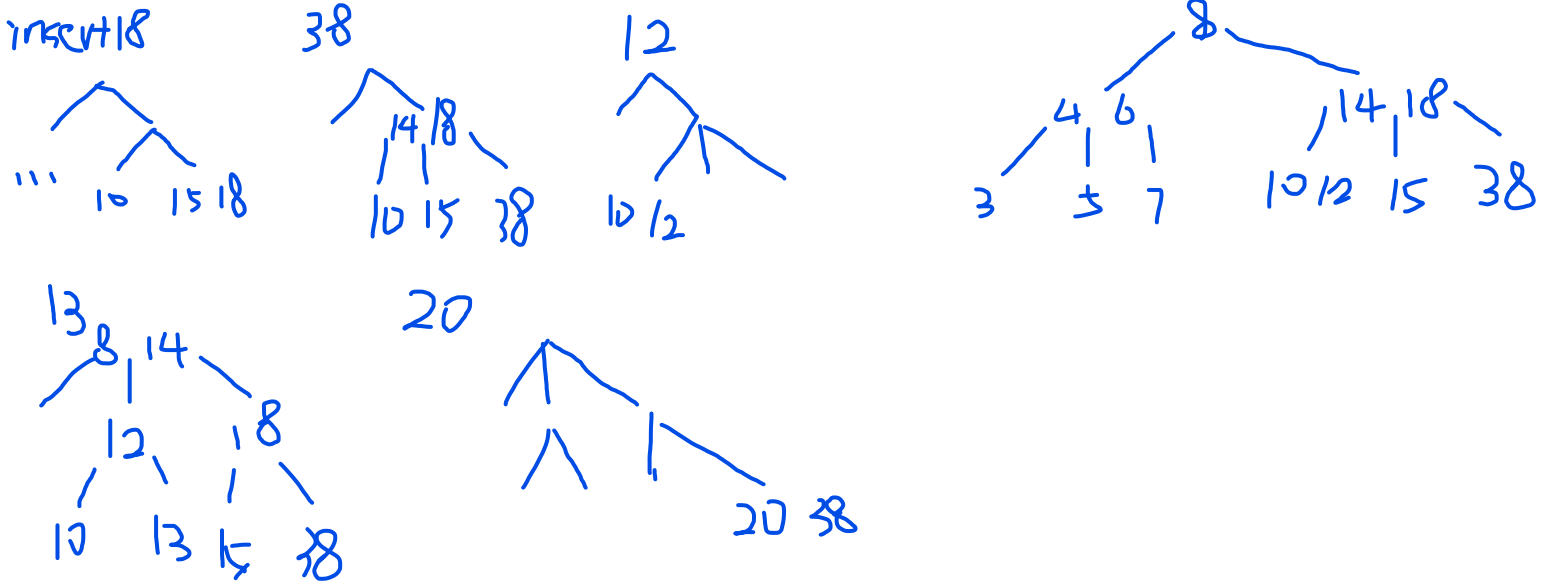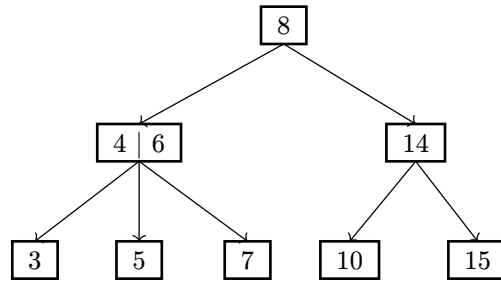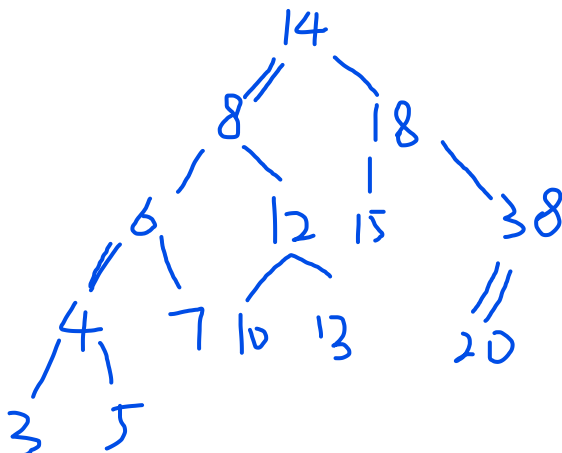# 1 2-3 Trees and LLRBs

(a) Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.

(b) Now, convert the resulting 2-3 tree to a left-leaning red-black tree.

(c) If a 2-3 tree has depth **H** (that is, the leaves are at distance **H** from the root), what is the maximum number of comparisons done in the corresponding red-black tree to find whether a certain key is present in the tree?

$$H_1 \qquad 2H+2$$

(d) Now, insert 9 into the LLRB Tree below. Describe where you would insert this node, and what balancing operations (**rotateLeft**, **rotateRight**, **colorSwap**) you'd take to balance the tree after insertion. Assume that in the given LLRB, dotted links between nodes are red and solid links between nodes are black.

rotateleft(8);
rotateright(10);
colorswap(10);   colorFlip(9);
colorswap(7);
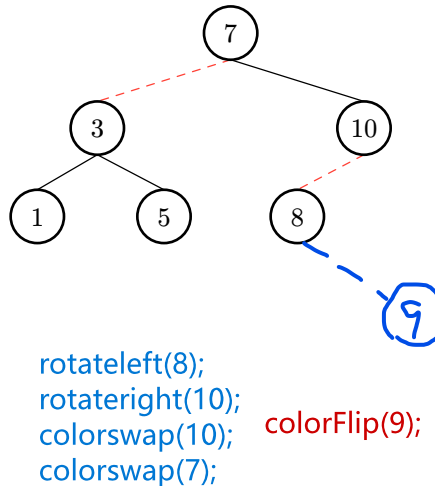
2 Hashing

(a) Here are five potential implementations of the **Integer** class's **hashCode()** method. Categorize each as
(1) invalid, (2) valid but not good, and (3) valid and good. If it is invalid, explain why. If it is valid
but not good, point out a flaw or disadvantage. For the 2nd implementation, note that **intValue()** will
return that **Integer**'s number value as an **int**, and assume that **Integer**'s **equals** method checks for
equality of the compared **Integer**s' **intValue**s.

```java
public int hashCode() {                    1      2
    return -1;
}
```

```java
public int hashCode() {                    2
    return intValue() * intValue();
}
```

```java
public int hashCode() {                         1
    return super.hashCode(); // Object's hashCode() is based on memory location
}
```

```java
public int hashCode() {                         1
    return (int) (new Date()).getTime(); // returns the current time as an int
}
```

```java
public int hashCode() {                    3
    return intValue() + 3;
}
```

(b) For each of the following questions, answer **Always**, **Sometimes**, or **Never**.

1. If you were able to modify a key that has been inserted into a **HashMap** would you be able to retrieve
that entry again later? For example, let us suppose you were mapping ID numbers to student names,
and you did a **put(303, "Elisa")** operation. Now, let us suppose we somehow went to that item in
our **HashMap** and manually changed the key to be 304. If we later do **get(304)**, will we be able to find
and return **"Elisa"**? Explain.

   sometimes

2. When you modify a value that has been inserted into a **HashMap** will you be able to retrieve that entry
again? For example, in the above scenario, suppose we first inserted **put(303, "Elisa")** and then
changed that item's value from **"Elisa"** to **"Daniel"**. If we later do **get(303)**, will we be able to find
and return **"Daniel"**? Explain.

   always

3 A Side of Hash Browns

We want to map food items to their yumminess. We want to be able to find this information in constant time, so we've decided to use Java's built-in **HashMap** class! Here, the key is an **String** representing the food item and the value is an **int** yumminess rating.

For simplicity, let's say that here a **String**'s hashcode is the first letter's position in the alphabet (A = 0, B = 1... Z = 25). For example, the **String "Hashbrowns"** starts with "H", and "H" is 7th letter in the alphabet (0 indexed), so the hashCode would be 7. Note that in reality, a **String** has a much more complicated **hashCode** implementation.

Our **HashMap** will compute the index as the key's hashcode value modulo the number of buckets in our **HashMap**. Assume the initial size is 4 buckets, and we double the size of our **HashMap** as soon as the load factor reaches 3/4. If we try to put in a duplicate key, simply replace the value associated with that key with the new value.

(a) Draw what the **HashMap** would look like after the following operations.

```java
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```

0   12

1   Brown   ...   13

2   14

3   Dim   15

4   Escar

5

6

7   Hash

8

9

10

11
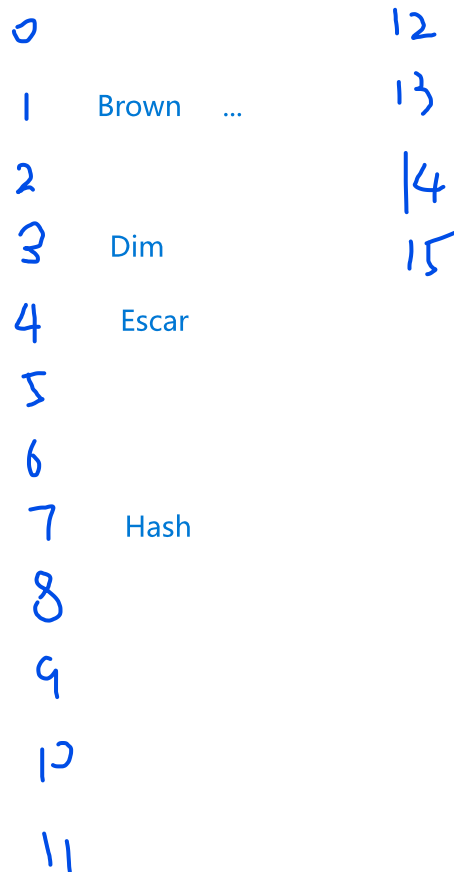
(b) Do you see a potential problem here with the behavior of our **HashMap**? How could we solve this?

Solution:
Here, adding a bunch of food items that start with the letter "B" result in one bucket with a lot of i
No matter how many times we resize, our current hashCode will result in this problem! Imagine if
added in 100 more items that started with the letter b. Though we would resize and keep our loac
low, it wouldn't change the fact that our operations will now be slow (hint: linear time) because n
essentially have to iterate over a linked list with pretty much all the items in the hashMap to do th
get("Burrito"), for example.
A solution to this would be to have a better hashCode implementation. A better implementation \
distribute the Strings more randomly and evenly. While knowing how to write such a hashCode is
and out of scope for this class, you can look at the real hashCode implementation for Java Strings
curious!