CSE 373: Hash Tables
(open addressing and applications)

Chapter 5

Open Addressing

Goal: Use available space in table to store collisions rather than lists or resizing
- linear probing
- quadratic probing
- double hashing
Open Addressing Requirements

- The selection of alternate slots must be recomputable and deterministic
  - so that we can \texttt{Find()} data that we’ve inserted
- Deletion from the table must be “lazy”
  - similar to binary search trees
  - don’t remove data, simply mark it as being deleted

Open Addressing: General Form

Open addressing is generally expressed as:

$$(\text{hash(key)} + f(i)) \mod \text{tablesize}, \text{ for } i = 0, 1, 2, \ldots$$

The hashing procedure is therefore:

1) Try $(\text{hash(key)} + f(0)) \mod \text{tablesize}$
2) If it’s full, try $(\text{hash(key)} + f(1)) \mod \text{tablesize}$
3) Continue until you find an empty slot

Design decision: what to use for $f()$?
### Linear Probing

Uses $f(i) = i$

<table>
<thead>
<tr>
<th>Insert (T, SPAN)</th>
<th>Insert (T, MATH)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>EE</td>
<td>EE</td>
</tr>
<tr>
<td>SPAN</td>
<td>SPAN</td>
</tr>
<tr>
<td>ACMS</td>
<td>ACMS</td>
</tr>
</tbody>
</table>

$h(\text{SPAN}) = 2$  
$h(\text{MATH}) = 2$

*Note that if there is an open slot in the table, linear probing will always find it (eventually)*

---

### Finding, Deleting w/ Linear Probing

<table>
<thead>
<tr>
<th>Find(T, SPAN)</th>
<th>Delete(T, SPAN)</th>
<th>Find(T, MATH)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EE</td>
<td>EE</td>
<td>EE</td>
</tr>
<tr>
<td>SPAN</td>
<td>SPAN</td>
<td>SPAN</td>
</tr>
<tr>
<td>MATH</td>
<td>MATH</td>
<td>MATH</td>
</tr>
<tr>
<td>ACMS</td>
<td>ACMS</td>
<td>ACMS</td>
</tr>
</tbody>
</table>

$h(\text{SPAN}) = 2$  
$h(\text{MATH}) = 2$
**Primary Clustering**

Linear probing has the tendency to result in clusters of data in the table
- increases search time for values hashing to that area

![Cluster Diagram]

**Quadratic Probing**

Uses $f(i) = i^2$

<table>
<thead>
<tr>
<th>Insert $(T, \text{SPAN})$</th>
<th>Insert $(T, \text{MATH})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE</td>
<td>EE</td>
</tr>
<tr>
<td>SPAN</td>
<td>SPAN</td>
</tr>
<tr>
<td>ACMS</td>
<td>ACMS</td>
</tr>
<tr>
<td>$h(\text{SPAN}) = 2$</td>
<td>$h(\text{MATH}) = 2$</td>
</tr>
</tbody>
</table>

![Probing Diagram]
**Quadratic Probing: Evaluation**

- *Intuition:* spreads things out more, so primary clustering should not be as much of a problem.

- It can be proven that quadratic probing is guaranteed to find a free slot if...
  - number of slots is prime
  - table is less than half full
  - (therefore, resize when $\lambda = 0.5$)

**Double Hashing**

$f(i) = i \cdot \text{hash}_2(\text{key})$

Intuition: since good hash functions result in fairly random distributions, this spreads values out in a less predictable pattern.
Applications: Compilers

Compilers use hash tables to store information about all user-defined identifiers

```c
void getname() {
    int i;
    char name[20];
    ...
}
```

Applications: AI

- Create a hash function for a game’s position
- Store “good moves” from each position as they are discovered
- While playing, can quickly check if there is a good move from the current position