Java Collections Framework

Abstract Data Structure Reference

CLASS SET #

Java Collections Framework

+ **Module:** `java.base`
+ **Package:** `java.util`

*NOTE: The Java Collections Framework includes additional classes, abstract classes, and interfaces not covered in this packet.

Collection<E> extends Iterable<E>

Description:

- The root interface in the collection hierarchy
- Represents a group of objects, known as its elements
- Some collections allow duplicate elements, while others do not
- Some collections are ordered, while others are unordered

Direct Sub-interfaces:

- **List<E>**
  - A linear, ordered collection
- **Set<E>**
  - A collection of uniquely different elements
- **Queue<E>**
  - A collection that adds and removes elements in a first-in/first-out (LIFO) order

interface Collection<E>

Interfaces contain no instance variables.

```
+ boolean add(E e)
+ boolean addAll(Collection<E> c)
+ boolean contains(Object o)
+ boolean containsAll(Collection<E> c)
+ boolean equals(Object o)*
+ boolean isEmpty()
+ Iterator<E> iterator()*
+ boolean remove(Object o)
+ boolean removeAll(Collection<E> c)
+ int size()
```

Additional methods not shown.

** Methods inherited from `Iterable<E>` interface.*

boolean add(E e)

Precondition: This `Collection` contains 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: If allowed, parameter `e` is added to this `Collection`. Each implementing subclass will define the criteria for which elements may be added to the Collection.
Returns `true` if the size of this `Collection` is changed. Otherwise, returns `false`.

boolean addAll(Collection<E> c)

Precondition: This `Collection` and parameter `c` each contain 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: If allowed, each element of parameter c is added to this `Collection`. Each implementing subclass will define the criteria for which elements may be added to the Collection.
Returns `true` if the size of this `Collection` is changed. Otherwise, returns `false`.

boolean contains(Object o)

Precondition: This `Collection` contains 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: Returns `true` if this `Collection` contains an element equivalent to parameter `o` (i.e., using `equals()`). Otherwise, returns `false`.

boolean containsAll(Collection<E> c)

Precondition: This `Collection` and parameter `c` each contain 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: Returns `true` if this `Collection` contains equivalent elements (i.e., using `equals()`) for each element of parameter `c`. Otherwise, returns `false`.

boolean equals(Object o)

Precondition: This `Collection` contains 0 or more elements of some data type (`E`) to be specified at runtime.
Parameter `o` may be `null`.

Postcondition: Returns true if parameter o is a Collection<E> and meets all of the appropriate requirements for equivalence of collections, as defined by the implementing subclass. Otherwise, returns `false`.

boolean isEmpty()

Precondition: This `Collection` contains 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: Returns `true` if this `Collection` currently contains 0 elements. Otherwise, returns `false`.

Iterator<E> iterator()

Precondition: This `Collection` contains 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: Returns an `Iterator` that can iterate through all elements contained within this `Collection` in some predetermined order. Each implementing subclass will define an appropriate order in which its custom `Iterator` shall visit the elements.

boolean remove(Object o)

Precondition: This `Collection` contains 0 or more elements of some data type (`E`) to be specified at runtime.
Parameter `o` may be `null`.

Postcondition: If this `Collection` contains an element equivalent to parameter `o` (i.e., using `equals()`), one matching element is removed from this `Collection` and the size of `Collection` size is reduced by 1.
Returns `true` if the size of this `Collection` is changed. Otherwise, returns `false`.

boolean removeAll(Collection<E> c)

Precondition: This `Collection` and parameter `c` each contain 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: For each item in parameter `c`, if this Collection contains an element equivalent to the item in `c` (i.e., using `equals()`) , one matching element is removed from this `Collection` and the size of `Collection` size is reduced by 1.
Returns `true` if the size of this `Collection` is changed. Otherwise, returns `false`.

int size()

Precondition: This `Collection` contains 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: Returns the number of elements currently contained within this `Collection`.

NOTE: This documentation is not exhaustive. Additional methods are not shown.

List<E>

Description:

- An ordered collection
- The user has precise control over where in the list each element is inserted
- Elements can be accessed by their integer index (position in the list)
- Elements can be search for in the list

Implementing Classes:

- **`ArrayList<E>`**
  - Array-based implementation that uses a static array as the underlying data structure for storing
  - User has direct access to any and all elements
  - Inserting and removing from the middle of the list requires shifting whole sections of elements within the array
- **`LinkedList<E>`**
  - Link-based implementation that uses a series of doubly-linked nodes to chain together elements in
  - User has direct access to the head and tail elements
  - All other elements require the list to be linearly traversed in order to reach them

interface List<E>

Interfaces contain no instance variables.

```
+ void add(int i, E e)
+ boolean add(E e)*
+ boolean addAll(int i, Collection<E> c)
+ boolean addAll(Collection<E> c)*
+ boolean contains(Object o)*
+ boolean containsAll(Collection<E> c)*
+ boolean equals(Object o)*
+ boolean isEmpty()*
+ E get(int i)
+ int indexOf(Object o)
+ Iterator<E> iterator()*
+ int lastIndexOf(Object o)
+ ListIterator<E> listIterator()
+ ListIterator<E> listIterator(int i)
+ E remove(int i)
+ boolean remove(Object o)*
+ boolean removeAll(Collection<E> c)*
+ E set(int i, E e)
+ int size()*
```

Additional methods not shown.

** Method inherited from `Collection<E>` interface.*

void add(int i, E e)

Precondition: Parameter `i` is a valid index position within this `List`.
Parameter `e` may be `null`.

Postcondition: Inserts parameter `e` into this `List` at index position `i`, shifting the relative positions of all subsequent elements to the next higher index positions. The size of this `List` is increased by 1.

boolean add(E e)

Precondition: Parameter `e` may be `null`.

Postcondition: Appends parameter `e` to the end of this `List` and the size of this `List` is increased by 1.
Returns `true`.

boolean addAll(int i, Collection<E> c)

Precondition: Parameter `i` is a valid index position within this `List`.
Parameter `c` may contain `null` elements.

Postcondition: Inserts all elements of parameter `c` into this `List` at index position `i`, shifting the relative positions of all subsequent elements to higher index positions such that lie beyond the last element of parameter `c`. The size of this `List` is increased by the number of elements added.

boolean equals(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns true if parameter `o` is a `List`, has the same size as this `List`, and contains all of the same elements in the same order as this `List`. Otherwise, returns `false`.

int get(int i)

Precondition: Parameter `i` is a valid index position within this `List`.

Postcondition: Returns the element at index position `i`.

int indexOf(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns the index position of the first occurrence in this `List` of an element equivalent to parameter `o` or -1 if no such element is contained within this `List`.

Iterator<E> iterator()

Precondition: The implementing class contains an inner class that implements the `ListIterator` interface.

Postcondition: Returns an `ListIterator` that can traverse through each element of this `List` in either ascending or descending order of index position.

int lastIndexOf(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns the index position of the last occurrence in this `List` of an element equivalent to parameter `o` or -1 if no such element is contained within this `List`.

ListIterator<E> listIterator()

Precondition: The implementing class contains an inner class that implements the `ListIterator` interface.

Postcondition: Returns a `ListIterator` that can traverse through each element of this `List` in either ascending or descending order of index position.

ListIterator<E> listIterator(int i)

Precondition: Parameter `i` is a valid index position within this `List`.
The implementing class contains an inner class that implements the `ListIterator` interface.

Postcondition: Returns a `ListIterator` that can traverse through each element of this `List` in either ascending or descending order of index position and is positioned to start at the index position specified by parameter `i`.

E remove(int i)

Precondition: Parameter `i` is a valid index position within this `List`.

Postcondition: The element at index position `i` has been removed and returned and the size of this `List` is reduced by 1.

E set(int i, E e)

Precondition: Parameter `i` is a valid index position within this `List`.
Parameter `e` may be `null`.

Postcondition: The element at index position `i` has been replaced by parameter `e` and the element originally at that position is returned.

NOTE: This documentation is not exhaustive. Additional methods are not shown.

ArrayList<E>

Description:

- Resizable-array implementation of the `List` interface
- Implements all optional list operations
- Permits all elements, including `null`
- Provides additional methods to manipulate the size of the array that is used internally to store the list.
- Each `ArrayList` instance has a capacity, dictated by the length of the array used to store the elements in the list
- Capacity is always at least as large as the list size
- As elements are added to an ArrayList, its capacity grows automatically
- Adding an element has constant *amortized* time cost.

class ArrayList<E>

```
- int size
- E[] elementData
```

Additional fields not shown.

```
+ void add(int i, E e)
+ boolean add(E e)
+ boolean addAll(int i, Collection<E> c)
+ boolean addAll(Collection<E> c)
+ boolean contains(Object o)
+ boolean containsAll(Collection<E> c)
+ void ensureCapacity(int minCapacity)
+ boolean equals(Object o)
+ boolean isEmpty()
+ int get(int i)
+ int indexOf(Object o)
+ Iterator<E> iterator()
+ int lastIndexOf(Object o)
+ ListIterator<E> listIterator()
+ ListIterator<E> listIterator(int i)
+ E remove(int i)
+ boolean remove(Object o)
+ boolean removeAll(Collection<E> c)
+ E set(int i, E e)
+ int size()
```

Additional methods not shown.

void ensureCapacity(int minCapacity)

Precondition: Parameter minCapacity is a positive integer that represents the minimum desired length of `elementData`.

Postcondition: If necessary to ensure the specified capacity, the length of `elementData` has been expanded by 50%.

Pseudocode: If the length of `elementData` is less than parameter `minCapacity`...
...declare a new `E[]` called `bigger` and assign to it a new array that is 50% longer than `elementData`.
...for each index position `i` in `elementData`...
...assign to index position `i` of `bigger` the element in index position `i` of `elementData`.
...reassign to `elementData` a reference to `bigger`.

void add(int i, E e)

Precondition: Parameter `i` is a valid index position within this `List`.
Parameter `e` may be `null`.

Postcondition: Inserts parameter `e` into this `List` at index position `i`, shifting the relative positions of all subsequent elements to the next higher index positions. The size of this `List` is increased by 1.

Pseudocode: Ensure the capacity of `elementData` to hold at least `size + 1` elements.
For each index position `j`, iterating backward from `size` through `i + 1`...
...Assign to index position `j` of `elementData` the element in index position `j - 1`.
Assign the value of parameter `e` into index position `i` of `elementData`.
Increment `size` by 1.

boolean add(E e)

Precondition: Parameter `e` may be `null`.

Postcondition: Appends parameter `e` to the end of this `List` and the size of this `List` is increased by 1.

Pseudocode:
Ensure the capacity of `elementData` to hold at least `size` + 1 elements.
Assign the value of parameter `e` into index position `size` of `elementData`.
Increment `size` by 1.

boolean addAll(int i, Collection<E> c)

Precondition: Parameter `i` is a valid index position within this `List`.
Parameter `c` may contain `null` elements.

Postcondition: Inserts all elements of parameter `c` into this `List` at index position `i`, shifting the relative positions of all subsequent elements to higher index positions such that lie beyond the last element of parameter `c`. The size of this `List` is increased by the number of elements added.

Pseudocode:
Ensure the capacity of `elementData` to hold at least `size` plus the length of parameter `c`.
For each element of parameter `c`, invoke the `add(E)` method to append the element to this `LinkedList`.

boolean equals(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns true if parameter `o` is a `List`, has the same size as this `List`, and contains all of the same elements in the same order as this `List`. Otherwise, returns `false`.

Pseudocode:
If parameter `o` is `null`, return `false`.
If parameter `o` is a reference to this `ArrayList`, return `true`.
If parameter `o` is not a `List`, return `false`.
If parameter `o` does not have the same size as this `ArrayList`, return `false`.
Declare a `List<E>` called `that` and assign to it the value of parameter `o` after typecasting it to a `List<E>`.
For each index position `i` in `this` list...
    ...If the element at position `i` from `this` list and the element from position `i` from `that` list do not contain equivalent elements (i.e., using `equals()`)...
        ...return `false`.
Return `true`.

int get(int i)

Precondition: Parameter `i` is a valid index position within this `List`.

Postcondition: Returns the element at index position `i`.

Pseudocode: Return the element at index position `i` of `elementData`.

int indexOf(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns the index position of the first occurrence in this `List` of an element equivalent to parameter `o` or -1 if no such element is contained within this `List`.

Pseudocode:
For each index position `i` in `elementData`...
    ...If the element at index position `i` is equivalent to parameter `o` (i.e., using `equals()`)...
        ...return the value of `i`.
Return -1.

int lastIndexOf(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns the index position of the last occurrence in this `List` of an element equivalent to parameter `o` or -1 if no such element is contained within this `List`.

Pseudocode:
For each index position `i` of `elementData`, from `size - 1` through `0`...
    ...If the element at index position `i` is equivalent to parameter `o` (i.e., using `equals()`)...
        ...return the value of `i`.
Return -1.

ListIterator<E> listIterator()

Precondition: `ArrayList` contains an inner class called `ListItr` that implements the `ListIterator` interface.

Postcondition: Returns a `ListIterator` that can traverse through each element of this `List` in either ascending or descending order of index position.

Pseudocode: Return a new instance of the `LinkedList` class' custom `ListItr` inner class.

ListIterator<E> listIterator(int i)

Precondition: Parameter `i` is a valid index position within this `List`.
`ArrayList` contains an inner class called `ListItr` that implements the `ListIterator` interface.

Postcondition: Returns a `ListIterator` that can traverse through each element of this `List` in either ascending or descending order of index position and is positioned to start at the index position specified by parameter `i`.

Pseudocode: Return a new instance of the `LinkedList` class' custom `ListItr` inner class initialized to start at index `i`.

E remove(int i)

Precondition: Parameter `i` is a valid index position within this `List`.

Postcondition: The element at index position `i` has been removed and returned and the size of this `List` is reduced by 1.

Pseudocode:
Declare an `E` variable called orig and assign to it the element at index position `i` of `elementData`.
For each index position `j`, iterating from `i + 1` through `size - 1`...
    ...Assign to index position `j - 1` of `elementData` the element in index position `j`.
Assign `null` into index position `size - 1` of `elementData`.
Decrement `size` by 1.
Return the value of `orig`.

E set(int i, E e)

Precondition: Parameter `i` is a valid index position within this `List`.
Parameter `e` may be `null`.

Postcondition: The element at index position `i` has been replaced by parameter `e` and the element originally at that position is returned.

Pseudocode:
Declare an `E` variable called orig and assign to it the element at index position `i` of `elementData`.
Assign the value of parameter `e` into index position `i` of `elementData`.
Return the value of `orig`.

NOTE: This documentation is not exhaustive. Additional methods are not shown.

LinkedList<E>

Description:

- Doubly-linked list implementation of the `List` and `Deque` interfaces
- Implements all optional list operations
- Permits all elements, including `null`
- Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

class LinkedList<E>

```
- int size
- Node<E> first
- Node<E> last
```

Additional fields not shown.

```
+ void add(int i, E e)
+ boolean add(E e)
+ boolean addAll(int i, Collection<E> c)
+ boolean addAll(Collection<E> c)
+ boolean contains(Object o)
+ boolean containsAll(Collection<E> c)
+ boolean equals(Object o)
+ boolean isEmpty()
+ int get(int i)
+ int indexOf(Object o)
+ Iterator<E> iterator()
+ int lastIndexOf(Object o)
+ ListIterator<E> listIterator()
+ ListIterator<E> listIterator(int i)
+ E remove(int i)
+ boolean remove(Object o)
+ boolean removeAll(Collection<E> c)
+ E set(int i, E e)
+ int size()
```

Additional methods not shown.

void add(int i, E e)

Precondition: Parameter `i` is a valid index position within this `List`.
Parameter `e` may be `null`.

Postcondition: Inserts parameter `e` into this `List` at index position `i`, shifting the relative positions of all subsequent elements to the next higher index positions. The size of this `List` is increased by 1.

Pseudocode: Compare `i` with half of the size of this `LinkedList` to determine whether the `first` or `last` is closer to `i`.
Starting at the closer end of the list, traverse to the `Node` at position `i`.
Insert a new `Node` at index position `i` and assign the value of parameter `e` into the `Node`.
Increment `size` by 1.

boolean add(E e)

Precondition: Parameter `e` may be `null`.

Postcondition: Appends parameter `e` to the end of this `List` and the size of this `List` is increased by 1.

Pseudocode: Insert a new `Node` at `last` and assign the value of parameter `e` into the `Node`.
Increment `size` by 1.

boolean addAll(int i, Collection<E> c)

Precondition: Parameter `i` is a valid index position within this `List`.
Parameter `c` may contain `null` elements.

Postcondition: Inserts all elements of parameter `c` into this `List` at index position `i`, shifting the relative positions of all subsequent elements to higher index positions such that lie beyond the last element of parameter `c`. The size of this `List` is increased by the number of elements added.

Pseudocode: Compare `i` with half of the size of this `LinkedList` to determine whether the `first` or `last` is closer to `i`.
Starting at the closer end of the list, traverse to the `Node` at position `i`.
For each element of parameter `c`, insert a new `Node` at index position `i`, assign the element into the `Node`, and increment `i`.
Increment `size` by the number of elements in parameter `c`.

boolean equals(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns true if parameter `o` is a `List`, has the same size as this `List`, and contains all of the same elements in the same order as this `List`. Otherwise, returns `false`.

Pseudocode: If parameter `o` is `null`, return `false`.
If parameter `o` is a reference to this `LinkedList`, return `true`.
If parameter `o` is not a `List`, return `false`.
If parameter `o` does not have the same size as this `LinkedList`, return `false`.
Declare a `List<E>` called `that` and assign to it the value of parameter `o` after typecasting it to a `List<E>`.
Traverse through each `Node` of `this` list and `that` list...
    ...If the `Node` from `this` list and the `Node` from `that` list do not contain equivalent elements (i.e., using `equals()`)...
        ...return `false`.
Return `true`.

int get(int i)

Precondition: Parameter `i` is a valid index position within this `List`.

Postcondition: Returns the element at index position `i`.

Pseudocode: Compare `i` with half of the size of this `LinkedList` to determine whether the `first` or `last` is closer to `i`.
Starting at the closer end of the list, traverse to the `Node` at position `i`.
Return the value stored in the `Node` at position `i`.

int indexOf(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns the index position of the first occurrence in this `List` of an element equivalent to parameter `o` or -1 if no such element is contained within this `List`.

Pseudocode: Starting at the `Node` referenced by `first`, traverse backwards through each `Node` of this `LinkedList`...
    ...If the element stored in the `Node` is equivalent to parameter `o` (i.e., using `equals()`)...
        ...return the relative index position of the `Node`.
Return -1.

int lastIndexOf(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns the index position of the last occurrence in this `List` of an element equivalent to parameter `o` or -1 if no such element is contained within this `List`.

Pseudocode:
Starting at the `Node` referenced by `last`, traverse backwards through each `Node` of this `LinkedList`...
    ...If the element stored in the `Node` is equivalent to parameter `o` (i.e., using `equals()`)...
        ...return the relative index position of the `Node`.
Return -1.

ListIterator<E> listIterator()

Precondition: `LinkedList` contains an inner class called `ListItr` that implements the `ListIterator` interface.

Postcondition: Returns a `ListIterator` that can traverse through each element of this `List` in either ascending or descending order of index position.

Pseudocode: Return a new instance of the `LinkedList` class' custom `ListItr` inner class.

ListIterator<E> listIterator(int i)

Precondition: Parameter `i` is a valid index position within this `List`.
`LinkedList` contains an inner class called `ListItr` that implements the `ListIterator` interface.

Postcondition: Returns a `ListIterator` that can traverse through each element of this `List` in either ascending or descending order of index position and is positioned to start at the index position specified by parameter `i`.

Pseudocode: Return a new instance of the `LinkedList` class' custom `ListItr` inner class initialized to start at index `i`.

E remove(int i)

Precondition: Parameter `i` is a valid index position within this `List`.

Postcondition: The element at index position `i` has been removed and returned and the size of this `List` is reduced by 1.

Pseudocode:
Compare `i` with half of the size of this `LinkedList` to determine whether the `first` or `last` is closer to `i`.
Starting at the closer end of the list, traverse to the `Node` at position `i`.
Remove the `Node` at index position `i`.
Decrement `size` by 1.
Return the value stored in the removed `Node`.

E set(int i, E e)

Precondition: Parameter `i` is a valid index position within this `List`.
Parameter `e` may be `null`.

Postcondition: The element at index position `i` has been replaced by parameter `e` and the replaced element is returned.

Pseudocode:
Compare `i` with half of the size of this `LinkedList` to determine whether the `first` or `last` is closer to `i`.
Starting at the closer end of the list, traverse to the `Node` at position `i`.
Store parameter `e` in the `Node`.
Return the element that was originally stored in the `Node`.

NOTE: This documentation is not exhaustive. Additional methods are not shown.

Stack<E>

Description:

- A collection designed for holding elements prior to processing
- Represents a last-in/first-out (LIFO) stack of objects
- The user can `push()` an element onto the top of the stack
- The user can `pop()` an element from the top of the stack
- The user can `peek()` at the top item on top of the stack without removing it
- The user can test for whether the stack `isEmpty()`

class Stack<E>

```
- int elementCount*
- Object[] elementData*

+ boolean contains(Object o)*
+ boolean equals(Object o)*
+ boolean isEmpty()*
+ E peek()
+ E pop()
+ E push(E e)
+ int size()*

Additional methods not shown.
```

** Method/field inherited from `Vector<E>` class.*

boolean equals(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns true if parameter `o` is a `Stack`, has the same size as this `Stack`, and contains all of the same elements in the same order as this `Stack`. Otherwise, returns `false`.

Pseudocode:
If parameter `o` is `null`, return `false`.
If parameter `o` is a reference to this `Stack`, return `true`.
If parameter `o` is not a `List`, return `false`.
If parameter `o` does not have the same size as this `Stack`, return `false`.
Declare a `Stack<E>` called `that` and assign to it the value of parameter `o` after typecasting it to a `Stack<E>`.
For each index position `i` in `elementData`...
    ...If the element at position `i` from `this.elementData` and the element from position `i` from `that.elementData` do not contain equivalent elements (i.e., using `equals()`)...
        ...return `false`.
Return `true`.

boolean isEmpty()

Precondition: This `Stack` may be empty.

Postcondition: Returns `true` if this `Stack` is empty. Otherwise, returns `false`.

Pseudocode: Return the result of comparing `elementCount` with 0.

E peek()

Precondition: This `Stack` may be empty.

Postcondition: If this `Stack` is empty, throws an `EmptyStackException`. Otherwise, returns the element at the top of this `Stack` without removing it from the `Stack`.

Pseudocode: If this `Stack` is empty...
    ...throw a new `EmptyStackException`.
Return a reference to the element at index position `size - 1` of `elementData`.

E pop()

Precondition: This `Stack` may be empty.

Postcondition: If this `Stack` is empty, throws an `EmptyStackException`. Otherwise, removes and returns the element at the top of this `Stack` and reduces the size of this `Stack` by 1.

Pseudocode:
If this `Stack` is empty...
    ...throw a new `EmptyStackException`.
Decrement the value of `elementCount`.
Return a reference to the element at index position `size - 1` of `elementData`.

E push(E e)

Precondition: Parameter `e` may be `null`.

Postcondition: Adds parameter `e` to the top of this `Stack` and returns the added element and increases the size of this `Stack` by 1.

Pseudocode:
Assign parameter `e` to index position `elementCount` of `elementData`.
Increment the value of `elementCount`.
Return parameter `e`.

int size()

Precondition: This `Stack` contains 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: Returns the number of elements currently contained within this `Stack`.

Pseudocode: Return the value of `elementCount`.

NOTE: This documentation is not exhaustive. Additional methods are not shown.

Queue<E>

Description:

- A collection designed for holding elements prior to processing
- Represents a first-in/first-out (FIFO) stack of objects
- The user can `add()` an element to the tail of the queue
- The user can `remove()` an element from the head of the queue
- The user can `peek()` at the next element to be removed from the queue without removing it
- The user can test for whether the queue `isEmpty()`

Implementing Classes:

- **`LinkedList<E>`**
  - Prioritizes elements by the order in which they are added to the queue
- **`PriorityQueue<E>`**
  - Prioritizes elements based on their relative values
  - Elements must have a "natural ordering" (i.e., implements the `Comparable<E>` interface)

interface Queue<E>

Interfaces contain no instance variables.

```
+ boolean add(E e)*
+ boolean contains(Object o)*
+ boolean equals(Object o)*
+ boolean isEmpty()*
+ E peek()
+ E remove()
+ int size()*

Additional methods not shown.
```

** Method inherited from `Collection<E>` interface.*

boolean add(E e)

Precondition: Parameter `e` may be `null`.

Postcondition: Appends parameter `e` to the end of this `Queue`, increases the size of this `Queue` by 1, and returns `true`.

boolean equals(Object o)

Precondition: This `Queue` may be empty.

Postcondition: Returns true if parameter `o` is a `Queue`, has the same size as this `Queue`, and contains all of the same elements in the same order as this `Queue`. Otherwise, returns `false`.

E peek()

Precondition: This `Queue` may be empty.

Postcondition: If this `Queue` is empty, returns `null`. Otherwise, returns the element at the front of this `Queue` without removing it from the `Queue`.

E remove()

Precondition: This `Queue` may be empty.

Postcondition: If this `Queue` is empty, throws a `NoSuchElementException`. Otherwise, removes and returns the element at the front of this `Queue` and reduces the size of this `Queue` by 1.

NOTE: This documentation is not exhaustive. Additional methods are not shown.

PriorityQueue<E>

Description:

- Min-heap implementation of the `Queue` interface
  - Stored elements are sorted vertically within a binary heap structure
- Elements are ordered by either:
  - The natural ordering of the element type
  - A `Comparator` provided to the constructor
- Elements cannot be `null`
- Elements must be comparable to one another (i.e., implements the `Comparable<E>` interface)
- The head of this queue is the element with the lowest relative value, as specified by the natural ordering of the elements

class PriorityQueue<E>

```
- int size
- Object[] queue

Additional fields not shown.
```

```
- void grow(int minCapacity)
+ boolean add(E e)
+ boolean contains(Object o)
+ boolean isEmpty()*
+ E peek()
+ E remove()
+ int size()

Additional methods not shown.
```

** Method inherited from `AbstractQueue<E>`.*

boolean add(E e)

Precondition: Parameter `e` is not `null`.

Postcondition: Adds parameter `e` to this `PriorityQueue`, increases the size of this `PriorityQueue` by 1, and returns `true`.

Pseudocode:
Increment the value of `size`.
Assign to index position `size` of `queue` the value of parameter `e`.
Declare an integer called i and assign to it the value of `size`.
While `i` is greater than 1 and the value at index position `i` is less than the value at index position `i / 2`...
    ...swap the values at index positions `i` and `i / 2`.
Return `true`.

boolean contains(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns `true` if parameter `o` is contained within this `PriorityQueue`. Otherwise, returns `false`.

Pseudocode:
Declare an integer variable called `i` and assign to it the value of 1.
While `i` is less than or equal to `size`...
    ...if the value at index position `i` is equivalent to parameter `o` (i.e., using `equals()`)...
        ...return `true`.
    ...if the value at index position `i` is less than the value at index position `i * 2`...
        ...assign to `i` the the value `i * 2`.
    ...otherwise...
        ...assign to `i` the the value `(i * 2) + 1`.
Return false.

boolean isEmpty()

Precondition: This `PriorityQueue` may be empty.

Postcondition: Returns `true` if this `PriorityQueue` is empty.

Pseudocode: Return the result of comparing `size` with 0.

E peek()

Precondition: This `PriorityQueue` may be empty.

Postcondition: If this `PriorityQueue` is empty, returns `null`. Otherwise, returns the lowest valued element in this `PriorityQueue` without removing it from the `PriorityQueue`.

Pseudocode:
If this `PriorityQueue` is empty...
    ...return `null`.
Returns the element in index position 1 of `queue`.

E remove()

Precondition: This `PriorityQueue` may be empty.

Postcondition: If this `PriorityQueue` is empty, throws a `NoSuchElementException`. Otherwise, removes and returns the lowest valued element in this `PriorityQueue` and reduces the size of this `PriorityQueue` by 1.

Pseudocode:
Declare an `E` variable called `head` and assign to it the value in index position 1 of `queue`.
Assign to index position 1 of `queue` the value stored in index position `size` of `queue`.
Assign to index position `size` of `queue` the value of `null`.
Declare an integer variable called `i` and assign to it the value of 1.
Declare an `E` variable called `root` and assign to it the value at index position `i`.
Declare an `E` variable called `left` and assign to it the value at index position `i * 2`.
Declare an `E` variable called `right` and assign to it the value at index position `i * 2 + 1`.
While `i` is less than `size` and `root` is less than either `left` or `right`...
    ...declare an `E` variable called `left` and assign to it the value at index position `i * 2`.
    ...declare an `E` variable called `right` and assign to it the value at index position `i * 2 + 1`.
    ...if `right` is equal to `null` or `left` is less than `right`...
        ...swap the values at index positions `i` and `i * 2`.
        ...assign to `i` the value of `i * 2`.
        ...assign to `root` the value of `left`.
    ...otherwise...
        ...swap the values at index positions `i` and `i * 2 + 1`.
        ...assign to `i` the value of `i * 2 + 1`.
        ...assign to `root` the value of `right`.
    ...increment the value of `i`.
Decrement the value of `size`.
Return the value of `head`.

int size()

Precondition: This `PriorityQueue` contains 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: Returns the number of elements currently contained within this `PriorityQueue`.

Pseudocode: Return the value of `size`.

NOTE: This documentation is not exhaustive. Additional methods are not shown.

Set<E>

Description:

- Models the mathematical set abstraction.
- A collection that contains no duplicate elements
  - More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one `null` element
- May be ordered or unordered, depending upon the implementation

Implementing Classes:

- **`HashSet<E>`**
  - Array-based implementation that uses a hash table to store elements based on their hash values
  - Maintains elements in an unspecified order
  - Storage and retrieval of elements relies on each element having a sufficiently defined `hashCode()` and `equals()` methods
- **`TreeSet<E>`**
  - Link-based implementation that arranges elements in a balanced, binary search tree
  - Maintains elements in a sorted order
  - Elements must have a "natural ordering" (i.e., implements the `Comparable<E>` interface)

interface Set<E>

Interfaces contain no instance variables.

```
+ boolean add(E e)*
+ boolean addAll(Collection<E> c)*
+ boolean contains(Object o)*
+ boolean containsAll(Collection<E> c)*
+ boolean equals(Object o)*
+ boolean isEmpty()*
+ Iterator<E> iterator()*
+ boolean remove(Object o)*
+ boolean removeAll(Collection<E> c)*
+ int size()*
```

Additional methods not shown.

** Method inherited from `Collection<E>` interface.*

boolean add(E e)

Precondition: Parameter `e` may be `null`.

Postcondition: If this `Set` does not contain parameter `e`, it is added to the `Set` and the size of the `Set` is increased by 1. Returns `true` if the size of the set was modified. Otherwise, returns `false`.

boolean addAll(Collection<E> c)

Precondition: Parameter `c` may contain `null` elements.

Postcondition: For each element in parameter `c`, if this `Set` does not contain the element, it is added to the `Set` and the size of the `Set` is increased by 1 for each added element.
Returns `true` if the size of the set was modified. Otherwise, returns `false`.

NOTE: This documentation is not exhaustive. Additional methods are not shown.

HashSet<E>

Description:

- Hash table implementation of the `Set` interface
  - Internally uses a `HashMap` instance
  - Elements are stored/accessed using the "bucket and chain" approach
  - Optimal performance is maintained through proper management of the table's capacity vs. the load factor
  - Makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time
    - Capacity is the number of buckets in the hash table
    - Load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased
    - Table is rehashed when the number of entries in the hash table exceeds the product of the load factor and the current capacity (i.e., the number of buckets is approximately doubled)
- Permits all elements, including `null`
- NOTE: Because `TreeSet` and `HashSet` are both backed by `Map` objects as their underlying data structures, most methods between the two classes are implemented identically to one another. The difference in runtime performance of the two `Set` classes comes from the different runtime performances of their underlying `Map` data structures.

class HashSet<E>

```
- HashMap<E,Object> map;

Additional fields not shown.

+ boolean add(E e)
+ boolean addAll(Collection<E> c)
+ boolean contains(Object o)
+ boolean containsAll(Collection<E> c)
+ boolean equals(Object o)
+ boolean isEmpty()
+ Iterator<E> iterator()
+ boolean remove(Object o)
+ boolean removeAll(Collection<E> c)
+ int size()

Additional methods not shown.
```

** Method inherited from `AbstractSet<E>`.*

boolean add(E e)

Precondition: Parameter `e` may be `null`.

Postcondition: If this `HashSet` does not contain parameter `e`, it is added to the `HashSet` and the size of the `HashSet` is increased by 1.
Returns `true` if the size of the set was modified. Otherwise, returns `false`.

Pseudocode: Put a new key-value pair into `map` with parameter `e` as the key and a default `Object` as the value and return `true` if the result is `null`. Otherwise, return `false`.

boolean contains(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns `true` if parameter `o` is contained within this `HashSet`. Otherwise, returns `false`.

Pseudocode: Return the result of testing whether `map` contains parameter `o` as a key.

boolean equals(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns true if parameter `o` is a `Set`, has the same size as this `HashSet`, and contains all of the same elements as this `HashSet`. Otherwise, returns `false`.

Pseudocode:
If parameter `o` is `null`, return `false`.
If parameter `o` is a reference to this `HashSet`, return `true`.
If parameter `o` is not a `Set`, return `false`.
If parameter `o` does not have the same size as this `HashSet`, return `false`.
Declare a `Set<E>` called `that` and assign to it the value of parameter `o` after typecasting it to a `Set<E>`.
Return the result of testing whether `map` contains all elements of `that`.

boolean remove(Object o)

Precondition: This `HashSet` contains 0 or more elements of some data type (`E`) to be specified at runtime.
Parameter `o` may be `null`.

Postcondition: If this `HashSet` contains an element equivalent to parameter `o` (i.e., using `equals()`), one matching element is removed from this `HashSet` and the size of `HashSet` size is reduced by 1.
Returns `true` if the size of this `HashSet` is changed. Otherwise, returns `false`.

Pseudocode: Remove the key-value pair from `map` that uses parameter `e` as the key and return `false` if the result is `null`. Otherwise, return `true`.

int size()

Precondition: This `HashSet` contains 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: Returns the number of elements currently contained within this `HashSet`.

Pseudocode: Return the size of `map` (i.e., the number of key-value pairs in the underlying map).

NOTE: This documentation is not exhaustive. Additional methods are not shown.

TreeSet<E>

Description:

- Binary search tree implementation of the `Set` interface
  - Internally uses a `TreeMap` instance
- Elements are ordered by either:
  - The natural ordering of the element type
  - A `Comparator` provided to the constructor
- NOTE: Because `TreeSet` and `HashSet` are both backed by `Map` objects as their underlying data structures, most methods between the two classes are implemented identically to one another. The difference in runtime performance of the two `Set` classes comes from the different runtime performances of their underlying `Map` data structures.

class TreeSet<E>

```
- NavigableMap<E,Object> map

Additional fields not shown.

+ boolean add(E e)
+ boolean addAll(Collection<E> c)
+ boolean contains(Object o)
+ boolean containsAll(Collection<E> c)
+ boolean equals(Object o)
+ boolean isEmpty()
+ Iterator<E> iterator()
+ boolean remove(Object o)
+ boolean removeAll(Collection<E> c)
+ int size()

Additional methods not shown.
```

** Method inherited from `NavigableSet<E>`.*

boolean add(E e)

Precondition: Parameter `e` may be `null`.

Postcondition: If this `TreeSet` does not contain parameter `e`, it is added to the `TreeSet` and the size of the `TreeSet` is increased by 1.
Returns `true` if the size of the set was modified. Otherwise, returns `false`.

Pseudocode: Put a new key-value pair into `map` with parameter `e` as the key and a default `Object` as the value and return `true` if the result is `null`. Otherwise, return `false`.

boolean contains(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns `true` if parameter `o` is contained within this `TreeSet`. Otherwise, returns `false`.

Pseudocode: Return the result of testing whether `map` contains parameter `o` as a key.

boolean equals(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns true if parameter `o` is a `Set`, has the same size as this `TreeSet`, and contains all of the same elements as this `TreeSet`. Otherwise, returns `false`.

Pseudocode: If parameter `o` is `null`, return `false`.
If parameter `o` is a reference to this `TreeSet`, return `true`.
If parameter `o` is not a `Set`, return `false`.
If parameter `o` does not have the same size as this `TreeSet`, return `false`.
Declare a `Set<E>` called `that` and assign to it the value of parameter `o` after typecasting it to a `Set<E>`.
Return the result of testing whether `map` contains all elements of `that`.

`boolean remove(Object o)`

Precondition: This `TreeSet` contains 0 or more elements of some data type (`E`) to be specified at runtime.
Parameter `o` may be `null`.

Postcondition: If this `TreeSet` contains an element equivalent to parameter `o` (i.e., using `equals()`), one matching element is removed from this `TreeSet` and the size of `TreeSet` size is reduced by 1.
Returns `true` if the size of this `TreeSet` is changed. Otherwise, returns `false`.

Pseudocode: Remove the key-value pair from `map` that uses parameter `e` as the key and return `false` if the result is `null`. Otherwise, return `true`.

`int size()`

Precondition: This `TreeSet` contains 0 or more elements of some data type (`E`) to be specified at runtime.

Postcondition: Returns the number of elements currently contained within this `TreeSet`.

Pseudocode: Return the size of `map` (i.e., the number of key-value pairs in the underlying map).

NOTE: This documentation is not exhaustive. Additional methods are not shown.

Map<K,V>

Description:

- Associates keys with values
- Cannot contain duplicate keys
- Each key can map to at most one value
- Allows a map's contents to be viewed as:
  - A set of keys
  - A collection of values
  - Set of key-value mappings
- Keys may be ordered or unordered, depending upon the implementation
- NOTE: `Map<K,V>` is closely related to other types of collections (in particular, sets), but does not actually extend from the `Collection<E>` interface due to differences in available methods and method signatures (i.e., Maps are defined generically in terms of 2 types: its *key* and its *value*)

Implementing Classes:

- **`HashMap<K,V>`**
  - Array-based implementation that uses a hash table to store elements based on their hash values
  - Maintains key-value pairs in an unspecified order, as determined by the key
  - Storage and retrieval of elements relies on each element having a sufficiently defined `hashCode()` and `equals()` methods
- **`TreeMap<K,V>`**
  - Link-based implementation that arranges elements in a balanced, binary search tree
  - Maintains key-value pairs in a sorted order, as determined by the key
  - Elements must have a "natural ordering" (i.e., implements the `Comparable<E>` interface)

interface Map<K,V>

Interfaces contain no instance variables.

```
+ boolean containsKey(Object k)
+ boolean containsValue(Object v)
+ boolean equals(Object o)
+ V get(Object k)
+ boolean isEmpty()
+ Set<K> keySet()
+ V put(K k, V v)
+ void putAll(Map<K, V> m)
+ V remove(Object k)
+ int size()
+ Collection<V> values()
```

Additional methods not shown.

boolean containsKey(Object k)

Precondition: Parameter `o` may be of any object data type and is not `null`.

Postcondition: Returns `true` if this `Map` contains a key-value pair whose key is equivalent to parameter `k` (i.e., `using equals()`). Otherwise, returns `false`.

boolean containsValue(Object v)

Precondition: Parameter `o` may be of any object data type and is not `null`.

Postcondition: Returns `true` if this `Map` contains a key-value pair whose value is equivalent to parameter `k` (i.e., `using equals()`). Otherwise, returns `false`.

boolean equals(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns true if parameter `o` is a `Map`, has the same size as this `Map`, and contains all of the same key-value pairs as this `Map`. Otherwise, returns `false`.

V get(Object k)

Precondition: Parameter `o` may be of any object data type and is not `null`.

Postcondition: Returns the value associated with the key specified by parameter `k`, if such a key-value pair exists within this `Map`. Otherwise, returns `null`.

Set<K> keySet()

Precondition: This `Map` contains 0 or more key-value pairs.
No 2 key-value pairs share the same key.

Postcondition: Returns the `Set` of all keys contained with this `Map`.
NOTE: The returned `Set` is "backed" by this `Map`. Any changes to the key-value pairs in this `Map` will automatically be reflected in the returned `Set`.

V put(K k, V v)

Precondition: Parameter `k` is not `null`.
Parameter `v` may be `null`.

Postcondition: Adds or updates a key-value pair to this `Map` that associates parameter `k` (i.e., the key) with parameter `v` (i.e., the value).
Returns the value previously associated with the key specified by parameter `k`, if such a key-value pair already existed with this `Map`. Otherwise, returns `null`.

V remove(Object k)

Precondition: Parameter `o` may be of any object data type and is not `null`.

Postcondition: Removes the key-value pair specified by parameter `k` (i.e., the key) from this `Map`.
Returns the value previously associated with the key specified by parameter `k`, if such a key-value pair already existed with this `Map`. Otherwise, returns `null`.

int size()

Precondition: This `Map` contains 0 or more key-value pairs.

Postcondition: Returns the number of key-value pairs currently contained within this `Map`.

Collection<V> values()

Precondition: This `Map` contains 0 or more key-value pairs.
Multiple key-value pairs may share the same value.

Postcondition: Returns the `Collection` of all values contained with this `Map`.
NOTE: The returned `Collection` is "backed" by this `Map`. Any changes to the key-value pairs in this `Map` will automatically be reflected in the returned `Collection`.

NOTE: This documentation is not exhaustive. Additional methods are not shown.

HashMap<K,V>

Description:

- Hash table implementation of the `Map` interface
  - Keys are stored/accessed using the "bucket and chain" approach
    - For 8 or fewer items in a bucket, key-value pairs are stored in a linked list of `Node` objects
    - For more than 8 items in a bucket, key-value pairs are stored in a binary search tree of `Node` objects
  - Optimal performance is maintained through proper management of the table's capacity vs. the load factor
  - Makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.
    - Capacity is the number of buckets in the hash table
    - Load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased
    - Table is rehashed when the number of entries in the hash table exceeds the product of the load factor and the current capacity (i.e., the number of buckets is approximately doubled)
- Permits all keys, including `null`
- Permits all values, including `null`
- Each key-value pairing is associated via a custom `Node<K,V>` object defined as a private inner class of `HashMap`
- NOTE: As a general rule, the default load factor (0.75) offers a good tradeoff between time and space costs
  - Higher values decrease the space overhead but increase the lookup cost
  - Lower values increase the likelihood of rehash operations

class HashMap<K,V>

```
- Node<K,V>[] table
- Set<Map.Entry<K,V>> entrySet
- Set<K> keySet*
- int size
- float loadFactor

Additional fields not shown.
```

```
+ boolean containsKey(Object k)
+ boolean containsValue(Object v)
+ boolean equals(Object o)
+ V get(Object k)
+ boolean isEmpty()
+ Set<K> keySet()
+ V put(K k, V v)
+ void putAll(Map<K, V> m)
+ V remove(Object k)
+ int size()
+ Collection<V> values()

Additional methods not shown.
```

** Method inherited from `AbstractMap<E>`.*

class HashMap.Node<K,V>

```
- K key
- V value
- int hash
- Node<K,V> next
```

```
+ K getKey()
+ V getValue()
+ V setValue(V newValue)

Additional methods not shown.
```

boolean containsKey(Object k)

Precondition: Parameter `o` may be of any object data type and is not `null`.

Postcondition: Returns `true` if this `HashMap` contains a key-value pair whose key is equivalent to parameter `k` (i.e., `using equals()`). Otherwise, returns `false`.

Pseudocode: Bitwise AND the hash value of `k` with the length of `table` to compute the `index` of the appropriate "bucket".
For each `Node` object in `table[index]`...
...if the `key` stored in the `Node` is equivalent to `k`...
...Return `true`.
Return `false`.

boolean containsValue(Object v)

Precondition: Parameter `o` may be of any object data type and is not `null`.

Postcondition: Returns `true` if this `HashMap` contains a key-value pair whose value is equivalent to parameter `k` (i.e., `using equals()`). Otherwise, returns `false`.

Pseudocode:
For each `index` position in `table`...
    ...for each `Node` object in `table[index]`...
        ...if the `value` stored in the `Node` is equivalent to `v`...
            ...Return `true`.
Return `false`.

boolean equals(Object o)

Precondition: Parameter `o` may be of any object data type and may be `null`.

Postcondition: Returns true if parameter `o` is a `HashMap`, has the same size as this `HashMap`, and contains all of the same key-value pairs as this `HashMap`. Otherwise, returns `false`.

Pseudocode:
If parameter `o` is `null`, return `false`.
If parameter `o` is a reference to this `HashMap`, return `true`.
If parameter `o` is not a `Map`, return `false`.
If parameter `o` does not have the same size as this `HashMap`, return `false`.
Declare a `Map<E>` called `that` and assign to it the value of parameter `o` after typecasting it to a `Map<E>`.
For each `index` position in `table`...
    ...for each `Node` object in `table[index]`...
        ...if `that` contains the key-value pair stored within the `Node`...
            ...return `true`.
Return `false`.

V get(Object k)

Precondition: Parameter `o` may be of any object data type and is not `null`.

Postcondition: Returns the value associated with the key specified by parameter `k`, if such a key-value pair exists within this `HashMap`. Otherwise, returns `null`.

Pseudocode:
Bitwise AND the hash value of `k` with the length of `table` to compute the `index` of the appropriate "bucket".
For each `Node` object in `table[index]`...
    ...if the `key` stored in the `Node` is equivalent to `k`...
        ...return the `value` stored in the `Node`.
Return `null`.

Set<K> keySet()

Precondition: This `HashMap` contains 0 or more key-value pairs.
No 2 key-value pairs share the same key.

Postcondition: Returns the `Set` of all keys contained with this `HashMap`.
NOTE: The returned `Set` is "backed" by this `HashMap`. Any changes to the key-value pairs in this `HashMap` will automatically be reflected in the returned `Set`.

Pseudocode:
If `keySet` is `null`...
    ...create a new `KeySet<K>` (private inner class that extends `AbstractSet<K>`) that can utilize the contents of `table` to implement the functionality of a `Set` of all keys stored within this `Map`.
Return `keySet`.

V put(K k, V v)

Precondition: Parameter `k` is not `null`.
Parameter `v` may be `null`.

Postcondition: Adds or updates a key-value pair to this `HashMap` that associates parameter `k` (i.e., the key) with parameter `v` (i.e., the value).
Returns the value previously associated with the key specified by parameter `k`, if such a key-value pair already existed with this `HashMap`. Otherwise, returns `null`.

Pseudocode:
Bitwise AND the hash value of `k` with the length of `table` to compute the `index` of the appropriate "bucket".
For each `Node` object in `table[index]`...
    ...if the `key` stored in the `Node` is equivalent to `k`...
        ...replace the `value` stored in the `Node` with parameter `v`.
        ...return the `value` that was originally stored in the `Node`.
Create a new `Node` with a key of `k`, a value of `v`, a `hash` of the hash value of `k`, and a `next` pointer of `null`.
If the number of `Node` objects stored in `table[index]` is less than `8`...
    ...append the newly created `Node` to the end of the linked chain of `Node` objects.
Otherwise if the number of `Node` objects stored in `table[index]` is equal to `8`...
    ...construct a binary search tree of `Node` objects from each of the nodes in the linked list of `Node` objects.
    ...replace the linked list of `Node` objects in `table[index]` with the binary search tree of `Node` objects.
Otherwise...
    ...insert the newly created `Node` into the balanced, binary search tree of `Node` objects.
    ...rebalance the tree, if necessary.
Increment `size`.
Return `null`.

V remove(Object k)

Precondition: Parameter `o` may be of any object data type and is not `null`.

Postcondition: Removes the key-value pair specified by parameter `k` (i.e., the key) from this `HashMap`.
Returns the value previously associated with the key specified by parameter `k`, if such a key-value pair already existed with this `HashMap`. Otherwise, returns `null`.

Pseudocode:
Bitwise AND the hash value of `k` with the length of `table` to compute the `index` of the appropriate "bucket".
For each `Node` object in `table[index]`...
    ...if the `key` stored in the `Node` is equivalent to `k`...
        ...remove the `Node` from the linked list or binary search tree of `Node` objects.
        ...decrement `size` by 1.
        ...return `value` stored in the removed `Node`.
Return `null`.

int size()

Precondition: This `HashMap` contains 0 or more key-value pairs.

Postcondition: Returns the number of key-value pairs currently contained within this `HashMap`.

Pseudocode: Return the value of `size`.

NOTE: This documentation is not exhaustive. Additional methods are not shown.

TreeMap<K,V>

Description:

- Binary search tree implementation of the `Map` interface
- Keys are ordered by either:
  - The natural ordering of the element type
  - A `Comparator` provided to the constructor
- The set of keys will be iterated over in ascending order

class TreeMap<K,V>

```
- Entry<K,V> root
- NavigableSet<K> navigableKeySet
- int size
```

Additional fields not shown.

```
+ boolean containsKey(Object k)
+ boolean containsValue(Object v)
+ boolean equals(Object o)
+ V get(Object k)
+ boolean isEmpty()
+ Set<K> keySet()
+ V put(K k, V v)
+ void putAll(Map<K, V> m)
+ V remove(Object k)
+ int size()
+ Collection<V> values()
```

Additional methods not shown.

** Method inherited from `NavigableMap<E>`.*

class TreeMap.Entry<K,V>

```
- K key
- V value
- Entry<K,V> left
- Entry<K,V> right
- Entry<K,V> parent
```

```
+ K getKey()
+ V getValue()
+ V setValue(V newValue)
```

Additional methods not shown.

boolean containsKey(Object k)

Precondition: Parameter `o` may be of any object data type and is not `null`.

Postcondition: Returns `true` if this `TreeMap` contains a key-value pair whose key is equivalent to parameter `k` (i.e., `using equals()`). Otherwise, returns `false`.

Pseudocode:

Declare an `Entry<K,V>` variable called `node` and assign to it a reference to `root`.
While `node` is not `null`...
    ...if the `key` stored in `node` is equivalent to parameter `k`...
        ...return `true`.
    ...if the `key` stored in `node` is greater than parameter `k`...
        ...reassign to `node` a reference to `node.left`.
    ...otherwise...
        ...reassign to `node` a reference to `node.right`.
Return `false`.

boolean equals(Object o)

Precondition: Parameter o may be of any object data type and may be null.

Postcondition: Returns true if parameter o is a TreeMap, has the same size as this TreeMap, and contains all of the same key-value pairs as this TreeMap. Otherwise, returns false.

Pseudocode:
If parameter o is null, return false.
If parameter o is a reference to this TreeMap, return true.
If parameter o is not a Map, return false.
If parameter o does not have the same size as this TreeMap, return false.
Declare a Map<E> called that and assign to it the value of parameter o after typecasting it to a Map<E>.
Declare a Stack<Entry<K,V>> variable called nodes and initialize it to an empty stack.
Push a reference to root onto nodes.
While nodes is not empty...
    ...Declare an Entry<K,V> variable called node and assign to it the Entry object popped from nodes.
    ...if that does not contain the key stored in node or the value associated with the key in this is not equivalent to the value associated with the key in that...
        ...return false.
    ...if node.left is not null...
        ...push onto nodes a reference to node.left.
    ...if node.right is not null...
        ...push onto nodes a reference to node.right.
Return true.

V get(Object k)

Precondition: Parameter o may be of any object data type and is not null.

Postcondition: Returns the value associated with the key specified by parameter k, if such a key-value pair exists within this TreeMap. Otherwise, returns null.

Pseudocode:
Declare an Entry<K,V> variable called node and assign to it a reference to root.
While node is not null...
    ...if the key stored in node is equivalent to parameter k...
        ...return the value stored in node.
    ...if the key stored in node is greater than parameter k...
        ...reassign to node a reference to node.left.
    ...otherwise...
        ...reassign to node a reference to node.right.
Return false.

Set<K> keySet()

Precondition: This TreeMap contains 0 or more key-value pairs.
No 2 key-value pairs share the same key.

Postcondition: Returns the Set of all keys contained with this TreeMap.
NOTE: The returned Set is "backed" by this TreeMap. Any changes to the key-value pairs in this TreeMap will automatically be reflected in the returned Set.

Pseudocode:
If navigableKeySet is null...
    ...create a new KeySet<K> (private inner class that extends AbstractSet<K>) that can utilize the balanced, binary search tree referenced by root to implement the functionality of a Set of all keys stored within this Map.
Return navigableKeySet.

V put(K k, V v)

Precondition: Parameter `k` is not `null`.
Parameter `v` may be `null`.

Postcondition: Adds or updates a key-value pair to this `TreeMap` that associates parameter `k` (i.e., the key) with parameter `v` (i.e., the value).
Returns the value previously associated with the key specified by parameter `k`, if such a key-value pair already existed with this `TreeMap`. Otherwise, returns `null`.

Pseudocode:
Declare an `Entry<K,V>` variable called `node` and assign to it a reference to `root`.
Declare an `Entry<K,V>` variable called `parent` and assign to it a value of `null`.
While `node` is not `null`...
    ...if the `key` stored in `node` is equivalent to parameter `k`...
        ...replace the `value` stored in `node` with `v`.
        ...return the original `value` stored in `node`.
    ...assign to `parent` the value of `node`.
    ...if the `key` stored in `node` is greater than parameter `k`...
        ...reassign to `node` a reference to `node.left`.
    ...otherwise...
        ...reassign to `node` a reference to `node.right`.
Create a new `Entry<K,V>` variable called `child` and initialize it with a `key` of `k` and a `value` of `v`.
If `k` is less than the `key` stored in `parent`...
    ...assign to `parent.left` a reference to `child`.
Otherwise...
    ...assign to `parent.right` a reference to `child`.
Rebalance the tree as necessary.
Increment `size` by `1`.
Return `null`.

V remove(Object k)

Precondition: Parameter `o` may be of any object data type and is not `null`.

Postcondition: Removes the key-value pair specified by parameter `k` (i.e., the key) from this `TreeMap`.
Returns the value previously associated with the key specified by parameter `k`, if such a key-value pair already existed with this `TreeMap`. Otherwise, returns `null`.

Pseudocode:
Declare an `Entry<K,V>` variable called `node` and assign to it a reference to `root`.
Decrement `size` by `1`.
While `node` is not `null`...
    ...if the `key` stored in `node` is equivalent to parameter `k`...
        ...promote the right-most node of the left subtree to replace `node`.
        ...rebalance the tree as necessary.
    ...if the `key` stored in `node` is greater than parameter `k`...
        ...reassign to `node` a reference to `node.left`.
    ...otherwise...
        ...reassign to `node` a reference to `node.right`.
Return `null`.

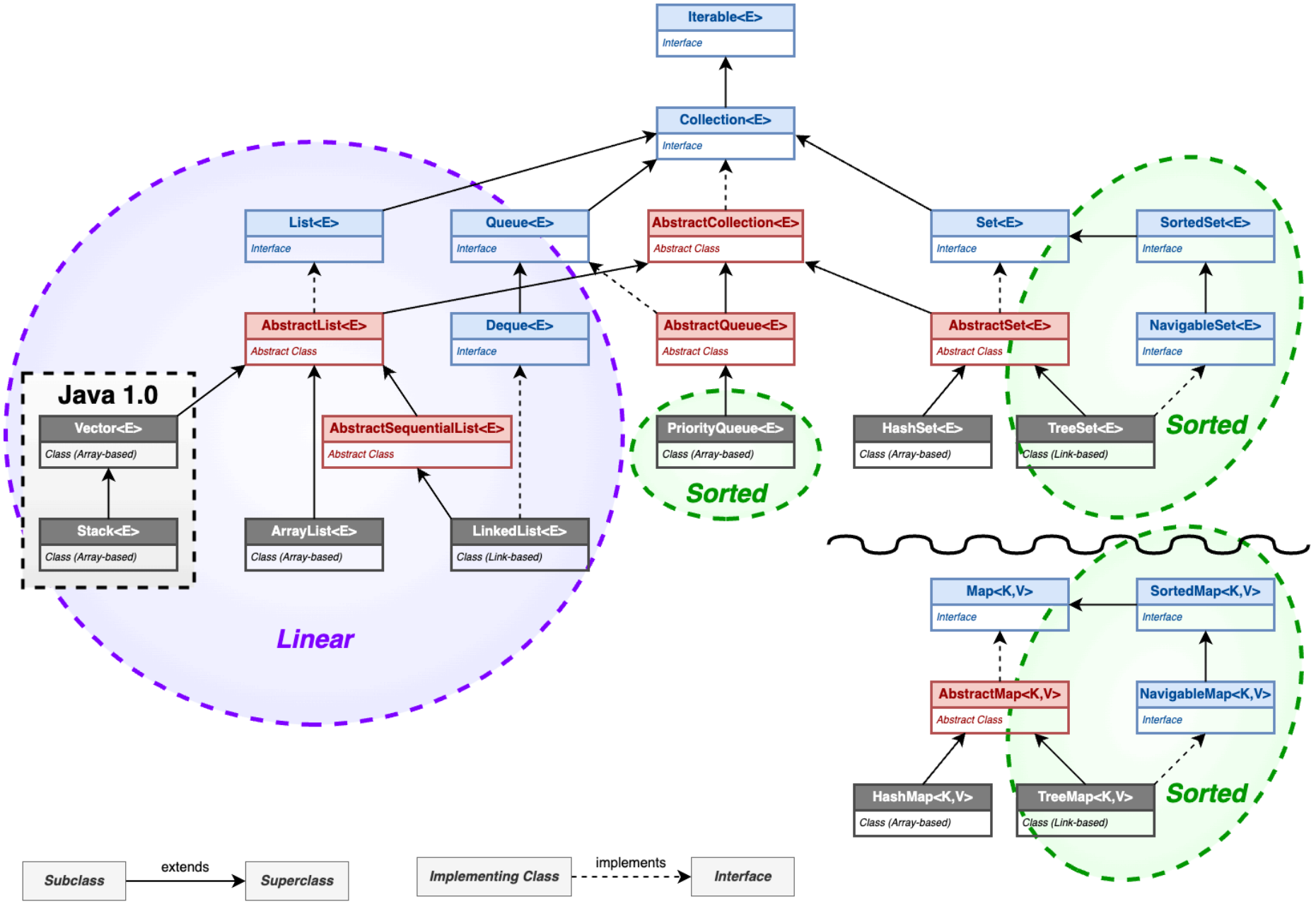
int size()

Precondition: This `TreeMap` contains 0 or more key-value pairs.

Postcondition: Returns the number of key-value pairs currently contained within this `TreeMap`.

Pseudocode: Return the value of `size`.

NOTE: This documentation is not exhaustive. Additional methods are not shown.


Iterable<E>
Interface
Collection<E>
Interface
List<E>
Interface
Queue<E>
Interface
AbstractCollection<E>
Abstract Class
Set<E>
Interface
SortedSet<E>
Interface
AbstractList<E>
Abstract Class
Deque<E>
Interface
AbstractQueue<E>
Abstract Class
AbstractSet<E>
Abstract Class
NavigableSet<E>
Interface
Java 1.0
Vector<E>
Class (Array-based)
AbstractSequentialList<E>
Abstract Class
PriorityQueue<E>
Class (Array-based)
HashSet<E>
Class (Array-based)
TreeSet<E>
Class (Link-based)
Sorted
Stack<E>
Class (Array-based)
ArrayList<E>
Class (Array-based)
LinkedList<E>
Class (Link-based)
Sorted
Linear
Map<K,V>
Interface
SortedMap<K,V>
Interface
AbstractMap<K,V>
Abstract Class
NavigableMap<K,V>
Interface
HashMap<K,V>
Class (Array-based)
TreeMap<K,V>
Class (Link-based)
Sorted
Subclass
extends
Superclass
Implementing Class
implements
Interface