

(c) Copyright A. Gerbessiotis. All rights reserved.

Posting this document on the Web, other than its original location at NJIT's LMS, is strictly prohibited unless there is an explicit written authorization by its author (name on top line of this first page of this document).

1 Programming Project (PrP) overview

Your submission will support

- algorithmic-related programming,
- command-line processing support, and
- file-based input and 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. The function names shown in the following pages are just for guidance. They help us organize the narrative and assist you in organizing your project. They are programming language agnostic and thus might feel irrelevant or foreign to you all. Feel free to ignore them but do not complain afterwards.

STEP-1. Read carefully Document 3. It saves you time later.

STEP-2. When the archive per Document 3 guidelines is ready, upload it and submit it to canvas through Canvas Assignments

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

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.

Hash Table Organization: A lexicon of (English character) words. 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. The implementation should be optimized enough for a test execution to take no more than few seconds, maximum 5 seconds.

2 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 4 mod N , where N is the number of slots of T . Thus 'alex' (the string is between the quotation marks) is mapped to $97 + 108 + 101 + 120 - 4 \bmod N$ whatever N is. In the example below, for $N = 11$, $h(\text{alex}, 0) = 4$. 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 $\backslash 0$. 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 $\backslash 0$ 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 the option that you will use. 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 already 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 $\backslash 0$ 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 could thus implement 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 (word). If you try to build an interactive system your submission will get 0 pts. Input can only be read from a file as specified.

(The assumption below is that filearbitrary.txt and file.txt contain the same six lines containing instances of commands 14,10,15 and 13.)

```
% java prplexicon1234 filearbitrary.txt
% ./prp123 file.txt
14 11
10 alex
10 tom
10 jerry
15 ready-to-print
13
```

The six-line batch file above will silently execute commands 14, 10, 15. Command 13 will print the following on standard output (screen): T and A. The *T* entries of 0, 5, 9 are the indexes (first position) for alex, tom, jerry of *A* respectively. Note that the ASCII values of 'alex' hash into a 4, but for 'tom' and 'jerry' both hash into a 2, i.e. a collision occurs. A minimal output for Print is available below. The backslash after the x or m or y in *A* indicate a NUL byte (8 bit that are zero). The formal ASCII representation is \ 0 but we need to use one character for a NUL byte and thus \ is shown.

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

If the following lines were added to file filearbitrary.txt or file.txt

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

they will generate in addition on standard output (i.e. 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 3 guidelines.