The University of Birmingham
School of Computer Science

Assignment 03

1 Introduction

This assignment will be marked out of 10 and contributes 10% of your final module mark.
Deadline for submission is:

14:00 Friday 28th February

While the first assignment was about linked lists and their implementation, and the second about using standard Java Collection classes for Lists, Maps, Sets and Arrays, this assignment is about implementing and manipulating binary trees.

As a motivating application we take the topic of *Propositional Logic*. We tackle the issues of representing propositional logic formulæ, evaluating them, simplifying them, outputting them and reducing them to *Conjunctive Normal Form*.

A set of Java sources have been provided for you that already supports creating a binary tree to represent a propositional logic expression, and constructing such a tree from a list of *node types* (propositional logic operators such as *AND*, *OR*, *NOT*, *IMPLIES*, *TRUE* and *FALSE*, and variables *A* to *Z*) in reverse polish notation (c.f. `https://en.wikipedia.org/wiki/Reverse_Polish_notation`). There is also a method to extract the reverse polish notation list of operators from the expression.

A sample propositional logic formula of the type that this program deals with is:

$$P \wedge (Q \vee \neg R) \rightarrow Q \wedge (\bot \vee (P \rightarrow R)) \wedge \top$$

2 Marking

- To get full marks, you will have to provide working implementations for all methods where the comment: "`//WRITE YOUR CODE HERE`" appears so that it matches the specification provided in the corresponding Javadoc comments.
- There will be 1 mark if your submission compiles correctly and can be run and passes the test provided in the initial assignment files.
- Unlike in previous exercises, a full set of tests have **NOT** been provided for you. In this assignment you should choose and write your own tests by first reading carefully this handout and the Javadoc comments and composing tests, in the style used in the previous assignments, to test that your implementations function as required. I strongly recommend that you use the Eclipse "*coverage*" feature, to ensure that there is at least some level of completeness of testing of your code.
- If your submission is not structured correctly or does not compile, or crashes or enters an infinite loop before any tests pass, then no marks will be earned.

3 Plagiarism

Plagiarism will not be tolerated: it is unfair to the other students and prevents you from learning, and would give you a mark you don't deserve, and qualifications you don't deserve, hence being unfair to Society as a whole too. Please behave well and reward the module lecturer and TA's by submitting your own, hard work.

All submissions will be checked for copying against other submissions and against sources on the web and elsewhere. Any student found to have

When the module lecturer decides that there is evidence of plagiarism, the case will be forwarded to the Senior Tutor, who will look at the evidence and apply penalties and warnings, or, if necessary, forward this to the University.

- Guidance Notes on Plagiarism: `https://www.cs.bham.ac.uk/internal/taught-students/plagiarism`
- Copying is plagiarism: `https://www.cs.bham.ac.uk/internal/taught-students/copying`
- University regulations on plagiarism: `https://intranet.birmingham.ac.uk/as/studentservices/conduct/plagiarism/index.aspx`

4 Student Welfare

If you miss or are going to miss a deadline, or are getting behind the learning material, for reasons outside of your control, please contact Student Welfare. Occasionally students represent the University in sports, programming competitions, etc., and if this is going to affect your ability to comply with a deadline, Welfare should be informed. It is the Welfare Team, not the teaching team, who can award extensions and cancellations, and devise other measures to cope with adverse situations. It is important to contact welfare as soon as possible when such situations arise.

5 Background

Most of the code required for this assignment is recursive. The general pattern is that something has to be done in the current node, and something must be done in the children of the node (if any). An important decision is whether you deal with the node itself before you deal with children (*preorder processing*), after you deal with the children (*postorder processing*) or, in the case of two children, whether you deal with the node between dealing with the left child and dealing with the right child (*inorder processing*). Different cases call for different choices in this respect. The methods `reversePolishBuilder(...)` and `getReversePolish()` provide good examples of recursion.

First get the code up and running and read the code provided to fully understand how it works. The code already handles creating a logical expression from a list of operators (`NodeType` enum values). See `https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html` for explanations on `enum`s.

Note how classes other than `PLTreeNode` must use `PLTreeNode.reversePolishBuilder(...)` to create formulæ. All the other constructors are intentionally private.

I recommend that you implement the necessary methods in the following order:

a. `toStringPrefix()`: A simple preorder recursion. Make sure that the spaces and the parentheses are **EXACTLY** as in the log output that is included in the comment at the end of `PLTreeNodeTest.java` and reproduced at the end of this document: the tests used for marking will be based in part on doing String comparisons with the return value from this method.

b. `toStringInfix()`: A simple inorder recursion. Make sure that the spaces and the parentheses are **EXACTLY** as in the log output that is included in the comment at the end of `PLTreeNodeTest.java` and reproduced at the end of this document: the tests used for marking will be based in part on doing String comparisons with the return value from this method.

c. `applyVarBindings(...)`: A very simple recursion. Note that you should **NOT** create new nodes for this, or simplify expressions. This is simply a matter of replacing the `type` class field of the variable nodes with `NodeType.TRUE` or `NodeType.FALSE` depending on what the binding argument parameter says it should be. Don't change the node if that variable is not in the binding map.

d. `replaceImplies()`: This is the simplest of the methods that edit the tree in a non-trivial way: Recurse down the tree. As you find nodes whose type is `IMPLIES`, i.e. `IMPLIES(x,y)`, where `x` and `y` are sub-trees, replace it with `OR(NOT(x),y)`. Note that you can just assign `NodeType.OR` to the type variable of the current node, the `child2` field (which contains the `y` sub-tree) stays the same, but you need to create a new `PLTreeNode` object for the new `NOT` sub-tree to assign to the `child1` node. Don't forget to put the `x` sub-tree as the first child of the new `NOT` node.

e. `pushNotDown()`: At this point you should be comfortable with editing the tree in the correct way. Make sure your approach handles long nested sub-trees of `NOT` nodes. hint: consider carefully whether the recursion should be preorder or postorder.

f. `pushOrBelowAnd()`: Be careful about putting a copy of a sub-tree into the tree. Where you need to have two copies of the same sub-tree in your tree, make a **deep** copy. There is a deep copy constructor provided for this purpose.

   There is a surprisingly tricky element to this method. Since it is particularly tricky, the following explicit guidance is provided:

   Note that when you find a pattern of an `OR` node with a `AND` node below it, you push the `OR` node down. However, that means that you also pull the `AND` node up! The trickiness involved is that you can introduce **NEW** instances of the pattern of an `AND` node as a direct child of an `OR` node in two different ways:

   i. Push an `OR` node down so that it ends up directly above an `AND` node that previously did not have an `OR` node above it.

   ii. Pull an `AND` node up so that it ends up directly below an `OR` node that previously did not have an `AND` node below it.

Consider an `OR` node with another `OR` node below it, and an `AND` node below that: `OR(OR(AND(W,X),Y),Z))`

Here if we recurse, the first `OR` does not match the pattern we are looking for, so we recurse further to the second `OR`. This matches the pattern of an `AND` below an `OR`, so we change it around to an `OR` below an `AND` as required: `OR(AND(OR(W,Y),OR(X,Y)),Z))`. That fixes the pattern we found, but, as a result, we have pulled the `AND` up one level and it is now sitting directly below the first `OR`, introducing a new instance of the pattern that we wanted to get rid of!

We can fix that by using pre-order recursion: First recurse down the sub-trees, then deal with the pattern on the current node. That guarantees to pull up any `AND` nodes right up to the child level before we deal with the pattern.

But wait: Consider the pattern `OR(AND(AND(W,X),Y),Z`. The pre-order recursion on the children won't make any changes because the there are no instances of the pattern below the top level. We then apply the correct change to get: `AND(OR(AND(W,X),Z),OR(Y,Z))`. However, having moved the `OR` down, we have introduced a **NEW** instance of the pattern and we already finished our recursion, so we don't ever get to deal with it!

We would fix this problem by switching to postorder recursion. But if we did that we would end up with the original problem above that we used preorder recursion to solve.

Here the solution is that we have to recurse **BEFORE** we deal with this node to handle `AND` pull ups, we then deal with the current node, and then we recurse again **AFTER** we deal with this node to handle the `OR` push downs.

g. `makeAndOrRightDeep()`: By this stage this should be very easy

h. `evaluateConstantSubtrees()`: This method is completely independent of the other methods: it is not required to get any of the other methods working and none of the other methods are required to get this working. The full version of this is a little complex. The idea is that if you have a subtree such as `AND(FALSE,x)` where `x` is some sub-tree, then you can replace it with `FALSE`. Similarly `AND(TRUE,x)` can be replaced with just `x`. To figure out what to do, you should consider the truth table for the different operators.

Normally the truth table says whether the results of the operator are true or false depending on whether the inputs are true or false. In our case, the inputs are the sub-trees, and we may not be able to resolve those sub-trees to true or false if they contain variables, because we can't evaluate the sub-tree fully until those variables are given a value.

Therefore we need to consider the truth tables where the inputs are true ($\top$), false ($\bot$) or *not known yet* (N):

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

To understand the table, consider the *AND* column: $A \wedge B$ where one or both of the inputs are unknown:

$\top \wedge N$, where the A sub-tree evaluates to true and the B sub-tree can't be evaluated yet, will have the same result as just the B sub-tree, so a node that represents `AND(TRUE,x)` where `x` is a sub-tree that can't be evaluated, can be replaced with the sub-tree `x`.

Similarly `AND(x,TRUE)` can be replaced with the sub-tree `x` and `AND(x,FALSE)` can be replaced with `FALSE`.

Finally `AND(x,y)`, where neither x nor y can be evaluated, can not be replaced at all and should remain unchanged.

6 Setup and Specification

You have been provided with a Java project folder, `dsa_assignment3`, that contains sources and tests for this assignment in the `dsa_2020` GIT repository (don't forget to do a "`git pull`" inside your copy of the repository to update it). You have been provided with the following files and packages:

```
dsa_assignment3/src/main/java/dsa_assignment3/
    NodeType.java
    PLTreeNode.java
    PLTreeNodeInterface.java
dsa_assignment3/src/test/java/dsa_assignment3/
    PLTreeNodeTest.java
```

If you set the top level `dsa_assignment3` as an Eclipse project, with `src/main/java` and `src/test/java` as the source folders on your build path, and add the user libraries for hamcrest-all (or hamcrest-core), JUnit and Log4J to your project, you should be able to compile and run the test (which should fail!) and which should produce some log output

Your task is to add missing code to the methods in `PLTreeNode.java` to make the methods function correctly according to the specifications in the Javadoc and this handout.

The `PLTreeNode.java` currently contains a test to check that you have entered your name and student id in the methods provided, and otherwise shows how the methods in PLTreeNode.java should be called with some sample suitable input. Please see the end of this handout for the output that the solution to this assignment generates.

- The precise information about the behaviour required of the missing code is detailed in the Javadoc comments in the files.
- You should modify the methods `PLTreeNode.getStudentId()` and `PLTreeNode.getStudentName()` in `PLTreeNode.java` to return your student id and name.
- You should **NOT** add any further class variables to `PLTreeNode.java`, or change the types or visibility of the existing class variables or methods nor modify any of the other files in the `main` part of the src tree.
- You can add extra **private** methods in PLTreeNode.java if you like but please note that they are not needed and my solution does not do so.
- You can (and should!) modify the `PLTreeNodeTest.java` file and you can add any other test files as you please. Your test files are for your own use and should NOT be submitted.
- You should not modify the package structure or add any new classes that `PLTreeNode.java` depends upon to run. Your final submission will be purely the `PLTreeNode.java` file, so any modifications outside that file will not be considered and the `PLTreeNode.java` file that you submit must work with the other files as they currently stand in the assignment structure.
- You should not use any print statements to any files or to standard out or standard error streams: if you want to have some debug output, use the logging calls for Log4j. The logging system will be switched to disable log output when your submission is run for marking purposes.
- For marking purposes, your code will be compiled and executed against a test set in a secure sandbox environment. Any attempts to break the security of the sandbox will cause the execution of your program to fail. This includes infinite loops in your code.
- When you have completed your code changes to your satisfaction, submit your `PLTreeNode.java`

7 Sample Log Output

You are expected to write your **OWN** tests for this assignment. Two test methods are provided for you at the end of the `testPLTreeConstruction()` to give you an idea of the kinds of test that you need to write. These tests are based on the contents of the log output from my solution to this assignment (see below). However, while you **SHOULD** put in tests for each of the outputs from the log, that is **NOT** sufficient: you should add further tests to test for anything that might not be correct.

The log output from the solution to the exercise for the `testPLTreeConstruction()` method in `PLTreeNodeTest.java` has been copied as a comment into the file below the method to assist you in writing your tests. You may find it useful to copy and paste some of the strings from that comment into some of your tests. The contents of that comment are listed below. **Do not try to copy and paste from the PDF document**: PDF will not necessarily produce the right unicode characters or the right ordering of the characters: use the comment in `PLTreeNodeTest.java`.

```
typeList: [R, P, OR, TRUE, Q, NOT, AND, IMPLIES]
Constructed: implies(or(R,P),and(true,not(Q)))
Constructed: ((R∨P)→(⊤∧¬Q))
typeListReturned: [R, P, OR, TRUE, Q, NOT, AND, IMPLIES]
Applied bindings : {P=true, R=false} to get: implies(or(false,true),and(true,not(Q)))
Applied bindings : {P=true, R=false} to get: ((⊥∨⊤)→(⊤∧¬Q))
typeListReturned: [FALSE, TRUE, OR, TRUE, Q, NOT, AND, IMPLIES]
Replace Implies: or(not(or(false,true)),and(true,not(Q)))
Replace Implies: (¬(⊥∨⊤)∨(⊤∧¬Q))
typeListReturned: [FALSE, TRUE, OR, NOT, TRUE, Q, NOT, AND, OR]
pushNotDown: or(and(not(false),not(true)),and(true,not(Q)))
pushNotDown: ((¬⊥∧¬⊤)∨(⊤∧¬Q))
typeListReturned: [FALSE, NOT, TRUE, NOT, AND, TRUE, Q, NOT, AND, OR]
pushOrBelowAnd: and(and(or(not(false),true),or(not(false),not(Q))),and(or(not(true),true),or(not(true),not(Q))))
pushOrBelowAnd: (((¬⊥∨⊤)∧(¬⊥∨¬Q))∧((¬⊤∨⊤)∧(¬⊤∨¬Q)))
typeListReturned: [FALSE, NOT, TRUE, OR, FALSE, NOT, Q, NOT, OR, AND, TRUE, NOT, TRUE, OR, TRUE, NOT, Q, NOT, OR, AND, AND]
makeAndOrRightDeep: and(or(not(false),true),and(or(not(false),not(Q)),and(or(not(true),true),or(not(true),not(Q)))))
```

```
makeAndOrRightDeep: ((¬⊥∨⊤)∧((¬⊥∨¬Q)∧((¬⊤∨⊤)∧(¬⊤∨¬Q))))
typeListReturned: [FALSE, NOT, TRUE, OR, FALSE, NOT, Q, NOT, OR, TRUE, NOT, TRUE, OR, TRUE, NOT, Q, NOT, OR, AND, AND, AND]
Evaluate constant subtrees to get: not(Q)
Evaluate constant subtrees to get: ¬Q
typeListReturned: [Q, NOT]
typeList: [R, P, IMPLIES, S, IMPLIES, NOT, Q, IMPLIES]
Constructed: implies(not(implies(implies(R,P),S)),Q)
Constructed: (¬((R→P)→S)→Q)
typeListReturned: [R, P, IMPLIES, S, IMPLIES, NOT, Q, IMPLIES]
ReduceToCNF to get: and(or(R,or(S,Q)),or(not(P),or(S,Q)))
ReduceToCNF to get: ((R∨(S∨Q))∧(¬P∨(S∨Q)))
typeListReturned: [R, S, Q, OR, OR, P, NOT, S, Q, OR, OR, AND]
Evaluate constant subtrees to get: and(or(R,or(S,Q)),or(not(P),or(S,Q)))
Evaluate constant subtrees to get: ((R∨(S∨Q))∧(¬P∨(S∨Q)))
typeListReturned: [R, S, Q, OR, OR, P, NOT, S, Q, OR, OR, AND]
typeList: [A, B, AND, C, OR, D, OR, E, OR, F, OR, G, OR, H, OR]
Constructed: or(or(or(or(or(or(and(A,B),C),D),E),F),G),H)
Constructed: (((((((A∧B)∨C)∨D)∨E)∨F)∨G)∨H)
typeListReturned: [A, B, AND, C, OR, D, OR, E, OR, F, OR, G, OR, H, OR]
pushOrBelowAnd: and(or(or(or(or(or(A,C),D),E),F),G),H),or(or(or(or(or(B,C),D),E),F),G),H))
pushOrBelowAnd: (((((((A∨C)∨D)∨E)∨F)∨G)∨H)∧((((((B∨C)∨D)∨E)∨F)∨G)∨H))
typeListReturned: [A, C, OR, D, OR, E, OR, F, OR, G, OR, H, OR, B, C, OR, D, OR, E, OR, F, OR, G, OR, H, OR, AND]
Extra tests of pushOrBelowAnd()
typeList: [A, B, AND, C, AND, D, OR, E, OR]
Constructed: or(or(and(and(A,B),C),D),E)
Constructed: ((((A∧B)∧C)∨D)∨E)
typeListReturned: [A, B, AND, C, AND, D, OR, E, OR]
pushOrBelowAnd: and(and(or(or(A,D),E),or(or(B,D),E)),or(or(C,D),E))
pushOrBelowAnd: ((((A∨D)∨E)∧((B∨D)∨E))∧((C∨D)∨E))
typeListReturned: [A, D, OR, E, OR, B, D, OR, E, OR, AND, C, D, OR, E, OR, AND]
```