Assignment 1

Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client‘s telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers

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

#define size 13

class linear_probing {
   int hash[size], flag[size];

public:
   linear_probing() {
      int i;
      for (i = 0; i < size; i++) {
         flag[i] = 0;
      }
   }

   void insert(int x) {
      int i, loc;
      loc = x % size;
      for (i = 0; i < size; i++) {
         if (flag[loc] == 0) {
            hash[loc] = x;
            flag[loc] = 1;
            break;
         }
         else {
            loc = (loc + 1) % size;
         }
      }
   }

   void create() {
      int x, i, n;
      cout << "\nEnter the number of phone numbers to be inserted:\n";
      cin >> n;
      cout << "\nEnter the Phone numbers to be inserted in Hash table ->\n";
      for (i = 0; i < n; i++) {
```

```
            cin >> x;
            insert(x);
        }
    }

    int find(int x) {
        int loc = x % size;
        int i;
        for (i = 0; i < size; i++) {
            if (flag[loc] == 1 && hash[loc] == x) {
                return loc;
            }
            else
                loc = (loc + 1) % size;
        }
        return -1;
    }

    void search() {
        int x, loc;
        cout << "\nEnter the phone number to be search:\n";
        cin >> x;
        if ((loc = find(x)) == -1)
            cout << "\nPhone number is not found\n";
        else
            cout << "\nPhone number " << x << " is found at " << loc << "th location\n\n";
    }

    void print() {
        int i;
        cout << "\nHash Table is ->"
             << endl;
        for (i = 0; i < size; i++) {
            cout << "(" << i << ") -> ";
            if (flag[i] == 1) {
                cout << hash[i] << "\n";
            }
            else if (flag[i] == 0)
                cout << "----\n";
        }
    }
};


struct node {
    node *next;
```

```
    int phone;
};

class Chaining {
    node *hash[size];

public:
    Chaining() {
        int i;
        for (i = 0; i < size; i++)
            hash[i] = NULL;
    }

    void create() {
        int x, n, i;
        cout << "\nEnter the number of phone numbers to be inserted :\n";
        cin >> n;
        cout << "\nEnter the Phone numbers to be inserted in Hash table ->\n";
        for (i = 0; i < n; i++) {
            cin >> x;
            insert(x);
        }
    }

    void insert(int key) {
        int loc = key % size;
        node *p = new node;
        p->phone = key;
        p->next = NULL;
        if (hash[loc] == NULL) {
            hash[loc] = p;
        }
        else {
            node *q = hash[loc];
            while (q->next != NULL) {
                q = q->next;
            }
            q->next = p;
        }
    }

    void display() {
        int i;
        cout << "\nHash Table is -> :\n";
        for (i = 0; i < size; i++) {
            node *q = hash[i];
```

```
            cout << "(" << i << ") -> ";
            while (q) {
                cout << q->phone;
                if (q->next)
                    cout << " -> ";
                q = q->next;
            }
            cout << "\n";
        }
    }
};


int main() {
    cout << "----------------- Linear Probing -----------------\n";
    linear_probing lp;
    lp.create();
    lp.print();
    lp.search();

    cout << "\n----------------- Separate Chaining -----------------\n";
    Chaining h;
    h.create();
    h.display();
    return 0;
}


/*
----------------- Linear Probing -----------------
Enter the number of phone numbers to be inserted:
5
Enter the phone numbers to be inserted in hash table ->
123 136 284 297 448
Hash Table is ->
(0) -> ----
(1) -> ----
(2) -> ----
(3) -> ----
(4) -> ----
(5) -> ----
(6) -> 123
(7) -> 136
(8) -> 448
(9) -> ----
(10) -> ----
```

(11) -> 284
(12) -> 297
Enter the phone number to be search:
448
Phone number 448 is found at 8th location
---------- Separate Chaining ----------
Enter the number of phone numbers to be inserted :
5
Enter the Phone numbers to be inserted in Hash table
123 136 284 297 448
Hash Table is -> :
(0) ->
(1) ->
(2) ->
(3) ->
(4) ->
(5) ->
(6) -> 123 -> 136 -> 448
(7) ->
(8) ->
(9) ->
(10) ->
(11) -> 284 -> 297
(12) ->
*/

Assignment 02

Problem Statement:

Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with / without replacement.

Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique

Program:

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

class HashFunction
  {
     typedef struct hash

  {

  long key;
  char name[10];
 }hash;
 hash h[10];
   public:
 HashFunction();
 void insert();
 void display();
 int find(long);
 void Delete(long);


  };


HashFunction::HashFunction()
  {
 int i;
 for(i=0;i<10;i++)
   {
  h[i].key=-1;
```

```
  strcpy(h[i].name,"NULL");
   }
  }


void HashFunction::Delete(long k)
  {
 int index=find(k);
 if(index==-1)
   {
  cout<<"\n\tKey Not Found";
   }
 else
   {
  h[index].key=-1;
  strcpy(h[index].name,"NULL");
  cout<<"\n\tKey is Deleted";
   }
}


int HashFunction::find(long k)
  {
 int i;
 for(i=0;i<10;i++)
   {

  if(h[i].key==k)

 {
   cout<<"\n\t"<<h[i].key<<" is Found at "<<i<<" Location With Name "<<h[i].name;
   return i;
    }
   }
 if(i==10)                  // Key not found
     {
  return -1;
   }
  return 0;
  }


void HashFunction::display()
  {

 int i;
 cout<<"\n\t\tKey\t\tName";
```

```
 for(i=0;i<10;i++)
    {

  cout<<"\n\th["<<i<<"]\t"<<h[i].key<<"\t\t"<<h[i].name;
   }
  }


void HashFunction::insert()
  {

 char ans,n[10],ntemp[10];
 long k,temp;
 int v,hi,cnt=0,flag=0,i;

 do
{

  if(cnt>=10)
    {
   cout<<"\n\tHash Table is FULL";
   break;
    }

  cout<<"\n\tEnter a Telephone No: ";
  cin>>k;
  cout<<"\n\tEnter a Client Name: ";
  cin>>n;

  hi=k%10;// hash function

  if(h[hi].key==-1)
    {

   h[hi].key=k;
   strcpy(h[hi].name,n);
    }

      else

     {

   if(h[hi].key%10!=hi)

            {
             temp=h[hi].key;
```

```
            strcpy(ntemp,h[hi].name);

             h[hi].key=k;

             strcpy(h[hi].name,n);


            for(i=hi+1;i<10;i++)

             {

             if(h[i].key==-1)

              {
                h[i].key=temp;
                 strcpy(h[i].name,ntemp);
                 flag=1;
                 break;

              }
            }


              for(i=0;i<hi && flag==0;i++)
                   {

                     if(h[i].key==-1)
                      {
                       h[i].key=temp;
                       strcpy(h[i].name,ntemp);
                        break;
                      }
                   }
            }

else
    {

for(i=hi+1;i<10;i++)
       {
    if(h[i].key==-1)
       {
     h[i].key=k;
     strcpy(h[i].name,n);
     flag=1;
     break;
       }
```

```
        }


    for(i=0;i<hi && flag==0;i++)
       {
     if(h[i].key==-1)
        {
      h[i].key=k;
      strcpy(h[i].name,n);
      break;
         }
         }
     }

    }


      flag=0;
      cnt++;
      cout<<"\n\t..... Do You Want to Insert More Key: y/n";
      cin>>ans;

}while(ans=='y'||ans=='Y');


}



int main()
  {
 long k;
 int ch,index;
 char ans;
 HashFunction obj;
 do
   {
  cout<<"\n\t***** Telephone (ADT) *****";
  cout<<"\n\t1. Insert\n\t2. Display\n\t3. Find\n\t4. Delete\n\t5. Exit";
  cout<<"\n\t..... Enter Your Choice: ";
  cin>>ch;
  switch(ch)
    {
   case 1:  obj.insert();
     break;
```

```
 case 2: obj.display();
   break;

 case 3: cout<<"\n\tEnter a Key Which You Want to Search: ";
   cin>>k;
   index=obj.find(k);
   if(index==-1)
     {
    cout<<"\n\tKey Not Found";
     }
   break;

 case 4: cout<<"\n\tEnter a Key Which You Want to Delete: ";
   cin>>k;
   obj.Delete(k);
   break;

 case 5:
   break;
  }

cout<<"\n\t..... Do You Want to Continue in Main Menu:y/n ";
cin>>ans;
 }while(ans=='y'||ans=='Y');
 return 0;

      }
```

Assignment 3

A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method.

```
#include<iostream>

#include <iostream>

#include <string.h>

using namespace std;

struct node // Node Declaration

{

   string label;

   //char label[10];

   int ch_count;

   struct node *child[10];

} * root;

class GT // Class Declaration

{

public:

   void create_tree();

   void display(node *r1);

   GT()

   {

      root = NULL;

   }
```

```
};
void GT::create_tree()
{
    int tbooks, tchapters, i, j, k;
    root = new node;
    cout << "Enter name of book : ";
    cin.get();
    getline(cin, root->label);
    cout << "Enter number of chapters in book : ";
    cin >> tchapters;
    root->ch_count = tchapters;
    for (i = 0; i < tchapters; i++)
    {
        root->child[i] = new node;
        cout << "Enter the name of Chapter " << i + 1 << " : ";
        cin.get();
        getline(cin, root->child[i]->label);
        cout << "Enter number of sections in Chapter : " << root->child[i]->label << " : ";
        cin >> root->child[i]->ch_count;
        for (j = 0; j < root->child[i]->ch_count; j++)
        {
            root->child[i]->child[j] = new node;
            cout << "Enter Name of Section " << j + 1 << " : ";
            cin.get();
            getline(cin, root->child[i]->child[j]->label);
```

```
            //cout<<"Enter no. of subsections in "<<r1->child[i]->child[j]->label;
            //cin>>r1->child[i]->ch_count;
        }
    }
}
void GT::display(node *r1)
{
    int i, j, k, tchapters;
    if (r1 != NULL)
    {
        cout << "\n-----Book Hierarchy---";
        cout << "\n Book title : " << r1->label;
        tchapters = r1->ch_count;
        for (i = 0; i < tchapters; i++)
        {
            cout << "\nChapter " << i + 1;
            cout << " : " << r1->child[i]->label;
            cout << "\nSections : ";
            for (j = 0; j < r1->child[i]->ch_count; j++)
            {
                cout << "\n"<< r1->child[i]->child[j]->label;
            }
        }
    }
    cout << endl;
```

```
}
int main()
{
    int choice;
    GT gt;
    while (1)
    {
        cout << "-----------------" << endl;
        cout << "Book Tree Creation" << endl;
        cout << "-----------------" << endl;
        cout << "1.Create" << endl;
        cout << "2.Display" << endl;
        cout << "3.Quit" << endl;
        cout << "Enter your choice : ";
        cin >> choice;
        switch (choice)
        {
        case 1:
            gt.create_tree();
        case 2:
            gt.display(root);
            break;
        case 3:

          exit(1);
```

```
        default:
            cout << "Wrong choice!!!" << endl;
        }
    }
    return 0;
}
```

Assignment 4

Construct an expression tree from the given prefix expression
eg. +--a*bc/def and traverse it using post order traversal (non recursive) and thendelete the entire tree.

```
#include<iostream>

#include <iostream>

#include <string.h>

using namespace std;


struct node

{

   char data;

   node *left;

   node *right;

};

class tree

{

   char prefix[20];


public:

   node *top;

   void expression(char[]);

   void display(node *);

   void non_rec_postorder(node *);

   void del(node *);
```

```
};
class stack1
{
    node *data[30];
    int top;

public:
    stack1()
    {
        top = -1;
    }
    int empty()
    {
        if (top == -1)
            return 1;
        return 0;
    }
    void push(node *p)
    {
        data[++top] = p;
    }
    node *pop()
    {
        return (data[top--]);
    }
```

```
};
void tree::expression(char prefix[])
{
    char c;
    stack1 s;
    node *t1, *t2;
    int len, i;
    len = strlen(prefix);
    for (i = len - 1; i >= 0; i--)
    {
        top = new node;
        top->left = NULL;
        top->right = NULL;
        if (isalpha(prefix[i]))
        {
            top->data = prefix[i];
            s.push(top);
        }
        else if (prefix[i] == '+' || prefix[i] == '*' || prefix[i] == '-' || prefix[i] == '/')
        {
            t2 = s.pop();
            t1 = s.pop();
            top->data = prefix[i];
            top->left = t2;
            top->right = t1;
```

```
            s.push(top);
        }
    }
    top = s.pop();
}
void tree::display(node *root)
{
    if (root != NULL)
    {
        cout << root->data;
        display(root->left);
        display(root->right);
    }
}
void tree::non_rec_postorder(node *top)
{
    stack1 s1, s2; /*stack s1 is being used for flag . A NULL data implies that the right subtree has not been visited */
    node *T = top;
    cout << "\n";
    s1.push(T);
    while (!s1.empty())
    {
        T = s1.pop();
        s2.push(T);
```

```
        if (T->left != NULL)
            s1.push(T->left);
        if (T->right != NULL)
            s1.push(T->right);
    }
    while (!s2.empty())
    {
        top = s2.pop();
        cout << top->data;
    }
}
void tree::del(node *node)
{
    if (node == NULL)
        return;
    /* first delete both subtrees */
    del(node->left);
    del(node->right);
    /* then delete the node */
    cout <<endl<<"Deleting node : " << node->data<<endl;
    free(node);
}
int main()
{
    char expr[20];
```

```
    tree t;


    cout <<"Enter prefix Expression : ";
    cin >> expr;
    cout << expr;
    t.expression(expr);
    //t.display(t.top);
    //cout<<endl;
    t.non_rec_postorder(t.top);
    t.del(t.top);
    // t.display(t.top);
}
```

Assignment 5

Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree - i. Insert new node ii. Find number of nodes in longest path from root iii. Minimum data value found in the tree iv. Change a tree so that the roles of the left and right pointers are swapped at every node v. Search a value.

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

class node {
    public:
        int data;
        node *left;
        node *right;
};

class Bst {
public:
        node *root;
    Bst () {
        root = NULL;
    }
    void create ();
```

```
    void insert ();
    void postorder (node*);
    void inorder (node*);
    void preorder (node*);
    void search (int key);
    void minimum ();
    int height (node*);
    void mirror (node*);
};

void Bst::create () {
    int ans;
    cout << "\nEnter number of keys to insert: ";
    cin >> ans;
    cout << '\n';
    while (ans--)
        insert();
}

void Bst::inorder (node *root) {
    if (root != NULL) {
        inorder (root -> left);
        cout << " -> " << root -> data;
        inorder (root -> right);
    }
```

```
}

void Bst::preorder (node *root) {
    if (root != NULL) {
        cout << " -> " << root -> data;
        preorder (root -> left);
        preorder (root -> right);
    }
}

void Bst::postorder (node *root) {
    if (root != NULL) {
        postorder (root -> left);
        postorder (root -> right);
        cout << " -> " << root -> data;
    }
}

void Bst::insert () {
    node *curr,*temp;
    cout << "Enter data: ";
    curr = new node;
    cin >> curr -> data;
    curr -> left = curr -> right = NULL;
```

```
    if (root == NULL)
        root = curr;

    else {
        temp = root;
        while (1) {
            if (curr -> data <= temp -> data) {
                if (temp -> left == NULL) {
                    temp -> left = curr;
                    break;
                }
                else
                    temp = temp -> left;
            }
            else {
                if (temp -> right == NULL) {
                    temp -> right = curr;
                    break;
                }
                else
                    temp = temp -> right;
            }
        }
    }
}
```

```
void Bst::search (int key) {
    node *curr;
    curr = root;

    while (curr != NULL) {
        if (curr -> data == key) {
            cout << key << " found";
            break;
        }
        else {
            if (key<curr -> data)
                curr = curr -> left;
            else
                curr = curr -> right;
        }
    }
    if (curr == NULL)
        cout << key << " not found";
}

void Bst::minimum () {
    node *temp = root;
    int min;
    while (temp -> left != NULL) {
```

```
        min = temp -> data;
        temp = temp -> left;
        if (temp -> data<min)
            min = temp -> data;
        else
            temp = temp -> left;
    }
    cout << "\nMinimum number is: " << min;
}

int Bst::height (node *root) {
    if (root == NULL)
        return 0;

    else {
        if (height (root -> right) > height (root -> left))
            return (1 + height (root -> right));

        else
            return (1 + height (root -> left));
    }
}

void Bst::mirror (node *root) {
    if (root == NULL)
```

```
        return;


    else {
        mirror(root -> left);
        mirror(root -> right);
        swap(root -> left, root -> right);
    }
}


int main () {
    Bst b;
    int key,ch;
    do {
        cout << "\n\n1.Create 2.Insert 3.Inorder 4.Preorder 5.Postorder 6.Search 7.Minimum 8.Height 9.Mirror\n";
        cout << "Your choice [1/2/3/4/5/6/7/8/9] ";
        cin >> ch;
        switch (ch) {
            case 1:
                b.create ();
                break;
            case 2:
                cout << '\n';
                b.insert ();
                break;
```

```
case 3:
    cout << "\nInorder   traversal is:";
    b.inorder (b.root);
    break;
case 4:
    cout << "\nPreorder  traversal is:";
    b.preorder (b.root);
    break;
case 5:
    cout << "\nPostorder traversal is:";
    b.postorder (b.root);
    break;
case 6:
    cout << "\nEnter search key: ";
    cin >> key;
    b.search (key);
    break;
case 7:
    b.minimum ();
    break;
case 8:
    cout << "\nHeight of tree: " << b.height (b.root);
    break;
case 9:
    b.mirror (b.root);
```

```
            cout << "\nTree is now mirrored!!!"
                 << "\nInorder   traversal is:";
            b.inorder (b.root);
            cout << "\nPreorder  traversal is:";
            b.preorder (b.root);
            cout << "\nPostorder traversal is:";
            b.postorder (b.root);
            break;
        }
    }while (ch < 10);
    return 0;
}
```

Assignment 6

Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent land marks as nodes and perform DFS and BFS on that.

```
#include <iostream>

#include <vector>

#include <list>

using namespace std;


class Graph {

    // Number of vertex

    int v;

    // Pointer to an array containing adjacency lists

    list<int> *adjL;


    // Adjacency matrix

    int **adjM;

    // Visited vector to so that a vertex is not visited more than once

    vector<bool> visited;

public:

    // Constructor create the initial /list

    Graph(int);

    // Function to insert a new edge

    void addEdge(int, int);
```

```
    // Function to display the DFS traversal on adjacency matrix
    void DFS(int);
    // Function to display the BFS traversal on adjacency list
    void BFS(int);
};



// Function to fill the empty adjacency matrix & initialize adjacency list
Graph::Graph(int v) {
    this->v = v;
    // Adjacency lists
    adjL = new list<int>[v];

    // Adjacency matrix
    adjM = new int*[v];
    // A visited array of initially false for all vertices
    visited.assign(v, false);
    for (int row = 0; row < v; row++) {
        adjM[row] = new int[v];
        for (int column = 0; column < v; column++) {
            adjM[row][column] = 0;
        }
    }
}
```

```
// Function to add an edge to the graph
void Graph::addEdge(int x, int y) {
    // Add y to x's adjacency list.
    adjL[x].push_back(y);
    // Add x to y's adjacency list.
    adjL[y].push_back(x);

    // Considering a bidirectional edge to adjacency matrix
    adjM[x][y] = 1;
    adjM[y][x] = 1;
}

// Function to perform DFS on the graph
void Graph::DFS(int start) {
    // Print the first node
    cout << start << " ";
    // Set current node as visited
    visited[start] = true;

    // For every node of the graph
    for (int i = 0; i < v; i++) {
        // If some node is adjacent to the current node
        // and it has not already been visited
        if (adjM[start][i] == 1 && (!visited[i])) {
```

```
            DFS(i);
        }
    }
}

void Graph::BFS(int start) {
    // A visited array of initially false for all vertices
    visited.assign(v, false);

    // Create a queue for BFS
    list<int> queue;
    // Mark the current node as visited and enqueue it
    visited[start] = true;
    queue.push_back(start);

    while(!queue.empty()) {
        // Dequeue a vertex from queue and print it
        start = queue.front();
        cout << start << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex start. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (int i : adjL[start]) {
```

```
            if (!visited[i]) {
                visited[i] = true;
                queue.push_back(i);
            }
        }
    }
}



int main() {
    int v = 8;

    Graph G(v);

    // Graph edges
    int edges[][2] = {{0, 1}, {0, 6}, {0, 5}, {1, 2}, {1, 6}, {2, 3}, {2, 4}, {2, 6}, {2, 7}, {3, 4}, {3, 7}, {4, 5}, {4, 6}, {4, 7}, {5, 6}};

    for (auto edge : edges) {
        G.addEdge(edge[0], edge[1]);
    }

    cout << "\nOperation on following Graph ->\n\n\n"
    "              (1)  @@  @  @  @  @ @  @ @@@  (2)\n"
    "                  @@@              @  @ @   @\n"
```

```
    "            @ @          @    @  @    @\n"
    "          @   @        @     @   @      @\n"
    "        @     @      @       @     @       @\n"
    "  (0)   @       @    @         @       @         @\n"
    "    @         @  @           @         @          @\n"
    "  @@@ @  @  @  @  @ @@  (6)          @        (7) @@           @@ (3)\n"
    "  @@@          @@             @          @@@  @  @  @  @  @@@\n"
    "    @         @  @           @         @          @\n"
    "      @       @    @         @       @         @\n"
    "        @     @      @       @       @       @\n"
    "          @   @        @     @     @      @\n"
    "            @  @         @    @   @    @\n"
    "              @@@           @  @  @  @\n"
    "          (5)  @@  @  @  @  @  @  @@@@  (4)\n";

    // Perform DFS
    cout << "\n\n  Depth First Traversal (starting from vertex 2) : ";
    G.DFS(2);

    // Perform BFS
    cout << "\n\nBreadth First Traversal (starting from vertex 2) : ";
    G.BFS(2);
    return 0;
}
```

Assignment 7

There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph.
Check whether the graph is connected or not. Justify the storage representation used.

```
#include <iostream>

#include <limits.h>

using namespace std;


class Office {

    int n;

    int adjacent[10][10];

    string office[10];


public:

    void input ();

    void display ();

    void Prims ();

};


void Office::input () {

    cout << "\nEnter no. of offices: ";

    cin >> n;

    cout << "\nEnter the names of offices: ";
```

```
    for (int i = 0 ; i < n ; i++)
        cin >> office[i];

    cout << "\nEnter the cost to connect the offices: \n";
    for (int i = 0 ; i < n ; i++)
        for (int j = i ; j < n ; j++) {
            if (i == j) {
                adjacent[i][j] = 0;
                continue;
            }

            cout << "Enter the cost to connect " << office[i] <<" and " << office[j]<< " : ";
            cin >> adjacent[i][j];
            adjacent[j][i] = adjacent[i][j];
        }
}

void Office::display () {

    for (int i = 0 ; i < n ; i++) {
        cout << "\n";
        for (int j = 0 ; j < n ; j++) {
            cout << adjacent[i][j] << "\t";
        }
    }
```

```
}

void Office::Prims () {
    int visit[n], minCost = 0, count = n - 1, minIndex, cost = 0;
    for (int i = 0 ; i < n ; i++)
        visit[i] = 0;

    cout << "\n\nShortest path: ";
    visit[0]=1;
    cout << office[0] << " -> ";

    while (count--) {
        minCost = INT_MAX;
        for (int i = 0 ; i < n ; i++) {
            for (int j = 0 ; j < n ; j++) {
                if (visit[i] == 1 && adjacent[i][j] != 0 && adjacent[i][j] < minCost && visit[j] == 0) {
                    minCost = adjacent[i][j];
                    minIndex = j;
                }
            }
        }
        visit[minIndex]=1;
        cout << office[minIndex] << " -> ";
        cost = cost + minCost;
    }
```

```
   cout << "End";
    cout << "\nMinimum cost: "<<cost;
}

int main () {
    Office o1;
    int choice;
    do {
        cout << "\n\nMINIMUM SPANNING TREE"
            << "\n1. Input data"
            << "\n2. Display data"
            << "\n3. Calculate minimum cost"
            << "\nEnter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                o1.input ();
                break;
            case 2:
                o1.display ();
                break;
            case 3:
                o1.Prims ();
                break;
        }
```

```
    } while (choice != 4);

    return 0;

}
```

Assignment 8

Given sequence k = k1 <k2 < … < kn of n sorted keys, with a search probability pi for each key ki. Build the Binary search tree that has the least search cost given the access probability for each key?

```
#include <iostream>

#include <limits.h>

using namespace std;

#define SIZE 15


class OBST {

   int prob[SIZE] = {};          //Probabilities with which we search for an element

   int keys[SIZE] = {};          //Elements from which OBST is to be built

   int weight[SIZE][SIZE] = {};   //Weight weight[i][j]' of keys tree having root 'root[i][j]'

   int cost[SIZE][SIZE] = {};     //Cost 'cost[i][j] of keys tree having root 'root[i][j]

   int root[SIZE][SIZE] = {};     //represents root

   int n;                        // number of nodes

public:

   void get_data();

   int Min_Value(int, int);

   void build_OBST();

   void build_tree();

   void print(int [][SIZE], int);

};


/* This function accepts the input data */
```

```
void OBST::get_data() {
    int i;
    cout << "\nOptimal Binary Search Tree \n\nEnter the number of nodes: ";
    cin >> n;
    cout << "\nEnter " << n << " nodes: ";
    for (i = 1; i <= n; i++)
        cin >> keys[i];

    cout << "\nEnter " << n << " probabilities: ";
    for (i = 1; i <= n; i++)
        cin >> prob[i];

}

/* This function returns keys value in the range ‘r[i][j-1]’ to ‘r[i+1][j]’so
that the cost ‘cost[i][k-1]+cost[k][j]’is minimum */
int OBST::Min_Value(int i, int j) {
    int l, k;
    int minimum = INT_MAX;
    for (l = root[i][j - 1]; l <= root[i + 1][j]; l++) {
        if ((cost[i][l - 1] + cost[l][j]) < minimum) {
            minimum = cost[i][l - 1] + cost[l][j];
            k = l;
        }
    }
```

```
    return k;
}

/* This function builds the table from all the given probabilities It
basically computes cost,root,weight values */
void OBST::build_OBST() {
    int i, j, k, l;
    for (i = 0; i < n; i++) {
        //initialize
        weight[i][i] = root[i][i] = cost[i][i] = 0;
        //Optimal trees with one node
        weight[i][i + 1] = cost[i][i + 1] = prob[i + 1];
        root[i][i + 1] = i + 1;
    }
    weight[n][n] = root[n][n] = cost[n][n] = 0;
    //Find optimal trees with 'm' nodes
    for (l = 2; l <= n; l++) {
        for (i = 0; i <= n - l; i++) {
            j = i + l;
            weight[i][j] = weight[i][j - 1] + prob[j];
            k = Min_Value(i, j);
            cost[i][j] = weight[i][j] + cost[i][k - 1] + cost[k][j];
            root[i][j] = k;
        }
    }
```

```
    cout << "\nCost are: \n";
    print(cost, n);

    cout << "\nRoot are: \n";
    print(root, n);
}

/* This function builds the tree from the tables made by the OBST function */
void OBST::build_tree() {
    int i, j, k;
    int queue[20], front = -1, rear = -1;
    cout << "\nThe Optimal Binary Search Tree For the Given Nodes Is…\n";
    cout << "\nThe Root of this OBST is:: " << keys[root[0][n]];
    cout << "\nThe Cost of this OBST is:: " << cost[0][n];
    cout << "\n\n\tNODE\tLEFT CHILD\tRIGHT CHILD";
    cout << "\n";
    queue[++rear] = 0;
    queue[++rear] = n;
    while (front != rear) {
        i = queue[++front];
        j = queue[++front];
        k = root[i][j];
        cout << "\n\t" << keys[k];
        if (root[i][k - 1] != 0) {
```

```
            cout << "\t\t" << keys[root[i][k - 1]];
            queue[++rear] = i;
            queue[++rear] = k - 1;
        }
        else
            cout << "\t\t";
        if (root[k][j] != 0) {
            cout << "\t" << keys[root[k][j]];
            queue[++rear] = k;
            queue[++rear] = j;
        }
        else
            cout << "\t";
    }
    cout << "\n";
}

void OBST::print(int arr[][SIZE], int n) {
    int i, j;
    for(i = 0; i <= n; i++) {
        for(j = 0; j <= n; j++)
            cout << arr[i][j] << '\t';
        cout << '\n';
    }
}
```

```
int main() {
    OBST obj;
    obj.get_data();
    obj.build_OBST();
    obj.build_tree();
    return 0;
}
```

Assignment 9

A Dictionary stores keywords and its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword.

```
#include <iostream>

using namespace std;


struct AVLnode {

   public:

   int cWord;

   string cMean;

   AVLnode *left,*right;

   int iHt;

};


class AVLtree {


public:

   AVLnode *Root;

   AVLtree () {

      Root = NULL;

   }

   AVLnode* insert (AVLnode*, int, string);
```

```
    AVLnode* deletE (AVLnode*, int);
    AVLnode* LL (AVLnode*);
    AVLnode* RR (AVLnode*);
    AVLnode* LR (AVLnode*);
    AVLnode* RL (AVLnode*);
    int height (AVLnode*);
    int bFactor (AVLnode*);
    void inOrder (AVLnode*);
    void preOrder (AVLnode*);
};


AVLnode* AVLtree::insert (AVLnode *root, int nWord, string nMean) {
    if (root == NULL) {
        root = new AVLnode;
        root -> left = root -> right = NULL;
        root -> cWord = nWord;
        root -> cMean = nMean;
        root -> iHt = 0;
    }

    else if (root -> cWord != nWord) {
        if (root -> cWord > nWord)
            root -> left = insert (root -> left, nWord, nMean);
```

```
        else
            root -> right = insert (root -> right, nWord, nMean);
    }

    else
        cout << "\nRedundant AVLnode\n";

    root -> iHt = max(height(root -> left), height(root -> right)) + 1;

    if (bFactor (root) == 2) {
        if (root -> left -> cWord > nWord)
            root = RR (root);
        else
            root = LR (root);
    }

    if (bFactor (root) == -2) {
        if (root -> right -> cWord > nWord)
            root = RL (root);
        else
            root = LL (root);
    }

    return root;
}
```

```
AVLnode *AVLtree::deletE (AVLnode *curr, int x) {
    AVLnode *temp;
    if (curr == NULL) {
        cout << "\nWord not present!\n";
        return curr;
    }

    else if (x > curr -> cWord)
        curr -> right = deletE (curr -> right, x);

    else if (x < curr -> cWord)
        curr -> left = deletE (curr -> left, x);

    else if (curr -> right == NULL || curr -> left == NULL) {
        curr = curr -> left ? curr -> left : curr -> right;
        cout << "\nWord deleted Successfully!\n";
    }

    else {
        temp = curr -> right;
        while (temp -> left)
            temp = temp -> left;
        curr -> cWord = temp -> cWord;
```

```
        curr -> right = deletE (curr -> right, temp -> cWord);
    }

    if (curr == NULL) return curr;

    curr -> iHt = max(height(curr -> left), height(curr -> right)) + 1;

    if (bFactor (curr) == 2) {
        if (bFactor (curr -> left) >= 0)
            curr = RR (curr);
        else
            curr = LR (curr);
    }

    if (bFactor (curr) == -2) {
        if (bFactor (curr -> right) <= 0)
            curr = LL (curr);
        else
            curr = RL (curr);
    }

    return (curr);
}
```

```
int AVLtree::height (AVLnode* curr) {
    if (curr == NULL)
        return -1;
    else
        return curr -> iHt;
}


int AVLtree::bFactor (AVLnode* curr) {
    int lh = 0, rh = 0;
    if (curr == NULL)
        return 0;
    else
        return height(curr -> left) - height(curr -> right);
}


AVLnode* AVLtree::RR (AVLnode* curr) {
    AVLnode* temp = curr -> left;
    curr -> left = temp -> right;
    temp -> right = curr;
    curr -> iHt = max(height(curr -> left), height(curr -> right)) + 1;
    temp -> iHt = max(height(temp -> left), height(temp -> right)) + 1;
    return temp;
}
```

```
AVLnode* AVLtree::LL (AVLnode* curr) {
    AVLnode* temp = curr -> right;
    curr -> right = temp -> left;
    temp -> left = curr;
    curr -> iHt = max(height(curr -> left), height(curr -> right)) + 1;
    temp -> iHt = max(height(temp -> left), height(temp -> right)) + 1;
    return temp;
}


AVLnode* AVLtree::RL (AVLnode* curr) {
    curr -> right = RR (curr -> right);
    return LL (curr);
}


AVLnode* AVLtree::LR (AVLnode* curr) {
    curr -> left = LL (curr -> left);
    return RR (curr);
}


void AVLtree::inOrder (AVLnode* curr) {
```

```
    if (curr != NULL) {
        inOrder (curr -> left);
        cout << "\n\t" << curr -> cWord << "\t" << curr -> cMean;
        inOrder (curr -> right);
    }
}
void AVLtree::preOrder (AVLnode* curr) {
    if (curr != NULL) {
        cout << "\n\t" << curr -> cWord << "\t" << curr -> cMean;
        preOrder (curr -> left);
        preOrder (curr -> right);
    }
}



int main () {
    int ch;
    AVLtree avl;
    AVLnode *temp = NULL;
    int word;
    string mean;

    cout << "\n---------------------------------------";
    cout << "\n\tAVL TREE IMPLEMENTATION";
    cout << "\n---------------------------------------";
```

```
do {
    cout << "\n\t\tMENU";
    cout << "\n1.Insert 2.Inorder 3.Delete 4.Exit";
    cout << "\n--------------------------------";
    cout << "\nEnter your choice: ";
    cin >> ch;

    switch (ch) {
        case 1:
            cout << "\nEnter Word: ";
            cin >> word;
            cout << "\nEnter Meaning: ";
            cin >> mean;
            avl.Root = avl.insert (avl.Root, word, mean);
            break;
        case 2:
            cout << "\nInorder Traversal:\n\tWORD\tMEANING";
            avl.inOrder (avl.Root);
            cout << "\n\nPreorder Traversal:\n\tWORD\tMEANING";
            avl.preOrder (avl.Root);
            cout << '\n';
            break;
        case 3:
            cout << "\nEnter the word to be deleted : ";
            cin >> word;
```

```
            avl.Root = avl.deletE (avl.Root, word);
            break;
        case 4:
            exit (0);
      }
   } while (ch != 4);
return 0;
}
```

Assignment 10

Implement the Heap/Shell sort algorithm implemented in Java demonstrating heap/shell datastructure with modularity of programming language

```
import java.util.*;

public class vivek_a12 {


 private static int N;

   public static void sort(int arr[]){

  heapMethod(arr);

      for (int i = N; i > 0; i--){

         swap(arr,0, i);

         N = N-1;

         heap(arr, 0);

      }

   }

   public static void heapMethod(int arr[]){

      N = arr.length-1;

      for (int i = N/2; i >= 0; i--)

         heap(arr, i);

   }

   public static void heap(int arr[], int i){

      int left = 2*i ;

      int right = 2*i + 1;

      int max = i;

      if (left <= N && arr[left] > arr[i])
```

```
        max = left;
    if (right <= N && arr[right] > arr[max])
        max = right;
      if (max != i){
        swap(arr, i, max);
        heap(arr, max);
      }
  }
  public static void swap(int arr[], int i, int j){
      int tmp = arr[i];
      arr[i] = arr[j];
      arr[j] = tmp;
  }
  public static void main(String[] args) {
      Scanner in = new Scanner( System.in );
      int n;
      System.out.println("Enter the number of elements to be sorted:");
      n = in.nextInt();
      int arr[] = new int[ n ];
      System.out.println("Enter "+ n +" integer elements");
      for (int i = 0; i < n; i++)
        arr[i] = in.nextInt();
      sort(arr);
      System.out.println("After sorting ");
      for (int i = 0; i < n; i++)
```

```
            System.out.println(arr[i]+" ");
        System.out.println();
    }


}
```

Assignment 11

Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular employee. If record of student does not exist an appropriate message is displayed. If it is, then the system displays the student details.

Use sequential file to main the data.

```
#include <iostream>

#include <fstream>

using namespace std;


class Record {

    int rollno;

    string name;

    int division;

    string address;


public:

    Record();

    int getRollno();

    void getData();

    void putData();

};


Record::Record() {

    rollno = 0;

    name = ' ';
```

```
    address = ' ';
    division = 0;
}

int Record::getRollno() {
    return (rollno);
}

void Record::getData() {
    cout << "\nEnter Details: ";
    cout << "\nRoll no: ";
    cin >> rollno;
    cout << "Name: ";
    cin >> name;
    cout << "Division Code: ";
    cin >> division;
    cout << "Address: ";
    cin >> address;
}

void Record::putData() {
    cout << "\nRoll No.: ";
    cout << rollno;
    cout << "\t\tName: ";
    cout << name;
```

```
    cout << "\nDivision Code: ";
    cout << division;
    cout << "\tAddress: ";
    cout << address;
}

class File {
    ifstream fin;
    ofstream fout;
    fstream fs;

public:
    void insert();
    void display();
    void search(int);
    int Delete(int);
    int edit(int);
};

void File::insert() {
    Record r;
    r.getData();
    fout.open("StudentDB", ios::ate | ios::app);
    fout.write((char *)&r, sizeof(r));
    fout.close();
```

```
}

void File::display() {
    Record r;
    fin.open("StudentDB");
    fin.seekg(0, ios::beg);
    while (fin.read((char *)&r, sizeof(r)))
        r.putData();

    fin.close();
}

void File::search(int rollno) {
    Record r;
    int flag = 0;
    fin.open("StudentDB");
    fin.seekg(0, ios::beg);
    while (fin.read((char *)&r, sizeof(r))) {
        if (r.getRollno() == rollno) {
            flag = 1;
            break;
        }
    }

    fin.close();
```

```
    if (flag == 1) {
        cout << "\nRecord Found:";
        r.putData();
    }
    else
        cout << "\nRecord not Found ";
}

int File::Delete(int rollno) {
    Record r;
    int flag = 0;
    fin.open("StudentDB");
    fout.open("Temp", ios::ate | ios::app);
    fin.seekg(0, ios::beg);
    while (fin.read((char *)&r, sizeof(r))) {
        if (r.getRollno() == rollno) {
            flag = 1;
        }
        else {
            fout.write((char *)&r, sizeof(r));
        }
    }
    fin.close();
    fout.close();
    remove("StudentDB");
```

```
    rename("Temp", "StudentDB");
    return (flag);
}

int File::edit(int rollno) {
    Record r;
    int flag = 0;
    fs.open("StudentDB");
    fs.seekg(0, ios::beg);
    while (fs.read((char *)&r, sizeof(r))) {
        if (r.getRollno() == rollno) {
            flag = 1;
            cout << "\nEnter New Details: ";
            r.getData();
            fs.seekp((int)fs.tellg() - sizeof(r), ios::beg);
            fs.write((char *)&r, sizeof(r));
        }
    }
    fs.close();
    return (flag);
}

int main() {
    File f;
    int ch, n, i, flag = 0;
```

```
do {
    cout << "\n\n\t-----M E N U-----";
    cout << "\n\n1. Build A Master Table";
    cout << "\n2. List A Table";
    cout << "\n3. Insert a New Entry";
    cout << "\n4. Delete Old Entry";
    cout << "\n5. Edit an Entry";
    cout << "\n6. Search for a Record";
    cout << "\n7. Quit";
    cout << "\nEnter your Choice: ";
    cin >> ch;
    switch (ch) {
       case 1:
          if (flag == 0) {
             cout << "\nEnter No of Records to insert : ";
             cin >> n;
             for (i = 0; i < n; i++) {
                f.insert();
             }
             flag = 1;
          }
          else {
             cout << "\nSorry.. Table is Already build... \n If want to add record please select Insert a New Entry in option.....";
          }
```

```
    break;
case 2:
    f.display();
    break;
case 3:
    f.insert();
    break;
case 4:
    cout << "\nEnter Roll No of Student Whose Record is to be Deleted: ";
    cin >> n;
    i = f.Delete(n);
    if (i == 1)
        cout << "\nRecord Deleted Successfully";
    else
        cout << "\nRecord not Found";
    break;
case 5:
    cout << "\nEnter Roll No of Student Whose Record is to be Edit: ";
    cin >> n;
    i = f.edit(n);
    if (i == 1)
        cout << "\nRecord Modified Successfully";
    else
        cout << "\nRecord not Found";
    break;
```

```
            case 6:
                cout << "\nEnter Roll No of Student Whose Record is to be Searched: ";
                cin >> n;
                f.search(n);
                break;
            case 7:
                break;
            default:
                cout << "\nEnter Valid Choice.....";
        }
    } while (ch != 7);
    return (0);
}
```

Assignment 12

Company maintains employee information as employee ID, name, designation and salary. Allow user to add, delete information of employee. Display information of particular employee. If employee does not exist an appropriate message is displayed. If it is, then the system displays the employee details. Use index sequential file to maintain the data.

```
#include <iostream>

#include <fstream>

using namespace std;


class Record {

    int id;

    string name;

    int salary;

    string designation;


public:

    Record();

    int getIdno();

    void getData();

    void putData();

};


Record::Record() {

    id = 0;

    name = ' ';
```

```
    designation = ' ';
    salary = 0;
}

int Record::getIdno() {
    return (id);
}

void Record::getData() {
    cout << "\nEnter Details: ";
    cout << "\nId no: ";
    cin >> id;
    cout << "Name: ";
    cin >> name;
    cout << "Salary: ";
    cin >> salary;
    cout << "Designation: ";
    cin >> designation;
}

void Record::putData() {
    cout << "\nId No.: ";
    cout << id;
    cout << "\t\tName: ";
    cout << name;
```

```
    cout << "\nSalary: ";
    cout << salary;
    cout << "\tDesignation: ";
    cout << designation;
}

class File {
    ifstream fin;
    ofstream fout;
    fstream fs;

public:
    void insert();
    void display();
    void search(int);
    int Delete(int);
    int edit(int);
};

void File::insert() {
    Record r;
    r.getData();
    fout.open("EmployeeDB", ios::ate | ios::app);
    fout.write((char *)&r, sizeof(r));
    fout.close();
```

```
}

void File::display() {
    Record r;
    fin.open("EmployeeDB");
    fin.seekg(0, ios::beg);
    while (fin.read((char *)&r, sizeof(r)))
        r.putData();

    fin.close();
}

void File::search(int id) {
    Record r;
    int flag = 0;
    fin.open("EmployeeDB");
    fin.seekg(0, ios::beg);
    while (fin.read((char *)&r, sizeof(r))) {
        if (r.getIdno() == id) {
            flag = 1;
            break;
        }
    }

    fin.close();
```

```
    if (flag == 1) {
        cout << "\nRecord Found:";
        r.putData();
    }
    else
        cout << "\nRecord not Found ";
}

int File::Delete(int id) {
    Record r;
    int flag = 0;
    fin.open("EmployeeDB");
    fout.open("Temp", ios::ate | ios::app);
    fin.seekg(0, ios::beg);
    while (fin.read((char *)&r, sizeof(r))) {
        if (r.getIdno() == id) {
            flag = 1;
        }
        else {
            fout.write((char *)&r, sizeof(r));
        }
    }
    fin.close();
    fout.close();
    remove("EmployeeDB");
```

```
    rename("Temp", "EmployeeDB");
    return (flag);
}

int File::edit(int id) {
    Record r;
    int flag = 0;
    fs.open("EmployeeDB");
    fs.seekg(0, ios::beg);
    while (fs.read((char *)&r, sizeof(r))) {
        if (r.getIdno() == id) {
            flag = 1;
            cout << "\nEnter New Details: ";
            r.getData();
            fs.seekp((int)fs.tellg() - sizeof(r), ios::beg);
            fs.write((char *)&r, sizeof(r));
        }
    }
    fs.close();
    return (flag);
}

int main() {
    File f;
    int ch, n, i, flag = 0;
```

```
do {
    cout << "\n\n\t-----M E N U-----";
    cout << "\n\n1. Build A Master Table";
    cout << "\n2. List A Table";
    cout << "\n3. Insert a New Entry";
    cout << "\n4. Delete Old Entry";
    cout << "\n5. Edit an Entry";
    cout << "\n6. Search for a Record";
    cout << "\n7. Quit";
    cout << "\nEnter your Choice: ";
    cin >> ch;
    switch (ch) {
       case 1:
          if (flag == 0) {
             cout << "\nEnter No of Records to insert : ";
             cin >> n;
             for (i = 0; i < n; i++) {
                f.insert();
             }
             flag = 1;
          }
          else {
             cout << "\nSorry.. Table is Already build... \n If want to add record please select Insert a New Entry in option.....";
          }
```

```
    break;
case 2:
    f.display();
    break;
case 3:
    f.insert();
    break;
case 4:
    cout << "\nEnter Id No of Employee Whose Record is to be Deleted: ";
    cin >> n;
    i = f.Delete(n);
    if (i == 1)
        cout << "\nRecord Deleted Successfully";
    else
        cout << "\nRecord not Found";
    break;
case 5:
    cout << "\nEnter Id No of Employee Whose Record is to be Edit: ";
    cin >> n;
    i = f.edit(n);
    if (i == 1)
        cout << "\nRecord Modified Successfully";
    else
        cout << "\nRecord not Found";
    break;
```

```
            case 6:
                cout << "\nEnter Id No of Employee Whose Record is to be Searched: ";
                cin >> n;
                f.search(n);
                break;
            case 7:
                break;
            default:
                cout << "\nEnter Valid Choice.....";
        }
    } while (ch != 7);
    return (0);
}
```