Sorting and Priority Queues

The purpose of this assignment is to familiarize the reader with sorting algorithms and the use of priority queue, one of the most basic data structures in algorithm design. The assignment deals with questions encountered in optimization problems related to the derivation of memory resources for large-scale data mining. We will consider the following simple assumption. Suppose that we want to place all the contents of a set of N file folders with files on large disks with a capacity of 1 TB each (consider, for example, the application of creating backup copies on a system file management system). We assume that all the volumes are between 0 and 1,000,000 MB (1 TB) in size. We have the constraint that each volume must be entirely stored on a disk. Ideally, the optimal solution would be to use the smallest possible number of hard disks. The problem is an example of the well-known bin packing problem, for which we do not yet know whether there is an efficient algorithm that can always find the optimal solution. We can however design efficient methods of approximating the optimal solution.

Part A

Let's see the Abstract Data Types that are required.

Disk ADT: To implement the algorithm you must first implement a data type representing a 1TB disk. Name this class Disk. Instances of the Disk class must:
- have unique id's that are referenced when a new disk is created (useful for debugging).
- contain a list named folders, defined as a empty in which the folders are placed that are saved to this disk over the duration of the save algorithm.
- have the getFreeSpace() method which, when called, returns the free space of the disk in MB. In the Disk class you can also add any other field you find useful. Finally, Disk must implement the Comparable<Disk> interface so that it can be used in a priority queue.

To implement the Comparable<Disk> interface (and thus the compareTo function) you can simply compare the free space of a disk: if disk A has more free space than disk B then we consider that A > B and the operation A.compareTo(B) must return 1. Similarly, when A < B the function returns -1 and when A == B the function returns 0.

Priority Queue ADT: For this task you will need an efficient data structure for a priority queue. You can either use the tutorial queue or create your own queue, based on the lecture slides. In this case, your queue should definitely include the insert and getmax functions, as we have seen in the example. Name this function MaxPQ.
For the list of folders of the Disk class, you are not allowed to use existing implementations of list type structures from the Java library (e.g. ArrayList, LinkedList, etc.). Use either the list in the tutorial, or make your own list.

Part B

Implement the save algorithm described below. Name the program Greedy.java.
Algorithm 1 – Greedy:

Edit the files one by one in the order they appear. If a file fits on any of the disks we have already used so far, we discard it on the tray with the most free space. Alternatively, if it doesn't fit on one, we use a new disk and we save the file there.

Example:

Suppose the algorithm sees a sequence of 5 files of size 200.000, 150.000, 700.000, 800.000, 100.000 (in MB). The result of the algorithm will be to use 3 disks where it will remove the folders 1, 2 and 5 with sizes 200.000, 150.000, 100.000 on disk 1, folder 3 with size 700.000 on disk 2 (because at the time we processed this file, it did not fit on disk 1), and folder 4 with a size of 800.000 on disk 3. **Note** that in the example described, the optimal solution would be to use 2 disks instead of 3 (disks 1, 4 on 1, and disks 2, 3, 5 on 2). Although Algorithm 1 is not always optimal in terms of the number of disks used, it is a reasonably fast algorithm that in many cases can approximate the optimal solution.

Input and Output

Input. The Greedy.java program will contain a main method which will first read the sizes of the folders from a txt file to enable it to run the algorithm. Each line of the file will have the size of a folder to be extracted in MB, so it will be an integer between 0 and 1000000. In the example mentioned above, the input file will have the following format:

200000
150000
700000
800000
100000

If a folder does not have a size between 0 and 1.000.000 you should print an error message and terminate the program.

Output. The program a should calculate and print the number of disks used by the algorithm, as well as the sum of the sizes of all disks in TB (note that this is the lowest limit to the minimum number of disks required). Also if the number of folders to be saved is not greater than 100, print the contents of the disks in descending order according to the empty space of the disk. For every disk, print its id, the size of its empty space and then the size of the file that has been removed from the disk.
An example of the form of the output:

% java Greedy input.txt

Sum of all folders = 6.580996 TB

Total number of disks used = 8

id 5 325754: 347661 326585

id 0 227744: 420713 351543

id 7 224009: 383972 392019

id 4 190811: 324387 484802

id 6 142051: 340190 263485 254274

id 3 116563: 347560 204065 331812

id 2 109806: 396295 230973 262926

id 1 82266: 311011 286309 320414

The above example shows that 8 disks were used to extract data with a total of size 6,580996 TB, e.g. disk with id 5 has 325754 MB of free space and contains 2 folders of size 347661 and 326585 respectively.