Data Structures and Algorithms

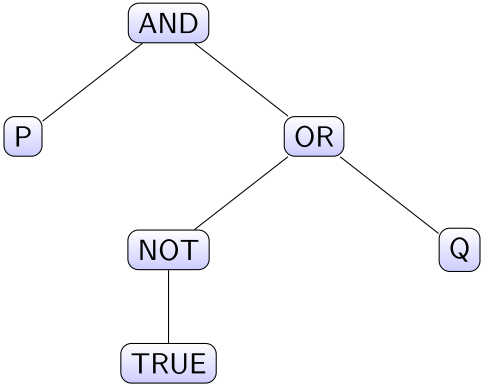
Assignment 3

Alan P. Sexton

University of Birmingham

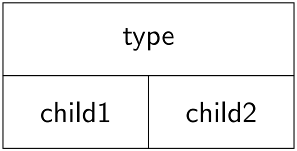
Spring 2019

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` $\rightarrow$ `child1 == null && child2 == null`
  `type.getArity()== 1` $\rightarrow$ `child1 != null && child2 == null`
  `type.getArity()== 2` $\rightarrow$ `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

}
```

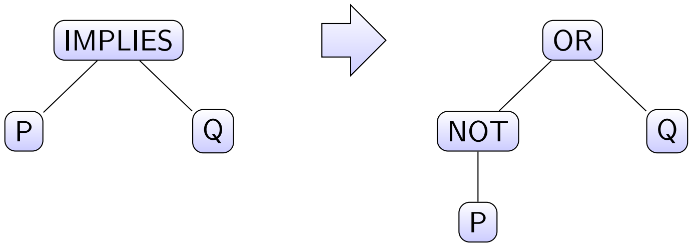
Processing Order Choice

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

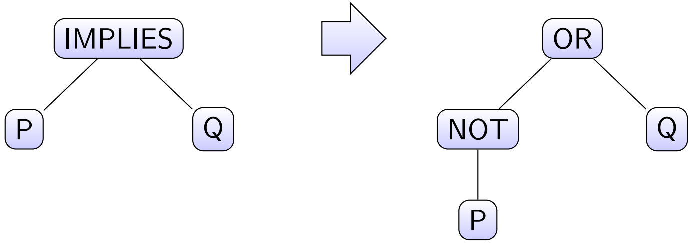
`eliminateImplies()`

Should have called it `replaceImplies()`



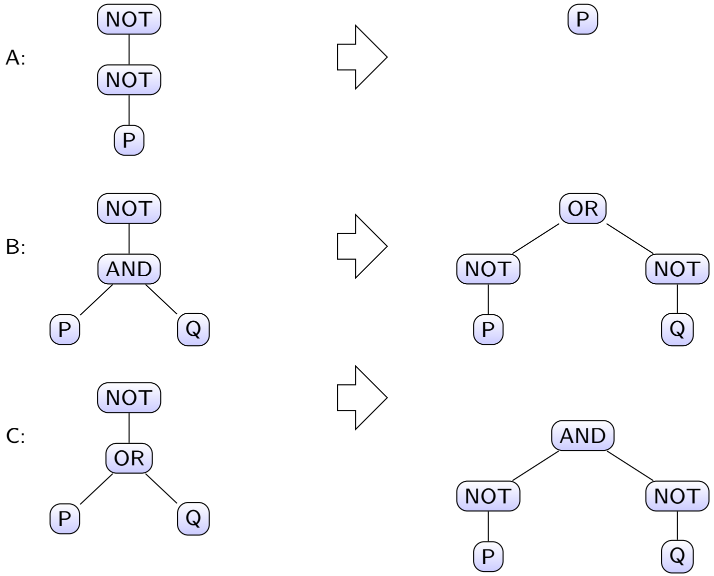
`eliminateImplies()`

Should have called it `replaceImplies()`



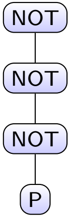
- Doesn't introduce new Left Hand Side (LHS) patterns so no problems with processing order choice

pushNotDown()

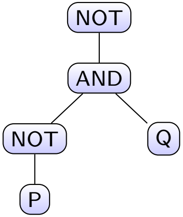


`pushNotDown()`: can produce new LHS patterns

A:

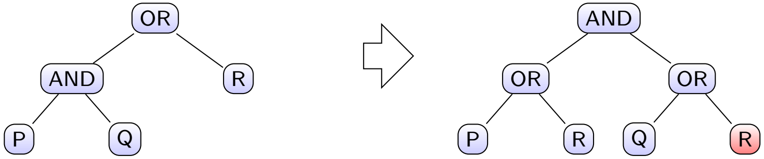


B:

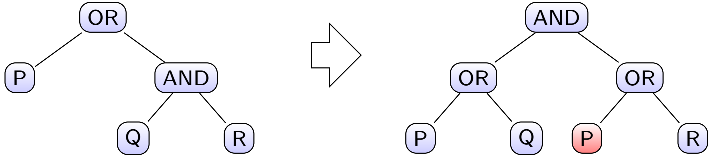


pushOrBelowAnd()

A:

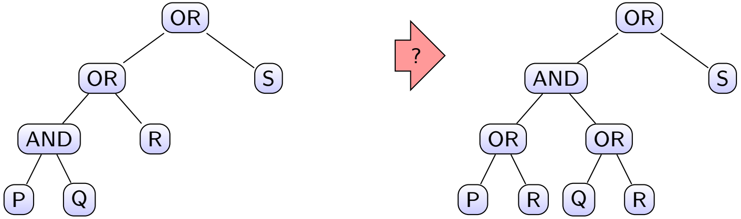


B:

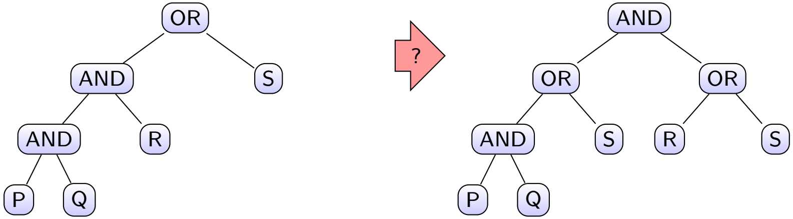


pushOrBelowAnd(): Issues

A:

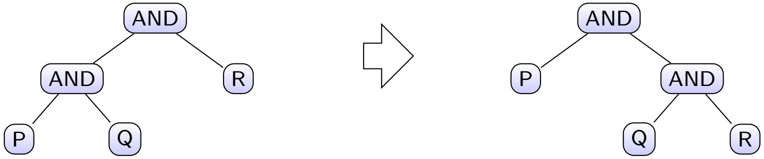


B:

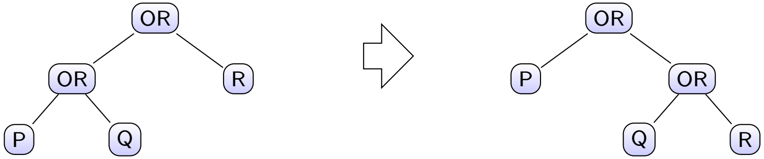


makeAndOrRightDeep()

A:



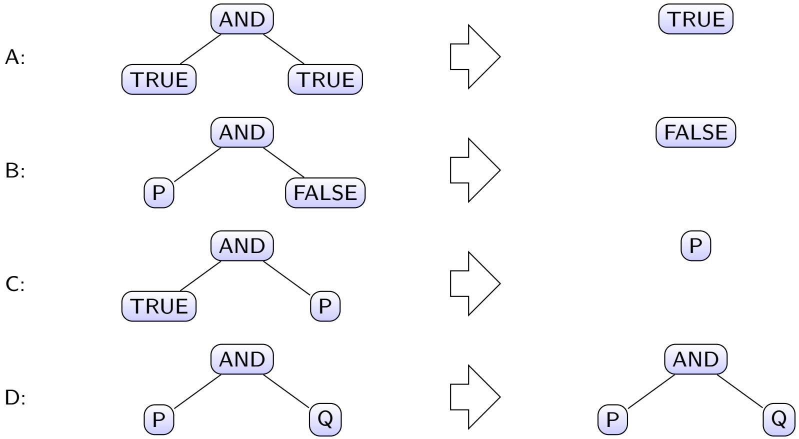
B:



evaluateConstantSubtrees()

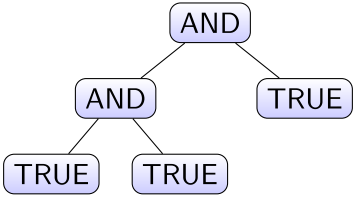
A	B	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$\neg B$
$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$
$\top$	N	B	$\top$	B	N
$\bot$	$\top$	$\bot$	$\top$	$\top$	
$\bot$	$\bot$	$\bot$	$\bot$	$\top$	
$\bot$	N	$\bot$	B	$\top$	
N	$\top$	A	$\top$	$\top$	
N	$\bot$	$\bot$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

evaluateConstantSubtrees(): some cases



`evaluateConstantSubtrees()`: Preorder or postorder?

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


AND
AND
TRUE
TRUE
TRUE