CPSC 221: Data Structures
Dictionary ADT
Binary Search Trees

Alan J. Hu
(Using Steve Wolfman's Slides)

Learning Goals

After this unit, you should be able to...

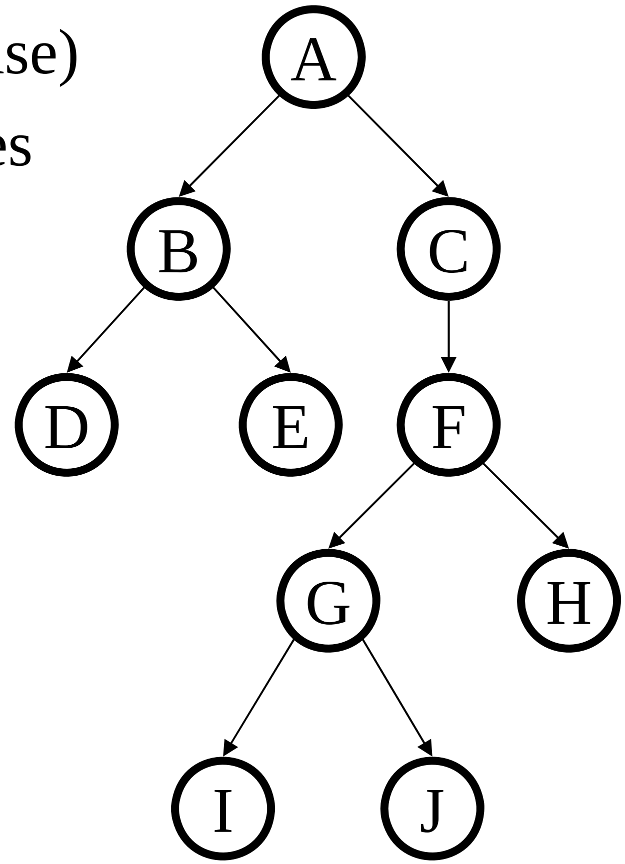
- Determine if a given tree is an instance of a particular type (e.g. binary search tree, heap, etc.)
- Describe and use pre-, in- and post-order traversal algorithms
- Describe the properties of binary trees, binary search trees, and more general trees; Implement iterative and recursive algorithms for navigating them in C++
- Compare and contrast ordered versus unordered trees in terms of complexity and scope of application
- Insert and delete elements from a binary tree

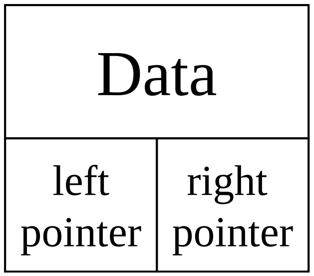
Today’s Outline

- Binary Trees
- Dictionary ADT
- Binary Search Trees
- Deletion
- Some troubling questions

Binary Trees

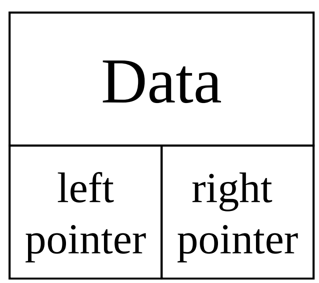
- Binary tree is *recursive* definition!
  - an empty tree (NULL, in our case)
  - or, a root node with two subtrees
- Properties
  - max # of leaves:
  - max # of nodes:
- Representation:

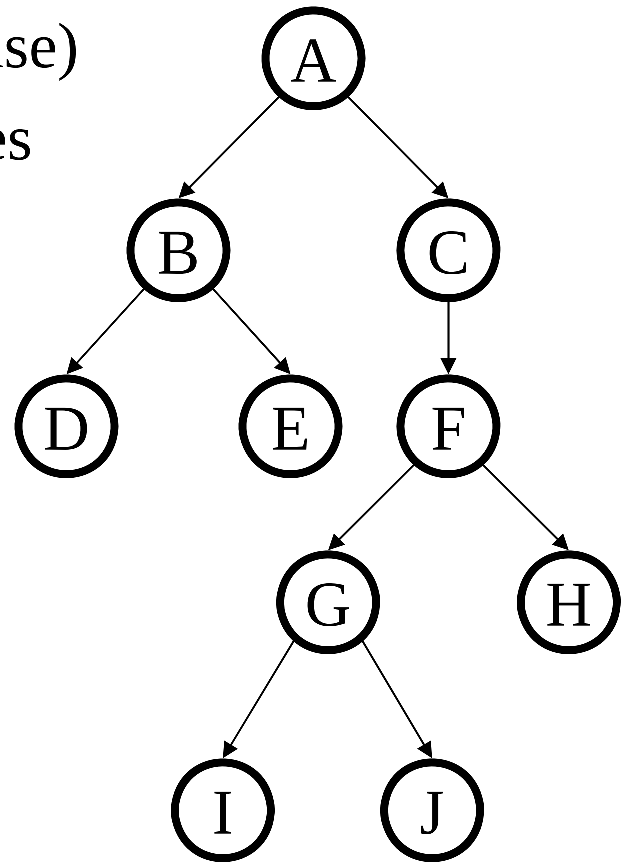




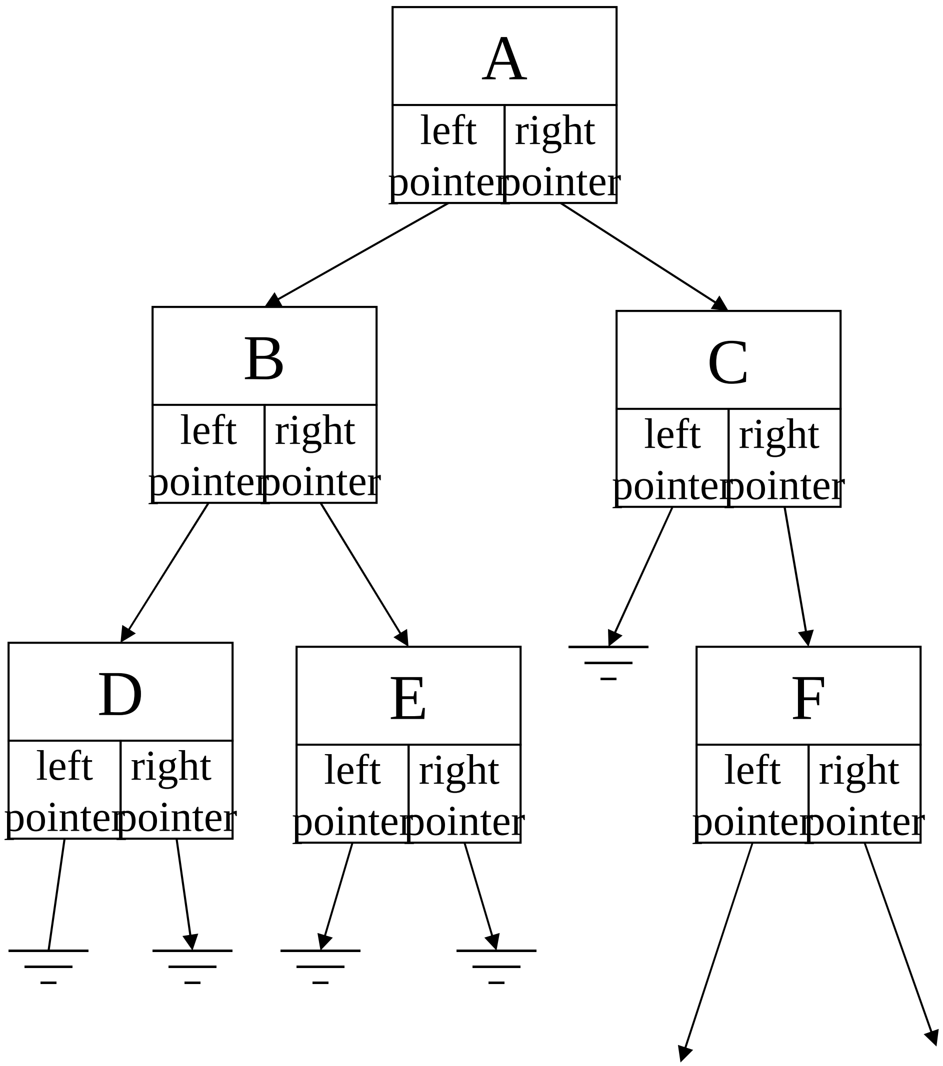
Binary Trees

- Binary tree is *recursive* definition!
  - an empty tree (NULL, in our case)
  - or, a root node with two subtrees
- Properties
  - max # of leaves: 2^h
  - max # of nodes: $2^{h+1}-1$
- Representation:

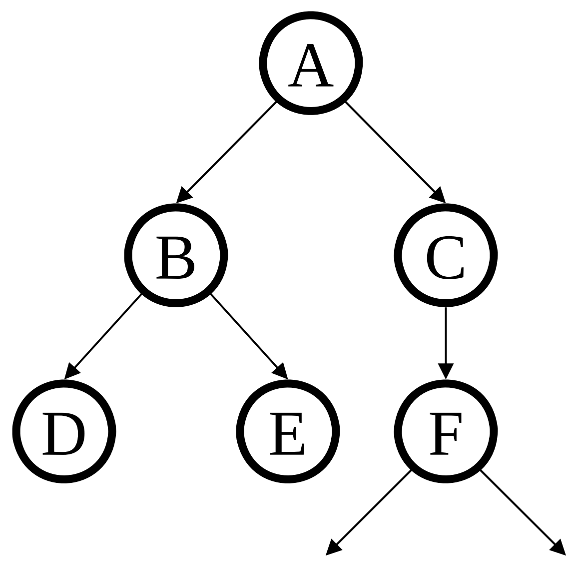




Representation




```
struct Node {
  KTYPE key;
  DTYPE data;
  Node * left;
  Node * right;
};
```



Today's Outline

- Binary Trees
- Dictionary ADT
- Binary Search Trees
- Deletion
- Some troubling questions

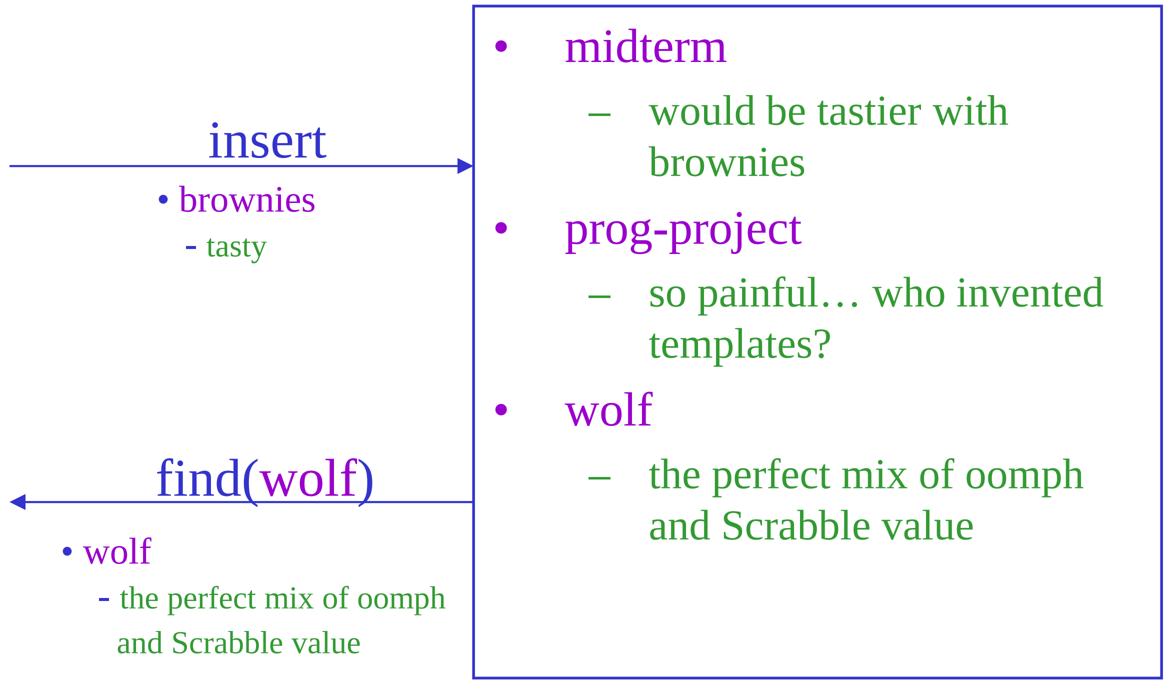
What We Can Do So Far

- Stack
  - Push
  - Pop
- Queue
  - Enqueue
  - Dequeue
- List
  - Insert
  - Remove
  - Find
- Priority Queue
  - Insert
  - DeleteMin

What's wrong with Lists?

Dictionary ADT

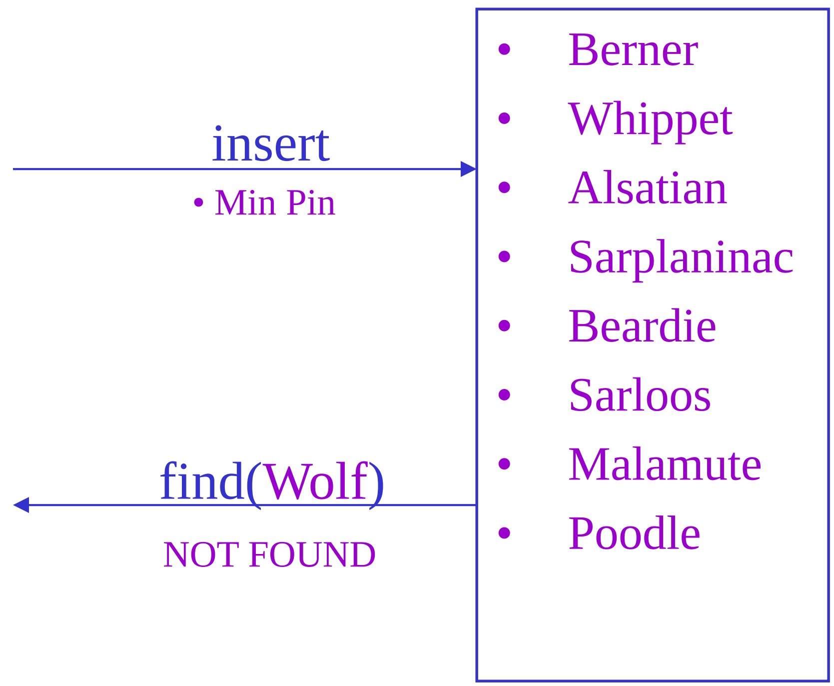
- Dictionary operations
  - create
  - destroy
  - insert
  - find
  - delete



- Stores *values* associated with user-specified *keys*
  - values may be any (homogenous) type
  - keys may be any (homogenous) comparable type

Search/Set ADT

- Dictionary operations
  - create
  - destroy
  - insert
  - find
  - delete



- Stores keys
  - keys may be any (homogenous) comparable
  - quickly tests for membership

A Modest Few Uses

- Arrays and “Associative” Arrays
- Sets
- Dictionaries
- Router tables
- Page tables
- Symbol tables
- C++ Structures
- Python’s __dict__ that stores fields/methods

Desiderata

- Fast insertion
  - runtime: $O(1)$ $O(\lg n)$
- Fast searching
  - runtime: $O(1)$ $O(\lg n)$
- Fast deletion
  - runtime: $O(1)$ $O(\lg n)$

Naïve Implementations

	insert	find	delete
• Linked list (unordered)			
• Unsorted array			
• Sorted array			

Naïve Implementations

	insert	find	delete (no find)
• unordered Linked list	O(1)	O(n)	O(1)/O(n)
• Unsorted array	O(1)	O(n)	~~O(n)/O(n)~~ O(1)/O(n)
• Sorted array	O(n)	O(log n)	O(n)

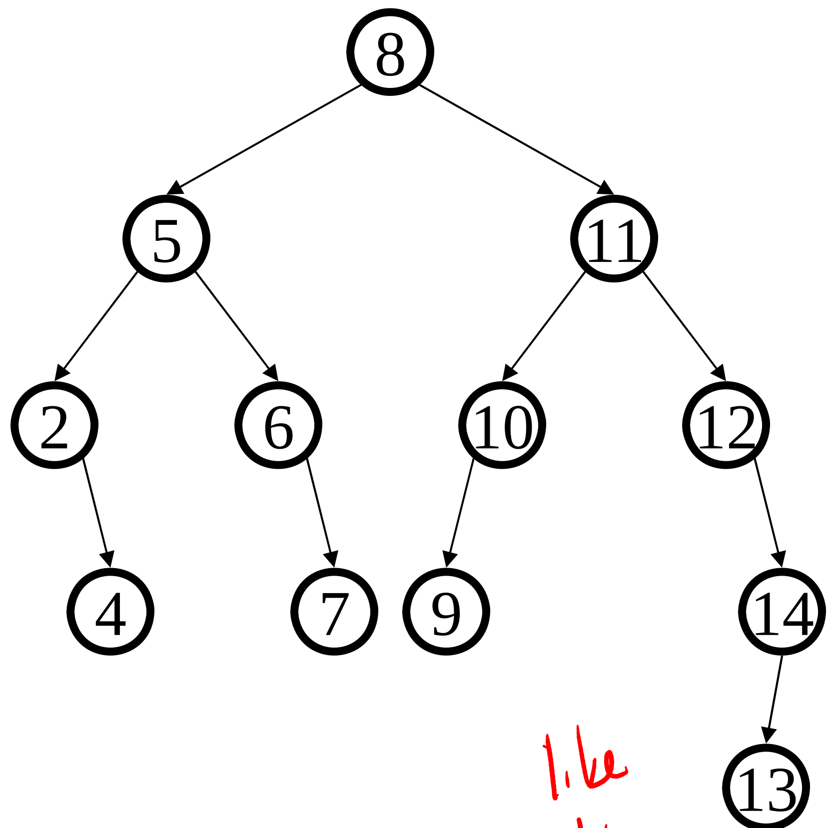
so close!

Today's Outline

- Binary Trees
- Dictionary ADT
- Binary Search Trees
- Deletion
- Some troubling questions

Binary Search Tree Dictionary Data Structure

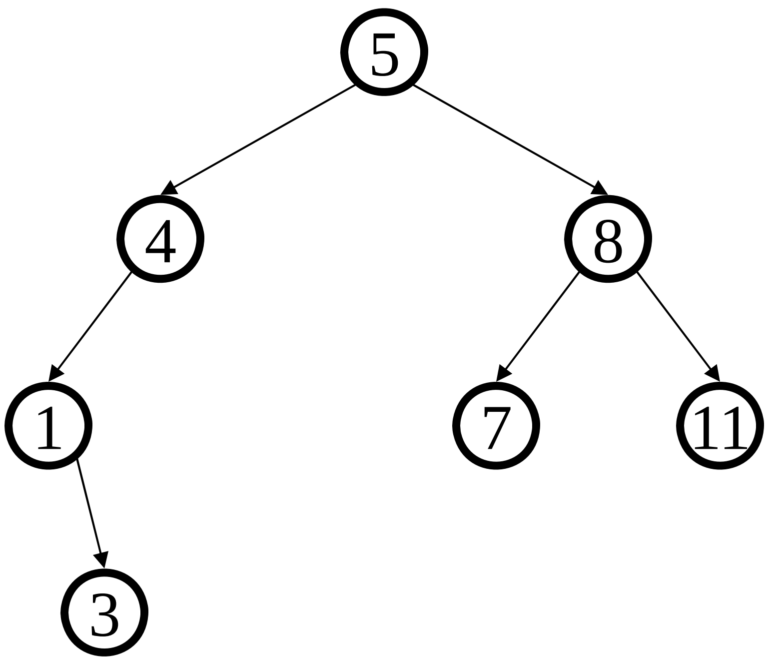
- Binary tree property
  - each node has ≤ 2 children
  - result:
    - storage is small
    - operations are simple
    - average depth is small
- Search tree property
  - all keys in left subtree smaller than root's key
  - all keys in right subtree larger than root's key
  - result:
    - easy to find any given key



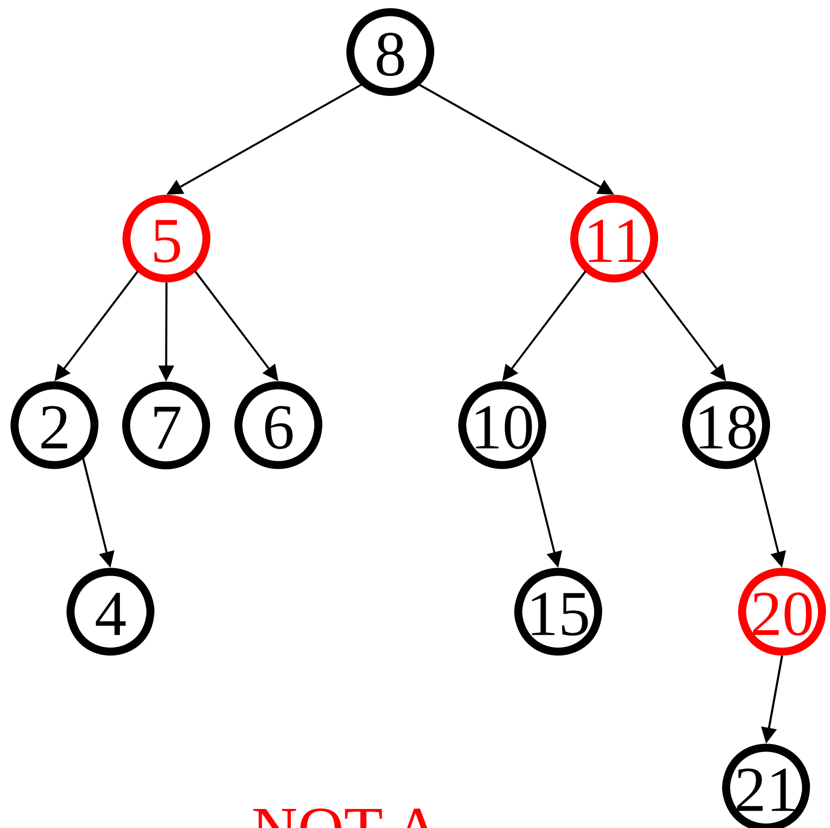
like QSort: avg O(log n) height/depth

Getting to Know BSTs

Example and Counter-Example



BINARY SEARCH TREE

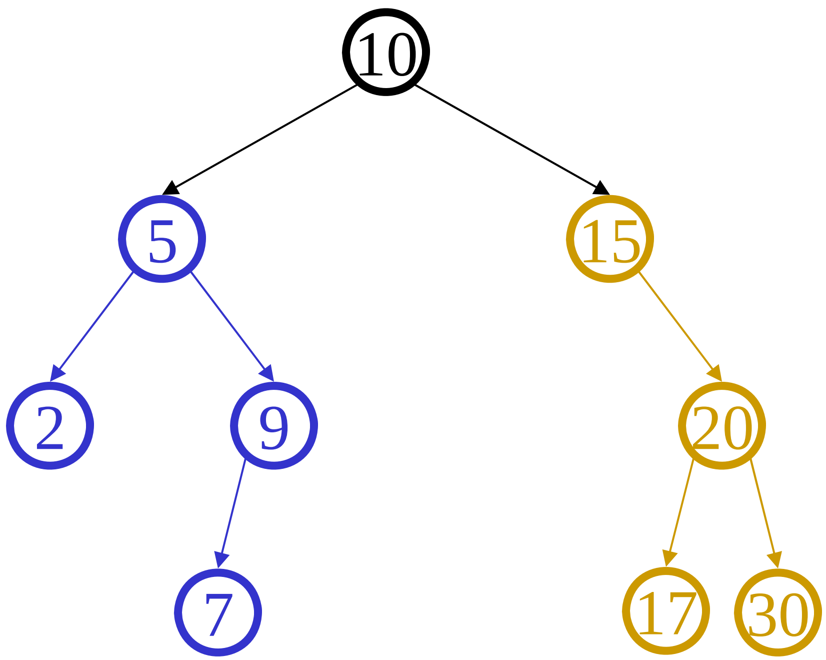


NOT A
BINARY SEARCH TREE

Getting to Know All About BSTs

In Order Listing

```
struct Node {
  // constructors omitted
  KTYPE key;
  DTYPE data;
  Node *left, *right;
};
```



In order listing:

2→5→7→9→10→15→17→20→30

Aside: Traversals

- Pre-Order Traversal: Process the data at the node first, then process left child, then process right child.
- Post-Order Traversal: Process left child, then process right child, then process data at the node.
- In-Order Traversal: Process left child, then process data at the node, then process right child.

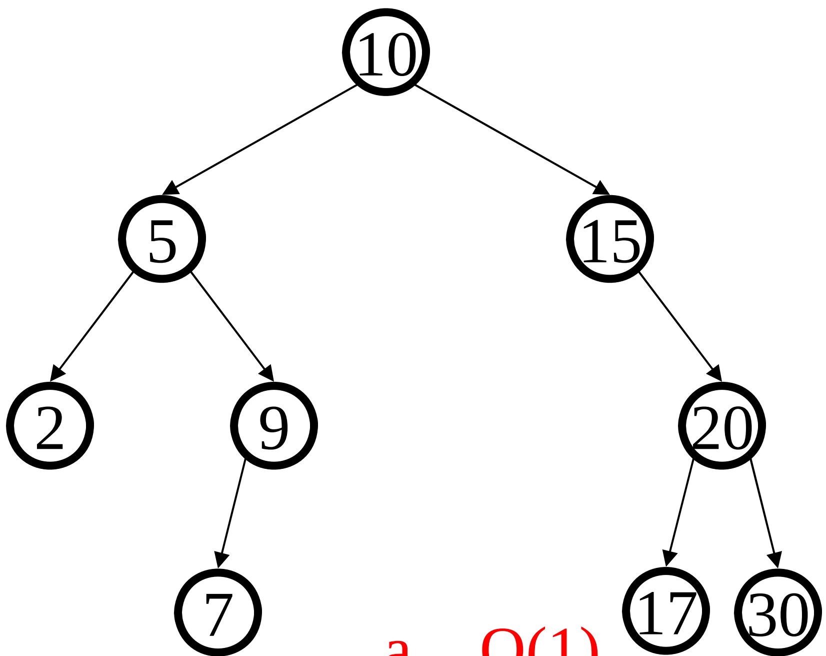
Code?

Aside: Traversals

- Pre-Order Traversal: Process the data at the node first, then process left child, then process right child.
- Post-Order Traversal: Process left child, then process right child, then process data at the node.
- In-Order Traversal: Process left child, then process data at the node, then process right child.

Who cares? These are the most common ways in which code processes trees.

Finding a Node

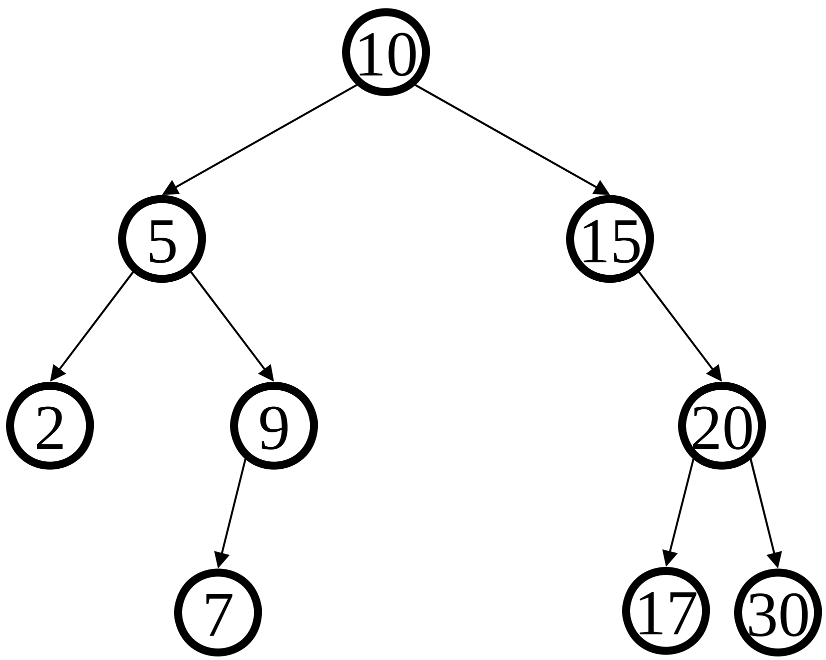



```
Node *& find(Comparable key,
             Node *& root) {
  if (root == NULL)
    return root;
  else if (key < root->key)
    return find(key,
                root->left);
  else if (key > root->key)
    return find(key,
                root->right);
  else
    return root;
}
```

runtime:

a. O(1)
b. O(lg n)
c. O(n)
d. O(n lg n)
e. None of these

Finding a Node

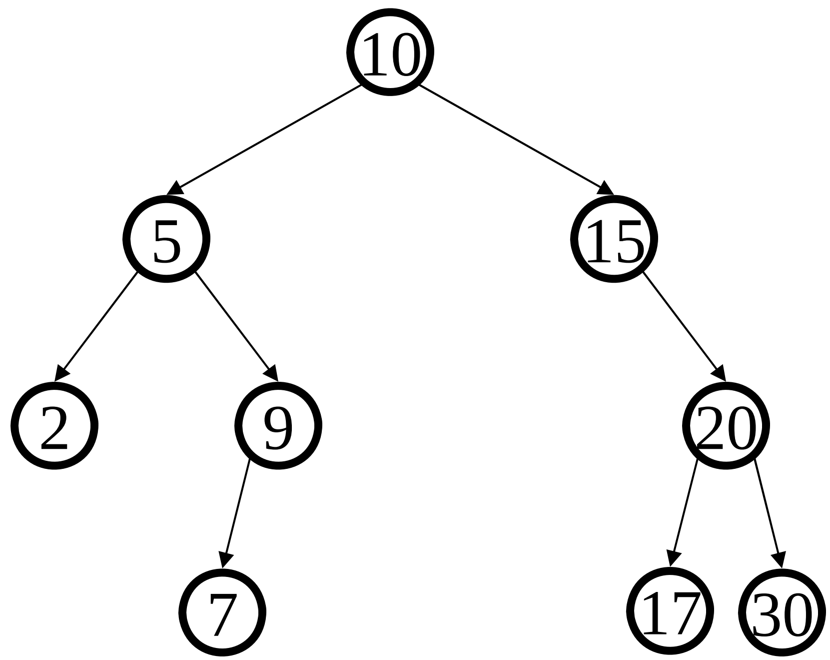



```
Node *& find(Comparable key,
                 Node *& root) {
  if (root == NULL)
    return root;
  else if (key < root->key)
    return find(key,
                root->left);
  else if (key > root->key)
    return find(key,
                root->right);
  else
    return root;
}
```

WARNING: Much fancy footwork with refs (&) coming. You can do ***all*** of this without refs... just watch out for special cases.

Iterative Find

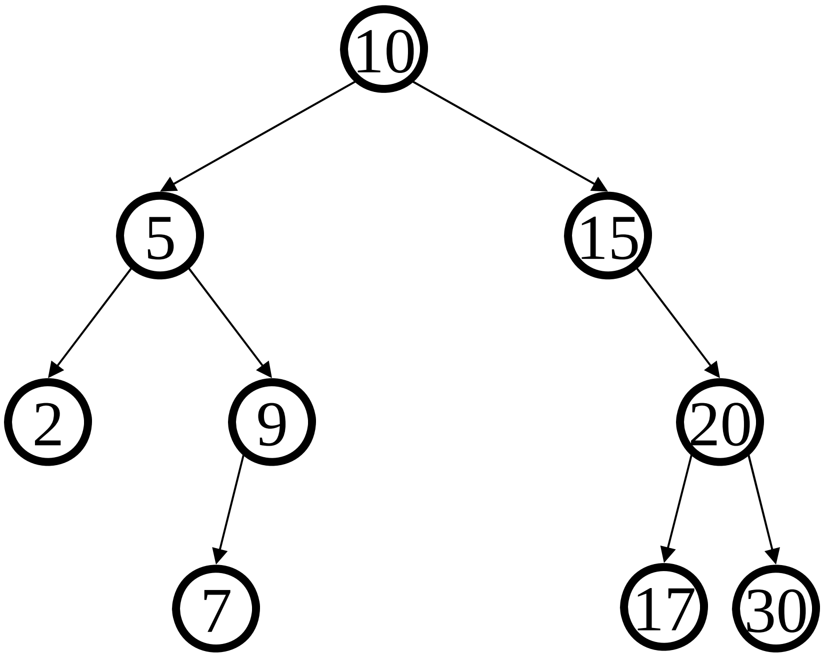
```
Node * find(Comparable key,
            Node * root) {
  while (root != NULL &&
         root->key != key) {
    if (key < root->key)
      root = root->left;
    else
      root = root->right;
  }

  return root;
}
```



Look familiar?

(It's trickier to get the ref return to work here. We won't worry.)

Insert




```
void insert(Comparable key,
                  Node *& root) {
   Node *& target(find(key,
                           root));
   assert(target == NULL);

   target = new Node(key);
}
```

runtime:

Funky game we can play with the *& version.

Reminder: Value vs. Reference Parameters

- Value parameters (Object foo)
  - copies parameter
  - no side effects
- Reference parameters (Object & foo)
  - shares parameter
  - can affect actual value
  - use when the value needs to be changed
- Const reference parameters (const Object & foo)
  - shares parameter
  - cannot affect actual value
  - use when the value is too intricate for pass-by-value

BuildTree for BSTs

- Suppose the data 1, 2, 3, 4, 5, 6, 7, 8, 9 is inserted into an initially empty BST:
  - in order
  - in reverse order
  - median first, then left median, right median, etc.

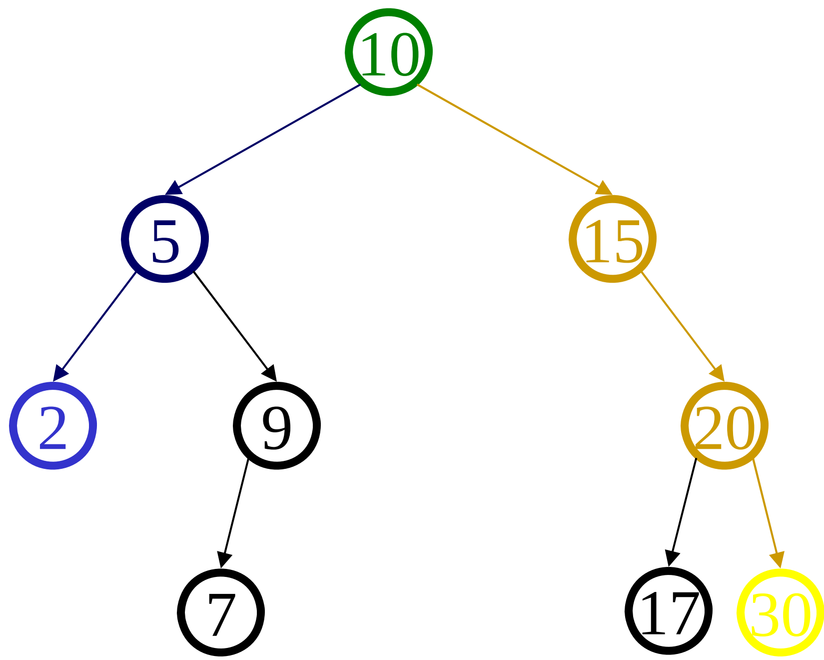
Analysis of BuildTree

- Worst case: $O(n^2)$ as we've seen
- Average case assuming all orderings equally likely turns out to be $O(n \lg n)$.

Bonus: FindMin/FindMax

- Find minimum

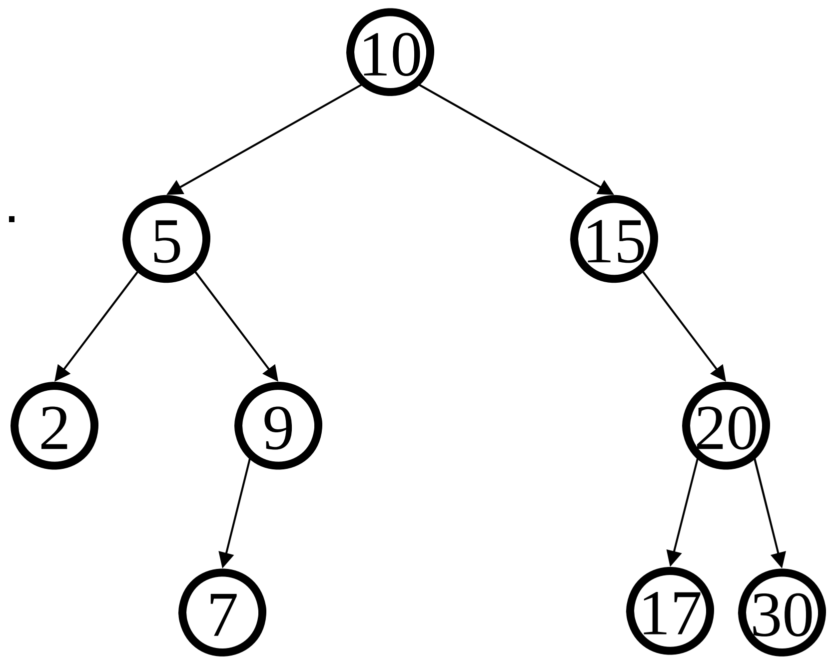
- Find maximum



Double Bonus: Successor

Find the next larger node
in this node's subtree.

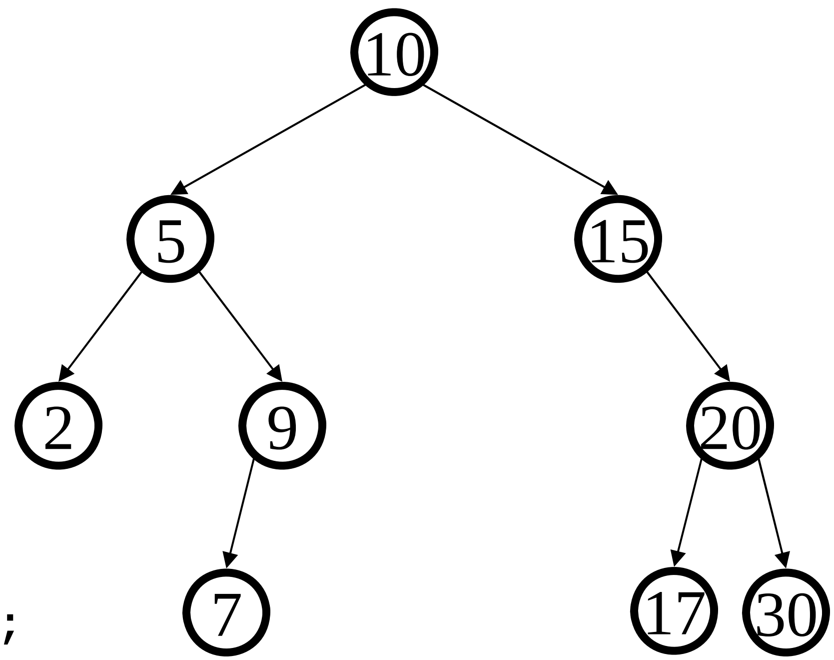
```
// Note: If no succ, returns (a useful) NULL.
Node *& succ(Node *& root) {
  if (root->right == NULL)
    return root->right;
  else
    return min(root->right);
}

Node *& min(Node *& root) {
  if (root->left == NULL) return root;
  else return min(root->left);
}
```



More Double Bonus: Predecessor

Find the next smaller node
in this node's subtree.

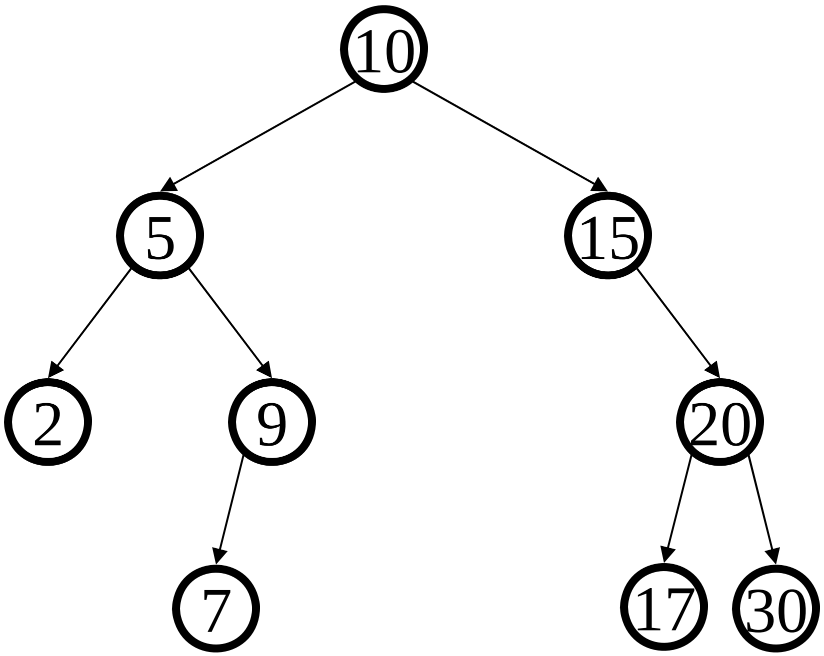
```
Node *& pred(Node *& root) {
  if (root->left == NULL)
    return root->left;
  else
    return max(root->left);
}

Node *& max(Node *& root) {
  if (root->right == NULL) return root;
  else return max(root->right);
}
```



Today's Outline

- Some Tree Review
  (here for reference, not discussed)
- Binary Trees
- Dictionary ADT
- Binary Search Trees
- Deletion
- Some troubling questions

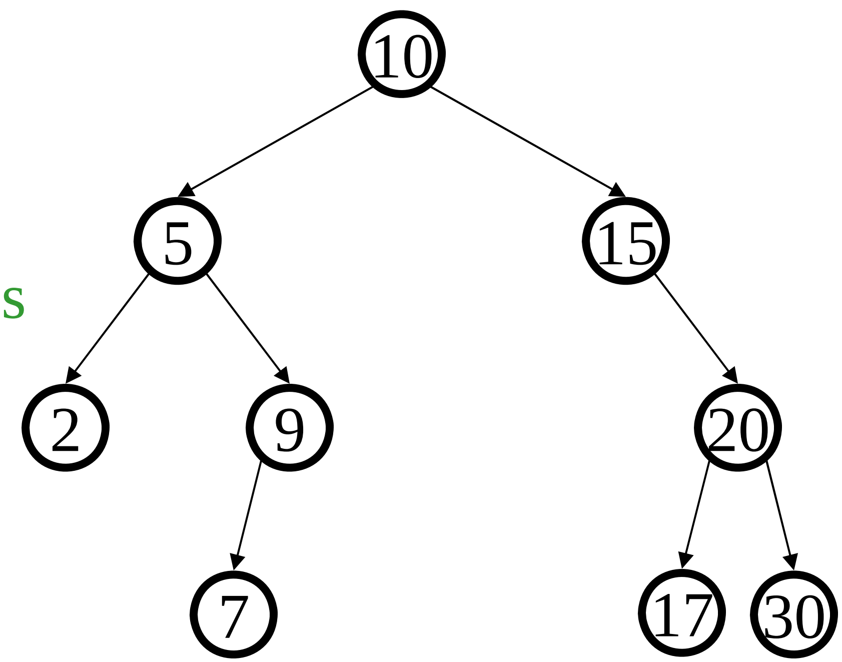
Deletion



Why might deletion be harder than insertion?

Lazy Deletion ("Tombstones")

- Instead of physically deleting nodes, just mark them as deleted
  - + simpler
  - + physical deletions done in batches
  - + some adds just flip deleted flag
  - – extra memory for "tombstone"
  - – many lazy deletions slow finds
  - – some operations may have to be modified (e.g., min and max)



Lazy Deletion

Delete(17)

Delete(15)

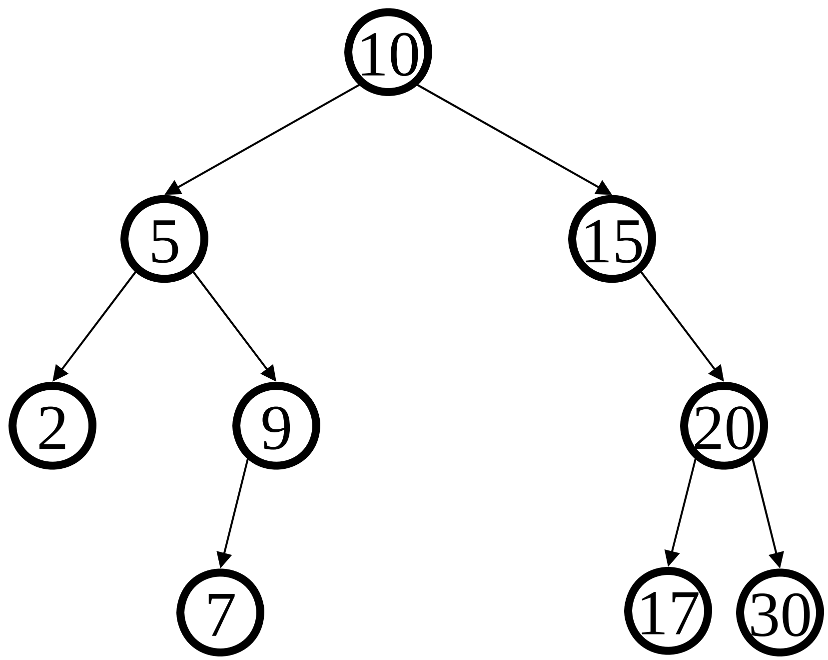
Delete(5)

Find(9)

Find(16)

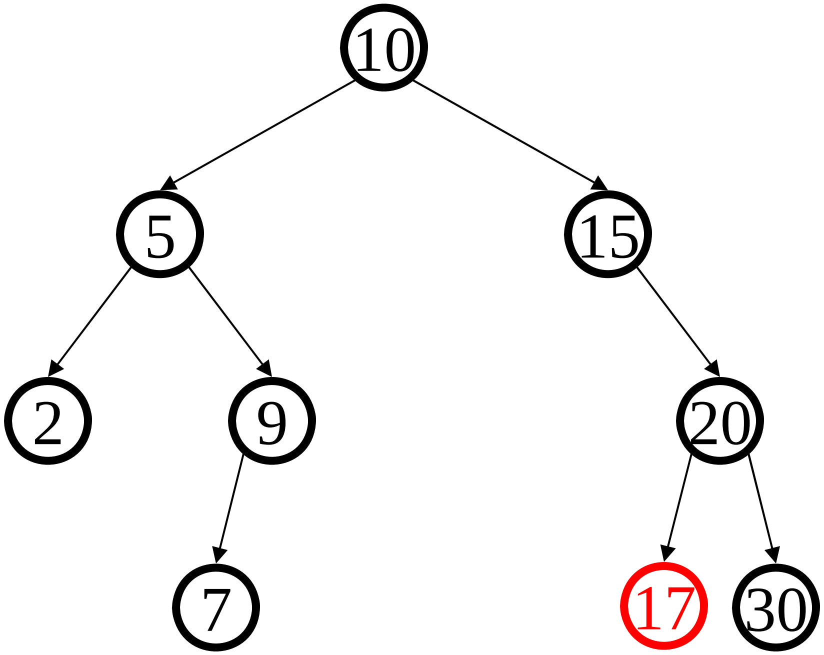
Insert(5)

Find(17)



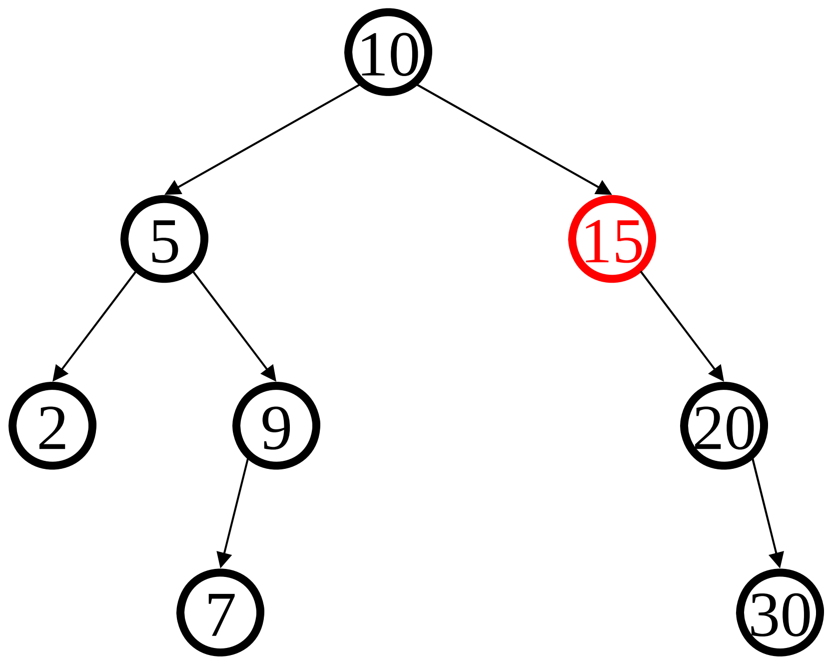
Real Deletion - Leaf Case

Delete(17)



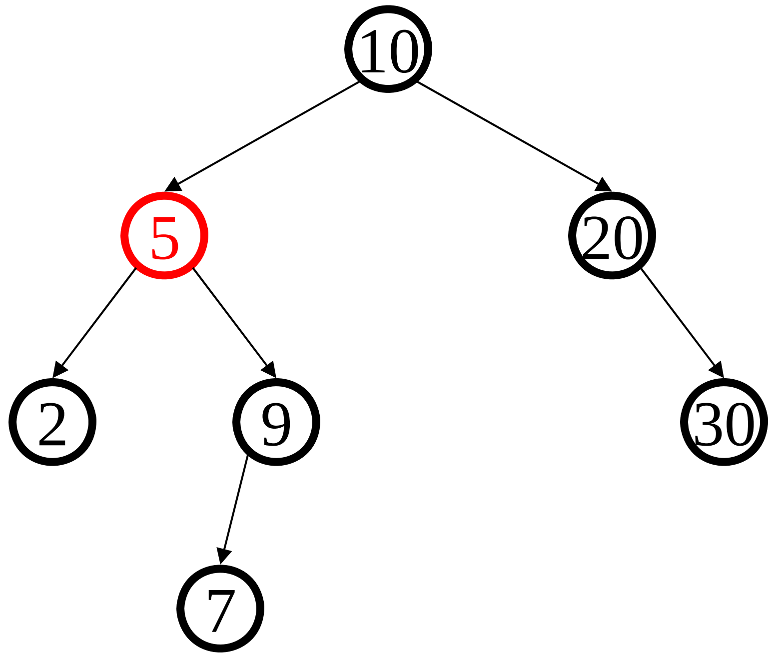
Real Deletion - One Child Case

Delete(15)

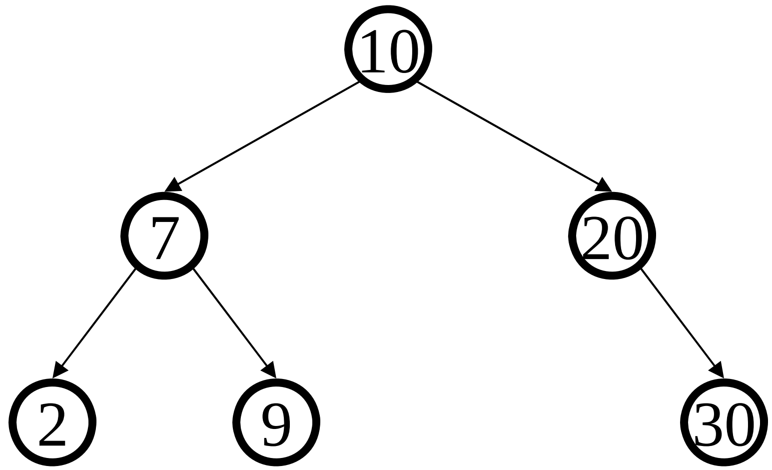


Real Deletion - Two Child Case

Delete(5)



Finally…



Delete Code

```
void delete(Comparable key, Node *& root) {
  Node *& handle(find(key, root));
  Node * toDelete = handle;
  if (handle != NULL) {
    if (handle->left == NULL) {         // Leaf or one child
      handle = handle->right;
    } else if (handle->right == NULL) { // One child
      handle = handle->left;
    } else {                            // Two child case
      Node *& successor(succ(handle));
      handle->data = successor->data;
      toDelete = successor;
      successor = successor->right;     // Succ has <= 1 child
    }
  }
  delete toDelete;
}
```

Refs make this short and "elegant"…
but could be done without them with a bit more work.

Today's Outline

- Binary Trees
- Dictionary ADT
- Binary Search Trees
- Deletion
- Some troubling questions

Thinking about Binary Search Trees

- Observations
  - Each operation views two new elements at a time
  - Elements (even siblings) may be scattered in memory
  - Binary search trees are fast *if they're shallow*
- Realities
  - For large data sets, disk accesses dominate runtime
  - Some deep and some shallow BSTs exist for any data

One more piece of bad news: what happens to a balanced tree after *many* insertions/deletions?

Solutions?

- Reduce disk accesses?

- Keep BSTs shallow?

Coming Up

- Self-balancing Binary Search Trees
- **Huge** Search Tree Data Structure