Denish Oleke

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.

Denish Oleke

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			98	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	70	42	

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

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

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?

Best Size would be 500 since we don't know the ~~total~~ entries in the hash table Therefore, picking a maximum size will be the best option to Initialize the table size.

Denish Oteke

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 $\lambda = \frac{53491}{106963} = 0.5000888158 \approx 0.5$

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

  Yes, we should rehash. Basing on the load factor of 0.5, this means we have just started probing; Therefore, it is good we rehash as the load factor approaches 1.

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

  No, we should not rehash. Since we are dealing with separate chaining, each bucket is independent with its own list of entry containing same index. Therefore, reharshing is not ideal in this case.

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)

Denish Oteke

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 start Sucking at line 3 of the code (→) where; ArrayList<HashItem><T>> OldArray = array; This is because this line makes a deep copy of array which takes longer to process since it is ~~expen~~ expensive when the table cell is 50% full after rehashing the table.

Denish Oteke

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.

A good application would be Operating system since it process jobs by priority.

So, in situation where Users / tasks are fighting for limited resources, OS decides on which task should be executed next since longer tasks may halt shorter, Important tasks from being Executed.

Therefore, by assigning priorities to tasks using Priority Queue, OS can manage time more wisely and execute Importan tasks first.

Denish Oteke

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 is at $i/2$

Children: Left child is at $2*i$

Right Child is at $(2*i)+1$

Denish Oteke

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):

12	10									

etc:

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		

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	14	11		

Demish Oteke

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

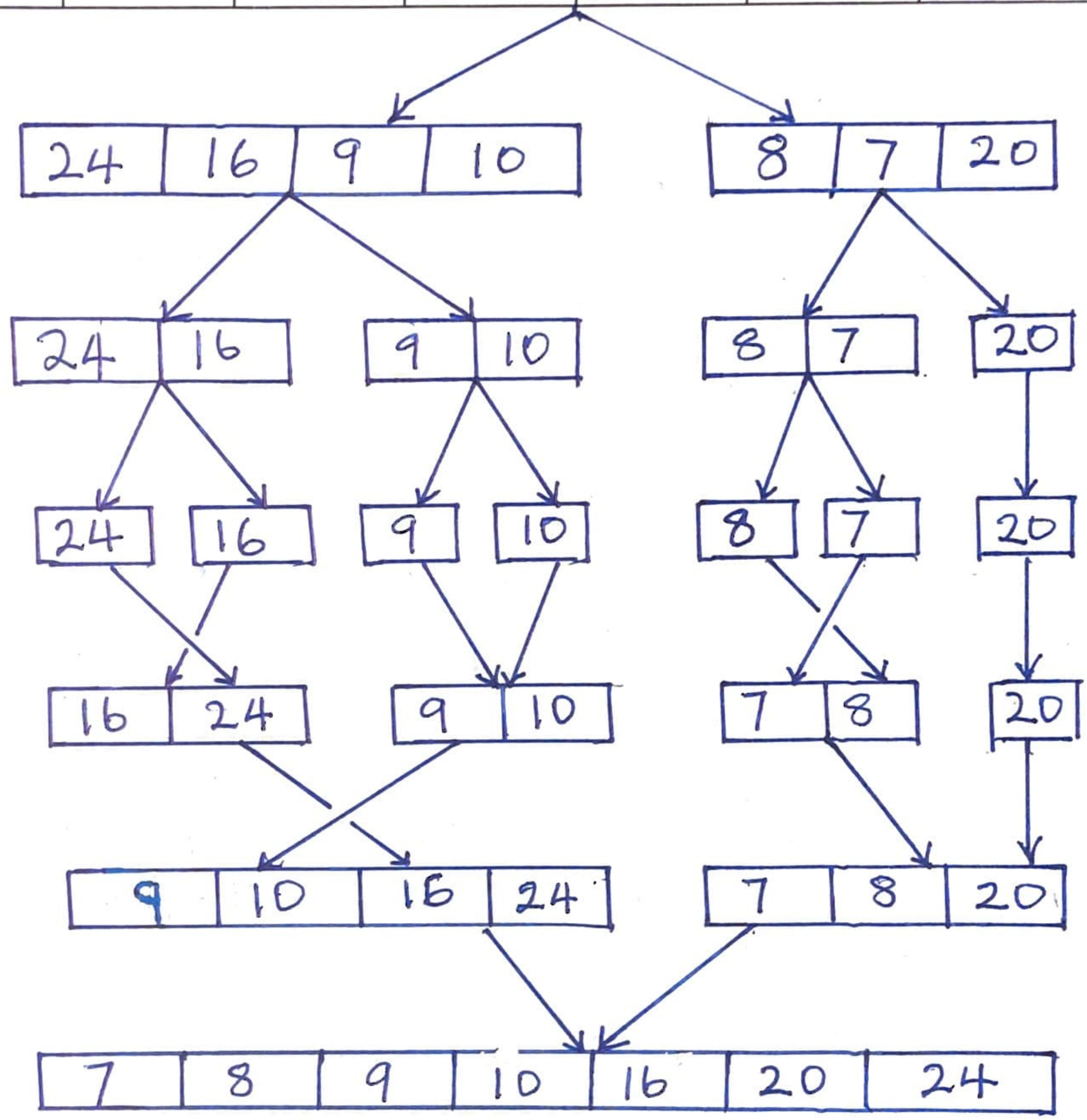
15. [3] What are the key differences between Mergesort and Quicksort? How does this influence why languages choose one over the other?

Mergesort requires extra memory in the order of $O(n)$ while quicksort does not require any additional memory.
When compared with running time, quick sort is fasterthan merge sort and does not demand extra storage/memory thus influencing languages to choose quick sort over merge sort.

Denish Oteke

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

24	16	9	10	8	7	20



Denish Oteke

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

							j
i	24	16	9	10	8	7	20

Let Pivot = 10 (middle value)
i = Index (0) – Element
j = Index (size – 1) element; for checked elements

24	16	9	10	8	7	20
					j	

7	16	9	10	8	24	20
	i			j		

			Correct			
i	p	j		i	p	j
7	8	9	10	16	24	20
					i	j

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

We compare i and j to Pivot and Move item larger than Pivot to the right and those less than pivot we move it to the left. Now that the left of 10 is Perfectly sorted, we will check the Right side of 10 and Sort the last 2 elements (24, 20) → (20, 24)

Final sorted list is below.

7	8	9	10	16	20	24