CSCI 104
Binary Search Trees

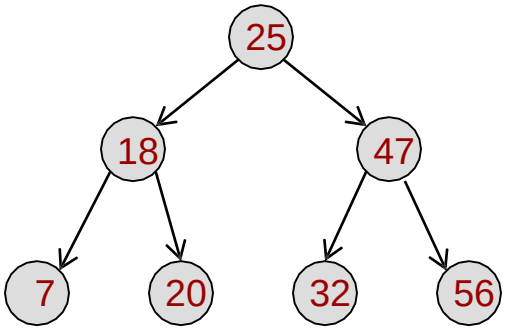
Mark Redekopp
Sandra Batista
David Kempe

Properties, Insertion and Removal

BINARY SEARCH TREES

Binary Search Tree

- Binary search tree = binary tree where all nodes meet the property that:
  - All values of nodes in left subtree are less than or equal to the parent's value
  - All values of nodes in right subtree are greater than the parent's value



If we wanted to print the values in sorted order would you use an pre-order, in-order, or post-order traversal?

BST Insertion

- Important: To be efficient (useful) we need to keep the binary search tree balanced
- Practice: Build a BST from the data values below
  - To insert an item walk the tree (go left if value is less than node, right if greater than node) until you find an empty location, at which point you insert the new value

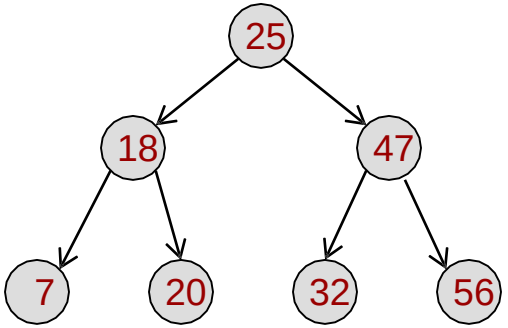
Insertion Order: 25, 18, 47, 7, 20, 32, 56

Insertion Order: 7, 18, 20, 25, 32, 47, 56

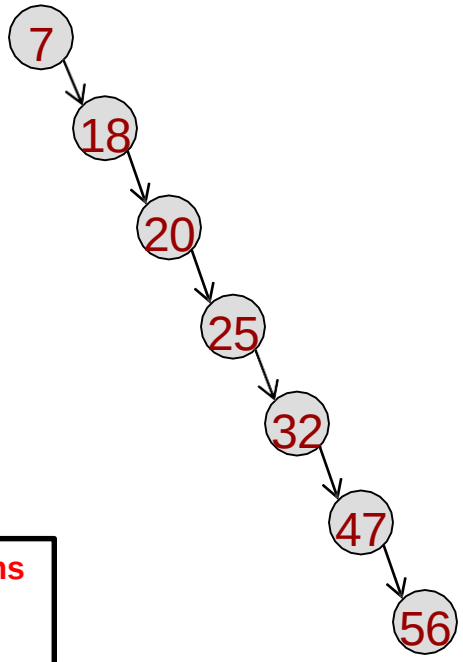
BST Insertion

- Important: To be efficient (useful) we need to keep the binary search tree balanced
- Practice: Build a BST from the data values below
  - To insert an item walk the tree (go left if value is less than node, right if greater than node) until you find an empty location, at which point you insert the new value
- https://www.cs.usfca.edu/~galles/visualization/BST.html

Insertion Order: 25, 18, 47, 7, 20, 32, 56



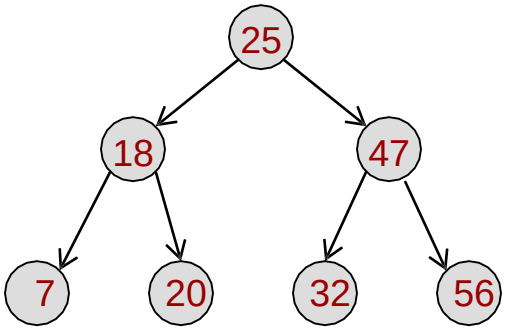
Insertion Order: 7, 18, 20, 25, 32, 47, 56



A major topic we will talk about is algorithms to keep a BST balanced as we do insertions/removals

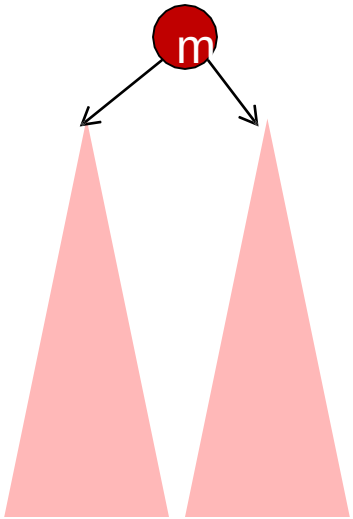
BST Find

- Exercise: Given a node to a root of a tree and a value, write a recursive function **find** that returns a pointer to the node containing the value or nullptr if not found:
- **shared_ptr<Node> find(shared_ptr<Node> root, int value);**



Successors & Predecessors

- Let's take a quick tangent that will help us understand how to do **BST Removal**
- Given a node in a BST
  - Its predecessor is defined as the next smallest value in the tree
  - Its successor is defined as the next biggest value in the tree
- Where would you expect to find a node's successor?
- Where would find a node's predecessor?



Predecessors

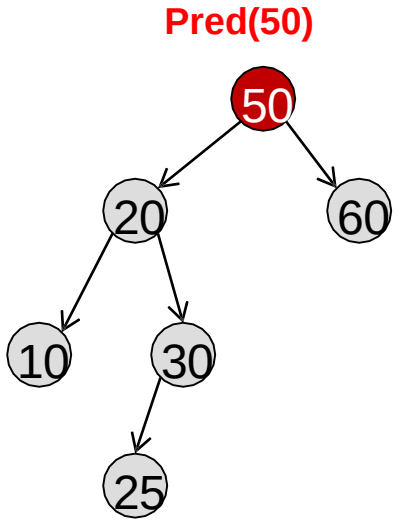
- If left child exists, predecessor is the right most node of the left subtree (case 1)
- Else walk up the ancestor chain

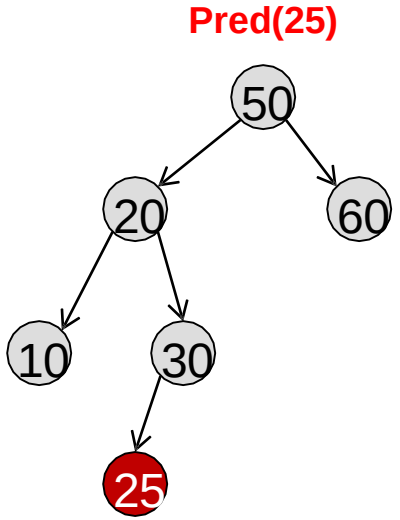
  if traverse the first right child pointer

  - Parent of right child is predecessor (case 2)

  else

  - reach root, then there is no predecessor (case 3)





Predecessors

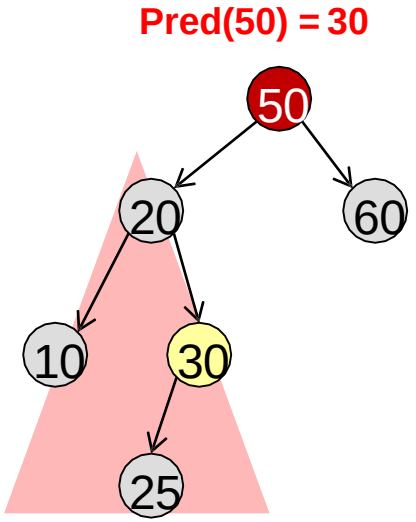
- If left child exists, predecessor is the right most node of the left subtree (case 1)
- Else walk up the ancestor chain

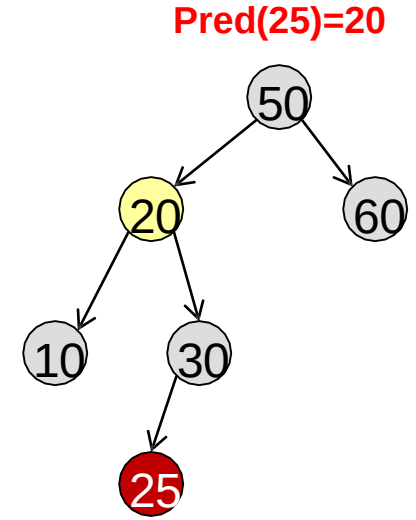
  if traverse the first right child pointer

  - Parent of right child is predecessor (case 2)

  else

  - reach root, then there is no predecessor (case 3)





Successors

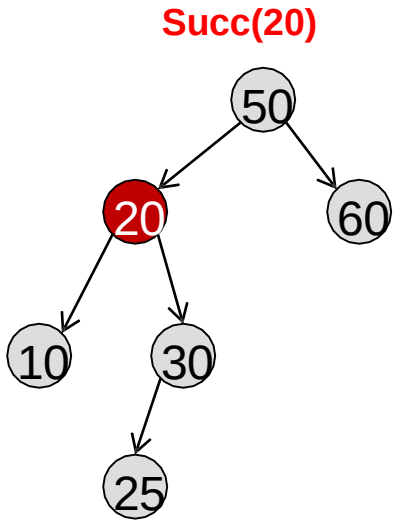
- If right child exists, successor is the left most node of the right subtree (case 1)
- Else walk up the ancestor chain

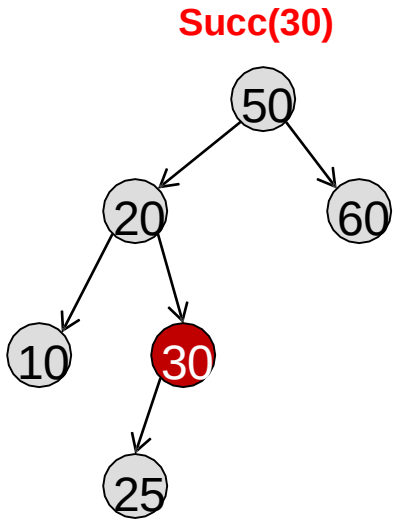
  if traverse the first left child pointer

  - parent of left child is successor (case 2)

  else

  - reach root, then there is no successor (case 3)





Successors

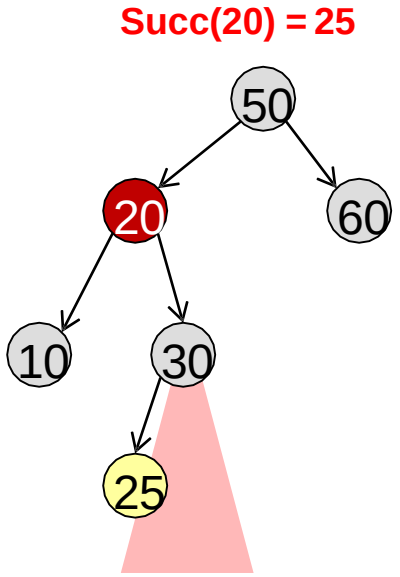
- If right child exists, successor is the left most node of the right subtree (case 1)
- Else walk up the ancestor chain

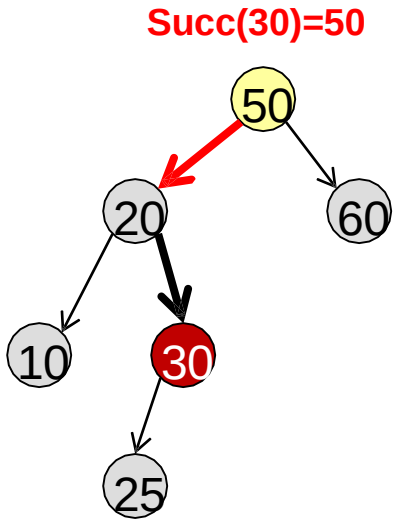
  if traverse the first left child pointer

  - parent of left child is successor (case 2)

  else

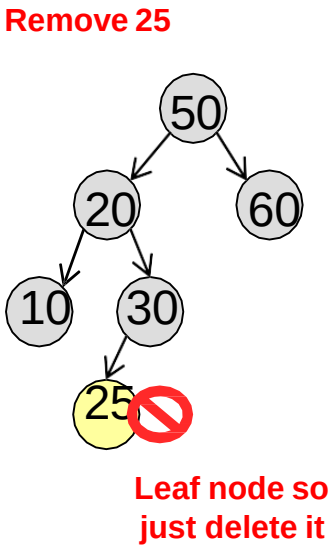
  - reach root, then there is no successor (case 3)





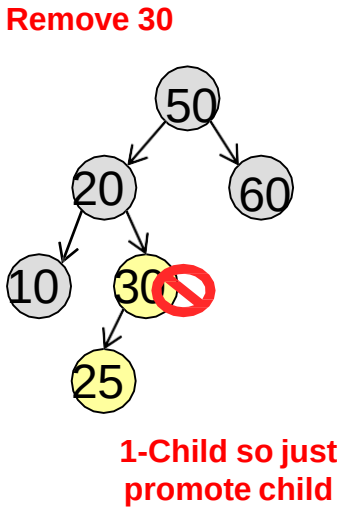
BST Removal

- To remove a value from a BST…
  - First find the node containing value using search
  - If the node is in a leaf node, simply remove that leaf node



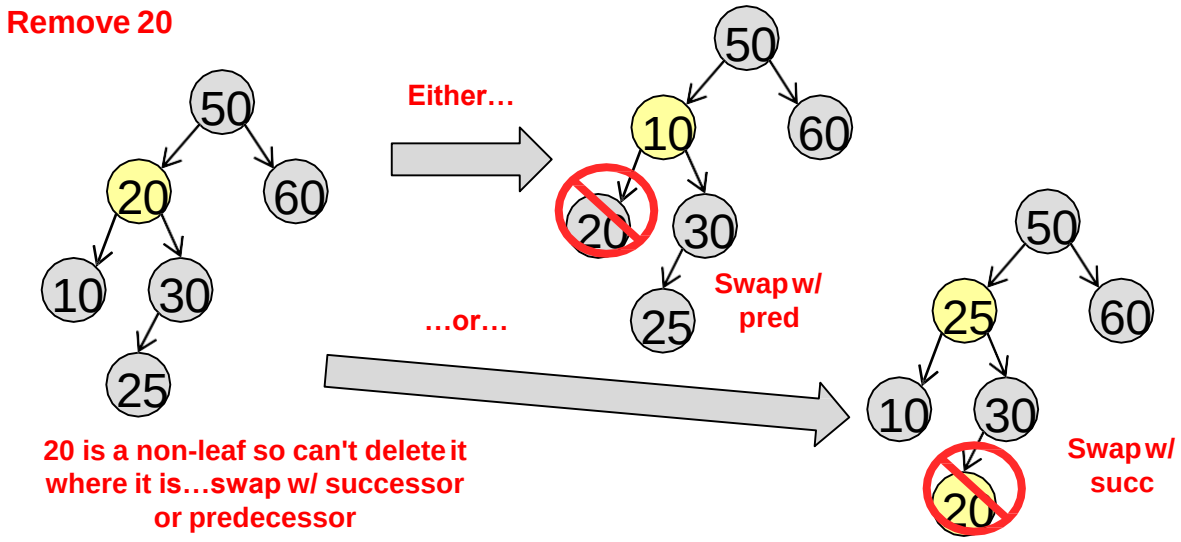
BST Removal

- To remove a value from a BST…
  - First find the node containing value using search
  - If the value is in a non-leaf node with only one child, promote the child



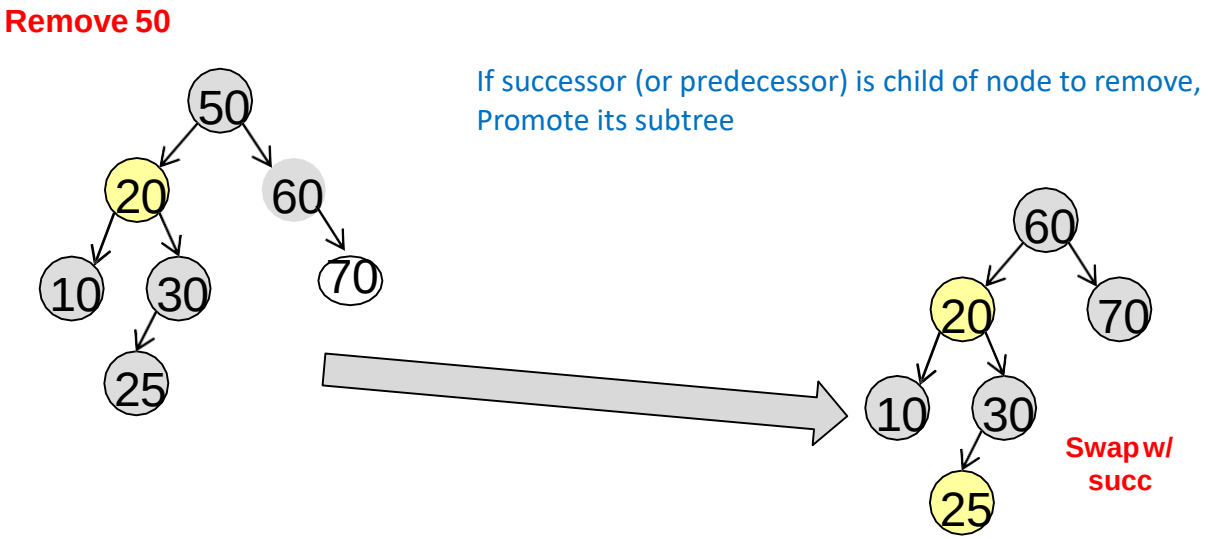
BST Removal

- To remove a value from a BST…
  - First find the node containing value using search
  - If non-leaf node with two children, swap the value with its in-order successor or predecessor and then remove the value
  - What if successor and predecessor are not leaf nodes?



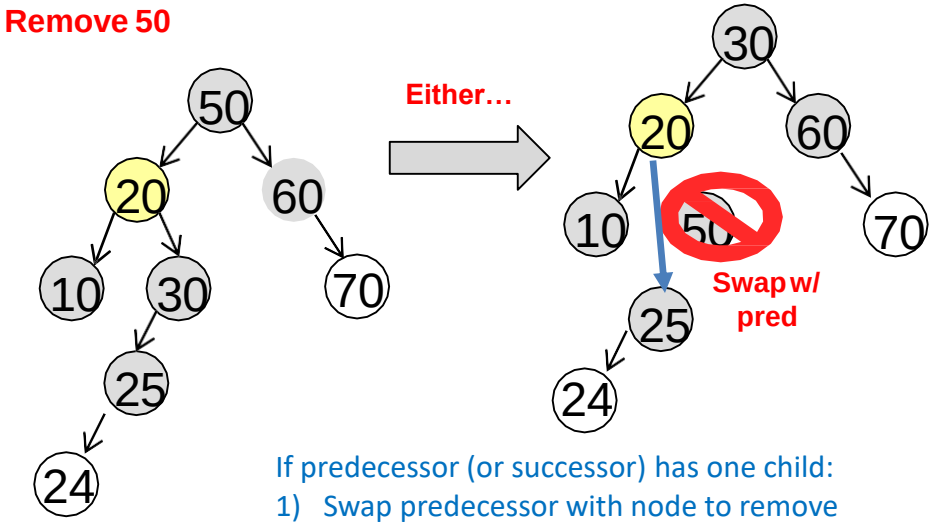
BST Removal

- To remove a value from a BST…
  - First find the node
  - If the node non-leaf node with two children, swap the value with its in-order successor or predecessor and then remove the value
  - A predecessor is either a leaf or has no right child
  - A successor is either leaf or has no left child



BST Removal

- To remove a value from a BST…
  - First find the node
  - If the node non-leaf node with two children, swap the value with its in-order successor or predecessor and then remove the value
  - A predecessor node is either a leaf or has no right child
  - A successor is either leaf or has no left child



If predecessor (or successor) has one child:

1) Swap predecessor with node to remove
2) Remove the predecessor node and promote its only child (and its subtree)

BST Efficiency

- Insertion
  - Balanced: O(log n)
  - Unbalanced: O(n)
- Removal
  - Balanced : O(log n)
  - Unbalanced: O(n)
- Find/Search
  - Balanced : O(log n)
  - Unbalanced: O(n)

```
#include<iostream>
using namespace std;

// Bin. Search Tree
template <typename T>
class BST
{
 public:
 BTree();
 ~BTree();
 virtual bool empty() = 0;
 virtual void insert(const T& v) = 0;
 virtual void remove(const T& v) = 0;
 virtual T* find(const T& v) = 0;
};
```

Trees & Maps/Sets

- C++ STL "maps" and "sets" use binary search trees internally to store their keys (and values) that can grow or contract as needed
- This allows O(log n) time to find/check membership
  - BUT ONLY if we keep the tree balanced!