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` and have `equals` defined.
Hash tables

public boolean equals(Object obj)  
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

  \[ \text{hash(item)} \mod \text{tableSize} \]

  \[ 2174 \equiv 10 = 4 \]

THE DESIGN QUESTIONS

1. Choosing `tableSize`
2. Choosing a hash function
3. What to do when a collision occurs
Hash tables – \texttt{tableSize}

- Should depend on the (maximum) number of values to be stored
- Let $\lambda = \left\lfloor \frac{\text{number of values stored}}{\text{tableSize}} \right\rfloor$
  - Load factor of the hash table
  - Restrict $\lambda$ to be at most 1 (or ½)
- Require \texttt{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), hash \( k = k \) is generally OK, unless the keys have "patterns". Otherwise, some "randomized" way to obtain an integer.

---

```java
1    /**
2       * A hash routine for String objects.
3       * @param key the String to hash.
4       * @param tableSize the size of the hash table.
5       * @return the hash value.
6       */
7    public static int hash( String key, int tableSize )
8    {
9        int hashVal = 0;
10
11        for( int i = 0; i < key.length( ); i++ )
12            hashVal = 37 * hashVal + key.charAt( i );
13
14        hashVal %= tableSize;
15        if( hashVal < 0 )
16            hashVal += tableSize;
17
18        return hashVal;
19    }
```

**Figure 5.4** A good 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-specific

• Every class has a default $\text{hashCode()}$ method that returns an integer
• May be (should be) overridden
• Required properties
  consistent with the class’s $\text{equals()}$ method
  need not be consistent across different runs of the program
different objects may return the same value!
Hash tables – the hash function

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.)

need not be consistent across different runs of the program
different objects may return the same value!

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

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

Restrict the load factor $\lambda = \frac{\text{[number of values stored]}}{\text{tableSize}}$ to be at most 1 (or $\frac{1}{2}$)

Require **tableSize** to be a prime number of the form $(4k + 3)$
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-specific
• Every class has a default \texttt{hashCode()} method that returns an integer
• May be \texttt{overridden}
• Required properties
  consistent with the class’s \texttt{equals()} method
  need not be consistent across different runs of the program
different objects may return the same value!
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-condition is “not already in list,” then faster
- in any case, later-inserted items often accessed more frequently (the LRU principle)

**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
- in any case, later-inserted items

**Example**: Insert $0^2, 1^2, 2^2, \ldots, 9^2$ into

[Note: bad choice of tableSize]

![A separate chaining hash table](image)
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

- The load factor: \([\text{number of items stored}] / \text{tableSize}\)
- 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**: \(\Theta(N)\) 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

**Avoids** the use of dynamic memory

- **Open address**
  
  **Probe sequence**: \( h_0(x), h_1(x), h_2(x), \ldots \), with \( h_i(x) = \text{[hash}(x) + f(i)] \mod \text{tableSize} \)

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>
<th>After 89</th>
<th>After 18</th>
<th>After 49</th>
<th>After 58</th>
<th>After 69</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>49</td>
<td>49</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>58</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>69</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</td>
<td>89</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
Hash tables - review

Supports the basic dynamic dictionary ops: **insert**, **find**, **remove**

Does **not** need class to be **Comparable**

Three design decisions: **tableSize**, **hash function**, **collision resolution**

Table size

- a **prime** of the form \((4k+3)\), keeping **load factor** constraints in mind

Hash function

- should “randomize” the items

  Java’s **hashCode**() method

Collision resolution: **chaining**

Collision resolution: **probing** (open addressing) – **linear probing**

The **clustering** problem
Hash tables - clustering

Two causes of clustering:

- multiple keys hash on to the same location (secondary clustering)
- multiple keys hash on to the same cluster (primary clustering)

Secondary clustering caused by hash function; primary, by choice of probe sequence

Number of probes per operation increases with load factor
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

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

**f(i) is a quadratic function of i (e.g., f(i) = i^2)**
Hash tables – collision resolution: **quadratic probing**

**Example:** insert 89, 18, 49, 58, and 69 into a table of size 10, using quadratic probing
Hash tables – collision resolution: quadratic probing

Two causes of clustering:

- multiple keys hash on to the same location (secondary clustering)
- multiple keys hash on to the same cluster (primary clustering)

Which one does quadratic probing solve?

primary clustering

Efficient implementation of \( i^2 \rightarrow (i+1)^2 \): (i+1) and (2i+1) in parallel, and then add \( i^2 \) and (2i+1)

Choosing tableSize:

- prime: at least half the table gets probed
- prime of the form \((4k+3)\) and probe sequence is \( \pm i^2 \): entire table gets probed

Remove: lazy delete must be used
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

To get rid of secondary clustering

Probing (open addressing)

Use two hash functions: $\text{hash}_1(.)$ and $\text{hash}_2(.)$

Probe sequence “step” size is $\text{hash}_2(.)$

- [Unlikely distinct items agree on both $\text{hash}_1(.)$ and $\text{hash}_2(.)$]

$\text{hash}_2(.)$ must never evaluate to zero!

A common (good) choice: $R - (x \mod R)$, for $R$ a prime smaller than tableSize

Example: insert 89, 18, 49, 58, and 69 into a table of size 10, using double hashing with $\text{hash}_2(x) = 7 - x \mod 7$
Hash tables – collision resolution: **double hashing**

<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>

**Example**: insert **89, 18, 49, 58, and 69** into a table of size 10, using double hashing with \( h_2(x) = 7 - x \mod 7 \)
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
Hash tables – collision resolution: **Cuckoo hashing**

**Goal:** constant-time $O(1)$ find in the worst case

Example application: network routing tables

[remove also takes $O(1)$ time]

**Insert** has worst-case $\Theta(N)$ run-time

Keep two hash tables, and use two different hash functions
Hash tables – collision resolution: **Cuckoo hashing**

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>TABLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

A: $\text{hash}_1(A) = 0$, $\text{hash}_2(A) = 2$

B: $\text{hash}_1(B) = 0$, $\text{hash}_2(B) = 0$
**Hash tables – collision resolution: Cuckoo hashing**

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>TABLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A</td>
</tr>
</tbody>
</table>

A: $\text{hash}_1(A) = 0$, $\text{hash}_2(A) = 2$

B: $\text{hash}_1(B) = 0$, $\text{hash}_2(B) = 0$

C: $\text{hash}_1(C) = 1$, $\text{hash}_2(C) = 4$

D: $\text{hash}_1(D) = 1$, $\text{hash}_2(D) = 0$
## Hash tables – collision resolution: Cuckoo hashing

<table>
<thead>
<tr>
<th></th>
<th>TABLE 1</th>
<th>TABLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>C</td>
</tr>
</tbody>
</table>

A: \(\text{hash}_1(A) = 0, \text{hash}_2(A) = 2\)

B: \(\text{hash}_1(B) = 0, \text{hash}_2(B) = 0\)

C: \(\text{hash}_1(C) = 1, \text{hash}_2(C) = 4\)

D: \(\text{hash}_1(D) = 1, \text{hash}_2(D) = 0\)

E: \(\text{hash}_1(E) = 3, \text{hash}_2(E) = 2\)

F: \(\text{hash}_1(F) = 3, \text{hash}_2(F) = 4\)
Hash tables – collision resolution: **Cuckoo hashing**

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>TABLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
</tbody>
</table>

- **A**: $\text{hash}_1(A) = 0$, $\text{hash}_2(A) = 2$
- **B**: $\text{hash}_1(B) = 0$, $\text{hash}_2(B) = 0$
- **C**: $\text{hash}_1(C) = 1$, $\text{hash}_2(C) = 4$
- **D**: $\text{hash}_1(D) = 1$, $\text{hash}_2(D) = 0$
- **E**: $\text{hash}_1(E) = 3$, $\text{hash}_2(E) = 2$
- **F**: $\text{hash}_1(F) = 3$, $\text{hash}_2(F) = 4$
Hash tables – collision resolution: **Cuckoo hashing**

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>TABLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

A: \( \text{hash}_1(A) = 0, \text{hash}_2(A) = 2 \)

B: \( \text{hash}_1(B) = 0, \text{hash}_2(B) = 0 \)

C: \( \text{hash}_1(C) = 1, \text{hash}_2(C) = 4 \)

D: \( \text{hash}_1(D) = 1, \text{hash}_2(D) = 0 \)

E: \( \text{hash}_1(E) = 3, \text{hash}_2(E) = 2 \)

F: \( \text{hash}_1(F) = 3, \text{hash}_2(F) = 4 \)
## Hash tables – collision resolution: **Cuckoo hashing**

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>TABLE 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
</tr>
</tbody>
</table>

- **A**: $\text{hash}_1(A) = 0$, $\text{hash}_2(A) = 2$
- **B**: $\text{hash}_1(B) = 0$, $\text{hash}_2(B) = 0$
- **C**: $\text{hash}_1(C) = 1$, $\text{hash}_2(C) = 4$
- **D**: $\text{hash}_1(D) = 1$, $\text{hash}_2(D) = 0$
- **E**: $\text{hash}_1(E) = 3$, $\text{hash}_2(E) = 2$
- **F**: $\text{hash}_1(F) = 3$, $\text{hash}_2(F) = 4$
Hash tables – collision resolution: **Cuckoo hashing**

Insert
- Insert into Table 1, using hash$_1$
- If cell is already occupied
  - **bump** item into other table (using appropriate hash function)
  - Repeat
- **Rehash** after k repetitions

Each table should be more than half empty

*Stronger* condition than load factor $\leq \frac{1}{2}$
Rehashing

When load factor becomes too large...

(Approximately) **double** tableSize

**Scan** old table, inserting each non-deleted item into the new table

**Worst-case** time?

- $O(N^2)$

**Average-case**: $O(N)$

**Amortized analysis**

Average cost per insert, over a sequence of repeated re-hashings

[Not great for interactive applications...]
Hash tables - review

Supports the basic dynamic dictionary ops: **insert, find, remove**

Three design decisions: **tableSize, hash function, collision resolution**

Table size: a prime of the form \((4k+3)\), keeping **load factor** constraints in mind

Hash function

- Java’s `hashCode()` method

**item** goes to `hash(item) % tableSize`

Collision: multiple items at the same location

Collision resolution: - **chaining**

- **probing** (open addressing)
  - Linear probing
  - Quadratic probing
  - Double Hashing
  - Cuckoo Hashing
Java-specific – `hashCode()` and `equals()`

```java
public class Employee {
    String name;
    int id;
    public Employee(String n, int i){name = n; id = i;}

    ... ...
}

public static void main(String[] args) {
    Employee e1=new Employee("weiss", 001);
    Employee e2 = e1;
    System.out.println(e1.hashCode() + " , " + e2.hashCode());
    System.out.println(e1 == e2);
    System.out.println(e1.equals(e2));
}```
Hash tables – collision resolution: **linear probing**

f(i) can be any **linear** function \((a \times i + b)\)

If \(\text{gcd}(a, \text{tableSize}) = 1\), then linear probing will probe the entire table

**Primary clustering**: blocks of occupied cells start forming even in a relatively empty table
Hash tables – collision resolution: **linear probing**

\[ f(i) \text{ can be any linear function } (a \cdot i + b) \]

If \( \gcd(a, \text{tableSize}) = 1 \), then linear probing will probe the entire table.

**Primary clustering**: blocks of occupied cells start forming even in a relatively empty table.

```
any item hashing here... grows the cluster by one
```
Hash tables – collision resolution: **linear probing**

f(i) can be any **linear** function \((a \times i + b)\)

If \(\text{gcd}(a, \text{tableSize}) = 1\), then linear probing will probe the entire table

**Primary clustering**: blocks of occupied cells start forming even in a relatively empty table

---

any item hashing here... **merges** the two clusters
Hash tables - clustering

Two causes of clustering:

- multiple keys hash on to the same location (secondary clustering)
- multiple keys hash on to the same cluster (primary clustering)

Secondary clustering caused by hash function; primary, by choice of probe sequence

Number of probes per operation increases with load factor
Hash tables – linear probing: **remove**

- Load factor computed including lazy-deleted items
- In inserts, may “reclaim” lazy-deleted cells

<table>
<thead>
<tr>
<th>Index</th>
<th>Hash Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
</tr>
</tbody>
</table>

```
insert A; hash(A) = 4
insert B; hash(B) = 5
insert C; hash(C) = 4
remove B
find C
```