**Course: ENSF 694** – Summer 2025

**Lab #:** 05

**Instructor:** Dr. Mahmood Moussavi

**Student Name:** Jack Shenfield

**Submission Date:** Wednesday, August 6th, 2025

**Exercise A**

Source Code:

/\*

\* AVL\_tree.cpp

\* ENSF 694 Lab 5, exercise A

\* Created by Mahmood Moussavi on 2024-05-22

\* Completed by: Jack Shenfield

\* Development Date: August 5th, 2025

\*/

#include "AVL\_tree.h"

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

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

// Student must complete and if necessary change the return value of

// this function this function

// return 0 if there are no children

// return height if there are children

return (N == nullptr) ? 0 : N->height;

}

int AVLTree::getBalance(Node\* N) {

// Student must complete and if necessary change the return value of

// this function this function

if (N == nullptr) {

return 0;

}

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

}

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

// Student must complete and if necessary change the return value of

// this function this function

// y is the unbalanced node. must pivot around y->right

// y is the parent node

// x is the pivot node

// T2 is tree 2, the right subtree of x

// these 3 nodes must be moved.

// extract nodes

Node\* x = y->left;

Node\* T2 = x->right;

// "rotate" parent node around

x->right = y;

y->left = T2;

// if T2 exists, it's new parent is y.

if (T2){

T2->parent = y;

}

// x is the new parent

x->parent = y->parent;

// adjust y to be x's child

y->parent = x;

// re-calculate heights

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

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

return x;

}

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

// Student must complete and if necessary change the return value of

// this function this function

// see comments above. same logic just for left rotate.

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

if (T2){

T2->parent = x;

}

y->parent = x->parent;

x->parent = y;

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

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

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) {

// Student must complete and if necessary change the return value of

// this function this function

// base case

if (node == nullptr) // insert where the current node points to nullptr

return new Node(key, value, parent);

if (key < node->data.key) // root node is less than current node key, recursively call

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

else if (key > node->data.key) // if it is more than current node key, recursively call back

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

else // key = node->data.key this is a duplicate, and we do not insert

return node;

node->height = 1 + std::max(height(node->left), height(node->right)); // update height at current node

int balance = getBalance(node); // calculate balance where node is being inserted

// Rotation may be required

// LL

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

return rightRotate(node);

}

// RR

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

return leftRotate(node);

}

// LR

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

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

return rightRotate(node);

}

// RL

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) {

// Student must complete this function

if (!root){ // IF DNE, return

return;

}

// recursive call/print order for inorder

inorder(root->left);

std::cout << "(" << root->data.key << " " << root->data.value << ") ";

inorder(root->right);

}

// Recursive function

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

// Student must complete this function

if (!root){ // if DNE, return

return;

}

// recursive call/print order for preorder

std::cout << "(" << root->data.key << " " << root->data.value << ") ";

preorder(root->left);

preorder(root->right);

}

// Recursive function

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

// Student must complete this function

// base case, if not root

if (!root){

return;

}

// recursive call/print order for postorder

postorder(root->left);

postorder(root->right);

std::cout << "(" << root->data.key << " " << root->data.value << ") ";

}

const Node\* AVLTree::getRoot(){

return root;

}

void AVLTree::find(int key) {

go\_to\_root();

if(root != nullptr)

find(root, key);

else

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

}

// Recursive funtion

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

// Student must complete this function

if (!root) { // If root DNE, print root not found

cursor = nullptr;

std::cout << "Key " << key << " NOT found...\n";

return;

}

if (key == root->data.key) { // if found, print key and value

cursor = root;

std::cout << "Key " << key << " found with value: " << root->data.value << "\n";

}

else if (key < root->data.key){ // if the key is less than current node, recursively call left (to lesser values)

find(root->left, key);

}

else{ // if the key is greater than current node, recursively call right (to greater values).

find(root->right, key);

}

}

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) {

// Student must complete and if necessary change the return value of this function this function

if (node == nullptr) return nullptr; // if node DNE, return nullptr

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

newNode->left = copy(node->left, newNode); // copy left sub-tree

newNode->right = copy(node->right, newNode); // recurisvely call to copy right sub-tree

newNode->height = node->height; // calculate new heights

return newNode;

}

// Recusive function

void AVLTree::destroy(Node\* node) {

if (node) {

destroy(node->left);

destroy(node->right);

delete node;

}

// Student must complete this function

}

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

if (cursor != nullptr)

return cursor->data.key;

else{

std::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{

std::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:

A screenshot of a computer program

AI-generated content may be incorrect.

**Exercise B**

Source Code:

#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

// chatgpt assisted a couple of lines of code to get me started.

clearAll(); // reset data

Vertex\* s = getVertex(start); // point s to start vertex

if (!s){ // return if DNE

return;

}

s->dist = 0; // set distance to zero

PriorityQueue pq; // create queue

pq.enqueue(s); // enqueue current vertex

while (!pq.isEmpty()) { // while there are vertices to visit, continue the following code

Vertex\* v = pq.dequeue();

if (v->scratch) continue; // already been visited

v->scratch = 1;

for (Edge\* e = v->adj; e != nullptr; e = e->next) { // iterate through all neighbours of v

Vertex\* w = e->des;

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

if (w->dist > newDist) {

w->dist = newDist;

w->prev = v;

pq.enqueue(w);

}

}

}

}

void Graph::unweighted(const char start) {

// STUDENTS MUST COMPLETE THE DEFINITION OF THIS FUNCTION

// a lot of logic copied from Dijkstra's solution.

clearAll(); // Reset all data

Vertex\* s = getVertex(start); // point s to start vertex

if (!s){ // if s DNE, return

return;

}

s->dist = 0; // set distance to 0

queue<Vertex\*> q; // new queue

q.push(s); // add s to queue

while (!q.empty()) { // check all edges connected to current vertex

Vertex\* v = q.front(); q.pop();

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

Vertex\* w = e->des;

if (w->dist == INFINITY) {

w->dist = v->dist + 1;

w->prev = v;

q.push(w);

}

}

}

}

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:  
  
(base) jbs@Jacks-MacBook-Air ENSF694\_LabAssignment5 % ./graphsolution graph.txt

Choose the type of graph:

1. Unweighted Graph

2. Weighted Graph

3. Quit

Enter your choice (1 or 2): 1

Enter the start vertex: A

A -> A 0 A

A -> B 1 A B

A -> E 1 A E

A -> C 2 A E C

A -> D 2 A E D

A -> M 2 A E M

Choose the type of graph:

1. Unweighted Graph

2. Weighted Graph

3. Quit

Enter your choice (1 or 2): 2

Enter the start vertex: C

C -> A 11 C D A

C -> B 19 C D A E B

C -> E 16 C D A E

C -> C 0 C

C -> D 4 C D

C -> M 116 C D A E M

Choose the type of graph:

1. Unweighted Graph

2. Weighted Graph

3. Quit

Enter your choice (1 or 2): 2

Enter the start vertex: A

A -> A 0 A

A -> B inf No path

A -> E inf No path

A -> C inf No path

A -> D inf No path

A -> M inf No path

Choose the type of graph:

1. Unweighted Graph

2. Weighted Graph

3. Quit

Enter your choice (1 or 2): 1

Enter the start vertex: C

C -> A 2 C D A

C -> B 3 C D A B

C -> E 3 C D A E

C -> C 0 C

C -> D 1 C D

C -> M 4 C D A E M

Choose the type of graph:

1. Unweighted Graph

2. Weighted Graph

3. Quit

Enter your choice (1 or 2): 1

Enter the start vertex: M

M -> A inf No path

M -> B inf No path

M -> E inf No path

M -> C inf No path

M -> D inf No path

M -> M 0 M

Choose the type of graph:

1. Unweighted Graph

2. Weighted Graph

3. Quit

Enter your choice (1 or 2): 3

(base) jbs@Jacks-MacBook-Air ENSF694\_LabAssignment5 %