CSCI 241

Lecture 16
A3 Overview, Map ADT, Rehashing, Open Addressing
Goals

• Understand the architecture of A3

• Understand the purpose and operations of the Map ADT.

• Know how to respond to large hash table load factors by resizing the array and rehashing.

• Know how to avoid using LinkedList buckets using open addressing with linear or quadratic probing.

• Understand the relationship between Java Object’s hashCode and equals methods.
Announcements

• A3 is out
A3 has 4 phases.
A3 has 4 phases.

It may sound scary.
A3 has 4 phases.

It may sound scary.

DON'T PANIC
A3 has 4 phases.

It may sound scary.

It isn’t so bad:

• total lines of code is probably $\leq A2$
• nothing here is as tricky as AVL rebalance
• you’re given unit tests
A3 has 4 phases.

0. Write an ArrayList clone
A3 has 4 phases.

0. Write an ArrayList clone
   (done in Lab 6!)
A3 has 3 phases.
A3 has 3 phases.

1. Write a min-heap to implement a priority queue with operations:
   - boolean add(V value, P priority)
   - V peek();
   - V poll();
A3 has 3 phases.

1. Write a min-heap to implement a priority queue with operations:
   - boolean add(V value, P priority)
   - V peek();
   - V poll();

use AList to handle growing the array!
A3 has 3 phases.

1. Write a min-heap to implement a priority queue with operations:
   - boolean add(V value, P priority)
   - V peek();
   - V poll();

2. Write a hash table.

use AList to handle growing the array!
A3 has 3 phases.

1. Write a min-heap to implement a priority queue with operations:
   - boolean add(V value, P priority)
   - V peek();
   - V poll();

2. Write a hash table.

3. Use the hash table to augment the heap, making the following operations efficient:
   - boolean contains(V v);
   - void changePriority(V v, P newP);
A3 has 3 phases.

1. Write a min-heap to implement a priority queue with operations:
   - boolean add(V value, P priority)
   - V peek();
   - V poll();

2. Write a hash table.

3. Use the hash table to augment the heap, making the following operations efficient:
   - boolean contains(V v);
   - void changePriority(V v, P newP);

use AList to handle growing the array!
Phase 3 - Hash your Heap

In Phase 1 Heap:

- **contains** requires searching the whole tree.
- **changePriority** requires searching the whole tree, then bubbling down or up.
Phase 3 - Hash your Heap

In Phase 3 Heap:

- Each heap value is stored in the heap and in a HashTable that tracks its index in the heap.

HashTable<V, Integer>:

<table>
<thead>
<tr>
<th>value</th>
<th>i (index in heap)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>35</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>19</td>
<td>12</td>
</tr>
</tbody>
</table>

Heap: [4 6 5 21 8 14 35 22 38 55 10 20 19]
Phase 3 - Hash your Heap

In Phase 3 Heap:

- Each heap value is stored in the heap and in a HashTable that tracks its index in the heap.

HashTable\(<V, \text{Integer}>\):

<table>
<thead>
<tr>
<th>value</th>
<th>i (index in heap)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>35</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

To maximize confusion:

- The hash table is used to map Heap values to heap indices.
- The hash table’s keys are the heap’s values.

Heap: [4 6 5 21 8 14 35 22 38 55 10 20 19]
Phase 3 - Hash your Heap

In Phase 3 Heap:

boolean contains(V v):
  true iff map contains key v

HashTable<V, Integer>:
  value  i (index in heap)

Heap: [ 4 6 5 21 8 14 35 22 38 55 10 20 19 ]
Phase 3 - Hash your Heap

In Phase 3 Heap:

```java
void changePriority(V v, P newP):
    i = map.get(v);
    change priority of heap entry
    bubble it up or down
```

HashTable<V, Integer>:

<table>
<thead>
<tr>
<th>value</th>
<th>i (index in heap)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>38</td>
<td>8</td>
</tr>
</tbody>
</table>

Heap: [4 6 5 21 8 14 35 22 38 55 10 20 19]
Questions?
Origins of the term “hash”

Hans Peter Luhn (July 1, 1896 – August 19, 1964) was a researcher in the field of computer science, and, Library & Information Science for IBM.
The Map ADT

• In math, a map is a function.

• What is a function, anyway?
The Map ADT

- In math, a **map** is a function.

- If F is a map then F(a) → b means that a maps to b.

- F has a:
  - **domain** - the set of values F maps **from**
  - **range** - the set of values that F maps a domain element **to**
  - **codomain** - the set of **all** possible values in the range’s type, regardless of whether any element in the domain maps to it
The Map ADT

```java
Thing[] a = new Thing[10];
```
The Map ADT

- Arrays are great!
- Domain: 0..a.length
- Range: all elements stored in the array
- Codomain: the type of elements stored in the array.

```java
Thing[] a = new Thing[10];
```
The Map ADT

- Arrays are great!
- Domain: 0..a.length
- Range: all elements stored in the array
- Codomain: the type of elements stored in the array.

We get to choose the codomain.

```java
Thing[] a = new Thing[10];
```

Domain: 0

Range: Thing objects.

Codomain: Thing objects.
The Map ADT

- Arrays are great!

- We get to choose the codomain - type of the array.

- Wouldn’t it be nice to choose the domain as well?

- The Map ADT represents a mapping from keys to values.

- we get to choose the type of the keys (domain) AND the values (codomain)
public interface Map<K,V> {
    /** Returns the value to which the specified key
     * is mapped, or null if this map contains no
     * mapping for the key. */
    V get(Object key);

    /** Associates the specified value with the
     * specified key in this map */
    V put(K key, V value);

    /** Removes the mapping for a key from this map
     * if it is present */
    V remove(Object key);

    // more methods
}
Implementing Map<K,V>

- Use a HashTable!
- Hash the key to determine array index
- Store values in array

\[ h(k) = k \mod A.length \]

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);
Implementing Map<K,V>

- Use a HashTable!
- Hash the key to determine array index
- Store values in array

\( h(k) = k \mod A.\text{length} \)

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);
Implementing Map<K,V>

- Use a HashTable (or a HashSet of Key-Value pairs)
- Hash the key to determine array index
- Store values in array
- Store (K,V) pairs in the array.

```java
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```
Hash Tables: Load Factor

\[
\frac{\text{# entries in table}}{\text{size of the array}}
\]
Hash Tables: Load Factor

How full is your hash table?

Load factor \( \lambda = \frac{\text{# entries in table}}{\text{size of the array}} \)

The average bucket size is \( \lambda \).

Average-case runtime is \( O(\lambda) \).
Hash Tables: Load Factor

\[
\text{Load Factor} = \frac{\text{# entries in table}}{\text{size of the array}}
\]
Load factor $\lambda = \frac{\# \text{ entries in table}}{\text{size of the array}}$

Average-case runtime is $O(\lambda)$.

- If $\lambda$ is large, runtime is slow.
- If $\lambda$ is small, memory is wasted.

Strategy: grow or shrink array when $\lambda$ gets too large or small.
Shrinking the array

Requires **rehashing**: put each element where it belongs in the new array.

```
<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></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>11</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>14</td>
<td></td>
<td></td>
<td></td>
<td>24</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></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Requires rehashing: put each element where it belongs in the new array.
Shrinking the array

Requires **rehashing**: put each element where it belongs in the new array.

<table>
<thead>
<tr>
<th>0</th>
<th>10 “bear”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 “dog”</td>
</tr>
<tr>
<td></td>
<td>11 “auk”</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14 “cat”</td>
</tr>
<tr>
<td></td>
<td>24 “ape”</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

\[(10 \% 3) \rightarrow 1\]
Shrinking the array

Requires **rehashing**: put each element where it belongs in the new array.

\[
\begin{array}{|c|c|}
\hline
0 & 10 \text{ "bear"} \\
1 & 1 \text{ "dog"} \\
2 & \\
3 & \\
4 & 14 \text{ "cat"} \\
5 & \\
6 & \\
7 & \\
8 & \\
9 & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
0 & \\
1 & 1 \text{ "dog"} \\
2 & \\
\hline
\end{array}
\]

\[
(10 \mod 3) \rightarrow 1 \\
(1 \mod 3) \rightarrow 1
\]
Shrinking the array

Requires **rehashing**: put each element where it belongs in the new array.

<table>
<thead>
<tr>
<th>0</th>
<th>10 “bear”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>“dog”</td>
</tr>
<tr>
<td></td>
<td>11 “auk”</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>“cat”</td>
</tr>
<tr>
<td></td>
<td>24 “ape”</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

(10 % 3) -> 1
(1 % 3) -> 1
(11 % 3) -> 2

Requires rehashing: put each element where it belongs in the new array.
Shrinking the array

Requires **rehashing**: put each element where it belongs in the new array.

<table>
<thead>
<tr>
<th>0</th>
<th>10 “bear”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 “dog”</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14 “cat”</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

(10 % 3) -> 1
(1 % 3) -> 1
(11 % 3) -> 2
(14 % 3) -> 2

Requires rehashing: put each element where it belongs in the new array.
Shrinking the array

Requires **rehashing**: put each element where in belongs in the new array.

<table>
<thead>
<tr>
<th>0</th>
<th>10 “bear”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 “dog”</td>
</tr>
<tr>
<td></td>
<td>11 “auk”</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14 “cat”</td>
</tr>
<tr>
<td></td>
<td>24 “ape”</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

(10 % 3) -> 1
(1 % 3) -> 1
(11 % 3) -> 2
(14 % 3) -> 2
(24 % 3) -> 0
Growing the array

Also requires **rehashing**: put each element where in belongs in the new array.

**Exercise:** Grow the array to size 6 and rehash:

<table>
<thead>
<tr>
<th>0</th>
<th>24   “ape”</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10   “bear”</td>
</tr>
<tr>
<td>1</td>
<td>1      “dog”</td>
</tr>
<tr>
<td>2</td>
<td>11   “auk”</td>
</tr>
<tr>
<td>14 “cat”</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

**ABCD:**

How many elements are in the most full bucket?

A. 1
B. 2
C. 3
D. 4
Rehashing: Runtime

Let \( N \) = array size

Let \( n \) = number of entries

Rehashing algorithm:

for each bucket \( b \):
  for each element \( e \) in \( b \):
    put \( e \) into the new array

visits \( N \) buckets

visits \( n \) entries (total)

could be \( O(n) \) =(
Rehashing: Runtime, take 1

Let C = array size
Let n = number of entries

Rehashing algorithm:

for each bucket b:
    for each element e in b:
        put e into the new array
### Rehashing: Runtime, take 1

#### Rehashing algorithm:

for each bucket $b$:

  for each element $e$ in $b$:

    put $e$ into the new array

Let $C = \text{array size}$

Let $n = \text{number of entries}$

<table>
<thead>
<tr>
<th>Bucket $b$</th>
<th>Element $e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;bear&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;dog&quot;</td>
</tr>
<tr>
<td>11</td>
<td>&quot;auk&quot;</td>
</tr>
<tr>
<td>4</td>
<td>&quot;cat&quot;</td>
</tr>
<tr>
<td>24</td>
<td>&quot;ape&quot;</td>
</tr>
</tbody>
</table>
Rehashing: Runtime, take 1

Rehashing algorithm:

for each bucket $b$:

for each element $e$ in $b$:

put $e$ into the new array

Let $C =$ array size
Let $n =$ number of entries

visits $C$ buckets
visits $n$ entries (total)
Rehashing: Runtime, take 1

Rehashing algorithm:
for each bucket b:
for each element e in b:
put e into the new array

Let \( C \) = array size
Let \( n \) = number of entries

visits \( C \) buckets
visits \( n \) entries (total)
could be \( O(n) \) =(
Rehashing: Runtime, take 1

Rehashing algorithm:
for each bucket b:
for each element e in b:
put e into the new array

Visits C buckets
visits n entries (total)
Overall runtime is:
• worst-case O(C + n^2)
• average-case O(C + n)

Let C = array size
Let n = number of entries

Visits n entries (total)
could be O(n) =(
Rehashing: Runtime, take 1

Let $C = \text{array size}$
Let $n = \text{number of entries}$

Rehashing algorithm:

for each bucket $b$:
  for each element $e$ in $b$:
    put $e$ into the new array

visits $C$ buckets
visits $n$ entries (total)
Rehashing: Runtime, take 1

Let $C = \text{array size}$

Let $n = \text{number of entries}$

Rehashing algorithm:

for each bucket $b$:

for each element $e$ in $b$:

put $e$ into the new array

visits $C$ buckets

visits $n$ entries (total)

could it be $O(n)$?
Rehashing: Runtime, take 1

Let $C = \text{array size}$

Let $n = \text{number of entries}$

Rehashing algorithm:

for each bucket $b$:

for each element $e$ in $b$:

put $e$ into the new array

visits $C$ buckets

visits $n$ entries (total)

could it be $O(n)$?

We can’t have duplicate keys: all $(k,v)$ pairs were already in the map!

Consequence: we don’t need to search the bucket when rehashing
Rehashing: Runtime, take 1

Let $C =$ array size
Let $n =$ number of entries

Overall runtime is:
• worst-case $O(C + n)$

visits $C$ buckets
visits $n$ entries (total)

could it be $O(n)$?

We can’t have duplicate keys: all (k,v) pairs were already in the map!
Consequence: we don’t need to search the bucket when rehashing
Hashing Multiple Integers

- Various heuristic methods:
  - \((a + b + c + d) \mod N\)
  - \((a k^1 + b k^2 + c k^3 + d k^4) \mod N\)

Hashing Strings

- Interpret ASCII (or unicode) representation as an integer.

- Java String uses:
  \(s[0] \times 31^{(n-1)} + s[1] \times 31^{(n-2)} + \ldots + s[n-1]\)
Collision Resolution

- **Chaining** - use a LinkedList to store multiple elements per bucket.

- **Open Addressing** - use empty buckets to store things that belong in other buckets.
  
  - Need some scheme for deciding which buckets to look in.
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

```
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

```
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

---

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>(1, dog)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
put(key):
  h = hash(key);
  while A[h] is full:
    h = (h+1) % N
  A[h] = value
```
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

```
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

<table>
<thead>
<tr>
<th></th>
<th>(1, dog)</th>
<th>(11, auk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

```plaintext
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(10, bear)</td>
</tr>
<tr>
<td>1</td>
<td>(1, dog)</td>
</tr>
<tr>
<td>2</td>
<td>(11, auk)</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**put(key):**

\[
h = \text{hash(key)}; \\
\text{while } A[h] \text{ is full:} \\
h = (h+1) \mod N \\
A[h] = \text{value}
\]
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

```plaintext
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(10, bear)</td>
</tr>
<tr>
<td>1</td>
<td>(1, dog)</td>
</tr>
<tr>
<td>2</td>
<td>(11, auk)</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(14, cat)</td>
</tr>
</tbody>
</table>

put(key):

```plaintext
h = hash(key);
while A[h] is full:
    h = (h+1) % N
A[h] = value
```
Open Addressing with Linear Probing

- **Open Addressing** - use empty buckets to store things that belong in other buckets.

- Which empty bucket? Using the next empty one is called **Linear Probing**

```
put(key):
    h = hash(key);
    while A[h] is full:
        h = (h+1) % N
    A[h] = value
```

```
put(1, "dog");
put(11, "auk");
put(10, "bear");
put(14, "cat");
put(24, "ape");
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(10, bear)</td>
</tr>
<tr>
<td>1</td>
<td>(1, dog)</td>
</tr>
<tr>
<td>2</td>
<td>(11, auk)</td>
</tr>
<tr>
<td>3</td>
<td>(24, ape)</td>
</tr>
<tr>
<td>4</td>
<td>(14, cat)</td>
</tr>
</tbody>
</table>
Open Addressing with Linear Probing

- Problem with linear probing:
  - Hashing clustered values (e.g., 1, 1, 3, 2, 3, 4, 6, 4, 5) will result in a lot of searching.

<table>
<thead>
<tr>
<th>put(key):</th>
<th>h = hash(key);</th>
</tr>
</thead>
<tbody>
<tr>
<td>while A[h] is full:</td>
<td>h = (h+1) % N</td>
</tr>
<tr>
<td>A[h] = value</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>put(key):</th>
<th>h = hash(key);</th>
</tr>
</thead>
<tbody>
<tr>
<td>while A[h] is full:</td>
<td>h = (h+1) % N</td>
</tr>
<tr>
<td>A[h] = value</td>
<td></td>
</tr>
</tbody>
</table>

```
put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);
```

```
| 0 | (10, bear) |
| 1 | (1, dog) |
| 2 | (11, auk) |
| 3 | (24, ape) |
| 4 | (14, cat) |
```
Open Addressing with Quadratic Probing

- **Quadratic Probing**: Jump further ahead to avoid clustering of full buckets.

Linear probing looks at $H, H+1, H+2, H+3, H+4, \ldots$

Quadratic probing looks at $H, H+1, H+4, H+9, H+16, \ldots$

```
put(key):
H = hash(key);
i = 0;
while A[h] is full:
    h = (H + i^2) % N
    i++;
A[h] = value
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(10, bear)</td>
</tr>
<tr>
<td>1</td>
<td>(1, dog)</td>
</tr>
<tr>
<td>2</td>
<td>(11, auk)</td>
</tr>
<tr>
<td>3</td>
<td>(24, ape)</td>
</tr>
<tr>
<td>4</td>
<td>(14, cat)</td>
</tr>
</tbody>
</table>

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);
Open Addressing with Quadratic Probing

- **Quadratic Probing:** Jump further ahead to avoid clustering of full buckets.

Linear probing looks at H, H+1, H+2, H+3, H+4, ...
Quadratic probing looks at H, H+1, H+4, H+9, H+16, ...

```
put(key):
H = hash(key);
i = 0;
while A[h] is full:
    h = (H + i^2) % N
    i++;
A[h] = value
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(10, bear)</td>
</tr>
<tr>
<td>1</td>
<td>(1, dog)</td>
</tr>
<tr>
<td>2</td>
<td>(11, auk)</td>
</tr>
<tr>
<td>3</td>
<td>(24, ape)</td>
</tr>
<tr>
<td>4</td>
<td>(14, cat)</td>
</tr>
</tbody>
</table>

put(1, “dog”);
put(11, “auk”);
put(10, “bear”);
put(14, “cat”);
put(24, “ape”);
Open Addressing with Quadratic Probing

- **Quadratic Probing**: Jump further ahead to avoid clustering of full buckets.

**Exercise**: Which buckets are full after the following insertions into an array size of 10 using quadratic probing?

```
put(0, "ape");
put(1, "dog");
put(20, "elf");
put(21, "auk");
put(40, "bear");
put(41, "cat");
put(60, "elk");
put(61, "imp");
```

```
put(key):
    H = hash(key);
    i = 0;
    while A[h] is full:
        h = (H + i^2) % N
        i++;
    A[h] = value
```
Open Addressing with Quadratic Probing

- **Quadratic Probing**: Jump further ahead to avoid clustering of full buckets.

**Exercise**: Which buckets are full after the following insertions into an array size of 10 using quadratic probing?

```plaintext
put(0, "ape"); 0
put(1, "dog"); 1
put(20, "elf"); 0, 1, 4
put(21, "auk"); 1, 2
put(40, "bear"); 0, 1, 4, 9
put(41, "cat"); 1, 2, 5
put(60, "elk"); 0, 1, 4, 9, 6
put(61, "imp"); 1, 2, 5, 10, 7
```

```plaintext
put(key):
    H = hash(key);
    i = 0;
    while A[h] is full:
        h = (H + i^2) % N
        i++;
    A[h] = value
```
Hashing in Java

- Object has a `hashCode` method. By default, this returns the object’s address in memory.

- It needs to have the properties of a hash function!
  1. Deterministic: always returns the same value for the same object.
  2. **Equal** objects have equal hash codes.

In Java, “equal” means whatever the `equals` method says.

**Consequence:** if you change the definition of `equals` (e.g., by overriding it), you may have to override `hashCode` make sure it’s consistent!
Hashing in Java

**Consequence:** if you override `equals`, you may have to override `hashCode` to match.

```java
class Person {
    String firstName;
    String lastName;

    public boolean equals(Person p) {
        return firstName.equals(p.firstName)
            && lastName.equals(p.lastName);
    }

    public int hashCode() {
        return auxHash(firstName)
            + auxHash(lastName);
    }
}
```
Open Addressing: Runtime

• May be faster, but may not be. Depends on keys.

• There’s no free lunch: worst-case is always $O(n)$.

• In practice, average-case is $O(1)$ if you make good design decisions and insertions are not done by an adversary.
Further Reading

• CLRS 11.5: Perfect Hashing

  • You can guarantee $O(1)$ lookups and insertions if the set of keys is fixed

• C++ implementations from Google:

  • `sparse_hash_map` - optimized for memory overhead
  • `dense_hash_map` - optimized for speed
Map and HashMap

• Map is an ADT

• HashMap is an implementation of a Map using a Hash Table.

• TreeMap is a thing too - some of you already wrote one!

  • AVL tree: store a key and a value in each node; BST property applies to keys only

  • Example: TreeMap<String, Integer> maps words to the number of times they have been seen
TreeMap vs HashMap

- Runtime of put, get, and remove:
  - TreeMap has $O(\log n)$ worst and expected
  - HashMap has $O(1)$ expected, $O(n)$ worst; better in practice

- Other considerations:
  - TreeMaps enable sorted traversal of keys
  - HashMaps are space-inefficient if load factor is small