Hash-based Dictionary Implementations

Chapters 21, 22

Supplementary Readings:

(a)  http://en.wikipedia.org/wiki/Hash_function
(b)  http://en.wikipedia.org/wiki/Hash_table
Recall Dictionary and Basic Implementations

- Chapter 19: Dictionary ADT
- Chapter 20: Basic Implementations
  - Either an array
  - Or a chain of linked nodes
  - Most operations $O(N)$
    - Except search on sorted array
  - Q? Can we do better than $O(N)$?
  - Chapter 21, 22: hashing

```
// Listing 19-1, pp. 476-477, Textbook
// Recall that keys are unique within a dictionary
import java.util.Iterator;
public interface DictionaryInterface < K, V > {
    public V add (K key, V value);
    public V remove (K key);
    public V getValue (K key);
    public boolean contains (K key);
    public Iterator < K > getKeyIterator();
    public Iterator < V > getValueIterator();
    public boolean isEmpty();
    public int getSize();
    public void clear();
}
```

<table>
<thead>
<tr>
<th>Dictionary Implementations</th>
<th>Unsorted</th>
<th>Sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array-based</td>
<td>- Linear Search</td>
<td>+ Binary Search</td>
</tr>
<tr>
<td></td>
<td>+ no shift needed for add(), remove()</td>
<td>- Shift entries for add(), remove()</td>
</tr>
<tr>
<td>Chain of linked Nodes</td>
<td>- Linear Search</td>
<td>- Linear Search (early exit)</td>
</tr>
<tr>
<td></td>
<td>+ no shift needed for add(), remove()</td>
<td>+ no shift needed for add(), remove()</td>
</tr>
</tbody>
</table>
Objectives

• Describe basic ideas
  ▪ Hashing, (perfect) hash function, hash table, index to hash table
  ▪ Why override `hashCode()` for objects used as search keys

• Use hashing to implement ADT dictionary
  ▪ Implementing operations `getValue`, `add`, and `remove`
  ▪ Handling collisions with separate chaining

• Discuss efficiencies
  ▪ Hash table’s load factor
Contents

• Basics
  • Hash Functions, Hash Codes, Index for the Hash Table
  • Resolving Collisions (Separate Chaining)
  • Efficiency of Hashing (Load Factor, Cost of Separate Chaining)

• A Dictionary Implementation Using Hashing
  • Entries in the Hash Table, Data Fields and Constructors
  • Methods ($\text{getValue}$, $\text{remove}$, $\text{add}$), Iterators

• Java Class Library:
  • method $\text{hashCode}$ for objects used as search keys
  • Classes $\text{HashMap}$, $\text{HashSet}$
What Is Hash?

• What is hash food?
  ▪ “chop and mix” (hash function)
  ▪ a traditional cheap and quick dish (Wikipedia)
  ▪ Ex. Corned beef hash, hash brown, etc.

• What is hash table in Computer Science?
  ▪ Method to locate data quickly
  ▪ Ideally has O(1) search times
  ▪ But not suitable for range query (data traversal in order)

• What is hashing?
  ▪ Comparison with array
  ▪ Hashing determines index using only a search key

• Hash function locates correct item in hash table
  ▪ Maps or “hashes to” entry
Typical Hashing

- **Hashing Algorithm** or hash function will
  - Convert search key to integer called hash code.
  - Compress hash code into range of indices for hash table.

- **Steps**: Typical hash functions has two steps
  - Search key $\rightarrow$ hash code
  - Compress hash code to smaller range

- **Example**: $\text{index} = \text{abs} ( \text{key}.\text{hashCode}() \% \text{hashTable}.\text{length} )$

Figure 21-1: Given a key, a hash function finds its index in a hash table.
Java Hash Codes

• Hash codes in Java

```java
Object[] x = {"a", "A", new Integer(10), "a0", "al", new Float(10.0)};
for ( int i = 0; i < x.length; i++)
    System.out.println( x[i].hashCode());
```

• The output is:

97
65
10
3055
3056
1092616192
Hash Code for Different Types

- For `int`: Use the value
- For `byte`, `short`, or `char`: Cast into an `int`
- Other primitive types
  - Manipulate internal binary representations
- Strings:
  - Chop into individual characters:
    - Assign integer to each character in string
    - E.g., use 1 – 26 for ‘a’ to ‘z’ (or Unicode integer)
  - Mix: sum or weighted sum
    - Simple: sum the integers of the characters for the hash code
    - Better: Multiply Unicode value of each character by factor based on position

\[
\sum_{i=0}^{n-1} u_i g^{n-i-1}
\]

- **Ex.** Determine value of `g` given following hashcodes:
  - \( H(0) = 0, \ H(1) = 1, \ H(\text{“a0”}) = 3055, \ H(\text{“a1”}) = 3056 \)
Compressing a Hash Code

• Scale an integer by using Java % operator
  ▪ For code c and size of table n, use c % n
  ▪ Result is remainder of division

• If n is prime, provides values distributed in range 0 to n – 1

• Q? What are benefits of compression?
  ▪ Reduce storage waste for empty table entries

• Leading Q? What is a downside of compression?

```java
private int getHashIndex(K key) {
    int hashIndex = key.hashCode() % hashTable.length;
    if (hashIndex < 0)
        hashIndex = hashIndex + hashTable.length;
    return hashIndex;
} // end getHashIndex
```
Challenges in Designing Hashing Functions

- **Challenge:** Typical hash functions *not perfect*
  - Can allow more than one search key to map into single index
  - Causes “collision” in hash table
  - Compression may increase collisions

- **Dealing with Collisions**
  - Open Addressing with Linear/Quadratic Probing or double hashing
  - Separate Chaining

Figure 21-2 A **collision** caused by the hash function $h$
Figure 22-4 The average number of comparisons required by a search of the hash table versus the load factor $\lambda$ for four collision resolution techniques when the search is (a) successful; (b) unsuccessful.
Contents

• Basics
  ▪ Hash Functions (Hash Codes, Index for the Hash Table)
  ▪ Resolving Collisions (Separate Chaining)
  ▪ Efficiency of Hashing (Load Factor, Cost of Separate Chaining)

• A Dictionary Implementation Using Hashing
  ▪ Entries in the Hash Table, Data Fields and Constructors
  ▪ Methods (getValue, remove, add), Iterators

• Java Class Library:
  ▪ method hashCode for objects used as search keys
  ▪ Classes HashMap, HashSet
Dictionary Implementation w/ Hashing

- Key Design Decision: Collision Handling
  - Choice 1: Separate Chaining
  - Choice 2: Open Addressing
- Each location of hash table can represent multiple values
  - Called a “bucket”
- Other Design Decisions: Hash Table Entries a.k.a buckets
  - Number of Buckets
  - Bucket Capacity
  - Bucket representation
    - List (sorted or not)
Dictionary Implementation w/ Separate Chaining

- Collision Handling with Separate Chaining
- Each location of hash table can represent multiple values
  - Called a “bucket”
- To add, hash to bucket, insert data in first available slot there
- To retrieve, hash to bucket, traverse bucket contents
- To delete, hash to bucket, remove item
- Bucket representation
  - List (sorted or not) using Chain of linked nodes
  - Arrays avoided due to storage overhead

Figure 21-9 A hash table for use with separate chaining; each bucket is a chain of linked nodes
FIGURE 21-10 Where to insert new entry into linked bucket when integer search keys are (a) unsorted and possibly duplicate; (c) sorted and distinct

Q? What about unsorted and distinct?
Performance: Separate Chaining

- **Load factor is**
  \[ \lambda = \frac{\text{Number of entries in the dictionary}}{\text{Number of chains}} \]

- **Average number of comparisons**
  - Unsuccessful search \( \lambda \)
  - Successful search \( 1 + \frac{\lambda}{2} \)
Figure 22-3 The average number of comparisons required by a search of the hash table for given values of the load factor $\lambda$ when using separate chaining.

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>Unsuccessful Search</th>
<th>Successful Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.1</td>
<td>1.1</td>
</tr>
<tr>
<td>0.3</td>
<td>0.3</td>
<td>1.2</td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>1.3</td>
</tr>
<tr>
<td>0.7</td>
<td>0.7</td>
<td>1.4</td>
</tr>
<tr>
<td>0.9</td>
<td>0.9</td>
<td>1.5</td>
</tr>
<tr>
<td>1.1</td>
<td>1.1</td>
<td>1.6</td>
</tr>
<tr>
<td>1.3</td>
<td>1.3</td>
<td>1.7</td>
</tr>
<tr>
<td>1.5</td>
<td>1.5</td>
<td>1.8</td>
</tr>
<tr>
<td>1.7</td>
<td>1.7</td>
<td>1.9</td>
</tr>
<tr>
<td>1.9</td>
<td>1.9</td>
<td>2.0</td>
</tr>
<tr>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>
A Hashing Based Dictionary

Exercise: Determine average-case for add(), remove(), contains(), getKeyIterator() with retrieval of all pairs, ...

Assume
- n = number of entries,
- b = number of buckets
- E = load-factor, i.e., \((n / b)\)

Average case efficiencies of dictionary operations for sorted linked implementation
- Addition \(O(1 + e)\)
- Removal \(O(1 + e)\)
- Retrieval \(O(1 + e)\)
- Traversal \(O(n)\)

Q? Why is add() \(O(1 + e)\)?

// Listing 19-1, pp. 476-477, Textbook
import java.util.Iterator;
public interface DictionaryInterface < K, V > {
    public V add (K key, V value);
    public V remove (K key);
    public V getValue (K key);
    public boolean contains (K key);
    public Iterator < K > getKeyIterator ();
    public Iterator < V > getValueIterator ();
    public boolean isEmpty ();
    public int getSize ();
    public void clear ();
}
Another Dictionary Implementation

- Book implement collision handing with linear probing
  - Other open addressing strategies involve few changes
- Note source code of class `HashedDirectory`, [Listing 22-1](#)

![Hash table and one of its entry objects](#)

**Figure 22-5** A hash table and one of its entry objects

**FIGURE 22-6** A hash table containing dictionary entries, removed entries, and **null** values

- Blue = current entry
- Light gray = removed entry
- Dark gray = **null**

Copyright ©2012 by Pearson Education, Inc. All rights reserved
Contents

• Basics
  ▪ Hash Functions (Hash Codes, Index for the Hash Table)
  ▪ Resolving Collisions (Separate Chaining)
  ▪ Efficiency of Hashing (Load Factor, Cost of Separate Chaining)

• A Dictionary Implementation Using Hashing
  ▪ Entries in the Hash Table, Data Fields and Constructors
  ▪ Methods (getValue, remove, add), Iterators

• Java Class Library:
  ▪ method hashCode for objects used as search keys
  ▪ Classes HashMap, HashSet
Computing Hash Codes

- Must override Java `Object` method `hashCode`
- Guidelines for new `hashCode` method
  - If class overrides method `equals`, it should override `hashCode`.
  - If method `equals` considers two objects equal, `hashCode` must return same value for both objects.
  - If you call an object’s `hashCode` more than once during execution of a program, and if object’s data remains same during this time, `hashCode` must return the same value.
  - Object’s hash code during one execution of a program can differ from its hash code during another execution of the same program.
Java Class Library: The Class `HashMap`

- Standard package `java.util` contains the class `HashMap<K, V>`
- Table is a collection of buckets
- Constructors
  - `public HashMap()`
  - `public HashMap(int initialSize)`
  - `public HashMap(int initialSize, float maxLoadFactor)`
  - `public HashMap(Map<? extends K, ? extends V> table)`
Java Class Library: The Class **HashMap**

- **Design**
  - Max $\lambda = 0.75$
  - Avoid necessity of rehashing by setting

\[
\text{Number of buckets} > \frac{\text{Max entries in dictionary}}{\lambda_{\text{max}}}
\]
Java Class Library: The Class `HashSet`

- Implements the interface `java.util.Set` of Chapter 1
  - Uses an instance of the class `HashMap` to contain entries in a set
- Constructors
  - `public HashSet()`
  - `public HashSet(int initialCapacity)`
  - `public HashSet(int initialCapacity, float loadFactor)`
Hashing Use Case Beyond Dictionary Implementation

- Databases: Detect duplicate records in a large database file, Bloom filters,
- Bio-informatics: Find similar stretches in DNA sequences (Robin-Karp algorithm)
- Graphics, Computational Geometry: Proximity queries (Geometric Hashing)

Related Concepts: check-sum, fingerprint, randomizing function, …
- Cyber-Security: Detect tampering of a datafile or software
- Operating Systems: Password checking