CMPSCI 187 (Spring 2019) Lab 13: Hashing

The goal of this lab is to get familiar with Hashing and Hash Table. To work on the assignment:

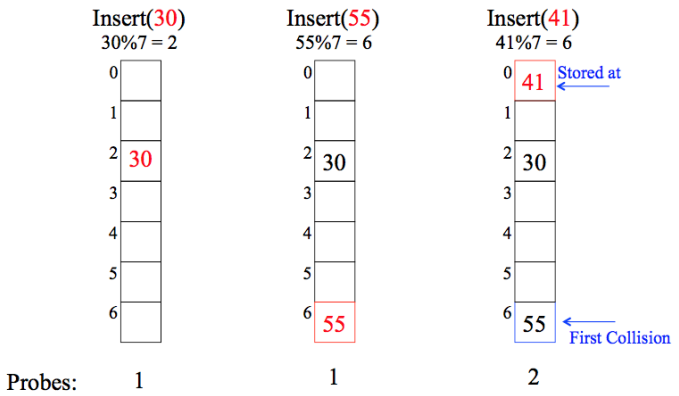
- Go to ***File -> Make a Copy*** to make an editable copy of this Google Doc for your account
- Complete the assignment. When you are done, go to ***File -> Download As -> PDF Document***
- Log in to Gradescope and submit your PDF

The lab is due by 5:00 pm today.

Hashing

Consider the problem of searching for an element in a list. An unsorted array or list requires linear search, taking O(n) time. With a sorted array, we can use binary search, taking O(log n) time, but to maintain elements in sorted order, insertion and deletion would have to take O(n) time.

Hash table is a surprisingly simple data structure that achieves O(1) average cost for all of searching, insertion and deletion. The basic idea of hashing is to take a key (such as student ID, credit card number, SSN etc.), and use a hashing function to convert it to an index, which represents the position of the key in the table. The simplest hashing function is the modulo operator: **index = key % capacity**. Below is an example where the capacity is 7.



Resolving collision: In the linear probing case, it gets stored in the next available free slot.

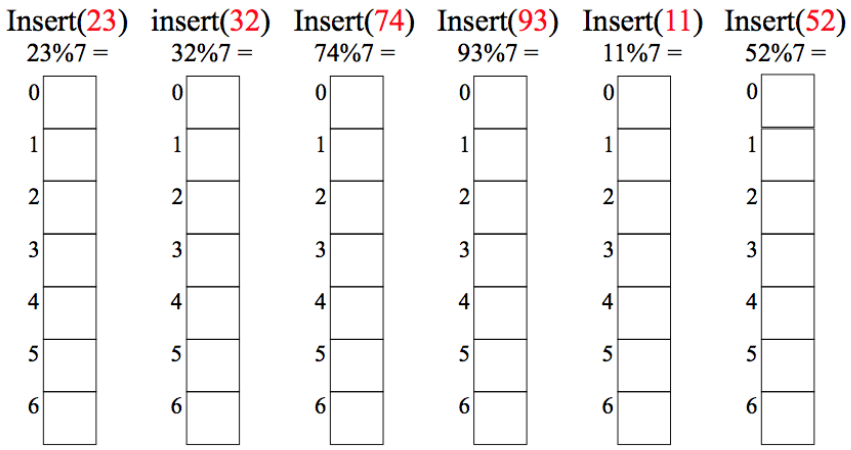
There will certainly be cases when multiple key values may map to the same index, which is known as collision. In the above example, when inserting 41, it maps to index 6, which is already taken by element 55. To resolve collision, there are two common solutions: open addressing and separate chaining. In today's lab we will be focusing on the first solution, and in fact the simplest type of open addressing called **linear probing**.

Specifically, in linear probing, if a collision is encountered, it searches the successive entries in the table until an empty slot is found, and inserts the new key there, as shown in the above example. Also in class you learned about lazy deletion. We don't simply set the deleted element to null, which will break searching. Instead we set a special flag to indicate it's gone. Insertion can overwrite a flagged slot; but search must continue across it.

All these topics have been covered in the lecture slides. **Before proceeding, please review the lecture slides (including code in the slides) to make sure you understand the concepts**. A hashtable is often used to store elements in the form of key-value pairs (e.g. student ID - grades; SSN - wages etc.) A good hashtable design will minimize the likelihood of collision, and even when collisions happen, you won't have to probe very far to find the element (since there are a lot of empty slots in the table). Therefore on average it achieve constant cost, which makes it a very desirable data structure for many practical applications.

Exercise 1: Understand Linear Probing

Complete the following table. Do this exercise on a piece of paper. Compare your answer with TA's solution. This exercise will NOT be graded.



Exercise 2: Programming

Implement the get(), delete(), put() and rehash() methods below. **Starter code with JUnit tests are provided, which you need to import into Eclipse**. Each hash table element is of type **`Pair`** that stores a key-value pair and a delete flag. Note that the key here is a String, and a string can be converted to an Integer value which represents its key value. The starter code already provides the `hash()` function, which calls Java String's hashCode() method to compute the hash key. **So just call the `hash()` function on the key to get the index.**

```
public String get(String key) {  // 3 points
       // Get element value using key and linear probing
       // If the element doesn't exist, return null
  int index = hash(key);
  int start = index;
       while(map[index] != null) {
       if(map[index].getKey() == key)
       return map[index].getValue();
       index = (index+1) % maxSize;
       if(index==start) return null;
       }
  return null;
}
public void put(String key, String value) {  // 4 points
       // Insert a new key-value pair into the table.
       if(isFull()) {rehash();}
       int index = hash(key);
       while(map[index]!=null) {
              index++;
       }
       map[index] = new Pair(key,val);
       currentSize++;

}
```

```
public void rehash() {  // 5 points
	// create a new table twice the size of the old one,
	// then insert all elements from the old hash table to new table.
	maxSize = maxSize*2;
	Pair[] temp = map;
	map = new Pair[maxSize];
	currentSize=0;
	for(int i = 0; i<temp.length; i++) {
		if(temp[i]!=null)
			put(temp[i].getKey(),temp[i].getValue());
		}

}
```

Exercise 3: Runtime Comparison

The test given you to contains `testlinearget()` which checks the runtime performance of hash table vs. unsorted array. It checks insertion time for your hash table vs. unsorted array, and also search time for the two, including the speedup factor. From this test you should be able to verify that the insertion time in hash table is almost as fast as insertion to an unsorted array (both about O(1)). Search is significantly faster in a hash table (O(1)) vs. unsorted array (O(N)). Below please copy and paste the result of this test.

testlinearget() output result

Hash Table: Key, Value
Maine, Augusta
Maryland, Annapolis
Massachusetts, Boston
Florida, Tallahassee
New Jersey, Trenton
Texas, Austin

6
Test inserting 1000 random elements to hash table...Total insertion time: 2, per 1K element cost: 2.0
Test inserting 1000 random elements to unsorted array...Total insertion time: 2, per 1K element cost: 2.0
Search for 1000 elements...
Hash table total search time: 1, Linear search total time: 15
Speedup: 15.0

Test inserting 10000 random elements to hash table...Total insertion time: 4, per 1K element cost: 0.39999998
Test inserting 10000 random elements to unsorted array...Total insertion time: 4, per 1K element cost: 0.39999998
Search for 10000 elements...
Hash table total search time: 3, Linear search total time: 757
Speedup: 252.33333

Test inserting 20000 random elements to hash table...Total insertion time: 6, per 1K element cost: 0.3
Test inserting 20000 random elements to unsorted array...Total insertion time: 6, per 1K element cost: 0.3
Search for 20000 elements...
Hash table total search time: 3, Linear search total time: 2827
Speedup: 942.3333

Exercise 4: Submit Feedback in Course Survey!

http://owl.umass.edu/partners/courseEvalSurvey/uma/

Sign your initials here after completing the survey: _____BM_____.