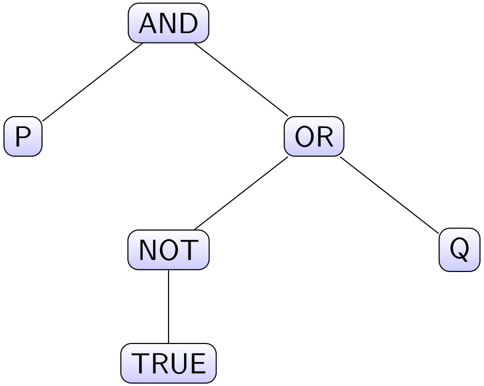
Data Structures and Algorithms
Assignment 3

Alan P. Sexton

University of Birmingham

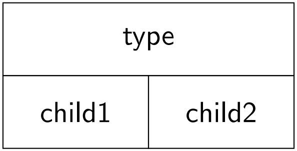
Spring 2020

A PLTreeNode tree



Prefix:	AND(P,OR(NOT(TRUE),Q))
Infix:	$((P) \wedge ((\neg\top) \vee (Q)))$
Reverse Polish:	`[P, TRUE, NOT, Q, OR, AND]`

A single PLTreeNode Object



- Make sure that your code **ALWAYS** maintains the invariant:

  ```
  type.getArity()== 0 → child1 == null && child2 == null
  type.getArity()== 1 → child1 != null && child2 == null
  type.getArity()== 2 → child1 != null && child2 != null
  ```

  EITHER test if child is null OR get the arity
- Change the type simply by assigning to it (but make sure you correct the children if necessary to preserve the invariant)
- Within `PLTreeNode`, don't use setters and getters to access `type`, `child1` and `child2`: just assign to or from them

Recursion: Preorder Recursion

```
myMethod()
{
    // do something to this node here

    if (child1 != null)
        child1.myMethod();

    if (child2 != null)
        child2.myMethod();
}
```

Recursion: Inorder Processing

```
myMethod()
{
    if (child1 != null)
        child1.myMethod();

    // do something to this node here

    if (child2 != null)
        child2.myMethod();
}
```

Recursion: Postorder Processing

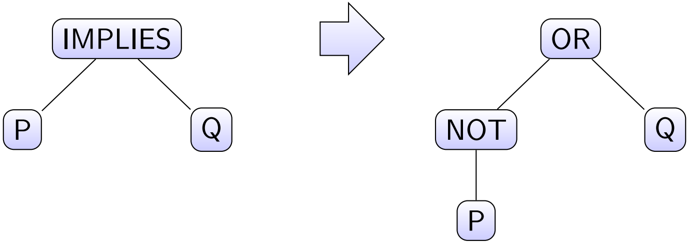
```
myMethod()
{
    if (child1 != null)
        child1.myMethod();

    if (child2 != null)
        child2.myMethod();

    // do something to this node here

}
```

Processing Order Choice

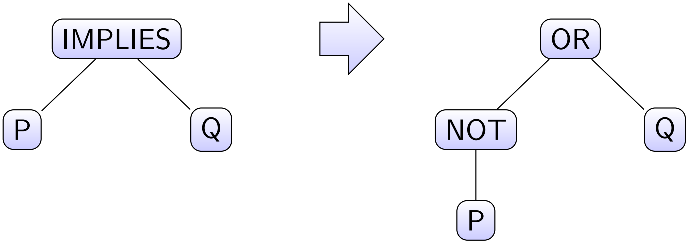
The choice of processing order is sometimes a free choice, sometimes dictated by the situation.

- If the recursion does not modify the tree, then the recursion is just gathering information
  - Choose the order so that the information is available when it is needed
  - Sometimes some recursion can be avoided if you are careful
    - e.g. if you find the necessary information in the left sub-tree, it may not be necessary to recurse down the right
- If the recursion does modify the tree, then you may need a specific order
  - If you are recursing to find and modify a pattern in the tree, then making that modification may introduce that pattern in other places: make sure that the process order will catch those newly introduced patterns

replaceImplies()



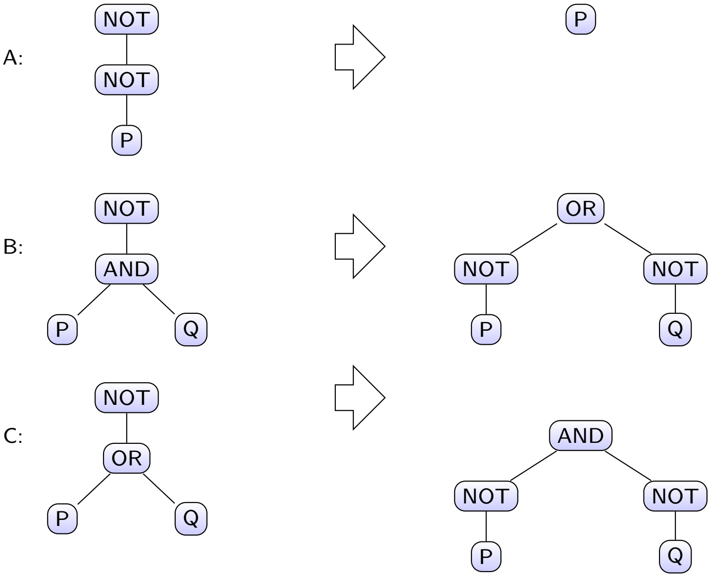
replaceImplies()



- Doesn't introduce new Left Hand Side (LHS) patterns so no problems with processing order choice

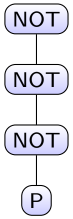
pushNotDown()

A:

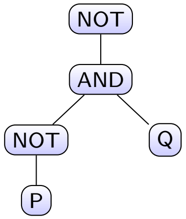


`pushNotDown()`: can produce new LHS patterns

A:

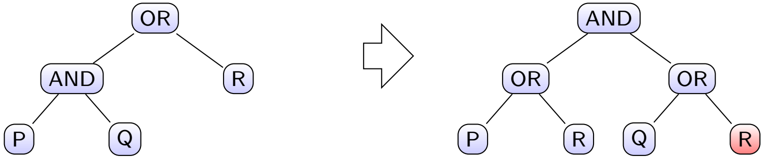


B:

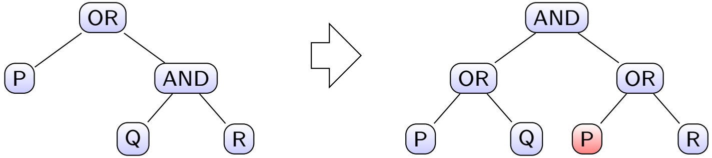


pushOrBelowAnd()

A:



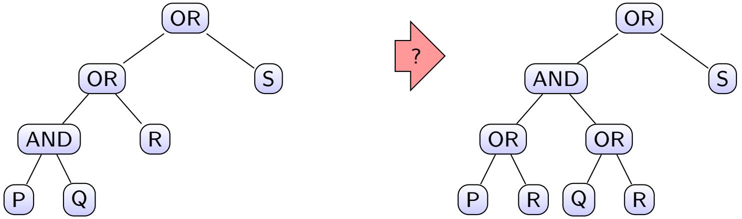
B:



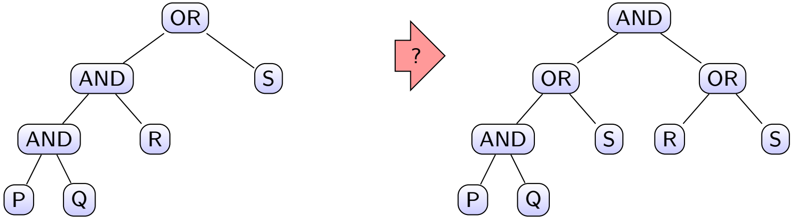
(See later about **Deep Copies**)

pushOrBelowAnd(): Issues

A:

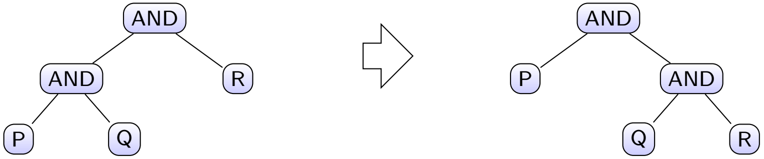


B:

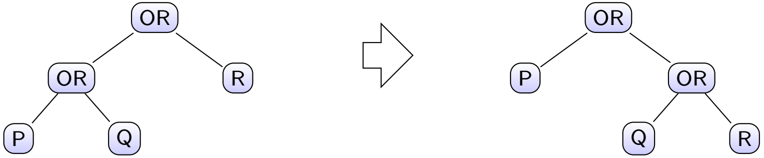


makeAndOrRightDeep()

A:



B:

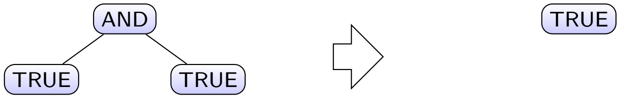


`evaluateConstantSubtrees()`

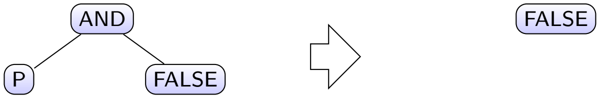
A	B	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$\neg B$
$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$
$\top$	N	B	$\top$	B	N
$\bot$	$\top$	$\bot$	$\top$	$\top$	
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	
$\bot$	N	$\bot$	B	$\top$	
N	$\top$	A	$\top$	$\top$	
N	$\bot$	$\bot$	A	$\neg A$	
N	N	N	N	N	

evaluateConstantSubtrees(): some cases

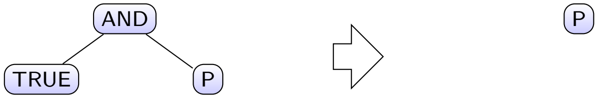
A:



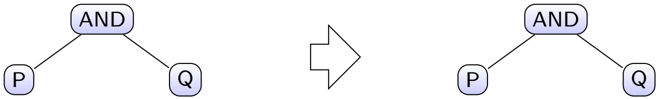
B:



C:

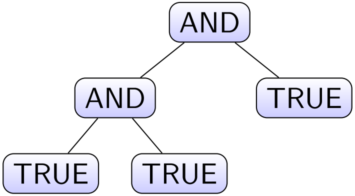


D:



`evaluateConstantSubtrees()`: Preorder or postorder?

Try (manually!) seeing what the difference is if you recurse before or after processing the node in this example:



Assignment vs Deep Copy

There is an extremely important difference between simply assigning a `PLTreeNode` object to a `child1` or `child2` field in a `PLTreeNode` object, or assigning a deep copy of such a `PLTreeNode` to the child field in question. That is, the difference between:

- `child1 = child2;`

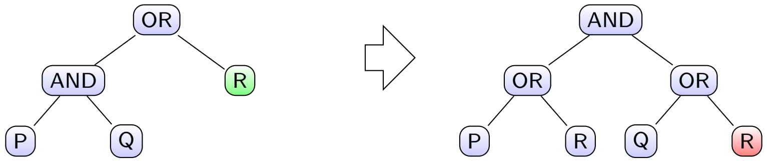
and

- `child1 = new PLTreeNode(child2);`

The short answer is that, if you are leaving the old copy in the tree, you should use the deep copy constructor. If you are **NOT** leaving the old copy in the tree, just assign it.

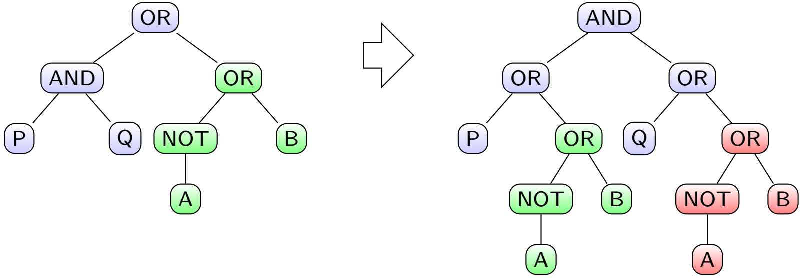
pushOrBelowAnd()

One of the cases that has to be dealt with in `pushOrBelowAnd()` is as follows:



`pushOrBelowAnd()`: larger case

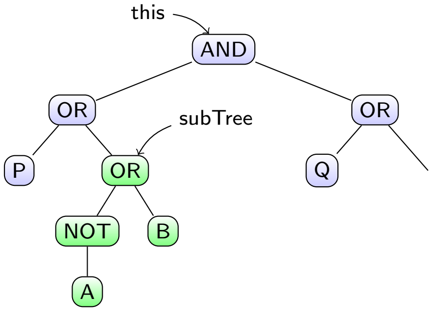
Consider a larger sub-tree in place of R:



`pushOrBelowAnd()`: partially completed

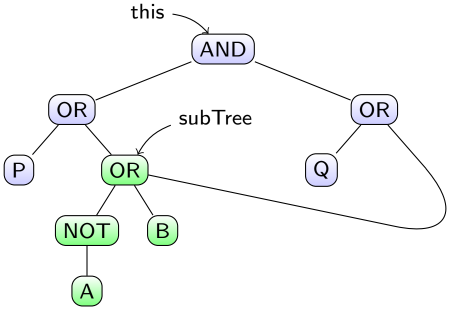
Now consider the situation when you are partially through the transformation:

- "`this`" is pointing to the current `PLTreeNode`
- You have already restructured the tree correctly, but have still to put the copy of the green sub-tree into position
- The variable `subTree`, which is the same as `child1.child2`, is pointing to the green sub-tree.



Simple assignment

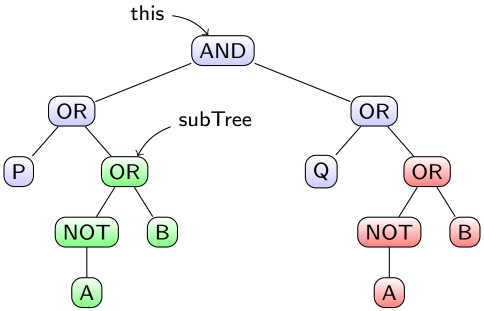
If you now execute: `child2.child2 = subTree;`



- `child1.child2` and `child2.child2` point to the **same** `PLTreeNode`
- This means that things seem okay, e.g. `toString...` methods still work, but anything that changes the subtree `child1.child2`, **ALSO** changes the `child2.child2` subtree

Deep Copy

If you execute: `child2.child2 = new PLTreeNode(subTree);`

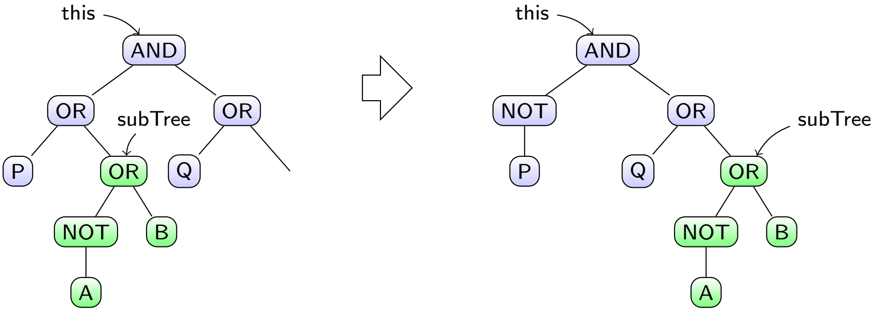


- The red sub-tree is a **DEEP COPY** of the green sub-tree: the nodes have the same `type` values, and the same tree structure, but they are different objects in memory
- `child1.child2` and `child2.child2` point to **DIFFERENT** sub-trees that have the same values
- No more problems with modifying one sub-tree changing another

Moving a subtree

What if you want to **MOVE** a subtree: say turn the left OR node into a NOT, and move the current child1.child2 subtree to the right child of the right OR node:

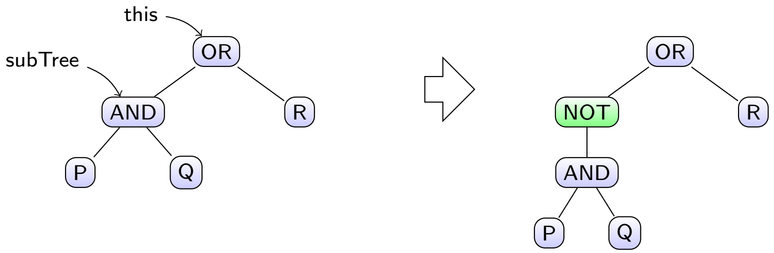
```
    child2.child2 = subTree ;
    child1.child2 = null;
    child1.type = NodeType.NOT;
```



Since we are **MOVING** the subtree, we can safely assign subTree to child2.child2 because we will not end up with two different pointers to the same sub-trees in different parts of the whole tree

Inserting a new node above a sub-tree

What if you want to insert a new node **ABOVE** a subtree?



Here we do need to create a new `PLTreeNode` for the `NOT`, but we are just moving the old `subTree`, so we do **not** need to make a deep copy:

```
this.child1 = new PLTreeNode(EntryNode.NOT,
    subTree, null);
```