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

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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 3 | 98 | 0 | 12 | | | 42 | 70 | 9 | 1 |
| 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?

I chose 101 because hash table sizes should be prime to avoid clustering of hash codes with similar multiples.

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):

  53491 / 106963 = 0.50 = 50%

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

  Yes, linear probing starts to slow down after a load factor of about 50%

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

  No, separate chaining can withstand very high load factors, so if we're only barely rehashing with linear probing we shouldn't rehash with separate chaining.

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) |

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 size of the Hash Table is not guaranteed to be prime, and in the long run this will cause clustering which ruins the time complexity of the 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(log n) |
| PriorityQueue(Collection<? extends E> c) // BuildHeap | O(n log 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 Queues are good for any situation one element might need to take priority over another. For example, lets say you're at my old job organizing races at a go kart track. Every rider is typically assigned a kart at random but it's common courtesy to give regulars the kart first in line. This process could be automated by placing riders in a max heap priority queue based on their previous races and giving the first kart to the rider at the top, the second to the next highest rider, etc.

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

Parent: $\frac{i+1}{2}$

Children: $2i$ and $2i+1$

which kind? question is vague, assuming max

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}

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 15 | 12 | 14 | 11 | 6 | 1 | 5 | 3 | 10 | |

Max previous question didn't specify max or min

13. [4] Now show the result of three successive delete~~Min~~Max / pop operations from the prior heap:

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 14 | 12 | 5 | 11 | 6 | 1 | | 3 | 10 | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 12 | 11 | 5 | 10 | 6 | 1 | | 3 | | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 11 | 10 | 5 | 3 | 6 | 1 | | | | |

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

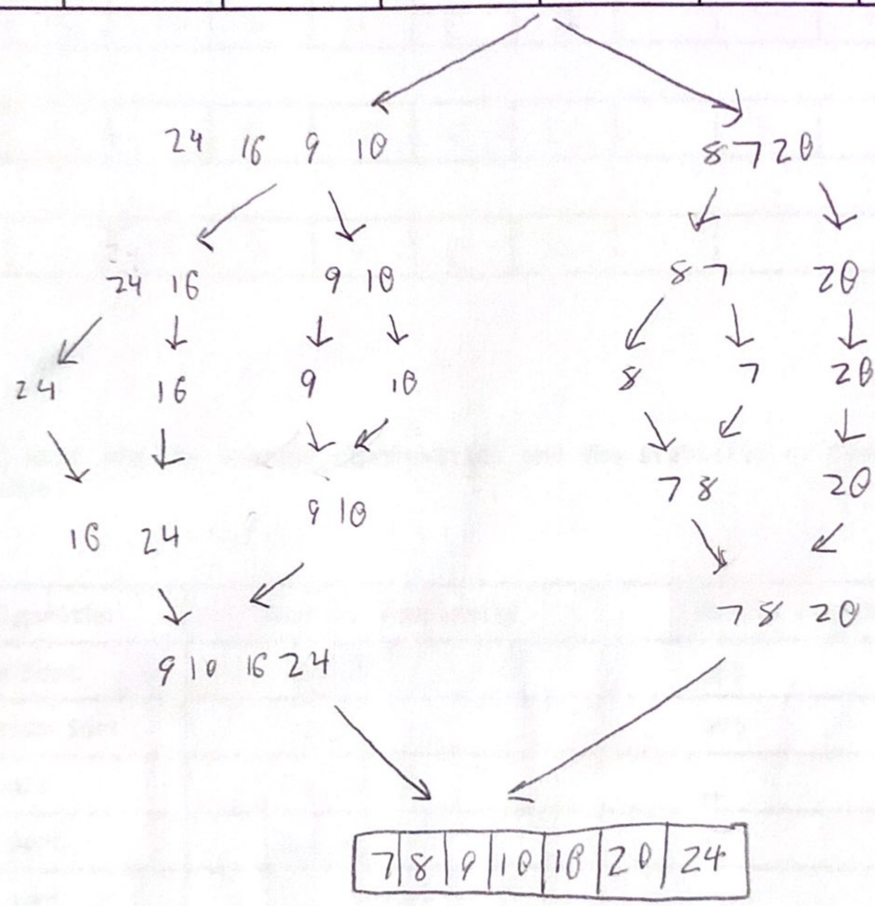
| Algorithm | Average complexity | Stable (yes/no)? |
|---|---|---|
| Bubble Sort | $O(n^2)$ | yes |
| Insertion Sort | $O(n^2)$ | yes |
| Heap sort | $O(n \log n)$ | no |
| Merge Sort | $O(n \log n)$ | yes |
| Radix sort | $O(n \log n)$ | yes |
| Quicksort | $O(n \log n)$ | no |

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

Quick sort runs faster than mergesort and also uses less space but mergesort is stable and doesn't have the $O(n^2)$ worst case for large data sets that quick sort does. When dealing with lots of data quickly is important, languages use mergesort, otherwise they use quicksort.

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

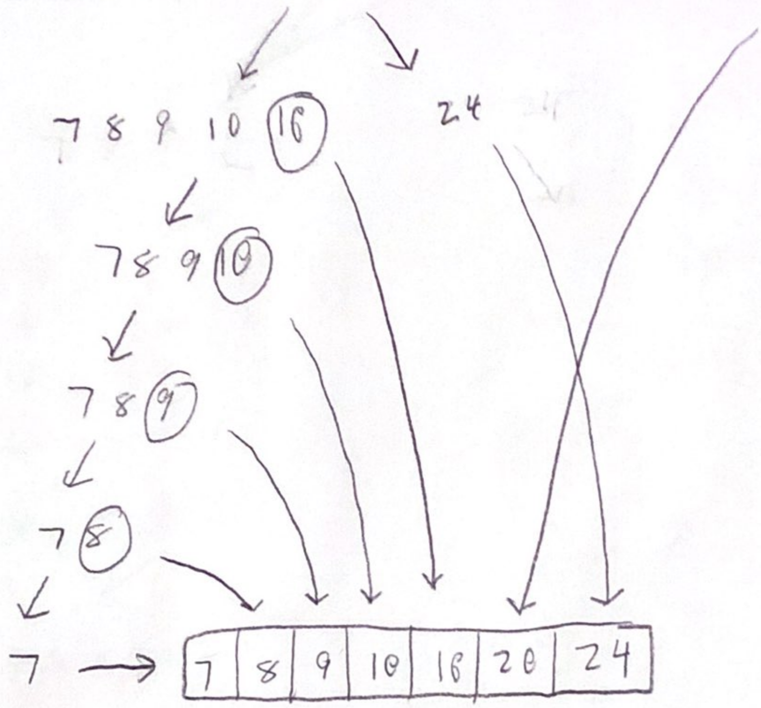
| | | | | | | |
|---|---|---|---|---|---|---|
| 24 | 16 | 9 | 10 | 8 | 7 | 20 |



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

pivot circled

| | | | | | | |
|---|---|---|---|---|---|---|
| 24 | 16 | 9 | 10 | 8 | 7 | 20 |




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