Binary Search Trees

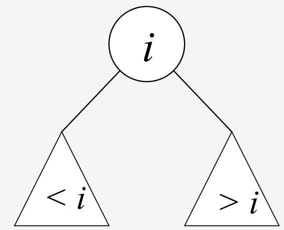
COMP 2210 – Dr. Hendrix

AUBURN UNIVERSITY

SAMUEL GINN COLLEGE OF ENGINEERING

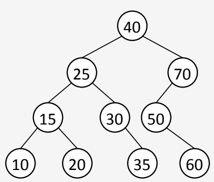
Binary Search Trees

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

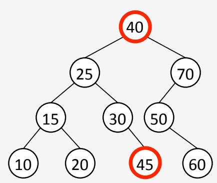


Every node ...

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



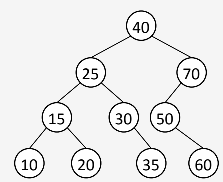
A binary search tree



NOT a binary search tree!

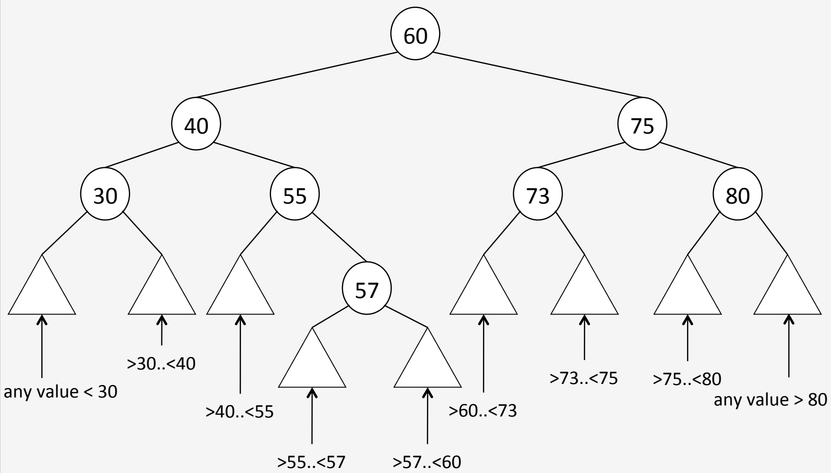
Total Order

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

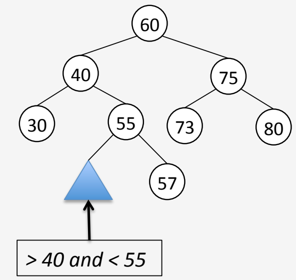
Where can values go?



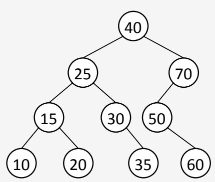
Participation question

Q. In the binary search tree below, what values could go in the subtree indicated by the triangle?

A. < 55
B. > 40 and < 55
C. > 55 and < 57
D. > 55 and < 60
E. > 30 and < 55

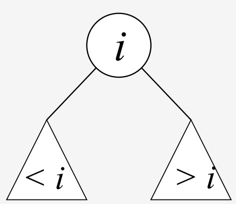


Searching for values



Begin at the root.

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



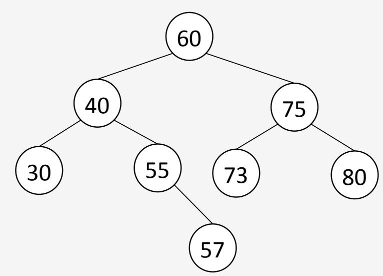
Recursive

```
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

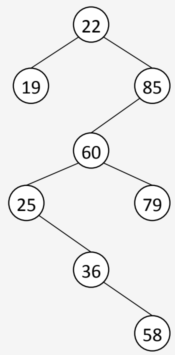


target	Number of comparisons
60	1
80	3
57	4
73	3
59	4

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

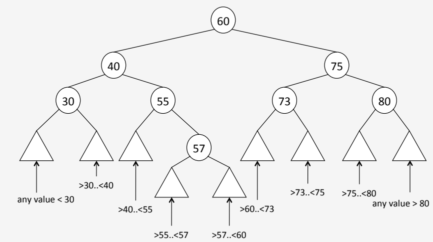
Searching a binary search tree is **O(height)**

~N *tall and narrow*

~log N *short and wide*

Inserting 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)

Insertion example

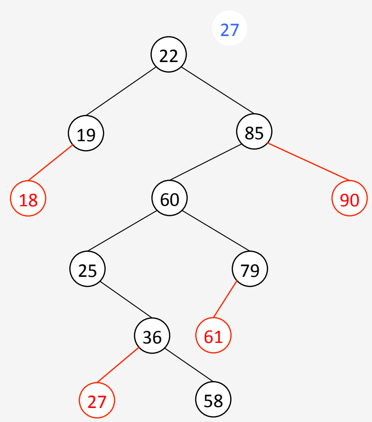
Insert the following values:

27

18

90

61



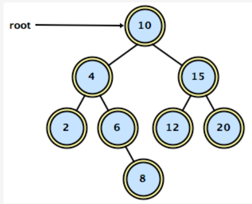
Same values, different heights

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

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

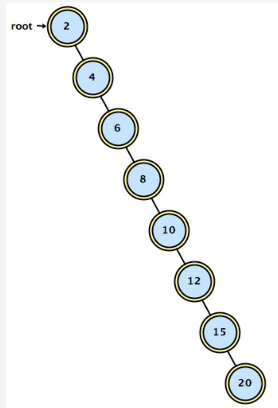
Same values, different heights

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



height is ~log N

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



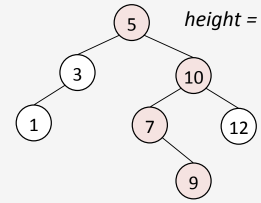
height is N

Participation question

Q. What is the height of the binary search tree that results from inserting the following values in the order in which they are written?

```
5, 3, 10, 7, 1, 9, 12
```

A. 3
B. 4
C. 5
D. 6
E. 7



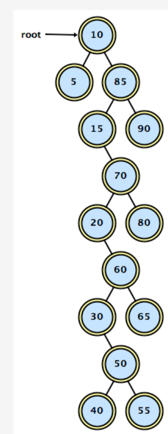
height = 4

Self-check exercise

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

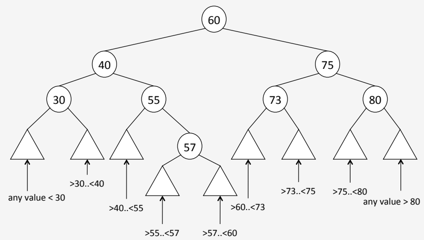
Self-check exercise

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



Deleting 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)

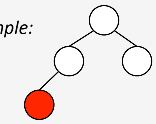
A worst case: Deleting the currently lowest leaf.

O(height)

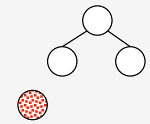
Three cases for deletion

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

Example:

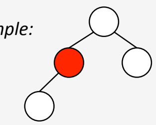


Set the parent's pointer to this node to null.

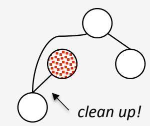


Case 1: The value to delete is in a node with exactly **one non-empty subtree**.

Example:

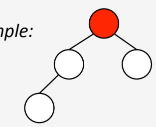


Set the parent's pointer to this node to this node's child.



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

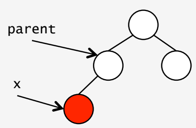
Example:



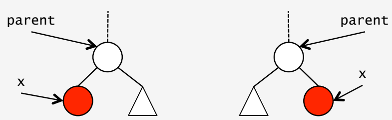
Don't delete this node! Find a ***replacement*** *for this node's value and delete the node containing the replacement.*

Case 0: a leaf

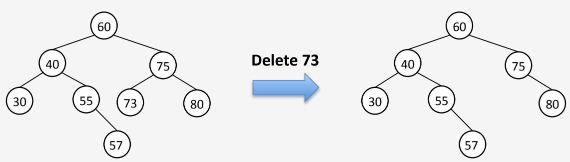
Delete x by setting to null the parent node's reference to x.



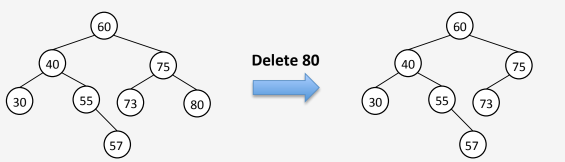
Structural possibilities:



Example:

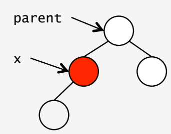


Example:

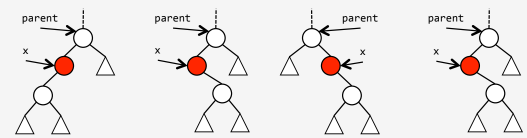


Case 1: one child

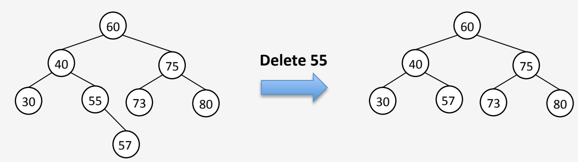
Delete x by replacing the parent node's reference to x with a reference to x's child.



Structural possibilities:



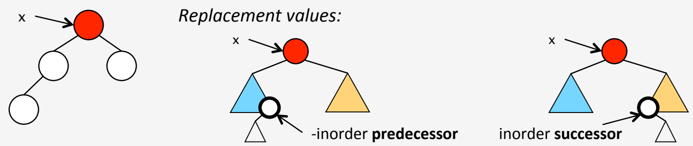
Example:



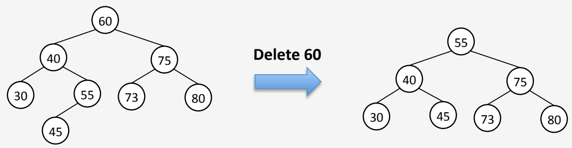
Example:

Case 2: two children

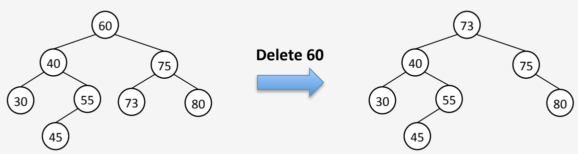
Replace the value in x, then delete the node that contained this replacement value.



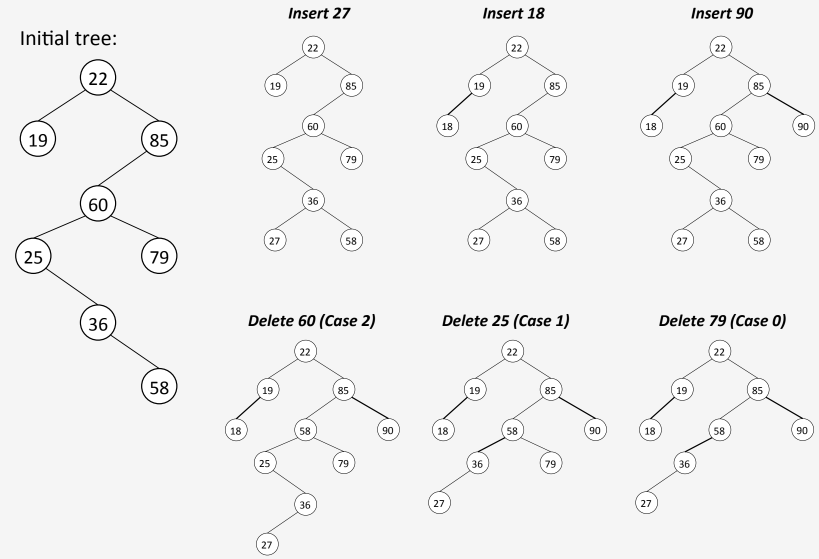
Example:



Example:

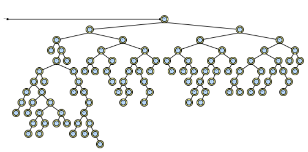


Example insertions and deletions

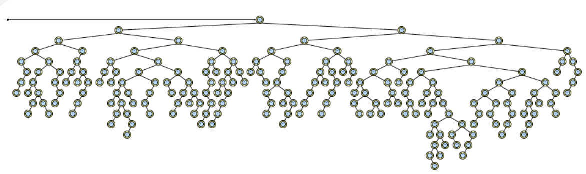


Random adds

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



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



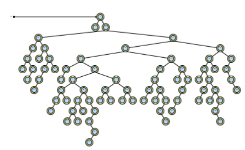
N = 200
max height = 15
average height = 10
optimal height = 8

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

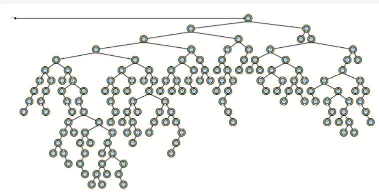
Fun for COMP 3270: Expected number of compares to add = ~1.39 log N

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

Even more fun for COMP 3270: Expected number of compares for remove = ~√N

This now changes search and add to ~√N.

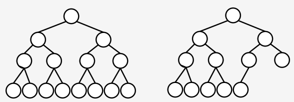
Shapes and height

height 

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

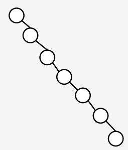
best-case BST

full complete



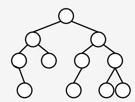
$h(t) = \lfloor \log_2 n \rfloor + 1$

worst-case BST



$h(t) = n$

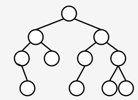
balanced BST



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

Balance

There are different definitions of balance, but they all meet the spirit of the following ideas:



A tree is balanced if a given leaf is *not much farther* away from the root than any other leaf in the tree.

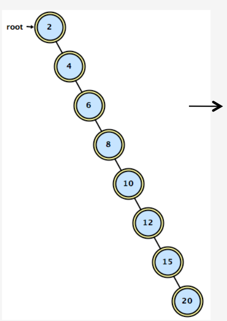
A tree is balanced if for any given node, the left and right subtrees of that node have *similar* heights.

Trees can be balanced **periodically** or **incrementally**.

Periodic balancing: The tree grows naturally, without regard for its height or shape. At some interval (either when a height threshold is reached or just on-demand), the tree is reorganized to reduce its height to O(log N). The cost of this operation is amortized over many inserts and deletes.

Incremental balancing: The insert and delete algorithms are modified to ensure that the tree is balanced after each operation that could affect its height. Trees that use incremental balancing are called **self-balancing** BSTs.

Simple periodic balancing

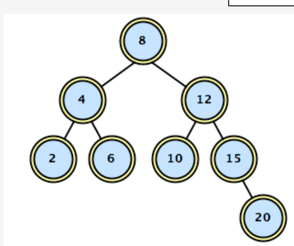


Use an inorder traversal to create an array of the tree elements.

2	4	6	8	10	12	15	20
0	1	2	3	4	5	6	7

Use a modified binary search to create the balanced tree.

```
toBST(a, left, right) {
   if (left > right)
      return null;
   else {
      mid = (left + right) / 2;
      n = new BTN(a[mid]);
      n.left = toBST(a, left, mid-1);
      n.right = toBST(a, mid+1, right);
      return n;
```



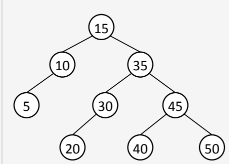
Self-balancing search trees

There are many different self-balancing search trees.

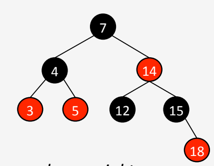
All SBSTs guarantee that the tree's height is O(log N) in the worst case, and that searching, inserting, and deleting have worst case time complexity O(log N).

We will discuss:

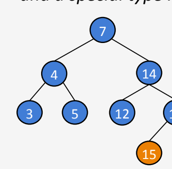
AVL Trees



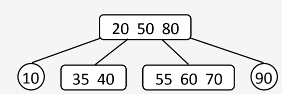
Red-Black Trees



and a special type …



2-4 Trees



and a generalization …