

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:

$$\text{hashkey}(\text{key}) = (\text{key} * \text{key} + 3) \% 11$$

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 $\text{hashkey}(\text{key})$ and add the current $\text{probe}(i')$ offset. If that bucket is full, increment i until you find an empty bucket.

Linear Probing: $\text{probe}(i') = (i + 1) \% \text{TableSize}$

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

Quadratic Probing: $\text{probe}(i') = (i * i + 5) \% \text{TableSize}$

	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?

Choose to use the largest prime number 101 to be the best initial table size to pick, the prime number 101 can only be divisible by 1 and itself. Hence, choose this largest prime number among them will greatly reduce the occurrence of collision.

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):

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

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

Yes, the lambda is greater than 0.5, the table is too full for linear probing, so we should rehash.

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

No, each bucket in separate chaining is independent, and may have some sort of list of entries with the same index. Since the lambda is less than 1 in this case, we shouldn't rehash.

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 (worst case)	Average case
Insert(x)	$O(n)$	$O(1)$
Rehash()	$O(n)$	$O(n)$
Remove(x)	$O(n)$	$O(1)$
Contains(x)	$O(n)$	$O(1)$

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

```
Int hashit (int key, int TS){  
    return (key * key + 3 ) % TS;  
}  
  
Int hashit (String key, int TS){  
    int hashKey = 0;  
    char ch[] = key.toCharArray();  
    for( int i = 0; i<key.length(); i++){  
        hashKey = hashKey +ch[ i ];  
    }  
    return (hashKey * hash+ 3) % 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());
        }
    }
}
```



The issues occur here, this only resize the array size not rehash the hash key. Hence, the hash key isn't change, even the table size increase, the performance won't improve for large data set.

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.

While in competition games, the less time the user spends, the higher rank the user obtains. We can use the min heap to determine a group of users's gaming result, and produce the ranking. Using a heap to find the smallest element is definitely a lot faster than sorting an array. So, using a binary here can definitely improve efficiencies.

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

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

Children:

$$2i, \quad 2i + 1$$

What if it's a d-heap?

Parent: $\frac{(i - 1)}{d + 1}$

Children:

$$(i - 1) \cdot d + j + 1$$

*(j is the nth of child)

Min Heap :

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: insert 1 and percolate up :

	1	12	10							
--	---	----	----	--	--	--	--	--	--	--

insert 14 :

	1	12	10	14						
--	---	----	----	----	--	--	--	--	--	--

insert 6 and percolate up :

	1	6	10	14	12					
--	---	---	----	----	----	--	--	--	--	--

insert 5 and percolate up :

	1	6	5	14	12	10				
--	---	---	---	----	----	----	--	--	--	--

insert 15 :

	1	6	5	14	12	10	15			
--	---	---	---	----	----	----	----	--	--	--

insert 3 and percolate up :

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

insert 11 :

	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	$O(N \log N)$	no
Merge Sort	$O(N \log N)$	yes
Radix sort	$O(KN)$	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 main difference between Mergesort and Quicksort is that the quicksort sorts the elements by comparing each element with a pivot element while mergesort divides the array into two subarrays again and again until one element is left.

Quicksort is an unstable sorting technique, it might change the occurrence of two similar elements in the array. So Quicksort is more suitable for small arrays, works faster for small data set and requires minimum space.

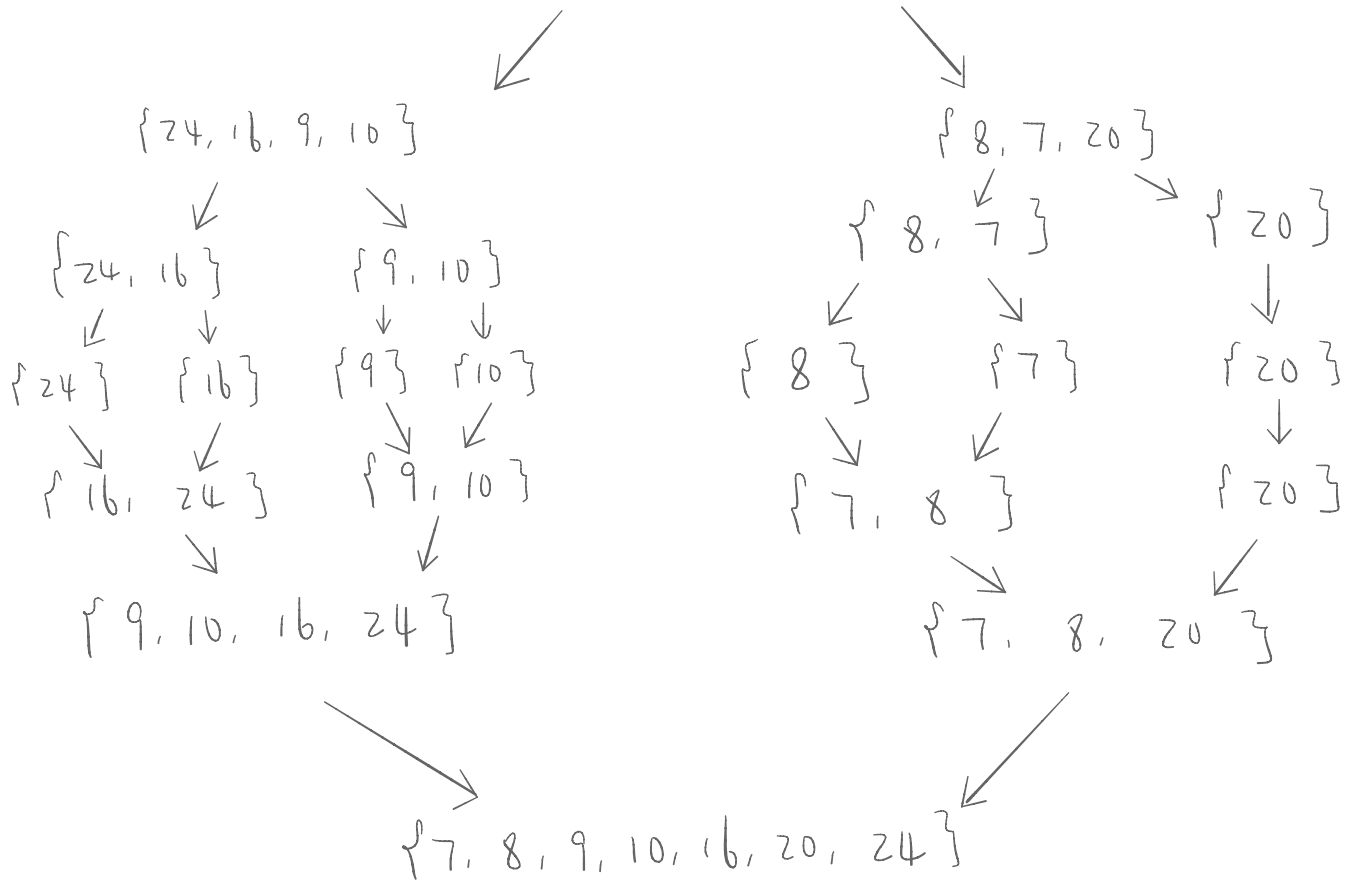
Mergesort is a stable algorithm, it works in consistent speed for all data sets and requires more space.

Using Mergesort for larger data set will work better than Quicksort.

Using Quicksort for smaller data set will work faster and save spaces than Mergesort.

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:

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

↑
pivot

i : index of smaller element = -1

j : loop element = 0

$20 > 24$ False, $i = -1, j = 1$ no action,

$20 > 16$ True, $i = 0, j = 1$, swap(arr[i], arr[j])

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

$20 > 9$ True, $i = 1, j = 2$, swap(arr[i], arr[j])

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

$20 > 10$ True, $i = 2, j = 3$, swap

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

$20 > 8$ True, $i = 3, j = 4$, swap

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

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

$20 > 7$ True, $i = 4, j = 5$, swap

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

Since j becomes 6, so we come out of index.

We know swap $arr[i+1] = arr[t]$ and pivot:

{ 16, 9, 10, 8, 7, 20, 24 } * 20 is sorted

Now we begin quick sorting the left part:

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

↓
pivot

$7 > 16$, false, $i = -1$, $j = 1$

$7 > 9$, false, $i = -1$, $j = 2$

$7 > 10$, false, $i = -1$, $j = 3$

$7 > 8$, false, $i = -1$, $j = 4$

Since j becomes out of index,

swap $arr[i+1] = arr[0]$ and pivot:

{ 7, 9, 10, 8, 16, 20, 24 } * 7 is sorted

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

↓
pivot

$8 > 9$ false, $i = -1$, $j = 1$

j becomes 2, swap $arr[i+1] = arr[0]$ and pivot.

* 8 is sorted

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

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

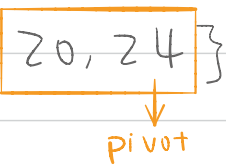
↓
pivot

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

Left array sorted.

Now we begin quick sorting the right part:

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



pivot

$24 > 20$ true \Rightarrow no action, 24 is already at its correct position.

Now, conquer up partition, the array sorted:

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