Searching: Balanced Search Trees

Neerja Mhaskar

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. Janicki's course slides

Smybol 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()`

Challenge: Guarantee performance.

Answer: 2-3 trees, left-leaning red-black BSTs, B-trees.

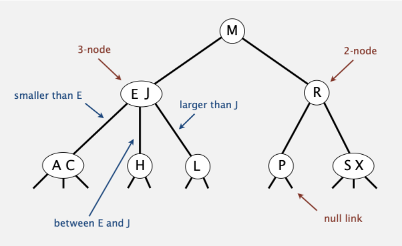
2-3 Search Trees

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

Symmetric order. Inorder traversal yields keys in ascending order.

Perfect balance. Every path from root to null link has same length - how to maintain?



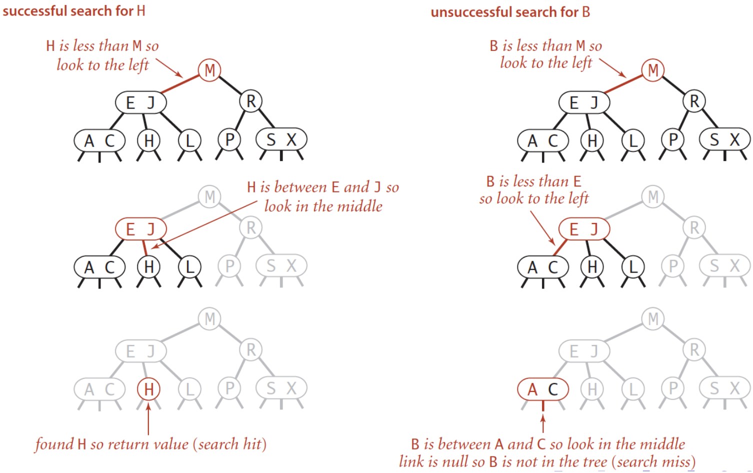
2-3 Search Tree: Search - I

Search

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

See Demo - `https://algs4.cs.princeton.edu/lectures/`

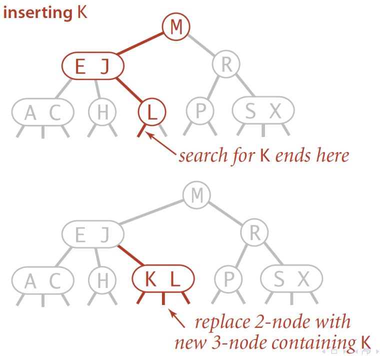
2-3 Search Tree: Search - II



2-3 Search Tree: Insert I

Insertion into a 2-node at bottom

- Add new key to 2-node to create a 3-node.



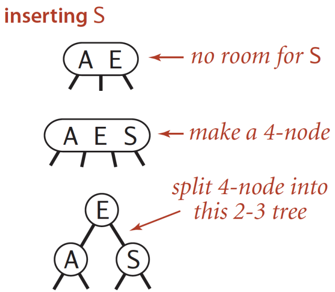
2-3 Search Tree: Insert II

Insertion into a 3-node at bottom.

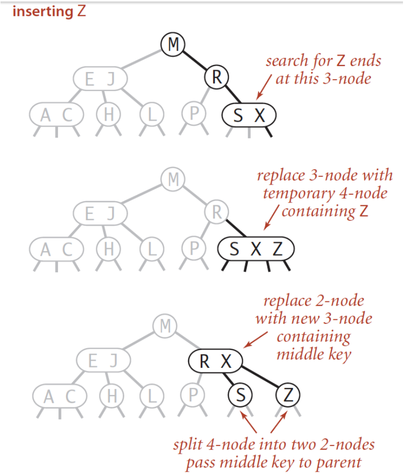
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node (3 keys and 4 links) into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

2-3 Search Tree: Insert II

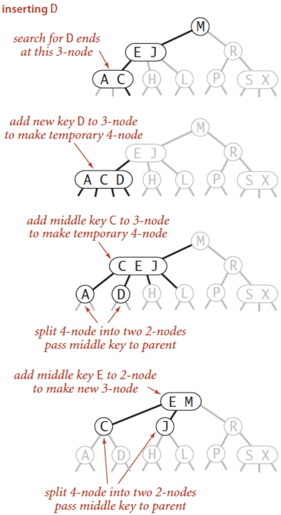
Insertion into a 3-node at bottom into a single 3-node root.



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



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



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

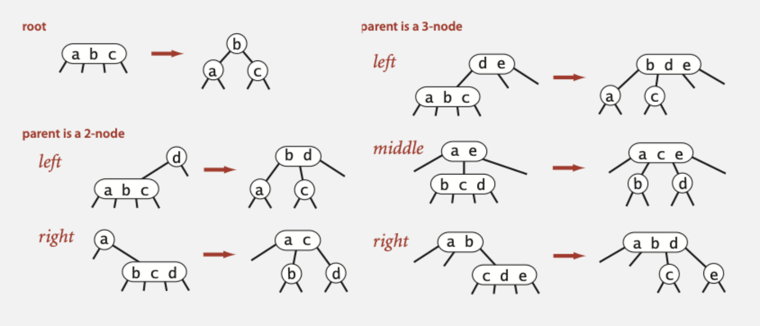
2-3 tree construction demo

See Demo - `https://algs4.cs.princeton.edu/lectures/`

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

Splitting a 4-node is a local transformation: constant number of operations as it involves one of six transformations:

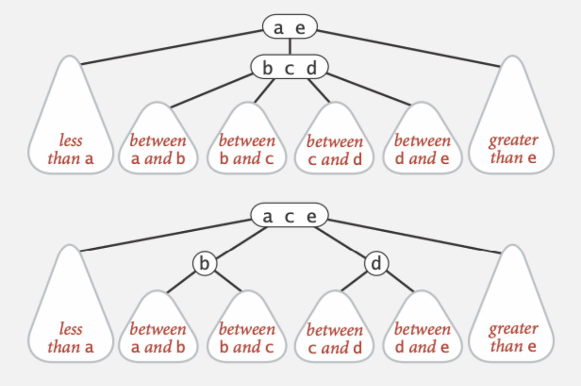
The 4-node may be the root; it may be the left or right child of a 2-node; or it may be the left, middle, or right child of a 3-node.



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

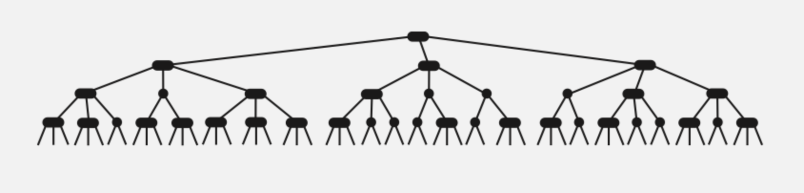
Invariants. Maintains symmetric order and perfect balance.

Proof. Each transformation maintains symmetric order and perfect balance.



2-3 tree: performance

Perfect balance. 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

Bottom line. 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

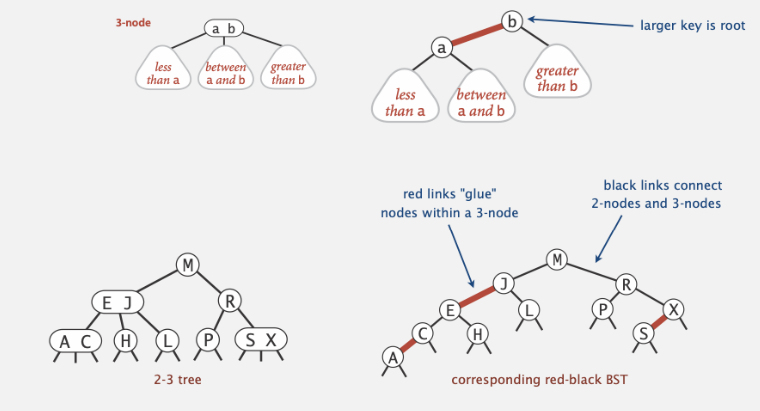
implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	½ N	½ N	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$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

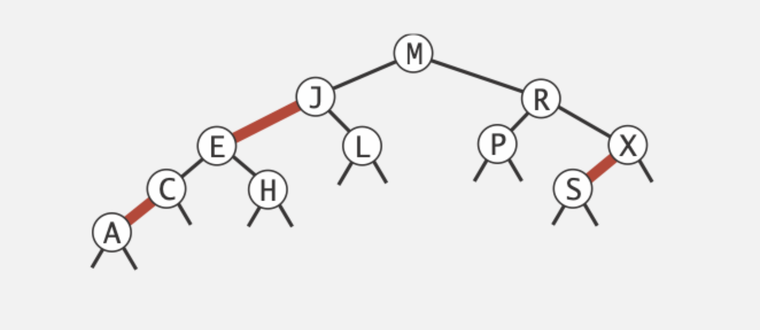
- Represent 2-3 tree as a BST.
- Use “internal” left-leaning links as “glue” for 3-nodes.



An equivalent definition of Red-Black Trees

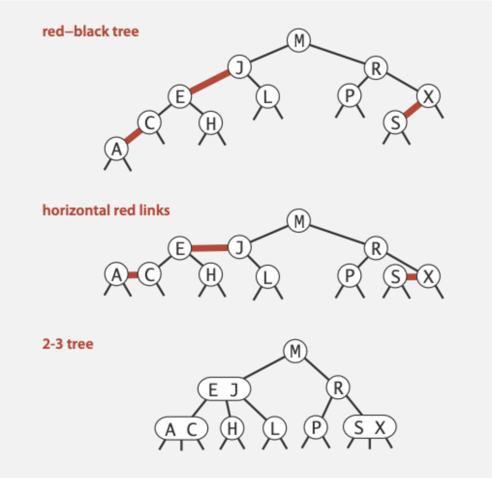
A BST such that:

- 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.



Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees

Key property. 1-1 correspondence between 2-3 and LLRB.



Red-black BST representation

Each node is pointed to by precisely one link (from its parent) ⇒ can encode color of links in nodes (use Boolean variable *color*).

Boolean variable *color* - is true if the link from the parent is red, and false if it is black. By convention, null links are black.

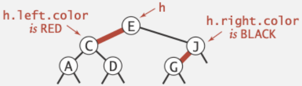
In the book, the color of a node ⇔ color of the link pointing to it

```
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;
}
```

null links are black

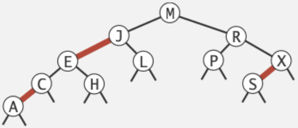


Search implementation for red-black BSTs

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

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

During internal operations, maintain:

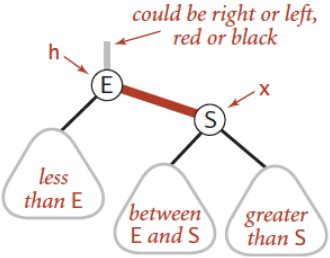
- Symmetric order.
- Perfect black balance.

[but not necessarily color invariants]

How? Apply elementary red-black BST operations: rotation and color flip.

Elementary red-black BST operations: Left Rotation I

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

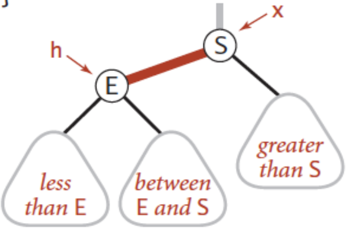



```
Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations: Left Rotation II

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



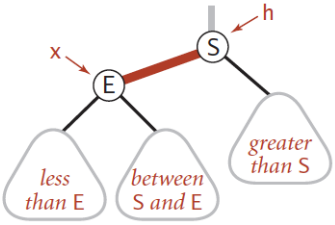
Left rotate (right link of h)

```
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;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations: Right Rotation I

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

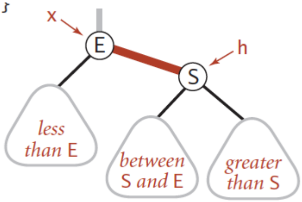



```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations: Right Rotation II

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



Right rotate (left link of h)

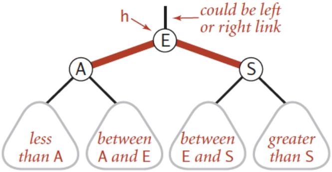
```
Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   x.N = h.N;
   h.N = 1 + size(h.left)
           + size(h.right);
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Note: Implementing a right rotation that converts a left-leaning red link to a right-leaning one amounts to the same code, with left and right interchanged.

Elementary red-black BST operations: Color Flip I

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

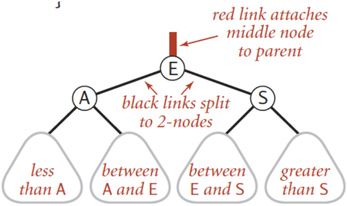



```
void flipColors(Node h)
{
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations: Color Flip II

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



Flipping colors to split a 4-node

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

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.

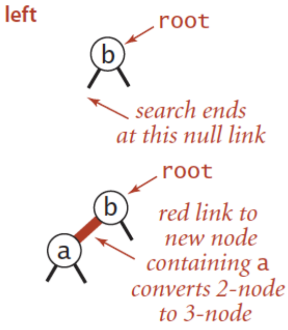


Figure 1: Case: 1

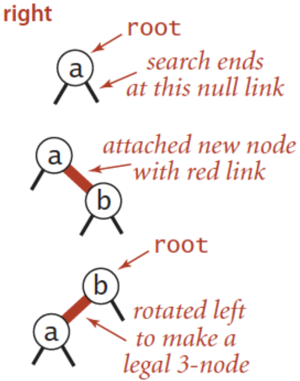


Figure 2: Case: 2

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 as usual into a BST, adding a new node at the bottom (respecting the order), but always connected to its parent 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 it is attached to a right link, we have a 3-node leaning the wrong way - fix it with a left rotation!

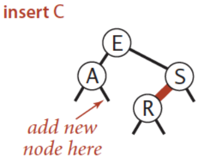


Figure 3: Case: 1

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

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

- Insert keys into a red-black BST as usual into a BST, adding a new node at the bottom (respecting the order), but always connected to its parent 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 it is attached to a right link, we have a 3-node leaning the wrong way - fix it with a left rotation!

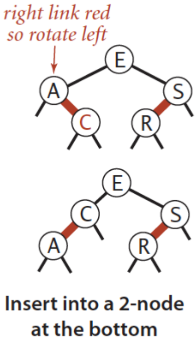
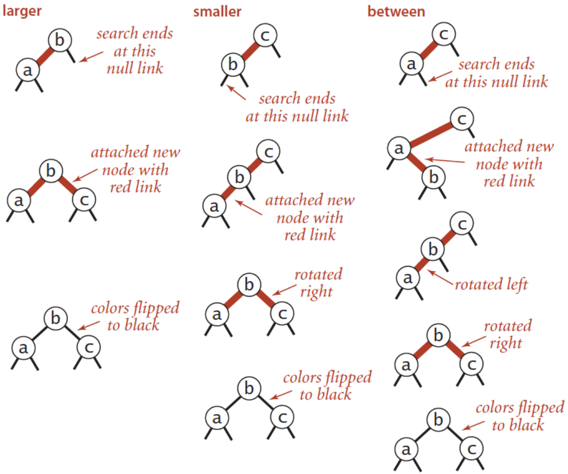


Figure 4: Case: 2

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

Warm up. Insert a node into a two node tree (single 3-node)

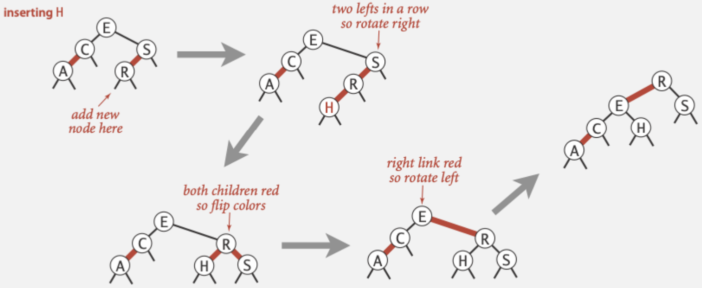


Insert into a tree: Insert a node at the bottom - I

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

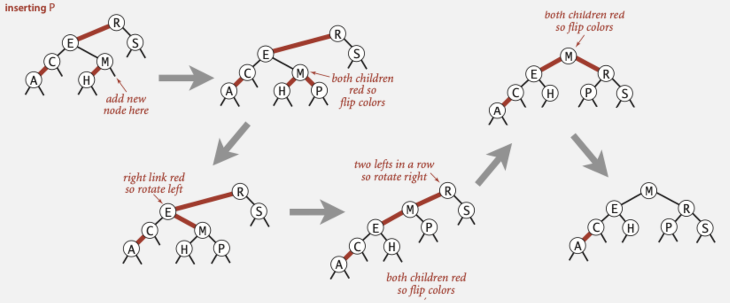


Insert into a tree: Insert a node at the bottom - II

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

- Do standard BST insert; color new link red. ← 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).
- Repeat case 1 or case 2 up the tree (if needed).

← to fix color invariants



Keeping the root black

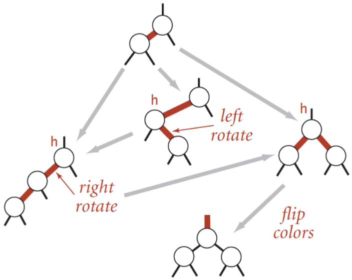
- A red root implies that the root is part of a 3-node, but that is not the case, so we color the root of the tree black after each insertion.
- Note that the black height of the tree increases by 1 whenever the color of the root is flipped from black to red.

See Demo - `https://algs4.cs.princeton.edu/lectures/`

Inserting in a LLRB Tree: Code

Same code for all cases.

- Right child red, left child black (right leaning link) : rotate left
- Left child, left-left grandchild red (two red lefts in a row): rotate right.
- Both children red: flip colors.

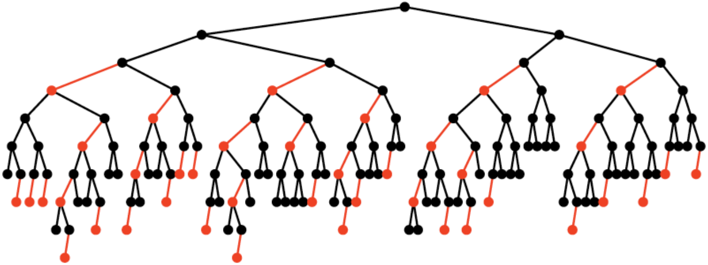


Passing a red link up a red-black BST

RBT Analysis

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

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	$\frac{1}{2} N$	N	$\frac{1}{2} N$		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✔	`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()`

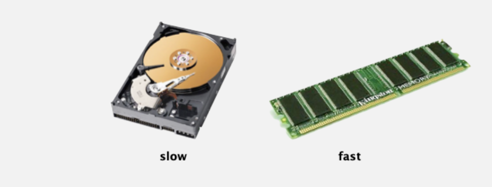
* exact value of coefficient unknown but extremely close to 1

B-Trees

File System Model

Page. Contiguous block of data (e.g., a file or 4,096-byte chunk).

Probe. First access to a page (e.g., from disk to memory).



Property. Time required for a probe is much larger than time to access data within a page.

Cost model. When studying algorithms for external searching, we count page accesses (the number of times a page is accessed, for read or write); that is, number of probes.

Goal. Access data using minimum number of probes.

B-Trees

B-tree. data structures generalize the 2-3 trees, and are based on a multiway balanced search trees. They are particularly devised for external searching.

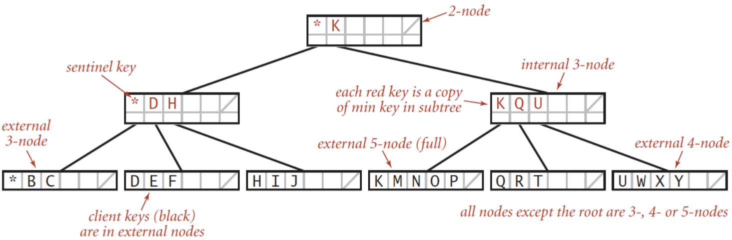
B-tree of order M.

- Allows up to $M - 1$ key-link pairs per node, where link is the address of a page instead of data.
- Choose M (an even number) as large as possible so that M links fit in a page, e.g., M $=$ 1024.
- 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 - that is defined to be less than all other keys is used in B-Trees. It helps to implement B-trees.



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

Search/Insert 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.

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.

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.

Pf. 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 $N = 62$ `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.