Announcements

• Project 3 is due today at 11pm
• Project 4 is out tonight, due Wed. May 13 at 11pm
  – You can work with a partner for project 4
  – Partner selection due next Wednesday
• Midterm next Wednesday in class
Midterm, in-class Wednesday May 6th

• In class, closed notes, closed book.

• Covers everything up to and including hashing.
  – Stacks, queues
  – Induction
  – Asymptotic analysis and Big-Oh
  – Dictionaries, BSTs, AVL Trees
  – Binary heaps and Priority Queues
  – Disjoint sets and Union-Find
  – Hash Tables and Collisions

• Information, sample past exams and solutions posted online.
Hash Tables: Review

• Aim for constant-time (i.e., $O(1)$) find, insert, and delete
  – “On average” under some reasonable assumptions

• A hash table is an array of some fixed size
  – But growable as we’ll see
Collision resolution

Collision:
When two keys map to the same location in the hash table

We try to avoid it, but number-of-keys exceeds table size

So hash tables should support collision resolution
– Ideas?
Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42
with mod hashing
and TableSize = 10
Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42 with mod hashing
and $\text{TableSize} = 10$
Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42 with mod hashing
and **TableSize** = 10
Separate Chaining

Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42 with mod hashing
and TableSize = 10
Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42
with mod hashing
and TableSize = 10
Separate Chaining

Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:
insert 10, 22, 107, 12, 42
with mod hashing
and TableSize = 10
Thoughts on chaining

• Worst-case time for find?
  – Linear
  – But only with really bad luck or bad hash function
  – So not worth avoiding (e.g., with balanced trees at each bucket)

• Beyond asymptotic complexity, some “data-structure engineering” may be warranted
  – Linked list vs. array vs. chunked list (lists should be short!)
  – Move-to-front
  – Maybe leave room for 1 element (or 2?) in the table itself, to optimize constant factors for the common case
    • A time-space trade-off…
Time vs. space (constant factors only here)
More rigorous chaining analysis

Definition: The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}} \quad \leftarrow \text{number of elements}$$

Under chaining, the average number of elements per bucket is ___
More rigorous chaining analysis

Definition: The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}} \quad \leftarrow \text{number of elements}$$

Under chaining, the average number of elements per bucket is $\lambda$

So if some inserts are followed by random finds, then on average:

- Each unsuccessful `find` compares against ____ items
More rigorous chaining analysis

Definition: The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}} \quad \leftarrow \text{number of elements}$$

Under chaining, the average number of elements per bucket is $\lambda$

So if some inserts are followed by random finds, then on average:

- Each unsuccessful \texttt{find} compares against $\lambda$ items
- Each successful \texttt{find} compares against _____ items
More rigorous chaining analysis

Definition: The load factor, $\lambda$, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}} \quad \leftarrow \text{number of elements}$$

Under chaining, the average number of elements per bucket is $\lambda$

So if some inserts are followed by random finds, then on average:
• Each unsuccessful `find` compares against $\lambda$ items
• Each successful `find` compares against $\lambda/2$ items

So we like to keep $\lambda$ fairly low (e.g., 1 or 1.5 or 2) for chaining
Alternative: Use empty space in the table

- Another simple idea: If $h(key)$ is already full,
  - try $(h(key) + 1) \mod \text{TableSize}$. If full,
  - try $(h(key) + 2) \mod \text{TableSize}$. If full,
  - try $(h(key) + 3) \mod \text{TableSize}$. If full...

- Example: insert 38, 19, 8, 109, 10
Alternative: Use empty space in the table

- Another simple idea: If $h(key)$ is already full,
  - try $(h(key) + 1) \% TableSize$. If full,
  - try $(h(key) + 2) \% TableSize$. If full,
  - try $(h(key) + 3) \% TableSize$. If full...

- Example: insert 38, 19, 8, 109, 10

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Alternative: Use empty space in the table**

- Another simple idea: If $h(key)$ is already full,
  - try $(h(key) + 1) \mod TableSize$. If full,
  - try $(h(key) + 2) \mod TableSize$. If full,
  - try $(h(key) + 3) \mod TableSize$. If full...

- Example: insert 38, 19, 8, 109, 10
Alternative: Use empty space in the table

- Another simple idea: If $h(\text{key})$ is already full,
  - try $(h(\text{key}) + 1) \% \text{TableSize}$. If full,
  - try $(h(\text{key}) + 2) \% \text{TableSize}$. If full,
  - try $(h(\text{key}) + 3) \% \text{TableSize}$. If full...

- Example: insert 38, 19, 8, 109, 10
Alternative: Use empty space in the table

- Another simple idea: If $h(key)$ is already full,
  - try $(h(key) + 1) \% \text{TableSize}$. If full,
  - try $(h(key) + 2) \% \text{TableSize}$. If full,
  - try $(h(key) + 3) \% \text{TableSize}$. If full...

- Example: insert 38, 19, 8, 109, 10

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>109</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Probing hash tables

Trying the next spot is called probing (also called open addressing)
  – We just did linear probing
    • $i^{th}$ probe was $(h(key) + i) \mod TableSize$
  – In general have some probe function $f$ and use $h(key) + f(i) \mod TableSize$

Open addressing does poorly with high load factor $\lambda$
  – So want larger tables
  – Too many probes means no more $O(1)$
Other operations

**insert** finds an open table position using a probe function

What about **find**?
- Must use same probe function to “retrace the trail” for the data
- Unsuccessful search when reach empty position

What about **delete**?
- **Must** use “lazy” deletion. Why?
  - Marker indicates “no data here, but don’t stop probing”
- Note: **delete** with chaining is plain-old list-remove
(Primary) Clustering

It turns out linear probing is a \textit{bad idea}, even though the probe function is quick to compute (which is a good thing)

Tends to produce \textit{clusters}, which lead to long probing sequences

- Called \textit{primary clustering}
- Saw this starting in our example
Analysis of Linear Probing

• Trivial fact: For any $\lambda < 1$, linear probing will find an empty slot
  – It is “safe” in this sense: no infinite loop unless table is full

• Non-trivial facts we won’t prove:
  Average # of probes given $\lambda$ (in the limit as $\text{TableSize} \to \infty$)
  – Unsuccessful search:
    \[
    \frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)^2} \right)
    \]
  – Successful search:
    \[
    \frac{1}{2} \left( 1 + \frac{1}{1 - \lambda} \right)
    \]

• This is pretty bad: need to leave sufficient empty space in the table to get decent performance (see chart)
**In a chart**

- Linear-probing performance degrades rapidly as table gets full
  – (Formula assumes “large table” but point remains)

- By comparison, chaining performance is linear in \( \lambda \) and has no trouble with \( \lambda > 1 \)
Quadratic probing

• We can avoid primary clustering by changing the probe function
  \((h(key) + f(i)) \mod \text{TableSize}\)

• A common technique is quadratic probing:
  \(f(i) = i^2\)
  – So probe sequence is:
    • 0\(^{th}\) probe: \(h(key) \mod \text{TableSize}\)
    • 1\(^{st}\) probe: \((h(key) + 1) \mod \text{TableSize}\)
    • 2\(^{nd}\) probe: \((h(key) + 4) \mod \text{TableSize}\)
    • 3\(^{rd}\) probe: \((h(key) + 9) \mod \text{TableSize}\)
    • …
    • \(i^{th}\) probe: \((h(key) + i^2) \mod \text{TableSize}\)

• Intuition: Probes quickly “leave the neighborhood”
Quadratic Probing Example

Table Size = 10
Insert:
89
18
49
58
79
## Quadratic Probing Example

<table>
<thead>
<tr>
<th>Table Size: 10</th>
<th>Insert:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>89</td>
</tr>
<tr>
<td></td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>79</td>
</tr>
</tbody>
</table>

```plaintext
0
1
2
3
4
5
6
7
8
9
```

89
Quadratic Probing Example

TableSize = 10
Insert:
89
18
49
58
79
Quadratic Probing Example

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>49</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td>89</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>49</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>58</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>79</td>
<td></td>
</tr>
</tbody>
</table>

TableSize=10
Insert:
89
18
49
58
79
### Quadratic Probing Example

Table Size = 10

Insert:
89
18
49
58
79

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>58</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Quadratic Probing Example

TableSize = 10
Insert:
89
18
49
58
79
Another Quadratic Probing Example

TableSize = 7

Insert:
76  (76 % 7 = 6)
40  (40 % 7 = 5)
48  (48 % 7 = 6)
5   ( 5 % 7 = 5)
55  (55 % 7 = 6)
47  (47 % 7 = 5)
Another Quadratic Probing Example

TableSize = 7

Insert:
76 \quad (76 \; \% \; 7 = 6)
40 \quad (40 \; \% \; 7 = 5)
48 \quad (48 \; \% \; 7 = 6)
5 \quad (5 \; \% \; 7 = 5)
55 \quad (55 \; \% \; 7 = 6)
47 \quad (47 \; \% \; 7 = 5)
Another Quadratic Probing Example

TableSize = 7

<table>
<thead>
<tr>
<th>Insert</th>
<th>Hash Value</th>
<th>Probing Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>6</td>
<td>(76 % 7 = 6)</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>(40 % 7 = 5)</td>
</tr>
<tr>
<td>48</td>
<td>6</td>
<td>(48 % 7 = 6)</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>(5 % 7 = 5)</td>
</tr>
<tr>
<td>55</td>
<td>6</td>
<td>(55 % 7 = 6)</td>
</tr>
<tr>
<td>47</td>
<td>5</td>
<td>(47 % 7 = 5)</td>
</tr>
</tbody>
</table>
Another Quadratic Probing Example

TableSize = 7

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</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>40</td>
</tr>
<tr>
<td>6</td>
<td>76</td>
</tr>
</tbody>
</table>

Insert:
- 76 \((76 \% 7 = 6)\)
- 40 \((40 \% 7 = 5)\)
- 48 \((48 \% 7 = 6)\)
- 5 \((5 \% 7 = 5)\)
- 55 \((55 \% 7 = 6)\)
- 47 \((47 \% 7 = 5)\)
Another Quadratic Probing Example

Table Size = 7

Insert:

<table>
<thead>
<tr>
<th></th>
<th>48</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>76</td>
</tr>
</tbody>
</table>

76  (76 % 7 = 6)
40  (40 % 7 = 5)
48  (48 % 7 = 6)
  5  ( 5 % 7 = 5)
55  (55 % 7 = 6)
47  (47 % 7 = 5)
Another Quadratic Probing Example

TableSize = 7

Insert:

<table>
<thead>
<tr>
<th>Insert</th>
<th>(Insert % TableSize)</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>(76 % 7 = 6)</td>
</tr>
<tr>
<td>40</td>
<td>(40 % 7 = 5)</td>
</tr>
<tr>
<td>48</td>
<td>(48 % 7 = 6)</td>
</tr>
<tr>
<td>5</td>
<td>(5 % 7 = 5)</td>
</tr>
<tr>
<td>55</td>
<td>(55 % 7 = 6)</td>
</tr>
<tr>
<td>47</td>
<td>(47 % 7 = 5)</td>
</tr>
</tbody>
</table>

```
0     48
1
2     5
3     55
4
5     40
6     76
```
**Another Quadratic Probing Example**

TableSize = 7

<table>
<thead>
<tr>
<th>Insert:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>(76 % 7 = 6)</td>
</tr>
<tr>
<td>40</td>
<td>(40 % 7 = 5)</td>
</tr>
<tr>
<td>48</td>
<td>(48 % 7 = 6)</td>
</tr>
<tr>
<td>5</td>
<td>(5 % 7 = 5)</td>
</tr>
<tr>
<td>55</td>
<td>(55 % 7 = 6)</td>
</tr>
<tr>
<td>47</td>
<td>(47 % 7 = 5)</td>
</tr>
</tbody>
</table>

Doh! For all $n$, $((n*n) + 5) \mod 7$ is 0, 2, 5, or 6

- Excel shows takes “at least” 50 probes and a pattern
- Proof (like induction) using $(n^2+5) \mod 7 = ((n-7)^2+5) \mod 7$
  - In fact, for all $c$ and $k$, $(n^2+c) \mod k = ((n-k)^2+c) \mod k$
From Bad News to Good News

• Bad news:
  – Quadratic probing can cycle through the same full indices, never terminating despite table not being full

• Good news:
  – If TableSize is prime and $\lambda < \frac{1}{2}$, then quadratic probing will find an empty slot in at most TableSize/2 probes
  – So: If you keep $\lambda < \frac{1}{2}$ and TableSize is prime, no need to detect cycles

  – Optional: Proof is posted in lecture14.txt
    • Also, slightly less detailed proof in textbook
    • Key fact: For prime $T$ and $0 < i, j < T/2$ where $i \neq j$, 
      $$(k + i^2) \% T \neq (k + j^2) \% T$$ (i.e., no index repeat)
Clustering reconsidered

- Quadratic probing does not suffer from primary clustering: no problem with keys initially hashing to the same neighborhood.

- But it’s no help if keys initially hash to the same index.
  - Called secondary clustering.

- Can avoid secondary clustering with a probe function that depends on the key: double hashing…
**Double hashing**

Idea:
- Given two good hash functions $h$ and $g$, it is very unlikely that for some key, $h(\text{key}) == g(\text{key})$
- So make the probe function $f(i) = i \times g(\text{key})$

Probe sequence:
- 0th probe: $h(\text{key}) \mod \text{TableSize}$
- 1st probe: $(h(\text{key}) + g(\text{key})) \mod \text{TableSize}$
- 2nd probe: $(h(\text{key}) + 2 \times g(\text{key})) \mod \text{TableSize}$
- 3rd probe: $(h(\text{key}) + 3 \times g(\text{key})) \mod \text{TableSize}$
- ...
- $i$th probe: $(h(\text{key}) + i \times g(\text{key})) \mod \text{TableSize}$

Detail: Make sure $g(\text{key})$ cannot be 0
Double-hashing analysis

• Intuition: Because each probe is “jumping” by \( g(key) \) each time, we “leave the neighborhood” and “go different places from other initial collisions”

• But we could still have a problem like in quadratic probing where we are not “safe” (infinite loop despite room in table)
  – It is known that this cannot happen in at least one case:
    • \( h(key) = key \mod p \)
    • \( g(key) = q - (key \mod q) \)
    • \( 2 < q < p \)
    • \( p \) and \( q \) are prime
More double-hashing facts

• Assume “uniform hashing”
  – Means probability of $g(key_1) \% p == g(key_2) \% p$ is $1/p$

• Non-trivial facts we won’t prove:
  Average # of probes given $\lambda$ (in the limit as $TableSize \to \infty$)
  – Unsuccessful search (intuitive):
    $$\frac{1}{1-\lambda}$$
  – Successful search (less intuitive):
    $$\frac{1}{\lambda \log_e \left( \frac{1}{1-\lambda} \right)}$$

• Bottom line: unsuccessful bad (but not as bad as linear probing), but successful is not nearly as bad
Charts

Uniform Hashing

Average # of Probes

Load Factor

Linear Probing

Average # of Probes

Load Factor

Spring 2015

CSE373: Data Structures & Algorithms

47
Rehashing

• As with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything

• With chaining, we get to decide what “too full” means
  – Keep load factor reasonable (e.g., < 1)?
  – Consider average or max size of non-empty chains?

• For probing, half-full is a good rule of thumb

• New table size
  – Twice-as-big is a good idea, except that won’t be prime!
  – So go about twice-as-big
  – Can have a list of prime numbers in your code since you won’t grow more than 20-30 times
Hashtable Scenarios

For each of the scenarios, answer the following questions:
- Is a hashtable the best-suited data structure?
- If so, what would be used at the keys? Values?
- If not, what data structure would be best-suited?
- What other assumptions, if any, about the scenario must you make to support your previous answers?

- Catalog of items (product id, name, price)
- Bookmarks in a web browser (favicon, URL, bookmark name)
- IT support requests (timestamp, ticket id, description)
- Character frequency analysis (character, # of appearances)
- Activation records for nested function calls (return addresses, local variables, etc.)