AUBURN

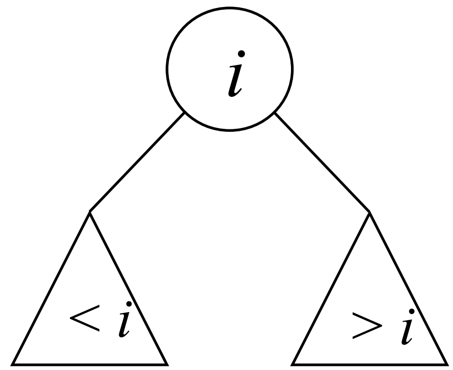
UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Binary Search Trees

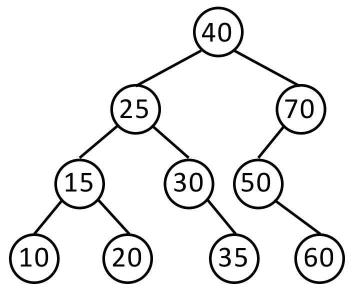
Binary search trees

A binary search tree is a **binary tree** in which the **search property** holds on *every* node.

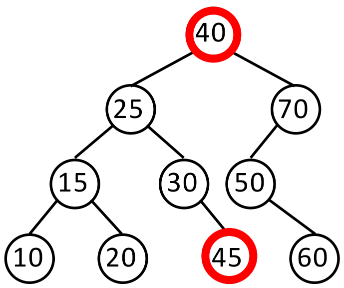


Binary search trees

The search property must hold on every node in the tree.



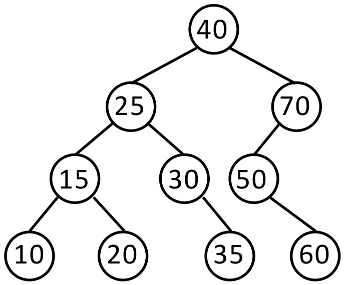
A binary search tree



NOT a binary search tree!

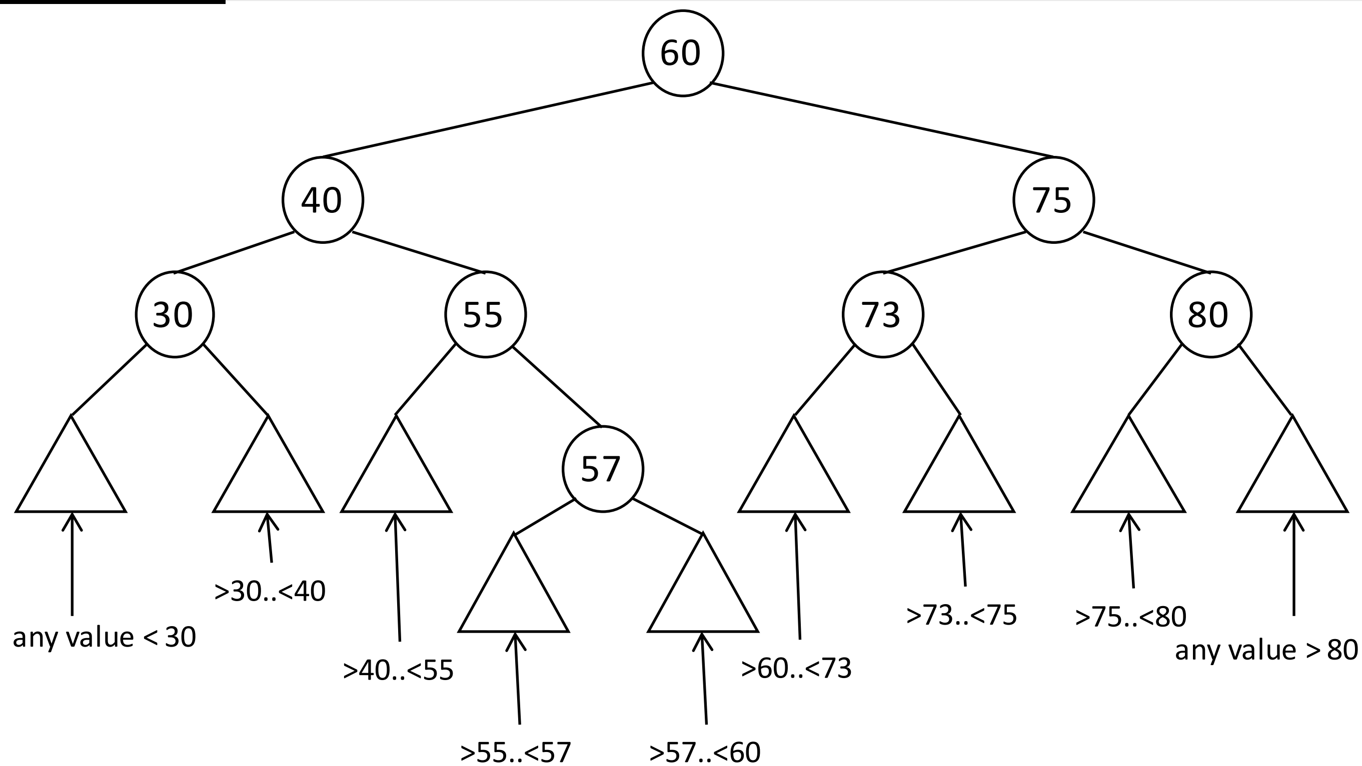
Binary search trees

A binary search tree imposes a **total order** on all its elements.



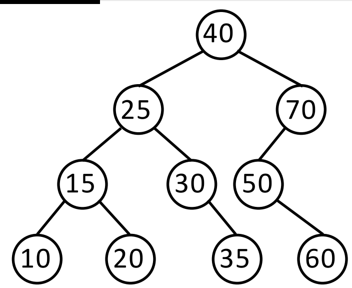
An inorder traversal: 10, 15, 20, 25, 30, 35, 40, 50, 60, 70

Where can values go?



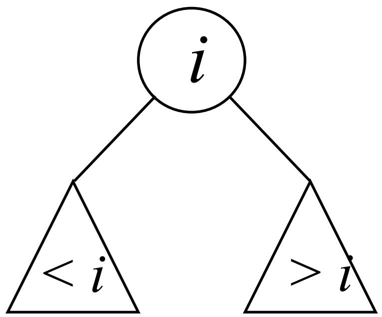
Searching for values

Searching for values



Begin at the root.

Use the search property (total order) of the nodes to guide the search downward in the tree.



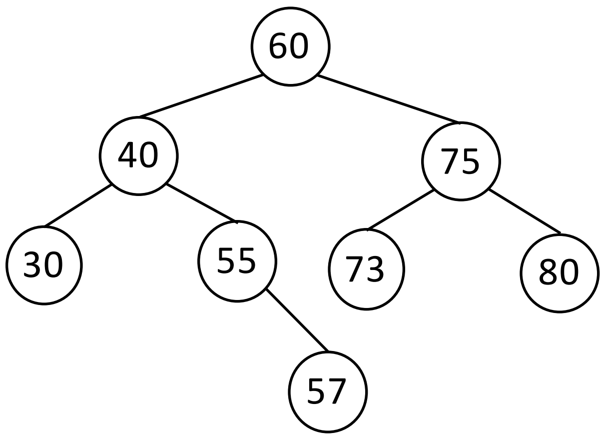
Recursive

```
boolean search(n, target) {
   if (n == null)
      return false
   else {
      if (n.element == target)
         return true
      else if (n.element > target)
         return search(n.left, target)
      else
         return search(n.right, target)
   }
}
```

Iterative

```
boolean search(n, target) {
   found = false
   while (n != null) && (!found) {
      if (n.element == target)
         found = true
      else if (n.element > target)
         n = n.left
      else
         n = n.right
   }
   return found
}
```

Searching for values

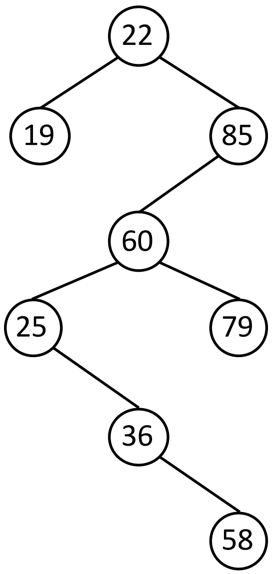


target	Number of comparisons
60	1
80	3
57	4
73	3
59	4

The number of comparisons to find a given value is equal to the depth of the node that contains it.

Worst Case: Searching for the value in the lowest leaf, in which case the entire **height of the tree** is traversed. (Or searching for a value not in the tree but < or > lowest leaf value.)

Searching for values



target	Number of comparisons
22	1
58	6
25	4
80	4
55	6

Searching a binary search tree is **O(height)**

~N *tall and narrow*

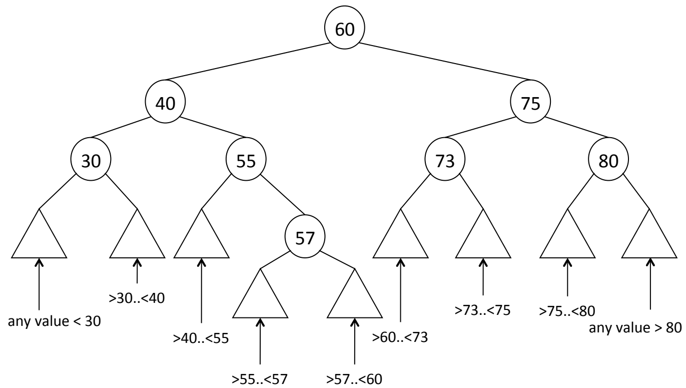
~log N *short and wide*

Adding values

Adding values

Use the search algorithm to locate the physical insertion point.

Exactly one!



New node will always be a new leaf.

Worst case: Inserting a new node as a child of the currently lowest leaf.

O(height)

Adding values

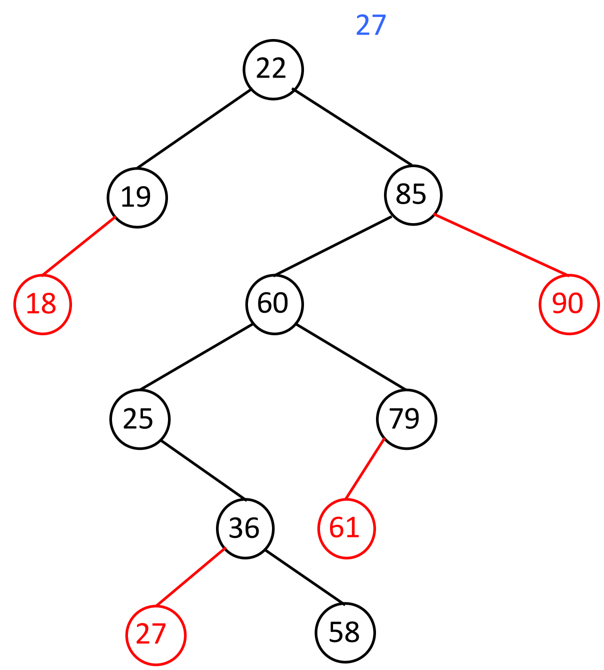
Add the following values:

27

18

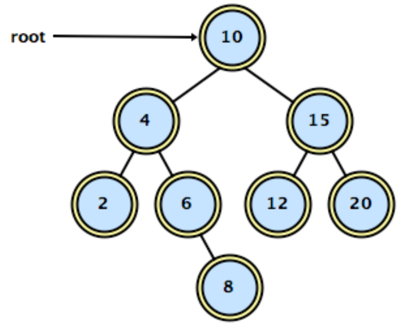
90

61



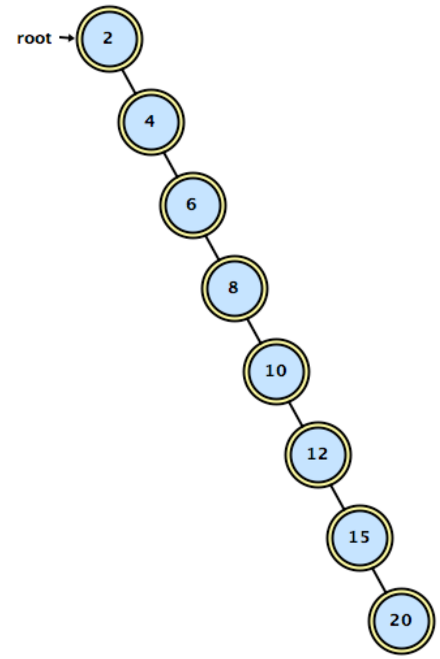
Order of insertion and height

Insert: 10, 4, 2, 15, 12, 6, 8, 20



height is ~log N

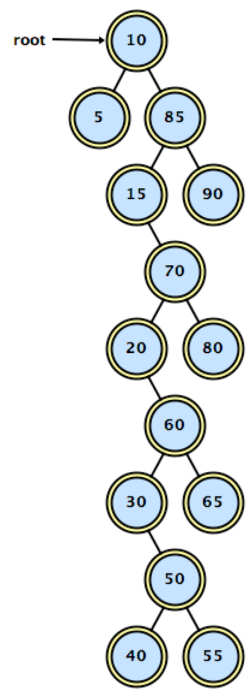
Insert: 2, 4, 6, 8, 10, 12, 15, 20



height is N

Self-check exercise

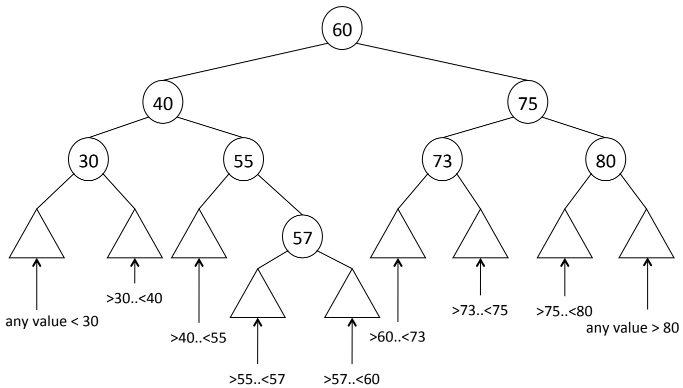
Insert: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



Removing values

Removing values

Use the search algorithm to locate the value to delete.



Node to delete could be anywhere, not just a leaf.

The number of children that the node has determines how the value gets deleted from the tree.

(Hibbard deletion)

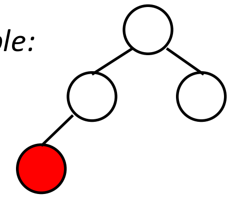
A worst case: Deleting the currently lowest leaf.

O(height)

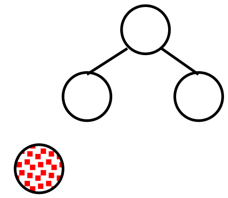
Removing values

Case 0: The value to delete is in a **leaf node**.

Example:

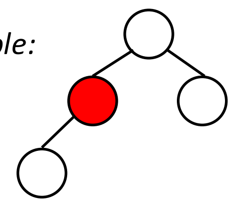


Set the parent's pointer to this node to null.

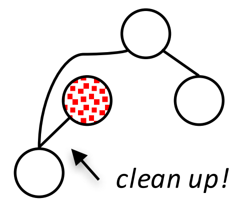


Case 1: The value to delete is in a node with exactly **one non-empty subtree**.

Example:

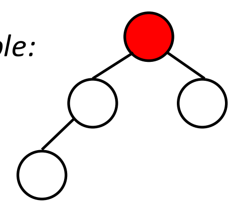


Set the parent's pointer to this node to this node's child.



Case 2: The value to delete is in a node with **two non-empty subtrees**.

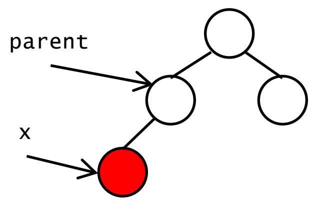
Example:



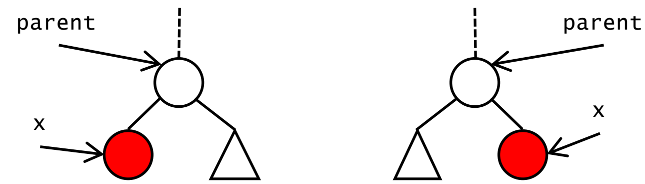
*Don't delete this node! Find a **replacement** for this node's value and delete the node containing the replacement.*

Removing a value with zero children

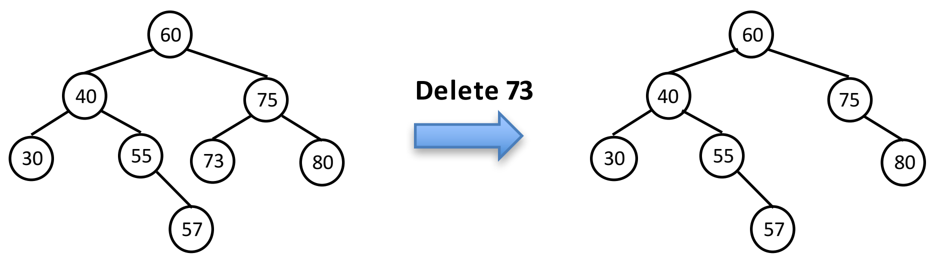
Delete x by setting to null the parent node's reference to x.



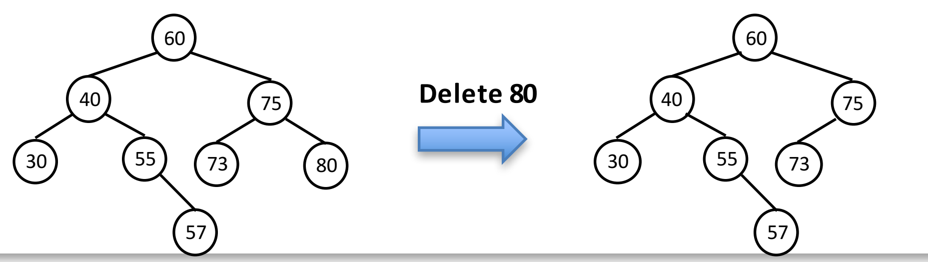
Structural possibilities:



Example:

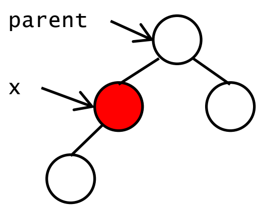


Example:

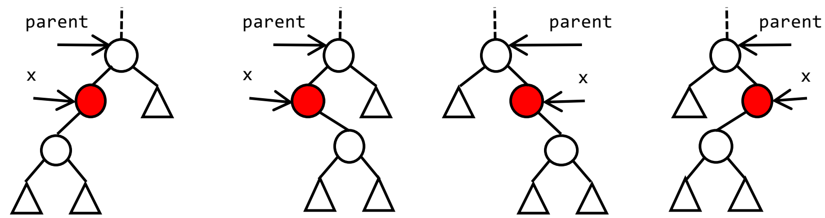


Removing a value with one child

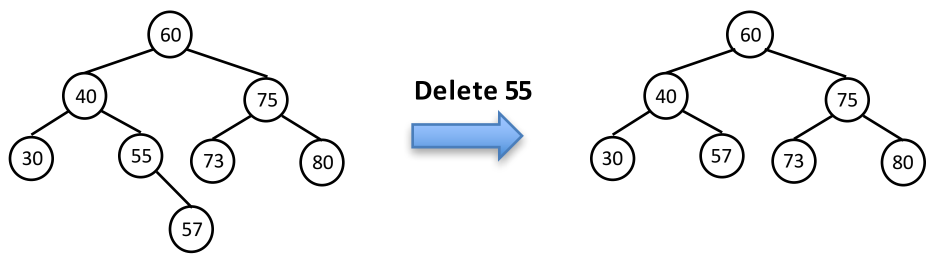
Delete x by replacing the parent node's reference to x with a reference to x's child.



Structural possibilities:



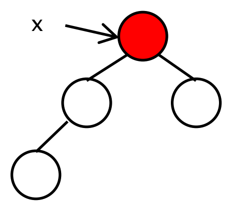
Example:



Example:

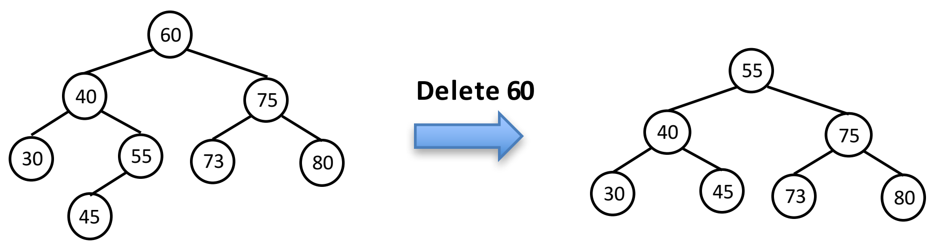
Removing a value with two children

Replace the value in x, then delete the node that contained this replacement value.



Replacement values:

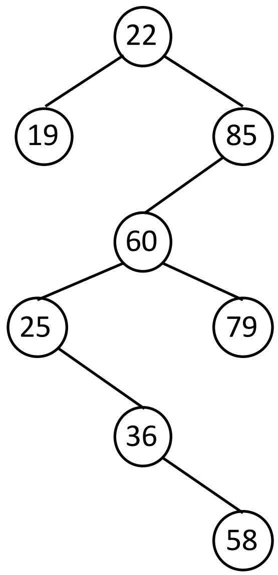
Example:



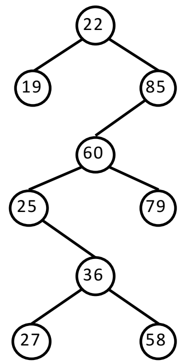
Example:

Example add and remove sequence

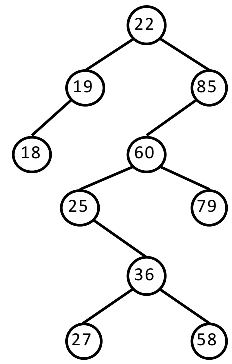
Initial tree:



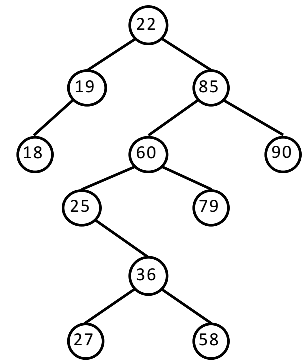
Insert 27



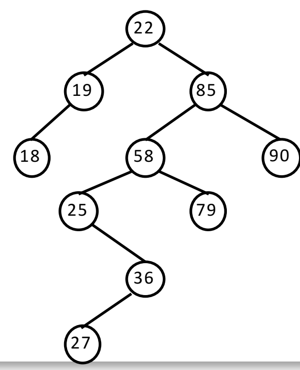
Insert 18



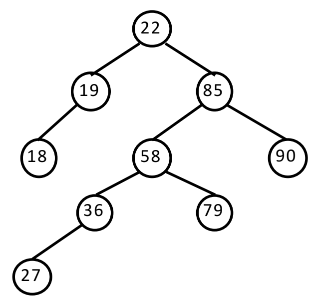
Insert 90



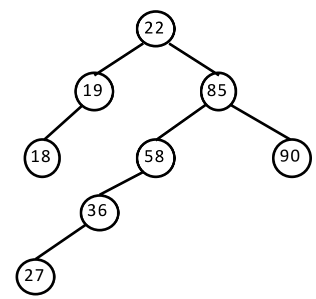
Delete 60 (Case 2)



Delete 25 (Case 1)



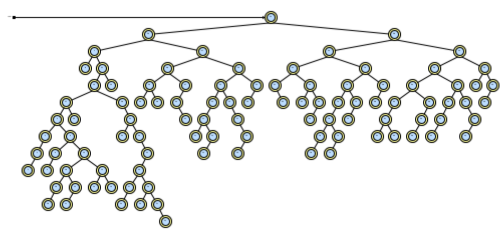
Delete 79 (Case 0)



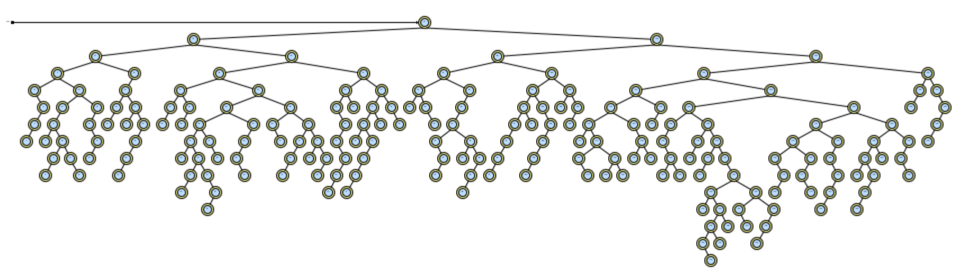
Balance

Random adds

If values are added in random order, the tree should stay relatively flat.



N = 100
max height = 13
average height = 9
optimal height = 7

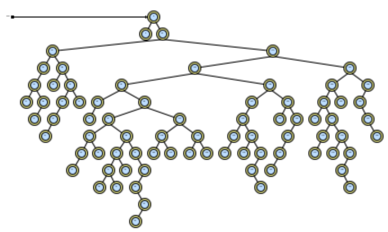


N = 200
max height = 15
average height = 10
optimal height = 8

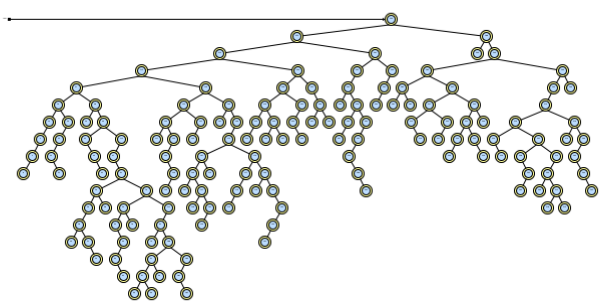
Worst-case height is, of course, N but “average” or expected height is much better.

Random removes

If values are removed in random order, the tree doesn't stay as well-structured.



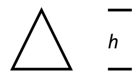
N = 100
max height = 15
average height = 13
optimal height = 7



N = 200
max height = 20
average height = 18
optimal height = 8

Shapes and height

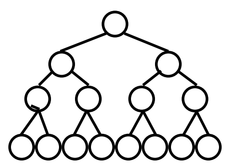
height



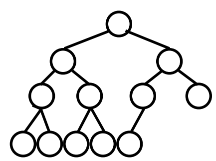
Many tree algorithms are dependent to some extent on the tree's height.

best-case BST

full

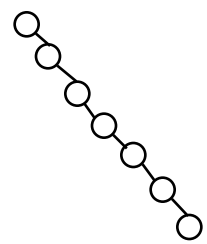


complete



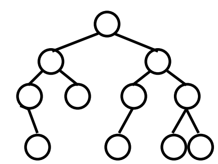
$$h(t) = \lfloor \log_2 n \rfloor + 1$$

worst-case BST



$$h(t) = n$$

balanced BST



$$h(t) = O(\log n)$$

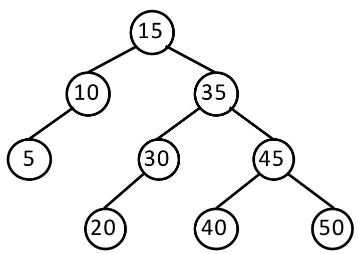
Self-balancing search trees

There are many different self-balancing search trees.

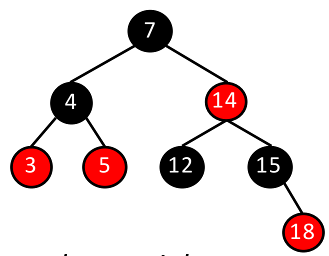
All SBSTs guarantee that the tree's height is O(log N) in the worst case, and that searching, inserting, and deleting have worst case time complexity O(log N).

We will discuss:

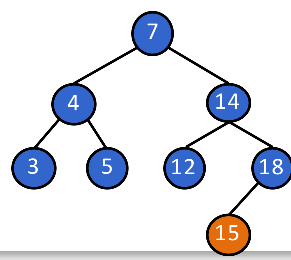
AVL Trees



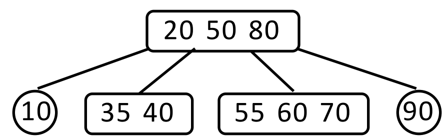
Red-Black Trees



and a special type …



2-4 Trees



and a generalization …