

Project 4: Hashing and Sorting

CSCI 2720 Data Structures (Spring 2009)

Due Thursday April 23, 2009

This programming project is to implement either a hashing function or a quicksort algorithm. The project has the following requirements.

1 Hashing Function

This is to implement a hashing function to facilitate the lookup of student IDs from a storage table. The number of students is limited to 1,000. You are allowed to use a table of length 1,100. You will choose/define your own hashing function. You will also need to choose a strategy to resolve collisions, which could be either a chaining strategy or an open addressing strategy. The goal is to make the average search time as small as possible.

1.1 Populating the ID table

Populating the ID table needs to repeatedly use at least three functions *ID generation*, *lookup*, and *insertion*. 1,000 IDs, positive integers ranging from 100,000 to 200,000, are randomly generated and inserted into the table (of length 1,100). Every ID is inserted to the location defined by the hashing function and the collision strategy that you choose to implement.

1.2 Testing the performance of your program

For any integer number N given by the user, your program will randomly generate N number of IDs (within the range between 100,000 and 200,000) and lookup them from the table. Note that, to make the test accurate, the number N may be much larger than 1,000, which the total number of IDs. That is, some IDs may be searched more than once, some IDs may not be searched, and some IDs may not exist in the table. You need to output the average time for successful searches and average time for unsuccessful searches. To do this, you need to embed some instructions in the program to count the total numbers of comparisons (with IDs) used in successful and unsuccessful searches, respectively. Normalize these two numbers by N should give the desired outputs respectively.

2 Quicksort

This is to implement the recursive quicksort algorithm using two different strategies for pivot finding/partitioning. You will need to implement both the "tricky but fast" and the "simple but slower" partition functions. The latter may need an additional array for storage.

2.1 Generating random lists

For any given number N , $N \leq 100,000$, your program will generate a list of N random integers (within the range of 0 to $N - 1$, some possibly identical).

2.2 Testing the performance of the programs

For each integer numbers $N = 1,000, 2,000, \dots, 10,000, 11,000, \dots, 100,000$, run your program on 50 different lists of the same length N and compute the average times for the quicksort using the two different partition functions respectively. These times for sorting are real times. You will need to embed some system calls within the two programs to record the real times used in sorting. Output the average times, which is the total time used by the 50 runs normalized by 50. Plot a curve for each of the average times for different values of N . Refer to Project 0 for more information on generating random lists and plotting results.

3 Language

You may use either C++ or Java language for the implementation. Make sure your programs contain sufficient documentations and indentations for format.

4 Other requirements

In the submission folder, include a "readme" file to describe your hashing function and your collision resolving strategy if you choose to the hashing function project. Otherwise, include in the "readme" file the algorithms for the two partition functions.

5 Submission

You need to submit the program named either `Hashing` or `QuickSort` electronically to the cs2720a account on `odin` machine.