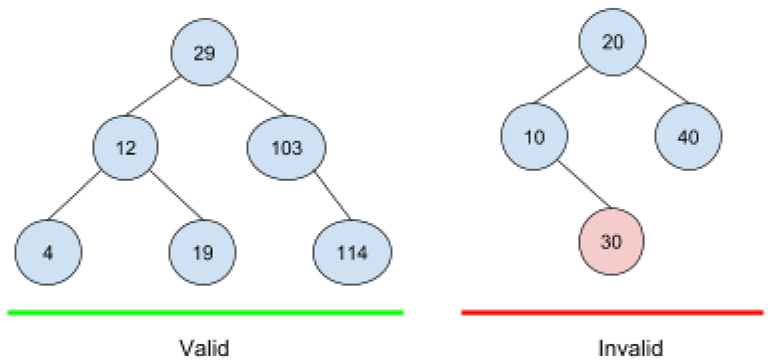
CS 1332 - Week 04

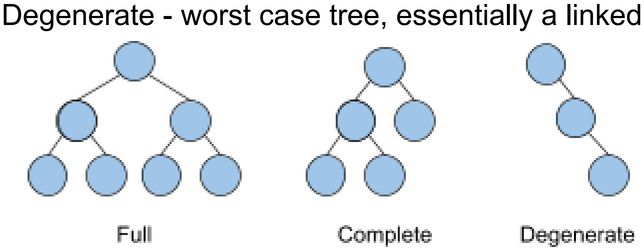
Monday

Topics

I. Binary Search Trees
   A. Binary tree review
      1. Binary trees nodes can have 0, 1 or 2 children
      2. Children are in positions defined as left and right children
      3. The root is the entry point for the tree (like the head of a linked list)
   B. Binary search trees are binary trees subject to an *order property*:
      1. All data in a node's left subtree are less than the data in the node and all data in a node's right subtree are greater than the data in the node
      2. data in left subtree < current node < data in right subtree

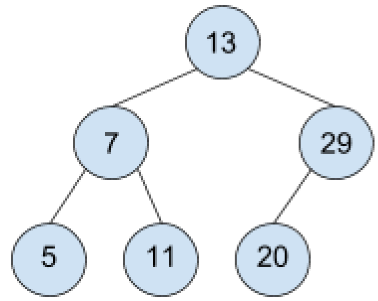


      3. Because of the order property, BSTs can only hold comparable data
         a) The data type must implement the Comparable interface with requires the implementation of the .compareTo() method
         b) If a.compareTo(b) < 0 → a < b
         c) If a.compareTo(b) > 0 → a > b
      4. Advantages of a BST:
         a) Searching is optimized: traverse fewer nodes to find data, every time you do a comparison, you can eliminate about half the data that you don't need to look at (like the binary search algorithm)
   C. Various shape properties of binary trees:
      1. Full - all nodes, except for the leaves, have 2 children, all searches will be guaranteed to be O(logn)
      2. Complete - leaves are filled level by level, left to right with no gaps
      3. Degenerate - worst case tree, essentially a linked list, O(n) operations



D. BST Traversals

1. Depth - follow one path as far as you can go before taking another path

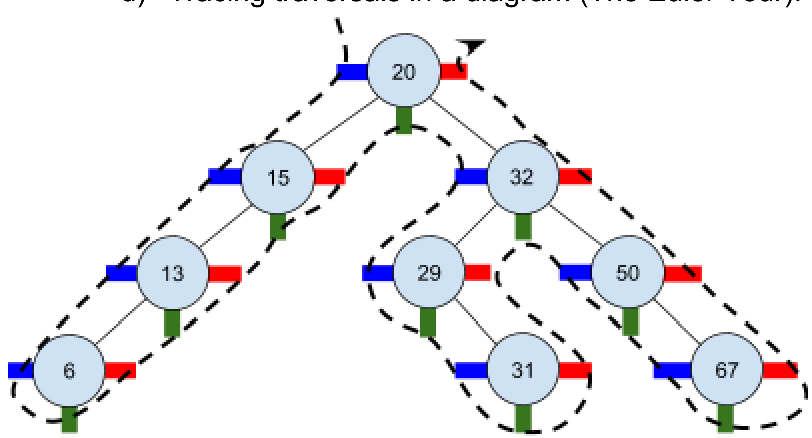


a) Pre-order traversal (of tree above): 13, 7, 5, 11, 29, 20
   (1) If current node is null → return
   (2) Else →
      (a) look at data (record it, print it, etc.)
      (b) recurse left
      (c) recurse right

b) Post-order traversal: 5, 11, 7, 20, 29, 13
   (1) If current node is null → return
   (2) Else →
      (a) recurse left
      (b) recurse right
      (c) look at data

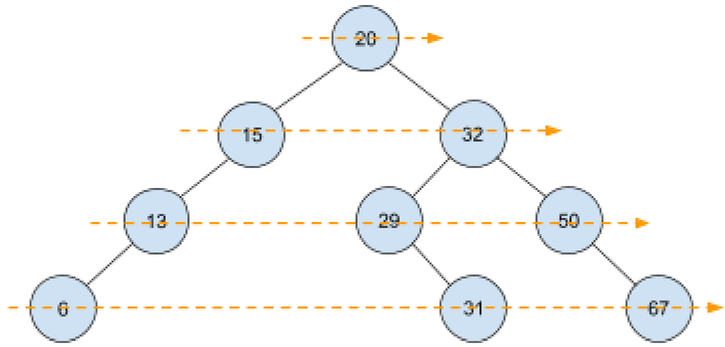
c) In-order traversal: 5, 7, 11, 13, 20, 29
   (1) If current node is null → return
   (2) Else →
      (a) recurse left
      (b) look at data
      (c) recurse right

d) Tracing traversals in a diagram (The Euler Tour):



Pre-order: 20, 15, 13, 6, 32, 29, 31, 50, 67
Post-order: 6, 13, 15, 31, 29, 67, 50, 32, 20
In-order: 6, 13, 15, 20, 29, 31, 32, 50, 67

2. Breadth - goes one level at a time
    a) Level-order - not recursive, uses a queue and a while-loop
        (1) Add the root to the queue
        (2) While the queue is not empty →
            (a) Remove one node from the queue
            (b) Enqueue its left and right children (in that order)



Level-order: 20, 15, 32, 13, 29, 50, 6, 31, 67

Wednesday

<u>Topics</u>

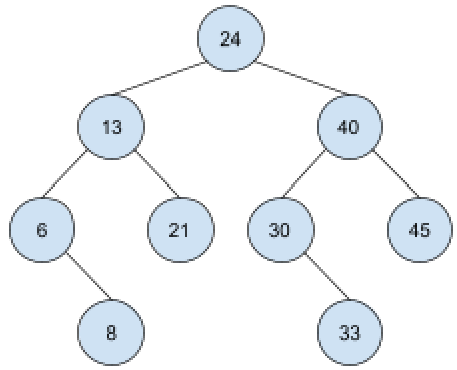
I. BST Operations
   A. Adding
      1. Start at the root → act like you are “searching” for the data
         a) If you reach a null node → add the data here
         b) If you reach a match → do nothing (we don’t want duplicates)
      2. Example pseudocode:

<table>
<tr><th>Look-Ahead Technique (don’t do this...)</th><th>Pointer Reinforcement Technique (do this!)</th></tr>
<tr><td><pre>
void add(data) // public method
  if root == null
    root = new Node(data)
  else add(data, root)

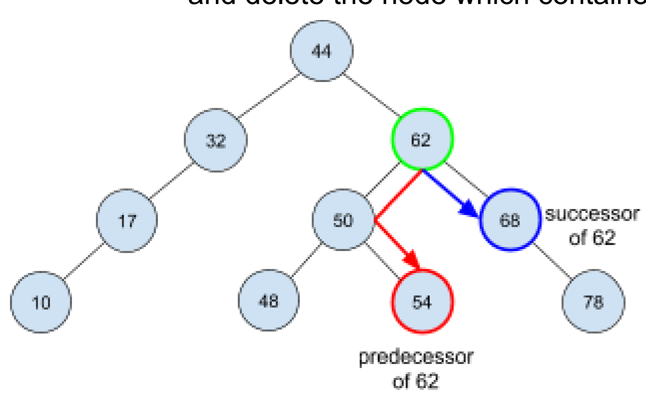
void add(data, node) // private
  if (data < node’s data)
    if node.left == null
      node.left = new Node(data)
    else add(data, node.left)
  else if (data > node’s data)
    if node.right == null
      node.right = new Node(data)
    else add(data, node.right)
  else // data == node’s data
    return
</pre></td><td><pre>
void add(data) // public method
  root = add(data, node)

Node add(data, node) // private
  if (node == null)
    return new Node(data)
  else if (data < node’s data)
    node.left = add(data,node.left)
  else if (data > node’s data)
    node.right = add(data, node.r)
  return node
</pre></td></tr>
</table>

3. Tracing an example: add 24, 13, 6, 21, 40, 30, 8, 45, 33



B. Removing
   1. Cases, removing a node with:
      a) no children - easiest
      b) one child - easy
      c) two children - hardest
   2. No child remove case (eg. remove 45 from the tree above)
      a) Set the parent's pointer to the node to null
   3. One child remove case (eg. remove 6 from the tree above)
      a) Set the parent's pointer to the node to the node's child
   4. Two children remove case (eg. remove 24 from the tree above)
      a) We can't easily move pointers to it
      b) Instead of removing the node itself, we'll replace the node's data with the predecessor or successor and remove THAT node:
         (1) Predecessor - largest value still smaller than the current value; go one to the left, then as far right as possible
         (2) Successor - smallest value still larger that the current value; go one to the right, then as far left as possible
      c) We'll delegate the removal of an actual node to the node containing the predecessor or successor since these will always be 0 or 1 child remove cases
         (1) To remove 62, we'd replace its data with the pred. or succ. and delete the node which contained the pred. or succ.



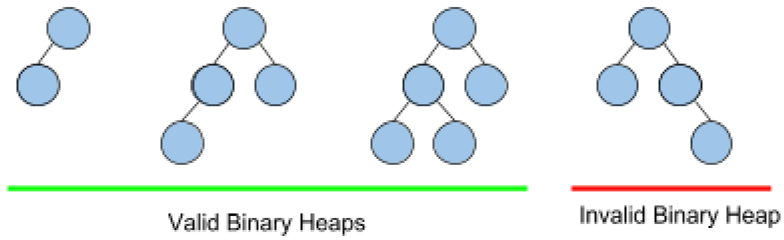
Friday

<u>Topics</u>

I. Binary Heaps

A. A heap is a binary tree but NOT a binary search tree

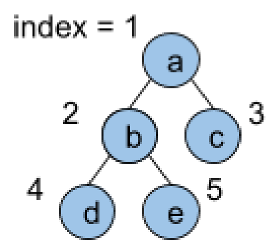
1. General shape property: nodes can have up to two children
2. Constrained shape property: must be complete (harder to enforce)



B. Implementing a binary heap

1. Since they are complete trees, they can be implemented with Arrays:
   a) For a node at index n:
      (1) its left child is at index n * 2
      (2) its right child is at index (n * 2) + 1
      (3) its parent is at index n / 2 (Java truncates decimals)

As a tree →

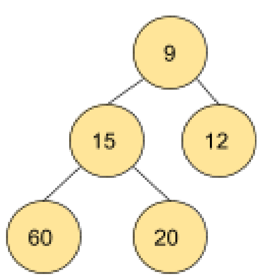


As an array →

0	1	2	3	4	5
	a	b	c	d	e

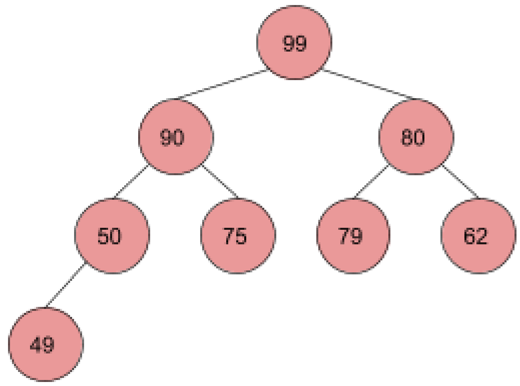
C. Min-Heap order property:

1. Minimum heaps (min-heaps) keep the *smallest* data in the set at the root
2. Children are always *greater* than the parent (there is not relationship between the siblings, i.e. the larger of the two doesn't need to be anywhere special)
3. The last element is at index size



0	1	2	3	4	5
	9	15	12	60	20

D. Max-Heap order property:
    1. Maximum heaps (max-heaps) keep the *largest* data in the set at the root
    2. Children are always *less* than the parent (also no sibling relationship)
    3. The last element is at index size



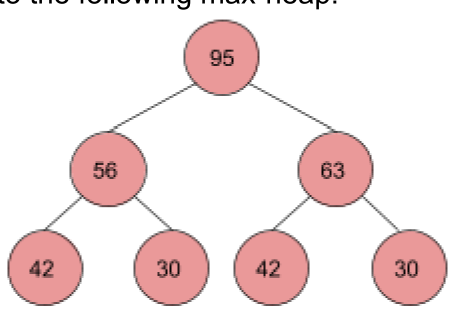
0	1	2	3	4	5	6	7	8
	99	90	80	50	75	79	62	49

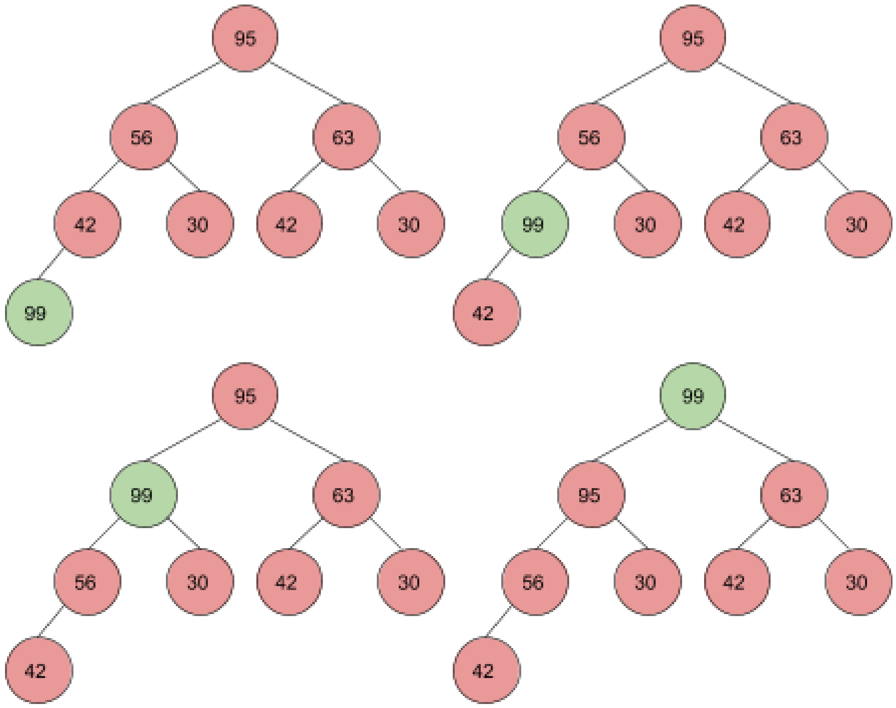
E. Pros:
    1. Used to back Priority Queues → return the most urgent/important data
        a) Eg. emergency room waiting room, those with more urgent emergencies are pushed to the front of the line and tended to first

F. Cons:
    1. If you want to search for an element, you'd have to search through the entire backing structure (heaps are not like binary search trees)

II. Binary Heap Operations

A. Adding:
    1. Add to the end of the array (this keeps the shape in place)
    2. Heapify-up/up-heap (this restores the order)
        a) Compare the new data with its parent (index / 2)
        b) If the new data does not have the correct relationship with its parent, swap them
        c) Repeat this process until the new data is at the root or the new data has the correct relationship with its parent.
    3. Eg. Add 99 to the following max-heap:




95
56
63
42
30
42
30
99
95
56
63
99
30
42
30
42
95
99
63
56
30
42
30
42
99
95
63
56
30
42
30
42