**Homework 7**

**ECE 309 Fall 2019**

**Due: October 16, 2019**

Upload an electronic copy of your answers to Moodle under HW7.

*This is a shared google document. This means (1) it may change to clarify content, and (2) other people can view your comments on this file. If you have questions, you are encouraged to comment directly on this document, but* ***do not add your answers here****. Make a copy into your private Google Drive and then edit the document.*

*DO NOT ADD ANSWERS TO THE SHARED DOC! THAT’S CONSIDERED CHEATING!*

# 1. Hash Table

[45 points] Assume we have a hash table for integers that allows duplicates and that a remove operation will remove all copies. Consider the following sequence of operations on a hash table for integers.

* Insert 14
* Insert 714
* Insert 97
* Insert 25
* Insert 6736
* Insert 103
* Insert 3
* Remove 14
* Remove 714
* Search 25

1. [15 points] Assume the hash table has length 10 and that the hash function is simply index = data % 10. Show a picture of the hash table after executing the operations above assuming Linear Probing is used on collisions. Assume a probeDistance = 5. If there’s no place for an item within the probeDistance, then state that, but do not evict or change the size of the table.
2. [15 points] Assume the hash table has length 10 and that the hash function is simply index = data % 10. Show a picture of the hash table after executing the operations above assuming chaining is used.
3. [5 points] For the two hash table designs, which one performs fewer table accesses and comparisons when performing the last search for 25.
4. [10 points] Consider that the size of a hash table array is M entries and the number of items inserted in the set is N. Evaluate the big-O for the amount of memory used for a hash table with linear probing versus a hash table with chaining. State any assumptions in the formation of your answer.

# 

# 

# 

# 

# 2. ZyLabs

1. [10 points - test cases passed] **ZyLab 15.10**. Modify the insert function of the chaining based hash table to avoid inserting duplicates.
2. [15 points - test cases passed] **ZyLab 15.11**. Implement a resize function for a chaining based hash table. Resizing is the process of increasing a hash table’s size when too many items have been added, reducing the hash table’s effectiveness.

We’ll add a field called maxChainLength to the object. This will be set by the constructor and will indicate the maximum allowable chain length. If any bucket ends up with a larger chain than maxChainLength after an insertion, the table must be resized.

To resize the hash table, increase the size of the table by some small factor, remove everything from the old table, and insert it into the new larger table. Make sure to free up memory that is no longer used after resizing. Implement the resize() function as a private member function. Follow any other specific guidelines or constraints specified in the ZyLab.

Warning: you may need to experiment to find a good factor to enlarge the table by. Don’t choose a larger value than necessary.

1. [30 points] **ZyLab 15.12**. Implement a hash table that supports iteration over its elements. You must add an iterator object to the hash table to support it.

When an object is inserted into the hash table, the iterator must be able to see it and include it in a traversal. However, when an element is removed, it’s no longer part of the traversal.

|  |
| --- |
| HashTableChain myset(10);  int sz = myset.getTableSize();    std::cout << "Hash table is " << myset.getTableSize()  << " entries long." << std::endl;  for(int i=0; i<40; i++) {  myset.insert(i);  }  std::cout << "Hash table has " << myset.getNumElements()  << " elements." << std::endl;  **// Use iterator**  **std::cout << "Hash table contains: ";**  **for(auto it = myset.begin(); !it.end(); it.next())**  **{**  **// Visit each element in hash table**  **std::cout << it.getItem() << ", ";**  **}**  **std::cout << std::endl;** |

After the addition of the iterator, the hash table must retain insertion, removal, and find operations in O(1) time. Also, the iterator must be able to traverse all elements in the hash table in O(N) time, where N is the number of items in the set. It must not be O(M), which is the number of entries in the hash table. In other words, you may not loop over the table array to find all of the items.

One way to achieve this design is by inserting items into a linked list at the same time they are inserted into the hash table. Insertion can be done efficiently since linked lists can be appended in O(1) time. Removal, on the other hand, is not as efficient, because it would require O(N) time to traverse the list to find the item and remove it. However, if in the hash table alongside the data, you store a pointer to the item’s node in the linked list, you can use the pointer to find the item quickly and remove it from the list. For example:

class Element {

public:

int data;

iterator it; // point to item in list, useful for removal later

};

List<Element> \*table; // array of lists of Element

Other strategies are also acceptable for achieving the same result, as long as they meet all the requirements specified above. Follow any other specific guidelines or constraints specified in the ZyLab.

Grading is as follows:

* [10 points] Achieving proper big-O comlexity in your design.
* [20 points] Test cases passed in ZyLabs.