Bridge course - Assignment 3

A.Jyothi

M22cs007

1. Write a program to implement:

a. Prim’s algorithm

b. Kruskal's algorithm.

a. prims

using namespace std;

#include<iostream>

#define V 5

void printST(int root[], int graf[V][V])

{

    cout<< "Edge\t Weight\n";

    for(int i = 1; i< V; i++)

    {

        cout << root[i] << " - "<< i<<" \t"<< graf[i][root[i]]<<"\n";

    }

}

//Utility function to find the minimum key which is not included in the mstSet

int minimumKey(int key[], bool mstSet[]){

    int min = INT\_MAX;

    int minIndex;

    int i=0;

    while(i< V)

    {

        if(mstSet[i]==false && key[i]< min){

            min = key[i];

            minIndex = i;

        }

        i++;

    }

    return minIndex;

}

void primsTree(int graph[V][V])

{

    //used to store constructed MST.

    //Stores the parent of each vertex. mnodes [0] has 0 vertex's parent(adjacent)

    int root[V];

    //Key values to pick from the minimum edges keys

    int key[V];

    //verticies included in the spanning tree.

    bool mstSet[V];

    for(int i =0; i< V; i++)

    {

        key[i]= INT\_MAX;

        mstSet[i] = false;

    }

    //Consider the 1st index and make it as parent.

    key[0] = 0;

    root[0] = -1;

    //Minumum Spanning tree contains V vertices. Already added 1.

    int j=0;

    while(j<V-1)

    {

        int pickedVertex = minimumKey(key,mstSet);

        mstSet[pickedVertex] = true;

        //Since we picked a vertex from the other vertices, we need to update the key for

        //all of those.

        int i=0;

        while(i<V)

        {

            if(graph[pickedVertex][i] && mstSet[i] == false && graph[pickedVertex][i]< key[i])

            {

                root[i] = pickedVertex;

                key[i] = graph[pickedVertex][i];

            }

            i++;

        }

        j++;

    }

    //parent array is filled is filled with the adjacents.

    printST(root,graph);

}

int main()

{

    int graf[V][V] =

    {

        {0,2,0,6,0},

        {2,0,3,8,5},

        {0,3,0,0,7},

        {6,8,0,0,9},

        {0,5,7,9,0}

    };

    primsTree(graf);

}

b. krushkals

using namespace std;

#include<vector>

#include<iostream>

#include<algorithm>

class Mz

{

    int\* mnodes;

    int\* count;

    public:

        Mz(int n)

        {

           mnodes = new int[n];

            count = new int[n];

            int i=0;

            while(i<n)

            {

                mnodes[i] = -1;

                count[i] = 1;

                i++;

            }

        }

        int search(int i)

        {

            if(mnodes[i] == -1)

            {

                return i;

            }

            return search(mnodes[i]);

        }

        void unionsets(int x, int y)

        {

            int set1 = search(x);

            int set2 = search(y);

            if(set1!=set2)

            {

                if(count[set1]<count[set2])

                {

                    mnodes[set1] = set2;

                    count[set2]+=count[set1];

                }

                else

                {

                    mnodes[set2] = set1;

                    count[set1]+=count[set2];

                }

            }

        }

};

class Graphs

{

    vector<vector<int>> edgeslist;

    int V;

    public:

    Graphs(int V)

    {

        this->V = V;

    }

    void Edgead(int x, int y, int z)

    {

        edgeslist.push\_back({z,x,y});

    }

    void mst()

    {

        //Sort the edges in the ascending order.

        std::sort(edgeslist.begin(), edgeslist.end());

        Mz d(V);

        int weight = 0;

        cout<<"The following are the edges in the Minimal Spanning Tree using Kruskal's algorithm:"<<endl;

        for(auto edge: edgeslist)

        {

            int z = edge[0];

            int x = edge[1];

            int y = edge[2];

            if(d.search(x)!= d.search(y))

            {

                d.unionsets(x,y);

                weight+=z;

                cout<< x << " ---- " << y << " == " <<z << endl;

            }

        }

    }

};

int main()

{

    Graphs g(4);

    g.Edgead(0,1,20);

    g.Edgead(1,3,25);

    g.Edgead(2,0,15);

    g.Edgead(0,3,10);

    g.Edgead(2,3,8);

    g.mst();

}

2. Write a program to find the weight of the minimum spanning tree given the graph.

//Minimum weight can be calculated by using the Kruskal's minimum spanning tree.

using namespace std;

#include<vector>

#include<iostream>

#include<algorithm>

class Mz{

    int\* mnodes;

       int\* count;

    public:

        Mz(int n){

            mnodes = new int[n];

            count = new int[n];

            int i=0;

            while(i<n)

           {

                mnodes[i] = -1;

               count[i] = 1;

               i++;

            }

        }

        int search(int i){

            if(mnodes[i] == -1){

                return i;

            }

            return search(mnodes[i]);

        }

        void unionsets(int x, int y){

            int set1 = search(x);

            int set2 = search(y);

            if(set1!=set2){

                if(count[set1]<count[set2]){

                    mnodes[set1] = set2;

                    count[set2]+=count[set1];

                }

                else{

                    mnodes[set2] = set1;

                    count[set1]+=count[set2];

                }

            }

        }

};

class Graphs{

    vector<vector<int>> edgeslist;

    int V;

    public:

    Graphs(int V){

        this->V = V;

    }

    void Edgead(int x, int y, int z){

        edgeslist.push\_back({z,x,y});

    }

    int mstWeight(){

        //Sort the edges in the ascending order.

        std::sort(edgeslist.begin(), edgeslist.end());

        Mz d(V);

        int weight = 0;

        cout<<"The following are the edges in the Minimal Spanning Tree using Kruskal's algorithm:"<<endl;

        for(auto edge: edgeslist){

            int z = edge[0];

            int x = edge[1];

            int y = edge[2];

            if(d.search(x)!= d.search(y)){

                d.unionsets(x,y);

                weight+=z;

                cout<< x << " ---- " << y << " == " <<z << endl;

            }

        }

        return weight;

    }

};

int main(){

    Graphs g(4);

    g.Edgead(0,1,20);

    g.Edgead(1,3,25);

    g.Edgead(2,0,15);

    g.Edgead(0,3,10);

    g.Edgead(2,3,8);

    int weight = g.mstWeight();

    cout<<"Weight of MST is :"<< weight;

}

5. Implement the following in AVL Tree a. Insertion b. Deletion c. Search

//AVL Trees.

#include<iostream>

using namespace std;

class Avl

{

    public:

    int data;

    Avl\* lt;

    Avl\* rt;

    int ht;

};

int ht(Avl\* n)

{

    if(n == nullptr){

        return 0;

    }

    return n->ht;

}

int max(int a1, int a2)

{

    return (a1 > a2)? a1 : a2;

}

Avl\* createNode(int data)

{

    Avl\* node = new Avl();

    node->data = data;

    node->lt = nullptr;

    node->rt = nullptr;

    node->ht = 1;

    return node;

}

Avl\* shiftRt(Avl\* y)

{

    Avl\* x = y->lt;

    Avl\* T2 = x->rt;

    x->rt = y;

    y->lt = T2;

    y->ht = max(ht(y->lt),ht(y->rt)) + 1;

    x->ht = max(ht(x->lt),ht(x->rt)) + 1;

    return x;

}

Avl\* shiftLt(Avl \*x)

{

    Avl \*y = x->rt;

    Avl \*T2 = y->lt;

    y->lt = x;

    x->rt = T2;

    x->ht = max(ht(x->lt),ht(x->rt)) + 1;

    y->ht = max(ht(y->lt),ht(y->rt)) + 1;

    return y;

}

int getBalanceFactor(Avl\* N){

    if(N== nullptr)

    {

        return 0;

    }

    return ht(N->lt)-ht(N->rt);

}

Avl\* insert(Avl\* node, int data)

{

    if (node == nullptr)

        return(createNode(data));

    if (data < node->data)

        node->lt = insert(node->lt, data);

    else if (data > node->data)

        node->rt = insert(node->rt, data);

    else // keys should not be same in binary search tree..

        return node;

    //we need to update the height to predecessor or i general nodes, after we add a new node.

    node->ht = max(ht(node->lt),ht(node->rt))+1;

    //finding out the balance factor for this node.

    int balanceFactor = getBalanceFactor(node);

    //there will be four cases depending on the balance factor.

    //Left Left Case, case 1

    if(balanceFactor > 1 && data < node->lt->data)

    {

        return shiftRt(node);

    }

    //RightRight Case

    if(balanceFactor < -1 && data > node->rt->data)

    {

        return shiftLt(node);

    }

    //Left->Right Case

    if(balanceFactor > 1 && data > node->lt->data)

    {

        node->lt = shiftLt(node->lt);

        return shiftRt(node);

    }

//Right->Left case

    if(balanceFactor < -1 && data < node->rt->data)

    {

        node->rt = shiftRt(node->rt);

        return shiftLt(node);

    }

    return node;

}

void preOrder(Avl \*root)

{

    if(root != NULL)

    {

        cout << root->data << " ";

        preOrder(root->lt);

        preOrder(root->rt);

    }

}

Avl \* getInorderSuccesor(Avl\* node)

{

    Avl\* current = node;

    /\* loop down to find the leftmost leaf \*/

    while (current->lt != NULL)

        current = current->lt;

    return current;

}

Avl\* deleteNode(Avl\* root, int key)

{

    //deletion

    if (root == NULL)

        return root;

    if ( key < root->data )

        root->lt = deleteNode(root->lt, key);

    else if( key > root->data )

        root->rt = deleteNode(root->rt, key);

    // if key is same as root's key, then that node should be deleted

    else

    {

        // node with only one child or no child

        if( (root->lt == NULL) || (root->rt == NULL) )

        {

            Avl \*temp = root->lt ? root->lt : root->rt;

            // No child case

            if (temp == NULL)

            {

                temp = root;

                root = NULL;

            }

            else // One child case

            \*root = \*temp; // Copy the contents of child.

            free(temp);

        }

        else

        {

            //inorder succesor of the root.. and delete

            Avl\* temp = getInorderSuccesor(root->rt);

            //Replace the root data with inordersuccessor data./

            root->data = temp->data;

            //Now delete the inorder successor.

            root->rt = deleteNode(root->rt,temp->data);

        }

    }

    // If the tree is having only one node, then delete

    if (root == NULL)

    return root;

        // Since the call is recursive, update the height

    root->ht = max(ht(root->lt),ht(root->rt)) + 1;

    // Get the balance factor for the current node

    int balance = getBalanceFactor(root);

    // LeftLeft Case..case 1

    if (balance > 1 && getBalanceFactor(root->lt) >= 0)

        return shiftRt(root);

    // LeftRight Case...case 2

    if (balance > 1 && getBalanceFactor(root->lt) < 0)

    {

        root->lt = shiftLt(root->lt);

        return shiftRt(root);

    }

    // RightRight Case..3

    if (balance < -1 && getBalanceFactor(root->rt) <= 0)

        return shiftLt(root);

    // Right Left Case..4

    if (balance < -1 && getBalanceFactor(root->rt) > 0)

    {

        root->rt = shiftRt(root->rt);

        return shiftLt(root);

    }

    return root;

}

int main()

{

    Avl \*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);

    cout << "Preorder traversal of the constructed AVL tree is:\n";

    preOrder(root);

    root = deleteNode(root, 10);

    cout << "\nPreorder traversal after"

         << " deletion of 10 \n";

    preOrder(root);

}

5