© Manfred Kerber
Ian Batten
Joseph Gardiner

School of Computer Science
University of Birmingham

Worksheet 5

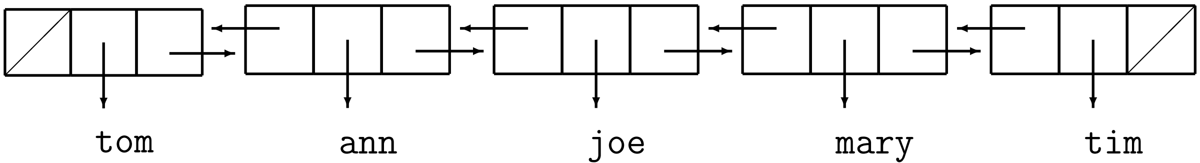
MSc/ICY Software Workshop

Assessed Exercise: 20% of this term's continuous assessment mark.

Submission: Wednesday 10 December 2014 2pm

5% late submission penalty within the first 24 hours. No submission after 24 hours. JUnit tests and JavaDoc comments are mandatory. All submissions must pass the tests provided on 3 December. Follow the submission guidelines on
`http://www.cs.bham.ac.uk/internal/courses/java/msc/submission.php`.

Exercise 1: (Basic, 30%) A doubly linked list is similar to a linked list, however, it has in addition to a pointer to the next element also one to its predecessor. E.g., a seating order in a cinema may be represented this way. Everybody (except the two at the outer ends) has two neighbours, their right and their left neighbour.



Assume a class `Person` with the three field variables `String name`, `String gender`, and `int age`. Build a recursive structure `DoublyLinkedList` of persons with corresponding constructors, getters (in particular `left()` and `right()` returning the left and the right sub-list, respectively), setters, a `toString` method, and an `equals` method. When you add elements to a doubly linked list put them always to the front, that is, write a method so that `cons(tom, cons(ann, cons(joe, cons(mary, cons(tim, empty())))))` generates the list displayed above (with appropriately generated objects such as `Person tom = new Person("Tom", "M", 22);`.) Write a method that checks for an element in the doubly linked list whether all their neighbours are of the opposite gender (i.e. for inner list members whether the left and the right neighbours are, and for the leftmost/rightmost whether the right/left neighbour is, respectively).

Exercise 2: (Basic, 30%) Use the `Person` class from the previous exercise and build a binary search tree of persons ordered by the name (in a class `BinarySearchPerson`). That is, when a new person is inserted into an empty tree then it should be put as the `value`, if the element is smaller than the `value` it should be inserted into the left sub-tree, if it is equal to the `value` it should be ignored, and if it is bigger than the `value` it should be inserted into the right subtree. The tree should be built following the lexicographical order of the String representing the `name` of the `Person`. Note, however, that you should not insert strings into the tree, but objects of type `Person`.
Two strings can be compared with respect to the lexicographical order by the `compareTo` method. For two strings `str1` and `str2` we have `str1.compareTo(str2)` returns an `int` less than 0 if `str1` comes before `str2` in the lexicographical order; it is equal to 0 if the strings are equal; and it is bigger than 0 if `str2` comes before `str1` in the lexicographical order. (E.g., `"abc"` comes before `"xyz"`, likewise `"abc"` before `"abxy"`. Shorter strings with the same start before longer ones, e.g., `"ab"` before `"abc"`. It is the order that is used in a lexicon.)

Write a method `lookupAge(String name)` which searches in the binary search tree for a person with name `name` and returns their `age` if such a person is found. It should return `-1` if no such person is found.

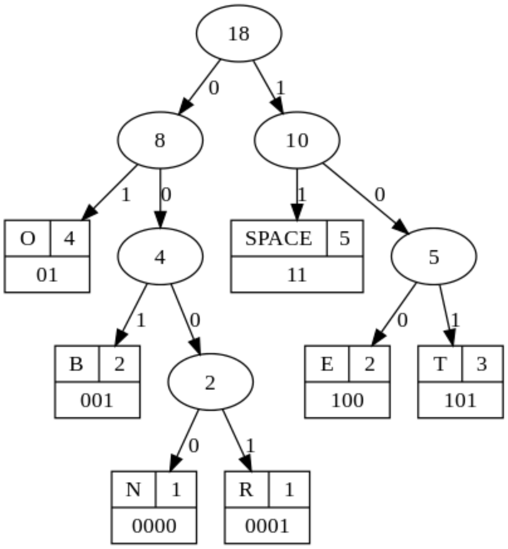
Exercise 3: (Medium, 20%) On `http://www.cs.bham.ac.uk/internal/courses/java/msc/handouts/exercises/DonQuixote.txt` you find a modified version of Don Quixote as an ebook of the project Gutenberg. Adjust the code from `http://www.cs.bham.ac.uk/internal/courses/java/msc/handouts/1-05/Html.java` to read from the file. Build a frequency table of the letters `a-z`, of the empty space, the full stop, and the newline sign in this order. That is, you should read in the file and build an array `long[] frequency = new long[29]` so that `frequency[0]` gives the number of occurrences of `'a'` in the text, `frequency[1]` that of the `'b'` and so on, `frequency[25]` that of the `'z'`, `frequency[26]` that of `' '`, `frequency[27]` that of `'.'`, and `frequency[28]` that of `'\n'`. Other symbols should be ignored.

Exercise 4: (Advanced, 20%) Huffman encoding is a lossless compression method. It is based on a frequency analysis with respect to the alphabet used. E.g., in English the letters ‘e’ and ‘t’ occur often, whereas ‘z’, ‘q’, and ‘x’ rarely. The essential idea is to represent all letters of the alphabet uniquely by sequences of bits (0 and 1 only), so that the frequently occuring letters are represented by short bit sequences and rarely occuring ones by longer bit sequences. This is done by using the frequencies in the language (or the text) and building up the so-called Huffman tree. For details, see e.g., `http://en.wikipedia.org/wiki/Huffman_coding`.

(a) Use the frequencies computed in Exercise 3 to build a Huffman tree.

(b) Write two static methods `encode` and `decode`. `encode` takes a `String` and a Huffman tree and returns a `String` of `0`s and `1`s that encodes the string according to the Huffman tree. `decode` is the inverse method that takes the String of `0`s and `1`s and returns the corresponding original string.

The following example of a Huffman tree is taken from Wikipedia, `http://en.wikipedia.org/wiki/DOT_%28graph_description_language%29#mediaviewer/File:Huffman_%28To_be_or_not_to_be%29.svg`.


18
0
1
8
10
1
0
O 4
01
4
1
0
SPACE 5
11
5
1
0
B 2
001
2
0
1
E 2
100
T 3
101
0
1
N 1
0000
R 1
0001