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.

You have a new branch for HW3 in your repository. Execute the merge request and put your PDF into that new HW3 directory. Once you do that, put a small file on Blackboard to let the TA know to grade your work.

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 1 98			9 42	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

	42		0	12		3	9	70	1	98
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?

101 because it is prime number and prime number gives table more evenly distributed and less clustered.

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 = .5

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

Yes we have to rehash linear probing collision function if the load factor is greater or equal to .5 because time required for search indexes in the hash table get increases from O(1) to O(2)when load factor is 0.5 .

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

NO because we rehash senate chaining collision function when load factor get greater or equal to 1. It is because searching time indexes is constant and each indexes is independent.

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)

6. [6] Enter a reasonable hash function to calculate a hash key for these function prototypes:

```
int hashit( int key, int TS )
{
int bucket = key% TS




}
```

```
int hashit( string key, int TS ){
int hashVal =0;
for(int i=0; i<key.lenght();i++){
hashVal = 37*hashVal+key.charAt(i);
}
hashVal = hashVal%TS;
if(hashVal<0){
hashVal = hashVal+TS
}
return hashVal
}
```

```
}
```

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());

            }

      }
}
```

Its because when you rehash your table , you need to find a table size that is 2 times the old table size and find out until you reached a prime number. This will help your table to be evenly distributed and less cluster.

Second when you put your old keys back to new table size , you can't just put them in the same bucket as before because they might not have same hash index. This will ruin your hash table.

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(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.

Priority queue can be used for scheduling treatment for patients in ER. ER patients with accident or gunshot needs to be treat earlier than the patient with fever despite their arrival time. For that we can put patients in the queue by using add method. And remove the most urgent patient by using get method.

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: 2i ,2i+1

What if it's a d-heap?

Parent: (i-1)/d

Children: (d*i)+1, (d*i)+2,…. (d*i)+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					

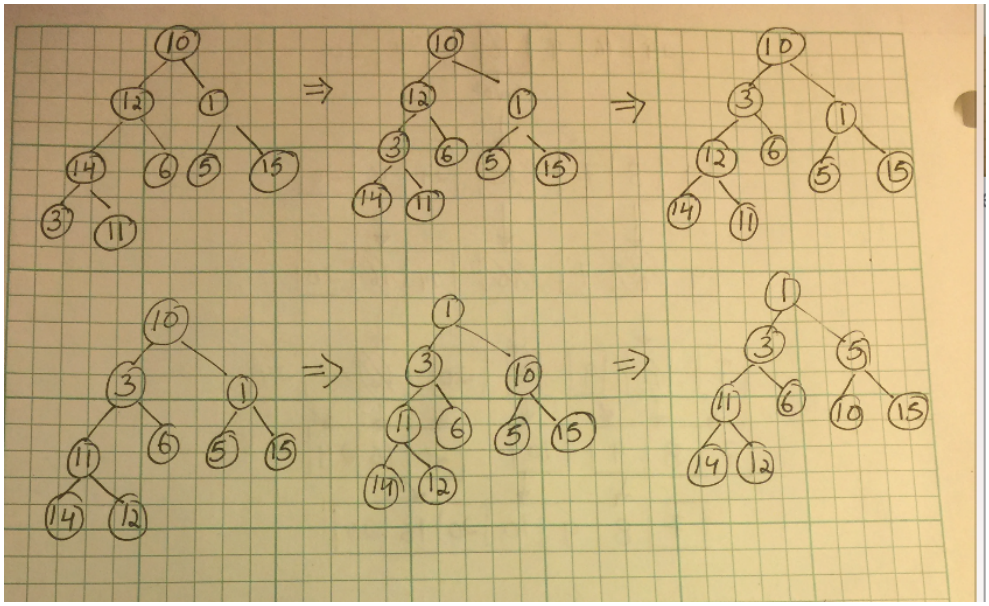
	1	6	5	14	12	10				

	1	6	5	14	12	10	15			

	1	3	5	6	12	10	15	14		

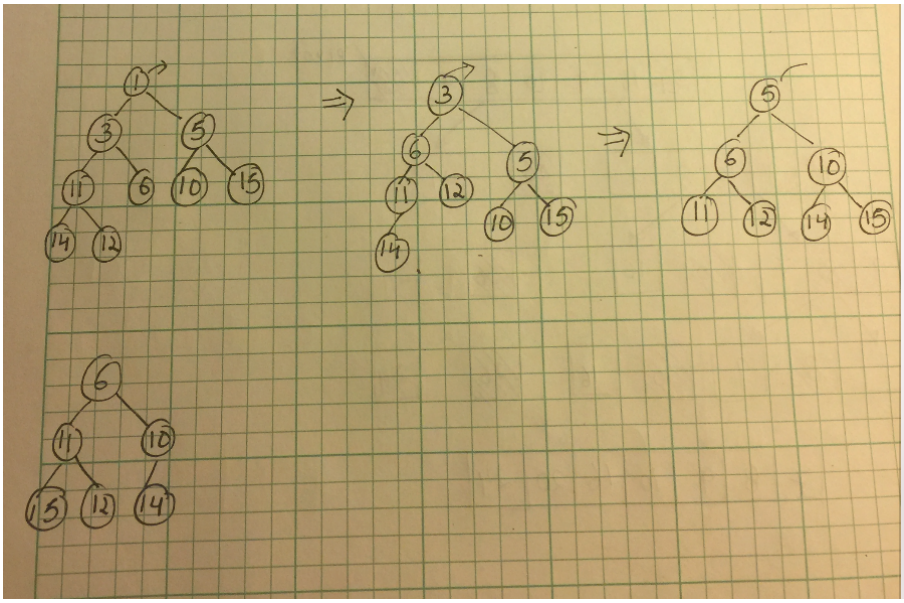
	1	3	5	6	12	10	15	14	11	

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(NlogN)	No
Merge Sort	O(NlogN)	Yes
Radix sort	O(N)	Yes
Quicksort	O(NlogN)	Yes

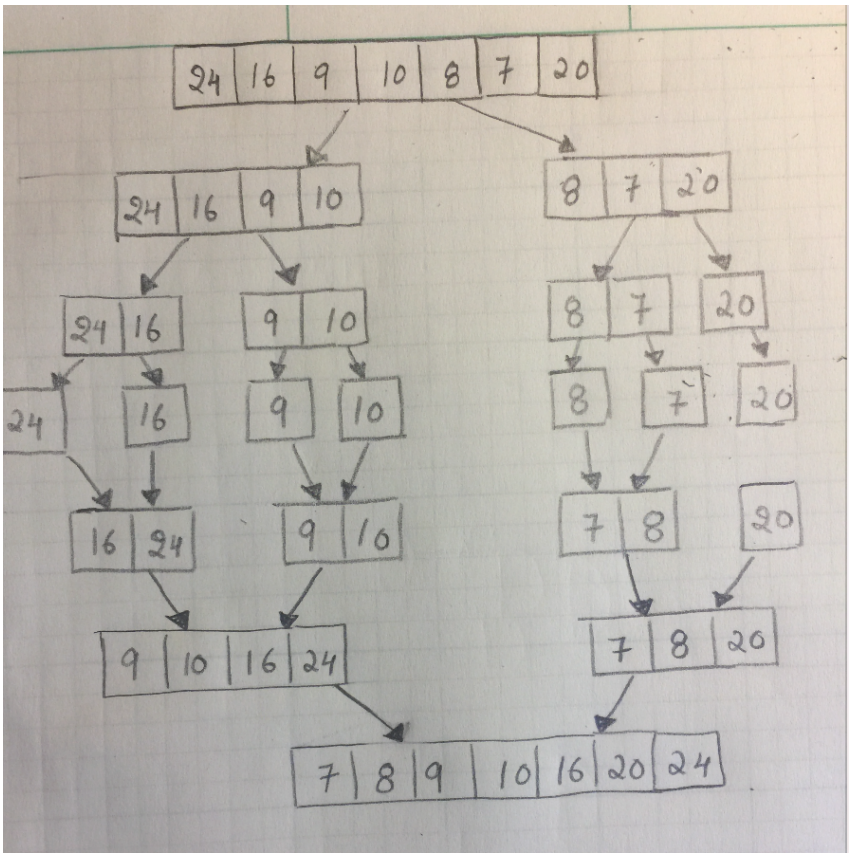
15. [3] What are the key differences between Mergesort and Quicksort? How does this influence why languages choose one over the other?

Even though both Merge and Quick sort has runtime of order O(N long N) and both follows divide and conquer strategy. But Merge sort so it needs extra memory space for its result. Whereas Quick sort doesn't required extra memory for copy it’s result.

Merge sort algorithm it perform less comparison and do more moves,therefore Java choose Merge sort over Quick sort because Java is object oriented language and it is comparison expensive. Since Quick sort perform more caparison compare to moves and caparison is more faster in C++ than moves.Therefore C++ language choose Quick sort over Merge sort.

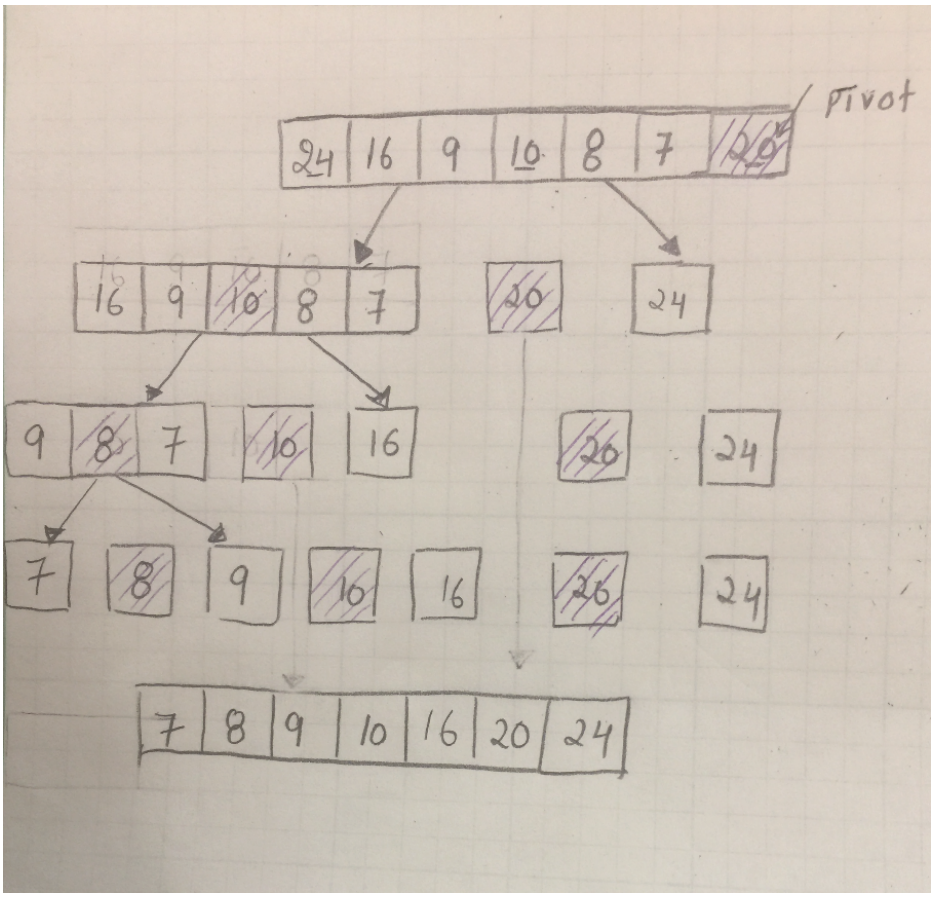
16. [4Draw out how Mergesort would sort this list:

24	16	9	10	8	7	20



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

24	16	9	10	8	7	20



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

I picked my pivot by taking median of first , middle and last element in the list.