CSE 373

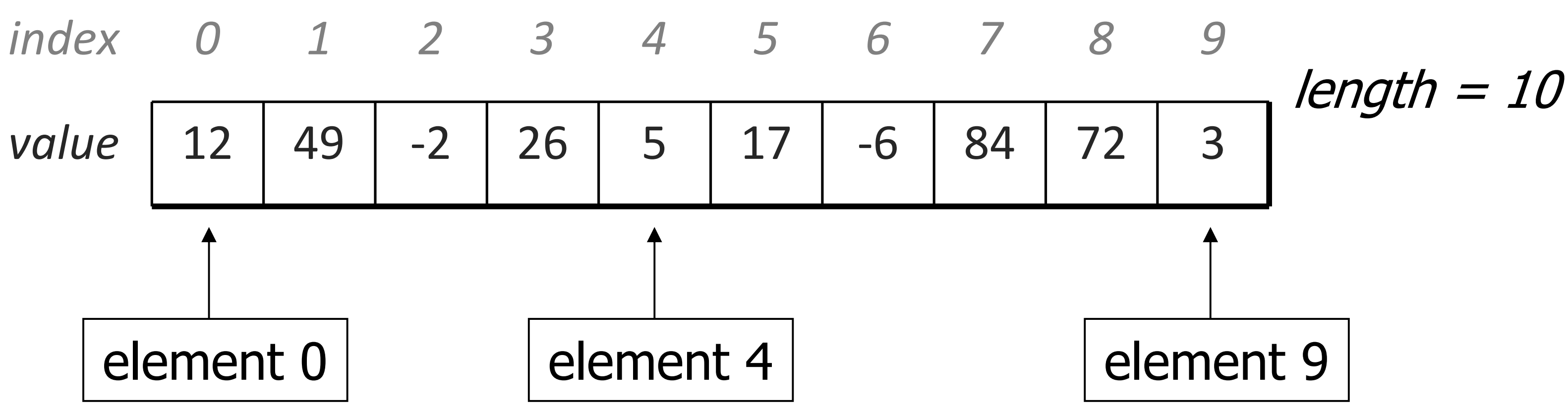
Java Collection Framework

reading: Weiss Ch. 3, 4.8

slides created by Marty Stepp
http://www.cs.washington.edu/373/

Arrays

- **array**: An object that stores many values of the same type.
  - **element**: One value in an array.
  - **index**: A 0-based integer to access an element from an array.
  - **length**: Number of elements in the array.



Array declaration

type`[]` **name** `= new` **type**`[`**length**`];`

- Length explicitly provided. All elements' values initially 0.

```
int[] numbers = new int[5];
```

index	0	1	2	3	4
value	0	0	0	0	0

type`[]` **name** `= {`**value**`,` **value**`,` … **value**`};`

- Infers length from number of values provided. Example:

```
int[] numbers = {12, 49, -2, 26, 5, 17, -6};
```

index	0	1	2	3	4	5	6
value	12	49	-2	26	5	17	-6

Accessing elements; length

```
name[index]              // access
name[index] = value;     // modify
name.length
```

- Legal indexes: between **0** and the **array's length - 1**.

```
numbers[3] = 88;
for (int i = 0; i < numbers.length; i++) {
    System.out.print(numbers[i] + " ");
}
System.out.println(numbers[-1]);   // exception
System.out.println(numbers[7]);    // exception
```

index	0	1	2	3	4	5	6
value	12	49	-2	88	5	17	-6

Limitations of arrays

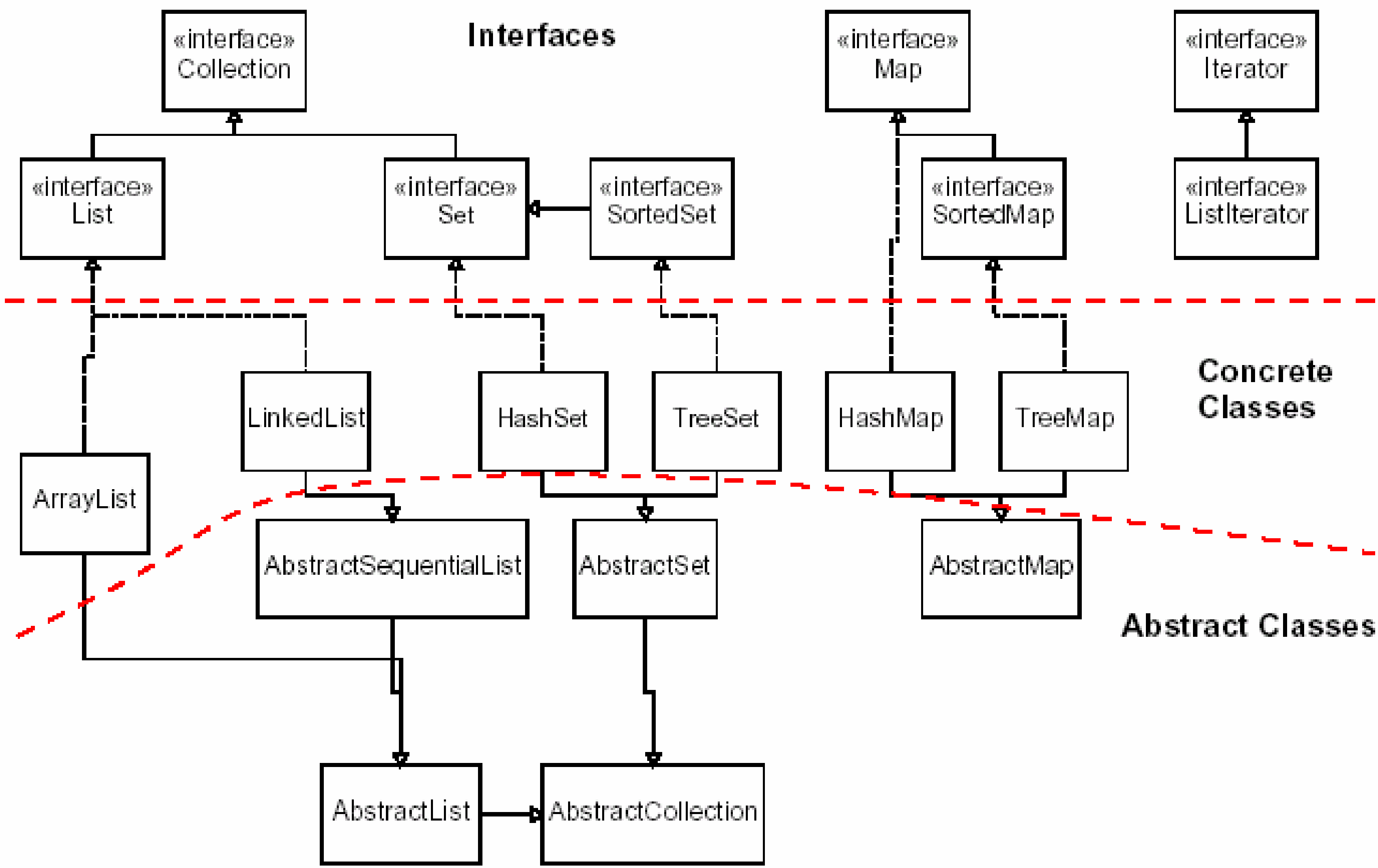
- Arrays are useful, but they have many flaws and limitations:
  - size cannot be changed after the array has been constructed
  - no built-in way to print the array
  - no built-in way to insert/remove an element
  - no search feature
  - no sort feature
  - no easy duplicate detection/removal
  - inconsistent syntax with other objects (length vs. length() vs. size())
  - ...

Collections

- **collection**: An object that stores data (objects) inside it.
  - the objects of data stored are called **elements**
  - typical operations: *add*, *remove*, *clear*, *contains* (search), *size*
  - some collections maintain an ordering; some allow duplicates
  - **data structure**: underlying implementation of a collection's behavior
    - most collections are based on an array or a set of linked node objects
  - examples found in the Java class libraries:
    - `ArrayList`, `LinkedList`, `HashMap`, `TreeSet`, `PriorityQueue`
  - all collections are in the `java.util` package

```
import java.util.*;
```

Java collection framework



Abstract data types (ADTs)

- **abstract data type (ADT)**: A specification of a collection of data and the operations that can be performed on it.
  - Describes *what* a collection does, not *how* it does it.

- Java's collection framework uses interfaces to describe ADTs:
  - `Collection`, `Deque`, `List`, `Map`, `Queue`, `Set`

- An ADT can be implemented in multiple ways by classes:
  - `ArrayList` and `LinkedList` implement `List`
  - `HashSet` and `TreeSet` implement `Set`
  - `LinkedList` , `ArrayDeque`, etc. implement `Queue`

Constructing a collection

Interface<**Type**> **name** = `new` **Class**<**Type**>`();`

- Use the ADT interface as the variable type.
  - Use the specific collection implementation class on the right.

- Specify the type of its elements between < and >.
  - This is called a *type parameter* or a *generic* class.
  - Allows the same `ArrayList` class to store lists of different types.

```
List<String> names = new ArrayList<String>();
names.add("Marty Stepp");
names.add("Stuart Reges");
```

Why use ADTs?

- **Q:** Why would we want more than one kind of list, queue, etc.?
  - (e.g. Why do we need both `ArrayList` and `LinkedList`?)

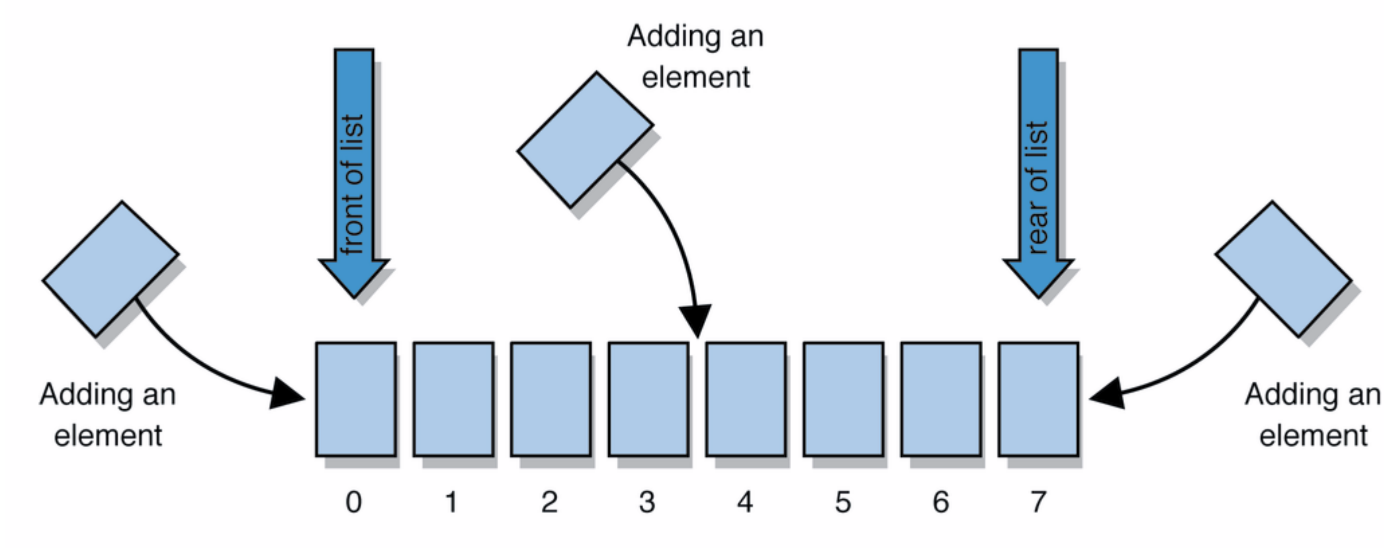
- **A:** Each implementation is more efficient at certain tasks.
  - `ArrayList` is faster for adding/removing at the end; `LinkedList` is faster for adding/removing at the front/middle.
  - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.

- **Q:** Why declare our variables using interface types (e.g. `List`)?
  - (e.g. `List<String> list = new ArrayList<String>();`)

- **A:** So that the program could be changed to use a different implementation later without needing to change the code much.

Lists

- **list**: a collection storing an ordered sequence of elements
  - each element is accessible by a 0-based **index**
  - a list has a **size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere
  - in Java, represented by the `List` interface, implemented by the `ArrayList` and `LinkedList` classes



`List` methods

constructor ( ) **constructor** (**list**)	creates a new empty list, or a set based on the elements of another list
`add(`**value**`)`	appends value at end of list
`add(`**index**`, `**value**`)`	inserts given value just before the given index, shifting subsequent values to the right
`clear()`	removes all elements of the list
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values to the left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`size()`	returns the number of elements in list
`toString()`	returns a string representation of the list such as `"[3, 42, -7, 15]"`

List methods 2

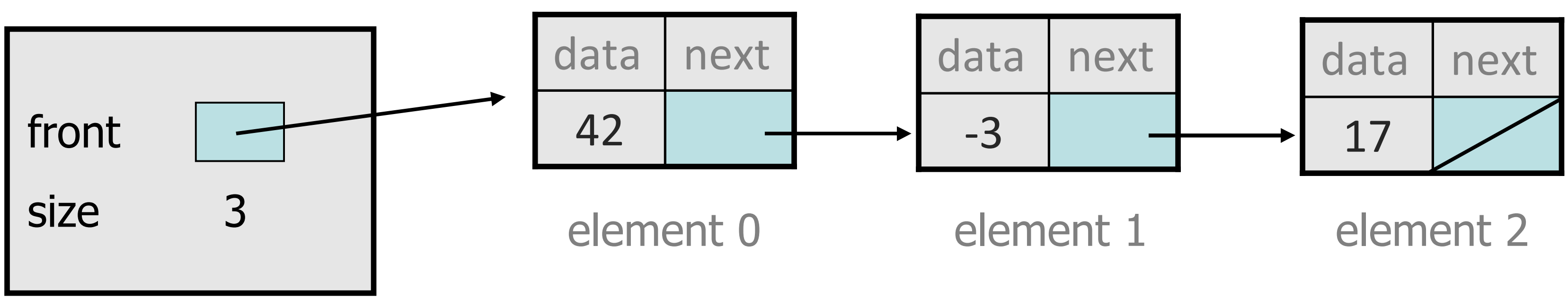
addAll(**list**) addAll(**index**, **list**)	adds all elements from the given list to this list (at the end of the list, or inserts them at the given index)
contains(**value**)	returns true if given value is found somewhere in this list
containsAll(**list**)	returns true if this list contains every element from given list
equals(**list**)	returns true if given other list contains the same elements
iterator() listIterator()	returns an object used to examine the contents of the list
lastIndexOf(**value**)	returns last index value is found in list (-1 if not found)
remove(**value**)	finds and removes the given value from this list
removeAll(**list**)	removes any elements found in the given list from this list
retainAll(**list**)	removes any elements *not* found in given list from this list
subList(**from**, **to**)	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
toArray()	returns the elements in this list as an array

List implementation

- `ArrayList` is built using an internal "unfilled" array and a size field to remember how many elements have been added

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	42	-3	17	0	0	0	0	0	0	0
size	3									

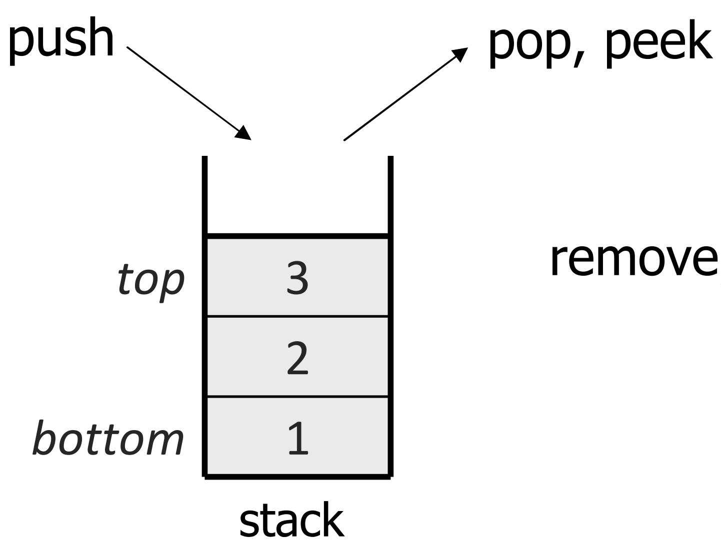
- `LinkedList` is built using a chain of small "node" objects, one for each element of the data, with a link to a "next" node object

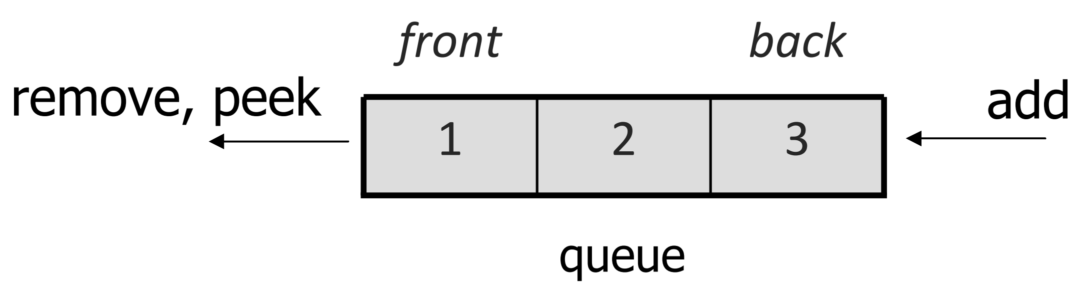


Stacks and queues

- **stack**: Retrieves elements in the reverse of the order they were added.
- **queue**: Retrieves elements in the same order they were added.

- **Q:** Similar to a list; why do we also have stacks and queues?
  - **A:** Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.





Class `Stack`

`Stack<`**E**`>()`	constructs a new stack with elements of type **E**
`push(`**value**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it; throws `EmptyStackException` if stack is empty
`peek()`	returns top value from stack without removing it; throws `EmptyStackException` if stack is empty
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

```
Stack<Integer> s = new Stack<Integer>();
s.push(42);
s.push(-3);
s.push(17);                   // bottom [42, -3, 17] top
System.out.println(s.pop()); // 17
```

Interface `Queue`

`add(`**value**`)`	places given value at back of queue
`remove()`	removes value from front of queue and returns it; throws a `NoSuchElementException` if queue is empty
`peek()`	returns front value from queue without removing it; returns `null` if queue is empty
`size()`	returns number of elements in queue
`isEmpty()`	returns `true` if queue has no elements

```
Queue<Integer> q = new LinkedList<Integer>();
q.add(42);
q.add(-3);
q.add(17);          // front [42, -3, 17] back
System.out.println(q.remove());    // 42
```

- When constructing a queue you must use a new `LinkedList` object instead of a `Queue` object.

Queue idioms

- As with stacks, must pull contents out of queue to view them.

```
// process (and destroy) an entire queue
while (!q.isEmpty()) {
    do something with q.remove();
}
```

  - another idiom: Examining each element exactly once.

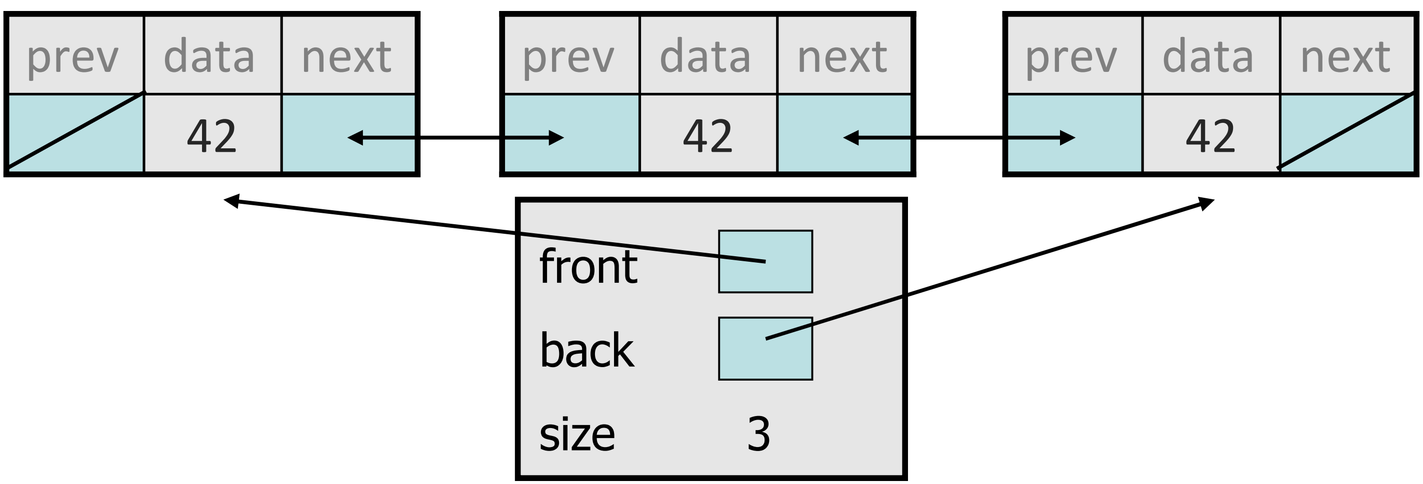
```
int size = q.size();
for (int i = 0; i < size; i++) {
    do something with q.remove();
    (including possibly re-adding it to the queue)
}
```

Stack/Queue implementation

- Stacks are almost always implemented using an array (why?)

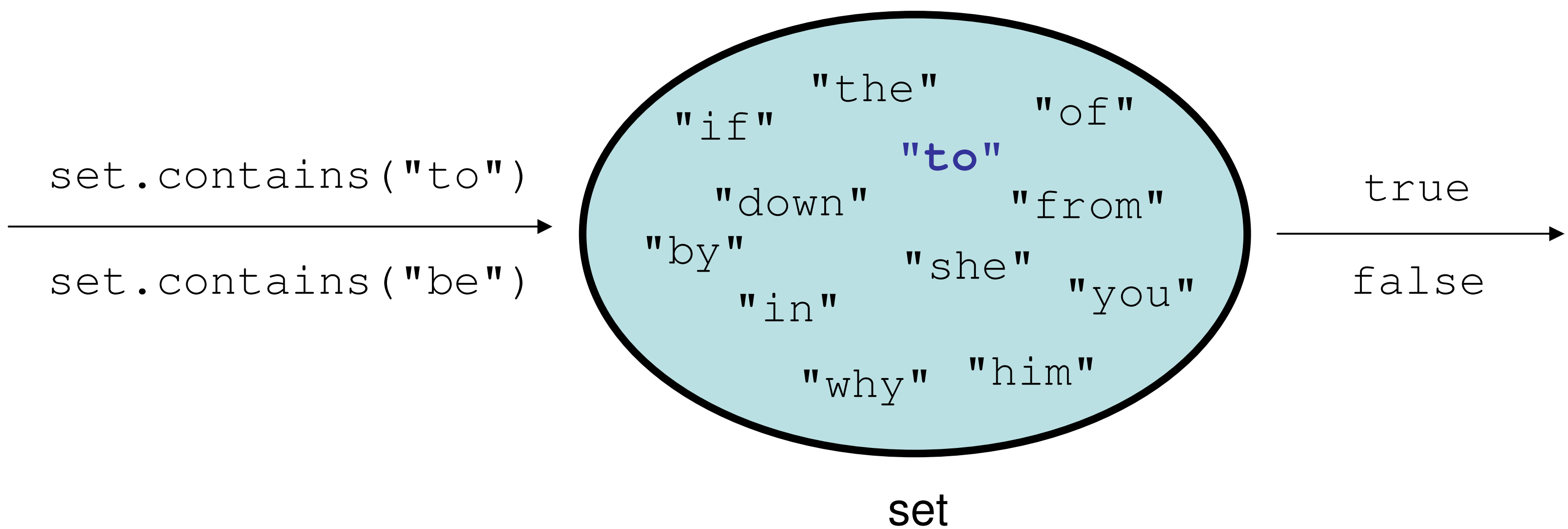
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	42	-3	17	0	0	0	0	0	0	0
size	3									

- Queues are built using a doubly-linked list with a front and back reference, or using an array with front and back indexes (why?)



Sets

- **set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - add, remove, search (contains)
  - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order.



Set implementation

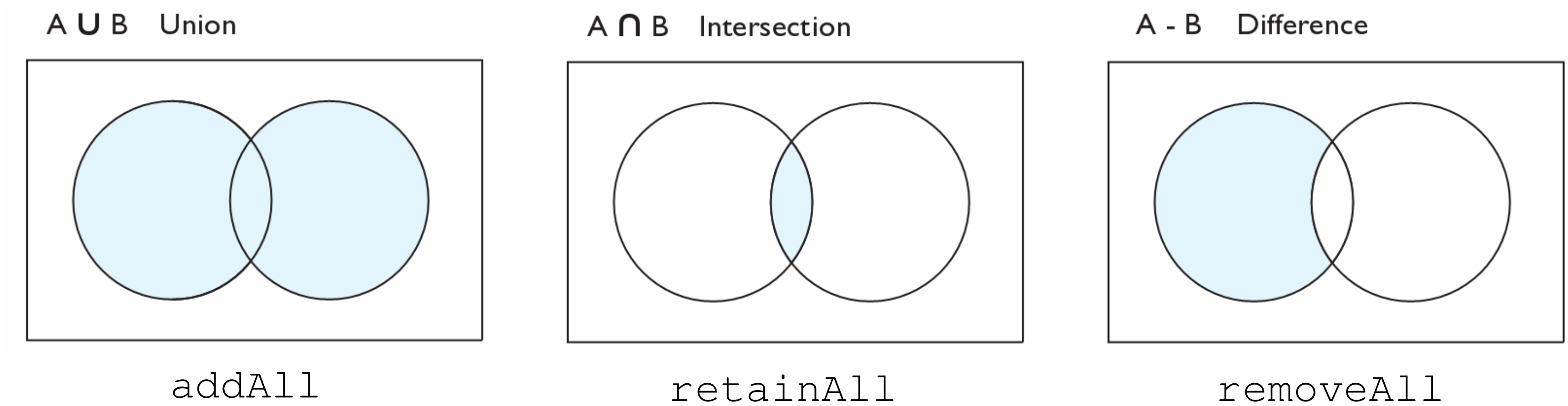
- in Java, sets are represented by `Set` interface in `java.util`
- `Set` is implemented by `HashSet` and `TreeSet` classes
  - `HashSet`: implemented using a "hash table" array;
    very fast: constant runtime (O(1)) for all operations
    elements are stored in unpredictable order
  - `TreeSet`: implemented using a "binary search tree";
    pretty fast: logarithmic runtime (O(log N)) for all operations
    elements are stored in sorted order
  - `LinkedHashSet`: O(1) but stores in order of insertion

Set methods

```
List<String> list = new ArrayList<String>();
...
Set<Integer> set = new TreeSet<Integer>();      // empty
Set<String> set2 = new HashSet<String>(list);
```

constructor () **constructor** (**collection**)	creates a new empty set, or a set based on the elements of a collection
add (**value**)	adds the given value to the set
contains (**value**)	returns true if the given value is found in this set
remove (**value**)	removes the given value from the set
clear ()	removes all elements of the set
size ()	returns the number of elements in list
isEmpty ()	returns true if the set's size is 0
toString ()	returns a string such as "[3, 42, -7, 15]"

Set operations



`addAll(`**collection**`)`	adds all elements from the given collection to this set
`containsAll(`**coll**`)`	returns `true` if this set contains every element from given set
`equals(`**set**`)`	returns `true` if given other set contains the same elements
`iterator()`	returns an object used to examine set's contents *(seen later)*
`removeAll(`**coll**`)`	removes all elements in the given collection from this set
`retainAll(`**coll**`)`	removes elements *not* found in given collection from this set
`toArray()`	returns an array of the elements in this set

Sets and ordering

- `HashSet` : elements are stored in an unpredictable order

```
Set<String> names = new HashSet<String>();
names.add("Jake");
names.add("Robert");
names.add("Marisa");
names.add("Kasey");
System.out.println(names);
// [Kasey, Robert, Jake, Marisa]
```

- `TreeSet` : elements are stored in their "natural" sorted order

```
Set<String> names = new TreeSet<String>();
...
// [Jake, Kasey, Marisa, Robert]
```

- `LinkedHashSet` : elements stored in order of insertion

```
Set<String> names = new LinkedHashSet<String>();
...
// [Jake, Robert, Marisa, Kasey]
```

Comparable

- If you want to store objects of your own class in a `TreeSet`:
  - Your class must implement the `Comparable` interface to define a natural ordering function for its objects.

```
public interface Comparable<E> {
    public int compareTo(E other);
}
```

- A call to `compareTo` must return:

a value <	0	if `this` object comes "before" the `other` object,
a value >	0	if `this` object comes "after" the `other` object,
or	0	if `this` object is considered "equal" to the `other`

The "for each" loop (7.1)

```
for (type name : collection) {
    statements;
}
```

- Provides a clean syntax for looping over the elements of a `Set`, `List`, array, or other collection

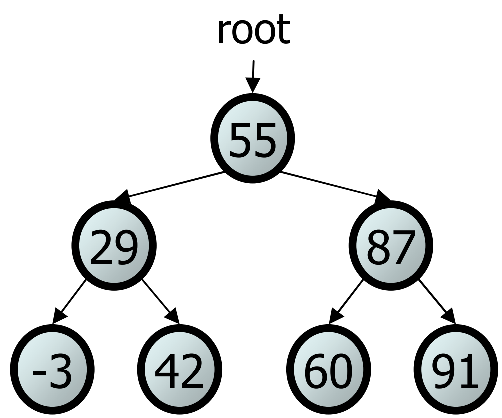
```
Set<Double> grades = new HashSet<Double>();
...

for (double grade : grades) {
    System.out.println("Student's grade: " + grade);
}
```

  - needed because sets have no indexes; can't `get` element `i`

Set implementation

- `TreeSet` is implemented using a *binary search tree*



- `HashSet` is built using a special kind of array called a *hash table*

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	60	91	42	-3	0	55	0	87	0	29
size	7									