Pointer Reinforcement

By: Kevin Chen

Contents of this Presentation

- How Parameters are Passed In
- Example: remove all leaves of a binary search tree
    - Implement using the Look-Ahead Method
    - Implement using the Pointer Reinforcement Method
- Extra Practice Problems

Before We Begin…

We will review how different types of parameters are passed in:

- Primitives
- Objects
- References

How Parameters are Passed

- If a parameter is passed in by **value**, then a copy of it is made for the function call. Any changes to this new copy would not affect the value passed in by the caller of the function.

How Parameters are Passed

- If a parameter is passed in by **value**, then a copy of it is made for the function call. Any changes to this new copy would not affect the value passed in by the caller of the function.
- If a parameter is passed in by **reference**, then a reference to the original value is passed in for the function call. Any changes to the value pointed to by the reference will affect the original value from the caller's side.

Is There Anything Wrong with This Method?

```
public static void increment(int a) {
	a = a + 1;
}
```

Is There Anything Wrong with This Method?

```
public static void increment(int a) {
        a = a + 1;
}
```

Answer: There is an error with this method. Primitives (e.g. int, booleans, chars) are passed in by value. Thus, any changes made within the method would not be seen by the method caller. This method does not actually increment the value of a.

Is There Anything Wrong with This Method?

```
//Assume that the Point class has two attributes: x and y
public static void incrementX(Point point) {
        point.x = point.x + 1;
}
```

Is There Anything Wrong with This Method?

```
//Assume that the Point class has two attributes: x and y
public static void incrementX(Point point) {
	point.x = point.x + 1;
}
```

Answer: There’s nothing wrong with this method. The Point object that the method caller is referencing will be modified by this method, so the value of x will be correctly incremented.

Is There Anything Wrong with This Method?

```
//Assume that the constructor that takes in two parameters: x and y
public static void setToOrigin(Point point) {
        point = new Point(0, 0);
}
```

Is There Anything Wrong with This Method?

```
//Assume that the constructor that takes in two parameters: x and y
public static void setToOrigin(Point point) {
        point = new Point(0, 0);
}
```

Answer: There is an error with this method. References are passed in by value. This means a new reference is created for this method call (though the reference points to the same Point object as the method caller’s reference does). Thus, modifying the reference would not affect the caller’s reference.

Recap

Primitives	Objects	References
Are passed in by value	Are passed in by reference	Are passed in by value

Now…A Problem to Solve

Problem: Write a method called removeAllLeaves() that removes all leaves of a BST

We will first solve this problem using the Look-Ahead Method.

Attempt #1

- Our recursive method will take in a reference to a BSTNode. We’ll refer to this parameter as the current node.

Attempt #1

- Our recursive method will take in a reference to a BSTNode. We’ll refer to this parameter as the current node.
- Our recursive method will first check if the current node is a leaf
    - If the current node is a leaf, then we will remove it by replacing it will null
    - If the current node is not a leaf, then we will recursively call the method on its left and right children.

Attempt #1

- Our recursive method will take in a reference to a BSTNode. We’ll refer to this parameter as the current node.
- Our recursive method will first check if the current node is a leaf
  - If the current node is a leaf, then we will remove it by replacing it will null
  - If the current node is not a leaf, then we will recursively call the method on its left and right children.
- Our public method will call the recursive method, passing in the root.

Attempt #1

- Our recursive method will take in a reference to a BSTNode. We'll refer to this parameter as the current node.
- Our recursive method will first check if the current node is a leaf
    - If the current node is a leaf, then we will remove it by replacing it will null
    - If the current node is not a leaf, then we will recursively call the method on its left and right children.
- Our public method will call the recursive method, passing in the root.

What problems do you see? (Hint: there are many!)

Attempt #1: What Problems Do You See?

```
procedure removeAllLeaves()
    removeAllLeaves(root)
end procedure

procedure removeAllLeaves(node):
    if node is leaf then
        node <- null
    end if
    removeAllLeaves(node.left)
    removeAllLeaves(node.right)
end procedure
```

Problem: No Terminating Condition!

removeAllLeaves() will be called on the current node’s left and right child even if the left and right child are null. This will throw NullPointerException.

```
procedure removeAllLeaves()
    removeAllLeaves(root)
end procedure

procedure removeAllLeaves(node):
    if node is leaf then
        node <- null
    end if
    removeAllLeaves(node.left)
    removeAllLeaves(node.right)
end procedure
```

Attempt #2

Add if-statements to prevent recursively calling on the current node’s children if they are null.

Attempt #2

Add if-statements to prevent recursively calling on the current node’s children if they are null.

```
procedure removeAllLeaves()
        removeAllLeaves(root)
end procedure

procedure removeAllLeaves(node):
        if node is leaf then
                node <- null
        end if
        if node.left is not null then
                removeAllLeaves(node.left)
        end if
        if node.right is not null then
                removeAllLeaves(node.right)
        end if
end procedure
```

Another problem: No Changes Are Made!

Remember that references are passed in by value! That means that any modification made to the reference itself will not affect the caller’s reference.

```
procedure removeAllLeaves()
    removeAllLeaves(root)
end procedure

procedure removeAllLeaves(node):
    if node is leaf then
        node <- null
    end if
    if node.left is not null then
        removeAllLeaves(node.left)
    end if
    if node.right is not null then
        removeAllLeaves(node.right)
    end if
end procedure
```

Attempt #3

- Recall that references are passed in by value, but objects are passed in by references. Thus, we want to make changes to the object instead of the reference.

Attempt #3

- Recall that references are passed in by value, but objects are passed in by references. Thus, we want to make changes to the object instead of the reference.
- The BSTNode has three attributes we can make changes to: the data, the left child, and the right child.

Attempt #3

- Recall that references are passed in by value, but objects are passed in by references. Thus, we want to make changes to the object instead of the reference.
- The BSTNode has three attributes we can make changes to: the data, the left child, and the right child.
- In regards to removing children, the only attributes that are relevant are the left and right child. We can set these to null if we want to remove the current node’s left or right subtree.

Attempt #3

- Recall that references are passed in by value, but objects are passed in by references. Thus, we want to make changes to the object instead of the reference.
- The BSTNode has three attributes we can make changes to: the data, the left child, and the right child.
- In regards to removing children, the only attributes that are relevant are the left and right child. We can set these to null if we want to remove the current node’s left or right subtree.
- This is the underlying principle of the look-ahead method: work with the children instead of the current node.

Attempt #3

Instead of trying to modify the current node, modify the children.

Attempt #3

Instead of trying to modify the current node, modify the children.

```
procedure removeAllLeaves()
    removeAllLeaves(root)
end procedure

procedure removeAllLeaves(node):
    if node.left is not null then
        if node.left is a leaf then
            node.left <- null
        else
            removeAllLeaves(node.left)
        end if-else
    end if
    if node.right is not null then
        if node.right is a leaf then
            node.right <- null
        else
            removeAllLeaves(node.right)
        end if-else
    end if
end procedure
```

Another Problem: Root Isn’t Handled!

We make changes to the current node’s left and right children. Thus, when root is passed into the function, we make changes to its left and right children, but never to root itself. This code would not work if root were a leaf.

```
procedure removeAllLeaves()
    removeAllLeaves(root)
end procedure

procedure removeAllLeaves(node):
    if node.left is not null then
        if node.left is a leaf then
            node.left <- null
        else
            removeAllLeaves(node.left)
        end if-else
    end if
    if node.right is not null then
        if node.right is a leaf then
            node.right <- null
        else
            removeAllLeaves(node.right)
        end if-else
    end if
end procedure
```

Attempt #4

Add code to handle the root. First check if root is not null. If there is a root, then check if it is a leaf to determine whether to set it to null.

Attempt #4

Add code to handle the root. First check if root is not null. If there is a root, then check if it is a leaf to determine whether to set it to null.

```
procedure removeAllLeaves()
    if root is not null then
        if root is a leaf then
            root <- null
        else
            removeAllLeaves(root)
        end if-else
    end if
end procedure

procedure removeAllLeaves(node):
    if node.left is not null then
        …
    end if
    if node.right is not null then
        …
    end if
end procedure
```

Final Problem: Current May Be Null

Another problem is that the current node may be null in the recursive method, which will cause NullPointerException to be thrown. An example of when this may occur is a BST with a root that has a left child.

```
procedure removeAllLeaves()
    if root is not null then
        if root is a leaf then
            root <- null
        else
            removeAllLeaves(root)
        end if-else
    end if
end procedure

procedure removeAllLeaves(node):
    if node.left is not null then
        …
    end if
    if node.right is not null then
        …
    end if
end procedure
```

(Correct) Attempt #5

Ensure that the current node is not null before working with its children.

This solution works.

(Correct) Attempt #5

Ensure that the current node is not null before working with its children.

This solution works.

```
procedure removeAllLeaves()
        …
end procedure

procedure removeAllLeaves(node):
        if node is not null then
                if node.left is not null then
                        …
                end if
                if node.right is not null then
                        …
                end if
        end if
end procedure
```

Full Solution (Public Method)

```
procedure removeAllLeaves()
    if root is not null then
        if root is a leaf then
            root <- null
        else
            removeAllLeaves(root)
        end if-else
    end if
end procedure
```

Full Solution (Private Method)

```
procedure removeAllLeaves(node):
    if node is not null then
        if node.left is not null then
            if node.left is a leaf then
                node.left <- null
            else
                removeAllLeaves(node.left)
            end if-else
        end if
        if node.right is not null then
            if node.right is a leaf then
                node.right <- null
            else
                removeAllLeaves(node.right)
            end if-else
        end if
    end if
end procedure
```

This was the Look-Ahead Method

- First, we handle the root. We check if it is null, a leaf, or neither. We then perform the appropriate action based on this result.

This was the Look-Ahead Method

- First, we handle the root. We check if it is null, a leaf, or neither. We then perform the appropriate action based on this result.
- Next, we handle the current node’s children. We check if a child is null, a leaf, or neither. We then perform the appropriate action based on this result.

This was the Look-Ahead Method

- First, we handle the root. We check if it is null, a leaf, or neither. We then perform the appropriate action based on this result.
- Next, we handle the current node’s children. We check if a child is null, a leaf, or neither. We then perform the appropriate action based on this result.
- The Look-Ahead method performs actions based on the next level of nodes instead of the current node.

Notice the Duplicate Code

We had to perform the algorithm three times: once for the root, once for the current node's left child, and once for the current node's right child.

We can fix this using **pointer reinforcement**.

```
procedure removeAllLeaves()
    if root is not null then
        if root is a leaf then
            root <- null
        else
            removeAllLeaves(root)
        end if-else
    end if
end procedure

procedure removeAllLeaves(node):
    if node is not null then
        if node.left is not null then
            if node.left is a leaf then
                node.left <- null
            else
                removeAllLeaves(node.left)
            end if-else
        end if
        if node.right is not null then
            if node.right is a leaf then
                node.right <- null
            else
                removeAllLeaves(node.right)
            end if-else
        end if
    end if
end procedure
```

Pointer Reinforcement Principles

- Principle #1: Work with the current node instead of the next level of nodes.

Pointer Reinforcement Principles

- Principle #1: Work with the current node instead of the next level of nodes.
- Principle #2: The recursive method must replace the current node with the modified node. This is done by returning the node that we want to replace the original node with.

Pointer Reinforcement Principles

- Principle #1: Work with the current node instead of the next level of nodes.
- Principle #2: The recursive method must replace the current node with the modified node. This is done by returning the node that we want to replace the original node with.
- Principle #3: Whenever the recursive method is called, the caller must set the updated node back to the reference of the original node.
  - myObj = change(myObj)   // where “myObj” represents any object, and “change” represents any method that modifies “myObj”

Implementing Pointer Reinforcement

We will now implement Pointer Reinforcement for removeAllLeaves().

We begin by writing the private recursive method. Afterwards, we will write the public method.

Recursive method: Cases to Handle

- Case #1: If the current node is null
- Case #2: If the current node is a leaf
- Case #3: If the current node is neither null nor a leaf.

Case #1: If the current node is null

```
**procedure** removeAllLeaves(node):
        **if** current node is null **then**
                // Do something
        **end if**
**end procedure**
```

What do we replace “Do something” with?

Case #1: If the current node is null

What do we replace “Do something” with?

Case #1: If the current node is null

What do we replace “Do something” with?

If the current node is null, we still want it to be null.

Case #1: If the current node is null

What do we replace “Do something” with?

If the current node is null, we still want it to be null.

<u>Recall Pointer Reinforcement Principle #2:</u> The recursive method must replace the current node with the modified node. This is done by returning the node that we want to replace the original node with.

Case #1: If the current node is null

What do we replace “Do something” with?

If the current node is null, we still want it to be null.

<u>Recall Pointer Reinforcement Principle #2:</u> The recursive method must replace the current node with the modified node. This is done by returning the node that we want to replace the original node with.

Since we still want the current node to be null, we return null.

Case #1: If the current node is null

```
procedure removeAllLeaves(node):
    if current node is null then
        return null
    end if
end procedure
```

Case #2: The current node is a leaf

```
procedure removeAllLeaves(node):
    if current node is null then
        return null
    end if
    if current node is a leaf then
        // Do something
    end if
end procedure
```

What should we replace “Do something” with?

Case #2: If the current node is a leaf

What do we replace “Do something” with?

Case #2: If the current node is a leaf

What do we replace “Do something” with?

If the current node is a leaf, we want to remove it. Removing the node means that we want to replace the leaf node will null.

Case #2: If the current node is a leaf

What do we replace “Do something” with?

If the current node is a leaf, we want to remove it. Removing the node means that we want to replace the leaf node will null.

<u>Recall Pointer Reinforcement Principle #2: </u>The recursive method must replace the current node with the modified node. This is done by returning the node that we want to replace the original node with.

Case #2: If the current node is a leaf

What do we replace “Do something” with?

If the current node is a leaf, we want to remove it. Removing the node means that we want to replace the leaf node will null.

<u>Recall Pointer Reinforcement Principle #2: </u>The recursive method must replace the current node with the modified node. This is done by returning the node that we want to replace the original node with.

Since we want to replace the leaf node with null, we want to return null.

Case #2: The current node is a leaf

```
procedure removeAllLeaves(node):
    if current node is null then
        return null
    end if
    if current node is a leaf then
        return null
    end if
end procedure
```

Case #3: The current node is a neither null nor a leaf

```
procedure removeAllLeaves(node):
    if current node is null then
        return null
    end if
    if current node is a leaf then
        return null
    end if
    //Do something
end procedure
```

What do we replace “Do something” with?

Case #3: The current node is a neither null nor a leaf

- We want to recursively call removeAllLeaves() on the node’s left and right child.

Case #3: The current node is a neither null nor a leaf

- We want to recursively call removeAllLeaves() on the node’s left and right child.
- Recall that we must use Principle #3 whenever performing a recursive call: Whenever the recursive method is called, the caller must set the updated node back to the reference of the original node.

Case #3: The current node is a neither null nor a leaf

- We want to recursively call removeAllLeaves() on the node’s left and right child.
- Recall that we must use Principle #3 whenever performing a recursive call: Whenever the recursive method is called, the caller must set the updated node back to the reference of the original node.
- Thus, we set the return value of removeAllLeaves back to the current node’s left and right child references.

Case #3: The current node is a neither null nor a leaf

```
procedure removeAllLeaves(node):
    if current node is null then
        return null
    end if
    if current node is a leaf then
        return null
    end if
    node.left <- removeAllLeaves(node.left)
    node.right <- removeAllLeaves(node.right)
end procedure
```

Case #3: The current node is a neither null nor a leaf

```
procedure removeAllLeaves(node):
    if current node is null then
        return null
    end if
    if current node is a leaf then
        return null
    end if
    node.left <- removeAllLeaves(node.left)
    node.right <- removeAllLeaves(node.right)
end procedure
```

Notice how we are assigning the return value of removeAllLeaves (the updated node) back to the original node for both the left and right child. This follows Principle #3.

Case #3: The current node is a neither null nor a leaf

- But don’t forget principle #2!

Case #3: The current node is a neither null nor a leaf

- But don’t forget principle #2!
- The recursive method must return the updated node

Case #3: The current node is a neither null nor a leaf

- But don’t forget principle #2!
- The recursive method must return the updated node
- If the current node is neither null nor a leaf, we don’t want to make any changes to the current node. (The only changes we make are to the current node’s left and right subtree.) If we don’t want to make any changes, that means we want to replace the node with itself. Thus, we return the original node.

Case #3: The current node is a neither null nor a leaf

```
procedure removeAllLeaves(node):
    if current node is null then
        return null
    end if
    if current node is a leaf then
        return null
    end if
    node.left <- removeAllLeaves(node.left)
    node.right <- removeAllLeaves(node.right)
    return node;
end procedure
```

Refactored Recursive Method

We can combine the first two if-statements to make the code shorter.

```
procedure removeAllLeaves(node):
        if current node is null OR current node is a leaf then
                return null
        end if
        node.left <- removeAllLeaves(node.left)
        node.right <- removeAllLeaves(node.right)
        return node;
end procedure
```

Implementing the Public Method

- We will now implement the public method. This is actually super easy!

Implementing the Public Method

- We will now implement the public method. This is actually super easy!
- We want to recursively call the method on the BST’s root. Remember that we must use Principle #3 whenever calling the recursive method: set the updated node back to the reference of the original node.

Implementing the Public Method

- We will now implement the public method. This is actually super easy!
- We want to recursively call the method on the BST's root. Remember that we must use Principle #3 whenever calling the recursive method: set the updated node back to the reference of the original node.
- Thus, we will set the updated root back to the result of calling the root.

Implementing the Public Method

```
procedure removeAllLeaves():
    root <- removeAllLeaves(root);
end procedure

procedure removeAllLeaves(node):
    if current node is null OR current node is a leaf then
        return null
    end if
    node.left <- removeAllLeaves(node.left)
    node.right <- removeAllLeaves(node.right)
    return node;
end procedure
```

And that's it! This code now works using Pointer-Reinforcement

Look Ahead

```
procedure removeAllLeaves()
    if root is not null then
        if root is a leaf then
            root <- null
        else
            removeAllLeaves(root)
        end if-else
    end if
end procedure

procedure removeAllLeaves(node):
    if node is not null then
        if node.left is not null then
            if node.left is a leaf then
                node.left <- null
            else
                removeAllLeaves(node.left)
            end if-else
        end if
        if node.right is not null then
            if node.right is a leaf then
                node.right <- null
            else
                removeAllLeaves(node.right)
            end if-else
        end if
    end if
end procedure
```

Look Ahead

```
procedure removeAllLeaves()
    if root is not null then
        if root is a leaf then
            root <- null
        else
            removeAllLeaves(root)
        end if-else
    end if
end procedure

procedure removeAllLeaves(node):
    if node is not null then
        if node.left is not null then
            if node.left is a leaf then
                node.left <- null
            else
                removeAllLeaves(node.left)
            end if-else
        end if
        if node.right is not null then
            if node.right is a leaf then
                node.right <- null
            else
                removeAllLeaves(node.right)
            end if-else
        end if
    end if
end procedure
```

Pointer Reinforcement

```
procedure removeAllLeaves():
    root <- removeAllLeaves(root);
end procedure

procedure removeAllLeaves(node):
    if current node is null OR current node is a leaf then
        return null
    end if
    node.left <- removeAllLeaves(node.left)
    node.right <- removeAllLeaves(node.right)
    return node;
end procedure
```

Recap How We Used Pointer Reinforcement Principles

- Principle #1: We worked with the current node in the private helper method instead of the node’s children.

Recap How We Used Pointer Reinforcement Principles

- Principle #1: We worked with the current node in the private helper method instead of the node’s children.
- Principle #2: Return the updated node

Recap How We Used Pointer Reinforcement Principles

- Principle #1: We worked with the current node in the private helper method instead of the node’s children.
- Principle #2: Return the updated node
  - If the current node is null or a leaf, return null
  - If the current node is not null nor a leaf, then we return the updated node after calling the function on its left and right children.

Recap How We Used Pointer Reinforcement Principles

- Principle #1: We worked with the current node in the private helper method instead of the node's children.
- Principle #2: Return the updated node
  - If the current node is null or a leaf, return null
  - If the current node is not null nor a leaf, then we return the updated node after calling the function on its left and right children.
- Principle #3: myObj = change(myObj)

Recap How We Used Pointer Reinforcement Principles

- Principle #1: We worked with the current node in the private helper method instead of the node’s children.
- Principle #2: Return the updated node
    - If the current node is null or a leaf, return null
    - If the current node is not null nor a leaf, then we return the updated node after calling the function on its left and right children.
- Principle #3: myObj = change(myObj)
    - root = removeAllLeaves(root);
    - node.left = removeAllLeaves(node.left);
    - node.right = removeAllLeaves(node.right);

Look-Ahead	Pointer-Reinforcement
Works with the with the next level of objects. For BSTs, this means that it works with the current node’s left and right child.	Works with the current object (principle #1).

Look-Ahead	Pointer-Reinforcement
Works with the with the next level of objects. For BSTs, this means that it works with the current node’s left and right child.	Works with the current object (principle #1).
Private helper does not return anything.	Private helper method returns the updated object (principle #2). For BSTs, the method returns what you want to replace the current node with.

Look-Ahead	Pointer-Reinforcement
Works with the with the next level of objects. For BSTs, this means that it works with the current node’s left and right child.	Works with the current object (principle #1).
Private helper does not return anything.	Private helper method returns the updated object (principle #2). For BSTs, the method returns what you want to replace the current node with.
For BSTs, the root has to be handled specially, resulting in inelegant, duplicate code.	The root does not have to be handled specially. It just needs to be handled using the recursive method.

Extra Practice Problems

- **Pointer Reinforcement** is a difficult, yet vital concept for BST and AVL coding.
- The following slides contain Pointer Reinforcement practice problems.
- Be sure to use Pointer Reinforcement for all your solutions.

Extra Practice Problems -- #1

Write a method called reduceSingleChildren() that eliminates any nodes with only one child. The nodes that are eliminated should be replaced its only child.

Extra Practice Problems -- #2

Write a method called endOddLevels() that modifies the tree so that all branches end on an odd level. (The root is on level 1; it’s children are on level 2; the children’s children are on level 3; etc.) If there is a leaf is on an even level, then it should be removed.

Extra Practice Problems -- #3

Write a method called truncateTree() that accepts an integer “n” and removes any nodes whose level is greater than “n”.