CSCI 241

Lecture 18
Map ADT, Rehashing, Open Addressing
Apply to premajor
Troubleshoot registration
Prepare for winter quarter registration
Get all your questions answered

3 pm
Thurs. Nov. 8
CF 420

Email Mary.Hall@wwu.edu with any questions!

Group Advising for CS Premajors!

For disability resources, contact 360-650-3083
MIX IT UP
INTERSECTIONS
BETWEEN THE
ARTS & SCIENCES

FEATURING:
SAMMUS
PHD SCIENCE AND TECHNOLOGY
CANDIDATE AND RAPPER

MAC GYM
NOVEMBER 8
5PM

FREE FOOD
LIVE PERFORMANCE
RAFFLE PRIZES
LIQUID NITROGEN ICE CREAM
SCIENCE CLUB DEMOS
PHOTO BOOTH

WESTERN UNIVERSITY
COLLEGE OF SCIENCES AND ENGINEERING

[Additional text and logos at the bottom]
Goals

• 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.
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 $F(a) \rightarrow b$ 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

- Arrays are great!
- Domain: 0..a.length
- Range: all elements stored in the array
- Codomain: the type of elements stored in the array.
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
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

**Range:**
- Thing
  - int thingField1
  - int thingField2
- Thing
  - int thingField1
  - int thingField2

**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)
The Map Interface

```java
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 *
     * if it is present */
    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

\[ h(k) = k \% A.\text{length} \]

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

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

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
</table>
| 0     |     | "bear"
| 1     | 1   | "dog"  |
| 2     |     | "auk"  |
| 3     |     |        |
| 4     | 14  | "cat"  |
| 5     |     |        |
| 6     |     |        |
| 7     |     |        |
| 8     |     |        |
| 9     |     |        |
| 10    | 10  | "bear" |
| 11    | 11  | "auk"  |
| 14    | 14  | "cat"  |
| 24    | 24  | "ape"  |

```java
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.

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

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 in belongs in the new array.

\[
\begin{array}{c|c}
0 & 10 \text{ “bear”} \\
1 & 1 \text{ “dog”} \rightarrow 11 \text{ “auk”} \\
2 & \\
3 & \\
4 & 14 \text{ “cat”} \rightarrow 24 \text{ “ape”} \\
5 & \\
6 & \\
7 & \\
8 & \\
9 & \\
\end{array}
\]

\[
\begin{array}{c|c}
0 & 24 \text{ “ape”} \\
1 & 10 \text{ “bear”} \rightarrow 1 \text{ “dog”} \\
2 & 11 \text{ “auk”} \rightarrow 14 \text{ “cat”} \\
3 & \\
\end{array}
\]

\[(10 \mod 3) \rightarrow 1\]
\[(1 \mod 3) \rightarrow 1\]
\[(11 \mod 3) \rightarrow 2\]
\[(14 \mod 3) \rightarrow 2\]
\[(24 \mod 3) \rightarrow 0\]
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>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 “bear”</td>
<td>11 “auk”</td>
<td></td>
</tr>
<tr>
<td>24 “ape”</td>
<td>1 “dog”</td>
<td>14 “cat”</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 $N = \text{array size}$
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 $N$ buckets
visits $n$ entries (total)

could be $O(n) = (\cdot$
Rehashing: Runtime

Rehashing algorithm:

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

Let N = array size
Let n = number of entries

Overall runtime is $O(N + n^2)$

visits N buckets
visits n entries (total)
could be $O(n) =$
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></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>h = hash(key);</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>while A[h] is full:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h = (h+1) % N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A[h] = value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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**

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

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

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></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**

<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>0</th>
<th>(10, bear)</th>
</tr>
</thead>
<tbody>
<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>
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**

<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>(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(key):`

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

```
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>

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

\[ H = \text{hash(key)}; \]
\[ i = 0; \]

while A[h] is full:

\[ h = (H + i^2) \mod N \]
\[ i++; \]

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
```

<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>(24, ape)</td>
</tr>
<tr>
<td>4</td>
<td>(14, cat)</td>
</tr>
</tbody>
</table>
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");
put(1, "dog");
put(20, "elf");
put(21, "auk");
put(40, "bear");
put(41, "cat");
put(60, "elk");
put(61, "imp");
```

```plaintext
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?

```
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
```

**put(key):**

\[
H = \text{hash(key)};
\]

\[
i = 0;
\]

while \(A[h]\) is full:

\[
h = (H + i^2) \mod N
\]

\[
i++;
\]

\[
A[h] = \text{value}
\]
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 override `equals`, you have to override `hashCode` to match.
Hashing in Java

**Consequence:** if you override `equals`, you 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