Hash Tables

• Hashing
• Probing
• Separate Chaining
• Hash Function
Introduction

• In Chapter 4 we saw:
  • linear search – $O(n)$
  • binary search – $O(\log n)$

• Can we improve the search operation to achieve better than $O(\log n)$ time?
Comparison-Based Searches

- To locate an item, the target search key has to be compared against the other keys in the collection.
  - $O(\log n)$ is the best that can be achieved.
  - We must use a different technique if we want to improve the search time.
Hashing

- The process of mapping a search key to a limited range of array indices.
  - The goal of providing direct access to the keys.
  - **hash table** – the array containing the keys.
  - **hash function** – maps a key to an array index.
Hashing Example

- Suppose we have the following set of keys
  
  \[765, 431, 96, 142, 579, 226, 903, 388\]

and a hash table, T, with \(M = 13\) elements.

- We can define a simple hash function \(h()\)

\[h(key) = key \% M\]
Adding Keys

- To add a key to the hash table:
  - Apply the hash function to determine the array index in which the key should be stored.
    - $h(765) \Rightarrow 11$
    - $h(431) \Rightarrow 2$
    - $h(96) \Rightarrow 5$
    - $h(142) \Rightarrow 12$
    - $h(579) \Rightarrow 7$
  - Store the key in the given slot.
Collisions

- What happens when we attempt to add key 226?
  \[ h(226) \Rightarrow 5 \]

- **collision** – when two or more keys map to the same hash location.
Probing

- If two keys map to the same table entry, we must resolve the collision to find another available slot.
  - **linear probe** – simplest approach which examines the table entries in sequential order.
Probing

- Consider adding key 903 to our hash table.

\[ h(903) \Rightarrow 6 \]
Probing

- If the end of the array is reached during the probe, it wraps around to the first entry and continues.
- Consider adding key 388 to our hash table.

\[ h(388) \Rightarrow 11 \]
Searching

- Searching a hash table for a specific key is very similar to the add operation.
  - Target key is mapped to an initial slot.
  - See if the slot contains the target.
  - Otherwise, apply the same probe used to add keys to locate the target.

- Example: search for key 903.
Searching

- What if the key is not in the hash table?

- The probe continues until either:
  - a null reference is reached, or
  - all slots have been examined.
Deleting Keys

- Deleting a key from a hash table is a bit more complicated than adding keys.
  - We can search for the key to be deleted.
  - But we cannot simply remove it by setting the entry to `None`. 
Incorrect Deletion

- Suppose we simply remove key 226 from slot 6.

- What happens if we search for key 903?
Correct Deletion

- We use a special flag to indicate the entry is now empty, but was previously occupied.

- When searching a hash table, the probe must continue past the slot(s) with the special flag.
Clustering

- The grouping of keys in a common area.
  - As more keys are added to the hash table, more collisions are likely to occur.
  - Clusters begin to form due to the probing required to find an empty slot.
  - As a cluster grows larger, more collisions will occur.

- **primary clustering** – clustering around the original hash position.
Probe Sequence

- The order in which the hash entries are visited during a probe.
  - The linear probe steps through the entries in sequential order.
  - The next array slot can be represented as
    \[ \text{slot} = (\text{home} + i) \mod M \]
- where
  - \( i \) is the \( i^{th} \) probe.
  - home is the **home position**
Modified Linear Probe

- We can improve the linear probe by changing the step size to some fixed constant.
  
  $\text{slot} = (\text{home} + i \times c) \mod M$

- Suppose we set $c = 3$ to build the hash table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>765</td>
<td>11</td>
</tr>
<tr>
<td>431</td>
<td>2</td>
</tr>
<tr>
<td>96</td>
<td>5</td>
</tr>
<tr>
<td>142</td>
<td>12</td>
</tr>
<tr>
<td>579</td>
<td>7</td>
</tr>
<tr>
<td>226</td>
<td>5</td>
</tr>
<tr>
<td>903</td>
<td>6</td>
</tr>
<tr>
<td>388</td>
<td>11</td>
</tr>
<tr>
<td>903</td>
<td>6</td>
</tr>
<tr>
<td>226</td>
<td>5</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9 10 11 12
Quadratic Probing

- A better approach for reducing primary clustering.
  
  \[ \text{slot} = (\text{home} + i**2) \mod M \]

- Increases the distance between each probe in the sequence.

- Example:
  
  \[
  \begin{align*}
  h(765) &\Rightarrow 11 & h(579) &\Rightarrow 7 \\
  h(431) &\Rightarrow 2 & h(226) &\Rightarrow 5 &\Rightarrow 6 \\
  h(96) &\Rightarrow 5 & h(903) &\Rightarrow 6 &\Rightarrow 7 &\Rightarrow 10 \\
  h(142) &\Rightarrow 12 & h(388) &\Rightarrow 11 &\Rightarrow 12 &\Rightarrow 2 &\Rightarrow 7 &\Rightarrow 1
  \end{align*}
  \]
Quadratic Probing

- Reduces the number of collisions.
- Introduces the problem of **secondary clustering**.
  - When two keys map to the same entry and have the same probe sequence.
- Example: add key 648
  - hashes to entry 11
  - follows the same sequence as key 388
Double Hashing

- When a collision occurs, a second hash function is used to build a probe sequence.
  \[ \text{slot} = (\text{home} + i \times \text{hp(key)}) \mod M \]

- Step size remains a constant throughout the probe.
- Multiple keys that have the same home position, will have different probe sequences.
Double Hashing

- A simple choice for the second hash function.

\[ h_p(key) = 1 + key \mod P \]

- Example: let \( P = 8 \)

<table>
<thead>
<tr>
<th>Input</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>765</td>
<td>11</td>
</tr>
<tr>
<td>431</td>
<td>2</td>
</tr>
<tr>
<td>96</td>
<td>5</td>
</tr>
<tr>
<td>142</td>
<td>12</td>
</tr>
<tr>
<td>579</td>
<td>7</td>
</tr>
<tr>
<td>226</td>
<td>5</td>
</tr>
<tr>
<td>903</td>
<td>6</td>
</tr>
<tr>
<td>388</td>
<td>11</td>
</tr>
</tbody>
</table>

![Hash Table Diagram]
Table Size

- How big should a hash table be?
  - If we know the max number of keys.
    - create it big enough to hold all of the keys.
  - In most instances, we don't know the number of keys.
- Most probing techniques work best when the table size is a prime number.
Rehashing

- We can start with a small table and expand it as needed.
  - Similar to the approach used with the vector.
- **load factor** – the ratio between the number of keys and the size of the table.
  - A hash table should be expanded before the load factor reaches 80%.
Rehashing Example

- After creating a larger array for the table, we can not simply copy the original keys to the new table.

- We must rebuild or rehash the entire table.

```
<table>
<thead>
<tr>
<th>388</th>
<th>431</th>
<th>96</th>
<th>226</th>
<th>579</th>
<th>903</th>
<th>765</th>
<th>142</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>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>765</th>
<th>579</th>
<th>903</th>
<th>226</th>
<th>431</th>
<th>142</th>
<th>388</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>
</tr>
</tbody>
</table>
```
Expansion Size

- Size of the expansion depends on the application.
- Good rule of thumb is to at least double its size.
- Two common approaches:
  - double the size of the table, then search for the first larger prime number.
  - double the size of the table and add one to ensure $M$ is odd.
Efficiency Analysis

- Depends on:
  - the hash function
  - size of the table
  - type of collision resolution probe
- Once an empty slot is located, adding or deleting a key can be done in O(1) time.
- The time required to perform the search is the main contributor to the overall time of all ops.
Efficiency Analysis

- Best case: $O(1)$
  - The key maps directly to the correct entry.
  - There are no collisions.

- Worst case: $O(m)$
  - Assume there are $n$ keys stored in a table of size $m$.
  - The probe has to visit every entry in the table.
Efficiency Analysis

- While hashing appears to be no better than a basic linear search, hashing is very efficient in the average case.

<table>
<thead>
<tr>
<th>Load Factor</th>
<th>0.25</th>
<th>0.5</th>
<th>0.67</th>
<th>0.8</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Successful search:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear probe</td>
<td>1.17</td>
<td>1.50</td>
<td>2.02</td>
<td>3.00</td>
<td>50.50</td>
</tr>
<tr>
<td>Quadratic probe</td>
<td>1.66</td>
<td>2.00</td>
<td>2.39</td>
<td>2.90</td>
<td>6.71</td>
</tr>
<tr>
<td><strong>Unsuccessful search:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear probe</td>
<td>1.39</td>
<td>2.50</td>
<td>5.09</td>
<td>13.00</td>
<td>5000.50</td>
</tr>
<tr>
<td>Quadratic probe</td>
<td>1.33</td>
<td>2.00</td>
<td>3.03</td>
<td>5.00</td>
<td>100.00</td>
</tr>
</tbody>
</table>
Separate Chaining

- We can eliminate collisions altogether if we store the keys outside the table.
  - **chains** – use linked lists to store keys that map to the same entry.
  - The hash table becomes an array of linked lists.
  - After mapping the key to an entry in the table, the linked list is searched for the key.
Separate Chaining
Efficiency: Separate Chaining

- Very efficient in the average case.
  - If there are $n$ keys and $m$ entries, the average list length is
    \[ \alpha = \frac{n}{m} \]
  - Successful search:
    \[ 1 + \frac{\alpha}{2} \]
  - Unsuccessful search:
    \[ 1 + \alpha \]
Hash Functions

- The efficiency of hashing depends in large part on the selection of a good hash function.
  - A “perfect” function will map every key to a different table entry.
    - This is seldom achieved except in special cases.
  - A “good” hash function distributes the keys evenly across the range of table entries.
Function Guidelines

- Important guidelines to consider in designing a hash function.
  - Computation should be simple.
  - Resulting index can not be random.
  - Every part of a multi-part key should contribute.
  - Table size should be a prime number.
Common Hash Functions

- **Division** – simplest for integer values.
  \[ h(key) = key \% M \]

- **Truncation** – some columns in the key are ignored.
  - Example: assume keys composed of 7 digits.
  - Use the 1\textsuperscript{st}, 3\textsuperscript{rd}, 6\textsuperscript{th} digits to form an index (M = 1000).
Common Hash Functions

- **Folding** – key is split into multiple parts then combined into a single value.
  - Given the key value 4873152, split it into three smaller values (48, 73, 152).
  - Add the values together and use with division.
Hashing Strings

- Strings can also be stored in a hash table.
  - Convert to an integer value that can be used with the division or truncation methods.
- Simplest approach: sum the ASCII values of individual characters.
  - Short strings will not hash to larger table entries.

- Better approach: use a polynomial.
  \[ s_0 a^{n-1} + s_1 a^{n-2} + \cdots + s_{n-3} a^2 + s_{n-2} a + s_{n-1} \]