Activity #5: Hashing
Recorder's Report

Manager: Reader:

Recorder: Driver:

Date: Score: Satisfactory / Not Satisfactory

Record your team's answers to the key questions (marked with ) below.

a) Model 1, Question #3

b) Model 2, Question #11

c) Model 2, Question #12

d) Model 4, Question #20

e) Model 4, Question #21

Activity #5: Hashing

Recall the List ADT. Its operations include insert, find, and delete. Earlier, we looked at how find can be implemented using binary search over a sorted list. One advantage of binary search is that it only cost $O(log_2 N)$ time to execute.
One common tradeoff in computer science is the time/space tradeoff. In some (maybe many) cases, one can increase runtime speed at the expense of storage space. On the flip side, one can often gain efficiency in storage at the expense of running time.
We'll look at a way to implement the Dictionary ADT using hash tables. As you might guess, we will speed up the dictionary operations at the expense of using more storage space.

Content Learning Objectives

After completing this activity, students should be able to:

- Explain a hash function
- Explain the structure of hash table
- Explain how collisions are managed

Process Skill Goals

During the activity, students should make progress toward:

- Write code that adds, removes, and accesses a hash table

Preston Carman derived this work from Tammy VanDeGrift work found at https://www.dropbox.com/sh/rl0yyth9g06psva/AADB0Cj4isIX5DAyrspqj8mFa and continues to be licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Model 1 Dictionary ADT

Questions (10 min) Start time:

1. Think about how you use a dictionary. What do you "search for" in a dictionary?

2. What information to you get back from a dictionary?

A dictionary ADT stores a collection of items and supports the following operations:

```
create()                              // create new Dictionary
insert(Dictionary d, key k, value v) // insert a new value with given key
find(Dictionary d, key k)       // search for a given key and return the value
delete(Dictionary d, key k)     // delete key and item that goes with the key
print(Dictionary d)             // print all items in dictionary

```

One common tradeoff in computer science is the time/space tradeoff. In some (maybe many) cases, one can increase runtime speed at the expense of storage space. On the flip side, one can often gain efficiency in storage at the expense of running time.

We'll look at a way to implement the Dictionary ADT using hash tables. As you might guess, we will speed up the dictionary operations at the expense of using more storage space.

Consider this scenario. Your dictionary should store (student ID, student record) information. Student IDs are non-negative numbers. Your dictionary stores only currently enrolled WWU students. Suppose the range of possible student IDs goes from 0 to 999,999,999.

3. How might you build a Dictionary ADT for this scenario so that insert, find, and delete are O(1) operations?

4. How much memory does your solution in #3 take?

Consider that there are about 4000 currently enrolled WWU students. If you implement an array of size 1,000,000,000 to store (student ID, student record) pairs, the array is sparse. Certainly, the insert, delete, and find operations are O(1) with this implementation, but the wasted space is quite large given the number of items in the dictionary.

Here is one way to reduce the size of the array. If we know there are 4000 items in the dictionary, let's instead create an array of size 100,000 to store these items. Note that there are still 96,000 empty cells with this approach, but that is quite a bit smaller than 1 billion.

So, in order to insert a (student ID, student record) pair into the dictionary of size 100,000, we need to map student IDs to the range [0...99,999]. One common way to do this is to use the mod operator:

```
location = studentID % 100000;

```

The mod (remainder) operator gives us values in the appropriate range.

5. What is your student ID?

6. In what array location would your ID be inserted, assuming the array has 100,000 items?

7. Consider student ID 1152436. In what array location would this ID be inserted?

8. What might be a downside to this approach?

Because we have limited the array size, it is now possible to have collisions. A collision happens when two dictionary items are mapped to the same location in the array.

9. What is another student ID number that would be mapped to the same array location as 1152436?

Model 2 Hash Functions

Questions (10 min) Start time:

Above, we have considered the situation where the items are numbers to insert into the dictionary. Suppose now we want to insert strings into the dictionary (like an actual dictionary!!). We first need to map the string to a number. In this case, the string is our key and the number is our hash value. Once we have the value, we need to map the value to a location in the array.

The mapping from a key to an integer is called the hash function. There are many implementations of hash functions. A good hash function spreads keys across the range of integers. A good hash function is fast to compute, given the length of the key. A really bad hash function is one that maps all keys to the value 6. You will learn more about creating good hash functions in CS 324, Algorithms, if you take that course. Another property of a good hash function is that it maps the strings "abc" and "cba" to different integers, so permutations of the same set of letters have different hash values.

Another property of a good hash function is that if two keys are considered equal, they will map to the same hash value. For example, if uppercase "SAM" and lowercase "sam" are considered equivalent in your application, then they should map to the same hash value.

Here is an example of a hash function:

```
unsigned int hash(std::string key) {
  unsigned int rtnVal = 3253;
  for (int i = 0; i < key.size(); i++) {
    rtnVal *= 28277;
    rtnVal += key[i] * 2749;
  }
  return rtnVal;
}

```

So, hash("a") is:

```
rtnVal = 3253
rtnVal = 3253 * 28277 // 91985081
rtnVal = 91985081 + 97*2749 // note: 'a' in ASCII is 97
rtnVal = 92251734

```

10. What is hash("ab")?

11. What is hash("ba")? (Note: 'b' in ASCII is value 98)

12. Explain one reason a Dictionary ADT might use a hash function?

Model 3 Hash Tables

Questions (10 min) Start time:

The array that was mentioned earlier is called a hash table. A hash table stores dictionary items. The storage location is based on the hash value and using % of the hash table size.

`Insertion`

Assume we have a hash table of size 10. We want to insert the following items:

- "coconut"
- "milk"
- "apple"

STEP 1: We'll use the hash function listed above to calculate the hash values for these keys:

- "coconut" (hash value = 2104178476)
- "milk" (hash value = 461110994)
- "apple" (hash value = 3515030035)

STEP 2: We'll map the hash values to the size for this hash table (% 10):

- "coconut" (location = 6)
- "milk" (location = 4)
- "apple" (location = 5)

13. Now, suppose we want to insert "orange" into this hash table. Its hash value is 3410197053. Insert this key into the table above.

14. Now, let's insert "corn". Its hash value is 1347376851. Insert this key into the table above.

15. Now, let's insert "eggs". Its hash value is 881505635. Insert this key into the table above.

What happens now? "apple" is already stored at position 5. There are several techniques (see textbook) to address collisions.

We'll first resolve collisions using open address linear probing.

Model 4 Open Address Linear Probing

Questions (20 min) Start time:

When a collision occurs, the linear probing technique finds the first unoccupied cell in the hash table to insert the item. In this case, "eggs" cannot go into position 5, cannot go into position 6, but position 7 is open. We put "eggs" into position 7 in the table above. Do that now.

One drawback of linear probing is that the runtime for insertion can now go to O(N) instead of O(1) for a hash table of size N. A second drawback of linear probing is that it produces primary clustering. When keys hash to the same value, the sequence of looking for open spots is the same. Suppose two new keys hash to location 5; both inserts would follow the same process of looking at position 5, 6, 7, 8 (and position 9 for the second key). Suppose a third key hashes to location 5; now this would try to insert the key into position 5, 6, 7, 8, 9. It would then wrap-around to looking at position 0 and insert it there since position 0 is empty.

But, the upside is that linear probing is simple to implement. Note that most hash table implementations use more sophisticated collision resolution techniques, which we will see later.

Delete

We have just examined the insert operation using linear probing. Now, let's consider the delete operation. Suppose the hash table contains the following keys:

	"corn"		"orange"	"milk"	"apple"	"coconut"	"eggs"	"potato"	"salad"
0	1	2	3	4	5	6	7	8	9

Now, we want to delete the key "potato". Suppose we just remove it by setting the position to NULL (empty box):

	"corn"		"orange"	"milk"	"apple"	"coconut"	"eggs"		"salad"
0	1	2	3	4	5	6	7	8	9

Now, let's suppose I want to delete "salad". Its hash value is 125082698. So, its location is 8. We look at position 8 in the table. It's null, so we conclude that "salad" is not in the table. Uh – that's not quite right. "salad" is in the table. We just didn't find it.

So, we need to do something a little more sophisticated to delete items. Table entries can store keys, NULL (empty), or a special symbol representing that the item was deleted. So, we'll use "D" as a special symbol to denote that an item was deleted. Suppose we delete "potato", as in the above example. This time, though, we put "D" in that position.

	"corn"		"orange"	"milk"	"apple"	"coconut"	"eggs"	"D"	"salad"
0	1	2	3	4	5	6	7	8	9

Now, when we go to delete "salad", we first look in position 8. We see that it has "D". So, we use linear probing to continue looking for the item "salad". We look at position 9. There it is, so we can set that position to "D".

	"corn"		"orange"	"milk"	"apple"	"coconut"	"eggs"	"D"	"D"
0	1	2	3	4	5	6	7	8	9

Now, suppose we want to delete "beans". Its hash location is 4. We look up T[4]. It has "milk", so we go to T[5]. It has "apple". We go to T[6]. It has "coconut". We go to T[7]. It has "eggs". We go to T[8]. It has "D". We go to T[9]. It has "D". We go to T[0] (wrap-around). It has NULL, so we conclude that "beans" is not in the dictionary.

16. Delete the key "eggs". Recall that this key maps to location 5. Update the table above. What strings are compared during the delete operation?

17. Now, insert the key "mango". This key maps to location 5. Update the table above. What location does "mango" go into? What strings are compared during the insert?

Note that a dictionary holds just one copy of each key. If we try to insert "milk" into the hash table, it would look at location 4 and see that it already has "milk" and not update the hash table.

Finding

Searching for items in a dictionary is similar to the insertion and deletion processes. We need to hash the key, gets its hash value, and map it to the hash table location. We look at that location. If it is empty, we return -1. If it has what we are looking for, we return the location (or the actual key/value contents in the dictionary). Else, we go to the next position and continue this process.

```
1  // D = special delete symbol
2  int Find(HashTable& H, string key) {
3      unsigned int value = hash(key);
4      int location = value % H.size;
5      int origLocation = location;
6      while (H[location] != empty) {
7          if (key == H[location]) {
8              return location;
9          }
10         // note: H[location] does not equal key or H[location] is D
11         location++;
12         location = location % H.size;
```

```
13          if (location == origLocation) {
14              return -1; // gone through entire hash table
15          }
16      }
17      return -1;
18  }
19
```

Suppose the hash table has:

	"corn"		"orange"	"milk"	"apple"	"coconut"	"mango"	"D"	"D"
0	1	2	3	4	5	6	7	8	9

18. Find "corn". This hashes to the location 1. How many string comparisons are done? Which strings are they ones compared?

19. Find "mango". This hashes to the location 5. How many string comparisons are done? Which strings are they ones compared?

20. Find "eggs". This hashes to the location 5. How many string comparisons are done? Which strings are they ones compared?

Chaining is another major technique to resolve collisions in a hash table. Instead of each hash table cell containing a single dictionary item, we instead have a pointer to a linked list of dictionary items.

21. Does your group have questions about hash functions and/or hash tables?