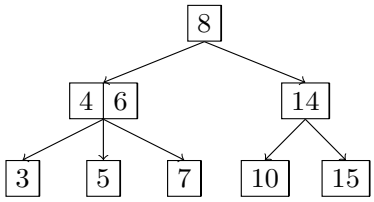
CS 61B Spring 2019

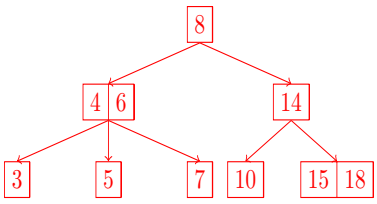
LLRBs, Hashing, Heaps

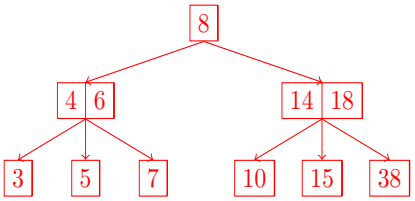
Discussion 8: March 11, 2019

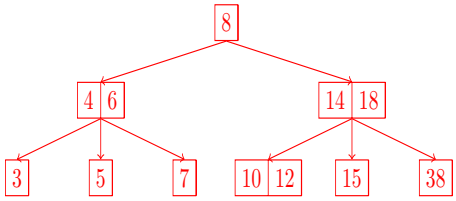
1 2-3 Trees and LLRB's

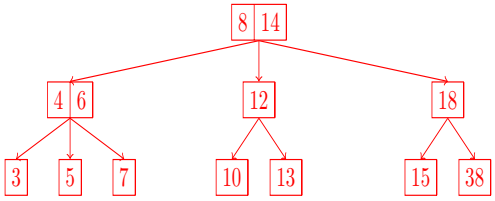
1.1 Draw what the following 2-3 tree would look like after inserting 18, 38, 12, 13, and 20.

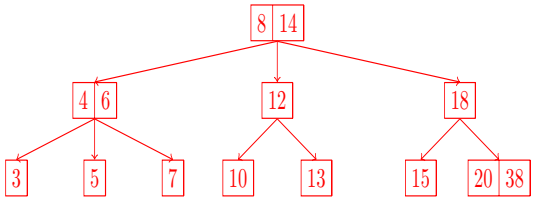




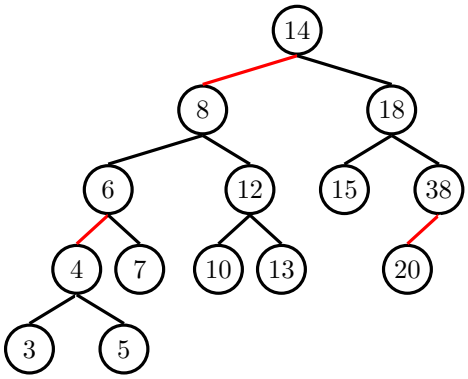








1.2 Now, convert the resulting 2-3 tree to a left-leaning red-black tree.



1.3 *Extra:* 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?

$2H + 2$ comparisons.

The maximum number of comparisons occur from a root to leaf path with the most nodes. Because the height of the tree is H, we know that there is a path down the leaf-leaning red-black tree that consists of at most $H + 1$ black links, for black links in the left-leaning red-black tree are the links that add to the height of the corresponding 2-3 tree.

In the worst case, in the 2-3 tree representation, this path can consist entirely of nodes with two items, meaning in the left-leaning red-black tree representation, each blank link is followed by a red link. This doubles the amount of nodes on this path from the root to the leaf.

This example will represent our longest path, which is $2H + 2$ nodes long, meaning we make at most $2H + 2$ comparisons in the left-leaning red-black tree.

2 Hashing

2.1 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.

```
1 public int hashCode() {
2     return -1;
3 }
```

Valid. As required, this hash function returns the same `hashCode` for Integers that are `equals()` to each other. However, this is a terrible hash code because collisions are extremely frequent (collisions occur 100% of the time).

```
1 public int hashCode() {
2     return intValue() * intValue();
3 }
```

Valid. Similar to (a), this hash function returns the same `hashCode` for integers that are `equals()`. However, integers that share the same absolute values will collide (for example, $x = 5$ and $x = -5$ will have the same hash code). A better hash function would be to just return the `intValue()` itself.

```
1 public int hashCode() {
2     return super.hashCode();
3 }
```

Invalid. This is not a valid hash function because integers that are `equals()` to each other will not have the same hash code. Instead, this hash function returns some integer corresponding to the integer object's location in memory.

2.2 *Extra, but highly recommended:* For each of the following questions, answer **Always**, **Sometimes**, or **Never**.

(a) When you modify a key that has been inserted into a `HashMap` will you be able to retrieve that entry again? Explain.

Sometimes. If the `hashCode` for the key happens to change as a result of the modification, then we won't be able to reliably retrieve the key.

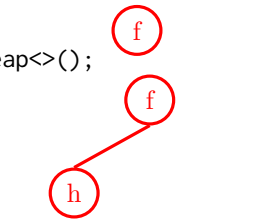
(b) When you modify a value that has been inserted into a HashMap will you be able to retrieve that entry again? Explain.

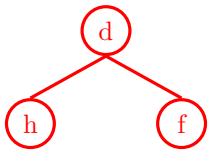
Always. The bucket index for an entry in a `HashMap` is decided by the key, not the value. Mutating the value does not affect the lookup procedure.

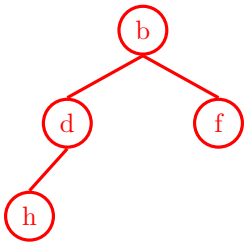
3 Heaps of Fun

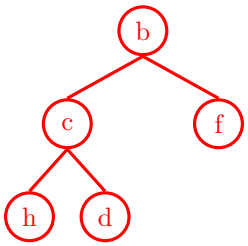
3.1 Assume that we have a binary min-heap (smallest value on top) data structure called `Heap` that stores integers, and has properly implemented `insert` and `removeMin` methods. Draw the heap and its corresponding array representation after each of the operations below:

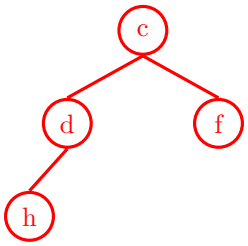
```
1 Heap<Character> h = new Heap<>();
2 h.insert('f');
3 h.insert('h');
4 h.insert('d');
5 h.insert('b');
6 h.insert('c');
7 h.removeMin();
8 h.removeMin();
```

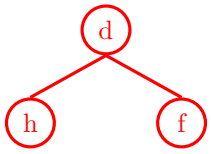












3.2 Your friendly TA Tina challenges you to quickly implement an integer max-heap data structure. "Hah! I'll just use my min-heap implementation as a template to write MaxHeap.java," you think to yourself. Unfortunately, due to following the instructions of a shady stackoverflow post, you manage to permanently delete your `MinHeap.java` file. Luckily, you notice that you still have `MinHeap.class`. Can you still complete the challenge before time runs out?

Hint: Although you cannot alter them, you can still use methods from `MinHeap`.

Yes. For every insert operation, negate the number and add it to the min-heap.

For a `removeMax` operation call `removeMin` on the min-heap and negate the number returned. Any number negated twice is itself (with one exception in Java, 2^{-31}), and since we store the negation of numbers, the order is now reversed (what used to be the max is now the min).