Balanced Search Trees
Outline

1. 2-3 Search Trees
2. Red-Black BSTs
3. Elementary Red-black BST Operations
4. Implementation of the Ordered Symbol Table API Using a Red-Black BST
5. Performance Characteristics
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

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

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

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

![Diagram of a 2-3 search tree]

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

Searching for a key in a 2-3 tree

H is less than M so look to the left

M
E J R
A C S X H L P

successful search for H

M
E J R
A C S X H L P

H is between E and J so look in the middle

M
E J R
A C S X H L P

Found H so return value (search hit)

M
E J R
A C S X H L P

B is less than M so look to the left

unsuccessful search for B

M
E J R
A C S X H L P

B is less than E so look to the left

M
E J R
A C S X H L P

B is between A and C so look in the middle

M
E J R
A C S X H L P

link is null so B is not in the tree (search miss)
2-3 Search Trees

Searching for a key in a 2-3 tree
**2-3 Search Trees**

**Searching for a key in a 2-3 tree**

- H is less than M so look to the left.
  - Successful search for H.

- H is between E and J so look in the middle.
  - Found H so return value (search hit).
2-3 Search Trees

Searching for a key in a 2-3 tree

**Successful search for H**

H is less than M so look to the left

```
  M
 / \ 
E   J
/     \
A C   H
|     |
 R    L
```

**Found H so return value (search hit)**

**Unsuccessful search for B**

B is less than M so look to the left

```
  M
 / \
E   J
/     \
A C   H
|     |
 R    L
```

B is less than E so look to the left

```
  M
 / \
E   J
/     \
A C   H
|     |
 R    L
```

B is between A and C so look in the middle

```
  M
 / \
E   J
/     \
A C   H
|     |
 R    L
```

Link is null so B is not in the tree (search miss)
2-3 Search Trees

Inserting a key into a 2-node

search for $K$ ends here

inserting $K$

replace 2-node with

new 3-node containing $K$
Inserting a key into a 2-node

inserting K

search for K ends here

replace 2-node with new 3-node containing K
2-3 Search Trees

Inserting a key into a single 3-node

A E

inserting S

A E S

E

A S

no room for S

make a 4-node

split 4-node into

this 2-3 tree
Inserting a key into a single 3-node

inserting S

no room for S

make a 4-node

split 4-node into this 2-3 tree
2-3 Search Trees
Inserting a key into a 3-node whose parent is a 2-node

inserting Z

search for Z ends at this 3-node

replace 3-node with temporary 4-node containing Z

replace 2-node with new 3-node containing middle key

split 4-node into two 2-nodes
pass middle key to parent
2-3 Search Trees

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

- Search for D ends at this 3-node.
- Inserting D, add new key D to 3-node to make temporary 4-node.
  - Add middle key C to 3-node to make a temporary 4-node.
  - Split 4-node into two 2-nodes, pass middle key to parent.
  - Add middle key E to 2-node to make a new 3-node.
  - Split 4-node into two 2-nodes, pass middle key to parent.
  - Split 4-node into two 2-nodes, pass middle key to parent.

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

inserting D

search for D ends at this 3-node

add new key D to 3-node to make temporary 4-node

add middle key C to 3-node to make a temporary 4-node

split 4-node into two 2-nodes pass middle key to parent

add middle key E to 2-node to make a new 3-node

split 4-node into two 2-nodes pass middle key to parent
2-3 Search Trees

Splitting the root

search for D ends at this 3-node

inserting D

add new key D to 3-node to make temporary 4-node

add middle key C to 3-node to make a temporary 4-node

split 4-node into two 2-nodes

pass middle key to parent

split 4-node into three 2-nodes

increasing tree height by 1

C J E
2-3 Search Trees

Splitting the root

- Inserting D
  - Search for D ends at this 3-node
  - Add new key D to 3-node to make temporary 4-node
  - Add middle key C to 3-node to make a temporary 4-node
  - Split 4-node into two 2-nodes, passing middle key to parent
  - Split 4-node into three 2-nodes, increasing tree height by 1
2-3 Search Trees

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: $\log_2 n$ (all 2-nodes)
- Best case: $\log_3 n \approx 0.631 \log n$ (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.
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)

Guaranteed logarithmic performance for search and insert.
2-3 Search Trees

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

Guaranteed logarithmic performance for search and insert.
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

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

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
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
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
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

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

Horizontal red links
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 BST representation: each node is pointed to by precisely one link (from its parent) ⇒ 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;
}
Red-black BST representation: each node is pointed to by precisely one link (from its parent) \(\implies\) can encode color of links in nodes

```java
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

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

Implementation of left rotation:

```java
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 (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.

\[
\begin{align*}
\text{less than E between E and S} \\
\text{greater than S}
\end{align*}
\]

\[
\text{x could be right or left, red or black}
\]

\[
\text{rotate E left (before)}
\]

\[
\begin{align*}
\text{S} \\
\text{E} \\
\text{less than E between E and S} \\
\text{greater than S} \\
\text{h} \\
\text{x}
\end{align*}
\]

\[
\text{rotate E left (after)}
\]

Implementation of left rotation

```java
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 (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.

```
rotate E left (before)

h

E

could be right or left, red or black

S

x

less than E

between E and S

greater than S
```

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 (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

```java
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

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

Element less than E between E and S greater than S

\[ x \]

rotate S right (before)

Element less than E between E and S greater than S

\[ x \]

rotate S right (after)

Implementation of right rotation

```java
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;
}
```
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

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

rotate S right (before)

Implementation of right rotation

```java
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

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

rotate S right (before)

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

Implementation of right rotation

```java
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

Color flip: recolor to split a (temporary) 4-node between $E$ and $S$ greater than $S$

flip $E$ (before)

A less than $A$ between $A$ and $E$

could be left or right link

E
S
between
E and S
greater
than S
h
flip E (after)
A
less
than A 
between
A and E 
red link attaches
middle node
to parent
black links split
to 2-nodes

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 ;
}
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

Implementation of color flip

```java
private void flipColors ( Node h) {
    h. color = !h. color ;
    h. left . color = !h. left . color ;
    h. right . color = !h. right . color ;
}
```
Elementary Red-black BST Operations

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

```
private void flipColors ( Node h) {
    h. color = !h. color ;
    h. left . color = !h. left . color ;
    h. right . color = !h. right . color ;
}
```
Elementary Red-black BST Operations

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

Implementation of color flip

```java
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

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

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

![Diagram of insertion process]

- Insert C
- insertion into 2-node
- right link red
- rotate left
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).

insert H

E

C S

RA

add new node here

E

A C R S

E

C S

RA
two lefts in a row so rotate right

H

E

A C H SR

E

C R

HA S

both children red so flip colors

E

C R

HA S

right link red so rotate left

A C

E R

H S

R

E S

HC

A
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

- 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)
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 (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

```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);
    }
```
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

```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;
}
```
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

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 \( \sim \log_2 n \).

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

Red-black BST built from ascending 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
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

<table>
<thead>
<tr>
<th>Symbol Table Operations</th>
<th>BST</th>
<th>Red-Black BST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>search</strong></td>
<td>$h \lg n$</td>
<td></td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>$h \lg n$</td>
<td></td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>$\sqrt n$</td>
<td>$h \lg n$</td>
</tr>
<tr>
<td><strong>min/max</strong></td>
<td>$h \lg n$</td>
<td></td>
</tr>
<tr>
<td><strong>floor/ceiling</strong></td>
<td>$h \lg n$</td>
<td></td>
</tr>
<tr>
<td><strong>rank</strong></td>
<td>$h \lg n$</td>
<td></td>
</tr>
<tr>
<td><strong>select</strong></td>
<td>$h \lg n$</td>
<td></td>
</tr>
<tr>
<td><strong>ordered iteration</strong></td>
<td>$n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

† $h$ is the height of BST, proportional to $\lg n$

†† If keys are inserted in random order, other operations also become $\sqrt n$. If deletions are allowed, all operations become $\sqrt n$. 
## Performance Characteristics

### Symbol table operations summary

<table>
<thead>
<tr>
<th>operation</th>
<th>BST</th>
<th>red-black BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>$h^\dagger$</td>
<td>$\lg n$</td>
</tr>
<tr>
<td>insert</td>
<td>$h$</td>
<td>$\lg n$</td>
</tr>
<tr>
<td>delete</td>
<td>$\sqrt{n}^{\dagger\dagger}$</td>
<td>$\lg n$</td>
</tr>
<tr>
<td>min/max</td>
<td>$h$</td>
<td>$\lg n$</td>
</tr>
<tr>
<td>floor/ceiling</td>
<td>$h$</td>
<td>$\lg n$</td>
</tr>
<tr>
<td>rank</td>
<td>$h$</td>
<td>$\lg n$</td>
</tr>
<tr>
<td>select</td>
<td>$h$</td>
<td>$\lg n$</td>
</tr>
<tr>
<td>ordered iteration</td>
<td>$n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

$^\dagger h$ is the height of BST, proportional to $\lg n$ if keys inserted in random order

$^{\dagger\dagger} \sqrt{n}$ other operations also become $\sqrt{n}$ if deletions are allowed