CS 10: Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson
260 (255) Sudikoff

Day 11 – Hashing

Agenda

1. Hashing

2. Computing Hash functions

3. Handling collisions
   1. Chaining
   2. Open Addressing

The old Sears catalog stores illustrate how hashing works

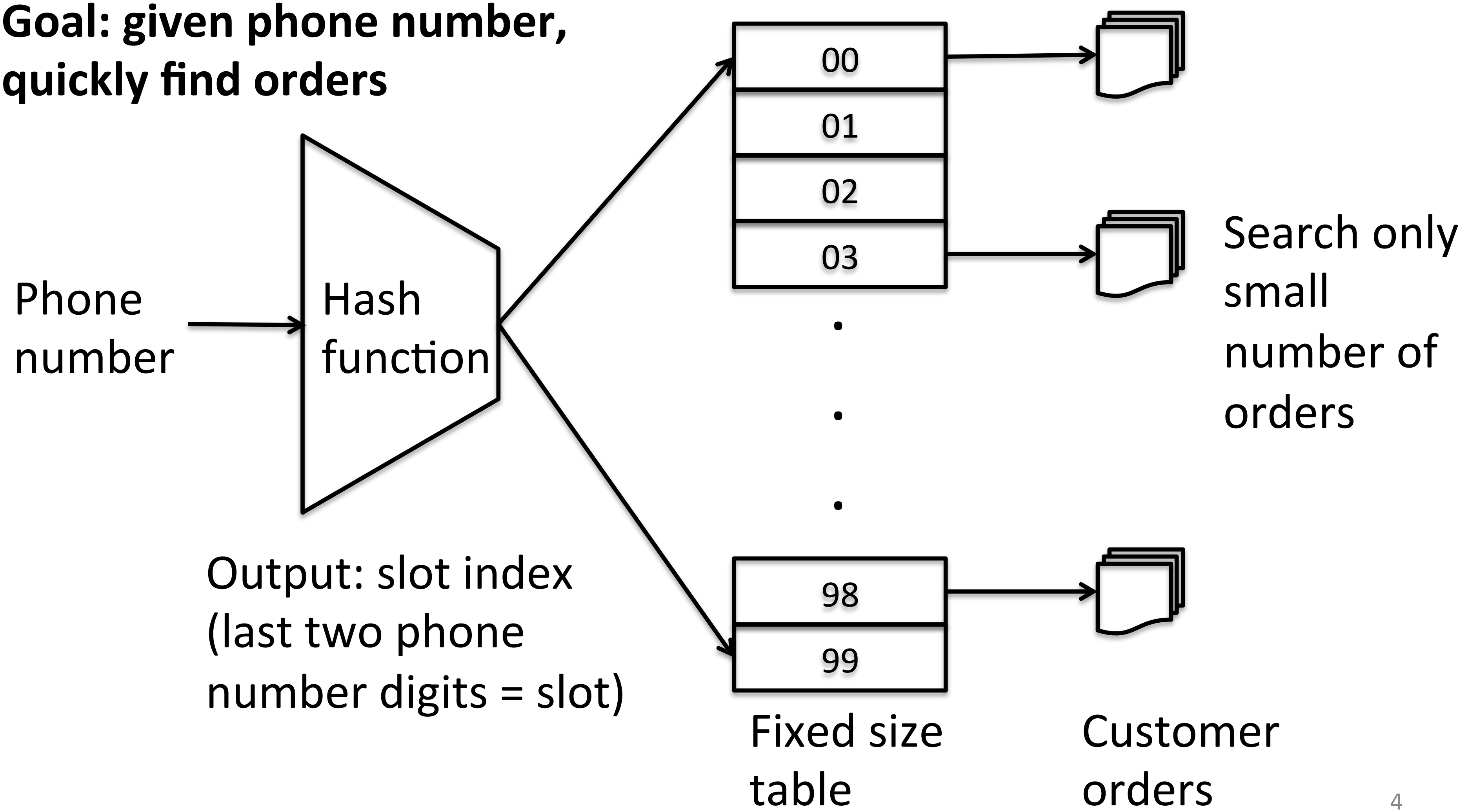
Sears store implementation of hash table

- 100 slots behind order desk, 0…99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone
- Customer arrives, clerk asks for last two digits of phone
- Given two-digit number, clerk finds slot with that number
- Clerk searches contents of that slot only
- Could be multiple orders, but can find the current customer's order quickly because only a few orders in slot
- This splits a set of (possibly) hundreds or thousands of orders into 100 slots of a few items each
- Trick: find a hash function that spreads customers evenly
- Last two digits work, why not first two?

The store is using a form of hashing based on customer's phone number

Hashing phone numbers to find orders

Goal: given phone number, quickly find orders



Can use the same idea to create Sets and Maps with better performance than Trees

Sets and Maps implemented with Trees

Set

`contains(object)` – true if has object
- Search for object in Tree
- O(h)

`add(object)` – puts object in Set
- Search for object in Tree
- Insert at leaf if not
- O(h)

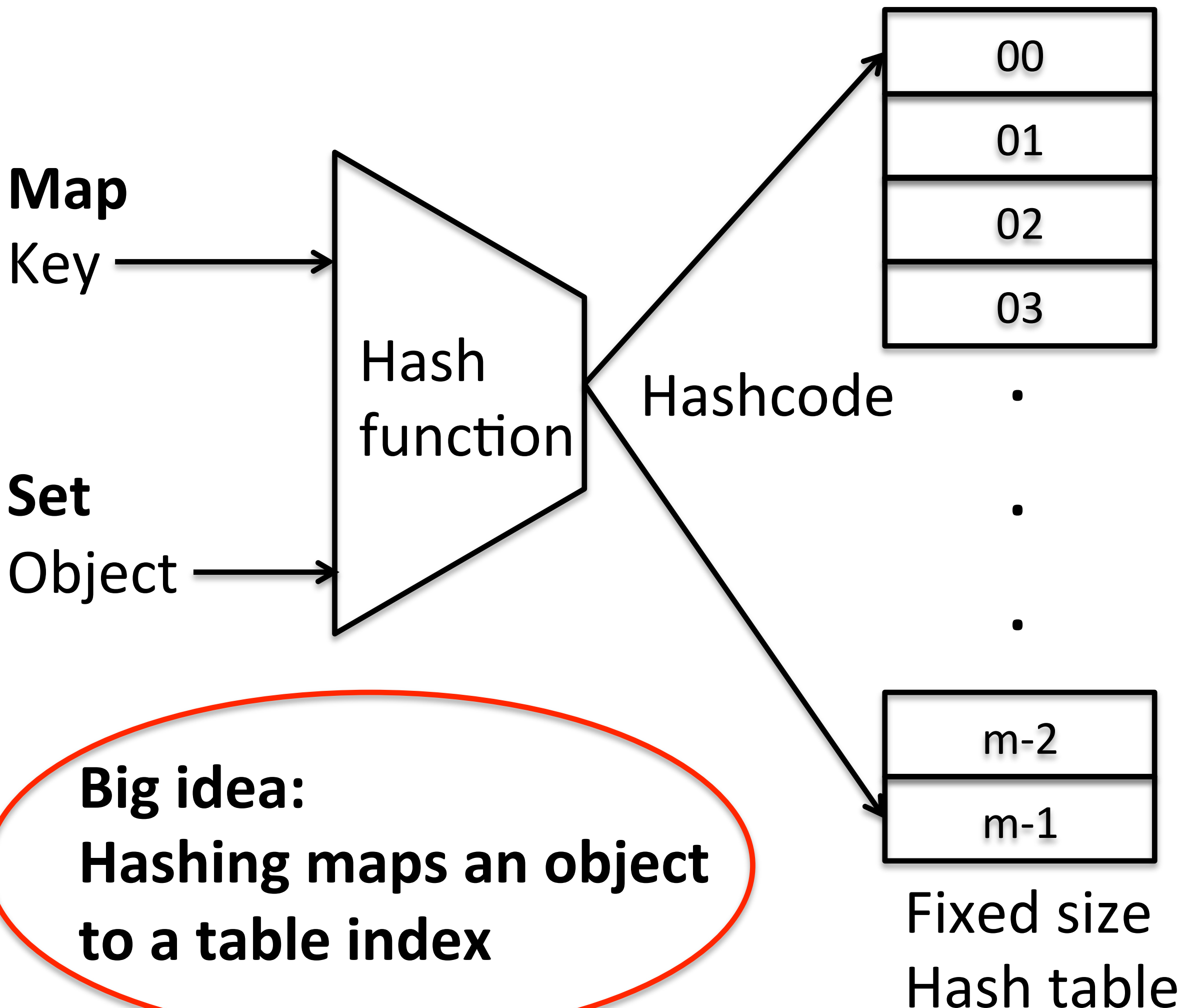
Map

`containsKey(key)` – true has key
- Search for key in Tree
- O(h)

`put(key,value)` – puts value in Map stored by key
- Search for key in Tree
- Update value if found
- Insert value at leaf if not
- O(h)

Can use the same idea to create Sets and Maps with better performance than Trees

High level overview of Hash tables



Big idea:
Hashing maps an object to a table index

Hash table

- Begin with fixed size Hash table to hold items we want to find
- Use hash function on Key or Object to give index into Hash table
- Get item from Hash table at index given by hash function O(1)

Agenda

1. Hashing

2. Computing Hash functions

3. Handling collisions
    1. Chaining
    2. Open Addressing

Good hash functions map keys to indexes in table with three desirable properties

Desirable properties of a hash function

1. Hash can be computed quickly and consistently

2. Hash spreads the universe of keys evenly over the table

3. Small changes in the key (e.g., changing a character in a string or order of letters) should result in different hash value

Suppose we used the first letter of people's names to hash, how would that work?

First letter of name as hash

1. It can be computed quickly

   Yes

2. It spreads the universe of keys evenly over the table

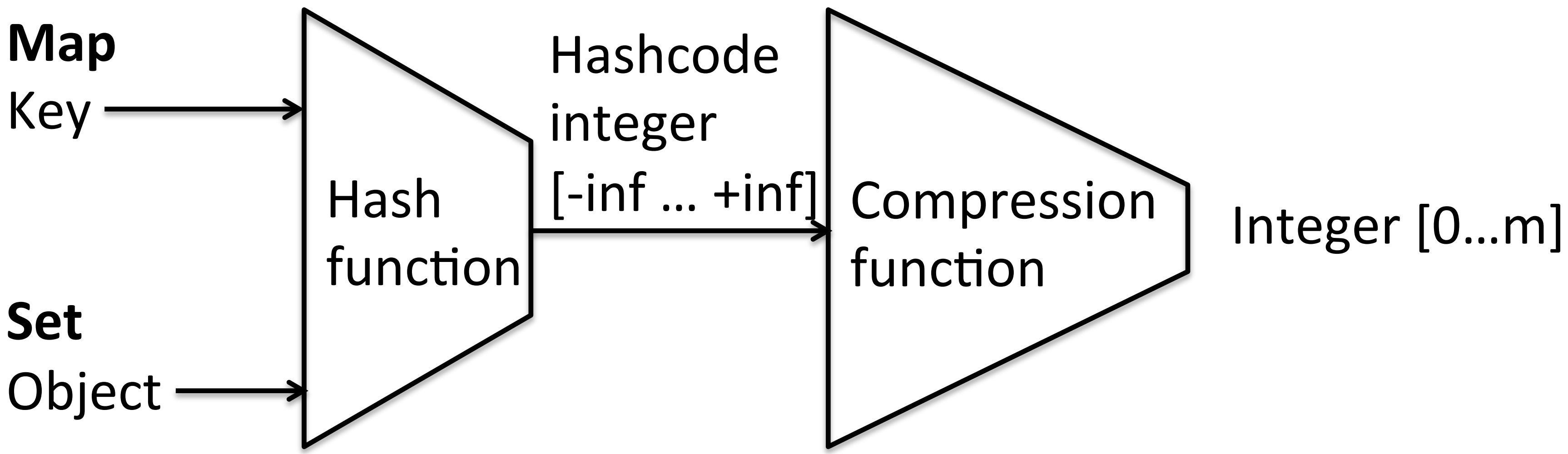
   No

3. Small changes in the key (e.g., changing a character in a string or order of letters) should result in different hash value

   Not really. Different, if change first letter, otherwise not.

Hashing is often done in two steps to map an object to a table index

Hashing as a two step process

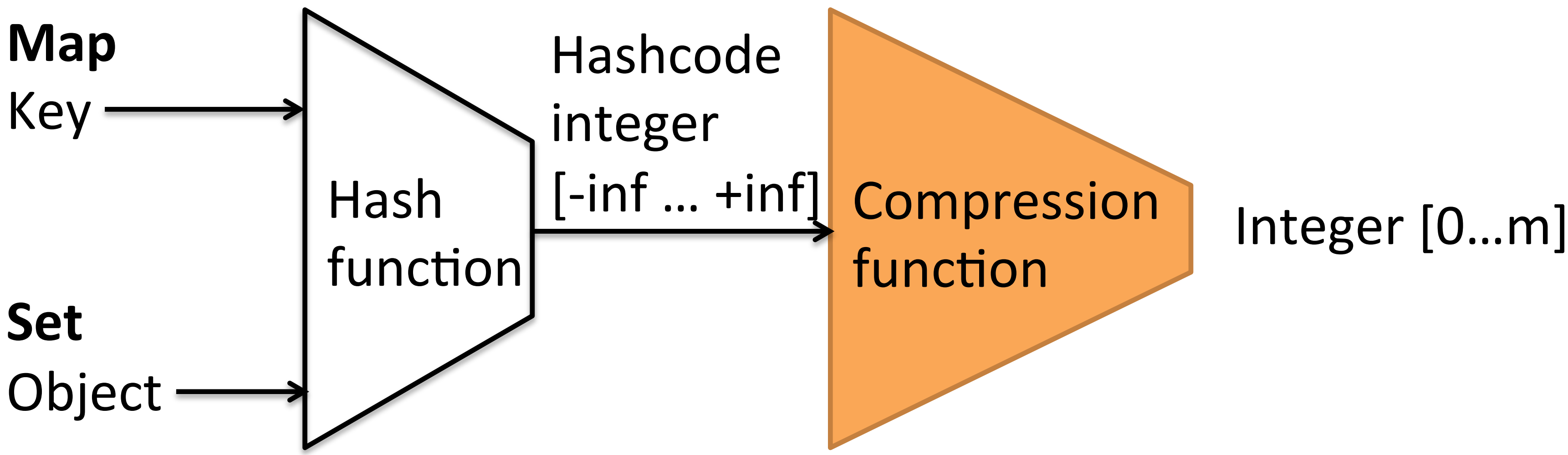


Step 1: Hash an object to an integer

Step 2: Constrain the integer hashcode to a table index

Compression function maps hashcode to table index

Hashing as a two step process



Division method:

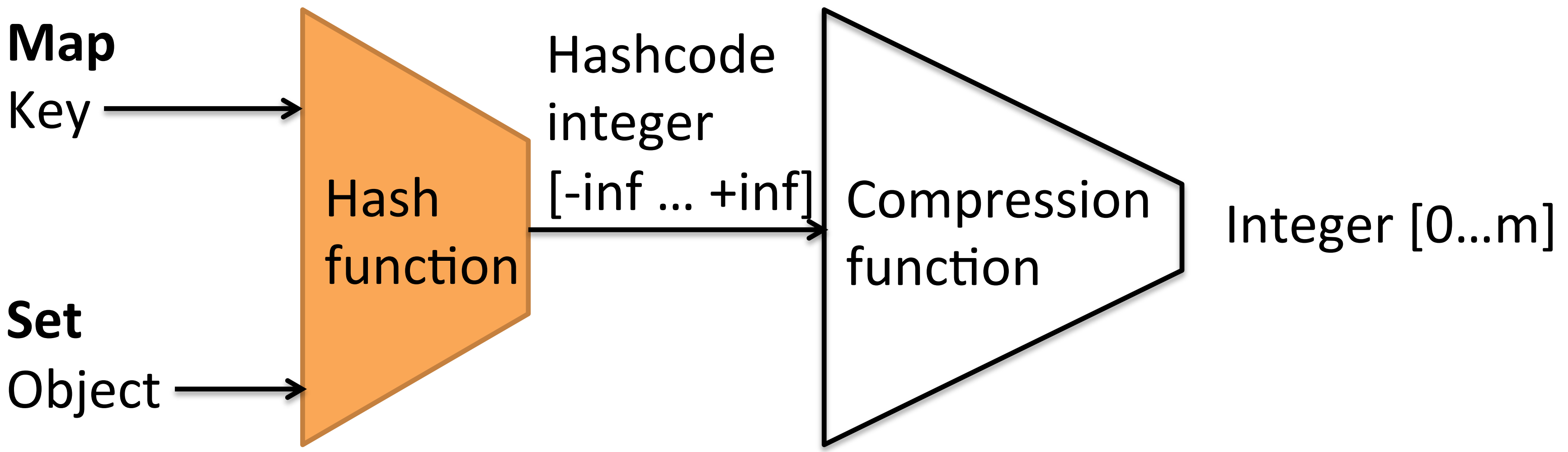
hashcode mod m

Works well if m is prime

MAD is a more complicated version

Hash function maps objects to an integer

Hashing as a two step process



For some types can just cast to integer
Works for byte, short, int, and char

For longer types, such as arrays or Strings, need a better solution
Convert object to integer with polynomial function

The polynomial method is often used for hashing more complex objects

Hashing complex objects

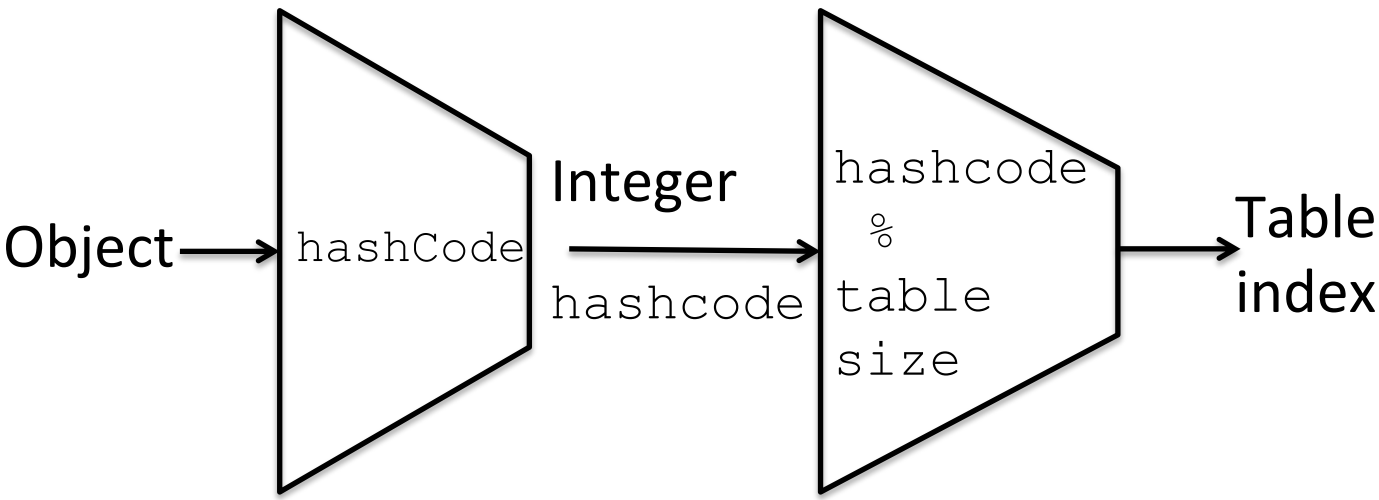
- Consider array of length s
- Pick prime number a (book recommends 33, 37, 39 or 41)
- Convert each array item to integer representation
- Calculate polynomial hashcode as $x_0a^{n-1} + x_1a^{n-2} + \ldots x_{n-2}a + x_{n-1}$
- Use Horner's rule to effeciently compute hash code

```
public int hashCode() {
    final int a=37;
    int sum = x[0]; //first item in array
    for (int j=1;j<s;j++) {
        sum = a*sum + x[j]; //array element j
    }
    return sum;
}
```

- Experiments show that when using a as above, 50,000 English words had fewer than 7 collisions

Java uses `hashCode` method and a compression function to find table index

Hash function + compression function



Hash function:
Objects must implement `hashCode` (default returns memory address)

Compression function

- Works well if table size is prime
- Books gives solution if not prime
- Java handles compression for us (take CS30 for more)

We should override `hashCode` to use objects as keys for Maps and Sets

`hashCode` for composite objects as keys

- In Java, all objects implement a `hashCode` function
- By default Java uses the memory address of the object as a hashcode and then compresses that to get a table index
- We want to hash based on values in object, not whatever memory location an object happened to be assigned
- This way two objects with same instance variables will map to the same table location (those objects are equal)
- Composite objects have several instance variables or are Strings or arrays or …
- Could just add all instance variables, but that wouldn't work well because changing order of items doesn't change hash (e.g. String)
- Can compute a polynomial based on composite object
- If you use Java's built in types (e.g., Strings, integers, doubles) as keys, you can use Java's `hashCode` methods

If you override `equals`, you must also override `hashCode`

Equals

- By default, Java will compare memory addresses to determine if two objects are equal
- Only equal when two objects point to the same memory address
- We can override equals to compare each instance variables in two objects (e.g., two Blobs, check both have same x, y, and r)

```
public boolean equals (Blob b2) {
    if (this.x != b2.getX()) return false;
    if (this.y != b2.getY()) return false;
    if (this.r != b2.getR()) return false;
    return true;
}
```

- If don't override `hashCode` function, even if equal according to code above, Java will use the memory address and look in the wrong slot

Agenda

1. Hashing

2. Computing Hash functions

3. Handling collisions
   1. Chaining
   2. Open Addressing

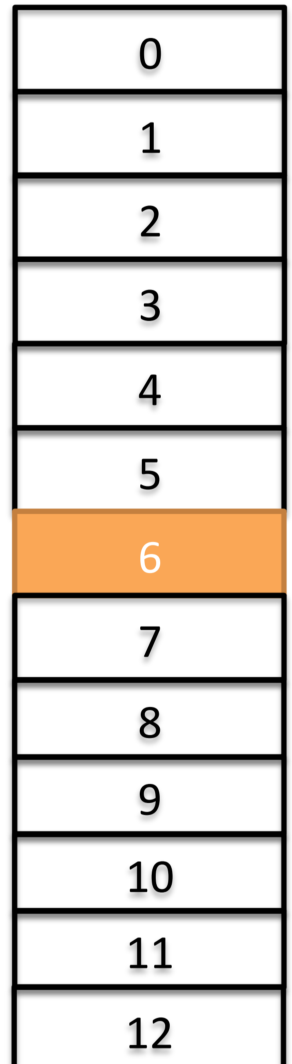
Collisions happen when multiple keys map to the same table index

Integer keys

Given table size m = 13

Compute h(key) = (key % m)

Example
- h(6) = 6



m = 13

Collisions happen when multiple keys map to the same table index

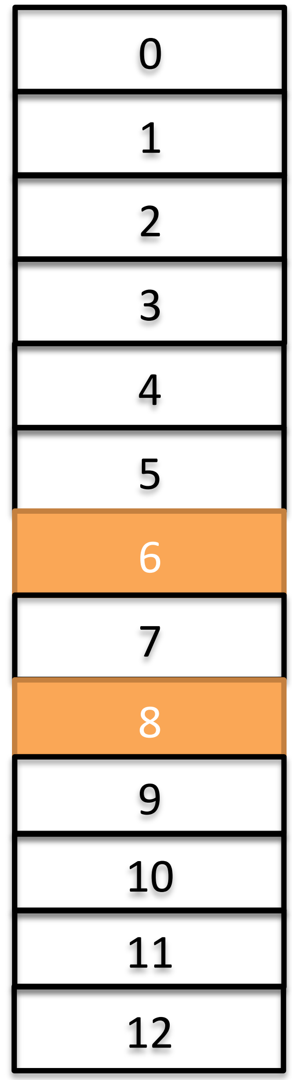
Integer keys

Given table size m = 13

Compute h(key) = (key %m)

Example

- h(6) = 6
- h(8) = 8



m = 13

Collisions happen when multiple keys map to the same table index

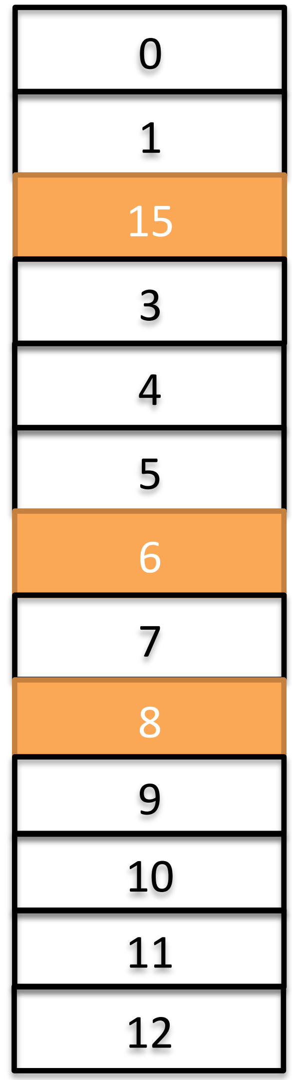
Integer keys

Given table size m = 13

Compute h(key) = (key %m)

Example

- h(6) = 6
- h(8) = 8
- h(15) = 2



m = 13

Collisions happen when multiple keys map to the same table index

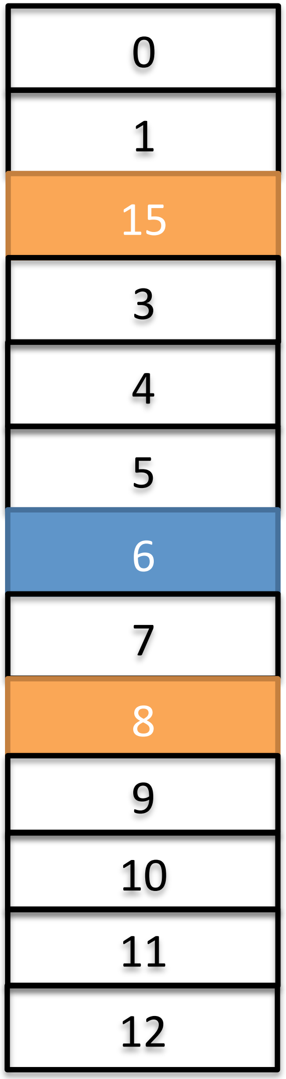
Integer keys

Given table size m = 13

Compute h(key) = (key %m)

Example

- h(6) = 6
- h(8) = 8
- h(15) = 2
- h(19) = 6



m = 13

Collision!
6 and 19 mapped to the same index

Agenda

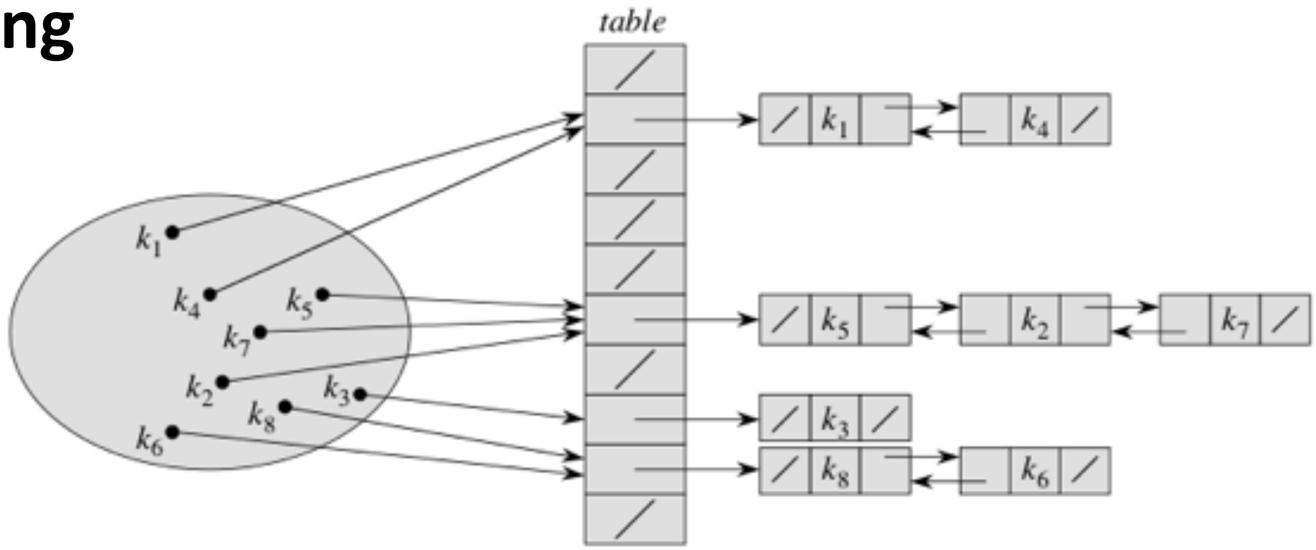
1. Hashing

2. Computing Hash functions

3. Handling collisions
   1. Chaining
   2. Open Addressing

Chaining handles collisions by creating a linked list for each table entry

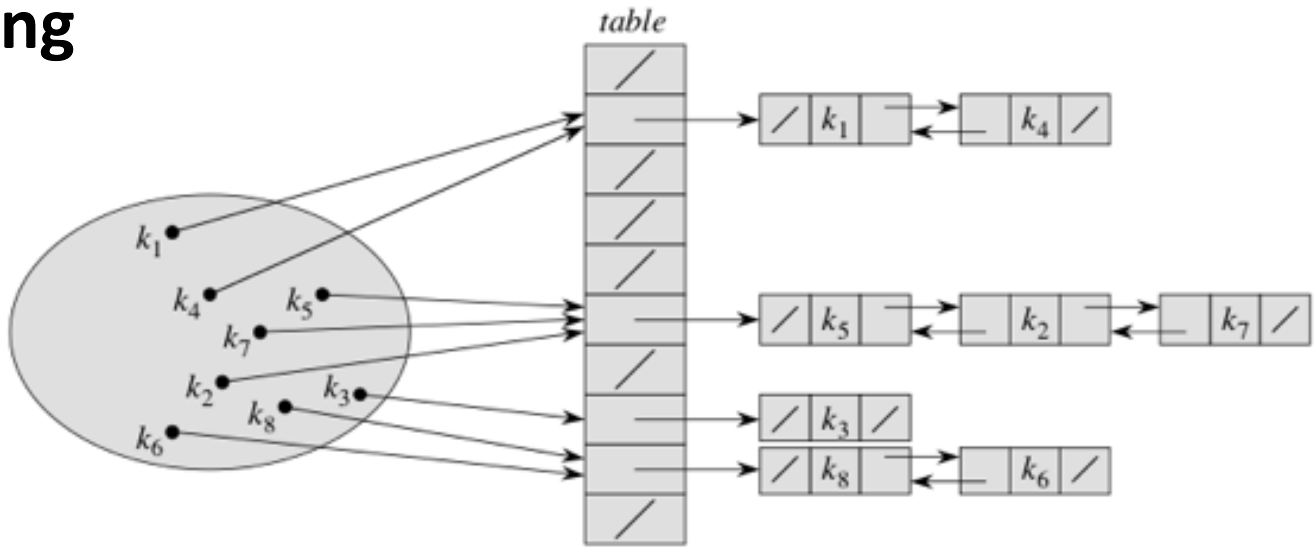
Chaining



- Create a table pointing to linked list of items that hash to the same index
- Slot *i* holds all keys *k* for which *h(k) = i*

Load factor measures number of items in the list that must be searched on average

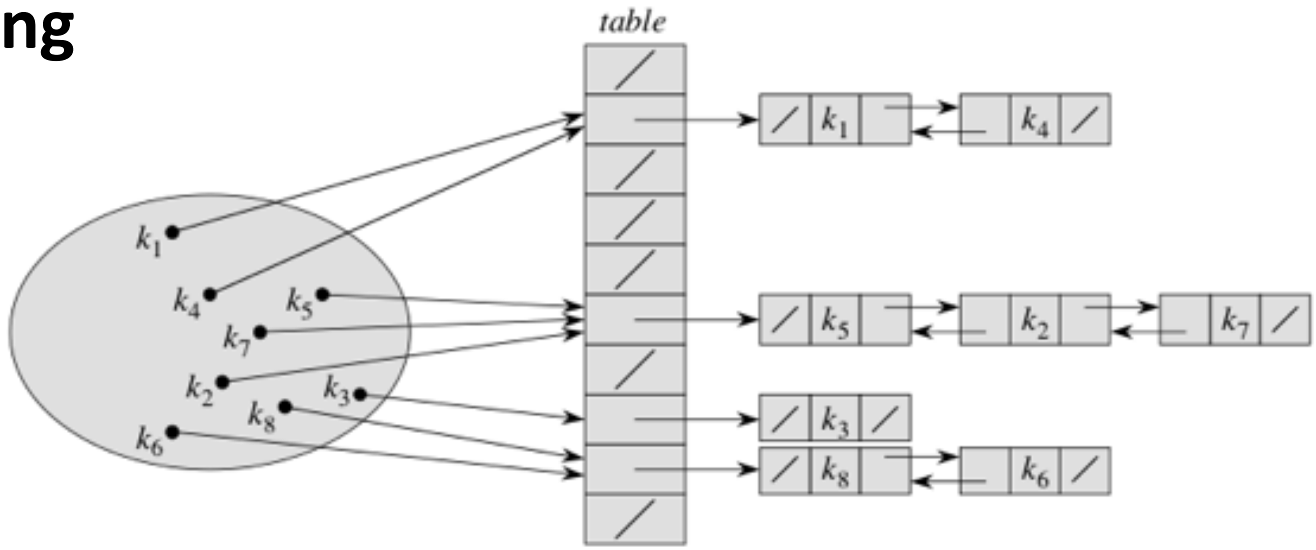
Chaining



- Assume table with *m* slots and *n* keys are stored in it
- On average, we expect *n/m* elements per collision list
- This is called the ***load factor*** *(λ)*
- So search time is *Θ(1+λ)*, assuming ***simple uniform hashing*** (each possible key equally likely to hash into a particular slot), worst case is 0(n)

Is the load factor gets too high, then we should increase the table size

Chaining



- If n (# elements) becomes larger than m (table size), then collisions are inevitable and search time goes up
- Java increases table size by 2X and rehashes into new table when $\lambda > 0.75$ to combat this problem
- Problem: memory fragmentation with link lists spread out all over, might not be good for embedded systems

Agenda

1. Hashing

2. Computing Hash functions

3. Handling collisions
   1. Chaining
   2. Open Addressing

Open addressing is different solution, everything is stored in the table itself

Open addressing using linear probing

- Insert item at hashed index (no linked list)
- For key *k* compute *h(k)=i,* insert at index *i*
- If collision, a simple solution is called linear probing
  - Try inserting at *i+1*
  - If slot *i+1* full, try *i+2*… until find empty slot
  - Wrap around to slot 0 if hit end of table at *m-1*
  - If λ <1 will find empty slot
  - If $\lambda \approx 1$, increase table size (*m*2*)
- Search analogous to insertion, compute key and probe until find answer or empty slot (key not in table)

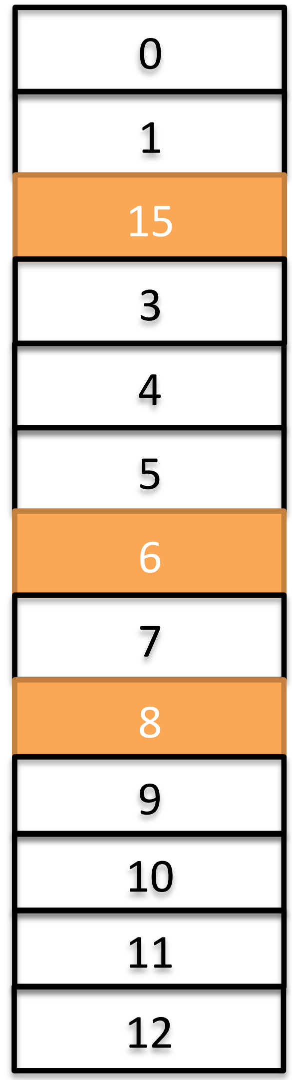
Linear probing is one way of handling collisions under open addressing

Integer keys

Given table size m = 13

Compute h(key) = (key %m)

Example
- h(6) = 6
- h(8) = 8
- h(15) = 2



m = 13

Linear probing is one method of open addressing

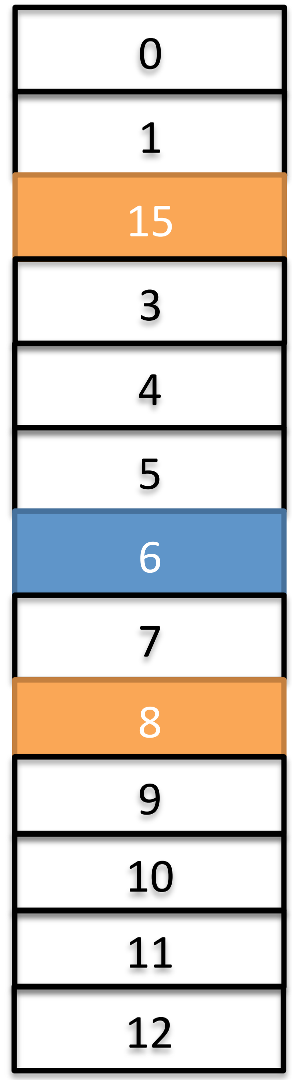
Integer keys

Given table size m = 13

Compute h(key) = (key %m)

Example

- h(6) = 6
- h(8) = 8
- h(15) = 2
- h(19) = 6



m = 13

Collision!

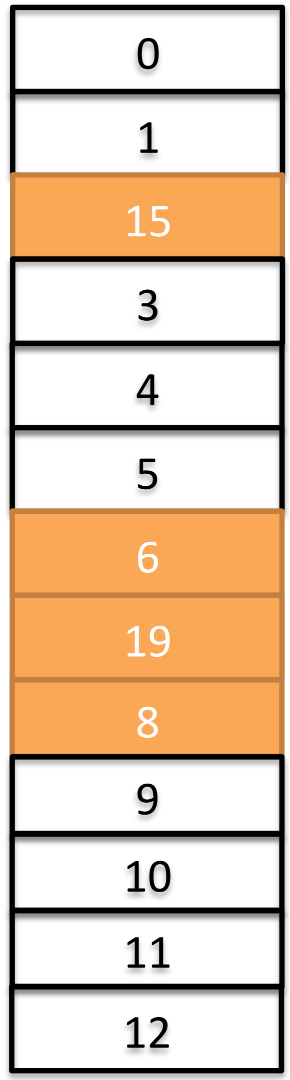
Linear probing is one method of open addressing

Integer keys

Given table size m = 13

Compute h(key) = (key %m)

Example
- h(6) = 6
- h(8) = 8
- h(15) = 2
- h(19) = 6



m = 13

Insert at i+1 = 7

Deleting items is tricky, need to mark deleted spot as available but not empty

Problems deleting items under linear probing

- Insert k_1, k_2, and k_3 where $h(k_1)=h(k_2)=h(k_3)$
- All three keys hash to the same slot in this example
- k_1 in slot *i*, k_2 in slot *i+1*, k_3 in slot *i+2*
- Remove k_2, creates hole at *i+1*
- Search for k_3
  - Hash k_3 to *i*, slot *i* holds $k_1 \neq k_3$, advance to slot *i+1*
  - Find hole at *i+1*, assume k_3 not in hash table
- Can mark deleted spaces as available for insertion, and search moves on from marked spaces
- This can be a problem if many deletes create many marked slots, search approaches linear

Clustering of keys can built up and reduce performance

Clustering problem

- Long runs of occupied slots (clusters) can build up increasing search and insert time
- Clusters happen because empty slot preceded by *t* full slots gets filled with probability *(t+1)/m,* instead of *1/m* (e.g., *t* keys can now fill open slot instead of just 1 key)
- Clusters can bump into each other exacerbating the problem

Clustering of keys can built up and reduce performance

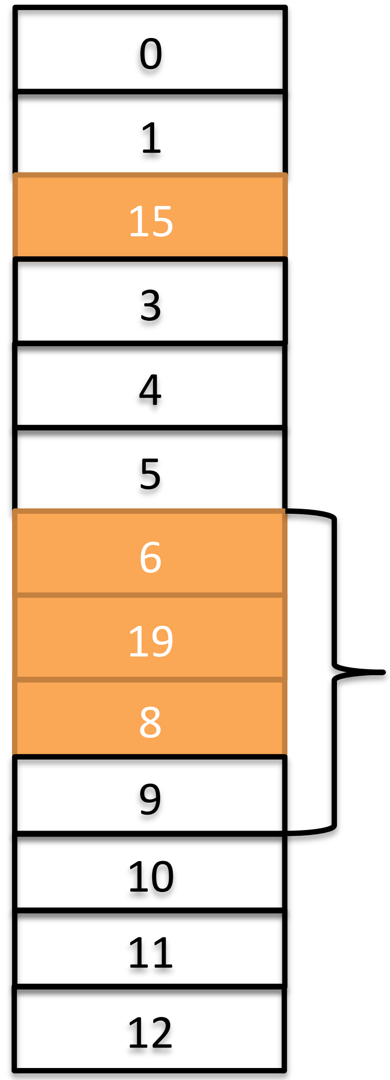
Integer keys

Given table size m = 13

Compute h(key) = (key %m)

Example

- h(6) = 6
- h(8) = 8
- h(15) = 2
- h(19) = 6



m = 13

Hashing 6,7,8, or 9 go into index 9

Makes index 9 more likely to be filled than other slots

Double hashing can help with the clustering problem

Double hashing

- Use two hash functions h_1 and h_2 to make a third *h'*
- $h'(k,p)=(h_1(k) + ph_2(k))$ mod *m*, where *p* number of probes
- First probe $h_1(k)$, $p=0$, then p incremented by 1
- If collision, next probe is offset by $h_2(k)$, then mod *m*
- Need to design hashes so that if $h_1(k_1)=h_1(k_2)$, then ***unlikely*** $h_2(k_1)=h_2(k_2)$

Run time complexity is O(1/(1-λ))

Insert and search time

- Run time gets large as λ gets large
- If table 90% full, then need about 10 probes for insert or unsuccessful search
- Successful search completes a little faster, about 2.5 probes (math on course web page)
- This means we need to grow table to keep it sparsely populated or performance suffers