Lab 7

Left-Leaning Red Black Trees (LLRBs)

Announcements

Project 2A is due Wednesday, 3/06 at 11:59 pm.

Project 2B will be due Monday, 4/01 at 11:59 pm (this is the Monday after spring break!).

2-3 Trees (Balanced Search Trees)

Binary Search Trees

Last time, we talked about binary search trees and how it made it much more efficient to look up items and insert them - but there are still some problems with them.

Binary Search Trees

Last time, we talked about binary search trees and how it made it much more efficient to look up items and insert them - but there are still some problems with them.

We can't guarantee that binary search trees will always have a "bushy" structure. In the worst case, we would have a "spindly" tree.

- As a reminder, a bushy structure would have a worst case of O(logN) and a spindly tree would have a worst case runtime of O(N) (and this applies for insertion and lookup).

Balanced Search Trees

Main Idea: Instead of adding a new node for each inserted element, we want to put multiple elements into a single node.

- Simply put, we always insert an element into an **existing node** and if the node becomes too big, we "split" the node.

Balanced Search Trees

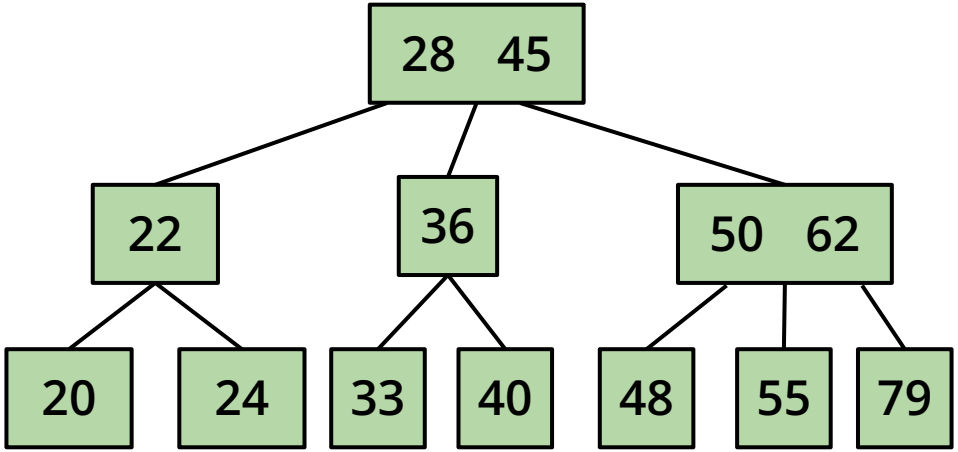
Main Idea: Instead of adding a new node for each inserted element, we want to put multiple elements into a single node.

- Simply put, we always insert an element into an **existing node** and if the node becomes too big, we "split" the node.

While there are several variants of a balanced search tree, we'll go with a 2-3 tree for this lab. What this means is that:

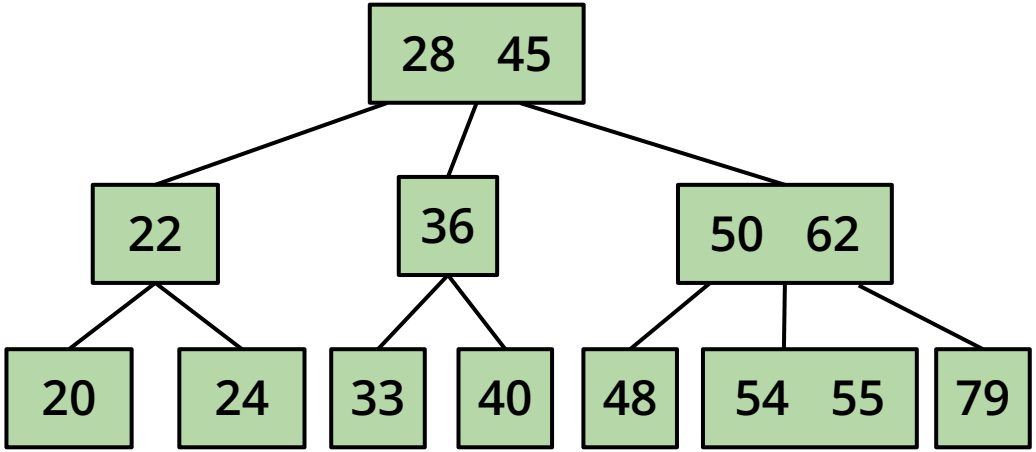
- We can only have **2 items max per node**.
- And each node can have **3 children max**.

Balanced Search Trees



If we insert 54, where would that go?

Balanced Search Trees



It would go into the same node as 55!

Left-Leaning Red Black Trees

Now that we have balanced search trees down, let's introduce the left-leaning red black trees. In practice, balanced search trees are actually kind of hard to implement.

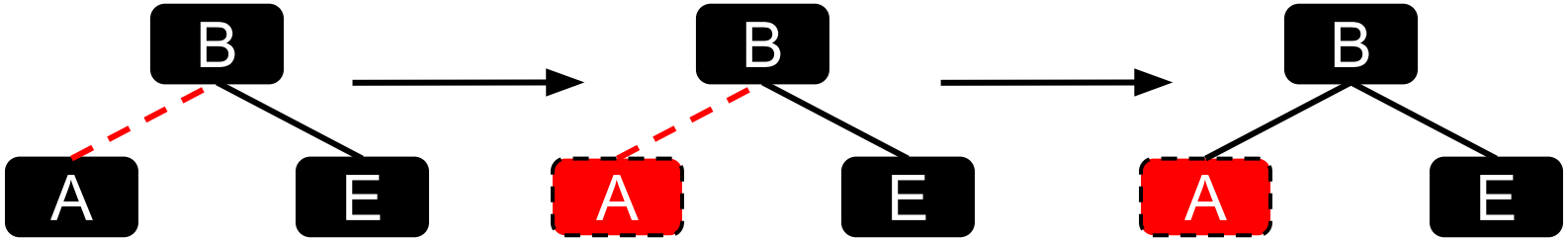
Left-Leaning Red Black Trees (LLRBs)

LLRBs

LLRBS are:

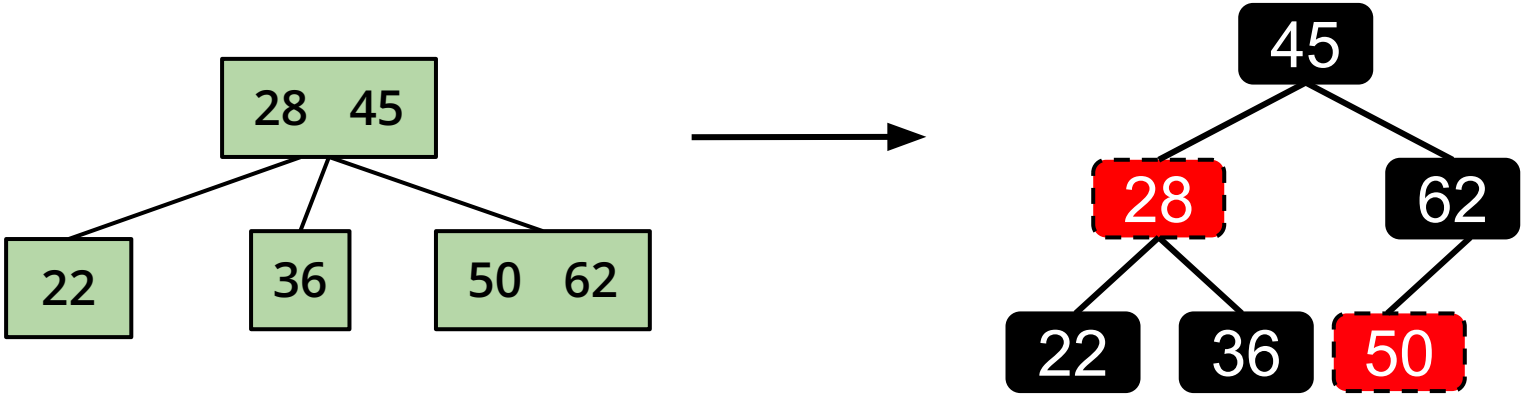
- A 2-3 tree represented in a binary search tree format (one item per node)
- This data structure is represented by two different types of nodes:
  - A **red** node means that the node is squished in the same node as the parent (in the 2-3 tree representation)
  - A **black** node means just a regular node in the 2-3 tree.

Links vs Nodes (Disclaimer)



Disclaimer: You may have wondered at this point why we're using colored nodes. In lecture, we introduce LLRBs with links. That is to say, if two nodes are connected with a red link, we say that the child node belongs in the same node as the parent in its 2-3 tree representation. For ease of implementation, the lab uses nodes instead (for the rest of lab, we'll use colored nodes).

LLRBs



Main Invariants (for Lab)

If a node has one red child, it must be on the left (i.e. all red nodes are left-leaning)

Main Invariants (for Lab)

If a node has one red child, it must be on the left (i.e. all red nodes are left-leaning)

No node can have two red children.

Main Invariants (for Lab)

If a node has one red child, it must be on the left (i.e. all red nodes are left-leaning)

No node can have two red children.

No red node can have a red parent (every red node's parent is black)

- This means that we can't have two left-leaning red nodes (as mentioned in lecture)
- Or a red right child of a red node (which is effectively no right-leaning red node).

Main Invariants (for Lab)

If a node has one red child, it must be on the left (i.e. all red nodes are left-leaning)

No node can have two red children.

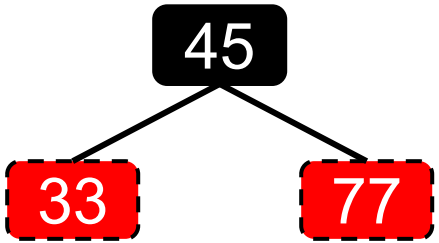
No red node can have a red parent (every red node's parent is black)

- This means that we can't have two left-leaning red nodes (as mentioned in lecture)
- Or a red right child of a red node (which is effectively no right-leaning red node).

As an important note, these red nodes come from insertion - whenever we insert a node into an LLRB, it's always as a red node.

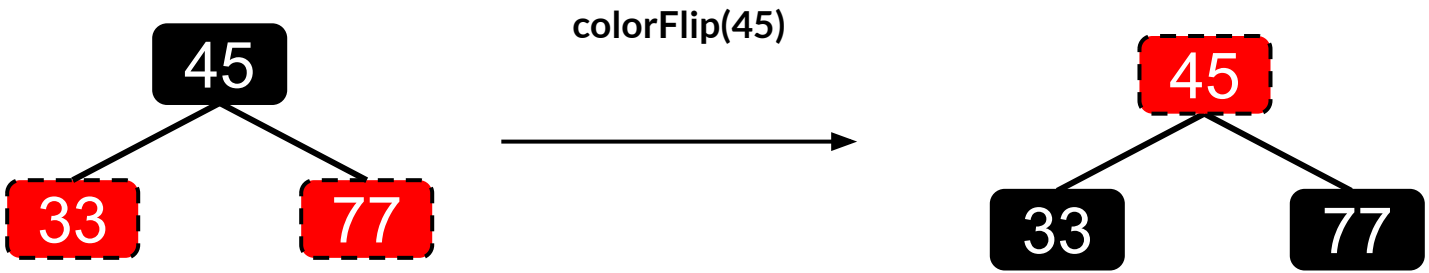
LLRB Operations

colorFlip



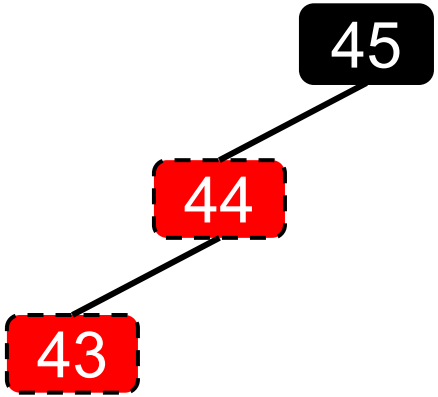
Which invariant is broken?

colorFlip



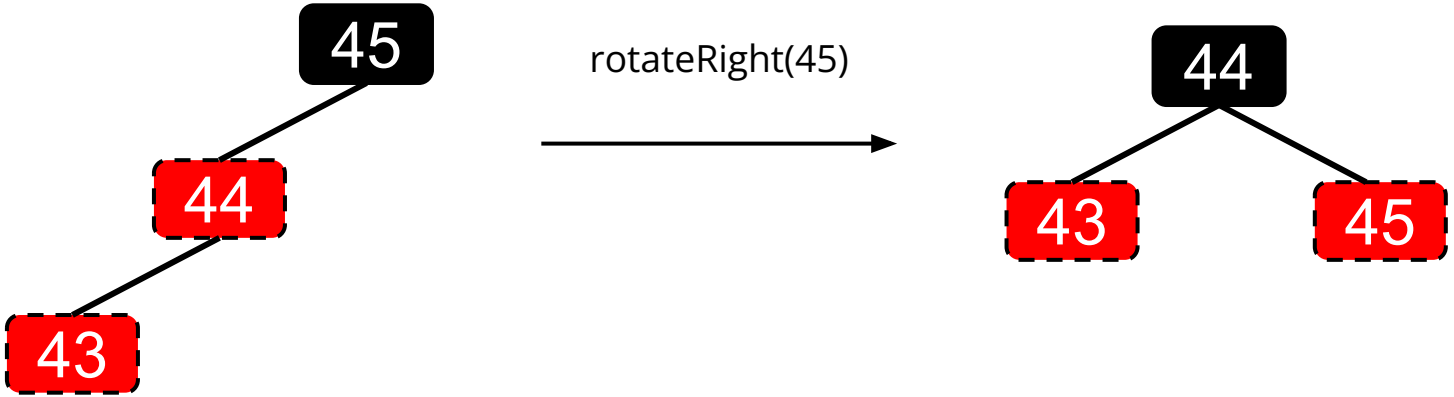
Invariant: No node can have two **red** children.

rotateRight



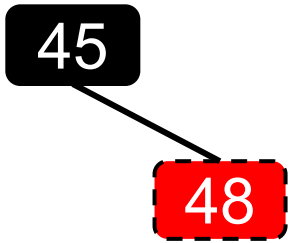
Which invariant is broken?

rotateRight



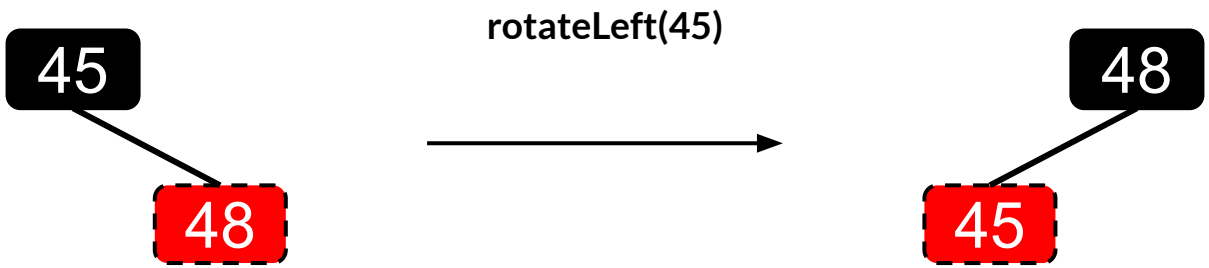
Invariant: The parent of a red node must be black (this means that we can not have two left-leaning red nodes).

rotateLeft



Which invariant is broken?

rotateLeft



Invariant: All red nodes must be left-leaning.

Lab Overview

An Overview

Lab 08 is due Friday, 3/08 at 11:59 pm.

Deliverables:

- Complete the following methods in `RedBlackTree.java`
  - `flipColors`
  - `rotateRight` and `rotateLeft`
  - `insert`

Lab Notes

The lab provides a Node class for you to use (`static class RBTreeNode<T>`). Make sure to read through it first to understand what it encompasses before starting on the lab! It is in `RedBlackTree.java`.

The syntax to declare the node object is not the same as Proj1A, but should be like below:

```
RBTreeNode<T> node = ...;
```