Admin

Assignment 5 and 6 back soon
Assignment 7 our and due on Sunday
Normal group session schedule this week

Hashtables

Constant time insertion and search (and deletion in some cases) for a large space of keys

Applications
- Does x belong to S?
- I’ve found them very useful (go by many names, maps, dictionaries, ...)
- compilers
- databases
- search engines
- storing and retrieving non-sequential data
- save memory over an array

For this class, we’ll just think of them as a collection of keys

For many applications/implementations, there is a value associated with the key, i.e., key/value pair (though lookup is still exclusively based on the key)
Why not just arrays aka direct-address tables?

Array must be as large as the universe of keys.

Why not just arrays?

Array must be as large as the universe of keys.
Space of actual keys is often much smaller than the actual keys.

Why not arrays?

Think of indexing all last names < 10 characters
- Census listing of all last names
  - 88,799 last names
- What is the size of our space of keys?
  - \(26^{10}\) is a big number
- Not feasible!
- Even if it were, not space efficient

The load of a table/hashtable

\(m\) \(=\) number of possible entries in the table
\(n\) \(=\) number of keys stored in the table
\(\alpha = n/m\) is the load factor of the hashtable

What is the load factor of the last example?
- \(\alpha = 88,799 / 26^{10}\) would be the load factor of last names using direct-addressing

The smaller \(\alpha\), the more wasteful the table.

The load also helps us talk about run time.
A hash function is a function that maps the universe of keys to the slots in the hashtable.

$$h: U \rightarrow m$$

What can happen if $m < |U|$? 

$|U| = m << |U|$
Collisions

If $m < |U|$, then two keys can map to the same position in the hashtable (pigeonhole principle)

A collision occurs when $h(x) = h(y)$, but $x \neq y$

A good hash function will minimize the number of collisions

Because the number of hashtable entries is less than the possible keys (i.e. $m < |U|$) collisions are inevitable!

Collision resolution techniques?

Collision resolution by chaining

Hashtable consists of an array of linked lists

When a collision occurs, the element is added to linked list at that location

If two entries $x \neq y$ have the same hash value $h(x) = h(y)$, then $T(h(x))$ will contain a linked list with both values

Insertion

ChainedInsert($x$):
- entry $= h(x)$
- Insert $x$ at the head of $T[entry]$

ChainedHashInsert($\square$)
**Insertion**

ChainedInsert(x):

- entry = h(x)
- insert x at the head of T[entry]

Insertion hash function is a mapping from the key to some value < m

**Deletion**

ChainedDelete(x):

- entry = h(x)
- delete x at the list at T[entry]

Deletion Search through the list!
Deletion

ChainedDelete(x):
entry = h(x)
delete x at the list at T[entry]
Search

ChainedHashSearch(x):
entry = h(x)
search for x in list T[entry]

32

h( )

33

Search

ChainedHashSearch(x):
entry = h(x)
search for x in list T[entry]

34

ChainedHashSearch( )

35
ChainedHashSearch(x):
  entry = h(x)
  search for x in list T[entry]

ChainedInsert(x):
  entry = h(x)
  insert x at the head of T[entry]

ChainedDelete(x):
  entry = h(x)
  delete x at the list at T[entry]

ChainedSearch(x):
  entry = h(x)
  search for x in list T[entry]

Running time

\[ \Theta(1) \]

O(length of the chain)

Length of the chain

Worst case?

- All elements hash to the same location
- h(k) = 4
- n
Length of the chain

Average case:
- Depends on how well the hash function distributes the keys

What is the best we could hope for a hash function?
- Simple uniform hashing: an element is equally likely to end up in any of the $m$ slots.

Under simple uniform hashing what is the average length of a chain in the table?
- $n$ keys over $m$ slots = $\frac{n}{m} = \alpha$

Average chain length

If you roll a fair $m$-sided die $n$ times, how many times are we likely to see a given value?

For example, 10 sided die:
- $1$ time
  - $\frac{1}{10}$
- $100$ times
  - $100/10 = 10$

Search average running time

Two cases:
- Key is not in the table
  - $O(1 + \alpha)$
- Key is in the table
  - On average search half of the entries
  - $O(1 + \alpha)$

Hash functions

What makes a good hash function?
- Approximates the assumption of simple uniform hashing
- Deterministic, $h(x)$ should always return the same value
- Low cost: if it is expensive to calculate the hash value (e.g., $\log n$) then we don’t gain anything by using a table

Challenge: we don’t generally know the distribution of the keys:
- Frequently data tend to be clustered (e.g., similar strings, runtimes, SSNs)
- A good hash function should spread these out across the table
Hash functions

What are some hash functions you've heard of before?

Division method

$h(k) = k \mod m$

<table>
<thead>
<tr>
<th>m</th>
<th>k</th>
<th>h(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>17</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>133</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>25</td>
<td>12</td>
</tr>
</tbody>
</table>

Don't use a power of two. Why?

<table>
<thead>
<tr>
<th>m</th>
<th>k</th>
<th>bin(k)</th>
<th>h(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>25</td>
<td>11001</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>00001</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>10001</td>
<td></td>
</tr>
</tbody>
</table>
Division method

Don't use a power of two. Why?

<table>
<thead>
<tr>
<th>m</th>
<th>k</th>
<th>bin(k)</th>
<th>h(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>25</td>
<td>11001</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>00001</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>10001</td>
<td>1</td>
</tr>
</tbody>
</table>

If \( h(k) = k \mod 2^p \), the hash function is just the lower \( p \) bits of the value.

Division method

Good rule of thumb for \( m \) is a prime number not too close to a power of 2

Pros:
- quick to calculate
- easy to understand

Cons:
- keys close to each other will end up close in the hashtable

Multiplication method

Multiply the key by a constant \( 0 < A < 1 \) and extract the fractional part of \( kA \), then scale by \( m \) to get the index

\[
h(k) = \left\lfloor m(kA - \lfloor kA \rfloor) \right\rfloor
\]

extracts the fractional portion of \( kA \)

Common choice is for \( m \) as a power of 2 and \( A = (\sqrt{3} - 1)/2 \approx 0.6180339887 \)

Book has other heuristics
### Multiplication method

<table>
<thead>
<tr>
<th>m</th>
<th>k</th>
<th>A</th>
<th>kA</th>
<th>h(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>15</td>
<td>0.618</td>
<td></td>
<td>9.27</td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td>0.618</td>
<td></td>
<td>14.214</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>0.618</td>
<td></td>
<td>61.8</td>
</tr>
</tbody>
</table>

\[ h(k) = \left\lfloor m(kA - \left\lfloor kA \right\rfloor) \right\rfloor \]

---

### Other hash functions

- Cyclic redundancy checks (i.e. disks, cds, dvds)
- Checksums (i.e. networking, file transfers)
- Cryptographic (i.e. MD5, SHA)

---

### Open addressing

Keeping around an array of linked lists can be inefficient and a hassle.

Like to keep the hashtable as just an array of elements (no pointers)

How do we deal with collisions?
- Compute another slot in the hashtable to examine
Hash functions with open addressing

Hash function must define a probe sequence which is the list of slots to examine when searching or inserting.

The hash function takes an additional parameter $i$ which is the number of collisions that have already occurred.

The probe sequence must be a permutation of every hashtable entry. Why?

$\{ h(k,0), h(k,1), h(k,2), ..., h(k, m-1) \}$ is a permutation of $\{ 0, 1, 2, 3, ..., m-1 \}$.

If not, we wouldn't explore all the possible location in the table!
Probe sequence

\[ h(k, 2) \]

must visit all locations

Open addressing: Insert

```plaintext
Hash-Insert(\(T, k\))
1 \( i = 0 \)
2 \( j = h(k, i) \)
3 while \( i < m - 1 \) and \( T[j] \neq \text{null} \)
4 \( i = i + 1 \)
5 if \( T[j] = \text{null} \)
6 return \( j \)
7 else error "hash is full"
```
Open addressing: Insert

```
Hast-Insert(T, k)
1 i = 0
2 j = h(k, i)
3 while i < m - 1 and T[j] ≠ null
4     i = i + 1
5     j = h(k, i)
6 if T[j] = null
7     return j
8 else
9     error “hash is full”
```

get the first hashtable entry to look in

follow the probe sequence until we find an open entry

return the open entry

hashtable can fill up
Open addressing: search

Hash-Search(T, k)
1. \( i = 0 \)
2. \( j = h(k, i) \)
3. while \( i < m \) and \( T[j] \neq \text{null} \) and \( T[j] \neq k \)
4. \( i = i + 1 \)
5. \( j = h(k, i) \)
6. if \( T[j] = k \)
7. return \( j \)
8. else
9. return \text{null}

"breaks" the probe sequence

Delete

Hash-Insert(T, k)
1. \( i = 0 \)
2. \( j = h(k, i) \)
3. while \( i < m - 1 \) and \( T[j] \neq \text{null} \) and \( T[j] \neq k \)
4. \( i = i + 1 \)
5. \( j = h(k, i) \)
6. if \( T[j] = k \)
7. return \( j \)
8. else
9. return \text{null}
10. else error "hash is full"
Can we just delete this node?

No! Then if we search for k we'll mistakenly think it's not there!

Open addressing: delete

Two options:
- mark node as "deleted" (rather than null)
- modify search procedure to continue looking if a "deleted" node is seen
- modify insert procedure to fill in "deleted" entries
- increases search times
- if a lot of deleting will happen, use chaining
Probing schemes

Linear probing = if a collision occurs, go to the next slot
- $h(k, i) = (h(k) + i) \mod m$
- Does it meet our requirement that it visits every slot
- For example, $m = 7$ and $h(k) = 4$

- $h(k, 0) = 4$
- $h(k, 1) = 5$
- $h(k, 2) = 6$
- $h(k, 3) = 0$
- $h(k, 3) = 1$

Linear probing: search

$h(\square, 0)$

$h(\square, 1)$

$h(\square, 2)$
Linear probing: search

\[ h(\_, 3) \]

81

Linear probing: search

\[ h(\_, 3) \]

82

**Linear probing**

*Problem:*

- primary clustering - long runs of occupied slots tend to build up and these tend to grow

- any value here results in an increase in the cluster

- becomes more and more probable for a value to end up in that range

83

**Quadratic probing**

**h(k, i) = (h(k) + c_1i + c_2i^2) \mod m**

Rather than a linear sequence, we probe based on a quadratic function

*Problems:*

- must pick constants and \( m \) so that we have a proper probe sequence
- if \( h(x) = h(y) \), then \( h(x, i) = h(y, i) \) for all \( i \)
- secondary clustering

84
Double hashing

Probe sequence is determined by a second hash function
\[ h(k,i) = (h_1(k) + i h_2(k)) \mod m \]

Problem:
- \( h_2(k) \) must visit all possible positions in the table

Running time of insert and search for open addressing

Depends on the hash function/probe sequence

Worst case?
- \( \Theta(n) \) = probe sequence visits every full entry first before finding an empty

Running time of insert and search for open addressing

Average case?
- We have to make at least one probe

What is the probability that the first probe will not be successful (assume uniform hashing function)?

\[ \alpha \]
Running time of insert and search for open addressing

Average case?

What is the probability that the first two probed slots will not be successful?

Why \( \sim \alpha^2 \) ?

Technically, second probe is \( \frac{n-1}{m-1} \) \( \sim \alpha^2 \)

Average case?

What is the probability that the first three probed slots will not be successful?

\( \sim \alpha^3 \)

Average case: expected number of probes

\[
E[\text{probes}] = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots
\]

\[
= \sum_{i=0}^{\infty} \alpha^i
\]

\[
< \sum_{i=0}^{\infty} \alpha^i
\]

\[
= \frac{1}{1-\alpha}
\]
Average number of probes

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>Average number of searches</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>( \frac{1}{1 - 0.1} = 1.11 )</td>
</tr>
<tr>
<td>0.25</td>
<td>( \frac{1}{1 - 0.25} = 1.33 )</td>
</tr>
<tr>
<td>0.5</td>
<td>( \frac{1}{1 - 0.5} = 2 )</td>
</tr>
<tr>
<td>0.75</td>
<td>( \frac{1}{1 - 0.75} = 4 )</td>
</tr>
<tr>
<td>0.9</td>
<td>( \frac{1}{1 - 0.9} = 10 )</td>
</tr>
<tr>
<td>0.95</td>
<td>( \frac{1}{1 - 0.95} = 20 )</td>
</tr>
<tr>
<td>0.99</td>
<td>( \frac{1}{1 - 0.99} = 100 )</td>
</tr>
</tbody>
</table>

How big should a hashtable be?

A good rule of thumb is the hashtable should be around half full.

What happens when the hashtable gets full?

Copy: Create a new table and copy the values over.
- Simple to implement.

Amortized copy: When a certain ratio is hit, grow the table, but copy the entries over a few at a time with every insert.
- No single insert is expensive and use guarantees per insert performance.
- More complicated to implement.

Questions

Why can't we just use an array?

What is a hash function? How does it differ from the hash_code method in Java?

What are the two ways we deal with collisions?

Why is it important that the probe sequence visits every spot in the hashtable?
Questions

What are three potential probing mechanisms?

If we insert random data into a hashtable, what is the worst case running time for searching for an item?

If an open-addressed hashtable is half full, on average, how many entries would we expect to search before finding an open one? 75% full?

If we plan to do a lot of deleting, what type of hashtable should we use?

Questions

What is the largest $\alpha$ can be for a hashtable with chaining? Open-addressed?

Fill in the table for division method

$h(k) = k \mod m$

<table>
<thead>
<tr>
<th>m</th>
<th>k</th>
<th>h(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>

Fill in the table for multiplication method

$h(k) = \left\lfloor m(kA - \left\lfloor kA \right\rfloor) \right\rfloor$

<table>
<thead>
<tr>
<th>m</th>
<th>k</th>
<th>A</th>
<th>kA</th>
<th>h(k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>15</td>
<td>0.618</td>
<td>h(k)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td>0.618</td>
<td>h(k)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>0.618</td>
<td>h(k)</td>
<td></td>
</tr>
</tbody>
</table>

$h(k) = \left\lfloor m(kA - \left\lfloor kA \right\rfloor) \right\rfloor$