1 Mini-Project 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 Handout 2 and follow all the requirements specified there; moreover, observe naming conventions, comments and testing as specified in sections 2-4 of Handout 2.

STEP-2. When the archive per Handout 2 guidelines is ready, upload it to moodle and do not forget to press the submit button; the submission timestamp depends on this (submission button).

BEFORE NOON-time of Wed 24 April, 2019

If you want to get 0 pts, ignore Handout 2.

You may do up to two of the 3 options. 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 (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)

Implement Huffman coding to compress and decompress arbitrary files (text-based or binary files such as a .pdf or .jpg or .mp4.) as explained in the Subject notes. You must implement your own Binary Heap and thus PriorityQueue as explained in class. Reminder: No HashMaps. Non-compliance to these will automatically gain you a zero. (See Handout 2.) WE SHALL REVIEW YOUR CODE TO VERIFY ADHERENCE TO THESE GUIDELINES. Suggestion: view byte values as 'characters' in a generic way independent of file type. Do not expect much compression from files that are small or already compressed (eg .jpg). In general for files of size about 10MiB, an implementation should take no more than a few seconds, may be 15 seconds forgivingly. Moreover, you will be writing bit aligned codes into files: practice with bit operations and packing or unpacking bits into/from bytes. If instead of writing bit 1 you write character 1 (in ASCII or UNICODE) this defeats the compression objective. Avoid getting distracted with the Wikipedia code. Moreover your code should print on screen (standard output) at the end of the compression two numbers: one is the number of bytes of the compressed file, the other number is the total number of bits utilized for the encoding of the original file. That latter number is the sum of products; each product is the frequency of a bytevalue (aka character) and the number of Huffman bits used for its encoding.

Input argument eg `filename` or `filename.pdf` in the command-line is an arbitrary file-name with or without a suffix.

```
 // Encode : henc  encodes filename  into  filename.huf  ; filename     gets erased
 //          thus x.pdf generates x.pdf.huf and then x.pdf gets erased
 // Decode : hdec  decodes filename.huf into    filename ; filename.huf gets erased
 //          if  filename already exists, it gets overwritten
 //          thus x.pdf.huf generates x.pdf and then x.pdf.huf gets erased
 // Note   : Per Handout 2  you should use   hencWXYZ or hdec_WXYZ for henc and hdec
% java henc filename.pdf        // output on stdout suppressed
% java hdec filename.pdf.huf
% ./henc    filename.pdf        // output on stdout suppressed
% ./hdec    filename.pdf.huf
  //   EXAMPLE testing steps
% cp p1435s19.pdf  filename     // copy into filename
% java henc filename            // Huffman encode filename into filename.huf
% java hdec filename.huf        // Huffman decode filename.huf that is still there
% md5sum p1435s19.pdf filename  // Should have same signatures
  //   EXAMPLE C or C++
% cp p1435s19.pdf  filename
% ./henc filename
% ./hdec filename.huf
% md5sum p1435s19.pdf filename
```

Deliverables. Include all implemented functions (source code only) into an archive per Handout 2 guidelines. Command-line execution: do not prompt to read a file-name.

3 OPTION 2: Hashing (120 points)

We are asking you to implement a Lexicon structure realized by Google in 1997-1998 to store words (aka arbitrarily long strings of characters) in main memory. Lexicon L uses a Hash Table T structure along with an Array A of NULL separated strings . 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 $h(k,i) = (h'(k) + i^2)$ mod 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 $97+108+101+120$ mod m whatever m is. In the example below, for $m = 11$, $h(\textbf{alex}, 0) = 8$. 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 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; 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, even if T remains the same. Doubling it, is an option. Likewise the size of T might also have to be increased. This causes more 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.) Function **Create** creates T with m slots, and A with $15m$ chars and initializes A to spaces. The following is a suggested minimal interface (we try to be language agnostic). 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 15m
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)
```

Testing utilizes `HashBatch`. Its argument filename, an arbitrary filename contains several operations that are executed in batch mode. Operation 10 is **Insert**, Operation 11 is **Deletion**, and Operation 12 is **Search**. Operation 13 is **Print**, Operation 14 is **Create**. Operation 15 is **Comment**; the rest of the line is ignored. (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 operations.

```
% 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 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:
```

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

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

Deliverables. 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 (source code only) in an archive per Handout 2 guidelines. Document bugs; no bug report no partial credit.