CSC148H Lecture 5

Dan Zingaro
OISE/UT

October 6, 2008

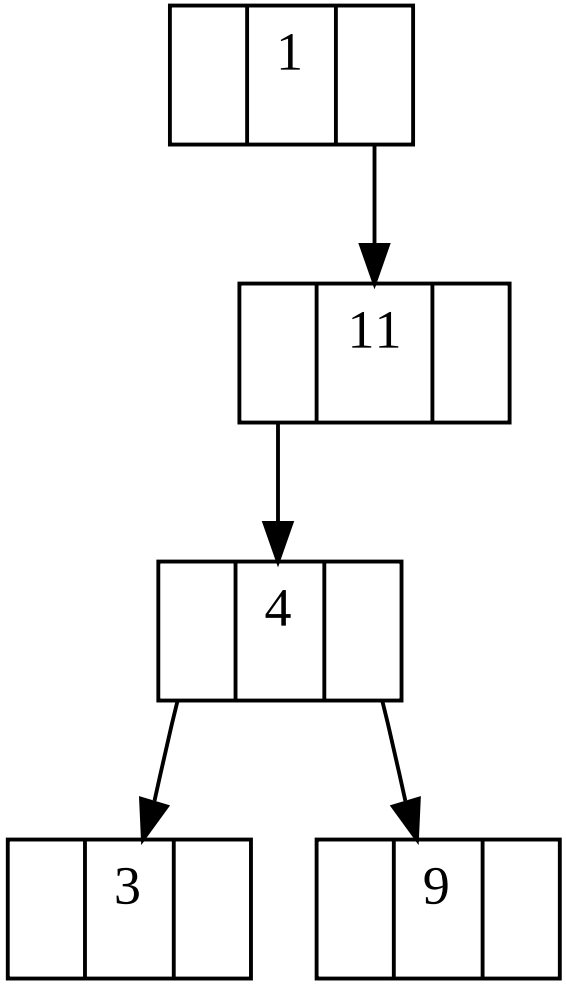
Motivating Binary Search Trees

- Last week we saw examples of where a tree is a more appropriate data structure than a linear structure.
- Sometimes we may use a tree structure even if a linear structure will do.
- For example, for certain tasks, trees can be more efficient
- Consider searching for an element in a linear structure
  - This means going through each element in the structure one at a time until you find the desired element.
  - Even if we have a sorted structure so this is fast, what if we want to insert a new item?
- For greater efficiency, we can use a binary search tree (BST) instead

What is a BST?

- A BST is a binary tree in which
  - Every node has a label (or "key")
  - Every node label is
    - Greater than the labels of all nodes in its left subtree
    - Less than the labels of all nodes in its right subtree

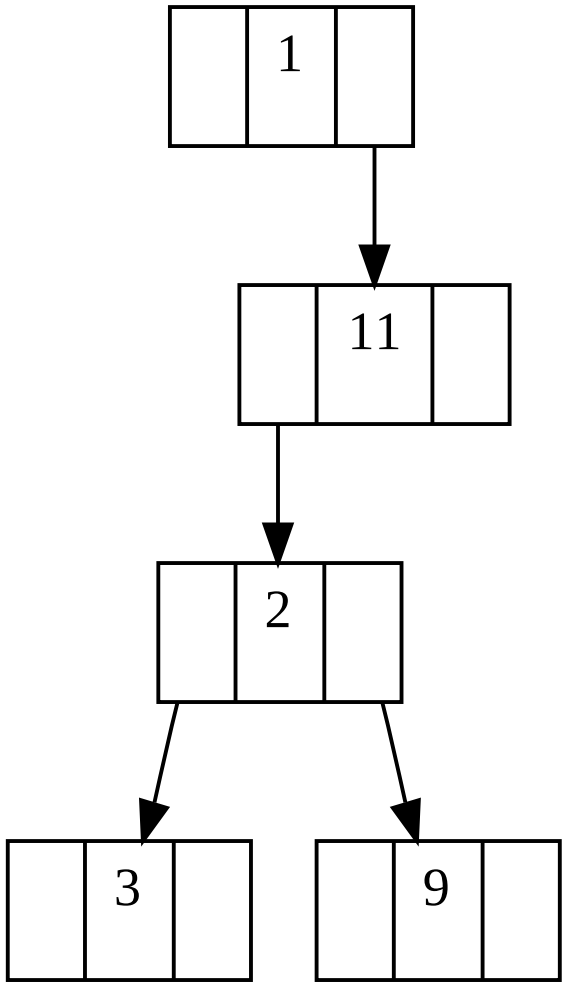
Example BST



- e.g. 11 is the right child of 1; 4 is the left child of 11; 3 is the left child of 4; 9 is the right child of 4

Potential BST

Is this a BST?

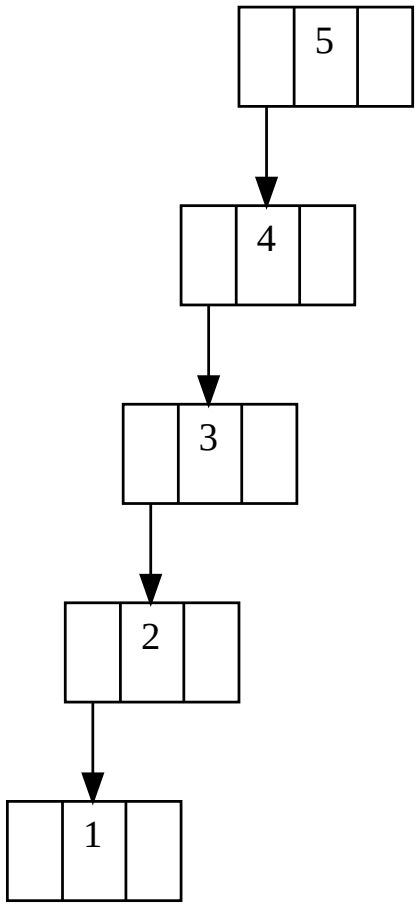


Searching a BST

- Suppose we want to know if key k exists in a BST
- We compare k to the key r at the root
- If $k < r$, we proceed down the left subtree and repeat the process
- If $k > r$, we proceed down the right subtree and repeat the process
- We'll eventually find k, or reach a dead end and conclude that k is not in the tree
- Let's try this ...

Efficiency of Searching

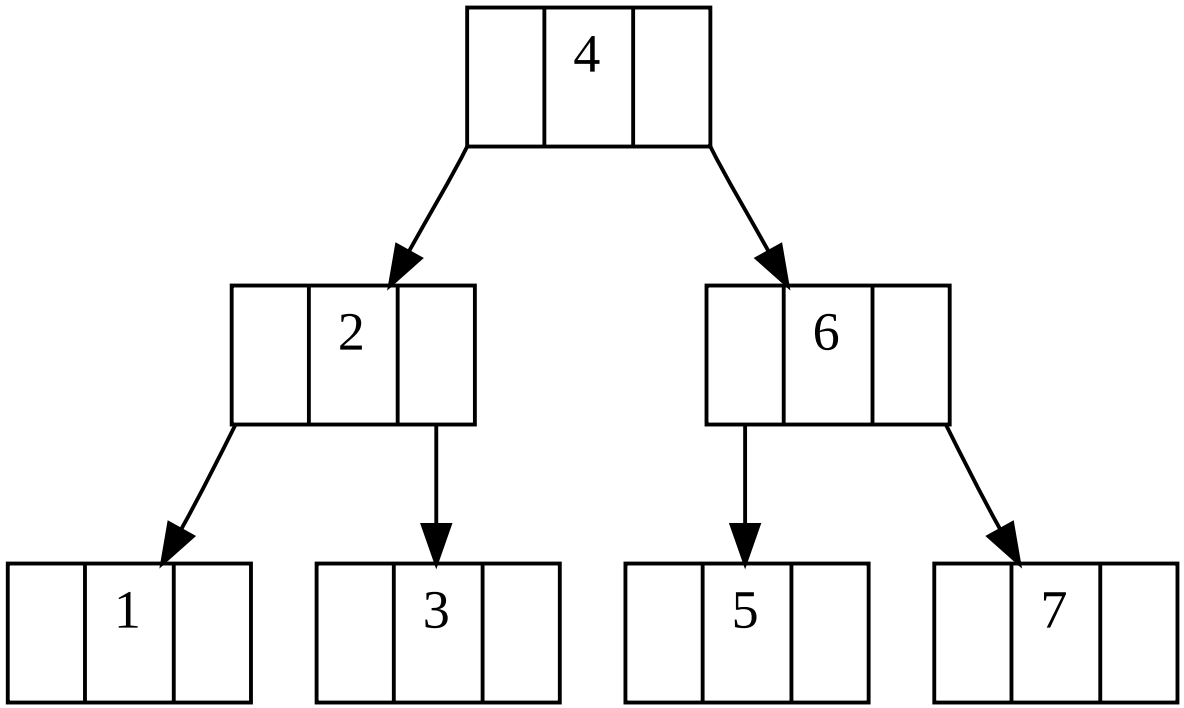
- It appears that searching for an element in a BST is more efficient than (linearly) searching for an element in a list
- But what happens when we search the tree below?



Height of a BST

- The efficiency of search comes down to the height of the BST
- If a "tree" is actually a chain, the height is $n - 1$, so searching may be no more efficient than a linear scan
- Consider a chain of right children
- If we search for a key bigger than all other keys, we will keep traversing right children until the end (and this is exactly what we'd do in a linear search)

Minimum Height BST



- A minimum-height binary tree with n nodes is a binary tree whose height is no greater than any other binary tree with n nodes

Complete Binary Trees

- To minimize height, we fill each position on each successive level before we create a new level
- A complete binary tree with n nodes is a binary tree such that every level is full, except possibly the bottom level which is filled in left to right
- We say that a level k is full if $k = 0$ and the tree is nonempty, or if $k > 0$, level $k - 1$ is full, and every node in level $k - 1$ has two children.
- Terminology alert: Some sources define a complete binary tree to be one in which all levels are full, and refer to the definition above as an "almost" complete binary tree

Maximum Nodes per Height

Height	Nodes
0	1
1	3
2	7
3	15
4	31

A tree of height h with n nodes satisfies $n <= 2^{h+1} - 1$, so h is approximately $\lg n$

Proof of Above

Proving $n <= 2^{h+1} - 1$

Base case: when $h = 0$, n is at most 1, and we have $1 <= 2^{0+1} - 1 = 1$

Inductive case: when $h > 0$, the left subtree has $l <= 2^h - 1$ nodes and the right subtree has $r <= 2^h - 1$ nodes from the inductive hypothesis, so $n <= l + r + 1 = 2^{h+1} - 1$

Balanced Trees

- If we can always ensure that a binary search tree is roughly in the shape of a minimal-height binary tree, then searching a binary tree will be much more efficient than linearly searching a list.
- You'll see more on this in later courses
  - AVL Trees
  - Red-black Trees
  - These types of trees cannot become unbalanced

BST Operations

- So far we've only discussed searching for an element in a BST
- What do we do once we found it?
- Right now, we can only really report whether it's found or not, but in many applications we may want to store some data with the node
- Your textbook calls the label of a node its key, and the data associated with the node its value

BST Operations

- In general, a BST can be used for mapping keys to data values. (This is much like a Python dictionary).
- Useful operations for such a structure include:
  - **has_key(key)**: test if a node with the given key is present in the tree
  - **get(key)**: get data associated with the key
  - **put(key, val)**: associate **val** with the given key
  - **delete(key)**: remove a node

BST Representation

- How are we going to represent a BST?
- We can use the nodes and references representation that we discussed last week.
- But what if a BST is empty? How do we keep track of this?

BST Representation...

- We use a **TreeNode** class to represent a node in the BST
- We use a **BinarySearchTree** class to represent the tree itself. This class has an attribute that points to the root **TreeNode** of the tree if it exists
- We'll define operations on the **BinarySearchTree** class, but most of them will just delegate to operations in the **TreeNode** class
- If the tree is empty (so we have no **TreeNode** reference), we proceed differently

BinarySearchTree Class

```
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def put(self, key, value):
        if self.root:
            self.root.put(key,value)
        else:
            self.root = TreeNode(key,value)

    def get(self, key):
        if self.root:
            return self.root.get(key)
        else:
            return None

    def has_key(self, key):
        return not self.get(key) is None

    def delete(self, key):
        if self.root:
            self.root = self.root.delete(key)
```

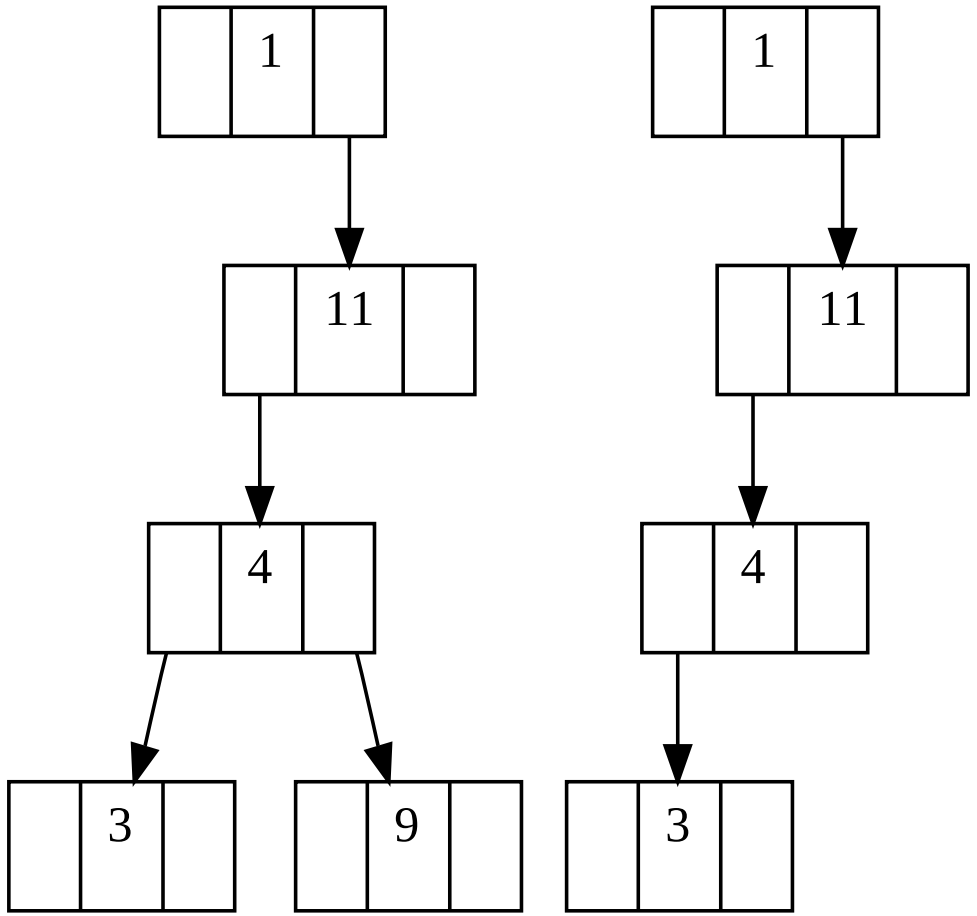
Implementing Get

```
class TreeNode:
    ...
    def get(self, key):
        """ get the value associated with the key """
        if self.key == key:
            return self.val
        if self.key > key:
            if self.left:
                return self.left.get(key)
            else:
                return None
        else:
            if self.right:
                return self.right.get(key)
            else:
                return None
```

Implementing Put

```
def put(self, key, val):
    """ add a new mapping between key and value in the tree """
    if self.key == key:
        self.val = val              # replace existing value
    elif self.key > key:            # key belongs in left subtree
        if self.left:
            self.left.put(key,val)
        else:                       # left subtree is empty
            self.left = TreeNode(key,val)
    else:                           # key belongs in right subtree
        if self.right:
            self.right.put(key,val)
        else:                       # right subtree is empty
            self.right = TreeNode(key,val)
```

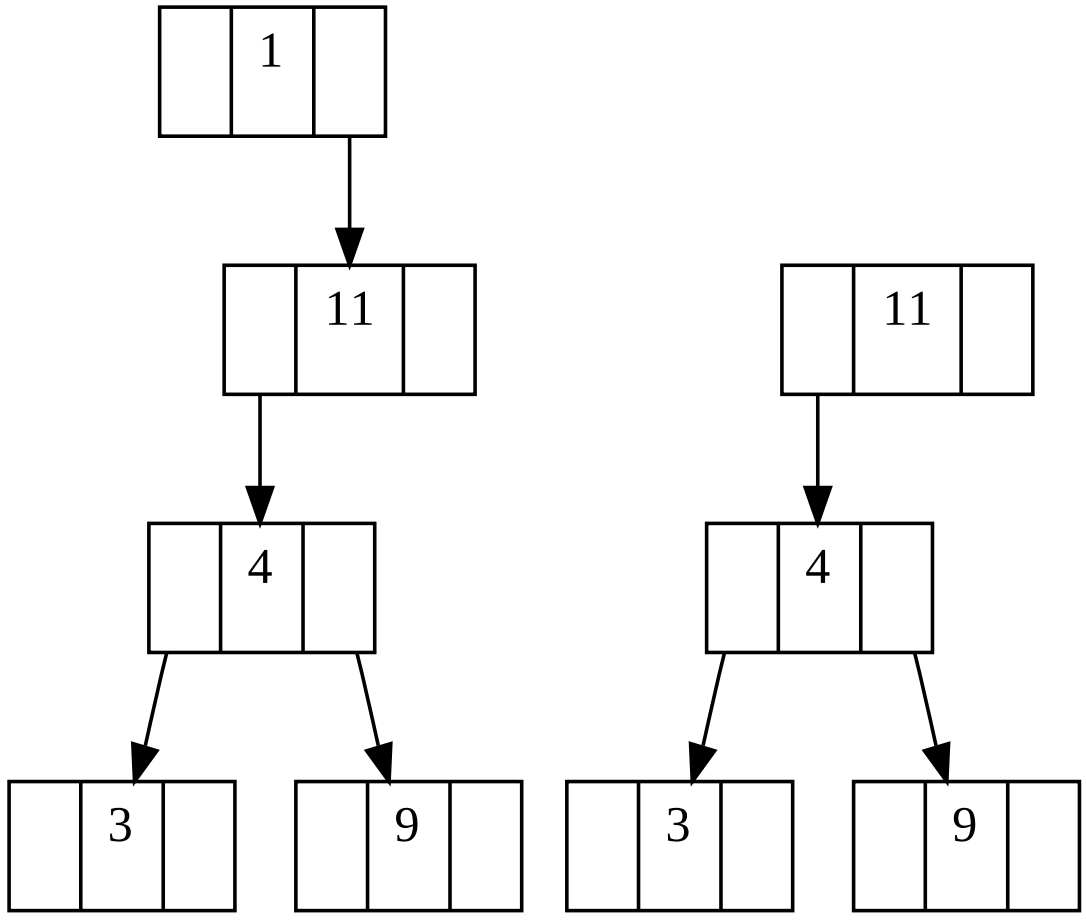
Node Deletions



(a) Original Tree (b) Tree After Deleting the Leaf 9

Figure: To delete a leaf, just remove it

Node Deletions...



(a) Original Tree

(b) Tree After Deleting the Node 1

Figure: To delete an interior node with one child, just promote that child

Node Deletions...

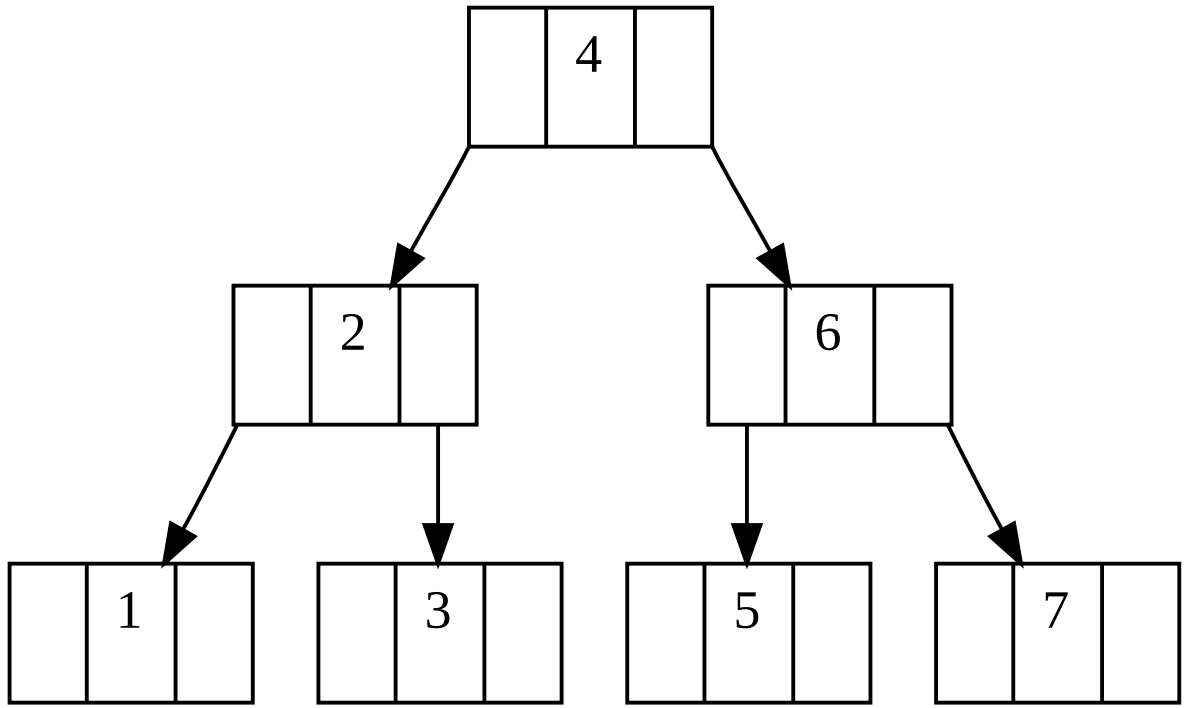


Figure: Why Can't we Just Remove 4?

Deleting a Node with Two Children

- To delete a node with two children, replace it by its successor
- This yields a new BST that cannot violate the BST property. But where's the successor?
- The successor of a node n with two children is the node with minimum key found in the right subtree of n. Why?
  - It cannot be in the left subtree (those are smaller than n)
  - The tree rooted at n contains n and our proposed successor s
  - If n is the left child of its parent, its parent (and everything in its right subtree) is bigger than s
  - If n is the right child of its parent, its parent (and everything in its left subtree) is smaller than n
  - Continue this reasoning all the way up to the root

Finding Minimum of Subtree

- To find the minimum of a subtree t, we keep traversing left children until we get to a node with no left child
- Proof: at each step, we reduce the portion of the tree that contains the minimum until we have one node remaining
- Since this node has no left child, we know how to remove it (i.e. promote its right child, if any)

Finding Minimum of Subtree...

```
def _findMin(self, parent):
    """ return the minimum node in the current tree and its parent """

    # we use an ugly trick: the parent node is passed in as an argument
    # so that eventually when the leftmost child is reached, the
    # call can return both the parent to the successor and the successor

    if self.left:
        return self.left._findMin(self)
    else:
        return [parent, self]
```

Code for Deleting a Node

```
def delete(self, key):
    """ delete the node with the given key and return the
    root node of the tree """
    if self.key == key:
        if self.right and self.left:
            # get the successor node and its parent
            [psucc, succ] = self.right._findMin(self)
            # splice out the successor
            if psucc.left == succ:
                psucc.left = succ.right
            else:
                psucc.right = succ.right
            # reset the left and right children of the successor
            succ.left = self.left
            succ.right = self.right
            return succ

        else: # one child
            if self.left:
                return self.left    # promote the left subtree
            else:
                return self.right   # promote the right subtree
    else:
        if self.key > key:          # key should be in the left subtree
            if self.left:
                self.left = self.left.delete(key)
            # else the key is not in the tree

        else:                       # key should be in the right subtree
            if self.right:
                self.right = self.right.delete(key)

    return self
```

Successor in General

- Now we know how to find the successor of a node with two children (for use in deletions)
- The same procedure can be used to find the successor of a node with only a right child
- What is the successor of a node n with no right child?
- It is the lowest ancestor whose left child is also an ancestor of n
- Prove this!