Data Structures and Algorithms

Lecture 8: Binary Search Trees

Department of Computing

Curtin University

Last updated: 30 March 2016

Copyright Warning

This Week

- Trees
- Binary search trees (BSTs)
  - Structure and data organisation
  - Terminology
  - Methods
- Big-O time complexity analysis
  - Complete, almost-complete and degenerate BSTs

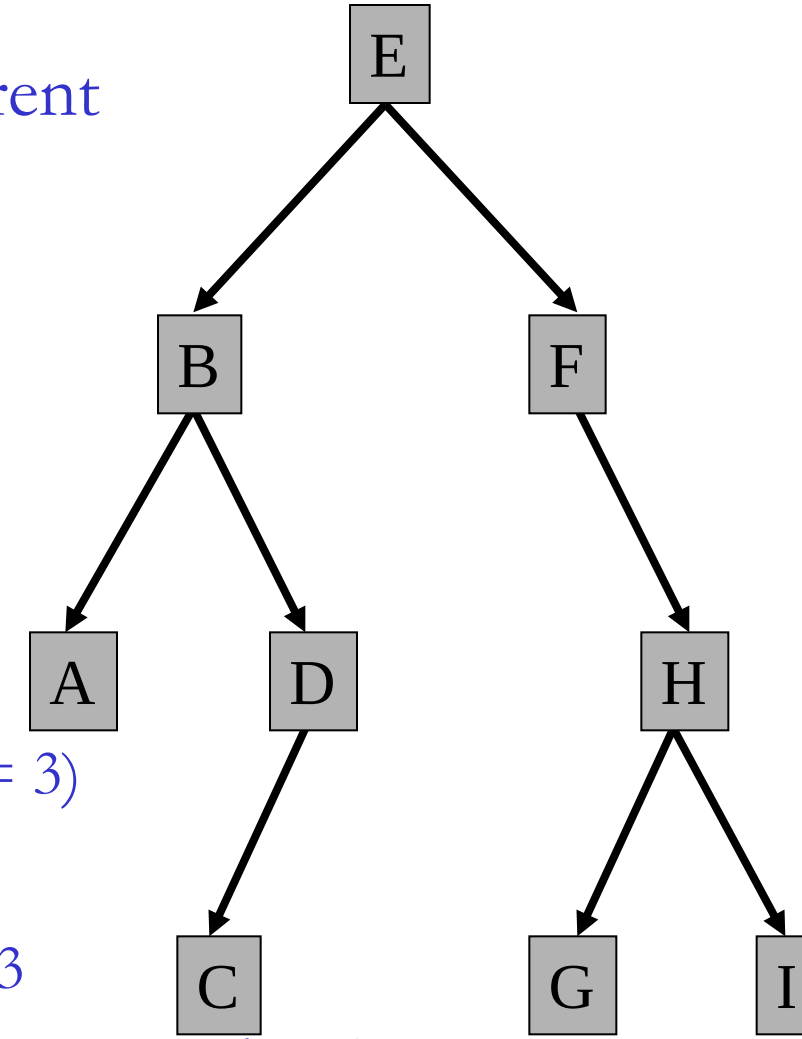
What is a Tree?

- A tree is a data structure to represent hierarchical relationships
  - It's still concerned with storing a set of values, but now the set is organised *hierarchically* rather than in a line
- Examples of trees in everyday life:
  - Genealogical tree (ancestry)
  - Armed forces command structure
- Examples in computer science
  - Binary search trees, heaps
  - Natural language processing
  - Unix file system

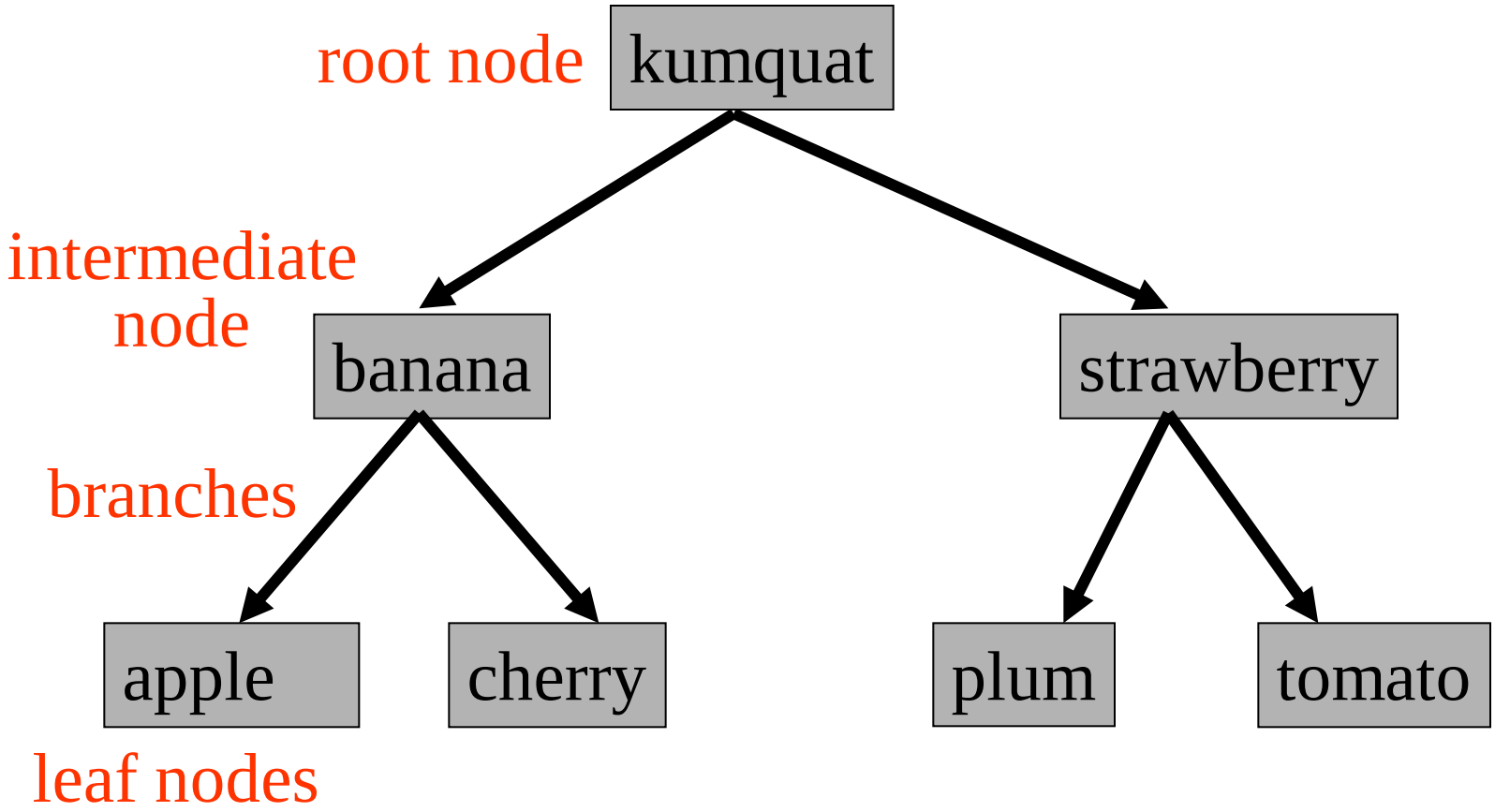
Unix file system tree

Tree Terminology

- Root: Top-most node that has no parent
- Parent: *e.g.*, B is parent of A and D
  - Each node only has one parent
- Child: *e.g.*, H is a child of F
- Leaf: Node with no children
  - A, C, G and I are the leaves in this tree
- Height: Length of path from root to the most distant leaf node (E to C, ht = 3)
- Depth: Path from root to given node
  - Root E is at depth = 0, G is at depth = 3
- Level: A set of nodes that are all at the same depth
  - *e.g.*, Nodes A, D, H are all at the third level. Tree has 4 levels

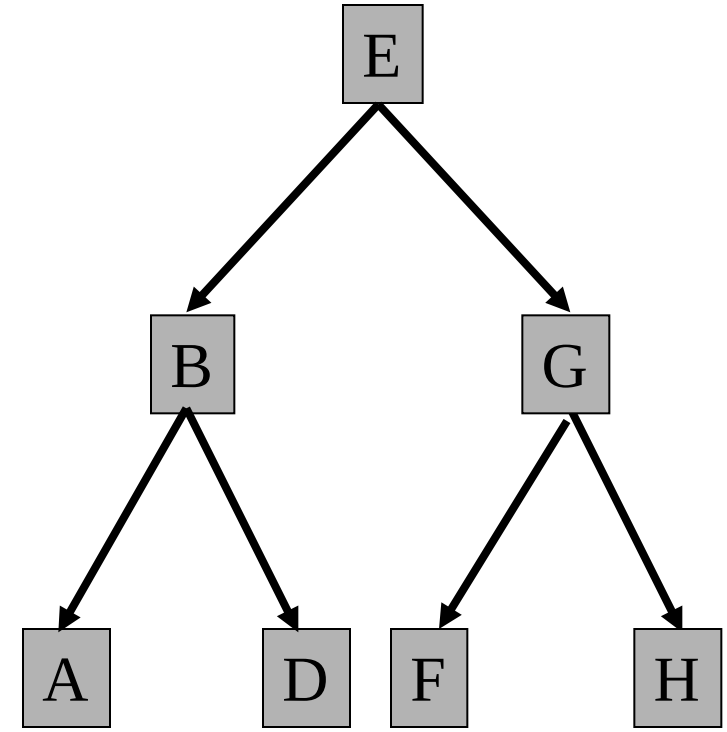


Binary Search Tree



Binary Search Trees (BSTs)

- Each node has 0, 1 or 2 children
  - Called left child and right child
  - Each node has a key that is used to organise the tree in a sorted way
    - » Keys must be unique - used for search
  - Nodes also contain a value (unshown)



- Sorting is 'horizontal':
  - Every left child's key is smaller than the key of its parent
  - Every right child's key is larger than the key of its parent

Java Tree Implementation in Memory

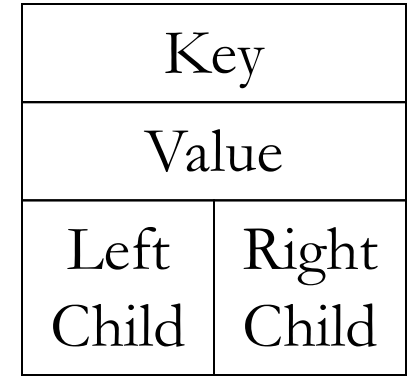
- Trees are put together much like a doubly linked list
  - *i.e.*, a set of nodes connected by references (pointers)
  - Each node has two references: one to the left child and one to the right child
    - » Similar to a linked list's "next" and "previous" reference,
    - » Each node also contains a key and a value
    - » Key is for finding, value is the data associated with that key
  - The tree itself is then a reference to the root node
    - » … just like a linked list is simply a reference to the head node
    - » Note that there is little point in holding a ref to the leaf nodes because there are many of them (unlike tail in linked lists)

TreeNode Class

- Key is usually a String or an int
  - *e.g.*, Name or Student ID are useful keys (must be unique)
- Value should be an Object (or use Generics)

```
CLASS TreeNode
FIELDS: key, value, leftChild, rightChild

CONSTRUCTOR TreeNode IMPORTS inKey, inValue

ACCESSOR getKey IMPORTS NONE EXPORTS key

ACCESSOR getValue IMPORTS NONE EXPORTS value
```

<Continues on next page>



TreeNode Class

<Continued from previous page>

```
ACCESSOR getLeft IMPORTS NONE EXPORTS leftChild
MUTATOR setLeft IMPORTS newLeft EXPORTS NONE
   leftChild ← newLeft                    ← Null is a valid child – indicates 'no child'

ACCESSOR getRight IMPORTS NONE EXPORTS rightChild
MUTATOR setRight IMPORTS newRight EXPORTS NONE
   rightChild ← newRight                  ← Null is a valid child – indicates 'no child'
```

TreeNode Class in Java

```
private class TreeNode { // Put inside Tree class
   private String m_key;
   private Object m_value;
   private TreeNode m_ leftChild;
   private TreeNode m_ rightChild;

   public TreeNode(String inKey, Object inVal) {
      if (inKey == null)
         throw new IllegalArgumentException("Key cannot be null");
      m_key = inKey;                                  //consider “owning” the key
      m_value = inVal;
      m_rightChild = null;
      m_leftChild = null;
   }
   public String getKey() { return m_key; }
   public Object getValue() { return m_value; }
   public TreeNode getLeft() { return m_leftChild; }
   public void setLeft(TreeNode newLeft) { m_leftChild = newLeft; }
   public TreeNode getRight() { return m_rightChild; }
   public void setRight(TreeNode newRight) { m_rightChild = newRight; }
 }
```

BinarySearchTree Class

```
public class BinarySearchTree {
   private class TreeNode
   {
     ...                  // From previous slide
   }

   private TreeNode m_root;

   public BinarySearchTree() {
      m_root = null;  // Start with an empty tree
   }
//wrapper methods, will call private recursive implementations
   public Object find(String key) { ... }
   public void insert(String key, Object value) { ... }
   public void delete(String key) { ... }

   public int height() { ... }
}
```

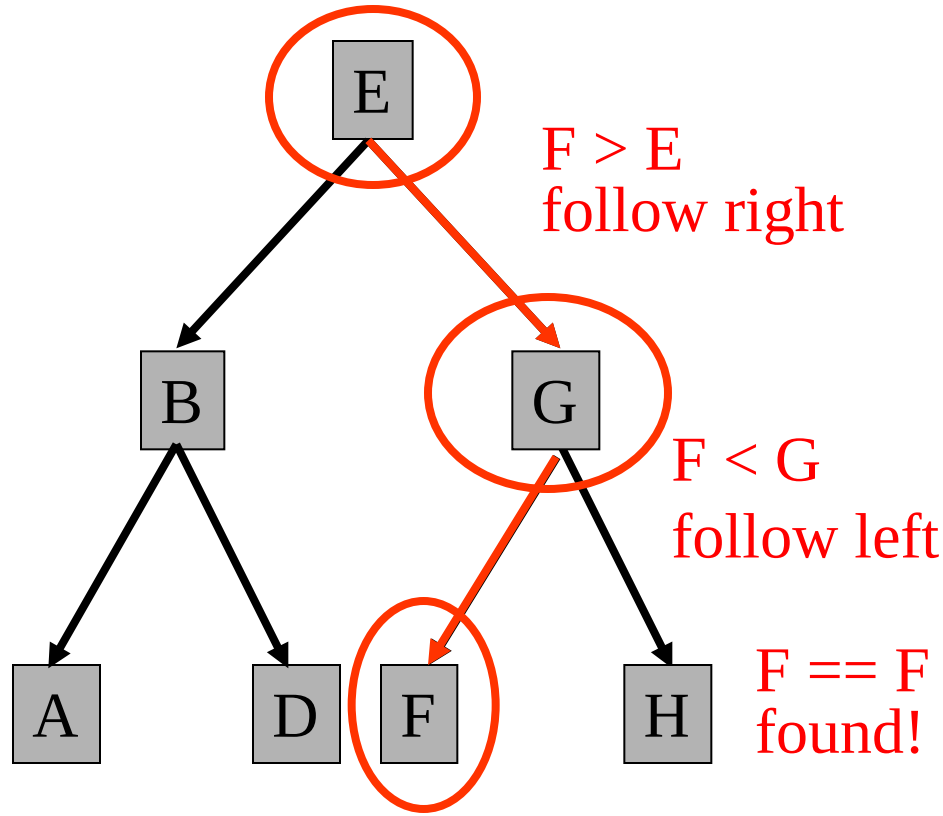
Finding in Binary Search Trees

Binary search trees are fast at finding keys. Steps:

1. Start at root node
2. Compare current node's key with key to find
   - If keys are equal, the key is found – return node's value
   - If key to find < node key, go to left child node
   - If key to find > node key, go to right child node
3. If no child exists in the chosen branch (ie: we are at a null child), then the tree doesn't contain that key
4. Otherwise repeat Steps 2-4 with the child node

Find Example

– find("F") :



Find – Time Complexity

- find() in a BST is very fast as a consequence of the tree structure (we'll examine Big-O a little later on)
  - If we decide to go down the left branch, all nodes in the right branch are no longer considered since they are too large to possibly be the key we are looking for
    - » And vice-versa if we take the right-child branch
  - In a (well-maintained) tree this will halve the amount of nodes we consider *every time* we move down a branch
  - In fact, we end up only checking at most one node for every level in the tree (less if item is higher than a leaf node)

Find Java Code - Recursive

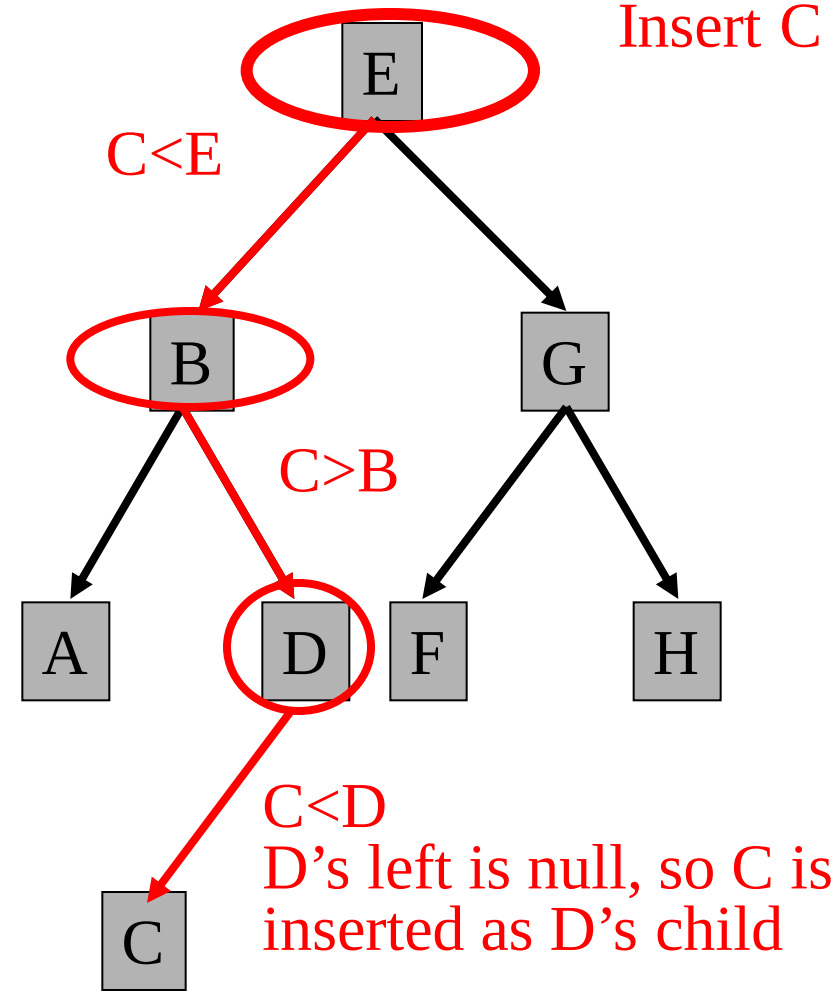
```
public Object find(String key)
{
    return findRecursive(key, m_root);
}

private Object findRecursive(String key, TreeNode currNode)
{
   Object val = null;

   if (currNode == null)
      throw new NoSuchElementException("Key " + key + " not found");
                                                    // Base case: not found
   else if (key.equals(currNode.getKey()))
      val = currNode.getValue();                    // Base case: found

   else if (key.compareTo(currNode.getKey()) < 0)
      val = findRecursive(key, currNode.getLeft());  // Go left (recursive)

   else
      val = findRecursive(key, currNode.getRight());  // Go right (recursive)

   return val;
}
```

Insert

- insert() is a lot like find(), except we are looking for the place to add a new node
  - The algorithm is almost identical except that we won't find the key (since it is being inserted)
    - » If we do find the key, insert must be aborted since we can't insert the same key twice (keys must be unique)
  - Instead, when we run out of nodes to search, we add the new node as the appropriate child of the last node
- Insert *always* adds the node to the bottom of a branch of the tree
  - In other words, it creates a new leaf node
  - Trying to insert it into the middle of the tree is too messy: hard to guarantee the sorting order is maintained

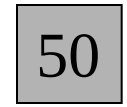
Insert Example

- Traverse the structure until you reach a null child
  - Checking for duplicate keys as you go!
- Then add the new value as a leaf node where the null child was.



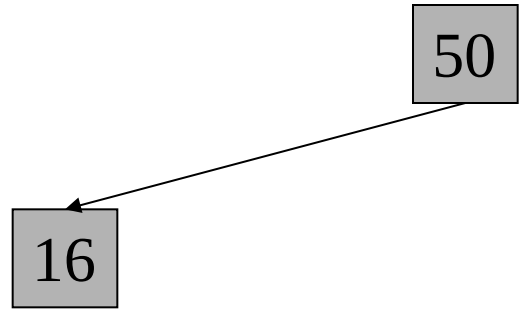
Insert: Worked Example

- Let's insert the following keys into a binary search tree in the given order:
  - **50**, 16, 7, 89, 70, 45, 10, 66, 95



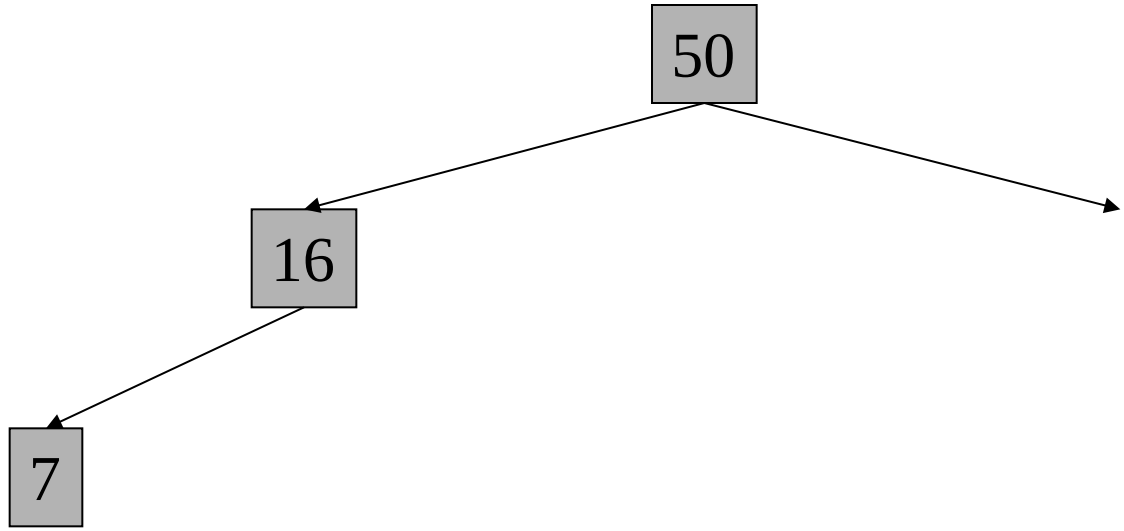
Insert: Worked Example

- Let's insert the following keys into a binary search tree in the given order:
  - 50, **16**, 7, 89, 70, 45, 10, 66, 95



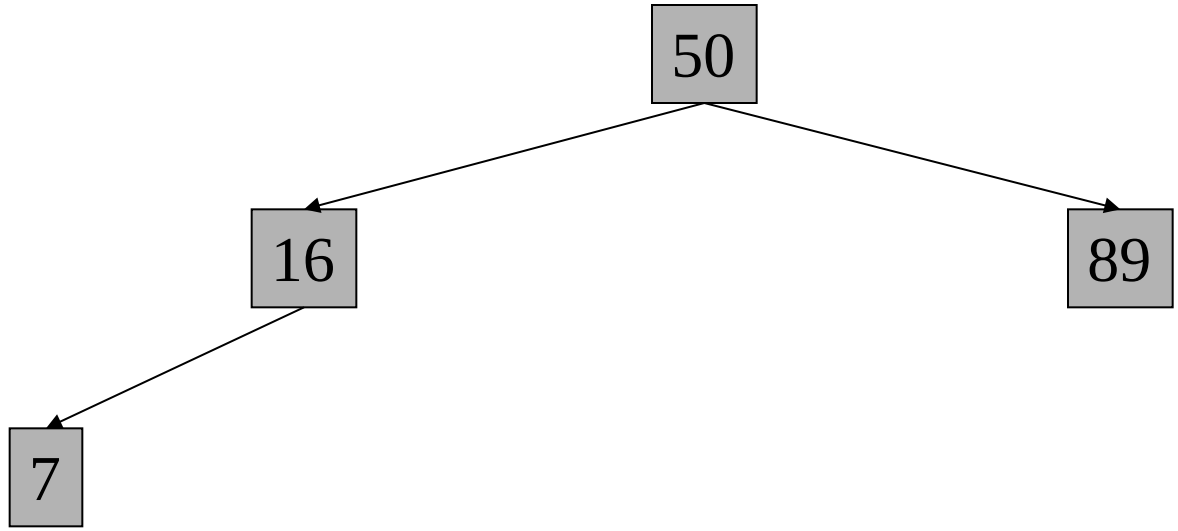
Insert: Worked Example

- Let's insert the following keys into a binary search tree in the given order:
  - 50, 16, 7, 89, 70, 45, 10, 66, 95



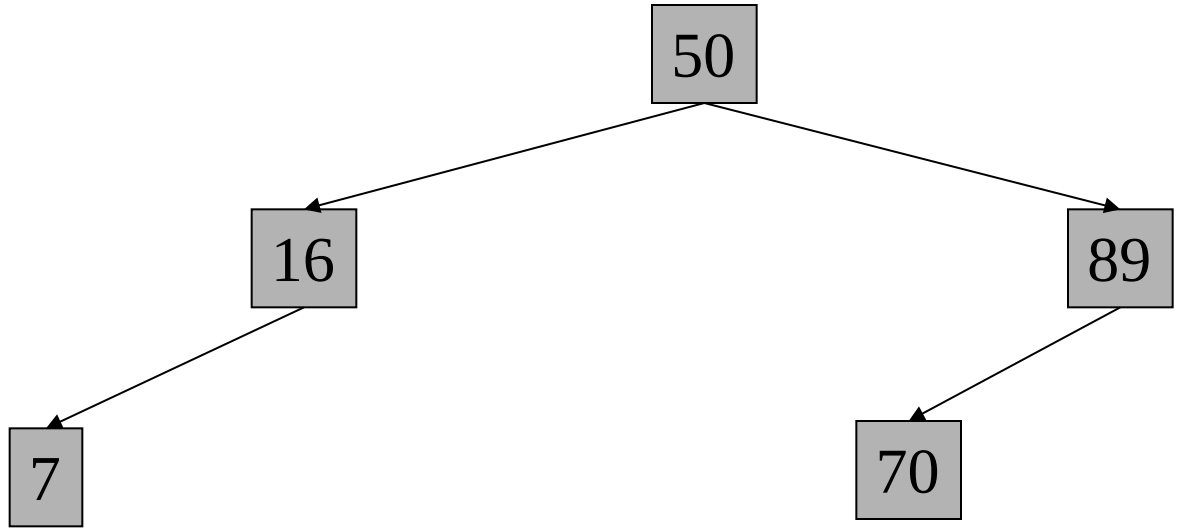
Insert: Worked Example

- Let's insert the following keys into a binary search tree in the given order:
  - 50, 16, 7, **89**, 70, 45, 10, 66, 95



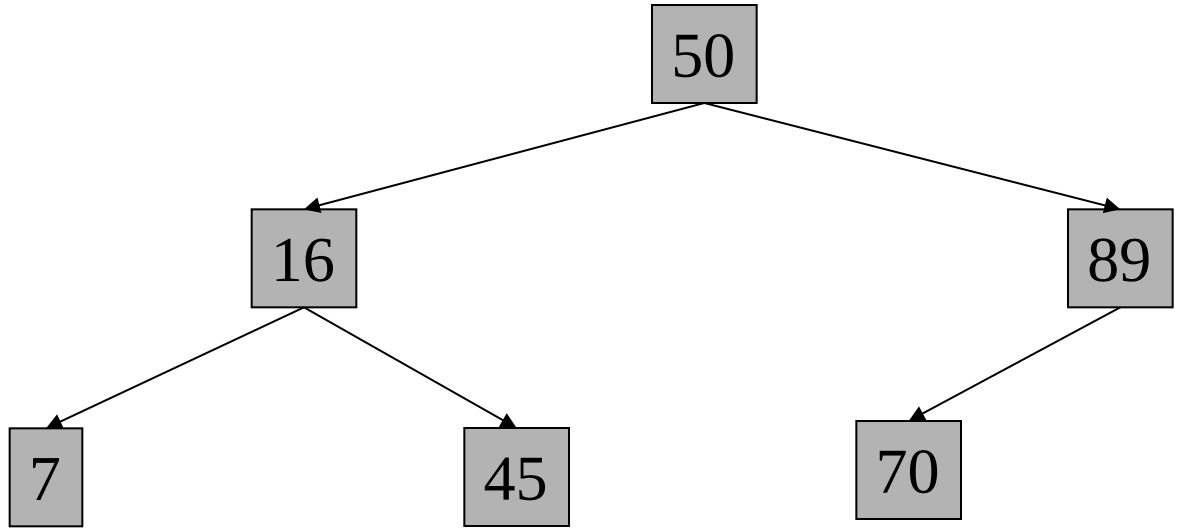
Insert: Worked Example

- Let's insert the following keys into a binary search tree in the given order:
  - 50, 16, 7, 89, **70**, 45, 10, 66, 95



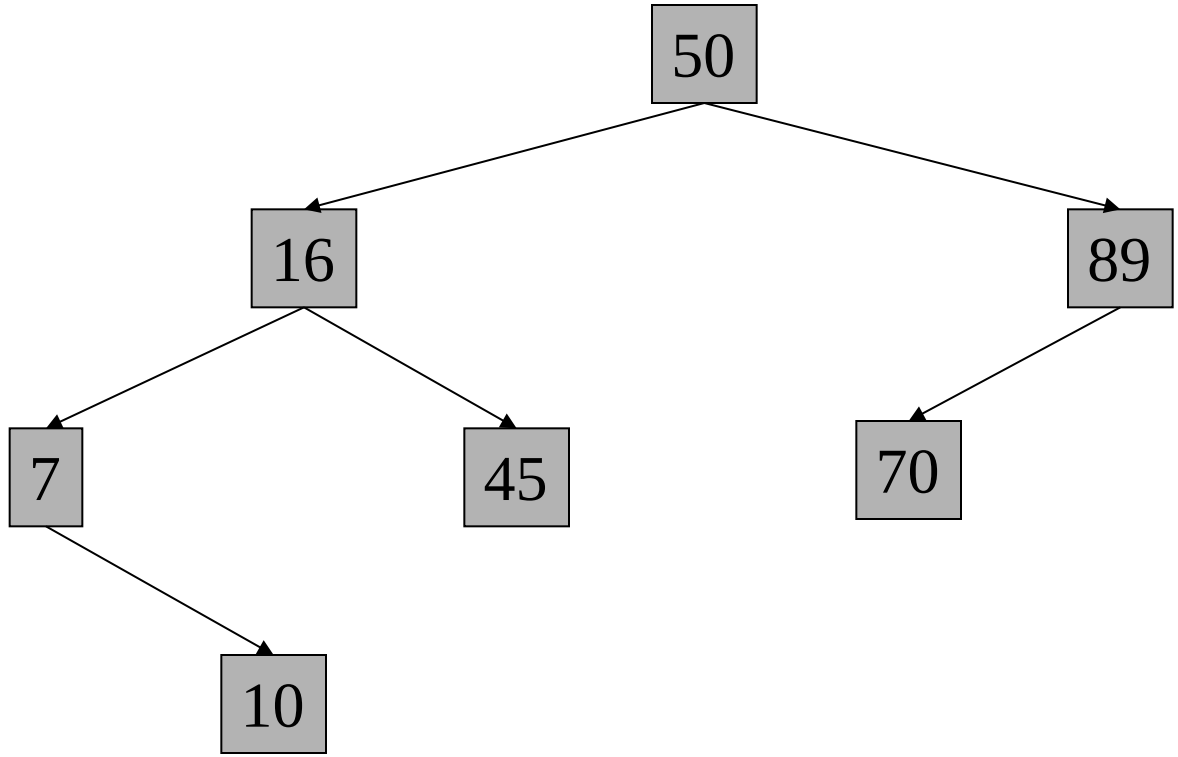
Insert: Worked Example

- Let's insert the following keys into a binary search tree in the given order:
  - 50, 16, 7, 89, 70, **45**, 10, 66, 95



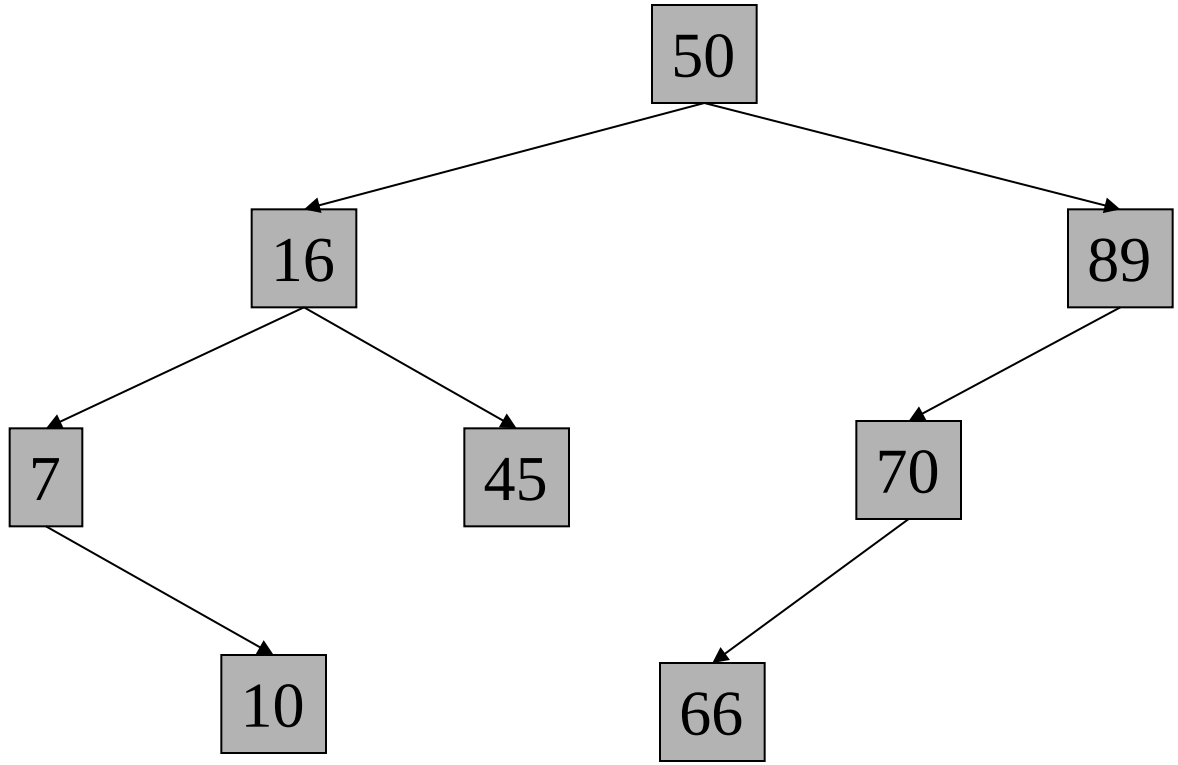
Insert: Worked Example

- Let's insert the following keys into a binary search tree in the given order:
  - 50, 16, 7, 89, 70, 45, **10**, 66, 95



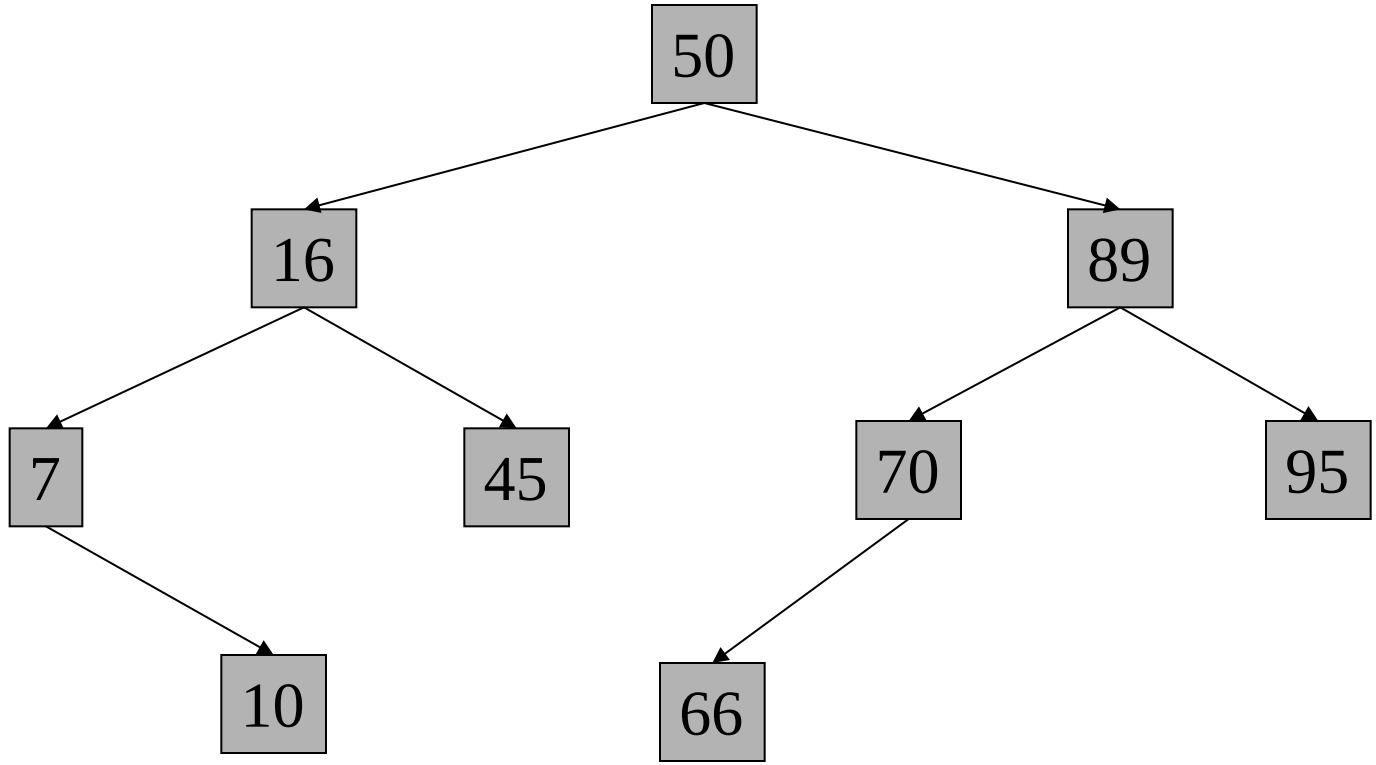
Insert: Worked Example

- Let's insert the following keys into a binary search tree in the given order:
  - 50, 16, 7, 89, 70, 45, 10, **66**, 95



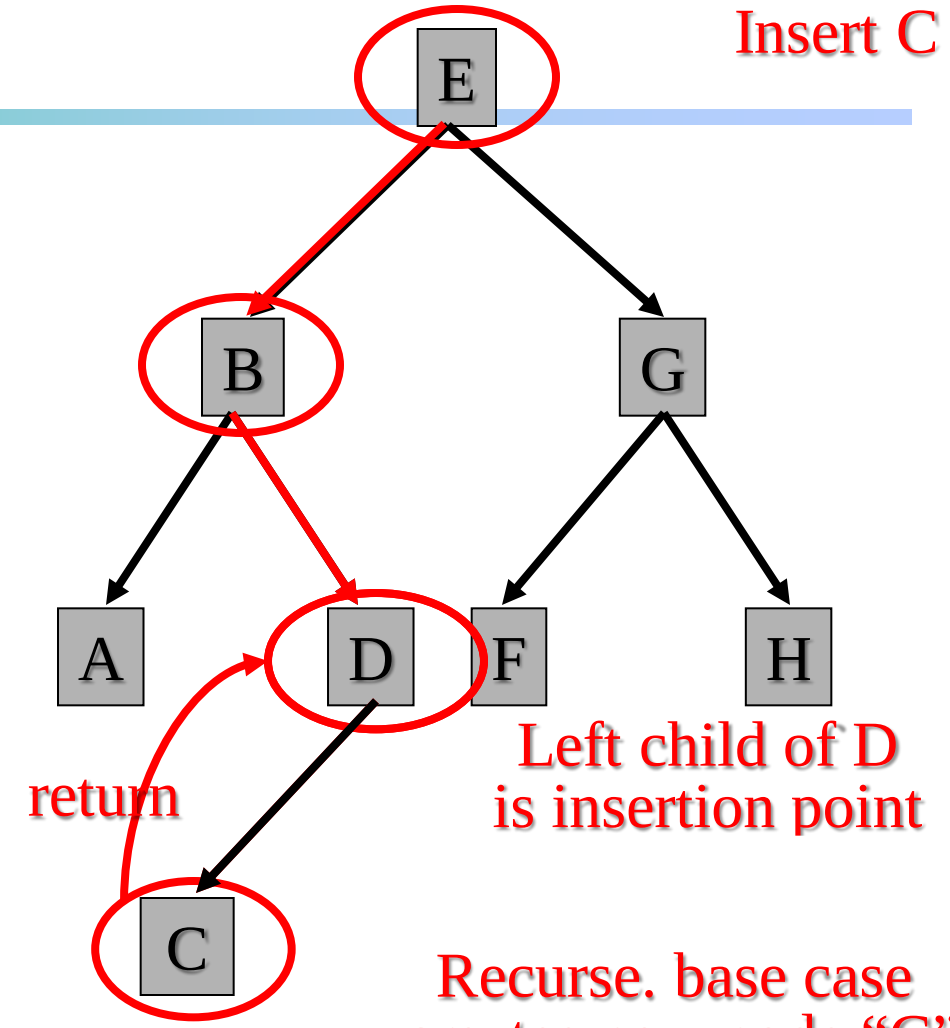
Insert: Worked Example

- Let's insert the following keys into a binary search tree in the given order:
  - 50, 16, 7, 89, 70, 45, 10, 66, **95**



Recursive insert

- Recursively find null insertion point.
- Final recursive call following correct null branch
- Base case stops recursion and creates a new node "C"
- Left link of the insertion point is corrected as the recursion unwinds.
- In all other cases, old child is returned



Insert Algorithm

- Needs wrapper method.

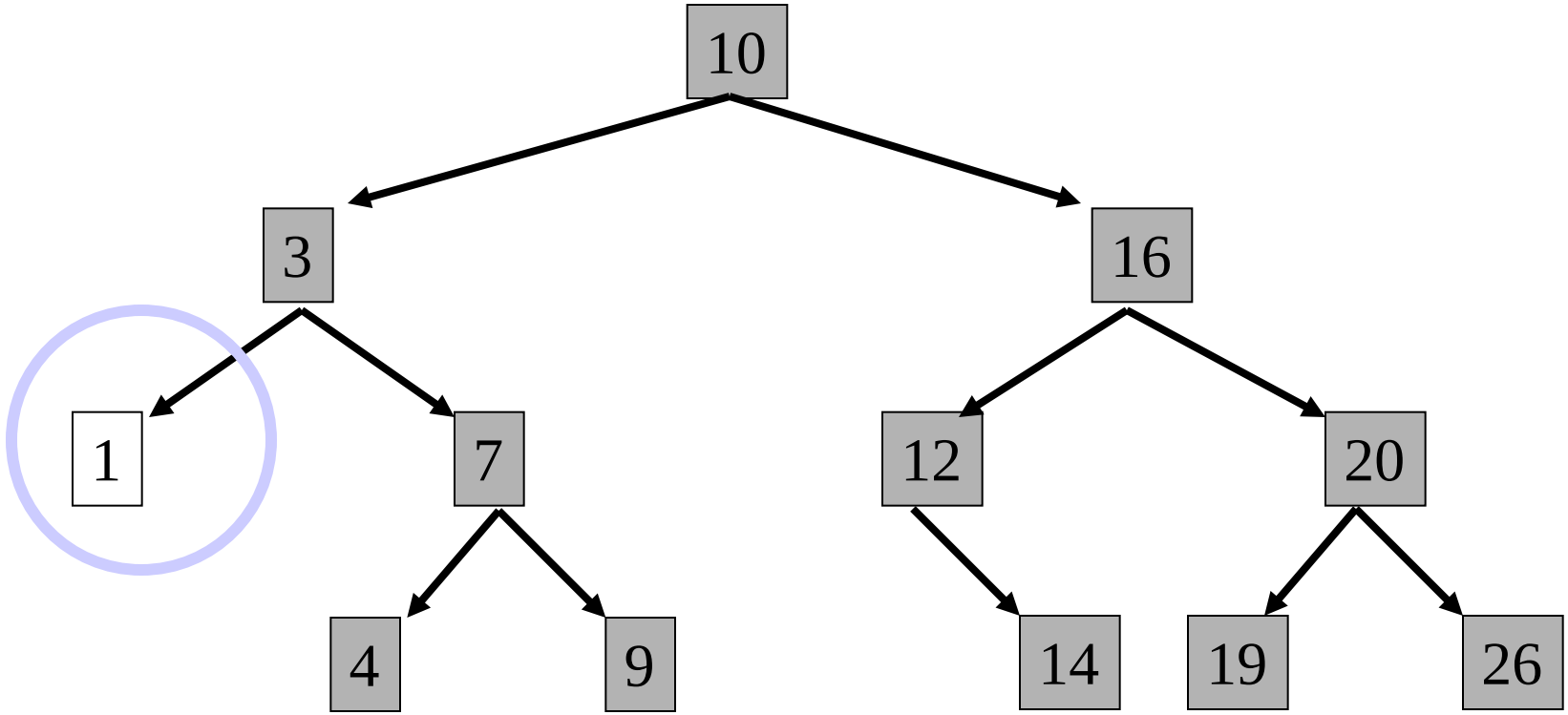
```
insertRec
   import key, data, cur
   export upDateNode

   upDateNode = cur
   IF cur == null                                  // base case - found
      create newNode <- key, data                  // insertion point
      upDateNode = newNode
   ELSE IF key.equals <- cur.getKey                // in the tree
      abort
   ELSE IF key.compareTo <- cur.getKey < 0
      cur.setLeft <- insertRec <- key, data, cur.getLeft  //recurse left
   ELSE                                                   //update current
      cur.setRight <- insertRec <- key, data, cur.getRight //recurse right
   ENDIFS
```

Delete

- Removing a node from the tree is just a matter of pointer operations, like remove() in linked lists
- However we need to make sure it is still a binary search tree after deletion
  - When the deleted node has children, this is not so easy
- Thus there are three cases when deleting a node:
  - The node to be deleted has no children
  - The node to be deleted has one child
  - The node to be deleted has two children
- In all cases, we first have to find the node to delete!

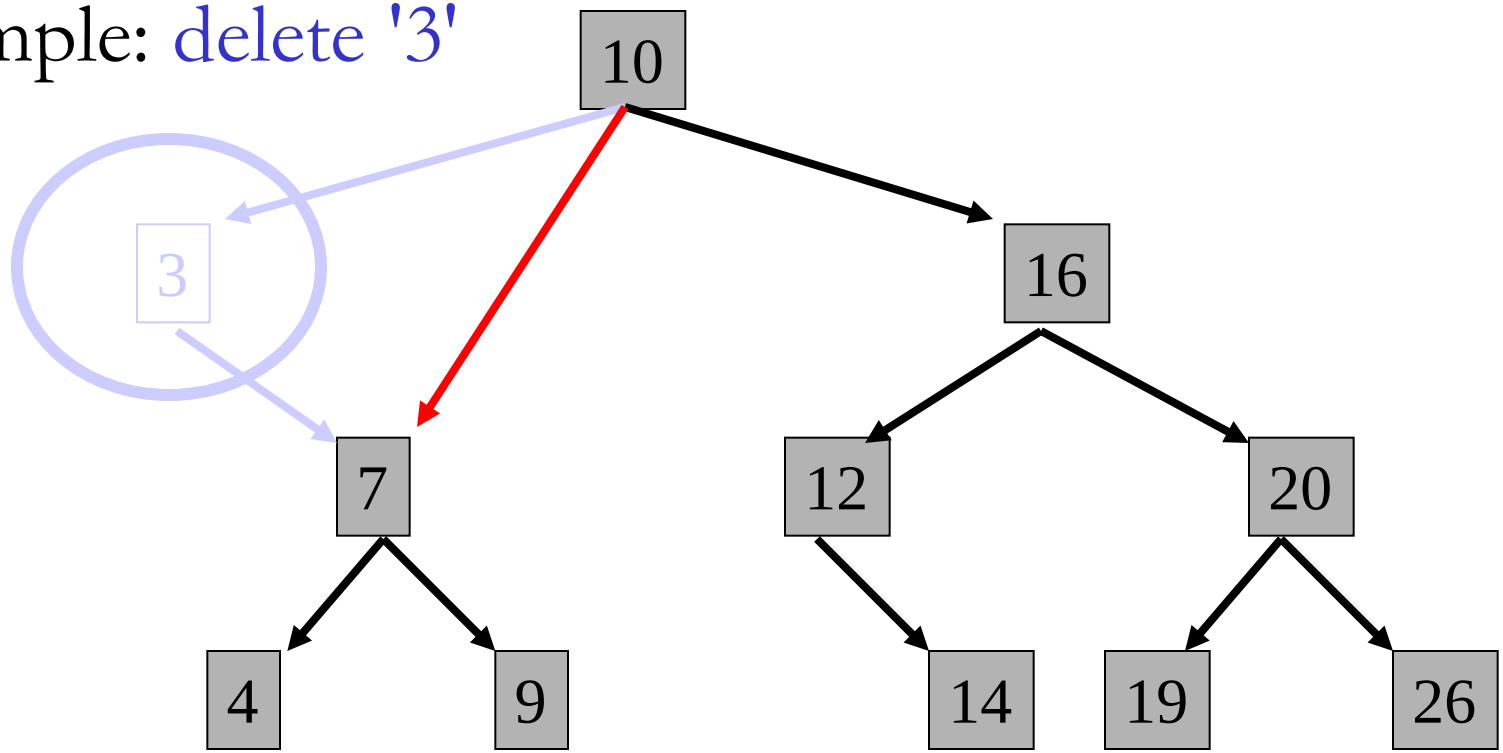
Delete Example – No Children

- Approach: Find node to delete. If no children (*i.e.*, leaf node), set the parent's link to null
- Example: delete '1'



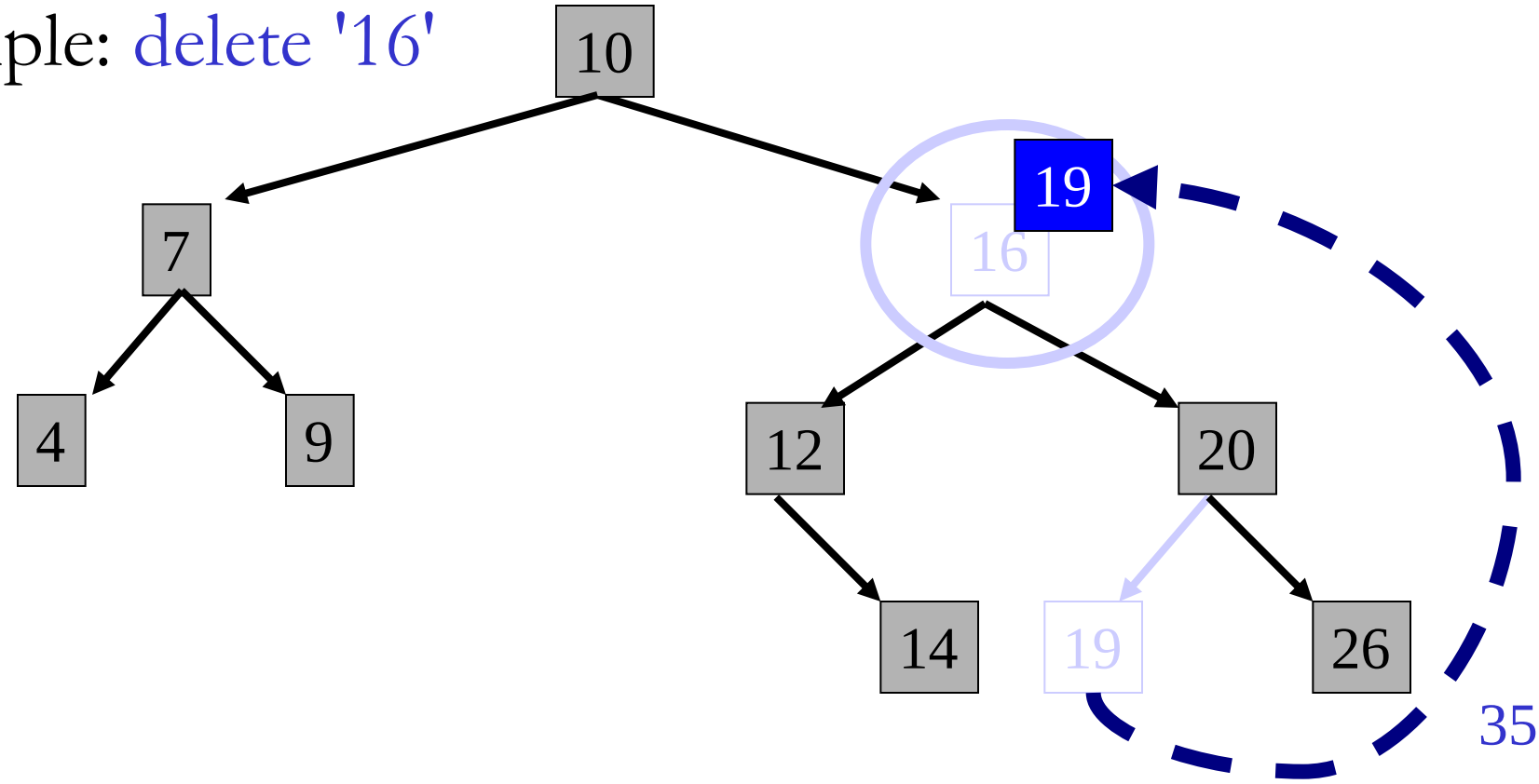
Delete Example – One Child

- Approach: Find node to delete. If only one child, have this 'orphan' become 'adopted' by the parent
  - *i.e.*, bypass the deleted node, just like linked list remove()
- Example: delete '3'



Delete Example – Two Children

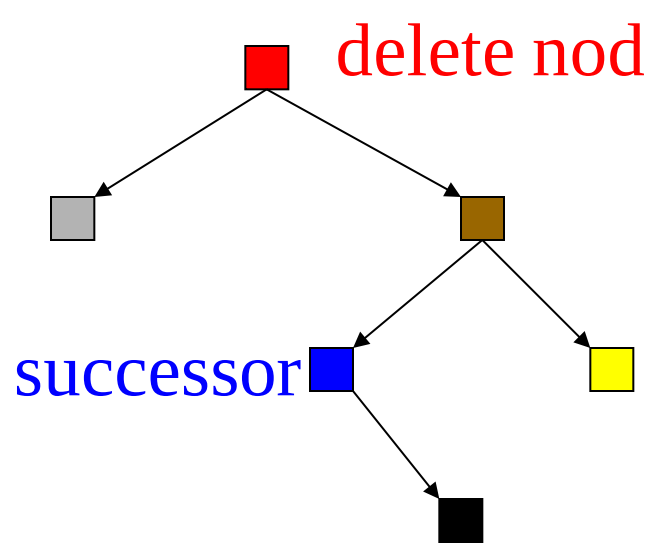
- Approach: Find node to delete. Then find 'successor'
  - Successor = next largest node = left-most of right sub-tree
  - Then replace the node to delete with the successor node
- Example: delete '16'



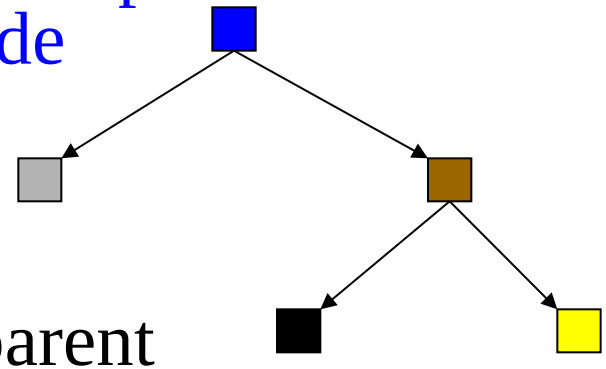
Successor Node

- Successor is chosen so as to guarantee that the tree will remain a binary search tree
  - Because it is the next larger than the deleted node, it will be larger than everything in the left sub-tree
  - And since it is also the smallest from the right sub-tree, it is smaller than all nodes on the right
  - Hence BST properties are maintained, without the need for complicated re-shuffling of the tree
- NOTE: Since the successor is being removed from its original location, if it has a right child we must do a 'grandparent adopt' for the successor

Successor Node Handling



Before delete



successor's parent
adopts successor's right child.

After delete

Recursive delete strategy

- First part is similar to recursive find since we must find the delete node.
- When a copy of the delete method returns, it returns a reference to the node found at that level of recursion.
- This is always used to reset the link in the parent as the recursion unwinds.
- Note that when we've found the delete node, we must return the replacement node or null, as appropriate!

Recursive delete base cases

- Stop recursion if the key to be deleted isn't in the tree.
  - This will be when we've walked down the correct branches all the way to a null reference.
- Stop recursion when the delete node has been determined and after the replacement node has been identified and descendent links patched.
  - At this point, reset links as recursion unwinds and go back up the tree.

Recursive Delete Algorithm

```
deleteRec
import key, cur
export updateNode

   updateNode = cur
   IF cur == null
      abort                                          //not in the tree
   ELSE IF key.equals <- cur.getKey
      updateNode = deleteNode <- key, cur            // base case - found
   ELSE IF  key.compareTo <- cur.getKey < 0
      cur.setLeft <- deleteRec <- key, cur.getLeft   //recurse left
   ELSE                                              //similar to insert
      cur.setRight <- deleteRec <- key, cur.getRight //recurse right
   ENDIFS
```

deleteNode

```
import key, delNode
export updateNode

    updateNode = null
   IF delNode.getLeft == null AND delNode.getRight == null
      upDateNode = null              //no children
   ELSE IF delNode.getLeft NOT null AND delNode.getRight == null
      upDateNode = delNode.getLeft  //one child - left
   ELSE IF delNode.getLeft == null AND delNode.getRight NOT null
      upDateNode = delNode.getRight  //one child - right
   ELSE                               //two children
      ... (next slide)
```

deleteNode continued

```
ELSE                                                    //two children
   upDateNode = promoteSuccessor <- delNode.getRight
   IF upDateNode NOT delNode.getRight                   //no cycles
      upDateNode.setRight <- delNode.getRight           //update right
   ENDIF
   upDateNode.setLeft <- delNode.getLeft                //and left
ENDIFS
```

promoteSuccessor

```
import: cur
export: successor
Assertion: successor will be the left most child
           of the right subtree

   successor = cur
// IF cur.getLeft == null          //not needed in code
//    successor = cur              //base case – no left children
// ELSE
   IF cur.getLeft NOT null
      successor = promoteSuccessor <- cur.getLeft
      IF successor == cur.getLeft          //parent of successor
         cur.setLeft <- successor.getRight //needs to adopt right child
      ENDIF
   ENDIF
```

Types of Trees

- When it comes to evaluating the efficiency of a tree it is necessary to know its *precise* structure
  - This wasn't an issue with linked lists or arrays since they are one-dimensional: N is the only variable
    - » *e.g.*, as soon as next = null, that's the end of the linked list
  - With trees, each node can have zero, one or two children
    - » So left = null only stops the tree in that direction, but there are other directions that it might still be 'growing'
  - *e.g.*, are all nodes in all levels 'filled up'? Only a few? *etc.*
- Obviously there are *lots* of possible configurations
  - Thus we must focus on the important cases

Types of Trees

- There are three binary tree classifications that we will explore in more detail:
  - Complete binary tree
  - Almost-complete binary tree
  - Degenerate binary tree
- Other classifications exist, *e.g.*,
  - Strictly binary tree: all nodes have either zero or two children
  - Balanced binary tree: there are roughly the same number of branches on either side of all nodes in the tree
    - » We will look at these in more detail in a later lecture

Complete Trees

– A complete binary tree is one where all levels in the tree are fully filled with nodes

- *i.e.*, with L levels, it contains $N = 2^L-1$ nodes
- The example to the left contains L=3 and $N = 2^L-1 = 7$ nodes
- We can also invert this equation:

$N = 2^L-1$

$N+1 = 2^L$

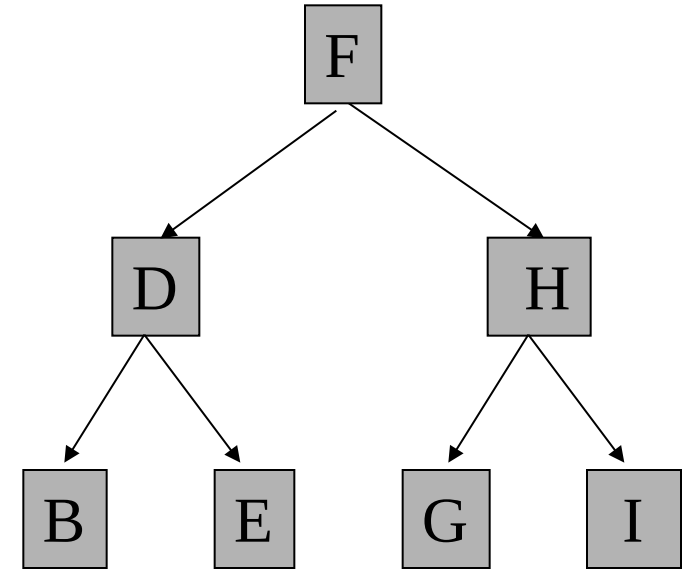
$\log_2(N+1) = \log_2 2^L$

$\log_2(N+1) = L \times \log_2 2$

$\log_2(N+1) = L \times 1$

$L = \log_2(N+1)$

$\mathbf{L \approx \log_2 N}$

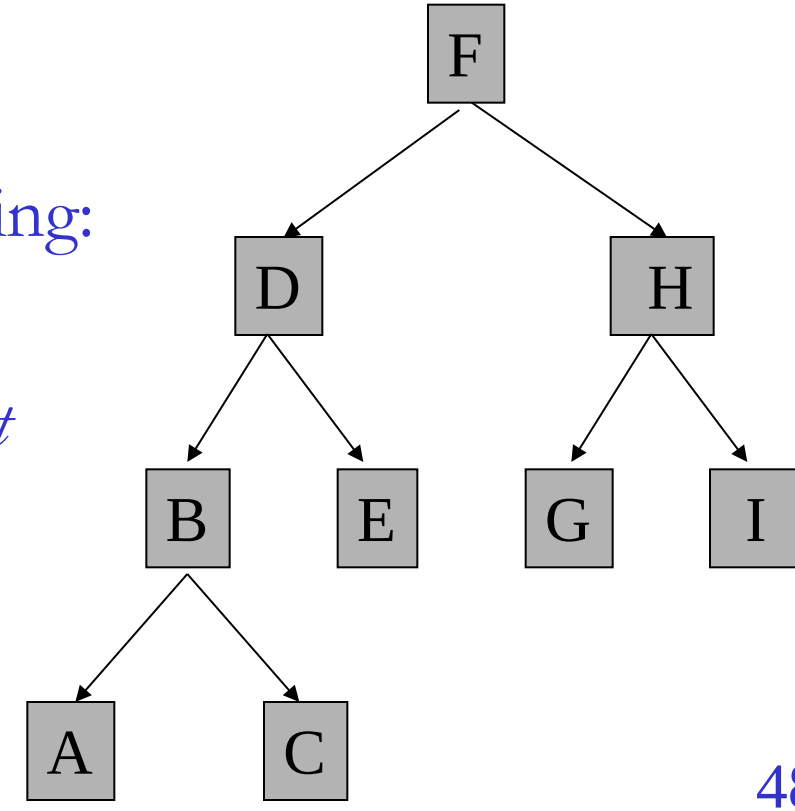


Significance of Complete Trees

- Complete trees are important since they have a very consistent structure
  - All nodes have two children except for the last level, which have zero children
- A complete tree is the tree that fits the most nodes into the fewest levels
  - This optimal compactness also means optimal efficiency in terms of time complexity and (to a lesser extent) space usage

Almost-Complete Trees

- However, we can't always have exactly enough nodes to totally fill a tree
  - What if we had 9 nodes instead of 7?
  - 9 won't fit exactly into 2^L-1
  - Thus we try for the next-best-thing: an almost-complete tree
  - Structure: every level is full *except* the last level, which is filled from left to right



Almost-Complete Trees

- So an almost-complete tree is just like a complete tree except it has too few nodes to fill in the last level
  - Thus its time complexity is also almost identical to a complete tree, so we will analyse them together
- Note: That last bit of the rule is strange: why left-to-right? Why not just allow any nodes in the last level?
  - This would still be just as efficient, so why the distinction?
  - The left-to-right structure turns out to be very useful when storing binary trees in an array (*e.g.*, Heaps)
    - » Yes, not much of a reason. But that's the reason.

Time Complexity: (Almost)-Complete Trees

- Three main operations on binary search trees:
  - Find
  - Insert
  - Delete
- The following Big-O time complexity analysis will assume that we have a complete or almost-complete tree
  - We will analyse other tree structures later

Almost-Complete Find – Big-O Analysis

- Best case: the root is the key we want to find
  - Root is first node checked, so O(1)
- Worst case: The key is a leaf node on the last level
  - A complete tree has $L = \log_2(N+1)$ levels, thus: O(log N)
  - Almost-complete trees have just one extra level at $1+\log_2(N+1)$, so its Big-O time complexity is identical
- Average case: The key is 2nd bottom level
  - $\log_2(N+1)$ -1 levels to get through = O(log N)
  - Again, complete/almost-complete have identical Big-O

Almost-Complete Insert – Big-O Analysis

- Best case: We must always insert at a null child, and only the leaf nodes (at the last level in a complete tree) have a null child
  - Hence must traverse to the last level: O(log N)
  - Almost-complete tree can be last or second-last level, which in Big-O is the same thing
- Worst case: Same as the best case: O(log N)
- Average case: Same: O(log N)

Almost-Complete Delete – Big-O Analysis

- Delete is a combination of finding the node and then promoting its successor
- Best, Worst and Average cases: All the same O(log N)
  - No children – leaf nodes at the bottom of the tree.
  - One child – must be at 2nd last level of the tree.
  - Two children – no matter where the node is, the successor must be at the bottom level.

Review of Complexity Scaling

- O(log N) scales *much* better than O(N)
  - Just like O(N log N) scaled much better than $O(N^2)$ in sorting

$\log_2$ N	N
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1,024
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536

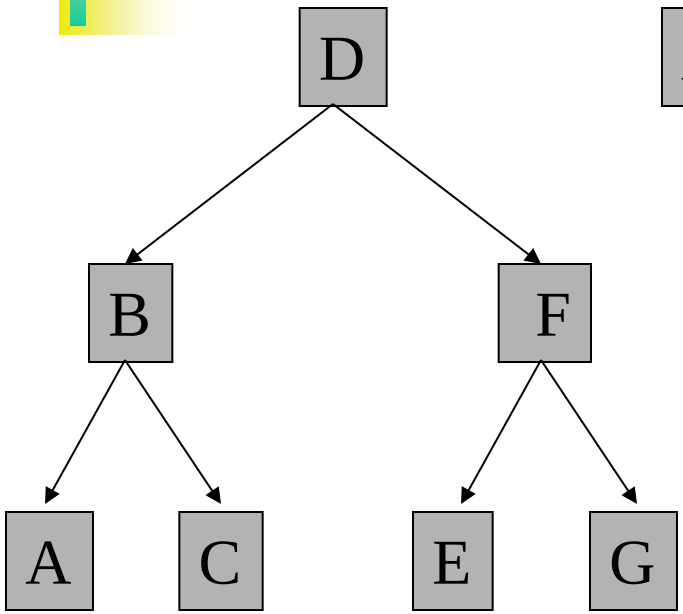
Balanced vs Unbalanced Trees

- Trees don't just become almost-complete by default
- In fact, it is very difficult to ensure a tree remains almost-complete
  - There is nothing in insert or delete to keep a BST almost-complete, or even sort-of-almost-complete!
  - In fact, keeping a BST roughly 'balanced' is the best we can do, and those are such complicated algorithms that we must wait to cover them in their own lecture
- This is a pity because almost-complete trees are the most efficient

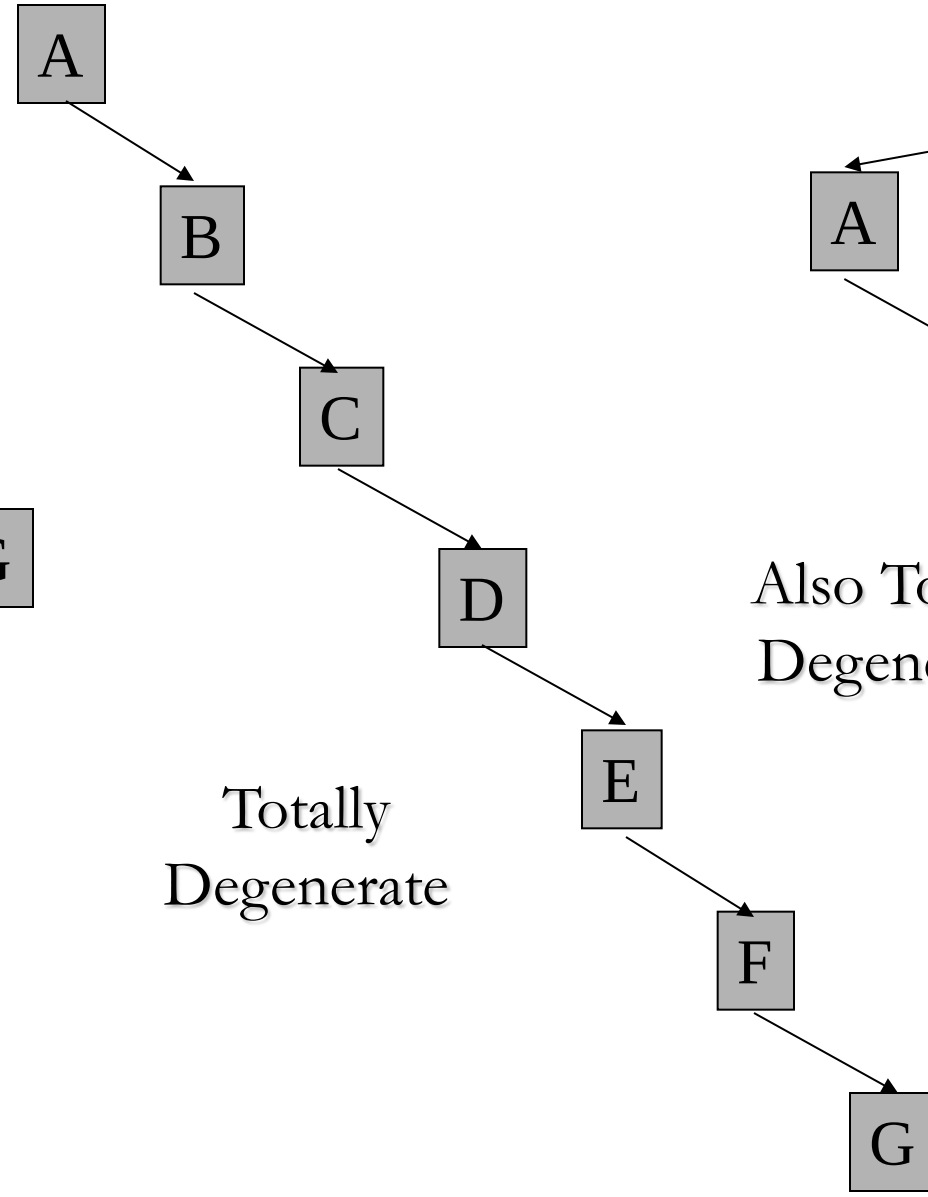
Degenerate Trees

- Let's take a look at the other extreme, and examine a tree that is the most *in*efficient
- Consider a tree where every node has only one child
  - Hence no branching occurs, and we cannot avoid visiting every node in turn when searching
- This is a totally degenerate tree
  - Such a tree is in fact essentially a linked list, and thus has the same time complexity as a linked list
    - » But with more memory space usage since one of the child pointers remains unused per node

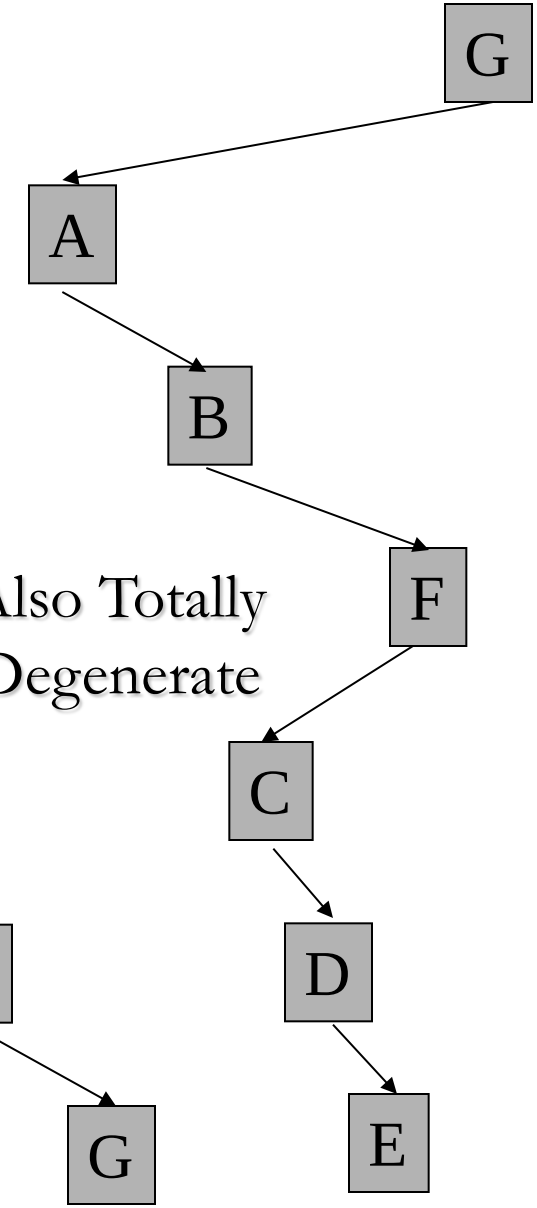
Complete vs Degenerate Trees



Complete



Totally Degenerate



Also Totally Degenerate

Degenerate Trees

- Degenerate trees can easily arise from a very common scenario: inserting items whose keys are already in sorted order into a binary search tree
  - If ascending order, every new item will be inserted to the right of the previous item, thus creating a linked list
- If a tree becomes unbalanced, it could be 'rebuilt'
  - An alternative is to partially rebuild the tree every time an item is inserted – we will explore this in a later lecture
- Interestingly, inserting items in a random order usually leads to reasonably balanced trees
  - Doesn't help once you start deleting though!

Degenerate Trees – Big-O Analysis

- Since a fully-degenerate tree is simply a linked list, all time complexities will be like a linked list:
  - Find: Best = O(1), Average = O(N), Worst = O(N)
  - Insert: Best = O(1), Average = O(N), Worst = O(N)
  - Delete: Best = O(1), Average = O(N), Worst = O(N)
- Although the O(1)'s for best insert and delete look good compared to a complete tree's O(log N) for the same, it's the average/worst cases that are important
  - And the O(log N) complete tree is *much* more scalable than O(N) degenerate trees (as we saw earlier)

Other Tree Structures

- So what about other tree structures?
  - They will exist somewhere between a complete tree's O(log N) and a totally-degenerate tree's O(N)
    - » Without knowing the exact structure it is not possible to specify Big-O notation for them
  - Thus definitions like 'balanced tree' are concerned with keeping the tree as close to a complete tree as possible
    - » 'Balanced' is usually defined to mean that the deepest leaf node is not much further from the root than the shallowest leaf node
      - Where 'not much further' is usually 3 or 4 levels
      - Red Black trees specify 3 levels

Tree Traversal

- What if we wanted to get *all* nodes in the tree?
- Three common traversal methods (others exist, but aren't so easy to implement)
  - In-order traversal
  - Pre-order traversal
  - Post-order traversal
- These are quite easy to implement recursively
  - All that differs between the three is the order in which you visit nodes
  - Which one you use depends on what you need to do

In-order Traversal

- Traverse the tree in ascending sorted order (assuming the tree is a binary search tree!)
  - *i.e.*, sweep from left-to-right
- Recursive traversal algorithm (starting at root):
  - First recurse to traverse left-child's branch
  - Then visit current node
  - Then recurse to traverse right-child's branch
- Good for recovering elements in sorted order from a binary search tree
  - Not quite as useful for other types of binary tree

Pre-order Traversal

- Visits nodes in a 'parents-first' order
- Recursive algorithm (starting at root):
  - First visit current node
  - Then recurse to traverse left-child's branch
  - Then recurse to traverse right-child's branch
- Useful for extracting a list that, when re-inserted in that order, recreates the original tree
  - In fact, any list extracted in which the parents come before the children will recreate the tree

Post-order Traversal

- Visits nodes in a 'children-first' order
- Recursive algorithm (starting at root):
  - First recurse to traverse left-child's branch
  - Then recurse to traverse right-child's branch
  - Then visit current node
- Has some uses, but not so obvious
  - One application is to build postfix equation expressions using 'expression trees' to represent the equation

Traversal

```
Method: traverseTree
Import: curNode
Export: nothing
Assertion: every node will be visited

IF curNode NOT null
   doPrefixStuff
   traverseTree <- curNode.getLeft
   doInFixStuff
   traverseTree <- curNode.getRight
   doPostFixStuff
ENDIF
```

More Code

- Finally, let's take a look at a few commonly-needed tasks that involve moving through the tree:
  - Finding the value of the minimum key in the tree
  - Finding the height of the tree (*i.e.*, from root to furthest leaf)

Tree Min Node – Recursive

```
public TreeNode min(TreeNode startNode)
{
   TreeNode minNode;
   if (startNode.getLeft() != null)           // not base case
      minNode = min(startNode.getLeft());     // Recursive call
   return minNode;
}
```

– max() would be simply working along the right children instead of the left

- Remember: it is the min/max *key*, not value

Tree Height – Recursive (BAD!!!)

```
public int height()
{
   return height(m_root, 0);
}

private int height(TreeNode startNode, int htSoFar)
{
   int iLeftHt, iRightHt;

   if (startNode == null)
        return htSoFar; // Base case – no more along this branch
   else {
         iLeftHt = height(startNode.getLeft(), htSoFar+1);   // Recurse
         iRightHt = height(startNode.getRight(), htSoFar+1); // Recurse

         // Return highest of left vs right branches
         if (iLeftHt > iRightHt)
            return iLeftHt;
         else
            return iRightHt;
   }
}
```

Tree Height – Another Recursive

```
public int height()
{
   return height(m_root);
}

public int heightRec(TreeNode curNode)
{
   int htSoFar, iLeftHt, iRightHt;

   if (curNode == null)
      htSoFar = -1;            // Base case – no more along this branch
   else
   {
      iLeftHt = heightRec(curNode.getLeft());   // Calc left height from here
      iRightHt = heightRec(curNode.getRight()); // Calc right height from here

      // Get highest of left vs right branches
      if (iLeftHt > iRightHt)
         htSoFar = iLeftHt + 1;
      else
         htSoFar = iRightHt + 1;
   }
   return htSoFar;
}
```

Tree Height

- Tree height is very difficult to do iteratively!
  - We need to search *all* branches, so we must remember where every branch occurred and backtrack to go down unexplored branches
  - This is perfectly suited to recursive implementation
- Other tree operations that work well with recursion are those that need to visit multiple nodes.
  - *e.g.*, In-/Pre-/Post-order traversals

The End

– Next Week - Heaps