CS 132 - Intro to Computer Science II - Spring 2005
WEEK #13 LAB EXERCISE and Homework #10

Week #13 Lab Exercise due: Wednesday, April 20th, END of lab
HW #10 due: Wednesday, April 27th, 1:00 pm

Purpose: thinking/experience with hashing and hash tables

WEEK #13 LAB EXERCISE

Answer the following questions on paper; after coming up with your initial answers, you may discuss them with another student before getting your answers checked, if you wish. When you are ready, put your name on the "Next:" list on the board so that your answers can be checked. [Note: check **carefully**! There will be **points docked** for errors, this time, to encourage you to double-check carefully *before* getting your work checked.]

[Note: a calculator may be handy here! There is one available on the NHW 244 computers, in a pinch, and I *think* that there's an option to "expand" it into a "scientific" calculator...]

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...!)**

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

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

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(694) __________ hash(202) __________

hash(350) __________ hash(182) __________ hash(644) __________

hash(367) __________ hash(820) __________ hash(422) __________

hash(168) __________

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 ________ 694 ________ 202 ________

350 ________ 182 ________ 644 ________

367 ________ 820 ________ 422 ________

168 ________

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) 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)	__________	hash1(694)	__________	hash1(202)	__________
hash2(988)	__________	hash2(694)	__________	hash2(202)	__________
hash1(350)	__________	hash1(182)	__________	hash1(644)	__________
hash2(350)	__________	hash2(182)	__________	hash2(644)	__________
hash1(367)	__________	hash1(820)	__________	hash1(422)	__________
hash2(367)	__________	hash2(820)	__________	hash2(422)	__________
hash1(168)	__________				
hash2(168)	__________				

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	________	694	________	202	________
350	________	182	________	644	________
367	________	820	________	422	________
168	________				

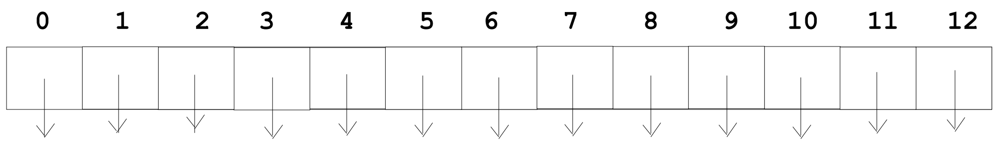
Amongst these 10 values for these 10 searches, then --- what was the **average** number of table elements examined in these successful searches?

____

3. And, finally...

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

ASSUMPTION: new elements are added to the HEAD of a bucket's linked list. [Yes, it is annoying in a pencil-and-paper situation, but it is also how this is often done...] Your results must reflect this assumption.

hash(988) __________ hash(694) __________ hash(202) __________

hash(350) __________ hash(182) __________ hash(644) __________

hash(367) __________ hash(820) __________ hash(422) __________

hash(168) __________

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 ________ 694 ________ 202 ________

350 ________ 182 ________ 644 ________

367 ________ 820 ________ 422 ________

168 ________

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

____

<u>HOMEWORK #10:</u>

1. There are a number of files posted on the public course web page along with this handout. Copy them into your new directory on cs-server that is going to hold this homework's files.

You should now have files named **node.h**, **node.template**, **hashtable.h**, **hashtable.template**, **stock_item.h**, and **test_stock_item.cpp**.

The **stock_item** files contain the definition for a very simple class representing a stock item. Read it over and get comfortable with it --- and notice that it meets the criteria for a **RecordType**, suitable for being stored in a **hashtable** instance.

BUT --- where is its **.cpp**file? It is so simple, everything is done in-line within **stock_item.h**!

And so, when you compile **test_stock_item,cpp**, you don't have to include **stock_item.cpp** (which is good, since it is nonexistent.) Compile it, and show me you've successfully tested it by running:

```
test_stock_item > 132hw10_1_out
```

...and submit your 132hw10_1_out.

2. Now, I've provided you with a **hashtable.h** that declares a template class **hashtable** able to hold **RecordType**'s, where a RecordType is expected to have a member function **get_key( )** that returns an int, assumed to be a unique key for a record instance. I have also provided **node.h** and **node.template** (which now does include the linked-list toolkit functions described in the course text).

Yes, **hashtable** is to be a buckets-and-chaining hash table, where the buckets are indeed linked lists of **node<RecordType>**.

Now, I've also given you the **beginning** of **hashtable.template** --- you are to finish "filling it in". (Look for YOU FILL IN... 8-)) I'm not promising an absence of typos, either --- start early, and e-mail me about "existing" code oddnesses, if you notice any.

3. You, of course, need to test your hash table implementation a bit.

Write **try_hashtable1.cpp** that:

* asks a user to enter in as many stock keys, names, quants, and prices as they wish --- you insert each into a hashtable.
* then shows the user the resulting hashtable contents (using **print_hashtable**)
* then allows them to enter keys for as many stock items as they wish, and says for each if it is in stock, and if so its details.
* then asks them if they wish to remove any stock items --- if so, it lets them enter as many stock item keys as they wish, removing each corresponding stock_item.
* it then again allows them to enter keys for as many stock items as they wish, and says for each if it is in stock, and if so its details.
* it then shows the user the resulting hashtable contents (using **print_hashtable**)

Then, write **try_hashtable2.cpp** that hard-codes in several actions for each of the above categories, but with no user input, so you can generate an output file...! (`try_hashtable2 > 132hw10_2_out`)

Turn in **try_hashtable1.cpp**, **try_hashtable2.cpp**, **hashtable.template**, and **132hw10_2_out**.

And, when you are satisfied with all of the above, submit them using **~st10/132submit** on cs-server:

132hw10_1_out
hashtable.template
try_hashtable1.cpp, try_hashtable2.cpp, 132hw10_2_out