CptS 223 Homework #3 - Heaps, Hashing, Sorting

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. Please scan the assignment and upload the PDF to Git.

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:

hashkey(key) = (key * key + 3) % 11

Separate Chaining (buckets)

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

To probe on a collision, start at hashkey(key) and add the current probe(i') offset. If that bucket is full, increment i until you find an empty bucket.

Linear Probing: probe(i') = (i + 1) % TableSize

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

Quadratic Probing: probe(i') = (i * i + 5) % TableSize

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

70 collision resolution unsuccessful

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

Table Sizes:

1 100 $\boxed{101}$ 15 500

Why did you choose that one?

it is the largest prime number

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):

.50009

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

yes, λ > .5, good time to rehash for linear probe

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

nah, separate chaning can handle up to ~.75 load factor

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 since everything is built in and no moves or copies are being made

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 hashval = 0;
    for (char ch: key){
        hashval = 37* hashval + ch
    }
    return hashval % TS
}
```

7. [3] I grabbed some code from the Internet for my linear probing based hash table at work because the Internet's always right (totally!). 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 *much* 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 is 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 ) );
}
```

never destroyed the old HT & not getting a prime number length on new HT length

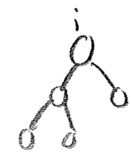
8. [4] Time for some heaping fun! What's the time complexity for these functions in a C++ STL binary heap of size N?

Function	Big-O complexity
push(x)	O(log N)
top()	O(1)
pop()	O(log N)
buildHeap(vector<int>{1...N})	O(N log N)

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

great for shortest Job first scheduling or merging lists together since they pop() off elements in order from smallest → largest or vice versa.

10. [4] For an entry in our heap (root @ index 1) located at position i, where are it's parent and children?



Parent: $i/2$

Children: $i \cdot 2$, $i \cdot 2 + 1$

What if it's a d-heap?

Parent: i/d

Children: $i \cdot d + k \bmod d$

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	11	6	10	15	14	12		

13. [4] Now show the result of three successive deleteMin / pop operations

from the prior heap:

3	6	5	11	12	10	15	14			

5	6	10	11	12	14	15				

6	11	10	15	12	14					

14. [4] What are the average complexities and the stability of these sorting algorithms:

Algorithm	Average complexity	Stable (yes/no)?
Bubble Sort	$O(n^2)$	yes
Insertion Sort	$O(n^2)$	yes
Heap sort	$O(n \log n)$	no
Merge Sort	$O(n \log n)$	yes
Radix sort	$O(cn)$	yes
Quicksort	$O(n \log n)$	no

15. [3] What are the key differences between Mergesort and Quicksort? How does this influence why languages choose one over the other?

Mergesort: lots of moves, low comparisons

quicksort: lots of comparisons, low move

Java: slow comparisons, fast moves → Mergesort

C++: fast comparisons, copying is expensive → quicksort

16. [4] Draw out how Mergesort would sort this list:

24	16	9	10	8	7	20

16	24					
16	24	9	10	8	7	20
		9	10			
16	24	9	10	8	7	20
9	10	16	24			
9	10	16	24	8	7	20
				7	8	
9	10	16	24	7	8	20
7	8	9	10	16	20	24

17. [4] Draw how Quicksort would sort this list:

24	16	9	10	8	7	20

20	16	9	10	8	7	24 ✓
7	16	9	10	8	20 ✓	24 ✓
7 ✓	16	9	10	8	20 ✓	24 ✓
7 ✓	8 ✓	9	10	16	20 ✓	24 ✓
7 ✓	8 ✓	9 ✓	10	16	20 ✓	24 ✓
7 ✓	8 ✓	9 ✓	10 ✓	16	20 ✓	24 ✓
7 ✓	8 ✓	9 ✓	10 ✓	16 ✓	20 ✓	24 ✓

Let me know what your pivot picking algorithm is (if it’s not obvious):

pivot = pivot + 1