

Problem Set 8

Due: Tuesday, December 10

Notes:

- Although this assignment is officially due on Tuesday, Dec. 10 (the last day of classes) it will be accepted without penalty through Thursday, Dec. 12.
- This problem set is worth 120 points rather than the usual 100 points.

Reading:

You are expected to read and understand all of the following handouts, even those parts not explicitly covered in class:

- #19 Recurrences
- #20 Asymptotics
- #24 Heaps
- #27 Rose Trees
- #28 2-3 Trees

Overview: In this problem set, you will get experience with recurrence equations, running times, and asymptotic notations; with sorting; with rose trees; and with heaps and 2-3 trees. Most of the problems on this assignment are pencil-and-paper problems; only Problem 6 requires any programming.

Download: To do Problem 6, you should download a copy of the directory `~cs230/download/ps8`.

Submission:

- For Problems 1–5, your hardcopy submission should include answers to the pencil and paper problems. There are no softcopy submissions for these problems.
- For Problem 6, your hardcopy submission should include your final version of `Indexer.java` and transcripts of your test cases. Your softcopy submission for this problem should be your entire `ps8` directory.

Remember to include a signed cover sheet (found at the end of this problem set description) at the beginning of your hardcopy submission.

Problem 1 [10]: Asymptotic Notation

Consider the following six functions:

- $a(n) = 2n \cdot \log_2(n) + n + 5$
- $b(n) = 3n^2 + 7n + 4$
- $c(n) = 17$
- $d(n) = 2^n + 1$
- $e(n) = \log_3(n)$
- $f(n) = 3n - 2$

For each of the following five sets, indicate which of the above functions are members of the set:

1. $O(n)$
2. $o(n^2)$
3. $\Omega(\log_2(n))$
4. $\Theta(n^2 + 5n + 2)$
5. $\omega(3^n)$

Problem 2 [20]: Running Time Analysis

In Fig. 1 are six class methods, all of which compute the value of 2 raised to the n th power. For each function, give the following:

1. A recurrence equation that approximates the worst-case running time $T_{\text{two}_i}(n)$ of the method `two_i` of the input n . In all cases, you may assume that $T(n) = 0$ for $n < 1$ and that the overhead of the divide and glue operations within the body of `two_i` has cost that is a constant (for convenience, assume that the constant is 1).
2. A solution to the recurrence equation expressed in Θ notation.

You may find the following facts helpful in some of your analyses, especially for `two6`.

- $\log_2(a) + \log_2(b) = \log_2(a \cdot b)$
- $\log_2(a) - \log_2(b) = \log_2(a/b)$
- $n! = n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1$
- $\log_2(n!)$ is an element of $\Theta(n \cdot \log_2(n))$
- if f is an element of $o(g)$ then $\Theta(f + g) = \Theta(g)$

```

public static int two1 (int n) {
    if (n == 0) {
        return 1;
    } else {
        return 2 * two1(n - 1);
    }
}

public static int two2 (int n) {
    if (n == 0) {
        return 1;
    } else {
        return two2(n - 1) + two2(n - 1);
    }
}

public static int two3 (int n) {
    if (n == 0) {
        return 1;
    } else {
        return two3(n - 1) + two1(n - 1); // Yes, the second call is to two1, not two3.
    }
}

public static int two4 (int n) {
    // To simplify analysis, assume that the input to two4 is a power of 2.
    if (n == 0) {
        return 1;
    } else if (n % 2 == 0) {
        int x = two4(n / 2);
        return x * x;
    } else {
        return 2 * two4(n - 1);
    }
}

public static int two5 (int n) {
    // To simplify analysis, assume that the input to two5 is a power of 2.
    if (n == 0) {
        return 1;
    } else if (n % 2 == 0) {
        two5(n / 2) * two5(n / 2)
    } else {
        return 2 * two5(n - 1);
    }
}

public static int two6 (int n) {
    if (n == 0) {
        return 1;
    } else {
        return two6(n - 1) + two4(n - 1) // Yes, the second call is to two4, not two6.
    }
}

```

Figure 1: Six different method for raising 2 to the n th power.

Problem 3 [15]: Identifying Sorting Algorithms

When working for AsSorted Systems as a summer intern, Bud Lojack (correctly) implemented four sorting algorithms for lists: selection sort, insertion sort, merge sort, and quick sort. But instead of giving meaningful names to his routines, Bud named them `sort1`, `sort2`, `sort3`, and `sort4`. When writing up a report at the end of the summer, Bud needs to figure out which routine corresponds to which algorithm. Unfortunately, Bud has accidentally deleted the source files for his routines and can't even inspect the code to determine which routine implements which algorithm. The only the thing he can do is test the routines on various inputs. Fig. 2 presents his timing results for running the routines on input lists of varous sizes. (All times are reported in milliseconds.) For each routine, he has tested the routine on both already sorted lists and on randomly ordered lists.

Time for sort1 to sort sorted list with 400 elements:	4
Time for sort1 to sort sorted list with 800 elements:	8
Time for sort1 to sort sorted list with 1600 elements:	16
Time for sort1 to sort sorted list with 3200 elements:	26
Time for sort1 to sort random list with 400 elements:	255
Time for sort1 to sort random list with 800 elements:	958
Time for sort1 to sort random list with 1600 elements:	4059
Time for sort1 to sort random list with 3200 elements:	16585
Time for sort2 to sort sorted list with 400 elements:	92
Time for sort2 to sort sorted list with 800 elements:	339
Time for sort2 to sort sorted list with 1600 elements:	1356
Time for sort2 to sort sorted list with 3200 elements:	5321
Time for sort2 to sort random list with 400 elements:	13
Time for sort2 to sort random list with 800 elements:	37
Time for sort2 to sort random list with 1600 elements:	71
Time for sort2 to sort random list with 3200 elements:	143
Time for sort3 to sort sorted list with 400 elements:	229
Time for sort3 to sort sorted list with 800 elements:	907
Time for sort3 to sort sorted list with 1600 elements:	3311
Time for sort3 to sort sorted list with 3200 elements:	13634
Time for sort3 to sort random list with 400 elements:	205
Time for sort3 to sort random list with 800 elements:	890
Time for sort3 to sort random list with 1600 elements:	3318
Time for sort3 to sort random list with 3200 elements:	13689
Time for sort4 to sort sorted list with 400 elements:	23
Time for sort4 to sort sorted list with 800 elements:	37
Time for sort4 to sort sorted list with 1600 elements:	87
Time for sort4 to sort sorted list with 3200 elements:	178
Time for sort4 to sort random list with 400 elements:	24
Time for sort4 to sort random list with 800 elements:	51
Time for sort4 to sort random list with 1600 elements:	102
Time for sort4 to sort random list with 3200 elements:	213

Figure 2: Timing results for Bud's sorting tests.

a. [8] Based on Bud's data, fill in the following table with the asymptotic notation that best describes the running time of the routine on a particular type of list. You should use one of the following for every entry of the table: $\Theta(1)$, $\Theta(\log(n))$, $\Theta(n)$, $\Theta(n \cdot \log(n))$, $\Theta(n^2)$, and $\Theta(n^3)$. Note: because the numbers are taken from an actual implementation, they may not fit any category exactly; if the category is ambiguous, say so!

Program	Sorted List	Random List
sort1		
sort2		
sort3		
sort4		

b. [4] Based on the table from part a, determine for each of Bud's four routines which of the four sorting algorithms it uses. Briefly explain your reasoning.

c. [3] Answer the following for the four sorting algorithms that Bud has implemented:

- Which sorting algorithm(s) does much better on sorted lists than on randomly ordered lists. Why?
- Which sorting algorithm(s) does much worse on sorted lists than on randomly ordered lists. Why?
- Which sorting algorithm(s) takes about the same amount of time on both sorted and randomly ordered lists. Why?

Problem 4 [25]: A Heap o' Heaps

In the following parts, you are asked to draw some complete *min* heaps (not max heaps) and leftist *min* heaps (not max heaps). As a simple check to avoid errors, verify that every binary tree you draw is a legal complete min heap or leftist min heap.

a. [5] Draw as trees the sequence of *complete min heaps* C_0, C_1, \dots, C_{17} that results from inserting the following letters into the empty heap C_0 :

T H E Q U I C K B R O W N Y A M S

For example, C_1 should be the result of inserting T into the empty heap C_0 ; C_2 should be the result of inserting H into C_1 ; and so on. Assume that letters appearing earlier in the alphabet are "less than" those appearing later. For example, the root node of C_{17} should be A. You need only show the final tree that results from each insertion; you need not show the individual "bubble up" steps that lead to the final tree.

b. [2] One advantage of complete min heaps is that they can be represented as arrays. Show how the complete min heap C_{17} would be represented as an array.

c. [4] Draw as trees the sequence of *complete min heaps* $C_{18}, C_{19}, C_{20}, C_{21}$ that result from dequeuing the *four* smallest items of C_{17} . You need only show the final tree that results from each deletion; you need not show the individual "bubble down" steps that lead to the final tree.

d. [4] Draw as trees the sequence of *leftist min heaps* $L_{10}, L_{11}, \dots, L_{18}$ that results from inserting the following letters into the empty heap L_{10} :

T H E Q U I C K

You need not show the intermediate results of individual merge steps.

e. [4] Draw as trees the sequence of *leftist min heaps* L_0, L_1, \dots, L_9 that results from inserting the following letters into the empty heap L_0 :

B R O W N Y A M S

You need not show the intermediate results of individual merge steps.

f. [2] Draw as a tree the *leftist min heap* L_0 that results from merging the two heaps L_8 and L_9 .

g. [4] Draw as trees the sequence of *leftist min heaps* L_1, L_2, L_3, L_4 that results from dequeuing the *four* smallest items of L_0 . You need only show the final tree that results from each deletion; you need not show the individual merge steps that lead to the final tree.

Problem 5 [15]: 2-3 Trees

In the following parts, you are asked to draw some 2-3 trees. As a simple check to avoid errors, verify that every tree you draw is a legal 2-3 tree.

a. [7] Draw the sequence of *2-3 trees* T_0, T_1, \dots, T_{17} that results from inserting the following letters into the empty tree T_0 :

T H E Q U I C K B R O W N Y A M S

You need only show the final tree that results from each insertion; you need not show the individual steps that lead to each intermediate tree.

b. [8] Draw the sequence of *2-3 trees* $T_{17}, T_{18}, \dots, T_{34}$

that results from deleting the following letters one-by-one from the initial tree T_{17} :

T H E Q U I C K B R O W N Y A M S

You need only show the final tree that results from each deletion; you need not show the individual steps that lead to each intermediate tree. When deleting a value from a non-terminal node, you may replace it with either its in-order predecessor or in-order successor, whichever is easier.

Problem 6 [40]: File Indexing

Background

Modern search engines like Google have sophisticated file indexing mechanisms that make it possible to quickly list web pages containing a given set of keywords. Abstractly, there are two distinct phases to file indexing:

1. An *indexing* phase in which a potentially very large database (think table) is constructed by processing all the files to be indexed.
2. A *query* phase in which users submit keyword queries and all file names in the database matching these queries are found and displayed.

In this problem, you will implement an extremely simple file indexing program for indexing all the files in a specified Linux file system tree. In this case, the database will simply be a table that maps each word that appears in at least one of the files to a set of the absolute filenames¹ of all files in which the word appears. Once the table is constructed, you will be able to query the table for all indexed files containing all keywords in a given list.

A skeleton of a program implementing this simple file indexer is presented in Figs. 3–5. Given an absolute filename, the `indexAndQuery` method first builds an indexing table via `indexFiles` and then enters a *read/query/print loop* in which a list of words entered by the user (and read via `getQuery`) is processed by `processQuery`, which prints out a list of all indexed files that contain all of the given words.

In this problem, you will flesh out the missing details in `Indexer.java`. To begin this problem, you should download the directory `~cs230/download/ps8`, which contains all the code you will need for this problem. In addition to containing the `Indexer.java` file, this directory also contains all the `Collection` classes we have studied and an implementation of rose trees.

Don't be scared by the length of the problem descriptions below. The amount of code that you have to write for each part is rather small.

Tasks

a. [10]: paths

One subproblem of the file indexing problem is to determine all the files in a file system that are in the subtree rooted at a given absolute pathname. We can decompose this into two smaller subproblems:

1. Creating a tree structure in Java that models the tree structure of the file system. We have seen in class that the directory structure of file systems is nicely modeled by rose trees, where directories are represented by `rnodes`, and files are represented by `leaves`.
2. Processing the rose tree to yield a list of absolute filenames for all files in tree.

In this part, you will write a `paths` method that solves the second of these two subproblems. In the next part, you will write a `fileToRoseTree` method that solves the first of these two subproblems.

The `paths` method has the following specification:

```
public static ObjectList paths (ObjectRoseTree fileTree);
```

Given a rose tree `fileTree` that represents a file system tree, returns an `ObjectList` of strings that are the pathnames of all files in the tree. The resulting list should only include pathnames for files, not directories. The file names should be listed from left to right as they appear in the tree.

¹A filename is *absolute* if it begins with a slash, which represents to the root of the file system, as in `/home/cs230/download`. In contrast, a *relative* filename is one that does not begin with a slash, as in `ps8/Indexer.java`; the meaning of such a name is relative to the directory to which the user is currently connected.

```

import java.io.*;
import java.util.*;

public class Indexer {

    // ----- index & query -----

    // Index the file subsystem rooted at pathname and then
    // enter a read/eval/print loop to query the index
    public static void indexAndQuery (String pathname) {

        // Phase 1: first create a table that indexes all the files rooted at pathname
        System.out.println("Creating an index table for files rooted at " + pathname);
        System.out.println("This may take a while.");
        Table indexTable = indexFiles(pathname);
        System.out.println("Index table successfully constructed.");

        // Phase 2: now read queries from user until the an empty line is reached
        ObjectList words = getQuery();
        while (! OL.isEmpty(words)) {
            dashes();
            System.out.println("Files containing all the keywords " + words + ":");

            processQuery(words, indexTable);
            words = getQuery();
        }

        // Exit the read/query/print loop
        stars();
        System.out.println("Thank you for using the indexing query system!");
    }

    public static Table indexFiles (String pathname) {
        // Flesh this out.
    }

    public static void processQuery (ObjectList words, Table tbl) {
        // Flesh out this method.
    }

    public static ObjectList getQuery () {
        stars();
        System.out.println("Please enter a sequence of keywords: ");
        Enumeration enum = new StringWords(readLine());
        ObjectList list = OL.empty();
        while (enum.hasMoreElements()) {
            list = OL.prepend(((String) enum.nextElement()).toLowerCase(), list);
        }
        return list;
    }

    public static void dashes () {
        System.out.println("-----");
    }

    public static void stars () {
        System.out.println("*****");
    }
}

```

Figure 3: Implementation of Indexer.java, part 1.

```

// ----- paths -----

// Returns an ObjectList containing all the absolute pathnames
// representing by the rose tree fileTree.
public static ObjectList paths (ObjectRoseTree fileTree) {
    // replace this stub:
    return OL.empty();
}

// Returns an ObjectList containing all the absolute pathnames
// representing by the rose tree list fileTrees.
public static ObjectList pathsList (ObjectRoseTreeList fileTrees) {
    // replace this stub:
    return OL.empty();
}

// ----- path to Rose Tree -----

// Returns the ObjectRoseTree representing the file structure rooted
// at the directory or file named by the absolute path pathname.
// If there is no file with the absolute path pathname, throws an exception.
public static ObjectRoseTree pathToRoseTree (String pathname) {
    File f = new File(pathname);
    if (! f.exists()) {
        throw new RuntimeException("Indexer.pathToRoseTree: file does not exist -- "
            + pathname);
    } else {
        ObjectRoseTree rt = fileToRoseTree(f);
        // Use absolute pathname for root
        if (ORT.isRLeaf(rt)) {
            return ORT.rleaf(pathname);
        } else {
            return ORT.rnode(pathname, ORT.subtrees(rt));
        }
    }
}

// Returns the ObjectRoseTree representing the file structure rooted
// at the directory or file named by the abstract file f.
// Assumes all files are either directories or regular files.
public static ObjectRoseTree fileToRoseTree (File f) {
    // replace this stub:
    return ORT.rempy();
}

```

Figure 4: Implementation of Indexer.java, part 2.

```

// ----- readLine() -----

private static int EOF = -1; // End of file integer.
private static int EOL = 10; // End of line integer.

private static String readLine() {
    StringBuffer sb = new StringBuffer();
    try {
        int i = System.in.read();
        while (i != EOL) {
            if (i == EOF) {
                return sb.toString(); // Shouldn't happen
            } else {
                sb.append((char) i);
                i = System.in.read();
            }
        }
        return sb.toString();
    } catch (IOException e) {
        System.err.println("Shouldn't happen: Error occurred during readLine()");
        return "";
    }
}

// ----- testing -----

public static void main (String [] args) {
    if (args[0].equals("paths")) {
        if (args.length == 1) {
            OL.prettyPrint1(paths(ORT.fromFile("files.txt")));
        } else {
            OL.prettyPrint1(paths(pathToRoseTree(args[1])));
        }
    } else if (args[0].equals("pathToRoseTree")) {
        ORT.prettyPrint(pathToRoseTree(args[1]));
    } else if (args[0].equals("indexFiles")) {
        Table t = indexFiles(args[1]);
        dashes();
        Enumeration keys = t.keys();
        while (keys.hasMoreElements()) {
            String key = (String) keys.nextElement();
            System.out.println(key + ": " + ((Set) t.lookup(key)).toList());
        }
        dashes();
    } else if (args[0].equals("indexAndQuery")) {
        indexAndQuery(args[1]);
    } else {
        throw new RuntimeException ("Unknown main option " + args[0]);
    }
}

//----- Local Abbreviations -----

// Hack for abbreviating ObjectList and ObjectListOps as OL
public static ObjectListOps OL;

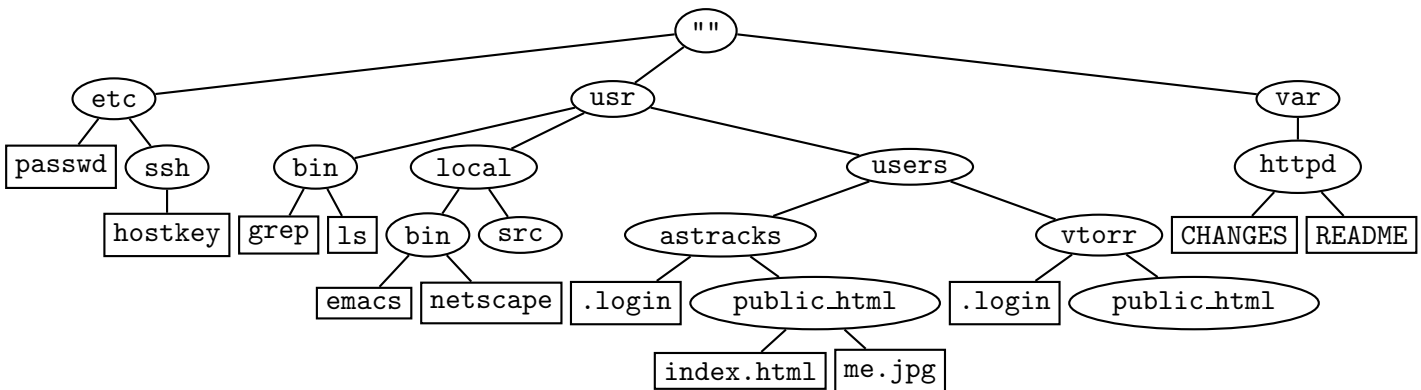
// Hack for abbreviating ObjectRoseTree and ObjectRoseTreeOps as ORT
public static ObjectRoseTreeOps ORT;

// Hack for abbreviating ObjectRoseTreeList and ObjectRoseTreeListOps as ORTL
public static ObjectRoseTreeListOps ORTL;
}

```

Figure 5: Implementation of Indexer.java, part 3.

As an example, consider the following filesystem, where the "" is an empty string that represents the root of the file system.

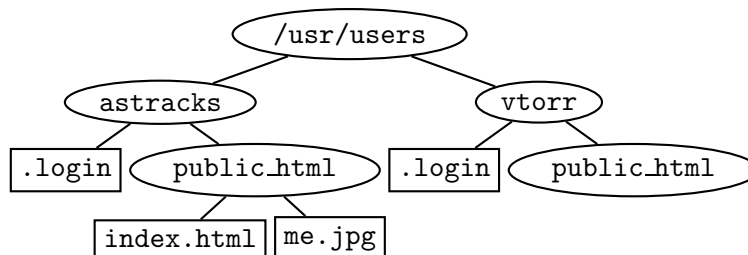


The result of invoking `paths` on this tree should be the following list:

```
[/etc/passwd,
 /etc/ssh/hostkey,
 /usr/bin/grep,
 /usr/bin/ls,
 /usr/local/bin/emacs,
 /usr/local/bin/netscape,
 /usr/users/astracks/.login,
 /usr/users/astracks/public_html/index.html,
 /usr/users/astracks/public_html/me.jpg,
 /usr/users/vtorr/.login,
 /var/httpd/CHANGES,
 /var/httpd/README
]
```

Note how `paths` inserts a slash (/) after the name of every directory.

The `paths` method can produce both absolute and relative filenames, depending on whether the root directory name begins with a slash (or is empty). The above example shows absolute filenames. Here is another example of a tree with absolute filenames:



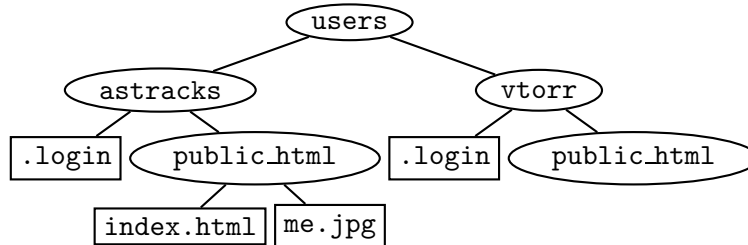
For this tree, `paths` should yield the following list:

```

[/usr/users/astracks/.login,
 /usr/users/astracks/public_html/index.html,
 /usr/users/astracks/public_html/me.jpg,
 /usr/users/vtorr/.login,
 ]

```

Here is an example of a tree with relative filenames:



For this tree, `paths` should yield the following list:

```

[users/astracks/.login,
 users/astracks/public_html/index.html,
 users/astracks/public_html/me.jpg,
 users/vtorr/.login,
 ]

```

Your job in this part is to flesh out the definition of `paths`. As in other rose tree manipulation problems we have studied, you will need to implement a method that works on `ObjectRoseTreeList` (`pathsList`) in addition to the method that works on an `ObjectRoseTree` (`paths`). You should also implement any auxiliary methods that you find useful.

Note that `Indexer.java` has been configured so that the prefix `ORT.` can be used to access any `ObjectRoseTree` method and `ORTL.` can be used to access any `ObjectRoseTreeList` method.

You can test your method by executing `java Indexer paths`. This runs `paths` on the rose tree corresponding to the large tree structure depicted above, which is read in from the parenthesized representation in the file `files.txt`. (Another means of testing `paths` is discussed at the end of the next part.)

b. [10]: `fileToRoseTree`

In this part, you will generate the rose tree that models the file subsystem rooted at a particular absolute pathname. This is accomplished by the `pathToRoseTree` method in Fig. 4. This method first converts the concrete pathname, represented as a string, into an instance `f` of the standard Java `File` class that represents an abstract file system file path. It then calls `fileToRoseTree` to turn this abstract file path into a rose tree. Finally, it replaces the root of the returned tree with the full absolute pathname (otherwise, only the last name in the path would be used).

Your task in this part is to complete the definition of `fileToRoseTree`, which constructs the rose tree that models the portion of the Linux file system rooted at the given abstract file path. You will need to use the following instance methods from the standard Java `File` class to navigate the

Linux file system: `isFile`, `isDirectory`, `listFiles`, `getName`. You can find the specifications for these in the online Java API (accessible from the CS230 home page). Note that `pathToRoseTree` also uses the `exists` instance method from the `File` class to determine if the specified pathname corresponds to a real file in the Linux filesystem.

In your `fileToRoseTree` method, you only need to handle directories and “regular” files. There are other kinds of files in Linux (such as symbolic links, which correspond to shortcuts in Windows or aliases on a Mac) that you do not need to handle and can ignore for the purpose of this problem.

The cool thing about `pathToRoseTree` is that it actually navigates the file system on *your* computer. Indeed, using the kinds of `File` methods mentioned above, you could write your own routines for manipulating and displaying information about your files. For instance, you could write your own version of the Linux `ls` and `find` commands from scratch!

You can test `pathToRoseTree` by invoking

```
java Indexer pathToRoseTree "pathname"
```

where *pathname* is an absolute pathname of a directory in your file system. A concrete example of a file directory we will be working with in the remainder of this problem is `/home/cs/test`, which contains a subset of the files from the CS department account, including the HTML files for the CS department home pages. Since these files are world-readable, you (and your programs) can manipulate them. For example,

```
java Indexer pathToRoseTree "/home/cs/test"
```

should display the parenthesized representation of the rose tree shown in Fig. 6.

You can test both `paths` and `pathsToRoseTree` at the same time by invoking

```
java Indexer paths "pathname"
```

This will display the list of filenames returned by `paths(pathsToRoseTree("pathname"))`. For example,

```
java Indexer paths "/home/cs/test"
```

should display the list of absolute filenames shown in Fig. 7.

c. [10]: `indexFiles`

In this part, your task is to implement the following method:

```
public static Table indexFiles (String pathname);
```

Returns a table that has as its keys all of the words appearing in all of the files in the file tree rooted at `pathname`. The table should map each key to a set of the absolute pathnames of all of the files in the file tree in which the word appears. As part of producing the table, this method should also display the name of each file that is processed.

Here are some hints on how to proceed:

- use `paths` and `pathsToRoseTree` to generate a list of the absolute filenames of all files in the file tree rooted at the given `pathname`;
- use `TableMBST` as your table implementation;

```

(/home/cs/test
  (marketing
    hires.tex
  )
  (spring03-hire
    spring03-hiring-blurb.txt
  )
  (tenure-track-hire
    tenure-track-blurb-long.txt
    tenure-track-blurb-short.txt
  )
  (handbook
    curriculum.2002.tex
    handbook.tex
    handbook.tex~
  )
  (public_html
    (About
      facilities.html
      newhistory.html
      history.html
      welcome.html
    )
    (Activities
      cspanels.html
      studentfun.html
      studentsem.html
      calendar.html
    )
    (Curriculum
      choosing.html
      Courselistings.html
      Major.html
      MITcourses.html
      OnlineCourses.html
      whichCS.html
    )
    (People
      address.html
      alumnae.html
      faculty.html
      form.html
    )
    (Research
      honors.html
      independ.html
    )
    (Resources
      gradprog.html
      (handouts
      )
      tutoring.html
      unix.html
      work.html
      (joblistings
        amazon02.html
      )
    )
  )
  (Tenureposition
    tenuretrack.html
  )
  index.html
  spring03hire.html
)

```

Figure 6: Printed representation of the rose tree resulting from the invocation `pathToRoseTree("/home/cs/test")`.

```
[/home/cs/test/marketing/hires.tex,  
/home/cs/test/spring03-hire/spring03-hiring-blurb.txt,  
/home/cs/test/tenure-track-hire/tenure-track-blurb-long.txt,  
/home/cs/test/tenure-track-hire/tenure-track-blurb-short.txt,  
/home/cs/test/handbook/curriculum.2002.tex,  
/home/cs/test/handbook/handbook.tex,  
/home/cs/test/handbook/handbook.tex~,  
/home/cs/test/public_html/About/facilities.html,  
/home/cs/test/public_html/About/newhistory.html,  
/home/cs/test/public_html/About/history.html,  
/home/cs/test/public_html/About/welcome.html,  
/home/cs/test/public_html/Activities/cspanels.html,  
/home/cs/test/public_html/Activities/studentfun.html,  
/home/cs/test/public_html/Activities/studentsem.html,  
/home/cs/test/public_html/Activities/calendar.html,  
/home/cs/test/public_html/Curriculum/choosing.html,  
/home/cs/test/public_html/Curriculum/Courselistings.html,  
/home/cs/test/public_html/Curriculum/Major.html,  
/home/cs/test/public_html/Curriculum/MITcourses.html,  
/home/cs/test/public_html/Curriculum/OnlineCourses.html,  
/home/cs/test/public_html/Curriculum/whichCS.html,  
/home/cs/test/public_html/People/address.html,  
/home/cs/test/public_html/People/alumnae.html,  
/home/cs/test/public_html/People/faculty.html,  
/home/cs/test/public_html/People/form.html,  
/home/cs/test/public_html/Research/honors.html,  
/home/cs/test/public_html/Research/independ.html,  
/home/cs/test/public_html/Resources/gradprog.html,  
/home/cs/test/public_html/Resources/tutoring.html,  
/home/cs/test/public_html/Resources/unix.html,  
/home/cs/test/public_html/Resources/work.html,  
/home/cs/test/public_html/Resources/joblistings/amazon02.html,  
/home/cs/test/public_html/Tenureposition/tenuretrack.html,  
/home/cs/test/public_html/index.html,  
/home/cs/test/public_html/spring03hire.html  
]
```

Figure 7: Printed representation of list of filenames resulting from the invocation `paths("/home/cs/test")`.

- use `SetVectorSorted` as your set implementation (this is not particularly efficient, but will suffice as long as not too many files are indexed);
- use `FileWords` to enumerate the words from each file;
- write any auxiliary methods that you find helpful;
- don't forget to lowercase all strings used as keys;

To test `indexFiles`, execute

```
java Indexer indexFiles "pathname"
```

which invokes `indexFiles` on `pathname` and then displays, one per line, each key from the resulting table along with the list of filenames in its associated set. As an example, Fig. 8 shows (part of) the result displayed by the invocation

```
java Indexer indexFiles "/home/cs/test/public_html/About"
```

This invocation results in indexing four files and printing out a long sequence of key/filename-list pairs, part of which is shown in Fig. 8.

d. [10]: `processQuery`

The final part of the problem is to implement the following method:

```
public static void processQuery (ObjectList words, Table tbl);
```

Given a list of lowercase keyword strings and a file indexing table produced by `indexFiles`, display (one per line) a sequence of all the filenames in which *all* the keywords appear.

For the purposes of program development, you may wish to first focus on only the first element of `words` before handling multiple keywords. The `Set intersection` method is particularly useful for handling multiple keywords.

To test both `indexFiles` and `processQuery`, execute

```
java Indexer indexAndQuery "pathname"
```

which indexes all the files rooted at the absolute pathname `pathname` and then enters a read/query/print loop that prompts the user for a list of keywords and calls `processQuery` on this list and the indexing table.

For example, Figs. 9–10 shows an indexing and query session associated with the invocation `java Indexer indexAndQuery "/home/cs/test"`. Note that some queries (such as the keyword `mit`) yield a long sequence of filenames, while others (such as the keywords `cs111` `tenure-track`) do not yield any filenames. The read/query/print loop is exited by entering a line with zero keywords.

```

Indexing file /home/cs/test/public_html/About/facilities.html
Indexing file /home/cs/test/public_html/About/newhistory.html
Indexing file /home/cs/test/public_html/About/history.html
Indexing file /home/cs/test/public_html/About/welcome.html
-----
0: [/home/cs/test/public_html/About/facilities.html, /home/cs/test/public_html/About/history.html,
    /home/cs/test/public_html/About/newhistory.html, /home/cs/test/public_html/About/welcome.html]
110: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
111: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
1130: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
121b: [/home/cs/test/public_html/About/facilities.html]
13: [/home/cs/test/public_html/About/facilities.html]
15: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
...
a: [/home/cs/test/public_html/About/facilities.html, /home/cs/test/public_html/About/history.html,
    /home/cs/test/public_html/About/newhistory.html, /home/cs/test/public_html/About/welcome.html]
ability: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
abound: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
about: [/home/cs/test/public_html/About/facilities.html, /home/cs/test/public_html/About/welcome.html]
academic: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html,
           /home/cs/test/public_html/About/welcome.html]
access: [/home/cs/test/public_html/About/facilities.html, /home/cs/test/public_html/About/history.html,
         /home/cs/test/public_html/About/newhistory.html]
accounts: [/home/cs/test/public_html/About/facilities.html]
acquire: [/home/cs/test/public_html/About/welcome.html]
across: [/home/cs/test/public_html/About/welcome.html]
activities: [/home/cs/test/public_html/About/facilities.html]
activity: [/home/cs/test/public_html/About/facilities.html]
added: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
...
wish: [/home/cs/test/public_html/About/welcome.html]
with: [/home/cs/test/public_html/About/facilities.html, /home/cs/test/public_html/About/history.html,
       /home/cs/test/public_html/About/newhistory.html, /home/cs/test/public_html/About/welcome.html]
without: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
women's: [/home/cs/test/public_html/About/welcome.html]
woolly: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
word: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
work: [/home/cs/test/public_html/About/facilities.html, /home/cs/test/public_html/About/welcome.html]
work.html: [/home/cs/test/public_html/About/facilities.html]
working: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
workstation: [/home/cs/test/public_html/About/facilities.html, /home/cs/test/public_html/About/history.html,
             /home/cs/test/public_html/About/newhistory.html]
workstations: [/home/cs/test/public_html/About/facilities.html, /home/cs/test/public_html/About/history.html,
              /home/cs/test/public_html/About/newhistory.html]
world: [/home/cs/test/public_html/About/welcome.html]
would: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
written: [/home/cs/test/public_html/About/facilities.html, /home/cs/test/public_html/About/history.html,
         /home/cs/test/public_html/About/newhistory.html]
www.wellesley.edu: [/home/cs/test/public_html/About/facilities.html,
                   /home/cs/test/public_html/About/history.html,
                   /home/cs/test/public_html/About/newhistory.html,
                   /home/cs/test/public_html/About/welcome.html]
year: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html,
      /home/cs/test/public_html/About/welcome.html]
years: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
yet: [/home/cs/test/public_html/About/history.html, /home/cs/test/public_html/About/newhistory.html]
you: [/home/cs/test/public_html/About/welcome.html]
you're: [/home/cs/test/public_html/About/welcome.html]
you've: [/home/cs/test/public_html/About/welcome.html]
-----

```

Figure 8: Partial result displayed by invoking `indexFiles("/home/cs/test/public_html/About")`. The ellipses (...) indicate large chunks of the result that have been omitted. The results have been manually reformatted to make them easier to read.

```

Creating an index table for files rooted at /home/cs/test/
This may take a while.
Indexing file /home/cs/test/marketing/hires.tex
Indexing file /home/cs/test/spring03-hire/spring03-hiring-blurb.txt
Indexing file /home/cs/test/tenure-track-hire/tenure-track-blurb-long.txt
Indexing file /home/cs/test/tenure-track-hire/tenure-track-blurb-short.txt
Indexing file /home/cs/test/handbook/curriculum.2002.tex
Indexing file /home/cs/test/handbook/handbook.tex
Indexing file /home/cs/test/public_html/About/facilities.html
Indexing file /home/cs/test/public_html/About/newhistory.html
Indexing file /home/cs/test/public_html/About/history.html
Indexing file /home/cs/test/public_html/About/welcome.html
Indexing file /home/cs/test/public_html/Activities/cspanels.html
Indexing file /home/cs/test/public_html/Activities/studentfun.html
Indexing file /home/cs/test/public_html/Activities/studentsem.html
Indexing file /home/cs/test/public_html/Activities/calendar.html
Indexing file /home/cs/test/public_html/Curriculum/choosing.html
Indexing file /home/cs/test/public_html/Curriculum/Courselistings.html
Indexing file /home/cs/test/public_html/Curriculum/Major.html
Indexing file /home/cs/test/public_html/Curriculum/MITcourses.html
Indexing file /home/cs/test/public_html/Curriculum/OnlineCourses.html
Indexing file /home/cs/test/public_html/Curriculum/whichCS.html
Indexing file /home/cs/test/public_html/People/address.html
Indexing file /home/cs/test/public_html/People/alumnae.html
Indexing file /home/cs/test/public_html/People/faculty.html
Indexing file /home/cs/test/public_html/People/form.html
Indexing file /home/cs/test/public_html/Research/honors.html
Indexing file /home/cs/test/public_html/Research/independ.html
Indexing file /home/cs/test/public_html/Resources/gradprog.html
Indexing file /home/cs/test/public_html/Resources/tutoring.html
Indexing file /home/cs/test/public_html/Resources/unix.html
Indexing file /home/cs/test/public_html/Resources/work.html
Indexing file /home/cs/test/public_html/Resources/joblistings/amazon02.html
Indexing file /home/cs/test/public_html/Tenureposition/tenuretrack.html
Indexing file /home/cs/test/public_html/index.html
Indexing file /home/cs/test/public_html/spring03hire.html
Index table successfully constructed.
*****
Please enter a sequence of keywords:
mit
-----
Files containing all the keywords [mit]:
/home/cs/test/handbook/curriculum.2002.tex
/home/cs/test/handbook/handbook.tex
/home/cs/test/public_html/About/history.html
/home/cs/test/public_html/About/newhistory.html
/home/cs/test/public_html/About/welcome.html
/home/cs/test/public_html/Activities/cspanels.html
/home/cs/test/public_html/Curriculum/Courselistings.html
/home/cs/test/public_html/Curriculum/MITcourses.html
/home/cs/test/public_html/Curriculum/Major.html
/home/cs/test/public_html/Curriculum/OnlineCourses.html
/home/cs/test/public_html/Curriculum/choosing.html
/home/cs/test/public_html/Curriculum/whichCS.html
/home/cs/test/public_html/People/address.html
/home/cs/test/public_html/People/form.html
/home/cs/test/public_html/Resources/gradprog.html
/home/cs/test/public_html/Resources/work.html
/home/cs/test/public_html/Tenureposition/tenuretrack.html
/home/cs/test/public_html/index.html
/home/cs/test/public_html/spring03hire.html
/home/cs/test/spring03-hire/spring03-hiring-blurb.txt
/home/cs/test/tenure-track-hire/tenure-track-blurb-long.txt
/home/cs/test/tenure-track-hire/tenure-track-blurb-short.txt

```

Figure 9: A sample session of the invocation `java Indexer indexAndQuery "/home/cs/test"`, Part 1.

```

*****
Please enter a sequence of keywords:
tenure-track
-----
Files containing all the keywords [tenure-track]:
/home/cs/test/marketing/hires.tex
/home/cs/test/public_html/Tenureposition/tenuretrack.html
/home/cs/test/tenure-track-hire/tenure-track-blurb-long.txt
/home/cs/test/tenure-track-hire/tenure-track-blurb-short.txt
*****
Please enter a sequence of keywords:
cs111
-----
Files containing all the keywords [cs111]:
/home/cs/test/handbook/handbook.tex
/home/cs/test/public_html/About/welcome.html
/home/cs/test/public_html/Curriculum/Courselistings.html
/home/cs/test/public_html/Curriculum/Major.html
/home/cs/test/public_html/Curriculum/OnlineCourses.html
/home/cs/test/public_html/Curriculum/choosing.html
/home/cs/test/public_html/Curriculum/whichCS.html
*****
Please enter a sequence of keywords:
mit tenure-track
-----
Files containing all the keywords [tenure-track, mit]:
/home/cs/test/public_html/Tenureposition/tenuretrack.html
/home/cs/test/tenure-track-hire/tenure-track-blurb-long.txt
/home/cs/test/tenure-track-hire/tenure-track-blurb-short.txt
*****
Please enter a sequence of keywords:
cs111 tenure-track
-----
Files containing all the keywords [tenure-track, cs111]:
*****
Please enter a sequence of keywords:
mit cs111
-----
Files containing all the keywords [cs111, mit]:
/home/cs/test/handbook/handbook.tex
/home/cs/test/public_html/About/welcome.html
/home/cs/test/public_html/Curriculum/Courselistings.html
/home/cs/test/public_html/Curriculum/Major.html
/home/cs/test/public_html/Curriculum/OnlineCourses.html
/home/cs/test/public_html/Curriculum/choosing.html
/home/cs/test/public_html/Curriculum/whichCS.html
*****
Please enter a sequence of keywords:

*****
Thank you for using the indexing query system!

```

Figure 10: A sample session of the invocation `java Indexer indexAndQuery "/home/cs/test"`, Part 2.

Going Further

As mentioned above, the file indexer you implemented in this problem is extremely simple. For extra credit, you can make it more sophisticated by implementing some of the following extensions (some of which are significantly more challenging than others):

- Many search engines do not index extremely frequent words like **a**, **an**, **the**, **and**, and **or**. Make a list of such words and modify the indexer so that it does not index them.
- Search engines often have a mechanism for *ranking* matches so that documents matching a query are listed in order of rank. For example, the ranking mechanism might give preference to a document where the frequency of one of the keywords is high compare to one where it's low. Modify the indexing and query phases to take the frequency of words into account.
- Some search engines show some of the *contexts* in which a word appears in a document. A context might be a few words to either side of the given word, or the entire line in which the word was found. Modify the indexer so that it keeps the context of the *first* occurrence of a word in the file.
- As currently designed, the indexer is only intended to index text and code files (e.g., files with extensions like `.txt`, `.html`, and `.java`). However, the indexer attempts to index all files, including files for which indexing does not make sense, such as pictures (e.g., files with extensions like `.gif`, `.jpg`, and `.png`) and binary files. By using file extensions and/or heuristics for determining if a file is a text file, modify the indexer to index only text files.
- When processing HTML files, the current indexer will index information in all the tags (such as ``, ``, ``, etc.). Modify the indexer so that information inside HTML tags is not indexed. To do this, you will need to create a modified version of `FileWords` that ignores information delimited by angle brackets.

*Problem Set Header Page
Please make this the first page of your hardcopy submission.*

CS230 Problem Set 8

Due Tuesday, December 10

Name:

Date & Time Submitted:

Collaborators (*anyone you worked with on the problem set*):

By signing below, I attest that I have followed the collaboration policy as specified in the Course Information handout.

Signature:

*In the **Time** column, please estimate the time you spend on the parts of this problem set. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	Time	Score
General Reading		
Problem 1 [10]		
Problem 2 [20]		
Problem 3 [15]		
Problem 4 [25]		
Problem 5 [15]		
Problem 6 [40]		
Total		