1 On Wednesdays We Wear Red & Black

1.1 Here is a refresher on inserting into a red-black tree:

```
public void put(Key key, Value val) {
    root = put(root, key, val);
    root.color = BLACK;
}

private Node put(Node h, Key key, Value val) {
    if (h == null){
        return new Node(key, val, RED, 1);
    }
    int cmp = key.compareTo(h.key);
    if  (cmp < 0) {
        h.left  = put(h.left,  key, val);
    } else if (cmp > 0) {
        h.right = put(h.right, key, val);
    } else {
         h.val = val;
    }
    if (isRed(h.right) && !isRed(h.left)) {
         h = rotateLeft(h);
    }
    if (isRed(h.left)  &&  isRed(h.left.left)) {
        h = rotateRight(h);
    }
    if (isRed(h.left)  &&  isRed(h.right)) {
        flipColors(h);
    }
    h.size = size(h.left) + size(h.right) + 1;

    return h;
}
```

Now draw out the left leaning red black tree resulting from inserting the following- 35, 11, 49, 9, 7, 51 and 50.

2 Asymptotics

2.1 Please give lower and upper bounds for the overall runtime of these functions.

```
void g(double N) {
    if (N<=1)
        return;
    if (N%2 == 0) {
        for(double i = N/2; i >= N/4; i = i/2) {
                func(i) // linear with respect to i
        }
    }
    g(N/2)
}
```

2.2 Under what conditions are the best case runtime achieved. Under what conditions are the worst case runtime achieved

```
// x has nonnegative integers and the size is M
void t(int[] x; int N) {
        boolean flag = true;
    while (flag) {
            flag = false;
            for (int i = 0; i < x.length; i++) {
                    if(x[i] < N) {
                            x[i] += 1;
                flag = true;
            }
        }
    }
}
```

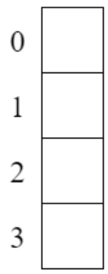
2.3 Give a tight bound on the worst case runtime of the following function

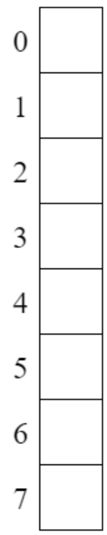
```
int f(int N) {
        if (N == 1)
                return 1;
        int y = 0;
        for (int x = 0; x < N; x += 1) {
                y += 1;
        }
        return f(N/2) + f(N/2) + y;
}
```

3 Hashing

3.1 Consider a hash table with external chaining, using the hash function $h(x) = x \mod 10$. The table resizes when the load factor exceeds 0.75. Draw out the hash table as we insert the following values:

3, 19, 15, 10, 17, 18





3.2 Here's a class for a hashtable that takes key, value pairs and stores them in a hashtable with external chaining. The keys in this problem are all strings, and the hashcode for each string is its length. Implement the hash function `hashfunc(String key)` and complete the method `insert` that inserts new key value pairs into the hashtable. You will be implementing the part of `insert` that handles resizing. It should be done recursively.

```
// A node of the external chains
class HashNode<String, V> {
    String key;
    V value;
    HashNode<String, V> next;

    public HashNode(String key, V value) {
        this.key = key;
        this.value = value;
    }
}

// class for the actual table
class HashTable<String, V> {
```

```
private HashNode<String,V>[] hashtable;
private int numBuckets;
private int numElements;
private int maxLoad = 0.75;

public HashTable() {
    numBuckets = 10;
    numElements = 0;
    hashtable = new HashNode<String,V>[numBuckets];

    // Create empty chains
    for (int i = 0; i < numBuckets; i++)
        hashtable[i] = null;
}

// Given a key, returns the hashcode for that key
private int hashfunc(String key) {
    return __________________;
}

public void insert(String key, V value) {
    int index = hashfunc(key);
    HashNode<String, V> head = hashtable[index];

    //check if key is already in table
    while (head != null) {
        if (head.key.equals(key)) {
            head.value = value;
            return;
        }
        head = head.next;
    }

    // resize
    if (_______________________) {
        numBuckets = ___________________;
        HashNode<String,V>[] temp = hashtable;
        hashtable = new HashNode<String,V>[numBuckets];
        numElements = _____________;

        for (int i = 0; i < numBuckets; i++) {
            ________________________________;
```

```
        }

            for (HashNode<String, V> headNode : __________) {
                while (_________________________) {
                    ________________________________________;
                    ________________________________________;
                }
            }
        }

        numElements++;
        HashNode<String, V> newNode = new HashNode<String, V>(key, value);
        newNode.next = head;
        hashtable[index] = newNode;
    }
}
```

4 Heaps

4.1 You have been hired by Alan to help design a priority queue implementation for *Kelp*, the new seafood review startup, ordered on the timestamp of each `Review`.

Describe a data structure that supports the following operations.

- `insert(Review r)` a `Review` in $O(\log N)$.
- `edit(int id, String body)` any one `Review` in $\Theta(1)$.
- `sixtyOne()`: return the sixty-first latest `Review` in $\Theta(1)$.
- `pollSixtyOne()`: remove and return the sixty-first latest `Review` in $O(\log N)$.

5 KD-Trees

5.1 1. The root node divides the space into…

   - 2 quadrants, positive and negative x
   - 2 quadrants, positive and negative y
   - 4 quadrants
   - None of the above

2. Suppose you have just one node (65, 12). You want to insert (1, 12). Where should this node be placed in the tree?

   - Left subtree
   - Right subtreee

3. Is the node with value (1, 12) x-aligned or y-aligned?

   - X-aligned
   - Y-aligned

4. Suppose after adding (1, 12), you decide to add (3, 15). Should this node be placed in the left or right subtree of (1, 12)?

   - Left
   - Right

5. When we want to find the minimum element of a specific dimension D, we like to go through the tree and ask at each dimension if the dimension we are currently standing in and D are the same. If true, then the minimum element is in…

   - The current node
   - The right subtree
   - The left subtree
   - An above node

6. If false, then the minimum element is in…

   - The current node
   - The right subtree
   - The left subtree
   - An above node

7. The average and worst runtime, respectively, for searching is…

- O(N), O(N)
- O(logN), O(logN)
- O(N), O(N*N)
- O(logN), O(N)

8. The average and worst runtime, respectively, for searching is...
   - O(N), O(N)
   - O(logN), O(logN)
   - O(N), O(N*N)
   - O(logN), O(N)
9. The space complexity is...
   - O(N)
   - O(logN)
   - O(N*N)
   - O(NlogN)