Course: ENSF 694 – Summer 2024
Lab #: 5
Instructor: M. Moussavi
Student Name: Jaskirat Singh (Jazz)
Submission Date: 2 August 2024

Exercise A HashTable Code:

```cpp
/*
 *  HashTable.cpp
 *  ENSF 694 Lab 5, exercise A
 *  Completed by: Jaskirat Singh
 *  Submission date: August 2
 */

#include "HashTable.h"

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

//Hash function
unsigned int HashTable::hashFunction(const string &flightNumber) const {
    //Implementation of a hash function based on the flight number
    unsigned int hash = 0;
    //Iterate through all characters
    for(char c : flightNumber) {
        //h(k) = ((a * k + b) % p) % m from notes
        hash = ((hash * 31 + c) % 17) % tableSize;
    }
    return hash;
}

//Insert flight into hash table
void HashTable::insert(const Flight &flight) {
    //Get hash
    unsigned int index = hashFunction(flight.flightNumber);
    //Insert into table
    table[index].insert(flight);
    numberOfRecords++;
}

//Insert flight into hash table in the first pass
bool HashTable::insertFirstPass(const Flight &flight) {
    unsigned int index = hashFunction(flight.flightNumber);
    //Check if space is available
    if(table[index].isEmpty()) {
        table[index].insert(flight);
        numberOfRecords++;
        return true;
    }
    return false;
}

//Insert flight into hash table in the second pass
void HashTable::insertSecondPass(const Flight &flight) {
    //Insert into linked list
    insert(flight);
}

//Search for a flight in the hash table
Flight* HashTable::search(const string &flightNumber) const {
    unsigned int index = hashFunction(flightNumber);
    return table[index].search(flightNumber);
}

//Calculate packing density
double HashTable::calculatePackingDensity() const {
```

```
    return numberOfRecords / tableSize;
}

//Calculate hash efficiency
double HashTable::calculateHashEfficiency() const {
    unsigned int reads = 0;

    //Go over each bucket, use new numNodes() function from List.cpp
    for(unsigned int i = 0; i < tableSize; i++) {
        //Get number of nodes in current bucket
        reads += table[i].numNodes();
    }

    //Calculate the average number of probes
    double averageReads = reads/numberOfRecords;

    //Hashing efficiency = packing density / average number of reads per record
    return calculatePackingDensity() / averageReads;
}

//Display the hash table
void HashTable::display() const {
    for(unsigned int i = 0; i < tableSize; i++){
        cout << "Bucket " << i << ": ";
        table[i].display();
        cout << endl;
    }
}
```

Exercise A numNodes() Code in List.cpp:

```
int List::numNodes() const {
    int count = 0;
    Node* current = head;

    //Loop while counting number of nodes
    while (current){
        count++;
        current = current->next;
    }
    return count;
}
```

Exercise A 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 = "/Users/aether/Documents/ENSF-694/ENSF-694Lab5/ExerciseA/";
    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();
}
```

Exercise A program output:

```
Packing Density: 2
Hash Efficiency: 2
Hash Table Contents:
Bucket 0: Flight Number: DELTA2331, Origin: Calgary, Destination: Toronto, Date: 2024-05-30, Time: 10:45, Capacity: 200

Bucket 1: Flight Number: DELTA2332, Origin: Otawa, Destination: Toronto, Date: 2024-05-30, Time: 10:45, Capacity: 200

Bucket 2: 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
Flight Number: AC123, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 1:45, Capacity: 376

Bucket 3: 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
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: AC1232, Origin: Otawa, Destination: Toronto, Date: 2024-05-30, Time: 1:45, Capacity: 376

Bucket 5: 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):
```

Exercise A packing density and hash efficiency:

Packing density = numberOfRecords / tableSize
Hashing efficiency = packing density / average number of reads per record

Exercise A Description:

Using the formula h(k) = ((a * k + b) % p) % m from the notes, I used the hashing formula of((hash * 31 + c) % 17) % tableSize. I chose a prime number larger than the key values, big enough to ensure an even distribution. % tableSize to ensure that the values stay within the bounds of the table.

Possible improvements to the function can be by using double hashing when a collision occurs, to help spread the values out evenly post-collision. Using a larger prime number can also help reduce the number of collisions.

Exercise B Code:

```
/*
 *  AVL_tree.cpp
 *  ENSF 694 Lab 5, exercise B
 *  Completed by: Jaskirat Singh
 *  Submission date: August 2
 */

#include "AVL_tree.h"

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

int AVLTree::height(const Node* N) {
    //If N is nullptr then return height = 0
    if(N == nullptr) return 0;
    return N->height;
}

int AVLTree::getBalance(Node* N) {
    //If N is nullptr then return height = 0
    if(N == nullptr) return 0;
    //Balance = left height - right height
    //Notes say right-left but online theory says left-right
    //Assignment results work better with left-right
    return height(N->left) - height(N->right);
}

Node* AVLTree::rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    //Rotate
    x->right = y;
    y->left = T2;

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

    return x;
 }

 Node* AVLTree::leftRotate(Node* x) {
     Node* y = x->right;
     Node* T2 = y->left;

     //Rotate
     y->left = x;
     x->right = T2;

     //New height
     x->height = max(height(x->left), height(x->right)) + 1;
     y->height = 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) {
     if (node == nullptr) {return new Node(key, value, parent);} // Node(left, right, parent) format

     //Call insert function with node depending on key
     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;} //No duplicate keys

     //Update height to the larger value
     node->height = 1 + max(height(node->left), height(node->right));

     //Balance tree
     int balance = getBalance(node);

     //4 Cases:
     //Right rotate
     if (balance > 1 && key < node->left->data.key)
         return rightRotate(node);

     //Left rotate
     if (balance < -1 && key > node->right->data.key)
         return leftRotate(node);

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

     //Right Left Case
     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){
        inorder(root->left);
        cout << root->data.key << " ";
        inorder(root->right);
    }
}

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

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

```
        postorder(root->left);
        postorder(root->right);
        cout << root->data.key << " ";
    }
}

const Node* AVLTree::getRoot(){
    cursor = root;
    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){
        cout << "Key not found" << endl;
        return;
    }
    //Recursively call the function until root is the desired node
    if(key < root->data.key) {find(root->left, key);}
    else if(key > root->data.key) {find(root->right, key);}
    else{
        //Point cursor to the desired node
        cursor = root;
        cout << "Key:" << root->data.key << " Value: " << root->data.value << endl;
    }
}

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

AVLTree::~AVLTree() {
    destroy(root);
    root = nullptr;
}

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; }

    //Create new node
    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{
        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(){
    cursor = root;
}
```

Exercise B Program Output:

```
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 8002 8004

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 Value: Tim Hardy
Key 8001 was found...
Key not 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 8002 8004

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 8001 8002 8003 8004

Display second_tree nodes in-order:
8001 8002 8004

More insersions into first_tree and second_tree

Values and keys in the first_tree after new 3 insersions
In-Order:
1001 2002 3003 8000 8001 8002 8003 8004
Pre-Order:
8002 8000 2002 1001 3003 8001 8004 8003
Post-Order:
1001 3003 2002 8001 8000 8003 8004 8002

Values and keys in second_tree after 3 new insersions
In-Order:
2525 4004 5005 8001 8002 8004
Pre-Order:
5005 4004 2525 8002 8001 8004
Post-Order:
2525 4004 8001 8004 8002 5005

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 4004 5005 8001 8002 8004
Program Ends...
Program ended with exit code: 0
```

Exercise C Code:

```
/*
 *  grah.cpp
 *  ENSF 694 Lab 5, exercise C
 *  Completed by: Jaskirat Singh
 *  Submission date: August 2
 */

#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) {
    //Clear any previous values
    clearAll();
    //Step 1: initialization
    PriorityQueue pq;
    Vertex* startVertex = getVertex(start);
    //Set distance to 0
    startVertex->dist = 0;
    pq.enqueue(startVertex);

    //Step 2: main loop
    while (!pq.isEmpty()){
        Vertex* v = pq.dequeue();

        //Consider neighbours
        for(Edge* e = v->adj; e != nullptr; e = e->next){
            Vertex* w = e->des;
            double newDist = v->dist + e->cost;

            if(newDist < w->dist){
                w->dist = newDist;
                w->prev = v;
                pq.enqueue(w);
            }
        }
    }
}
```

```cpp
void Graph::unweighted(const char start) {
    //Clear any previous values
    clearAll();
    queue<Vertex*> q;

    //Visit start vertex and label it 0
    Vertex* startVertex = getVertex(start);
    //Initialize dist of 0
    startVertex->dist = 0;
    q.push(startVertex);

    while (!q.empty()) {
        Vertex* v = q.front();
        //Remove vertex V
        q.pop();

        //Consider neighbours
        for(Edge* e = v->adj; e != nullptr; e = e->next){
            Vertex* w = e->des;

            //Check if visited before
            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;
    }
}
```

Exercise C Program Output:

```
(base) aether@Jaskirats-MacBook-Pro ExerciseC % ./
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 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) aether@Jaskirats-MacBook-Pro ExerciseC %
```