Hash Tables
Objectives

- Understand the basic structure of a hash table and its associated hash function
  - Understand what makes a good (and a bad) hash function
- Understand how to deal with collisions
  - Open addressing
  - Separate chaining
- Be able to implement a hash table
- Understand how occupancy affects the efficiency of hash tables
What can we do if we want rapid access to individual data items?

- Looking up data for a flight in an air traffic control system
- Looking up the address of someone making a 911 call
- Checking the spelling of words by looking up each one in a dictionary

In each case speed is very important

- But the data does not need to be maintained in order
Possible Solutions

- Balanced binary search tree
  - Lookup and insertion in $O(\log n)$ time
    - Which is relatively fast
  - Binary search trees also maintain data in order, which may be not necessary for some problems

- Arrays
  - Allow insertion in constant time, but lookup requires linear time
  - But, if we know the index of a data item lookup can be performed in constant time
Can we use an array to insert and retrieve data in constant time?

- Yes – as long as we know an item's index
- Consider this (very) constrained problem domain:
  - A phone company wants to store data about its customers in Convenientville
  - The company has approximately 9,000 customers
  - Convenientville has a single area code (555)
Create an array of size 10,000
- Assign customers to array elements using their (four digit) phone number as the index
- Only around 1,000 array elements are wasted
- Customer data can be found in constant time using their phone numbers

Of course this is not a general solution
- It relies on having conveniently numbered key values
In the Convientville example each possible key value was assigned an array element

- With the index being the 4 digit phone number
- Therefore the array size is the number of possible values (10,000 in the example)  
  - Not the number of actual values (9,000 in the example)

Consider two more examples that use this same general idea

- Canadian phone numbers
- Names
Phone Numbers in General

- Let's consider storing information about Canadians given their phone numbers
  - Between 000-000-000 and 999-999-9999
- It's easy to convert phone numbers to integers
  - Just get rid of the "-"s
  - The keys range between 0 and 9,999,999,999
- Use *Convenientville* scheme to store data
  - But will this work?
A Really Big Array!

- If we use Canadian phone numbers as the index to an array how big is the array?
  - 9,999,999,999 (ten billion)
  - That's a really big array!

- An estimate of the current population of Canada is 35,623,680 [source: CIA World Fact Book]
  - That means that we will use around 0.3% of the array
    - That's a lot of wasted space
    - And the array may not fit in main memory...
What if we had to store data by name?
  - We would need to convert strings to integer indexes
Here is one way to encode strings as integers
  - Assign a value between 1 and 26 to each letter
    - a = 1, z = 26 (regardless of case)
    - Sum the letter values in the string
Not a very good method ...

"dog" = 4 + 15 + 7 = 26
"god" = 7 + 15 + 4 = 26
Ideally we would like to have a unique integer for each possible string
- The “sum the letters” encoding scheme does not achieve this

There is a simple method to achieve this goal
- As before, assign each letter a value between 1 and 26
- Multiply the letter's value by $26^i$, where $i$ is the position of the letter in the word:
  - "dog" = $4\times26^2 + 15\times26^1 + 7\times26^0 = 3,101$
  - "god" = $7\times26^2 + 15\times26^1 + 4\times26^0 = 5,126$
The proposed system generates a unique integer for each string

- But most strings are not meaningful
- Given a string containing ten letters there are $26^{10}$ possible combinations of letters
  - Which gives 141,167,095,653,376 different possible strings
  - There are around 200,000 words in the English language

It is not practical to create an array large enough to store all possible strings

- Just like the general telephone number problem
In an ideal world we would know which key values were going to be recorded

- The *Convenientville* example was close to ideal

Most of the time this is not the case

- Usually, key values are not known in advance
- And, in many cases, the universe of possible key values is very large (e.g. names)
- So it is not practical to reserve space for all possible key values
A Different Approach

- Don't determine the array size by the maximum possible number of keys
- Fix the array size based on the amount of data to be stored
  - Map the key value (phone number or name or some other data) to an array element
  - We will need to convert the key value to an integer index using a hash function
- This is the basic idea behind hash tables
Hash Tables
Hash Tables

- A hash table consists of an array to store data
  - Data often consists of complex types
    - Or pointers to such objects
  - An attribute of the object is designated as the table's key

- A hash function maps the key to an index
  - The key must first be converted to an integer
  - And mapped to an array index using a function
    - Often the modulo function
Collisions

- A hash function may map two different keys to the same index.
  - Referred to as a *collision*.
  - Consider mapping phone numbers to an array of size 1,000 where \( h = \text{phone mod 1,000} \).
    - Both 604-555-1987 and 512-555-7987 map to the same index.
      \( 6,045,551,987 \mod 1,000 = 987 \).
- A good hash function can significantly reduce the number of collisions.
- It is still necessary to have a policy to deal with any collisions that may occur.
Hash Functions
A hash function is a function that maps key values to array indexes

Hash functions are performed in two steps
- Map the key value to an integer
- Map the integer to a legal array index

Hash functions should have the following properties
- Fast
- Deterministic
- Uniformity
Hash functions should be fast and easy to calculate
- Access to a hash table should be nearly instantaneous and in constant time
- Most common hash functions require a single division on the representation of the key
- Converting the key to a number should also be able to be performed quickly
Deterministic Hash Functions

- A hash function must be *deterministic*
  - For a given input it must always return the same value
    - Otherwise it will not generate the same array index
    - And the item will not be found in the hash table
- Hash functions should therefore not be determined by
  - System time
  - Memory location
  - Pseudo-random numbers
A typical hash function usually results in some collisions
- Where two different search keys map to the same index
- A *perfect* hash function avoids collisions entirely
  - Each search key value maps to a different index
The goal is to *reduce* the number and effect of collisions
To achieve this, the data should be distributed evenly over the table
Possible Values

- Any set of values stored in a hash table is an instance of the universe of possible values.
- The universe of possible values may be much larger than the instance we wish to store.
  - There are many possible combinations of 10 letters: $26^{10}$.
  - But we might want a hash table to store just 1,000 names.
Uniformity

- A good hash function generates each value in the output range with the same probability
  - That is, each legal hash table index has the same chance of being generated
- This property should hold for the universe of possible values and for the expected inputs
  - The expected inputs should also be scattered evenly over the hash table
A hash table is to store 1,000 numeric estimates that can range from 1 to 1,000,000

- Hash function is \( \text{estimate} \% n \)
  - Where \( n = \text{array size} = 1,000 \)
- Is the distribution of values from the universe of all possible values uniform?
- And what about the distribution of expected values?
A hash table is to store 676 names
- The hash function considers just the first two letters of a name
  - Each letter is given a value where a = 1, b = 2, ...
  - Function = (1\text{st} letter × 26 + value of 2\text{nd} letter) \% 676

- Is the distribution of values from the universe of all possible values uniform?
- And what about the distribution of expected values?
General Principles

- Use the entire search key in the hash function.
- If the hash function uses modulo arithmetic, make the table size a prime number.
- A simple and effective hash function is:
  - Convert the key value to an integer, $x$.
  - $h(x) = x \mod \text{tableSize}$
    - Where $\text{tableSize}$ is the first prime number larger than twice the size of the number of expected values.
Consider mapping \( n \) values from a universe of possible values \( U \) into a hash table of size \( m \)

- If \( U \geq n \times m \)
- Then for any hash function there is a set of values of size \( n \) where all the keys map to the same location!

Determining a good hash function is a complex subject

- That is only introduced in this course
Converting Strings to Integers
A simple method of converting a string to an integer is to:
- Assign the values 1 to 26 to each letter
- Concatenate the binary values for each letter
  - Similar to the method previously discussed

Using the string *cat* as an example:
- c = 3 = 00011, a = 00001, t = 20 = 10100
- So *cat* = 000110000110100 (or 3,124)
- Note that $32^2 \times 3 + 32^1 \times 1 + 20 = 3,124$
If each letter of a string is represented as a 32 bit number then for a length $n$ string

- value = $c_0 \times 32^{n-1} + ... + c_{n-2} \times 32^1 + c_{n-1} \times 32^0$
- For large strings, this value will be very large
  - And may result in overflow

This expression can be *factored*

- $(...((c_0 \times 32 + c_1) \times 32 + c_2) \times ...)) \times 32 + c_{n-1}$
- This technique is called *Horner's Method*
- This minimizes the number of arithmetic operations

Overflow can then be prevented by applying the *mod* operator after each expression in parentheses
Horner's Method Example

- Consider the integer representation of some string
  - \(6 \times 32^3 + 18 \times 32^2 + 15 \times 32^1 + 8 \times 32^0\)
  - \(= 196,608 + 18,432 + 480 + 8 = 215,528\)
- Factoring this expression results in
  - \(((6 \times 32 + 18) \times 32 + 15) \times 32 + 8) = 215,528\)
- Assume that this key is to be hashed to an index using the hash function \(\text{key} \% 19\)
  - \(215,528 \% 19 = 11\)
  - \(((6 \times 32 + 18) \% 19 \times 32 + 15) \% 19 \times 32 + 8) \% 19 = 11\)
  - \(210 \% 19 = 1, \text{ and } 47 \% 19 = 9, \text{ and } 296 \% 19 = 11\)
Collisions
A collision occurs when two different keys are mapped to the same index
- Collisions may occur even when the hash function is good
- There are two main ways of dealing with collisions
  - Open addressing
  - Separate chaining
Open Addressing

- Idea – when an insertion results in a collision look for an empty array element
  - Start at the index to which the hash function mapped the inserted item
  - Look for a free space in the array following a particular search pattern, known as *probing*
- There are three open addressing schemes
  - Linear probing
  - Quadratic probing
  - Double hashing
Linear Probing

- The hash table is searched sequentially
  - Starting with the original hash location
  - For each time the table is probed (for a free location) add one to the index
    - Search $h(search\ key) + 1$, then $h(search\ key) + 2$, and so on until an available location is found
    - If the sequence of probes reaches the last element of the array, wrap around to $array[0]$
- Linear probing leads to primary clustering
  - The table contains groups of consecutively occupied locations
  - These clusters tend to get larger as time goes on
    - Reducing the efficiency of the hash table
Hash table is size 23

The hash function, $h = x \mod 23$, where $x$ is the search key value

The search key values are shown in the table

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>29</td>
<td>32</td>
<td>58</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linear Probing Example

- Insert 81, \( h = 81 \mod 23 = 12 \)
- Which collides with 58 so use linear probing to find a free space
- First look at 12 + 1, which is free so insert the item at index 13
Linear Probing Example

- Insert 35, \( h = 35 \mod 23 = 12 \)
- Which collides with 58 so use linear probing to find a free space
- First look at 12 + 1, which is occupied so look at 12 + 2 and insert the item at index 14

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10|11|12|13|14|15|16|17|18|19|20|21|22|
|   |   |   |   |   |   |29 |32 |58|81|35|   |   |   |   |   |   |   |   |   |   |   |21|   |
Insert 60, \( h = 60 \mod 23 = 14 \)

Note that even though the key doesn’t hash to 12 it still collides with an item that did

First look at 14 + 1, which is free
Linear Probing Example

- Insert 12, \( h = 12 \mod 23 = 12 \)
- The item will be inserted at index 16
- Notice that primary clustering is beginning to develop, making insertions less efficient
Searching

- Searching for an item is similar to insertion
- Find 59, $h = 59 \mod 23 = 13$, index 13 does not contain 59, but is occupied
- Use linear probing to find 59 or an empty space
- Conclude that 59 is not in the table

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 29 | 32 | 58 | 81 | 35 | 60 | 12 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   | 21 |   |
Quadratic Probing

- Quadratic probing is a refinement of linear probing that prevents primary clustering
  - For each probe, $p$, add $p^2$ to the original location index
    - $1^{st}$ probe: $h(x)+1^2$, $2^{nd}$: $h(x)+2^2$, $3^{rd}$: $h(x)+3^2$, etc.
- Results in secondary clustering
  - The same sequence of probes is used when two different values hash to the same location
  - This delays the collision resolution for those values
- Analysis suggests that secondary clustering is not a significant problem
- Hash table is size 23
- The hash function, $h = x \mod 23$, where $x$ is the search key value
- The search key values are shown in the table

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|   |   |   |   |   |   |   |   |   | 29 | 32 | 58  |     |     |     |     |     |     |     |     |     |     |     |
|   |   |   |   |   |   |   |   |   |     |     |     |     |     |     |     |     |     |     |     |     |     | 21  |
Insert 81, $h = 81 \mod 23 = 12$
Which collides with 58 so use quadratic probing to find a free space
First look at $12 + 1^2$, which is free so insert the item at index 13
- Insert 35, $h = 35 \mod 23 = 12$
- Which collides with 58
- First look at $12 + 1^2$, which is occupied, then look at $12 + 2^2 = 16$ and insert the item there

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |10 |11 |12 |13 |14 |15 |16 |17 |18 |19 |20 |21 |22 |
|   |   |   |   |   |   |29 |32 |58 |81 |   |   |   |   |   |   |35 |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |21 |
Insert 60, $h = 60 \ mod \ 23 = 14$

The location is free, so insert the item
- Insert 12, \( h = 12 \mod 23 = 12 \)
- First check index \( 12 + 1^2 \),
- Then \( 12 + 2^2 = 16 \),
- Then \( 12 + 3^2 = 21 \) (which is also occupied),
- Then \( 12 + 4^2 = 28 \), wraps to index 5 which is free
Quadratic Probe Chains

- Note that after some time a sequence of probes repeats itself
  - In the preceding example \( h(key) = key \% 23 = 12 \), resulting in this sequence of probes (table size of 23)
    - 12, 13, 16, 21, 28(5), 37(14), 48(2), 61(15), 76(7), 93(1), 112(20), 133(18), 156(18), 181(20), 208(1), 237(7), ...
  - This generally does not cause problems if
    - The data is not significantly skewed,
    - The hash table is large enough (around 2 * the number of items), and
    - The hash function scatters the data evenly across the table
In both linear and quadratic probing the probe sequence is independent of the key.

Double hashing produces *key dependent* probe sequences:
- In this scheme a second hash function, $h_2$, determines the probe sequence.
- The second hash function must follow these guidelines:
  - $h_2(key) \neq 0$
  - $h_2 \neq h_1$
  - A typical $h_2$ is $p - (key \ mod \ p)$ where $p$ is a prime number.
Double Hashing Example

- Hash table is size 23
- The hash function, \( h = x \mod 23 \), where \( x \) is the search key value
- The second hash function, \( h_2 = 5 - (key \mod 5) \)
Double Hashing Example

- Insert 81, \( h = 81 \ mod \ 23 = 12 \)
- Which collides with 58 so use \( h_2 \) to find the probe sequence value
- \( h_2 = 5 - (81 \ mod \ 5) = 4 \), so insert at 12 + 4 = 16
Double Hashing Example

- Insert 35, $h = 35 \mod 23 = 12$
- Which collides with 58 so use $h_2$ to find a free space
- $h_2 = 5 - (35 \mod 5) = 5$, so insert at $12 + 5 = 17$
Double Hashing Example

- Insert 60, $h = 60 \mod 23 = 14$

```
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
  29  32  58  60  81  35    21
```
Double Hashing Example

- Insert 83, \( h = 83 \mod 23 = 14 \)
- \( h_2 = 5 - (83 \mod 5) = 2 \), so insert at 14 + 2 = 16, which is occupied
- The second probe increments the insertion point by 2 again, so insert at 16 + 2 = 18

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   | 29 | 32 | 58 | 60 | 81 | 35 | 83 | 21 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |


Deletions and Open Addressing

- Deletions add complexity to hash tables
  - It is easy to find and delete a particular item
  - But what happens when you want to search for some other item?
  - The recently empty space may make a probe sequence terminate prematurely
- One solution is to mark a table location as either empty, occupied or deleted
  - Locations in the deleted state can be re-used as items are inserted
Separate Chaining

- Separate chaining takes a different approach to collisions
- Each entry in the hash table is a pointer to a linked list
  - If a collision occurs the new item is added to the end of the list at the appropriate location
- Performance degrades less rapidly using separate chaining
  - But each search or insert requires an additional operation to access the list
Efficiency
When analyzing the efficiency of hashing it is necessary to consider *load factor*, $\alpha$

- $\alpha = \frac{\text{number of items}}{\text{table size}}$
- As the table fills, $\alpha$ increases, and the chance of a collision occurring also increases
  - Performance decreases as $\alpha$ increases
  - Unsuccessful searches make more comparisons
    - An unsuccessful search only ends when a free element is found
- It is important to base the table size on the largest possible number of items
  - The table size should be selected so that $\alpha$ does not exceed $2/3$
Linear probing
- When $\alpha = 2/3$ unsuccessful searches require 5 comparisons, and
- Successful searches require 2 comparisons

Quadratic probing and double hashing
- When $\alpha = 2/3$ unsuccessful searches require 3 comparisons
- Successful searches require 2 comparisons

Separate chaining
- The lists have to be traversed until the target is found
- $\alpha$ comparisons for an unsuccessful search
  - Where $\alpha$ is the average size of the linked lists
- $1 + \alpha / 2$ comparisons for a successful search
If $\alpha$ is less than $\frac{1}{2}$, open addressing and separate chaining give similar performance

- As $\alpha$ increases, separate chaining performs better than open addressing
- However, separate chaining increases storage overhead for the linked list pointers

It is important to note that in the worst case hash table performance can be poor

- That is, if the hash function does not evenly distribute data across the table