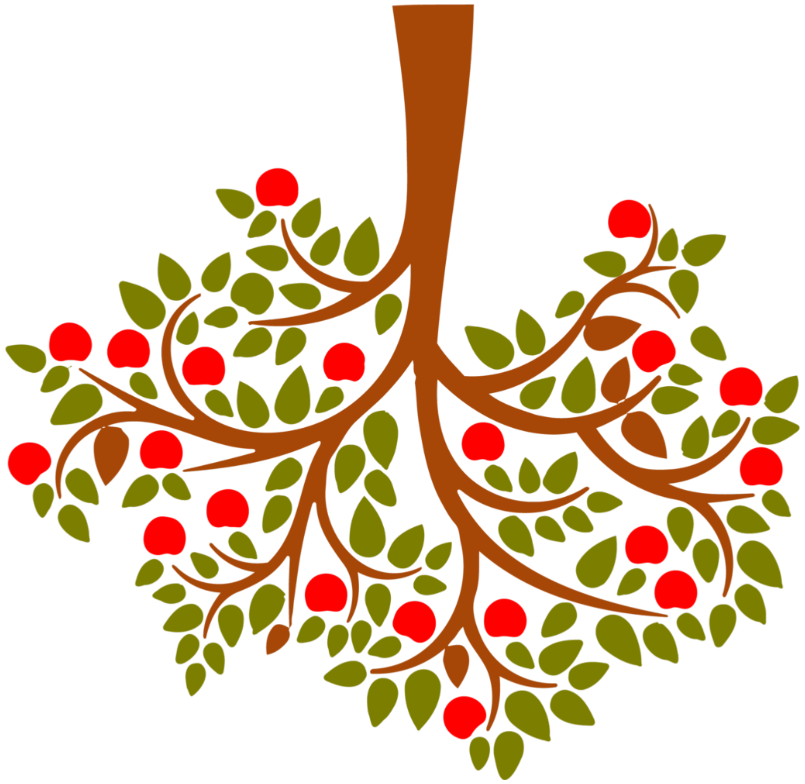
CS11212 - **Spring** 2022

Data Structures & Introduction to **Algorithms**

Data Structures

Trees: Tree Traversals

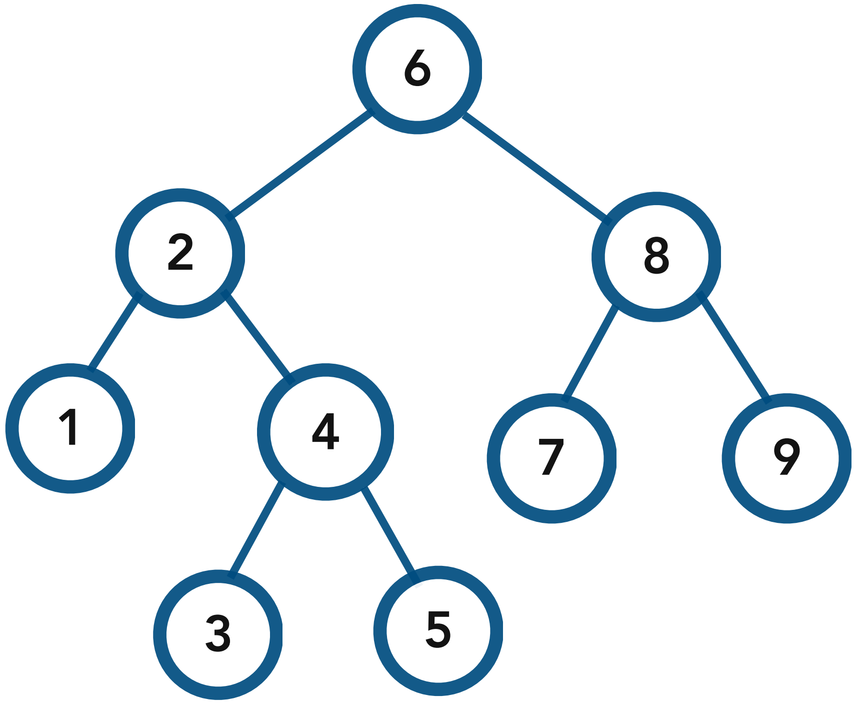
Ibrahim Albluwi



Tree Data Structures

- Definitions and properties
- Basic operations
- Balanced binary search trees
- ● Tree traversals

Printing the Tree (in order)



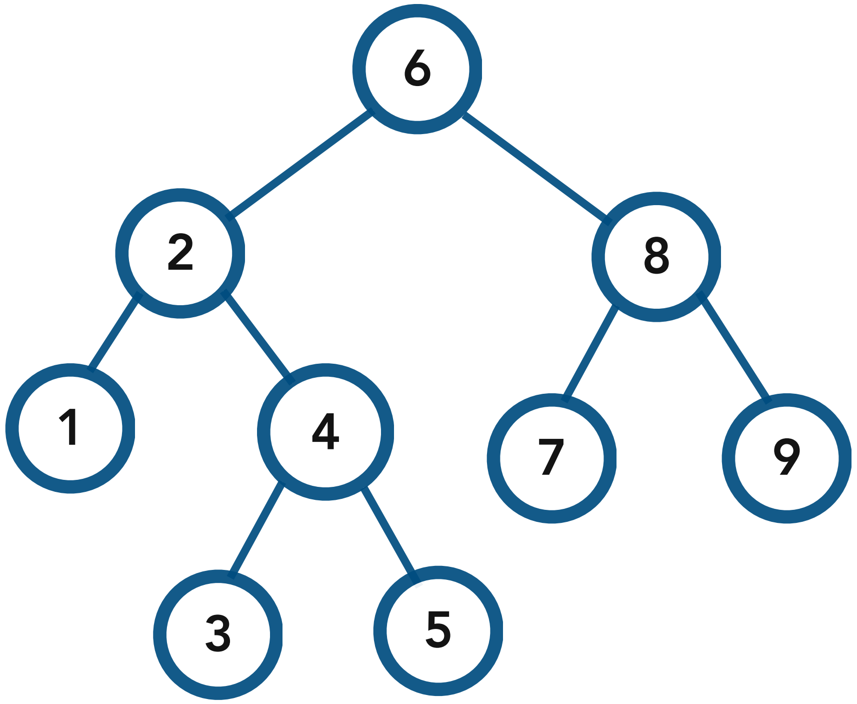
Assuming the tree is a binary search tree, how can we traverse it *in order*?

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes

(recursion ♥)

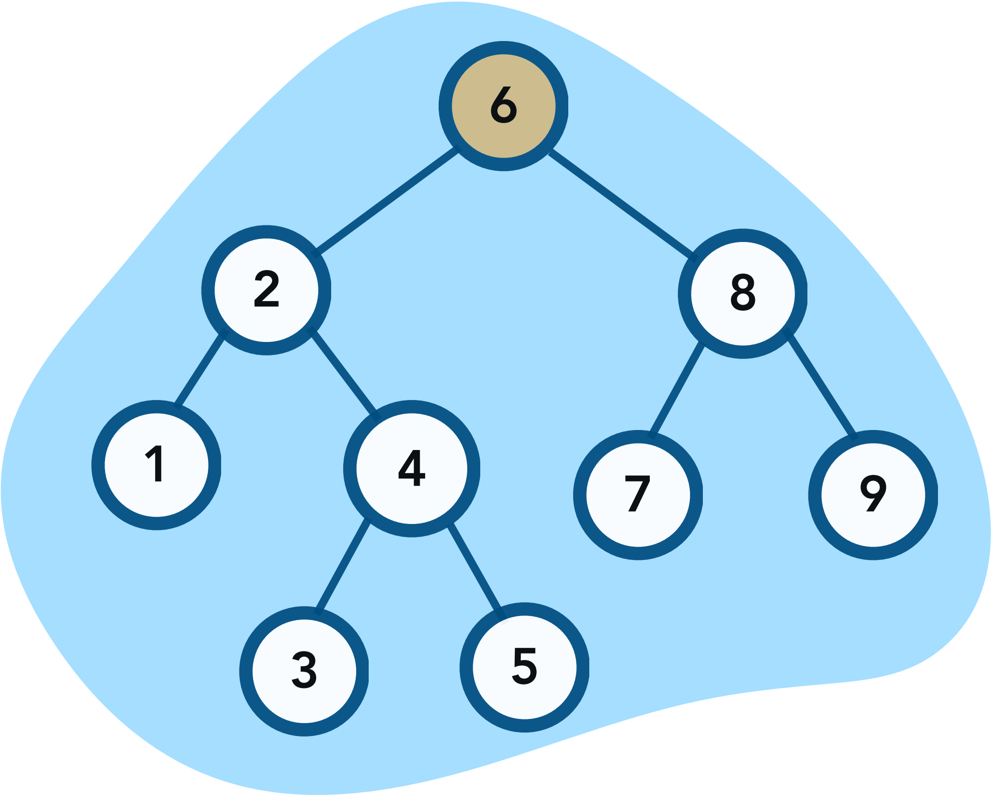


Assuming the tree is a binary search tree, how can we traverse it *in order*?

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
  print the current node, then
  print the larger nodes



print the tree rooted at 6

do not print 6 yet!

print left subtree first.

at (6) left - current - right

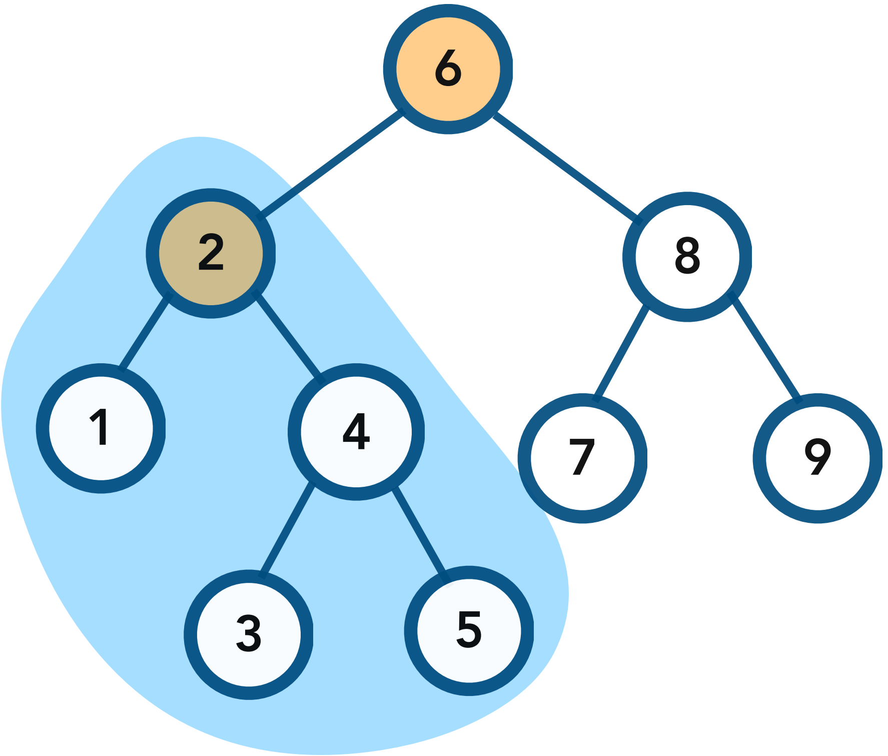
stack frames

Console

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
    print the current node, then
    print the larger nodes



`print the tree rooted at 2`

do not print 2 yet!

print left subtree first.

```
at (2)  left - current - right
at (6)  left - current - right
```

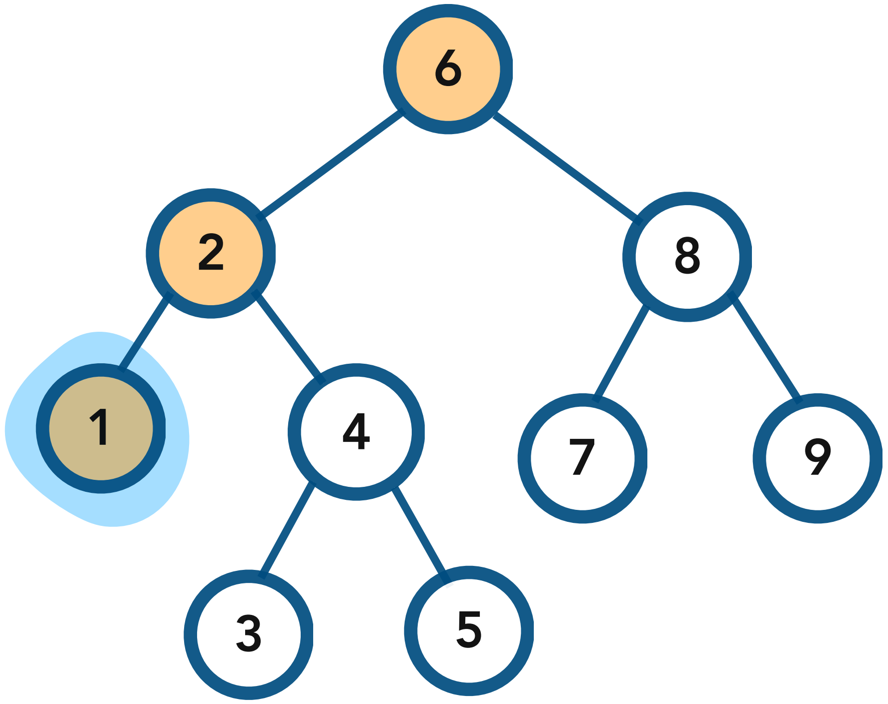
`stack frames`

Console

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print the tree rooted at 1

do not print 1 yet!

print left subtree first.

at (1)	left - current - right
at (2)	left - current - right
at (6)	left - current - right

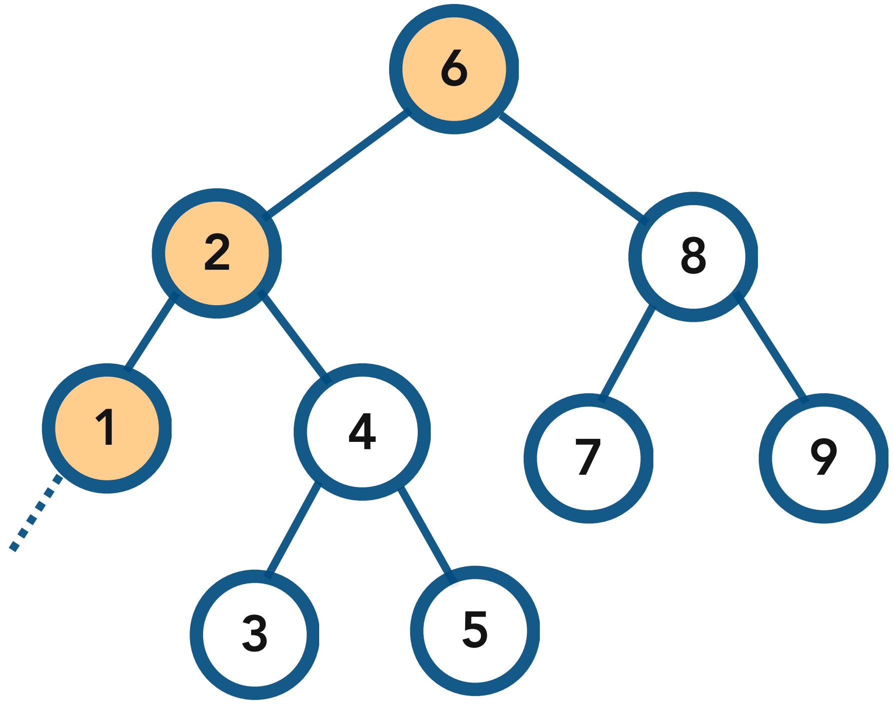
stack frames

Console

Printing the Tree (in order)

Idea. Print *all* of the left subtree and
then print the current node and
then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print the tree rooted at NULL

nothing to be done!

at (NULL)	do nothing!
at (1)	left - current - right
at (2)	left - current - right
at (6)	left - current - right

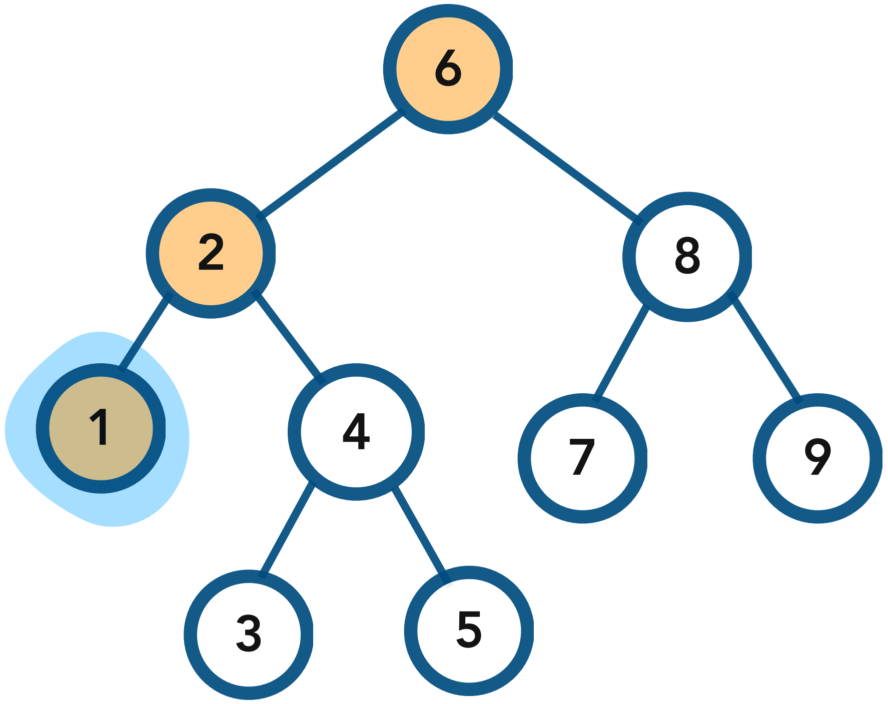
stack frames

Console

Printing the Tree (in order)

Idea. Print *all* of the left subtree and
then print the current node and
then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print the tree rooted at 1

ready to print 1

at (1)	left - current - right
at (2)	left - current - right
at (6)	left - current - right

stack frames

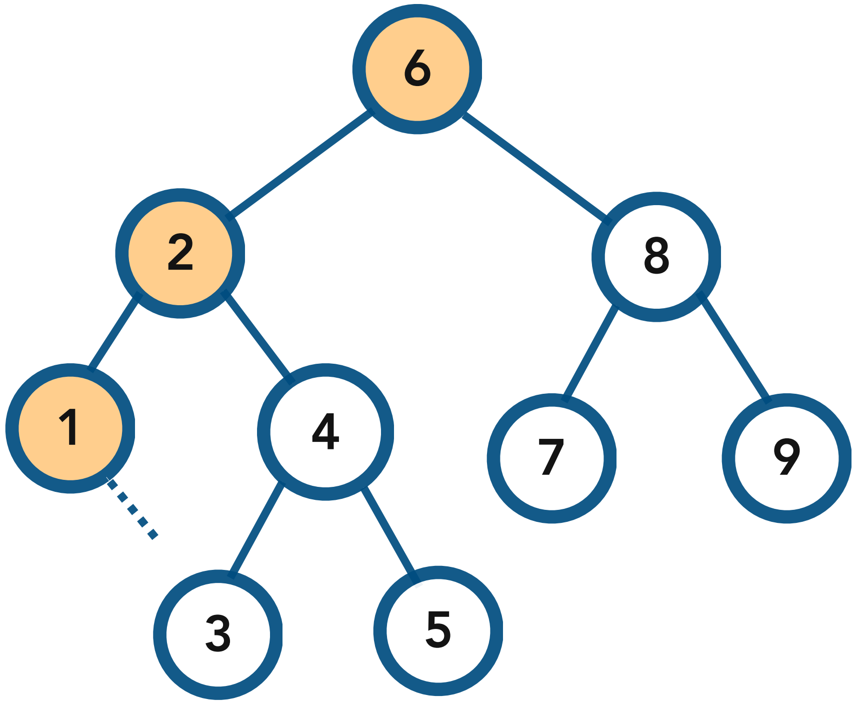
Console

```
1
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print the tree rooted at NULL
nothing to be done

at (NULL)	do nothing!
at (1)	left - current - right
at (2)	left - current - right
at (6)	left - current - right

stack frames

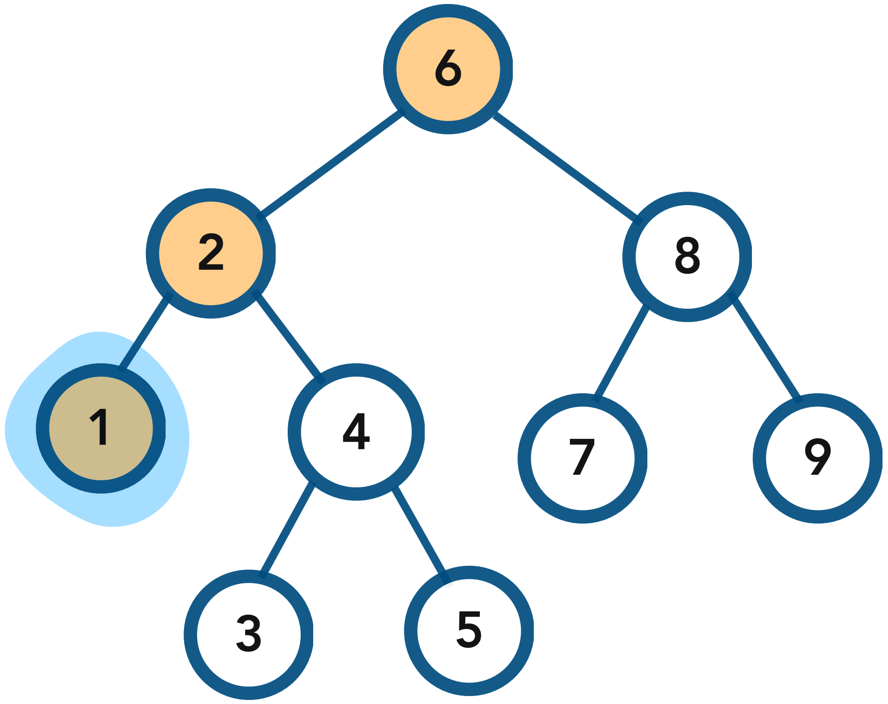
Console

```
1
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



done with tree rooted at 1

at (1) left - current - right

at (2) left - current - right

at (6) left - current - right

stack frames

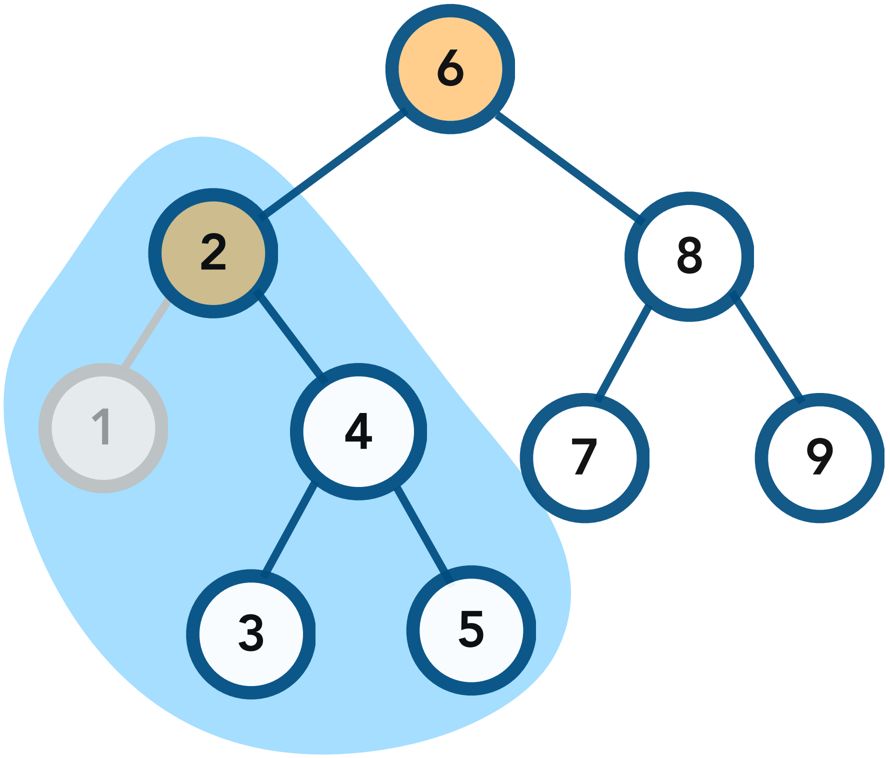
Console

```
1
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print the tree rooted at 2

ready to print 2

right subtree still needs to be printed.

```
at (2)  left - current - right
at (6)  left - current - right
```

stack frames

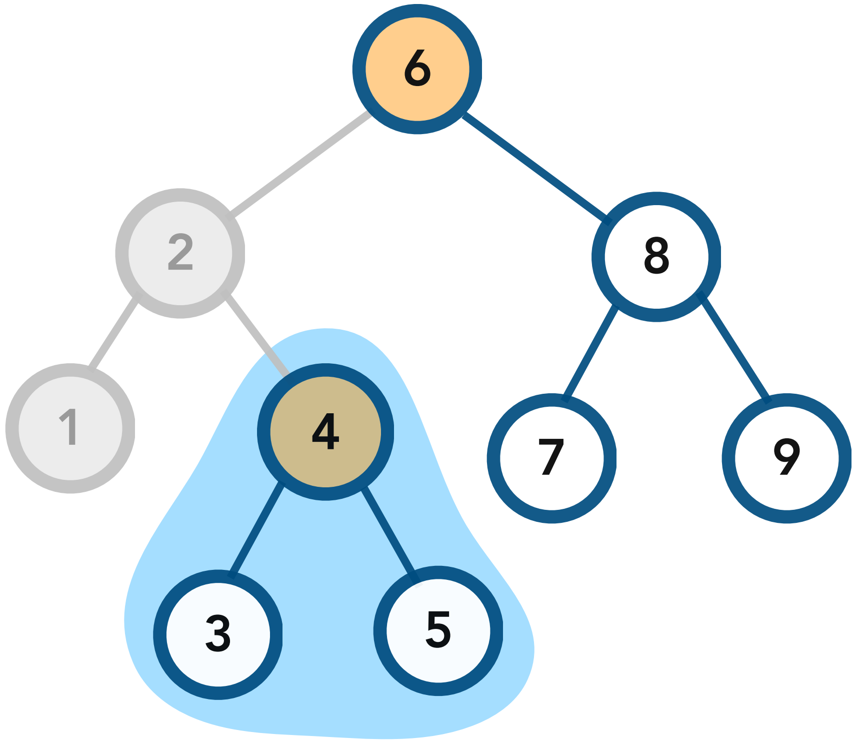
Console

```
1 2
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print tree rooted at 4

do not print 4 yet!

print left subtree first.

at (4)	left - current - right
at (2)	left - current - right
at (6)	left - current - right

stack frames

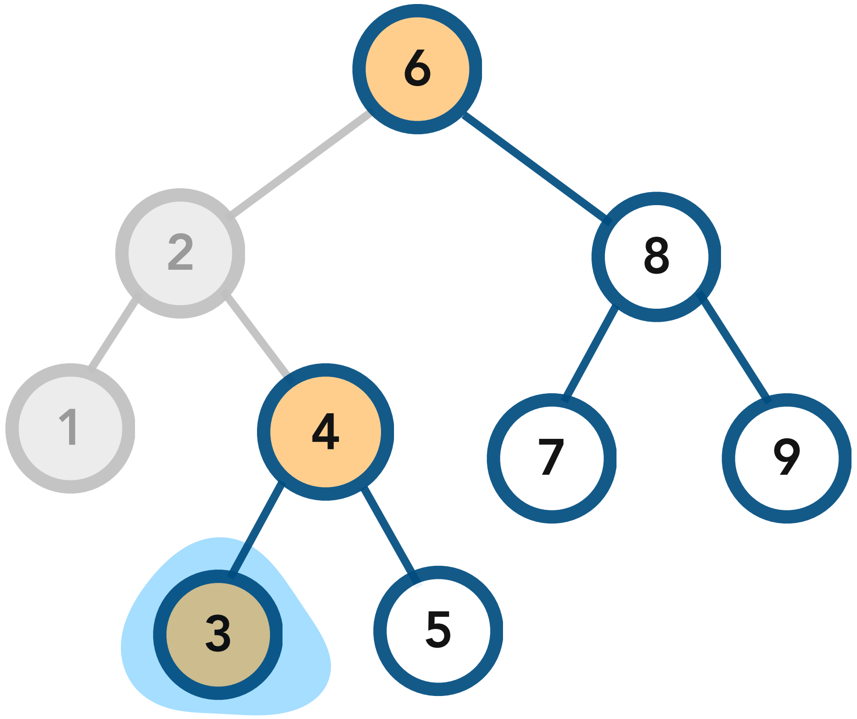
Console

```
1 2
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and
then print the current node and
then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print tree rooted at 3

ready to print 3 after going left

at (3) left - current - right

at (4) left - current - right

at (2) left - current - right

at (6) left - current - right

stack frames

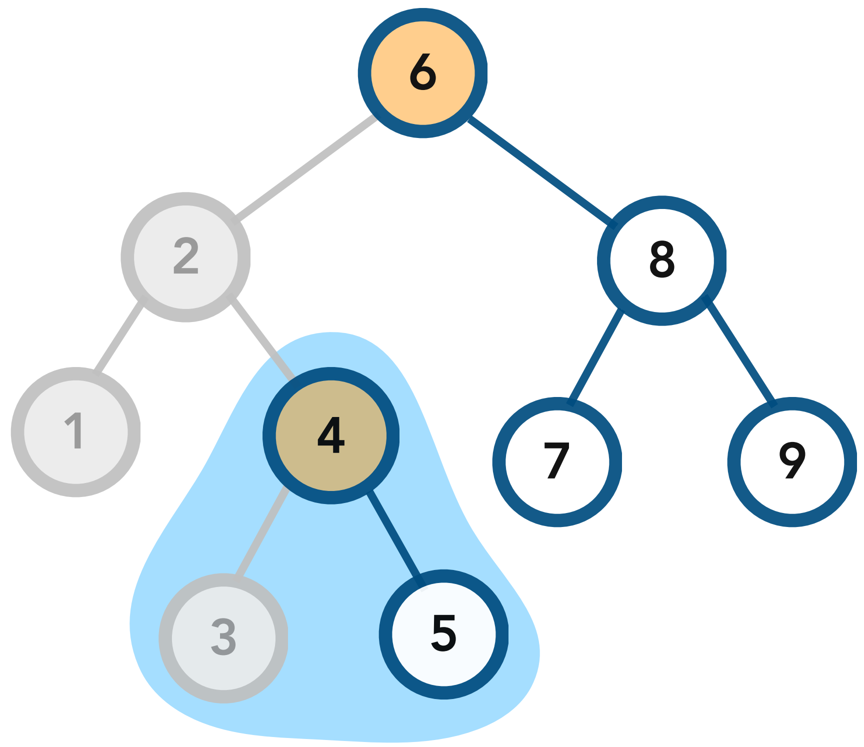
Console

```
1 2 3
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print tree rooted at 4

ready to print 4

right subtree still needs to be printed.

at (4)	left - current - right
at (2)	left - current - right
at (6)	left - current - right

stack frames

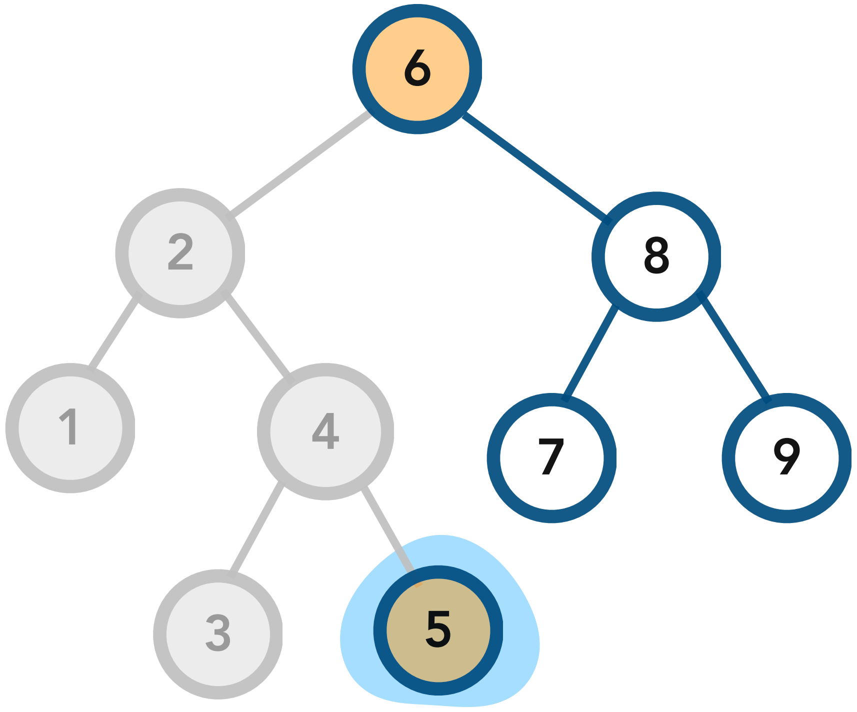
Console

```
1 2 3 4
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print tree rooted at 5

ready to print 5 after going left

at (5)	left - current - right
at (4)	left - current - right
at (2)	left - current - right
at (6)	left - current - right

stack frames

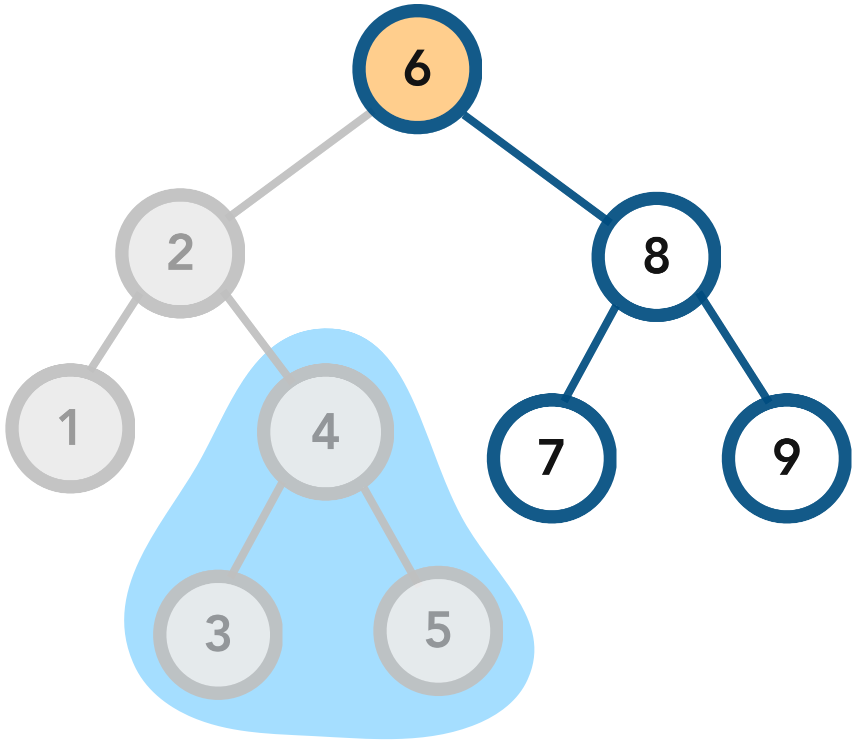
Console

```
1 2 3 4 5
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



done with tree rooted at 4

at (4) left - current - right

at (2) left - current - right

at (6) left - current - right

stack frames

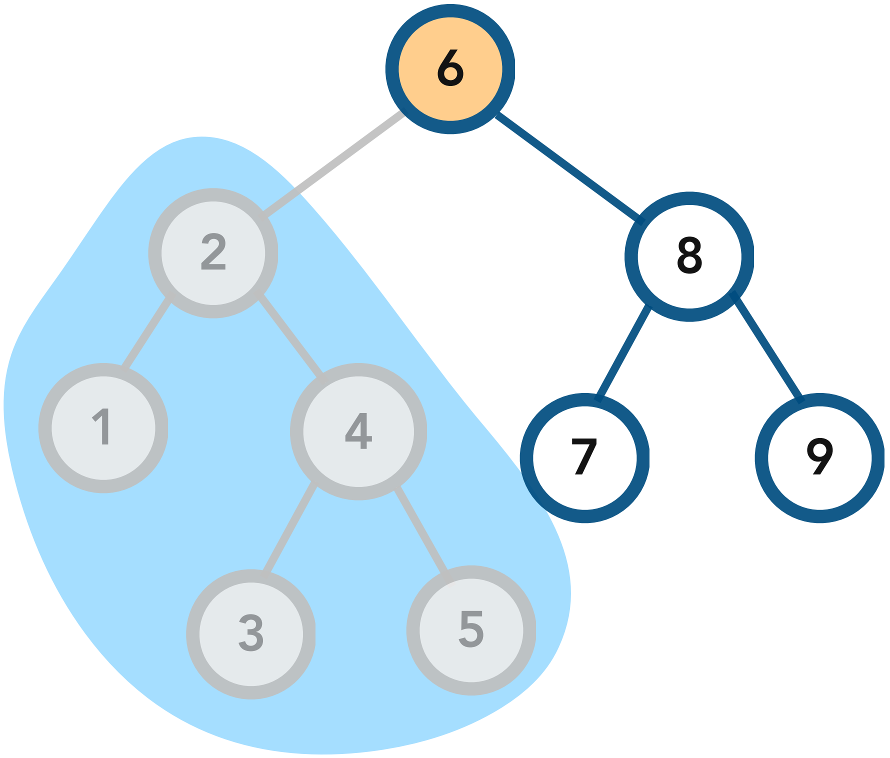
Console

```
1 2 3 4 5
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



done with tree rooted at 2

```
at (2) left - current - right
at (6) left - current - right
```

stack frames

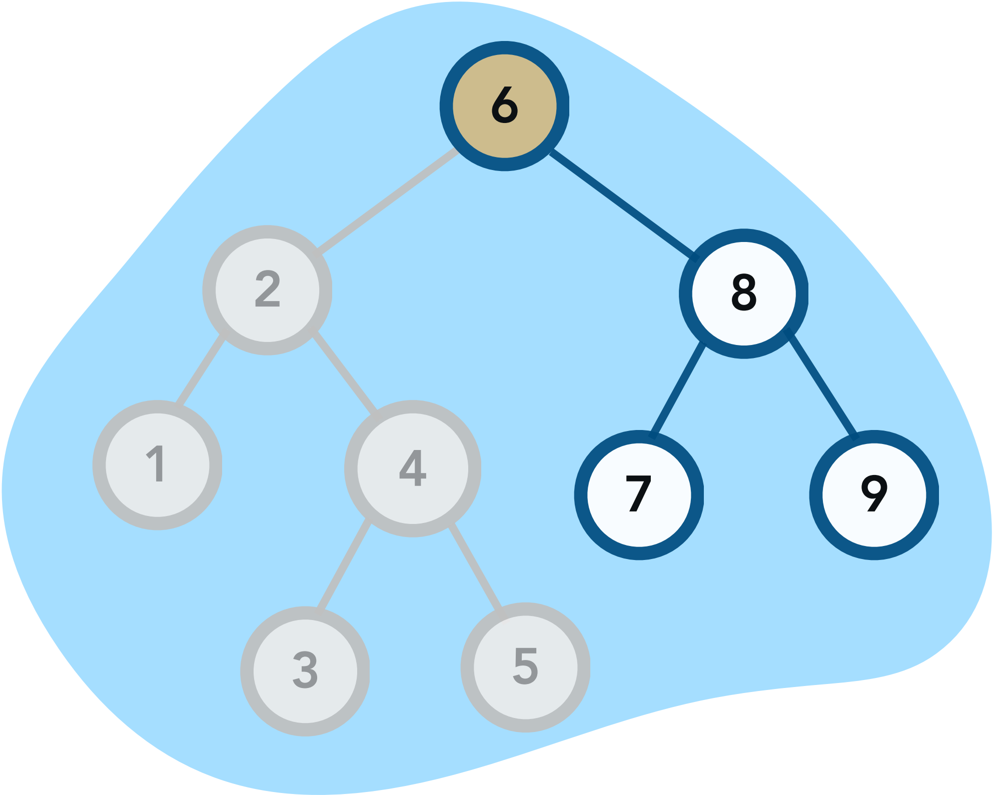
Console

```
1 2 3 4 5
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print tree rooted at 6

ready to print 6

right subtree still needs to be printed.

```
at (6) left - current - right
```

stack frames

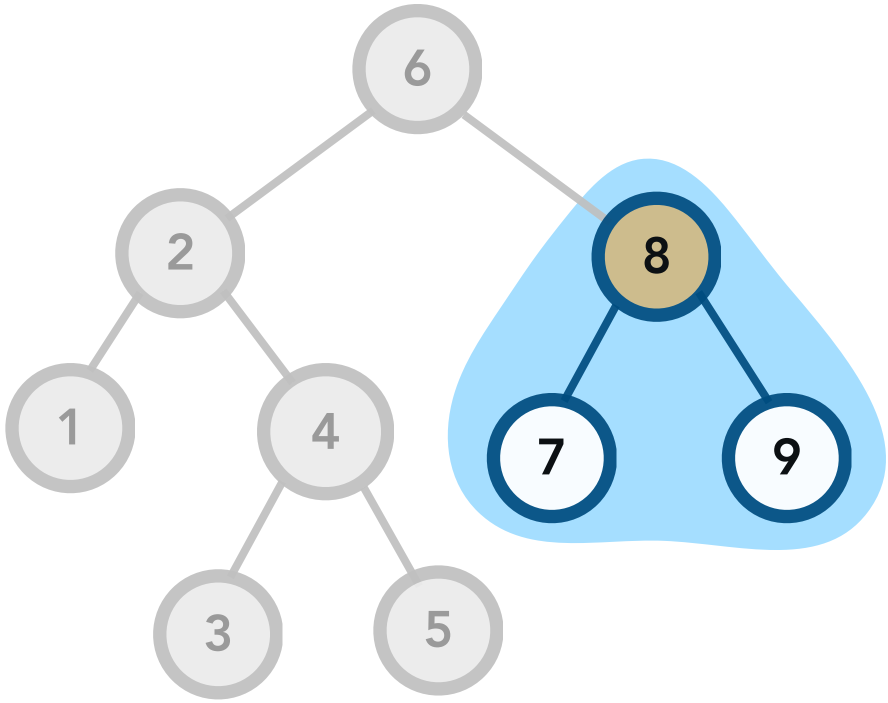
Console

```
1 2 3 4 5 6
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print tree rooted at 8

do not print 8 yet!

print left subtree first.

```
at (8)  left - current - right
at (6)  left - current - right
```

stack frames

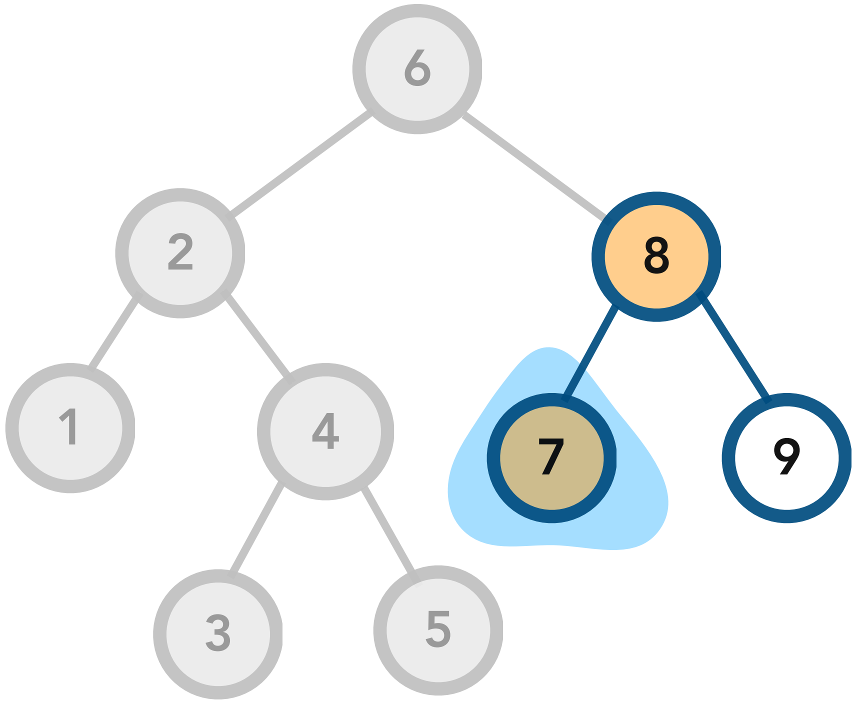
Console

```
1 2 3 4 5 6
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print tree rooted at 7

ready to print 7 after going left

at (7)	left - current - right
at (8)	left - current - right
at (6)	left - current - right

stack frames

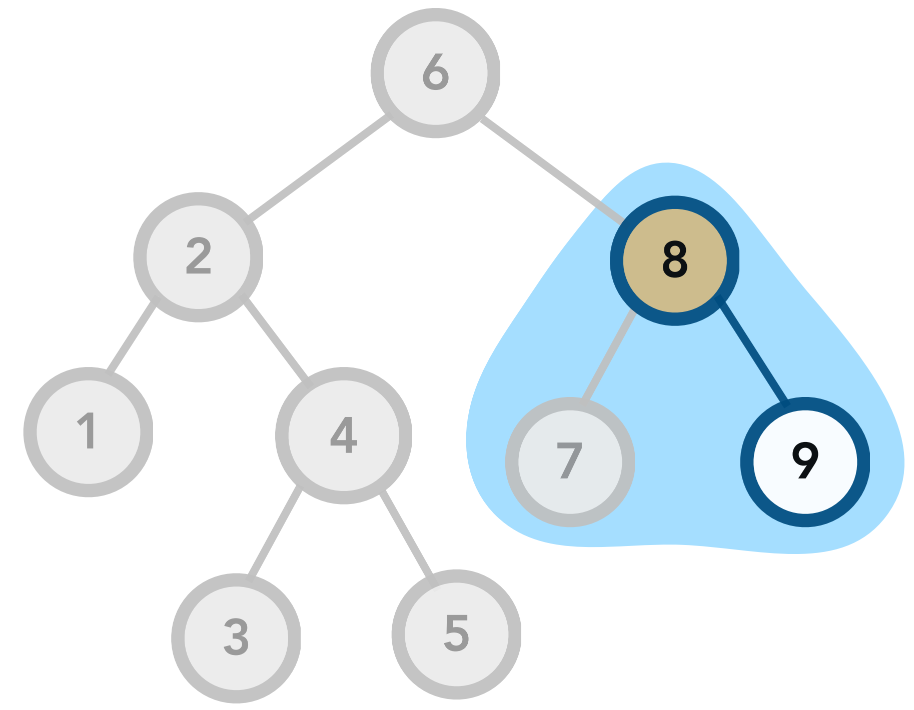
Console

```
1 2 3 4 5 6 7
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print tree rooted at 8

ready to print 8

right subtree still needs to be printed

at (8) left - current - right

at (6) left - current - right

stack frames

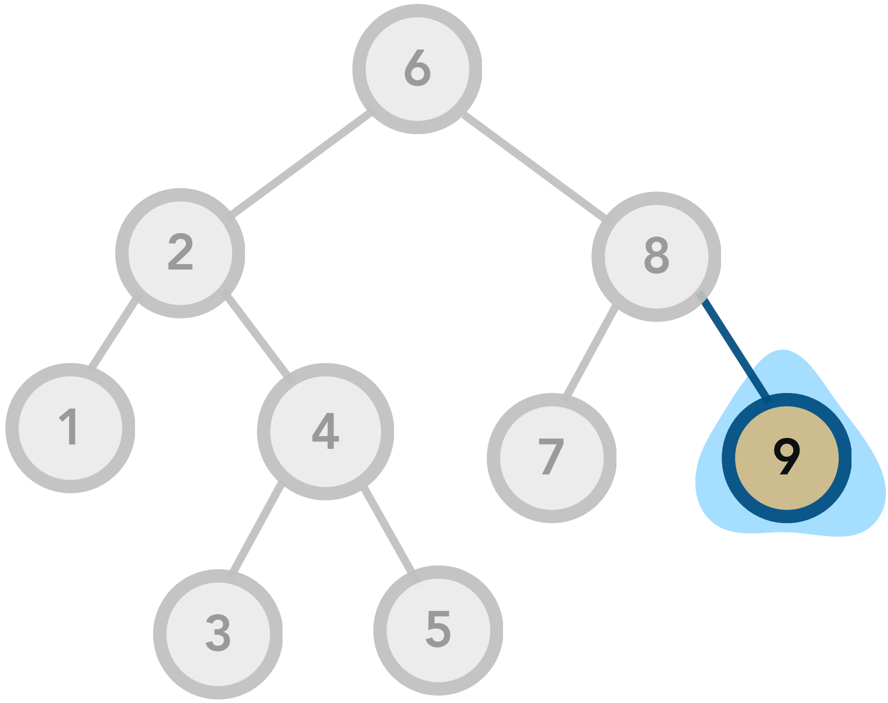
Console

```
1 2 3 4 5 6 7 8
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



print tree rooted at 9

ready to print 9 after going left

at (9)	left - current - right
at (8)	left - current - right
at (6)	left - current - right

stack frames

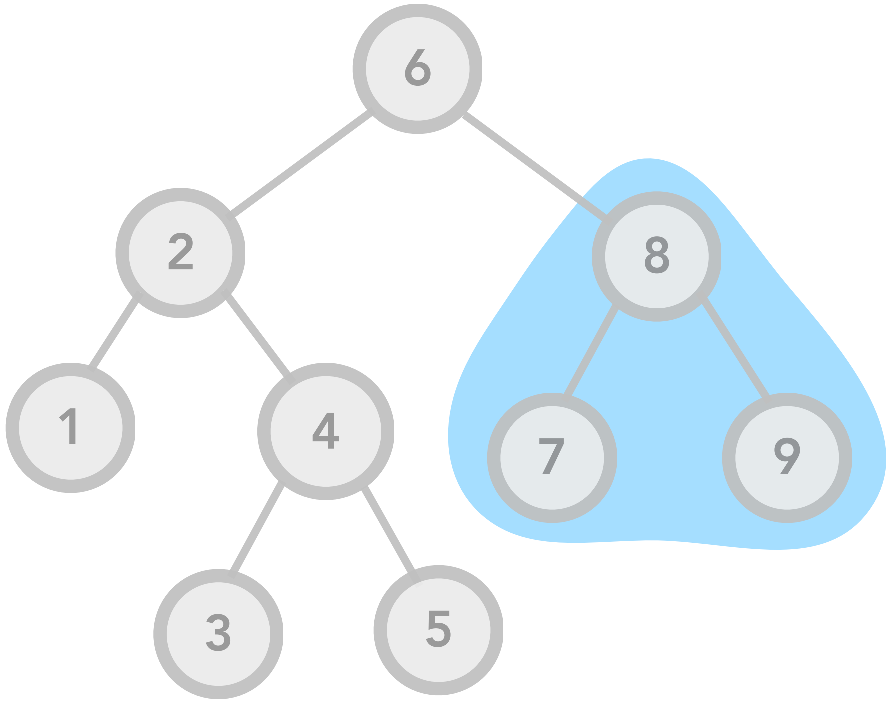
Console

```
1 2 3 4 5 6 7 8 9
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
    print the current node, then
    print the larger nodes



done with tree rooted at 8

at (8)	left - current - right
at (6)	left - current - right

stack frames

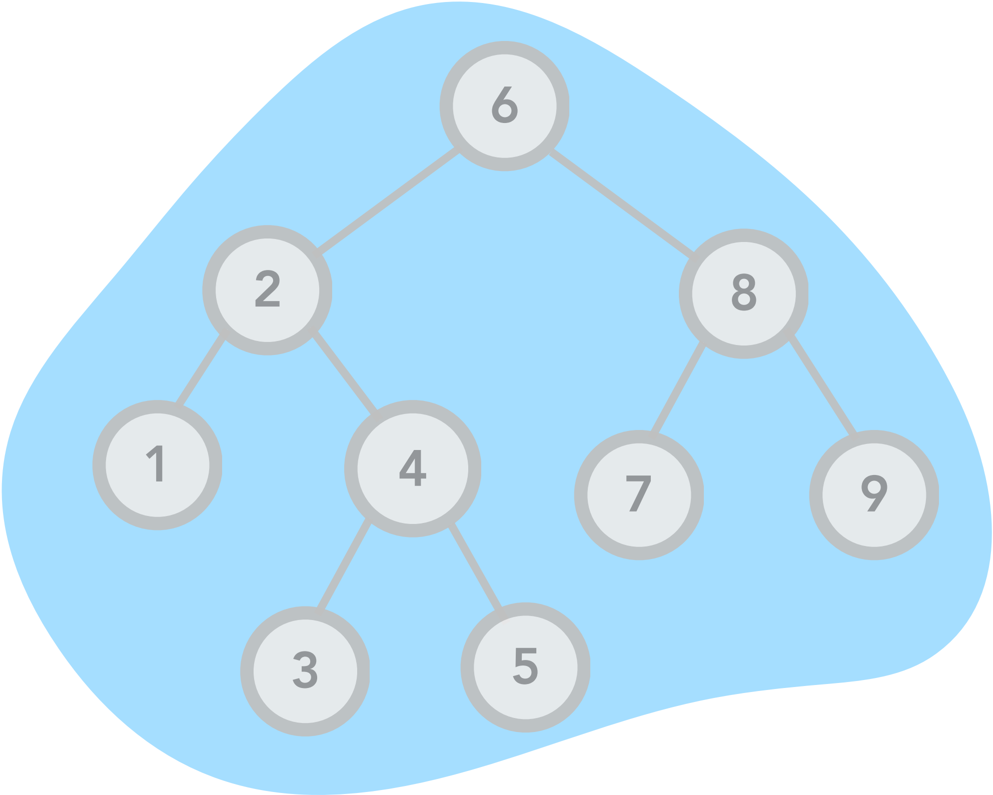
Console

```
1 2 3 4 5 6 7 8 9
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



done with tree rooted at 6

Console

```
1 2 3 4 5 6 7 8 9
```

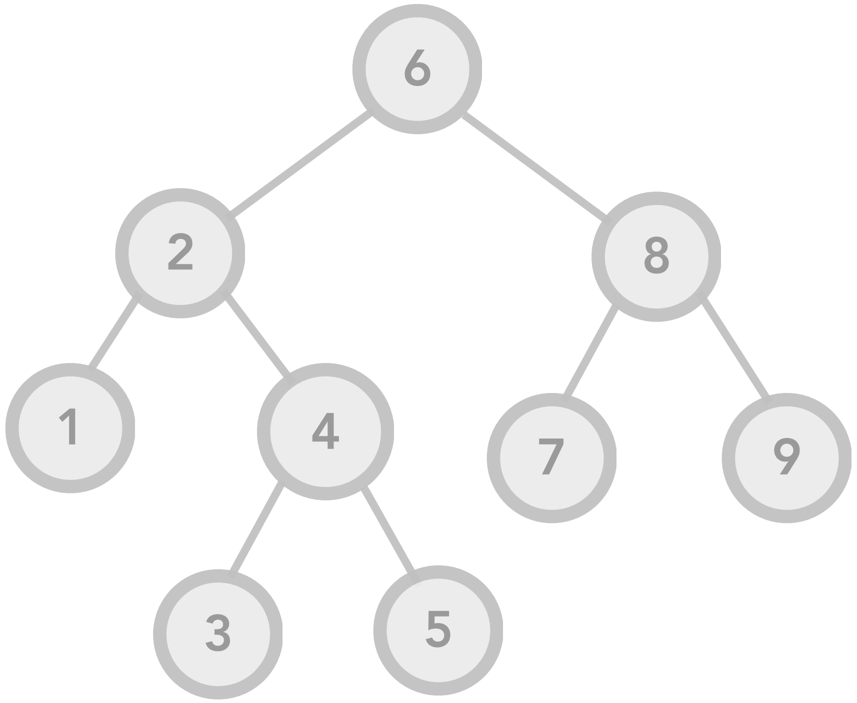
at (6) left - current - right

stack frames

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes



done!

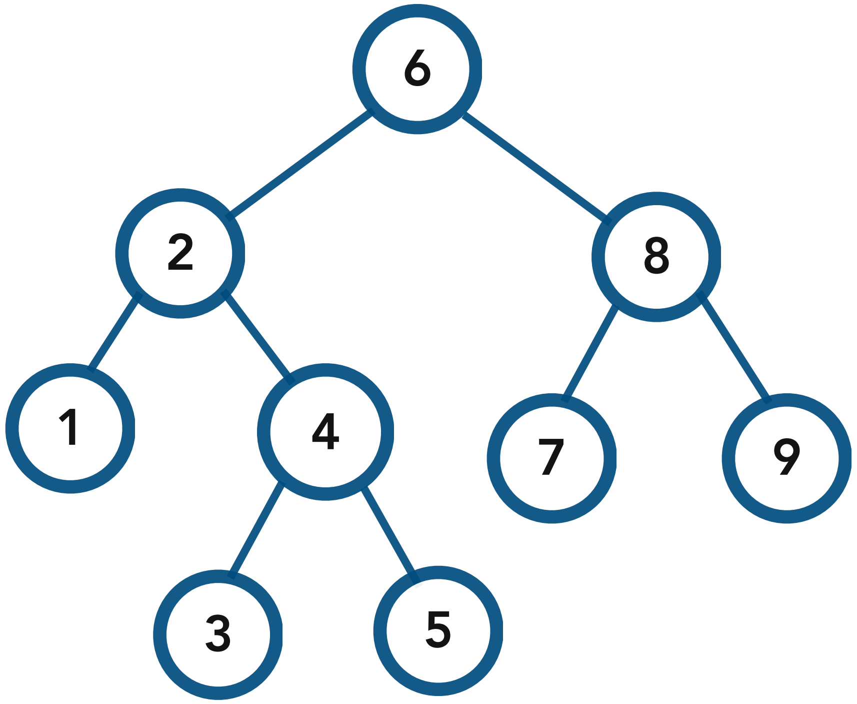
Console

```
1 2 3 4 5 6 7 8 9
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
print the current node, then
print the larger nodes

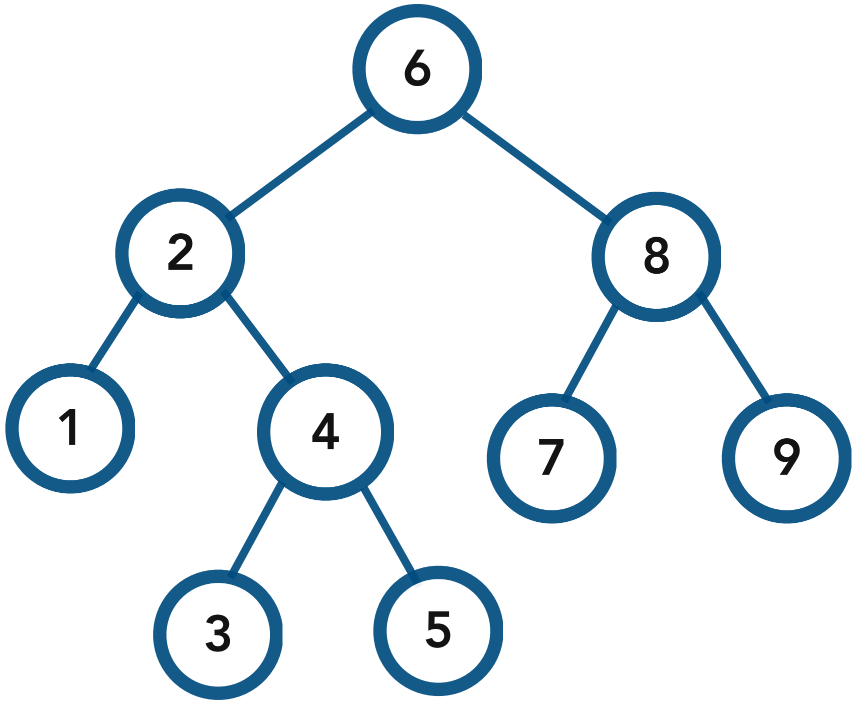


```
template <class T>
void BST<T>::print_in_order() const {
    print_in_order(root);
}

template <class T>
void BST<T>::print_in_order(Node<T>* node) const {
    if (node == nullptr) return;

    print_in_order(node->left);
    cout << node->val << " ";
    print_in_order(node->right);
}
```

Printing the Tree (in order)

Idea. Print *all* of the left subtree and then print the current node and then print *all* of the right subtree.

I.e. Print the smaller nodes then
  print the current node, then
  print the larger nodes



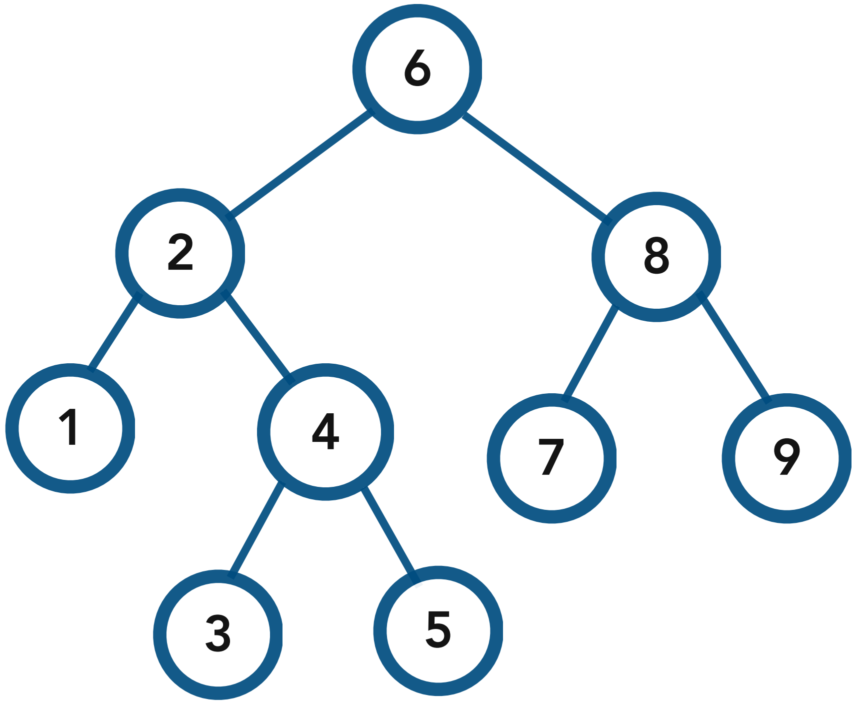
```cpp
template <class T>
void BST<T>::print_in_order() const {
    print_in_order(root);
}
```

public function used by the user

```cpp
template <class T>
void BST<T>::print_in_order(Node<T>* node) const {
    if (node == nullptr) return;

    print_in_order(node->left);
    cout << node->val << " ";
    print_in_order(node->right);
}
```

private helper (recursive) function

Clearing the Tree

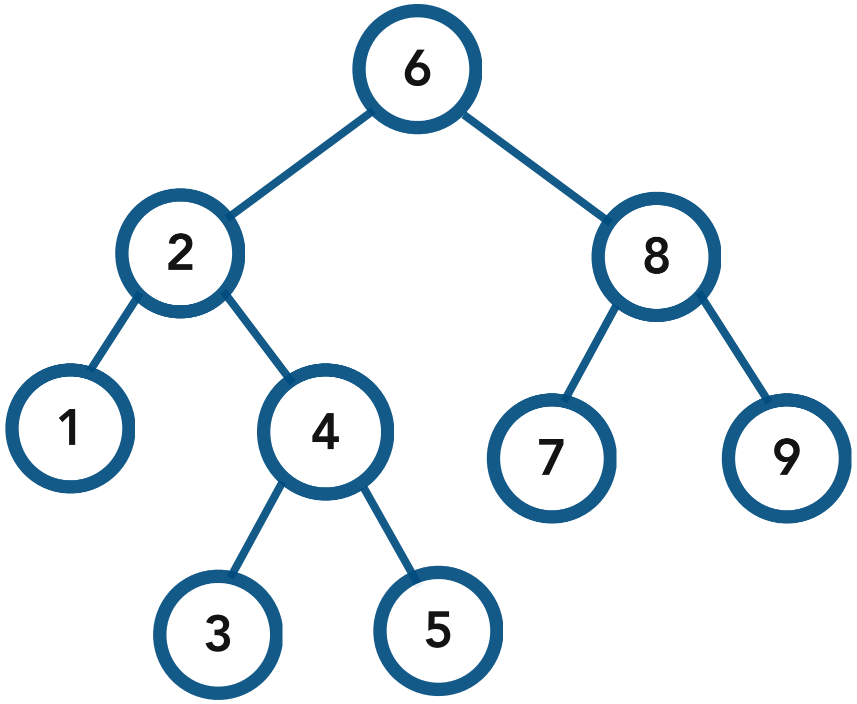


How can we traverse the tree and *delete* every node?

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
Clear the right subtree, then
delete the current node

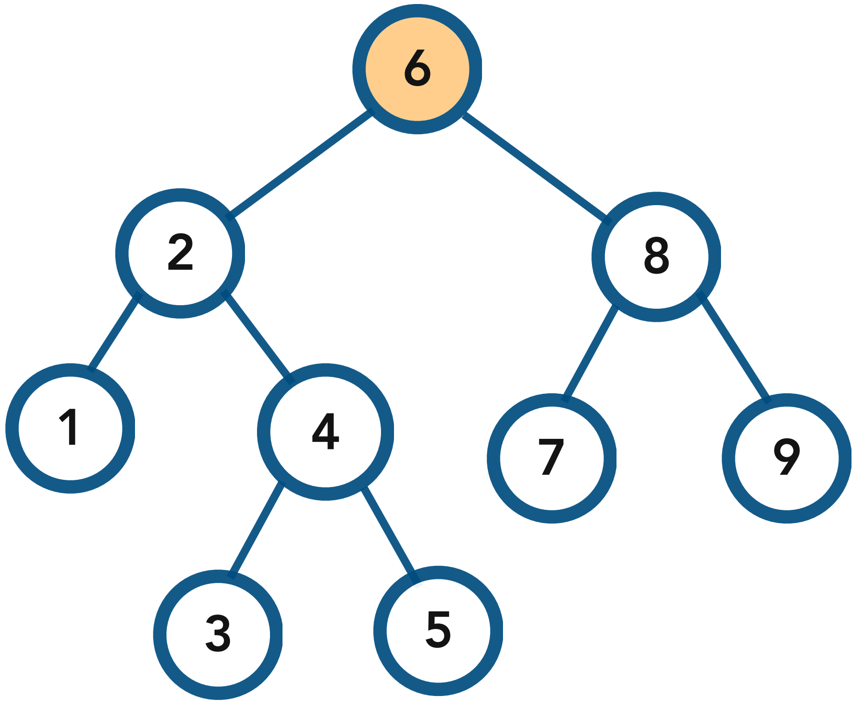


How can we traverse the tree and *delete* every node?

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
Clear the right subtree, then
delete the current node

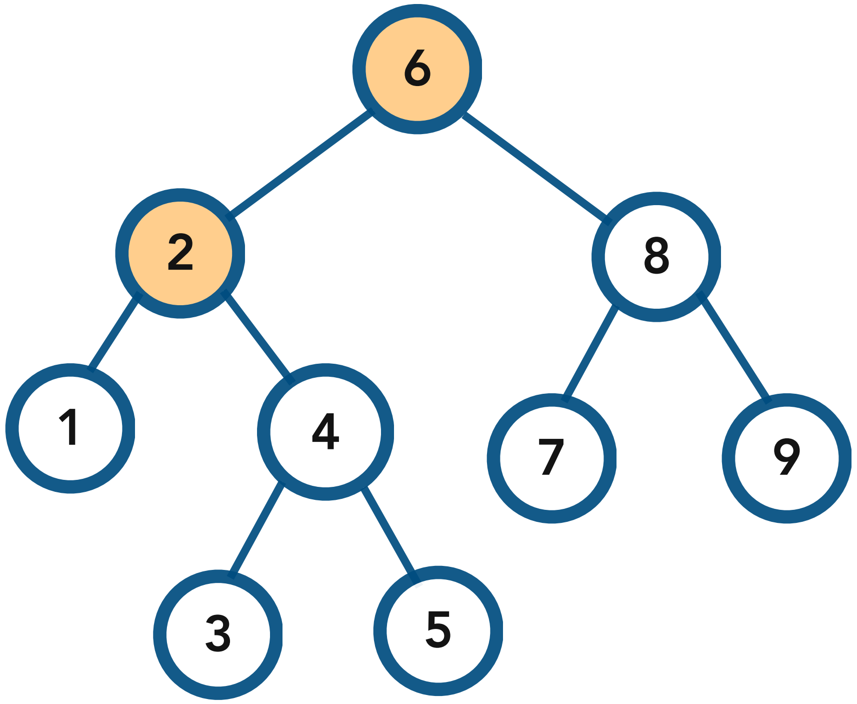


To clear the tree rooted at 6,
2 and 8 have to be cleared first

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
Clear the right subtree, then
delete the current node

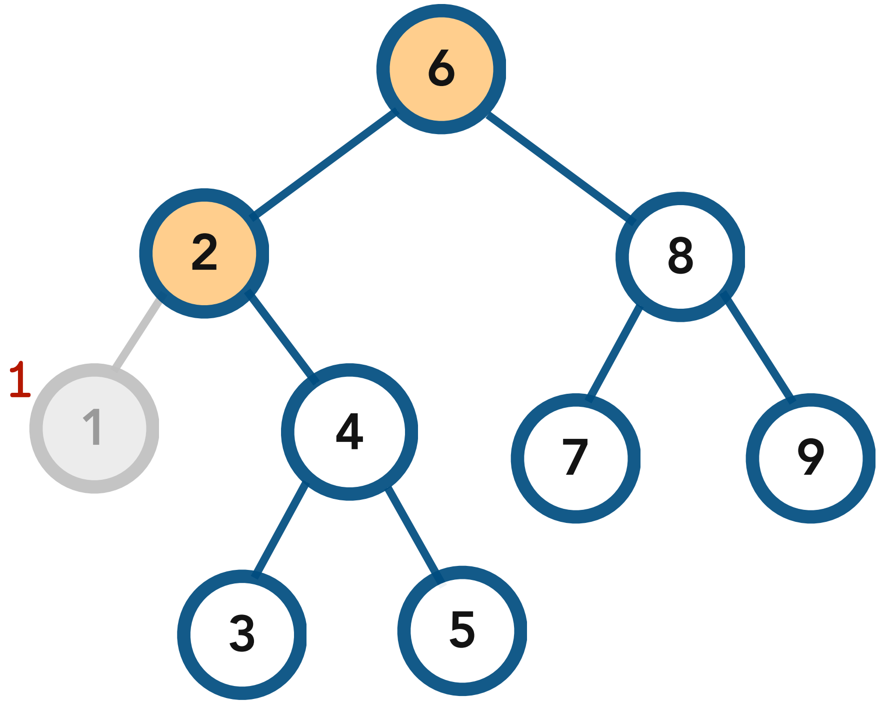


To clear the tree rooted at 2,
1 and 4 have to be cleared first

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
Clear the right subtree, then
delete the current node

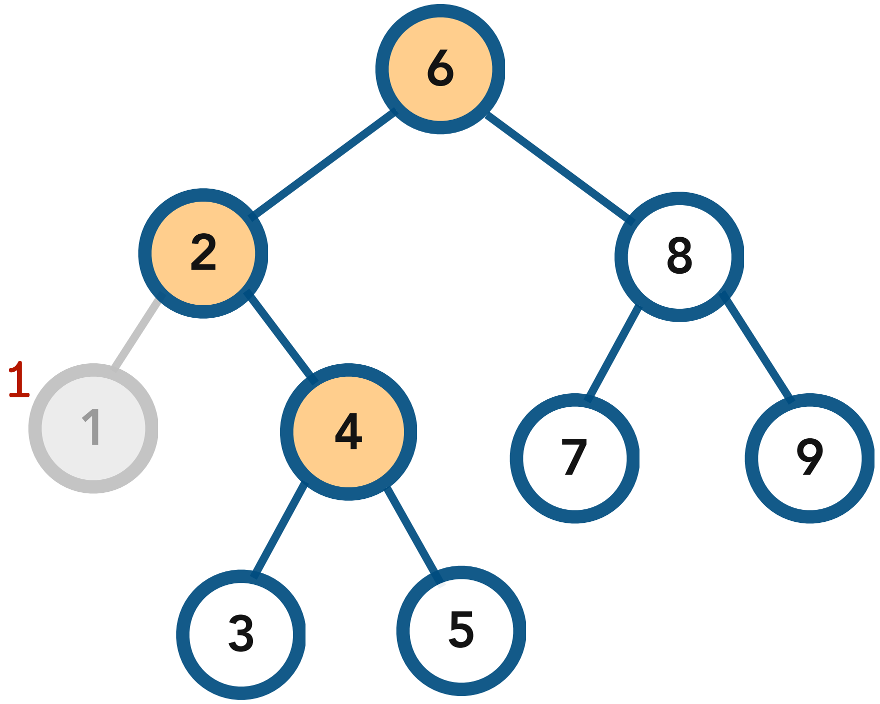


1 can be cleared!

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
Clear the right subtree, then
delete the current node

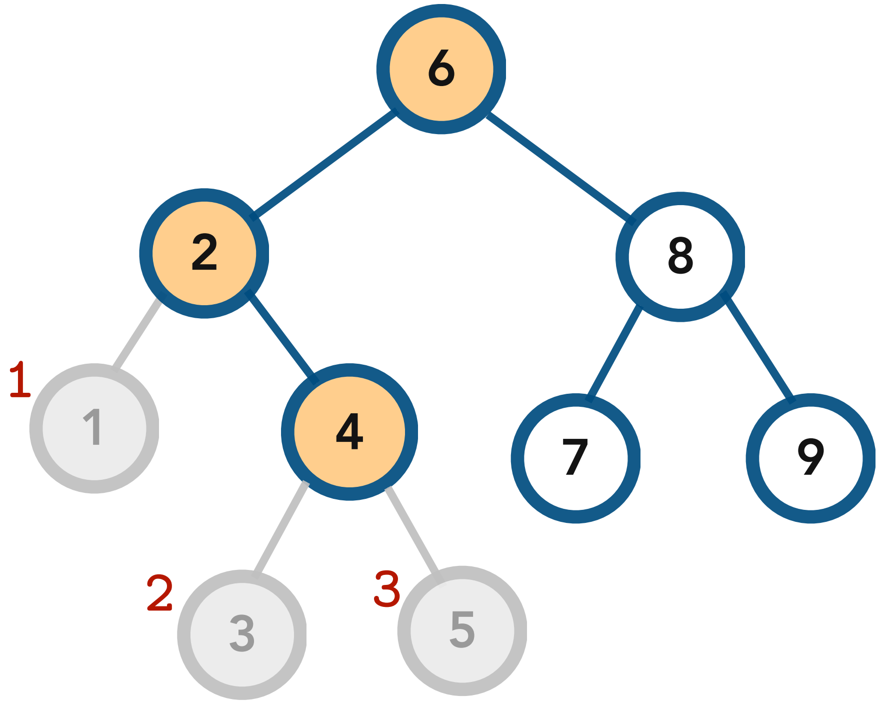


To clear the tree rooted at 4,
3 and 5 have to be cleared first

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
    Clear the right subtree, then
    delete the current node

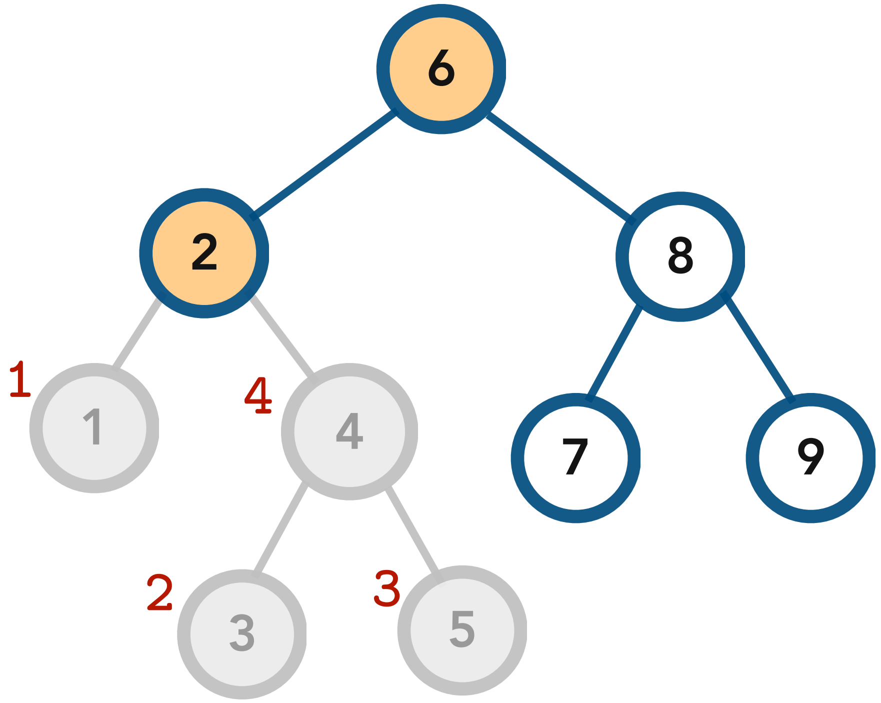


1 and 5 can be cleared!

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
Clear the right subtree, then
delete the current node

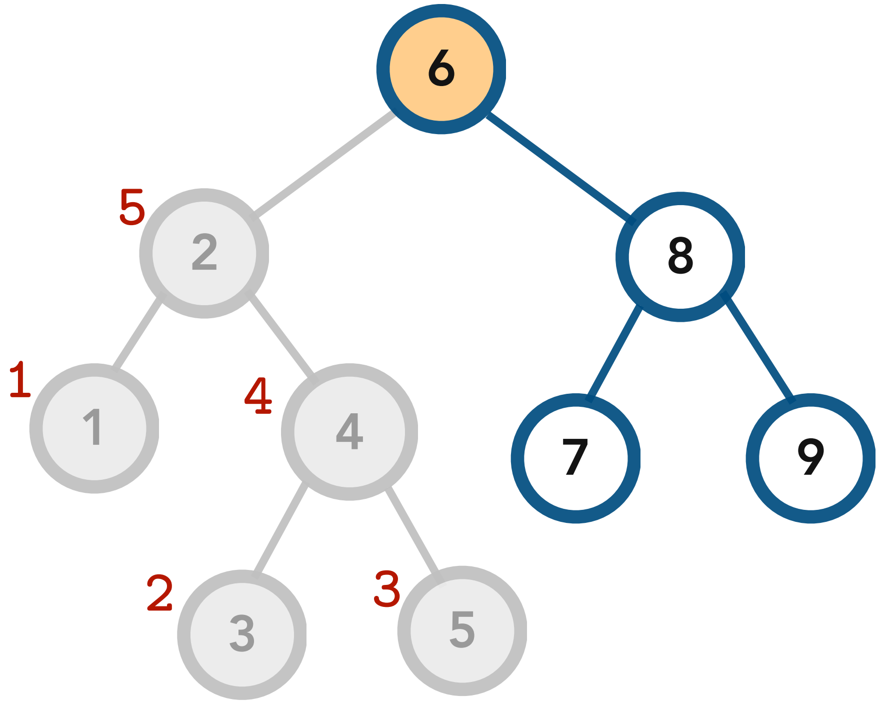


4 can be cleared!

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
Clear the right subtree, then
delete the current node

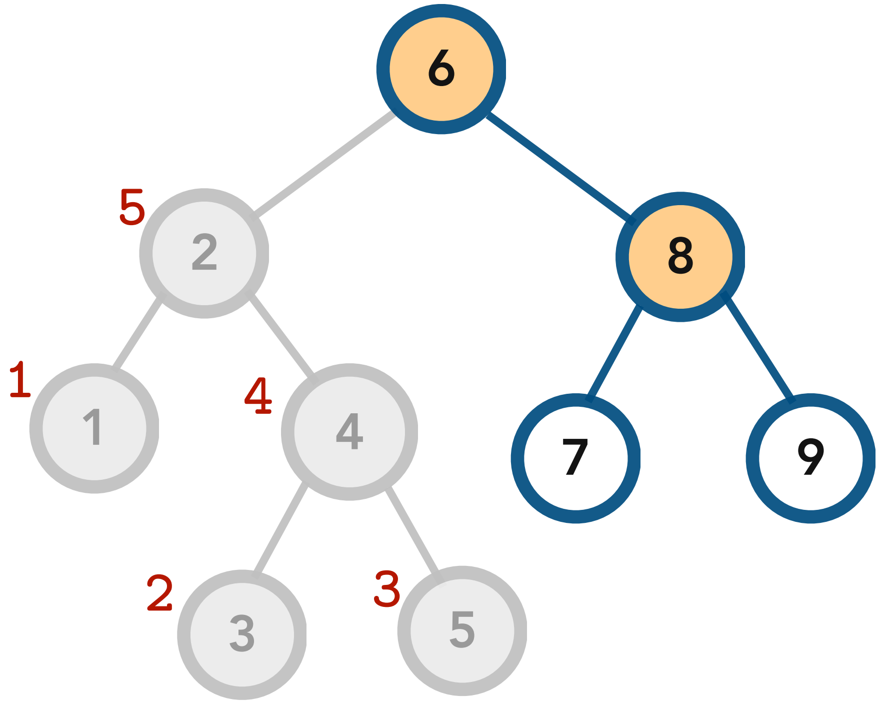


2 can be cleared!

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
Clear the right subtree, then
delete the current node

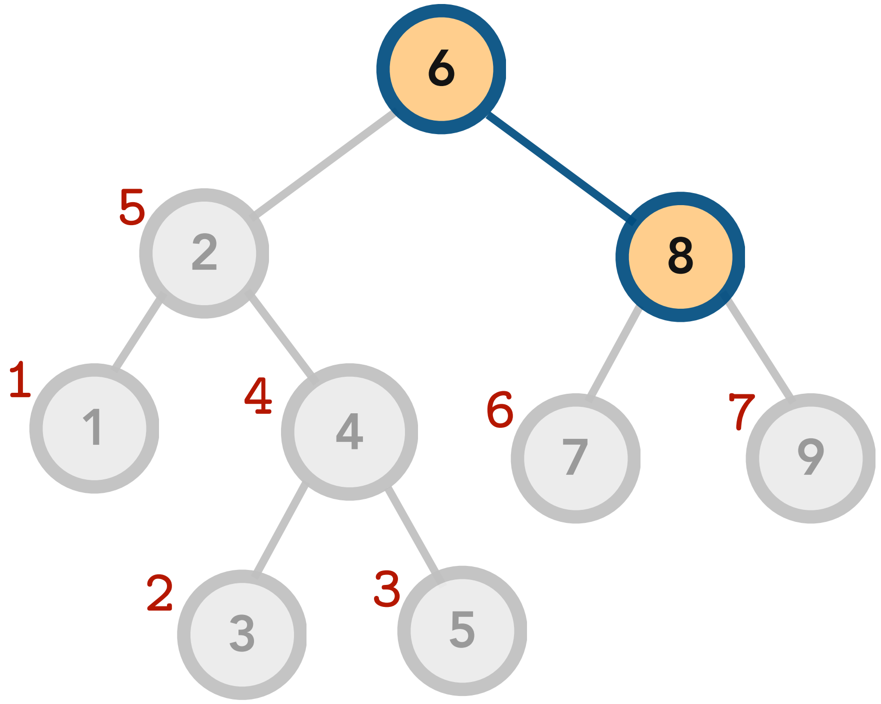


To clear the tree rooted at 8,
7 and 9 have to be cleared first

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
Clear the right subtree, then
delete the current node

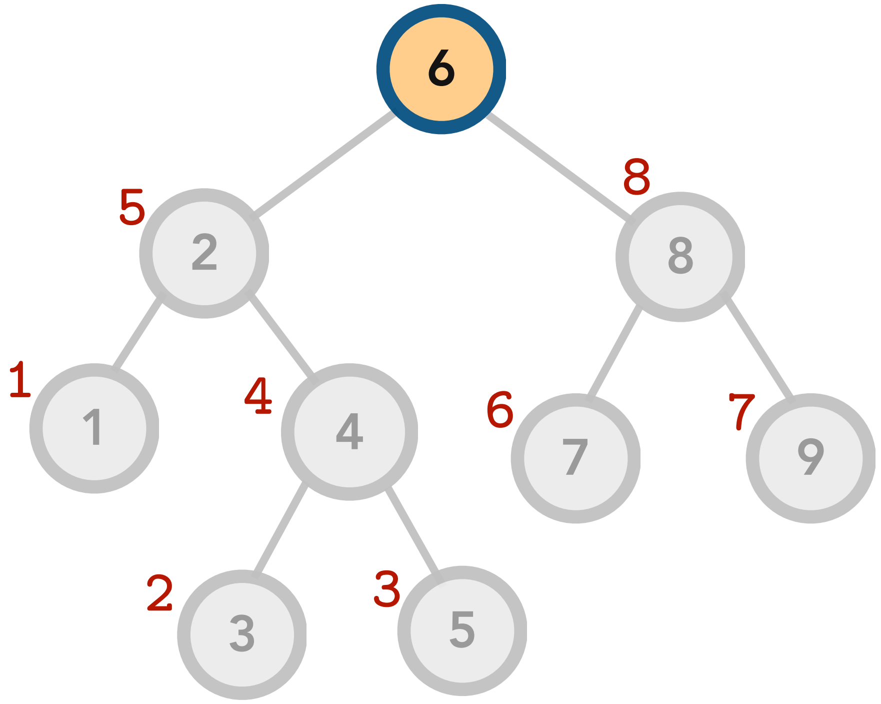


7 and 9 can be cleared!

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
Clear the right subtree, then
delete the current node

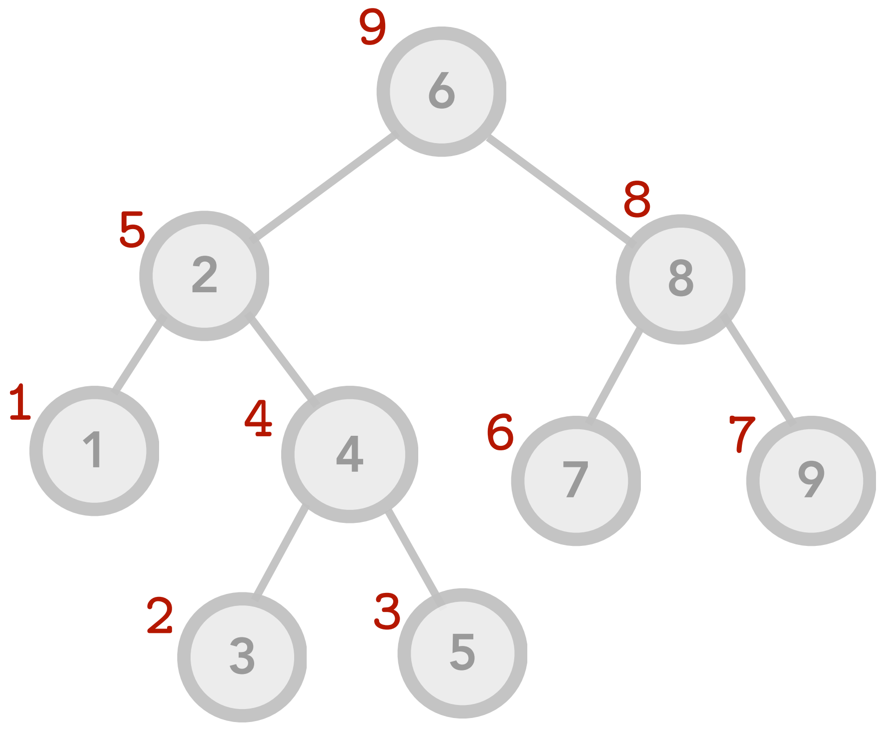


8 can be cleared!

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
    Clear the right subtree, then
    delete the current node

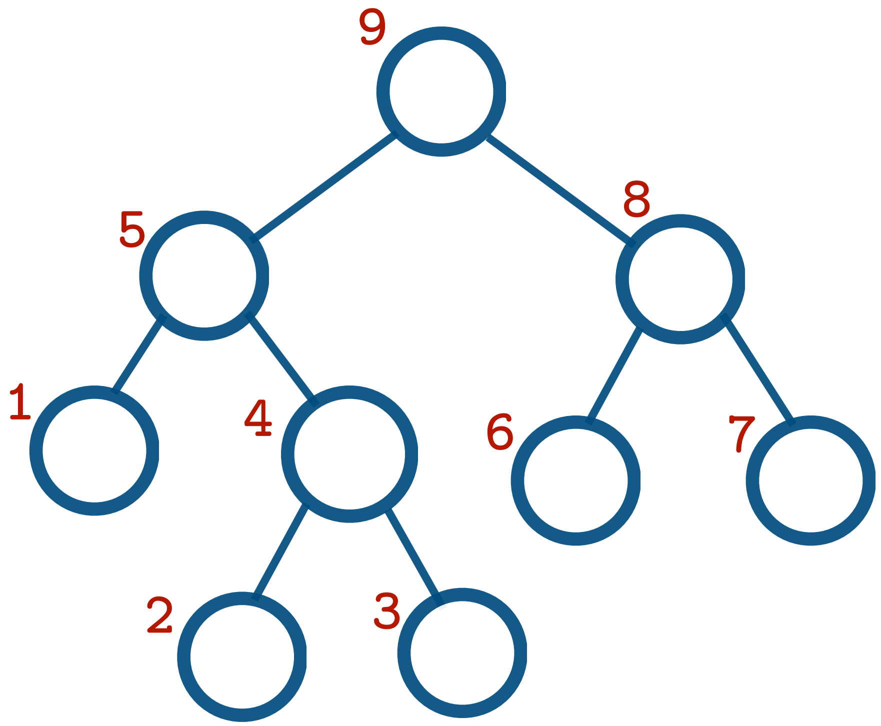


6 can be cleared!

Clearing the Tree

Idea. Since the children of a node are accessible only through the node, *do not delete the node until its children have been deleted*.

I.e. Clear the left subtree
    Clear the right subtree, then
    delete the current node

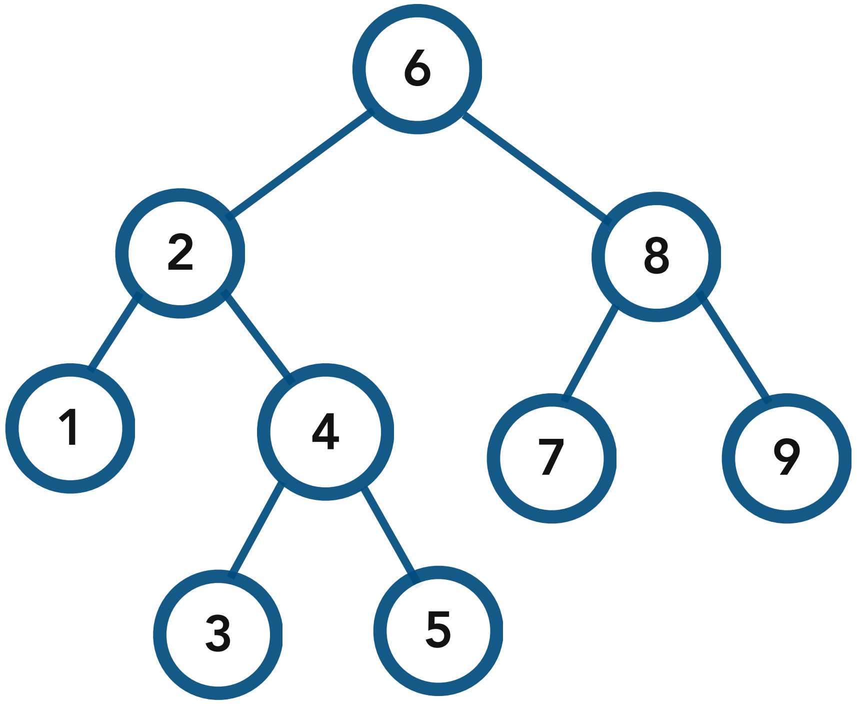


```
template <class T>
void BST<T>::clear() {
    clear(root);
    root = nullptr;
}

template <class T>
void BST<T>::clear(Node<T>* node) {
    if (node == nullptr) return;

    clear(node->left);
    clear(node->right);
    delete node;
}
```

Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

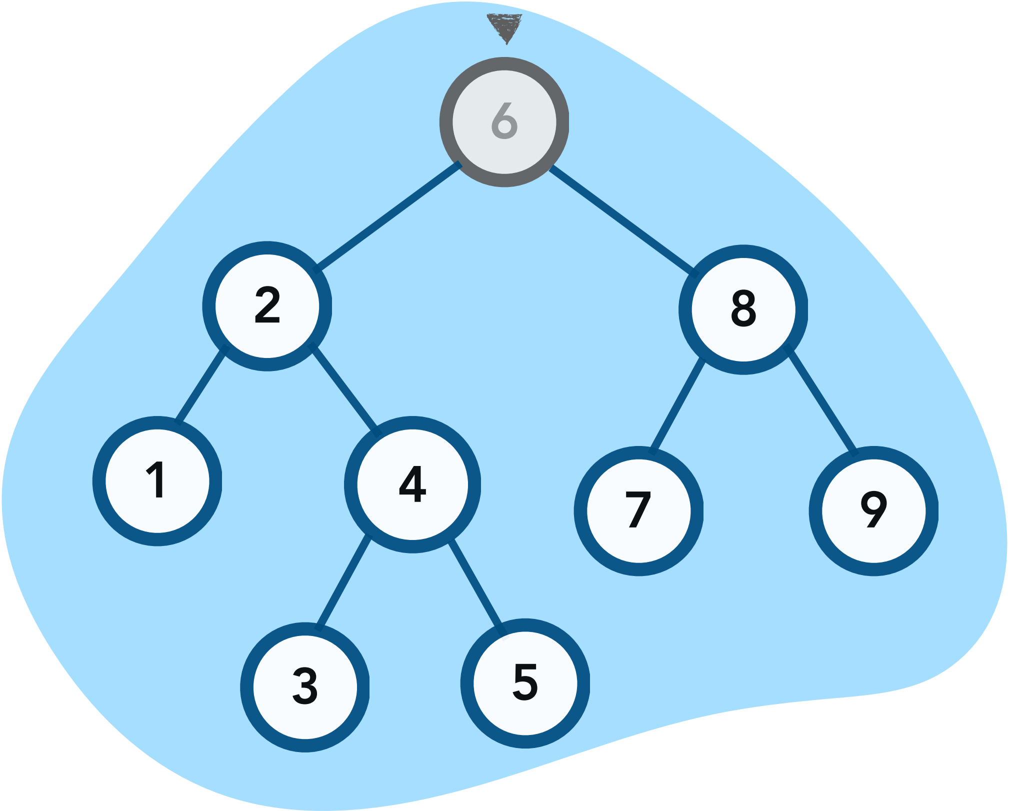
Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

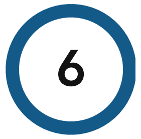


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

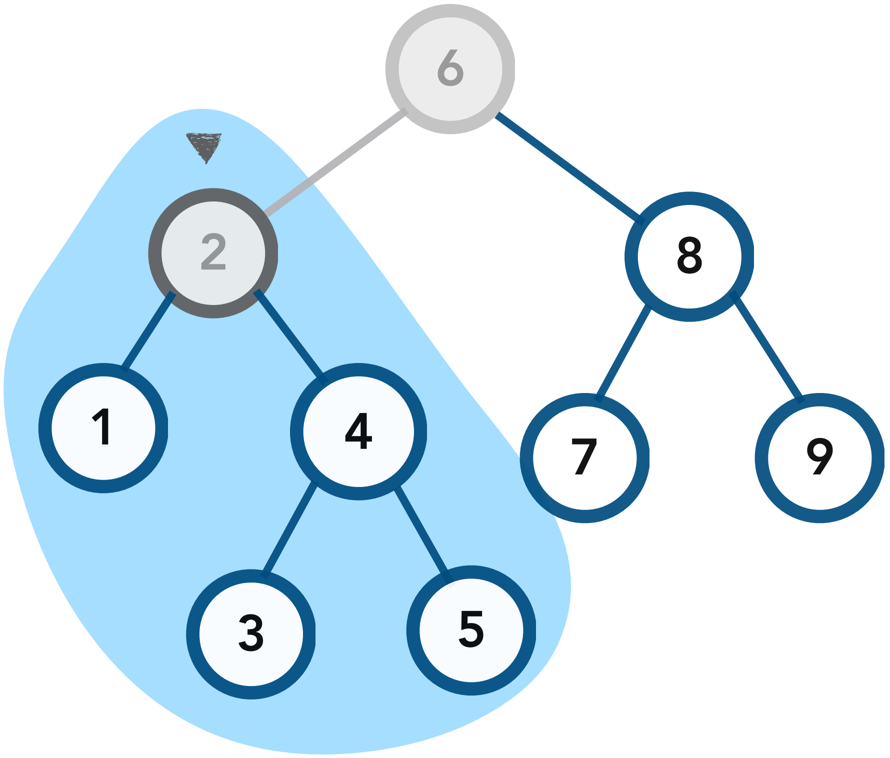


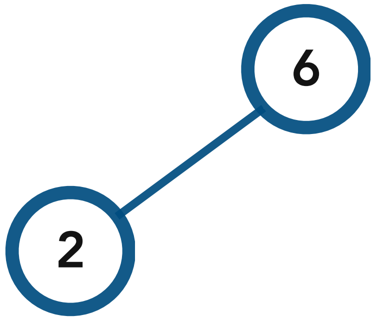


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

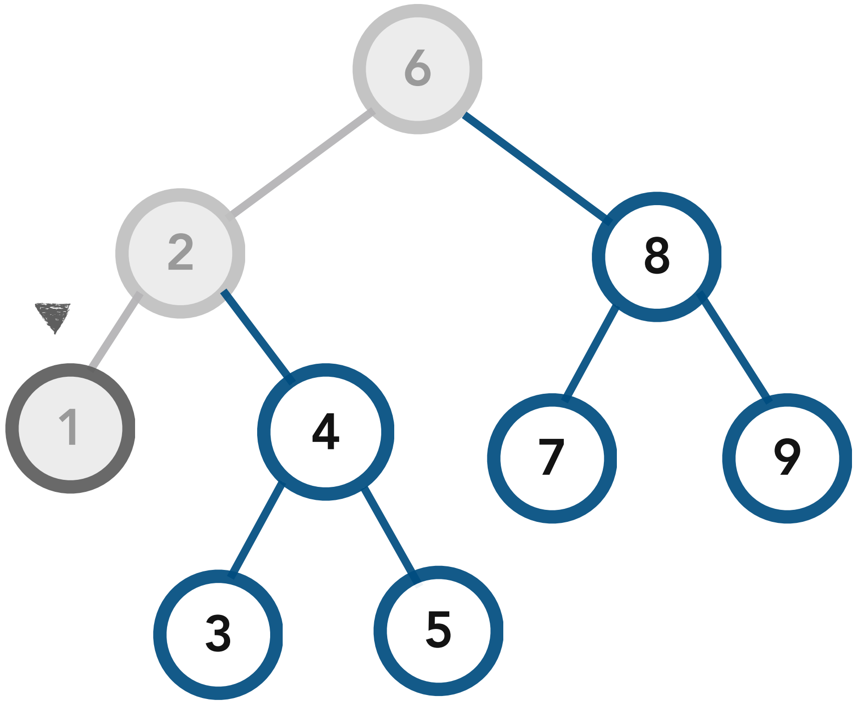


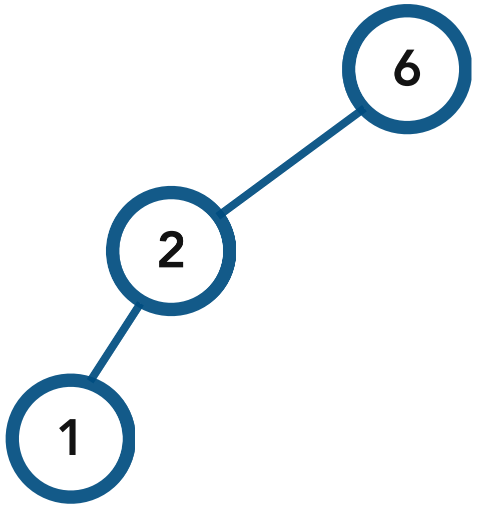


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

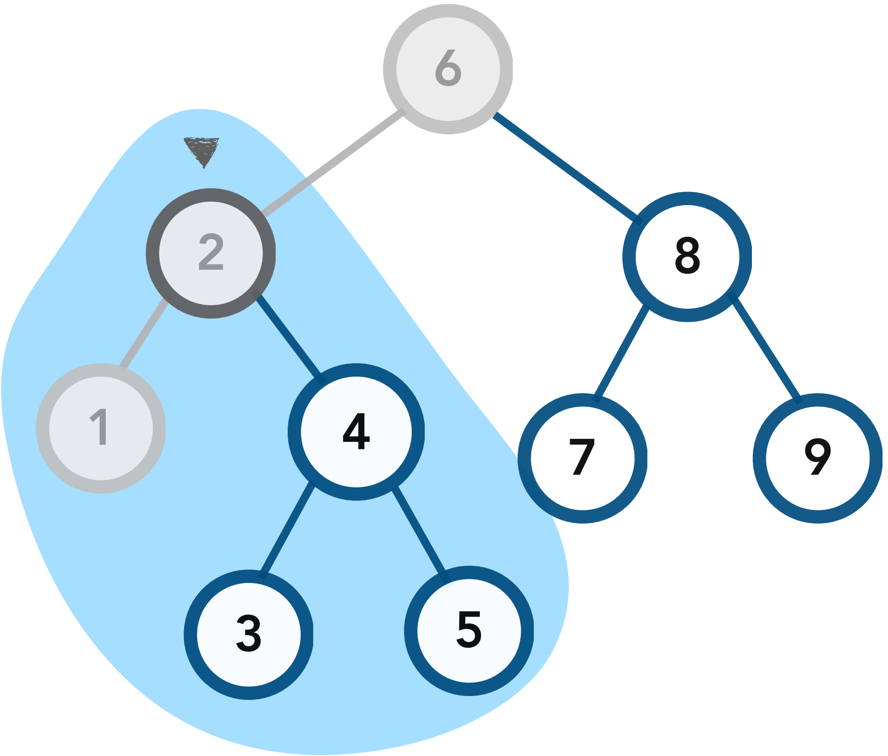


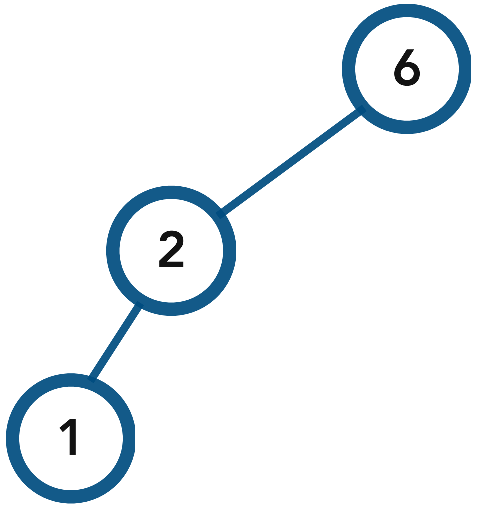


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

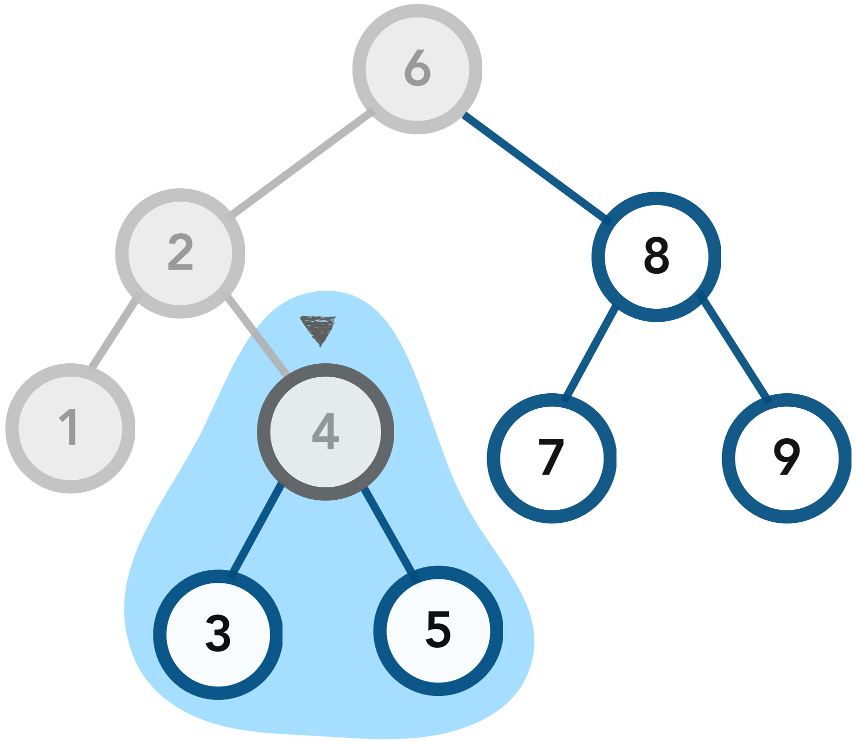


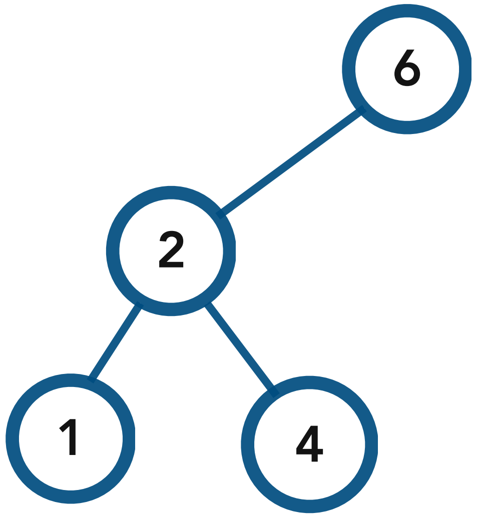


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

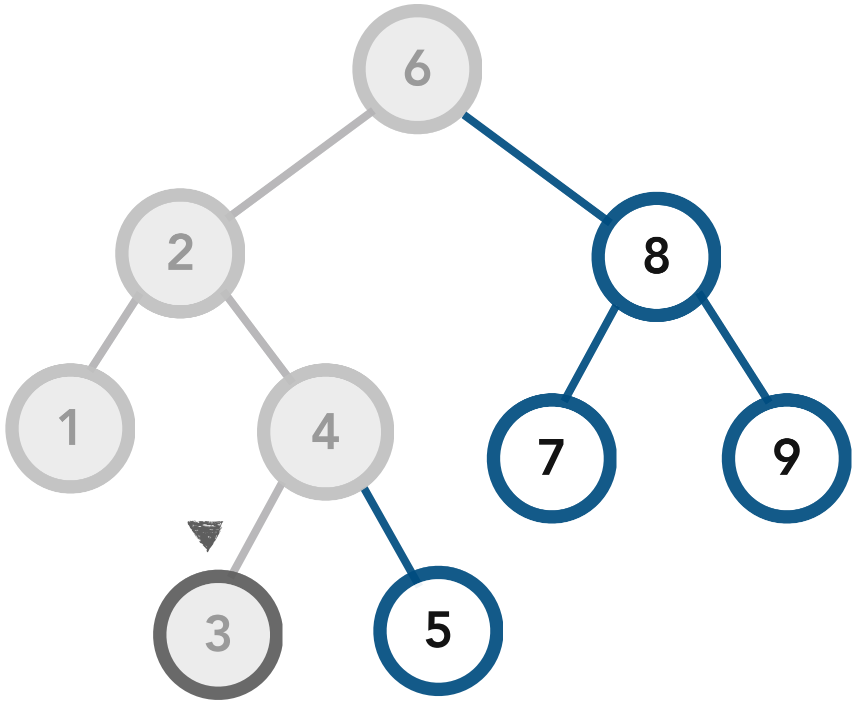


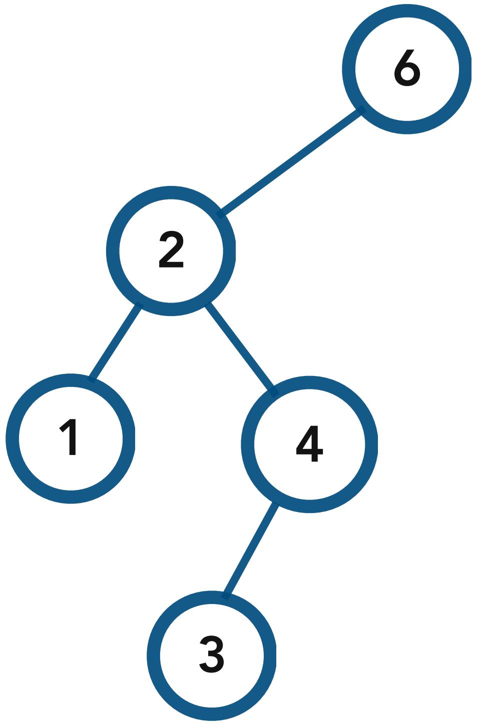


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

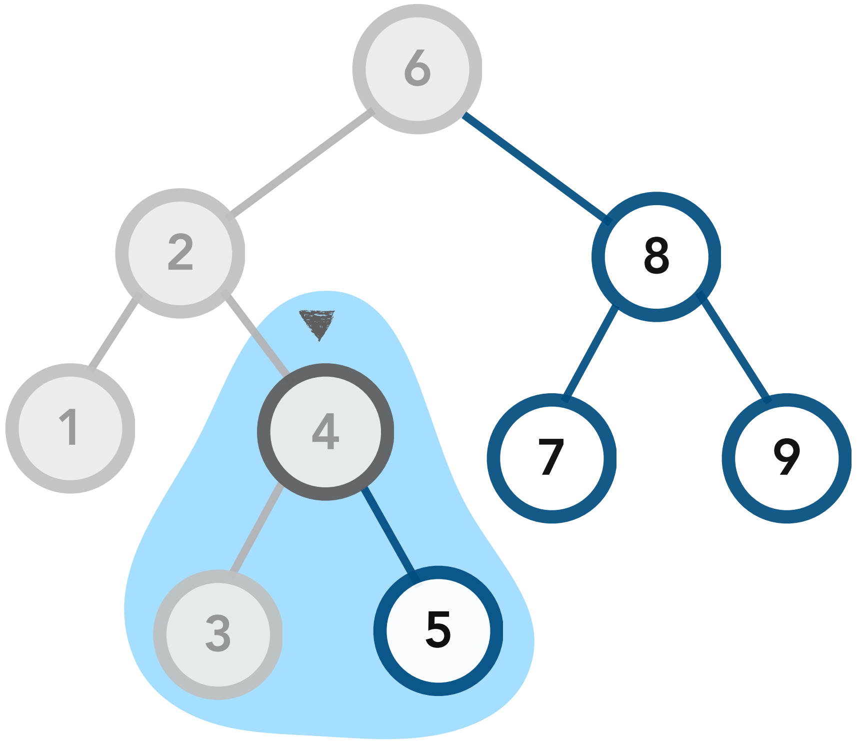


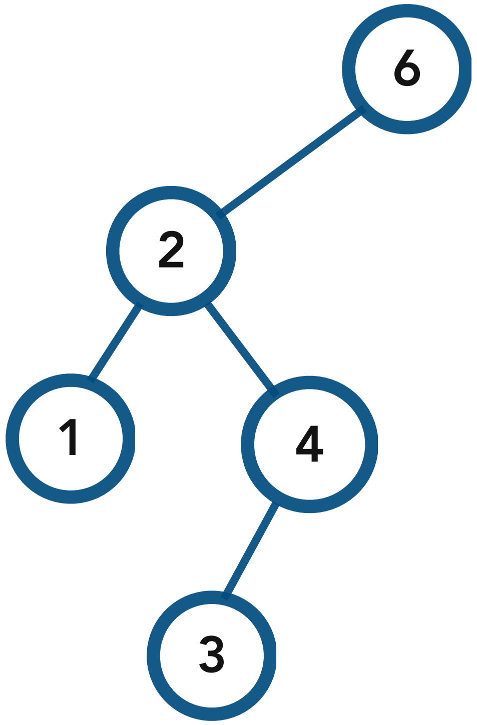


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

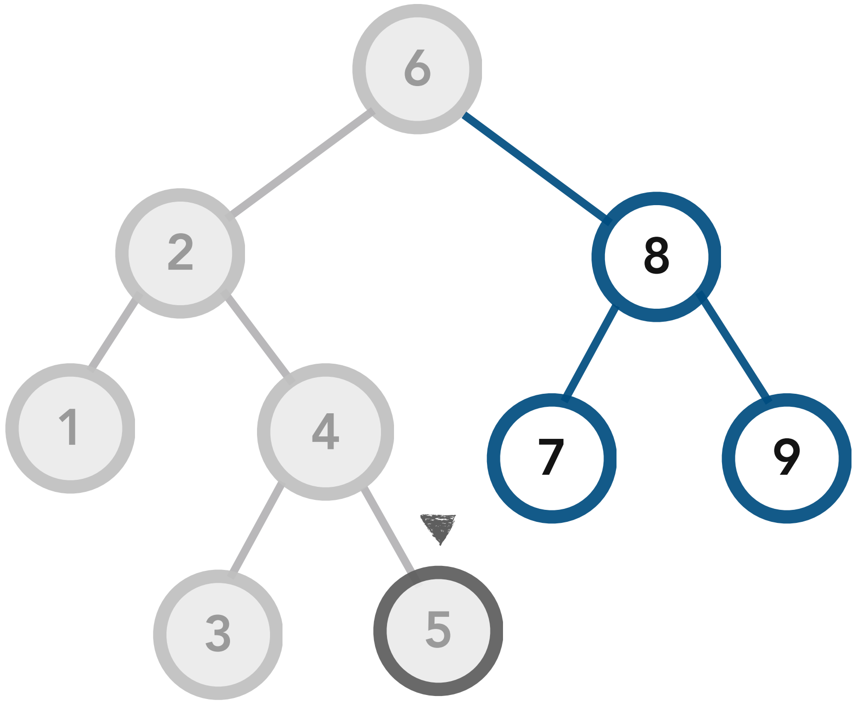


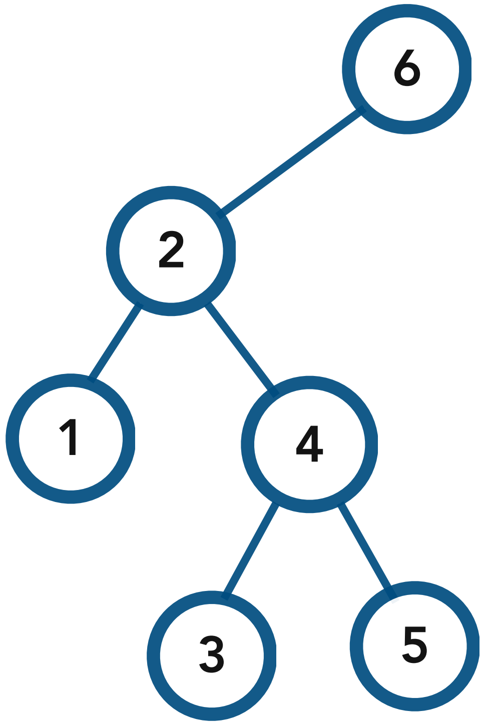


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

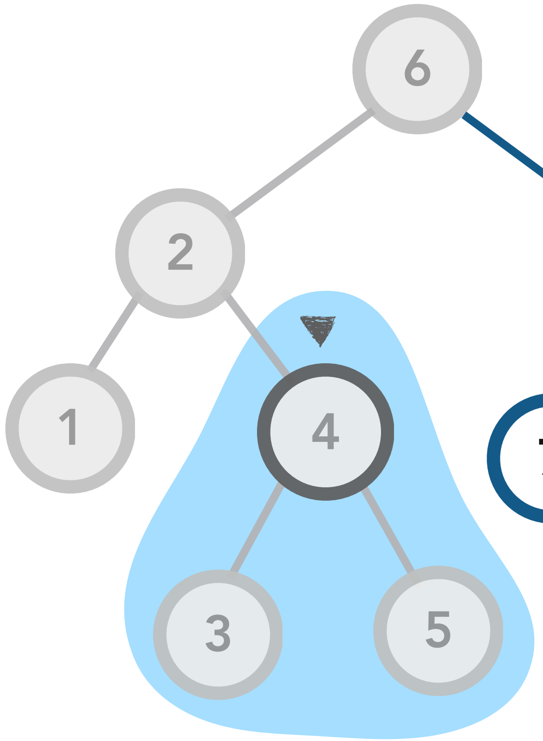


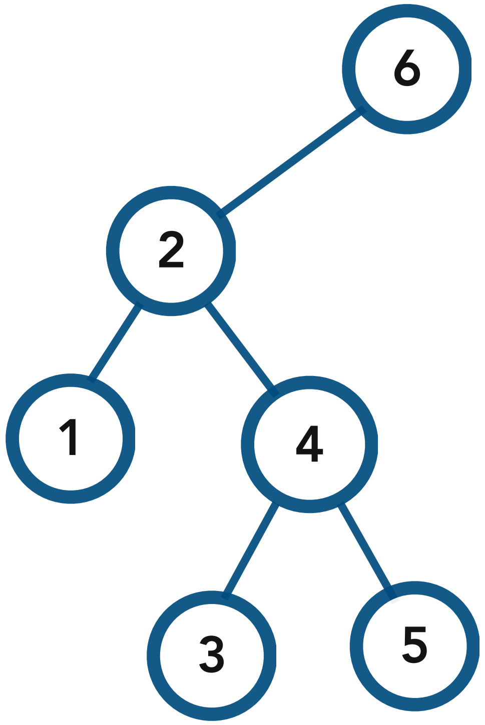


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

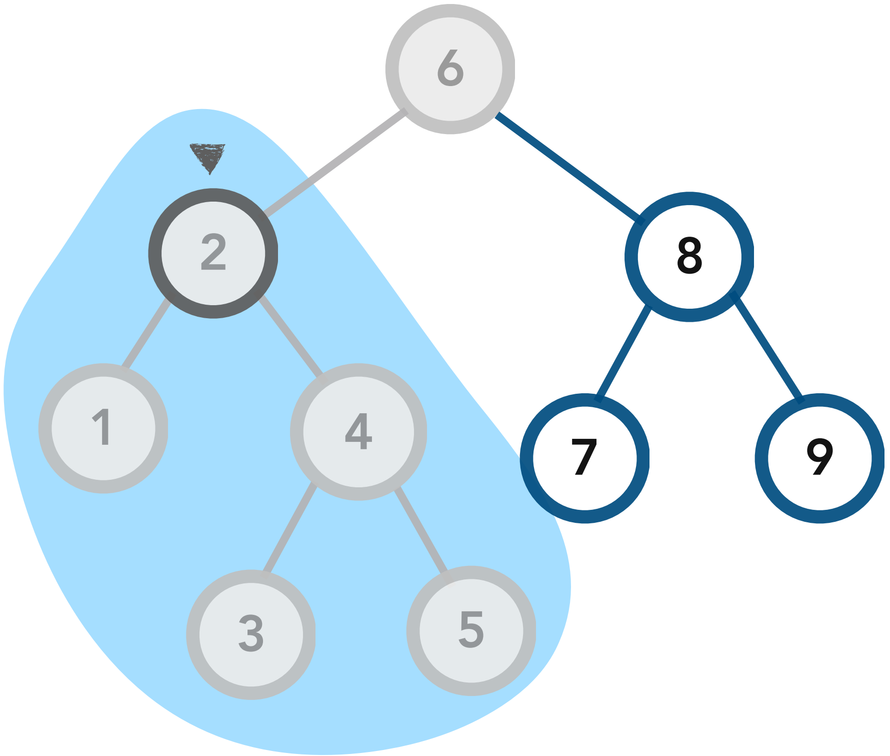


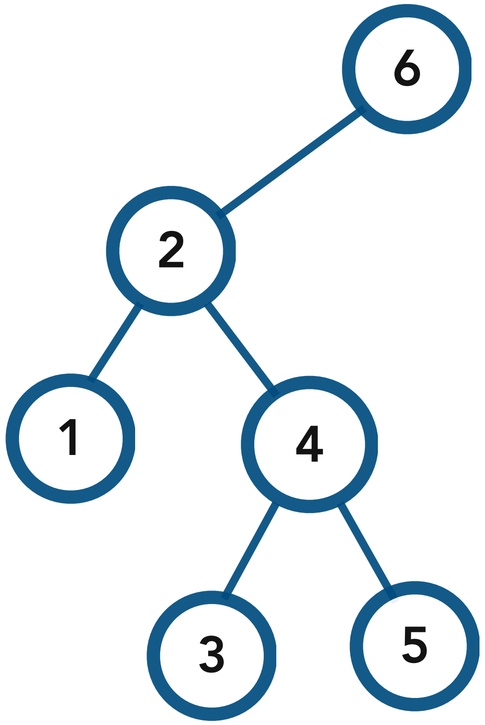


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

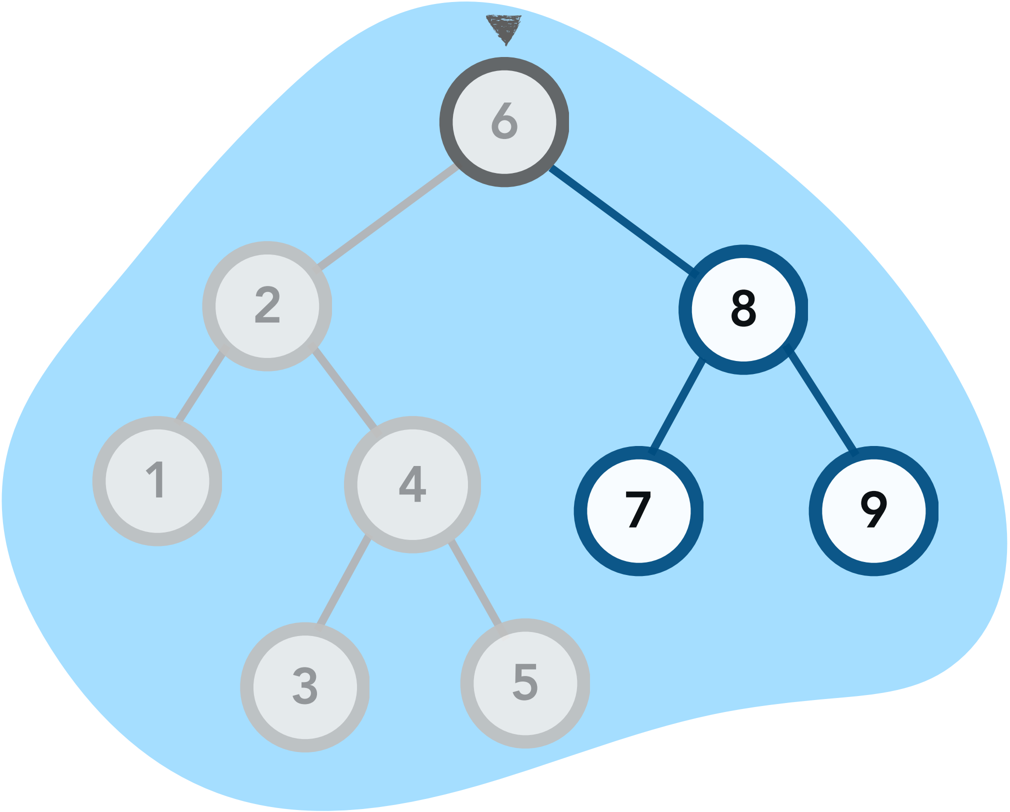


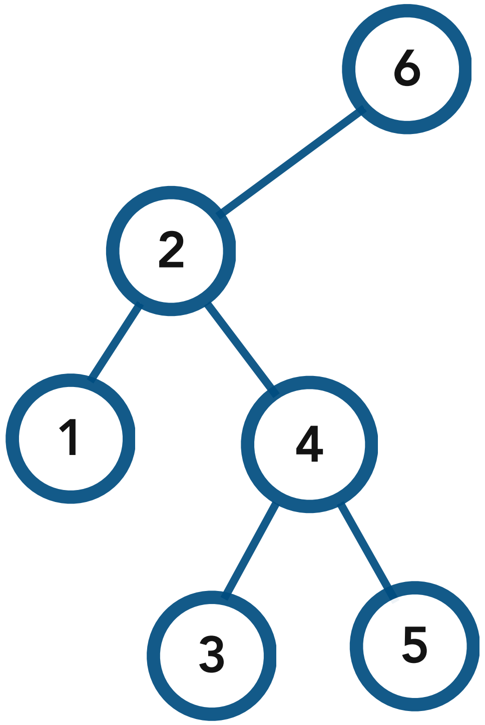


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

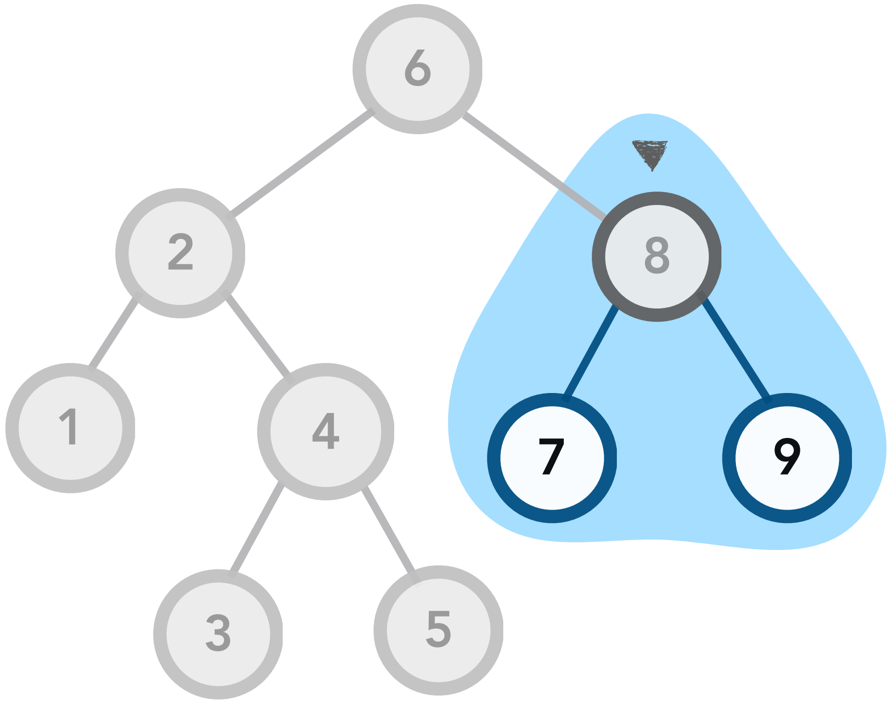


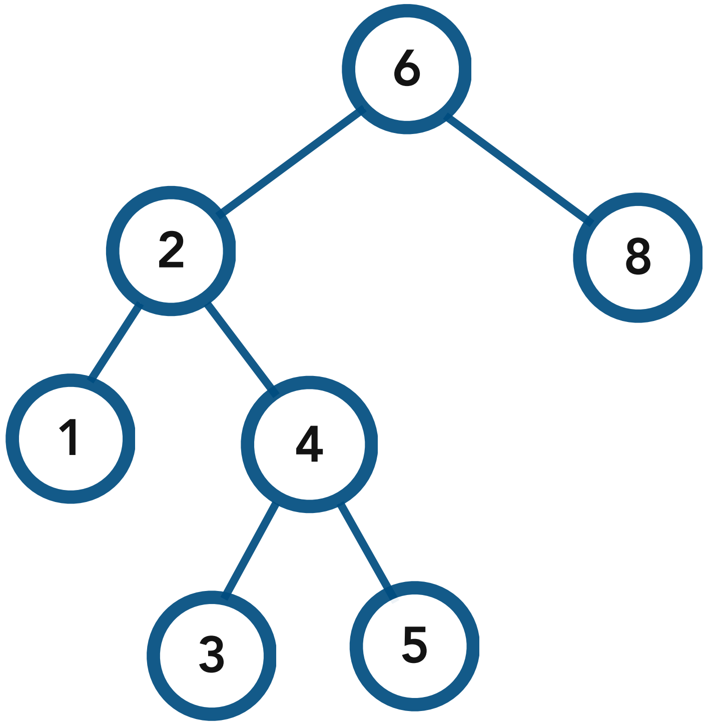


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

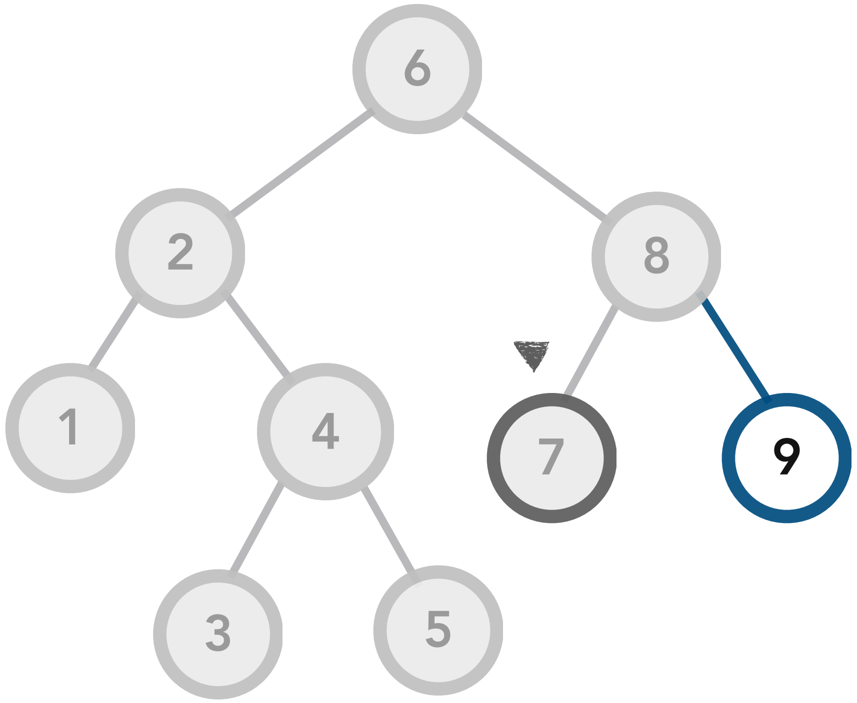


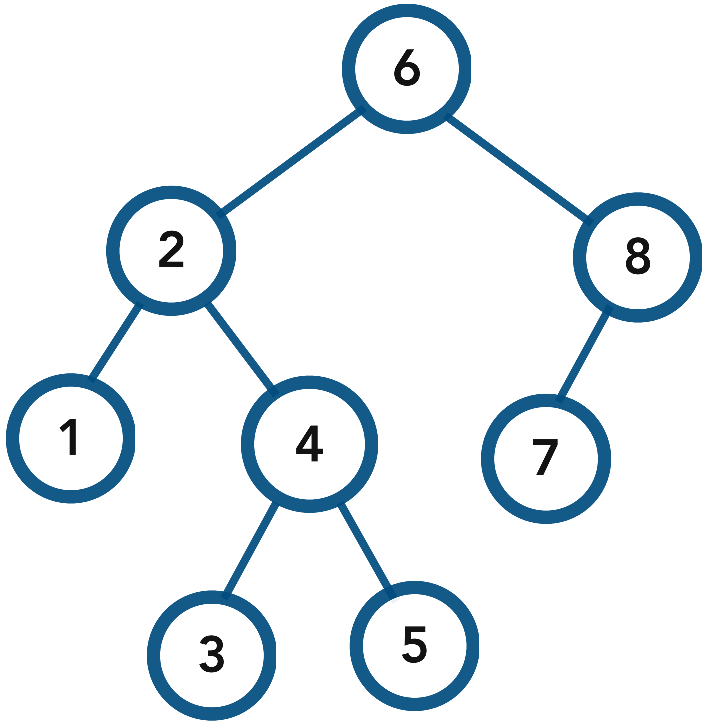


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

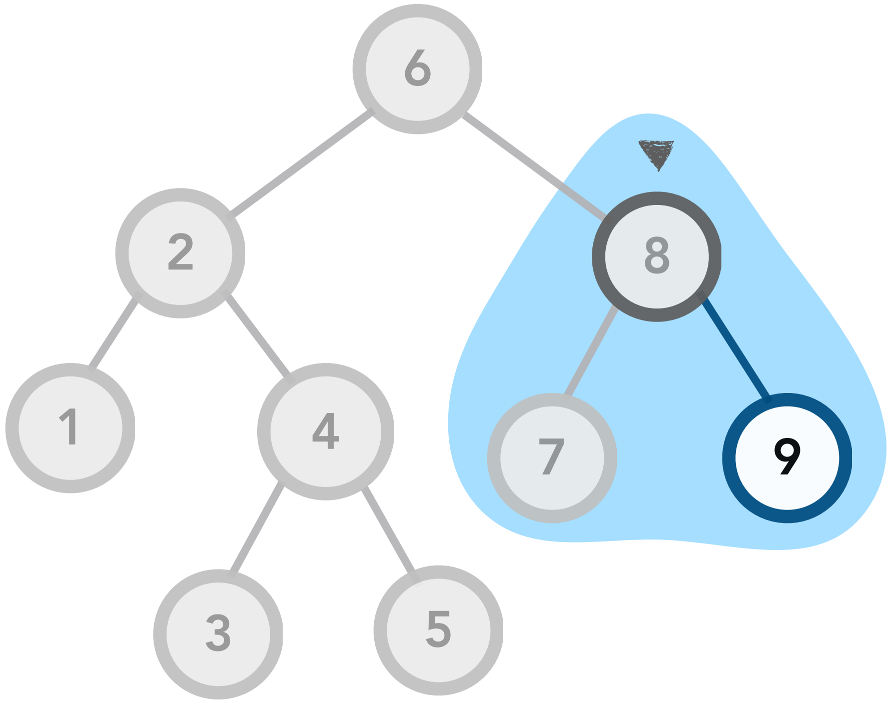


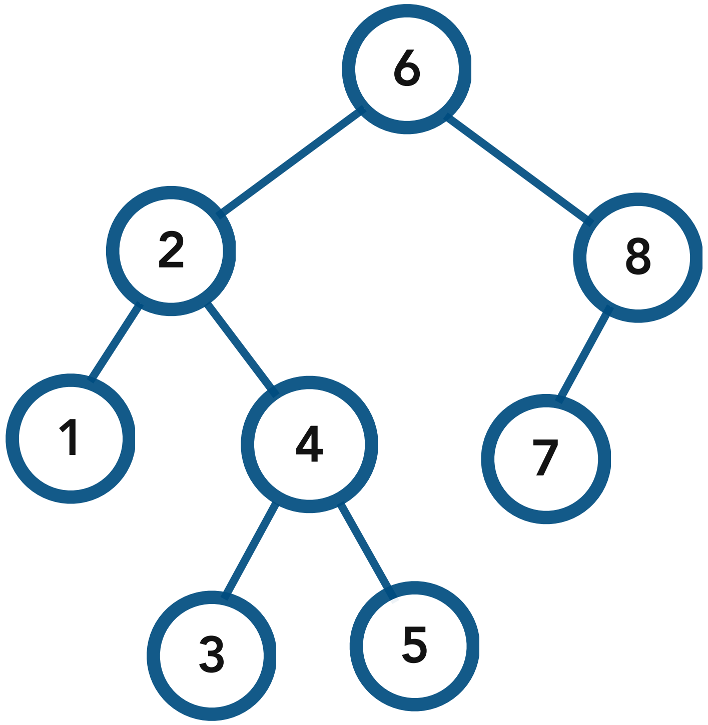


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

```
Procedure. Insert the current node value
           Copy the left subtree
           Copy the right subtree
```

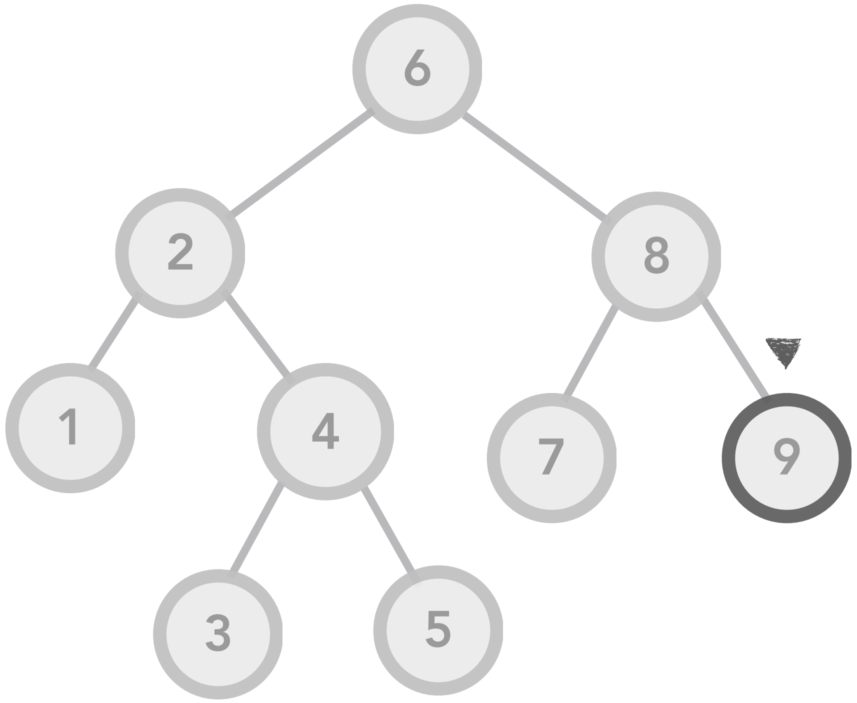


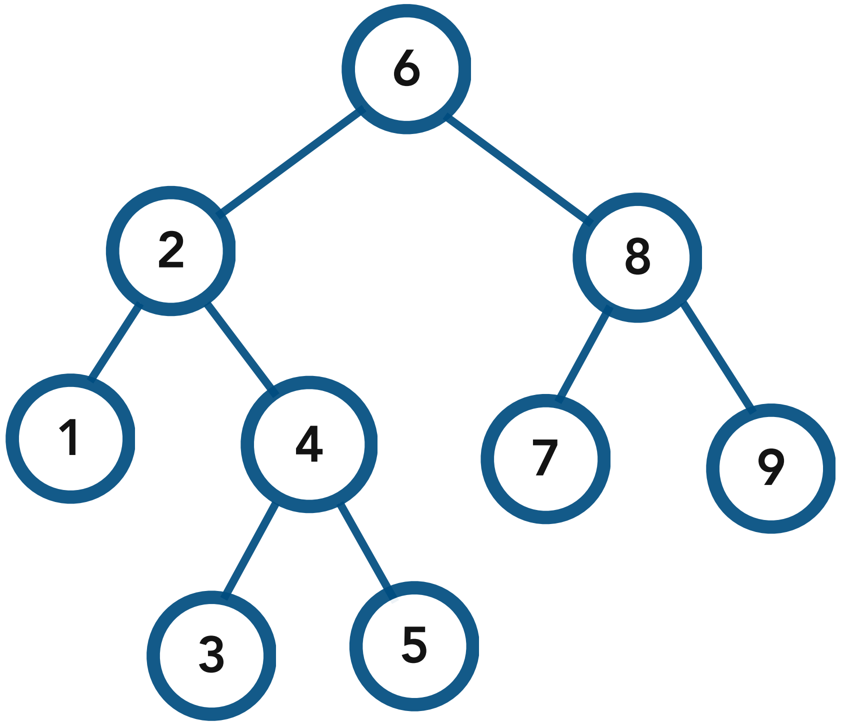


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other.*

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

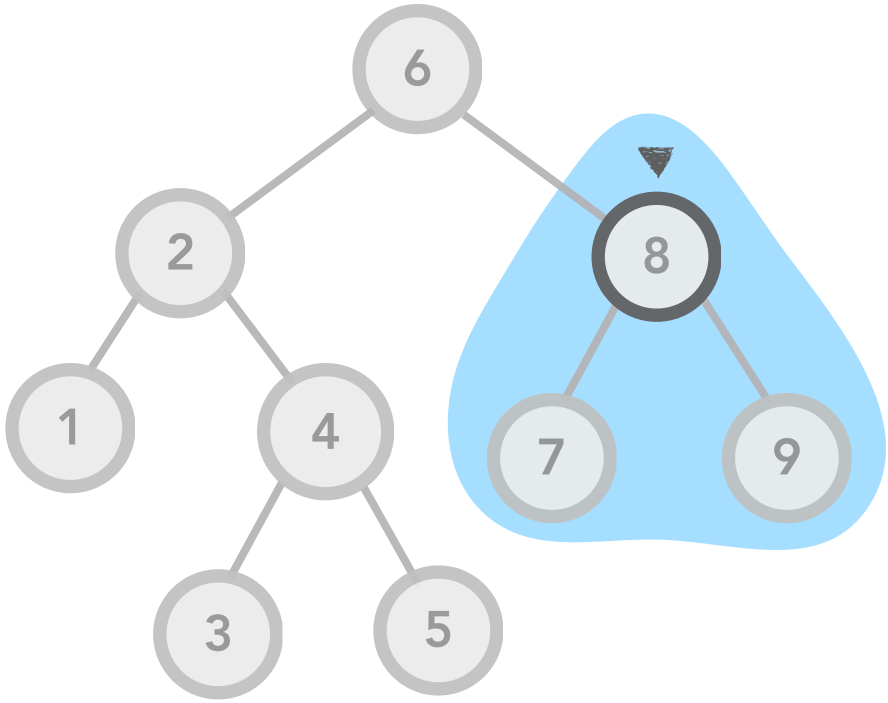


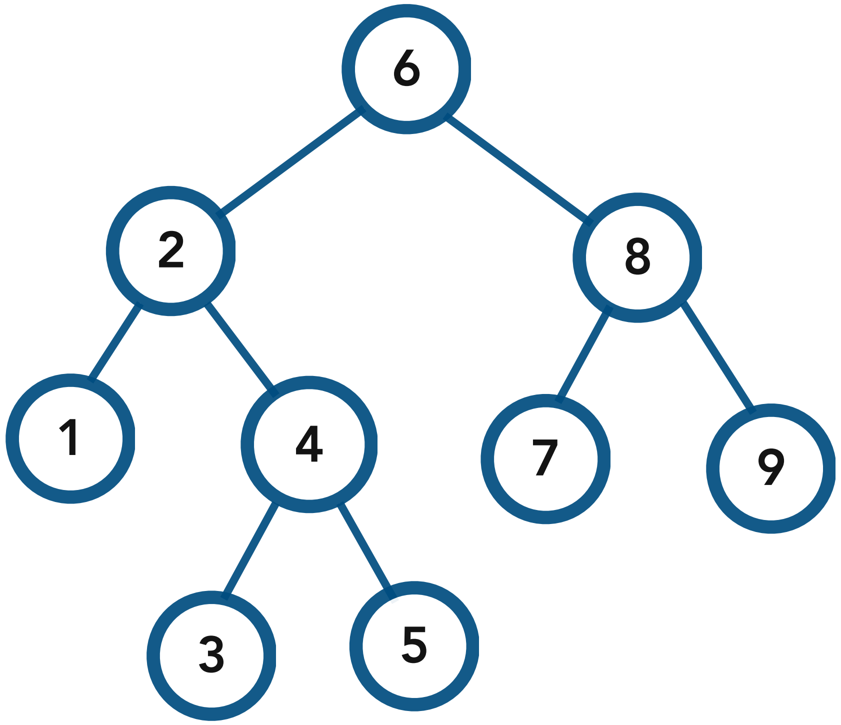


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

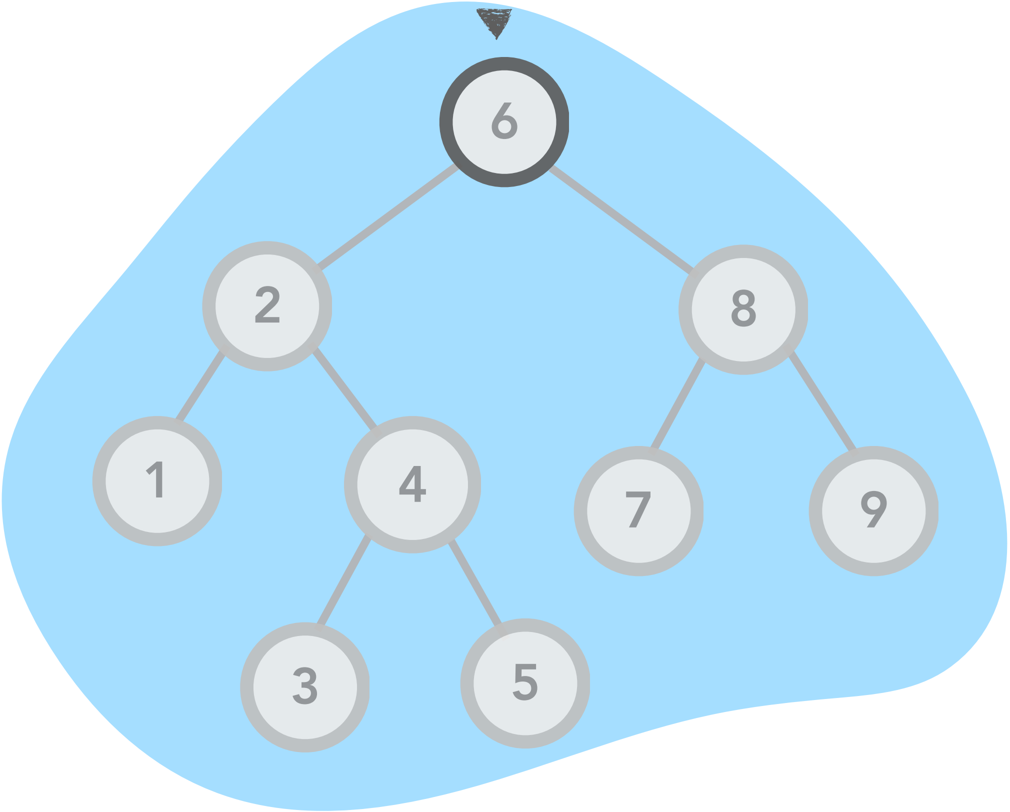


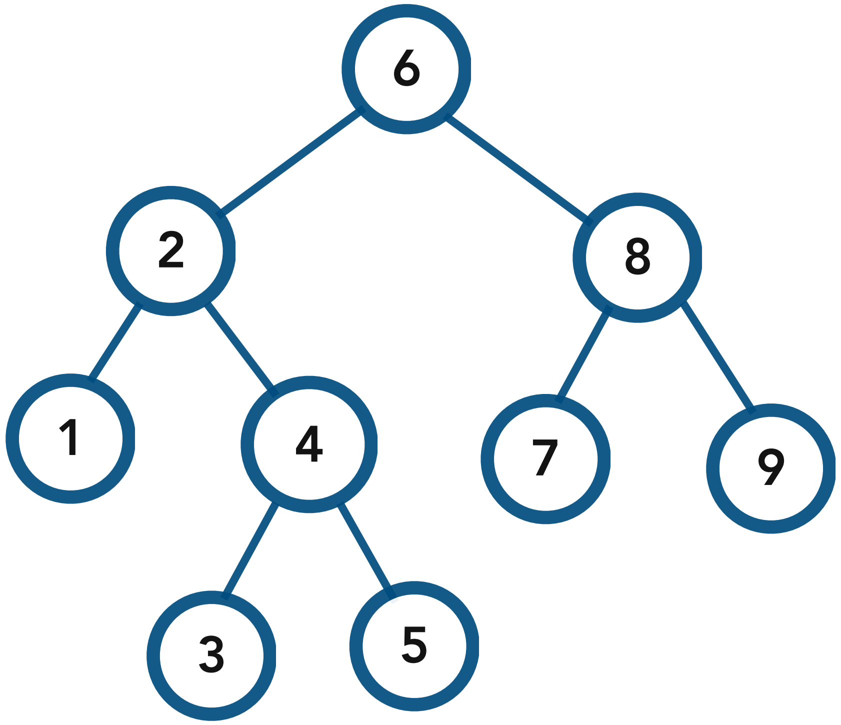


Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree

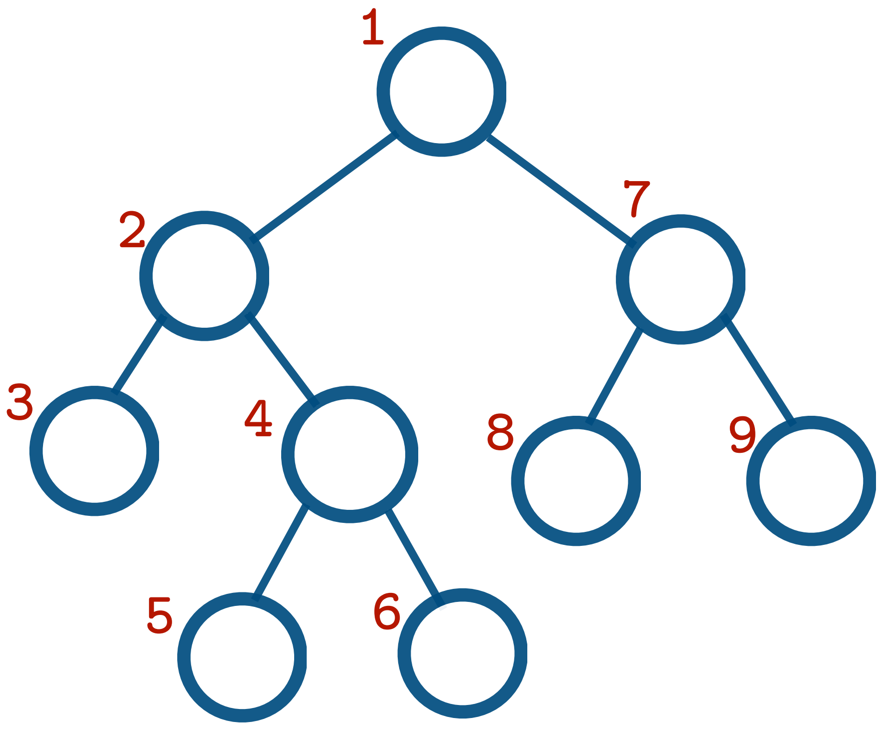




Copying the Tree

BST Copy Constructor. Given another BST named *other*, insert every node in *other* into the current tree, such that the current tree becomes exactly like *other*.

Procedure. Insert the current node value
Copy the left subtree
Copy the right subtree



```
template <class T>
BST<T>::BST(const BST<T>& other) {
    root = nullptr;
    copy_from(other.root);
}

template <class T>
void BST<T>::copy_from(Node<T>* node) {
    if (node == nullptr) return;

    insert(node->val);
    copy_from(node->left);
    copy_from(node->right);
}
```

Running Time. n nodes are inserted.
In general: $O(n \times \mathit{height})$
Balanced Trees: $O(n \log n)$
Worst Case: $O(n^2)$

Note. This code creates an *exact* copy of the tree assuming the insert function does not rebalance with rotations!

Depth-First Traversals

Problem. Given a binary tree, visit every node in the tree and perform some operation (e.g. delete the node, print the node, etc.)

Solution. Start at the root and use recursion to traverse the left and right subtrees.

In-order Traversal. Perform the operation *after* traversing the left subtree and *before* traversing the right subtree.

```
void inOrder(Node<T>* node) {
    if (node == nullptr) return;
    inOrder(node->left);
    do_something();
    inOrder(node->right);
}
```

Pre-order Traversal. Perform the operation *before* traversing the left and right subtrees.

```
void preOrder(Node<T>* node) {
    if (node == nullptr) return;
    do_something();
    preOrder(node->left);
    preOrder(node->right);
}
```

Post-order Traversal. Perform the operation *after* traversing the left and right subtrees.

```
void preOrder(Node<T>* node) {
    if (node == nullptr) return;
    preOrder(node->left);
    preOrder(node->right);
    do_something();
}
```

Depth-First Traversals

Problem. Given a binary tree, visit every node in the tree and perform some operation (e.g. delete the node, print the node, etc.)

Solution. Start at the root and use recursion to traverse the left and right subtrees.

Example applications. Copying a BST and computing node depths.

Pre-order Traversal. Perform the operation *before* traversing the left and right subtrees.

```
void preOrder(Node<T>* node) {
    if (node == nullptr) return;
    do_something();
    preOrder(node->left);
    preOrder(node->right);
}
```

In-order Traversal. Perform the operation *after* traversing the left subtree and *before* traversing the right subtree.

```
void inOrder(Node<T>* node) {
    if (node == nullptr) return;
    inOrder(node->left);
    do_something();
    inOrder(node->right);
}
```

Post-order Traversal. Perform the operation *after* traversing the left and right subtrees.

```
void preOrder(Node<T>* node) {
    if (node == nullptr) return;
    preOrder(node->left);
    preOrder(node->right);
    do_something();
}
```

Depth-First Traversals

Problem. Given a binary tree, visit every node in the tree and perform some operation (e.g. delete the node, print the node, etc.)

Solution. Start at the root and use recursion to traverse the left and right subtrees.

Example application.
Printing a BST in order

In-order Traversal. Perform the operation *after* traversing the left subtree and *before* traversing the right subtree.

```
void inOrder(Node<T>* node) {
    if (node == nullptr) return;
    inOrder(node->left);
    do_something();
    inOrder(node->right);
}
```

Pre-order Traversal. Perform the operation *before* traversing the left and right subtrees.

```
void preOrder(Node<T>* node) {
    if (node == nullptr) return;
    do_something();
    preOrder(node->left);
    preOrder(node->right);
}
```

Post-order Traversal. Perform the operation *after* traversing the left and right subtrees.

```
void preOrder(Node<T>* node) {
    if (node == nullptr) return;
    preOrder(node->left);
    preOrder(node->right);
    do_something();
}
```

Depth-First Traversals

Problem. Given a binary tree, visit every node in the tree and perform some operation (e.g. delete the node, print the node, etc.)

Solution. Start at the root and use recursion to traverse the left and right subtrees.

Pre-order Traversal. Perform the operation *before* traversing the left and right subtrees.

```
void preOrder(Node<T>* node) {
    if (node == nullptr) return;
    do_something();
    preOrder(node->left);
    preOrder(node->right);
}
```

In-order Traversal. Perform the operation *after* traversing the left subtree and *before* traversing the right subtree.

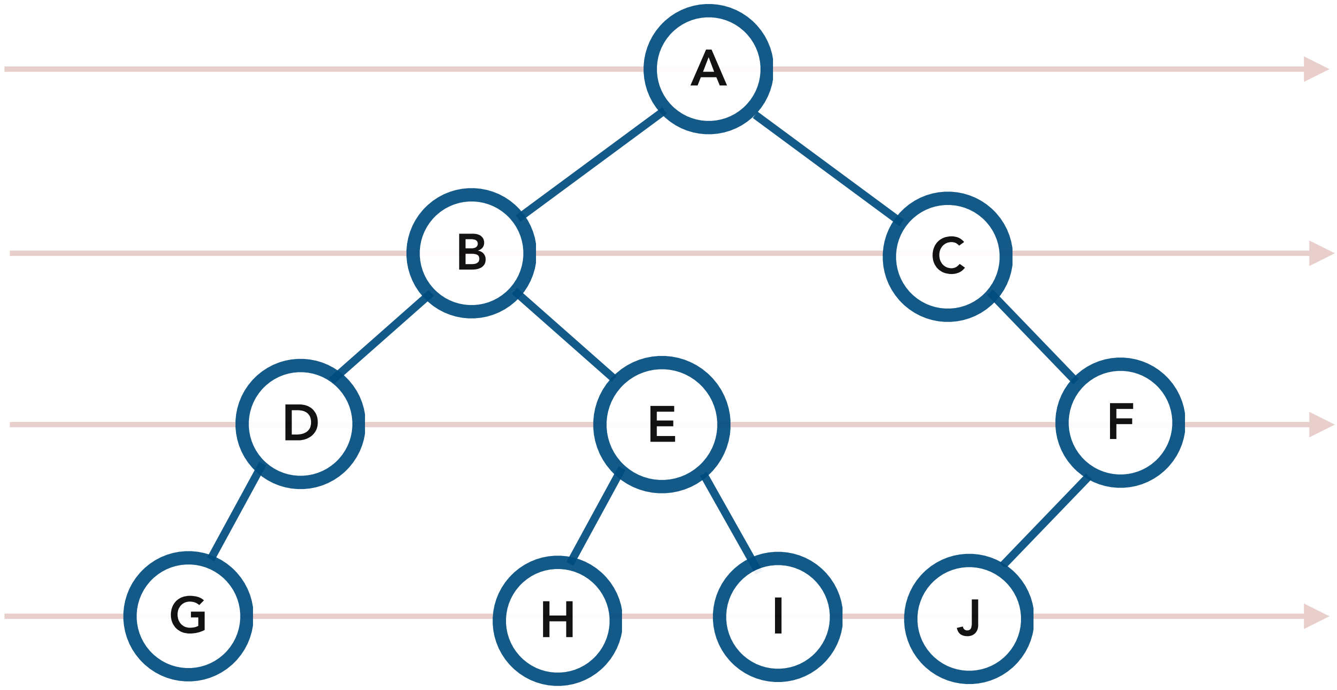
```
void inOrder(Node<T>* node) {
    if (node == nullptr) return;
    inOrder(node->left);
    do_something();
    inOrder(node->right);
}
```

Post-order Traversal. Perform the operation *after* traversing the left and right subtrees.

```
void preOrder(Node<T>* node) {
    if (node == nullptr) return;
    preOrder(node->left);
    preOrder(node->right);
    do_something();
}
```

Example applications. Clearing a tree and computing node heights.

Printing the Tree (level-by-level)

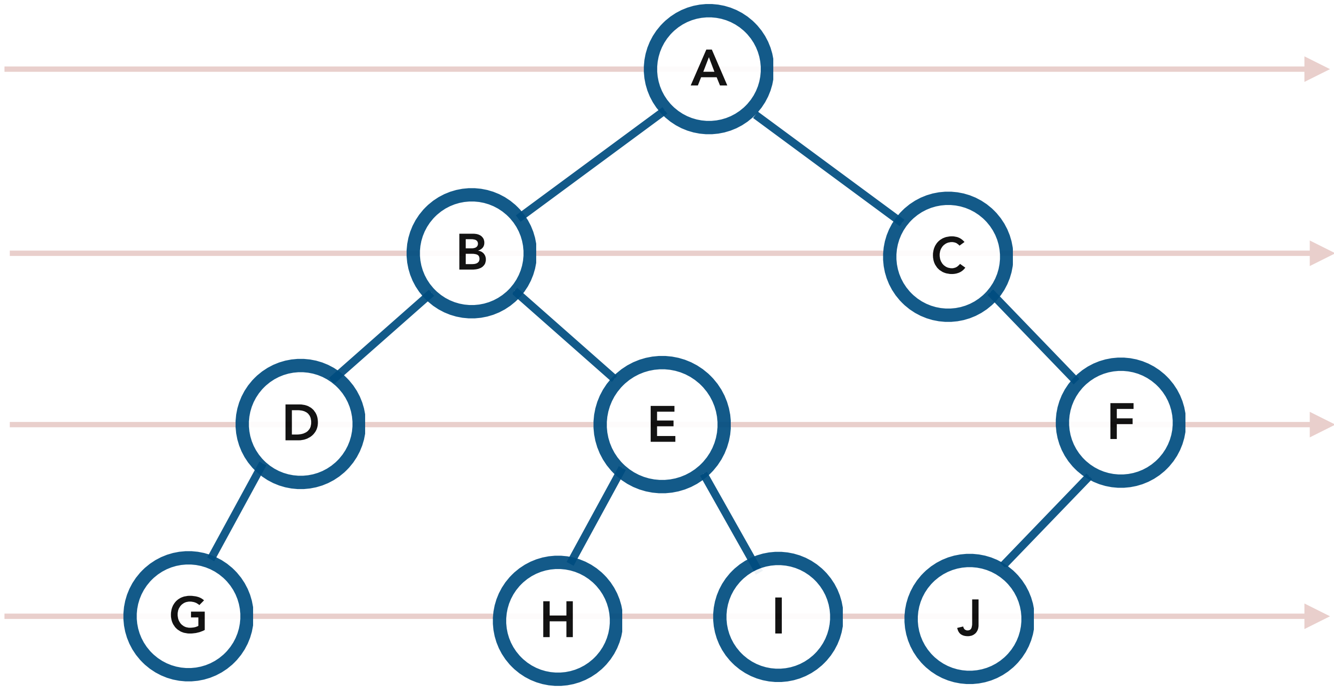


How can we traverse the tree level-by-level?

Printing the Tree (level-by-level)

Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



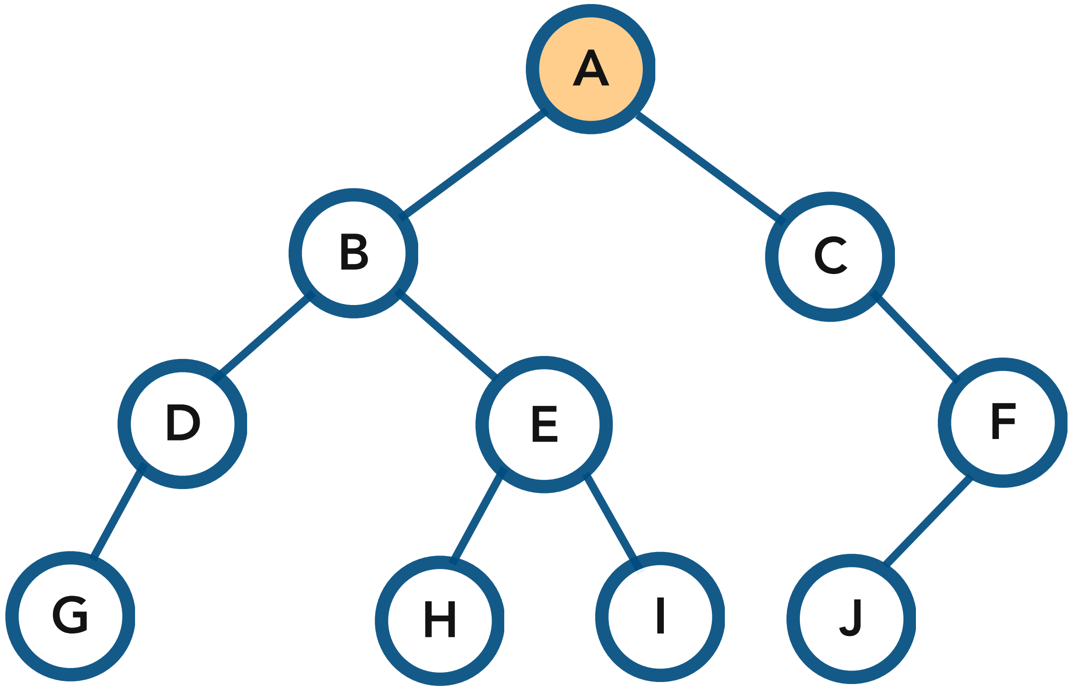
Queue

Console

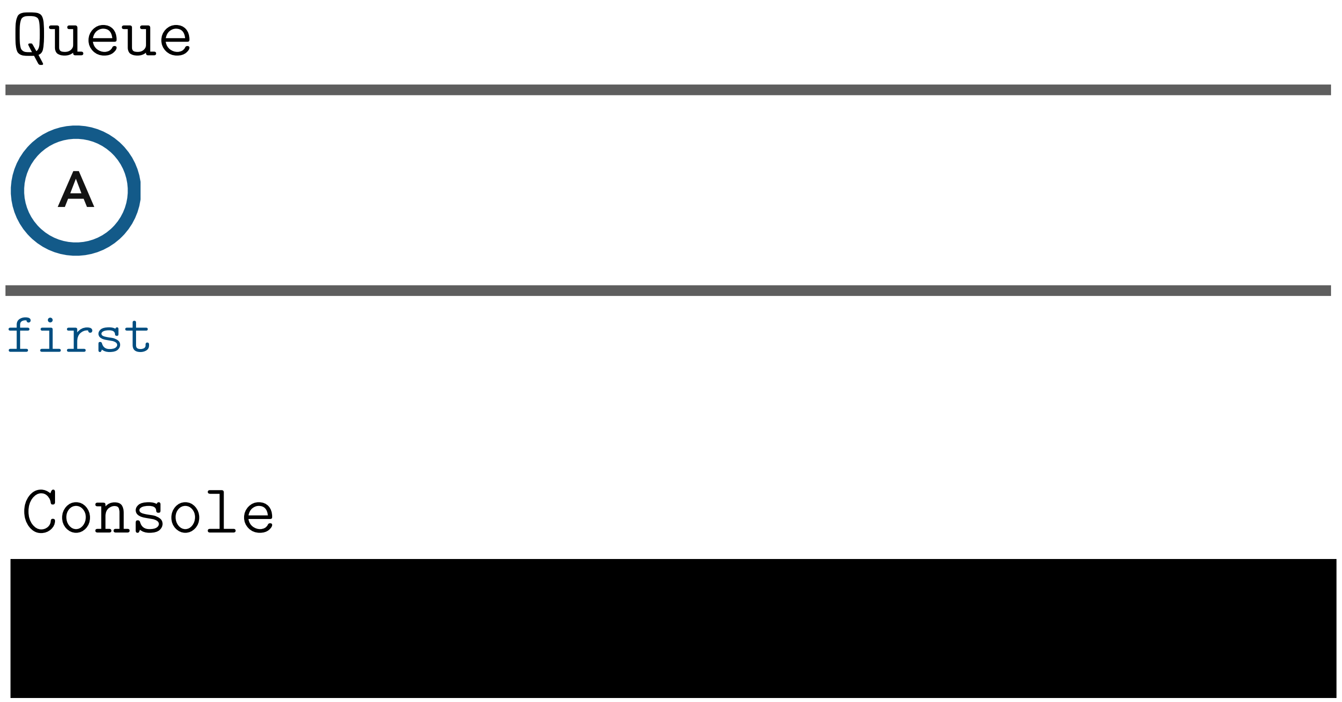
Printing the Tree (level-by-level)

Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



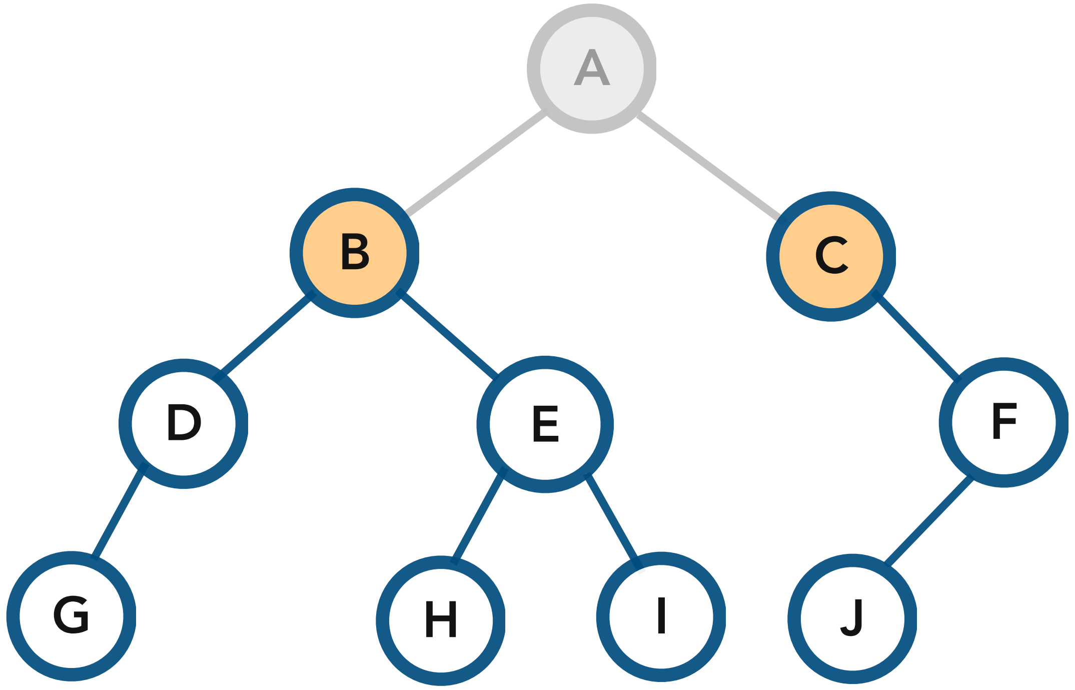
Initially, the root is added to the queue



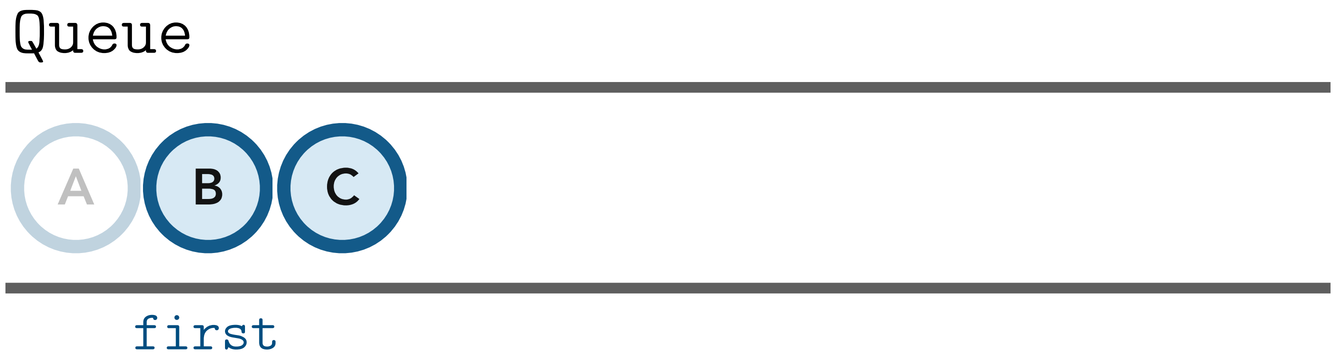
Printing the Tree (level-by-level)

Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



Remove **A** and add its children



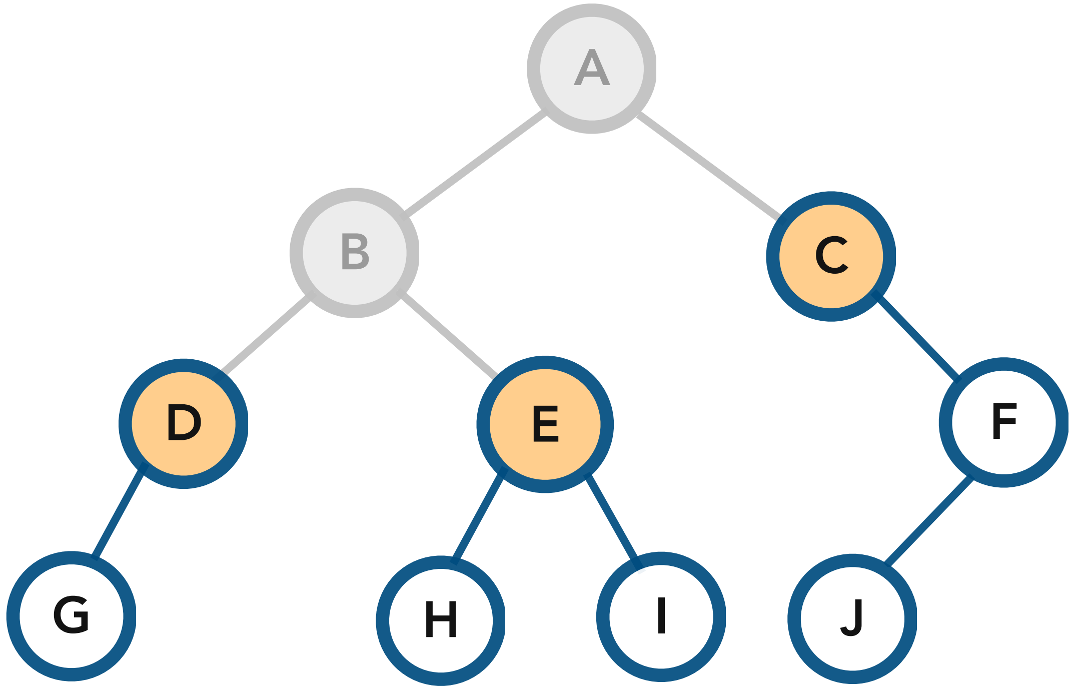
Console

```
A
```

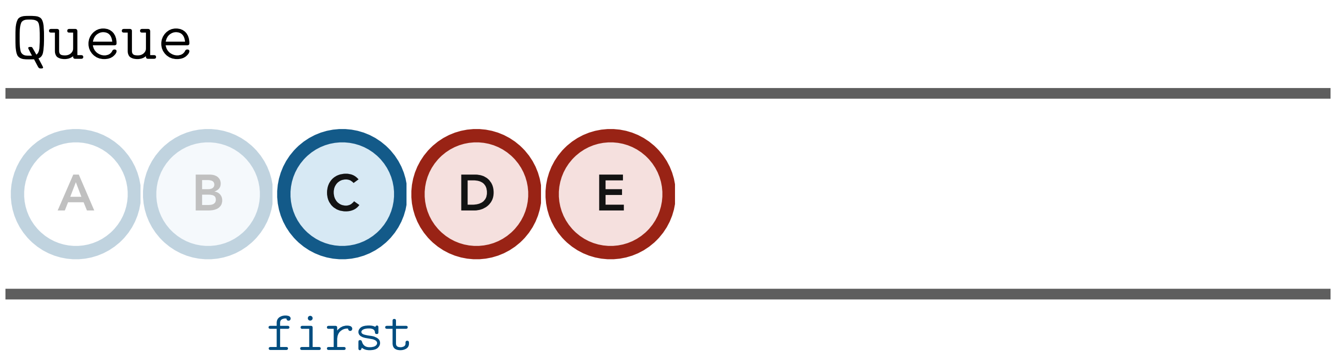
Printing the Tree (level-by-level)

Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



Remove **B** and add its children



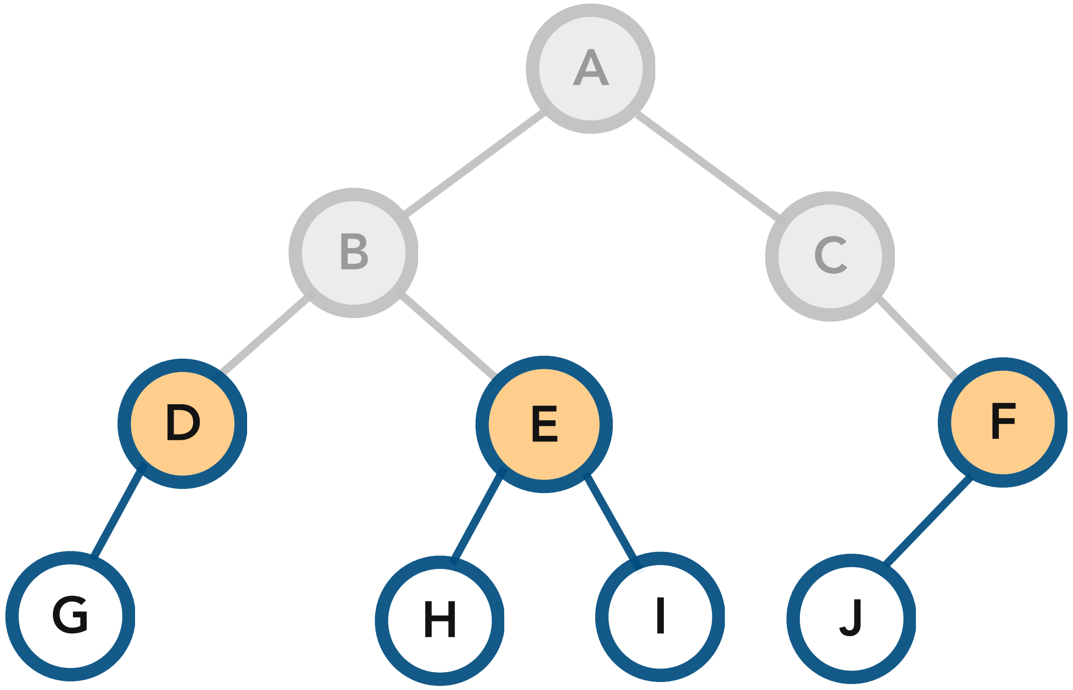
Console

```
A B
```

Printing the Tree (level-by-level)

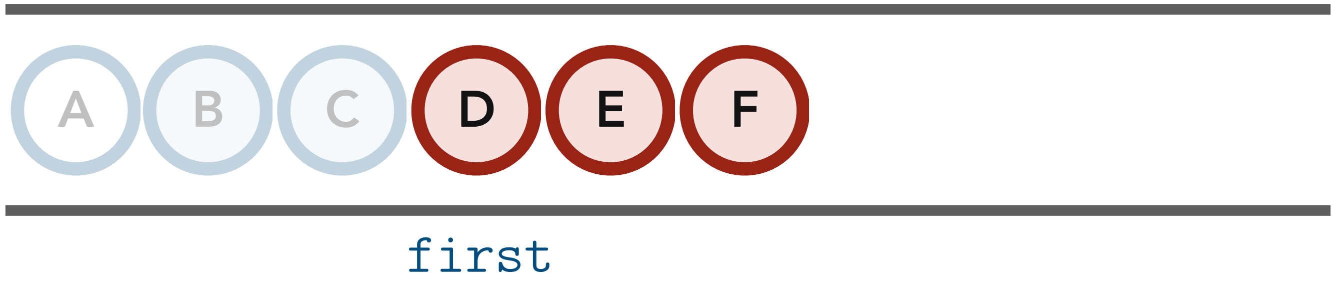
Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



Remove **C** and add its children

Queue



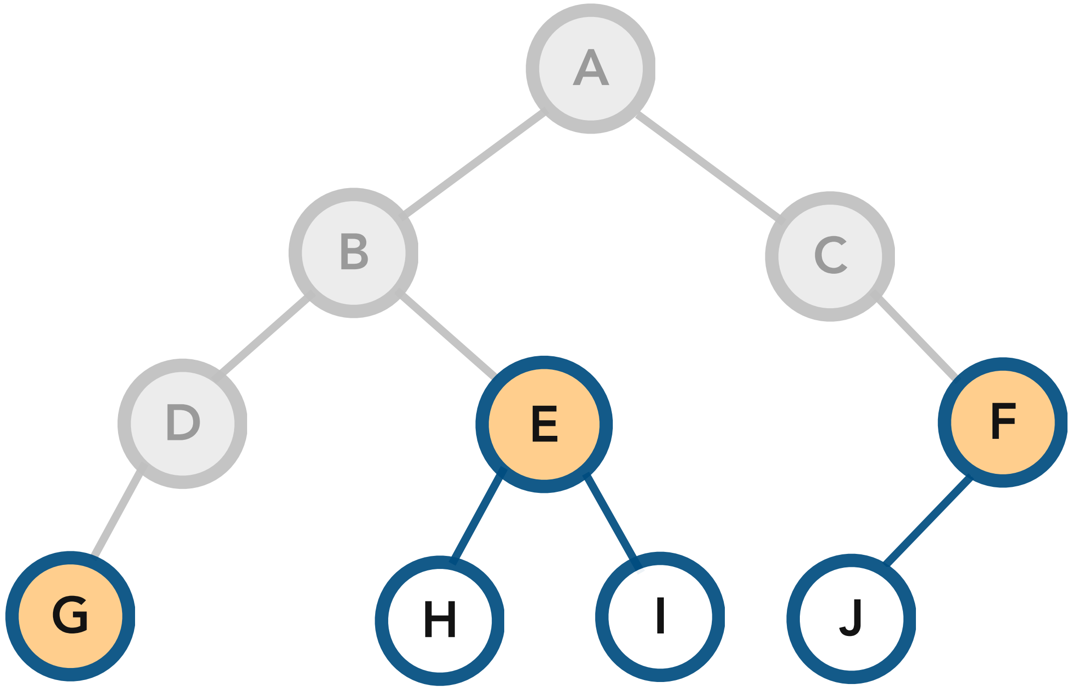
Console

```
A B C
```

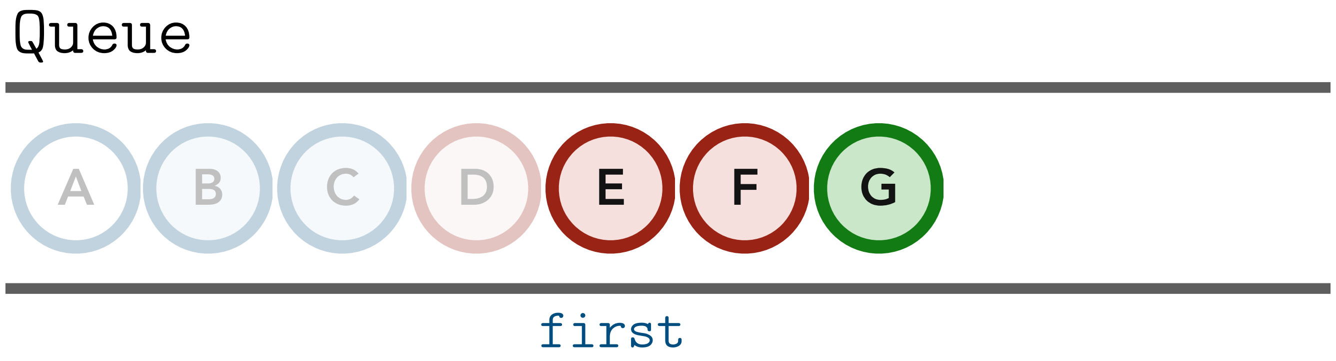
Printing the Tree (level-by-level)

Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



Remove **D** and add its children



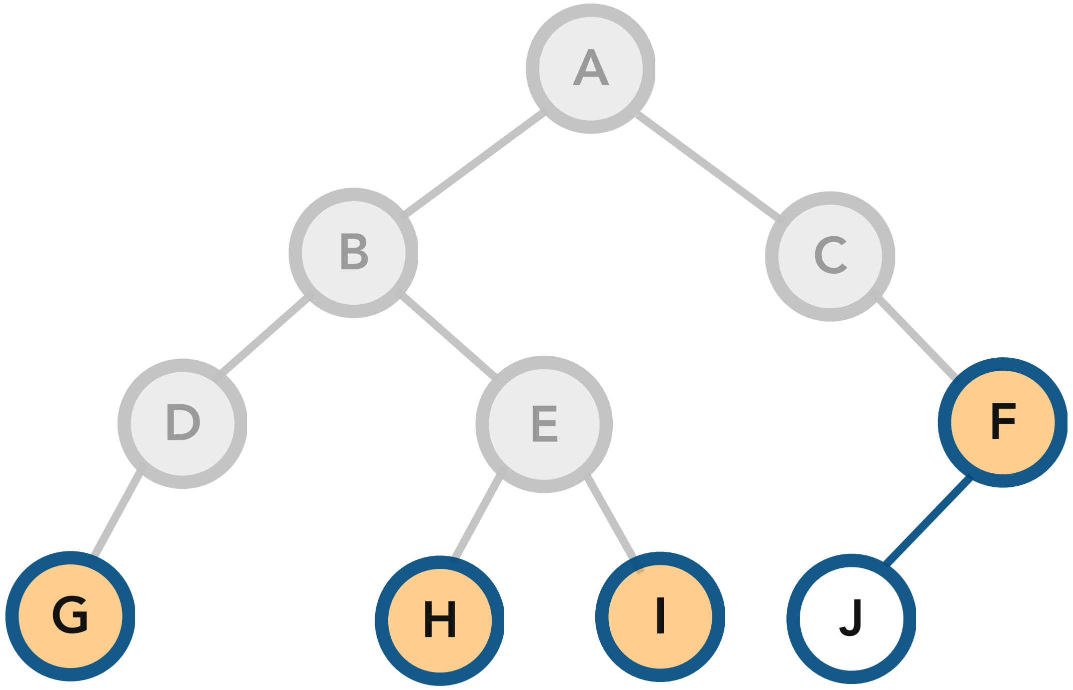
Console

```
A B C D
```

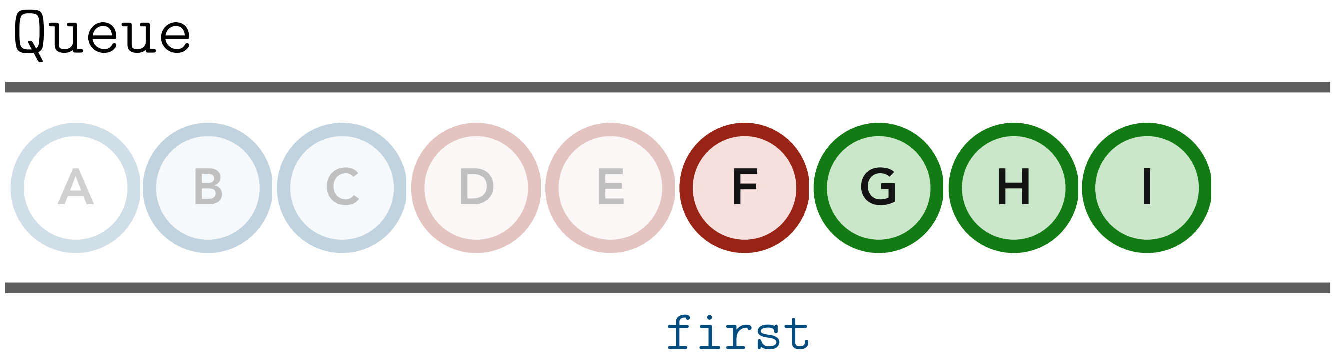
Printing the Tree (level-by-level)

Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



Remove **E** and add its children



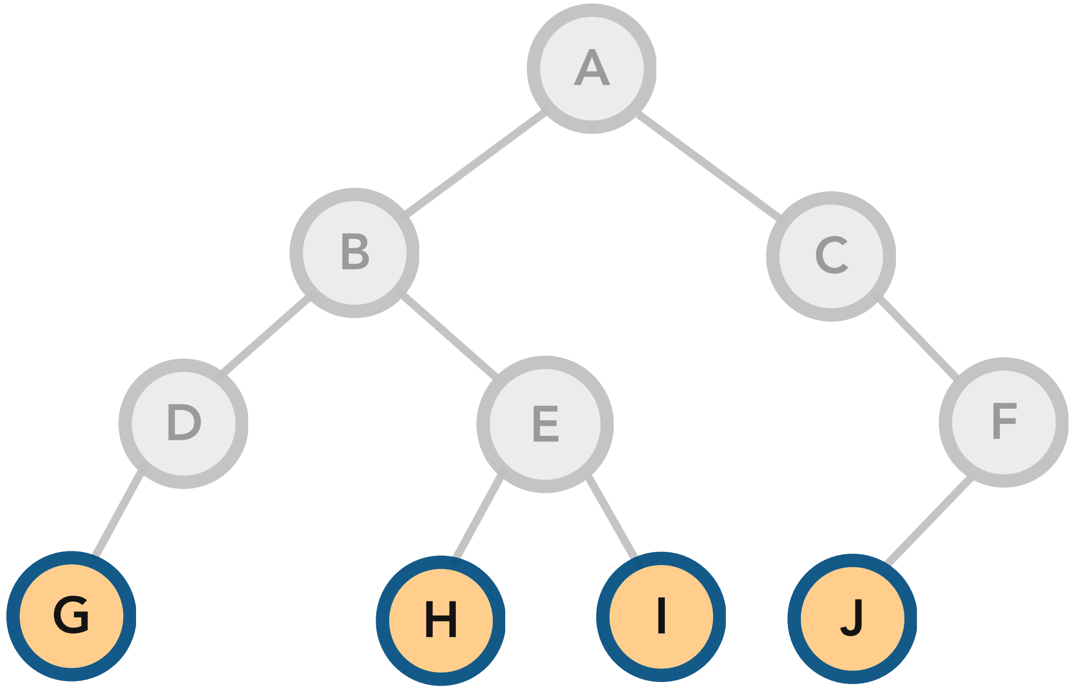
Console

```
A B C D E
```

Printing the Tree (level-by-level)

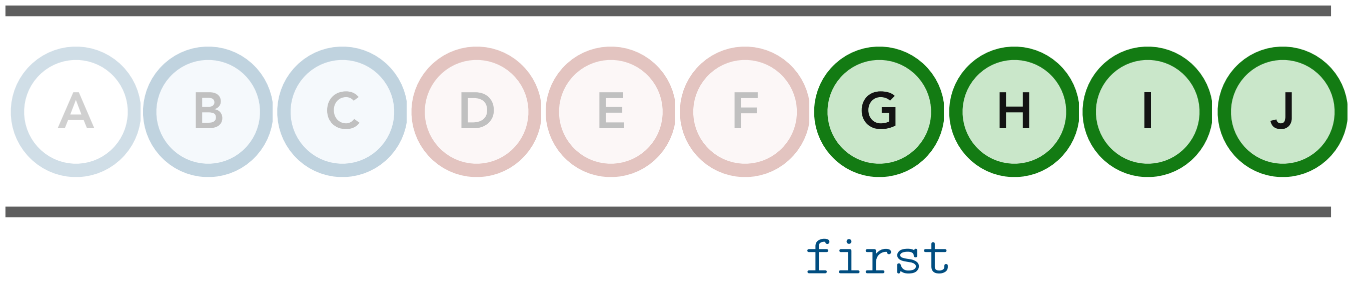
Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



Remove **F** and add its children

Queue



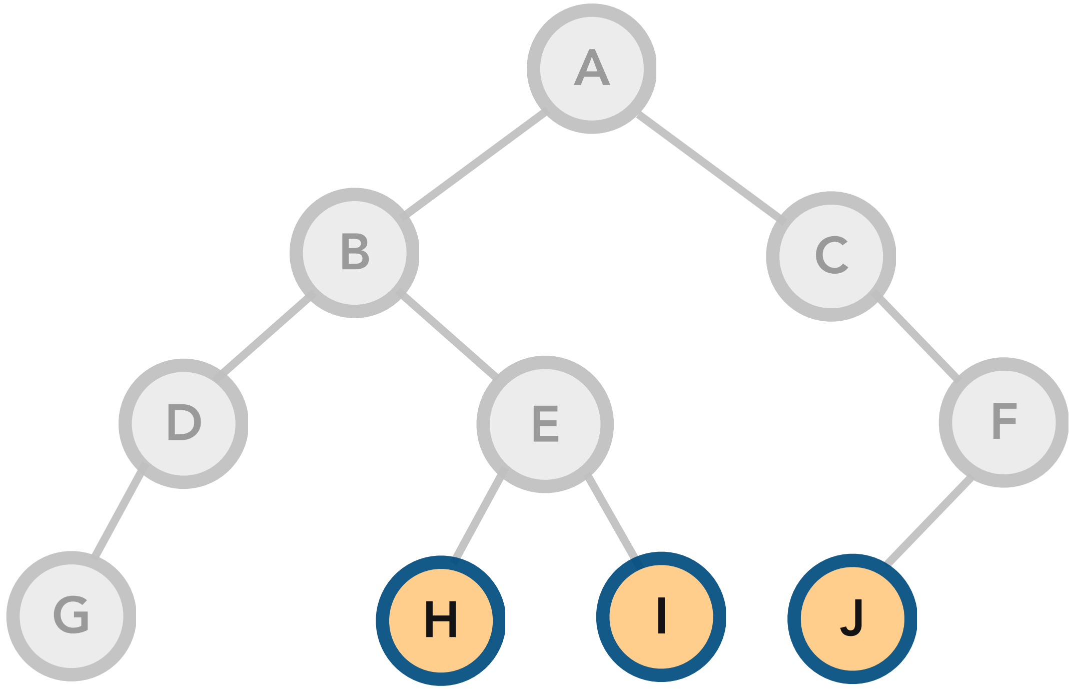
Console

```
A B C D E F
```

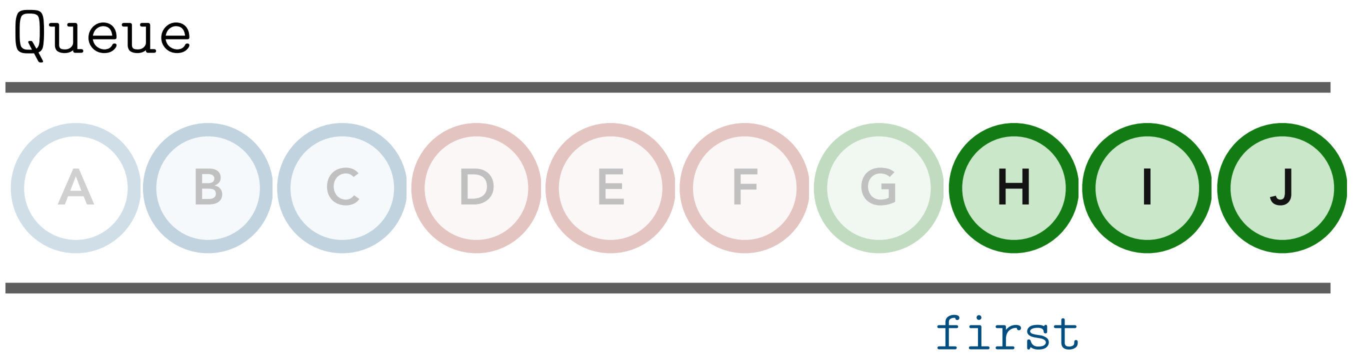
Printing the Tree (level-by-level)

Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



Remove **G**



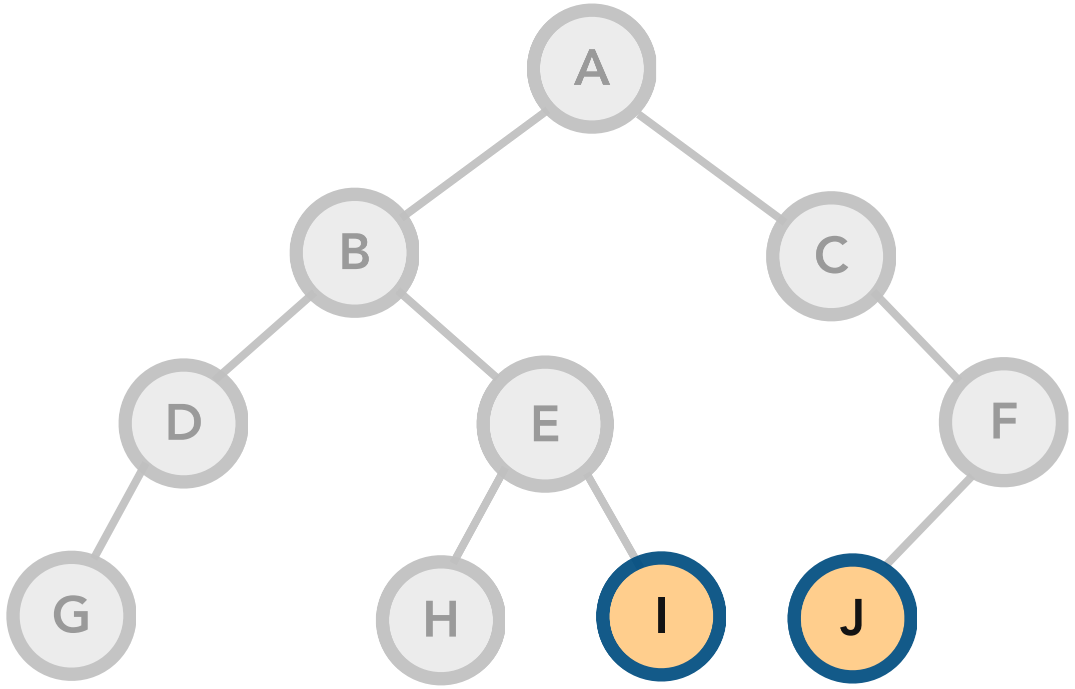
Console

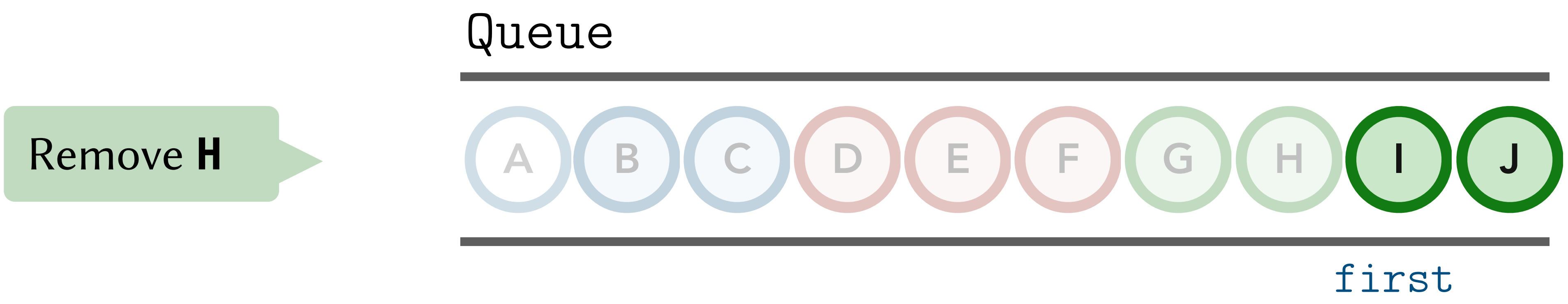
```
A B C D E F G
```

Printing the Tree (level-by-level)

Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.





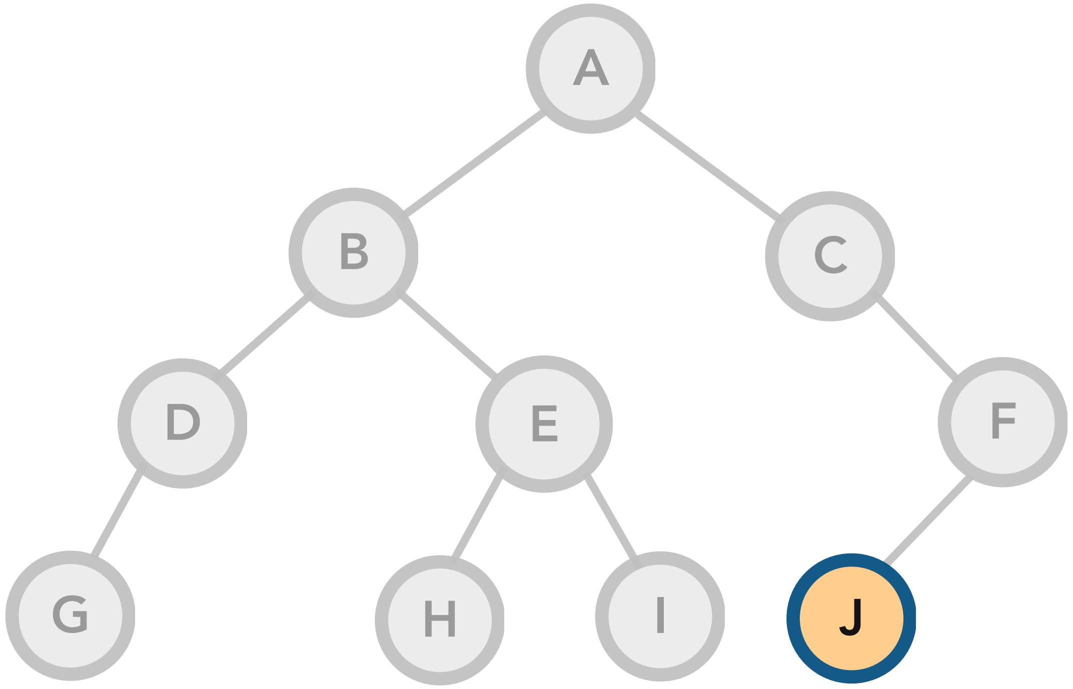
Console

```
A B C D E F G H
```

Printing the Tree (level-by-level)

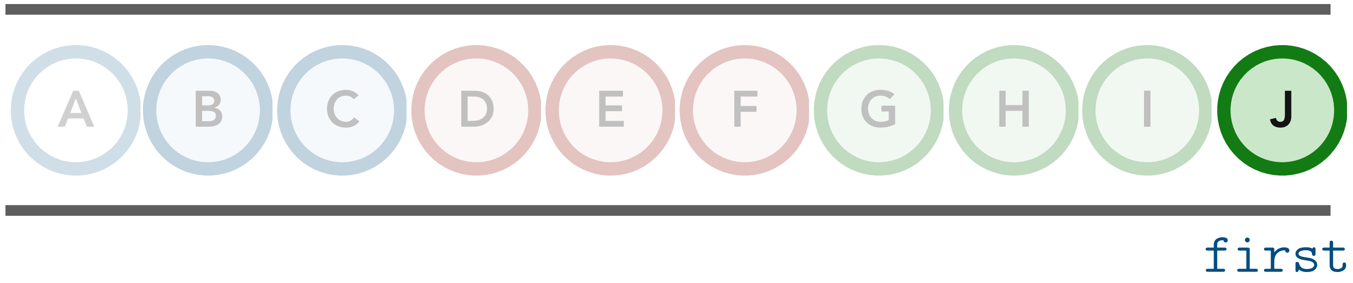
Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



Remove **I**

Queue



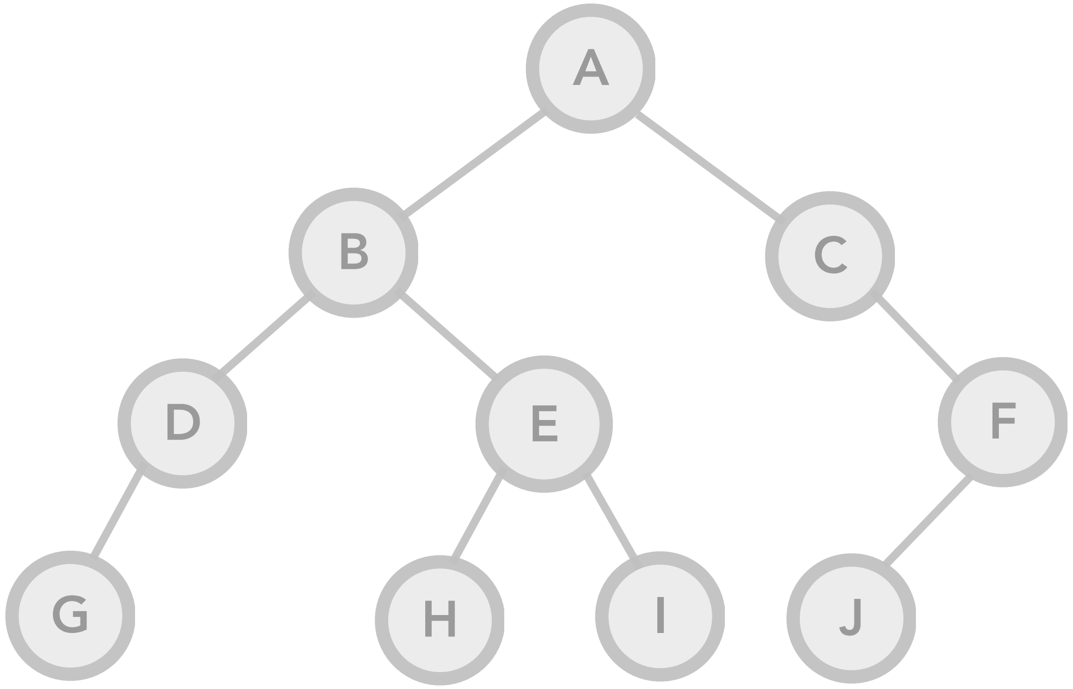
Console

```
A B C D E F G H I
```

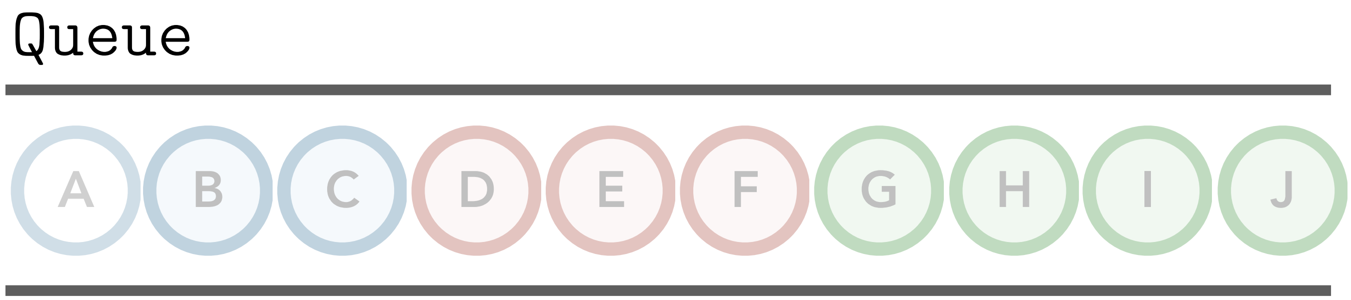
Printing the Tree (level-by-level)

Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



Remove **J**



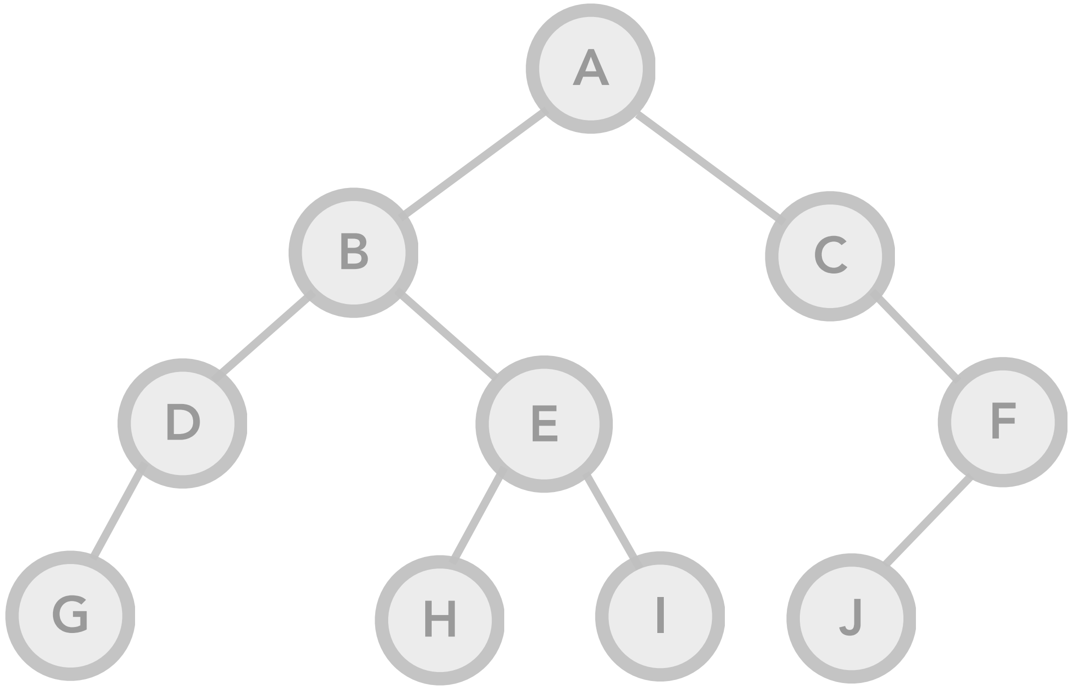
Console

```
A B C D E F G H I J
```

Printing the Tree (level-by-level)

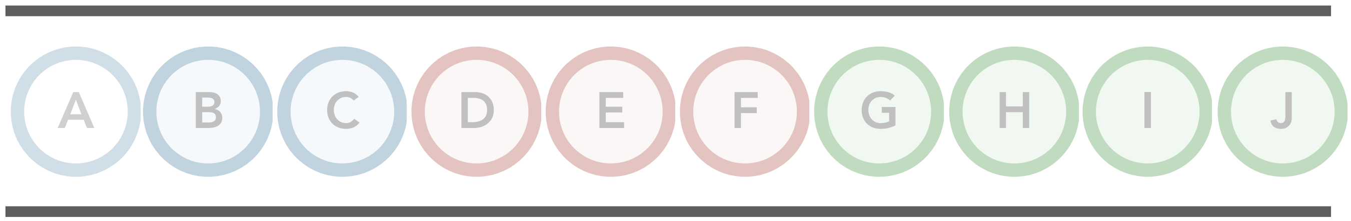
Idea. Maintain a *queue* of the nodes yet to be visited.

Repeat until the queue is empty: Remove a node and add its children.



The queue is empty!

Queue



Console

```
A B C D E F G H I J
```

Printing the Tree (level-by-level)

```
template <class T>
void BST<T>::level_order() const {
    if (is_empty()) return;

    Queue< Node<T>* > queue;
    queue.enqueue(root);

    while (!queue.is_empty()) {
        Node<T>* node = queue.dequeue();
        cout << node->val << " ";

        if (node->left  != nullptr) queue.enqueue(node->left);
        if (node->right != nullptr) queue.enqueue(node->right);
    }
}
```

Printing the Tree (level-by-level)

```
template <class T>
void BST<T>::level_order() const {
    if (is_empty()) return;

    Queue< Node<T>* > queue;
    queue.enqueue(root);

    while (!queue.is_empty()) {
        Node<T>* node = queue.dequeue();
        cout << node->val << " ";

        if (node->left  != nullptr) queue.enqueue(node->left);
        if (node->right != nullptr) queue.enqueue(node->right);
    }
}
```

Create a queue of *pointers* to nodes

No need to store complete copies of nodes!

Printing the Tree (level-by-level)

```
template <class T>
void BST<T>::level_order() const {
    if (is_empty()) return;

    Queue< Node<T>* > queue;
    queue.enqueue(root);

    while (!queue.is_empty()) {
        Node<T>* node = queue.dequeue();
        cout << node->val << " ";

        if (node->left  != nullptr) queue.enqueue(node->left);
        if (node->right != nullptr) queue.enqueue(node->right);
    }
}
```

Repeat until the queue is empty

Printing the Tree (level-by-level)

```
template <class T>
void BST<T>::level_order() const {
    if (is_empty()) return;

    Queue< Node<T>* > queue;
    queue.enqueue(root);

    while (!queue.is_empty()) {
        Node<T>* node = queue.dequeue();
        cout << node->val << " ";

        if (node->left  != nullptr) queue.enqueue(node->left);
        if (node->right != nullptr) queue.enqueue(node->right);
    }
}
```

Remove from the queue and print!

Printing the Tree (level-by-level)

```
template <class T>
void BST<T>::level_order() const {
    if (is_empty()) return;

    Queue< Node<T>* > queue;
    queue.enqueue(root);

    while (!queue.is_empty()) {
        Node<T>* node = queue.dequeue();
        cout << node->val << " ";

        if (node->left  != nullptr) queue.enqueue(node->left);
        if (node->right != nullptr) queue.enqueue(node->right);
    }
}
```

Add the children only if they are not null

Printing the Tree (level-by-level)

```
template <class T>
void BST<T>::level_order() const {
    if (is_empty()) return;

    Queue< Node<T>* > queue;
    queue.enqueue(root);

    while (!queue.is_empty()) {
        Node<T>* node = queue.dequeue();
        cout << node->val << " ";

        if (node->left  != nullptr) queue.enqueue(node->left);
        if (node->right != nullptr) queue.enqueue(node->right);
    }
}
```

Add the children only if they are not null

Terminology. Breadth-First Traversal (BFT) = Level-Order Traversal

Printing the Tree (level-by-level)

```
template <class T>
void BST<T>::level_order() const {
    if (is_empty()) return;

    Queue< Node<T>* > queue;
    queue.enqueue(root);

    while (!queue.is_empty()) {
        Node<T>* node = queue.dequeue();
        cout << node->val << " ";

        if (node->left  != nullptr) queue.enqueue(node->left);
        if (node->right != nullptr) queue.enqueue(node->right);
    }
}
```

Add the children only if they are not null

Terminology. Breadth-First Traversal (BFT) = Level-Order Traversal

Right-to-left BFT. Enqueue the right child before the left child.

Printing the Tree (level-by-level)

```
template <class T>
void BST<T>::level_order() const {
    if (is_empty()) return;

    Queue< Node<T>* > queue;
    queue.enqueue(root);

    while (!queue.is_empty()) {
        Node<T>* node = queue.dequeue();
        cout << node->val << " ";

        if (node->left  != nullptr) queue.enqueue(node->left);
        if (node->right != nullptr) queue.enqueue(node->right);
    }
}
```

Add the children only if they are not null

Terminology. Breadth-First Traversal (BFT) = Level-Order Traversal

Right-to-left BFT. Enqueue the right child before the left child.

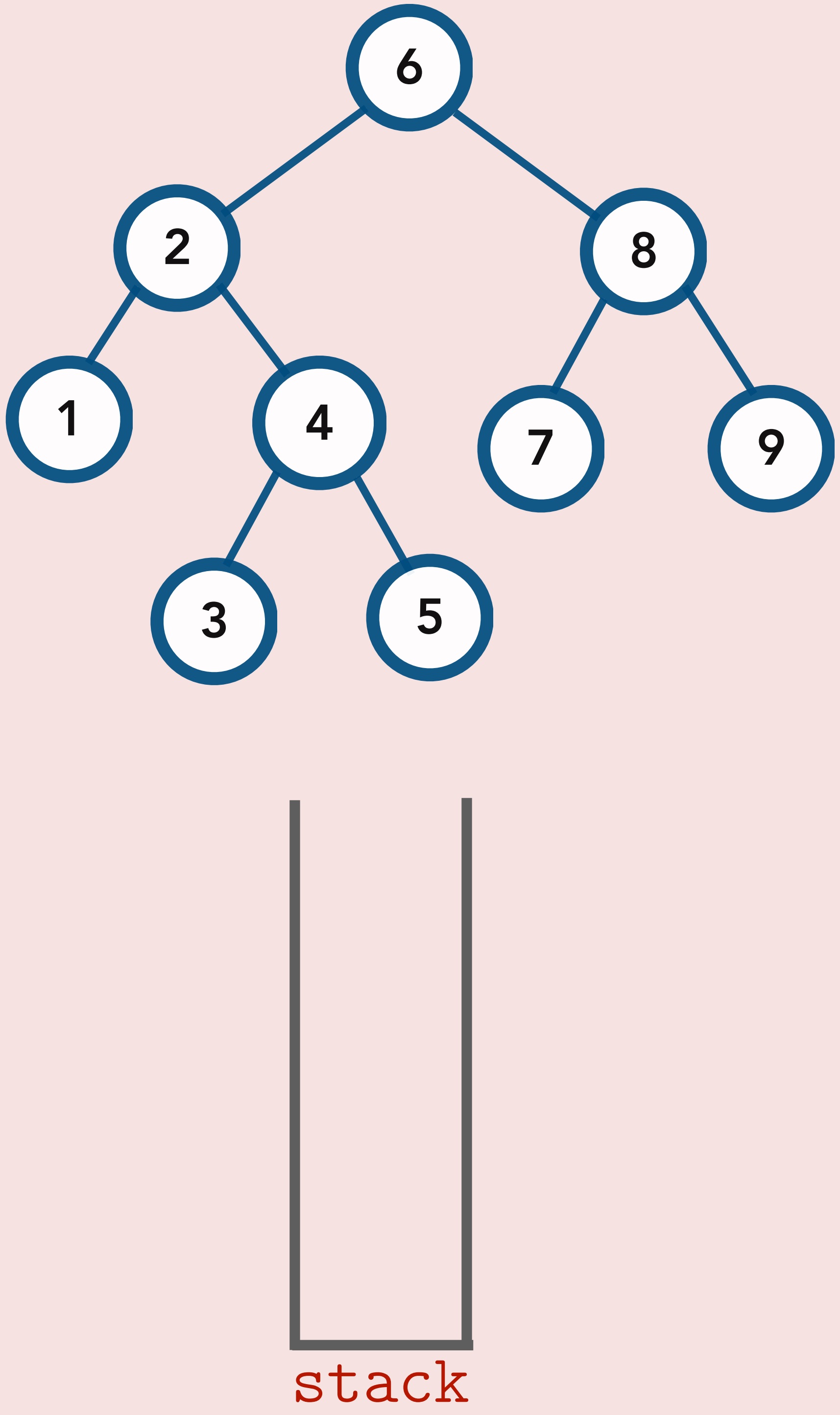
Note. While in this code we print each dequeued node, the BFT is a general-purpose traversal that can be used to go through all the nodes in the tree and perform some operation (e.g. printing the node value, computing and storing the node depth, checking if the node is a leaf, etc.)

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

What does the following function do?

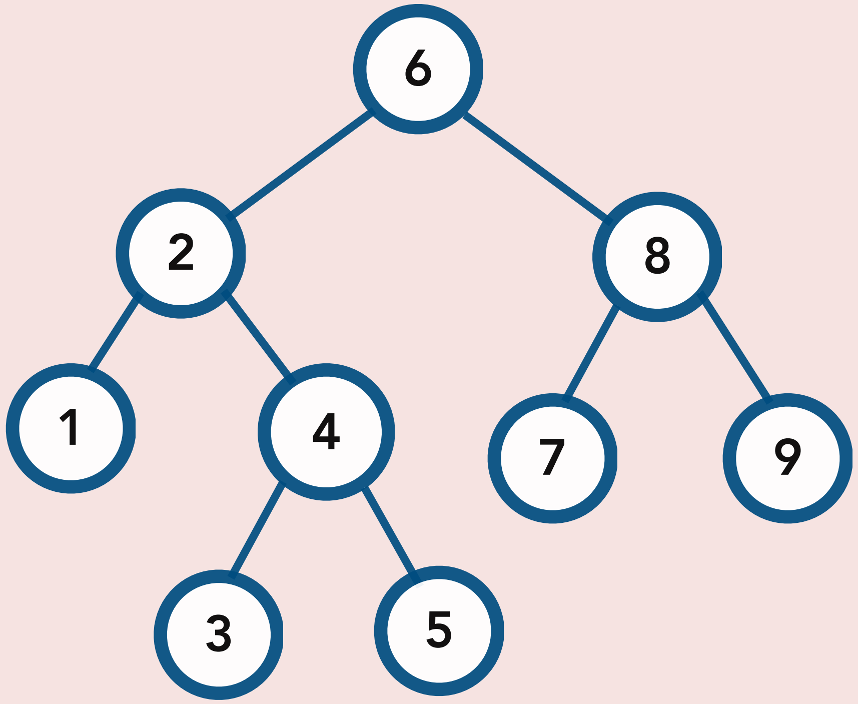
```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

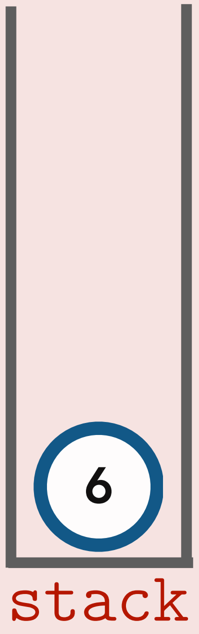


Console

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
  ● stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

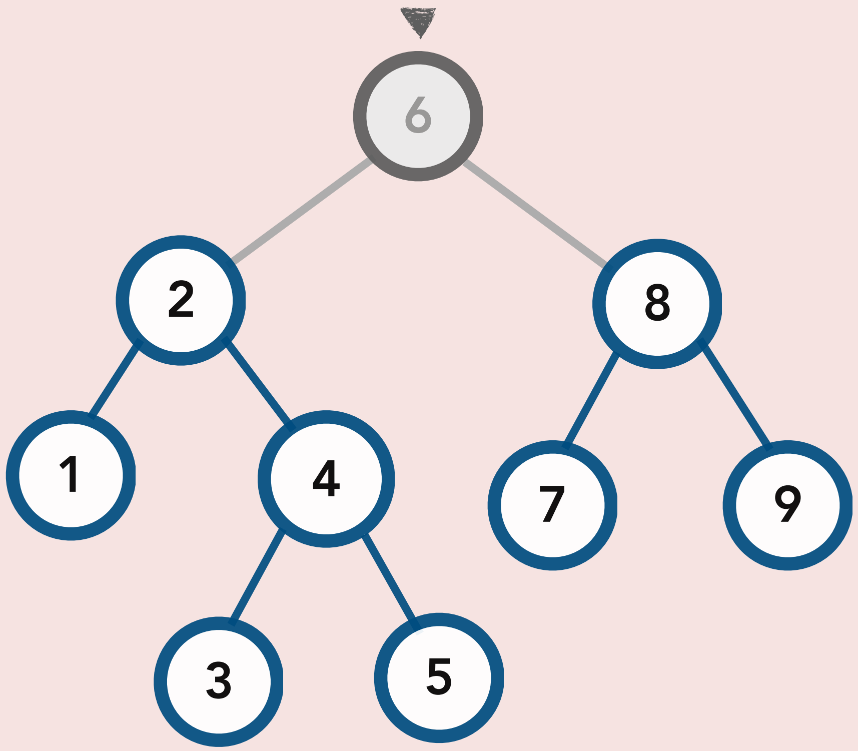


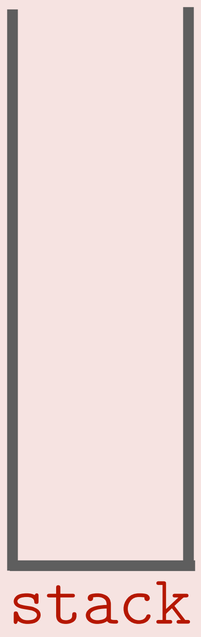


Console

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
      ● Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```



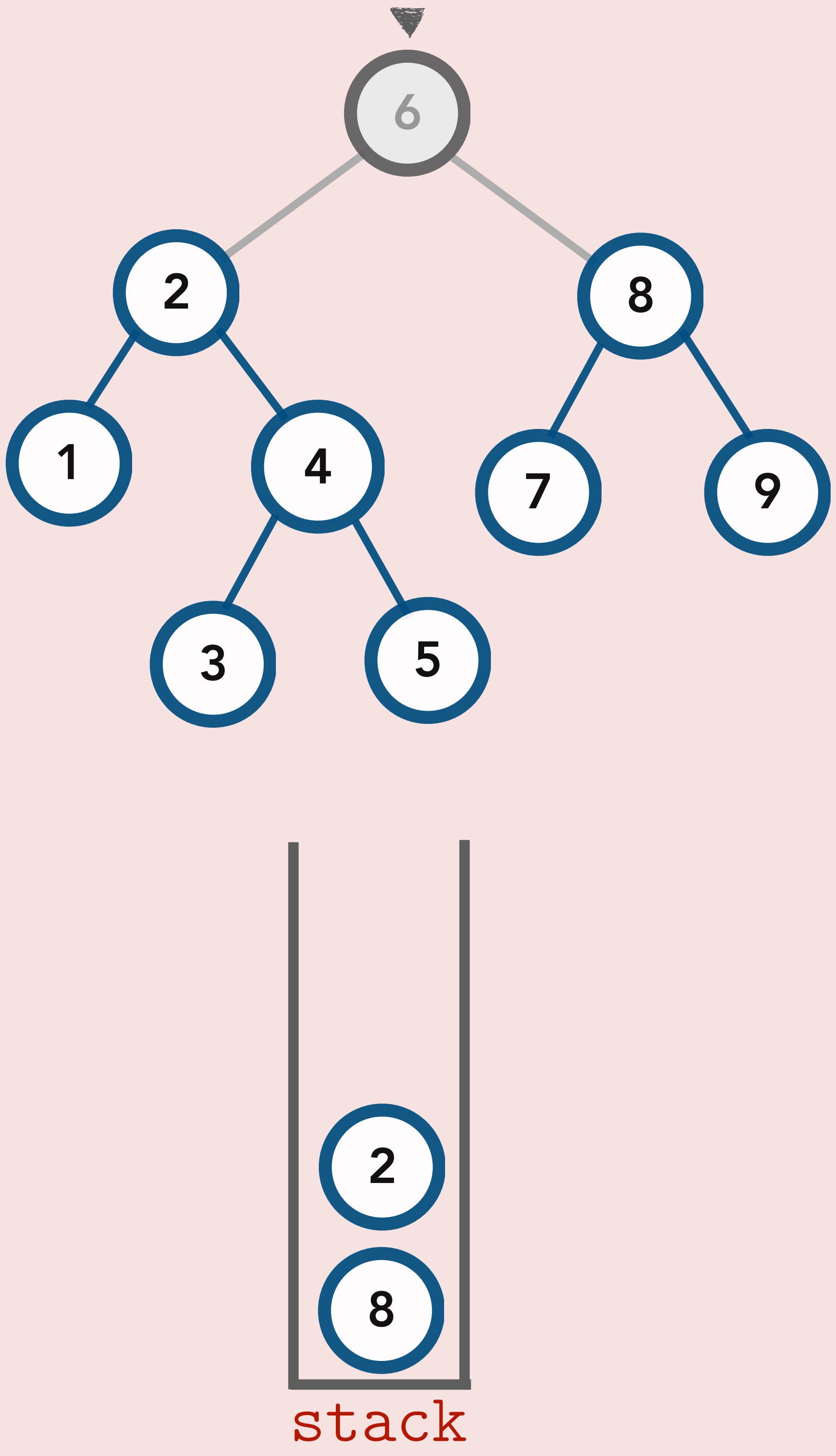


Console

```
6
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

      ● if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

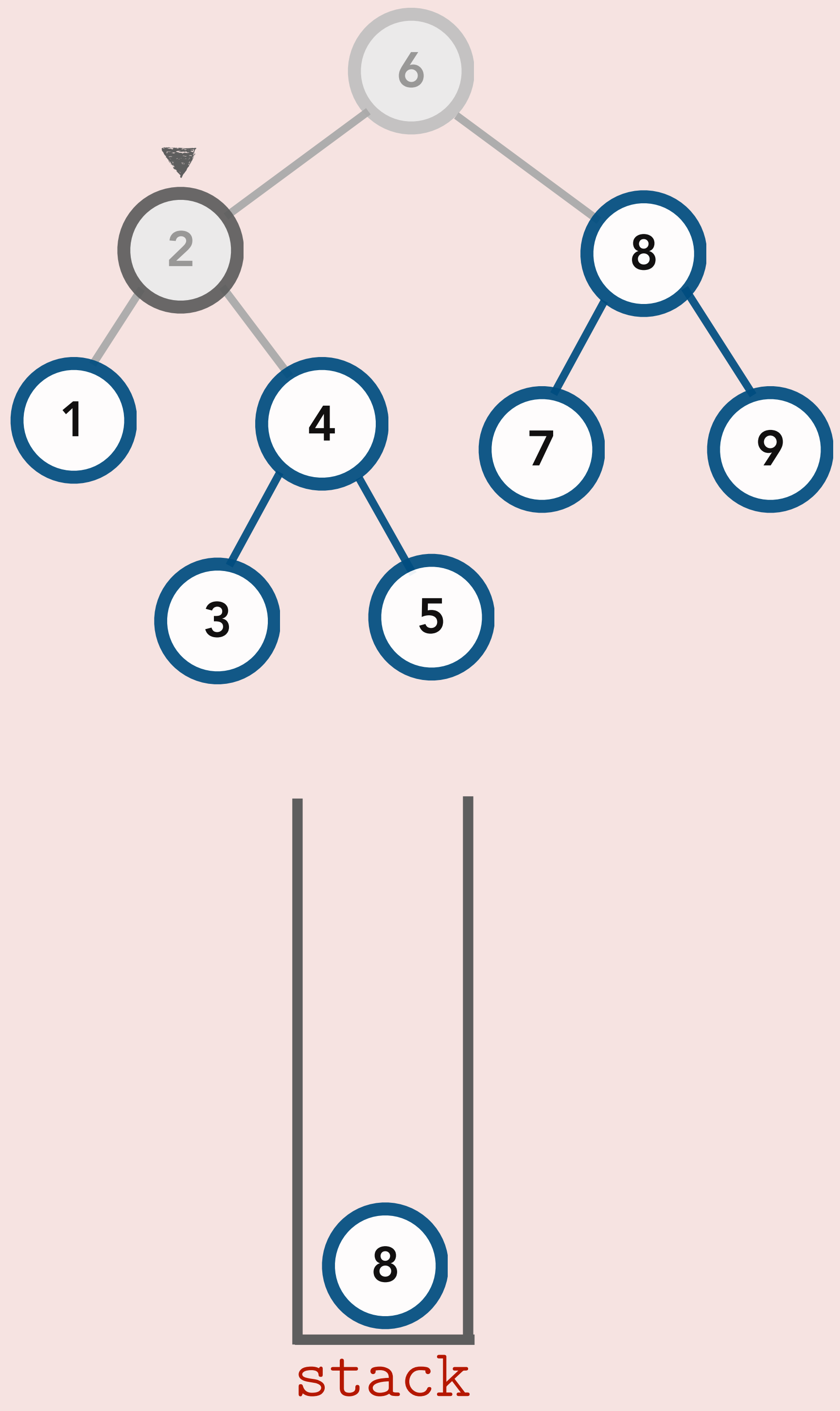


Console

```
6
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
      ● Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

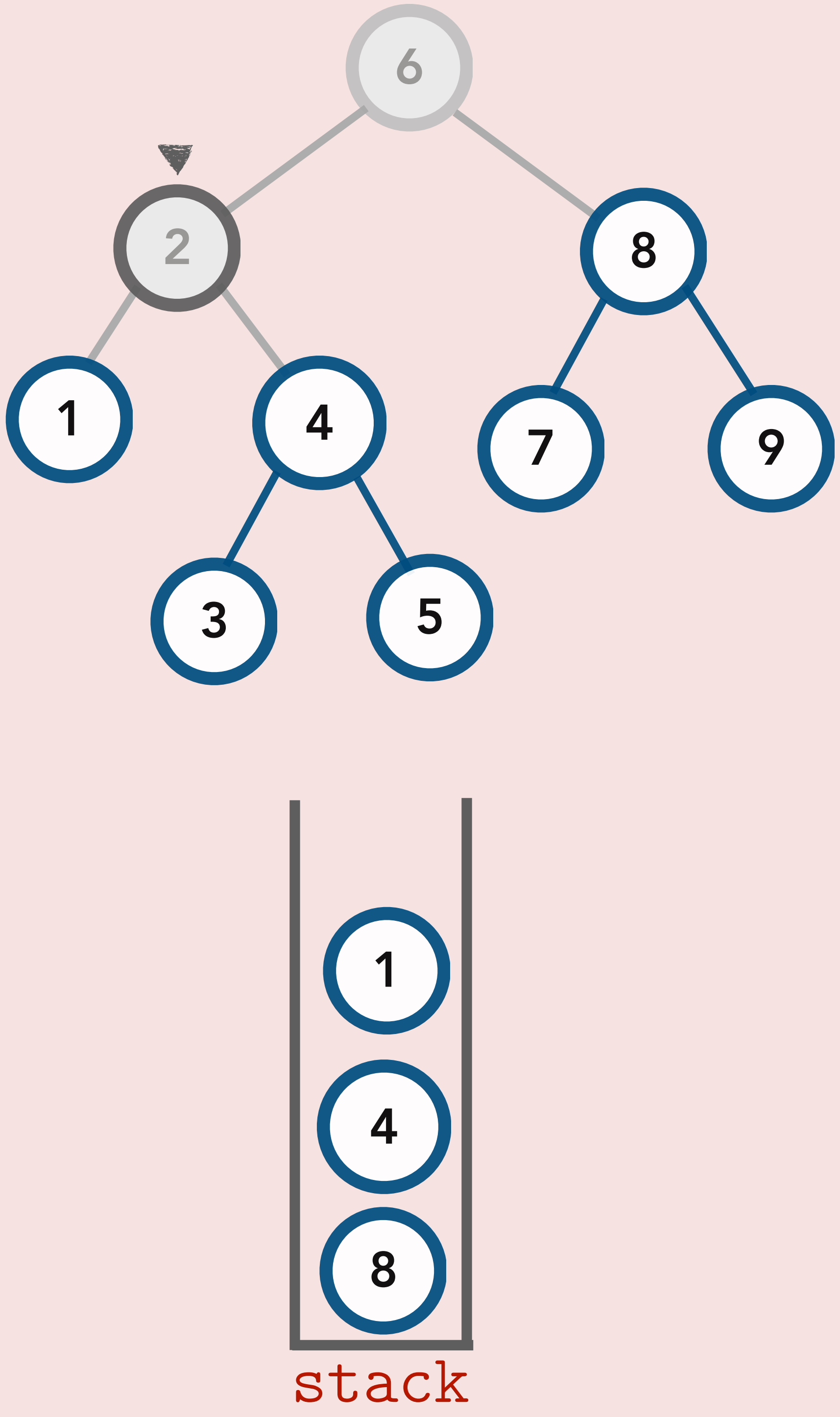


Console

```
6 2
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

      ● if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

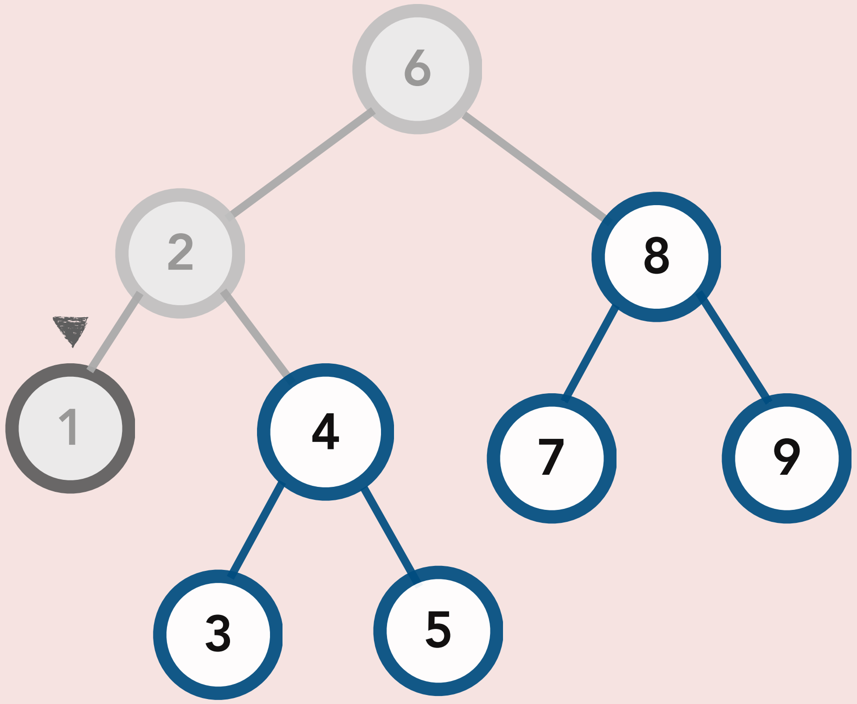


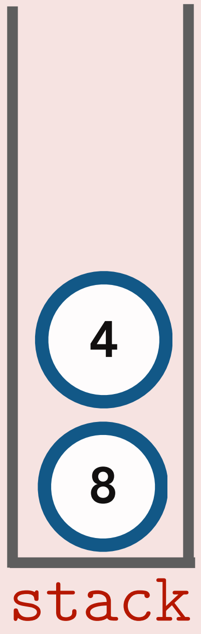
Console

```
6 2
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```



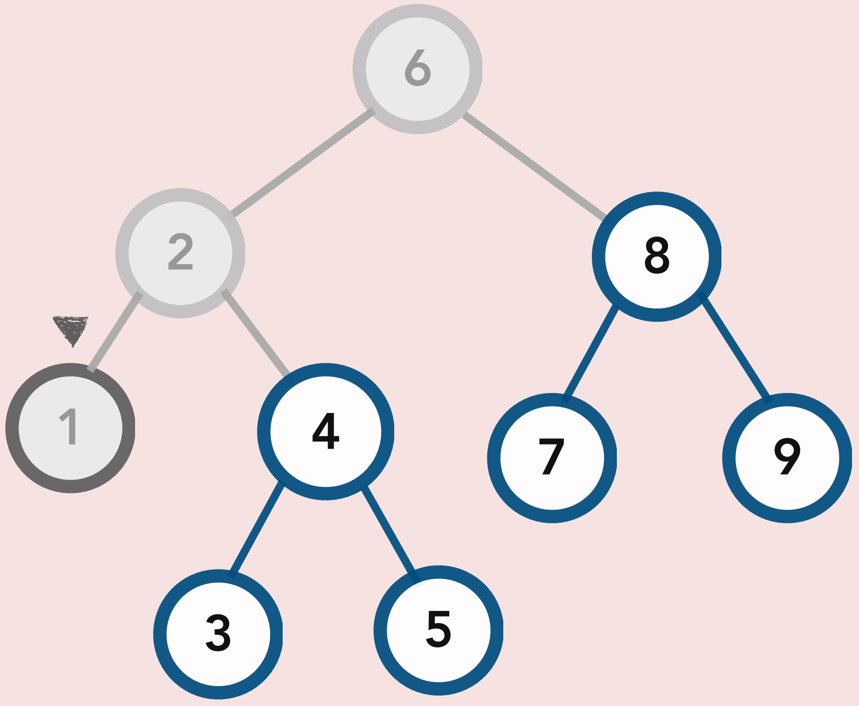


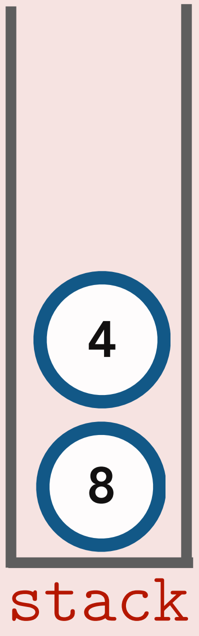
Console

```
6 2 1
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

      ● if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```



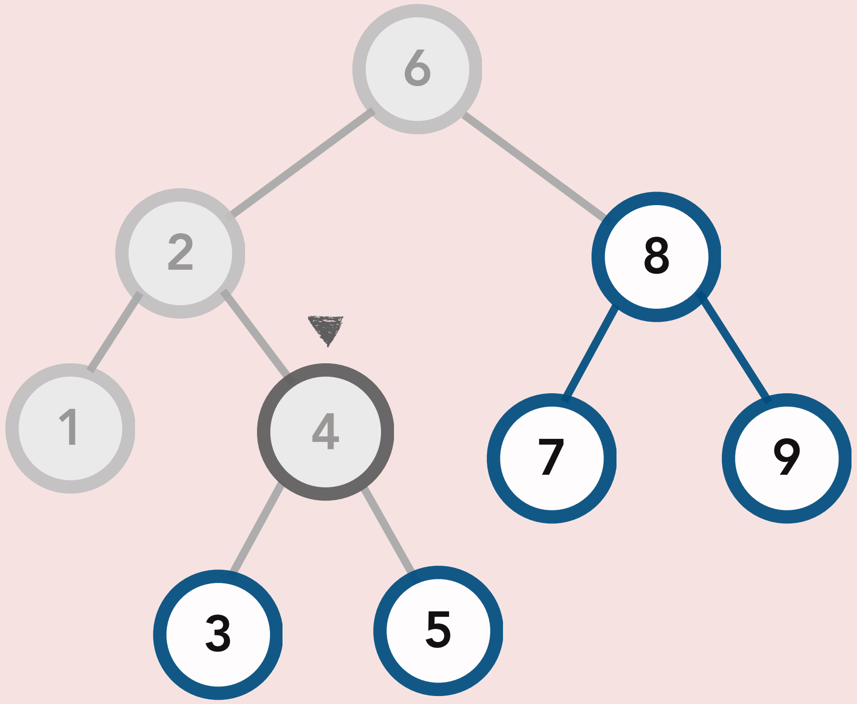


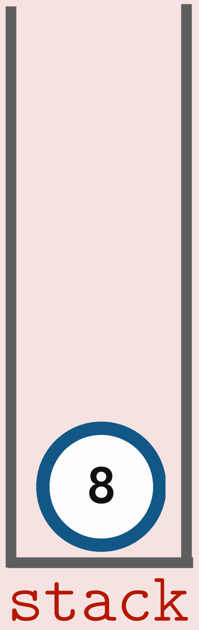
Console

```
6 2 1
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
      ● Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```



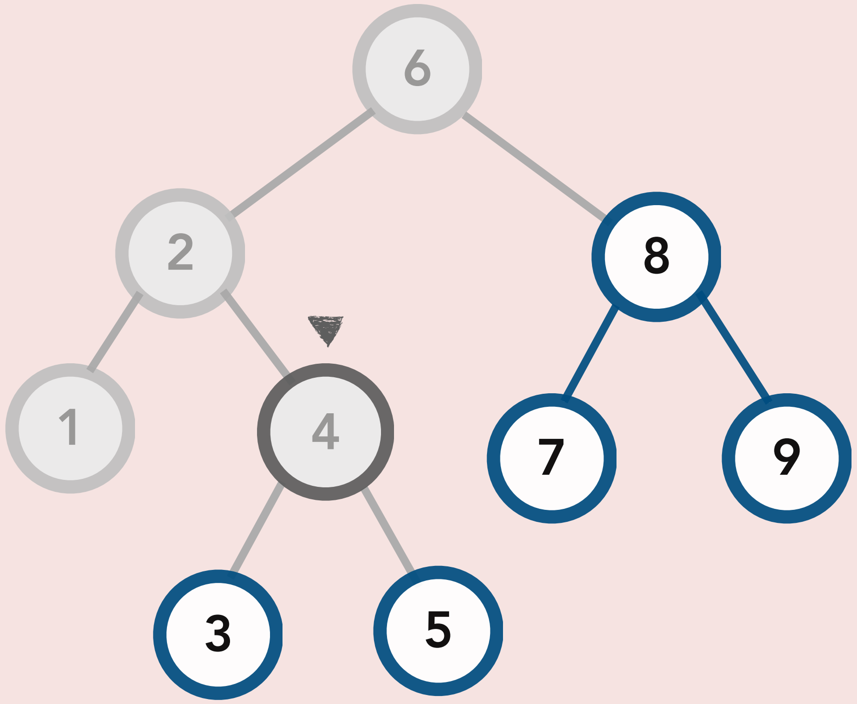


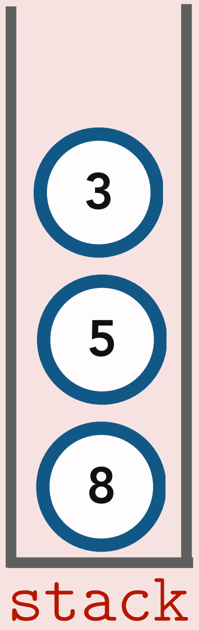
Console

```
6 2 1 4
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

      ● if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```



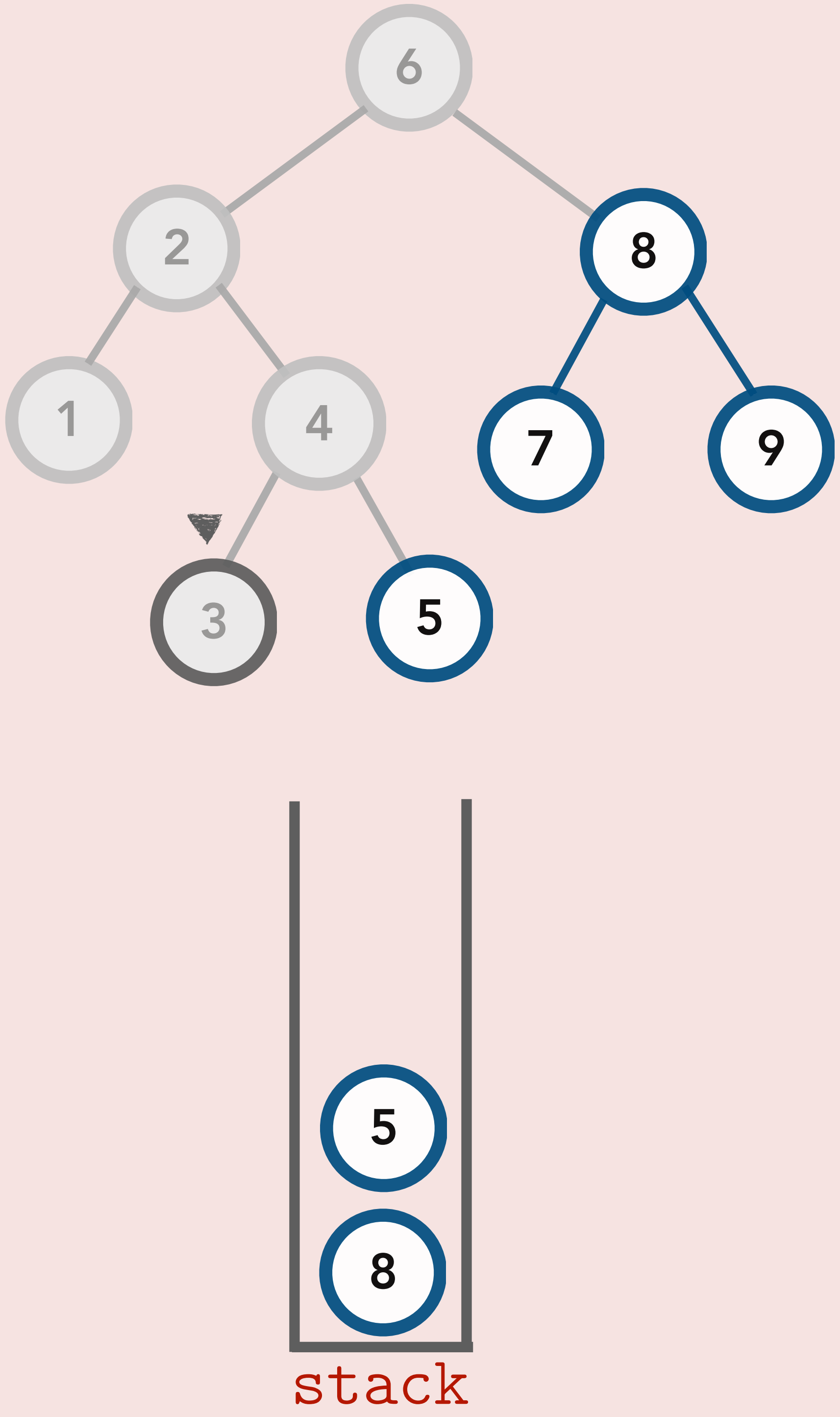


Console

```
6 2 1 4
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
      ● Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

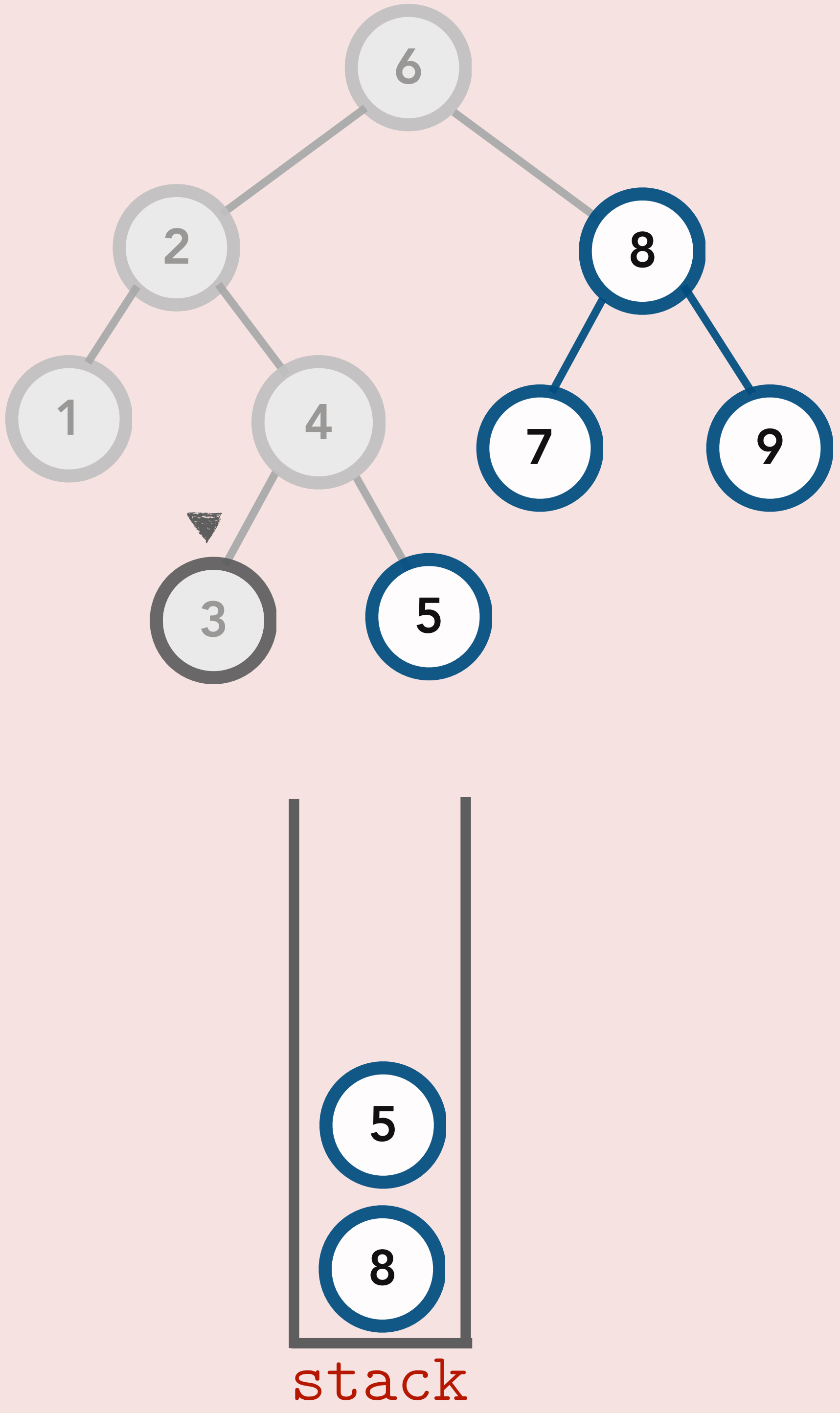


Console

```
6 2 1 4 3
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

      ● if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

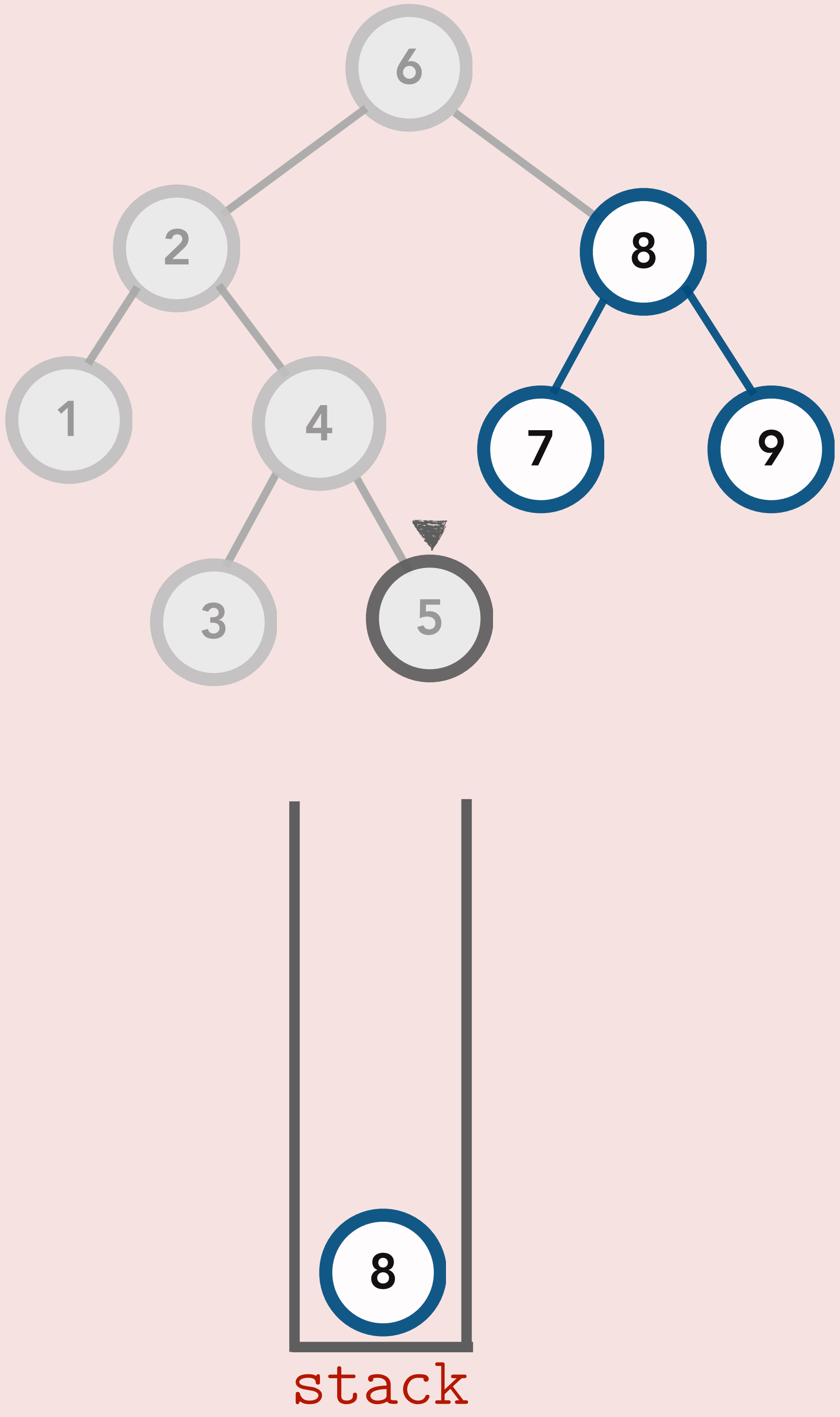


Console

```
6 2 1 4 3
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
      ● Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

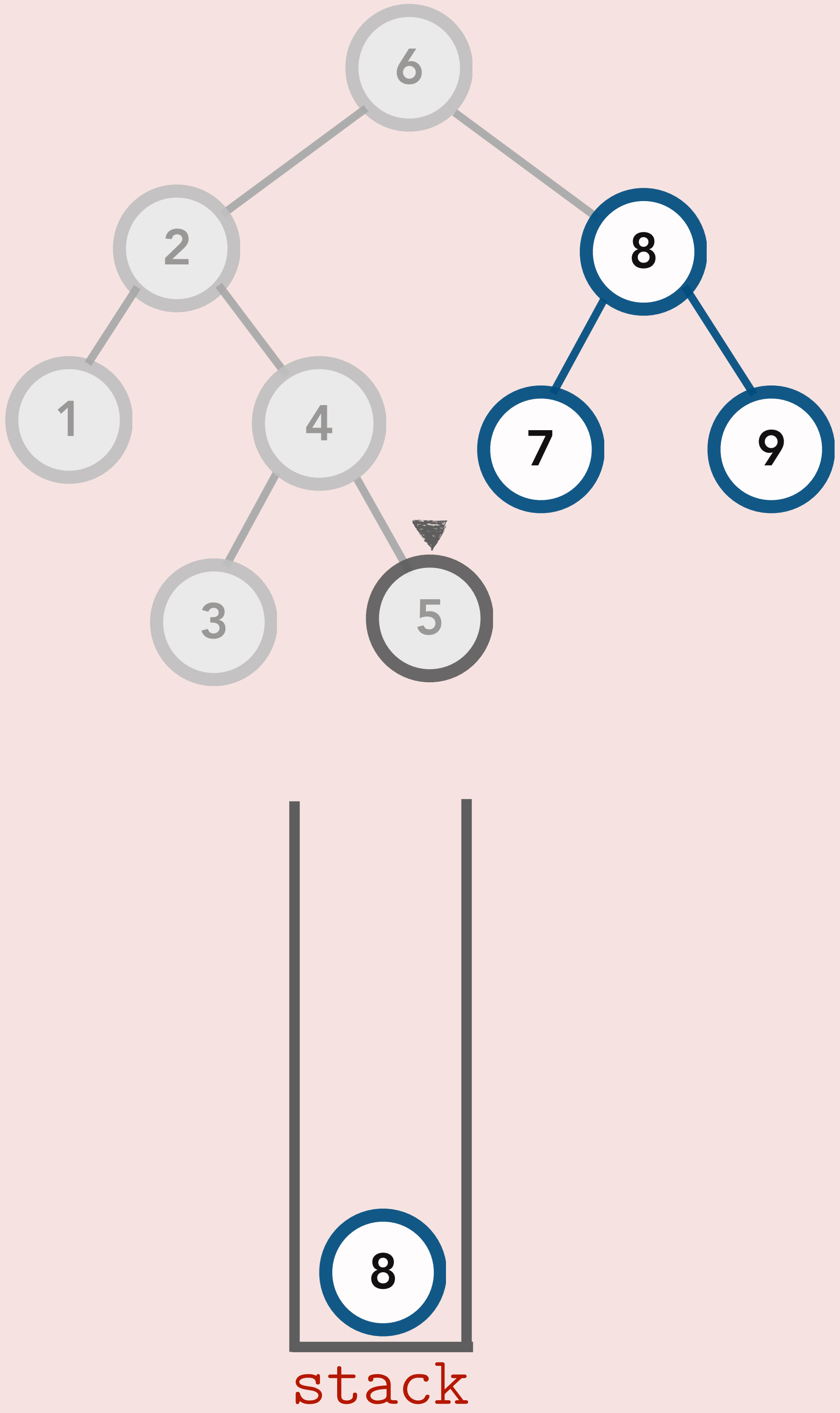


Console

```
6 2 1 4 3 5
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

      ● if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

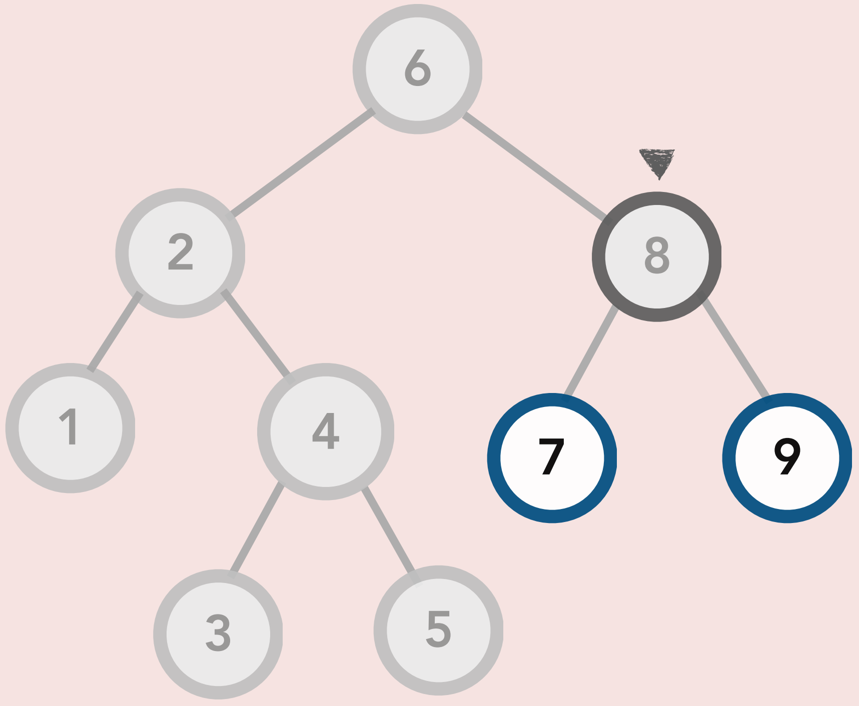


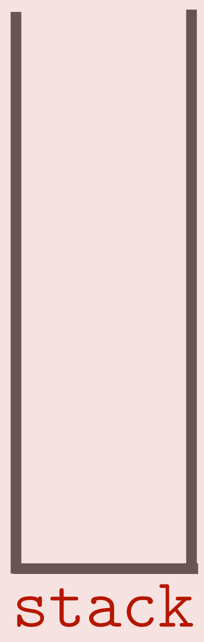
Console

```
6 2 1 4 3 5
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
      ● Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```



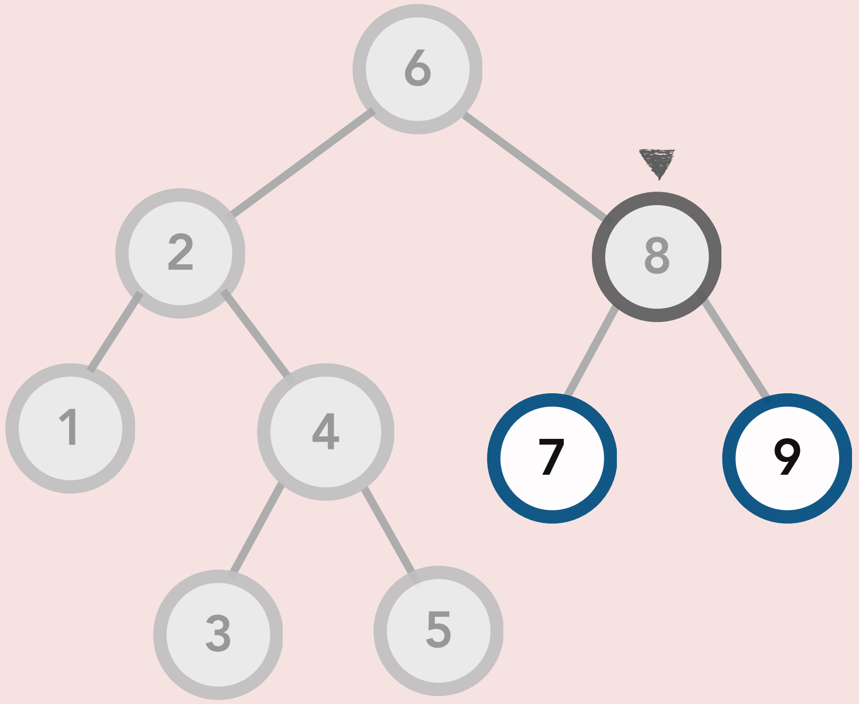


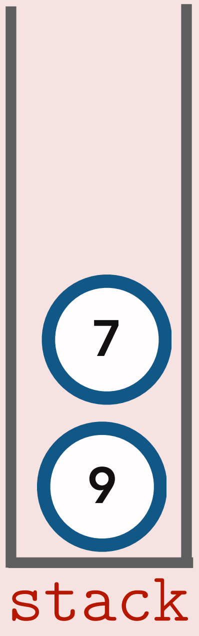
Console

```
6 2 1 4 3 5 8
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

      ● if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```



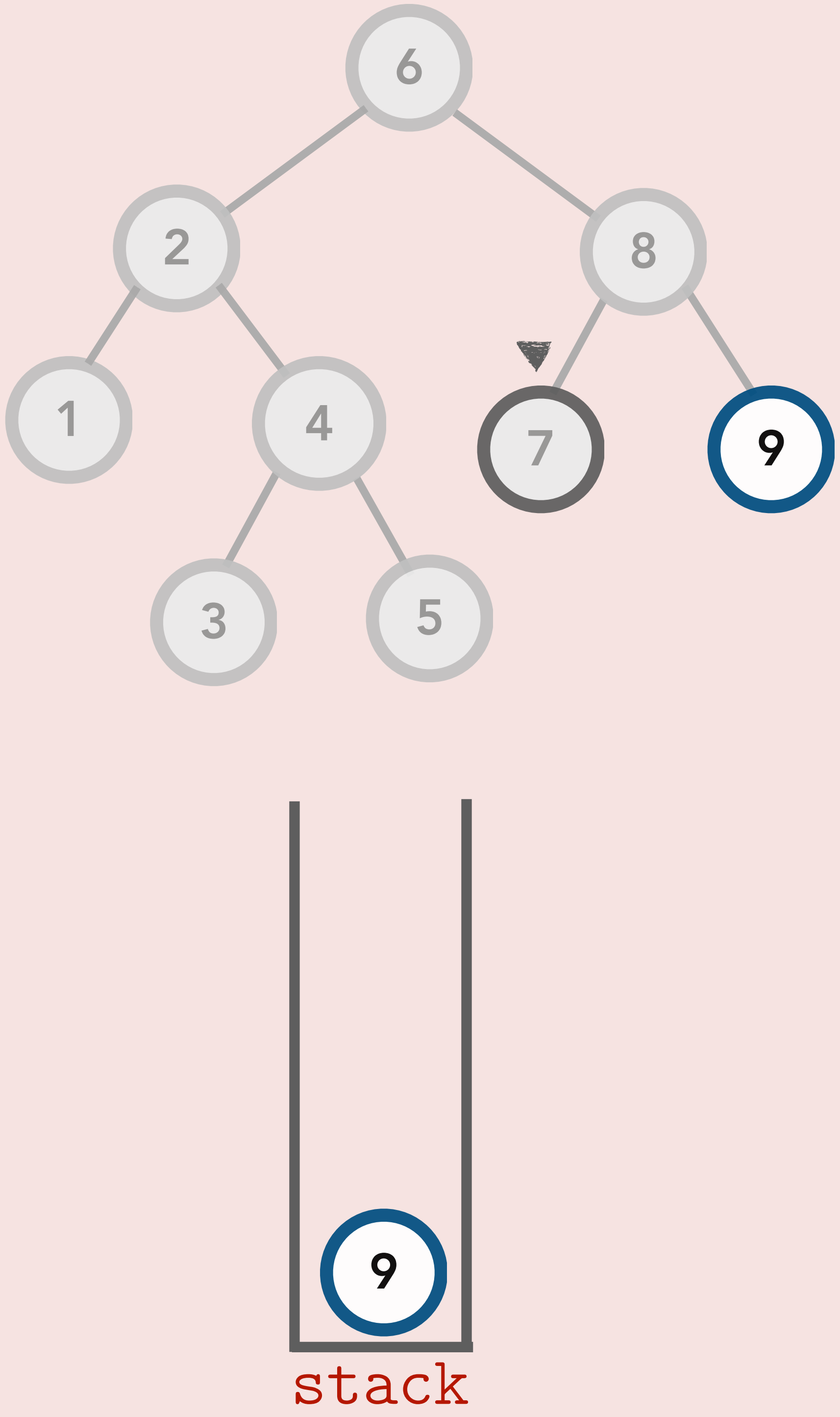


Console

```
6 2 1 4 3 5 8
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
      ● Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

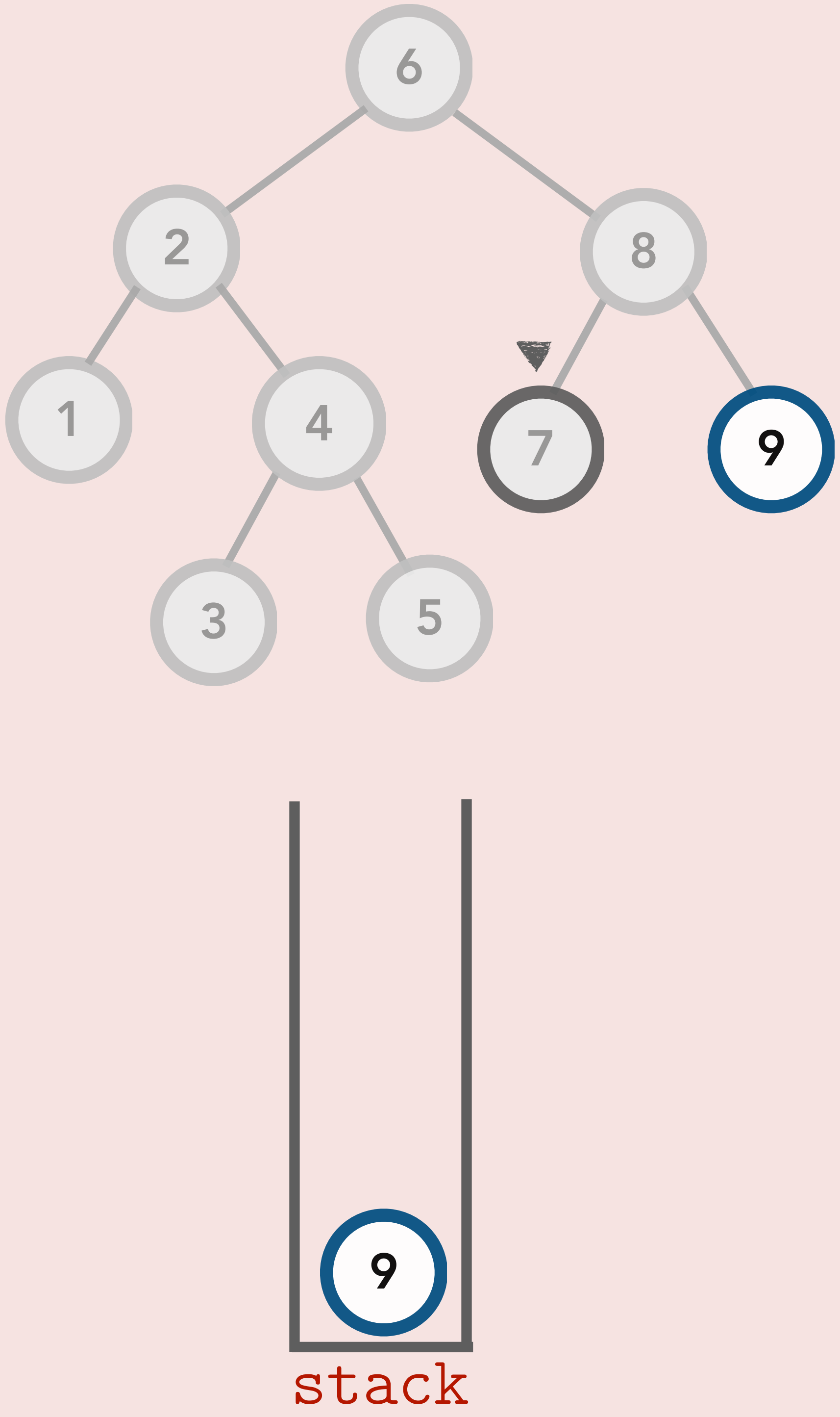


Console

```
6 2 1 4 3 5 8 7
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

      ● if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

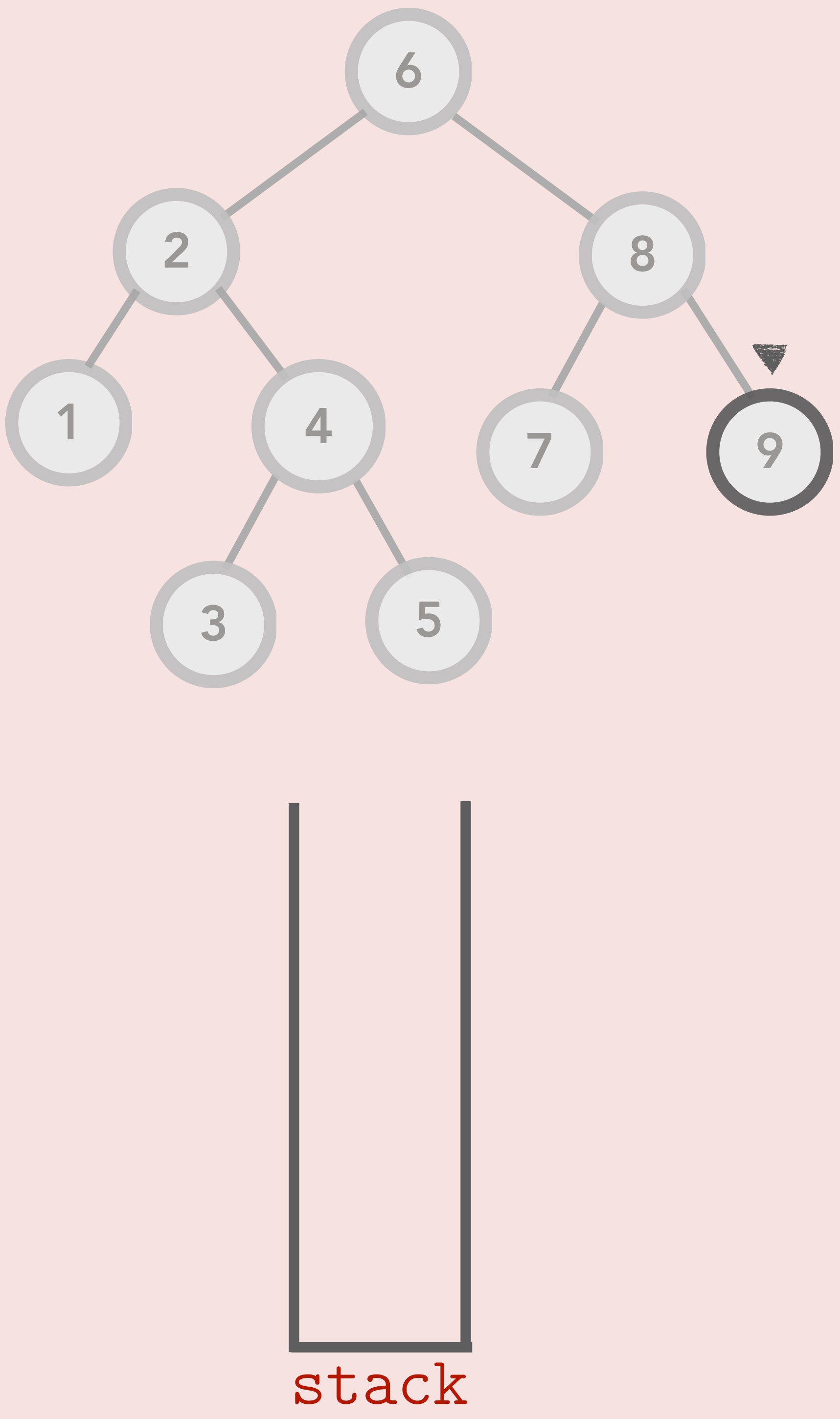


Console

```
6 2 1 4 3 5 8 7
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
     ● Node<T>* node = stack.pop();
       cout << node->val << " ";

       if (node->right != nullptr)
           stack.push(node->right);
       if (node->left  != nullptr)
           stack.push(node->left);
    }
}
```

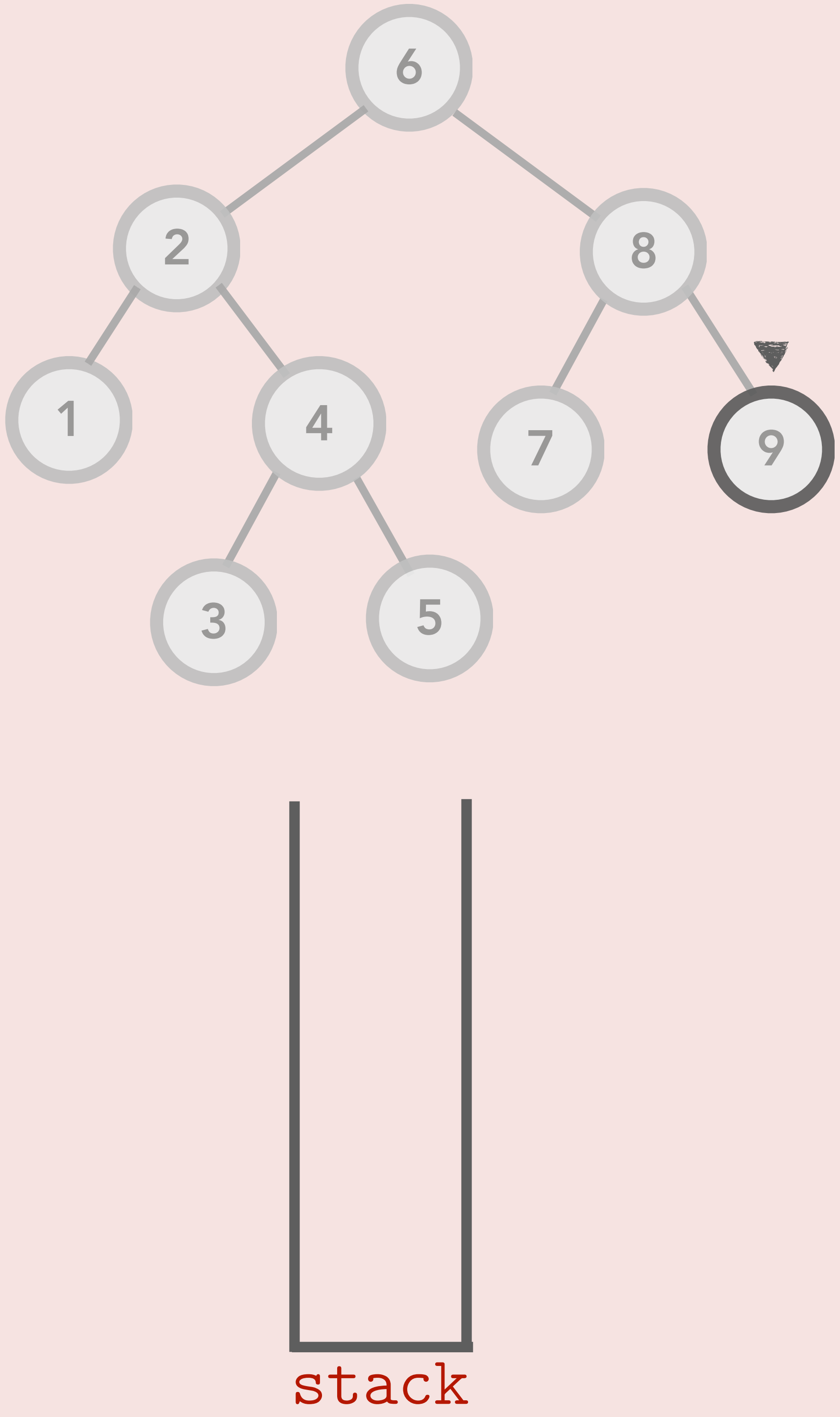


Console

```
6 2 1 4 3 5 8 7 9
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

      ● if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```



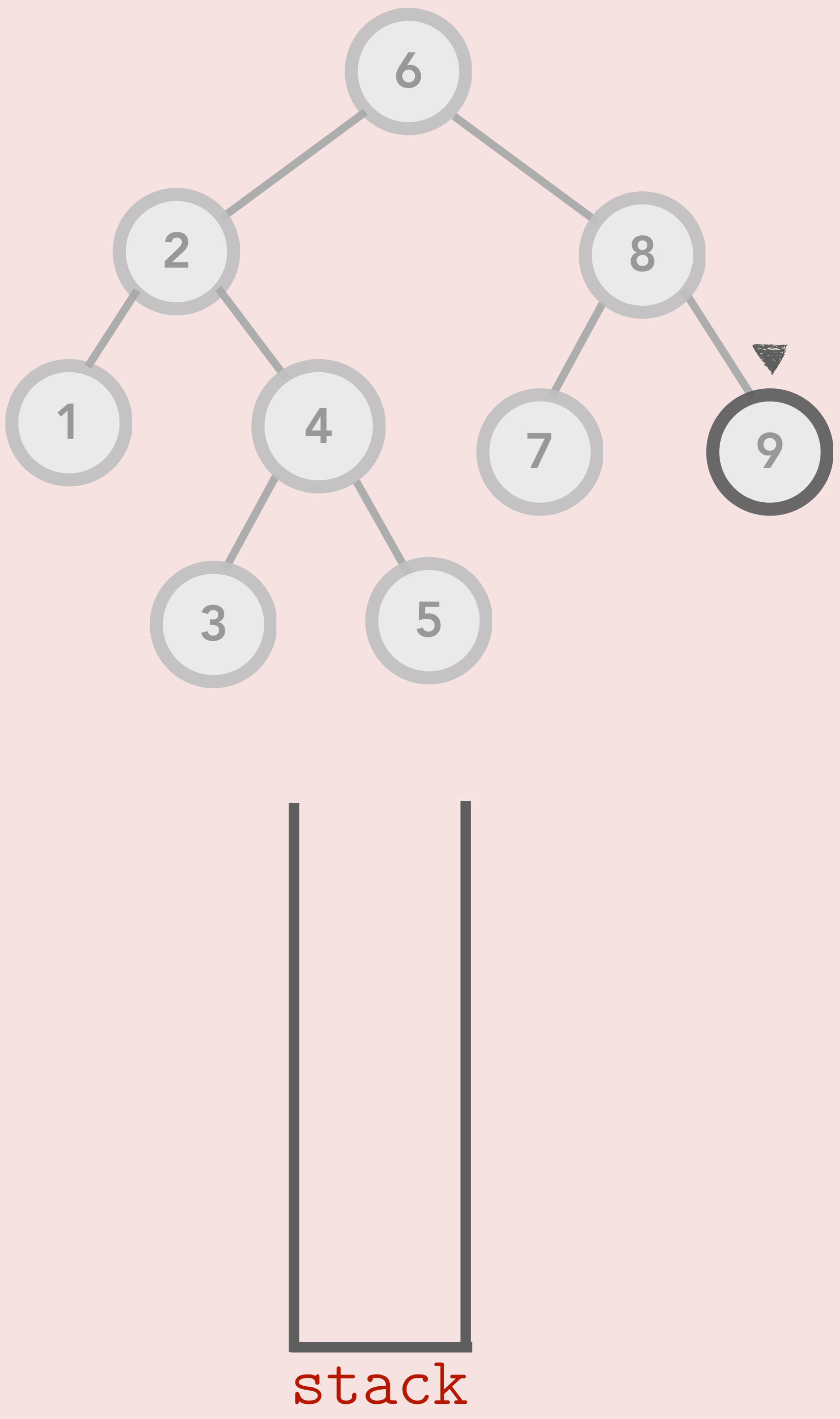
Console

```
6 2 1 4 3 5 8 7 9
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

Pre-order Traversal !



Console

```
6 2 1 4 3 5 8 7 9
```

What does the following function do?

```
template <class T>
void BST<T>::mystery() const {
    if (is_empty()) return;

    Stack< Node<T>* > stack;
    stack.push(root);

    while (!stack.is_empty()) {
        Node<T>* node = stack.pop();
        cout << node->val << " ";

        if (node->right != nullptr)
            stack.push(node->right);
        if (node->left  != nullptr)
            stack.push(node->left);
    }
}
```

Pre-order Traversal !

Console

```
6 2 1 4 3 5 8 7 9
```

More Practice Exercises

1. Store in every node its height.
2. Store in every node its depth.
3. Count the number of nodes in the tree
   or count the number of leafs in the tree.
4. Store in every node the number of nodes in the subtree rooted at that node.
5. Find the maximum in a general binary tree (not a BST)
6. Print the tree in reverse level-order (from the right-most node in the last level to the root).
7. Find the median in a BST (in $O(n)$)
8. Find the median in a balanced BST (in $O(\log n)$) assuming exercise 4 is solved.
9. Remove all the leafs from the tree.
10. Count all the nodes in the last level.

… and many more!

Back to the Set ADT

Problem. Design a data structure to support the following operations:

- `insert(val)   // add val to the set if it is not already in the set.`
- `remove(val)   // remove val from the set of items.`
- `contains(val) // check if val belongs to the set.`

Candidate implementations.

	`insert(val)`	`remove(val)`	`contains(val)`
Unordered DLL	`O(n)`	`O(n)`	`O(n)`
Unordered SLL	`O(n)`	`O(n)`	`O(n)`
Ordered DLL	`O(n)`	`O(n)`	`O(n)`
Ordered SLL	`O(n)`	`O(n)`	`O(n)`
Unordered Array	`O(n)`	`O(n)`	`O(n)`
Ordered Array	`O(n)`	`O(n)`	`O(log n)`
Balanced BST	`O(log n)`	`O(log n)`	`O(log n)`

Winner!

Another ADT: A Map (or Dictionary)

Optional

Problem. Design a data structure to support the following operations:

```
• insert(key, val) // insert a new key-value pair or reset
                   // the current value of they key
• remove(key)      // remove the key and its corresponding value
• get(key)         // return the value corresponding to the key
```

Another ADT: A Map (or Dictionary)

Optional

Problem. Design a data structure to support the following operations:

```
• insert(key, val) // insert a new key-value pair or reset
                   // the current value of they key
• remove(key)      // remove the key and its corresponding value
• get(key)         // return the value corresponding to the key
```

Example Applications.

- A mapping between *words* and their *meanings* (`key` and `val` are `string`)
- A mapping between *usernames* and *passwords* (`key` and `val` are `string`)
- A mapping between *IDs* and *GPAs* (`key` is `string` and `val` is `float`)
- A mapping between *years* and *number of new borns* (`key` and `val` are `int`)

Another ADT: A Map (or Dictionary)

Optional

Problem. Design a data structure to support the following operations:

- `insert(key, val) // insert a new key-value pair or reset`
  `// the current value of they key`
- `remove(key)      // remove the key and its corresponding value`
- `get(key)         // return the value corresponding to the key`

Example Applications.

- A mapping between *words* and their *meanings* (`key` and `val` are `string`)
- A mapping between *usernames* and *passwords* (`key` and `val` are `string`)
- A mapping between *IDs* and *GPAs* (`key` is `string` and `val` is `float`)
- A mapping between *years* and *number of new borns* (`key` and `val` are `int`)

Solution. Use a balanced BST. Modify the `Node` class to have a `key` and a `value`.

- `insert(key, val) // search based on key. If key is found, change the`
  `// current val, else insert a new node ⟶ O(log n)`
- `remove(key)      // same as remove in the set ADT ⟶ O(log n)`
- `get(key)         // search based on key ⟶ O(log n)`