CS 106B

Lecture 21: Binary Heaps

Friday, November 11, 2016

Programming Abstractions
Fall 2016
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Sections 16.1-16.3

Today's Topics

- Logistics
  - Regrade requests due Friday
  - We know you are working hard on the assignments. We have decided to give everyone an extra late day.

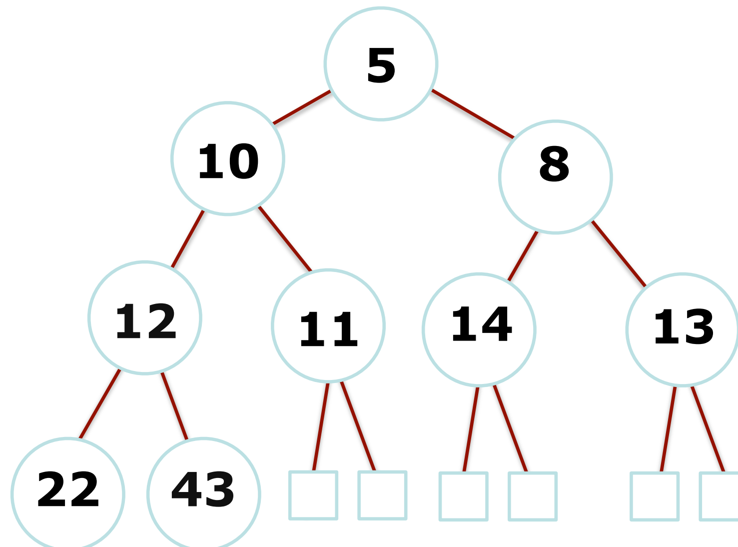
- Recap: Heap enqueue and dequeue
- Binary Search Trees

Heap Operations

Remember that there are three important priority queue operations:

1. **`peek()`**: return an element of h with the smallest key.
2. **`enqueue(k,e)`**: add an element e with key k into the heap.
3. **`dequeue()`**: removes the smallest element from h.

We can use a heap for this. The "heap invariant" is: all children must have a lower priority than their parent.



http://www.cs.usfca.edu/~galles/visualization/Heap.html

YEAH Hours animation: https://www.youtube.com/watch?v=Z86ZDMdkUyo#t=39m36s

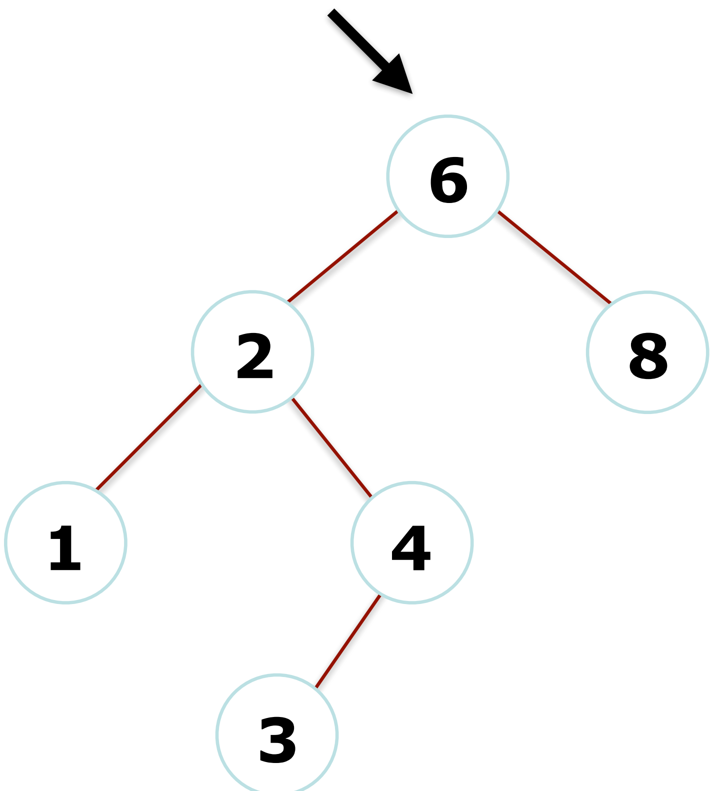
Binary Search Trees

- Binary trees are frequently used in searching.
- Binary Search Trees (BSTs) have an *invariant* that says the following:

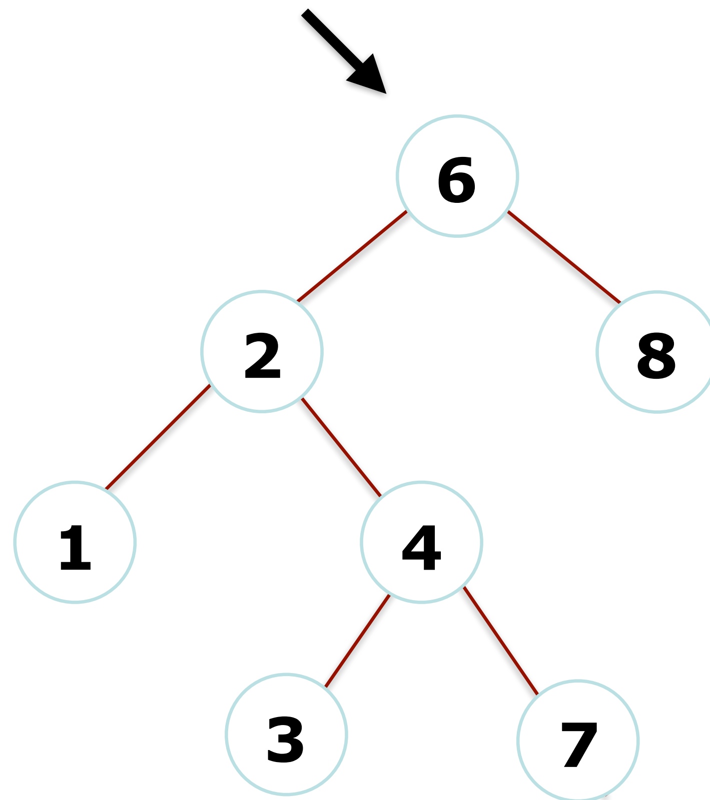
For every node, X, all the items in its left subtree are smaller than X, and the items in the right tree are larger than X.

Binary Search Trees

Binary Search Tree

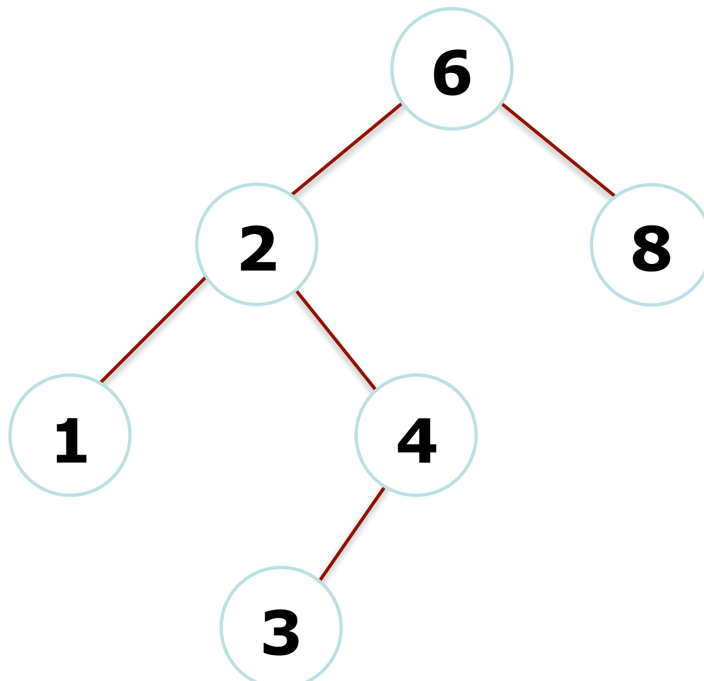


Not a Binary Search Tree



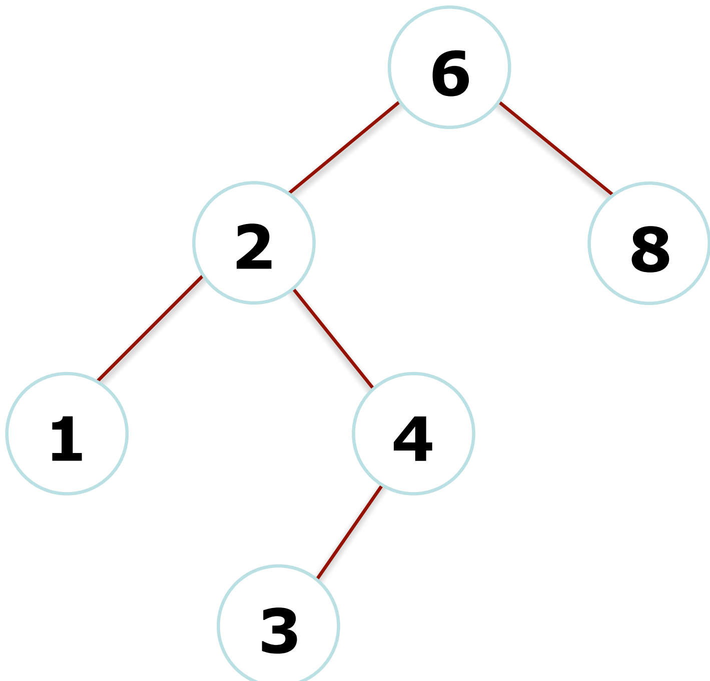
Binary Search Trees

Binary Search Trees have an average depth on the order of $\log_2(n)$: very nice!



Binary Search Trees

In order to use binary search trees (BSTs), we must define and write a few methods for them (and they are all recursive!)



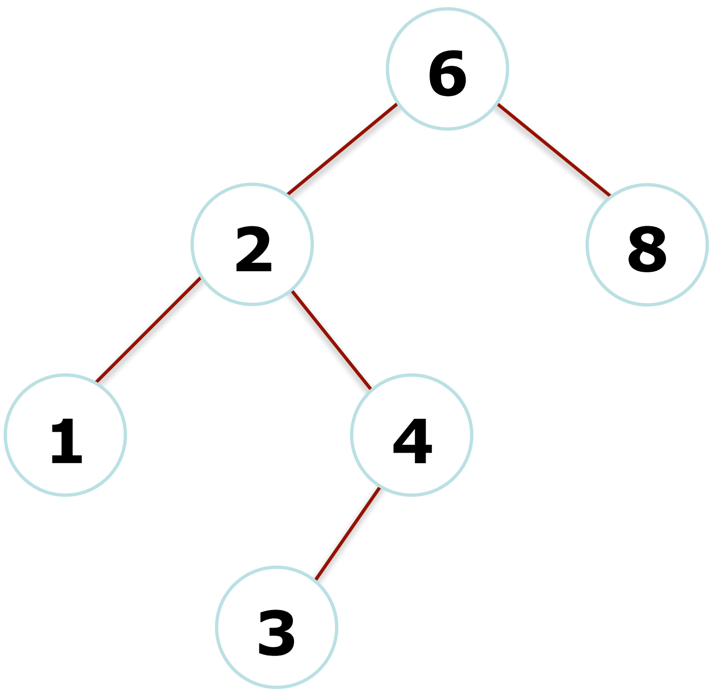
Easy methods:

1. findMin()
2. findMax()
3. contains()
4. add()

Hard method:

5. remove()

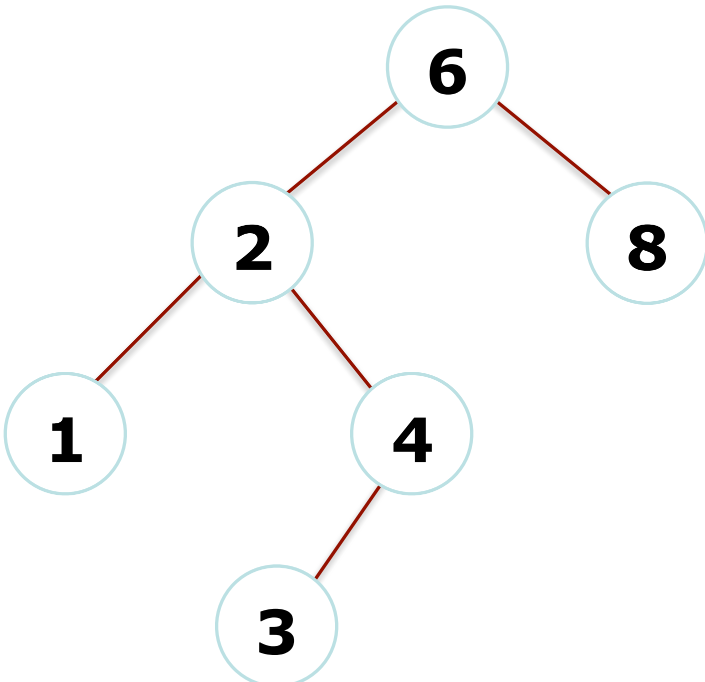
Binary Search Trees: findMin()



findMin():
Start at root, and go left until a node doesn't have a left child.

findMax():
Start at root, and go right until a node doesn't have a right child.

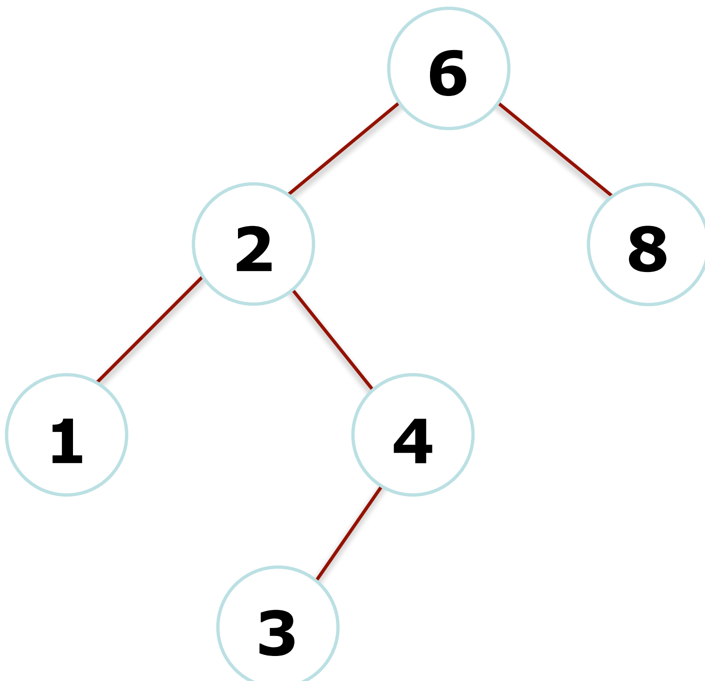
Binary Search Trees: contains()



Does tree T contain X?

1. If T is empty, return false
2. If T is X, return true
3. Recursively call either T→left or T→right, depending on X's relationship to T (smaller or larger).

Binary Search Trees: add(value)

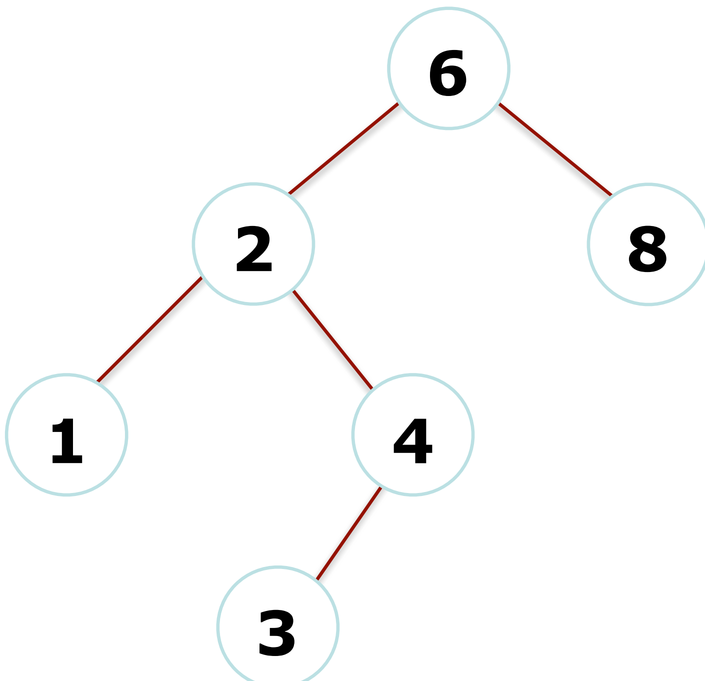


How do we add 5?

Similar to contains()

1. If T is empty, add at root
2. Recursively call either T→left or T→right, depending on X's relationship to T (smaller or larger).
3. If node traversed to is NULL, add

Binary Search Trees: remove(value)

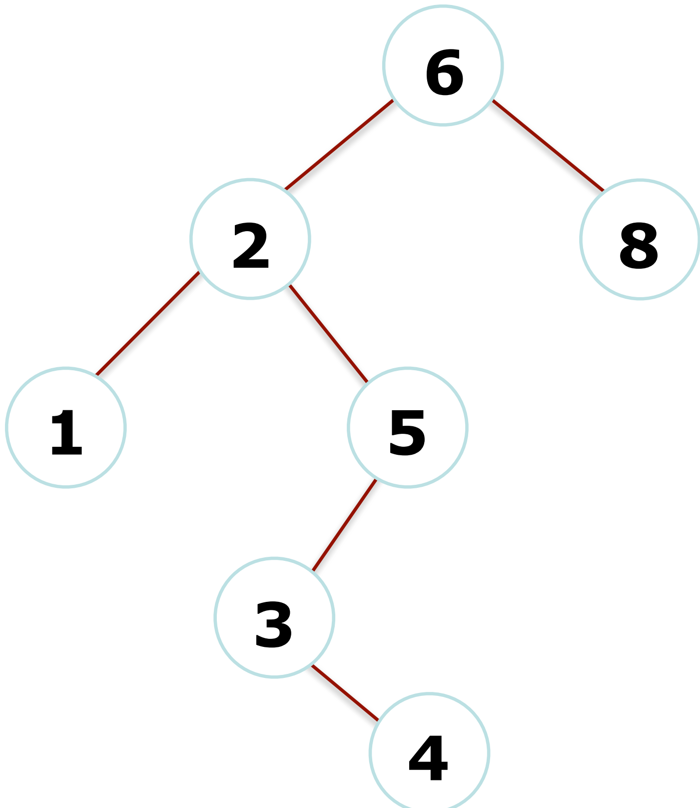


How do we delete 4?

Harder. Several possibilities.

1. Search for node (like contains)
2. If the node is a leaf, just delete (phew)
3. If the node has one child, “bypass” (think linked-list removal)
4. …

Binary Search Trees: remove(value)



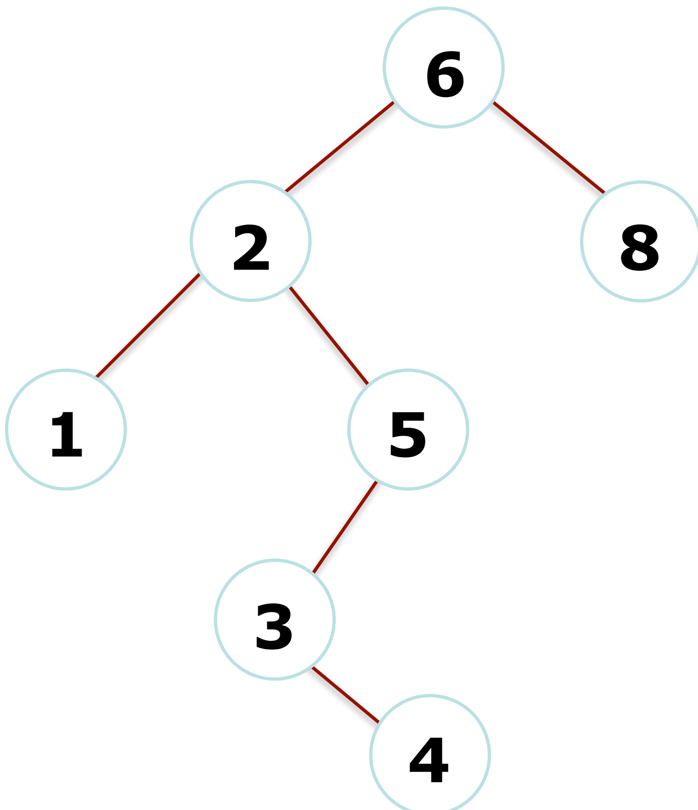
How do we remove 2?

4. If a node has two children:

   Replace with smallest data in the right subtree, and recursively delete that node (which is now empty).

   Note: if the root holds the value to remove, it is a special case…

BSTs and Sets



Guess what? BSTs make a terrific container for a *set*

Let's talk about Big O (average case)

findMin()? O(log n)

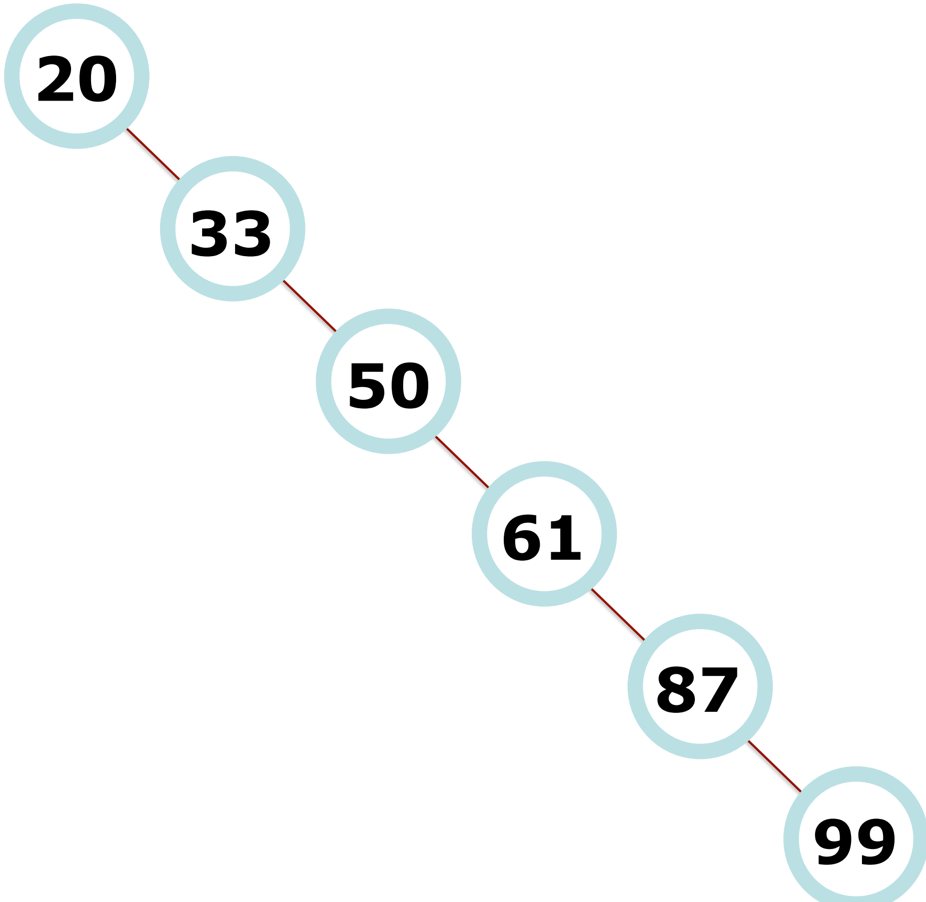
findMax()? O(log n)

insert()? O(log n)

remove()? O(log n)

Great! That said...what about worst case?

Balancing Trees



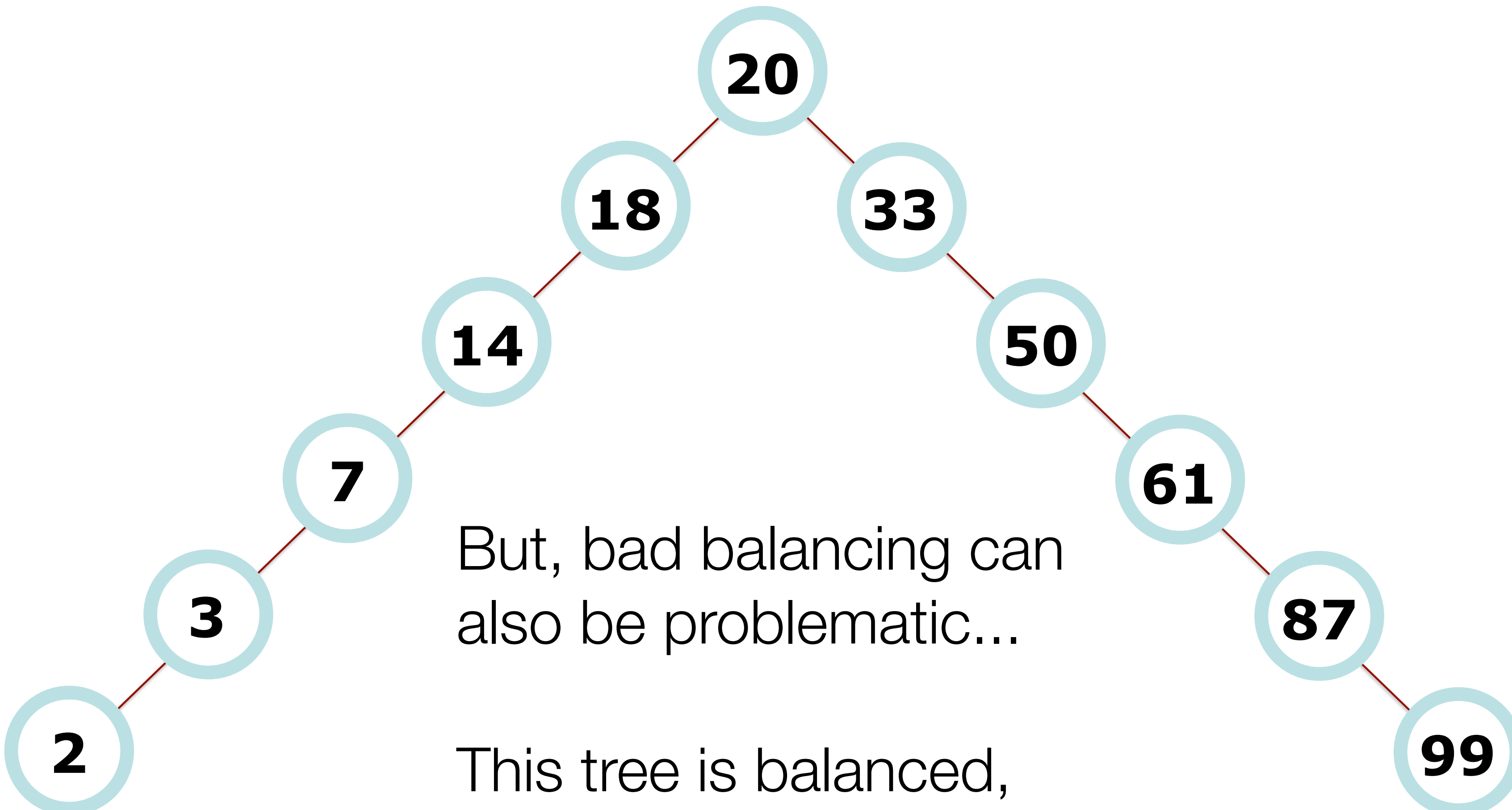
Insert the following into a BST:
20, 33, 50, 61, 87, 99

What kind of tree to we get?

We get a Linked List Tree, and O(n) behavior :(

What we want is a "balanced" tree (that is one nice thing about heaps -- they're always balanced!)

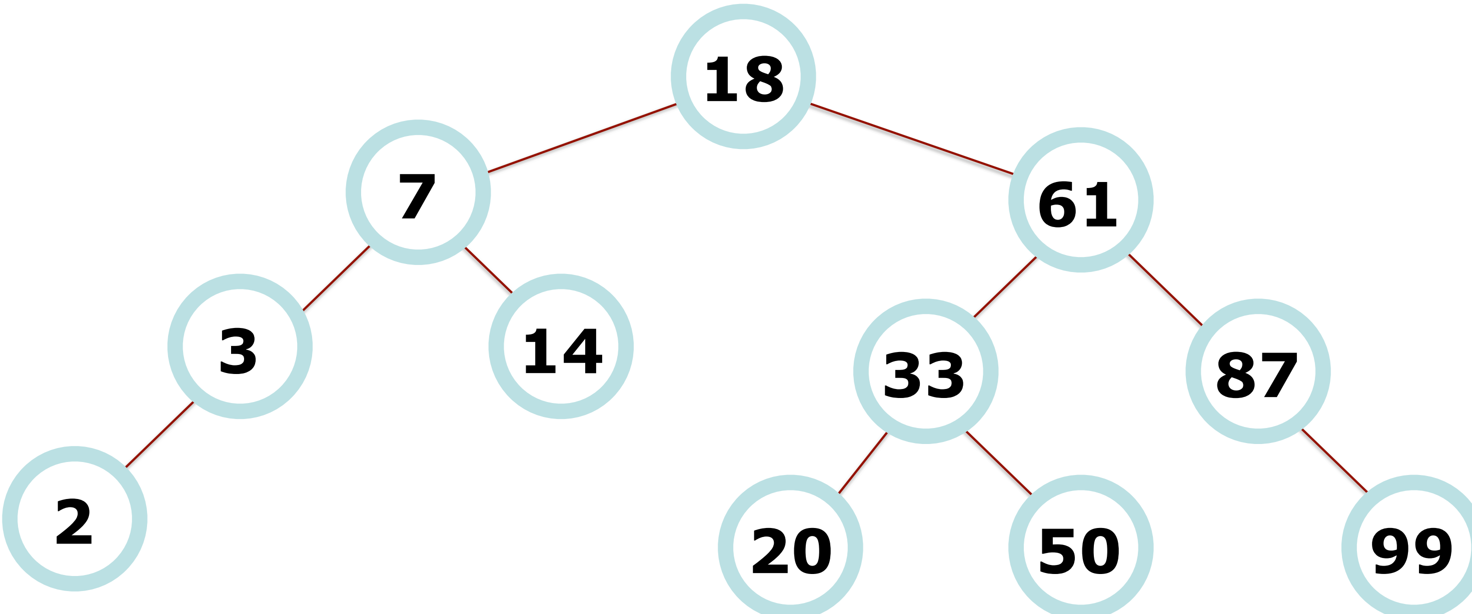
Balancing Trees



But, bad balancing can also be problematic...

This tree is balanced, but only at the root.

Balancing Trees: What we want



There are algorithms (AVL, Red-Black, etc.) that will balance during insertion. Knowing the algorithms is beyond the scope of this class, but you can play around: https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

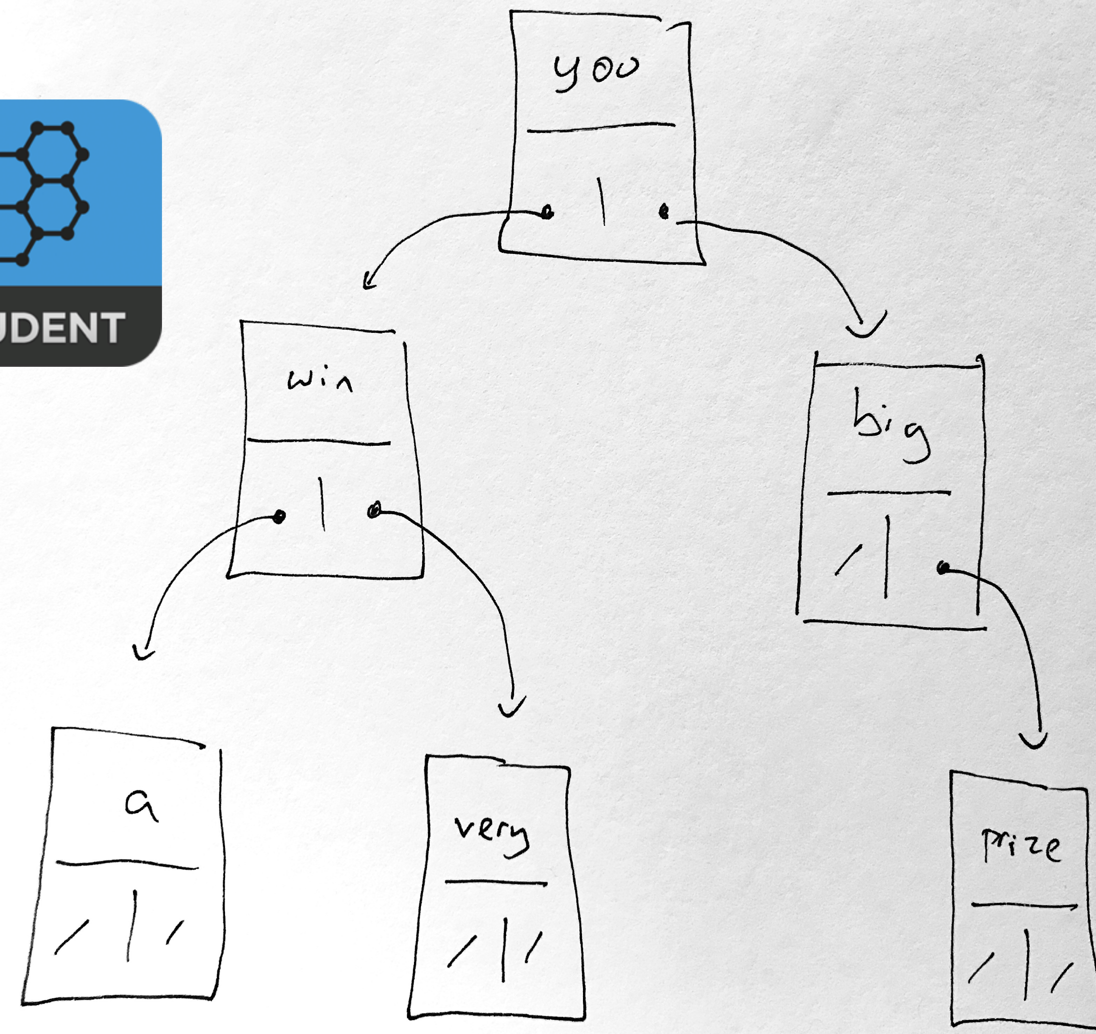
From Monday: Game Show Tree

A
```
void doorOne(Tree * tree) {
  if(tree == NULL) return;
  cout<<tree->value<<” “;
  doorOne(tree->left);
  doorOne(tree->right);
}
```

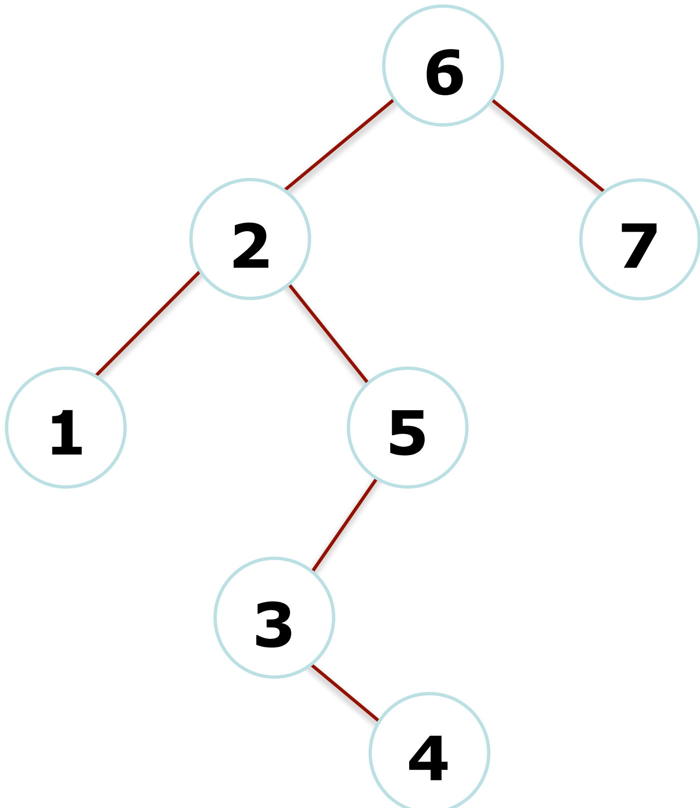
B
```
void doorTwo(Tree * tree) {
  if(tree == NULL) return;
  doorTwo(tree->left);
  cout<<tree->value<<” “;
  doorTwo(tree->right);
}
```

C
```
Void doorThree(Tree * tree) {
  if(tree == NULL) return;
  doorThree(tree->left);
  doorThree(tree->right);
  cout<<tree->value<<” “;
}
```

STUDENT



Traversing a BST

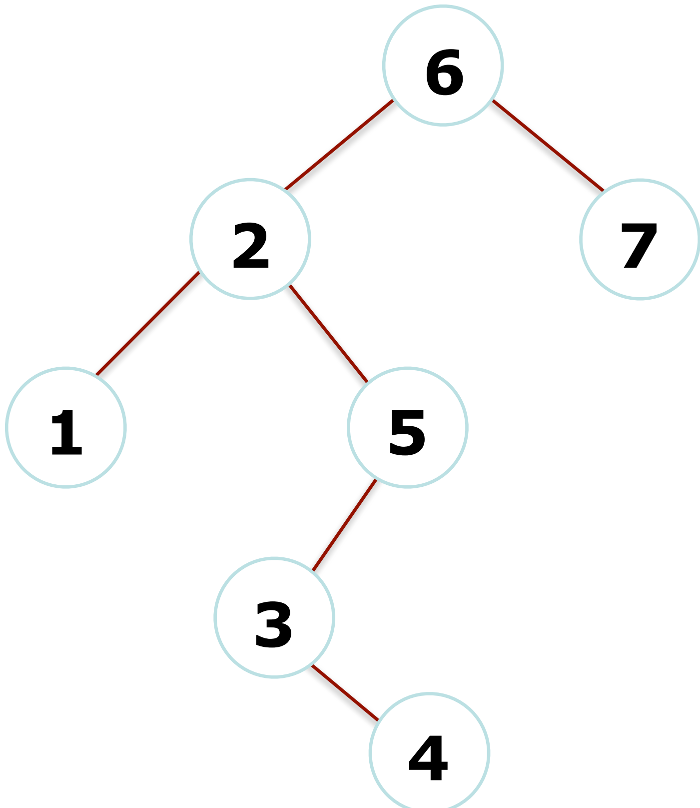


There are four different ways to traverse a BST:

1. In-order traversal (recursive)
2. Pre-order traversal (recursive)
3. Post-order traversal (recursive)
4. Level-order traversal ("breadth first search") (*not* recursive) -- we will have an entire class on BFS!

There are different reasons for traversing in different ways (we'll see some!)

Traversing a BST

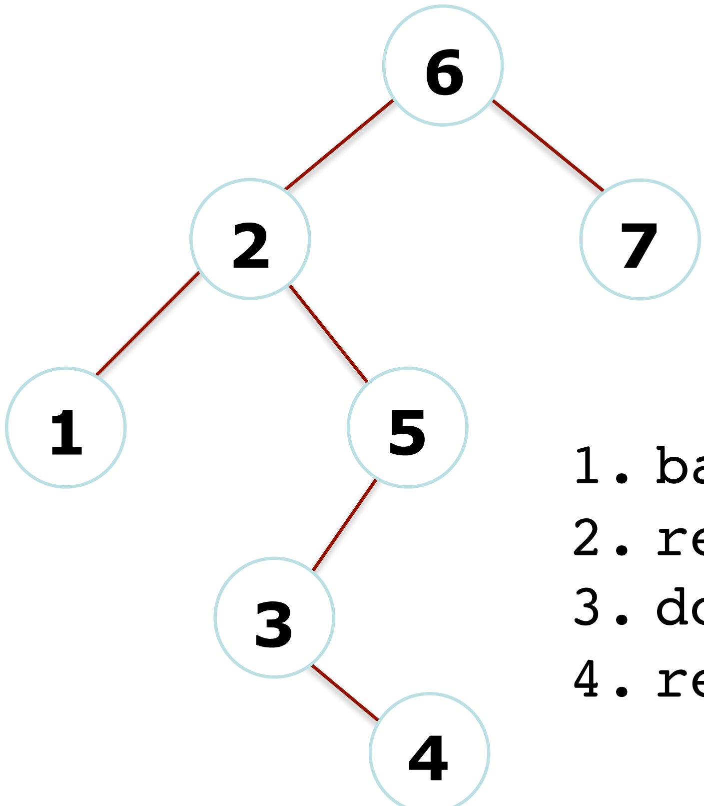


There are four different ways to traverse a BST:

1. In-order traversal (recursive)
2. Pre-order traversal (recursive)
3. Post-order traversal (recursive)
4. Level-order traversal ("breadth first search") (*not* recursive) -- we will have an entire class on BFS!

There are different reasons for traversing in different ways (we'll see some!)

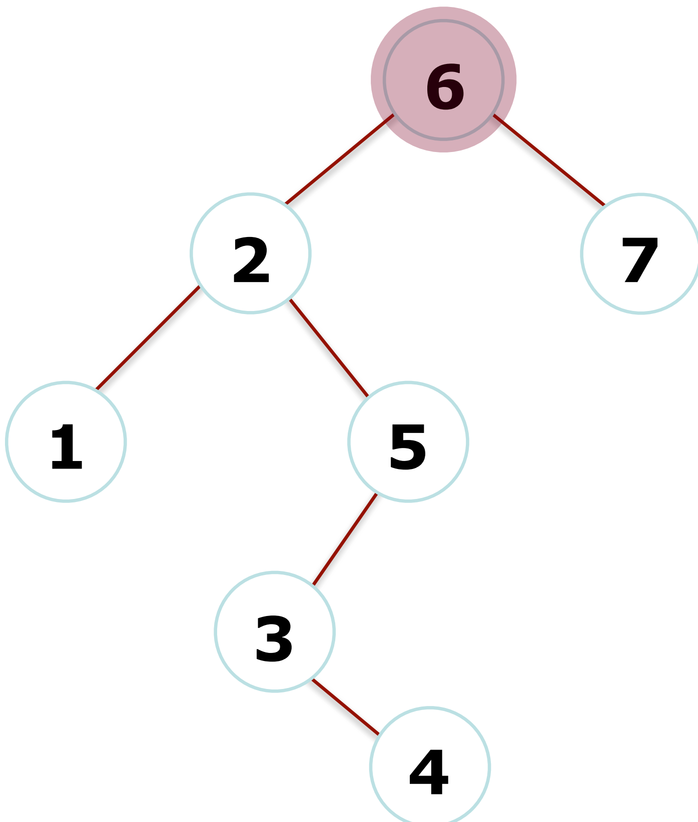
In-Order Traversal



Pseudocode:

```
1. base case: if current == NULL, return
2. recurse left
3. do something with current node
4. recurse right
```

In-Order Traversal Example: printing

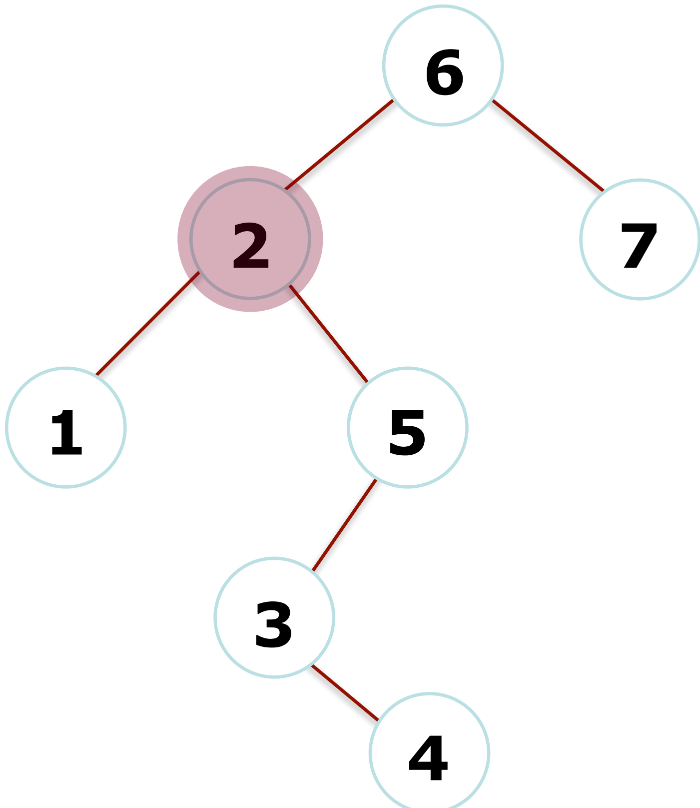


Current Node: 6

1. current not NULL
2. recurse left

Output:

In-Order Traversal Example: printing

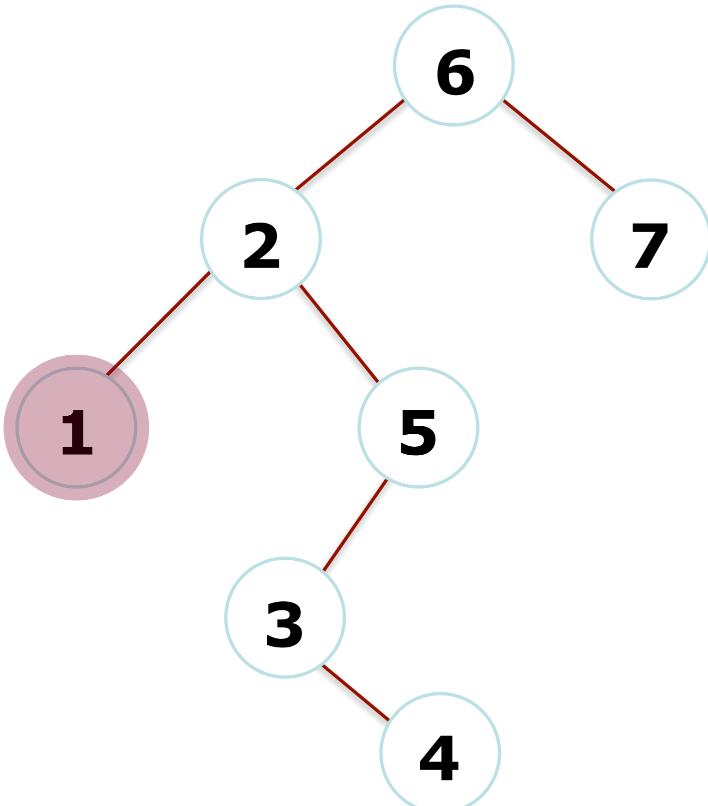


Current Node: 2

1. current not NULL
2. recurse left

Output:

In-Order Traversal Example: printing

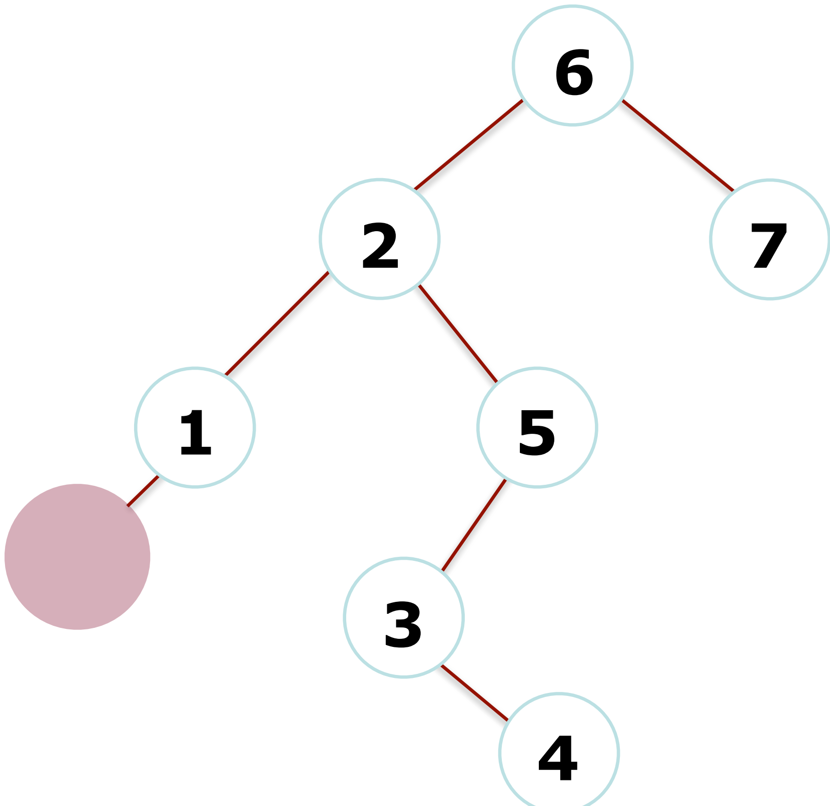


Current Node: 1

1. current not NULL
2. recurse left

Output:

In-Order Traversal Example: printing

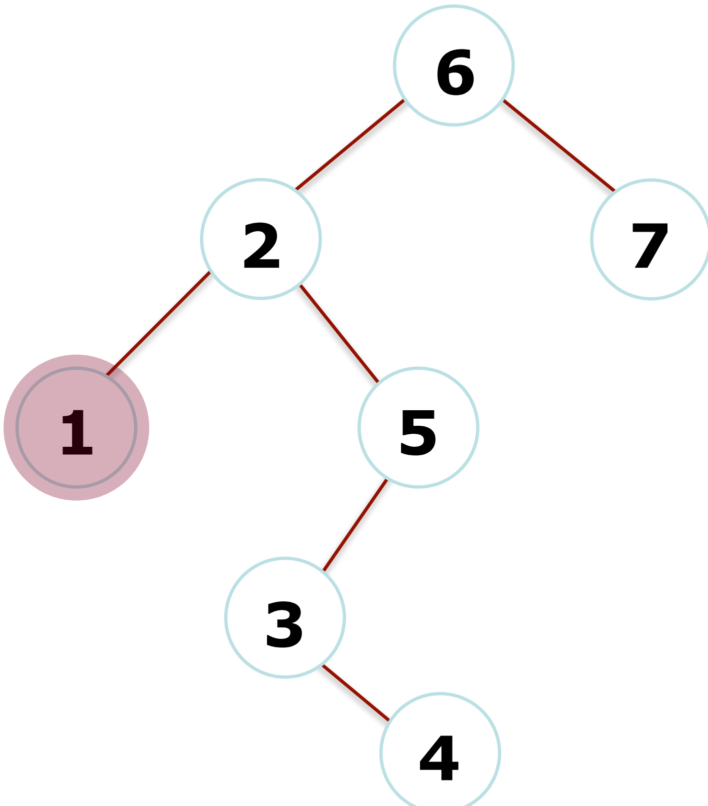


Current Node: NULL

1. current NULL: return

Output:

In-Order Traversal Example: printing

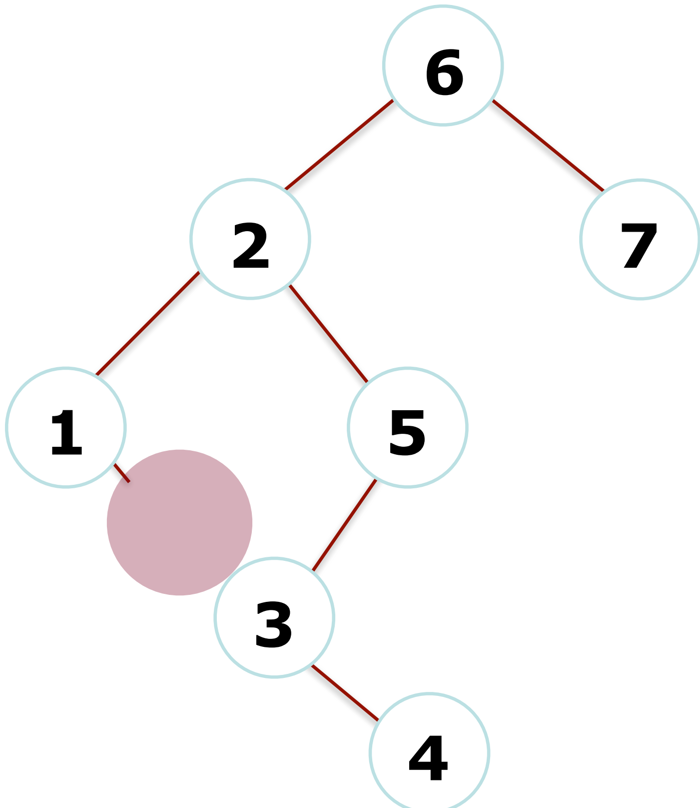


Current Node: 1

1. ~~current not NULL~~
2. ~~recurse left~~
3. print "1"
4. recurse right

Output: 1

In-Order Traversal Example: printing

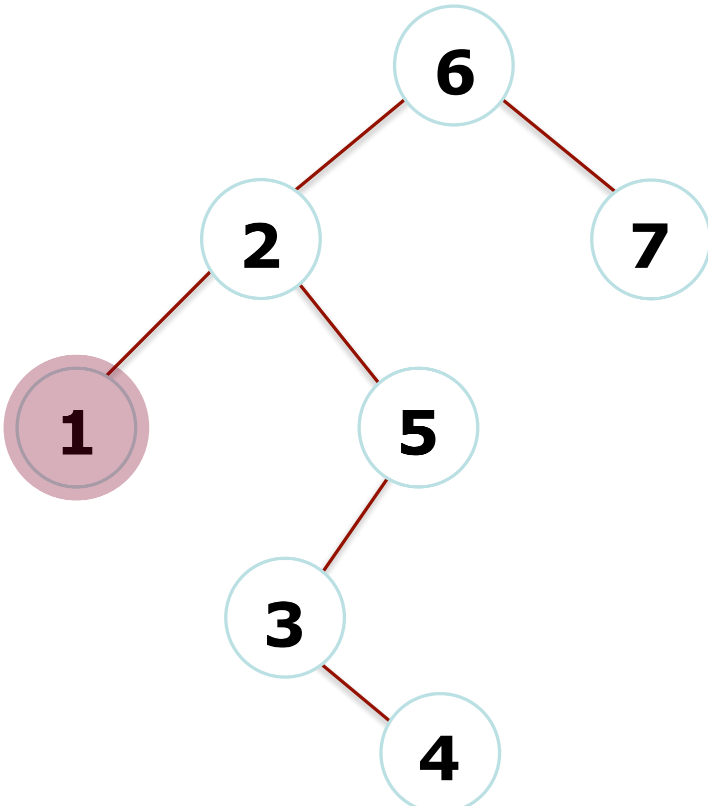


Current Node: NULL

1. current NULL: return

Output: 1

In-Order Traversal Example: printing



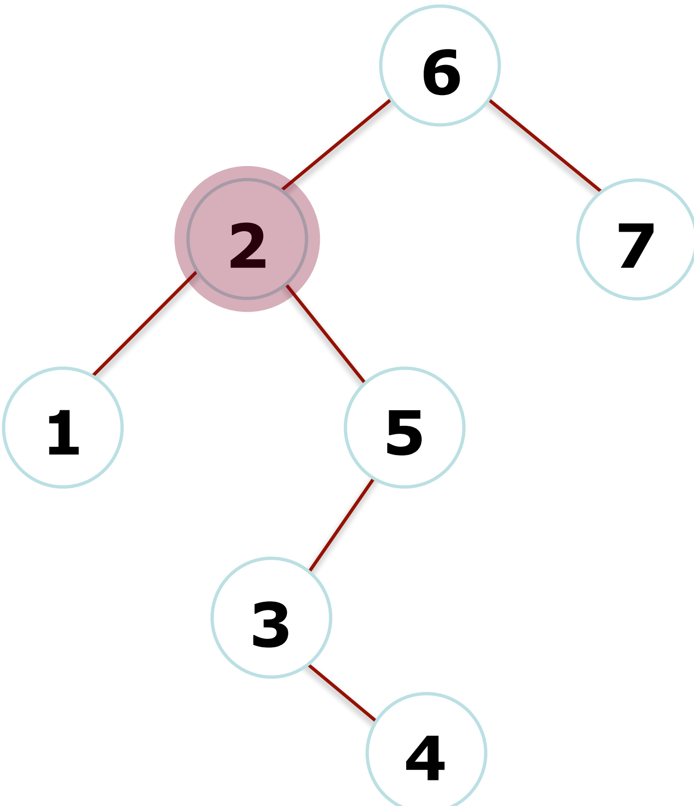
Current Node: 1

1. ~~current not NULL~~
2. ~~recurse left~~
3. ~~print "1"~~
4. ~~recurse right~~

(function ends)

Output: 1

In-Order Traversal Example: printing

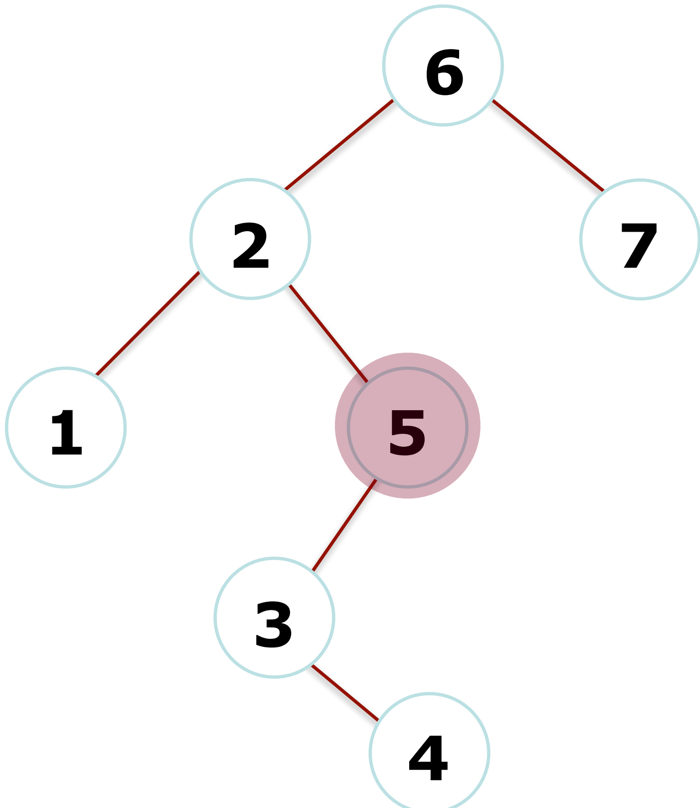


Current Node: 2

1. ~~current not NULL~~
2. ~~recurse left~~
3. print "2"
4. recurse right

Output: 1 2

In-Order Traversal Example: printing

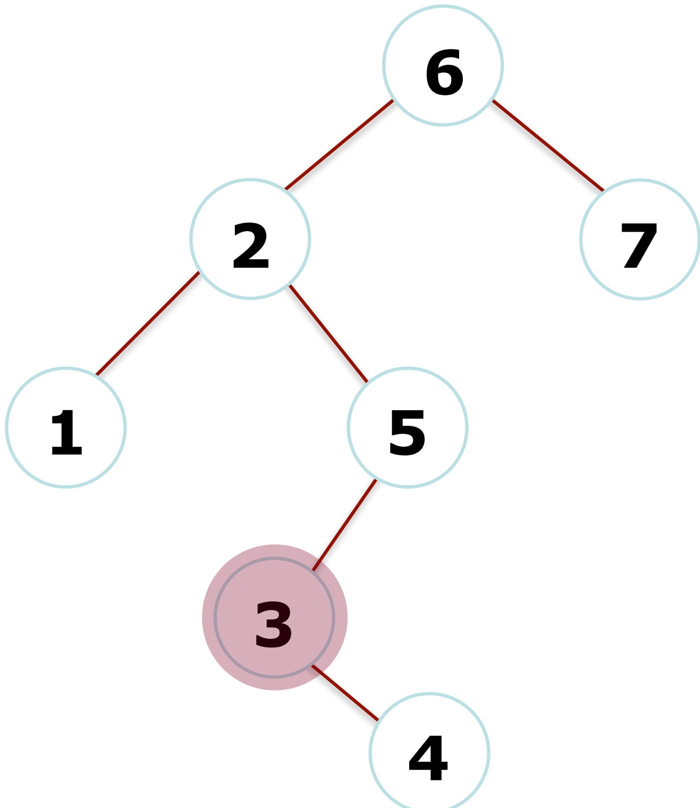


Current Node: 5

1. current not NULL
2. recurse left

Output: 1 2

In-Order Traversal Example: printing

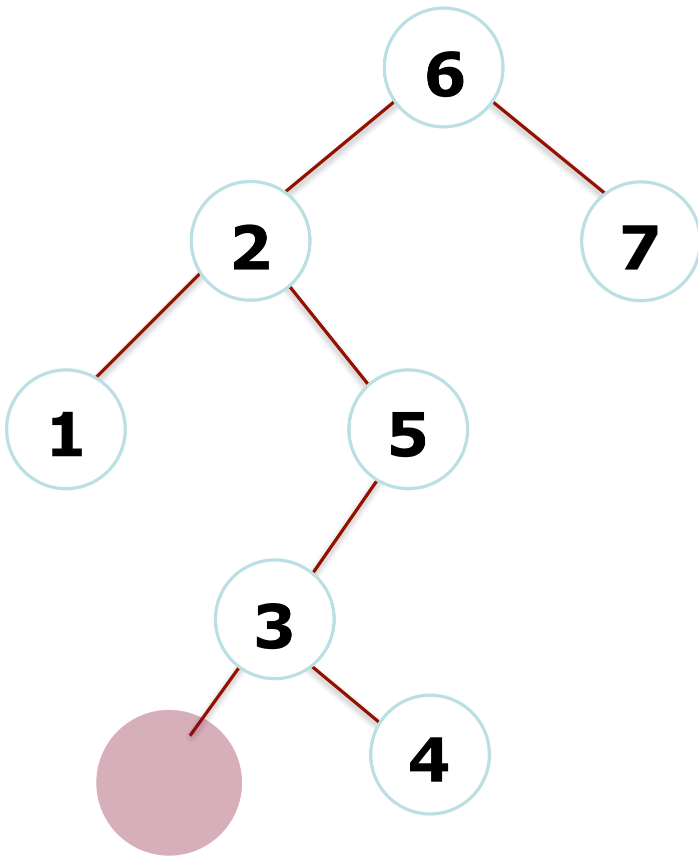


Current Node: 3

1. current not NULL
2. recurse left

Output: 1 2

In-Order Traversal Example: printing

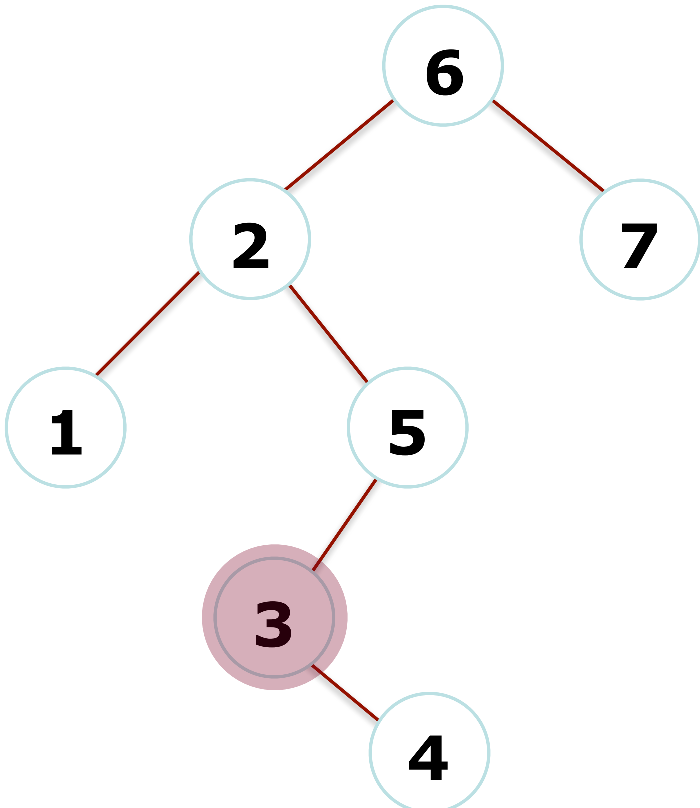


Current Node: NULL

1. current NULL: return

Output: 1 2

In-Order Traversal Example: printing

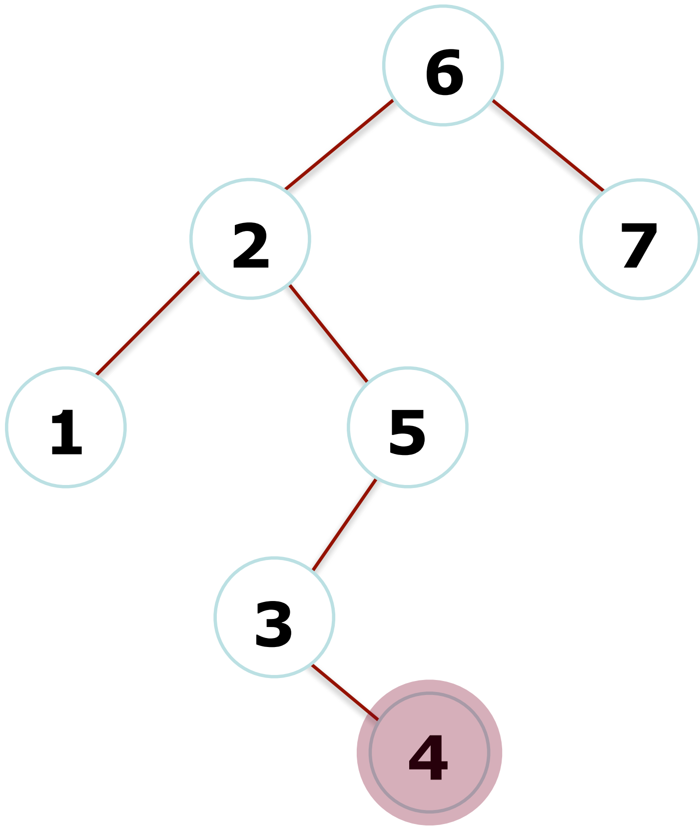


Current Node: 3

1. ~~current not NULL~~
2. ~~recurse left~~
3. print "3"
4. recurse right

Output: 1 2 3

In-Order Traversal Example: printing

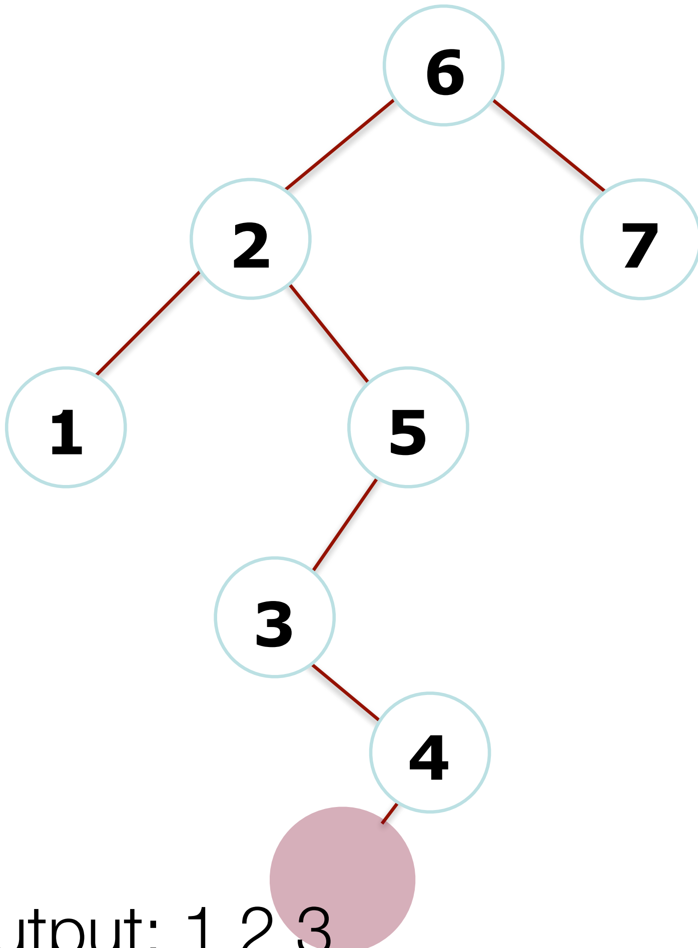


Current Node: 4

1. current not NULL
2. recurse left

Output: 1 2 3

In-Order Traversal Example: printing

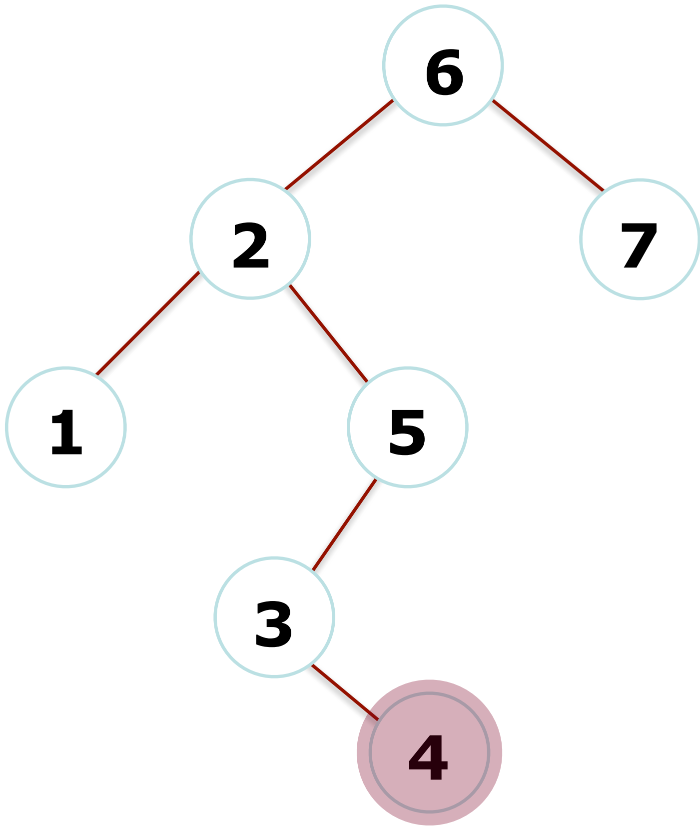


Current Node: NULL

1. current NULL, return

Output: 1 2 3

In-Order Traversal Example: printing

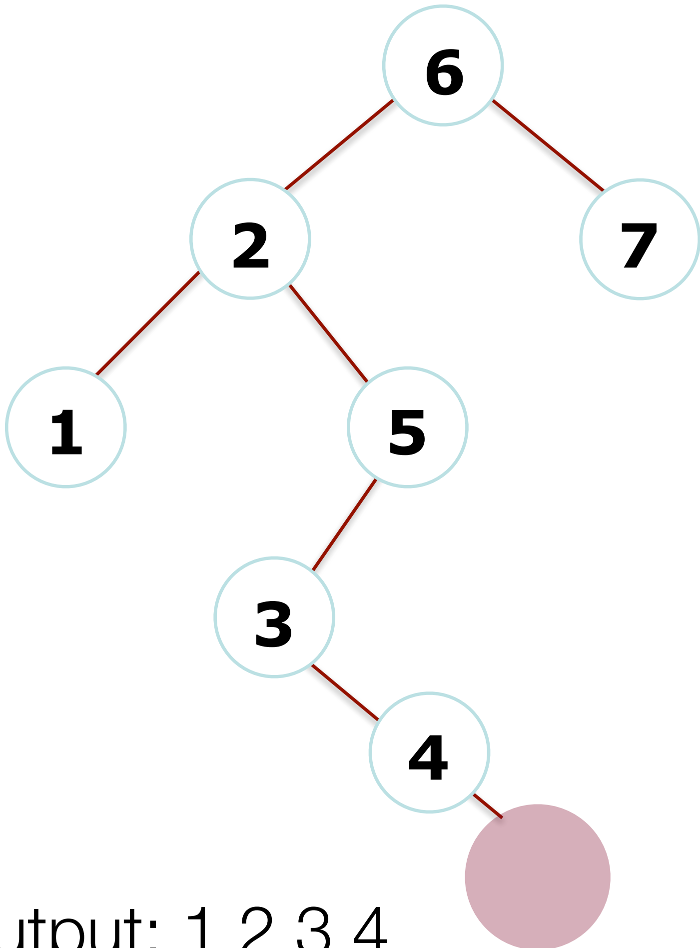


Current Node: 4

1. ~~current not NULL~~
2. ~~recurse left~~
3. print "4"
4. recurse right

Output: 1 2 3 4

In-Order Traversal Example: printing

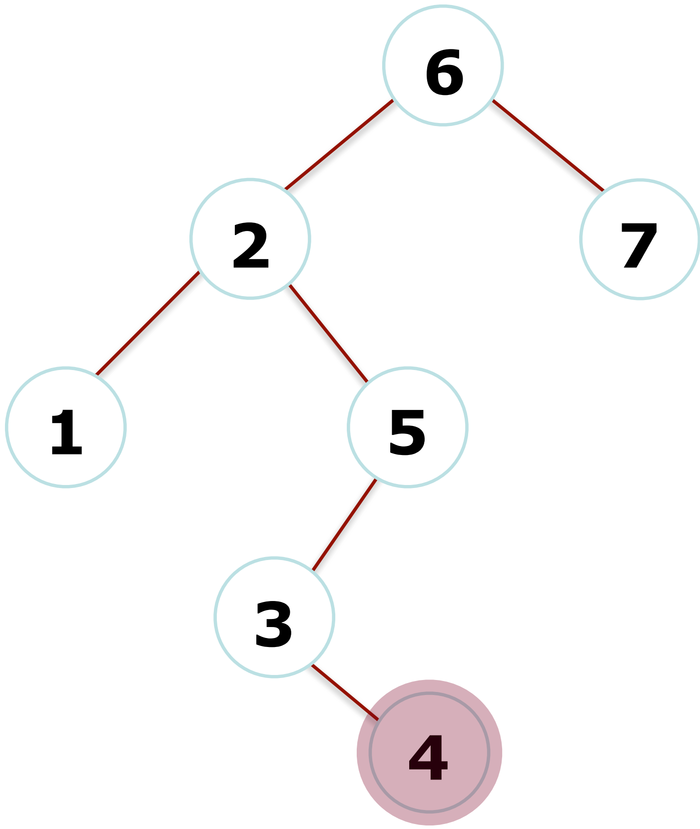


Current Node: NULL

1. current NULL, return

Output: 1 2 3 4

In-Order Traversal Example: printing



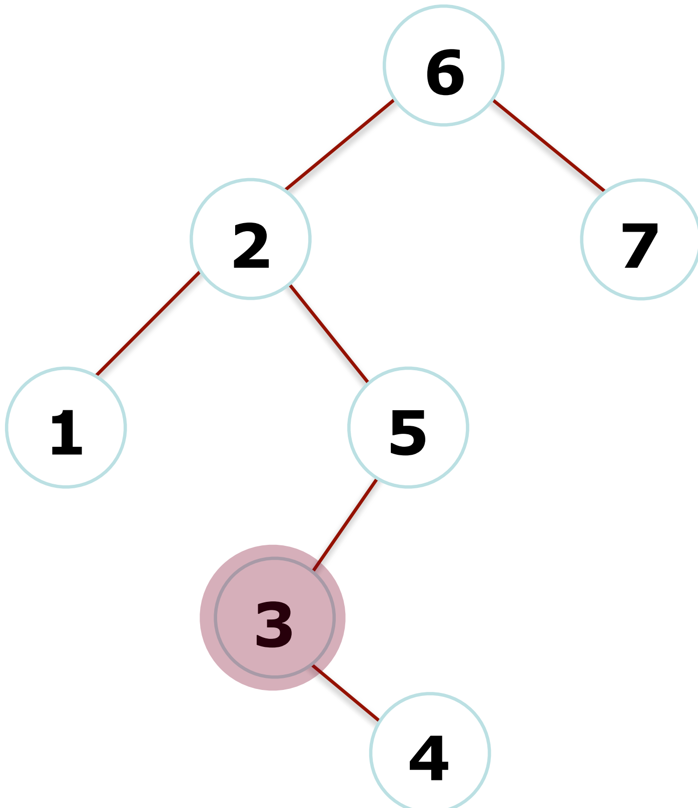
Current Node: 4

1. ~~current not NULL~~
2. ~~recurse left~~
3. ~~print "4"~~
4. ~~recurse right~~

(function ends)

Output: 1 2 3 4

In-Order Traversal Example: printing



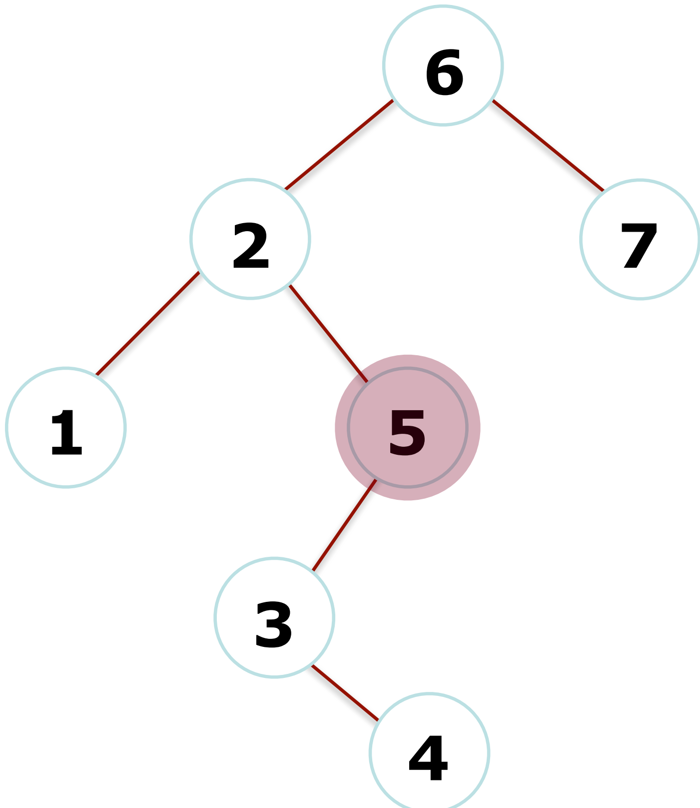
Current Node: 3

1. ~~current not NULL~~
2. ~~recurse left~~
3. ~~print "3"~~
4. ~~recurse right~~

(function ends)

Output: 1 2 3 4

In-Order Traversal Example: printing

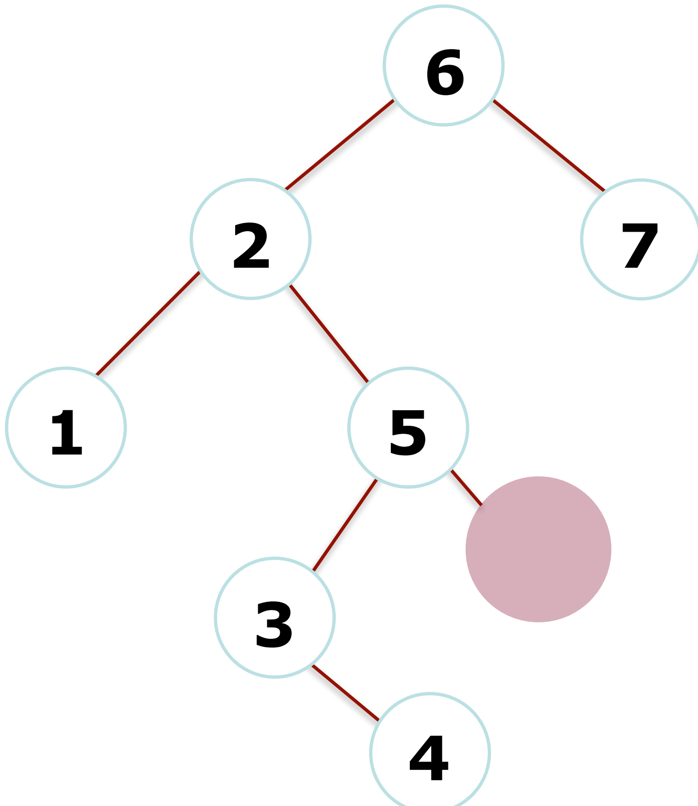


Current Node: 5

1. ~~current not NULL~~
2. ~~recurse left~~
3. print "5"
4. recurse right

Output: 1 2 3 4 5

In-Order Traversal Example: printing

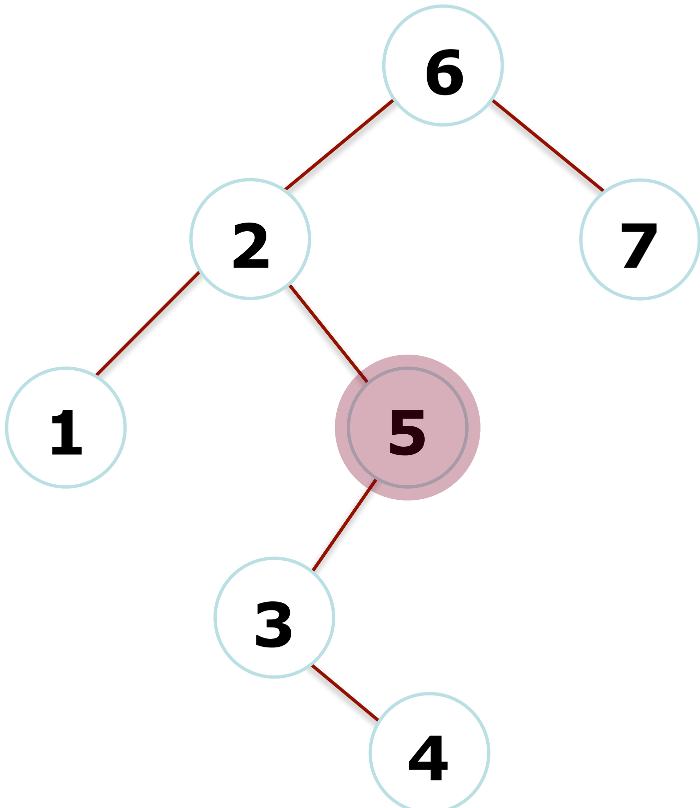


Current Node: NULL

1. current NULL, return

Output: 1 2 3 4 5

In-Order Traversal Example: printing



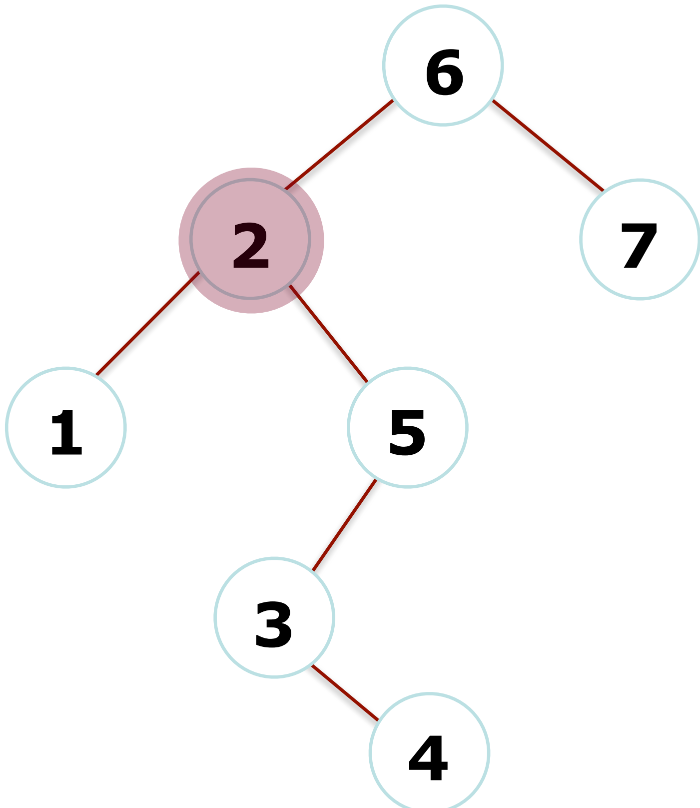
Current Node: 5

1. ~~current not NULL~~
2. ~~recurse left~~
3. ~~print "5"~~
4. ~~recurse right~~

(function ends)

Output: 1 2 3 4 5

In-Order Traversal Example: printing



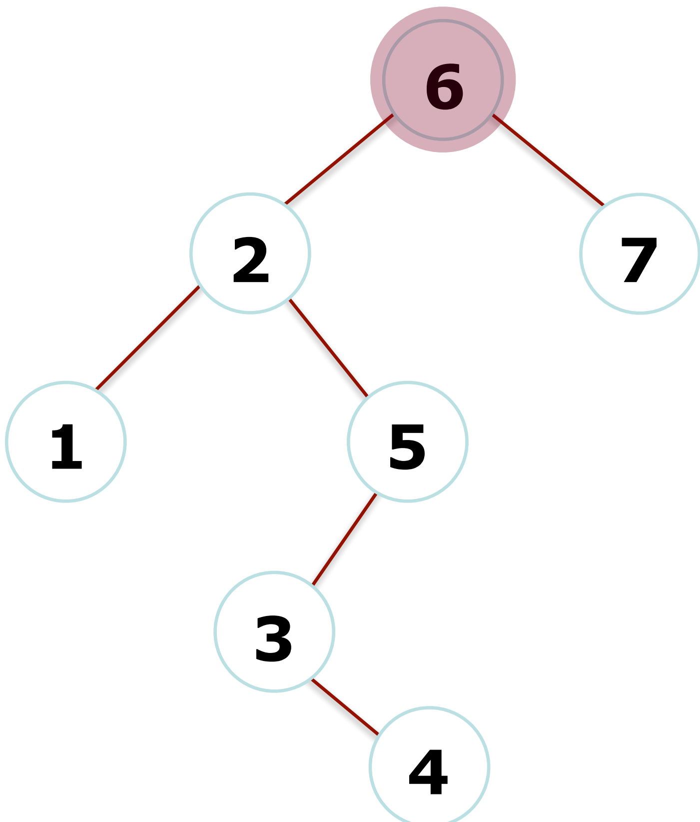
Current Node: 2

1. ~~current not NULL~~
2. ~~recurse left~~
3. ~~print "2"~~
4. ~~recurse right~~

(function ends)

Output: 1 2 3 4 5

In-Order Traversal Example: printing

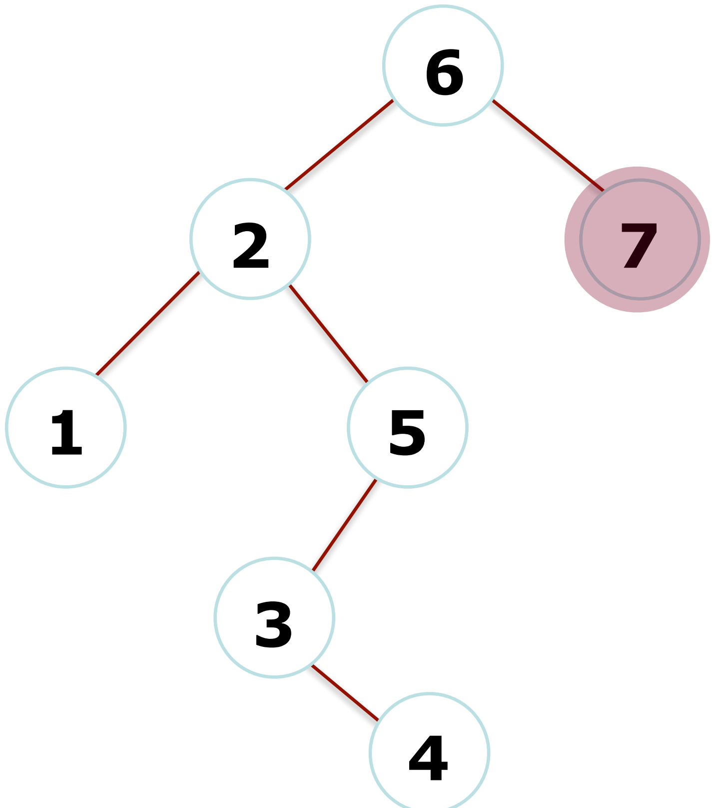


Current Node: 6

1. ~~current not NULL~~
2. ~~recurse left~~
3. print "6"
4. recurse right

Output: 1 2 3 4 5 6

In-Order Traversal Example: printing

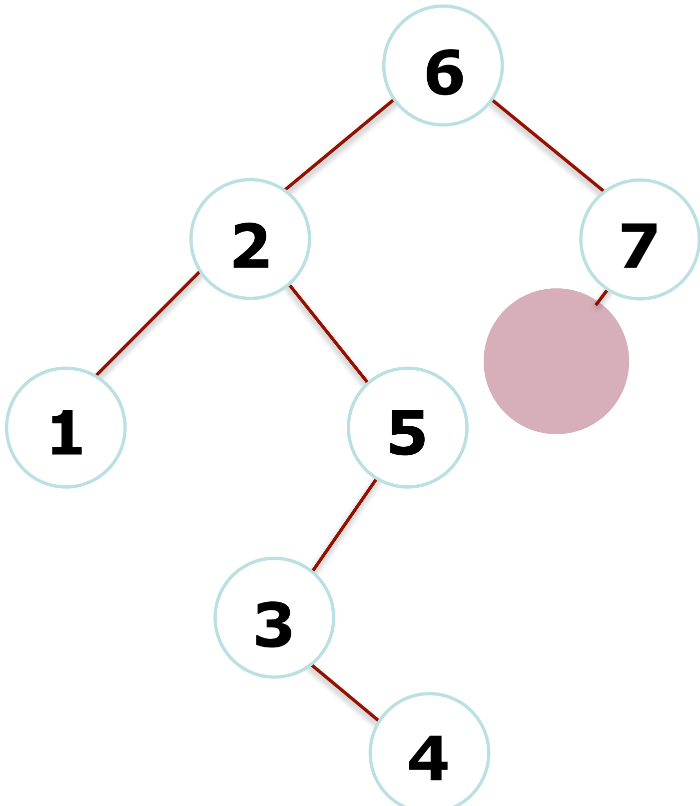


Current Node: 7

1. current not NULL
2. recurse left

Output: 1 2 3 4 5 6

In-Order Traversal Example: printing

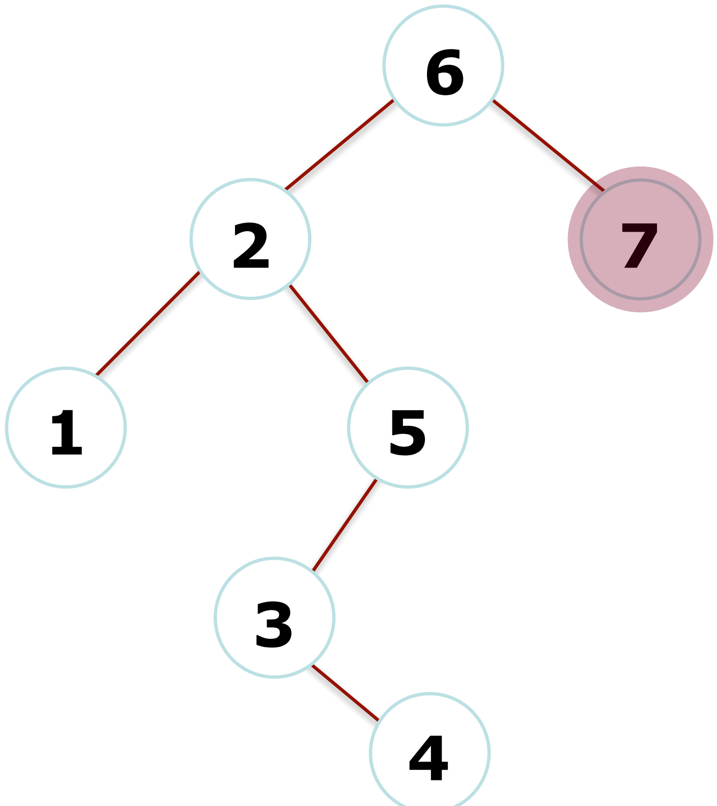


Current Node: NULL

1. current NULL, return

Output: 1 2 3 4 5 6

In-Order Traversal Example: printing

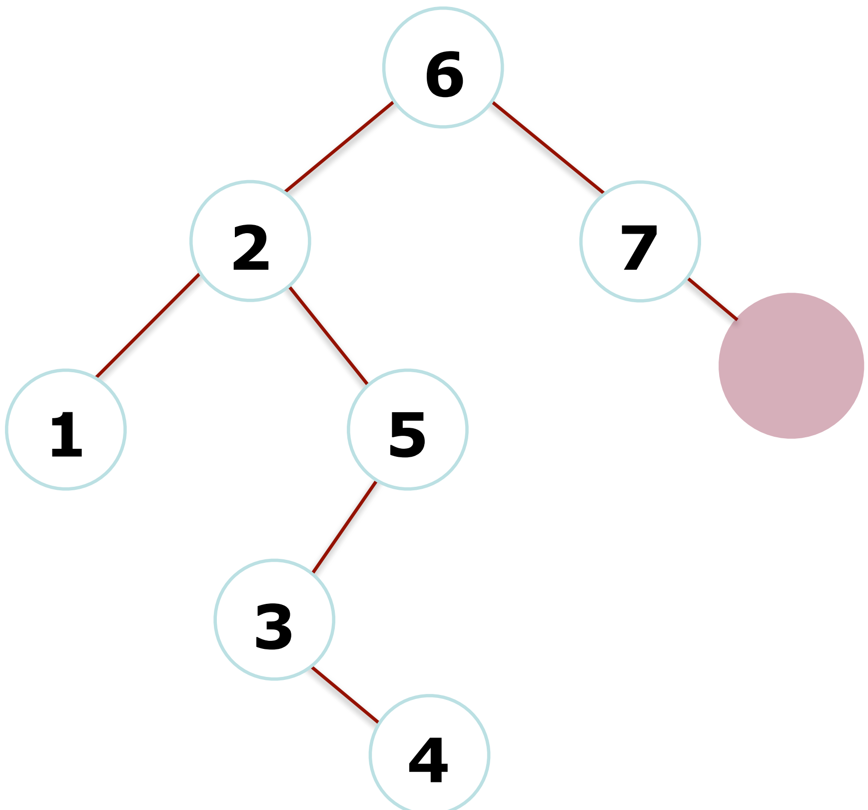


Current Node: 7

1. ~~current not NULL~~
2. ~~recurse left~~
3. print "7"
4. recurse right

Output: 1 2 3 4 5 6 7

In-Order Traversal Example: printing

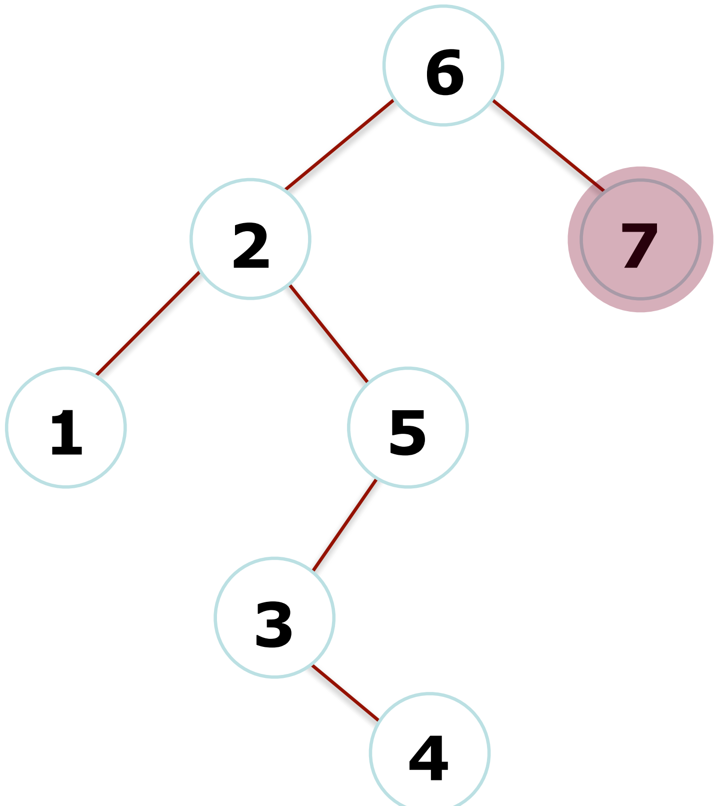


Current Node: NULL

1. current NULL, return

Output: 1 2 3 4 5 6

In-Order Traversal Example: printing



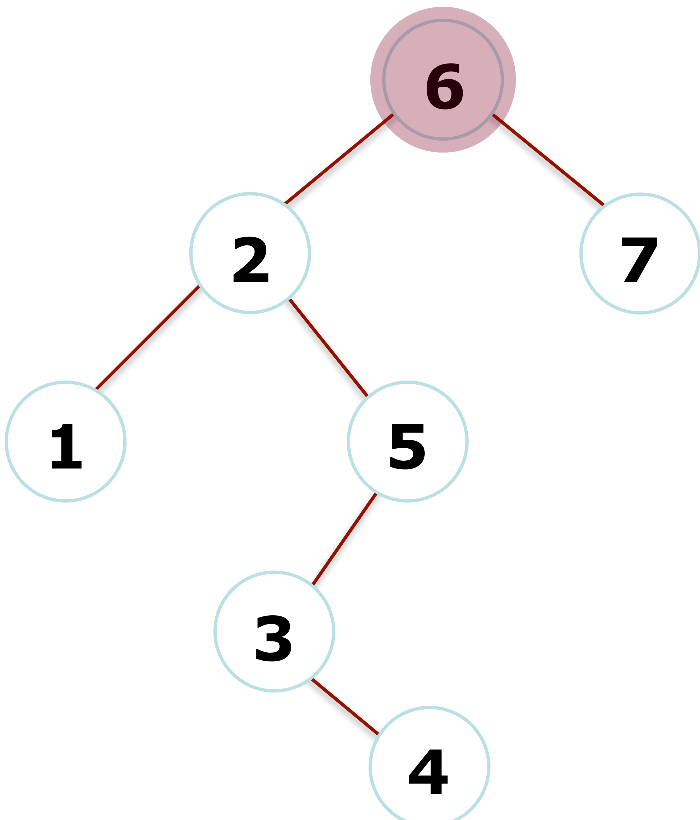
Current Node: 7

1. ~~current not NULL~~
2. ~~recurse left~~
3. ~~print "7"~~
4. ~~recurse right~~

(function ends)

Output: 1 2 3 4 5 6 7

In-Order Traversal Example: printing



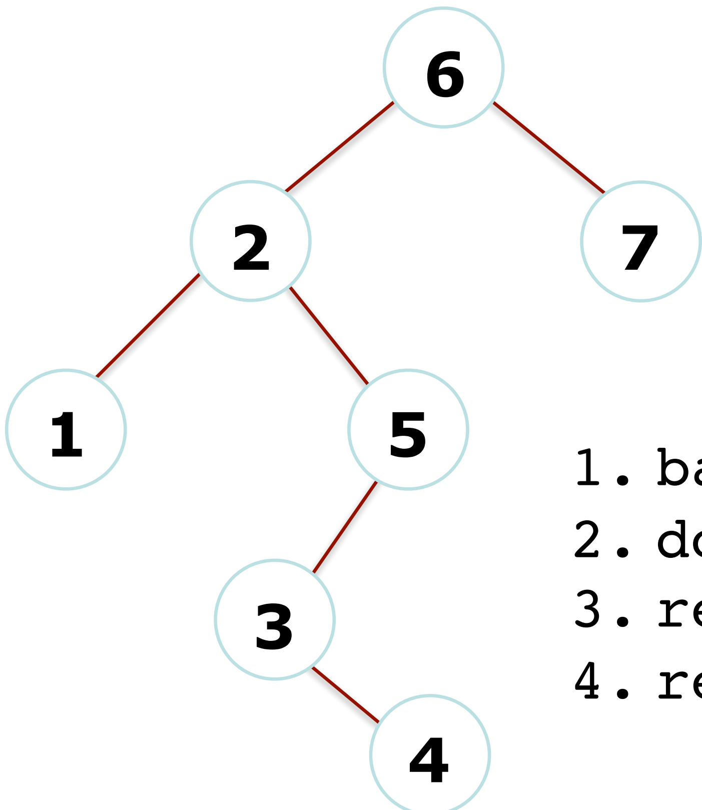
Current Node: 6

1. ~~current not NULL~~
2. ~~recurse left~~
3. ~~print "6"~~
4. ~~recurse right~~

(function ends)

Output: 1 2 3 4 5 6 7

Pre-Order Traversal



Pseudocode:

```
1. base case: if current == NULL, return
2. do something with current node
3. recurse left
4. recurse right
```

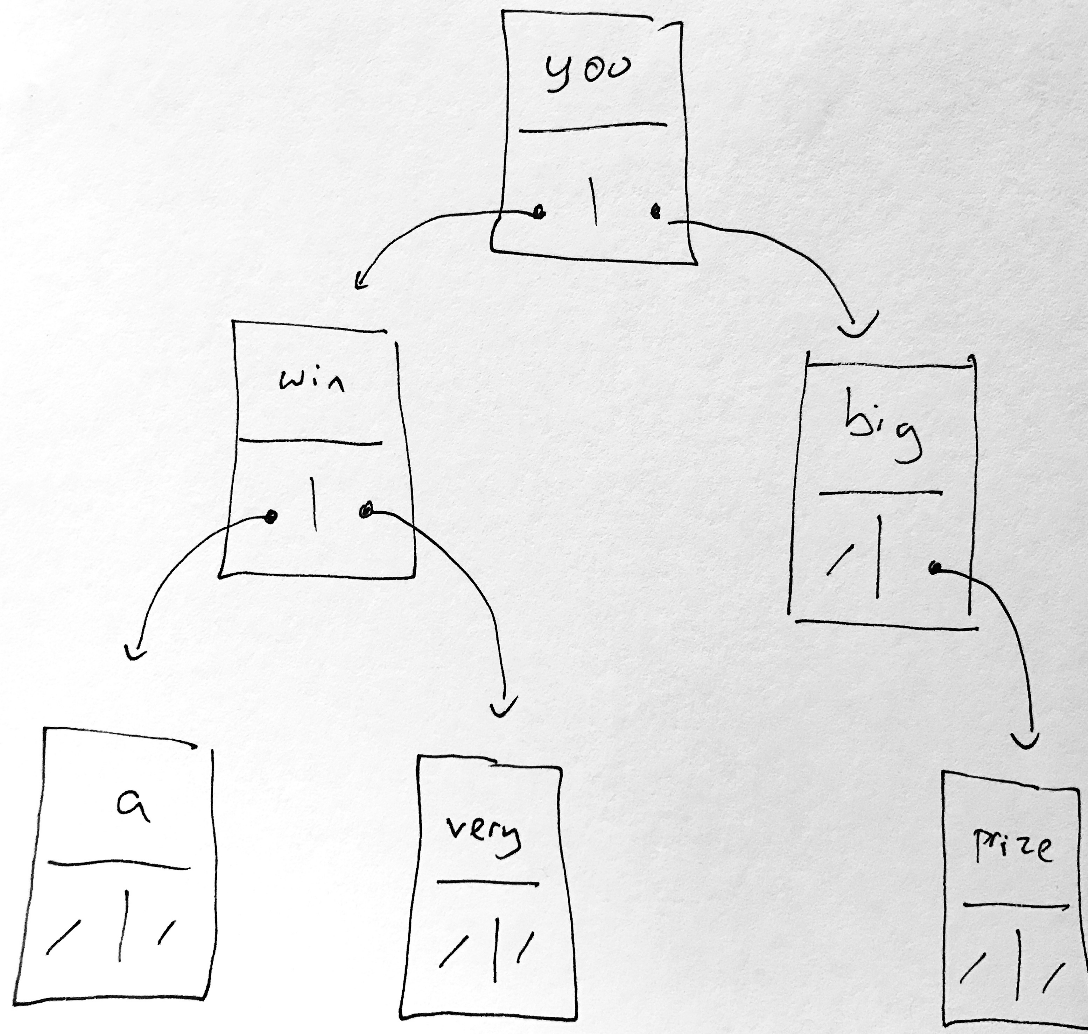
Output: 6 2 1 5 3 4 7

Game Show Tree

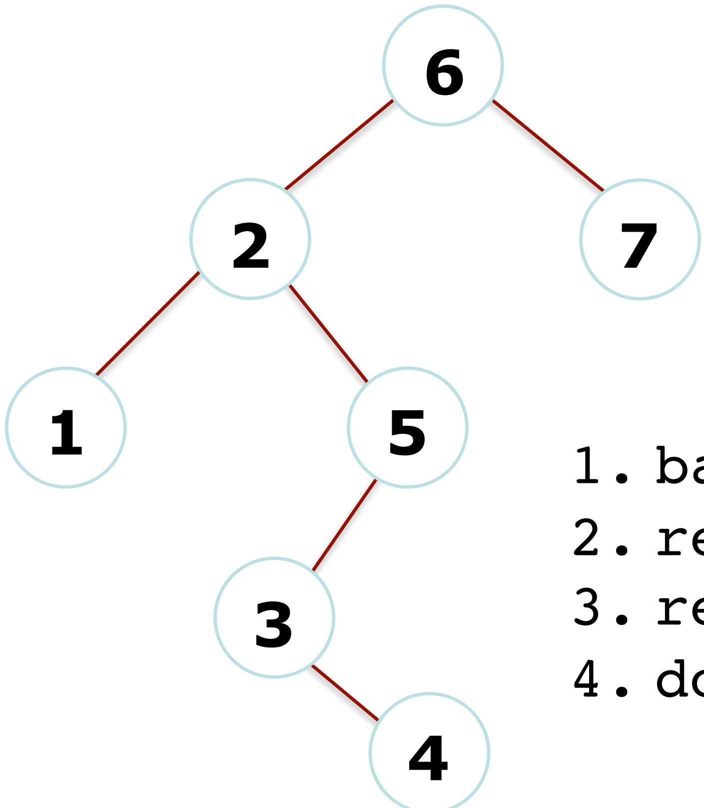
Winner!

A

```
void doorOne(Tree * tree) {
  if(tree == NULL) return;
  cout<<tree->value<<" ";
  doorOne(tree->left);
  doorOne(tree->right);
}
```



Post-Order Traversal



Pseudocode:

```
1. base case: if current == NULL, return
2. recurse left
3. recurse right
4. do something with current node
```

Output: 1 4 3 5 2 7 6

Coding up a StringSet

```
struct Node {
    string str;
    Node *left;
    Node *right;

    // constructor for new Node
    Node(string s) {
        str = s;
        left = NULL;
        right = NULL;
    }
};

class StringSet {
    ...
}
```

References and Advanced Reading

- **References:**
  - http://www.openbookproject.net/thinkcs/python/english2e/ch21.html
  - https://www.tutorialspoint.com/data_structures_algorithms/binary_search_tree.htm
  - https://en.wikipedia.org/wiki/Binary_search_tree

- **Advanced Reading:**
  - Tree (abstract data type), Wikipedia: http://en.wikipedia.org/wiki/Tree_(data_structure)
  - Binary Trees, Wikipedia: http://en.wikipedia.org/wiki/Binary_tree
  - Tree visualizations: http://vcg.informatik.uni-rostock.de/~hs162/treeposter/poster.html
  - Wikipedia article on self-balancing trees (be sure to look at all the implementations): http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

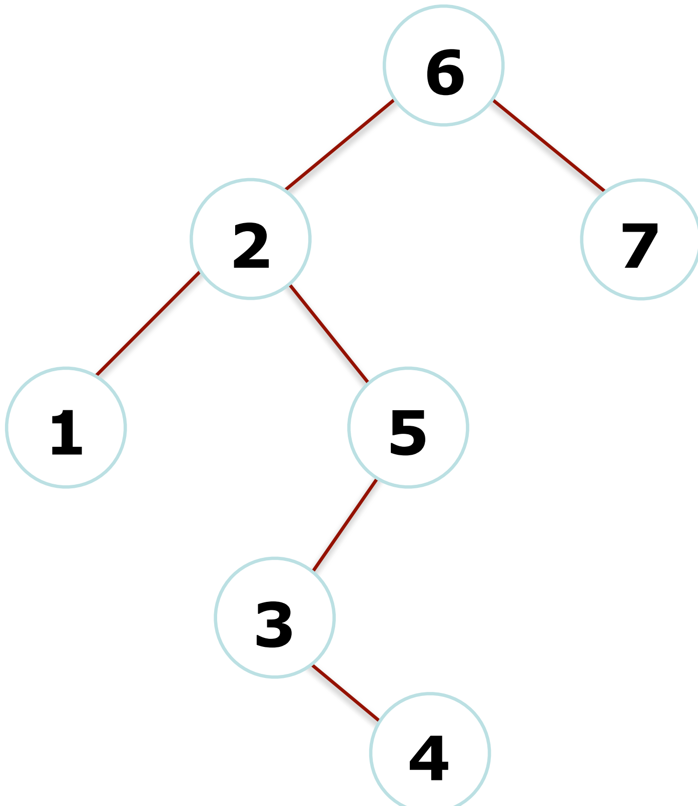
  - Red Black Trees:
  - https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html

  - YouTube AVL Trees: http://www.youtube.com/watch?v=YKt1kquKScY
  - Wikipedia article on AVL Trees: http://en.wikipedia.org/wiki/AVL_tree

  - Really amazing lecture on AVL Trees: https://www.youtube.com/watch?v=FNeL18KsWPc

Extra Slides

Level-Order Traversal



Output: 6 2 7 1 5 3 4

Level-order traversal:

We need a queue, and we can't do this recursively.

Pseudocode:

```
insert root into queue
while queue not empty:
    dequeue node
    print node value
    enqueue left
    enqueue right
```

Some Balanced Tree Data Structures

2-3 tree
AA tree
AVL tree
Red-black tree
Scapegoat tree
Splay tree
Treap