Course: ENSF694 – Summer 2025

Lab #: Lab 5

Instructor: Mahmood Moussavi

Student Name: John Zhou

Submission Date: August 7th, 2025  
  
  
  
  
  
  
I have been keeping all the files in github. I hope by providing this github link will help you a little bit.  
https://github.com/JZ-Zhou-UofC/ENSF-604-assignment-repo

Exercise A

Code:

//

//  AVL tree.cpp

//  AVL tree

// ENSF 694 - Lab 5 Exercise A

//  Created by Mahmood Moussavi on 2024-05-22.

// Completed by: John Zhou

#include "AVL\_tree.h"

using namespace std;

AVLTree::AVLTree() : root(nullptr), cursor(nullptr) {}

int AVLTree::height(const Node \*N)

{

    if (N == nullptr)

        return 0;

    return N->height;

}

int AVLTree::getBalance(Node \*N)

{

    if (N == nullptr)

        return 0;

    int leftHeight = (N->left != nullptr) ? N->left->height : 0;

    int rightHeight = (N->right != nullptr) ? N->right->height : 0;

    return leftHeight - rightHeight;

}

Node \*AVLTree::rightRotate(Node \*y)

{

    Node \*x = y->left;

    Node \*T2 = x->right;

    x->right = y;

    y->left = T2;

    x->parent = y->parent;

    y->parent = x;

    if (T2 != nullptr)

        T2->parent = y;

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

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

    return x;

}

Node \*AVLTree::leftRotate(Node \*x)

{

    Node \*y = x->right;

    Node \*T2 = y->left;

    y->left = x;

    x->right = T2;

    y->parent = x->parent;

    x->parent = y;

    if (T2 != nullptr)

        T2->parent = x;

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

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

    return y;

}

void AVLTree::insert(int key, Type value)

{

    root = insert(root, key, value, nullptr);

}

// Recursive function

Node \*AVLTree::insert(Node \*node, int key, Type value, Node \*parent)

{

    if (node == nullptr)

        return new Node(key, value, parent);

    if (key < node->data.key)

    {

        node->left = insert(node->left, key, value, node);

    }

    else if (key > node->data.key)

    {

        node->right = insert(node->right, key, value, node);

    }

    else

    {

        return node;

    }

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

    int balance = getBalance(node);

    if (balance > 1 && key < node->left->data.key)

    {

        return rightRotate(node);

    }

    if (balance > 1 && key > node->left->data.key)

    {

        node->left = leftRotate(node->left);

        return rightRotate(node);

    }

    if (balance < -1 && key > node->right->data.key)

    {

        return leftRotate(node);

    }

    if (balance < -1 && key < node->right->data.key)

    {

        node->right = rightRotate(node->right);

        return leftRotate(node);

    }

    return node;

}

// Recursive function

void AVLTree::inorder(const Node \*root)

{

    if (root == nullptr)

        return;

    inorder(root->left);

    cout << root->data.key << " ";

    inorder(root->right);

}

// Recursive function

void AVLTree::preorder(const Node \*root)

{

    if (root == nullptr)

        return;

    cout << root->data.key << " ";

    preorder(root->left);

    preorder(root->right);

}

// Recursive function

void AVLTree::postorder(const Node \*root)

{

    if (root == nullptr)

        return;

    postorder(root->left);

    postorder(root->right);

    cout << root->data.key << " ";

}

const Node \*AVLTree::getRoot()

{

    return root;

}

void AVLTree::find(int key)

{

    go\_to\_root();

    if (root != nullptr)

        find(root, key);

    else

        cout << "It seems that tree is empty, and key not found." << endl;

}

// Recursive funtion

void AVLTree::find(Node \*root, int key)

{

    if (root == nullptr)

    {

        return;

    }

    if (key < root->data.key)

    {

        find(root->left, key);

    }

    else if (key > root->data.key)

    {

        find(root->right, key);

    }

    else

    {

        cursor = root;

    }

}

AVLTree::AVLTree(const AVLTree &other) : root(nullptr), cursor(nullptr)

{

    root = copy(other.root, nullptr);

    cursor = root;

}

AVLTree::~AVLTree()

{

    destroy(root);

}

AVLTree &AVLTree::operator=(const AVLTree &other)

{

    if (this == &other)

        return \*this;

    destroy(root);

    root = copy(other.root, nullptr);

    cursor = root;

    return \*this;

}

// Recursive funtion

Node \*AVLTree::copy(Node \*node, Node \*parent)

{

    if (node == nullptr)

    {

        return nullptr;

    }

    Node \*newNode = new Node(node->data.key, node->data.value, parent);

    newNode->left = copy(node->left, newNode);

    newNode->right = copy(node->right, newNode);

    newNode->height = node->height;

    return newNode;

}

// Recusive function

void AVLTree::destroy(Node \*node)

{

    if (node)

    {

        destroy(node->left);

        destroy(node->right);

        delete node;

    }

}

const int &AVLTree::cursor\_key() const

{

    if (cursor != nullptr)

        return cursor->data.key;

    else

    {

        cout << "looks like tree is empty, as cursor == Zero.\n";

        exit(1);

    }

}

const Type &AVLTree::cursor\_datum() const

{

    if (cursor != nullptr)

        return cursor->data.value;

    else

    {

        cout << "looks like tree is empty, as cursor == Zero.\n";

        exit(1);

    }

}

int AVLTree::cursor\_ok() const

{

    if (cursor == nullptr)

        return 0;

    return 1;

}

void AVLTree::go\_to\_root()

{

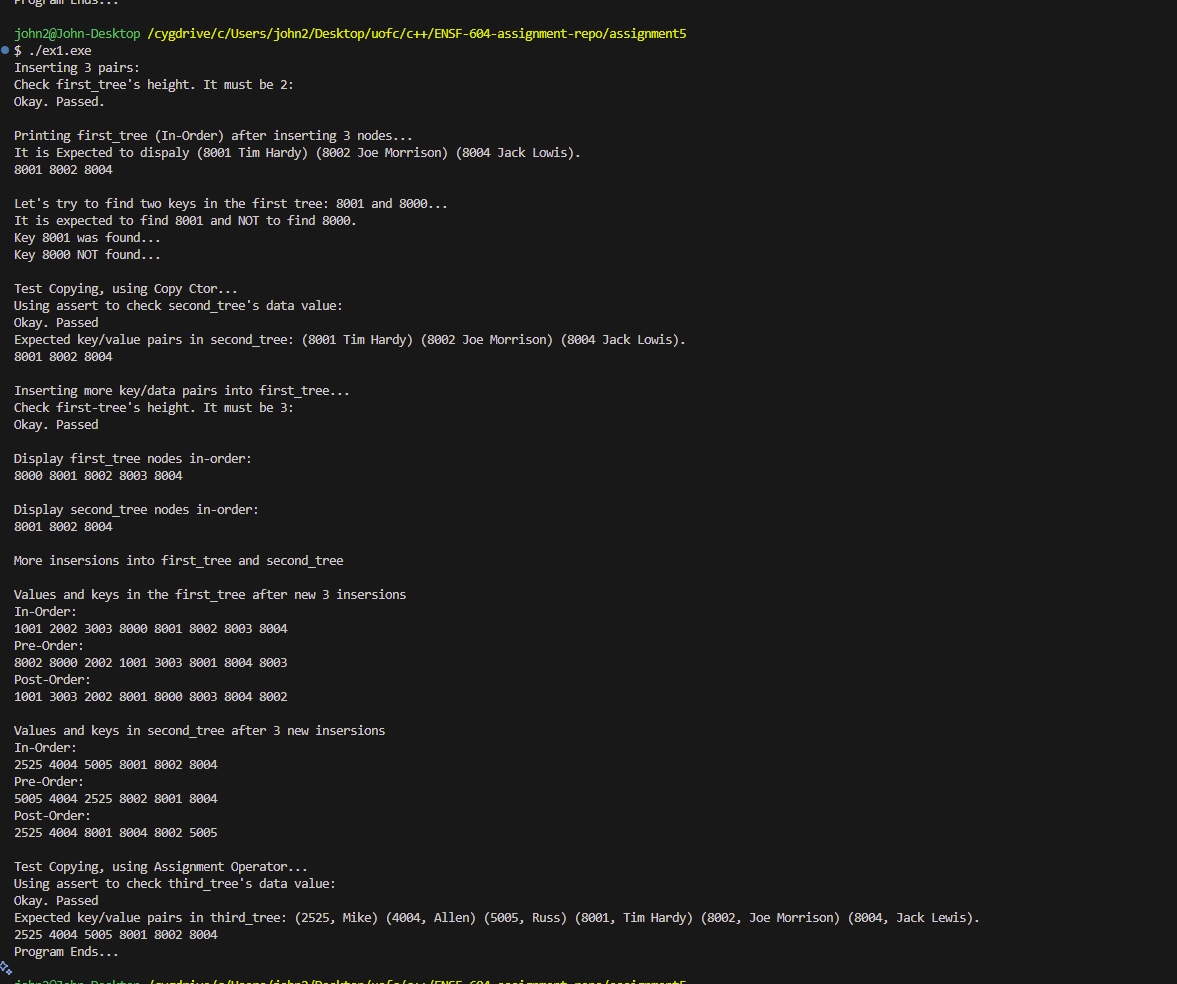
    if (!root)

        cursor = root;

    // cursor = nullptr;

}

Program Output



Exercise B

//

//  graph.cpp

//  graph

// ENSF 694 - Lab 5 Exercise B

//  Created by Mahmood Moussavi

// Completed by: John Zhou

#include "graph.h"

PriorityQueue::PriorityQueue() : front(nullptr) {}

bool PriorityQueue::isEmpty() const

{

    return front == nullptr;

}

void PriorityQueue::enqueue(Vertex \*v)

{

    ListNode \*newNode = new ListNode(v);

    if (isEmpty() || v->dist < front->element->dist)

    {

        newNode->next = front;

        front = newNode;

    }

    else

    {

        ListNode \*current = front;

        while (current->next != nullptr && current->next->element->dist <= v->dist)

        {

            current = current->next;

        }

        newNode->next = current->next;

        current->next = newNode;

    }

}

Vertex \*PriorityQueue::dequeue()

{

    if (isEmpty())

    {

        cerr << "PriorityQueue is empty." << endl;

        exit(0);

    }

    Vertex \*frontItem = front->element;

    ListNode \*old = front;

    front = front->next;

    delete old;

    return frontItem;

}

void Graph::printGraph()

{

    Vertex \*v = head;

    while (v)

    {

        for (Edge \*e = v->adj; e; e = e->next)

        {

            Vertex \*w = e->des;

            cout << v->name << " -> " << w->name << "  " << e->cost << "   " << (w->dist == INFINITY ? "inf" : to\_string(w->dist)) << endl;

        }

        v = v->next;

    }

}

Vertex \*Graph::getVertex(const char vname)

{

    Vertex \*ptr = head;

    Vertex \*newv;

    if (ptr == nullptr)

    {

        newv = new Vertex(vname);

        head = newv;

        tail = newv;

        numVertices++;

        return newv;

    }

    while (ptr)

    {

        if (ptr->name == vname)

            return ptr;

        ptr = ptr->next;

    }

    newv = new Vertex(vname);

    tail->next = newv;

    tail = newv;

    numVertices++;

    return newv;

}

void Graph::addEdge(const char sn, const char dn, double c)

{

    Vertex \*v = getVertex(sn);

    Vertex \*w = getVertex(dn);

    Edge \*newEdge = new Edge(w, c);

    newEdge->next = v->adj;

    v->adj = newEdge;

    (v->numEdges)++;

    // point 1

}

void Graph::clearAll()

{

    Vertex \*ptr = head;

    while (ptr)

    {

        ptr->reset();

        ptr = ptr->next;

    }

}

void Graph::dijkstra(const char start)

{

    // STUDENTS MUST COMPLETE THE DEFINITION OF THIS FUNCTION

    clearAll();

    Vertex \*startingVertex = getVertex(start);

    startingVertex->dist = 0;

    PriorityQueue que;

    que.enqueue(startingVertex);

    while (!que.isEmpty())

    {

        Vertex \*currentVertex = que.dequeue();

        for (Edge \*e = currentVertex->adj; e != nullptr; e = e->next)

        {

            Vertex \*neighbourVertex = e->des;

            double  newDist = currentVertex->dist + e->cost;

            if (newDist < neighbourVertex->dist)

            {

                neighbourVertex->dist = newDist;

                neighbourVertex->prev = currentVertex;

                que.enqueue(neighbourVertex);

            }

        }

    }

}

void Graph::unweighted(const char start)

{

    // STUDENTS MUST COMPLETE THE DEFINITION OF THIS FUNCTION

        clearAll();

    Vertex \*startingVertex = getVertex(start);

    startingVertex->dist = 0;

    PriorityQueue que;

    que.enqueue(startingVertex);

    while (!que.isEmpty())

    {

        Vertex \*currentVertex = que.dequeue();

        for (Edge \*e = currentVertex->adj; e != nullptr; e = e->next)

        {

            Vertex \*neighbourVertex = e->des;

            double  newDist = currentVertex->dist + 1.0;

            if (newDist < neighbourVertex->dist)

            {

                neighbourVertex->dist = newDist;

                neighbourVertex->prev = currentVertex;

                que.enqueue(neighbourVertex);

            }

        }

    }

}

void Graph::readFromFile(const string &filename)

{

    ifstream infile(filename);

    if (!infile)

    {

        cerr << "Could not open file: " << filename << endl;

        exit(1);

    }

    char sn, dn;

    double cost;

    while (infile >> sn >> dn >> cost)

    {

        addEdge(sn, dn, cost);

    }

    infile.close();

}

void Graph::printPath(Vertex \*dest)

{

    if (dest->prev != nullptr)

    {

        printPath(dest->prev);

        cout << " " << dest->name;

    }

    else

    {

        cout << dest->name;

    }

}

void Graph::printAllShortestPaths(const char start, bool weighted)

{

    if (weighted)

    {

        dijkstra(start);

    }

    else

    {

        unweighted(start);

    }

    setiosflags(ios::fixed);

    setprecision(2);

    Vertex \*v = head;

    while (v)

    {

        if (v->name == start)

        {

            cout << start << " -> " << v->name << "     0   " << start << endl;

        }

        else

        {

            cout << start << " -> " << v->name << "     " << (v->dist == INFINITY ? "inf" : to\_string((int)v->dist)) << "   ";

            if (v->dist == INFINITY)

            {

                cout << "No path" << endl;

            }

            else

            {

                printPath(v);

                cout << endl;

            }

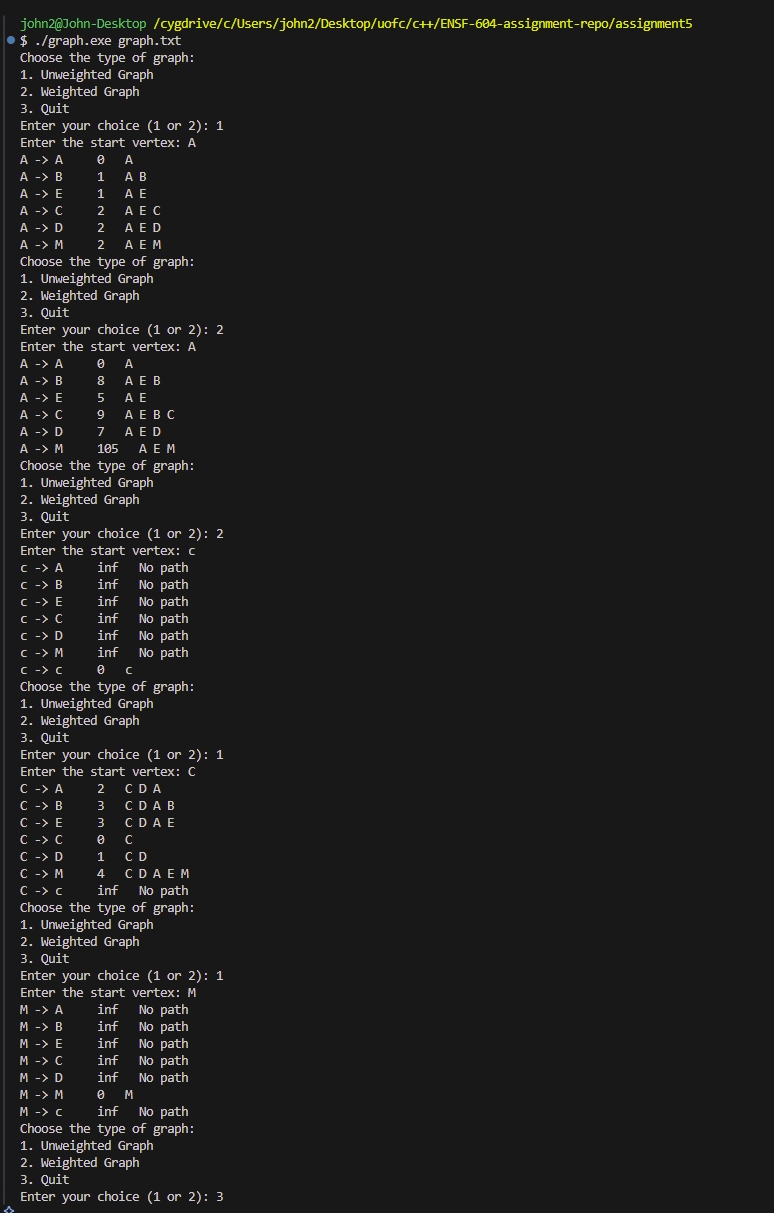
        }

        v = v->next;

    }

}

Program output 1



Program output 2

