CIS 291 – Data Structures in C++ - Spring 2005
Week 5 Lab Exercise
Week 5 Lab Exercise due: Tuesday, February 15th, END of lab

__Purpose__: thinking/experimentation related to hashing

Note: a calculator could be handy here – remember the desktop calculator program if you do not have a "physical" calculator handy...

Answer the following questions on a piece of paper individually. Then, compare and discuss your answers with at least one other class member. Then, write your name on the "Next:" list to get your work checked over. As always, to receive credit for this lab exercise, your work must be completed by the end of the lab period.

NOTE that this time, you will lose points for incorrect answers --- I want you to be especially careful in your checking with one another, and to really __discuss__ any differences until you have resolved them.

1. The following represents a **hash table** implemented using **open addressing** and **linear probing**. Its **table_size** is 13 (as you can see) and its hash function is simply:

`hash(int key) -> key % table_size` **(it's good enough for Savitch and Main, __Data Structures and Other Objects using C++__)**

0	1	2	3	4	5	6	7	8	9	10	11	12

"Fill in" the above hash table appropriately, inserting the following items (in the order shown):

988, 350, 367, 168, 694, 182, 820, 202, 644, 422

(Note: I generated the above using a pseudo-random-number generator, asking for values in the range [0, 1000), I chose 10 values because that will give this hash table a load factor of 77%. I was curious how this would work... 8-))

hash(988) __________

hash(350) __________

hash(367) __________

hash(168) __________

hash(694) __________

hash(182) __________

hash(820) __________

hash(202) __________

hash(644) __________

hash(422) __________

Do we see clustering, above? ______________

Now, try to **retrieve** each value. How many values did you need to search, **including** the desired value once found? (That is, give the actual number of table elements examined in each successful search... 8-))

988 ________

350 ________

367 ________

168 ________

694 ________

182 ________

820 ________

202 ________

644 ________

422 ________

Amongst these 10 values for these 10 searches, then --- what was the **average** number of table elements examined in these successful searches?

____

2. The following represents a **hash table** implemented using **open addressing** and **double hashing**. It's **table_size** is 13 (as you can see) and its hash functions (also from Savitch and Main, Data Structures and Other Objects using C++) are:

```
hash1(int key) -> key % table_size

hash2(int key) -> 1 + (key % (table_size - 2))
```

(note: 11, 13 ARE twin primes)

0	1	2	3	4	5	6	7	8	9	10	11	12

"Fill in" the above hash table appropriately, again inserting the following items (in the order shown). (Be careful --- remember that, in double-hashing, you only call hash2 if hash1 leads to a collision --- and then hash2 is providing how much to add to the current "collided" index. ASK ME if this is not clear to you.)

988, 350, 367, 168, 694, 182, 820, 202, 644, 422

(note: below, you only need to fill in hash2 if you NEED it. Put a dash or X for hash2 if you do NOT need it.)

hash1(988) __________

hash2(988) __________

hash1(350) __________

hash2(350) __________

hash1(367) __________

hash2(367) __________

hash1(168) __________

hash2(168) __________

hash1(694) __________

hash2(694) __________

hash1(182) __________

hash2(182) __________

hash1(820) __________

hash2(820) __________

hash1(202) __________

hash2(202) __________

hash1(644) __________

hash2(644) __________

hash1(422) __________

hash2(422) __________

Now, try to **retrieve** each value. How many values did you need to search, **including** the desired value once found? (That is, give the actual number of table elements examined in each successful search... 8-))

988 ________

350 ________

367 ________

168 ________

694 ________

182 ________

820 ________

202 ________

644 ________

422 ________

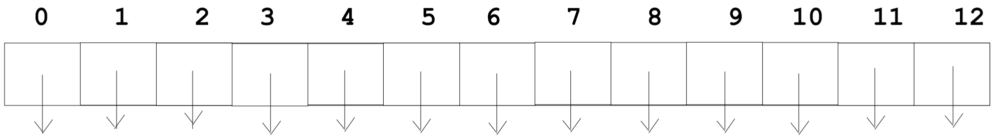
Amongst these 10 values for these 10 searches, then --- what was the **average** number of table elements examined in these successful searches?

____

3. Okay, can't you guess what this one is going to ask? 8-)

The following represents a **hash table** implemented using **buckets and chaining**. It's **table_size** is 13 (as you can see) and its hash function is still:

```
hash(int key) -> key % table_size
```



"Fill in" the above hash table appropriately, again inserting the following items (in the order shown).

988, 350, 367, 168, 694, 182, 820, 202, 644, 422

Because you will not be able to answer the question below without it, note the following **IMPORTANT assumption**:

* although we haven't formally discussed them yet, we are assuming here that simple singly-**linked lists** are being used for the chains. In a simple singly-linked list, it is more efficient to insert at the **beginning** of the list (as we will discuss later).

THUS, you need to insert at the BEGINNING of a bucket's list EACH time you add an element --- and, thus, using a PENCIL for this question would be a GOOD idea. It is inconvenient for pencil and paper, but it is a better simulation of how it is actually done.

hash(988) __________

hash(350) __________

hash(367) __________

hash(168) __________

hash(694) __________

hash(182) __________

hash(820) __________

hash(202) __________

hash(644) __________

hash(422) __________

Now, try to **retrieve** each value. How many values did you need to search, **including** the desired value once found? (That is, give the actual number of table elements examined in each successful search... 8-))

988 ________

350 ________

367 ________

168 ________

694 ________

182 ________

820 ________

202 ________

644 ________

422 ________

Amongst these 10 values for these 10 searches, then --- what was the **average** number of table elements examined in these successful searches?

____