CS311 Data Structures
Lecture 05 — Hash

Jyh-Ming Lien

June 18, 2018

Logistics

Reading

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

Goals Today

- Hash Functions
- Separate Chaining In Hash Tables
- Closed Hashing

A Small Problem

- Small office building, 50 offices
- Office numbers 0-49
- Building owner wants to track which offices are occupied along with names of occupants

Office 32	Unoccupied
Office 43	Fakebook Inc
Office 19	Unoccupied
Office 9	Banana Corp

- Suggest a standard data structure and how one would manipulate it

Arrays Rock, except...

- Small office building, 50 offices
- Office numbers based on floor
  - Floor 1: 101, 102, 103,...,110
  - Floor 2: 201, 202, 203,...,210
- Building owner wants to track which are occupied/names of occupants

Office 402	Unoccupied
Office 503	Fakebook Inc
Office 209	Unoccupied
Office 109	Banana Corp

- Adapt the earlier approach with arrays: difficulties?

How about Reverse Lookup:

- "Fakebook Inc" $\rightarrow$ Office 403
- "Banana Corp" $\rightarrow$ Office 109

Hash Tables Surmount this difficulty

- Hash Tables $\approx$ Dictionaries (Python)
- Also called *associative arrays*, sometimes *maps*
- Store objects in an array in a retrievable way
- Involves computing a number for objects to be stored
- Have $O(1)$ `add(x)/remove(x)` (sort of...)

Hash Tables are Simple

Succinctly

- Have `x` (object) to put in a hash table
- Compute integer `xhc` from `x`
  (hash code for `x` computed via a hash function provided by class of `x`)
- Put `x` in array `hta` at index `xhc`: `hta[xhc] = x;`
- `x` is now in the hash table

Things to consider

1. How do you compute `xhc`? Where should that code exist?
2. What if `xhc` is beyond of `hta.length`?
3. What if `hta[xhc]` is occupied?

Every Object's Doin' it... but not well

Every object in java has a `hashCode()` method

- Why?
- How are hash codes computed by default?
- Link to official docs

Override `hashCode()`

- For your own classes, override default `hashCode()`
- Compute hash based on the internal data of an object
- Return an integer "representing" the object
- Class is now "hashable"

Computing a Hash Code

Hash Code from Hash Function

- An integer computed for an object
- Computed via a function provided by an object:

```
int hc = thing.hashCode();
```

Hash Contract

- If `x.equals(y)` is true, then `x.hashCode()==y.hashCode()`
- Equal object $\rightarrow$ Same hash code
- Important: If `x.equals(y)` is false, hash codes may be different or the same
  - May be `x.hashCode()==y.hashCode()`
  - May be `x.hashCode()!=y.hashCode()`
- Leads to *collisions* in a hash table

Goals of a Hash Function

1. Adhere to the Hash Contract
   - If `x` and `y` are equal, must have same hash code
2. Distribute different objects "fairly" across integers
   - If `x` and `y` not equal, try to make `x.hashCode()` different from `y.hashCode()`
   - Making hash codes different reduces collisions in hash tables
3. Compute `x.hashCode()` as quickly as possible
   - Adding/looking up objects in a hash table requires computation of an object's hash code
   - Reducing time spent on computing hash code improves performance

These three goals almost always involve tradeoffs

Discussion: Hash Codes for these Fine Fellows?

`public int hashCode()`

Ideas for `hashCode()` implementation of the following things

Fundamental Types

- `Integer`
- `Long`
- `Character`
- `Boolean`
- `Float`
- `Double`

Custom Classes

- ```
  class Initials{
    char first, last;
  }
  ```
- ```
  class Coord{
    int row, col;
  }
  ```

Hash Codes for 64-bit Primitives

Straight from the Java class library source code

```
package java.lang;
public final class Double
  extends Number implements Comparable<Double>
{
    @Override
    public int hashCode() {
        return Double.hashCode(value);
    }

    public static int hashCode(double value) {
        long bits = doubleToLongBits(value);
        return (int)(bits ^ (bits >>> 32));  //^ is XOR
    }

}
```

Why XOR? What does (int)(a long number) do?

First Aggregate Example: `String.hashCode()`

```
class String {

public int hashCode(){ .. }
  Returns a hash code for this string. The hash code for a
  String object is computed as
```

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + ... + s[n-1]$$

```
  using int arithmetic, where s[i] is the ith character of
  the string, n is the length of the string, and ^ indicates
  exponentiation.
}
```

Examples

```
> "a".hashCode()
97
> "b".hashCode()
98
> "ab".hashCode()
3105
> "ba".hashCode()
3135
```

```
> String s = "Hash!";
> s.hashCode()
69497011
> (31*31*31*31)*'H' + (31*31*31)*'a' +
  (31*31)*'s' + (31)*'h' + '!'
69497011
```

Polynomial Hash Code Tricks

`String` uses a polynomial hash code

$$a_0X^{n-1} + a_1X^{n-2} + a_2X^{n-3} + \cdots + a_{n-1}X^0$$

31 is X in the above

- 31 is not special
- Early java used 37 instead

A Trick

Can regroup a polynomial of any degree
Example of regrouping degree 3 polynomial

$$a_0X^3 + a_1X^2 + a_2X^1 + a_3$$

regrouped becomes

$$(((a_0)X + a_1)X + a_2)X + a_3$$

Implementations

Slow: Original

$s[0]*31^{(n-1)}+s[1]*31^{(n-2)}+...+s[n-1]$

```
char s[];
public int hashCode() {
  int h = 0, i, n=s.length;
  for(i=0; i<n; i++){
    h += s[i] * ((int) pow(31,n-i-1));
  }
  return h;
}
```

Faster: Exploit Regrouping

```
(...(((s[0])*31
     + s[1])*31
     + s[2])*31
     + ...)
```

```
char s[];
public int hashCode() {
  int h = 0, i;
  for (i=0; i<s.length; i++){
    h = 31 * h + s[i];
  }
  return h;
}
```

Examine parens carefully in expression

The Full Implementation uses Caching

Compute once, save for later

```
class String{
  private char[] str; // Chars of string
  private int hash;   // Default to 0

  public int hashCode() {
    // Check if the hash has already been computed
    if(this.hash!=0 || this.str.length==0){
      return this.hash;
    }
    // Hasn't been computed, compute and store
    for(int i=0; i < this.str.length; i++) {
      this.hash = 31 * this.hash + this.str[i];
    }
    return this.hash;
  }
}
```

Not *exactly* how `java.util.String` looks but it's the general idea

Practice: Hash Codes for these Fine Fellows?

```
public int hashCode()
```

Ideas for `hashCode()` implementation of the following things

Fundamental Types (Done)

- `Integer`
- `Long`
- `Character`
- `Boolean`
- `Float`
- `Double`

Container Types

- `Integer []`
- `Double []`
- `String []`
- `ArrayList<T>`
- `LinkedList<T>`
-
```
class Flurb{
  int x;
  double y;
  String s;
  int [] a;
}
```

Example: Flurb Class hashCode()

```
class Flurb{
  int x;
  double y;
  String s;
  int [] a;

  public int hashCode(){
    int h = 0;
    h = h*31 + x;
    h = h*31 + (new Double(y)).hashCode();
    h = h*31 + s.hashCode();
    for(int i=0; i<a.length; i++){
      h = h*31 + a[i];
    }
    return h;
  }
}
```

Basic `hashCode()` Strategy

Poor man's strategy: `x.toString().hashCode()`
More thoroughly ...

Fundamental Types

- All have a fixed size in bytes
- `int` has 4 bytes
- Convert bytes of intrinsic to 4 bytes
- If shorter than 4 bytes like `Character`, done
- If 8 bytes like `Long`,`Double`, use `XOR` to reduce 8 to 4 bytes

Container Types

- Use `String` approach
- Polynomial hash code of elements
- For each element compute its hash code
- Update polynomial hash code
- Treat fields as part of the sequence

Summary

- Two equal objects must have the same `hashCode()` and as much as possible unequal objects should have differing hashcodes
- Consequently, every class has a `hashCode()` method but should override it when overriding `equals()`
- Fundamental types with 32 bits ore less like Integer are their own hash codes
- Fundamental types with more than 32 bits like Long can use XOR to combine 4-byte quantities to get a 32-bit hash
- Aggregate data like `String` often uses polynomial codes to calculate hash codes which differ when the order of constituents changes.
- The same approach is used for other containers and custom classes that need the order of elements reflected in their hashcodes

Hash Table Class So Far...

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)`?

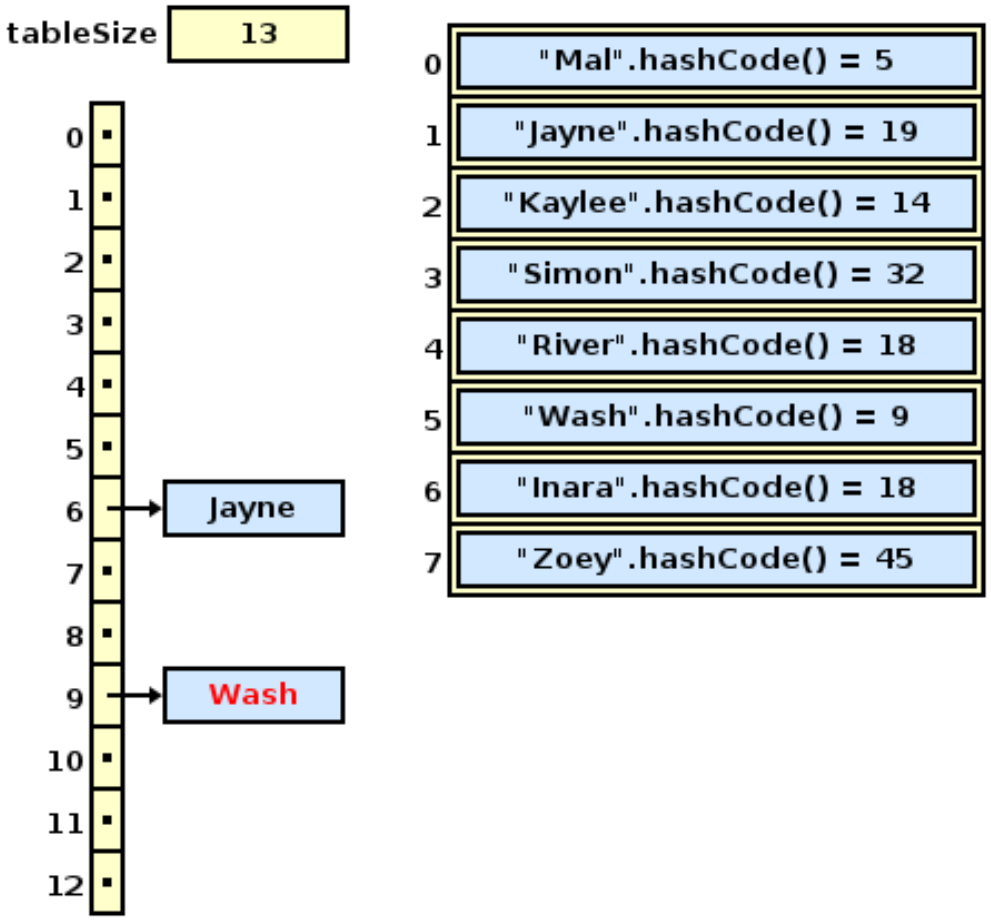
```
class MyHashSet<T>{
  T hta[];  int size;
  boolean contains(T x){
    int xhc = x.hashCode();
    // If xhc out of bounds?
    xhc = ???;
    // Is this okay?
    return
      x.equals(this.hta[xhc]);
  }
  void add(T x){
    int xhc = x.hashCode();
    // If xhc out of bounds?
    xhc = ???;
    // Is this okay?
    this.hta[xhc] = x;
    this.size++;
  }
}
```

Getting Hash Codes in Bounds

- `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*



Pragmatic Collision Resolution: Separate Chaining

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]`

```
public class HashTable<T>{
  private List<T> hta[];
  ...
```

Separate Chaining: Example

Code

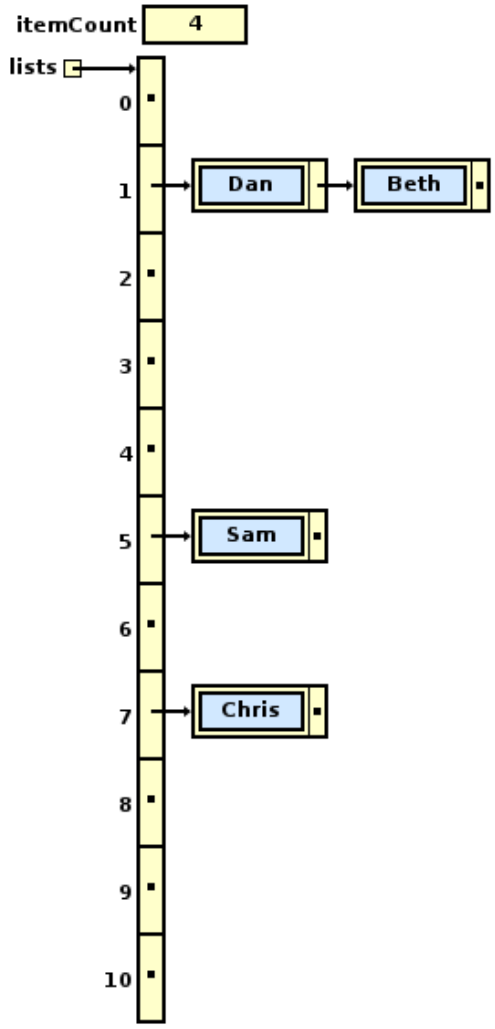
```
String [] sa1 = new String[]{
  "Chris","Sam","Beth","Dan"
};

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

for(String s : sa1){
  h.add(s);
}
print(h.load());
// load = 4 / 11
// 0.3636363636363636365
```

$$load = \frac{\text{item count}}{\text{array length}}$$

Load = 0.36



Separate Chaining: Example

Code

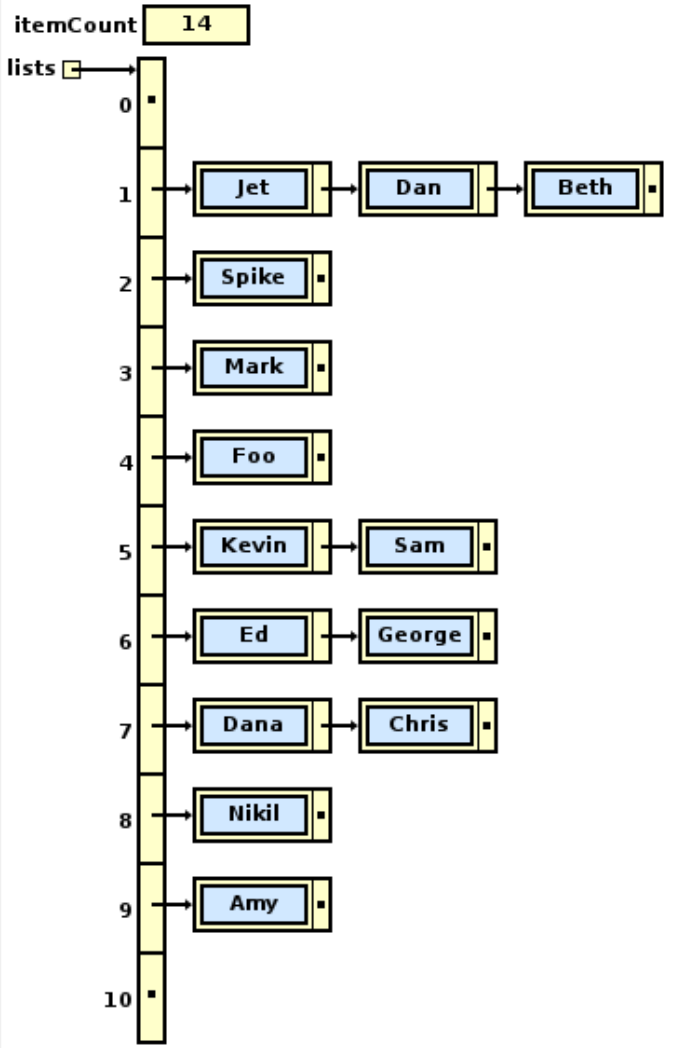
```
String [] sa2 = new String[]{
  "Chris","Sam","Beth","Dan",
  "George","Kevin","Nikil",
  "Mark","Dana","Amy","Foo",
  "Spike","Jet","Ed"
};

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

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

h.load();
// load = 14 / 11
// 1.2727272727272727
```

Load = 1.27



Implement Separate Chaining

- 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?

```
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 alread present
  public void remove(T x);       // Remove x if present
  public boolean contains(T x); // Return true if x present, false o/w
}
```

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 $O(1)$

```
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 $O(Load) = O(itemCount/arraySize)$

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 auxilliary lists)
- Probe a sequence of entries for object

```
# Generic pseudocode for a probe sequence
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

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

Try the following sequence until an empty array element is found

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

Process of add(x) in hash table

```
// 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)
```

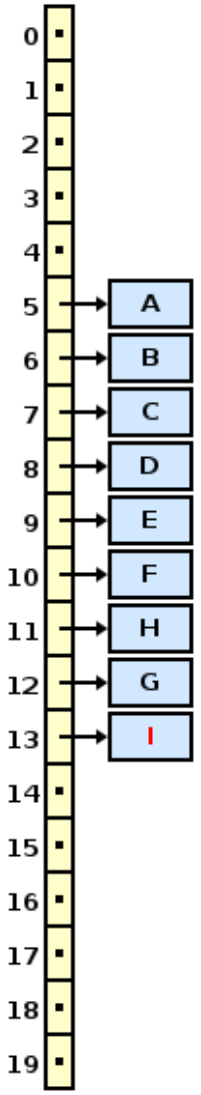
Consequences of Open Address Hashing

With linear probing

- Can `add(x)` fail? Under what conditions?
- Code for `contains(x)`?
- How does `remove(x)` work?

Removal in Open Addressing: Follow Chain

Item	Code	Pos	Added
A	5	5	1
B	6	6	2
C	5	7	3
D	7	8	4
E	5	9	5
F	8	10	6
G	11	11	7
H	12	12	8
I	9	13	9



- 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`

```
private static 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 `add(x)/contains(x)`

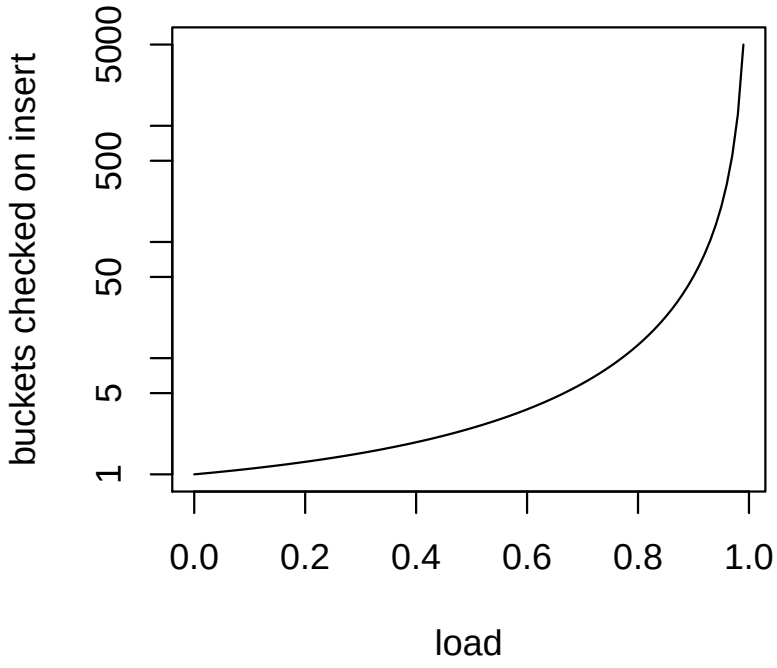
Theorem

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

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

Where,

$$load = \frac{\text{item count}}{\text{array length}}$$

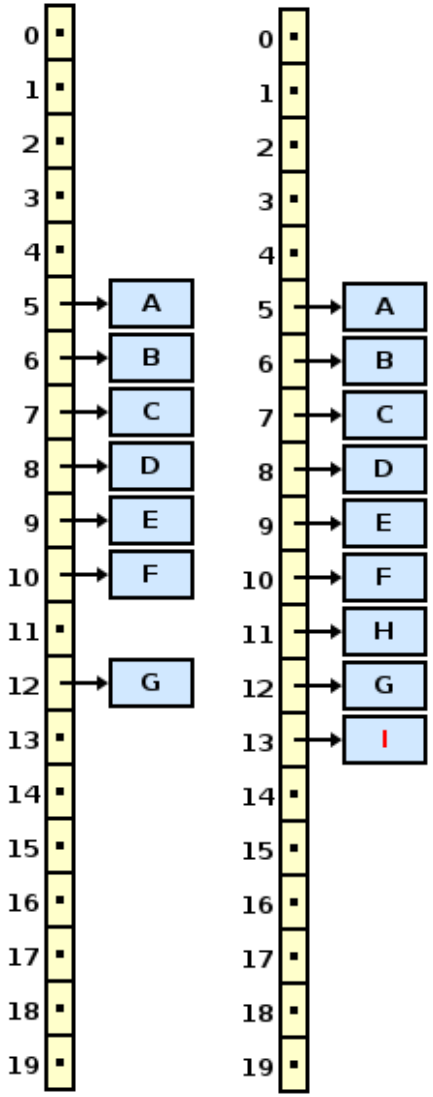


Why does this happen?

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

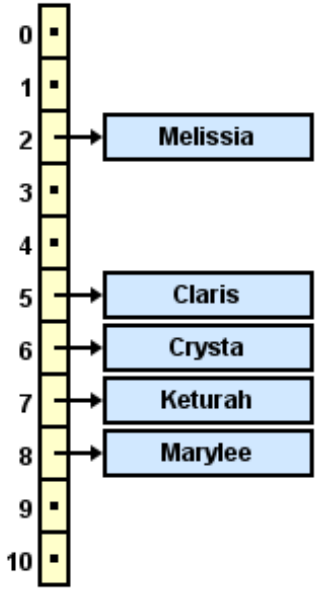
```
pos, pos+1^2, pos+2^2, pos+3^2, ... pos+i^2
```

- Primary clustering fixed: not putting in adjacent cells
- add works up to $load = 0.5$
  - 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

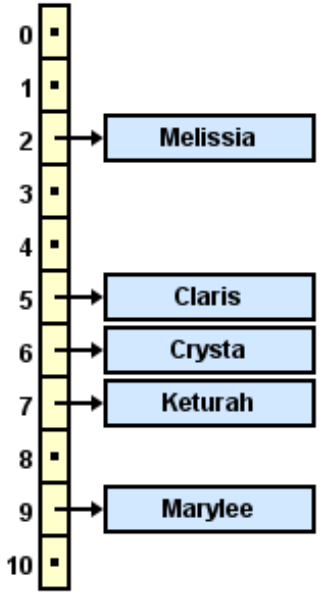
Probe Sequence Differences

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

Linear Probe



Quadratic Probe




```
> 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