York University
CSE 2011 - Binary Tree
Instructor: Jeff Edmonds

Your Tasks:

1. Read and understand:
   - This current document
   - Assignment 2
   - http://www.eecs.yorku.ca/∼jeff/courses/3101/ass/steps0.pdf
   - http://www.eecs.yorku.ca/∼jeff/courses/3101/ass/steps1.pdf
   - http://www.eecs.yorku.ca/∼jeff/courses/3101/ass/steps2.pdf
   - Assignment 1.
2. Run Jeff's MainGUI code that calls the TreeWalker methods that you will have to write in Assignment 2.
3. Write up Assinment 1 answering questions about this material.
4. Code Assignments 2, 3, and 4.

Tree Walk-About: Your Assignment 2 task will be to write and test a program that allows a user to walk about and modify a binary tree.

AVL Trees: Another assignment will be to automatically search a *binary search tree*, add and delete nodes, and keep the tree almost balanced in the process.

Evaluate, Differentiate, Simplify: Your Assignment ? task will be to write and test a program that allows a user manipulate, evaluate, differentiate, and simplify polynomials represented by binary trees.

Graph: Maybe I will get you to change these trees into directed graphs and then do Dijkstra's shortest paths on them.

Parse: Maybe I will get you to modify the parser.

Don't panic: Available is the code for the tree written by Franck van Breugel (see *http://www.eecs.yorku.ca/∼jeff/courses/2011/franck* for more such code) and the code for entering and displaying the tree written by Jeff Edmonds. You need only play with a few pointers.

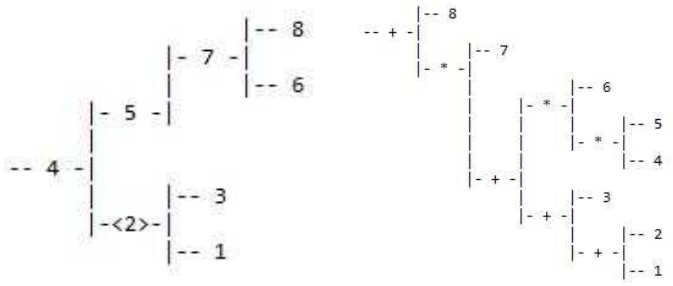

Figure 1: (a) A tree created by parsing the string "((1 2 3) 4 (null 5 (6 7 8)))". You have to rotate your head to see it. As the user walks around, the current node is indicated with < .. >. (b) An expression tree created by parsing the string "(1+2+3+4*5*6)*7+8".

Before starting to code yourself, lets try to understand the code provide, learn a few things, and try a few things.

Files: See http://www.eecs.yorku.ca/~jeff/courses/2011/ass/tree.zip (bin/*.class and src/*.java)

- MainGUI: The class MainGUI has a main routine that runs a graphical interface.
- MainTree: A second main routine that Jeff used to test his routines.
- TreeWalker: The code you are to write. I gave you the .class of my implementation to run so that you see how it is too work. (I have been told that decompiling it would be more work than writing it yourself.) I am also giving you the .java with the headers and not the actual code.
- IOTree: The class Jeff wrote to input and output trees.
- Element: The class Jeff wrote to define what will be contained in each element of the trees.
- BinaryTree: The class Franck wrote to implement binary trees. All the other classes are used by it. (BTNode, InspectableBinaryTree, InspectableContainer, InspectablePositionalContainer, InspectableTree, InvalidPositionException, LinkedBinaryTree, Position, PositionalContainer, TNode)

Running the GUI: My son Josh learned in grade 10 how to make a graphical user interface. He helped me make the class MainGUI. Executing it will bring up a window with a tree in it. The Figure 1.a tree is produced by typing the string "((1 2 3) 4 (null 5 (6 7 8)))" in the top line of the window and hitting return. The b figure is produced from "(1+2+3+4*5*6)*7+8". See the section *Parsing a String with a Grammar* below to understand how to describe a tree with a string in this way. Try building trees. Try pushing the buttons to see what they do.

Object Oriented Code: Before writing MainGUI, Jeff used the routine MainTree to test his code. The code to produce the trees in Figure 1 is as follows. It builds each tree by parsing a string description of the tree and then prints these out as seen.

```
IOTree tree = new IOTree("((1 2 3) 4 (null 5 (6 7 8)))");
tree.makeCurrent(tree.leftChild(tree.root()));
System.out.println(tree.toString());
System.out.println(tree.PrettyPrint());
tree = new IOTree("(1+2+3+4*5*6)*7+8"); // These strings need a '+' or a '*'
System.out.println(tree.PrettyPrint());
```

Understanding this Code:

- The first line constructs a new IOTree object and has the variable *tree* point at it. The *System Invariants* for this *abstract data type* are as follows.
  - The field *tree.root* points at the root node. In Figure 1.a the root node contains a 4.
  - $tree.size = 8$ knows the number of nodes in the tree.
  - For each node *node*, $node.left$, *node.right*, *node.parent*, and *node.element* must always point at its left child, right child, parent, and data or contains the special value *null* indicating that such a structure does not exist. For example, the node containing 2 has left child 1, right 3, parent 4, and element 2 (among other data). Both nodes 1 and 5 have left child $node.left = null$ and the root node has $root.parent = null$.
  - The *element* data stored in each node contains a char c, an int x (printed as in integer), a string s, a flag indicating which combination of these should be displayed, and boolean *current* stating whether this node is the current node. In the figure, the current node is node 2.
- The constructor in the first line of code calls a method *ReadTree* within the tree object that parses the string description of a tree and constructs the nodes in the tree.
- The second line makes the tree's root's left child the *current node*. In the figure this is node 2.

- The third line calls a method *toString* which is a method that is standard within most objects for returning a string description of all the data in the object. In this case, the string returned is an *infix traversal* of the tree that looks much like the string used to produce the tree. $System.out.println()$ then prints this string.
- The fourth line calls *PrettyPrint* which produces a string displaying the tree as seen in the figure.
- The last two lines do the same to produce the tree in Figure 1.b.

Hierarchy of Interfaces and Classes: An *interface* for a class of objects is supposed to be its contract with the outside world. Jeff complains that it tends to only give the names of the *methods* (i.e. procedures) and types of their parameters. He would like to see clear *preconditions*, *post-conditions*, and *system invariants* included. The interface *InspectableContainer* gives you a container of elements that you can iterate through. *Position* defines the position object, which is acts like a location or cursor within some larger object. The *InspectablePositionalContainer* extends InspectableContainer so that the elements are positions. *PositionalContainer* extends this to allow you to not just iterate through these positions but also replace them. *InspectableTree* gives you a tree where each node is a position and has an ordered list of children nodes. *InspectableBinaryTree* extends (or restricts) this so that each node has a left and right child. *BinaryTree* includes being able to change the positions and the links between them. *LinkedBinaryTree* and *BTNode* are our first *classes*, i.e. they actually have code for data and methods that act upon that data. They respectively *implement* BinaryTree and position. A LinkedBinaryTree object contains two variables, *root* and $size$. A BTNode claims to contain a left child, a right child, a parent, and an element, but really it contains a pointer to each of these. Because we do not want to pin down what this will be a tree of, the type of this element is *Object*. This is the pinnacle class of which all other types are subclasses. Jeff wrote *IOtree* extending LinkedBinaryTree to contain two new methods. The first constructs a new tree for the object by parsing a string. The second print out this tree in a "pretty" way. In order to be able to do this, the elements needed to contain data in a specified format. Hence, Jeff made a new class *Element* to extend the class *Object*. To carry this even further, you are to write a class *TreeWalker* to extend IOTree that allows the user to walk around the positions in the tree.

Overriding Methods: When a class extends another class it can override the methods of this class. Also when an object from one class contains an object from another, the two classes can have similar methods. Be careful to keep track these.

BTNode: BTNode getLeft()
If *node* is a BTNode, then $node.getLeft()$ returns the node's left child node.

LinkedBinaryTree: Position leftChild(Position position)
If *tree* is a LinkedBinaryTree, and *position* is a position (i.e. a BTNode) within it, then $tree.leftChild(position)$ returns the position's left child position. The code *casts position* to be a *node* BTNode, checks for errors, and then calls $node.getLeft()$.

IOTree: Position leftChild(Position position)
IOTree over rides this method because it does not think it is an error when a null pointer is returned.

TreeWalker: Position leftChild(Position position)
TreeWalker over rides this method to automatically add a new child if such a child does not already exist.

Be aware of who you are and of *Java variable scoping*.

Main: If you are the main method and you have a variable *tree* of type TreeWalker and you want to move your *current* node, then you call the above method with $current = tree.leftChild(current)$.

TreeWalker: If you are a TreeWalker object, then you call it with $position = this.leftChild(position)$ or even more simply with $position = leftChild(position)$. Note this method naturally works on the tree that you are with the version of the method from your class. Note, TreeWalker does not

know about the current node. (Though it does turn on and off the *element.current* flag.) As far as it knows, there could be many current nodes.

Super: Again suppose your are a TreeWalker object, but you want to call the version of $leftChild$ from your *super class* IOTree. Then you call it with $position = super.leftChild(position)$. One place that you would do this for sure is within your code for $leftChild$.

Not Overwritten: As a TreeWalker object, *root*(), *this.root*(), and *super.root*() all do the same thing, accessing the same variable *root* because TreeWalker does not overwrite IOTree's method *root*().

Following Pointers: Recall the second line in the above code that makes the tree's root's left child the current node.

```
tree.makeCurrent(tree.leftChild(tree.root()));
```

This is the object oriented way of doing it. It can be broken into multiple lines.

```
position = tree.root()                  % returns the position of the root.
position = tree.leftChild(position)     % moves that position to its left child.
tree.makeCurrent(position)              % makes this position the current node.
```

For learning purposes, it is good to understand following pointers. If we did not have to worry about privacy and types, this line could simply be

```
tree.root.left.element.current = true;
```

Follow the path of pointers. *tree* is the object tree. *tree.root* points at the node to be the root. $tree.root.left$ points at this root node's left child. $tree.root.left.element$ points this left child's data element. $tree.root.left.element.current$ is this element's flag for indicating whether or not this node is the current node.

Private Variables and Casting Types: The object oriented way is for a class to make *public* only what the user really needs to see to order to use the object in an "abstract" way and to make *private* all of the "implementation details." This both keeps things simpler for the user and prevents him from unintentionally and intentionally messing up the *system invariants* of the data structure. For example, the variable *size* must contain the number of nodes in the tree. Hence the user should not be able to change it on his own. Instead, there will be a method *size*() that returns the size, while only the internal methods are allowed to change its value, which it only does when the number of nodes changes. The class might also want to do some error checking or extra side effects as the user accesses this variable. I confess, I (Jeff) can find this a little silly when an object has both a public "int getx()" and a "setx(int x)" method that do nothing except get and set the value of x and yet the variable x is made private. This fores the user to write "y = t.getx()" or even "y = t.x()" instead of "y = t.x". Partially the frustration is that it forces me to remember more names, i.e. x, vs $x()$, vs $getx()$ vs $getX()$ .... Recall the above line of code.

```
tree.root.left.element.current = true;
      vs
tree.root().getLeft().element().current = true;
```

The first line would work great, if we had public access to all of the variable fields of the objects. Given we must access them through methods, the second line is necessary. Note that I broke the rules when I wrote the class *Element*. This is why the code ends with $current = true$ and not $setCurrent(true)$.

There are also a lot more types to keep track of. When *tree* is a $LinkedBinaryTree$ or is an $IOTree$, then *tree.root* is of type $BTNode$. However, this might be considered an "implementation" detail. All the user "needs" to know is that it is a *Position*. I suppose this is why the method *tree.root*() returns a *Position*. The following code is equivalent to my one complicated line above.

```
Position position = tree.root();
position = tree.leftChild(position);
Element element = (Element) position.element();
element.current = true;
```

- The first line retrieves the position (node) at the root.
- The next moves the position to that position's left child.
- The interface *Position* knows that it contains an data element and hence has a method *element*() to get it for the user. However, in order to be abstract, the type of this element is *Object*. The class $BTNode$ wants to be a tree of any type of object that the user wants. Hence, when it implements *Position*, it adds the $left$, $right$, and *parent* pointers, but it carries on with this abstraction by having its variable *element* variable still of type *Object*. When extending this class to $IOTree$, I wanted be able to input and output actual elements. Hence, I replaced the *element* being of type *Object* to it being of type *Element*. Before we can use it as such, we need to *cast* it (i.e. change it's type) from type *Object* to type *Element*. This is done in the third line.
- Since the class *Element* broke the rules and made its variable *current* public, the forth line can set it to be true.

Being a person who wants to be able to freely follow pointers, I wanted to put all that code into one line.

```
tree.root().getLeft().element().current = true;
      vs
((Element) ((BTNode) tree.root()).getLeft().element()).current = true;
      vs
tree.makeCurrent(tree.leftChild(tree.root()));
```

This first line would have worked fine had their not been type checking. I am very grateful to Eclipse for showing me the error of my ways and for showing me how to fix it. In the second line *tree.root*() returns a *Position* which needs to be cast as a $BTNode$ before its $getLeft()$ method could be used. Similarly, *element*() returns an *Object* which needs to be cast as an *Element* before its *current* variable could be used. The third line is the object oriented way of doing this. It does seem easier.

Parsing a String with a Grammar: In 2001 you will learn about *Context Free Grammars*. Using these, both the concept of a *tree* and of a string describing the tree can be defined recursively. Jeff's program uses two such grammars.

The Tree Grammar: The first has the rules

$$T \Rightarrow (TNT) \mid N$$
$$N \Rightarrow Int \mid Str \mid \ < Int, Str > \ \mid null$$

Here T represents a tree and N represents a node. The $'|'$ represents *OR*. Hence the second rule says that a node could be an integer, a string, or a record containing a integer as a key and a string, or it could be null. In that last case, the node does not really exist. The rule $T \Rightarrow N$ says that a tree could be a singe node. Hence "1" is a string representing the tree with the one node with the integer 1. In contrast, "null" produces the empty tree with no nodes. The class pointer tree contains the value null. The rule $T \Rightarrow (TNT)$ says that a tree could also be a root node in the middle with a subtree to its left and a subtree to its right. Hence "(1 2 3)" is a string representing the tree with root node 2, left child 1 and right child 3. Similarly "((1 2 3) 4 (null 5 (6 7 8)))" is a string representing the tree with root node 4, left subtree represented by string "(1 2 3)", and right subtree represented by string "(null 5 (6 7 8))".

An Expression: A tree can also represent an expression like that in Figure 1.b. As long as the description string contains a least one '+' or '*', then this second grammar is used.

tree = new IOTree("(1+2+3+4*5*6)*7+8");

Notice that brackets are needed to differentiate between the order of $(1+2)*3$ and $1+2*3$. This order of operations is represented, however, quite elegantly in the tree structure.

The grammar for expressions has as its first rule $Exp \Rightarrow Term + Term - \ldots + Term$ because an expression is a sum of terms. Its second rule is $Term \Rightarrow Factor * Factor / \ldots * Factor$ because a term is the product of factors. Its last rule is $Factor \Rightarrow (Exp) \mid Int \mid Var$ because a factor is either an expression in brackets, an integer, or a variable like x.

We will not really need them for assignment 1 or 2, but they are fun too play with. In a later assignment, you will need to evaluate such expressions. You will need to use calculus to differentiate them. You will also need to simplify them. I mention them now so that you can start thinking about it. However, these algorithms need recursion so we will leave these until recursion is covered.

Null Pointers and the Empty Tree: Consider node 5 in Figure 1.a. It has a right but does not "have" a left child. Its $node.left$ pointer has the value *null*. If the variable *root* contains the *null* pointer, then we call this the *empty tree* because it contains zero nodes. Jeff finds these types of trees interesting. Neither LinkedBinaryTree nor Expression trees, however, include such trees.

The class LinkedBinaryTree has a *system invariant* that its trees have two types of nodes: *internal* nodes with two children and *external* nodes with no children. When a tree is constructed, all of its constructors start the tree with at least one node. The tree can never be empty. The only method that adds nodes is $expandExternal(position)$ which gives a node with no children two children. The only method that removes nodes is $removeAboveExternal(position)$ which removes an external node and its sibling. Both of these maintain that there are only these two types of nodes.

Similarly, the trees representing *Expressions* as in Figure 1.b have only two types of nodes: *operators* and *operands*. The operator nodes have operators '+' or '*' as their element's data and they have both a left and right subtrees. The operand nodes have integers or variables x, y as their element data and they have neither a left nor a right subtree. A recursive program evaluates such a tree by recursively evaluating the left and right subtree and then applying the root operator on these left and right evaluations. The base case is an operand whose value is given.

A recursive algorithm on input I can recursively give *friends* any input I' as long as its is both smaller and meets the precondition. Here our input is a tree. Hence, the input that any recursive algorithm gives his friends must also be a tree. For this reason, it is helpful to view a tree recursively. Instead of saying that each node has a left and right child node, when speaking recursively we say that it has a left and a right child subtree. Sometimes a node is missing a child node, but it is never missing a child subtree. For example, the tree in Figure 1.a represented by the string "((1 2 3) 4 (null 5 (6 7 8)))" has at its root the node 4. This node has "(1 2 3)" as its left subtree and "(null 5 (6 7 8))" as its right subtree. Similarly the tree "(null 5 (6 7 8)" has at its root the node 5. This node has "null" as its left subtree and "(6 7 8)" as its right subtree. This left subtree "null" is a perfectly good tree. It just happens to be the empty tree. The code then no longer needs to have a lot of cases based on how many children the root has. The base case will be the empty tree.

For this reason, IOTree makes the following changes to LinkedBinaryTree. The constructor LinkedBinaryTree() constructs the initial tree to have one node. Instead, constructor IOTree() constructs it to be the empty tree. In LinkedBinaryTree, the method Position leftChild(Position node) throws an error if the node being considered does not have a left child. In IOTree, this same method returns the value Position=null in this case. Therefore, your code when using this method must check yourself whether the returned value is null.

Warning: Before using any variable, make sure it is not null.

Binary Search Trees: We say that a tree is a *binary search tree* if for **each** node in the tree all the keys in the node's left subtree are at most the key at that node which in turn is at most all the keys in the node's right subtree. See the order of the keys in the tree in Figure 1. Play around to convince yourself that a tree produced using the $T \Rightarrow (TNT)$ grammar is a binary search tree if and only if the numbers

in its string "((1 2 3) 4 (null 5 (6 7 8)))" appear in sorted order. The locations of the brackets and the nulls change order the nodes are stored but not that the tree has this binary search property.

Infix Traversal: An *Infix Traversal* of a tree visits each node in the tree in the following order. It first recursively visits each node in the left subtree of the root. Then *in* the middle, it visits the root node of the tree. Finally, it recursively visits each node in the right subtree. Play around to convince yourself that such a traversal of a binary search tree visits (prints) the nodes in sorted order. Similarly, for any tree, the order the nodes are visited is the order they appear in its string "((1 2 3) 4 (null 5 (6 7 8)))".

Mostly Balanced (AVL)) Trees: The time needed to walk from the root to a node is the node's depth. This depth is, of course, bounded by the tree's height. We say that a binary tree is mostly balanced if most all its branches are filled to mostly the same depth and hence its height is at most $\approx \log n$ where n is the number of nodes. (Note that $\log n$ is the number of time that you need to divide n by two until you get one.) For example, a tree is said to be an *AVL Tree* (named after its inventors) if and only if for **each** node in the tree, the heights of its two child subtrees differ by at most one. To help keep the tree balanced, an AVL stores at each node, the height of the subtree that it is a root of. In the tree in Figure 1.a, node 2 stores that its subtree has height 2 and node 5 stores 3. Then we know that node 4 is sufficiently balanced because this 2 and 3 of its children differ by at most one. In contrast, node 5 is not sufficiently balanced because its subtrees are are of height 0 and 2. Because of this imbalance, the entire tree is NOT an AVL tree.

A List of the Methods Available:

TreeWalker: Your task is to write these. See their descriptions below. Stated is the number of lines of code in Jeff's solution.

- Position root(Position position) (lines of code = 7)
- Position parent(Position position) (lines of code = 9)
- Position leftChild(Position position) (lines of code = 8)
- Position rightChild(Position position) (symmetric)
- Position rotateR(Position position) (lines of code = 15)
- Position rotateL(Position position) (symmetric)
- Position first(Position position) (lines of code = 4)
- Position last(Position position) (symmetric)
- Position next(Position position) (lines of code = 13)
- Position previous(Position position) (symmetric)
- Position set(Position position) (lines of code = 11)
- Position insert(Position position) (lines of code = 5)
- Position delete(Position position) (lines of code = 8)

IOTree: (Jeff's)

- Constructors construct a tree using the stated information:
  - IOTree()
  - IOTree(Object element)
  - IOTree(Object element, BinaryTree left, BinaryTree right)
  - IOTree(String toParse)
- Moves to stated position (node) from stated position (node). These could return null.
  - Position parent(Position position)
  - Position leftChild(Position position)
  - Position rightChild(Position position)
- Get or Set the value in the stated position (node)
  - Position makeCurrent(Position position)

  - void notCurrent(Position position) % (this sets node's flag to false.)
  - int value(Position position)
  - void setValue(Position position, int x)
  - String string(Position position)
  - void setString(Position position, String s)
- Produces a string representation of the entire tree.
  - String toString()
  - String PrettyPrint()

Use in IOTree because inherited from LinkedBinaryTree: (Franck's)

- void setSize(int size) Jeff added this.
- int size()
- boolean isEmpty()
- void setRoot(Position position) Jeff added this.
- Position root()

BTNode: (Franck's)

- Constructors BTNode() and BTNode(Object element, BTNode left, BTNode right, BTNode parent)
- Object element()
- BTNode getLeft()
- BTNode getRight()
- BTNode getParent()
- void setElement(Object element)
- void setLeft(BTNode left)
- void setRight(BTNode right)
- void setParent(BTNode parent)
- String toString()

Element: (Jeff's)

- Constructor Element(char c, int x, String s, char type)
- char c;
- int x;
- String s;
- char type $\in \{c, x, s, k\}$;
- boolean current;

In IOTree but I don't think they will be useful for this assignment: (Franck's)

- Iterator children(Position position)
- boolean isInternal(Position position)
- boolean isExternal(Position position)
- boolean isRoot(Position position)
- Iterator elements()
- Iterator positions()
- void swapElements(Position first, Position second)
- Object replaceElement(Position position, Object element)
- Position sibling(Position position)
- void expandExternal(Position position)
  This gives a node with no children two children.

- Object removeAboveExternal(Position position)
  This removes an external node and its parent.
- String toString()

Hopefully, you now know enough about the code provided and binary trees to start thinking about the code you must write.