CPSC 221
Basic Algorithms and Data Structures

Binary Search Trees

Textbook References:
Koffman: 8.1 – 8.4
EPP 3rd edition:11.5
EPP 4th edition:10.5

Hassan Khosravi
January - April 2015

Borrowing many slides from Steve Wolfman

CPSC Administrative Notes

- Written Assignment 2 extension
  - Due date is changed to Friday, March 20

- Lab 7: Starting Friday, on AVL trees

- Midterms were handed back
  - Scaled Average ~69%

- PeerWise

Thanks for Your Feedback

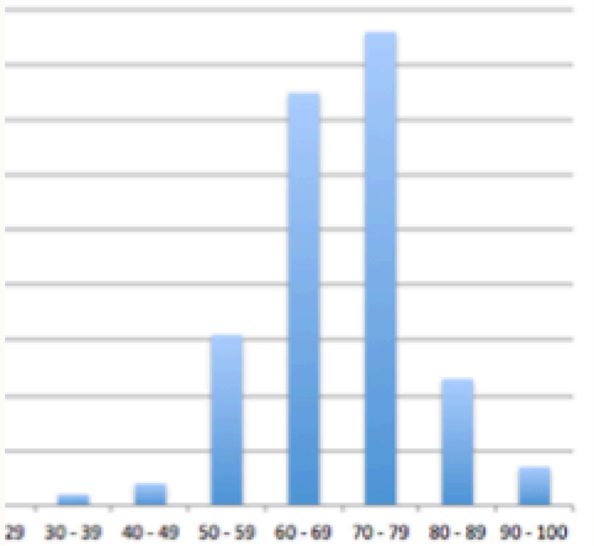
- Things that many of you commented on
  - Exam was too hard/long

Pros

- Prepares students for the final.
- Prepare students for interviews.
- Better indicator of students performance (diverse spread)
- Provides opportunity for great students to show their capabilities.

Cons

- Students get low grades and feel that the exam was unfair
  - But we can fix that by scaling!



Thanks for Your Feedback

- Things that many of you commented on
  - Too much work/better spread of assignments.
  - Sometimes unclear what students need to learn.
  - Too much theory, math, and proofs.
  - Unclear how much C++ we should know
  - Labs and assignments are disjoint from lecture!!!!
  - 2 people hated my ppt background ☹

- You can always submit anonymous feedback through my website.

PeerWise Claims

- Generating a question requires students to think carefully about the topics of the course and how they relate to the learning outcomes.
- Writing questions focuses attention on the learning outcomes and makes teaching and learning goals more apparent to students.
- The act of creating plausible distracters (multiple-choice alternatives) requires students to consider misconceptions
- Explanations require students to express their understanding of a topic with as much clarity as possible.
- Answering questions in a drill and practice fashion reinforces learning, and incorporates elements of self-assessment.

PeerWise

- Authoring and answering questions on PeerWise has helped me better learn and understand the material, which is covered in CPSC 221.

A: Strongly Agree

B: Agree

C: Neutral

D: Disagree

E: Strongly Disagree

PeerWise

- I think

A: PeerWise should be used in future offerings of 221

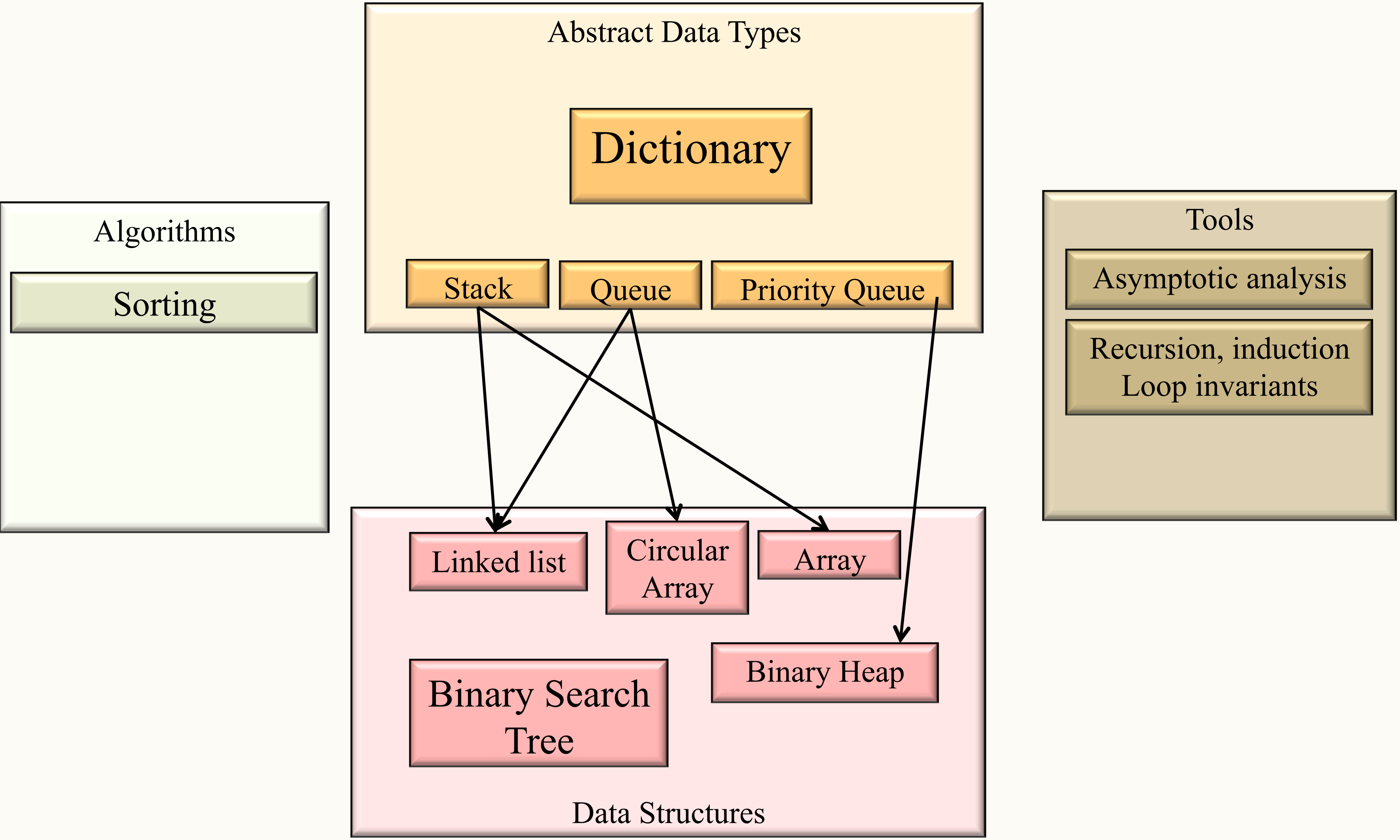
B: PeerWise should NOT be used in future offerings of 221

C: Neutral

Learning Goals

- 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

CPSC 221 Journey



Abstract Data Types

Dictionary

Stack

Queue

Priority Queue

Algorithms

Sorting

Tools

Asymptotic analysis

Recursion, induction
Loop invariants

Linked list

Circular Array

Array

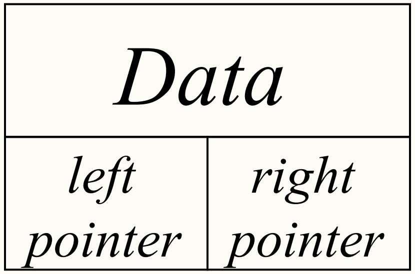
Binary Search Tree

Binary Heap

Data Structures

Binary Trees

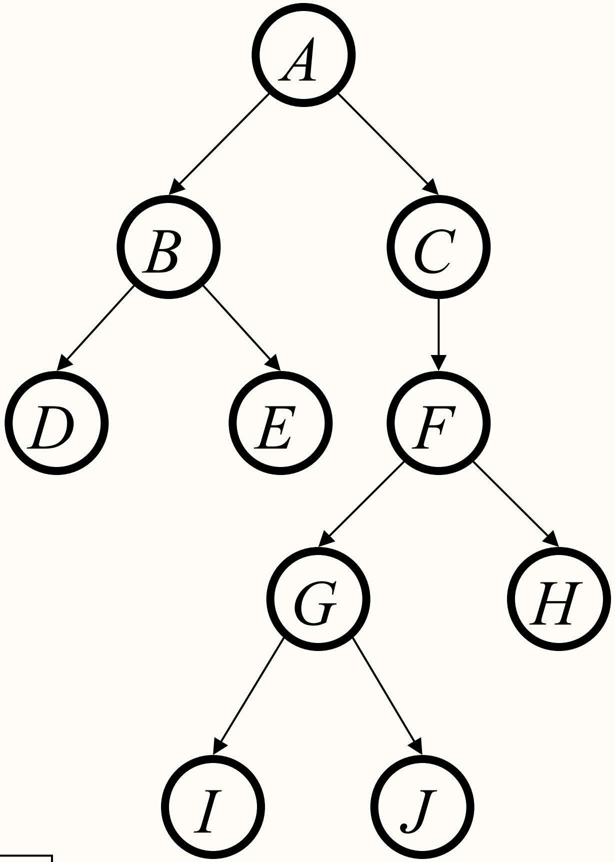
- Binary tree is either
  - empty (NULL for us), or
  - a datum, a left subtree, and a right subtree
- Properties
  - max # of leaves: 2^h
  - max # of nodes: 2^{h+1}
- Representation:



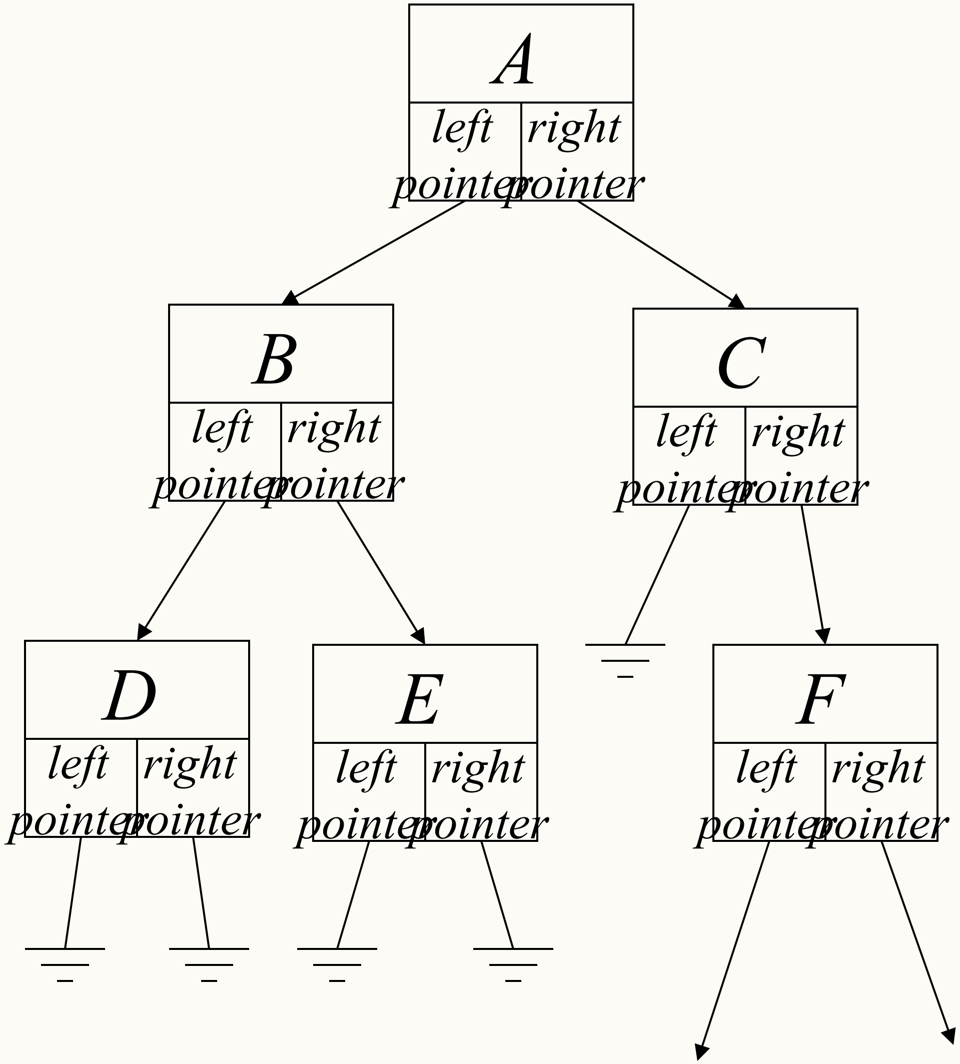
Data

left pointer

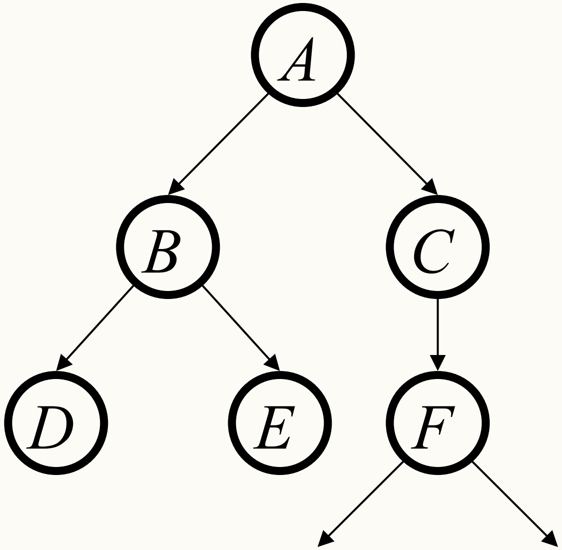
right pointer



Representation



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



Tree Traversal

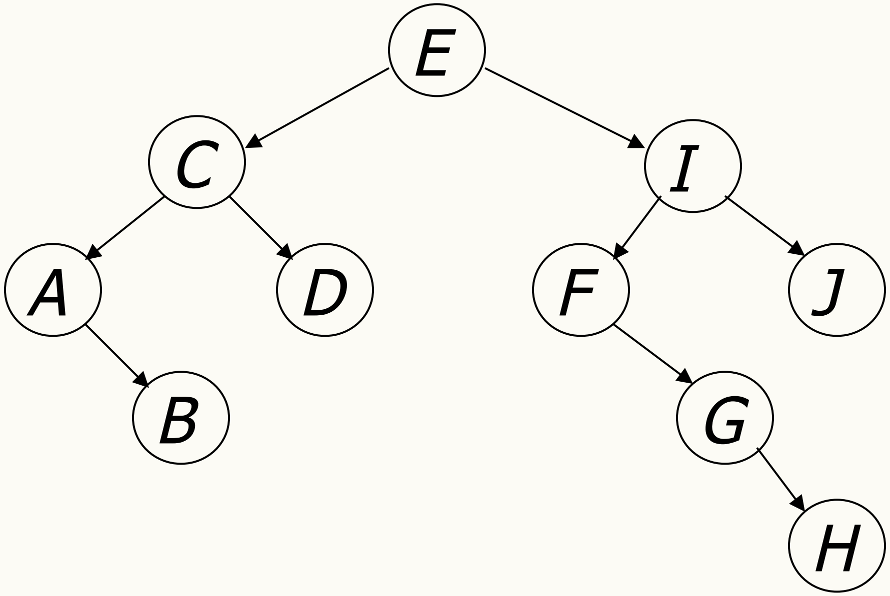
There are three common types of binary tree traversal:

Preorder: visit the current node, then its left sub-tree, then its right sub-tree

Inorder: visit the left sub-tree, then the current node, then the right sub-tree

Postorder: visit the left sub-tree, then the right sub-tree, then the current node

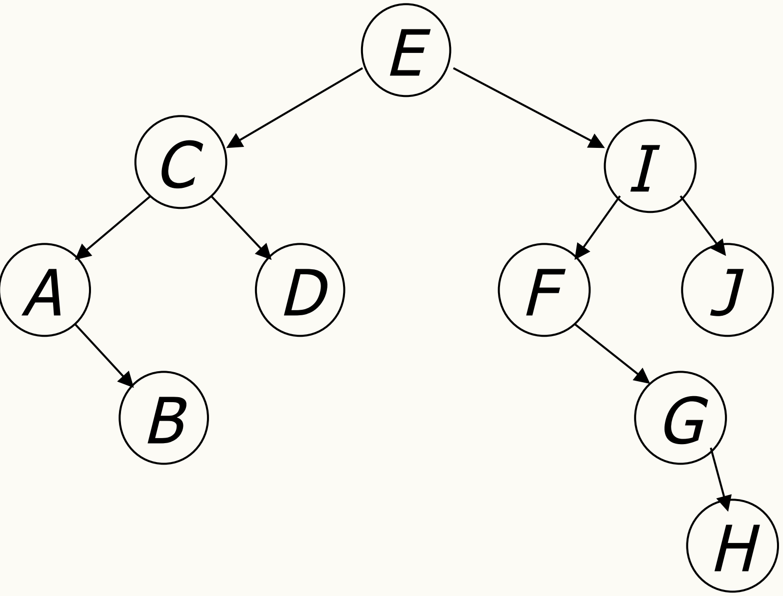
Preorder: visit the current node, then its left sub-tree, then its right sub-tree



Data printed using preorder traversal:

E C A B D I F G H J

Preorder: visit the current node, then its left sub-tree, then its right sub-tree

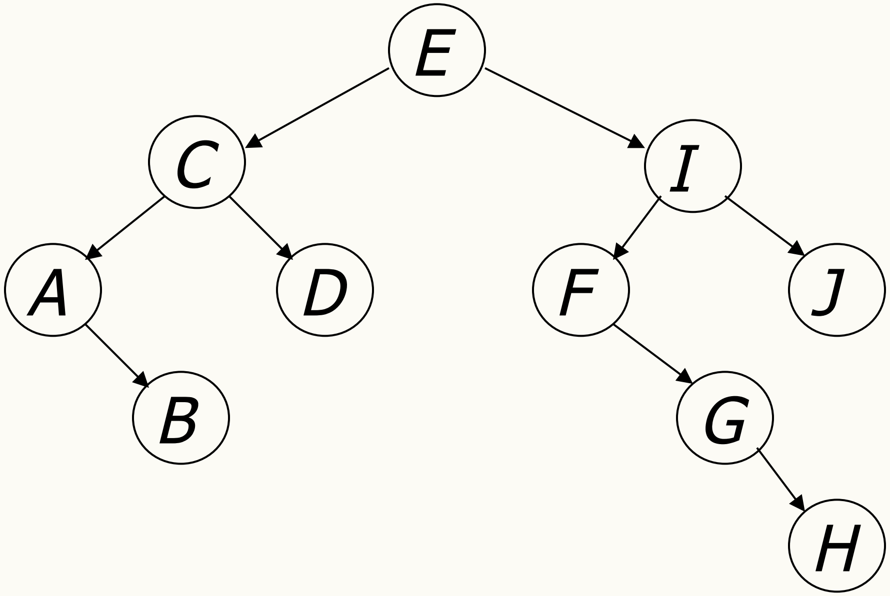


```
void printPreorder( Node*& node) {
  if (! node) return;

  /* first print data of node */
  std::cout<< node->data;

  /* then recur on left sutree */
  printPreorder(node->left);

  /* now recur on right subtree */
  printPreorder(node->right);
}
```

Data printed using *preorder* traversal:

E C A B D I F G H J

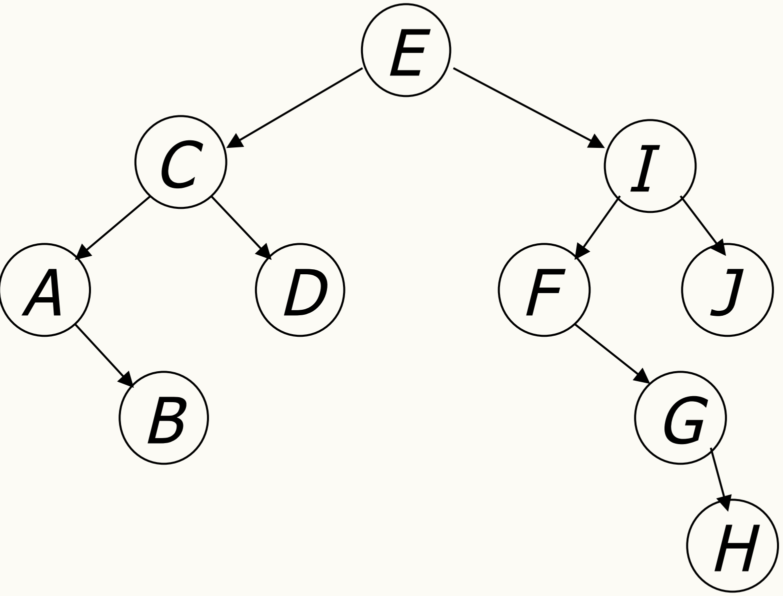
<u>Inorder</u>: visit the left sub-tree, then the current node, then the right sub-tree



Data printed using <u>inorder</u> traversal:

A B C D E F G H I J

Inorder: *visit the left sub-tree, then the current node, then the right sub-tree*

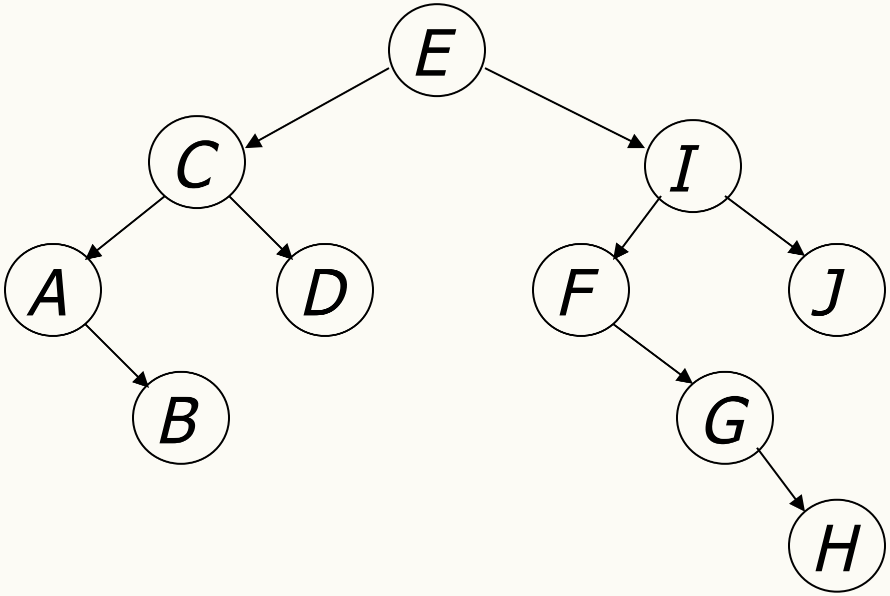


```
void printInorder( Node*& node) {
  if (! node) return;

  /* then recur on left sutree */
  printInorder(node->left);

  /* first print data of node */
  std::cout<< node->data;

  /* now recur on right subtree */
  printInorder(node->right);
}
```

Data printed using inorder traversal:

A B C D E F G H I J

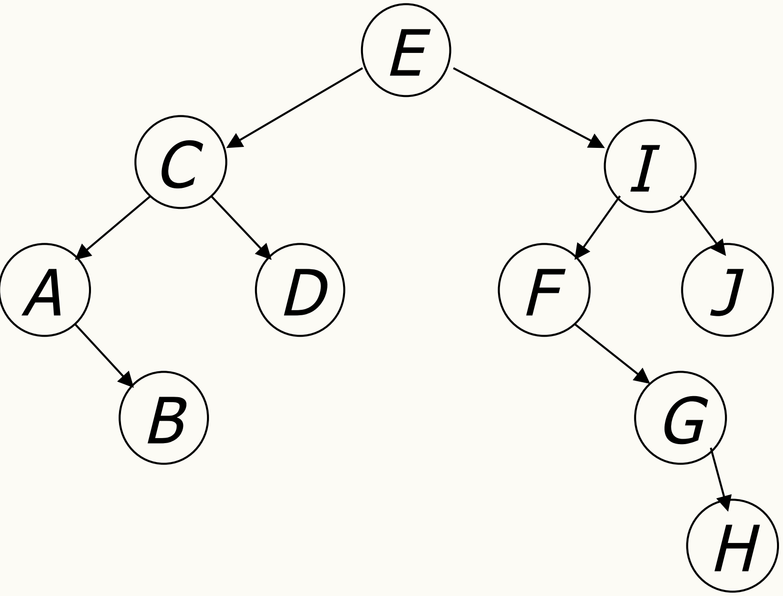
Postorder: *visit the left sub-tree, then the right sub-tree, then the current node*



Data printed using postorder traversal:

B A D C H G F J I E

Postorder: visit the left sub-tree, then the right sub-tree, then the current node

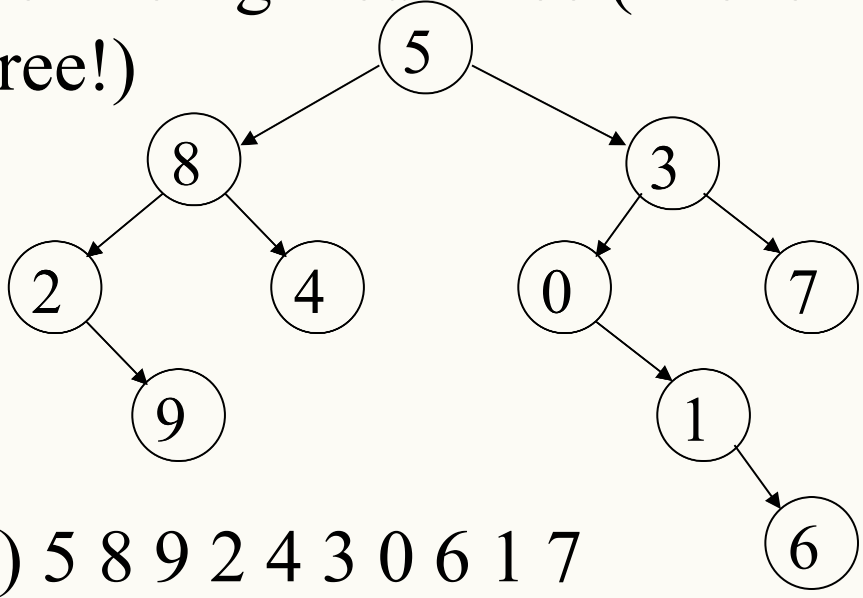


```
void printPostorder( Node*& node) {
  if (! node) return;

  /* then recur on left sutree */
  printPostorder(node->left);

  /* now recur on right subtree */
  printPostorder(node->right);

  /* first print data of node */
  std::cout<< node->data;
}
```

Data printed using postorder traversal:

B A D C H G F J I E

<u>Preorder</u>: visit the current node, then its left sub-tree, then its right sub-tree (this is NOT a Binary Search Tree!)



Nodes visited using <u>preorder</u> traversal:

a) 5 8 9 2 4 3 0 6 1 7

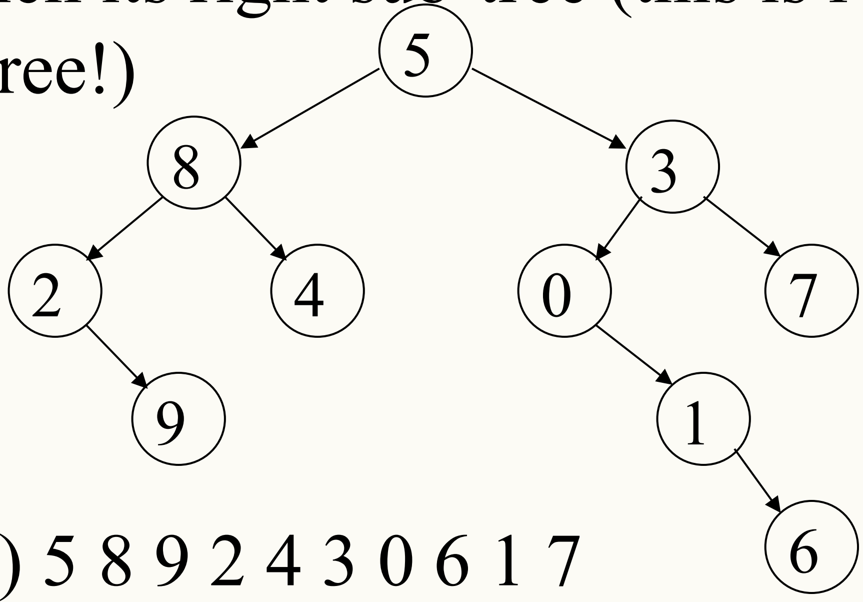
b) 5 8 2 9 4 3 0 1 6 7

c) 5 8 3 2 4 0 7 9 1 6

d) 6 1 0 7 3 5 9 2 4 8

e) 9 2 4 8 6 1 0 7 3 5

Preorder: visit the current node, then its left sub-tree, then its right sub-tree (this is NOT a Binary Search Tree!)



Nodes visited using preorder traversal:

a) 5 8 9 2 4 3 0 6 1 7

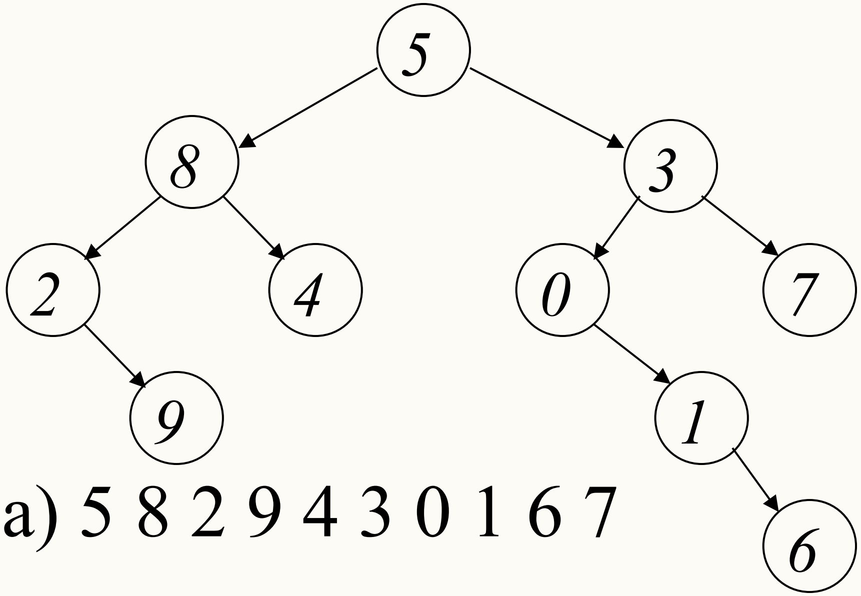
b) 5 8 2 9 4 3 0 1 6 7

c) 5 8 3 2 4 0 7 9 1 6

d) 6 1 0 7 3 5 9 2 4 8

e) 9 2 4 8 6 1 0 7 3 5

Inorder: visit the left sub-tree, then the current node, then the right sub-tree (this is NOT a BST!)



Nodes visited using inorder traversal:

a) 5 8 2 9 4 3 0 1 6 7

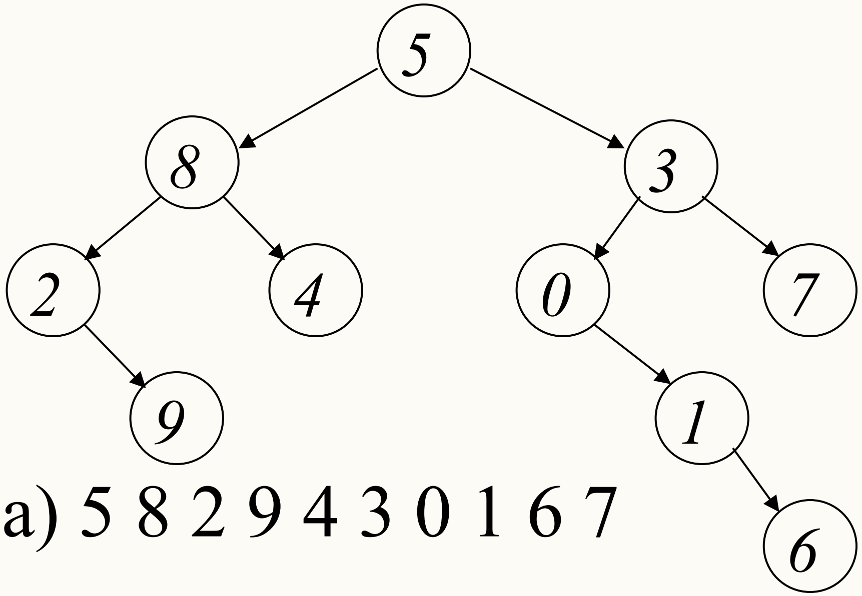
b) 5 8 3 2 4 0 7 9 1 6

c) 6 1 0 7 3 5 9 2 4 8

d) 2 9 8 4 5 0 1 6 3 7

e) 9 2 4 8 6 1 0 7 3 5

Inorder: visit the left sub-tree, then the current node, then the right sub-tree (this is NOT a BST!)



Nodes visited using inorder traversal:

a) 5 8 2 9 4 3 0 1 6 7

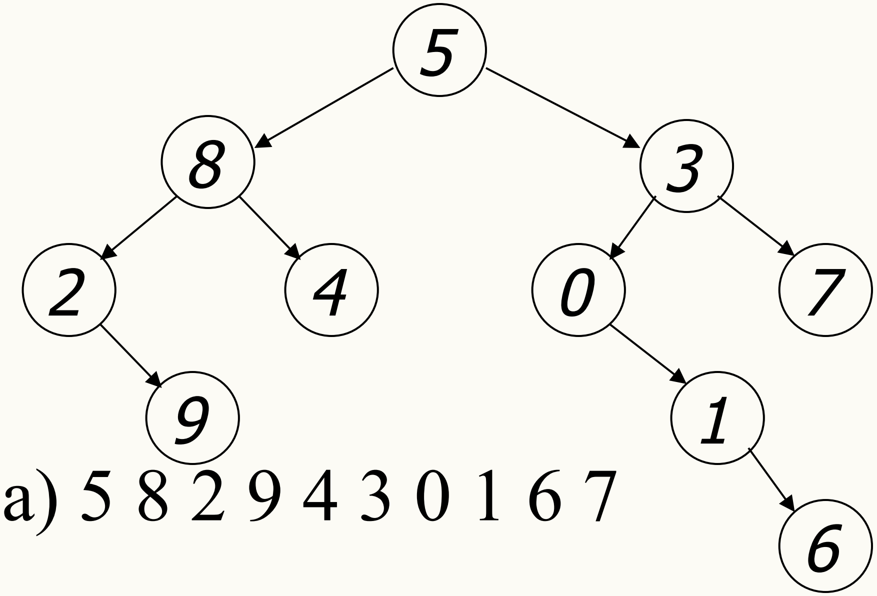
b) 5 8 3 2 4 0 7 9 1 6

c) 6 1 0 7 3 5 9 2 4 8

d) 2 9 8 4 5 0 1 6 3 7

e) 9 2 4 8 6 1 0 7 3 5

Postorder: visit the left sub-tree, then the right sub-tree, then the current node (this is NOT a BST!)



Nodes visited using postorder traversal:

a) 5 8 2 9 4 3 0 1 6 7

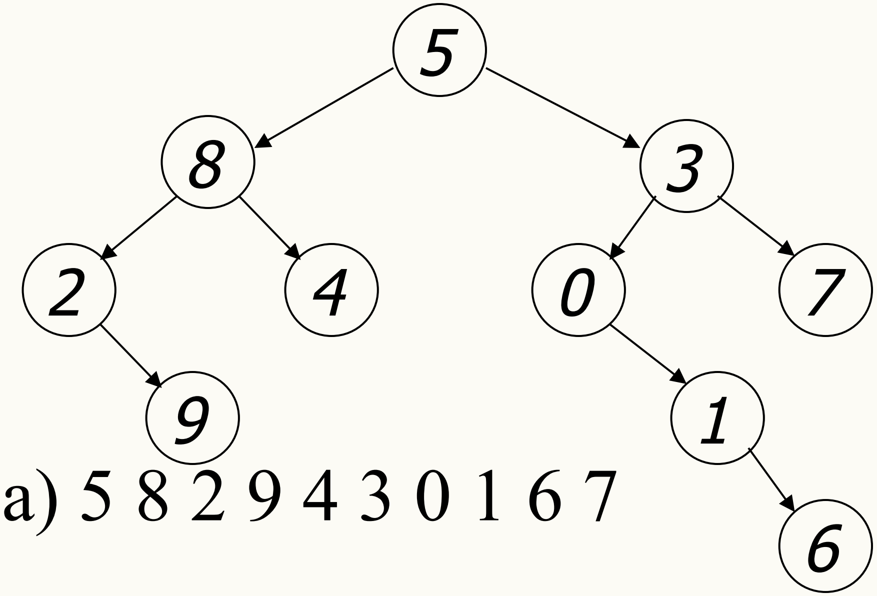
b) 2 9 8 4 0 1 6 7 3 5

c) 6 1 0 7 3 5 9 2 4 8

d) 9 2 4 8 6 1 0 7 3 5

e) 2 9 8 4 5 0 1 6 3 7

Postorder: visit the left sub-tree, then the right sub-tree, then the current node (this is NOT a BST!)



Nodes visited using postorder traversal:

a) 5 8 2 9 4 3 0 1 6 7

b) 2 9 8 4 0 1 6 7 3 5

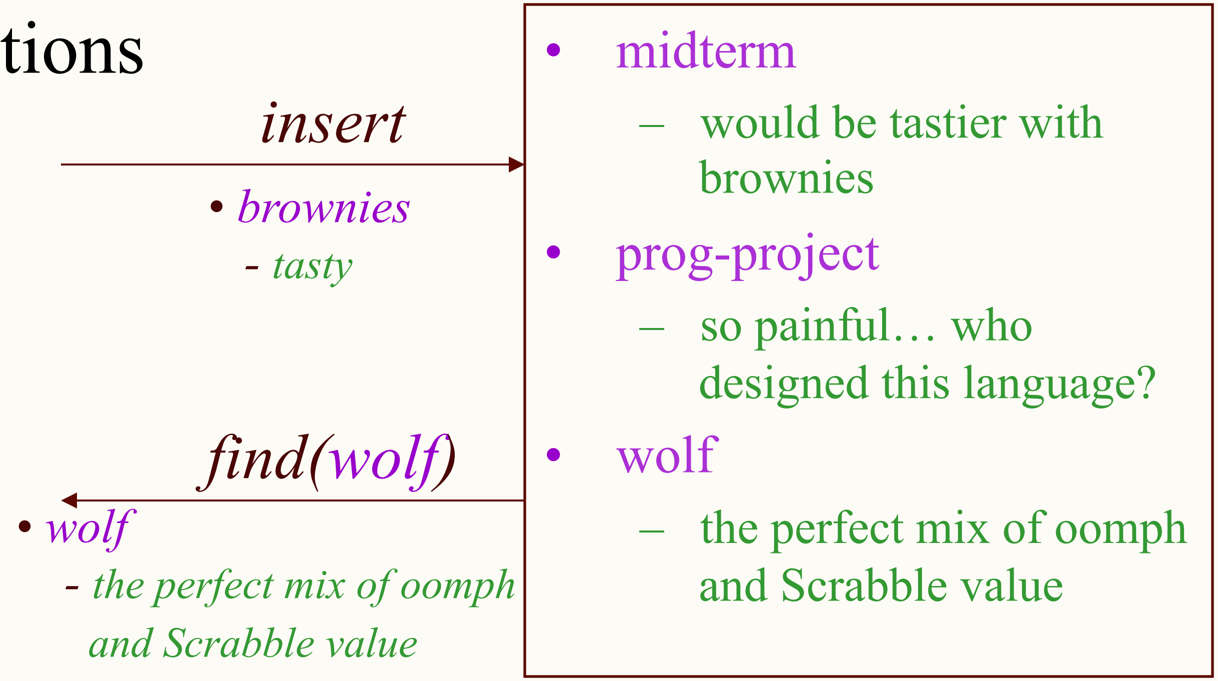
c) 6 1 0 7 3 5 9 2 4 8

d) 9 2 4 8 6 1 0 7 3 5

e) 2 9 8 4 5 0 1 6 3 7

Dictionary ADT

- Dictionary operations
  - create
  - destroy
  - insert
  - find
  - Delete



insert

- *brownies*
  - *tasty*

find(wolf)

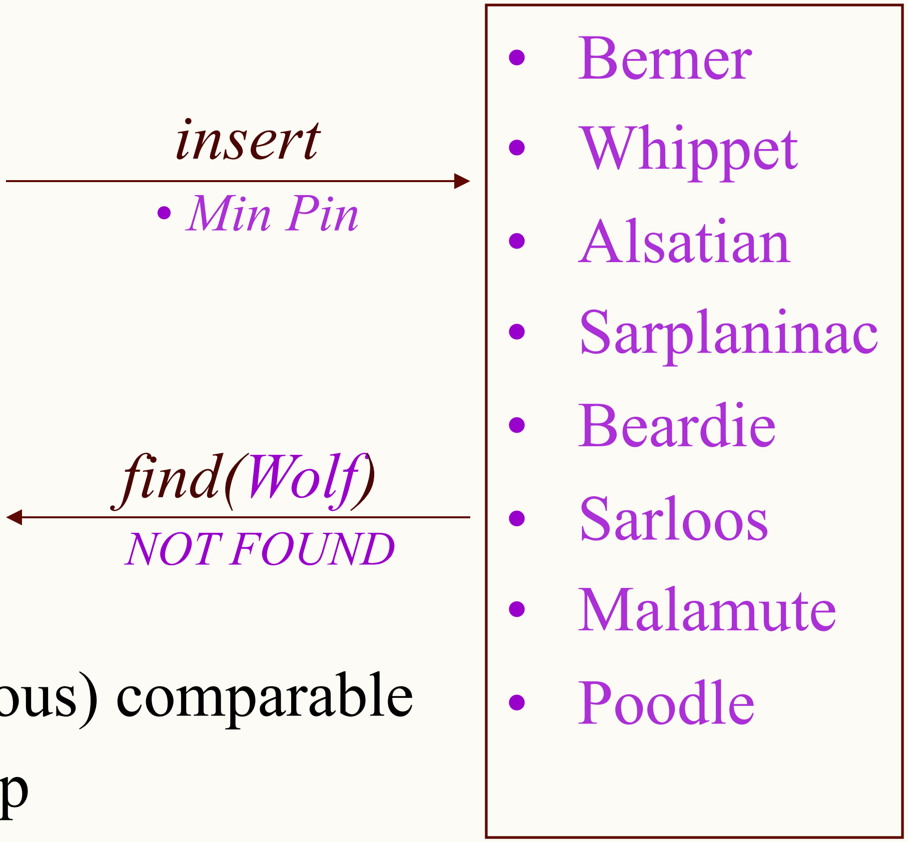
- *wolf*
  - *the perfect mix of oomph and Scrabble value*

- midterm
  - would be tastier with brownies
- prog-project
  - so painful… who designed this language?
- wolf
  - the perfect mix of oomph and Scrabble value

- 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



insert

- *Min Pin*

find(Wolf)

NOT FOUND

- Berner
- Whippet
- Alsatian
- Sarplaninac
- Beardie
- Sarloos
- Malamute
- Poodle

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

A Modest Few Uses

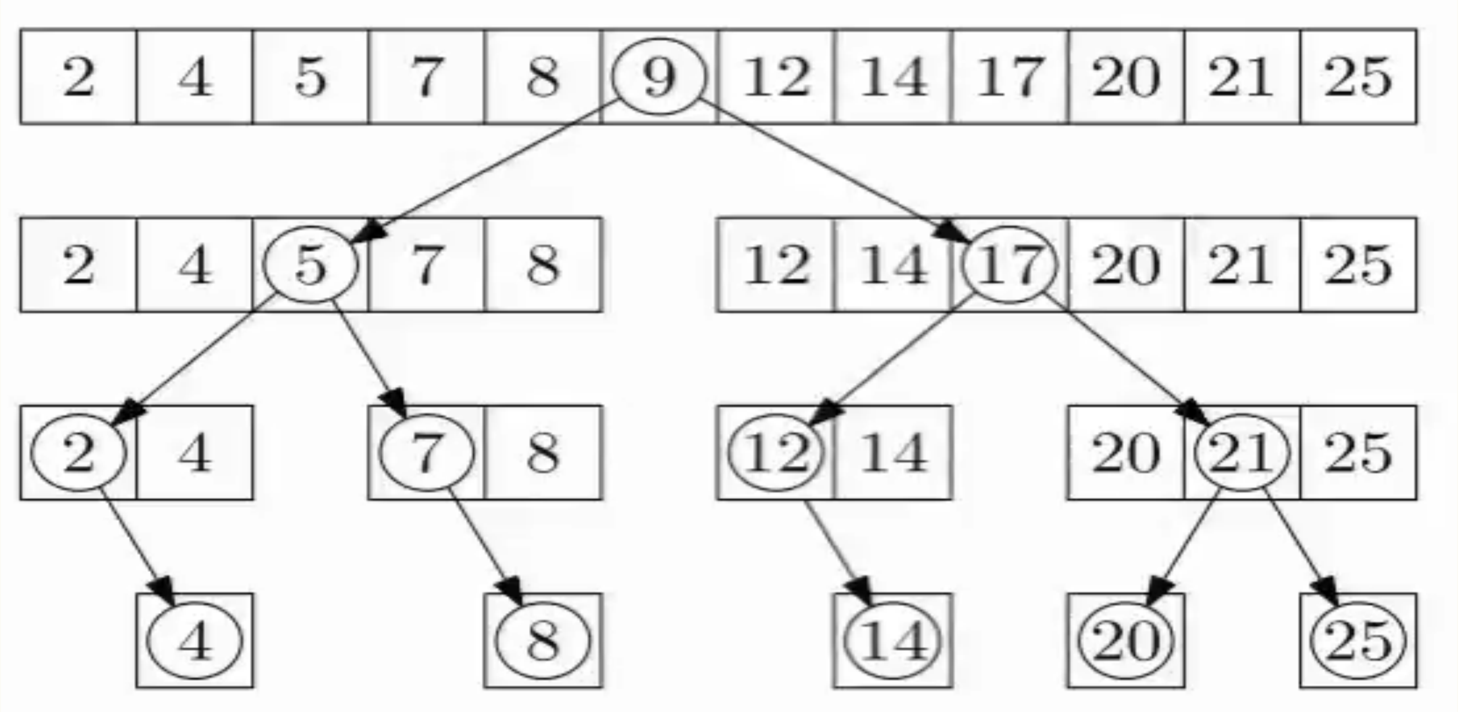
- Arrays and “Associative” Arrays
- Sets
- Dictionaries
- Router tables
- Page tables
- Symbol tables
- C++ Structures

Naïve Implementations

	insert	*find*	*delete + find*	*delete after find*
• Linked list				
– Unsorted	$O(1)$	$O(n)$	$O(n)$	$O(1)$
– Sorted	$O(n)$	$O(n)$	$O(n)$	$O(1)$
• Array				
– Unsorted	$O(1)$	$O(n)$	$O(n)$	$O(1)$
– Sorted	$O(n)$	$O(\lg n)$	$O(n)$	$O(n)$

worst one… yet so close!

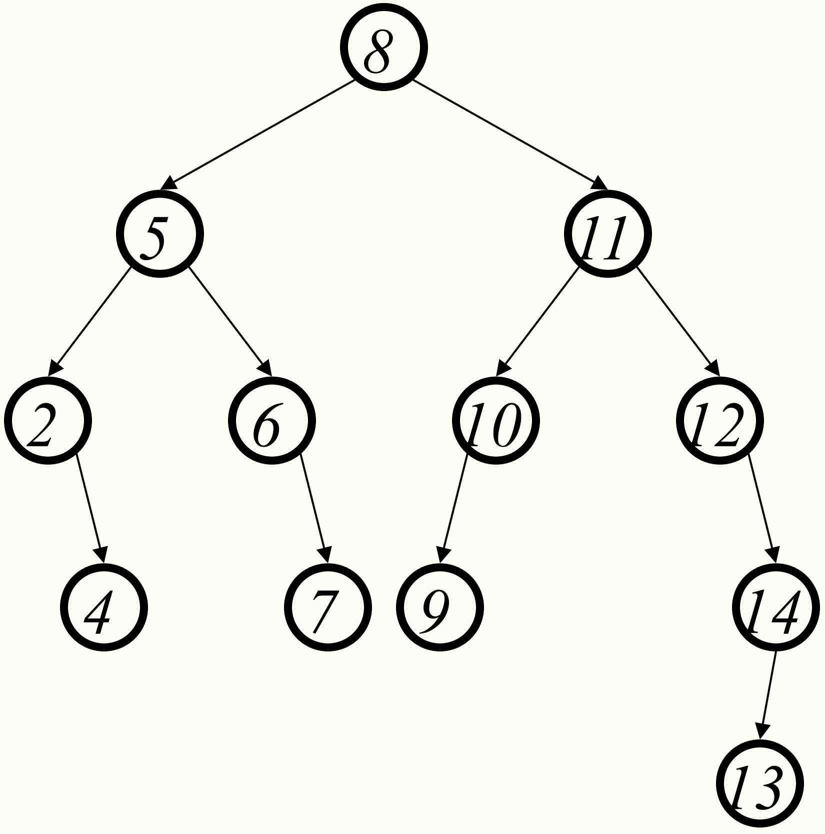
Binary Search into Binary Search Trees



```
/* Search an array, recursively, for a given search key. */
int  search( int array[], int key, int low_index, int high_index ){
    int mid = ( low_index + high_index ) / 2;

    if ( high_index < low_index )  return -1;
    if ( array[mid] > key ) return search(array, key, low_index, mid-1);
    else if ( array[mid] < key) /* search right half of array */
        return search( array, key, mid + 1, high_index );
    else   return mid;
}
```

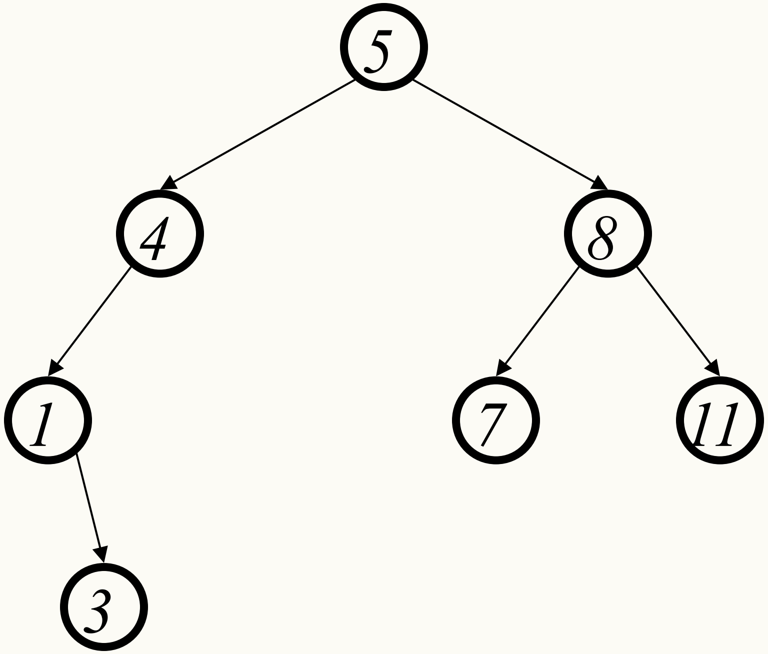
Binary Search Tree

- Binary tree property
  - each node has ≤ 2 children
  - result:
    - operations are simple
- 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

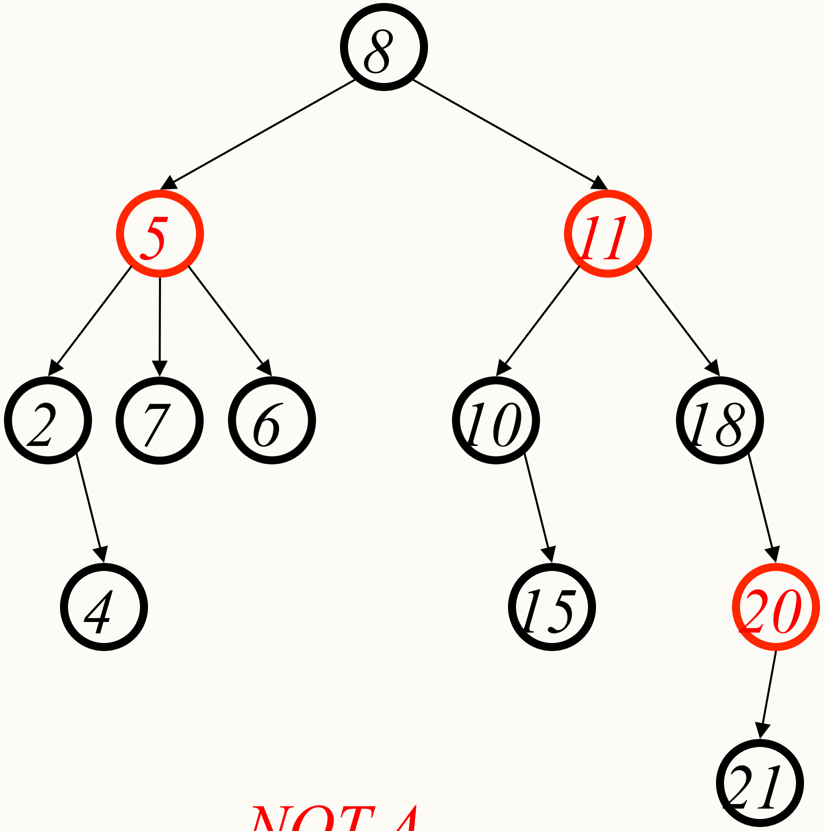


Getting to Know BSTs

Example and Counter-Example



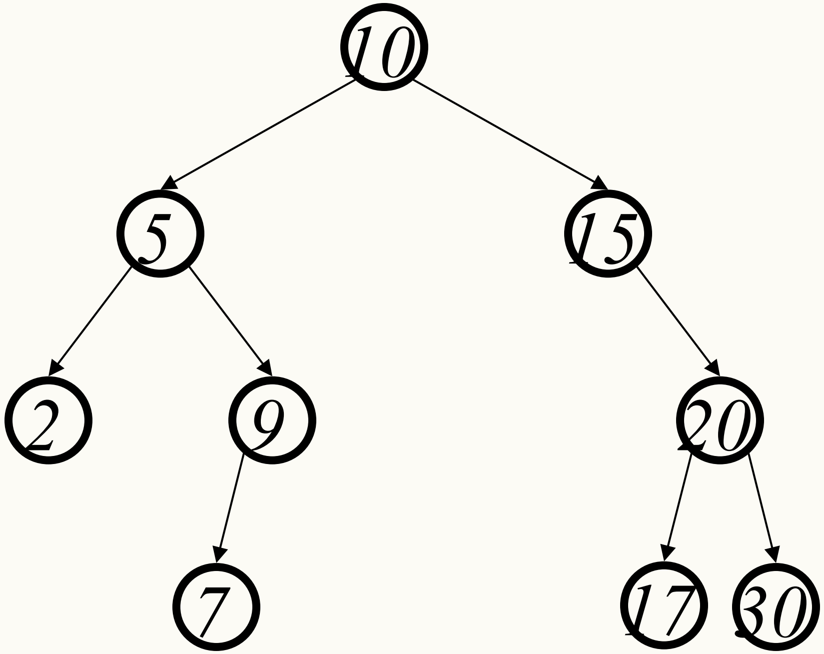
BINARY SEARCH TREE



NOT A
BINARY SEARCH TREE

Getting to Like BSTs

Finding a Node

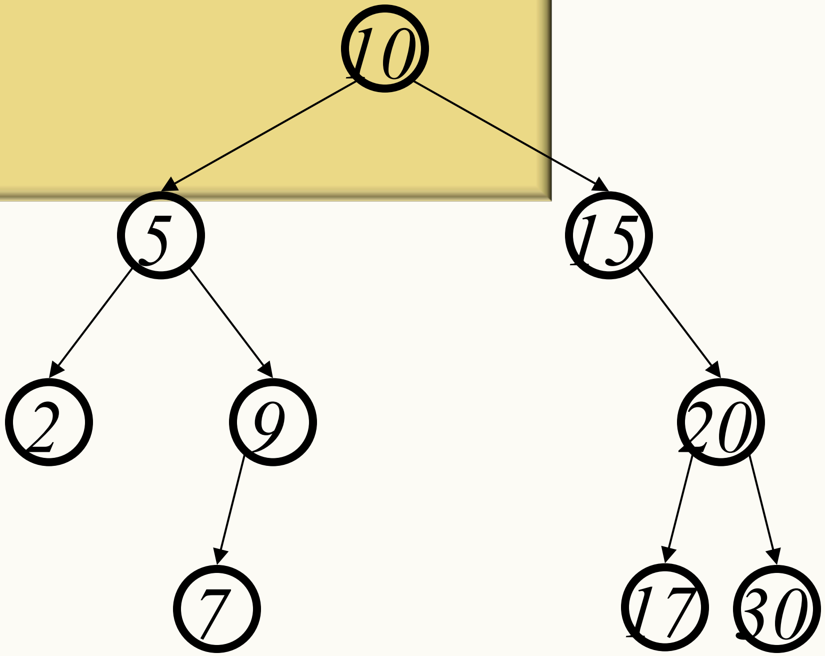


runtime:

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;
}
```

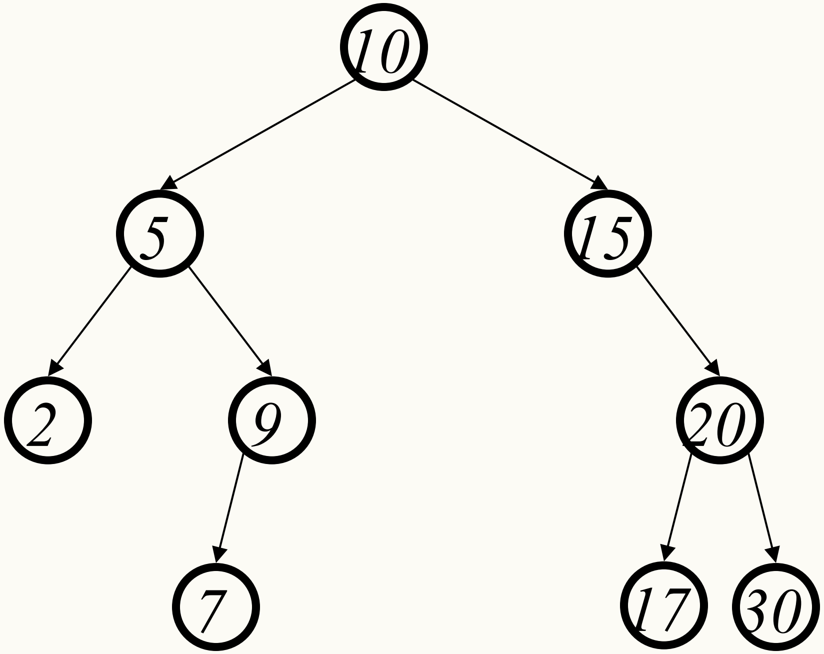


What does find 20 return?

What does find 8 return?

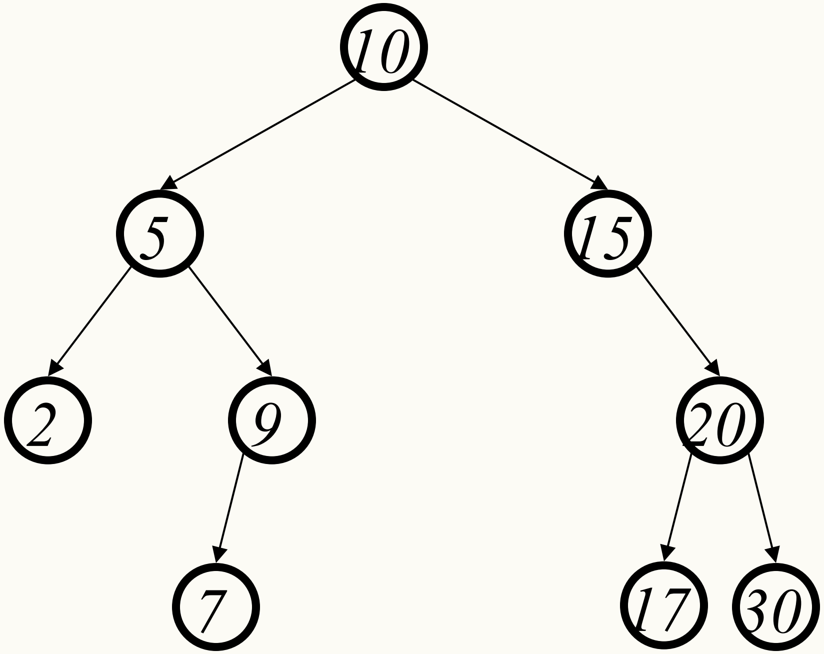
Iterative Find

```
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;
}
```



(It's trickier to get the ref return to work here.)

Insert



```
// Precondition: key is not
// already in the tree!
void insert(Comparable key,
            Node * root) {
  Node *& target(find(key,root));
  assert(target == NULL);

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

insert(8)
insert (11)
insert(31)

*Funky game we can play with the *& version.*

Runtime?

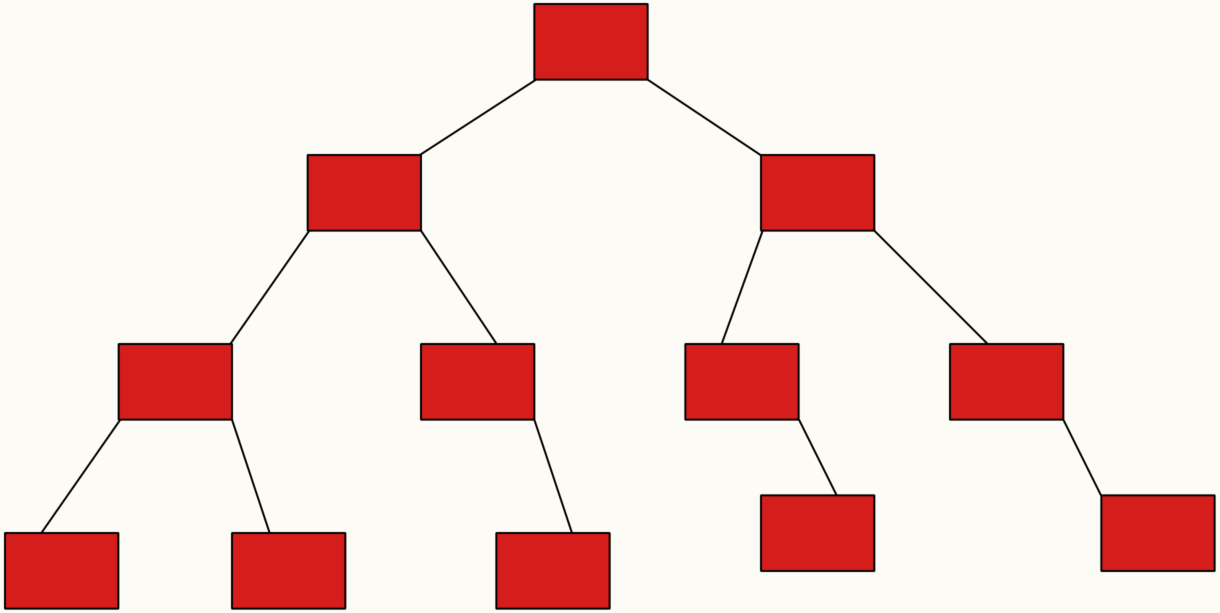
Digression: Value vs. Reference Parameters

- 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 (Object const & foo)
  - shares parameter
  - cannot affect actual value
  - use when the value is too big to be passed-by-value

BuildTree for BSTs

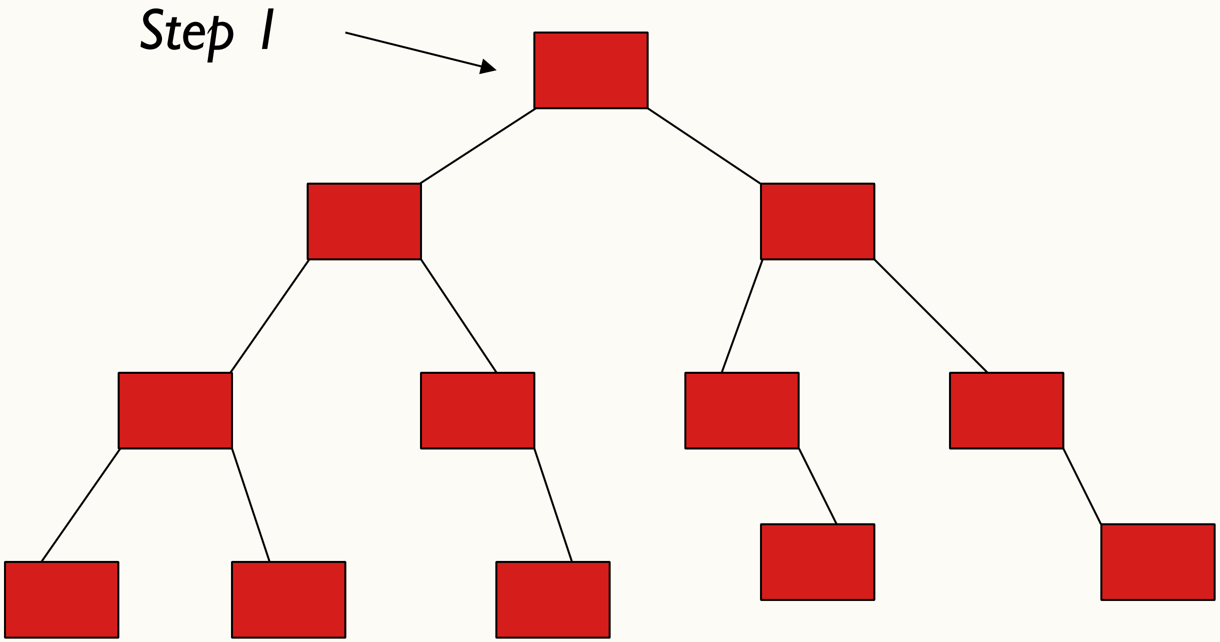
- Suppose the data 1, 2, 3, 4, 5, 6, 7, 8, 9 is inserted into an initially empty BST:
  - in order
  - in reverse order
  - median first, then left median, right median, etc.
    so: 5, 3, 8, 2, 4, 7, 9, 1, 6

What makes a balanced BST efficient for searching?



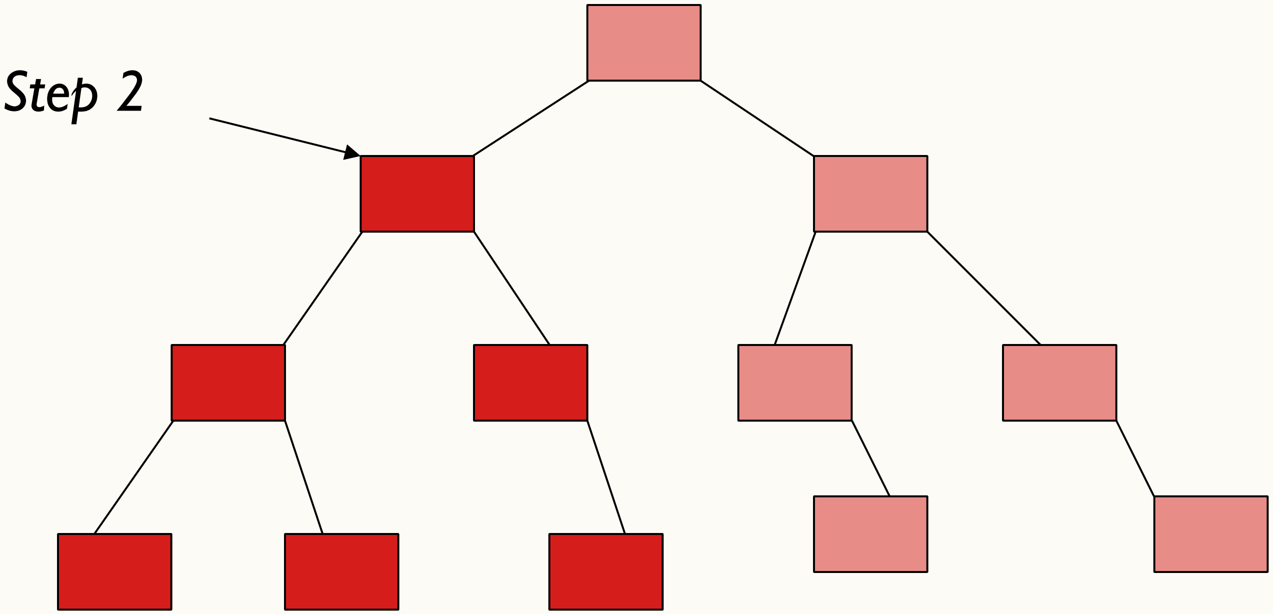
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



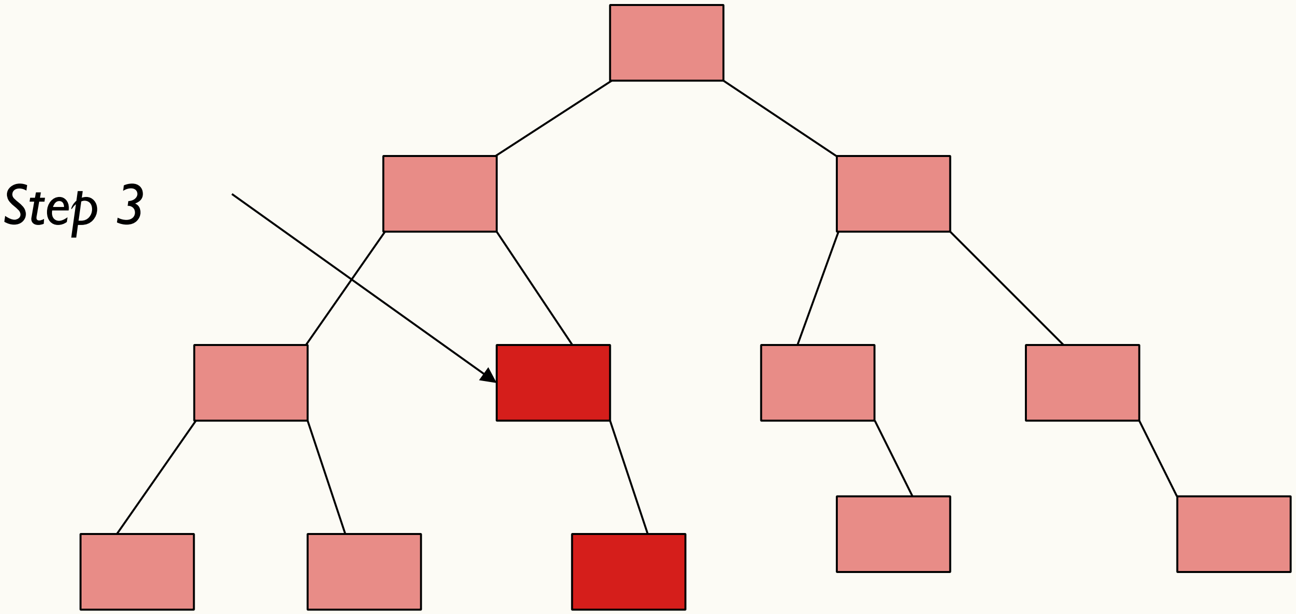
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



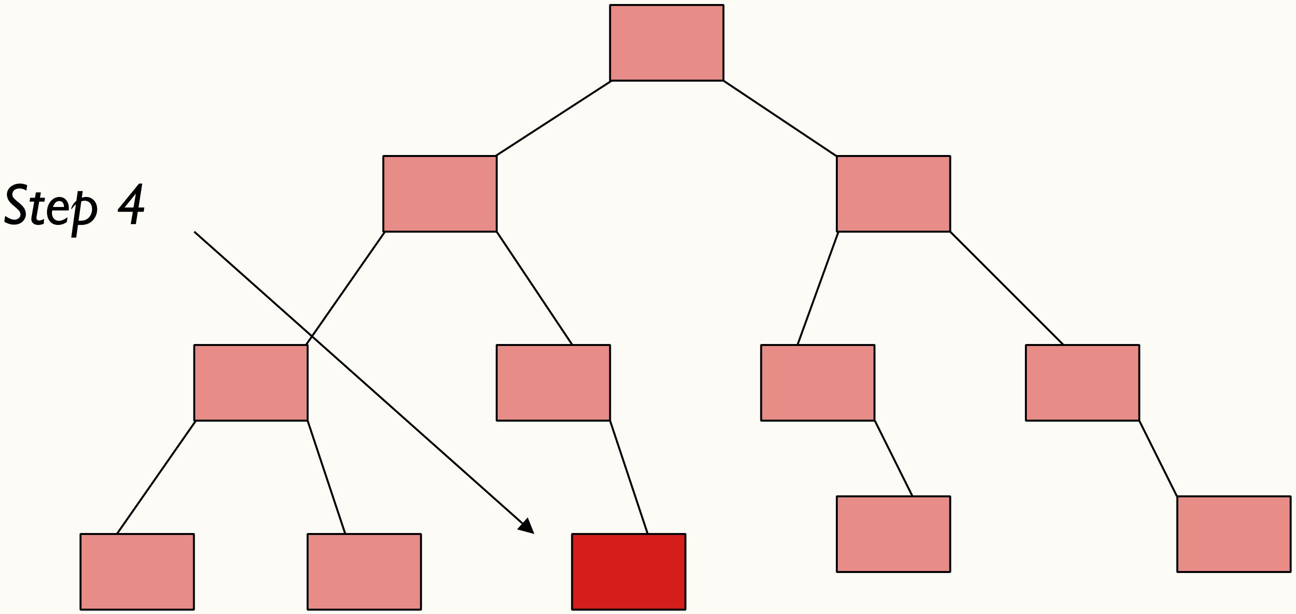
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



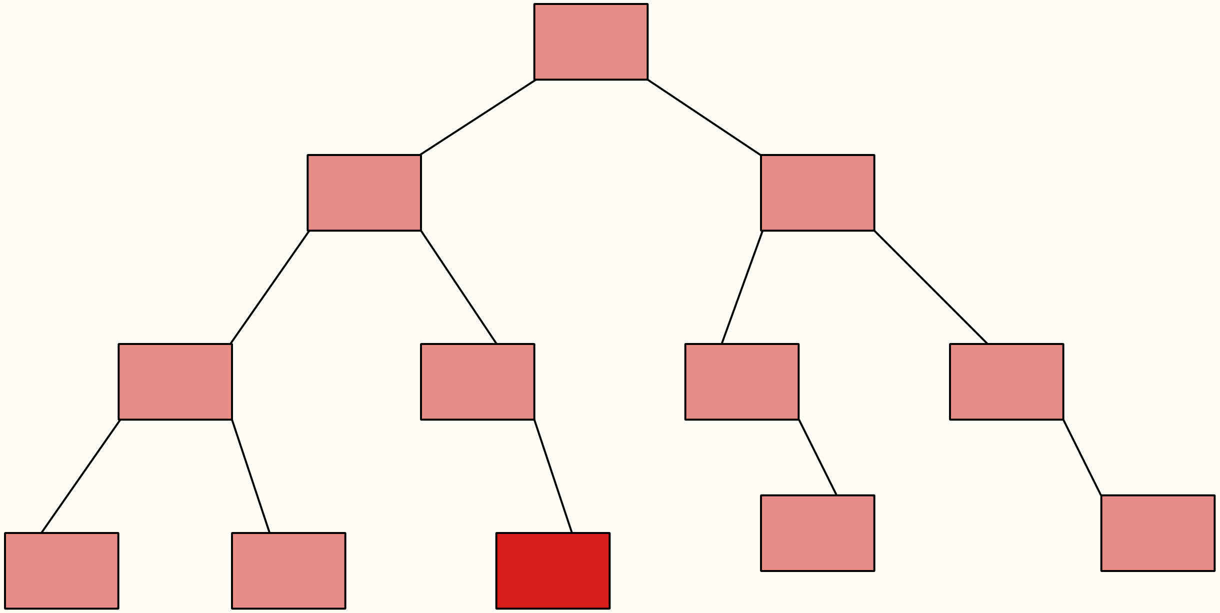
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



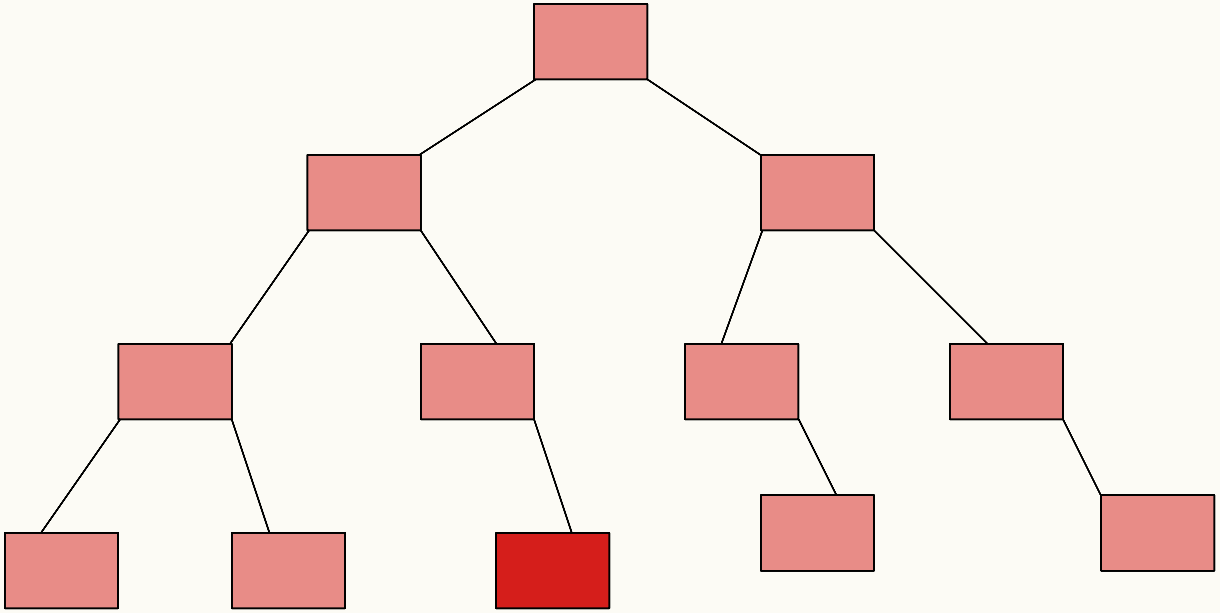
Each step we take as we search the tree reduces the remaining search space by half.

Sample Search



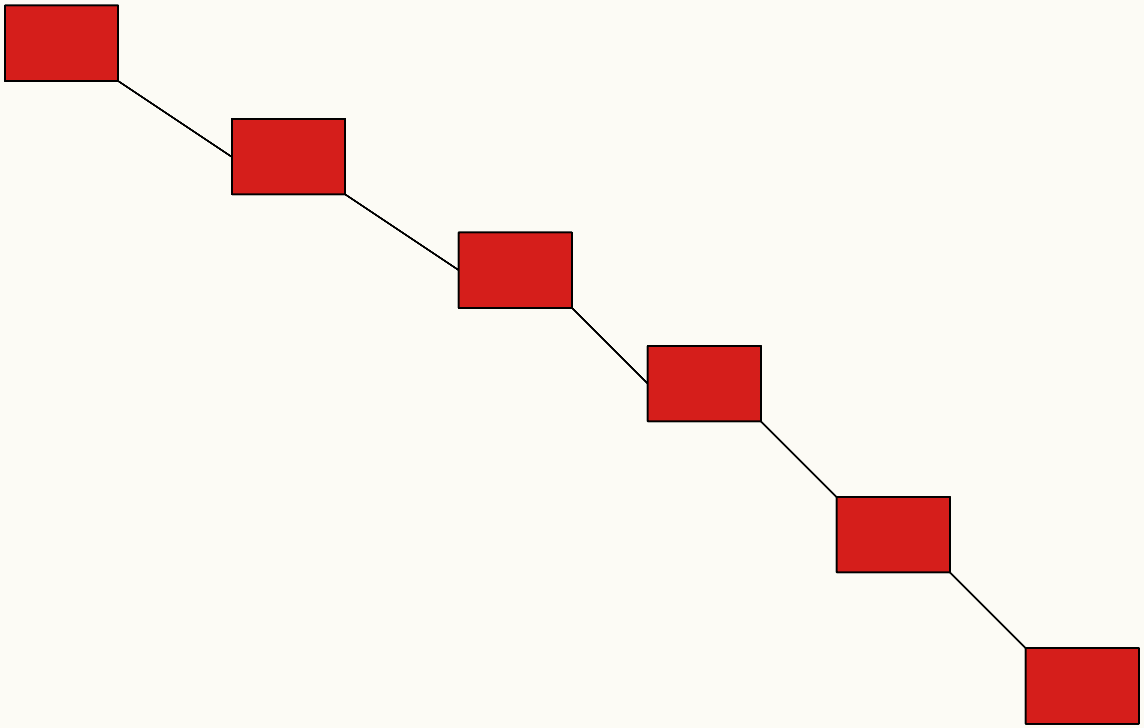
Each step we take as we search the tree reduces the remaining search space by half.

Height



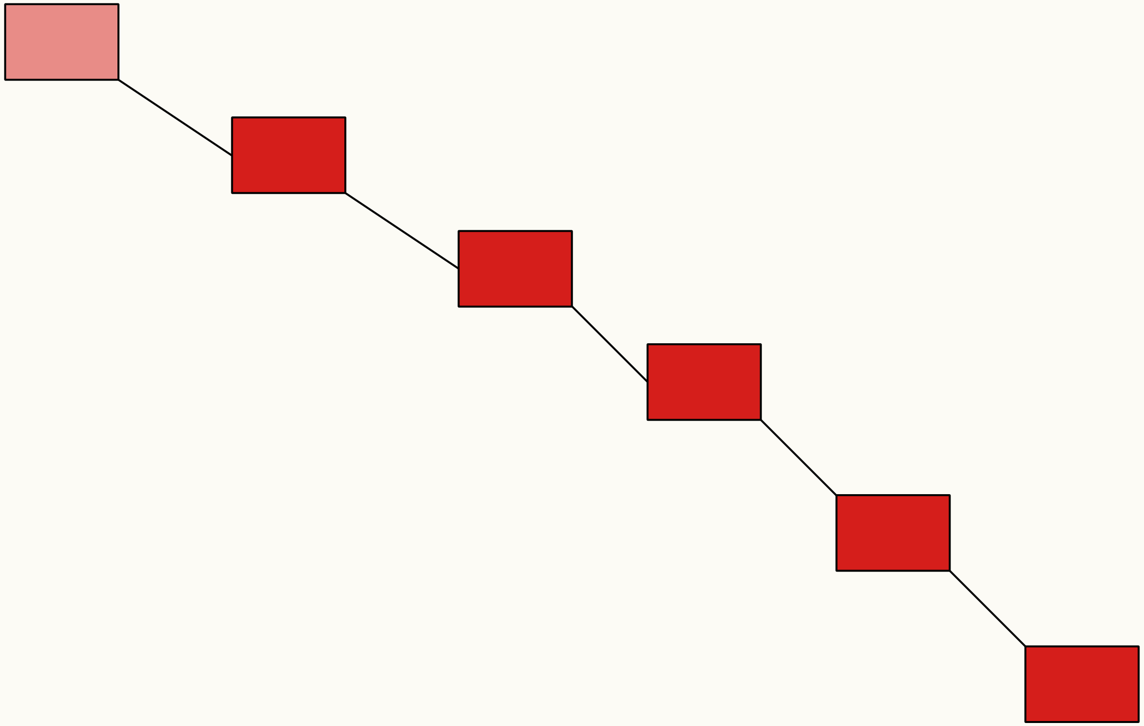
The tree has low height and all paths from the root node to other nodes are relatively short.

Unbalanced Trees



In contrast, this unbalanced tree is very high and has long paths from the root to other nodes. It essentially has degenerated to a linked list, which is very slow to search through.

Unbalanced Trees



Now, with each step we take, we have only reduced the search space by one node.

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).

CPSC Administrative Notes

- Written Assignment 2 extension
  - Due date is changed to Friday, March 20

- Lab 8 is posted
  - Starting Friday, on AVL trees
  - Marking lab 7 on QuickSort

The other section will be missing two lectures

- I think we should use that time

  A: To do more in class exercises while we're coving AVL's, Hashing, parallel processing, and B+ trees

  B: Hear about some cool research related stuff to data structures and algorithms, which wouldn't on the final.

  C: To relax at home (cancel a lecture)

  D: I don't care

So, Where were we?

- 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++

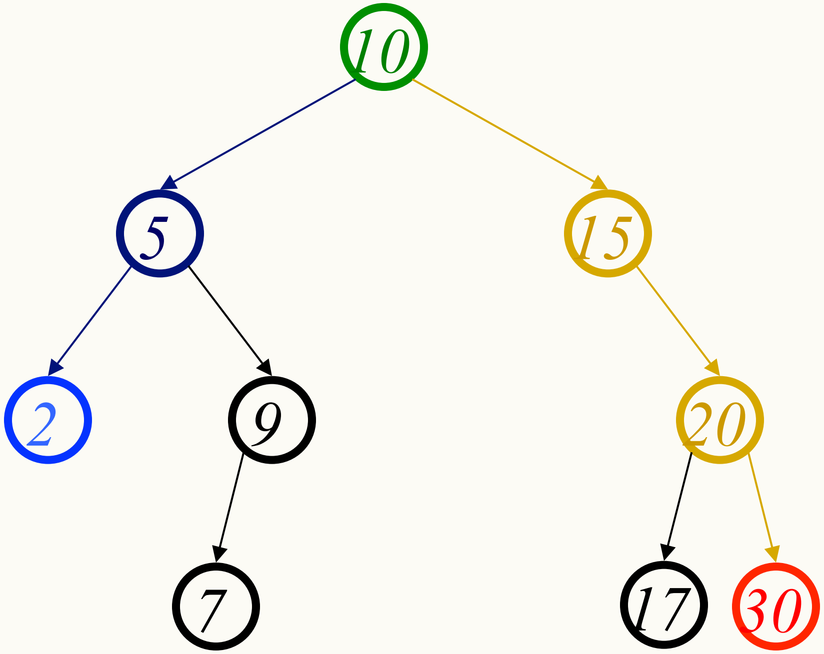
Bonus: FindMin/FindMax

- Find minimum

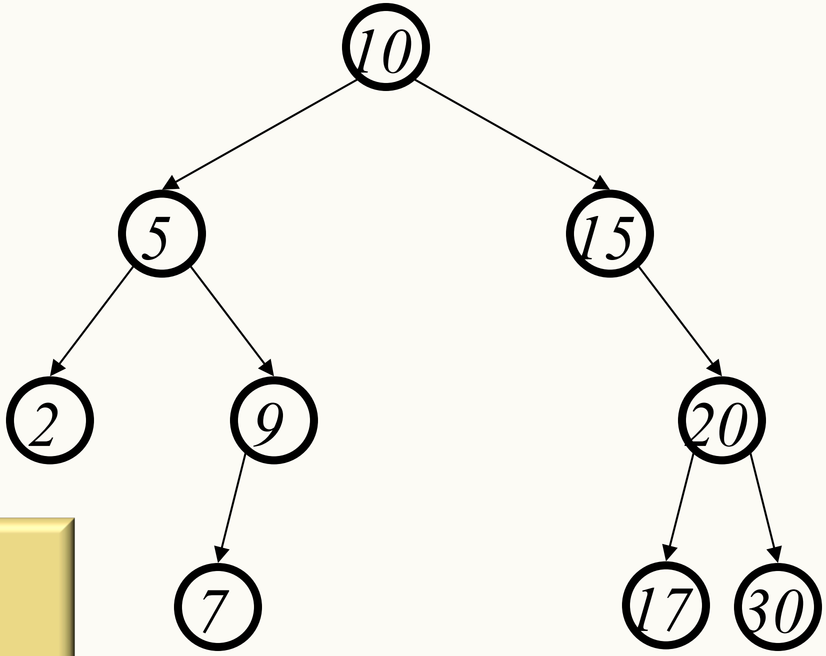
```
Node *& min(Node *& root) {
  if (root->left == NULL)
     return root;
  else
    return min(root->left);
}
```

- Find maximum

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



Double Bonus: Successor

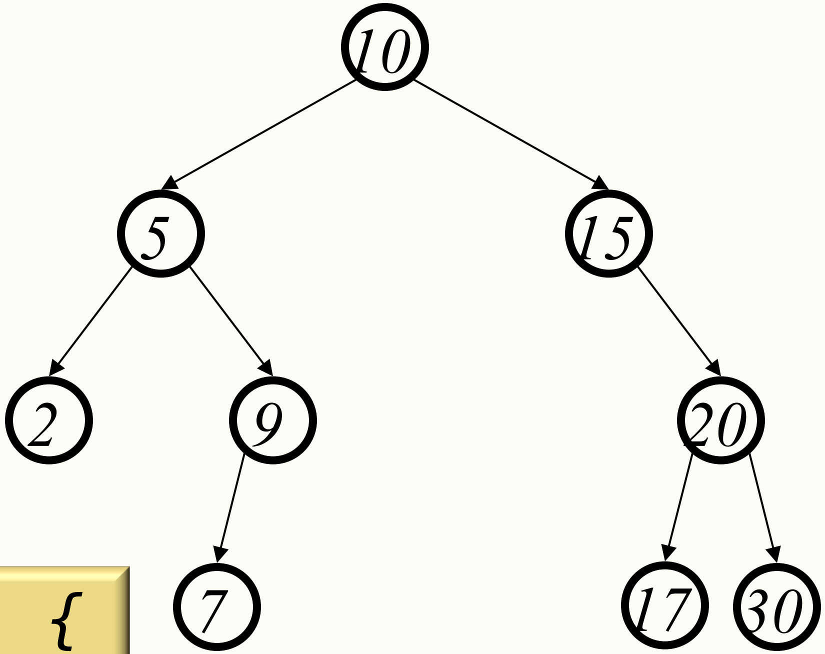


Find the next larger node in this node's subtree.

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

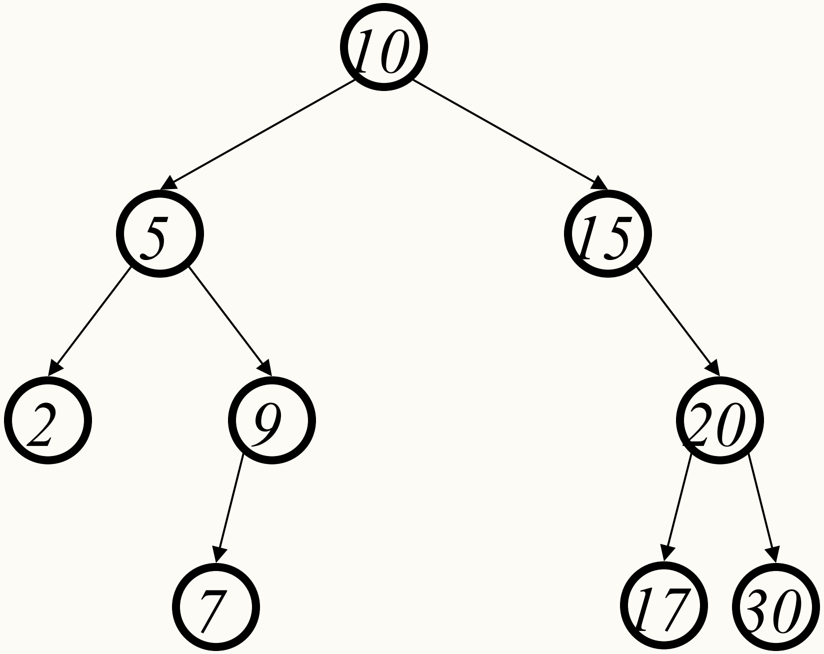
More Double Bonus: Predecessor

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);
}
```

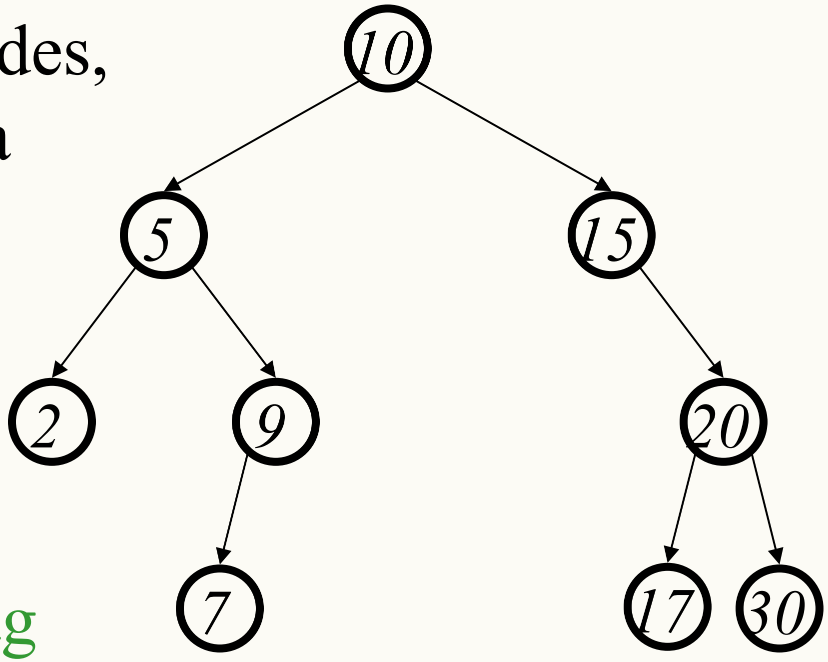
Deletion



Why might deletion be harder than insertion?

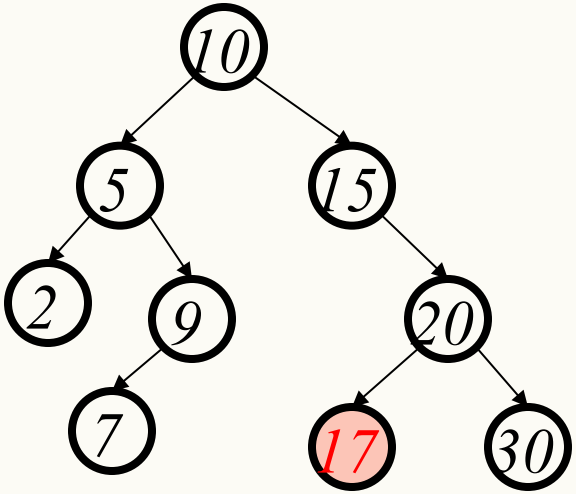
Lazy Deletion

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

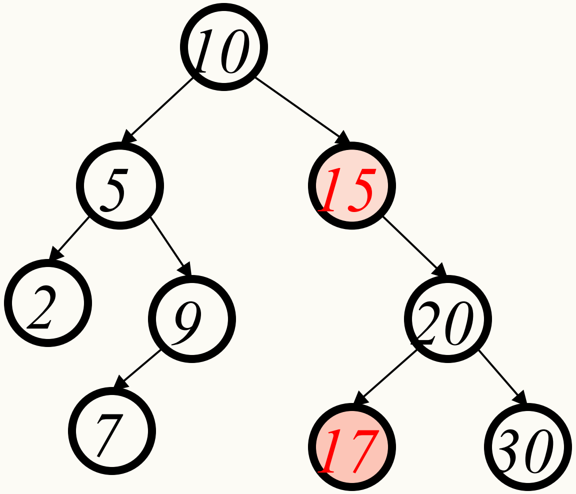


Lazy Deletion

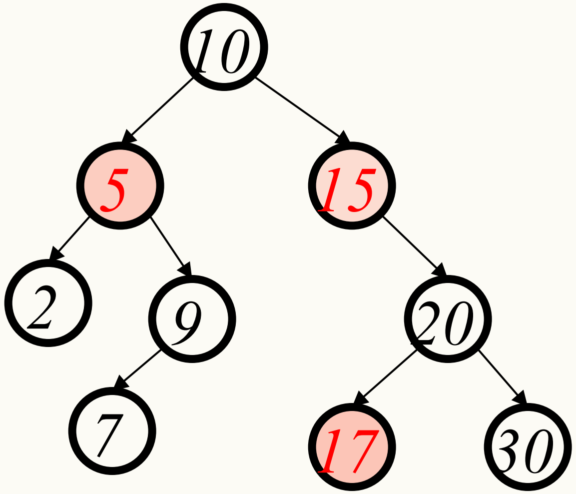
Delete(17)



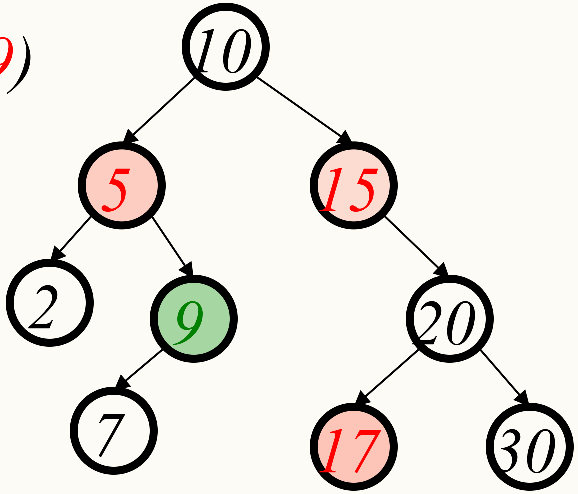
Delete(15)



Delete(5)

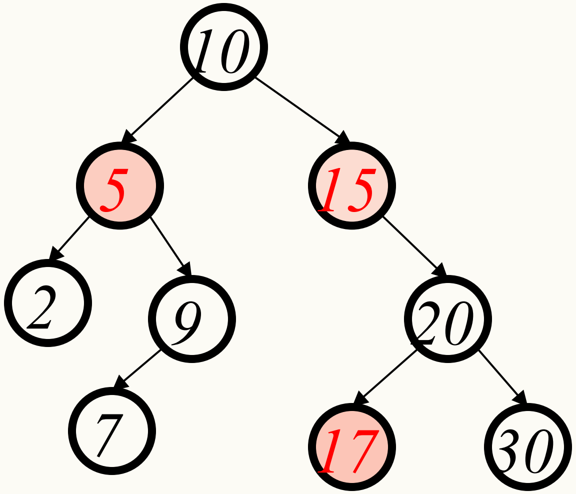


Find(9)

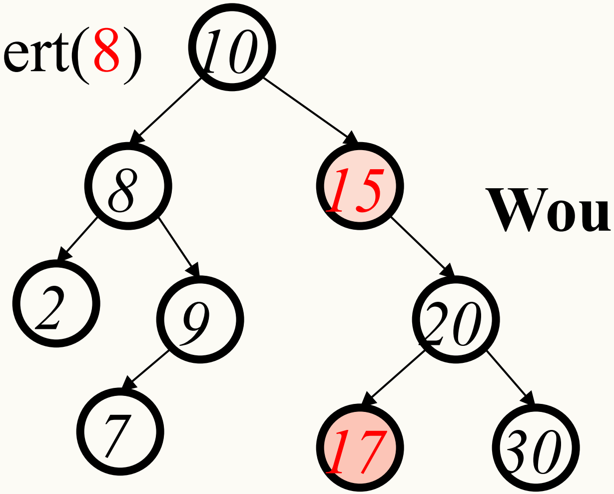


Lazy Deletion

Find(16)

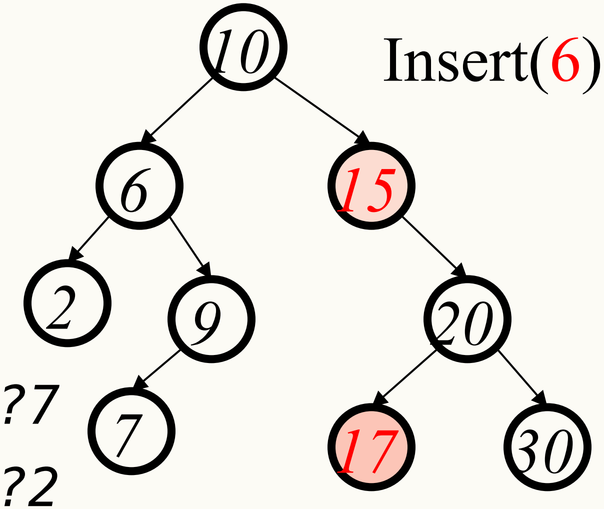


Insert(8)



Would this work?

Insert(6)

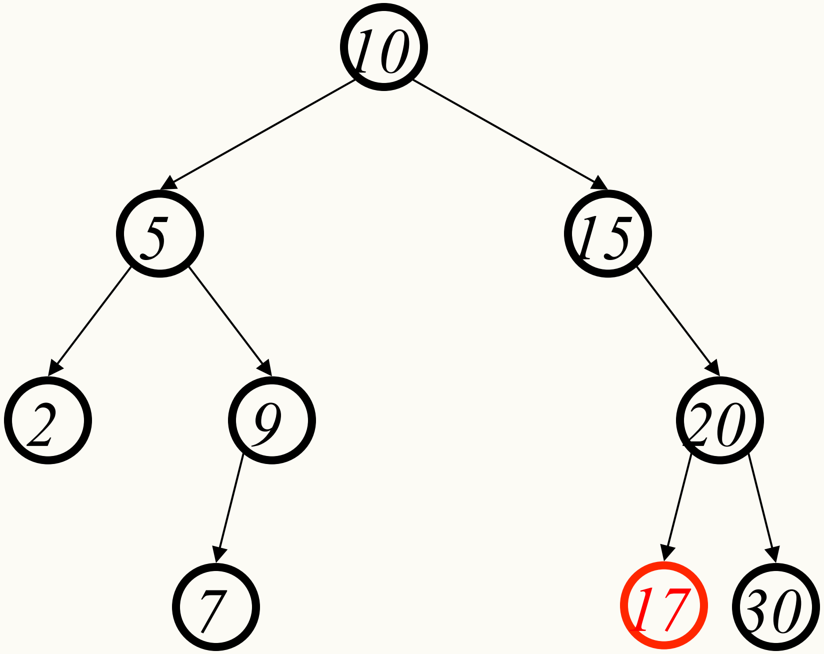


Succ(5)?7

pred(5)?2

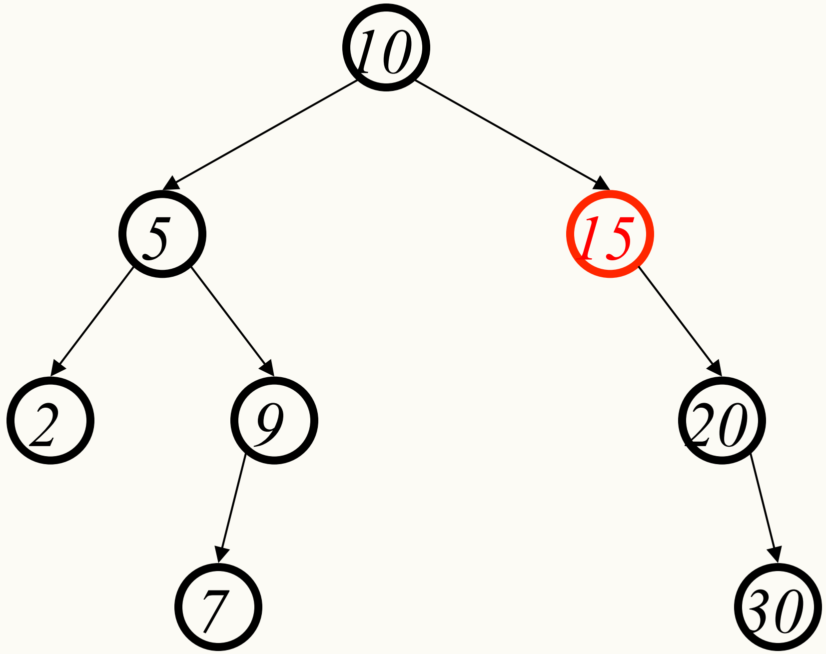
Deletion - Leaf Case

Delete(17)



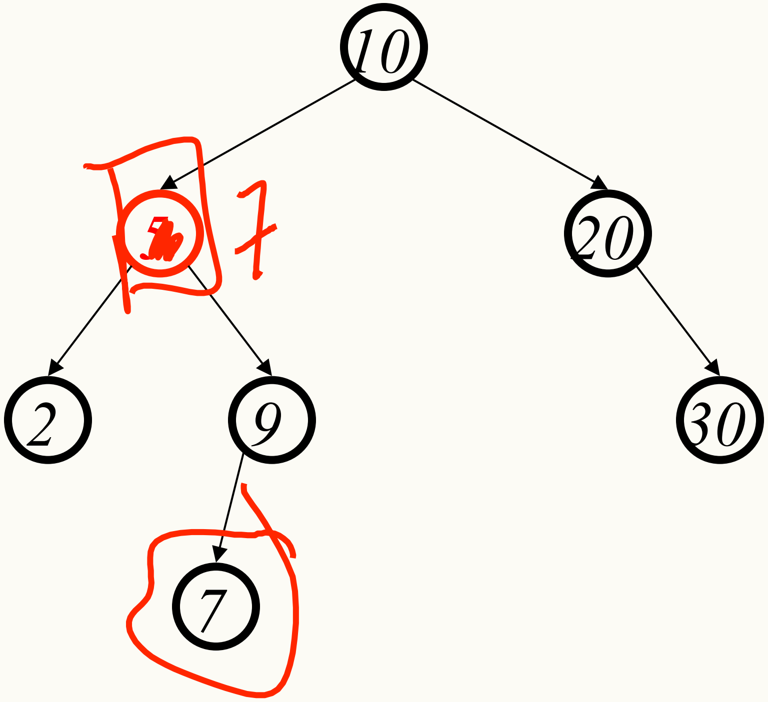
Deletion - One Child Case

Delete(15)



Deletion - Two Child Case

Delete(5)

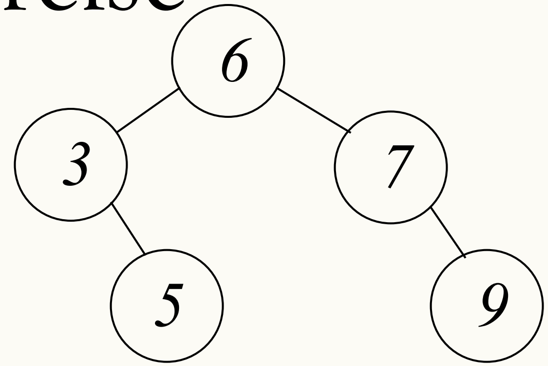


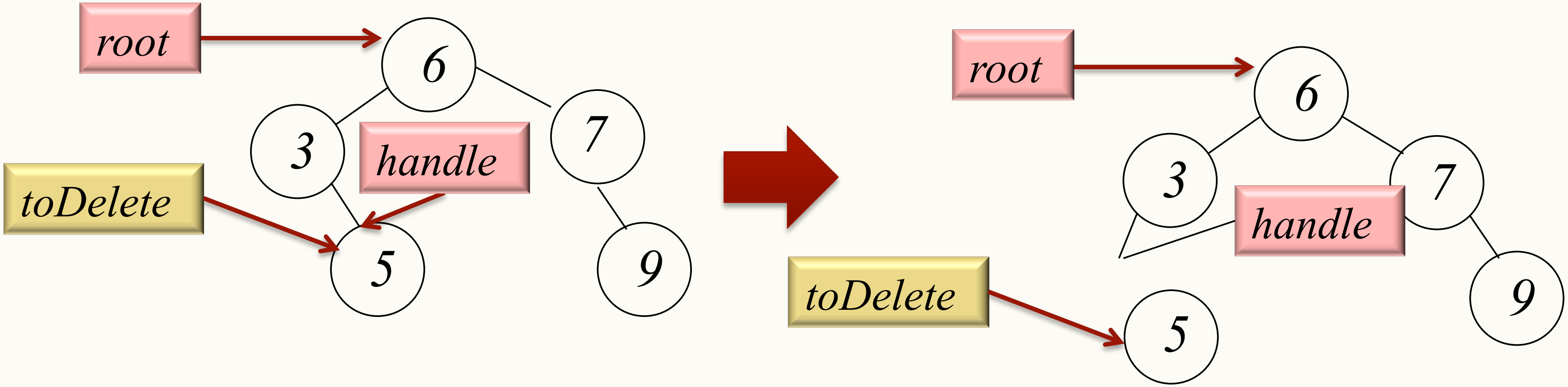
Delete Code

```
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;
  }
}
```

Deleting (leaf case) exercise

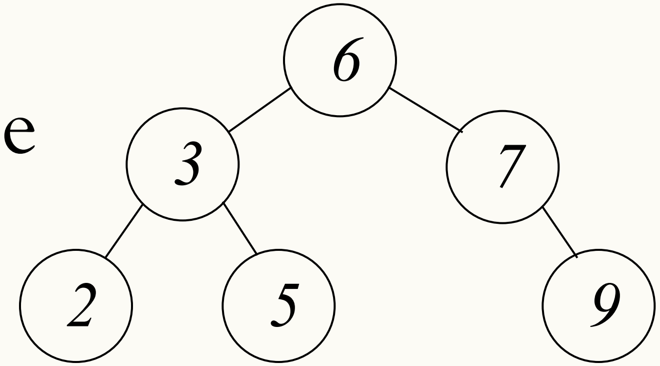
- Trace the code to see what the tree would look like after executing
  - `delete(5, Node *& 6)`

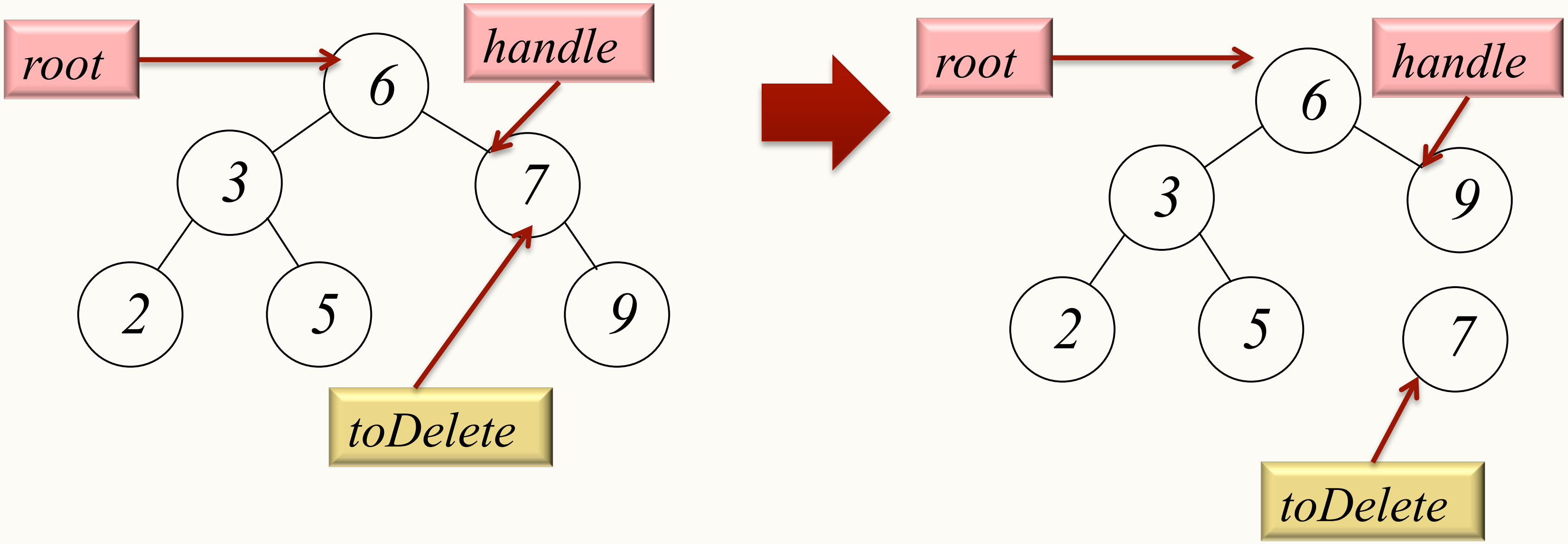




Deleting a BNode (one child) exercise

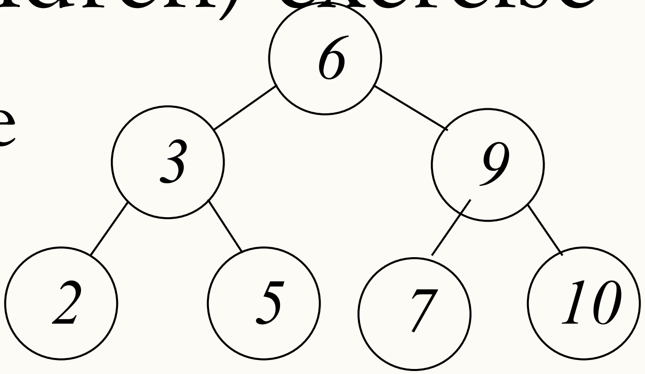
- Trace the code to see what the tree would look like after executing
  - `delete(7, Node *& 6)`

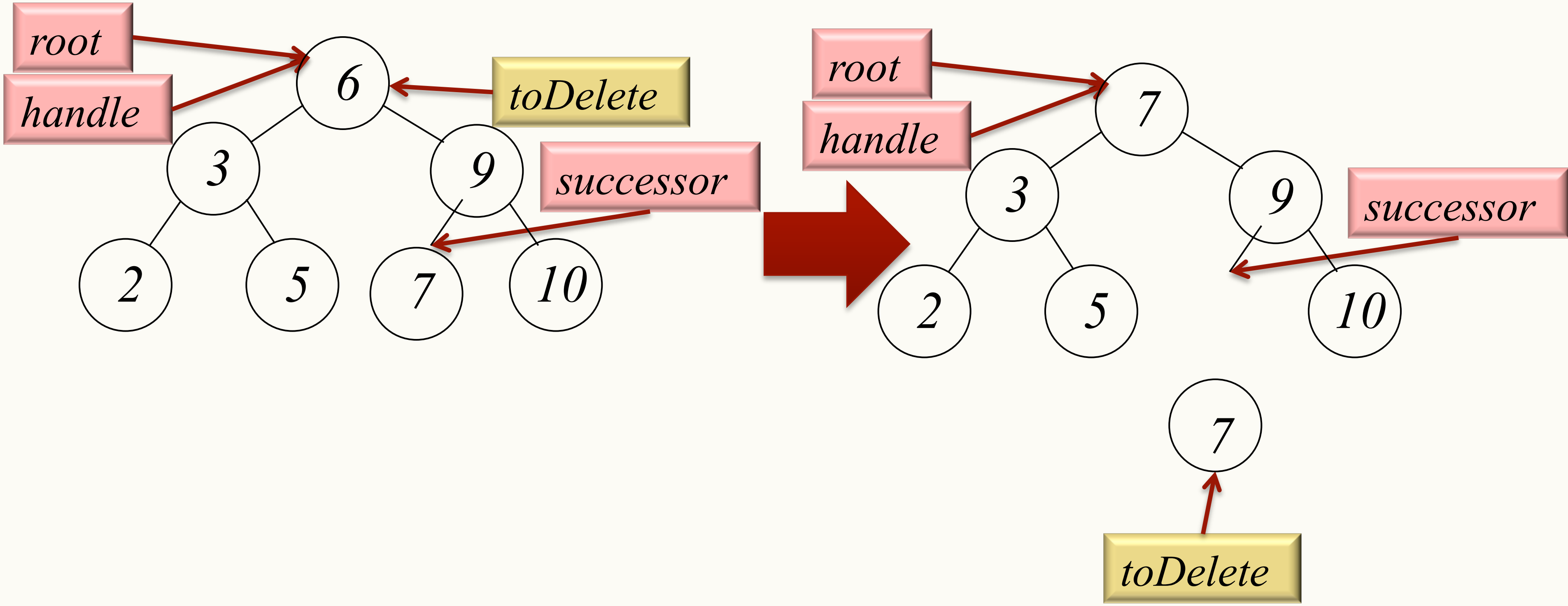




Deleting a BNode (both children) exercise

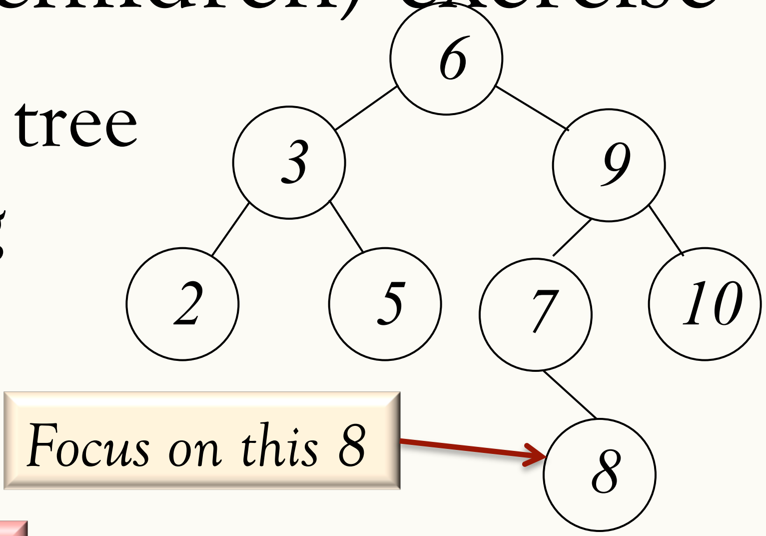
- Trace the code to see what the tree would look like after executing
  - `Delete(6, Node *& 6)`



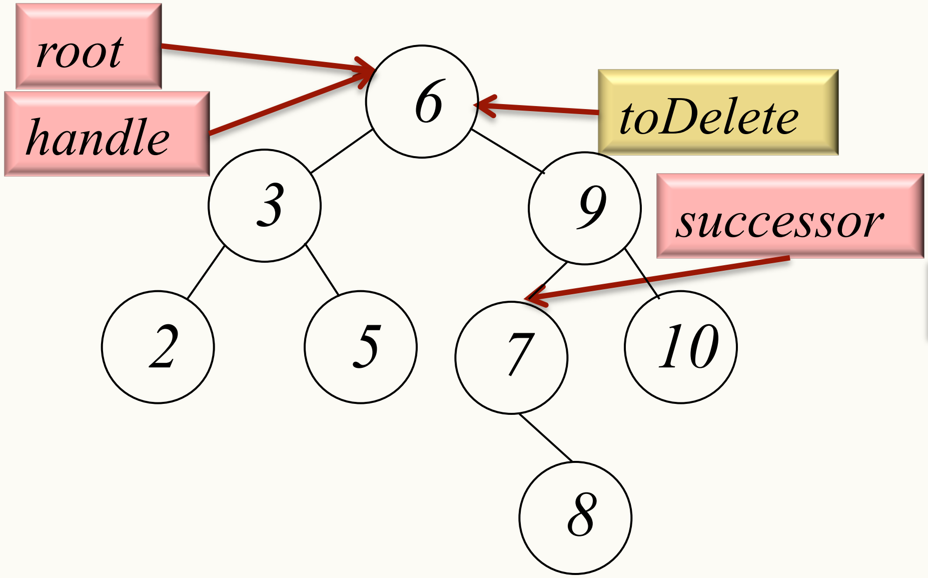


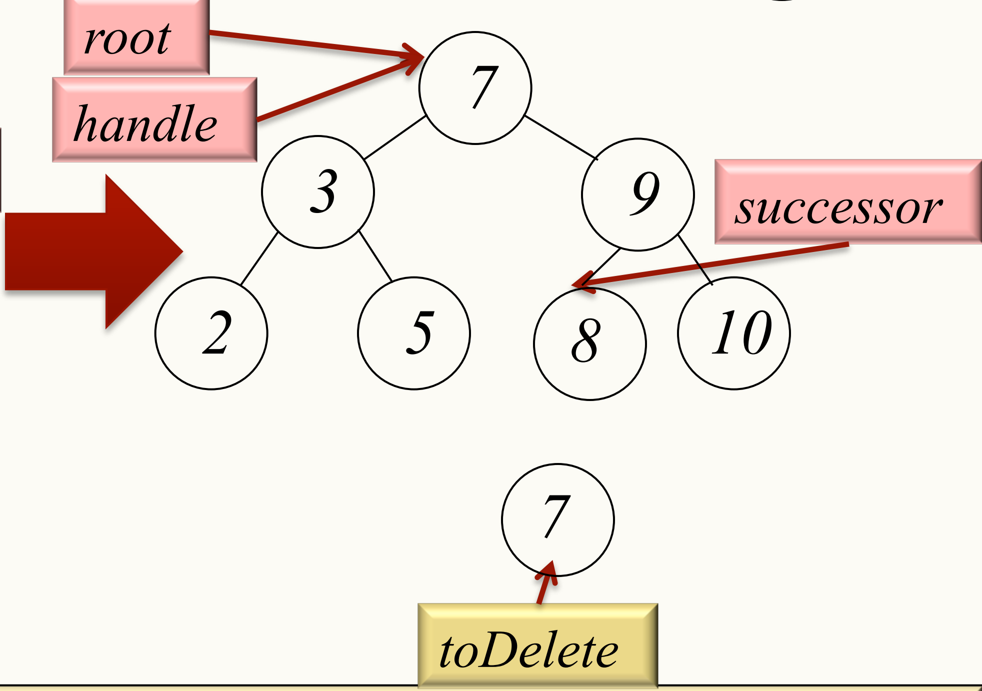
Deleting a BNode (both children) exercise

- Trace the code to see what the tree would look like after executing
  - `Delete(6, Node *& 6)`



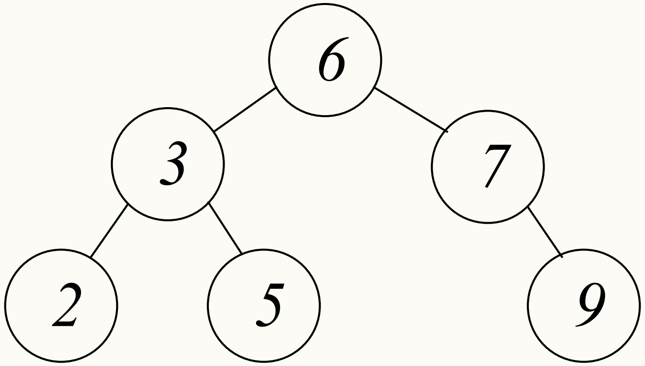
Focus on this 8





An Application of in-order traversing

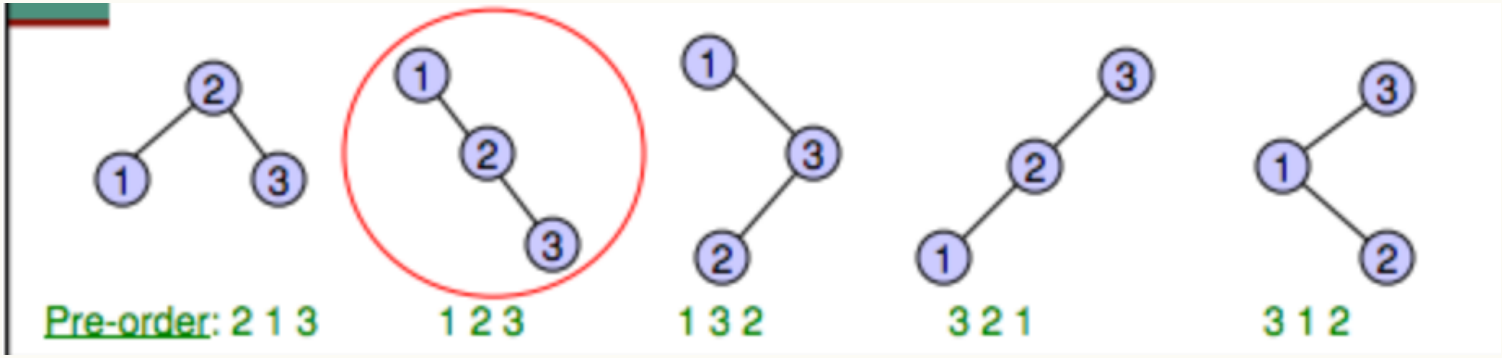
Sorting values in a binary search tree



In-order = 2, 3, 5, 6, 7, 9

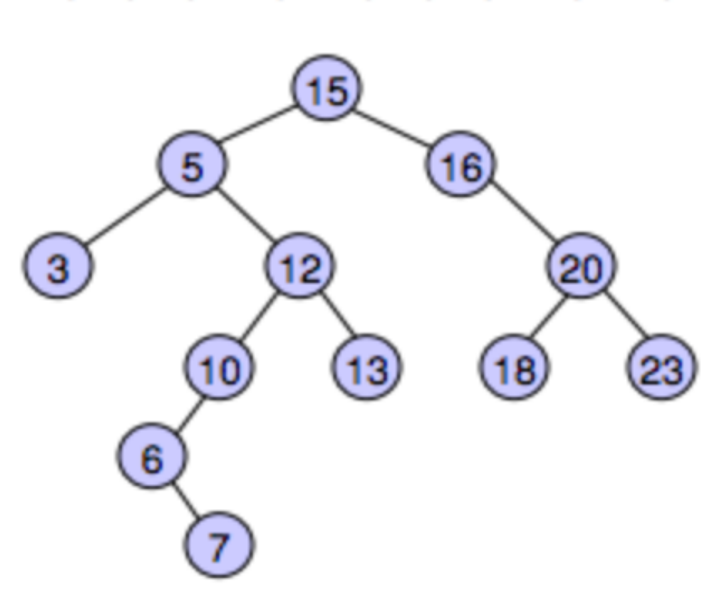
An Application of pre-order traversing

- Suppose we want to transmit our tree across the country to another programmer. Sending the in-order list would tell them the values, but would not communicate how the tree is built.
- All of the tree below have the in-order walk: 1 2 3. But only one of the trees below has the pre-order walk 1 2 3.
  - Note that we expect the tree to hold the BST property



In-class exercise

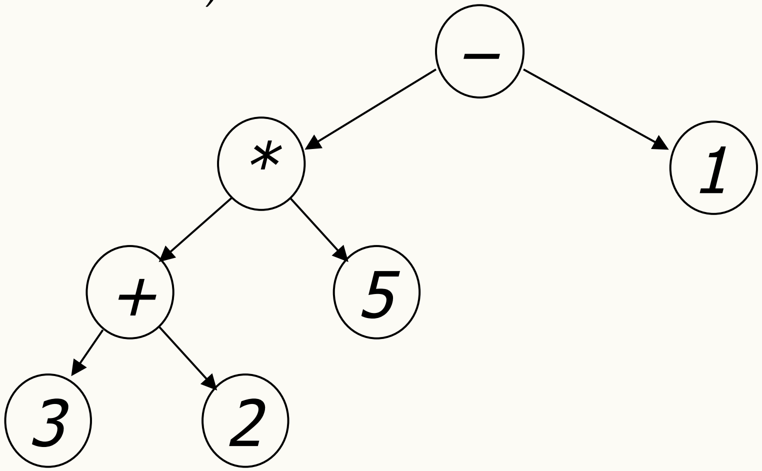
- Ex Can you recover the binary search tree from its pre-order traversal?
  - 15, 5, 3, 12, 10, 6, 7, 13, 16, 20, 18, 23



An Application of post-order traversing

Traverse the tree in post-order (left, right, current)

3 2 + 5 * 1 –



Use a stack to compute the value

Character scanned	Stack
3	3
2	3, 2
+	5
5	5, 5
*	25
1	25,1
-	24

Thinking about BSTs

- 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

Solutions?

- Reduce disk accesses?
- Keep BSTs shallow?

Learning Goals revisited

- 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