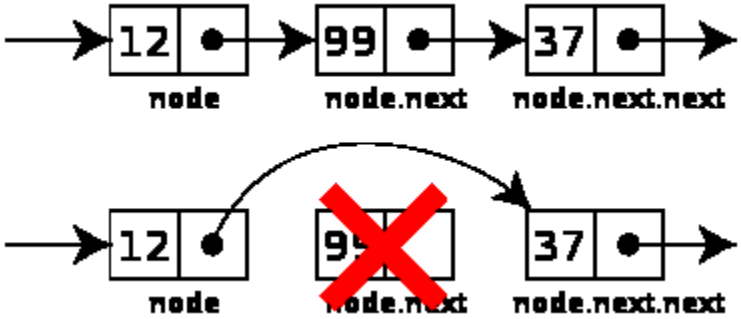


Figure 3.2: Deletion

2) Searching a node based on content. If the specified node is not present in the list an error message should be displayed.

Exercise 1: Implement a singly linked list.

Implement all the methods given in the following class definition for singly linked list. Your code should work for the main function that follows.

```
template <typename T>
class Node
{
   public:
           // constructor
           Node(T element);

             //sets the KeyType data in the Node
             void setData(T pVal);

             // returns the KeyType data in the Node
             T getData();

             // returns the link to the next node
             Node* GetNext();

             // sets the link to the next node
             void SetNext(Node *x);

   private:
           T data;
             Node *link ;
};

template <typename T>
class List
{
   public:
           // constructor of the Singly Linked List
           List();

            //Inserts the node pNew after the node pBefore
// if the list is empty, it makes pNew the first node of the list
             void Insert(Node<T>* pBefore, Node<T>* pNew);

             //Deletes the node pToBeDeleted
             void Delete(Node<T>* pToBeDeleted);

             //prints the contents of the list
             void printList();

//ADD ONE OR TWO MEMBER FUNCTION(s) DIFFERENT FOR EACH LAB SECTION
   private:
           Node<T> *first ;
```

```
int main()
{
      Node<int> *a, *b, *c, *d, *e;

      a = new Node<int>(1);
      b = new Node<int>(2);
      c = new Node<int>(3);
      d = new Node<int>(4);
      e = new Node<int>(5);


      List<int> *list;

      list = new List<int>();

      list->Insert(0 , a);
      list->Insert(a , b);
      list->Insert(b , c);
      list->Insert(a , d);
      list->Insert(b, e);

   list->printList();

      list->Delete(a);

     cout<<"\nAfter deleting first node"<<endl;
      list->printList();

// PLEASE ADD CALL TO MEMBER FUNCTION WHICH WOULD DIFFER SECTION-
WISE e.g. list->reverse etc.

      cout<<"\nAfter calling function"<<endl;
      list->printList();

      system("pause");
      return 0;

}
```

POST LAB:

Implement all the methods given in the following class definition for Singly Circular linked list. Implement a client program as well which demonstrates use of the functions of singly circular linked list

Circular Singly Linked List:

A linked list in which the last node points to the first node is called a circular linked list as shown in Figure 3.3.

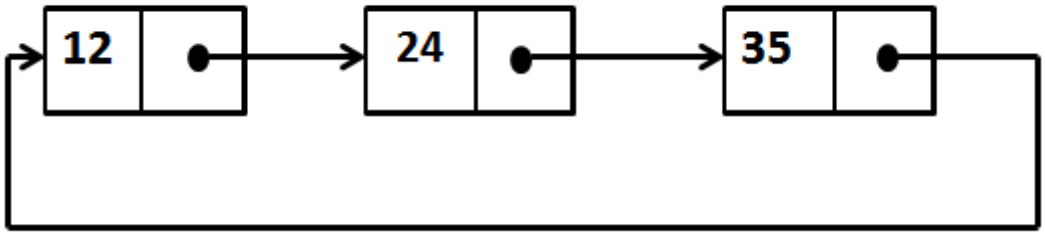


Figure 3.3: Singly Circular Linked List

```
template<typename T>
class Node
{   public:
        // constructor
        Node(T element);
        //sets the KeyType data in the Node
                void setData(T pVal);
                // returns the KeyType data in the Node
                 T getData();
                // returns the link to the next node
                 Node* GetNext();
                // sets the link to the next node
                void SetNext(Node *x);
private:
            T data;
            Node *link;
};
template <typename T>
class SCList
{public: // constructor of the Singly Circular Linked List
        SCList();
        /*Inserts the node pNew after the node pBefore
        if the list is empty, it makes pNew the first node of the list*/
                void Insert(Node<T>* pBefore, Node<T>* pNew);
                //Deletes the node pToBeDeleted
                void Delete(Node<T>* pToBeDeleted);
                //prints the contents of the list
                void printList();
private:
            Node<T> *last ;
};
```

Lab 4

Doubly Linked List

Objective:
The objective of this lab is to implement Doubly Linked List.

Introduction:
A doubly linked list is a linked list in which every node has a next pointer and a back pointer. In other words, every node contains the address of the next node (last node points to null as next node), and every node contains the address of the previous node (first node points to null as the previous node) as shown in Figure 4.1. Insertion and Deletion of Node are depicted in Figure 4.1 and 4.2 respecitvely.

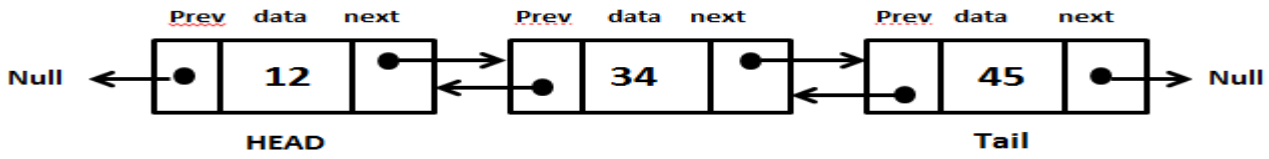


Figure 4.1: Doubly Linked List

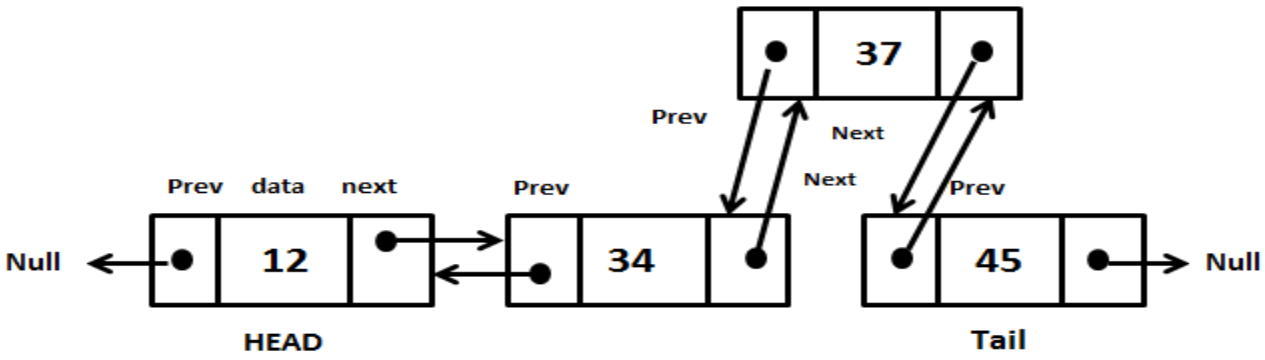


Figure 4.2: Insertion

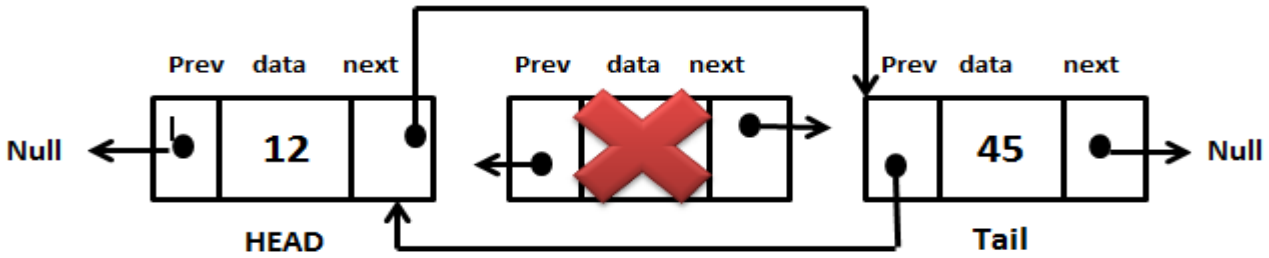


Figure 4.3: Deletion

Exercise 1: Implement a doubly linked list.

Implement all the methods given in the following class definition for doubly linked list. Run the main program which follows, demonstrating the use of the functions of the doubly linked list.

```
#ifndef DLIST_H
#define DLIST_H

template<class T>
class Node
{
      private:
            T data;
            Node *next ;
            Node *previous ;

      public:
            // constructor
            Node(T pdata);
            //sets the data in the Node
            void setData(T pVal);
            // returns the T data in the Node
            T getData();
            // returns the link to the next node
            Node* getNext();
            // sets the link to the next node
            void setNext(Node* x);
            // returns the link to the previous node
            Node* getPrevious();
            // sets the link to the previous node
            void setPrevious(Node* x);
};
template<class T>
class DList
{
private:
      Node<T> *first ;
public:
      DList();

      //Inserts the node pNew after the node pBefore
      // if the list is empty, it makes pNew the first node of the
list
      void Insert(Node<T>* pBefore, Node<T>* pNew);
      //Deletes the node pToBeDeleted
      void Delete(Node<T>* pToBeDeleted);
      //prints the contents of the list
      void printList();

//ADD ONE OR TWO MEMBER FUNCTION(s) DIFFERENT FOR EACH LAB
SECTION
};
#endif
```

```
int main()
{
	Node<int> *a, *b, *c, *d, *e,*f;

	a = new Node<int>(200);
	b = new Node<int>(30);
	c = new Node<int>(40);
	d = new Node<int>(45);
	e = new Node<int>(450);

	DList<int> *list;

	list = new DList<int>();

	list->Insert(0 , a);
	list->Insert(a , b);
	list->Insert(b , c);
	list->Insert(a , d);
	list->Insert(b, e);
	list->printList();

	list->delete(a);
	cout<<"\nAfter deleting first node"<<endl;
	list->printList();

	// PLEASE ADD CALL TO MEMBER FUNCTION(s) WHICH WOULD
DIFFER SECTION-WISE

	cout<<"\nAfter calling function"<<endl;
	list->printList();

	return 0;
}
```

POST LAB:

Implement all the functionalities of doubly circular linked list as given below and write a main function demonstrating their use.

```
template<class T>
class DCList
{
private:
	Node<T> *first ;
public:
	DCList();

	//Inserts the node pNew after the node pBefore
	// if the list is empty, it makes pNew the first node of
the list
	void Insert(Node<T>* pBefore, Node<T>* pNew);
	//Deletes the node pToBeDeleted
	void Delete(Node<T>* pToBeDeleted);
	//prints the contents of the list
	void printList();


//ADD ONE OR TWO MEMBER FUNCTION(s) DIFFERENT FOR EACH
LAB SECTION
};
```

Lab 5

Stacks

Objective:
The objective of the lab is to develop understanding of the Stack data structure and its basic functions.

Introduction:
A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations occur at one end only, called the top of the stack. It has a LIFO (Last In First Out) behavior.

Exercise 1: Implement Stack using Singly Linked list.
Implement all the methods given in the following class definition for Stack using array as the underlying data structure. Your implementation should work for the main function that follows.

```
#ifndef STACK_H
#define STACK_H
template<class KeyType>
class Stack
{ //
public:
	// constructor , creates an empty stack
	Stack();

	// returns true if Stack is full, otherwise return false
	bool IsFull();

	//If number of elements in the Stack is zero return true,
	otherwise return false
	bool IsEmpty();

	// If Stack is not full, insert item into the Stack. Must be an
	O(1) operation
	void Push(const KeyType item);

	// If Stack is empty return 0 or NULL;
	// else return appropriate item from the Stack. Must be an O(1)
	operation
	KeyType Pop();

private:
	SList <KeyType>* list;
	Node<KeyType>* top;

};
#endif
```

```
#include "Stack.h"
#include "Stack.cpp"
#include <iostream>
using namespace std;

int main()
{
	Stack<int> *st =new Stack<int>();

	if(st->IsEmpty())
		cout<<"Stack is currently empty"<<endl;

	st->Push(1);
	st->Push(2);
	st->Push(3);

	while (!st->IsEmpty())
	{
		int value=st->Pop();
		cout<<value<<endl;
	}
	return 0;

}
```

Exercise 2: Application of Stack

A postfix expression is an expression in which each operator follows its operands. The advantage of this form is that there is no need to group sub-expressions in parentheses or to consider operator precedence. Stack can be used to evaluate a postfix (or prefix) expression. Please implement the pseudo code given below:

1. Make a left to right scan of the postfix expression
2. If the element is an operand push it on Stack
3. If the element is operator, evaluate it using as operands the correct number from stack and pushing the result onto the stack

Post Lab:

Write a program to convert the infix expression to postfix expression by using Stack.

Lab 6

QUEUES

Objective:
The objective of the lab is to develop understanding of the Queue data structure and its basic functions.

Description:
A *queue* is also a data structure which works according to FIFO (First IN First Out) manner. Of the two ends of the queue, one is designated as the *front* − where elements are extracted (operation called *dequeue*), and another is the *rear*, where elements are inserted (operation called *enqueue*). A *queue* may be depicted as in Figure 6.1. The main operations are:

enqueue− place an element at the tail/rear of the queue;
dequeue− take out an element form the front/head of the queue;

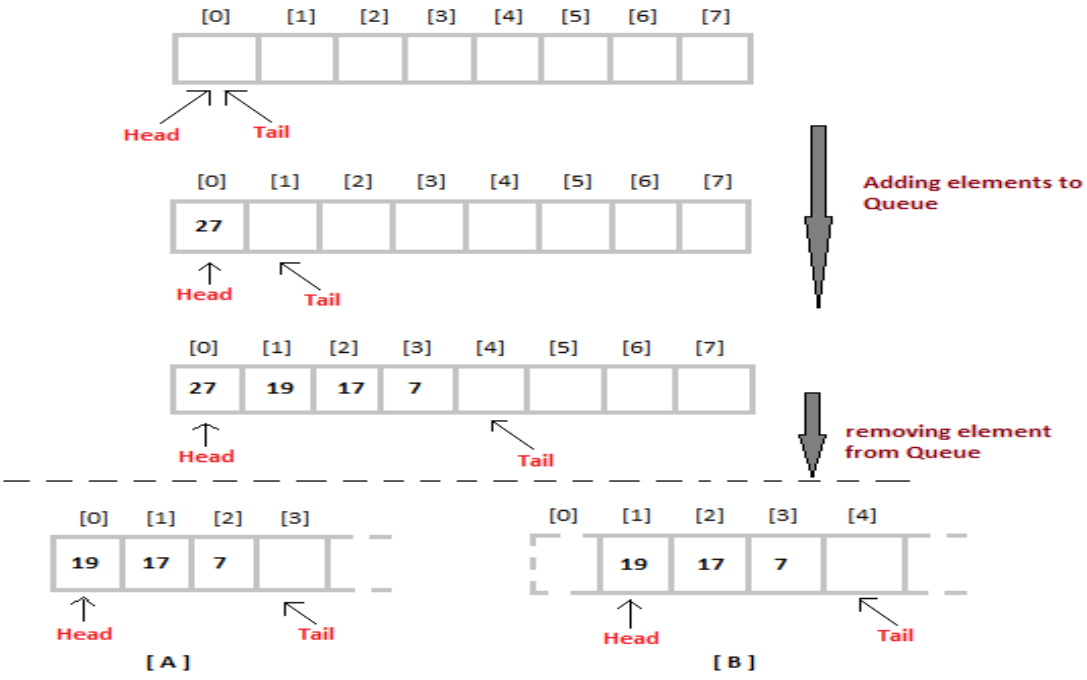


Figure 6.1

Exercise 1:

Please implement the following Queue definition using singly linked list (you may use the Singly Linked List that you already developed in Lab3), you may also add any functions needed in the Singly Linked List definition given in Lab3. Your implementation should work for the main function provided.

```
#ifndef QUEUE_H
#define QUEUE_H
template<class DT>
class Queue
{
private:
	//include private variables according to the underlying data structure
public:
	//contructor
	Queue();

//puts element at the rear end of the Queue if it is not full. Must be O(1)
	void Put(DT element);

//if queue not empty then delete the element at front of the Queue. Must be O(1)
	DT Get();

	//return true if the Queue is empty and false if it is not
	bool IsEmpty();

	//return true if the Queue is full and false if it is not
	bool IsFull();

};

#endif
```

```
#include "Queue.h"
#include "Queue.cpp"
#include <iostream>
using namespace std;

int main()
{
	Queue<int> *q =new Queue<int>();

	if(q->IsEmpty())
		cout<<"Queue is currently empty"<<endl;

	q->Put(1);
	q->Put(2);
	q->Put(3);

	while (!q->IsEmpty())
	{
		int value=q->Get();
		cout<<value<<endl;
	}
	return 0;

}
```

Exercise 2:
Give the C++ code to implement the above Queue using array. Please change the private data members as you are now using an array.

POST LAB:
Use the Queue you implemented in the in lab to check whether the word is a palindrome or not in a non-member function as given below.
#include “Queue.h”

bool isPalindrome(string word);

Lab 7

Binary Tree

Objective:
The objective of this experiment is to build a binary tree and then implement the basic binary tree traversals recursively.

Introduction:
A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.

Array representation of a binary tree:
If a complete binary tree with n nodes is represented using an array, then for any node with index i, $1 \le i \le n$, we have

- *parent(i)* is at $\lfloor i/2 \rfloor$ if $i \neq 1$.
  If $i = 1$, i is at the root and has no parent.
- *LeftChild(i)* is at $2i$ if $2i \le n$.
  If $2i > n$, then i has no left child.
- *RightChild(i)* is at $2i+1$ if $2i+1 \le n$.
  If $2i + 1 > n$, then i has no right child

For example the following array represents the tree shown in Figure 7.1:

Figure 7.1

Binary tree traversals:
Since a binary tree is a non-linear data structure, the problem is how to traverse it so we visit each node exactly once. Following are the three basic traversal methodologies in which we traverse the left sub tree before the right sub tree.

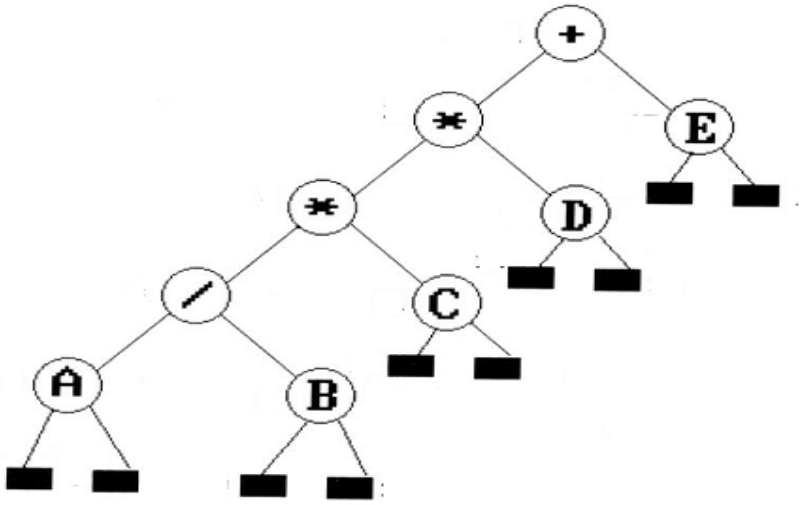


Figure 7.2

Preorder traversal: To traverse a binary tree in Preorder (VLR) manner 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 tree shown in Figure 7.2 will output: +**/ABCDE

Inorder traversal: To traverse a binary tree in Inorder (LVR), following operations are carried-out (i) Traverse the left subtree (ii) Visit the root, and (iii) Traverse the right subtree .
Therefore, the Inorder traversal of the tree shown in Figure 7.2 will output: A/B*C*D+E

Postorder traversal: To traverse a binary tree in Postorder (LRV), following operations are carried-out (i) Traverse all the leftsubtree (ii) Traverse the right subtree and (iii) Visit the root.
Therefore, the Postorder traversal of the tree shown in Figure 7.2 will output:AB/C*D*E+

Exercise 1: Implement the Binary tree node class and part1 and part 2 of the Binary tree class definition given in the header file (name it as BinaryTree.h) below:

```
#ifndef BINARYTREE_H
#define BINARYTREE_H
template<class DT>
class BNode
{
public:
	BNode();
	void setLeftChild(BNode<DT>* n);
	BNode<DT>* getLeftChild();
	void setRightChild(BNode<DT>* n);
	BNode<DT>* getRightChild();
	void setData(DT pdate);
	DT getData();
private:
	DT data;
	BNode* leftchild;
	BNode* rightchild;
};
template<class DT>
class BinaryTree
{
public:
	//part1: constructor
	BinaryTree ();
	//part 2:
	//Build the binary tree from the data given in the array.
	//If a node doesn't exist the array element is 0
	void BuildTree(DT *Arr, int Size);
	//part3: post order traversal (recursive)
	//you may call any other function with parameters which might be needed
	void PostOrder();
	//part4: pre order traversal (recursive)
	// you may call any other function with parameters which might be needed
	void PreOrder();
	//part5: in order traversal (recursive)
	// you may call any other function with parameters which might be needed
	void InOrder();

// part6: prints the height of the binary tree, you may pass any parameters needed
	int calculateDepth();

private:
// you may add any other private members which might be needed by recursive functions
	BNode<DT>* root;

};
#endif
```

You may test your code using the client program given below:

```
//Following is a sample client
#include<iostream>
#include "BinaryTree.h"
#include "BinaryTree.cpp"
using namespace std;
int main()
{
	//creating an object of binary tree
	BinaryTree<int> *BT=new BinaryTree<int>();

    //array to pass, 0 means no node exists
	int Arr[15]={0,1,2,3,4,5,6,7,8,9,10,0,12,13,14};

	BT->BuildTree(Arr,15); //building the tree from the array
	cout<<"*****************************************************"<<endl;


	return 0;
}
```

Exercise 2:

Implement part3, part4 and part5 of the header file given in exercise 1. Please test it using the following client

```
//Following is a sample client
#include<iostream>
#include "Binarytree.h"
using namespace std;
int main()
{
	//creating an object of binary tree
	BinaryTree<int> *BT=new BinaryTree<int>();

    //array to pass, 0 means no node exists
	int Arr[15]={0,1,2,3,4,5,6,7,8,9,10,0,12,13,14};

	BT->BuildTree(Arr,15); //building the tree from the array
	cout<<"*****************************************************"<<endl;

	cout<<"Preorder Traversal(Recursive) is: "<<endl;
	BT->PreOrder();
	cout<<"*****************************************************"<<endl;
	cout<<"Post order Traversal(Recursive) is: "<<endl;
	BT->PostOrder();
	cout<<"*****************************************************"<<endl;
	cout<<"Inorder Traversal(Recursive) is: "<<endl;
	BT->InOrder();
	cout<<"*****************************************************"<<endl;

	return 0;
}
```

POST LAB:

Implement part6 of the header file given in exercise 1 that should calculate and print the height of the Binary tree recursively. Please test it using the following client

```
//Following is a sample client
#include<iostream>
#include "BinaryTree.h"
#include "BinaryTree.cpp"
using namespace std;
int main()
{
	//creating an object of binary tree
	BinaryTree<int> *BT=new BinaryTree<int>();

    //array to pass, 0 means no node exists
	int Arr[15]={0,1,2,3,4,5,6,7,8,9,10,0,12,13,14};

	BT->BuildTree(Arr,15); //building the tree from the array
	cout<<"**************************************************"<<endl;

	cout<<"Height of the Binary Tree is: "<<endl;
	BT-> calculateDepth();
	cout<<"**************************************************"<<endl;

	return 0;
}
```

Lab 8

Binary Tree Traversals (iterative)

Objective:
The objective of this experiment is to implement the basic binary tree traversals iteratively.

Introduction:
A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left sub tree and the right sub tree

Binary tree traversals:
Since a binary tree is a non-linear data structure, the problem is how to traverse it so we visit each node exactly once. Following are the basic traversal methodologies in which we traverse the left sub tree before the right sub tree.

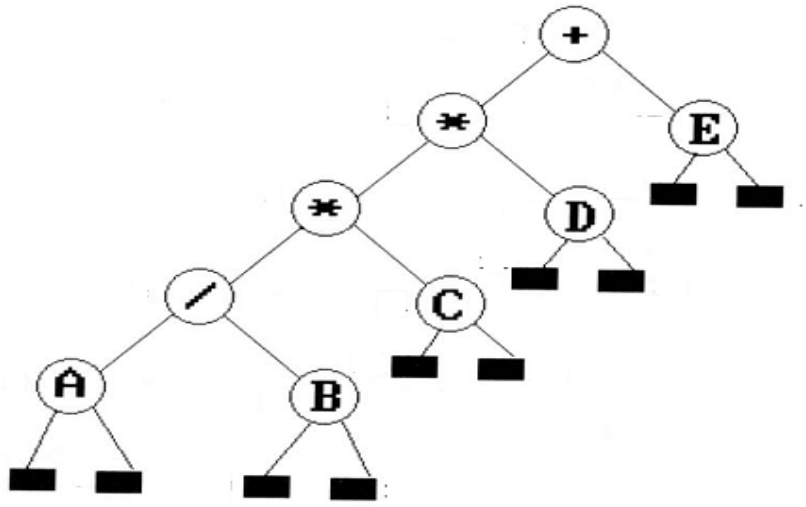


Figure 8.1

Preorder traversal: To traverse a binary tree in Preorder (VLR) manner 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 tree shown in Figure 8.1 will output: +**/ABCDE

Inorder traversal: To traverse a binary tree in Inorder (LVR), following operations are carried-out (i) Traverse the left subtree (ii) Visit the root, and (iii) Traverse the right subtree .
Therefore, the Inorder traversal of the tree shown in Figure 8.1 will output: A/B*C*D+E

Postorder traversal: To traverse a binary tree in Postorder (LRV), following operations are carried-out (i) Traverse all the leftsubtree (ii) Traverse the right subtree and (iii) Visit the root.
Therefore, the Postorder traversal of the tree shown in Figure 8.1 will output:AB/C*D*E+

Level order traversal: To traverse a binary tree in level order manner, we traverse each level of the tree from left to right starting from level 1 and moving down.
Therefore, the Level order traversal of the tree shown in Figure 8.1 will output:+*E*D/CAB

Exercise 1: Using the Binary tree node class and buildtree methods that you implemented in Lab7, implement Preorder, Inorder, Postorder and level order traversals (part1 to 4) of Binary tree **iteratively** as given in the class definition on next page (name it as BinaryTree.h). Please include and use the stack and queue that comes with C++ for implementing the traversals iteratively:

The following sample code demonstrates how we can use the built-in stack and queue that comes with C++ for storing BNode objects

```
#include<iostream>
#include<stack>
#include<queue>
#include "BinaryTree.h"
using namespace std;

void sample()
{
	//Using stack for storing objects of BNode
	stack<BNode<int>*> *s= new stack<BNode<int>*>();
	BNode<int>* temp=root;
	s->push(temp);
	temp=s->top();
	s->pop();
	cout<<temp->getData()<<" ";

	//Using queue for storing objects of BNode
	queue<BNode<int>*> *q= new queue<BNode<int>*>();
	BNode<int>* temp=root;
	q->push(temp);
	temp=q->front();
	q->pop();
	cout<<temp->getData()<<" ";


}
```

```
#ifndef BINARYTREE_H
#define BINARYTREE_H
template<class DT>
class BNode
{
public:
	BNode();
	void setLeftChild(BNode<DT>* n);
	BNode<DT>* getLeftChild();
	void setRightChild(BNode<DT>* n);
	BNode<DT>* getRightChild();
	void setData(DT pdate);
	DT getData();
private:
	DT data;
	BNode* leftchild;
	BNode* rightchild;
};
template<class DT>
class BinaryTree
{
public:
	//constructor already done in Lab7, please reuse that code
	BinaryTree ();

	//Build Tree method already done in Lab7, please reuse that code
	void BuildTree(T *Arr, int Size);

	//part 1: pre order traversal (iterative)
	// If a stack is needed please use the one that comes with C++
	void PreOrder();

	//part2: in order traversal (iterative)
	// If a stack is needed please use the one that comes with C++
	void InOrder();

	//part3: post order traversal (iterative)
	// If a stack is needed please use the one that comes with C++
	void PostOrder();

	// part4: level order traversal (iterative)
	// If a queue is needed please use the one that comes with C++
	void LevelOrder();

	// part5: calculate and return height of the tree iteratively (iterative)
	int calculateHeightItr();

private:
	BNode<DT>* root;
};
```

You may test your code using the following client:

```
//Following is a sample client
#include<iostream>
#include<stack>
#include<queue>
#include "Binarytree.h"
using namespace std;
int main()
{
	BinaryTree<int> *BT; //creating an object of binary tree
	BT=new BinaryTree<int>();

	//array to pass,0 means no node exists
	int Arr[15]={0,1,2,3,4,5,6,7,8,9,10,0,12,13,14};

	BT->BuildTree(Arr,15); //building the tree from the array
	cout<<"*****************************************************"<<endl;
	cout<<"Inorder Traversal(Iterative is: "<<endl;
	BT->InOrder();
	cout<<"*****************************************************"<<endl;
	cout<<"Preorder Traversal(Iterative) is: "<<endl;
	BT->PreOrder();
	cout<<"*****************************************************"<<endl;
	cout<<"Post order Traversal(Iterative) is: "<<endl;
	BT->PostOrder();
	cout<<"*****************************************************"<<endl;
	cout<<"Level order Traversal(Iterative) is: "<<endl;
	BT->LevelOrder();
	cout<<"*****************************************************"<<endl;

	return 0;
}
```

POST LAB:

Implement part5 of the header file given in exercise 1 that should calculate and return the height of the Binary tree **iteratively**. You may test it using the following client:

```
//Following is a sample client
#include<iostream>
#include "Binarytree.h"
using namespace std;
int main()
{
	BinaryTree<int> *BT; //creating an object of binary tree
	BT=new BinaryTree<int>();

	//array to pass,0 means no node exists
	int numbers[15]={0,1,2,3,4,5,6,7,8,9,10,0,12,13,14};

	BT->BuildTree(numbers,15); //building the tree from the array
	cout<<"*****************************************************"<<endl;
	cout<<"Height of the Binary Tree is: "<<endl;
	BT-> calculateHeightItr();
	cout<<"*****************************************************"<<endl;

	return 0;
}
```

Lab 9

Binary Search Tree

Objective:
The objective of this experiment is to implement a Binary Search Tree to be used for search applications.

Introduction:

A *binary search tree(shown in Figure 9.1)* satisfies the following conditions:

- It is a binary tree
- Every element in it has a unique key
- The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree
- The left and right subtrees are also binary search trees

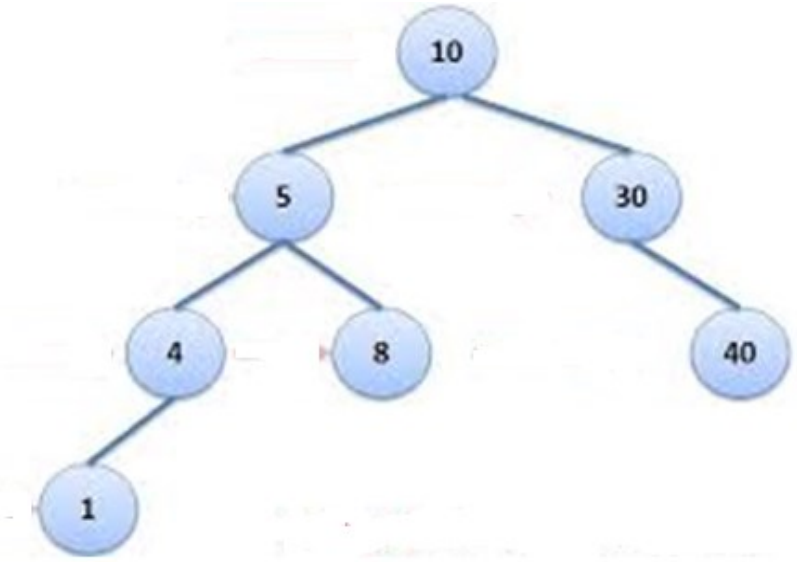


Figure 9.1

Exercise 1:

Using the Binary tree node class that you implemented in Lab7, implement part1 to part4 given in the Binary Search Tree class definition given as follows (you may name it as BinarySearchTree.h).

```
#ifndef BINARY_SEARCH_TREE_H
#define BINARY_SEARCH_TREE_H

#include "BNode.h"
using namespace std;

template<class DType>
class BinarySearchTree
{
public:
	//part1: constructor
	BinarySearchTree();

	//part2: Create and insert a BNode carrying data
	//in the binary search tree. It return true if
	//insertion takes place successfully and false otherwise
	bool insert(const DType data);

	//part3: Search for data in the binary search tree
	// and return the pointer of the node carrying data
	//return null/0 if data doesn't exist
	BNode<DType> * search(const DType data);

	//part4: prints all the data present in the tree
	//sorted in ascending order
	void printSorted();


	//part5: delete the BNode carrying data from the
	//binary search tree. It return true if
	//deletion takes place successfully and false otherwise
	bool delete(const DType data);


	//part6: destructor, delete all nodes
	~BinarySearchTree();


private:
	BNode<DType> * root;

};
#endif
```

You may test your code using the following client

```
//Following is a sample client
#include<iostream>
#include "BinarySearchTree.h"
using namespace std;
int main()
{
     //creating an object of binary search tree
       BinarySearchTree<int> *BST=new BinaryTree<int>();

       //following insertions should happen successfully as we are inserting unique
values
       BST->insert(12);
       BST->insert(4);
       BST->insert(9);
       BST->insert(2);
       BST->insert(14);
       BST->insert(16);
       BST->insert(13);

       //this insertion should fail as 12 already exists in the Binary Search tree
       BST->insert(12);

       //prints data carried by the BST in sorted manner
       BST->printSorted();

       //the first search would be successful and second would fail
       BNode<int>*n =BST->search(12);
       If(n)
       {
              cout<<"Value exists"<<endl;
       }
       else
       {
              cout<<"Value does not exist"<<endl;
       }

       BNode<int>*n =BST->search(23);
       If(n)
       {
              cout<<"Value exists"<<endl;
       }
       else
       {
              cout<<"Value does not exist"<<endl;
       }

return 0;
}
```

POST LAB:

Implement part5 and part6 given in the Binary Search Tree class definition given in Exercise1. You may test it using the following client:

```
//Following is a sample client
#include<iostream>
#include "BinarySearchTree.h"
using namespace std;
int main()
{
	//creating an object of binary search tree
	BinarySearchTree<int> *BST=new BinaryTree<int>();

	//following insertions should happen successfully as we are inserting
	//unique values
	BST->insert(12);
	BST->insert(4);
	BST->insert(9);
	BST->insert(2);
	BST->insert(14);
	BST->insert(16);
	BST->insert(13);
	BST->insert(1);

	//prints data carried by the BST in sorted manner
	BST->printSorted();

	//deleting  leaf node
	If(BST->delete(16))
	{
		cout<<”node carrying 16 deleted successfully”<<endl;
	}
	//deleting  degree 1 node
	If(BST->delete(2))
	{
		cout<<”node carrying 2 deleted successfully”<<endl;
	}
	//deleting  degree 2 node
	If(BST->delete(12))
	{
		cout<<”node carrying 12 deleted successfully”<<endl;
	}
	//prints data carried by the BST in sorted manner
	BST->printSorted();

	//destructor called
	delete BST;

	return 0;

}
```

Lab 10

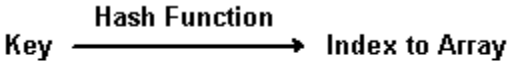
Hashing

Objective:
The objective of this lab is to implement data insertion and search from Hash table using linear probing and Chaining as overflow handling techniques, then compare the two techniques on the basis of time taken to execute search.

Introduction:
Hash tables are good for doing a quick search on things. If we have an array full of data (say 100 items), if we knew the position where a specific item is stored in the array then we could quickly access it. For instance, we just happen to know that the item that we want is at position 3. Then we can apply

```
myitem=myarray[3];
```

With this, we don't have to search through each element in the array, we just access position 3.The question is, how do we know that position 3 stores the data that we are interested in? This is where hashing comes in handy. Given some key, we can apply a hash function to it to find an index or position that we want to access.



Hash function:
There are many different hash functions. Some hash functions will take an integer key and turn it into an index. A common one is the division method.
Let's learn through an example:

Division method (one hash method for integers)
Let's say you had the following numbers or keys that you wanted to map into an array of 10 elements:
123456
123467
123450

To apply the division method, you could divide the number by 10 (or the maximum number of elements in the array) and use the remainder (the modulo) as an index. The following would result:
123456 % 10 = 6 (the remainder is 6 when dividing by 10)
123467 % 10 = 7 (the remainder is 7)
123450 % 10 = 0 (the remainder is 0)

These numbers would be inserted into the array at positions 6, 7, and 0 respectively. HashTable might look Figure 10.1:

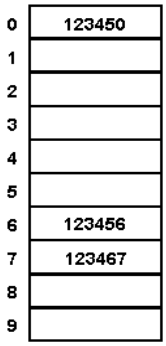


Figure 10.1

The important thing with the division method is that the keys are integers**.** However, what happens when the keys aren't integers? In that case, you have to apply another hash function to turn them into integers. Effectively, you get two hash functions in one as depicted in Figure 10.2:

1. function to get an integer
2. function to apply a hash method from above to get an index to an array

Figure 10.2

Overflow Handling techniques:

A problem occurs when two keys yield the same index. For Instance, say we wanted to include: 123477% 10 --> 7

We have a **collision** because 123467 is already stored at array index 7.

We need a method to resolve this. The resolution comes in how you create your hash table. There two major approaches given in the book:

1. **Linear Probing:**

   When a new identifier is hashed into a full bucket, we need to find another bucket for this identifier. The simplest solution is to find the closest unfilled bucket. This is called *linear probing* or *linear open addressing*

2. **Chaining:**
   In chaining you have an array of linked lists as shown in Figure 10.3. All the data in the "same link", have Colliding Hash values.

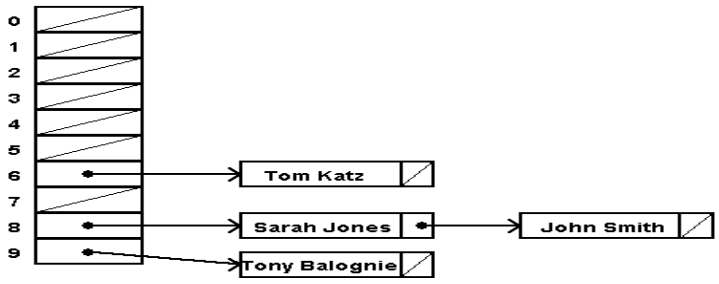


Figure 10.3

Exercise 1: Generate n random numbers (using rand%1000) and store them is a hash table.The value of n should be asked from the user at runtime and decide the size of array to use accordingly (hint: big prime number).

Please use the modulus function (%) as the hash function and **Open addressing (or linear probing)** as the Overflow handling technique. You have to implement the following ADT of Hashtable:

```
#ifndef HSH_H
#define HSH_H

template<class DT>
class Hashtable
{
public:
	Hashtable(int sizehash);
	bool store(DT key);
	bool search(DT key);

private:
	int size;
	DT* arr;
};
#endif
```

Once all the n numbers have been stored in this hashtable, search for a particular number entered by the user and note the time it takes to carry out the search. If the number is found (or not) let the user know accordingly.

SAMPLE SEARCH CODE

```
template<class DT>
bool Hashtable<DT>::search_in_hashtable(DT identifier)
{
	bool success=true;
	int location = identifier%size;
	int j;
	for( j=location; arr[j]!=identifier; )
	{
		j=(j+1)%size;
		if(j==location || arr[j]==-1)
		{ success=false;//doesn't exist
		  break;
		}
	}
	if(success)
	{
		cout<<"found at "<<j<<endl;
	}
	return success;
}
```

You may test your code using the following client

```
#include<iostream>
#include<Windows.h>
#include "Hashtable.h"
#include "Hashtable.cpp"
using namespace std;
int main()
{
	int num_of_identifiers;
	cout<<"Enter maximum number of keys that need to be stored in the hashtable:
";
	cin>>num_of_identifiers;

	int size_hashtable;
	cout<<"Enter size of hashtable needed to store these many identifiers (hint:
use prime number): ";
	cin>>size_hashtable;

	//create a hashtable of this size
	Hashtable<int>* ht=new Hashtable<int>(size_hashtable);

	for (int i=0; i<num_of_identifiers; i++)
	{
		int key=rand()%1000;
		bool was_stored = ht->store(key);
		if(!was_stored)
			cout<<key<<" could not be stored as it already exists or table
is full"<<endl;
	}

	int find_key;
	cout<<"Enter the key to search for "<<endl;
	cin>>find_key;

	DWORD start, end;//measure time
	start= GetTickCount();
	bool found= ht->search(find_key);
	end= GetTickCount();
	double cpu_time_used = end - start;
	if(found)
		cout<<"it was found in "<<cpu_time_used<<" milliseconds"<<endl;
	else
		cout<<"not found in "<<cpu_time_used<<" milliseconds"<<endl;
	return 0;
}
```

Exercise 2: Generate n random numbers (using rand%1000) and store them is a hashtable (use an array of singly linked lists). The value of n should be asked from the user at runtime and decide the size of array to use accordingly (hint: big prime number).

Please use the modulus function (%) as the hashfunction and **Chaining** as the Overflow handling technique. You may use the singly linked list code implemented in lab 3 to manage the chain held at each index of the hashtable. Following sample code is given for your guidance:

```
template<class DT>
bool Hashtable<DT>::search(DT identifier)
{
	bool success=false;
	int location = identifier%size;
	if(arr[location])
	{
		Node<DT> *n= arr[location]->GetFirst();
		while(n)
		{
			if(n->getData()==identifier)
			{
				success=true;
				break;
			}
			n=n->GetNext();
		}
	}
	else
		success=false;

return success;
}
```

You have to implement the following ADT of Hashtable:

```
#ifndef HSH_H
#define HSH_H
#include "SList.h"

template<class DT>
class Hashtable
{
public:
	Hashtable(int size);
	bool store(DT key);
	bool search(DT key);
	void printdata();

private:
	int size;
	List<DT>** arr;
};
#endif
```

You may test your code using the following client

```
#include<iostream>
#include<Windows.h>
#include "Hashtable.h"
#include "Hashtable.cpp"
using namespace std;
int main()
{
	int num_of_identifiers;
	cout<<"Enter maximum number of keys that need to be stored in the hashtable:
";
	cin>>num_of_identifiers;

	int size_hashtable;
	cout<<"Enter size of hashtable needed to store these many identifiers (hint:
use prime number): ";
	cin>>size_hashtable;

	//create a hashtable of this size
	Hashtable<int>* ht=new Hashtable<int>(size_hashtable);

	for (int i=0; i<num_of_identifiers; i++)
	{
		int key=rand()%1000;
		bool was_stored = ht->store(key);
		if(!was_stored)
			cout<<key<<" could not be stored as it already exists or table
is full"<<endl;
	}
	ht->printdata();
	int find_key;
	cout<<"Enter the key to search for "<<endl;
	cin>>find_key;

	DWORD start, end;//measure time
	start= GetTickCount();
	bool found= ht->search(find_key);
	end= GetTickCount();
	double cpu_time_used = end - start;

	if(found)
		cout<<"it was found in "<<cpu_time_used<<" milliseconds"<<endl;
	else
		cout<<"not found in "<<cpu_time_used<<" milliseconds"<<endl;

	return 0;
}
```

POST LAB:

1. For different number of identifiers and hashtable sizes, execute the Linear probing and Chaining overflow handling technique(s) codes implemented in the inLab and fill the following table with time taken to execute search.

Number of Identifiers	Hashtable size	Time taken by Linear Probing in milliseconds	Time taken by Chaining in milliseconds
100	113		
1000	1009		
5000	4999		
8000	7013		

2. Add a member function called delete in the Hashtable ADT for Exercise2 (using chaining as overflow handling technique) of inLab and implement it.
3. Add a member function called delete in the Hashtable ADT for Exercise1(using open addressing as overflow handling technique) and implement it.

Lab 11

Heaps

Objective:

The objective of this experiment is to implement a Max Heap as well as a MinHeap using array representation of binary trees.

Introduction:

A binary heap is a heap data structure created using a binary tree. It can be seen as a binary tree with two additional constraints.

i. Shape property:
A binary heap is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

ii. Heap property:
All nodes are either *greater than or equal to* **(Max-Heaps)** *or less than or equal to* **(Min-Heaps)** each of its children, according to a comparison predicate defined for the heap.

Heap operations:

i. Insert:

1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

As an example of binary heap insertion, say we have a max-heap as shown in Figure 11.1.

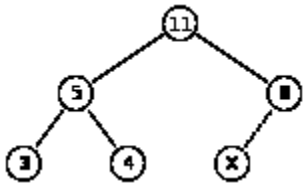


Figure 11.1

We want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However, the heap property is violated since 15 > 8, so we need to swap the 15 and the 8. So, we have the heap looking as show in Figure 11.2 after the first swap:

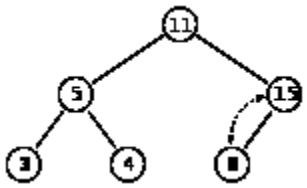


Figure 11.2

However the heap property is still violated since 15 > 11, so we need to swap again So, we have the heap looking as show in Figure 11.3 after the second swap swap:

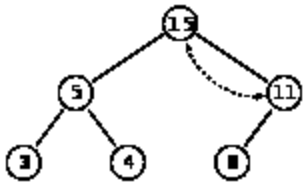


Figure 11.3

ii. Delete:

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

So, if we have the same max-heap as before shown in Figure 11.1. We remove the 11 and replace it with the 4 as shown in Figure 11.4.

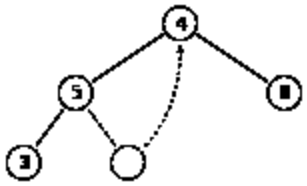


Figure 11.4

Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements, 4 and 8, is enough to restore the heap property and we need not swap elements further as shown in Figure 11.5.

Figure 11.5

The downward-moving node is swapped with the *larger* of its children in a max-heap

Exercise 1: Implement the MaxHeap class definition given as follows (you may name it as MaxHeap.h). For ease, this Heap is not template based and is carrying integer data.

```
#ifndef MAX_HEAP_H
#define MAX_HEAP_H

using namespace std;
template<class DT>
class MaxHeap
{
public:
	//part1: constructor initializes array of size maxsize
	MaxHeap(int maxsize);

	//part2: Inserts data into its appropriate position
	//within the Heap
	bool insert(const DT data);

	//part3: removes the element present in the the root
	//of the Heap and readjusts it to form MaxHeap again
	DT delete();

	//part4: prints all the data present in the Heap
	//use the appropriate traversal
	void printContents();

	//part5: destructor, deletes the MaxHeap
	~MaxHeap();


private:
	DT *arr;

};
#endif
```

You may test your code using the following client

```
//Following is a sample client
#include<iostream>
#include "MaxHeap.h"
using namespace std;
int main()
{
	MaxHeap<int> *mxHeap; //creating an object of maxheap
	mxHeap=new MaxHeap<int>(40);

	//insert following data in the MaxHeap
	mxHeap->insert(12);
	mxHeap ->insert(43);
	mxHeap ->insert(9);
	mxHeap ->insert(2);
	mxHeap ->insert(14);
	mxHeap ->insert(16);
	mxHeap ->insert(13);
	mxHeap ->insert(12);

	mxHeap->printContents();

	//Carry out 2 deletions from the MaxHeap
	int output;
	output=mxHeap->delete();

	cout<<"Output of first deletion is  "<<output<<endl;
	mxHeap->printContents();

	output=mxHeap->delete();

	cout<<"Output of second deletion is  "<<output<<endl;
	mxHeap->printContents();

	return 0;
}
```

Exercise 2: Alter the definition as well as implementation of the MaxHeap done in Exercise1 to create a MinHeap carrying integer data. You may test your code using the following client

```
//Following is a sample client
#include<iostream>
#include "MinHeap.h"
using namespace std;
int main()
{
	MinHeap<int> *mnHeap; //creating an object of MinHeap
	mnHeap=new MinHeap<int>(40);

	//insert following data in the MinHeap
	mnHeap->insert(12);
	mnHeap ->insert(43);
	mnHeap ->insert(9);
	mnHeap ->insert(2);
	mnHeap ->insert(14);
	mnHeap ->insert(16);
	mnHeap ->insert(13);
	mnHeap ->insert(12);

	mnHeap->printContents();

	//Carry out 2 deletions from the MinHeap
	int output;
	output=mnHeap->delete();

	cout<<"Output of first deletion is  "<<output<<endl;
	mnHeap->printContents();

	output=mnHeap->delete();

	cout<<"Output of second deletion is  "<<output<<endl;
	mnHeap->printContents();

	return 0;
}
```

POST LAB:

Implement a menu driven Airport landing system, which asks the user for the amount of fuel (integer value) present in the aircraft. The lower this value, higher the priority of the aircraft to be given access to runway. Use the appropriate Heap developed the in Lab to insert incoming aircrafts in the heap and then grant access to runway according to priority. A sample is given below:

```
Welcome to Lahore airport civil aviation landing system

Is there an aircraft in airspace requesting to land?
Yes
Please enter the amount of fuel present in the aircraft:
20
Is there an aircraft in airspace requesting to land?
Yes
Please enter the amount of fuel present in the aircraft:
40
Is there an aircraft in airspace requesting to land?
Yes
Please enter the amount of fuel present in the aircraft:
10
Is there an aircraft in airspace requesting to land?
No

Grant access of runway to the aircrafts in the following order:
Aircraft with 10 units of fuel
Aircraft with 20 units of fuel
Aircraft with 40 units of fuel
```

Lab 12

Graphs

Objective:

The objective of this experiment is to implement graph using adjacency matrix and adjacency list.

Introduction:

Graph is a data structure that consists of following two components:

- A finite set of vertices also called as nodes.
- A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph (di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight.

Graphs are used to represent many real life applications as they can be used to represent networks. The networks may include paths in a city or telephone network or circuit network.

An example of undirected graph with 5 vertices is shown in Figure 12.1.

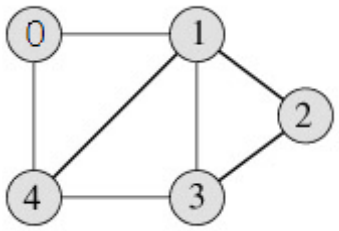


Figure 12.1

Graph representations:

Following two are the most commonly used representations of graph.

- Adjacency Matrix
- Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The graph shown in Figure 12.1 is represented using adjacency matrix in Figure 12.2:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Figure 12.2

Adjacency Matrix Representation of the above graph

Exercise 1: Implement the following Graph ADT using Adjacency matrix representation of graph

```
#ifndef GRAPH_H
#define GRAPH_H

class Graph
{
public:
	//part1: constructor initializes adjacency matrix
	Graph(int numVertex);
	//part2: returns the number of vertices in the graph
	int GetNumVertices();
	//part3: returns the number of edges in the graph
	int numberOfEdges();
	//part4: inserts edge going from one vertex to another
	void insertEdge(int frmVertex, int toVertex);
	//part5: removes edge going from one vertex to another
	void removeEdge((int frmVertex, int toVertex);
	//part6: returns the degree of the node passed
	int degree(int vertex);

	//part7: outputs the order in which vertices are visited during DFS
	//Starting from node s.
	void depthfirstSearch(int s);

	//part8: outputs the order in which vertices are visited during BFS
	//Starting from node s.
	void breadthfirstSearch(int s);


private:
	int **adj_matrix;
	int numVertices;

};
#endif
```

You may test your implementation using the following client

```
//Following is a sample client
#include<iostream>
#include "Graph.h"
using namespace std;
int main()
{
	Graph *g; //creating an object of graph with 5 vertices
	g=new Graph(5);

	//inserting edges in the graph
	g->inserEdge(0,1);
	g->inserEdge(0,4);
	g->inserEdge(1,0);
	g->inserEdge(1,2);
	g->inserEdge(1,3);
	g->inserEdge(1,4);
	g->inserEdge(2,1);
	g->inserEdge(2,3);
	g->inserEdge(3,1);
	g->inserEdge(3,2);
	g->inserEdge(3,4);
	g->inserEdge(4,0);
	g->inserEdge(4,1);
	g->inserEdge(4,3);

	//display total number of edges
	cout<<"Number of edges are "<<g->numberOfEdges()<<endl;

	//display degree of vertex number 4
	cout<<"Degree of vertex "<<g->degree(4)<<endl;

	cout<<"Output for Depth first search starting from vertex 0 "<<endl;
	g->depthfirstSearch(0);
	cout<<"Output for Breadth first search starting from vertex 0 "<<endl;
	g->breadthfirstSearch(0);
	return 0;
}
```

Post Lab: Implement the inLab Exercise1 using Adjacency List for representing the graph.

Appendix A: Lab Evaluation Criteria

Labs with projects

1. Experiments and their report 50%
    a. Experiment 60%
    b. Lab report 40%
2. Quizzes (3-4) 15%
3. Final evaluation 35%
    a. ProjectImplementation 60%
    b. Project report and quiz 40%

Labs without projects

1. Experiments and their report 50%
    a. Experiment 60%
    b. Lab report 40%
2. Quizzes (3-4) 20%
3. Final Evaluation 30%
    i. Experiment 60%
    ii. Lab report, pre and post experiment quiz 40%

Notice:

Copying and plagiarism of lab reports is a serious academic misconduct. First instance of copying may entail ZERO in that experiment. Second instance of copying may be reported to DC. This may result in awarding FAIL in the lab course.

Appendix B: Guidelines on Preparing Lab Reports

You will maintain a lab notebook for this lab course. You will write a report for each experiment you perform in his notebook. The lab report format is as follows:

1. **Introduction**: Introduce the new constructs/ commands being used, and their significance.
2. **Objective**: What are the learning goals of the experiment?
3. **Design**: If applicable, draw the flow chart for the program. How do the new constructs facilitate achievement of the objectives; if possible, a comparison in terms of efficacy and computational tractability with the alternate constructs?
4. **Issues**: The bugs encountered and the way they were removed.
5. **Conclusions**: What conclusions can be drawn from experiment?
6. **Application**: Suggest a real world application where this exercise may apply.
7. Answers to post lab questions (if any).

Sample Lab Report for Labs

Introduction

The ability to control the flow of the program, letting it make decisions on what code to execute, is important to the programmer. The if-else statement allows the programmer to control if a program enters a section of code or not based on whether a given condition is true or false. If-else statements control *conditional branching*.

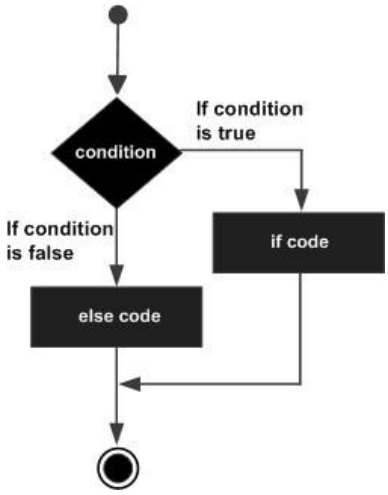
if (expression)

statement1

else

statement2

If the value of *expression* is nonzero, *statement1* is executed. If the optional **else** is present, *statement2* is executed if the value of *expression* is zero. In this lab, we use this construct to select an action based upon the user's input, or a predefined parameter.



Objective:

To use if-else statements for facilitation of programming objectives: A palindrome is a number or a text phrase that reads the same backward as forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. We have written a C++ program that reads in a five-digit integer and determines whether it is a palindrome.

Measurements:

The objective was achieved with the following code:

```
#include<iostream>

usingnamespacestd;

intmain()

{

   inti,temp,d,revrs=0;

   cout<<"enter the number to check :";

      cin>>i;

   temp=i;

   while(temp>0)

   {

      d=temp%10;
```

```
        temp/=10;
        revrs=revrs*10+d;
        }
    if(revrs==i)
    cout<<i<<" is palindorme";
    else
    cout<<i<<" is not palindrome";
    }
}
```

Screen shots of the output for various inputs are shown in Figure 1:

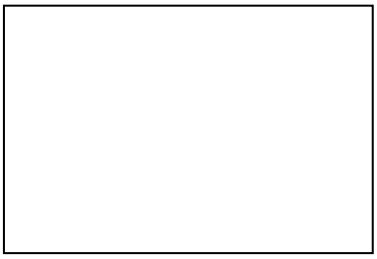


Fig.1. Screen shot of the output

The conditional statement made this implementation possible; without conditional branching, it is not possible to achieve this objective.

Issues:

Encountered bugs and issues; how were they identified and resolved.

Conclusions:

The output indicates correct execution of the code.

Applications:

If-else statements are a basic construct for programming to handle decisions.