CSE 20312
Open Addressing
Goals for today (learning objs)

1. What is the difference between separate chaining and open addressing?

2. What are some different probing sequences used for open addressing?

3. What is a hash table’s load factor and why is it important?
Open Addressing
Open Addressing: Concept

Instead of a container, each bucket only stores one value.

When there is a collision, we use a probing sequence to search through alternate buckets until:

Find the target
Find an unused bucket

To mark a bucket as unused, we need some sort of sentinel.

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Open Addressing: Pseudo-Code

**Insert**(table, value):
   bucket = Locate(table, value)
   table[bucket] = value

**Lookup**(table, value):
   bucket = Locate(table, value)
   return table[bucket] == value

**Locate**(table, value):
   bucket = Hash(value) % tsize
   while table[bucket] != value and table[bucket] != SENTINEL:
       bucket = (bucket + 1) % tsize
   return bucket

Consider sequence of values:

7, 1, 5, 10, 12, 13

Collision with 5 and 13!
Open Addressing: Implementation

Demonstrate `open_addressing`:
- Students must implement `insert` and `find`
- `nitems`: 5, 10, 20
- Students must fix `infinite loop`

Considerations:
- `probe sequence`?
- `resizing`?
- `sentinel`?
- `infinite loops`?!?!?
- `deletion`?

Source Code
Open Addressing: Pseudo-Code ++

**Locate**(table, value):
  bucket = Hash(value) % tsize
  step = 0
  **while** table[bucket] != value and table[bucket] != SENTINEL and step < tsize:
    bucket = (bucket + ssize) % tsize
    step = step + ssize
  **return** bucket **if** step < tsize **else** throw Exception()

Linear probing sequence is now parameterized
Avoid infinite loop
Open Addressing: Deletion

**Deletion** is problematic because it breaks probing sequence.

Rather than actually deleting the item, mark entry as "deleted"

When doing a **Lookup**, we skip the deleted entries.

When doing an **Insert**, we treat the deleted entries as empty **buckets**.

Consider the operations:
- Insert(2)
- Insert(10)
- Insert(18)
- Remove(10)
- Lookup(18)
Probe Sequences
Probe Sequences: Linear Probing

To search the table for a free location, we use the following formula:

\[
\text{Bucket}(i) = (\text{Hash}(v) + i \times \text{StepSize}) \mod \text{TableSize}
\]

Normally, \text{StepSize} = 1

\text{StepSize} is normally relatively prime to \text{TableSize}

\text{TableSize} is normally prime to provide better distribution

Good for \text{caching}, but susceptible to \text{clustering}
To search the table for a free location, we use the following formula:

\[
\text{Bucket}(i) = (\text{Hash}(v) + c_1 \cdot i + c_2 \cdot i^2) \mod \text{TableSize}
\]

\(c_2\) is non-zero

Common choices are \(c_1 = c_2 = \frac{1}{2}\), \(c_1 = c_2 = 1\), \(c_1 = 0\) and \(c_2 = 2\)

Better avoids clustering, but can miss empty buckets
Probe Sequences: **Double Hashing**

To search the table for a free location, we use the following formula:

\[
\text{Bucket}(i) = (\text{Hash}_1(v) + i \times \text{Hash}_2(v)) \mod \text{TableSize}
\]

Use two hash functions rather than just one. Using a second independent hash function minimizes repeated *collisions* and *clustering*. 
Probe Sequences: **Cuckoo Hashing**

The table uses multiple hash functions:

During **Lookup**, we try location at $\text{Hash}_1$, $\text{Hash}_2$, etc. until we find item or exhaust set of hash functions.

During **Insertion**, we try locations $\text{Hash}_1$, $\text{Hash}_2^2$, etc. If there is **collision**, we replace current entry with new value and then rehash old entry with next hash function (repeat until no more **collisions** or exhausted table).

This scheme allows for very **high space utilization** and **worst case constant lookup time**.
Complexity Analysis
Complexity Analysis: Load Factor

Load Factor is measure of table occupancy:

\[ \text{alpha} = \frac{n}{m} \]

\(n\) is number of elements (including “deleted” entries)
\(m\) is number of buckets

Assuming hash function is uniform:

Separate chaining: average size of bucket is \(\text{alpha}\), so operations are \(O(1 + \text{alpha})\)

Open addressing: probability of collision is \(\text{alpha}\), so average number of tries is \(\text{alpha} / (1 - \text{alpha})\), so operations are \(O(1 / (1 - \text{alpha}))\)

DON’T LET ALPHA GET BIG!
Complexity Analysis: Resizing

As long as we maintain an upper bound on alpha, we can have constant time operations.

When load factor is too high (\(\frac{2}{3}\), \(\frac{3}{4}\)), we need to resize by a constant factor (2).

Just as with a vector, the amortized cost of resizing is \(O(1)\).
Comparison
Chaining vs Open Addressing

**Separate Chaining**

Straight-forward to implement

More tolerant of high load factor

Poor locality (cache behavior)

**Open Addressing**

Requires hash function to be uniform and minimize clustering

Requires limited load factor (and thus memory waste)

Less overhead of memory allocation
Hash Functions: Better Hashes

Lookup2 and Lookup3
SpookyHash
MurmurHash
CityHash and FarmHash
Hash Functions: Visualizations

Dumb Hash

Dumber Hash
Hash Functions: Visualizations

STL Hash

SpookyHash
Hash Functions: Properties

A good hash function is:
  - Fast (quick to compute)
  - Provides uniform mapping of elements regardless of values

A hash function for a hash table is not the same as a cryptographic hash!
  - Cryptographic hash functions go to extremes to minimize collisions, and thus are much slower