Data Structures (810:052) Lab 12 - Hashing Name:_____

Objective: To experiment with searching and get a feel for the performance of hashing.

Download the following file to your desktop: http://www.cs.uni.edu/~fienup/cs052s10/labs/lab12.zip
Extract this file to the Desktop by right-clicking on lab12.zip icon and selecting Extract All.

Part A: The `lab12.zip` file you downloaded and extracted contains a `binarySearch` folder with a Visual Studio C++ project file: `binarySearch.sln` inside. Double-click on it to open this project in Visual Studio.

a) The `main.cpp` file starts by timing the binarySearch of an array of a user specified size. Try 20,000,000 elements, but don't input the commas. If you look at the code, it creates a list, evenList, that holds 20,000,000 sorted, even values (e.g., evenList = [0, 2, 4, 6, 8, ..., 39999996, 39999998]). It then times the searching for target values from 0, 1, 2, 3, 4, ..., 39999998, 39999999 so half of the searches are successful and half are unsuccessful. How long does it take to binary search for target values from 0, 1, 2, 3, 4, ..., 39999998, 39999999?

b) After the binary searches, the `main` adds the 20,000,000 even values (i.e., 0, 2, 4, 6, 8, ..., 39999996, 39999998) to a HashMap with 20,000,000 slots (i.e., buckets), so the load factor is 1.0. The HashMap implementation uses chaining with linked-lists for buckets. Its then times the searching for target values from 0, 1, 2, 3, 4, ..., 39999998, 39999999 so half of the searches are successful and half are unsuccessful. How long does it take to search for target values from 0, 1, 2, 3, 4, ..., 39999998, 39999999 in the HashMap?

c) Explain why searching of the HashMap is faster than binary searching.

d) After timing the 40,000,000 searches of the HashMap with a load factor of 1, the `main` repeats the experiment by doubling the load factor of the HashMap between 0.2 and 25.6. Completing the following table:

	Load Factor							
	0.2	0.4	0.8	1.6	3.2	6.4	12.8	25.6
Execution time with 20,000,000 even items in hash table (seconds)								
Hash table size								

e) Why does the performance of the HashMap get worse as the load factor increases?

f) Why does the performance of the HashMap degrade slowly as the load factor increases?

After you have completed the above table and answered the questions, raise your hand.

Part B: The `lab12.zip` file you downloaded and extracted contains a `HashMapOpenAddr` folder with a Visual Studio C++ project file: `HashMapOpenAddr.sln` inside. Double-click on it to open this project.

a) The `main.cpp` file starts with an interactive test program which creates a small (10 slot) open-address hash table using linear probing for rehashing. The *identity* function is passed as the hash function, i.e., the key value is returned by the hash function. Thus, the home address is the key % (hash table size). Perform the following steps:

- Repeatedly (p)ut the keys of: 2, 12, 13, 3, 22 use the (d)isplay command after each put to view the hash table
- Check to see if the hash table co(n)tains the keys 22 and 100
- (R)emove key 12 and then (d)isplay the hash table
- Check to see if the hash table co(n)tains the keys 12 and 22
- Predict where would key 32 be added?
- (P)ut key 32 and then (d)isplay to check your prediction

b) Edit the `main` function by commenting out the first line of the file and uncommenting the second line to get a small (16 slot) open-address hash table using quadratic probing for rehashing. The *identity* function is still passed as the hash function. The table size is automatically justed up to make it a power of 2, and the **CORRECTED** quadratic probing rehash formula used is:

check slot at = (home address + ((rehash attempt #)^2 +(rehash attempt #)) **/ 2**) % (hash table size)

Does this hash table size and quadratic probing rehash formula seem to give complete non-repetitive coverage of the hash table slots?

c) After you (q)uit the interact menu, the `main` function adds the 100,000 even values (i.e., 0, 2, 4, 6, 8, ..., 199996, 199998) to myHashTable of size 500,000 (i.e., load factor of 0.2) using linear probing for rehashing. It times the searching for target values from 0, 1, 2, 3, 4, ..., 199998, 199999 so half of the searches are successful and half are unsuccessful. It then repeats this experiment by changing the load factor of the HashTable between 0.2 and 0. Completing the following table:

	Load Factor							
	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Execution time with 100,000 items in hash table (seconds)								
Hash Table Size								

d) Explain why the performance of the hash table with linear probing degrades so badly at high load factors.

e) Edit the non-interactive part of the `main` function by switching the type of rehashing from linear probing to quadratic probing for rehashing. Repeat the experiment using quadratic probing for load factors between 0.2 and 0.9. Completing the following table:

	Load Factor							
	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Execution time with 100,000 items in hash table using quadratic probing (seconds)								
Hash Table Size								
Actual Load Factor								

f) Explain why quadratic probing performs better than linear probing.

After you have completed the above tables and answered the questions, raise your hand.

EXTRA CREDIT:

a) Edit the main program to experiment with performance as the number of items in the hash table grows, but load factor remains fixed at 0.2. Completing the following table:

	Items in the Hash Table			
	10,000	100,000	1,000,000	10,000,000
Execution time with with load factor 0.2 (seconds) using linear probing				
Execution time with with load factor 0.2 (seconds) using quadratic probing				

b) What type of growth rate did you observe as the number of items grew?

c) Explain why you would expect this growth rate after studying the HashTable code.