CS180 - Hashing (part 3)

Announcements
- Checkpoint on Monday
- 1 more HW due final day of class
In BitStream

BinaryTree< ? > myTree;

int input;

InBitStream variable;

variable.open("banana.my.zip");

input = variable.read(); // will be 0 for root

if (input == 0)
    myTree.createRoot();

(note - can draw tree!)
### Data Storage - Dictionary:

- **Ex.:**
  - **Key:*** Locker #
  - **Name:**
    - 26: Dan
    - 355: Kevin
    - 101: Tracy
    - 53: Nitish
    - 201: David

- Insert
- Find
- Remove

We want to be able to retrieve a name quickly when given a locker number.

\[
\begin{align*}
  n &= \text{# of people} \\
  m &= \text{# of lockers}
\end{align*}
\]

\[
\begin{align*}
  n &= m \Rightarrow n = m
\end{align*}
\]
Good hash functions:
- Are fast: \( O(1) \)
- Don't have collisions (these are unavoidable, but we want to minimize)

```
key space (size m)
```

```
(k,e) -> h(k) -> table
\[ h(k) \approx \frac{N-1}{2} \]
```

```
h < N << m
```
Step 1: Turn key into an integer
   Cyclic permutations
   or polynomial

Step 2: Compression map
takes us to a value in $[0, \ldots, N-1]$%
   \( \% N \)
   MAD, etc.
Collisions

Can we ever totally avoid collisions?

No

$m$ is bigger than $n$ or $N$
Step 3: Handle collisions (gracefully & quickly)

So how can we handle collisions?

[Hint: Do we have any data structures that can store more than 1 element?]

Possibilities:
- Store in auxiliary DS, such as
  - Vector
  - List
  - AVL trees
Linear Probing

Instead of lists, if we hash to a full spot, just keep checking next spot (as long as the next spot is not empty).
Example

\[ h(k) = k \mod 11 \]

\[ h(k) = (k+i) \mod 11 \quad i = 0, 1, 2, \ldots \]

**Insert:**
- (12, E)
- (21, R)
- (37, I)
- (26, N)
- (116, C)
- (5, H)
- (15, A)
- (18, M)

**Remainder:**
- 12 mod 11 = 1
- 21 mod 11 = 10
- 37 mod 11 = 4
- 26 mod 11 = 4
- 116 mod 11 = 5
- 5 mod 11 = 5
- 15 mod 11 = 4

**Find:** (48, -)
Running Time for Linear Probing

Insert: $O(n)$

Remove: $O(n)$ dirty bit

Find: $O(n)$
Issues with linear probing

- "Clusters" form
  - worse if hash function is not "good"
  - terrible when array nears 1/2 full

- Removing doesn't actually reduce # of elements - just sets the "dirty" bit.
Quadratic Probing

Linear probing checks \( A[h(k)+j \mod \lfloor N/j \rfloor] \) if previous spot is full (for \( j = 1, 2, \ldots \)).

To avoid primary clusters, try:

\[
A[h(k) + j^2 \mod \lfloor N/j \rfloor]
\]

where \( j = 0, 1, 2, 3, 4, \ldots \)

- If \( A[h(k)] \) is full:
  - Try \( A[h(k) + 1] \)
  - Try \( A[h(k) + 2^2] \)
  - Try \( A[h(k) + 9] \)
Example

\[ h(k) = k \mod 11 \]

Secondary clusters

<table>
<thead>
<tr>
<th>EM</th>
<th>INC</th>
<th>AIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Insert:

- (12, E)
- (21, R)
- (37, I)
- (26, N)
- (16, C)
- (5, H)
- (15, A)
- (4, M)

\[ h(12) = 1 \]
\[ h(21) = 10 \]
\[ h(37) = 4 \]
\[ h(26) = 4 \rightarrow 4 + 1^2 \]
\[ h(16) = 5 \rightarrow 5 + 1^2 \]
\[ h(5) = 5 \rightarrow 5 + 1^2 \rightarrow 5 + 2^2 \]
\[ h(15) = 4 \rightarrow 4 + 1^2 \rightarrow 4 + 2^2 \]
\[ h(4) = 4 \rightarrow 5 \rightarrow 8 \rightarrow 4 + 3^2 \]
Issues with Quadratic Probing:

- Can still cause secondary clustering
- N really must be prime for this to work
- Even with N prime, starts to fail when array gets half full
- Can fail entirely even if array not full

(Runtimes are essentially the same)
Secondary Hashing

- Try $A[h(k)]$

- If full, try $A[h(k)] + f(j) \mod NJ$
  for $j = 1, 2, 3, ...$

where $f(j) = j \cdot e(k)$ with $e$ a different hash function

Suppose $h'(k) = 4$
Load Factors

Separate chaining actually works as well as most others in practice, although it does use more space.

Most of these methods only work well if \( \frac{n}{N} < 0.5 \).

(Even chaining starts to fail if \( \frac{n}{N} > 0.9 \))

\( \frac{n}{N} \) load factor
Rehashing

Because we need \( \frac{n}{N} < 0.5 \) most hash code checks if the array has become more than half full.

If so, it stops and recomputes everything for a larger \( N \), usually at least twice as big.

(Still not too bad in an amortized sense—think vectors.)