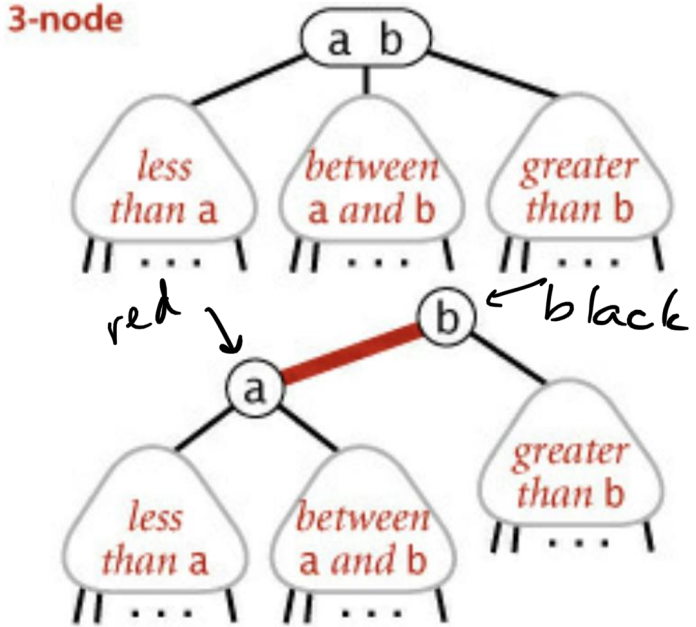
Left-Leaning Red-Black Trees

Left-Leaning Red-Black Trees

- An *ordered* symbol table
- A self-balancing tree
- Combines the simple implementation of a BST with the balancing ideas of a 2-3 tree
- Provides a tree implementation with a 1-to-1 correspondence with the more complex 2-3 tree

(Left-leaning) Red-Black Trees

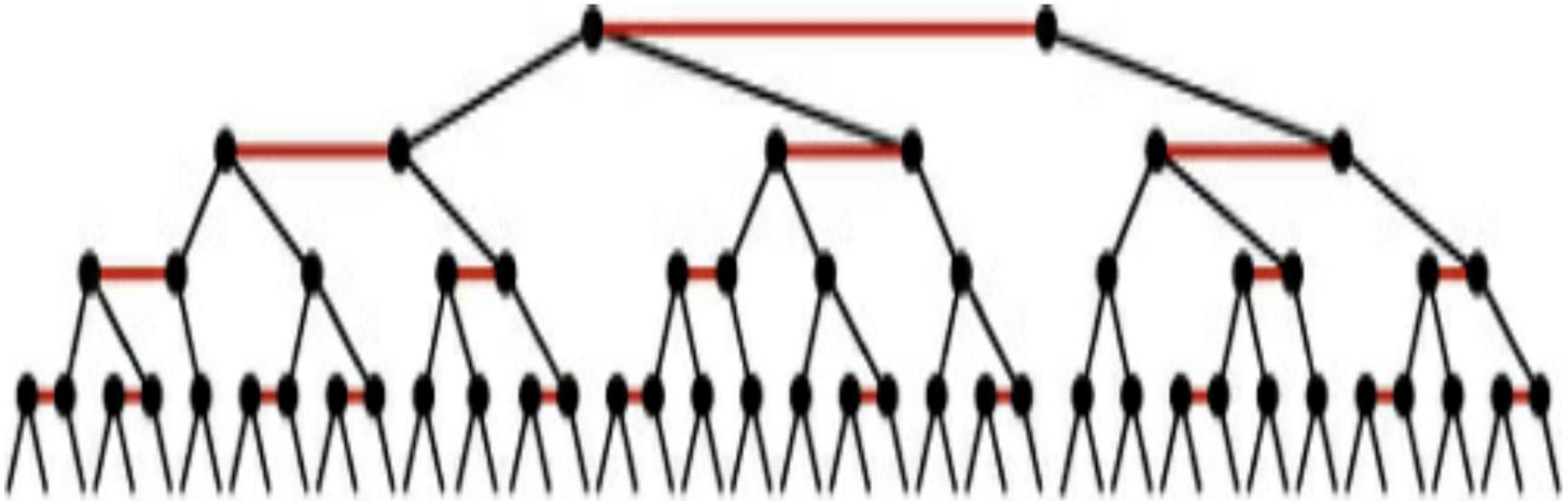
- Representation for 2-3 trees
- Basic Idea: Start with standard BST (2-nodes only) and add extra information to the nodes to represent 3-nodes
- 2 kinds of links:
  - red—bind together two nodes to represent 3-nodes
  - black—bind together the 2-3 tree
- 3-nodes are encoded as two nodes connected by a single left-leaning red link
- NOTE: We don't really encode "links." We just indicate a node's link to it's parent by coloring the node itself.



Encoding a 3-node with two 2-nodes connected by a left-leaning red link

Picture from [1]

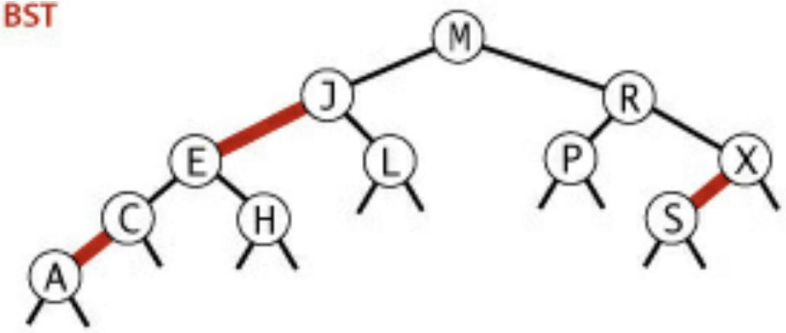
“Flattening” the red links into horizontal links essentially turns the binary R-B tree into a 2-3 tree.



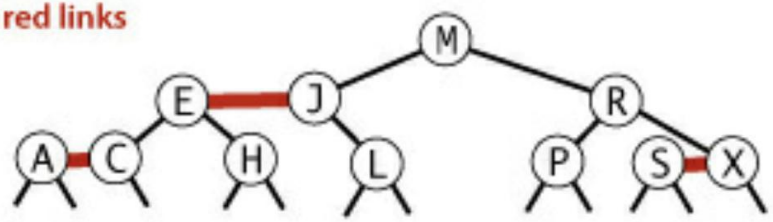
A red-black tree with horizontal red links is a 2-3 tree

Picture from [1]

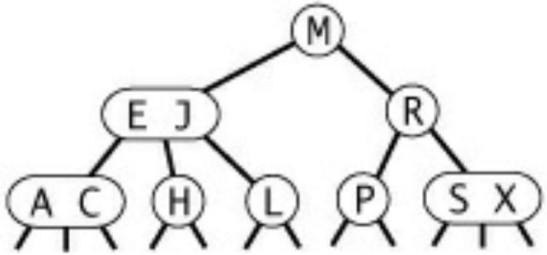
red-black BST



horizontal red links



2-3 tree



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

“Flattening” the red links into horizontal links essentially turns the binary R-B tree into a 2-3 tree.

Picture from [1]

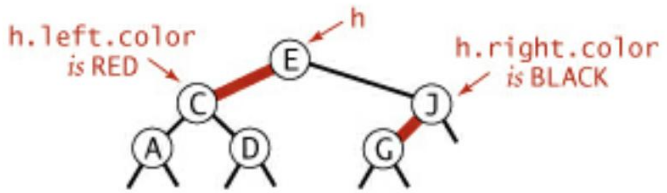
Red-Black Tree: A BST with these properties...

- Links are **red** or **black**
  - Typically, we call the nodes red or black depending on the color of the link that connects it to its parent because there is no “link” object. It’s just a pointer from one node to another.
  - We can encode this by adding a boolean variable to each node (**true** if **red**, **false** if **black**).
- A black node has *at most* one red child. (i.e. red nodes cannot have red siblings)
- Red nodes have black children. (i.e. red nodes cannot have red children)
- The root is always black.
- Tree has perfect *black balance* (all paths from the root to a leaf have the same number of black links—also known as the **black height** of the tree)

In a *left-leaning* red-black tree, *all red links lean to the left*. This is an implementation decision that can reduce the number of “cases” and therefore shorten the code.

Color Representation

Each node has a new boolean variable for color (according to the link connecting it to its parent)—**true** if **red, false** if **black.**




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

private class Node
{
   Key key;           // key
   Value val;         // associated data
   Node left, right; // subtrees
   int N;             // # nodes in this subtree
   boolean color;     // color of link from
                      //   parent to this node

   Node(Key key, Value val, int N, boolean color)
   {
      this.key   = key;
      this.val   = val;
      this.N     = N;
      this.color = color;
   }
}

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

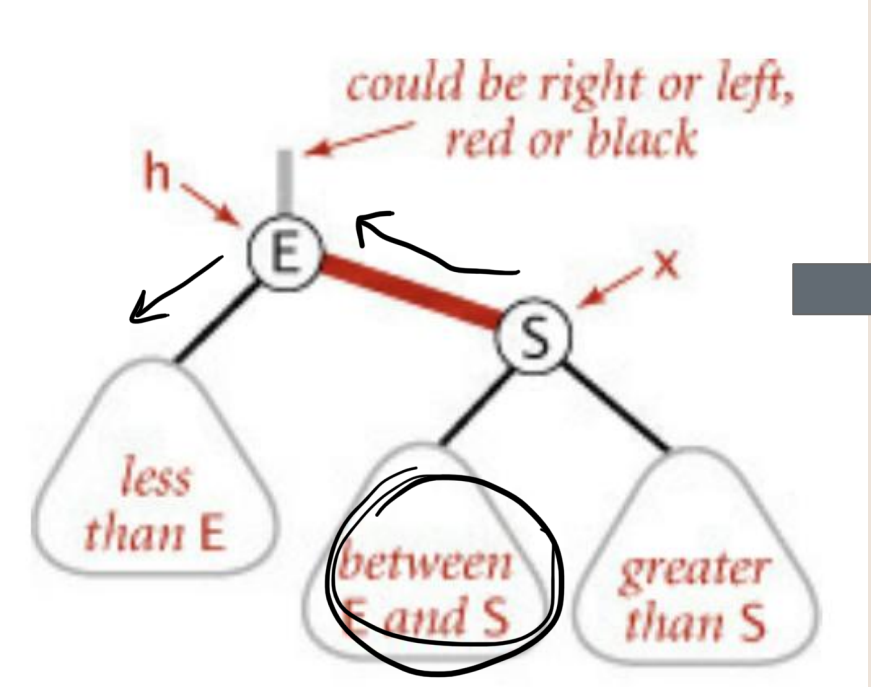
Node representation for red-black BSTs

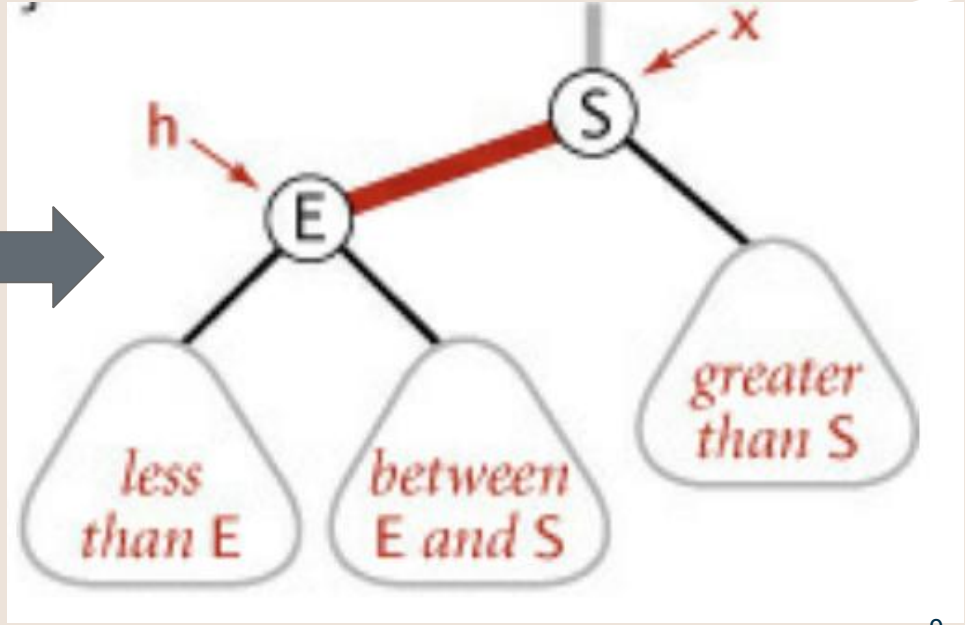
Picture from [1]

Transformations

This is how we maintain the properties that ensure black height balance.

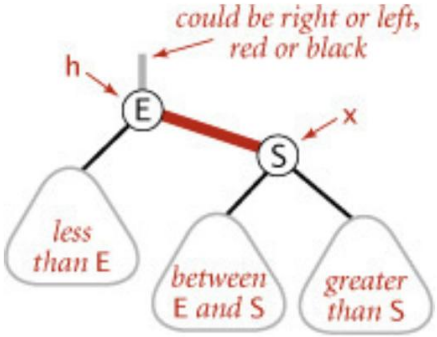
Left Rotation



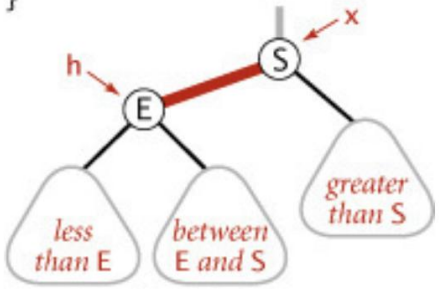


Pictures from [1]

Left Rotation Code



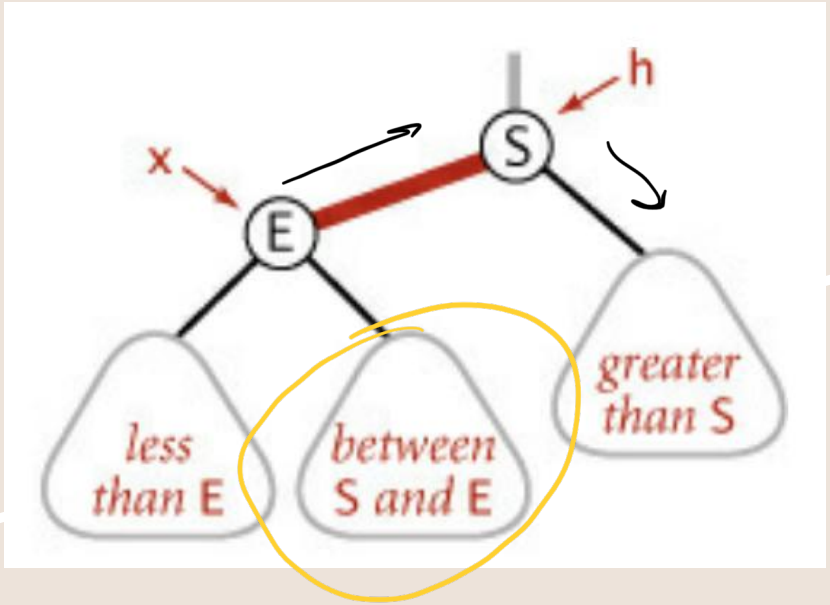

```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

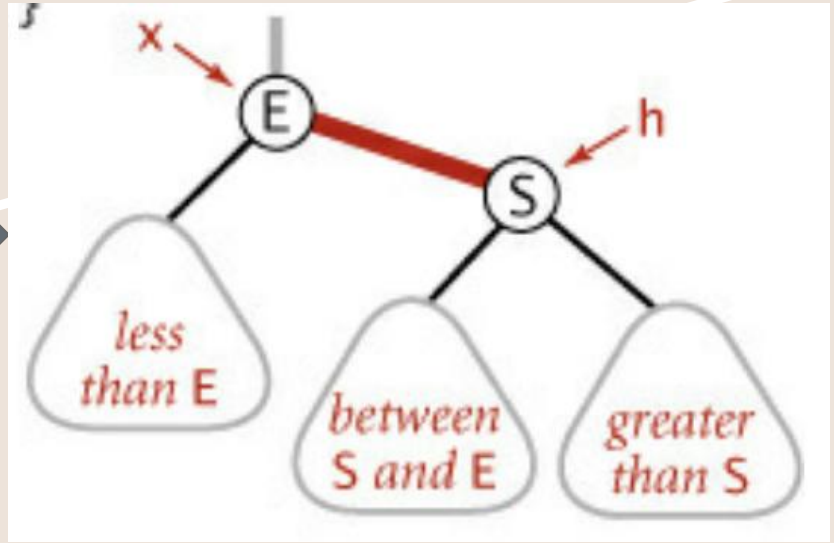


Left rotate (right link of h)

Pictures from [1]

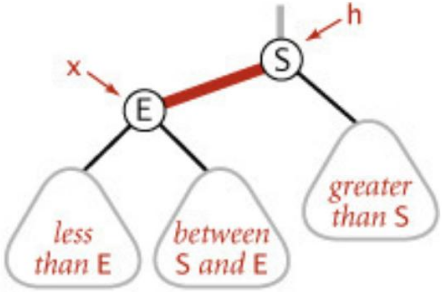
Right Rotation



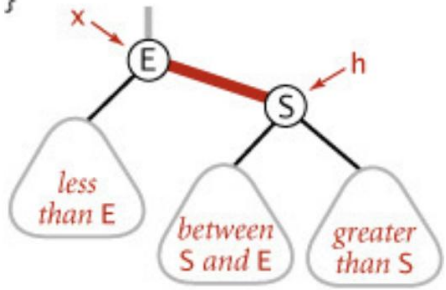


Pictures from [1]

Right Rotation Code



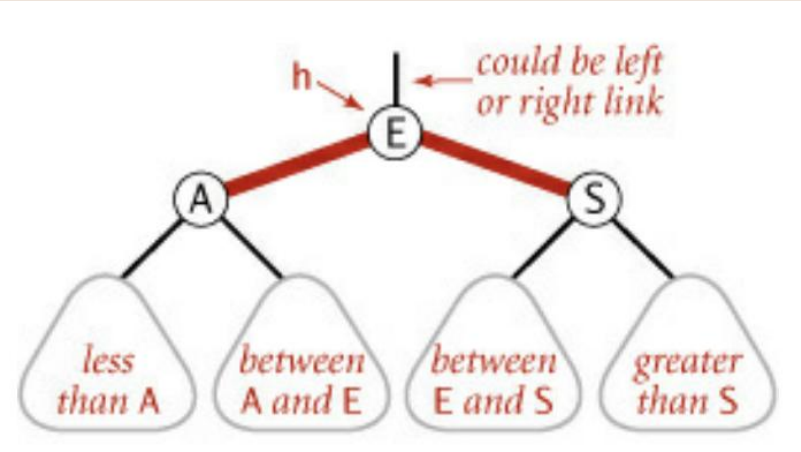

```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

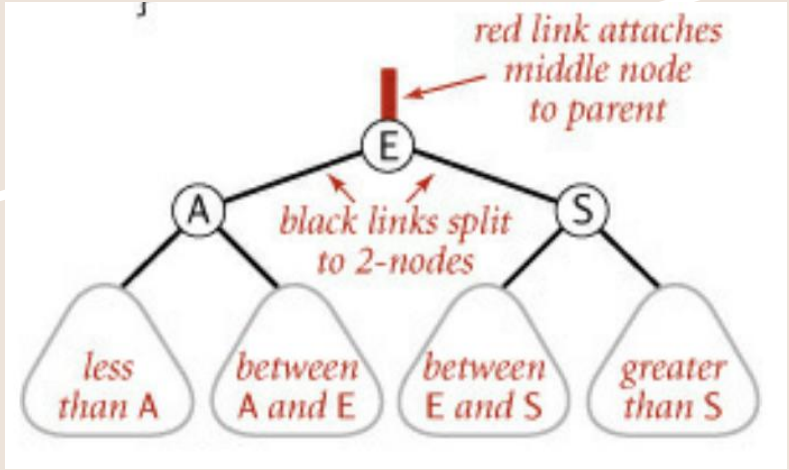


Right rotate (left link of h)

Pictures from [1]

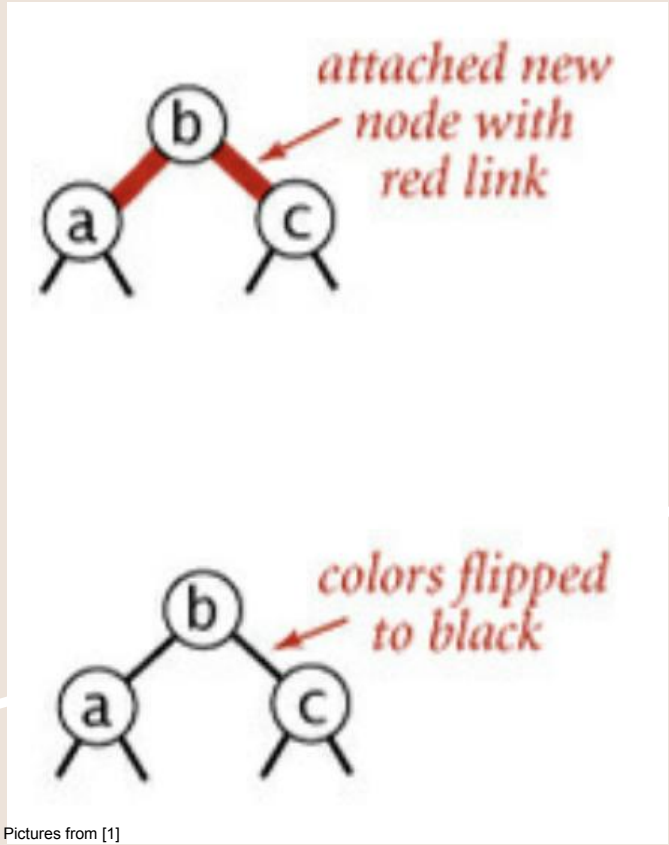
Color Flip (equivalent to splitting a 4-node)





Pictures from [1]

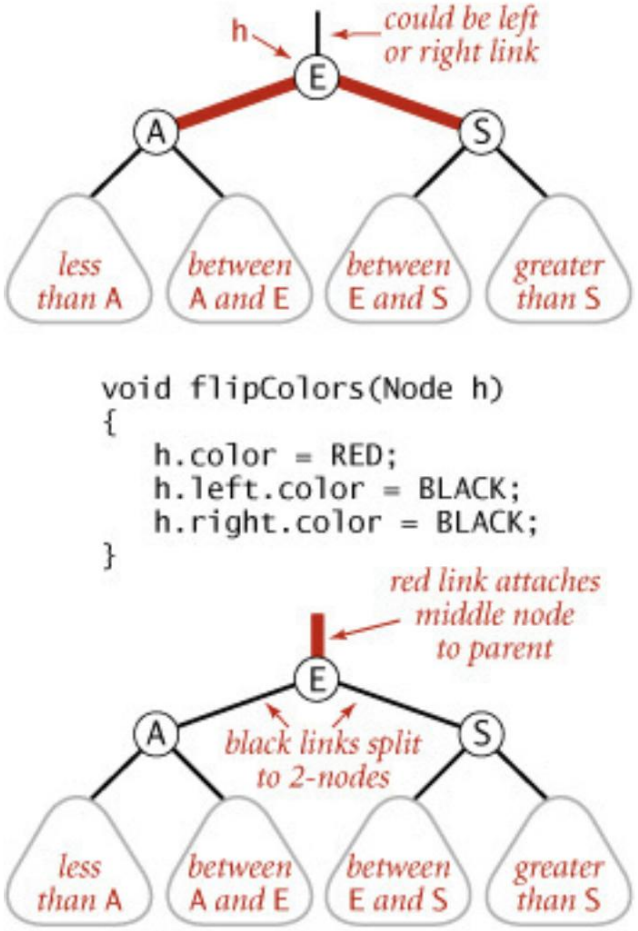
Color Flip at the root



This is when the black height increases (like splitting a 4-node at the root).

Pictures from [1]

Color Flip Code



Flipping colors to split a 4-node

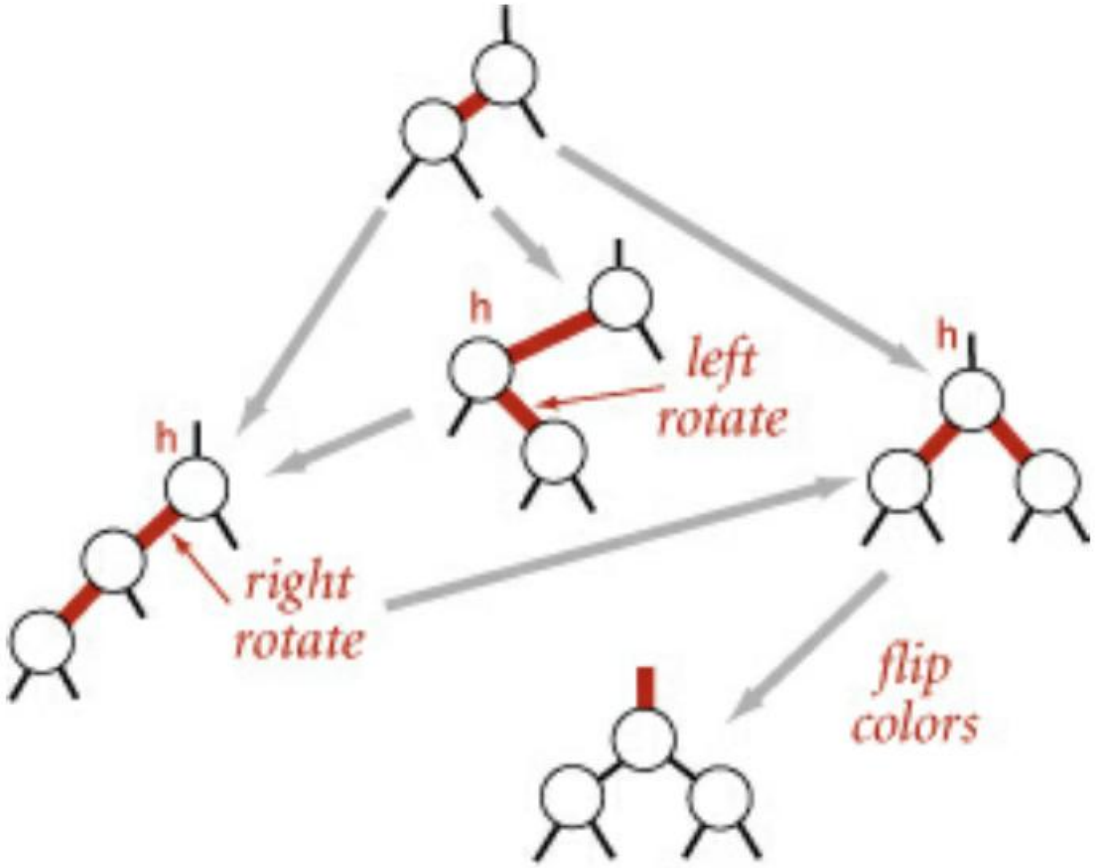
Pictures from [1]

Operations

put

- New nodes are always inserted at the bottom of the tree.
- New nodes are always red.
- Transformations are used to re-balance the black height of the tree by ensuring the properties discussed previously.
- Transformations may pass red links up the tree just like splitting 4-nodes.

Fixing the tree...



Passing a red link up a red-black BST

Picture from [1]

3 main operations to maintain balance:

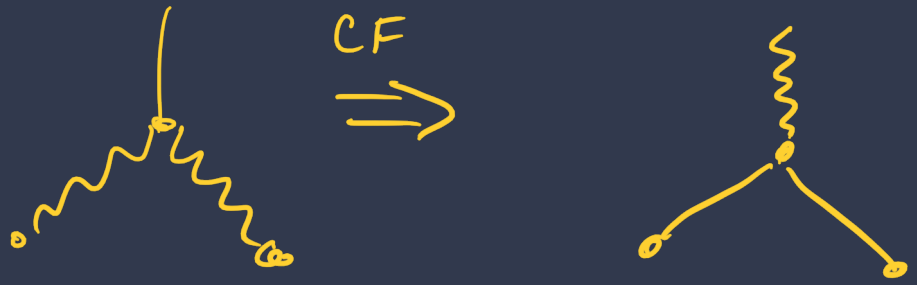
- rotate left
- rotate right
- color flip

Insertion uses those three operations as you “pass the red link up the tree”

Case 1: *X.left* is red, *X.right* is null or black.

Case 2: *X.right* is red, *X.left* is null or black.

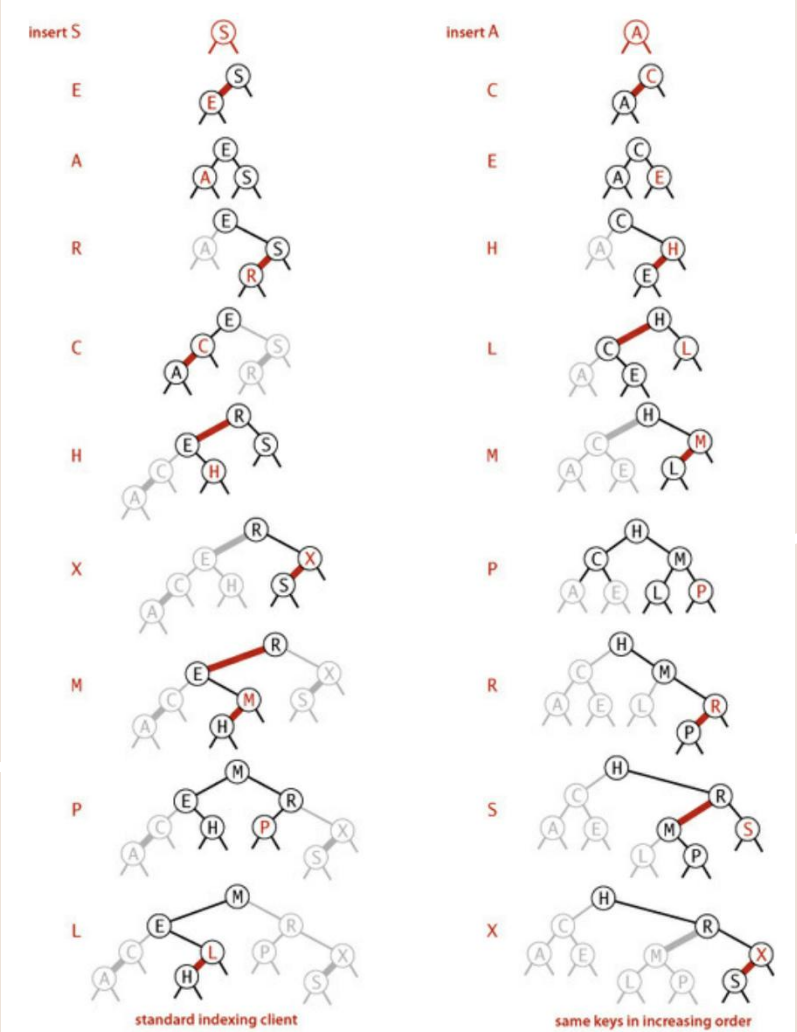
Case 3: *X.left* is red, *X.right* is red. (Note that *X* itself will always be black in this case.)



Case 4: *X.left* is red, *X.left.left* is red.

This case is only possible with *delete*.

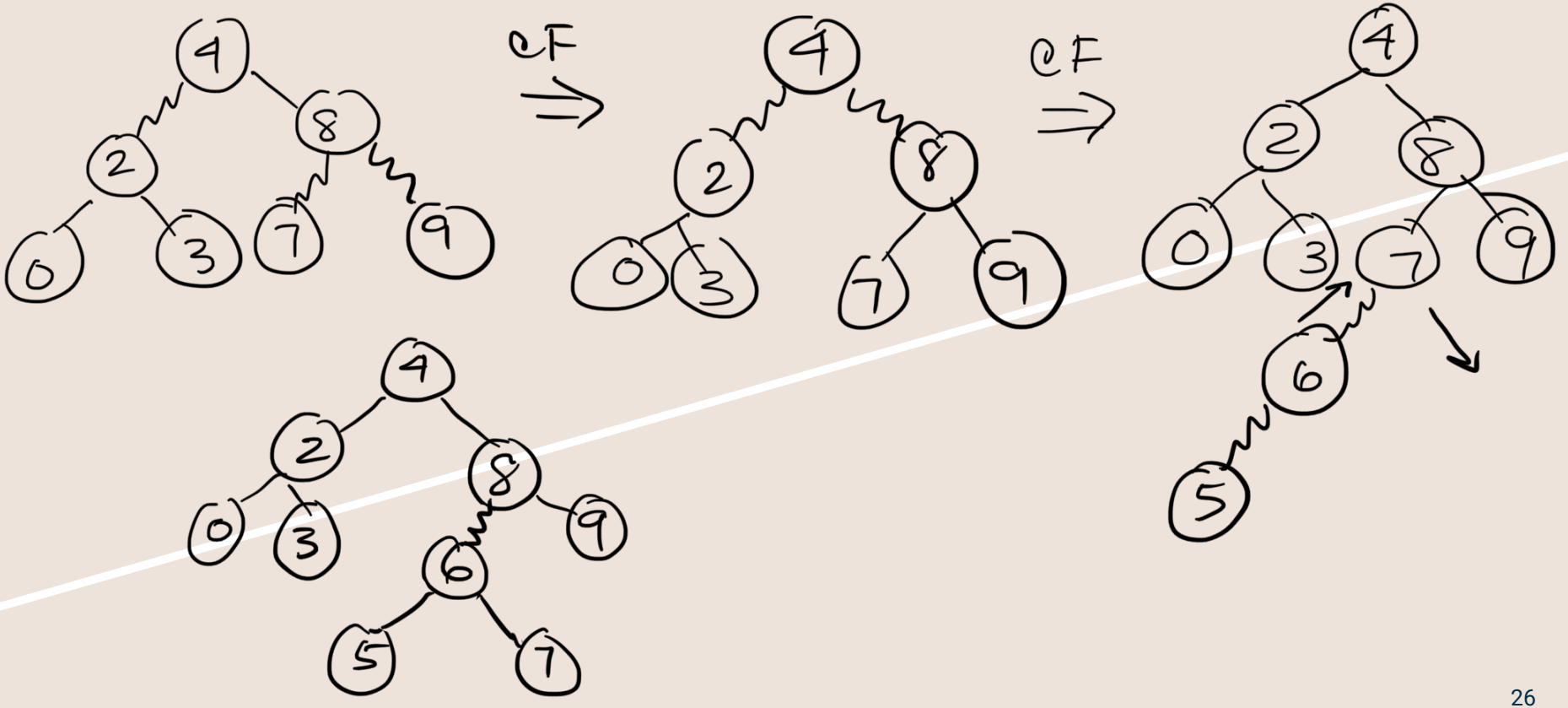
Case 5: *X.right* is red, *X.right.left* is red.



Red-black BST construction traces

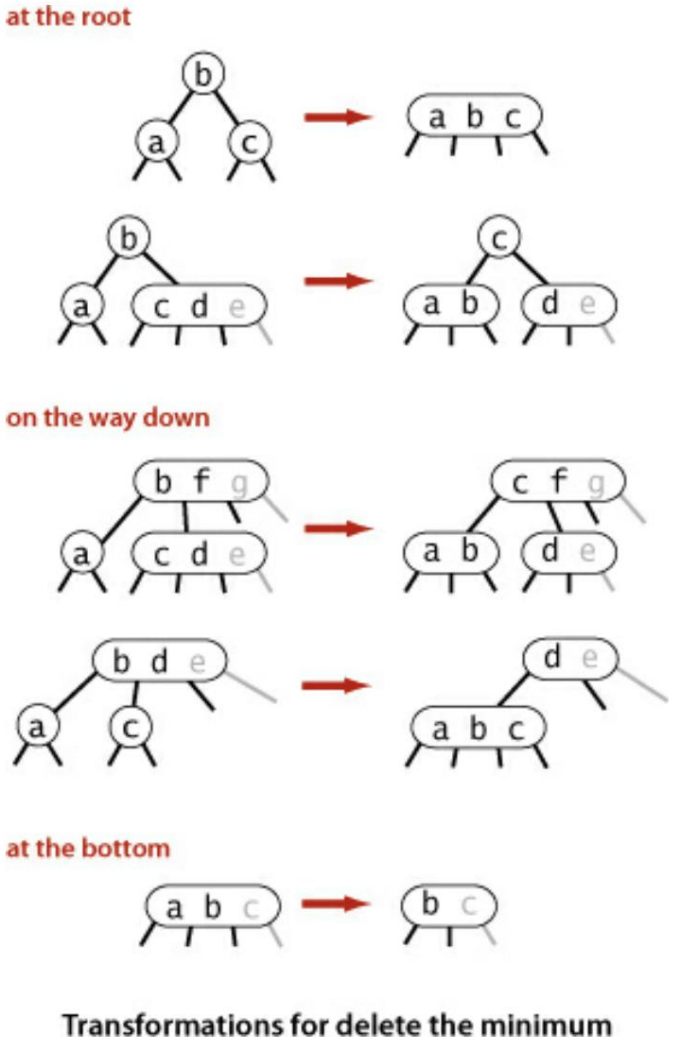
Picture from [1]

Example: *insert 0, insert 4, insert 7, insert 2, insert 3, insert 8, insert 9, insert 6, insert 5*



delete

- It’s easier to delete a red node than a black node. Why?
- Note: The computer won’t know if a node to be deleted is red or black until it finds that node.
- Key Idea: Use the inverses of the transformations discussed previously to “carry a red link down the line”--just in case.
- How is this related to deleting from a 2-3 tree?



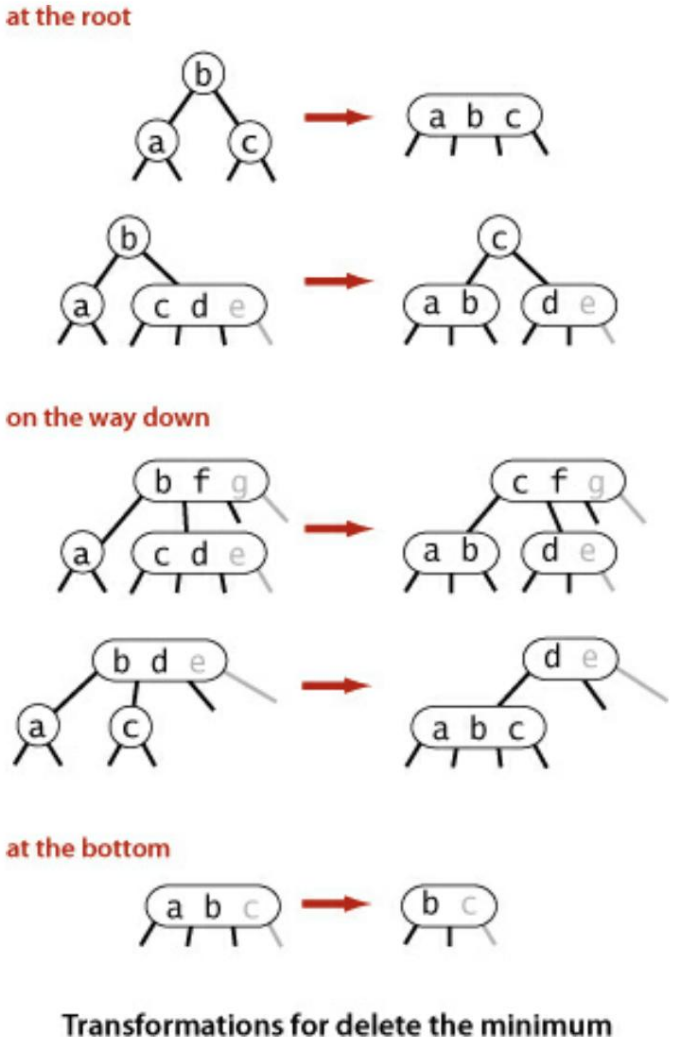
Big Idea for Deletion:

It is easy to delete a key from a 3-node (or a 4-node) at the bottom of the tree. The tricky part is deleting a 2-node. (WHY?)

So we can transform the tree on the way down to ensure that the current node is not a 2-node. (HOW?)

What is the equivalent idea for a red-black tree?

Picture from [1]



Transformations for delete the minimum

Big Idea for Deletion:

It is easy to delete a key from a 3-node (or a 4-node) at the bottom of the tree. The tricky part is deleting a 2-node. (WHY?)

So we can transform the tree on the way down to ensure that the current node is not a 2-node. (HOW?)

What is the equivalent idea for a red-black tree?

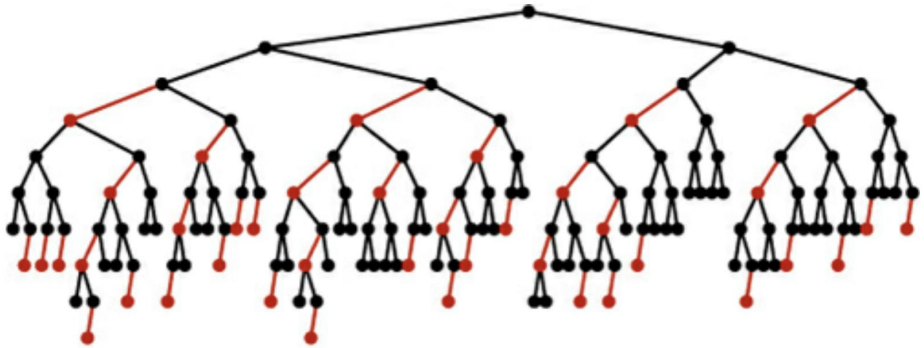
Make sure the current node is always red.

Picture from [1]

Example: delete 1, delete 6, delete 2, delete 7, delete 4, delete 3, delete 5

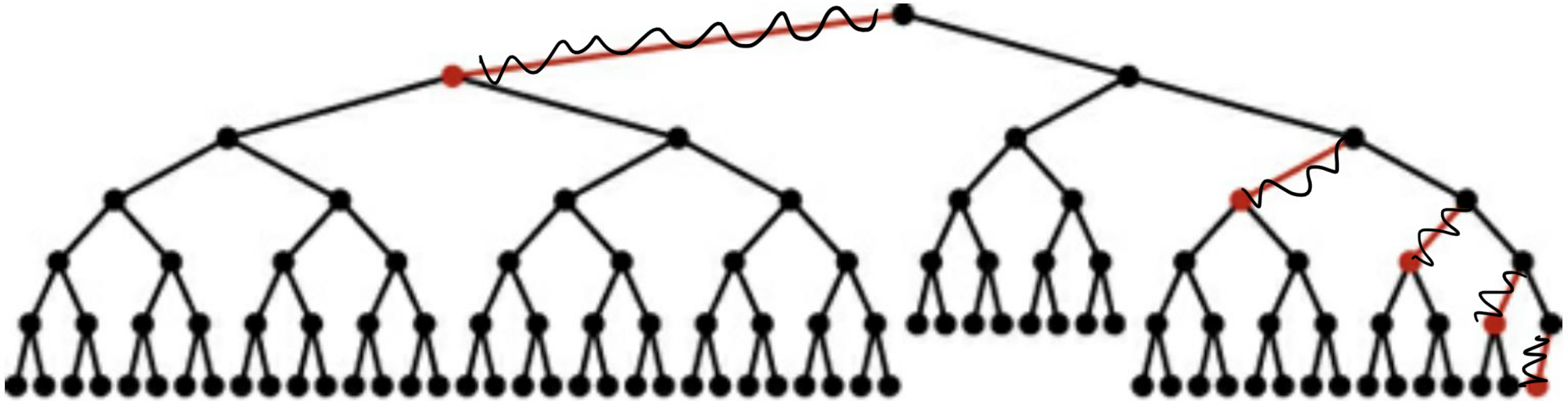
Properties of Red Black Trees

- **Near** perfect balance
- 1-1 correspondence with 2-3 Trees
- Height is no more than 2lgN (worst case is all 2-nodes except the leftmost path, which is 3 nodes and so twice as long)
- Average length of root to a node is ~1.00lgN



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

Picture from [1]



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

Picture from [1]

Proposition I. In a red-black BST, the following operations take logarithmic time in the worst case: search, insertion, finding the minimum, finding the maximum, floor, ceiling, rank, select, delete the minimum, delete the maximum, delete, and range count.

Picture from [1]

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
sequential search (unordered linked list)	N	N	$N/2$	N	no
binary search (ordered array)	$\lg N$	N	$\lg N$	$N/2$	yes
binary tree search (BST)	N	N	$1.39 \lg N$	$1.39 \lg N$	yes
2-3 tree search (red-black BST)	$2 \lg N$	$2 \lg N$	$1.00 \lg N$	$1.00 \lg N$	yes

Cost summary for symbol-table implementations (updated)

Picture from [1]

References

[1] *Algorithms, Fourth Edition;* Robert Sedgewick and Kevin Wayne (and associated slides)
[2] Slides from https://www.cs.princeton.edu/~rs/talks/LLRB/RedBlack.pdf