CSCI 136
Data Structures & Advanced Programming

Lecture 23

Fall 2019

Instructor: Bill & Sam

Administrative Details

- Lab 8: Simulations
  - You will simulate two queuing strategies
  - You can work with a partner
  - Time spent on lab before Wed. is time well-spent!
- Problem Set 3 is online
  - Due this Friday at beginning of class

Last Time

- Improving Huffman's Algorithm
- Priority Queues & Heaps
  - A "somewhat-ordered" data structure

Today

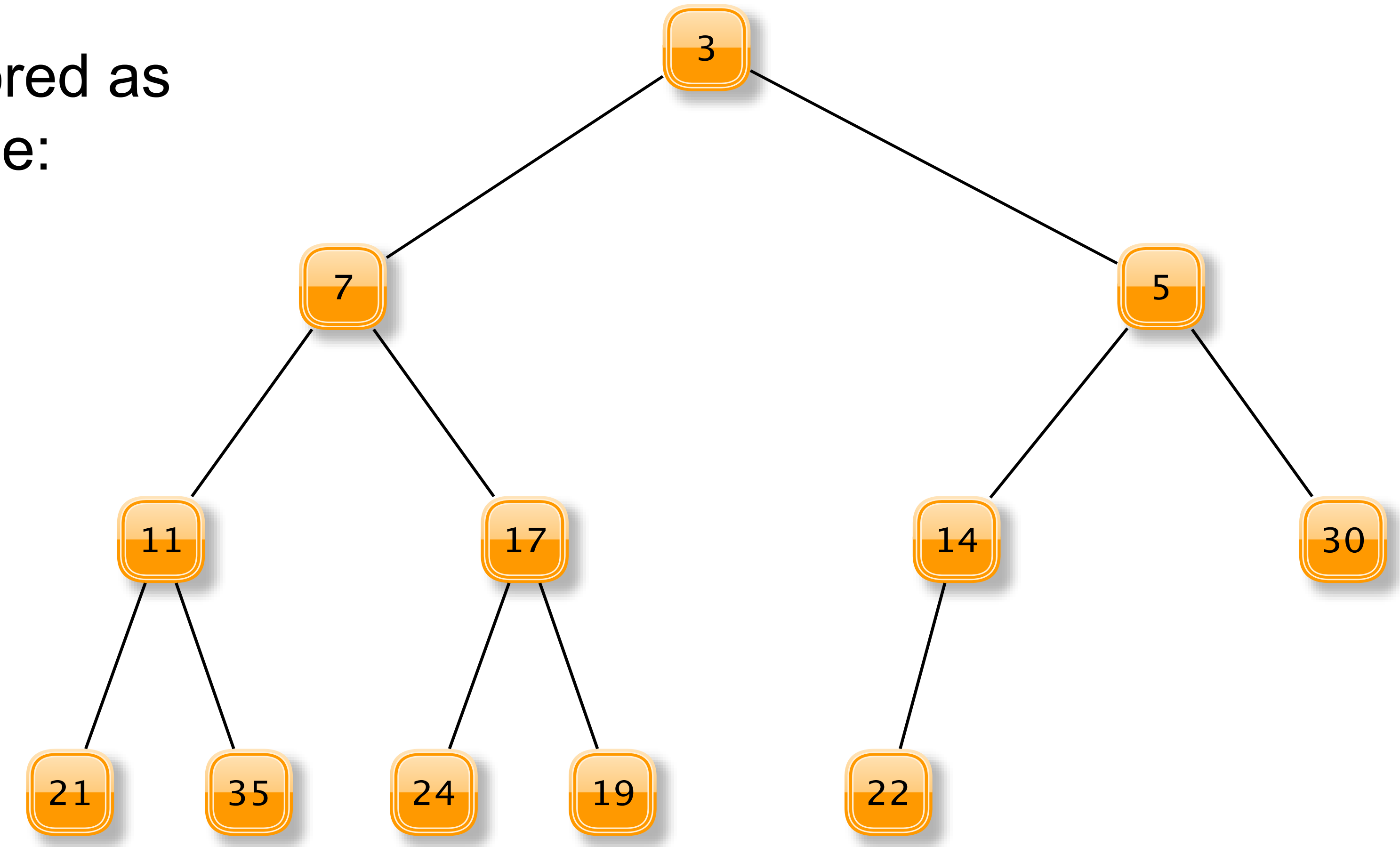
- Finishing up with heaps
  - HeapSort
  - Alternative Heap Structures
- Binary Search Tree: A New Ordered Structure
  - Definitions
  - Implementation

Implementing Heaps

- VectorHeap
  - Use conceptual array representation of BT (ArrayTree)
  - But use extensible Vector instead of array (makes adding elements easier)
  - Note:
    - Root of tree is location 0 of Vector
    - Children of node in location i are in locations 2i+1 (left) and 2i+2 (right)
    - Parent of node i is in location (i-1)/2

Heap

Stored as Tree:



Stored as Vector:

3	**7**	**5**	**11**	**17**	**14**	**30**	**21**	**35**	**24**	**19**	**22**
0	1	2	3	4	5	6	7	8	9	10	11

Implementing Heaps

- Features
  - No gaps in array (array is *complete*)-- why?
    - We always add in next available array slot (left-most available spot in binary tree;
    - We always remove using "final" leaf
  - *Heap Invariant becomes*
    - data[i] <= data[2i+1]; data[i]<=data[2i+2] (or kids might be null)
  - When elements are added and removed, do small amount of work to "re-heapify"
    - How small? Note: finding a node's child or parent takes constant time, as does finding "final" leaf or next slot for adding
    - Since this heap corresponds to a full binary tree, the depth of the tree is O(log n), so percolate/pushDown takes O(log n) time!

VectorHeap Summary

- Let's look at VectorHeap code....

- Add/remove are both O(log n)
- Data is not completely sorted
  - "Partial" order is maintained
- Note: VectorHeap(Vector<E> v)
  - Takes an unordered Vector and uses it to construct a heap
  - How?

Heapifying A Vector (or array)

- Method I: Top-Down
  - Assume V[0...k] satisfies the heap property
  - Now call percolate on item in location k+1
  - Then V[0..k+1] satisfies the heap property
- Method II: Bottom-up
  - Assume V[k..n] satisfies the heap property
  - Now call pushDown on item in location k-1
  - Then V[k-1..n] satisfies heap property

Heapifying A Vector (or array)

- Method I: Top-Down
  - Assume V[0...k] satisfies the heap property
  - Now call percolate on item in location k+1
  - Then V[0..k+1] satisfies the heap property
- Method II: Bottom-up
  - Assume V[k..n] satisfies the heap property
  - Now call pushDown on item in location k-1
  - Then V[k-1..n] satisfies heap property

Top-Down vs Bottom-Up

- Top-down heapify: elements at depth d may be swapped d times: Total # of swaps is at most

$$\sum_{d=0}^{h} d2^d = (h-1)2^{h+1} + 2 = (\log n - 1)2n + 2$$

- This is O(n log n)
- Some intuition: most of the elements are in the lowest levels of the tree, so each of them might have to move to root: O(log n) swaps per element

Top-Down vs Bottom-Up

- Bottom-up heapify: elements at depth d may be swapped h-d times: Total # of swaps is at most

$$\sum_{d=0}^{h} (h-d)2^d = 2^{h+1} - h - 2 = 2n - \log n + 2$$

  - This is O(n) --- beats top-down!
  - Some intuition: most of the elements are in the lowest levels of the tree, so each of them will only be pushed down (swapped) a small number of times SO COOL!!!

Some Sums

$$\sum_{d=0}^{d=k} 2^d = 2^{k+1} - 1$$

$$\sum_{d=0}^{d=k} r^d = (r^{k+1} - 1)/(r-1)$$

$$\sum_{d=0}^{d=k} d * 2^d = (k-1) * 2^{k+1} + 2$$

$$\sum_{d=0}^{d=k} (k-d) * 2^d = 2^{k+1} - k - 2$$

All of these can be proven by (weak) induction.

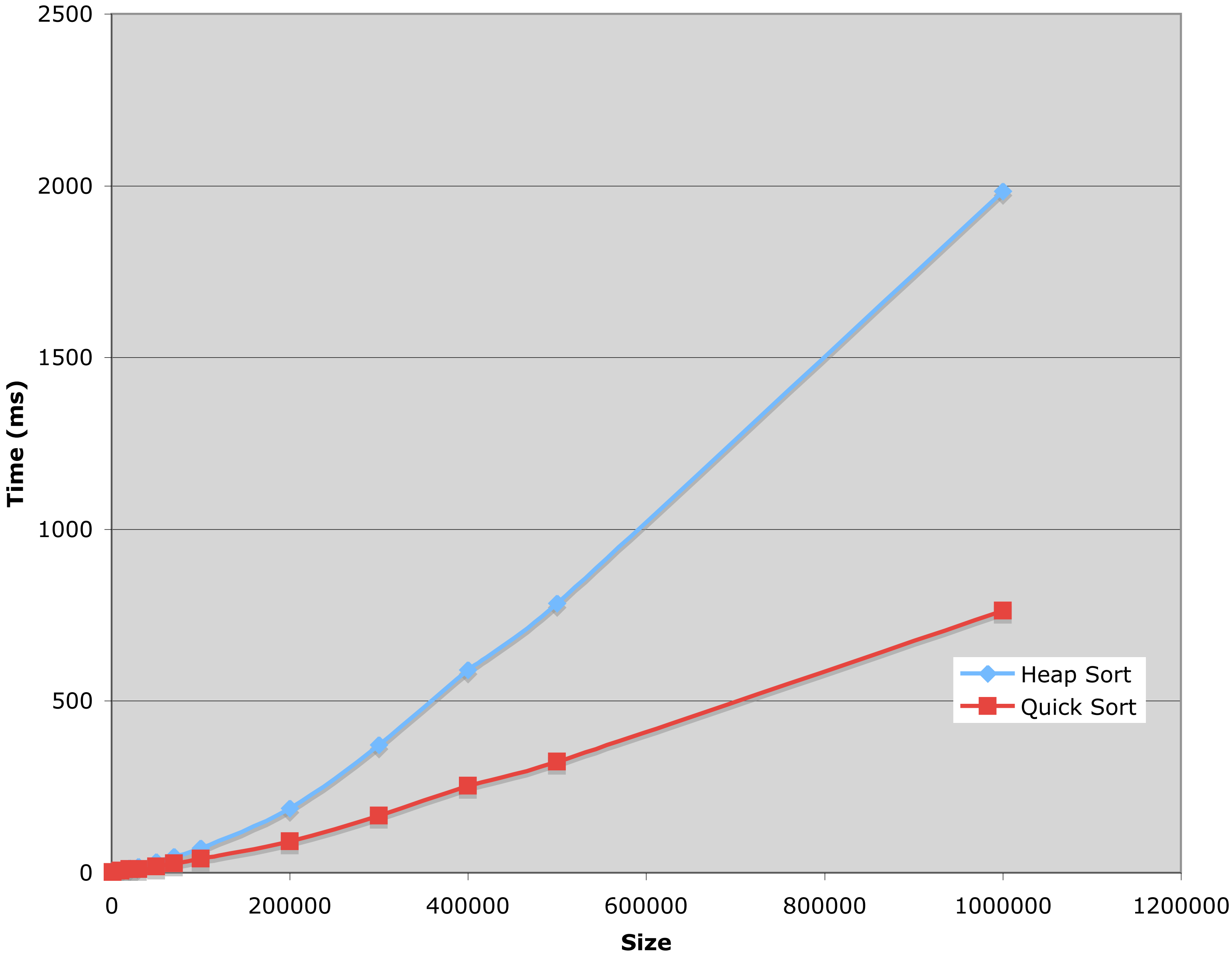
Try these to hone your skills

The second sum is called a geometric series. It works for any $r \neq 1$

HeapSort

- Heaps yield another O(n log n) sort method
- To HeapSort a Vector “in place”
  - Perform bottom-up heapify on the reverse ordering: that is: highest rank/lowest priority elements are near the root (low end of Vector)
  - Now repeatedly remove elements to fill in Vector from tail to head
    - For(int i = v.size() – 1; i > 0; i--)
      - RemoveMin from v[0..i] // v[i] is now not in heap
      - Put removed value in location v[i]

Heap Sort vs QuickSort



Why Heapsort?

- Heapsort is slower than Quicksort in general
- Any benefits to heapsort?
  - *Guaranteed* O(n log n) runtime
- Decent performance on mostly sorted data, unlike quicksort
- Good for incremental sorting

More on Heaps

- Set-up: We want to build a *large* heap. We have several processors available.
- We'd like to use them to build smaller heaps and then merge them together
- Suppose we can share the array holding the elements among the processors.
  - How long to merge two heaps?
  - How complicated is it?
- What if we use BinaryTrees for our heaps?

Mergeable Heaps

- We now want to support the additional operation merge(heap1, heap2)
- Basic idea: heap with larger root somehow points into heap with smaller root
- Challenges
  - Points how? Where?
  - How much reheapifying is needed
  - How deep do trees get after many merges?

Skew Heap

- Don’t force heaps to be complete BTs?
- Develop recursive merge algorithm that keeps tree shallow over time
- Theorem: Any set of m SkewHeap operations can be performed in O(m log n) time, where n is the total number of items in the SkewHeaps
- Let’s sketch out merge operation....

Skew Heap: Merge Pseudocode

```
SkewHeap merge(SkewHeap S, SkewHeap T)
    if either S or T is empty, return the other
    if T.minValue < S.minValue
        swap S and T        (S now has minValue)
    if S has no left subtree, T becomes left subtree
    else
        let temp point to right subtree of S
        left subtree of S becomes right subtree of S
        merge(temp, T) becomes left subtree of S
    return S
```

Tree Summary

- Trees
  - Express hierarchical relationships
  - Tree structure captures relationship
    - i.e., ancestry, game boards, decisions, etc.
- Heap
  - Partially ordered tree based on item priority
  - Node invariants: parent has higher priority than each child
  - Provides efficient PriorityQueue implementation

Improving on OrderedVector

- The OrderedVector class provides O(log n) time searching for a group of n comparable objects
    - add() and remove(), though, take O(n) time in the worst case---and on average!
- Can we improve on those running times without sacrificing the O(log n) search time?
- Let's find out....

Binary Trees and Orders

- Binary trees impose multiple orderings on their elements (pre-/in-/post-/level-orders)
- In particular, in-order traversal suggests a natural way to hold comparable items
  - For each node v in tree
    - All values in left subtree of v are at most v
    - All values in right subtree of v are at least v
- This leads us to...

Binary Search Trees

- Binary search trees maintain a *total* ordering among elements
- Definition: A BST T is either:
  - Empty
  - Has root r with subtrees T_L and T_R such that
    - All nodes in T_L have smaller value than r
    - All nodes in T_R have larger value than r
    - T_L and T_R are also BSTs
- Examples

BST Observations

- The same data can be represented by many BST shapes
- Searching for a value in a BST takes time proportional to the height of the tree
  - Reminder: trees have height, nodes have depth
- Additions to a BST happen at nodes missing at least one child (*a constraint!*)
- Removing from a BST can involve *any* node

BST Operations

- BSTs will implement the OrderedStructure Interface
  - `add(E item)`
  - `contains(E item)`
  - `get(E item)`
  - `remove(E item)`
  - `iterator()`
    - This will provide an in-order traversal
- Runtime of add, contains, get, remove: O(height)
- Goal: Keep the height to O(log n)
    - Duane's BinarySearchTree class doesn't achieve this…
    - But his RedBlackSearchTree does!

Application: Dictionary

- Create a BST of ComparableAssociations
  - Order BST by key
  - Two objects are equal if keys are equal

- Example: Symbol tables (PostScript lab) are Dictionaries
  - But would only use a BST if the set of possible symbols was very large
- What lab used a large dictionary?

Application: Tree Sort

- Can we sort data using a BST?
  - Yes!
- Runtime?
  - To build a tree with n elements, we do n insertions: $O(n*h)$, where h is the maximum height attained by the tree
  - In order traversal: $O(n)$
  - Total runtime: $O(n*h)$

BST Implementation

- The BST holds the following items
  - BinaryTree root: the root of the tree
  - BinaryTree EMPTY: a static empty BinaryTree
    - To use for all empty nodes of tree
  - int count: the number of nodes in the BST
  - Comparator<E> ordering: for comparing nodes
    - Note: E must implement Comparable
- Two constructors: One takes a Comparator
  - The other creates a NaturalComparator

BST Implementation: locate

- Several methods search the tree: add, remove, contains
- We factor out common code: locate method
- *protected* locate(BinaryTree<E> *b*, E *v*)
  - Returns a BinaryTree<E> in the subtree with root *node* such that either
    - *node* has its value equal to *v*, or
    - *v* is not in this subtree and *node* is where *v* would be added as a (left or right) child
- How would we implement locate()?

BST Implementation: locate

```
BinaryTree locate(BinaryTree root, E value)
    if root's value equals value return root
    child ← child of root that should hold value
    if child is empty tree, return root
        // value not in subtree based at root
    else //keep looking
        return locate(child, value)
```

BST Implementation: locate

- What about this line?

  child ← *child of root that should hold value*
- If the tree can have multiple nodes with same value, then we need to be careful
- Convention: During *add* operation, only move to right subtree if value to be added is *greater than* value at node
- We'll look at *add* later
- Let's look at *locate* now....

The code : locate

```
protected BinaryTree<E> locate(BinaryTree<E> root, E value) {
       E rootValue = root.value();
       BinaryTree<E> child;

       // found at root: done
       if (rootValue.equals(value)) return root;

       // look left if less-than, right if greater-than
       if (ordering.compare(rootValue,value) < 0)
           child = root.right();
       else
           child = root.left();

       // no child there: not in tree, return this node,
       // else keep searching
       if (child.isEmpty()) return root;
       else
           return locate(child, value);
}
```

Other core BST methods

- locate(v) returns either a node containing v or a node where v can be added as a child
- locate() is used by
  - `public boolean contains(E value)`
  - `public E get(E value)`
  - `public void add(E value)`
  - `Public void remove(E value)`
- Some of these also use another utility method
  - `protected BT predecessor(BT root)`
- Let's look at contains() first...

Contains

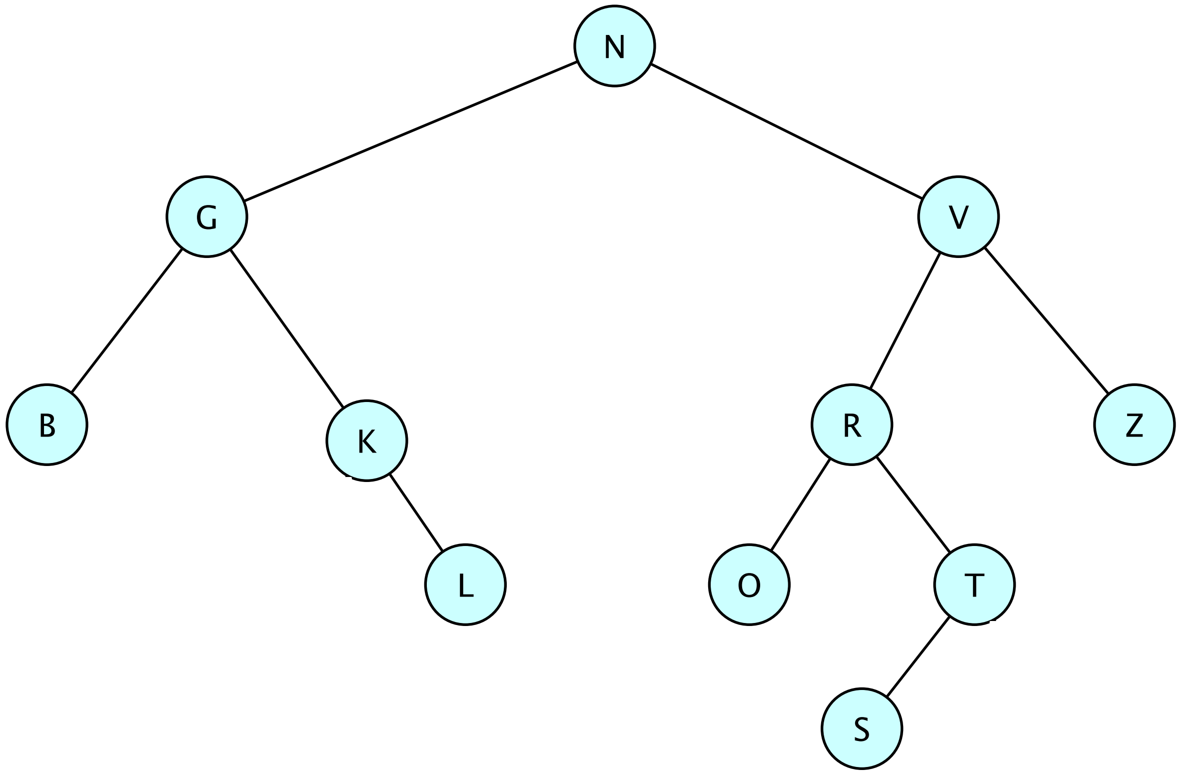
```
public boolean contains(E value){
   if (root.isEmpty()) return false;

   BinaryTree<E> possibleLocation = locate(root,value);

   return value.equals(possibleLocation.value());
}
```

First (Bad) Attempt: add(E value)

```
public void add(E value) {
        BinaryTree<E> newNode = new BinaryTree<E>(value,EMPTY,EMPTY);
        if (root.isEmpty()) root = newNode;
        else {
                BinaryTree<E> insertLocation = locate(root,value);
                E nodeValue = insertLocation.value();
        if (ordering.compare(nodeValue,value) < 0)
                insertLocation.setRight(newNode);
        else
                insertLocation.setLeft(newNode);
        }
        count++;
}
```

Problem: If repeated values are allowed, left subtree might not be empty when setLeft is called

Add: Repeated Nodes



Where would a new K be added?
A new V?

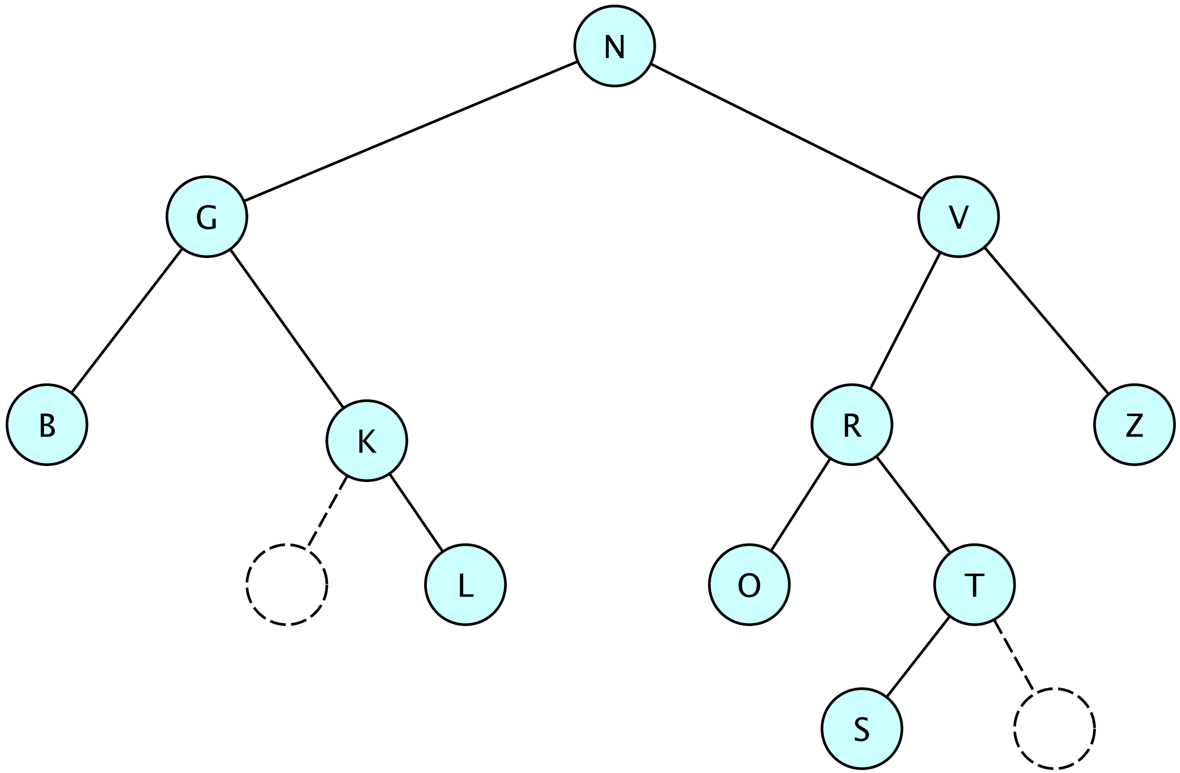
Add Duplicate to Predecessor

- If insertLocation has a left child then
  - Find insertLocation’s predecessor
  - Add repeated node as right child of predecessor
  - Predecessor will be in insertLocation’s left sub-tree
    - Do you believe me?

Corrected Version: add(E value)

```
BinaryTree<E> newNode = new BinaryTree<E>(value,EMPTY,EMPTY);
if (root.isEmpty()) root = newNode;
else {
   BinaryTree<E> insertLocation = locate(root,value);
   E nodeValue = insertLocation.value();
   if (ordering.compare(nodeValue,value) < 0)
      insertLocation.setRight(newNode);
   else
      if (insertLocation.left().isEmpty())
         insertLocation.setLeft(newNode);
      else
         // if value is in tree, we insert just before
         predecessor(insertLocation).setRight(newNode);
}
count++;
```

How to Find Predecessor



Where would a new K be added?
A new V?

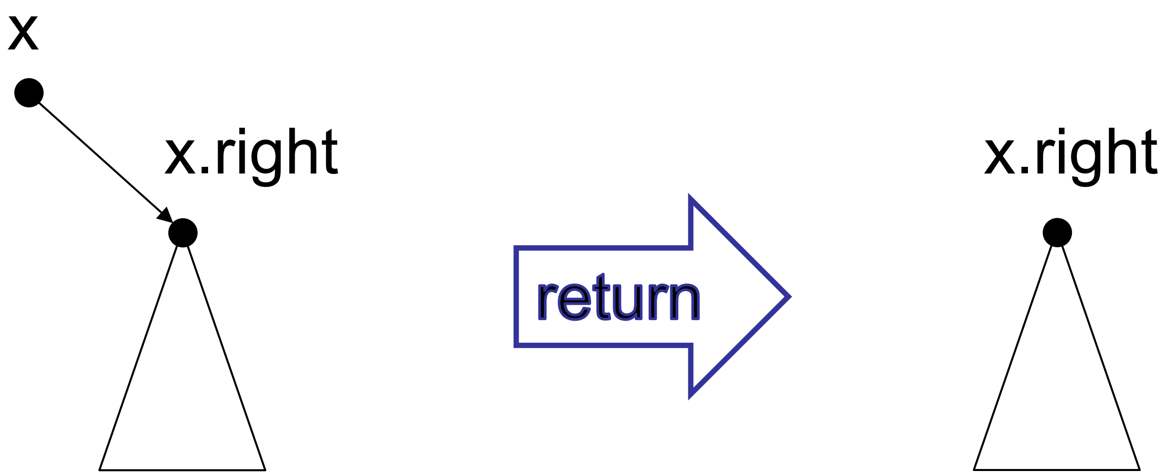
Predecessor

```
protected BinaryTree<E> predecessor(BinaryTree<E> root) {
    Assert.pre(!root.isEmpty(), ”Root has predecessor");
    Assert.pre(!root.left().isEmpty(),"Root has left child.");

    BinaryTree<E> result = root.left();

    while (!result.right().isEmpty())
        result = result.right();

    return result;
}
```

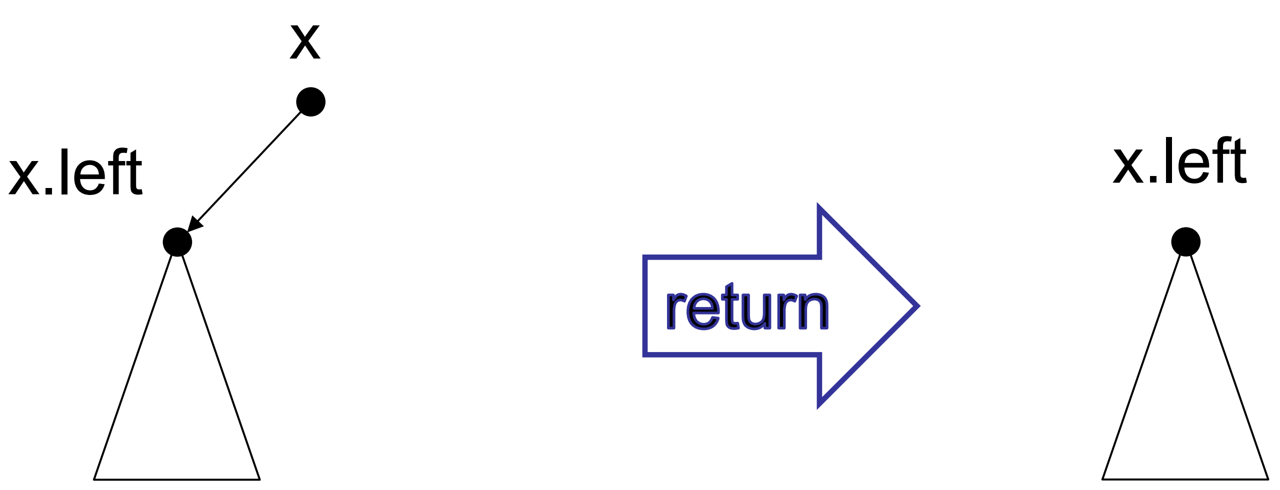
Removal

- Removing the root is a (not so) special case
- Let's figure that out first
  - If we can remove the root, we can remove any element in a BST in the same way
    - Do you believe me?
- We need to implement:
  - `public E remove(E item)`
  - `protected BT removeTop(BT top)`

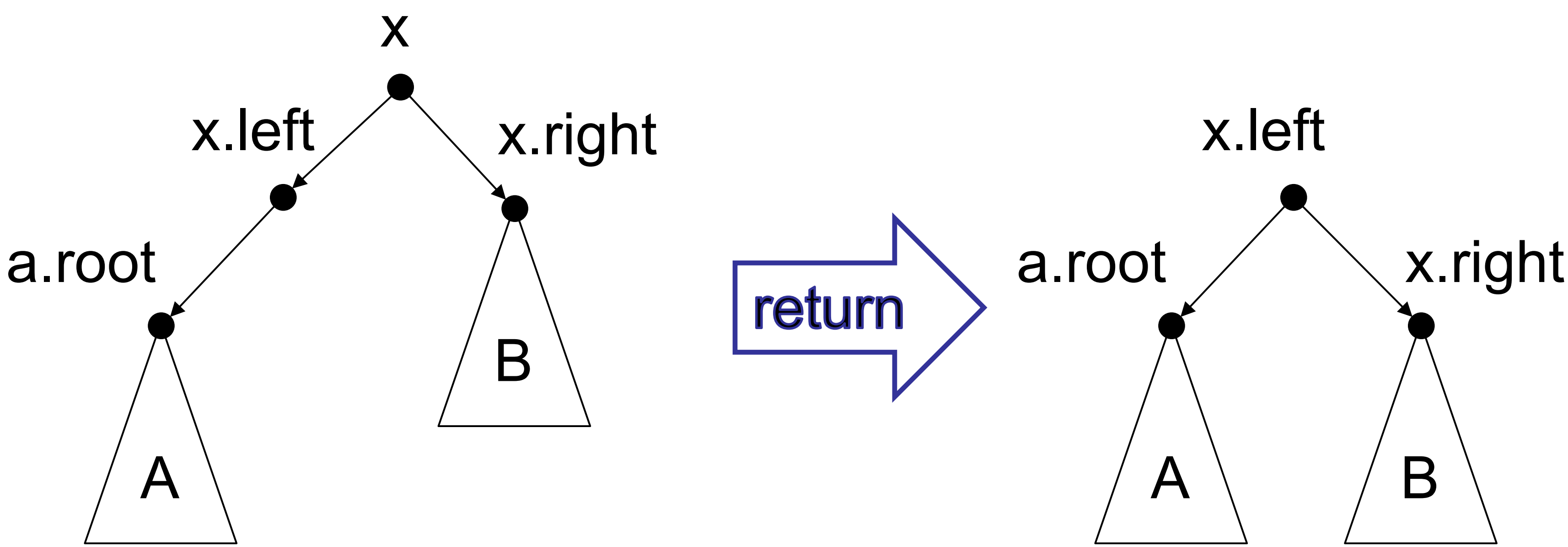
Case 1: No left binary tree



Case 2: No right binary tree



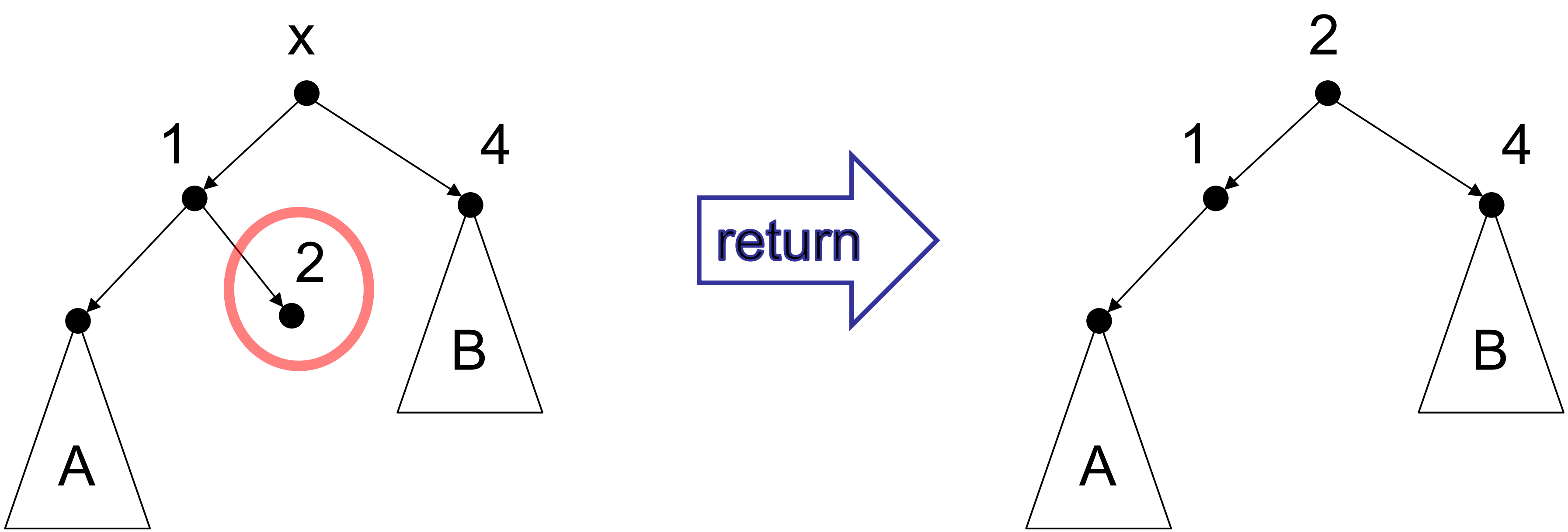
Case 3: Left has no right subtree



Case 4: General Case (HARD!)

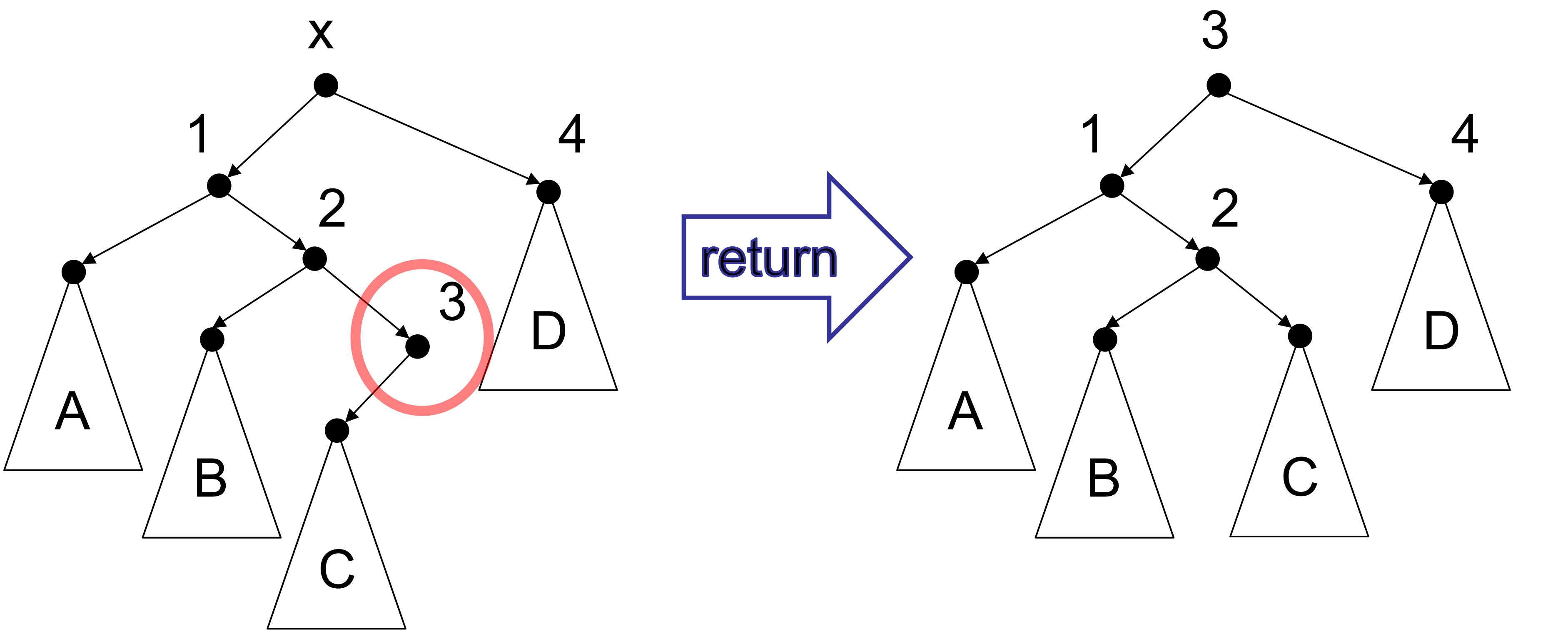
- Consider BST requirements:
  - Left subtree must be <= root
  - Right subtree must be > root
- Strategy: replace the root with the largest value that is less than or equal to it
  - predecessor(root) : rightmost left descendant
- This may require reattaching the predecessor's left subtree!

Case 4: General Case (HARD!)



Replace root with predecessor(root),
then patch up the remaining tree

Case 4: General Case (HARD!)



Replace root with predecessor(root), then patch up the remaining tree

RemoveTop(topNode)

Detach left and right sub-trees from root (i.e. topNode)

If either left or right is empty, **return** the other

If left has no right child

  make right the right child of left then **return** left

Otherwise find largest node C in left

  // C is the right child of its own parent P

  // C is the predecessor of right (ignoring topNode)

Detach C from P; make C's left child the right child of P

Make C new root with left and right as its sub-trees

But What About Height?

- Can we design a binary search tree that is always “shallow”?
- Yes! In many ways. Here’s one
- AVL trees
  - Named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published a paper about AVL trees in 1962 called "An algorithm for the organization of information"