CS 106B, Lecture 20
Advanced Binary Trees

Plan for Today

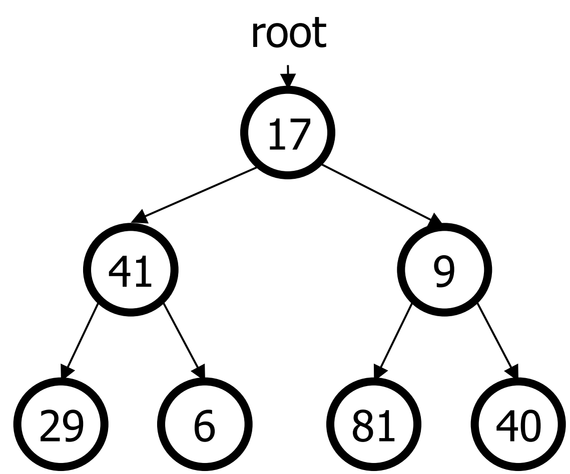
- Discuss how to make a BST class (hint: useful for MiniBrowser)
- Advanced BSTs
  - Balancing
  - Red-Black Trees
  - Splay Trees
- Non-BST binary trees
  - Heaps
  - Cartesian Trees

Implementing TreeSet and TreeMap

A BST set class

```
// TreeSet.h
// A set of integers represented as a binary search tree.
class TreeSet {
    members;
    ...
private:
    TreeNode* root;    // NULL for an empty tree
};
```

- This is basically how Stanford library's `Set` class is implemented.
- Client code talks to the `TreeSet`, not to the node objects inside it.
- Members of `TreeSet` create and manipulate nodes and pointers.



Tree member template

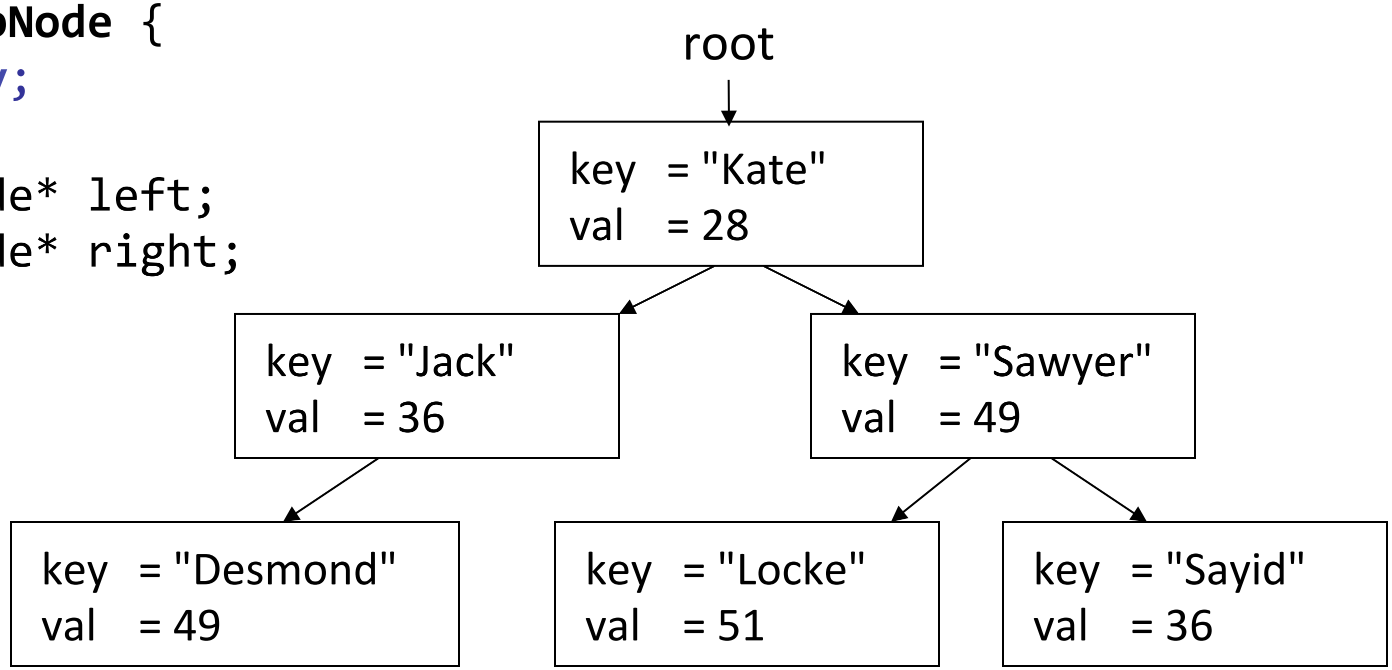
```
returnType TreeSet::functionName(parameters) {
   helperName(root, parameters);
}

returnType helperName(TreeNode* node, parameters) {
    ...
}
```

- Tree methods are often implemented recursively in 2 steps:
  - a public function intended to be called by the client
  - a "helper" function that accepts a pointer to the node to process
    - the public function typically just calls the helper and passes root node

Tree maps

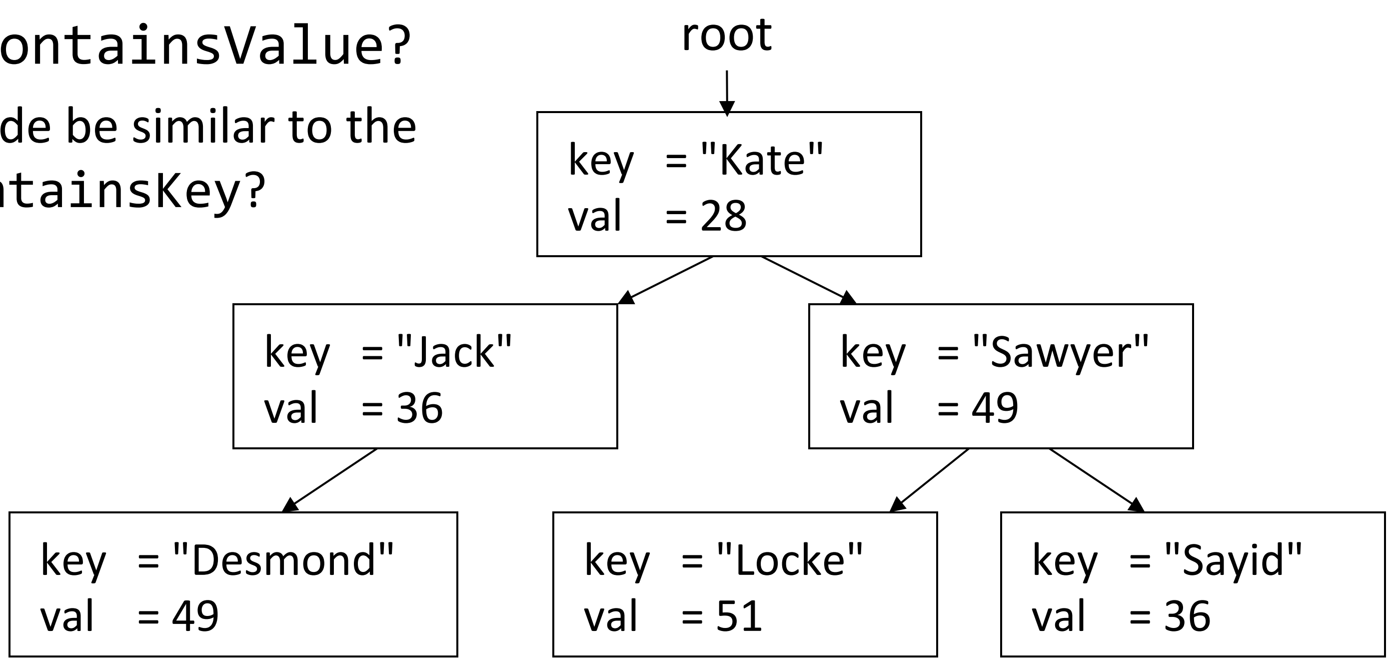
- Converting a tree set into a **tree map**:
  - Each tree node will store both a *key* and a *value*
  - tree is BST-ordered by its keys
  - **keys must be comparable (have a < operator) for ordering**

```
struct TreeMapNode {
    string key;
    int value;
    TreeMapNode* left;
    TreeMapNode* right;
};
```



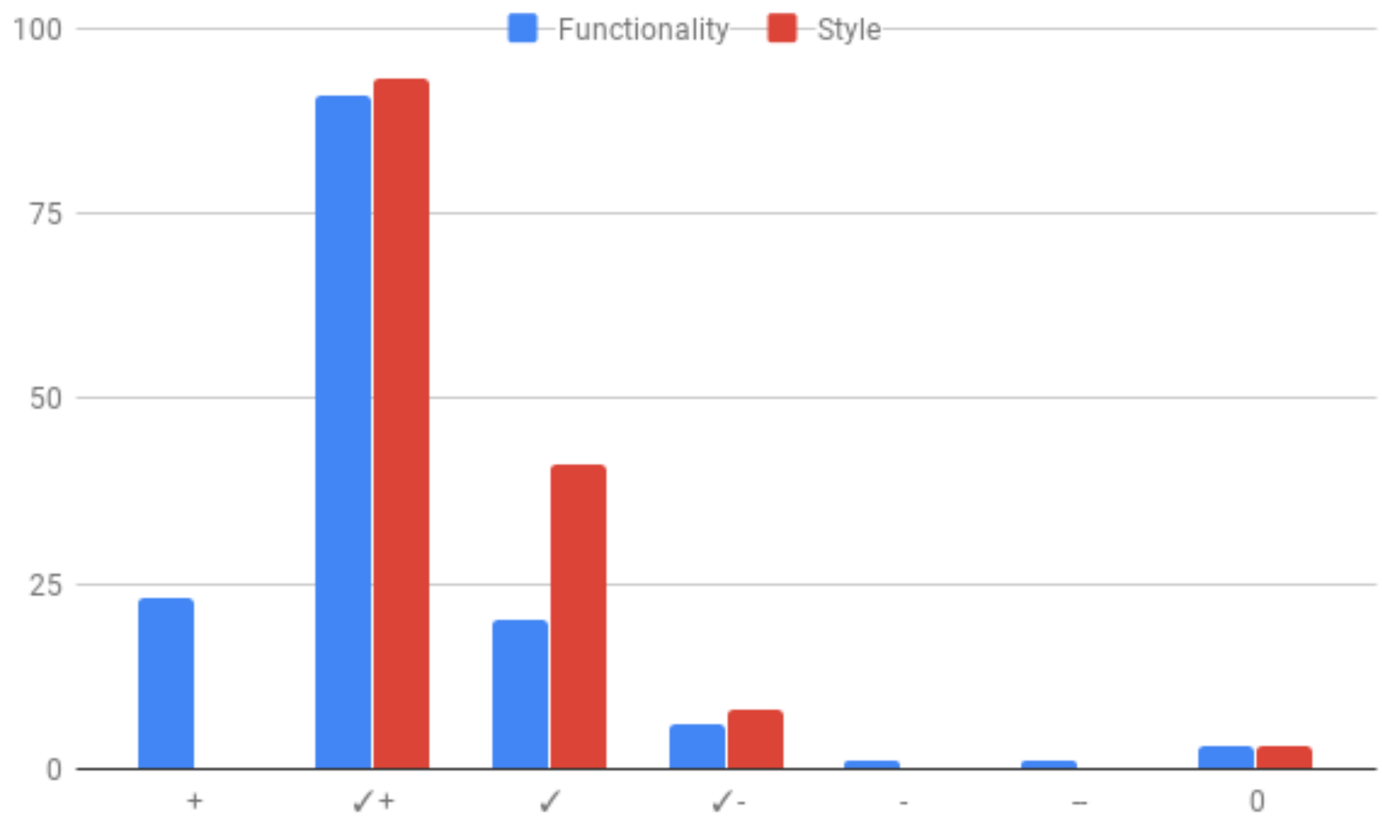
Tree map details

- Each tree set operation corresponds to one in the tree map:
  - `add(`***`value`***`)` → `put(`***`key`***`, `***`value`***`)`
  - `contains(`***`value`***`)` → `containsKey(`***`key`***`)`
  - `remove(`***`value`***`)` → `remove(`***`key`***`)`
  - must add an operation: `get(`***`key`***`)`
  - What about `containsValue`?
    - Would its code be similar to the code for `containsKey`?



Announcements

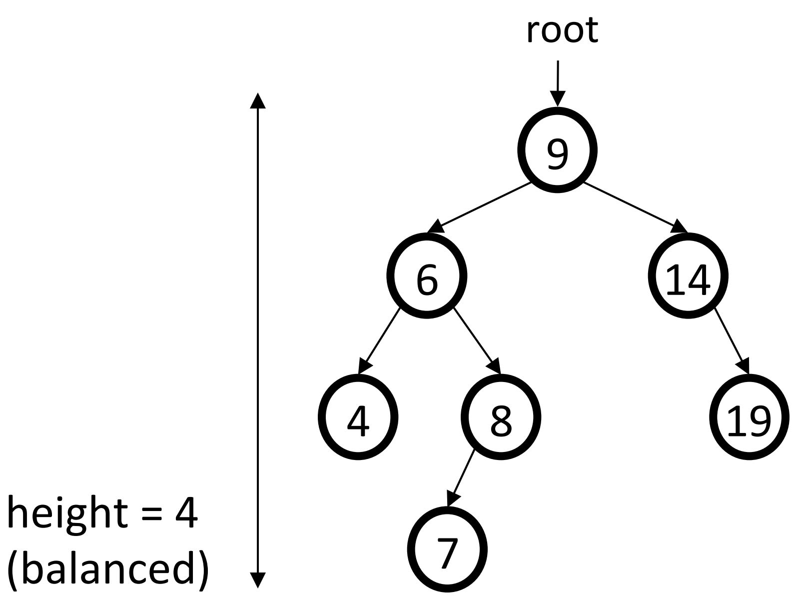
- You should be mostly done with Cache in MiniBrowser. LineManager is hard
- Homework 3 is graded. Here's the grade distribution:

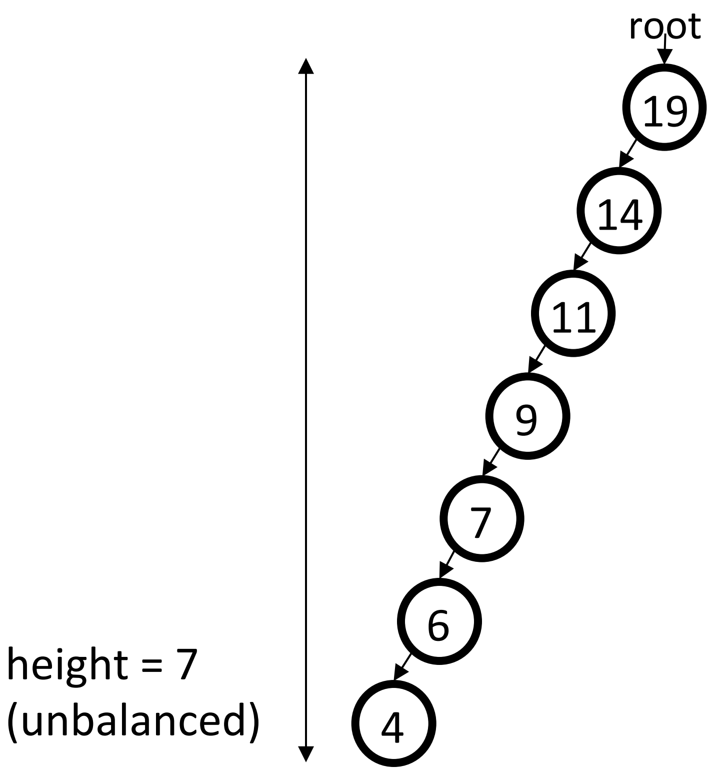


Balanced Trees

Trees and balance

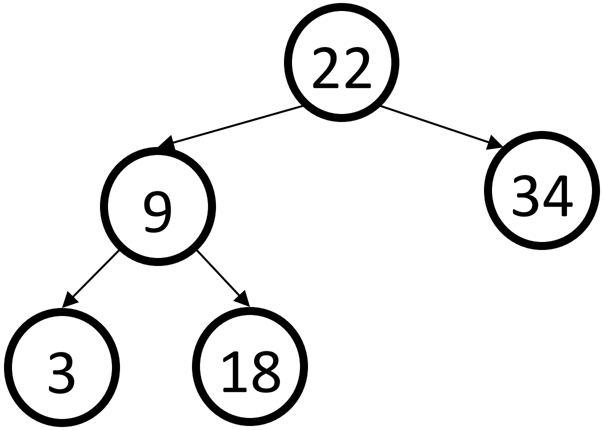
- **balanced tree**: One where for every node R, the height of R's subtrees differ by at most 1, and R's subtrees are also balanced.
  - Runtime of add / remove / contains are closely related to height.
  - Balanced tree's height is roughly $\mathbf{log_2}$ ***N***. Unbalanced is closer to ***N***.





BST balance question

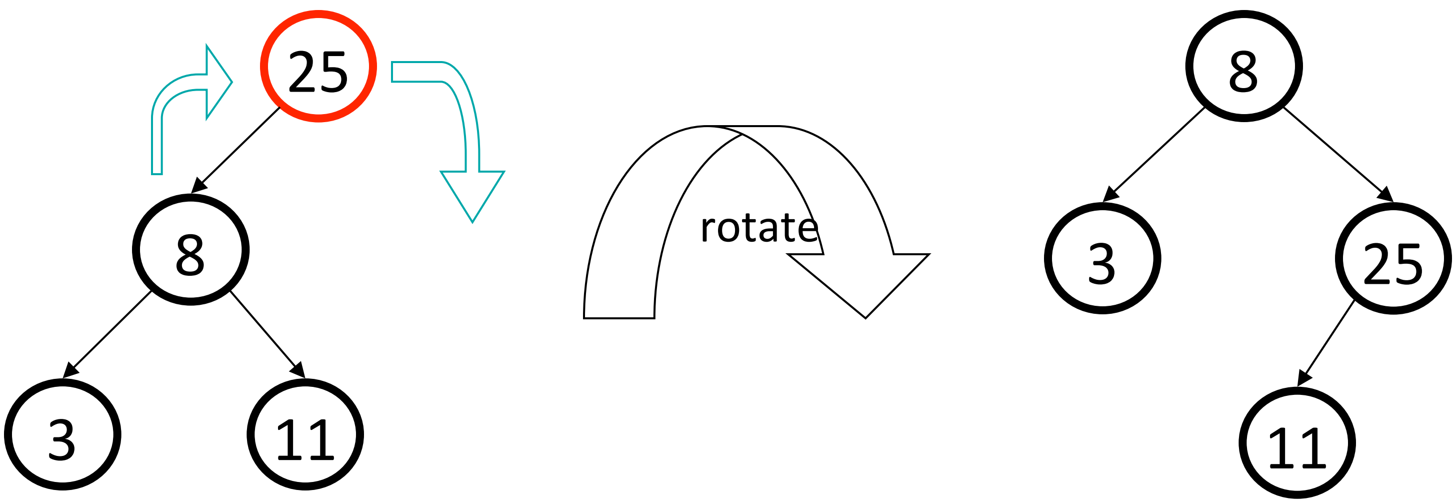
- Adding the following nodes to an empty BST in the following order produces the tree at right: 22, 9, 34, 18, 3.



- **Q:** What is an order in which we could have added the nodes to produce an unbalanced tree?
  - **A.** 18, 9, 34, 3, 22
  - **B.** 9, 18, 3, 34, 22
  - **C.** 9, 22, 3, 18, 34
  - **D.** none of the above

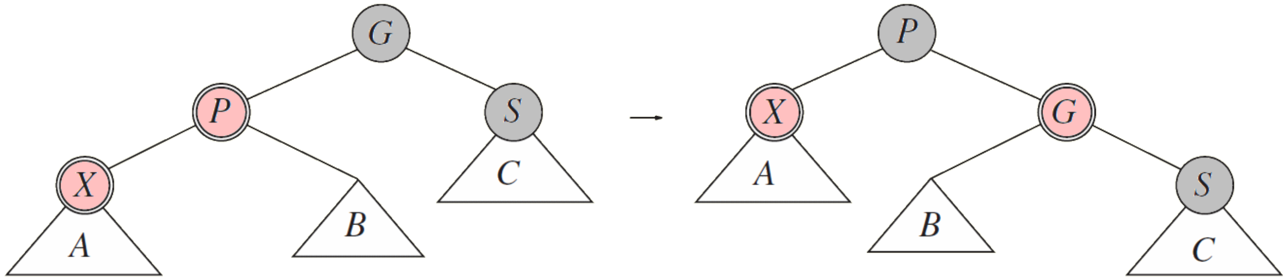
AVL trees

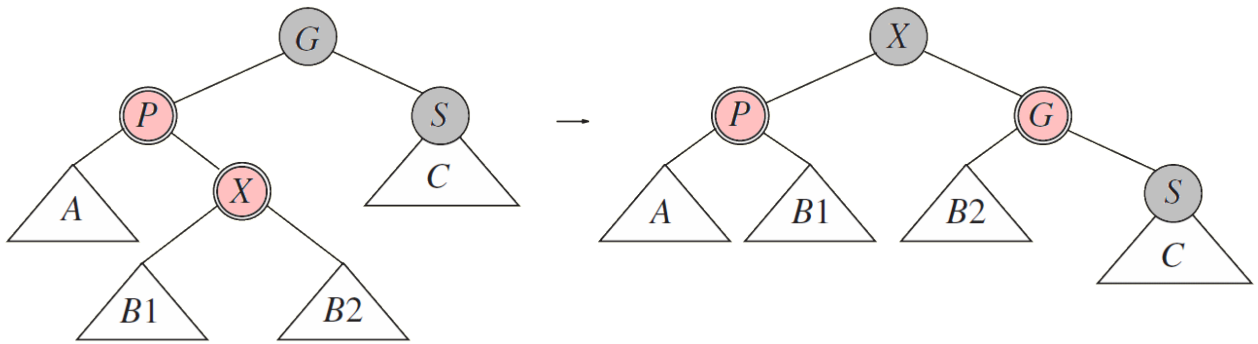
- **AVL tree**: A binary search tree that uses modified add and remove operations to stay balanced as its elements change.
  - *basic idea:* When nodes are added/removed, repair tree shape until balance is restored.
    - rebalancing is O(1); overall tree maintains an O(log *N*) height



Red-Black trees

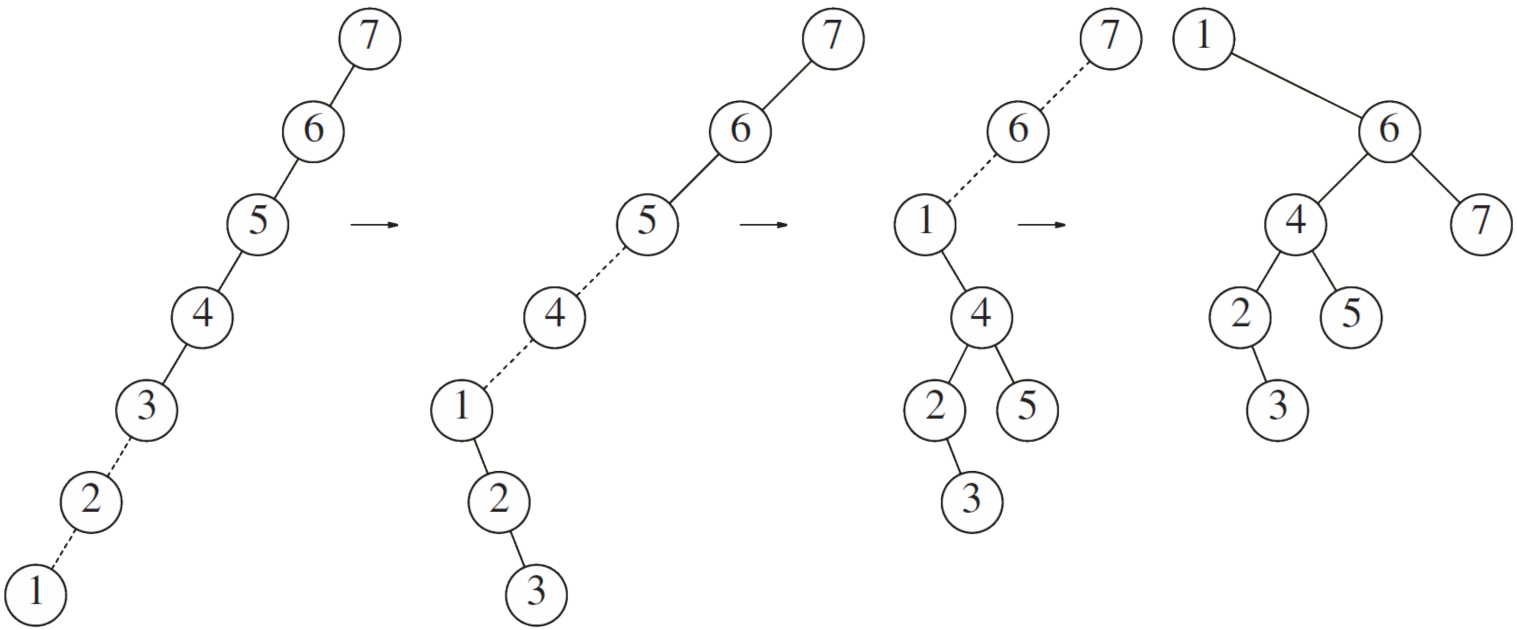
- **red-black tree:** Gives each node a "color" of red or black. (video)
  - Root is black. Root's direct children are red. All leaves are black.
  - If a node is red, its children must all be black.
  - Every path downward from a node to the bottom must contain the same number of "black" nodes.





Splay trees

- **splay tree:** Rotates each element you access to the top/root
  - very efficient when that element is accessed again (happens a lot)
  - easy to implement and does not need height field in each node



Non-BST Binary Trees

Heaps

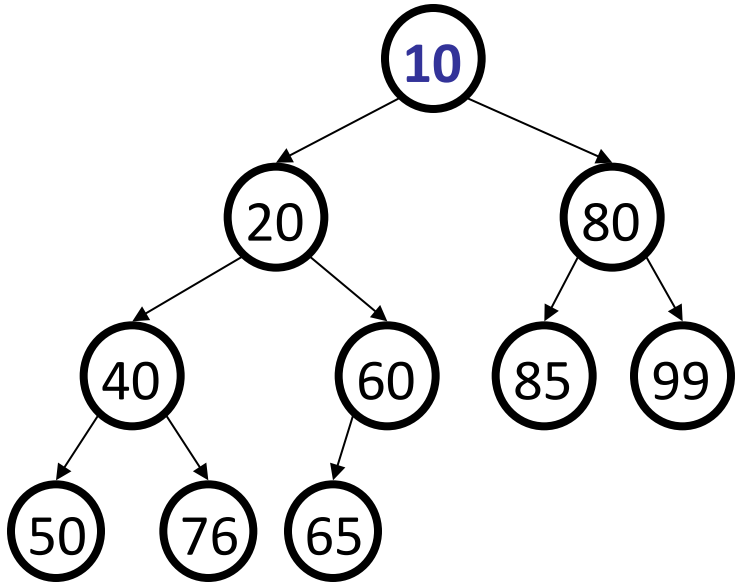
- What if you want to find the k-smallest elements in an unsorted Vector?
  - Find the top 10 students in a class?
- What if you wanted to constantly insert and remove in sorted order?
  - Model a hospital emergency room where individuals are seen in order of their urgency
  - **Priority Queue**
- What's a good choice?

Heaps

- Idea: if we use a Vector, it takes a long time to insert or remove in sorted order (or search the Vector for the smallest element)
- If we use a binary search tree, it's fast to insert and remove (O(log N)) but it's slow to find the minimum/maximum element (O(log N))
- Idea: use a tree, but store the minimum/maximum element as the root
  - Trees have log(N) insertion/deletion
  - Looking at the root is O(1)

Heaps

- **heap**: A complete binary tree with vertical ordering:
  - **min-heap**: all children must be ≥ parent's value
  - **max-heap**: all children must be ≤ parent's value

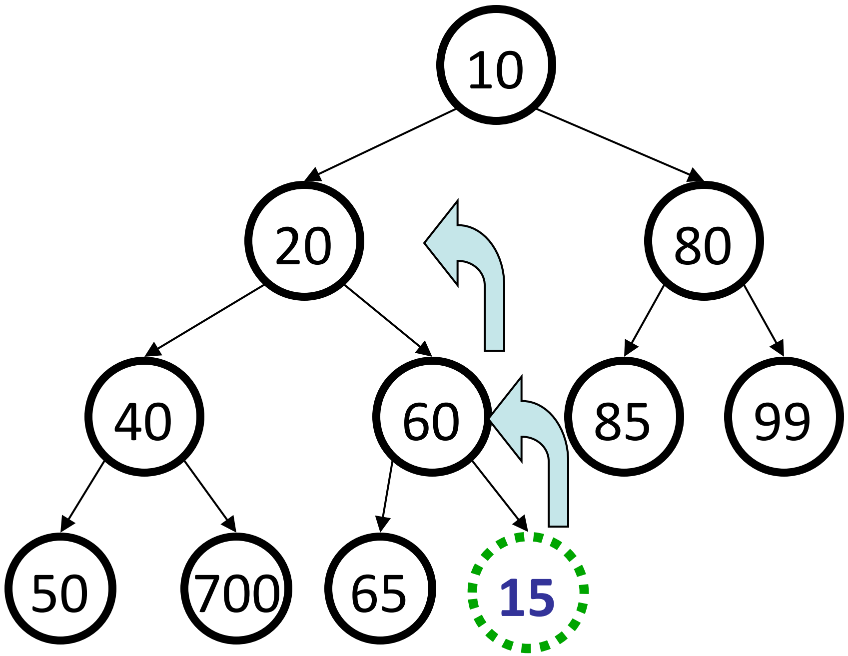


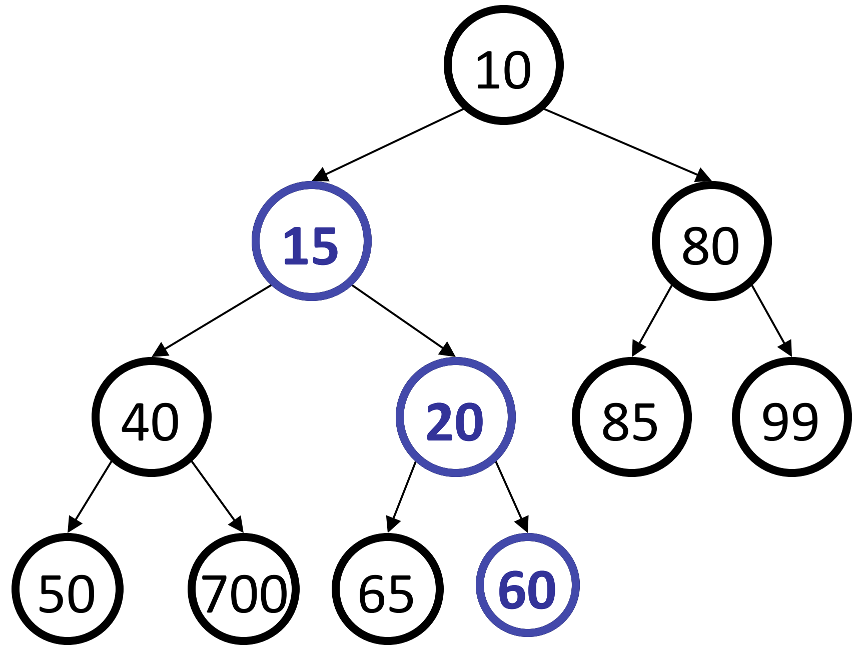
a min-heap

  - **complete tree**: all levels are full of children except perhaps the bottom level, in which all existing nodes are maximally to the left.
    - Nice corollary: heaps are *always* balanced

Heap enqueue

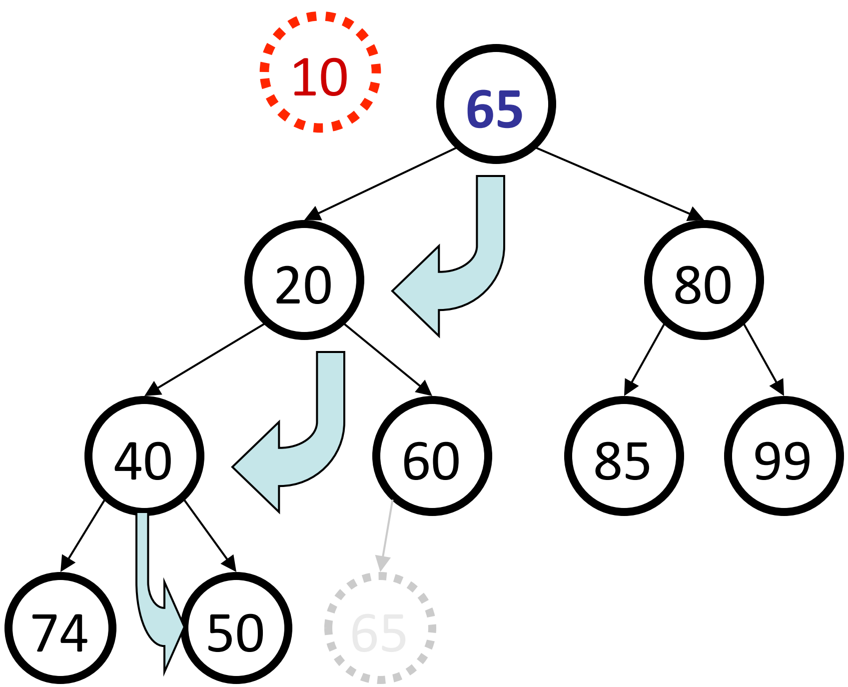
- When adding to a heap, the value is first placed at bottom-right.
  - To restore heap ordering, the newly added element is shifted ("**bubbled**") up the tree until it reaches its proper place (we reach the root, or the element is smaller than its parent [min-heap]).
  - Enqueue 15 at bottom-right; bubble up until in order.

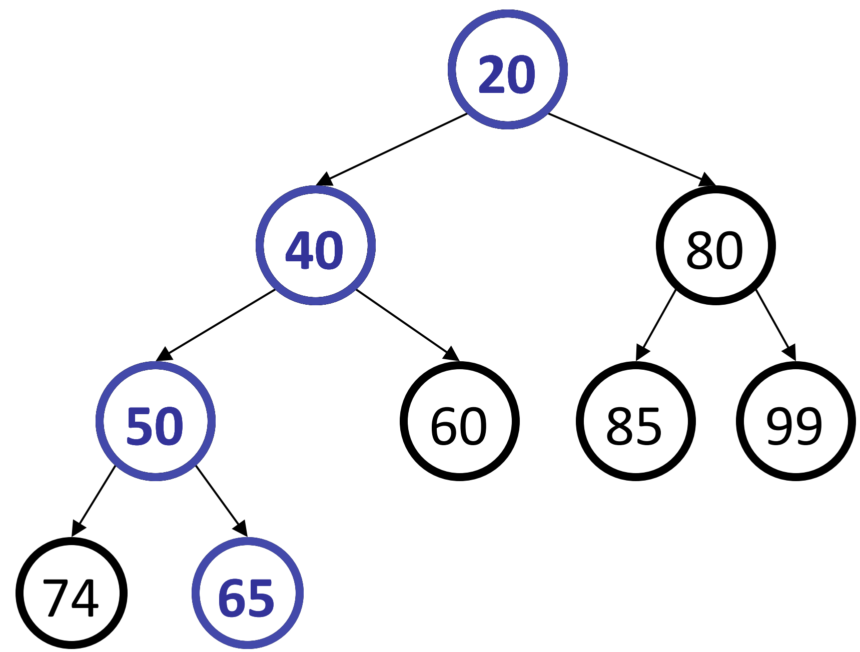




Heap dequeue

- Remove the root, and replace it with the furthest-right ancestor
- To restore heap order, the improper root is shifted ("bubbled") down the tree by swapping with its smaller child.
  - dequeue min of 10; swap up bottom-right leaf of 65; bubble down.





Cartesian Trees

- How would you quickly find the minimum/maximum element in an range?
  - Maximum elevation on a hike?
  - Best time to buy/sell a stock within a certain range of times?

Cartesian Trees

- The root stores the minimum (or maximum) element in the entire array
- The left subtree is then the minimum (or maximum) element in the range to the left of the root; the right subtree is the minimum (or maximum) element in the range to the right of the root
  - Follows the min- (or max)-heap property: every parent is smaller (or bigger) than its child

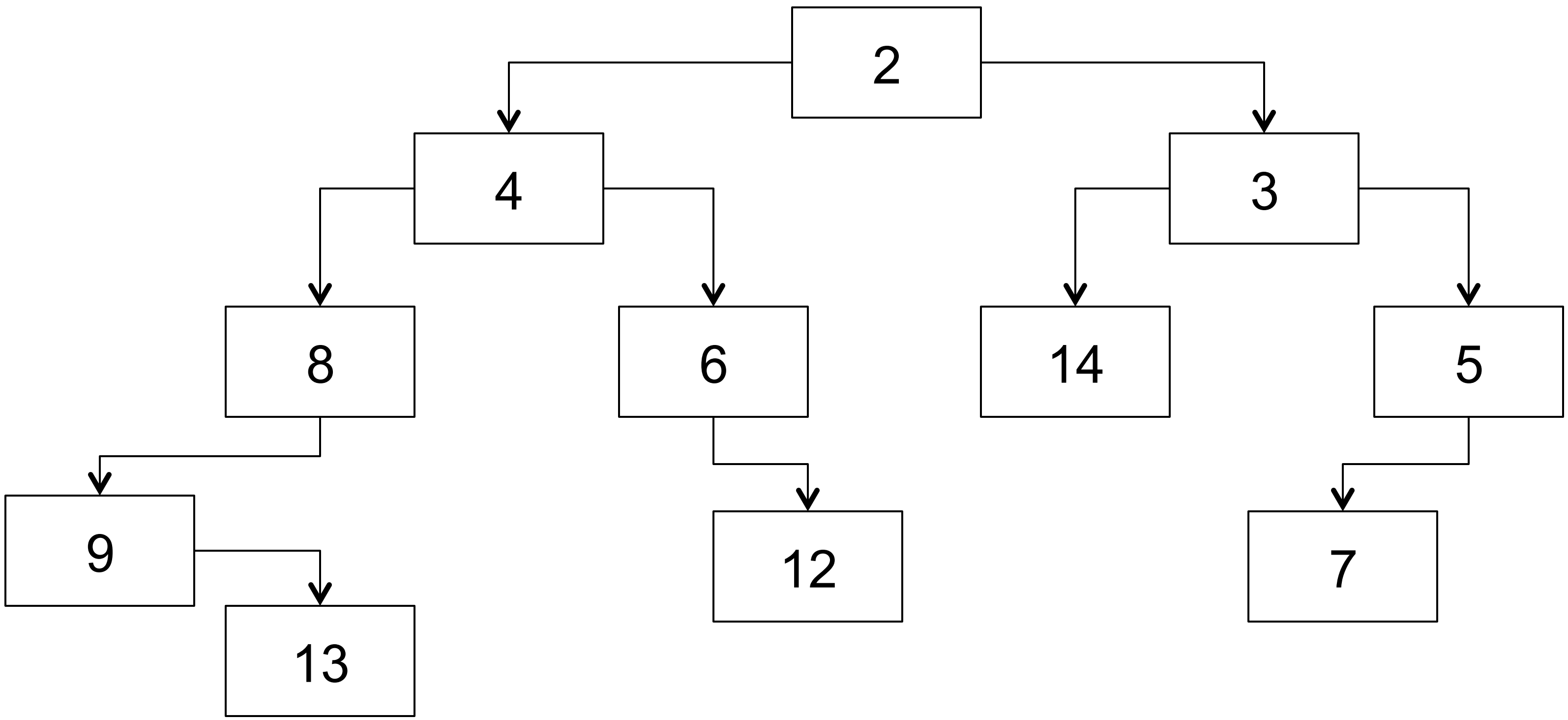
Cartesian Trees

- What would the Cartesian tree look like for this array if we're trying to find the minimum value in a range?

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
9	13	8	4	6	12	2	14	3	7	5

Cartesian Trees

- What would the Cartesian tree look like for this array if we're trying to find the minimum value in a range?



0	1	2	3	4	5	6	7	8	9	10
9	13	8	4	6	12	2	14	3	7	5

Cartesian Trees

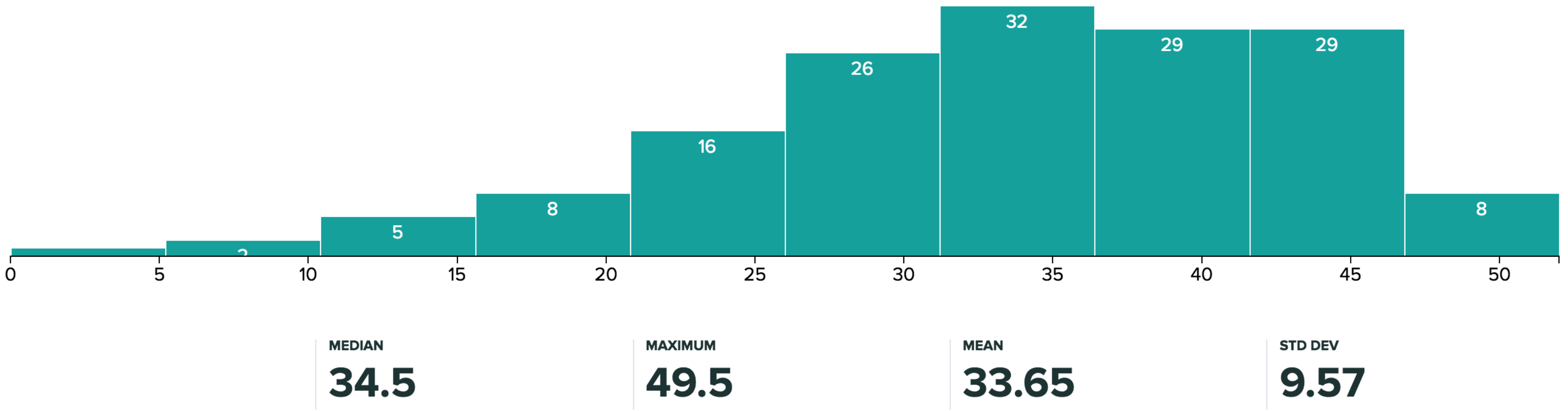
- How would we write the following function:

```
findMinElemInRange(CartesianNode *node, int start, int end)

struct CartesianNode {
    int index;
    CartesianNode *left;
    CartesianNode *right;
};
```

Exam

- This exam was a little harder than I intended. You all did really well and showed a lot of knowledge of hard topics.
- Common mistakes:
  - Big O question, part c: the inner for loop is actually O(1)
  - ADT trace problem: misreading the first for loop
  - Recursive trace: integer division with the parameters, wrong indices with substring
  - Recursive Backtracking: modified parameters after the recursive call, going out of bounds on the grid, not declaring and returning the Grid, improper base cases (returning true/false or failing to prune the tree)
  - ADT write, part a: bad scoping for the values of the Map and OBOB for string parsing
  - ADT write, part b: incomplete submissions (low on time?)

Exam


32
29
29
26
16
8
5
2
8
0
5
10
15
20
25
30
35
40
45
50
MEDIAN
34.5
MAXIMUM
49.5
MEAN
33.65
STD DEV
9.57