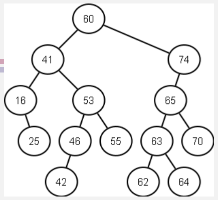
Binary Search Trees

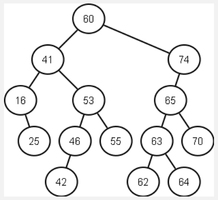


(dummy leaves not shown)

- find
  - moving down, 1-finger (only go to one child) pattern → loop
  - observation: if the element isn't there, search ends at a (dummy) leaf
- insert
  - can only insert at a leaf
  - the correct insertion point is the leaf where an unsuccessful search for the element ends up
- remove
  - can only remove above a leaf
  - if the element to remove does not have at least one leaf child, swap it with a safe element which does has at least one leaf child
    - i.e. the next element larger or smaller than the one to remove

Binary Search Trees

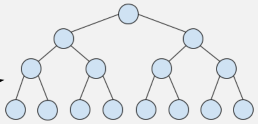
- visit all elements in order
  - moving down, both children pattern → recursion
  - need to visit smaller elements before the current node's element before the larger elements → inorder traversal

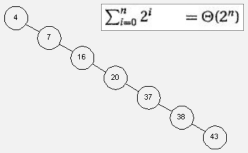


(dummy leaves not shown)

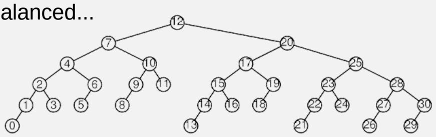
BST Height

- height of a binary search tree
  - best case is O(log n)
  - worst case is O(n)
- whether a BST of a given size is *balanced* (O(log n) height) or unbalanced (O(n) height) depends on the order of insertions and removals, not the elements in the tree
- can we do better?
  - try to keep the tree balanced...

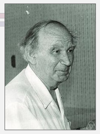




$$\sum_{i=0}^{n} 2^i \quad = \Theta(2^n)$$



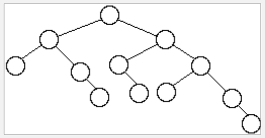
AVL Trees



- invented by Georgy Adelson-Velsky and Evgenii Landis in 1962
- first known balanced BST data structure

An AVL tree is a BST + a height balance property:

- for every node, the height of the node's left subtree is no more than one different from the height of the node's right subtree



The height balance property ensures that the height of an AVL tree with n nodes is O(log n).

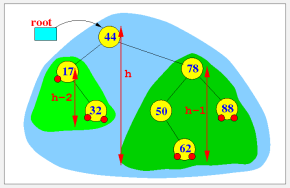
Height of AVL Trees

Let N(h) be the minimum number of nodes in an AVL tree of height h.

- a tree with the minimum number of nodes for its height is also the tallest possible for that number of nodes

Then

- N(h) = 1+N(h-1)+N(h-2)
  - one child must have height h-1 in order for the whole tree to have height h, and N(h-1) is the minimum number of nodes that subtree can have
  - the other child's height can be no more than one different, so it can't have height less than h-2, and N(h-2) is the minimum number of nodes that subtree can have
  - +1 for the root
- N(1) = 1, N(2) = 2
  - can't have fewer than one node per level of the tree



http://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Trees/AVL-height.html

Height of AVL Trees

- N(h) = 1+N(h-1)+N(h-2) ≤ 1+2N(h-1)

T(n) = a T(n-b) + f(n) where f(n) = $\Theta(n^c \log^d n)$

Cases are based on the number of subproblems and f(n).

a	f(n)	behavior	solution
> 1	any	base case dominates (too many leaves)	T(n) = $\Theta(a^{n/b})$
1	≥ 1	all levels are important	T(n) = Θ(n f(n))

N(h) = $O(2^h)$
→ h = log(N(h))

Operations on AVL Trees

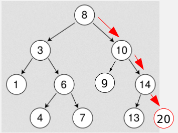
An AVL tree is a BST, so the find operation is no different.

For insert and remove:

- insert/remove as dictated by the (BST) structural and ordering rules
- fix up the broken balance property as needed

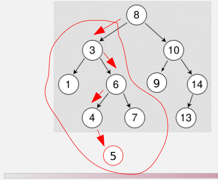
Insert

- structural property dictates that insertion only occurs at a node with fewer than 2 children
- ordering property dictates where



insert 20

no height-balance violations – we're done!

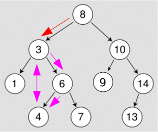


insert 5

height-balance property violated – uh oh!

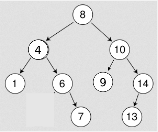
Remove

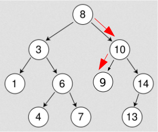
- structural property dictates that removal only occurs at a node with fewer than 2 children
  - may need to swap desired element with next larger/smaller in order to satisfy the structural property



remove 3

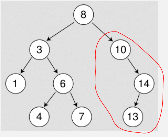
swap with 4 and remove
no height-balance violations –
we're done!





remove 9

height-balance property
violated – uh oh!

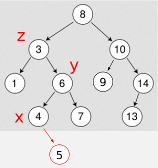


Restructuring

Both insertion and deletion may break the height balance property.

Restore it by performing one or more *restructuring operations* (or *rotations*).

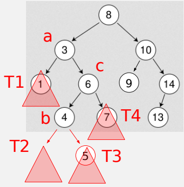
Restructuring



let *z* be the first unbalanced node (working up the tree from the point of insertion/deletion)

let *y* be z's tallest child

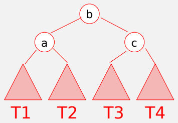
let *x* be y's tallest child



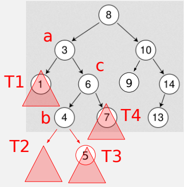
relabel x, y, z as a, b, c according to their correct sorted order

label the other subtree children of a, b, c as T1, T2, T3, T4 according to their correct sorted order

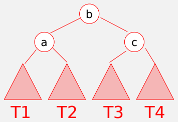
rearrange as shown:

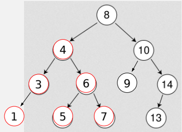


Restructuring



rearrange as shown:





height balance property restored!

Restructuring

How many restructuring operations are needed?

Observation.

- restructuring reduces the height of a subtree

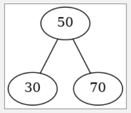
Insertion –

- insertion increases the height of a subtree, so one restructuring is sufficient to shorten it and restore balance

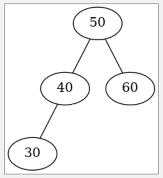
Removal –

- removal decreases the height of a subtree, so one restructuring may result in only pushing the imbalance higher up the tree
- O(log n) restructurings may be required

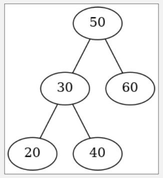
Warmup



insert 65



insert 20



insert 45

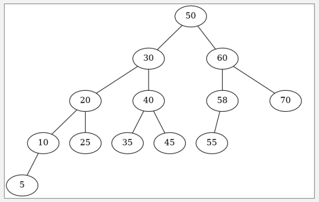
– insert/remove as normal for a BST, then fix the balance property if broken

– must check from the inserted/removed node back up to the root to find unbalanced nodes – what becomes unbalanced might not be directly above the new/removed node

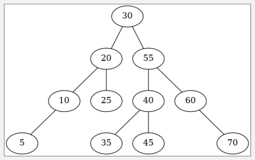
– at most one restructuring needed for insertion, but have to check the whole removed node to root path for removal (may need multiple restructurings)

Warmup

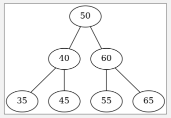
swap with successor



delete 70



delete 20

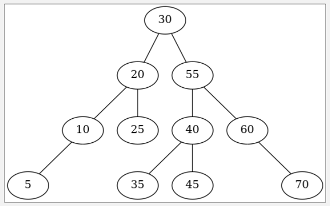


delete 50

– insert/remove as normal for a BST, then fix the balance property if broken

– must check from the inserted/removed node back up to the root to find unbalanced nodes – what becomes unbalanced might not be directly above the new/removed node

– at most one restructuring needed for insertion, but have to check the whole removed node to root path for removal (may need multiple restructurings)



can you rearrange leaves e.g. swap 25 and 35?

→ no, that would break the ordering property

does it matter if you swap with successor or predecessor?

→ no, but you should be consistent within a given implementation

→ yes, for problems that specify a particular swap, you should do that swap

Running Time

- initial BST insert/remove – O(log n)
- number of nodes to check for balance – O(log n)
- time to perform a balance check – O(1) if height info is stored for each node
- time to perform one restructuring – O(1)
- number of restructurings performed – 1 for insertion, O(log n) for removal
- time to update stored balance information – O(log n) nodes affected, O(1) per

Total time: O(log n) for insert/remove

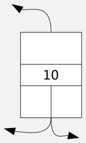
Implementation

- start with an implementation of a binary tree

ADT	characterized by	typical operations
BinaryTree	hierarchical ordering	size(), isEmpty() structural accessors structural mutators: expand leaf, remove above leaf manipulate elements: set, swap

Implementing BinaryTree – TreeNode

operation	linked structure
instance variables	• element, parent, left child, right child
getElement()	O(1) – return element



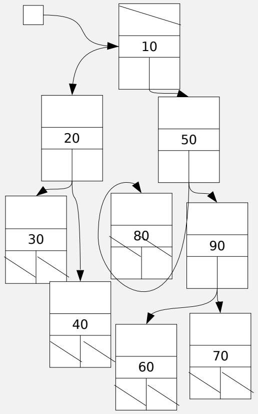
Implementing BinaryTree

(parent pointers not shown)

operation	linked structure
instance variables	• root, size
size()	Th(1) – return size
isEmpty()	Th(1) – return size == 0

getParent(node)
getLeftChild(node)
getRightChild(node)
	Th(1) – return value of instance variable in the node

expandLeaf(node)	Th(1) – create two new nodes, update links, size += 2
removeAboveLeaf(node)	Th(1) – relink grandparent to sibling, size -= 2
setElement(node,elt)	Th(1) – change instance var in node
swapElements(node1, node2)	Th(1) – essentially 2 setElements


10
20
50
30
80
90
40
60
70