Csci 1933: Introduction to Algorithms and Data Structures

Fall 2017

Overview

- **Last time:**
  - Started on dictionaries

- **Today:**
  - More dictionaries

- **Next time:**
  - Dictionaries, start on graphs (Project #4)

- **Announcements:**
  - Midterm #2 THIS week: Friday, 11/17 (in class)

Midterm #2 guidelines

- **Friday, 11/17, 2:30pm** (be here on time!)

- Similar format as last time:
  mix of questions that ask you to interpret code, fill in short pieces of missing code, and write larger sections of code to accomplish a small problem, analyze complexity of code

- Closed book, closed notes, and no electronic devices will be allowed. We will allow you one 8.5”x11” sheet with notes (only handwritten), both front and back.

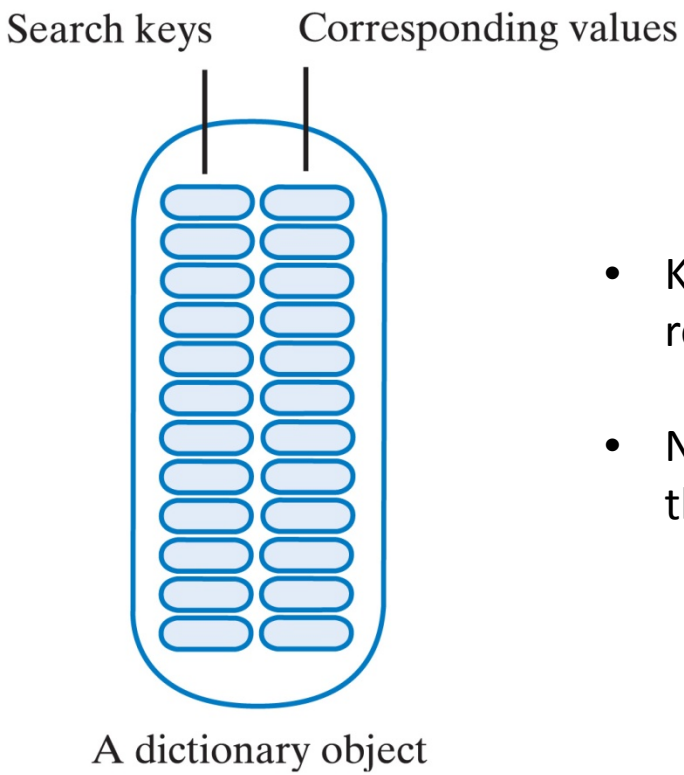
- List of topics: see PDF in Week 10 folder on Moodle page

- Lab 10 will also be a midterm review (attendance required as usual)– Lab 10 is already posted, solutions will be posted tonight

Dictionary abstract data type (review)

Specifications for the ADT Dictionary

"key-value" pair



- Keys are unique! (not required of values)
- No inherent ordering of the key-value pairs

An instance of an ADT dictionary has pairs of search keys and corresponding values

Dictionary examples

Map (from → to)

- User id (Integer) → FacebookAccount
- Name (String) → Telephone number
- Word (String) → Definition
- User id (Integer) → List of Facebook accounts (e.g. friends)

Note: key type and value types are flexible

Specifications for the ADT Dictionary

- Data
  - Pairs of objects (key, value)
  - Number of pairs in the collection
- Operations
  - add(key,value)
  - remove(key)
  - getValue(key)
  - contains(key)
  - getSize()
  - isFull()
  - isEmpty()
  - getKeyIterator()

Formal definition of a Dictionary interface

```
/** A dictionary with distinct search keys. */
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 boolean isEmpty();

    public boolean isFull();

    public int getSize();

    public void clear();

    public Iterator<K> getKeyIterator();

    public Iterator<K> getValueIterator();

} // end DictionaryInterface
```

(See Carrano, Ch. 19 for more details.)

Dictionary exercise #2

- Assume you're given a genome sequence in a String

```
String sequence = "ATACGGAATTCCTATACGGGATTATACCCG…")
```

- You need to count the number of occurrences of all possible length-4 sequences

  e.g. ATAC occurs 3 times

  ATACGGAATTCCTATACGGGATTATACCCG…

- Write pseudo-code that uses a Dictionary to do this

Dictionary exercise #2

Dictionary exercise #2

```
DictionaryInterface<String,Integer> patterns = new
   ArrayDictionary<String, Integer>();

for(int i=0; i <= sequence.length-4; i++) {
   currString = sequence.substring(i,i+4)

   if(patterns.contains(currString))
        patterns.add(currString,patterns.get(currString)+1);
   else
        patterns.add(currString,1);
}

//now just get all of the keys from the dictionary, ask for
   //corresponding values one-by-one
Iterator<String> keys = patterns.getKeyIterator();
while(keys.hasNext()) {
   String curr = keys.next();
   System.out.println(curr+":"+patterns.get(curr));
}
```

Dictionaries in Java: the built-in Map interface

```
public interface Map<K,V> {
    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size(); boolean isEmpty();

    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear(); // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Java built-in class that implement the Map interface: HashMap, TreeMap, and LinkedHashMap

Dictionaries in Java: Map interface

Interface Map<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:

Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Dictionaries in Java: the built-in Map interface (an example)

```
import java.util.*;
public class Freq {

    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();

        for (int i=0; i < args.length; i++) {
          if(m.containsKey(args[i])) {
            m.put(args[i], m.get(args[i])+1);
          else
            m.put(args[i],1);
        }

        System.out.println(m.size() + " distinct words.");

        Iterator<String> it = m.keySet().iterator();
        while(it.hasNext()) {
          String curr = it.next();// do something with key here
          System.out.println(curr+":"+m.get(curr));
      }
    }
}
```

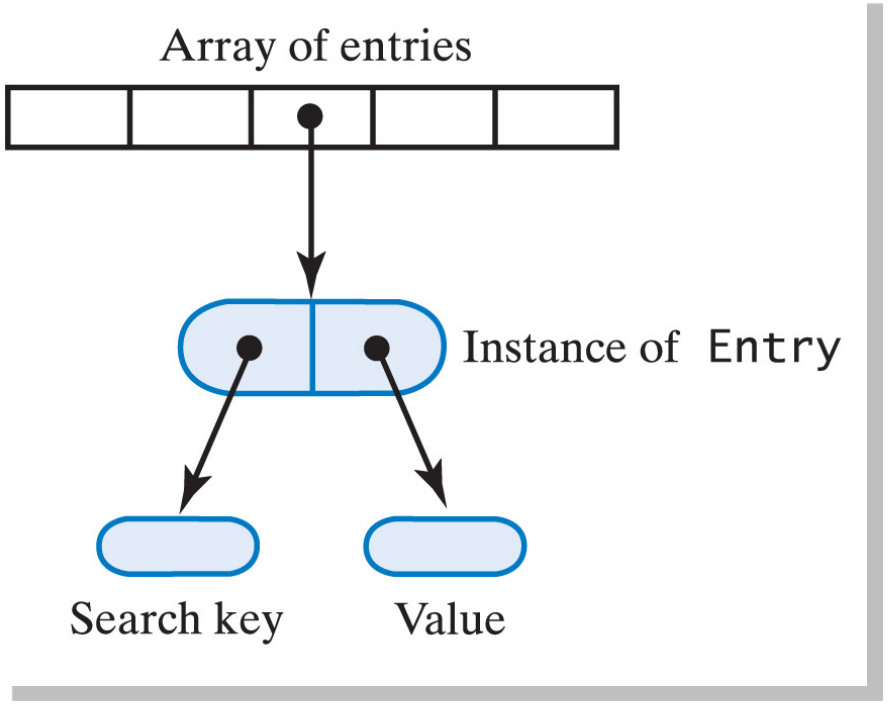
http://java.sun.com/docs/books/tutorial/collections/interfaces/map.html

Implementation of a Dictionary (Map)

- We'll discuss:
  - Array-based implementation
  - Operations:
    - Addition
    - Removal
    - Retrieval
    - Traversal (iterate over all elements)
  - Implementation logic for each operation + the complexity

Array-Based Dictionary Implementation

One possible way to use arrays to represent dictionary entries: an array of objects that encapsulates each search key and corresponding value



Implementation of a Dictionary (Map)

Array-based implementation complexity

– Operations:

- Addition - O(n)
- Removal - O(n)
- Retrieval - O(n)
- Traversal (iterate over all elements) - O(n)

Pop quiz: assume we're using a key type that implements Comparable– can we improve the array implementation at all?

Implementation of a Dictionary (Map)

Array-based implementation complexity

– Operations:

- Addition - O(n)
- Removal - O(n)
- Retrieval - O(n)
- Traversal (iterate over all elements) - O(n)

Pop quiz: assume we're using a key type that implements Comparable– can we improve the array implementation at all?

Answer: let's maintain a sorted array of entries!

Search on a sorted array

Goal: find entry corresponding to Integer key 8

Rule: you can only check one element at a time, each check costs you 1 operation

Note: this problem is equivalent to the “guess a number” game

Unsorted version:

10	18	5	26	21	2	12	15	4	**8**	24	7

Sorted version:

2	4	5	7	**8**	10	12	15	18	21	24	26

- What’s a “good” search algorithm for the sorted case?

Search on a sorted array

Sorted version:

2	4	5	7	**8**	10	12	15	18	21	24	26

- Binary search algorithm
  - Basic idea:
    - Pick the element in the middle of the array
    - Check if the element you're searching for is greater than *middle*
    - If yes, throw away left half of the array, repeat the process on the right half
    - If no, throw away right half of the array, repeat the process on the left half

Search on a sorted array

Sorted version:

2	4	5	7	**8**	10	12	15	18	21	24	26

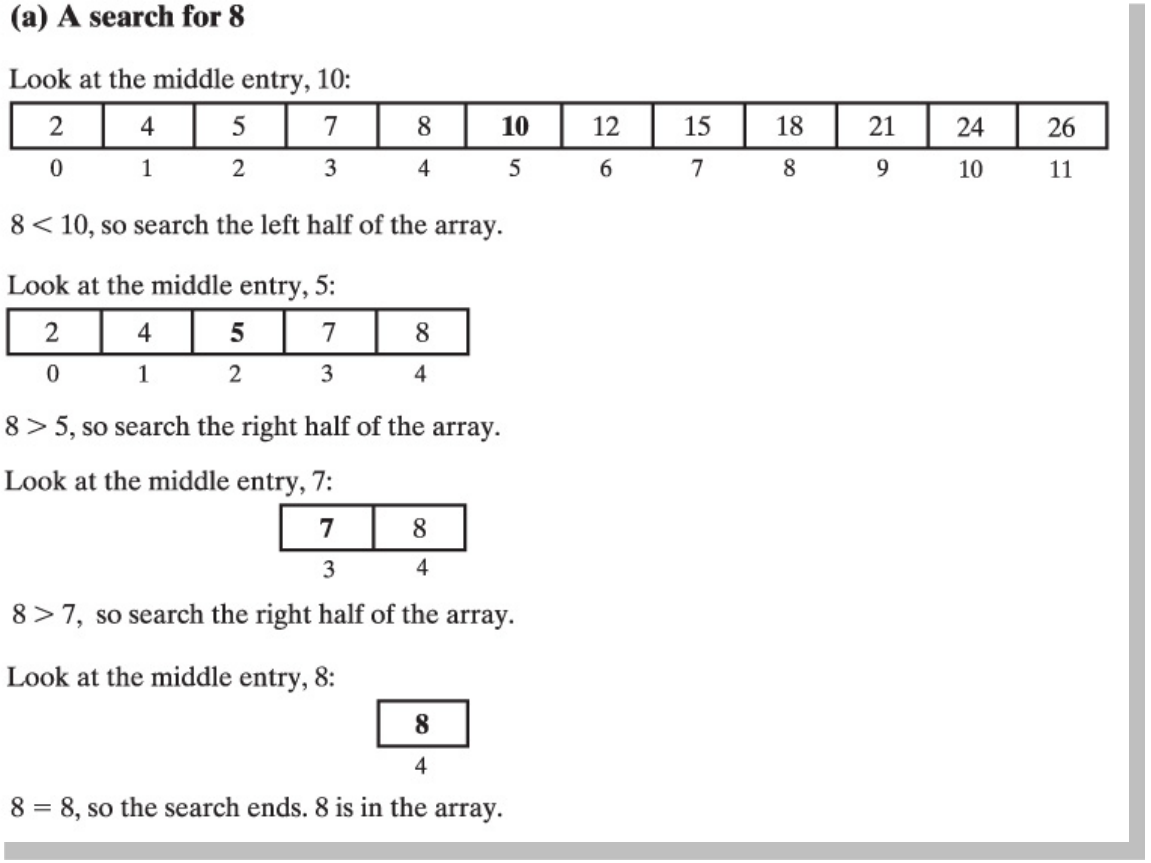
- Binary search algorithm pseudo-code

  (recursive version)

```
Algorithm binarySearch(array, first, last, desiredItem)
  mid = (first+last)/2  //get middle
  if(desiredItem == array[mid])
      return mid
  else if(desiredItem > a[mid])
      return binarySearch(a,mid+1,last,desiredItem)
  else if(desiredItem < a[mid])
      return binarySearch(a,first,mid-1,desiredItem)
```

Binary Search of Sorted Array: example

A recursive binary search of a sorted array that (a) finds its target;



(a) A search for 8

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

$8 < 10$, so search the left half of the array.

Look at the middle entry, 5:

2	4	**5**	7	8
0	1	2	3	4

$8 > 5$, so search the right half of the array.

Look at the middle entry, 7:

7	8
3	4

$8 > 7$, so search the right half of the array.

Look at the middle entry, 8:

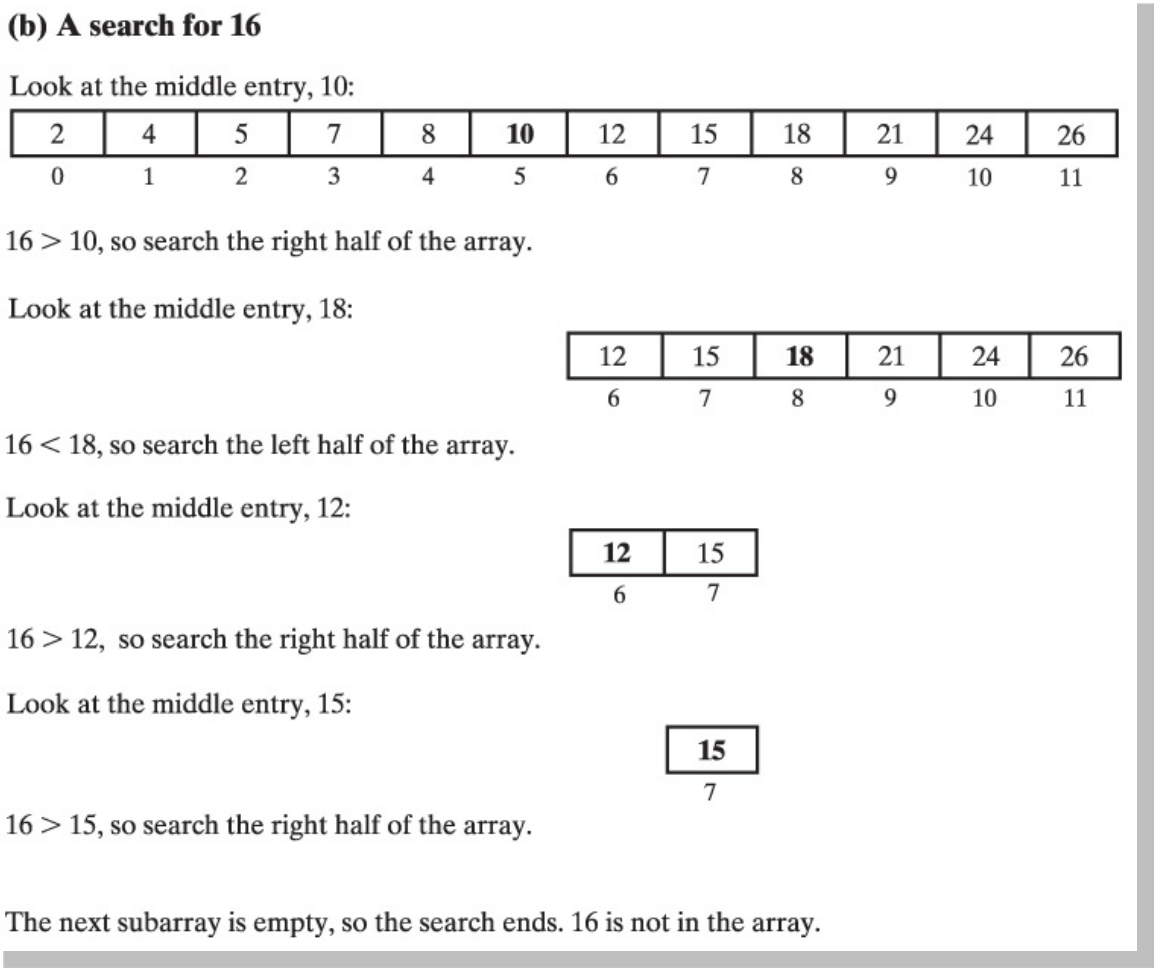
8
4

$8 = 8$, so the search ends. 8 is in the array.

Carrano and Henry, Data Structures and Abstractions with Java.

Binary Search of Sorted Array: example

A recursive binary search of a sorted array that (b) does not find its target.



(b) A search for 16

Look at the middle entry, 10:

2	4	5	7	8	**10**	12	15	18	21	24	26
0	1	2	3	4	5	6	7	8	9	10	11

16 > 10, so search the right half of the array.

Look at the middle entry, 18:

12	15	**18**	21	24	26
6	7	8	9	10	11

16 < 18, so search the left half of the array.

Look at the middle entry, 12:

12	15
6	7

16 > 12, so search the right half of the array.

Look at the middle entry, 15:

15
7

16 > 15, so search the right half of the array.

The next subarray is empty, so the search ends. 16 is not in the array.

Carrano and Henry, Data Structures and Abstractions with Java.

Efficiency of a Binary Search?

- Best case
- Worst case
- Average case

Efficiency of a Binary Search?

- Best case — Locate desired item first- O(1)
- Worst case — Can’t find/find at last element - O(log n)
- Average case — O(log n)

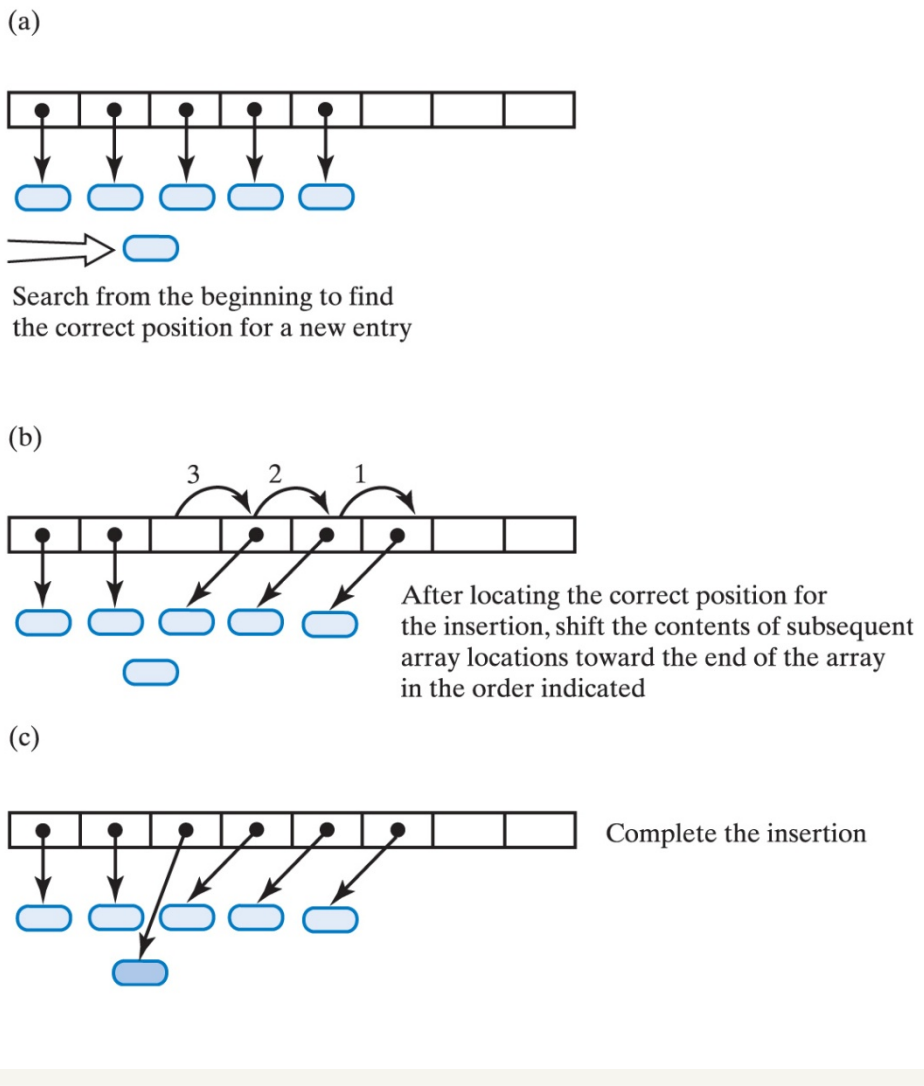
Java Class Library: The Method **`binarySearch`**

- The class Arrays in java.util defines versions of a static method with following specification:

```
/** Task: Searches an entire array for a given item.
* @param array the array to be searched
* @param desiredItem the item to be found in the array
* @return index of the array element that equals desiredItem;
* otherwise returns -belongsAt-1, where belongsAt is
* the index of the array element that should contain
* desiredItem */
public static int binarySearch
         (type[] array, type desiredItem);
```

Sorted Array-Based Dictionary

Adding an entry to a sorted array-based dictionary:
(a) search;
(b) make room;
(c) insert.



Sorted Array-Based Dictionary Implementation

- Sorted worst-case efficiencies
  - Addition
  - Removal
  - Retrieval
  - Traversal

Compare to (unsorted):
Addition - O(n)
Removal- O(n)
Retrieval- O(n)
Traversal- O(n)

Sorted Array-Based Dictionary Implementation

- Sorted worst-case efficiencies
  - Addition O(n)
  - Removal O(n)
  - Retrieval O(log n)
  - Traversal O(n)

Compare to (unsorted):
Addition - O(n)
Removal- O(n)
Retrieval- O(n)
Traversal- O(n)

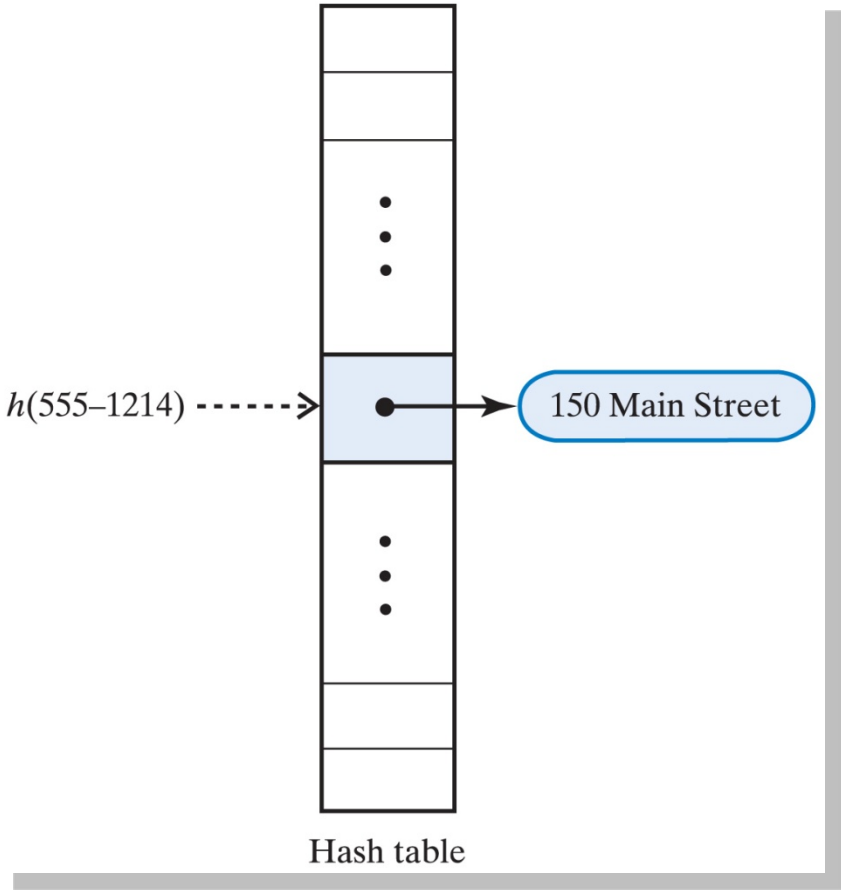
Another dictionary implementation: hashtable

Basic idea:

- Develop a hash function that maps the key into an index

  e.g. the string “555-1214” maps to index 23 (sum of all chars)
- The index refers to a location in an array

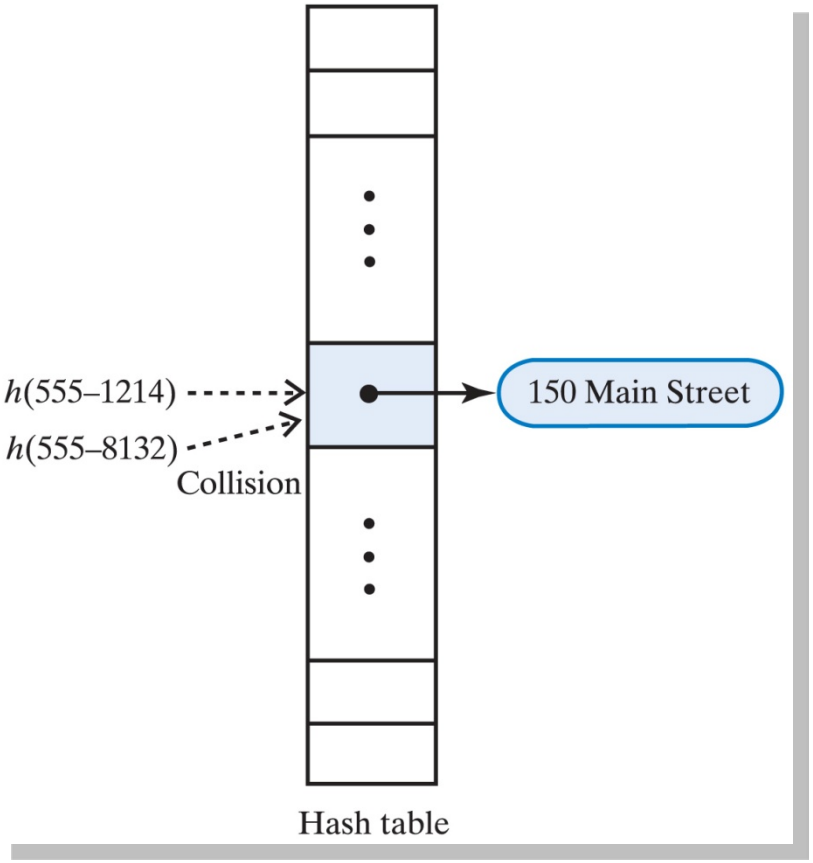
→No search on the array is necessary!



Picking a good hash function

h(key) → index

- General characteristics of a good hash function
  - Minimize collisions
  - Distribute entries uniformly throughout the hash table
  - Be fast to compute



An example hash function for Strings

- An example hash code for a string, **s**

$$s = "c_0 c_1 \ldots c_{n-1}"$$

$$h(s) = u_0 g^{n-1} + u_1 g^{n-2} + \ldots u_{n-2} g + u_{n-1}$$

$$u_i = unicode(c_i)$$

(A=65, Z=90)

in Java String class hashCode method: g=31

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

To translate hashcode into Key index:
hashCode()%hashTable.length

Resolving hash table collisions

- Collision: two different keys map to the same hash function value (same index in the array)
- Even a good hash function will cause collisions

- Options for handling collisions
  - Use a nearby location in the table (e.g. neighbor index, …) (called “open addressing”)
  - Change the structure of the hash table so that each array location can represent multiple values (called “separate chaining”)

Collision-handling hashtable implementation: separate chaining

- Alter the structure of the hash table
- Each location can represent multiple values
  - Each location called a bucket
- Bucket could be
  - List
  - Sorted list
  - Chain of linked nodes
  - Array
  - Vector