CS21120 2015/6 Assignment 2

Tree-based Competition Tournament Selection

Release date: 03-03-2016

Hand in date: 11-04-2016 (23:59 via Blackboard)

Feedback date: 02-05-2016 (via Blackboard)

Aims and Objectives

The aim of this assignment is to test your ability to implement and use some basic data structures.

The objective is to write a system for managing a single elimination style competition.

Overview

You are asked to develop a system for tournament elimination. This is the type of approach used in tennis or football championships. In each game, two teams or players compete. One is eliminated (the loser), whilst the other continues to the next round or is declared 'winner'.

Code is provided to do the following:

a) read a list of players (e.g. individuals or teams) from a text file (one per line),
b) draw the competition tree
c) handle text based input and output.

You will need to:

1) complete the system by implementing a class that decides which teams should play in each game.
2) Use your final program and run it on the file given in Appendix 1 of this document.

Requirements

You are required to:

1) implement the competition logic by implementing a binary tree and methods to iterate over it in the correct order.
2) implement a class or classes that implement both the *IManager* and *IBinaryTree* interfaces provided, in order to complete the system outlined in 1) above.

The following gives an indication of how the marks are distributed and some suggestions for implementation:

1) *Class structure* **(20%):** Implement a sensible structure for the classes, including declaration of classes, fields, appropriate methods defined properly etc. The principle public class in your submission should implement the *IManager* interface. You will also need a class to represent the game tree that implements the *IBinaryTree* interface, to allow it to be printed

to the console. To run the competition you can add a *main* method to your class, create a *CompetitionManager* and call its *runCompetition* method.

2) *setPlayers and tree construction* **(30%):** The tree should be constructed when the *setPlayers* method of the *IManager* interface is called. The tree should be initialised with the players at the leaves of the tree, with roughly half of the players in the left subtree and half in the right subtree at every node. A suggestion is to use a recursive method, passing half the list of players to the left subtree and half to the right subtree recursively until the list of players is only one long, where the name of the player (or team) is stored in the node and the recursion stops.

3) *Tree traversal including hasNextMatch, nextMatch and setScore methods* **(30%):** The system should decide which match to play next (if any) and set the result after input from the user. The result should be stored by setting a score value in the left and right child nodes and copying the player/team name of the winning child to the parent node. To keep track of matches a suggestion is to use a stack and a queue as follows (see Appendix 2 for an illustration):
    a. Queue can be used to perform a *breadth first traversal* of the tree. The root is added to the queue. Whilst the queue is not empty, the front of the queue is removed and if its children are not null they are then added to the back of the queue.
    b. The stack can be used to store the matches (as tree nodes, in reverse order) by adding the node to the stack every time a node (with two non-null children) is taken from the queue during the breadth first traversal.
    c. Matches can be created by popping tree nodes from the stack when *nextMatch* is called. A variable should store the current tree node in order to update the score and winner when the *setScore* method is called. The *hasNextMatch* method just needs to check that the stack is not empty. You may use *java.util* classes for the stack and queue, you do not need to implement your own.

4) *Documentation **(20%)***. You are not asked to provided a separate report, but you should thoroughly comment your code using correctly formatted javadoc comments. This should:

    a. explain any unusual implementation features,
    b. explain any problems you have faced (including a clear statement that a particular feature isn't working) and
    c. reference any third party code or other resources /fragments of code that you had help with/etc.

Submission

1. You should submit only a **single** *.java* file containing all your code and documentation (in the form of comments).
2. Your code **must** be defined in the package *cs21120.assignment.solution* and all of your code should be contained in a **single file** called *BTSingleElim<userid>.java* , where *<userid>* is replaced with your Aber userID. You must use the class name *BTSingleElim<userid>* for the

primary (public) class, and any support classes you implement should be inner classes of this i.e. defined within the same file and within the scope of your primary class.

Marking

1. This assignment will use an element of automated marking. *JUnit* tests will be applied to your code, so it is important that you follow the specification exactly.

2. It is essential that you follow the instructions for naming and packaging of your classes. Also, **do not modify** any of the supplied code, including package names. If you find any bugs please report them.
3. A marking grid is included in the appendix which indicates the expected

Resources

1. For this assignment you are permitted to use classes from the java collections classes (java.util.* package). You should implement your own classes for the binary tree data structure and the implementation of *IManager*.
2. Any resources that you use must be acknowledged in the javadoc comments.
3. You are provided with compiled versions of the *Match* class, the *CompetitionManager* class, the *TreePrinter* class, the *IBinaryTree* interface, the *IManager* interface with accompanying html documentation and some example input data.

Academic conduct

1. The work in the assignment must be your own.
2. Do not share code with your colleagues and ensure that your code is securely stored and not accessible to others.
3. All sources should be properly referenced with a citation. Also help obtained (e.g. from demonstrators) should be acknowledged.
4. Documentation must accurately reflect your code, any discrepancies between the code and documentation will be investigated at marking time.

References

The java code for drawing the tree (TreePrinter.java) was adapted from here: https://gist.github.com/MightyPork.

Support

If you have any problems with understanding or completing the assignment please ask for help, either through advisory, by email or via the Blackboard forum for the assignment.

Marking Grid

Aspect	70-100%	60-70%	50-60%	40-50%	0-40%
Correct Functionality	All interface methods implemented and behave in the expected manner.	Most methods implemented and function in the expected manner, but there may be some bugs in more complex code.	Most methods function as expected, but may have bugs in simpler or more complex code.	Code compiles and some methods function as expected, but may have some serious bugs.	Code largely non-functional, may not compile or have serious flaws.
Class Structure	Clear and detailed declaration of classes, fields, appropriate methods. Consistency of coding conventions, properly labelled code that is easy to read.	Mostly clear and sensible layout of code, classes and fields and appropriate use of conventions.	Reasonably readable code and appropriate structure, but may have some elements which are ambiguous or where structure is not entirely clear.	Some evident structure, but may lack clarity and the purpose /functionality of some fragments of code may not be obvious	No coherent structure or sections/fragments of code may be mixed with one another with no clear separation.
Documentation	Clear and detailed description of algorithm, javadoc correctly formatted and complete, good use of language. Good use of citations. If the algorithm and implementation is very simple it will be difficult to achieve this grade.	Mostly clear and detailed description of algorithm. javadoc correctly formatted and complete. Good use of language, and citations. If the algorithm is very simple it will be difficult to achieve this grade.	Description of algorithm that may be lacking in detail (or could be a very simple algorithm), javadoc mostly correctly formatted and complete, reasonable use of language, some citations.	Some documentation present, but may have gaps or lack detail in describing the algorithm or implementation. Some poor use of language/grammar and citations may be limited or missing.	Documentation largely missing, incorrectly formatted or missing javadoc, poor use of language/grammar, citations missing.
Self Evaluation	Self evaluation complete with an estimated mark and comment for each part and overall.	Self evaluation mostly complete with some comments missing.	Self evaluation with some comments or marks missing.	Self evaluation with marks but no comments.	Self evaluation missing.

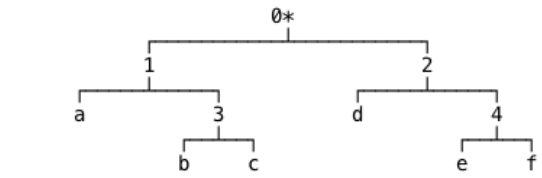
Appendix 1: Example run of program

Below is an example competition with 6 teams. The competition tree is printed before and after the competition for illustration. Matches are played from right to left in each round, and a score of 0 is initialised before each match is played.

```
IManager selected successfully.
                                                                                     (not played)
                                              (not played)                                                                   (not played)
                       Manchester United: 0                       (not played)                         Bangor: 0                                 (not played)
                                                         Liverpool: 0      Aberystwyth: 0                                             Swansea: 0       Newtown: 0
Player 1: Swansea
Player 2: Newtown
1 0
Player 1: Liverpool
Player 2: Aberystwyth
0 10
Player 1: Bangor
Player 2: Swansea
2 1
Player 1: Manchester United
Player 2: Aberystwyth
0 5
Player 1: Aberystwyth
Player 2: Bangor
4 1
Winner is: Aberystwyth
                                                                                    Aberystwyth: 0
                                             Aberystwyth: 4                                                                   Bangor: 1
                       Manchester United: 0                      Aberystwyth: 5                        Bangor: 2                                  Swansea: 1
                                                         Liverpool: 0      Aberystwyth: 10                                            Swansea: 1       Newtown: 0
```

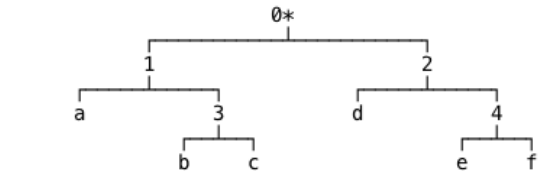
Appendix 2: Example of tree traversal

The following example illustrates the algorithm to initialise the stack of nodes used to decide matches. Internal nodes are labelled with a number for illustration, and leaf nodes are labelled with a letter. An asterix (*) indicates the current node being processed.



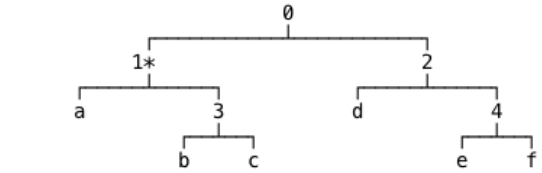
Queue: 0,
Stack:

The traversal starts at the root, with the queue initialised with the root and the stack empty.



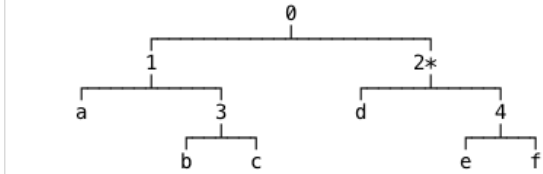
Queue: 1, 2,
Stack: 0,

The front of the queue (0) is removed and its two children added to the back of the queue. As the node has two children it is added to the stack.



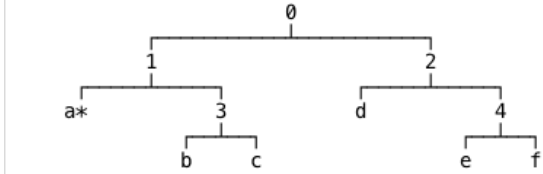
Queue: 2, a, 3,
Stack: 0, 1,

As above for the next node off the queue (1).



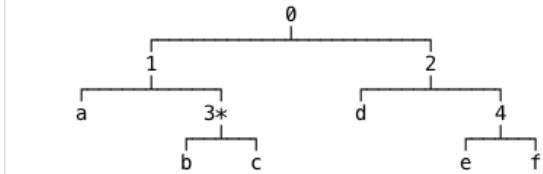
Queue: a, 3, d, 4,
Stack: 0, 1, 2,

Same again for the next node (2).



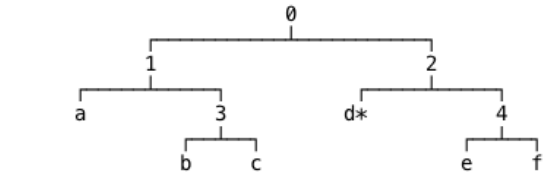
Queue: 3, d, 4,
Stack: 0, 1, 2,

This time the node (a) has no children, so it is not added to the stack and no children are added to the queue.



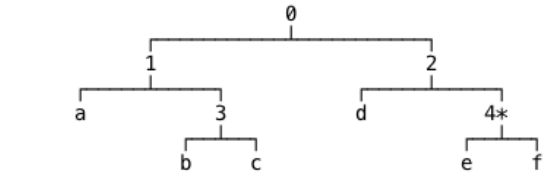
Queue: d, 4, b, c,
Stack: 0, 1, 2, 3,

The node (3) is added to the stack and its children are added to the queue.



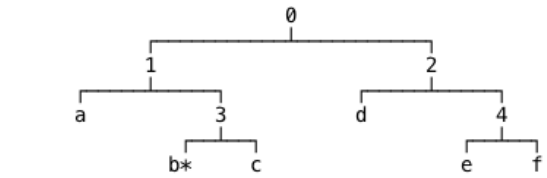
Queue: 4, b, c,
Stack: 0, 1, 2, 3,

Again, the node (d) has no children so it is not added to the stack.



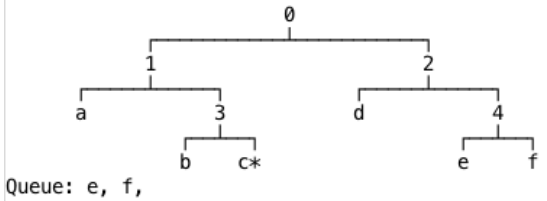
Queue: b, c, e, f,
Stack: 0, 1, 2, 3, 4,

Children of node (4) added and node put on stack.



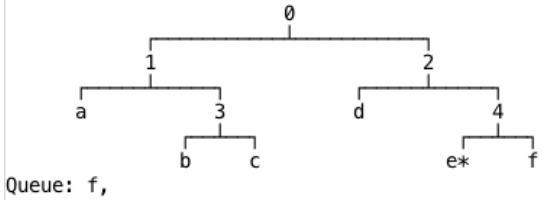
Queue: c, e, f,
Stack: 0, 1, 2, 3, 4,

Node (b) has no children, so not added to stack.



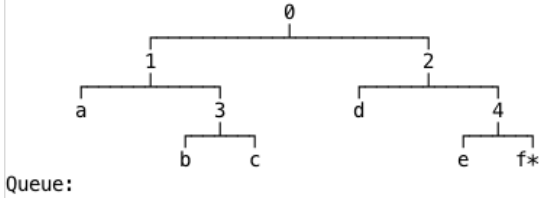
Queue: e, f,
Stack: 0, 1, 2, 3, 4,

As above for node (c).



Queue: f,
Stack: 0, 1, 2, 3, 4,

As above for node (e).



Queue:
Stack: 0, 1, 2, 3, 4,

As above for node (f).

Queue is empty so done. Nodes should be popped off the stack (top at right) so order of matches will be 4, 3, 2, 1, 0 as required.