Balanced Search Trees

Outline

2-3 Search Trees

2-3 Search Trees

A 2-3 search tree is a tree that is either empty (null link) or

2-3 Search Trees

A 2-3 search tree is a tree that is either empty (null link) or

- A 2-node with one key (and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys

2-3 Search Trees

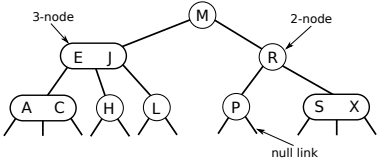
A 2-3 search tree is a tree that is either empty (null link) or

- A 2-node with one key (and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
- A 3-node with two keys (and associated values) and three links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys, and a right link to a 2-3 search tree with larger keys

2-3 Search Trees

A 2-3 search tree is a tree that is either empty (null link) or

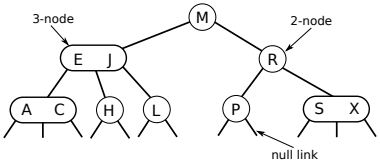
- A 2-node with one key (and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
- A 3-node with two keys (and associated values) and three links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys, and a right link to a 2-3 search tree with larger keys



2-3 Search Trees

A 2-3 search tree is a tree that is either empty (null link) or

- A 2-node with one key (and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
- A 3-node with two keys (and associated values) and three links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys, and a right link to a 2-3 search tree with larger keys

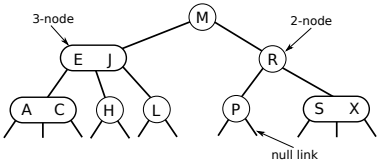


A 2-3 search tree has symmetric order — inorder traversal yields keys in ascending order

2-3 Search Trees

A 2-3 search tree is a tree that is either empty (null link) or

- A 2-node with one key (and associated value) and two links, a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
- A 3-node with two keys (and associated values) and three links, a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node's keys, and a right link to a 2-3 search tree with larger keys



A 2-3 search tree has symmetric order — inorder traversal yields keys in ascending order

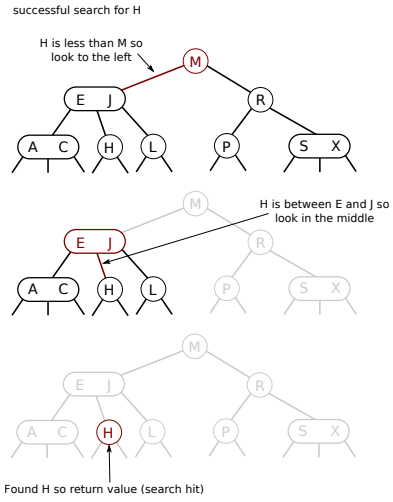
A perfectly balanced 2-3 search tree is one whose null links are all the same distance from the root

2-3 Search Trees

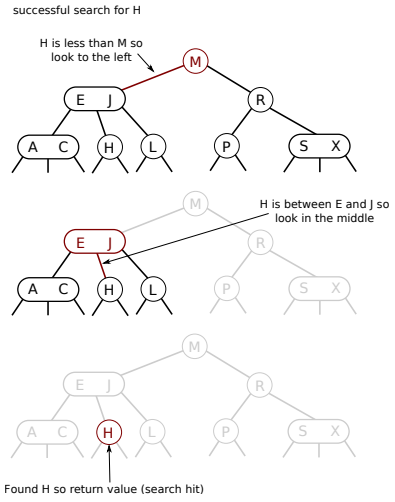
2-3 Search Trees

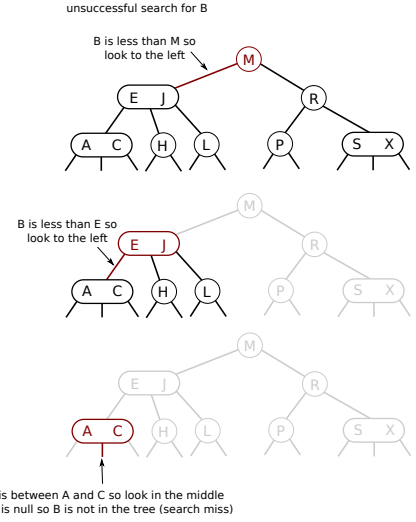
Searching for a key in a 2-3 tree

Searching for a key in a 2-3 tree



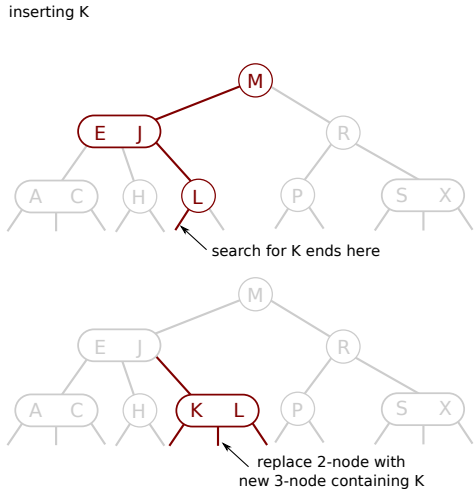
Searching for a key in a 2-3 tree





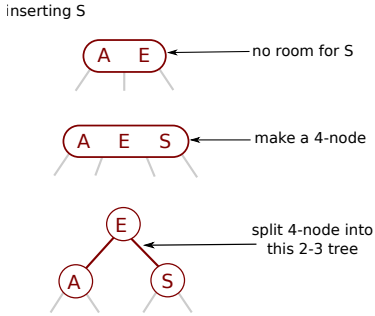
2-3 Search Trees

Inserting a key into a 2-node



2-3 Search Trees

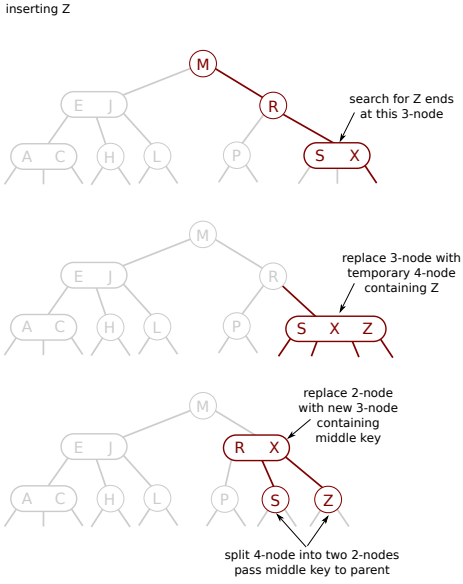
Inserting a key into a single 3-node



2-3 Search Trees

2-3 Search Trees

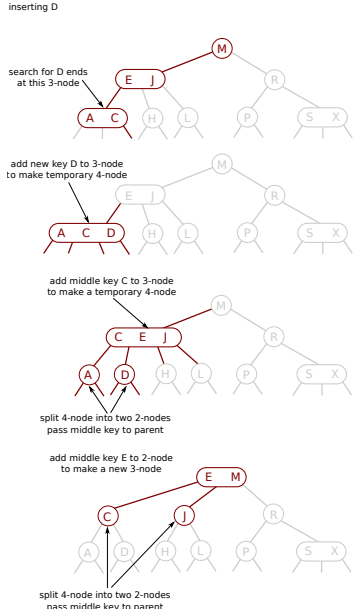
Inserting a key into a 3-node whose parent is a 2-node



2-3 Search Trees

2-3 Search Trees

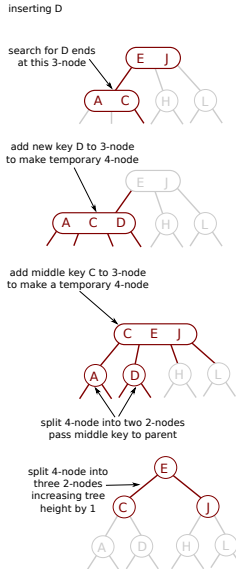
Inserting a key into a 3-node whose parent is a 3-node



2-3 Search Trees

2-3 Search Trees

Splitting the root



2-3 Search Trees

Splitting a 4-node is a local transformation, and thus involves constant number of operations

Splitting a 4-node is a local transformation, and thus involves constant number of operations

Insert operation maintains symmetric order and perfect balance

Splitting a 4-node is a local transformation, and thus involves constant number of operations

Insert operation maintains symmetric order and perfect balance

Tree height

Splitting a 4-node is a local transformation, and thus involves constant number of operations

Insert operation maintains symmetric order and perfect balance

Tree height

- Worst case: $\lg n$ (all 2-nodes)

Splitting a 4-node is a local transformation, and thus involves constant number of operations

Insert operation maintains symmetric order and perfect balance

Tree height

- Worst case: $\lg n$ (all 2-nodes)
- Best case: $\log_3 n \approx 0.631 \lg n$ (all 3-nodes)

Splitting a 4-node is a local transformation, and thus involves constant number of operations

Insert operation maintains symmetric order and perfect balance

Tree height

- Worst case: $\lg n$ (all 2-nodes)
- Best case: $\log_3 n \approx 0.631 \lg n$ (all 3-nodes)
- Between 12 and 20 for a million nodes

Splitting a 4-node is a local transformation, and thus involves constant number of operations

Insert operation maintains symmetric order and perfect balance

Tree height

- Worst case: $\lg n$ (all 2-nodes)
- Best case: $\log_3 n \approx 0.631 \lg n$ (all 3-nodes)
- Between 12 and 20 for a million nodes
- Between 18 and 30 for a billion nodes

Splitting a 4-node is a local transformation, and thus involves constant number of operations

Insert operation maintains symmetric order and perfect balance

Tree height

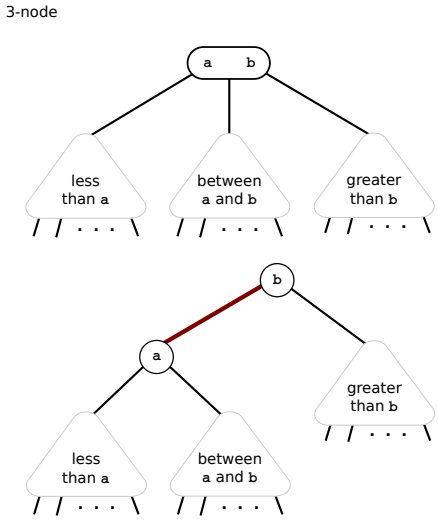
- Worst case: $\lg n$ (all 2-nodes)
- Best case: $\log_3 n \approx 0.631 \lg n$ (all 3-nodes)
- Between 12 and 20 for a million nodes
- Between 18 and 30 for a billion nodes

Guaranteed logarithmic performance for search and insert

Red-Black BSTs

Red-Black BSTs

We represent a 2-3 tree as a BST, using "internal" left-leaning links as "glue" for 3-nodes



Red-Black BSTs

Red-Black BSTs

A red-black tree is a BST such that

Red-Black BSTs

A red-black tree is a BST such that

- No node has two red links connected to it

Red-Black BSTs

A red-black tree is a BST such that

- No node has two red links connected to it
- Every path from root to null link has the same number of black links (perfect black balance)

Red-Black BSTs

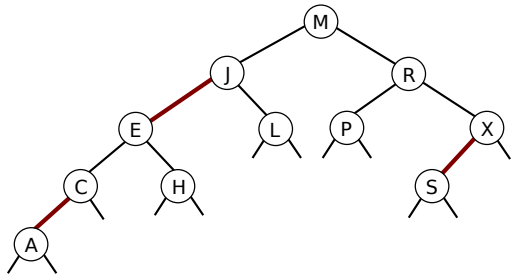
A red-black tree is a BST such that

- No node has two red links connected to it
- Every path from root to null link has the same number of black links (perfect black balance)
- Red links lean left

Red-Black BSTs

A red-black tree is a BST such that

- No node has two red links connected to it
- Every path from root to null link has the same number of black links (perfect black balance)
- Red links lean left

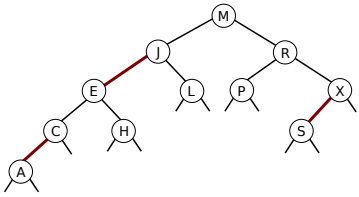


Red-Black BSTs

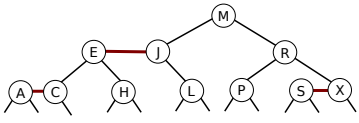
Red-Black BSTs

One-to-one correspondence between red-black BSTs and 2-3 trees

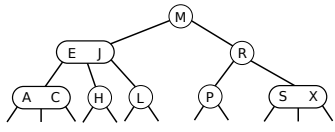
red-black BST



horizontal red links



2-3 tree



Red-Black BSTs

Red-Black BSTs

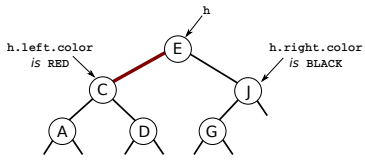
Red-black BST representation: each node is pointed to by precisely one link (from its parent) $\Longrightarrow$ can encode color of links in nodes

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

private class Node {
    private Key key;
    private Value val;
    private int size;
    private boolean color;
    private Node left, right;

    public Node(Key key, Value value) {
        this.key = key;
        this.val = value;
        this.color = RED;
        this.size = 1;
    }
}

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



Elementary Red-black BST Operations

Elementary Red-black BST Operations

Elementary red-black BST operations (left/right rotation and color flip) maintain symmetric order and perfect black balance

Elementary Red-black BST Operations

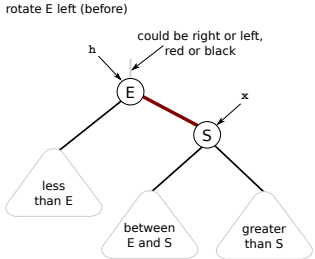
Elementary red-black BST operations (left/right rotation and color flip) maintain symmetric order and perfect black balance

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

Elementary Red-black BST Operations

Elementary red-black BST operations (left/right rotation and color flip) maintain symmetric order and perfect black balance

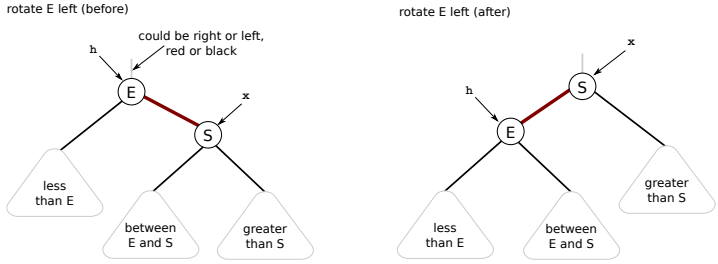
Left rotation: orient a (temporarily) right-leaning red link to lean left



Elementary Red-black BST Operations

Elementary red-black BST operations (left/right rotation and color flip) maintain symmetric order and perfect black balance

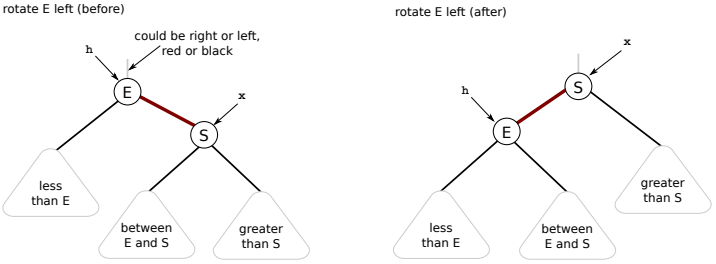
Left rotation: orient a (temporarily) right-leaning red link to lean left



Elementary Red-black BST Operations

Elementary red-black BST operations (left/right rotation and color flip) maintain symmetric order and perfect black balance

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



Implementation of left rotation

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

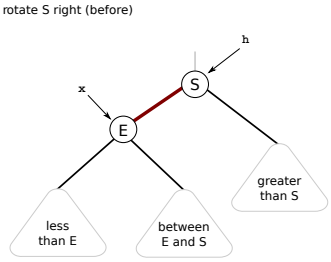
Elementary Red-black BST Operations

Elementary Red-black BST Operations

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

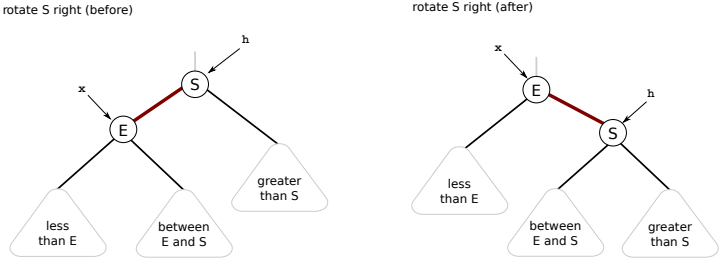
Elementary Red-black BST Operations

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



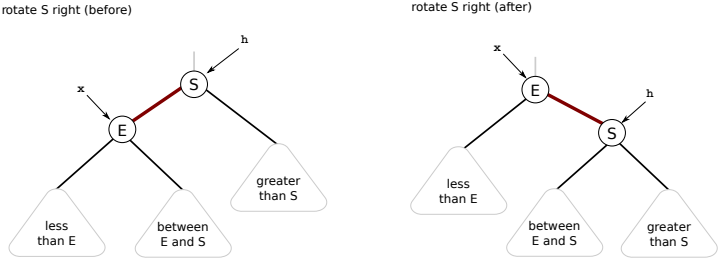
Elementary Red-black BST Operations

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



Elementary Red-black BST Operations

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



Implementation of right rotation

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

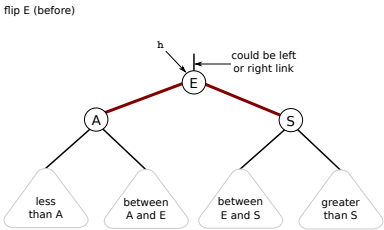
Elementary Red-black BST Operations

Elementary Red-black BST Operations

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

Elementary Red-black BST Operations

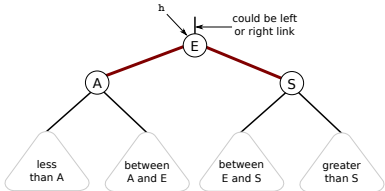
Color flip: recolor to split a (temporary) 4-node



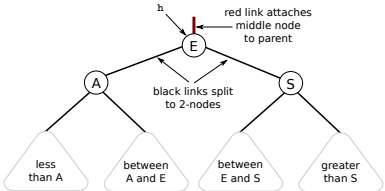
Elementary Red-black BST Operations

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

flip E (before)

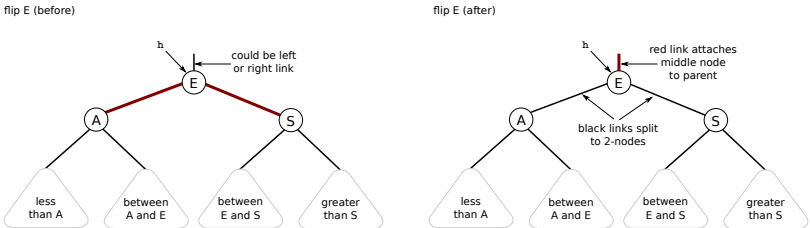


flip E (after)



Elementary Red-black BST Operations

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



Implementation of color flip

```
private void flipColors(Node h) {
    h.color = !h.color;
    h.left.color = !h.left.color;
    h.right.color = !h.right.color;
}
```

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Most operations are the same as for BST-based implementation — ignore color

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Most operations are the same as for BST-based implementation — ignore color

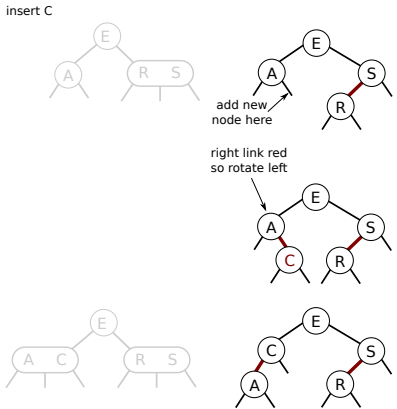
Insertion: the basic strategy is to maintain 1-1 correspondence with 2-3 trees, using the elementary red-black BST operations (left/right rotation and color flip) to maintain symmetric order and perfect balance, but not necessarily color invariants

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Most operations are the same as for BST-based implementation — ignore color

Insertion: the basic strategy is to maintain 1-1 correspondence with 2-3 trees, using the elementary red-black BST operations (left/right rotation and color flip) to maintain symmetric order and perfect balance, but not necessarily color invariants

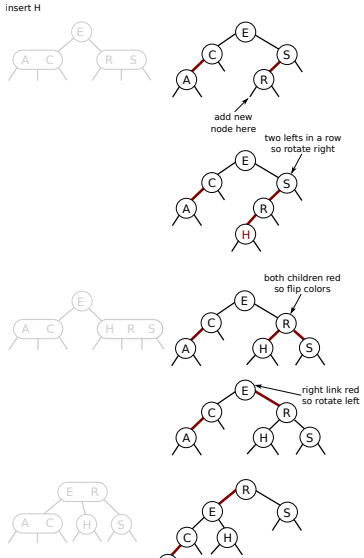
Case 1 (insert into a 2-node at the bottom): do standard BST insert; color new link red; if new red link is a right link, rotate left



Implementation of the Ordered Symbol Table API Using a Red-Black BST

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Case 2 (insert into a 3-node at the bottom): do standard BST insert; color new link red; rotate to balance the 4-node (if needed); flip colors to pass red link up one level; rotate to make lean left (if needed); repeat case 1 or case 2 up the tree (if needed)



Implementation of the Ordered Symbol Table API Using a Red-Black BST

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Implementation (same code for all cases)

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Implementation (same code for all cases)

- Right child red, left child black:
  rotate left

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Implementation (same code for all cases)

- Right child red, left child black:
  rotate left
- Left child, left-left grandchild red:
  rotate right

Implementation of the Ordered Symbol Table API Using a Red-Black BST

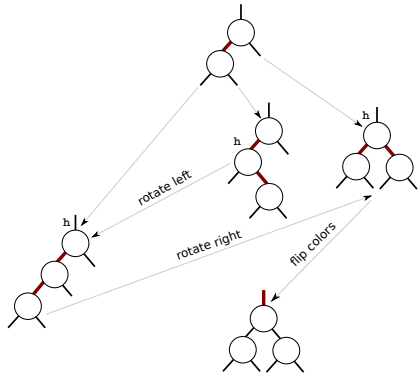
Implementation (same code for all cases)

- Right child red, left child black: rotate left
- Left child, left-left grandchild red: rotate right
- Both children red: flip colors

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Implementation (same code for all cases)

- Right child red, left child black: rotate left
- Left child, left-left grandchild red: rotate right
- Both children red: flip colors



Implementation of the Ordered Symbol Table API Using a Red-Black BST

Implementation of the Ordered Symbol Table API Using a Red-Black BST

RedBlackBinarySearchTreeST.java

```
package dsa;

import java.util.NoSuchElementException;
import stdlib.StdIn;
import stdlib.StdOut;

public class RedBlackBinarySearchTreeST<Key extends Comparable<Key>, Value>
        implements OrderedST<Key, Value> {
    private Node root;

    public void put(Key key, Value value) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        if (value == null) {
            throw new IllegalArgumentException("value is null");
        }
        root = put(root, key, value);
        root.color = BLACK;
    }

    private Node put(Node x, Key key, Value value) {
        if (x == null) {
            return new Node(key, value);
        }
        int cmp = key.compareTo(x.key);
        if (cmp < 0) {
            x.left = put(x.left, key, value);
        } else if (cmp > 0) {
            x.right = put(x.right, key, value);
        } else {
            x.val = value;
        }
        return balance(x);
    }
```

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Implementation of the Ordered Symbol Table API Using a Red-Black BST

RedBlackBinarySearchTreeST.java

```
    private Node balance(Node h) {
        if (!isRed(h.left) && isRed(h.right)) {
            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;
    }
}
```

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Deletion: see exercises 3.3.39 – 3.3.41

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Deletion: see exercises 3.3.39 – 3.3.41

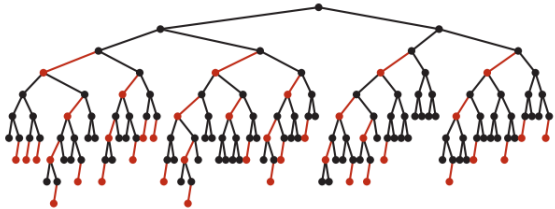
The average length of a path from the root to a node in a red-black BST with n nodes is $\sim \lg n$

Implementation of the Ordered Symbol Table API Using a Red-Black BST

Deletion: see exercises 3.3.39 – 3.3.41

The average length of a path from the root to a node in a red-black BST with n nodes is $\sim \lg n$

Typical red-black BST built from random keys (null links omitted)

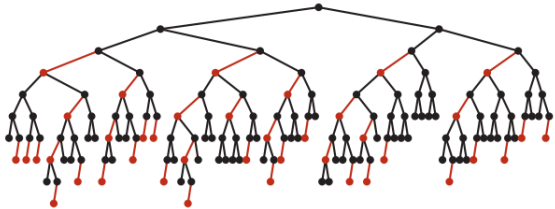


Implementation of the Ordered Symbol Table API Using a Red-Black BST

Deletion: see exercises 3.3.39 – 3.3.41

The average length of a path from the root to a node in a red-black BST with n nodes is $\sim \lg n$

Typical red-black BST built from random keys (null links omitted)



Red-black BST built from ascending keys (null links omitted)

Performance Characteristics

Performance Characteristics

Symbol table operations summary

operation	BST	red-black BST
search	$h^{\dagger}$	$\lg n$
insert	h	$\lg n$
delete	$\sqrt{n}^{\dagger\dagger}$	$\lg n$
min/max	h	$\lg n$
floor/ceiling	h	$\lg n$
rank	h	$\lg n$
select	h	$\lg n$
ordered iteration	n	n

† h is the height of BST, proportional to $\lg n$ if keys inserted in random order
†† $\sqrt{n}$ other operations also become $\sqrt{n}$ if deletions are allowed