1. Arrays

Overview

- Fixed length once initialized.
- Add or remove element at a specified position without shifting the other elements to make room/fill in the resulting gap.
- One feature that the array data structure provides that these classes (List classes) don't is the ability to store primitive-type values. The List classes all store references to Objects, so all primitive-type values must be wrapped in objects.
    - Note that all generic classes in Java extends `Object` class.

NOTE: Circular array implementation will be shown in `Queue` chapter.

Performance

Operations/Complexity	Best-Case	Average-Case	Worst-Case
Add	O(1) (add to the end)	O(N)	O(N)
Delete	O(1) (delete from the end)	O(N)	O(N)
Search	O(1)	O(N)	O(N)
Get	O(1)	O(1)	O(1)

Class `java.util.Arrays`

Method Summary

Modifier and Type	Method	Description
static boolean	equals(Object[] a, Object[] a2)	Returns true if the two specified arrays of Objects are equal to one another.
static void	fill(Object[] a, Object val)	Assigns the specified Object reference to each element of the specified array of Objects.
static <T> T[]	copyOf(T[] original, int newLength)	Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length.
static <T> T[]	copyOfRange(T[] original, int from, int to)	Copies the specified range of the specified array into a new array.

Modifier and Type	Method	Description
static String	toString(Object[] a)	Returns a string representation of the contents of the specified array.
static void	sort(Object[] a)	Sorts the specified array of objects into ascending order, according to the natural ordering of its elements.
static <T> void	sort(T[] a, Comparator<? super T> c)	Sorts the specified array of objects according to the order induced by the specified comparator.
int	size()	Returns the number of elements in this list.
static int	binarySearch(Object[] a, Object key)	Searches the specified array for the specified object using the binary search algorithm.

Reference: https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

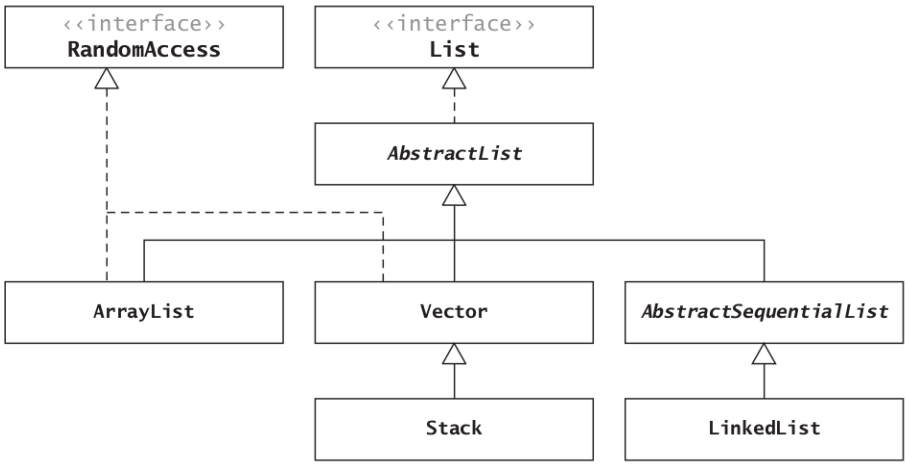
```
// A more general method of copying an array
System.arraycopy(source, sourcePos, destination, destPos, numElements);
```

2. List Interface

Modifier and Type	Method	Description
boolean	add(E e)	Appends the specified element to the end of this list (optional operation).
void	add(int index, E element)	Inserts the specified element at the specified position in this list (optional operation).
E	get(int index)	Returns the element at the specified position in this list.
int	indexOf(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
Iterator<E>	iterator()	Returns an iterator over the elements in this list in proper sequence.
E	remove(int index)	Removes the element at the specified position in this list (optional operation).

Modifier and Type	Method	Description
boolean	remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present (optional operation).
E	set(int index, E element)	Replaces the element at the specified position in this list with the specified element (optional operation).
int	size()	Returns the number of elements in this list.
Object[]	toArray()	Returns an array containing all of the elements in this list in proper sequence (from first to last element).

FIGURE 2.4
The `java.util.List` Interface and Its Implementers



3. ArrayList

Overview

- Java uses `array` internally to contain the data of a `ArrayList`.
- The physical size of the array is indicated by the data field `capacity`.
- The number of data items is indicated by the data field `size`.
- The data type of the references stored in the underlying array theData (type E[]) is also determined when the `MyArrayList` object is declared. If no parameter type is specified, the implicit parameter type is Object, and the underlying data array is type Object[].

Class `java.util.ArrayList<E>`

Modifier and Type	Method and Description
boolean	`add(E e)` Appends the specified element to the end of this list.

Modifier and Type	Method and Description
void	`add(int index, E element)` Inserts the specified element at the specified position in this list.
E	get(int index) Returns the element at the specified position in this list.
Iterator<E>	iterator() Returns an iterator over the elements in this list in proper sequence.
E	remove(int index) Removes the element at the specified position in this list.
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present.
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	size() Returns the number of elements in this list.
Object[]	toArray() Returns an array containing all of the elements in this list in proper sequence (from first to last element).

Implementation

```
import java.util.AbstractList;
import java.util.Arrays;
import java.util.List;

public class MyArrayList<E> extends AbstractList<E> implements List<E> {

    // Data Fields
    /** The default initial capacity */
    private static final int INITIAL_CAPACITY = 10;
    /** The underlying data array */
    private E[] data;
    /** The current size */
    private int size = 0;
    /** The current capacity */
    private int capacity = 0;

    @SuppressWarnings("unchecked")
    public MyArrayList(){
        this.capacity = INITIAL_CAPACITY;
        data = (E[]) new Object[capacity];
```

```
    }

    public boolean add(E anEntry) {
        if (size == capacity) {
            reallocate();
        }
        data[size] = anEntry;
        size++;
        return true;
    }

    public void add(int index, E anEntry) {
        if (index < 0 || index > size) {
            throw new ArrayIndexOutOfBoundsException(index);
        }
        if (size == capacity) {
            reallocate();
        }
        // Shift data in elements from index to size - 1
        for (int i = size; i > index; i--) {
            data[i] = data[i - 1];
        }
        // Insert the new item.
        data[index] = anEntry;
        size++;
    }

    public E get(int index) {
        if (index < 0 || index >= size) {
            throw new ArrayIndexOutOfBoundsException(index);
        }
        return data[index];
    }
    public E set(int index, E newValue) {
        if (index < 0 || index >= size) {
            throw new ArrayIndexOutOfBoundsException(index);
        }
        E oldValue = data[index];
        data[index] = newValue;
        return oldValue;
    }

    public E remove(int index) {
        if (index < 0 || index >= size) {
            throw new ArrayIndexOutOfBoundsException(index);
        }
        E returnValue = data[index];
        for (int i = index + 1; i < size; i++) {
            data[i - 1] = data[i];
        }
        size--;
        return returnValue;
    }

    private void reallocate() {
        capacity = 2 * capacity;
```

```
            data = Arrays.copyOf(data, capacity);
        }

        public int size() {
            return this.size;
        }
    }
```

Performance

Operations/Complexity	Big O (Worst-Case Complexity)
add(E e)	O(1)
add(int index, E element)	O(N)
get(int index)	O(1)
set(int index, E element)	O(1)
remove(int index)	O(N)
remove(Object o)	O(N)
size()	O(1)

- Recall that when we reallocate the array, we double its size. Doubling an array of size n allows us to add n more items before we need to do another array copy. Therefore, we can add n new items after we have copied over n existing items. This averages out to 1 copy per add. Therefore, reallocation is effectively an O(1) operation, so the insertion (to any index) is still O(n).

Extra

> ### Syntax: Creating a Generic Collection
>
> **FORM:**
>
> CollectionClassName<E> variable = new CollectionClassName<>();
> CollectionClassName<E> variable = new CollectionClassName<E>();
>
> **EXAMPLE:**
>
> List<Person> people = new ArrayList<>();
> List<String> myList = new ArrayList<String>();
> ArrayList<Integer> numList = new ArrayList<>();
>
> **MEANING:**
> An initially empty CollectionClassName<E> object is created that can be used to store references to objects of type E (the type parameter). The actual object type stored in an object of type CollectionClassName<E> is specified when the object is created. If the CollectionClassName on the left is an interface, the CollectionClassName on the right must be a class that implements it. Otherwise, it must be the same class or a subclass of the one

on the left.
The examples above show different ways to create an ArrayList. In this text, we normally specify the interface name on the left of the = operator and the implementing class name on the right as shown in the first two examples. Since the type parameter E must be the same on both sides of the assignment operator, Java 7 introduced the diamond operator <> which eliminates the need to specify the type parameter twice. We will follow this convention. In some cases, we will declare the variable type in one statement and create it in a later statement. In earlier versions of Java, generic collections were not supported. In these versions, you use the statement

```
List yourList = new ArrayList();
```

to create an initially empty ArrayList. Each element of yourList is a type Object reference. The data types of the actual objects referenced by elements of yourList are not specified, and in fact, different elements can reference objects of different types. Use of the adjective "generic" is a bit confusing. A nongeneric collection in Java is very general in that it can store objects of different data types. A generic collection, however, can store objects of one specified data type only. Therefore, generics enable the compiler to do more strict type checking to detect errors at compile time instead of at run time. They also eliminate the need to downcast from type Object to a specific type. For these reasons, we will always use generic collections.

Constructor Declaration for Generic Classes

The constructor declaration follows. Because the constructor is for a generic class, the type parameter <E> is implied but it must not appear in the constructor heading.

```
public MyArrayList() {
    capacity = INITIAL_CAPACITY;
    data = (E[]) new Object[capacity];
}
```

The statement

```
data = (E[]) new Object[capacity];
```

allocates storage for an array with type Object references and then casts this array object to type E[] so that it is type compatible with variable theData. Because the actual type corresponding to E is not known, the compiler issues the warning message: MyArrayList.java uses unchecked or unsafe operations. Don't be concerned about this warning—everything is fine.

PITFALL

Declaring a Generic Array

Rather than use the approach shown in the above constructor, you might try to create a generic array directly using the statement

```
theData = new E[capacity]; // Invalid generic array type.
```

However, this statement will not compile because Java does not allow you to create an array with an unspecified type. Remember, E is a type parameter that is not specified until a generic ArrayList object is created. Therefore, the constructor must create an array of type Object[] since Object is the superclass of all types and then downcast this array object to type E[].

4. Single-Linked Lists

Overview

- Java does not have a class that implements single-linked lists. Instead, it has a more general double-linked list class.

Class `LinkedList<E>`

Modifier and Type	Method	Description
boolean	add(E e)	Appends the specified element to the end of this list.
void	add(int index, E element)	Inserts the specified element at the specified position in this list.
E	get(int index)	Returns the element at the specified position in this list.
int	indexOf(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
Iterator<E>	iterator()	Returns an iterator over the elements in this list in proper sequence.
E	remove(int index)	Removes the element at the specified position in this list.
boolean	remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present.
E	set(int index, E element)	Replaces the element at the specified position in this list with the specified element.
int	size()	Returns the number of elements in this list.
Object[]	toArray()	Returns an array containing all of the elements in this list in proper sequence (from first to last element).

- Note that the `LinkedList` class, part of Java API package `java.util`, is a double-linked list. However, the API description above is still valid.

Implementation

```
import java.util.AbstractSequentialList;
import java.util.List;
import java.util.ListIterator;

/** Class to represent a linked list with a link from each node to the next
 node. SingleLinkedList does not implement the List interface.
 */

public class SingleLinkedList<E> extends AbstractSequentialList<E> implements
List<E> {

    /** A Node is the building block for a single-linked list. */
    private static class Node<E> {
        // Data Fields
        /** The reference to the data. */
        private E data;
        /** The reference to the next node. */
        private Node<E> next;

        // Constructors
        /** Creates a new node with a null next field.
         @param dataItem The data stored
         */
        private Node(E dataItem) {
            data = dataItem;
            next = null;
        }

        /** Creates a new node that references another node.
         @param dataItem The data stored
         @param nodeRef The node referenced by new node
         */
        private Node(E dataItem, Node<E> nodeRef) {
            data = dataItem;
            next = nodeRef;
        }
    }

    /**
     * Reference to list head.
     */
    private Node<E> head = null;
    /**
     * The number of items in the list
     */
    private int size = 0;

    /** Insert the specified item at index
     @param index The position where item is to be inserted
     @param item The item to be inserted
     @throws IndexOutOfBoundsException if index is out of range
     */
    public void add(int index, E item) {
```

```
        if (index < 0 || index > size) {
            throw new IndexOutOfBoundsException(Integer.toString(index));
        }
        if (index == 0) {
            addFirst(item);
        } else {
            Node<E> node = getNode(index-1);
            addAfter(node, item);
        }
    }

    /** Append item to the end of the list
     @param item The item to be appended
     @return true (as specified by the Collection interface)
     */
    public boolean add(E item) {
        add(size, item);
        return true;
    }

    /** Add an item to the front of the list.
     @param item The item to be added
     */
    public void addFirst(E item) {
        head = new Node<>(item, head);
        size++;
    }

    /** Add a node after a given node
     @param node The node preceding the new item
     @param item The item to insert
     */
    private void addAfter(Node<E> node, E item) {
        node.next = new Node<>(item, node.next);
        size++;
    }

    /**
     * Returns the list iterator object.
     * @param i the index of the iterator.
     * @return the list iterator object.
     */
    @Override
    public ListIterator<E> listIterator(int i) {
        throw new UnsupportedOperationException();
    }

    /**
     * Removes the element at the specified position in this list.
     *
     * @param index the index of the element to be removed
     * @return the element previously at the specified position
     * @throws IndexOutOfBoundsException if the index is out of range
     */
    public E remove(int index) {
        if (index < 0 || index >= size) {
```

```
            throw new IndexOutOfBoundsException(Integer.toString(index));
        }

        E removedData;
        if (index == 0) {
            removedData = removeFirst();
        } else {
            Node<E> prevNode = getNode(index - 1);
            removedData = removeAfter(prevNode);
        }
        return removedData;
    }

    /** Remove the node after a given node
     @param node The node before the one to be removed
     @return The data from the removed node, or null
     if there is no node to remove
     */
    private E removeAfter(Node<E> node) {
        Node<E> temp = node.next;
        if (temp != null) {
            node.next = temp.next;
            size--;
            return temp.data;
        } else {
            return null;
        }
    }

    /** Remove the first node from the list
     @return The removed node's data or null if the list is empty
     */
    private E removeFirst() {
        Node<E> temp = head;
        if (head != null) {
            head = head.next;
        }
        // Return data at old head or null if list is empty
        if (temp != null) {
            size--;
            return temp.data;
        } else {
            return null;
        }
    }

    /** Get the data at index
     @param index The position of the data to return
     @return The data at index
     @throws IndexOutOfBoundsException if index is out of range
     */
    public E get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException(Integer.toString(index));
        }
        Node<E> node = getNode(index);
```

```
        return node.data;
    }

    /** Find the node at a specified position
     @param index The position of the node sought
     @return The node at index or null if it does not exist
     */
    private Node<E> getNode(int index) {
        Node<E> node = head;
        for (int i = 0; i < index && node != null; i++) {
            node = node.next;
        }
        return node;
    }

    /** Store a reference to anEntry in the element at position index.
     @param index The position of the item to change
     @param newValue The new data
     @return The data previously at index
     @throws IndexOutOfBoundsException if index is out of range
     */
    public E set(int index, E newValue) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException(Integer.toString(index));
        }
        Node<E> node = getNode(index);
        E result = node.data;
        node.data = newValue;
        return result;
    }

    /**
     * Return the current number of elements in the list.
     * @return The size of the list
     */
    public int size(){
        return size;
    }

    /**
     * Returns the index of the first occurrence of the specified element in this
list,
     * or -1 if this list does not contain the element. This method allows
searching for
     * an object of type Object.
     *
     * @param target the object to search for in the list
     * @return the index of the first occurrence of the object in the list, or -1
if not found
     */
    public int indexOf(Object target) {
        Node<E> current = head;
        int index = 0;
        while (current != null) {
            if (current.data.equals(target)) {
                return index;
```

```
            }
            current = current.next;
            index++;
        }
        return -1; // Return -1 if the element is not found
    }

}
```

Performance

Operations/Complexity	Big O (Worst-Case Complexity)
add(E e)	O(N) - Insertion at the beginning: O(1) - Insertion in the end: O(N) - Insertion into the middle: O(N)
add(int index, E element)	O(N)
get(int index)	O(N)
set(int index, E element)	O(N)
remove(int index)	O(N)
remove(Object o)	O(N)
size()	O(1)

Extra

> The keyword static in the class header indicates that the Node class will not reference its outer class. (It can't because it has no methods other than constructors.) In the Java API documentation, static inner classes are also called nested classes.
> Generally, we want to keep the details of the Node class private. Thus, the qualifier private is applied to the class as well as to the data fields and the constructor. However, the data fields and methods of an inner class are visible anywhere within the enclosing class (also called the parent class).

5. The `LinkedList` Class

Overview

- The `LinkedList` class, part of the Java API package `java.util`, is a double-linked list that implements the `List` interface.

Class `java.util.LinkedList<E>`

Modifier and Type	Method	Description
boolean	add(E e)	Appends the specified element to the end of this list.
void	add(int index, E element)	Inserts the specified element at the specified position in this list.
void	addFirst(E e)	Inserts the specified element at the beginning of this list.
void	addLast(E e)	Appends the specified element to the end of this list.
E	get(int index)	Returns the element at the specified position in this list.
E	getFirst()	Returns the first element in this list.
E	getLast()	Returns the last element in this list.
int	indexOf(Object o)	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
Iterator<E>	iterator()	Returns an iterator over the elements in this list in proper sequence.
E	remove(int index)	Removes the element at the specified position in this list.
boolean	remove(Object o)	Removes the first occurrence of the specified element from this list, if it is present.
E	remove()	Retrieves and removes the head (first element) of this list.
E	removeFirst()	Removes and returns the first element from this list.
E	removeLast()	Removes and returns the last element from this list.
E	set(int index, E element)	Replaces the element at the specified position in this list with the specified element.
int	size()	Returns the number of elements in this list.
Object[]	toArray()	Returns an array containing all of the elements in this list in proper sequence (from first to last element).

- Reference: https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

Performance

Operations/Complexity	Big O (Worst-Case Complexity)
add(E e)	- General case: O(N) - Insertion at the beginning: O(1) - Insertion in the end: O(1) - Insertion into the middle: O(N)
add(int index, E element)	O(N)
addFirst(E e)	O(1)
addLast(E e)	O(1)
get(int index)	O(N)
getFirst()	O(1)
getLast()	O(1)
set(int index, E element)	O(N)
remove(int index)	O(N)
remove(Object o)	O(N)
remove()	O(1)
removeFirst()	O(1)
removeLast()	O(1)
size()	O(1)

6. The `Iterator`, `ListIterator`, and `Iterable` Interfaces

The `Iterator` Interface

Overview

- The List interface declares the method iterator, which returns an Iterator object that will iterate over the elements of that list.
- The requirement for the iterator method is actually in the Collection interface, which is the superinterface for the List interface. The `Collection` interface extends the `Iterable` interface, so all classes that implement the `List` interface (a subinterface of `Collection`) must provide an `Iterator` method.

- An Iterator does not refer to or point to a particular object at any given time. Rather, you should think of an Iterator as pointing between objects within a list.
- Think of an iterator as a moving place marker that keeps track of the current position in a particular linked list. The Iterator object for a list starts at the first element in the list. The programmer can use the Iterator object's `next` method to retrieve the next element. Each time it does a retrieval, the Iterator object advances to the next list element, where it waits until it is needed again. We can also ask the Iterator object to determine whether the list has more elements left to process (method `hasNext`). Iterator objects throw a `NoSuchElementException` if they are asked to retrieve the next element after all elements have been processed.
- You can use the Iterator remove method to remove elements from a list as you access them. You can remove only the element that was most recently accessed by next. Each call to remove must be preceded by a call to next to retrieve the next element.

Interface `java.util.Iterator<E>`

Modifier and Type	Method	Description
boolean	hasNext()	Returns true if the iteration has more elements.
E	next()	Returns the next element in the iteration.
default void	remove()	Removes from the underlying collection the last element returned by this iterator (optional operation).

Extra

> ### Efficient Access to List Elements by Iterator
>
> We can use the following loop to access the list elements in sequence, starting with the one at index 0.
>
> ```
> // Access each list element.
> for (int index = 0; index < aList.size(); index++) {
>     E nextElement = aList.get(index);
>     // Do something with the element at position index (nextElement)
>     . . .
> }
> ```
>
> The loop is executed aList.size() times; thus it is O(n). During each iteration, we call the method get to retrieve the element at position index. If we assume that the method get begins at the first list node (head), each call to method get must advance a local reference (nodeRef) to the node at position index using a loop such as:
>
> ```
> // Advance nodeRef to the element at position index.
> Node nodeRef = head;
> for (int j = 0; j < index; j++) {
>     nodeRef = nodeRef.next;
> }
> ```

> This loop (in method get) executes index times, so it is also O(n). Therefore, the performance of the nested loops used to process each element in a LinkedList is O(n^2) and is very inefficient. We would like to have an alternative way to access the elements in a linked list sequentially.

> ### Removal Using `Iterator.remove` versus `List.remove`
>
> You could also use method LinkedList.remove to remove elements from a list. However, it is more efficient to remove multiple elements from a list using Iterator.remove than it would be to use LinkedList.remove. The LinkedList.remove method removes only one element at a time, so you would need to start at the beginning of the list each time and advance down the list to each element that you wanted to remove (O(n2) process). With the Iterator.remove method, you can remove elements as they are accessed by the Iterator object without having to go back to the beginning of the list (O(n) process).

> ### The Enhanced `for` Loop
>
> The enhanced for loop creates an Iterator object and implicitly calls its hasNext and next methods. Other Iterator methods, such as remove, are not available.
>
> **FORM**:
>
> for (formalParameter : expression) { . . . }
>
> **EXAMPLE**:
>
> for (String nextStr : myList) { . . . }
> for (int nextInt : aList) { . . . }
>
> **MEANING**:
> During each repetition of the loop, the variable specified by formalParameter accesses the next element of expression, starting with the first element and ending with the last. The expression must be an array or a collection that implements the Iterable interface. The Collection interface extends the Iterable interface so that all classes that implement it are implementors of the Iterable interface (see next section).

The `ListIterator` Interface

Overview

- The Iterator has some limitations. It can traverse the List only in the forward direction. It also provides only a remove method, not an add method. Also, to start an Iterator somewhere other than at first List element, you must write your own loop to advance the Iterator to the desired starting position.
- The `next` method moves the iterator forward and returns the element that was jumped over. The `previous` method moves the iterator backward and also returns the element that was jumped over.

Interface `java.util.ListIterator<E>`

Modifier and Type	Method	Description
void	add(E e)	Inserts the specified element into the list (optional operation). Inserts object obj into the list just before the item that would be returned by the next call to method next and after the item that would have been returned by method previous. If the method previous is called after add, the newly inserted object will be returned
boolean	hasNext()	Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	hasPrevious()	Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E	next()	Returns the next element in the list and advances the cursor position.
int	nextIndex()	Returns the index of the element that would be returned by a subsequent call to next().
E	previous()	Returns the previous element in the list and moves the cursor position backwards.
int	previousIndex()	Returns the index of the element that would be returned by a subsequent call to previous().
void	remove()	Removes from the list the last element that was returned by next() or previous() (optional operation). Removes the last item returned from a call to next or previous. If a call to remove is not preceded by a call to next or previous, the IllegalStateException is thrown
void	set(E e)	Replaces the last element returned by next() or previous() with the specified element (optional). Replaces the last item returned from a call to next or previous with obj. If a call to set is not preceded by a call to next or previous, the IllegalStateException is thrown

Methods in `java.util.LinkedList<E>` that Return `ListIterator`

Modifier and Type	Method	Description
ListIterator<E>	listIterator()	Returns a ListIterator that begins just before the first list element.
ListIterator<E>	listIterator(int index)	Returns a ListIterator that begins just before the position index.

Comparison of `Iterator` and `ListIterator`

- Because the interface `ListIterator<E>` is a subinterface of `Iterator<E>`, classes that implement `ListIterator` must provide all of the capabilities of both.
- The `Iterator` interface requires fewer methods and can be used to iterate over more general data structures—that is, structures for which an index is not meaningful and ones for which traversing in only the **forward** direction is required.
- It is for this reason that the Iterator is required by the `Collection` interface (more general), whereas the `ListIterator` is required only by the `List` interface (more specialized).

The `Iterable` Interface

Overview

- This interface requires only that a class that implements it provides an `iterator` method. As mentioned above, the `Collection` interface extends the `Iterable` interface, so all classes that implement the `List` interface (a subinterface of `Collection`) must provide an iterator method.

Interface `java.lang.Iterable<T>`

Modifier and Type	Method and Description
default void	forEach(Consumer<? super T> action) Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
Iterator	iterator() Returns an iterator over elements of type T.
default Spliterator	spliterator() Creates a Spliterator over the elements described by this Iterable.

7. Double-Linked List (Linked List)

Overview

- A double-linked list object would consist of a separate object with data fields `head` (a reference to the first list Node), `tail` (a reference to the last list Node), and size (the number of Nodes). Because both ends of the list are directly accessible, now insertion at either end is O(1); insertion else- where is still O(n).

Implementation

- We can implement most of the `MyLinkedList` methods by delegation to the class `MyListIterator`, which will implement the `ListIterator` interface

Data Fields for Class `MyLinkedList<E>` (Double-Linked List)

Data Field	Attribute
private Node head	A reference to the first item in the list
private Node tail	A reference to the last item in the list
private int size	A count of the number of items in the list

```
package datastructures.list;

import java.util.*;
/** Class KWLinkedList implements a double-linked list and
 a ListIterator. */
public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>{

    /** A Node is the building block for a single-linked list. */
    private static class Node<E> {
        // Data Fields
        /** The reference to the data. */
        private E data;
        /** The reference to the next node. */
        private Node<E> next;
        /** The reference to the previous node. */
        private Node<E> prev;

        // Constructors
        /** Creates a new node with a null next field.
         @param dataItem The data stored
         */
        private Node(E dataItem) {
            data = dataItem;
            next = null;
```

```
        }

        /** Creates a new node that references another node.
         @param dataItem The data stored
         @param nodeRef The node referenced by new node
         */
        private Node(E dataItem, Node<E> nodeRef) {
            data = dataItem;
            next = nodeRef;
        }
    }

    /** Inner class to implement the ListIterator interface. */
    private class MyListIterator implements ListIterator<E> {
        /** A reference to the next item. */
        private Node<E> nextItem;
        /** A reference to the last item returned. */
        private Node<E> lastItemReturned;
        /** The index of the current item. */
        private int index = 0;

        /** Construct a KWListIter that will reference the ith item.
         @param i The index of the item to be referenced
         */
        public MyListIterator(int i) {
            // Validate i parameter.
            if (i < 0 || i > size) {
                throw new IndexOutOfBoundsException("Invalid index " + i);
            }
            lastItemReturned = null; // No item returned yet.
            // Special case of last item.
            if (i == size) {
                index = size;
                nextItem = null;
            } else { // Start at the beginning
                nextItem = head;
                for (index = 0; index < i; index++) {
                    nextItem = nextItem.next;
                }
            }
        }

        /** Indicate whether movement forward is defined.
         @return true if call to next will not throw an exception
         */
        public boolean hasNext() {
            return nextItem != null;
        }

        /** Move the iterator forward and return the next item.
         @return The next item in the list
         @throws NoSuchElementException if there is no such object
         */
        public E next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
```

```
            }
            lastItemReturned = nextItem;
            nextItem = nextItem.next;
            index++;
            return lastItemReturned.data;
        }

        /** Indicate whether movement backward is defined.
         @return true if call to previous will not throw an exception
         */
        public boolean hasPrevious() {
            return (nextItem == null && size != 0)
                    || nextItem.prev != null;
        }

        /** Move the iterator backward and return the previous item.
         @return The previous item in the list
         @throws NoSuchElementException if there is no such object
         */
        public E previous() {
            if (!hasPrevious()) {
                throw new NoSuchElementException();
            }
            if (nextItem == null) { // Iterator is past the last element
                nextItem = tail;
            } else {
                nextItem = nextItem.prev;
            }
            lastItemReturned = nextItem;
            index--;
            return lastItemReturned.data;
        }

        /**
         * Returns the index of the element that would be returned by a
subsequent call to next().
         *
         * @return the index of the element that would be returned by a
subsequent call to next(),
         *         or the size of the list if there is no next element
         */
        @Override
        public int nextIndex() {
            return (hasNext()) ? index : size;
        }

        /**
         * Returns the index of the element that would be returned by a
subsequent call to previous().
         *
         * @return the index of the element that would be returned by a
subsequent call to previous(),
         *         or -1 if there is no previous element
         */
        @Override
        public int previousIndex() {
```

```
            return (hasPrevious()) ? index - 1 : -1;
        }

        /**
         * Removes from the list the last element that was returned by next() or
previous().
         *
         * This method has not been verified!
         *
         * @throws IllegalStateException if no element has been previously
returned by next() or previous()
         */
        @Override
        public void remove() {
            if (lastItemReturned == null) {
                throw new IllegalStateException("No element to remove");
            }
            Node<E> lastNext = lastItemReturned.next;
            Node<E> lastPrev = lastItemReturned.prev;

            if (lastPrev == null) { // Removing first element
                head = lastNext;
            } else {
                lastPrev.next = lastNext;
                lastItemReturned.prev = null;
            }

            if (lastNext == null) { // Removing last element
                tail = lastPrev;
            } else {
                lastNext.prev = lastPrev;
                lastItemReturned.next = null;
            }

            if (nextItem == lastItemReturned) {
                nextItem = lastNext;
            } else {
                index--;
            }

            lastItemReturned.data = null;
            lastItemReturned = null;
            size--;
        }

        /**
         * Replaces the last element returned by next() or previous() with the
specified element.
         *
         * @param e the element with which to replace the last returned element
         * @throws IllegalStateException if no element has been previously
returned by next() or previous()
         */
        @Override
        public void set(E e) {
            if (lastItemReturned == null) {
```

```
                throw new IllegalStateException("No element to set");
            }
            lastItemReturned.data = e;
        }

        /** Add a new item between the item that will be returned
         by next and the item that will be returned by previous.
         If previous is called after add, the element added is
         returned.
         @param obj The item to be inserted
         */
        public void add(E obj) {
            if (head == null) { // Add to an empty list.
                head = new Node<>(obj);
                tail = head;
            } else if (nextItem == head) { // Insert at head
                // Create a new node.
                Node<E> newNode = new Node<>(obj);
                // Link it to the nextItem.
                newNode.next = nextItem;
                // Link nextItem to the new node.
                nextItem.prev = newNode;
                // The new node is now the head.
                head = newNode;
            }
            else if(nextItem == null){ // Insert at tail
                // Create a new node.
                Node<E> newNode = new Node<>(obj);
                // Link the tail to the new node.
                tail.next = newNode;
                // Link the new node to the tail.
                newNode.prev = tail;
                // The new node is the new tail.
                tail = newNode;
            }
            else { // Insert into the middle.
                // Create a new node.
                Node<E> newNode = new Node<>(obj);
                // Link it to nextItem.prev.
                newNode.prev = nextItem.prev;
                nextItem.prev.next = newNode;
                // Link it to the nextItem.
                newNode.next = nextItem;
                nextItem.prev = newNode;
            }
            // Increase size and index and set lastItemReturned.
            size++;
            index++;
            lastItemReturned = null;
        } // End of method add.


    }
    // Data Fields
    /** A reference to the head of the list. */
    private Node<E> head = null;
```

```
    /** A reference to the end of the list. */
    private Node<E> tail = null;
    /** The size of the list. */
    private int size = 0;

    /** Add an item at position index.
     @param index The position at which the object is to be
     inserted
     @param obj The object to be inserted
     @throws IndexOutOfBoundsException if the index is out
     of range (i < 0 || i > size())
     */
    public void add(int index, E obj) {
        listIterator(index).add(obj);
    }

    @Override
    public ListIterator<E> listIterator(int i) {
        return new MyListIterator(i);
    }

    /** Get the element at position index.
     @param index Position of item to be retrieved
     @return The item at index
     */
    public E get(int index) {
        return listIterator(index).next();
    }

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

Extra

> ### Inner Classes: Static and Nonstatic
>
> There are two inner classes in class LinkedList<E>: class Node and class MyListIterator. We declare Node<E> to be static because there is no need for its methods to access the data fields of its parent class (LinkedList<E>). We can't declare MyListIterator to be static because its methods access and modify the data fields of the LinkedList object that creates the MyListIterator object. An inner class that is not static contains an implicit reference to its parent object, just as it contains an implicit reference to itself. Because MyListIterator is not static and can reference data fields of its parent class LinkedList<E>, the type parameter is considered to be previously defined; therefore, it cannot appear as part of the class name.

> **PITFALL**
>
> **Defining MyListIterator as a Generic Inner Class**
> If you define class MyListIterator as
> private class MyListIterator...
> you will get an incompatible types syntax error when you attempt to reference data field head or tail (type Node<E>) inside class MyListIterator.

8. Circular Linked List

Overview

- **Circularly Linked Lists:** Linked lists are traditionally viewed as storing a sequence of items in a linear order, from first to last. However, there are many applications in which data can be more naturally viewed as having a cyclic order, with well-defined neighboring relationships, but no fixed beginning or end.
- Example Application: Round-Robin Scheduling
  - A process is given a short turn to execute, known as a time slice, but it is interrupted when the slice ends, even if its job is not yet complete. Each active process is given its own time slice, taking turns in a cyclic order.

Class `CircularlyLinkedList<E>`

Data Fields

Data Field	Attribute
private Node tail	A reference to the last item in the list
private int size	A count of the number of items in the list

Method Summary

Modifier and Type	Method	Description
void	addFirst(E e)	Inserts the specified element at the beginning of this list.
void	addLast(E e)	Appends the specified element to the end of this list.
E	getFirst()	Returns the first element in this list.
E	getLast()	Returns the last element in this list.
void	rotate()	Rotates the first element to the back of the list.
E	removeFirst()	Removes and returns the first element from this list.

Implementation

```
public class CircularLinkedList<E> {
    // Nested node class identical to that of the SinglyLinkedList class
    private static class Node<E> {
        private E element;
        private Node<E> next;

        public Node(E e, Node<E> n) {
            element = e;
            next = n;
        }

        public E getElement() {
            return element;
        }

        public Node<E> getNext() {
            return next;
        }

        public void setNext(Node<E> n) {
            next = n;
        }
    }

    private Node<E> tail = null; // We store tail (but not head)
    private int size = 0; // Number of nodes in the list

    public CircularLinkedList() { } // Constructs an initially empty list

    // Access methods
    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public E getFirst() {
        // Returns (but does not remove) the first element
        if (isEmpty()) return null;
        return tail.getNext().getElement(); // The head is *after* the tail
    }

    public E getLast() {
        // Returns (but does not remove) the last element
        if (isEmpty()) return null;
        return tail.getElement();
    }

    // Update methods
    public void rotate() {
        // Rotate the first element to the back of the list
```

```
        if (tail != null) // If empty, do nothing
            tail = tail.getNext(); // The old head becomes the new tail
    }

    public void addFirst(E e) {
        // Adds element e to the front of the list
        if (size == 0) {
            tail = new Node<>(e, null);
            tail.setNext(tail); // Link to itself circularly
        } else {
            Node<E> newest = new Node<>(e, tail.getNext());
            tail.setNext(newest);
        }
        size++;
    }

    public void addLast(E e) {
        // Adds element e to the end of the list
        addFirst(e); // Insert new element at front of list
        tail = tail.getNext(); // Now new element becomes the tail
    }

    public E removeFirst() {
        // Removes and returns the first element
        if (isEmpty()) return null; // Nothing to remove
        Node<E> head = tail.getNext();
        if (head == tail)
            tail = null; // Must be the only node left
        else
            tail.setNext(head.getNext()); // Removes ”head” from the list
        size--;
        return head.getElement();
    }
}
```

> TODO: The implementations of some methods are missing and should be fixed in the future.

Performance

Operations/Complexity	Big O
addFirst(E e)	O(1)
addLast(E e)	O(1)
getFirst()	O(1)
getLast()	O(1)
rotate()	O(1)

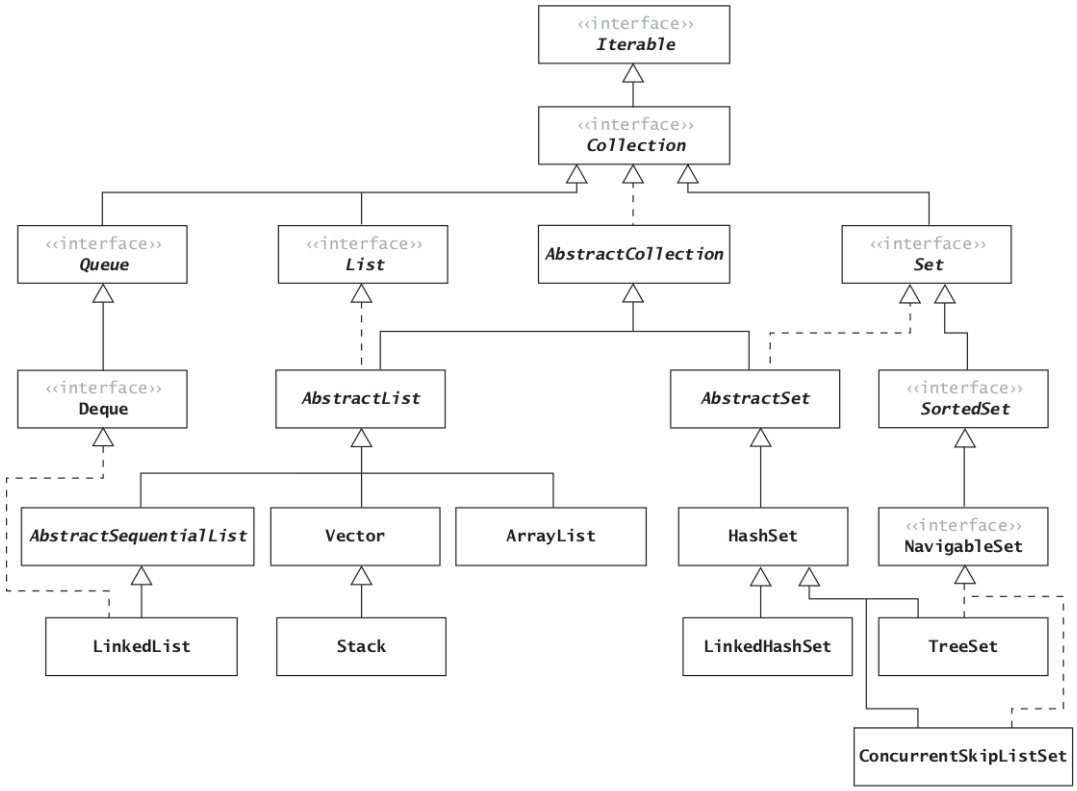
Operations/Complexity	Big O
removeFirst()	O(1)

9. The Collections Framework Design

The `Collection` Interface

- The `Collection` interface specifies a subset of the methods specified in the `List` interface.
- Specifically, the add(int, E), get(int), remove(int), set(int, E), and related methods (all of which have an int parameter that represents a position) are not in the `Collection` interface, but the add(E) and remove(Object) methods, which do not specify a position, are included.
- The `iterator` method is also included in the `Collection` interface. Thus, you can use an `Iterator` to access all of the items in a `Collection`, but the order in which they are retrieved is not necessarily related to the order in which they were inserted
- The Collection interface is part of the Collections Framework. This interface has three subinterfaces: the `List` interface, the `Queue` interface, and the `Set` interface. The Java API does not provide any direct implementation of the `Collection` interface. The interface is used to reference collections of data in the most general way.

FIGURE 2.37
The Collections Framework



Common Features of Collections

- A few features can be considered fundamental:
  - Collections grow as needed.
  - Collections hold references to objects.
  - Collections have at least two constructors: one to create an empty collection and one to make a copy of another collection.
- For collections implementing the List interface, the order of the elements is determined by the index of the elements. In the more general Collection, the order is not specified.

Interface `java.util.Collection<E>`

Modifier and Type	Method and Description
boolean	add(E e) Ensures that this collection contains the specified element (optional operation).
boolean	addAll(Collection<? extends E> c) Adds all of the elements in the specified collection to this collection (optional operation).
void	clear() Removes all of the elements from this collection (optional operation).
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.
boolean	isEmpty() Returns true if this collection contains no elements.
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 (optional operation).
boolean	removeAll(Collection<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).

Modifier and Type	Method and Description
boolean	retainAll(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	size() Returns the number of elements in this collection.
Object[]	toArray() Returns an array containing all of the elements in this collection.

The `AbstractCollection`, `AbstractList`, and `AbstractSequentialList` Classes

- If you look at the Java API documentation, you will see that the `Collection` and `List` interfaces specify a large number of methods. To help implement these interfaces, the Java API includes the `AbstractCollection` and `AbstractList` classes. You can think of these classes as a
  kit that can be used to build implementations of their corresponding interface. Most of the methods are provided, but you need to add a few to make it complete.
  - To implement the `Collection` interface completely, you need only extend the `AbstractCollection` class, provide an implementation of the add, size, and iterator methods, and supply an inner class to implement the Iterator interface.
  - To implement the `List` interface, you can extend the `AbstractList` class and provide an implementation of the add(int, E), get(int), remove(int),
    set(int, E), and size() methods.
    - Since we provided these methods in our `ArrayList`, we can make it a complete implementation of the `List` interface by changing the class declaration to

      ```
      public class ArrayList<E> extends AbstractList<E> implements List<E>
      ```

      Note that the `AbstractList` class implements the iterator and listIterator methods using the index associated with the elements.
    - Another way to implement the List interface is to extend the `AbstractSequentialList` class, implement the listIterator and size methods, and provide an inner class that implements the `ListIterator` interface. This was the approach we took in our `LinkedList`. Thus, by
      changing the class declaration to

      ```
       public class LinkedList<E> extends AbstractSequentialList<E>
      implements List<E>
      ```

      it becomes a complete implementation of the List interface.

    - Our `LinkedList` class included the add, get, remove, and set methods. These are provided by the `AbstractSequentialList`, so we could remove them from our `LinkedList` class and still have a complete List implementation.

The `List` and `RandomAccess` Interface

- The `RandomAccess` interface is applied only to those implementations in which indexed operations are efficient (e.g., ArrayList). An algorithm can then test to see if a parameter of type List is also of type `RandomAccess` and, if not, copy its contents into an `ArrayList` temporarily so that the indexed operations can proceed more efficiently. After the indexed operations are completed, the contents of the `ArrayList` are copied back to the original.

1. Stack

Overview

- A stack is a data structure with the property that only the top element of the stack is accessible. In a stack, the top element is the data value that was most recently stored in the stack.
- Sometimes this storage policy is known as last-in, first-out, or LIFO.

Stack ADT

Modifier and Type	Method	Description
boolean	empty()	Tests if this stack is empty.
E	peek()	Looks at the object at the top of this stack without removing it from the stack.
E	pop()	Removes the object at the top of this stack and returns that object as the value of this function.
E	push(E item)	Pushes an item onto the top of this stack.

Class `java.util.Stack<E>`

- The `java.util.Stack` class is part of the original Java API but is not recommended for new applications. Instead, the Java designers recommend that we use the `java.util.Deque` interface and the `java.util.ArrayDeque` class to provide the methods listed above. The `Deque` interface specifies the methods in our interface `StackInt` (see Table) and also those needed for a queue. We will discuss the `Deque` interface and class `ArrayDeque` later.

Our Interface `Stack<E>`

```
/**
 * A collection of objects that are inserted and removed according to the last-in
 * first-out principle. Although similar in purpose, this interface differs from
 * java.util.Stack.
 */
public interface Stack<E> {

    /**
     * Returns the number of elements in the stack.
     *
     * @return number of elements in the stack
```

```
     */
    int size();

    /**
     * Tests whether the stack is empty.
     *
     * @return true if the stack is empty, false otherwise
     */
    boolean empty();

    /**
     * Inserts an element at the top of the stack.
     *
     * @param e the element to be inserted
     */
    void push(E e);

    /**
     * Returns, but does not remove, the element at the top of the stack.
     *
     * @return top element in the stack (or null if empty)
     */
    E top();

    /**
     * Removes and returns the top element from the stack.
     *
     * @return element removed (or null if empty)
     */
    E pop();
}
```

Performance

Operations	Big O (Worst-case Complexity)
empty()	O(1)
peek()	O(1)
pop()	O(1)
push(E item)	O(1)

Implementation

- We are going to cover two ways of implementing a Stack:
  - `Array (or ArrayList can also be adapted)`
  - `LinkedList`

Array-Based Stack Implementation

- The following class implements generic `Stack<E>` interface.

Implementation

```
public class ArrayStack<E> implements Stack<E> {

    public static final int INITIAL_CAPACITY = 10; // Default array capacity
    private E[] data; // Generic array used for storage
    private int top = -1; // Index of the top element in the stack
    private int capacity;

    public ArrayStack() {
        this(INITIAL_CAPACITY); // Constructs stack with default capacity
    }

    public ArrayStack(int capacity) {
        // Constructs stack with given capacity
        this.capacity = capacity;
        data = (E[]) new Object[capacity]; // Safe cast; compiler may give
warning
    }

    public int size() {
        return (top + 1);
    }

    public boolean empty() {
        return (top == -1);
    }

    public void push(E e) throws IllegalStateException {
        if (size() == data.length) {
            reallocate();
        }
        data[++top] = e; // Increment t before storing the new item
    }

    public E top() {
        if (empty()) return null;
        return data[top];
    }

    public E pop() {
        if (empty()) return null;
        E answer = data[top];
        data[top] = null; // Dereference to help garbage collection
        top--;
        return answer;
    }

    private void reallocate(){
```

```
            capacity = 2 * capacity;
            data = Arrays.copyOf(data, capacity);
        }
    }
```

Drawbacks of Array-Based Implementation

- Fixed-capacity array
    - If the application needs much less space than the reserved capacity, memory is **wasted**.
        - Performance of a stack realized by an array. The space usage is $O(N)$, where N is the size of the array, determined at the time the stack is instantiated, and independent from the number n $\leq$ N of elements that are actually in the stack.
    - When the stack has reached the capacity, it will refuse storing a new element throwing **IllegalStateException**. This problem can be fixed by using `ArrayList` as adapter instead of implementation from scratch using `Array`.

Singly-Linked List-Based Stack

Implementation

```
import java.util.LinkedList;

public class LinkedStack<E> implements Stack<E> {

    private LinkedList<E> list = new LinkedList<>(); // An empty list

    public LinkedStack() { }
    // New stack relies on the initially empty list

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

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

    public void push(E element) {
        list.addFirst(element);
    }

    public E top() {
        return list.getFirst();
    }

    public E pop() {
        return list.removeFirst();
    }
}
```

Extra

> ### The Adapter Pattern
>
> The **adapter** design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface. One general way to apply the adapter pattern is to define a new class in such a way that it contains an instance of the existing class as a hidden field, and then to implement each method of the new class using methods of this hidden instance variable. By applying the adapter pattern in this way, we have created a new class that performs some of the same functions as an existing class, but repackaged in a more convenient way.

Comparisons of Stack Implementations

- As we discussed before, in array-based implementation If the application needs much less space than the reserved capacity, memory is **wasted**.
- Whereas a linked list based implementation has the advantage of using exactly as much storage as needed for the stack. However, also note that since a linked-list node stores 2 data field references for the previous and next node references and 1 data field for stored data element, a linked-list based implementation of the full size is 3 times more expensive in terms of memory than a full array-based stack.

2. Queue

Overview

- Another fundamental data structure is the queue. It is a close “cousin” of the stack, but a queue is a collection of objects that are inserted and removed according to the first-in, first-out (FIFO) principle.

Interface `java.util.Queue<E>`

Modifier and Type	Method	Description
`boolean`	`add(E e)`	Inserts the specified element into this queue if space is available, returning true upon success. Throws IllegalStateException if no space is available.
`boolean`	`offer(E e)`	Inserts the specified element into this queue if space is available, returning true upon success.
`E`	`element()`	Retrieves, but does not remove, the head of this queue.
`E`	`peek()`	Retrieves, but does not remove, the head of this queue. Returns null if the queue is empty.

Modifier and Type	Method	Description
`E`	`poll()`	Retrieves and removes the head of this queue. Returns null if the queue is empty.
`E`	`remove()`	Retrieves and removes the head of this queue. Throws NoSuchElementException if the queue is empty.

Summary of Queue methods

	Throws exception	*Returns special value*
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

Implementation

Circular Array-Based Queue

```java
package datastructures.queue;

import java.util.AbstractQueue;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Queue;

/** Implements the Queue interface using a circular array. */
public class ArrayQueue<E> extends AbstractQueue<E> implements Queue<E> {
    // Data Fields
    /** Index of the front of the queue. */
    private int front;
    /** Index of the rear of the queue. */
    private int rear;
    /** Current size of the queue. */
    private int size;
    /** Current capacity of the queue. */
    private int capacity;
    /** Default capacity of the queue. */
    private static final int DEFAULT_CAPACITY = 10;
    /** Array to hold the data. */
    private E[] data;

    // Constructors
    /** Construct a queue with the default initial capacity. */
    public ArrayQueue() {
        this(DEFAULT_CAPACITY);
    }

    @SuppressWarnings("unchecked")
    /** Construct a queue with the specified initial capacity.
```

```
    @param initCapacity The initial capacity
    */
    public ArrayQueue(int initCapacity) {
        capacity = initCapacity;
        data = (E[]) new Object[capacity];
        front = 0;
        rear = capacity-1;
        size = 0;
    }

    // Public Methods
    /** Inserts an item at the rear of the queue.
     @post item is added to the rear of the queue.
     @param item The element to add
     @return true (always successful)
     */
    @Override
    public boolean offer(E item) {
        if (size == capacity) {
            reallocate();
        }
        size++;
        rear = (rear + 1) % capacity; data[rear] = item;
        return true;
    }

    /** Returns the item at the front of the queue without removing it.
     @return The item at the front of the queue if successful; return null if
     the queue is empty
     */
    @Override
    public E peek() {
        if (size == 0)
            return null;
        else
            return data[front];
    }

    /** Removes the entry at the front of the queue and returns it if the queue
is
     not empty.
     @post front references item that was second in the queue.
     @return The item removed if successful or null if not
     */
    @Override
    public E poll() {
        if (size == 0) {
            return null;
        }
        E result = data[front];
        front = (front + 1) % capacity;
        size--;
        return result;
    }

    @Override
```

```
    public boolean add(E e) {
        return offer(e);
    }

    @Override
    public E element() {
        return peek();
    }

    @Override
    public E remove() {
        return poll();
    }

    @Override
    public Iterator<E> iterator() {
        return new ArrayQueueIterator();
    }

    @Override
    public int size() {
        return size;
    }

    // Private Methods
    /** Double the capacity and reallocate the data.
     @pre The array is filled to capacity.
     @post The capacity is doubled and the first half of the expanded array is
     filled with data.
     */
    @SuppressWarnings("unchecked")
    private void reallocate() {
        int newCapacity = 2 * capacity;
        E[] newData = (E[]) new Object[newCapacity];
        int j = front;
        for (int i = 0; i < size; i++) {
            newData[i] = data[j];
            j = (j + 1) % capacity;
        }
        front = 0;
        rear = size-1;
        capacity = newCapacity;
        data = newData;
    }

    /** Inner class to implement the Iterator<E> interface. */
    private class ArrayQueueIterator implements Iterator<E> {
        // Data Fields
        // Index of next element
        private int index;
        // Count of elements accessed so far
        private int count = 0;

        // Methods

        // Constructor
```

```
        /** Initializes the Iter object to reference the first queue element. */
        public ArrayQueueIterator() {
            index = front;
        }
        /** Returns true if there are more elements in the queue to access. */
        @Override
        public boolean hasNext() {
            return count < size;
        }

        /** Returns the next element in the queue.
         @pre index references the next element to access.
         @post index and count are incremented.
         @return The element with subscript index
         */
        @Override
        public E next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            E returnValue = data[index];
            index = (index + 1) % capacity;
            count++;
            return returnValue;
        }

        /** Remove the item accessed by the Iter object – not implemented. */
        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

Increasing Queue Capacity

- When the capacity is reached, we double the capacity and copy the array into the new one, as was done for the ArrayList. However, we can't simply use the reallocate method we developed for the ArrayList because of the circular nature of the array. We can't copy over elements from the original array to the first half of the expanded array, maintaining their position.
- We must first copy the elements from position front through the end of the original array to the beginning of the expanded array; then copy the elements from the beginning of the original array through rear to follow those in the expanded array.

Single-Linked List-Based Queue

```
package datastructures.queue;

import java.util.*;

/** Implements the Queue interface using a single-linked list. */
public class ListQueue<E> extends AbstractQueue<E> implements Queue<E> {

    /** A Node is the building block for a single-linked list. */
    private static class Node<E> {
        // Data Fields
        /** The reference to the data. */
        private E data;
        /** The reference to the next node. */
        private Node<E> next;

        // Constructors
        /** Creates a new node with a null next field.
         @param dataItem The data stored
         */
        private Node(E dataItem) {
            data = dataItem;
            next = null;
        }

        /** Creates a new node that references another node.
         @param dataItem The data stored
         @param nodeRef The node referenced by new node
         */
        private Node(E dataItem, Node<E> nodeRef) {
            data = dataItem;
            next = nodeRef;
        }
    }

    // Data Fields
    /** Reference to front of queue. */
    private Node<E> front;
    /** Reference to rear of queue. */
    private Node<E> rear;
    /** Size of queue. */
    private int size;

    // Insert inner class Node<E> for single-linked list here.
    // (See Listing 2.1.)

    // Methods

    /** Insert an item at the rear of the queue.
     @post item is added to the rear of the queue.
     @param item The element to add
     @return true (always successful)
     */
    @Override
```

```
    public boolean offer(E item) {
        // Check for empty queue.
        if (front == null) {
            rear = new Node<>(item);
            front = rear;
        } else {
        // Allocate a new node at end, store item in it, and
        // link it to old end of queue.
            rear.next = new Node<>(item);
            rear = rear.next;
        }
        size++;
        return true;
    }

    /** Remove the entry at the front of the queue and return it
         if the queue is not empty.
        @post front references item that was second in the queue.
        @return The item removed if successful, or null if not
     */

    @Override
    public E poll() {
        E item = peek();
        // Retrieve item at front.
        if (item == null)
            return null;
        // Remove item at front.
        front = front.next; size--;
        return item;
        // Return data at front of queue.
    }

    /** Return the item at the front of the queue without removing it.
     @return The item at the front of the queue if successful;
     return null if the queue is empty
     */
    @Override
    public E peek() {
        if (size == 0)
            return null;
        else
            return front.data;
    }

    @Override
    public boolean add(E e) {
        return offer(e);
    }

    @Override
    public E element() {
        return peek();
    }

    @Override
```

```
    public E remove() {
        return poll();
    }

    // Insert class Iter.

    @Override
    public Iterator<E> iterator() {
        return null;
    }

    @Override
    public int size() {
        return size;
    }
}
```

Double-Linked List-Based Queue

- Use `java.util.LinkedList` class as adapter class.

```
public class DoubleLinkedQueue<E> extends AbstractQueue<E> implements Queue<E> {

    private LinkedList<E> list; // an empty list

    public DoubleLinkedQueue( ) {
        list = new LinkedList<>();
    }

    @Override
    public boolean add(E e) {
        return list.add(e); // Appends e to the end of the list
    }

    @Override
    public boolean offer(E e) {
        return list.add(e); // Appends e to the end of the list
    }

    @Override
    public E remove() {
        return list.remove(); // Removes the element in the beginning of the list
    }

    @Override
    public E poll() {
        return list.remove();
    }

    @Override
    public E element() {
```

```
        return list.getFirst();
    }

    @Override
    public E peek() {
        return list.getFirst();
    }

    @Override
    public Iterator<E> iterator() {
        return list.iterator();
    }

    @Override
    public int size() {
        return list.size();
    }

}
```

Comparing the Three Implementations

- As mentioned earlier, all three implementations of the Queue interface are comparable in terms of computation time. All operations are O(1) regardless of the implementation. Although reallocating an array is an O(n) operation, it is amortized over n items, so the cost per item is O(1).
- In terms of storage requirements, both linked-list implementations require more storage because of the extra space required for links. To perform an analysis of the storage requirements, you need to know that Java stores a reference to the data for a queue element in each node in addition to the links. Therefore, each node for a single-linked list would store a total of two references (one for the data and one for the link), a node for a double-linked list would store a total of three references, and a node for a circular array would store just one reference. Therefore, a double-linked list would require 1.5 times the storage required for a single-linked list with the same number of elements. A circular array that is filled to capacity would require half the storage of a single-linked list to store the same number of elements. However, if the array were just reallocated, half the array would be empty, so it would require the same storage as a single-linked list.

3. Deque

Overview

- The name deque (pronounced "deck") is short for double-ended queue, which means that it is a data structure that allows insertions and removals from both ends (front and rear).

Interface `java.util.Deque<E>`

Modifier and Type	Method	Description
`boolean`	`add(E e)`	Inserts the specified element into the queue represented by this deque (at the tail) if space is available, throws IllegalStateException if no space is available.
`void`	`addFirst(E e)`	Inserts the specified element at the front of this deque, throws IllegalStateException if no space is available.
`void`	`addLast(E e)`	Inserts the specified element at the end of this deque, throws IllegalStateException if no space is available.
`boolean`	`offer(E e)`	Inserts the specified element into the queue represented by this deque (at the tail) if space is available, returns true upon success, false if no space is available.
`boolean`	`offerFirst(E e)`	Inserts the specified element at the front of this deque unless it would violate capacity restrictions.
`boolean`	`offerLast(E e)`	Inserts the specified element at the end of this deque unless it would violate capacity restrictions.
`E`	`poll()`	Retrieves and removes the head of the queue represented by this deque, returns null if this deque is empty.
`E`	`pollFirst()`	Retrieves and removes the first element of this deque, returns null if this deque is empty.
`E`	`pollLast()`	Retrieves and removes the last element of this deque, returns null if this deque is empty.
`E`	`remove()`	Retrieves and removes the head of the queue represented by this deque.

`Modifier and Type`	`Method`	**Description**
`E`	`removeFirst()`	Retrieves and removes the first element of this deque.
`E`	`removeLast()`	Retrieves and removes the last element of this deque.
`E`	`peek()`	Retrieves, but does not remove, the head of the queue represented by this deque, returns null if this deque is empty.
`E`	`peekFirst()`	Retrieves, but does not remove, the first element of this deque, returns null if this deque is empty.
`E`	`peekLast()`	Retrieves, but does not remove, the last element of this deque, returns null if this deque is empty.
`E`	`element()`	Retrieves, but does not remove, the head of the queue represented by this deque.
`E`	`getFirst()`	Retrieves, but does not remove, the first element of this deque.
`E`	`getLast()`	Retrieves, but does not remove, the last element of this deque.
`boolean`	`removeFirstOccurrence(Object o)`	Removes the first occurrence of the specified element from this deque.
`boolean`	`removeLastOccurrence(Object o)`	Removes the last occurrence of the specified element from this deque.
`int`	`size()`	Returns the number of elements in this deque.
`Iterator<E>`	`iterator()`	Returns an iterator over the elements in this deque in proper sequence.
`Iterator<E>`	`descendingIterator()`	Returns an iterator over the elements in this deque in reverse sequential order.
`boolean`	`contains(Object o)`	Returns true if this deque contains the specified element.

Modifier and Type	Method	Description
`void`	`push(E e)`	Pushes an element onto the stack represented by this deque (at the head) if space is available, throws IllegalStateException if no space is available.
`E`	`pop()`	Pops an element from the stack represented by this deque.
`boolean`	`remove(Object o)`	Removes the first occurrence of the specified element from this deque.

Summary of Deque methods

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Implementation

- The Java Collections Framework provides four implementations of the Deque interface, including **ArrayDeque** and **LinkedList**.
    - ArrayDeque utilizes a **resizable circular array** like our class **ArrayQueue** and is the **recommended** implementation because, unlike **LinkedList, it does not support indexed operations.**

Using a Deque as a Queue

Comparison of Queue and Deque methods

Queue Method	Equivalent Deque Method
`add(e)`	`addLast(e)`
`offer(e)`	`offerLast(e)`
`remove()`	`removeFirst()`
`poll()`	`pollFirst()`
`element()`	`getFirst()`
`peek()`	`peekFirst()`

Using a Deque as a Stack

Comparison of Stack and Deque methods

Stack Method	Equivalent `Deque` Method
`push(e)`	`addFirst(e)`
`pop()`	`removeFirst()`
`peek()`	`peekFirst()`

5. Trees

Tree Terminology

- `root`: The node at the top of a tree. It has no parents.
- `edge (branch)`: The link from a node to its successor.
- `children`: The successors of a node.
- `parent`: The predecessor of a node.
- `siblings`: The nodes that have the same parent.
- `leaf`: A node that has no children.
- `external nodes`: Leaves are also known as external nodes.
- `internal nodes`: Non-leaf nodes are also known as internal nodes.
- `subtree of a node`: A tree whose root is a child of that node.
- `depth`: The depth of a node is the length of the path from the root to the node.
- `level`: The set of all nodes at given depth is called the `level` of the tree. The `root` node is at level `zero`.
    - `level` and `depth` describe the same concept but level is a tree-oriented term while depth is a node-oriented term (e.g. level of tree vs depth of node).
- `height`: The height of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of `zero`.
- `height of the tree`: The height of root node.
- `depth of the tree`: The maximum depth among all the nodes in the tree.
    - For a given tree, `height of the tree` $=$ `depth of the tree`.
- `size`: The size of a node is the number of descendants it has including itself.
- `skew tree`: If every node in a tree has only one child (except leaf nodes).
    - `left skew tree`: If every node has only left child.
    - `right skew tree:` If every node has only right child.
- `strict binary tree`: All nodes have either 2 children or 0 children (leaf nodes).
- `full binary tree`: All nodes have exactly 2 children except the leaf nodes in the final level, level n, for a tree of height n. In other words, a binary tree is full binary tree if each node has exactly two children and all leaf nodes are at the same level.
- `complete binary tree`: A complete binary tree is a full binary tree through level n -1 with some extra leaf nodes at level n and all of the nodes at level n are towards the left of the tree.

Binary Trees

- A set of nodes `T` is a `binary tree` if either of the following is true:
    - T is empty

    - Else, its root node has two subtrees, T_L and T_R, such that they are binary trees.
- A tree is called `binary tree` if each node has at most two children.
    - Observe that a binary tree has a recursive definition.

Binary Search Tree

- A set of nodes in T is a `binary search tree` if either of the following is true
    - T is empty
    - Else, its root node has two subtrees, T_L and T_R, such that they are binary search trees and the value in the root node of T is greater than all values in T_L and is less than all values in T_R.
- Observe that a binary search tree has a recursive definition as well.

Searching a Binary Tree

```
if the tree is empty
    Return null (target is not found).
else if the target matches the root node's data
    Return the data stored at the root node.
else if the target is less than the root node's data
    Return the result of searching the left subtree of the root.
else
    Return the result of searching the right subtree of the root.
```

- Just as with a binary search of an array, each probe into the binary search tree has the potential of eliminating half the elements in the tree. If the binary search tree is relatively balanced (i.e., the depths of the leaves are approximately the same), searching a binary search tree is an $O(logn)$ process, just like a binary search of an ordered array.

Tree Traversals

Preorder Traversal

```
if the tree is empty
    Return.
else
    Visit the root.
    Preorder traverse the left subtree
    Preorder traverse the right subtree
```

Inorder Traversal

```
if the tree is empty
    Return.
else
    Inorder traverse the left subtree
    Visit the root.
    Inorder traverse the right subtree
```

Postorder Traversal

```
if the tree is empty
    Return.
else
    Postorder traverse the left subtree
    Postorder traverse the right subtree
    Visit the root.
```

Implementing a `Binary Tree` Class

Class `BinaryTree<E>`

Data Field/Attribute	Description
`protected Node<E> root`	Reference to the root of the tree

Constructor/Behavior	Description
`public BinaryTree()`	Constructs an empty binary tree
`protected BinaryTree(Node<E> root)`	Constructs a binary tree with the given node as the root
`public BinaryTree(E data, BinaryTree<E> leftTree, BinaryTree<E> rightTree)`	Constructs a binary tree with the given data at the root and the two given subtrees

Method/Behavior	Description
`public BinaryTree<E> getLeftSubtree()`	Returns the left subtree
`public BinaryTree<E> getRightSubtree()`	Returns the right subtree
`public E getData()`	Returns the data in the root
`public boolean isLeaf()`	Returns true if this tree is a leaf, false otherwise

Method/Behavior	Description
`public String toString()`	Returns a String representation of the tree
`public void preOrderTraversal()`	Prints the output of preorder traversal
`public void inOrderTraversal()`	Prints the output of inorder traversal
`public void postOrderTraversal()`	Prints the output of postorder traversal

```
package datastructures.tree;

import java.io.Serializable;
import java.util.Comparator;
import java.util.Scanner;

public class BinaryTree<E> implements Serializable{

    /** Class to encapsulate a tree node. */
    protected static class Node<E> implements Serializable {
        // Data Fields
        /** The information stored in this node. */
        protected E data;
        /** Reference to the left child. */
        protected Node<E> left;
        /** Reference to the right child. */
        protected Node<E> right;

        // Constructors
        /** Construct a node with given data and no children.
         @param data The data to store in this node
         */
        public Node(E data) {
            this.data = data;
            left = null;
            right = null;
        }
        // Methods
        /** Return a string representation of the node.
         @return A string representation of the data fields
         */
        public String toString () {
            return data.toString();
        }
    }

    // Data Field
    /** The root of the binary tree */
    protected Node<E> root;

    // Data Fields
    /** Return value from the public add method. */
    protected boolean addReturn;
    /** Return value from the public delete method. */
    protected E deleteReturn;
```

```
    public BinaryTree() {
        root = null;
    }

    protected BinaryTree(Node<E> root) {
        this.root = root;
    }

    /** Constructs a new binary tree with data in its root leftTree
     as its left subtree and rightTree as its right subtree.
     */
    public BinaryTree(E data, BinaryTree<E> leftTree,
                      BinaryTree<E> rightTree) {
        root = new Node<>(data);
        if (leftTree != null) {
            root.left = leftTree.root;
        } else {
            root.left = null;
        }
        if (rightTree != null) {
            root.right = rightTree.root;
        } else {
            root.right = null;
        }
    }

    /** Return the left subtree.
     @return The left subtree or null if either the root or
     the left subtree is null
     */
    public BinaryTree<E> getLeftSubtree() {
        if (root != null && root.left != null) {
            return new BinaryTree<>(root.left);
        } else {
            return null;
        }
    }

    /** Return the right subtree.
     @return The right subtree or null if either the root or
     the right subtree is null
     */
    public BinaryTree<E> getRightSubtree() {
        if (root != null && root.right != null) {
            return new BinaryTree<>(root.right);
        } else {
            return null;
        }
    }

    /** Determine whether this tree is a leaf.
     @return true if the root has no children
     */
    public boolean isLeaf() {
        return (root.left == null && root.right == null);
```

```
    }

    public String toString() {
        StringBuilder sb = new StringBuilder();
        toString(root, 1, sb);
        return sb.toString();
    }

    /** Converts a sub-tree to a string.
     Performs a preorder traversal.
     @param node The local root
     @param depth The depth
     @param sb The StringBuilder to save the output
     */
    private void toString(Node<E> node, int depth,
                          StringBuilder sb) {
        for (int i = 1; i < depth; i++) {
            sb.append(" ");
        }
        if (node == null) {
            sb.append("null\n");
        } else {
            sb.append(node.toString());
            sb.append("\n");
            toString(node.left, depth + 1, sb);
            toString(node.right, depth + 1, sb);
        }
    }

    public void preOrderTraversal(){
        preOrderTraversal(root);
    }

    private void preOrderTraversal(Node<E> node){
        if(node != null){
            System.out.println(node.toString());
            preOrderTraversal(node.left);
            preOrderTraversal(node.right);
        }
    }

    public void inOrderTraversal(){
        inOrderTraversal(root);
    }

    private void inOrderTraversal(Node<E> node){
        if(node != null){
            inOrderTraversal(node.left);
            System.out.println(node.toString());
            inOrderTraversal(node.right);
        }
    }

    public void postOrderTraversal(){
        postOrderTraversal(root);
    }
```

```
    private void postOrderTraversal(Node<E> node){
        if(node != null){
            postOrderTraversal(node.left);
            postOrderTraversal(node.right);
            System.out.println(node.toString());
        }
    }
}
```

Implementing a `BinarySearchTree` Class

Interface `SearchTree<E>`

Method	Description
`boolean add(E item)`	Inserts `item` where it belongs in the tree. Returns true if `item` is inserted; false if it isn't (already in tree)
`boolean contains(E target)`	Returns true if `target` is found in the tree
`E search(E target)`	Returns a reference to the data in the node that is equal to `target`. If no such node is found, returns `null`
`E delete(E target)`	Removes `target` (if found) from the tree and returns it; otherwise, returns `null`
`boolean remove(E target)`	Removes `target` (if found) from the tree and returns true; otherwise, returns false

Operations

Search

```
if the tree is empty
    Return null (target is not found).
else if the target matches the root node's data
    Return the data stored at the root node.
else if the target is less than the root node's data
    Return the result of searching the left subtree of the root.
else
    Return the result of searching the right subtree of the root.
```

Insert

```
if the root is null
    Replace empty tree with a new tree with the item at the root and return true.
else if the item is equal to root.data
    The item is already in the tree; return false.
else if the item is less than root.data
    Recursively insert the item in the left subtree.
else
    Recursively insert the item in the right subtree.
```

Remove

```
if the root is null
    The item is not in tree – return null.
Compare the item to the data at the local root.
if the item is less than the data at the local root
    Return the result of deleting from the left subtree.
else if the item is greater than the local root
    Return the result of deleting from the right subtree.
else # The item is in the local root
    Store the data in the local root in deleteReturn.
    if the local root has no children
        Set the parent of the local root to reference null.
    else if the local root has one child
        Set the parent of the local root to reference that child.
    else # Find the inorder predecessor
        if the left child has no right child it is the inorder predecessor
            Set the parent of the local root to reference the left child.
        else
            Find the rightmost node in the right subtree of the left child.
            Copy its data into the local root's data and remove it by setting its
parent to reference its left child.
```

Implementation

```
package datastructures.tree;

import java.util.NoSuchElementException;

public class BinarySearchTree<E extends Comparable<E>>
        extends BinaryTree<E> implements SearchTree<E> {
    // Data Fields
    /** Return value from the public add method. */
    protected boolean addReturn;
    /** Return value from the public delete method. */
    protected E deleteReturn;

    /** Starter method find.
     pre: The target object must implement
     the Comparable interface.
     @param target The Comparable object being sought
```

```
     @return The object, if found, otherwise null
     */
    public E search(E target) {
        return search(root, target);
    }

    /** Recursive find method.
     @param localRoot The local subtree's root
     @param target The object being sought
     @return The object, if found, otherwise null
     */
    private E search(Node<E> localRoot, E target) {
        if (localRoot == null)
            return null;
        // Compare the target with the data field at the root.
        int compResult = target.compareTo(localRoot.data);
        if (compResult == 0)
            return localRoot.data;
        else if (compResult < 0)
            return search(localRoot.left, target);
        else
            return search(localRoot.right, target);
    }

    /** Starter method add.
     pre: The object to insert must implement the
     Comparable interface.
     @param item The object being inserted
     @return true if the object is inserted, false
     if the object already exists in the tree
     */
    public boolean add(E item) {
        root = add(root, item);
        return addReturn;
    }

    /** Recursive add method.
     post: The data field addReturn is set true if the item is added to
     the tree, false if the item is already in the tree.
     @param localRoot The local root of the subtree
     @param item The object to be inserted
     @return The new local root that now contains the
     inserted item
     */
    private Node<E> add(Node<E> localRoot, E item) {
        if (localRoot == null) {
            // item is not in the tree — insert it.
            addReturn = true;
            return new Node<>(item);
        } else if (item.compareTo(localRoot.data) == 0) {
            // item is equal to localRoot.data
            addReturn = false;
            return localRoot;
        } else if (item.compareTo(localRoot.data) < 0) {
            // item is less than localRoot.data
            localRoot.left = add(localRoot.left, item);
```

```
            return localRoot;
        } else {
            // item is greater than localRoot.data
            localRoot.right = add(localRoot.right, item);
            return localRoot;
        }
    }

    /** Starter method delete.
     post: The object is not in the tree.
     @param target The object to be deleted
     @return The object deleted from the tree
     or null if the object was not in the tree
     @throws ClassCastException if target does not implement
     Comparable
     */
    public E delete(E target) {
        root = delete(root, target);
        return deleteReturn;
    }

    /** Recursive delete method.
     post: The item is not in the tree;
     deleteReturn is equal to the deleted item
     as it was stored in the tree or null
     if the item was not found.
     @param localRoot The root of the current subtree
     @param item The item to be deleted
     @return The modified local root that does not contain
     the item
     */
    private Node<E> delete(Node<E> localRoot, E item) {
        if (localRoot == null) {
            // item is not in the tree.
            deleteReturn = null;
            return localRoot;
        }
        // Search for item to delete.
        int compResult = item.compareTo(localRoot.data);
        if (compResult < 0) {
            // item is smaller than localRoot.data.
            localRoot.left = delete(localRoot.left, item);
        } else if (compResult > 0) {
            // item is larger than localRoot.data.
            localRoot.right = delete(localRoot.right, item);
        } else {
            // item is at local root.
            deleteReturn = localRoot.data;
            if (localRoot.left == null) {
                // If there is no left child, return right child
                // which can also be null.
                return localRoot.right;
            } else if (localRoot.right == null) {
                // If there is no right child, return left child.
                return localRoot.left;
            } else {
```

```
                // Node being deleted has 2 children, replace the data with
inorder predecessor.
                localRoot.data = getMin(localRoot.right);
                localRoot.right = delete(localRoot.right, localRoot.data);
            }
        }
        return localRoot;
    }

    private E getMin(){
        if(root == null){
            throw new NoSuchElementException();
        }
        return getMin(root);
    }

    private E getMin(Node<E> localRoot){
        if(localRoot.left == null){
            return localRoot.data;
        }
        else{
            return getMin(localRoot.left);
        }
    }

    @Override
    public boolean remove(E target) {
        root = delete(root, target);
        return deleteReturn != null;
    }

    @Override
    public boolean contains(E target) {
        return search(target) != null;
    }

}
```

1. Heaps

Overview

- At each level of a heap, the value in a node is less than all values in its two subtrees.
  - Implies that minimum element is always the root (a "min-heap").
  - **Variation**: "max-heap" stores largest element at root, reverses ordering
- More formally, a heap is a complete binary tree with the following properties:
  - The value in the root is the smallest item in the tree.
  - Every subtree is a heap.

Operations

Insertion

```
Insert the new item in the next position at the bottom of the heap.
while new item is not at the root and new item is smaller than its parent
    Swap the new item with its parent, moving the new item up the heap.
```

Removal

```
Remove the item in the root node by replacing it with the last item in the heap
(LIH).
while item LIH has children, and item LIH is larger than either of its children
    Swap item LIH with its smaller child, moving LIH down the heap.
```

Implementation

- Because a heap is a complete binary tree, we can implement it efficiently using an `array` (or ArrayList) instead of a linked data structure.
- Definitions:
  - Root index = 0
  - For a node at position p
    - left child = 2p + 1
    - right child = 2p +2
    - parent: (c - 1)/2

```java
public class Heap<E extends Comparable<E>> implements IPriorityQueue<E> {

    private E[] elements;
    private int size;
    private int capacity;
```

```
    public static final int INITIAL_CAPACITY = 10;

    // constructs a new empty priority queue
    public Heap() {
        this(INITIAL_CAPACITY);
    }

    public Heap(int capacity){
        this.elements = (E[]) new Object[capacity];
        this.size = 0;
        this.capacity = capacity;
    }

    // Adds the given value to this priority queue in order.
    public void add(E value) {
        // resize to enlarge the heap if necessary
        if (size == capacity) {
            reallocate();
        }

        elements[size] = value;  // add as rightmost leaf

        // "bubble up" as necessary to fix ordering
        int index = size;
        boolean found = false;
        while (!found && hasParent(index)) {
            int parent = parent(index);
            if (elements[index].compareTo(elements[parent]) < 0) {
                swap(elements, index, parent(index));
                index = parent(index);
            } else {
                found = true;  // found proper location; stop
            }
        }

        size++;
    }

    public E peek() {
        return elements[0];
    }

    public E remove() {   // precondition: queue is not empty
        E result = elements[0];      // last leaf -> root
        elements[0] = elements[size - 1];
        size--;
        int index = 0;    // "bubble down" to fix ordering
        boolean found = false;
        while (!found && hasLeftChild(index)) {
            int left = leftChild(index);
            int right = rightChild(index);
            int child = left;
            if (hasRightChild(index) &&
                    elements[right].compareTo(elements[left]) < 0) {
                child = right;
            }
```

```
            if (elements[index].compareTo(elements[child]) > 0) {
                swap(elements, index, child);
                index = child;
            } else {
                found = true;  // found proper location; stop
            }
        }
        return result;
    }

    @Override
    public void clear() {
        Arrays.fill(elements, null);
        size = 0;
    }

    @Override
    public boolean isEmpty() {
        return size > 0;
    }

    @Override
    public int size() {
        return size;
    }

    // helpers for navigating indexes up/down the tree
    private int parent(int index)        { return (index - 1)/2; }
    private int leftChild(int index)     { return 2 * index + 1; }
    private int rightChild(int index)    { return 2 * index + 2; }
    private boolean hasParent(int index) { return index > 0; }
    private boolean hasLeftChild(int index) {
        return leftChild(index) < size;
    }
    private boolean hasRightChild(int index) {
        return rightChild(index) < size;
    }
    private void swap(E[] a, int index1, int index2) {
        E temp = a[index1];
        a[index1] = a[index2];
        a[index2] = temp;
    }

    private void reallocate(){
        this.capacity = 2 * this.capacity;
        elements = Arrays.copyOf(elements, capacity);
    }
}
```

Performance

Operations	Big-O Complexity
Add	O(logN) [height of tree]
Peek	O(1)
Remove	O(logN) [height of tree]

2. Priority Queues

Overview

- A priority queue is a data structure in which only the highest priority item is accessible.
  - In computer science, a heap is used as the basis of a very efficient algorithm for sorting arrays, called heapsort (we'll cover later).
  - The heap is also used to implement a special kind of queue called a priority queue. However, the heap is not very useful as an abstract data type (ADT) on its own. Consequently, we will not create a Heap interface or code a class that implements it.
  - Instead we will incorporate its algorithms when we implement a priority queue class and heapsort.

Class `java.util.PriorityQueue<E>`

Constructor	Description
`PriorityQueue()`	Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
`PriorityQueue(Comparator<? super E> comparator)`	Creates a PriorityQueue with the default initial capacity and whose elements are ordered according to the specified comparator.

Modifier and Type	Method	Description
`boolean`	`add(E e)`	Inserts the specified element into this priority queue.
`void`	`clear()`	Removes all of the elements from this priority queue.

Modifier and Type	Method	Description
`Comparator<? super E>`	`comparator()`	Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements.
`boolean`	`contains(Object o)`	Returns true if this queue contains the specified element.
`Iterator<E>`	`iterator()`	Returns an iterator over the elements in this queue.
`boolean`	`offer(E e)`	Inserts the specified element into this priority queue.
`E`	`peek()`	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
`E`	`poll()`	Retrieves and removes the head of this queue, or returns null if this queue is empty.
`boolean`	`remove(Object o)`	Removes a single instance of the specified element from this queue, if it is present.
`int`	`size()`	Returns the number of elements in this collection.

- The class `java.util.PriorityQueue` uses an array of type `Object[]` for heap storage.

Class `PriorityQueue<E>`

- In our case, as an example, we use `ArrayList` for implementing a PriorityQueue class.

Data Field/Attribute	Description
`ArrayList<E> theData`	An ArrayList to hold the data
`Comparator<E> comparator`	An optional object that implements the `Comparator<E>` interface by providing a `compare` method

Constructor	Description
`KWPriorityQueue()`	Constructs a heap-based priority queue that uses the elements' natural ordering
`KWPriorityQueue(Comparator<E> comp)`	Constructs a heap-based priority queue that uses the `compare` method of `Comparator comp` to determine the ordering of the elements

Private Method	Description
`private int compare(E left, E right)`	Compares two objects and returns a negative number if object `left` is less than object `right`, zero if they are equal, and a positive number if object `left` is greater than object `right`
`private void swap(int i, int j)`	Exchanges the object references in `theData` at indexes `i` and `j`

```
package datastructures.heap;

import java.util.*;

/** The PriorityQueue implements the Queue interface
 by building a heap in an ArrayList. The heap is structured
 so that the “smallest” item is at the top.
 */

public class PriorityQueue<E> extends AbstractQueue<E> implements Queue<E> {
// Data Fields
    /**
     * The ArrayList to hold the data.
     */
    private ArrayList<E> data;
    /**
     * An optional reference to a Comparator object.
     */
    Comparator<E> comparator = null;

    // Methods
// Constructor
    public PriorityQueue() {
        data = new ArrayList<>();
    }

    /** Creates a heap-based priority queue with the specified initial
     capacity that orders its elements according to the specified
     comparator.
     @param capacity The initial capacity for this priority queue
     @param comp The comparator used to order this priority queue
     @throws IllegalArgumentException if capacity is less than 1
     */
    public PriorityQueue(int capacity, Comparator<E> comp) {
        if (capacity < 1)
            throw new IllegalArgumentException();
        data = new ArrayList<>();
        comparator = comp;
    }

    /**
     * Insert an item into the priority queue.
     * pre: The ArrayList theData is in heap order.
     * post: The item is in the priority queue and
```

```
     * theData is in heap order.
     *
     * @param item The item to be inserted
     * @throws NullPointerException if the item to be inserted is null.
     */
    @Override
    public boolean offer(E item) {
        // Add the item to the heap.
        data.add(item);
        // child is newly inserted item.
        int child = data.size()-1;
        int parent = (child - 1) /2; // Find child's parent.
        // Reheap
        while (parent >= 0 && compare(data.get(parent),
                data.get(child)) > 0) {
            swap(parent, child);
            child = parent;
            parent = (child - 1) /2;
        }
        return true;
    }

    /**
     * Remove an item from the priority queue
     * pre: The ArrayList theData is in heap order.
     * post: Removed smallest item, theData is in heap order.
     *
     * @return The item with the smallest priority value or null if empty.
     */
    @Override
    public E poll() {
        if (isEmpty()) {
            return null;
        }
        // Save the top of the heap.
        E result = data.get(0);
        // If only one item then remove it.
        if (data.size() == 1) {
            data.remove(0);
            return result;
        }
        /* Remove the last item from the ArrayList and place it into the first
position. */
        data.set(0, data.remove(data.size() - 1));
        // The parent starts at the top.
        int parent = 0;
        while (true) {
            int leftChild = 2 * parent + 1;
            if (leftChild >= data.size()) {
                break; // Out of heap.
            }
            int rightChild = leftChild + 1;
            int minChild = leftChild; // Assume leftChild is smaller.
            // See whether rightChild is smaller.
            if (rightChild < data.size()
                    && compare(data.get(leftChild),
```

```
                data.get(rightChild)) > 0) {
            minChild = rightChild;
        }
        // assert: minChild is the index of the smaller child.
        // Move smaller child up heap if necessary.
        if (compare(data.get(parent),
                data.get(minChild)) > 0) {
            swap(parent, minChild);
            parent = minChild;
        } else { // Heap property is restored.
            break;
        }
    }
    return result;
}

@Override
public E peek() {
    return data.get(0);
}

/** Compare two items using either a Comparator object's compare method
 or their natural ordering using method compareTo.
 @pre: If comparator is null, left and right implement Comparable<E>.
 @param left One item
 @param right The other item
 @return Negative int if left less than right,
 0 if left equals right,
 positive int if left > right
 @throws ClassCastException if items are not Comparable
 */
@SuppressWarnings("unchecked")
private int compare(E left, E right) {
    if (comparator != null) {
        // A Comparator is defined.
        return comparator.compare(left, right);
    } else {
        // Use left's compareTo method.
        return ((Comparable<E>) left).compareTo(right);
    }
}

@Override
public Iterator<E> iterator() {
    return null;
}

@Override
public int size() {
    return data.size();
}

private void swap(int index1, int index2) {
    E temp = data.get(index1);
    data.set(index1, data.get(index2));
    data.set(index2, temp);
```

```
    }
}
```

Interface `Comparable<E>` and `Comparator<E>`

- How do we compare elements in a PriorityQueue?

Interface `Comparable<E>`

- In many cases, we will insert objects that implement `Comparable<E>` and use their natural ordering as specified by method `compareTo`.
- `Comparable<E>` is used to define the natural ordering of an object type.

Interface `Comparator<E>`

- However, we may need to insert objects that do not implement `Comparable<E>`, or we may want to specify a different ordering from that defined by the object's `compareTo` method.
    - For example, files to be printed may be ordered by their name using the compareTo method, but
      we may want to assign priority based on their length.
- The Java API contains the `Comparator<E>` interface, which allows us to specify alternative ways to compare objects. An implementer of the `Comparator<E>` interface must define a `compare` method that is similar to `compareTo` except that it has two parameters.

```
public interface Comparator<T> {
    public int compare(T first, T second);
}
```

Example

```
public class RectangleAreaComparator implements Comparator<Rectangle> {
    // compare in ascending order by area (WxH)
    public int compare(Rectangle r1, Rectangle r2) {
        return r1.getArea() - r2.getArea();
    }
}
```

Using Comparators

1. TreeSet, TreeMap , PriorityQueue can use Comparator:

```
Comparator<Rectangle> comp = new RectangleAreaComparator();
Set<Rectangle> set = new TreeSet<Rectangle>(comp);
Queue<Rectangle> pq = new PriorityQueue<Rectangle>(10,comp);
```

2. Searching and sorting methods can accept Comparators.

```
Arrays.binarySearch(array, value, comparator)
Arrays.sort(array, comparator)
Collections.binarySearch(list, comparator)
Collections.max(collection, comparator)
Collections.min(collection, comparator)
Collections.sort(list, comparator)
```

3. Methods are provided to reverse a Comparator's ordering:

```
public static Comparator Collections.reverseOrder()
public static Comparator Collections.reverseOrder(comparator)
```

Example: Using Comparator for ordering a Priority Queue

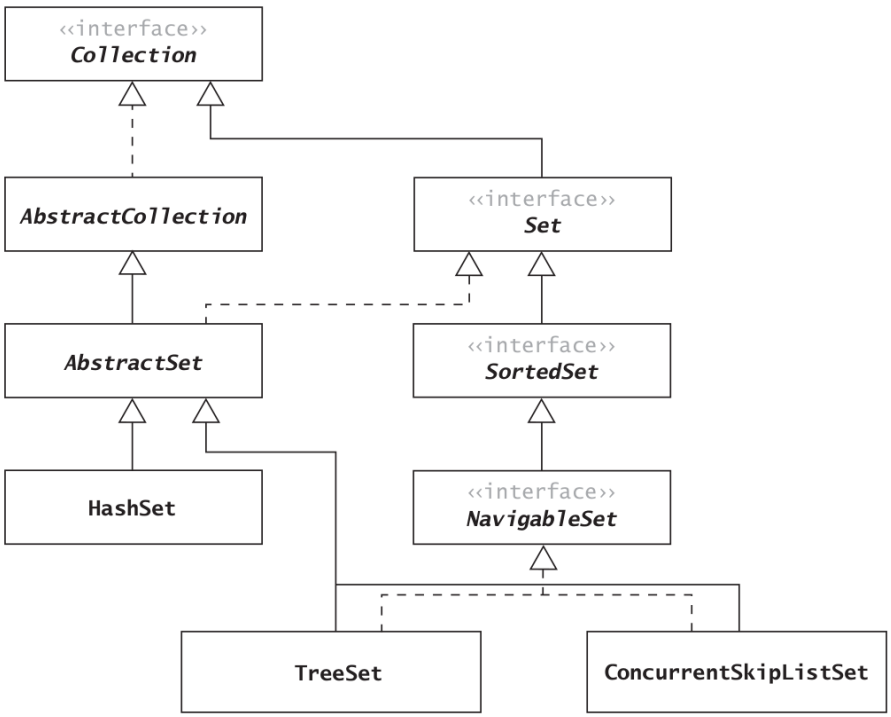
```
public class LengthComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        if (s1.length() != s2.length()) {
            // if lengths are unequal, compare by length
            return s1.length() - s2.length();
        } else {
            // break ties by ABC order
            return s1.compareTo(s2);
        }
    }
}

Queue<String> pq = new PriorityQueue<String>(100, new LengthComparator());
```

- Observe that, in the example, we are still making use of the fact that the class `String` is implementing `Comparable<E>` interface to define its own natural ordering by implementing `compareTo` method.

1. Sets

The `Set` Hierarchy

FIGURE 7.1
The `Set` Hierarchy



- Figure above shows the part of the `Collections Framework` that relates to sets. It includes interfaces `Set`, `Sortedset`, and `Navigableset`; abstract class `AbstractSet`; and actual classes `HashSet`, `TreeSet`, and `ConcurrentSkipListSet`. The `HashSet` is a set that is implemented using a hash table (discussed later). The `TreeSet` is implemented using a special kind of binary search tree, called the `Red-Black tree` (discussed in future chapters). The `ConcurrentSkipListSet` is implemented using a `skip list` (discussed in future chapters).

Interface `java.util.Set`

Modifier and Type	Method	Description
`boolean`	`add(E e)`	Adds the specified element to this set if it is not already present (optional operation).
`boolean`	`addAll(Collection<? extends E> c)`	Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
`void`	`clear()`	Removes all of the elements from this set (optional operation).

Modifier and Type	Method	Description
`boolean`	`contains(Object o)`	Returns true if this set contains the specified element.
`boolean`	`containsAll(Collection<?> c)`	Returns true if this set contains all of the elements of the specified collection.
`boolean`	`equals(Object o)`	Compares the specified object with this set for equality.
`int`	`hashCode()`	Returns the hash code value for this set.
`boolean`	`isEmpty()`	Returns true if this set contains no elements.
`Iterator<E>`	`iterator()`	Returns an iterator over the elements in this set.
`boolean`	`remove(Object o)`	Removes the specified element from this set if it is present (optional operation).
`boolean`	`removeAll(Collection<?> c)`	Removes from this set all of its elements that are contained in the specified collection (optional operation).
`boolean`	`retainAll(Collection<?> c)`	Retains only the elements in this set that are contained in the specified collection (optional operation).
`int`	`size()`	Returns the number of elements in this set (its cardinality).
`Object[]`	`toArray()`	Returns an array containing all of the elements in this set.

Comparison of `List` and `Set`

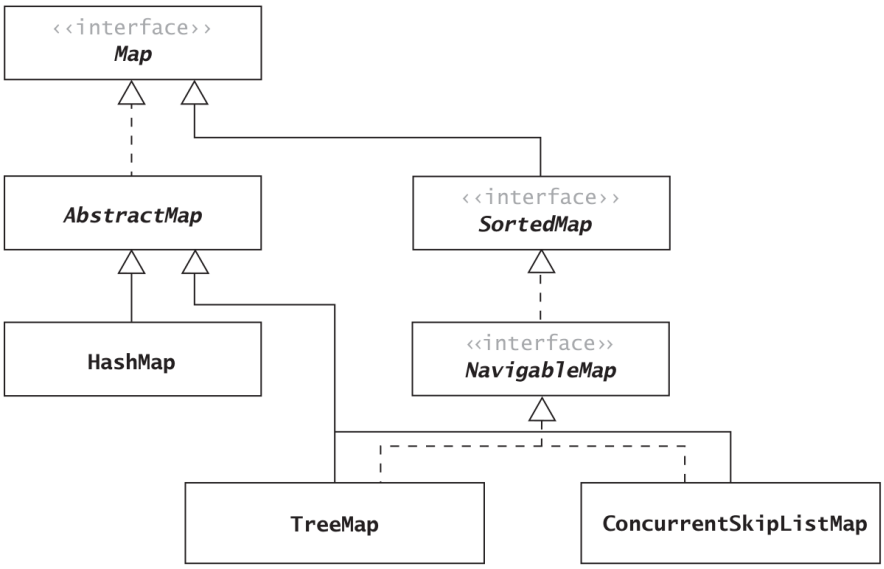
- Collections implementing the Set interface must contain unique elements. Unlike the List.add method, the Set.add method will return false if you attempt to insert a duplicate item.
- Unlike a List, a Set does not have a get method. Therefore, elements cannot be accessed by index. So if setA is a Set object, the method call setA.get(0) would cause the syntax error method get(int) not found.
- Although you can't reference a specific element of a Set, you can iterate through all its elements using an Iterator object. The loop below accesses each element of Set object setA. However, the elements will be accessed in arbitrary order. This means that they will not necessarily be accessed in the order in which they were inserted.

2. Maps

The `Map` Hierarchy

- Figure below shows part of the `Map` hierarchy in the `Java API`. Although not strictly part of the `Collection` hierarchy, the `Map` interface defines a structure that relates elements in one set to elements in another set. The first set, called the keys, must implement the `Set` interface; that is, the keys are unique. The second set is not strictly a `Set` but an arbitrary `Collection` known as the values. These are not required to be unique. The `Map` is a more useful structure than the `Set`. In fact, the `Java API` implements the `Set` using a `Map`.

FIGURE 7.3
The Map Hierarchy



- The `TreeMap` uses a `Red-Black binary search tree` (discussed in future chapters) as its underlying data structure, and the `ConcurrentSkipListMap` uses a `skip list` (discussed in future chapters) as its underlying data structure. We will focus on the `HashMap` and show how to implement it later in the chapter.

Interface `java.util.Map`

Modifier and Type	Method	Description
`void`	`clear()`	Removes all of the mappings from this map (optional operation).
`boolean`	`containsKey(Object key)`	Returns true if this map contains a mapping for the specified key.
`boolean`	`containsValue(Object value)`	Returns true if this map maps one or more keys to the specified value.
`Set<Map.Entry<K, V>>`	`entrySet()`	Returns a Set view of the mappings contained in this map.

Modifier and Type	Method	Description
`boolean`	`equals(Object o)`	Compares the specified object with this map for equality.
`default void`	`forEach(BiConsumer<? super K, ? super V> action)`	Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
`V`	`get(Object key)`	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
`default V`	`getOrDefault(Object key, V defaultValue)`	Returns the value to which the specified key is mapped, or `defaultValue` if this map contains no mapping for the key.
`int`	`hashCode()`	Returns the hash code value for this map.
`boolean`	`isEmpty()`	Returns true if this map contains no key-value mappings.
`Set<K>`	`keySet()`	Returns a Set view of the keys contained in this map.
`default V`	`merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)`	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
`V`	`put(K key, V value)`	Associates the specified value with the specified key in this map (optional operation).
`void`	`putAll(Map<? extends K, ? extends V> m)`	Copies all of the mappings from the specified map to this map (optional operation).
`default V`	`putIfAbsent(K key, V value)`	If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.

Modifier and Type	Method	Description
`V`	`remove(Object key)`	Removes the mapping for a key from this map if it is present (optional operation).
`default boolean`	`remove(Object key, Object value)`	Removes the entry for the specified key only if it is currently mapped to the specified value.
`default V`	`replace(K key, V value)`	Replaces the entry for the specified key only if it is currently mapped to some value.
`default boolean`	`replace(K key, V oldValue, V newValue)`	Replaces the entry for the specified key only if currently mapped to the specified value.
`default void`	`replaceAll(BiFunction<? super K, ? super V, ? extends V> function)`	Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
`int`	`size()`	Returns the number of key-value mappings in this map.
`Collection<V>`	`values()`	Returns a Collection view of the values contained in this map.

3. Hash Tables

- Before we discuss the details of implementing the required methods of the Set and Map interfaces, we will describe a data structure, the hash table, that can be used as the basis for such an implementation.
- Using a hash table enables us to retrieve an item in constant time (expected O(1)). We say expected O(1) rather than just O(1) because there will be some cases where the performance will be much worse than O(1) and may even be O(n), but on the average, we expect that it will be O(1).
- Properties of Hash Functions
    - Simple and efficient to compute
    - Provides a large set of possible values to map domain
    - Should provide an (almost) equal distribution (mapping) of values

- The probability of two distinct items to generate the same hash value should be as low as possible

Open Addressing

- Two ways to organize a hash tables:
  - open addressing
  - chaining
- **Procedure:** Open Addressing
  - Each hash table element (type Object) references a single key–value pair.
  - We can use the following simple approach (called `linear probing`) to access an item in a hash table.
  - If the index calculated for an item's key is occupied by an item with that key, we have found
    the item.
  - If that element contains an item with a different key, we increment the index by 1.
  - We keep incrementing the index (modulo the table length) until either we find the key we are seeking or we reach a null entry.
  - A null entry indicates that the key is not in the table.

```
Algorithm For Accessing an Item in a Hash Table

Compute the index by taking the item's hashCode() % table.length.
if table[index] is null
    The item is not in the table.
else if table[index] is equal to the item
    The item is in the table.
else
    Continue to search the table by incrementing the index until either the item
is found   or a null entry is found.
```

Table Wraparound and Search Termination

- Note that as you increment the table index, your table should wrap around (as in a circular array) so that the element with subscript 0 "follows" the element with subscript table.length - 1.
- This enables you to use the entire table, not just the part with subscripts larger than the hash code value, but it leads to the potential for an infinite loop in Step 6 of the algorithm. If the table is full and the objects examined so far do not match the one you are seeking, how do you know when to stop?
- One approach would be to stop when the index value for the next probe is the same as the hash code value for the object. This means that you have come full circle to the starting value for the index.

- A second approach would be to ensure that the table is never full by increasing its size after an insertion **if its occupancy rate exceeds a specifieif its occupancy rate exceeds a specified thresholdd threshold**. **This is the approach that we take in our implementation.**

Traversing a Hash Table

- One thing that you cannot do is traverse a hash table in a meaningful way.

Deleting an Item Using Open Addressing

- When an item is deleted, we cannot just set its table entry to `null`. If we do, then when we search for an item that may have collided with the deleted item, we may incorrectly conclude that the item is not in the table. (Because the item that collided was inserted after the deleted item, we will have stopped our search prematurely.)
- By storing a dummy value when an item is deleted, we force the search algorithm to keep looking until either the desired item is found or a null value, representing a free cell, is located. Although the use of a dummy value solves the problem, keep in mind that it can lead to search inefficiency, particularly when there are many deletions. Removing items from the table does not reduce the search time because the dummy value is still in the table and is part of a search chain.
- In fact, you **cannot** even replace a deleted value with a new item because you still need to go to the end of the search chain to ensure that the new item is not already present in the table. So deleted items waste storage space and reduce search efficiency.
- In the worst case, if the table is almost full and then most of the items are deleted, you will have $O(n)$ performance when searching for the few items remaining in the table.

Reducing Collisions by Expanding the Table Size

- The first step in reducing these collisions is to use a prime number for the size of the table.
- In addition, the probability of a collision is proportional to how full the table is. Therefore, when the hash table becomes sufficiently full, a larger table should be allocated and the entries reinserted.
- You expand a hash table using an algorithm called `rehashing`.

```
Algorithm For Rehashing
1. Allocate a new hash table with twice the capacity of the original.
2. Reinsert each old table entry that has not been deleted into the new hash
table.
3. Reference the new table instead of the original.
```

Reducing Collisions Using Quadratic Probing

- The problem with `linear probing` is that it tends to form clusters of keys in the table, causing longer search chains.
- One approach to reduce the effect of clustering is to use `quadratic probing` instead of `linear probing`. In quadratic probing, the increments form a quadratic series $(1 + 2^2 + 3^2 + \cdots)$.

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

- **Problems with Quadratic Probing**
  - One disadvantage of quadratic probing is that the next index calculation is a bit time-consuming as it involves a multiplication, an addition, and a modulo division.
  - A more efficient way to calculate the next index follows:

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

  - A more serious problem with quadratic probing is that not all table elements are examined when looking for an insertion index, so it is possible that an item can't be inserted even when the table is not full.
  - It is also possible that your program can get stuck in an infinite loop while searching for an empty slot.
  - It can be proved that if the table size is a prime number and the table is never more than half full, this can't happen. However, requiring that the table be half empty at all times wastes quite a bit of memory.
  - For these reasons, we will use linear probing in our implementation.

Chaining

- An alternative to open addressing is a technique called `chaining`, in which each table element references a linked list that contains all the items that hash to the same table index.
- This linked list is often called a `bucket`, and this approach is sometimes called `bucket hashing`.
- Instead of incrementing the table index to access the next item with a particular hash code value, you traverse the linked list referenced by the table element with index `hashCode() % table.length`.
- One advantage of chaining is that only items that have the same value for `hashCode() % table. length` will be examined when looking for an object. In open addressing, search chains can overlap, so a search chain may include items in the table that have different starting index values.
- A second advantage is that you can store more elements in the table than the number of table slots (indexes), which is not the case for open addressing.

- To **delete** an item, simply remove it from the list. In contrast to open addressing, removing an item actually deletes it, so it will not be part of future search chains.

Performance of Hash Tables

- `load factor`: The number of filled cells divided by table size.
  - The load factor has the greatest effect on hash table performance.
  - Lower the load factor the better the performance because there is less chance of collision.
  - If there are no collisions, the performance for search and retrieval is $O(1)$.
- **Performance of Open Addressing versus Chaining**
  - Expected number of comparisons, c, for open addressing with linear probing and a load factor L:
    - $c = \frac{1}{2}(1 + \frac{1}{1 - L})$
    - L = number of filled cells/table size
  - Expected number of comparisons, c, for chaining and a load factor L (L is average number of items in a list here = # of items divided by table size:
    - $c = 1 + \frac{L}{2}$
  - For values of L between 0.0 and 0.75, the results for chaining are similar to those of linear probing. But chaining gives better performance than linear probing for higher load factors.
  - **Quadratic probing** gives performance that is between those of linear probing and chaining.
- **Performance of Hash Tables versus Sorted Arrays and Binary Trees**
  - The performance of hashing is certainly preferable to that of binary search of an array (or a binary search tree), particularly if L is less than 0.75.
  - However, the trade-off is that the lower the load factor, the more unfilled storage cells there are in a hash table, whereas there are no empty cells in a sorted array.
  - Because a binary search tree requires three references per node (the item, the left subtree, and the right subtrees), more storage would be required for a binary search tree than for a hash table with a load factor of 0.75.
- **Storage Requirements for Open Addressing and Chaining**
  - Next, we consider the effect of chaining on storage requirements. For a table with a load factor of L, the number of table elements required is n (the size of the table).
  - For open addressing, the number of references to an item (a key–value pair) is n.
  - For chaining, the average number of nodes in a list is **L**. If we use the `Java API LinkedList`, there will be three references in each node (the item, the next list element, and the previous element). However, we could use our own single-linked list and eliminate the previous-element reference (at some time cost for deletions). Therefore, we will require storage for $n + n * 2L$ references.

Implementing the Hash Table

Interface `IHashMap`

```
package datastructures.setmap;

public interface IHashMap<K, V> {
    V get(Object key);

    boolean isEmpty();

    V put(K key, V value);

    V remove(Object key);

    int size();
}
```

Class `HashTableOpen`

```
package datastructures.setmap;

/** Hash table implementation using open addressing. */
public class HashTableOpen<K, V> implements IHashMap<K, V> {
    // Insert inner class Entry<K, V> here.
    /** Contains key-value pairs for a hash table. */
    private static class Entry<K, V> {
        /** The key */
        private final K key;
        /** The value */
        private V value;
        /** Creates a new key-value pair.
         @param key The key
         @param value The value
         */
        public Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }
        /** Retrieves the key.
         @return The key
         */
        public K getKey() {
            return key;
        }
        /** Retrieves the value.
         @return The value
         */
        public V getValue() {
            return value;
```

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

    // Data Fields
    private Entry<K, V>[] table;
    private static final int START_CAPACITY = 101;

    private double LOAD_THRESHOLD = 0.75;
    private int numKeys;
    private int numDeletes;
    private final Entry<K, V> DELETED =
            new Entry<>(null, null);

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

    /** Finds either the target key or the first empty slot in the
     search chain using linear probing.
     @pre The table is not full.
     @param key The key of the target object
     @return The position of the target or the first empty slot if
     the target is not in the table.
     */
    private int find(Object key) {
        // Calculate the starting index.
        int index = key.hashCode() % table.length;
        if (index < 0)
            index += table.length;
            // Make it positive.
            // Increment index until an empty slot is reached or the key is
found.
        while ((table[index] != null)
                && (!key.equals(table[index].getKey()))) {
            index++;
            // Check for wraparound.
            if (index >= table.length)
                index = 0;
                // Wrap around.
        }
        return index;
    }

    /** Method get for class HashtableOpen.
     @param key The key being sought
     @return the value associated with this key if found;
```

```
     otherwise, null
     */
    @Override
    public V get(Object key) {
        // Find the first table element that is empty
        // or the table element that contains the key.
        int index = find(key);
        // If the search is successful, return the value.
        if (table[index] != null)
            return table[index].getValue();
        else
            return null; // key not found.
    }

    /** Method put for class HashtableOpen.
     @post This key-value pair is inserted in the
     table and numKeys is incremented. If the key is already
     in the table, its value is changed to the argument
     value and numKeys is not changed. If the LOAD_THRESHOLD
     is exceeded, the table is expanded.
     @param key The key of item being inserted
     @param value The value for this key
     @return Old value associated with this key if found;
     otherwise, null
     */
    @Override
    public V put(K key, V value) {
        // Find the first table element that is empty
        // or the table element that contains the key.
        int index = find(key);
        // If an empty element was found, insert new entry.
        if (table[index] == null) {
            table[index] = new Entry<>(key, value);
            numKeys++;
            // Check whether rehash is needed.
            double loadFactor =
                    (double) (numKeys + numDeletes) / table.length;
            if (loadFactor > LOAD_THRESHOLD)
                rehash();
            return null;
        }
        // assert: table element that contains the key was found.
        // Replace value for this key.
        V oldVal = table[index].getValue();
        table[index].setValue(value);
        return oldVal;
    }

    @Override
    public V remove(Object key) {
        // Find the first table element that is empty or the table element that
contains the key.
        int index = find(key);
        // if an empty element was found return null
        if(table[index] == null || table[index] == this.DELETED){
            return null;
```

```
        }
        // Key was found. Remove this table element by setting it to reference
DELETED, increment
        // numDeletes, and decrement numKeys.
        V oldVal = table[index].getValue();
        table[index] = this.DELETED;
        this.numDeletes++;
        this.numKeys--;
        // Return the value associated with this key.
        return oldVal;
    }

    @Override
    public int size() {
        return this.numKeys;
    }

    @Override
    public boolean isEmpty() {
        return (this.numKeys != 0);
    }

    /** Expands table size when loadFactor exceeds LOAD_THRESHOLD
     @post The size of the table is doubled and is an odd integer.
     Each nondeleted entry from the original table is
     reinserted into the expanded table.
     The value of numKeys is reset to the number of items
     actually inserted; numDeletes is reset to 0.
     */
    private void rehash() {
        // Save a reference to oldTable.
        Entry<K, V>[] oldTable = table;
        // Double capacity of this table.
        table = new Entry[2 * oldTable.length + 1];
        // Reinsert all items in oldTable into expanded table.
        numKeys = 0;
        numDeletes = 0;
        for (int i = 0; i < oldTable.length; i++) {
            if ((oldTable[i] != null) && (oldTable[i] != DELETED)) {
                // Insert entry in expanded table
                put(oldTable[i].getKey(), oldTable[i].getValue());
            }
        }
    }
}
```

Class `HashTableChain`

```
package datastructures.setmap;

import java.util.*;
/** Hash table implementation using chaining. */
public class HashTableChain<K, V> implements IHashMap<K, V> {
```

```
    // Insert inner class Entry<K, V> here.
    /** Contains key-value pairs for a hash table. */
    private static class Entry<K, V> {
        /** The key */
        private final K key;
        /** The value */
        private V value;
        /** Creates a new key-value pair.
         @param key The key
         @param value The value
         */
        public Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }
        /** Retrieves the key.
         @return The key
         */
        public K getKey() {
            return key;
        }
        /** Retrieves the value.
         @return The value
         */
        public V getValue() {
            return value;
        }
        /** Sets the value.
         @param val The new value
         @return The old value
         */
        public V setValue(V val) {
            V oldVal = value;
            value = val;
            return oldVal;
        }
    }

    /** The table */
    private LinkedList<Entry<K, V>>[] table;
    /** The number of keys */
    private int numKeys;
    /** The capacity */
    private static final int CAPACITY = 101;
    /** The maximum load factor */
    private static final double LOAD_THRESHOLD = 3.0;
    // Constructor
    public HashTableChain() {
        table = new LinkedList[CAPACITY];
    }

    /** Method get for class HashtableChain.
     @param key The key being sought
     @return The value associated with this key if found;
     otherwise, null
     */
```

```
    @Override
    public V get(Object key) {
        int index = key.hashCode() % table.length;
        if (index < 0)
            index += table.length;
        if (table[index] == null)
            return null; // key is not in the table.
        // Search the list at table[index] to find the key.
        for (Entry<K, V> nextItem : table[index]) {
            if (nextItem.getKey().equals(key))
                return nextItem.getValue();
        }
        // assert: key is not in the table.
        return null;
    }

    /** Method put for class HashtableChain.
     @post This key-value pair is inserted in the
     table and numKeys is incremented. If the key is already
     in the table, its value is changed to the argument
     value and numKeys is not changed.
     @param key The key of item being inserted
     @param value The value for this key
     @return The old value associated with this key if
     found; otherwise, null
     */
    @Override
    public V put(K key, V value) {
        int index = key.hashCode() % table.length;
        if (index < 0)
            index += table.length;
        if (table[index] == null) {
            // Create a new linked list at table[index].
            table[index] = new LinkedList<>();
        }
        // Search the list at table[index] to find the key.
        for (Entry<K, V> nextItem : table[index]) {
            // If the search is successful, replace the old value.
            if (nextItem.getKey().equals(key)) {
                // Replace value for this key.
                V oldVal = nextItem.getValue();
                nextItem.setValue(value);
                return oldVal;
            }
        }
        // assert: key is not in the table, add new item.
        table[index].addFirst(new Entry<>(key, value));
        numKeys++;
        if (numKeys > (LOAD_THRESHOLD * table.length))
            rehash();
        return null;
    }

    @Override
    public V remove(Object key) {
        // Set index to key.hashCode() % table.length.
```

```
        // if index is negative, add table.length.
        int index = key.hashCode() % table.length;
        if(index < 0){
            index += table.length;
        }

        // if table[index] is null
        // key is not in the table; return null.
        if(table[index] == null){
            return null;
        }
        for(int i = 0 ; i < table[index].size(); i++){
            //Search the list at table[index] to find the key.
            if(table[index].get(i).getKey() == key){
                V oldVal = table[index].get(i).getValue();

                // if the search is successful
                // Remove the entry with this key and decrement numKeys.
                table[index].remove(i);
                this.numKeys--;

                //if the list at table[index] is empty
                //Set table[index] to null.
                if(table[index].isEmpty()){
                    table[index] = null;
                }
                //Return the value associated with this key.
                return oldVal;
            }
        }
        //The key is not in the table; return null.
        return null;
    }

    @Override
    public int size() {
        return this.numKeys;
    }

    @Override
    public boolean isEmpty() {
        return (this.numKeys != 0);
    }

    private void rehash() {
        // Save a reference to oldTable.
        LinkedList<Entry<K, V>>[] oldTable = table;
        // Double capacity of this table.
        table = new LinkedList[2 * oldTable.length + 1];

        // Reinsert all items in oldTable into expanded table.
        this.numKeys = 0;
        for (LinkedList<Entry<K, V>> entries : oldTable) {
            for (Entry<K, V> entry : entries) {
                int index = entry.getKey().hashCode() % table.length;
                if (index < 0)
```

```
                    index += table.length;
                if (table[index] == null) {
                    // Create a new linked list at table[index].
                    table[index] = new LinkedList<>();
                }
                table[index].add(entry);
                this.numKeys++;
            }
        }
    }
}
```

4. Implementation Considerations for Maps and Sets

Methods `hashCode` and `equals`

- Method `Object.equals` compares two objects based on their addresses, not their contents. Similarly, method `Object.hashCode` calculates an object's hash code based on its address, not its contents.
- Most predefined classes (e.g., String and Integer) override method `equals` and method `hashCode`.
- If you override the equals method, Java recommends you also override the `hashCode` method. Otherwise, your class will violate the Java contract for `hashCode`, which states:

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

Implementing `HashSetOpen`

- We can modify the hash table methods to implement a hash set. Table below compares corresponding Map and Set methods.

Map Method	Set Method
V get(Object key)	boolean contains(Object key)
V put(K key, V value)	boolean add(K key)
V remove(Object key)	boolean remove(Object key)

Writing `HashSetOpen` as an Adapter Class

- Instead of writing new methods from scratch, we can implement `HashSetOpen` as an adapter class with the data field.

```
private final IHashMap<K, V> setMap = new HashtableOpen<>();
```

- We can write methods contains, add, and remove as follows. Because the map stores key–value pairs, we will have each set element reference an Entry object with the same key and value.

Implementing the Java `Map` and `Set` Interfaces

- Java API uses a hash table to implement both the `Map` and `Set` interfaces (class `HashMap` and class `HashSet`).
- The task of implementing these interfaces is simplified by the inclusion of abstract classes `AbstractMap` and `AbstractSet` in the `Collections` framework. These classes provide implementations of several methods for the Map and Set interfaces. So if class `HashTableOpen` extends class `AbstractMap`, we can reduce the amount of additional work we need to do.
    - We should also replace `IHashMap` with `Map`. Thus, the declaration for `HashTableOpen` would be class `HashTableOpen<K, V>` extends `AbstractMap<K, V>` implements `Map<K, V>`.
- The `AbstractMap` provides relatively inefficient $O(n)$ implementations of the get and put methods. Because we overrode these methods in both our implementations (`HashTableOpen` and `HashTableChain`), we will get $O(1)$ expected performance. There are other, less critical methods that we don't need to provide because they are implemented in `AbstractMap` or its superclasses, such as `clear`, `isEmpty`, `putAll`, `equals`, `hashCode`, and `toString`.

Interface `Map.Entry` and Class `AbstractMap.SimpleEntry`

- One requirement on the key–value pairs for a Map object is that they implement the interface `Map.Entry<K, V>`, which is an inner interface of interface `Map`.
- This may sound a bit confusing, but what it means is that an implementer of the Map interface must contain an inner class `Entry`. The `AbstractMap` includes the inner class `SimpleEntry` that implements the `Map.Entry` interface. We can remove the inner class `Entry<K, V>` and replace new `Entry` with new `SimpleEntry`.

Creating a Set View of a Map

- Method `entrySet` creates a set view of the entries in a Map. This means that method entrySet
returns an object that implements the Set interface—that is, a set.

Classes `TreeMap` and `TreeSet`

- Besides `HashMap` and `HashSet`, the `Java Collections Framework` provides classes `TreeMap` and
  `TreeSet` that implement the `Map` and `Set` interfaces. These classes use a `Red–Black tree`, which is a balanced binary search tree.
- We discussed earlier that the performances for search, retrieval, insertion, and removal operations are better for a hash table than for a binary search tree (expected O(1) versus O(log n)).
- However, the primary advantage of a binary search tree is that it can be traversed in sorted order. Hash tables, however, can't be traversed in any meaningful way. Also, subsets based on a range of key values can be selected using a `TreeMap` but not by using a `HashMap`