ITI 1121. Introduction to Computing II - Summer 2015

Assignment 4 (100 points, weight 6.25%)

(Last Modified in July 25, 2015)

Due date: Friday Jul 31 at 11:59PM.
The assignment must be uploaded on Virtual Campus by the due date.
Late assignments are accepted between 1 min late up to a maximum of 24 hours late and they will receive a 30% penalty.

Learning objectives

- Gaining more experience with data structures like Queues, Linked Lists and Binary Search Trees.
- Reviewing several concepts learned as a practice for the final exam.

Part 1: Generating Partial Permutations (45/100)

A **permutation** of n is an ordering of all the numbers in $\{1,2,\ldots,n\}$.
For example, the following are two different permutations of 5:
(2,5,3,1,4)
(1,3,4,5,2)

A **partial permutation** of n is a (ordered) sequence of numbers from $\{1,2,\ldots,n\}$ that does not have repeated numbers.
For example, the following are different partial permutations of 5 (of sizes 3, 4, and 5 respectively):
(3,1,5)
(1,4,2,5)
(1,4,2,5,3)
In class, you have learned how to generate all the strings from a given alphabet by using a Queue data structure. In this assignment, you will design a similar algorithm to generate a collection of partial permutations of specified type. This method differs from the one that generates all the strings since not every string and substring is valid; more specifically you need not consider strings or substrings that have repetitions. It is expected that you do this efficiently. That is, using an algorithm that generates all strings first and filters out the invalid ones at the time of printing is not an acceptable solution as the memory requirements for the Queue would make it impossible to solve even moderate size instances. You are better off restricting the sequences and subsequences considered to the ones that do not contain repetitions. Three variations of generation of partial permutations will be employed:

- FIXEDSIZE: given an alphabet of N symbols and a size SZ, generates all partial permutations of the alphabet that has size equals SZ.
- UPTOSIZE: given an alphabet of N symbols and a size SZ, generates all partial permutations of the alphabet that has the size smaller than or equal SZ.
- MAXSIZE: given an alphabet of N symbols, generates all permutations of the alphabet (these are partial permutations of maximum size N)

You are given classes:

- Queue interface and implementation: Queue.java and LinkedQueue.java **(These classes must not be changed)**
- PartialPermutations.java: the main class the deals with the Partial Permutation Generation. The main attributes of the generation are: alphabet (letters or single number symbols), size (a number smaller or equal to N, the size of the alphabet) and mode (specifying the type of generation - UPTOSIZE,FIXEDSIZE,MAXSIZE). In addition to auxiliary methods (geters and seters of the main variables) you need to implement the following methods:
  - **public String generateWords()**
    returns a string containing all partial permutations of the specified types and sizes, separated by ";"
    The list must be in order of size and for the same size, in alphabetical order (the order that respects the order in which the alphabet was given). The idea is that method visit (see below) takes care of the selection of what to keep depending on the mode of the generation; while generateWords() pretty much implements a basic generation algorithm that works for all types of generation. We recommend that you use LinkedQueue to generate and store partial permutations. Please do not change the return format, since correctness of your method is tested in PartialPermutationTest by comparing the expected return value with the value returned by this method.
  - **private String visit(String word)**
    This method is auxiliary to generateWords(). Method generateWords() may be going through several intermediate partial permutations it needs to examine and to enqueue but may not be needed to be included in the final generation list; the method visit(word) decides if the word must go to the final list. It returns the empty string "" if the word does not go to the final list, it returns the string "<empty>" if the word goes in the final list and is the empty word, otherwise (nonempty word that goes to the final list) it returns the word itself.

- PartialPermutationsTest.java: a class to test the Partial Permutation Generation.
  (This class can be changed for your own testing of different inputs; however the original version must still work with the classes you modified)

We give a sample output for alphabets {a,b,c,d}, {1,2,3,4,5}, respectively:

Output Mode UPTOSIZE size=3 alphabet size=4
<empty>;a;b;c;d;ab;ac;ad;ba;bc;bd;ca;cb;cd;da;db;dc;abc;abd;acb;acd;adb;adc;bac;bad;bca;bcd;bda;bdc;cab;cad;cba;cbd;cda;cdb;dab;dac;dba;dbc;dca;dcb;

Output Mode FIXEDSIZE size=4 alphabet size=5
1234;1235;1243;1245;1253;1254;1324;1325;1342;1345;1352;1354;1423;1425;1432;1435;1452;1453;1523;1524;1532;1534;1542;1543;2134;2135;2143;2145;2153;2154;2314;23

Part 2: Practice with Binary Search Tree and Ordered Lists (50/100)

This part builds on classes you have used in Laboratory 11 and Laboratory 13.

The first thing to be done will be do add an auxiliary method to the class BinarySearchTree.java:

- **private E find(E obj, Node current)**
  This method behaves pretty much like the method **contains(obj,current)** in that it verifies whether the binary search tree contains an element whose comparisonTo(obj) gives zero. The difference is that instead of returning **true**, the find method returns the element found, and instead of returning **false**, the find method returns **null**.
  Note that this method is the private recursive method used by the method **public E find(E obj)** already given in the class.

In this exercise, you will be using a Binary Search Tree whose "key" is a course letter grade (A+, A, A-, B+, B, C+, C, etc.) and stores in addition to the letter grade, an ordered

list of student names with this letter grade.

To fit this more complex structure as the element stored in the BinarySearchTree.java class, we have created a class StudentListWithGrade.java
This class is an ordered list of Strings representing student names and also contains a letter grade. It implements the interface Comparable<StudentListWithGrade> and the comparison is solely based on the letter grade (the list of student does not enter into account for the comparison of objects in this class.

The effect of creating a BinarySearchTree<StudentListWithGrade > is that the binary search tree will store the list of students with a certain grade in each node and the binary search tree comparison of elements will be based on the letter grade associated with the list.

If the complete list of students for each grade were available before creating this binary search tree, there would be not much else to be done in this assignment, other than creating BinarySearchTree and inserting these lists one by one. However, the students with their grades can become available at any given point in time, even after the tree has being created and a list with some students with that grade have already been inserted into the binary search tree. This require that the method "add" for this structure has to be done a bit more carefully.

The BinarySearchTree<StudentListWithGrade> will be stored inside a class StudentByGrade.java.
This class will be responsible to handle the different types of insertion via two methods add that must be implemented by you:

- **public boolean add(String grade, String name)**
  This method receives a letter **grade** and student **name**. The insertion must behave in the following way.
  If the binary search tree does not contain a list of student with this **grade**, such an object of class StudentListWithGrade must be created with this **grade** and student **name**, and inserted into the binary search tree.
  If the binary search tree already contains a student with this grade, this element (an instance of class StudentListWithGrade) should be located in the tree and the name must be inserted into this existing list.

- **public boolean add(StudentListWithGrade myList)**
  This method receives an object of class StudentListWithGrade. The insertion must behave in the following way.
  If the binary search tree does not contain a list of students with the same grade as the grade of **myList**, then **myList** is simply inserted into the binary search tree.
  If the binary search tree already contains a student list with the same grade as the grade of **myList**, this element (an instance of class StudentListWithGrade) should be located and **myList** should be merged into it.

Note that the methods above will need to use the new **find** method of BinarySearchTree.java

To test the class StudentByGrade.java, you have been given StudentByGradeTest.java that must not be modified.

A SAMPLE OF BEHAVIOUR FOR THE ADD METHOD

The following statements:
```
StudentByGrade course = new StudentByGrade();
course.add("A+","Smith"); course.add("B+","Curtis"); course.add("A+","Black");
course.add("B","Silva"); course.add("A-","Moura"); course.add("A+","Collins");
course.add("F","Maradona"); course.add("B","Rosa"); course.add("B","Pitt");
course.add("A","Bryan"); course.add("C","Ryan"); course.add("D", "Danny");
course.add("C", "Lennon"); course.add("E", "Newman"); course.add("C+", "Harrison");
System.out.println("Showing Tree Format:\n"+course.treeToString());
System.out.println("\nShowing list Format:\n"+course);
```

Produce the output:
```
Showing Tree Format:
(((((F[Maradona](((()E[Newman]())D[Danny]())C[Lennon,Ryan](()C+[Harrison]())))B[Pitt,Rosa,Silva]())B+[Curtis](()A-[Moura](()A[Bryan]())))A+[Black,Collins,Smith]())

Showing list Format:
F[Maradona]
E[Newman]
D[Danny]
C[Lennon,Ryan]
C+[Harrison]
B[Pitt,Rosa,Silva]
B+[Curtis]
A-[Moura]
A[Bryan]
A+[Black,Collins,Smith]
```

Rules and regulations (5/100)

Please follow the general directives for assignments.
You must abide by the following submission instructions:
Include all your java files into a director called **u1234567** where 1234567 is your student id.
Zip this directory and submit via blackboard.

The directory must contain the following files (with your implementation):

- Files for both parts:
  - README.txt
  - StudentInfo.java
- Files for Part 1:
  - Queue.java (given; not to be changed)
  - LinkedQueue.java (given; not to be changed)
  - PartialPermutations.java **(implement methods: visit, generateWords)**
  - PartialPermutationsTest.java (given; not to be changed)
- Files for Part 2:
  - OrderedStructure.java (given; not to be changed)
  - OrderedList.java (given; not to be changed)

- BinarySearchTree.java **(implement method: private E find(E obj, Node current))**
- StudentListWithGrade.java (given; not to be changed)
- StudentByGrade.java **(implement 2 methods "add" with the required signatures)**
- StudentByGradeTest.java (given; not to be changed)

The assignment is individual (plagiarism or collaborations towards the solution of the assignment between students will not be tolerated).
Read the information on academic fraud from other assignments.