SEARCHING

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

20: Left-Leaning Red-Black Trees

Alexandra Papoutsaki
she/her/hers

Lecture 20: Left-leaning Red-Black Trees

- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

Some slides adopted from Algorithms 4th Edition or COS226

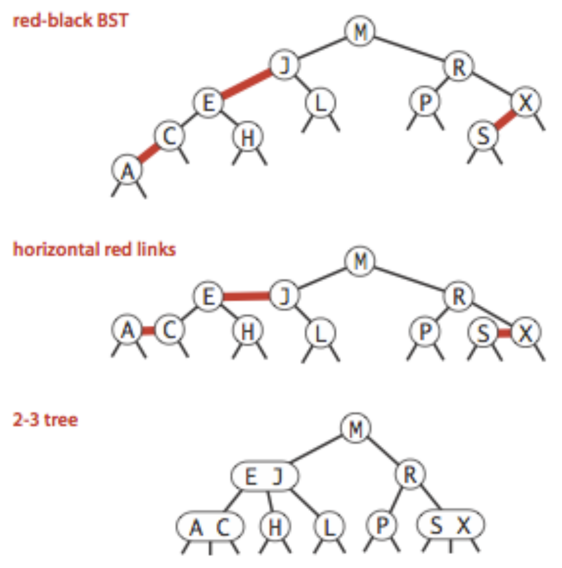
The insertion algorithm for 2-3 trees we saw last time is not difficult to understand but we said that can be hard to implement. We will instead consider a simple representation known as a red-black BST that leads to a natural implementation.

Left-leaning red-black BSTs correspond 1-1 with 2-3 trees

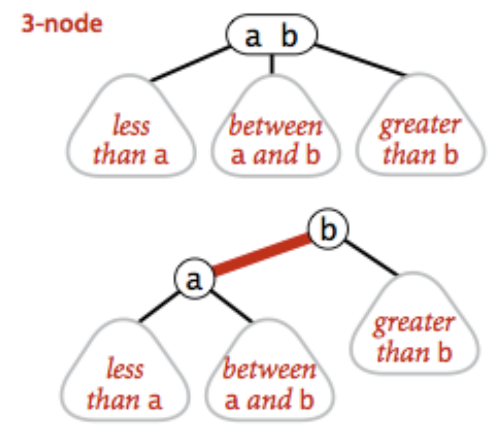
- Start with standard BSTs which are made up of 2-nodes.
- Add extra information to encode 3-nodes. We will introduce two types of links.
- Red links: bind together two 2-nodes to represent a 3-node.
  - Specifically, 3-nodes are represented as two 2-nodes connected by a single red link that leans left (one of the 2-nodes is the left child of the other).
- Black links: bind together the 2-3 tree.
- Advantage: Can use BST code with minimal modification.

A left-leaning red-black BST corresponds 1-1 with a 2-3 search tree. Essentially, we start with a standard BST made up of 2-nodes. Any extra information added is encoded in 3-nodes. For that, we will introduce two types of links. A red link binds together two 2-nodes to represent a 3-node. Specifically, 3-nodes are represented as two 2-nodes connected by a single red link that leans left (one of the 2-nodes is the left child of the other). A black link binds together 2-3 tree. The advantage of this approach is that we can reuse code from BST with minimal modification.

Left-leaning red-black BSTs correspond 1-1 with 2-3 trees



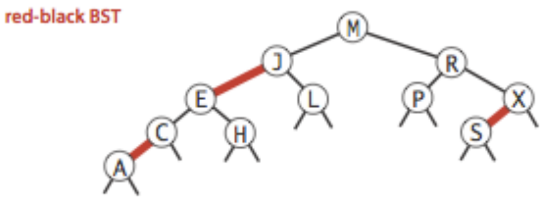
1-1 correspondence between red-black BSTs and 2-3 trees



Here's what I mean by this 1-1 correspondence. On the top left you can see a left-leaning red-black BST. If we consider the red links horizontally, we can quickly see how they translate to 3 nodes. Any 3 node can be translated into two nodes linked by a red link.

Definition

- A left-leaning red-black tree is a BST such that:
  - No node has two red links connected to it.
  - Red links lean left.
  - Every path from root to leaves has the same number of black links (perfect black balance).



Overall, a left-leaning red-black tree is a BST such that: No node has two red links connected to it.
Red links lean left. And every path from root to leaves has the same number of black links (that is known as perfect black balance).

Search

- Exactly the same as for elementary BSTs (we ignore the color).
  - But runs faster because of better balance.

```
public Value get(Key key) {
     if (key == null) throw new IllegalArgumentException("argument to get() is null");
     return get(root, key);
 }

 // value associated with the given key in subtree rooted at x; null if no such key
 private Value get(Node x, Key key) {
     while (x != null) {
         int cmp = key.compareTo(x.key);
         if      (cmp < 0) x = x.left;
         else if (cmp > 0) x = x.right;
         else              return x.val;
     }
     return null;
 }
```

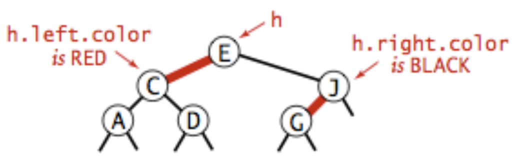
- Operations such as floor, iteration, rank, selection are also identical.

When it comes to searching for a value given a key, this is done exactly in the same way as with regular BSTs, i.e. we ignore the color of links. Because we have perfect balance, the height of the tree is shorter than what a regular BST would have. Operations such as floor, iteration, rank, selection are also identical.

Representation

- Each node is pointed to by one node, its parent. We can use this to encode the color of the links in nodes.
- True if the link from the parent is red and false if it is black. Null links are black.

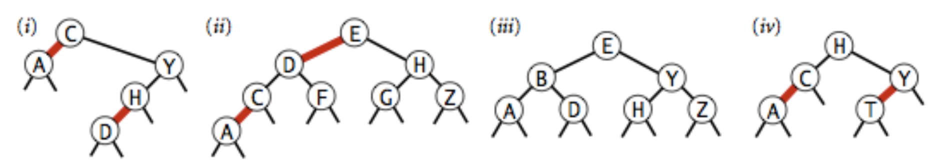
```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private Node root;     // root of the BST

// BST helper node data type
private class Node {
    private Key key;           // key
    private Value val;         // associated data
    private Node left, right;  // links to left and right subtrees
    private boolean color;     // color of parent link
    private int size;          // subtree count

private boolean isRed(Node x) {
    if (x == null) return false;
    return x.color == RED;
}
```



How do we actually represent a left-leaning red-black tree in code? Each node is pointed to by one node, its parent. We can use this to encode the color of the links in nodes. True if the link from the parent is red and false if it is black. Null links are black.

Practice Time

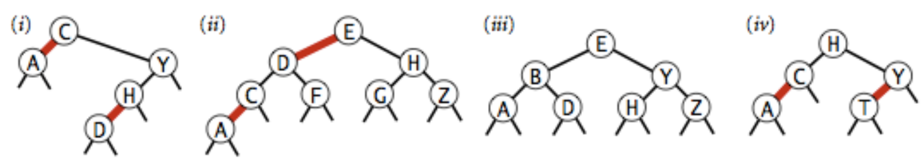
- Which of the following are legal LLRB trees?



Which of the following are legal LLRB trees? Remember you need to see perfect black balance and symmetric order.

Answer

- Which of the following are legal LLRB trees?
- iii and iv
  - i is not balanced and ii is also not in symmetrical order



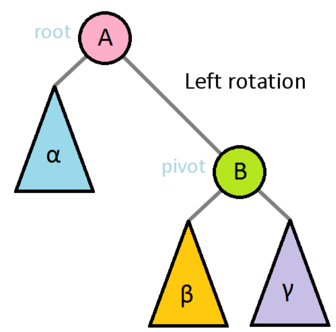
The answer is iii and iv. i is not balanced and ii is also not in symmetrical order

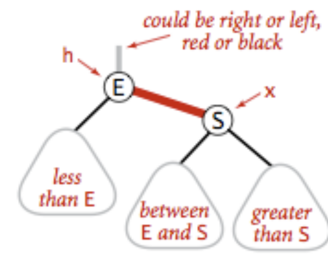
Lecture 20: Left-leaning Red-Black Trees

- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

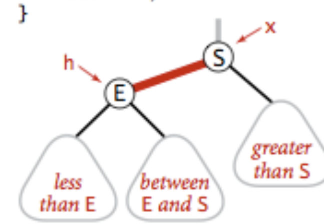
Let's look into some elementary operations that will allow us to insert information in a left-leaning red-black tree.

Left rotation: Orient a (temporarily) right-leaning red link to lean left





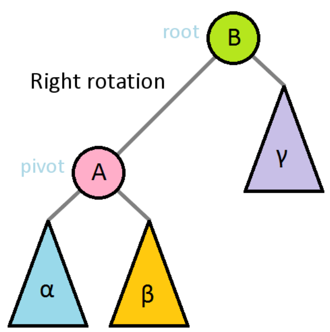

```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

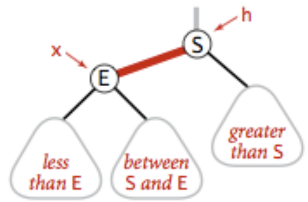


Left rotate (right link of h)

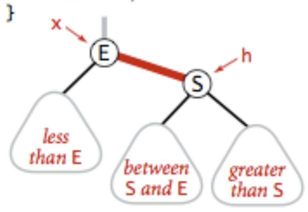
The first such operation is a left rotation. It orients a (temporarily) right-leaning red link to lean left. For example, the connection between E and S is initially right-leaning; we turn it into left leaning and make sure we exchange the left child of S as the right child of E.

Right rotation: Orient a left-leaning red link to a (temporarily) lean right





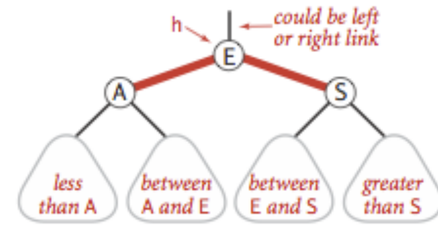

```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```



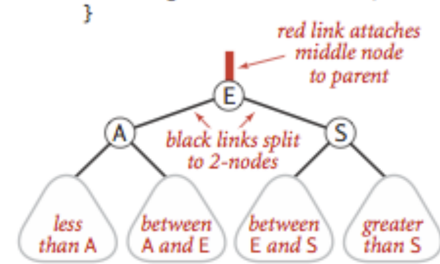
Right rotate (left link of h)

The second operation is the mirror concept of right rotation. It orients a left-leaning red link to (temporarily lean right. For example, the connection between E and S is initially left-leaning; we turn it into right leaning and make sure we exchange the right child of E as the left child of S.

Color flip: Recolor to split a (temporary) 4-node




```
void flipColors(Node h)
{
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```



Flipping colors to split a 4-node

The third elementary operation is a color flip. When we have a node with both links to its children being red (i.e. equivalently that would be a temporary 4-node), we just flip the color to turn the links to children to black and the link to parent to red.

Lecture 20: Left-leaning Red-Black Trees

- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

How do those three operations help us? They come handy during insertion.

Basic strategy: Maintain 1-1 correspondence with 2-3 trees

- During internal operations, maintain:
  - symmetric order
  - perfect black balance.
- But we might violate color invariants. For example:
  - Right-leaning red link.
  - Two red children (temporary 4-node).
  - Left-left red (temporary 4-node).
  - Left-right red (temporary 4-node).
- To restore color invariant we will be performing rotations and color flips.

Our basic strategy will be to maintain symmetric order and perfect black balance. Along the way, we might violate color invariants, e.g., Right-leaning red link.
Two red children (temporary 4-node).
Left-left red (temporary 4-node).
Left-right red (temporary 4-node).
To restore color invariant we will be performing rotations and color flips.

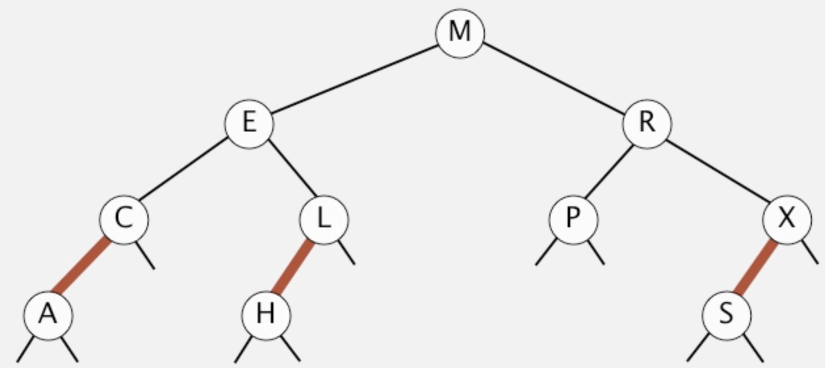
Insertion into a LLRB

- Do standard BST insertion and color the new link red.
- Repeat until color invariants restored:
  - Both children red? Flip colors.
  - Right link red? Rotate left.
  - Two left reds in a row? Rotate right.

To insert in a left-leaning red black BST, we will do a standard insertion and color the new link red. We will then repeat until color invariants are restored the following:
Both children red? Flip colors.
Right link red? Rotate left.
Two left reds in a row? Rotate right.

Red-black BST construction demo

red–black BST

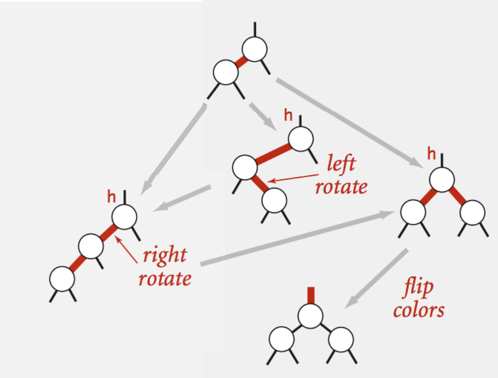


Here is a demonstration of all those steps.

Implementation

- Only three cases:
  - Right child red; left child black: rotate left.
  - Left child red; left-left grandchild red: rotate right.
  - Both children red: flip colors.

```
// insert the key-value pair in the subtree rooted at h
 private Node put(Node h, Key key, Value val) {
     if (h == null) return new Node(key, val, RED, 1);

     int cmp = key.compareTo(h.key);
     if      (cmp < 0) h.left  = put(h.left,  key, val);
     else if (cmp > 0) h.right = put(h.right, key, val);
     else              h.val   = val;

     // fix-up any right-leaning links
     if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
     if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
     if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);
     h.size = size(h.left) + size(h.right) + 1;

     return h;
 }
```



If you think about it, every insertion will only result to three cases:
Right child red; left child black: rotate left.
Left child red; left-left grandchild red: rotate right.
Both children red: flip colors.
This makes the implementation of it rather easy!

Visualization of insertion into a LLRB tree

- 255 insertions in ascending order.

Here is a visualization of 255 insertions in ascending order.

Visualization of insertion into a LLRB tree

- 255 insertions in descending order.

Here is a visualization of 255 insertions in descending order.

Visualization of insertion into a LLRB tree

- 255 insertions in random order.

And here is a visualization of 255 insertions in random order. Pretty short in all 3!

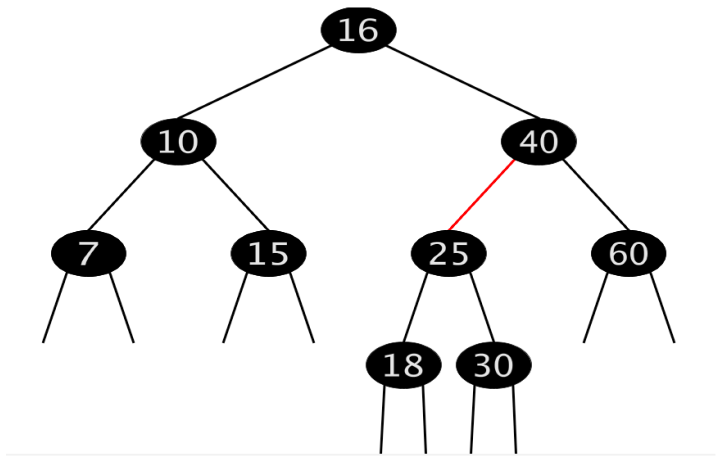
Practice Time - Worksheet #20

- Draw the LLRB tree that results when you insert the keys 10, 18, 7, 15, 16, 30, 25, 40, 60 in that order in an initially empty tree.

Let's get some practice.

ANSWER - Worksheet #20

- Draw the LLRB tree that results when you insert the keys 10, 18, 7, 15, 16, 30, 25, 40, 60 in that order in an initially empty tree.



Let's get some practice.

Lecture 20: Left-leaning Red-Black Trees

- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

How short?

Balance in LLRB trees

- Height of LLRB trees is $\leq 2 \log n$ in the worst case.
- Worst case is a 2-3 tree that is all 2-nodes except that the left-most path is made up of 3-nodes.
- All ordered operations (min, max, floor, ceiling) etc. are also $O(\log n)$.

The heigh of a left leaning red black BST is guaranteed to be at most 2logn in the worst case. Worst case is a 2-3 tree that is all 2-nodes except that the left-most path is made up of 3-nodes.
That means that insertion, deletion, and all ordered operations (min, max, floor, ceiling) etc. are also logarithmic!

Summary for dictionary operations

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
BST	n	n	n	$\log n$	$\log n$	$\sqrt{n}$
2-3 search tree	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
Red-black BSTs	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$

Which brings us here in the cost for implementing dictionaries.

Lecture 20: Left-leaning Red-Black Trees

- Introduction
- Elementary red-black BST operations
- Insertion
- Mathematical analysis
- Historical context

Let's look into some historical context about using search trees.

Red-black trees

- A dichromatic framework for balanced trees. [Guibas and Sedgewick, 1978].
- Why red-black? Xerox PARC had a laser printer and red and black had the best contrast…
- Left-leaning red-black trees [Sedgewick, 2008]
  - Inspired by difficulties in proper implementation of RB BSTs.
- RB BSTs have been involved in lawsuit because of improper implementation.

Red-black trees were created in 1978 by Guibas and Sedgewick (the author of our recommended textbook). You might be wondering why those colors. Xerox PARC had a laser printer and red and black had the best contrast… Sometimes history is made out of convenience. Sedgewick in 2008 developed left-leaning red-black trees inspired by difficulties in proper implementation of red black binary search trees. Red black binary search trees are very famous but also hard to implement which led to a lawsuit when a telephone company contracted with database provider to build real-time database to store customer information.
Database implementation. They chose red-black BSTs but the improper implementation left to high trees which led to service outages. As a result, the telephone company sued the database provider and Sedgewick had to provide legal testimony about the implementation of the red-black trees.

Balanced trees in the wild

- Red-black trees are widely used as system dictionaries.
  - e.g., Java: `java.util.TreeMap` and `java.util.TreeSet`.
- Other balanced BSTs: AVL, splay, randomized.
- 2-3 search trees are a subset of b-trees.
  - See recommended textbook for more.
  - B-trees are widely used for file systems and databases.

Red-black trees are widely used as system dictionaries. E.g., Java has tree implementations in TreeMap and TreeSet. There are many examples of other balanced BSTs, such as AVL and splay. 2-3 search trees are a subset of b-trees (See recommended textbook for more)
B-trees are widely used for file systems and databases.

Readings:

- Recommended Textbook: Chapter 3.3 (Pages 424-447)
- Website:
  - https://algs4.cs.princeton.edu/33balanced/
- Visualization:
  - https://algs4.cs.princeton.edu/GrowingTree/ (for LLRB trees)

Worksheet:

- Lecture 20 worksheet

Problem 1

- Draw the left-leaning red-black BST that results when you insert items with the keys E, A, S, Y, Q, U, T, I, O, N in that order into an initially empty tree.

ANSWER 1

- Draw the left-leaning red-black BST that results when you insert items with the keys E, A, S, Y, Q, U, T, I, O, N in that order into an initially empty tree.