ENSF 694 – Summer 2024

Principles of Software Development II

University of Calgary

Lab Assignment 5

Student Name: Yael Gonzalez

Instructor: M. Moussavi, PhD, Peng

Submission Date: August 2, 2024

Exercise A

Source code

HashTable.cpp

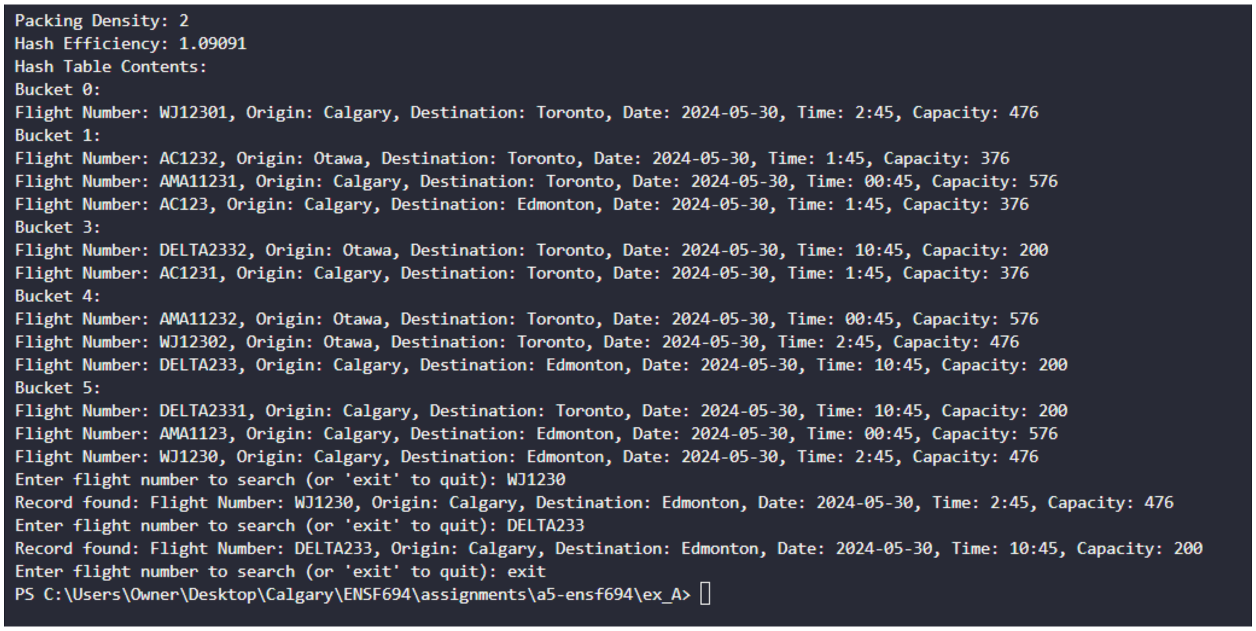
```cpp
/**
 *  File Name: HashTable.cpp
 *  Assignment: ENSF 694 Summer 2024 - Lab 5 Exercise A
 *  Created by: Mahmood Moussavi
 *  Completed by: Yael Gonzalez
 *  Submission Date: August 2, 2024
 */
#include "HashTable.h"
#include <cmath>
#include <iostream>

// Implemented using Knuth's Multiplicative Method
unsigned int HashTable::hashFunction(const string &flightNumber) const
{
    const double A = (sqrt(5) - 1) / 2; // Knuth's constant
    unsigned int sum = 0;

    for (auto &ch : flightNumber)
    {
        sum += (unsigned int)ch;
    }
    double fractionalPart = sum * A - floor(sum * A); // Same as (sum * A) % 1
    return floor(tableSize * fractionalPart);
}

HashTable::HashTable(unsigned int size) : tableSize(size), numberOfRecords(0)
{
    table.resize(size);
}

void HashTable::insert(const Flight &flight)
{
    table.at(hashFunction(flight.flightNumber)).insert(flight);
    numberOfRecords++;
}

bool HashTable::insertFirstPass(const Flight &flight)
{
    unsigned int index = hashFunction(flight.flightNumber);
    if (table.at(index).isEmpty())
    {
        table.at(index).insert(flight);
        numberOfRecords++;
        return true;
    }
    return false;
}

void HashTable::insertSecondPass(const Flight &flight)
{
    insert(flight);
}

Flight *HashTable::search(const string &flightNumber) const
{
    return table.at(hashFunction(flightNumber)).search(flightNumber);
}

double HashTable::calculatePackingDensity() const
{
    return (double)numberOfRecords / tableSize;
}

double HashTable::calculateHashEfficiency() const
{
    int numberOfReads = 0;
    for (const auto &list : table)
    {
        const Node *curr = list.get_head();
        for (int i = 1; curr != nullptr; i++)
        {
            numberOfReads += i;
            curr = curr->next;
        }
    }
    double averageReads = (double)numberOfReads / numberOfRecords;
    return calculatePackingDensity() / averageReads;
}

void HashTable::display() const
{
    for (unsigned int i = 0; i < tableSize; i++)
    {
        if (!table.at(i).isEmpty())
        {
            cout << "Bucket " << i << ":" << endl;
            table.at(i).display();
        }
    }
}
```

Global function read_flight_info

Note: This function was already completely implemented when files were received.

```
void read_flight_info(int argc, char **argv, vector<Flight> &records)
{
    // open the stream to read the text file
    if (argc != 2)
    {
        cerr << "Usage: hashtable input.txt" << endl;
        exit(1);
    }
    string fileName = "./"; // Change path to your input file
    fileName += string(argv[1]);
    ifstream inputFile;
    inputFile.open(fileName.c_str());

    if (!inputFile)
    {
        cerr << "Error opening file: " << argv[1] << endl;
        exit(1);
    }

    string line;
    while (getline(inputFile, line))
    {
        stringstream ss(line);
        string flightNumber, origin, destination, departureDate, departureTime;
        int craftCapacity;

        ss >> flightNumber >> origin >> destination >> departureDate >>
departureTime >> craftCapacity;

        Flight record(flightNumber, Point(origin), Point(destination),
departureDate, departureTime, craftCapacity);
        records.push_back(record);
    }
    inputFile.close();
}
```

Program output



```
Packing Density: 2
Hash Efficiency: 1.09091
Hash Table Contents:
Bucket 0:
Flight Number: WJ12301, Origin: Calgary, Destination: Toronto, Date: 2024-05-30, Time: 2:45, Capacity: 476
Bucket 1:
Flight Number: AC1232, Origin: Otawa, Destination: Toronto, Date: 2024-05-30, Time: 1:45, Capacity: 376
Flight Number: AMA11231, Origin: Calgary, Destination: Toronto, Date: 2024-05-30, Time: 00:45, Capacity: 576
Flight Number: AC123, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 1:45, Capacity: 376
Bucket 3:
Flight Number: DELTA2332, Origin: Otawa, Destination: Toronto, Date: 2024-05-30, Time: 10:45, Capacity: 200
Flight Number: AC1231, Origin: Calgary, Destination: Toronto, Date: 2024-05-30, Time: 1:45, Capacity: 376
Bucket 4:
Flight Number: AMA11232, Origin: Otawa, Destination: Toronto, Date: 2024-05-30, Time: 00:45, Capacity: 576
Flight Number: WJ12302, Origin: Otawa, Destination: Toronto, Date: 2024-05-30, Time: 2:45, Capacity: 476
Flight Number: DELTA233, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 10:45, Capacity: 200
Bucket 5:
Flight Number: DELTA2331, Origin: Calgary, Destination: Toronto, Date: 2024-05-30, Time: 10:45, Capacity: 200
Flight Number: AMA1123, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 00:45, Capacity: 576
Flight Number: WJ1230, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 2:45, Capacity: 476
Enter flight number to search (or 'exit' to quit): WJ1230
Record found: Flight Number: WJ1230, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 2:45, Capacity: 476
Enter flight number to search (or 'exit' to quit): DELTA233
Record found: Flight Number: DELTA233, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 10:45, Capacity: 200
Enter flight number to search (or 'exit' to quit): exit
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a5-ensf694\ex_A>
```

Packing density and hash efficiency

Packing density as defined in the course:

$$Packing\ Density = \frac{Number\ of\ Records}{Number\ of\ Spaces}$$

In our code, the total space available is defined by the table size in the input, so:

$$calculatePackagingDensity = \frac{numberOfRecords}{tableSize}$$

Where table size is computed as half the number of records in the input. Considering the provided input file input.txt:

$$calculatePackagingDensity = \frac{18}{9} = 2$$

Hashing Efficiency as defined in the course:

$$Hashing\ Efficiency = \frac{Packing\ Density}{Average\ Number\ of\ Reads\ per\ Record}$$

To calculate the average number of reads per record, we iterate through each list and count the number of reads required.

```
double HashTable::calculateHashEfficiency() const
{
    int numberOfReads = 0;
    for (const auto &list : table)
    {
        const Node *curr = list.get_head();
        for (int i = 1; curr != nullptr; i++)
        {
            numberOfReads += i;
            curr = curr->next;
        }
    }
    double averageReads = (double)numberOfReads / numberOfRecords;
    return calculatePackingDensity() / averageReads;
}
```

$$calculateHashEfficiency = \frac{calculatePackagingDensity}{averageReads}$$

The empty buckets require zero reads, and a bucket with one or more records will require 1 + 2 + ... + n reads, depending on the number of n records added to the linked list / bucket.

In the case of our input and hashing function results, the number of reads can be computed as follows:

- Bucket 0: 1 record = 1 read
- Bucket 1: 3 records = 1 + 2 + 3 = 6 reads
- Bucket 2: 0 records = 0 reads
- Bucket 3: 2 records = 1 + 2 = 3 reads
- Bucket 4: 3 records = 1 + 2 + 3 = 6 reads
- Bucket 5: 3 records = 1 + 2 + 3 = 6 reads

numberOfReads = 1 + 6 + 3 + 6 + 6 = 22 reads

$$averageReads = \frac{numberOfReads}{numberOfRecords} = \frac{22}{12} \approx 1.83$$

So, our hash efficiency is calculated as:

$$calculateHashEfficiency = \frac{2}{1.83} \approx \mathbf{1.09}$$

Please note that the calculateHashEfficiency function was implemented with this definition in mind, and because of our collisions and linked list implementation, the calculated hashing efficiency is over 1, which suggests that our definition is not suited for this case.

A more proper definition for describing the hashing efficiency of our algorithm is:

$$Hashing\ Efficiency = \frac{Ideal\ Average\ Reads\ per\ Record}{Actual\ Average\ Reads\ per\ Record}$$

Where the ideal average reads per record are the minimum average reads considering 6 buckets and 12 records. An ideal distribution occurs when all the buckets are filled on the first pass and on the second pass the buckets are equally filed:

- Bucket 0: 2 records = 1 + 2 = 3 reads
- Bucket 1: 2 records = 1 + 2 = 3 reads
- Bucket 2: 2 records = 1 + 2 = 3 reads
- Bucket 3: 2 records = 1 + 2 = 3 reads
- Bucket 4: 2 records = 1 + 2 = 3 reads
- Bucket 5: 2 records = 1 + 2 = 3 reads

Total ideal reads = 3 * 6 = 18 reads

$$Ideal\ Average\ Reads\ per\ Record = \frac{Total\ Ideal\ Reads}{Number\ of\ Records} = \frac{18}{12} = 1.5$$

So the proper hashing efficiency of our algorithm is:

$$Hashing\ Efficiency = \frac{Ideal\ Average\ Reads\ per\ Record}{Actual\ Average\ Reads\ per\ Record} = \frac{1.5}{1.83} \approx \mathbf{0.82}$$

Hashing function

The implemented hash function uses Knuth's Multiplicative Method, which was presented during the course as a better alternative to the ordinary Multiplicative Method for its simplicity and effective distribution properties. The algorithm provides a good distribution, as evidenced by the calculated hashing efficiency of 0.82.

The current hash function sums the ASCII values of the characters in the flight number, creating a unique value for each string. This sum is then multiplied by Knuth's constant, ensuring uniform distribution within the range.

Instead of simply summing ASCII values, an improvement is to use a polynomial hash or bitwise operations to better distribute the values.

Exercise B

Source code

AVL_tree.cpp

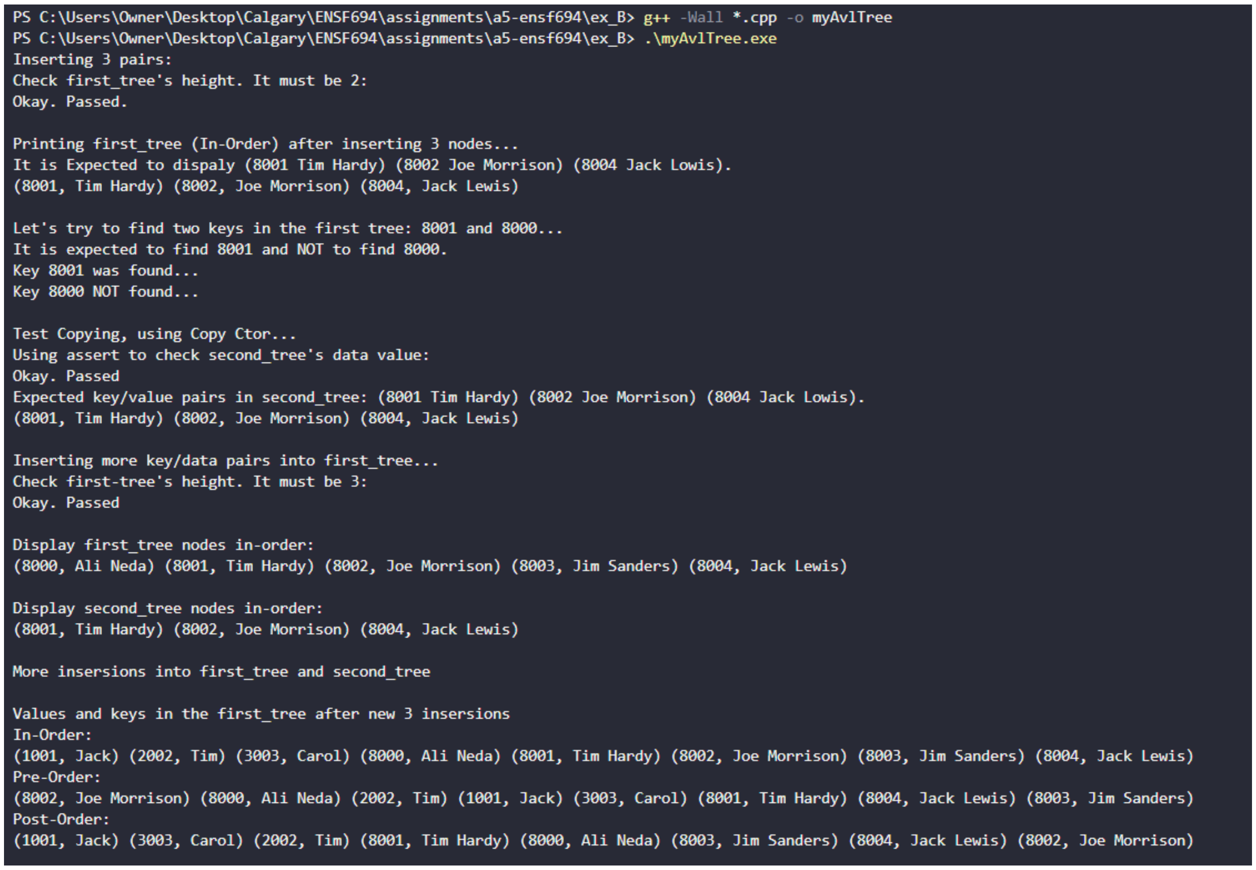
```
/**
 *  File Name: AVL_tree.cpp
 *  Assignment: ENSF 694 Summer 2024 - Lab 5 Exercise B
 *  Created by: Mahmood Moussavi on 2024-05-22
 *  Completed by: Yael Gonzalez
 *  Submission Date: August 2, 2024
 */

#include "AVL_tree.h"

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

int AVLTree::height(const Node *N)
{
    return (N == nullptr) ? 0 : N->height;
}

int AVLTree::getBalance(Node *N)
{
    return (N == nullptr) ? 0 : height(N->right) - height(N->left);
}

// Implemented based on: https://www.geeksforgeeks.org/insertion-in-an-avl-tree/
Node *AVLTree::rightRotate(Node *y)
{
    Node *x = y->left;
    Node *T2 = x->right;

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

    // Update parent pointers
    if (T2 != nullptr)
    {
        T2->parent = y;
    }
    x->parent = y->parent;
    y->parent = x;
```

```cpp
    // Update parent link to this node
    if (x->parent != nullptr)
    {
        if (x->parent->left == y)
        {
            x->parent->left = x;
        }
        else
        {
            x->parent->right = x;
        }
    }
    else
    {
        root = x;
    }

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

    // Return new root
    return x;
}

// Implemented based on: https://www.geeksforgeeks.org/insertion-in-an-avl-tree/
Node *AVLTree::leftRotate(Node *x)
{
    Node *y = x->right;
    Node *T2 = y->left;

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

    // Update parent pointers
    if (T2 != nullptr)
    {
        T2->parent = x;
    }
    y->parent = x->parent;
    x->parent = y;

    // Update parent link to this node
    if (y->parent != nullptr)
    {
        if (y->parent->left == x)
        {
            y->parent->left = y;
        }
        else
        {
            y->parent->right = y;
        }
    }
    else
    {
        root = y;
    }

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

    // Return new root
    return y;
}

void AVLTree::insert(int key, Type value)
{
    root = insert(root, key, value, nullptr);
}

// Recursive function
// Implemented based on: https://www.geeksforgeeks.org/insertion-in-an-avl-tree/
Node *AVLTree::insert(Node *node, int key, Type value, Node *parent)
{
    // 1. Do ordinary Binary Search Tree insertion
    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;
    }

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

    // 3. Get balance factor of this ancestor node
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases
    // Case 1: Left-Left
    if (balance < -1 && key < node->left->data.key)
    {
        return rightRotate(node);
    }

    // Case 2: Right-Right
    if (balance > 1 && key > node->right->data.key)
    {
        return leftRotate(node);
    }

    // Case 3: Left-Right
    if (balance < -1 && key > node->left->data.key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Case 4: Right-Left
    if (balance > 1 && key < node->right->data.key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // Otherwise, return (unchanged) node pointer
    return node;
}

// Recursive function
```

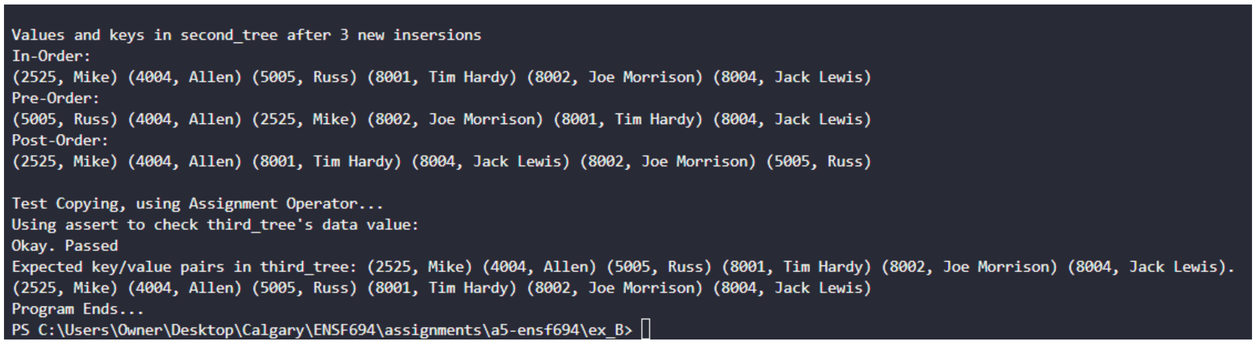
```cpp
void AVLTree::inorder(const Node *root)
{
    if (root == nullptr)
        return;

    // LVR
    inorder(root->left);
    std::cout << "(" << root->data.key << ", " << root->data.value << ")" << " ";
    inorder(root->right);
}

// Recursive function
void AVLTree::preorder(const Node *root)
{
    if (root == nullptr)
        return;

    // VLR
    std::cout << "(" << root->data.key << ", " << root->data.value << ")" << " ";
    preorder(root->left);
    preorder(root->right);
}

// Recursive function
void AVLTree::postorder(const Node *root)
{
    if (root == nullptr)
        return;

    // LRV
    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)
{
    if (root == nullptr)
    {
        cursor = nullptr;
        return;
    }

    if (root->data.key == key)
    {
        cursor = root;
        return;
    }

    if (root->data.key > key)
    {
        find(root->left, key);
    }
    else
    {
        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)
{
    if (node)
    {
        Node *new_node = new Node(node->data.key, node->data.value, parent);
        new_node->left = copy(node->left, new_node);
        new_node->right = copy(node->right, new_node);
        new_node->height = node->height;
        return new_node;
    }

    return nullptr;
}

// 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
    {
        std::cout << "looks like tree is empty, as cursor == Zero.\n";
        exit(1);
    }
}

const Type &AVLTree::cursor_datum() const
{
```

```cpp
    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



```
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a5-ensf694\ex_B> g++ -Wall *.cpp -o myAvlTree
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a5-ensf694\ex_B> .\myAvlTree.exe
Inserting 3 pairs:
Check first_tree's height. It must be 2:
Okay. Passed.

Printing first_tree (In-Order) after inserting 3 nodes...
It is Expected to dispaly (8001 Tim Hardy) (8002 Joe Morrison) (8004 Jack Lowis).
(8001, Tim Hardy) (8002, Joe Morrison) (8004, Jack Lewis)

Let's try to find two keys in the first tree: 8001 and 8000...
It is expected to find 8001 and NOT to find 8000.
Key 8001 was found...
Key 8000 NOT found...

Test Copying, using Copy Ctor...
Using assert to check second_tree's data value:
Okay. Passed
Expected key/value pairs in second_tree: (8001 Tim Hardy) (8002 Joe Morrison) (8004 Jack Lowis).
(8001, Tim Hardy) (8002, Joe Morrison) (8004, Jack Lewis)

Inserting more key/data pairs into first_tree...
Check first-tree's height. It must be 3:
Okay. Passed

Display first_tree nodes in-order:
(8000, Ali Neda) (8001, Tim Hardy) (8002, Joe Morrison) (8003, Jim Sanders) (8004, Jack Lewis)

Display second_tree nodes in-order:
(8001, Tim Hardy) (8002, Joe Morrison) (8004, Jack Lewis)

More insersions into first_tree and second_tree

Values and keys in the first_tree after new 3 insersions
In-Order:
(1001, Jack) (2002, Tim) (3003, Carol) (8000, Ali Neda) (8001, Tim Hardy) (8002, Joe Morrison) (8003, Jim Sanders) (8004, Jack Lewis)
Pre-Order:
(8002, Joe Morrison) (8000, Ali Neda) (2002, Tim) (1001, Jack) (3003, Carol) (8001, Tim Hardy) (8004, Jack Lewis) (8003, Jim Sanders)
Post-Order:
(1001, Jack) (3003, Carol) (2002, Tim) (8001, Tim Hardy) (8000, Ali Neda) (8003, Jim Sanders) (8004, Jack Lewis) (8002, Joe Morrison)
```



```
Values and keys in second_tree after 3 new insersions
In-Order:
(2525, Mike) (4004, Allen) (5005, Russ) (8001, Tim Hardy) (8002, Joe Morrison) (8004, Jack Lewis)
Pre-Order:
(5005, Russ) (4004, Allen) (2525, Mike) (8002, Joe Morrison) (8001, Tim Hardy) (8004, Jack Lewis)
Post-Order:
(2525, Mike) (4004, Allen) (8001, Tim Hardy) (8004, Jack Lewis) (8002, Joe Morrison) (5005, Russ)

Test Copying, using Assignment Operator...
Using assert to check third_tree's data value:
Okay. Passed
Expected key/value pairs in third_tree: (2525, Mike) (4004, Allen) (5005, Russ) (8001, Tim Hardy) (8002, Joe Morrison) (8004, Jack Lewis).
(2525, Mike) (4004, Allen) (5005, Russ) (8001, Tim Hardy) (8002, Joe Morrison) (8004, Jack Lewis)
Program Ends...
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a5-ensf694\ex_B>
```

Exercise C

Source code

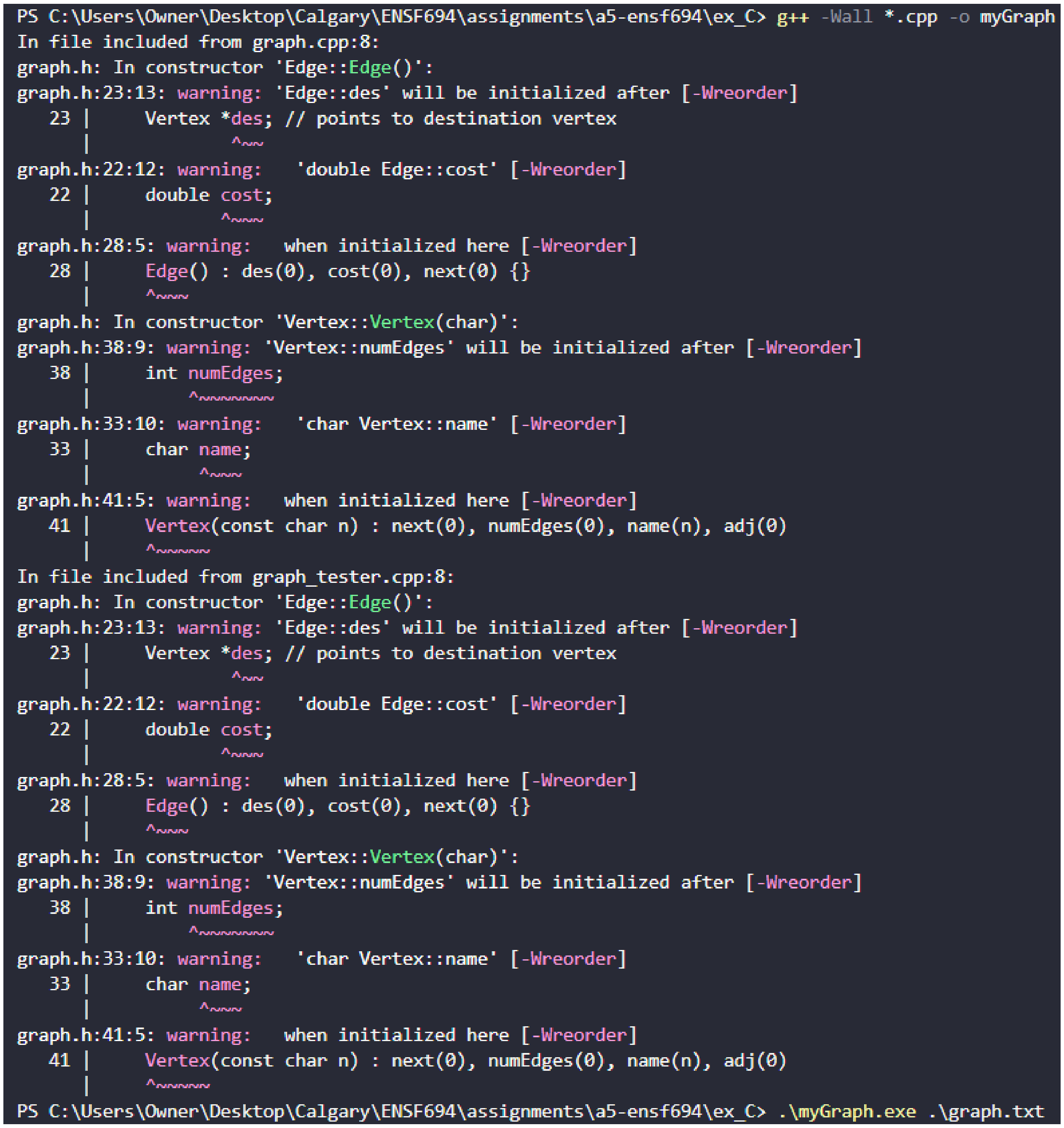
graph.cpp

```
/**
 *  File Name: graph.cpp
 *  Assignment: ENSF 694 Summer 2024 - Lab 5 Exercise C
 *  Created by: Mahmood Moussavi
 *  Completed by: Yael Gonzalez
 *  Submission Date: August 2, 2024
 */
#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;
```

```cpp
}

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)
{
    clearAll(); // Reset all vertices' distances and predecessors

    Vertex *s = getVertex(start);
    s->dist = 0;
    PriorityQueue q;
    q.enqueue(s);

    while (!q.isEmpty())
    {
        Vertex *v = q.dequeue();
        for (Edge *e = v->adj; e != nullptr; e = e->next)
        {
            Vertex *w = e->des;
            if (w->dist > v->dist + e->cost)
            {
                w->dist = v->dist + e->cost;
                w->prev = v;
                q.enqueue(w);
            }
        }
    }
}

void Graph::unweighted(const char start)
{
```

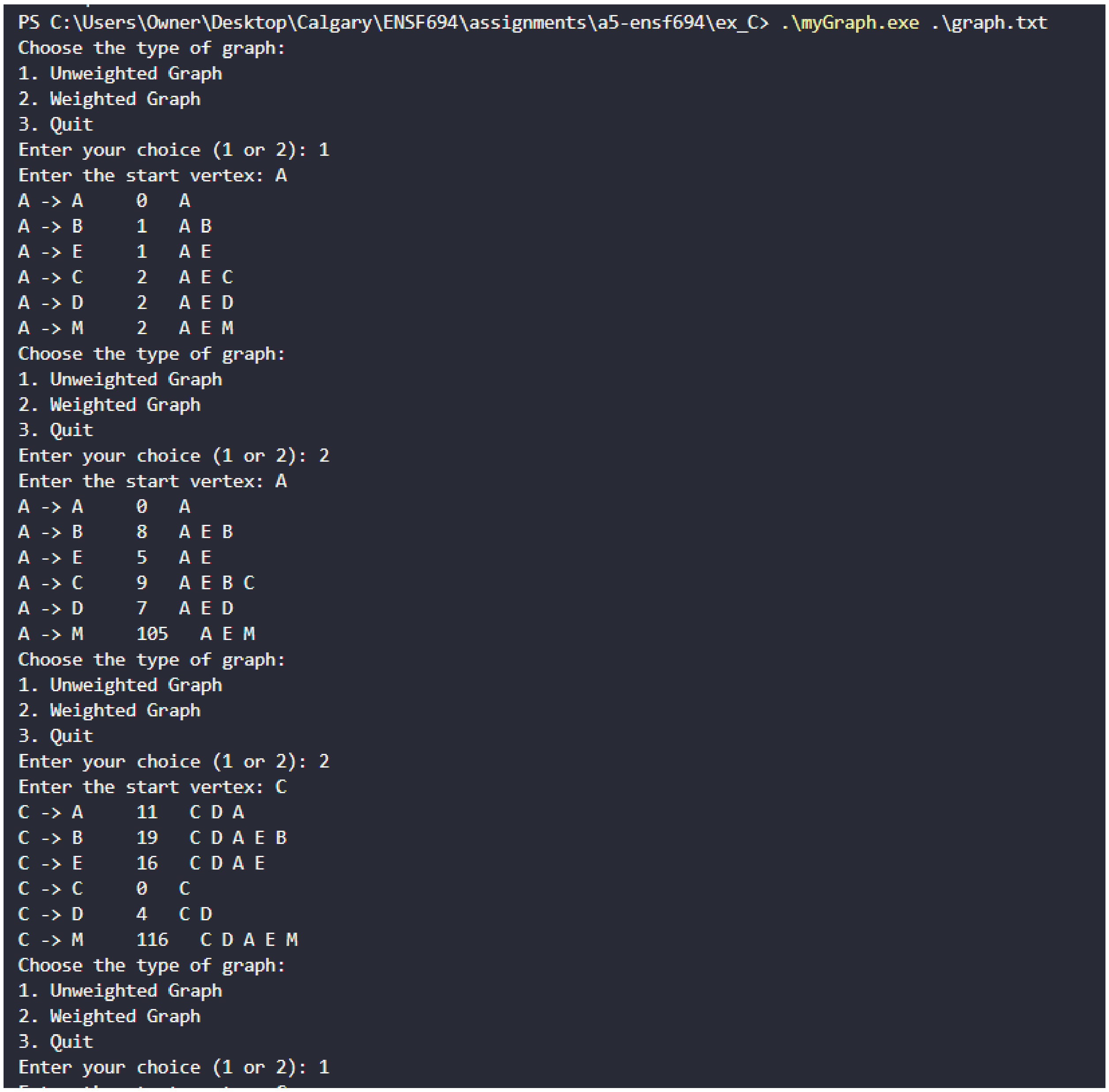
```cpp
    clearAll(); // Reset all vertices' distances and predecessors

    Vertex *s = getVertex(start);
    s->dist = 0;
    PriorityQueue q;
    q.enqueue(s);

    while (!q.isEmpty())
    {
        Vertex *v = q.dequeue();
        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.enqueue(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

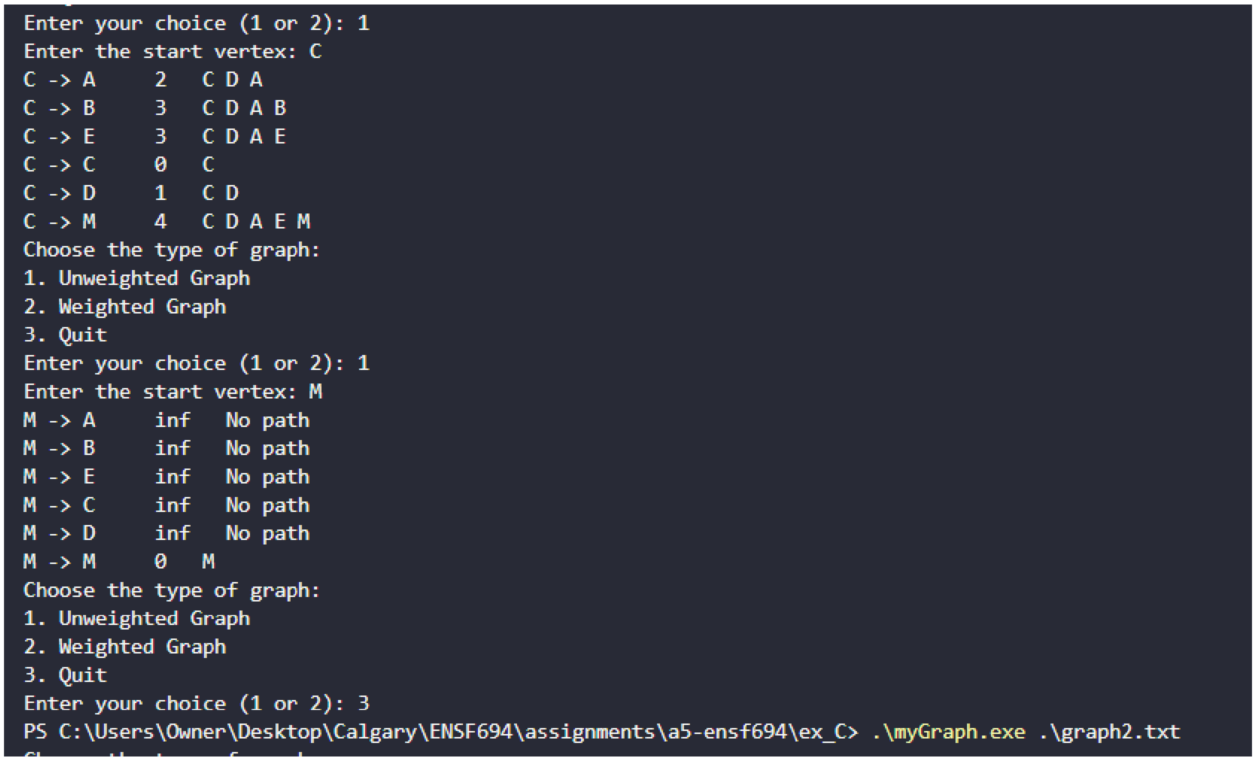


```
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a5-ensf694\ex_C> g++ -Wall *.cpp -o myGraph
In file included from graph.cpp:8:
graph.h: In constructor 'Edge::Edge()':
graph.h:23:13: warning: 'Edge::des' will be initialized after [-Wreorder]
   23 |     Vertex *des; // points to destination vertex
      |             ^~~
graph.h:22:12: warning:   'double Edge::cost' [-Wreorder]
   22 |     double cost;
      |            ^~~~
graph.h:28:5: warning:   when initialized here [-Wreorder]
   28 |     Edge() : des(0), cost(0), next(0) {}
      |     ^~~~
graph.h: In constructor 'Vertex::Vertex(char)':
graph.h:38:9: warning: 'Vertex::numEdges' will be initialized after [-Wreorder]
   38 |     int numEdges;
      |         ^~~~~~~~
graph.h:33:10: warning:   'char Vertex::name' [-Wreorder]
   33 |     char name;
      |          ^~~~
graph.h:41:5: warning:   when initialized here [-Wreorder]
   41 |     Vertex(const char n) : next(0), numEdges(0), name(n), adj(0)
      |     ^~~~~~
In file included from graph_tester.cpp:8:
graph.h: In constructor 'Edge::Edge()':
graph.h:23:13: warning: 'Edge::des' will be initialized after [-Wreorder]
   23 |     Vertex *des; // points to destination vertex
      |             ^~~
graph.h:22:12: warning:   'double Edge::cost' [-Wreorder]
   22 |     double cost;
      |            ^~~~
graph.h:28:5: warning:   when initialized here [-Wreorder]
   28 |     Edge() : des(0), cost(0), next(0) {}
      |     ^~~~
graph.h: In constructor 'Vertex::Vertex(char)':
graph.h:38:9: warning: 'Vertex::numEdges' will be initialized after [-Wreorder]
   38 |     int numEdges;
      |         ^~~~~~~~
graph.h:33:10: warning:   'char Vertex::name' [-Wreorder]
   33 |     char name;
      |          ^~~~
graph.h:41:5: warning:   when initialized here [-Wreorder]
   41 |     Vertex(const char n) : next(0), numEdges(0), name(n), adj(0)
      |     ^~~~~~
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a5-ensf694\ex_C> .\myGraph.exe .\graph.txt
```

graph.txt

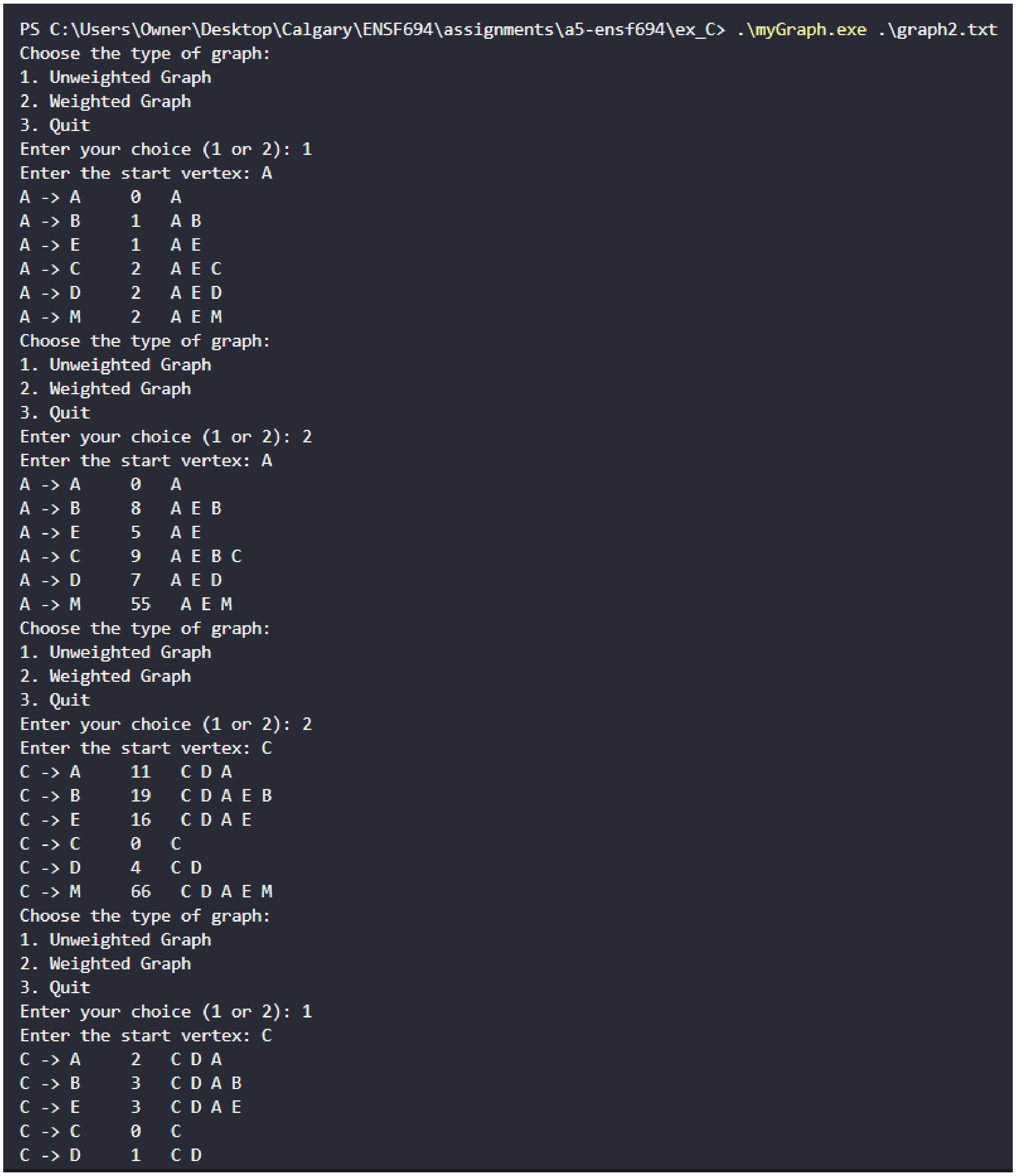


```
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a5-ensf694\ex_C> .\myGraph.exe .\graph.txt
Choose the type of graph:
1. Unweighted Graph
2. Weighted Graph
3. Quit
Enter your choice (1 or 2): 1
Enter the start vertex: A
A -> A     0   A
A -> B     1   A B
A -> E     1   A E
A -> C     2   A 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: A
A -> A     0   A
A -> B     8   A E B
A -> E     5   A E
A -> C     9   A E B C
A -> D     7   A E D
A -> M     105   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): 1
```

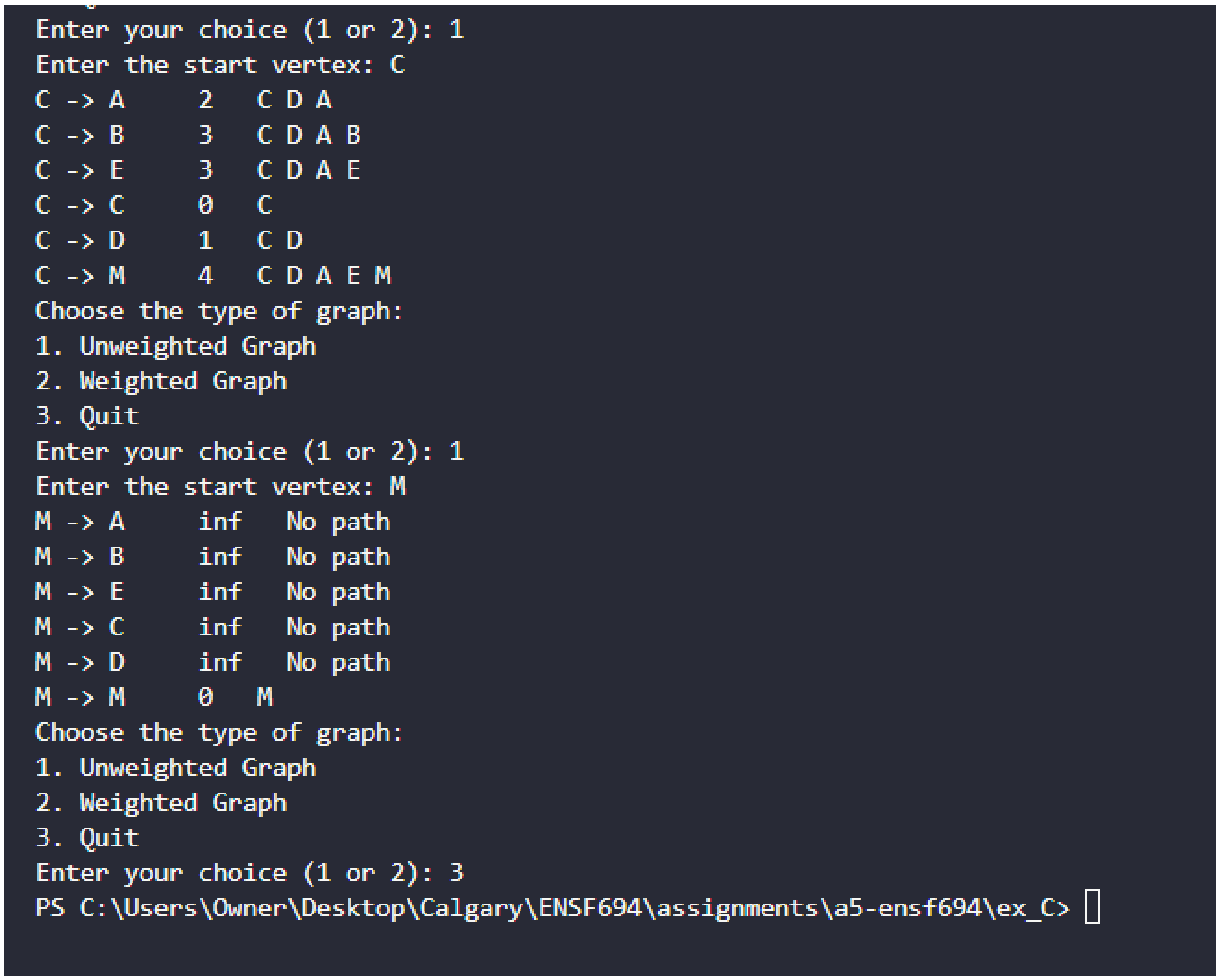


```
Enter your choice (1 or 2): 1
Enter the start vertex: C
C -> A     2   C D A
C -> B     3   C D A B
C -> E     3   C D A E
C -> C     0   C
C -> D     1   C D
C -> M     4   C D A E M
Choose the type of graph:
1. Unweighted Graph
2. Weighted Graph
3. Quit
Enter your choice (1 or 2): 1
Enter the start 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
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a5-ensf694\ex_C> .\myGraph.exe .\graph2.txt
```

graph2.txt



```
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a5-ensf694\ex_C> .\myGraph.exe .\graph2.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: A
A -> A     0   A
A -> B     8   A E B
A -> E     5   A E
A -> C     9   A E B C
A -> D     7   A E D
A -> M     55   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     66   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: 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
```



```
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
PS C:\Users\Owner\Desktop\Calgary\ENSF694\assignments\a5-ensf694\ex_C>
```