IE 172 Laboratory 6: Hashing Out Hash Tables II

Dr. T.K. Ralphs

Due March 24, 2016

1 Laboratory Description and Procedures

1.1 Learning Objectives

You should be able to do the following after completing this laboratory.

1. Understand how to use command-line options and arguments.
2. Understand how to interactively communicate with your script.

1.2 Key Words

You should be able to define the following key words after completing this laboratory.

1. Command-line options and arguments
2. Interactive script

1.3 Scenario

Being able to start and interact with a program/script from a command line is an extremely important part of any programming/scripting language. In this lab, we will experiment with command line arguments and interactive environment for Python language. This lab builds on lab 5 by extending the use of dictionary

The scenario you can envision initially is the following. You are a linguist doing analysis of various classic books. You have these books stored in plain text form and you want do things like determine in what books a specific word/phrase is used, and other such things.

1.4 Program Specifications

In Lab 5 you have been provided with a full implementation of a hash table class implemented using both chaining and linear probing. Your job will be to use a hash table class of your choice and build an interface to communicate with your script to choose different modus operandi.

1.4.1 Algorithms

The algorithms to be implemented in this lab are various hash functions and methods of resolving collisions supporting the operations on a hash table specified by the interface in `hashTable.h`.

1.4.2 Data Structures

The basic data structure required for this laboratory is a hash table.

2 Laboratory Test Files

The main files for this laboratory are `dictionary_oa.py` (the open addressing implementation), `dictionary_chain.py` (the chaining implementation). In addition, you should choose a few favorite classic book that are out of copyright and find a plain text copy of it on-line. Project Gutenberg is a good place to browse. The books should have at least 10K words.

3 Laboratory Assignments

3.1 Programming and Analysis (20 points)

1. (5 points) Implement a command line argument parser. Your script should be able to interpret the input arguments and perform the following functionality:

   (a) `my_script.py --add x y` : prints out x+y

   (b) `my_script.py --sub x y` : prints out x-y

   (c) `my_script.py --file file.txt --word x` : if word x exists in file.txt, return number of occurrences, otherwise return string "Not found"

   (d) `my_script.py --help` : print out all commands and description thereof used by the script

   Make sure that if the command line argument (or a parameter) is not recognized the script returns the help list of all commands and description thereof used by the script. You may find the Python package `argparse` useful.

2. (10 points) Implement the script that will process a list of "books" given as the command line argument and will enable you to interactively search for words/phrases within the books. Your script should do something similar to this:

```
my_script.py -books file1.txt file2.txt ... fileN.txt
>Input the word:  something
>Word "something" found in the following books:
    file7
    file22
    file43
    ...
>Input the word:  something2
>Word "something2" found in the following books:
    file2
>Input the word:  something3
>Word "something3" not found in any book.
>Input the word:....
```

3. (5 points) Improve (2) by sorting the list of "books" in the descending order with respect to number of occurrences of the word.

3.2 Follow-up Questions (60 points)

1. Consider the following sorting algorithm.

```
def fsort(A, beg = None, end = None):
    if beg == None:
        beg = 0
    if end == None:
        end = len(A) - 1
    if beg >= end:
        return
    if A[beg] > A[end]:
        A[beg], A[end] = A[end], A[beg]
    fsort(A, beg+1, end-1)
    if A[beg] > A[beg+1]:
        A[beg], A[beg+1] = A[beg+1], A[beg]
    fsort(A, beg+1, end)
```

   (a) (10 points) Show that this sorting algorithm is correct (hint: use induction).

   (b) (10 points) Write a recurrence for the running time of this algorithm. Is this algorithm efficient?

2. Suppose you have an array `A` of n distinct integers in sorted order. Consider the problem of finding an index `i` such that `A[i] = i` or proving that none exists.

   (a) (10 points) Suppose the integers could be positive, zero, or negative. Describe and analyze an efficient algorithm for solving this problem (you will only get full credit if you come up with something better than the "naive" algorithm). You must justify (at least informally) that your algorithm is correct.

   (b) (10 points) Now suppose that the integers must be non-negative. Describe an even faster algorithm for solving the problem in this case and prove that this is the fastest possible algorithm

3. Consider sorting a list of numbers using only two operations: `compare` and `flip`. The operation `flip(i)` reverses the order of the first i elements of the list. In other words, if we start with the list

$$[1\ 4\ 6\ 5\ 3\ 10\ 1], \tag{1}$$

   then applying the operation `flip(4)` results in the list

$$[5\ 6\ 4\ 1\ 3\ 10\ 1]. \tag{2}$$

   (a) (10 points) Write pseudo code for an algorithm for sorting a list using only the operations `compare` and flip. There is more than one solution to this. You should justify (at least informally) that your algorithm is correct.

   (b) (10 points) How many flip operations are required for your algorithm in the worst case?

   (c) (5 points) What is the running time of the flip operation itself as a function of the argument `i`?

(d) (5 points) Derive the overall worst-case running time of your sorting algorithm, taking into account the running time of each flip operation and also the number of comparison operations required.