1 Mini-Project Logistics

Rule 1. Read carefully and observe Handout 2 guidelines/requirements. WE URGE THAT YOU START WORKING ON COMMAND LINE and FILE-BASED INPUT/OUTPUT AS SOON AS POSSIBLE. ERRORS AT SUCH LEVEL CAUSE SEVERAL SUBMISSIONS TO RECEIVE 0 POINTS BECAUSE THEY MAKE ALTERNATE TESTING DIFFICULT. You submit ONE archive named `mp_WXYZ.tar` or `mp_WXYZ.zip` and up to two options of the 3 available. Each option is worth 120 points and the combination of the two can earn you no more than 160 points . Observe naming conventions (Handout 2). Thus there should be no `huffman.java` file but only a `huffman_4567.java` if the last 4 digits of your NJIT ID are indeed `4567` and you are expected to implement a class `huffman`. Check my.njit.edu for your ID! Files with `.c .cc .cpp .java .txt .tar .zip` suffixes are acceptable. Anything else will be **REJECTED**.

Rule 2. Email submission to three addresses per Handout 2 guidelines. Read Handout 2 before you proceed below. Make sure that the the three words of the subject line are as in Handout 2.

2a. Grader's Email: `Check the course web-page under Email(Assistant)`
2b. Instructor's Email: `alexg+cs435@njit.edu`
2c. Yourself: CC the email to yourself (DO NOT COMPLAIN otherwise).

Rule 3. Emailed message must be received (per Handout 2 guidelines)

BEFORE NOON-time of 19 April 2017

For penalties check Handout 1 (Syllabus). If the NJIT email system breaks down that day (eg power outage), we will not grant any extensions; submit it the day before.

Rule 4. Submissions that deviate from Rules 1-3/Handout 2 get 0 points.

You may do up to two of the 3 options. You may utilize the same language or not.

OPTION 1 (Heap and Data compression related). Do the programming related to Huffman coding in C, C++, or Java.

OPTION 2 (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 used by Google around 1997-1998.

OPTION 3 (Graph and numerical related). Do the programming related to the implementation of Google's PageRank algorithm in C, C++, or Java.

2 OPTION 1: Huffman Coding (120 points)

This programming option can be implemented in C, C++ or Java. We will try to give a description of the requirements in a programming language independent way.

It is an assignment that requires the use of heap-related structures and operations as defined in the textbook (CLRS); of course a binary heap array in the textbook starts from 1 but in C/C++/Java starts from 0. Some adjustments are thus necessary. YOU MUST IMPLEMENT YOUR OWN heap/priority queue using a binary heap. No java supplied functions should be used for heap, priority queue operations or otherwise (eg HashMap).

If the naming conventions below are not facilitated or accommodated by the programming language you plan to use, you are free to modify the names of the various functions. If this happens, however, you MUST provide sufficient information in the file indicated in Handout 2.

Huffman coding. Says all. Input argument `file` in the command-line is an arbitrary file-name. You need to provide a correct Huffman coding implementation that implements the algorithm described in class and also available in the notes or the textbook. Such an algorithm must work not just in text files but also binary files (pdf, images, video, etc). For small files do not get bothered if savings due to compression are small or negligible. In general for files less than 10MiB, an implementation should take more than a few seconds, may be 15 seconds forgivingly

Note that `henc` for Huffman encoding, or `hdec` for Huffman decoding should adhere to the Handout 2 requirements (i.e. they are more likely to be `henc_4567` or `hdec_4567`.

```
  //  converts file into file.huf myfile.pdf into myfile.pdf.huf and so on

% java      henc file
% ./henc    file
% java      henc myfile.pdf
% ./henc    myfile.pdf

  //  converts file.huf into file , myfile.pdf.huf into myfile.pdf and so on

% java      hdec file.huf
% ./hdec    myfile.pdf.huf

 // Operation henc is NON destructive! myfile.pdf would coexist with myfile.pdf.huf
 // Operation hdec is destructive!     file created from file.huf
 //                                    would ovewrite a previous file

 Test Files Example
% ./henc h1435s17.pdf
% ./henc nytimes_google3.pdf
```

Note 1: Reminder. Document your bugs.

Note 2: Deliverables. Include all implemented functions or classes in an archive per Handout 2 guidelines. Command-line execution: do not prompt to read a file-name. Everything command-line based.

3 OPTION 2: Hashing (120 points)

This component may be implemented in Java, C, or C++. We are asking you to implement a Lexicon structure maintained by Google in 1997-1998 to store words (aka arbitrarily long strings of characters) in main memory extracted from a large collection of documents. This lexicon L used a Hash Table T structure along with an Array A of NULL separated strings .

In our case the 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 $h(k, i) = (h'(k) + i^2) \bmod m$. 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 mod m, where m is the slot-size of the hash table. Thus 'alex' (the string is between the quotation marks) is mapped to $97 + 108 + 101 + 120 \bmod m$ whatever m is. In the example below, for $m = 11$, $h(\mathbf{alex}, 0) = 8$.

Table T however won't store key values k in it. This is because the keys are strings of arbitrary length. Instead, T will store pointers/references to another array A. Furthermore pointers/references are programming-language dependent: we will just use integer indexes to a second array named A.

The second table, array A will be a character array and will store the words maintained in T separated by null values `\0`. A null, is one character not a two character string consisting of a back-slash and the digit zero; it is a zero-bit filled word of the appropriate size (1B for ASCII; 2B for Unicode nowadays). If you don't know what B is, it is a byte; never use b for a bit, write instead `bit` or `bits`.

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 need to increase the size of A accordingly. Doubling it, is a wise choice. Likewise the size of T might also have to be increased. This causes more problems that you 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**.

However you need to support three more operations: **Print** , **Create** and **Cleanup**. (Moreover, the implementation probably will use a check for an empty or full table/array and a mechanism to perform operation in batch: a HashBatch function/method.)
The former 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.) Function **Create** creates T, A and initializes them. The number of slots of T would be m. Allocate for A size $15m$ characters and initialize A to spaces.

The following is a minimal interface maintained. We call the class that supports and realizes A and T a lexicon: L is one instance of a lexicon.

```
HashCreate (lexicon L, int m);   // Create T, A. T will have m slots; A should be 8m
HashEmpty  (lexicon L);          // Check  if L is empty
HashFull   (lexicon L);          // Check  if L can maintain more words
HashPrint  (lexicon L);          // Print    of L
HashInsert (lexicon L, word  w); //Insert w  into L (and T and A)
HashDelete (lexicon L, word  w); //Delete  w from L (but not necessarily from A)
HashSearch (lexicon L, word  w); //Search for string in L (and this means T)
HashBatch  (lexicon L, file filename)
```

The testing will be performed through `HashBatch`. It uses as an argument a filename where several operations will be listed and executed in batch.

Operation 10 is **Insert**, Operation 11 is **Deletion**, and Operation 12 is **Search**. Operation 13 is **Print**, Operation 14 is **Create**. (Create accepts as its second parameter and that of HashCreate, an integer value next to its code 14; this becomes m.) The HashBatch accepts an arbitrary filename such as command.txt or file.txt that contains a sequence of commands. Instances are shown in the example below.

```
% java mplexicon  command.txt
% ./mplexicon     file.txt
```

Thus

```
14 11
10 alex
10 tom
10 jerry
13
```

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 for 'alex' mod 11 give an 8, but for 'tom' and 'jerry' give 6, 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:
 6: 5
 7: 9
 8: 0
 9:
10:
```

A subsequent

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

will generate on screen

```
 alex   found at slot 8
 tom    found at slot 6
 jerry  found at slot 7
 mary   not found
 tom    deleted from slot 6
```

and the following will be printed

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

Note. In both `Print` operations we intentionally left blank what happens with the other slots and also slot 6 after the deletion of tom. It's up to you to decide the meaning of empty.

Deliverables. Include all implemented functions or classes (no .class files) in an archive per Handout 2 guidelines.

4 OPTION 3: Google's PageRank (120 points)

Implement the Google PageRank algorithm as explained below. The input for this problem would be a graph represented through an adjacency list representation. The command-line interface that would be used is as follows The first two of the three parameters hold integer values; the last parameter is a filename. (This is the variant that will be implemented i.e. the first two lines of invocation.) You need to implement class or function `pagerank` (in fact `pagerank_4567` or whatever Handout 2 dictates). (The other variant, i.e. the lines using oerrorate are implicit in the variant to be implemented.)

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

The PageRank algorithm is iterative. At iteration t all pagerank values are computed using results from iteration $t-1$. The `initialvalue` helps us to start this process. Moreover, in the PageRank computation, a parameter d would be set to 0.85. The PageRank of vertex A depends on the PageRanks of vertices $T_1, \ldots, T_m$ incident to A, i.e. pointing to A. The contribution of T_i to the PageRank of A would be the PageRank of T_i i.e. PR(T_i) divided by $C(T_i)$), where $C(T_i)$ is the out-degree of vertex T_i.

$$PR(A) = (1-d)/n + d\,(PR(T_1)/C(T_1) + \ldots + PR(T_m)/C(T_m))$$

When we compute ranks (or PageRanks) iteratively we use the previous iteration values to update the current iteration values! Thus PR(A) is the value to be obtained in the current iteration t, but all PR (T_i) values are from the previous iteration $t-1$. This is called a synchronized update. (In an asynchronous update, we use whatever we have!) Be careful! Be synchronized!

In order to run the 'algorithm' we either run it for a fixed number of iterations and `iterations` determines that, or for a fixed `errorate` (that is going to be 10^{-4}) when `iterations` is 0. We know (theory-wise) that PageRank should "converge" within 60-70 iterations; if not we can increase from the command-line `iterations`. Alternatively instead of controlling the iterations, we control the error-rate. Thus `errorrate` can be a single negative digit such as $-2, -3, \ldots, -6$. In such an approach at the end of iteration t when all PageRanks for t have been computed we compare for every vertex these values to the ones for iteration $t-1$. If the difference is less than $10^{\textbf{errorrate}}$ for EVERY VERTEX, we can stop: we have achieved convergence to the desired error-rate. We make this option easier for you: (a) if `iterations` is an integer greater than zero you run pagerank for that number of iterations, (b) if `iterations` is equal to zero, you run for as many iterations needed to achieve the FIXED errorrate of 10^{-4}.

The second parameter `initialvalue` shows the initial values for the ranks. If it is `0` all ranks 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 $1/\sqrt{N}$, where N is the number of web-pages (vertices of the graph). (In order to determine N you need to construct the graph described in file `filename` first.)

The third parameter `filename` describes the input (directed) graph and it has the following form. The first line contains two numbers: the number of vertices (in the example below, this is equal to four and is denoted by the first four) and the number of edges that follow on separate lines (the second four in the example). In each line an edge (i, j) is represented by `i j`. Thus our graph has (directed) edges $(0,2), (0,3), (1,0), (2,1)$.

Pageranks are printed to six decimal digits.

If $N > 10$ then the values for `iterations, initialvalue` are to be `0` and `-1` respectively. In such a case the pageranks at the stopping iteration 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).

```
% ./pagerank  15 -1 samplegraph.txt
Base   :  0 :P[ 0]=0.250000 P[ 1]=0.250000 P[ 2]=0.250000 P[ 3]=0.250000
Iter   :  1 :P[ 0]=0.250000 P[ 1]=0.250000 P[ 2]=0.143750 P[ 3]=0.143750
Iter   :  2 :P[ 0]=0.250000 P[ 1]=0.159687 P[ 2]=0.143750 P[ 3]=0.143750
Iter   :  3 :P[ 0]=0.173234 P[ 1]=0.159687 P[ 2]=0.143750 P[ 3]=0.143750
Iter   :  4 :P[ 0]=0.173234 P[ 1]=0.159687 P[ 2]=0.111125 P[ 3]=0.111125
Iter   :  5 :P[ 0]=0.173234 P[ 1]=0.131956 P[ 2]=0.111125 P[ 3]=0.111125
Iter   :  6 :P[ 0]=0.149663 P[ 1]=0.131956 P[ 2]=0.111125 P[ 3]=0.111125
Iter   :  7 :P[ 0]=0.149663 P[ 1]=0.131956 P[ 2]=0.101107 P[ 3]=0.101107
Iter   :  8 :P[ 0]=0.149663 P[ 1]=0.123441 P[ 2]=0.101107 P[ 3]=0.101107
Iter   :  9 :P[ 0]=0.142425 P[ 1]=0.123441 P[ 2]=0.101107 P[ 3]=0.101107
Iter   : 10 :P[ 0]=0.142425 P[ 1]=0.123441 P[ 2]=0.098030 P[ 3]=0.098030
Iter   : 11 :P[ 0]=0.142425 P[ 1]=0.120826 P[ 2]=0.098030 P[ 3]=0.098030
Iter   : 12 :P[ 0]=0.140202 P[ 1]=0.120826 P[ 2]=0.098030 P[ 3]=0.098030
Iter   : 13 :P[ 0]=0.140202 P[ 1]=0.120826 P[ 2]=0.097086 P[ 3]=0.097086
Iter   : 14 :P[ 0]=0.140202 P[ 1]=0.120023 P[ 2]=0.097086 P[ 3]=0.097086
Iter   : 15 :P[ 0]=0.139520 P[ 1]=0.120023 P[ 2]=0.097086 P[ 3]=0.097086

% ./pagerank  0 -1 samplegraph.txt
Base   :  0 :P[ 0]=0.250000 P[ 1]=0.250000 P[ 2]=0.250000 P[ 3]=0.250000
Iter   :  1 :P[ 0]=0.250000 P[ 1]=0.250000 P[ 2]=0.143750 P[ 3]=0.143750
Iter   :  2 :P[ 0]=0.250000 P[ 1]=0.159687 P[ 2]=0.143750 P[ 3]=0.143750
Iter   :  3 :P[ 0]=0.173234 P[ 1]=0.159687 P[ 2]=0.143750 P[ 3]=0.143750
Iter   :  4 :P[ 0]=0.173234 P[ 1]=0.159687 P[ 2]=0.111125 P[ 3]=0.111125
Iter   :  5 :P[ 0]=0.173234 P[ 1]=0.131956 P[ 2]=0.111125 P[ 3]=0.111125
Iter   :  6 :P[ 0]=0.149663 P[ 1]=0.131956 P[ 2]=0.111125 P[ 3]=0.111125
Iter   :  7 :P[ 0]=0.149663 P[ 1]=0.131956 P[ 2]=0.101107 P[ 3]=0.101107
Iter   :  8 :P[ 0]=0.149663 P[ 1]=0.123441 P[ 2]=0.101107 P[ 3]=0.101107
Iter   :  9 :P[ 0]=0.142425 P[ 1]=0.123441 P[ 2]=0.101107 P[ 3]=0.101107
Iter   : 10 :P[ 0]=0.142425 P[ 1]=0.123441 P[ 2]=0.098030 P[ 3]=0.098030
Iter   : 11 :P[ 0]=0.142425 P[ 1]=0.120826 P[ 2]=0.098030 P[ 3]=0.098030
Iter   : 12 :P[ 0]=0.140202 P[ 1]=0.120826 P[ 2]=0.098030 P[ 3]=0.098030
Iter   : 13 :P[ 0]=0.140202 P[ 1]=0.120826 P[ 2]=0.097086 P[ 3]=0.097086
Iter   : 14 :P[ 0]=0.140202 P[ 1]=0.120023 P[ 2]=0.097086 P[ 3]=0.097086
Iter   : 15 :P[ 0]=0.139520 P[ 1]=0.120023 P[ 2]=0.097086 P[ 3]=0.097086
Iter   : 16 :P[ 0]=0.139520 P[ 1]=0.120023 P[ 2]=0.096796 P[ 3]=0.096796
Iter   : 17 :P[ 0]=0.139520 P[ 1]=0.119776 P[ 2]=0.096796 P[ 3]=0.096796
Iter   : 18 :P[ 0]=0.139310 P[ 1]=0.119776 P[ 2]=0.096796 P[ 3]=0.096796
Iter   : 19 :P[ 0]=0.139310 P[ 1]=0.119776 P[ 2]=0.096707 P[ 3]=0.096707

% ./pagerank  0  -1 verylargegraph.txt
Iter   : 3
P[ 0] = 0.021429
P[ 1] = 0.030536
P[ 2] = 0.027500
   ...
   ...
   other vertices omitted
   ....
```

Deliverables. Include all implemented functions or classes in an archive per Handout 2 guidelines. Document bugs; no bug report no partial points.