CptS 223 Homework #3 - Heaps, Hashing, Sorting

Due Date: Nov 20th 2020

Please complete the homework problems and upload a pdf of the solutions to blackboard 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 three 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)

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

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

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

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

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

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 did you choose that one?

101, because it is a prime number and this helps avoid clustering of values

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): 0.50009

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

  Yes, because 50% of the time we are going to spend probing. This will slow down the efficiency of the data structure

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

  No, because collisions will just add to a linked list

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)

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( )
{
    ArrayList<HashItem<T>> oldArray = array;

    array = new ArrayList<HashItem<T>>( 2 * oldArray.size() );

    for( int i = 0; i < array.size(); i++ )
        array.get(i).info = EMPTY;
// Copy old table over to new larger array
    for( int i = 0; i < oldArray.size(); i++ ) {
        if( oldArray.get(i).info == FULL )
        {
            addElement(oldArray.get(i).getKey(),
                       oldArray.get(i).getValue());
        }
    }
}
```

You are using two for loops when only one is necessary.

8. [4] Time for some heaping fun! What's the time complexity for these functions in a Java Library priority queue (binary heap) of size N?

Function	Big-O complexity
push(x)	O(1)
top()	O(1)
pop()	O(logn)
PriorityQueue(Collection<? extends E> c) // BuildHeap	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.

Good applications of priority queues are where you need fast access for either the smallest or largest item. That specific value will always be in the first element of the queue (root of the tree). Insertions are also fast, so its good for incoming data at the cost of being partially unsorted. A good application would probably be an emergency room queue where severely injured patients are set on high priority in the queue and less urgent patients are placed into the queue at low priority. These patients are coming in at random times with random levels of priority, so it make sense to use a priority queue. Queue insertion is fast, and access to the highest priority is fast.

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

Parent:

index / 2

Children:

right = index*2 +1
left = index*2

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(nlogn)	no
Merge Sort	O(nlogn)	yes
Radix sort	O(nk)	yes
Quicksort	O(nlogn)	no

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

The worst case in merge sort is O(n^2), where mergesort is O(nlogn).

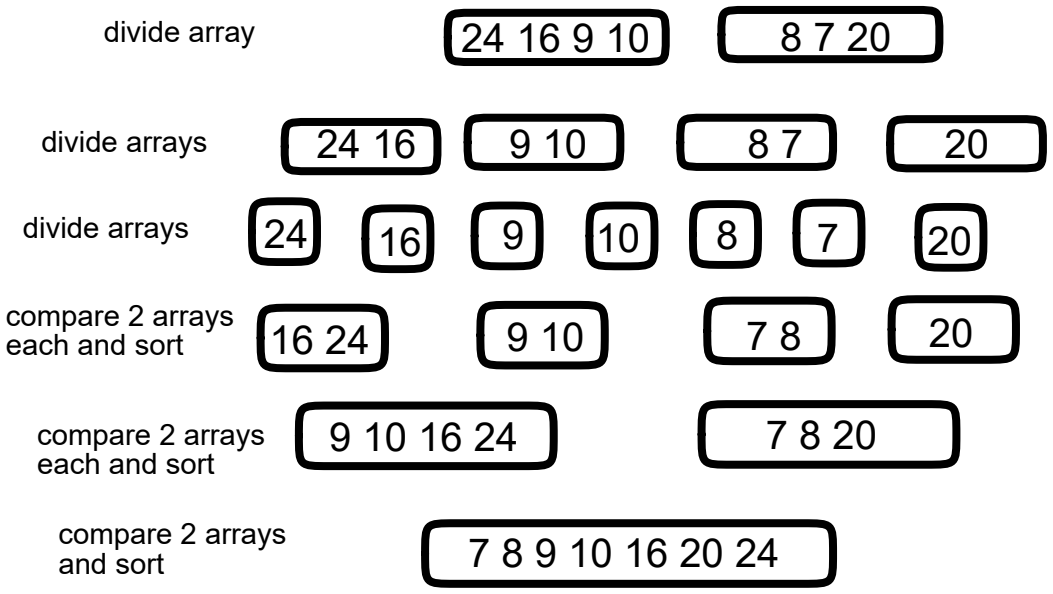
With quicksort using partitioning it works better on smaller arrays, while mergesort is efficient on any array size.

Quicksort is unstable with two equal element values, mergesort is stabe in this scenario.

This may be the influencing factor when using languages that work with big data.
Mergesort would be the favor in this scenario.

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

24	16	9	10	8	7	20



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

24	16	9	10	8	7	20

swap pivot to the end of the array

24 16 9 20 8 7 10

swap 7, 24

7 16 9 20 8 24 10

swap 16,8

7 8 9 20 16 24 10

swap pivot

7 8 9 10 16 24 20

quicksort left sublist(a swap between 8/9 occurs twice, but essentially nothing happens)

quicksort right sub list, swap 24,20

7 8 9 10 16 20 24

swap 16,20 and then swap back

7 8 9 10 16 20 24

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

pivot = pick center and swap with end value