Dictionaries

- Collection of pairs.
  - (key, element)
  - Pairs have different keys.
- Operations.
  - get(theKey)
  - put(theKey, theElement)
  - remove(theKey)

Application

- Collection of student records in this class.
  - (key, element) = (student name, linear list of assignment and exam scores)
  - All keys are distinct.
- Get the element whose key is John Adams.
- Update the element whose key is Diana Ross.
  - put() implemented as update when there is already a pair with the given key.
  - remove() followed by put().
Dictionary With Duplicates

- Keys are not required to be distinct.
- Word dictionary.
  - Pairs are of the form \((\text{word, meaning})\).
  - May have two or more entries for the same word.
    - (bolt, a threaded pin)
    - (bolt, a crash of thunder)
    - (bolt, to shoot forth suddenly)
    - (bolt, a gulp)
    - (bolt, a standard roll of cloth)
    - etc.

Represent As A Linear List

- \(L = (e_0, e_1, e_2, e_3, \ldots, e_{n-1})\)
- Each \(e_i\) is a pair \((\text{key, element})\).
- 5-pair dictionary \(D = (a, b, c, d, e)\).
  - \(a = (a\text{Key, aElement}), b = (b\text{Key, bElement})\), etc.
- Array or linked representation.
### Array Representation

| a | b | c | d | e |

- **get(theKey)**
  - $O(\text{size})$ time
- **put(theKey, theElement)**
  - $O(\text{size})$ time to verify duplicate, $O(1)$ to add at right end.
- **remove(theKey)**
  - $O(\text{size})$ time.

### Sorted Array

| A | B | C | D | E |

- elements are in ascending order of key.
- **get(theKey)**
  - $O(\log \text{ size})$ time
- **put(theKey, theElement)**
  - $O(\log \text{ size})$ time to verify duplicate, $O(\text{size})$ to add.
- **remove(theKey)**
  - $O(\text{size})$ time.
**Unsorted Chain**

- get(theKey)
  - $O(\text{size})$ time
- put(theKey, theElement)
  - $O(\text{size})$ time to verify duplicate, $O(1)$ to add at left end.
- remove(theKey)
  - $O(\text{size})$ time.

**Sorted Chain**

- Elements are in ascending order of Key.
- get(theKey)
  - $O(\text{size})$ time
- put(theKey, theElement)
  - $O(\text{size})$ time to verify duplicate, $O(1)$ to put at proper place.
**Sorted Chain**

- Elements are in ascending order of Key.
- remove(theKey)
  - O(size) time.

**Skip Lists**

- Worst-case time for get, put, and remove is O(size).
- Expected time is O(log size).
- We’ll skip skip lists.
Hash Tables

- Worst-case time for get, put, and remove is $O(\text{size})$.
- Expected time is $O(1)$.

Ideal Hashing

- Uses a 1D array (or table) $\text{table}[0:b-1]$.
  - Each position of this array is a bucket.
  - A bucket can normally hold only one dictionary pair.
- Uses a hash function $f$ that converts each key $k$ into an index in the range $[0, b-1]$.
  - $f(k)$ is the home bucket for key $k$.
- Every dictionary pair (key, element) is stored in its home bucket $\text{table}[f[key]]$. 
Ideal Hashing Example

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is table[0:7], h = 8.
- Hash function is key/11.
- Pairs are stored in table as below:

<table>
<thead>
<tr>
<th>table[0:7]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3,d)</td>
<td>(22,a)</td>
<td>(33,c)</td>
<td>(73,e)</td>
<td>(85,f)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- get, put, and remove take O(1) time.

What Can Go Wrong?

- Where does (26,g) go?
- Keys that have the same home bucket are synonyms.
  - 22 and 26 are synonyms with respect to the hash function that is in use.
- The home bucket for (26,g) is already occupied.
What Can Go Wrong?

| (3,d) | (22,a) | (33,c) | (73,e) | (85,f) |

- A **collision** occurs when the home bucket for a new pair is occupied by a pair with a different key.
- An **overflow** occurs when there is no space in the home bucket for the new pair.
- When a bucket can hold only one pair, collisions and overflows occur together.
- Need a method to handle overflows.

Hash Table Issues

- Choice of hash function.
- Overflow handling method.
- Size (number of buckets) of hash table.
Hash Functions

- Two parts:
  - Convert key into an integer in case the key is not an integer.
    - Done by the method `hashCode()`.
  - Map an integer into a home bucket.
    - $f(k)$ is an integer in the range $[0, b-1]$, where $b$ is the number of buckets in the table.

String To Integer

- Each Java character is 2 bytes long.
- An `int` is 4 bytes.
- A 2 character string $s$ may be converted into a unique 4 byte `int` using the code:
  ```java
  int answer = s.charAt(0);
  answer = (answer << 16) + s.charAt(1);
  ```
- Strings that are longer than 2 characters do not have a unique `int` representation.
public static int integer(String s) {
    int length = s.length();
    // number of characters in s
    int answer = 0;
    if (length % 2 == 1) {
        // length is odd
        answer = s.charAt(length - 1);
        length--;
    }

    // length is now even
    for (int i = 0; i < length; i += 2) {
        // do two characters at a time
        answer += s.charAt(i);
        answer += ((int) s.charAt(i + 1)) << 16;
    }
    return (answer < 0) ? -answer : answer;
}
Map Into A Home Bucket

- Most common method is by division.
  
  ```java
  homeBucket = Math.abs(theKey.hashCode()) % divisor;
  ```
- divisor equals number of buckets b.
- 0 <= homeBucket < divisor = b

Uniform Hash Function

- Let keySpace be the set of all possible keys.
- A uniform hash function maps the keys in keySpace into buckets such that approximately the same number of keys get mapped into each bucket.
Uniform Hash Function

• Equivalently, the probability that a randomly selected key has bucket $i$ as its home bucket is $1/b$, $0 \leq i < b$.

• A uniform hash function minimizes the likelihood of an overflow when keys are selected at random.

<table>
<thead>
<tr>
<th>(3,d)</th>
<th>(22,a)</th>
<th>(33,c)</th>
<th>(73,e)</th>
<th>(85,f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
</tr>
</tbody>
</table>

Hashing By Division

• $\text{keySpace} = \text{all ints}$.  

• For every $b$, the number of ints that get mapped (hashed) into bucket $i$ is approximately $2^{32}/b$.

• Therefore, the division method results in a uniform hash function when $\text{keySpace} = \text{all ints}$.

• In practice, keys tend to be correlated.

• So, the choice of the divisor $b$ affects the distribution of home buckets.
Selecting The Divisor

- Because of this correlation, applications tend to have a bias towards keys that map into odd integers (or into even ones).
- When the divisor is an even number, odd integers hash into odd home buckets and even integers into even home buckets.
  - $20 \div 14 = 6$, $30 \div 14 = 2$, $8 \div 14 = 8$
  - $15 \div 14 = 1$, $3 \div 14 = 3$, $23 \div 14 = 9$
- The bias in the keys results in a bias toward either the odd or even home buckets.

Selecting The Divisor

- When the divisor is an odd number, odd (even) integers may hash into any home.
  - $20 \div 15 = 5$, $30 \div 15 = 0$, $8 \div 15 = 8$
  - $15 \div 15 = 0$, $3 \div 15 = 3$, $23 \div 15 = 8$
- The bias in the keys does not result in a bias toward either the odd or even home buckets.
- Better chance of uniformly distributed home buckets.
- So do not use an even divisor.
Selecting The Divisor

• Similar biased distribution of home buckets is seen, in practice, when the divisor is a multiple of prime numbers such as \(3, 5, 7, \ldots\).
• The effect of each prime divisor \(p\) of \(b\) decreases as \(p\) gets larger.
• Ideally, choose \(b\) so that it is a prime number.
• Alternatively, choose \(b\) so that it has no prime factor smaller than 20.

Java.util.HashTable

• Simply uses a divisor that is an odd number.
• This simplifies implementation because we must be able to resize the hash table as more pairs are put into the dictionary.
  • Array doubling, for example, requires you to go from a 1D array table whose length is \(b\) (which is odd) to an array whose length is \(2b+1\) (which is also odd).