Git link: https://github.com/TorNATO-PRO/DataStructuresHW.git

Nathanael Waltz

04 November 2020

Computer Science 223 Homework #3

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 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 hash-table type, compute the hash as follows: hash-key(key) = (key * key + 3) % 11

Key	12	9	1	0	42	98	70	3
Hash-key	4	7	4	3	7	4	8	1

Separate Chaining (buckets)

Note: ↑ is a pointer

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

To probe on a collision, start at hash-key(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) % Table-size

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

Increment	0	1	2	3	4	5	6	7
Probe offset	5	6	9	3	10	8	8	10

Quadratic Probing: probe(i') = (i * i + 5) % Table-size

Value		42		0	12		3	9	70	1	98
Index	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 did you choose that one?

   In choosing a hash table size, there are two things to consider. 1 – The hash table size should be a prime number. 2 – It should be around 1.3 times the maximum number of keys. Since there is not a given number of keys, I would say 101 as it is a prime number, unlike 15 which would be my next choice. Also, 101 is a happy medium and it is not potentially wasting space like 500 is (and 500 isn't prime).

3. [4] For our running hash table, you will need to decide if you need to rehash. You just inserted a new item into the table, bringing your data count to 53491 entries. The table's vector is currently sized at 106963 buckets.
   - Calculate the load factor (λ):

   $$\lambda = \frac{elements}{buckets} = \frac{53491}{106963} \approx 0.500089$$

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

     We should indeed rehash, as performance quickly degrades for a load factor greater that 0.5, which is currently the case with our load factor.

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

     We should not rehash as the load factor is not really at a high enough value to consider hashing given that it is a separate chaining collision function (i.e. > 1, or maybe > 0.75).

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 (well designed)	Big-O Complexity (average design)
Insert(x)	$O(1)$	O(n)
Re-hash()	$O(n)$	O(n)
Remove(x)	$O(1)$	O(n)
Contains(x)	$O(1)$	O(n)

O(1) Assuming its perfectly designed (see problem description) and nice load factor

5. [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 am not sure where. Any ideas why my hash table starts to suck as it grows bigger?

```
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;
    }

    /*
        The problem with this code is as follows:

        See, you are reusing the same hash value, which will not really help
        things much since values will be clustered in one place in the array
        when they have a new hashtable, which has another size. Since these keys
        are all some value modulo tablesize, they will all be placed in the same
area.
        This will serve the dual purpose of taking up more memory and providing n
o
        performance benefits. That's not a good day!!!
    */

    for (int i = 0; i < oldArray.size(); i++) {
        if (oldArray.get(i).info == FULL) {
            addElement(oldArray.get(i).getKey(), oldArray.get(i).getValue());
        }
    }

}
```

6. [4] Time for some heaping fun! What is the time complexity for these functions in a Java Library priority queue (binary heap) of size N?

Function	**Big-O Complexity**
push(x)	O(log(n))
top() - ?	O(1)
pop()	O(1)
Buildheap()	O(nlog(n))

7. [4] What would a good application be for a priority queue (a binary heap)? Describe it in at least a paragraph of why it is a good choice for your example situation.

Binary Heaps have a good Big-O time complexity but are slower than merge sort or quick sort. However, there is still a use case. Consider for a second the case in which you have objects with a specific priority denoted some number flowing in, maybe we can use the analogy of an NSA hacker dude that is going through our data. Now, if you are acting like your average joe, the need to put you under greater surveillance is low, so you go to the bottom of the list. However, if you are hacking stuff or trafficking drugs or any other fun pastime, then when your data goes into this priority queue, you go to the top of the queue, and thus are a high priority to be surveilled. A priority queue is good in this case as you never know what someone is going to do, so you cannot immediately sort it like you can a list. (If you could, then you could prevent crimes before they happen).

8. [4] For an entry in our heap (root @ index 1) located at position i, where are its parent and children?

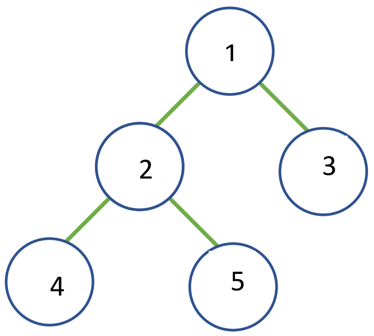
Parent:

$$\lfloor i/2 \rfloor = Index_{parent}$$

Children:

$$2i + 1 = Index_{Left\ Child}$$

$$2i + 2 = Index_{Right\ Child}$$



9. [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):
Max-Heap

	10								

	12	10							

	12	10	1						

	14	12	1	10					

	14	12	1	10	6				

	14	12	5	10	6	1			

	15	12	14	10	6	1	5		

	15	12	14	10	6	1	5	3	

	15	12	14	11	6	1	5	3	10

10. [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}

	15	12	14	11	6	1	5	3	10

11. [4] Now show the result of three successive deleteMin / pop operations from the prior heap:

	14	12	10	11	6	1	5	3	

	12	11	10	3	6	1	5		

	11	6	10	3	5	1			

12. [4] What are the average complexities and the stability of these sorting algorithms:

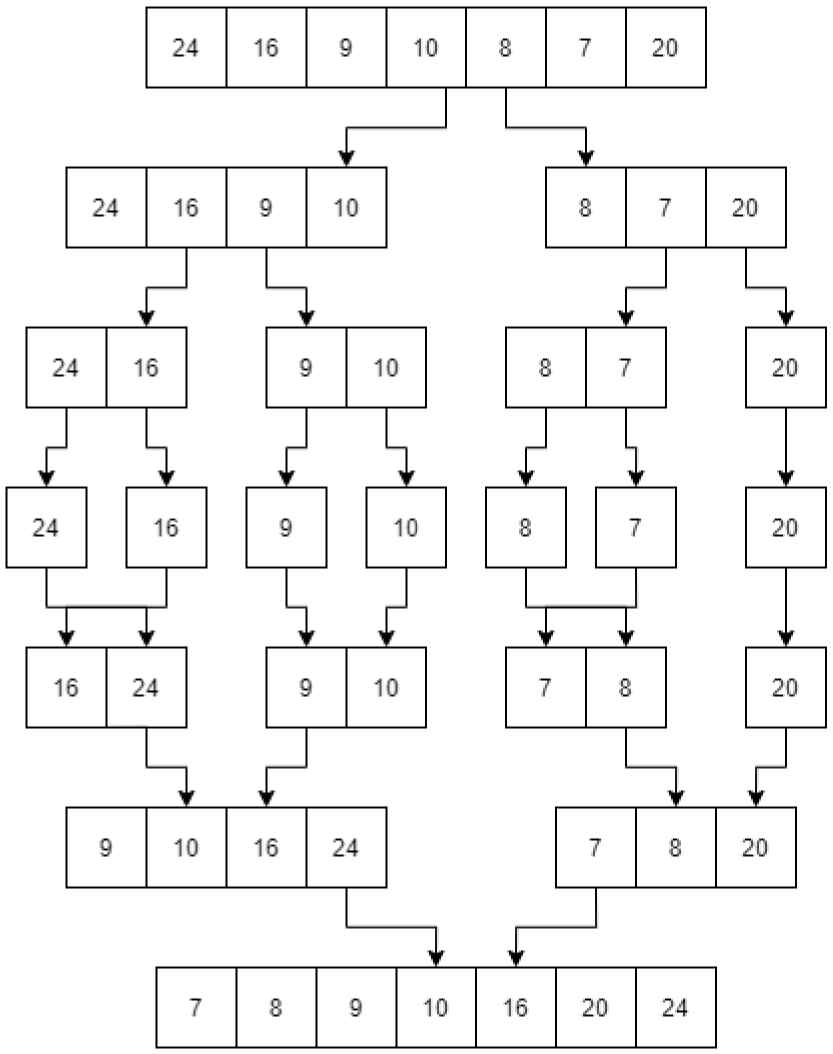
Algorithm	Average Complexity	Stable (yes/no)?
Bubble Sort	$O(n^2)$	Yes
Insertion Sort	$O(n^2)$	Yes
Heap Sort	$O(nlog(n))$	No
Merge Sort`	$O(nlog(n))$	Yes
Radix Sort	$O(nk)$	Yes
Quick Sort	$O(nlog(n))$	No

13. [3] What are the key differences between Mergesort and Quicksort? How does this influence why languages choose one over the other?

Merge sort is not a stable sort however it uses more memory than quick sort and thus when memory is a factor rather than the order of equivalent elements, quick sort is the wiser choice. However, if memory is not a major limiting factor and you want equivalent elements ordered in their original order, merge sort is the better choice.

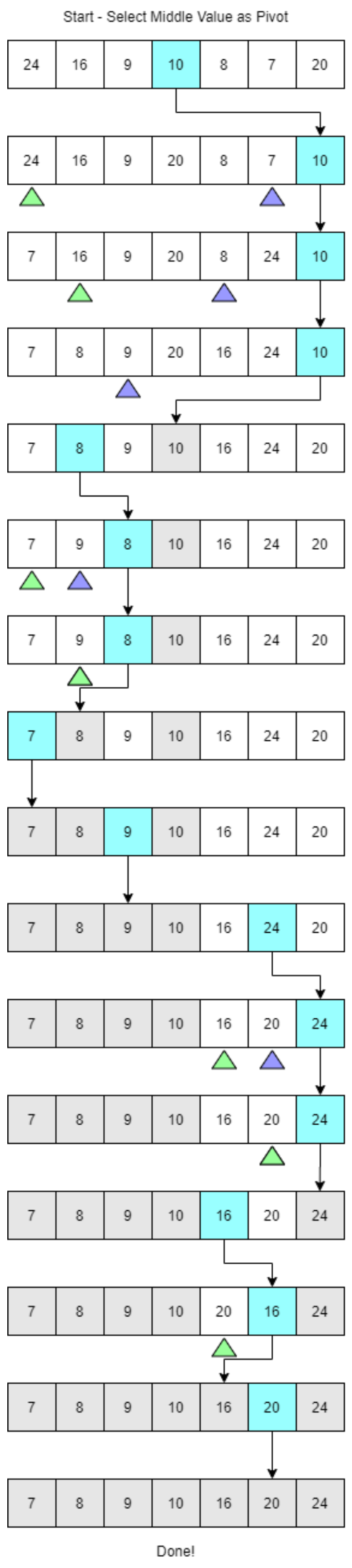
14. [4] Draw out how Mergesort would sort this list:

24	16	9	10	8	7	20



15. [4] Draw how Quicksort would sort this list:

24	16	9	10	8	7	20


Start - Select Middle Value as Pivot
24 16 9 10 8 7 20
24 16 9 20 8 7 10
7 16 9 20 8 24 10
7 8 9 20 16 24 10
7 8 9 10 16 24 20
7 9 8 10 16 24 20
7 9 8 10 16 24 20
7 8 9 10 16 24 20
7 8 9 10 16 24 20
7 8 9 10 16 24 20
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 20 16 24
7 8 9 10 16 20 24
7 8 9 10 16 20 24
Done!