CS 10: Problem solving via Object Oriented Programming

Hashing

Java provides us faster Sets and Maps using hashing instead of Trees

- Sets hold unique objects, Maps hold Key/Value pairs
- Map Keys are unique, but Values may be duplicated
- As we saw last class, using a Tree is a natural fit for implementing Sets and Maps
- Performance with a Tree is *generally* better than a List
- We can do better than Tree performance by using today's topic of discussion – hashing
- We trade memory for speed!
- Java provides the HashSet and HashMap out-of-the-box that do a lot of the hard work for us

Agenda

1. Hashing

Key points:

1. **Hashing maps a key to a table index**
2. **We can use this concept to implement Maps and Sets**

2. Computing Hash functions

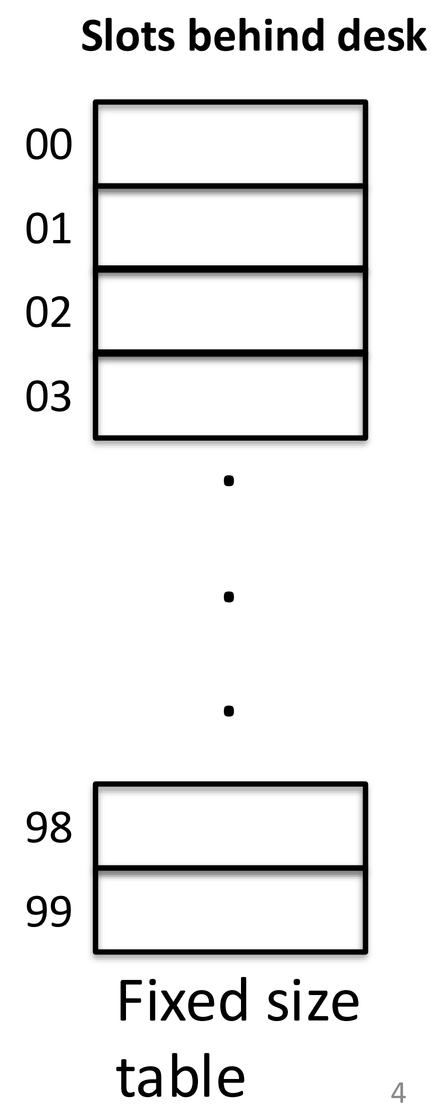
3. Implementing Maps/Sets with hashing

4. Handling collisions
    1. Chaining
    2. Open Addressing

The old Sears catalog orders illustrate how hashing works

Sears store implementation of hash table

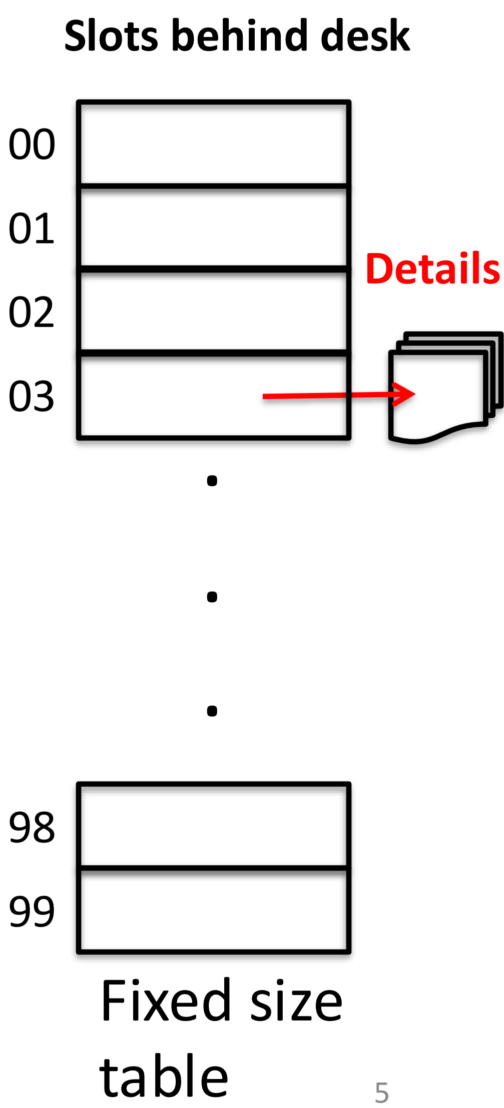
- Used to have 100 slots behind order desk, 0…99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone number (e.g., 03)



The old Sears catalog orders illustrate how hashing works

Sears store implementation of hash table

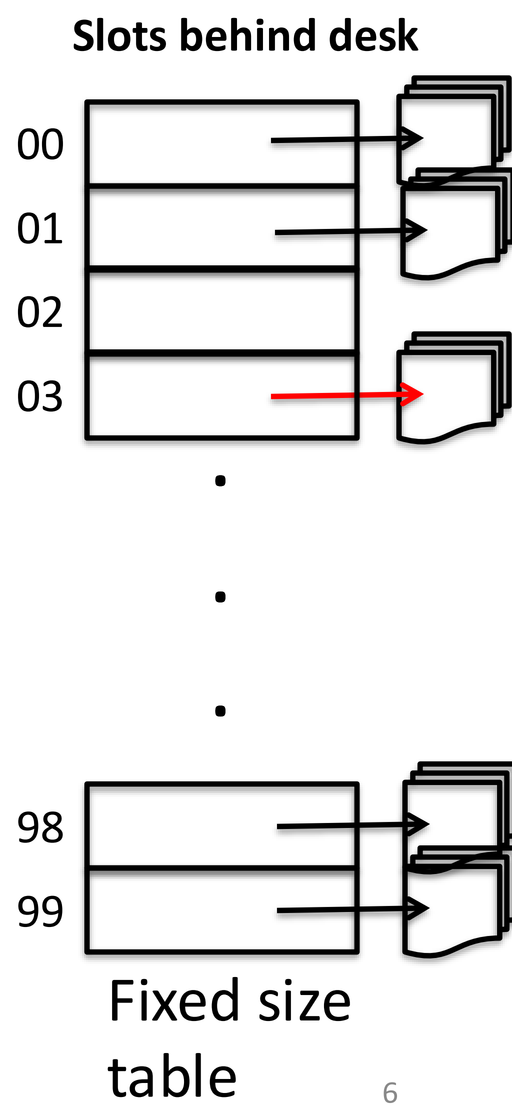
- Used to have 100 slots behind order desk, 0…99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone number (e.g., 03)



The old Sears catalog orders illustrate how hashing works

Sears store implementation of hash table

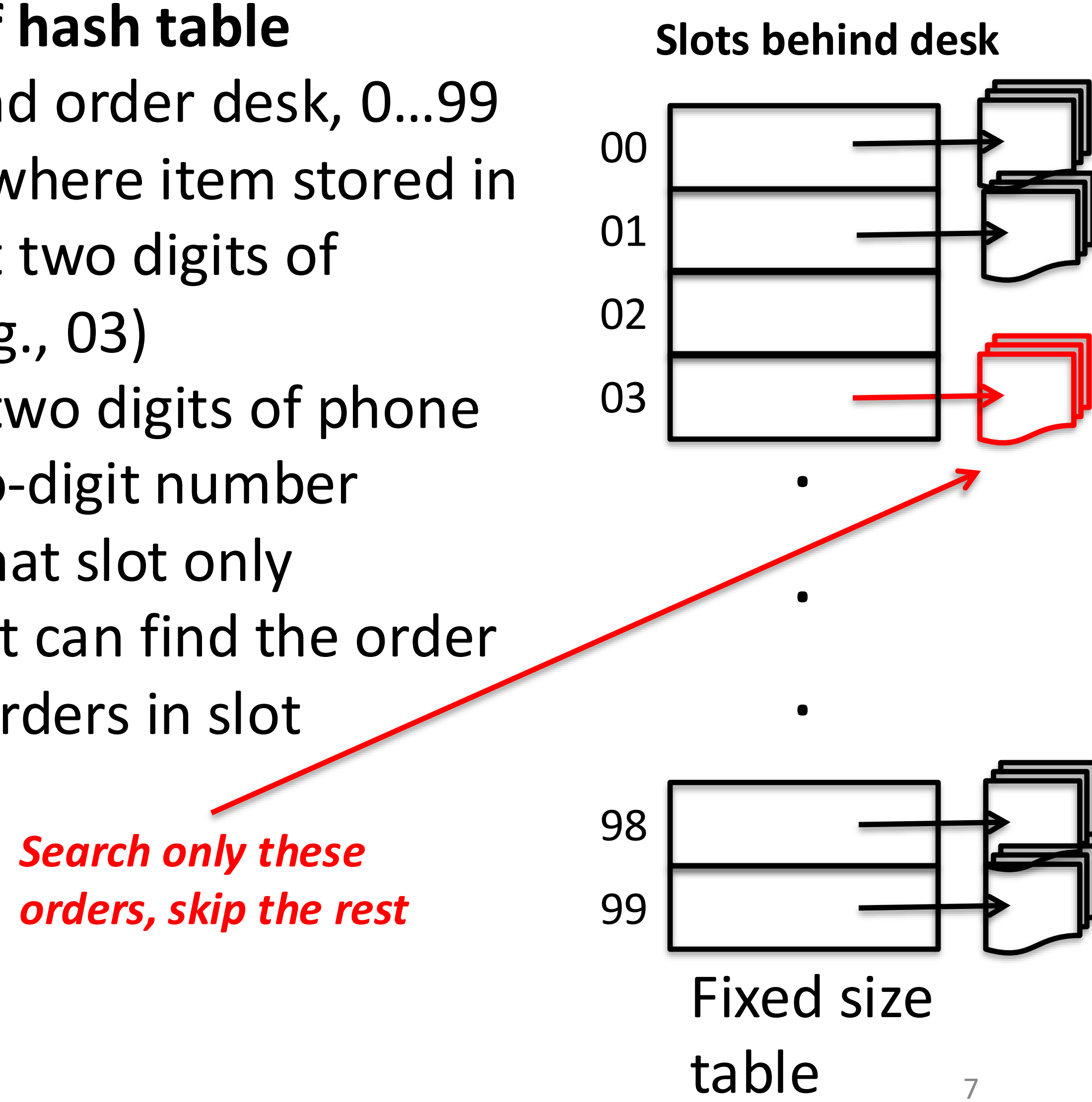
- Used to have 100 slots behind order desk, 0…99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone number (e.g., 03)
- Customer arrives, gives last two digits of phone



The old Sears catalog orders illustrate how hashing works

Sears store implementation of hash table

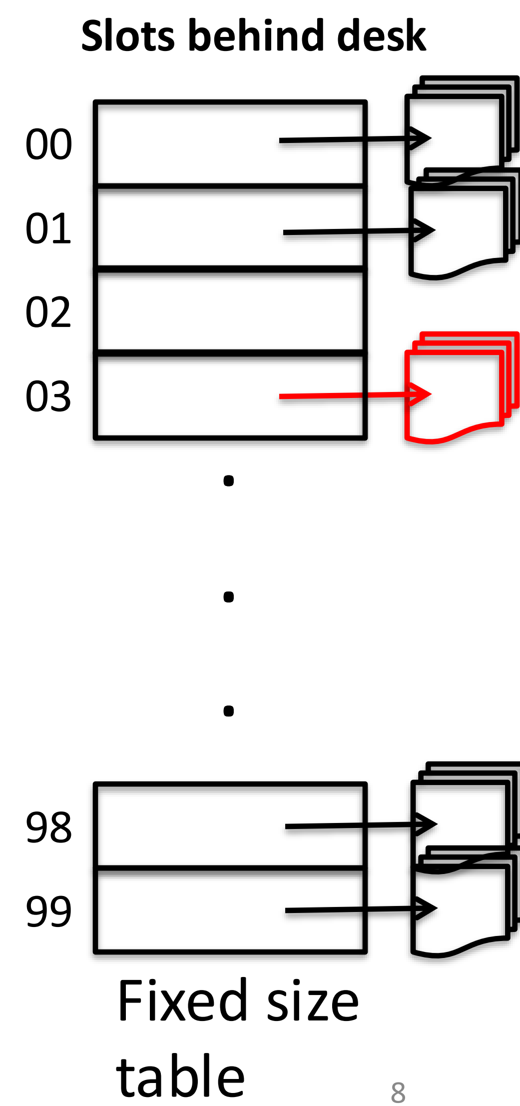
- Used to have 100 slots behind order desk, 0…99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone number (e.g., 03)
- Customer arrives, gives last two digits of phone
- Clerk finds slot with that two-digit number
- Clerk searches contents of that slot only
- Could be multiple orders, but can find the order quickly because only a few orders in slot



The old Sears catalog orders illustrate how hashing works

Sears store implementation of hash table

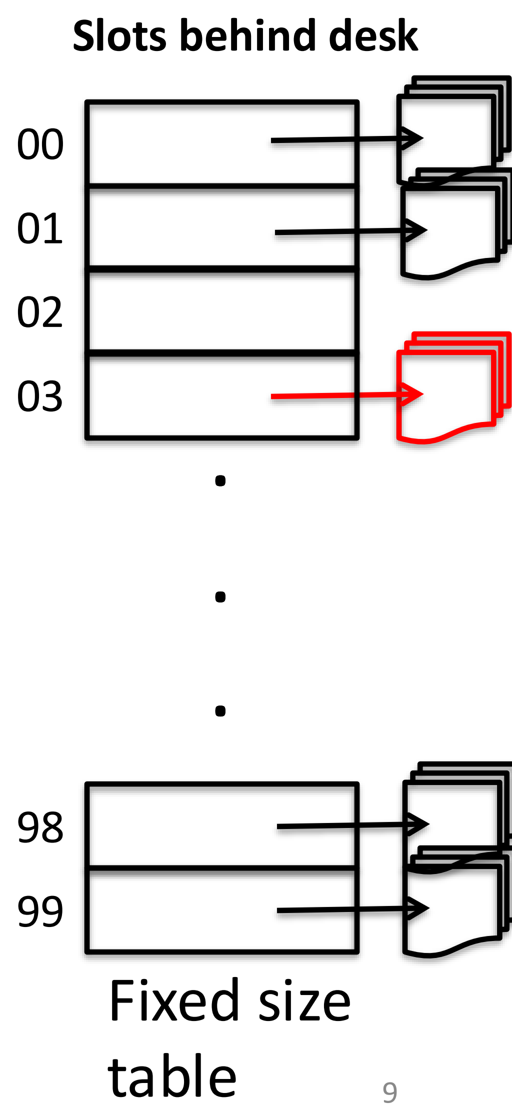
- Used to have 100 slots behind order desk, 0…99
- Shipments arrive, details of where item stored in warehouse put in slot by last two digits of customer phone number (e.g., 03)
- Customer arrives, gives last two digits of phone
- Clerk finds slot with that two-digit number
- Clerk searches contents of that slot only
- Could be multiple orders, but can find the order quickly because only a few orders in slot
- Splits set of (possibly) hundreds or thousands of orders into 100 slots of a few items each



The old Sears catalog orders illustrate how hashing works

Sears store implementation of hash table

- Used to have 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 number (e.g., 03)
- Customer arrives, gives last two digits of phone
- Clerk finds slot with that two-digit number
- Clerk searches contents of that slot only
- Could be multiple orders, but can find the order quickly because only a few orders in slot
- Splits 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?

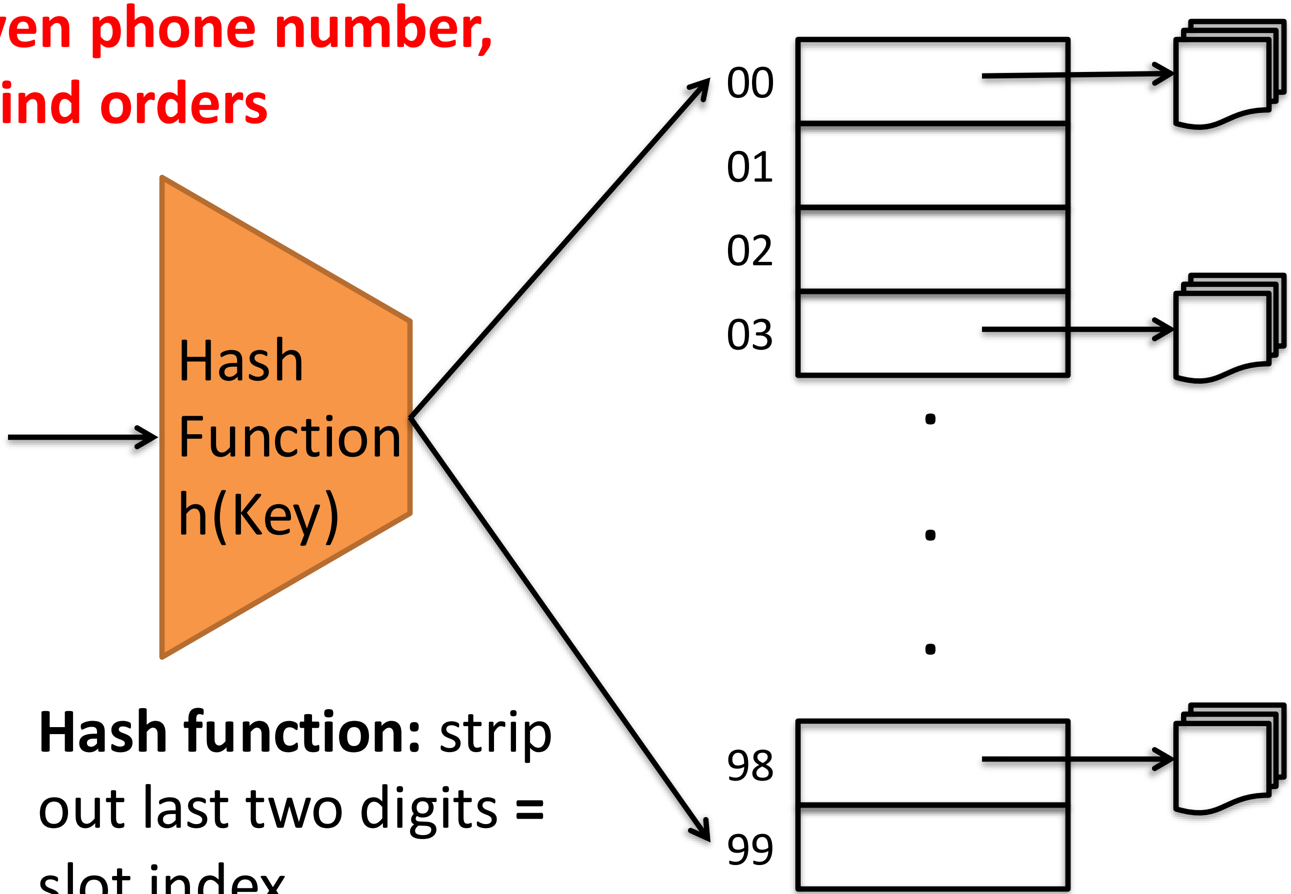


The store is using a form of hashing based on customer's phone number

Hashing phone numbers to find orders

Goal: given phone number, quickly find orders

Input: Phone number (Key)



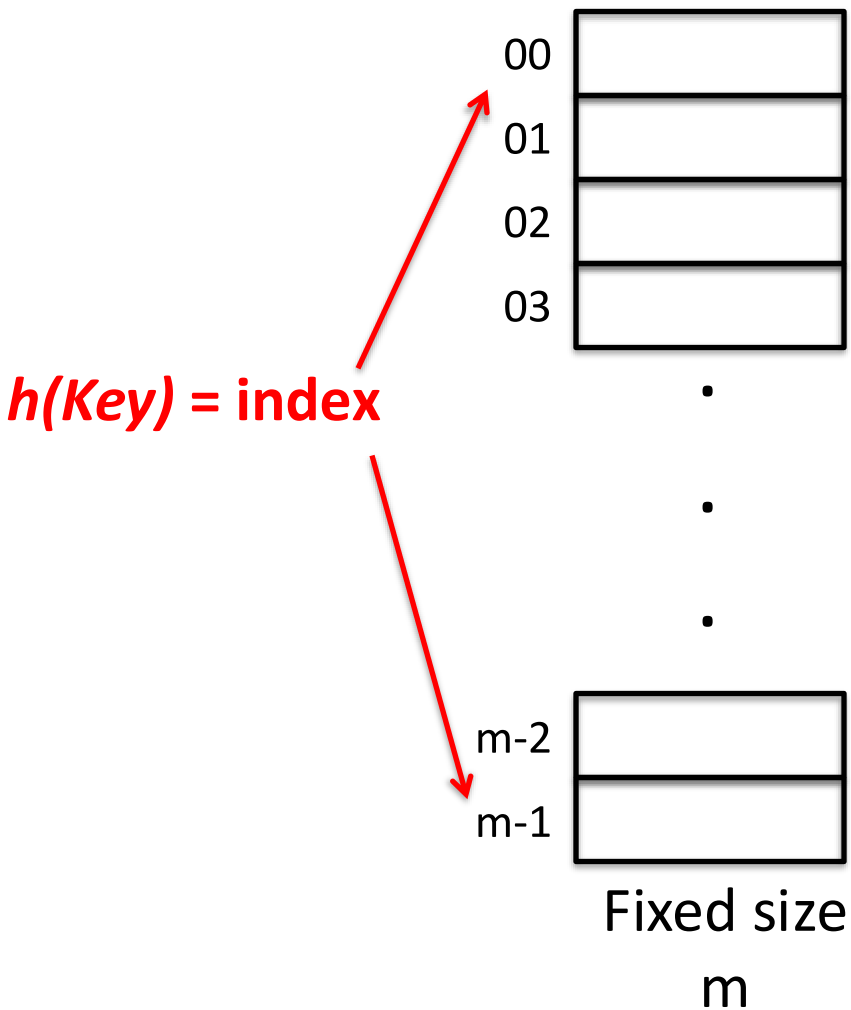
Search only small number of orders

Hash function: strip out last two digits = slot index

Fixed size table

Customer orders

Hashing's big idea: map a Key to an array index, then access is fast



Map hash table implementation

- Begin with array of fixed size *m* (called a hash table)
- Each array index holds item we want to find (e.g., warehouse location of customer's order)
- Use hash function *h* on *Key* to give index into hash table
- ***h(Key)* = table index *i = 0..m-1***
- Get item from hash table at index given by hash function
- <u>*Fast*</u> to *get/set/add/remove* items
- What about a HashSet?
- Use object itself as Key
- How to hash Key or object?

Agenda

1. Hashing
2. Computing Hash functions
3. Implementing Maps/Sets with hashing
4. Handling collisions
    1. Chaining
    2. Open Addressing

Key points:

1. **Hash function: fast and consistent, spread keys over table (simple uniform hashing), small key changes make different hash values**
2. **Hashing process: (1) convert to key integer, (2) constrain key to fall on table index**
3. **hashCode method returns integer representation of key**

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

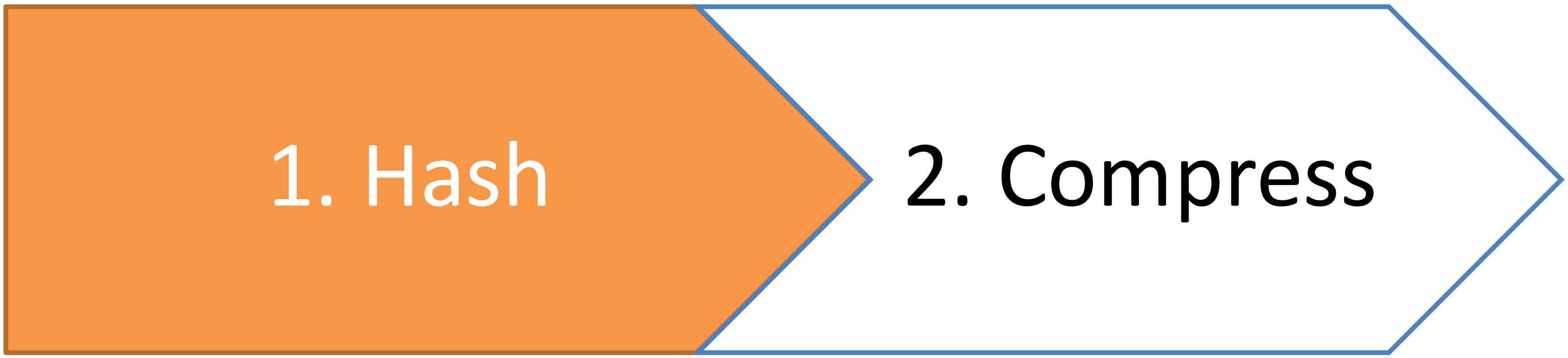
Desirable properties of a hash function

1. Hash can be computed quickly and consistently
2. Hash spreads the universe of keys evenly over the table (simple uniform hashing)
3. Small changes in the key (e.g., changing a character in a string or order of letters) should result in different hash value

Cryptographic hash function also:

- Difficult to determine key given the result of hash
- Unlikely that different keys will result in same hash
- We will not focus on crypto requirements

Hashing is often done in two steps: hash then compress



- Get an integer representation of Key
- Integer could be in range –infinity to +infinity

Constrain integer to table index [0..m)

First step in hashing is to get an integer representation of the key

Goal: given key compute an index into hash table array

Some Java objects can be directly cast to integers

- byte
- short
- int
- char

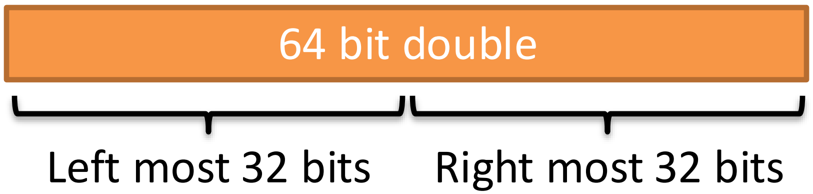
```
char a = 'a';
int b = (int)a;
```

b = 97

Some items too long cast to integers

- double (64 bits)
- long (64 bits)

- Too long to make 32-bit integers



XOR each half

Complex objects such as Strings can also be hashed to a single integer

Hashing complex objects

- Consider String `x` of length `n` where $x = x_0 x_1 \ldots x_{n-2} x_{n-1}$
- Pick prime number `a` (book recommends 31, 37, or 41)
- Cast each character in `x` to an integer
- Calculate polynomial hashcode as $a^{n-1}x_0 + a^{n-2}x_1 + \ldots ax_{n-2} + x_{n-1}$
- Use Horner's rule to efficiently compute hash code

```
public int hashCode() {
    final int a=37;
    int sum = x[0]; //first item in array
    for (int j=1;j<n;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

Good news: Java provides a *hashCode()* method to compute hashes for us!

hashCode()
Java does the hashing for us for Strings and autoboxed types with *hashCode()* method

Character a = ‘a’;
a.hashCode() returns 97

String b = “Hello”;
b.hashCode() returns 69609650

Bad news: We need to override *hashCode()* and *equals()* for our own Objects

- By default, Java uses memory address of objects as a *hashCode*
- But we typically want to hash based on properties of object, not whatever memory location an object happened to be assigned
- This way two objects with same instance variables will hash to the same table location (those objects are considered equal)
- Java says that two *equal* objects must return same *hashCode()*

```
public class PointHash extends Point {
   int r;

   public PointHash(int x, int y, int r) {
      super(x,y);
      this.r = r;
   }

   @Override
   public boolean equals(PointHash p) { //equal if same x,y,r
      return (x == p.x && y == p.y && r == p.r);
   }

   @Override
   public int hashCode() {
      final int a=37;
      int sum = a * a * x;
      sum += a * y;
      sum += r;
      return sum;
   }
```

Extend Point class to have a radius like PS-2

Here we consider two Points *equal* if they have the same *x, y* and *r* values
<u>*equals()* IS THE RIGHT WAY TO COMPARE OBJECT EQUALITY</u> (not ==)

Override *hashCode()* to provide the same hash if two Points are *equal*

If don't override *hashCode()* then even though two objects are considered equal, Java will look in the wrong slot

Java *hashCode()* example

```
public static void main(String[] args) {
  char a = 'a';
  int b = (int)a;
  System.out.println("Casting 'a' to int is: "+ b);
```

Casting 'a' to int is: 97

Some types can be directly cast to an integer

Java *hashCode()* example

```
public static void main(String[] args) {
   char a = 'a';
   int b = (int)a;
   System.out.println("Casting 'a' to int is: "+ b);
   Character z = 'a';
   System.out.println("hashCode for 'a' is: " + z.hashCode());
```

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
```

Java computes hash for autoboxed types with *hashCode()*

Java *hashCode()* example

```java
public static void main(String[] args) {
   char a = 'a';
   int b = (int)a;
   System.out.println("Casting 'a' to int is: "+ b);
   Character z = 'a';
   System.out.println("hashCode for 'a' is: " + z.hashCode());
   String y = "Hello";
   System.out.println("hashCode for 'hello' is: " + y.hashCode());
```

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
hashCode for 'hello' is: 69609650
```

***hashCode()* also works for more complex built-in types**

Java *hashCode()* example

```java
public static void main(String[] args) {
   char a = 'a';
   int b = (int)a;
   System.out.println("Casting 'a' to int is: "+ b);
   Character z = 'a';
   System.out.println("hashCode for 'a' is: " + z.hashCode());
   String y = "Hello";
   System.out.println("hashCode for 'hello' is: " + y.hashCode());
   System.out.println();

   //create new Point with overridden equals and hashCode functions
   PointHash b1 = new PointHash(5, 5, 5);
   PointHash b2 = new PointHash(0, 0, 5); //create new HashPoint
   System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
   System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
   System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
```

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
hashCode for 'hello' is: 69609650

b1 is at (x,y,r): 5, 5, 5
b2 is at (x,y,r): 0, 0, 5
hashCode b1: 7035 b2:5
```

For our own objects, we can provide our own *hashCode()* otherwise we get the memory location by default

Java *hashCode()* example

```
public static void main(String[] args) {
   char a = 'a';
   int b = (int)a;
   System.out.println("Casting 'a' to int is: "+ b);
   Character z = 'a';
   System.out.println("hashCode for 'a' is: " + z.hashCode());
   String y = "Hello";
   System.out.println("hashCode for 'hello' is: " + y.hashCode());
   System.out.println();

   //create new Point with overridden equals and hashCode functions
   PointHash b1 = new PointHash(5, 5, 5);
   PointHash b2 = new PointHash(0, 0, 5); //create new HashPoint
   System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
   System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
   System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
```

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
hashCode for 'hello' is: 69609650

b1 is at (x,y,r): 5, 5, 5
b2 is at (x,y,r): 0, 0, 5
hashCode b1: 7035 b2:5
```

```
@Override
public int hashCode() {
   final int a=37;
   int sum = a * a * x;
   sum += a * y;
   sum += r;
   return sum;
}
```

For our own objects, we can provide our own *hashCode()* otherwise we get the memory location by default

Java *hashCode()* example

```java
public static void main(String[] args) {
   char a = 'a';
   int b = (int)a;
   System.out.println("Casting 'a' to int is: "+ b);
   Character z = 'a';
   System.out.println("hashCode for 'a' is: " + z.hashCode());
   String y = "Hello";
   System.out.println("hashCode for 'hello' is: " + y.hashCode());
   System.out.println();

   //create new Blob with overridden equals
   PointHash b1 = new PointHash(5,5, 5);
   PointHash b2 = new PointHash(0, 0, 5); //
   System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
   System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
   System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
   System.out.println("b1 is equal to b2: " + b1.equals(b2));
```

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
hashCode for 'hello' is: 69609650

b1 is at (x,y,r): 5, 5, 5
b2 is at (x,y,r): 0, 0, 5
hashCode b1: 7035 b2:5
b1 is equal to b2: false
```

```java
@Override
public boolean equals(PointHash p) {
   return (x == p.x && y == p.y && r == p.r);
}
```

Override *equals()* to test if objects are equivalent

Otherwise *equals()* checks if same memory location

Java *hashCode()* example

```java
public static void main(String[] args) {
  char a = 'a';
  int b = (int)a;
  System.out.println("Casting 'a' to int is: "+ b);
  Character z = 'a';
  System.out.println("hashCode for 'a' is: " + z.hashCode());
  String y = "Hello";
  System.out.println("hashCode for 'hello' is: " + y.hashCode());
  System.out.println();

  //create new Blob with overridden equals
  PointHash b1 = new PointHash(5,5, 5);
  PointHash b2 = new PointHash(0, 0, 5); //
  System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
  System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
  System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
  System.out.println("b1 is equal to b2: " + b1.equals(b2));
```

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
hashCode for 'hello' is: 69609650

b1 is at (x,y,r): 5, 5, 5
b2 is at (x,y,r): 0, 0, 5
hashCode b1: 7035 b2:5
b1 is equal to b2: false
```

```java
@Override
public boolean equals(PointHash p) {
  return (x == p.x && y == p.y && r == p.r);
}
```

Override *equals()* to test if objects are equivalent
Otherwise *equals()* checks if same memory location

This is the right way to compare if two objects are equivalent (not b1 == b2)

Java *hashCode()* example

```
public static void main(String[] args) {
   char a = 'a';
   int b = (int)a;
   System.out.println("Casting 'a' to int is: "+ b);
   Character z = 'a';
   System.out.println("hashCode for 'a' is: " + z.hashCode());
   String y = "Hello";
   System.out.println("hashCode for 'hello' is: " + y.hashCode());
   System.out.println();

   //create new Blob with overridden equals and hashCode functions
   PointHash b1 = new PointHash(5,5, 5);
   PointHash b2 = new PointHash(0, 0, 5); //create new HashBlob
   System.out.println("b1 is at (x,y,r): " + b1.x + ", " + b1.y + ", " + b1.r);
   System.out.println("b2 is at (x,y,r): " + b2.x + ", " + b2.y + ", " + b2.r);
   System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
   System.out.println("b1 is equal to b2: " + b1.equals(b2));
   b2.x = 5; b2.y = 5; b2.r = 5; //set b2 to same location as b1
   System.out.println("after update b1 equals b2: " + b1.equals(b2));
   System.out.println("hashCode b1: " + b1.hashCode() + " b2:" + b2.hashCode());
}
```

```
Casting 'a' to int is: 97
hashCode for 'a' is: 97
hashCode for 'hello' is: 69609650

b1 is at (x,y,r): 5, 5, 5
b2 is at (x,y,r): 0, 0, 5
hashCode b1: 7035 b2:5
b1 is equal to b2: false
after update b1 equals b2: true
hashCode b1: 7035 b2:7035
```

After updating *x,y,* and *r* two Blobs are now equal

***hashCode()* returns same value for equivalent objects**

HashMap and HashSet will now put equivalent objects in the same slot

Hashing is often done in two steps: hash then compress

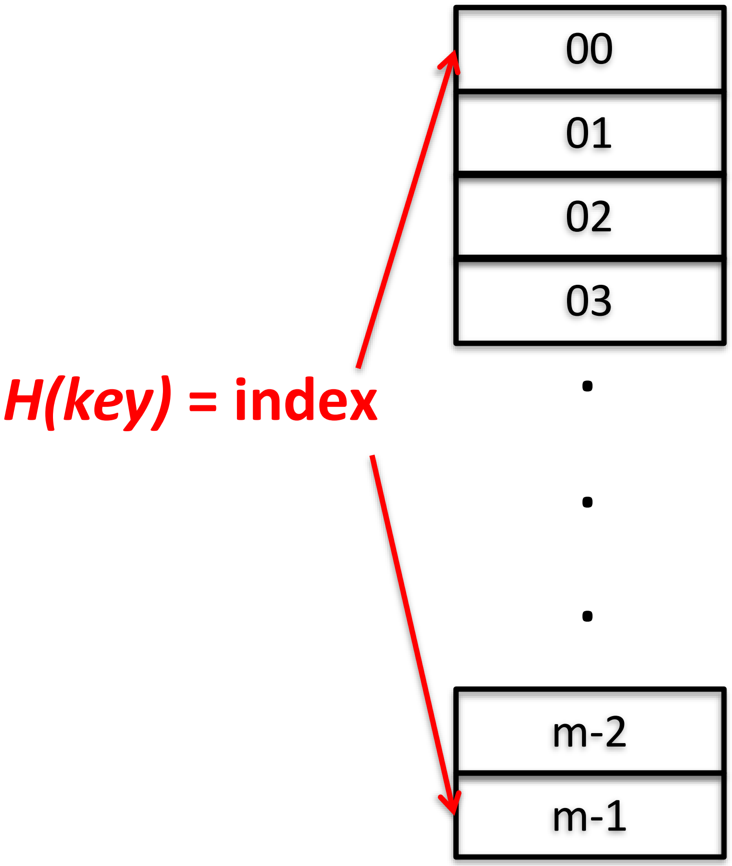
1. Hash

- Get an integer representation of Key
- Integer could be in range –infinity to +infinity

2. Compress

Constrain integer to table index [0..m)

May have to compress hash value to table index [0..m)



Fixed size *m* hash table

Compressing

- *hashCode()* value may be larger than the table (or negative!)
- Need to constrain value to one of the table slots [0..*m*)
- “Division method” is simple:
  h(key) = key.hashCode() % m
- Works well if *m* is prime
- Book gives a more advanced version called Multiply-Add-And-Divide (MAD)
- Java takes care of this for us ☺
- Eventually will encounter collisions where multiple keys map to the same slot ☹

Agenda

1. Hashing
2. Computing Hash functions
3. Implementing Maps/Sets with hashing
4. Handling collisions
    1. Chaining
    2. Open Addressing

Key points:

1. **Use hashCode to get integer representation of key**
2. **Constrain integer to fall on table index**
3. **Implement Map (or Set)**
    - **Put: store item at table index**
    - **Get: return value at table index**
    - **Remove: remove item at table index**

Map methods can be easily implemented with hashing

put(key, value)

- Hash key to get table index
  - Get i=key.hashCode()
  - Compress i to 0..m-1 with i% m
- Store key/value

get(key)

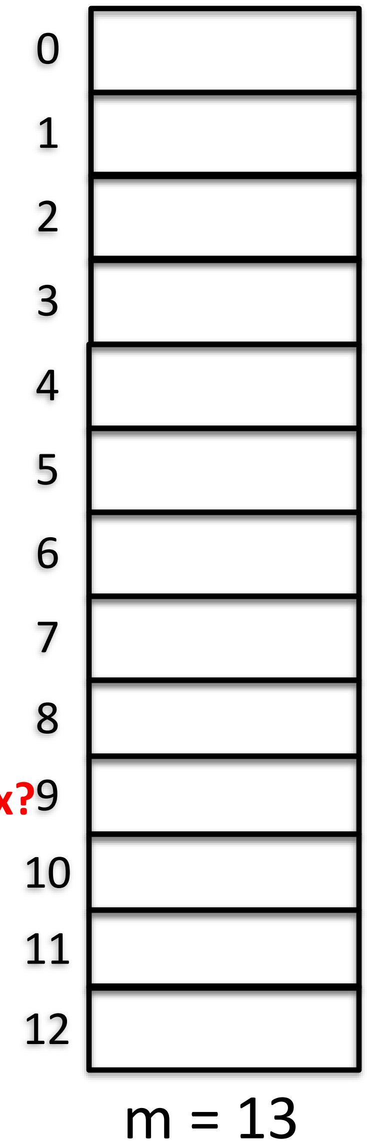
- Hash key to get table index
  - Get i=key.hashCode()
  - Compress i to 0..m-1 with i%m
- Return stored value

remove(key)

- Hash key to get table index
  - Get i=key.hashCode()
  - Compress i to 0..m-1 with i%m
- Remove stored key/value

Open questions:

- **What if multiple items hash to the same index?**
- **What if table fills up?**



Agenda

1. Hashing
2. Computing Hash functions
3. Implementing Maps/Sets with hashing
4. Handling collisions
   1. Chaining
   2. Open Addressing

Key points:

1. **Collisions result when different keys map to the same table index**
2. **Handle collisions in one of two ways:**
   1. **Chaining**
   2. **Open Addressing**
3. **Map/Set operations are constant time using hash table with low load factor and simple uniform hashing**

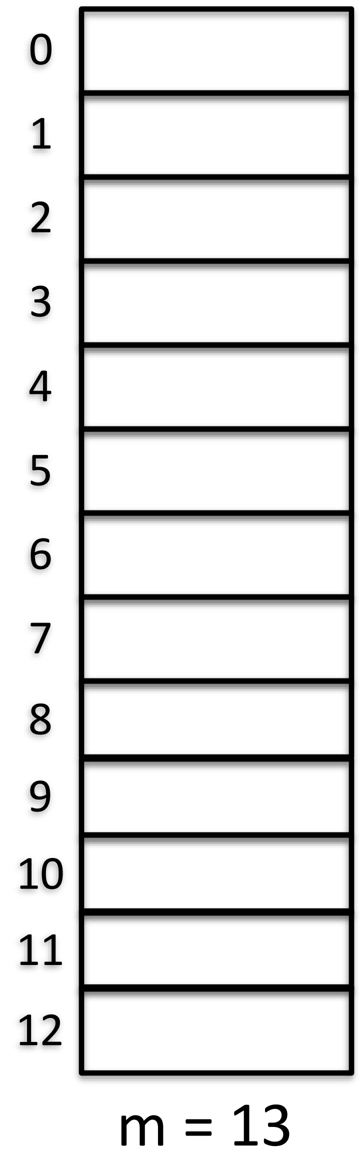
Collisions happen when multiple keys map to the same table index

Integer keys

Given table size m = 13
put(key,value)

- Hash & constrain key
- Store value at index

index = key.*hashCode()* % *m*



Collisions happen when multiple keys map to the same table index

Integer keys

Given table size m = 13

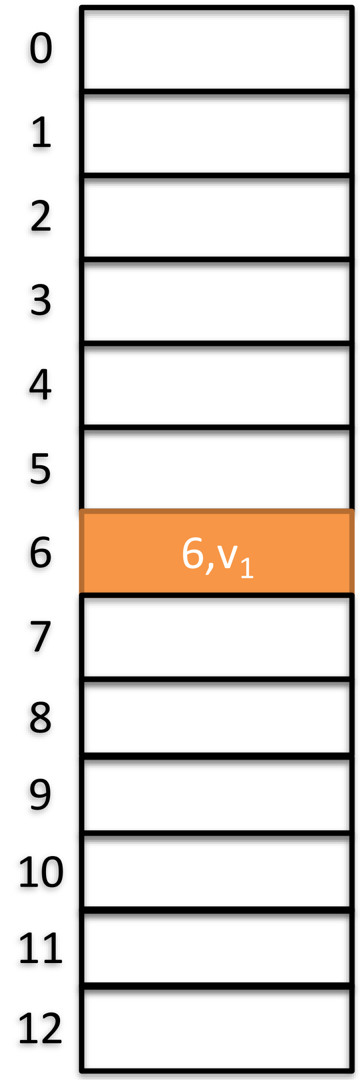
put(key,value)

- Hash & constrain key
- Store value at index

index = key.*hashCode() % m*

Example

- **put(6,v_1) = 6 % 13 = 6**



m = 13

Collisions happen when multiple keys map to the same table index

Integer keys

Given table size m = 13

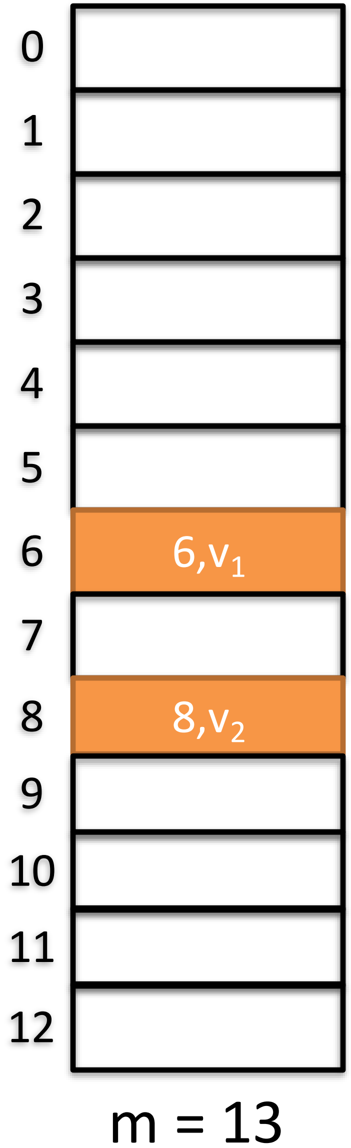
put(key,value)

- Hash & constrain key
- Store value at index

index = key.*hashCode() % m*

Example

- put(6,v_1) = 6 % 13 = 6
- **put(8,v_2) = 8 % 13 = 8**



Collisions happen when multiple keys map to the same table index

Integer keys

Given table size m = 13

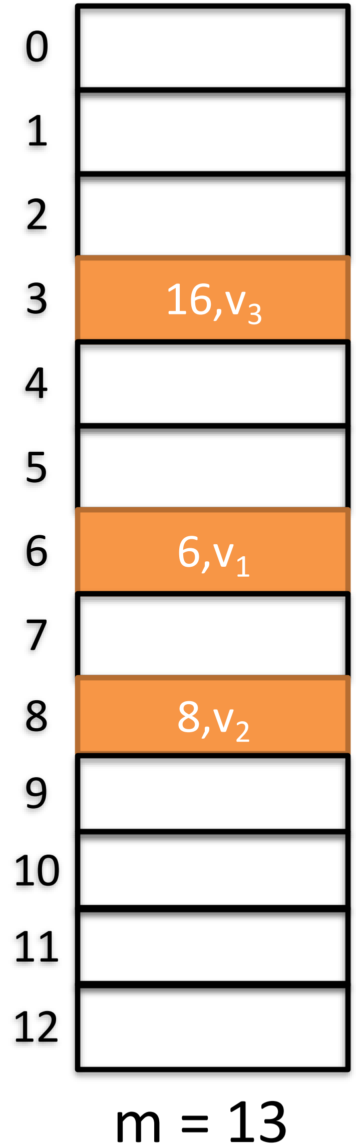
put(key,value)

- Hash & constrain key
- Store value at index

index = key.*hashCode() % m*

Example

- put(6,v_1) = 6 % 13 = 6
- put(8,v_2) = 8 % 13 = 8
- **put(16,v_3) = 16 % 13 = 3**



Collisions happen when multiple keys map to the same table index

Integer keys

Given table size m = 13

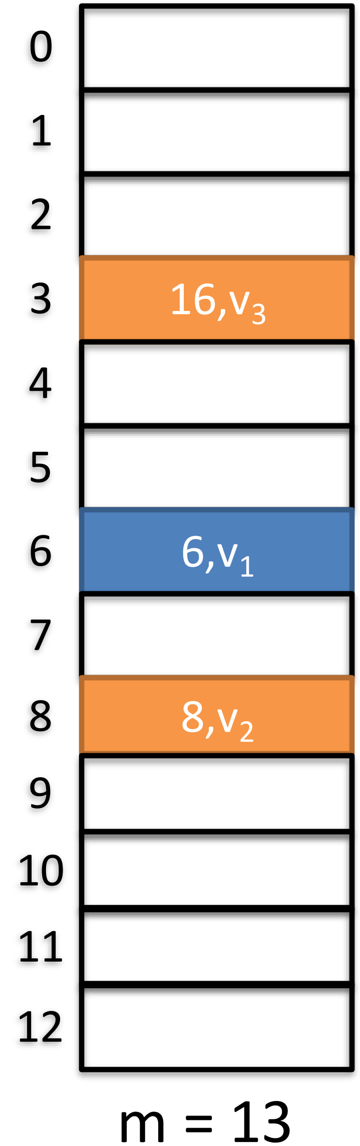
put(key,value)

- Hash & constrain key
- Store value at index

index = key.*hashCode() % m*

Example

- put(6,v_1) = 6 % 13 = 6
- put(8,v_2) = 8 % 13 = 8
- put(16,v_3) = 16 % 13 = 3
- **put(19,v_4) = 19 % 13 = 6**



Collision!

6 and 19 mapped to the same index

h(6)=h(19)

Agenda

1. Hashing

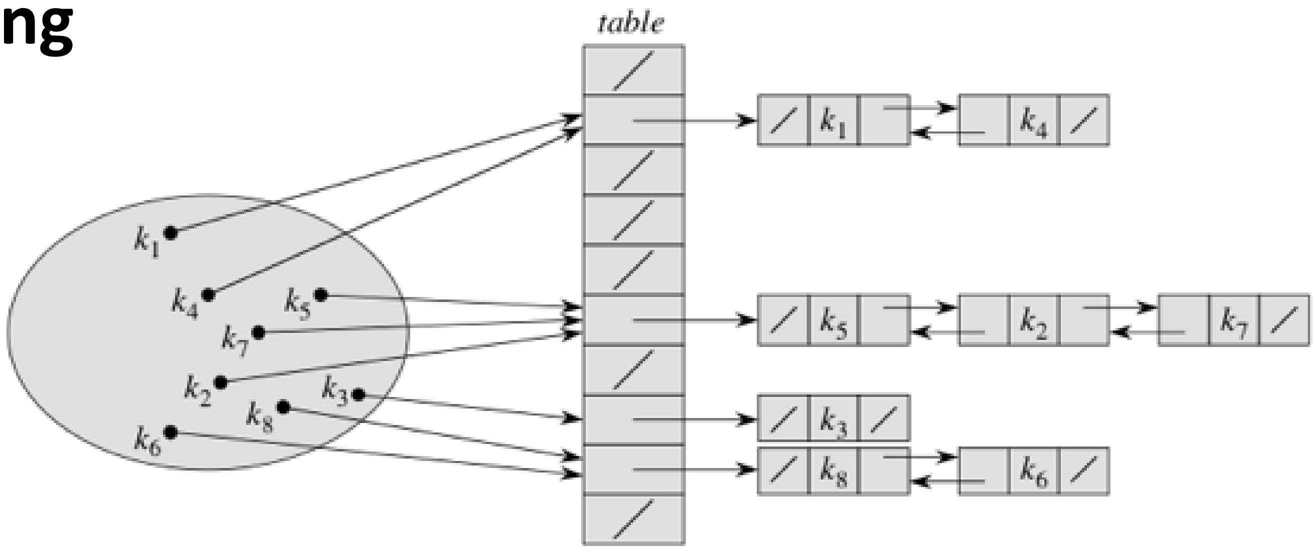
2. Computing Hash functions

3. Implementing Maps/Sets with hashing

1. Handling collisions
   1. Chaining
   2. Open Addressing

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

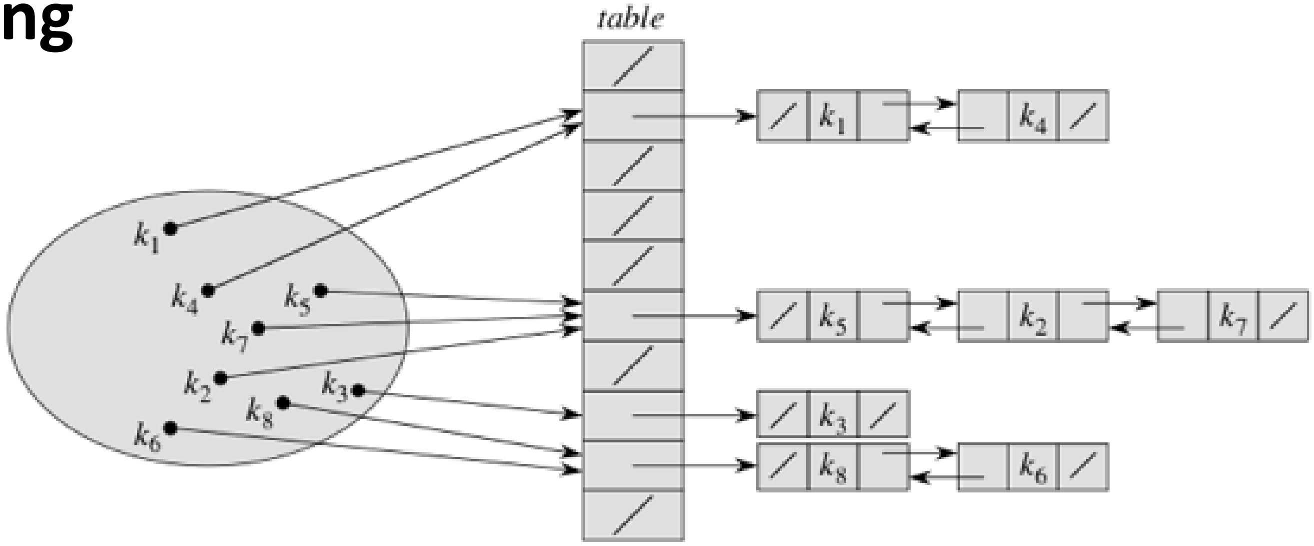
Chaining



- Create a table pointing to linked list of items that hash to the same index (similar to last class word positions)
- Slot *i* holds all keys *k* for which *h(k) = i*
- Splice in new elements at head
- NOTE: Values associated with Keys are not shown, here just showing Keys

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

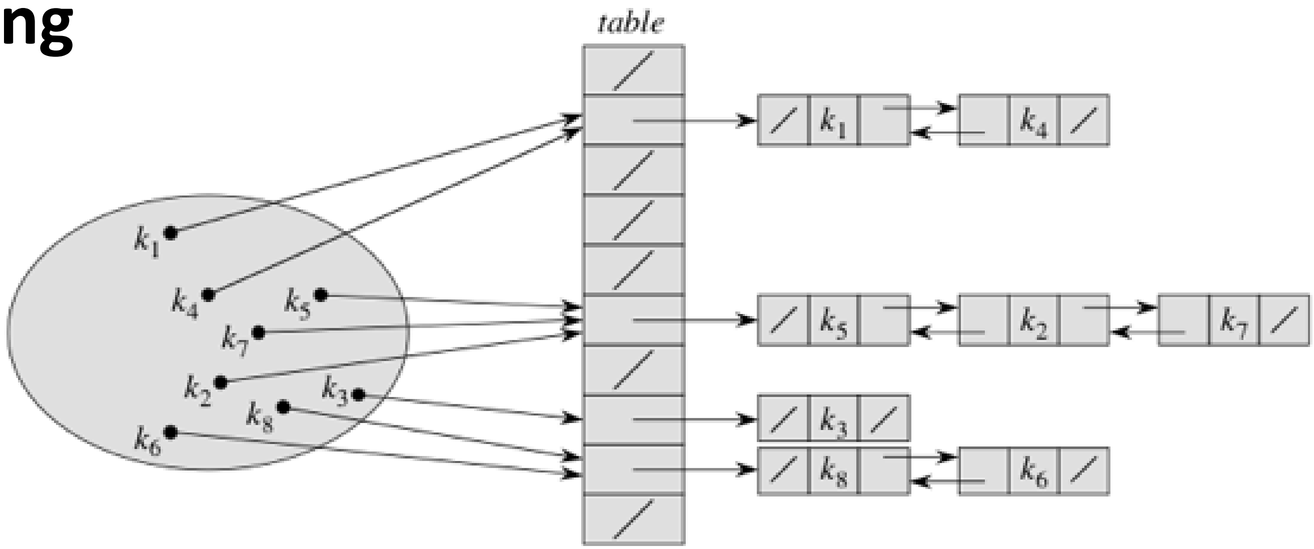
Chaining



- 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=n/m)*
- <u>*Expected*</u> search time is *Θ(1+λ)*, assuming ***simple uniform hashing*** (each possible key equally likely to hash into a particular slot), worst case *O(n)* if bad hash function

If 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 λ > 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. Implementing Maps/Sets with hashing

1. 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 λ ≈ 1, increase table size (*m*2*) and rehash
- Search analogous to insertion, compute key and probe until find item or empty slot (key not in table)

Linear probing is one way of handling collisions under open addressing

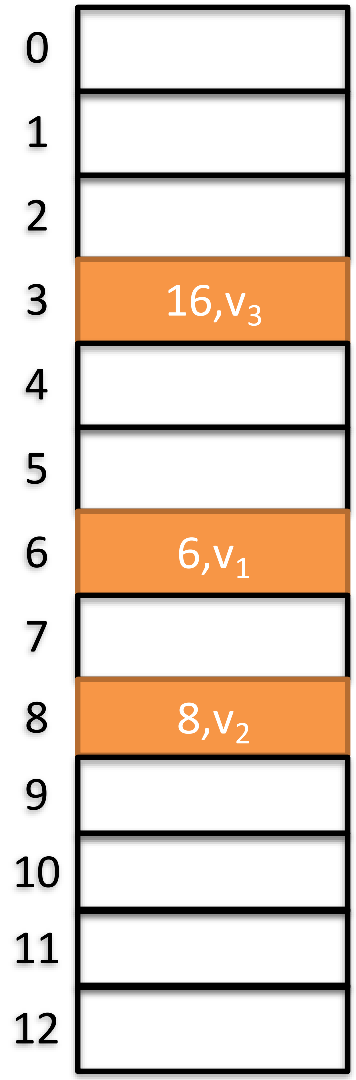
Integer keys

Given table size m = 13

index = key.*hashCode() % m*

Example

- put(6,v_1) = 6 % 13 = 6
- put(8,v_2) = 8 % 13 = 8
- put(16,v_3) = 16 % 13 = 3



m = 13

Linear probing is one way of handling collisions under open addressing

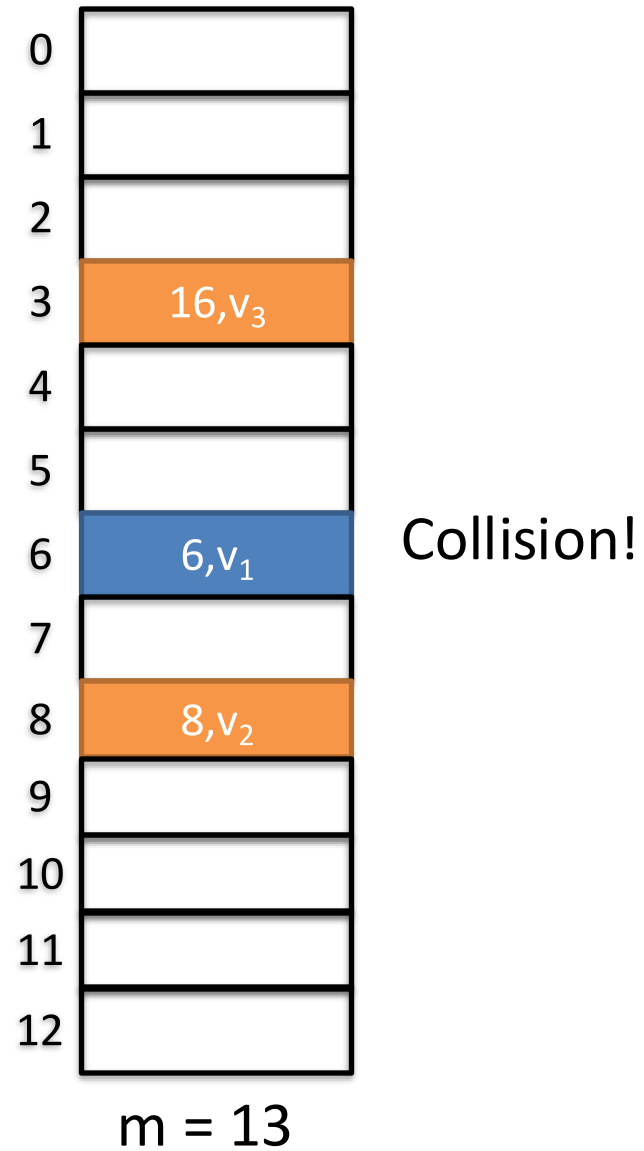
Integer keys

Given table size m = 13

index = key.*hashCode()* % *m*

Example

- put(6,v_1) = 6 % 13 = 6
- put(8,v_2) = 8 % 13 = 8
- put(16,v_3) = 16 % 13 = 3
- **put(19,v_4) = 19 % 13 = 6**



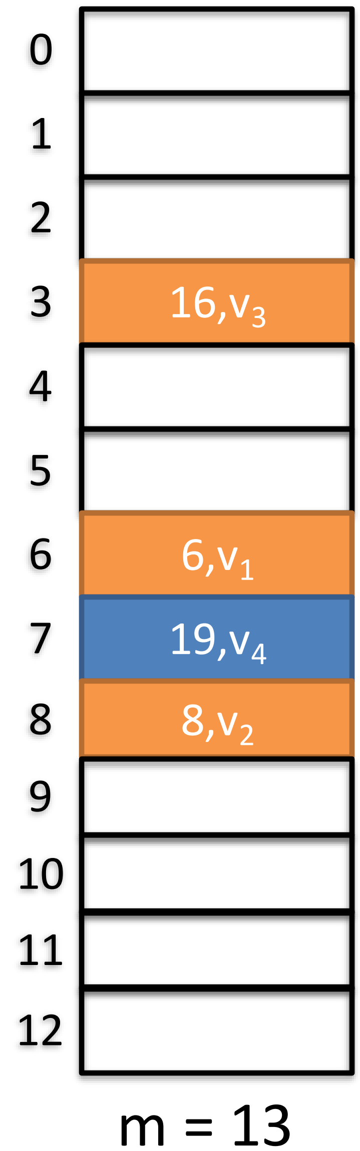
Try next index if hashed index is full, repeat if next index is also full

Integer keys

Given table size m = 13

index = key.*hashCode() % m*

Example
- put(6,v_1) = 6 % 13 = 6
- put(8,v_2) = 8 % 13 = 8
- put(16,v_3) = 16 % 13 = 3
- **put(19,v_4) = 19 % 13 = 6**



Insert at *i+1* = 7

To find items, probe until find Key or hit an empty space

Integer keys

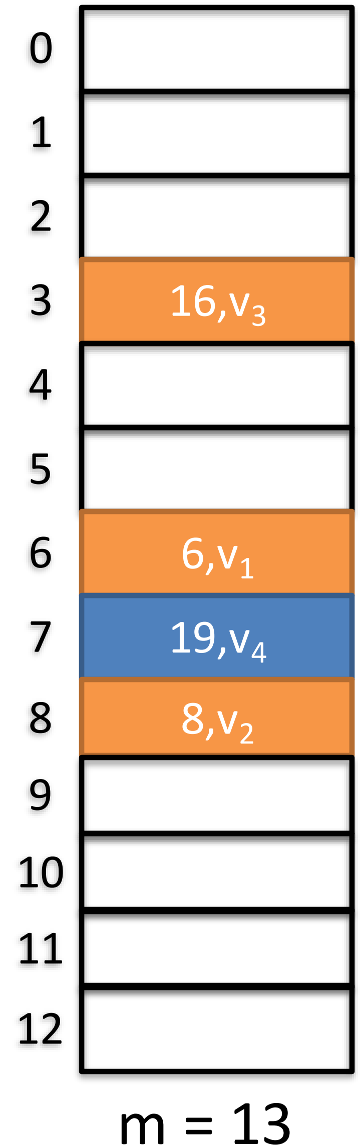
Given table size m = 13

index = key.*hashCode() % m*

Example

- put(6,v_1) = 6 % 13 = 6
- put(8,v_2) = 8 % 13 = 8
- put(16,v_3) = 16 % 13 = 3
- put(19,v_4) = 19 % 13 = 6

- **get(19)**



Insert at *i+1* = 7

To find items later, hash to table index, then probe until find item or hit empty slot

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 skips over marked spaces
- This can be a problem if many deletes create many marked slots, search approaches linear time

Clustering of keys can build 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 build up and reduce performance

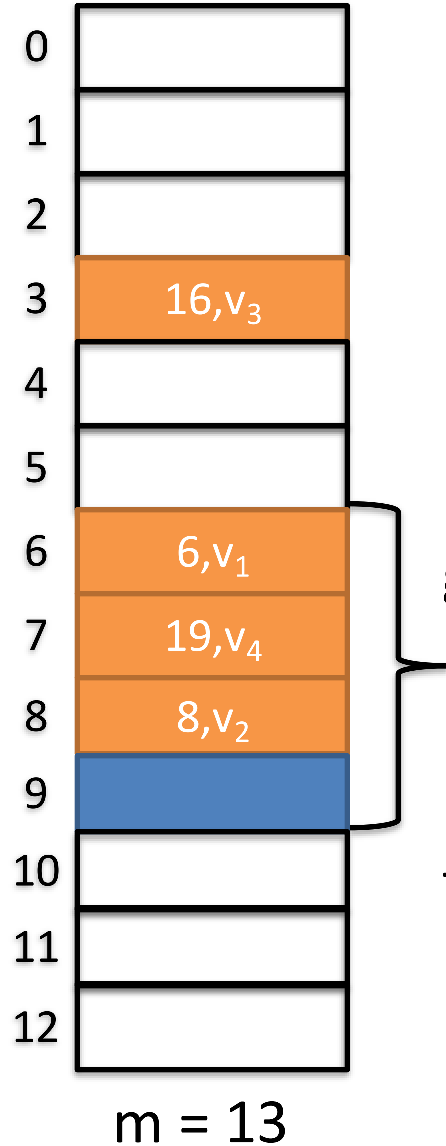
Integer keys

Given table size m = 13

index = key.*hashCode() % m*

Example

- put(6,v_1) = 6 % 13 = 6
- put(8,v_2) = 8 % 13 = 8
- put(16,v_3) = 16 % 13 = 3
- put(19,v_4) = 19 % 13 = 6



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

- **<u>Big idea</u>: instead of stepping by 1 at each collision like linear probing, step by a different amount where the step size depends on the key**
- 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 on each collision until space is found
- Result is a step by $h_2(k)$ on each collision (then mod m to stay inside table size), instead of 1
- Need to design hashes so that if $h_1(k_1)=h_1(k_2)$, then ***unlikely*** $h_2(k_1)=h_2(k_2)$

Double hashing can help with the clustering problem

put(6,v_1)

h_1 same as before
h_2 new hash function
p = probe number
(initially 0)

Given table size m = 13

Compute

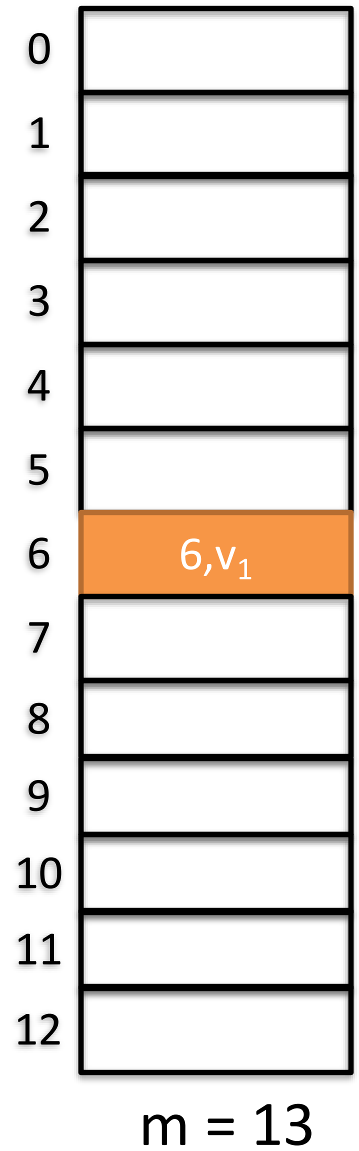
$h_1(key) = (key \% m)$

$h_2(key) = 1 + (key \% (m-1))$

$h'(k,p)=(h_1(k) + ph_2(k)) \% m$

Example

Key	**p**	**h_1**	**h_2**	**h’**
6	0	6	7	(6+0*7)%13 = 6



Double hashing can help with the clustering problem

put(8,v_2)

h_1 same as before
h_2 new hash function
p = probe number (initially 0)

Given table size m = 13

Compute

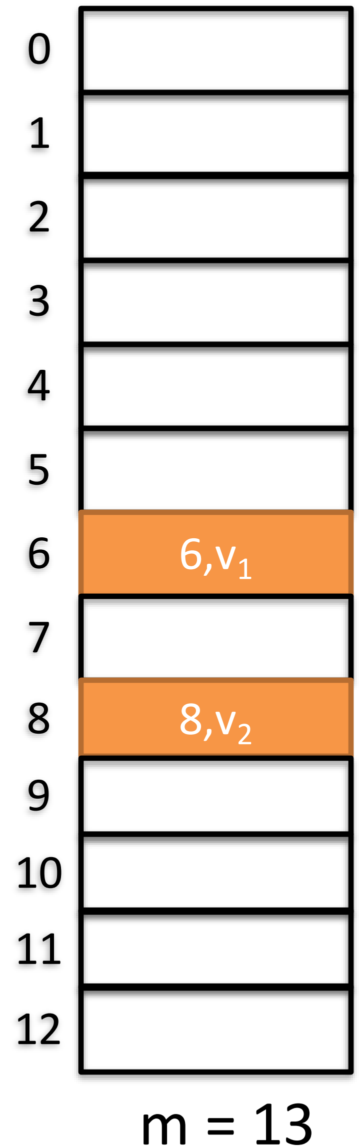
$h_1(key) = (key \% m)$

$h_2(key) = 1 + (key \% (m-1))$

$h'(k,p) = (h_1(k) + p h_2(k)) \% m$

Example

Key	p	h_1	h_2	h’
6	0	6	7	(6+0*7)%13 = 6
8	0	8	9	(8+0*9)%13 = 8



Double hashing can help with the clustering problem

put(16,v_3)

h_1 same as before
h_2 new hash function
p = probe number (initially 0)

Given table size m = 13

Compute

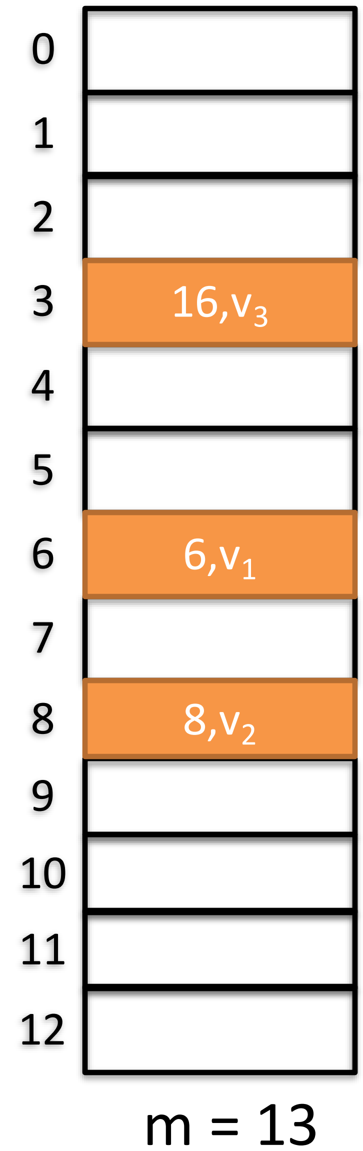
$h_1(key) = (key \% m)$

$h_2(key) = 1 + (key \% (m-1))$

$h'(k,p)=(h_1(k) + ph_2(k)) \% m$

Example

Key	p	h_1	h_2	h'
6	0	6	7	(6+0*7)%13 = 6
8	0	8	9	(8+0*9)%13 = 8
16	0	3	5	(3+0*5)%13 = 3



Double hashing can help with the clustering problem

put(19,v_4)

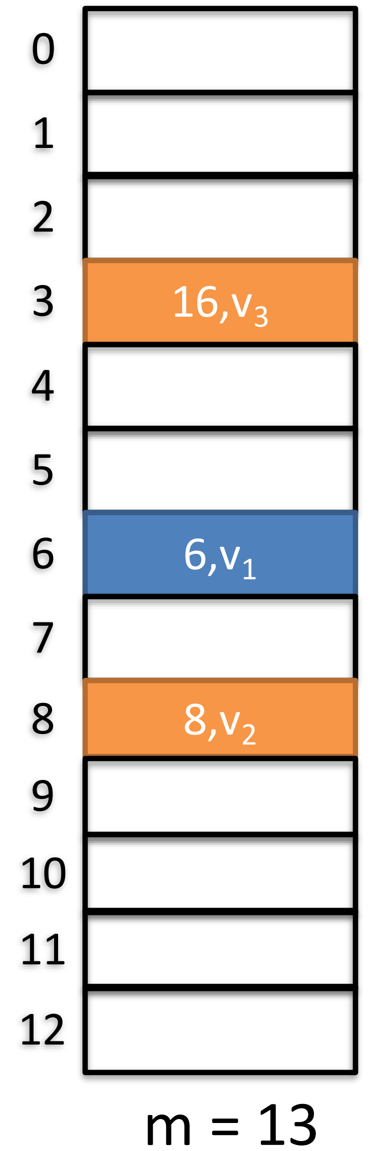
h_1 same as before
h_2 new hash function
p = probe number (initially 0)

Given table size m = 13
Compute

$$h_1(key) = (key \% m)$$
$$h_2(key) = 1 + (key \% (m-1))$$
$$h'(k,p) = (h_1(k) + p h_2(k)) \% m$$

Example

Key	p	h_1	h_2	h'
6	0	6	7	(6+0*7)%13 = **6**
8	0	8	9	(8+0*9)%13 = 8
16	0	3	5	(3+0*5)%13 = 3
19	0	6	8	(6+0*8)%13 = **6**



Collision!

Double hashing can help with the clustering problem

put(19,v_4)

h_1 same as before
h_2 new hash function
p = probe number (initially 0)

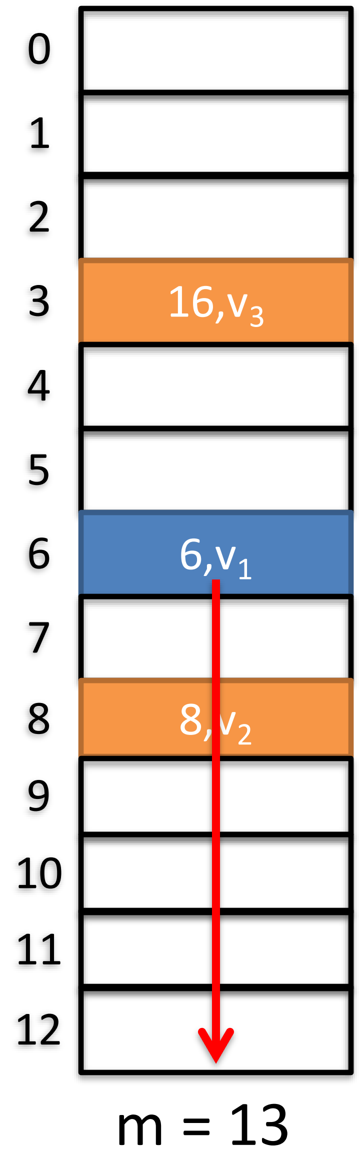
Given table size m = 13

Compute

$$h_1(key) = (key \% m)$$
$$h_2(key) = 1 + (key \% (m-1))$$
$$h'(k,p)=(h_1(k) + ph_2(k)) \% m$$

Example

Key	p	h_1	h_2	h’
6	0	6	7	(6+0*7)%13 = 6
8	0	8	9	(8+0*9)%13 = 8
16	0	3	5	(3+0*5)%13 = 3
19	0	6	8	(6+0*8)%13 = 6
19	**1**	6	**8**	(6+**1*8**)%13 = **1**



Collision!

Increment p

Step forward by h_2(key) = 8 spaces

Wrap around if needed

Double hashing can help with the clustering problem

put(19,v_4)

h_1 same as before
h_2 new hash function
p = probe number (initially 0)

Given table size m = 13

Compute

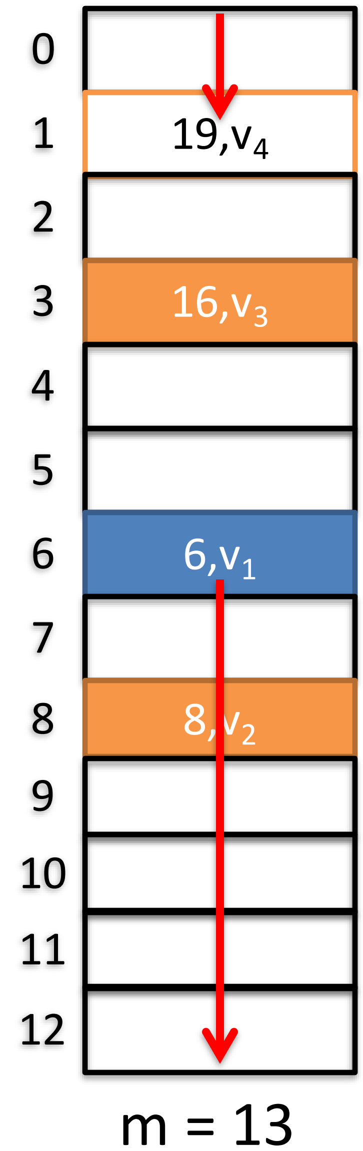
$h_1(key) = (key \% m)$

$h_2(key) = 1 + (key \% (m-1))$

$h'(k,p)=(h_1(k) + ph_2(k)) \% m$

Example

Key	**p**	**h_1**	**h_2**	**h'**
6	0	6	7	(6+0*7)%13 = 6
8	0	8	9	(8+0*9)%13 = 8
16	0	3	5	(3+0*5)%13 = 3
19	0	6	8	(6+0*8)%13 = 6
19	**1**	6	**8**	(6+**1*8**)%13 = **1**



Insert here

Collision!

Increment p

Step forward by h_2(key) = 8 spaces

Wrap around if needed

Run time degrades as λ gets large, so keep λ small by growing hash table

Expected insert and search time

- Average number of probes is approximately $1/(1-\lambda)$
- As $\lambda \to 1$, expected number of probes becomes large, when λ small, number of probes approaches 1
- If table 90% full, then expect about 10 probes for unsuccessful search
- Successful search generally a little faster, about 2.5 probes (math on course web page and in book)
- Must grow table and *rehash* when copying to new table to keep the table sparsely populated or performance suffers

Sparsely populated table trades memory for speed

Assuming load factor λ is small and hashing spreads keys, core operations are O(1)

Operation	Expected run time	Notes
hash(k)	O(1)	• Math operations on key *k* to hash and compress, outputs 0...m-1 • Constant time, does not depend on number of items in Set or Map

Assuming load factor λ is small and hashing spreads keys, core operations are O(1)

Operation	**Expected run time**	**Notes**
hash(k)	O(1)	• Math operations on key *k* to hash and compress, outputs 0...m-1 • Constant time, does not depend on number of items in Set or Map

find(k)	O(1)	• Once have index of table due to hash:
		  • Chaining: traverse linked list O(λ) = O(1)
		  • Probing: probe until find O(1/(1-λ)) = O(1)

Assuming load factor λ is small and hashing spreads keys, core operations are O(1)

Operation	Expected run time	Notes
hash(k)	O(1)	• Math operations on key *k* to hash and compress, outputs 0...m-1 • Constant time, does not depend on number of items in Set or Map

find(k)	O(1)	• Once have index of table due to hash:
		  • Chaining: traverse linked list $O(\lambda) = O(1)$
		  • Probing: probe until find $O(1/(1-\lambda)) = O(1)$

get(k)	O(1+1) = O(1)	• *Hash + find*: • chaining = $O(1+\lambda) = O(1)$, probing = $O(1+(1/(1-\lambda))) = O(1)$

Assuming load factor λ is small and hashing spreads keys, core operations are O(1)

Operation	Expected run time	Notes
hash(k)	O(1)	• Math operations on key *k* to hash and compress, outputs 0...m-1 • Constant time, does not depend on number of items in Set or Map

find(k)	O(1)	• Once have index of table due to hash:
		  • Chaining: traverse linked list O(λ) = O(1)
		  • Probing: probe until find O(1/(1-λ)) = O(1)

get(k)	O(1+1) = O(1)	• *Hash + find*: • chaining = O(1+λ) = O(1), probing = O(1+(1/(1-λ))) = O(1)

put(k,v)	O(1)
	+O(1)
	O(1)
		• *Hash + find* = O(1)
		• Plus update or add element:
		  • Chaining: update value or add at head O(1)
		  • Probing: store value in array O(1)

Assuming load factor λ is small and hashing spreads keys, core operations are O(1)

Operation	**<u>Expected</u> run time**	**Notes**
hash(k)	O(1)	• Math operations on key *k* to hash and compress, outputs 0...m-1 • Constant time, does not depend on number of items in Set or Map

find(k)	O(1)	• Once have index of table due to hash:
		  • Chaining: traverse linked list O(λ) = O(1)
		  • Probing: probe until find O(1/(1-λ)) = O(1)

get(k)	O(1+1) = O(1)	• *Hash + find*: • chaining = O(1+λ) = O(1), probing = O(1+(1/(1-λ))) = O(1)

put(k,v)	O(1)
	<u>+O(1)</u>
	O(1)
		• *Hash + find* = O(1)
		• Plus update or add element:
		  • Chaining: update value or add at head O(1)
		  • Probing: store value in array O(1)

remove(k)	O(1)
	<u>+O(1)</u>
	O(1)
		• *Hash + find* = O(1)
		• Plus remove element:
		  • Chaining: update one pointer O(1)
		  • Probing: mark space empty O(1)

Assuming a small load factor and uniform hashing, the core operations of HashSets and HashMaps are constant time!

Key points

1. Hashing maps a key to a table index
2. We can use this concept to implement Maps and Sets
3. Hash function: fast and consistent, spread keys over table (simple uniform hashing), small key changes make different hash values
4. Hashing process: (1) convert to key integer, (2) constrain key to fall on table index
5. hashCode method returns integer representation of key
6. Constrain hashCode integer to fall on table index (easy way: modulo table size)
7. Implement Map (or Set)
    - Put: store item at constrained table index
    - Get: return value at constrained table index
    - Remove: remove constrained item at table index
8. Collisions result when different keys map to the same table index
9. Handle collisions in one of two ways:
    - Chaining
    - Open Addressing
10. Map/Set operations are constant time using hash table assuming low load factor and simple uniform hashing