1. [6] Starting with an empty hash table with a fixed size of 11, insert the following keys in order into four 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

12 × 12 + 3 = 147 → 4
9 × 9 + 3 = 84 → 7
1 × 1 + 3 = 4 → 4
0 × 0 + 3 = 3 → 3
42 × 42 + 3 = 1767 → 7
98 × 98 + 3 → 4
70 × 70 + 3 →
3 × 3 + 3 = 12 → 1

Separate Chaining (buckets)

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

To probe, start at i = hashkey and do i++ if collisions continue

Linear Probing: probe(i') = (i + 1) % TableSize

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

→ 42
98 → 98 → 98 → 98 → 98
1 → 1
3 → 3 → 3
42 →

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

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

→ 70
1 → 1
98 → 98 → 98
42 → 42
70 →

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? Prime numbers ensure that fewer collisions occur

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):

λ = size / buckets

$= \frac{53491}{106963} = 0.5000888157...$

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

yes, table is now full; since it is linear, 53491 entries are taking up the total 106963 of spaces. Since $\lambda > 0.5$, table is too full

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

Not necessarily, mainly if number of buckets being used is more than half of the buckets total. We should only rehash if $\lambda > 1.0$. In this case, we would not need to rehash. (on average, lists are $\frac{1}{2}$ nodes long)

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)

5. [3] If your hash table is made in C++11 with a vector for the table, has integers for the keys, uses linear probing for collision resolution and only holds strings... would we need to implement the Big Five for our class? Why or why not?

(unless working w/ objects) vector < string > foo ;

Not really, there is no need to construct anything nor copy anything since we are only inserting / moving things around, not actually changing values.

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

```
int hashit( int key, int TS )   // ? tablesize ?
{

        return  key % TS;

        // clean and simple :)


}


int hashit( string key, int TS )   5.4 from book.
{

        int  hash = 0;
        for ( char ch : key )
        {
           hash = 37 * hash + ch;
        }
        return  hash % TS;
}
```

7. [3] I grabbed some code from the Internet for my linear probing based hash table because the Internet's always right. 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 *way* 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?

Haha.

```
/**
* Rehashing for linear probing hash table.     5.22 in the book
*/

void rehash( )
{
      vector<HashEntry> oldArray = array;

      // Create new double-sized, empty table
      array.resize( 2 * oldArray.size( ) );
      for( auto & entry : array )
            entry.info = EMPTY;

      // Copy table over
      currentSize = 0;
      for( auto & entry : oldArray )
            if( entry.info == ACTIVE )
                  insert( std::move( entry.element ) );
}
```

not prime, having to look through more elements than necessary

is this legal?
array.clear();

why do you need to look, just move everything over

```
for (auto & x : array)
{
   insert(std::move(x))
```

8. [4] Time for some heaping fun! What's the time complexity for these functions in a binary heap of size N?

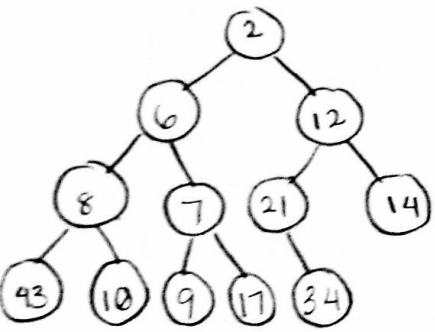
root

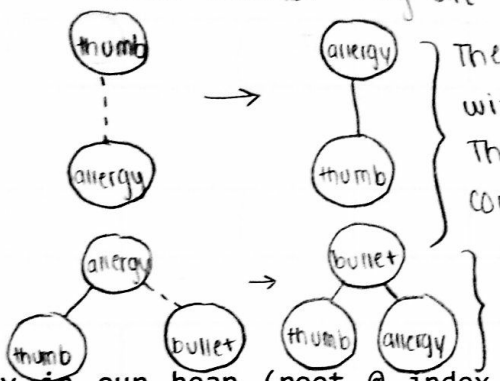
Function	Big-O complexity
insert(x)	O(log N)
findMin()	O(1)
deleteMin()	O(log N)
buildHeap(vector<int>{1...N})	O(N (log N))

prioritizing

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.

How injured someone is when they come into the emergency room. Suppose someone comes in with a broken thumb. They are placed on the queue.





Then, someone else comes in with a severe allergic reaction. They are moved up since their condition is more severe.

Then, someone else comes in with a bullet wound. They are moved up so now it would first be the bullet wound, then the allergic reaction, then the broken thum

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: $d(i)+1$, $d(i)+2$, $d(i)+3 \ldots d(i)+d$

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

1	12	10								

14

1	12	10	14							

6

1	6	10	14	12						

5

1	6	5	14	12	10					

15

1	6	5	14	12	10	15				

3

1	3	5	6	12	10	15	14			
1	2	3	4	5	6	7	8			

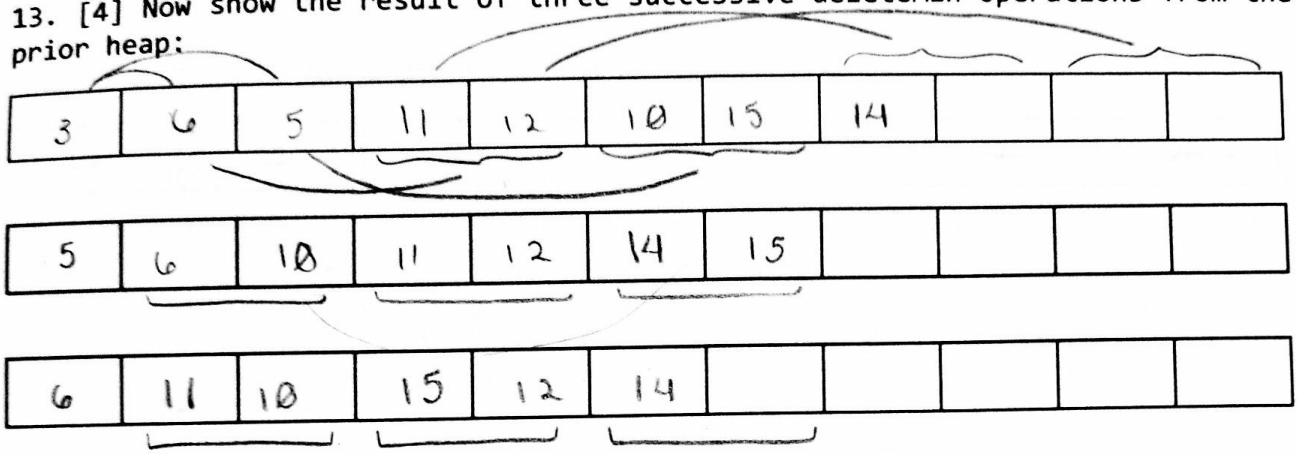
11

1	3	5	6	12	10	15	14	11		
1	2	3	4	5	6	7	8	9		

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		

13. [4] Now show the result of three successive deleteMin operations from the prior heap:



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

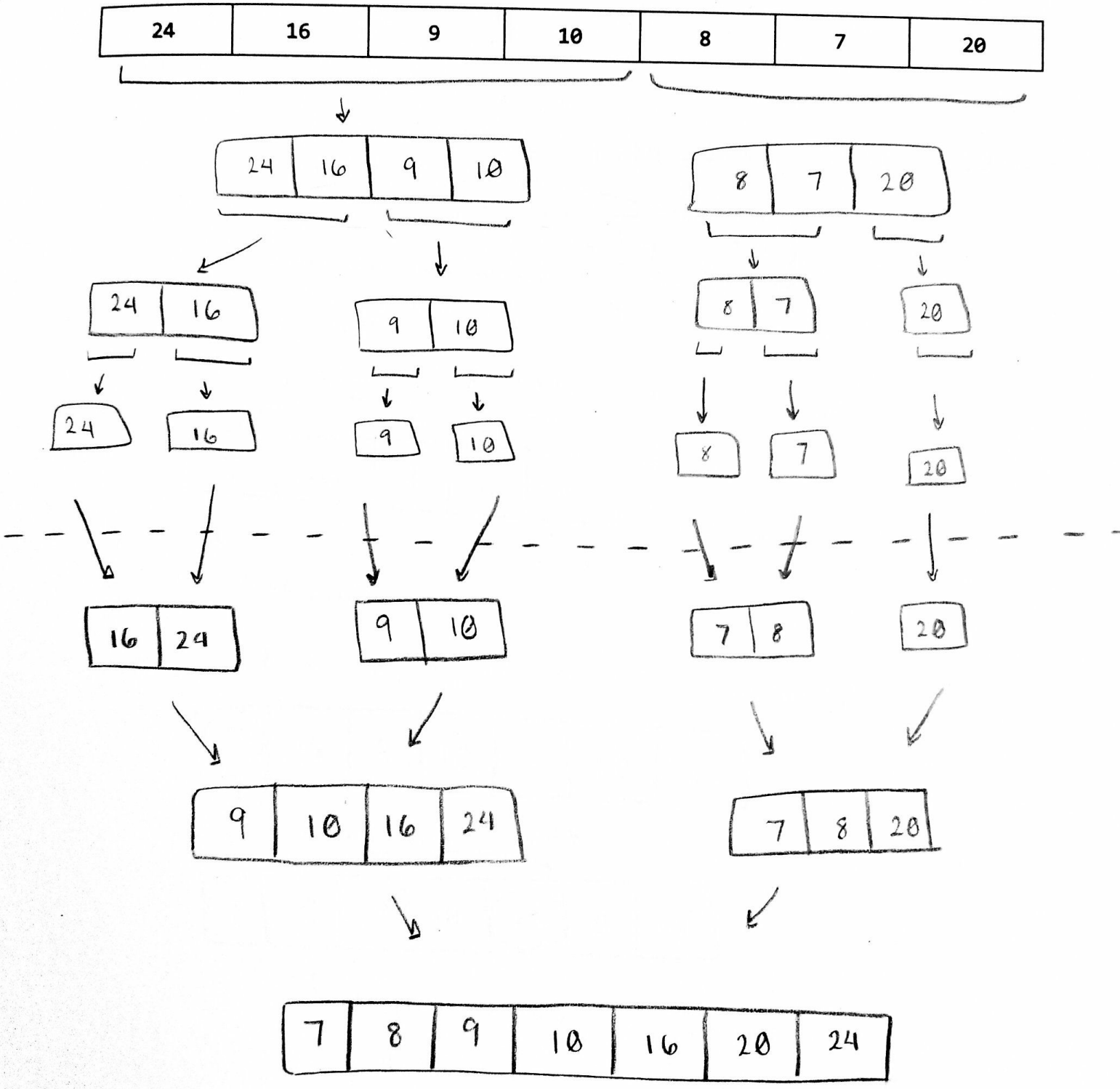
Algorithm	Average complexity	Stable (yes/no)?
Bubble Sort	$O(N^2)$	
Insertion Sort	$O(N^2)$	
Heap sort	$O(N(\log N))$	
Merge Sort	$O(N(\log N))$	
Radix sort	$O(N)$	
Quicksort	$O(N(\log N))$	

15. [2] What are the key differences between Mergesort and Quicksort? How does this influence why languages choose one over the other?

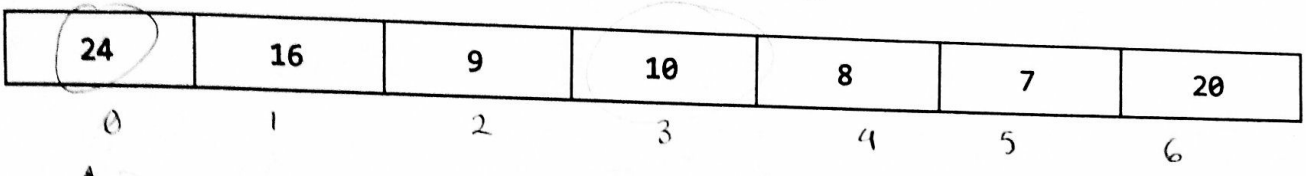
merge → divide and conquer; split up until individual values, then sort one by one.

quick → uses pivot then compares value by chosen pivot

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

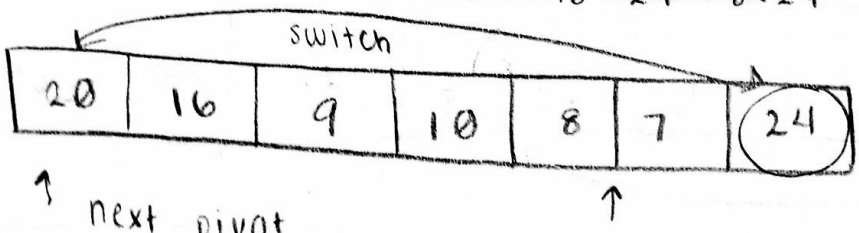


17. [4] Draw how Quicksort would sort this list:



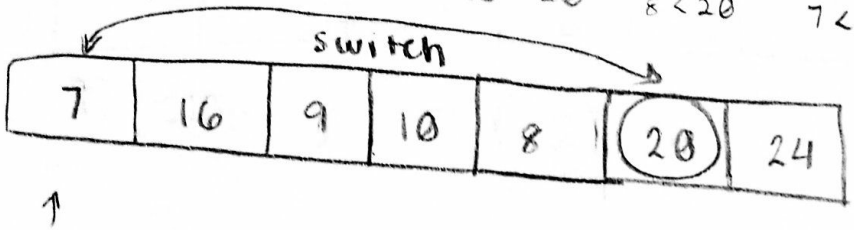
↑ starting pivot

16 < 24 9 < 24 10 < 24 8 < 24 7 < 24 20 < 24



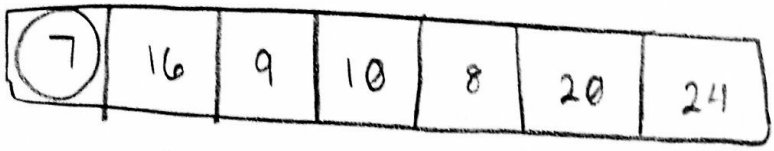
↑ next pivot ↑

16 < 20 9 < 20 10 < 20 8 < 20 7 < 20

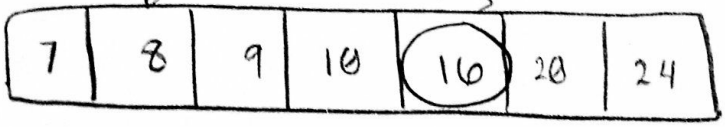


↑ next pivot

7 < 16, 9, 10, 8 → do nothing



↑ next pivot switch



↑ next pivot → OK!

↑ next pivot → OK!