Admin

- Lab tomorrow
- Midterm recap (save questions for then)
- Course feedback discussion
- Start next assignment (2 week assignment)

Quiz on Thursday

Sets

- An unordered collection
- Things can be added and removes
- Check if things are in the set

```java
public interface Set<E> {
    public void put(E key);
    public boolean containsKey(E key);
    public E remove(E key);
    public boolean isEmpty();
    public int size();
}
```

Why not just arrays?

- Universe of keys - $U$
- 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 used is often much, much smaller than the universe of keys.

Hashables

- Using an array is still a good idea.
- Key idea: need to translate from the key into an index in the array.

Hash function, h

- A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array).

    - universe of keys - U
    - hash function, h: U → m
    - m << |U|
A hash function is a function that maps the universe of keys to a restricted range (e.g., the size of an array). 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 (array) entries is less than the possible keys (i.e., \( m < |U| \)) collisions are inevitable! We need to handle collisions!

Collision resolution techniques:

1. **Collision resolution by chaining**
   - Hashtable consists of an array of linked lists.
   ```java
   private LinkedList<E>[] table;
   ```
   - When a collision occurs, the element is added to the linked list at that location.
   - If two entries \( x \neq y \) have the same hash value, \( h(x) = h(y) \), then \( table[h(x)] \) will contain a linked list with both values.
   ```java
   put
   containsKey
   remove
   ```
Collision resolution by chaining

put: addFirst h(key)
containsKey: contains h(key)
remove: remove h(key)

Running time?

put: O(1)
containsKey: O(length of linked list)
remove: O(length of linked list)

Length of the chain

Worst case?
**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 = \( n / m = \alpha \)

---

**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

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

The load also helps us talk about run time

---

**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
  - \( 1 / 10 \)
- 100 times
  - \( 100 / 10 = 10 \)
containsKey average running time

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

Hash functions

Function takes as input a key and return a value from 0 to m-1 (the size of the hashtable)

What makes a good hash function?
- Approximates the assumption of simple uniform hashing
- Deterministic – \(h(s)\) 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, run-times, SSNs). A good hash function should spread these out across the table

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>
Division method

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

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

Don’t use a power of two. Why?

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>00001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>10001</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Don’t use a power of two. Why?

Don’t use a power of two. Why?

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

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

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

exacts the fractional portion of $kA$

Common choice is for $m$ as a power of 2 and

$$A = (\sqrt{5} - 1)/2 = 0.6180339887$$

Why a power of 2?

Book has other heuristics

<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></td>
</tr>
<tr>
<td>8</td>
<td>23</td>
<td>0.618</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>0.618</td>
<td></td>
<td></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

Open addressing

Hash function must define a **probe sequence** which is the list of slots to examine when doing a put or containsKey

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. \textbf{Why?}

\[ \{ h(k,0), h(k,1), h(k,2), \ldots, h(k, m-1) \} \text{ is a permutation of } \{ 0, 1, 2, 3, \ldots, m-1 \} \]

Hash functions with open addressing

Hash function must define a **probe sequence** which is the list of slots to examine when doing a put or containsKey

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. \textbf{Why?}

\textbf{If not, we wouldn’t explore all the possible location in the table!}
Probe sequence

h(k, 0)

h(k, 1)

h(k, 2)

h(k, 3)
Probe sequence

\[ h(k, \ldots) \]

must visit all locations

Open addressing: put

```java
public void put(E key) {
    int i = 0;
    int next = probeSequence(key, i);
    while( i < table.length && table[next] != null ){
        i++;
        next = probeSequence(key, i);
    }
    table[next] = key;
    count++;
}
```

What does this code do?
Open addressing: put

```java
public void put(E key) {
    int i = 0;
    int next = probeSequence(key, i);
    while (i < table.length && table[next] != null) {
        i++;
        next = probeSequence(key, i);
    }
    table[next] = key;
    count++;
}
```

Open addressing: containsKey

```java
public boolean containsKey(E key) {
    int i = 0;
    int next = probeSequence(key, i);
    while (i < table.length && table[next] != null) {
        i++;
        next = probeSequence(key, i);
    }
    if (i == table.length || table[next] == null) {
        return false;
    }
    return equals(key, table[next]);
}
```

// only 3 ways to exit the while loop
// the two of which below mean we didn't find it
// return false; table[next] == null;

Open addressing: containsKey

```java
public boolean containsKey(E key){
    int i = 0;
    int next = probeSequence(key, 1);
    while(i < table.length && table[next] == null &&
            next != probeSequence(key, i)){
        ++i;
        next = probeSequence(key, i);
    }
    // only 3 ways to exit the while loop
    // the two of which below mean we didn't find it
    return i == table.length || table[next] == null;
} // very similar to put!
```

also need to check if we've found the key

Open addressing: remove

Two options:
- mark node as “deleted” (rather than null)
  - modify containsKey to continue looking if a “deleted” node is seen
  - modify put 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 \( h(k) = 4 \)

\[
\begin{align*}
h(k,0) &= 4 \\
h(k,1) &= 5 \\
h(k,2) &= 6 \\
h(k,3) &= 0 \\
h(k,4) &= 1 \\
h(k,5) &= 2 \\
h(k,6) &= 3 
\end{align*}
\]
Linear probing: put

h(□, 0)

53

Linear probing: put

h(□, 1)

54

Linear probing: put

h(□, 2)

55

Linear probing: put

h(□, 3)

56
Linear probing: put

\[ h(k, 3) \]

Problem:
- Primary clustering — long runs of occupied slots tend to build up and these tend to grow

Quadratic probing

\[ h(k, i) = (h(k) + c_1 i + c_2 i^2) \mod m \]

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

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 put and containsKey for open addressing

- Depends on the hash function/probe sequence

**Worst case?**
- \( O(n) \) – probe sequence visits every full entry first before finding an empty

**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)?

\[ \alpha \]

Why \( \sim \alpha^2 \)?
Running time of put and containsKey for open addressing

Average case?

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

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

Running time of put and containsKey for open addressing

Average case?

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

~$\alpha^3$

Running time of insert and search for open addressing

Average case: expected number of probes

sum of the probability of making 1 probe, 2 probes, 3 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

\[
E[\text{probes}] = \frac{1}{1-\alpha}
\]

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>Average number of searches</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1/(1 - 0.1) = 1.11</td>
</tr>
<tr>
<td>0.25</td>
<td>1/(1 - 0.25) = 1.33</td>
</tr>
<tr>
<td>0.5</td>
<td>1/(1 - 0.5) = 2</td>
</tr>
<tr>
<td>0.75</td>
<td>1/(1 - 0.75) = 4</td>
</tr>
<tr>
<td>0.9</td>
<td>1/(1 - 0.9) = 10</td>
</tr>
<tr>
<td>0.95</td>
<td>1/(1 - 0.95) = 20</td>
</tr>
<tr>
<td>0.99</td>
<td>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
- results in one expensive put
- 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 put is expensive and can guarantee per put performance
- more complicated to implement

To the code...

abstract classes!

Making your classes hashable:
- hashCode
- equals

HashSet:
https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

HashMap:
https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html