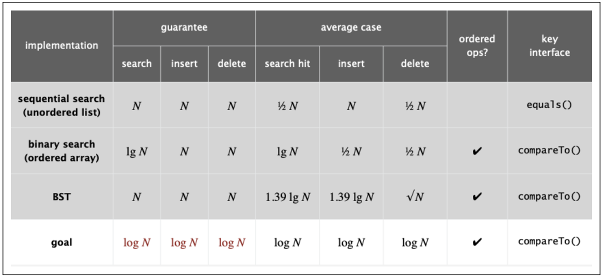
Searching: Balanced Search Trees

Nicholas Moore

Dept. of Computing and Software, McMaster University, Canada

Acknowledgments: Material mainly based on the textbook Algorithms by Robert Sedgewick and Kevin Wayne (Chapters 3.3, 6), Prof. Mhaskar's course slides

Symbol Table Review



implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		`equals()`
binary search (ordered array)	lg N	N	N	lg N	½ N	½ N	✔	`compareTo()`
BST	N	N	N	1.39 lg N	1.39 lg N	$\sqrt{N}$	✔	`compareTo()`
goal	log N	log N	log N	log N	log N	log N	✔	`compareTo()`

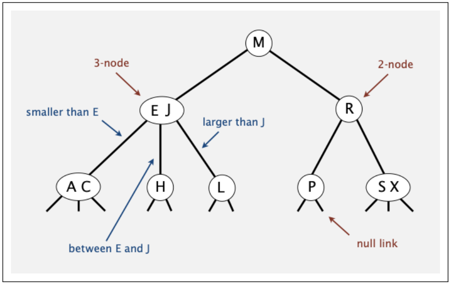
The further a Binary Search Tree departs from being fully filled, the worse our runtime is.

- In the worst case, BSTs have no advantage over unsorted arrays!
- Thus, we commonly employ **tree balancing algorithms** to improve our worst-case runtime.

2-3 Search Trees

Our goal is to modify our tree so that it can't be unbalanced by insert operations. One way of doing this is the **2-3 Tree**.

- 2-nodes have one key and two children.
- 3-nodes have two keys and three children.

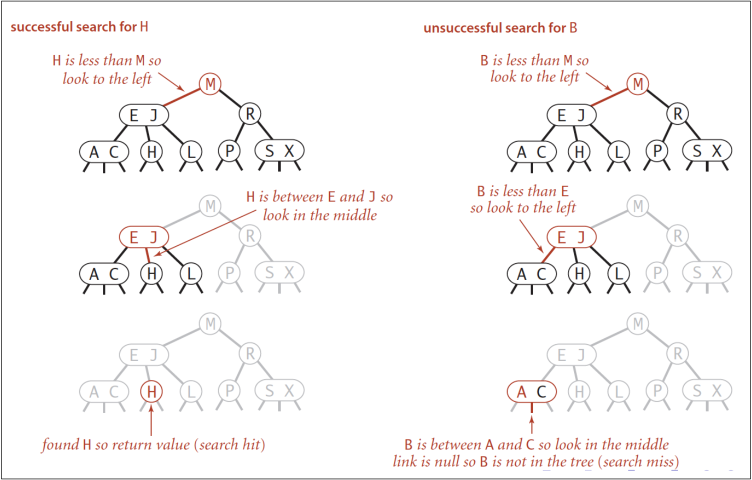


The two values in a 3-node are the upper and lower bounds of a range. All elements in that range fall into the center branch. Otherwise, 3-nodes function similarly to a normal BST.

- A BST is a 2-3 Tree containing only 2-nodes!

2-3 Search Tree: Search

Searching goes exactly as you might expect.


successful search for H
H is less than M so look to the left
H is between E and J so look in the middle
found H so return value (search hit)
unsuccessful search for B
B is less than M so look to the left
B is less than E so look to the left
B is between A and C so look in the middle
link is null so B is not in the tree (search miss)

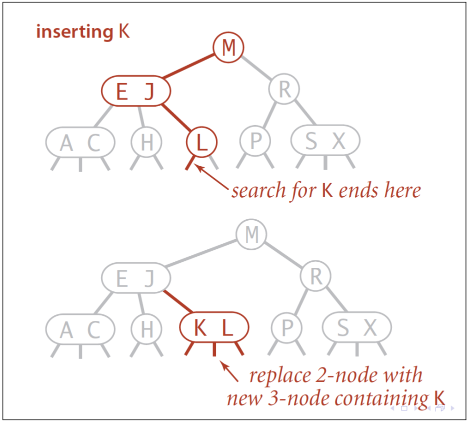
Insertion Philosophy

All paths through a 2-3 Tree are of the same length *by construction!*

- The reason that BSTs create unequal branch lengths is that we insert nodes by simply appending them to branches.
- The insertion procedure for 2-3 Trees avoids this by expanding the tree **from the root**, rather than from individual branches.
- We are still, however, required to add our values at the ends of branches to maintain sortedness.

Contradiction? Not hardly! Since nodes can hold more than one value now, we can insert into nodes as well as creating new ones!

2-3 Search Tree: Insert I

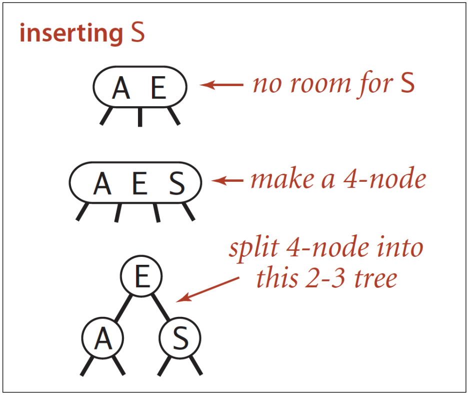


- We still proceed by finding the branch we would insert into under normal BST rules.
- If the leaf node we wish to add to is a 2-node, we add the value to this node (in the correct position) and change it into a 3-node!
- Therefore, we avoid creating a new node and extending the tree's branches, thus maintaining perfect balance.

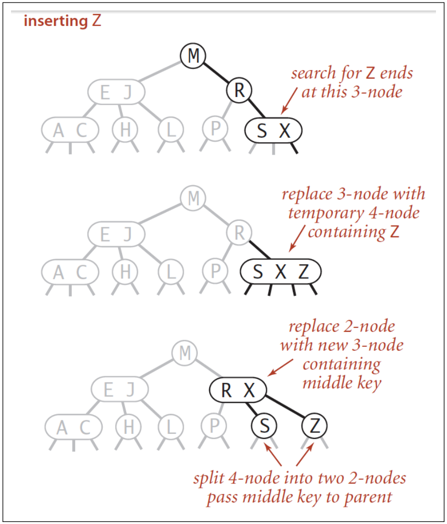
2-3 Search Tree: Insert II

But professor! What if the node you're adding to already has two values!?

- You add it anyways, creating an unstable, **temporary 4-node**
  - 4-nodes have 3 values and 4 children.
- 4-nodes are quickly decomposed into three 2-nodes, with the middle value as parent and the other two as children.
- If we stopped here, we would still see uneven branch growth.

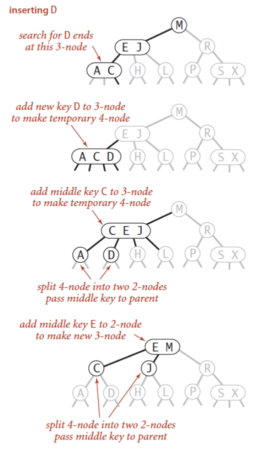


2-3 Search Tree: Insert a single 3-node whose parent is 2-node


inserting Z
M
E J
R
A C
H
L
P
S X
search for Z ends at this 3-node
replace 3-node with temporary 4-node containing Z
S X Z
replace 2-node with new 3-node containing middle key
R X
S
Z
split 4-node into two 2-nodes pass middle key to parent

- The generated parent node is quickly added to it's own parent.
- This may make the parent unstable as well, so this procedure follows the tree upwards until it hits the root node.
- If the root node decomposes into three 2-nodes, there is no parent to merge into. Therefore, the new branches add to the depth of the tree.
- Thus, 2-3 Trees can only expand at the root, maintaining balance!

2-3 Search Tree: Insert a single 3-node whose parent is 3-node

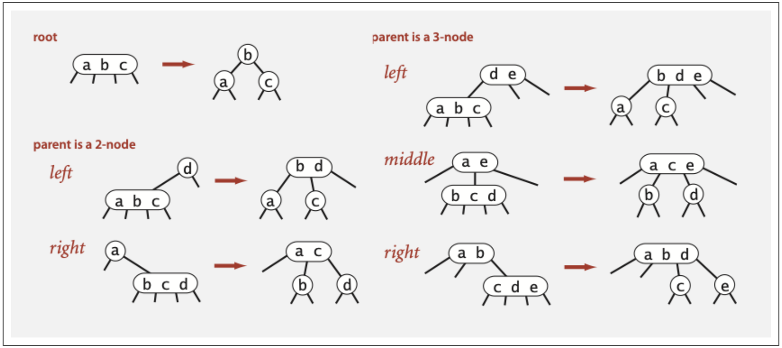

inserting D
search for D ends at this 3-node
M
E J
R
A C
H
L
P
S X
add new key D to 3-node to make temporary 4-node
A C D
add middle key C to 3-node to make temporary 4-node
C E J
A
D
split 4-node into two 2-nodes pass middle key to parent
add middle key E to 2-node to make new 3-node
E M
C
J
split 4-node into two 2-nodes pass middle key to parent

Insert into a 3-node whose parent is a 3-node

NOT ADDING NODES TO BRANCHES
ROOT NODES EXPANDING EVENLY
Perfectly balanced...
2-3 TREES
...As all things should be
imgflip.com

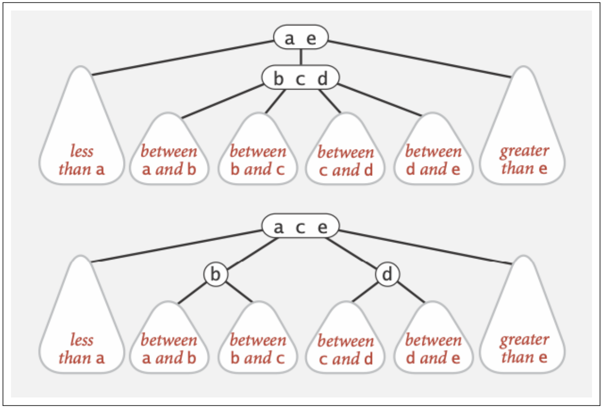
2-3 tree Analysis: Local transformations in a 2-3 Tree

At each level of the tree, splitting a 4-node requires a constant number of operations, as it involves one of six transformations:



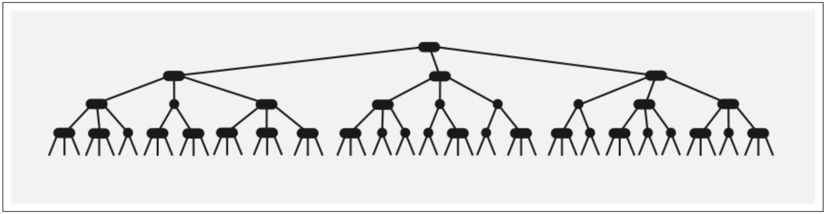
2-3 tree Analysis: Global properties in a 2-3 tree

Each transformation maintains symmetric order and perfect balance.



2-3 tree: performance

Every path from root to null link has same length.



Tree height.

- Worst case: $\log_2 N$ – all two nodes in the tree
- Best case: $\log_3 N \approx .631 \log_2 N$ – all three nodes in the tree

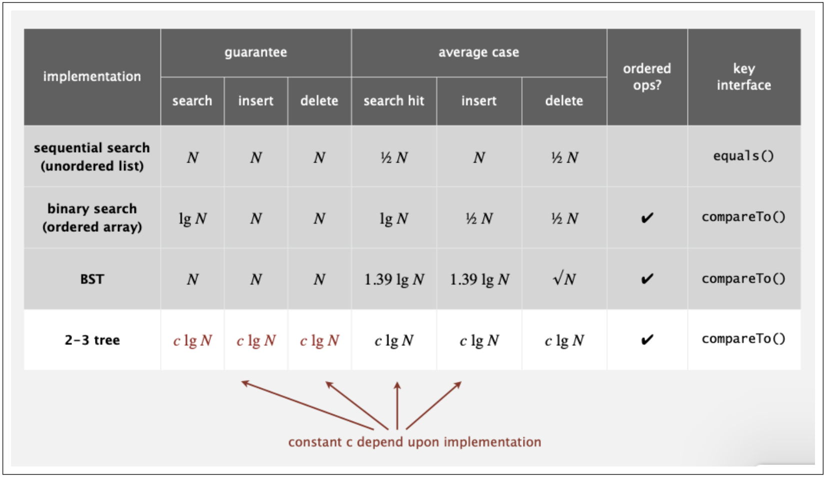
Either way, we are guaranteed logarithmic performance for search and insert.

ST implementation

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

ST implementations: summary



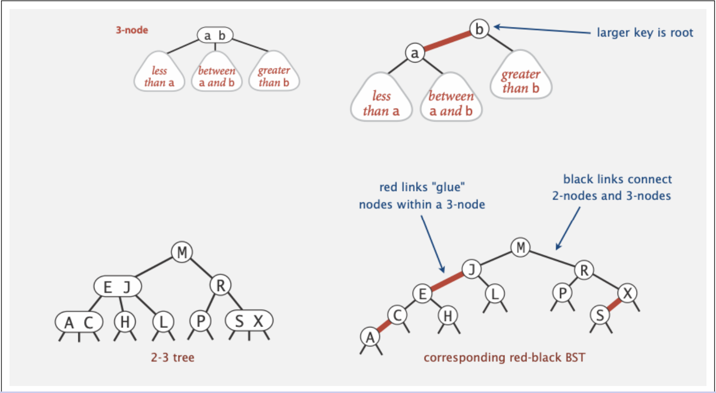
implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		equals()
binary search (ordered array)	lg N	N	N	lg N	½ N	½ N	✔	compareTo()
BST	N	N	N	1.39 lg N	1.39 lg N	$\sqrt{N}$	✔	compareTo()
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✔	compareTo()

constant c depend upon implementation

Red-Black Trees

Left-leaning red-black BSTs

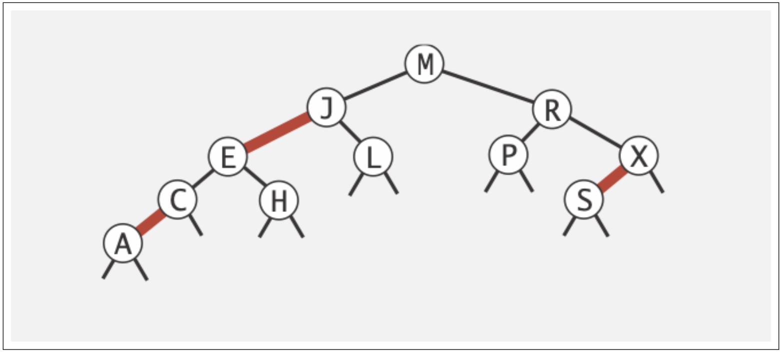
- Red-Black Trees are a way to encode 2-3 trees without needing to keep track of multiple node types and a variable number of values per node.
- Black links are the same in red-black and 2-3 trees.
- Red links connect values that are in the same nodes in 2-3 trees.



An equivalent definition of Red-Black Trees

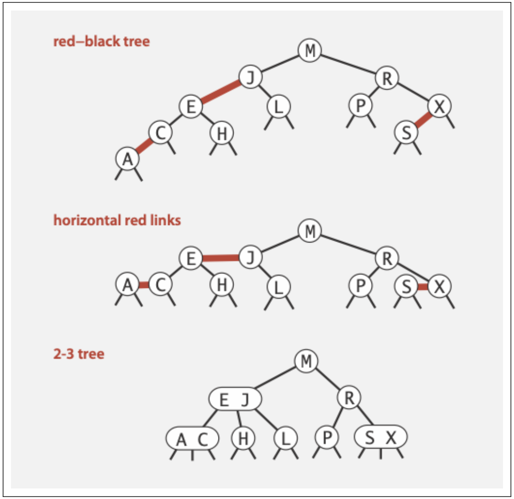
A BST such that:

1. No node has two red links connected to it.
2. Every path from root to null link has the same number of black links. – "Perfect Black Balance"
3. Red links lean left.



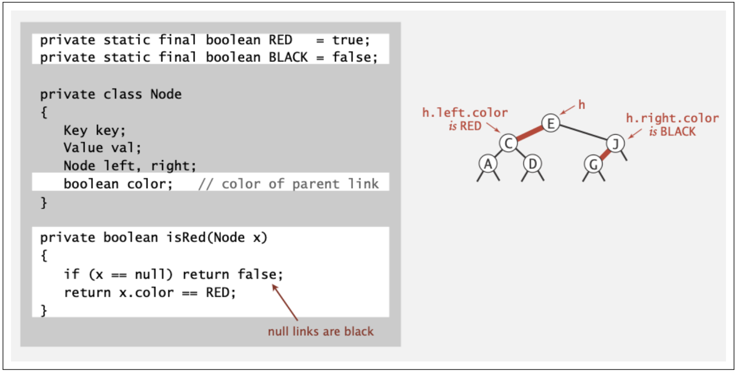
Like Checkers!

Key property. 2-3 and LLRB trees exactly correspond.



Red-black BST representation

- Each node has only one parent, so we can encode color of links in nodes (use Boolean variable *color*).
    - True = Red, False = Black
- The color of a node is the color of its parent link.



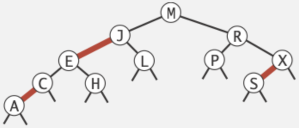
```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node
{
   Key key;
   Value val;
   Node left, right;
   boolean color;    // color of parent link
}

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

Search implementation for red-black BSTs

Observation. Search is the same as for elementary BST (ignore color).

but runs faster
because of better balance

```
public Val get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if       (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```



Remark. Most other ops (e.g., floor, iteration, selection) are also identical.

Insertion in a LLRB tree: overview

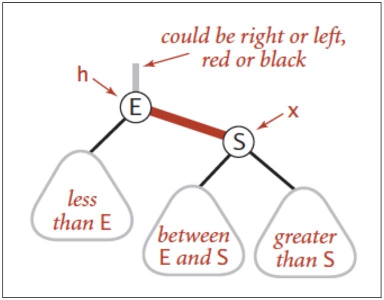
In general, we wish to maintain our one-to-one correspondence with 2-3 trees.

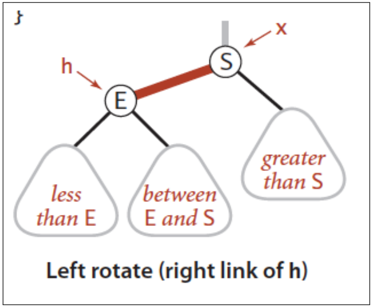
- In order to achieve this we will need to introduce two additional sub-operations
    - Rotation
    - Colour flip

Elementary red-black BST operations: Left Rotation

In the same way that 2-3 trees had unstable 4-nodes during some operations, left leaning red-black trees may have temporary right leaning links.

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



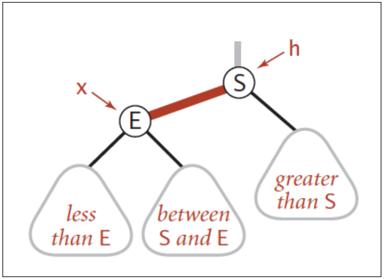


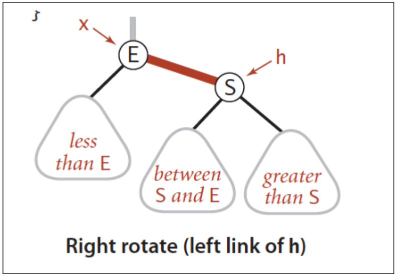
Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations: Right Rotation

We will also require the ability to rotate our left leaning red link right temporarily.

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

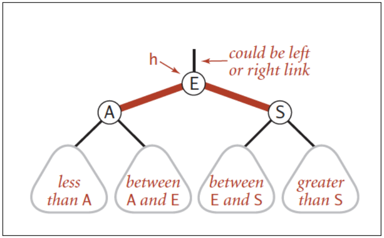


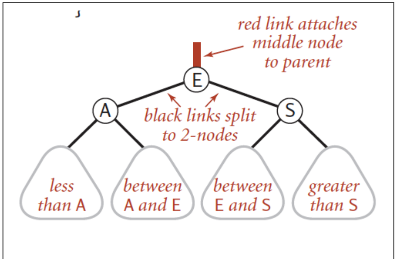


Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations: Color Flip

Color Flip. Recolor to split a (temporary) 4-node.



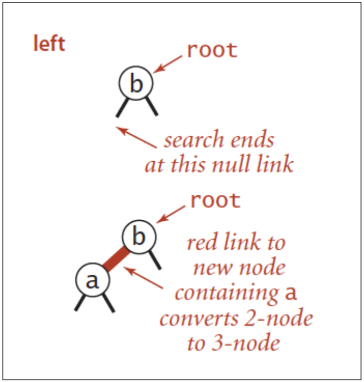

red link attaches middle node to parent
E
A
S
black links split to 2-nodes
less than A
between A and E
between E and S
greater than S

Flipping colors to split a 4-node

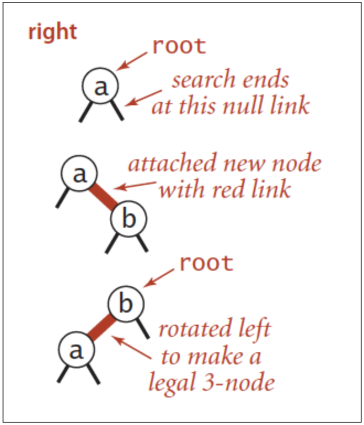
Invariants. Maintains symmetric order and perfect black balance.

Insertion in a LLRB tree: Insert into a single 2-node tree

Warmup 1. Insert into a tree with exactly one 2-node.



Case 1

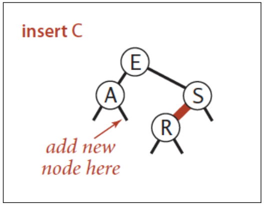


Case 2

Insertion in a LLRB tree: Insert into a 2-node at the bottom - I

Case 1. Insert into a 2-node at the bottom.

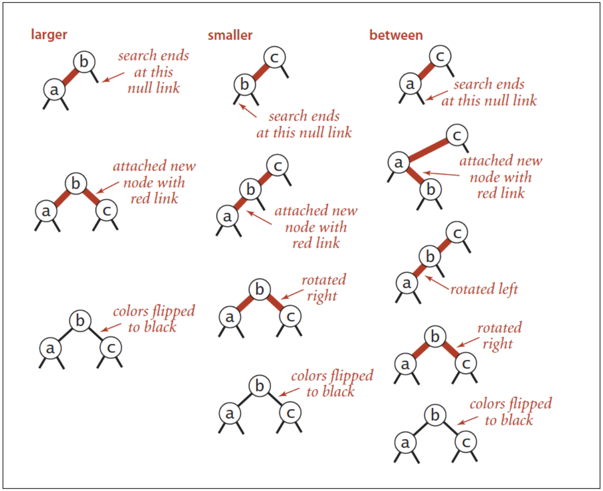
- Insert keys into a red-black BST normally, adding a new node at the bottom with a red link.
- If the parent is a 2-node, then the same two cases just discussed are effective.
- **In particular, if the new node is attached to the left link, the parent simply becomes a 3-node!**
- If the new red link is a right link, we simply left-rotate it.



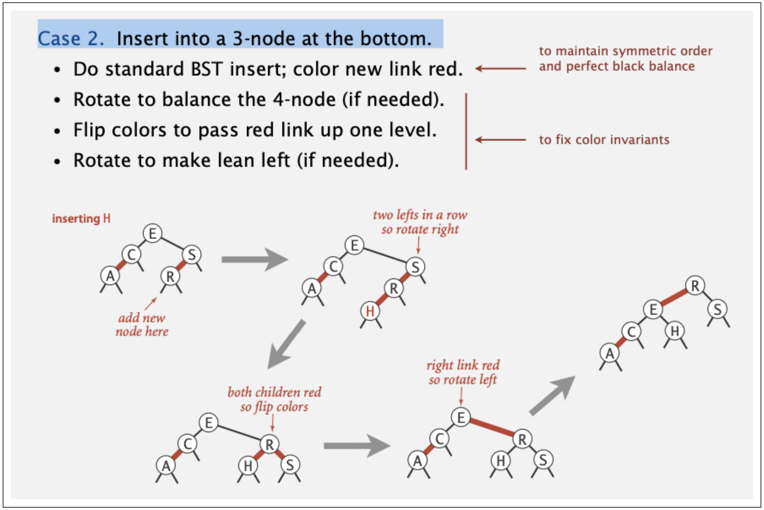
Case 1

Insert a node into a two node in a (single 3-node tree)

Insert a node into a two node tree (equivalent to a single 3-node 2-3 tree)


larger
search ends at this null link
attached new node with red link
colors flipped to black
smaller
search ends at this null link
attached new node with red link
rotated right
colors flipped to black
between
search ends at this null link
attached new node with red link
rotated left
rotated right
colors flipped to black

Insert into a tree: Insert a node at the bottom - I



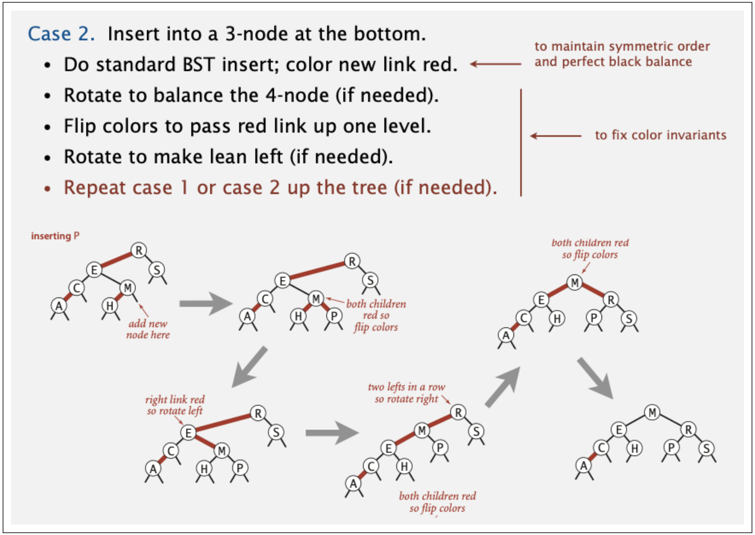
Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).

← to fix color invariants

inserting H
add new node here
two lefts in a row so rotate right
both children red so flip colors
right link red so rotate left
E
C
S
A
R
H

Insert into a tree: Insert a node at the bottom - II



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).

to maintain symmetric order and perfect black balance
to fix color invariants
inserting P
add new node here
both children red so flip colors
right link red so rotate left
two lefts in a row so rotate right
both children red so flip colors
both children red so flip colors

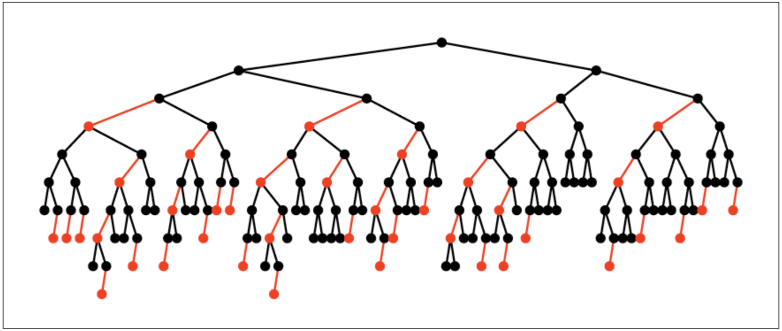
Keeping the root black

- A red root implies that the root is part of a 3-node, but that is not the case, so we color the root of the tree black after each insertion.
- Note that the black height of the tree increases by 1 whenever the color of the root is flipped from black to red.

RBT Analysis

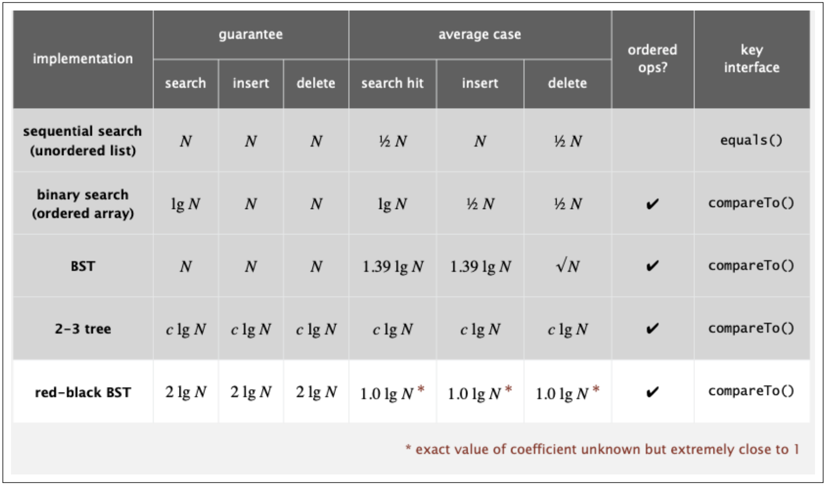
Proposition. Height of tree is $\leq 2\log_2 N$ in the worst case – why?

Below is the typical red-black BST built from random keys (null links omitted)



Property. Height of tree is $\sim 1.0\log_2 N$ in typical applications.

ST implementations: summary



implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	½ N	½ N	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✔	`compareTo()`
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$ *	$1.0 \lg N$ *	$1.0 \lg N$ *	✔	`compareTo()`

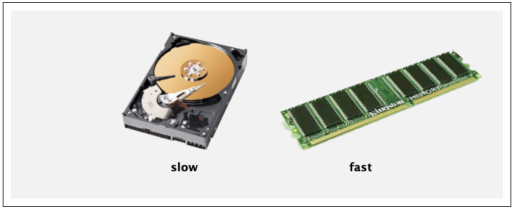
* exact value of coefficient unknown but extremely close to 1

B-Trees

File System Model

A Page is a contiguous block of data (e.g., a file or 1024 or 4,096-byte chunk).

A Probe is the first access of a page (e.g., from disk to memory).



- In general, the time required for a probe is much larger than time to access data within a page.
- When studying algorithms for external searching, we count the number of probes, or page accesses.
- We therefore wish to minimize the number of probes.

Cost model. Goal. Access data using minimum number of probes.

B-Trees

The B-Tree generalizes the 2-3 tree, and is a multi-way balanced search trees.

- B-Trees are used particulary for searching external filesystems.

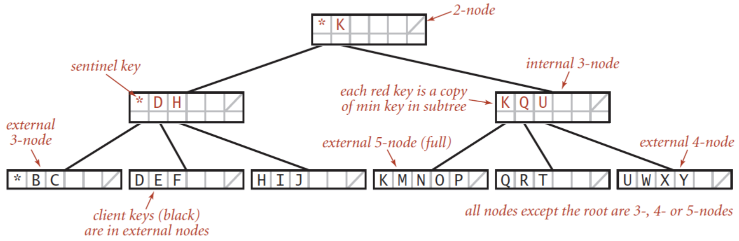
A B-tree of order M

- Allows up to $M - 1$ key-link pairs per node, where a link is the address of a page, rather than a symbol table value.
- We can choose a very large M, though it must be an even number.
  - For example, $M = 1024$
- We maintain at least 2 key-link pairs at root.
- At least $M/2$ key-link pairs in other nodes.
- External nodes contain client keys, and have references to actual data.
- Internal nodes contain copies of keys to guide search.

Example: In a B-tree of order 4, each node has at most 3 and at least 2 key-link pairs.

B-Tree Example - I

A special key (*), known as a sentinel, helps implement the B-Tree. Sentinels are a kind of wild-card entry that is defined as less than all other keys.

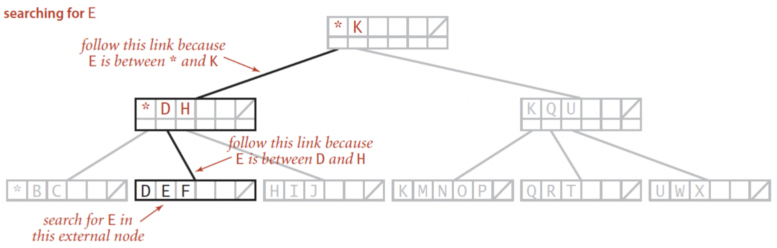


Anatomy of a B-tree set (M = 6)

Search in a B-tree

Search in a B-tree

- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.


searching for E
follow this link because E is between * and K
* K
* D H
K Q U
follow this link because E is between D and H
* B C
D E F
H I J
K M N O P
Q R T
U W X
search for E in this external node

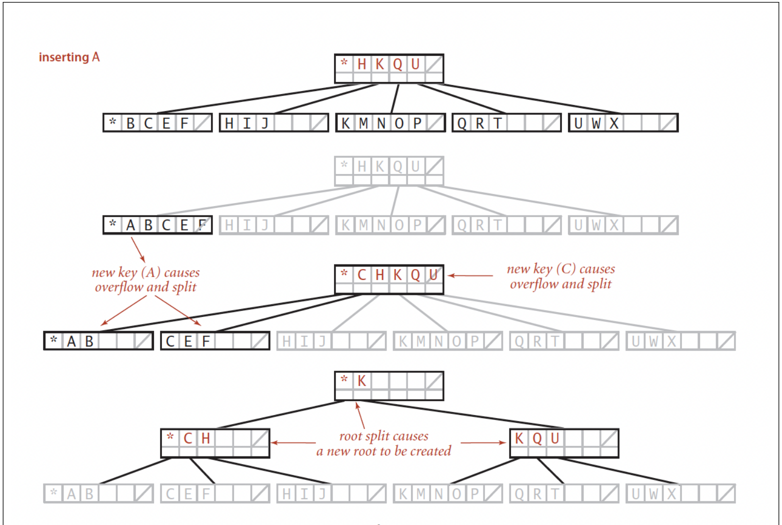
Searching in a B-tree set (M = 6)

Insert in a B-tree - I

Insert in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with M key-link pairs on the way up the tree.

Insert in a B-tree - II


inserting A
* H K Q U
* B C E F
H I J
K M N O P
Q R T
U W X
* A B C E F
new key (A) causes overflow and split
* C H K Q U
new key (C) causes overflow and split
* A B
C E F
* K
* C H
root split causes a new root to be created
K Q U

Inserting a new key into a B-tree set

Balance in B-Trees

Proposition A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.

Proof Sketch All internal nodes (besides root) have between $M/2$ and $M-1$ links.

In practice Number of probes is at most 4. For instance, $M = 1024$ and $N = 62 \texttt{ billion}$, $\log_{M/2} N \leq 4$.

Optimization Always keep root page in memory.

Balanced Trees Uses

Red-black trees are widely used as system symbol tables.

- Java: java.util.TreeMap, java.util.TreeSet.
- C++ STL: map, multimap, multiset.
- Linux kernel: completely fair scheduler, linux/rbtree.h.

B-tree variants. B+ tree, B* tree, B# tree, ...

B-trees (and variants) are widely used for file systems and databases.

- Windows: NTFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.