CSCI 241

Lecture 17
Some more hashing, Intro to Graphs
Announcements
Goals

• 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.

• Know the definition of a **graph** and basic associated terminology:
  
  • Node/vertex; edge/arc; directed, undirected; adjacent; (in/out)-degree; path; cycle;
Hash Functions:
Necessary Properties
Hash Functions: Necessary Properties

If $h$ is a hash function, then:
Hash Functions: 
Necessary Properties

If $h$ is a hash function, then:

• $h$ is **deterministic** and **fast to compute**: for some fixed key $k$, $h(k)$ always returns the same value and is efficiently computable (usually $O(1)$)
Hash Functions: Necessary Properties

If $h$ is a hash function, then:

- $h$ is **deterministic** and **fast to compute**: for some fixed key $k$, $h(k)$ always returns the same value and is efficiently computable (usually $O(1)$)

- Equal objects hash to equal values: $h(i) == h(j)$ if $i.equals(j)$
Hash Functions: Necessary Properties

If $h$ is a hash function, then:

- $h$ is **deterministic** and **fast to compute**: for some fixed key $k$, $h(k)$ always returns the same value and is efficiently computable (usually $O(1)$)

- Equal objects hash to equal values:
  $h(i) == h(j)$ if $i.equals(j)$

- **Collisions are possible**: If $\neg i.equals(j)$ it is possible that $h(i) == h(j)$
  or: $h(i) == h(j)$ does not imply $i.equals(j)$
Hash Functions:
Desirable Properties
Hash Functions: Desirable Properties

We would *like* our hash functions distribute values evenly among buckets.

It’s hard to guarantee this without knowing keys ahead of time, but usually easy in practice using heuristics.
Hash Functions: Desirable Properties

A universally terrible hash function: $h(k) = 0$

Hash function quality often depends on the keys. e.g., if keys are WWU CSCI course numbers:

- $h(k) = k \% 100$ (1’s place)
  - bad because many collisions (141, 241, 301, ...)

- $h(k) = k / 100$ (100’s place)
  - bad because this will only use buckets 0..6

One weird tip: make the table size prime so divisibility patterns in keys don’t result in patterns in hash buckets.
Hashing Multiple Integers
• Various heuristic methods:
  • \((a + b + c + d) \% N\)
  • \((a k^1 + b k^2 + c k^3 + d k^4) \% N\)

Hashing Strings
• Interpret ASCII (or unicode) representation as an integer.
• Java String uses:
  \(s[0]\times31^{(n-1)} + s[1]\times31^{(n-2)} + \ldots + s[n-1]\)
Collision Resolution
Collision Resolution

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

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

• + Easy to implement
Collision Resolution

- **Chaining** - use a LinkedList to store multiple elements per bucket.
  - + Easy to implement
  - - Wastes space (linked list overhead)
Collision Resolution

- **Chaining** - use a LinkedList to store multiple elements per bucket.
  - + Easy to implement
  - - Wastes space (linked list overhead)
  - - Wastes time (pointer lookups, cache locality)
Collision Resolution

• **Chaining** - use a LinkedList to store multiple elements per bucket.
  
  • + Easy to implement
  
  • - Wastes space (linked list overhead)
  
  • - Wastes time (pointer lookups, cache locality)

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

• **Chaining** - use a LinkedList to store multiple elements per bucket.
  • + Easy to implement
  • - Wastes space (linked list overhead)
  • - Wastes time (pointer lookups, cache locality)

• **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");
```

**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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<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></td>
</tr>
</tbody>
</table>

**put(key):**

```python
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");
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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 = 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**

<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></td>
</tr>
</tbody>
</table>

```java
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**

<table>
<thead>
<tr>
<th>Key</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>(14, cat)</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>h</th>
<th>(key, 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>
```

```python
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 \( H, H+1, H+2, H+3, H+4, \ldots \)

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

\[
\text{put}(key):
\]
\[
H = \text{hash}(key);
\]
\[
i = 0;
\]
\[
\text{while A[h] is full:}
\]
\[
h = (H + i^2) \mod N
\]
\[
i++;
\]
\[
A[h] = \text{value}
\]

<table>
<thead>
<tr>
<th>Index</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>bear</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>dog</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>auk</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td>ape</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>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 $H, H+1, H+2, H+3, H+4, \ldots$

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

<table>
<thead>
<tr>
<th>put(key):</th>
<th>$H = \text{hash}(\text{key});$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$i = 0;$</td>
</tr>
<tr>
<td>while $A[h]$ is full:</td>
<td>$h = (H + i^2) \mod N$</td>
</tr>
<tr>
<td></td>
<td>$i++;$</td>
</tr>
<tr>
<td></td>
<td>$A[h] = \text{value}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>put(key)</th>
<th>(key, value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(1, “dog”);</td>
<td>(10, bear)</td>
</tr>
<tr>
<td>put(11, “auk”);</td>
<td>(1, dog)</td>
</tr>
<tr>
<td>put(10, “bear”);</td>
<td>(11, auk)</td>
</tr>
<tr>
<td>put(14, “cat”);</td>
<td>(24, ake)</td>
</tr>
<tr>
<td>put(24, “ape”);</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?

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

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

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

• **Scenario 1: You are using a class** that someone else wrote.
  • All Java objects (i.e., non-primitive types) inherit from Object.
  • If you want to put an instance of the class in a hash table, you don’t need to know how to hash it!
  • Just call its **hashCode** method.
Hashing in Java

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

• Scenario 2: You are writing a class.
  • Its `hashCode` method 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 it means.
Hashing in Java

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

• Scenario 2: You are writing a class.
  
  • Its `hashCode` method 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 it means.

Consequence: if you change the definition of `equals` (e.g., by overriding it), you may have to override `hashCode` make sure that equal objects have equal hash codes!
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);
    }
}
```
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
CSCI 241

Lecture 17
Some more hashing, Intro to Graphs
THESE AREN'T THE GRAPHS YOU'RE LOOKING FOR
Graph: a bunch of points connected by lines. The lines may have directions, or not.
This is a graph:

The internet's undersea world

The vast majority of the world's communications are not carried by oldfashioned submarine cables, but by a network of undersea optical fibres. The orange lines show the main routes of these cables. The blue symbols indicate the location of submarine cable landing stations. The green lines show the main undersea fibre optic cables. The red lines show the main undersea cable routes. The yellow lines show the main undersea cable systems. The blue symbols show the location of submarine cable landing stations.
The edges are made of these:
Social Networks
(before they were cool)
Social Networks
(before they were cool)
The USA as a graph:

- Neighboring states are connected by edges.
Electrical circuit
A bigger electrical circuit
This is not a graph:

it is a cat.
This is a graph that can recognize cats.
Graphs: Abstract View

K₅
K₃,₃
Graphs, Formally

- A directed graph (digraph) is a pair \((V, E)\) where:
  - \(V\) is a (finite) set
  - \(E\) is a set of ordered pairs \((u, v)\) where \(u, v\) are in \(V\)
  - Often (not always): \(u \neq v\) (i.e. no edges from a vertex to itself)

- An element in \(V\) is called a vertex or node

- Elements in \(E\) are called edges or arcs

- \(|V| = \) size of \(V\) (traditionally called \(n\))

- \(|E| = \) size of \(E\) (traditionally called \(m\))
An example directed graph

\[ V = \{A, B, C, D, E\} \]
\[ E = \{(A, C), (B, A), (B, C), (C, D), (D, C)\} \]
\[ |V| = 5 \]
\[ |E| = 5 \]
Graphs, Formally

• An **undirected graph** is a just like a digraph, but
  
  • $E$ is a set of **unordered** pairs $(u, v)$ where $u, v$ are in $V$
  
  \[ V = \{ A, B, C, D, E \} \]
  \[ E = \{ \{A,C\}, \{B,A\}, \{B,C\}, \{C,D\} \} \]
  \[ |V| = 5 \]
  \[ |E| = 4 \]

• An **undirected** graph can be converted to an equivalent **directed** graph:
  
  • Replace each undirected edge with two directed edges in opposite directions

• A **directed** graph can’t always be converted to an **undirected** graph.
Graph Terminology: Adjacency, Degree

• Two vertices are adjacent if they are connected by an edge
• Nodes \( u \) and \( v \) are called the source and sink of the directed edge \((u, v)\)
• Nodes \( u \) and \( v \) are endpoints of an edge \((u, v)\) (directed or undirected)
• The outdegree of a vertex \( u \) in a directed graph is the number of edges for which \( u \) is the source
• The indegree of a vertex \( v \) in a directed graph is the number of edges for which \( v \) is the sink
• The degree of a vertex \( u \) in an undirected graph is the number of edges of which \( u \) is an endpoint
Graph Terminology: Paths, Cycles

- A **path** is a sequence of vertices where each consecutive pair are adjacent.
- In a directed graph, paths must follow the direction of the edges (nodes must be ordered source then sink).
- A **cycle** is a path that ends where it started, e.g.: x, y, z, x
- A graph is **acyclic** if it has no cycles.