Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

3.3 BALANCED SEARCH TREES

- *AVL Trees*
- *2-3 search trees*
- *red-black BSTs*
- *B-trees*

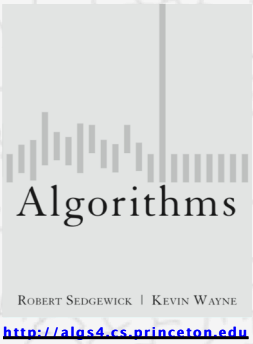
Symbol table review

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search(unordered list)	N	N	N	$\frac{1}{2}\ N$	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()`
goal	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$		`compareTo()`

Challenge. Guarantee performance.

This lecture. 2-3 trees, left-leaning red-black BSTs, B-trees.

2



3.3 Balanced Search Trees

- *AVL trees*
- *2-3 search trees*
- *red-black BSTs*
- *B-trees*

AVL Trees

An AVL tree is a binary search tree with a *balance* condition.

AVL is named for its inventors: Adel'son-Vel'skii and Landis

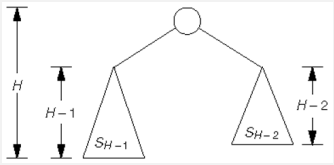
AVL tree *approximates* the ideal tree (completely balanced tree).

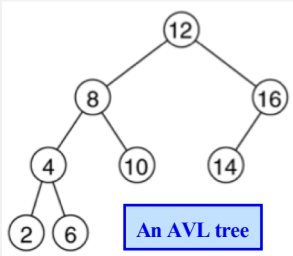
AVL Tree maintains a height close to the minimum.

AVL Trees

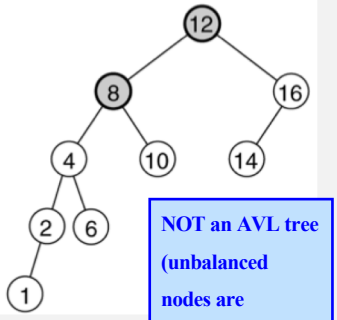
Definition:

An AVL tree is a binary search tree such that for any node in the tree, the height of the left and right subtrees can differ by at most 1.





An AVL tree



NOT an AVL tree (unbalanced nodes are darkened)

AVL Trees -- Properties

The depth of a typical node in an AVL tree is very close to the optimal $log_2 N$.

Consequently, all searching operations in an AVL tree have logarithmic worst-case bounds.

An update (insert or delete) in an AVL tree could destroy the balance.

It must then be rebalanced before the operation can be considered complete.

AVL Tree -- Insertions

Insert in the same as **Binary Search Tree** insertion

Then, starting from insertion point, check for balance at each node

It is enough to perform correction "rotation" only at the first node where imbalance occurs on the path from the inserted node to the root.

AVL -- Insertions

An AVL violation might occur in four possible cases:

1) Insertion into left subtree of left child of node *n*
2) Insertion into right subtree of left child of node *n*
3) Insertion into left subtree of right child of node *n*
4) Insertion into right subtree of right child of node *n*

   (1) and (4) are mirror cases

   (2) and (3) are mirror cases

If insertion occurs "**outside**" (1 & 4), then perform **single rotation**.

If insertion occurs "**inside**" (2 & 3), then perform **double rotation.**

AVL Trees -- Balance Operations

Balance is restored by tree rotations.

There are four different cases for rotations:

1. Single Right Rotation
2. Single Left Rotation
3. Double Right-Left Rotation
4. Double Left-Right Rotation

AVL Trees -- Single Rotation

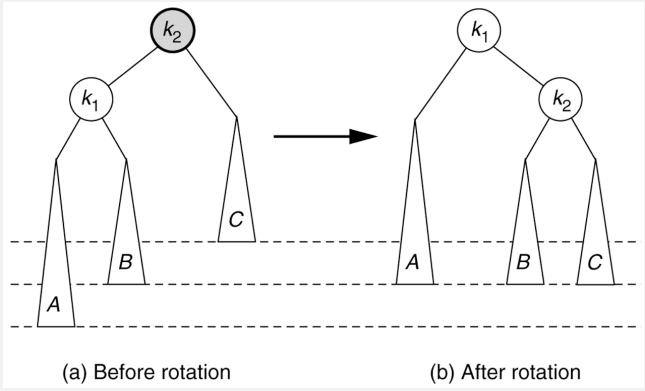
A single rotation switches the roles of the parent and the child while maintaining the search order.

We rotate between a node and its child (left or right).

- Child becomes parent
- Parent becomes right child in Case 1 (single right rotation)
  Parent becomes left child in Case 2 (single left rotation)

The result is a binary search tree that satisfies the AVL property.

Case 1 -- Single Right Rotation

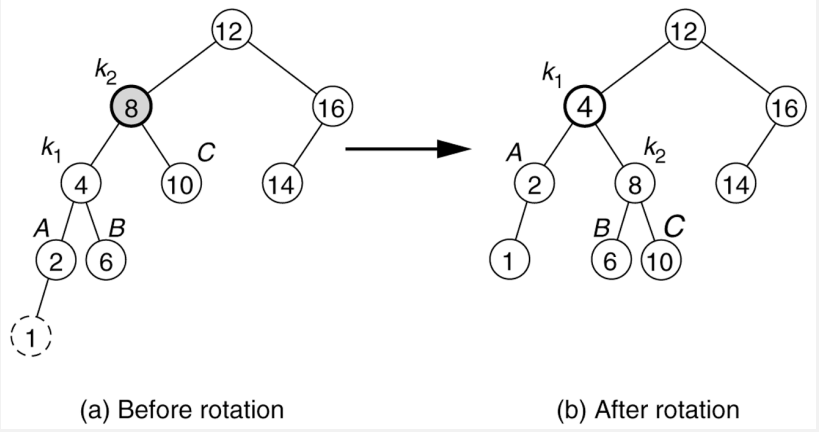


(a) Before rotation (b) After rotation

Child becomes parent

Parent becomes right child

Case 1 -- Single Right Rotation

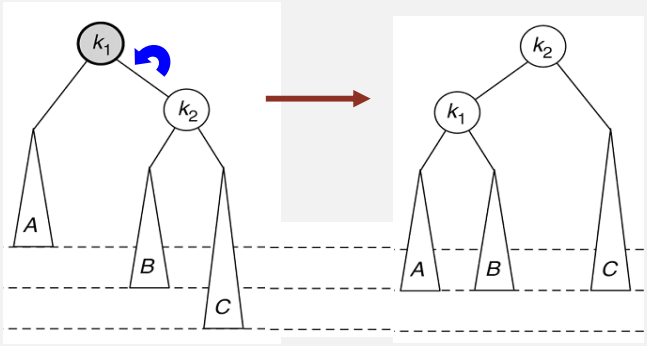


(a) Before rotation (b) After rotation

Child becomes parent

Parent becomes right child

Case 2 – Single Left Rotation



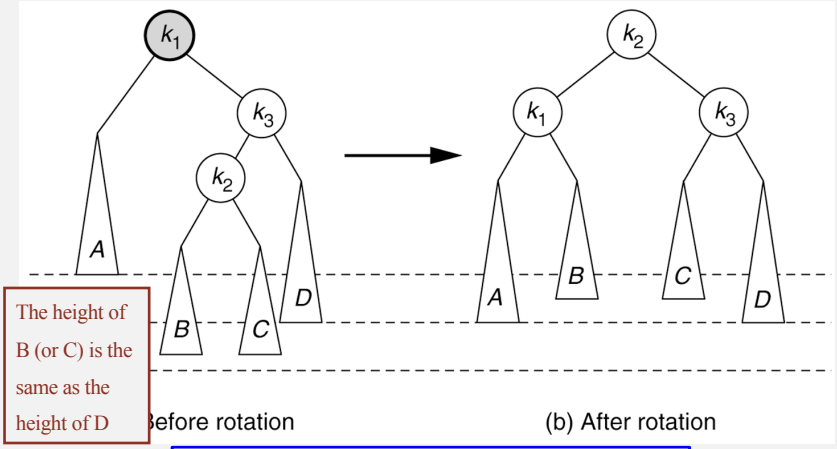
Before Rotation

After Rotation

Child becomes parent

Parent becomes left child

Case 3 -- Double Right-Left Rotation



The height of B (or C) is the same as the height of D

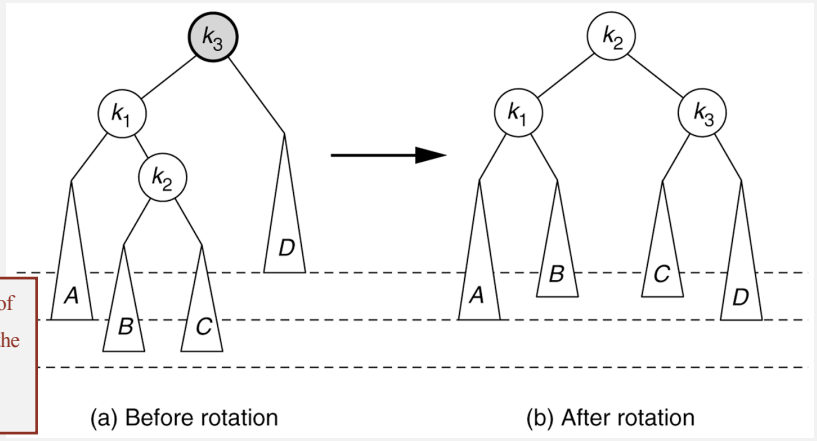
efore rotation

(b) After rotation

First perform single right rotation on k2 and k3

Then perform single left rotation on k2 and k1

Case 4 -- Double Left-Right Rotation



The height of B (or C) is the same as the height of A

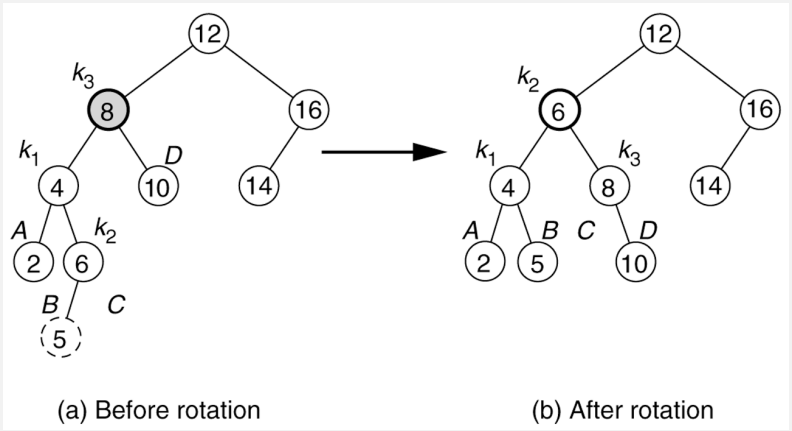
(a) Before rotation (b) After rotation

First perform single left rotation on k2 and k1

Then perform single right rotation on k2 and k3

6/1/21

Case 4 -- Double Left-Right Rotation



(a) Before rotation (b) After rotation

First perform single left rotation on k2 and k1

Then perform single right rotation on k2 and k3

6/1/21

AVL Trees -- Insertion

It is enough to perform rotation only at the first node

- Where imbalance occurs
- On the path from the inserted node to the root.

The rotation takes O(1) time.

After insertion, only nodes that are on the path from the insertion point to the root can have their balances changed.

Hence insertion is O(logN)

AVL Trees -- Insertion

***Exercise*:** Starting with an empty AVL tree, insert the following items

7 6 5 4 3 2 1 8 9 10 11 12

Check the following applet for more exercises.

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

AVL Trees -- Deletion

Deletion is more complicated.

- It requires both single and double rotations
- We may need more than one rebalance operation (rotation) on the path from the deleted node back to the root.

Steps:

- First delete the node the same as deleting it from a binary search tree
  - Remember that a node can be either ***a leaf node*** or ***a node with a single child*** or ***a node with two children***
- Walk through from the deleted node back to the root and rebalance the nodes on the path if required
  - Since a rotation can change the height of the original tree

AVL Trees -- Deletion

Deletion is O(logN)

- Each rotation takes O(1) time
- We may have at most h (height) rotations, where h = O(logN)

AVL Trees -- Deletion

Implementation:

- We have a **`shorter`** flag that shows if a subtree has been shortened
- Each node is associated with a **`balance factor`**
  - ***left-high*** the height of the left subtree is higher than that of the right subtree
  - ***right-high*** the height of the right subtree is higher than that of the left subtree
  - ***equal*** the height of the left and right subtrees is equal

Deletion algorithm:

- Shorter is initialized as true
- Starting from the deleted node back to the root, take an action depending on
  - The value of shorter
  - The balance factor of the current node
  - Sometimes the balance factor of a child of the current node
- Until shorter becomes false

AVL Trees -- Deletion

Three cases according to the balance factor of the current node

1.The balance factor is equal

➔ ***no rotation***

2.The balance factor is not equal and the taller subtree was shortened

➔ ***no rotation***

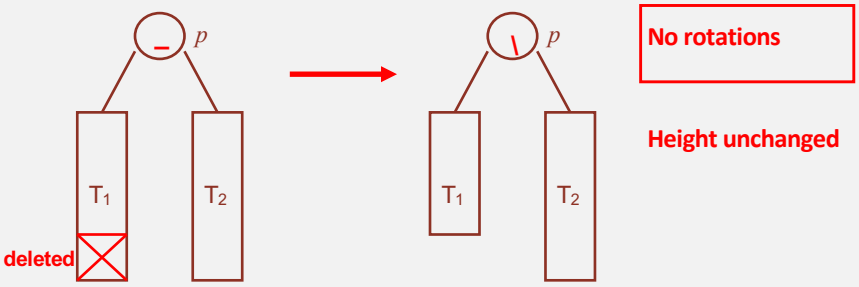
3.The balance factor is not equal and the shorter subtree was shortened

➔ ***rotation is necessary***

AVL Trees -- Deletion

Case 1: ***The balance factor of p is equal.***

- Change the balance factor of *p* to *right-high* (or *left-high*)
- Shorter becomes false



No rotations

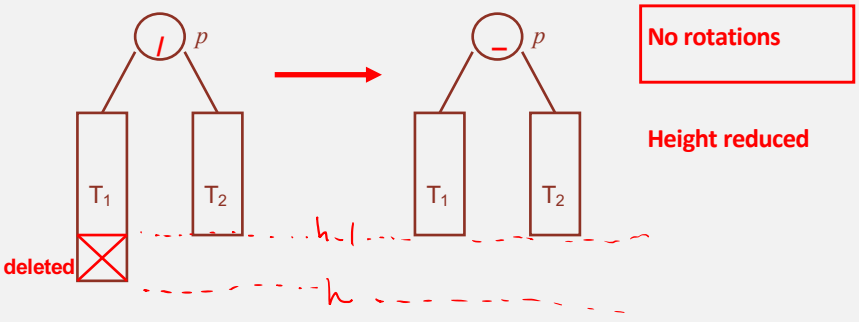
Height unchanged

6/1/21

AVL Trees -- Deletion

Case 2: ***The balance factor of p is not equal and the taller subtree is shortened.***

- Change the balance factor of *p* to *equal*
- Shorter remains true



No rotations

Height reduced

6/1/21

AVL Trees -- Deletion

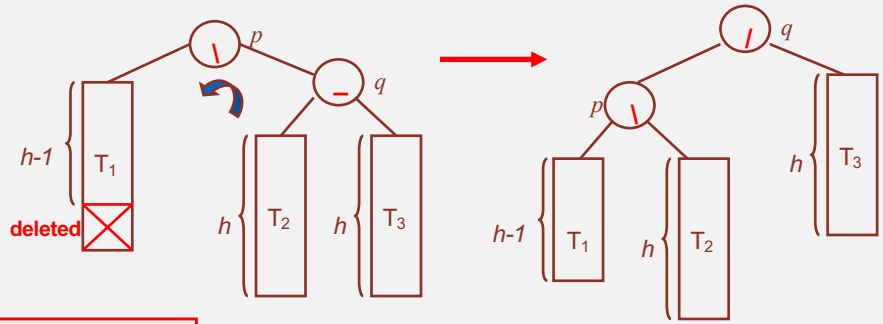
Case 3: ***The balance factor of p is not equal and the shorter subtree is shortened.***

- Rotation is necessary
- Let *q* be the root of the taller subtree of *p*
- We have three sub-cases according to the balance factor of *q*

AVL Trees -- Deletion

Case 3a: ***The balance factor of q is equal.***

- Apply a single rotation
- Change the balance factor of *q* to *left-high* (or *right-high*)
- Shorter becomes false



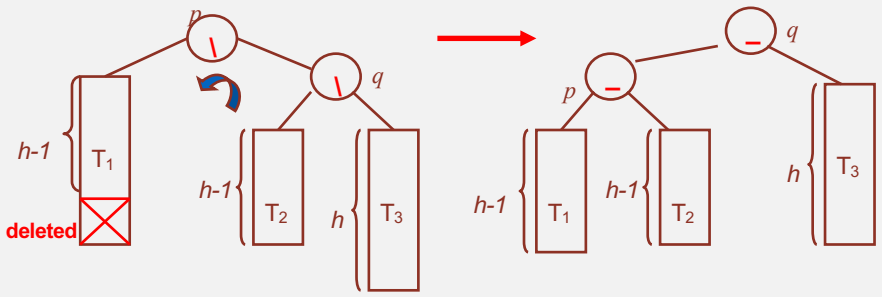
Single rotation

Height unchanged

AVL Trees -- Deletion

Case 3b: ***The balance factor of q is the same as that of p.***

- Apply a single rotation
- Change the balance factors of *p* and *q* to *equal*
- Shorter remains true



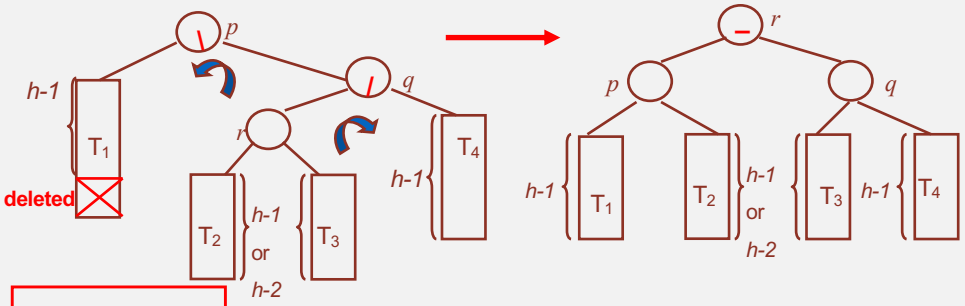
Single rotation

Height reduced

AVL Trees -- Deletion

Case 3c: ***The balance factor of q is the opposite of that of p.***

- Apply a double rotation
- Change the balance factor of the new root to *equal*
- Also change the balance factors of *p* and *q*
- Shorter remains true

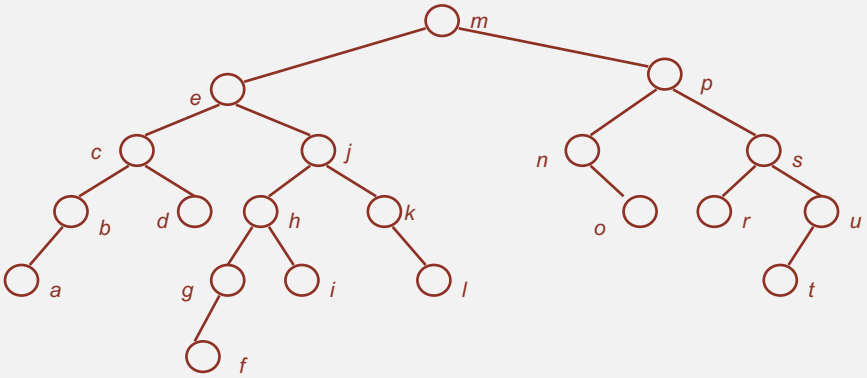


Double rotation

Height reduced

AVL Trees -- Deletion

Exercise: Delete *o* from the following AVL tree



Check the following applets for more exercises.

http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html
http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html

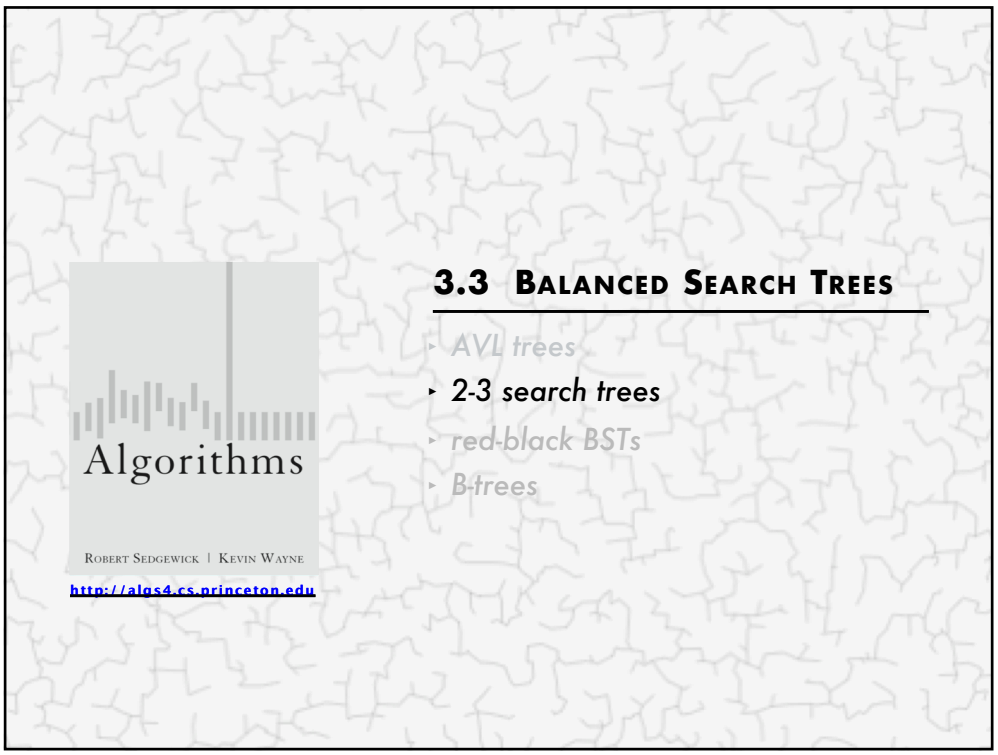
AVL Trees -- Analysis

H	minN	logN	H / logN
4	7	2,81	1,42
5	12	3,58	1,39
6	20	4,32	1,39
7	33	5,04	1,39
8	54	5,75	1,39
9	88	6,46	1,39
10	143	7,16	1,40
11	232	7,86	1,40
12	376	8,55	1,40
13	609	9,25	1,41
14	986	9,95	1,41
15	1.596	10,64	1,41
16	2.583	11,33	1,41
17	4.180	12,03	1,41
18	6.764	12,72	1,41
19	10.945	13,42	1,42
20	17.710	14,11	1,42
30	**2.178.308**	**21,05**	**1,42**
40	**267.914.295**	**28,00**	**1,43**
50	**32.951.280.098**	**34,94**	**1,43**

What is the minimum number of nodes in an AVL tree?

minN(0) = 0
minN(1) = 1
minN(2) = 2
minN(3) = 4
...
minN(h) = minN(h-1) + minN(h-2) + 1

Max height of an N-node AVL tree is less than 1.44 log N



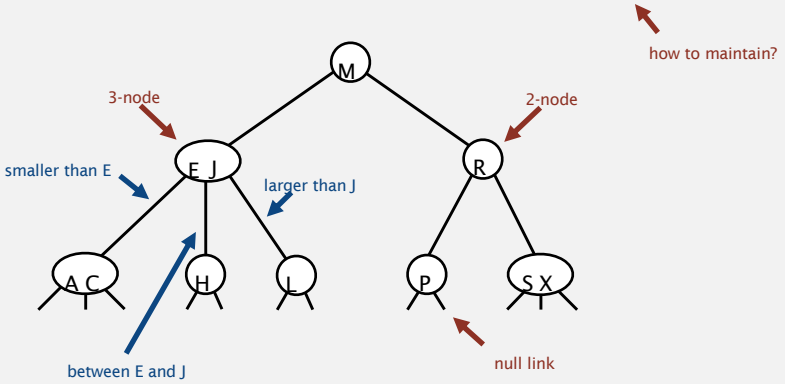
2-3 tree

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.



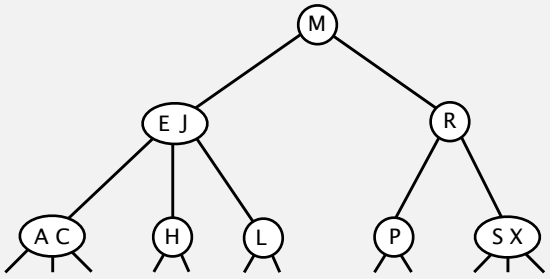
32

2-3 tree demo

Search.

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

search for H



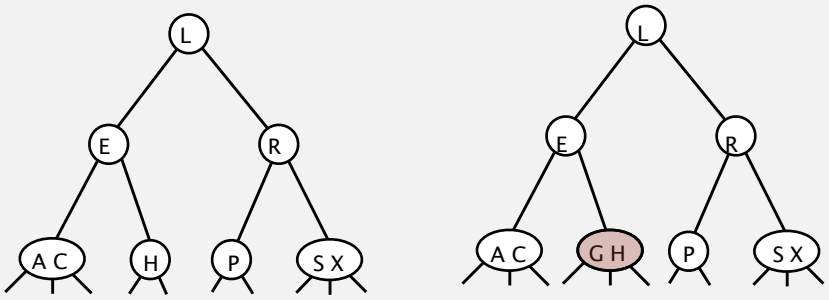
33

Insertion into a 2-3 tree

Insertion into a 2-node at bottom.

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

insert G



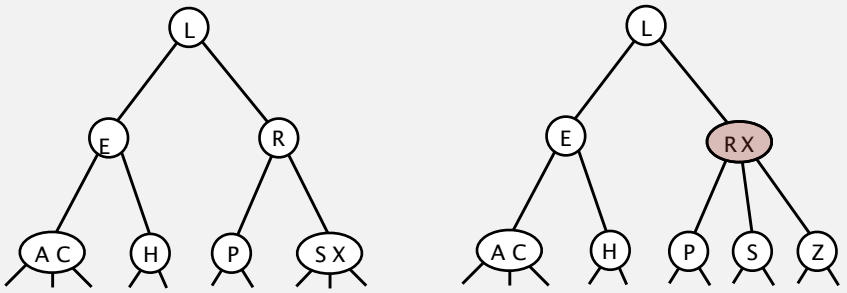
34

Insertion into a 2-3 tree

Insertion into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node 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.

insert Z



35

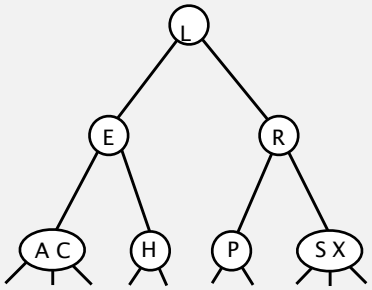
2-3 tree construction demo

insert S

36

2-3 tree construction demo

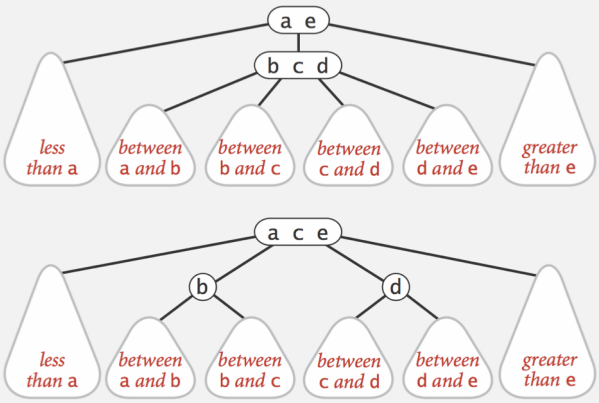
2–3 tree



37

Local transformations in a 2-3 tree

Splitting a 4-node is a local transformation: constant number of operations.

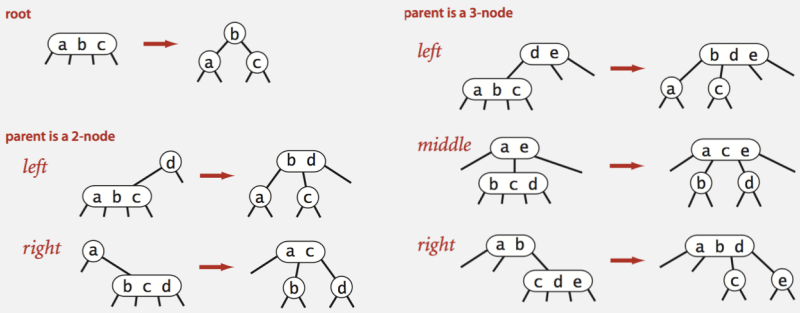


38

Global properties in a 2-3 tree

Invariants. Maintains symmetric order and perfect balance.

Pf. 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:
- Best case:

2-3 tree: performance

Perfect balance. Every path from root to null link has same length.

Tree height.

- Worst case: $\lg N$. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

Bottom line. Guaranteed logarithmic performance for search and insert.

41

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()`

constant c depend upon implementation

42

2-3 tree: 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.

fantasy code

```
public void put(Key key, Value val)
{
   Node x = root;
   while (x.getTheCorrectChild(key) != null)
   {
      x = x.getTheCorrectChildKey();
      if (x.is4Node()) x.split();
   }
   if      (x.is2Node()) x.make3Node(key, val);
   else if (x.is3Node()) x.make4Node(key, vl);
}
```

Bottom line. Could do it, but there's a better way.

43

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

3.3 Balanced Search Trees

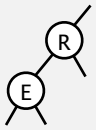
- AVL trees
- 2-3 search trees
- red-black BSTs
- B-trees

How to implement 2-3 trees with binary trees?

Challenge. How to represent a 3 node?

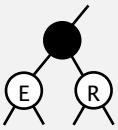
Approach 1: regular BST.

- No way to tell a 3-node from a 2-node.
- Cannot map from BST back to 2-3 tree.



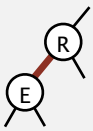
Approach 2: regular BST with "glue" nodes.

- Wastes space, wasted link.
- Code probably messy.



Approach 3: regular BST with red "glue" links.

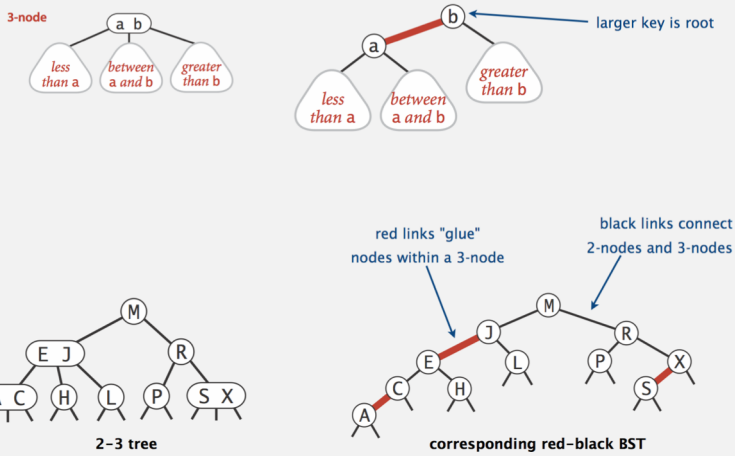
- Widely used in practice.
- Arbitrary restriction: red links lean left.



46

Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

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

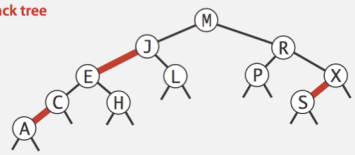


47

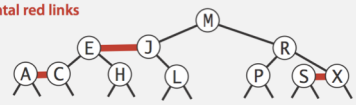
Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees

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

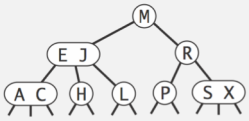
red–black tree



horizontal red links



2-3 tree



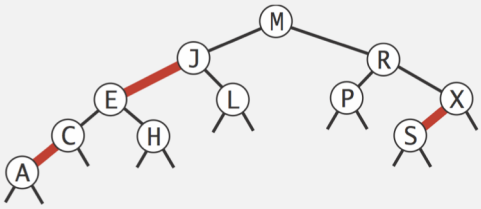
48

An equivalent definition

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.
- Red links lean left.

"perfect black balance"



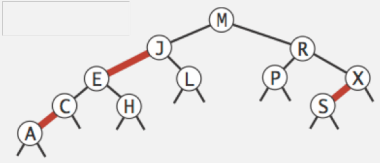
49

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.

50

Red-black BST representation

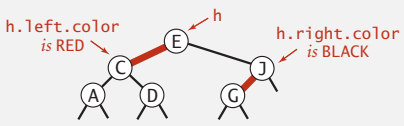
Each node is pointed to by precisely one link (from its parent) ⇒ can encode color of links in nodes.

```
private static 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



51

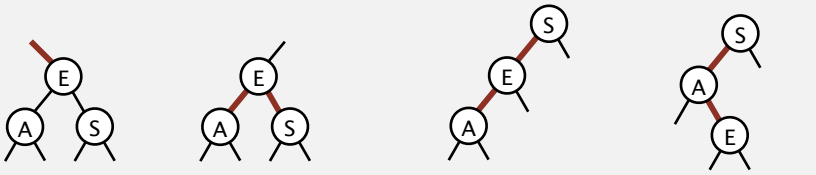
Insertion in a LLRB tree: overview

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

During internal operations, maintain:

- Symmetric order.
- Perfect black balance.

  [but not necessarily color invariants]



right-leaning red link | two red children (a temporary 4-node) | left-left red (a temporary 4-node) | left-right red (a temporary 4-node)

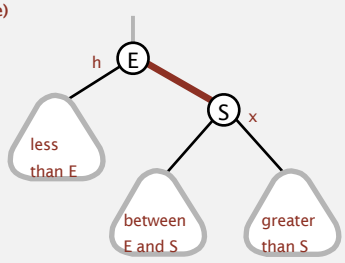
How? Apply elementary red-black BST operations: rotation and color flip.

52

Elementary red-black BST operations

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

rotate E left (before)



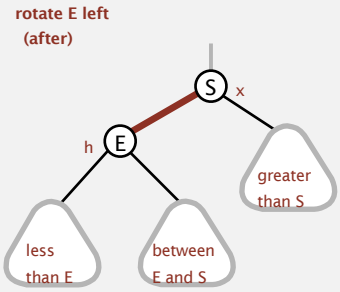

```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

53

Elementary red-black BST operations

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



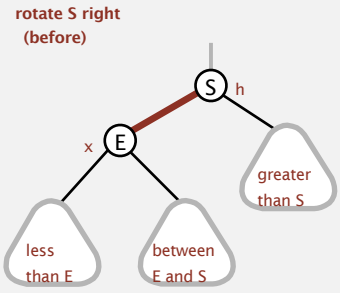

```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

54

Elementary red-black BST operations

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



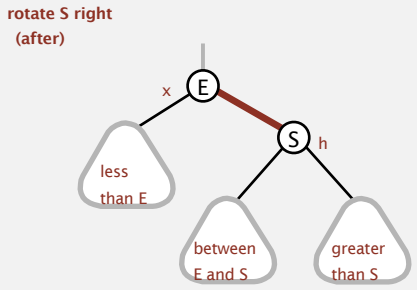

```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

55

Elementary red-black BST operations

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



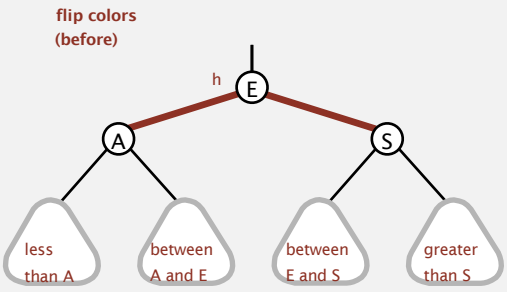

```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

56

Elementary red-black BST operations

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



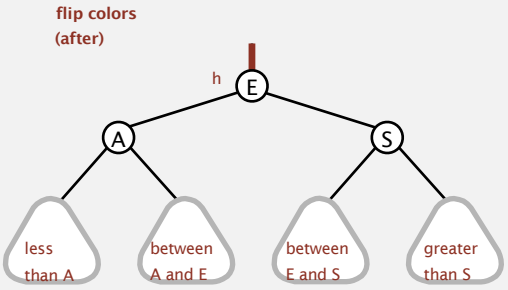

```
private void flipColors(Node h)
{
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

57

Elementary red-black BST operations

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



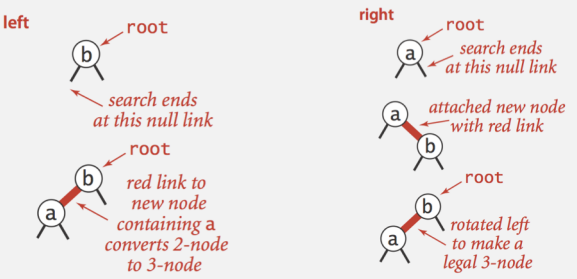

```
private void flipColors(Node h)
{
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

58

Insertion in a LLRB tree

Warmup 1. Insert into a tree with exactly 1 node.

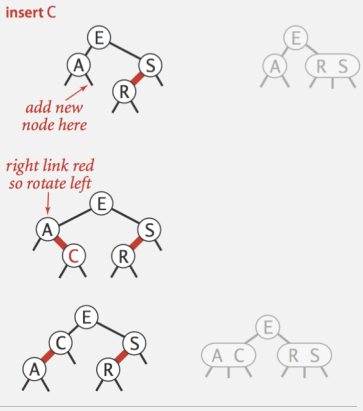


59

Insertion in a LLRB tree

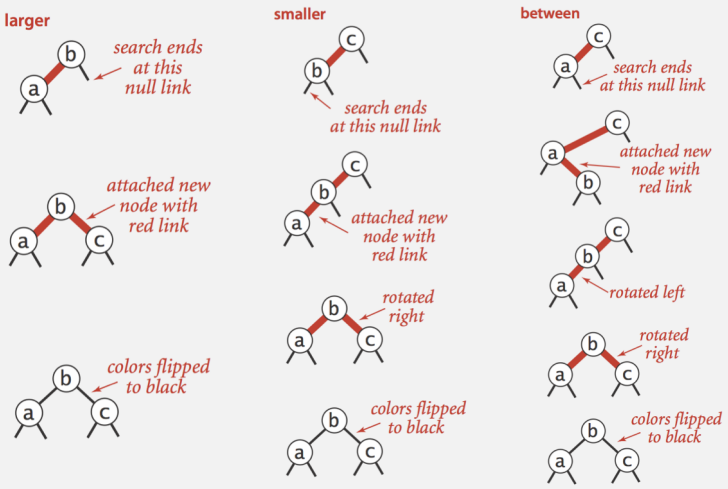
Case 1. Insert into a 2-node at the bottom.

- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- If new red link is a right link, rotate left. ← to fix color invariants



Insertion in a LLRB tree

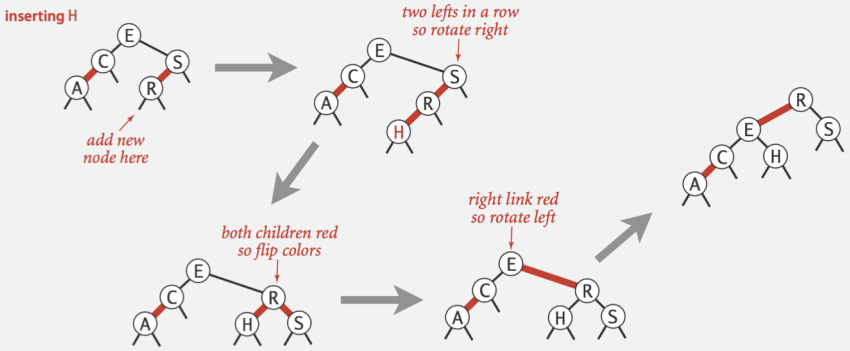
Warmup 2. Insert into a tree with exactly 2 nodes.



Insertion in a LLRB tree

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

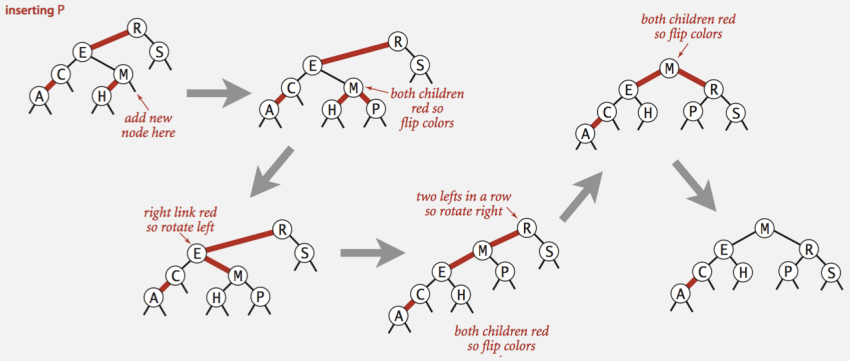
- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed). ← to fix color invariants



Insertion in a LLRB tree: passing red links up the tree

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
- Repeat case 1 or case 2 up the tree (if needed).



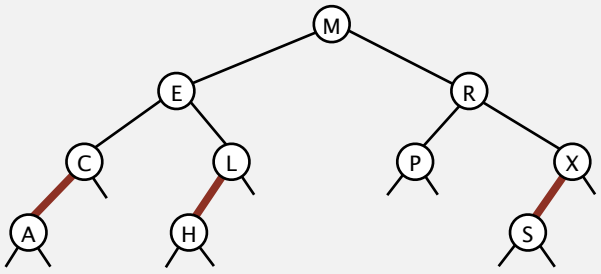
Red-black BST construction demo

insert S

64

Red-black BST construction demo

red-black BST

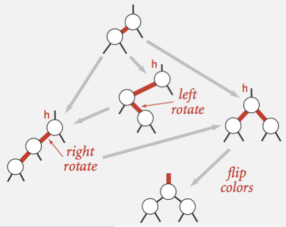


65

Insertion in a LLRB tree: Java 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.




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

   if (isRed(h.right) && !isRed(h.left))     h = rotateLeft(h);
   if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h);
   if (isRed(h.left)  && isRed(h.right))     flipColors(h);

   return h;
}
```

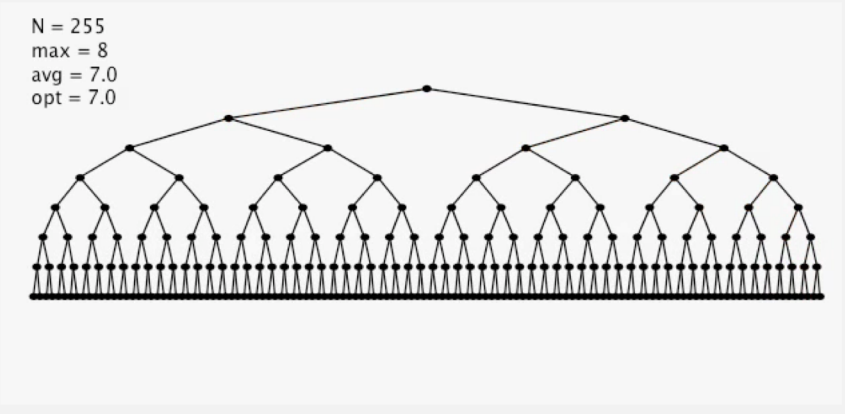
insert at bottom (and color it red)

lean left
balance 4-node
split 4-node

only a few extra lines of code provides near-perfect balance

66

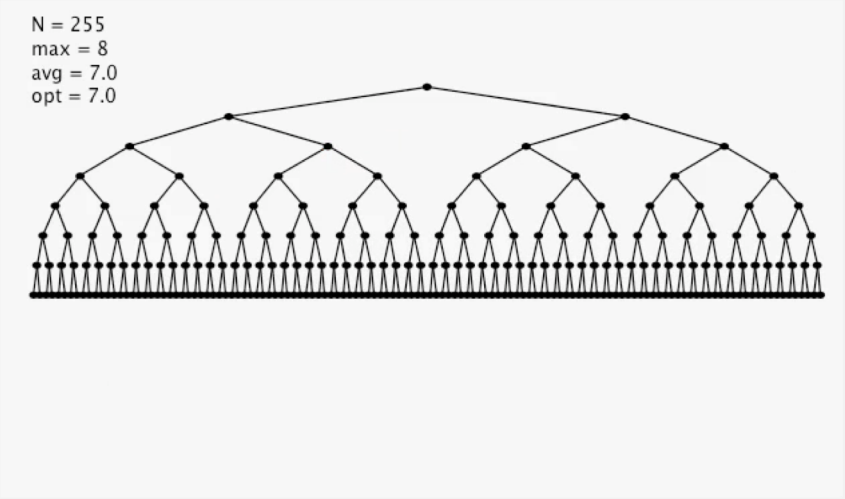
Insertion in a LLRB tree: visualization



255 insertions in ascending order

67

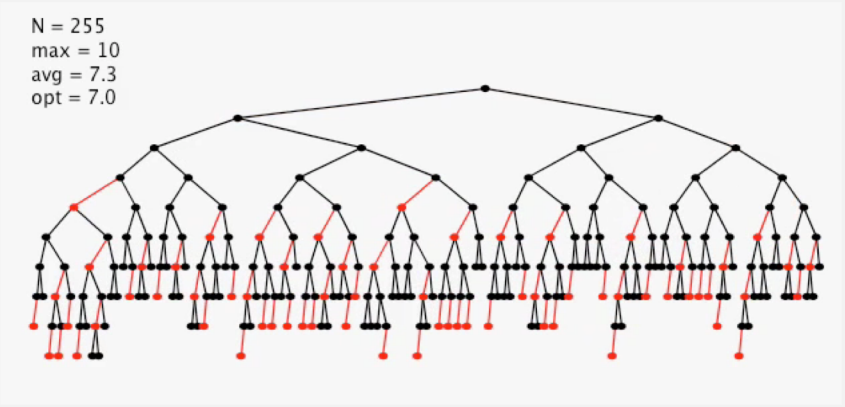
Insertion in a LLRB tree: visualization



255 insertions in descending order

68

Insertion in a LLRB tree: visualization



255 random insertions

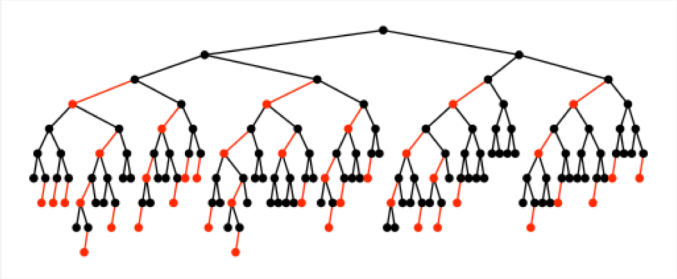
69

Balance in LLRB trees

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

Pf.

- Every path from root to null link has same number of black links.
- Never two red links in-a-row.



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

70

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

71

Why left-leaning trees?

old code (that students had to learn in the past)

```
private Node put(Node x, Key key, Value val, boolean sw)
{
   if (x == null)
      return new Node(key, value, RED);
   int cmp = key.compareTo(x.key);

   if (isRed(x.left) && isRed(x.right))
   {
      x.color = RED;
      x.left.color  = BLACK;
      x.right.color = BLACK;
   }
   if (cmp < 0)
   {
      x.left = put(x.left, key, val, false);
      if (isRed(x) && isRed(x.left) && sw)
         x = rotateRight(x);
      if (isRed(x.left) && isRed(x.left.left))
      {
         x = rotateRight(x);
         x.color = BLACK; x.right.color = RED;
      }
   }
   else if (cmp > 0)
   {
      x.right = put(x.right, key, val, true);
      if (isRed(h) && isRed(x.right) && !sw)
         x = rotateLeft(x);
      if (isRed(h.right) && isRed(h.right.right))
      {
         x = rotateLeft(x);
         x.color = BLACK; x.left.color = RED;
      }
   }
   else x.val = val;
   return x;
}
```

extremely tricky

new code (that you have to learn)

```
public Node put(Node h, Key key, Value val)
{
   if (h == null)
      return new Node(key, val, RED);
   int cmp = kery.compareTo(h.key);
   if (cmp < 0)
      h.left  = put(h.left,  key, val);
   else if (cmp > 0)
      h.right = put(h.right, key, val);
   else h.val = val;

   if (isRed(h.right) && !isRed(h.left))
      h = rotateLeft(h);
   if (isRed(h.left) && isRed(h.left.left))
      h = rotateRight(h);
   if (isRed(h.left) && isRed(h.right))
      flipColors(h);

  return h;
}
```

straightforward
(if you've paid attention)

72

Why left-leaning red-black BSTs?

Simplified code.

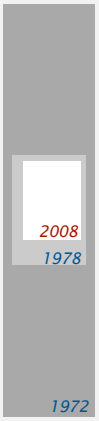
- Left-leaning restriction reduces number of cases.
- Short inner loop.

Same ideas simplify implementation of other operations.

- Delete min/max.
- Arbitrary delete.

Improves widely-used balanced search trees.

- AVL trees, splay trees, randomized BSTs, ...
- 2-3 trees, 2-3-4 trees.
- Red-black BSTs.



Bottom line. Left-leaning red-black BSTs are among the simplest balanced BSTs to implement and among the fastest in practice.

73

War story: why red-black?

Xerox PARC innovations. [1970s]

- Alto.
- GUI.
- Ethernet.
- Smalltalk.
- InterPress.
- Laser printing.
- Bitmapped display.
- WYSIWYG text editor.
- ...

Xerox Alto

A DICHROMATIC FRAMEWORK FOR BALANCED TREES

Leo J. Guibas
Xerox Palo Alto Research Center,
Palo Alto, California, and
Carnegie-Mellon University

and

Robert Sedgewick*
Program in Computer Science
Brown University
Providence, R. I.

ABSTRACT

In this paper we present a uniform framework for the implementation and study of balanced tree algorithms. We show how to imbed in this

the way down towards a leaf. As we will see, this has a number of significant advantages over the older methods. We shall examine a number of variations on a common theme and exhibit full implementations which are notable for their brevity. One implementation is examined carefully, and some properties about its

74

War story: red-black BSTs

Telephone company contracted with database provider to build real-time database to store customer information.

Database implementation.

- Red-black BST search and insert; Hibbard deletion.
- Exceeding height limit of 80 triggered error-recovery process.

allows for up to 2^{40} keys

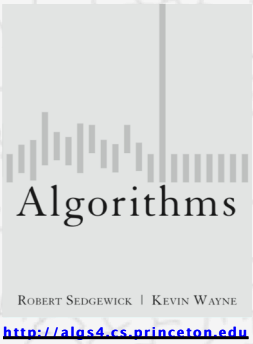
Extended telephone service outage.

Hibbard deletion was the problem

- Main cause = height bound exceeded!
- Telephone company sues database provider.
- Legal testimony:

> *" If implemented properly, the height of a red-black BST with N keys is at most 2 lg N. "* — *expert witness*

75



3.3 Balanced Search Trees

- AVL trees
- 2-3 search trees
- red-black BSTs
- 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).

slow

fast

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

Cost model. Number of probes.

Goal. Access data using minimum number of probes.

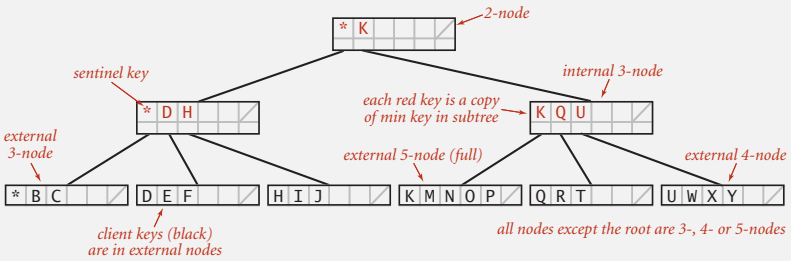
78

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2-3 trees by allowing up to $M - 1$ key-link pairs per node.

- At least 2 key-link pairs at root.
- At least $M / 2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

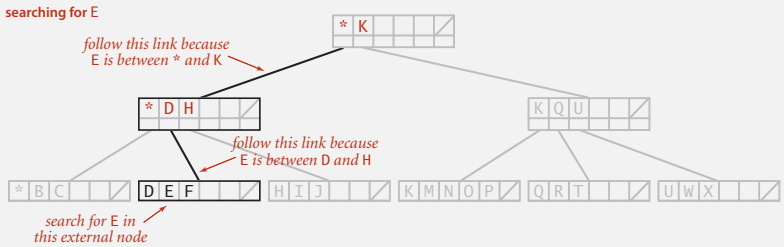
choose M as large as possible so that M links fit in a page, e.g., M = 1024



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

Searching in a B-tree

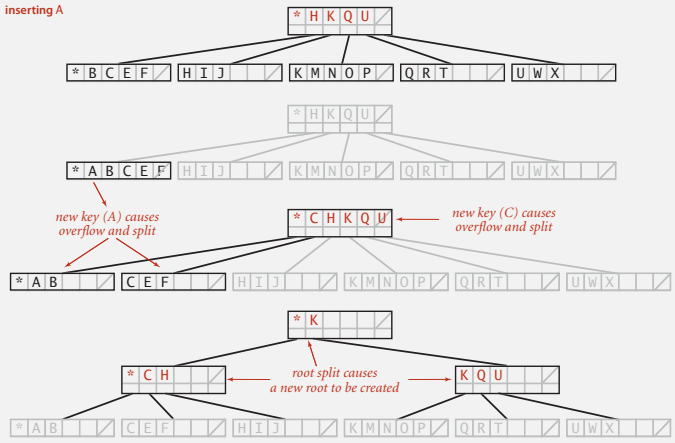
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



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

Insertion in a B-tree

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



Inserting a new key into a B-tree set

81

Balance in B-tree

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. ← M = 1024; N = 62 billion
$\log_{M/2} N \leq 4$

Optimization. Always keep root page in memory.

82

Building a large B tree

83

Balanced trees in the wild

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`.
- Emacs: conservative stack scanning.

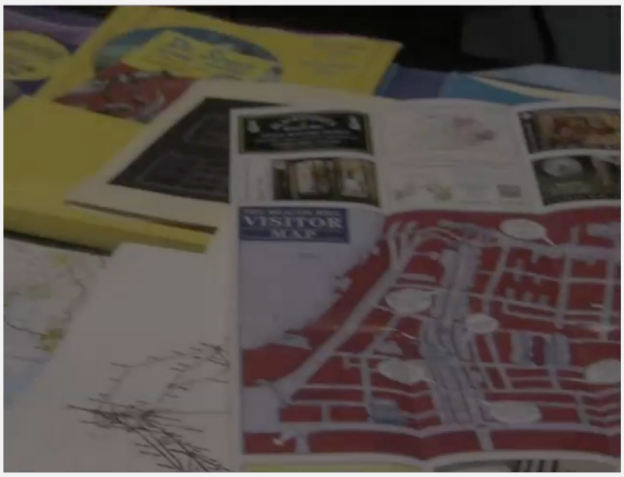
B-tree variants. B+ tree, B*tree, B# tree, …

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.

84

Red-black BSTs in the wild



Common sense. Sixth sense. Together they're the FBI's newest team.

85

Red-black BSTs in the wild

ACT FOUR

FADE IN:

48 INT. FBI HQ - NIGHT 48

Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS OF PRINTOUTS.

JESS
It was the red door again.

POLLOCK
I thought the red door was the storage container.

JESS
But it wasn't red anymore. It was black.

ANTONIO
So red turning to black means... what?

POLLOCK
Budget deficits? Red ink, black ink?

NICOLE
Yes. I'm sure that's what it is. But maybe we should come up with a couple other options, just in case.

Antonio refers to his COMPUTER SCREEN, which is filled with mathematical equations.

ANTONIO
It could be an algorithm from a binary search tree. A red-black tree tracks every simple path from a node to a descendant leaf with the same number of black nodes.

JESS
Does that help you with girls?

86