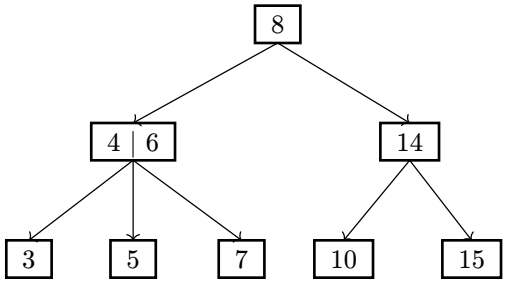
CS 61B
Fall 2025

B-Trees, LLRBs, and Heaps

Discussion 07: October 13, 2025

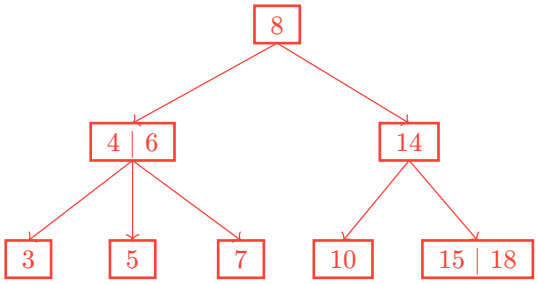
1 2-3 Trees and LLRBs

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

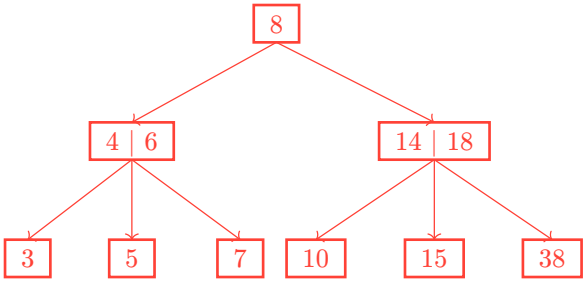


Solution:

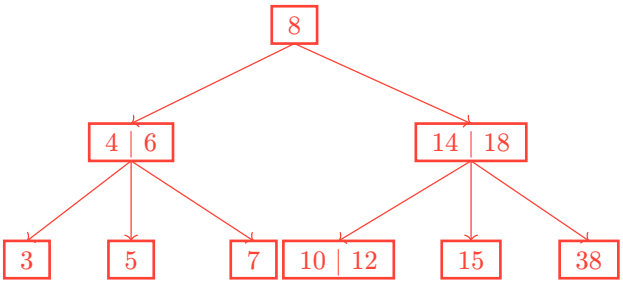
Adding 18:



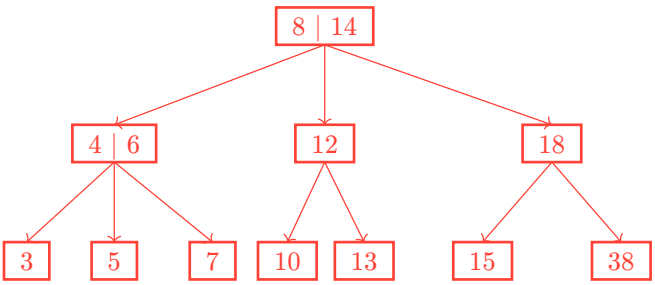
Adding 38:



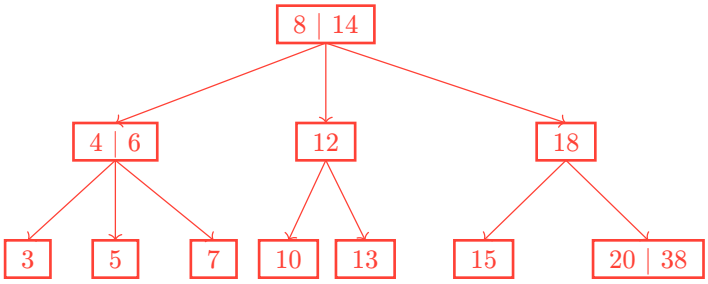
Adding 12:



Adding 13:

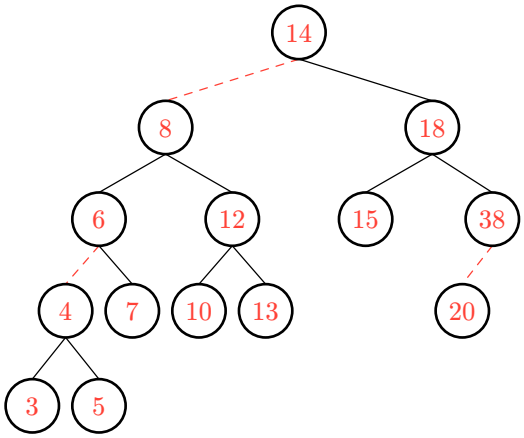


Adding 20:



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

Solution:



(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?

Solution:

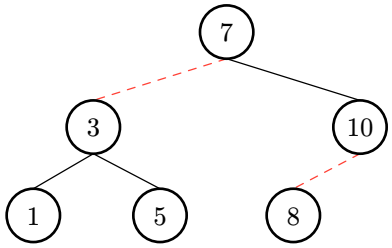
$2H + 2$ comparisons.

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 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. This means that there are $H + 1$ nodes on the path from the root to the leaf, since there is one less link than nodes.

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 black link is followed by a red link. This doubles the amount of nodes on this path from the root to the leaf.

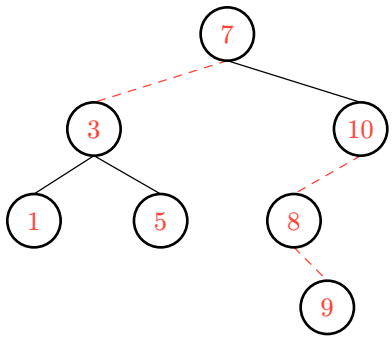
This example will represent our longest path, which is $2H + 2$ nodes long, meaning we make at most $2H + 2$ comparisons in the left-leaning red-black tree.

(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.

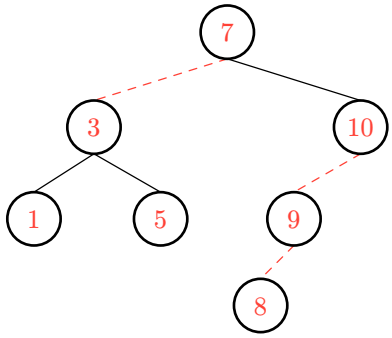


Solution:

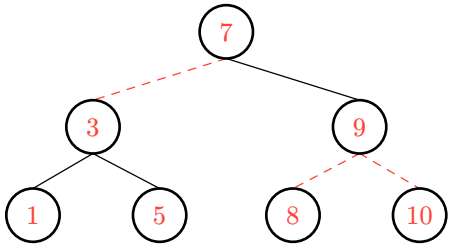
Remember that we always insert elements as leaf nodes with red links, so after initial insertion, the tree looks like:



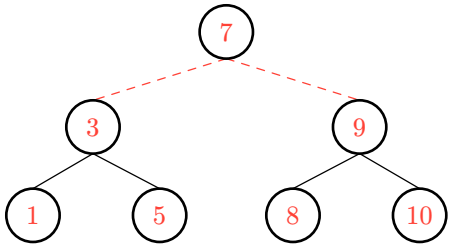
Now there is a right-leaning red link, so we will have to `rotateLeft(8)`:



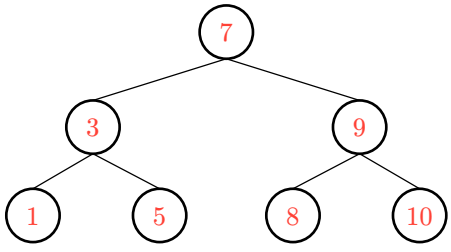
With the previous rotation, we now have two consecutive red left links, which makes our tree unbalanced. To fix, we `rotateRight(10)`:



Next, we have red links on both the left and the right, which can be fixed with a `colorFlip(9)`:



Finally, we still have red links on both the left and the right, which can be fixed with a `colorFlip(7)`:



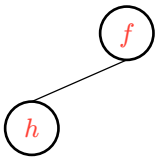
2 Absolutely Valuable Heaps

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

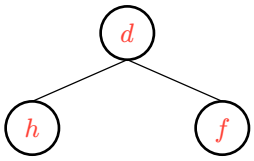
```
MinHeap<Character> h = new MinHeap<>();
h.insert('f');
h.insert('h');
h.insert('d');
h.insert('b');
h.insert('c');
h.removeMin();
h.removeMin();
```

after inserting 'f': `[-, 'f']`

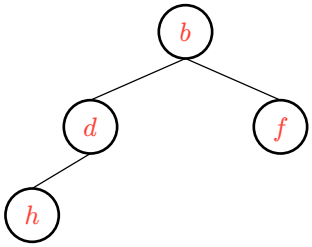
after inserting 'h': `[-, 'f', 'h']`



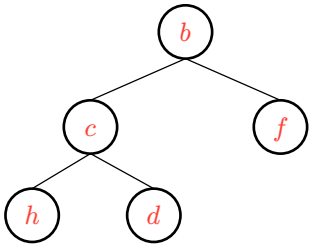
after inserting 'd': `[-, 'd', 'h', 'f']`



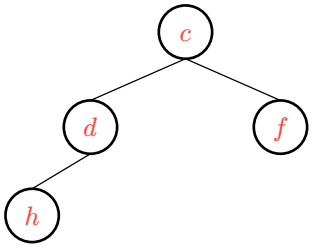
after inserting 'b': `[-, 'b', 'd', 'f', 'h']`



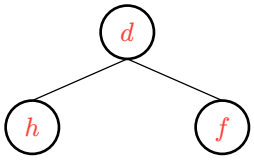
after inserting 'c': `[-, 'b', 'c', 'f', 'h', 'd']`



after removing min: `[-, 'c', 'd', 'f', 'h']`



after removing min: `[-, 'd', 'h', 'f']`



(b) Your friendly TA Mihir challenges you to create an integer max-heap without writing a whole new data structure. Can you use your min-heap to mimic the behavior of a max-heap? Specifically, we want to be able to get the largest item in the heap in constant time, and add things to the heap in $\Theta(\log n)$ time, as a normal max heap should.

Hint: You should treat the `MinHeap` as a black box and think about how you should modify the arguments/ return values of the heap functions.

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, and since we store the negation of numbers, the order is now reversed (what used to be the max is now the min).

Small note: There's actually one exception in Java to what we said about negation above: -2^{-31}, the most negative number that we can represent in Java, will not be itself when negated twice. This is mostly due to number representation constraints in code, but you don't need to worry about that for this question.

3 Extra: Heap Mystery

We are given the following array representing a min-heap where each letter represents a **unique** number. Assume the root of the min-heap is at index one, i.e. `A` is the root. Our task is to figure out the numeric ordering of the letters. Therefore, there is **no** significance of the alphabetical ordering. i.e. just because B precedes C in the alphabet, we do not know if B is less than or greater than C.

Array: [-, A, B, C, D, E, F, G]

Four unknown operations are then executed on the min-heap. An operation is either a `removeMin` or an `insert`. The resulting state of the min-heap is shown below.

Array: [-, A, E, B, D, X, F, G]

(a) Determine the operations executed and their appropriate order. The first operation has already been filled in for you!

Hint: Which elements are gone? Which elements are newly added? Which elements are removed and then added back?

1. `removeMin()`
2. `insert(X)`
3. `removeMin()`
4. `insert(A)`

Explanation: We know immediately that A was removed. Then, after looking at the final state of the min-heap, we see that C was removed. Then, for A to remain in the min-heap, we see that A must have been inserted afterwards. And, after seeing a new value X in the min-heap, we see that X must have been inserted as well. We just need to determine the relative ordering of the `insert(X)` in between the operations `removeMin()` and `insert(A)`, and we see that the `insert(X)` must go before both.

(b) Fill in the following comparisons with either $>$, $<$, or ? if unknown. We recommend considering which elements were compared to reach the final array.

1. X `?` D
2. X `>` C
3. B `>` C
4. G `<` X

Reasoning:
1. X is never compared to D
2. X must be greater than C since C is removed after X's insertion.
3. B must also be greater than C otherwise the second call to `removeMin` would have removed B
4. X must be greater than G so that it can be "promoted" to the top after the removal of C. It needs to be promoted to the top to land in its new position.