DSA Project 2008

Trees, Iterators and File Systems

Due: 12pm, Tuesday 27th May, 2008
Worth: 20% of your final grade

This semester the programming project will be broken into two parts.

For the first part you will be required to write an ADT, BasicTree. You will be supplied with the interfaces Tree and TreeIterator, and you must implement the specified functionality, and write a short report on your design. You code and report must be submitted by the due date, and is worth 50% of your project mark. You may do this part of the project in pairs if you wish.

The second part will be done in your lab, where you will be required to use a complete implementation of the ADT, Tree, to implement a (very) basic Unix file system simulator. You will have 60 minutes, and must work alone. This is worth 50% of your project mark.

Trees

A tree is a recursive data structure that consists of a set of nodes, where each node contains some data, and a list (possibly empty) of child nodes. There should be one node that every other node descends from called the root node.

Here, a is the root node, and b, c, and d are the children of a. We say b, c, d, e, f, and g are the descendents of a, and a is their ancestor. Trees are covered in topic 12 of the lecture notes, and chapter 11 of the textbook.
The Tree ADT has the following methods specified:

```java
/**
 * @return TreeIterator starting at the root
 * @exception TreeException if root is not defined
 */
public TreeIterator getIterator() throws TreeException;

/**
 * For testing purposes.
 * @return a multiline String representation of the elements in the
 * tree.
 */
public String toString();

/**
 * @return the item contained in the root of the tree
 * @exception TreeException if root is not defined
 */
public Object getRoot() throws TreeException;

/**
 * @param item the new Object to be stored in the root
 * @return the old item contained in the root of the tree
 * @exception TreeException if root is not defined
 */
public Object setRoot(Object item) throws TreeException;

/**
 * @param item the item to be contained in the root
 * @exception TreeException if root is already defined
 */
public void addRoot(Object item) throws TreeException;

/**
 * @return true if the Tree is empty
 */
public boolean isEmpty();
```

The data inside the tree is accessed using a TreeIterator:

```java
/**
 * @return the item of the first child
 * @throws TreeException if there is no child
 */
public Object getChild() throws TreeException;

/**
 * @return the item of parent of the current node
 * @throws TreeException if the current node has no parent
 */
public Object getParent() throws TreeException;

/**
 * @return the item of the next sibling of the current node
 * @throws TreeException if the current node has no parent or next
 * sibling
 */
public Object getNext() throws TreeException;

/**
 * @return the name of the current node
 */
public Object getItem();

/**
 * @return the old item of the current node
 */
public Object setItem(Object item);

/**
 * Creates a node and inserts it as the first child of the current
 * node.
 * @param item the item of the new node
 */
public void addChild(Object item);
```
/**
 * Creates a node and inserts it as the next sibling of the current
 * node.
 * @param item the item of the new node
 * @throws TreeException if the current node has no parent
 **/    
public void insertAfter(Object item) throws TreeException;

/**
 * Creates a node and inserts it as the previous sibling to the
 * current node.
 * @param item the item of the new node
 * @throws TreeException if the current node has no parent
 **/
public void insertBefore(Object item) throws TreeException;

/**
 * Removes the current node and all its descendents and moves the
 * iterator to the parent of the current node.
 * @return the item of the removed node
 * @throws TreeException if the current node has no parent
 **/
public Object remove() throws TreeException;

/**
 * @return true if and only if the current node has at least one
 * child.
 **/
public boolean hasChild();

/**
 * @return true if and only if the current node has a parent.
 **/
public boolean hasParent();

/**
 * @return true if and only if the current node has a next sibling.
 **/
public boolean hasNext();

/**
 * Creates a shallow copy of the iterator, so that no part of the
 * tree is copied, but the Iterator and the path is.
 * @return an Object equivalent, but unequal to this iterator
 **/
public Object clone();
Inner Classes and Clones

In this project you will benefit from a good understanding of inner classes and cloning. While these do not relate directly to ADT's, they are important aspects of good OO design.

You can define one class inside another class (an inner class). The inner class can access all the private variables and all the methods of the outer class. This is discussed in Topic 10.

Cloning is a way of copying an instance of a class. this will be useful since you will find it useful to have two TreeIterators acting on the Tree at the same time.

Every object inherits a clone() method from class Object, but this can only be used if it implements the interface Cloneable.

The clone() method will create an identical copy of the original object, however both the original and the copy will share any variables. Therefore you may override the method to make sure the variables are also cloned.

File Systems

Trees can be used to represent many different things.

In Unix "/" is the root of the directory tree, and "/home" is a child of "/". For the test, you will be required to write a data type for a tree of Strings, and use that data type to simulate a Unix directory tree.

Once you have completed an implementation of Tree you should use your ADT to practise writing the methods specified in the Shell class.

All Input and Output can be handled by the class ShellIO, which is provided. However you will still have to check input to prevent empty, null, or special strings (eg " ..") naming directories.

In the Lab test you will be required to implement a selection of these methods using a provided implementation of Tree.
java.lang.String `changeDirectory(java.lang.String s)`

**Unix Equivalent:** `cd s`
**ShellSim command:** `cd1 s`
Changes directory to named subdirectory

java.lang.String `changeDirectory(java.lang.String[] s)`

**Unix Equivalent:** `cd s`
**ShellSim command:** `cd s` where s is a path
Changes directory to named path

java.lang.String `copy(java.lang.String[] directory, java.lang.String[] location)`

**Unix Equivalent:** `cp directory location`
**ShellSim command:** `cp directory location`
Copys the contents of one directory to another.

java.lang.String `find(java.lang.String s)`

**Unix Equivalent:** `find -n s`
**ShellSim command:** `find s`
finds any subdirectories (recursively) matching the given name

java.lang.String `list()`

**Unix Equivalent:** `ls`
**ShellSim command:** `ls`
Lists the contents of the current directory

java.lang.String `listRecursive()`

**Unix Equivalent:** `ls -R`
**ShellSim command:** `lsR`
Lists the contents of the current directory recursively

java.lang.String `makeDirectory(java.lang.String s)`

**Unix Equivalent:** `mkdir s`
**ShellSim command:** `mkdir s`
Creates a new subdirectory

java.lang.String `move(java.lang.String[] directory, java.lang.String[] location)`

**Unix Equivalent:** `mv directory location`
**ShellSim command:** `mv directory location`
Moves the contents of one directory to another.

java.lang.String `path()`

**Unix Equivalent:** `pwd`
**ShellSim command:** `pwd`
Displays the path to the current directory

java.lang.String `remove(java.lang.String s)`

**Unix Equivalent:** `rm -R s`
**ShellSim command:** `rm1 s`
removes (recursively) the named subdirectory

java.lang.String `remove(java.lang.String[] s)`

**Unix Equivalent:** `rm -R s`
**ShellSim command:** `rm s`
removes (recursively) the directory indicated by the relative path
Here is some sample output from the class ShellIO:

ShellSim-> mkdir a3
ShellSim-> mkdir a1
ShellSim-> mkdir a2
ShellSim-> ls
    a1
    a2
    a3
ShellSim-> cd a1
ShellSim-> mkdir b1
ShellSim-> mkdir b2
ShellSim-> ls
    b1
    b2
ShellSim-> cd ../..
ShellSim-> cp a2 a1/b1
ShellSim-> lsR
      a1
     a1/b1
     a1/b2
     a2
     a3
ShellSim-> cd a1/b2
ShellSim-> pwd
    /a1/b2
ShellSim-> cd ../..
ShellSim-> cp a2 a1/b1
ShellSim-> lsR
      a1
     a1/b1
     a1/b1/a2
     a1/b2
     a2
     a3
ShellSim-> logout
    Good bye!

Marking

The marks for the assignment will be as follows:

- Functionality of Tree and TreeIterator implementations. 15 marks
- Report describing the design of the Tree and TreeIterator classes, the rationale behind this design, and the complexity of each method with justification. Also describe your testing and validation procedure. This should be about 5 pages. 10 marks
- The in-lab test will be assessed on the correctness of the methods (15 marks), the efficiency of the methods (5 marks), and the coding style (5 marks). 25 marks

Help?

- Practise classes are held each a week
- help2200
- Consultation (Wed 9.30-11, rm 2.14)
Good luck!