Chapter 14

Hashing

averageTime$_S$(n), the average time for a successful search

averageTime$_U$(n), … unsuccessful …

worstTime$_S$(n)

worstTime$_U$(n)

Let's start with a review of earlier search techniques:

Sequential Search

```
/**
 *  Determines if this AbstractCollection object contains
 *  a specified element.
 *  The worstTime(n) is O(n).
 *
 *  @param obj – the element searched for in this
 *              AbstractCollection object.
 *
 *  @return true – if this AbstractionCollection object
 *              contains obj; otherwise, return false.
 */
```

```
public boolean contains(Object obj)
{
    Iterator<E> e = iterator();
    if (obj == null)
    {
       while (e.hasNext())
        if (e.next()==null)
           return true;
    }  // if obj == null
    else
    {
       while (e.hasNext())
        if (obj.equals(e.next()))
           return true;
    } // obj != null
    return false;
} // method contains
```

The worstTime$_U$(*n*) is linear in *n*.

Ditto for worstTime$_S$(*n*), averageTime$_U$(*n*), and averageTime$_S$(*n*).

Binary search of an array

Note: The array must be sorted.

The following method is in Arrays.java:

```
public static int binarySearch(Object[ ] a, Object key)
{
    int low = 0;
    int high = a.length-1;

    while (low <= high) {
        int mid =(low + high)/2;
        Comparable midVal = (Comparable)a[mid];
        int cmp = midVal.compareTo(key);
        if (cmp < 0)
            low = mid + 1;
        else if (cmp > 0)
            high = mid - 1;
        else
            return mid; // key found
    } // while
    return -(low + 1);  // key not found
} // method binarySearch
```

The $worstTime_U(n)$ is logarithmic in *n*.

Ditto for $worstTime_S(n)$, $averageTime_U(n)$, and $averageTime_S(n)$.

Red-Black Tree Search

The following method is in TreeMap.java:

```
private Entry<K, V> getEntry(Object key)
{
    Entry<K, V> p = root;
    K k = (K)key;
    while (p != null)
    {
        int cmp = compare(k,p.key);
        if (cmp == 0)
            return p;
        else if (cmp < 0)
            p = p.left;
        else
            p = p.right;
    } // while
    return null;
} // method getEntry
```

The $worstTime_U(n)$ is logarithmic in *n*.

Ditto for $worstTime_S(n)$, $averageTime_U(n)$, and $averageTime_S(n)$.

Now let's focus on an unusual but very efficient search technique:

Hashing

The class in which hashing is implemented is the HashMap class.

To a user, the HashMap **class seems almost identical to the** TreeMap **class, except for the timing estimates.**

```
public class HashMap<K,V>
    extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
```

Recall that each element in a map consists of a unique key and a value.

Method descriptions for the HashMap **class:**

```
/**
 *  Initializes this HashMap object to be empty, with a
 *  default initial capacity.
 */
public HashMap( )
```

```
/**
 *  Initializes this HashMap object to be empty, with a
 *  default initial capacity.
 */
public HashMap( )
```

Where have you seen this before?

```
/**
 *  Initializes this HashMap object to be empty, with a
 *  specified initial capacity.
 *
 *  @param initialCapacity – the specified initial capacity.
 *
 */
public HashMap (int initialCapacity)
```

```
/**
 * Determines if this HashMap object contains a mapping
 * with a specified value.
 *
 * @param value – the specified value
 *
 * @return true – if this HashMap object contains a mapping
 *              with the specified value; otherwise, false.
 */
public boolean containsValue (Object value)
```

```
/**
 * Determines if this HashMap object contains a mapping
 * with a specified key.
 *
 * @param key – the specified key
 *
 * @return true – if this HashMap object contains a mapping
 *              with the specified key; otherwise, false.
 */
public boolean containsKey (Object key)
```

```
/**
 *  Determines if this HashMap object has a mapping
 *  that has a specified key.
 *
 *  @param key – the specified key
 *  @return the value corresponding to the specified key,
 *              if this HashMap object has a mapping with
 *              the specified key; otherwise, returns null.
 */
public V get (Object key)
```

In what sense is this method "better" than containsKey**? In what sense is it worse?**

```
/**
 *  Ensures that there is an element in this HashMap object
 *  with the specified key&value pair. If this HashMap
 *  object had an element with the specified key before
 *  this method was called, the previous value associated
 *  with that key has been returned.  Otherwise, null
 *  has been returned.
 *
 *  @param key – the specified key
 *  @param value – the specified value
 *  @return the previous value associated with key, if
 *              there was such a mapping; otherwise, null.
 *
 */
public V put (K key, V value)
```

```
/**
 *  Ensures that there is no mapping in this HashMap object
 *  with the specified key. If this HashMap object had such
 *  a mapping before this method was called, the value
 *   has been returned.  Otherwise, null has been returned.
 *
 *  @param key – the specified key
 *
 *  @return the value associated with key, if
 *              there was such a mapping; otherwise, null.
 *
 */
public V remove (Object key)
```

And other methods you also saw in the TreeMap **class:**

size, keySet, entrySet, values, toString, …

We'll study the time estimates after we define the methods. But basically, for containsKey**,** get**,** put**, and** remove**,**

averageTime$_S$(n) is constant!

```
HashMap<String, Integer> ageMap =
        new HashMap<String, Integer>();

ageMap.put ("dog", 15);
ageMap.put ("cat", 20);
ageMap.put ("human", 75);
ageMap.put ("turtle", 100);
System.out.println (ageMap);
for (Map.Entry<String, Integer> entry :
                                ageMap.entrySet())
    if (entry.getValue() > 50)
        System.out.println (entry.getKey());

Iterator<String, Integer> itr =
                ageMap.entrySet().iterator();
while (itr.hasNext())
    if (itr.next().getValue() >= 20)
    itr.remove();
System.out.println (ageMap);
```

Here's the output:

{dog=15, cat=20, turtle=100, human=75}
turtle
human
{dog=15}

Recall that the TreeMap **class used the "natural" ordering supplied by the** Comparable **interface, or an ordering supplied by a comparator.**

What about HashMap **objects? Are they ordered?**

Stick around!

Fields in the HashMap **class**

Continguous

array? ArrayList**?** Heap**?**

Linked

LinkedList? TreeMap?

But none of these will give constant average time for searches, insertions and removals.

Here is the main idea:

private transient Entry table[ ];   // to hold the elements;

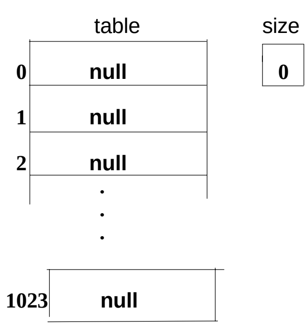
private transient int size; // number of elements in the
// HashMap object

Let's see where that leads. Suppose we have

HashMap<Integer, String> persons =
new HashMap<Integer, String> (1024);

Each key will be a (unique) 3-digit integer.

Each value will be a name.



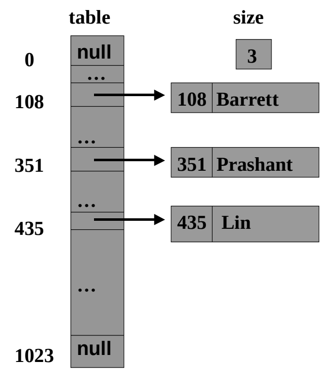
---

persons.put (351, "Prashant");

persons.put (108, "Barrett");

persons.put (435, "Lin");

Where should we store the element whose key is 351?



Now for something slightly different:

```
HashMap<Integer, String> persons =
        new HashMap<Integer, String> (1024);
```

There will be at most 1000 persons. Each key will be a 9-digit social security number. Each value will be a name.

```
persons.put (123456789, "Prashant");
persons.put (428671256, "Barrett");
persons.put (884739816, "Lin");
persons.put (403578063, "Sutey");
```

We want these elements scattered throughout the table.

The Integer **class has a** hashCode( ) **method that simply returns the underlying** int. **The HashMap class has a** hash **method:**

```
static int hash(Object x) {
    int h = x.hashCode();

    h += ~(h << 9);
    h ^=  (h >>> 14);
    h +=  (h << 4);
    h ^=  (h >>> 10);
    return h;
}
```

This hash **method scrambles up the key. For example,**

hash (123456789)

Returns 1272491941

We can get an index in the range 0 … 1023 as follows:

```
int index = hash (123456789) % 1024;      // index = 933
```

We can get the same index a little faster:

```
int index = hash (123456789) & 1023;
```

The & operator performs a "bitwise and" on its operands.

For each pair of bits a and b, if a and b are both 1 bits, a & b = 1. Otherwise, a & b = 0.

For example,

```
  10100001101001
& 00000000001111
  00000000001001
```

1023, as a 32-bit integer, is

00000000000000000000000111111111

so

w & 1023

returns the rightmost 9 bits of the operand w. In general, this works well as long as the table length is a power of 2.

123-45-6789 → 933

428-67-1256 → 500

884-73-9816 → 234

123-45-6789 → 933

428-67-1256 → 500

884-73-9816 → 234

403-57-8063 → 933 Oops!

When two different keys yield the same index, that is called a *collision*.

Keys that yield the same index are called *synonyms*.

Hashing:

The process of transforming a key into an array index.

Here is the general idea:

key —hash (key) & table.length – 1→ index

and then handle collisions. We'll study collisions handlers soon.

In the String class:

```
public int hashCode( ) {

    int h = 0;
    int off = offset;  // index of first character in array
                       // value
    char val[ ] = value;    // value is the array of char that
                            // holds the String
    int len = count; // count holds the number of
                     // characters in the String

    for (int i = 0; i < len; i++)
        h = 31*h + val[off++];

    return h;

} // method hashCode
```

Exercise: Calculate "cat".hashCode( ).

Hint: 'c' has an integer value of 99, 'a' … 97, 't' … 116

This is mainly an arithmetic exercise to show you how keys of type String **get hashed into a table. For example, hash ("cat") & 127 = 91.**

As you might have guessed, hashing is inefficient when there are a lot of collisions.

Users of the HashMap **class "hope" that the keys are scattered randomly throughout the table. This hope is formally stated as follows:**

The Uniform Hashing Assumption

Each key is equally likely to hash to any one of the table addresses, independently of where the other keys have hashed.

Even if the uniform hashing assumption holds, there may still be collisions.

Now we'll look at collision handlers.

***Chaining*: At index** i **in** table**, store the linked list of all elements whose keys hash to** i**.**

This is how the Java collections framework implements hashing. Note: The table length must be a power of 2.

```
transient Entry table[ ];   // an array of type Entry;
                            // at each index in table,
                            // we will  store the
                            // singly-linked list of all
                            // those elements whose
                            // keys that hash to that
                            // index

transient int size;  // the number of elements in the
                     // HashMap;

float loadFactor; // the maximum ratio of  size /
                  // table.length before resizing of table
                  // will occur;

int threshold; // = (int) (table.length * loadFactor);
               //  when size reaches threshold, the
               //  table is resized (to 2 * table.length)
```

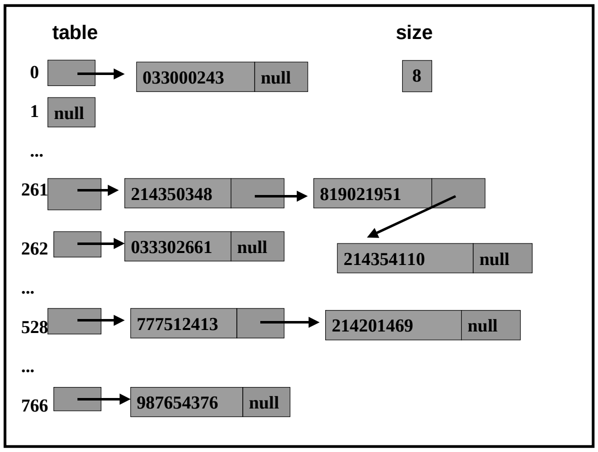
```
static class Entry<K,V> implements Map.Entry<K,V>
{
      final K key;  // key, once set, cannot be changed
      V value;
      final int hash;  // to avoid recalculation
      Entry<K,V> next;

      Entry(int h, K k, V v, Entry<K,V> n)
      {
         value = v;
         next = n;
         key = k;
         hash = h;
      }
```

Insert elements with these keys into a table of length 1024:

214-20-1469
987-65-4376
214-35-4110
033-00-0243
819-02-1951
777-51-2413
214-35-0348
033-30-2661

Note: These numbers were "rigged" to get collisions.



Exercise: Assume table.length = **1024 and** loadFactor = **0.75. Then** table **will be resized, to 2048, when** size >= **768 and** put **is called.**

1. **What is the maximum number of elements that can be stored at an index when** table.length = **1024?**

2. **What is the average number of elements that have been stored at each index when** size = **512 and** table.length = **1024?**

Implementation of the HashMap **Class**

For the containsKey, get, put, **and** remove **methods, the initial strategy is the same:**

Hash key **to** index;

Search linked list at table [index]**.**

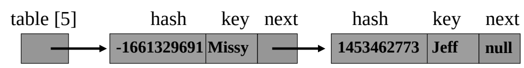
```
public boolean containsKey(Object key) {

    Object k = maskNull(key);
    int hash = hash(k);
    int i = indexFor(hash, table.length);
    Entry e = table[i];
    while (e != null) {
        if (e.hash == hash && eq(k, e.key))
            return true;
        e = e.next;
    }
    return false;
}
```

The code for the put **method is similar, except we need to replace and return the old value if there is a matching key. And before we can insert a new key-value pair, we have to consider resizing.**

To rehash, the size of the table is doubled, and then each entry from the old table is hashed to the new table. Since each entry includes a hash **field, the** hash **value is not re-calculated.**

For example, suppose the old table had length 16, and the following list at table [5]:

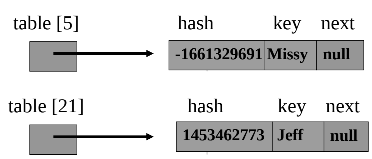


If resize( ) **is called, the new length of** table **will be 32.**

-1661329691 & 31 = 5

1453462773 & 31 = 21

Part of table **will be:**



The remove **method follows the same search through** table [index] **as** containsKey **and** put**, except that there is a reference,** prev**, to the entry before the entry to be removed. To remove entry** e**:**

```
if (table [index] == e)
    table[index] = e.next;
else
    prev.next = e.next;
```

Time estimates:

Let *n* = size**, let *m*** = table.length**.**

Assume the uniform hashing assumption holds.

The average size of each list is

n* / *m

For the containsKey **method,**

averageTime$_S$(*n*, *m*) ≈ *n* / 2*m* iterations.

but *n* / *m* <= loadFactor, a constant (assigned in the constructor)

so averageTime$_S$(*n*, *m*) < a constant.

averageTime$_S$(*n*, *m*) is constant.

Even if the uniform hashing assumption holds, it is possible for each key to hash to the same index. To search the list at that index takes linear-in-n time.

So worstTime$_S$(*n*, *m*) is linear in *n*.

The same results, constant average time and linear worst time, hold for unsuccessful searches with

- containsKey
- get
- put
- remove

The HashIterator **class**

Iterate through table **starting at** table [length – 1]

Not at table [0]

The put **method inserts each element at the *front* of the linked list, and the iterator starts at the front of a linked list, so the elements are accessed in opposite order from insertion.**

Note: Users iterate through a HashMap **object by choosing a view:** entrySet( ), keySet( ), **or** values( ) **.**

Worst case for next( )**:**

Let *n* = size**. Let *m*** = table.length**.**

Suppose the iterator is currently at the last entry in the list at table [length –1] **, and the next entry is at** table [0]**.**

The worstTime(*n*, *m*) is ???

Exercise: **Develop a** main **method that constructs an empty** HashMap **object,** studentMap**, with an initial capacity of 1024. Each key will represent a student's ID (L-number) and each value will represent the student's grade point average.**

Insert three elements into studentMap **and then develop an enhanced** for **statement to print out the student ID of each student whose grade point average is above 3.0.**

The HashSet **class: See** TreeSet **class**

```
/**
 *  Inserts an element into this HashSet object, unless the element
 *  was already in this HashSet object before this method was
 *  called.  The worstTime(n, m) is O(n).  If the Uniform Hashing
 *  Assumption holds, averageTime(n, m) is constant.
 *
 *  @param element – the element whose insertion is attempted
 *  @return true – if element was inserted as a result of this call
 *
 */
public boolean add(E element)
{
      return map.put (element, PRESENT) == null;
                                          // PRESENT is a
                                          // dummy value-part
} // method add
```