CIS 313:
Intermediate Data Structure

second slide

algorithm time bounds

Let $\mathcal{A}$ be some algorithm operating on an input x

- worst case
  - $\mathcal{A}$ has worst case time O(t(n)) if there are constants c and N such that for all n>N and all inputs x of length n, $\mathcal{A}$ completes its computation on input x using at most c*t(n) steps
  - $\mathcal{A}$ has worst case time $\Omega(t(n))$ if there are constants c and N such that for all n>N there exists an input x of length n such that $\mathcal{A}$ uses at least c*t(n) steps to finish its computation on x
- average case
- expected case (a measure that makes sense if algorithm is randomized)
- best case (not very useful)
- smoothed analysis (complicated)

Theme:
Abstract Data Types vs. Implementation

- Abstract data type: Set of operations.

For example, a list should support:

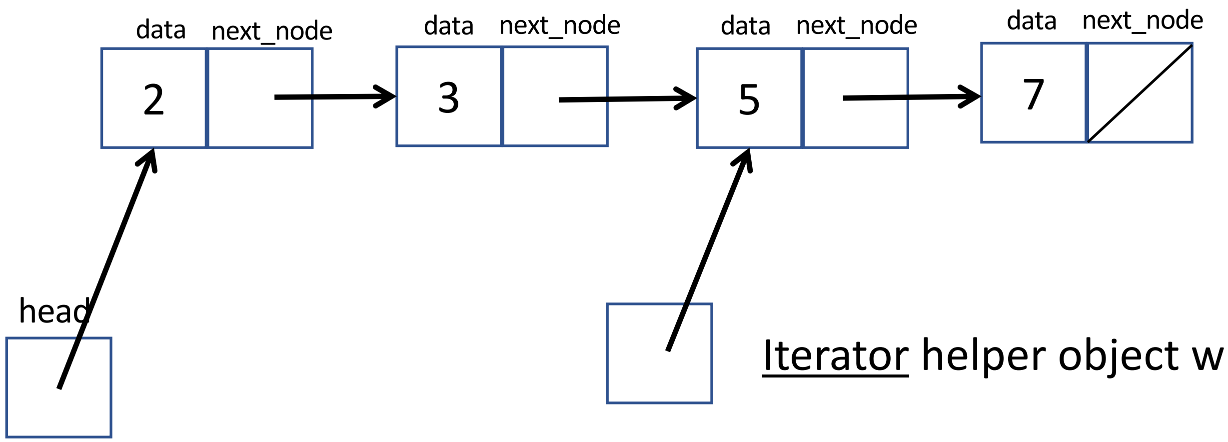
  - append (adding item to end of list)
  - length (number of items in list)
  - get (access ith element of list)
  - insert/remove (add or remove element at position i)
  - Iterator (get an iterator helper object)
  - Etc.

- Implementation: How those operations are implemented.

For example, a list could be implemented as a linked list or array list.

Linked List Implementation

Node object with fields: data, next_node, prev_node (*optional*)



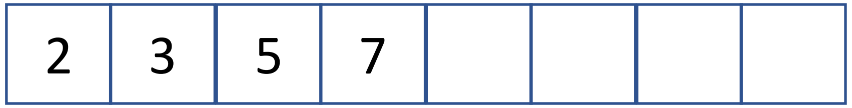
Iterator helper object with fields: curr_node

LinkedList object with fields: head, tail (*optional*), length (*optional*)

Array List Implementation

ArrayList object with fields: elements, size, capacity

elements:



2	3	5	7				

size: 4

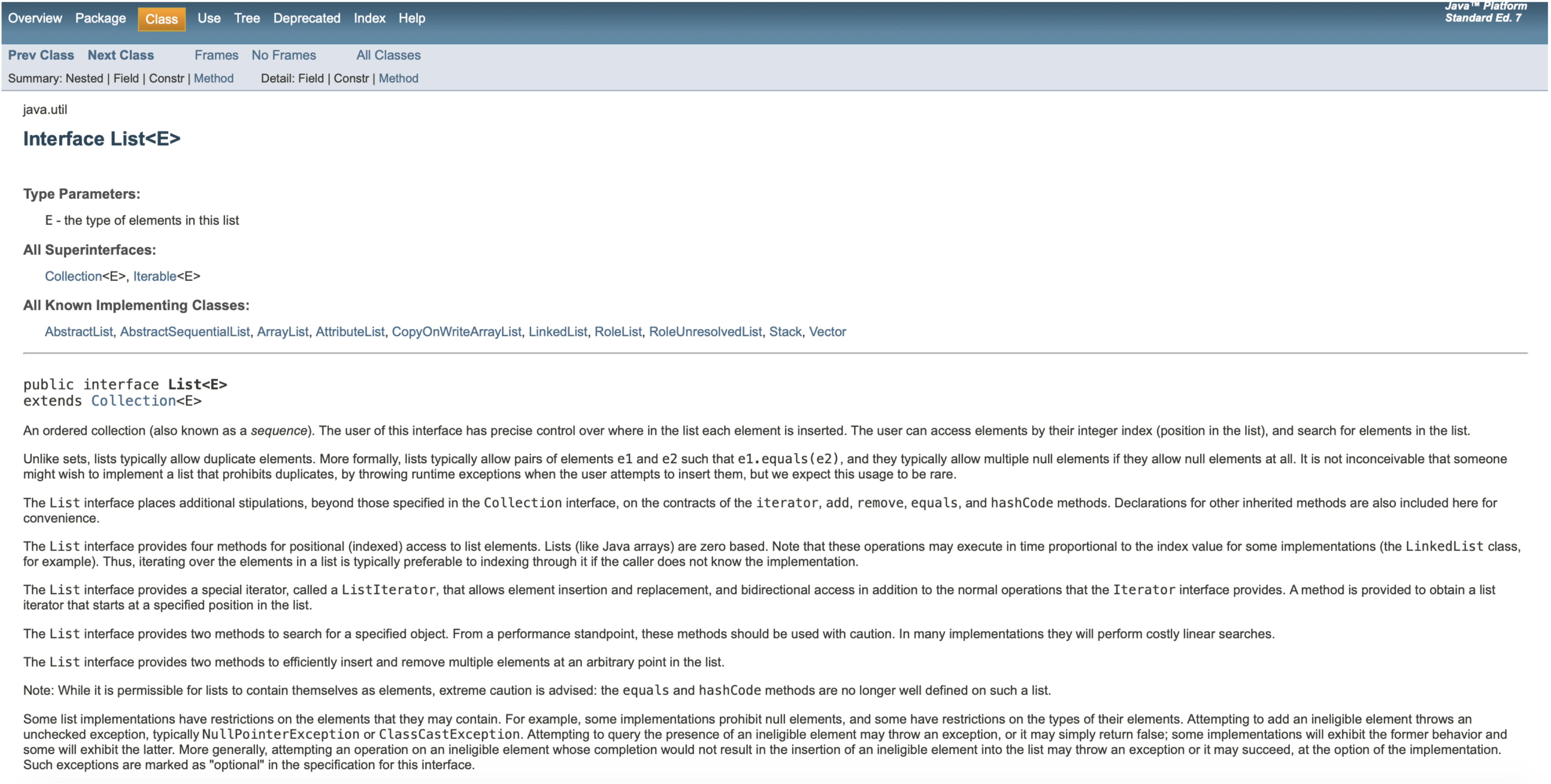
capacity: 8

What would an iterator look like for an ArrayList?

Appending:

```
if size == capacity:
   new_elements = new array[capacity * 2]
   for i = 0 to size – 1:
      new_elements[i] = elements[i]
   elements = new_elements
   capacity = capacity * 2

elements[size] = data
size = size + 1
```



java.util

Interface List<E>

Type Parameters:

E - the type of elements in this list

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

---

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements `e1` and `e2` such that `e1.equals(e2)`, and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that someone might wish to implement a list that prohibits duplicates, by throwing runtime exceptions when the user attempts to insert them, but we expect this usage to be rare.

The `List` interface places additional stipulations, beyond those specified in the `Collection` interface, on the contracts of the `iterator`, `add`, `remove`, `equals`, and `hashCode` methods. Declarations for other inherited methods are also included here for convenience.

The `List` interface provides four methods for positional (indexed) access to list elements. Lists (like Java arrays) are zero based. Note that these operations may execute in time proportional to the index value for some implementations (the `LinkedList` class, for example). Thus, iterating over the elements in a list is typically preferable to indexing through it if the caller does not know the implementation.

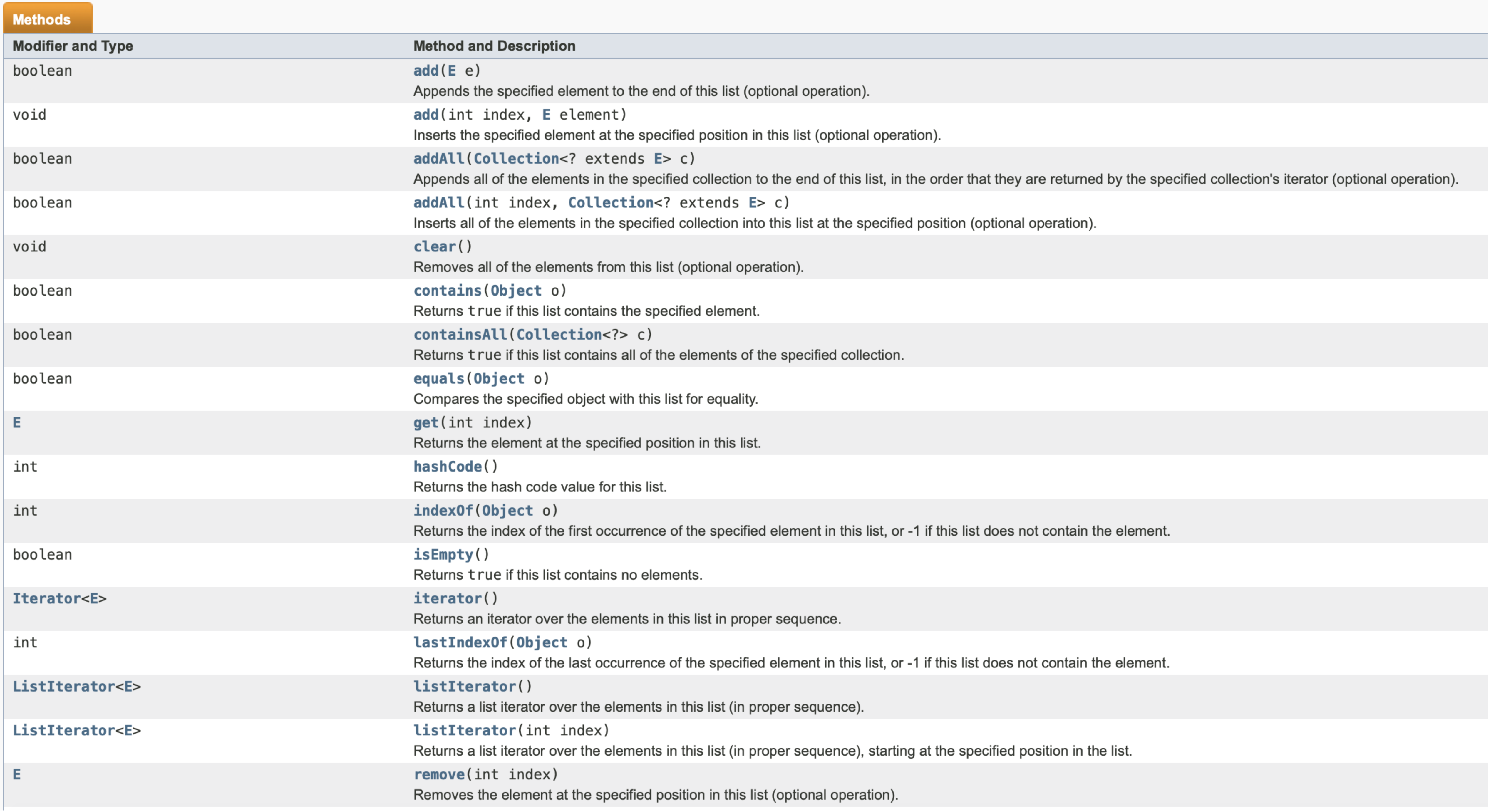
The `List` interface provides a special iterator, called a `ListIterator`, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the `Iterator` interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

The `List` interface provides two methods to search for a specified object. From a performance standpoint, these methods should be used with caution. In many implementations they will perform costly linear searches.

The `List` interface provides two methods to efficiently insert and remove multiple elements at an arbitrary point in the list.

Note: While it is permissible for lists to contain themselves as elements, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a list.

Some list implementations have restrictions on the elements that they may contain. For example, some implementations prohibit null elements, and some have restrictions on the types of their elements. Attempting to add an ineligible element throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible element may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible element whose completion would not result in the insertion of an ineligible element into the list may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.



Methods

Modifier and Type	Method and 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).
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator (optional operation).
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	**clear**() Removes all of the elements from this list (optional operation).
boolean	**contains**(Object o) Returns `true` if this list contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns `true` if this list contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this list for equality.
E	**get**(int index) Returns the element at the specified position in this list.
int	**hashCode**() Returns the hash code value for 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.
boolean	**isEmpty**() Returns `true` if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator**() Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	**listIterator**(int index) Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
E	**remove**(int index) Removes the element at the specified position in this list (optional operation).

	Linked List	Array List
append		
get		
length		
find		
insert		

What’s the complexity of each operation, if the list currently has n elements?

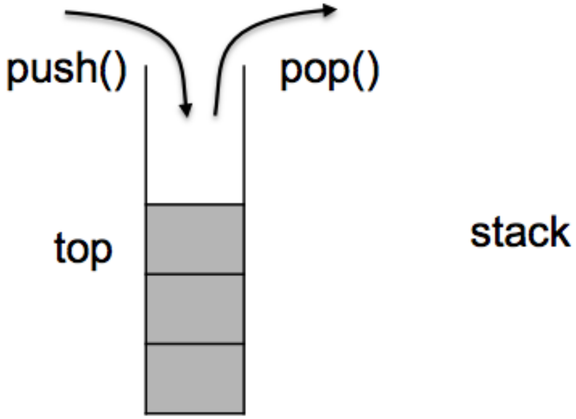
linear data structures

Our basic structures: quick review

- arrays
- linked lists
- stacks
- queues
- priority queue
- binary heap

stacks

- LIFO: last-in first-out
- can implement stack with array, linked list, …
- uses of stack
  - implement recursion
  - expression evaluation
  - depth-first search
- stack operations
  - push
  - pop
  - top (or peek)
  - init, isEmpty, isFull



example use of stack: evaluate postfix

postfix: operator after the operands

- (2+3)*7 becomes 2 3 + 7 *
- 2+(3*7) is 2 3 7 * +
- no need for parens

```
to evaluate a postfix expression E:

use operand stack S

for each token x in E, scanning L to R
    if x is operand (value)
          S.push(x)
    else x is operator (+, *, -, …)
         v=S.pop
         w=S.pop
         z = result of applying operator x to (w,v)
         S.push(z)

return S.pop

note: if try to pop on empty stack, then underflow error
and if stack not empty after last pop then overflow error
```

queues

- FIFO: first-in, first-out
- useful in job scheduling, models “standing in line”
- implementation: linked list, array (wraparound)
- use to compute breath-first search of tree, graph
- operations
  - enqueue
  - dequeue
  - front, isEmpty, isFull

example with tree: stack vs queue

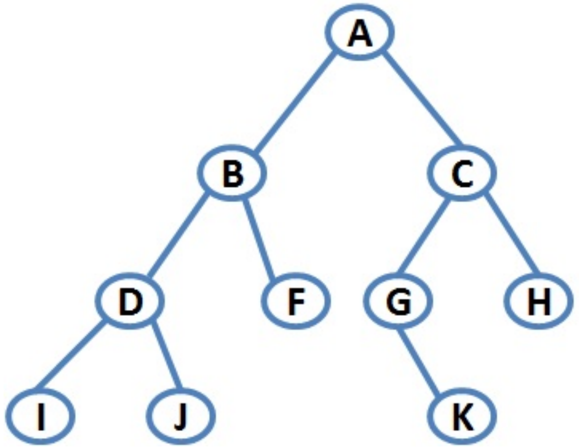
Consider a tree T consisting of simple nodes p: fields p.left, p.right, and p.value

We have a simple recursive preorder traversal whose initial call is preorderTrav(T.root)

```
preorderTrav(node p)

   print p.value
   if p.left != null
        preorderTrav(p.left)
   if p.right != null
        preorderTrav(p.right)
```

example with tree (cont'd)



preorder traversal:

A B D I J F C G K H

note
inorder: I D J B F A G K C H
postorder: I J D F B K G H C A

example with tree (cont'd)

implement that traversal with a stack:

```
stack S of node

S.push(T.root)

while (not S.isEmpty)
    p = S.pop
    print p.value
    if p.right!=null
         S.push(p.right)
     if p.left!=null
         S.push(p.left)
```

note: need to push the right side first so left side gets visited before it

step through traversal with tree on previous slide

example with tree (cont'd)

implement that traversal with a queue:

```
queue Q of node

Q.enqueue(T.root)

while (not Q.isEmpty)
    p = Q.dequeue
    print p.value
    if p.right!=null
        Q.enqueue(p.right)
    if p.left!=null
        Q.enqueue(p.left)
```

stack S -> queue Q
pop -> dequeue
push -> enqueue

what order do we get with this method?

try example

example with tree (conclusion)

previous queue algorithm gives a (reverse) level-order:

A C B H G F D K J I

nice, somewhat unrelated question,
Reconstruct a binary tree from two of the traversal sequences

example: you are given only
A B D I J F C G K H (preorder)
I D J B F A G K C H (inorder)
now build the tree

priority queues

- chapter 6
- abstract operations (implementation independent)
- maintains a set S of elements
- operations
  - insert(x)
  - max (or returnMax)
  - extractMax (removes it)
  - increaseKey(x,k) (set key of x to a new larger value)
  - -OR- insert, min, extractMin, decreaseKey

can sort with priority queue (assuming the descending order)

```
PQSort(array A)
//array A has n elements

create PQ Q

for i=1 to n
     Q.insert(A[i])

for i = n down to 1
     A[i] = Q.extractMax
```

cannot analyze time
without implementation

unordered list implementation of PQ

- simple
- insert(x) is O(1)
- extractMax is O(n)
- What does PQSort look like?
  - selection sort
  - time O(n^2), work done in second loop

ordered list implementation of PA

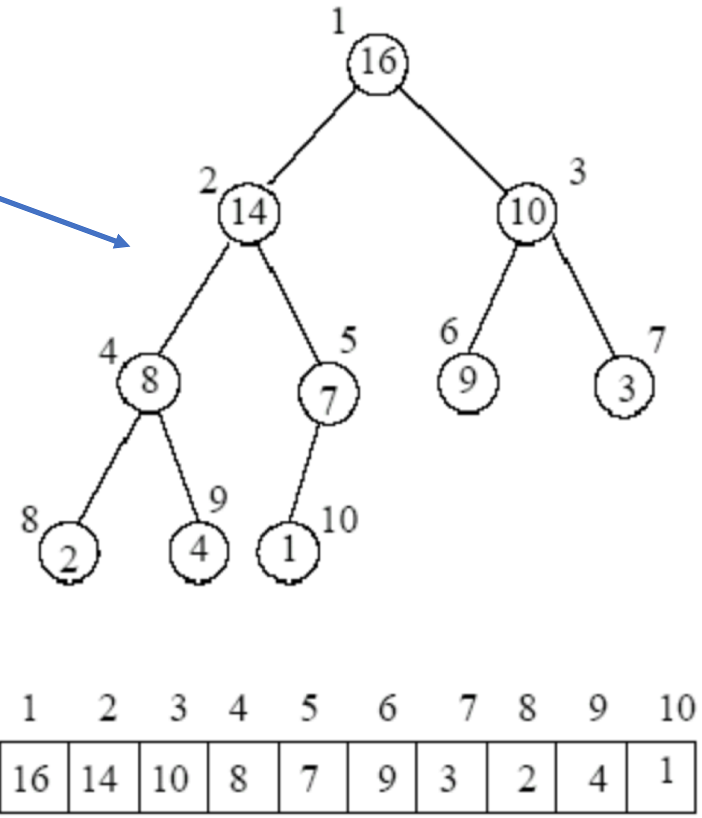
- also simple
- insert(x) is O(n)
- extractMax is O(1)
- What does PQSort look like?
  - insertion sort
  - time O(n^2), work done in first loop

binary heap implementation of PQ

- most common implementation
- operations are O(*log* n)
- uses a binary tree structure
- except that the tree is stored in an array with no pointers
- it is an *implicit* tree, children and parents inferred from location in array

- PQSort becomes *heapsort*

binary heap



height of tree is O(*lg* n)

- stored in array
- item located in postion i
  - parent in location $\lfloor i/2 \rfloor$
  - left child in position $2i$
  - right child in postion $2i+1$
- tree is complete
  - all nodes have two children, except maybe parent of “last” one
- tree maintains heap property
  - value stored at location i is greater than or equal to values stored in both its children
- fact: a binary heap with n elements has the height of $\lfloor \lg n \rfloor$ (why?)