Hash Tables

CMPT 225
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
Dictionary ADT

- Operations
  - Insert (key, value) pair
  - Lookup value for a key
  - Remove (key, value) pair
  - Modify (key, value) pair

- Dictionary ADT also known as
  - Associative Array
  - Map
Balanced binary search tree

- Binary search trees allow 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
Thinking About Arrays

- 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 around 9,000 customers
  - Convenientville has a single area code (604-555?)
Living in Convenientville

- 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
  - Customers can be looked up in constant time using their phone numbers
- Of course this is not a general solution
  - It relies on having conveniently numbered key values
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!
- Consider that the estimate of the current population of Canada is $33,476,688^*$
  - That means that we will use around $0.3\%$ of the array
    - That's a lot of wasted space
    - And the array probably won't fit in main memory ...
- *According to the 2011 Census*
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

- "dog" = 4 + 15 + 7 = 26
- "god" = 7 + 15 + 4 = 26
Finding Unique String Values

- Ideally we would like to have a unique integer for each possible string
- This is relatively straightforward
  - As before, assign each letter a value between 1 and 26
  - And multiply the letter's value by $26^i$, where $i$ is the position of the letter in the word:
    - "dog" = $4 \times 26^2 + 15 \times 26^1 + 7 \times 26^0 = 3,101$
    - "god" = $7 \times 26^2 + 15 \times 26^1 + 4 \times 26^0 = 5,126$
The proposed system generates a unique number for each string

- However most strings are not meaningful
- Given a string containing ten letters there are 26^{10} possible combinations of letters
  - That is, 141,167,095,653,376 different possible strings
- 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 to be recorded
  - The Convenientville example was very close to this 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 still need to convert the key value to an integer index using a hash function
- This is the basic idea behind hash tables
Hash Tables

CMPT 225
Hash Tables

- A hash table consists of an array to store the data in
  - The table may contain complex types, or pointers to objects
  - One attribute of the object is designated as the table's key
- And a hash function that maps a key to an array index
Hash Table Example

- Consider Customer data from A3
- Create array of pointers to Customer objects
  - This is the hash table
  - Customer *hash_table[H_SIZE];
Consider Customer data from A3
  - Say we wish to insert c = Customer (Mori, G., 500)
  - Where does it go?
  - Suppose we have a hash function \( h \)
    - \( h(c) = 7 \) (G is 7th letter in alphabet)

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mori, G, 500
Consider Customer data from A3

- Say we wish to insert \( d = \) Customer (Drew, M., 600)
- Where does it go?
  - \( h(d) = 13 \) (M is 13\(^{th}\) letter in alphabet)
Consider Customer data from A3
  ▪ Say we wish to search for Customer c (Baker, G, 480)
  ▪ Where could it be?
    ▪ \( h(c) = 7 \) (G is 7\(^{th}\) letter in alphabet)

Nope, (Baker, G) not in table!
Consider Customer data from A3

- Say we wish to insert e = Customer (Gould, G, 420)
- Where does it go?
  - \( h(e) = 7 \) (G is 7th letter in alphabet)
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 and Modulo

- A simple and effective hash function is:
  - Convert the key value to an integer, \( x \)
  - \( h(x) = x \mod \text{tableSize} \)
- We want the keys to be distributed evenly over the underlying array
  - This can usually be achieved by choosing a prime number as the table size
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 $32^2 \times 3 + 32^1 \times 1 + 20 = 3,124$
Strings to Integers

- 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 Rule
  - This minimizes the number of arithmetic operations
  - Overflow can be prevented by applying the mod operator after each expression in parentheses
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
- Should scatter data evenly through the hash table
A typical hash function usually results in some collisions

- A *perfect* hash function avoids collisions entirely
  - Each search key value maps to a different index
  - Only possible when all of the search key values actually stored in the table are known

The goal is to reduce the number and effect of collisions

To achieve this the data should be distributed evenly over the table
Assume that every search key is equally likely (i.e. uniform distribution, random)
A good hash function should scatter the search keys evenly
- There should be an equal probability of an item being hashed to each location
- For example, consider hashing 9 digit SFU ID numbers \(x\) on \(h = \text{last 2 digits of } x \mod 40\)
- Some of the 40 table locations are mapped to by 3 prefixes, others by only 2
- A better hash function would be \(h = x \mod 101\)
Evenly scattering non random data can be more difficult than scattering random data

- As an example of non random data consider a key: \( \{\text{last name}, \text{first name}\} \)
- Some first and last names occur much more frequently than others

While this is a complex subject there are two general principles

- Use the entire search key in the hash function
- If the hash function uses modulo arithmetic, the base should be prime
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
**Hash Table Example**

- Consider Customer data from A3
  - Say we wish to insert e = Customer (Gould, G, 420)
  - Where does it go?
    - \( h(e) = 7 \)  (G is 7\(^{th}\) letter in alphabet)

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Mori, G, 500
- Gould, G, 420
- Drew, M, 600
Open Addressing

CMPT 225
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
Open Addressing I – Linear Probing

CMPT 225
The hash table is searched sequentially
- Starting with the original hash location
- Search \( h(\text{search key}) + 1 \), then \( h(\text{search key}) + 2 \), and so on until an available location is found
- If the sequence of probes reaches the last element of the array, wrap around to \( arr[0] \)
Linear Probing Example

- 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></th>
<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>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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
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
Linear Probing

- The hash table is searched sequentially
  - Starting with the original hash location
  - Search \( h(\text{search key}) + 1 \), then \( h(\text{search key}) + 2 \), and so on until an available location is found
  - If the sequence of probes reaches the last element of the array, wrap around to \( arr[0] \)
- 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
Open Addressing II – Quadratic Probing

CMPT 225
Quadratic Probing

- Quadratic probing is a refinement of linear probing that prevents primary clustering
  - For each successive probe, \( i \), add \( i^2 \) to the original location index
    - 1\textsuperscript{st} probe: \( h(x)+1^2 \), 2\textsuperscript{nd}: \( h(x)+2^2 \), 3\textsuperscript{rd}: \( h(x)+3^2 \), etc.
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
Quadratic Probing Example

- 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
Quadratic Probing Example

- 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|   |   |   |    |    |    |    |    |    |    |    |    |    |    |   |
|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |   |   |
|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |   |   |   |
Insert 60, \( h = 60 \ mod \ 23 = 14 \)

- The location is free, so insert the item
Quadratic Probing Example

- 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

<table>
<thead>
<tr>
<th></th>
<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>12</td>
<td>29</td>
<td>32</td>
<td>58</td>
<td>81</td>
<td>60</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>29</td>
<td>32</td>
<td>58</td>
<td>81</td>
<td>60</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Quadratic Probe Chains

- Note that after some time a sequence of probes repeats itself
  - e.g. 12, 13, 16, 21, 28(5), 37(14), 48(2), 61(15), 76(7), 93(1), 112(20), 133(18), 156(18), 181(20)
- This generally does not cause problems if
  - The data are 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
Quadratic Probing

- Quadratic probing is a refinement of linear probing that prevents primary clustering.
- 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.
Open Addressing III – Double Hashing
CMPT 225
Double Hashing

- 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 prime
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, \( h_2 = 5 - (key \mod 5) \)

| 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 |   |     |     |     |     |     |     |     |     |     |     |     |   |   |
**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 $12 + 4 = 16$
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, \( h = 60 \mod 23 = 14 \)
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
Deletions and Open Addressing
Deletion Example

- Linear probing, $h(x) = x \mod 23$
- Suppose I want to delete 60
- Any problems?

<p>| | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>29</td>
<td>32</td>
<td></td>
<td>58</td>
<td>81</td>
<td>35</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
Deletion Example

- Linear probing, $h(x) = x \mod 23$
- Suppose I want to delete 60
Deletion Example

- Linear probing, $h(x) = x \mod 23$
- Search for 12
Deletion Example

- Linear probing, $h(x) = x \mod 23$
- Insert 15

<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>29</td>
<td>32</td>
<td>58</td>
<td>81</td>
<td>35</td>
<td>15</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>21</td>
</tr>
</tbody>
</table>
Separate Chaining

CMPT 225
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
Consider Customer data from A3

- Say we wish to insert e = Customer (Gould, G, 420)
- Where does it go?
  - \( h(e) = 7 \)  (G is 7\(^{th}\) letter in alphabet)
Consider Customer data from A3

- Say we wish to insert e = Customer (Minsky, M, 220)
- Where does it go?

<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>
</table>

- Mori, G, 500
- Gould, G, 420
- Drew, M, 600
- Minsky, M, 220
Consider Customer data from A3

- Say we wish to find e = Customer (Baker, G)
- Where could it be?
  - \( h(e) = 7 \) (G is 7\(^{th}\) letter in alphabet)

Nope, (Baker, G) not in table!
When analyzing the efficiency of hashing it is necessary to consider load factor, $\alpha$

- $\alpha = \text{number of items} / \text{table size}$
- As the table fills, $\alpha$ increases, and the chance of a collision occurring also increases
- So performance decreases as $\alpha$ increases
- Unsuccessful searches require more comparisons than successful searches

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 $2/3$
### Average Comparisons

- **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
  - $1 + \alpha / 2$ comparisons for a successful search
If $\alpha$ is less than 0.5 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
Summary

CMPT 225
Hash tables
- Store data in array
- Position in array determined by hash function

Hash functions can map different items to the same position (collision)
- Resolve via linear/quadratic probing, double hashing, or open chaining

Performance of hash table can be very fast (constant time)
- Actual performance depends on load factor and hash function
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
Readings

- Carrano: Ch. 12