# CS 106B

## Lecture 20: Binary Search Trees

Wednesday, May 17, 2017

---

Programming Abstractions
Spring 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg

reading:
Programming Abstractions in C++, Sections 16.1-16.3 (binary search trees), Chapter 15 (hashing)

# Today's Topics

- Logistics
  - Assignment 5 handout: not our best work…very sorry about that.
    - Because of our mistakes, we have a gift…
  - I will post a "Tiny Feedback questions and answers" page soon

- Binary Search Trees
  - Definition
  - Traversing
  - Tree functions
  - References to pointers
- Keeping Trees Balanced

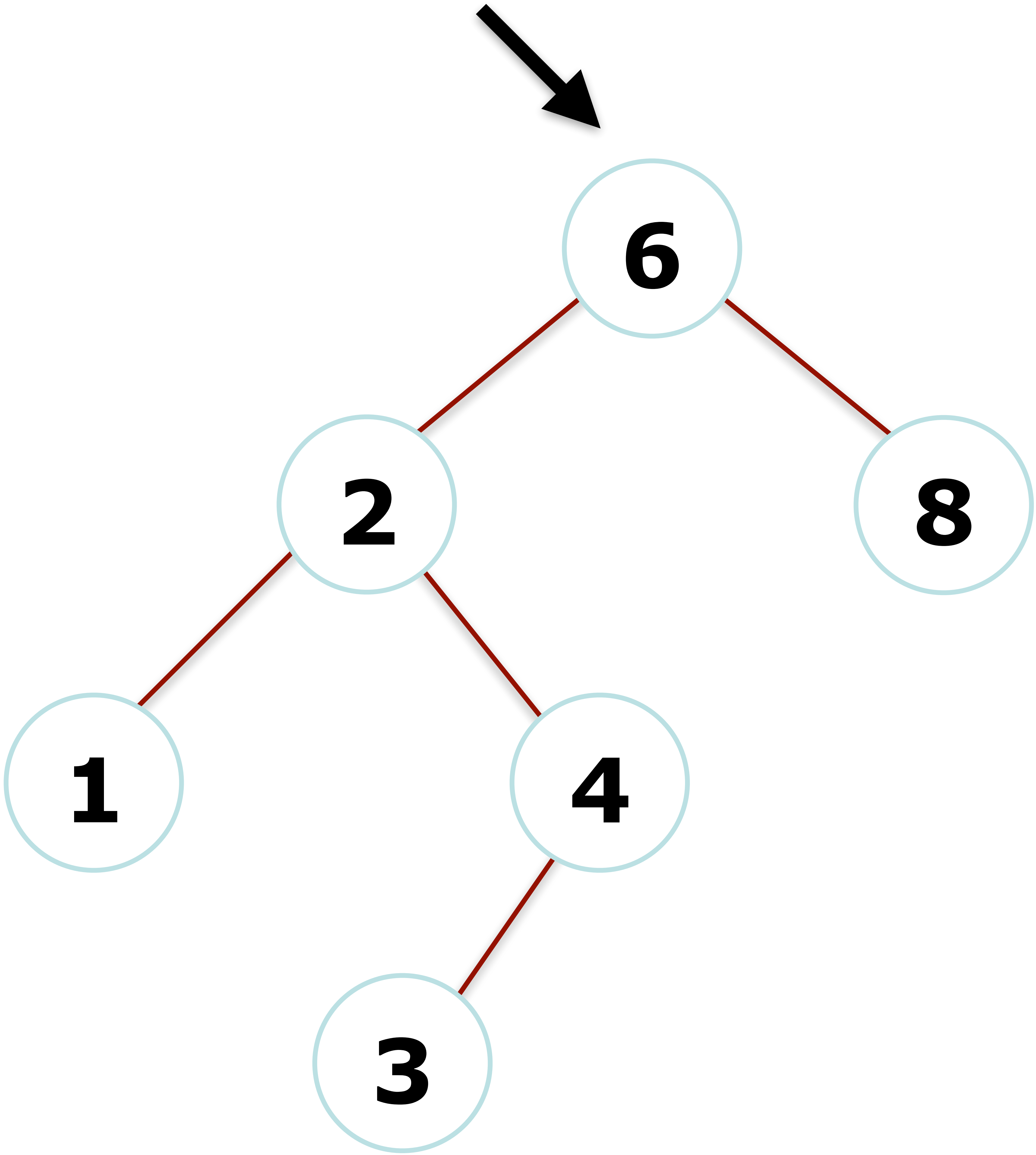
# Binary Search Trees

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

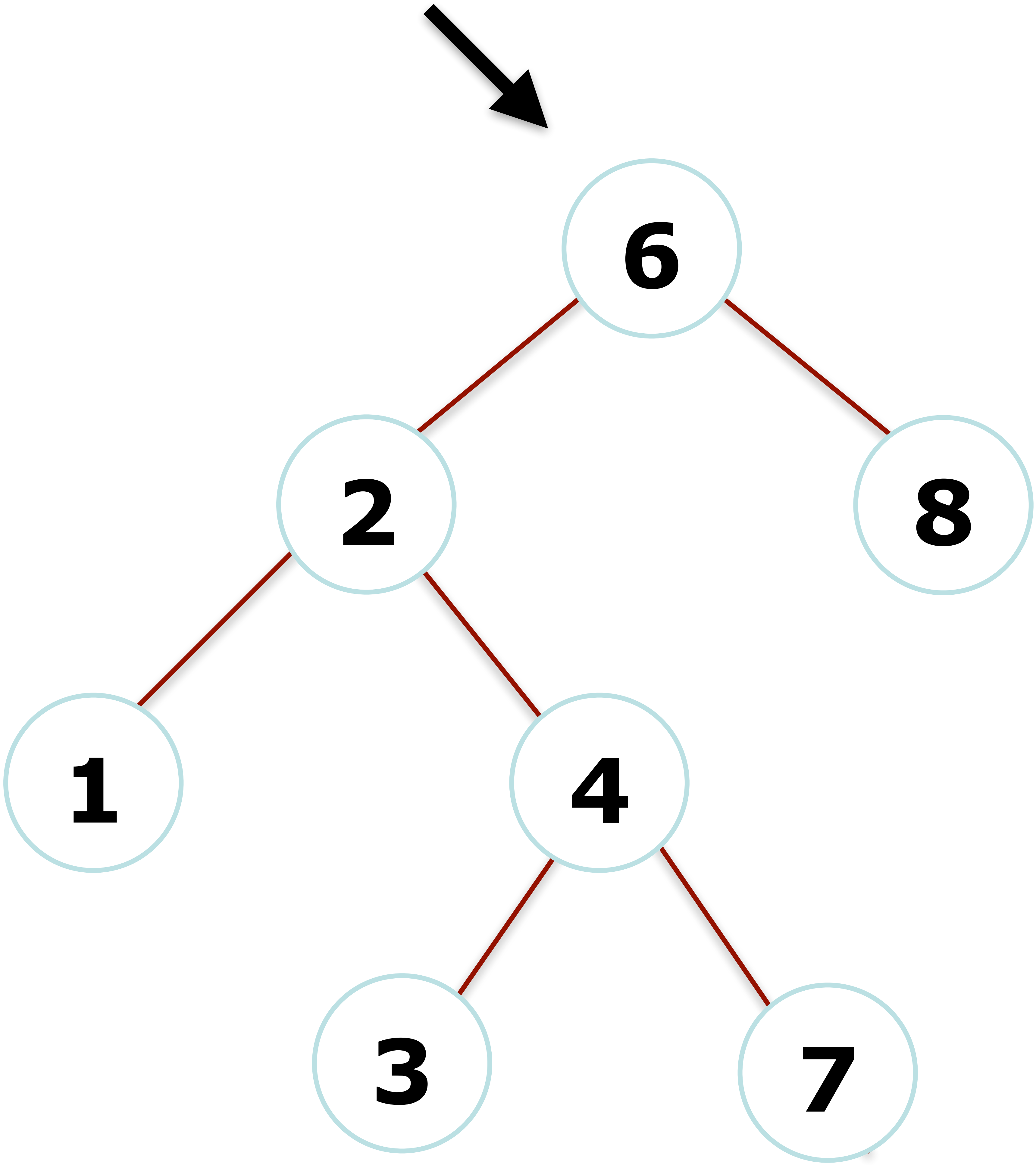
For every node, X, all the items in its left subtree are smaller than X, and the items in the right tree are larger than X.

# Binary Search Trees

Binary Search Tree

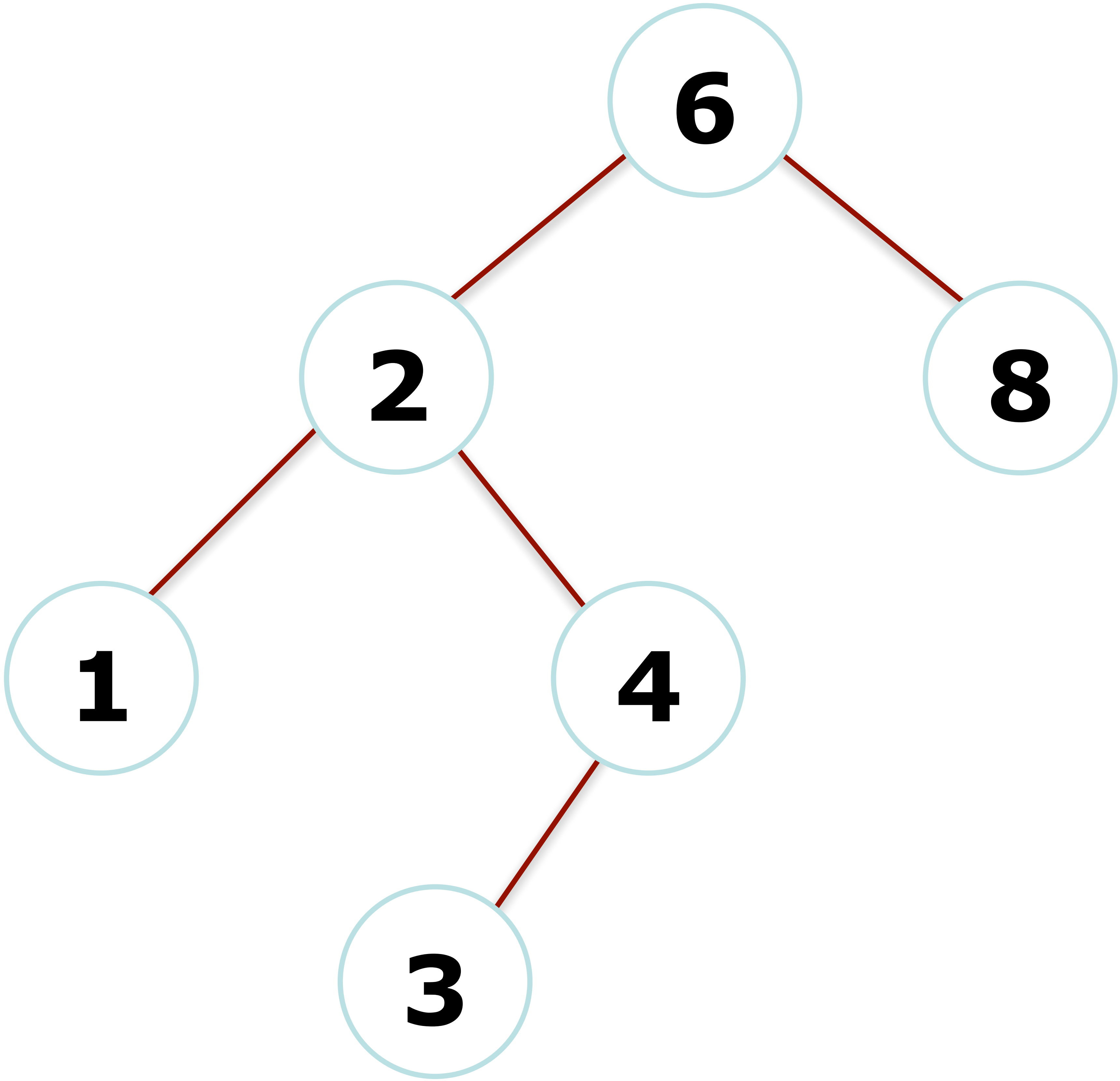



Not a Binary Search Tree



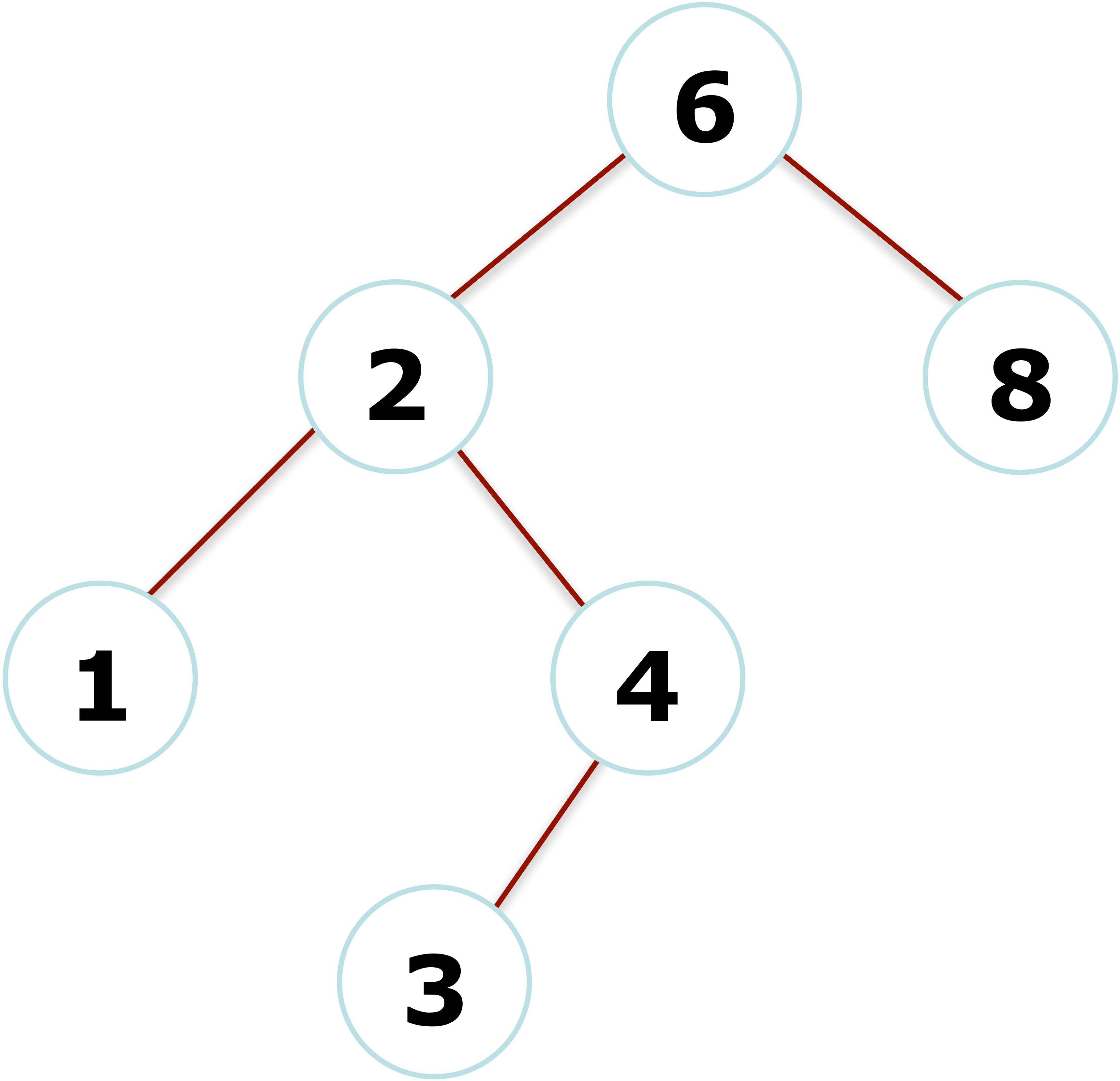
# Binary Search Trees

Binary Search Trees (if built well) 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!)



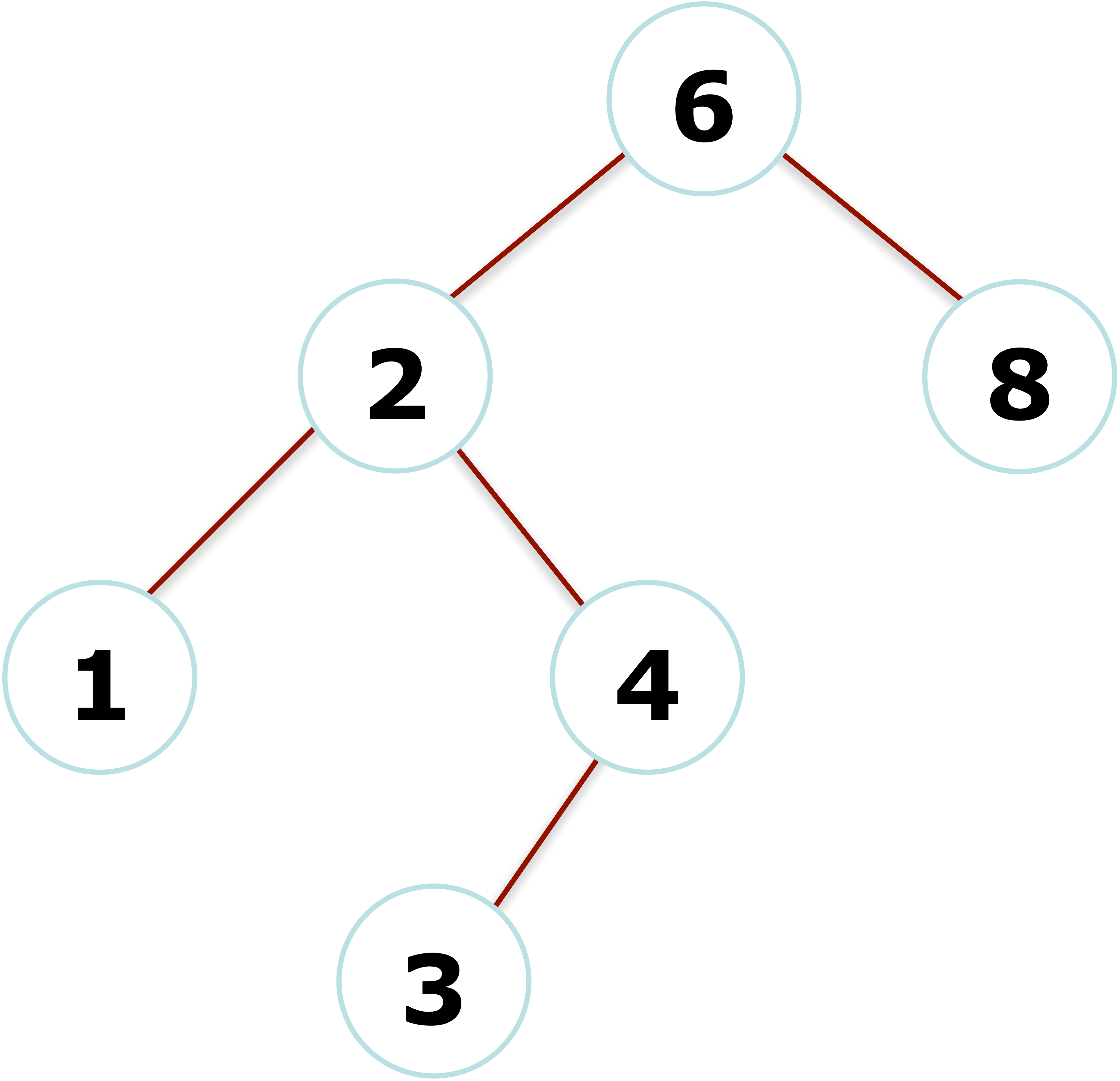

Easy methods:

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

Hard method:

5. remove()

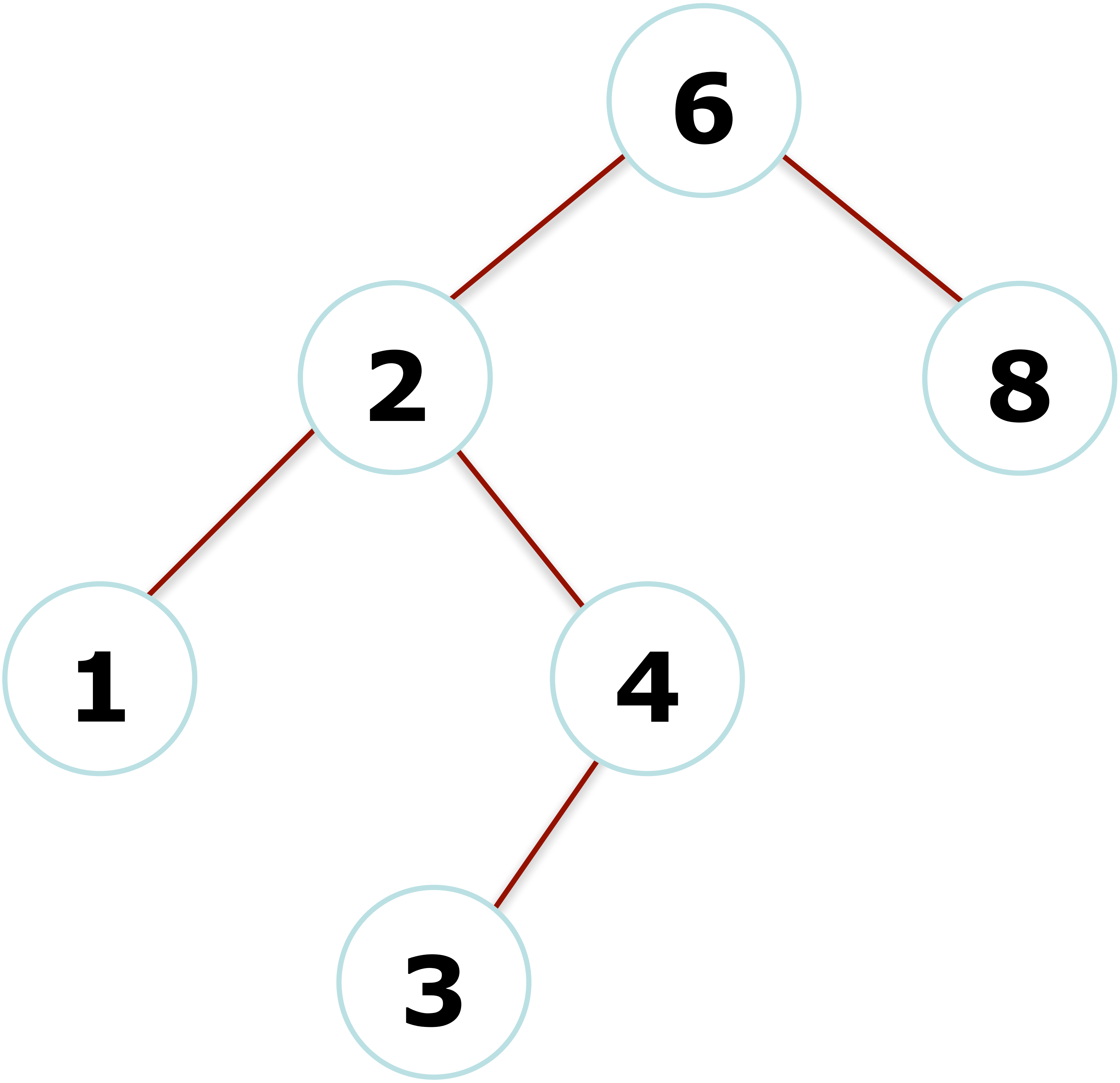
# Binary Search Trees: findMin()




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.

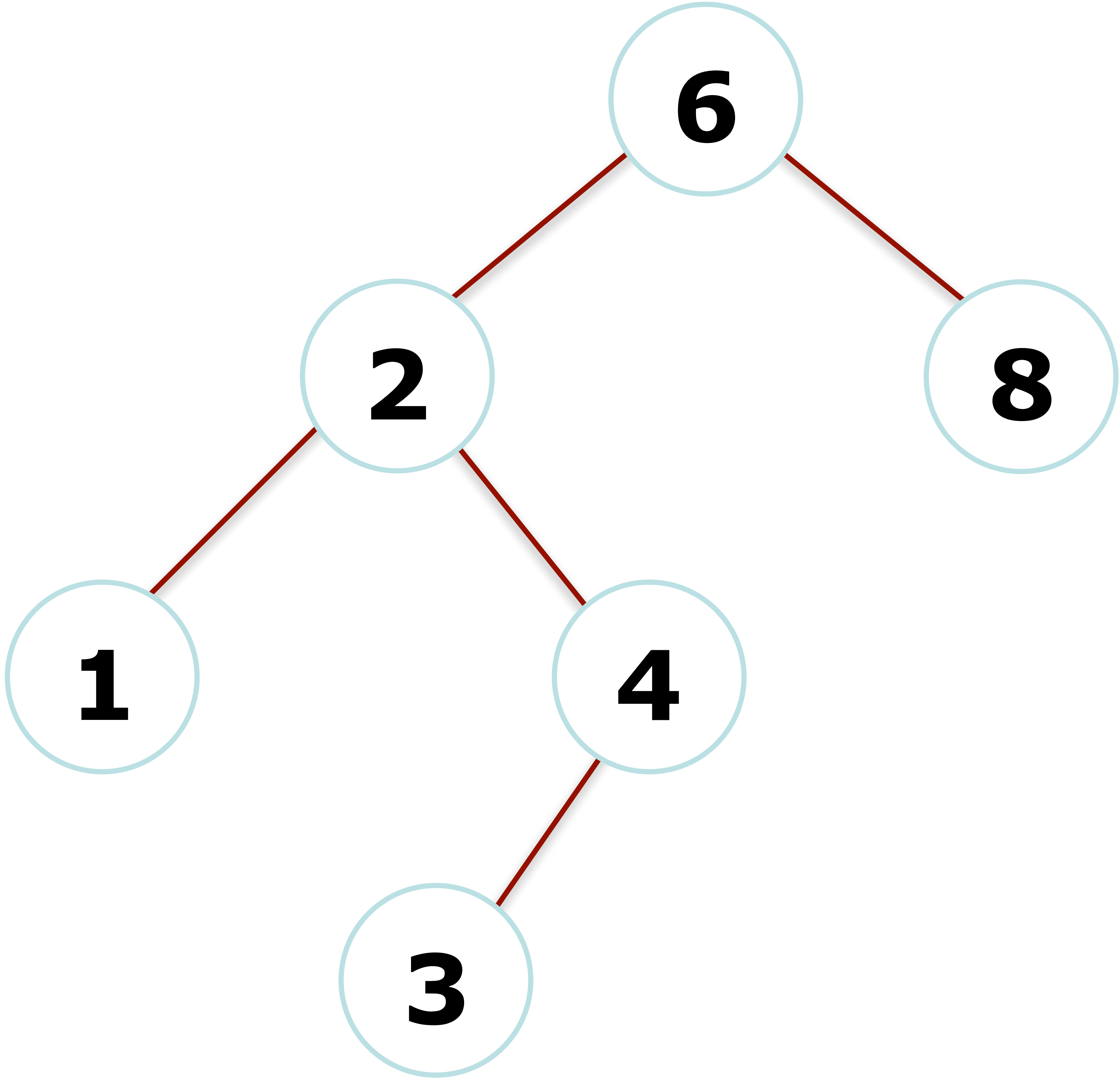
# Binary Search Trees: contains()




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)

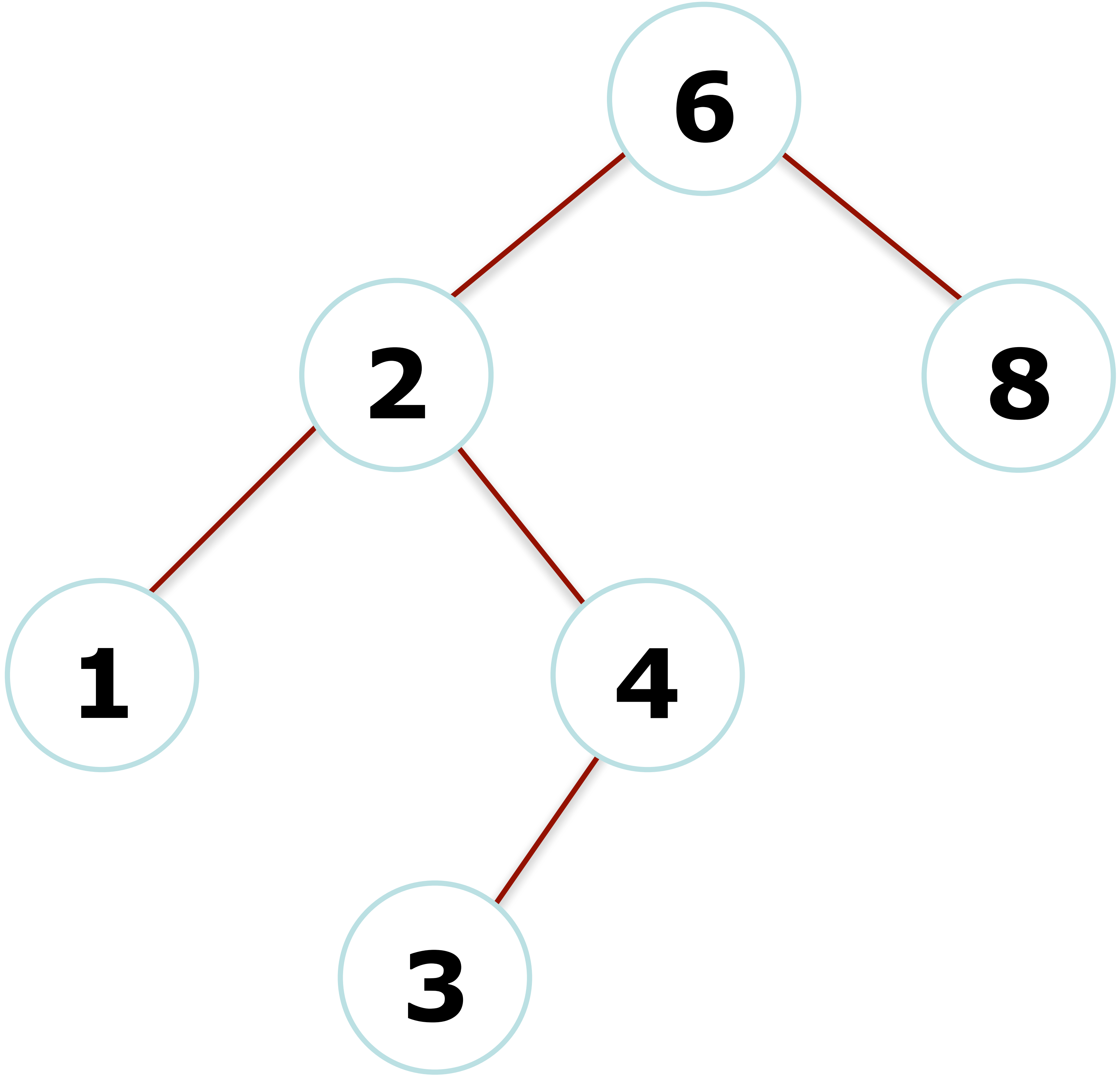



Similar to contains()

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

How do we add 5?

# Binary Search Trees: remove(value)

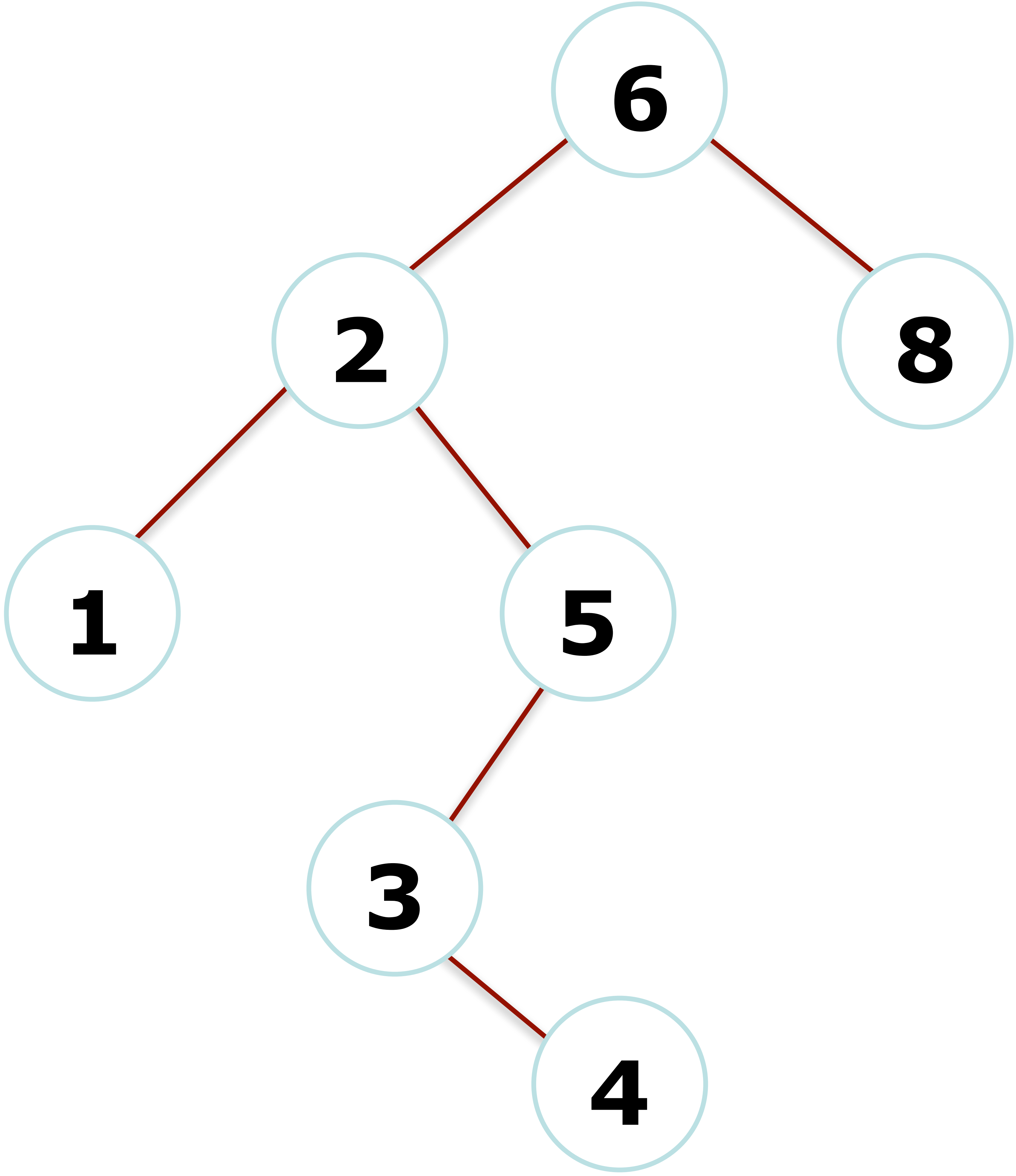



How do we delete 4?

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)



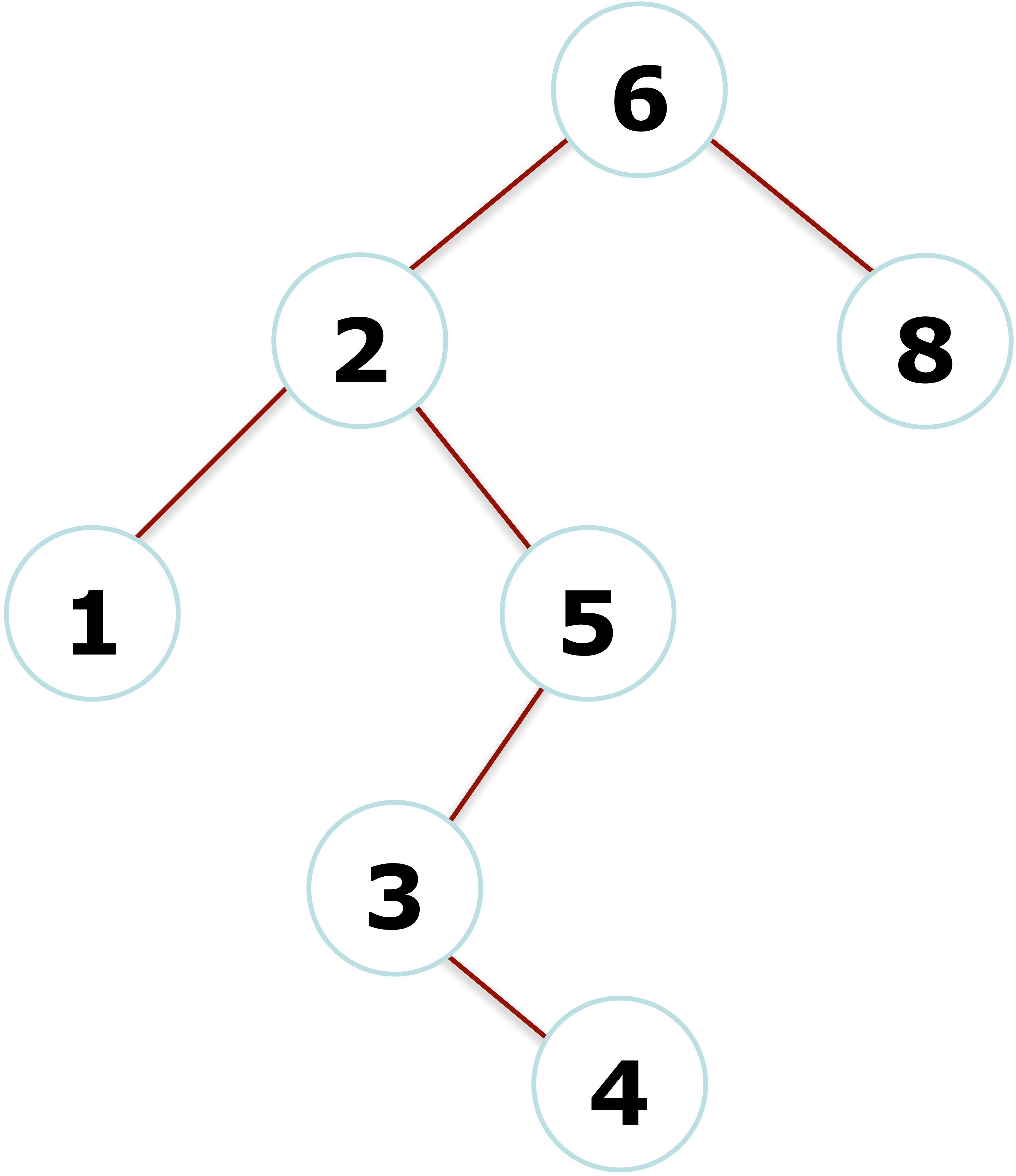

How do we remove 2?

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)

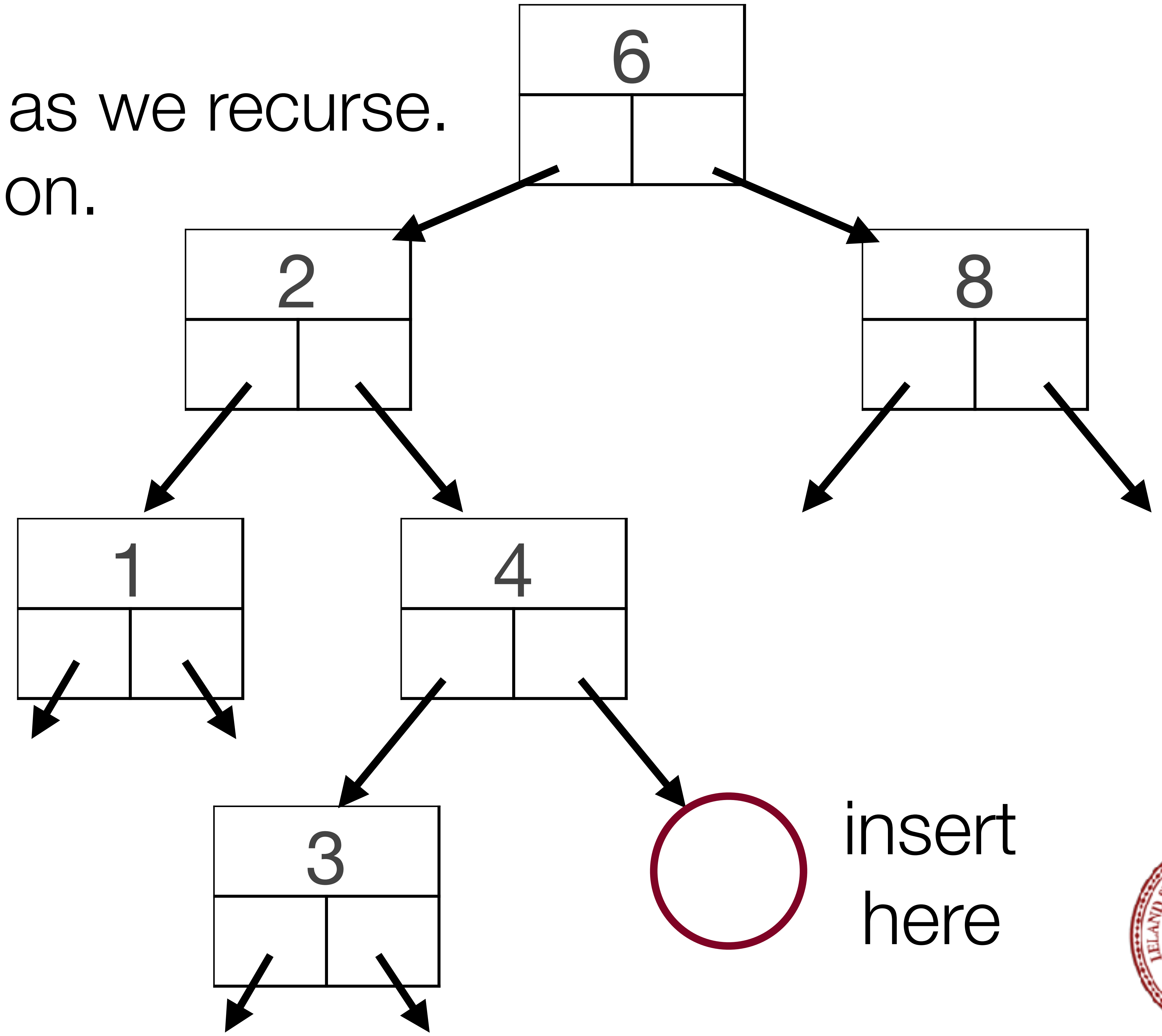
remove()? O(log n)

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

# Using References to Pointers

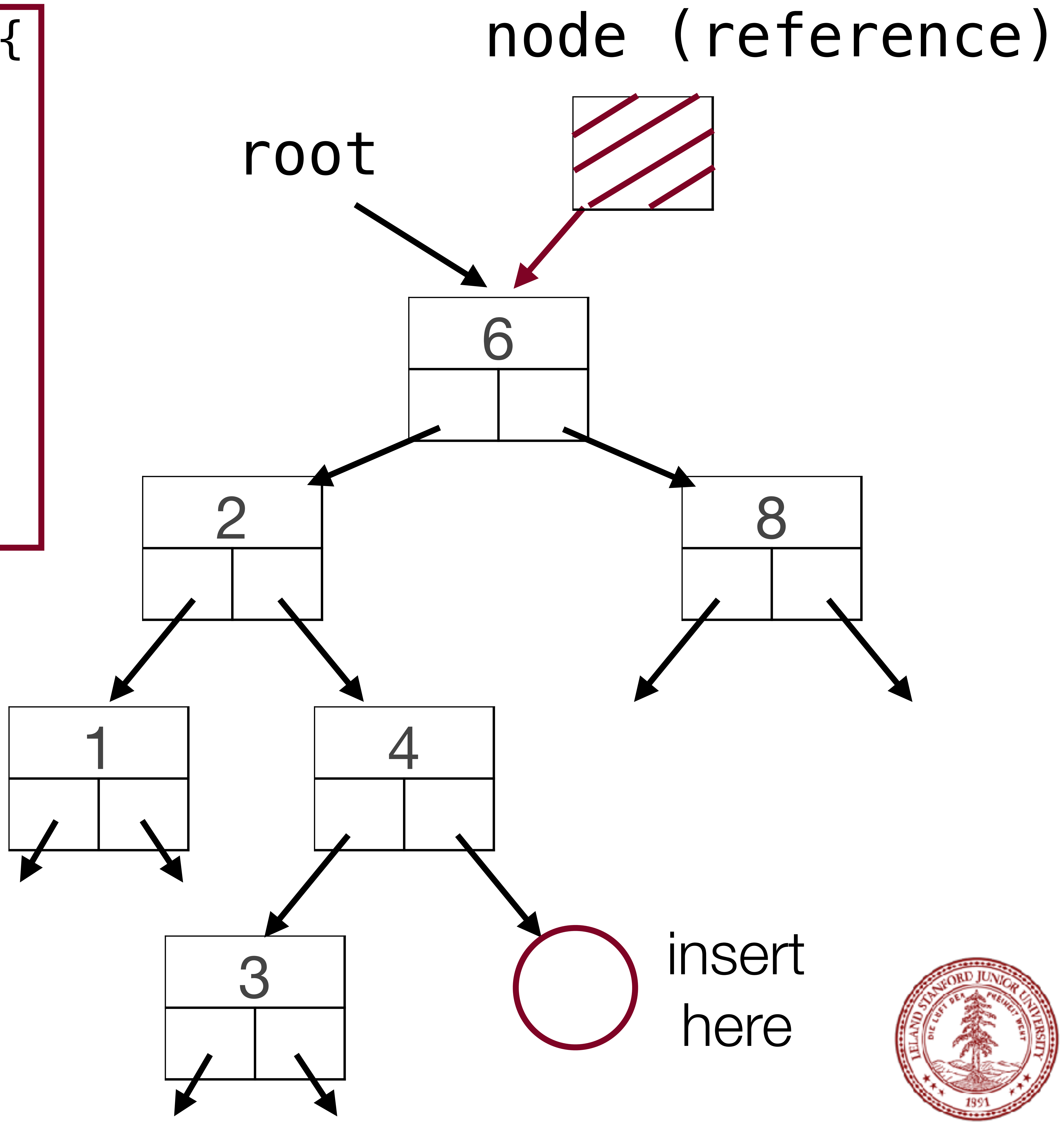
- To insert into a binary search tree, we must update the left or right pointer of a node when we find the position where the new node must go.
- In principle, this means that we could either
  1. Perform arms-length recursion to determine if the child in the direction we will insert is NULL, or
  2. Pass a *reference to a* pointer to the parent as we recurse.
- The second choice above is the cleaner solution.

```
set.add(5)
```



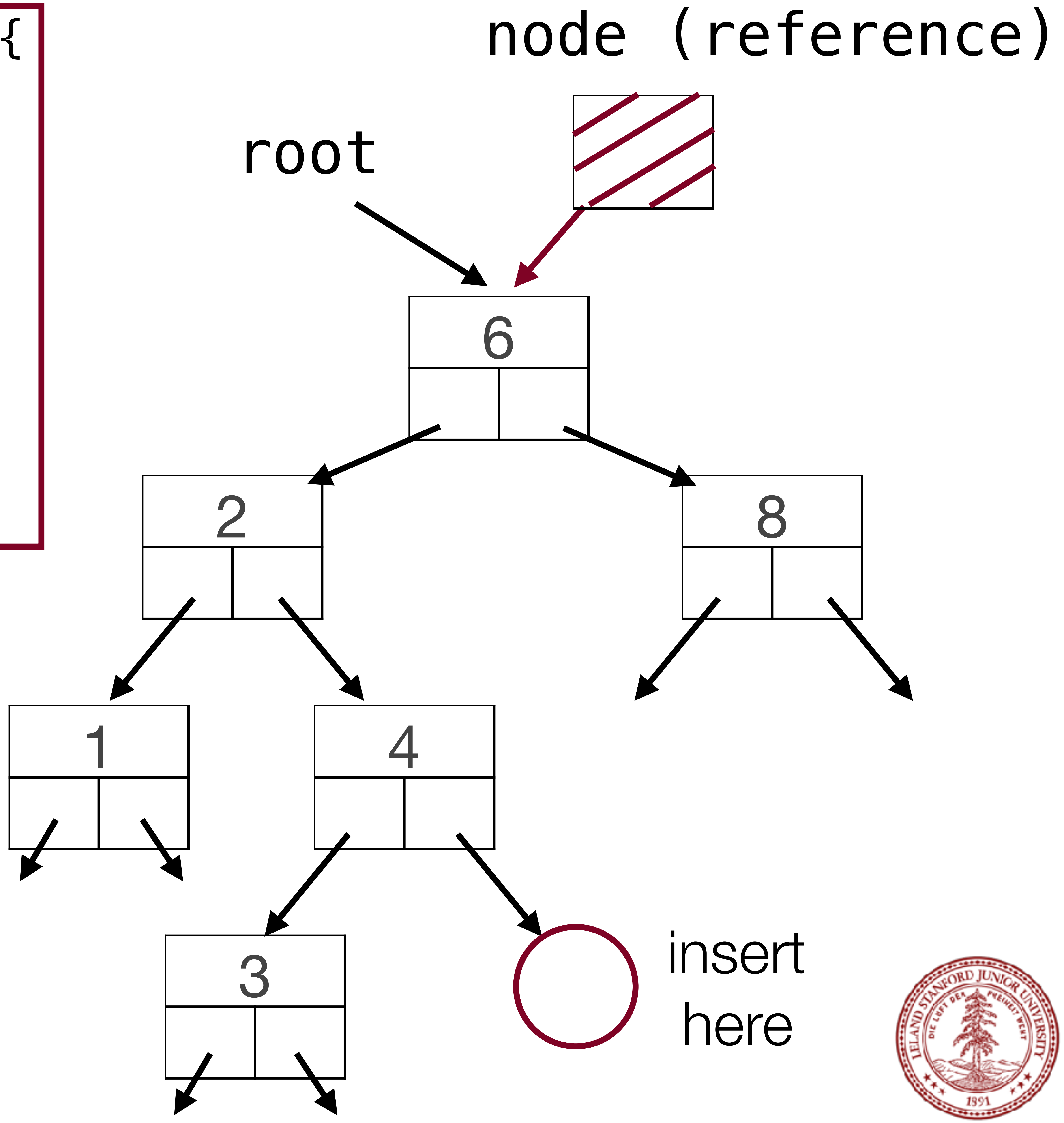
# Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```



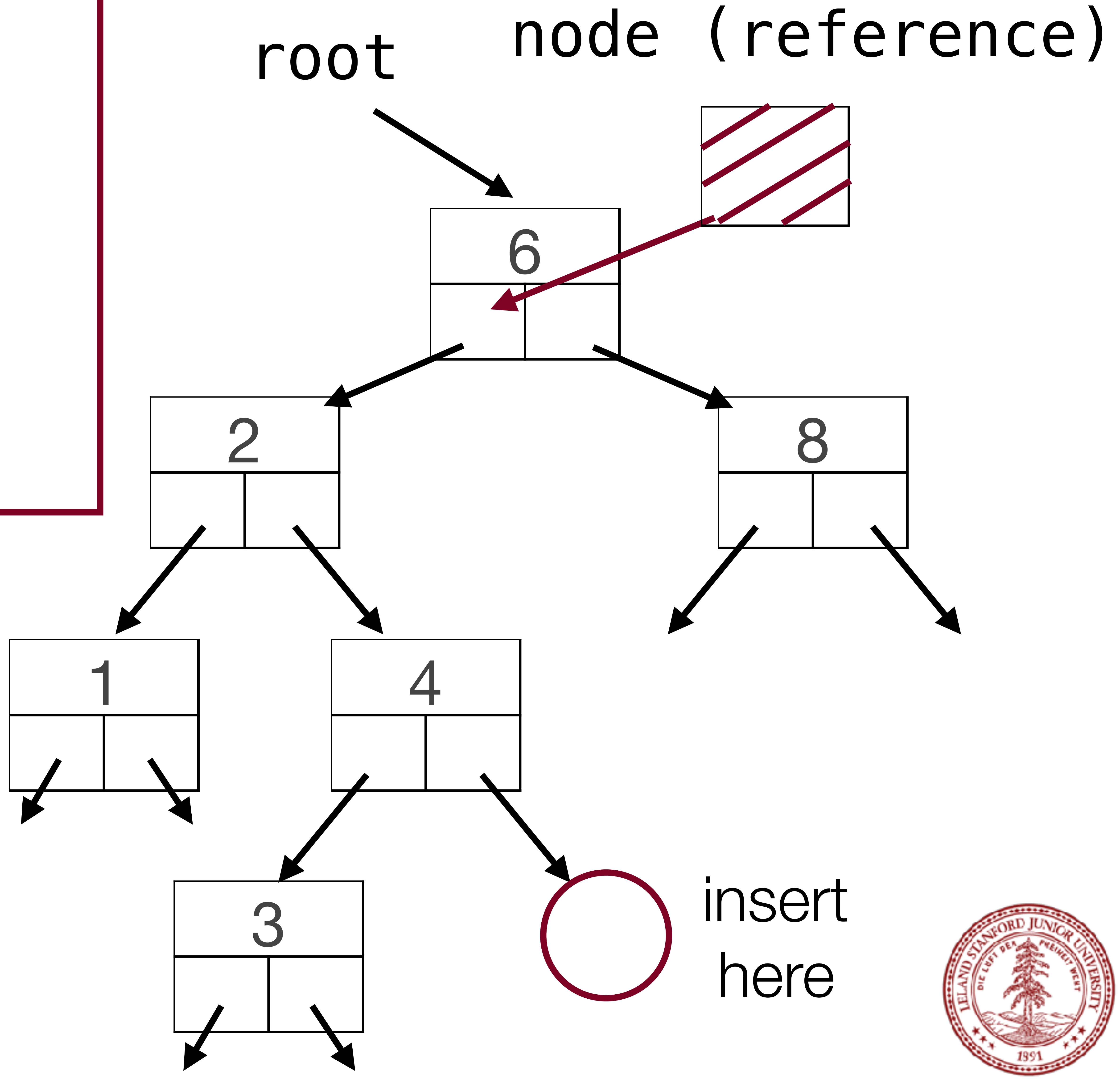
# Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```



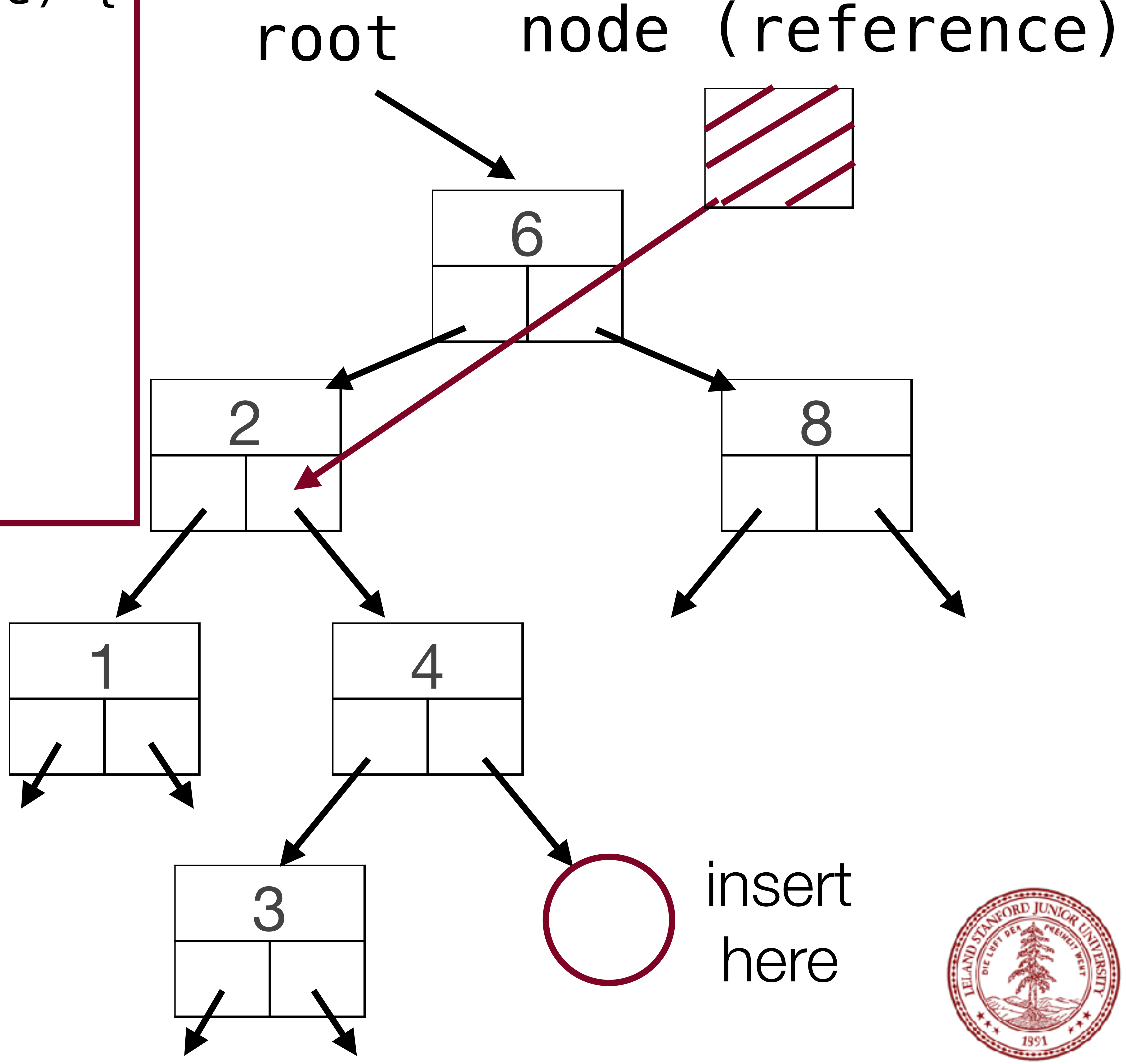
# Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```



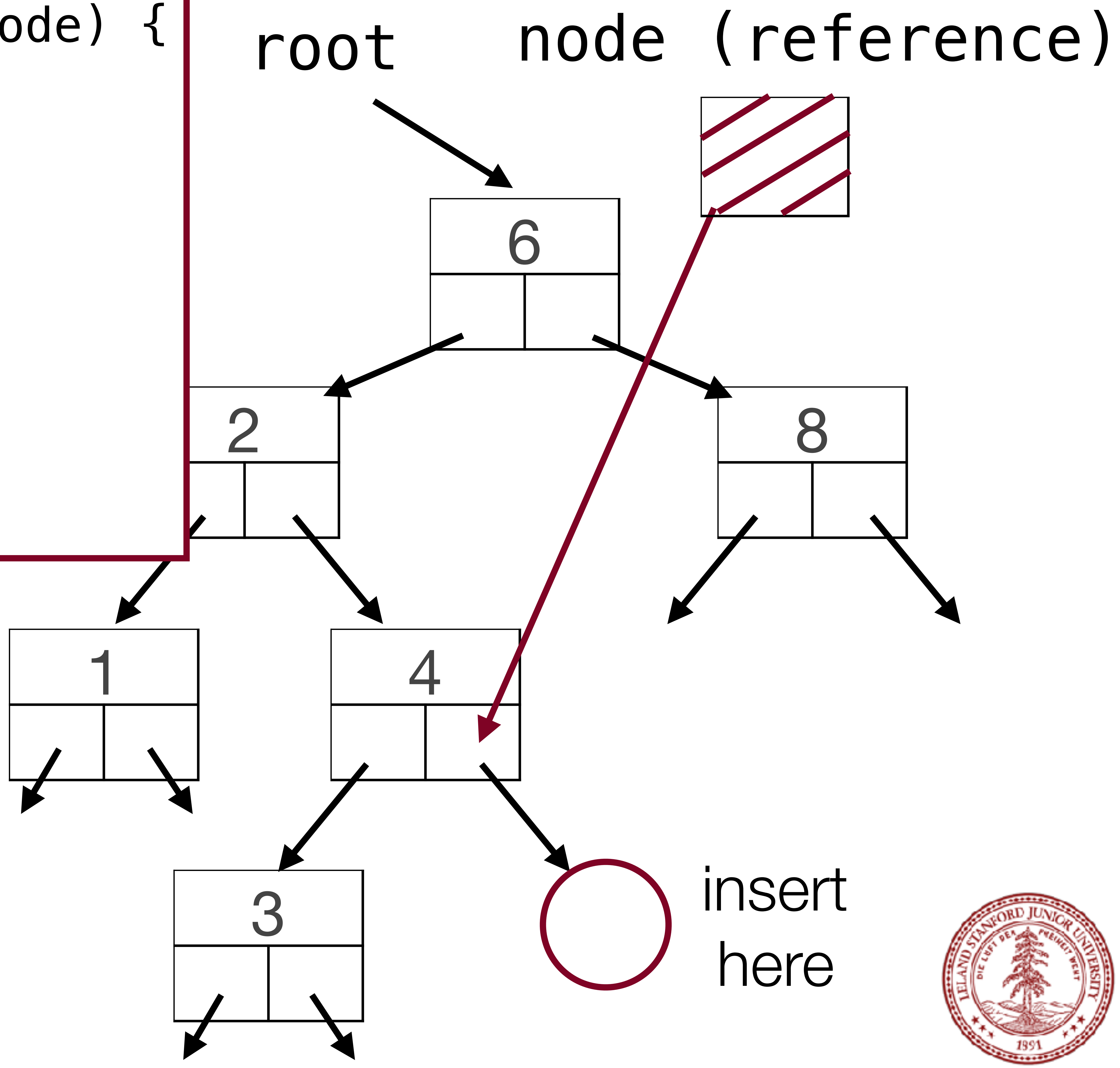
# Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```



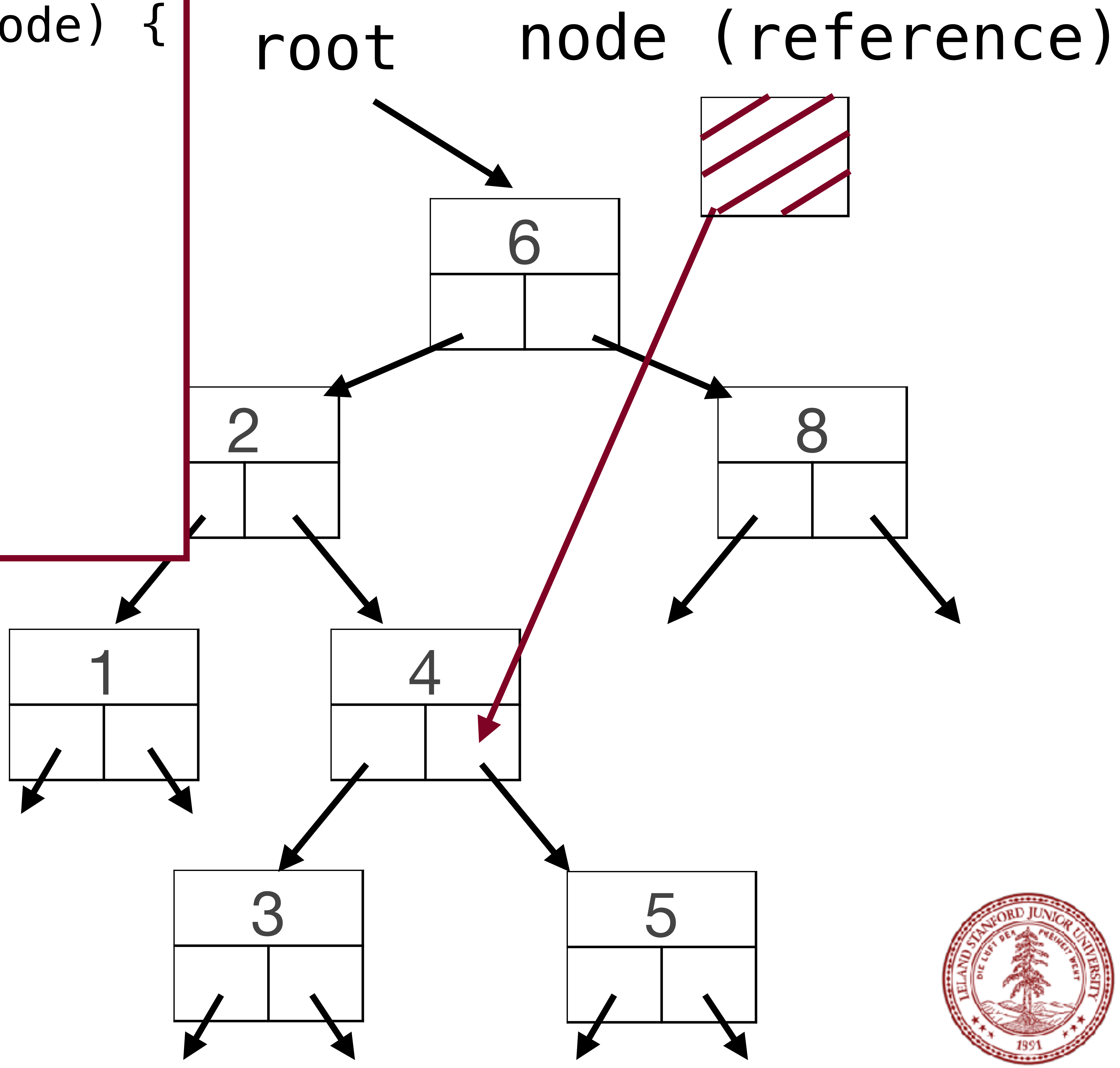
# Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```

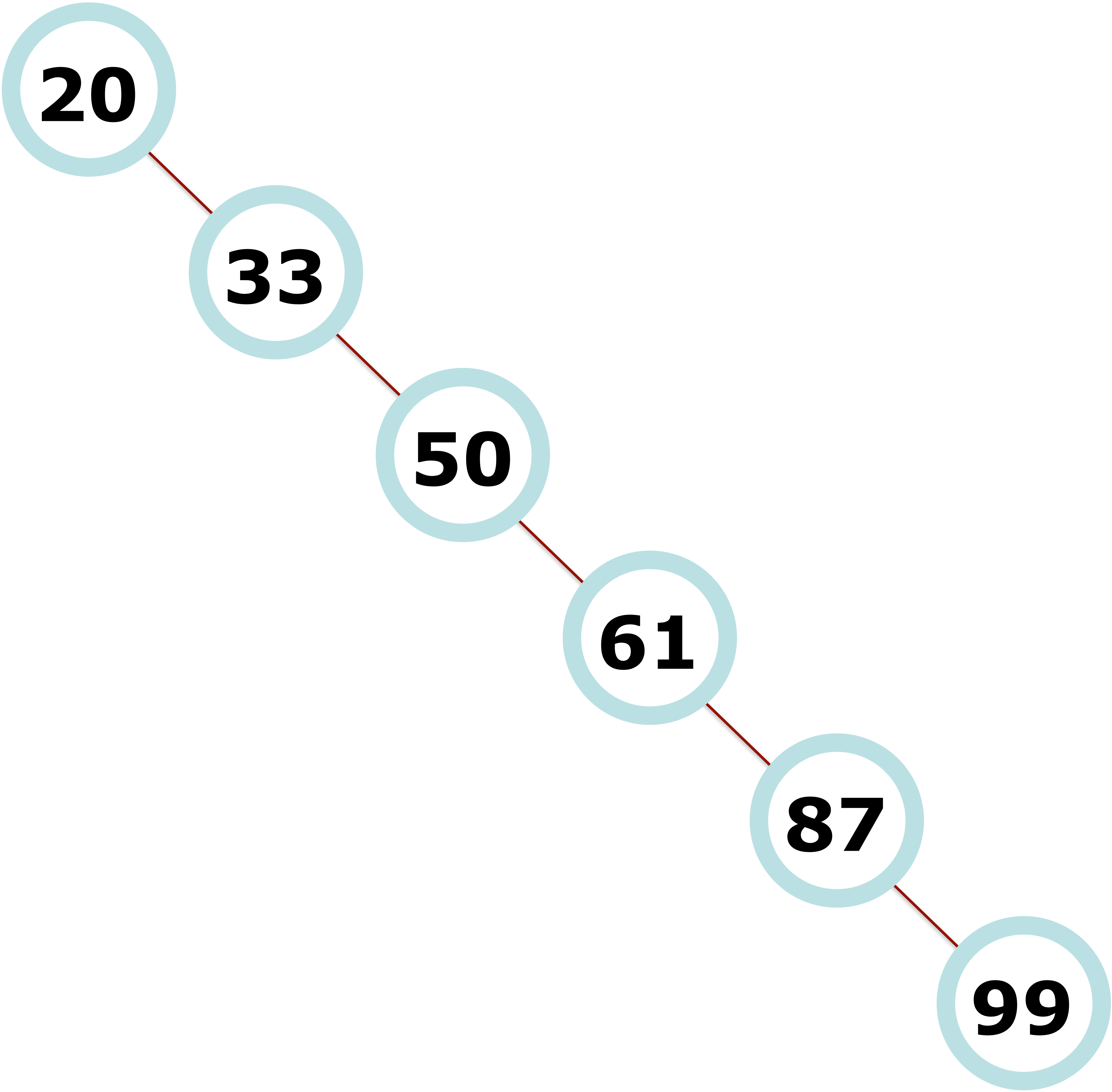


# Using References to Pointers

```
void StringSet::add(string s, Node* &node) {
    if (node == NULL) {
        node = new Node(s);
        count++;
    } else if (node->str > s) {
        add(s, node->left);
    } else if (node->str < s) {
        add(s, node->right);
    }
}
```



# Balancing Trees




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

What kind of tree to we get?

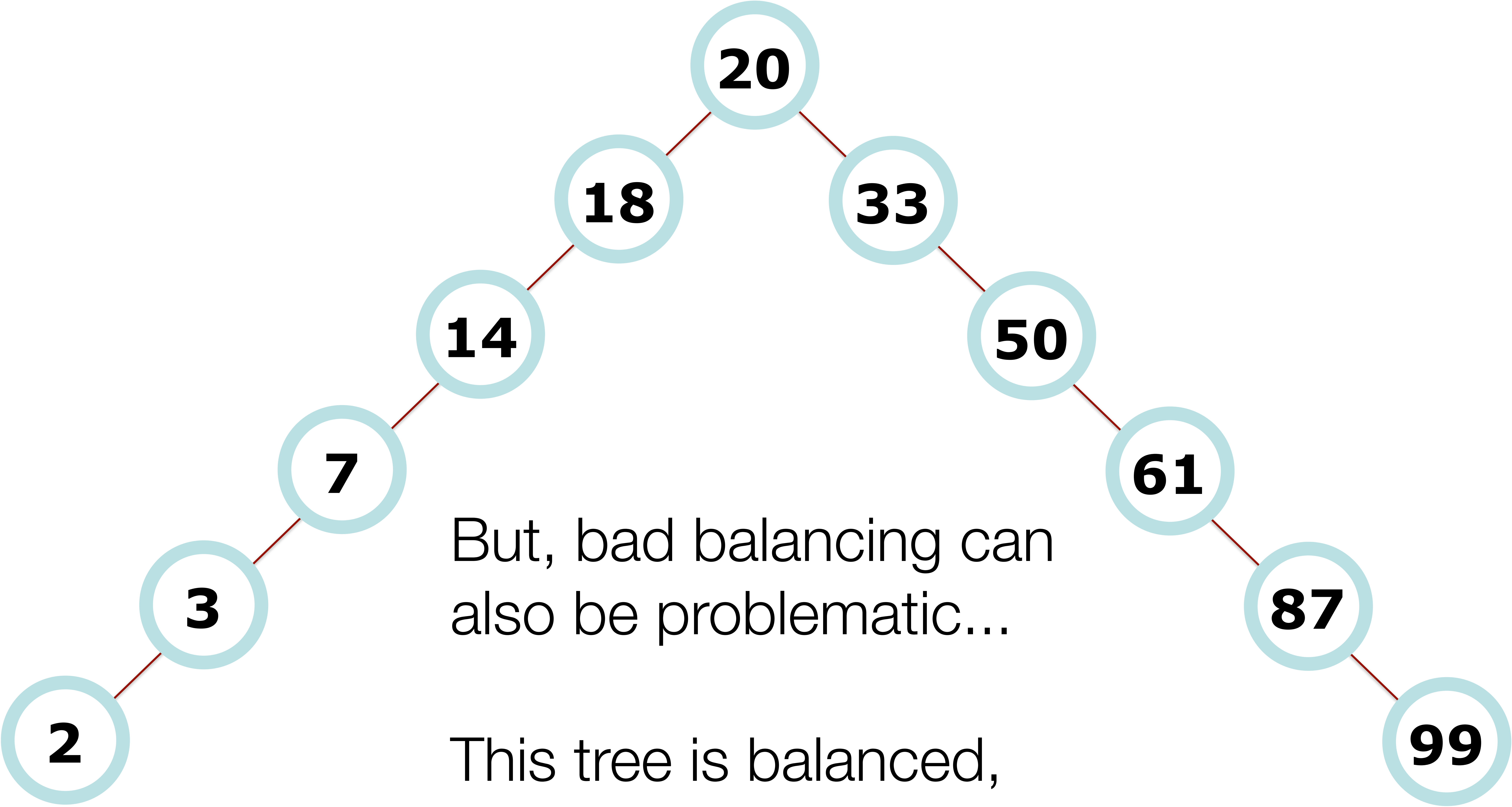
We get a Linked List Tree, and O(n) behavior :(

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

# Balancing Trees

Possible idea: require that the left and right subtrees in a BST have the same height.

# Balancing Trees

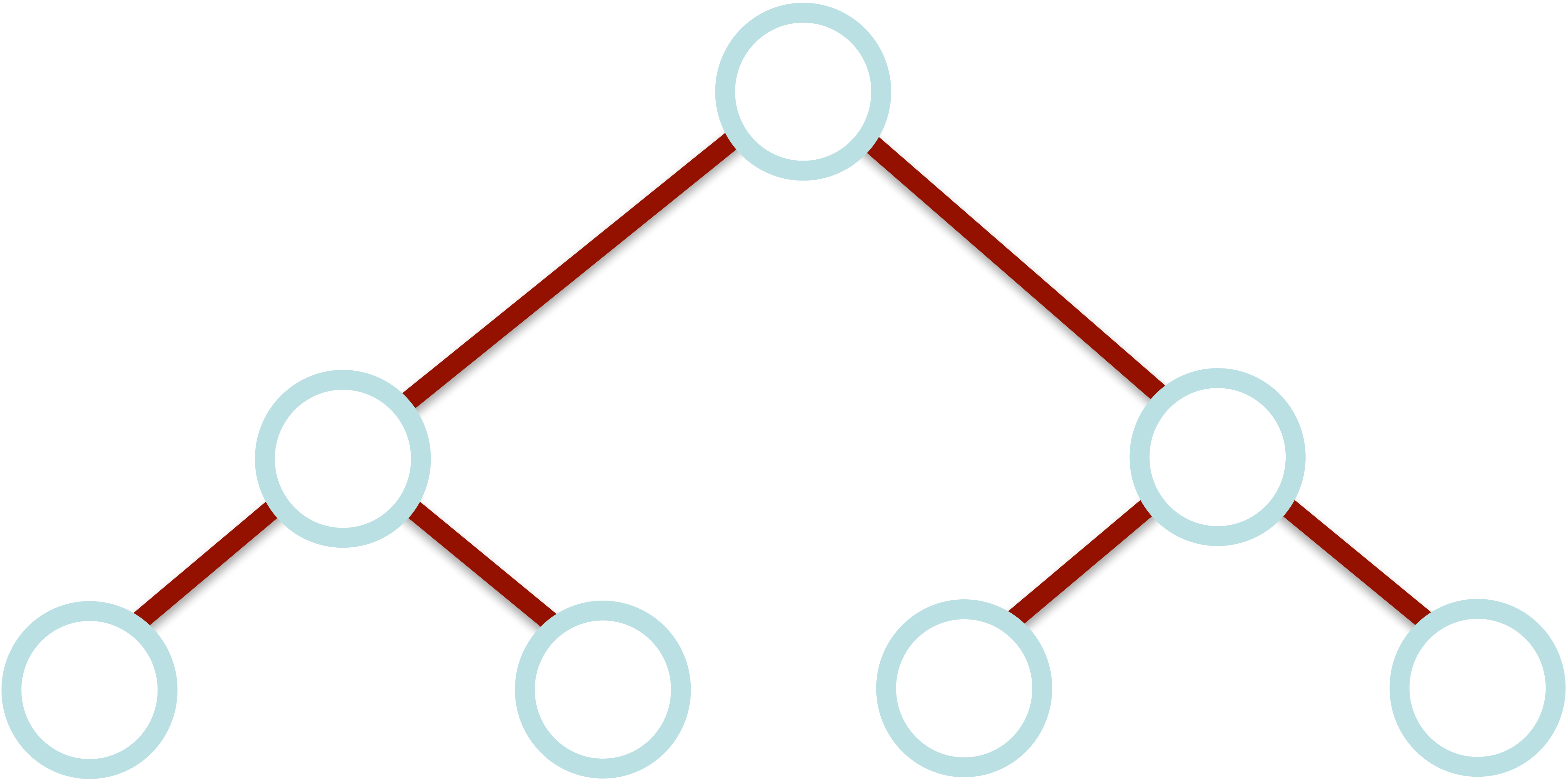



But, bad balancing can also be problematic...

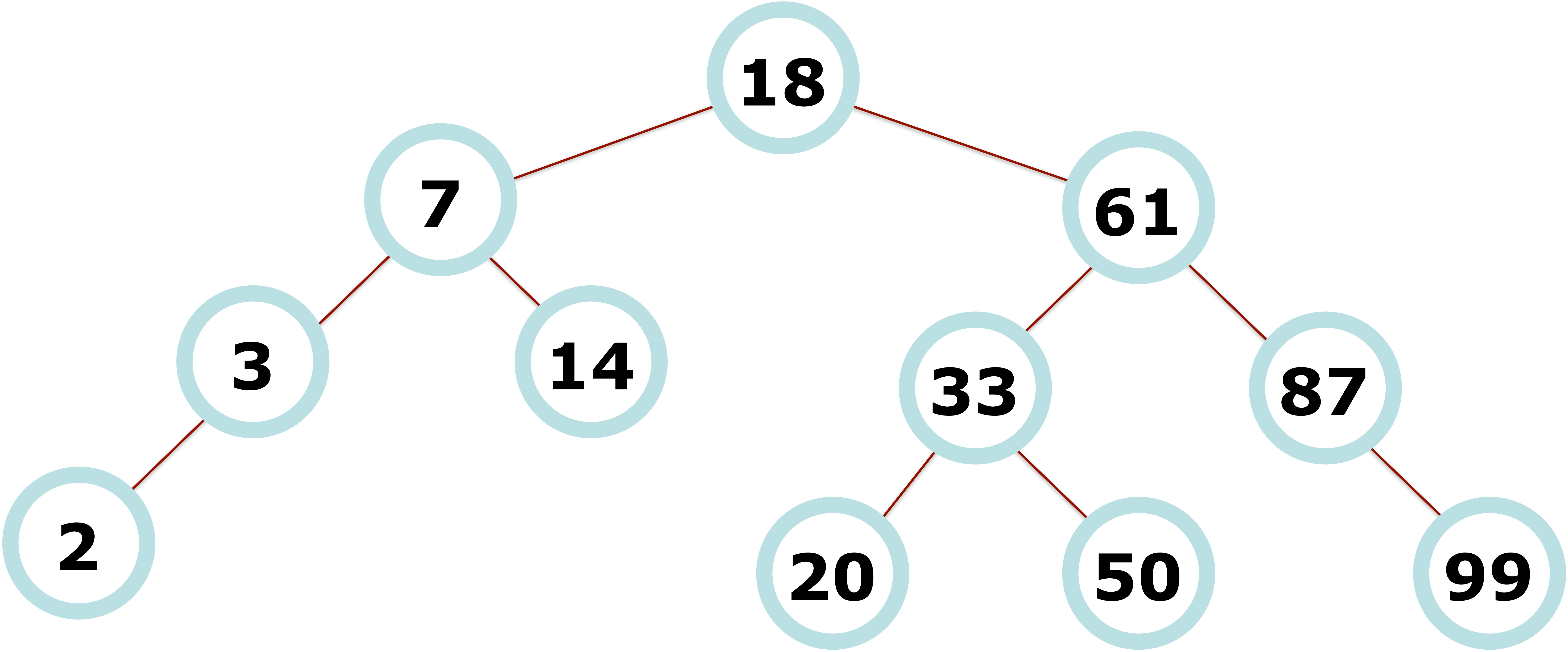
This tree is balanced, but only at the root.

# Balancing Trees: What we want

Another balance condition could be to insist that every node must have left and right subtrees of the same height: too rigid to be useful: only perfectly balanced trees with 2k-1 nodes would satisfy the condition (even with the guarantee of small depth).



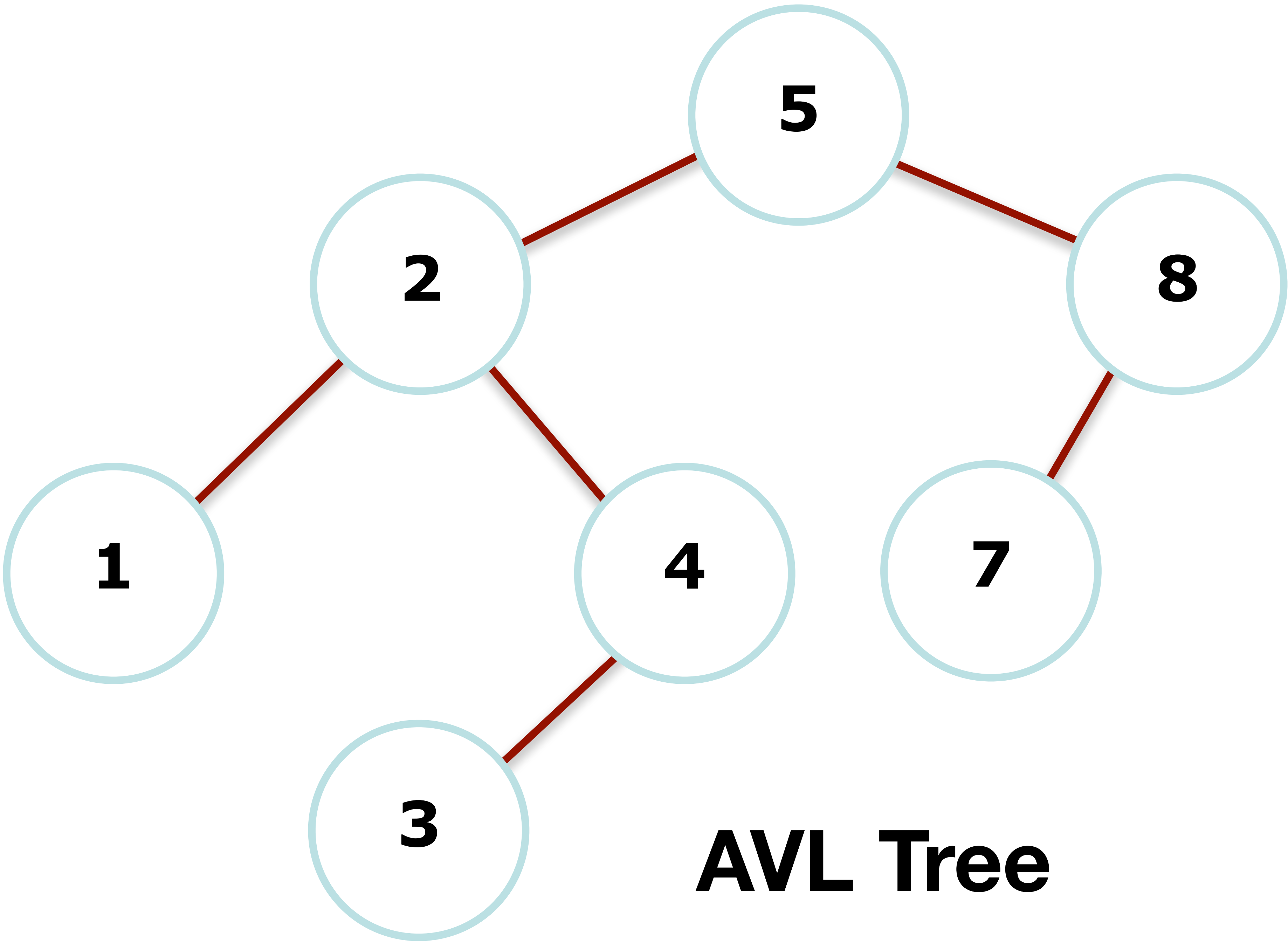
# Balancing Trees: What we want



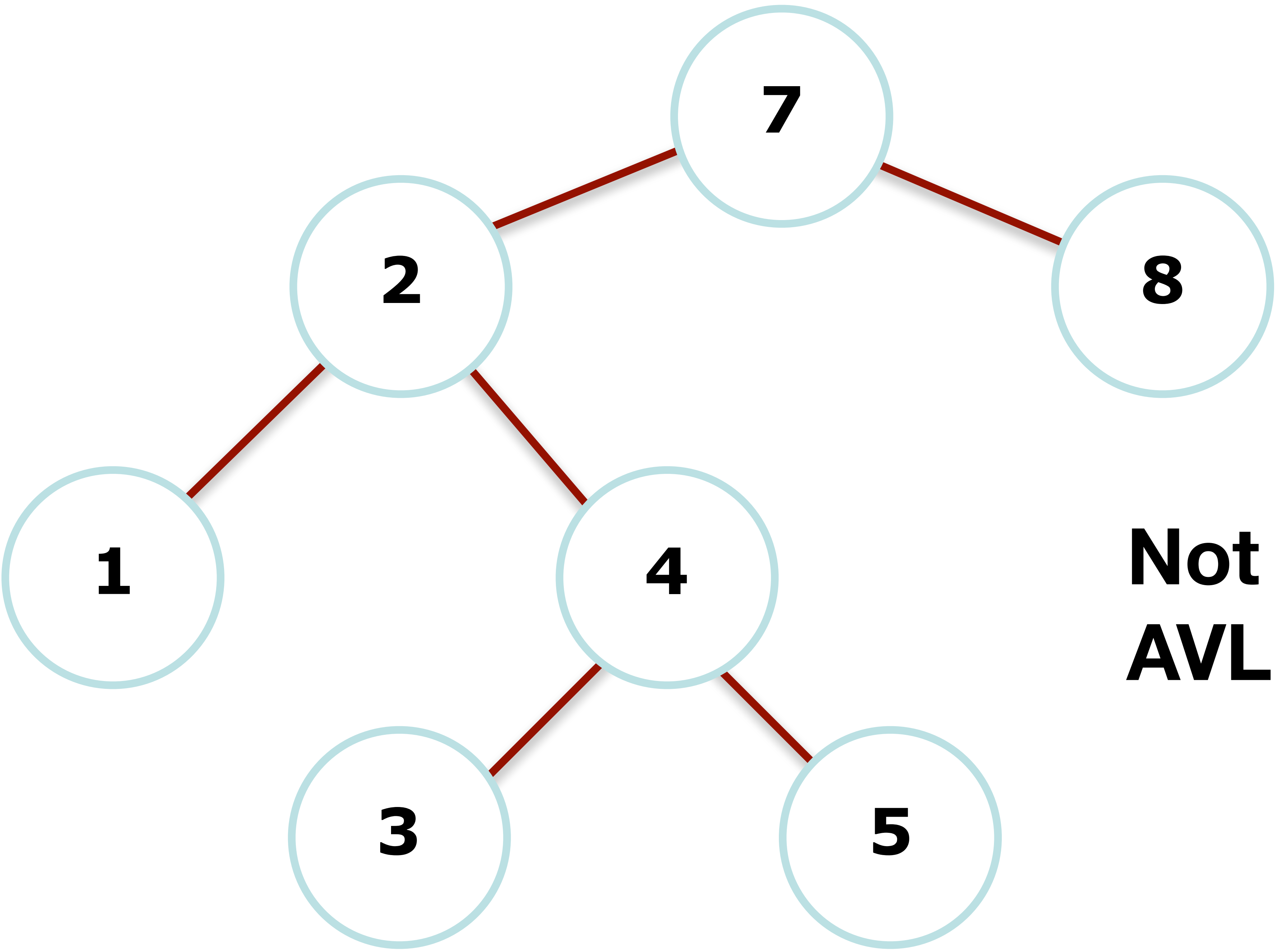

We are going to look at one balanced BST in particular, called an "AVL tree" You can play around with AVL trees here: https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

# AVL Trees

An AVL tree (Adelson-Velskii and Landis) is a compromise. It is the same as a binary search tree, except that for every node, the height of the left and right subtrees can differ only by 1 (and an empty tree has a height of -1).



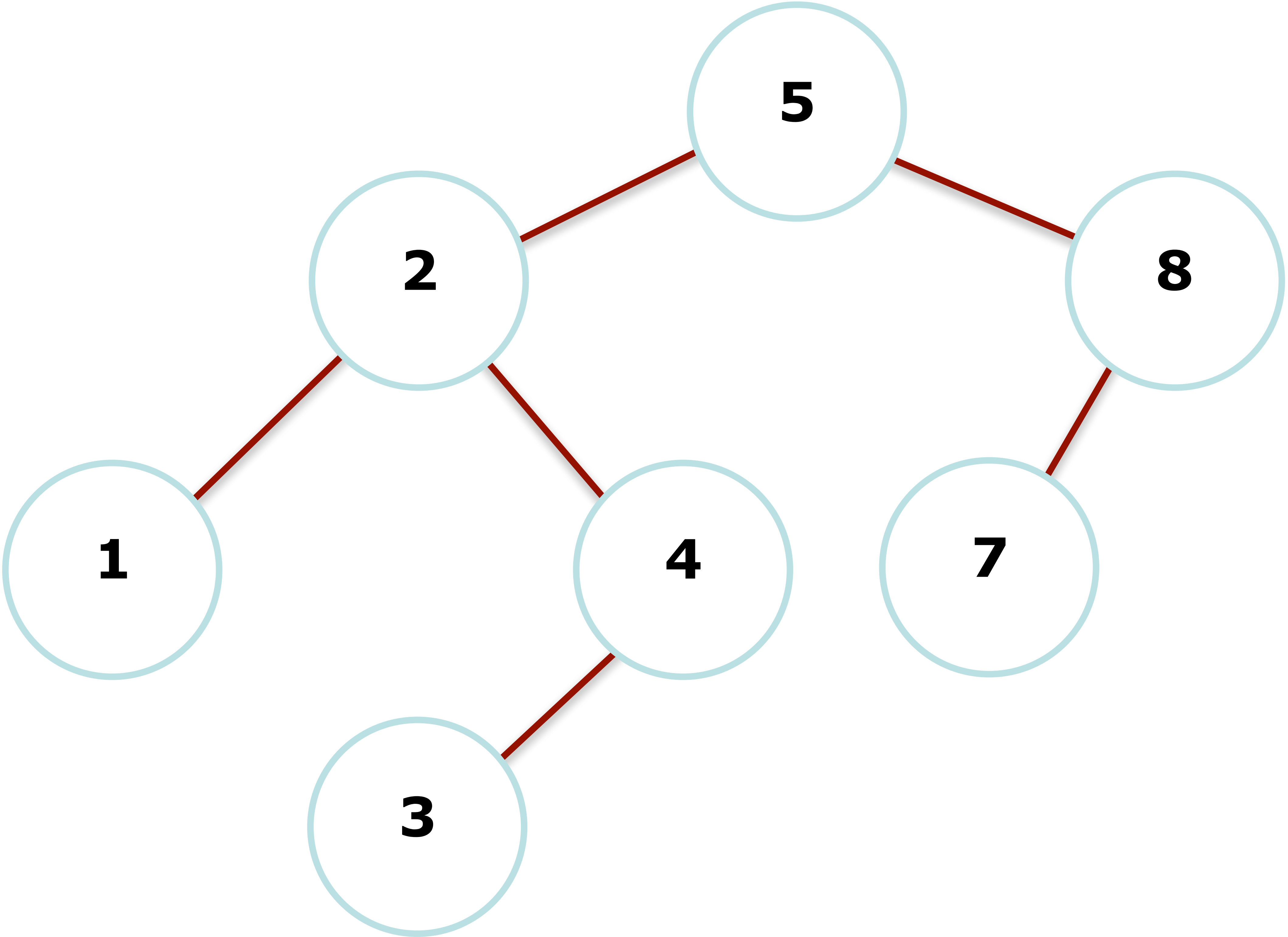

**AVL Tree**




**Not an AVL Tree**

# AVL Trees

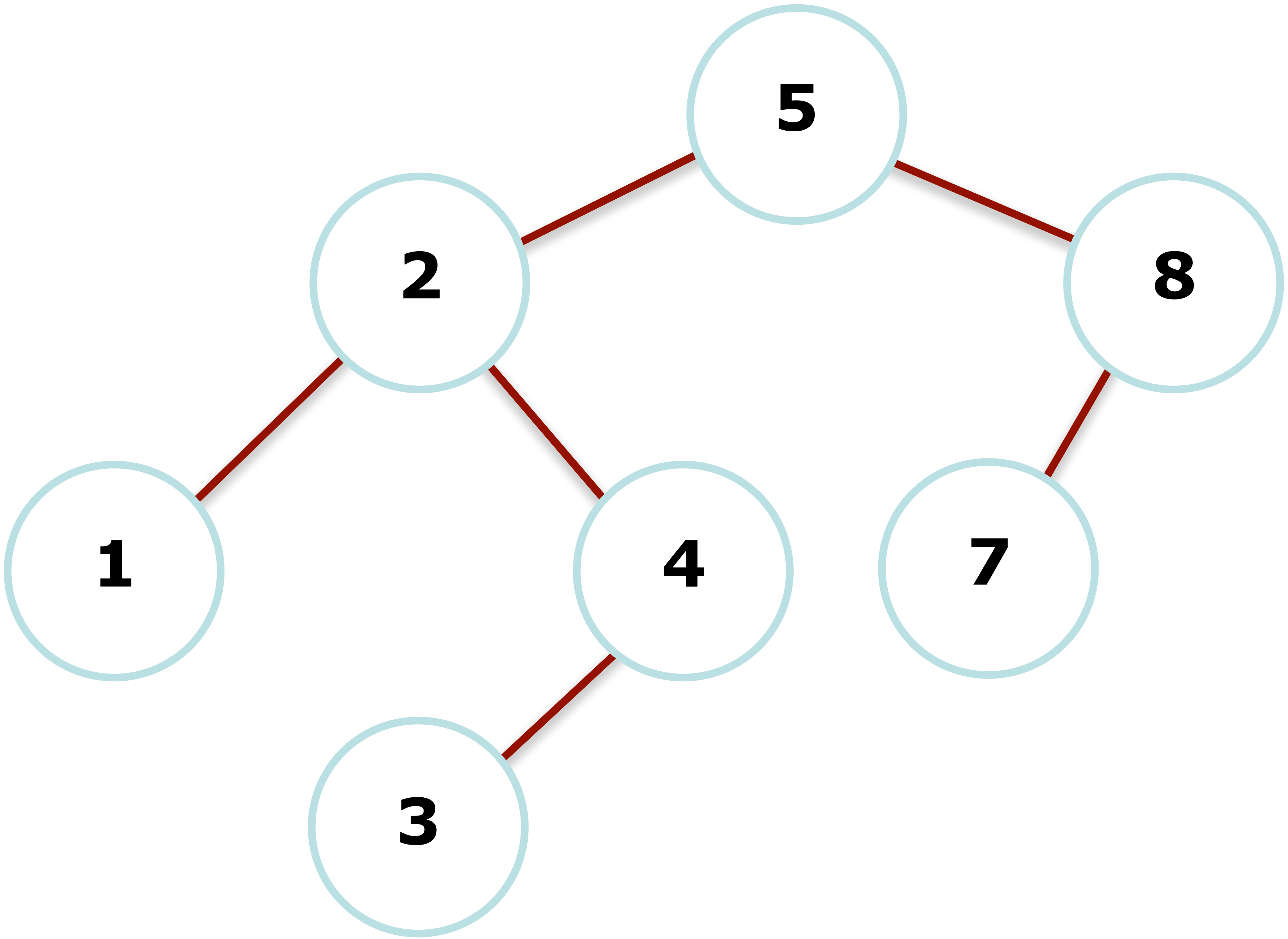
- Height information is kept for each node, and the height is *almost* log N in practice.




- When we insert into an AVL tree, we have to update the balancing information back up the tree
- We also have to maintain the AVL property — tricky! Think about inserting 6 into the tree: this would upset the balance at node 8.

# AVL Trees: Rotation

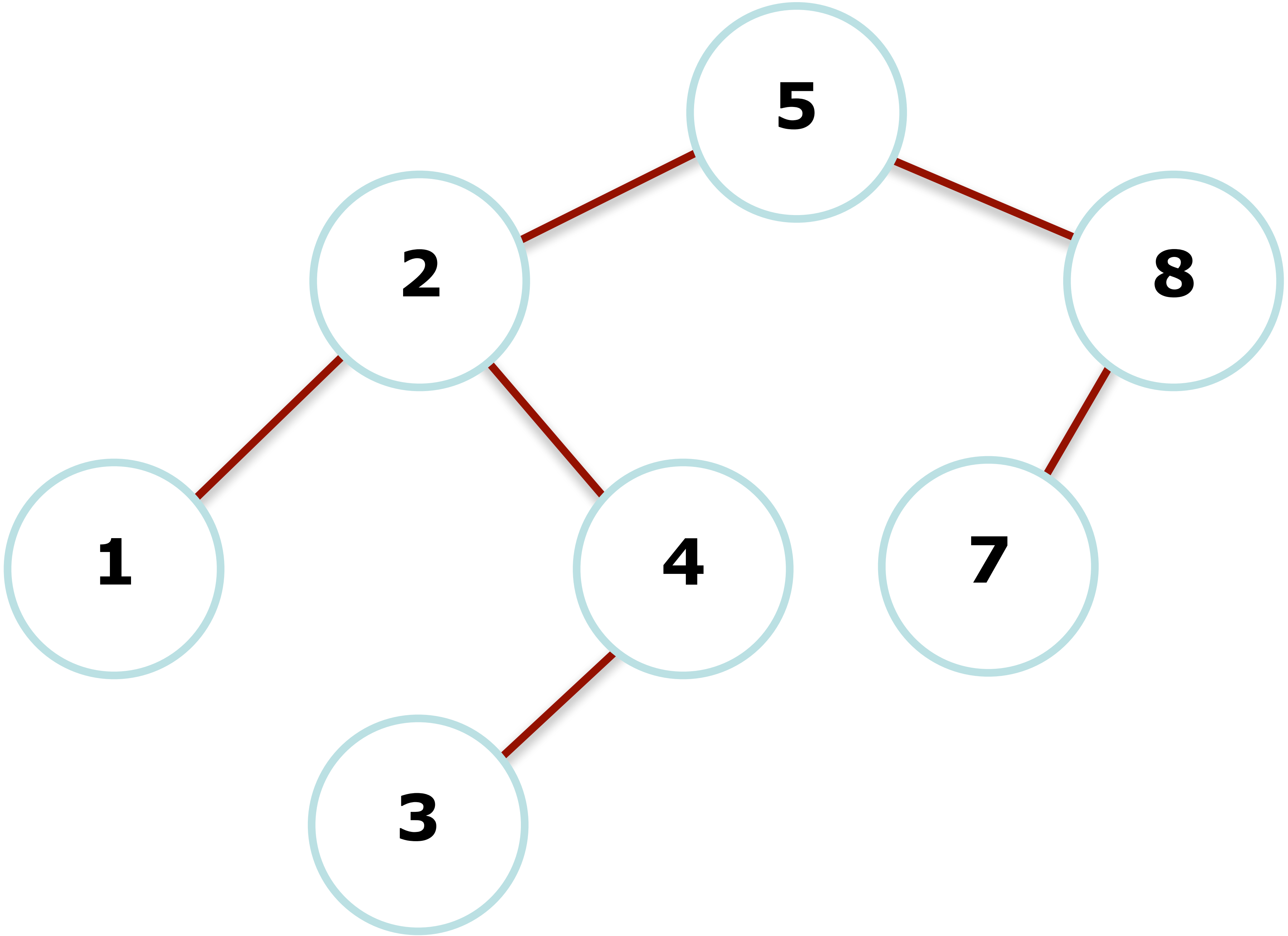
- As it turns out, a simple modification of the tree, called *rotation*, can restore the AVL property.




- After insertion, only nodes on the path from the insertion might have their balance altered, because only those nodes had their subtrees altered.
- We will re-balance as we follow the path up to the root updating balancing information.

# AVL Trees: Rotation

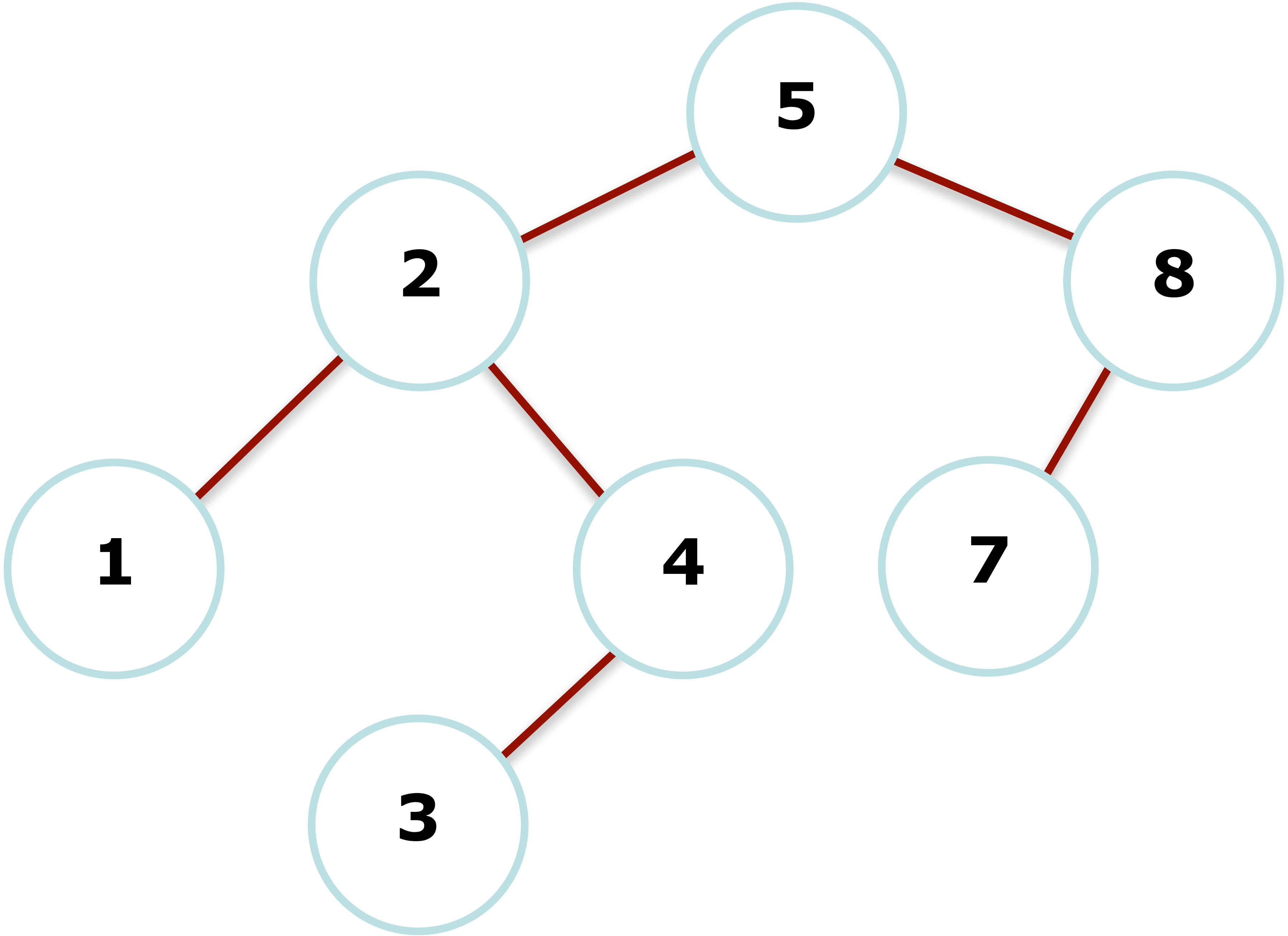
- We will call the node to be balanced, **α**




- Because any node has at most two children, and a height imbalance requires that **α**'s two subtrees' heights differ by two, there can be four violation cases:

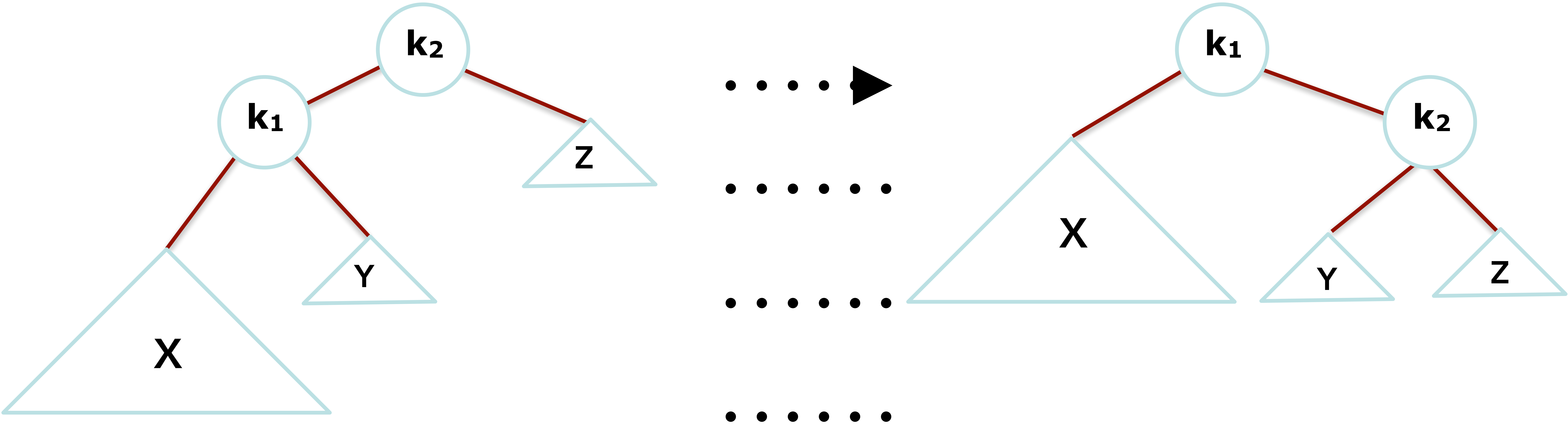
1. An insertion into the left subtree of the left child of **α**.
2. An insertion into the right subtree of the left child of **α**.
3. An insertion into the left subtree of the right child of **α**.
4. An insertion into the right subtree of the right child of **α**.

# AVL Trees: Rotation



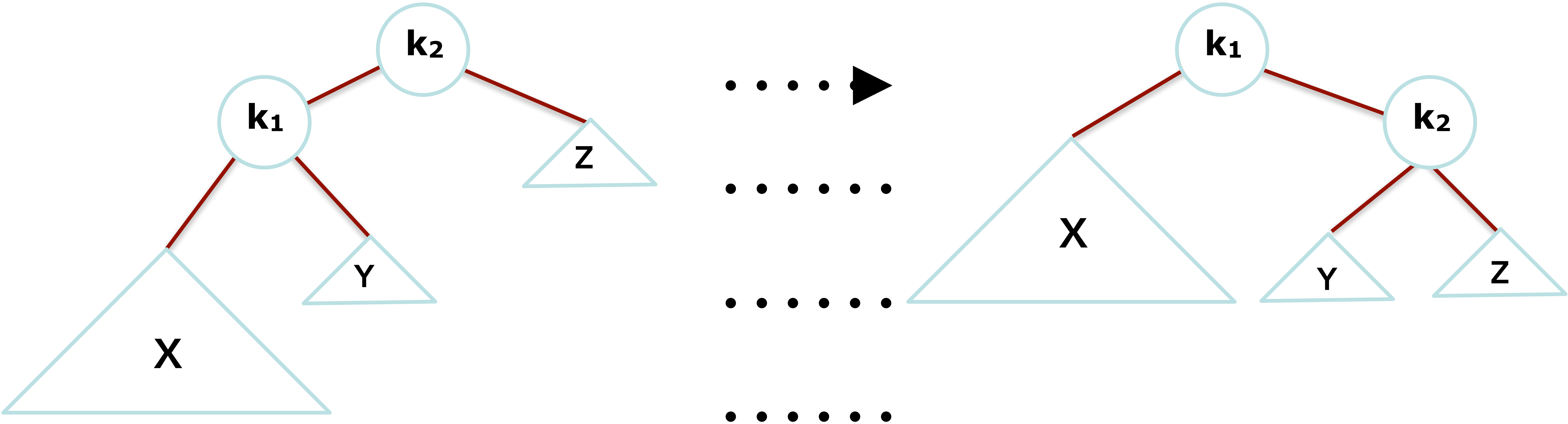

- For "outside" cases (left-left, right-right), we can do a "single rotation"
- For "inside" cases (left-right, right, left), we have to do a more complex "double rotation."

# AVL Trees: Single Rotation



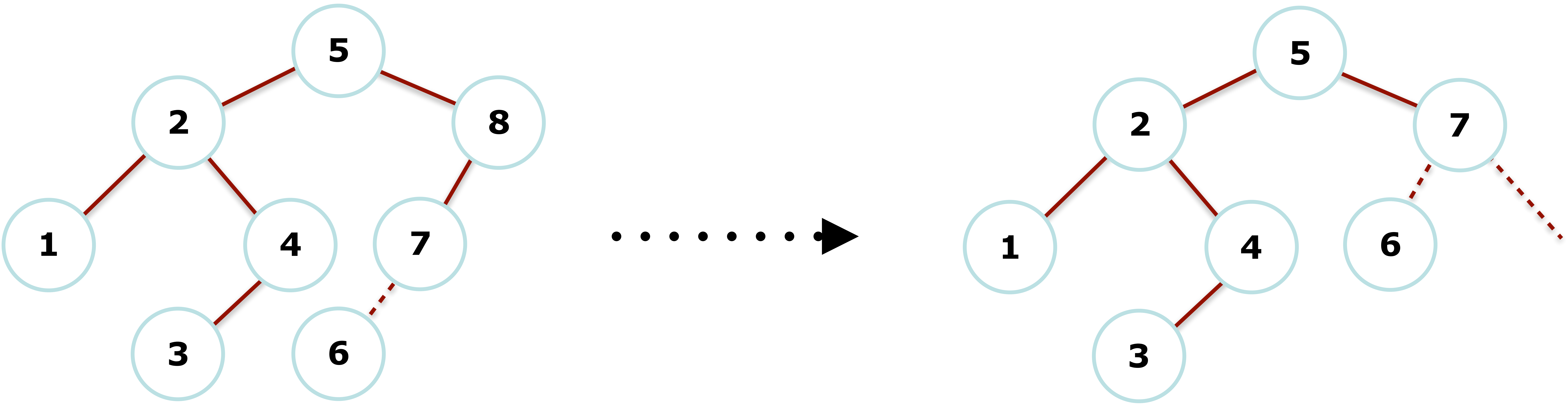

$k_2$ violates the AVL property, as X has grown to be 2 levels deeper than Z. Y cannot be at the same level as X because $k_2$ would have been out of balance before the insertion. We would like to move X up a level and Z down a level (fine, but not strictly necessary).

# AVL Trees: Single Rotation



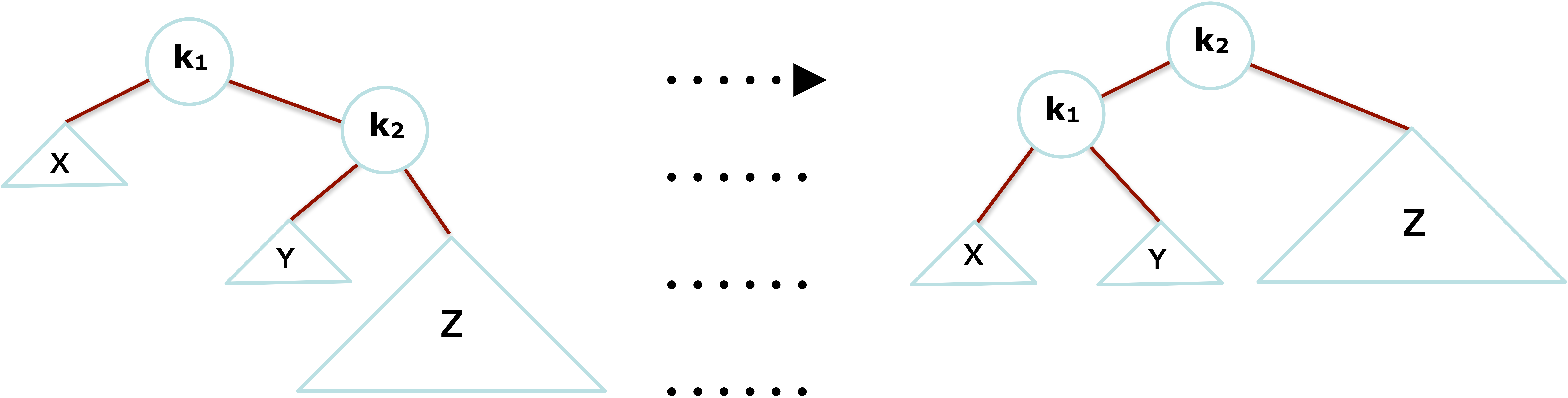

Visualization: Grab $k_1$ and shake, letting gravity take hold. $k_1$ is now the new root. In the original, $k_2 > k_1$, so $k_2$ becomes the right child of $k_1$. X and Z remain as the left and right children of $k_1$ and $k_2$, respectively. Y can be placed as $k_2$'s left child and satisfies all ordering requirements.

# AVL Trees: Single Rotation




Insertion of 6 breaks AVL property at 8 (not 5!), but is fixed with a single rotation (we “rotate 8 right” by grabbing 7 and hoisting it up)

# AVL Trees: Single Rotation




It is a symmetric case for the right-subtree of the right child. $k_1$ is unbalanced, so we “rotate k1 left” by hoisting k2)

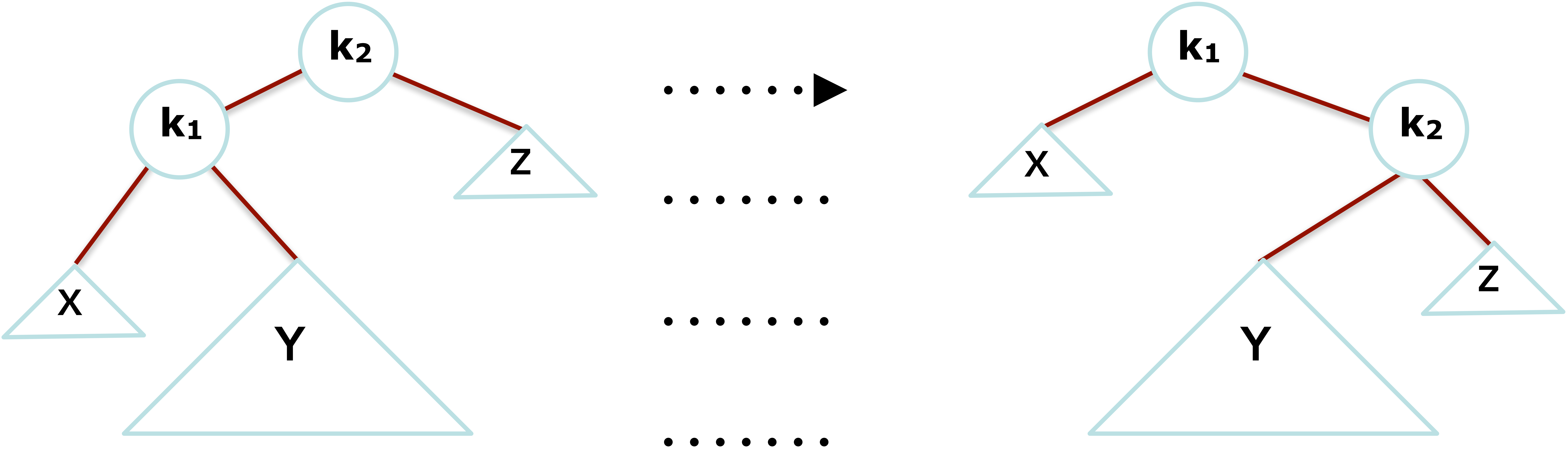
# AVL Trees: Single Rotation

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

**Insert 3, 2, 1, 4, 5, 6, 7**

# AVL Trees: Double Rotation

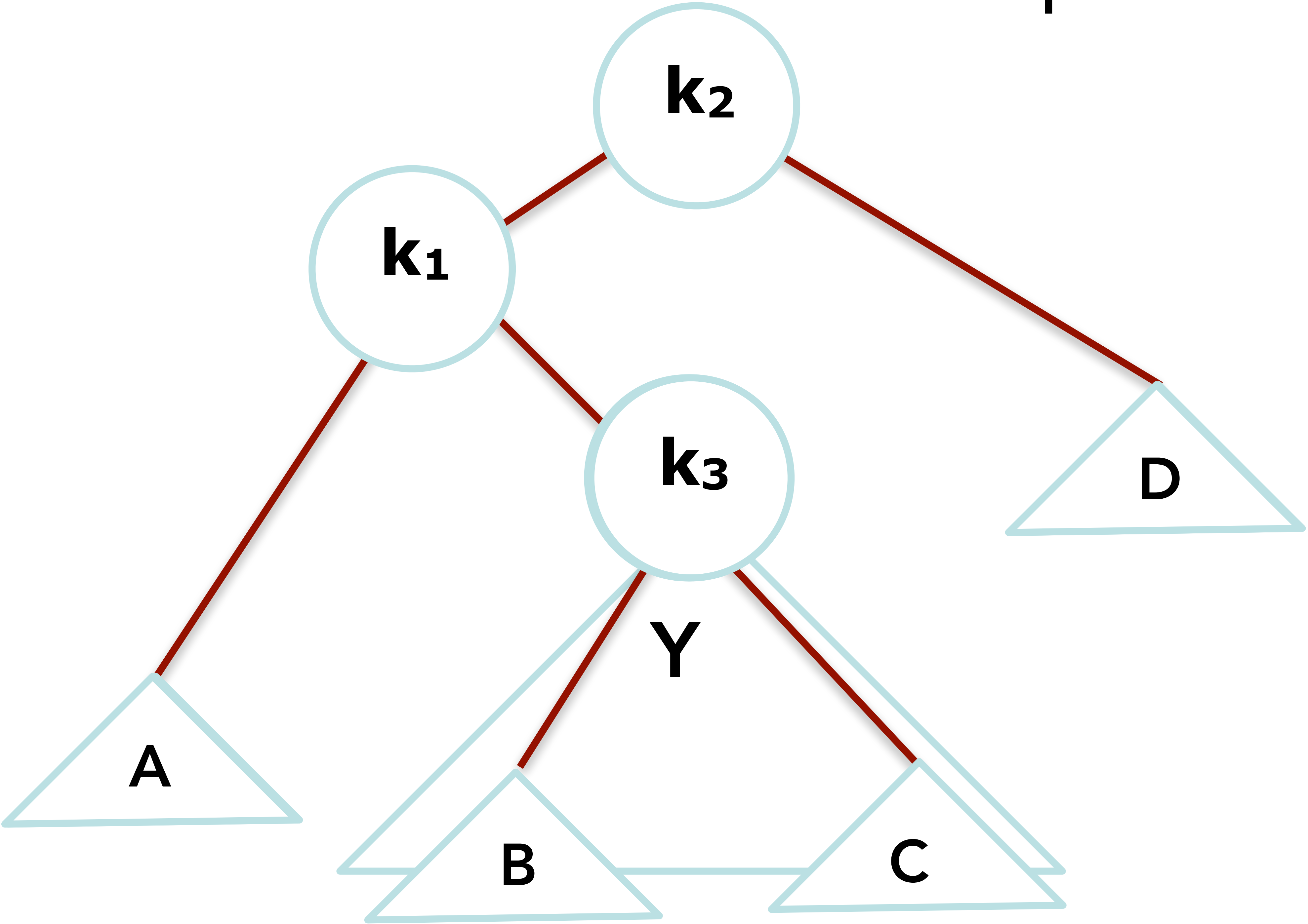
## AVL Trees: Single Rotation doesn't work for right/left, left/right!




Subtree Y is too deep (unbalanced at $k_2$), and the single rotation does not make it any less deep.

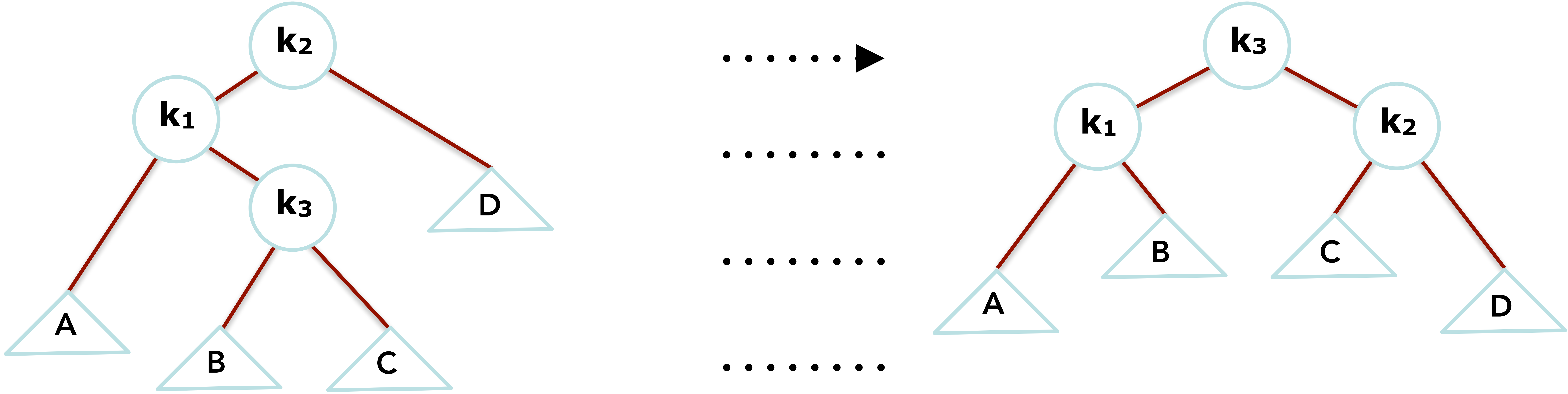
# AVL Trees: Double Rotation

AVL Trees: Double Rotation (can be thought of as one complex rotation or two simple single rotations)



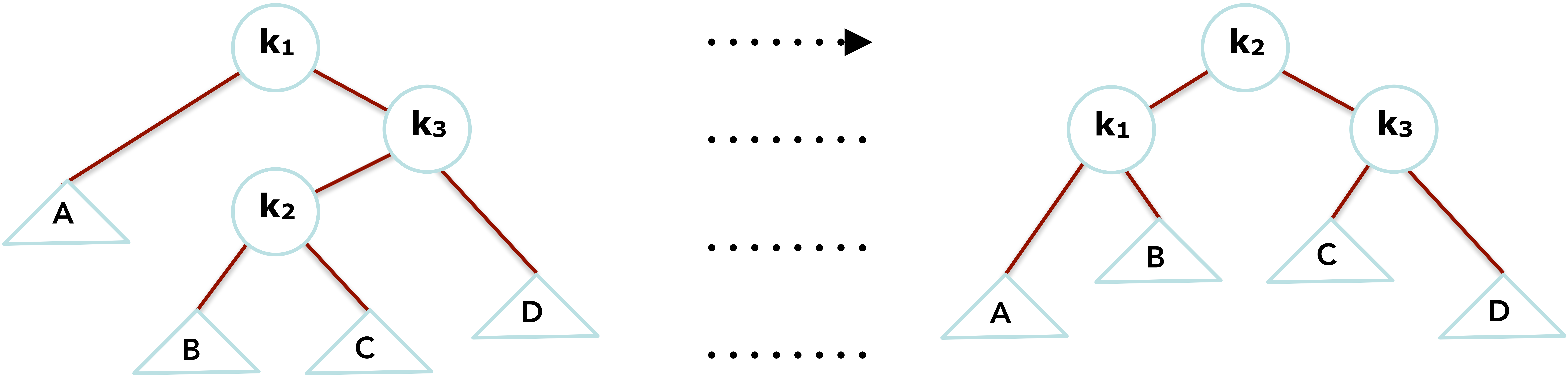

Instead of three subtrees, we can view the tree as four subtrees, connected by three nodes.

# AVL Trees: Double Rotation




We can't leave $k_2$ as root, nor can we make $k_1$ root (as shown before). So, $k_3$ must become the root.

# AVL Trees: Double Rotation




Double rotation also fixes an insertion into the left subtree of the right child ($k_1$ is unbalanced, so we first rotate $k_3$ right, then we rotate $k_1$ left)

# AVL Trees: Double Rotation

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

**Before: Insert 17, 12, 23, 9, 14, 19**

**Insert: 20**

# AVL Trees: Double Rotation

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

**Before: Insert 20, 10, 30, 5, 25, 40, 35, 45**

**Insert: 34**

# AVL Trees: Rotation Practice

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

**Before: Insert 30, 20, 10, 40, 50, 60, 70**

**Continuing: Insert 160, 150, 140, 130, 120, 110, 100, 80, 90**

# AVL Trees: How to Code

- Coding up AVL tree rotation is straightforward, but can be tricky.
- A recursive solution is easiest, but not too fast. However, clarity generally wins out in this case.
- To insert a new node into an AVL tree:
  1. Follow normal BST insertion.
  2. If the height of a subtree does not change, stop.
  3. If the height does change, do an appropriate single or double rotation, and update heights up the tree.
  4. One rotation will always suffice.

Example code can be found here: [http://www.sanfoundry.com/cpp-program-implement-avl-trees/](http://www.sanfoundry.com/cpp-program-implement-avl-trees/)

# Other Balanced Tree Data Structures

Other Balanced Tree Data Structures

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

# 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
  - https://www.cise.ufl.edu/~nemo/cop3530/AVL-Tree-Rotations.pdf

- **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

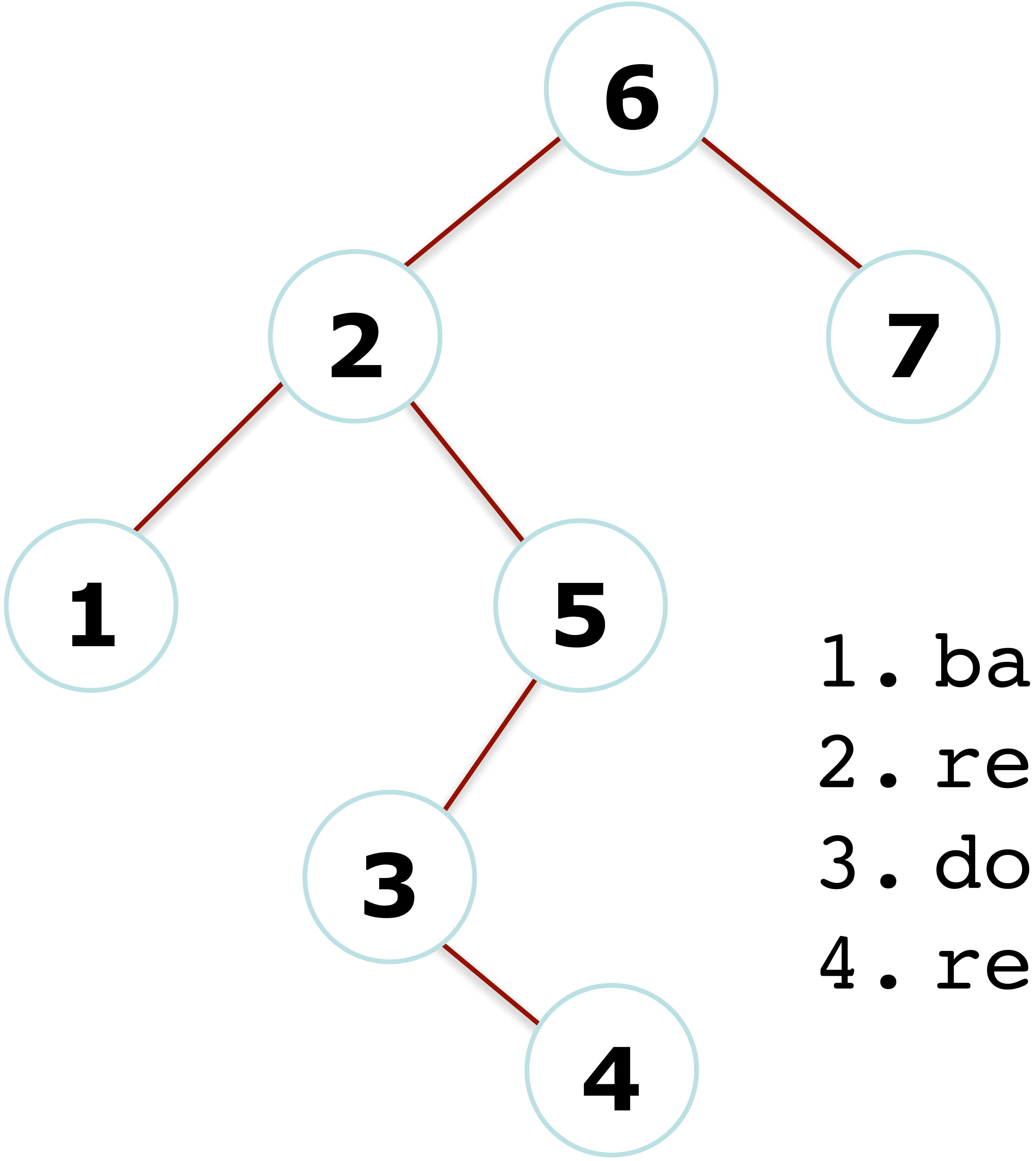
  - Red Black Trees:
  - https://www.cs.auckland.ac.nz/software/AlgAnim/red_black.html

  - 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

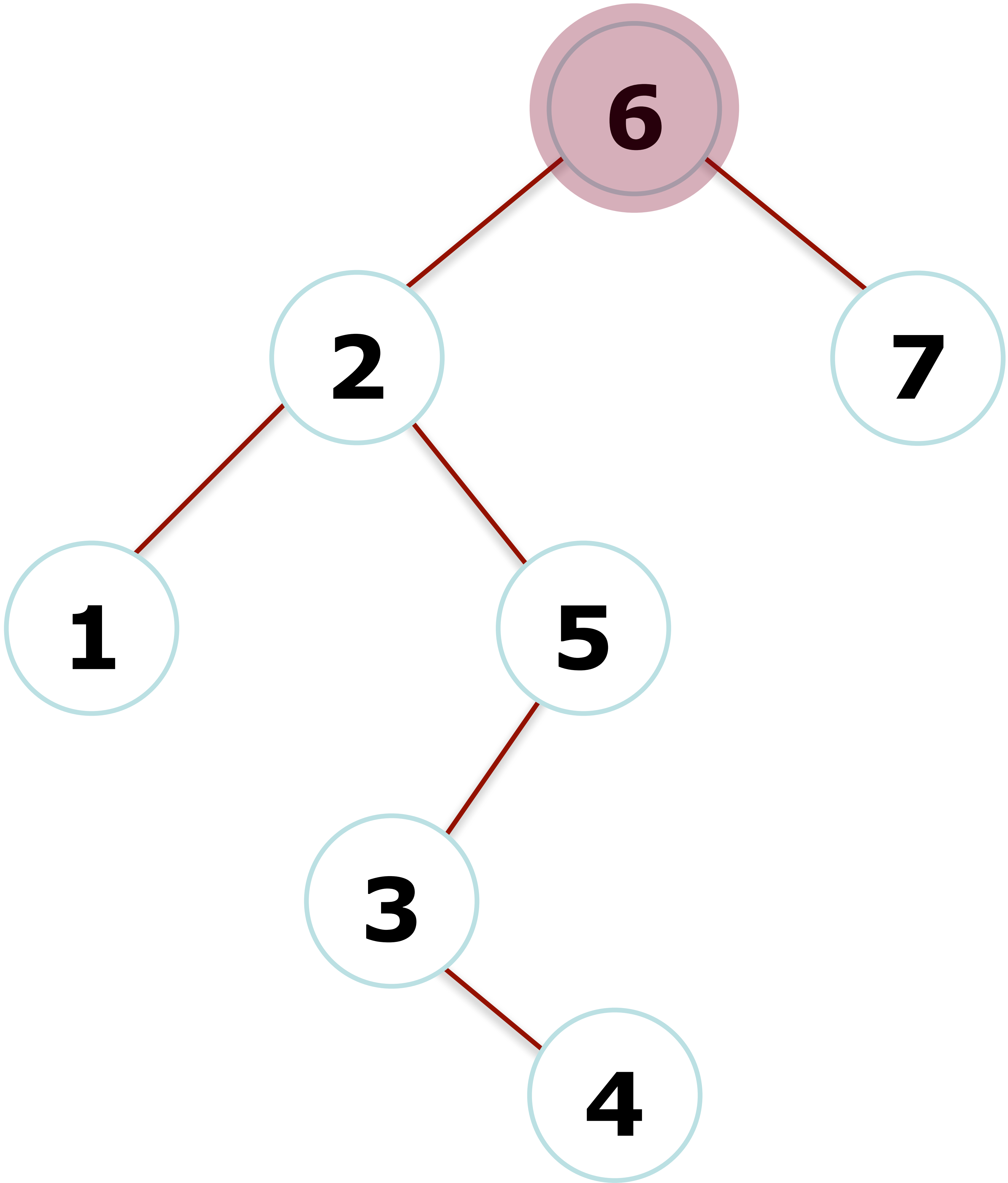
# In-Order Traversal: It is called "in-order" for a reason!




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

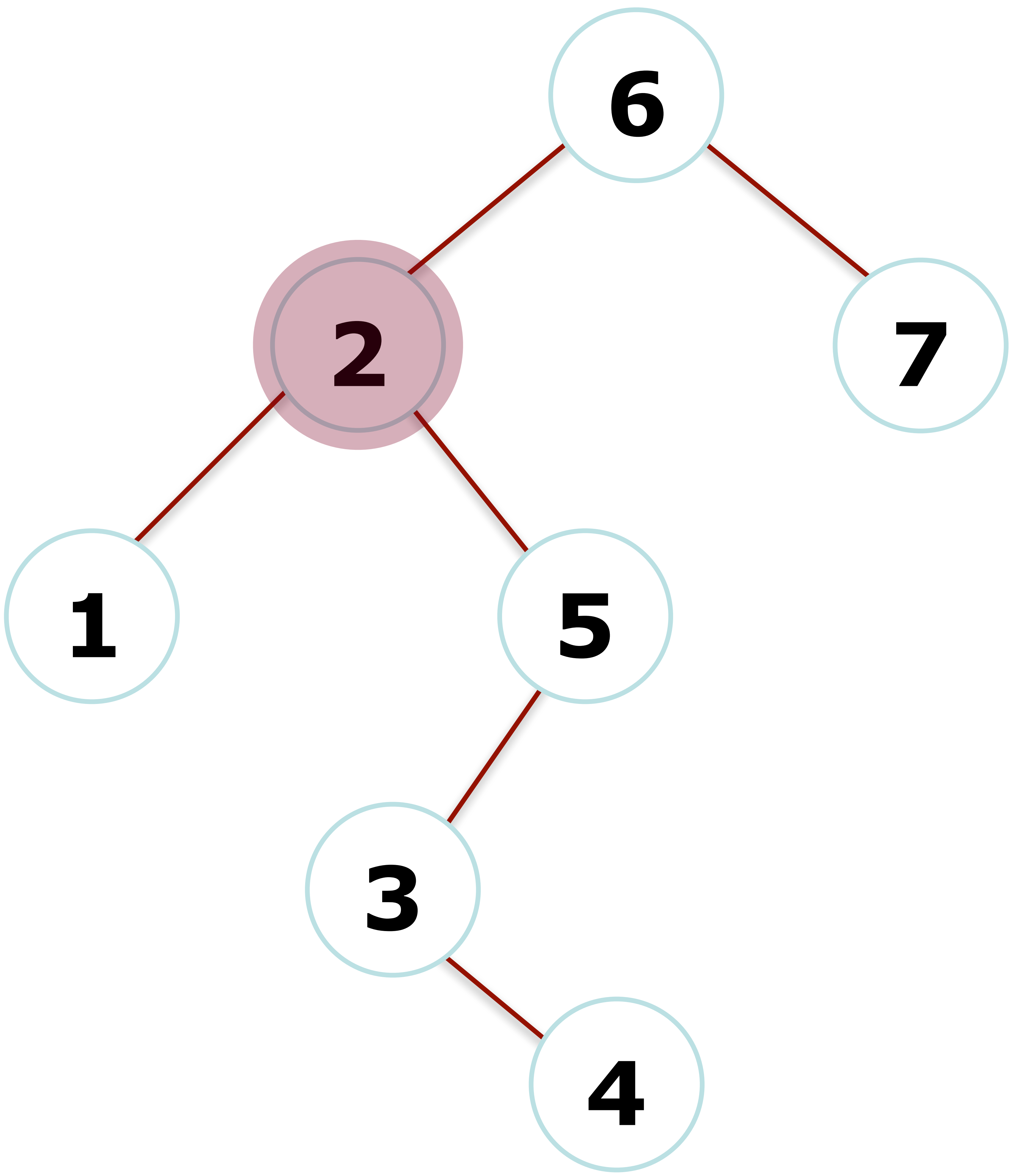



Current Node: 6

1. current not NULL
2. recurse left

Output:

# In-Order Traversal Example: printing

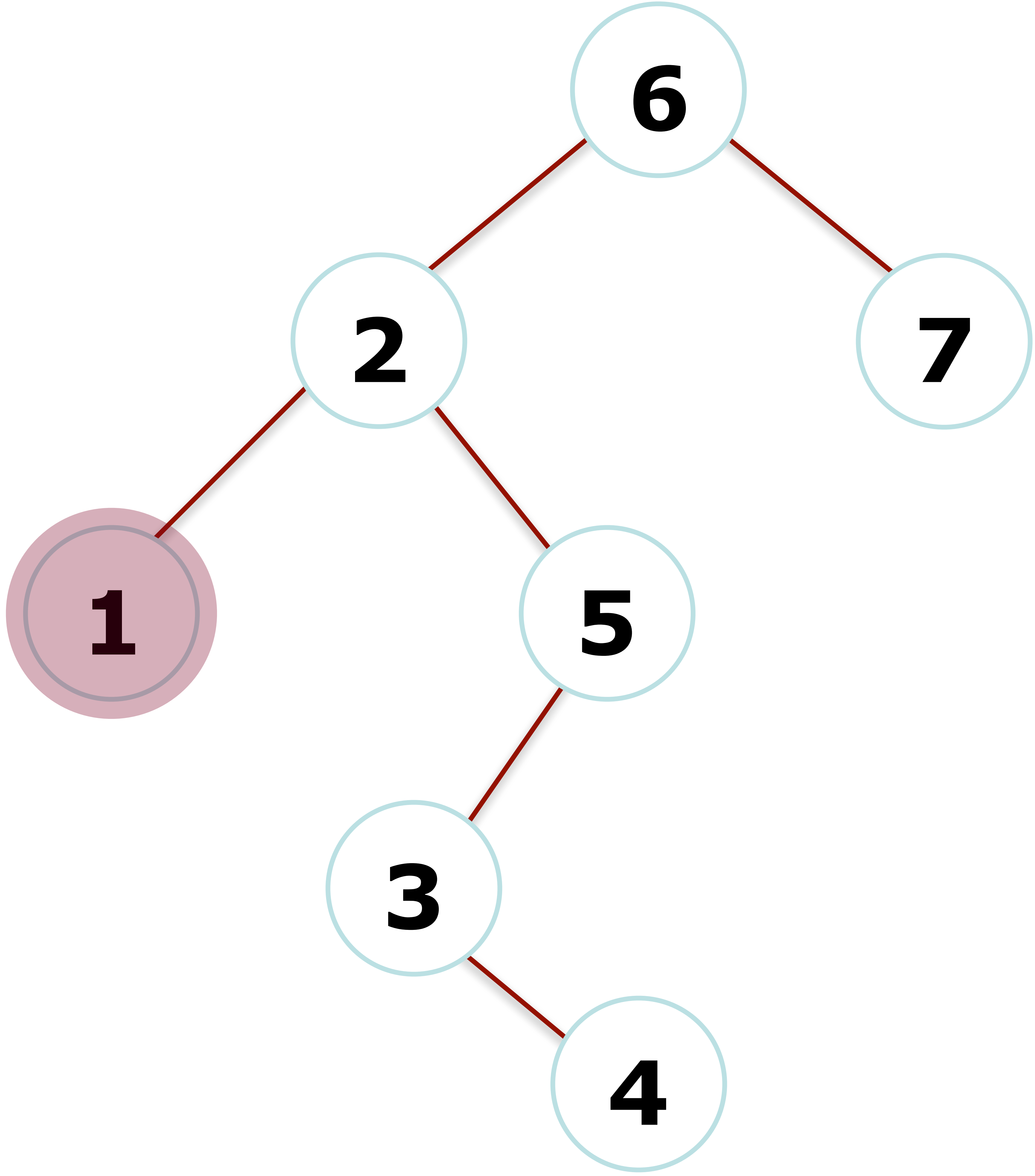



Current Node: 2

1. current not NULL
2. recurse left

Output:

# In-Order Traversal Example: printing

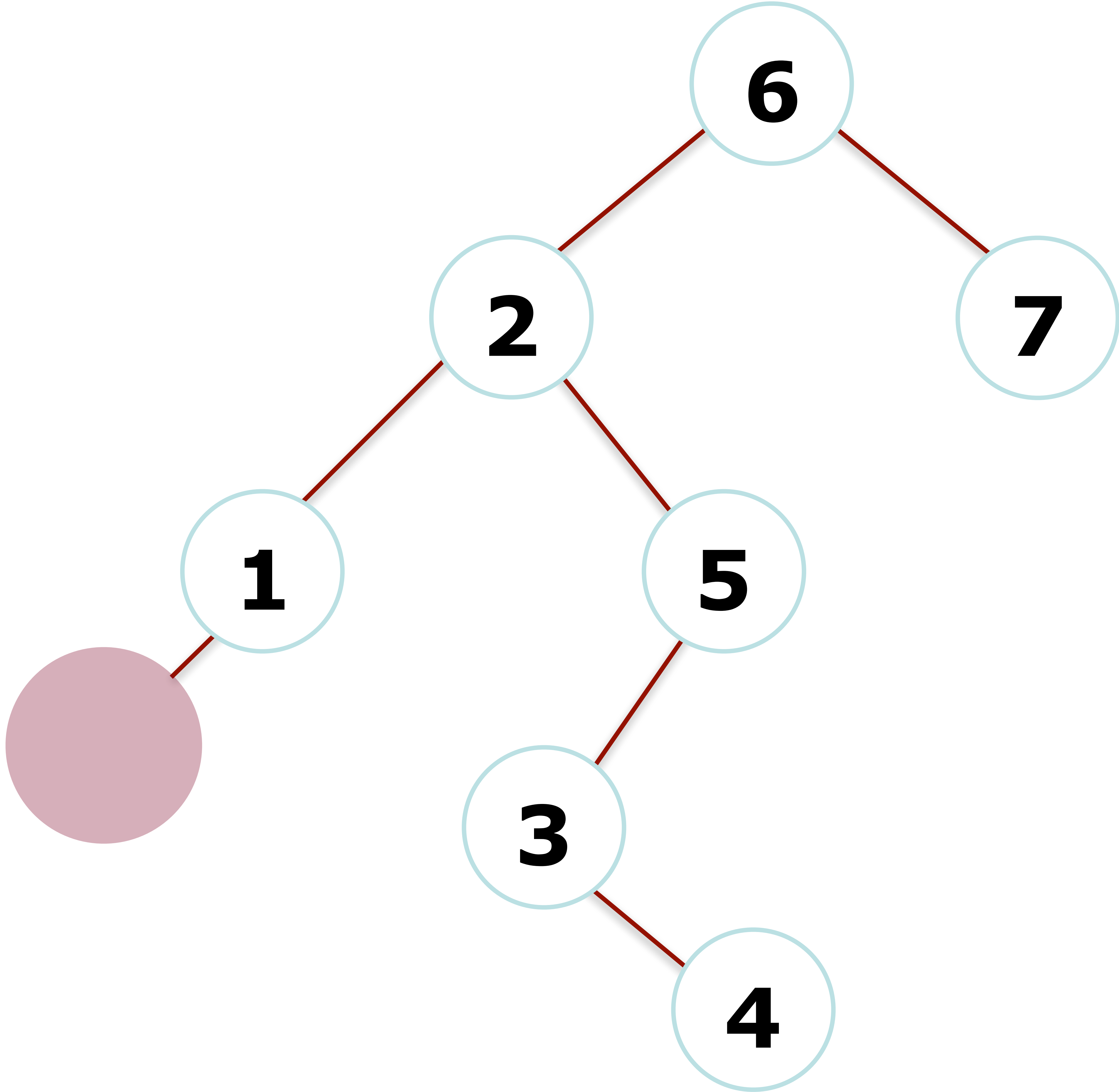



Current Node: 1

1. current not NULL
2. recurse left

Output:

# In-Order Traversal Example: printing

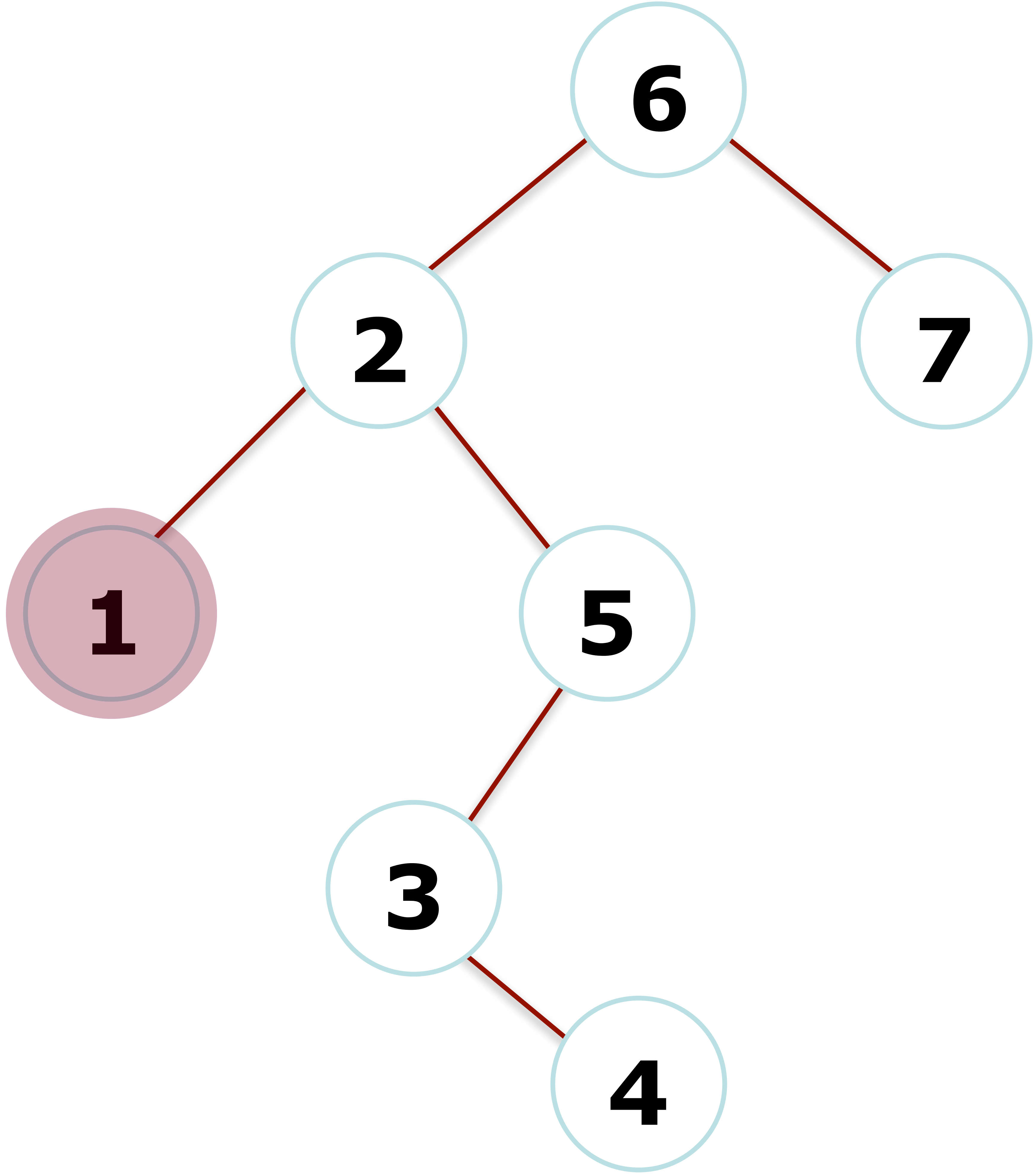



Current Node: NULL

1. current NULL: return

Output:

# In-Order Traversal Example: printing

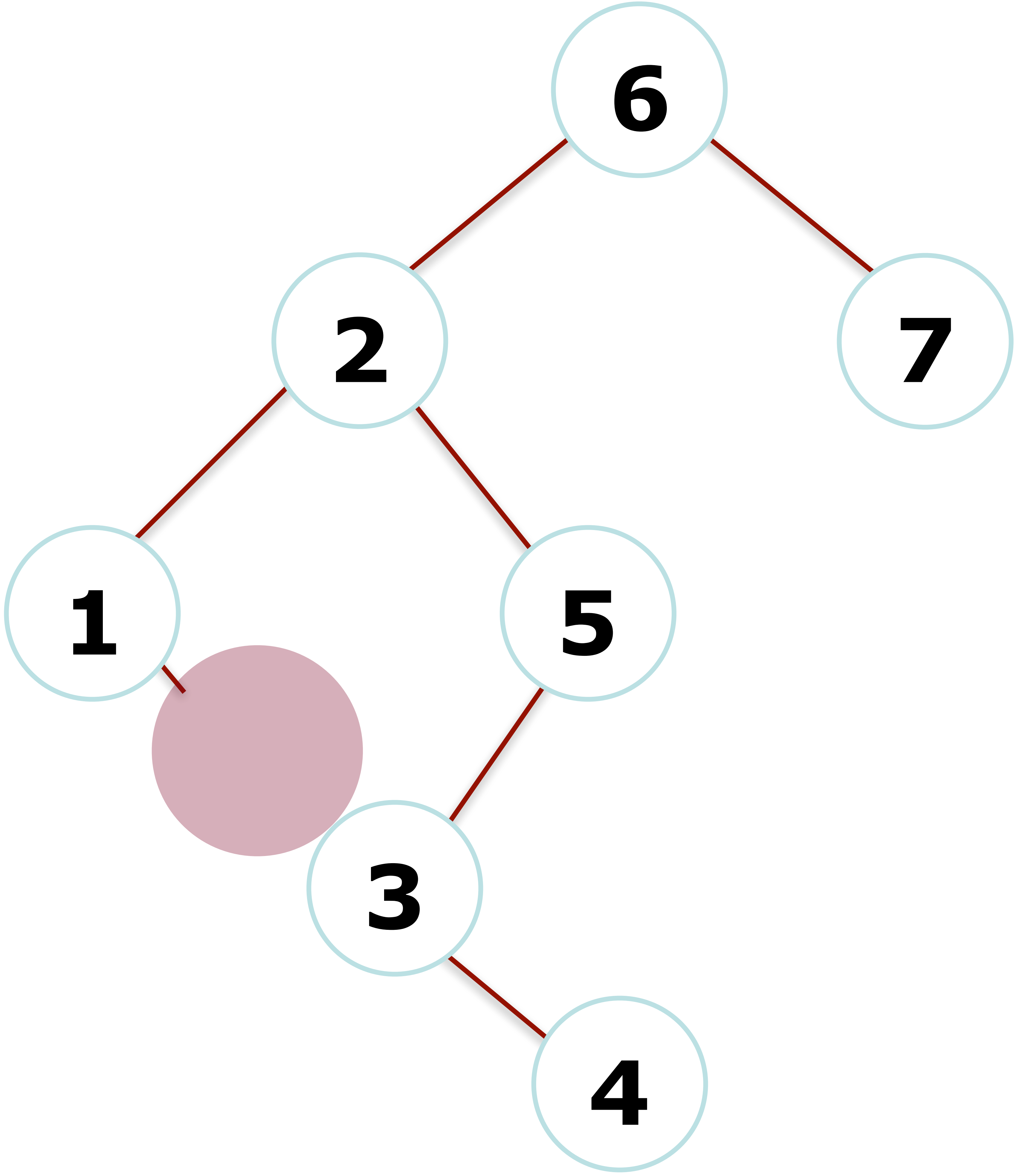



Current Node: 1

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

Output: 1

# In-Order Traversal Example: printing

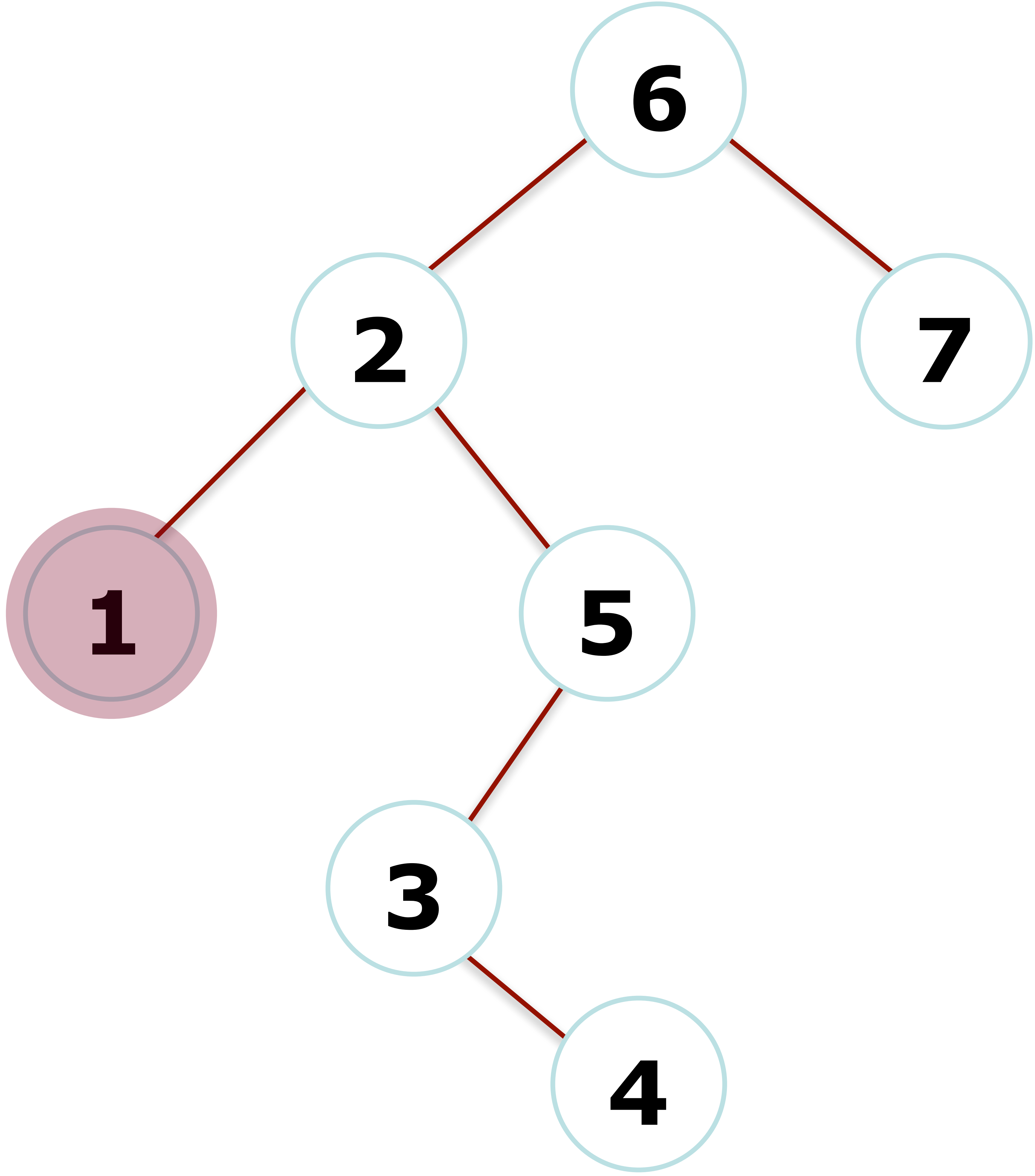



Current Node: NULL

1. current NULL: return

Output: 1

# In-Order Traversal Example: printing



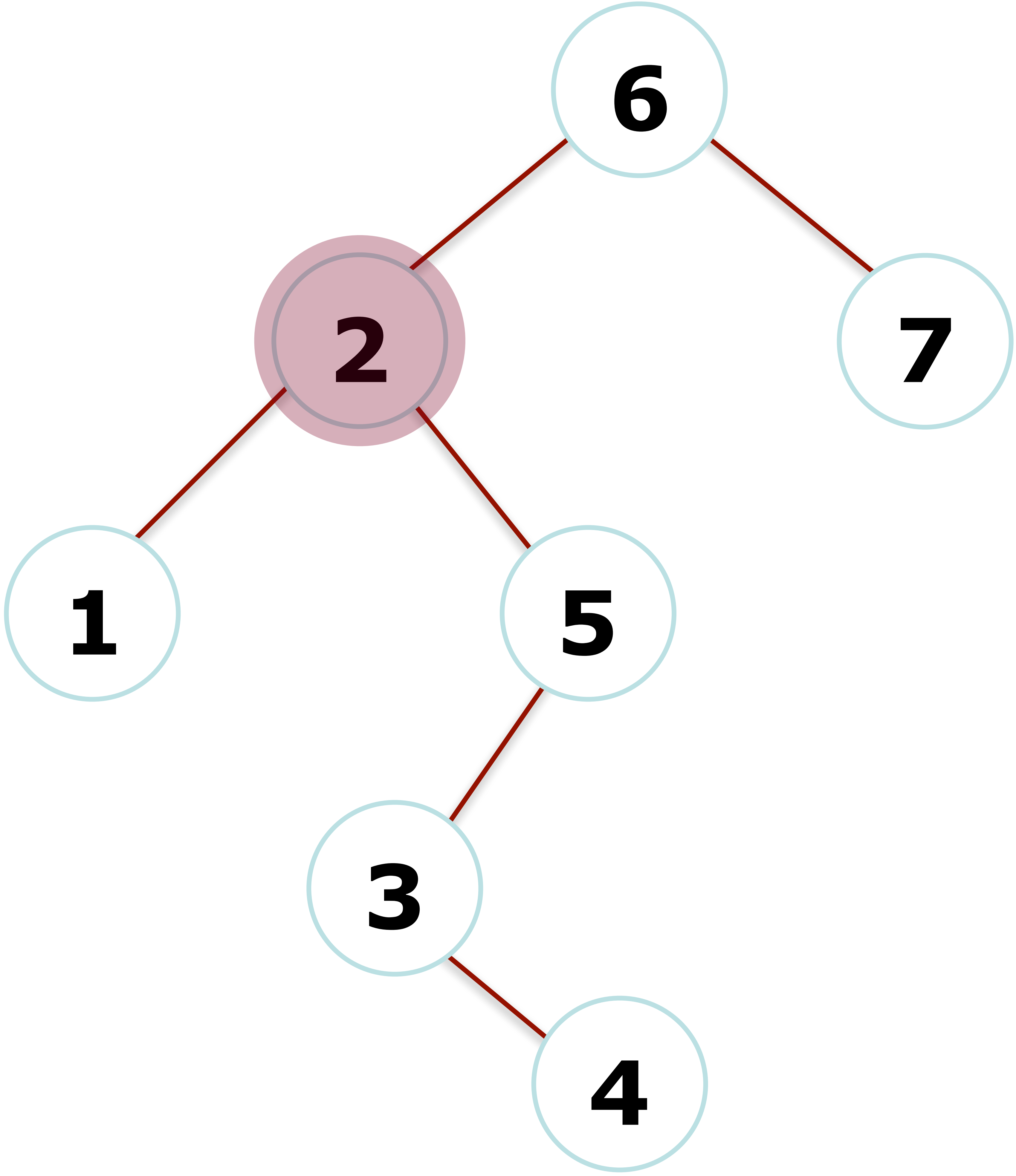

Current Node: 1

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

(function ends)

Output: 1

# In-Order Traversal Example: printing

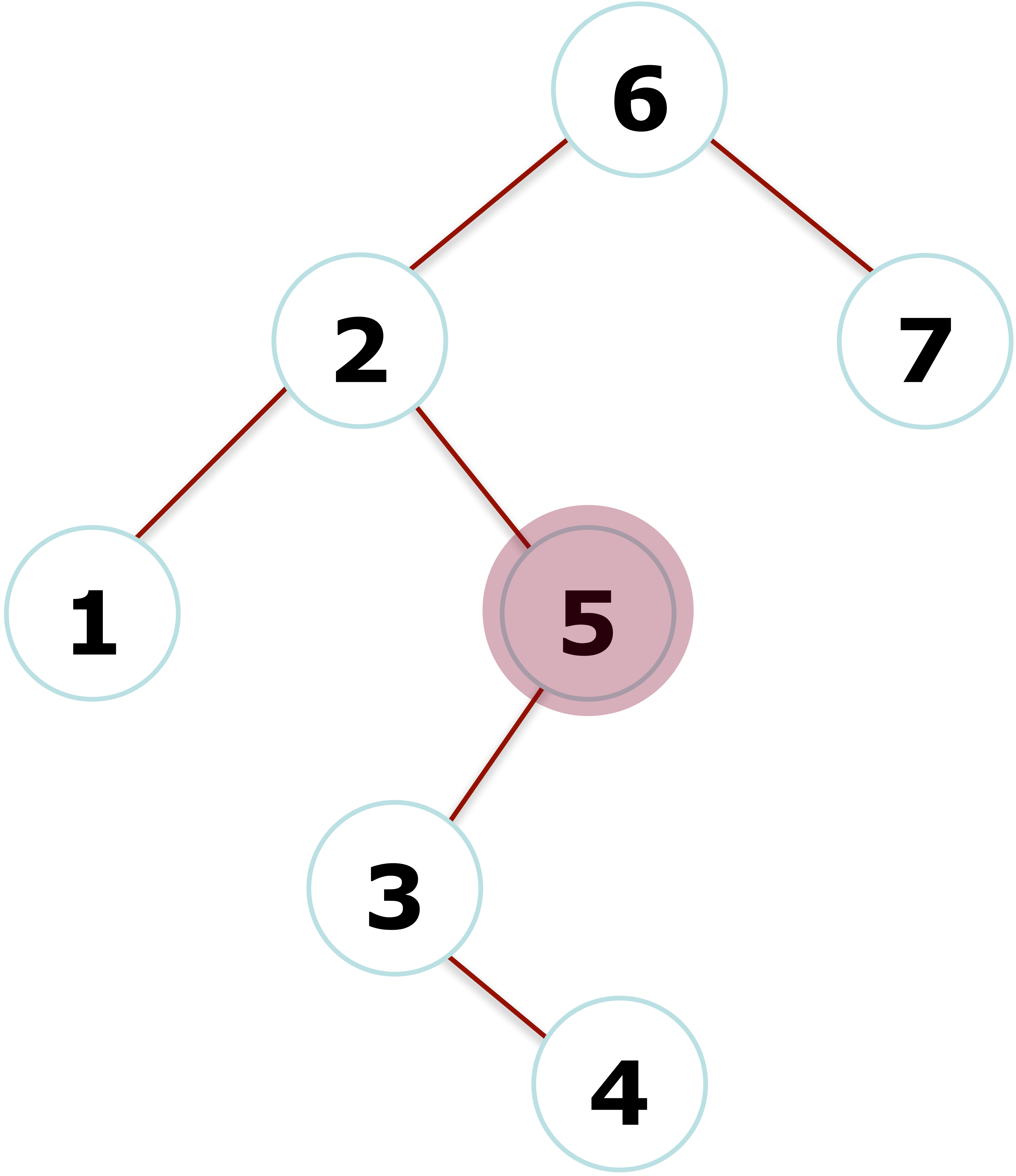



Current Node: 2

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

Output: 1 2

# In-Order Traversal Example: printing

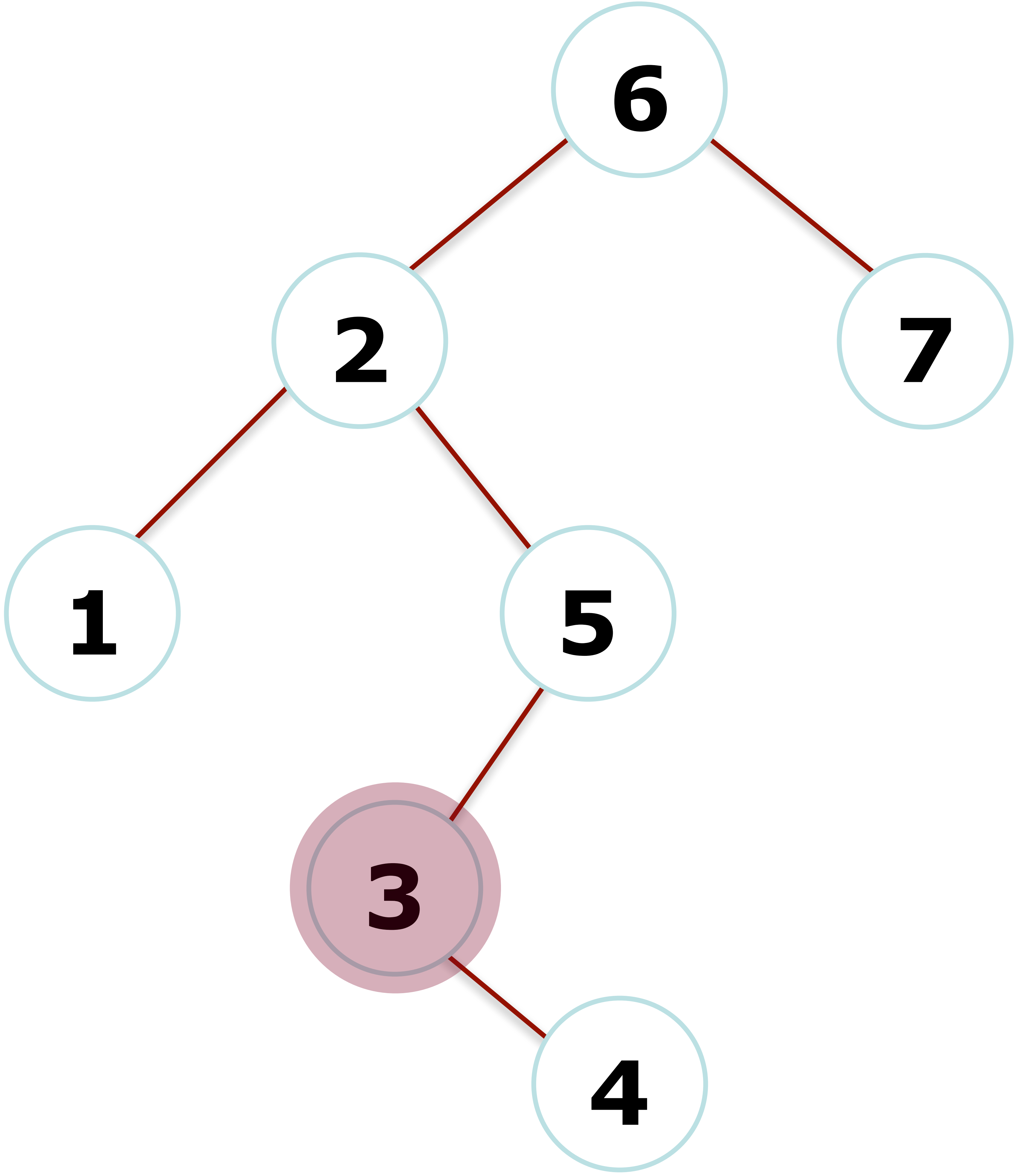



Current Node: 5

1. current not NULL
2. recurse left

Output: 1 2

# In-Order Traversal Example: printing

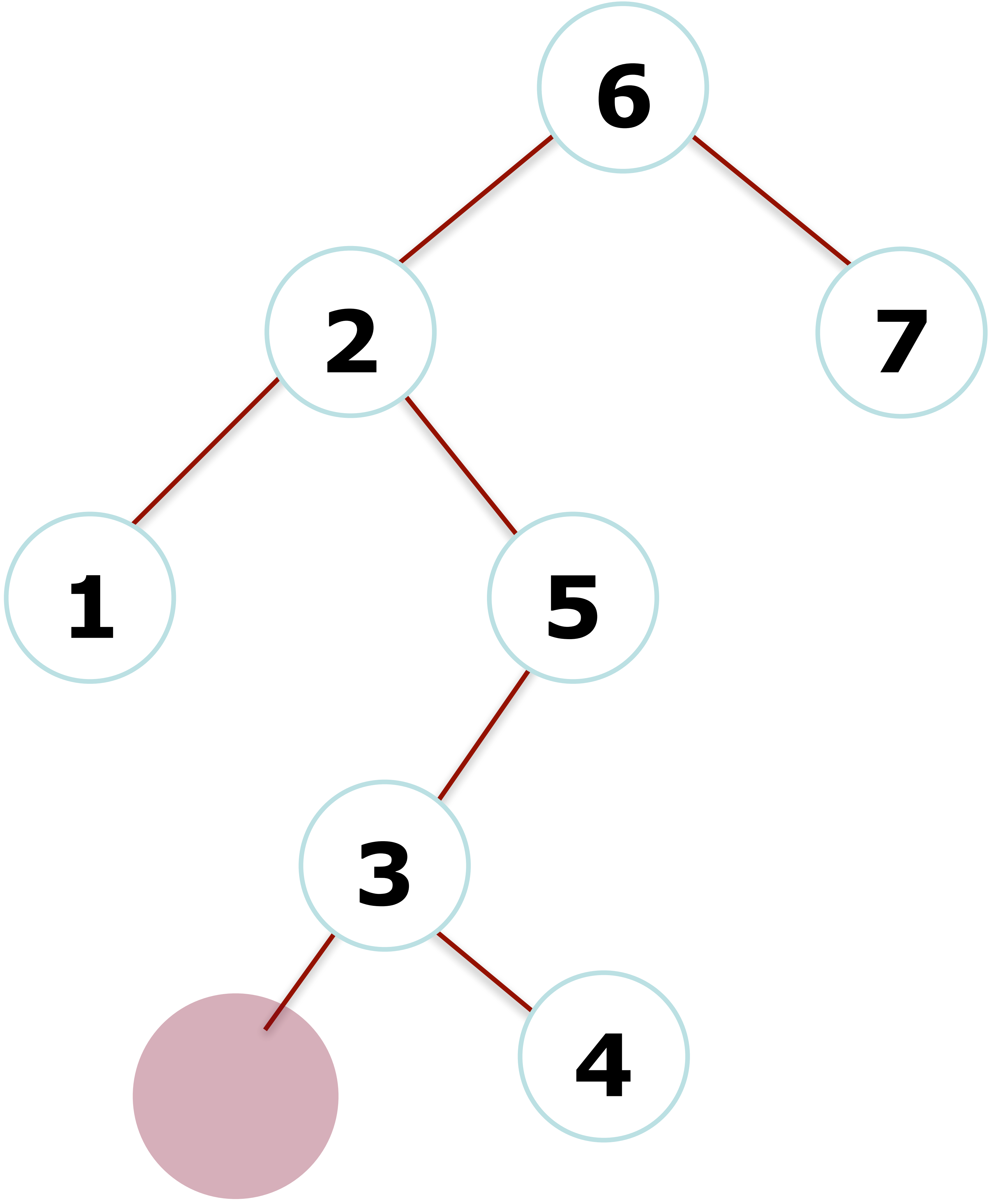



Current Node: 3

1. current not NULL
2. recurse left

Output: 1 2

# In-Order Traversal Example: printing

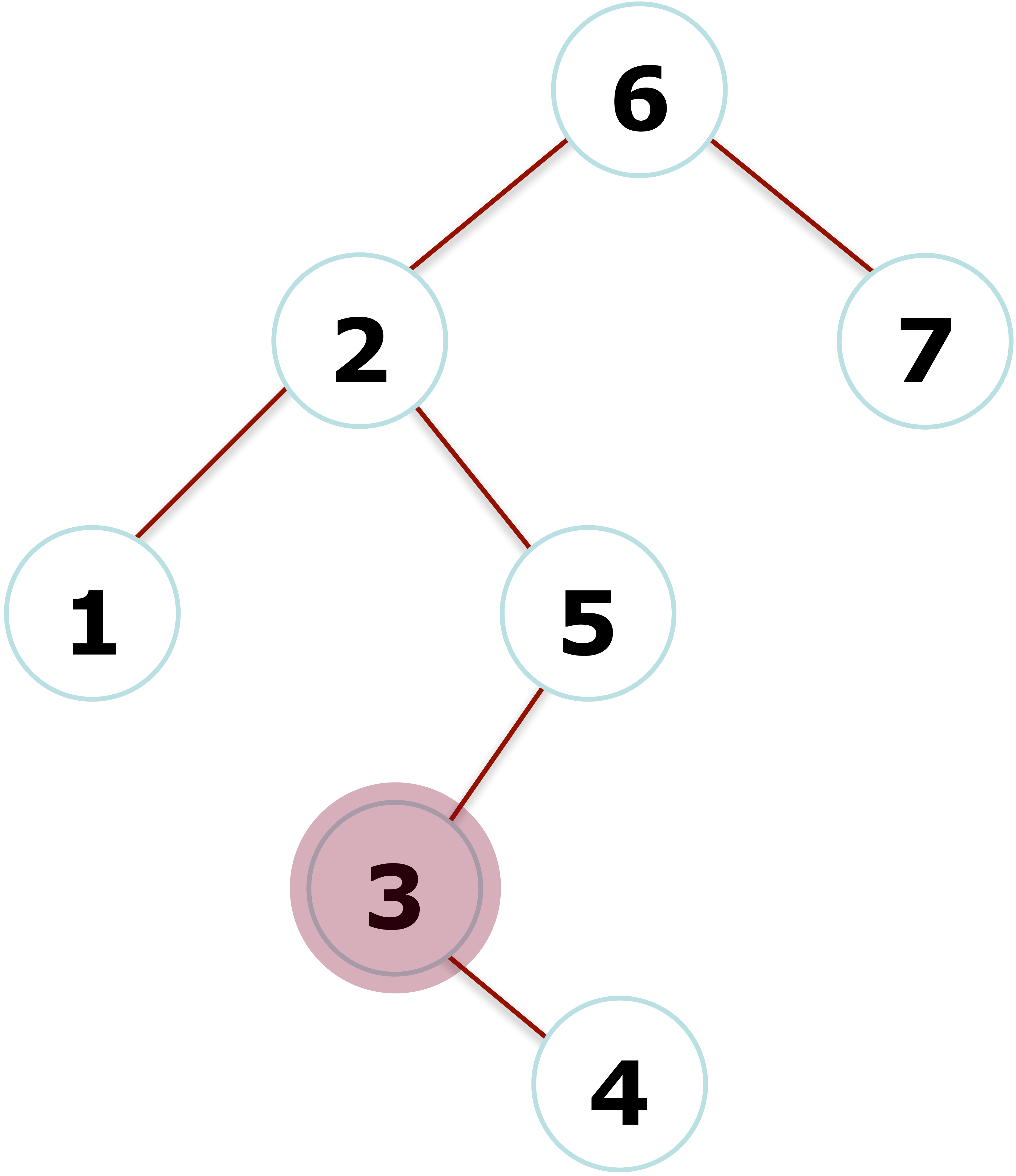



Current Node: NULL

1. current NULL: return

Output: 1 2

# In-Order Traversal Example: printing

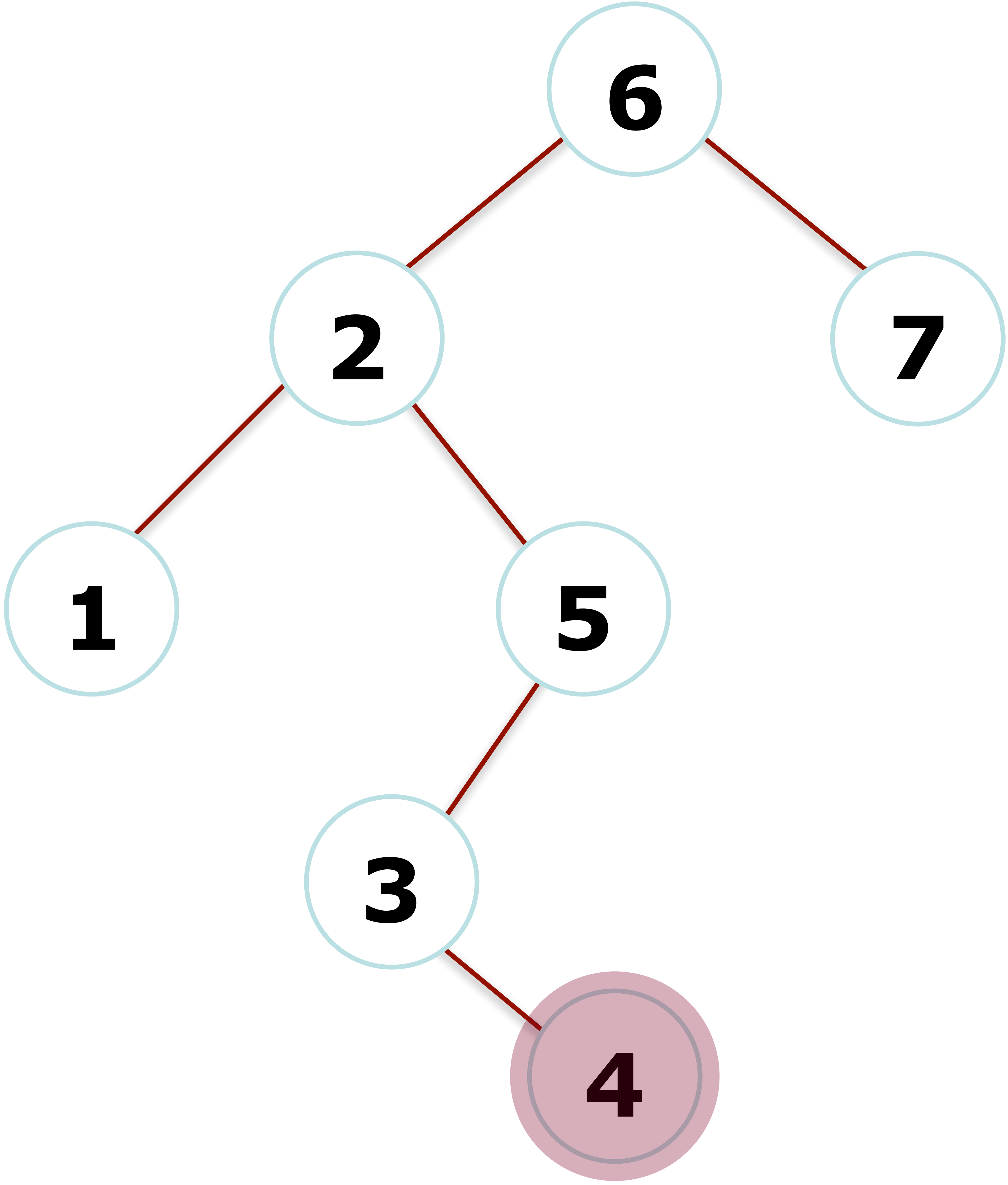



Current Node: 3

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

Output: 1 2 3

# In-Order Traversal Example: printing

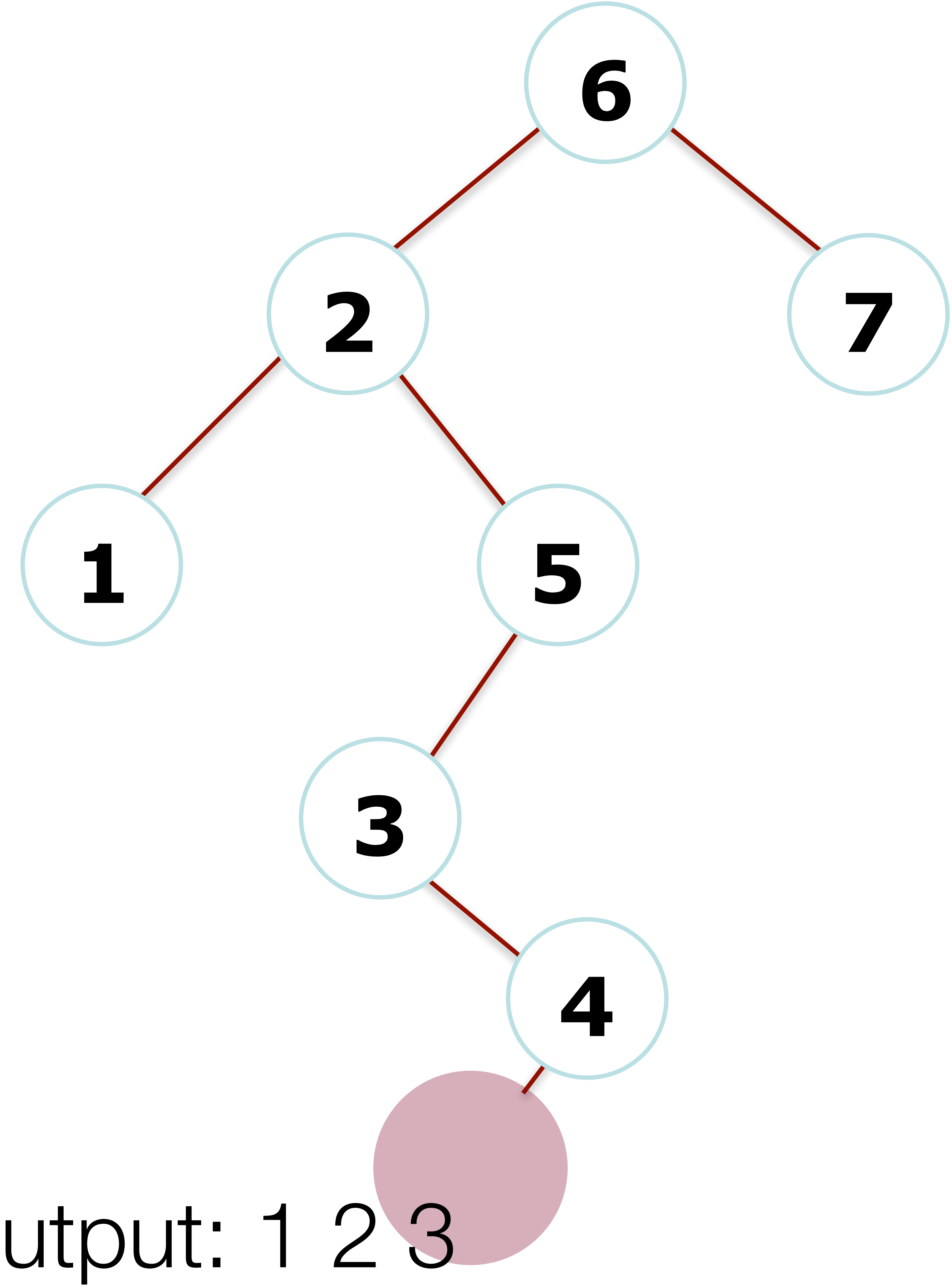



Current Node: 4

1. current not NULL
2. recurse left

Output: 1 2 3

# In-Order Traversal Example: printing

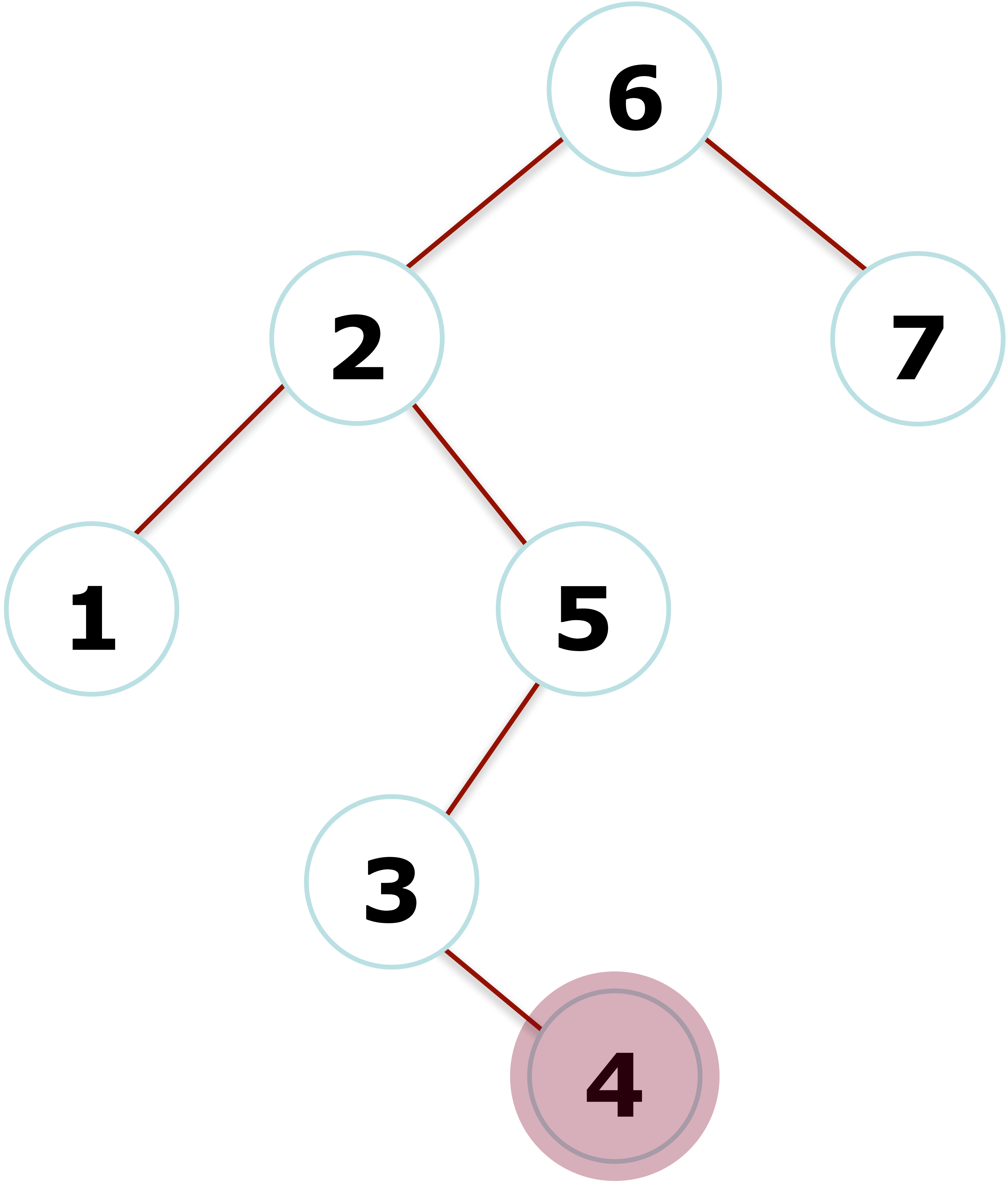



Current Node: NULL

1. current NULL, return

Output: 1 2 3

# In-Order Traversal Example: printing

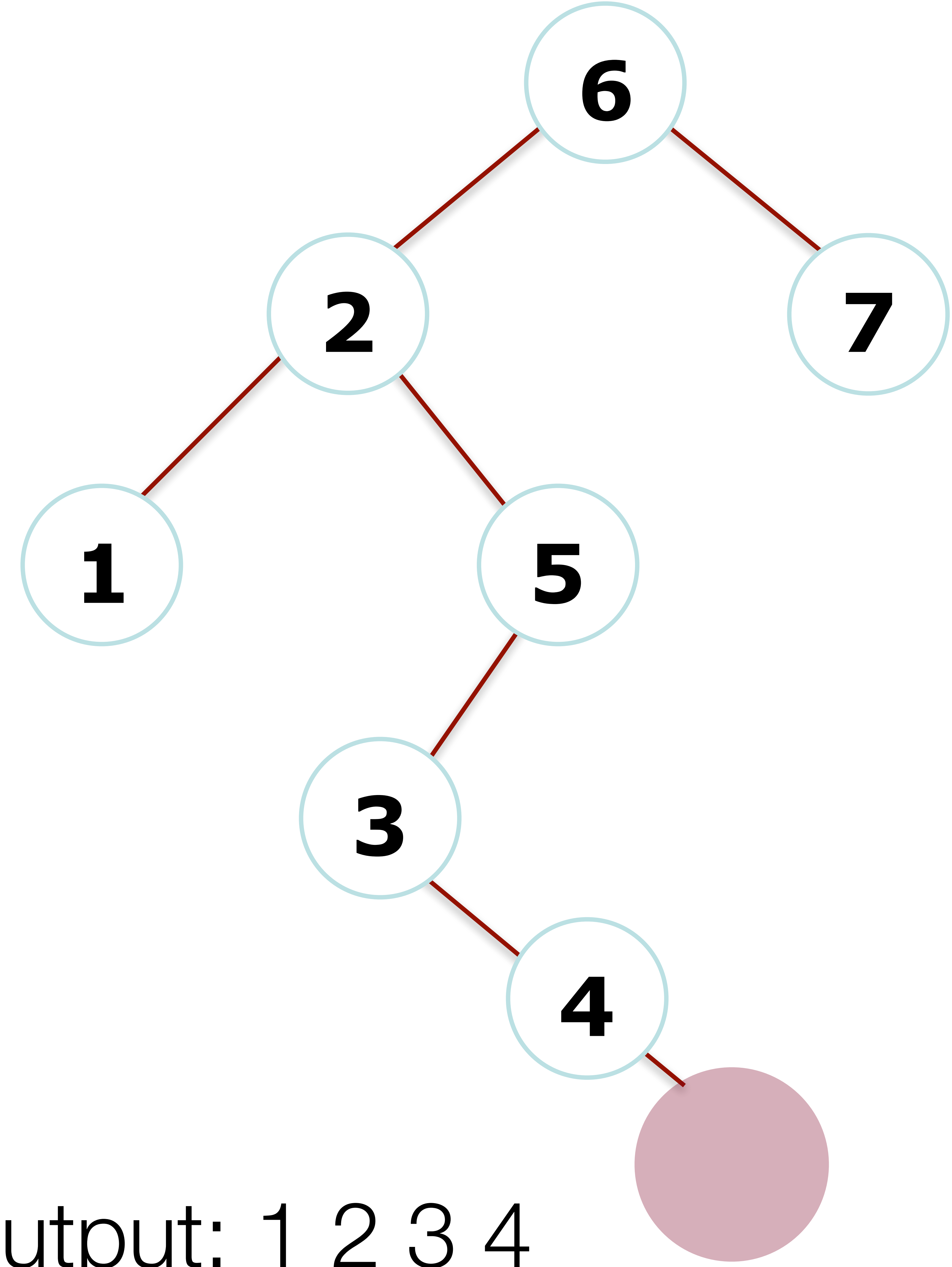



Current Node: 4

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

Output: 1 2 3 4

# In-Order Traversal Example: printing

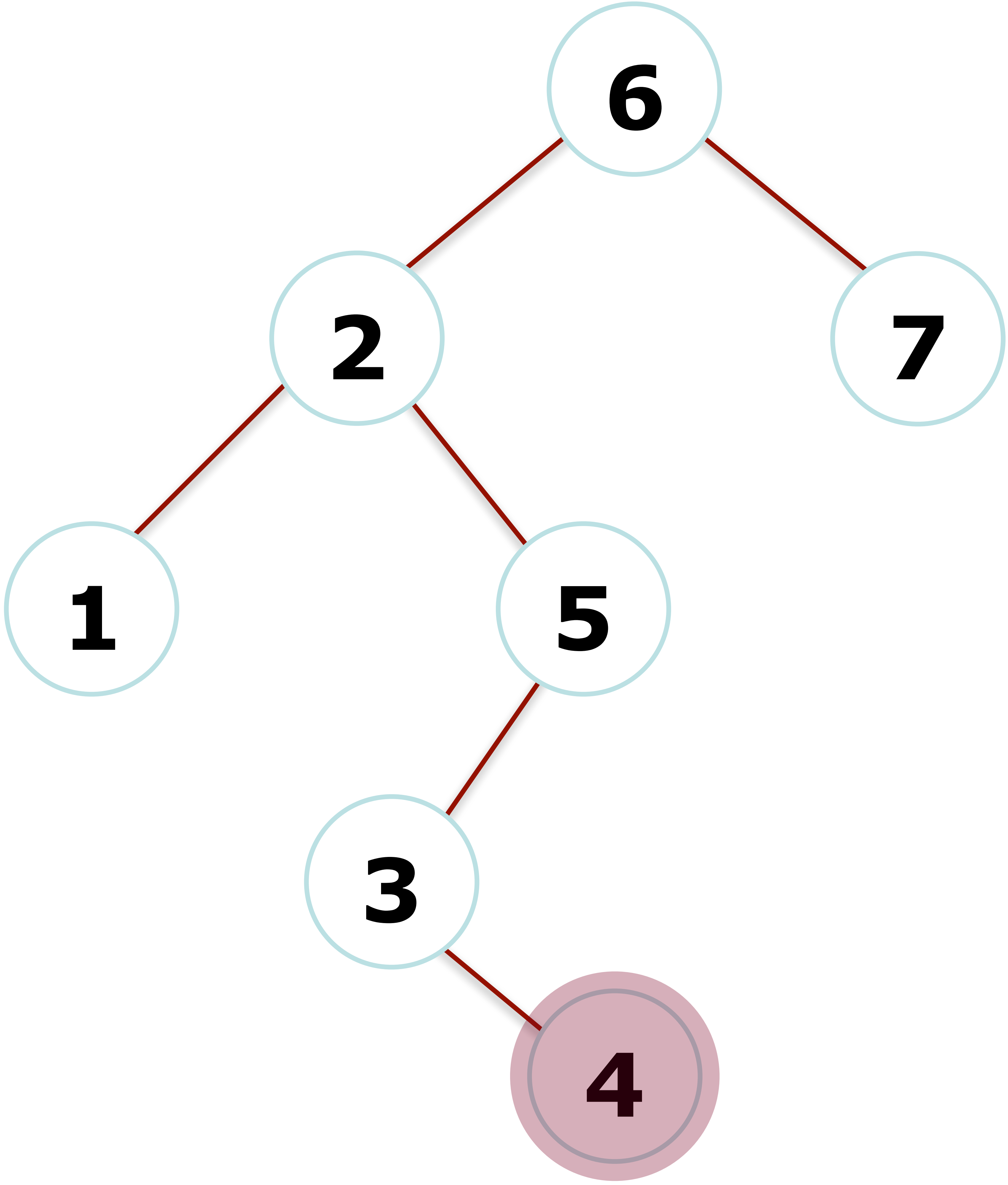



Current Node: NULL

1. current NULL, return

Output: 1 2 3 4

# In-Order Traversal Example: printing



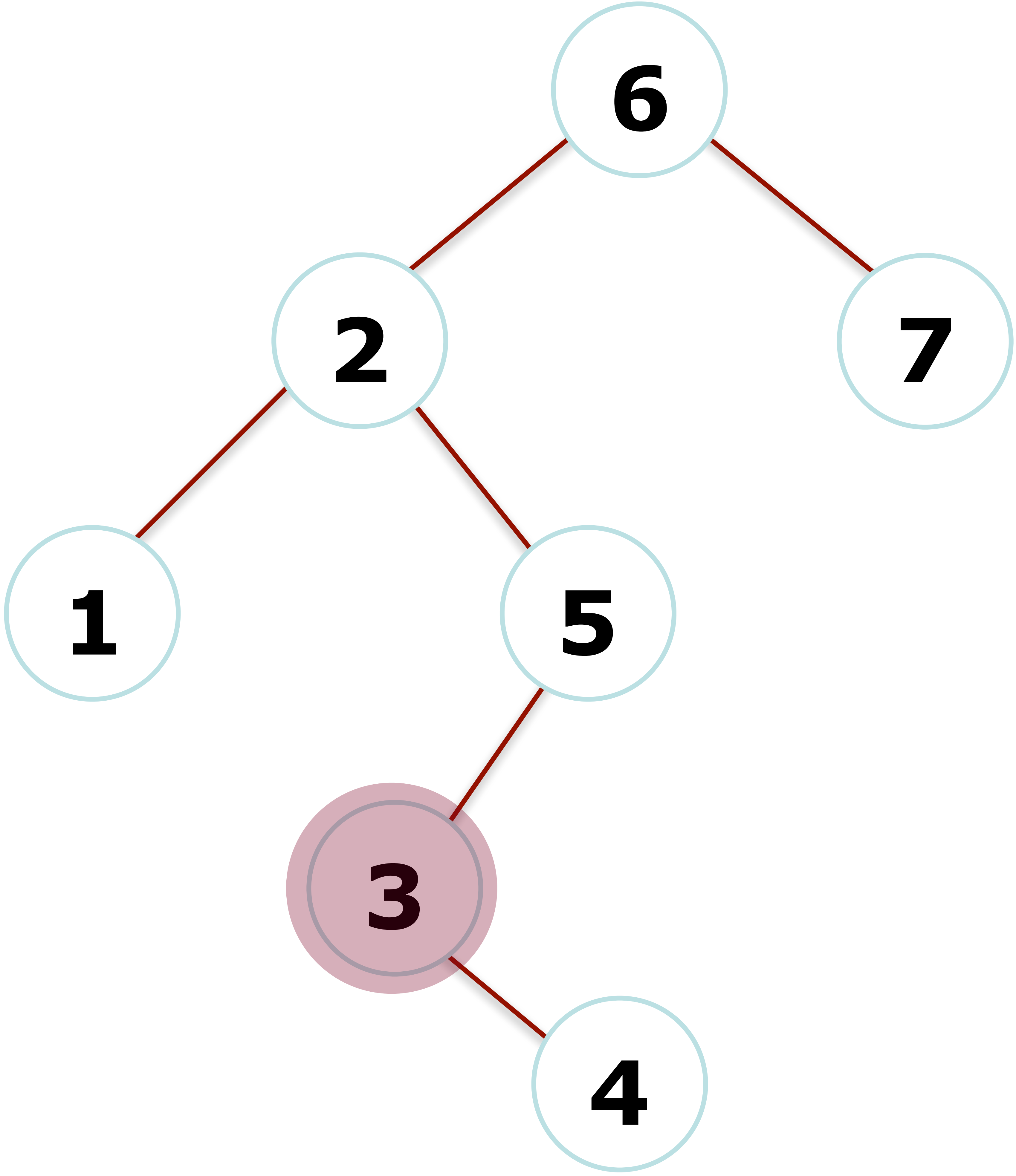

Current Node: 4

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



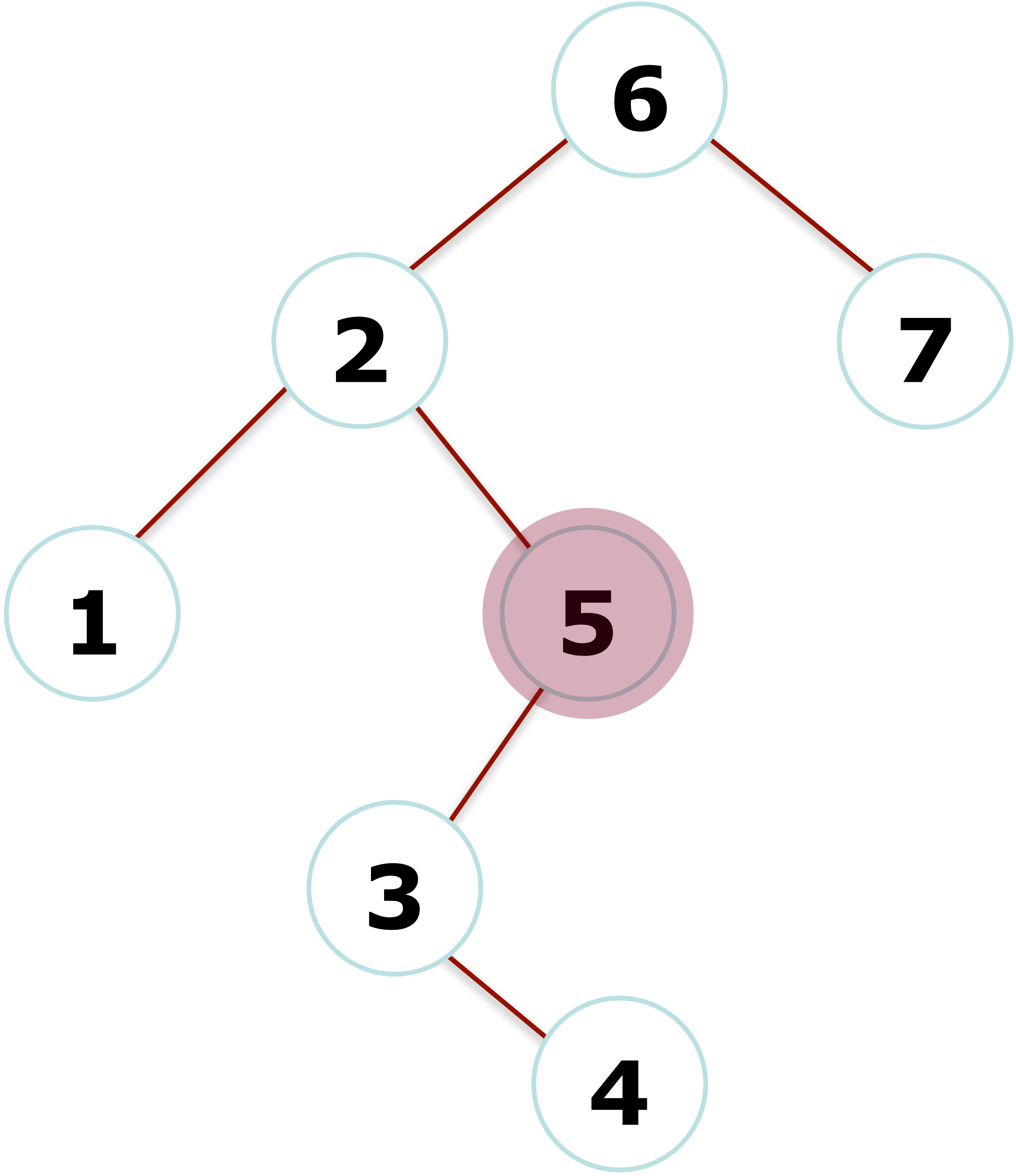

Current Node: 3

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

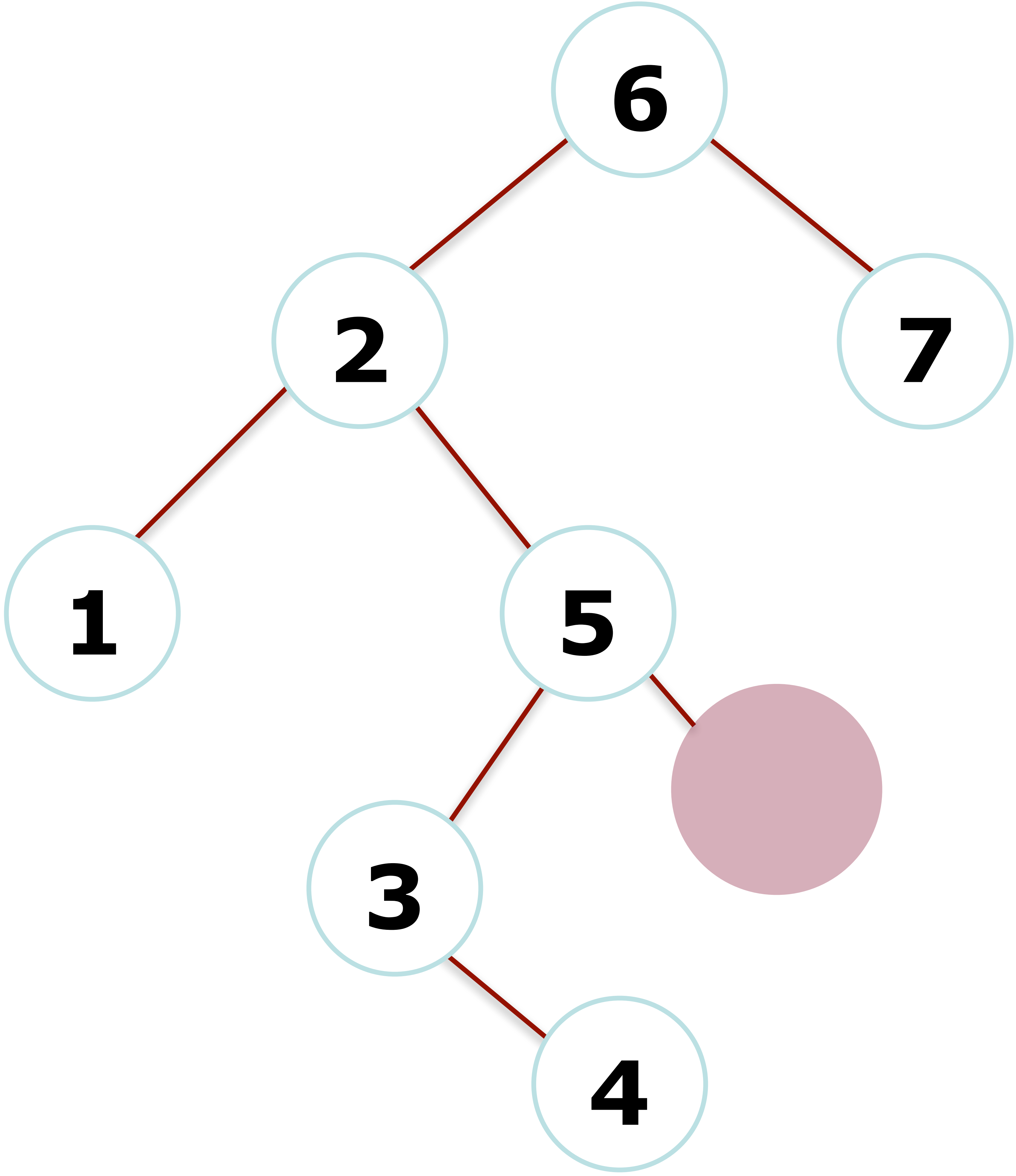



Current Node: 5

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

Output: 1 2 3 4 5

# In-Order Traversal Example: printing

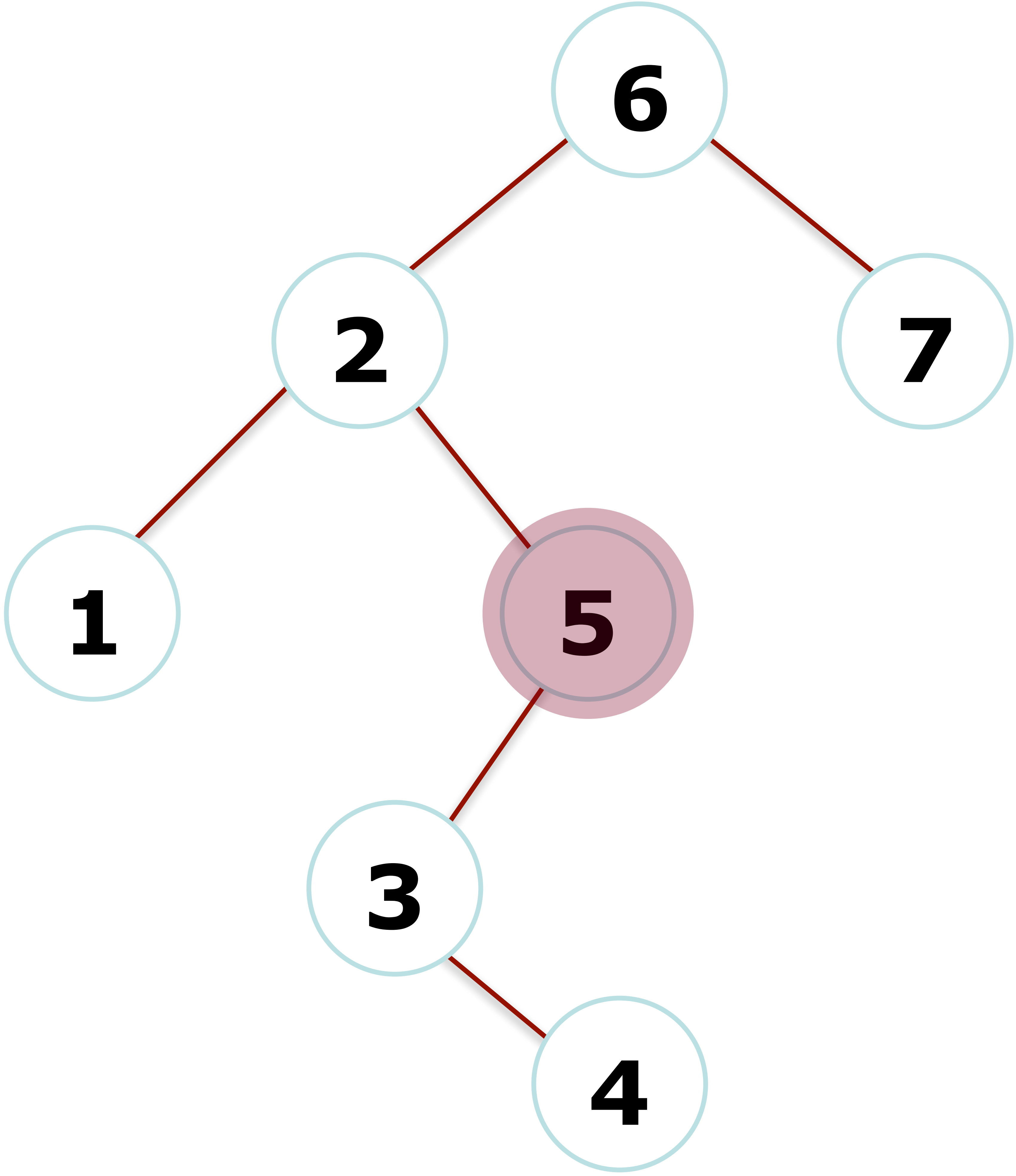



Current Node: NULL

1. current NULL, return

Output: 1 2 3 4 5

# In-Order Traversal Example: printing



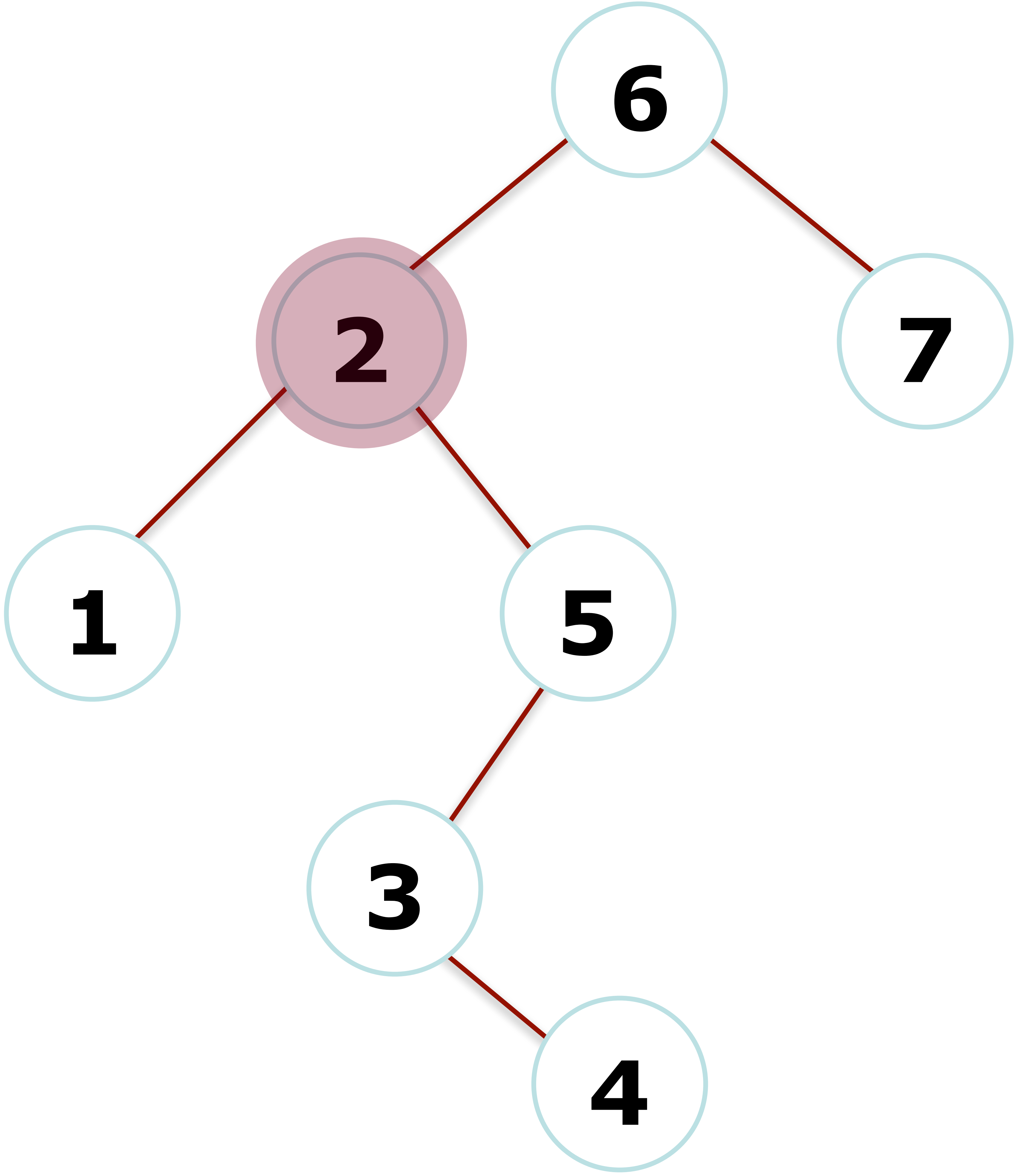

Current Node: 5

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



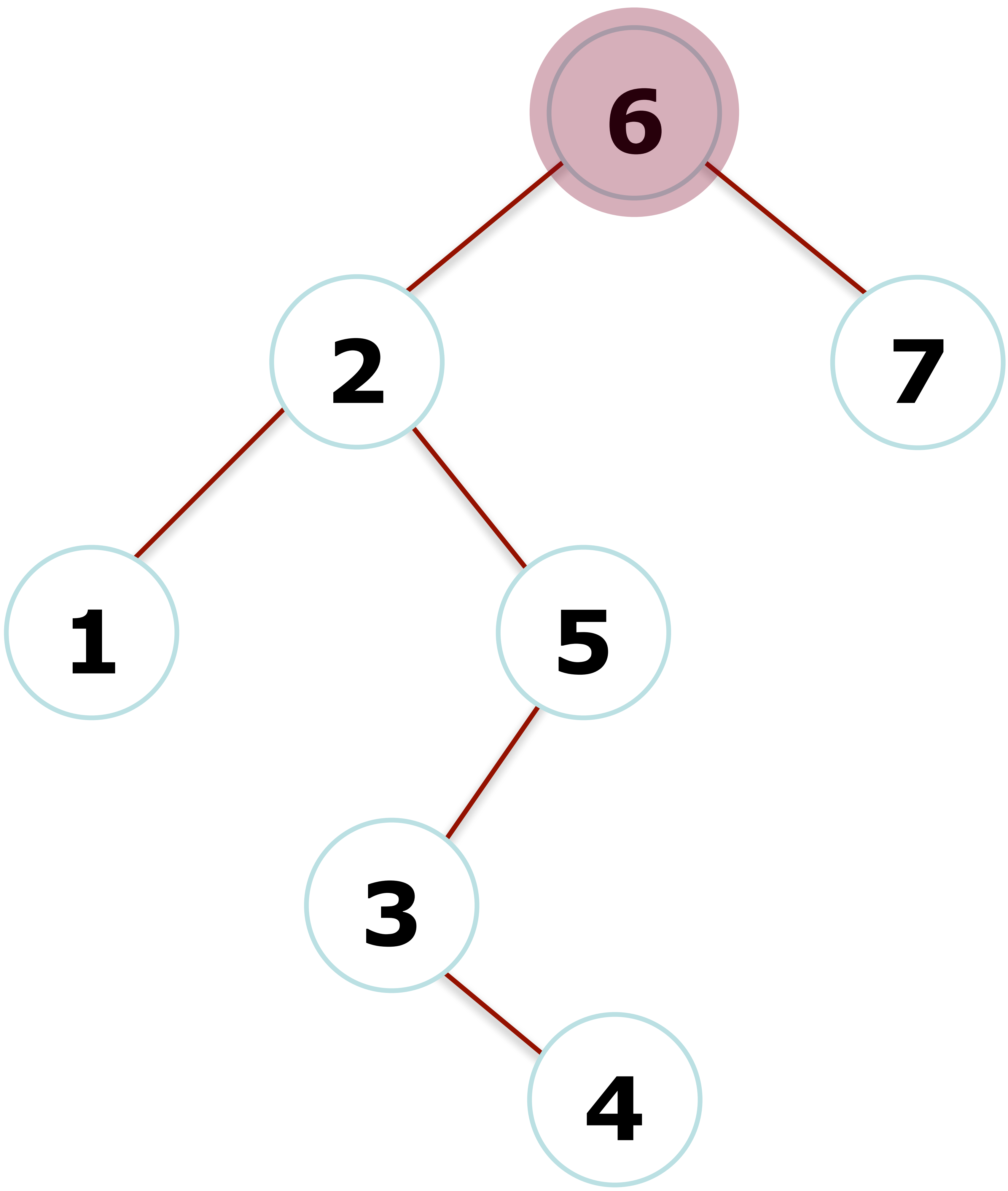

Current Node: 2

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

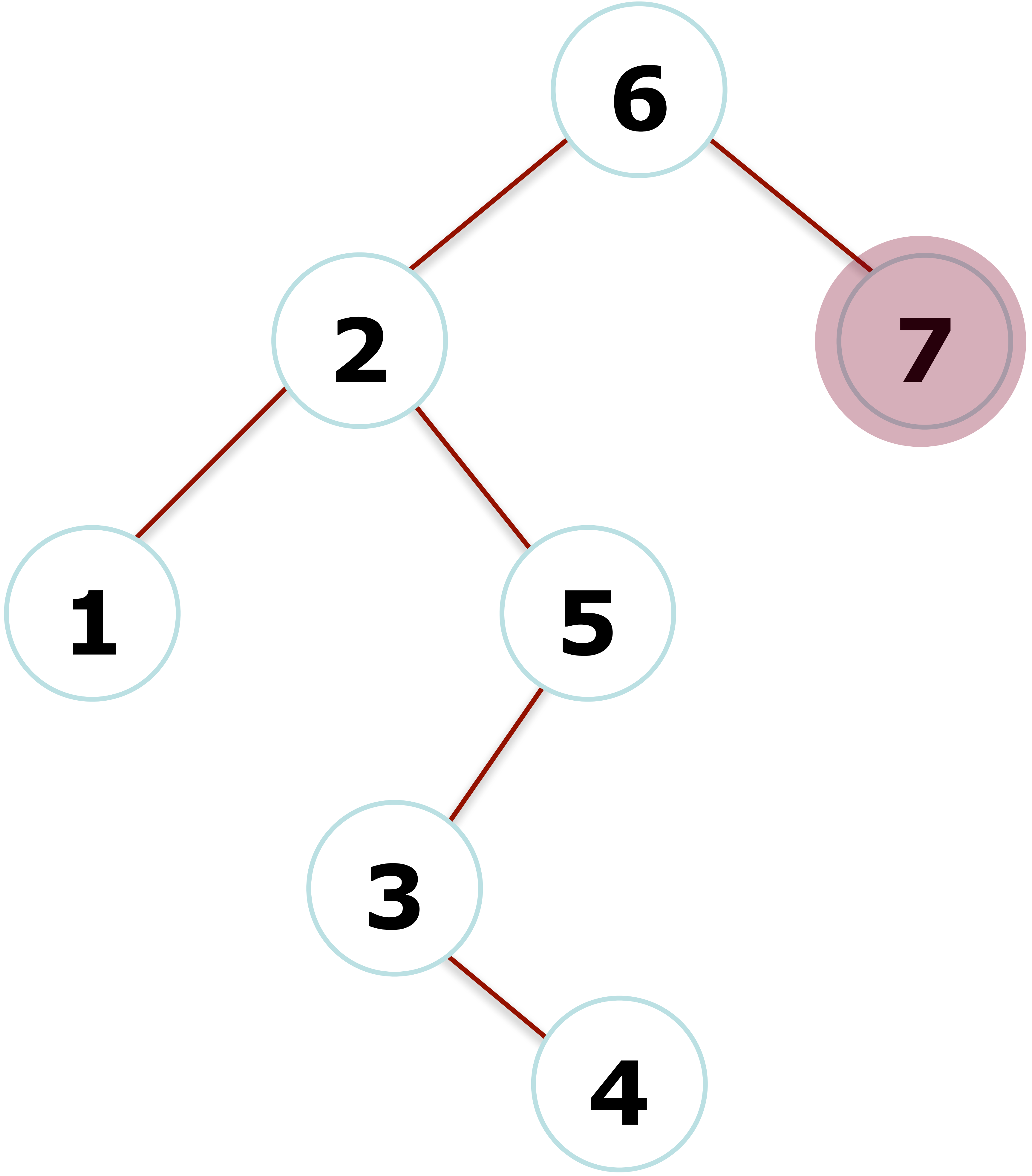



Current Node: 6

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

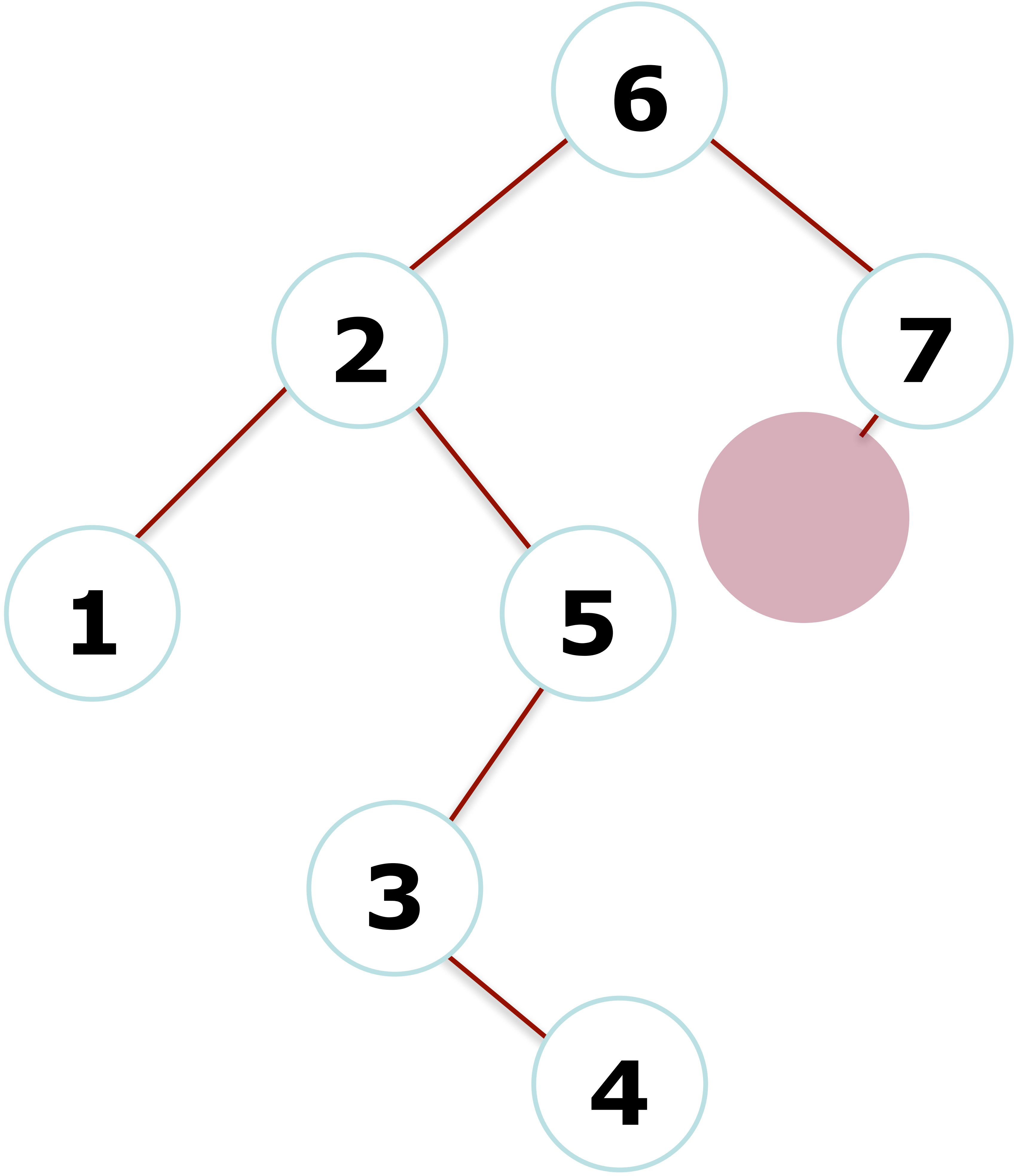



Current Node: 7

1. current not NULL
2. recurse left

Output: 1 2 3 4 5 6

# In-Order Traversal Example: printing

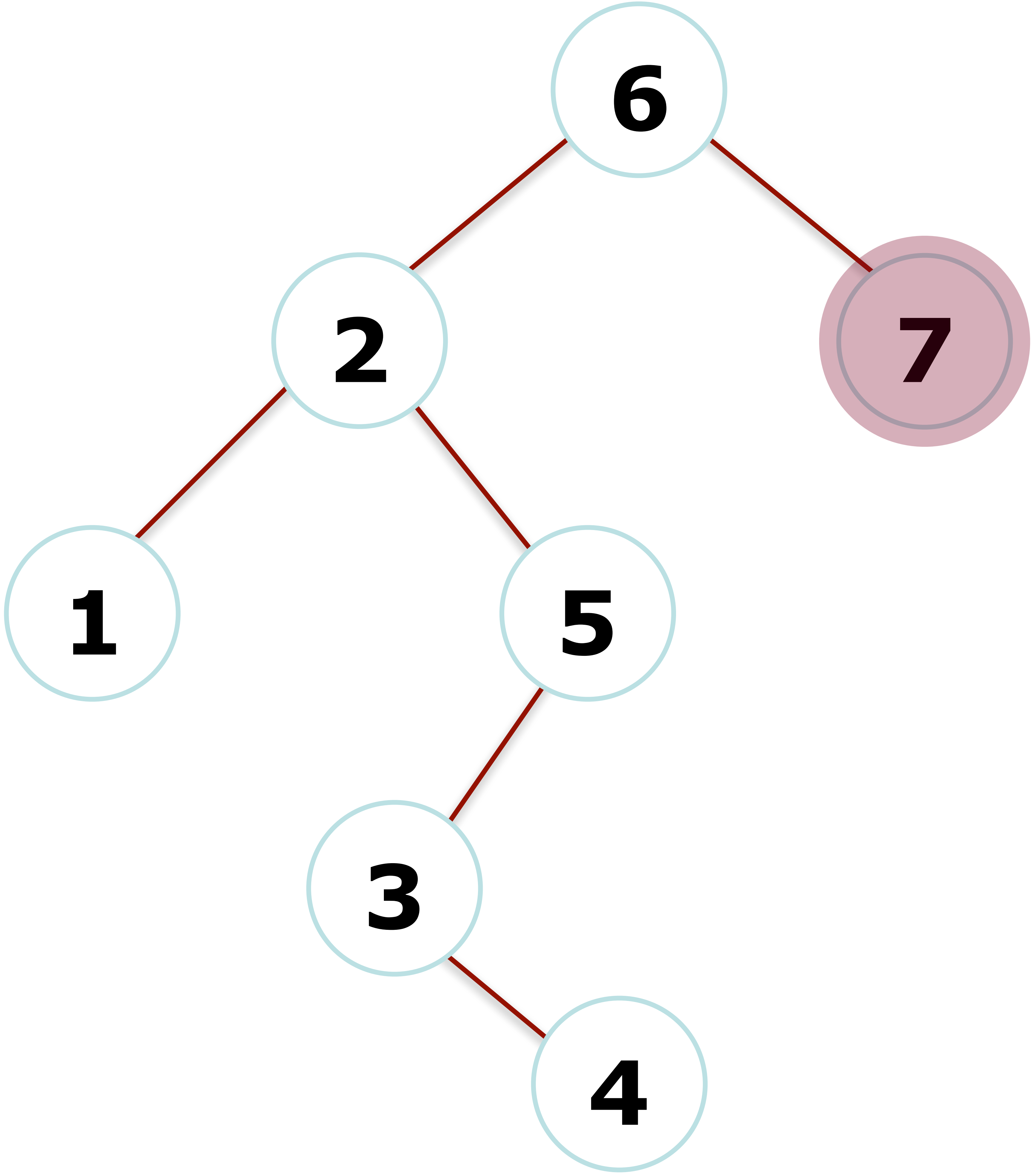



Current Node: NULL

1. current NULL, return

Output: 1 2 3 4 5 6

# In-Order Traversal Example: printing

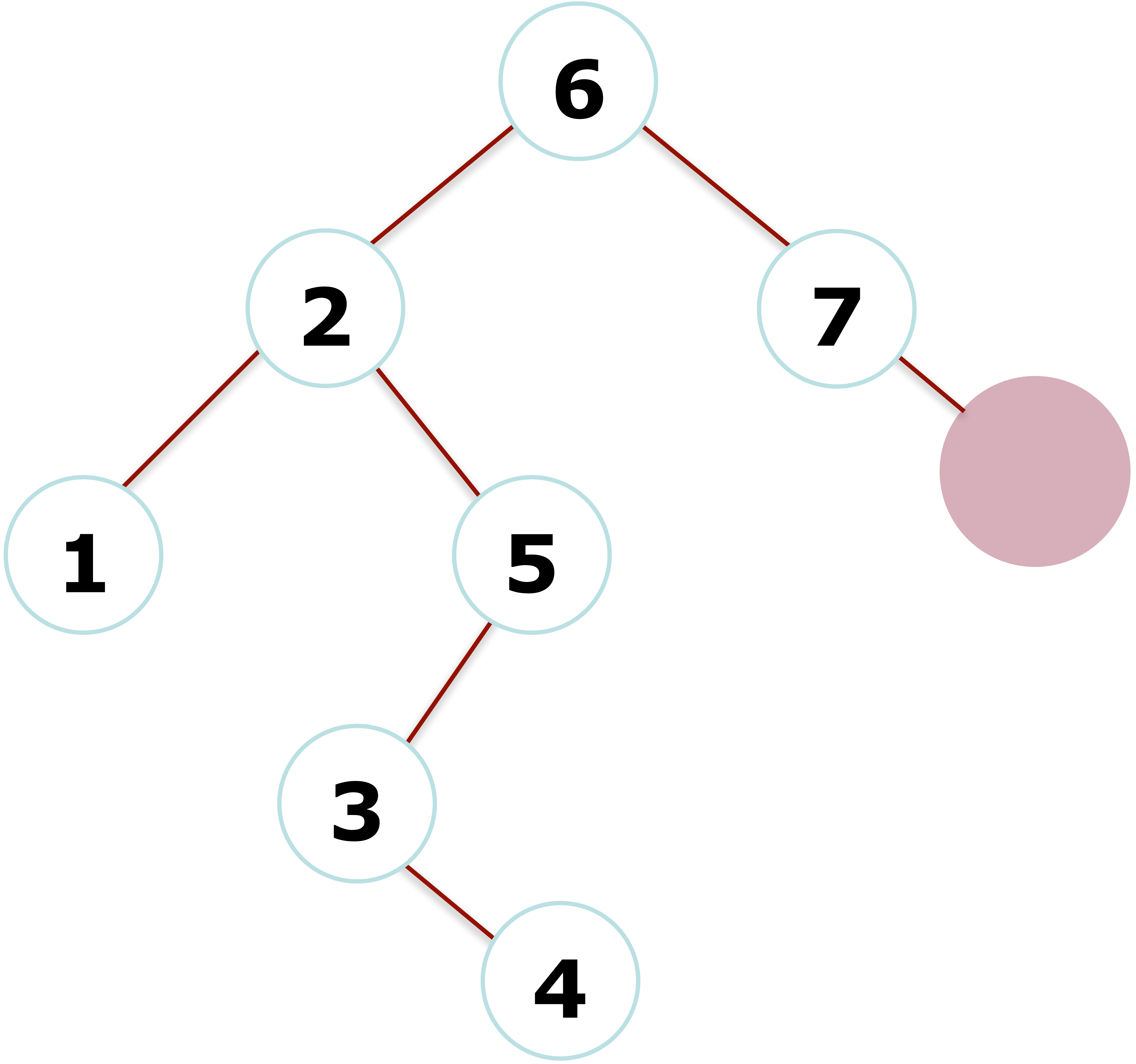



Current Node: 7

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

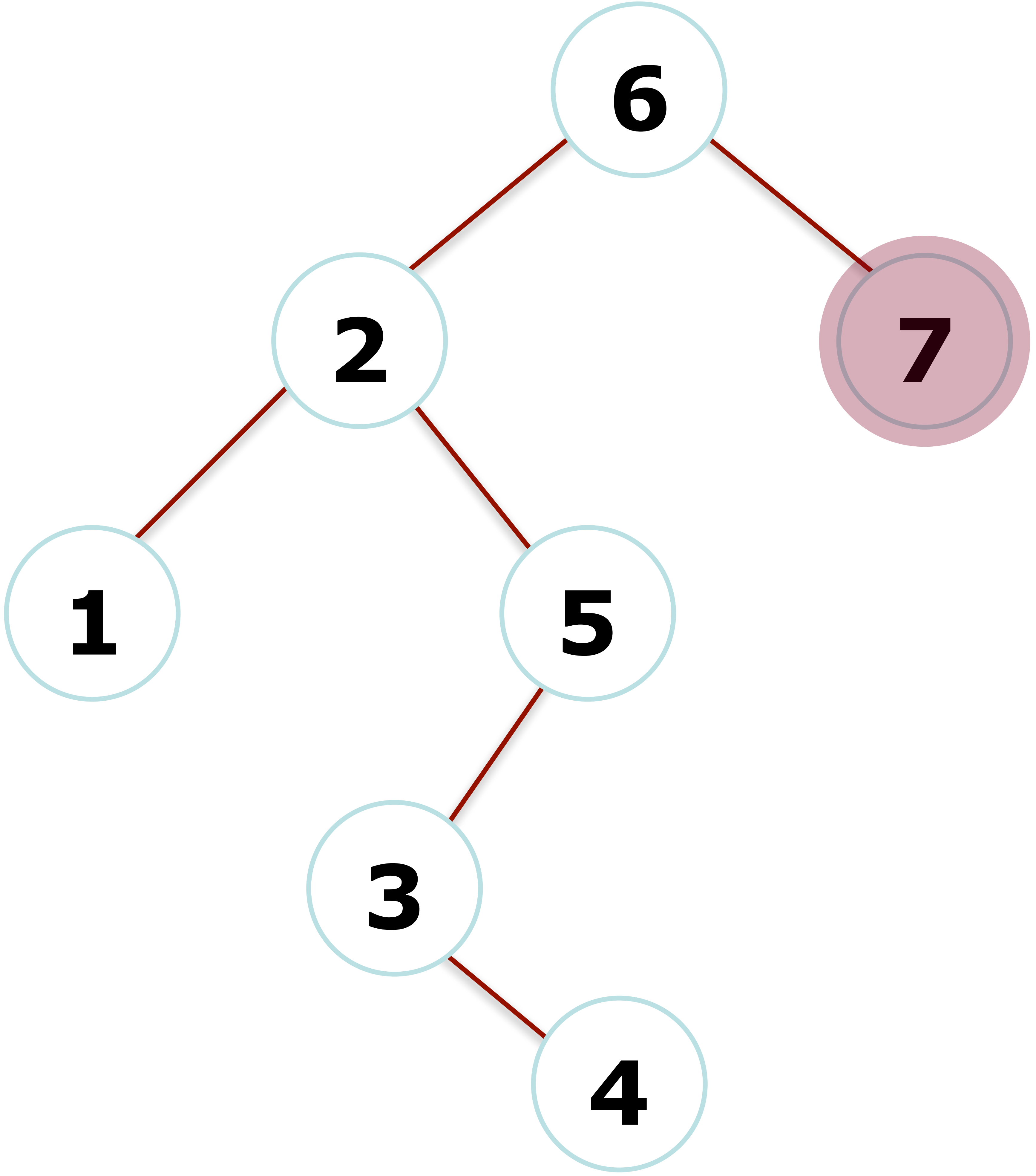



Current Node: NULL

1. current NULL, return

Output: 1 2 3 4 5 6

# In-Order Traversal Example: printing



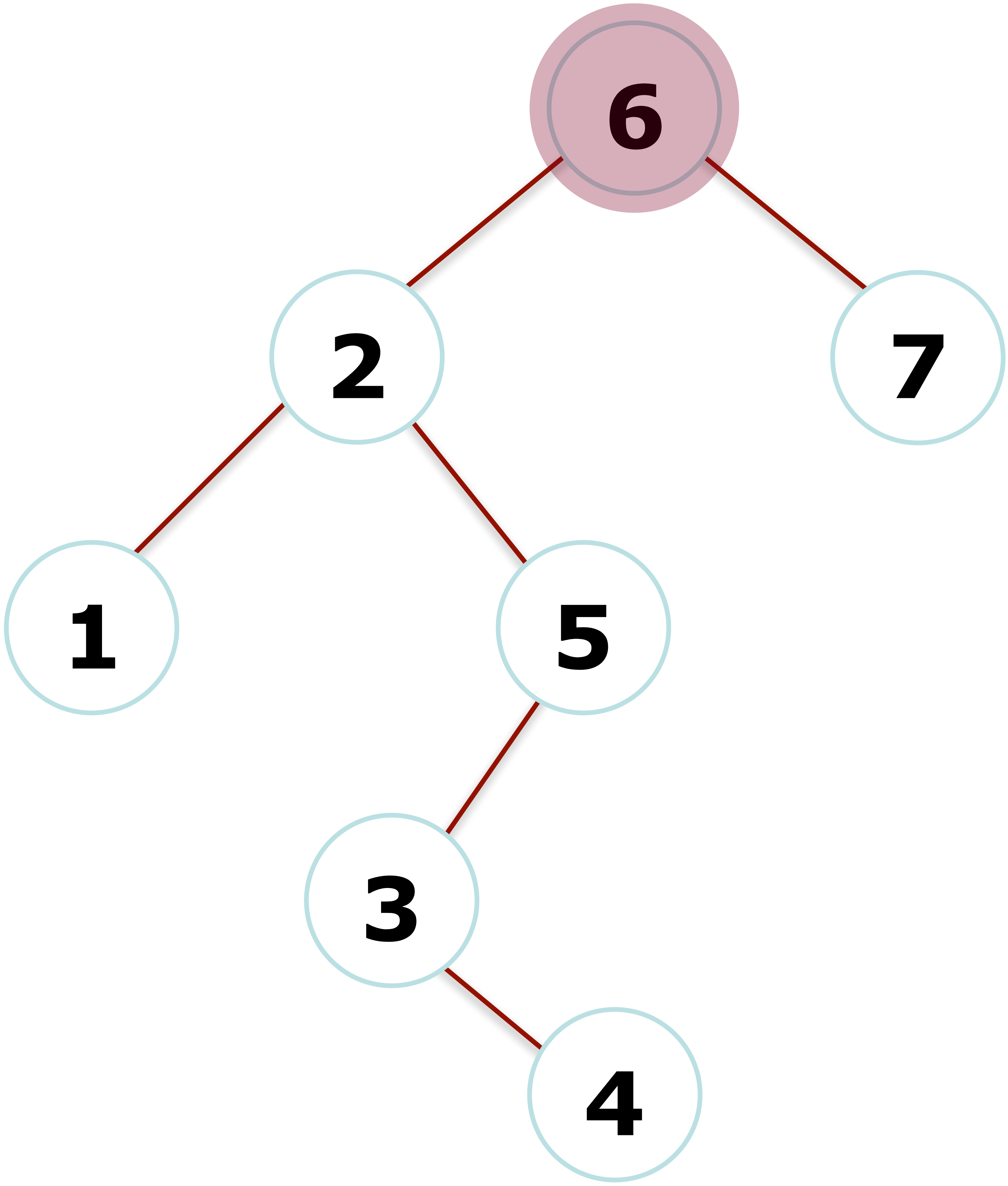

Current Node: 7

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