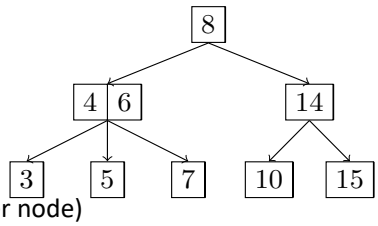
CS 61B Spring 2020 — LLRBs, Hashing

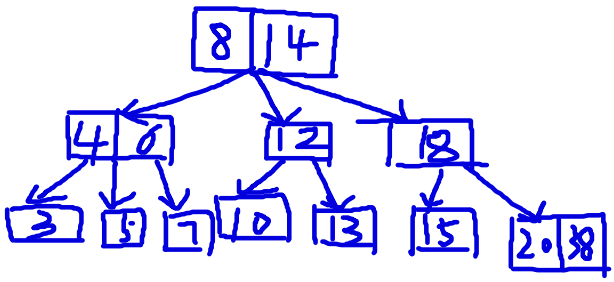
Discussion 8: March 8, 2021

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.



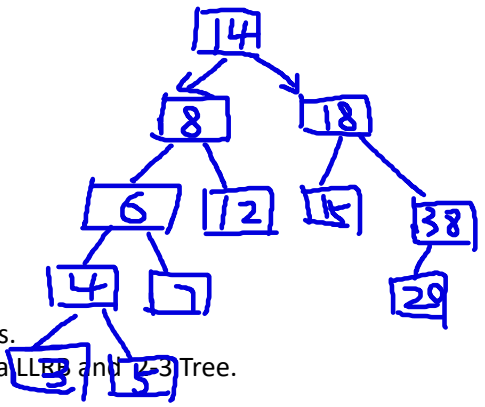
L表示B-Trees的阶数(max L items per node)
B-Trees: (1) 2-3 Trees (L=2)
节点可能有2或3个孩子
(2) 2-3-4 Trees/2-4 Trees (L=3)
节点可能有2或3或4个孩子
(3) L is very large (say thousands)



B-Trees的不变量：(1) 所有叶子节点到跟的距离大小相等
(2) 有k个items的非叶子节点必有k+1个孩子

B-Trees的高度: between logL+1(N) ~ log2(N)
B-Trees的时间复杂度: O(logN)

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



LLRBs(Left-leaning Red Black Binary Search Trees)
LLRBs: (1) LLRBs是正常的BSTs
(2)一个LLRB 与一个等价的 2-3 Tree之间存在一一对应关系

LLRBs的不变量: (1) No node has two red links.
(2) Every path from root to a leaf has same number of black links.
LLRBs的实现: Use zero or more rotations to maintain the 1-1 mapping between a LLRB and a 2-3 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

- When inserting: Use a red link.
- If there is a *right leaning "3-node"*, we have a **Left Leaning Violation.**
  - Rotate left the appropriate node to fix.
- If there are *two consecutive left links*, we have an **Incorrect 4 Node Violation.**
  - Rotate right the appropriate node to fix.
- If there are any nodes with two red children, we have a **Temporary 4 Node.**
  - Color flip the node to emulate the split operation.

One last detail: Cascading operations.

- It is possible that a rotation or flip operation will cause an additional violation that needs fixing.

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;
}
```

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

```
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.

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.

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.

```
1 HashMap<Integer, String> hm = new HashMap<>();
2 hm.put("Hashbrowns", 7);
3 hm.put("Dim sum", 10);
4 hm.put("Escargot", 5);
5 hm.put("Brown bananas", 1);
6 hm.put("Burritos", 10);
7 hm.put("Buffalo wings", 8);
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?