Assignment 7 – Hashing

This document is based on: Active Learning Approach to Data Structures using C++ by Dr. Timothy Budd

Due: Nov 11, 2014, 8:00 AM

Late penalties will be as described in the syllabus.
Points may be deducted for failure to follow these instructions.

General Objectives:

Learn how to use and apply Hashing Methods
Demonstrate ability to think and communicate logically

Graded Part of the Assignment:

1. Read all the background material
2. Answer all questions on the green pages and as otherwise indicated
3. Type your name into the header blank (above right corner)

What to turn in:

Save a copy of this Word Document named: **A07_Hashing_*yourlastname*.docx**
Be certain the items noted above are completed.
Submit the completed document to the appropriate D2L dropbox.

Background Material:

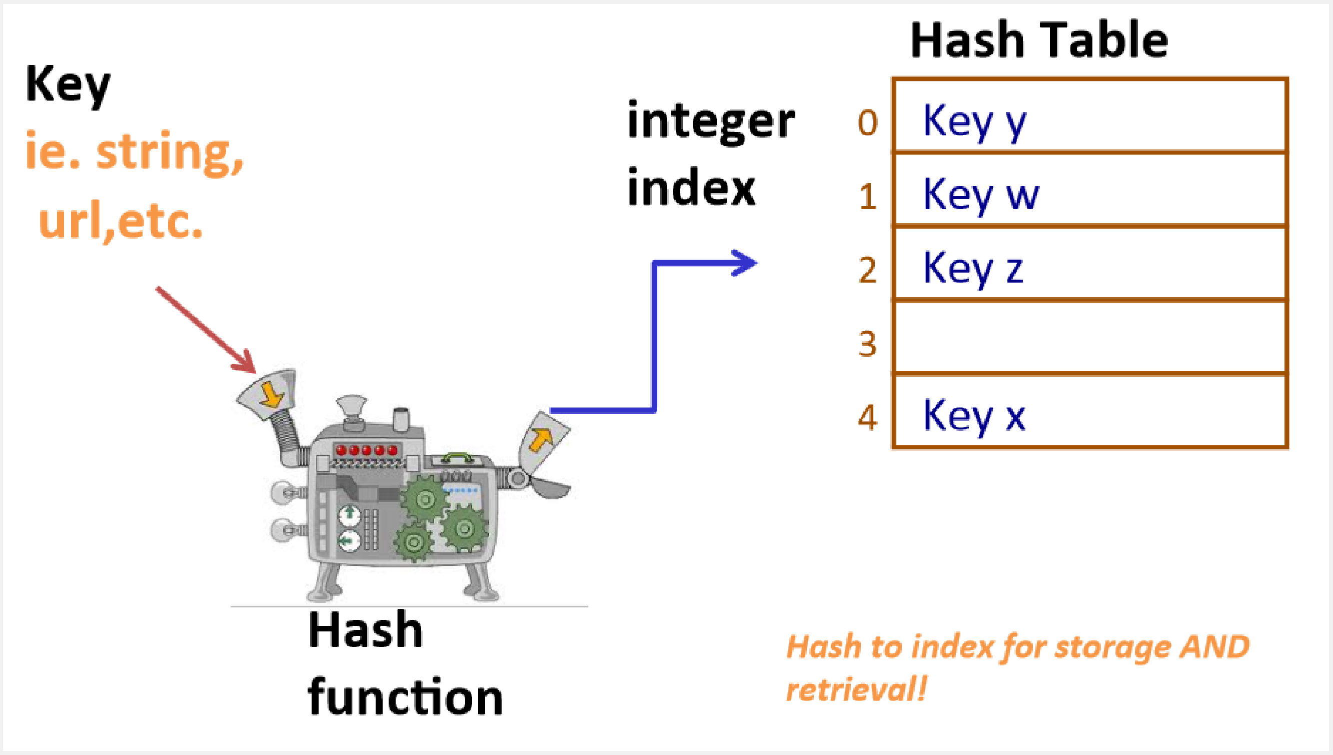
Hash Tables (Open Address Hashing – Linear Probing)

Consider the following story. Six friends; Alfred, Alessia, Amina, Amy, Andy and Anne, have a club. Amy is in charge of writing a program to do bookkeeping. Dues are paid each time a member attends a meeting, but not all members attend all meetings. To help with the programming Amy uses a six-element array to store the amount each member has paid in dues.

Amy uses an interesting fact. If she selects the **third letter of each name**, treating the letter as a number from 0 to 25, and then divides the number by 6, each name yields a different number. So in O(1) time Amy can change a name into an integer index value, then use this value to index into a table. This is faster than an ordered data structure, indeed almost as fast as a subscript calculation.

Al**f**red	F = 5 % 6 = 5
Al**e**ssia	E = 4 % 6 = 4
Am**i**na	I = 8 % 6 = 2
Am**y**	Y = 24 % 6 = 0
An**d**y	D = 3 % 6 = 3
An**n**e	N = 13 % 6 = 1

What Amy has discovered is called a *perfect hash function* for her current set of names. Specifically no two names map to the same index, thus no collisions occur. In general, a *hash function* is a function that takes as input an element and returns an integer value. Almost always the index used by a hash algorithm is the remainder after dividing this value by the hash table size. So, for example, Amy's hash function returns values from 0 to 25. She divided by the table size (i.e. 6) in order to get an index.



The idea of *hashing* can be used to create a variety of different data structures. Of course, Amy's "perfect" system falls apart when the set of names is different. Suppose Alan wishes to join the club. Amy's calculation for Alan will yield 0, the same value as Amy. Two values that have the same hash are said to have *collided*. The way in which collisions are handled is what separates different hash table techniques. Almost any process that converts a value into an integer can be used as a hash function. Strings can interpret characters as integers (as in Amy's club), doubles can use a portion of their numeric value, objects can use one or more fields.

The first technique you will explore is termed *open-address hashing*. Here all elements are stored in a single large table. Positions that are not yet filled are given a **null** value. An eight-element table using Amy's algorithm would look like the following:

0 = aiqy	1 = bjrz	2 = cks	3 = dlt	4 = emu	5 = fnv	6 = gow	7 = hpx
Amina			Andy	Alessia	Alfred		Aspen

Notice that the table size is different, and so the index values are also different (i.e. mod 8 was used instead of mod 6). The letters at the top show characters that hash into the indicated locations. If Anne now joins the club, we will find that the hash value (namely, 5) is the same as for Alfred. So to find a location to store the value Anne, we ***probe linearly*** for the next free location. This means to simply move forward, position by position, until an empty location is found. In this example the next free location is at position 6. This results in the following:

0 = aiqy	1 = bjrz	2 = cks	3 = dlt	4 = emu	5 = fnv	6 = gow	7 = hpx
Amina			Andy	Alessia	Alfred	Anne	Aspen

Now suppose Agnes wishes to join the club. Her hash value, 5, is already filled. The probe moves forward to the next position, and then the next, and when the end of the array is reached it continues with the first element, eventually finding position 1:

0 = aiqy	1 = bjrz	2 = cks	3 = dlt	4 = emu	5 = fnv	6 = gow	7 = hpx
Amina	Agnes		Andy	Alessia	Alfred	Anne	Aspen

Finally, suppose Alan wishes to join the club. He finds that his hash location, 0, is filled by Amina. The next free location is not until position 2:

0 = aiqy	1 = bjrz	2 = cks	3 = dlt	4 = emu	5 = fnv	6 = gow	7 = hpx
Amina	Agnes	Alan	Andy	Alessia	Alfred	Anne	Aspen

We now have as many elements as can fit into this table. The ratio of the number of elements to the table size is known as the *load factor*, **written** λ. For open address hashing the load factor is never larger than 1. Just as a Vector was doubled in size when necessary, a common solution to a full hash table is to move all values into a new and larger table when the load factor becomes larger than some threshold, such as 0.85. To do so a new table is created, and every entry in the old table is rehashed this time dividing by the new table size to find the index to place into the new table.

Searching for values and related actions

To see if a value is contained in a hash table the test value is first hashed. But just because the value is not found at the given location doesn't mean that it is not in the table. Think about searching the table above for the value Alan, for example. Instead, an unsuccessful test must continue to probe, moving forward until either the value is found or an empty location is encountered.

Removing an element from an open hash table is problematic. We cannot simply replace the location with a null entry, as this might interfere with subsequent search operations. Imagine that we replaced Agnes with a null value in the table given above, and then once more performed a search for Alan. What would happen?

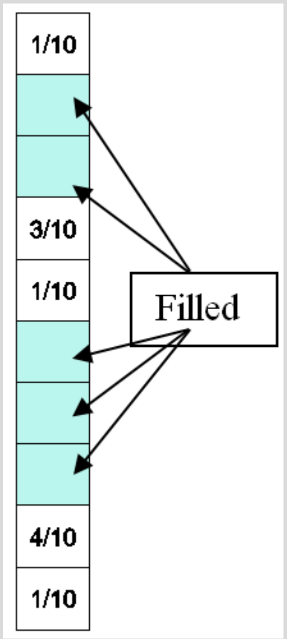
One solution to this problem is to not allow removals. The second solution is to create a special type of marker termed a ***tombstone***. A tombstone replaces a deleted value, can be replaced by another newly inserted value, but does not halt the search.

How fast are hash table operations? The analysis depends upon several factors. We assume that the time it takes to compute the hash value is constant. But what about distribution of the integers returned by the hash function? It would be perfectly legal for a hash function to always return the value zero → legal, but not very useful.

The best case occurs when the hash function returns values that are uniformly distributed among all possible index values; that is, for any input value each index is equally likely. In this situation one can show that the number of elements that will be examined in performing an addition, removal or test will be roughly $1/(1 - \lambda)$. For a small load factor this is acceptable, but degrades quickly as the load factor increases. This is why hash tables typically increase the size of the table if the load factor becomes too large.

λ	$(1/(1 - \lambda))$
0.25	1.3
0.5	2.0
0.6	2.5
0.75	4.0
0.85	6.6
0.95	19.0

Imagine that the **colored squares in the ten-element table at right indicate values in a hash table that have already been filled**. Now assume that the next value will, with equal probability, be any of the ten values. What is the probability that each of the free squares will be filled? Since both positions 1 and 2 are filled, any value that maps into these locations must go into the next free location, which if 3. So the probability that square 3 will be filled in 3/10, while the probability that square 0 will be filled is only 1/10. This phenomenon, where the larger a block of filled cells becomes, the more likely it is to become even larger, is known as clustering. (A similar phenomenon explains why groups of cars on a freeway tend to become larger). Clustering is just one reason why it is important to keep the load factor of hash tables low. Simply moving to the next free location is known as linear probing.



Open Address Hashing – Questions to Answer

1. What does it mean to "hash" a value?

2. What is a hash function?

3. What is a perfect hash function?

4. What does it mean when two key values collide?

5. What does it mean to (linearly) probe for a free location in an open address hash table?

6. Why is it "bad" when the load factor becomes too large?

7. Using (linear) probing hashing, fill in the below hash table with a hash function of: mod 7 (i.e. *x* % 7)
with the following elements: 14, 22, 33, 3, 21, 55

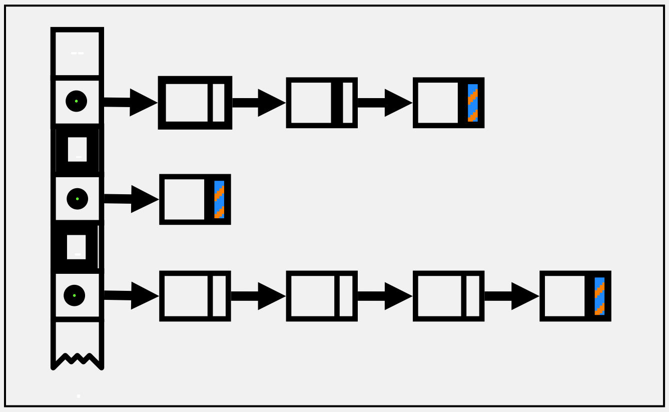
0	1	2	3	4	5	6

COMPLETE THIS PAGE in the document you turn in to D2L --- indicate your answers above

Hash Tables using Buckets

Some call this type of method **chaining**.

In the previous lesson you learned about the concept of hashing, and how it was used in an open address hash table (with linear probing). In this lesson you will explore a different approach to dealing with collisions, the idea of hash tables using chaining. A hash table that uses chaining is really a combination of an array and a linked list. Each element in the array (the hash table) is a head pointer for a linked list. All elements that hash into the same location (same bucket) will be stored in the list. This is illustrated in the below image:



There are 2 basic steps to putting an element into the table. The first is to use the hash function to get the index into the table (i.e. what bucket does the element go to). Assuming the bucket at that index is empty then the element is added as the first element in the bucket's list. If there is already an element at that index, the element to be added is just added onto the bucket's list. This results in a chaining of elements for each index (i.e. collisions mean the bucket at the indicated index gets more stuff in it).

Notice the major behavior difference between chaining and probing is what happens when a collision occurs.

As before, the load factor (λ) is defined as the number of elements divided by the table size. In this structure the load factor can be larger than one, and represents the average number of elements stored in each list, assuming that the hash function distributes elements uniformly over all positions. Since the running time of the Find() and Delete() functions are both proportional to the length of the list, they are considered to be O(λ). Therefore the execution time for chaining-based hash tables is fast only if the load factor remains small. A typical technique is to resize the table (doubling the size, as with the vector and the open address, probing-based, hash table) if the load factor becomes larger than 10.

Name: ___________________________

1. What is a bucket (with regard to chaining based hashing)?

2. How are chaining based hash tables similar to those used in open address hashing? How are they different?

3. What is the definition of the load factor for a chaining based hash table?

4. Assuming that the chaining based hash function in use distributes the elements evenly over all buckets, what is another interpretation of the load factor?

5. Explain how the chaining based hash table combines features of an array and a linked list.

COMPLETE THIS PAGE in the document you turn in to D2L --- indicate your answers above