CS151 Intro to Data Structures

Hashmaps

Announcements

HW05 due Wednesday

HW06 due next Wednesday (11/22)
- Will be released tonight
- Lab08 due next Wednesday too

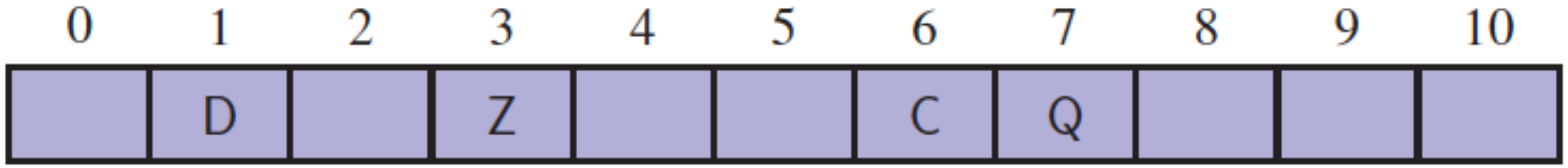
No lab next Wednesday

HW07 due 12/05
- Lab09 (today's lab) due then

Notion of a Map

Intuitively, a map `M` supports the abstraction of using keys as indices with a syntax such as `M[k]`.

Simplest setting is a map with n items using keys that are known to be integers from 0 to $N-1$, for some $N \geq n$.

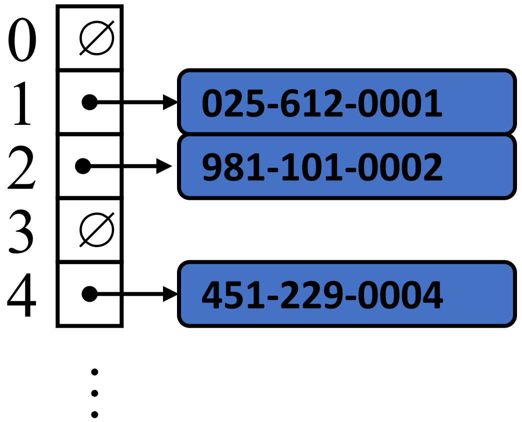


More General Keys

What if our keys are not integers in range 0 to $N-1$?

Use a hash function to map keys to integers into the right range

- Example: last 4 digits of SSN



Hash Functions and Tables

A hash function h maps a key to integers in a fixed interval $[0, N-1]$

$h(x) = x\%N$ is such a function for integers

$h(x)$ is the *hash value* of key x

A hash table is an array of size N

- associated hash function h
- item (k, v) is stored at index $h(k)$

Example

A hash table storing entries as (SSN, Name), where SSN is a nine-digit positive integer

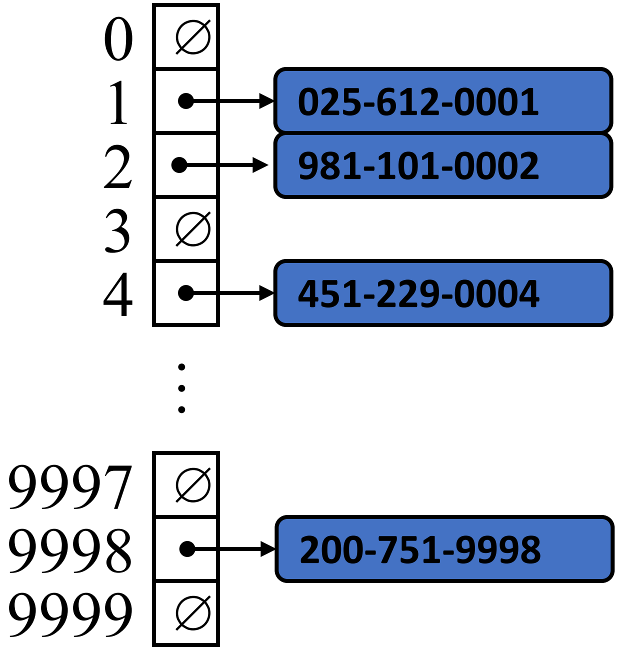
A hash table is an array of size N

- associated hash function h
- item (k, v) is stored at index $h(k)$

Example

A hash table storing entries as (SSN, Name), where SSN is a nine-digit positive integer

Use an array of size $N = 10000$ and the hash function $h(x) = \text{last 4 digits of } x$



Hash Function

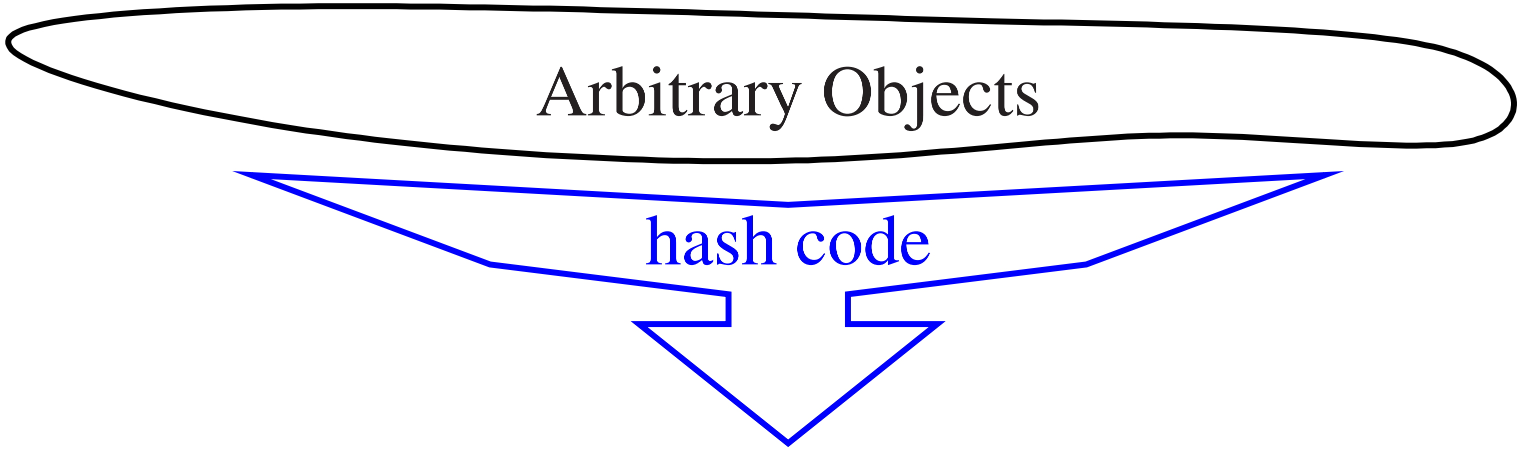
A hash function is usually specified as the composition of two functions:

- hash code: h_1: key → integers
- compression: h_2: integers → $[0, N-1]$
- $h(x) = h_2(h_1(x))$

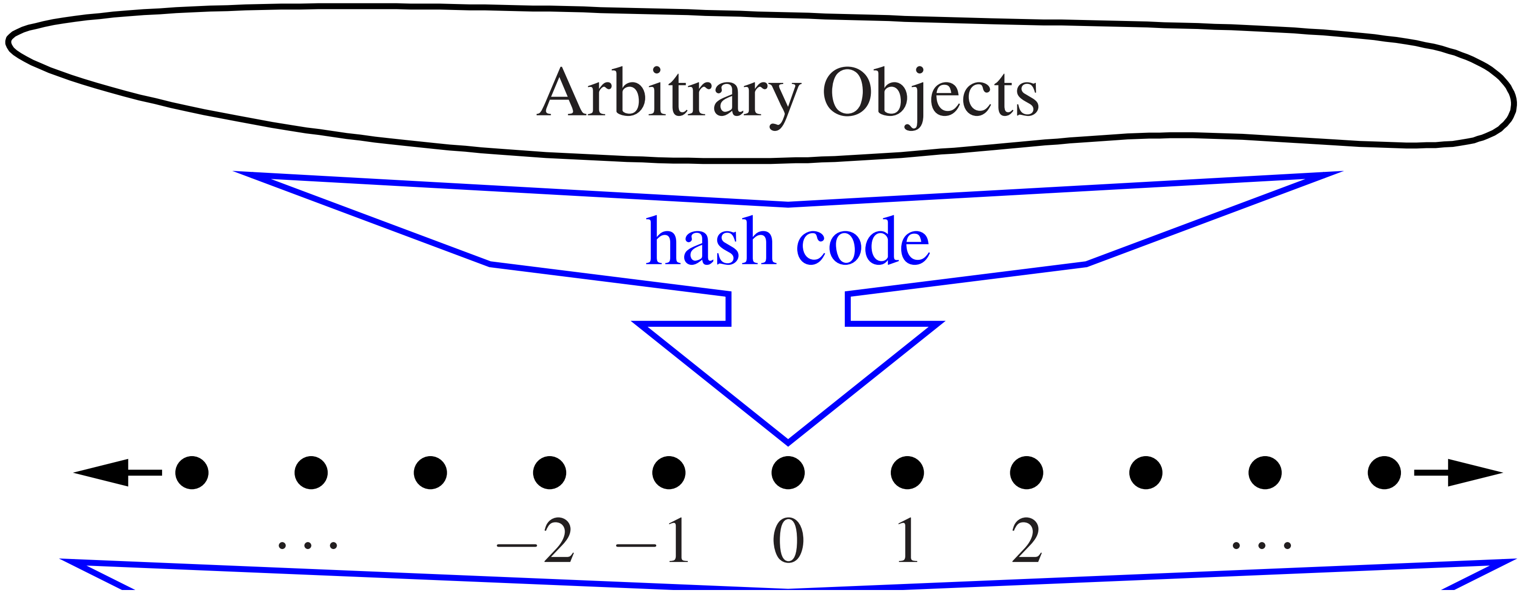
The goal is to "disperse" the keys in an appropriately random way

Hash Function Illustration

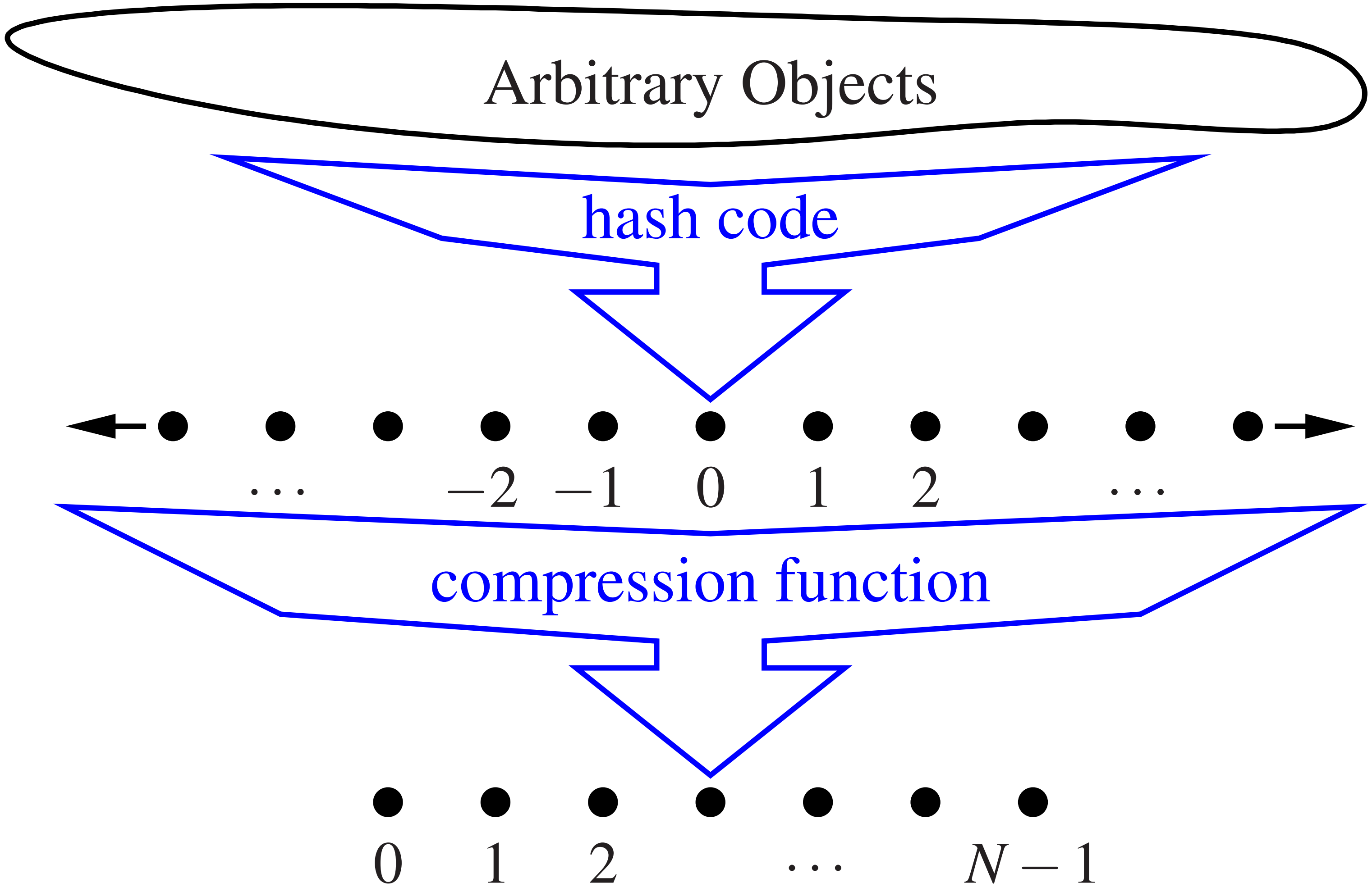
Hash Function Illustration



Hash Function Illustration



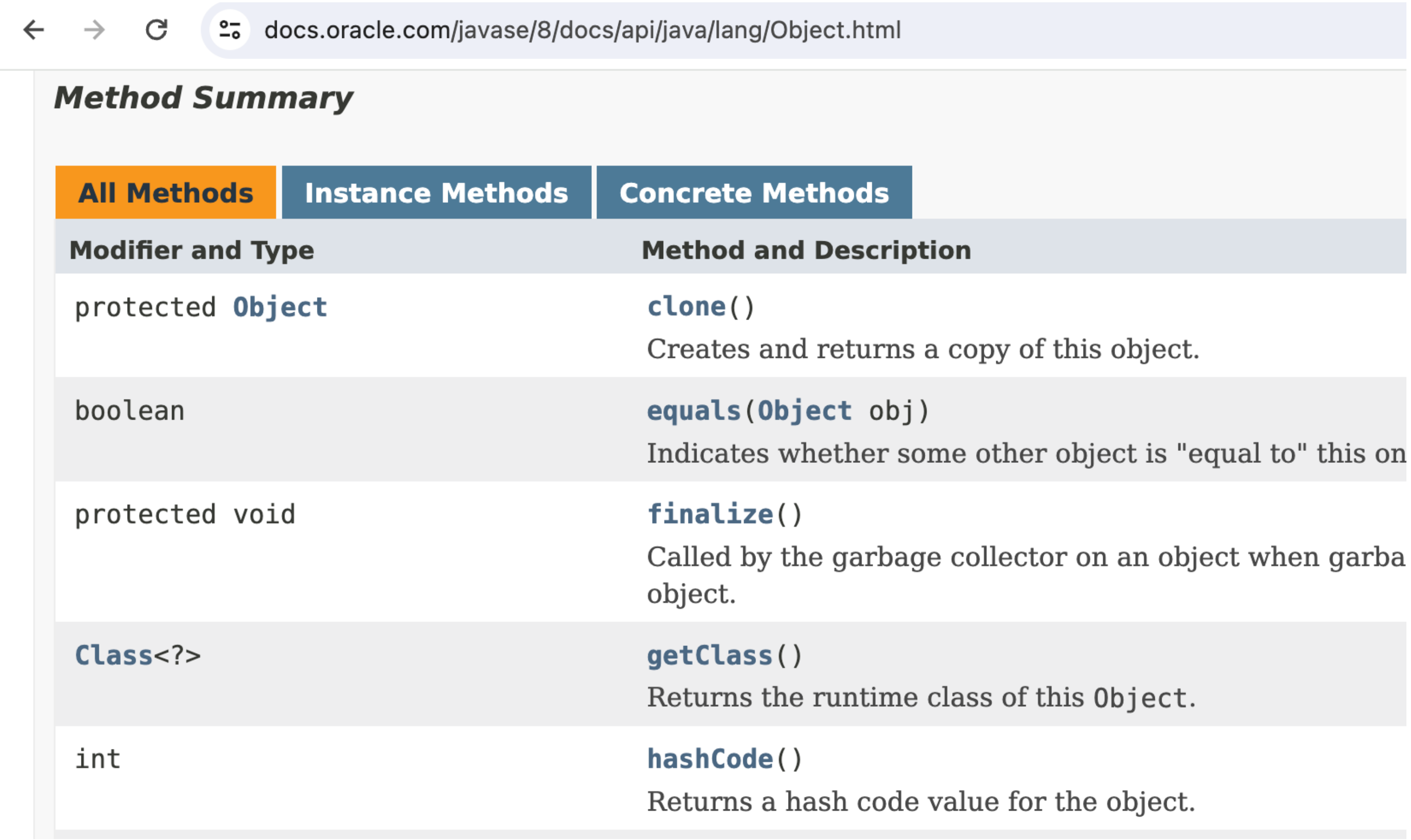
Hash Function Illustration



Hash Codes (h_1)

Memory address:

- use the memory address where the keys are stored
- default hash code for Java objects



docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this on
protected void	finalize() Called by the garbage collector on an object when garba object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.

Hash Codes (h_1)

- Memory address:
  - use the memory address where the keys are stored
  - default hash code for Java objects

- Integer cast: interpret the bits storing the keys as integer – `byte`, `short`, `int` and `float`
- Component sum: partition bits into int components and sum them – `long` and `double`

Compression (h_2)

Division: $h_2(x) = x\%N$

N is usually chosen to be a prime

MAD: $h_2(x) = ((ax + b)\%p)\%N$

- N is the size
- p is a prime number, $p > N$
- $0 < a \leq p - 1, 0 \leq b \leq p - 1$
- a scales the range and b shifts the start
- minimize the probability of two keys colliding – $\frac{1}{N}$

AbstractHashMap

```
public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
  protected int n = 0;                        // number of entries in the dictionary
  protected int capacity;                     // length of the table
  private int prime;                          // prime factor
  private long scale, shift;                  // the shift and scaling factors
  public AbstractHashMap(int cap, int p) {
    prime = p;
    capacity = cap;
    Random rand = new Random();
    scale = rand.nextInt(prime−1) + 1;
    shift = rand.nextInt(prime);
    createTable();
  }
  public AbstractHashMap(int cap) { this(cap, 109345121); }   // default prime
  public AbstractHashMap() { this(17); }                      // default capacity
  // public methods
  public int size() { return n; }
  public V get(K key) { return bucketGet(hashValue(key), key); }
  public V remove(K key) { return bucketRemove(hashValue(key), key); }
  public V put(K key, V value) {
    V answer = bucketPut(hashValue(key), key, value);
    if (n > capacity / 2)                     // keep load factor <= 0.5
      resize(2 * capacity − 1);               // (or find a nearby prime)
    return answer;
  }
```

AbstractHashMap

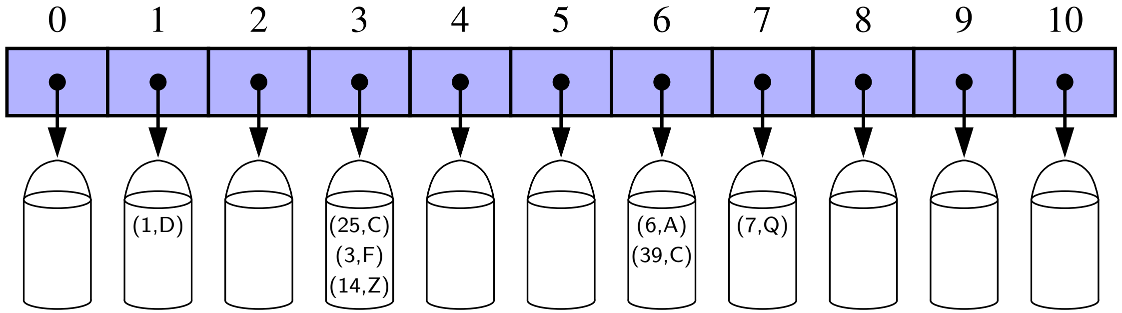
```
  // private utilities
  private int hashValue(K key) {
    return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
  }
  private void resize(int newCap) {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
    for (Entry<K,V> e : entrySet())
      buffer.add(e);
    capacity = newCap;
    createTable();                    // based on updated capacity
    n = 0;                            // will be recomputed while reinserting entries
    for (Entry<K,V> e : buffer)
      put(e.getKey(), e.getValue());
  }
  // protected abstract methods to be implemented by subclasses
  protected abstract void createTable();
  protected abstract V bucketGet(int h, K k);
  protected abstract V bucketPut(int h, K k, V v);
  protected abstract V bucketRemove(int h, K k);
}
```

Collision

A hash function does not guarantee one-to-one mapping – no hash function does

When more than one key hash to the same index, we have a bucket

Each index holds a collection of entries



Collision Handling

Collisions occur when elements with different keys are mapped to the same cell

Separate Chaining: let each cell in the table point to a linked list of entries that map there

Simple, but requires additional memory besides the table

Separate Chaining

using a list-based map at each cell. `A` is the table

```
get(k):
```

```
put(k, v):
```

```
remove(k):
```

Skip

skip

Separate Chaining

using a list-based map at each cell. `A` is the table

```
get(k):
  return A[h(k)].get(k)
put(k, v):
  t = A[h(k)].put(k)
  if t == null then n++  //k is new
  return t
remove(k):
  t = A[h(k)].remove(k)
  if t != null then n--  //k found
  return t
```

ChainHashMap

```java
public class ChainHashMap<K,V> extends AbstractHashMap<K,V> {
  // a fixed capacity array of UnsortedTableMap that serve as buckets
  private UnsortedTableMap<K,V>[ ] table;   // initialized within createTable
  public ChainHashMap( ) { super( ); }
  public ChainHashMap(int cap) { super(cap); }
  public ChainHashMap(int cap, int p) { super(cap, p); }
  /** Creates an empty table having length equal to current capacity. */
  protected void createTable( ) {
    table = (UnsortedTableMap<K,V>[ ]) new UnsortedTableMap[capacity];
  }
  /** Returns value associated with key k in bucket with hash value h, or else null. */
  protected V bucketGet(int h, K k) {
    UnsortedTableMap<K,V> bucket = table[h];
    if (bucket == null) return null;
    return bucket.get(k);
  }
  /** Associates key k with value v in bucket with hash value h; returns old value. */
  protected V bucketPut(int h, K k, V v) {
    UnsortedTableMap<K,V> bucket = table[h];
    if (bucket == null)
      bucket = table[h] = new UnsortedTableMap<>( );
    int oldSize = bucket.size( );
    V answer = bucket.put(k,v);
    n += (bucket.size( ) − oldSize);      // size may have increased
    return answer;
  }
```

ChainHashMap

```
/** Removes entry having key k from bucket with hash value h (if any). */
protected V bucketRemove(int h, K k) {
  UnsortedTableMap<K,V> bucket = table[h];
  if (bucket == null) return null;
  int oldSize = bucket.size();
  V answer = bucket.remove(k);
  n -= (oldSize - bucket.size());      // size may have decreased
  return answer;
}
/** Returns an iterable collection of all key-value entries of the map. */
public Iterable<Entry<K,V>> entrySet() {
  ArrayList<Entry<K,V>> buffer = new ArrayList<>();
  for (int h=0; h < capacity; h++)
    if (table[h] != null)
      for (Entry<K,V> entry : table[h].entrySet())
        buffer.add(entry);
  return buffer;
}
}
```

Separate Chaining

using a list-based map at each cell. `A` is the table

```
get(k):
  return A[h(k)].get(k)
put(k, v):
  t = A[h(k)].put(k)
  if t == null then n++  //k is new
  return t
remove(k):
  t = A[h(k)].remove(k)
  if t != null then n--  //k found
  return t
```

Open Addressing and Probing

- Colliding item is put in a different cell
- Linear probing: place the colliding item in the next (circularly) available table cell
- Colliding items cluster together – future collisions to cause a longer sequence of probes

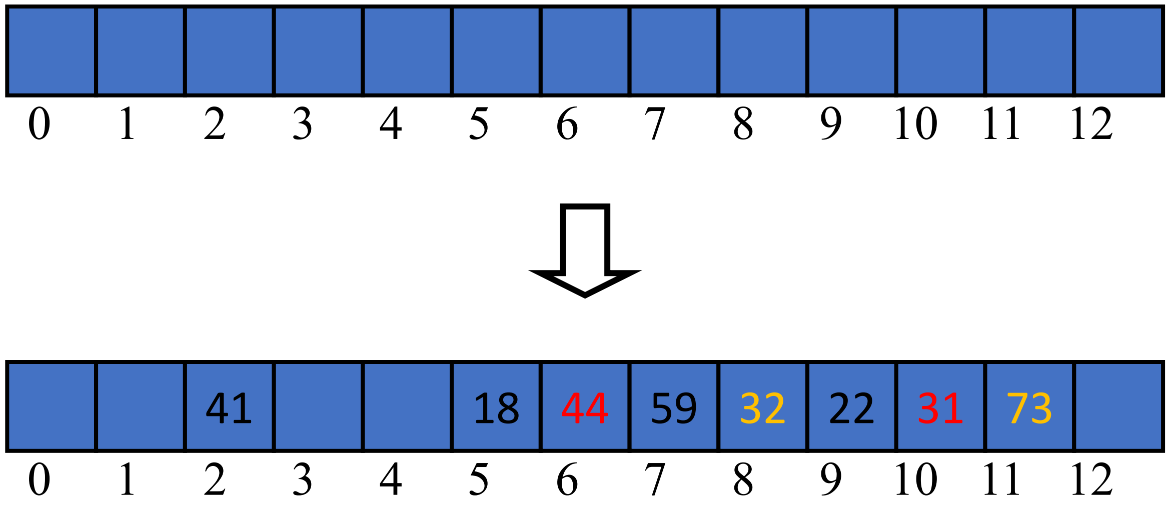
- Example: $h(x) = x\%13$
- insert 18(5), 41(2), 22(9), 44(5), 59(7), 32(6), 31(5), 73(8)

skip

Skip

Open Addressing and Probing

- Example: $h(x) = x\%13$
- insert 18(5), 41(2), 22(9), 44(5), 59(7), 32(6), 31(5), 73(8)



- collision: 4

ProbeHashMap

```
public class ProbeHashMap<K,V> extends AbstractHashMap<K,V> {
  private MapEntry<K,V>[ ] table;        // a fixed array of entries (all initially null)
  private MapEntry<K,V> DEFUNCT = new MapEntry<>(null, null);   //sentinel
  public ProbeHashMap() { super(); }
  public ProbeHashMap(int cap) { super(cap); }
  public ProbeHashMap(int cap, int p) { super(cap, p); }
  /** Creates an empty table having length equal to current capacity. */
  protected void createTable() {
    table = (MapEntry<K,V>[ ]) new MapEntry[capacity];   // safe cast
  }
  /** Returns true if location is either empty or the "defunct" sentinel. */
  private boolean isAvailable(int j) {
    return (table[j] == null || table[j] == DEFUNCT);
  }
```

ProbeHashMap

```
/** Returns index with key k, or −(a+1) such that k could be added at index a. */
private int findSlot(int h, K k) {
  int avail = −1;                              // no slot available (thus far)
  int j = h;                                   // index while scanning table
  do {
    if (isAvailable(j)) {                      // may be either empty or defunct
      if (avail == −1) avail = j;              // this is the first available slot!
      if (table[j] == null) break;             // if empty, search fails immediately
    } else if (table[j].getKey( ).equals(k))
      return j;                                // successful match
    j = (j+1) % capacity;                      // keep looking (cyclically)
  } while (j != h);                            // stop if we return to the start
  return −(avail + 1);                         // search has failed
}
/** Returns value associated with key k in bucket with hash value h, or else null. */
protected V bucketGet(int h, K k) {
  int j = findSlot(h, k);
  if (j < 0) return null;                      // no match found
  return table[j].getValue( );
}
```

ProbeHashMap

```
/** Associates key k with value v in bucket with hash value h; returns old value. */
protected V bucketPut(int h, K k, V v) {
  int j = findSlot(h, k);
  if (j >= 0)                                      // this key has an existing entry
    return table[j].setValue(v);
  table[−(j+1)] = new MapEntry<>(k, v);           // convert to proper index
  n++;
  return null;
}
/** Removes entry having key k from bucket with hash value h (if any). */
protected V bucketRemove(int h, K k) {
  int j = findSlot(h, k);
  if (j < 0) return null;                          // nothing to remove
  V answer = table[j].getValue();
  table[j] = DEFUNCT;                              // mark this slot as deactivated
  n−−;
  return answer;
}
/** Returns an iterable collection of all key-value entries of the map. */
public Iterable<Entry<K,V>> entrySet() {
  ArrayList<Entry<K,V>> buffer = new ArrayList<>();
  for (int h=0; h < capacity; h++)
    if (!isAvailable(h)) buffer.add(table[h]);
  return buffer;
}
}
```

Probing Distance

- Given a hash value $h(x)$, linear probing generates $h(x), h(x) + 1, h(x) + 2, \ldots$
- Primary clustering – the bigger the cluster gets, the faster it grows
- **Quadratic probing** – $h(x), h(x) + 1, h(x) + 4, h(x) + 9, \ldots$
- Quadratic probing leads to secondary clustering, more subtle, not as dramatic, but still systematic

Double Hashing

- Interval between probes is fixed but computed by a second hash function

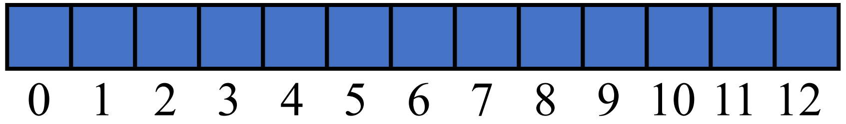
- Use a secondary hash function $d(k)$ to handle collisions by placing an item in the first available cell of the series

$$\big(h(k) + i \times d(k)\big)\%N, \qquad 0 \leq i \leq N-1$$

- N must be prime
- $d(k) = q - k\%q, q < N, q$ is prime

Example

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44, 59, 32, 31, 73



Skip slide

Skip slide

Example

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44, 59, 32, 31, 73
- Collision: 2

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Performance Analysis

- In the worst case, searches, insertions and removals take $O(n)$ time
  - when all the keys collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
  - expected number of probes for an insertion with open addressing is $\frac{1}{1-\alpha}$
- Expected time of all operations is $O(1)$ provided α is not close to 100%

Open Addressing vs Chaining

- Probing is significantly faster in practice
- locality of references – much faster to access a series of elements in an array than to follow the same number of pointers in a linked list
- Efficient probing requires soft/lazy deletions – tombstoning, why?
- May require graveyard defragmenting

Probing Tradeoffs

- Linear probing – best cache performance but most sensitive to clustering
- Double hashing – poor cache performance but exhibits virtually no clustering
- Quadratic – inbetween

- As load factor approaches 100%, number of probes rises dramatically
- Even with good hash functions, keep load factor 80% or below (50% is typical)
- Other open addressing methods besides probing

Good Hash Function

- is critical to performance
- A poor hash function can lead to poor performance even at very low load factor
- It is easy to unintentionally write a hash function that leads to severe clustering
- Testing your hash function is paramount
- stick with `.hashCode()`

Implementation

- Interface `Entry` defines the expected behaviors of an entry (key-value pair)
- Interface `Map` defines the expected behaviors of a hashmap as an ADT

Implementation

- Abstract class `AbstractMap` implements `Map`
  - base class to provide support for misc utilities
  - `isEmpty, toString`
  - nested class `MapEntry` implements `Entry`
  - iterators
    - `keyset, values`
    - depends on `entrySet` to be provided by concrete child class

Implementation

- `AbstractHashMap` extends `AbstractMap`
  - base class to provide support for hashing
  - `capacity, prime, scale, shift, n`
  - expands table when out of space on put
  - `.hashCode` implements h_1
  - `hashValue` implements h_2 with MAD

Implementation

- `ProbeHashMap` extends `AbstractHashMap`
  - Concrete class
  - Open-addressing with linear probing
  - `bucketGet`, `bucketPut`, `bucketRemove`
  - `findSlot`

Performance of Hashtable

	Hash Expected	Hash Worst
search		
insert		
remove		
min/max		

	Unsorted array	Sorted array	Unsorted list	Sorted list	BST balanced	Hash Expected
search	$O(n)^*$	$O(logn)$	$O(n)$	$O(n)$	$O(logn)$	$O(1)$
insert	$O(1)^*$	$O(n)$	$O(1)$	$O(n)$	$O(logn)$	$O(1)$
remove	$O(1)^*$	$O(n)$	$O(1)$	$O(1)$	$O(logn)$	$O(1)$
min/max	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(logn)$	$O(n)$

Performance of Hashtable

	Hash Expected	Hash Worst
search	$O(1)$	$O(n)$
insert	$O(1)$	$O(n)$
remove	$O(1)$	$O(n)$
min/max	$O(n)$	$O(n)$

	Unsorted array	Sorted array	Unsorted list	Sorted list	BST balanced	Hash Expected
search	$O(n)^*$	$O(logn)$	$O(n)$	$O(n)$	$O(logn)$	$O(1)$
insert	$O(1)^*$	$O(n)$	$O(1)$	$O(n)$	$O(logn)$	$O(1)$
remove	$O(1)^*$	$O(n)$	$O(1)$	$O(1)$	$O(logn)$	$O(1)$
min/max	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(logn)$	$O(n)$

Hashtable vs Array

- A hashtable is an unsorted array with a fast search – $O(1)$ expected
- An array is more memory efficient, but slower for searching (without key-index pairing)
- If your data has natural indexing – a way to assign/associate an ID/unique integer to each entry, then you are better off using an array. You have a hash function with 1-to-1 mapping and guaranteed no collisions

Hashtable Size

- Should be a prime
- twice the size of max number of keys
- or 1.3 times if n is very large
- 1/1.333 = 75% load factor
- Keep track of load factor and expand (rehash) the hash table when necessary