Hashing - Handling Collisions
Hashing - Recap

- We store the $n$ keys in a table containing $m$ entries
- We assume that the table size $m$ is at least a small constant factor larger than $n$
- We scatter the keys throughout the table using a pseudo-random hash function
  - $h(x) \in [0 \ldots m - 1]$
  - Store $x$ at entry $h(x)$ in the table
- Sometimes different keys collide: $x \neq y$, but $h(x) = h(y)$
What is the hash function? Recall common methods:

- Multiplicative hashing: \( h(x) = (ax) \mod p \mod m \) (for \( a \neq 0 \) and prime \( p \))
- Linear hashing: \( h(x) = (ax + b) \mod p \mod m \) (for \( a \neq 0 \) and prime \( p \))
- Polynomial: \( x = (c_0, c_1, c_2, c_3, \ldots) \), \( h(x) = (c_0 + c_1 p + c_2 p^2 + c_3 p^3 + \cdots) \mod m \)
- Universal hashing: \( h_{a,b}(x) = ((ax + b) \mod p) \mod m \) (where, \( a \) and \( b \) are random and \( p \) is prime)

How to resolve collisions? We will consider several methods:

- Separate chaining
- Linear probing
- Quadratic probing
- Double hashing
Separate Chaining

- Given a hash table `table[]` with \( m \) entries
- `table[i]` stores a linked list containing the keys \( x \) such that \( h(x) = i \)

\[
\begin{align*}
\text{insert("d")} & \quad h("d") = 1 \\
\text{insert("z")} & \quad h("z") = 4 \\
\text{insert("p")} & \quad h("p") = 7 \\
\text{insert("w")} & \quad h("w") = 0 \\
\text{insert("t")} & \quad h("t") = 4 \\
\text{insert("f")} & \quad h("f") = 0
\end{align*}
\]
Separate Chaining
Hash operations reduce to linked-list operations

- **insert**(x, v): Compute i=h(x), invoke table[i].insert(x,v)
- **delete**(x): Compute i=h(x), invoke table[i].delete(x)
- **find**(x): Compute i=h(x), invoke table[i].find(x)
Separate Chaining
Load factor and running time

- Given a hash table $\text{table}[m]$ containing $n$ entries
- Define load factor: $\lambda = \frac{n}{m}$
- Assuming keys are uniformly distributed, there are on average $\lambda$ entries per list
- Expected search times:
  - Successful search (key found): Need to search half the list on average
    \[S_{SC} = 1 + \frac{\lambda}{2}\]
  - Unsuccessful search (key not found): Need to search entire list
    \[U_{SC} = 1 + \lambda\]
Controlling the Load Factor

Rehashing

- Clearly, we want to keep load factors small, typically $0 < \lambda < 1$
- Select min and max load factors, $\lambda_{\text{min}}$ and $\lambda_{\text{max}}$, where $0 < \lambda_{\text{min}} < \lambda_{\text{min}} < 1$
- Define ideal load factor $\lambda_0 = \frac{(\lambda_{\text{min}} + \lambda_{\text{max}})}{2}$
- Rehashing (after insertion):
  - If insertion causes load factor to exceed $\lambda_{\text{max}}$
    - Allocate a new hash table of size $m' = \frac{n}{\lambda_0}$
    - Create a new hash function $h'$ for this table
    - Rehash all old entries into the new table using $h'$
  - After rehashing, the load factor is now $\frac{n}{m'} = \lambda_0$, that is, “ideal”
Rehashing

Example: $\lambda_{\text{min}} = \frac{1}{4}$, $\lambda_{\text{max}} = \frac{3}{4}$ and $\lambda_0 = \frac{1}{2}$

$\lambda = \frac{7}{8} > \lambda_{\text{max}}$ !!

Insert(a)

$n = 7$
$m = 8$

$\lambda = \frac{7}{14} = \lambda_0$
Controlling the Load Factor
Rehashing

- **Underflow**: Rehashing can also be applied when the load factor is too small
- **Rehashing** (after deletion):
  - If deletion causes load factor to be smaller than $\lambda_{min}$:
    - Allocate a new hash table of size $m' = \frac{n}{\lambda_0}$
    - Create a new hash function $h'$ for this table
    - Rehash all old entries into the new table using $h'$
  - After rehashing, the load factor is now $\frac{n}{m'} = \lambda_0$, that is, “ideal”
Rehashing - Amortized Analysis

How expensive is rehashing?

- Rehashing takes time - How bad is it?
- Rehashing takes $O(n)$ time, but once done we are good for a while
- Example:
  
  - Suppose $m = 1000$, $\lambda_{\min} = \frac{1}{4}$ and $\lambda_{\max} = \frac{3}{4}$, $(\lambda_0 = \frac{1}{2})$
  
  - After insertion, if $n > \lambda_{\max} = 750$, then we allocate a new table of size $m' = n/\lambda_0 \approx 1500$, and rehash the entries here
  
  - In order to overflow again, we need $n'/m' > \lambda_{\max}$
  
  - That is, we need $n' = 1125$ keys, or equivalently at least $1125 - 750 = 375$ insertions
  
  - Amortization: We charge the (expensive) work of rehashing to these (cheap) insertions
Rehashing - Amortized Analysis

How expensive is rehashing?

- **Theorem**: Assuming that individual hashing operations take $O(1)$ time each, if we start with an empty hash table, the amortized complexity of hashing using the above rehashing method with load factors of $\lambda_{min}$ and $\lambda_{max}$, respectively, is at most $1 + 2\lambda_{max}/(\lambda_{max} - \lambda_{min})$.

- **Proof**:
  - **Token-based argument**: Each time we perform a hash-table operation, we assess 1 unit for the actual operation and save $2\lambda_{max}/(\lambda_{max} - \lambda_{min})$ work tokens for future use.
  - **Two cases**: Overflow and underflow.
Rehashing - Amortized Analysis

How expensive is rehashing?

- **Token-based argument**: Each time we perform a hash-table operation, we assess 1 unit for the actual operation and save \(2\lambda_{max}/(\lambda_{max} - \lambda_{min})\) work tokens for future use.

- **Overflow**:
  - Current table has \(n \approx \lambda_{max}m\) entries. This is the cost of rehashing.
  - Just after the previous rehash, table contained \(n' = \lambda_0 m\) entries.
  - Since then, we performed at least \(n - n' = (\lambda_{max} - \lambda_0)m\) insertions.
  - By simple math, we have \(\lambda_{max} - \lambda_0 = \lambda_{max} - (\frac{\lambda_{max} + \lambda_{min}}{2}) = (\lambda_{max} - \lambda_{min})/2\).
  - Thus, the number of tokens collected is at least \((2\lambda_{max}/(\lambda_{max} - \lambda_{min})) \cdot (\lambda_{max} - \lambda_0)m = \lambda_{max}m \approx n\).
  - In summary, we have enough tokens to pay for rehashing!

- **Underflow**: (Similar...see lecture notes)
Open Addressing

- Separate chaining requires additional storage. Can we avoid this?
- Store everything in the table
- Requires that $n \leq m$, that is, $\lambda \leq 1$.
- Open Addressing:
  - Special entry "empty" indicates that this table entry is unused
  - To insert $x$, first check $\text{table}[h(x)]$. If empty, then store here
  - Otherwise, probe subsequent table entries until finding an empty location
  - Which entries to probe? Does it matter?
  - Yes! As the load factor approaches 1, some probe methods have good performance and others do not
Open Addressing - Linear Probing
Quick and dirty (maybe too quick and dirty)

- **Linear probing:**
  - If table[h(x)] is not empty, try h(x)+1, h(x)+2, ..., h(x)+j, until finding the first empty entry
  - **Wrap around if needed:** table[(h(x)+j) % m]

- **Example:**

```
  insert(d)  0  1  2  3  4  5
h("d") = 0
h("z") = 2
h("p") = 2
h("w") = 0
h("t") = 1

  insert(z)  2  3

  insert(p)  2  3

  insert(w)  2  3  4  5

  insert(t)  2  3  4  5
```
Open Addressing - Linear Probing

Secondary clustering

- **Primary clustering**: Clusters that occurs due to many keys hashing to the same location. (Should not occur if you use a good hash function)
- **Secondary clustering**: Clustering that occurs because collision resolution fails to disperse keys effectively
- **Bad news**: Linear probing is highly susceptible to secondary clustering

```
insert(d) = 0
h("d") = 0
h("z") = 2
h("p") = 2
h("w") = 0
h("t") = 1

insert(z) = 1
insert(p) = 2

insert(w) = 0
insert(t) = 1
```

!!
Open Addressing - Linear Probing

Secondary clustering

- **Expected search times:**
  - **Successful search** (key found): \( S_{LP} = \frac{1}{2} \left( 1 + \frac{1}{1-\lambda} \right) \)
  - **Unsuccessful search** (key not found): \( U_{LP} = \frac{1}{2} \left( 1 + \frac{1}{1-\lambda} \right)^2 \)
  - A table becomes full, \( \lambda \to 1 \), \( U_{LP} \) grows very rapidly
Open Addressing - Quadratic Probing

An attempt to avoid secondary clustering

- **Linear probing**: $h(x) + 1, 2, 3, \ldots, i$ clusters keys very close to the insertion point

- **Quadratic probing**: $h(x) + 1, 4, 9, \ldots, i^2$ disperses keys better, reducing clustering
Open Addressing - Quadratic Probing
An attempt to avoid secondary clustering

- **Quadratic probing:** $h(x) + 1, 4, 9, ... , i^2$ disperses keys better, reducing clustering
- Let $table[i].key$ and $table[i].value$ be the key and value
- **Cute trick:** $i^2 = (i - 1)^2 + (2i - 1)$. For next offset, add $2i + 1$ to previous offset
- Pseudo-code for `find(x)`:

```java
Value find(Key x) {
    int c = h(x) // initial probe location
    int i = 0 // probe offset
    while (table[c].key != empty) && (table[c].key != x) {
        c += 2*(++i) – 1 // next position
        c = c % m // wrap around
    }
    return table[c].value // return associated value (or null if empty)
}
```
Open Addressing - Quadratic Probing

An attempt to avoid secondary clustering

- **Quadratic probing:**
  - More formally, the probe sequence is \( h(x) + f(i) \), where \( f(i) = i^2 \)

- **Complete coverage?**
  - Does the probe sequence hit every possible table location?
  - No! For example, if \( m = 4 \), \( i^2 \mod 4 \) is either 0 or 1, never 2 or 3. (Try it!)

- **Any hope?** Can we select \( m \) so that quadratic probing hits all entries?
  - If \( m \) is prime of the form \( 4k + 3 \), quadratic probing will hit every table entry before repeating (source: Wikipedia - Related to quadratic residues)
  - If \( m \) is a power of 2, and we use \( f(i) = \frac{1}{2}(i^2 + i) \), quadratic probing will hit every table entry before repeating (source: Wikipedia)
Open Addressing - Quadratic Probing

An attempt to avoid secondary clustering

- **Theorem**: If quadratic probing is used, and the table size $m$ is a prime number, the first $\left\lfloor \frac{m}{2} \right\rfloor$ probe sequences are distinct.

- **Proof**:
  - By contradiction. Suppose that there exist $i, j$, such that $0 \leq i < j \leq \left\lfloor \frac{m}{2} \right\rfloor$ and $h(x) + i^2$ and $h(x) + j^2$ are equivalent modulo $m$.
  - Then the following equivalences hold mod $m$:
    \[ i^2 \equiv j^2 \iff i^2 - j^2 \equiv 0 \iff (i + j)(i - j) \equiv 0 \pmod{m} \]
  - Since $m$ is prime, both $i + j$ and $i - j$ must be multiples of $m$. But since $0 \leq i < j \leq \left\lfloor \frac{m}{2} \right\rfloor$, both quantities are smaller than $m$, and hence cannot be multiples. Contradiction!
Open Addressing - Double Hashing

Saved the best for last

- Linear probing suffers from secondary clustering
- Quadratic probing may fail to hit all cells
- Double hashing:
  - Probe offset is based on a second hash function \( g(x) \)
  - Probe sequence: \( h(x), h(x) + g(x), h(x) + 2g(x), h(x) + 3g(x), \ldots \)
Open Addressing - Double Hashing

Saved the best for last

- Double hashing:
  - Probe offset is based on a second hash function $g(x)$
  - Probe sequence: $h(x)$, $h(x) + g(x)$, $h(x) + 2g(x)$, $h(x) + 3g(x)$, ...

- Will this hit all entries before cycling?
  - Yes! If $m$ and $g(x)$ are relatively prime, share no common factors. (E.g., Making $g(x)$ a prime greater than $m$ guarantees this)
Double hashing has the best search times among all the methods covered so far:

- Successful search (key found): \( S_{DH} = \frac{1}{\lambda} \ln \left( \frac{1}{1-\lambda} \right) \)
- Unsuccessful search (key not found): \( U_{DH} = \frac{1}{1-\lambda} \)

Some sample values:

<table>
<thead>
<tr>
<th>( \lambda )</th>
<th>0.5</th>
<th>0.75</th>
<th>0.9</th>
<th>0.95</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>( U(\lambda) )</td>
<td>2.00</td>
<td>4.00</td>
<td>10.0</td>
<td>20.0</td>
<td>100</td>
</tr>
<tr>
<td>( S(\lambda) )</td>
<td>1.39</td>
<td>1.89</td>
<td>2.56</td>
<td>3.15</td>
<td>4.65</td>
</tr>
</tbody>
</table>
Open Addressing - Deletion

Deletion requires care!

- Deleted entries can create the illusion we are at the end of the probe sequence

```
insert("a")
```

```
delete("f")
```

```
find("a")
```

"a" not found!
Open Addressing - Deletion
Quick and dirty fix

- Special entry “deleted”: The item at this location has been deleted
  - When searching: don’t stop here
  - When inserting: a key can be placed here

```
delete("f")
find("a")
```

Hashing - Further Refinements

- Hashing has been around a long time, and numerous refinements have been proposed.
- Example: Brent’s Method
  - When using double hashing, multiple probe sequences (with different values of \( g(x) \)) may overlap at a common cell of the hash table, say \( \text{table}[i] \).
  - One of these sequence places its key in \( \text{table}[i] \), and for the other, this wasted cell just adds to the search times.
  - To improve average search times, we should give ownership of the cell to the longer of the two probe sequences (and move the other key later in its probe sequence).
  - Brent’s algorithm optimizes the placement of keys in overlapping probe sequences.
Summary

- Hashing - The fastest implementation of the dictionary data type
  - Does **not** support ordered operations (min, max, range query, kth smallest, ...)
  - Key elements:
    - Hash function - Linear, Polynomial, Universal hashing
    - Collision resolution
      - Separate chaining
      - Open Addressing:
        - Linear probing
        - Quadratic probing
        - Double hashing
    - Analysis: Load factors, rehashing and amortized efficiency