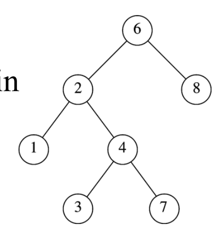
Search Trees

- Support dynamic set operations:
  - Insert
  - Delete
  - Search
  - Min
  - Max
  - Predecessor
  - Successor
- Can simultaneously be used as a *dictionary* and as a *priority queue*
- Operations run in $O(h)$ time, h = height of tree
- Many variants: binary search trees, red-black trees, AVL trees, 2-3-4 trees, B-trees, randomized search trees, splay trees, etc.

1

Binary Search Trees

- A type of search tree
- *Linked* binary tree structure
- Each node p contains:
  - *key*: unique value from totally ordered domain
  - *left*: points to left child
  - *right*: points to right child
  - *p*: points to parent



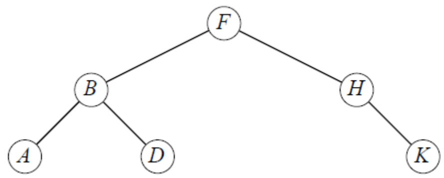
- $root[T]$ points to root node (also denoted by $T.root$)
- Binary search tree property:

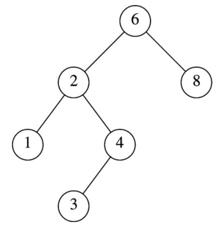
  if q is a descendant of $p.left \Rightarrow q.key < p.key$

  if q is a descendant of $p.right \Rightarrow q.key > p.key$

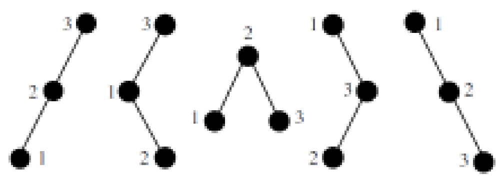
2

Examples, $n = 6$





- How many BST's are there for each n?

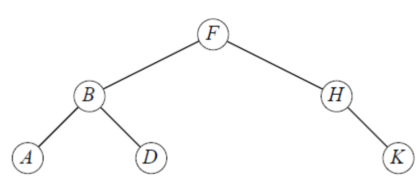


3

Traversals

- Systematic way to visit all nodes
- Three types: *in-order*, *pre-order*, *post-order*
- Each takes $\Theta(n)$ time

```
INORDER-TREE-WALK(x)
if x ≠ NIL
  then INORDER-TREE-WALK(left[x])
       print key[x]
       INORDER-TREE-WALK(right[x])
```



4

Search

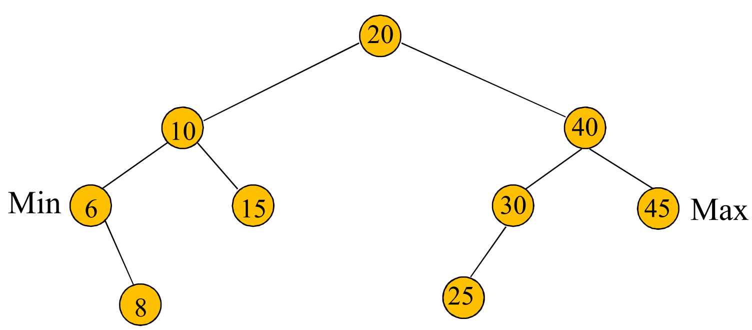
- Initial call Tree-Search(*root*[*T*], *k*)

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
   then return x
if k < key[x]
   then return TREE-SEARCH(left[x], k)
   else  return TREE-SEARCH(right[x], k)
```

- Time? $O(h)$

5

Minimum and Maximum



- Can be found using tree structure alone
- Follow rightmost path to max element
- Follow leftmost path to min element

6

Minimum and Maximum

```
TREE-MINIMUM(x)
while left[x] ≠ NIL
    do x ← left[x]
return x
```

```
TREE-MAXIMUM(x)
while right[x] ≠ NIL
    do x ← right[x]
return x
```

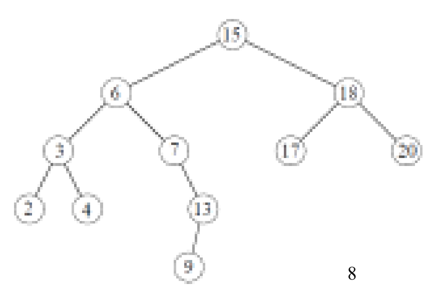
- Can also write recursively.
- Time? $O(h)$

7

Successor

- Successor of x is node y containing the smallest key > $x.key$
- Can be found using tree structure (no key comparisons necessary!)
- Two cases depending on whether $x.right$ = NIL

```
TREE-SUCCESSOR(x)
if right[x] ≠ NIL
  then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
    do x ← y
       y ← p[y]
return y
```

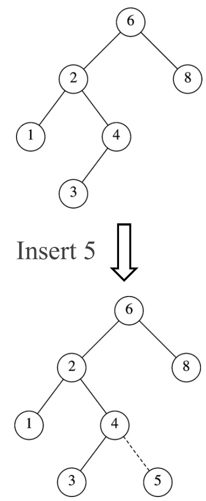


8

Insertion

- *z* initialized with NIL left and right pointers

```
TREE-INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
    do y ← x
       if key[z] < key[x]
         then x ← left[x]
         else x ← right[x]
p[z] ← y
if y = NIL
  then root[T] ← z               ▷ Tree T was empty
  else if key[z] < key[y]
         then left[y] ← z
         else right[y] ← z
```



9

Tree-Delete(T, z)

TREE-DELETE is broken into three cases.

Case 1: z has no children.

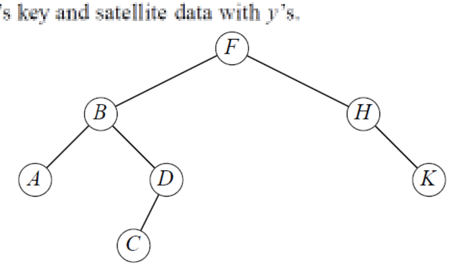
- Delete z by making the parent of z point to NIL, instead of to z.

Case 2: z has one child.

- Delete z by making the parent of z point to z's child, instead of to z.

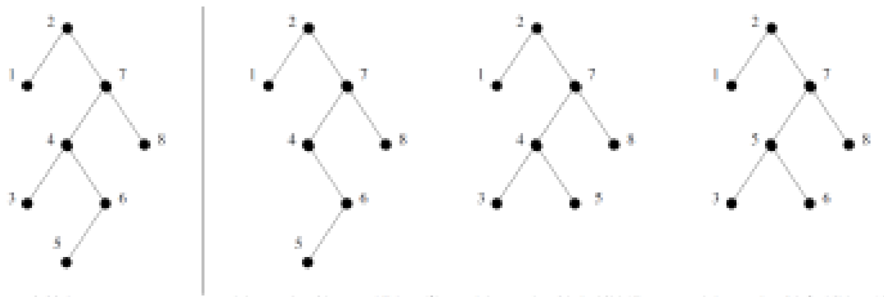
Case 3: z has two children.

- z's successor y has either no children or one child. (y is the minimum node—with no left child—in z's right subtree.)
- Delete y from the tree (via Case 1 or 2).
- Replace z's key and satellite data with y's.



10

Examples



initial tree — delete node with zero children (3) — delete node with 1 child (6) — delete node with 2 children (4)

11

Analysis

- All operations take $O(h)$ time and h is $\Theta(\log n)$ in the best case and $\Theta(n)$ in the worst case
- Key to efficiency is to restructure the tree when h gets too big
- Does randomness help?

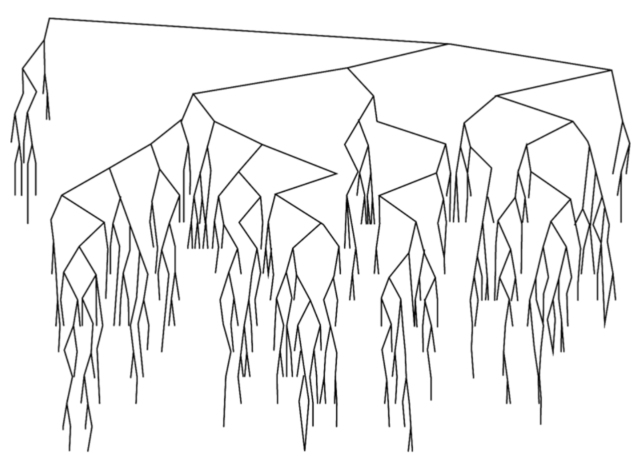
12

Randomly Built BST

- Given a set of n distinct keys enter them in random order into empty BST
- Each of n! permutations equally likely for insertion order
- Different from assuming that every BST on n keys is equally likely.
- Can show:
  - average node depth is $O(\log n)$
  - expected height is $O(\log n)$

13

Example: $n = 500$

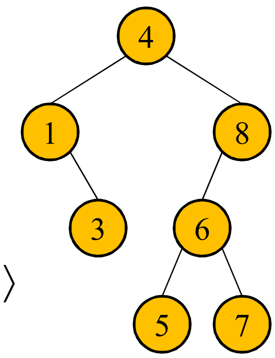


14

Using a BST to Sort

BST-Sort(A)
  $T = \varnothing$
  for $i = 1$ **to** n **do**
    Tree-Insert(T, $A[i]$)
  In-order(T)
end

- Example: sort $\langle 4,1,8,6,3,7,5 \rangle$
- Time?



15

Analysis

- The in-order traversal takes $O(n)$ time.
- How long do n tree inserts take?

$$P_n = \sum_{v \in T} \text{depth}(v)$$

  – Worst case
  – Best case
  – Average case
- Running time is always $\Omega(n \log n)$ and $O(n^2)$

  $O(\log n)$ height $\Rightarrow O(n \log n)$ running time

16

Average Case

Let T be a BST of n nodes

Internal path length $P_n := \sum_{x \in T} \text{depth}(x)$

$P_1 = 0$ and $P_n = P_{i-1} + P_{n-i} + n - 1$

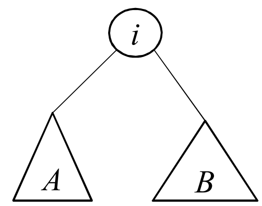
Define $P_0 := 0$ and let $\overline{P}_n := E(P_n)$

Then, if root has rank $R_n = i$

$P_n = P_{i-1} + P_{n-i} + n - 1$, and

$$\overline{P}_n = \frac{1}{n}\sum_{i=1}^{n}(\overline{P}_{i-1} + \overline{P}_{n-i} + n - 1)$$

$$n\overline{P}_n = 2\sum_{i=0}^{n-1}\overline{P}_i + n(n-1)$$



$|A| = i-1, \ |B| = n-i$

17

Average Case…

$$n\overline{P}_n = 2\sum_{i=0}^{n-1}\overline{P}_i + n(n-1)$$

$$(n-1)\overline{P}_{n-1} = 2\sum_{i=0}^{n-2}\overline{P}_i + (n-1)(n-2)$$

$$n\overline{P}_n - (n-1)\overline{P}_{n-1} = 2\overline{P}_{n-1} + 2(n-1)$$

$$n\overline{P}_n = (n+1)\overline{P}_{n-1} + 2(n-1)$$

$$n\overline{P}_n \le (n+1)\overline{P}_{n-1} + 2n$$

$$\frac{\overline{P}_n}{n+1} \le \frac{\overline{P}_{n-1}}{n} + \frac{2}{n+1}$$

18

Average Case…

$$
\begin{aligned}
\frac{\overline{P}_n}{n+1} &\le \frac{\overline{P}_{n-1}}{n} + \frac{2}{n+1} \\
&\le \frac{\overline{P}_{n-2}}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\le \frac{\overline{P}_{n-3}}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\vdots \\
&\le \frac{\overline{P}_1}{2} + \frac{2}{3} + \frac{2}{4} + \cdots + \frac{2}{n} + \frac{2}{n+1} < 2H_{n+1}
\end{aligned}
$$

$$\Rightarrow \overline{P}_n \le 2(n+1)H_{n+1} = O(n \log n)$$

19

Analysis

- How long do n tree inserts take?

$$P_n = \sum_{v \in T} \text{depth}(v)$$

- Worst case is $\Theta(n^2)$.
- Best case is $\Theta(n \log n)$
- Average case (when n! permutations equally likely) is $\Theta(n \log n)$
- What algorithm does this remind you of?

20

Relation to Quicksort

- BST sort and quicksort make the same comparisons, although in different order

  Example: $\langle 4,1,8,6,3,7,5 \rangle \Leftrightarrow \langle 5,7,3,6,8,1,4 \rangle$
- Worst case is $\Theta(n^2)$
- Best case is $\Theta(n \log n)$
- Average case comes from Quicksort analysis

  $\Rightarrow$ Use Randomized BST Sort

  1. Randomly permute A
  2. BST-Sort(A)

21

Expected Tree Height

Let X_n = height of T (a random variable)

$Y = 2^{X_n}$, exponential height of T (also a r.v.)

Suppose the root has rank i, $1 \le i \le n$. Then,

$$X_n = 1 + \max\{X_{i-1}, X_{n-i}\}$$

$$Y_n = 2^{X_n} = 2 \cdot \max\{Y_{i-1}, Y_{n-i}\} \le 2 \cdot (Y_{i-1} + Y_{n-i})$$

$$E[Y_n] \le \frac{1}{n} \sum_{i=1}^{n} 2 \cdot E(Y_{i-1} + Y_{n-i})$$

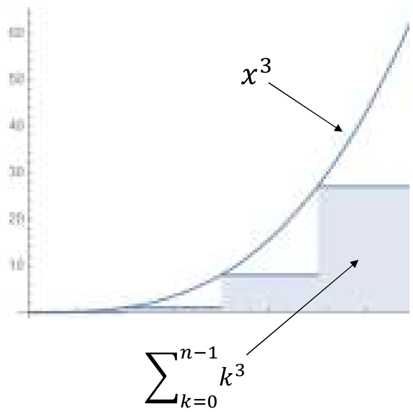
$$= \frac{4}{n} \sum_{i=1}^{n} E[Y_{i-1}] = \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k]$$

22

Claim. $E[Y_n] = O(n^3)$.

Proof (by induction): will show $E[Y_n] \le cn^3$

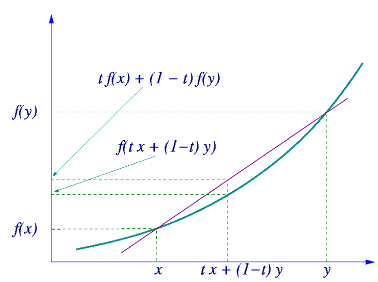
$$
\begin{aligned}
E[Y_n] &\le \frac{4}{n}\sum_{k=0}^{n-1} E[Y_k] \\
&\le \frac{4}{n}\sum_{k=0}^{n-1} ck^3 \\
&= \frac{4c}{n}\sum_{k=0}^{n-1} k^3 \\
&\le \frac{4c}{n}\int_0^n x^3 dx \\
&= cn^3
\end{aligned}
$$



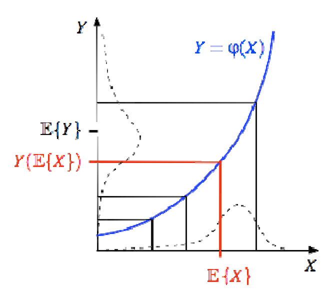
23

Jensen's Inequality

Definition. A real valued function f defined on an interval is **convex** if for any two points x and y in its domain and $0 \le t \le 1$,
$f(tx + (1-t)y) \le t f(x) + (1-t)f(y)$



Theorem. Let X be a random variable and f: $R \rightarrow R$, a convex function. Then $f(E[X]) \le E[f(X)]$



24

Conclusion

$f(x) = 2^x$ is convex!

$$f(E(X_n)) = 2^{E(X_n)} \le E(f(X_n)) = E(2^{X_n}) = E(Y_n)$$
$$2^{E(X_n)} \le E(Y_n) \le cn^3$$

$$E(X_n) \le 3\log n + c'$$

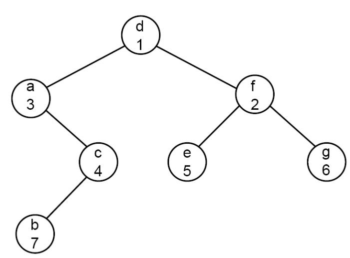
Theorem. The expected height of a binary search tree randomly built on n keys is $O(\log n)$

25

Treap = (Search) tree + Heap

- A treap is a binary tree.
- Each node contains an element x with key(x) $\in U$ and priority $\rho(x) \in R$.
- The following hold
  - *Search tree property*. Same as in regular BSTs
  - *Heap property*. For all x, $\rho(x) > \rho(\text{parent}[x])$

key	a	b	c	d	e	f	g
priority	3	7	4	1	5	2	6



26

Treap Uniqueness

- **Lemma.** Let X be a set of keys with distinct priorities $\rho : X \to R$. Then there is a unique binary search tree for X that satisfies the heap order given by ρ.

  Proof. By induction
- Structurally, the treap has the structure that would result if the elements were inserted in priority order

27

Randomized Search Trees

- Treaps with <u>random</u> priorities on a subset S of a universe $(U, <)$ of keys with a total order
- Priorities interpreted as "arrival times"
- Operations
  - Search(x,S): Is $x \in S$?
  - Insert(x,S): Insert x into S if not already in S
  - Delete(x,S): Delete x from S
  - Minimum(S): Return smallest key.
  - Maximum(S): Return largest key.
  - Union(S_1 ,S_2): Merge S_1 and S_2 .

    Precondition: $\forall\, x_1 \in S_1$, $x_2 \in S_2$: $x_1 < x_2$
  - Split(S,x,S_1,S_2): Split S into S_1, $\{x\}$ and S_2.

    $\forall\, x1 \in S_1$, $x2 \in S_2$: $x1 < x$ and $x < x2$

28

Operations

- Search (x,S), Min(S), Max(S), Pred(x,S), Succ(x,S):
  - Same as BSTs $\Rightarrow$ can do in $O(\log n)$ expected time
- Insert(x,S)
- Delete(x,S)
- Split(S,x,S_1,S_2): Split S into S_1, $\{x\}$ and S_2.

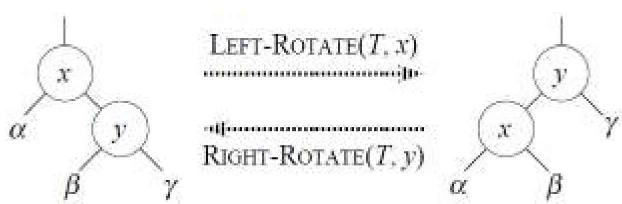
  Goal: $\forall\ x1 \in S_1$, $x2 \in S_2$: $x1 < x$ and $x < x2$
- Union(S_1 ,S_2): Merge S_1 and S_2 .

  Precondition: $\forall\ x_1 \in S_1$, $x_2 \in S_2$: $x_1 < x_2$

29

Rotations

- Basic tree restructuring operation
- Preserves BST (order) property
- Accomplished by $O(1)$ pointer changes

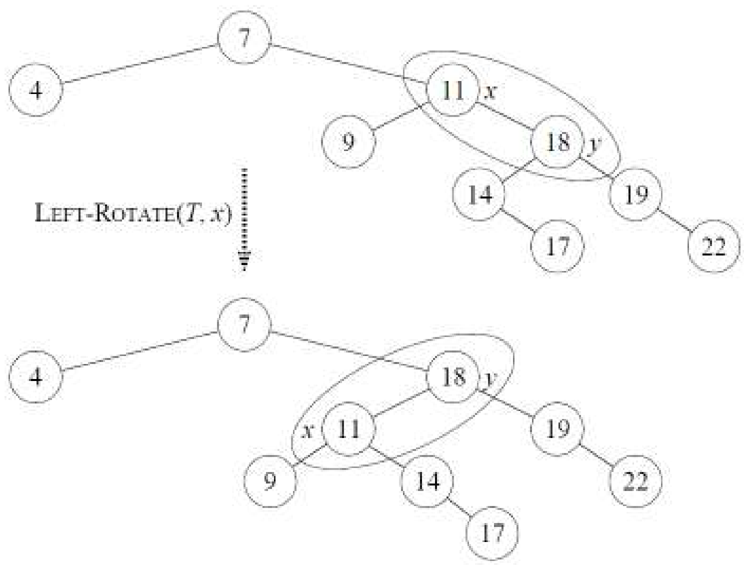


- In both cases, in-order traversal yields

$$\langle\alpha\rangle\ x\ \langle\beta\rangle\ y\ \langle\gamma\rangle$$

30

Example



Operations…

- Insert(x,S)

  Insert x as in regular BSTs. Assign random $\rho(x)$ in [0,1]. Rotate x up until $\rho(x) > \rho(\mathrm{p}[x])$
- Delete(x,S)

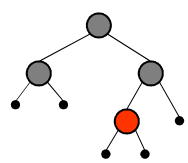
  Change $\rho(x)$ to ∞. Rotate x down (by rotating child with smaller ρ up) until heap order is restored. Remove x which is now a leaf.
- Split(S,x,S_1,S_2): Split S into S_1, $\{x\}$ and S_2 with $y < x$ if y in S_1 and $y > x$ if y in S_2

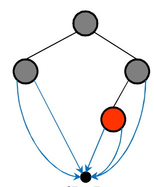
  Change $\rho(x)$ to $-\infty$. Rotate x up to root. Return S_1=left(x) and S_1=right(x)

32

Red-Black Trees

- A type of binary search trees
  - Additional color attribute: red or black
  - Tree is full (all nodes, except leaves, have degree 2)
  - All leaves are empty (keys reside in internal nodes)
- Balanced: height is $O(\log n)$
- Operations take $\Theta(\log n)$ in worst case





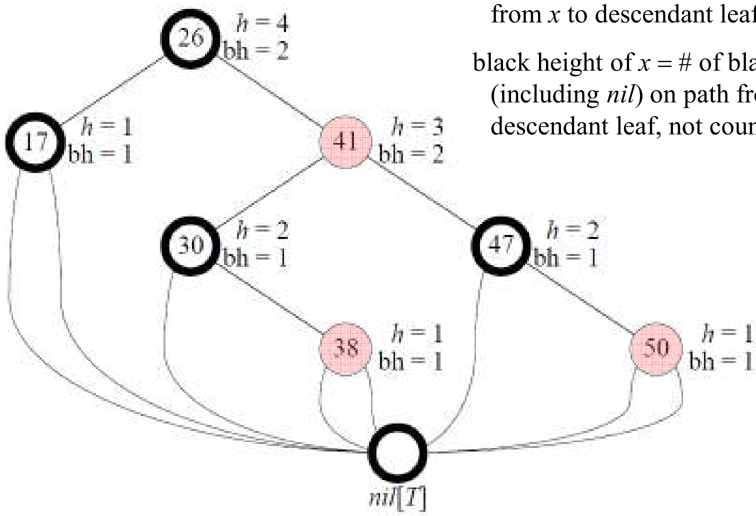
33

RB-Tree Properties

1. Every node is red or black.
2. The root is black.
3. Every leaf (*nil*[*T*]) is black.
4. If a node is red, then its parent is black.
5. All paths from a node x to descendant leaves contain the same number of black nodes, called the **black height** of x (exclude the color of x when counting).

34

Example



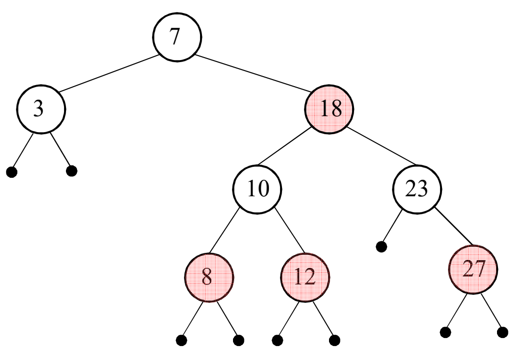
height of x = length of longest path from x to descendant leaf

black height of x = # of black nodes (including *nil*) on path from x to any descendant leaf, not counting x.

35

Exercise

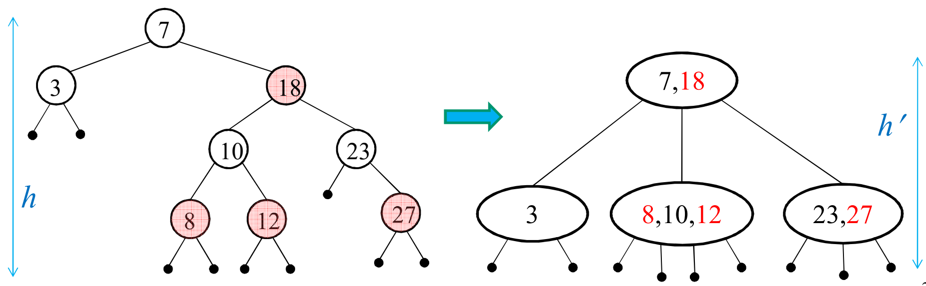
- Can you turn the following into a RB tree with keys ⟨3,7,8,10,12,18,23,27⟩



36

RB-Tree or 2-3-4 Tree?

- Merge each red node into black parent $\Rightarrow$ 2-3-4 Tree
  - Every internal node has 2 to 4 descendants
  - Every leaf has same depth
  - How many leaves for n keys?
- What is the depth of leaves in a 2-3-4 tree?
- What does height of 2-3-4 tree tells us about RB-tree?



37

Property Consequences

Claim. A node with height h has black height $\geq h/2$
Proof. At most $h/2$ red nodes $\Rightarrow$ at least $h/2$ black nodes

Claim. Subtree rooted at x contains $\geq 2^{\mathrm{bh}(x)} - 1$ keys
Proof (by induction). Induction on what? h, n, bh ?

Claim. A RB-tree with n keys has height $\leq 2\log(n+1)$
Proof. $n \geq 2^b - 1 \geq 2^{h/2} - 1$, where h=height, b=black height

Claim. In a RB-tree with n keys Min, Max, Pred, Succ, and Search run in $O(\log n)$ time.

38

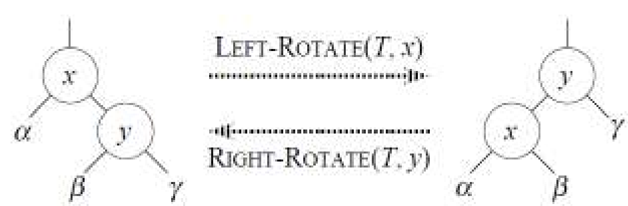
Tree Modifying Operations

- Can proceed as with regular BST, but…
- When inserting what color is the new node?
  - If red ⇒ may violate Property 4 (red node ⇒ black parent)
  - If black ⇒ may violate Property 5 (equal black height)
- When deleting, we remove one node
  - If this node is red we are ok
  - If node is black ⇒ may violate properties 2, 4, 5

39

Rotations

- Basic tree restructuring operation
- Preserves BST (order) property
- Accomplished by $O(1)$ pointer changes



- In both cases, in-order traversal yields

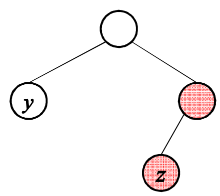
$$\langle\alpha\rangle\ x\ \langle\beta\rangle\ y\ \langle\gamma\rangle$$

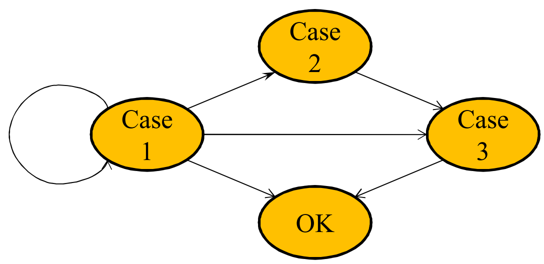
40

Insert

- Use standard BST insert algorithm
- Initially paint new node *z* red
- If parent of *z* is black we are done

  else, there are 3 cases, depending on relative position of *z* wrt *p*[*z*], and color of *y* (*z*'s uncle)

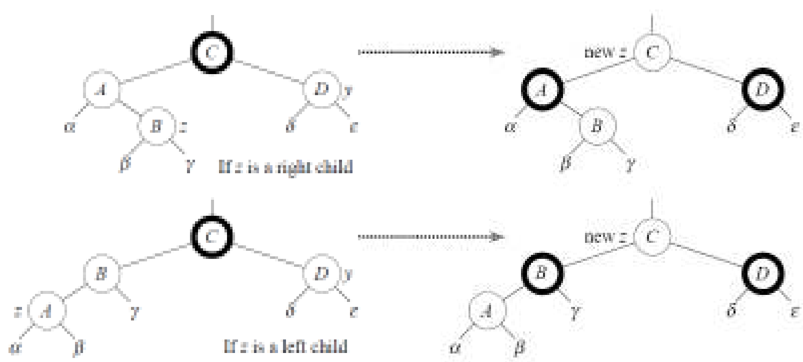




41

Case 1

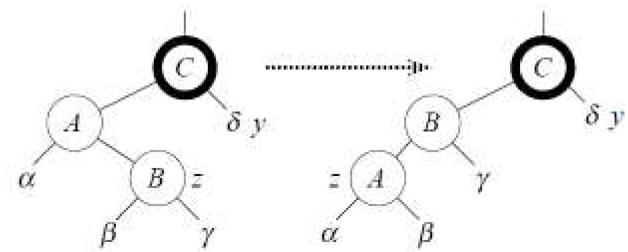
- *z*'s uncle (*y*) is red
- Relative position of *z* not relevant



42

Case 2

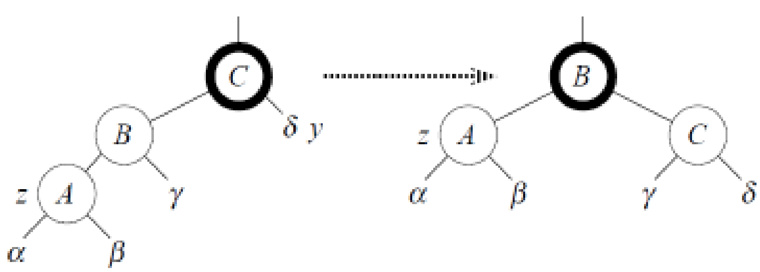
- *y* is black
- *z*'s parent and *z* are on opposite sides of their parents
- Convert to Case 3



43

Case 3

- *y* is black
- Both *z*'s parent and *z* are left or both right children
- Make *p*[*z*] black and *p*[*p*[*z*]] red, then rotate at *p*[*p*[*z*]]

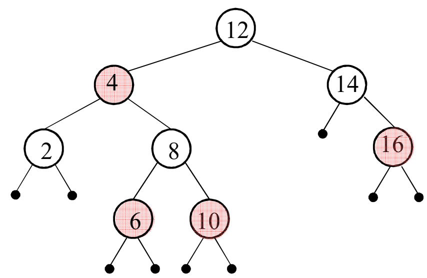


Case 3

44

Example

- Insert 5 into the following tree



45

Analysis

- Regular BST insert now takes only $O(\log n)$ time (why?)
- Each transition in state diagram takes $O(1)$ time and terminates or moves z two levels up
- $O(\log n)$ levels $\Rightarrow$ insert takes $O(\log n)$ time
- Each insert requires 0, 1, or 2 rotations

46

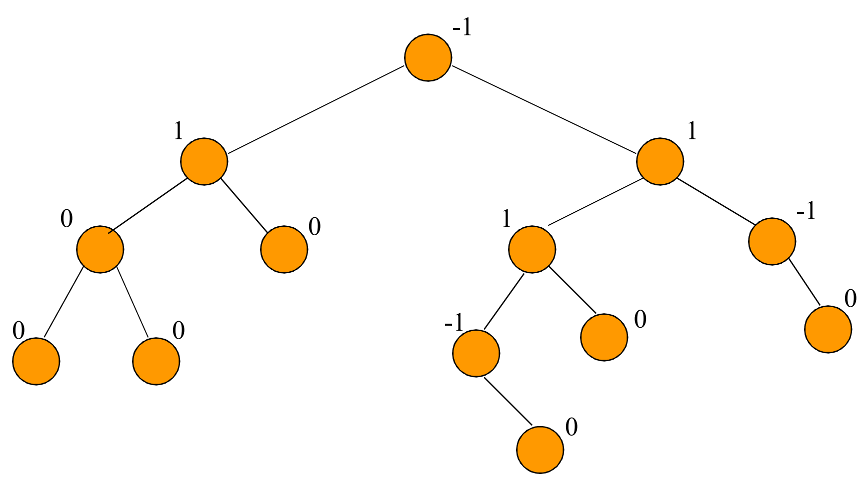
AVL Trees

- Type of "balanced" BST
- Named after Adelson-Velsky and Landis
- for each node, define its **balance factor** as:

  height of left subtree – height of right subtree
- balance factor of every node is –1, 0, or 1
  - Note: height of *nil* is –1

47

Example: Balance Factors



48

Insertion Algorithm

1. Insert node x using regular BST insert
2. Restructure the tree if necessary
   - Only nodes on the insertion path can have their balance factor altered
   - Walk up the path towards root and update b.f.'s
   - Stop at deepest node α (if any) whose balance factor is out of range
   - Correct imbalance at α via rotations (4 cases)
   - After "fixing" α there is no need to continue up

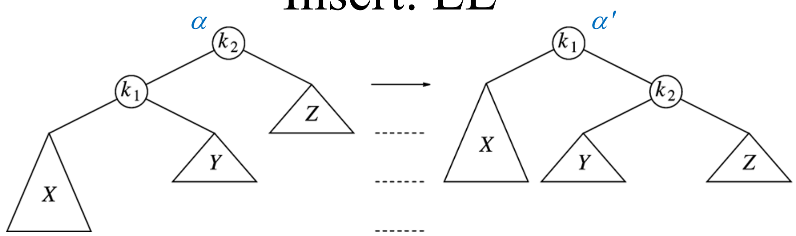
49

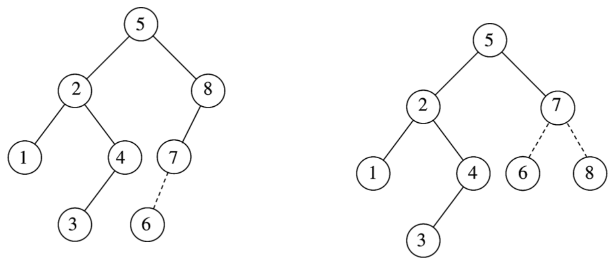
Rebalancing

- Let α be the node that needs rebalancing
- The 2 subtrees of α differ by 2 in height
- There are 4 cases, depending on where the insertion of the new value occurred:
  1. In the left subtree of the left child of α (**LL**)
  2. In the right subtree of the left child of α (**LR**)
  3. In the left subtree of the right child of α (**RL**)
  4. In the right subtree of the right child of α (**RR**)

50

Insert: LL



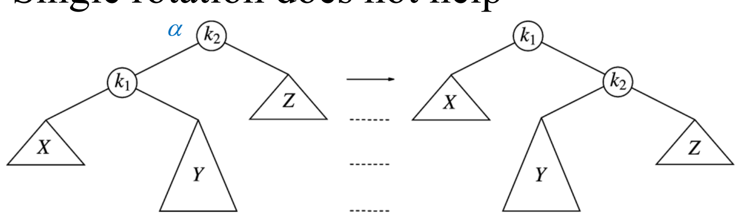


Exercise: Insert 3,2,1,4,5,6,7 into empty tree

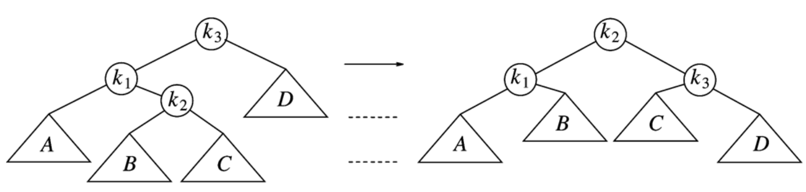
51

Insert: LR

- Single rotation does not help



- Can fix with 2 rotations



52

Height of AVL Tree

- $\lfloor \log n \rfloor \le h < 1.44 \log (n+1) + c$
- N_h = min # of nodes in AVL tree of height h

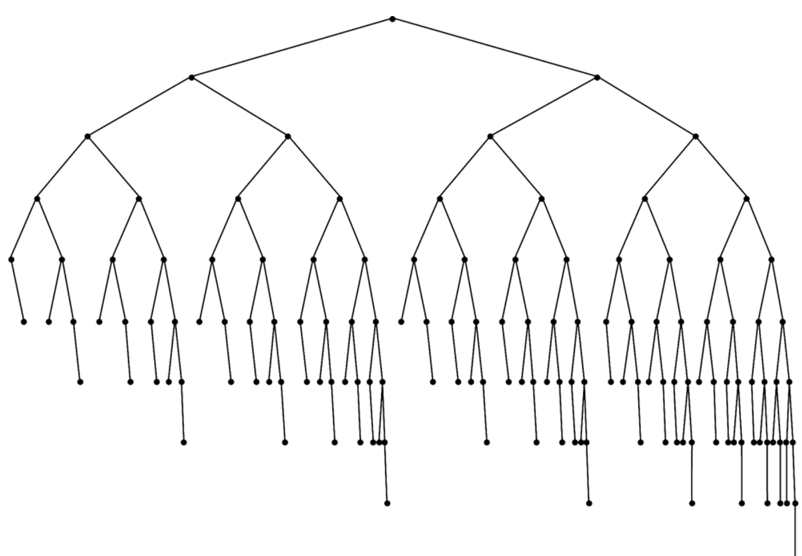
  $N_{-1} = 0, N_0 = 1, N_1 = 2, \ldots$

  $N_h = N_{h-1} + N_{h-2} + 1$

0, 1, 2, 4, 7, 12, 20, 33,…. 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,….

$$F_i = \frac{\varphi^i - (1-\varphi)^i}{\sqrt{5}}, \text{ where } \varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180339887...$$

Smallest AVL Tree of Height 9



54