lecture23: Hash Tables

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

18th July, 2013
Announcements

- mp6.1 extra credit due tomorrow tonight (7/19)
- lab_avl due tonight
- mp6 due Monday (7/22)
A hash table is a Dictionary ADT implementation that makes use of hash functions to map (key, value) pairs to slots in an array. A hash table consists of:

1. An array ✓
2. A hash function ✓
3. A collision resolution strategy (?)
Separate chaining

Strategy: keep a linked list (or other structure) at each index in the array. If two or more keys hash to the same index, add them into the linked list. How do find, insert, and remove work?

\[
\begin{array}{c}
0 & \rightarrow (k_5, v_2) & \rightarrow (k_1, v_5) \\
1 & \cdot & \\
2 & \rightarrow (k_9, v_2) \\
3 & \cdot & \\
4 & \cdot & \rightarrow (k_0, v_1) \rightarrow (k_2, v_0) \rightarrow (k_8, v_8) \\
5 & \cdot & \\
6 & \cdot & 
\end{array}
\]
Define the **load factor** to be the number of elements stored in the hash table divided by the number of buckets: \( \alpha = \frac{n}{N} \). We want to keep \( \alpha \) approximately constant. How can we do this?

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \rightarrow ) (( k_5, v_2 ))</td>
<td>( \rightarrow ) (( k_1, v_5 ))</td>
<td>( \rightarrow ) (( k_9, v_2 ))</td>
<td>( \rightarrow ) (( k_0, v_1 ))</td>
<td>( \rightarrow ) (( k_2, v_0 ))</td>
<td>( \rightarrow ) (( k_8, v_8 ))</td>
</tr>
</tbody>
</table>
Resizing

- When $\alpha$ becomes too large (usually chosen to be between .66 and .75), we need to increase $N$
- This means doubling the table size and rehashing each element
- Why is it necessary to rehash each element? Can’t we just copy them over to corresponding cells in the larger table?
SUHA and hash table running times

We’ve mentioned previously that all dictionary ADT operations are constant (or amortized constant) time. Let’s see how SUHA and table resizing can ensure this is true.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst Case</th>
<th>SUHA</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>$O(1)^*$</td>
<td>$O(1)^*$</td>
</tr>
<tr>
<td>successful remove/find</td>
<td>$O(n)$</td>
<td>$O\left(\frac{1}{2}(\frac{n}{N})\right) = O\left(\frac{\alpha}{2}\right) = O(1)$</td>
</tr>
<tr>
<td>unsuccessful remove/find</td>
<td>$O(n)$</td>
<td>$O\left(\frac{n}{N}\right) = O(\alpha) = O(1)$</td>
</tr>
</tbody>
</table>

Recall we are using linked lists as our “buckets”. Could we use some other data structure? How (if at all) would that change our running times?
Separate chaining: insert

Linear probing in some senses is simpler than separate chaining.

**insert**: if a key is already in the desired spot, keep moving along the table until you find and empty spot and put it there!

From the previous example, we know $k_5$ and $k_1$ have the same hash as well as $k_0$, $k_2$, and $k_8$.

Let’s insert in this order: $k_0$, $k_1$, $k_2$, $k_5$

Make note $h(k_0) = 5$, $h(k_1) = 1$, $h(k_2) = 5$, $h(k_5) = 1$
Linear probing: insert

Linear probing in some senses is simpler than separate chaining.

**insert**: if a key is already in the desired spot, keep moving along the table until you find an empty spot and put it there!

From the previous example, we know $k_5$ and $k_1$ have the same hash as well as $k_0$, $k_2$, and $k_8$.

Let’s insert in this order: $k_0, k_1, k_2, k_5$

Make note $h(k_0) = 5$, $h(k_1) = 1$, $h(k_2) = 5$, $h(k_5) = 1$
Linear probing: find

For find, we need to hash the key as usual, and look up that index in the hash table.

Let’s call `find` on these objects: $k_0, k_5, k_6$. Note $h(k_6) = 1$.

If the key we’re looking for is not in the cell we hash to, we keep looking (probing) until we either find it or get to a completely empty cell.
Linear probing: remove

When removing elements, we need to mark the cell it was in to indicate something was there at one point. Why is this necessary in order for \texttt{find} to work properly?

Let’s remove $k_1$, marking its cell that something existed there.

Now, when we call \texttt{find} on $k_5$, we know to keep looking past cell 1, since there used to be something there.
Clustering

- In linear probing, clusters of keys are likely to appear as keys hash into a similar range and have to traverse to adjacent indices.
- Hashing into a cluster becomes more and more likely as more data is inserted.
- This increases the cost of insert!
Solutions to clustering

- **Quadratic hashing**
  - If the hashed cell index is occupied, advance forward a number of steps determined by a quadratic polynomial.
  - For example, if the initial hash is $i$, advance in steps $(i, i + 1^2, i + 2^2, i + 3^2, i + 4^2, ...)$

- **Double hashing**
  - Uses two hash functions, $h_1, h_2$
  - If the cell $h_1(k)$ is occupied, advance in a fixed step size determined by $h_2(k)$ until there is an open cell.

- **Cuckoo hashing**
  - Uses two hash functions, $h_1, h_2$
  - Use $h_1$ to insert. If there is a collision, “kick out” the offender and rehash it using $h_2$
  - Repeat this process until an open cell is found or an infinite loop is encountered.
LP performance

insert, find, and remove expected number of probes for linear probing:

successful:
\[
\frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right)
\]

unsuccessful:
\[
\frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right)^2
\]

Don’t memorize these! Just know that they are all constant assuming \( \alpha \) is held constant.
Open vs closed hashing

- Separate chaining is an example of *open hashing*: the actual objects live “outside” the hash table
  - Variants use different data structures as the buckets
- Linear probing is an example of *closed hashing*: the objects live inside the hash table (array) itself
  - Variants use *non*-linear probing strategies; this is in hopes to disperse the keys more throughout the table and avoid clustering
Philosophical questions

Why do we even have balanced BSTs if hash tables are so awesome?

- insert, find, delete running times
- sort running times
- Memory usage
- operator< vs a hash function
- Complexity (not in the running time sense)
Denial of Service via Algorithmic Complexity Attacks

- Paper by Crosby and Wallach in 2003, currently cited by 215
- They explicitly made the worst case happen every time in order to slow down (and break) operations
  - Intrusion detection system: send specific packets to server, using very low bandwidth
  - Perl: insert specific strings into an associative array
  - Linux kernel (2.4.20): save files with specific names