LAB MANUAL

Course: CSC211-Data Structures and Algorithms

Department of Computer Science

Java Learning Procedure

1) Stage J (Journey inside-out the concept)
2) Stage a_1 (Apply the learned)
3) Stage V (Verify the accuracy)
4) Stage a_2 (Assess your work)

COMSATS Institute of Information Technology (CIIT)
Islamabad

Table of Contents

LAB # 01

Statement Purpose:

This lab will give you an overview of C++ language.

Activity Outcomes:

This lab teaches you the following topics:

- Basic syntax of C++
- Data types and operators in C++
- Control flow statements in C++
- Arrays and Functions

Instructor Note:

Introduction to Programming in C++

- https://ece.uwaterloo.ca/~dwharder/aads/Tutorial/
- http://www.cplusplus.com

1) Stage J (Journey)

Introduction

C++, as we all know is an extension to C language and was developed by **Bjarne stroustrup** at bell labs. C++ is an intermediate level language, as it comprises a confirmation of both high level and low level language features.

Following features of C++ makes it a stronger language than C,

1. There is Stronger Type Checking in C++.
2. C++ supports and allows user defined operators (i.e Operator Overloading) and function overloading is also supported in it.
3. Exception Handling is there in C++.
4. Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.
5. Variables can be declared anywhere in the program in C++, but must be declared before they are used.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a Hello World program in C++.

Solution:

```
#include <iostream>
using namespace std;

int main ()
{
  cout << "Hello World! ";
  cout << "I'm a C++ program";
}
```

Activity 2:

Write a program to use different data types in C++

Solution:

```
#include <iostream>
using namespace std;

int main ()
{
  float a=5.5;               // initial value: 5
  int b(3);                 // initial value: 3
  int c{2};                 // initial value: 2
  float result;               // initial value undetermined

  a = a + b;
  result = a - c;
  cout << result;

  return 0;
}
```

Activity 3:

Write a program to use string data type in C++

Solution:

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystring;
  mystring = "This is the initial string content";
  cout << mystring << endl;
  mystring = "This is a different string content";
  cout << mystring << endl;
  return 0;
}
```

Activity 4:

Write a program to use arithmetic operators in C++

Solution:

```
#include <iostream>
using namespace std;

int main ()
{
  int a, b=3;
  a = b;
  a+=2;              // equivalent to a=a+2
  cout << a;
}
```

Activity 5:

Write a program to use relational operators in C++

Solution:

```
#include <iostream>
using namespace std;

int main ()
{
  int a,b,c;

  a=2;
  b=7;
  c = (a>b) ? a : b;

  cout << c << '\n';
}
```

Activity 6:

Write a program to use if-else statement in C++

Solution:

```
#include <iostream>
using namespace std;

int main ()
{
int x;
cin>>x;
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
}
```

Activity 7:

Write a program to use while loop in C++

Solution:

```
#include <iostream>
```

```
using namespace std;

int main ()
{
  int n = 10;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "liftoff!\n";
}
```

Activity 8:

Write a program to use do-while loop in C++

Solution:

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
  do {
    cout << "Enter text: ";
    getline (cin,str);
    cout << "You entered: " << str << '\n';
  } while (str != "goodbye");
}
```

Activity 9:

Write a program that will add two numbers in a function and call that function in main.

Solution:

```
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return r;
}

int main ()
{
```

```
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
}
```

Activity 10:

Write a program to use the concept of Arrays in C++

Solution:

```
#include <iostream>
using namespace std;

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
  cout << result;
  return 0;
}
```

3) Stage V (verify)

Home Activities:

1. Use nested loops that print the following patterns in three separate programs.

```
Pattern II
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

```
Pattern III
          1
        2 1
      3 2 1
    4 3 2 1
  5 4 3 2 1
6 5 4 3 2 1
```

```
Pattern IV
1 2 3 4 5 6
  1 2 3 4 5
    1 2 3 4
      1 2 3
        1 2
          1
```

2. (Printing numbers in a pyramid pattern) Write a nested for loop that prints the following output:

```
            1
          1 2 1
        1 2 4 2 1
      1 2 4 8 4 2 1
    1 2 4 8 16 8 4 2 1
  1 2 4 8 16 32 16 8 4 2 1
1 2 4 8 16 32 64 32 16 8 4 2 1
```

1 2 4 8 16 32 64 128 64 32 16 8 4 2 1

4) Stage a_2 (assess)

Lab Assignment and Viva voce

1. Write a program that prompts the user to enter the number of students and each student's name and score, and finally displays **the student with the highest score**.
2. Write a program that displays all the numbers from 100 to 1000, **ten per line, that are divisible by 5 and 6**.
3. Write a C++ program to generate Fibonacci Series like given below using loop up to given number by user. If user enter number = 20, your output must be
   Fibonacci Series upto 20
   0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

[Hint: add the last two number in the series to get new number of the series**]**

LAB # 02

Statement Purpose:

The purpose of this lab session is to acquire skills in working with singly linked lists.

Activity Outcomes:

This lab teaches you the following topics:

- Creation of singly linked list
- Insertion in singly linked list
- Deletion from singly linked list
- Traversal of all nodes

Instructor Note:

1) Stage J (Journey)

Introduction

A *list* is a finite ordered set of elements of a certain type.
The elements of the list are called *cells* or *nodes*.
A list can be represented *statically*, using arrays or, more often, *dynamically*, by allocating and releasing memory as needed. In the case of static lists, the ordering is given implicitly by the one-dimension array. In the case of dynamic lists, the order of nodes is set by *pointers*. In this case, the cells are allocated dynamically in the heap of the program. Dynamic lists are typically called *linked lists*, and they can be singly- or doubly-linked.
The structure of a node may be:
typedef struct *nodetype*

```
{
  int key; /* an optional field */
  ... /* other useful data fields */
  struct nodetype *next; /* link to next node */
} NodeT;
```

A singly-linked list may be depicted as in Figure 1.1.

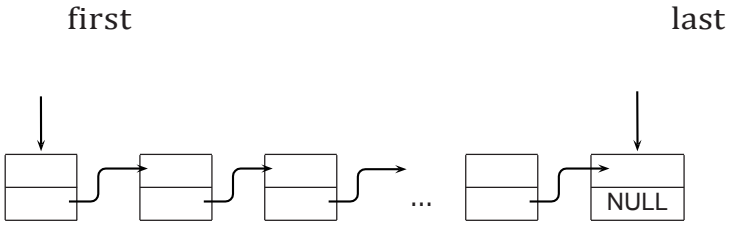


Figure 1.1.: A singly-linked list model.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Creation of linked list

Solution:

Consider the following steps which apply to *dynamic* lists:

1. Initially, the list is empty. This can be coded by setting the pointers to the first and last cells to the special value *NULL*, i.e. *first* = *NULL*, *last* = *NULL*.

2. Reserve space for a new node to be inserted in the list:

```
/* reserve space */
p = ( NodeT * )malloc( sizeof( NodeT ) );
```

   Then place the data into the node addressed by *p*. The implementation depends on the type of data the node holds. For primitive types, you can use an assignment such as **p = data*.

3. Make the necessary connections:

```
p->next = NULL;        /* node is appended to the list */
if ( last != NULL ) /* list is not empty */
last->next = p;
else
  first = p; /* first node */
last = p;
```

Activity 2:

Accessing nodes of linked list

Solution:

The nodes of a list may be accessed *sequentially*, gathering useful information as needed. Typically, a part of the infor- mation held at a node is used as a *key* for finding the desired information. The list is scanned linearly, and the node may or may not be present on the list. A function for searching a specific key may contain the following sequence of statements:

```
NodeT *p;
p = first;
while ( p != NULL )
  if ( p->key = givenKey )
  {
    return p; /* key found at address p */
```

```
}
else
  p = p->next;
return NULL; /* not found */
```

Activity 3:

Insertion in single linked list

Solution:

The node to be inserted may be created as shown at §1.3. We will assume here that the node to insert is pointed to by *p*.

- If the list was empty then there would be only one node, i.e.

```
if ( first == NULL )
{
  first = p;
  last = p;
  p->next = NULL;
}
```

- If the list was not empty, then insertion follows different patterns depending on the position where the node is to be inserted. Thus, we have the following cases:

1. Insertion before the first node of the list:

```
if ( first != NULL )
{
  p->next = first;
  first = p;
}
```

2. Insertion after the last node of the list (this operation is also called *append*):

```
if ( last != NULL )
{
  p->next = NULL;
  last->next
```

```
= p;
last
= p;
}
```

3. Insertion before a node given by its *key*. There are two steps to execute:
   a) Search for the node containing the *givenKey*:

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = first;
while ( q != NULL )
{
  if ( q->key == givenKey ) break;
  q1 = q;
  q = q->next;
}
```

   b) Insert the node pointed to by *p*, and adjust links:

```
if ( q != NULL )
{
 /* node with key givenKey has address q */
  if ( q == first )
  { /* insert after first node */
    p->next = first;
    first = p;
  }
  else
  {
    q1->next = p;
    p->next = q;
  }
}
```

4. Insertion after a node given by its *key*. Again, there are two steps to execute:
   a) Search for the node containing the *givenKey*:

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
```

```
q = first;
while ( q != NULL )
{
  if ( q->key == givenKey ) break;
  q1 = q;
  q = q->next;
}
```

b) Insert the node pointed to by *p*, and adjust links:

```
if ( q != NULL )
{
  p->next = q->next; /* node with key givenKey has address q */
  q->next = p;
  if ( q == last ) last = p;
}
```

Activity 4:

Deleting a node from single linked list

Solution:

When we are to remove a node from a list, there are some aspects to take into account: (i) list may be empty; (ii) list may contain a single node; (iii) list has more than one node. And, also, deletion of the first, the last or a node given by its key may be required. Thus we have the following cases:

1. Removing the first node of a list

```
NodeT *p;
if ( first != NULL )
{ /* non-empty list
     */
  p = first;
  first = first->next;
  free( p ); /* free up memory */
  if ( first == NULL ) /* list is now empty */
last = NULL;
```

```
}
```

2. Removing the last node of a list

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = first;
if ( q != NULL )
{  /* non–empty list */
  while ( q != last )
  {  /* advance towards end */
q1 = q;
    q = q–>next;
  }
  if ( q == first )
  { /* only one node */
    first = last = NULL;
  }
  else
  { /* more than one node */
    q1–>next = NULL;
    last = q1;
  }
  free( q );
}
```

3. Removing a node given by a *key*

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = first;
/* search node */
while ( q != NULL )
{
  if ( q–>key == givenKey ) break;
```

```
        q1 = q;
        q = q->next;
      }
      if ( q != NULL )
{   /* found a node with supplied
              key */
        if ( q == first )
        { /* is the first node */
          first = first->next;
          free( q ); /* release memory */
          if ( first == NULL ) last = NULL;
        }
        else
        { /* other than first node */
          q1->next = q->next;
          if ( q == last ) last = q1;
          free( q ); /* release memory */
        }
      }
```

Activity 5:

Complete deletion of single linked list

Solution:

For a complete deletion of a list, we have to remove each node of that list, i.e.

```
NodeT *p;
while ( first != NULL )
{
  p = first;
  first = first->next;
  free( p );
}
last = NULL;
```

3) Stage V (verify)

Home Activities:

1. Write a function that prints all nodes of a linked list in the reverse order.

2. Write a function which reverses the order of the linked list.

3. Write a function which rearranges the linked list by group nodes having even numbered and odd numbered value in their data part.

4. Write a function which takes two values as input from the user and searches them in the list. If both the values are found, your task is to swap both the nodes in which these values are found. Note, that you are not supposed to swap values.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 03

Statement Purpose:

In this lab session we will enhance singly-linked lists with another feature: we'll make them circular. And, as was the case in the previous lab session, we will show how to implement creation, insertion, and removal of nodes.

Activity Outcomes:

This lab teaches you the following topics:

- Creation of circular linked list
- Insertion in circular linked list
- Deletion from circular linked list
- Traversal of all nodes

Instructor Note:

1) Stage J (Journey)

Introduction

A *circular singly linked list* is a singly linked list which has the last element linked to the first element in the list. Being circular it really has no ends; then we'll use only one pointer *pNode* to indicate one element in the list – the newest element. Figure 3.1 show a model of such a list.

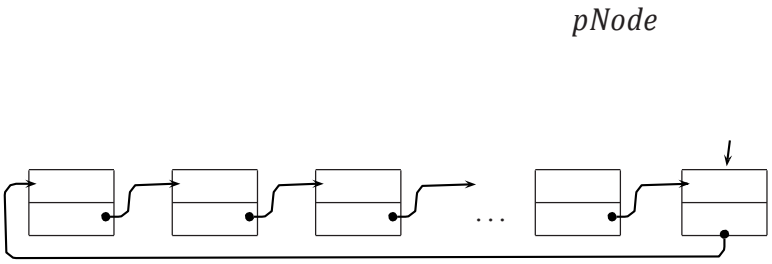


Figure 3.1.: A model of a circular singly-linked list.

The structure of a node may be:

```
typedef struct nodetype
{
  int key; /* an optional field */
  /* other useful data fields */
  struct nodetype *next;
  /* link to next node */
} NodeT;
```

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Creation of circular linked list

Solution:

1. Initially, the list is empty, i.e. *pNode* = *NULL*.

2. Generate a node to insert in the list:

```
/* reserve space */

p = ( NodeT* )malloc(
sizeof( NodeT ) ); Then
read the data into the
node addressed by p
```

3. Link it in the list:

```
p–>next = NULL;

/* node is appended to the list */

if ( pNode == NULL )

{ /*
  empt
  y lis
  t */
  pNode
  = p;
  pNode
  –>nex
  t = p;

}

else

{ /*
  nonempty
  list */
  p–>next =
  pNode–>nex
  t;
  pNode–>nex
  t = p;

  pNode = p;

  /* pNode points
    to the newest
    node in the lis
    t */

}
```

Activity 2:

Accessing nodes of circular linked list

Solution:

The nodes of a list may be accessed sequentially, starting at node *pNode*, as follows:

```
NodeT *p;

p = pNode;
```

```
if ( p != NULL )
  do
  {
    access current node and get data;
    p = p–>next;
  }
  while ( p != pNode );
```

Another choice is to look for a key, say *givenKey*. Code for such list scan is given below:

```
NodeT *p;
p = pNode;
if ( p != NULL )
  do
  {
    if ( p–>key = givenKey )
    { /* key found at address p */
      return p;
    }
    p = p–>next;
  }
  while ( p != NULL );
return NULL; /* not found */
```

Activity 3:

Insertion in circular linked list

Solution:

A node may be inserted *before* a node containing a given key, or *after* it. Both cases imply searching for that key, and, if that key exists, creating the node to insert, and adjusting links accordingly.

Inserting Before a node with key *givenKey*

There are two steps to execute:

1. Find the node with key *givenKey*:

```
NodeT *p, *q, *q1;

q1 = NULL; /* initialize */
q = pNode;
do
{
  q1 = q;
  q = q->next;
  if ( q->key == givenKey ) break;
}
while ( q != pNode );
```

2. Insert the node pointed to by *p*, and adjust links:

```
if ( q->key == givenKey )
{ /* node with key givenKey has address q */
  q1->next = p;
  p->next = q;
}
```

Insertion after a node with key

Again, there are two steps to execute:

1. Find the node with key *givenKey*:

```
NodeT *p, *q;

q = pNode;
do
{
  if ( q->key == givenKey ) break;
  q = q->next;
}
while ( q != pNode );
```

2. Insert the node pointed to by *p*, and adjust links:

```
if ( q->key == givenKey )
```

```
{ /* node with key givenKey has address q */
  p->next = q->next;
  q->next = p;
}
```

Activity 4:

Deleting a node from circular linked list

Solution:

Again there are two steps to take:

1. Find the node with key *givenKey*:
   *NodeT *p, *q, *q1*;
   q = pNode;
   do
   {
   q1 = q;
   q = q->next;
   if (*q->key == givenKey*) **break**;
   }
   while (*q* != *pNode*);
2. Delete the node pointed to by *q*. If that node is *pNode* then we adjust *pNode* to point to its previous.

```
if ( q->key == givenKey )
{ /* node with key givenKey has address q */
  if ( q == q->next )
  {
    /* list now empty */
  }
  else
  {
    q1->next = q->next;
    if ( q == pNode ) pNode = q1;
  }
  free( q );
```

```
}
```

Activity 5:

Complete Deletion of Circular Linked list

Solution:

For complete deletion of a list, we have to remove each node of that list, i.e.

```
NodeT *p, *p1;
p = pNode;
do
{
  p1 = p;
  p = p->next;
  free( p1 );
}
while ( p != pNode );
pNode = NULL;
```

3) Stage V (verify)

Home Activities:

1. Write a function that deletes all those nodes from a linked list which have even/odd numbered value in their data part.

2. Write a function that implements Josephus problem.

3. Write a function that deletes all even positioned nodes from a linked list. Last node should also be deleted if its position is even.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

Lab Assignment 1:

Define and implement functions for operating on the data structure given below and the model of figure 3.2.

```
struct circularList
{
  int length;
  NodeT *first;
}
```

Operations should be coded as: ins *data*=insert *data* in the list if it is not already there (check the key part of data area for that purpose), del *key*=delete data containing *key* from list (if it is on the list), ist *data*=insert node with *data* as first node, dst=delete first node, prt=print list, fnd *key*=find data node containing *key* in its data area.

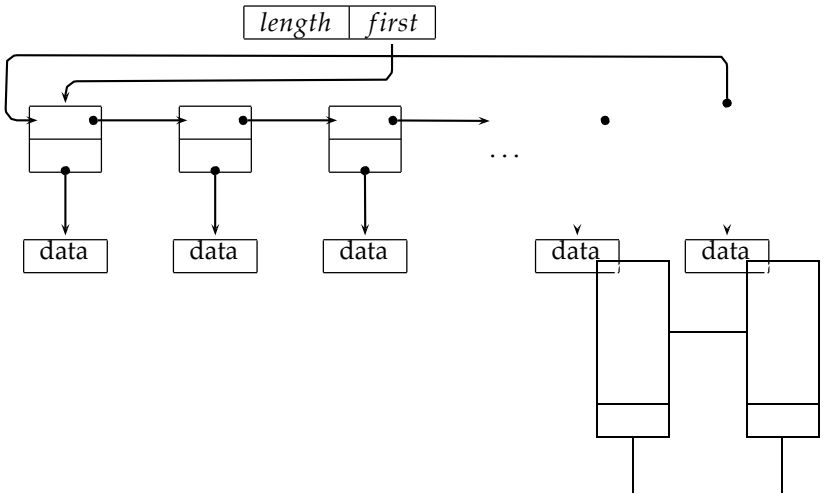


Figure 3.2.: Another model of a circular singly-linked list.

Lab Assignment 2:

Define and implement functions for operating on the data structure given below and the model of figure 3.3.

```
struct circularList
{
  int length;
  NodeT *current;
}
```

The field current changes to indicate: the last inserted node if the last operation was insert, the found node for find, the next node for delete, Operations should

be coded as: ins *data*=insert a node containing *data* in the list if an element with the same key is not already there, fnd *key*=find data node containing *key* in its data area (check the key part of data area for that purpose), del *key*=delete node with *key* in ints data area from list (if it is on the list), icr *data*=insert after current position, dcr=delete current node, prt=print list.

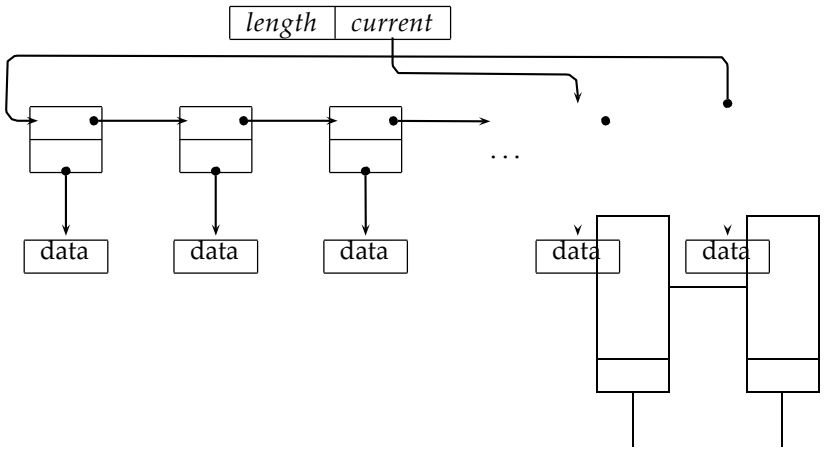


Figure 3.3-...: Yet another model of a circular singly-linked list.

LAB # 04

Statement Purpose:

This lab session is intended to help you develop the operations on doubly-linked lists.

Activity Outcomes:

This lab teaches you the following topics:

- Creation of doubly linked list
- Insertion in doubly linked list
- Deletion from doubly linked list
- Traversal of all nodes

Instructor Note:

1) Stage J (Journey)

Introduction

A *doubly-linked* list is a (dynamically allocated) list where the nodes feature two relationships: *successor* and *predecessor*. A model of such a list is given in figure 4.1. The type of a node in a doubly-linked list may be defined as follows:

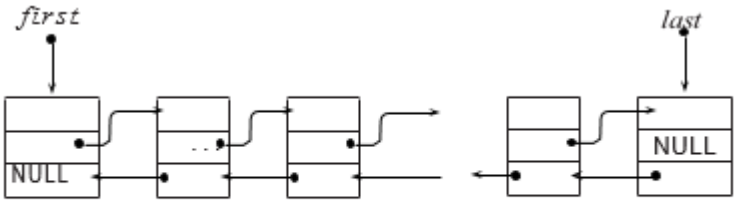


Figure 4.1.: A model of a doubly linked list.

```
typedef struct node_type
{
  KeyT key; /* optional */
  ValueT value;
  /* pointer to next node */
  struct node_type *next;
  /* pointer to previous node */
  struct node_type *prev;
} NodeT;
```

As we have seen when discussing singly-linked lists, the main operations for a doubly-linked list are:

- creating a cell;
- accessing a cell;
- inserting a new cell;
- deleting a cell;
- deleting the whole list.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Creation of doubly linked list

Solution:

In what follows we shall assume that the list is given by a pointer to its header cell, i.e.

```
/* header cell */
struct list_header
{
  NodeT *first;
  NodeT *last;
};
/* list is
   defined as a
   pointer to its
   header */
struct list_header *L;
```

Creating a Doubly Linked List

We shall take the list to be initially empty, i.e.

```
L->first = L->last = NULL;
```

After allocating space for a new node and filling in the data (the *value* field), insertion of this node, pointed by *p*, is done as follows:

```
if ( L->first == NULL )
{ /* empty list */
  L->first = L->last = p;
  p->next = p->prev = NULL;
}
else
{ /*
  nonemp
  ty list
```

```
   */
   L->last-
   >next =
   p;
   p->prev
   = L->l
   ast;
   L->last
   = p;
}
```

Activity 2:

Accessing nodes of doubly linked list

Solution:

Stating at a certain position (i.e. at a certain node) we can access a list:

- In sequential forward direction

```
for ( p = L->first; p != NULL; p = p->next )
{
  /* some operation o current cell */
}
```

- In sequential backward direction

```
for ( p = L->last; p != NULL; p->p->prev )
{
  /* some operation o current cell */
}
```

Activity 3:

Inserting node in doubly linked list

Solution:

We can insert a node before the first node in a list, after the last one, or at a position specified by a given key:

- before the first node:

```
if ( L->first == NULL )
{ /* empty list */
  L->first = L->last = p;
  p_>next = p->prev = NULL;
}
```

else

```
{ /*
  nonemp
  ty list
  */
  p->next
  = L->fir
  st;
  p->prev
  = NULL;
  L->first
  ->prev
  = p;
  L->firs
  t = p;
}
```

- after the last node:

```
if ( L->first == NULL )
{ /* empty list */
  L->first = L->last = p;
  p_>next = p->prev = NULL;
}
else
{ /*
  nonemp
  ty list
  */
  p->next
  = NULL;
  p->prev
  = L->l
  ast;
  L->last-
  >next =
  p; L->l
  ast = p;
}
```

- After a node given by its key:

```
p->prev = q;
p->next = q->next;
if ( q->next != NULL ) q->next->prev = p;q->next = p;
if ( L->last == q ) L->last = p;
```

Here we assumed that the node with the given key is present on list *L* and it we found it and placed a pointer to it in variable *q*.

Activity 4:

Deletion from doubly linked list

Solution:

When deleting a node we meet the following cases:

- Deleting the first node:

```
p = L->first;
L->first = L->first->next; /* nonempty list assumed */
free( p ); /* release memory taken by node */
if ( L->first == NULL )
  L->last == NULL; /* list became empty */
else
  L->first->prev = NULL;
```

- Deleting the last node:

```
p = L->last;
L->last = L->last->prev; /* nonempty list assumed */
if ( L->last == NULL )
  L->first = NULL; /* list became empty */
else
  L->last->next = NULL;
free( p ); /* release memory taken by node */
```

- Deleting a node given by its key. We will assume that the node of key *givenKey* exists and it is pointed to by *p* (as a result of searching for it)

```
if ( L->first == p && L->last == p )
{ /* list has a single node */
  L->first = NULL;
  L->last = NULL; /* list became empty */
  free( p );
}
else
if ( p == L->first )
{ /* deletion of
  first node */
  L->first = L->f
  irst->next; L->f
```

```
    irst->prev =
    NULL;

    free( p );

  }

  else

  { /* deletion of
    an inner node */
    p->next->prev =
    p->prev;
    p->prev->next =
    p->next;

    free( p );

  }
```

Activity 5:

Deleting complete doubly linked list

Solution:

Deleting a list completely means deleting each of its nodes, one by one.

```
NodeT *p;

while ( L->first != NULL )
{
  p = L->first;
  L->first = L->first->next;
  free( p );
}
L->last = NULL;
```

3) Stage V (verify)

Home Activities:

1. Write a function which reverses the order of a doubly linked list.

2. Write a function which rearranges the linked list by group nodes having even numbered and odd numbered value in their data part.

3. Write a function which takes two values as input from the user and searches them in the list. If both the values are found, your task is to swap both the nodes in which these values are found. Note, that you are not supposed to swap values.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 05

Statement Purpose:

This lab will introduce you the concept of Stack data structure

Activity Outcomes:

This lab teaches you the following topics:

- How to access the top of the stack
- How to push data onto the stack
- How to pop data from the stack

Instructor Note:

1) Stage J (Journey)

Introduction

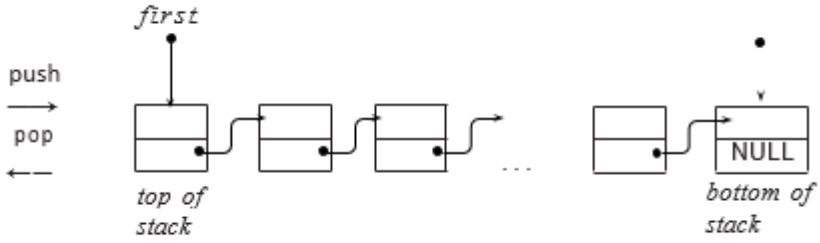
A *stack* is a special case of a singly-linked list which works according to LIFO algorithm. Access is restricted to one end, called the *top* of the stack. A *stack* may be depicted as given in the figure below . Its operations are:

push – push an element onto the top of the stack;

pop – pop an element from the top of the stack;

top – retrieve the element at the top of the stack;

delete – delete the whole stack. This can be done as explained in the previuos paragraph.



2) Stage a1 (apply)

Lab Activities:

Activity 1

Write a function that checks whether parentheses in a mathematical expression are balanced or not.

Activity 2

Write a function that converts a mathematical expression with no parentheses from infix form to postfix form

Activity 3

Write a function that converts an expression from infix form to prefix form.

3) Stage V (verify)

Home Activities:

Write a function that converts a mathematical expression containing parentheses from infix form to postfix form

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 06

Statement Purpose:

This lab will introduce you the concept of Queue data structure

Activity Outcomes:

This lab teaches you the following topics:

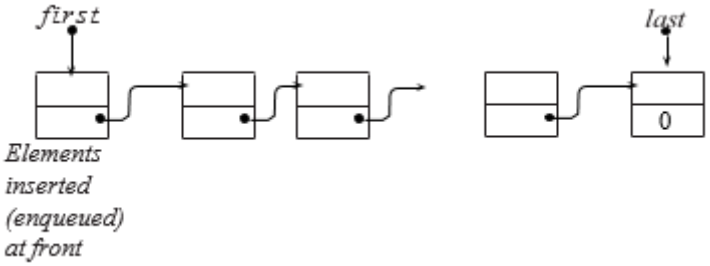
- How to access the front and rear pointers
- How to enqueuer/insert the data
- How to dequeuer/ delete the data

Instructor Note:

`1) Stage J (Journey)

Introduction

A *queue* is also a special case of a singly-linked list, which works according to FIFO[2] algorithm. 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 below:



The main operations are:

Enqueue – place an element at the tail of the queue;
Dequeue – take out an element form the front of the queue;
Delete – delete the whole queue

2) Stage a1 (apply)

Lab Activities:

3) Stage V (verify)

Home Activities:

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 07

Statement Purpose:

This lab will introduce you the concept of Recursion

Activity Outcomes:

This lab teaches you the following topics:

- How to formulate programs recursively.
- How to apply the three laws of recursion.
- How to implement the recursive formulation of a problem.
-

Instructor Note:

`1) Stage J (Journey)

Introduction

An algorithm is **recursive** if it calls itself to do part of its work. For this approach to be successful, the "call to itself" must be on a smaller problem then the one originally attempted. In general, a recursive algorithm must have two parts: the **base case**, which handles a simple input that can be solved without resorting to a recursive call, and the recursive part which contains one or more recursive calls to the algorithm where the parameters are in some sense "closer" to the base case than those of the original call.

We divide/decompose the problem into smaller (sub) problems:

- Keep on decomposing until we reach to the smallest sub-problem (base case) for which a solution is known or easy to find
- Then go back in reverse order and build upon the solutions of the sub-problems

A recursive function is defined in terms of base cases and recursive steps.

- In a **base case**, we compute the result immediately given the inputs to the function call.
- In a **recursive step**, we compute the result with the help of one or more recursive calls to this same function, but with the inputs somehow reduced in size or complexity, closer to a base case.

What Happens When a Method is called?

Activation records (AR) will store the following information about the method:

- Local variables of the method.
- Parameters passed to the method.
- Value returned to the calling code (if the method is not a void type).
- The location in the calling code of the instruction to execute after returning from the called method.

Two ways of thinking:

For something simple to start with – let's write a function pow(x, n) that raises x to a natural power of n. In other words, multiplies x by itself n times.

```
pow(2, 2) = 4
pow(2, 3) = 8
```

pow(2, 4) = 16
There are two ways to implement it.

Iterative thinking: the for loop:

```
function pow(x, n) {
  let result = 1;
  // multiply result by x n times in the loop
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
```

Recursive thinking: simplify the task and call self:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}
```

When pow(x, n) is called, the execution splits into two branches:

```
            if n==1  = x     (Base case)
           /
pow(x, n) =
           \
            else    = x * pow(x, n - 1)     (Recursive call)
```

If n == 1, then everything is trivial. It is called the base of recursion, because it immediately produces the obvious result: pow(x, 1) equals x.

Otherwise, we can represent pow(x, n) as x * pow(x, n - 1). In maths, one would write xn = x * xn-1. This is called a recursive step: we transform the task into a simpler action (multiplication by x) and a simpler call of the same task (powwith lower n). Next steps simplify it further and further until n reaches 1.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a recursive program to calculate factorial of a positive integer

Solution:

```
#include<iostream>
```

```
using namespace std;

int factorial(int n);

int main()
{
   int n;

   cout << "Enter a positive integer: ";
   cin >> n;

   cout << "Factorial of " << n << " = " << factorial(n);

   return 0;
}

int factorial(int n)
{
   if(n > 1)
      return n * factorial(n - 1);
   else
      return 1;
}
```

Activity 2:

Write a recursive program to find sum of an array elements

Solution:

```
#include <stdio.h>

// Return sum of elements in A[0..N-1]
// using recursion.
int findSum(int A[], int N)
{
   if (N <= 0)
      return 0;
   return (findSum(A, N - 1) + A[N - 1]);
```

```
}

int main()
{
   int A[] = { 1, 2, 3, 4, 5 };
   int N = sizeof(A) / sizeof(A[0]);
    cout<< findSum(A, N);
return 0;
}
```

Activity 3:

Write a recursive program to find reverse of an array

Solution:

```
#include <bits/stdc++.h>
using namespace std;

/* Function to reverse arr[] from start to end*/
void rvereseArray(int arr[], int start, int end)
{
   if (start >= end)
   return;

   int temp = arr[start];
   arr[start] = arr[end];
   arr[end] = temp;

   // Recursive Function calling
   rvereseArray(arr, start + 1, end - 1);
}


/* Utility function to print an array */
void printArray(int arr[], int size)
{
  for (int i = 0; i < size; i++)
  cout << arr[i] << " ";

  cout << endl;
}

/* Driver function to test above functions */
int main()
{
   int arr[] = {1, 2, 3, 4, 5, 6};

   // To print original array
```

```
 printArray(arr, 6);
// Function calling
   rvereseArray(arr, 0, 5);

   cout << "Reversed array is" << endl;

   // To print the Reversed array
   printArray(arr, 6);

   return 0;
}
```

Activity 4:

Write a recursive function to find the sum of array elements using binary recursion

Solution:

```
int BinarySum(int A[], int i, int n){
        if (n == 1)then                          // base case
          return A[i];
        else                                  // recursive case I
          return BinarySum(A, i, n/2) + BinarySum(A, i+n/2, n/2);
}
```

3) Stage V (verify)

Home Activities:

1) Write the recursive method to find the binary number of decimal number given by user.
2) Write the method which takes the integer value (n) as input and prints the sequence of numbers from 0 to n in ascending order.
3) Write the method which takes the integer value (n) as input and prints the sequence of numbers from n to 0 in descending order.
4) Write the method which takes the integer value (n) as input and prints the sequence of numbers from 0 to n (ascending order) and n to 0 (descending order).
5) Write a recursive function to compute first N Fibonacci numbers. Test and trace for N = 6

1 1 2 3 5 8

6) Write a recursive function that has a parameter representing a list of integers and returns the maximum stored in the list. Thinking recursively, the maximum is either the first value in the list or the maximum of the rest of the list, whichever is larger. If the list only has 1 integer, then its maximum is this single value, naturally.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

1) Implementation of **Euclid's algorithm** for finding the **largest common divisor** of two integers. It is based on the observation that the greatest common divisor of two integers **m** and **n** with **m** > **n** is the same as the greatest common divisor of **n** and **m mod n**.
2) Write the recursive method to find all the subsets of given string. Assume that if the given input is "abcd", the output will be:

Abcd, , abc, abd, ab, acd, ac, ad, a, bcd, bc, bd, b, cd, c, d

3) Write the computer program to solve the puzzle of "tower of Hanoi". The detailed description of the task is given on following link:
https://www.tutorialspoint.com/data_structures_algorithms/tower_of_hanoi.htm

4) Here is a simple recursive function to compute the Fibonacci sequence:

```
long fibr(int n) { // Recursive Fibonacci generator
   // fibr(46) is largest value that fits in a long Assert((n > 0) && (n
   < 47), "Input out of range"); if ((n == 1) || (n == 2)) return 1; //
   Base cases return fibr(n-1) + fibr(n-2);          // Recursion
}
```

This algorithm turns out to be very slow, calling **Fibr**a total of Fib(n) times. Contrast this with the following iterative algorithm:

```
long fibi(int n) { // Iterative Fibonacci generator
   // fibi(46) is largest value that fits in a long Assert((n > 0) && (n
   < 47), "Input out of range"); long past, prev, curr;   // Store
   temporary values past = prev = curr = 1;              // initialize

   for (int i=3; i<=n; i++) { // Compute next value past = prev;
                                        // past holds fibi(i-2)
      prev = curr;                      // prev holds fibi(i-1) curr = past
      + prev;                           // curr now holds fibi(i)
   }
```

```
    return curr;
}
```

Function **Fibi**executes the **for**loop $n - 2$ times.

(a) Which version is easier to understand? Why?
(b) Explain why **Fibr**is so much slower than **Fibi**.

LAB # 08

Statement Purpose:

This lab will present to you bubble sort, insertion sort and selection sort

Activity Outcomes:

This lab teaches you the following topics:

- How to sort the input data

Instructor Note:

`1) Stage J (Journey)

Introduction

Bubble Sort

Bubble sort is a simple and well-known sorting algorithm. It is used in practice once in a blue moon and its main application is to make an introduction to the sorting algorithms. Bubble sort belongs to O(n2) sorting algorithms, which makes it quite inefficient for sorting large data volumes. Bubble sort is stable and adaptive.

Algorithm

1. Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
2. If at least one swap has been done, repeat step 1.

Insertion Sort

Insertion sort belongs to the O(n2) sorting algorithms. Unlike many sorting algorithms with quadratic complexity, it is actually applied in practice for sorting small arrays of data. For instance, it is used to improve quicksort routine. Some sources notice, that people use same algorithm ordering items, for example, hand of cards.

Algorithm

Insertion sort algorithm somewhat resembles selection sort. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part contains **first element** of the array and unsorted one contains the rest. At every step, algorithm takes **first element** in the unsorted part and **inserts** it to the right place of the sorted one. When unsorted part becomes **empty**, algorithm *stops*.

Selection Sort

Selection sort is one of the O(n2) sorting algorithms, which makes it quite inefficient for sorting large data volumes. Selection sort is notable for its programming simplicity and it can over perform other sorts in certain situations (see complexity analysis for more details).

Algorithm

The idea of algorithm is quite simple. Array is imaginary divided into two parts - sorted one and unsorted one. At the beginning, sorted part is **empty**, while unsorted one contains **whole array**. *At every step,* algorithm finds **minimal element** in the unsorted part and adds it to the end of the sorted one. When unsorted part becomes **empty**, algorithm *stops*.

When algorithm sorts an array, it swaps first element of unsorted part with minimal element and then it is included to the sorted part. This implementation of selection sort in **not stable**.

In case of linked list is sorted, and, instead of swaps, minimal element is linked to the unsorted part, selection sort is **stable**.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a function to sort array elements using bubble sort

Solution:

```
Void bubbleSort(int arr[], int n)
{
    boolswapped = true;
    int j = 0;
    int tmp;
    while (swapped)
        {
        swapped = false;
        j++;
        for (inti = 0; i<n - j; i++)
                {
            if (arr[i] >arr[i + 1])
                        {
                tmp = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = tmp;
                swapped = true;
            }
        }
    }
}
```

Activity 2:

Write a function to sort array elements using insertion sort

Solution:

```
Void insertionSort(int arr[], int length)
{
    inti, j, tmp;
```

```
    for (i = 1; i<length; i++)
        {
        j = i;
        while (j> 0 &&arr[j - 1] >arr[j])
                {
            tmp = arr[j];
            arr[j] = arr[j - 1];
            arr[j - 1] = tmp;
            j--;
        }
    }
}
```

Activity 3:

Write a function to sort array elements using selection sort

Solution:

```
Void selectionSort (int arr[], int n)
{
    inti, j, minIndex, tmp;
    for (i = 0; i<n - 1; i++)
        {
        minIndex = i;
        for (j = i + 1; j<n; j++)
            if (arr[j] <arr[minIndex])
                minIndex = j;
        if (minIndex != i)
                {
            tmp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = tmp;
        }
    }
}
```

3) Stage V (verify)

Home Activities:

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 09

Statement Purpose:

This lab will teaches you Quick and Merge sort

Activity Outcomes:

This lab teaches you the following topics:

- How to sort the input data using Quick and Merge sort

Instructor Note:

`1) Stage J (Journey)

Introduction

Quick Sort

Quicksort is a fast sorting algorithm, which is used not only for educational purposes, but widely applied in practice. On the average, it has O(n log n) complexity, making quicksort suitable for sorting big data volumes. The idea of the algorithm is quite simple and once you realize it, you can write quicksort as fast as bubble sort.

Algorithm

The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

1. Choose a pivot value. We take the value of the middle element as pivot value, but it can be any value, which is in range of sorted values, even if it doesn't present in the array.
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 ibecomes 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.

Merge Sort

Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with subproblems, we state each subproblem as sorting a subarrayA[p .. r]. Initially, p = 1 and r = n, but these values change as we recurse through subproblems.

Algorithm

To sort A[p .. r]:

1. Divide Step

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A[p .. r] into two subarrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That is, q is the halfway point of A[p .. r].

2. Conquer Step

Conquer by recursively sorting the two subarraysA[p .. q] and A[q + 1 .. r].

3. Combine Step

Combine the elements back in A[p .. r] by merging the two sorted subarrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Note that the recursion bottoms out when the subarray has just one element, so that it is trivially sorted.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a function to sort array elements using quick sort

Solution:

```
Void quickSort(intarr[], int left, int right)
{
      inti = left, j = right;
      inttmp;
      intpivot = arr[(left + right) / 2];

      /* partition */
      while (i<= j)
      {
            while (arr[i] <pivot)
                  i++;
            while (arr[j] >pivot)
                  j--;
            if (i<= j)
             {
                  tmp = arr[i];
                  arr[i] = arr[j];
                  arr[j] = tmp;
                  i++;
                  j--;
            }
      };

      /* recursion */
      if (left<j)
```

```
            quickSort(arr, left, j);
      if (i<right)
            quickSort(arr, i, right);
}
```

Activity 2:

Write a function to sort array elements using merge sort

Solution:

```
Void mergeSort(int numbers[], int temp[], intarray_size)

{
     m_sort(numbers, temp, 0, array_size - 1);
}

void m_sort(int numbers[], int temp[], int left, int right)
{
     int mid;

     if (right > left)
     {
       mid = (right + left) / 2;
       m_sort(numbers, temp, left, mid);
       m_sort(numbers, temp, mid+1, right);

       merge(numbers, temp, left, mid+1, right);
     }
}

// m - size of A
// n - size of B
// size of C array must be equal or greater than
// m + n
void merge(int m, int n, int A[], int B[], int C[]) {
      inti, j, k;
      i = 0;
      j = 0;
      k = 0;
      while (i< m && j < n)
      {
            if (A[i] <= B[j])
            {
                  C[k] = A[i];
                  i++;
            } else {
                  C[k] = B[j];
                  j++;
            }
            k++;
      }
      if (i< m)
      {
            for (int p = i; p < m; p++)
            {
```

```
                    C[k] = A[p];
                    k++;
              }
        } else
        {
              for (int p = j; p < n; p++)
              {
                    C[k] = B[p];
                    k++;
              }
        }
}
```

3) Stage V (verify)

Home Activities:

4) Stage a_2 (assess)

Lab Assignment and Viva voce

LAB # 10

Statement Purpose:

This lab will introduce you the concept of Hashing

Activity Outcomes:

This lab teaches you the following topics:

- How to implement hash function
- How to create hash tables
- How to insert, search and retrieve values from hash tables

Instructor Note:

`1) Stage J (Journey)

Introduction

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given key to a smaller number and uses the small number as index in a table called hash table.

Example:

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. Insert a phone number and corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.
2. Linked List of phone numbers and records.
3. Balanced binary search tree with phone numbers as keys.
4. Direct Access Table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With **balanced binary search tree**, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(Logn) time.

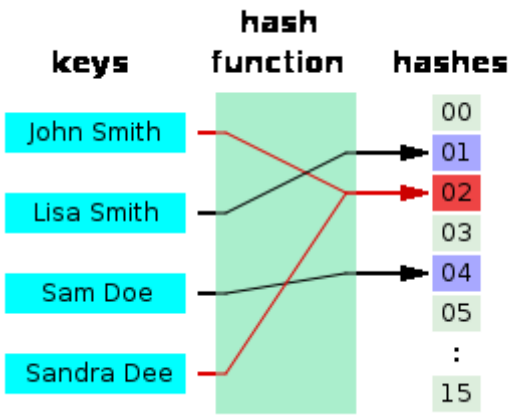
Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in O(1) time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table. This solution has many practical limitations. First problem with this solution is extra space required is huge. For example if phone number is n digits, we need $O(m * 10^n)$ space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits.

Due to above limitations Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well compared

to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

Hash Function:

A function that converts a given big number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table. For example, a person's name, having a variable length, could be hashed to a single integer. The values returned by a hash function are called hash values, hash codes, hash sums, checksums or simply hashes as shown in figure below:



A good hash function should have following properties

1) Efficiently computable

2) Should uniformly distribute the keys (Each table position equally likely for each key)

Hash Table:

An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- **Chaining:** The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

- **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a function to compute a simple hash function

Solution:

```
int key;
int SIZE=20;
int hashCode(int key){
   return key % SIZE;
}
```

Activity 2:

Write a function to insert an item in a hash table

Solution:

```
#define SIZE 20

struct DataItem {
   int data;
   int key;
};

struct DataItem* hashArray[SIZE];
```

```
struct DataItem* dummyItem;
struct DataItem* item;

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

struct DataItem *search(int key) {
   //get the hash
   int hashIndex = hashCode(key);
void insert(int key,int data) {
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;

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

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

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

   hashArray[hashIndex] = item;
}
```

Activity 3:

Write a function to search an item from hash table

Solution:

```
#define SIZE 20

struct DataItem {
   int data;
   int key;
};

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

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

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

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

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

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

      //go to next cell
      ++hashIndex;

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

   return NULL;
}
```

Activity 4:

Write a complete program to create a hash table using simple division hash function. Also perform searching and deletion in hash table.

Solution:

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

#define SIZE 20

struct DataItem {
  int data;
```

```
   int key;
};

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

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

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

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

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

      //go to next cell
      ++hashIndex;

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

   return NULL;
```

```
}

void insert(int key,int data) {

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

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

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

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

   hashArray [hashIndex] = item;
}

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

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

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

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

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

      //go to next cell
      ++hashIndex;

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

   return NULL;
}

void display() {
   int i = 0;

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

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

```
      printf(" ~~ ");
   }

   printf("\n");
}

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

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

   display();
   item = search(37);

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

```
  delete(item);
  item = search(37);

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

3) Stage V (verify)

Home Activities:

1) Consider "key mod 7" as a hash function and a sequence of keys as 50, 700, 76, 85, 92, 73, 101. Create hash table for to store these keys using linear probing.
2) Use mid square hash function to store the data of employees of a company. Employee ids are:
   4176, 5678, 5469, 1245, 8907
3) Apply folding method to store the above data.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

1) Consider the data with keys: 24, 42, 34, 62, 73. Store this data into a hash table of size 10 using quadratic probing to avoid collision.
2) Store the keys **18, 26, 35, 9, 64, 47, 96, 36, and 70** in an empty hash table of size **13 using double hashing.** Use h(key) = key % 13 as a first hash function while the second hash function is: h_p(key) = 1 + key % 12

LAB # 11

Statement Purpose:

This lab will introduce you the concept of Binary Trees

Activity Outcomes:

This lab teaches you the following topics:

- How to insert and delete data from binary trees
- How to traverse and search the binary trees

Instructor Note:

`1) Stage J (Journey)

Introduction

Non-Linear Data Structure-Trees

A tree is a Non-Linear Data Structure which consists of set of nodes called vertices and set of edges which links vertices

A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.

Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.

Recursive definition:

A binary tree is a finite set of nodes that is either empty or consists of a root and maximum of two disjoint binary trees called the left subtree and the right subtree. The binary tree may also have only right or left subtree.

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minumum effort

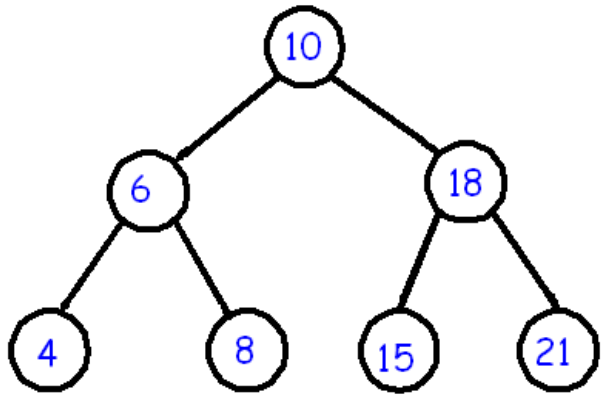
Terminology:

- **Root Node:** The starting node of a tree is called Root node of that tree
- **Terminal Nodes:** The node which has no children is said to be terminal node or leaf .
- **Non-Terminal Node:** The nodes which have children is said to be Non-Terminal Nodes
- **Degree:** The degree of a node is number of sub trees of that node
- **Depth:** The length of largest path from root to terminals is said to be depth or height of the tree
- **Siblings:** The children of same parent are said to be siblings

- **Ancestors:** The ancestors of a node are all the nodes along the path from the root to the node

Binary Search Trees

We consider a particular kind of a binary tree called a Binary Search Tree (BST). The basic idea behind this data structure is to have such a storing repository that provides the efficient way of data sorting, searching and retrieving.



A BST is a binary tree where nodes are ordered in the following way:

- each node contains three parts (left subtree, right subtree and data part
- the data/keys in the left subtree are less then the data/key in its parent node, in short L < P;
- the keys in the right subtree are greater the key in its parent node, in short P < R;

In the left tree all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree > 10. Because both the left and right subtrees are again BST; the above definition is recursively applied to all internal nodes:

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a complete program to insert data into binary search tree.

Solution:

```
// Structure to store a Binary Search Tree node
struct Node
{
    int data;
```

```
    Node *left, *right;
};

// Function to create a new binary tree node having given key
Node* newNode(int key)
{
    Node* node = new Node;
    node->data = key;
    node->left = node->right = NULL;

    return node;
}

// Recursive function to insert an key into BST using a reference parameter
void insert(Node* &root, int key)
{
    // if the root is null, create a new node an return it
    if (root == NULL)
    {
        root = newNode(key);
        return;
    }

    // if given key is less than the root node, recurse for left subtree
    // else recurse for right subtree
    if (key < root->data)
        insert(root->left, key);
    else    // key >= root->data
        insert(root->right, key);
}

// main function
int main()
{
    Node* root = NULL;
    int keys[] = { 15, 10, 20, 8, 12, 16, 25 };

     for (int i=0; i<7;i++)
        insert(root, keys[i]);

     return 0;
}
```

Activity 2:

Write functions for in-order, pre-order and post-order traversal of binary search trees.

Solution:

Pre-order Traversal

```
void preorder(BTNode *p)
{
        if(p != 0)
        {
                cout<<p->info;
                preorder(p->left);
                preorder(p->right);
        }
}
```

In-order Traversal

```
void inorder(BTNode *p)
{
        if(p != 0)
        {
                inorder(p->left);
                cout<<p->info;
                inorder(p->right);
        }
}
```

Post-order Traversal

```
void postorder(BTNode *p)
{
        if(p != 0)
        {
                postorder(p->left);
                postorder(p->right);
                cout<<p->info;
        }
}
```

Activity 3:

Write a function to search data in a binary tree.

Solution:

```
void search(Node* root, int key, Node* parent)
{
    // if key is not present in the key
    if (root == nullptr)
    {
        cout << "Key Not found";
        return;
    }

    // if key is found
    if (root->data == key)
    {
        if (parent == nullptr)
            cout << "The node with key " << key << " is root node";

        else if (key < parent->data)
            cout << "Given key is left node of node with key "  << parent->data;

        else cout << "Given key is right node of node with key " << parent->data;

        return;
    }

    // if given key is less than the root node, recurse for left subtree
    // else recurse for right subtree

    if (key < root->data)
        return search(root->left, key, root);

    return search(root->right, key, root);
}
```

Activity 4:

Write a function to delete a node from binary tree.

Solution:

```
// Function to delete node from a BST
void deleteNode(Node*& root, int key)
{
    // pointer to store parent node of current node
    Node* parent = nullptr;

    // start with root node
    Node* curr = root;

    // search key in BST and set its parent pointer
```

```
searchKey(curr, key, parent);

// return if key is not found in the tree
if (curr == nullptr)
    return;

// Case 1: node to be deleted has no children i.e. it is a leaf node
if (curr->left == nullptr && curr->right == nullptr)
{
    // if node to be deleted is not a root node, then set its
    // parent left/right child to null
    if (curr != root)
    {
        if (parent->left == curr)
            parent->left = nullptr;
        else
            parent->right = nullptr;
    }
    // if tree has only root node, delete it and set root to null
    else
        root = nullptr;

    // deallocate the memory
    free(curr);     // or delete curr;
}

// Case 2: node to be deleted has two children
else if (curr->left && curr->right)
{
    // find its in-order successor node
    Node* successor  = minimumKey(curr->right);

    // store successor value
    int val = successor->data;

    // recursively delete the successor. Note that the successor
    // will have at-most one child (right child)
    deleteNode(root, successor->data);

    // Copy the value of successor to current node
    curr->data = val;
}

// Case 3: node to be deleted has only one child
else
{
    // find child node
    Node* child = (curr->left)? curr->left: curr->right;


    // if node to be deleted is not a root node, then set its parent to its child
```

```
        if (curr != root)
        {
            if (curr == parent->left)
                parent->left = child;
            else
                parent->right = child;
        }

        // if node to be deleted is root node, then set the root to child
        else
            root = child;

        // deallocate the memory
        free(curr);
    }
}
```

Activity 5:

Write a function to find minimum value in a subtree of binary search tree.

Solution:

```
// Helper function to find minimum value node in subtree rooted at curr
Node* minimumKey(Node* curr)
{
    while(curr->left != nullptr) {
        curr = curr->left;
    }
    return curr;
}
```

3) Stage V (verify)

Home Activities:

Consider the BST in which you have stored the record of COMSATS' students. The data part of the BST is:

reg_num, name, address, phone_no and gpa

Write the following methods:

- Insert the record of new student (data is inserted by order of reg_num which is a unique number generated by **rand**() mehod).
- Search the record of student
- Display the record of student in ascending order
- Display record of student having highest GPA
- Calculate the average GPA of class

- Delete the record of specific student

1. What are the minimum and maximum number of elements in a heap of height *h*?
2. Where in a max-heap might the smallest element reside?
3. Show the max-heap that results from running **buildHeap** on the following values stored in an array:

10 5 12 3 2 1 8 7 9 4

4. **(a)** Show the heap that results from deleting the maximum value from the max-heap of Figure 5.20b.
   (b) Show the heap that results from deleting the element with value 5 from the max-heap of Figure 5.20b.
5. Revise the heap definition of Figure 5.19 to implement a min-heap. The member function **removemax** should be replaced by a new function called **removemin**.
6. Build the Huffman coding tree and determine the codes for the following set of letters and weights:

Letter	A	B	C	D	E	F	G	H	I	J	K	L
Frequency	2	3	5	7	11	13	17	19	23	31	37	41

What is the expected length in bits of a message containing *n* characters for this frequency distribution?

7. What will the Huffman coding tree look like for a set of sixteen characters all with equal weight? What is the average code length for a letter in this case? How does this differ from the smallest possible fixed length code for sixteen characters?
8. A set of characters with varying weights is assigned Huffman codes. If one of the characters is assigned code 001, then,
   (a) Describe all codes that *cannot* have been assigned.
   (b) Describe all codes that *must* have been assigned.
9. Assume that a sample alphabet has the following weights:

Letter	Q	Z	F	M	T	S	O	E
Frequency	2	3	10	10	10	15	20	30

(a) For this alphabet, what is the worst-case number of bits required by the Huffman code for a string of *n* letters? What string(s) have the worst- case performance?

(b) For this alphabet, what is the best-case number of bits required by the Huffman code for a string of *n* letters? What string(s) have the best- case performance?

(c) What is the average number of bits required by a character using the Huffman code for this alphabet?

10. You must keep track of some data. Your options are:

(1) A linked-list maintained in sorted order.

(2) A linked-list of unsorted records.

(3) A binary search tree.

(4) An array-based list maintained in sorted order.

(5) An array-based list of unsorted records.

For each of the following scenarios, which of these choices would be best? Explain your answer.

a) The records are guaranteed to arrive already sorted from lowest to highest (i.e., whenever a record is inserted, its key value will always be greater than that of the last record inserted). A total of 1000 inserts will be interspersed with 1000 searches.

b) The records arrive with values having a uniform random distribution (so the BST is likely to be well balanced). 1,000,000 insertions are performed, followed by 10 searches.

c) The records arrive with values having a uniform random distribution (so the BST is likely to be well balanced). 1000 insertions are interspersed with 1000 searches.

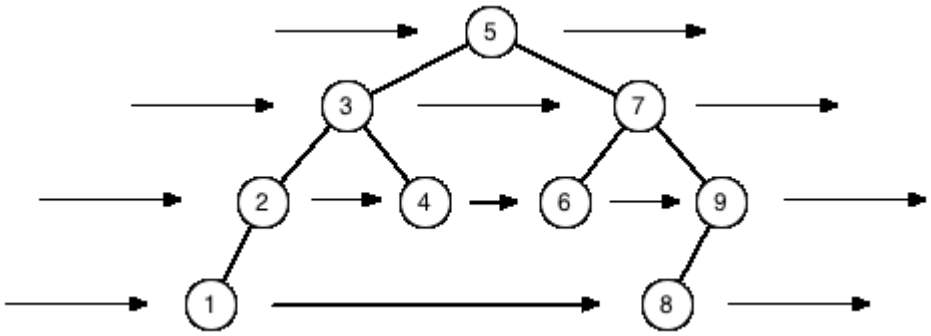
d) The records arrive with values having a uniform random distribution (so the BST is likely to be well balanced). 1000 insertions are performed, followed by 1,000,000 searches.

4) Stage a_2 (assess)

Lab Assignment and Viva voce

1) Write a C function that determines whether a given binary tree is a binary search tree.
2) Implement the BSTreeNumLeaves() which returns a count of the number of leaf nodes in the BSTree. A leaf node is any node with empty left and right subtrees.

3) Implement the BSTreeLevelOrder() function which prints the values in the BSTree in level-order on a single line separated by spaces (i.e. the same format as the other traverse-and-print functions). The following diagram aims to give an idea of how level-order traversal scans the nodes in a tree:



The output from this traversal would be 5 3 7 2 4 6 9 1 8.

Level-order traversal cannot be done recursively (at least not easily) and is typically implemented using a queue. The algorithm is roughly as follows:

```
Level Order Traversal (BSTree T):
        initialise queue
        add T's root node to queue
        WHILE the queue still has some entries DO
                take the head of the queue
                print the value from its BSTree node
                add the left child (if any) to the queue
                add the right child (if any) to the queue
        END
```

END

4. write a function treeLeaves :: BinaryTree a -> [a]. This function should take a binary tree and return a list containing the leaves of that tree—that is, all the nodes with no subtrees beneath them.
5. The successor and predecessor methods take a node pointer as input and return the node pointer to the item that is the successor or predecessor to the input item. They should return NULL only if the input item is the max or min respectively. Each may return a pointer to a node of the same value if there are multiple copies of that value in the tree; either way, successive calls to each method should eventually move beyond the copies of that shared value.
6. Write the method to find the height the height of the tree.
7. Assume that a given BST stores integer values in its nodes. Write a recursive function that traverses a binary tree, and prints the value of every node who's grandparent has a value that is a multiple of five.
8. Write a function that prints out the node values for a BST in sorted order from highest to lowest.
9. Write a recursive function named **smallcount** that, given the pointer to the root of a BST and a key *K*, returns the number of nodes having key values less than or equal to *K*. Function **smallcount**should visit as few nodes in the BST as possible.

10. Write a recursive function named **printRange** that, given the pointer to the root to a BST, a low key value, and a high key value, prints in sorted order all records whose key values fall between the two given keys. Function **printRange** should visit as few nodes in the BST as possible.

LAB # 12

Statement Purpose:

This lab will introduce you the concept of AVL trees

Activity Outcomes:

This lab teaches you the following topics:

- How to insert and delete data from AVL trees
- How to search and traverse AVL trees

Instructor Note:

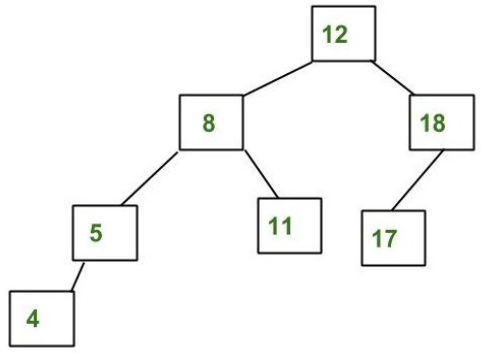
`1) Stage J (Journey)

Introduction

AVL Tree:

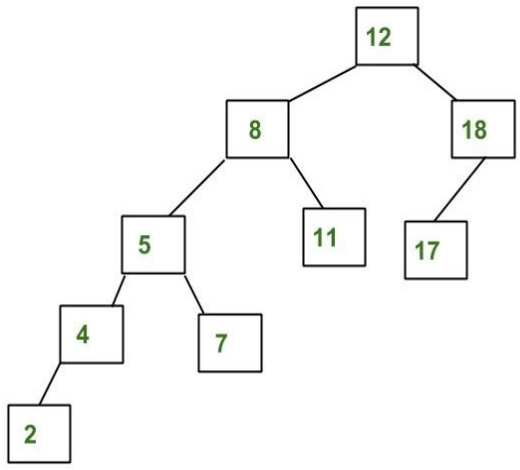
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

An Example Tree that is an AVL Tree



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

An Example Tree that is NOT an AVL Tree



The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where

h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a program to insert a node in AVL tree while balancing.

Solution:

```
// C program to insert a node in AVL tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct Node
{
   int key;
   struct Node *left;
   struct Node *right;
   int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get the height of the tree
int height(struct Node *N)
{
   if (N == NULL)
       return 0;
   return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
   return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct Node* newNode(int key)
{
```

```
    struct Node* node = (struct Node*)
                        malloc(sizeof(struct Node));
    node->key   = key;
    node->left   = NULL;
    node->right  = NULL;
    node->height = 1;  // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    //  Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
```

```
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int key)
{
    /* 1.  Perform the normal BST insertion */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
                           height(node->right));

    /* 3. Get the balance factor of this ancestor
          node to check whether this node became
          unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left =  leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
```

```
    /* return the (unchanged) node pointer */
    return node;
}

// A utility function to print preorder traversal
// of the tree.
// The function also prints height of every node
void preOrder(struct Node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
  struct Node *root = NULL;


  root = insert(root, 10);
  root = insert(root, 20);
  root = insert(root, 30);
  root = insert(root, 40);
  root = insert(root, 50);
  root = insert(root, 25);

  /* The constructed AVL Tree would be
            30
           /  \
         20   40
        /  \     \
       10  25    50
  */

  printf("Preorder traversal of the constructed AVL"
         " tree is \n");
  preOrder(root);

  return 0;
}
```

Activity 2:

Write a function to delete a node from an AVL tree and display the updated tree.

Solution:

```
// C program to delete a node from AVL Tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct Node
{
   int key;
   struct Node *left;
   struct Node *right;
   int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct Node *N)
{
   if (N == NULL)
       return 0;
   return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
   return (a > b)? a : b;
}
/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct Node* newNode(int key)
{
   struct Node* node = (struct Node*)
                       malloc(sizeof(struct Node));
   node->key   = key;
   node->left   = NULL;
   node->right  = NULL;
   node->height = 1;  // new node is initially added at leaf
   return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct Node *rightRotate(struct Node *y)
{
   struct Node *x = y->left;
   struct Node *T2 = x->right;

   // Perform rotation
   x->right = y;
   y->left = T2;
```

```
    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    //  Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(struct Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct Node* insert(struct Node* node, int key)
{
    /* 1.  Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys not allowed
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
                           height(node->right));
```

```
    /* 3. Get the balance factor of this ancestor
          node to check whether this node became
          unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left =  leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the
   node with minimum key value found in that tree.
   Note that the entire tree does not need to be
   searched. */
struct Node * minValueNode(struct Node* node)
{
    struct Node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

// Recursive function to delete a node with given key
// from subtree with given root. It returns root of
// the modified subtree.
```

```
struct Node* deleteNode(struct Node* root, int key)
{
   // STEP 1: PERFORM STANDARD BST DELETE

   if (root == NULL)
      return root;

   // If the key to be deleted is smaller than the
   // root's key, then it lies in left subtree
   if ( key < root->key )
      root->left = deleteNode(root->left, key);

   // If the key to be deleted is greater than the
   // root's key, then it lies in right subtree
   else if( key > root->key )
      root->right = deleteNode(root->right, key);

   // if key is same as root's key, then This is
   // the node to be deleted
   else
   {
      // node with only one child or no child
      if( (root->left == NULL) || (root->right == NULL) )
      {
         struct Node *temp = root->left ? root->left :
                                    root->right;

         // No child case
         if (temp == NULL)
         {
            temp = root;
            root = NULL;
         }
         else // One child case
          *root = *temp; // Copy the contents of
                    // the non-empty child
         free(temp);
      }
      else
      {
         // node with two children: Get the inorder
         // successor (smallest in the right subtree)
         struct Node* temp = minValueNode(root->right);

         // Copy the inorder successor's data to this node
         root->key = temp->key;

         // Delete the inorder successor
         root->right = deleteNode(root->right, temp->key);
      }
   }
```

```
    // If the tree had only one node then return
    if (root == NULL)
      return root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root->height = 1 + max(height(root->left),
                           height(root->right));

    // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to
    // check whether this node became unbalanced)
    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0)
    {
        root->left =  leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// A utility function to print preorder traversal of
// the tree.
// The function also prints height of every node
void preOrder(struct Node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

```
/* Driver program to test above function*/
int main()
{
  struct Node *root = NULL;


    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);

    /* The constructed AVL Tree would be
            9
           /  \
          1    10
        /  \     \
       0    5     11
      /    /  \
     -1   2    6
    */

    printf("Preorder traversal of the constructed AVL "
           "tree is \n");
    preOrder(root);

    root = deleteNode(root, 10);

    /* The AVL Tree after deletion of 10
            1
           /  \
          0    9
        /     /  \
       -1    5     11
           /  \
          2    6
    */

    printf("\nPreorder traversal after deletion of 10 \n");
    preOrder(root);

    return 0;
}
```

Activity 3:

Write a function to search in AVL tree.

Solution:

Algorithm Search(x, root)

{

if(tree is empty) then print" tree is empty"

otherwise

If(x grater than root) search(root-right);

Otherwise if(x less than root) search(root-left)

Otherwise return true

}

}

```
int avltree[40],t=1,s,x,i;
void insert(int,int );
int search1(int,int);


void main()
{
 Insert(root, 9);
Insert(root. 7);
Search1(root,7);
}
void insert(int s,int ch )
{
        int x,y;
        if(t==1)
        {
                avltree[t++]=ch;
                return;
        }
        x=search1(s,ch);
        if(avltree[x]>ch)
            {   avltree[2*x]=ch;
                y=log(2*x)/log(2);
                if(height(1,y))
                {
                if( x%2==0 )
                  update1();
                 else
```

```
                update2();
                }

            }
        else  {
                avltree[2*x+1]=ch;
                 y=log(2*x)/log(2);
                if(height(1,y))
                {
                if(x%2==1)
                  update1();
                else
                  update2();
                }
            }

        t++;
}

int search1(int s,int ch)
{
        if(t==1)
        {
                cout <<"no element in avltree";
                return -1;
        }
        if(avltree[s]==-1)
        return s/2;
        if(avltree[s] > ch)
        search1(2*s,ch);
        else search1(2*s+1,ch);
}
int height(int s,int y)
{
        if(avltree[s]==-1)
                return;

}
```

3) Stage V (verify)

Home Activities:

1. Search the minimum number in AVL tree

2. Search the maximum number in the AVL tree

3. Search the minimum subtree in the AVL tree

4) Stage a_2 (assess)

Lab Assignment and Viva voce

1. Write a program to insert data in Red Black trees.
2. Write a program to delete a node from red black trees and display the tree in post order.
3. Write a program to search an element in red black trees.

LAB # 13

Statement Purpose:

This lab will introduce you the concept of Heap data structure

Activity Outcomes:

This lab teaches you the following topics:

- How to insert and delete data from heap
- How to find minimum value

Instructor Note:

`1) Stage J (Journey)

Introduction

Why Heaps?

Queues are a standard mechanism for ordering tasks on a first-come, first-served basis. In many situations, simple queues are inadequate, since first in/first out scheduling has to be overruled using some priority criteria like:

- In a sequence of processes, process P_2 may need to be executed before process P_1 for the proper functioning of a system, even though P_1 was put on the queue of waiting processes before P_2.
- Banks managing customer information often will remove or get the information about the customer with the minimum bank account balance.
- In shared printer queue list of print jobs, must get next job with highest *priority*, and higher-priority jobs always print before lower-priority jobs.

In situations like these, a modified queue or *priority queue,* is needed in which elements are dequeued according to their priority and their current queue positions. The problem with a priority queue is in finding an efficient implementation which allows relatively fast enqueuing and dequeuing. Since elements may arrive randomly to the queue, there is no guarantee that the front elements will be the most likely to be dequeued and that the elements put at the end will be the last candidates for dequeuing. The situation is complicated because a wide spectrum of possible priority criteria can be used in different cases.

Consider an example of a printer; if the three jobs have been submitted to print, the jobs have sizes 100, 10, 1 page. The average waiting time for FIFO service, (100+110+111)/3 = 107 time units and average waiting time for shortest job first service, (1+11+112)/3 = 41 time units.

So a heap is an excellent way to implement a priority queue as **binary heap** is a complete or almost complete **binary tree** which satisfies the **heap** ordering property. The ordering can be one of two types: the min-**heap** property: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root and max-heap each node is smaller than its parent node.

Heaps

Heap has the following properties:

- The value of each node is not less than the values stored in each of its children
- The tree is an almost complete binary tree.

These two properties define a ***max heap*** and if "less" in the first property is replaced with "greater"; then the definition specifies a ***min heap.***

Heap is generally preferred for priority queue implementation by array list because it provide better performance as a method **getHighestPriority()** to access the element with highest priority can be implemented in **O(1)** time, method to **insert()** new element can be implemented in **O(Logn)** time and **deleteHighestPriority()** can also be implemented in **O(Logn)** time.

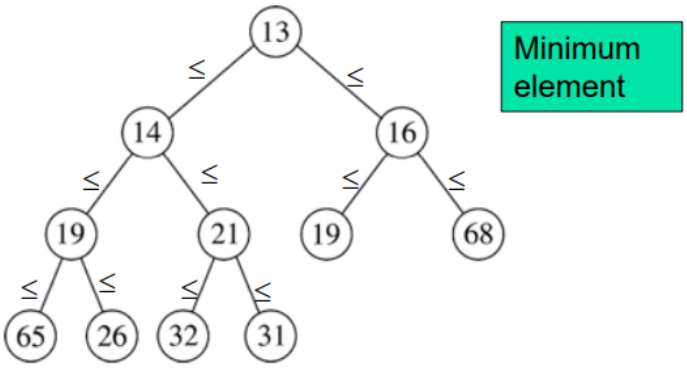
Applications of Priority Queue:

1) CPU Scheduling
2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3) All queue applications where priority is involved

A Binary Heap is a Binary Tree with following properties:
1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

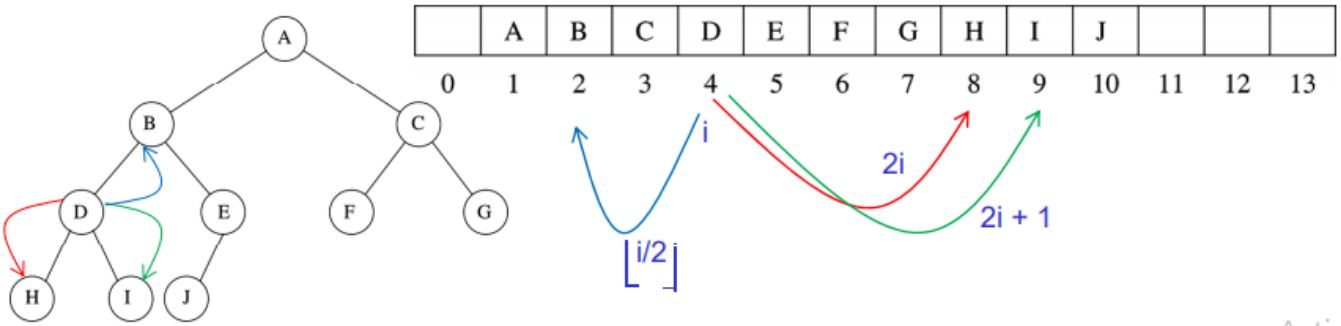
Examples of Min-Heap



How is Binary Heap represented?

Given element at position i in the array:

- Left child (i) = at position 2i
- Right child(i) = at position 2i + 1
- Parent(i) = at position $\lfloor i / 2 \rfloor$



2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a program to insert and delete the data from a heap while maintaining its properties.

Solution:

```
// A C++ program to demonstrate common Binary Heap Operations
#include<iostream>
#include<climits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    // Constructor
    MinHeap(int capacity);

    // to heapify a subtree with the root at given index
    void MinHeapify(int );

    int parent(int i) { return (i-1)/2; }

    // to get index of left child of node at index i
```

```
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to extract the root which is the minimum element
    int extractMin();

    // Decreases key value of key at index i to new_val
    void decreaseKey(int i, int new_val);

    // Returns the minimum key (key at root) from min heap
    int getMin() { return harr[0]; }

    // Deletes a key stored at index i
    void deleteKey(int i);

    // Inserts a new key 'k'
    void insertKey(int k);
};

// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(int cap)
{
    heap_size = 0;
    capacity = cap;
    harr = new int[cap];
}

// Inserts a new key 'k'
void MinHeap::insertKey(int k)
{
    if (heap_size == capacity)
    {
        cout << "\nOverflow: Could not insertKey\n";
        return;
    }

    // First insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the min heap property if it is violated
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}
```

```
// Decreases value of key at index 'i' to new_val.  It is assumed that
// new_val is smaller than harr[i].
void MinHeap::decreaseKey(int i, int new_val)
{
    harr[i] = new_val;
    while (i != 0 && harr[parent(i)] > harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    if (heap_size <= 0)
        return INT_MAX;
    if (heap_size == 1)
    {
        heap_size--;
        return harr[0];
    }

    // Store the minimum value, and remove it from heap
    int root = harr[0];
    harr[0] = harr[heap_size-1];
    heap_size--;
    MinHeapify(0);

    return root;
}


// This function deletes key at index i. It first reduced value to minus
// infinite, then calls extractMin()
void MinHeap::deleteKey(int i)
{
    decreaseKey(i, INT_MIN);
    extractMin();
}



// A recursive method to heapify a subtree with the root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
```

```
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Driver program to test above functions
int main()
{
    MinHeap h(11);
    h.insertKey(3);
    h.insertKey(2);
    h.deleteKey(1);
    h.insertKey(15);
    h.insertKey(5);
    h.insertKey(4);
    h.insertKey(45);
    cout << h.extractMin() << " ";
    cout << h.getMin() << " ";
    h.decreaseKey(2, 1);
    cout << h.getMin();
    return 0;
}
```

3) Stage V (verify)

Home Activities:

1. Revise the heap definition of above class to implement a max-heap. The member function **removemin** should be replaced by a new function called **removemax**.

2. Build the Huffman coding tree and determine the codes for the

following set of letters and weights:

Letter	A	B	C	D	E	F	G	H	I	J	K	L
Frequency	2	3	5	7	11	13	17	19	23	31	37	41

4) Stage a_2 (assess)

Lab Assignment and Viva voce

1. Implement a priority queue class based on the max-heap class implementation. The following methods should be supported for manipulating the priority queue:

- **void enqueue(int ObjectID, int priority);**
- **int dequeue();**
- **void changeweight(int ObjectID, int newPriority);**

Method **enqueue** inserts a new object into the priority queue with ID number **ObjectID** and priority **priority**. Method **dequeue** removes the object with highest priority from the priority queue and returns its object ID. Method **changeweight** changes the priority of the object with ID number **ObjectID** to be **newPriority**. The type for **Elem** should be a class that stores the object ID and the priority for that object. You will need a mechanism for finding the position of the desired object within the heap. Use an array, storing the object with **ObjectID** *i* in position *i*. (Be sure in your testing to keep the **ObjectID**s within the array bounds.) You must also modify the heap implementation to store the object's position in the auxiliary array so that updates to objects in the heap can be updated as well in the array.

2. You are required to design a computer program for a printer. Suppose there are N printing Jobs in a queue to be done, and each job has its own priority. The job with maximum priority (less number of pages) will get completed first than others. At each instant we are completing a job with maximum priority and at the same time we are also interested in inserting a new job in the queue with its own priority. Your maintained priority queue must fulfill the following conditions:

- **getHighestPriority()** should be implemented in **O(1)** time
- **insert()** should be implemented in **O(Logn)** time
- **deleteHighestPriority()** should also be implemented in **O(Logn)** time.

LAB # 14

Statement Purpose:

This lab will introduce you the concept of graph data structure

Activity Outcomes:

This lab teaches you the following topics:

- How to create a graph
- How to search a graph using breadth and depth first search

Instructor Note:

`1) Stage J (Journey)

Introduction

A graph G consists of a set V, whose members are called the vertices of G, together with a set E of pairs of distinct vertices from V. These pairs are called the edge of G. If e = v;w. is an edge with vertices v and w, then v and w are said to lie on e, and e is said to be incident with v and w. If the pairs are unordered, then G is called an undirected graph; if the pairs are ordered, then G is called a directed graph. The term directed graph is often shortened to digraph, and the unqualified term graph usually means undirected graph. The natural way to picture a graph is to represent vertices as points or circles and edges as line segments or arcs connecting the vertices. If the graph is directed, then the line segments or arcs have arrowheads indicating the direction.

Some Basic terms:

- **Degree** of a vertex, v, denoted $deg(v)$ is the number of incident edges.
- **Parallel edges** or multiple edges are edges of the same type and end-vertices
- **Self-loop** is an edge with the end vertices the same vertex
- **Simple graphs** have **no** parallel edges or self-loops
- **Path** is a sequence of alternating vetches and edges such that each successive vertex is connected by the edge. Frequently only the vertices are listed especially if there are no parallel edges.
- **Cycle** is a path that starts and ends at the same vertex.
- **Simple path** is a path with distinct vertices.
- **Directed path** is a path of only directed edges
- **Directed cycle** is a cycle of only directed edges.
- **Sub-graph** is a subset of vertices and edges.
- **Spanning sub-graph** contains all the vertices.
- **Connected graph** has all pairs of vertices connected by at least one path.
- **Connected component** is the maximal connected sub-graph of a unconnected graph.
- **Forest** is a graph without cycles.

Representation of Graph

Graph is a mathematical structure and finds its application is many areas, where the problem is to be solved by computers. The problems related to graph G must be represented in computer memory using any suitable data structure to solve the same. There are two standard ways of maintaining a graph G in the memory of a computer.

- Sequential representation of a graph using adjacent

Operations on Graph

Suppose a graph G is maintained in memory by the linked list representation. Basic operations are such as creating a graph, searching, deleting a vertices or edges.

a) Creating A Graph

To create a graph, first adjacency list array is created to store the vertices name, dynamically at the run time. Then the node is created and linked to the list array if an edge is there to the vertex.

b) Searching And Deleting From A Graph

Suppose an edge (1, 2) is to be deleted from the graph G. First we will search through the list array whether the initial vertex of the edge is in list array or not by incrementing the list array index. Once the initial vertex is found in the list array, the corresponding link list will be search for the terminal vertex.

Traversing a Graph

Many application of graph requires a structured system to examine the vertices and edges of a graph G. That is a graph traversal, which means visiting all the nodes of the graph. There are two graph traversal methods.

a) Breadth First Search

Given an input graph G = (V, E) and a source vertex S, from where the searching starts. The breadth first search systematically traverses the edges of G to explore every vertex that is reachable from S. Then we examine the entire vertices neighbor to source vertex S. Then we traverse all the neighbors of the neighbors of source vertex S and so on. A queue is used to keep track of the progress of traversing the neighbor nodes.

b) Depth First Search

The depth first search (DFS), as its name suggest, is to search deeper in the graph, whenever possible. Given an input graph G = (V, E) and a source vertex S, from where the searching starts. First we visit the starting node. Then we travel through each node along a path, which begins at S. That is we visit a neighbor vertex of S and again a neighbor of a neighbor of S, and so on. The implementation of BFS is almost same except a stack is used instead of the queue.

2) Stage a1 (apply)

Lab Activities:

Activity 1:

Write a program to create a graph.

Solution:

```
#include<iostream.h>
#include<conio.h>

class graph
{
      private:int n;
            int **a;
            int *reach;
            int *pos;
      public:graph(int k=10);
            void create();
            void dfs();
            void dfs(int v,int label);
            int begin(int v);
            int nextvert(int v);
};
void graph::graph(int k)
{
      n=k;
      a=newint *[n+1];
      reach=newint[n+1];
      pos=newint [n+1];
      for(int i=1;i<=n;i++)
            pos[i]=0;
      for(int j=1;j<=n;j++)
            a[j]=newint[n+1];
}
void graph::create()
{
      for(int i=1;i<=n;i++)
      {
            cout<<"Enter the "<<i<<"th row of matrix a:
";
            for(int j=1;j<=n;j++)
                  cin>>a[i][j];
      }
      for(int k=1;k<=n;k++)
            reach[k]=0;
}
void graph::dfs()
{
      int label=0;
      for(int i=1;i<=n;i++)
            if(!reach[i])
            {
                  label++;
                  dfs(i,label);
            }
      cout<<"
The contents of the reach array is:
;
      for(int j=1;j<=n;j++)
            cout<<reach[j]<<" ";
}
void graph::dfs(int v,int label)
{
      cout<<v<<"  ";
      reach[v]=label;
      int u=begin(v);
      while(u)
```

```
        {
                if(!reach[u])
                        dfs(u,label);
                u=nextvert(v);
        }
}
int graph::begin(int v)
{
        if((v<1)&&(v>n))
                cout<<"Bad input
";
        else
                for(int i=1;i<=n;i++)
                        if(a[v][i]==1)
                        {
                                pos[v]=i;
                                return i;
                        }
        return 0;
}
int graph::nextvert(int v)
{
        if((v<1)&&(v>n))
                cout<<"Bad input
";
        else
                for(int i=pos[v]+1;i<=n;i++)
                        if(a[v][i]==1)
                        {
                                pos[v]=i;
                                return i;
                        }
        return 0;
}
void main()
{
        clrscr();
        int x;
        cout<<"Enter the no of vertices:
";
        cin>>x;
        graph g(x);
        g.create();
        cout<<"dfs is.....";
        g.dfs();
        getch();

}
```

Activity 2:

Write a program to search a graph using breadth and depth first search.

Solution:

```
#include<conio.h>
#include<iostream.h>
#include<stdlib.h>

void create();  // For creating a graph
void dfs();  // For Deapth First Search(DFS) Traversal.
void bfs();  // For Breadth First Search(BFS) Traversal.
```

```
struct node  // Structure for elements in the graph
{
int data,status;
struct node *next;
struct link *adj;
};

struct link  // Structure for adjacency list
{
struct node *next;
struct link *adj;
};

struct node *start,*p,*q;
struct link *l,*k;

int main()
{
int choice;
   clrscr();
   create();
while(1)
   {
      cout<<"

------------------------";
      cout<<"
1: DFS
2: BSF
3: Exit
Enter your choice: ";
      cin>>choice;
switch(choice)
      {
      case 1:
          dfs();
      break;
      case 2:
          bfs();
      break;
      case 3:
          exit(0);
      break;
      default:
          cout<<"
Incorrect choice!
Re-enter your choice.";
          getch();
      }
   }
return 0;
}

void create()    // Creating a graph
{
int dat,flag=0;
   start=NULL;
   cout<<"
Enter the nodes in the graph(0 to end): ";
while(1)
   {
      cin>>dat;
```

```
if(dat==0)
      break;
      p=new node;
      p->data=dat;
      p->status=0;
      p->next=NULL;
      p->adj=NULL;
if(flag==0)
      {
       start=p;
       q=p;
       flag++;
      }
else
      {
       q->next=p;
       q=p;
      }
   }
   p=start;
while(p!=NULL)
   {
      cout<<"Enter the links to "<<p->data<<" (0 to end) : ";
      flag=0;
while(1)
      {
       cin>>dat;
      if(dat==0)
      break;
       k=new link;
       k->adj=NULL;
      if(flag==0)
       {
          p->adj=k;
          l=k;
          flag++;
       }
      else
       {
          l->adj=k;
          l=k;
       }
       q=start;
      while(q!=NULL)
       {
      if(q->data==dat)
             k->next=q;
          q=q->next;
       }
      }
      p=p->next;
   }
   cout<<"

-------------------------";
return;
}


// Deapth First Search(DFS) Traversal.
void bfs()
{
int qu[20],i=1,j=0;
```

```
   p=start;
while(p!=NULL)
   {
      p->status=0;
      p=p->next;
   }
   p=start;
   qu[0]=p->data;
   p->status=1;
while(1)
   {
if(qu[j]==0)
      break;
      p=start;
while(p!=NULL)
      {
      if(p->data==qu[j])
      break;
       p=p->next;
      }
      k=p->adj;
while(k!=NULL)
      {
       q=k->next;
      if(q->status==0)
       {
          qu[i]=q->data;
          q->status=1;
          qu[i+1]=0;
          i++;
       }
       k=k->adj;
      }
      j++;
   }
   j=0;
   cout<<"

Breadth First Search Results
";
   cout<<"
---------------------------
";
while(qu[j]!=0)
   {
      cout<<qu[j]<<"  ";
      j++;
   }
   getch();
return;
}


// Breadth First Search(BFS) Traversal.
void dfs()
{
int stack[25],top=1;
   cout<<"

Deapth First Search Results
";
   cout<<"
---------------------------
```

```
";
   p=start;
while(p!=NULL)
   {
      p->status=0;
      p=p->next;
   }
   p=start;
   stack[0]=0;
   stack[top]=p->data;
   p->status=1;
while(1)
   {
if(stack[top]==0)
      break;
      p=start;
while(p!=NULL)
      {
      if(p->data==stack[top])
      break;
       p=p->next;
      }
      cout<<stack[top]<<"  ";
      top--;
      k=p->adj;
while(k!=NULL)
      {
       q=k->next;
      if(q->status==0)
       {
          top++;
          stack[top]=q->data;
          q->status=1;
       }
       k=k->adj;
      }
   }
   getch();
return;
}
```

3) Stage V (verify)

Home Activities:

1. Following is the algorithm for finding shortest path in a graph. Implement it in C/C++.

   Set V = {V1, V2, V3 ..... Vn} contains the vertices and the edges E = {e1, e2, ...em} of the graph G. W(e) is the weight of an edge e, which contains the vertices V1 and V2. Q is a set of vertices, which are not visited. m is the vertex in Q for which weight W(m) is minimum, i.e., minimum cost edge. S is a source vertex.

1. Input the source vertices and assign it to S

    (a) Set W(s) = 0 and

    (b) Set W (v) = ___ for all vertices V is not equal to S

2. Set Q = V which is a set of vertices in the graph

3. Suppose m be a vertices in Q for which W(m) is minimum

4. Make the vertices m as visited and delete it from the set Q

5. Find the vertices I which are incident with m and member of Q (That is the vertices which are not visited)

6. Update the weight of vertices I = {i1, i2 ..... ik} by

    (a) W(i1) = min [W(i1), W(m) + W(m, i1)]

7. If any changes is made in W(v), store the vertices to corresponding vertices i, using the array, for tracing the shortest path

8. Repeat the process from step 3 to 7 until the set Q is empty

9. Exit

4) Stage a_2 (assess)

Lab Assignment and Viva voce

1. Implement dijkstra's algorithm to find the shortest path in a graph.