1 Programming Project (PrP) Logistics

Warning: Besides the algorithmic-related programming, for some or all of the options you need to provide command-line processing or file-based input/output. If you are not familiar with those topics and requirements we urge you to become so as soon as possible. Thus start familiarizing with them early in the semester. The more you procrastinate the more problems you will potentially face when you integrate these components with the algorithmic-related material.

STEP-1. Read carefully Document 4 and follow all the requirements specified there; moreover, observe naming conventions, comments and testing as specified in sections 2-4 of Document 4 and how to avoid having 80pts deducted.

STEP-2. When the archive per Document 4 guidelines is ready, upload it and submit it to canvas

BEFOR NOON-time as in CALENDAR (Document 1, Syllabus)

You may do one or two options (or none at all). You may utilize the same language or not. **We provide descriptions that are to the extent possible language independent: thus a reference in Java is a pointer in C or C++ for example.**

OPTION 1 (Hash Table related). Do the programming related to the building of a Hash Table that can maintain arbitrarily long strings in C, C++, or Java; it is similar to that also used by Google around 1997-1998.

OPTION 2. Do the programming part related to Kleinberg's HITS, and Google's PageRank algorithms in Java, C, or C++.

Either implementation should be optimized enough for a test execution to take no more than few seconds, maximum 15 seconds forgivingly.

2 OPTION 1: Hashing (130 points)

We are asking you to implement a Lexicon structure to store arbitrarily long strings of ASCII chars (i.e. words). Lexicon L uses a Hash Table T structure along with an Array A of NUL separated words. In our case words are going to be English character words only (upper-case or lower case). Table T will be organized as a hash-table using collision-resolution by open-addressing as specified in class. You are going to use quadratic probing for $h(k,i)$ and keep the choice of the quadratic function simple: i^2 so that $h(k,i) = (h'(k) + i^2) \bmod N$. The keys that you will hash are going to be English words. Thus function $h'(k)$ is also going to be kept simple: the sum of the ASCII/Unicode values of the characters minus 2 mod N, where N is the number of slots of T. Thus 'alex' (the string is between the quotation marks) is mapped to $97 + 108 + 101 + 120 - 2 \bmod N$ whatever N is. In the example below, for $N = 11$, $h(\mathbf{alex}, 0) = 6$. Table T however won't store key k in it. This is because the keys are of arbitrary length. Instead, T will store pointers/references in the form of an index to another array A. The second table, array A will be a character array and will store the words maintained in T separated by NUL values `\0`. This is not 2 characters a backslash followed by a zero. It is 1B (ASCII), 2B (UNICODE) whose all bits are set to 0, the NUL value. If you don't know what B is, it is a byte; read Document 2.

An **insertion** operation affects T and A. A word w is hashed, an available slot in T is computed and let that slot be t. In $T[t]$ we store an index to table A. This index is the first location that stores the first character of w. The ending location is the `\0` following w in A. New words that do not exist (never inserted, or inserted but subsequently deleted) are appended in A. Thus originally you need to be wise enough in choosing the appropriate size of A. If at some point you run-out of space, you increase T and thus this increases A as well. Doubling T i.e. N, is an option. This causes problems that you also need to attend to. A **deletion** will modify T as needed but will not erase w from A. Let it be there. So A might get dirty (i.e. it contains garbage) after several deletions. If several operations later you end up inserting w after deleting it previously, you do it the **insertion** way and you reinsert w, even if a dirty copy of it might still be around. You DO NOT DO a linear search to find out if it exists arleady in A; it is inefficient. There is not much to say for a **search**.

You need to support few more operations: **Print** , **Create**, **Empty/Full/Batch** with the last of those checking for an empty or full table/array and a mechanism to perform multiple operations in batch form. **Print** prints nicely T and its contents i.e. index values to A. In addition it prints nicely (linear-wise in one line) the contents of A. (For a `\0` you will do the SEMI obvious: print a backslash but not its zero). The intent of **Print** is to assist the grader. **Print** however does not print the words of A for deleted words. It prints stars for every character of a deleted word instead. (An alternative is that during deletion each such character has already been turned into a star: if you do so report it to the .txt file to avoid penalties.) Function **Create** creates T with N slots, and A with $15N$ chars and initializes A to spaces. We call a class that supports and realizes A and T a lexicon: L is one instance of a lexicon. Your code should thus implement as functions minimally the functions mentioned above: Create, Print, Empty, Full, Batch, Insert, Delete, Search.

Testing utilizes a `Batch` function with argument a filename that is going to be provided as a command line argument. That file consists of multiple lines containing one command per line. An example file is provided in Section B of the course web-page related to the example below. Each command is a numeric equivalent of the function named earlier plus one more (for comment). Command 10 is **Insert**, Command 11 is **Deletion**, and Command 12 is **Search**. Command 13 is **Print**, Command 14 is **Create**. Command 15 is **Comment**: the rest of the line marked by a 15 is ignored. Command 14 for create has an argument which is the value of N. Each one of 10, 11, and 12 has an argument which is a string.

```
% java mplexicon  filearbitrary.txt
% ./mplexicon     file.txt
14 11
10 alex
10 tom
10 jerry
15 ready-to-print                CAUTION: 15 is a comment string (chars,numbers,-)
13                                        operation 15 is skipped/ignored
```

The six-line batch file above will print the following. The *T* entries for 0, 5, 9 are the indexes (first position) for `alex, tom, jerry` respectively. Note that the ASCII values has for 'alex' as prescribed give a 6, but for 'tom' and 'jerry' give 2, i.e. a collision occurs. A minimal output for Print is available below.

```
    T                 A: alex\tom\jerry\
 0:                                   CAUTION: \ means \0
 1:                                   \t is not a tab character !!!
 2:
 3:
 4: 5
 5: 9
 6: 0
 7:
 8:
 9:
10:
```

If the following lines were added to the file

```
12 alex
12 tom
12 jerry
12 mary
11 tom
13
```

they will generate in addition on screen

```
 alex   found at slot 6
 tom    found at slot 4
 jerry  found at slot 5
 mary   not found
 tom    deleted from slot 4

    T                A: alex\***\jerry\
 0:
 1:
 2:
 3:
 4:
 5: 9
 6: 0
 7:
 8:
 9:
10:
```

Deliverables. An archive per Document 4 guidelines.

3 OPTION 2: HITS and PageRank implementations (130 points)

Implement Kleinberg's HITS Algorithm, and Google's PageRank algorithm in Java, C, or C++ as explained.

(A) Implement the HITS algorithm as explained in class/Subject notes adhering to the guidelines of Document 4. Pay attention to the sequence of update operations and the scaling. For an example of the Subject notes, you have output for various initialization vectors. You need to implement class or function `hits` (e.g. `hitsWXYZ`. For an explanation of the arguments see the discussion on PageRank to follow.

```
% java hits iterations initialvalue filename
% ./hits    iterations initialvalue filename
```

(B) Implement Google's PageRank algorithm as explained in class/Subject notes adhering also to the guidelines of Document 4 and this description. The input for this (and the previous) problem would be a file containing a graph represented through an adjacency list representation. The command-line interface is as follows. First we have the class/binary file (eg pgrk). Next we have an argument that denotes the number of `iterations` if it is a positive integer or an `errorrate` for a negative or zero integer value. The next argument `initialvalue` indicates the common initial values of the vector(s) used. The final argument is a string indicating the `filename` that stores the input graph.

```
% ./pgrk     iterations initialvalue filename  // in fact pgrkWXYZ
% java pgrk  iterations initialvalue filename  // in fact pgrkWXYZ
```

The two algorithms are iterative. In particular, at iteration t all pagerank values are computed using results from iteration $t-1$. The `initialvalue` helps us to set-up the initial values of iteration 0 as needed. Moreover, in PageRank, parameter d would be set to 0.85. The PageRank $PR(A)$ of vertex A depends on the PageRanks of vertices $T_1,\ldots,T_m$ incident to A, i.e. pointing to A; check Subject 7 section 3.6 for more details. The pageranks at iteration t use the pageranks of iteration $t-1$ (synchronous update). Thus PR(A) on the left in the PageRank equation is for iteration t, but all PR (T_i) values are from the previous iteration $t-1$. Be careful and synchronize! In order to run the 'algorithm' we either run it for a fixed number of iterations and `iterations` determines that, or for a fixed `errorrate` (an alias for `iterations`); an `iterations` equal to 0 corresponds to a default errorrate of 10^{-5}. A `-1, -2, etc , -6` for `iterations` becomes an errorrate of $10^{-1}, 10^{-2}, \ldots, 10^{-6}$ respectively. At iteration t when all authority/hub/PageRank values have been computed (and auth/hub values scaled) we compare for every vertex the current and the previous iteration values. If the difference is less than **errorrate** for EVERY VERTEX, then and only then can we stop at iteration t.

Argument `initialvalue` sets the initial vector values. If it is `0` they are initialized to 0, if it is `1` they are initialized to 1. If it is `-1` they are initialized to $1/N$, where N is the number of web-pages (vertices of the graph). If it is `-2` they are initialized to $1/\sqrt{N}$.

Argument `filename` describes the input (directed) graph and it has the following form. The first line contains two numbers: **the number of vertices followed by the number of edges** which is also the number of remaining lines. PAY ATTENTION THAT NUMBER of VERTICES comes first. The sample graph is treacherous: n and m are the same! All vertices are labeled $0,\ldots,N-1$. Expect N to be less than 1,000,000. In each line an edge (i,j) is represented by `i j`. Thus our graph has (directed) edges $(0,2),(0,3),(1,0),(2,1)$. Vector values are printed to 7 decimal digits. If the graph has N GREATER than 10, then the values for `iterations, initialvalue` are automatically set to `0` and `-1` respectively. In such a case the hub/authority/pageranks at the stopping iteration (i.e t) are ONLY shown, one per line. The graph below will be referred to as `samplegraph.txt`

```
4 4
0 2
0 3
1 0
2 1
```

The following invocations relate to `samplegraph.txt`, with a fixed number of iterations and the fixed error rate that determines how many iterations will run. Your code should compute for this graph the same rank values (intermediate and final). A sample of the output for the case of $N > 10$ is shown (output truncated to first 4 lines of it).

```
% ./pgrk  15 -1 samplegraph.txt
Base   :  0 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.2500000 P[ 3]=0.2500000
Iter   :  1 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  2 :P[ 0]=0.2500000 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  3 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  4 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  5 :P[ 0]=0.1732344 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  6 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  7 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  8 :P[ 0]=0.1496625 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  9 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   : 10 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 11 :P[ 0]=0.1424245 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 12 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 13 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0970858 P[ 3]=0.0970858
Iter   : 14 :P[ 0]=0.1402020 P[ 1]=0.1200230 P[ 2]=0.0970858 P[ 3]=0.0970858
Iter   : 15 :P[ 0]=0.1395195 P[ 1]=0.1200230 P[ 2]=0.0970858 P[ 3]=0.0970858
```

```
% ./pgrk -3 -1 samplegraph.txt
Base   :  0 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.2500000 P[ 3]=0.2500000
Iter   :  1 :P[ 0]=0.2500000 P[ 1]=0.2500000 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  2 :P[ 0]=0.2500000 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  3 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1437500 P[ 3]=0.1437500
Iter   :  4 :P[ 0]=0.1732344 P[ 1]=0.1596875 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  5 :P[ 0]=0.1732344 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  6 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1111246 P[ 3]=0.1111246
Iter   :  7 :P[ 0]=0.1496625 P[ 1]=0.1319559 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  8 :P[ 0]=0.1496625 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   :  9 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.1011066 P[ 3]=0.1011066
Iter   : 10 :P[ 0]=0.1424245 P[ 1]=0.1234406 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 11 :P[ 0]=0.1424245 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 12 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0980304 P[ 3]=0.0980304
Iter   : 13 :P[ 0]=0.1402020 P[ 1]=0.1208259 P[ 2]=0.0970858 P[ 3]=0.0970858
```

```
% ./pgrk  0  -1 verylargegraph.txt
Iter   : 4
P[ 0]=0.0136364
P[ 1]=0.0194318
P[ 2]=0.0310227
   ... other vertices omitted
```

For the HITS algorithm, you need to print two values not one. Follow the convention of the Subject notes

```
Base   :  0 :A/H[ 0]=0.3333333/0.3333333 A/H[ 1]=0.3333333/0.3333333 A/H[ 2]=0.3333333/0.3333333
Iter   :  1 :A/H[ 0]=0.0000000/0.8320503 A/H[ 1]=0.4472136/0.5547002 A/H[ 2]=0.8944272/0.0000000
```

or for large graphs

```
Iter   : 37
A/H[ 0]=0.0000000/0.0000002
A/H[ 1]=0.0000001/0.0000238
A/H[ 2]=0.0000002/1.0000000
A/H[ 3]=0.0000159/0.0000000
 ...
```

Deliverables. Include source code of all implemented functions or classes in an archive per Document 4 guidelines. Document bugs; no bug report no partial points.