Course: ENSF 694 – Summer 2024
Lab Assignment #: Lab 5
Instructor: Mahmood Moussavi
Student Name: Jeff Wheeler, UCID: 30265340
Submission Date: August 2, 2024

Exercise A

HashTable Code:

```
/*
 *  HashTable.cpp
 *  ENSF 694 Lab 5 - Exercise A
 *  Created by Jeff Wheeler
 *  Submission date: August 2, 2024
 */

#include "HashTable.h"
#include <iostream>

unsigned int HashTable::hashFunction(const string &flightNumber) const {
	unsigned int hash_number;
	for (int i = 0; i < int(flightNumber.size()); i++){
	hash_number += int(flightNumber[i]);
	}
	hash_number = ((67*hash_number + 41)%101);
	return hash_number % this->tableSize;
}

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

void HashTable::insert(const Flight &flight){
	if(this->insertFirstPass(flight)){
	return;
	}
	this->insertSecondPass(flight);
}

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

```
void HashTable::insertSecondPass(const Flight &flight){
      unsigned int position = this->hashFunction(flight.flightNumber);
      table.at(position).insert(flight);
      numberOfRecords++;
}

Flight* HashTable::search(const string &flightNumber) const{
      Flight* temp = nullptr;
      for (int i = 0; i < int(tableSize); i++){
      temp = table.at(i).search(flightNumber);
      if (temp != nullptr){
            return temp;
      }
      }
      return temp;
}

double HashTable::calculatePackingDensity() const{
      if (tableSize > 0)
      return double(firstPassRecords)/tableSize;
      return -1.0;
}

double HashTable::calculateHashEfficiency() const{
      if (firstPassRecords > 0)
      return
this->calculatePackingDensity()/(numberOfRecords/firstPassRecords);
      return -1.0;
}

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

read_flight_info Code:

```
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 = "C:/Users/jeffw/Documents/_Software Masters/ENSF 
694/ensf694_assignment5/";
	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();
}
```

Output:

```
jeffw@DESKTOP-SR7596T /cygdrive/c/Users/jeffw/Documents/_Software Masters/ENSF 694/ensf694_assignment5
$ ./hashtable input.txt
Packing Density: 2
Hash Efficiency: 1
Hash Table Contents:
Bucket 0:
Flight Number: AMA11231, Origin: Calgary, Destination: Toronto, Date: 2024-05-30, Time: 00:45, Capacity: 576
Flight Number: WJ12301, Origin: Calgary, Destination: Toronto, Date: 2024-05-30, Time: 2:45, Capacity: 476
Bucket 1:
Flight Number: DELTA2332, Origin: Otawa, Destination: Toronto, Date: 2024-05-30, Time: 10:45, Capacity: 200
Flight Number: AMA11232, Origin: Otawa, Destination: Toronto, Date: 2024-05-30, Time: 00:45, Capacity: 576
Flight Number: AMA1123, Origin: Calgary, Destination: Edmonton, 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
Bucket 2:
Flight Number: DELTA233, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 10:45, Capacity: 200
Flight Number: WJ1230, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 2:45, Capacity: 476
Bucket 3:
Flight Number: AC1231, Origin: Calgary, Destination: Toronto, Date: 2024-05-30, Time: 1:45, Capacity: 376
Bucket 4:
Flight Number: AC1232, Origin: Otawa, Destination: Toronto, Date: 2024-05-30, Time: 1:45, Capacity: 376
Bucket 5:
Flight Number: DELTA2331, Origin: Calgary, Destination: Toronto, Date: 2024-05-30, Time: 10:45, Capacity: 200
Flight Number: AC123, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 1:45, Capacity: 376
Enter flight number to search (or 'exit' to quit):
```

Calculation and discussion:

Packing Density = # of records / # of spaces = 12 / 6 = 2

Hashing Efficiency = packing density / average # of reads per record

Average reads per record = reads per record / # of spaces = (2 + 4 + 2 + 1 + 1+ 2)/ 6 = 2

Hashing efficiency = 2 / 2 = 100%

I used the fold and sum method using the flight number as the unique identifier. Then I used a universal hash function on that result to improve the hashing efficiency. The hash function could be improved by using more data than just the key when doing the fold and add method to avoid potentially duplicates.

Exercise B

Code:

```
/*
 *  AVL_tree.cpp
 *  ENSF 694 Lab 5 - Exercise B
 *  Created by Mahmood Moussavi
 *  Completed by Jeff Wheeler
 *  Submission date: August 2, 2024
 */

#include "AVL_tree.h"

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

Node* AVLTree::rightRotate(Node* node) {
    Node* pivot = node->left;

    node->left = pivot->right;
    if (pivot->right != nullptr)
        pivot->right->parent = node;

    if (node->parent == nullptr)
        root = pivot;
    else if (node->parent->right == node)
        node->parent->right = pivot;
    else if (node->parent->left == node)
        node->parent->left = pivot;

    pivot->parent = node->parent;
    pivot->right = node;
    node->parent = pivot;
    node->height = 1;
    pivot->height = 2;
    return pivot;
}
```

```cpp
Node* AVLTree::leftRotate(Node* node) {
    Node* pivot = node->right;

    node->right = pivot->left;
    if (pivot->left != nullptr)
        pivot->left->parent = node;

    if (node->parent == nullptr)
        root = pivot;
    else if (node->parent->right == node)
        node->parent->right = pivot;
    else if (node->parent->left == node)
        node->parent->left = pivot;

    pivot->parent = node->parent;
    pivot->left = node;
    node->parent = pivot;
    node->height = 1;
    pivot->height = 2;
    return pivot;
}

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){ // setup initial root node
        return new Node(key, value, parent);
    }
    else if (key < node->data.key){ // insert lower key value to left
        node->left = insert(node->left, key, value, node);
    }
    else if (key > node->data.key){ // insert higher key value to right
        node->right = insert(node->right, key, value, node);
    }

    // adjust height of node
    if (height(node->left) > height(node->right)){
        node->height = height(node->left) + 1;
    }
    else if (height(node->left) < height(node->right)){
        node->height = height(node->right) + 1;
    }
```

```
    // else height stays the same

    // 4 cases
    if (getBalance(node) == -2){
        // LL
        if (key < node->left->data.key) {
            return rightRotate(node);
        }
         // LR
         node->left = leftRotate(node->left);
         return rightRotate(node);
    }
    else if (getBalance(node) == 2){
        // RR
        if (key > node->data.key){
            return leftRotate(node);
        }
        // RL
        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);
    std::cout << "(" << root->data.key << " " << root->data.value << ")"
<< std::endl;
    inorder(root->right);
}

// Recursive function
void AVLTree::preorder(const Node* root) {
    if (root == nullptr){
        return;
    }
    std::cout << "(" << root->data.key << " " << root->data.value << ")"
<< std::endl;
    preorder(root->left);
    preorder(root->right);
}
```

```cpp
// Recursive function
void AVLTree::postorder(const Node* root) {
    if (root == nullptr){
        return;
    }
    postorder(root->left);
    postorder(root->right);
    std::cout << "(" << root->data.key << " " << root->data.value << ")"
<< std::endl;
}

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* node, int key){
    if (node == nullptr){
        return;
    }

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

    if (key < node->data.key)
        find(node->left, key);
    else if (key > node->data.key)
        find(node->right, key);
}

AVLTree::AVLTree(const AVLTree& other) : root(nullptr), cursor(nullptr) {
    root = copy(other.root, nullptr);
    cursor = root;
}
```

```cpp
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* new_node = new Node(node->data.key, node->data.value, parent);
    new_node->height = node->height;

    new_node->left = copy(node->left, new_node);
    new_node->right = copy(node->right, new_node);
    return new_node;
}

// Recusive function
void AVLTree::destroy(Node* node) {
    if (node) {
        destroy(node->left);
        destroy(node->right);
        delete node;
    }
    root = nullptr;
}

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;
}
```

Output:

```
jeffw@DESKTOP-SR7596T /cygdrive/c/Users/jeffw/Documents/_Software Masters/ENSF 694/ensf694_assignment5
$ ./AVL_tree
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 display (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...
```

Exercise C

Code:

```
/*
 *  graph.cpp.cpp
 *  ENSF 694 Lab 5 - Exercise C
 *  Created by Mahmood Moussavi
 *  Completed by Jeff Wheeler
 *  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;
}
```

```
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
}
```

```cpp
void Graph::clearAll() {
    Vertex* ptr = head;
    while (ptr) {
        ptr->reset();
        ptr = ptr->next;
    }
}

void Graph::dijkstra(const char start) {
    Vertex* s = getVertex(start);

    Vertex* traverse = head;
    while (traverse != nullptr){
        traverse->reset();
        traverse = traverse->next;
    }

    PriorityQueue q;
    q.enqueue(s);
    s->dist = 0;
    double new_dist;
    while(!q.isEmpty()){
        Vertex* v = q.dequeue();
        for (Edge* edge = v->adj; edge != nullptr; edge = edge->next){
            Vertex* w = edge->des;
            new_dist = v->dist + edge->cost;
            if (new_dist < w->dist){
                w->dist = new_dist;
                w->prev = v;
                q.enqueue(w);
            }
        }
    }



}

void Graph::unweighted(const char start) {
    Vertex* s = getVertex(start);

    Vertex* traverse = head;
    while (traverse != nullptr){
        traverse->reset();
        traverse = traverse->next;
    }
```

```cpp
    PriorityQueue q;
    q.enqueue(s);
    s->dist = 0;
    while(!q.isEmpty()){
        Vertex* v = q.dequeue();
        for (Edge* edge = v->adj; edge != nullptr; edge = edge->next){
            Vertex* w = edge->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;
    }
}
```

Output (using graph2.txt):

```
jeffw@DESKTOP-SR7596T /cygdrive/c/Users/jeffw/Documents/_Software Masters/ENSF 694/ensf694_assignment5
$ ./graph_test 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): 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): 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
```