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

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

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

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

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

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 would be the best to pick since it is recommended that hash tables should be prime to avoid collisions.

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 = 0.5

- Given a linear probing collision function should we rehash? Why?
  Yes, the time to load is very high at 0.5 so it would be best to rehash.

- Given a separate chaining collision function should we rehash? Why?
  No, for separate chaining its best to rehash at a factor of 3 so 0.5 is very low for this situation.

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(n)
Rehash()	O(n^2)
Remove(x)	O(n)
Contains(x)	O(n)

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?
Yes, how else would we be able to perform those operations if we didn't implement the big five for our class.

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

```
int hashit( int key, int TS )
{
        int num = key * key + 3;
        return num % TS;

}



int hashit( string key, int TS )
{

         int length = key.length();
         for (sum = 0, I = 0; i < length; i++)
        {
            sum += key[i];
        }
        return sum % 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 ) );
}
```

Your function increases in size but you didn't implement a NextPrime() function of some variation into your rehashing.

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 (log n)
findMin()	O(1)
deleteMin()	O(1)
buildHeap(vector<int>{1...N})	O(log 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.

I just hacked into Google's database and I'm downloading their data. It's a ton of data so it takes a while, so it's placed into a queue. Once one chunk of data finishes downloading the next starts downloading. Then I stumble upon some really juicy data as I'm looking through Google's database and I want it right now. But it is last in my Queue so it may take hours or even days to download it. Luckily my queue is a priority queue so I set the key of this data that I want right now, so that it is first in line to be downloaded. In conclusion priority queues are good if there's a long wait but you need something immediately.

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

    Parent:
    floor(i/2)

    Children:
    Left: i*2
    Right: i*2+1

    What if it's a d-heap?
    Parent:
    Floor(i/d)

    Children:
    i*d
    i*d+1
    i*d+2
    i*d+…….+(d-1)

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	11	14	

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

	3	5	10	6	12	14	15	11		

	5	6	11	10	12	14	15			

	6	10	14	11	12	15				