SETS AND MAPS

Chapter 7

Chapter Objectives

- To understand the Java Map and Set interfaces and how to use them
- To learn about hash coding and its use to facilitate efficient insertion, removal, and search
- To study two forms of hash tables—open addressing and chaining—and to understand their relative benefits and performance trade-offs

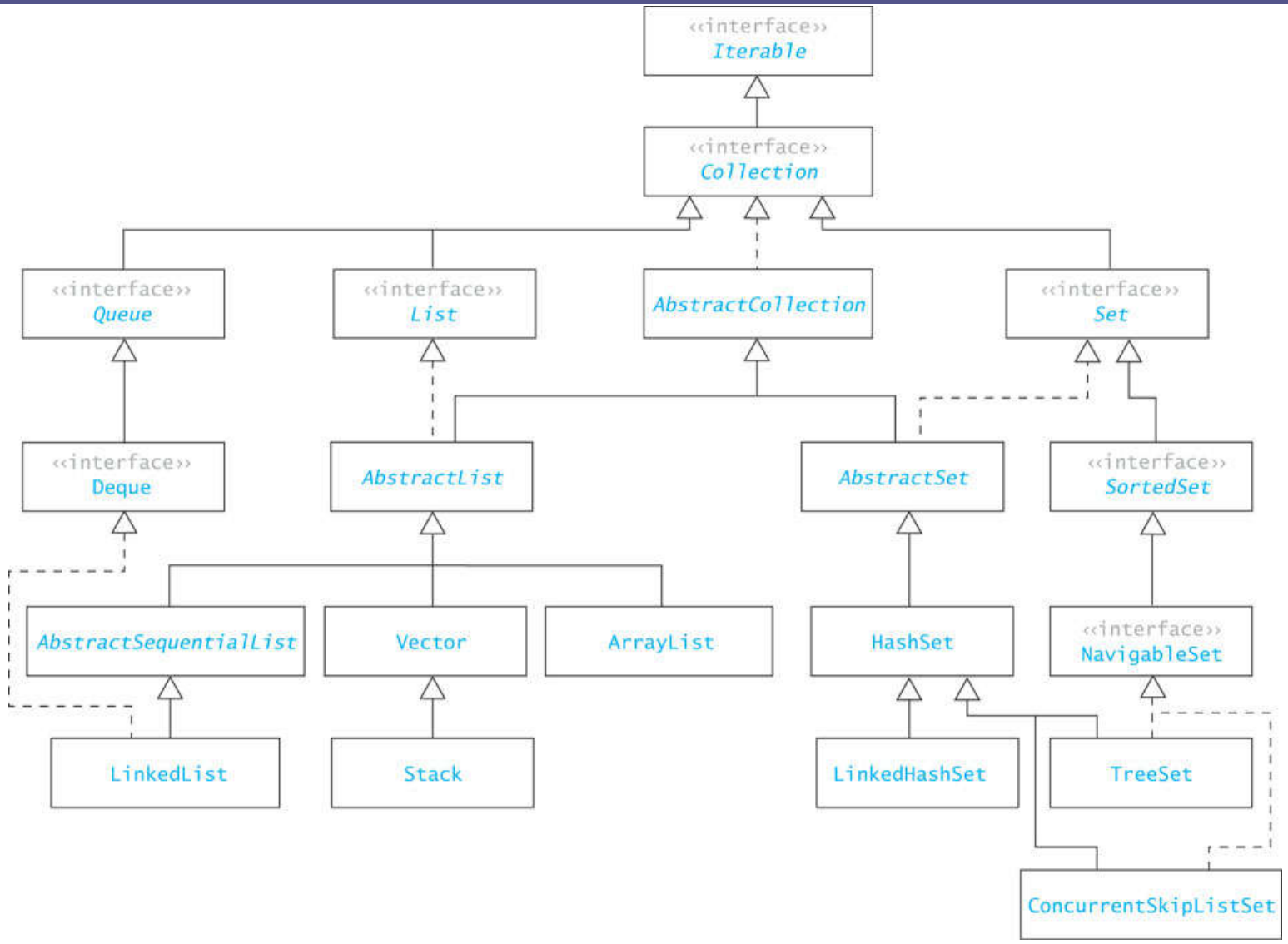
Chapter Objectives (cont.)

- To learn how to implement both hash table forms
- To be introduced to the implementation of `Map`s and `Set`s
- To see how two earlier applications can be implemented more easily using `Map` objects for data storage

Introduction

- We learned about part of the Java Collection Framework in Chapter 2 (`ArrayList` and `LinkedList`)
- The classes that implement the `List` interface are all *indexed* collections
  - An index or subscript is associated with each element
  - The element's index often reflects the relative order of its insertion into the list
  - Searching for a particular value in a list is generally O(*n*)
  - An exception is a binary search of a sorted object, which is O(log *n*)

The `Collection` Framework



Introduction (cont.)

- In this chapter, we consider another part of the `Collection` hierarchy: the `Set` interface and the classes that implement it
- `Set` objects
  - are not indexed
  - do not reveal the order of insertion of items
  - enable efficient search and retrieval of information
  - allow removal of elements without moving other elements around

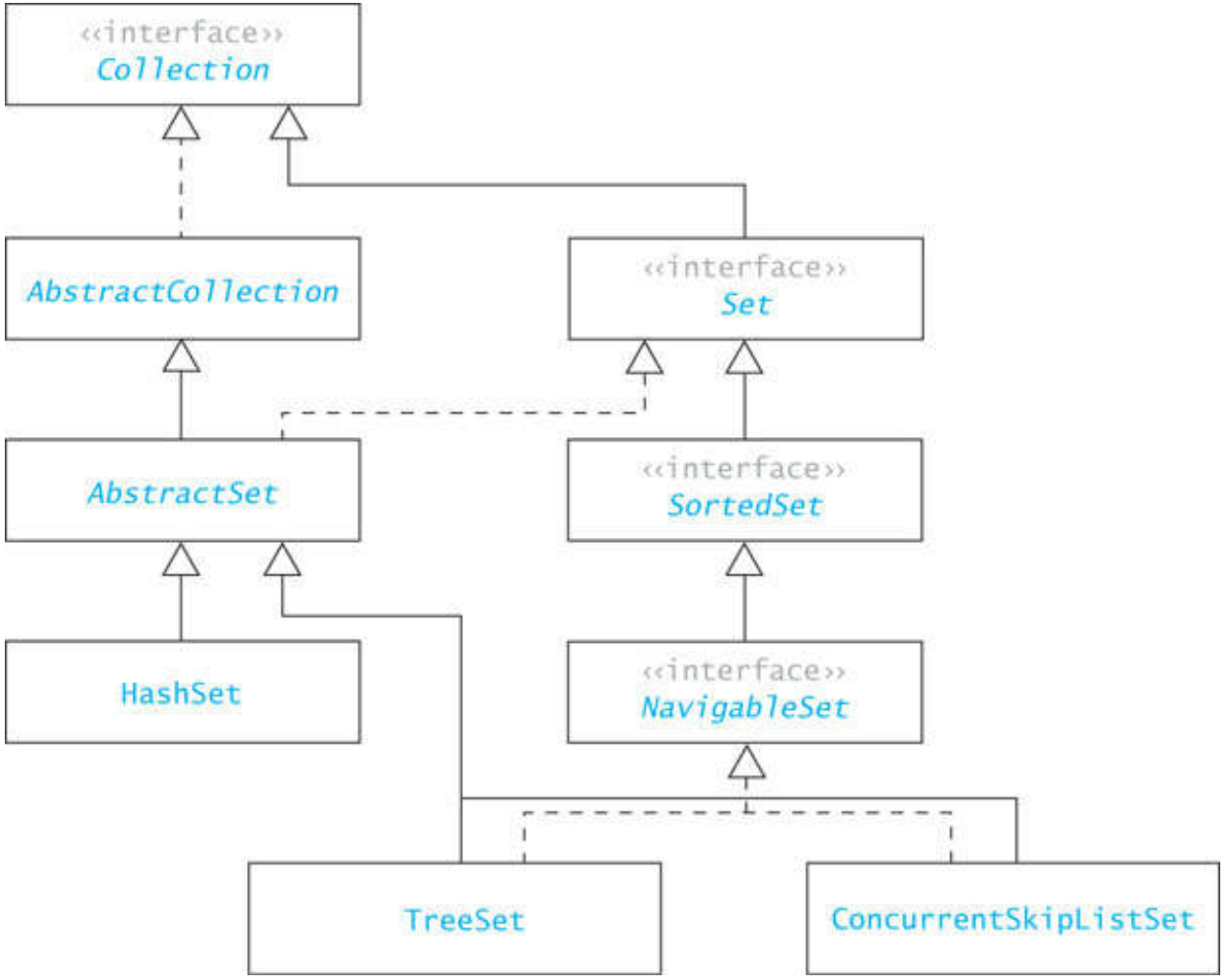
Introduction (cont.)

- Relative to a `Set`, `Map` objects provide efficient search and retrieval of entries that contain pairs of objects (a unique key and the information)
- Hash tables (implemented by a `Map` or `Set`) store objects at arbitrary locations and offer an average constant time for insertion, removal, and searching

Sets and the `Set` Interface

Section 7.1

Sets and the `Set` Interface



The `Set` Abstraction

- A set is a collection that contains no duplicate elements and at most one `null` element
  - adding `"apples"` to the set `{"apples", "oranges", "pineapples"}` results in the same set (no change)
- Operations on sets include:
  - testing for membership
  - adding elements
  - removing elements
  - union $A \cup B$
  - intersection $A \cap B$
  - difference $A - B$
  - subset $A \subset B$

The `Set` Abstraction(cont.)

- The union of two sets A, B is a set whose elements belong either to A or B or to both A and B.

  Example: {1, 3, 5, 7} ∪ {2, 3, 4, 5} is {1, 2, 3, 4, 5, 7}
- The intersection of sets A, B is the set whose elements belong to both A and B.

  Example: {1, 3, 5, 7} ∩ {2, 3, 4, 5} is {3, 5}
- The difference of sets A, B is the set whose elements belong to A but not to B.

  Examples: {1, 3, 5, 7} – {2, 3, 4, 5} is {1, 7}; {2, 3, 4, 5} – {1, 3, 5, 7} is {2, 4}
- Set A is a subset of set B if every element of set A is also an element of set B.

  Example: {1, 3, 5, 7} ⊂ {1, 2, 3, 4, 5, 7} is true

The `Set` Interface and Methods

- Required methods: testing set membership, testing for an empty set, determining set size, and creating an iterator over the set
- Optional methods: adding an element and removing an element
- Constructors to enforce the “no duplicate members” criterion
  - The `add` method does not allow duplicate items to be inserted

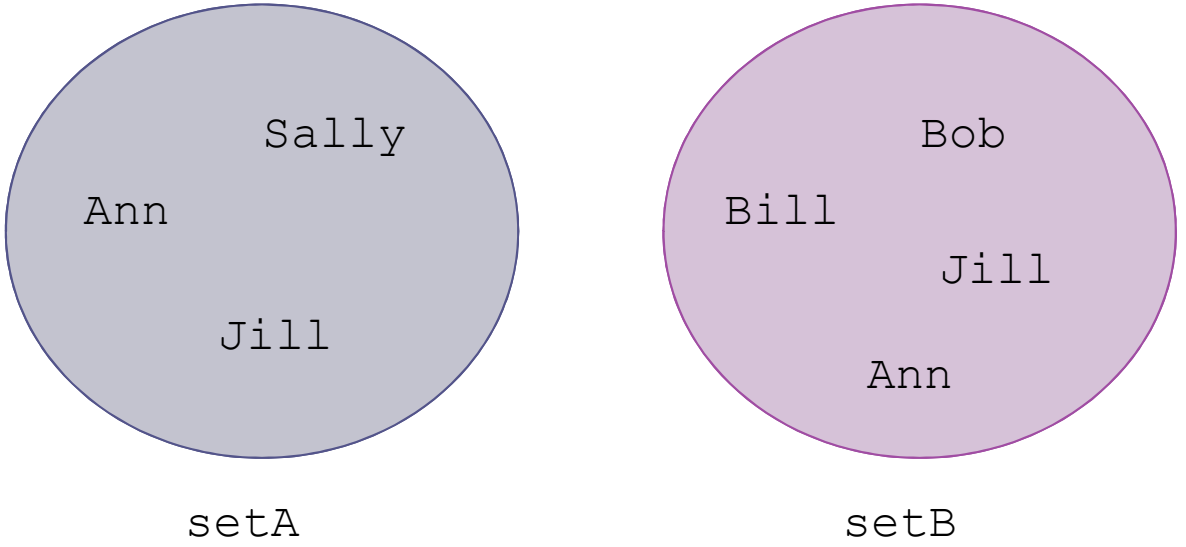
The `Set` Interface and Methods(cont.)

- Required method: `containsAll` tests the subset relationship
- Optional methods: `addAll`, `retainAll`, and `removeAll` perform union, intersection, and difference, respectively

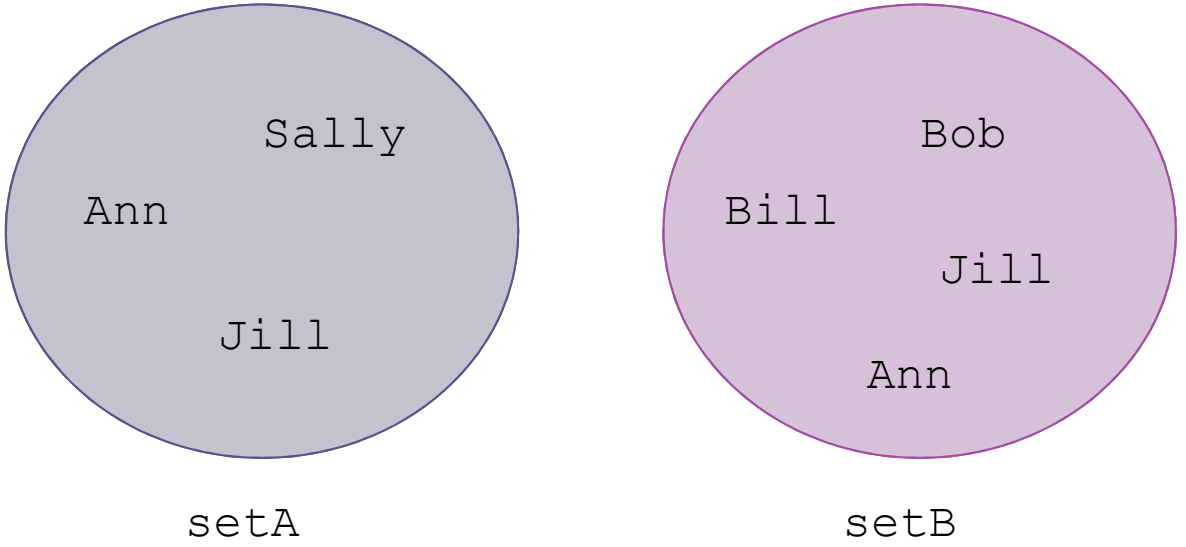
The `Set` Interface and Methods(cont.)

Method	Behavior
`boolean add(E obj)`	Adds item `obj` to this set if it is not already present (optional operation) and returns **`true`**. Returns false if `obj` is already in the set.
`boolean addAll(Collection<E> coll)`	Adds all of the elements in collection `coll` to this set if they're not already present (optional operation). Returns **`true`** if the set is changed. Implements *set union* if `coll` is a `Set`.
`boolean contains(Object obj)`	Returns **`true`** if this set contains an element that is equal to `obj`. Implements a test for *set membership*.
`boolean containsAll(Collection<E> coll)`	Returns **`true`** if this set contains all of the elements of collection `coll`. If `coll` is a set, returns **`true`** if this set is a subset of `coll`.
`boolean isEmpty()`	Returns **`true`** if this set contains no elements.
`Iterator<E> iterator()`	Returns an iterator over the elements in this set.
`boolean remove(Object obj)`	Removes the set element equal to `obj` if it is present (optional operation). Returns **`true`** if the object was removed.
`boolean removeAll(Collection<E> coll)`	Removes from this set all of its elements that are contained in collection `coll` (optional operation). Returns **`true`** if this set is changed. If `coll` is a set, performs the *set difference* operation.
`boolean retainAll(Collection<E> coll)`	Retains only the elements in this set that are contained in collection `coll` (optional operation). Returns **`true`** if this set is changed. If `coll` is a set, performs the *set intersection* operation.
`int size()`	Returns the number of elements in this set (its cardinality).

The `Set` Interface and Methods(cont.)

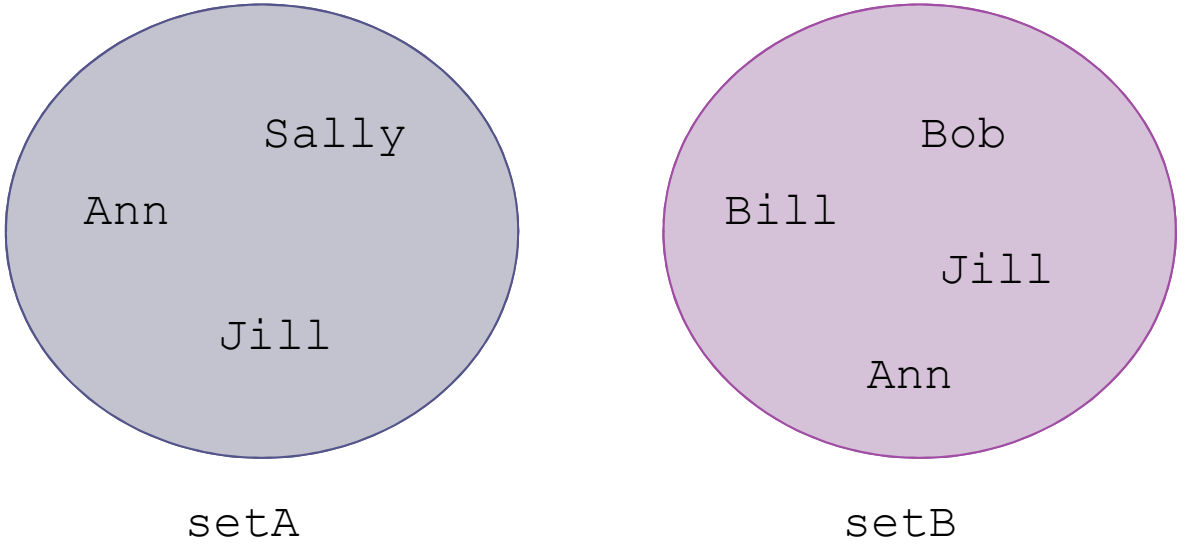


The `Set` Interface and Methods(cont.)



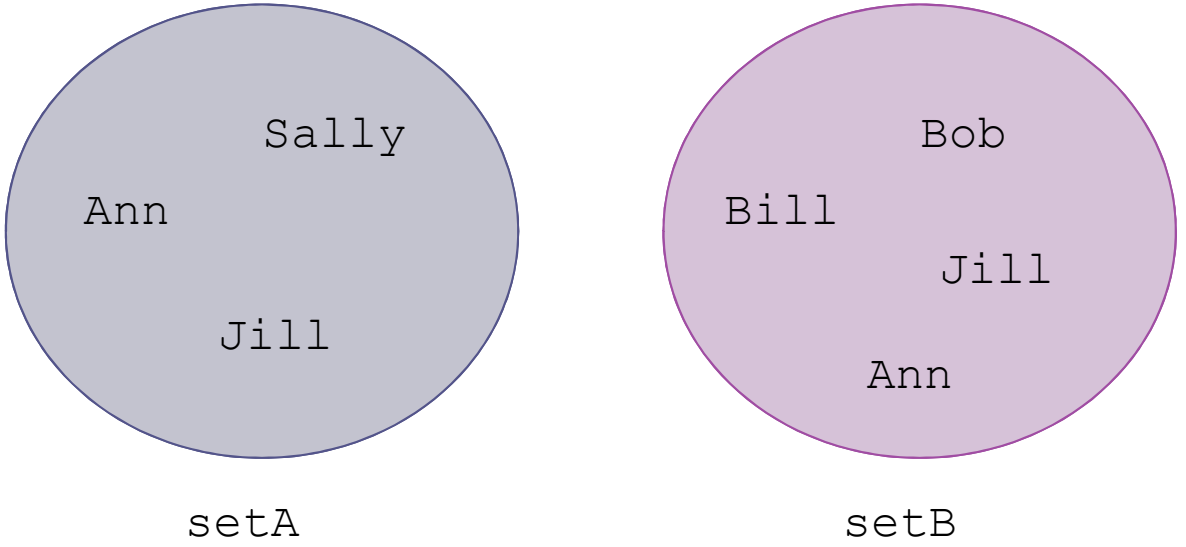

```
setA.addAll(setB);
```

The `Set` Interface and Methods(cont.)



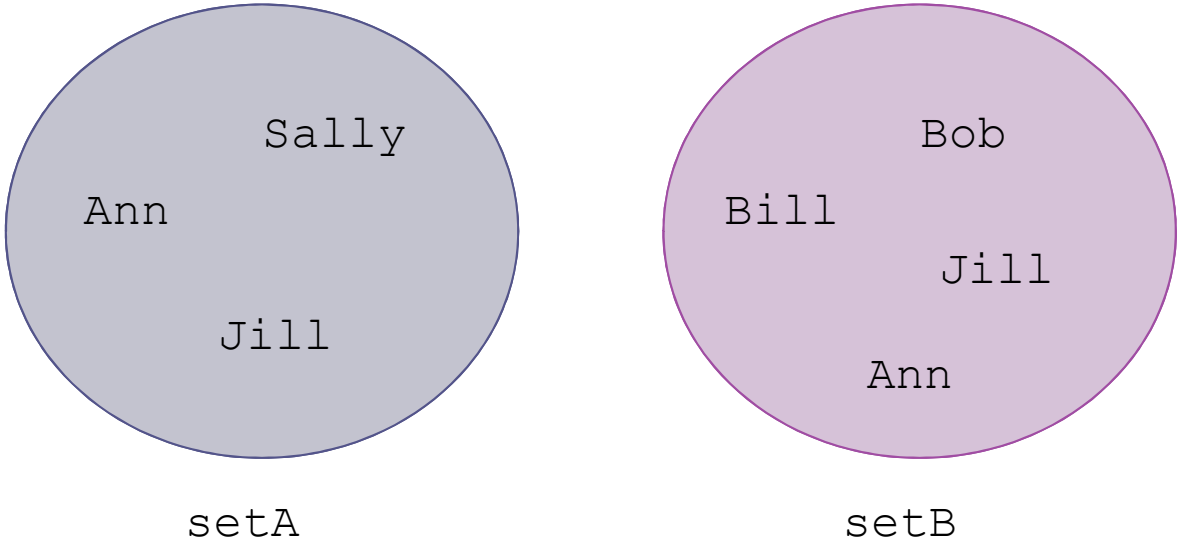

```
setA.addAll(setB);

System.out.println(setA);

Outputs:
[Bill, Jill, Ann, Sally, Bob]
```

The `Set` Interface and Methods(cont.)



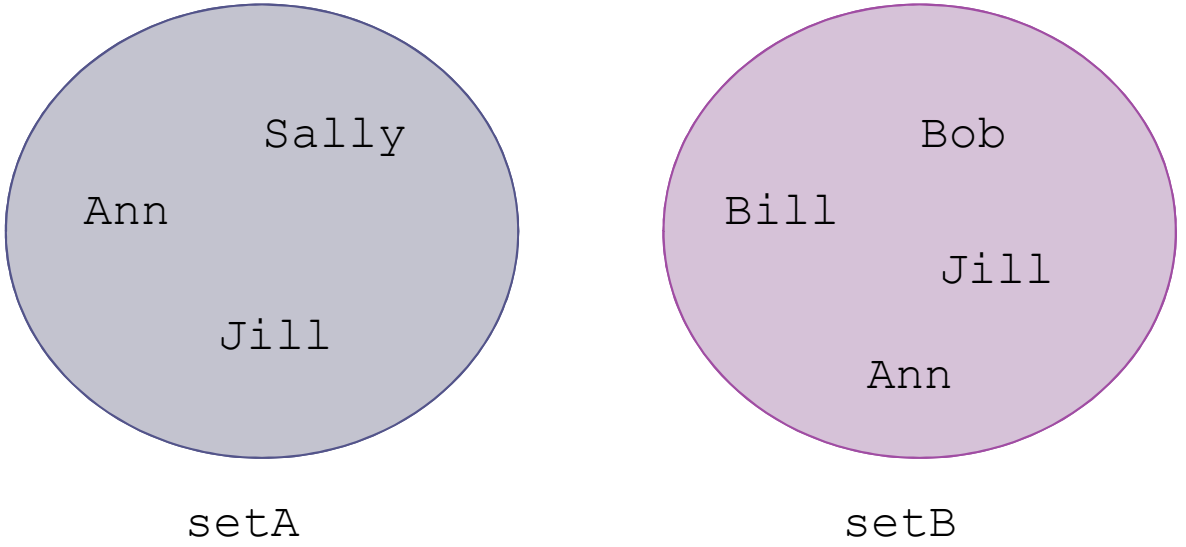
If a copy of original `setA` is in `setACopy`, then . . .

The `Set` Interface and Methods(cont.)



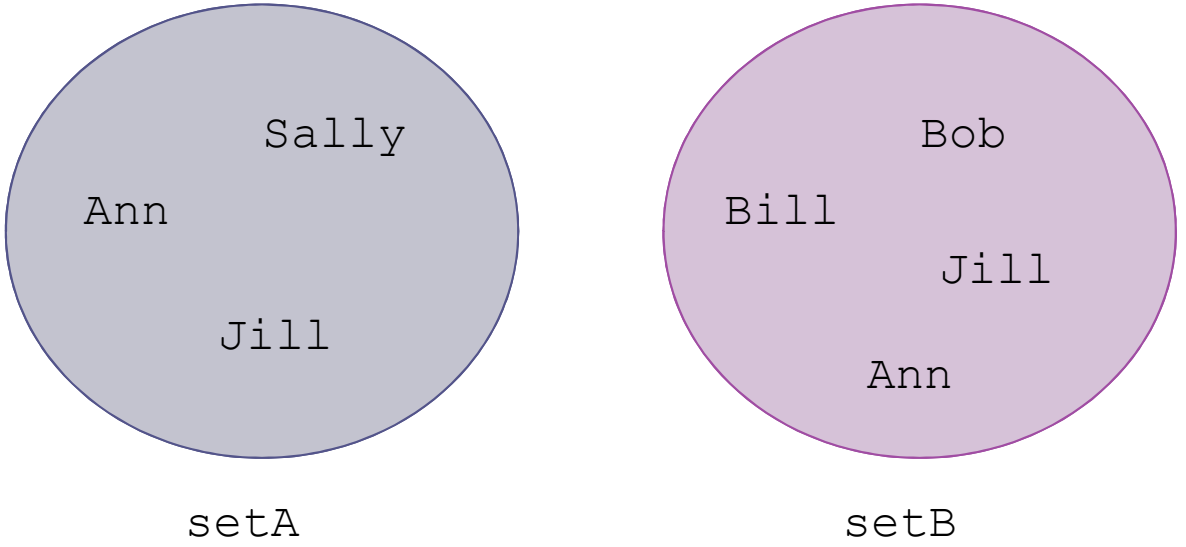

```
setACopy.retainAll(setB);
```

The `Set` Interface and Methods(cont.)




```
setACopy.retainAll(setB);

System.out.println(setACopy);

Outputs:
[Jill, Ann]
```

The `Set` Interface and Methods(cont.)




```
setACopy.removeAll(setB);

System.out.println(setACopy);

Outputs:
[Sally]
```

The `Set` Interface and Methods(cont.)

```
public static void main(String[] args) {

    // Create the sets.
    String[] listA = {"Ann", "Sally", "Jill", "Sally"};
    String[] listB = {"Bob", "Bill", "Ann", "Jill"};
    Set<String> setA = new HashSet<String>();
    Set<String> setAcopy = new HashSet<String>();  // Copy of setA
    Set<String> setB = new HashSet<String>();

    // Load sets from arrays.
    for (int i = 0; i < listA.length; i++) {
        setA.add(listA[i]);
        setAcopy.add(listA[i]);
    }

    for (int i = 0; i < listB.length; i++) {
        setB.add(listB[i]);
    }
    System.out.println("The 2 sets are: " + "\n" + setA
                       + "\n" + setB);
    // Display the union and intersection.
    setA.addAll(setB);          // Set union
    setAcopy.retainAll(setB);  // Set intersection
    System.out.println("Items in set union are: " + setA);
    System.out.println("Items in set intersection are: "
                       + setAcopy);
}
```

Comparison of Lists and Sets

- Collections implementing the `Set` interface may contain only unique elements
- Unlike the `List.add` method, the `Set.add` method returns `false` if you attempt to insert a duplicate item
- Unlike a `List`, a `Set` does not have a `get` method—elements cannot be accessed by index

Comparison of Lists and Sets (cont.)

- You can iterate through all elements in a `Set` using an `Iterator` object, but the elements will be accessed in arbitrary order

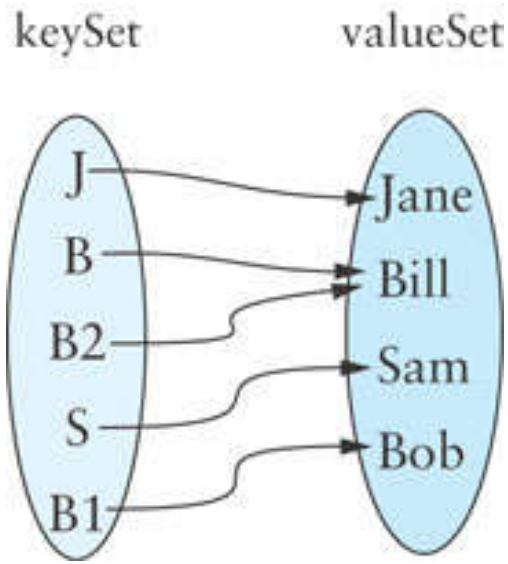
```
for (String nextItem : setA) {
  //Do something with nextItem
  …
}
```

Maps and the `Map` Interface

Section 7.2

Maps and the `Map` Interface

- The `Map` is related to the `Set`
- Mathematically, a `Map` is a set of ordered pairs whose elements are known as the key and the value
- Keys must be unique, but values need not be unique
- You can think of each key as a "mapping" to a particular value
- A map provides efficient storage and retrieval of information in a table
- A map can have *many-to-one* mapping: (`B, Bill`), (`B2, Bill`)




```
{(J, Jane), (B, Bill),
 (S, Sam), (B1, Bob),
 (B2,  Bill)}
```

Maps and the `Map` Interface(cont.)

- In an *onto* mapping, all the elements of valueSet have a corresponding member in keySet
- The `Map` interface should have methods of the form

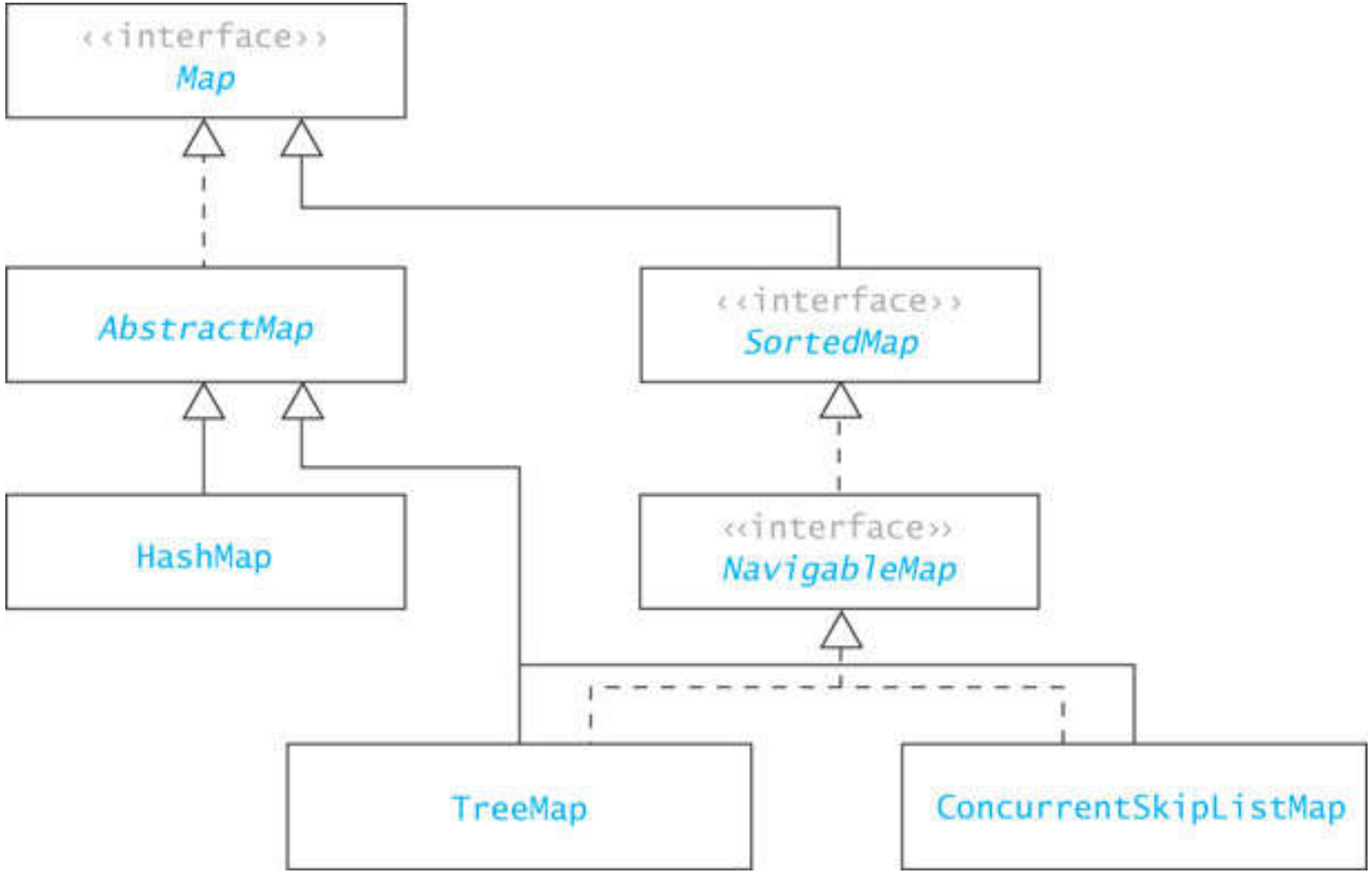
```
V get (Object key)
V put (K key, V value)
```

Maps and the `Map` Interface(cont.)

- When information about an item is stored in a table, the information should have a unique ID
- A unique ID may or may not be a number
- This unique ID is equivalent to a key

Type of item	Key	Value
University student	Student ID number	Student name, address, major, grade point average
Online store customer	E-mail address	Customer name, address, credit card information, shopping cart
Inventory item	Part ID	Description, quantity, manufacturer, cost, price

Map **Hierarchy**



`Map` **Interface**

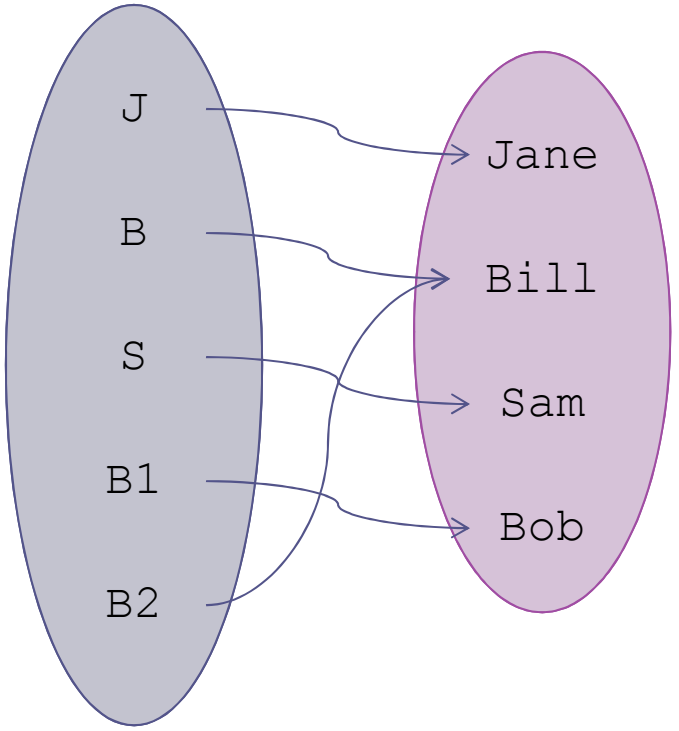
Method	Behavior
`V get(Object key)`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present.
`boolean isEmpty()`	Returns **`true`** if this map contains no `key-value` mappings.
`V put(K key, V value)`	Associates the specified `value` with the specified `key` in this map (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`V remove(Object key)`	Removes the mapping for this `key` from this map if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`int size()`	Returns the number of `key-value` mappings in this map.

`Map` **Interface** (cont.)

- The following statements build a `Map` object:

```
Map<String, String> aMap =
   new HashMap<String,
   String>();
```

```
aMap.put("J", "Jane");
aMap.put("B", "Bill");
aMap.put("S", "Sam");
aMap.put("B1", "Bob");
aMap.put("B2", "Bill");
```

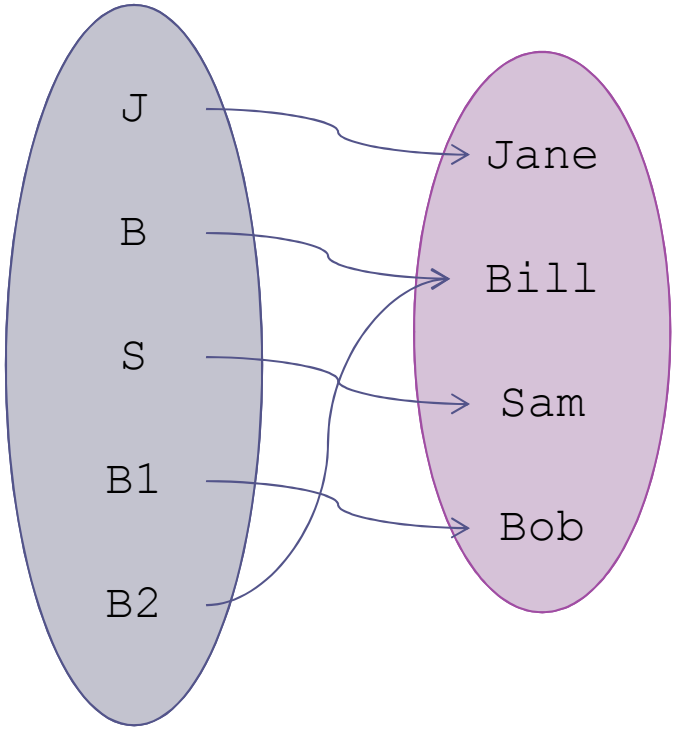


`Map` **Interface** (cont.)

```
aMap.get("B1")
```

returns:

```
"Bob"
```



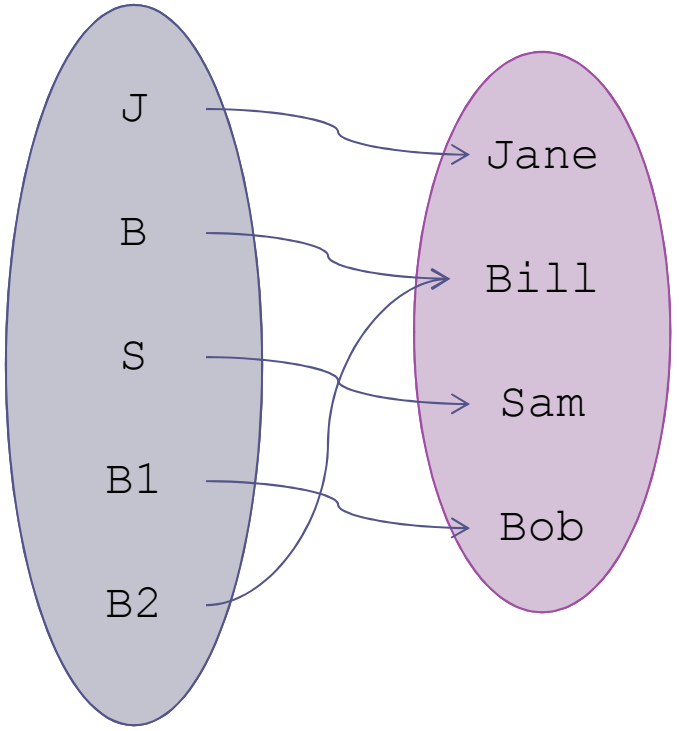
`Map` **Interface** (cont.)

```
aMap.get("Bill")
```

returns:

```
null
```

("`Bill`" is a value, not a key)



Hash Tables

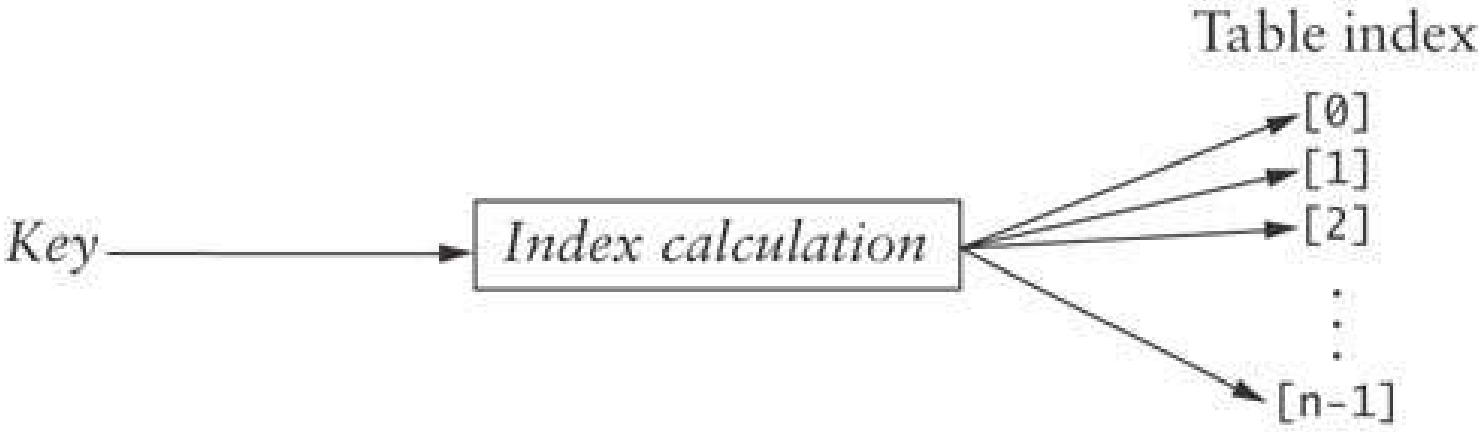
Section 7.3

Hash Tables

- The goal of hash table is to be able to access an entry based on its key value, not its location
- We want to be able to access an entry directly through its key value, rather than by having to determine its location first by searching for the key value in an array
- Using a hash table enables us to retrieve an entry in constant time (on *average*, O(1))

Hash Codes and Index Calculation

- The basis of hashing is to transform the item's key value into an integer value (its *hash code*) which is then transformed into a table index



Hash Codes and Index Calculation (cont.)

- Consider the Huffman code problem from the last chapter.
- If a text contains only ASCII values, which are the first 128 Unicode values we could use a table of size 128 and let its Unicode value be its location in the table

Hash Codes and Index Calculation (cont.)

- However, what if all 65,536 Unicode characters were allowed?
- If you assume that on average 100 characters were used, you could use a table of 200 characters and compute the index by:

```
int index = unicode % 200
```

. . .	. . .
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
. . .	. . .

Hash Codes and Index Calculation (cont.)

- If a text contains this snippet:

```
. . . mañana (tomorrow), I'll finish my program. . .
```

- Given the following Unicode values:

Hexadecimal	Decimal	Name	Character
0x0029	41	right parenthesis	)
0x00F1	241	small letter n with tilde	ñ

- The indices for letters '`ñ`' and '`)`' are both 41

`41 % 200 = 41` and `241 % 200 = 41`

- This is called a *collision;* we will discuss how to deal with collisions shortly

Methods for Generating Hash Codes

- In most applications, a key will consist of strings of letters or digits (such as a social security number, an email address, or a partial ID) rather than a single character
- The number of possible key values is much larger than the table size
- Generating good hash codes typically is an experimental process
- The goal is a *random distribution of values*
- Simple algorithms sometimes generate lots of collisions

Java `HashCode` Method

- For strings, simply summing the `int` values of all characters returns the same hash code for `"sign"` and `"sing"`
- The Java API algorithm accounts for position of the characters as well
- `String.hashCode()` returns the integer calculated by the formula:

$$s_0 \times 31^{(n-1)} + s_1 \times 31^{(n-2)} + \ldots + s_{n-1}$$

  where s_i is the *i*th character of the string, and *n* is the length of the string
- "Cat" has a hash code of:

$$\text{'C'} \times 31^2 + \text{'a'} \times 31 + \text{'t'} = 67{,}510$$

- 31 is a prime number, and prime numbers generate relatively few collisions

Java `HashCode` Method (cont.)

- Because there are too many possible strings, the integer value returned by `String.hashCode` can't be unique
- However, because the `String.hashCode` method distributes the hash code values fairly evenly throughout the range, the probability of two strings having the same hash code is low
- The probability of a collision with

  ```
  s.hashCode() % table.length
  ```

  is proportional to how full the table is

Methods for Generating Hash Codes (cont.)

- A good hash function should be relatively simple and efficient to compute
- It doesn't make sense to use an O(*n*) hash function to avoid doing an O(*n*) search

Open Addressing

- We now consider two ways to organize hash tables:
  - open addressing
  - chaining
- In open addressing, *linear probing* can be used to access an item in a hash table
  - If the index calculated for an item's key is occupied by an item with that key, we have found the item
  - If that element contains an item with a different key, increment the index by one
  - Keep incrementing until you find the key or a `null` entry (assuming the table is not full)

Open Addressing (cont.)

Algorithm for Accessing an Item in a Hash Table

1. Compute the index by taking the item's `hashCode() % table.length`.
2. `if table[index]` is `null`
3. The item is not in the table.
4. `else if table[index]` is equal to the item
5. The item is in the table.

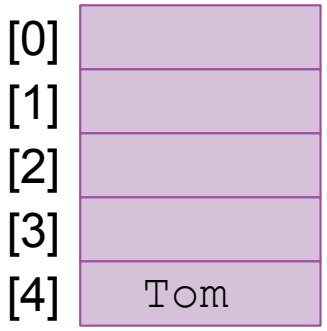
   `else`
6. Continue to search the table by incrementing the index until either the item is found or a `null` entry is found.

Table Wraparound and Search Termination

- As you increment the table index, your table should wrap around as in a circular array
- This enables you to search the part of the table before the hash code value in addition to the part of the table after the hash code value
- But it could lead to an infinite loop
- How do you know when to stop searching if the table is full and you have not found the correct value?
  - Stop when the index value for the next probe is the same as the hash code value for the object
  - Ensure that the table is never full by increasing its size after an insertion when its load factor exceeds a specified threshold

Hash Code Insertion Example

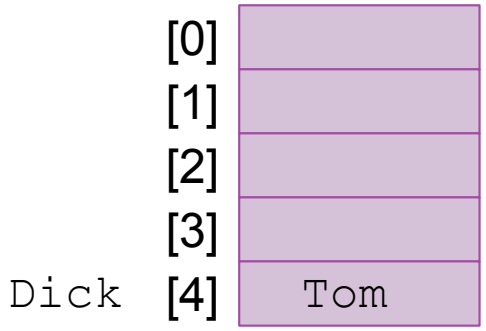
Tom Dick Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)

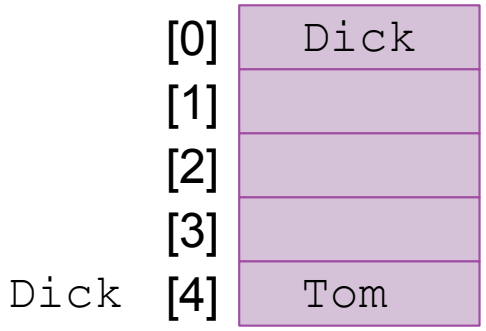
Dick Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)

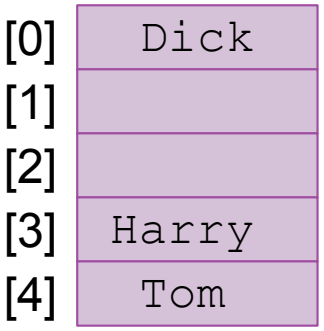
Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)

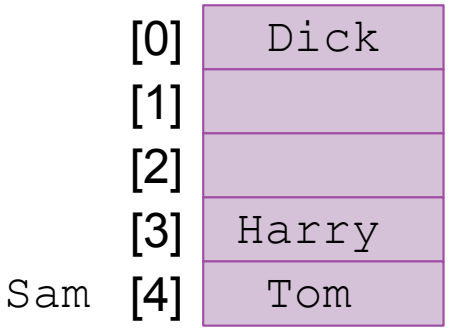
Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)

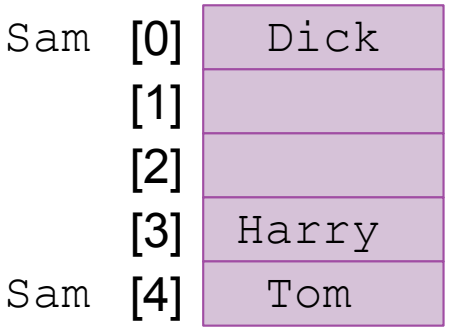
Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)

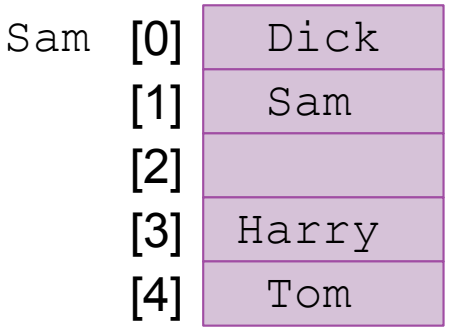
Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)

Pete

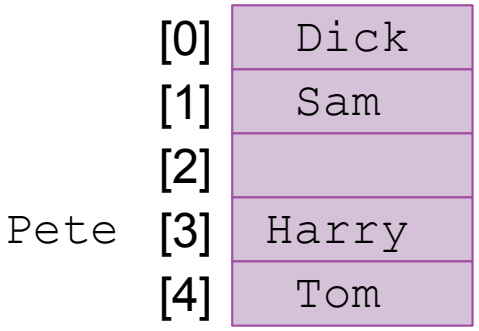


Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

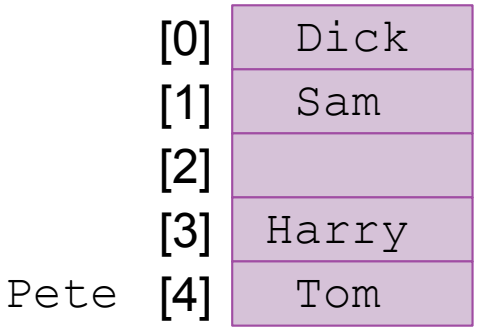
Hash Code Insertion Example (cont.)

Pete

Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

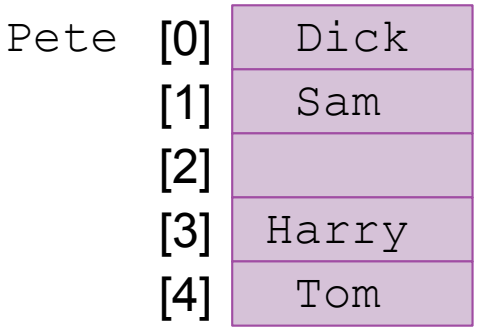


Hash Code Insertion Example (cont.)



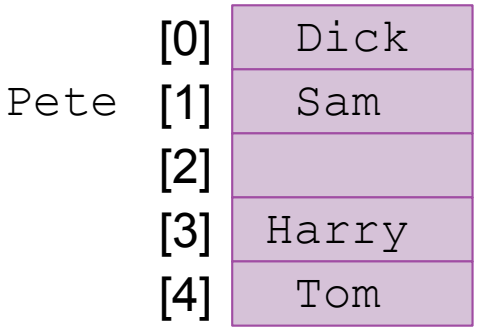
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



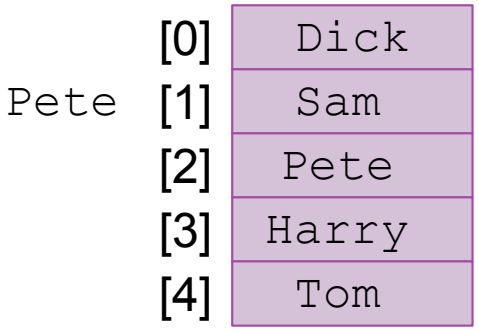
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



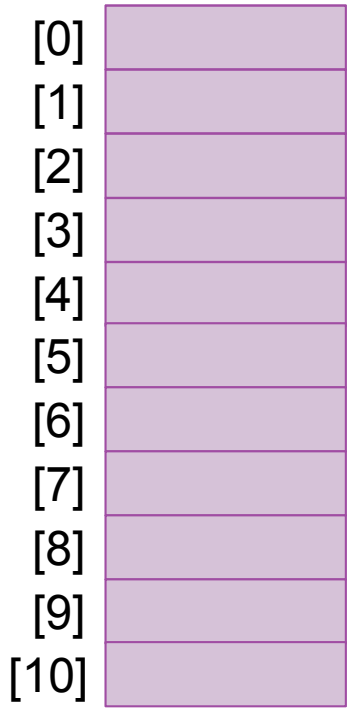
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Retrieval of "Tom" or "Harry" takes one step, O(1)

Because of collisions, retrieval of the others requires a linear search

Hash Code Insertion Example (cont.)

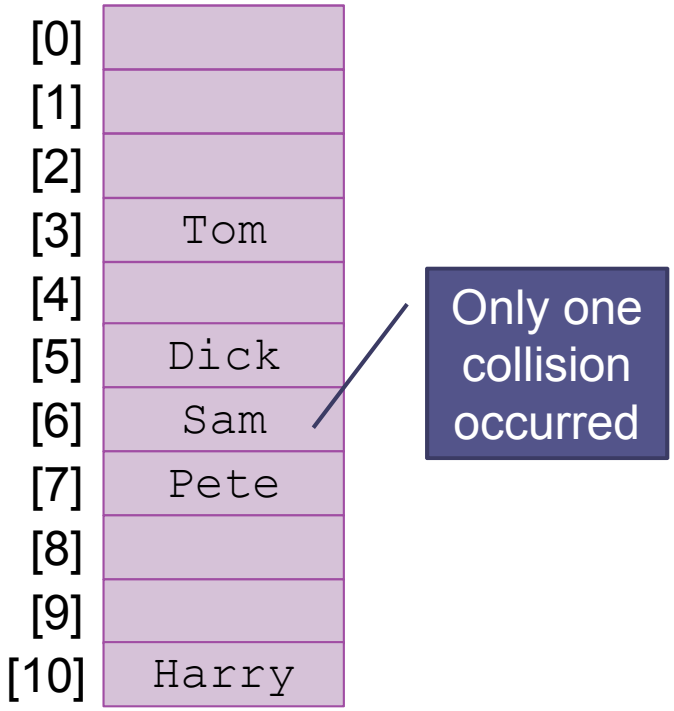
Name	hashCode()	hashCode()%11
"Tom"	84274	3
"Dick"	2129869	5
"Harry"	69496448	10
"Sam"	82879	5
"Pete"	2484038	7



Hash Code Insertion Example (cont.)

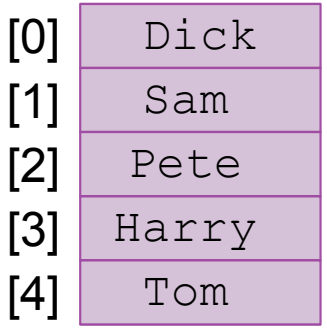
Name	hashCode()	hashCode()%11
"Tom"	84274	3
"Dick"	2129869	5
"Harry"	69496448	10
"Sam"	82879	5
"Pete"	2484038	7

The best way to reduce the possibility of collision (and reduce linear search retrieval time because of collisions) is to increase the table size



Traversing a Hash Table

- You cannot traverse a hash table in a meaningful way since the sequence of stored values is arbitrary



Dick, Sam, Pete, Harry, Tom

Tom, Dick, Sam, Pete, Harry

Deleting an Item Using Open Addressing

- When an item is deleted, you cannot simply set its table entry to null
- If we search for an item that may have collided with the deleted item, we may conclude incorrectly that it is not in the table.
- Instead, store a dummy value or mark the location as available, but previously occupied
- Deleted items reduce search efficiency which is partially mitigated if they are marked as available
- You cannot simply replace a deleted item with a new item until you verify that the new item is not in the table

Reducing Collisions by Expanding the Table Size

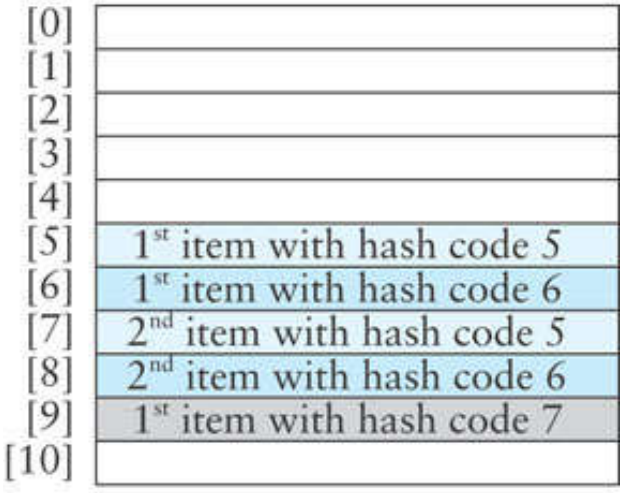
- Use a prime number for the size of the table to reduce collisions
- A fuller table results in more collisions, so, when a hash table becomes sufficiently full, a larger table should be allocated and the entries reinserted
- You must reinsert (*rehash*) values into the new table; do not copy values as some search chains which were wrapped may break
- Deleted items are not reinserted, which saves space and reduces the length of some search chains

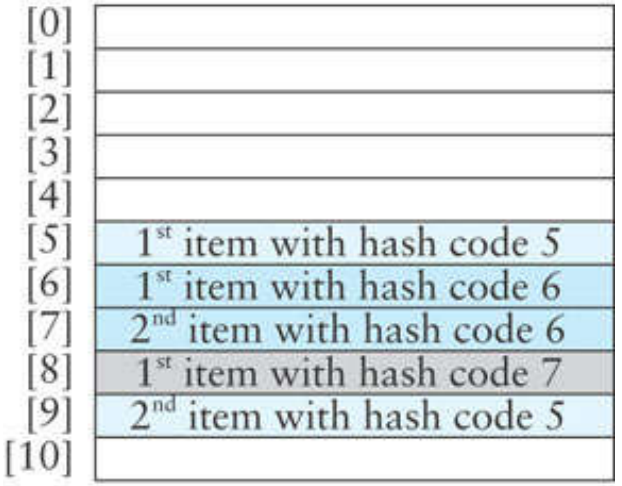
Reducing Collisions Using Quadratic Probing

- Linear probing **tends to form clusters of keys** in the hash table, causing longer search chains
- *Quadratic probing* can reduce the effect of clustering
  - Increments form a quadratic series ($1 + 2^2 + 3^2 + ...$)

```
probeNum++;
index = (startIndex + probeNum * probeNum) % table.length
```

- If an item has a hash code of 5, successive values of index will be 6 (5+1), 9 (5+4), 14 (5+9), . . .





Problems with Quadratic Probing

- The disadvantage of quadratic probing is that the next index calculation is time-consuming, involving multiplication, addition, and modulo division
- A more efficient way to calculate the next index is:

```
k += 2;
index = (index + k) % table.length;
```

Problems with Quadratic Probing (cont.)

- Examples:
  - If the initial value of k is -1, successive values of k will be 1, 3, 5, …
  - If the initial value of index is 5, successive value of index will be 6 (= 5 + 1), 9 (= 5 + 1 + 3), 14 (= 5 + 1 + 3 + 5), …
- The proof of the equality of these two calculation methods is based on the mathematical series:

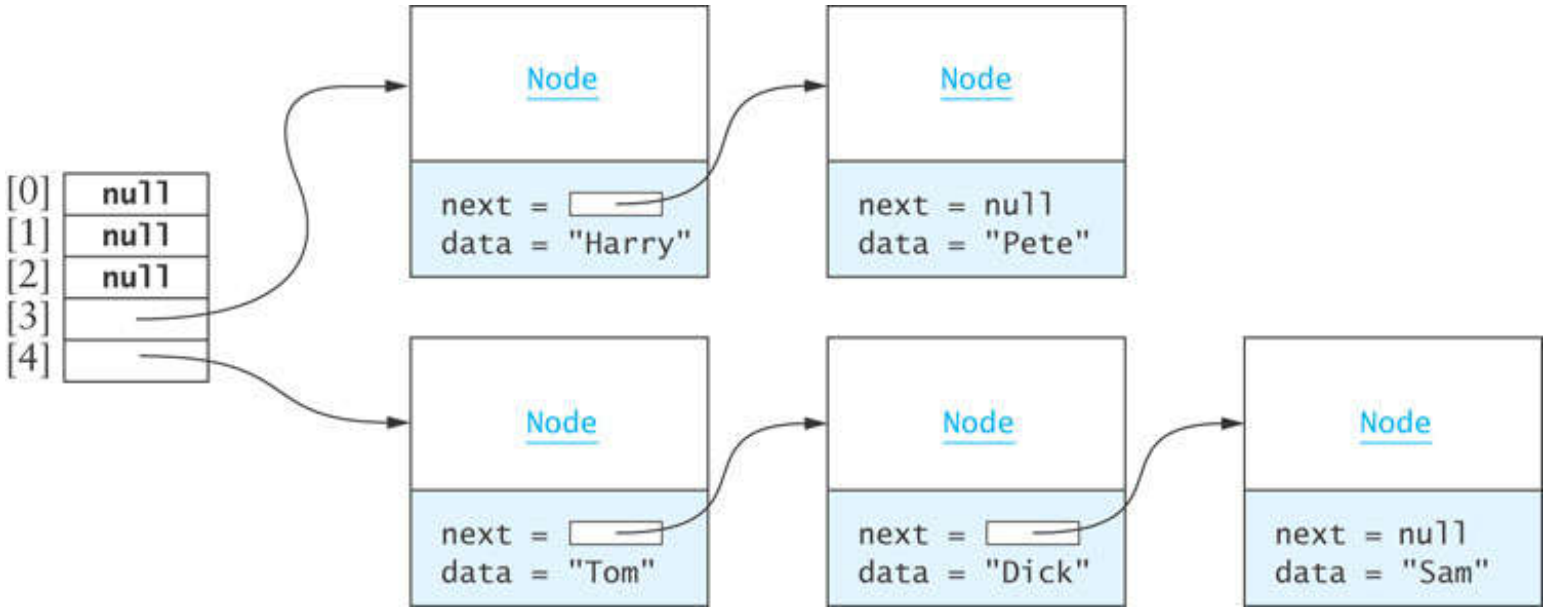
$$n^2 = 1 + 3 + 5 + \ldots + 2n - 1$$

Problems with Quadratic Probing (cont.)

- A **more serious problem** is that not all table elements are examined when looking for an insertion index; this may mean that
  - an item can't be inserted even when the table is not full
  - the program will get stuck in an infinite loop searching for an empty slot
- If the table size is a prime number and it is never more than half full, this won't happen
- However, requiring a half empty table wastes a lot of memory

Chaining

- *Chaining* is an alternative to open addressing
- Each table element references a linked list that contains all of the items that hash to the same table index
  - The linked list often is called a *bucket*
  - The approach sometimes is called *bucket hashing*



Chaining (cont.)

- Advantages relative to open addressing:
  - Only items that have the same value for their hash codes are examined when looking for an object
  - You can store more elements in the table than the number of table slots (indices)
  - Once you determine an item is not present, you can insert it at the beginning or end of the list
  - To remove an item, you simply delete it; you do not need to replace it with a dummy item or mark it as deleted

Performance of Hash Tables

- *Load factor* is the number of filled cells divided by the table size
- Load factor has the greatest effect on hash table performance
- The lower the load factor, the better the performance as there is a smaller chance of collision when a table is sparsely populated
- If there are no collisions, performance for search and retrieval is O(1) regardless of table size

Performance of Open Addressing versus Chaining

- Donald E. Knuth derived the following formula for the expected number of comparisons, *c*, required for finding an item that is in a hash table using open addressing with linear probing and a load factor *L*

$$c = \frac{1}{2}\left(1 + \frac{1}{1 - L}\right)$$

Performance of Open Addressing versus Chaining (cont.)

- Using chaining, if an item is in the table, on average we must examine the table element corresponding to the item's hash code and then half of the items in each list

$$c = 1 + \frac{L}{2}$$

where L is the average number of items in a list

Performance of Open Addressing versus Chaining (cont.)

L	Number of Probes with Linear Probing	Number of Probes with Chaining
0.0	1.00	1.00
0.25	1.17	1.13
0.5	1.50	1.25
0.75	2.50	1.38
0.85	3.83	1.43
0.9	5.50	1.45
0.95	10.50	1.48

Performance of Hash Tables versus Sorted Array and Binary Search Tree

- The number of comparisons required for a binary search of a sorted array is O(log *n*)
  - A sorted array of size 128 requires up to 7 probes (2^7 is 128) which is more than for a hash table of any size that is 90% full
  - A binary search tree performs similarly
- Insertion or removal

hash table	O(1) expected; worst case O(*n*)
unsorted array	O(*n*)
binary search tree	O(log *n*); worst case O(*n*)

Storage Requirements for Hash Tables, Sorted Arrays, and Trees

- The performance of hashing is superior to that of binary search of an array or a binary search tree, particularly if the load factor is less than 0.75
- However, the lower the load factor, the more empty storage cells
  - there are no empty cells in a sorted array
- A binary search tree requires three references per node (item, left subtree, right subtree), so more storage is required for a binary search tree than for a hash table with load factor 0.75

Storage Requirements for Open Addressing and Chaining

- For open addressing, the number of references to items (key-value pairs) is *n* (the size of the table)
- For chaining , the average number of nodes in a list is *L* (the load factor) and *n* is the number of table elements
  - Using the Java API `LinkedList`, there will be three references in each node (item, next, previous)
  - Using our own single linked list, we can reduce the references to two by eliminating the previous-element reference
  - Therefore, storage for $n + 2L$ references is needed

Storage Requirements for Open Addressing and Chaining (cont.)

- Example:
  - Assume open addressing, 60,000 items in the hash table, and a load factor of 0.75
  - This requires a table of size 80,000 and results in an expected number of comparisons of 2.5
  - Calculating the table size *n* to get similar performance using chaining

    $2.5 = 1 + L/2$

    $5.0 = 2 + L$

    $3.0 = 60{,}000/n$

    $n = 20{,}000$

Storage Requirements for Open Addressing and Chaining (cont.)

- A hash table of size 20,000 provides storage space for 20,000 references to lists
- There are 60,000 nodes in the table (one for each item)
- This requires storage for 140,000 references (2 x 60,000 + 20,000), which is 175% of the storage needed for open addressing

Double hashing

Double hashing uses the idea of applying a second hash function to a key when a collision occurs

hash1(key) + i * hash2(key)) % TABLE_SIZE

Keep increasing i while collisions occur

where hash2(key) = PRIME – (key % PRIME)

where PRIME is a prime smaller than the TABLE_SIZE

Just make sure that hash2 never evaluates to zero and that it can probe all cells...

Implementing the Hash Table

Section 7.4

Interface `KWHashMap`

Method	Behavior
`V get(Object key)`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present.
`boolean isEmpty()`	Returns **`true`** if this table contains no `key-value` mappings.
`V put(K key, V value)`	Associates the specified `value` with the specified `key`. Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`V remove(Object key)`	Removes the mapping for this `key` from this table if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping.
`int size()`	Returns the size of the table.

Class `Entry`

Data Field	Attribute
`private K key`	The key.
`private V value`	The value.
Constructor	**Behavior**
`public Entry(K key, V value)`	Constructs an `Entry` with the given values.
Method	**Behavior**
`public K getKey()`	Retrieves the key.
`public V getValue()`	Retrieves the value.
`public V setValue(V val)`	Sets the value.

Class Entry

```
/** Contains key-value pairs for a hash table. */
private static class Entry<K, V> {

    /** The key */
    private K key;
    /** The value */
    private V value;

    /** Creates a new key-value pair.
        @param key The key
        @param value The value
    */
    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }

    /** Retrieves the key.
        @return The key
    */
    public K getKey() {
        return key;
    }

    /** Retrieves the value.
        @return The value
    */
    public V getValue() {
        return value;
    }

    /** Sets the value.
        @param val The new value
        @return The old value
    */
    public V setValue(V val) {
        V oldVal = value;
        value = val;
        return oldVal;
    }
}
```

Class `HashTableOpen`

Data Field	Attribute
`private Entry<K, V>[] table`	The hash table array.
`private static final int START_CAPACITY`	The initial capacity.
`private double LOAD_THRESHOLD`	The maximum load factor.
`private int numKeys`	The number of keys in the table excluding keys that were deleted.
`private int numDeletes`	The number of deleted keys.
`private final Entry<K, V> DELETED`	A special object to indicate that an entry has been deleted.

Data Field	Attribute
`private Entry<K, V>[] table`	The hash table array.
`private static final int START_CAPACITY`	The initial capacity.
`private double LOAD_THRESHOLD`	The maximum load factor.
`private int numKeys`	The number of keys in the table excluding keys that were deleted.
`private int numDeletes`	The number of deleted keys.
`private final Entry<K, V> DELETED`	A special object to indicate that an entry has been deleted.

```
/** Hash table implementation using open addressing. */
public class HashtableOpen<K, V> implements KWHashMap<K, V> {
        // Data Fields
        private Entry<K, V>[] table;
        private static final int START_CAPACITY = 101;
        private double LOAD_THRESHOLD = 0.75;
        private int numKeys;
        private int numDeletes;
        private final Entry<K, V> DELETED =
        new Entry<K, V>(null, null);

        // Constructor
        public HashTableOpen() {
                table = new Entry[START_CAPACITY];
        }

        // Insert inner class Entry<K, V> here.
      . . .
```

Class `HashTableOpen` (cont.)

Method	Behavior
`private int find(Object key)`	Returns the index of the specified key if present in the table; otherwise, returns the index of the first available slot.
`private void rehash()`	Doubles the capacity of the table and permanently removes deleted items.

Algorithm for `HashtableOpen.find(Object key)`

1. Set `index` to `key.hashCode() % table.length`.
2. `if index` is negative, add `table.length`.
3. `while table[index]` is not empty and the key is not at `table[index]`
4. increment index.
5. `if index` is greater than or equal to `table.length`
6. Set `index` to 0.
7. Return the index.

```
/** Finds either the target key or the first empty slot in the
    search chain using linear probing.
    pre: The table is not full.
    @param key The key of the target object
    @return The position of the target or the first empty slot if
            the target is not in the table.
*/
private int find(Object key) {
    // Calculate the starting index.
    int index = key.hashCode() % table.length;
    if (index < 0)
        index += table.length;    // Make it positive.

    // Increment index until an empty slot is reached
    // or the key is found.
    while ((table[index] != null)
            && (!key.equals(table[index].key))) {
        index++;
        // Check for wraparound.
        if (index >= table.length)
            index = 0;    // Wrap around.
    }
    return index;
}
```

Class `HashTableOpen` (cont.)

Algorithm for `get(Object key)`

1. Find the first table element that is empty or the table element that contains the key.
2. `if` the table element found contains the key
   return the value at this table element.
3. `else`
4. return `null`.

Class `HashTableOpen` (cont.)

```
/** Method get for class HashtableOpen.
    @param key The key being sought
    @return the value associated with this key if found;
            otherwise, null
*/
@Override
public V get(Object key) {
    // Find the first table element that is empty
    // or the table element that contains the key.
    int index = find(key);

    // If the search is successful, return the value.
    if (table[index] != null)
        return table[index].value;
    else
        return null;  // key not found.
}
```

Class `HashTableOpen` (cont.)

Algorithm for `HashtableOpen.put(K key, V value)`

1. Find the first table element that is empty or the table element that contains the key.
2. `if` an empty element was found
3. insert the new item and increment `numKeys`.
4. check for need to rehash.
5. return `null`.
6. The key was found. Replace the value associated with this table element and return the old value.

Class `HashTableOpen` (cont.)

```
public V put(K key, V value)  {
    // Find the first table element that is empty
    // or the table element that contains the key.
    int index = find(key);

    // If an empty element was found, insert new entry.
    if (table[index] == null) {
        table[index] = new Entry<K, V>(key, value);
        numKeys++;
        // Check whether rehash is needed.
        double loadFactor =
                (double) (numKeys + numDeletes) / table.length;
        if (loadFactor > LOAD_THRESHOLD)
            rehash();
        return null;
    }

    // assert: table element that contains the key was found.
    // Replace value for this key.
    V oldVal = table[index].value;
    table[index].value = value;
    return oldVal;
}
```

Class `HashTableOpen` (cont.)

Algorithm for `remove(Object key)`

1. Find the first table element that is empty or the table element that contains the key.
2. `if` an empty element was found
3. return `null`.
4. Key was found. Remove this table element by setting it to reference DELETED, increment `numDeletes`, and decrement `numKeys`.
5. Return the value associated with this key.

Class `HashTableOpen` (cont.)

Algorithm for `HashtableOpen.rehash`

1. Allocate a new hash table that is at least double the size and has an odd length.
2. Reset the number of keys and number of deletions to 0.
3. Reinsert each table entry that has not been deleted in the new hash table.

Class `HashTableOpen` (cont.)

```
/** Expands table size when loadFactor exceeds LOAD_THRESHOLD
    post: The size of the table is doubled and is an odd integer.
          Each nondeleted entry from the original table is
          reinserted into the expanded table.
          The value of numKeys is reset to the number of items
          actually inserted; numDeletes is reset to 0.
*/
private void rehash() {
    // Save a reference to oldTable.
    Entry<K, V>[] oldTable = table;
    // Double capacity of this table.
    table = new Entry[2 * oldTable.length + 1];

    // Reinsert all items in oldTable into expanded table.
    numKeys = 0;
    numDeletes = 0;
    for (int i = 0; i < oldTable.length; i++) {
        if ((oldTable[i] != null) && (oldTable[i] != DELETED)) {
            // Insert entry in expanded table
            put(oldTable[i].key, oldTable[i].value);
        }
    }
}
```

Class `HashTableChain`

Data Field	Attribute
`private LinkedList<Entry<K, V>>[] table`	A table of references to linked lists of `Entry<K, V>` objects.
`private int numKeys`	The number of keys (entries) in the table.
`private static final int CAPACITY`	The size of the table.
`private static final int LOAD_THRESHOLD`	The maximum load factor.

Data Field	Attribute
`private LinkedList<Entry<K, V>>[] table`	A table of references to linked lists of `Entry<K, V>` objects.
`private int numKeys`	The number of keys (entries) in the table.
`private static final int CAPACITY`	The size of the table.
`private static final int LOAD_THRESHOLD`	The maximum load factor.

```
/** Hash table implementation using chaining. */
public class HashtableChain<K, V> implements KWHashMap<K, V> {
    /** The table */
    private LinkedList<Entry<K, V>>[] table;
    /** The number of keys */
    private int numKeys;
    /** The capacity */
    private static final int CAPACITY = 101;
    /** The maximum load factor */
    private static final double LOAD_THRESHOLD = 3.0;

    // Insert inner class Entry<K, V> here.

    // Constructor
    public HashtableChain() {
        table = new LinkedList[CAPACITY];
    }
. . .
}
```

Class `HashTableChain` (cont.)

Algorithm for `HashtableChain.get(Object key)`

1. Set `index` to `key.hashCode() % table.length`.
2. `if index` is negative
3. add `table.length`.
4. if `table[index]` is `null`
5. `key` is not in the table; return `null`.
6. For each element in the list at `table[index]`
7. `if` that element's key matches the search key
8. return that element's value.
9. `key` is not in the table; return `null`.

Class `HashTableChain` (cont.)

```
/** Method get for class HashtableChain.
    @param key The key being sought
    @return The value associated with this key if found;
            otherwise, null
*/
@Override
public V get(Object key) {
    int index = key.hashCode() % table.length;
    if (index < 0)
        index += table.length;
    if (table[index] == null)
        return null;  // key is not in the table.

    // Search the list at table[index] to find the key.
    for (Entry<K, V> nextItem : table[index]) {
        if (nextItem.key.equals(key))
            return nextItem.value;
    }

    // assert: key is not in the table.
    return null;
}
```

Class `HashTableChain` (cont.)

Algorithm for `HashtableChain.put(K key, V value)`

1. Set index to `key.hashCode() % table.length.`
2. `if index` is negative, add `table.length.`
3. `if table[index]` is `null`
4. create a new linked list at `table[index].`
5. Search the list at `table[index]` to find the key.
6. `if` the search is successful
7. replace the value associated with this key.
8. return the old value.
9. `else`
10. insert the new key-value pair in the linked list located at `table[index]`
11. increment `numKeys.`
12. `if` the load factor exceeds the `LOAD_THRESHOLD`
13. Rehash.
14. return `null.`

```
public V put(K key, V value) {
    int index = key.hashCode() % table.length;
    if (index < 0)
        index += table.length;
    if (table[index] == null) {
        // Create a new linked list at table[index].
        table[index] = new LinkedList<Entry<K, V>>();
    }

    // Search the list at table[index] to find the key.
    for (Entry<K, V> nextItem : table[index]) {
        // If the search is successful, replace the old value.
        if (nextItem.key.equals(key)) {
            // Replace value for this key.
            V oldVal = nextItem.value;
            nextItem.setValue(value);
            return oldVal;
        }
    }

    // assert: key is not in the table, add new item.
    table[index].addFirst(new Entry<K, V>(key, value));
    numKeys++;
    if (numKeys > (LOAD_THRESHOLD * table.length))
        rehash();
    return null;
}
```

Class `HashTableChain` (cont.)

Algorithm for `HashtableChain.remove(Object key)`

1. Set `index` to `key.hashCode() % table.length.`
2. `if index` is negative, add `table.length`.
3. `if table[index]` is `null`
4. `key` is not in the table; return `null`.
5. Search the list at `table[index]` to find the key.
6. `if` the search is successful
7. remove the entry with this key and decrement `numKeys`.
8. `if` the list at `table[index]` is empty
9. Set `table[index]` to `null`.
10. return the value associated with this key.
11. The `key` is not in the table; return `null`.

Implementation Considerations for Maps and Sets

Section 7.5

Methods `hashCode` and `equals`

- Class `Object` implements methods `hashCode` and `equals`, so every class can access these methods unless it overrides them
- `Object.equals` compares two objects based on their addresses, not their contents
- Most predefined classes override method `equals` and compare objects based on content
- If you want to compare two objects (whose classes you've written) for equality of content, you need to override the `equals` method

Methods `hashCode` and `equals` (cont.)

- `Object.hashCode` calculates an object's hash code based on its address, not its contents
- Most predefined classes also override method `hashcode`
- Java recommends that if you override the `equals` method, then you should also override the `hashCode` method
- Otherwise, you violate the following rule:

```
If obj1.equals(obj2) is true,
then obj1.hashCode() == obj2.hashCode()
```

Methods `hashCode` and `equals` (cont.)

- Make sure your `hashCode` method uses the same data field(s) as your `equals` method

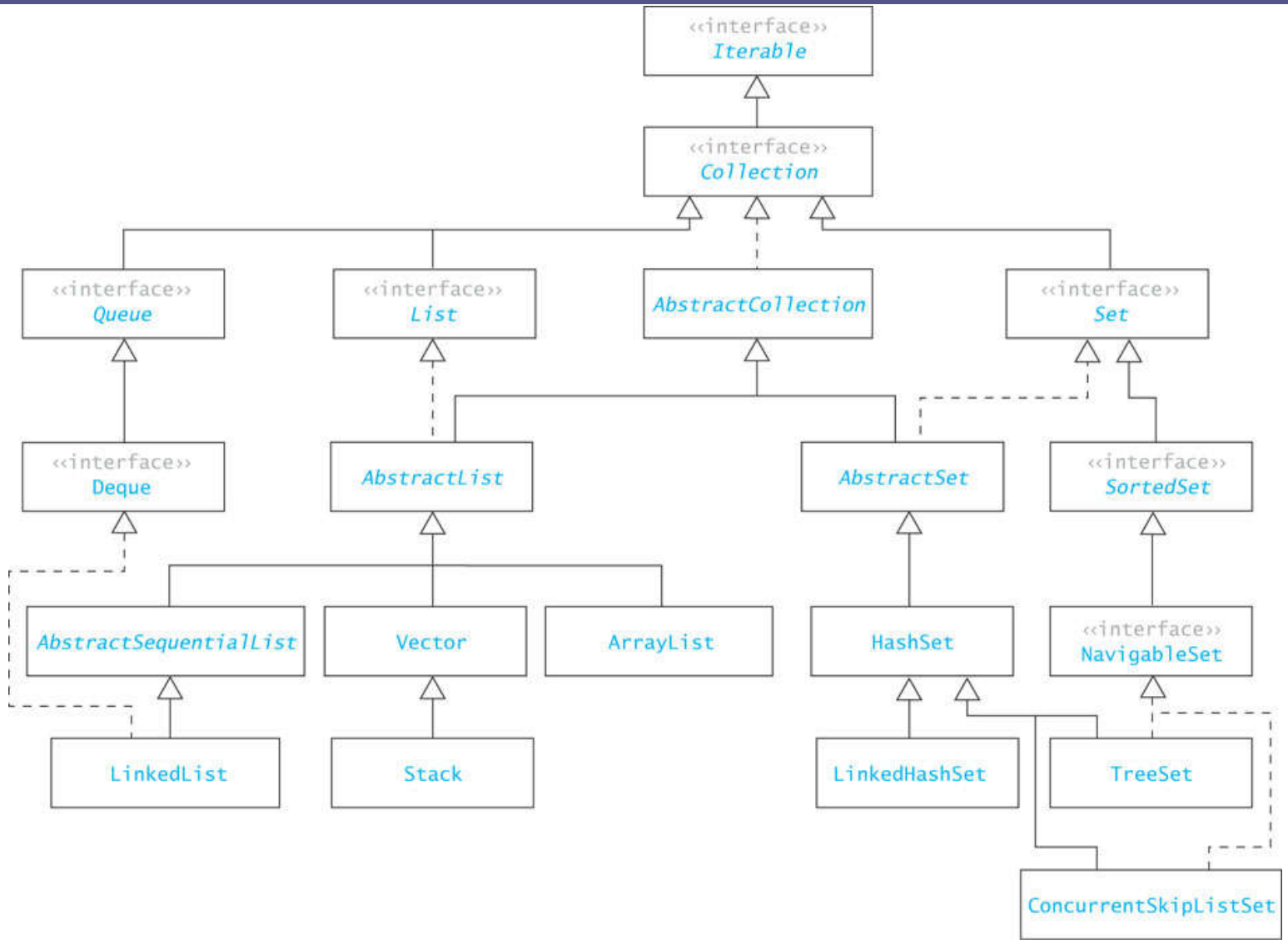
Navigable Sets and Maps

Section 7.7

`SortedSet` and `SortedMap`

- Java 5.0's `SortedSet` interface extends `Set` by providing the user with an ordered view of the elements with the ordering defined by a `compareTo` method
- Because the elements are ordered, additional methods can return the first and last elements and define subsets
- The ability to define subsets was limited because subsets always had to include the starting element and exclude the ending element
- `SortedMap` interface provides an ordered view of a map with elements ordered by key value

The `Collection` Framework



`NavigableSet` and `NavigableMap`

- Java 6 added `NavigableSet` and `NavigableMap` interfaces as extensions to `SortedSet` and `SortedMap`
- Java retains `SortedSet` and `SortedMap` for compatibility with existing software
- The new interfaces allow the user to specify whether the start or end items are included or excluded
- They also enable the user to specify a subset or submap that is traversable in the reverse order

NavigableSet Interface

Method	Behavior
`E ceiling(E e)`	Returns the smallest element in this set that is greater than or equal to `e`, or `null` if there is no such element.
`Iterator<E> descendingIterator()`	Returns an `iterator` that traverses the `Set` in descending order.
`NavigableSet<E> descendingSet()`	Returns a reverse order view of this set.
`E first()`	Returns the smallest element in the set.
`E floor(E e)`	Returns the largest element that is less than or equal to `e`, or `null` if there is no such element.
`NavigableSet<E> headset(E toEl, boolean incl)`	Returns a view of the subset of this set whose elements are less than `toEl`. If `incl` is `true`, the subset includes the element `toEl` if it exists.
`E higher(E e)`	Returns the smallest element in this set that is strictly greater than `e`, or `null` if there is no such element.
`Iterator<E> iterator()`	Returns an iterator to the elements in the set that traverses the set in ascending order.
`E last()`	Returns the largest element in the set.
`E lower(E e)`	Returns the largest element in this set that is strictly less than `e`, or `null` if there is no such element.
`E pollFirst()`	Retrieves and removes the first element. If the set is empty, returns `null`.
`E pollLast()`	Retrieves and removes the last element. If the set is empty, returns `null`.
`NavigableSet<E> subSet(E fromEl, boolean fromIncl, E toEl, boolean toIncl)`	Returns a view of the subset of this set that ranges from `fromEl` to `toEl`. If the corresponding `fromIncl` or `toIncl` is `true`, then the `fromEl` or `toEl` elements are included.
`NavigableSet<E> tailSet(E fromEl, boolean incl)`	Returns a view of the subset of this set whose elements are greater than `fromEl`. If `incl` is `true`, the subset includes the element `fromE` if it exists.

`NavigableSet` **Interface** (cont.)

Using a NavigableSet

```
public static void main(String[] args) {
    // Create and fill the sets
    NavigableSet<Integer> odds = new TreeSet<Integer>();
    odds.add(5); odds.add(3); odds.add(7); odds.add(1); odds.add(9);
    System.out.println("The original set odds is " + odds);
    NavigableSet b = odds.subSet(1, false, 7, true);
    System.out.println("The ordered set b is " + b);
    System.out.println("Its first element is " + b.first());
    System.out.println("Its smallest element >= 6 is " + b.ceiling(6));
}
```

Listing 7.13 illustrates the use of a NavigableSet. The output of this program consists of the lines:

```
The original set odds is [1, 3, 5, 7, 9]
The ordered set b is [3, 5, 7]
Its first element is 3
Its smallest element >= 6 is 7
```

`NavigableMap` Interface

Method	Behavior
`Map.Entry<K, V> ceilingEntry(K key)`	Returns a key-value mapping associated with the least key greater than or equal to the given key, or `null` if there is no such key.
`K ceilingKey(K key)`	Returns the least key greater than or equal to the given key, or `null` if there is no such key.
`NavigableSet<K> descendingKeySet()`	Returns a reverse-order `NavigableSet` view of the keys contained in this map.
`NavigableMap<K, V> descendingMap()`	Returns a reverse-order view of this map.
`NavigableMap<K, V> headMap(K toKey, boolean incl)`	Returns a view of the submap of this map whose keys are less than `toKey`. If `incl` is `true`, the submap includes the entry with key `toKey` if it exists.
`NavigableMap<K, V> subMap(K fromKey, boolean fromIncl, K toKey, boolean toIncl)`	Returns a view of the submap of this map that ranges from `fromKey` to `toKey`. If the corresponding `fromIncl` or `toIncl` is `true`, then the entries with key `fromKey` or `toKey` are included.
`NavigableSet<E> tailMap(K fromKey, boolean fromIncl)`	Returns a view of the submap of this map whose elements are greater than `fromKey`. If `fromIncl` is `true`, the submap includes the entry with key `fromKey` if it exists.
`NavigableSet<K> navigableKeySet()`	Returns a `NavigableSet` view of the keys contained in this map.

Classes `TreeMap` and `TreeSet`

- Besides `HashMap` and `HashSet`, the Java Collections Framework provides classes `TreeMap` and `TreeSet`
- `TreeMap` and `TreeSet` use a Red-Black tree, which is a balanced binary tree (introduced in Chapter 9)
- Search, retrieval, insertion and removal are performed better using a hash table (expected O(1)) than using a binary search tree (expected O(log *n*))
- However, a binary search tree can be traversed in sorted order while a hash table cannot be traversed in any meaningful way
- In the previous example of building an index for a term paper, use of a `TreeMap` allows the list to be displayed in alphabetical order