Today

• Hash Table Addressing

Assignments

• HW 6 out
Resolving collisions (on insert) in hash tables

Assume the hash function is `hascode % 97` for a 97-element table

- \( h(236) = 42 \) ... insert at index 42
- \( h(270) = 76 \) ... insert at index 76
- ...
- \( h(527) = 42 \) ... collision!

The two general approaches for dealing with collisions:

**Open Addressing**: on collision, search for another location
- we’ll look at different ways to “find another location”

**Restructure the Table**: add more room to fit new item ... HW-6
- we’ll look at “separate chaining” and table “resizing”
### Open Addressing Approach 1: Linear Probing

**Basic Idea:**
- hash to “bucket” index
- bucket may be empty or not empty
- If occupied, move to next empty bucket
- If get to end of table, keep searching at 0

![Linear Probing Diagram]

Linear probing can result in large “primary” clusters of keys
- i.e., large sequences of non-empty buckets

Maintain two types of empty “buckets” (for searching)
- empty-since-start ... empty since start of hash table
- empty-after-removal ... item removed making bucket empty
How linear probing works ...

Insert:

- Linear probe (sequentially) until first empty bucket found (either type)
- Insert at first empty bucket
- Note: The table could be full! (which means we can’t insert)

Removal:

- Use similar approach to find key item
- Stop at empty-since-start bucket (item not in table)
- Mark item as empty-after-removal

Search:

- Go to hashed index
- Start searching sequentially for key
- Stop when an empty-since-start bucket is found or searched entire table
Open Addressing Approach 2: Quadratic Probing

Basic Idea:

- similar to linear probing
- but probe “quadratic” sequences
- \( i + 1, i + 2^2, i + 3^2, i + 4^2, \) and so on
- helps “spread out” primary clusters

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>k3</td>
</tr>
<tr>
<td>3</td>
<td>k1</td>
</tr>
<tr>
<td>4</td>
<td>k2</td>
</tr>
</tbody>
</table>
```

```
Table
```

```
i + 1
```

```
i + 4
...```

```
i + 9
```

```
k4
```

```
“probe”
```

Creates “secondary” clusters since collisions use same quadratic sequences
Open Addressing Approach 3: Double Hashing

Basic Idea:

- use a secondary hash function $h_2$ to determine size of sequence “steps”
- where number of steps depend on the key
- can further help to reduce clustering
Open Addressing Approach 4: Cuckoo Hashing

Basic Idea:
- use two hash functions $h_1$ and $h_2$ denoting two possible buckets for each key
- for keys $k$ and $k'$ low odds that both $h_1(k) = h_1(k')$ and $h_2(k) = h_2(k')$

To insert a new key value $k$ ...
- if one of the buckets $h_1(k)$ or $h_2(k)$ is empty place key $k$ there
- if both buckets are full:
  - “evict” one of the keys $k'$ and place $k$ in the bucket
  - now re-insert $k'$
  - repeat until all keys inserted or some number of keys tried
  - if insert fails, re-hash all keys using new $h_1, h_2$ hash functions
Alternative Approach: Restructuring the Table

Instead of probing, allow multiple items to be stored at each index

“Separate Chaining”

- Each array index has its own **linked list** of elements (the “chain”)
- The list grows and shrinks with collisions and removals
Benefits of Separate Chaining

Ideally, insert, remove, search are $O(1)$

However, collisions increase the cost

- Cost depends on the load factor ... how full the table is
  
  \[ \text{load\_factor} = \frac{\text{collection\_size}}{\text{table\_capacity}} \]

- As table fills, chance of collision increases

- And hashing efficiency decreases (e.g., using open addressing)

Cost of separate chaining

- Insertion is still $O(1)$ (e.g., insert at front of linked list chains)

- Removal and search require navigating linked list chain ...
  - cost to search chains depends on number of collisions
  - assuming a “good” hash function (distributes keys)
  - load_factor < 100% implies at most length 1 chains
  - load_factor > 100% implies average chain length over 1

- Worst case find and remove cost is $O(n)$ since ...
  
  \[ \text{avg\_chain\_len} = \text{load\_factor} = \frac{n}{\text{table\_capacity}} \]

- But in practice (like with quick sort), hash tables provide efficient search