CmpSci 187 Discussion #4: Sorting Strings with Two Stacks Individual Handout

Marc Liberatore and John Ridgway

17 February 2015

Our goals here are to write a program using external code and an external file, and to demonstrate the use of stacks. The goal of the program is to take a file containing words (using lower case letters only), one word to a line, and print out the words in alphabetical order.

The following classes are in the `sorting-with-stacks.zip` code zip file on the course web site:

- a complete `Words` class that contains a `Stack<String>` object and can load this object with the contents of a file;
- a stub for a `LeftRightSorter` class that will use two stacks to put the words in an input stack into alphabetical order, and then either print the words out or find a specific one; and
- a complete `Driver` class that tests these classes.

Our idea to sort the words is to use two stacks, `left` and `right`. We want to insert words into the two stacks, maintaining the invariants that:

1. the words in `left` are in alphabetical order with the first word on the bottom;
2. every word in `left` comes before every word in `right`;
3. The words in `right` are in alphabetical order with the first word on the top.

To maintain these invariants while inserting a new word, you need the new word to be between the words on the tops of the two stacks. To get that to happen, you need methods to shift the top word of `left` on to the top of `right`, and vice versa. Before you start coding, we suggest you work through this using a few sheets of paper with words written on them.

Note that we are using the Java `Stack` class, so that an invocation of `pop` both removes and returns the top element, and the test for an empty stack is `empty`, not `isEmpty`.

Your task is to add the following methods to `LeftRightSorter`:

- `public void shiftLeft()`, which moves the top element of the right stack to the top of the left stack;
- `public void shiftRight()`, which moves the top of the left stack to the top of the right;

- `public void makeRoom(String w)`, which shifts elements from one stack to the other until `w` can legally be pushed onto the left stack;
- `public void loadStacks(Stack<String> words)`, which moves all the `String`s from the stack `words` into the `left` and `right` stacks, maintaining the conditions 1-3 above; and
- `public String wordAt(int n)`, which returns the `n`th string in alphabetical order among those stored in the `left` and `right` stacks, assuming the conditions hold. (Of course the bottom of the `left` stack is the 0th string, not the 1st.)

We give you `public void printStacks()`, which prints out the contents of the `left` and `right` stacks.

Once you've written these, and tested the resulting class with the driver, we'd like you to find the 1,000th, 2,000th, 3,000th, 4,000th, and 5,000th strings in the file `data/knuthWords.dat` that we are also giving you – it contains a list of 5,727 English five-letter words compiled by Donald E. Knuth.

There's no reason that `wordAt` has to restore the stacks to the state they were in — in fact you may do whatever you want to the two stacks as long as you preserve all three conditions. That being said, the easiest way to implement `wordAt` is probably to move everything to the `right` stack, execute `shiftLeft` n times, then `peek` at the top of the `right` stack.

It's fine if `wordAt` throws an exception if you ask it for an index that is too large. And your shift methods may also cause stack underflow if they are called at the wrong time. But the `makeRoom` and `wordAt` methods should have guards so that they don't cause exceptions in normal operation.