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.

work on following page for first problem on this page!

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)

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

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

	3		0	12			9	70	1	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 $\boxed{500}$

Why did you choose that one?

I chose 500 as the best initial table size to pick when implementing a hash table because we don't know the total entries in the hash table. Therefore, choosing the maximum size available will be the best choice.

$\boxed{1}$ {12, 9, 1, 0, 42, 98, 70, 3}

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

Separate Chaining

h(12) = (12 * 12 + 3) % 11 = 4
h(9) = (9 * 9 + 3) % 11 = 7
h(1) = (1 * 1 + 3) % 11 = 4 collision → chaining
h(0) = (0 * 0 + 3) % 11 = 3
h(42) = (42 * 42 + 3) % 11 = 7 collision → chaining
h(98) = (98 * 98 + 3) % 11 = 4 collision → chaining
h(70) = (70 * 70 + 3) % 11 = 8
h(3) = (3 * 3 + 3) % 11 = 1

probe(i') = (i + 1) % $\overbrace{\text{TableSize}}^{11}$

Linear Probing

for 1: p(0) = (0 + 1) % 11 = 1 + h(1) = 1 + 4 = 5
for 42: p(0) = (0 + 1) % 11 = 1 + h(42) = 1 + 7 = 8
p(1) = (1 + 1) % 11 = 2 + h(42) = 2 + 7 = 9
for 98: p(0) = (0 + 1) % 11 = 1 + h(98) = 1 + 4 = 5
p(1) = (1 + 1) % 11 = 2 + h(98) = 2 + 4 = 6

probe(i') = (i * i + 5) % $\overbrace{\text{TableSize}}^{11}$

Quadratic Probing

for 1: p(0) = (0 * 0 + 5) % 11 = 5 + h(1) = 5 + 4 = 9
for 42: p(0) = (0 * 0 + 5) % 11 = 5 + h(42) = 5 + 7 = 12
p(1) = (1 * 1 + 5) % 11 = 6 + h(42) = 6 + 7 = 13
p(2) = (2 * 2 + 5) % 11 = 9 + h(42) = 9 + 7 = 16
p(3) = (3 * 3 + 5) % 11 = 3 + h(42) = 3 + 7 = 10
for 98: p(0) = (0 * 0 + 5) % 11 = 5 + h(98) = 5 + 4 = 9
p(1) = (1 * 1 + 5) % 11 = 6 + h(98) = 6 + 4 = 10
p(2) = (2 * 2 + 5) % 11 = 9 + h(98) = 9 + 4 = 13
p(3) = (3 * 3 + 5) % 11 = 3 + h(98) = 3 + 4 = 7
p(4) = (4 * 4 + 5) % 11 = 10 + h(98) = 10 + 4 = 14
p(4) = (5 * 5 + 5) % 11 = 8 + h(98) = 8 + 4 = 12

↳ cannot be resolved... will rotate through same 6 indices over & over again...

↳ full pattern of repetition: 9 10 13 7 14 12 12 14 7 13 10

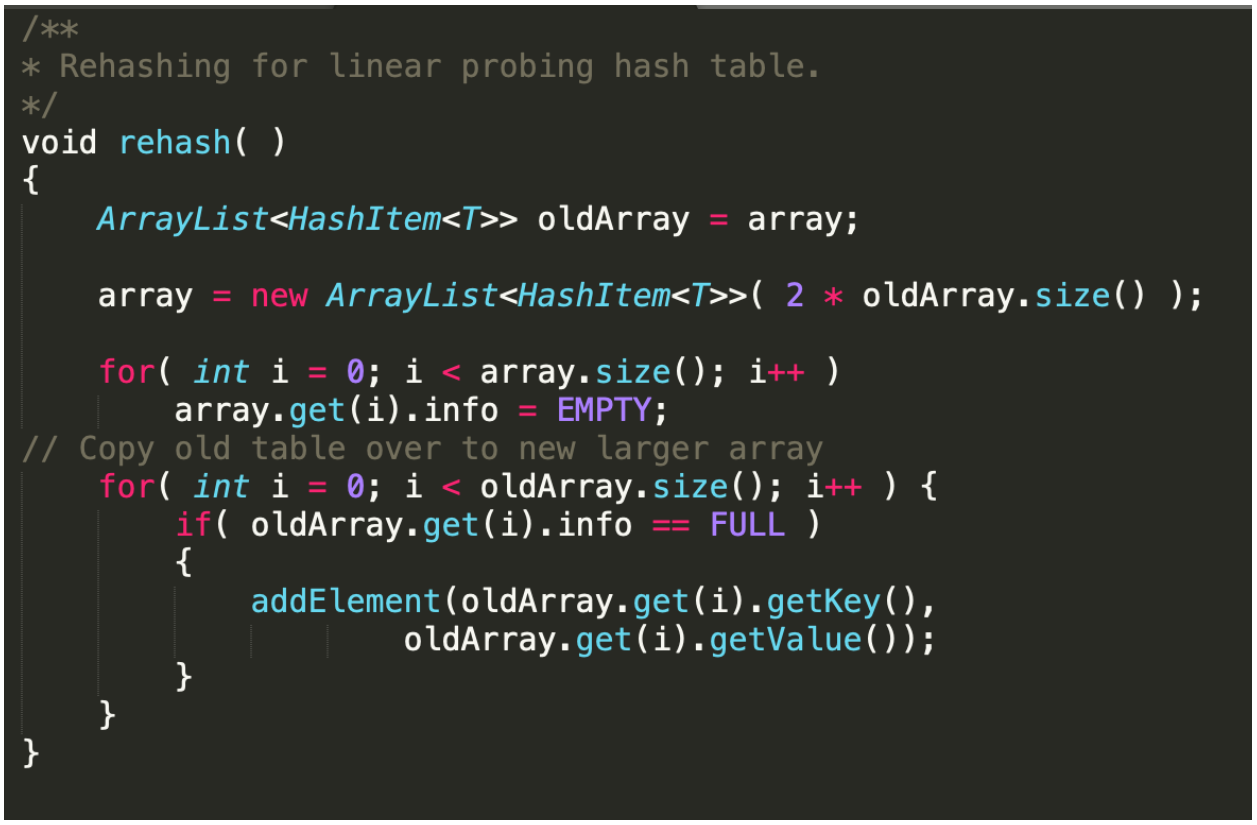
3. [4] For our running hash table, you'll need to decide if you need to rehash. You just inserted a new item into the table, bringing your data count up to 53491 entries. The table's vector is currently sized at 106963 buckets.

- Calculate the load factor (λ):
  Load factor (lambda) = elements in hash table / table size
  = 53491 entries / 106963 buckets
  = 0.50 (approx.)
- Given a linear probing collision function should we rehash? Why?
  Given a linear probing collision function, we should rehash as the load factor is now below 0.5.
- Given a separate chaining collision function should we rehash? Why?
  Given a separate chaining collision function, we should not rehash as the load factor is less than 1.

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 (Assuming best case)
Insert(x)	O(1)
Rehash()	O(1)
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());
        }
    }
}
```

The hash table starts to suck as it grows bigger because the issue lies in the first line of the code:

ArrayList<HashItem<T>> oldArray = array;

That line essentially makes a copy of "array" when array is already overridden in the subsequent line, and nothing changes to "array" between the first and second line. Thus, especially when the hash table gets filled and is eventually rehashed, a lot of unnecessary memory is used, making it a very expensive line of code (memory-wise).

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(log N)
top()	O(1)
pop()	O(log N)
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.

A good application for a priority queue (a binary heap) would be in operating systems design. Let's take for example an operating system scheduler in an multiuser environment. The scheduler must decide which of several processes to run, where generally, one process is allowed to run for only a fixed period of time. Using a queue, jobs will be initially placed at the end of the queue, and the scheduler will take and run the first job on the queue. It will run until it either finishes the job, or its time limit is up and place the job at the end of the queue. Furthermore, in order to implement a precedence or priority to certain jobs so that both short and important jobs finish as quick as possible, a priority queue would need to be applied.

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

Parent:

The parent would be located at index floor(i/2).

Children:

The left child would be located at index 2*i.
The right child would be located at index 2*i+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): Assuming min heap...

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	12	6	10	15	14	11		

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(nk)	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?

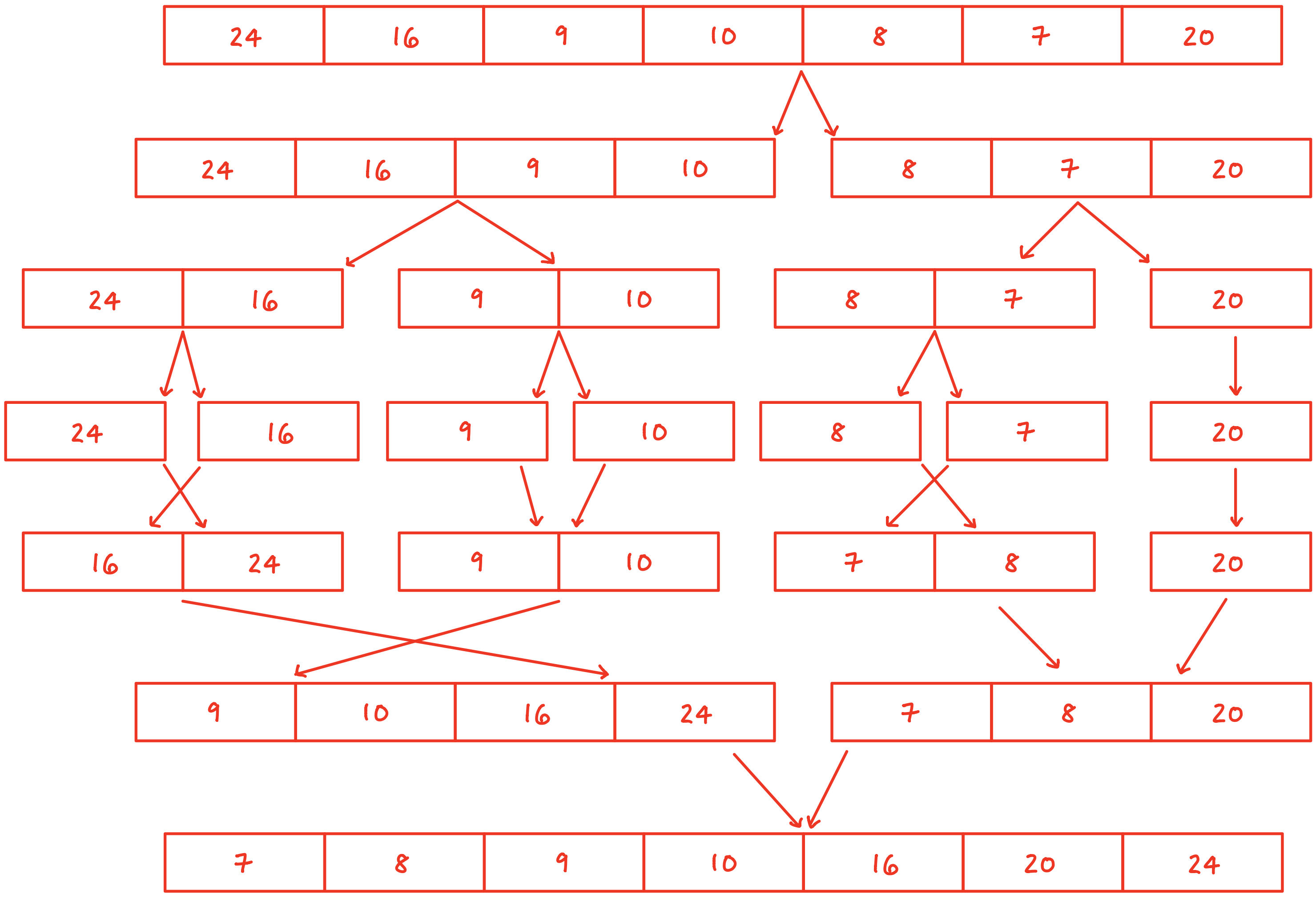
The key differences between Mergesort and Quicksort includes:

- Quicksort is not a stable sorting algorithm as it does not guarantee that the relative order of elements with equal values is preserved.
- Quicksort does no require any additional memory while sorting while Mergesort does (in the order of O(n)).

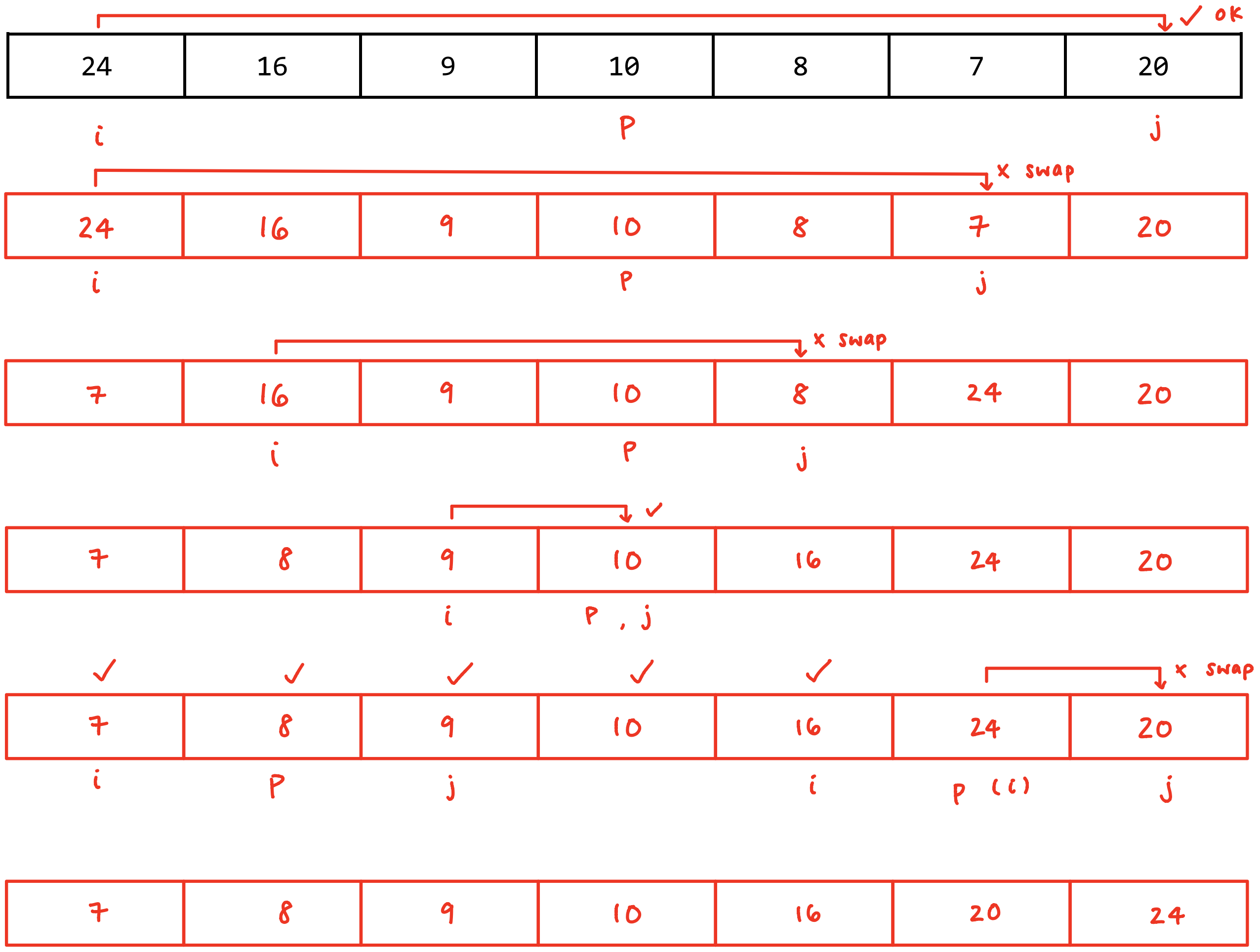
Thus, languages usually choose Quicksort over Mergesort due to its faster run time (just by a bit) and the fact that it uses no extra memory. Furthermore, the faster run time is because the partitioning step can actually be performed in place and very efficiently, which can make up for the lack of ewqual-sized recursive calls.

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:



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

median