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)

0	1	2	3	4	5	6	7	8	9	10
	3		0	98 1 12			42 9	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		0	12		70	9	42	98	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 is prime and fairly large which will result in a good spread (meaning less collisions, meaning faster lookup and insertion).

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):

$$\lambda = \frac{\#\ items}{table\ size} = \frac{53491}{106963} \approx \boxed{0.5001}$$

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

Yes, because cache misses per lookup grows exponentially past $\lambda = 0.5$, which we are just over.

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

NO, because cache misses per lookup grows fairly linearly, so we don't need to rehash until $\lambda > 1.0$.

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)	1
Rehash()	N
Remove(x)	1
Contains(x)	1

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

```
int hashit( int key, int TS )
{
    return key % TS;

}
```

```
int hashit( string key, int TS )
{
    int hash = 0
    for (char c : key.toCharArray()) {
        hash += c;
    }
    return hash % TS;
}
```

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());

            }

      }
}
```

nextPrime()

The new array size should be the next biggest (or a bigger) prime number, not double the previous array size. This is because when your hash function mods by table size, you don't want table size to have a lot of factors (meaning more collisions, meaning slower searches and inserts).

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)	log(N)
top()	1
pop()	1
PriorityQueue(Collection<? extends E> c) // BuildHeap	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.

I want to schedule print jobs so that the next job printed is the one that takes the least time to print. I would store the print jobs in a (min) priority queue sorted by "estimated time to print". This would be a good choice because each time a job is added it only takes O(logN) time, and the printer can receive a new job from the queue in O(1) time. If I were to use a different approach, such as sorting a list with quick sort for example, it would take O(N logN) time each time the list is sorted. So a priority queue would be an efficient and elegant solution.

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: $2i$, $2i+1$

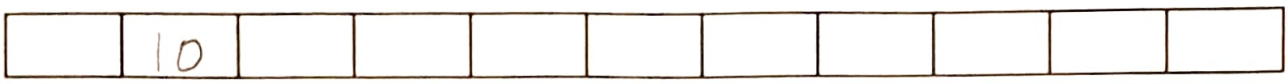
What if it's a d-heap?

Parent: i/d

Children: $di, di+1, di+2, \ldots, di+(d-1)$

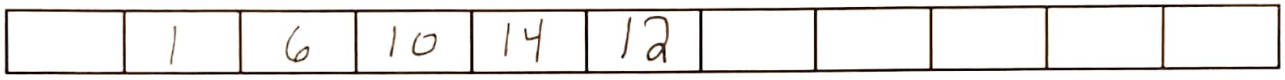
$$= \sum_{n=0}^{d-1} di + n$$

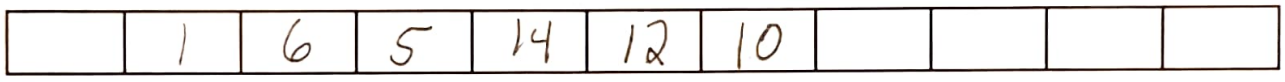
11. [6] Show the result of inserting 10, 12, 1, 14, 6, 5, 15, 3, and 11, one at a time, into an initially empty binary heap. Use a 1-based array like the book does. After insert(10):

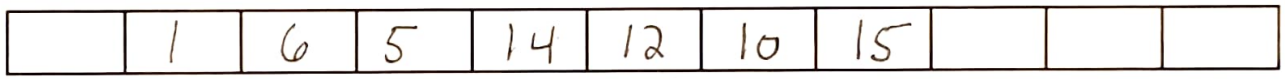


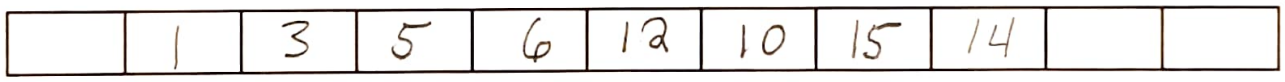
After insert (12):

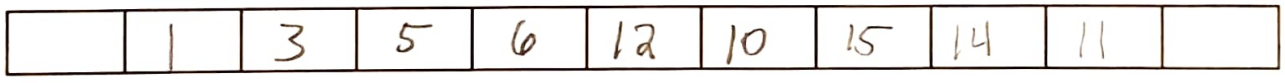
etc:



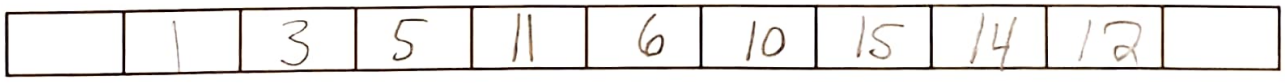






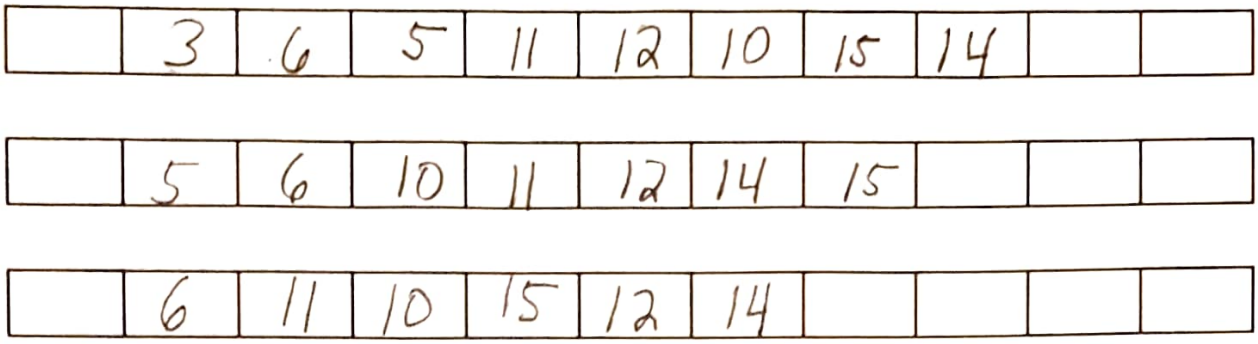


12. [4] Show the same result (only the final result) of calling buildHeap() on the same vector of values: {10, 12, 1, 14, 6, 5, 15, 3, 11}



13. [4] Now show the result of three successive deleteMin / pop operations from

the prior heap:



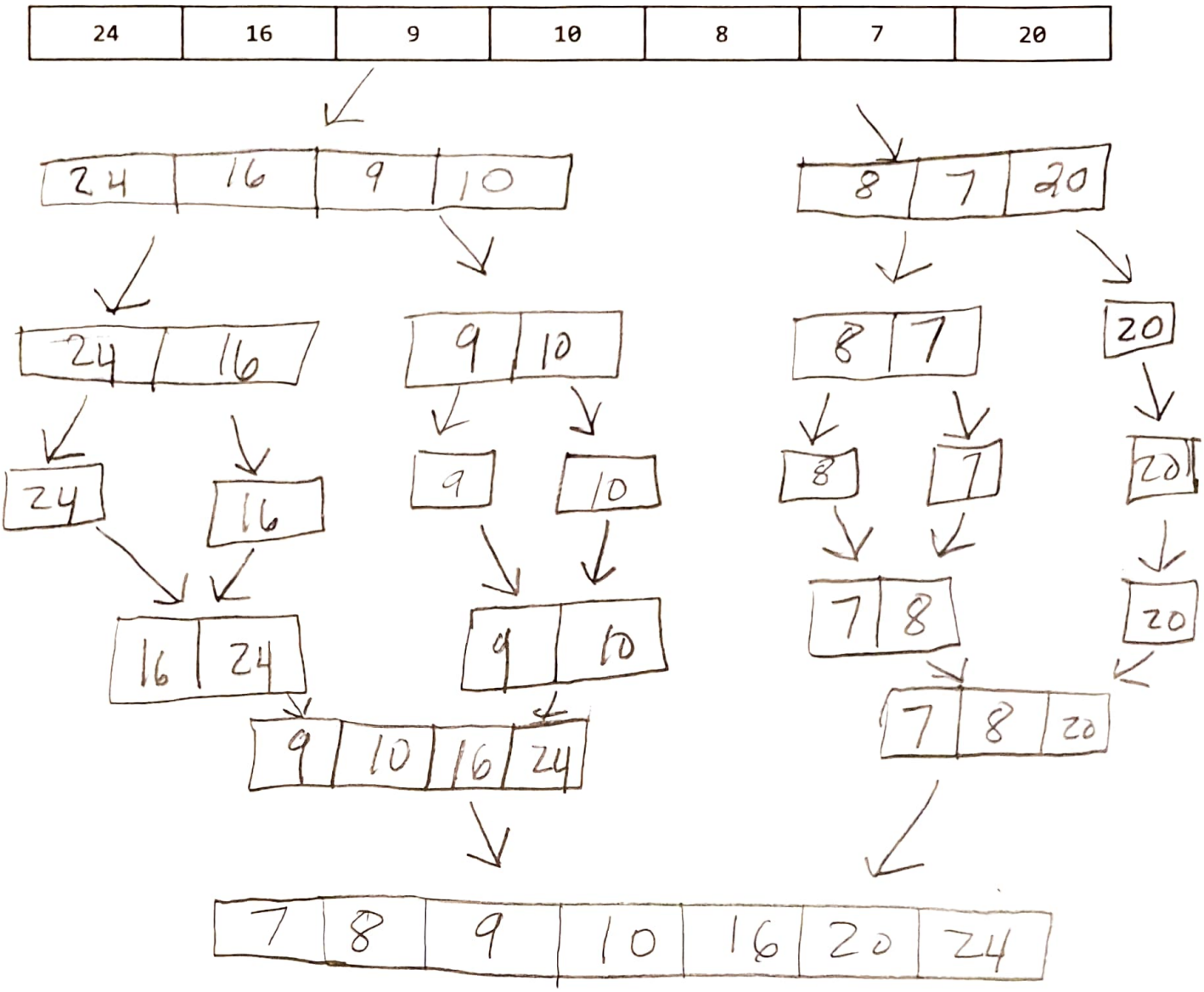
14. [4] What are the average complexities and the stability of these sorting algorithms:

Algorithm	Average complexity	Stable (yes/no)?
Bubble Sort	n^2	Yes
Insertion Sort	n^2	Yes
Heap sort	$n \log n$	No
Merge Sort	$n \log n$	Yes
Radix sort	$p(n+b)$	Yes
Quicksort	$n \log n$	No

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

It depends on the cost to compare elements and move elements. If compares are faster: quicksort. If moves are faster: mergesort.

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



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

24	16	9	10	8	7	20

Med(24, 10, 20) = 20

16	9	10	8	7

, 20 , [24]

Med(16, 10, 7) = 10

Med(24) = 24

[7 8 9], 10, [16], 20, 24

Med(7,8,9) = 8

Med(16) = 16

[7], 8, [9], 10, 16, 20, 24

Med(7) = 7

Med(9) = 9

7, 8, 9, 10, 16, 20, 24

7	8	9	10	16	20	24

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

Median(first, middle, last)