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)

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

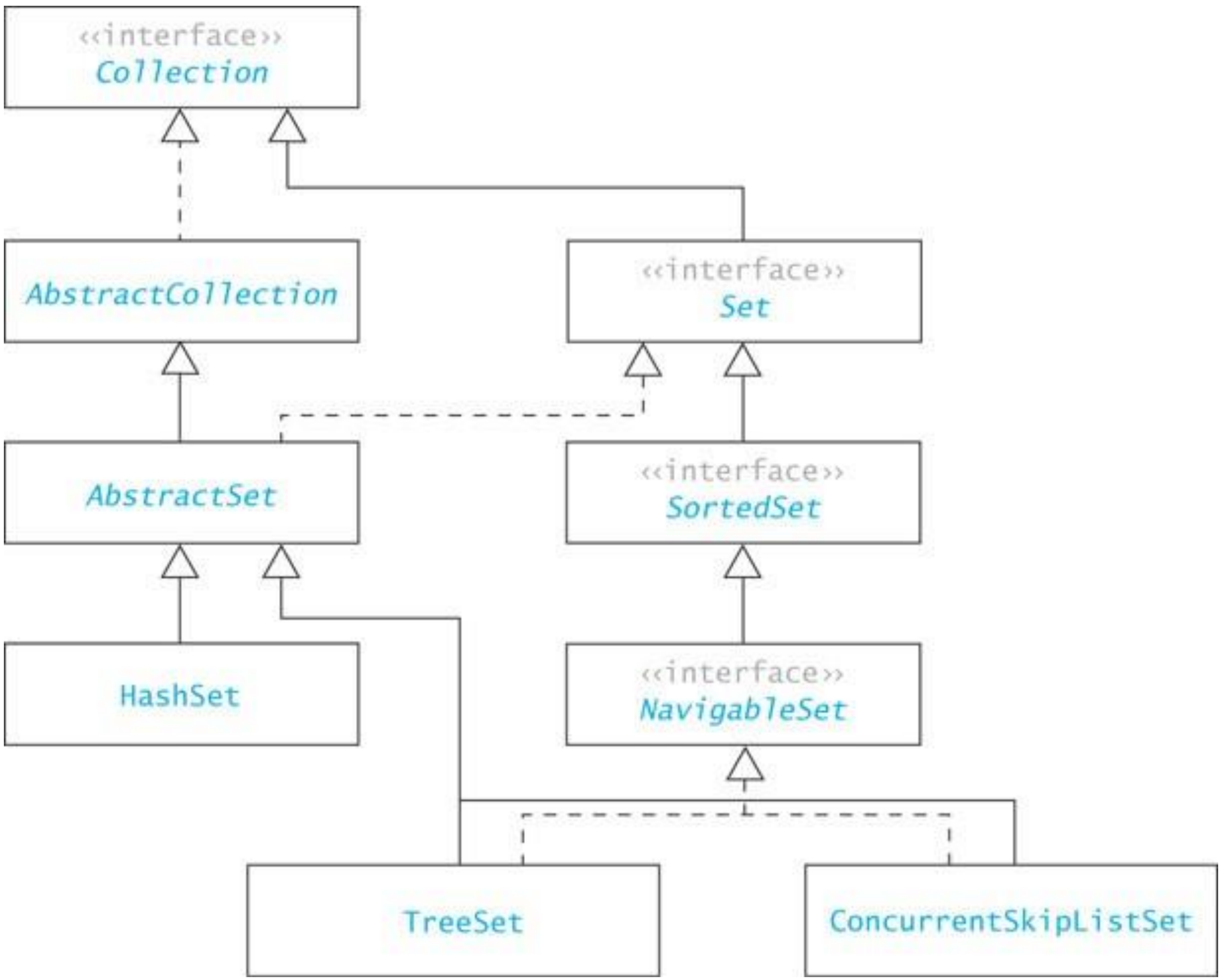
Introduction (cont.)

- 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: $\{1, 3, 5, 7\} \cup \{2, 3, 4, 5\}$ 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: $\{1, 3, 5, 7\} \cap \{2, 3, 4, 5\}$ 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: $\{1, 3, 5, 7\} \subset \{1, 2, 3, 4, 5, 7\}$ 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

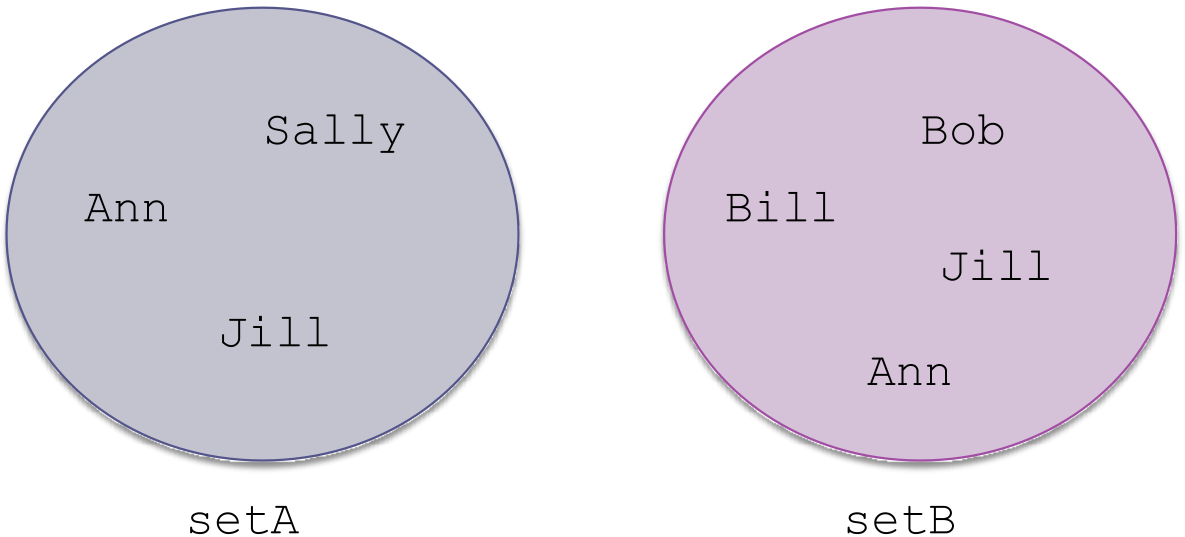
The `Set` Interface and Methods(cont.)

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

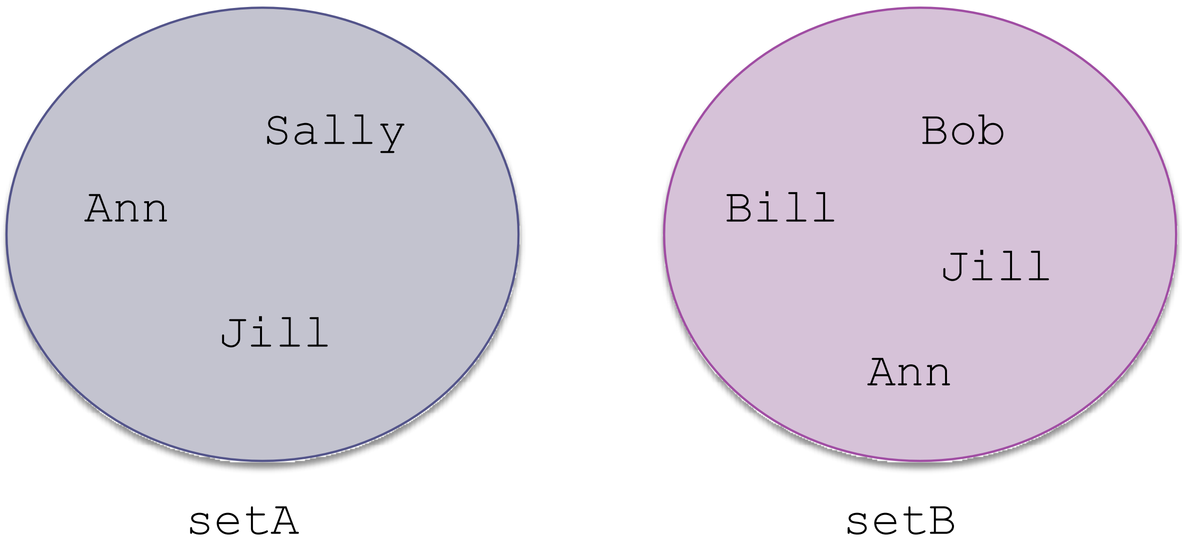
The `Set` Interface and Methods(cont.)

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.)

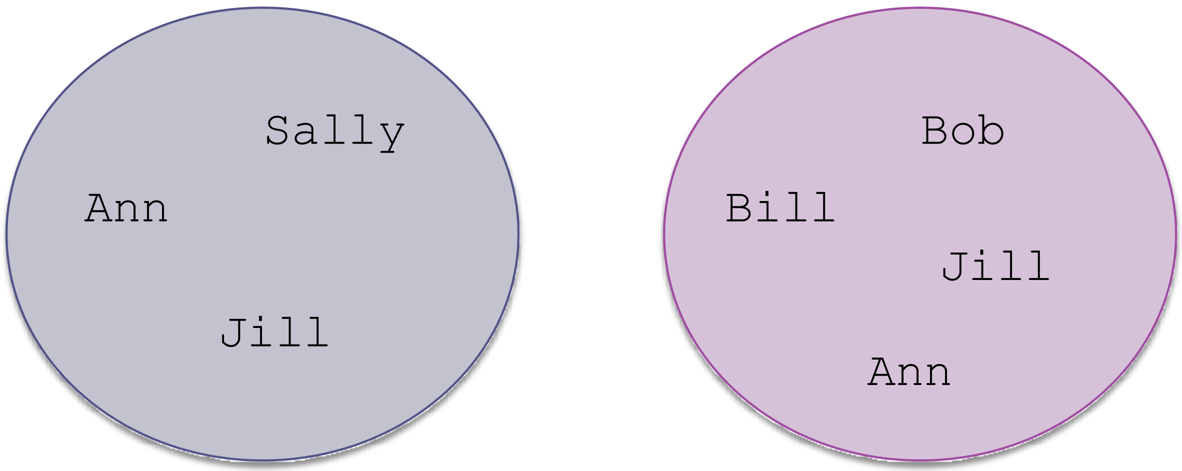


The `Set` Interface and Methods(cont.)




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

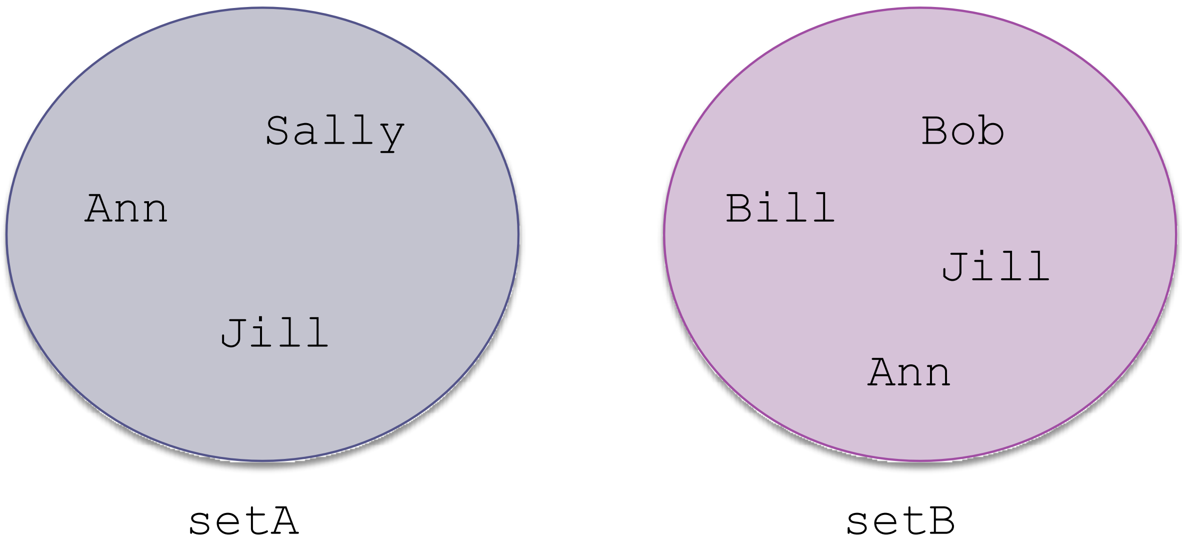
The `Set` Interface and Methods(cont.)



setA setB

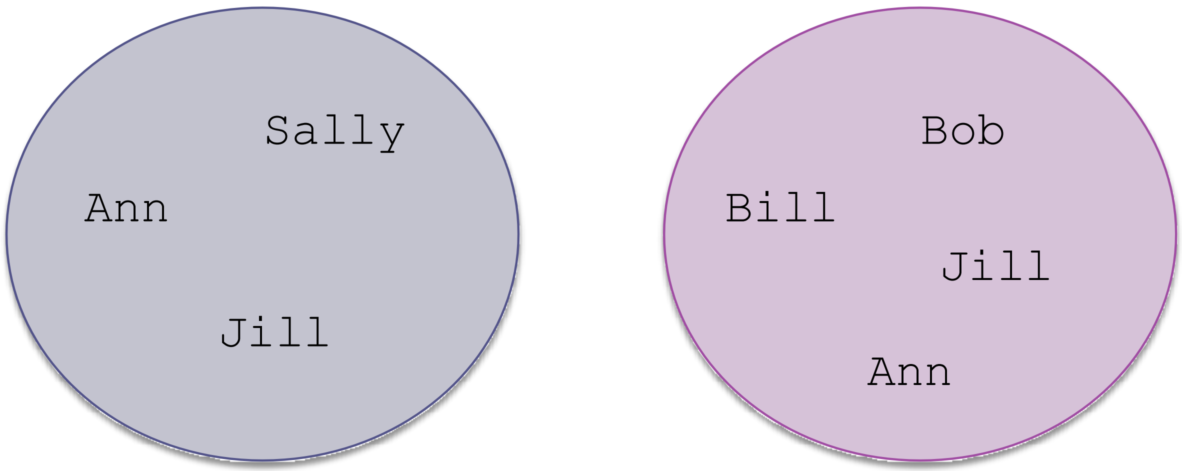
```
setA.addAll(setB); // Perform Union

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 . . .

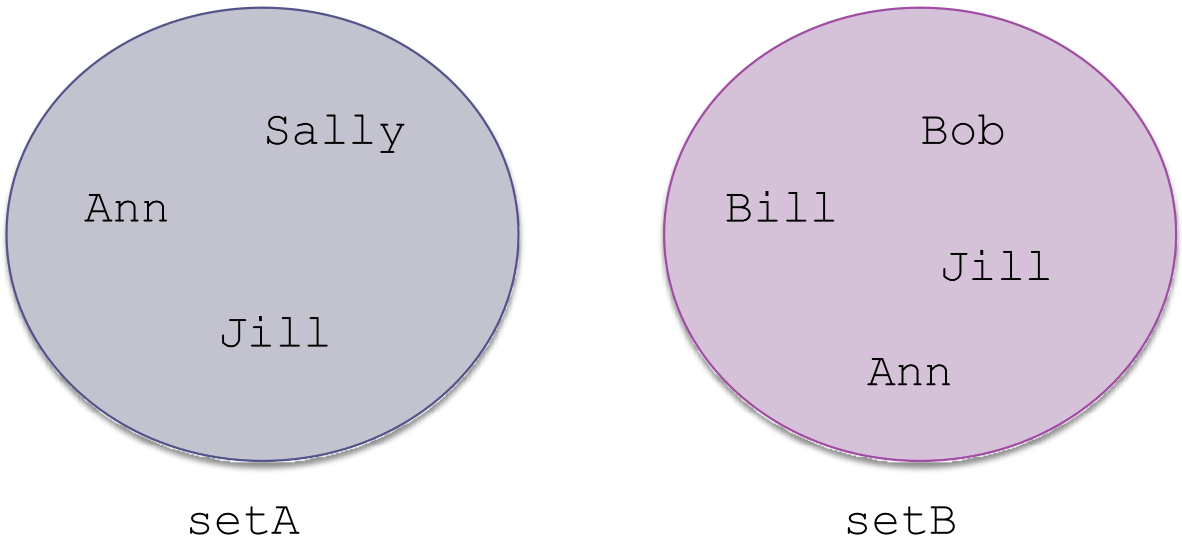
The `Set` Interface and Methods(cont.)



setA setB

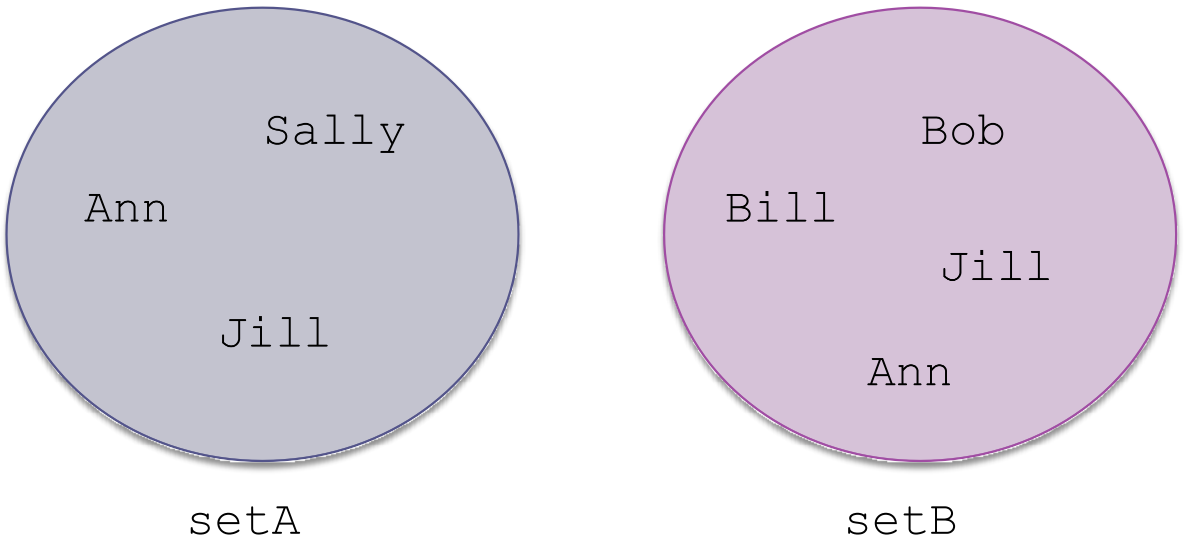
```
setACopy.retainAll(setB);
```

The `Set` Interface and Methods(cont.)




```
setACopy.retainAll(setB); // Perform intersection

System.out.println(setACopy);


Outputs:
[Jill, Ann]
```

The Set Interface and Methods(cont.)




```
setACopy.removeAll(setB); // Perform difference

System.out.println(setACopy);


Outputs:
[Sally]
```

The `Set` Interface and Methods(cont.)

- Listing 7.1 (Illustrating the Use of Sets; pages 365-366)
- Refer : UseofSets.java from w4l7.api pacakege

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
  …
}
```

Comparison of sets

HashSet Vs. LinkedHashSet Vs. TreeSet

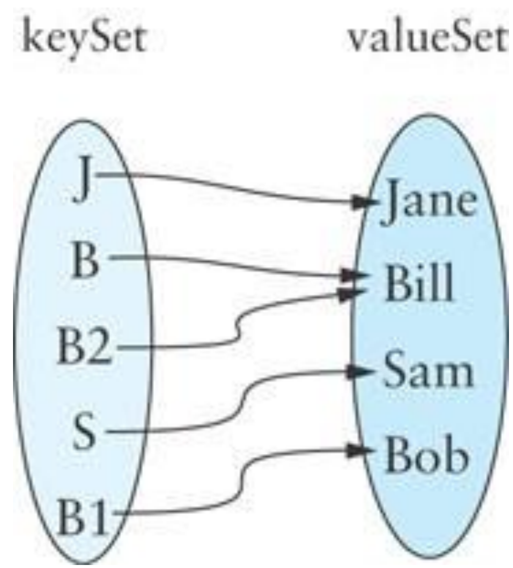
	HashSet	LinkedHashSet	TreeSet
How they compare the elements?	HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements.	LinkedHashSet also uses equals() and hashCode() methods to compare the elements.	TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn't use equals() and hashCode() methods for comparison of elements.
Null elements	HashSet allows maximum one null element.	LinkedHashSet also allows maximum one null element.	TreeSet doesn't allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException.
Memory Occupation	HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements.	LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements.	TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap.
When To Use?	Use HashSet if you don't want to maintain any order of elements.	Use LinkedHashSet if you want to maintain insertion order of elements.	Use TreeSet if you want to sort the elements according to some Comparator.

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

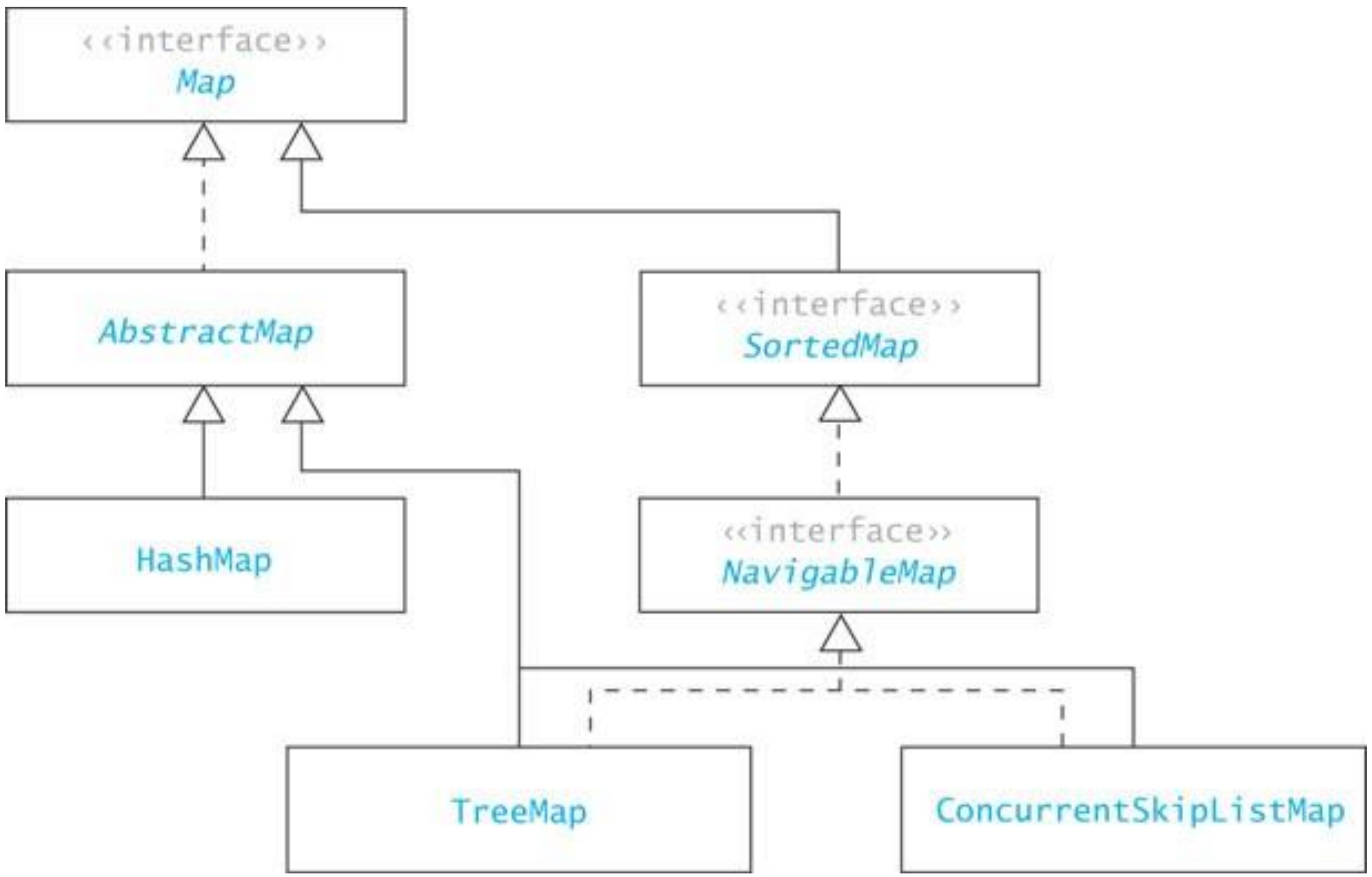
```
V.get (Object key)
V.put (K key, V value)
```

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

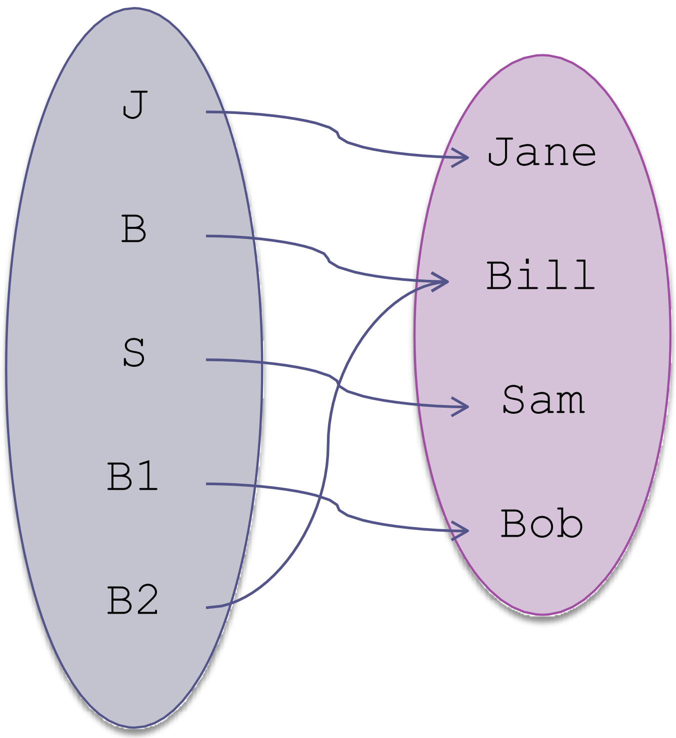
Method	Behavior
V get(Object key)	Returns the value associated with the specified key. Returns **null** if the key is not present.
boolean isEmpty()	Returns **true** if this map contains no key-value mappings.
V put(K key, V value)	Associates the specified value with the specified key in this map (optional 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");
```

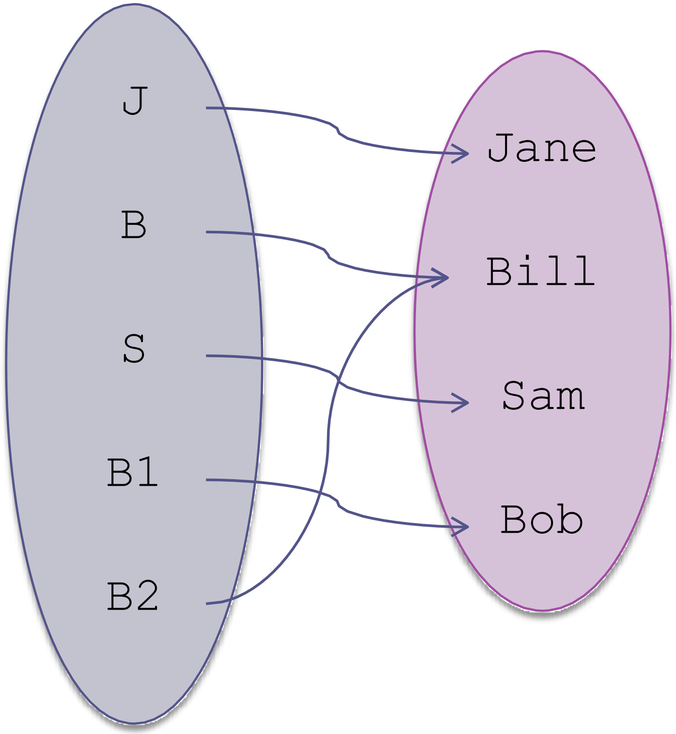


Map **Interface (cont.)**

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

returns:

```
"Bob"
```



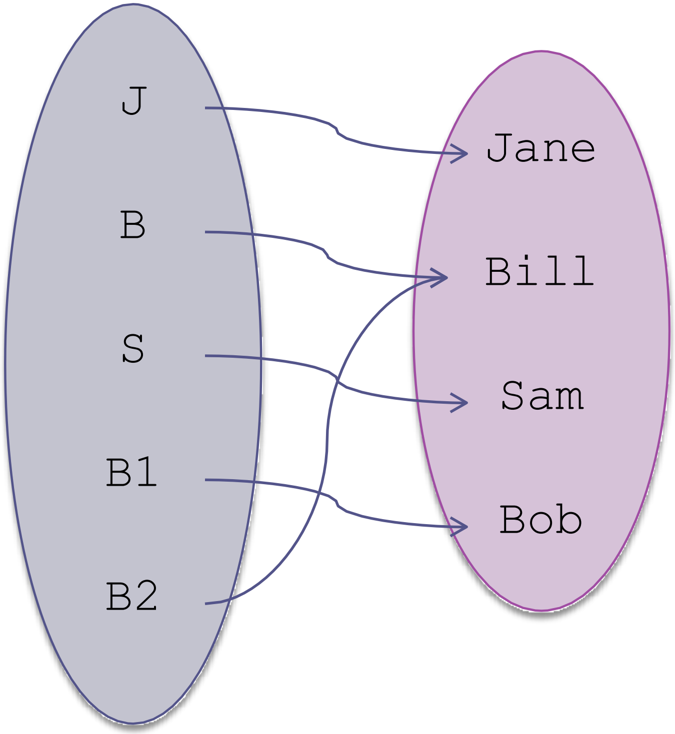
Map **Interface** (cont.)

`aMap.get("Bill")`

returns:

`null`

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



Predefined Hash Tables(API)

- java.util.HashMap
  - Allows null keys
  - Allows null values
  - Not synchronized for safe multithreading
- java.util.Hashtable
  - Does not allow null keys
  - Does not allow null values
  - Synchronized for safe multithreading

The above two classes are roughly equivalent except for the differences noted above.

Predefined Library for Hash Concepts

HashSet : Hash table implementation of the Set interface.

- HashSet does not allow duplicate values but allow null value.
- It provides add method rather put method.
- HashSet can be used where you want to maintain a unique list.

HashMap : Hash table implementation of the Map interface.

- Like Hashtable it also accepts key value pair.
- It allows null for both key and value.
- Does not allow duplicate keys.
- It is unsynchronized. So come up with better performance
- Iterator is used to Iterate.
- HashMap is the subclass of the AbstractMap class.

Hashtable : Hash table implementation of the Map interface.

- Hashtable is basically a datastructure to retain values of key-value pair.
- Does not allow duplicate keys.
- It didn't allow null for both key and value. You will get NullPointerException if you add null value.
- It is synchronized. Only one thread can access in one time. Useful in Multithreaded Environment.
- Enumeration is used to Iterate.
- Hashtable is a subclass of Dictionary Abstract class.
- The Dictionary class is the abstract parent of a class, which maps keys to values.

Refer : w4l7.api pacakege

Hash Tables

Section 7.3

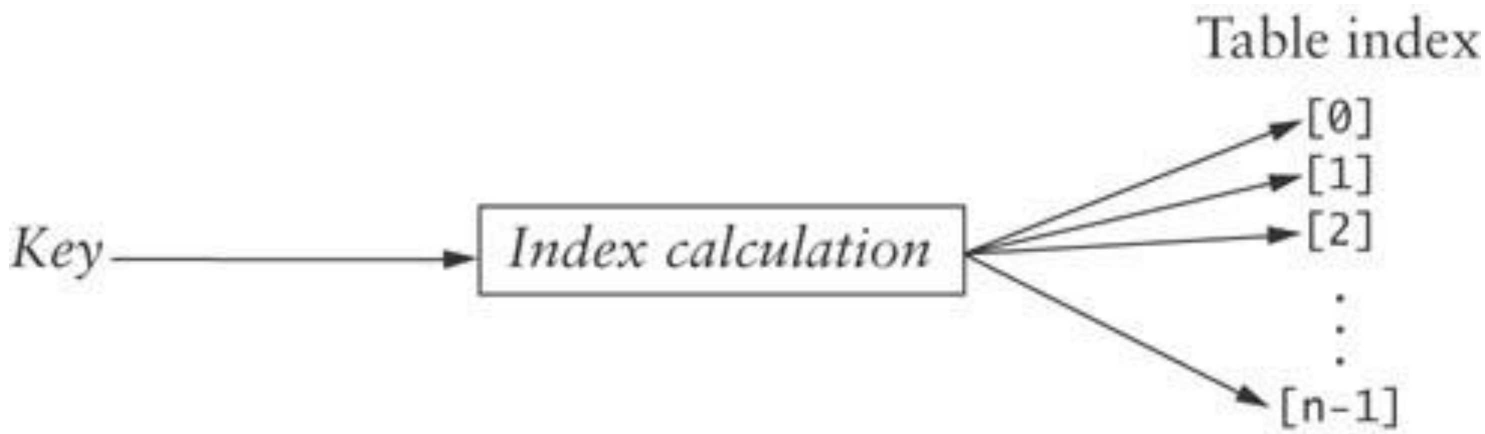
Day - 2

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



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
```

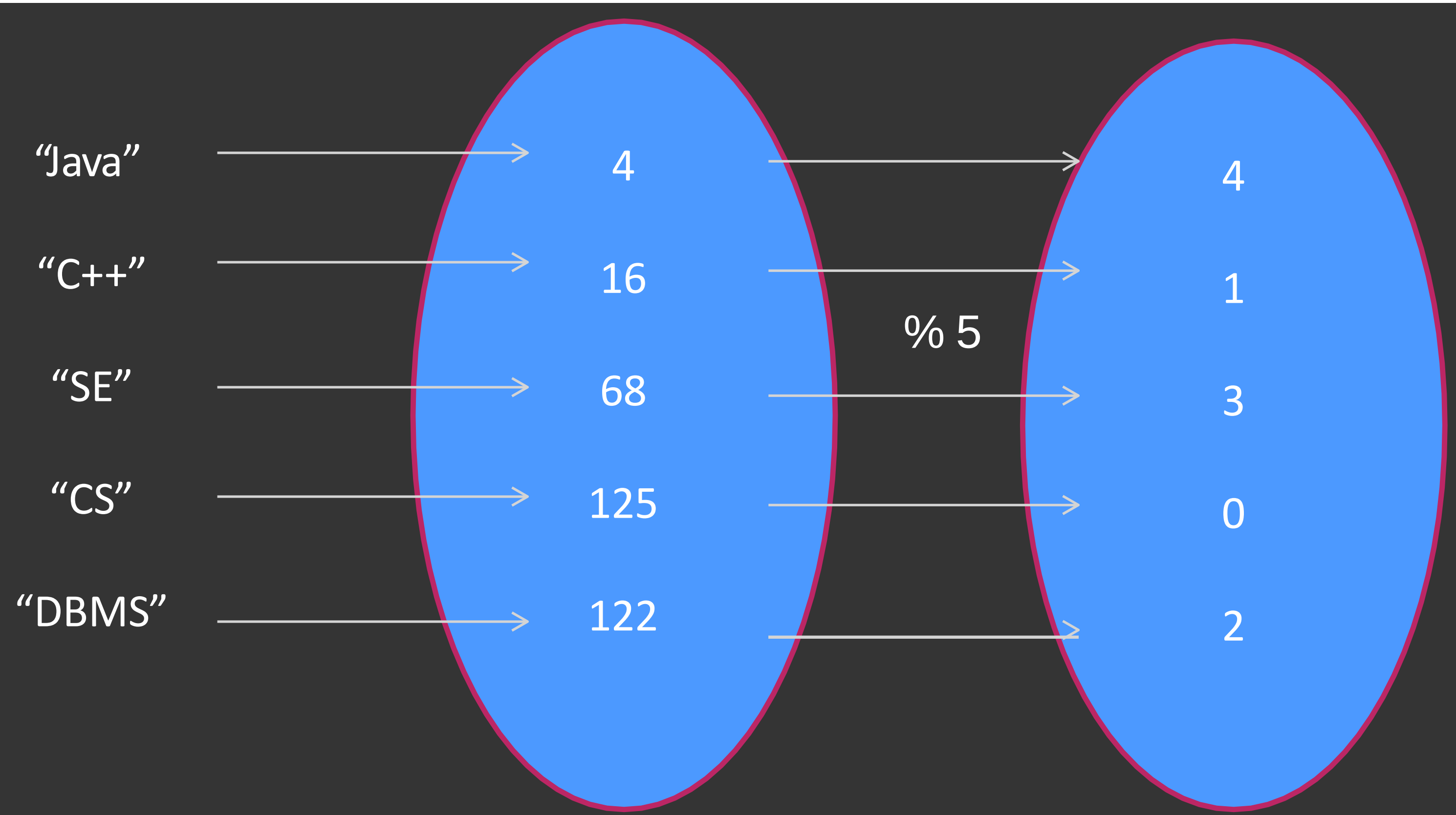
is proportional to how full the table is

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

Example

Data item   Key Value   Table index = Key Value % table size



CS	**C++**	**DBMS**	**SE**	**JAVA**
0	1	2	3	4

Collisions

Two values can hash to the same array index, resulting in collision. It can be resolved using

- **Open Addressing:** Search the array in some systematic way for an empty cell and insert the new item there if collision occurs.
  - **Linear Probing** (Search sequentially for vacant cells until find an empty)
  - **Quadratic Probing** (In quadratic probing, probes go to x+1, x+4, x+9, and so on)
  - **Double Hashing** (Double the sequence with constant factor)
- **Separate chaining:** Create an array of linked list, so that the item can be inserted into the linked list if collision occurs.

Try Animation

- Try the Hash Table animation using :

  http://iswsa.acm.org/mphf/openDSAPerfectHashAnimation/perfectHashAV.html

  http://www.cs.armstrong.edu/liang/animation/web/LinearProbing.html

  https://yongdanielliang.github.io/animation/web/SeparateChaining.html

  https://yongdanielliang.github.io/animation/web/QuadraticProbing.html

  https://liveexample.pearsoncmg.com/dsanimation/DoubleHashingeBook.html

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

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

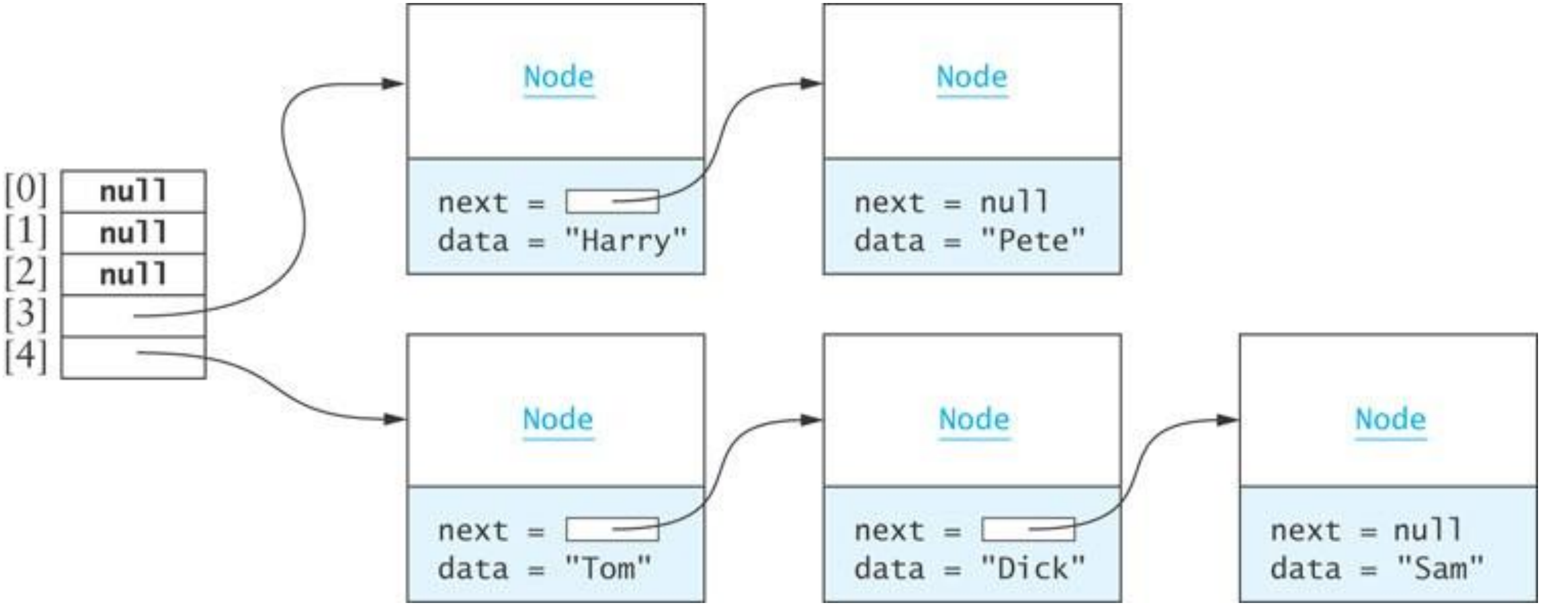
1. Compute the index by taking the item's `hashCode() % table.length`.
2. `if table[index]` is `null`
3. The item is not in the table.
4. `else if table[index]` is equal to the item
5. The item is in the table.

   `else`

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

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
- Refer user Implementation of hash table for linear probing : HashTableApp.java

Implementation Considerations for Maps and Sets from API

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()
```

Creating a Hash Value from Object Data (From Effective Java, 2nd Ed.)

- You are trying to define a hash value for each instance variable of a class. Suppose f is such an instance variable.

- If f is boolean, compute (f ? 1 : 0)
- If f is a byte, char, short, or int, compute (int) f
- If f is a long, compute (int) (f ^ (f >>> 32))
- If f is a float, compute Float.floatToIntBits(f)
- If f is a double, compute Double.doubleToLongBits(f) which produces a long f1, then return (int) (f1 ^ (f1 >>> 32))
- If f is an object, compute f.hashCode()

Formula for creating your hashCode function

Step 1. Use the table above to produce a temporary hash of each variable in your class.

Example: You have variables u, v, w. Produce (using the chart above) temporary hash vals hash_u, hash_v, hash_w.

Step 2. Combine these temporary hashes into a final hashCode that is to be returned

Example:

```
int result = 17;
result += 31 * result + hash_u;
result += 31 * result + hash_v;
result += 31 * result + hash_w;
return result;
```

Combining HashCodes of Instance Variables to Produce a Final HashCode (Modern Approach)

- Use the following method in the `Objects` class to compute hash code.

```
public static int hash(Object... values)
```

- For example, if an object has three fields, x, y, and z, we could compute hash code in this way:

```
@Override
public int hashCode() {
  return Objects.hash(x, y, z);
}
```

- Make sure your `hashCode` method uses the same data field(s) as your `equals` method.
- Refer : w4l7.api.NeedOverride

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

Additional Applications of Maps

Section 7.6

Cell Phone Contact List

- Problem
  - A cell phone manufacturer wants a Java program to maintain of list of contacts (phone numbers) for each cell phone owner
  - The manufacturer has provided the software interface:

Method	Behavior
`List<String> addOrChangeEntry(String name, List<String> numbers)`	Changes the numbers associated with the given name or adds a new entry with this name and list of numbers. Returns the old list of numbers or `null` if this is a new entry.
`List<String> lookupEntry(String name)`	Searches the contact list for the given name and returns its list of numbers or `null` if the name is not found.
`List<String> removeEntry(String name)`	Removes the entry with the specified name from the contact list and returns its list of numbers or `null` if the name is not in the contact list.
`void display();`	Displays the contact list in order by name.

Cell Phone Contact List (cont.)

- Analysis
  - A map will associate the name (the key) with a list of phone numbers (value)
  - Implement `ContactListInterface` by using a `Map<String, List<String>>` object for the data type

Cell Phone Contact List (cont.)

- Design

```
public class MapContactList
                      implements ContactListInterface {


   Map<String, List<String>> contacts =
                 new TreeMap<String,  List<String>>();
   . . .
}
```

Cell Phone Contact List (cont.)

- Implementation: writing the required methods using the `Map` methods is straightforward

Cell Phone Contact List (cont.)

- Testing
  - Write a main function that creates a new `MapContactList` object
  - Apply the `addOrChangeEntry()` method several times with new names and numbers to build the initial contact list
  - Display and update the list to verify that all methods are functioning correctly

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

`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.

Application of a `NavigableMap` Interface

- `computeAverage` computes the average of the values defined in a `Map`
- `computeSpans` creates a group of submaps of a `NavigableMap` and passes each submap to `computeAverage`
- Given a `NavigableMap` in which the keys represent years and the values are some statistics for the year, we can generate a table of averages covering different periods

Application of a `NavigableMap` Interface (cont.)

- Example:

  Given a map of tropical storms representing the number of tropical storms from 1960 through 1969

  ```
  List<Number> stormAverage = computeSpans(storms,2)
  ```

  Calculates the average number of tropical storms for each successive pair of years

  Refer : NavigableMapDemo.java

Guidelines for Use of Common Data Structures

1. **Array List**
   - <u>Use When</u>: Main need for a list is random access reads, relatively infrequent adds (beyond initial capacity) and/or number of list elements is known in advance. Sorting routines run faster on an ArrayList than on a Linked List.
   - <u>Avoid When</u>: Many inserts and removes will be needed and/or when many adds expected, but number of elements unpredictable. Also: Maintaining data in sorted order is very inefficient.

2. **Linked List**
   - <u>Use When</u>: Insertions and deletions are frequent, and/or many elements need to be added, but total number is unknown in advance. There is no faster data structure for repeatedly adding new elements than a Linked List (since elements are always added to the front).
   - <u>Avoid When</u>: There is a need for repeated access to ith element as in binary search – random access is not supported.

Guidelines for Use of Common Data Structures

3. **Binary Search Tree**
   - Use When: Data needs to be maintained in sorted order. Faster than Linked Lists for insertions and deletions, but ordinary adds are slower. Provides very fast search for keys.
   - Avoid When: The extra benefit of keeping data in sorted order is not needed and rapid read access is needed (Array List provides faster read access by index and hashtables provide faster read access by key)
4. **Hashtable/Hashmap**
   - Use When: Random access to objects is needed but array indexing is not practical. Provides fastest possible insertion and deletion (faster than BST's).
   - Avoid When: The order of data must be preserved (example: you want to find all employees whose salaries are in the range 60000..65000) or "find Max" or "find Min" operations are needed.
5. **Sets**
   - Use When: Duplicates should be disallowed, and there is no need for rapid lookup of individual set elements. Example: `keySet()` in `HashMap` returns a `Set`. To order the elements, use `TreeSet`.