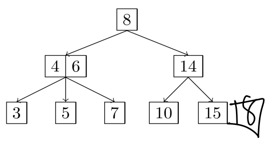
CS 61B
Spring 2020

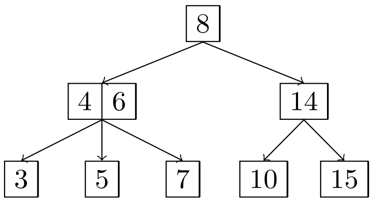
LLRBs, Hashing

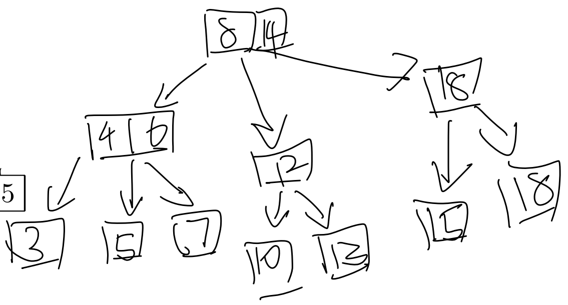
Discussion 8: March 8, 2021

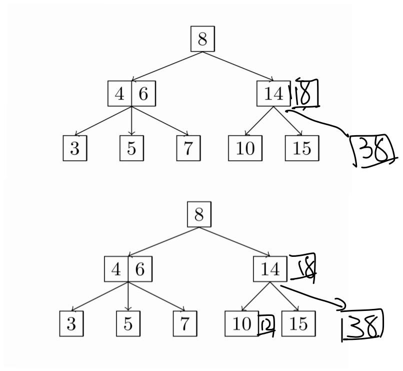
1 2-3 Trees and LLRB's

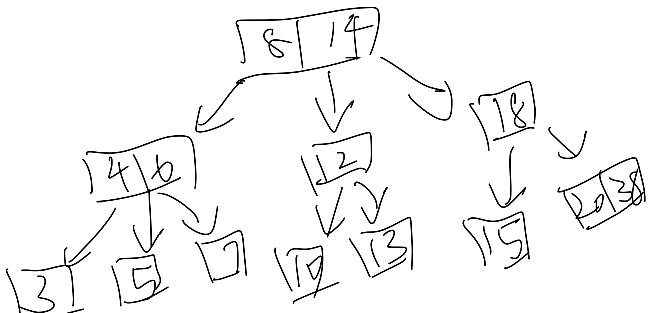
(a) Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.



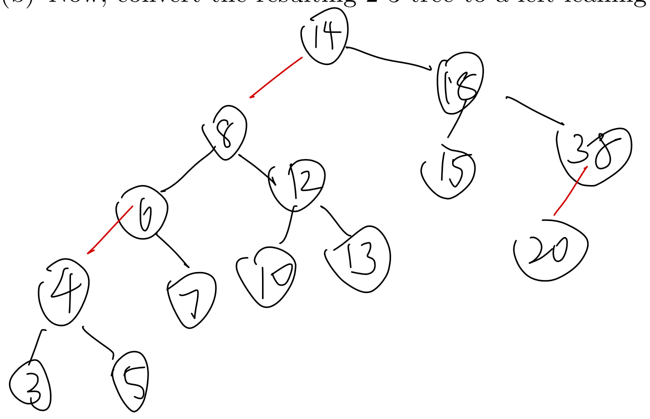








(b) Now, convert the resulting 2-3 tree to a left-leaning red-black tree.



(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?

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`.

Shortcoming

valid

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

valid

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

Invalid. When we call super.hashCode() here, we will ultimately end up returning Object.hashCode(). This hash function returns some number corresponding to the Integer object's location in memory.

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

(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

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

3 A Side of Hashbrowns

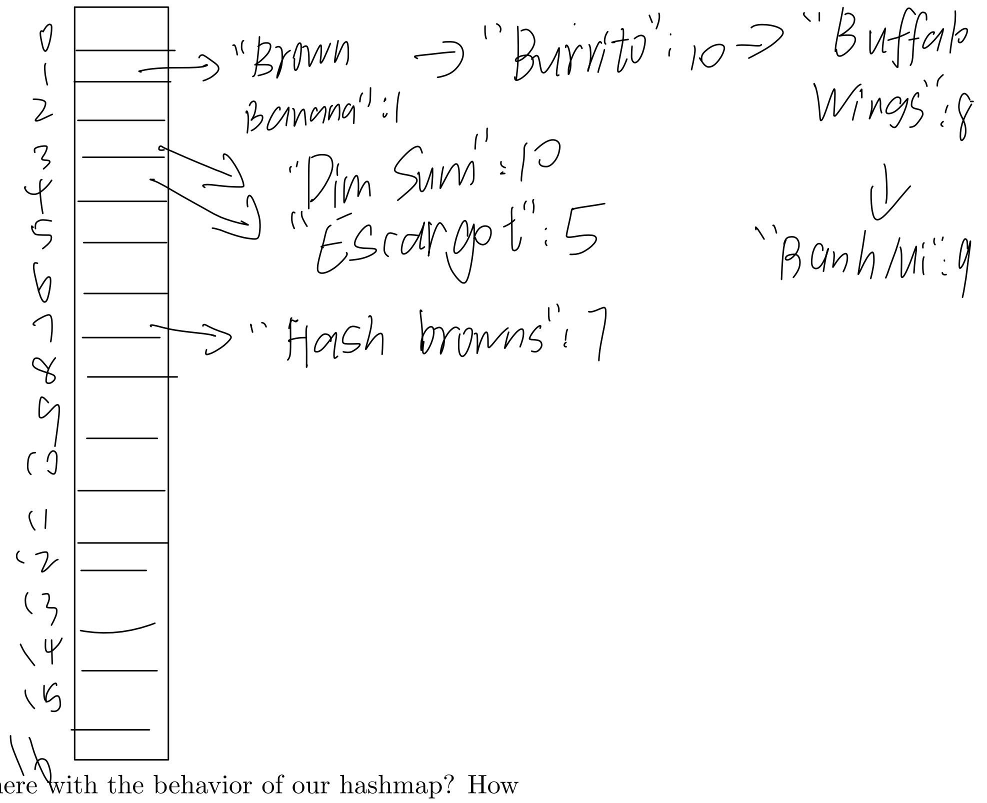
We want to map food items to their yumminess. We want to be able to find this information in constant time, so we've decided to use java's built-in HashMap class! Here, the key is an `String` representing the food item and the value is an `int` yumminess rating.

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);
```



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

Here, adding a bunch of food items that start with the letter "B" result in one bucket with a lot of items. No matter how many times we resize, our current hashCode( ) will result in this problem! Imagine if we added in 100 more items that started with the letter b. Through we would resize and keep our load factor low, it wouldn't change the fact that our operations will now be slow (hint: linear time) because now we essentially have to iterate over a linked list with pretty much all the items in the hashMap to do things like get ("Burrito"), for example.

A solution to this would be to have a better hash Code() implementation. A better implementation would distribute the Strings more randomly and evenly. While knowing how to write such a hashCode() is difficult and out of scope for this class, you can look at the real hashCode(). implementation for Java Strings if you're curious!