The UNIX File System

Project 2
Computer Science 61BL, Summer 2009
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

COLLEEN LEWIS, GEORGE WANG, JONATHAN KOTKER, KAUSHIK IYER, DAVID ZENG

Contents

1 Overview

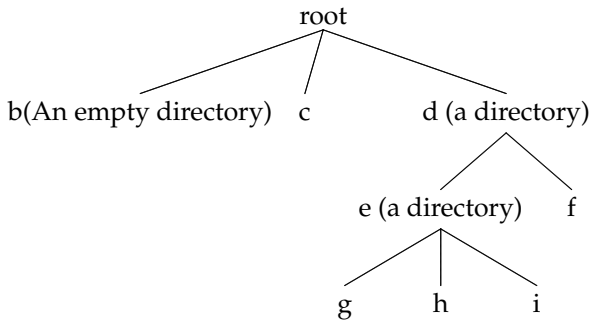
This is a group project; you may (but are not required to) work in a partnership of two students. Submit one solution per partnership, and include both your names in all files you submit. Your final solution to this project is due **Friday, July 24, 2009** at **10PM**. There will be one checkoff, listed in section 6, for this project, that is due by the beginning of lab on Thursday, July 16th. The check-off is described in detail later in the spec.

This project is intended to give you lots of practice with trees and project design. Furthermore, you will become much more familiar with modularizing your code so that pieces fit together through the use of interfaces. Our goal is to give you as much freedom as possible, including the freedom to make very poor design choices. Thus, you should treat the design of the project at least as important as actually writing the code. We will be providing help and advice to help you come up with a sensible design.

1.1 Background

Most of you have had experience working with the UNIX operating system. UNIX provides a tree-structured file system and operations for managing it. In this project, you will implement an interpreter for commands that navigate through, print, create, and delete nodes in a tree that loosely simulates a UNIX directory. (An actual UNIX file system is organized somewhat differently; see The UNIX Programming Environment, by Brian Kernighan and Rob Pike (Prentice-Hall, 1984), for further information.)

The nodes in the tree represent text files and directory files. A directory file contains zero or more text files and zero or more directory files. A text file contains characters, and corresponds to a leaf in the tree. (An empty directory also corresponds to a leaf.) Each file has a name which is a sequence of alphanumeric characters. The root of the tree represents the root directory in the file system. The working directory is also a directory in the file system; the cd command (for "change directory") lets the user move the working directory up and down in the tree. Given below is a sample directory.



Any file may be referenced either with its absolute file name, which specifies the (unique) location of the file with respect to the root, or by a relative file name, which specifies the location of the file with respect to the working directory. The absolute file name is sometimes referred to as the full path name, since it represents the path from the root of the tree down to the file. An absolute file name begins with a slash ("/"). It then contains the name of the subdirectory of the root that contains the file, followed by a slash, followed by the name of the second-level subdirectory that contains the file, followed by a slash, and so on. Note, a slash will never be part of the name of a text file. The absolute file names of the files in the sample directory just given are listed below.

```
/ (the name of the root directory)
/b
```

```
/c
/d
/d/e
/d/f
/d/e/g
/d/e/h
/d/e/i
```

A relative file name represents a path from the working directory to the referenced file. If the working directory contains the file, the relative file name is just the file name itself. If the file is further down in the directory structure, its relative file name is constructed in a similar way to the absolute file name: names of directories on the path are delimited by slashes. If the file is further up the directory structure, the string "../" is used to refer to the parent directory. "../../" is the grandparent, and so on. The root directory is the parent of itself. Suppose that the working directory is e in the sample directory given above. A relative reference to the file c would be

```
../../c
```

Another is

```
../../b/../c
```

Nothing restricts the path to be as short as possible. If the working directory is d, the relative path names

```
e
e/g
e/h
e/i
```

may be used to access all the files in d and its subdirectories.

2 Your Task

2.1 Design and Overview

In this project, we'll consider the advantages of dividing your project into two separate modules. First, you will have a package that stores the data structure of the tree. This will be much like the Amoeba class you study in lab. The second is a package that parses commands from the user and manipulates the data structure. The idea is that much of the low-level pointer manipulation will be done by the first package, and much of the higher-level tasks will be handled by the second package.

These modules are designed to be communicating via an interface that we have provided. We require that you use the interface provided and not write your own. This is so that our testing code will work seamlessly with your design, if you have satisfied the interface. Thus, the only methods in the tree package that any other package may call are the public methods defined in TreeStructure.java. **You may not add any public methods.**

The first module is a fairly generic tree structure located in TreeStructure.java. We are not providing any requirements as to how you should implement your internal nodes, or how the structure should be stored. We have provided a TreeNode interface, designed as a handle for the FileSystem to be able to pass information to the Tree without having access to the private and protected members of the Tree package. You may write your own implementation of the interface that contains various private or protected methods, so long

as they are only called internally in the tree package, and not in the FileSystem package. Again, you are required to satisfy the specifications for the public methods listed in the .java file with regards to running time and behavior.

Your second module is the FileSystem package which contains the FileSystem class. We will run your program using 'java FileSystem', so this class should contain a main method. We have provided a simple skeleton which is able to capture the user input through the use of the InputSource class. You are not required to use what is provided, but it will simplify your task by not having to deal with collecting user input. This class must be able to handle all the commands (listed below) and return accurate and useful error messages without crashing.
Part of the grade for your solution will be for your organization of the additions: reasonable task vs. method correspondence (each of the methods should do a well defined task), appropriate use of inheritance to represent text and directory files, no unduly repetitive code, and suitable use of public, private, and protected information.

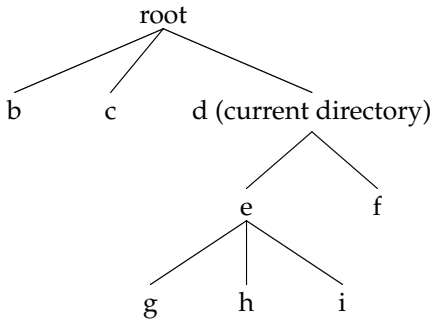
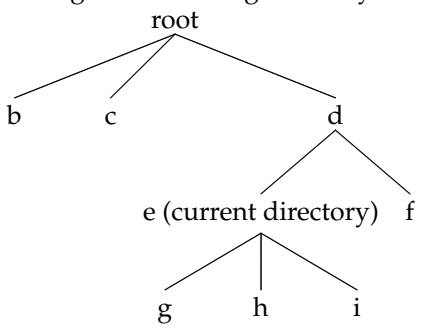
2.2 Commands to Support

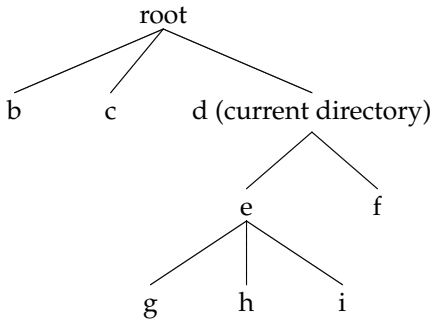
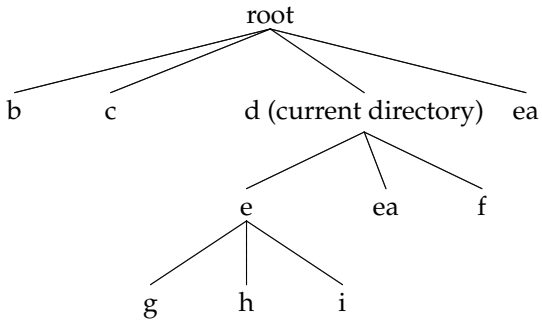
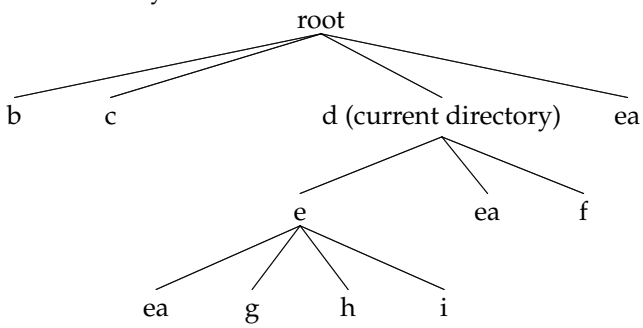
The commands to be supported by the interpreter are described in the table below. They are simpler than their counterparts in the UNIX shell: as noted above, there are only two kinds of files, no command options (normally specified using "–") are allowed, and most commands take only a fixed number of arguments.

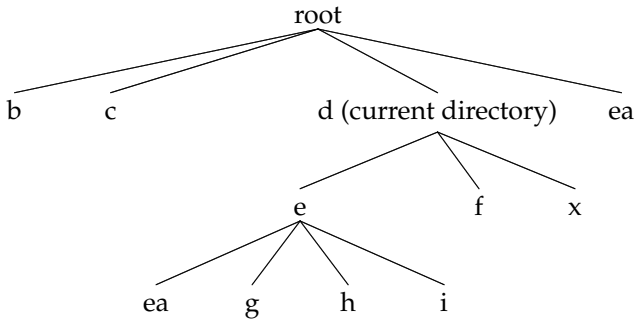
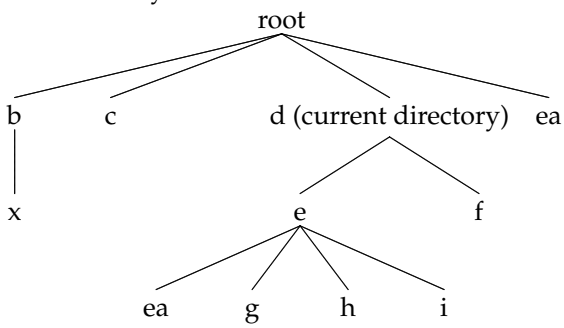
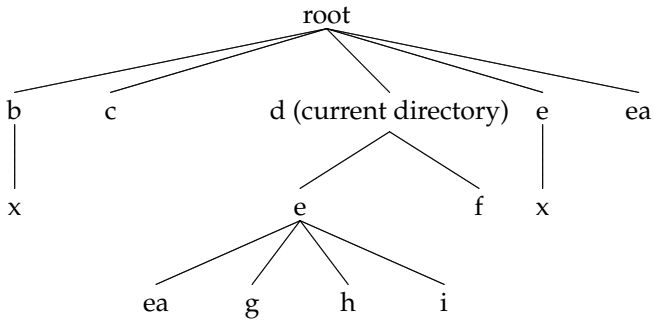
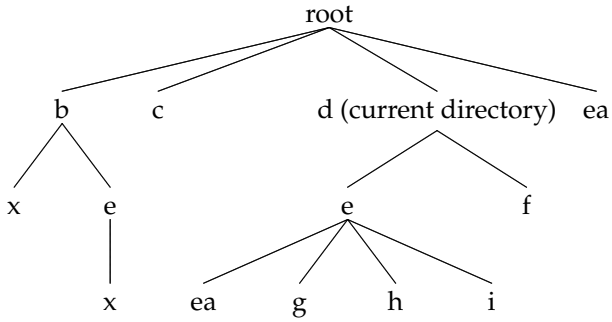
Command	**Number of Arguments**	**Description**
pwd	0	("Print Working Directory") Prints the full path name of the working directory, followed by a RETURN.
cd	1	("Change Directory") Changes the working directory to that named by the argument.
ls	0 or 1	("LiSt files") If given without arguments, prints the names of files and subdirectories in the working directory in alphabetical order. Numbers should come before capital letters, which will come before lower case letters. If given with the name of a text file, prints that name; if given with the name of a directory, prints the names of files in that directory in alphabetical order by name. Each file name should be printed on a line by itself.
lstree	0 or 1	("LiSt TREE") If given without arguments, prints the names of files in the working directory and all its subdirectories in preorder, with files within a directory listed in alphabetical order by name; if given with the name of a directory, prints the names of files in that directory and all its subdirectories in preorder, with files within a directory listed in alphabetical order by name. In either case, each file name should be printed on a line by itself. Alphabetical order refers to the same order as 'ls'.
cat	≥ 1	("conCATenate") Prints the contents of the text files named by the arguments.
mkdir	1	("MaKe DIRectory") Creates a directory with the given name. The name could be a relative or absolute path with the name of the directory as the last "section", e.g. `mkdir /a/b` creates a subdirectory b under the directory a in the root directory. There must not already be a directory with that name.
rmdir	1	("ReMove DIRectory") Removes the named directory from the directory structure. The named directory must be empty. The root directory cannot be removed.

create	1	("CREATE") Creates a "text file" with the given name, then reads the contents of the file from the current input source. Everything up to but not including the first empty line in the input becomes the "contents" of the file. In general, a text file may contain any number of lines (including 0). The named file must not already exist.
rm	1	("ReMove") Removes the named text file from the directory structure. The file must already exist.
cp	2	("CoPy") Creates a copy of the first argument, the source, to the second argument: the destination. If the first argument names a directory, the copy will contain copies of the directory's files and subdirectories. (I.e. it is a deep copy.) There are two legal possibilities for the destination: 1. The destination does not name an existing file. A copy is then made with the new name. That the parent folder must already exist. 2. The destination names a directory. That directory must not contain a child file (directory or text file) with the same name. The copy is made into the named directory, retaining the source's name.
mv	2	("MoVe") Works similarly to cp. There are two legal possibilities for the destination: 1. It does not name an existing file. The file named by the first argument is moved so that it has the destination's name. 2. It names a directory. That directory must not contain a child file (directory or text file) with the same name. The file named by the first argument is moved into the named directory, retaining the source's name. If the the source directory is the same as the present working directory, i.e.,mv command is executed with the source directory being the same as the output of executing pwd, and provided that the command succeeds, the present working directory should be changed to the destination directory.

2.3 Examples

Command	output	Other Effects
ls	e f	Current Tree: 
lstree	e e/g e/h e/i f	none
lstree ..	b c d d/e d/e/g d/e/h d/e/i d/f	none
cd e	none	Changes the working directory to e: 
ls	g h i	none
pwd	/d/e	none

cd ..	none	changes the working directory to d: 
create ea text in file ea more text	none	creates the text file named ea that contains the given two lines of text
cat ea	text in file ea more text	
ls	e ea f	none
cp ea /	none	The directory structure is now 
cp ea e	none	The directory structure is now 

mv ea x	none	The directory structure is now 
mv x /b	none	The directory structure is now 
cp /b /e	none	The directory structure is now 
mv /e /b	none	The directory structure is now 

2.4 Error Handling

Your program should detect all errors in user commands and print the result of calling the message method of the ErrorMessage class (we supply this in the cs61bl/labcode/proj2 directory or online at http://inst.eecs.berkeley.edu/~cs61b/su09/code/proj2/) in response. The possible errors are listed

below.

- Unrecognized command: the first word on the command line isn't one of the commands to be interpreted. (Note: a empty command line isn't an error; it should be ignored.)
- Wrong number [of] arguments.
- Illegal file name: an attempt is being made to create a fileusing create, mkdir, cp, or mvwhose name contains other than alphanumeric characters.
- File not found: an argument, in any command other than pwd, names a file that doesn't exist. (In a path name argument, the nonexistent file could be one of the directories on the path; for this message to be generated by a create or mkdir command, this would have to be the case.)
- File not a directory: an argument to cd, lstree, or rmdir is a text file rather than a directory, or a component of a relative or absolute path name argument to any command but pwd is a text file when it should be a directory. Example: if you had a directory in the root called d that contained a text file named t, and you called mkdir /d/t/new
- File not a text file: an argument to cat or rm is a directory rather than a text file.
- File exists: an attempt is being made in a create, mkdir, cp, or mv command to create a file that already exists. Files are case senstitive. Thus the file named "abc" is different from the file "ABC".
- Current directory: an attempt is being made in a rmdir command to remove the present working directory.
- Nonempty directory: an attempt is being made to remove a nonempty directory with rmdir.
- Identical files: the arguments to cp or mv refer to the same file.
- Destination within source: an attempt is being made to move (mv) or copy (cp) a directory into one of its subdirectories.

Any given user command may be incorrect for more than one of the errors described above. Your shell program should stop parsing at the first error encountered and report the appropriate error with one of the command-line arguments (if one is supplied). Here are few more details on resolving priorities between equally applicable error messages:

- Given a directory structure with an empty root directory, the command mkdir /a/b/c 1/2/3 contains multiple errors: (a)mkdr is an unrecognized command, (b) it contains an incorrect number of arguments (c) the directories do not exist. In this case, you should output the first error encountered, i.e. unrecognized command, with the first word on the command line.
- The commands mv and cp have multiple error conditions. The errors should be checked and reported in the order they are listed in the spec. Consider mv a/x y for example, if both a/x and y are existing text files, then the command fails because (a) y already exists, and (b) y is not a directory. In this case, you should report the first error condition at which the command fails, i.e., File exists. Furthermore, the first argument to the command a/x should be output with the error message.
- If you attempt to create a file create a/ in an empty directory, then there are two possible interpretations of the error: a) the directory a does not exist, and b) the file name "empty space" is an illegal name. In this case you should report FILE_NOT_FOUND to indicate case (a) as the source of error. If, however, a exists, then you should output to indicate that (b) is the source.

2.5 Testing

We have provided you with a framework for running your project as part of a JUnit test, and allows you to specify input and output files. See FileSystemTest.java for an example as to how it's used. In short, you write an input file that has a list of commands that are sent to the program, then you write what you expect to be printed out. The methods provided to you will compare the output of your program with the expected output file.
Also provide a writeup, in a file named testing.readme, that describes the following:

- the rationale for each test, what bugs each test would direct you to if it failed, and why your tests provide sufficient evidence of the absence of bugs in your program.
- what bugs you encountered during the testing process.

In the writeup, you should be very specific in describing what cases you tested and how your inputs test those cases. You should avoid writing something like "I tested boundary cases for the 'mv' command'". Instead, you should say "I tried the 'mv' command moving a file out of a 1-element directory (boundary case), and moving the first and last files in the directory (more boundary cases)." Remember it is your job to convince us that you have thought carefully about what to test and how to test it in your writeup. However, the idea is not to aim for quantity – each test case should target something slightly different from the previous test case and improve your confidence that the program is working under all scenarios. You can lose marks for being ambiguous in your writeup.

Refer to your Pragmatic Unit Testing book on the topics of completeness and organization of tests in your writeup. We strongly encourage you to apply the techniques of test-driven development to build your program.

3 What We Provide

This section is not meant to be a reference or comprehensive in any way. The files should have a much better and more detailed description of what each class does. This is merely an overview, so you can see how things fit together.

3.1 FileSystem.java

This package and class handles the input from the user, and directs the TreeStructure to do things in order for the various commands to function. We have provided enough to take in input from the command line, although you should understand how it works. Of note are the methods exit() and main(). Be sure to understand how these work.

3.2 FileSystemTest.java

This is a JUnit test for your entire project. Look at the provided test case, and notice how it takes in an input and an output file and runs your program with the input to compare the output of your program with the output file.

3.3 ErrorMessage.java

Do not modify this file, however, use this file to generate the various error messages you will be using.

3.4 InputSource.java

You shouldn't find the need to modify this file, but you certainly can. This file uses a BufferedReader to read commands from either the command line or an input file.

3.5 ExitException.java

This file throws an exception that exits the program. This is useful for debugging, since this exception can be caught by the debug suite, and thus proceed onwards to the next test. That would not be possible with System.exit().

3.6 TreeStructure.java

You must implement all public methods exactly as specified. Anything not specified is likely to be a design decision for you to make, but if you have any confusion, please ask your TA or post on newsgroup. You may not add any additional public methods. See the comments in the .java file for an update as to what they must do, as well as online in the FAQ of the google group.

3.7 TreeNode.java

This is something that is used for the TreeStructure to be able to identify nodes. You may write a class that implements this interface and pass references to them around, or you may not. The key is that these may not contain any public variables or public methods.

4 Grading and Submission Details

This project will earn up to 100 points, allocated 90 for the program and 10 for the writeup. These points will be scaled to 6% of your total grade. Grading will proceed as follows.
Your program will be compiled using the command javac FileSystem.java
If it fails to compile, you get no more program points. Although you will normally run your program in Eclipse, please make sure that it runs correctly from a unix command line.

- 30 points will be assigned to correctly interpreting the user commands for non-erroneous inputs.
- 20 points will be assigned to handling the all the errors that can occur.
- 10 points will be assigned to resolving all of the ways that a legal file can be referred, i.e., absolute, relative pathnames.
- 15 points will be assigned to following the interface
- 5 points will be assigned to following the style guidelines in section 5. These guidelines are the same as the first project.
- 10 points will be assigned to writing your test code
- 10 points will be assigned to detailing and explaining your test cases.

You will submit all your `.java` files and ensure that they are in their proper packages/directories. Your testing.readme file should be in the parent directory to both of these files. If you've reached this point, you should know how to move files around so that is the case! From the parent directory, submit your program with `submit proj2`. Congratulations, you're done!

5 Style Guide

We will deduct points for code that does not match the following style, documentation and encapsulation guidelines. (This guide is based upon the style guide written by Professor Jonathan Shewchuk's and used in the spring 2009 version of CS61B.)

1. Each method must be preceded by a comment describing its behavior unambiguously. These comments must include descriptions of what each parameter is for, and what the method returns (if anything). They must also include a description of what the method does (though not necessarily how it does it) detailed enough that somebody else could implement a method that does the same thing from scratch using only the provided documentation. See the comments in the framework code provided for an example of what we mean.
2. All classes, fields, and methods must have the proper `public/private/protected` qualifier. We will deduct points if you make things `public` that could conceivably allow a user to corrupt the data structure.
3. Classes that contain extraneous debugging code, print statements, or meaningless comments that make the code hard to read will be penalized.
4. Your file should be indented to clearly show the structure of nested statements like loops and `if` statements. Sloppy indentation will be penalized. Eclipse will automatically indent your code. Highlight your code and press `Ctrl+I` or `Ctrl+Shift+F`.
5. All `if`, `else`, `while`, `do`, and `for` statements should use braces, even if they are not syntactically required.
6. All classes start with a capital letter, all methods and (non-final) data fields start with a lower case letter, and in both cases, each new word within the name starts with a capital letter. Constants (final fields) are all capital letters only.
7. Numerical constants with special meaning should always be represented by all-caps `final static` constants.
8. All class, method, field, and variable names should be meaningful to a human reader.
9. Methods should not exceed about 50 lines. Any method that long can probably be broken up into logical pieces. The same is probably true for any method that needs more than 7 levels of indentation.
10. Avoid unnecessary duplicated code; if you use the same (or very similar) fifteen lines of code in two different places, those lines should probably be a separate method call.
11. Programs should be easy to read.
12. Keep lines of reasonable length (say, 72 characters, with a maximum of 80 characters) for readability.

6 Checkoff

Due: By the beginning of lab on **Thursday, July 17th**.
Please bring a write-up with the listed information along with the test-cases described below. The write-up is an opportunity to get feedback on your design from your TA. You are welcomed to get checked-off prior to the deadline of Thursday July 17th.

1. A list of the classes you are going to use in your project.
   (a) For each class, list the private methods will you implement.

(b) For each class, list the protected methods will you implement.

(c) For each class, list the instance variables will you add. (The names for all of these should be relatively self-explanatory. If it is not, please provide a written description.)

At least 5 test cases, testing any of the public methods. These test cases do not need to pass. In fact if youre doing test driven development (and havent finished the project) these tests should fail.

7 Frequently Asked Questions

This section will be updated as more questions appear.
Please check the Stickied FAQ on the google group!

Q. What's the point of the TreeNode interface if it has no methods?

A. We wanted to be able to give you as much flexibility as we possibly could. The idea is that TreeStructure needs to reference a location in the tree (using a TreeNode), but we didn't want to specify the relationship between these classes any further. Thus, regardless of whether you have TreeNode objects that are like nodes, or whether your TreeStructure implements TreeNode (not recommended), or something else entirely, you had the flexibility to do so. If we had made it an abstract class, that would be the only thing you could extend from, and we didn't want that.

8 Acknowledgments

This project was modified from code written by MICHAEL CLANCY and PAUL HILFINGER.

9 Changelog

1. 7/19 'lstree ..' should not print e.
2. 7/17 Fixed a typo in the tree. Modified the alphabetical order of the ls method to print number, then upper case, then lower case, in alphabetical order.
3. 7/16 The FAQ is 'official' spec.
4. 7/15 Modified the last image in the examples section. It originally displayed the last command as a copy instead of a move.
5. 7/15 Fixed various punctuation details. Fixed URL.