Algorithms and Data Structures (CSci 115)

California State University Fresno

College of Science and Mathematics

Department of Computer Science

H. Cecotti

Learning outcomes

- **Hash Tables**
  - What is it for?
  - How to implement it?
    - Array
    - Array + List
  - How to use it?

- A well-used data structures
  - /!\: Typical topic for job interview question
  - Example: improving the performance of search

Hash Table

- A **HASH TABLE** is a data structure that offers
  - **Fast insertion**
  - **Fast searching (key function when users wish to find an item)**
- No matter how many data items there are *insertion* and *searching* operations take close to **constant time** - O(1)
- Very fast
  - Used when you need to look up tens of thousands of items quickly
  - Spelling checkers, directories
- **Disadvantages**:
  - Based on **Arrays** - so difficult to extend after they have been created!!!!
  - Performance degrades when a table becomes too full
  - It is not a convenient way to visit the items in a hash table in any kind of order

Hashing

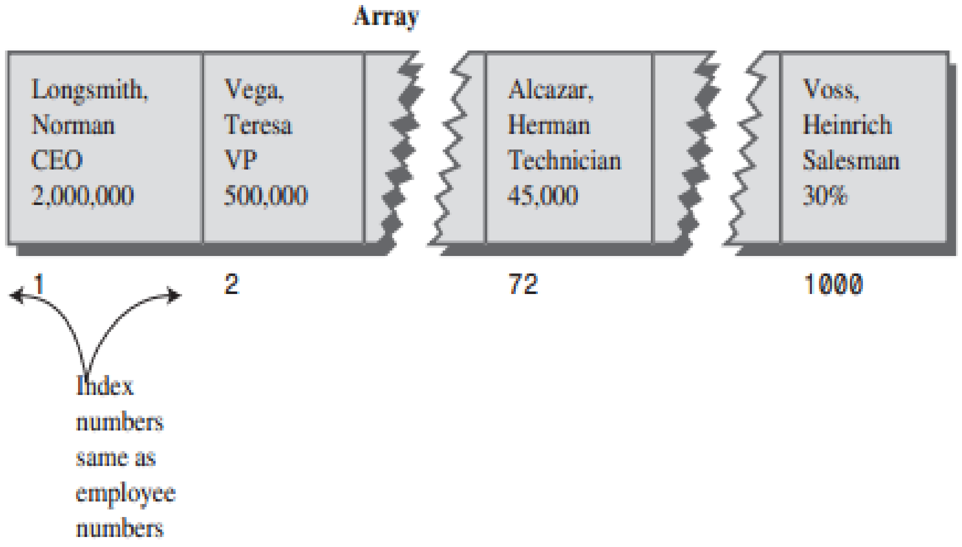
- **Definition**
  - Hashing is a technique that converts a **key value** into an **address**
  - A **range of key values** is transformed into a **range of array index values**
- When using a **Hash Table**, this is accomplished with a **Hash Function**
  - We use a **Hash Function** to convert a **key value** into an **Hash Table address**
  - Key value → Hash Table address
  - HashTableAddress=HashFunction(key value)
- For certain kinds of key values (e.g. when keys are distributed in an orderly fashion), **no hash function is necessary**; the key values can be used directly as **array indices !**
  - If you want to store integers
  - If you want to store characters – count the occurrence of each character in a string

Example: Real World Scenario

- A company has 1,000 employees
  - Every employee has been given an Employee number from 1 to 1,000
  - Employees are seldom laid off, but even when they are, their records remain in the database for reference
  - Not so good if there is a high staff churn
- The company's personnel director has specified that it wants the **fastest possible access to any individual record**
  - We need a program to access these 1,000 employee records
  - The **employee numbers** can be used as **keys** to access the records
- Note that:
  - Each employee record requires 1k byte of storage
  - Entire database uses only 1 megabyte, which **easily fits in the computer's memory**

Index Numbers as Keys

- Use a **Simple array**?
- Each Employee record occupies **one element of the array**
- The **index number** of the element is the Employee number for that record



Index Numbers as Keys

- Accessing a specified array element is very fast if you know its index number:

```
EmployeeRecord employee = databaseArray [72];
```

- Adding a new item is also very quick:

```
databaseArray [totalEmployees++] = newRecord;
```

- Array should be made **larger than the current number of employees**
  - This allows free space for the (limited) expansion anticipated
- Furthermore
  - If we expect no (or very few) deletions then wasteful gaps will not develop in the sequence
  - New employees (items) can be added at the end of the array in a sequential manner
  - The array does not need to be significantly larger than the current number of items

A Dictionary

- Assume we wish to store a 50,000 word English language dictionary in main memory
- Ideally, we would like each word to occupy its own cell in a 50,000 element array
- We can access the word using an **index number** which will make access very fast
- **However**
  - What is the relationship between these index numbers and the words?
  - For example, how do we find out the index number of the word “cat”?

Converting Words to Numbers

- We need to be able to convert an **English word** into an appropriate **index number**
- Adding the Digits **(Refer to Hashing Notes)**
  - Does not discriminate enough
  - Too few elements
  - Range of possible indices needs to be more "spread out"
- Multiplying by Powers **(Refer to Hashing Notes)**
  - Assigns an **array element** to every potential word
    - Cells for *aaaaaaaaaa*, … … … *zzzzzzzzzz*
  - Only a small fraction of these cells are necessary for real words
  - Most array cells are empty

Hashing

- Multiplying By Powers
  - We need a method for **compressing the huge range of numbers** we obtain from this system into a range that **matches a reasonably sized array**
- How large an array do we need for the English dictionary?
  - Say 50,000 words
  - Our array needs minimally this many elements ….
  - In practice, we need **an array with about twice this number of cells**
    - An array with 100,000 elements
  - We need to convert: 0 - 7,000,000,000,000   into the range   0 - 100,000
- Simple approach is to use the **modulus operator** (%)
  - Remainder when a number is divided by another number

Hashing (0 – 199) into (0 – 9)

0 to 199	0 to 9
`largeNumber`	`smallNumber`

- There are 200 values in the `largeNumber` range
- There are just 10 values in the `smallNumber` range
- We will say that a variable `smallRange` has the value **10**
- The expression for the conversion is:

```
smallNumber = largeNumber % smallRange
```

13 % 10 gives us 3

15 % 10 gives us 5

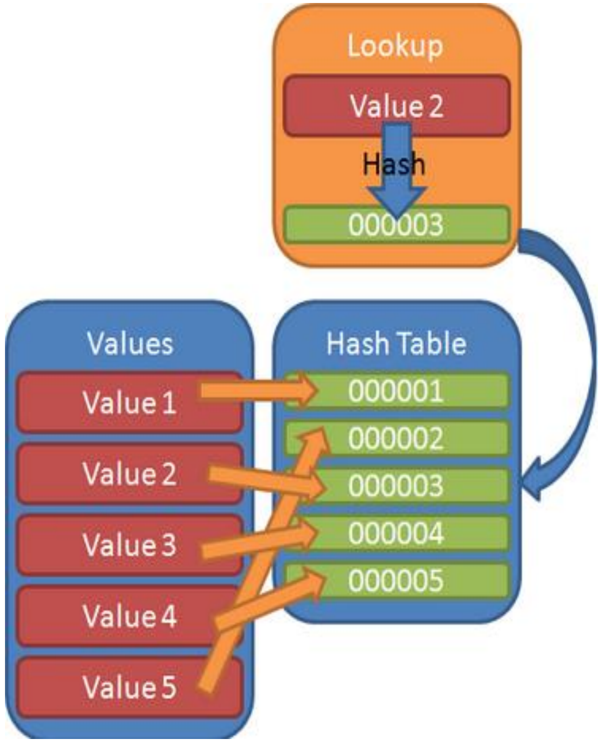
- We have squeezed the range 0-199 into the range 0-9, a 20-to-1 compression ratio (20:1)

Hashing

- A similar expression can be used to *compress* these huge numbers that uniquely represent every English word *into* index numbers that fit in our dictionary array

```
arrayIndex = hugeNumber % arraySize;
```

- This is an example of a **hash function**
- It hashes (converts) a number lying **within a large range** into a different number lying **within a smaller range**
- This **smaller range** corresponds to the **index numbers in an array**
- An array into which data is inserted using a *hash function* is called a **hash table**



Hashing

- Divide a word (string) up in its individual characters
- Allocate each character a value
- Convert a word into a **`hugeNumber`** by multiplying each character's integer value by an appropriate power of 27 (a-z+space)
  - Numbers in the base *b* system

$$(a_n a_{n-1} \cdots a_1 a_0 . c_1 c_2 c_3 \cdots)_b = \sum_{k=0}^{n} a_k b^k + \sum_{k=1}^{\infty} c_k b^{-k}$$

- Use the **modulus operator** (%), to reduce the resulting huge range of numbers into a range about twice as big as the number of items we need to store
- This is an example of a **hash function**:

```
arraySize = numberWords * 2;
arrayIndex = hugeNumber % arraySize;
```

Hashing

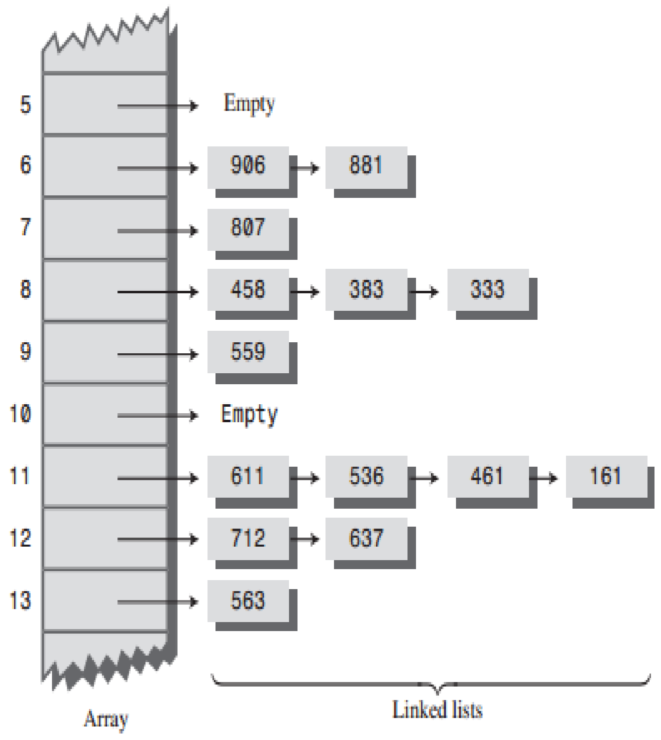
- Within this huge range, each number represents a **potential data item but** very few of these numbers are actually generated by a an English word
- They do not represent actual data items
- While we would expect that, on average, there will be one word for every 2 cells **(array elements)** in practice:
  - some cells will have no words
  - other cells will have more than one word
- Major Problem!
  - Our **`hugeNumber`** will probably overflow its variable size even for type long ☹

Collisions

- We (always) pay a price for **squeezing a large range** into a **small one**
- Why?
  - Because there is NO GUARANTEE that all the words will hash to a unique array index
  - You can have several words corresponding to the same elements in the array
- Where 2 words hash to the same array index we say we have a **hash clash**
- Similar to when we added together the letter codes, but not as bad:
  - Adding codes gave us 260 possible results (for words up to 10 letters)
  - Problem is now spread out over 50,000 (or 100,000) possible results
- **Impossible to avoid several different words hashing to the same array location**
  - → All we can hope for is that not too many words hash to the same index

Separate Chaining

- **Separate Chaining**
- Data structure:
  - Array + List
  - An array of lists
    - The head of each list points to an item
- One approach is to create an array that consists of elements that are **linked lists of words** instead of the words themselves
  - Some of the linked lists may be empty (empty buckets)
- When a collision occurs
  - the new item is simply inserted in the **list** at that index



Alternative Approach - Open Addressing

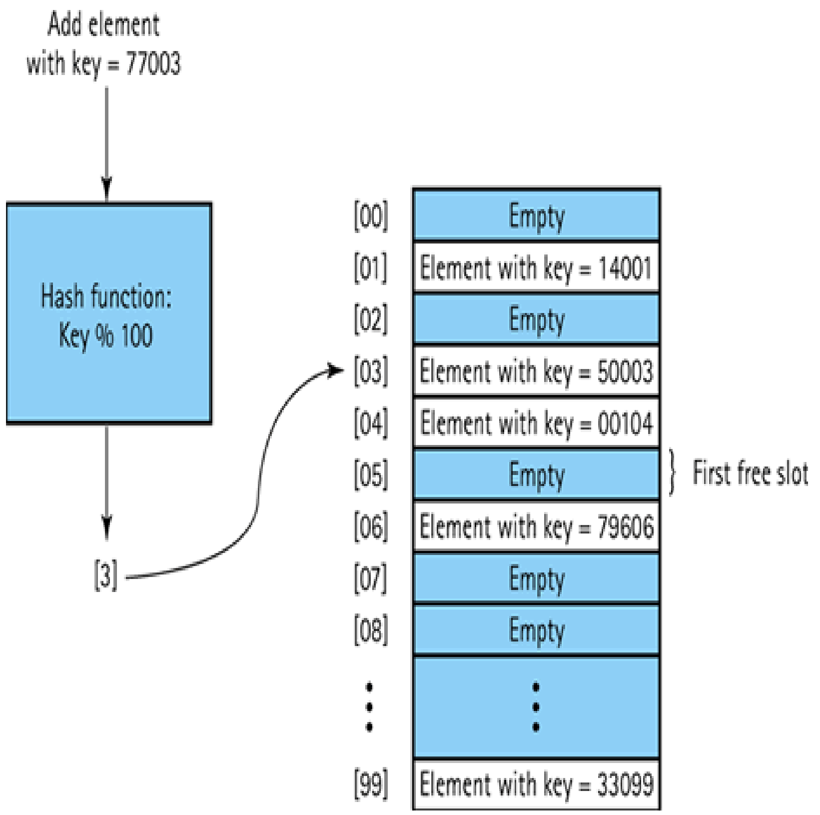
- When a collision occurs search the array in some systematic manner for the 'next available empty cell'
- The 'next available' empty cell is located
  - ➢ then we insert the new item there
- Suppose we wish to insert ***cats*** into the hash table
  - ➢ The word ***cats*** hashes to 5,421
  - ➢ This location is already occupied by ***parsnip***
- We might try to insert ***cats*** in the next adjacent element i.e. 5,422
  - ➢ This approach is called **Open Addressing**
- There are 3 methods of **Open Addressing**
  1. **Linear Probing**
  2. **Quadratic Probing**
  3. **Double Hashing**

Linear Probing

- For resolving collisions in hash tables
  - Creation: Gene Amdahl, Elaine M. McGraw, and Arthur Samuel (1954)
  - Analysis: Donald Knuth (1963)
- **Definition**
  - When the hash function causes a collision by mapping a new key to a cell of the hash table, which is already occupied by another key,
  - Linear probing searches the table for the **closest** following free location and inserts the new key there

Linear Probing

- In **Linear Probing**, we search **sequentially** for vacant cells
- If 5,421 is occupied when we try to insert a data item there, we simply move to see whether 5,422 is free and then onto 5,423 …
- We increase the index by 1 **until** we find an empty cell
- It steps sequentially along the line of cells used in the Hashing function



Linear Probing

- Search/Insert
- Delete
  - When a cell *i* is emptied
  - Necessary to search **forward** through the following cells of the table **until** we find
    - another empty cell or
    - a key that can be moved to cell *i*
      - a key whose hash value is equal to or earlier than *i*.
  - When an empty cell is found
    - emptying cell *i* is safe and the deletion process terminates.
    - when the search finds a key that can be moved to cell *i*, it performs it

Clusters

- A problem when using Linear Probing is that we can get Clusters
  - ➢ A cluster is a “run of occupied cells” in as hash table
  - ➢ When clusters start to form, they tend to grow larger quite quickly !!!
  - ➢ The larger the cluster, the faster it grows ☹
- Clusters can form even when the load factor is low
- Parts of the hash table may consist of large clusters, while other parts may have little occupancy and are sparsely inhabited: Clusters reduce performance
- **Definition**: The load factor = number of items in a table / size of the table
  - ➢ Tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost
- A table with 10,000 cells and 6,667 items has a load factor of 2/3
- loadFactor = noOfItems / arraySize = 6,667 / 10,000 = 2 / 3
- Remark: The default load factor of HashMap in Java is **0.75**

Quadratic Probing

- To avoid Primary Clustering, **Quadratic Probing** is used
  - If the primary hash index is **X**, then using a:
    - **Linear Probe:** we would visit (in turn)

X + 1 then **X + 2** then **X + 3** etc.

- **Quadratic Probe:** we would visit (in turn)

X + 1 then **X + 4** then **X + 9** then **X + 16**

then **X + 25** then **X+36** then **X + 49**

- The distance from the initial probe is the **square of the step number**:

$$x + 1^2 \quad x + 2^2 \quad x + 3^2 \quad x + 4^2 \quad x + 5^2 \quad x + 6^2 \quad x + 7^2$$

- **Secondary Clustering**
  - Problem - All the keys that hash to a particular cell **follow the same sequence** in trying to find a vacant space

Quadratic Probing

- Example

```c
int quadratic_probing_insert(int *hashtable, int key, int *empty) {
    // hashtable[] is an int hash table
    // empty[] is  array that indicates whether the key space is occupied
    int i, index;
    for (i = 0; i < SIZE; i++) {
        index = (key + i*i) % SIZE;
        if (empty[index]) {
            hashtable[index] = key;
            empty[index] = 0;
            return index;
        }
    }
    return -1;
}

int quadratic_probing_search(int *hashtable, int key, int *empty) {
    // If the key is found in the hash table
    // the function returns the index of the hashtable where the key is inserted, otherwise it
    int i, index;
    for (i = 0; i < SIZE; i++) {
        index = (key + i*i) % SIZE;
        if (!empty[index] && hashtable[index] == key)
            return index;
    }
    return -1;
}
```

Double Hashing

- Generate **PROBE SEQUENCES** that **depend on the KEY**
  - Numbers with different keys that hash to the same index will use different **probe sequences**
- **Hash the key a second time**
  - Use a **different** hash function
  - Use the **result as the step size**
    - Step size remains constant throughout a probe
    - Step size is different for different keys

Double Hashing

- Secondary hash function:
  - Must not be the same as the primary hash function
  - Must never output a 0
  - Otherwise
    - there would be no step; every probe would land on the same cell, and the algorithm would go into an endless loop
- Functions of the following form work well:

```
stepSize = constant – (key % constant);
```

where **`constant`** is **prime** and **smaller** than the array size

Example:

```
stepSize = 5 – (key % 5);
```

Hash Function – Speed

- Definition
  - The purpose of a hash function is to:

  "**take a range of key values** and **transform them into index values** so that the key values are (relatively) **distributed in a random manner** across all the indices of the hash table"
- A good hash function
  - is **simple**
  - can be **computed quickly**
- The major advantage of hash tables is their **speed**
- Hence, if a hash function is slow, the advantage of using the hash function is diminished
- Hash functions with many multiplications and divisions are not a good idea! (Tend to be slow)

Random Keys

- A **perfect** hash function **maps every key into a different table location** resulting in zero collisions
  - Normally only possible for keys where the range is small enough to be used directly as array indices
  - → employee-number example
- Most hash functions will need to **compress a larger range of keys into a smaller range of index numbers**
- The distribution of key values in a particular database determines what the hash function needs to do
- If the data is randomly distributed over the entire range, the following hash function can be used:

```
index = key % arraySize;
```

- It involves only **one mathematical operation**, and if the keys are truly random, the resulting indices will be random too, and therefore well distributed

Non-Random Keys

- Data is often distributed in a non-random manner
- Consider a database that uses **CAR-PART** numbers as **keys**
- For example: 033-400-03-14-05-1-165 is interpreted as follows:

Digits 0 - 2:	**Supplier number**	033	
Digits 3 - 5:	**Category code**	400	
Digits 6 - 7:	**Month of introduction**	03	(March)
Digits 8 - 9:	**Year of introduction** 14	(2014)	
Digits 10 - 11:	**Serial number**	05	
Digit 12:	**Toxic risk flag**	1	(true)
Digit 13 - 15:	**Check Sum**	165	

- The key used for the part number shown would be

  0,334,000,314,051,165

Non-Random Keys

- Note:
  - Such keys are not randomly distributed
  - The majority of numbers from 0 to 9,999,999,999,999,999 cannot actually occur
  - The **checksum** is dependent on the other numbers
- The key fields should be reduced until every bit counts
- For example
  - the category codes could be changed to run from 0 to 15 (4 bits)
- The checksum should be removed because it does not add any additional information - it is deliberately redundant

Retrieving Data

- **Every part of the key** should **contribute to the hash function**
  - Do not just use (say) the first four digits
- The more data that contributes to the key
  - →the more likely it is that the keys will hash evenly into the entire range of indices
- The trick is to find a hash function that is **simple** and **fast**, yet **excludes the non-data parts of the key** and **uses all the data**

- Generally easy to retrieve values as long as you know:
  - Hash function
  - Re-hash function
- Follow the path in accordance with these functions

Less Obvious Problems

- Retrieve **345766** FOUND at Location 766
- Retrieve **873766**
  - Hashes to 766 NOT FOUND (but occupied)
  - Rehash to 767 NOT FOUND (but occupied)
  - Rehash to 768 FOUND
- Retrieve **113768**
  - Hashes to 768 NOT FOUND (but occupied)
  - Rehash to 769 NOT FOUND (but EMPTY)
  - Item is NOT PRESENT IN THE TABLE
- Delete **542766**
- Retrieve **873766**
  - Hashes to 766 NOT FOUND (but occupied)
  - Rehash to 767 NOT FOUND (but EMPTY)
  - Likely to assume item is **NOT PRESENT IN THE TABLE!!**

key % 1000

Location	Key
762	132762
763	
764	984764
765	981765
766	345766
767	(542766)
768	873766
769	
770	
771	

Solution

- Do NOT DELETE ANY ITEM ☺
  - Instead **mark** the location as having been **DELETED** and continue searching (re-hashing)
- Marking as deleted is only practical when there are only a small number of deletions
  - Alternatively, we should mark as DELETED but regularly restructure the entire table by rehashing (which is time consuming)

Hashing Efficiency

- Insertion and searching in hash tables can approach O(1) time
  - In the absence of any collisions all that is necessary to insert a new item or find an existing item
  - a single call to the hash function and a single array reference
- This is the **minimum access time**
- If (collision)
  - access time becomes dependent on the **resulting probe lengths**
- Each cell accessed during a probe adds another time increment to the search for a vacant cell (for insertion) or for an existing cell.

Hashing Efficiency

- If we assume a constant time to evaluate the hash function,
  - then an individual search or insertion time is simply **proportional to the length of the probe**
- The average probe length (and therefore the average access time)
  - dependent on the **load factor**
- As the **load factor** increases → the probe lengths grow longer

Summary

- Hash tables: main features
  - Hash tables are based on arrays
  - The range of key values is usually greater than the size of the array
  - A key value is hashed to an array index by a hash function
  - The hashing of a key to an already-filled array cell is called a collision
  - Collisions can be handled in two major ways:
    - **Separate Chaining**
      - each array element consists of a linked list. All data items hashing to a given array index are inserted in that list
    - **Open Addressing**
      - data items that hash to a full array cell are placed in another cell in the array

Summary

- **Linear Probing**
  - Step size is always 1
  - If X is the array index calculated by the hash function, the probe goes to (X) to (X + 1) to (X + 2) to (X + 3) to (X + 4) etc.
- **Probe Length**
  - The number of such steps required to find a specified item
- Contiguous Sequences of filled (occupied) cells appear
  - Primary Clusters
  - Reduce performance

Summary

- **Quadratic Probing**
  - The offset from x is the square of the step number
  - So the probe goes to (x) to (x + 1) to (x + 4) to (x + 9) to (x + 16) etc.
  - Eliminates **primary clustering** but suffers from (less severe) **secondary clustering**
  - **Secondary clustering** occurs because all the keys that hash to the same initial value follow the same sequence of steps during a probe
    - The step size does NOT depend on the key, but only on the hash value
- **Double Hashing**
  - Step size depends on the key and is obtained from a secondary hash function
  - If the secondary hash function returns a value **s** in double hashing,
    - the probe goes to (**x**)
    - then (**x + s**) then (**x + 2s**) then (**x + 3s**) then (**x + 4s**) etc.
      - where **s** depends on the key but remains constant during the probe

Summary

- Load factors
  - Defined as the ratio of **data items** in a hash table to the array size
    - The **maximum load factor** in open addressing **should** be around 0.5
  - For double hashing at this load factor, searches will have an average probe length of 2
  - Search times go to infinity as load factors approach 1.0 in open addressing.
  - Crucial that an open-addressing hash table does not become too full
  - A load factor of 1.0 is appropriate for separate chaining
  - At this load factor, a successful search has an average probe length of 1.5, and an unsuccessful search, 2.0
  - Probe lengths in separate chaining increase linearly with load factor

Summary

- A string (sequence of characters) can be hashed by:
  - Multiplying each individual character by a different power of a constant
  - Adding the products, and
  - using the modulus operator (%) to reduce the result to the size of the hash table
- To avoid overflow, we can apply the modulus operator at each step in the process
- Hash table sizes should generally be **prime numbers** because it minimizes clustering in the hashed table
  - For example, use 1997 instead of 2000
  - Especially important in quadratic probing and separate chaining
  - To minimize collisions, it is important to reduce the number of common factors between the number of buckets and the elements of K.
  - How can this be achieved?
    - By choosing m to be a number that has very few factors: a prime number.
- **A perfect hash function is one with no collisions**

Questions ?

- There are HashTable already in C++ 11 ☺
  - ➢Unordered associative containers in the standard library
    - https://en.cppreference.com/w/cpp/named_req/UnorderedAssociativeContainer
- Reading
  - ➢CSci 115 book: Chapter 6
  - ➢Chapter 11: Introduction to Algorithms 3rd Edition.