CSC148H Week 9

Sadia Sharmin

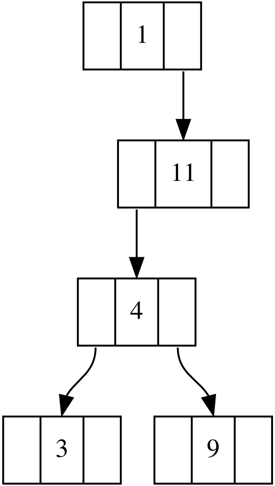
Motivating Binary Search Trees

- We've seen examples of where a tree is a more appropriate data structure than a linear structure
  - e.g. directory hierarchy, representing relationships between items
- We will use **binary search trees** to allow for efficient searching of a collection of data
  - Don't confuse **binary trees** and **binary search trees**!
  - Binary tree: branching factor at most 2
  - Binary search trees: this week

What is a BST?

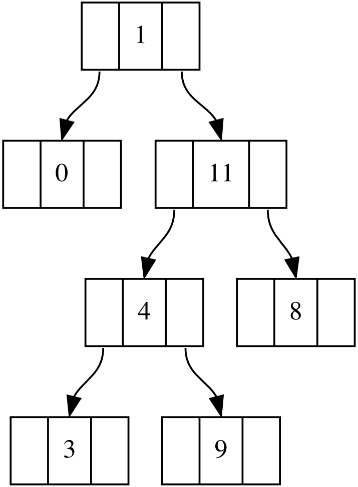
- A Binary Search Tree (BST) is a binary tree in which
  - Every node has a value
  - Every node value is
    - Greater than the values of all nodes in its left subtree
    - Less than the values of all nodes in its right subtree
  - This is called the **BST property**

Example BST



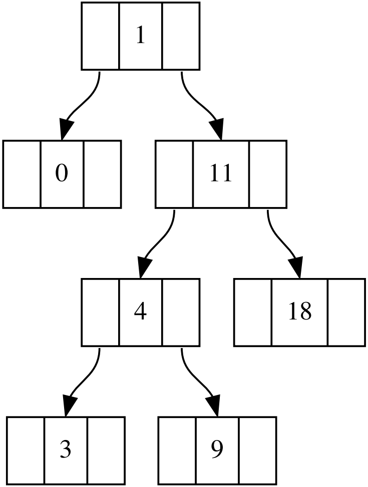
Potential BST

Is this a BST?



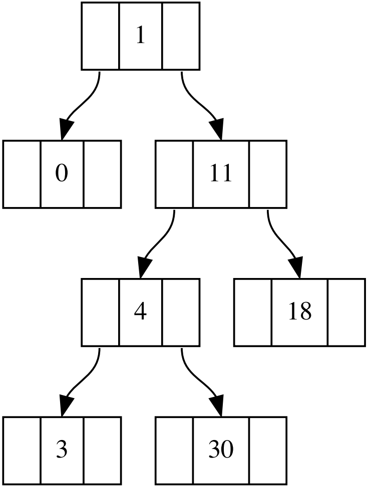
Potential BST...

Is this one a BST?



Potential BST...

This one? BST?



Searching a BST

- Suppose we want to know whether value v exists in a BST
- We compare v to the value r at the root
  - If $v = r$, then the value is found and we are done
  - If $v < r$, we proceed down the left subtree and repeat the process
  - If $v > r$, we proceed down the right subtree and repeat the process
- If we go off the tree in this process, then the value isn't in the BST
- Let's try this ...

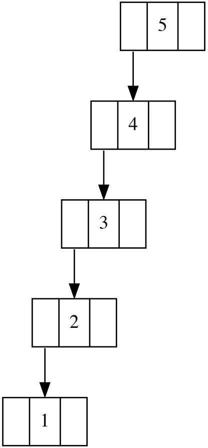
BST visualizer: https://visualgo.net/bn/bst

BST Insertion

- Insertion is very similar to searching for a node
- We compare v to the value r at the root
  - If $v = r$, then the value is already in the tree and we are done
  - If $v < r$, we proceed down the left subtree and repeat the process
  - If $v > r$, we proceed down the right subtree and repeat the process
- Once we go off the tree, that's where the new node goes

Efficiency of Searching

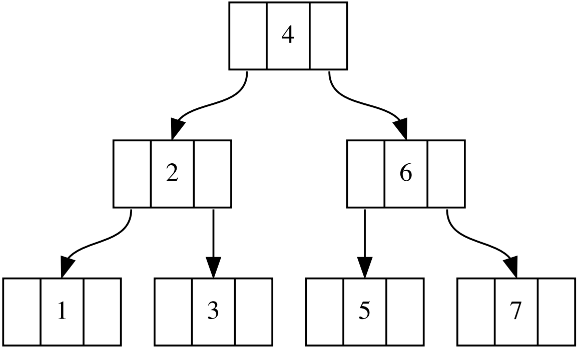
- It appears that searching for an element in a BST is more efficient than (linearly) searching for an element in a list
- But what happens when we search the tree below?



Height of a BST

- The efficiency of search depends on 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 search
- Consider the chain of left children on the previous slide
  - If we search for a value that is smaller than all existing values, we will keep traversing left children until the end
  - This is exactly how linear search works

Minimum-Height BST



No other BST of 7 nodes can have less height.

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
  - $k > 0$, level $k - 1$ is full, and every node in level $k - 1$ has two children

See: http://web.cecs.pdx.edu/~sheard/course/Cs163/Doc/FullvsComplete.html

Balanced Trees

Tree in which the depth of the subtrees differ by no more than one.

- If we can always ensure that a binary search tree is roughly in the shape of a minimal-height binary tree, then searching the BST will be much more efficient than linearly searching a list
- You'll see more on this in later courses
  - e.g. AVL Trees, red-black Trees . . . trees that "balance themselves"

Note: Every complete binary tree is balanced but not the other way around.

BST Representation

- We have seen two ways to represent trees so far
  - List of lists
  - Nodes and references
- We'll use a form of nodes and references to represent a BST

BST Representation...

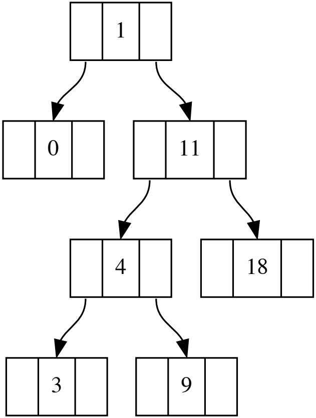
- We'll use a `BTNode` class to represent a node in the BST
- We'll use a `BST` class to represent the tree itself
- The `BST` class has a `root` attribute that is
  - `None` when the BST is empty, or
  - A reference to the root `BTNode` of the tree otherwise

Deleting a Node

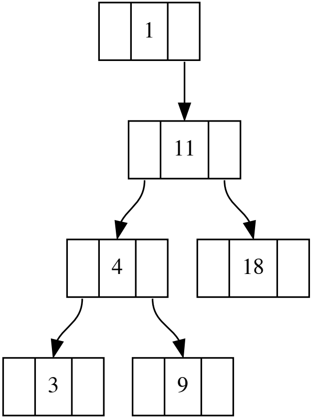
- There are several cases to consider, depending on where the node exists in the tree
- We must be able to delete the node without violating the BST property
- We will discuss how to delete
  - A leaf node (easy)
  - A node with one child (not bad)
  - A node with two children (a bit tricky)

Deleting a Node: Leaf

To delete a leaf, just remove it



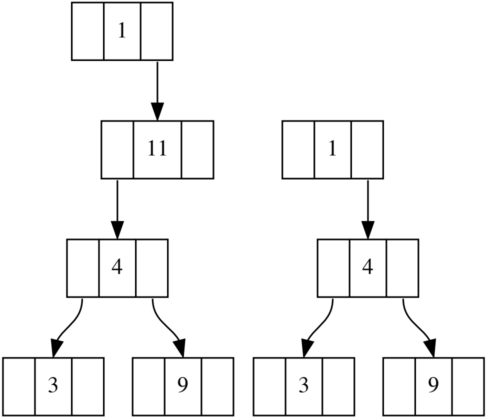
(a) Original Tree



(b) Tree After Deleting the Leaf 0

Deleting a Node: One Child

To delete a node with a single child, cut out that node

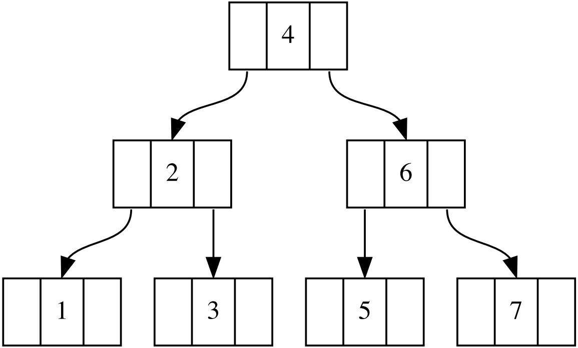


(c) Original Tree

(d) Tree After Deleting the node 11

Deleting a Node: Two Children

When a node has two children, it may not be correct to move one of the children up. e.g. let's try to remove 4.



Deleting a Node with Two Children

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

Finding Maximum of Subtree

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