Algorithms and Data Structures

Spring 2019

Assignment 9

Date: April 23, 2019

Taiyr Begeyev

Problem 9.1 Hash Tables

(a) Given the sequence 3, 10, 2, 4 , apply the double-hashing strategy for open addressing to store the sequence in the given order in a hash table of size $m = 5$ with hash functions $h_1(k) = k$ mod 5 and $h_2(k) = 7k$ mod 8. Document all collisions and how they are resolved. Write down your computations.

Double hashing uses a hash function of the form:

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

The initial probe goes to position $T[h_1(k)]$; successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m.

Insertion by double-hashing:
1. Insert 3
First, we calculate $h_1(3) = 3 \bmod 5 = 3$
Since the slot 3 in the hash table is empty, we insert the key 3 into it.
2. Insert 10
Again, we start with calculating $h_1(k) = 10 \bmod 5 = 0$
Since the slot 0 in the hast table is empty, we insert the key 10 into it.
3. Insert 2
$h_1(k) = 2 \bmod 5 = 2$
Since the slot 2 in the hash table is not occupied, we insert the key 2 into it.
4. Insert 4
$h_1(k) = 4 \bmod 5 = 4$
Since the slot 4 in the hash table is not occupied, we insert the key 4 into it.
There are 0 collisions. That's how the final hash table looks like:

10
2
3
4

(b) Implement a hash table that supports insertion and querying with open addressing using linear probing. Select an h' function and explain why your selected h' is well-suited for your test data. The implementation should be consistent with the following or equivalent class specifications:

Implementation of the **HashTable** class. **hashTable.cpp**

```
/*
__________           .__
\__    ___/ _____    |__|  ___.__. _______
  |    |    \__  \   |  | <   |  | \_  __ \
  |    |     / __ \_ |  |  \___  |  |  | \/
  |____|    (____  / |__| / ____|  |__|
                \/        \/
*/

#include "hashTable.h"
#include <iostream>
using namespace std;

/*
    @brief parametrical constructor
*/
```

```
Node::Node(int key, int value) {
    this->key = key;
    this->value = value;
}

/*
    @brief constructor
    by default maxSize is 100
*/

HashTable::HashTable(int maxSize) {
    this->maxSize = maxSize;
    currentSize = 0;

    // allocate memory
    arr = new Node*[maxSize];
    for (int i = 0; i < maxSize; i++) {
        arr[i] = nullptr;
    }
}

/*
    @brief destructor
*/
HashTable::~HashTable() {
    for (int i = 0; i < maxSize; i++) {
        delete arr[i];
    }
    delete[] arr;
}

/*
    @brief Hash Function
*/

int HashTable::hashCode(int key) {
    return key % maxSize;
}

/*
    @brief Insert element at a key
*/

void HashTable::insertCode(int key, int value) {
    int hashValue = (hashCode(key)) % maxSize;
    int init = -1;

    int i = 1;
    while (hashValue != init && arr[hashValue]!= nullptr && arr[hashValue]->key != key)
    {
        if (init == -1) {
            init = hashValue;
        }
        hashValue = (hashCode(key) + i) % maxSize;
        i++;
    }

    if (arr[hashValue] == nullptr)
    {
        arr[hashValue] = new Node(key, value);
        currentSize++;
    }
}

/*
    @brief Search for an element with the given key
    @return the position of the element we are looking for
*/
int HashTable::get(int key) {
    for (int i = 0; i < maxSize; i++) {
        int hashValue = (hashCode(key) + i) % maxSize;
        if (arr[hashValue] == nullptr) {
            return -1;
        }
        else if (arr[hashValue]->key == key) {
            return hashValue;
        }
    }
    return -1;
}

/*
    @brief check whether the HashTable is empty
*/

bool HashTable::isEmpty() {
    return currentSize == 0;
}
```

To execute run **make**

Problem 9.2 Greedy Algorithms

(a) Show that a greedy algorithm for the activity-selection problem that makes the greedy choice of selecting the activity with shortest duration may fail at producing a globally optimal solution.

Our lemma is that the greedy choice of picking activity with the shortest duration as a first choice is optimal choice. Let's prove that it is wrong. Let's consider the following example. We have the set S containing 3 activities. Each of these activities has its corresponding start and end times. $S = \{[2,6],[5,7],[6,13]\}$. The activity with the shortest duration is $[5,7]$. The following one is $[2,6]$. Since it overlaps with $[5,7]$, we cannot pick it. The same applies for $[6,13]$. Consequently, our result is only one activity, which is definitely not the most optimal one. If we used another approach with the greedy choice of picking a_1 as first choice, it would lead to the optimal choice, which is indeed $\{[2,6],[6,13]\}$.

(b) Assuming an unsorted sequence of activities, derive a greedy algorithm for the activity-selection problem that selects the activity with the latest starting time. Your solution should not simply sort the activities and then select the activity.

```
/*
    Algorithms and Data Structures
    Spring 2019
    Assignment #8
    @file ActivitySelectionProblem.cpp
    @author Taiyr Begeyev
    @version 1.0 23/04/19
*/
```

```
/*
__________           .__
\__    ___/ _____    /__/  ___.__. _______
  |    |    \__  \   |  | <   |  | \_  __ \
  |    |     / __ \_ |  |  \___  |  |  | \/
  |____|    (____  / |__|  / ____|  |__|
                \/        \/
*/
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

struct Activity {
    int startTime;
    int finishTime;
};

/*
    @brief print the elements
*/
void print(vector<Activity> S) {
    int n = S.size();
    for (int i = 0; i < n; i++) {
        cout << "Activity " << i + 1<< ":" << endl;
        cout << S[i].startTime << " " << S[i].finishTime << endl << endl;
    }
}

/*
    @brief Greedy algortithm that solves the activity selection problem

    As our greedy choice we pick the activitu that has the latest start time

    @params the structure with our activities
    @return the structure with the solution
*/

vector <Activity> ActivitySelection(vector <Activity> S) {
    // this struct will contain the solution
    vector<Activity> ActivitySolution;

    int latestStartTime;
    int latestStartTimeIndex;
    while (!S.empty()) {
        latestStartTime = 0;
        // select activity with the latest start time
        for (int i = 0; i < S.size(); i++) {
            if (S[i].startTime > latestStartTime) {
                latestStartTime = S[i].startTime;
                latestStartTimeIndex = i;
            }
        }

        // check if it overlaps with activities in the ActivitySolution
        bool overLaps = false;
        for (int i = 0; i < ActivitySolution.size(); i++) {
            if (S[latestStartTimeIndex].finishTime > ActivitySolution[i].startTime) {
                overLaps = true;
            }
        }

        // push it to the ActivitySolution
        if (!overLaps)
            ActivitySolution.push_back(S[latestStartTimeIndex]);

        // delete this activity from S
        S.erase(S.begin() + latestStartTimeIndex);
    }
    return ActivitySolution;
}

int main() {
    vector <Activity> myActivities1, myActivities2, myActivities3;
    myActivities1 = {{1, 2}, {2, 3}, {4, 10}};
    myActivities2 = {
        {1, 4}, {3, 5},
        {0, 6}, {5, 7},
        {3, 8}, {5, 9},
        {6, 10}, {8, 11},
        {8, 12}, {2, 13}, {2, 14}
    };
    myActivities3 = {
        {2, 14}, {2, 13},
        {8, 12}, {8, 11},
        {5, 9}, {3, 8}
    };

    myActivities1 = ActivitySelection(myActivities1);
    myActivities2 = ActivitySelection(myActivities2);
    myActivities3 = ActivitySelection(myActivities3);

    // print the result
    print(myActivities1);
    print(myActivities2);
    print(myActivities3);

    return 0;
}
```