LISTS AND ITERATORS

Abstract data types that represent a linear sequence of elements, with more general support for adding or removing elements at arbitrary positions.

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

THE JAVA.UTIL.LIST ADT

- The java.util.List interface includes the following **index based** methods:

```
/** A simplified version of the java.util.List interface. */
public interface List<E> {
  /** Returns the number of elements in this list. */
  int size();

  /** Returns whether the list is empty. */
  boolean isEmpty();

  /** Returns (but does not remove) the element at index i. */
  E get(int i) throws IndexOutOfBoundsException;

  /** Replaces the element at index i with e, and returns the replaced element. */
  E set(int i, E e) throws IndexOutOfBoundsException;

  /** Inserts element e to be at index i, shifting all subsequent elements later. */
  void add(int i, E e) throws IndexOutOfBoundsException;

  /** Removes/returns the element at index i, shifting subsequent elements earlier. */
  E remove(int i) throws IndexOutOfBoundsException;
}
```

Code Fragment 7.1: A simple version of the List interface.

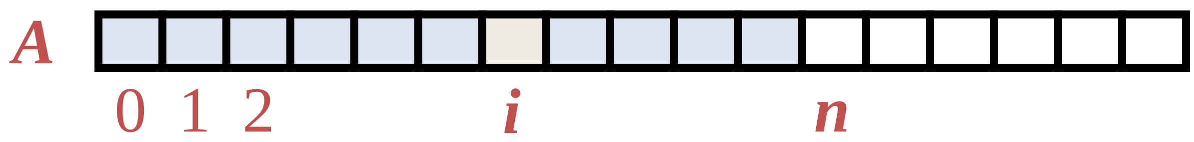
EXAMPLE

- A sequence of List operations:

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

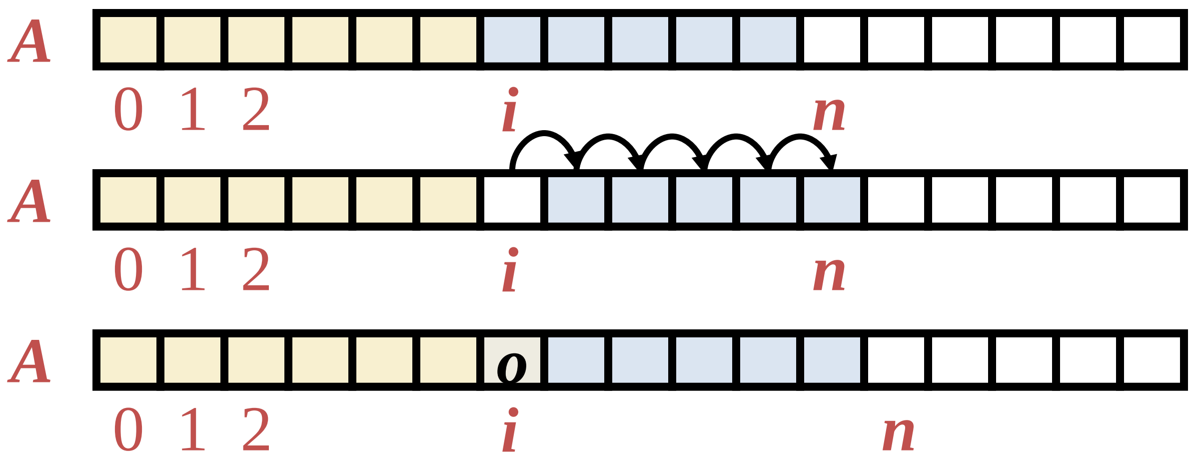
ARRAY LISTS

- An obvious choice for implementing the **list ADT** is to use an array, **A**, where **A[i]** stores (a reference to) the element with index **i**.
- With a representation based on an array **A**, the get(**i**) and set(**i**, **e**) methods are easy to implement by accessing **A[i]** (assuming **i** is a legitimate index).



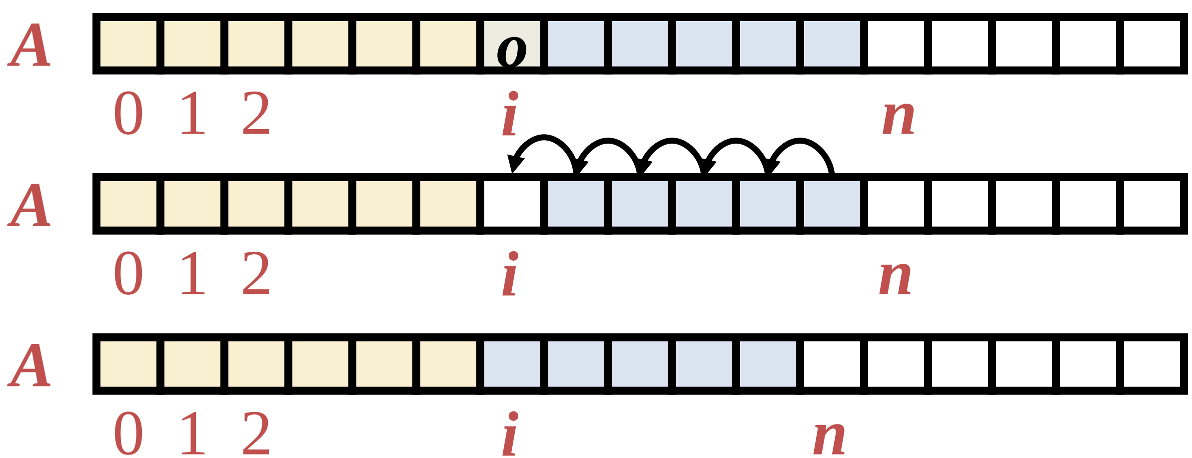
INSERTION

- In an operation $\boldsymbol{add(i, o)}$, we need to make room for the new element by shifting forward the $\boldsymbol{n - i}$ elements $\boldsymbol{A}[\boldsymbol{i}], \ldots, \boldsymbol{A}[\boldsymbol{n} - 1]$
- In the worst case ($\boldsymbol{i} = 0$), this takes $\boldsymbol{O}(\boldsymbol{n})$ time



ELEMENT REMOVAL

- In an operation ***remove***(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



PERFORMANCE OF *ARRAY LIST*

- In an array-based implementation of a list (*array list*):
  - The space used by the data structure is $O(n)$
  - Indexing the element at i takes $O(1)$ time
  - ***add*** and ***remove*** run in $O(n)$ time
- In an ***add*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one ...

Method	Running Time
size()	$O(1)$
isEmpty()	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

JAVA IMPLEMENTATION: BOUNDED CAPACITY

An implementation of a simple ArrayList class with <u>bounded capacity</u>

```
public class ArrayList<E> implements List<E> {
  // instance variables
  public static final int CAPACITY=16;     // default array capacity
  private E[ ] data;                       // generic array used for storage
  private int size = 0;                    // current number of elements
  // constructors
  public ArrayList( ) { this(CAPACITY); }  // constructs list with default capacity
  public ArrayList(int capacity) {         // constructs list with given capacity
    data = (E[ ]) new Object[capacity];    // safe cast; compiler may give warning
  }

  // public methods
  /** Returns the number of elements in the array list. */
  public int size( ) { return size; }
  /** Returns whether the array list is empty. */
  public boolean isEmpty( ) { return size == 0; }
  /** Returns (but does not remove) the element at index i. */
  public E get(int i) throws IndexOutOfBoundsException {
    checkIndex(i, size);
    return data[i];
  }
  /** Replaces the element at index i with e, and returns the replaced element. */
  public E set(int i, E e) throws IndexOutOfBoundsException {
    checkIndex(i, size);
    E temp = data[i];
    data[i] = e;
    return temp;
  }
```

JAVA IMPLEMENTATION, CONT.

```
  /** Inserts element e to be at index i, shifting all subsequent elements later. */
  public void add(int i, E e) throws IndexOutOfBoundsException,
                                      IllegalStateException {
    checkIndex(i, size + 1);
    if (size == data.length)                  // not enough capacity
      throw new IllegalStateException("Array is full");
    for (int k=size−1; k >= i; k−−)           // start by shifting rightmost
      data[k+1] = data[k];
    data[i] = e;                              // ready to place the new element
    size++;
  }
  /** Removes/returns the element at index i, shifting subsequent elements earlier. */
  public E remove(int i) throws IndexOutOfBoundsException {
    checkIndex(i, size);
    E temp = data[i];
    for (int k=i; k < size−1; k++)            // shift elements to fill hole
      data[k] = data[k+1];
    data[size−1] = null;                      // help garbage collection
    size−−;
    return temp;
  }
  // utility method
  /** Checks whether the given index is in the range [0, n−1]. */
  protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
    if (i < 0 || i >= n)
      throw new IndexOutOfBoundsException("Illegal index: " + i);
  }
}
```

DYNAMIC ARRAY:

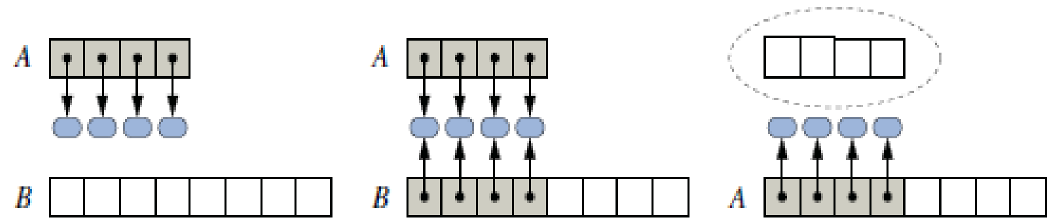
- Let push(o) be the operation that adds element o at the end of the list
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

Algorithm *push*(*o*)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$

IMPLEMENTING A DYNAMIC ARRAY

- Provide means to "grow" the array *A*

1. Allocate a new array *B* with larger capacity.
2. Set *B*[*k*]=*A*[*k*], for *k*=0, . . . ,*n*−1, where *n* denotes current number of items.
3. Set *A* = *B*, that is, we henceforth use the new array to support the list.
4. Insert the new element in the new array.



create new array *B*

store elements of *A* in *B*

reassign reference *A* to the new array

REVISION TO OUR ORIGINAL ARRAYLIST IMPLEMENTATION,

```
/** Resizes internal array to have given capacity >= size. */
protected void resize(int capacity) {
  E[ ] temp = (E[ ]) new Object[capacity]; // safe cast; compiler may give warning
  for (int k=0; k < size; k++)
    temp[k] = data[k];
  data = temp;                             // start using the new array
}
```

```
  /** Inserts element e to be at index i, shifting all subsequent elements later. */
  public void add(int i, E e) throws IndexOutOfBoundsException {
    checkIndex(i, size + 1);
    if (size == data.length)              // not enough capacity
      resize(2 * data.length);            // so double the current capacity
    // rest of method unchanged...
```

Strategy #2: new array to have twice the capacity of the existing array

ADVANCE TOPIC: COMPARISON OF THE STRATEGIES

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations (amortization)
- We assume that we start with an empty list represented by a growable array of size 1
- We call amortized time of a push operation the average time taken by a push operation over the series of operations, i.e., $T(n)/n$

INCREMENTAL STRATEGY ANALYSIS

- Over n push operations, we replace the array $k = n/c$ times, where c is a constant
- The total time $T(n)$ of a series of n push operations is proportional to

Actual push op.

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- Thus, the amortized time of a push operation is $O(n)$

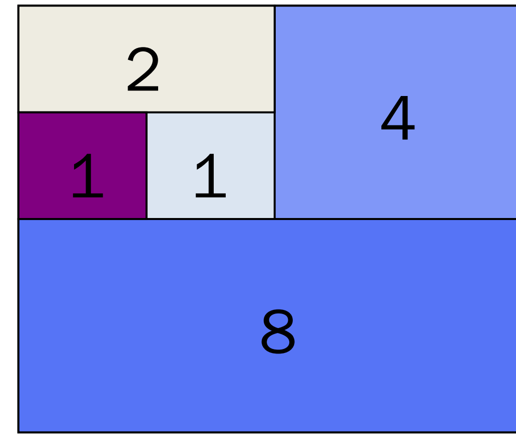
DOUBLING STRATEGY ANALYSIS

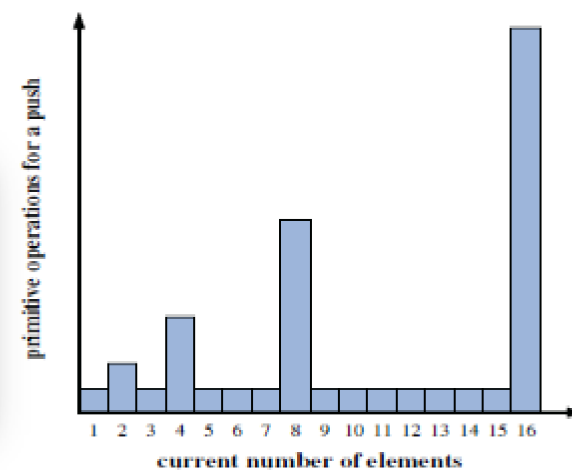
- We replace the array $k = \log_2 n$ times (2^{k+1} - $1 = n$; solve for k)
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$

geometric series





Proposition A.12: *If $k \geq 1$ is an integer constant, then*

$$\sum_{i=1}^{n} i^k \text{ is } \Theta(n^{k+1}).$$

Another common summation is the *geometric sum*, $\sum_{i=0}^{n} a^i$, for any fixed real number $0 < a \neq 1$.

DYNAMIC ARRAY: ANALYSIS EXAMPLE

```
/** Uses repeated concatenation to compose a String with n copies of character c. */
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;
}

/** Uses StringBuilder to compose a String with n copies c
public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder( );
  for (int j=0; j < n; j++)
    sb.append(c);
  return sb.toString( );
}
```

Code Fragment 4.2: Two algorithms for composi

Uses Regular Array: $O(n^2)$

Uses Dynamic Array: $O(n)$

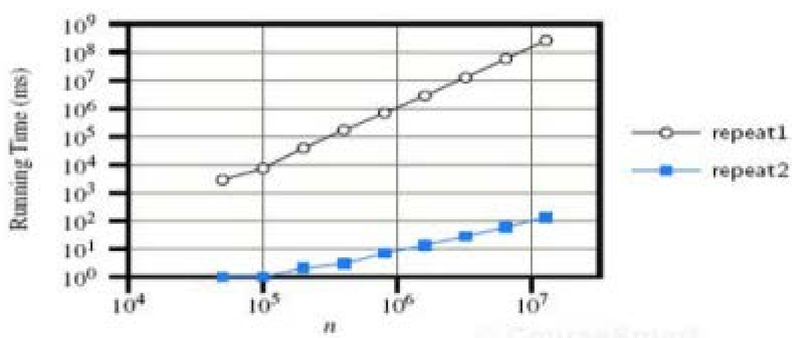
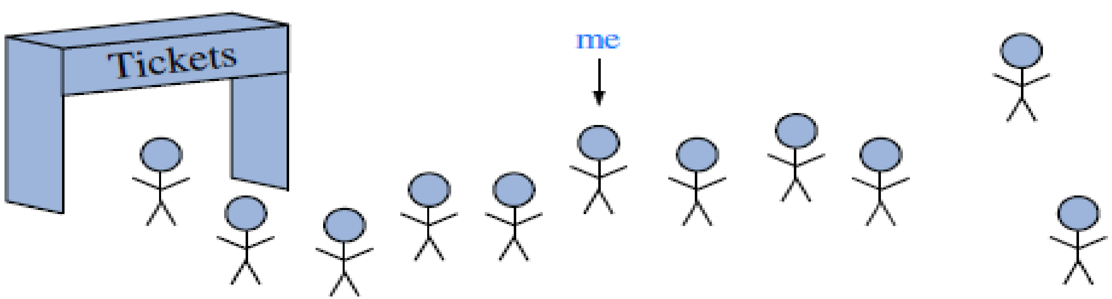


Figure 4.1: Chart of the results of the timing experiment from Code Fragment 4.2, displayed on a log-log scale. The divergent slopes demonstrate an order of magnitude difference in the growth of the running times.

POSITIONAL LISTS

- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a **positional list** ADT.
- A position acts as a marker or token within the broader positional list.
- A position *p* is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
  - P.getElement( ): Return the element stored at position p.



IMMEDIATE CHALLENGE IN DESIGNING THE ADT;

- Challenge: Achieve constant time insertions and deletions at arbitrary locations:
  - we effectively need a reference to the node at which an element is stored.
- We introduce the concept of a ***position***, which formalizes the intuitive notion of the “location” of an element relative to others in the list.
- Bad: ADT in which a node reference serves as the mechanism for describing a position.
  - Details of our implementation need to be known
  - Not a robust data structure (user can access or manipulate the nodes <- cause problems)
  - Bad encapsulating (implementation details exposed)

POSITIONAL LIST ADT

- Accessor methods:

We can subsequently use the returned position to traverse the list

```
Position<String> cursor = guests.first();
while (cursor != null) {
  System.out.println(cursor.getElement());
  cursor = guests.after(cursor);
}
```

first( ): Returns the position of the first element of L (or null if empty).

last( ): Returns the position of the last element of L (or null if empty).

before(p): Returns the position of L immediately before position p (or null if p is the first position).

after(p): Returns the position of L immediately after position p (or null if p is the last position).

isEmpty( ): Returns true if list L does not contain any elements.

size( ): Returns the number of elements in list L.

Positional List ADT, 2

- Update methods:

addFirst(e): Inserts a new element e at the front of the list, returning the position of the new element.

addLast(e): Inserts a new element e at the back of the list, returning the position of the new element.

addBefore(p, e): Inserts a new element e in the list, just before position p, returning the position of the new element.

addAfter(p, e): Inserts a new element e in the list, just after position p, returning the position of the new element.

set(p, e): Replaces the element at position p with element e, returning the element formerly at position p.

remove(p): Removes and returns the element at position p in the list, invalidating the position.

EXAMPLE

- A sequence of Positional List operations:

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first()	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement()	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last())	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

position instances, we use variables such as p and q

EXAMPLE CONT.

PositionList interface

```
/** An interface for positional lists. */
public interface PositionalList<E> {

  /** Returns the number of elements in the list. */
  int size( );

  /** Tests whether the list is empty. */
  boolean isEmpty( );

  /** Returns the first Position in the list (or null, if empty). */
  Position<E> first( );

  /** Returns the last Position in the list (or null, if empty). */
  Position<E> last( );

  /** Returns the Position immediately before Position p (or null, if p is first). */
  Position<E> before(Position<E> p) throws IllegalArgumentException;

  /** Returns the Position immediately after Position p (or null, if p is last). */
  Position<E> after(Position<E> p) throws IllegalArgumentException;

  /** Inserts element e at the front of the list and returns its new Position. */
  Position<E> addFirst(E e);

  /** Inserts element e at the back of the list and returns its new Position. */
  Position<E> addLast(E e);

  /** Inserts element e immediately before Position p and returns its new Position. */
  Position<E> addBefore(Position<E> p, E e)
    throws IllegalArgumentException;

  /** Inserts element e immediately after Position p and returns its new Position. */
  Position<E> addAfter(Position<E> p, E e)
    throws IllegalArgumentException;

  /** Replaces the element stored at Position p and returns the replaced element. */
  E set(Position<E> p, E e) throws IllegalArgumentException;

  /** Removes the element stored at Position p and returns it (invalidating p). */
  E remove(Position<E> p) throws IllegalArgumentException;
}
```

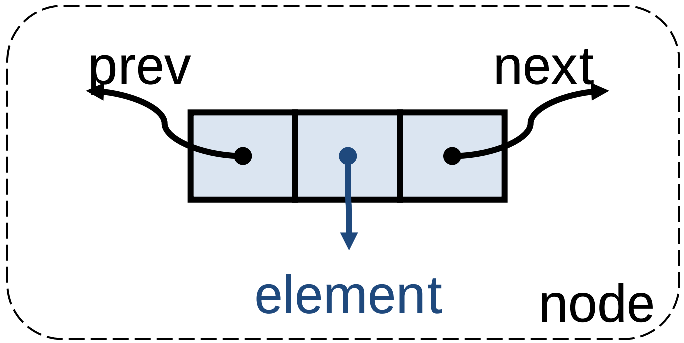
Code Fragment 7.8: The PositionalList interface.

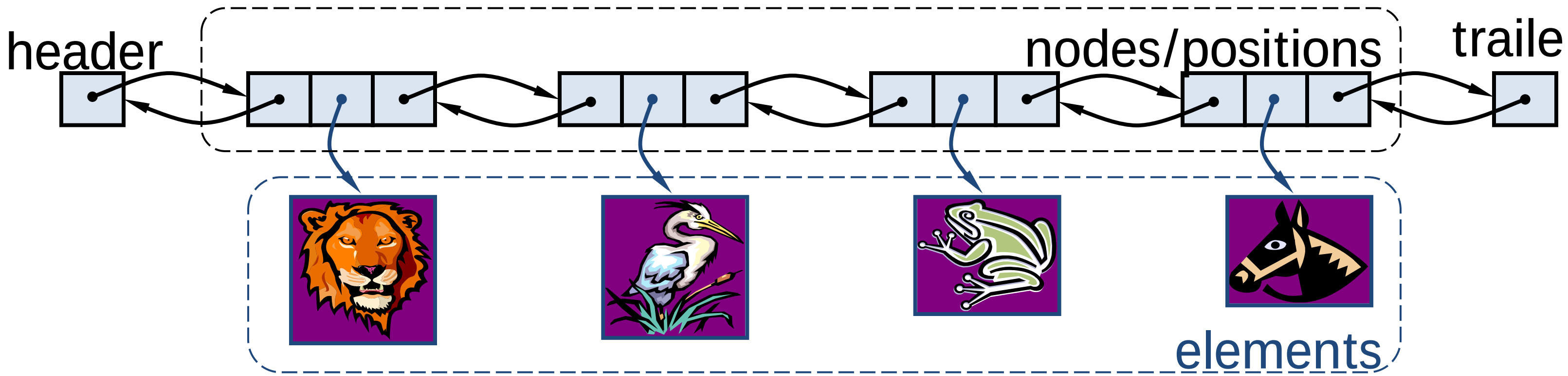
Position interface

```
public interface Position<E> {
  /**
   * Returns the element stored at this position.
   *
   * @return the stored element
   * @throws IllegalStateException if position no longer valid
   */
  E getElement( ) throws IllegalStateException;
}
```

POSITIONAL LIST IMPLEMENTATION USING DOUBLY LIKED LIST

- The most natural way to implement a positional list is with a **doubly-linked list.**
- NOTE: Not the same as the DoublyLinkedList class in Ch3
  - Difference in the management of the positional abstraction





```
/** Implementation of a positional list stored as a doubly linked list. */
public class LinkedPositionalList<E> implements PositionalList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> implements Position<E> {
    private E element;               // reference to the element stored at this node
    private Node<E> prev;            // reference to the previous node in the list
    private Node<E> next;            // reference to the subsequent node in the list
    public Node(E e, Node<E> p, Node<E> n) {
      element = e;
      prev = p;
      next = n;
    }
    public E getElement( ) throws IllegalStateException {
      if (next == null)                          // convention for defunct node
        throw new IllegalStateException("Position no longer valid");
      return element;
    }
    public Node<E> getPrev( ) {
      return prev;
    }
    public Node<E> getNext( ) {
      return next;
    }
    public void setElement(E e) {
      element = e;
    }
    public void setPrev(Node<E> p) {
      prev = p;
    }
    public void setNext(Node<E> n) {
      next = n;
    }
  } //----------- end of nested Node class -----------
```

definition of the nested Node<E> class, which implements the Position<E> interface.

```
  // instance variables of the LinkedPositionalList
  private Node<E> header;                          // header sentinel
  private Node<E> trailer;                         // trailer sentinel
  private int size = 0;                            // number of elements in the list

  /** Constructs a new empty list. */
  public LinkedPositionalList( ) {
    header = new Node<>(null, null, null);         // create header
    trailer = new Node<>(null, header, null);      // trailer is preceded by header
    header.setNext(trailer);                       // header is followed by trailer
  }
```

The declaration of the instance variables of the outer LinkedPositionalList class and its constructor.

The private **validate(p**) method is called anytime the user sends a Position instance as a parameter. It throws an exception if it determines that the position is invalid, and otherwise returns that instance, implicitly cast as a Node, so that methods of the Node class can subsequently be called.

The private **position(node)** method is used when about to return a Position to the user. Its primary purpose is to make sure that we do not expose either sentinel node to a caller, returning a null reference in such a case.

```
  // private utilities
  /** Validates the position and returns it as a node. */
  private Node<E> validate(Position<E> p) throws IllegalArgumentException {
    if (!(p instanceof Node)) throw new IllegalArgumentException("Invalid p");
    Node<E> node = (Node<E>) p;     // safe cast
    if (node.getNext( ) == null)    // convention for defunct node
      throw new IllegalArgumentException("p is no longer in the list");
    return node;
  }

  /** Returns the given node as a Position (or null, if it is a sentinel). */
  private Position<E> position(Node<E> node) {
    if (node == header || node == trailer)
      return null;    // do not expose user to the sentinels
    return node;
  }
```

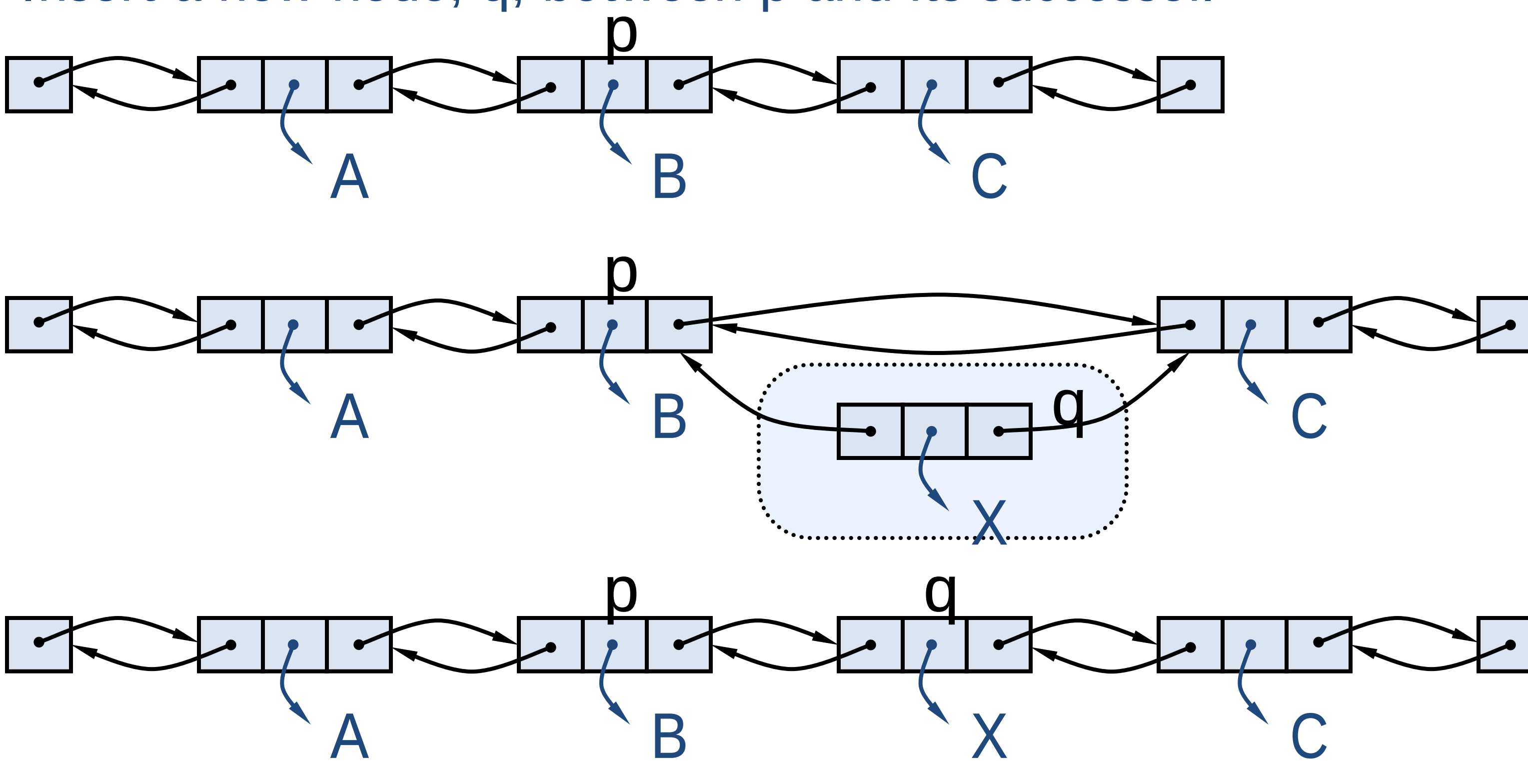
public accessor methods

```
// public accessor methods
/** Returns the number of elements in the linked list. */
public int size( ) { return size; }

/** Tests whether the linked list is empty. */
public boolean isEmpty( ) { return size == 0; }

/** Returns the first Position in the linked list (or null, if empty). */
public Position<E> first( ) {
  return position(header.getNext( ));
}

/** Returns the last Position in the linked list (or null, if empty). */
public Position<E> last( ) {
  return position(trailer.getPrev( ));
}

/** Returns the Position immediately before Position p (or null, if p is first). */
public Position<E> before(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return position(node.getPrev( ));
}

/** Returns the Position immediately after Position p (or null, if p is last). */
public Position<E> after(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return position(node.getNext( ));
}
```

Method	Running Time
size( )	$O(1)$
isEmpty( )	$O(1)$
first( ), last( )	$O(1)$
before(p), after(p)	$O(1)$

INSERTION

- Insert a new node, q, between p and its successor.



```
// private utilities
/** Adds element e to the linked list between the given nodes. */
private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
  Node<E> newest = new Node<>(e, pred, succ);  // create and link a new node
  pred.setNext(newest);
  succ.setPrev(newest);
  size++;
  return newest;
}
```

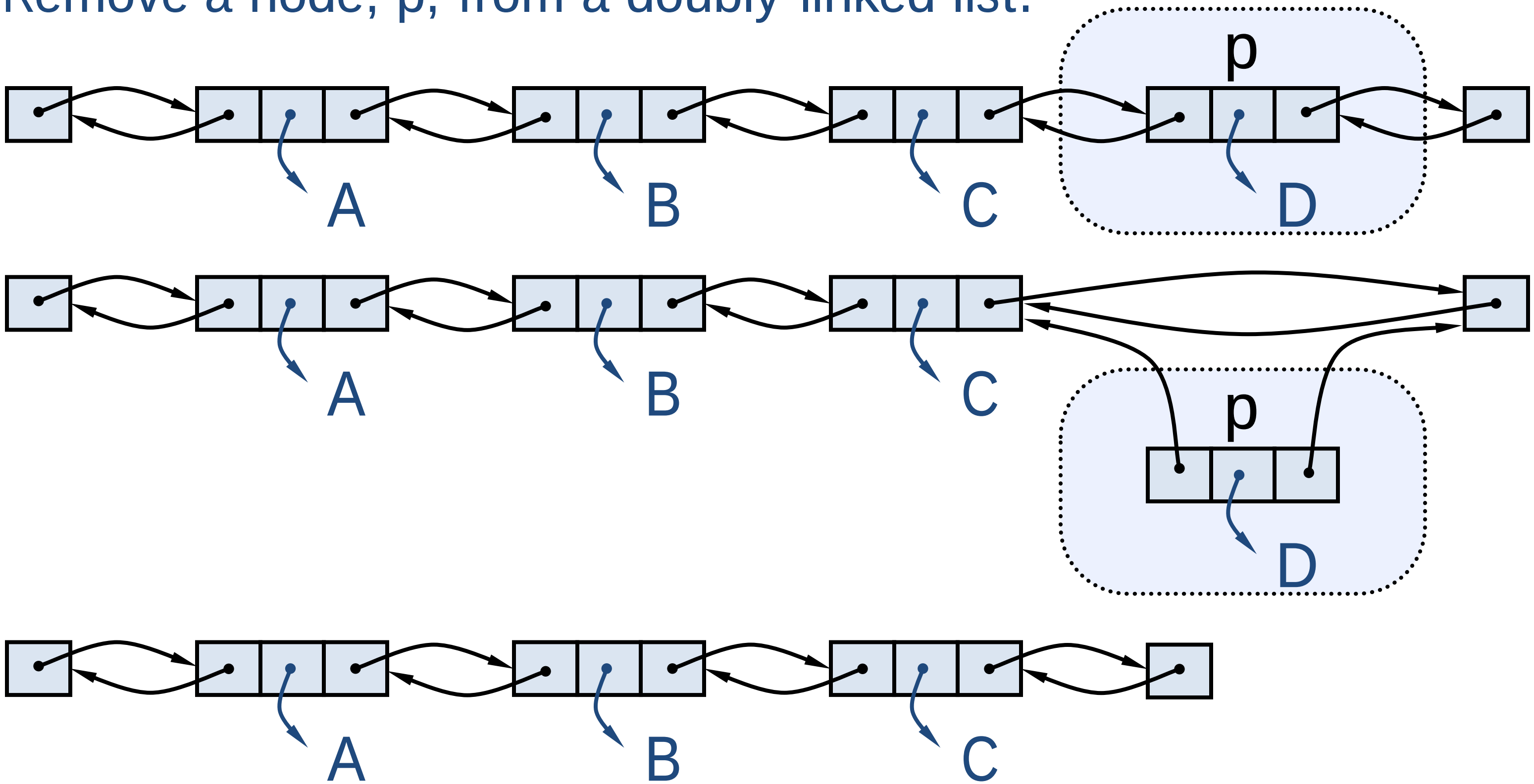
```java
// public update methods
/** Inserts element e at the front of the linked list and returns its new Position. */
public Position<E> addFirst(E e) {
  return addBetween(e, header, header.getNext());        // just after the header
}

/** Inserts element e at the back of the linked list and returns its new Position. */
public Position<E> addLast(E e) {
  return addBetween(e, trailer.getPrev(), trailer);        // just before the trailer
}

/** Inserts element e immediately before Position p, and returns its new Position.*/
public Position<E> addBefore(Position<E> p, E e)
                              throws IllegalArgumentException {
  Node<E> node = validate(p);
  return addBetween(e, node.getPrev(), node);
}

/** Inserts element e immediately after Position p, and returns its new Position. */
public Position<E> addAfter(Position<E> p, E e)
                              throws IllegalArgumentException {
  Node<E> node = validate(p);
  return addBetween(e, node, node.getNext());
}

/** Replaces the element stored at Position p and returns the replaced element. */
public E set(Position<E> p, E e) throws IllegalArgumentException {
  Node<E> node = validate(p);
  E answer = node.getElement();
  node.setElement(e);
  return answer;
}
```

public update methods, relying on a private **addBetween** method to unify the implementations of the various insertion operations.

addFirst(e), addLast(e)	$O(1)$
addBefore(p, e), addAfter(p, e)	$O(1)$
set(p, e)	$O(1)$

DELETION

- Remove a node, p, from a doubly-linked list.



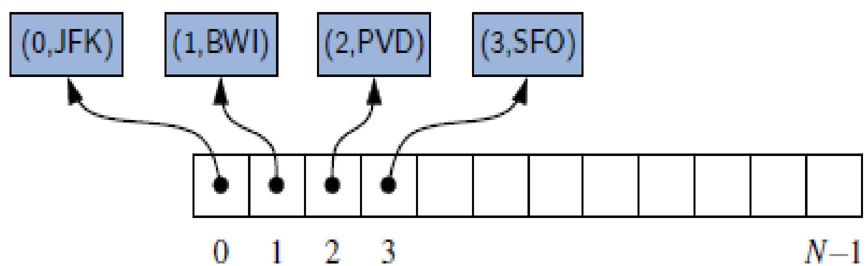
Public **remove** method. Note that it sets all fields of the removed node back to null—a condition we can later detect to recognize a defunct position.

```
/** Removes the element stored at Position p and returns it (invalidating p). */
public E remove(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  Node<E> predecessor = node.getPrev( );
  Node<E> successor = node.getNext( );
  predecessor.setNext(successor);
  successor.setPrev(predecessor);
  size−−;
  E answer = node.getElement( );
  node.setElement(null);              // help with garbage collection
  node.setNext(null);                 // and convention for defunct node
  node.setPrev(null);
  return answer;
 }
}
```

remove(p)	$O(1)$

IMPLEMENTING A POSITIONAL LIST WITH AN ARRAY

- The **problem** with using index number to keep track of an element: the index of an element *e* changes when other insertions or deletions occur before it.
- **Solution** approach: Instead of storing the elements of *L* directly in array *A*, store a new kind of position object in each cell of *A*. A position *p* stores the element *e* as well as the current index *i* of that element within the list.



- addFirst, addBefore, addAfter, and remove methods take $O(n)$ time

Iterators

- An **iterator** is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time.

hasNext(): Returns true if there is at least one additional element in the sequence, and false otherwise.

next(): Returns the next element in the sequence.

THE ITERABLE INTERFACE

- Java defines a parameterized interface, named Iterable, that includes the following single method:
  - iterator( ): Returns an iterator of the elements in the collection.
- An instance of a typical collection class in Java, such as an ArrayList, is iterable (but not itself an iterator); it produces an iterator for its collection as the return value of the iterator( ) method.
- Each call to iterator( ) returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.

THE FOR-EACH LOOP

- Java's Iterable class also plays a fundamental role in s upport of the "for-each" loop syntax:

```
for (ElementType variable : collection) {
  loopBody                              // may refer to "variable"
}
```

is equivalent to:

```
Iterator<ElementType> iter = collection.iterator();
while (iter.hasNext()) {
  ElementType variable = iter.next();
  loopBody                              // may refer to "variable"
}
```