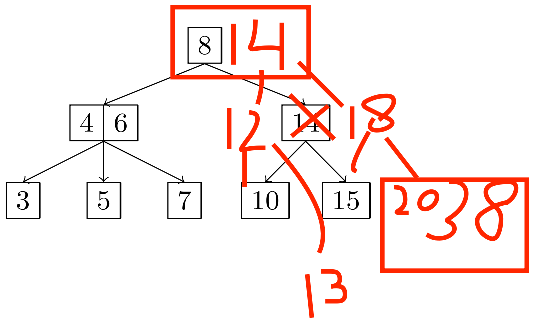
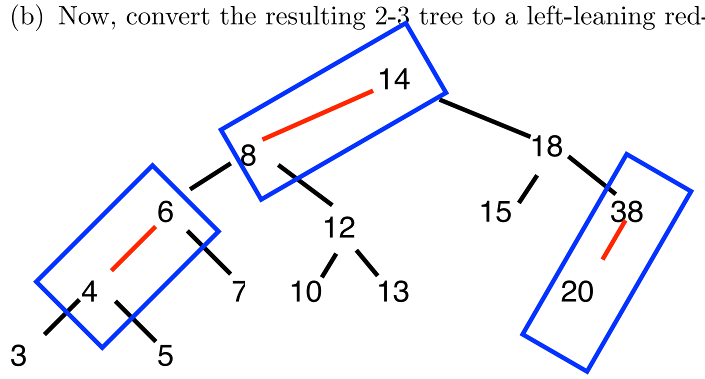
CS 61B | LLRBs, Hashing
Spring 2020 | Discussion 8: March 8, 2021

1 2-3 Trees and LLRB's

(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.



You can draw boxes around 2 nodes to see the correspondence between 2-3 and LLRBs

(c) If a 2-3 tree has depth H (that is, there are H number of edges in the path from leaf to 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?

Take the tree above with H=2 in 2-3 form, we want to find 5. Traversing the RB

Compare 5 < 14, go left
Compare 5 < 8, go left
Compare 5 < 6, go left
Compare 5 > 4, go right
Compare 5 == 5. Done
But assume we could also have another red branch, which gives us 6 operations
So the maximun numbers of is 2H + 2

2 Hashing

(a) Here are three potential implementations of the `Integer`'s `hashCode()` function. Categorize each as either a valid or an invalid hash function. If it is invalid, explain why. If it is valid, point out a flaw or disadvantage. For the 2nd implementation, note that `intValue()` will return that `Integer`'s number value as an `int`.

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

This is technically valid, but everything collides, so we just have a list. But in theory, the set and map operations will make sure we never get duplicates, meaning its just a super inefficient hashSet/map

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

Valid, but its flaw is there will be collisions for -5 and 5

```
public int hashCode() {
    return super.hashCode();
}
```

Invalid, super.hashCode() will return to the Object's location in memory. Two Integer objects with value 5 will have different location's in memory, so we get duplicates

(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, "Anjali")` 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 `"Anjali"`? Explain.

Sometimes,
No if, "Anjali".hashcode() = 303, when we look through bucket 304 and call "Anajali".hashCode().equals(304) we get False
Yes if Hashcode has nothing to do with "Anjali"'s properties, we can manually change it

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, "Anjali")` and then changed that item's value from `"Anjali"` to `"Ajay"`. If we later do `get(303)`, will we be able to find and return `"Ajay"`? Explain.

Always, if are now directly looking in Bucket 304 -> you will find Ajay in there.

Hashmaps are always sorted by their key

3 A Side of Hashbrowns

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` "hashbrown" 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 our HashMap as soon as the load factor reaches 3/4.

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

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

0	
1	[["Brown bananas", 1], ["Burritos", 10], ["Buffalo wings", 8], ["Banh mi", 9]]
2	
3	[Dim Sum,10]
4	[[Escargot, 5]]
5	
6	
7	[[Hashbrown, 7]]
8	
9	
10	
11	
12	
13	
14	
15	

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

It will continue to grow beyond size 25 (Z), but any letter can still get arbitrarily large. Its an inefficient implementation.

The load factor is scaling beyond Z, but we aren't retrieving items any quicker (say from B)

Solve by using hashcode on the first two letters, or first n numbers