Topic 2: Dictionary and Hash Tables

Read: Chpt. 1, 3, and 5, Weiss

Q: Given a large data set S of size n. How do we store, organize, and maintain this set of data objects such that “useful” operations can be performed on them efficiently?

Some Useful Operations:
   - Static Operations: find, findMin, findmax, …
   - Dynamic Operations: insert, delete, merge, …

ADT = {Data, Operations}.

Some Important ADTs:
   1. Dictionary: find, insert, delete, …
   2. Priority Queue: insert, findMax, deleteMax, …
   3. Double-Ended Queue: insert, findMax, findMin, deleteMax, deleteMin,…
   4. Concatenated Queue: insert, findMax, deleteMax, merge, …

Dictionary:
   A collection class of data objects supporting insert, delete and find operations effectively.

Hash Table:
   A simple yet powerful implementation of dictionary allowing us to store a vast amount of data in a set of locations.
Simplest Approach in Constructing a Hash Table:
Use an array (table) $B$ of size $m$ to store the objects in $S$.

**Q:** Given any object $x \in S$. How do we find a location in $B$ to store $x$?

**A:** Use a mapping function to convert the key/ID of each object in $S$ into a location in $B$.

**Hashing:** A process in finding the locations in a hash table to store a given set of data objects.

**Ideal Case:** Distinct objects correspond to distinct locations in $B$.

**In Practice:** Distinct objects may be sent to same location in $B$. 
Structure of a Hash Table:

1. *A set of m locations* (buckets), B[0..m–1]:
   Used to store a set S of n data objects with keys \{x_1, \ldots, x_n\}.

2. *A hash function* \( h: \{x_1, \ldots, x_n\} \rightarrow \{0, 1, \ldots, m–1\} \) with \( h(x_i) = j, 1 \leq i \leq n, 0 \leq j \leq m–1 \). For any given data object with key \( x_i \in S \), the location B[h(x_i)] will be used to store the given object, either internally or externally, if it is not already occupied by another object.

3. *A collision resolution scheme*:
   Used to determine either an alternate location in the table, or an auxiliary data structure is used, so as to store an item that is being *hashed* into an already occupied location.

**Q:** How do we store a data object in B?

**Hash Table Organization:**

1. **Open (External) hashing:**
   Locations store the addresses (reference locations) of objects.

2. **Closed (Internal) hashing:**
   Locations store actual objects.

**Remark:** In closed hashing, \( n \leq m \).
Recall that a collision occurs whenever two or more objects are hashed into the same location.

Some Simple Collision Resolution Schemes:
1. (Open) Hashing with Separate Chaining:
   Objects hashed into the same address are simply linked (chained) together.

![Diagram of hash table with separate chaining]

Consider the find(x) operation:
1. Compute h(x) to find location B[h(x)].
2. Search the linked structure at B[h(x)] sequentially for object with key x.

\[ T(n) = \text{cost in computing } h(x_i) + \text{cost in searching the linked list at } B[h(x_i)] \]

Q: How should a hash table be designed so that it will have good performance?
In the worst-case, a long chain may form to contain $\Theta(n)$ objects. To minimize searching time, each table location should contain a chain with roughly $n/m$ objects.

A “good” hash function is a function that
(1) Can be computed in $\Theta(1)$ time, and
(2) Reduce number of collisions by distributing the $n$ objects evenly over all $m$ locations with each location having roughly $n/m$ items.

Define $\text{load factor} = \lambda = \frac{n}{m}$.

Assuming that a good hash function $h$ is used, we have

**Unsuccessful search:**
\[
T_a(n) = \Theta(1) + \Theta(1)\left(\frac{n}{m}\right) \\
= \Theta\left(\frac{n}{m}\right) \\
= \Theta(\lambda)
\]

**Successful search:**
\[
T_a(n) = \Theta(1) + \Theta(1)[\left(\frac{n}{m}\right)/2] \\
= \Theta\left(\frac{n}{m}\right) \\
= \Theta(\lambda)
\]

When $m = \Theta(n)$, we have
\[
T_a(n) = \Theta\left(\frac{n}{m}\right) \\
= \Theta(1), \text{ which is the best possible!}
\]
Observe that $T_a(n) = \Theta(\lambda)$. As $n$ increases, $\lambda$ also increases, and efficiency of operations decreases.

**Q:** How do we design a “good” hash function?

**A Simple Hash Function:**
Use mod $m$ function. If $h(x_i)$ has int value, define $h(x_i) = x_i \mod m$, where $m$ is chosen to be a prime.

**Example:** Take $m = 7$. Insert 64, 26, 56, 72, 8, 36, and 42 into an initially empty hash table using separate chaining and hash function $h(x) = x \mod m$.

\[
\begin{align*}
64 \mod 7 &= 1, \\
26 \mod 7 &= 5, \\
56 \mod 7 &= 0, \\
72 \mod 7 &= 2, \\
8 \mod 7 &= 1, \\
36 \mod 7 &= 1, \\
42 \mod 7 &= 0.
\end{align*}
\]

**Remarks:**
1. Insertions are done at the beginning of a chain.
2. If duplicate element is not allowed, one must search the correspond chain for current element before insertion. Hence, insertions should be done at the end of a chain.
Possible Extension
Singly linked list can be replaced with more advanced data structures such as binary search tree and heap so as to speed up searching once a location is found.

Advantages of Hashing with Chaining
1. Simplicity (in concept and implementation).
2. Insertion is always possible; hence, a small table can be used to store any number of data (efficiency will suffer).

Disadvantages of Hashing with Chaining:
1. Can degenerate into a single chain with $T_w(n) = O(n)$.
2. Memory intensive: Need to implement/store pointers.
3. Slower speed: Indirect accessing data; need to follow pointers to data. Also, need to allocate and de-allocate dynamic memory.

Other Hashing Scheme:
A Simple Approach:
Assume that $h(x) = j$ and $B[j]$ is occupied. Search $B[j+1], B[j+2], \ldots, B[m-1], B[0], B[1], \ldots, B[j-1]$ sequentially to find the first available location to insert $x$. If no vacant location is found, report overflow. This collision resolution scheme is called Linear Probing, which is a special case of Open Addressing.
2. (Closed) Hashing with Open Addressing:
Given a hash function $h$, for some fixed integer $k$, define a sequence of $k$ hash functions $\{h_0, h_1, \ldots, h_{k-1}\}$ such that
$$h_i(x) = (h(x) + f_i) \mod m,$$
with $0 \leq i \leq k-1$, $f_0 = 0$.
This set of $k$ functions $\{f_0, f_1, \ldots, f_{k-1}\}$ is called collision resolution functions.

For any given object with key $x$, we compute
$$h_0(x) = h(x) + f_0 \mod m,$$
$$h_1(x) = h(x) + f_1 \mod m,$$
$$h_2(x) = h(x) + f_2 \mod m,$$
$$\ldots,$$
$$h_{k-1}(x) = h(x) + f_{k-1} \mod m,$$
so as to find the first available location for inserting $x$. If all these $k$ locations are occupied, $x$ will not be inserted and an error message will be generated.

Some Simple Open Addressing Schemes
2.1: Hashing with Linear Probing:
Define $f_i = i$, $\forall i$, $1 \leq i \leq k-1$, which is a family of linear functions, we have
$$h_0(x) = (h(x) + 0) \mod m = h(x),$$
$$h_1(x) = (h(x) + f_1) \mod m = (h(x) + 1) \mod m,$$
$$h_2(x) = (h(x) + f_2) \mod m = (h(x) + 2) \mod m,$$
$$\ldots,$$
$$h_{k-1}(x) = (h(x) + f_{k-1}) \mod m = (h(x) + k-1) \mod m.$$
Example: Take m = 7. Insert 64, 26, 56, 72, 8, 36, 42, using linear probing and hash function $h(x) = x \mod m$, into an initially empty hash table.

$64 \mod 7 = 1, \quad 26 \mod 7 = 5, \quad 56 \mod 7 = 0, \quad 72 \mod 7 = 2, \quad 8 \mod 7 = 1 \rightarrow 2 \rightarrow 3, \quad 36 \mod 7 = 1 \rightarrow 2 \rightarrow 3 \rightarrow 4, \quad 42 \mod 7 = 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6.$

Hash table using linear probing:

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Disadvantages in using Linear Probing:

Closed hashing with linear probing may result in **primary clustering**, which are blocks of occupied locations in B.
Primary clustering behaves like long chain and degrades the performance of the table!

**Remedy:** Use quadratic probing to eliminate primary clustering.
2.2: Hashing with Quadratic Probing:
Define \( f_i = i^2 \), \( \forall i, 1 \leq i \leq k-1 \), which is a family of quadratic functions, we have

\[
\begin{align*}
  h_0(x) &= (h(x) + 0^2) \mod m \\
  &= h(x), \\
  h_1(x) &= (h(x) + f_1) \mod m, \\
  &= (h(x) + 1^2) \mod m, \\
  h_2(x) &= (h(x) + f_2) \mod m, \\
  &= (h(x) + 2^2) \mod m, \\
  \vdots \\
  h_{k-1}(x) &= (h(x) + f_{k}) \mod m, \\
  &= (h(x) + (k-1)^2) \mod m.
\end{align*}
\]

Example: Take \( m = 7 \). Insert 64, 26, 56, 72, 8, 36, 42, using quadratic probing and hash function \( h(x) = x \mod m \), into an initially empty hash table.

Addresses Computation:

\[
\begin{align*}
  64 \mod 7 &= 1, \\
  26 \mod 7 &= 5, \\
  56 \mod 7 &= 0, \\
  72 \mod 7 &= 2, \\
  8 \mod 7 &= 1 \rightarrow 2 \rightarrow 5 \rightarrow 3, \\
  36 \mod 7 &= 1 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow \ldots
\end{align*}
\]
Hash table using quadratic probing:

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
</tr>
<tr>
<td>64</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>26</td>
</tr>
</tbody>
</table>

Problems with Closed Hashing:

1. Insertion may fail even though the table is not empty.
2. It may form clustering.
3. Problem in searching.

Consider the following example.

Example: Take \( m = 7 \). Insert 64, 56, 72, 8, followed by delete 64 and then delete 8, using linear probing and hash function \( h(x) = x \mod m \), into an initially empty hash table.

Addresses Computation:

\[
\begin{align*}
64 \mod 7 &= 1, \\
56 \mod 7 &= 0, \\
72 \mod 7 &= 2, \\
8 \mod 7 &= 1 \rightarrow 2 \rightarrow 3,
\end{align*}
\]
Hash table after inserting 64, 56, 72 and 8:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>64</td>
<td>72</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Hash table after deleting 64:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td></td>
<td>72</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Q: How do we delete 8?

Recall that 8 % 7 = 1 but B[1] is empty. Hence, we must continue searching for x even though an empty bucket is found!

Q: When can we stop searching?
**Observation:**

Two types of empty buckets:
1. A bucket that is always empty: Searching terminates.
2. A bucket that is emptied by deletion: Searching must continue.

**Data Structure for Bucket:**

Using an extra flag/Boolean field:
- \( \text{flag} = \text{true} \Rightarrow \) Bucket is emptied by deletion; searching must continues.
- \( \text{flag} = \text{false} \Rightarrow \) Bucket is always empty; searching terminates.

**Advantages of Closed Hashing with Open Addressing:**

1. Faster speed: No need to follow pointers.
2. Less memory consumption: No need to implement pointers.

**Disadvantages of Hashing with Quadratic Probing:**

1. Much more complex.
2. Can degenerate into primary or secondary clustering.
3. Deletion/Find operations are much more complex. Insertion is not always possible even though the table is not empty.
An Important Result in Hashing with Quadratic Probing:

**Theorem:** Let m be a prime number > 3. If quadratic probing is used in a table of size m, the first $\left\lfloor \frac{m}{2} \right\rfloor$ probes must all be distinct.

**Proof.** Assume not to obtain a contradiction. By assumption, there exist integers $i$ and $j$, $0 \leq i < j < \left\lfloor \frac{m}{2} \right\rfloor$, such that $h_i(x) = h_j(x)$, implying that

\[
\begin{align*}
    h(x) + i^2 &= h(x) + j^2 \pmod{m}, \\
    i^2 &= j^2 \pmod{m}, \\
    i^2 - j^2 &= 0 \pmod{m}, \\
    (i-j)(i+j) &= 0 \pmod{m}.
\end{align*}
\]

Since $m$ is an odd prime, either $(i-j) \equiv 0 \pmod{m}$, or $(i+j) \equiv 0 \pmod{m}$ must be true. Since $0 \leq i < j < \left\lfloor \frac{m}{2} \right\rfloor$, we have $i-j \neq 0$ and $i+j \neq m$. Hence, both cases will lead to a contradiction and the assertion is proven.

**Corollary:** When $m$ is prime and the table is at least half-empty; i.e., $\lambda < 1/2$, we can always insert a new item into a closed hash table using quadratic probing.
Conclusions:
1. To guarantee good performance in a hash table, a prime number should be chosen for \( m \) such that \( \lambda < 1 \) for open hashing and \( \lambda < 1/2 \) for closed hashing.
2. Must monitor \( \lambda \) during the lifetime of your hash table.
3. Hashing with open addressing (eg. quadratic probing) outperforms hashing with chaining only if implemented correctly!

Q: What happens when insertion/deletion become increasingly difficult?
   Need a new hash table with larger/smaller size!

Rehashing:
A process in hashing all the elements of an existing hash table \( H \) into a new hash table \( H^* \).

\[
H \leftrightarrow H^*
\]

\( \text{tableSize m (prime)} \leftrightarrow \text{tableSize m* (prime } \approx 2m) \)

Remark: Rehashing is a very expensive process and should only be performed infrequently.
Q: When do we rehash?
1. When $\lambda \to 1$ for open hashing and $\lambda \to 1/2$ for closed hashing.
2. Use a pre-specified $\lambda$ to determine when to rehash.
3. When insertion becomes increasingly difficult or fails.
4. When deletion becomes increasingly difficult.

3. Double Hashing:
Use two hash functions $h$ and $h^+$ such that the collision functions $f_i$'s are functions of $i$ and $h^+$.
Now, define $f_i = ih^+$. (or $i^2h^+$, or others)

Observe that
\[
\begin{align*}
h_0(x) &= (h(x) + 0h^+(x)) \mod m \\
&= h(x), \\
h_1(x) &= (h(x) + f_1) \mod m, \\
&= (h(x) + 1h^+(x)) \mod m, \\
h_2(x) &= (h(x) + f_2) \mod m, \\
&= (h(x) + 2h^+(x)) \mod m, \\
& \quad \vdots \\
h_k(x) &= (h(x) + f_k) \mod m, \\
&= (h(x) + kh^+(x)) \mod m.
\end{align*}
\]

A Simple $h^+$ Function:
Define $h^+(x) = R - (x \mod R)$, where $R < m$ is a prime.
Example: Take \( m = 7, R = 5 \). Insert 64, 26, 56, 72, 8, 36, 42, using double hashing with hash functions \( h(x) = x \mod m \), \( h^+(x) = R - (x \mod R) \), and \( f_i = ih^+ \), into an initially empty hash table.

**Addresses Computation:**
- \( 64 \mod 7 = 1 \), \( h^+(x) = R - (x \mod R) = 5 - (8 \mod 5) = 2 \), \( h_1(x) = (h(x) + 1h^+(x)) \mod m = (1 + 2) \mod 7 = 3 \),
- \( 26 \mod 7 = 5 \), \( h^+(x) = R - (x \mod R) = 5 - (3 \mod 5) = 4 \), \( h_1(x) = (h(x) + 1h^+(x)) \mod m = (5 + 4) \mod 7 = 5 \), \( h_2(x) = (h(x) + 2h^+(x)) \mod m = (5 + 8) \mod 7 = 2 \), \( h_3(x) = (h(x) + 3h^+(x)) \mod m = (5 + 12) \mod 7 = 6 \),
- \( 56 \mod 7 = 0 \), \( h^+(x) = R - (x \mod R) = 5 - (0 \mod 5) = 3 \), \( h_1(x) = (h(x) + 1h^+(x)) \mod m = (0 + 3) \mod 7 = 3 \), \( h_2(x) = (h(x) + 2h^+(x)) \mod m = (0 + 6) \mod 7 = 6 \), \( h_3(x) = (h(x) + 3h^+(x)) \mod m = (0 + 9) \mod 7 = 2 \), \( h_4(x) = (h(x) + 4h^+(x)) \mod m = (0 + 12) \mod 7 = 5 \), \( h_5(x) = (h(x) + 5h^+(x)) \mod m = (0 + 15) \mod 7 = 1 \), \( h_6(x) = (h(x) + 6h^+(x)) \mod m = (0 + 18) \mod 7 = 4 \).
Hash table using double hashing:

<table>
<thead>
<tr>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
</tr>
<tr>
<td>72</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>42</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>36</td>
</tr>
</tbody>
</table>