HW5: Building and Processing an N-ary tree

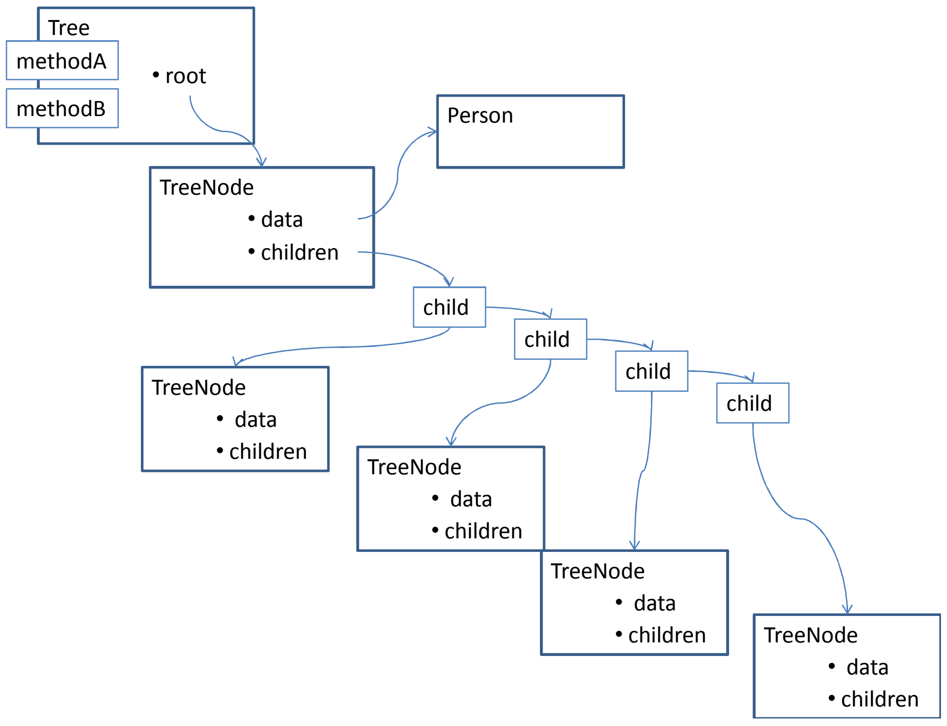
DUE DATE: Monday March 3rd, 2008

An N-ary Tree is a fixed data structure that stores information through nodes. We have seen some examples already: the instruments in the orchestra organized in sections, and the players in the baseball game organized by their roles on the field.

In general, an N-ary tree is more structured. Each node is a data structure itself that contains two bits of information: the data for that node and a collection of references to the next nodes in that sub-tree. We suggest that you use a linked list for the collection of child references. Notice that the relationship between a node and its children is again one of containment, not inheritance.

As with a linked list, you only know about the first node – the root of the tree. That node has its data element and then points you to the next nodes. To process the tree, we suggest that you start at the root node and move about the tree using recursion.

Please note: Word has the ugly habit of capitalizing words at the beginning of a line, which is sometimes not desired. In the description below, remember the standard naming convention: class names begin with a capital letter; methods and field names begin with a small letter. Remember also that fields should be private – if you need to read or write them from outside the class, write set… and get… methods to do this, but only when necessary.



For Homework 5 you will be creating and processing a representation of the British royal family, starting with Queen Victoria. You will be provided with four files:

- `royalty.txt` contains information about each individual,
- `familytree.txt` defining the parent / child relationships,
- `Person.java` – the class that contains the description of a person, and
- `TreeNode.java` – the class holding one Person object and a linked list of children of that person.

You will be turning in this file:

- `FamilyTree.java` – the class that holds the tree root and all of the tree processing methods

Person.java

This class holds information provided for a person. If you look at the file `royalty.txt`, the following fields are provided in the data source:

- ID – a unique symbol identifying the person
- Name
- DoB – date of birth
- DoD – date of death (-1 if still alive)
- Spouse – the ID of the person's spouse
- Title – the person's title

In addition, to support the logic to follow, you will need one more field:

- Children – a linked list initially empty to contain the IDs of the children – suggestion: LinkedList<String> might work well here.

The constructor provides values for each field except the linked list that is initialized with no content.

The toString() method and main function make sure this class is working properly.

TreeNode.java

The TreeNode class is similar to the LLNode class for linked lists. It contains the following fields:

- Data – a Person object
- Children – a linked list of the children of this class.

The constructor accepts only a Person object, and initializes but leave empty the children list.

The method call addChild with another TreeNode object as a parameter adds it to the list of children.

The toString() method using the Person's toString() for each node and the test main method allow you to verify the behavior of this class.

FamilyTree.java

Now, you're ready to build the family tree. Start with the constructor with this header:

```
public FamilyTree(String pplFile, String treeFile, String rootID) {…};
```

It accepts the two file names , royalty and family, and the ID of the tree root. It should have the following capabilities:

1. Read the royalty file

- Create a new linked list called People – suggestion: LinkedList<People> might work well here.
- Read each line of the royalty file. If the line begins with "%", just skip that line.
- Parse each line with a tokenizer, construct a Person object for each line. Hint: don't use space as a delimiter, but each time you extract a token, use trim() to remove leading and trailing spaces.
- Add each person to the People list.

2. Read the family file

- The first token in this file is the ID of a parent.
- The remaining tokens are the identities of that parent's children
- Write a function that is passed an ID as a string and looks up the Person with that ID on the People list:

```
public Person find(String ID) { … };
```

- As you read the file, use this function to find the object in the People list matching the first ID. Then add each subsequent ID to the children of that node.

3. Construct the tree

- Now, write a recursive program with this header:

```
public TreeNode makeNode(String ID) {  … };
```

Its function is to find the Person with the given ID and put that Person object into a new TreeNode. However, it must also set the children of that tree node as follows:

a. Use the find function above to find the Person object with the given ID.
b. Create a new TreeNode object containing that Person
c. Since that Person object already contains a list of the IDs of its children, we can a traverse that Person's children IDs
d. For each ID on that list, make a call to makeNode(…) to create a TreeNode with that ID.
e. Add that TreeNode object to the children of your TreeNode object.

- You start the recursion by setting the root of the tree as follows:

```
Root = makeNode(rootID);
```

4. Tree Methods

Now we need the following recursive methods:

- toString() – display each person on a separate line.
- Count – count the total number of people in the tree.
- Depth – what is the maximum depth of the tree?
- Oldest – who lived longest. If the doD is -1, use 2008. Hint: write an age() method for the Person class.

5. Test Main

Your test main program which, of course, will expand as you write and test each method, should do the following:

- Construct a new FamilyTree using the files provided, initially using an ID low down the tree like GeorgeVI.
- Print to the interactions window the total number of people in the tree, the maximum depth of the tree, the name and title of the people who died oldest and reigned the longest.

When all these methods are working for the short tree, change the tree root ID to Victoria7 and make sure you can process the full tree.

What to Turn In

- `FamilyTree.java –` the class that holds the tree root and all of the tree processing methods, and

How to Turn In

Turn in using T-Square.