Admin

- Sign up to be a TA! Deadline Friday
- Masks for now in class
- Any questions?
Let’s look at last week’s quiz
Binary Tree
Binary Tree

- **Binary Tree**: A tree where each node has at most 2 children

- The *degree* of a node is the number of children it has. So a binary tree is a tree where all nodes have degree at most 2.

- Let’s see an example of a binary tree. Then, we’ll discuss the `BinaryTree` class that comes with `structure5`
BinaryTree plan

- Each node is stored as a BinaryTree object
BinaryTree plan

• Each node is stored as a BinaryTree object
• Stores the value stored at the node
BinaryTree plan

- Each node is stored as a BinaryTree object
- Stores the value stored at the node
- Stores the parent (of type BinaryTree)
  - The root of the tree stores null for its parent
### BinaryTree plan

- Each node is stored as a `BinaryTree` object
- Stores the value stored at the node
- Stores the parent (of type `BinaryTree`)
  - The root of the tree stores `null` for its parent
- Stores the left and right children (both are of type `BinaryTree`)
  - If either doesn’t exist, points to an *empty node* (similar to dummy nodes)
  - Children of an empty node *point to the node itself*
  - There are other ways to implement missing children in a binary tree; this is just one
BinaryTree plan

- Each node is stored as a BinaryTree object
- Stores the value stored at the node
- Stores the parent (of type BinaryTree)
  - The root of the tree stores null for its parent
- Stores the left and right children (both are of type BinaryTree)
  - If either doesn’t exist, points to an empty node (similar to dummy nodes)
  - Children of an empty node point to the node itself
  - There are other ways to implement missing children in a binary tree; this is just one
- Let’s take a look at the code for BinaryTree
BinaryTree plan

- Each node is stored as a BinaryTree object
- Stores the value stored at the node
- Stores the parent (of type BinaryTree)
  - The root of the tree stores `null` for its parent
- Stores the left and right children (both are of type BinaryTree)
  - If either doesn’t exist, points to an empty node (similar to dummy nodes)
  - Children of an empty node point to the node itself
  - There are other ways to implement missing children in a binary tree; this is just one
- Let’s take a look at the code for BinaryTree
- Now, let’s look at how we can evaluate a tree of expressions stored in a BinaryTree
• The *size* of a tree is the number of nodes it contains
More Binary Tree Vocabulary: Height

- The **size** of a tree is the number of nodes it contains.

- The **depth** of a node \( n \) is the number of edges between \( n \) and the root.
More Binary Tree Vocabulary: Height

- The **size** of a tree is the number of nodes it contains.

- The **depth** of a node \( n \) is the number of edges between \( n \) and the root.

- The **height** of a tree is the largest depth of any node in the tree.
Binary Tree Practice

- How can we calculate the size of a binary tree?
Binary Tree Practice

- How can we calculate the size of a binary tree?
  - Hint: use recursion!
Binary Tree Practice

• How can we calculate the size of a binary tree?
  • Hint: use recursion!
  • Let’s look at how the BinaryTree class implements this
Binary Tree Practice

• How can we calculate the size of a binary tree?
  • Hint: use recursion!
  • Let’s look at how the BinaryTree class implements this

• How can we calculate the height of a binary tree?
Binary Tree Practice

• How can we calculate the size of a binary tree?
  • Hint: use recursion!
  • Let’s look at how the BinaryTree class implements this

• How can we calculate the height of a binary tree?
  • Recursion again!
Binary Tree Practice

• How can we calculate the size of a binary tree?
  • Hint: use recursion!
  • Let’s look at how the BinaryTree class implements this

• How can we calculate the height of a binary tree?
  • Recursion again!
  • Let’s look at how this is implemented
Correctness on Trees
Proving Algorithms Correct

- How can we prove that one of these algorithms is correct? (Let’s say the \texttt{size()} method.)
Proving Algorithms Correct

- How can we prove that one of these algorithms is correct? (Let’s say the `size()` method.)
- It’s a recursive algorithm, so: induction!

- Let’s show that `size()` correctly returns the number of nodes in the tree using induction.
- What should our induction be on?
  - The number of nodes in the tree
- To show: for any tree with \( n \geq 0 \) nodes, `size()` correctly returns \( n \).
- Do we want strong induction or weak induction? What will our proof look like?
  - Answer: strong induction. When proving correctness for a method call of size \( n + 1 \), the recursive calls may be on a tree of size less than \( n \).
Proving Algorithms Correct

- How can we prove that one of these algorithms is correct? (Let’s say the `size()` method.)

- It’s a recursive algorithm, so: induction!

- Let’s show that `size()` correctly returns the number of nodes in the tree using induction.
• How can we prove that one of these algorithms is correct? (Let’s say the `size()` method.)

• It’s a recursive algorithm, so: induction!

• Let’s show that `size()` correctly returns the number of nodes in the tree using induction.

• What should our induction be on?
Proving Algorithms Correct

- How can we prove that one of these algorithms is correct? (Let’s say the size() method.)

- It’s a recursive algorithm, so: induction!

- Let’s show that size() correctly returns the number of nodes in the tree using induction.

- What should our induction be on?
  - The number of nodes in the tree
Proving Algorithms Correct

- How can we prove that one of these algorithms is correct? (Let’s say the `size()` method.)
- It’s a recursive algorithm, so: induction!
- Let’s show that `size()` correctly returns the number of nodes in the tree using induction.
- What should our induction be on?
  - The number of nodes in the tree
  - To show: for any tree with \( n \geq 0 \) nodes, `size()` correctly returns \( n \).
How can we prove that one of these algorithms is correct? (Let’s say the size() method.)

It’s a recursive algorithm, so: induction!

Let’s show that size() correctly returns the number of nodes in the tree using induction.

What should our induction be on?

- The number of nodes in the tree
- To show: for any tree with \( n \geq 0 \) nodes, size() correctly returns \( n \).

Do we want strong induction or weak induction? What will our proof look like?
Proving Algorithms Correct

• How can we prove that one of these algorithms is correct? (Let’s say the `size()` method.)

• It’s a recursive algorithm, so: induction!

• Let’s show that `size()` correctly returns the number of nodes in the tree using induction.

• What should our induction be on?
  • The number of nodes in the tree
  • To show: for any tree with \( n \geq 0 \) nodes, `size()` correctly returns \( n \).

• Do we want strong induction or weak induction? What will our proof look like?
• Answer: strong induction. When proving correctness for a method call of size \( n + 1 \), the recursive calls may be on a tree of size less than \( n \).
Base case

- $n = 0$

That means the root node is empty so size returns 0 correctly.
Base case

- $n = \emptyset$

- That means the root node is empty
Base case

- $n = 0$

- That means the root node is empty

- So size returns 0 correctly.
Inductive Hypothesis

- Let’s look back at what we’re trying to prove. (This often helps fill in the inductive hypothesis.)
Inductive Hypothesis

Let’s look back at what we’re trying to prove. (This often helps fill in the inductive hypothesis.)

I.H. (strong induction): There exists some $n$ such that for all $k$ from 0 to $n$, `size()` returns $k$ correctly on all trees of size $k$. 
Inductive step

Let’s look at a tree of size $n+1$ with root $i$. Every node in the tree rooted at $i$ is in the tree rooted at the left child, or in the tree rooted at the right child, or $i$ itself.
Inductive step

- Let’s look at a tree of size $n + 1$ with root $i$. Every node in the tree rooted at $i$ is in the tree rooted at the left child, or in the tree rooted at the right child, or $i$ itself.

- `size()` returns `left.size() + right.size() + 1`
Inductive step

- Let’s look at a tree of size $n + 1$ with root $i$. Every node in the tree rooted at $i$ is in the tree rooted at the left child, or in the tree rooted at the right child, or $i$ itself.

- $\text{size()}$ returns $\text{left.size()} + \text{right.size()} + 1$

- The left child and right child both have size $< n + 1$. Therefore, both $\text{left.size()}$ and $\text{right.size()}$ correctly return the size of the subtree.
Inductive step

- Let’s look at a tree of size $n + 1$ with root $i$. Every node in the tree rooted at $i$ is in the tree rooted at the left child, or in the tree rooted at the right child, or $i$ itself.

- $\text{size()}$ returns $\text{left.size()} + \text{right.size()} + 1$

- The left child and right child both have size $< n + 1$. Therefore, both $\text{left.size()}$ and $\text{right.size()}$ correctly return the size of the subtree

- Putting these together, $\text{size()}$ returns the size of the tree correctly.
Induction on Trees

- You’ll typically want to use strong induction
Induction on Trees

- You’ll typically want to use strong induction

- Induction is often on the size or height of the tree
Induction on Trees

- You’ll typically want to use strong induction

- Induction is often on the size or height of the tree

- Inductive proofs map very closely to correct recursive algorithms
Iterating Over Trees
Let’s say I want to iterate through each of the items in my tree, one at a time.
• Let’s say I want to iterate through each of the items in my tree, one at a time
• In what order should I go through the nodes?
• Let’s say I want to iterate through each of the items in my tree, one at a time

• In what order should I go through the nodes?
  • We say that we *traverse* the tree
Let’s say I want to iterate through each of the items in my tree, one at a time.

In what order should I go through the nodes?
- We say that we *traverse* the tree.

We’ll see four different methods of traversing a tree today.
Let’s say I want to iterate through each of the items in my tree, one at a time.

In what order should I go through the nodes?
- We say that we *traverse* the tree.

We’ll see four different methods of traversing a tree today.

Any ideas?
Pre-order traversal

- *Pre-order traversal*: First we visit the root. Then, we recursively traverse its left child. Then, we recursively traverse its right child.
Pre-order traversal

- **Pre-order traversal**: First we visit the root. Then, we recursively traverse its left child. Then, we recursively traverse its right child.

- Let’s see an example
Pre-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Pre-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Pre-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Pre-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Pre-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Pre-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Post-order traversal

- **Post-order traversal**: First, we recursively traverse the left child of the root. Then, we recursively traverse its right child. Finally, we visit the root.
Post-order traversal

- **Post-order traversal**: First, we recursively traverse the left child of the root. Then, we recursively traverse its right child. Finally, we visit the root.

- Note that **pre-** vs **post-** refers to when we visit the root
Post-order traversal

- *Post-order traversal*: First, we recursively traverse the left child of the root. Then, we recursively traverse its right child. Finally, we visit the root.

- Note that pre- vs post- refers to when we visit the root

- Let’s see an example
Post-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Post-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Post-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Post-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Post-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
nodes that we have already traversed are marked in green. the node we are currently traversing is marked in orange.
Post-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
In-order traversal

- *In-order traversal*: First, we recursively traverse the left child of the root. Then, we visit the root. Then, we recursively traverse its right child.
• **In-order traversal**: First, we recursively traverse the left child of the root. Then, we visit the root. Then, we recursively traverse its right child.

• Visually: in-order scans the tree from left to right.
In-order traversal

- **In-order traversal**: First, we recursively traverse the left child of the root. Then, we visit the root. Then, we recursively traverse its right child.

- Visually: in-order scans the tree from left to right.

  - This is just a mnemonic! The tree traversal depends on its edges, not the way it’s drawn.
In-order traversal

- **In-order traversal**: First, we recursively traverse the left child of the root. Then, we visit the root. Then, we recursively traverse its right child.

- Visually: in-order scans the tree from left to right.
  - This is just a mnemonic! The tree traversal depends on its edges, not the way it’s drawn.

- Let’s see an example
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Level-order traversal

- **Level-order traversal**: We visit all nodes at the same depth from left to right
Level-order traversal

- *Level-order traversal*: We visit all nodes at the same depth from left to right.
- Unlike the other traversals, doesn’t recursively order the children vs the root.
Level-order traversal

- **Level-order traversal**: We visit all nodes at the same depth from left to right.

- Unlike the other traversals, doesn’t recursively order the children vs the root.

- Let’s see an example.
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
A Question

- We saw how to evaluate an expression tree.

We had to traverse all of the tree to evaluate the expression. What kind of traversal was that?
A Question

- We saw how to evaluate an expression tree.
- We had to traverse all of the tree to evaluate the expression. What kind of traversal was that?
Tree Iterators
Implementing Tree Iterators

- Goal: implement the traversals above as an iterator

  - Can do `next()` and `hasNext()` on demand
  - Problem: want to get values on demand (should be updated as the tree is updated)
  - Don't want to traverse the tree, store all tree values, and then dispense them one by one
  - Instead: each call to `next()` should go to the next node in the tree we want to output
  - Challenge: implementing a recursive traversal piece-by-piece
  - To think about: what data structure helps with recursion?
Implementing Tree Iterators

- Goal: implement the traversals above as an iterator
- Can do `next()` and `hasNext()` on demand
Implementing Tree Iterators

- Goal: implement the traversals above as an iterator
- Can do `next()` and `hasNext()` on demand
- Problem: want to get values on demand (should be updated as the tree is updated)
Implementing Tree Iterators

- Goal: implement the traversals above as an iterator
- Can do `next()` and `hasNext()` on demand
- Problem: want to get values on demand (should be updated as the tree is updated)
  - Don’t want to traverse the tree, store all tree values, and then dispense them one by one
Implementing Tree Iterators

- Goal: implement the traversals above as an iterator

- Can do `next()` and `hasNext()` on demand

- Problem: want to get values on demand (should be updated as the tree is updated)
  - Don’t want to traverse the tree, store all tree values, and then dispense them one by one
  - Instead: each call to `next()` should go to the next node in the tree we want to output
Implementing Tree Iterators

- Goal: implement the traversals above as an iterator
- Can do `next()` and `hasNext()` on demand
- Problem: want to get values on demand (should be updated as the tree is updated)
  - Don’t want to traverse the tree, store all tree values, and then dispense them one by one
  - Instead: each call to `next()` should go to the next node in the tree we want to output
- Challenge: implementing a recursive traversal piece-by-piece
Implementing Tree Iterators

- Goal: implement the traversals above as an iterator
- Can do `next()` and `hasNext()` on demand
- Problem: want to get values on demand (should be updated as the tree is updated)
  - Don’t want to traverse the tree, store all tree values, and then dispense them one by one
  - Instead: each call to `next()` should go to the next node in the tree we want to output
- Challenge: implementing a recursive traversal piece-by-piece
- To think about: what data structure helps with recursion?
Pre-order traversal

- Visits the node, then recursively traverses the left child, then the right child

![Pre-order traversal diagram]

- Keep track of the current node we're traversing
- What happens when we hit a leaf?
- Could backtrack by following pointers; might get confusing
- Instead: maintain nodes to visit on a stack!
Pre-order traversal

- Visits the node, then recursively traverses the left child, then the right child
- Keep track of the current node we’re traversing
Pre-order traversal

- Visits the node, then recursively traverses the left child, then the right child
- Keep track of the current node we’re traversing
- What happens when we hit a leaf?
Pre-order traversal

- Visits the node, then recursively traverses the left child, then the right child
- Keep track of the current node we’re traversing
- What happens when we hit a leaf?
- Could backtrack by following pointers; might get confusing
Pre-order traversal

• Visits the node, then recursively traverses the left child, then the right child
• Keep track of the current node we’re traversing
• What happens when we hit a leaf?
• Could backtrack by following pointers; might get confusing
• Instead: maintain nodes to visit on a stack!
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we still need to traverse.
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we *still need to traverse*

- So `next()`:
  - pops the top item off the stack
  - Stores its value to be returned
  - Pushes its right child onto the stack if nonempty
  - Pushes its left child onto the stack if nonempty

- `hasNext()`?
  - Just returns if the stack is empty
Pre-order traversal

- Stack maintains the non-empty BinaryTree\(<E>\) objects that we *still need to traverse*

- So `next()`:
  - *pops* the top item off the stack
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we still need to traverse

- So `next()`:
  - pops the top item off the stack
  - Stores its value to be returned
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we still need to traverse

- So `next()`:
  - **pops** the top item off the stack
  - Stores its value to be returned
  - Pushes its right child onto the stack if nonempty

  hasNext()?
  - Just returns if the stack is empty
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we *still need to traverse*

- So `next()`:
  - *pops* the top item off the stack
  - Stores its value to be returned
  - Pushes its right child onto the stack if nonempty
  - Pushes its left child onto the stack if nonempty
Pre-order traversal

- Stack maintains the non-empty BinaryTree\<E\> objects that we *still need to traverse*

- So `next()`:
  - **pops** the top item off the stack
  - Stores its value to be returned
  - Pushes its right child onto the stack if nonempty
  - Pushes its left child onto the stack if nonempty

- `hasNext()`?
Pre-order traversal

- Stack maintains the non-empty BinaryTree<E> objects that we **still need to traverse**

- So next():
  - **pops** the top item off the stack
  - Stores its value to be returned
  - Pushes its right child onto the stack if nonempty
  - Pushes its left child onto the stack if nonempty

- hasNext()?
  - Just returns if the stack is empty
In-order traversal

- A little less clear how to keep the stack: want to output the root only after the left side is completed; then output the right side
In-order traversal

- A little less clear how to keep the stack: want to output the root only after the left side is completed; then output the right side
- In other words: want to output the root after the left child has been completely traversed
In-order traversal

- A little less clear how to keep the stack: want to output the root only after the left side is completed; then output the right side
- In other words: want to output the root after the left child has been completely traversed
- Seems like we want the root at the very bottom of the stack. We’ll keep it at the bottom of the stack as we traverse the left subtree; then when we pop the root off we’ll output its value and traverse the right child
In-order traversal

- A little less clear how to keep the stack: want to output the root only after the left side is completed; then output the right side.
- In other words: want to output the root after the left child has been completely traversed.
- Seems like we want the root at the very bottom of the stack. We’ll keep it at the bottom of the stack as we traverse the left subtree; then when we pop the root off we’ll output its value and traverse the right child.
- Nice idea, but it takes some care. Let’s be a bit more specific.
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on

- On a call to `next()`:
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on

- On a call to `next()`:
  - pop node from stack; store its value to be returned
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on

- On a call to `next()`:
  - pop node from stack; store its value to be returned
  - Push its right child onto the stack if nonempty
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on

- On a call to `next()`:
  - pop node from stack; store its value to be returned
  - Push its right child onto the stack if nonempty
  - Push the left child of this right child onto the stack, and its left child, and so on
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on

- On a call to `next()`:
  - pop node from stack; store its value to be returned
  - Push its right child onto the stack if nonempty
  - Push the left child of this right child onto the stack, and its left child, and so on

- `hasNext()`: return if the stack is nonempty
In-order traversal

- To begin: push root onto the stack, then push its left child onto the stack, and so on

- On a call to `next()`:
  - pop node from stack; store its value to be returned
  - Push its right child onto the stack if nonempty
  - Push the left child of this right child onto the stack, and its left child, and so on

- `hasNext()`: return if the stack is nonempty

- Let’s look at the code
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type `BinaryTree`

**Stack:** 18 9 5
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the *values* of the nodes, but in reality the objects stored are of type `BinaryTree`

**Stack:** 18 9
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.

Stack: 18 12
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.

Stack: 18
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type \texttt{BinaryTree}

\textbf{Stack:} \texttt{24 22}
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.

Stack: 24
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the *values* of the nodes, but in reality the objects stored are of type `BinaryTree`.

**Stack:** 30 29
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the *values* of the nodes, but in reality the objects stored are of type *BinaryTree*.

**Stack:** 30
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the *values* of the nodes, but in reality the objects stored are of type `BinaryTree`.

**Stack:** 35
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree

Stack:
In-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Stack is labelled with the *values* of the nodes, but in reality the objects stored are of type `BinaryTree`
Post-order traversal

- Same idea as in-order traversal
Post-order traversal

- Same idea as in-order traversal
- Output the node when popping from the stack
Post-order traversal

- Same idea as in-order traversal

- Output the node when popping from the stack

- If you pop a node, and it’s the left child of its parent, push the parent’s right child (and leftmost descendants) onto the stack
Post-order traversal

- Same idea as in-order traversal

- Output the node when popping from the stack

- If you pop a node, and it’s the left child of its parent, push the parent’s right child (and leftmost descendants) onto the stack

- Let’s look at the code
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange.
Level-order traversal

- Level-order traversal is not recursive!
Level-order traversal

• Level-order traversal is not recursive!

• How do we keep track of what nodes to visit next?

Key insight: the order we visit nodes at a given “level” is the same order we visited their parents. So the first parents to be visited have the first children that are visited. Can we use a queue?
Level-order traversal

• Level-order traversal is not recursive!

• How do we keep track of what nodes to visit next?

• Key insight: the order we visit nodes at a given “level” is the same order we visited their parents
Level-order traversal

- Level-order traversal is not recursive!

- How do we keep track of what nodes to visit next?

- Key insight: the order we visit nodes at a given “level” is the same order we visited their parents

- So the $first$ parents to be visited have the $first$ children that are visited
Level-order traversal

- Level-order traversal is not recursive!

- How do we keep track of what nodes to visit next?

- Key insight: the order we visit nodes at a given “level” is the same order we visited their parents

- So the *first* parents to be visited have the *first* children that are visited

- …Can we use a queue?
Level-order iterator

- To begin: push root onto the queue
Level-order iterator

- To begin: push root onto the queue

- next():
Level-order iterator

- To begin: push root onto the queue

- next():
  - Dequeue node off the queue; store its value to be returned
Level-order iterator

• To begin: push root onto the queue

• `next()`: 
  • Dequeue node off the queue; store its value to be returned
  • Enqueue its non-empty children onto the queue
Level-order iterator

• To begin: push root onto the queue

• `next()`:
  • Dequeue node off the queue; store its value to be returned
  • Enqueue its non-empty children onto the queue

• `hasNext()`: return if queue is empty
Level-order iterator

• To begin: push root onto the queue

• `next()`:
  • Dequeue node off the queue; store its value to be returned
  • Enqueue its non-empty children onto the queue

• `hasNext()`: return if queue is empty

• Let’s look at the code
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.

Queue: 9 24
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.

Queue: 24 5 12
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the *values* of the nodes, but in reality the objects stored are of type `BinaryTree`.

**Queue:** 5 12 22 30
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.

Queue: 12 22 30
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree

Queue: 22 30
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.

Queue: 30
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.

Queue: 29 35
Node traversal, Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type `BinaryTree`.

**Queue:** 35
Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the values of the nodes, but in reality the objects stored are of type BinaryTree.
Level-order Traversal

Nodes that we have already traversed are marked in green. The node we are currently traversing is marked in orange. Queue is labelled with the *values* of the nodes, but in reality the objects stored are of type `BinaryTree`.

**Queue:**