### Logistics

#### Reading
- Weiss Ch 20: Hash Table
- Weiss Ch 6.7-8: Maps/Sets

#### Upcoming
- Tue 3/21: Midterm review
- Tue 3/21: PA03 due at midnight
- Wed 3/23: Midterm Exam

#### Goals Today
- Hash Functions
- Separate Chaining In Hash Tables
So far

- **Know:** how to use int xhc = x.hashCode();
- Simple Hash Set with add(x)/contains(x) has an array hta
- Put x in hta[] based on xhc

Answer

- What if xhc is out of bounds in hta?
- Unconditionally set hta[xhc] to x in add(x)?

```java
class MyHashSet<T>{
    T hta[]; int size;
    boolean contains(T x){
        int xhc = x.hashCode();
        // If xhc out of bounds?
        xhc = ???;
        // Is this okay?
        return x.equals(this.hta[xhc]);
    }
    void add(T x){
        int xhc = x.hashCode();
        // If xhc out of bounds?
        xhc = ???;
        // Is this okay?
        this.hta[xhc] = x;
        this.size++;
    }
}
```
hta[] has a fixed size
The hash code xhc can be any integer
Take an absolute value of xhc if negative
Use modulo to get xhc in bounds

int n = hta.length;
hta[abs(xhc) % n] = x;

Note: For mathy reasons we’ll briefly discuss, usually make hash table size n a *prime number*
Motivation

- Put $x$ in table at $hta[xhc]$
- **Problem:** What if $hta[xhc]$ is occupied?

Separate Chaining

Most of you recognize this problem can be solved simply

- Internal array contains lists
- Add $x$ to the list at $hta[xhc]$

```java
public class HashTable<T>{
    private List<T> hta[];
    ...
```
Separate Chaining: Example

**Code**

```java
String[] sa1 = new String[] {
    "Chris", "Sam", "Beth", "Dan"
};

SeparateChainHS<String> h =
    new SeparateChainHS<String>(11);

for(String s : sa1) {
    h.add(s);
}
print(h.load());
// load = 4 / 11
// 0.36363636363636365
```

**Load = 0.36**

```
load = \frac{\text{item count}}{\text{array length}}
```
Separate Chaining: Example

Code

String [] sa2 = new String[]{
    "Chris","Sam","Beth","Dan",
    "George","Kevin","Nikil",
    "Mark","Dana","Amy","Foo",
    "Spike","Jet","Ed"
};

SeparateChainHS<String> h =
    new SeparateChainHS<String>(11);

for(String s : sa2){
    h.add(s);
}

h.load();
// load = 14 / 11
// 1.2727272727272727
A Set has at most one copy of any element (no duplicates)

Write add/remove/contains for SeparateChainingHS

What are the time complexities of each method?

```java
public class SeparateChainingHS<T>{
    private List<T> hta[];
    private int itemCount;

    // Constructor, n is initial size of hta[]
    public SeparateChainingHS(int n){
        this.itemCount = 0;
        this.hta = new List<T>[n];
        for(int i=0; i<n; i++){
            this.hta[i]=new LinkedList<T>();
        }
    }

    public void add(T x);       // Add x if not already present
    public void remove(T x);    // Remove x if present
    public boolean contains(T x); // Return true if x present, false o/w
}
```

Jyh-Ming Lien  
CS311 Data Structures Lecture 10 — Hash Strategy
Separate Chaining Viable in Practice

Java’s built-in hash tables use it

- Simple to code
- Reasonably efficient
- `java.util.HashSet / HashMap / Hashtable` all use separate chaining

Analyses of methods are influenced by Load

\[
load = \frac{\text{item count}}{\text{array length}}
\]
Analysis

add()

add(x) is $O(1)$ assuming adding to a list is $O(1)$

```java
int xhc = x.hashCode();
List l = hta[ abs(xhc) % hta.length];
l.add(x);
```

remove() / contains()

- Assume fair hash function (distributes well)
- **Load** is the average number of things in each list in the array.
- remove(x) / contains(x) must potentially look through **Load** elements to see if x is present
- Therefore complexity $O(\text{Load}) = O(\text{itemCount}/\text{arraySize})$
Alternatives to Separate Chaining

Separate Chaining works well but has some disadvantages
- Requires separate data structure (lists)
- Involves additional level of indirection: elements are two or three additional memory references away from the hash table array
- Adding requires memory allocation for nodes/lists

Alternative: Open Address Hashing
- Ban the use of lists in the hash table
- Store element references *directly in hash table array*
- Why do it this way?
- How can we handle collisions now?
Open Addressing

Basic Design

- Hash table elements stored in array hta (no auxiliary lists)
- **Probe a sequence** of entries for object

# Generic pseudocode for a probe sequence
```java
pos = abs(x.hashCode() % hta.length);
repeat
  if hta[pos] is empty
    hta[pos] = x
    return
  else
    pos = someplace else
```

Design Issues

- Obvious *next* places to look after pos?
- How to indicate an entry is empty?
- Limits?
Linear Probing

Start with normal insertion position $pos$

```java
int pos = Math.abs(x.hashCode() % hta.length);
```

Try the following sequence until an empty array element is found

```java
pos, pos+1, pos+2, pos+3, ... pos+i
```

Process of $\text{add}(x)$ in hash table

```java
// General idea of linear probing sequence
pos = Math.abs(x.hashCode() % hta.length);
if hta[pos] empty, put x there
else if hta[(pos+1)] empty, put x there
else if hta[(pos+2)] empty, put x there
...

// Insert x using linear probe sequence
public void add(T x)
```
With linear probing

- Can \( \text{add}(x) \) fail? Under what conditions?
- Code for \( \text{contains}(x) \)?
- How does \( \text{remove}(x) \) work?
Removal in Open Addressing: Follow Chain

<table>
<thead>
<tr>
<th>Item</th>
<th>Code</th>
<th>Pos</th>
<th>Added</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>5</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>F</td>
<td>8</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>G</td>
<td>11</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>H</td>
<td>12</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>I</td>
<td>9</td>
<td>13</td>
<td>9</td>
</tr>
</tbody>
</table>

Suppose `remove(X)` sets position to null.

What are the booleans assigned to?

```
h.remove(A); boolean b1 = h.contains(C);
h.remove(D); boolean b2 = h.contains(F);
h.remove(E); boolean b3 = h.contains(I);
```
Avoid Breaking Chains in Removal

- Don’t set removed records to null
- Use place-holders, in Weiss it’s `HashSet.HashEntry`

```java
class HashEntry {
    public Object element; // the element
    public boolean isActive; // false if marked deleted
    public HashEntry( Object e ) {
        this( e, true );
    }
    public HashEntry( Object e, boolean i ){
        element   = e;
        isActive = i;
    }
}
```

Explore `weiss/code/HashSet.java`
- `remove(x)` sets `isActive` to `false`
- `contains(x)` treats slot as filled
- `rehash()` ignores inactive entries
Load and Linear Probing

Load has a big effect on performance in linear probing

- When Inserting $x$
- If $h[cx]$ full, $cx++$ and repeat
- When $h$ is nearly full, scan most of array
- $load \approx 1 \rightarrow O(n)$ for $\text{add}(x)/\text{contains}(x)$

Theorem

The average number of cells examined during insertion with linear probing is

\[
\frac{1}{2} \left( 1 + \frac{1}{(1 - load)^2} \right)
\]

Where,

\[
load = \frac{\text{item count}}{\text{array length}}
\]
Why does this happen?

### Primary Clustering

Many keys group together, clusters degrade performance

- Table size 20
- Filled cells 5-10, 12
- Insert H hashes to 6  
  - Must put at 11
- Insert I hashes to 10  
  - Must put at 13
- Hashes from 5-13 have clustered
Quadratic Probing

Try the following sequence until an empty array element is found:

\[ pos, pos+1^2, pos+2^2, pos+3^2, \ldots pos+i^2 \]

- Primary clustering fixed: not putting in adjacent cells
- Add works up to \( \text{load} = 0.5 \)
  - Weiss Theorem 20.4, pg 786
- Can be done efficiently (Weiss pg 787)
- Complexity Not fully understood
  - No known relation of load to average cells searched
  - Interesting open research problem
Probe Sequence Differences

> Math.abs("Marylee".hashCode()) % 11
5

> Math.abs("Barb".hashCode()) % 11
5 --> Where?
Rehashing

High load $\rightarrow$ make a bigger array, rehash, get small load

- Akin to expanding backing array in ArrayList
- Allocate a new larger array
- Copy over all active items to the new array
- Array should have prime number size
- $O(n)$ to rehash
Hash Tables in Java

`java.util.HashMap` Map built from hashing
`java.util.HashSet` Set built from hashing
`java.util.Hashtable` Map built from hashing, earlier class, `synchronized` for multithread apps
Hash Take-Home

- Provide $O(1)$ add/remove/contains
- Separate chaining is a pragmatic solution
  - Hash buckets have lists
- Open Address Hashing
  - Look in a sequence of buckets for an object
- Linear probing is one way to do open address hashing
  - Simple to implement: look in adjacent buckets
  - Performance suffers load approaches 1
  - Primary clustering hurts performance
- Quadratic probing is another way to do open address hashing
  - Prevents primary clustering
  - Must keep hash half-empty to guarantee successful add
  - Not fully understood mathematically