Contents

Lab 11

Intro to Dictionaries

Group work: Group work is allowed for the labs, but each person must do their own coding and each person must turn in an assignment. Copying other peoples' code / code files is not allowed. Copied assignments will receive 0% for everybody involved.

Assignments must build: Assignments that don't build will automatically just be given a flat 50% score. I may still give feedback on what went wrong, and 50% is better than 0%, so turn in something - but ideally, make sure it builds.

Paper assignments: Type out your answers or print and write answers. Upload lab digitally (scanned/photographed).

11.1 Key-value pairs

In a normal array, elements are stored in the array according to position, so the key is the index in the array.

Index (Key)	`0`	`1`	`2`	`3`	`4`	`5`
Value	Cat	Dog	Mouse	Fish	Cake	Bug

For a dictionary (aka table, aka map), a key can be created as any data-type, so that you're able to look for one set of information using that key.

For example, perhaps you're storing a list of students. Instead of 0, 1, 2, etc. as the keys to access each student, we store students by their StudentIDs so we can easily access their data without having to search the entire array. This is the idea behind a Dictionary.

Key (Employee ID)	T1068	Q213	Z1328
Value	Sana	Riyann	Xing

A Dictionary can be implemented on top of several different data structures: Arrays, linked lists, or even trees. Implementations can differ in efficiency for different types of functions.

A dictionary might need functionality like:

- Get amount of items
- Get item by its key
- Add new items (the value) with some key
- Does a certain key exist in the dictionary?
- Remove an item by key
- Clear the entire dictionary

Because elements of a dictionary need to have a key and a value, we will usually implement a special class for the data, similar to how we use Nodes in Linked Lists.

```
1 template <typename KT, typename VT>
2 struct Element
3 {
4     KT key;
5     VT value;
6 };
```

But to maximize the efficiency of a Dictionary, usually you will make a Dictionary as an array type...

```
1 template <typename KT, typename VT>
2 class Dictionary
3 {
4     private:
5     Element<KT,VT>* m_array;
6 };
```

Just like our other data structures, we will need functions to **add new items**, **remove items**, and **find items**.

```
1  template <typename KT, typename VT>
2  class Dictionary
3  {
4      public:
5      void Push( KT key, VT value );   // Add
6      void RemoveItem( KT key );       // Remove
7      VT& GetItem( KT key );           // Access
8
9      // Utilities
10     int GetSize();
11     bool Exists( KT key );
12     void Clear();
13
14     private:
15     Element<KT,VT>* m_array;
16     int m_itemCount;
17 };
```

Now the question is... how do we associate the **key** (which can be any data type) with the array's **index** (0, 1, 2, ...)?

11.2 Hash Functions

For a dictionary, there is usually some sort of **hashing strategy** to mathematically convert the **key** into an array **index**.

For example, if the keys in a dictionary were `'A'`, `'B'`, `'C'`, `'D'`, we could create a function that maps a letter to a number like this:

Key	`A`	`B`	`C`	`D`	...
Index	0	1	2	3	...

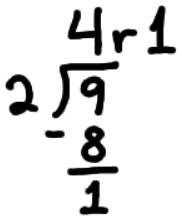
But what about integers that are bigger than the array size, such as arbitrary employee IDs?

Key	1054	513	789	6889	...
Index	?	?	?	?	...

How do we convert *any key value* into an index from `0` to `size-1`?

The simplest way to write a hash function is to use **modulus**. Modulus is how we can get the *remainder* from the result of a division. Modulus also restricts the number values, as we cannot equal or exceed the number we're dividing by.

If we were solving a division problem by hand...



The whole-number quotient of $9 \div 2$ is 4, and the remainder, 9%2 is 1. Here, % stands for the modulus operator.

So let's see how modulus affects a list of numbers...:

- index `i` mod 3...

Index `i`	0	1	2	3	4	5	6
`i % 3`	0	1	2	0	1	2	0
$i \div 3$	0 r**0**	0 r**1**	0 r**2**	1 r**0**	1 r**1**	1 r**2**	2 r**0**

- index `i` mod 4...

Index `i`	0	1	2	3	4	5	6
`i % 4`	0	1	2	3	0	1	2
$i \div 4$	0 r**0**	0 r**1**	0 r**2**	0 r**3**	1 r**0**	1 r**1**	1 r**2**

- index `i` mod 5...

Index `i`	0	1	2	3	4	5	6
`i % 5`	0	1	2	3	4	0	1
$i \div 5$	0 r**0**	0 r**1**	0 r**2**	0 r**3**	0 r**4**	1 r**0**	1 r**1**

Notice that for each of these, 0 is the lower bound of values, and $n - 1$ is the upper-bound (for taking i mod n). Once it hits the maximum value, it wraps back around to 0.

With a dictionary, we can simply use the table size in our modulus equation.

```
1  // Simple hash function
2  int GetIndex( int key )
3  {
4      return ( key % m_tableSize );
5  }
6
7  // Add Item
8  int Push( int key, string data )
9  {
10     int index = GetIndex( key );
11     m_array[index] = data;
12 }
```

11.2.1 Efficiency

If all we have to do to find a specific item in a dictionary is a simple math problem $O(1)$, this ends up being way more efficient than doing a linear search $O(n)$ to find an item, so dictionaries can be a good option for certain designs.

However, there's a problem we have to deal with...

11.3 Collisions

Using a Hash function doesn't guarantee you'll get a unique index every time it's run. Whenever two different keys result in the same index, we call this a **collision**. We have to come up with some strategy to find a *new* index that isn't taken.

Key	10	20	30
Index (key % 10)	0	0	0
		COLLISION	COLLISION

Types of collision strategies

Linear probing: When a collision is encountered, you could simply add some constant value c to the index until an empty slot is found – but this adds an element of traversing to our hash tables, though overall less traversing than searching a linear structure.

```
1 int LinearProbeA( int index )
2 {
3     return index + 1;
4 }
```

With the above, we pass in the new `index` every time there's a collision.

```
1 int LinearProbeB( int originalIndex, int collisions )
2 {
3     return originalIndex + collisions;
4 }
```

With this version, we keep track of the amount of collisions we've had (1, 2, 3, ...) and add it to the index that was originally generated by the HashFunction. This results in the same result as `LinearProbeA`.

Quadratic probing: Instead of searching forward by 1 each time from the collision location, we could use other locations, such as `index` + 1^2, `index` + 2^2, `index` + 3^2, etc.

```
1 int QuadraticProbe( int originalIndex, int collisions )
2 {
3     // collisions = # of collisions (1, 2, 3, ...)
4     int quadratic = collisions * collisions;
5     // return original index + the offset
6     return originalIndex + quadratic;
7 }
```

We're offseting the original index (generated by `HashFunction(key)`) and adding +1, +4, +9, etc. each time to find an available space.

Double hashing: We could utilize a secondary hash function to find an index when the primary hash fails to find an empty space.

```
1 int HashFunction( int key )      // first hash
2 {
3     return key % TABLE_SIZE;
4 }
5
6 int HashFunction2( int key )     // second hash
7 {
8     return 7 - ( key % 7 );
9 }
```

The first and second hash functions you write depend on your design, but this is an example.

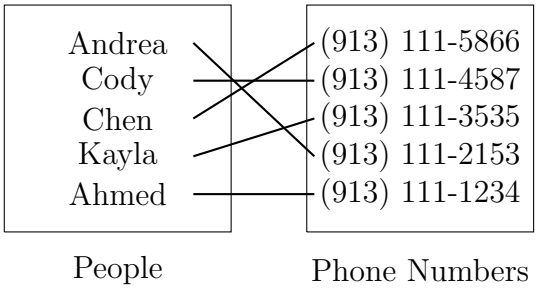
To use the second hash to get a new index for each collision, it would look something like this:

```
1 int originalIndex = index;
2 int collisions = 0;
3
4 while ( collision_occurs ) {
5     collisions++;
6     index = ( HashFunction(key)
7         + collisions * HashFunction2(key) )
8         % TABLE_SIZE;
9 }
```

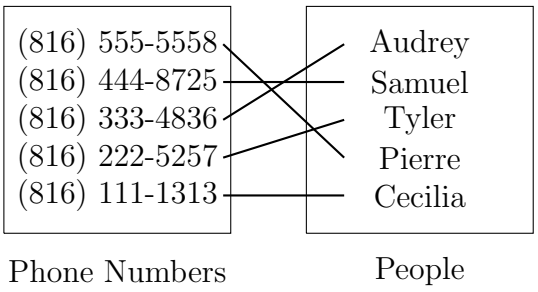
11.4 Questions

Question 1
Given the sets of information, identify what you might store as the `key` and what you might store as the `value`?

a. A phone book with a list of names in alphabetical order, and the corresponding phone numbers.



b. A phone book with a list of numbers in ascending order, and the corresponding peoples' names that each number belongs to.



Question 2
Given the sets of information, identify what you might store as the `key` and what you might store as the `value`? If you think a new piece of data needs to be added to make a **unique key**.

a. A list of class IDs (like "CS200") to search by, and the list of class information (Name, Schedule, Prereqs).

b. A list of customers in a store database, and their mailing information.

c. A list of items for sale in a store, including their name, price, and a photo.

Question 3
Find the result of each modulus problem.

a. 9 mod 3

b. 10 mod 3

c. 11 mod 3

d. 12 mod 3

e. 16 mod 5

f. 20 mod 5

g. 35 mod 4

h. 31 mod 4

Question 4
Assume the following keys will be inserted into an array. For any collisions encountered, use the **Linear Probing** strategy.

Array size: 10
Insert keys: 0, 10, 20, 30
Hash function: `index = key % arraySize;`
Linear Probe: `newIndex = index + collisionCount;`

Index	0	1	2	3	4	5	6	7	8	9
Key										

Question 5
Assume the following keys will be inserted into an array. For any collisions encountered, use the **Quadratic Probing** strategy.

Array size: 10
Insert keys: 0, 10, 20, 30
Hash function: `index = key % arraySize;`
Linear Probe: `newIndex = index + (collisionCount * collisionCount);`

Index	0	1	2	3	4	5	6	7	8	9
Key										

Question 6
Assume the following keys will be inserted into an array. For any collisions encountered, use the **Double Hashing** strategy.

Array size: 10
Insert keys: 0, 10, 20, 30
Hash function: `index = key % arraySize;`
Second Hash: `newIndex = 7 - (key % 7);`

Index	0	1	2	3	4	5	6	7	8	9
Key										