The Java Collection Framework: Interfaces, Classes, and Algorithms
What is a Framework?

- “A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes.”
  - “Designing Reuseable Classes”, Johnson and Foote 1988

- “A framework is a set of prefabricated software building blocks that programmers can use, extend, or customize for specific computing solutions.”
  - “Building Object-Oriented Frameworks” Taligent/IBM
More than just a Good O/O Implementation?

- “An abstract class is a design for a single object. A framework is the design of a set of objects that collaborate to carry out a set of responsibilities. Thus frameworks are larger scale designs than abstract classes. Frameworks are a way to reuse high-level design.”
  - “Reusing Object-Oriented Designs”, Johnson and Russo, 1991

- “Frameworks are not simply collections of classes. Rather, frameworks come with rich functionality and strong wired-in interconnections between object classes that provide an infrastructure for the developer.”
  - Taligent, 1993
Why Write a Framework?

- Frameworks decrease programmer effort
  - Make it easier to use different APIs together (Interoperability)
  - Reduces the effort required to learn APIs
  - Reduces the effort required to design and implement APIs
  - Increases code reliability
  - Fosters software reuse

- Programming with Components
  - Finding functionality
  - Learning the structures and dependencies among components
  - Reading more than you write but more than you are used to...?
What is a Collection?

A collection (sometimes called a container) is an object that groups multiple elements into a single unit.

- Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.
- Collections typically represent data items that form a natural group, a card hand, a mail folder, a telephone directory...

Recall the difference between ADTs and Data Structures:

- **ADT**: collection of data and a set of operations on that data
- **Data structure**: construct within a programming language that stores a collection of data

<table>
<thead>
<tr>
<th>ADT</th>
<th>Data Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td><strong>Storage Space</strong></td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td><strong>Functions</strong></td>
</tr>
<tr>
<td>Data Items: Logical Form</td>
<td>Data Items: Physical Form</td>
</tr>
</tbody>
</table>
Given a Container, You ...

- Put an object in
- Take an object out
- Ask about a specific object
  - Is it in the container?
  - Is an equivalent object in the container?
  - How many times?
- Retrieve an object by a key value
- Iterate over everything in the container

- Putting ADT theory into practice
The Java Collections Framework

- A new framework in Java 2: Provide a basic set of “Object Containers”
  - Core data structures for everyday coding
- Predecessors:
  - C++’s Standard Template Library and ObjectSpace’s JGL
  - JDK 1.02: Vector, Hashtable, and Enumeration
- The Java Collections Framework provides:
  - Interfaces: abstract data types representing collections
  - Implementations: concrete implementations of the collection interfaces
  - Algorithms: methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces

- Core Interfaces:
  - Collection
  - Set
  - List
  - Map
  - SortedSet
  - SortedMap

- Utility Interfaces
  - Comparator
  - Iterator

- Utility Classes
  - Collections
  - Arrays
## Types of Collections

- **Duplicates**
  - Allowed
  - Not allowed

<table>
<thead>
<tr>
<th>Ordered</th>
<th>Not ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>Multiset (Bag)</td>
</tr>
<tr>
<td>Map</td>
<td>Set</td>
</tr>
</tbody>
</table>

- Collections can be classified by two parameters:
  - Is duplicates allowed?
  - Are the elements ordered?
The Java Collections Framework

Abstract Classes
- Abstract Collection
  (is a collection)
- Abstract List
  (is a List)
- Abstract Set
  (is a Set)
- Abstract Map
  (is a Map)
- AbstractSequential List
- ArrayList (is a Clonable, List, Serializable)
- HashSet (is a Clonable, Serializable, Set)
- TreeSet (is a Clonable, Serializable, SortedSet)
- HashMap (is a Clonable, Map, Serializable)
- TreeMap (is a Clonable, Serializable, SortedMap)
- WeakHashMap (is a Map)
- LinkedList (is a Clonable, List, Serializable)

Interfaces
- Collection
- List
- Set
- SortedSet
- Comparator
- Iterator
- List Iterator
- Map
- SortedMap
- Map.Entry
The Collection interface is a group of objects, with duplicates allowed

- Set extends Collection but forbids duplicates
- List extends Collection and allows duplicates and positional indexing
- Map extends neither Set nor Collection and forbids duplicates

Simple Container

Collection

Set

SortedSet

List

SortedMap

Keyed Access

Generalized Arrays

No Duplicates
The Main Implementations

- Collection
  - Abstract Collection
    - SortedSet
      - TreeSet
      - HashSet
    - AbstractSet
      - AbstractList
    - List
      - LinkedList
      - ArrayList
  - Set
  - Map
    - AbstractMap
      - HashMap
    - SortedMap
      - TreeMap
Collection versus Map

- A Collection is a group of objects
  - Designed for access via Iterators
    - You can add, remove, lookup isolated items in the collection
  - Easily extended for sorting elements

- Examples:
  - distinct words in a novel
  - reserved keywords in java
  - students taking HY252

- A Map (mathematical function) associates a value with each member key in its domain
  - set of pairs (key-value), each pair representing a “one-directional mapping” from one set to another
  - Designed for providing associative access to values stored by key
    - The collection operations are available but they work with a key-value pair instead of an isolated element
    - Listing all (key,value) pairs requires using an iterator on the key set (domain of the map)

- Examples
  - A map of keys to database records
  - A dictionary (words mapped to meanings)
Collection versus Map

- Map keeps association between key and value objects.
- Every key in a map has a unique value.
- A value may be associated with several keys.

Keys
- Vassilis
- Dimitris
- Grigoris
- Giorgos

Values
Set versus List

- Sets are faithful to their mathematical definition
- No methods added to Collection but simply enforces “no duplicates” rule using the equals method
  - More formally, sets contain no pair of elements $e_1$ and $e_2$ such that $e_1.equals(e_2)$, and at most one null element
- Access via Iterators
- Examples
  - The set of uppercase letters ‘A’ through ‘Z’
  - The set of nonnegative integers \{ 0, 1, 2, … \}
  - The empty set {}  
- Lists are sequential structures
- Duplicates allowed
  - More formally, lists typically allow pairs of elements $e_1$ and $e_2$ such that $e_1.equals(e_2)$, and they typically allow multiple null elements
  - It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare
- Implements get and remove methods using integer index
ArrayList versus LinkedList

- **ArrayList** is a wrapper around an array (resizes when it gets full)
  - Implements all optional list operations, and permits all elements, including null
  - This class provides methods to manipulate the size of the array that is used to store the list
- Insertion to the front or middle is expensive (O(n))
- Provides fast iterator and get(int) methods

- **LinkedList** is a classic doubly linked list
  - Implements all optional list operations, and permits all elements, including null
  - Provides uniformly named methods to get, remove and insert an element at the beginning and end of the list (i.e., they can be used as a stack, queue, or double-ended queue (deque))
- Constant time insertion and removal (O(1)) anywhere in the list
  - Slower iterator
  - get(int) very slow
    - implemented by iterating
ArrayList versus LinkedList

- **ArrayList implementation** using simple, growing, possibly shrinking, arrays

- **LinkedList implementation**

  ![LinkedList diagram]

  - list
  - head
  - prev
  - tail
Trees and Hashes

- **Trees** provide a **two dimensional organization of data**
  - Binary search trees (BST) are binary trees that impose an ordering on the elements
  - **TreeMap** is a **red-black tree implementation of the Map interface**
    - a semi-balanced BST
    - the keys are kept in a TreeSet
  - TreeMap underlies TreeSet as well
  - Fast iteration through values
    - TreeMap keys are stored according to the ordering of keys e.g., we would access the names of a TreeMap in alphabetical order
  - get/put operations are slower \((\log(n) + \text{rebalancing cost)}) but faster than ArrayList

- Hashing is the process of transforming a key into an array index
  - \(h: \text{key} \rightarrow \text{hashvalue} \rightarrow \text{index}\)
  - **HashMap** is a **hash table-based implementation of the Map interface**
    - uses a table containing a **singly-linked list** of elements whose key hash to the index of that particular location
  - HashMap underlies HashSet as well
  - Constant time get/put \((O(1)))
    - hashing is very fast if the **load factor** is not close to 100%
    - resize hash table when the load factor gets too large
  - Iteration through values is slow
Trees and Hashes

- TreeSet implements Set as a Binary Search Tree
- HashSet implements Set as a hash table
Recall Hash Values and Hash Tables

- An algorithm for converting a key to an hash value which is used as an index into a hash table array

```
<table>
<thead>
<tr>
<th>Range of key values</th>
<th>hash value</th>
<th>table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>null</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>null</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>null</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>null</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>7</td>
</tr>
</tbody>
</table>

hash table array
```

```
<table>
<thead>
<tr>
<th>buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj1</td>
</tr>
<tr>
<td>key=15</td>
</tr>
<tr>
<td>Obj3</td>
</tr>
<tr>
<td>key=4</td>
</tr>
<tr>
<td>Obj2</td>
</tr>
<tr>
<td>key=28</td>
</tr>
<tr>
<td>Obj4</td>
</tr>
<tr>
<td>key=2</td>
</tr>
<tr>
<td>Obj5</td>
</tr>
<tr>
<td>key=1</td>
</tr>
</tbody>
</table>
```

null
null
null
null
null
null
null
null
The Collection Interface

Found in the `java.util` package

Optional methods throw `UnsupportedOperationException` if the implementing class does not support the operation.

Bulk operations perform some operation on an entire Collection in a single shot.

The `toArray` methods allow the contents of a Collection to be translated into an array.

```java
// Basic Operations
size():int;
isEmpty():boolean;
contains(Object):boolean;
add(Object):boolean;       // Optional
remove(Object):boolean;    // Optional
iterator():Iterator;

// Bulk Operations
containsAll(Collection):boolean;
addAll(Collection):boolean; // Optional
removeAll(Collection):boolean; // Optional
retainAll(Collection):boolean; // Optional
clear():void;              // Optional

// Array Operations
toArray():Object[];
toArray(Object[]):Object[];
```
The Collection Interface

boolean **add**(Object o): Ensures that this collection contains the specified element

boolean **addAll**(Collection c): Adds all of the elements in the specified collection to this collection

void **clear**(): Removes all of the elements from this collection

boolean **contains**(Object o): Returns true if this collection contains the specified element

boolean **containsAll**(Collection c): Returns true if this collection contains all of the elements in the specified collection

boolean **equals**(Object o): Compares the specified object with this collection for equality

int **hashCode**(): Returns the hash code value for this collection

boolean **isEmpty**(): Returns true if this collection contains no elements.
The Collection Interface

**Iterator iterator()**: Returns an iterator over the elements in this collection

**boolean remove(Object o)**: Removes a single instance of the specified element from this collection, if it is present

**boolean removeAll(Collection c)**: Removes all this collection's elements that are also contained in the specified collection

**boolean retainAll(Collection c)**: Retains only the elements in this collection that are contained in the specified collection

**int size()**: Returns the number of elements in this collection

**Object[] toArray()**: Returns an array containing all of the elements in this collection

**Object[] toArray(Object[] a)**: Returns an array containing all of the elements in this collection whose runtime type is that of the specified array

- **Examples:**
  ```java
  Object[] a = c.toArray();
  String[] a = (String[]) c.toArray(new String[0]);
  ```
Creating and Using a Collection

- Collection is actually an interface
  - Each kind of `Collection` has one or more implementations
  - You can create new kinds of Collections
  - When you implement an interface, you supply all the required methods

- All `Collection` implementations should have two constructors:
  - A no-argument constructor to create an empty `Collection`
  - A constructor with another `Collection` as argument

- All the standard implementations obey this rule, but if you implement your own `Collection` type, this rule cannot be enforced, because an `Interface` cannot specify constructors

- Note that most methods e.g. `boolean containsAll(Collection c);` are defined for any type of `Collection`, and take any type of `Collection` as an argument
  - This makes it very easy to work with different types of Collections
The Set Interface

- A **Set** is a **Collection** that cannot contain duplicate elements
  - **Set** models the mathematical **set abstraction**
- The **Set** interface extends **Collection** and contains no methods other than those inherited from **Collection**
  - It adds the restriction that duplicate elements are prohibited
  - Two **Set** objects are equal if they contain the same elements
- The **bulk operations** perform standard set-algebraic operations:
  - Suppose s1 and s2 are **Sets**
    - s1.**containsAll**(s2): Returns true if s2 is a **subset** of s1
    - s1.**addAll**(s2): Transforms s1 into the **union** of s1 and s2 (The union of two sets is the set containing all the elements contained in either set)
    - s1.**retainAll**(s2): Transforms s1 into the **intersection** of s1 and s2 (The intersection of two sets is the set containing only the elements that are common in both sets)
<table>
<thead>
<tr>
<th>Operation</th>
<th>Mathematical Notation</th>
<th>Java Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>union</strong></td>
<td>$s \leftarrow s \cup t$</td>
<td>$s$.addAll($t$)</td>
</tr>
<tr>
<td><strong>intersection</strong></td>
<td>$s \leftarrow s \cap t$</td>
<td>$s$.retainAll($t$)</td>
</tr>
<tr>
<td><strong>difference</strong></td>
<td>$s \leftarrow s \setminus t$</td>
<td>$s$.removeAll($t$)</td>
</tr>
<tr>
<td><strong>inclusion</strong></td>
<td>$s \supseteq t$</td>
<td>$s$.containsAll($t$)</td>
</tr>
<tr>
<td><strong>insert</strong></td>
<td>$s \leftarrow s \cup {o}$</td>
<td>$s$.add($o$)</td>
</tr>
<tr>
<td><strong>delete</strong></td>
<td>$s \leftarrow s \setminus {o}$</td>
<td>$s$.remove($o$)</td>
</tr>
<tr>
<td><strong>membership</strong></td>
<td>$o \in s$</td>
<td>$s$.contains($o$)</td>
</tr>
<tr>
<td><strong>cardinality</strong></td>
<td>$</td>
<td>s</td>
</tr>
<tr>
<td><strong>emptiness</strong></td>
<td>$s = \emptyset$</td>
<td>$s$.isEmpty()</td>
</tr>
<tr>
<td><strong>make empty</strong></td>
<td>$s \leftarrow \emptyset$</td>
<td>$s$.clear()</td>
</tr>
<tr>
<td><strong>iteration</strong></td>
<td></td>
<td>$s$.iterator()</td>
</tr>
</tbody>
</table>
The List Interface

- A **List** is an ordered **Collection** (sometimes called a **sequence**)
  - In a sequence every object (except the first and last) has a predecessor and a successor, while occupies a unique position in the sequence
  - Lists may contain duplicate elements
- Inherits operations from **Collection** as:
  - **add**(Object) // Append Object to receiver (add to the end of the sequence)
  - **contains**(Object) // Returns TRUE if the receiver contains the Object
  - **remove**(Object) // Remove the first occurrence of Object (adjust rest of the list)
- The **List** interface includes additional operations for:
  - Positional Access
  - Search
  - List Iteration
  - Range-view
- Think of the **Vector** class
The List Interface

// Positional Access
// Returns a copy of the Object located at the index
Object get(int);
// Overwrite contents of list at position of index with Object
Object set(int, Object);   // Optional
// Insert Object at indicated index (adjust the rest list)
void add(int, Object);     // Optional
// Remove Object at indicated index (adjust the rest list)
Object remove(int index);  // Optional
boolean addAll(int, Collection); // Optional

// Search
// Returns the index of the first occurrence of Object –
// returns length of list if the Object is not present
int indexOf(Object);
int lastIndexOf(Object);

// Iteration
// Returns a ListIterator to traverse the referenced container
ListIterator listIterator();
ListIterator listIterator(int);

// Range-view List
subList(int, int):List;
The Map Interface

- Replaces java.util.Dictionary interface
  - An object that maps keys to values
  - Each key can have at most one value
- Ordering may be provided by implementation class, but not guaranteed
- Methods
  - `Set keySet();`
  - `Collection values();`
  - `Set entrySet();` // returns a set view of the mappings
- `Map.Entry`
  - Object that contains a key-value pair
    - `getKey()`, `getValue()`
- Thread safety
  - The collections returned are backed by the map
    - When the map changes, the collection changes
  - Behavior can easily become undefined
    - Be very careful and read the docs closely
The Map Interface

// Basic Operations
Object **put**(Object, Object); //If the map already contains a
// given key it replaces the value associated with that key
Object **get**(Object); // returns the corresponding value
Object **remove**(Object); // removes the pair with that key
boolean **containsKey**(Object);
boolean **containsValue**(Object);
int **size**();
boolean **isEmpty**();

// Bulk Operations
void **putAll**(Map t); //copies one Map into another
void **clear**();

// Collection Views
Set **keySet**();
Collection **values**();
Set **entrySet**();//returns a set of Map.Entry (key-value) pairs
The Map.Entry Interface

- This is a small interface for working with the Collection returned by \texttt{entrySet()}.

```
public interface Entry {
    Object getKey();
    Object getValue();
    Object setValue(Object);
}
```

- Can get elements only from the \texttt{Iterator}, and they are only valid during the iteration.
Map Operations

- **restriction**
  \[ f \leftarrow f \big|_{\text{dom}(f) \setminus \{k\}} \]

- **extension**
  \[ f \leftarrow [k \mapsto o] \cup f \big|_{\text{dom}(f) \setminus \{k\}} \]

- **extension**
  \[ f \leftarrow g \cup f \big|_{\text{dom}(f) \setminus \text{dom}(g)} \]

- **make empty**
  \[ f \leftarrow [] \]

- **lookup**
  \[ f(k) \]

- **defined**
  \[ k \in \text{dom}(f) \]

- **hits**
  \[ \exists k. f(k) = o \]

- **size**
  \[ |\text{dom}(f)| \]

- **emptiness**
  \[ \text{dom}(f) = \emptyset \]

- **domain**
  \[ \text{dom}(f) \]

- **co-domain**
  \[ \{ f(k) | k \in \text{dom}(f) \} \]

- **map pairs**
  \[ \{(k, f(k)) | k \in \text{dom}(f)\} \]

- **f.remove(k)**
- **f.put(k, o)**
- **f.putAll(g)**
- **f.clear()**
- **f.get(k)**
- **f.containsKey(k)**
- **f.containsValue(o)**
- **f.size()**
- **f.isEmpty()**
- **f.keySet()**
- **f.values()**
- **f.entrySet()**
The Iterator Interface

- Created by `Collection.iterator()`, similar to Enumeration
  - Improved method names
  - Allows a `remove()` operation on the current item
- An object implementing the `Iterator` interface generates a series of elements, one at a time
  - A call to the method `iterator()` returns an iterator object and sets it to read the first element in the list.
  - Successive calls to the `next()` method return successive elements of the series
  - See if there are any more elements in the sequence with a call to `hasNext()`
  - The `remove()` method removes from the underlying `Collection` the last element that was returned by `next()`
- Works for non-list collections
- Allows the iterators to have state

```java
boolean hasNext();
Object next();
void remove();
```
Using Iterators

- Replace

```
int counter;
int size = collection.size();
for (counter=0; counter<size; counter++) {
    Object value = collection.get(counter);
    ...
}
```

- with:

```
Iterator i = collection.iterator();
while(i.hasNext()) {
    Object value = i.next();
    ....
}
```

- or even

```
for (Object o: collection) {
    System.out.println(o);
}
```
Using Iterators

- With an iterator one can write generic methods to handle any collection

```java
public class Printer {
    static void printAll (Iterator e) {
        while (e.hasNext())
            System.out.print(e.next() + " ");
        System.out.println();
    }
}

public class FrogPond {
    public static void main(String [] args) {
        ArrayList v = new ArrayList();
        for (int i = 0; i < args[0]; i++)
            v.add(new Frog(i));
        Printer.printAll(v.iterator());
    }
}
```

The method receives an iterator object as its argument with no distinction made about what collection produced it.

Obtain an iterator from the ArrayList and pass it to the Printer.
Removing items via an Iterator

- Iterators can also be used to modify a Collection:

```java
Iterator i = collection.iterator();
while ( i.hasNext() )
{
    if (object.equals(i.next()) )
    {
        i.remove();
    }
}
```
The ListIterator Interface

- A `ListIterator` is produced by any `Collection` implementing the `List` interface
  - interface `ListIterator` extends `Iterator`
  - Created by `List.listIterator()`

- It has an additional constructor that takes an index value as a parameter and returns an iterator set to begin visiting the element at this location on the first call to `next()`
  - `ListIterator(int);`

- Adds methods to
  - traverse the `List` in either direction
  - modify the `List` during iteration

- A client program can maintain multiple iterators to facilitate searches and swaps inside of a list
The ListIterator Interface

- In addition to methods `next()`, `hasNext()`, and `remove()` that are inherited from interface `Iterator`, `ListIterator` provides the following additional methods:

  - `public int nextIndex();` //returns index of the element that would be called by `next()`
  - `public int previousIndex();` //returns index of the element that would be called by `previous()`
  - `public boolean hasPrevious();`
  - `public Object previous() throws NoSuchElementException;`
  - `public void add(Object o) throws 
    UnsupportedOperationException, ClassCastException, 
    IllegalArgumentException;` //inserts the object into the list immediately before the current index position
  - `public void set(Object o) throws 
    UnsupportedOperationexception, ClassCastException, 
    IllegalArgumentException;` //overwrites the last element returned by `next` or `previous` with the specified element
Using a ListIterator

List list = new LinkedList();
//fill the list with objects …
//Produces a ListIterator set to visit the element
//at index 5
ListIterator itr = list.getListIterator(5);
while (itr.hasNext()) {
    if (object.compareTo(itr.next()) < 0) {
        break;
    }
}
//A ListIterator allows items to be inserted in a List
itr.add(object);
The Abstract Class AbstractCollection

- Implements the following methods:
  - boolean `containsAll` (Collection c)
  - boolean `contains` (Object o)
  - boolean `removeAll` (Collection c)
  - boolean `remove` (Object o)
  - boolean `retainAll` (Collection c)

- The implementation of the methods `add` and `iterator` are under the responsibility of the subclasses:
  - boolean `add` (Object obj);
  - Iterator `iterator`();
public boolean retainAll(Collection c) {
    boolean modified = false;
    Iterator e = iterator();
    while (e.hasNext()) {
        if (!c.contains(e.next())) {
            e.remove();
            modified = true;
        }
    }
    return modified;
}
The Abstract Class AbstractSet

- The AbstractSet class is a convenience class that extends AbstractCollection and implements Set.
- Provides concrete implementations for the equals and hashCode methods.
  - The hash code of a set is the sum of the hash code of all the elements in the set.
- Since the size and iterator methods are not implemented, AbstractSet is an abstract class.
  - The implementation of these methods is under the responsibility of the subclasses.
The SortedSet Interface

**SortedSet**

- `comparator()` : Comparator
- `first()` : Object
- `headSet(toElement : Object) : SortedSet`
- `last()` : Object
- `subSet(fromElement : Object, toElement : Object) : SortedSet`
- `tailSet(fromElement : Object) : SortedSet`

**Comparable**

- `compareTo(element : Object) : int`
The Abstract Class AbstractList

- The AbstractList class is a convenience class that extends AbstractCollection and implements List

- Provides concrete implementations for the `indexOf` and `lastIndexOf` methods

- Since the `add` and `listIterator` methods are not implemented, AbstractList is an abstract class
  - The implementation of these methods is under the responsibility of the subclasses
## Implementation Classes

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implementation</th>
<th>Historical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash table</td>
<td>Resizable array</td>
<td>Tree (sorted)</td>
</tr>
<tr>
<td>Set</td>
<td>HashSet</td>
<td>TreeSet</td>
</tr>
<tr>
<td>List</td>
<td>ArrayList</td>
<td>LinkedList</td>
</tr>
<tr>
<td>Map</td>
<td>HashMap</td>
<td>TreeMap</td>
</tr>
</tbody>
</table>

- When writing programs think about interfaces and not implementations
  - This way the program does not become dependent on any added methods in a given implementation, leaving the programmer with the freedom to change implementations.

- An implementation class may elect not to support a particular method of the interface
  - `UnsupportedOperationException` is a runtime (unchecked) exception
Set Implementations

- **HashSet**
  - A `Set` backed by a hash table

- **TreeSet**
  - A semi-balanced binary tree implementation
  - Imposes an ordering on its elements
List Implementations

- **ArrayList**
  - A resizable-array implementation like **Vector**
    - Unsynchronized, and **without** "legacy" methods
  - **LinkedList**
    - A doubly-linked list implementation
    - May provide better performance than **ArrayList**
    - If elements frequently inserted/deleted within the **List**
    - For queues and double-ended queues (deques)
  - **Vector**
    - A synchronized resizable-array implementation of a **List** with additional "legacy" methods
The LinkedList Class

- The **LinkedList** class offers a few additional methods for directly manipulating the ends of the list:

  ```java
  void addFirst(Object)
  void addLast(Object);
  Object getFirst();
  Object getLast();
  Object removeFirst();
  Object removeLast();
  ```

- These methods make it natural to implement other simpler data structures, like Stacks and Queues
Using LinkedLists

- A few things to be aware of:
  - It is really bad to use the positional indexing features copiously of LinkedList if you care at all about performance
    - This is because the LinkedList has no memory and must always traverse the chain from the beginning
  - Elements can be changed both with the List and ListIterator objects
    - That latter is often more convenient
  - You can create havoc by creating several iterators that you use to mutate the List
    - There is some protection built-in, but best is to have only one iterator that will actually mutate the list structure
The ArrayList Class

- Additional methods for managing size of underlying array
  - **void** `ensureCapacity`(int `minCapacity`): Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument
  - **void** `trimToSize`(): Trims the capacity of this `ArrayList` instance to be the list's current size
- `size()`, `isEmpty()`, `get()`, `set()`, `iterator()` and `listIterator()` all run in constant time
- Adding `n` elements take `O[n]` time
- Can explicitly grow capacity in anticipation of adding many elements
- Note: legacy Vector class almost identical
  - Main differences are naming and synchronization
Map Implementations

- **HashMap**
  - A hash table implementation of `Map`
  - Like `Hashtable`, but supports null keys & values

- **TreeMap**
  - A semi-balanced binary tree implementation
  - Imposes an ordering on its elements

- **Hashtable**
  - Synchronized hash table implementation of `Map` interface, with additional "legacy" methods

### Load Factor & Initial Size
- Number of entries in the hashtable exceeds the product of the load factor and the current capacity
- Will rehash if load factor exceeds specified (time consuming)
Load Factor

- The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.
  - When the number of entries in the hashtable exceeds the product of the load factor and the current capacity, the capacity is increased by calling the rehash method.
- Generally, the default load factor (.75) offers a good tradeoff between time and space costs.
  - Higher values decrease the space overhead but increase the time cost to look up an entry (which is reflected in most Hashtable operations, including get and put).
- The initial capacity controls a tradeoff between wasted space and the need for rehash operations, which are time-consuming.
  - No rehash operations will ever occur if the initial capacity is greater than the maximum number of entries the Hashtable will contain divided by its load factor. However, setting the initial capacity too high can waste space.
- If many entries are to be made into a Hashtable, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table.
The Hashtable Class

- Like a **HashMap** but older version
- Constructors:
  - `Hashtable()` //Constructs a new, empty hashtable with a default initial capacity (11) and load factor, which is 0.75.
  - `Hashtable(int initialCapacity)` //Constructs a new, empty hashtable with the specified initial capacity and default load factor, which is 0.75
  - `Hashtable(int initialCapacity, float loadFactor)` //Constructs a new, empty hashtable with the specified initial capacity and the specified load factor
  - `Hashtable(Map t)` //Constructs a new hashtable with the same mappings as the given Map
Other J2SE Implementations

- **Legacy (since 1.0)**
  - `java.util.Vector`
  - `java.util.Stack`
  - `java.util.Hashtable`
  - `java.util.Properties`

- **J2SE 1.2**
  - `java.util.WeakHashMap`

- **J2SE 1.4**
  - `java.util.LinkedHashSet`
  - `java.util.LinkedHashMap`
  - `java.util.IdentityHashMap`

- **J2SE 1.5**
  - `java.util.EnumSet`
  - `java.util.EnumMap`
  - `java.util.PriorityQueue`
  - `java.util.concurrent.*`
public class TestCollection {
    public static void main(String args[]) {
        // Create an empty set
        Collection set = new HashSet();
        // Populate the set
        set.add(new Integer(47));
        set.add(new Double(3.14));
        set.add(new Character('h'));
        // iterate through the set of keys
        Iterator iter = set.iterator();
        while (iter.hasNext()) {
            //Assume items are printed in same order
            //they were put in
            System.out.println(iter.next());
        } // end while
    } // end main
} // end TestCollection
Example: Instantiating and Using Maps

// instantiate a concrete map
Map favoriteColors = new HashMap();
// Adding an association (key-value pair)
favoriteColors.put("Juliet", Color.pink);
// Changing an existing association
favoriteColors.put("Juliet", Color.red);
// Getting the value associated with a key
Color julietsFavoriteColor = (Color) favoriteColors.get("Juliet");
// Removing a key and its associated value
favoriteColors.remove("Juliet");
// Printing key/value Pairs
Set keySet = favoriteColors.keySet(); // get the set of keys
// iterate through the set of keys
Iterator iter = keySet.iterator();
while (iter.hasNext()) {
    Object key = iter.next();
    Object value = favoriteColors.get(key);
    System.out.println(key + " -> " + value);
}
Hash based Implementation

- **Hash based** implementation stores set elements (map keys) using a hash function

- A **suitable hash function** is defined in class `Object` (method `hashCode`) and inherited by all subclasses

- If a class overrides the `equals` method defined in class `Object` it is also necessary to override the `hashCode` method to make `HashSet` and `HashMap` work correctly
Tree based Implementation

- **TreeSet**
  - iteration gives elements in sorted order
  - To use a **TreeSet**, 
    - either your objects must implement interface **Comparable**
    - or you must provide a **Comparator** object

- **TreeMap**
  - the keys are kept in a **TreeSet**
  - To use a **TreeMap**
    - either your keys must implement interface **Comparable**
    - or you must provide a **Comparator** object for the keys
    - there is no requirement for the values
What About User Objects?

- The Collections framework will work with any Java class
- You need to be sure you have defined
  - `equals()`
  - `hashCode()`
  - `compareTo()` or `compare()`
- Don’t use mutable objects for keys in a `Map`
- The Map `hashCode()` returns distinct integers for distinct objects
  - If two objects are equal according to the `equals()` method, then the `hashCode()` method on each of the two objects must produce the same integer result
  - When `hashCode()` is invoked on the same object more than once, it must return the same integer, provided no information used in equals comparisons has been modified
  - It is *not* required that if two objects are unequal according to `equals()` that `hashCode()` must return distinct integer values
Sorting and Comparing

- **Comparable interface**
  - Must be implemented by all elements in `SortedSet`
  - Must be implemented by all keys in `SortedMap`
  - Method: `int compareTo(Object o)`
  - Defines "natural order" for that object class

- **Comparator interface**
  - Defines a function that compares two objects
  - Can design custom ordering scheme
  - Method: `int compare(Object o1, Object o2)`

- **Total vs. Partial Ordering**
  - Technical, changes behavior per object class
Return Abstract Collections

- OO is based on the notion of encapsulation
  - hiding implementation details behind interfaces

- Therefore: Return the most abstract interface that can possibly work
  - Return a List instead of an ArrayList
  - Return a Collection instead of a List
  - Return an Iterator instead of a Collection
The Collections Utility Class

- This class consists exclusively of static methods that operate on or return Collections and Maps
  - Polymorphic algorithms that operate on collections
  - Wrappers which return a new collection backed by a specified collection
  - Plus a few other odds and ends for creating synchronized collection classes, and for creating read-only collection classes
- Most of the algorithms operate on List objects, but a couple of them (max and min) operate on arbitrary Collection objects

**Collections**

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarySearch</td>
<td>list: List, key: Object : int</td>
</tr>
<tr>
<td>binarySearch</td>
<td>list: List, key: Object, c: Comparator : int</td>
</tr>
<tr>
<td>copy</td>
<td>src: List, des: List : void</td>
</tr>
<tr>
<td>enumeration</td>
<td>c: final Collection : Enumeration</td>
</tr>
<tr>
<td>fill</td>
<td>list: List, o: Object : void</td>
</tr>
<tr>
<td>max</td>
<td>c: Collection : Object</td>
</tr>
<tr>
<td>max</td>
<td>c: Collection, c: Comparator : Object</td>
</tr>
<tr>
<td>min</td>
<td>c: Collection : Object</td>
</tr>
<tr>
<td>min</td>
<td>c: Collection, c: Comparator : Object</td>
</tr>
<tr>
<td>nCopies</td>
<td>n: int, o: Object : List</td>
</tr>
<tr>
<td>reverse</td>
<td>list: List : void</td>
</tr>
<tr>
<td>reverseOrder</td>
<td>: Comparator</td>
</tr>
<tr>
<td>shuffle</td>
<td>list: List : void</td>
</tr>
<tr>
<td>shuffle</td>
<td>list: List, rnd: Random : void</td>
</tr>
<tr>
<td>singleton</td>
<td>o: Object : Set</td>
</tr>
<tr>
<td>singletonList</td>
<td>o: Object : List</td>
</tr>
<tr>
<td>singletonMap</td>
<td>key: Object, value: Object : Map</td>
</tr>
<tr>
<td>sort</td>
<td>list: List : void</td>
</tr>
<tr>
<td>sort</td>
<td>list: List, c: Comparator : void</td>
</tr>
<tr>
<td>synchronizedCollection</td>
<td>c: Collection : Collection</td>
</tr>
<tr>
<td>synchronizedList</td>
<td>list: List : List</td>
</tr>
<tr>
<td>synchronizedMap</td>
<td>m: Map : Map</td>
</tr>
<tr>
<td>synchronizedSet</td>
<td>s: Set : Set</td>
</tr>
<tr>
<td>synchronizedSortedMap</td>
<td>s: SortedMap : SortedMap</td>
</tr>
<tr>
<td>synchronizedSortedSet</td>
<td>s: SortedSet : SortedSet</td>
</tr>
<tr>
<td>unmodifiedCollection</td>
<td>c: Collection : Collection</td>
</tr>
<tr>
<td>unmodifiedList</td>
<td>list: List : List</td>
</tr>
<tr>
<td>unmodifiedMap</td>
<td>m: Map : Map</td>
</tr>
<tr>
<td>unmodifiedSet</td>
<td>s: Set : Set</td>
</tr>
<tr>
<td>unmodifiedSortedMap</td>
<td>s: SortedMap : SortedMap</td>
</tr>
<tr>
<td>unmodifiedSortedSet</td>
<td>s: SortedSet : SortedSet</td>
</tr>
</tbody>
</table>
The sort operation uses a slightly optimized merge sort algorithm

- **Fast**: This algorithm is guaranteed to run in $n \log(n)$ time, and runs substantially faster on nearly sorted lists
- **Stable**: That is to say, it doesn't reorder equal elements

**SortedSet, SortedMap interfaces**

- Collections that keep their elements sorted
- Iterators are guaranteed to traverse in sorted order (only SortedSet)

**Ordered Collection Implementations**

- TreeSet, TreeMap

```java
public static void sort(List list, Comparator c){
    Object a[] = list.toArray();
    Arrays.sort(a, c);
    ListIterator i = list.listIterator();
    for (int j=0; j<a.length; j++) {
        i.next();
        i.set(a[j]);
    }
}
```
import java.util.*;

public class SortExample {
    public static void main( String args[] ) {
        List l = new ArrayList();

        for ( int i = 0; i < args.length; i++ )
            l.add( args[ i ] );

        Collections.sort( l );

        System.out.println( l );
    }
}

Other Algorithms

- Other algorithms provided by the `Collections` class include:
  - **Singleton**
    - `collections.singleton(e)` returns an immutable set containing only the element `e`
    - `c.removeAll(Collections.singleton(e));` will remove all occurrences of `e` from the Collection `c`
  - **Shuffling**
  - **Data manipulation**
    - `reverse()`
    - `fill()`
    - `copy()`
  - **Searching**
  - **Finding extreme values**
    - `max()`
    - `min()`
The Arrays Utility Class

- It is too bad that arrays are not collections
  - You lose all of the power provided by the collection framework

- The class **Arrays** contains
  - Various static methods for manipulating arrays (such as sorting, searching, and comparing arrays, as well as filling array elements)
  - It also contains a method for converting arrays to lists

- Object-based Array sorting
  - **Arrays.sort(Object[])**
  - Equivalent methods for all primitive types (e.g., `int[]`)
import java.util.*;

public class SortExample {
    public static void main( String args[] ) {
        Arrays.sort( args );

        List l = Arrays.asList( args );

        System.out.println( l );
    }
}
Utility Classes Summary

- **Collections** class static methods:
  - `sort(List)`
  - `binarySearch(List, Object)`
  - `reverse(List)`
  - `shuffle(List)`
  - `fill(List, Object)`
  - `copy(List dest, List src)`
  - `min(Collection)`
  - `max(Collection)`
  - `synchronizedX, unmodifiableX` factory methods

- **Arrays** class static methods that act on Java arrays:
  - `sort`
  - `binarySearch`
  - `equals`
  - `fill`
  - `asList` - returns an `ArrayList` composed of this array's contents
Example: Counting UniqueWords

```java
import java.io.*;
import java.util.*;
public class UniqueWords {
    public static void main( String args[] ) {
        // Usage check & open file
        if ( args.length != 1 ) {
            System.err.println( "Usage: java UniqueWords word-file" );
            System.exit( 1 );
        }
        StreamTokenizer in = null;
        try {
            in = new StreamTokenizer( new BufferedReader ( new FileReader ( args[ 0 ] ) ) );
            in.ordinaryChar( '.' );
        } catch ( FileNotFoundException e ) { System.err.println( "UniqueWords: " + e.getMessage() );
            System.exit( 1 );
        }
        System.exit( 1 );
    }
    StreamTokenizer in = null;
    try {
        in = new StreamTokenizer( new BufferedReader ( new FileReader ( args[ 0 ] ) ) );
        in.ordinaryChar( '.' );
    } catch ( FileNotFoundException e ) { System.err.println( "UniqueWords: " + e.getMessage() );
        System.exit( 1 );
    }
```
try {
    Set set = new HashSet();
    while (( in.nextToken() != in.TT_EOF ) ) {
        if ( in.ttype == in.TT_WORD )
            set.add( in.sval );
    }
    System.out.println("There are " + set.size() + " unique words");
    System.out.println( set );
}
catch ( IOException e ) {
    System.err.println("UniqueWords: " + e.getMessage());
    System.exit( 1 );
}
You Want Them Sorted?

```java
try {
    SortedSet set = new TreeSet();
    while ( ( in.nextToken() != in.TT_EOF ) ) {
        if ( in.ttype == in.TT_WORD )
            set.add( in.sval );
    }
    System.out.println("There are " + set.size() + " unique words");
    System.out.println( set );
}
catch ( IOException e ) {
    System.err.println("UniqueWords: " + e.getMessage() );
    System.exit( 1 );
}
```
try {
    Set set = new HashSet();
    while ( ( in.nextToken() != in.TT_EOF ) ) {
        if ( in.ttype == in.TT_WORD )
            set.add( in.sval );
    }
    System.out.println("There are " + set.size() + " unique words");
    System.out.println(new TreeSet(set));
}
catch ( IOException e ) {
    System.err.println("UniqueWords: " + e.getMessage() );
    System.exit( 1 );
}
try {
    SortedSet set = new TreeSet();
    while ( ( in.nextToken() != in.TT_EOF ) ) {
        if ( in.ttype == in.TT_WORD )
            set.add( in.sval );
    }
    System.out.println("There are " + set.size() + " unique words");
    Iterator elements = set.iterator();
    System.out.println();
    while ( elements.hasNext() )
    {
        System.out.println( elements.next() );
    }
} catch ( IOException e ) {
    System.err.println( "UniqueWords: " + e.getMessage() );
    System.exit( 1 );
} }
Example: Counting Unique Words

```java
try {
    Map map = new HashMap();
    Integer one = new Integer(1);

    while ((in.nextToken() != in.TT_EOF)) {
        if (in.ttype == in.TT_WORD) {
            Integer freq = (Integer) map.get(in.sval);
            if (freq == null)
                freq = one;
            else
                freq = new Integer(freq.intValue() + 1);
            map.put(in.sval, freq);
        }
    }
}
Example: Counting UniqueWords

System.out.println("There are " + map.size() + ", unique words");

SortedMap sorted = new TreeMap(map.entrySet());
Iterator elements = sorted.iterator();

while ( elements.hasNext() ) {
    Map.Entry cur = ( Map.Entry )elements.next();
    System.out.println(cur.getValue() + ", " + cur.getKey());
}

} catch ( IOException e ) {
    System.err.println( "UniqueWords: " + e.getMessage() );
    System.exit( 1 );
}
Dealing with Changes of Collection Objects

- Modifiable / Unmodifiable
  - **Modifiable**: Collections that support modification, “transformer” operations, e.g., add(), remove(), clear()
  - **Unmodifiable**: Collections that do not support any modification operations

- Mutable / Immutable
  - **Immutable**: Collections that guarantee that no change in the Collection will ever be observable via “selector” operations, e.g., such as iterator(), size(), contains()
  - **Mutable**: Collections that are not immutable

- Fixed-size / Variable-size
  - **Fixed-size**: Lists that guarantee that their size will remain constant even though the elements may change
  - **Variable-size**: Lists that are not fixed-size are referred to as
Wrapper Implementations

- Basic Containers are sometimes inadequate
  - You want to return an immutable collection
  - You want a threadsafe collection

- Wrapper implementations add some functionality on top of what a collection offer
  - Unmodifiable
  - Synchronization

- Wrappers simply delegate all of their real work to a specified collection

```java
Collection.unmodifiableSortedSet(SortedSet s)
```
The unmodifiable wrappers, rather than adding functionality to the wrapped collection, take functionality away

- Any attempt to modify the collection generates an `UnsupportedOperationException`

The unmodifiable wrappers have two main uses:

- To make a collection immutable once it has been built
- To allow "second-class citizens" read-only access to your data structures. You keep a reference to the backing collection, but hand out a reference to the wrapper

```java
public static Collection unmodifiableCollection(Collection c);
public static Set unmodifiableSet(Set s);
public static List unmodifiableList(List list);
public static Map unmodifiableMap(Map m);
public static SortedSet unmodifiableSortedSet(SortedSet s);
public static SortedMap unmodifiableSortedMap(SortedMap m);
```
Thread Safety

- Collections, by default, are NOT thread-safe
  - Design decision for performance and "conceptual weight"

- Solutions:
  - Encapsulated Collections: In general, if the only access to a collection is through a thread-safe object, then that collection is safe
  - Synchronized Collections: Wrapper implementations that synchronize all relevant methods
    - Factory methods inside the Collections class:
      ```java
      List list = Collections.synchronizedList(new ArrayList(...));
      ```
  - Unmodifiable Collections: If an object can't be modified, it is thread-safe by definition
    - Factory methods inside the Collections class
      ```java
      List list = Collections.unmodifiableList(new ArrayList(...));
      ```
  - Fail-fast iterators
“Fail-fast” Iterators

● What happens if the underlying collection changes during iteration?
   ▶ Iterators store enough information to detect concurrent modification

● Iterators throw exceptions if the underlying collection has changed
   ▶ If collection is modified during the life of an iterator, then that iterator fails immediately rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future
   ▶ Exception: the iterator's own add() and remove() methods work fine

```java
final void checkForComodification() {
    if (modCount != expectedModCount)
        throw new ConcurrentModificationException();
}
```
Avoiding Co-modification Exceptions

```java
public void addListSelectionListener
    (ListSelectionListener foo) {
    _listenerList = new ArrayList(_listenerList);
    _listenerList.add(foo);
}
```
How To Select a Container

- Determine how you access elements
  - it doesn't matter
  - access by key
  - access by integer index

- Determine whether iteration order matters
  - it doesn't matter
  - elements must be sorted
  - elements must stay in order as inserted

- Determine which operations need to be fast
  - it doesn't matter
  - adding and removing elements must be fast
  - Finding elements must be fast
For Sets and Maps: Tree- or Hash-based?

- If `hashCode` is consistent with `equals` (all API classes) and iteration order doesn't matter
  - Use hash-based implementation
  - Otherwise use tree based implementation

- If both `Hash` and `Tree` based implementations are applicable, then the `Hash` based is usually the faster

- For tree based implementation:
  - Is `Comparable` implemented or must `Comparator` be specified? (on objects for `Sets`, on keys for `Maps`)

Basic Analysis of Algorithms

Figure 1 Big-o, big-Ω and big-Θ

\[ g(n) \text{ is } O(f(n)) \]

\[ g(n) \text{ is } \Omega(h(n)) \]

Figure 3 Orders of algorithms

Table 2 Complexity classes

<table>
<thead>
<tr>
<th>class</th>
<th>order</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>(O(1))</td>
</tr>
<tr>
<td>logarithmic</td>
<td>(O(\lg n))</td>
</tr>
<tr>
<td>linear</td>
<td>(O(n))</td>
</tr>
<tr>
<td>n log n</td>
<td>(O(n \lg n))</td>
</tr>
<tr>
<td>quadratic</td>
<td>(O(n^2))</td>
</tr>
<tr>
<td>cubic</td>
<td>(O(n^3))</td>
</tr>
<tr>
<td></td>
<td>(O(n^4))</td>
</tr>
<tr>
<td></td>
<td>(O(2^n))</td>
</tr>
</tbody>
</table>

Table 2 Complexity classes
# Summary of Set implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Member array representation</th>
<th>SLL representation</th>
<th>Boolean array representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>add</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>equals</td>
<td>$O(n_2)$</td>
<td>$O(n_2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>containsAll</td>
<td>$O(n_2)$</td>
<td>$O(n_2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>addAll</td>
<td>$O(n_1+n_2)$</td>
<td>$O(n_1+n_2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>removeAll</td>
<td>$O(n_1+n_2)$</td>
<td>$O(n_1+n_2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>retainAll</td>
<td>$O(n_1+n_2)$</td>
<td>$O(n_1+n_2)$</td>
<td>$O(m)$</td>
</tr>
</tbody>
</table>
## Summary of Set Implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Algorithm</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains</td>
<td>CBHT search</td>
<td>$O(1)$ best, $O(n)$ worst</td>
</tr>
<tr>
<td>add</td>
<td>CBHT insertion</td>
<td>$O(1)$ best, $O(n)$ worst</td>
</tr>
<tr>
<td>remove</td>
<td>CBHT deletion</td>
<td>$O(1)$ best, $O(n)$ worst</td>
</tr>
</tbody>
</table>

- **Closed-bucket hash table (CBHT):**
  - Each bucket may be occupied by several entries
  - Buckets are completely separate
  - Simplest implementation: each bucket is an SLL
## Summary of Set implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Algorithm</th>
<th>Time complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains</td>
<td>BST search</td>
<td>$O(\log n)$ best $O(n)$ worst</td>
</tr>
<tr>
<td>add</td>
<td>BST insertion</td>
<td>$O(\log n)$ best $O(n)$ worst</td>
</tr>
<tr>
<td>remove</td>
<td>BST deletion</td>
<td>$O(\log n)$ best $O(n)$ worst</td>
</tr>
</tbody>
</table>
# Summary of Map implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Key-indexed array representation</th>
<th>Array representation</th>
<th>SLL representation</th>
<th>BST representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>best</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>worst</td>
</tr>
<tr>
<td>remove</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>best</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>worst</td>
</tr>
<tr>
<td>put</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>best</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>worst</td>
</tr>
<tr>
<td>putAll</td>
<td>$O(m)$</td>
<td>$O(n_1+n_2)$</td>
<td>$O(n_1+n_2)$</td>
<td>$O(n_2 \log (n_1+n_2))$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>best</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>worst</td>
</tr>
<tr>
<td>equals</td>
<td>$O(m)$</td>
<td>$O(n_2)$</td>
<td>$O(n_2)$</td>
<td>$O(n_1 \log n_2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>best</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>worst</td>
</tr>
</tbody>
</table>
# Summary of Map implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Algorithm</th>
<th>Time complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>CBHT search</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove</td>
<td>CBHT deletion</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>put</td>
<td>CBHT insertion</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>putAll</td>
<td>merge on corresponding buckets of both CBHTs</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>equals</td>
<td>equality test on corresponding buckets of both CBHTs</td>
<td>$O(m)$</td>
</tr>
</tbody>
</table>
## Summary of List Implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array representation</th>
<th>SLL representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>set</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>add(int, Object)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>add(Object)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>equals</td>
<td>$O(n_2)$</td>
<td>$O(n_2)$</td>
</tr>
<tr>
<td>addAll</td>
<td>$O(n_2)$</td>
<td>$O(n_2)$</td>
</tr>
</tbody>
</table>