Implementation comparisons

- Options for implementing the list ADT
  - Array
    - Has a fixed size
    - Data must be shifted during insertions and deletions
  - Linked list
    - Is able to grow in size as needed
    - Does not require the shifting of items during insertions and deletions

Resizable Arrays

- The number of references in a Java array is of fixed size
- Resizable array
  - An array that grows and shrinks (don't normally bother) as the program executes
  - An illusion that is created by using an allocate and copy strategy with fixed-size arrays
- `java.util.ArrayList` class
  - Uses a similar technique to implement a growable array of objects

Comparing Array-Based and Referenced-Based Implementations

- Size
  - Array-based
    - Fixed size
      - Issues
        - » Can you predict the maximum number of items in the ADT?
        - » Will an array waste storage?
      - Resizable array
        - » Increasing the size of a resizable array can waste storage and time

Comparing Array-Based and Referenced-Based Implementations

- Size (Continued)
  - Reference-based
    - Do not have a fixed size
      - Do not need to predict the maximum size of the list
      - Will not waste storage
- Storage requirements
  - Array-based
    - Can require less memory than a reference-based implementation
      - There is no need to store information about where to find the next data item

Comparing Array-Based and Referenced-Based Implementations

- Storage requirements (Continued)
  - Reference-based
    - Can requires more storage
      - An item explicitly references the next item in the list
- Access time
  - Array-based
    - Constant access time
  - Reference-based
    - The time to access the i^{th} node depends on i

Comparing Array-Based and Referenced-Based Implementations

- Insertion and deletions
  - Array-based
    - Require you to shift the data
  - Reference-based
    - Do not require you to shift the data
    - Require a list traversal

Big-O comparisons

	Array	Resizable array	Linked List
get(index) add(index, value) remove(index) add on front add on end	O(1) O(n) O(n) O(n) O(1)	O(1) O(n) O(n) O(n) O(1)	O(n) O(n) O(n) O(1) O(n) (or O(1) with a tail pointer)

The Java Collections Framework

- Implements many of the more commonly used ADTs
- Collections framework
  - Unified architecture for representing and manipulating collections
  - Includes
    - Interfaces
    - Implementations
    - Algorithms

The Java Collection's Framework `List` Interface

- JCF provides an interface `java.util.List`
- List interface supports an ordered collection
  - Also known as a sequence
- Methods
  - `boolean add(E o)`
  - `void add(int index, E element)`
  - `void clear()`
  - `boolean contains(Object o)`
  - `boolean equals(Object o)`
  - `E get(int index)`
  - `int indexOf(Object o)`

etc

The `List` Interface example

```
private static void putNumbers(List<Integer> list, int
   size) {
  for (int i = 0; i < size; i++) {
    int random = (int) (Math.random() * 100);
    list.add(random);
  }
}
```

"list" could be any class that implements List.

```
putNumbers(new ArrayList<Integer>(), 1000);
putNumbers(new LinkedList<Integer>(), 1000);
```

See `TimeJCFLists.java` for a comparison of adding elements.

The `List` Interface example (cont.)

```
private static void listMethods (List<Integer> list) {
  System.out.println("\n" + list.getClass().getName() + "\n");
  System.out.println(list);
  list.add(2, 100);
  list.add(101);
  System.out.println("element 2: " + list.get(2));
  list.remove(2);
  System.out.println("element 2: " + list.get(2));
  list.remove(new Integer(101));
  System.out.println("element 2: " + list.get(2));
  list.set(2, 101);
  System.out.println("element 2: " + list.get(2));
  System.out.println("contains 101: " + list.contains(101));
  System.out.println("101 is element: " + list.indexOf(101));
  list.clear();
  System.out.println("contains 101: " + list.contains(101));
  System.out.println("101 is element: " + list.indexOf(101));
}
```


Iterators

- Iterator
  - Gives the ability to cycle through items in a collection
  - Access next item in a collection by using iter.next()
- JCF provides two primary iterator interfaces
  - java.util.Iterator
  - java.util.ListIterator (can move backwards and modify the list)
- Every ADT collection in the JCF has a method to return an iterator object

Iterators

- `ListIterator` methods (E is the type of elements)
  - `void add(E o)`
  - `boolean hasNext()`
  - `boolean hasPrevious()`
  - `E next()`
  - `int nextIndex()`
  - `E previous()`
  - `int previousIndex()`
  - `void remove()`
  - `void set(E o)`

Traversing Lists example

```
private static void traversalExample(List<Integer>
                                                list) {
 System.out.println(list);
 for (ListIterator<Integer> iterator =
         list.listIterator(); iterator.hasNext();)
   iterator.set(iterator.next() * 2);

 System.out.println(list);
 List<Integer> result = new LinkedList<Integer>();
 for (Integer i : list)
   result.add(i * 2);
 System.out.println(result);
}
```


Summary

- Reference variables can be used to implement the data structure known as a linked list
- Each reference in a linked list is a reference to the next node in the list
- Algorithms for insertions and deletions in a linked list involve
  - Traversing the list from the beginning until you reach the appropriate position
  - Performing reference changes to alter the structure of the list

Summary

- Inserting a new node at the beginning of a linked list and deleting the first node of a linked list are special cases
- An array-based implementation uses an implicit ordering scheme; a reference-based implementation uses an explicit ordering scheme
- Any element in an array can be accessed directly; you must traverse a linked list to access a particular node
- Items can be inserted into and deleted from a reference-based linked list without shifting data

Summary

- The `new` operator can be used to allocate memory dynamically for both an array and a linked list
  - The size of a linked list can be increased one node at a time more efficiently than that of an array
- A binary search of a linked list is impractical
- Recursion can be used to perform operations on a linked list

Summary

- A tail reference can be used to facilitate locating the end of a list
- In a circular linked list, the last node references the first node
- Dummy head nodes eliminate the special cases for insertion into and deletion from the beginning of a linked list
- A head record contains global information about a linked list
- A doubly linked list allows you to traverse the list in either direction

Summary

- Java Collections Framework
  - Contains interfaces, implementations, and algorithms for many common ADTs
- Collection
  - Object that holds other objects
  - Iterator cycles through its contents