Hashing

Dynamic Dictionaries

Operations:
- create
- insert
- find
- remove
- max/min
- write out in sorted order

Only defined for object classes that are Comparable
Hash tables

Operations:
• create
• insert
• find
• remove
• max/min
• write out in sorted order

Only defined for object classes that are Comparable have equals defined

---

public boolean equals(Object obj)  

Java specific: From the Java documentation

Indicates whether some other object is “equal to” this one.

The equals method implements an equivalence relation on non-null object references:

• It is reflexive: for any non-null reference value x, x.equals(x) should return true.
• It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
• It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
• It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
• For any non-null reference value x, x.equals(null) should return false.

The equals method for class object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.
Hash tables – implementation

- Have a table (an array) of a fixed `tableSize`
- A hash function determines where in this table each item should be stored
  
  \[ \text{hash}(\text{item}) \mod \text{tableSize} \]
  
  [a positive integer]

THE DESIGN QUESTIONS

1. Choosing `tableSize`
2. Choosing a hash function
3. What to do when a collision occurs

Hash tables - `tableSize`

- Should depend on the (maximum) number of values to be stored
- Let \( \lambda = \frac{\text{number of values stored}}{\text{tableSize}} \)
  - Load factor of the hash table
  - Restrict \( \lambda \) to be at most 1 (or \( \frac{1}{2} \))
- Require `tableSize` to be a prime number
  - to “randomize” away any patterns that may arise in the hash function values
- The prime should be of the form \((4k+3)\)
  [for reasons to be detailed later]
Hash tables – the hash function

If the objects to be stored have integer keys (e.g., student IDs)

\[ \text{hash}(k) = k \] is generally OK, unless the keys have "patterns"

Otherwise, some "randomized" way to obtain an integer

```java
public static int hash(String key, int tableSize)
{
    int hashVal = 0;
    for (int i = 0; i < key.length(); i++)
        hashVal += key.charAt(i);
    return hashVal % tableSize;
}
```

**Figure 5.2** A simple hash function

Hash tables – the hash function

If the objects to be stored have integer keys (e.g., student IDs)

\[ \text{hash}(k) = k \] is generally OK, unless the keys have "patterns"

Otherwise, some "randomized" way to obtain an integer

```java
public static int hash(String key, int tableSize)
{
    return (key.charAt(0) + 27 * key.charAt(1) +
        729 * key.charAt(2)) % tableSize;
}
```

**Figure 5.3** Another possible hash function—not too good
Hash tables – the hash function

If the objects to be stored have integer keys (e.g., student IDs)

\[ \text{hash}(k) = k \text{ is generally OK, unless the keys have "patterns"} \]

Otherwise, some "randomized" way to obtain an integer

Java-specific

- Every class has a default `hashCode()` method that returns an integer
- May be (should be) overridden
- Required properties
  - consistent with the class's `equals()` method
  - need not be consistent across different runs of the program
  - different objects may return the same value!
Hash tables - the hash function

From the Java 1.5.0 documentation
http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html#hashCode%28%29

As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

The Java™ programming language need not be consistent across different runs of the program different objects may return the same value!

```java
public class Employee {
    int employeeId;
    String name;
    Department dept;

    // other methods would be in here

    // Override
    public int hashCode() {
        int hash = 1;
        hash = hash * 17 + employeeId;
        hash = hash * 31 + name.hashCode();
        hash = hash * 13 + (dept == null ? 0 : dept.hashCode());
        return hash;
    }
}
```
Hash tables - collision resolution

The universe of possible items is usually far greater than tableSize

Collision: when multiple items hash on to the same location (aka cell or bucket)

Collision resolution strategies specify what to do in case of collision

1. Chaining (closed addressing)
2. Probing (open addressing)
   a. Linear probing
   b. Quadratic probing
   c. Double Hashing
   d. Perfect Hashing
   e. Cuckoo Hashing

Hash tables - collision resolution: chaining

Maintain a linked list at each cell/ bucket

(The hash table is an array of linked lists)

Insert: at front of list
- if pre-cond is that not already in list, then faster
- in any case, later-inserted items often accessed more frequently

Example: Insert $0^2$, $1^2$, $2^2$, ..., $9^2$ into an initially empty hash table with tableSize = 10

[Note: bad choice of tableSize - only to make the example easier!!]
Hash tables – collision resolution: **chaining**

Maintain a **linked list** at each cell/ bucket

(The hash table is an **array of linked lists**)

**Insert**: at front of list
- if pre-cond is that not already in list, then faster
- in any case, later-inserted items often accessed more frequently

**Example**: Insert 0, 2, 1, 2, …, 9 into an initially empty hash table with tableSize = 10

[Note: bad choice of tableSize – only to make the example easier!!]

**Hash tables – collision resolution: chaining**

Maintain a **linked list** at each cell/ bucket

(The hash table is an array of linked lists)

**Insert**: at front of list
- if pre-cond is that not already in list, then faster
- in any case, later-inserted items often accessed more frequently

**Find** and **Remove**: obvious implementations

**Worst-case** run-time: $O(N)$ per operation (all elements in the same list)

**Average case**: $O(\lambda)$ per operation

Design rule: for chaining, keep $\lambda \leq 1$

If $\lambda$ becomes greater than 1, rehash (later)
Hash tables - collision resolution: probing

1. Chaining (closed addressing)
2. Probing (open addressing)
   a. Linear probing
   b. Quadratic probing
   c. Double Hashing
   d. Perfect Hashing
   e. Cuckoo Hashing

In case of collision, try alternative locations until an empty cell is found
   • [Open address]

Probe sequence: $h_0(x), h_1(x), h_2(x), \ldots$, with $h_i(x) = [\text{hash}(x) + f(i)] \% \text{tableSize}$

The function $f(i)$ is different for the different probing methods

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using linear probing

---

Hash tables - collision resolution: linear probing

<table>
<thead>
<tr>
<th>Empty Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

**Figure 5.11** Hash table with linear probing, after each insertion

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using linear probing