# CptS 223 Homework #3

Please complete the homework problems on the following page using a separate piece of paper. Note that this is an individual assignment and all work must be your own. Be sure to show your work when appropriate. This assignment is due end of day on Monday March 27th.

1. [6] Starting with an empty hash table with a fixed size of 11, insert the following keys in order into four distinct hash tables (one for each collision mechanism): {12, 9, 1, 0, 42, 98, 70, 3}. You are only required to show the final result of each hash table. In the **very likely** event that a collision resolution mechanism is unable to successfully resolve, simply record the state of the last successful insert and note that collision resolution failed. For each hashtable type, compute the hash as follows:

hash(x) = (x \* x + 3) % 11

### Separate Chaining (buckets)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 |  | 0 | 98, 1, 12 |  |  | 42, 9 | 70 |  |  |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |

### Linear Probing; probe(i) = (i + 1) %11

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 |  | 0 | 12 | 1 |  | 9 | 42 | 70 |  |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |

Failed to insert: 98

### Quadratic Probing probe(i) = (i \* i + 5) % 11

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 |  | 0 | 12 |  |  | 9 | 70 |  | 1 |
| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |

Failed to insert: 42, 98

2. [3] For implementing a hash table. Which of these would probably be the best initial table size to pick?

Table Sizes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 100 | 101 | 15 | 500 |

Why?

101, because it is a prime number. 1 wouldn’t work because all the values would hash to the same bucket. The rest are all divisible by 5.

3. [4] For our running hash table, you’ll need to decide if you need to rehash. You just inserted a new item into the table, bringing your data count up to 53491 entries. The table’s vector is currently sized at 106963 buckets.

* Calculate the load factor (λ):

53491/106963 = .500088816

* Given a linear probing collision function should we rehash? Why?

Yup, load factor is greater than .5

* Given a separate chaining collision function should we rehash? Why?

Nope, the load factor would have to be greater then 1.0

4. [4] What is the Big-O of these actions for a well designed and properly loaded hash table with N elements?

|  |  |
| --- | --- |
| Function | Big-O complexity |
| Insert(x) | O(1) |
| Rehash() | O(N) |
| Remove(x) | O(1) |
| Contains(x) | O(1) |

5. [3] If your hash table is made in C++11 with a vector for the table, has integers for the keys, uses linear probing for collision resolution and only holds strings… would we need to implement the Big Five for our class? Why or why not?

No, because these are all written within std::vector. Just use their functions.

6. [6] Enter a reasonable hash function to calculate a hash key for these function prototypes:

int hashit( int key, int TS )  
{

return ( key % TS );

}  
  
  
  
int hashit( string key, int TS )  
{

int hash;

for ( char ch:key )

hash += ch;

return ( hash % TS );

}

7. [3] I grabbed some code from the Internet for my linear probing based hash table because the Internet’s always right. The hash table works, but once I put more than a few thousand entries, the whole thing starts to slow down. Searches, inserts, and contains calls start taking \*way\* longer than O(1) time and my boss is pissed because it’s slowing down the whole application services backend I’m in charge of. I think the bug’s in my rehash code, but I’m not sure where. Any ideas why my hash table starts to suck as it grows bigger?  
  
/\*\*  
\* Rehashing for linear probing hash table.  
\*/

void rehash( )  
{  
 vector<HashEntry> oldArray = array;

// Create new double-sized, empty table  
 array.resize( 2 \* oldArray.size( ) );  
 for( auto & entry : array )  
 entry.info = EMPTY;

// Copy table over  
 currentSize = 0;  
 for( auto & entry : oldArray )  
 if( entry.info == ACTIVE )  
 insert( std::move( entry.element ) );  
}

When the hash table resizes, it simply doubles. What you should do is to double it and then find the next largest prime number. Prime numbers will be more efficient for hash tables.

8. [4] Time for some heaping fun! What’s the time complexity for these functions in a binary heap of size N?

|  |  |
| --- | --- |
| Function | Big-O complexity |
| insert(x) | O(logN) |
| findMin() | O(1) |
| deleteMin() | O(logN) |
| buildHeap( vector<int>{1...N} ) | O(N) |

9. [4] What would a good application be for a priority queue (a binary heap)? Describe it in at least a paragraph of why it’s a good choice for your example situation.

A print job application. You can prioritize the printer so that fast jobs and/or important jobs print first before others. If the boss wants to print out something, he/she should have priority over the printer. If a worker has something important to print that needs to be done now for a business meeting, he/she can prioritize themselves first. This would be a better system than having everyone wait. The print jobs with a higher priority should print first.

10. [4] For an entry in our heap located at position i, where are its parent and children?

Parent: i / 2 (assuming heap starts at position 1 and not position 0)

Children: i \* 2 and (i \* 2) + 1 (assuming the same thing)

What if it’s a d-heap?

Parent: i / d (assuming above)

Children: i \* d and (i \* d) + 1 (assuming above)

11. [6] Show the result of inserting 10, 12, 1, 14, 6, 5, 15, 3, and 11, one at a time, into an initially empty binary heap. Use a 1-based array like the book does. After insert(10):

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 |  |  |  |  |  |  |  |  |  |

After insert (12):

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 12 |  |  |  |  |  |  |  |  |

etc:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 12 | 10 |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 12 | 10 | 14 |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 6 | 10 | 14 | 12 |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 6 | 5 | 14 | 12 | 10 |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 6 | 5 | 14 | 12 | 10 | 15 |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 3 | 5 | 6 | 12 | 10 | 15 | 14 |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 3 | 5 | 6 | 12 | 10 | 15 | 14 | 11 |  |

12. [4] Show the same result (only the final result) of calling buildHeap() on the same vector of values: {10, 12, 1, 14, 6, 5, 15, 3, 11}

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 3 | 5 | 6 | 12 | 10 | 15 | 14 | 11 |  |

13. [4] Now show the result of three successive deleteMin operations from the prior heap:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 | 6 | 5 | 11 | 12 | 10 | 15 | 14 |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 5 | 6 | 10 | 11 | 12 | 15 | 14 |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 6 | 11 | 10 | 12 | 15 | 14 |  |  |  |  |