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

/\*\*

 \*  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

A screenshot of a computer program

Description automatically generated

## Packing density and hash efficiency

Packing density as defined in the course:

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

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

Hashing Efficiency as defined in the course:

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;

}

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

So, our hash efficiency is calculated as:

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:

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

So the proper hashing efficiency of our algorithm is:

## 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;

    // 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

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

{

    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

Description automatically generated

A screenshot of a computer code

Description automatically generated

# 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;

}

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*)

{

    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

A computer screen shot of a program code

Description automatically generated

### graph.txt

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

### graph2.txt

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated