CSC148H Week 8

Ilir Dema, Michael Miljanovic

Summer 2021

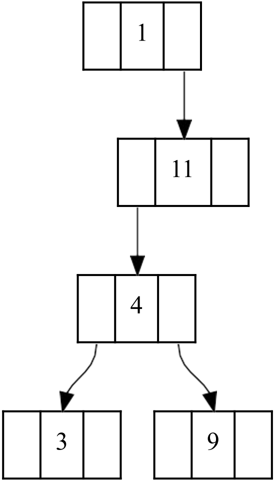
Motivating Binary Search Trees

- We've seen examples of where a tree structure is more appropriate 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: binary tree with extra constraint

What is a BST?

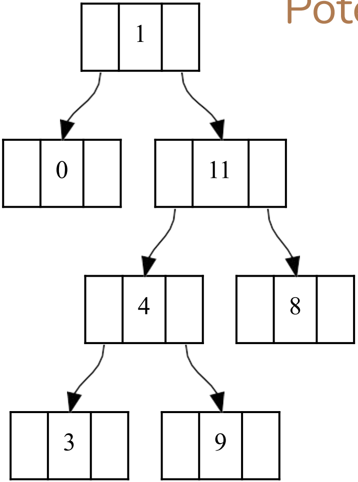
- A Binary Search Tree (BST) is a binary tree in which
  - Every node has a value
  - Every node value is
    - Greater than or equal to the values of all nodes in its left subtree
    - Less than or equal to the values of all nodes in its right subtree
  - This is called the **BST property**

Example BST



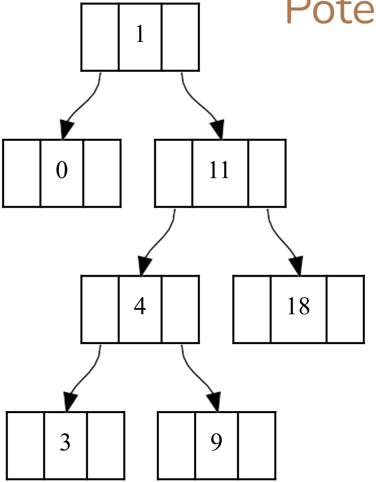
Potential BST

Is this a BST?



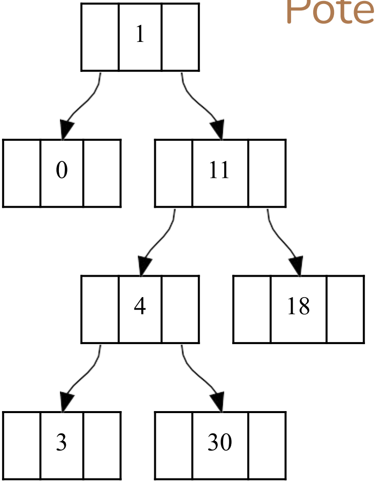
Potential BST

Is this a BST?



Potential BST

Is this a 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 is **not** in the BST
- Let's try this . . .

BST Insertion

- Insertion into a BST is very similar to searching a BST
- To insert v , we compare v to the value r at the root
  - If $v = r$, then we proceed down the tree of our choice
  - 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

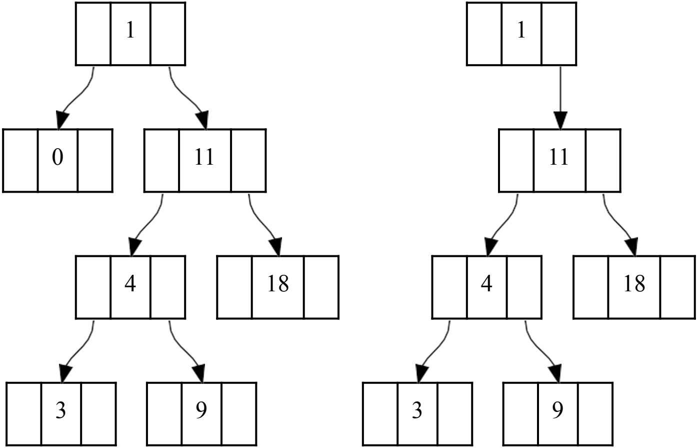
Worksheet 1

- BST Traversal

Deleting a Node

- Deleting has more cases than insertion
- What we do depends 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)

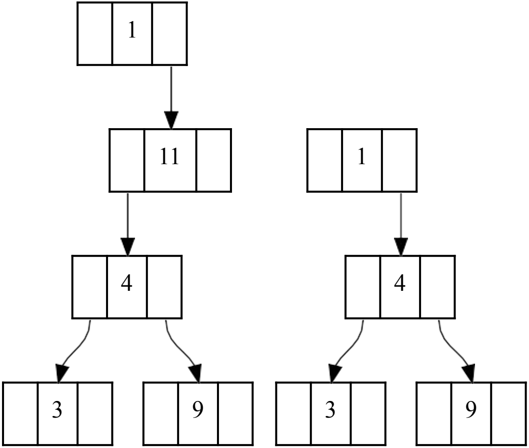
Deleting a Node: Leaf



(a) Original Tree (b) Tree After Deleting the Leaf 0

Figure:To delete a leaf, just remove it

Deleting a Node: One Child



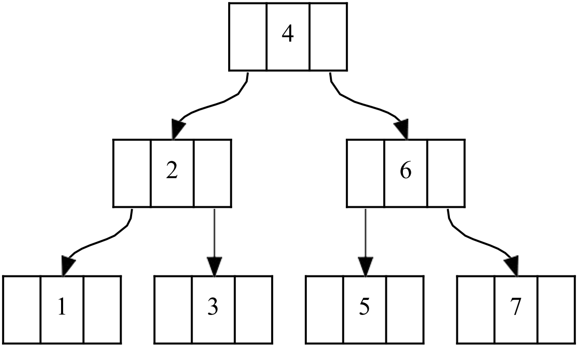
(a) Original Tree

(b) Tree After Deleting the Node 11

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

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.



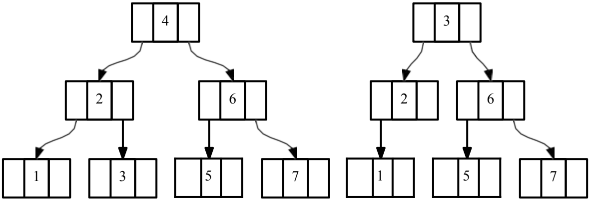
Deleting a Node: Two Children…

- 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 maximum node 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
- Intuition: 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. we are now in the easier case of deleting a leaf or a node with a single child)

Deleting a Node: Two Children...



(a) Original Tree

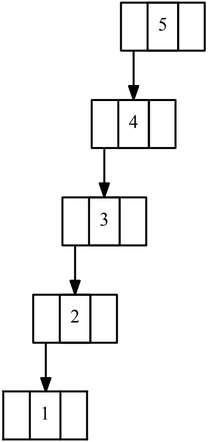
(b) Tree After Deleting the Node 4

Figure:To delete a node with two children, replace by predecessor

Worksheet 2

- BST Delete

Efficiency of Searching

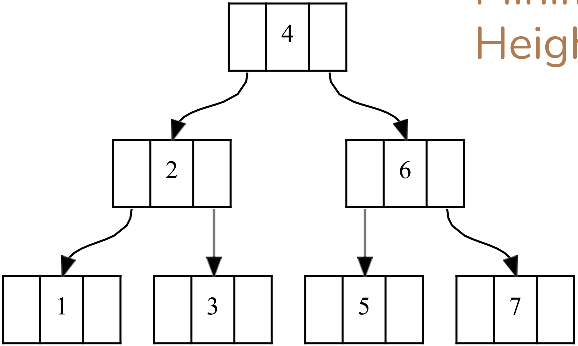


- It appears that searching for an element in a BST is more efficient than (linearly) searching for an element in a list
- But what happens when we search this tree?

Height of a BST

- The efficiency of search depends on the height of the BST
- If a “tree” is actually a chain, then searching it is 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 eliminate just one node at a time
  - This is exactly how linear search works

Minimum Height BST



No other BST of 7 nodes can have less height.

Maximum Nodes Per Height

Tree Height	max Nodes
1	1
2	3
3	7
4	15

A binary tree of height h with n nodes satisfies $n \leq 2^h - 1$.

Proof of Maximum Nodes

Proving $n \leq 2^h - 1$
Base case: when $h = 1$, we have at most 1 node,
and $1 <= 2^1 - 1 = 1$.
Inductive step: $h > 1$

- Suppose that $n \leq 2^p - 1$ for all trees of height $p < h$
- The left and right subtree each has height at most $h - 1$
- From the inductive hypothesis we have that the left subtree has $l \leq 2^{h-1} - 1$ nodes and the right subtree has $r \leq 2^{h-1} - 1$ nodes
- Adding 1 for the root, we have $n = l + r + 1 \leq 2^h - 1$

Balanced Trees

- If we can always ensure that a BST 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"
- For now, we will be using trees that can unfortunately become very unbalanced!

Worksheet 3

- BST Efficiency