CH08-320143: Assignment #10

Harit Krishan
h.krishan@jacobs-university.de

Jacobs University Bremen — April 19, 2020

1 Hash Tables

A hash table is a data structure which stores data in an associative manner where each data value has a unique index value, determined by the hash function. Double hashing is a technique used to resolve collisions. For example, we may have more than one inputs which generate the same index through the hash function. In double hashing, the hash function takes the following form:

$$h(k,i) \;=\; [\,hash1(key) \;+\; i * hash2(key)\,] \;\%\; m$$

where $hash1$ and $hash2$ are two different hash functions and m is the predetermined size of the hash table. The variable i is used once a collision is detected. The second hash function is only applied if a collision is detected.

1.1 Example

Given the sequence $< 3, 10, 2, 4 >$, the double hashing technique can be applied. Let's assume the size of the table to be $m = 5$ and the two hash functions $h1(k) = k \;\%\; 5, \quad h2(k) = 7k \;\%\; 8$.

$$< NIL, NIL, NIL, NIL, NIL >$$

A table has been initialized with size 5, hence 5 empty slots holding NIL.

First element is inserted - $h(3,i) = h_1(3) = (3 \;\%\; 5) \;\%\; 5 = 3$. Third element of the table is where the element will be inserted.

$$< NIL, NIL, NIL, 3, NIL >$$

Next, we have the following insertion - $h(10,i) = h_1(10) + i * h_2(10) = (10\%5)\%5 = 0$. Element will be inserted in zero-th index.

$$< 10, NIL, NIL, 3, NIL >$$

Next, we have the following insertion - $h(2,i) = h_1(2) + i * h_2(2) = (2\%5)\%5 = 2$. Element will be inserted in index position 2.

$$< 10, NIL, 2, 3, NIL >$$

Next, we have the following insertion - $h(4,i) = h_1(4) + i * h_2(4) = (4\%5)\%5 = 4$. Element will be inserted in index position 4.

$$< 10, NIL, 2, 3, 4 >$$

This is the resulting hash table after the last insertion from the input sequence.

1.2 Implementation

Here we are going to implement a hash table in C++ in which collisions are handled through linear probing. We use the following hash function for generating indices:

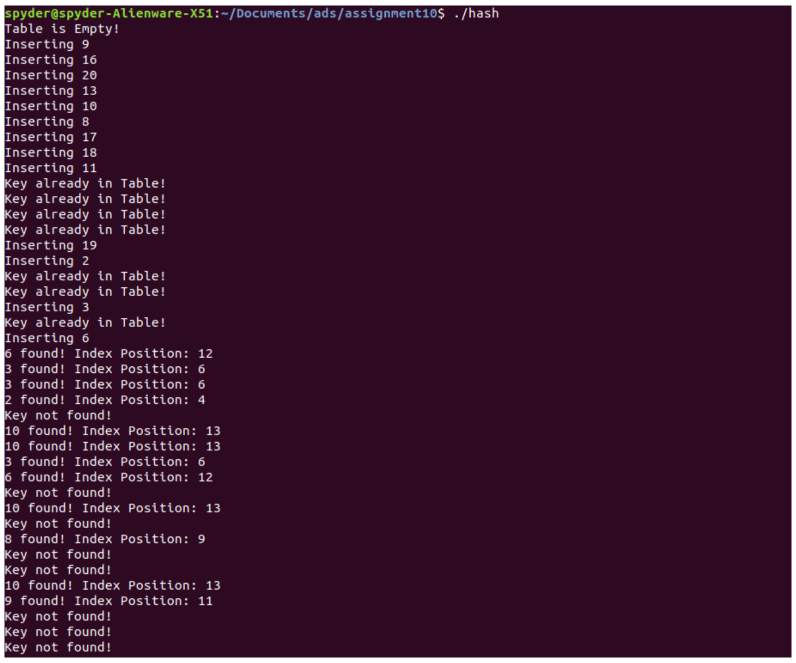
$$h(k,i) = (h'(k)\ +\ i)\ \%\ m$$

where $h'(k)$ is the hash function (assuming no collisions), m is the size of the table and i is the variable which will iterate until the insertion is a success (starting at $i = 0$). As a result, every successive slot will be checked for insertion if a collision is detected. The variable i will be the offset value from the generated hash index. The class declarations of this implementation are illustrated in Figure 1.

```
class Node {
  public:
    int key;
    int value;
    Node(int key, int value);
}
class HashTable {
  private:
    Node **arr;
    int maxSize;
    int currentSize;
  public:
    HashTable();
    hashCode(int key);
    void insertNode(int key, int value);
    int get(int key);
    bool isEmpty();
}
```

Figure 1: Class Declerations for Hash Table Implementation with Linear Probing

The completed class implementation of this hash table can be found in the root directory of this submission labeled $HashTable.h$. The file $testHash.cpp$ is the driver file which tests the implementation. In the driver, we create a hash table, check if it's empty, insert random numbers and then access those inserted keys. The console output logs the execution sequence as can be seen in Figure 2.



```
spyder@spyder-Alienware-X51:~/Documents/ads/assignment10$ ./hash
Table is Empty!
Inserting 9
Inserting 16
Inserting 20
Inserting 13
Inserting 10
Inserting 8
Inserting 17
Inserting 18
Inserting 11
Key already in Table!
Key already in Table!
Key already in Table!
Key already in Table!
Inserting 19
Inserting 2
Key already in Table!
Key already in Table!
Inserting 3
Key already in Table!
Inserting 6
6 found! Index Position: 12
3 found! Index Position: 6
3 found! Index Position: 6
2 found! Index Position: 4
Key not found!
10 found! Index Position: 13
10 found! Index Position: 13
3 found! Index Position: 6
6 found! Index Position: 12
Key not found!
10 found! Index Position: 13
Key not found!
8 found! Index Position: 9
Key not found!
Key not found!
10 found! Index Position: 13
9 found! Index Position: 11
Key not found!
Key not found!
Key not found!
```

Figure 2: Execution of the Driver Program Testing the User Class

2 Greedy Algorithms

2.1 Optimality

Assume we have a greedy algorithm which selects activities with the shortest run time (for maximizing execution efficiency). Suppose we are presented the following activity list:

$$< (3,6) \ , \ (1,4) \ , \ (1,2) \ , \ (6,8) \ , \ (0,2) >$$

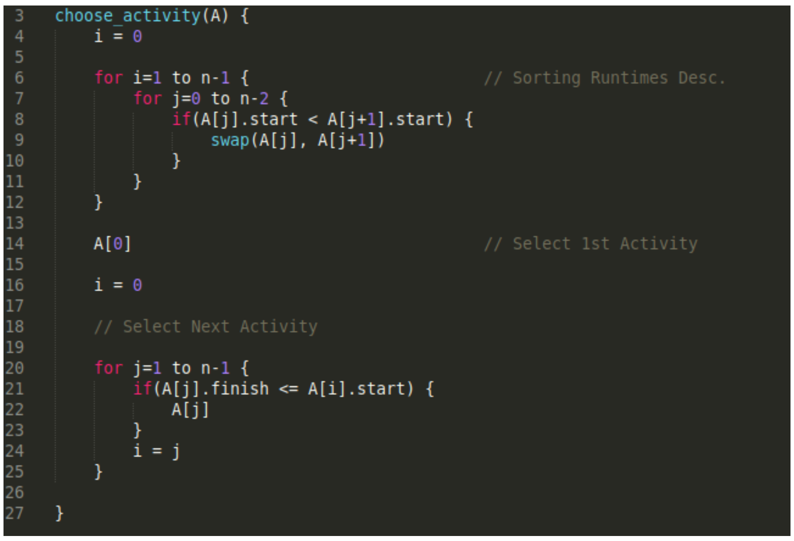
Clearly, the tuple $(1,2)$ is the activity with the shortest run time. The next valid activity is $(6,8)$. This algorithm results in executing two activities. The following activities were, however, also valid options:

$$< (3,6) \ , \ (6,8) \ , \ (0,2) >$$

thus proving, that a greedy algorithm does not always produce a globally optimum solution to a given problem.

2.2 Latest Start Selection

The pseudo code snapshot in Figure 3 shows a rough outline of an algorithm which chooses the activity with the latest starting time.



```
choose_activity(A) {
    i = 0

    for i=1 to n-1 {                        // Sorting Runtimes Desc.
        for j=0 to n-2 {
            if(A[j].start < A[j+1].start) {
                swap(A[j], A[j+1])
            }
        }
    }

    A[0]                                    // Select 1st Activity

    i = 0

    // Select Next Activity

    for j=1 to n-1 {
        if(A[j].finish <= A[i].start) {
            A[j]
        }
        i = j
    }

}
```

Figure 3: Pseudo Code of Proposed Activity Selection Algorithm

This code file is also available in the root directory of this submission labeled *latest_pseudo.h*.