Week 2

{ Data Structures: Stacks, Queues, LinkedLists

Big-Oh Notation

Definitions

- Big-Oh notation provides an upper bound on a function to within a constant factor.
- Let f and g be functions from nonnegative numbers to nonnegative numbers.
- $f(n)$ is $O(g(n))$ if there are positive constants C and k such that:

  $f(n) \leq C * g(n)$ whenever $n > k$
- $f(n)$ is $O(g(n))$ is equivalent to: $\exists C \exists k \forall n (n > k \rightarrow f(n) \leq C * g(n))$
- To prove big-Oh, find witnesses, specific values for C and k, and prove

  $n > k$ implies $f(n) \leq C * g(n)$.

One Approach for Finding Witnesses

- Generate a table for $f(n)$ and $g(n)$, using $n = 1$, $n = 10$ and $n = 100$.
  [Use values smaller than 10 and 100 if you wish.]
- Guess $C = \lceil f(1)/g(1) \rceil$
  (or $C = \lceil f(10)/g(10) \rceil$).
- Check that $f(10) \leq C * g(10)$ and $f(100) \leq C * g(100)$.
  [If this is not true, $f(n)$ might not be $O(g(n))$.]
- Choose $k = 1$ (or $k = 10$).
- Prove that $\forall n (n > k \rightarrow f(n) \leq C * g(n))$.
  [It's ok if you end up with a larger, but still constant, value for C.]

Tricks for Proving Big-Oh

- Assume $n > 1$ if you chose $k = 1$ (or $n > 10$ if you chose $k = 10$).
- To prove $f(n) \leq C * g(n)$, you need to find expressions larger than $f(n)$ and smaller than $C * g(n)$.
- If the lowest-order term is negative, just eliminate it to obtain a larger expression.
- Repeatedly use $n > k$ and $2n > 2k$ and $3n > 3k$ and so on to "convert" the lowest-order term into a higher-order term.
- Check that your expressions are less than $C * g(n)$ by using $n = 100$.

Proving Big-Oh: Example 1

Show that $3n + 7$ is $O(n)$. In this case, $f(n) = 3n + 7$ and $g(n) = n$.

n	$f(n)$	$g(n)$	$\lceil f(n)/g(n) \rceil$
1	10	1	10
10	37	10	4
100	307	100	4

This table suggests trying $k = 1$ and $C = 10$ or $k = 10$ and $C = 4$.

Proving either one is good enough to prove big-Oh.

Example 1, Slide 2

Try $k = 1$ and $C = 10$.
Want to prove $n > 1$ implies $3n + 7 \leq 10n$.

Assume $n > 1$. Want to show $3n + 7 \leq 10n$.

7 is the lowest order term, so work on that first.
$n > 1$ implies $7n > 7$, which implies
$3n + 7 < 3n + 7n = 10n$.

This finishes the proof. A short version is:
$n > 1$ implies $3n + 7 < 3n + 7n = 10n$.

Example 1, Slide 3

Try $k = 10$ and $C = 4$.
Want to prove $n > 10$ implies $3n + 7 \leq 4n$.

Assume $n > 10$. Want to show $3n + 7 \leq 4n$.

Work on the lowest-order term first.
$n > 10$ implies $n > 7$, which implies
$3n + 7 < 3n + n = 4n$.

This finishes the proof. A short version is:
$n > 10$ implies $3n + 7 < 3n + n = 4n$.

Review:

http://www.cs.utsa.edu/~bylander/cs2233/bigohhandout.pdf

Tricks for Proving Big-Omega

- Assume $n > 1$ if you chose $k = 1$ (or $n > 10$ if you chose $k = 10$).
- To prove $f(n) \geq C * g(n)$, you need to find expressions smaller than $f(n)$ and larger than $C * g(n)$.
- If the lowest order term is positive, just eliminate it to obtain a larger expression.
- Repeatedly use $-k > -n$ and $-0.1k > -0.1n$ and so on to "convert" the lowest order term into a higher order term.
- Check that your expressions are greater than $C * g(n)$ by using $n = 100$.

CS 2233 Discrete Mathematical Structures — Order Notation and Time Complexity – 21

Finding Big-Omega Witnesses

- Generate a table for $f(n)$ and $g(n)$, using $n = 1$, $n = 10$ and $n = 100$.
  [Use values smaller than 10 and 100 if you wish.]
- Guess $1/C = [g(1)/f(1)]$
  (or more likely $1/C = [g(10)/f(10)]$).
- Check that $f(10) \geq C * g(10)$ and $f(100) \geq C * g(100)$.
  [If this is not true, $f(n)$ might not be $\Omega(g(n))$.]
- Choose $k = 1$ (or $k = 10$).
- Prove that $\forall n(n > k \rightarrow f(n) \geq C * g(n))$.
  [It's ok if you end up with a smaller, but still positive, value for C.]

CS 2233 Discrete Mathematical Structures — Order Notation and Time Complexity – 22

Proving Big-Omega: Example 1

Show that $3n + 7$ is $\Omega(n)$. In this case, $f(n) = 3n + 7$ and $g(n) = n$.

n	$f(n)$	$g(n)$	$[g(n)/h(n)]$	C?
1	10	1	1	1
10	37	10	1	1
100	307	100	1	1

This table suggests trying $k = 1$ and $C = 1$.
Want to prove $n > 1$ implies $3n + 7 \geq n$.

Short proof: $n > 1$ implies $3n + 7 > 3n > n$.

CS 2233 Discrete Mathematical Structures — Order Notation and Time Complexity – 23

11

Proving Big-Omega: Example 2

Show that $n^2 - 2n + 1$ is $\Omega(n^2)$. In this case, $f(n) = n^2 - 2n + 1$ and $g(n) = n^2$.

n	$f(n)$	$g(n)$	$[g(n)/f(n)]$	C?
1	1	1	1	1
10	81	100	2	1/2
100	9801	10000	2	1/2

This table suggests trying $k = 10$ and $C = 1/2$. Trying $C = 1$ is too large.

CS 2233 Discrete Mathematical Structures — Order Notation and Time Complexity – 24

Example 2, Slide 2

Try $k = 10$ and $C = 1/2$.
Want to prove $n > 10$ implies $n^2 - 2n + 1 \geq n^2/2$.

Assume $n > 10$. Want to show $f(n) \geq n^2/2$.

The lowest-order term is positive, so eliminate.
$n^2 - 2n + 1 > n^2 - 2n$

$n > 10$ implies $-10 > -n$, implies $-2 > -0.2n$.
$-2 > -0.2n$ implies $n^2 - 2n > n^2 - 0.2n^2 = 0.8n^2$.

$n > 10$ implies $0.8n^2 > n^2/2$.

This finishes the proof. A short version is:
$n > 10$ implies $n^2 - 2n + 1 > n^2 - 2n > 0.8n^2 > n^2/2$

CS 2233 Discrete Mathematical Structures — Order Notation and Time Complexity – 25

Proving Big-Omega: Example 3

Show that $(n - 1)^3$ is $\Omega(n^3)$. In this case, $f(n) = (n - 1)^3$ and $g(n) = n^3$.

n	$f(n)$	$g(n)$	$[g(n)/f(n)]$	C?
1	0	1	undefined	undefined
10	729	1000	2	1/2
100	970299	1000000	2	1/2

This table suggests trying $k = 10$ and $C = 1/2$.
Trying $k = 1$ does not look like a good idea.

CS 2233 Discrete Mathematical Structures — Order Notation and Time Complexity – 26

12

Review:

http://www.cs.utsa.edu/~bylander/cs2233/bigohhandout.pdf

A ***linked list*** is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. Linked lists provide a simple, flexible representation for dynamic sets, supporting (though not necessarily efficiently) all the operations listed

Linked Lists Cite: CLRS - 10

SEARCH(S, k)
A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S.

INSERT(S, x)
A modifying operation that augments the set S with the element pointed to by x. We usually assume that any attributes in element x needed by the set implementation have already been initialized.

DELETE(S, x)
A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation takes a pointer to an element x, not a key value.)

MINIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.

MAXIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the largest key.

SUCCESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

PREDECESSOR(S, x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

Figure 10.3 (a) A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. (b) Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. (c) The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

Linked Lists

Searching a linked list

The procedure LIST-SEARCH(L, k) finds the first element with key k in list L by a simple linear search, returning a pointer to this element. If no object with key k appears in the list, then the procedure returns NIL. For the linked list in Figure 10.3(a), the call LIST-SEARCH$(L, 4)$ returns a pointer to the third element, and the call LIST-SEARCH$(L, 7)$ returns NIL.

```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

To search a list of n objects, the LIST-SEARCH procedure takes $\Theta(n)$ time in the worst case, since it may have to search the entire list.

Linked Lists

Inserting into a linked list

Given an element x whose *key* attribute has already been set, the LIST-INSERT procedure "splices" x onto the front of the linked list, as shown in Figure 10.3(b).

```
LIST-INSERT(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

(Recall that our attribute notation can cascade, so that $L.head.prev$ denotes the *prev* attribute of the object that $L.head$ points to.) The running time for LIST-INSERT on a list of n elements is $O(1)$.

Linked Lists

Deleting from a linked list

The procedure LIST-DELETE removes an element x from a linked list L. It must be given a pointer to x, and it then "splices" x out of the list by updating pointers. If we wish to delete an element with a given key, we must first call LIST-SEARCH to retrieve a pointer to the element.

```
LIST-DELETE(L, x)
if x.prev ≠ NIL
    x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
    x.next.prev = x.prev
```

Figure 10.3(c) shows how an element is deleted from a linked list. LIST-DELETE runs in $O(1)$ time, but if we wish to delete an element with a given key, $\Theta(n)$ time is required in the worst case because we must first call LIST-SEARCH to find the element.

Linked Lists

Figure 10.4 A circular, doubly linked list with a sentinel. The sentinel $L.nil$ appears between the head and tail. The attribute $L.head$ is no longer needed, since we can access the head of the list by $L.nil.next$. **(a)** An empty list. **(b)** The linked list from Figure 10.3(a), with key 9 at the head and key 1 at the tail. **(c)** The list after executing LIST-INSERT$'(L, x)$, where $x.key = 25$. The new object becomes the head of the list. **(d)** The list after deleting the object with key 1. The new tail is the object with key 4.

Linked Lists

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a ***stack***, the element deleted from the set is the one most recently inserted: the stack implements a ***last-in, first-out***, or ***LIFO***, policy. Similarly, in a ***queue***, the element deleted is always the one that has been in the set for the longest time: the queue implements a ***first-in, first-out***, or ***FIFO***, policy. There are several efficient ways to implement stacks and queues on a computer. In this section we show how to use a simple array to implement each.

Stack versus Queue

Stacks

```
STACK-EMPTY(S)
if S.top == 0
    return TRUE
else return FALSE

PUSH(S, x)
S.top = S.top + 1
S[S.top] = x

POP(S)
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top + 1]
```

Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH(S, 17) and PUSH(S, 3). **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

Figure 10.1 shows the effects of the modifying operations PUSH and POP. Each of the three stack operations takes $O(1)$ time.

Stacks

Linked List Implementation of Stacks

The first implementation of a stack uses a singly linked list. We perform a `push` by inserting at the front of the list. We perform a `pop` by deleting the element at the front of the list. A `top` operation merely examines the element at the front of the list, returning its value. Sometimes the `pop` and `top` operations are combined into one.

Array Implementation of Stacks

An alternative implementation avoids links and is probably the more popular solution. Mimicking the `ArrayList add` operation, the implementation is trivial. Associated with each stack is `theArray` and `topOfStack`, which is `-1` for an empty stack (this is how an empty stack is initialized). To push some element `x` onto the stack, we increment `topOfStack` and then set `theArray[topOfStack] = x.` To pop, we set the return value to `theArray[topOfStack]` and then decrement `topOfStack`.

Implementation of Stacks Weiss Chapter 3

We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE; like the stack operation POP, DEQUEUE takes no element argument. The FIFO property of a queue causes it to operate like a line of customers waiting to pay a cashier. The queue has a ***head*** and a ***tail***. When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the customer at the head of the line who has waited the longest.

Queues

```
ENQUEUE(Q, x)
Q[Q.tail] = x
if Q.tail == Q.length
    Q.tail = 1
else Q.tail = Q.tail + 1

DEQUEUE(Q)
x = Q[Q.head]
if Q.head == Q.length
    Q.head = 1
else Q.head = Q.head + 1
return x
```

Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE(Q, 17), ENQUEUE(Q, 3), and ENQUEUE(Q, 5). **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

Figure 10.2 shows the effects of the ENQUEUE and DEQUEUE operations. Each operation takes $O(1)$ time.

Queues

Reference Liang
Chapter 22
Lists, Stacks, Queues

Objectives

- To explore the relationship between interfaces and classes in the Java Collections Framework hierarchy.
- To use the common methods defined in the **Collection** interface for operating collections.
- To use the **Iterator** interface to traverse the elements in a collection.
- To use a for-each loop to traverse the elements in a collection.
- To explore how and when to use **ArrayList** or **LinkedList** to store elements.
- To compare elements using the **Comparable** interface and the **Comparator** interface.
- To use the static utility methods in the **Collections** class for sorting, searching, shuffling lists, and finding the largest and smallest element in collections.
- To develop a multiple bouncing balls application using **ArrayList**.
- To distinguish between **Vector** and **ArrayList** and to use the **Stack** class for creating stacks.
- To explore the relationships among **Collection**, **Queue**, **LinkedList**
- To use stacks to write a program to evaluate expressions.

Java Collection Framework hierarchy

A *collection* is a container object that holds a group of objects, often referred to as *elements*. The Java Collections Framework supports three types of collections, named *sets, lists,* and *maps*.

Java Collection Framework hierarchy, cont.

Set and List are subinterfaces of Collection.

The Collection Interface

The Collection interface is the root interface for manipulating a collection of objects.

Returns an iterator for the elements in this collection.

Adds a new element o to this collection.

Adds all the elements in the collection c to this collection.

Removes all the elements from this collection.

Returns true if this collection contains the element o.

Returns true if this collection contains all the elements in c.

Returns true if this collection is equal to another collection o.

Returns the hash code for this collection.

Returns true if this collection contains no elements.

Removes the element o from this collection.

Removes all the elements in c from this collection.

Retains the elements that are both in c and in this collection.

Returns the number of elements in this collection.

Returns an array of Object for the elements in this collection.

Returns true if this iterator has more elements to traverse.

Returns the next element from this iterator.

Removes the last element obtained using the next method.

The List Interface

A set stores non-duplicate elements. To allow duplicate elements to be stored in a collection, you need to use a list. A list can not only store duplicate elements, but can also allow the user to specify where the element is stored. The user can access the element by index.

The List Interface, cont.

Adds a new element at the specified index.

Adds all the elements in c to this list at the specified index.

Returns the element in this list at the specified index.

Returns the index of the first matching element.

Returns the index of the last matching element.

Returns the list iterator for the elements in this list.

Returns the iterator for the elements from startIndex.

Removes the element at the specified index.

Sets the element at the specified index.

Returns a sublist from fromIndex to toIndex.

The List Iterator

Adds the specified object to the list.

Returns true if this list iterator has more elements when traversing backward.

Returns the index of the next element.

Returns the previous element in this list iterator.

Returns the index of the previous element.

Replaces the last element returned by the previous or next method with the specified element.

ArrayList and LinkedList

The ArrayList class and the LinkedList class are concrete implementations of the List interface. Which of the two classes you use depends on your specific needs. If you need to support random access through an index without inserting or removing elements from any place other than the end, ArrayList offers the most efficient collection. If, however, your application requires the insertion or deletion of elements from any place in the list, you should choose LinkedList. A list can grow or shrink dynamically. An array is fixed once it is created. If your application does not require insertion or deletion of elements, the most efficient data structure is the array.

java.util.ArrayList

Creates an empty list with the default initial capacity.

Creates an array list from an existing collection.

Creates an empty list with the specified initial capacity.

Trims the capacity of this ArrayList instance to be the list's current size.

java.util.LinkedList

Creates a default empty linked list.

Creates a linked list from an existing collection.

Adds the object to the head of this list.

Adds the object to the tail of this list.

Returns the first element from this list.

Returns the last element from this list.

Returns and removes the first element from this list.

Returns and removes the last element from this list.

Example: Using ArrayList and LinkedList

This example creates an array list filled with numbers, and inserts new elements into the specified location in the list. The example also creates a linked list from the array list, inserts and removes the elements from the list. Finally, the example traverses the list forward and backward.

```java
import java.util.*;

public class TestIterator {
  public static void main(String[] args) {
    Collection<String> collection = new ArrayList<String>();
    collection.add("New York");
    collection.add("Atlanta");
    collection.add("Dallas");
    collection.add("Madison");

    Iterator<String> iterator = collection.iterator();
    while (iterator.hasNext()) {
      System.out.print(iterator.next().toUpperCase() + " ");
    }
    System.out.println();
  }
}
```

```java
import java.util.*;

public class TestArrayAndLinkedList {
  public static void main(String[] args) {
    List<Integer> arrayList = new ArrayList<Integer>();
    arrayList.add(1); // 1 is autoboxed to new Integer(1)
    arrayList.add(2);
    arrayList.add(3);
    arrayList.add(1);
    arrayList.add(4);
    arrayList.add(0, 10);
    arrayList.add(3, 30);

    System.out.println("A list of integers in the array list:");
    System.out.println(arrayList);

    LinkedList<Object> linkedList = new LinkedList<Object>(arrayList);
    linkedList.add(1, "red");
    linkedList.removeLast();
    linkedList.addFirst("green");

    System.out.println("Display the linked list backward:");
    for (int i = linkedList.size() - 1; i >= 0; i--) {
      System.out.print(linkedList.get(i) + " ");
    }
  }
}
```

The Comparator Interface

Sometimes you want to insert elements of different types into a tree set. The elements may not be instances of Comparable or are not comparable. You can define a comparator to compare these elements. To do so, create a class that implements the java.util.Comparator interface. The Comparator interface has two methods, compare and equals.

The Comparator Interface

public int compare(Object element1, Object element2)

Returns a negative value if element1 is less than element2, a positive value if element1 is greater than element2, and zero if they are equal.

public boolean equals(Object element)

Returns true if the specified object is also a comparator and imposes the same ordering as this comparator.

```java
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /** Construct a default geometric object */
  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }

  /** Construct a geometric object with color and filled value */
  protected GeometricObject(String color, boolean filled) {
    dateCreated = new java.util.Date();
    this.color = color;
    this.filled = filled;
  }

  /** Return color */
  public String getColor() {
    return color;
  }

  /** Set a new color */
  public void setColor(String color) {
    this.color = color;
  }

  /** Return filled. Since filled is boolean,
   *  the get method is named isFilled */
  public boolean isFilled() {
    return filled;
  }
```

```java
  /** Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }

  /** Get dateCreated */
  public java.util.Date getDateCreated() {
    return dateCreated;
  }

  @Override
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }

  /** Abstract method getArea */
  public abstract double getArea();

  /** Abstract method getPerimeter */
  public abstract double getPerimeter();
}
```

```java
public class Rectangle extends GeometricObject {
  private double width;
  private double height;

  public Rectangle() {
  }

  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }

  /** Return width */
  public double getWidth() {
    return width;
  }

  /** Set a new width */
  public void setWidth(double width) {
    this.width = width;
  }

  /** Return height */
  public double getHeight() {
    return height;
  }

  /** Set a new height */
  public void setHeight(double height) {
    this.height = height;
  }
```

```java
  @Override /** Return area */
  public double getArea() {
    return width * height;
  }

  @Override /** Return perimeter */
  public double getPerimeter() {
    return 2 * (width + height);
  }
}
```

```java
public class Circle extends GeometricObject {
  private double radius;

  public Circle() {
  }

  public Circle(double radius) {
    this.radius = radius;
  }

  /** Return radius */
  public double getRadius() {
    return radius;
  }

  /** Set a new radius */
  public void setRadius(double radius) {
    this.radius = radius;
  }

  @Override /** Return area */
  public double getArea() {
    return radius * radius * Math.PI;
  }

  /** Return diameter */
  public double getDiameter() {
    return 2 * radius;
  }

  @Override /** Return perimeter */
  public double getPerimeter() {
    return 2 * radius * Math.PI;
  }
```

```java
  /* Print the circle info */
  public void printCircle() {
    System.out.println("The circle is created " + getDateCreated() +
      " and the radius is " + radius);
  }
}
```

```java
public class TestGeometricObject {
  /** Main method */
  public static void main(String[] args) {
    // Declare and initialize two geometric objects
    GeometricObject geoObject1 = new Circle(5);
    GeometricObject geoObject2 = new Rectangle(5, 3);

    System.out.println("The two objects have the same area? " +
      equalArea(geoObject1, geoObject2));

    // Display circle
    displayGeometricObject(geoObject1);

    // Display rectangle
    displayGeometricObject(geoObject2);
  }

  /** A method for comparing the areas of two geometric objects */
  public static boolean equalArea(GeometricObject object1,
      GeometricObject object2) {
    return object1.getArea() == object2.getArea();
  }

  /** A method for displaying a geometric object */
  public static void displayGeometricObject(GeometricObject object) {
    System.out.println();
    System.out.println("The area is " + object.getArea());
    System.out.println("The perimeter is " + object.getPerimeter());
  }
}
```

```java
import java.util.Comparator;

public class GeometricObjectComparator
    implements Comparator<GeometricObject>, java.io.Serializable {
  public int compare(GeometricObject o1, GeometricObject o2) {
    double area1 = o1.getArea();
    double area2 = o2.getArea();

    if (area1 < area2)
      return -1;
    else if (area1 == area2)
      return 0;
    else
      return 1;
  }
}
```

```java
import chapter15.GeometricObject;
import chapter15.Rectangle;
import chapter15.Circle;
import java.util.Comparator;

public class TestComparator {
  public static void main(String[] args) {
    GeometricObject g1 = new Rectangle(5, 5);
    GeometricObject g2 = new Circle(5);

    GeometricObject g =
      max(g1, g2, new GeometricObjectComparator());

    System.out.println("The area of the larger object is " +
      g.getArea());
  }

  public static GeometricObject max(GeometricObject g1,
      GeometricObject g2, Comparator<GeometricObject> c) {
    if (c.compare(g1, g2) > 0)
      return g1;
    else
      return g2;
  }
}
```

The Collections Class

The Collections class contains various static methods for operating on collections and maps, for creating synchronized collection classes, and for creating read-only collection classes.

The Collections Class UML Diagram

	java.util.Collections	
List	+sort(list: List): void	Sorts the specified list.
	+sort(list: List, c: Comparator): void	Sorts the specified list with the comparator.
	+binarySearch(list: List, key: Object): int	Searches the key in the sorted list using binary search.
	+binarySearch(list: List, key: Object, c: Comparator): int	Searches the key in the sorted list using binary search with the comparator.
	+reverse(list: List): void	Reverses the specified list.
	+reverseOrder(): Comparator	Returns a comparator with the reverse ordering.
	+shuffle(list: List): void	Shuffles the specified list randomly.
	+shuffle(list: List): void	Shuffles the specified list with a random object.
	+copy(des: List, src: List): void	Copies from the source list to the destination list.
	+nCopies(n: int, o: Object): List	Returns a list consisting of *n* copies of the object.
	+fill(list: List, o: Object): void	Fills the list with the object.
Collection	+max(c: Collection): Object	Returns the max object in the collection.
	+max(c: Collection, c: Comparator): Object	Returns the max object using the comparator.
	+min(c: Collection): Object	Returns the min object in the collection.
	+min(c: Collection, c: Comparator): Object	Returns the min object using the comparator.
	+disjoint(c1: Collection, c2: Collection): boolean	Returns true if c1 and c2 have no elements in common.
	+frequency(c: Collection, o: Object): int	Returns the number of occurrences of the specified element in the collection.

Case Study: Multiple Bouncing Balls

```java
import javax.swing.Timer;
import java.util.ArrayList;
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class MultipleBallApp extends JApplet {
  public MultipleBallApp() {
    add(new BallControl());
  }

  class BallControl extends JPanel {
    private BallPanel ballPanel = new BallPanel();
    private JButton jbtSuspend = new JButton("Suspend");
    private JButton jbtResume = new JButton("Resume");
    private JButton jbtAdd = new JButton("+1");
    private JButton jbtSubtract = new JButton("-1");
    private JScrollBar jsbDelay = new JScrollBar();

    public BallControl() {
      // Group buttons in a panel
      JPanel panel = new JPanel();
      panel.add(jbtSuspend);
      panel.add(jbtResume);
      panel.add(jbtAdd);
      panel.add(jbtSubtract);

      // Add ball and buttons to the panel
      ballPanel.setBorder(
        new javax.swing.border.LineBorder(Color.red));
      jsbDelay.setOrientation(JScrollBar.HORIZONTAL);
      ballPanel.setDelay(jsbDelay.getMaximum());
      setLayout(new BorderLayout());
      add(jsbDelay, BorderLayout.NORTH);
      add(ballPanel, BorderLayout.CENTER);
      add(panel, BorderLayout.SOUTH);
```

```java
      add(ballPanel, BorderLayout.CENTER);
      add(panel, BorderLayout.SOUTH);

      // Register listeners
      jbtSuspend.addActionListener(new Listener());
      jbtResume.addActionListener(new Listener());
      jbtAdd.addActionListener(new Listener());
      jbtSubtract.addActionListener(new Listener());
      jsbDelay.addAdjustmentListener(new AdjustmentListener() {
        @Override
        public void adjustmentValueChanged(AdjustmentEvent e) {
          ballPanel.setDelay(jsbDelay.getMaximum() - e.getValue());
        }
      });
    }

    class Listener implements ActionListener {
      @Override
      public void actionPerformed(ActionEvent e) {
        if (e.getSource() == jbtSuspend)
          ballPanel.suspend();
        else if (e.getSource() == jbtResume)
          ballPanel.resume();
        else if (e.getSource() == jbtAdd)
          ballPanel.add();
        else if (e.getSource() == jbtSubtract)
          ballPanel.subtract();
      }
    }
  }
```

```java
  /** Main method */
  public static void main(String[] args) {
    JFrame frame = new JFrame();
    JApplet applet = new MultipleBallApp();
    frame.add(applet);
    frame.setTitle("MultipleBallApp");
    frame.setLocationRelativeTo(null); // Center the frame
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(400, 200);
    frame.setLocationRelativeTo(null); // Center the frame
    frame.setVisible(true);
  }
}
```

```java
class Ball {
  int x = 0;
  int y = 0; // Current ball position
  int dx = 2; // Increment on ball's x-coordinate
  int dy = 2; // Increment on ball's y-coordinate
  int radius = 5; // Ball radius
  Color color = new Color((int)(Math.random() * 256),
      (int)(Math.random() * 256), (int)(Math.random() * 256));
}
```

```java
class BallPanel extends JPanel {
  private int delay = 10;
  private ArrayList<Ball> list = new ArrayList<Ball>();

  // Create a timer with the initial dalay
  protected Timer timer = new Timer(delay, new ActionListener() {
    @Override /** Handle the action event */
    public void actionPerformed(ActionEvent e) {
      repaint();
    }
  });

  public BallPanel() {
    timer.start();
  }

  public void add() {
    list.add(new Ball());
  }

  public void subtract() {
    if (list.size() > 0)
      list.remove(list.size() - 1); // Remove the last
  }

  @Override
  protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    for (int i = 0; i < list.size(); i++) {
      Ball ball = (Ball)list.get(i); // Get a ball
      g.setColor(ball.color); // Set ball color

      // Check boundaries
      if (ball.x < 0 || ball.x > getWidth())
        ball.dx = -ball.dx;
      if (ball.y < 0 || ball.y > getHeight())
        ball.dy = -ball.dy;

      // Adjust ball position
      ball.x += ball.dx;
      ball.y += ball.dy;
      g.fillOval(ball.x - ball.radius, ball.y - ball.radius,
        ball.radius * 2, ball.radius * 2);
    }
  }

  public void suspend() {
    timer.stop();
  }

  public void resume() {
    timer.start();
  }

  public void setDelay(int delay) {
    this.delay = delay;
    timer.setDelay(delay);
  }
}
```

The Vector and Stack Classes

The Java Collections Framework was introduced with Java 2. Several data structures were supported prior to Java 2. Among them are the Vector class and the Stack class. These classes were redesigned to fit into the Java Collections Framework, but their old-style methods are retained for compatibility. This section introduces the Vector class and the Stack class.

The Vector Class

In Java 2, Vector is the same as ArrayList, except that Vector contains the synchronized methods for accessing and modifying the vector. None of the new collection data structures introduced so far are synchronized. If synchronization is required, you can use the synchronized versions of the collection classes. These classes are introduced later in the section, “The Collections Class.”

The Vector Class, cont.

+Vector()	Creates a default empty vector with initial capacity 10.
+Vector(c: Collection<? extends E>)	Creates a vector from an existing collection.
+Vector(initialCapacity: int)	Creates a vector with the specified initial capacity.
+Vector(initCapacity:int, capacityIncr: int)	Creates a vector with the specified initial capacity and increment.
+addElement(o: E): void	Appends the element to the end of this vector.
+capacity(): int	Returns the current capacity of this vector.
+copyInto(anArray: Object[]): void	Copies the elements in this vector to the array.
+elementAt(index: int): E	Returns the object at the specified index.
+elements(): Enumeration<E>	Returns an enumeration of this vector.
+ensureCapacity(): void	Increases the capacity of this vector.
+firstElement(): E	Returns the first element in this vector.
+insertElementAt(o: E, index: int): void	Inserts o to this vector at the specified index.
+lastElement(): E	Returns the last element in this vector.
+removeAllElements(): void	Removes all the elements in this vector.
+removeElement(o: Object): boolean	Removes the first matching element in this vector.
+removeElementAt(index: int): void	Removes the element at the specified index.
+setElementAt(o: E, index: int): void	Sets a new element at the specified index.
+setSize(newSize: int): void	Sets a new size in this vector.
+trimToSize(): void	Trims the capacity of this vector to its size.

The Stack Class

The Stack class represents a last-in-first-out stack of objects. The elements are accessed only from the top of the stack. You can retrieve, insert, or remove an element from the top of the stack.

+Stack()	Creates an empty stack.
+empty(): boolean	Returns true if this stack is empty.
+peek(): E	Returns the top element in this stack.
+pop(): E	Returns and removes the top element in this stack.
+push(o: E) : E	Adds a new element to the top of this stack.
+search(o: Object) : int	Returns the position of the specified element in this stack.

Queues

A queue is a first-in/first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue.

The Queue Interface

Inserts an element to the queue.

Retrieves and removes the head of this queue, or null if this queue is empty.

Retrieves and removes the head of this queue and throws an exception if this queue is empty.

Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.

Retrieves, but does not remove, the head of this queue, throwing an exception if this queue is empty.

```java
public class TestQueue {
  public static void main(String[] args) {
    java.util.Queue<String> queue =
      new java.util.LinkedList<String>();
    queue.offer("Oklahoma");
    queue.offer("Indiana");
    queue.offer("Georgia");
    queue.offer("Texas");

    while (queue.size() > 0)
      System.out.print(queue.remove() + " ");
  }
}
```

Case Study: Evaluating Expressions

Stacks can be used to evaluate expressions.

```java
import java.util.Stack;

public class EvaluateExpression {
  public static void main(String[] args) {
    // Check number of arguments passed
    if (args.length != 1) {
      System.out.println(
        "Usage: java EvaluateExpression \"expression\"");
      System.exit(1);
    }

    try {
      System.out.println(evaluateExpression(args[
    }
    catch (Exception ex) {
      System.out.println("Wrong expression: " + a
    }
  }
```

```java
/** Process one operator: Take an operator from operatorStack and
 *  apply it on the operands in the operandStack */
public static void processAnOperator(
    Stack<Integer> operandStack, Stack<Character> operatorStack) {
  char op = operatorStack.pop();
  int op1 = operandStack.pop();
  int op2 = operandStack.pop();
  if (op == '+')
    operandStack.push(op2 + op1);
  else if (op == '-')
    operandStack.push(op2 - op1);
  else if (op == '*')
    operandStack.push(op2 * op1);
  else if (op == '/')
    operandStack.push(op2 / op1);
}

public static String insertBlanks(String s) {
  String result = "";

  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == '(' || s.charAt(i) == ')' ||
        s.charAt(i) == '+' || s.charAt(i) == '-' ||
        s.charAt(i) == '*' || s.charAt(i) == '/')
      result += " " + s.charAt(i) + " ";
    else
      result += s.charAt(i);
  }

  return result;
}
}
```

```java
/** Evaluate an expression */
public static int evaluateExpression(String expression) {
  // Create operandStack to store operands
  Stack<Integer> operandStack = new Stack<Integer>();

  // Create operatorStack to store operators
  Stack<Character> operatorStack = new Stack<Character>();

  // Insert blanks around (, ), +, -, /, and *
  expression = insertBlanks(expression);

  // Extract operands and operators
  String[] tokens = expression.split(" ");

  // Phase 1: Scan tokens
  for (String token: tokens) {
    if (token.length() == 0) // Blank space
      continue; // Back to the while loop to extract the next token
    else if (token.charAt(0) == '+' || token.charAt(0) == '-') {
      // Process all +, -, *, / in the top of the operator stack
      while (!operatorStack.isEmpty() &&
        (operatorStack.peek() == '+' ||
         operatorStack.peek() == '-' ||
         operatorStack.peek() == '*' ||
         operatorStack.peek() == '/')) {
        processAnOperator(operandStack, operatorStack);
      }

      // Push the + or - operator into the operator stack
      operatorStack.push(token.charAt(0));
    }
    else if (token.charAt(0) == '*' || token.charAt(0) == '/') {
      // Process all *, / in the top of the operator stack
      while (!operatorStack.isEmpty() &&
        (operatorStack.peek() == '*' ||
        operatorStack.peek() == '/')) {
```

```
        processAnOperator(operandStack, operatorStack);
      }

      // Push the * or / operator into the operator stack
      operatorStack.push(token.charAt(0));
    }
    else if (token.trim().charAt(0) == '(') {
      operatorStack.push('('); // Push '(' to stack
    }
    else if (token.trim().charAt(0) == ')') {
      // Process all the operators in the stack until seeing '('
      while (operatorStack.peek() != '(') {
        processAnOperator(operandStack, operatorStack);
      }

      operatorStack.pop(); // Pop the '(' symbol from the stack
    }
    else { // An operand scanned
      // Push an operand to the stack
      operandStack.push(new Integer(token));
    }
  }

  // Phase 2: process all the remaining operators in the stack
  while (!operatorStack.isEmpty()) {
    processAnOperator(operandStack, operatorStack);
  }

  // Return the result
  return operandStack.pop();
}
```

Chapter 26
Implementing Lists, Stacks, Queues

Objectives

- To design common features of lists in an interface and provide skeleton implementation in an abstract class.
- To design and implement a dynamic list using an array.
- To design and implement a dynamic list using a linked structure.
- To design and implement a stack class using an array list and a queue class using a linked list.

Lists

A list is a popular data structure to store data in sequential order. For example, a list of students, a list of available rooms, a list of cities, and a list of books, etc. can be stored using lists. The common operations on a list are usually the following:

- Retrieve an element from this list.
- Insert a new element to this list.
- Delete an element from this list.
- Find how many elements are in this list.
- Find if an element is in this list.
- Find if this list is empty.

Two Ways to Implement Lists

There are two ways to implement a list.

Using arrays. One is to use an array to store the elements. The array is dynamically created. If the capacity of the array is exceeded, create a new larger array and copy all the elements from the current array to the new array.

Using linked list. The other approach is to use a linked structure. A linked structure consists of nodes. Each node is dynamically created to hold an element. All the nodes are linked together to form a list.

Design of ArrayList and LinkedList

For convenience, let's name these two classes: MyArrayList and MyLinkedList. These two classes have common operations, but different data fields. The common operations can be generalized in an interface or an abstract class. A good strategy is to combine the virtues of interfaces and abstract classes by providing both interface and abstract class in the design so the user can use either the interface or the abstract class whichever is convenient. Such an abstract class is known as a convenience class.

MyList Interface and MyAbstractList Class

Appends a new element at the end of this list.

Adds a new element at the specified index in this list.

Removes all the elements from this list.

Returns true if this list contains the element.

Returns the element from this list at the specified index.

Returns the index of the first matching element in this list.

Returns true if this list contains no elements.

Returns the index of the last matching element in this list.

Removes the element from this list.

Returns the number of elements in this list.

Removes the element at the specified index and returns the removed element.

Sets the element at the specified index and returns the element you are replacing.

The size of the list.

Creates a default list.

Creates a list from an array of objects.

Implements the add method.

Implements the isEmpty method.

Implements the size method.

Implements the remove method.

```java
public interface MyList<E> extends java.lang.Iterable<E> {
  /** Add a new element at the end of this list */
  public void add(E e);

  /** Add a new element at the specified index in this list */
  public void add(int index, E e);

  /** Clear the list */
  public void clear();

  /** Return true if this list contains the element */
  public boolean contains(E e);

  /** Return the element from this list at the specified index */
  public E get(int index);

  /** Return the index of the first matching element in this list.
   *  Return -1 if no match. */
  public int indexOf(E e);

  /** Return true if this list contains no elements */
  public boolean isEmpty();

  /** Return the index of the last matching element in this list
   *  Return -1 if no match. */
  public int lastIndexOf(E e);

  /** Remove the first occurrence of the element
   *  Shift any subsequent elements to the left
   *  Return true if the element is removed. */
  public boolean remove(E e);

  /** Remove the element at the specified position in this list
   *  Shift any subsequent elements to the left.
   *  Return the element that was removed from the list. */
  public E remove(int index);

  /** Replace the element at the specified position in this list
   *  with the specified element and returns the new set. */
  public Object set(int index, E e);

  /** Return the number of elements in this list */
  public int size();

  /** Return an iterator for the list */
  public java.util.Iterator<E> iterator();
}
```

```java
public abstract class MyAbstractList<E> implements MyList<E> {
  protected int size = 0; // The size of the list

  /** Create a default list */
  protected MyAbstractList() {
  }

  /** Create a list from an array of objects */
  protected MyAbstractList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      add(objects[i]);
  }

  @Override /** Add a new element at the end of this list */
  public void add(E e) {
    add(size, e);
  }

  @Override /** Return true if this list contains no elements */
  public boolean isEmpty() {
    return size == 0;
  }

  @Override /** Return the number of elements in this list */
  public int size() {
    return size;
  }
```

```java
  @Override /** Remove the first occurrence of the element e
   *  from this list. Shift any subsequent elements to the left.
   *  Return true if the element is removed. */
  public boolean remove(E e) {
    if (indexOf(e) >= 0) {
      remove(indexOf(e));
      return true;
    }
    else
      return false;
  }
}
```

Array Lists

Array is a fixed-size data structure. Once an array is created, its size cannot be changed. Nevertheless, you can still use array to implement dynamic data structures. The trick is to create a new larger array to replace the current array if the current array cannot hold new elements in the list.

Initially, an array, say data of Object[] type, is created with a default size. When inserting a new element into the array, first ensure there is enough room in the array. If not, create a new array with the size as twice as the current one. Copy the elements from the current array to the new array. The new array now becomes the current array.

Array List Animation

www.cs.armstrong.edu/liang/animation/ArrayListAnimation.html

Insertion

Before inserting a new element at a specified index, shift all the elements after the index to the right and increase the list size by 1.

Deletion

To remove an element at a specified index, shift all the elements after the index to the left by one position and decrease the list size by 1.

Implementing MyArrayList

Creates a default array list.

Creates an array list from an array of objects.

Doubles the current array size if needed.

MyArrayList

TestMyArrayList

Linked Lists

Since MyArrayList is implemented using an array, the methods get(int index) and set(int index, Object o) for accessing and modifying an element through an index and the add(Object o) for adding an element at the end of the list are efficient. However, the methods add(int index, Object o) and remove(int index) are inefficient because it requires shifting potentially a large number of elements. You can use a linked structure to implement a list to improve efficiency for adding and removing an element anywhere in a list.

```java
public class MyArrayList<E> extends MyAbstractList<E> {
  public static final int INITIAL_CAPACITY = 16;
  private E[] data = (E[])new Object[INITIAL_CAPACITY];

  /** Create a default list */
  public MyArrayList() {
  }

  /** Create a list from an array of objects */
  public MyArrayList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      add(objects[i]); // Warning: don't use super(object
  }

  @Override /** Add a new element at the specified index
  public void add(int index, E e) {
    ensureCapacity();

    // Move the elements to the right after the specified
    for (int i = size - 1; i >= index; i--)
      data[i + 1] = data[i];

    // Insert new element to data[index]
    data[index] = e;

    // Increase size by 1
    size++;
  }
```

```java
/** Create a new larger array, double the current size + 1 */
private void ensureCapacity() {
  if (size >= data.length) {
    E[] newData = (E[])(new Object[size * 2 + 1]);
    System.arraycopy(data, 0, newData, 0, size);
    data = newData;
  }
}

@Override /** Clear the list */
public void clear() {
  data = (E[])new Object[INITIAL_CAPACITY];
  size = 0;
}

@Override /** Return true if this list contains the element */
public boolean contains(E e) {
  for (int i = 0; i < size; i++)
    if (e.equals(data[i])) return true;

  return false;
}

@Override /** Return the element at the specified index */
public E get(int index) {
  checkIndex(index);
  return data[index];
}

private void checkIndex(int index) {
  if (index < 0 || index >= size)
    throw new IndexOutOfBoundsException
      ("Index: " + index + ", Size: " + size);
}
```

```java
@Override /** Return the index of the first matching element
 *  in this list. Return -1 if no match. */
public int indexOf(E e) {
  for (int i = 0; i < size; i++)
    if (e.equals(data[i])) return i;

  return -1;
}

@Override /** Return the index of the last matching ele
 *  in this list. Return -1 if no match. */
public int lastIndexOf(E e) {
  for (int i = size - 1; i >= 0; i--)
    if (e.equals(data[i])) return i;

  return -1;
}

@Override /** Remove the element at the specified posit
 *  in this list. Shift any subsequent elements to the
 *  Return the element that was removed from the list.
public E remove(int index) {
  checkIndex(index);

  E e = data[index];

  // Shift data to the left
  for (int j = index; j < size - 1; j++)
    data[j] = data[j + 1];

  data[size - 1] = null; // This element is now null

  // Decrement size
  size--;

  return e;
}
```

```java
@Override /** Replace the element at the specified position
 *  in this list with the specified element. */
public E set(int index, E e) {
  checkIndex(index);
  E old = data[index];
  data[index] = e;
  return old;
}

@Override
public String toString() {
  StringBuilder result = new StringBuilder("[");

  for (int i = 0; i < size; i++) {
    result.append(data[i]);
    if (i < size - 1) result.append(", ");
  }

  return result.toString() + "]";
}

/** Trims the capacity to current size */
public void trimToSize() {
  if (size != data.length) {
    E[] newData = (E[])(new Object[size]);
    System.arraycopy(data, 0, newData, 0, size);
    data = newData;
  } // If size == capacity, no need to trim
}

@Override /** Override iterator() defined in Iterable */
public java.util.Iterator<E> iterator() {
  return new ArrayListIterator();
}
```

```java
  private class ArrayListIterator
      implements java.util.Iterator<E> {
    private int current = 0; // Current index

    @Override
    public boolean hasNext() {
      return (current < size);
    }

    @Override
    public E next() {
      return data[current++];
    }

    @Override
    public void remove() {
      MyArrayList.this.remove(current);
    }
  }
}
```

```java
public class TestMyArrayList {
  public static void main(String[] args) {
    // Create a list
    MyList<String> list = new MyArrayList<String>();

    // Add elements to the list
    list.add("America"); // Add it to the list
    System.out.println("(1) " + list);

    list.add(0, "Canada"); // Add it to the beginning of the list
    System.out.println("(2) " + list);

    list.add("Russia"); // Add it to the end of the list
    System.out.println("(3) " + list);

    list.add("France"); // Add it to the end of the list
    System.out.println("(4) " + list);

    list.add(2, "Germany"); // Add it to the list at index 2
    System.out.println("(5) " + list);

    list.add(5, "Norway"); // Add it to the list at index 5
    System.out.println("(6) " + list);

    // Remove elements from the list
    list.remove("Canada"); // Same as list.remove(0) in this case
    System.out.println("(7) " + list);

    list.remove(2); // Remove the element at index 2
    System.out.println("(8) " + list);

    list.remove(list.size() - 1); // Remove the last element
    System.out.print("(9) " + list + "\n(10) ");

    for (String s: list)
      System.out.print(s.toUpperCase() + " ");
  }
}
```

Linked List Animation

www.cs.armstrong.edu/liang/animation/LinkedListAnimation.html

Nodes in Linked Lists

A linked list consists of nodes. Each node contains an element, and each node is linked to its next neighbor. Thus a node can be defined as a class, as follows:

```
class Node<E> {
  E element;
  Node next;

  public Node(E o) {
    element = o;
  }
}
```

Adding Three Nodes

The variable head refers to the first node in the list, and the variable tail refers to the last node in the list. If the list is empty, both are null. For example, you can create three nodes to store three strings in a list, as follows:

Step 1: Declare head and tail:

```
Node<String> head = null;
Node<String> tail = null;
```

The list is empty now

Adding Three Nodes, cont.

Step 2: Create the first node and insert it to the list:

```
head = new Node<String>("Chicago");
tail = head;
```

After the first node is inserted

Adding Three Nodes, cont.

Step 3: Create the second node and insert it to the list:

```
tail.next = new Node<String>("Denver");
```

```
tail  = tail.next;
```

Adding Three Nodes, cont.

Step 4: Create the third node and insert it to the list:

```
tail.next =
  new Node<String>("Dallas");
```

```
tail = tail.next;
```

Traversing All Elements in the List

Each node contains the element and a data field named *next* that points to the next element. If the node is the last in the list, its pointer data field next contains the value null. You can use this property to detect the last node. For example, you may write the following loop to traverse all the nodes in the list.

```
Node<E> current = head;
while (current != null) {
  System.out.println(current.element);
  current = current.next;
}
```

MyLinkedList

Creates a default linked list.

Creates a linked list from an array of objects.

Adds the object to the head of the list.

Adds the object to the tail of the list.

Returns the first object in the list.

Returns the last object in the list.

Removes the first object from the list.

Removes the last object from the list.

```java
public class MyLinkedList<E> extends MyAbstractList<E> {
  private Node<E> head, tail;

  /** Create a default list */
  public MyLinkedList() {
  }

  /** Create a list from an array of objects */
  public MyLinkedList(E[] objects) {
    super(objects);
  }

  /** Return the head element in the list */
  public E getFirst() {
    if (size == 0) {
      return null;
    }
    else {
      return head.element;
    }
  }

  /** Return the last element in the list */
  public E getLast() {
    if (size == 0) {
      return null;
    }
    else {
      return tail.element;
    }
  }
```

```java
  /** Add an element to the beginning of the list */
  public void addFirst(E e) {
    Node<E> newNode = new Node<E>(e); // Create a new node
    newNode.next = head; // link the new node with the head
    head = newNode; // head points to the new node
    size++; // Increase list size

    if (tail == null) // the new node is the only node in list
      tail = head;
  }

  /** Add an element to the end of the list */
  public void addLast(E e) {
    Node<E> newNode = new Node<E>(e); // Create a new for element e

    if (tail == null) {
      head = tail = newNode; // The new node is the only node in list
    }
    else {
      tail.next = newNode; // Link the new with the last node
      tail = tail.next; // tail now points to the last node
    }

    size++; // Increase size
  }
```

```java
@Override /** Add a new element at the specified index
 * in this list. The index of the head element is 0 */
public void add(int index, E e) {
  if (index == 0) {
    addFirst(e);
  }
  else if (index >= size) {
    addLast(e);
  }
  else {
    Node<E> current = head;
    for (int i = 1; i < index; i++) {
      current = current.next;
    }
    Node<E> temp = current.next;
    current.next = new Node<E>(e);
    (current.next).next = temp;
    size++;
  }
}

/** Remove the head node and
 *  return the object that is contained in the r
public E removeFirst() {
  if (size == 0) {
    return null;
  }
  else {
    Node<E> temp = head;
    head = head.next;
    size--;
    if (head == null) {
      tail = null;
    }
    return temp.element;
  }
}
```

```java
/** Remove the last node and
 * return the object that is contained in the removed node. */
public E removeLast() {
  if (size == 0) {
    return null;
  }
  else if (size == 1) {
    Node<E> temp = head;
    head = tail = null;
    size = 0;
    return temp.element;
  }
  else {
    Node<E> current = head;

    for (int i = 0; i < size - 2; i++) {
      current = current.next;
    }

    Node<E> temp = tail;
    tail = current;
    tail.next = null;
    size--;
    return temp.element;
  }
}
```

```java
@Override /** Remove the element at the specified position in this
 *  list. Return the element that was removed from the list. */
public E remove(int index) {
  if (index < 0 || index >= size) {
    return null;
  }
  else if (index == 0) {
    return removeFirst();
  }
  else if (index == size - 1) {
    return removeLast();
  }
  else {
    Node<E> previous = head;

    for (int i = 1; i < index; i++) {
      previous = previous.next;
    }

    Node<E> current = previous.next;
    previous.next = current.next;
    size--;
    return current.element;
  }
}
```

```java
@Override /** Override toString() to return elements in the list */
public String toString() {
  StringBuilder result = new StringBuilder("[");

  Node<E> current = head;
  for (int i = 0; i < size; i++) {
    result.append(current.element);
    current = current.next;
    if (current != null) {
      result.append(", "); // Separate two elements with a comma
    }
    else {
      result.append("]"); // Insert the closing ] in the string
    }
  }

  return result.toString();
}

@Override /** Clear the list */
public void clear() {
  size = 0;
  head = tail = null;
}

@Override /** Return true if this list contains the element e */
public boolean contains(E e) {
  System.out.println("Implementation left as an exercise");
  return true;
}

@Override /** Return the element at the specified index */
public E get(int index) {
  System.out.println("Implementation left as an exercise");
  return null;
}
```

```java
@Override /** Return the index of the head matching element in
 *  this list. Return -1 if no match. */
public int indexOf(E e) {
  System.out.println("Implementation left as an exercise");
  return 0;
}

@Override /** Return the index of the last matching element in
 *  this list. Return -1 if no match. */
public int lastIndexOf(E e) {
  System.out.println("Implementation left as an exercise");
  return 0;
}

@Override /** Replace the element at the specified posi
 *  in this list with the specified element. */
public E set(int index, E e) {
  System.out.println("Implementation left as an exercis
  return null;
}

@Override /** Override iterator() defined in Iterable
public java.util.Iterator<E> iterator() {
  return new LinkedListIterator();
}

private void checkIndex(int index) {
  if (index < 0 || index >= size)
    throw new IndexOutOfBoundsException
      ("Index: " + index + ", Size: " + size);
}
```

```java
private class LinkedListIterator
    implements java.util.Iterator<E> {
  private Node<E> current = head; // Current index

  @Override
  public boolean hasNext() {
    return (current != null);
  }

  @Override
  public E next() {
    E e = current.element;
    current = current.next;
    return e;
  }

  @Override
  public void remove() {
    System.out.println("Implementation left as an exercise");
  }
}

private static class Node<E> {
  E element;
  Node<E> next;

  public Node(E element) {
    this.element = element;
  }
}
}
```

```java
public class TestMyLinkedList {
  /** Main method */
  public static void main(String[] args) {
    // Create a list for strings
    MyLinkedList<String> list = new MyLinkedList<String>();

    // Add elements to the list
    list.add("America"); // Add it to the list
    System.out.println("(1) " + list);

    list.add(0, "Canada"); // Add it to the beginning of the list
    System.out.println("(2) " + list);

    list.add("Russia"); // Add it to the end of the list
    System.out.println("(3) " + list);

    list.addLast("France"); // Add it to the end of the list
    System.out.println("(4) " + list);

    list.add(2, "Germany"); // Add it to the list at index 2
    System.out.println("(5) " + list);

    list.add(5, "Norway"); // Add it to the list at index 5
    System.out.println("(6) " + list);

    list.add(0, "Poland"); // Same as list.addFirst("Poland")
    System.out.println("(7) " + list);

    // Remove elements from the list
    list.remove(0); // Same as list.remove("Australia") in this case
    System.out.println("(8) " + list);
```

```java
    list.remove(2); // Remove the element at index 2
    System.out.println("(9) " + list);

    list.remove(list.size() - 1); // Remove the last element
    System.out.print("(10) " + list + "\n(11) ");

    for (String s: list)
      System.out.print(s.toUpperCase() + " ");

    list.clear();
    System.out.println("\nAfter clearing the list, the list size is "
      + list.size());
  }
}
```

Implementing addFirst(E o)

```
public void addFirst(E o) {
  Node<E> newNode = new
Node<E>(o);
  newNode.next = head;
  head = newNode;
  size++;
  if (tail == null)
    tail = head;
}
```

Implementing addLast(E o)

```
public void addLast(E o) {
  if (tail == null) {
    head = tail = new Node<E>(element);
  }
  else {
    tail.next = new Node(element);
    tail = tail.next;
  }
  size++;
}
```

Implementing add(int index, E o)

```
public void add(int index, E o) {
  if (index == 0) addFirst(o);
  else if (index >= size) addLast(o);
  else {
    Node<E> current = head;
    for (int i = 1; i < index; i++)
      current = current.next;
    Node<E> temp = current.nex
    current.next = new Node<E>
    (current.next).next = temp;
    size++;
  }
}
```

(a) Before a new node is inserted.

(b) After a new node is inserted.

Implementing removeFirst()

```
public E removeFirst() {
  if (size == 0) return null;
  else {
    Node<E> temp = head;
    head = head.next;
    size--;
    if (head == null) tail = null;
    return temp.element;
  }
}
```

(a) Before the node is deleted.

(b) After the first node is deleted

Implementing removeLast()

```
public E removeLast() {
  if (size == 0) return null;
  else if (size == 1)
  {
    Node<E> temp = head;
    head = tail = null;
    size = 0;
    return temp.element;
  }
  else
  {
    Node<E> current = head;
    for (int i = 0; i < size - 2; i++)
      current = current.next;
    Node temp = tail;
    tail = current;
    tail.next = null;
    size--;
    return temp.element;
  }
}
```

(a) Before the node is deleted.

(b) After the last node is deleted

Implementing remove(int index)

```
public E remove(int index) {
  if (index < 0 || index >= size) return null;
  else if (index == 0) return removeFirst();
  else if (index == size - 1) return
removeLast();
  else {
    Node<E> previous = head;
    for (int i = 1; i < index; i++) {
      previous = previous.next;
    }
    Node<E> current = previous.r
    previous.next = current.next;
    size--;
    return current.element;
  }
}
```

(a) Before the node is deleted.

(b) After the node is deleted.

Time Complexity for ArrayList and LinkedList

Methods	MyArrayList/ArrayList	MyLinkedList/LinkedList
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`get(index: int)`	$O(1)$	$O(n)$
`indexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`size()`	$O(1)$	$O(1)$
`remove(index: int)`	$O(n)$	$O(n)$
`set(index: int, e: E)`	$O(n)$	$O(n)$
`addFirst(e: E)`	$O(n)$	$O(1)$
`removeFirst()`	$O(n)$	$O(1)$

Circular Linked Lists

- A *circular, singly linked list* is like a singly linked list, except that the pointer of the last node points back to the first node.

Doubly Linked Lists

- A *doubly linked list* contains the nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. So, a doubly linked list can be traversed forward and backward.

Circular Doubly Linked Lists

- A *circular, doubly linked list* is doubly linked list, except that the forward pointer of the last node points to the first node and the backward pointer of the first pointer points to the last node.

Stacks

A stack can be viewed as a special type of list, where the elements are accessed, inserted, and deleted only from the end, called the top, of the stack.

Queues

A queue represents a waiting list. A queue can be viewed as a special type of list, where the elements are inserted into the end (tail) of the queue, and are accessed and deleted from the beginning (head) of the queue.

Stack Animation

www.cs.armstrong.edu/liang/animation/StackAnimation.html

Queue Animation

www.cs.armstrong.edu/liang/animation/QueueAnimation.html

Implementing Stacks and Queues

- Using an array list to implement Stack
- Use a linked list to implement Queue

Since the insertion and deletion operations on a stack are made only at the end of the stack, using an array list to implement a stack is more efficient than a linked list. Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list. This section implements a stack class using an array list and a queue using a linked list.

Design of the Stack and Queue Classes

There are two ways to design the stack and queue classes:

- Using inheritance: You can define the stack class by extending the array list class, and the queue class by extending the linked list class.

- Using composition: You can define an array list as a data field in the stack class, and a linked list as a data field in the queue class.

Composition is Better

Both designs are fine, but using composition is better because it enables you to define a complete new stack class and queue class without inheriting the unnecessary and inappropriate methods from the array list and linked list.

MyStack and MyQueue

GenericStack<E>	
-list: java.util.ArrayList<E>	An array list to store elements.
+GenericStack()	Creates an empty stack.
+getSize(): int	Returns the number of elements in this stack.
+peek(): E	Returns the top element in this stack.
+pop(): E	Returns and removes the top element in this stack.
+push(o: E): void	Adds a new element to the top of this stack.
+isEmpty(): boolean	Returns true if the stack is empty.

GenericStack

GenericQueue<E>	
-list: MyLinkedList<E>	
+enqueue(e: E): void	Adds an element to this queue.
+dequeue(): E	Removes an element from this queue.
+getSize(): int	Returns the number of elements from this queue.

GenericQueue

Example: Using Stacks and Queues

Write a program that creates a stack using MyStack and a queue using MyQueue. It then uses the push (enqueu) method to add strings to the stack (queue) and the pop (dequeue) method to remove strings from the stack (queue).

```java
public class GenericQueue<E> {
  private java.util.LinkedList<E> list
    = new java.util.LinkedList<E>();

  public void enqueue(E e) {
    list.addLast(e);
  }

  public E dequeue() {
    return list.removeFirst();
  }

  public int getSize() {
    return list.size();
  }

  @Override
  public String toString() {
    return "Queue: " + list.toString();
  }
}
```

```java
public class GenericStack<E> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<E>();

  public int getSize() {
    return list.size();
  }

  public E peek() {
    return list.get(getSize() - 1);
  }

  public void push(E o) {
    list.add(o);
  }

  public E pop() {
    E o = list.get(getSize() - 1);
    list.remove(getSize() - 1);
    return o;
  }

  public boolean isEmpty() {
    return list.isEmpty();
  }

  @Override
  public String toString() {
        return "stack: " + list.toString();
  }
}
```

```java
public class TestStackQueue {
  public static void main(String[] args) {
    // Create a stack
    GenericStack<String> stack = new GenericStack<String>();
    // Add elements to the stack
    stack.push("Tom"); // Push it to the stack
    System.out.println("(1) " + stack);
    stack.push("Susan"); // Push it to the the stack
    System.out.println("(2) " + stack);
    stack.push("Kim"); // Push it to the stack
    stack.push("Michael"); // Push it to the stack
    System.out.println("(3) " + stack);
    // Remove elements from the stack
    System.out.println("(4) " + stack.pop());
    System.out.println("(5) " + stack.pop());
    System.out.println("(6) " + stack);
    // Create a queue
    GenericQueue<String> queue = new GenericQueue<String>();
    // Add elements to the queue
    queue.enqueue("Tom"); // Add it to the queue
    System.out.println("(7) " + queue);
    queue.enqueue("Susan"); // Add it to the queue
    System.out.println("(8) " + queue);
    queue.enqueue("Kim"); // Add it to the queue
    queue.enqueue("Michael"); // Add it to the queue
    System.out.println("(9) " + queue);
    // Remove elements from the queue
    System.out.println("(10) " + queue.dequeue());
    System.out.println("(11) " + queue.dequeue());
    System.out.println("(12) " + queue);
  }
}
```