**20CS2028 Data Structures**

**Programming assignment #6**

**Assigned on Tuesday, July 21, 2015**

**Due on Thursday, July 30, 2015 on Blackboard at 11:59PM**

**Total number of points: 50**

**Purpose:** The purpose of this programming project is to gain experience with (1) hash functions, (2) hash table implementation, and (3) evaluate the performance of different collision-resolution methods.

**Part 1: Building and testing your programs.**

For the purpose of this assignment we assume that each data item is a key (for real applications, each such item should contain a key and other useful information).

Implement a hash table using an array that contains 23 (this value must be prime) keys.

For a hash table of size N, the hash function is **h1(key)= key % N**. Use appropriate, user-friendly prompts to have the user input the **load ratio** (e.g., 20%, or 0.2) for the hash table.

Also, allow the user to input (i.e., insert) the numbers to the hash table. Your program stops part 1 when the load ratio is exceeded.

Use **linear probing** as the collision-resolution scheme.

Finally, allow the user to print the hash table. Note that this step is used to allow yourself and TA to check the result for your implementation.

Replace the collision-resolution scheme using each of the following methods:

(a) Double hashing using **h2(key)=q-(key mod q)** (for example, you could choose q = 20)

(b) Quadratic probing.

(c) Chaining within the hash table.

Fully test each of the collision-resolution schemes.

*Input to your main program*: Select collision resolution scheme (a) to (c) any other value will mean quit. Load ratio: 0 to 1.0

Your program will then fill the table with random numbers until the load ratio has been reached or exceeded. If a number is inserted again, ignore it and print out a message to that effect.

*Output*: a print out of the hash table.

To test your program the user must then be able to enter a key and the program has to report if that number is in the hash table or not.

**Part 2: Performance Evaluation by random numbers.**

Use your program from part 1 but now use a larger hash table say 1009 or another prime number.

Then, based on the same hash function and each collision-resolution method described above, use a random number generator to generate integer values and place them into the hash table.

For each random number insertion, count the number of the **comparisons** required to place the number at its final location. If a new random number exists already in the hash table count the number of comparisons to find it but do not insert it again. Sum up all comparisons to a variable *TotalComparisons*.

By dividing *TotalComparisons* by *TotalInsertions* (i.e., the number of values inserted into the hash table), you get the value of *AverageComparisons*.

Record the *Average\_Comparisons* for load ratios 0.1, 0.15, up to 0.85 in increments of 5%. Then plot the *Average\_Comparisons* against the *Load\_Ratio* for the three collision-resolution methods in one graph. Note that each experiment (for a specific collision-resolution method) must use the **same** random number sequence. This is very important in comparing their performances.

Try to generate **different** random number sequences to observe if the performance of each collision-resolution method will change. **To avoid any statistical biasing, it is highly suggested that you use different sets of random numbers to generate 3 or more sets of data and then plot the average of these data.** Write a short discussion about what you conclude to be the best collision-resolution method. Include all these discussions in a text file called **discussion**.

*Input*: Random numbers (be careful not to generate the same sequence – need to use srand to set the seed).

*Outputs:* The average number of comparisons for different load ratios, for each collision-resolution method. Make it a table. Write whatever you observe in the discussion file.

**Part 3: Table Size Selection**

In 1970, C. E. Radke’s study (The use of quadratic residue research) suggested that a good choice is a table size that is a prime number of the form **4m+3**. So, change your table size in Part 2 to **811=(4x202)+3**, and repeat Part 2. Discuss the difference in performance for both Parts 2 and 3 in the discussion file. (Note that 4m+3 = 2(2m+1) +1. 2m+1 is always odd, so if it is also prime, then 4m+3 is actually a Germaine prime. But the point that Radke makes in his study is that 4m+3 can be selected close to an integral power of 2 (i.e. 2n).