COMP20010

Data Structures and Algorithms I

09 - Tutorial: Stacks

Dr. Aonghus Lawlor
aonghus.lawlor@ucd.ie

Stacks

- Implement the Stack ADT using an Array type
- Implement the Stack ADT using a singly linked list
- Comment on the differences in the implementations.

ArrayStack

My ArrayStack implementation- you can refer to this, but please write your own.

```
package comp20010;

public class ArrayStack<E> implements Stack<E> {

   private int t = -1;
   private E[] data;
   private int MAXSIZE = 512;

   public ArrayStack() {
      data = (E[]) new Object[MAXSIZE];
   }

   @Override
   public int size() {
      return t + 1;
   }

   @Override
   public boolean isEmpty() {
      return size() == 0;
   }

   @Override
   public void push(E e) {
      data[++t] = e;
   }

   @Override
   public E top() {
      // TODO: error handling
      return data[t];
   }

   @Override
   public E pop() {
      if(isEmpty()) {
         return null;
      }
      E topElement = data[t];
      //data[t] = null; // prompt for java garbage collection
      --t;
      return topElement;
   }

   public static void main(String[] args) {
   }
}
```

LinkedListStack

My LinkedListStack implementation- you can refer to this, but please write your own.

```
package comp20010;
```

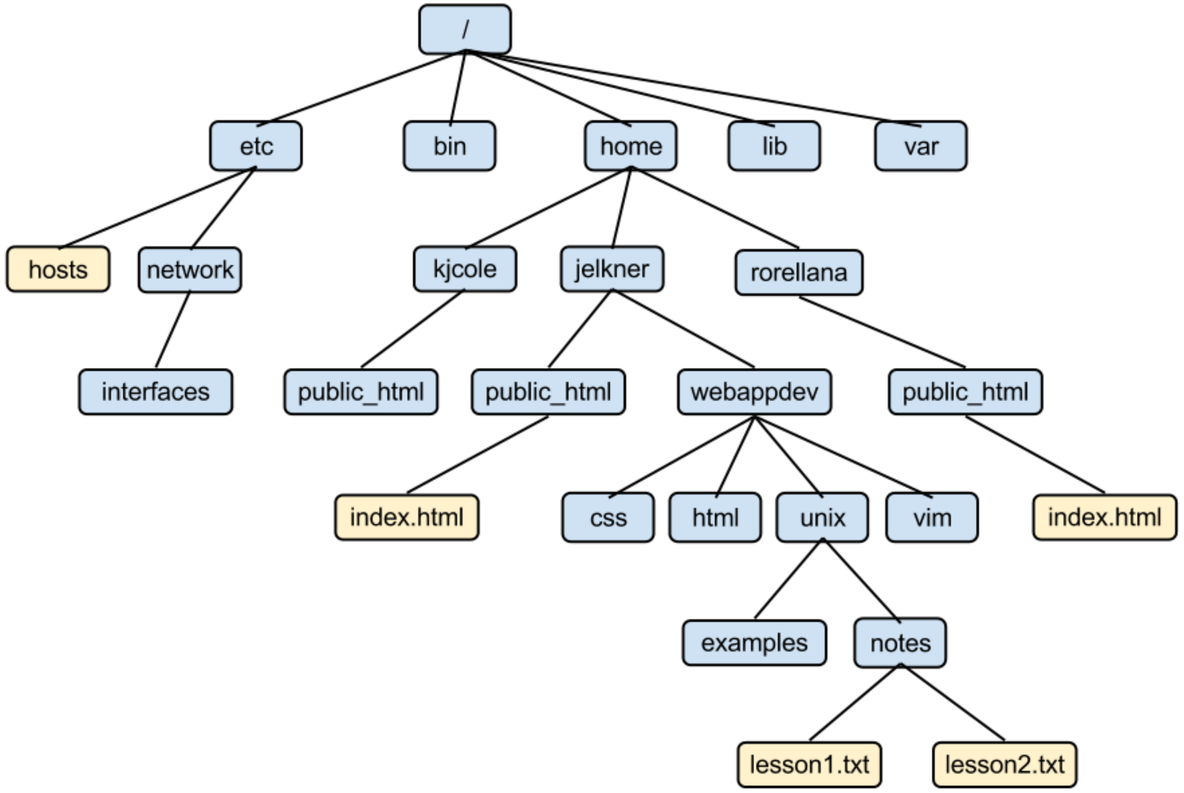
YOUR IMPLEMENTATION HERE!

Stack Testing

```
package comp20010;

public class StackTester {

   public static void testJavaStack() {
      java.util.Stack<Integer> S = new java.util.Stack<>();
      // Stack<Integer> S = new ArrayStack<>();
      S.push(5); // contents: (5)
      S.push(3); // contents: (5, 3)
      System.out.println(S.size() + " = 2"); // contents: (5, 3) outputs 2
      System.out.println(S.pop() + " = 3"); // contents: (5) outputs 3
      System.out.println(S.isEmpty() + " = false"); // contents: (5) outputs false
      System.out.println(S.pop() + " = 5"); // contents: () outputs 5
      System.out.println(S.isEmpty() + " = true"); // contents: () outputs true
      try {
         System.out.println(S.pop() + " = null"); // contents: () outputs null
      } catch(java.util.EmptyStackException e) {
         System.out.println("# Empty Stack");
      }
      S.push(7); // contents: (7)
      S.push(9); // contents: (7, 9)

      // NOTE: we need S.peek() instead of S.top()
      System.out.println(S.peek() + " = 9"); // contents: (7, 9) outputs 9
      S.push(4); // contents: (7, 9, 4)
      System.out.println(S.size() + " = 3"); // contents: (7, 9, 4) outputs 3
      System.out.println(S.pop() + " = 4"); // contents: (7, 9) outputs 4
      S.push(6); // contents: (7, 9, 6)
      S.push(8); // contents: (7, 9, 6, 8)
      System.out.println(S.pop() + " = 8"); // contents: (7, 9, 6) outputs 8
   }

   public static void main(String[] args) {
      //testArrayStack();
      testJavaStack();
   }
}
```

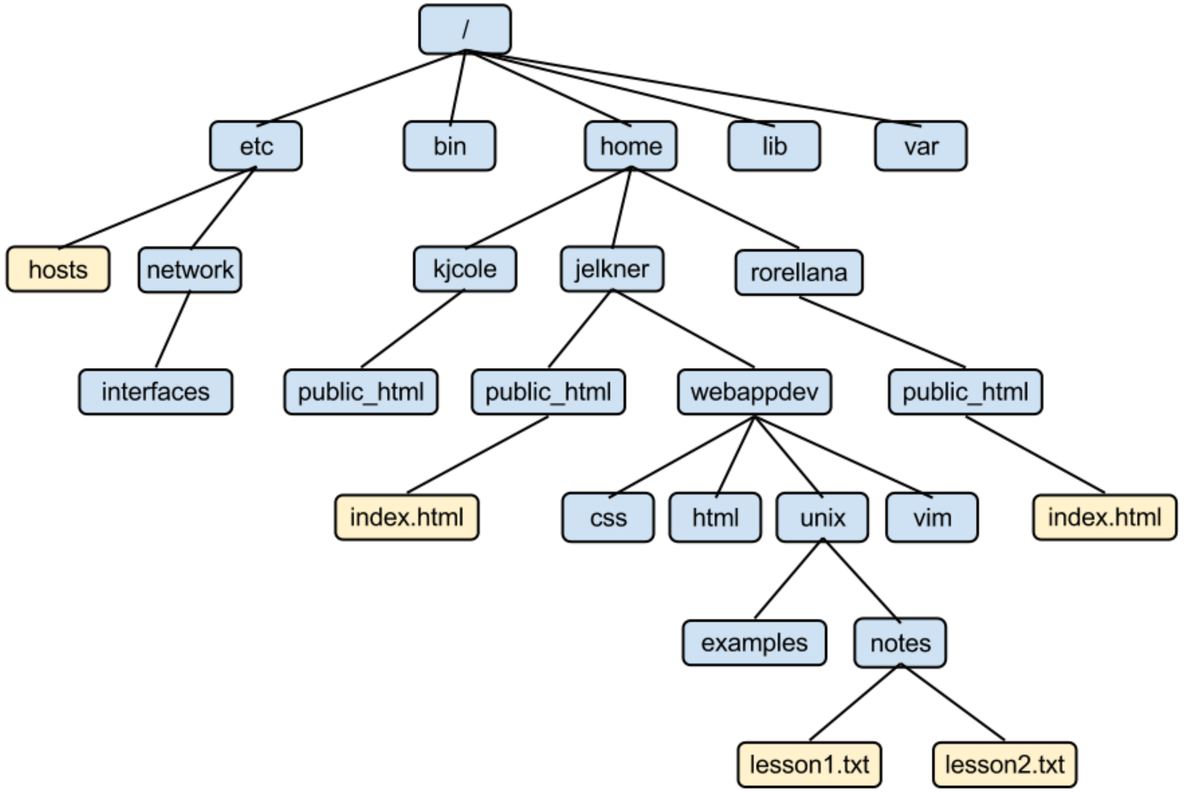
Stacks

- Review the java implementation of file counting
- directories are tree structures
- can hold files or directories



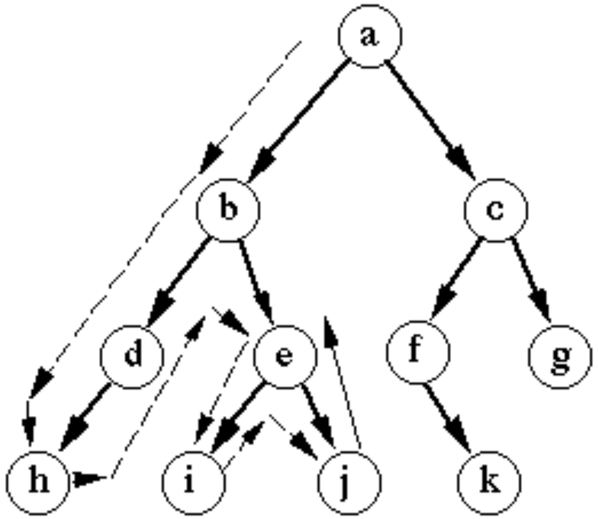
Stacks

- iterative file counting
- implement solution using stacks
- this is a form of graph search

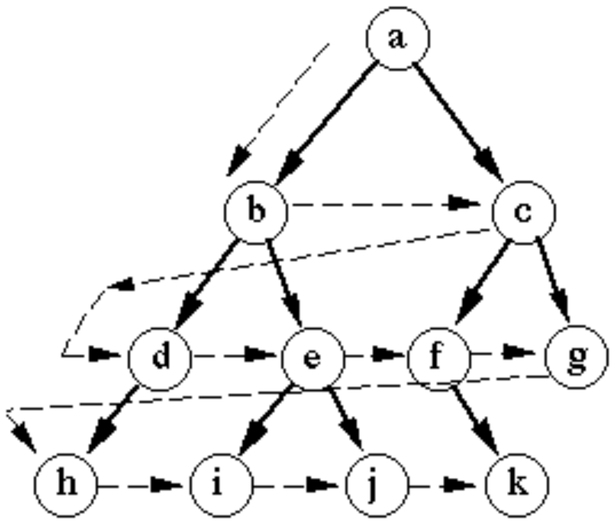


Stacks

- iterative file counting
- implement solution using stacks
- this is a form of graph search
- often implemented using stacks or queues



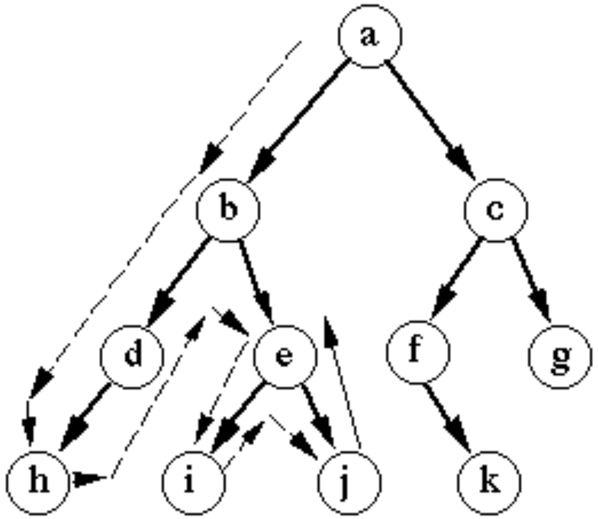
Depth-first search



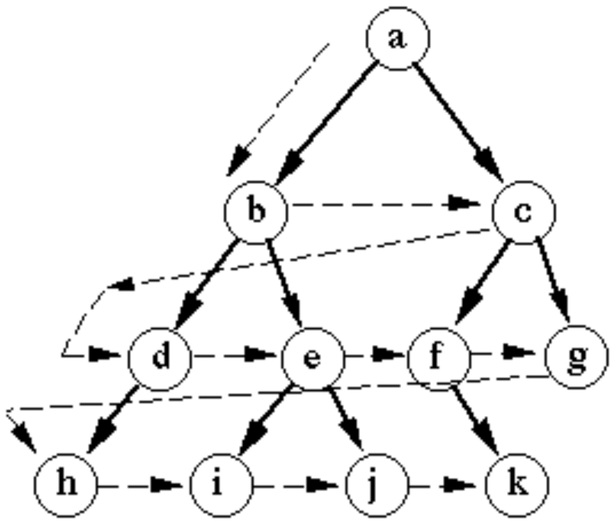
Breadth-first search

Stacks

- write a java function which counts the number of files (with a particular suffix) in a directory, using stacks and no recursion.



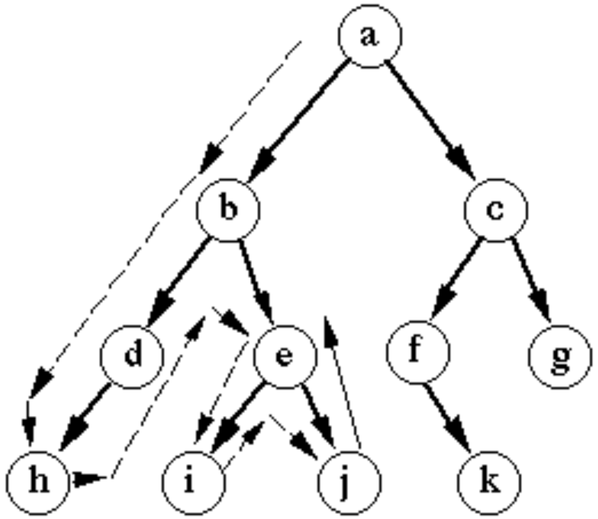
Depth-first search



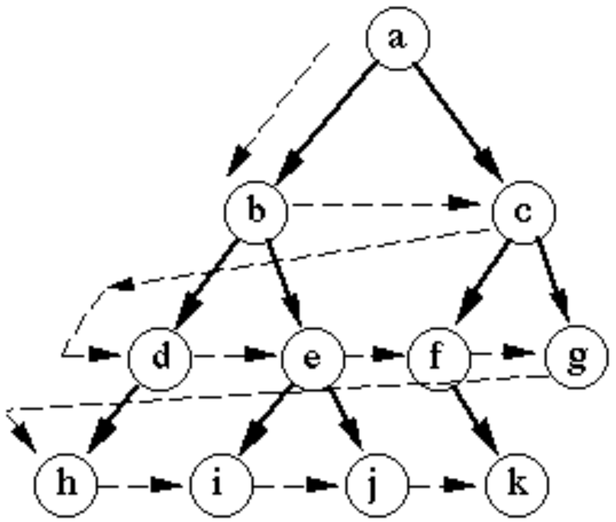
Breadth-first search

Stacks

- write a java function which counts the number of files (with a particular suffix) in a directory, using only recursion.



Depth-first search



Breadth-first search

Stacks

Using a stack to count files in directory.

```
package comp20010;

import java.io.File;

public class FileCounter {

   public static int countFilesStack(String dir, String suffix) {

      java.util.Stack<File> stack = new java.util.Stack<File>();
      int count = 0;

      stack.push(new File(dir));

      while (stack.isEmpty() == false) {
         File top = stack.pop();
         System.out.println("top:" + top.getAbsolutePath());
         for (File f : top.listFiles()) {
            System.out.println("f: " + f);
            if (f.isFile() && f.getName().endsWith(suffix)) {
               ++count;
            }
            if (f.isDirectory()) {
               stack.push(f);
            }
         }
      }
      return count;
   }
```

It is worth checking the api for file.list() and file.listFiles(). They are similar but list() returns only a list of strings of file names. For the FileCounter to work, we need the full path of the sub directories, and listFiles() gives us this.

`public String[] list(FilenameFilter filter)`
Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter. The behavior of this method is the same as that of the `list()` method, except that the strings in the returned array must satisfy the filter. If the given `filter` is `null` then all names are accepted. Otherwise, a name satisfies the filter if and only if the value `true` results when the `FilenameFilter.accept(File, String)` method of the filter is invoked on this abstract pathname and the name of a file or directory in the directory that it denotes.

`public File[] listFiles()`
Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
If this abstract pathname does not denote a directory, then this method returns `null`. Otherwise an array of `File` objects is returned, one for each file or directory in the directory. Pathnames denoting the directory itself and the directory's parent directory are not included in the result. Each resulting abstract pathname is constructed from this abstract pathname using the `File(File, String)` constructor. Therefore if this pathname is absolute then each resulting pathname is absolute; if this pathname is relative then each resulting pathname will be relative to the same directory.

```
   public static void main(String[] args) {
         String dir = new String("/Users/aonghus/work/spark.git");
         String suffix = new String(".java");
         System.out.println("countFilesStack() " + countFilesStack(dir, suffix));
      }
```

Count files recursively

- this is how we count files recursively
- make sure you can write this code yourself

```
/**
 * Count files in a directory (including files in all subdirectories)
 *
 * @param directory the directory to start in
 * @return the total number of files
 */
public static int countFilesInDirectoryRecursive(File dir, String suffix) {
   int count = 0;
   for (File file : dir.listFiles()) {
      if (file.isFile() && file.getName().endsWith(suffix)) {
         count++;
      }
      if (file.isDirectory()) {
         count += countFilesInDirectoryRecursive(file, suffix);
      }
   }
   return count;
}
```

Count files java streams

- this is how we count files using Java streams
- If you look up the docs for Files.walk(dir) you will find it is executing a depth first search (check out Slide.8)
- You can find some more information on Streams here:
- http://ocpj8.javastudyguide.com/ch25.html
- You don’t need to know Java Streams for this module.

```
public static long countPath(Path dir, String suffix) throws IOException {
    return Files.walk(dir)
            .parallel()
            .filter(Files::isRegularFile)
            .filter(p -> p.toString().endsWith(suffix))
            .count();
}
```

Testing File Counter

I tested the different ways of counting files to check they give the same answer (they did!).

```
public static void main(String[] args) throws IOException {
   StopWatch stopwatch = new StopWatch();
   String dir = new String("/Users/aonghus/work/spark.git/"); // change this
   String suffix = new String(".java");

   stopwatch.start();
   System.out.println("stack: " + countFilesInDirectoryStack(dir, suffix) + " -> " + stopwatch.elapsedSeconds());

   stopwatch.reset();
   stopwatch.start();
   System.out.println("recursive: " + countFilesInDirectoryRecursive(new File(dir), suffix) + " -> " + stopwatch.elapsedSeconds());

   stopwatch.reset();
   stopwatch.start();
   System.out.println("java nio: " + countPath(Paths.get(dir), suffix) + " -> " + stopwatch.elapsedSeconds());
}
```

StopWatch.java

```java
package comp20010;

// this Stopwatch class is modified from Google Guava Stopwatch:
// https://github.com/google/guava/blob/master/guava/src/com/google/common/base/Stopwatch.java

public class StopWatch {

     private boolean isRunning;
     private long elapsedNanos;
     private long startTick;

     /**
      * Creates (but does not start) a new stopwatch using {@link System#nanoTime} as
      * its time source.
      *
      */
     public static StopWatch createUnstarted() {
          return new StopWatch();
     }

     /**
      * Creates (and starts) a new stopwatch using {@link System#nanoTime} as its
      * time source.
      *
      */
     public static StopWatch createStarted() {
          return new StopWatch().start();
     }

     /**
      * Starts the stopwatch.
      *
      * @return this {@code Stopwatch} instance
      */
     public StopWatch start() {
          if (isRunning) {
               return this;
          }
          isRunning = true;
          startTick = System.nanoTime();
          return this;
     }

     /**
      * Stops the stopwatch. Future reads will return the fixed duration that had
      * elapsed up to this point.
      *
      * @return this {@code Stopwatch} instance
      */
     public StopWatch stop() {
          if (!isRunning) {
               return this;
          }
          long tick = System.nanoTime();
          isRunning = false;
          elapsedNanos += tick - startTick;
          return this;
     }

     /**
      * Sets the elapsed time for this stopwatch to zero, and places it in a stopped
      * state.
      *
      */
     public StopWatch reset() {
          elapsedNanos = 0;
          isRunning = false;
          return this;
     }

     private long elapsedNanos() {
          return isRunning ? System.nanoTime() - startTick + elapsedNanos : elapsedNanos;
     }

     public double elapsedSeconds() {
          long nanos = elapsedNanos();
          double seconds = (double) nanos / 1_000_000_000.0;
          return seconds;
     }

     public boolean isRunning() {
          return isRunning;
     }

     public static void main(String[] args) {

          StopWatch stopwatch = StopWatch.createStarted();
          try {
               Thread.sleep(1000);// 1000ms = 1sec
          } catch (InterruptedException e) {
               e.printStackTrace();
          }
          stopwatch.stop(); // optional

          //long millis = stopwatch.elapsed(MILLISECONDS);
          double secs = stopwatch.elapsedSeconds();

          System.out.println("time: " + secs); // formatted string like "12.3 ms"
     }

}
```