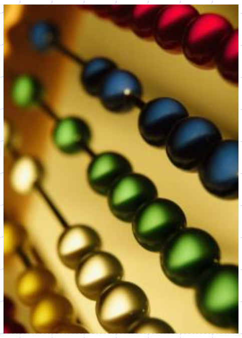
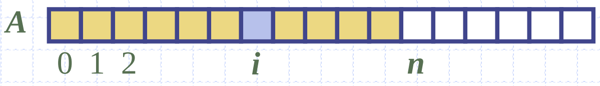
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

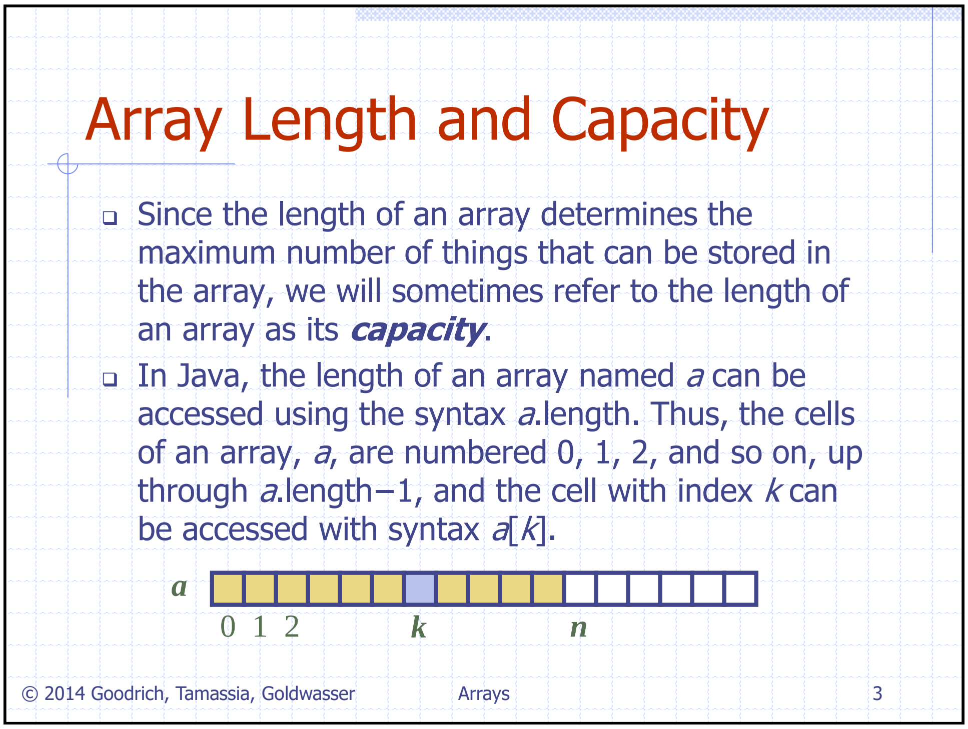
Arrays



Array Definition

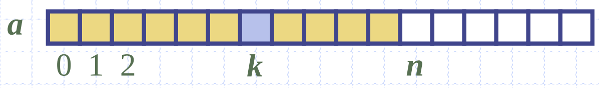
- An ***array*** is a sequenced collection of variables all of the same type. Each variable, or ***cell***, in an array has an ***index***, which uniquely refers to the value stored in that cell. The cells of an array, A, are numbered 0, 1, 2, and so on.
- Each value stored in an array is often called an ***element*** of that array.

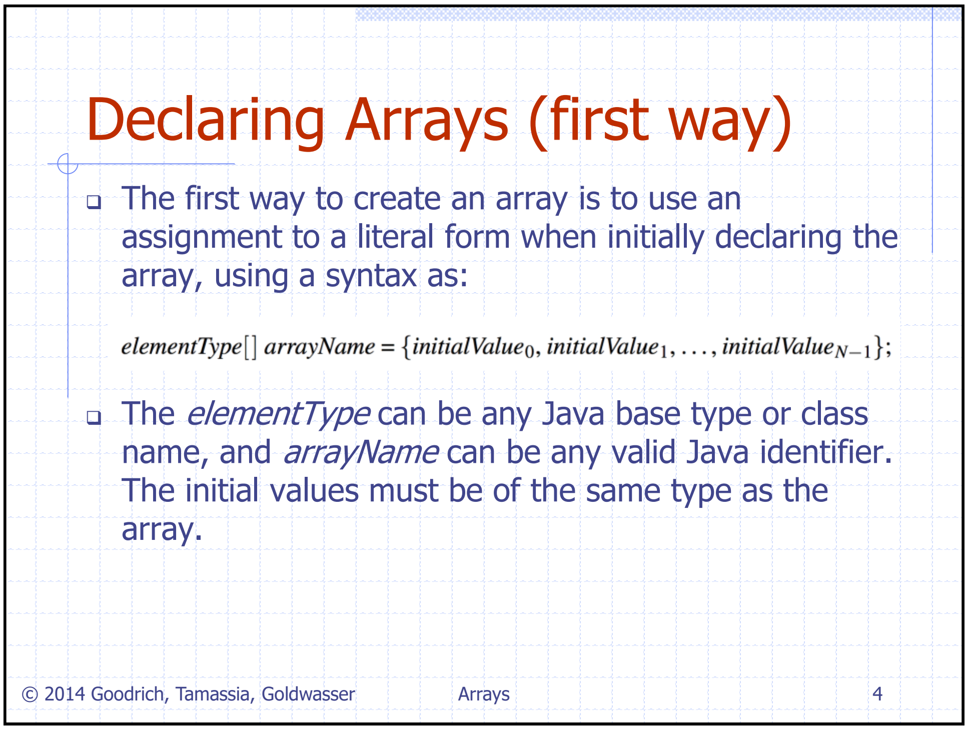




Array Length and Capacity

- Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its ***capacity***.
- In Java, the length of an array named *a* can be accessed using the syntax *a*.length. Thus, the cells of an array, *a*, are numbered 0, 1, 2, and so on, up through *a*.length−1, and the cell with index *k* can be accessed with syntax *a*[*k*].





Declaring Arrays (first way)

- The first way to create an array is to use an assignment to a literal form when initially declaring the array, using a syntax as:

$$elementType[\,]\ arrayName = \{initialValue_0, initialValue_1, \ldots, initialValue_{N-1}\};$$

- The *elementType* can be any Java base type or class name, and *arrayName* can be any valid Java identifier. The initial values must be of the same type as the array.

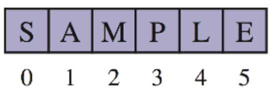
Declaring Arrays (second way)

- The second way to create an array is to use the **new** operator.
  - However, because an array is not an instance of a class, we do not use a typical constructor. Instead we use the syntax:

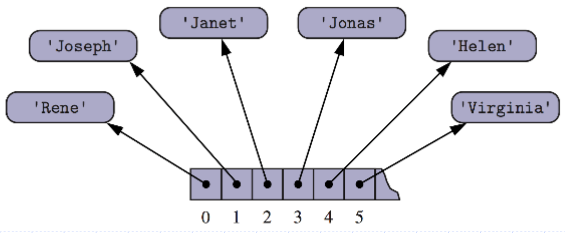
    new *elementType*[*length*]
- *length* is a positive integer denoting the length of the new array.
- The **new** operator returns a reference to the new array, and typically this would be assigned to an array variable.

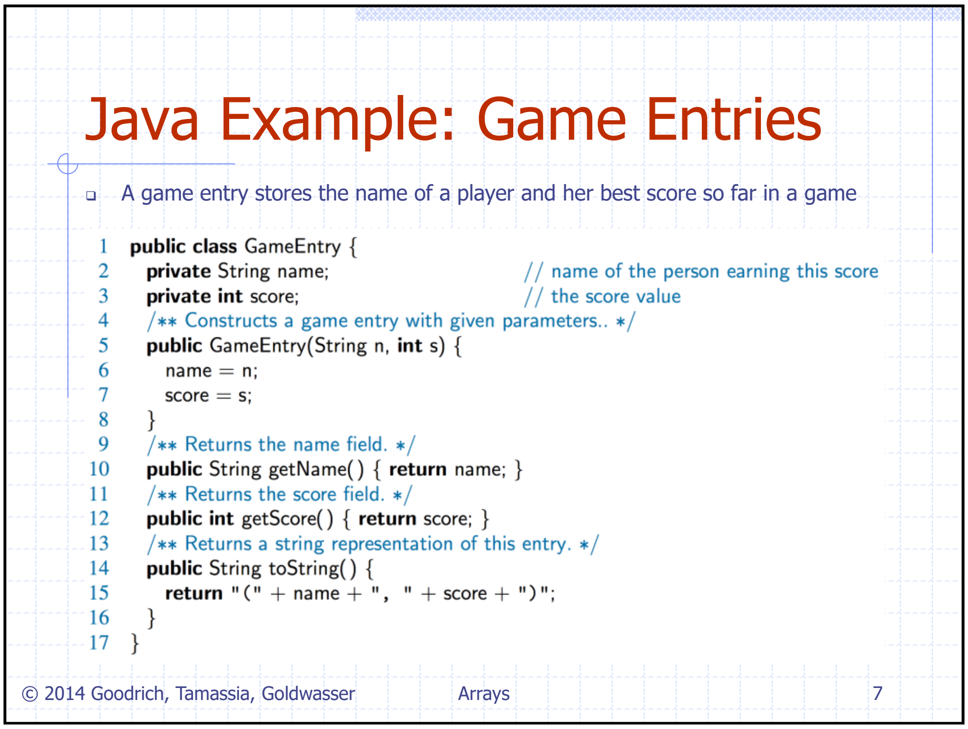
Arrays of Characters or Object References

- An array can store primitive elements, such as characters**.**



- An array can also store references to objects.

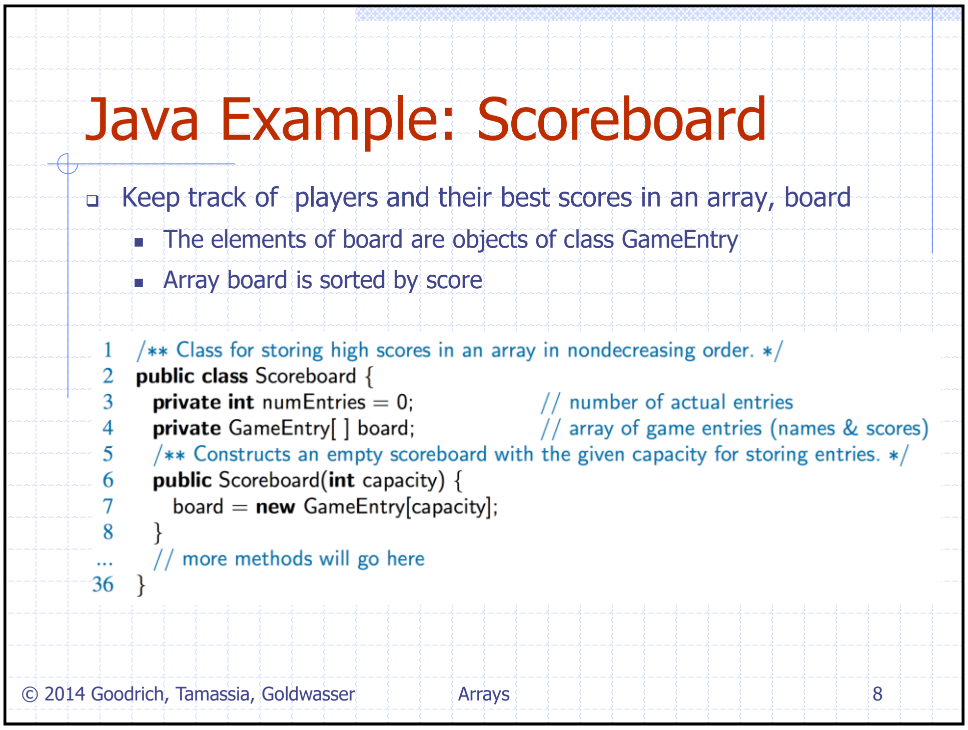




Java Example: Game Entries

- A game entry stores the name of a player and her best score so far in a game

```
public class GameEntry {
  private String name;                        // name of the person earning this score
  private int score;                          // the score value
  /** Constructs a game entry with given parameters.. */
  public GameEntry(String n, int s) {
    name = n;
    score = s;
  }
  /** Returns the name field. */
  public String getName( ) { return name; }
  /** Returns the score field. */
  public int getScore( ) { return score; }
  /** Returns a string representation of this entry. */
  public String toString( ) {
    return "(" + name + ", " + score + ")";
  }
}
```



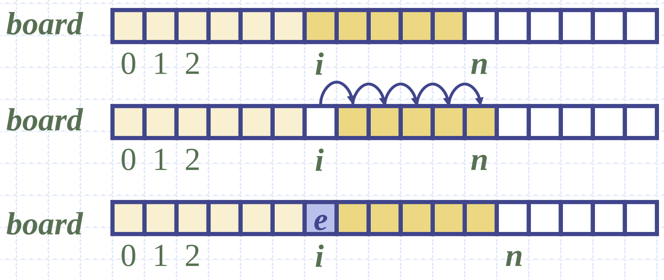
Java Example: Scoreboard

- Keep track of players and their best scores in an array, board
  - The elements of board are objects of class GameEntry
  - Array board is sorted by score

```
/** Class for storing high scores in an array in nondecreasing order. */
public class Scoreboard {
  private int numEntries = 0;              // number of actual entries
  private GameEntry[ ] board;              // array of game entries (names & scores)
  /** Constructs an empty scoreboard with the given capacity for storing entries. */
  public Scoreboard(int capacity) {
    board = new GameEntry[capacity];
  }
  // more methods will go here
}
```

Adding an Entry

- To add an entry e into array board at index i, we need to make room for it by shifting forward the ***n* – *i*** entries ***board***[***i***], …, ***board***[***n*** □ 1]

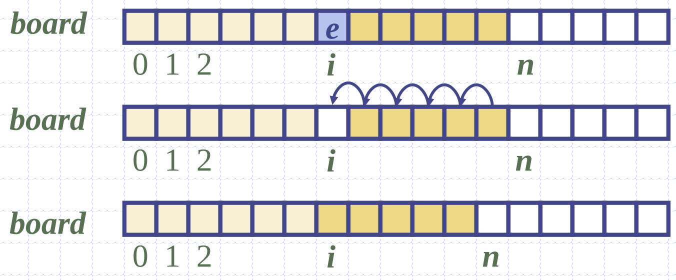


Java Example

```java
/** Attempt to add a new score to the collection (if it is high enough) */
public void add(GameEntry e) {
  int newScore = e.getScore();
  // is the new entry e really a high score?
  if (numEntries < board.length || newScore > board[numEntries−1].getScore()) {
    if (numEntries < board.length)          // no score drops from the board
      numEntries++;                         // so overall number increases
    // shift any lower scores rightward to make room for the new entry
    int j = numEntries − 1;
    while (j > 0 && board[j−1].getScore() < newScore) {
      board[j] = board[j−1];                // shift entry from j-1 to j
      j−−;                                  // and decrement j
    }
    board[j] = e;                           // when done, add new entry
  }
}
```

Removing an Entry

- To remove the entry e at index i, we need to fill the hole left by e by shifting backward the $n - i - 1$ elements ***board***[***i*** + 1], …, ***board***[***n*** □ 1]

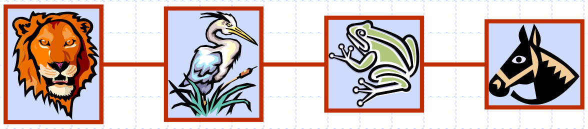


Java Example

```
/** Remove and return the high score at index i. */
public GameEntry remove(int i) throws IndexOutOfBoundsException {
  if (i < 0 || i >= numEntries)
    throw new IndexOutOfBoundsException("Invalid index: " + i);
  GameEntry temp = board[i];                      // save the object to be removed
  for (int j = i; j < numEntries − 1; j++)        // count up from i (not down)
    board[j] = board[j+1];                        // move one cell to the left
  board[numEntries −1 ] = null;                   // null out the old last score
  numEntries−−;
  return temp;                                    // return the removed object
}
```

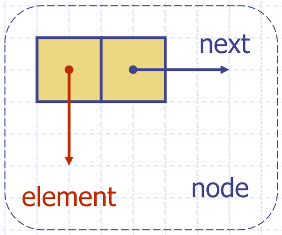
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

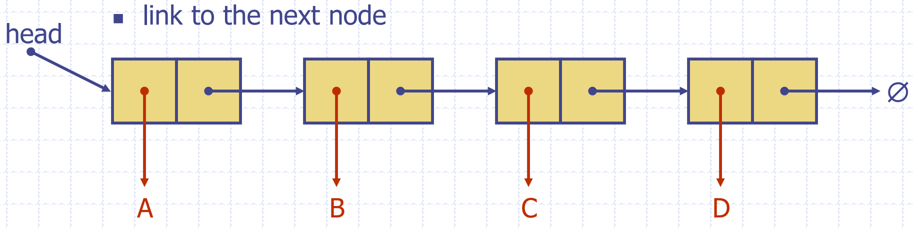
Singly Linked Lists



Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node



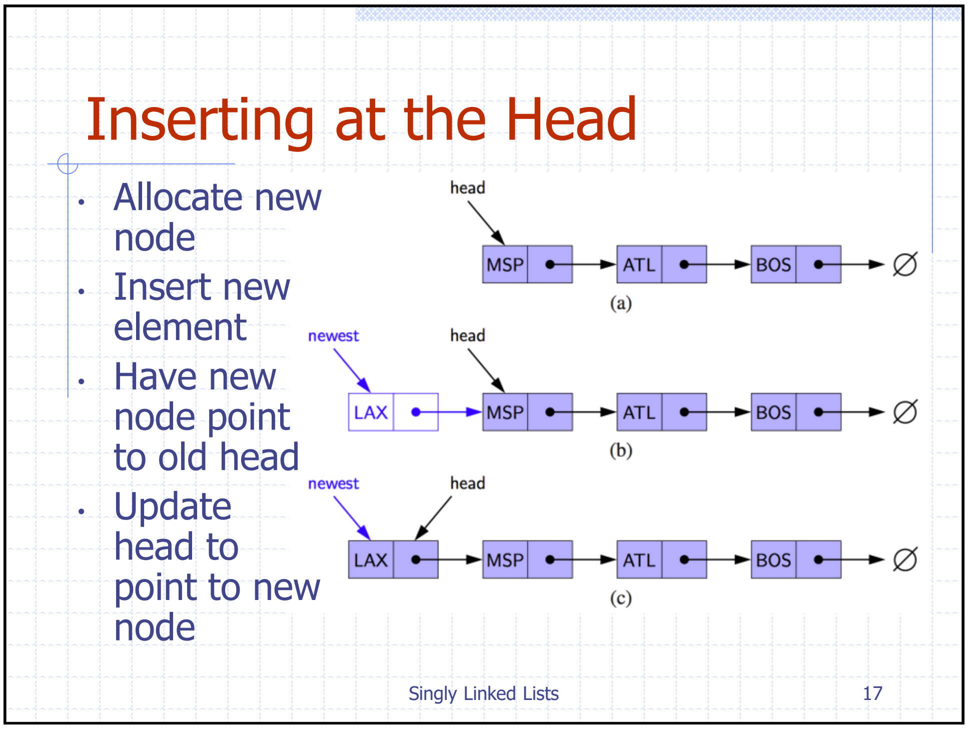


A Nested Node Class

```
public class SinglyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;              // reference to the element stored at this node
    private Node<E> next;           // reference to the subsequent node in the list
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }
    public E getElement( ) { return element; }
    public Node<E> getNext( ) { return next; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------
  ... rest of SinglyLinkedList class will follow ...
```

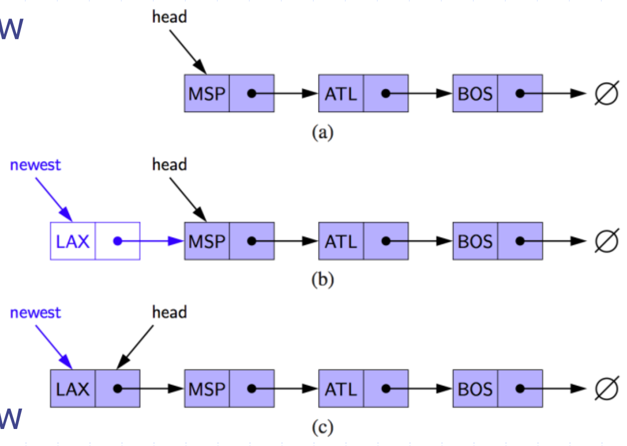
Accessor Methods

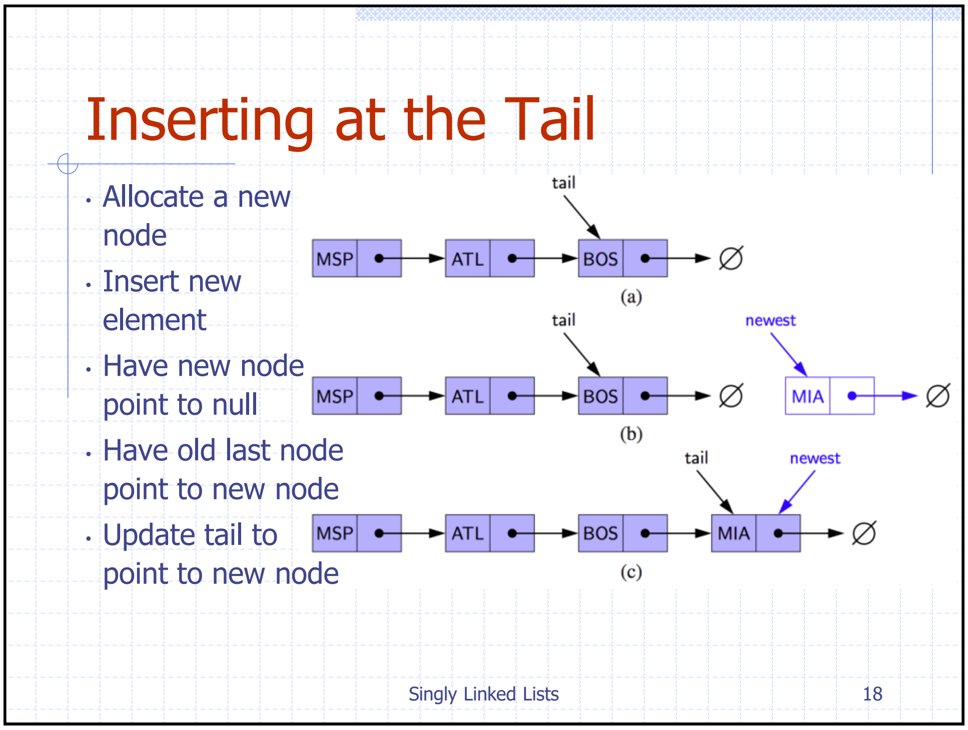
```
public class SinglyLinkedList<E> {
...  (nested Node class goes here)
  // instance variables of the SinglyLinkedList
  private Node<E> head = null;          // head node of the list (or null if empty)
  private Node<E> tail = null;          // last node of the list (or null if empty)
  private int size = 0;                 // number of nodes in the list
  public SinglyLinkedList( ) { }        // constructs an initially empty list
  // access methods
  public int size( ) { return size; }
  public boolean isEmpty( ) { return size == 0; }
  public E first( ) {               // returns (but does not remove) the first element
    if (isEmpty( )) return null;
    return head.getElement( );
  }
  public E last( ) {                // returns (but does not remove) the last element
    if (isEmpty( )) return null;
    return tail.getElement( );
  }
```



Inserting at the Head

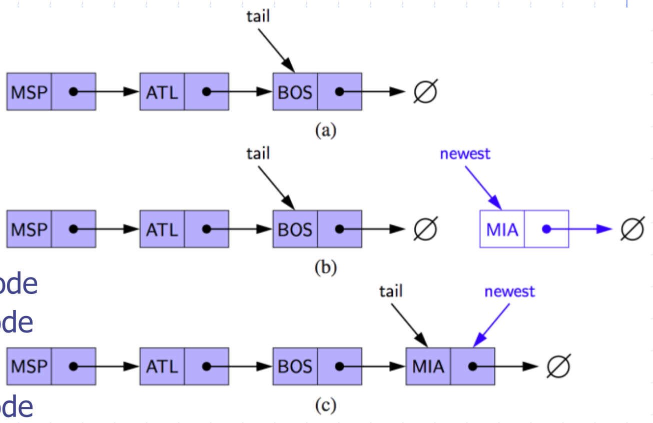
- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node





Inserting at the Tail

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node

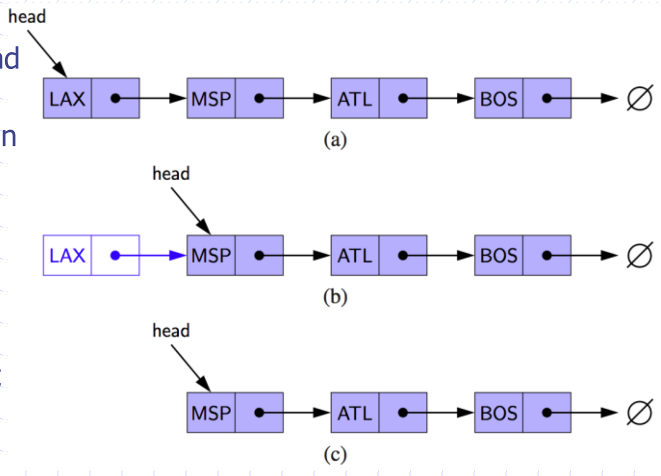


Java Methods

```
public void addFirst(E e) {                 // adds element e to the front of the list
  head = new Node<>(e, head);               // create and link a new node
  if (size == 0)
    tail = head;                            // special case: new node becomes tail also
  size++;
}
public void addLast(E e) {                  // adds element e to the end of the list
  Node<E> newest = new Node<>(e, null);     // node will eventually be the tail
  if (isEmpty())
    head = newest;                          // special case: previously empty list
  else
    tail.setNext(newest);                   // new node after existing tail
  tail = newest;                            // new node becomes the tail
  size++;
}
```

Removing at the Head

- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node

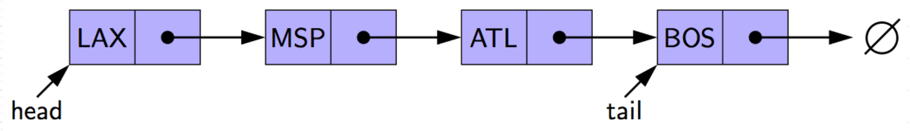


Java Method

```
  public E removeFirst( ) {                  // removes and returns the first element
    if (isEmpty( )) return null;             // nothing to remove
    E answer = head.getElement( );
    head = head.getNext( );                  // will become null if list had only one node
    size−−;
    if (size == 0)
      tail = null;                           // special case as list is now empty
    return answer;
  }
}
```

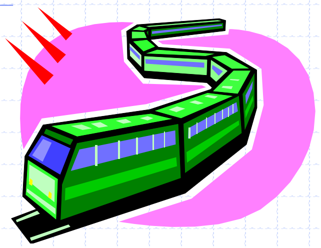
Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



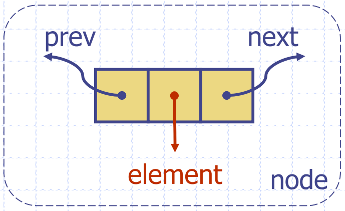
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

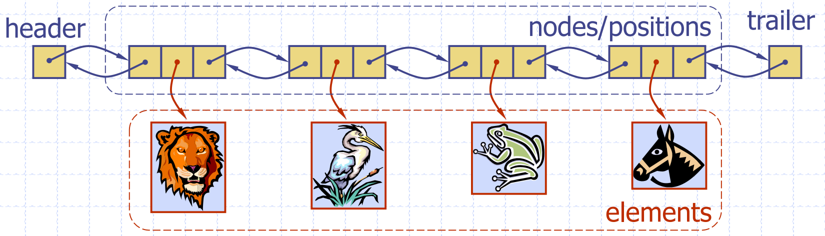
Doubly Linked Lists



Doubly Linked List

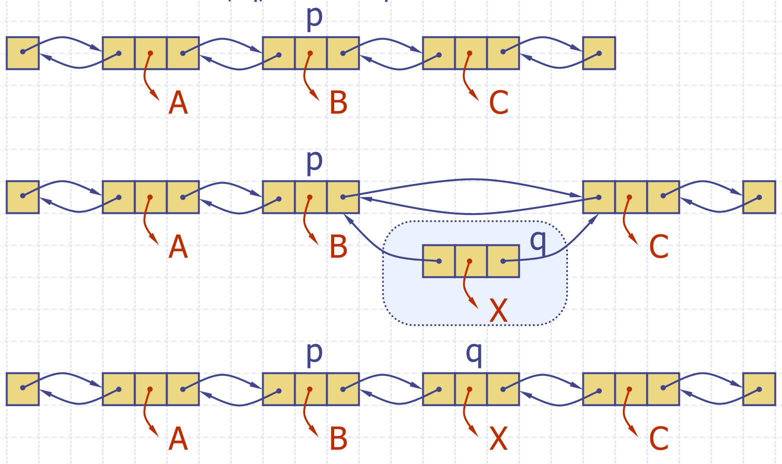
- A doubly linked list can be traversed forward and backward
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





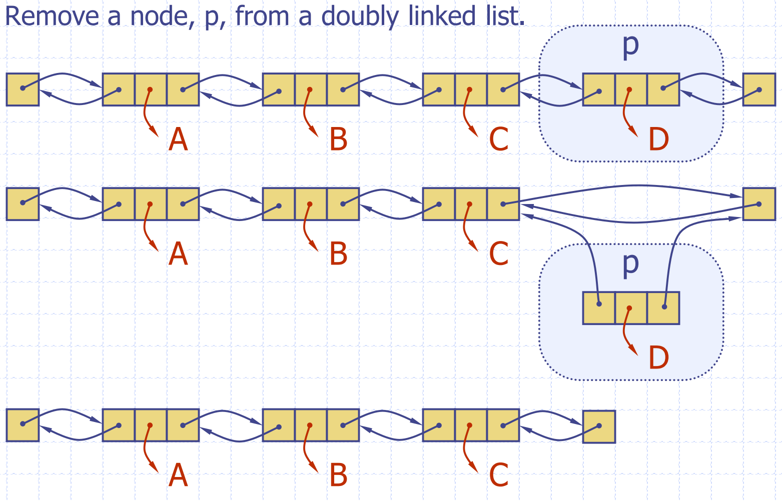
Insertion

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



Deletion

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



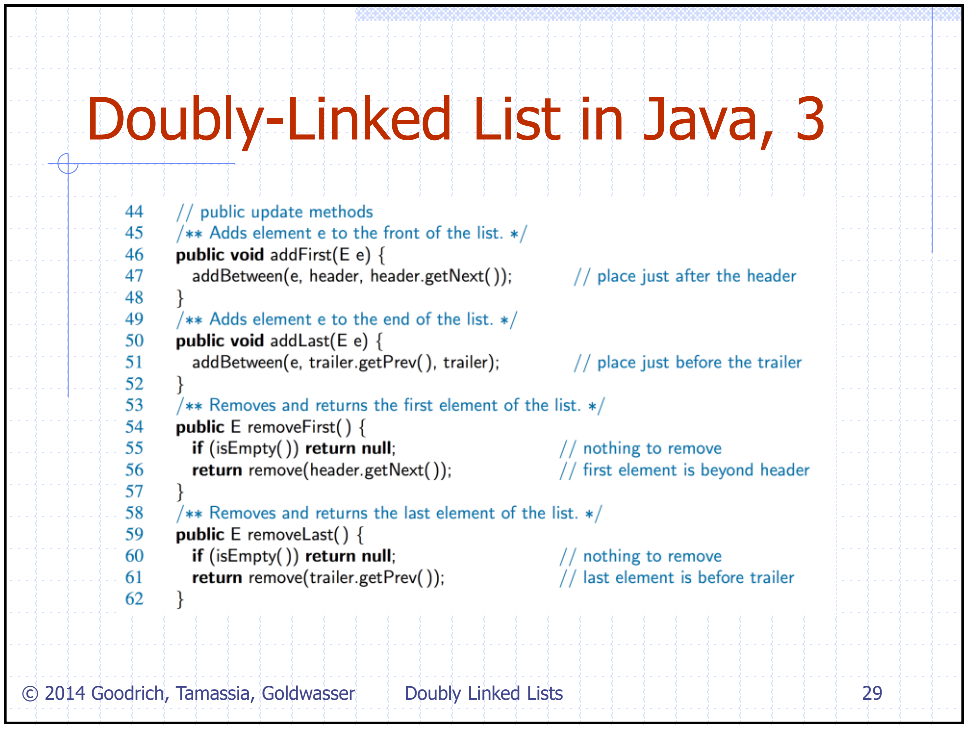
Doubly-Linked List in Java

```
/** A basic doubly linked list implementation. */
public class DoublyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<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( ) { return element; }
    public Node<E> getPrev( ) { return prev; }
    public Node<E> getNext( ) { return next; }
    public void setPrev(Node<E> p) { prev = p; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------

```

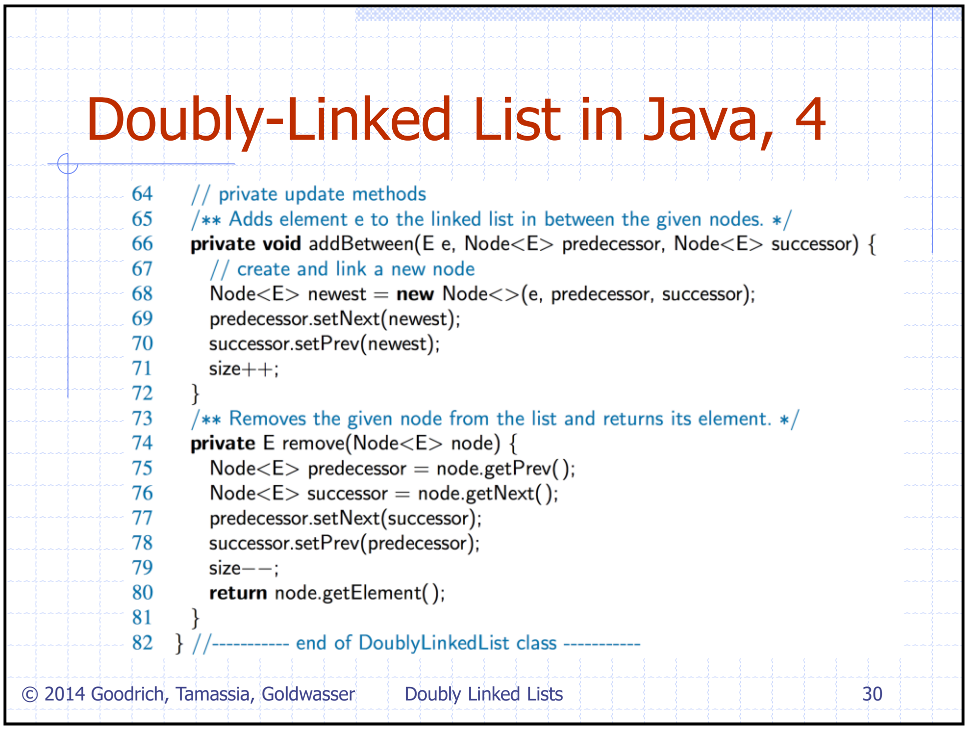
Doubly-Linked List in Java, 2

```
  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 DoublyLinkedList( ) {
    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
  }
  /** 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 (but does not remove) the first element of the list. */
  public E first( ) {
    if (isEmpty( )) return null;
    return header.getNext( ).getElement( );      // first element is beyond header
  }
  /** Returns (but does not remove) the last element of the list. */
  public E last( ) {
    if (isEmpty( )) return null;
    return trailer.getPrev( ).getElement( );     // last element is before trailer
  }
```



Doubly-Linked List in Java, 3

```
// public update methods
/** Adds element e to the front of the list. */
public void addFirst(E e) {
  addBetween(e, header, header.getNext());          // place just after the header
}
/** Adds element e to the end of the list. */
public void addLast(E e) {
  addBetween(e, trailer.getPrev(), trailer);        // place just before the trailer
}
/** Removes and returns the first element of the list. */
public E removeFirst() {
  if (isEmpty()) return null;                      // nothing to remove
  return remove(header.getNext());                 // first element is beyond header
}
/** Removes and returns the last element of the list. */
public E removeLast() {
  if (isEmpty()) return null;                      // nothing to remove
  return remove(trailer.getPrev());                // last element is before trailer
}
```



Doubly-Linked List in Java, 4

```
  // private update methods
  /** Adds element e to the linked list in between the given nodes. */
  private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
    // create and link a new node
    Node<E> newest = new Node<>(e, predecessor, successor);
    predecessor.setNext(newest);
    successor.setPrev(newest);
    size++;
  }
  /** Removes the given node from the list and returns its element. */
  private E remove(Node<E> node) {
    Node<E> predecessor = node.getPrev();
    Node<E> successor = node.getNext();
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size--;
    return node.getElement();
  }
} //----------- end of DoublyLinkedList class -----------
```

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

Lists and Iterators

The java.util.List ADT

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

size( ):	Returns the number of elements in the list.
isEmpty( ):	Returns a boolean indicating whether the list is empty.
get(i):	Returns the element of the list having index i; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.
set(i, e):	Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.
add(i, e):	Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range $[0, \text{size}()]$.
remove(i):	Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

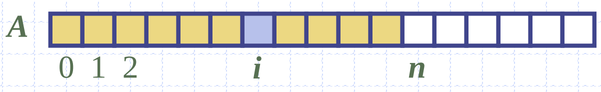
Example

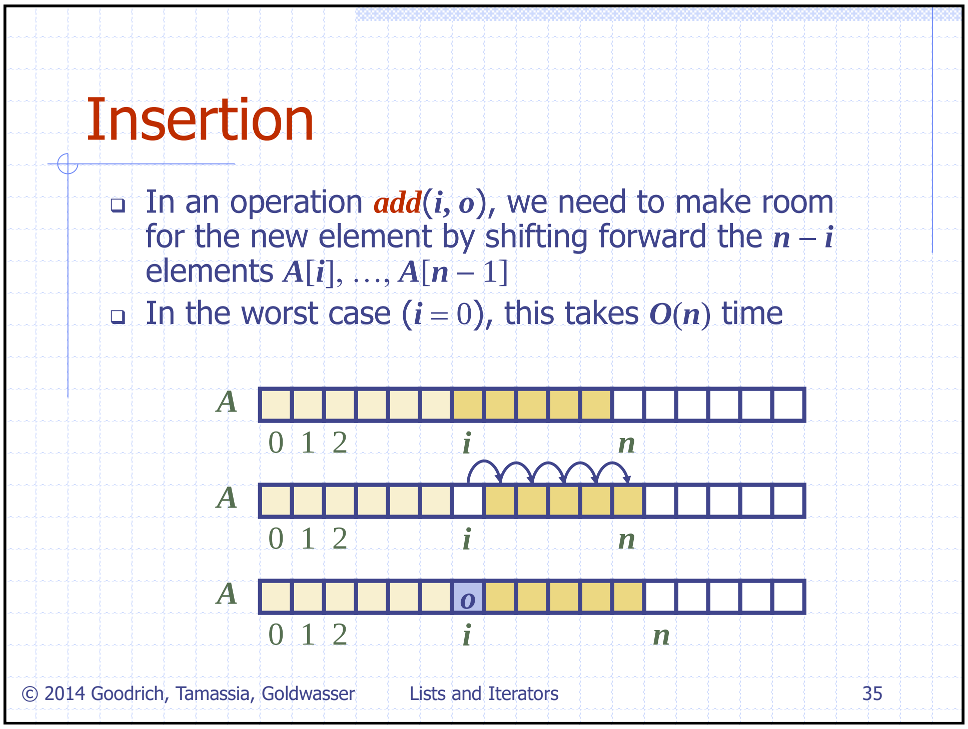
- A sequence of List operations:

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

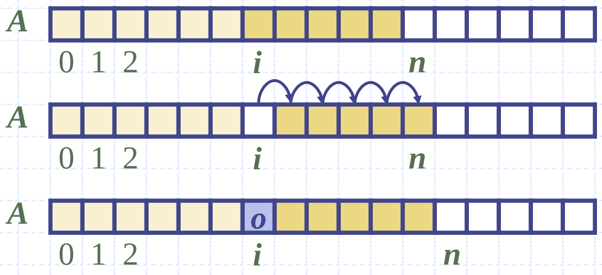
- An obvious choice for implementing the list ADT is to use an array, **A**, where **A[i]** stores (a reference to) the element with index **i.**
- With a representation based on an array **A**, the get(**i**) and set(**i**, **e**) methods are easy to implement by accessing **A[i]** (assuming **i** is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

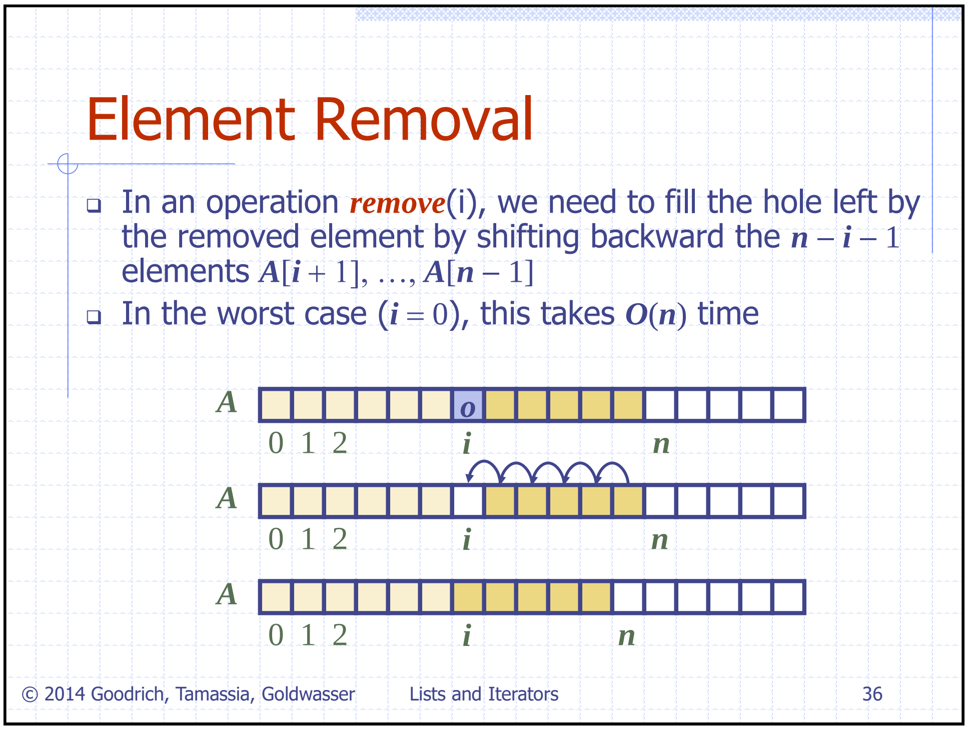




Insertion

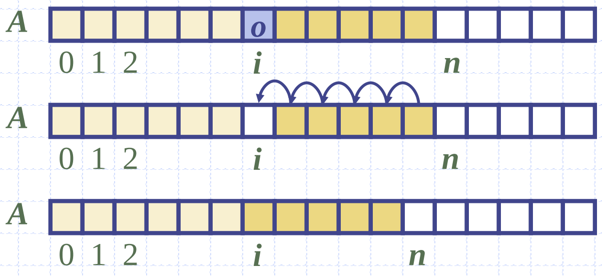
- In an operation ***add***(i, o), we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \dots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time





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 $A[i+1], \dots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time

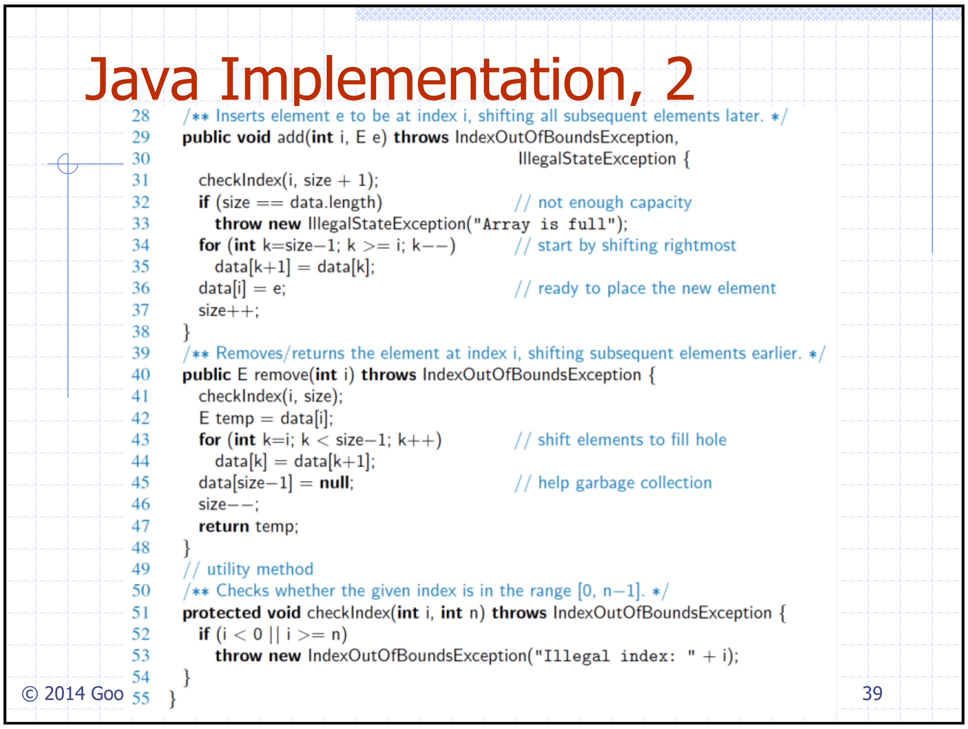


Performance

- In an array-based implementation of a dynamic 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 …

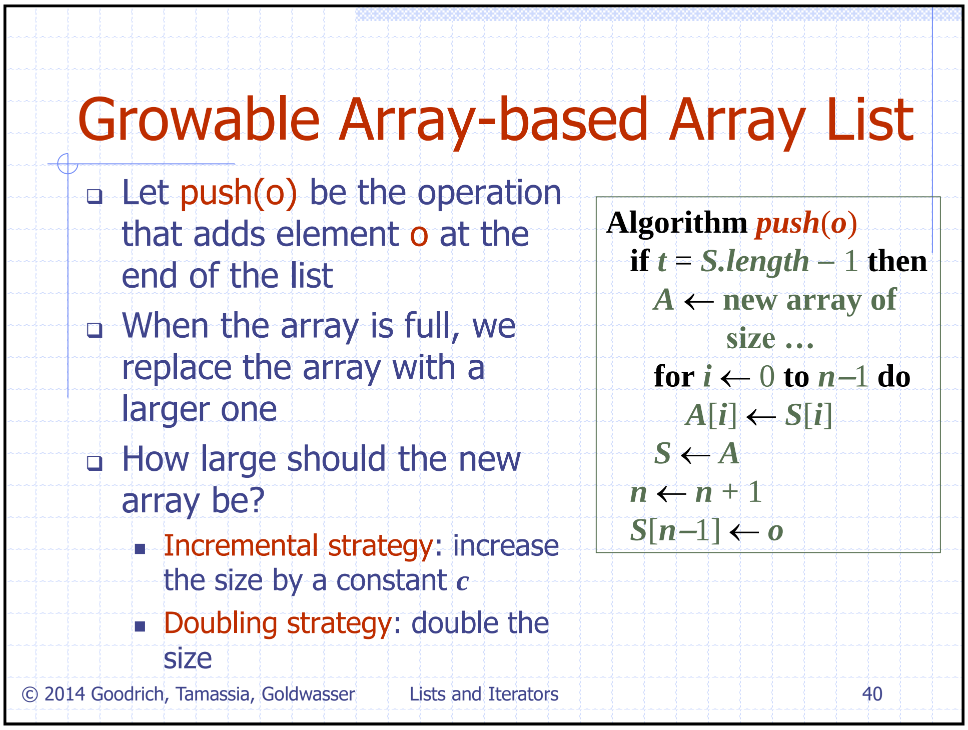
Java Implementation

```
// 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, 2

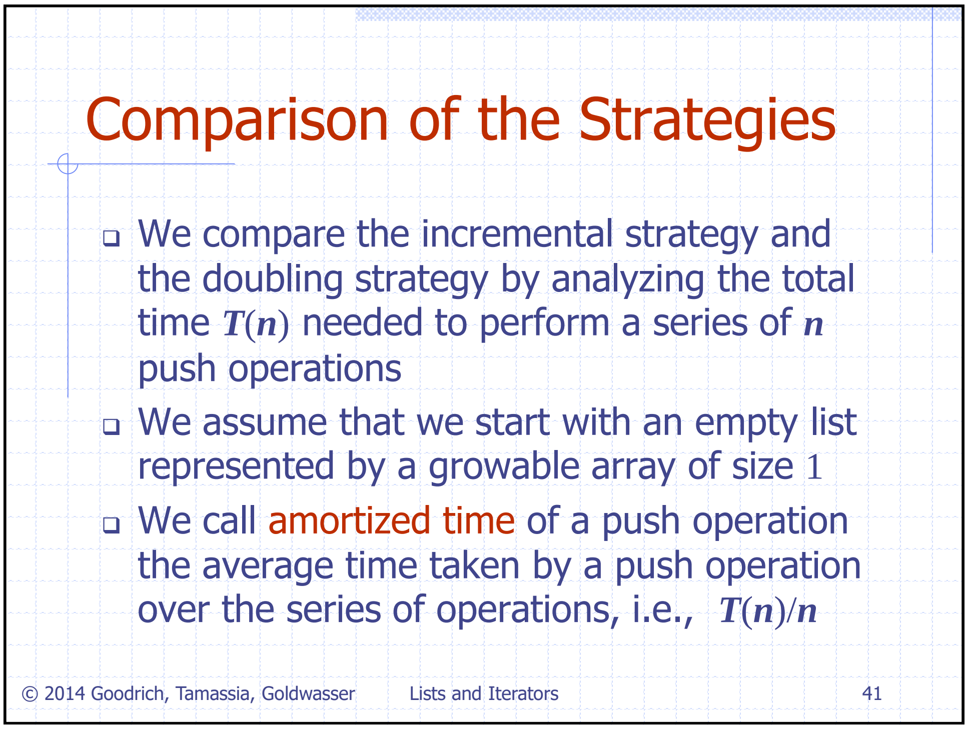
```
/** Inserts element e to be at index i, shifting all subsequent elements later. */
public void add(int i, E e) throws IndexOutOfBoundsException,
                                   IllegalStateException {
  checkIndex(i, size + 1);
  if (size == data.length)                // not enough capacity
    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);
}
}
```



Growable Array-based Array List

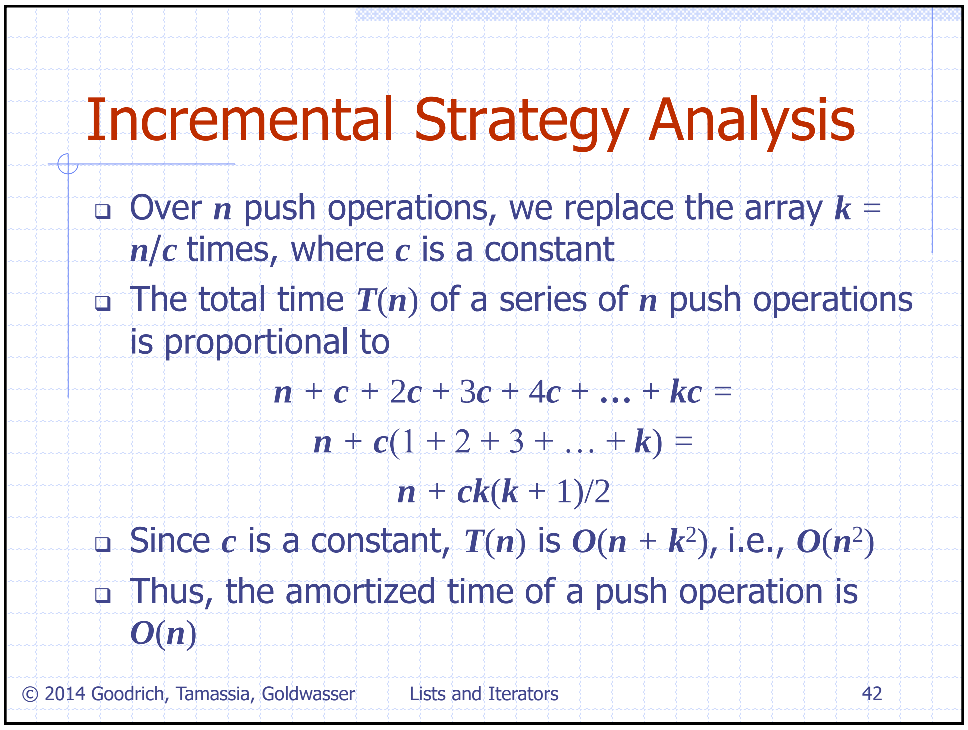
- Let push(o) be the operation that adds element o at the end of the list
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

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



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
- 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

$$n + c + 2c + 3c + 4c + \dots + kc =$$
$$n + c(1 + 2 + 3 + \dots + k) =$$
$$n + ck(k + 1)/2$$

- 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 $O(n)$

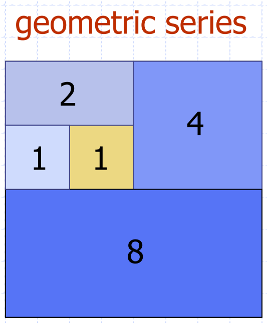
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

  $n + 1 + 2 + 4 + 8 + \ldots + 2^k =$

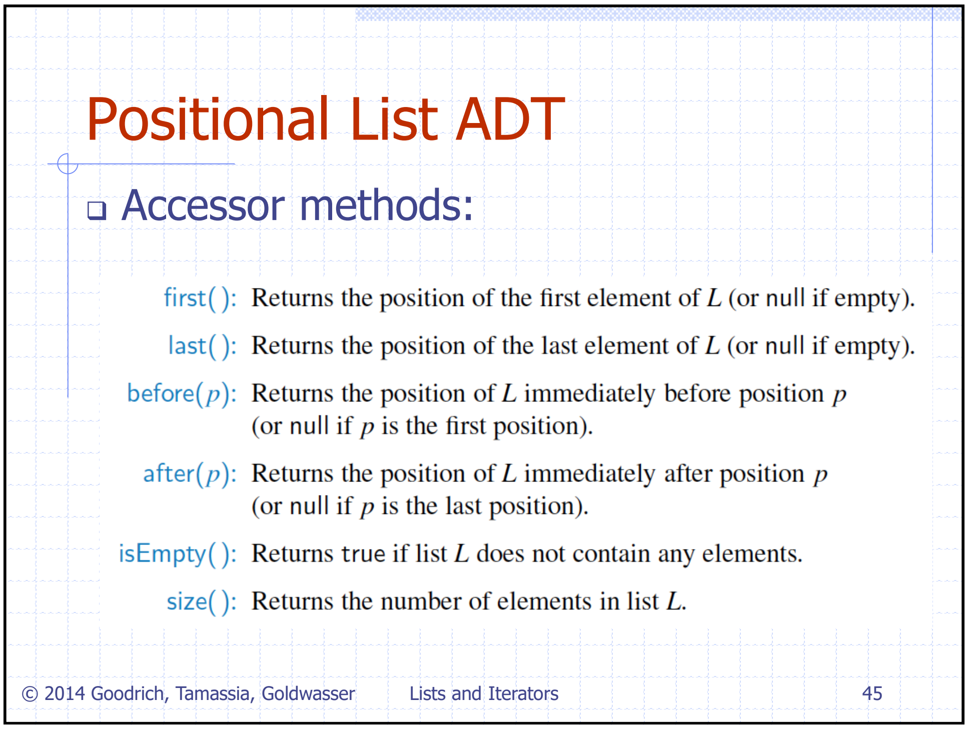
  $n + 2^{k+1} - 1 =$

  $3n - 1$
- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$



Positional Lists

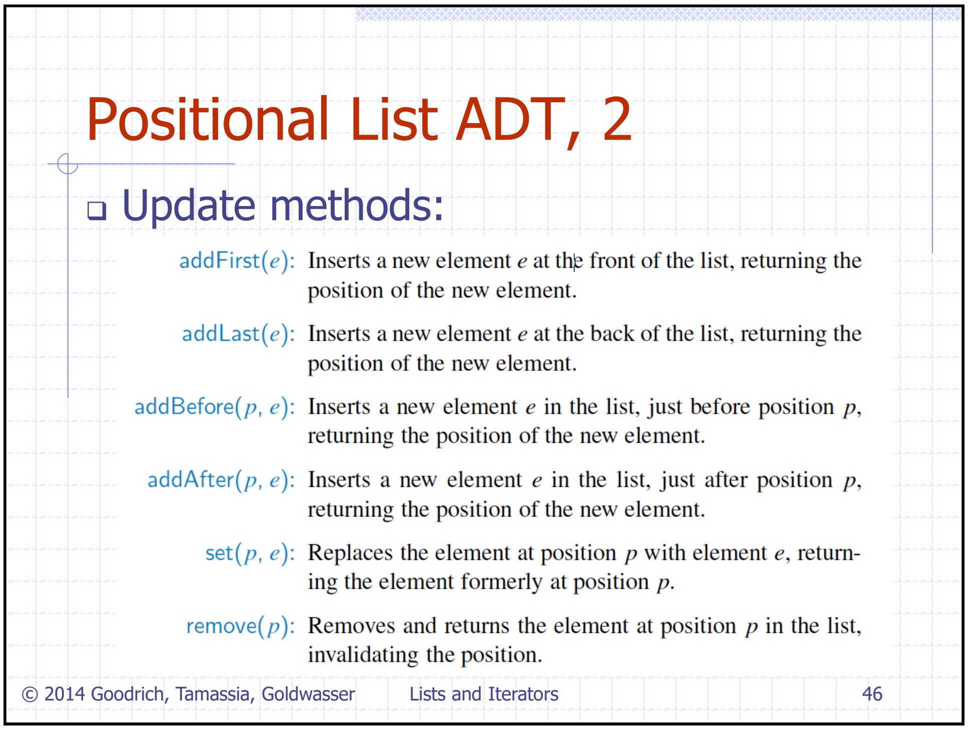
- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a **positional list** ADT.
- A position acts as a marker or token within the broader positional list.
- A position p is unaffected by changes elsewhere in a list; the only 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.



Positional List ADT

- Accessor methods:

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.

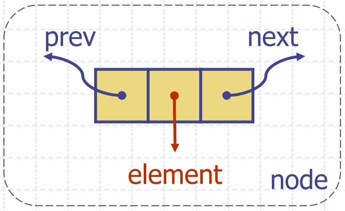
Example

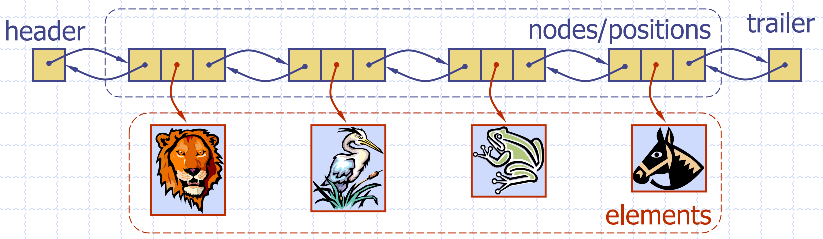
- A sequence of Positional List operations:

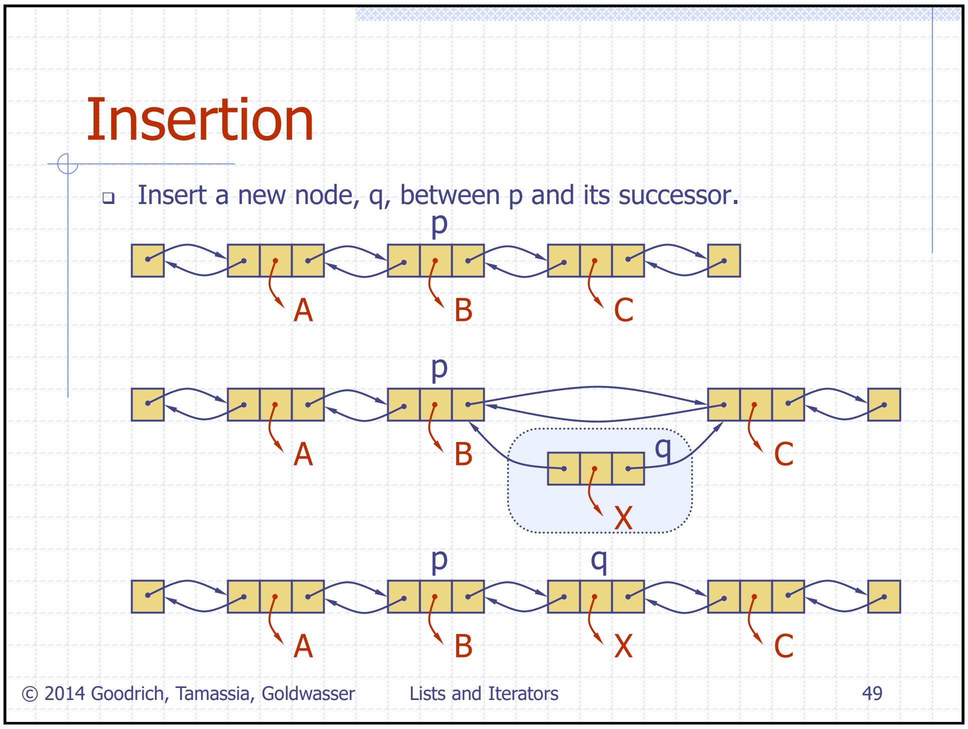
Method	Return Value	List Contents
addLast(8)	p	(8_p)
first()	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement()	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last())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

Positional List Implementation

- The most natural way to implement a positional list is with a doubly-linked list.

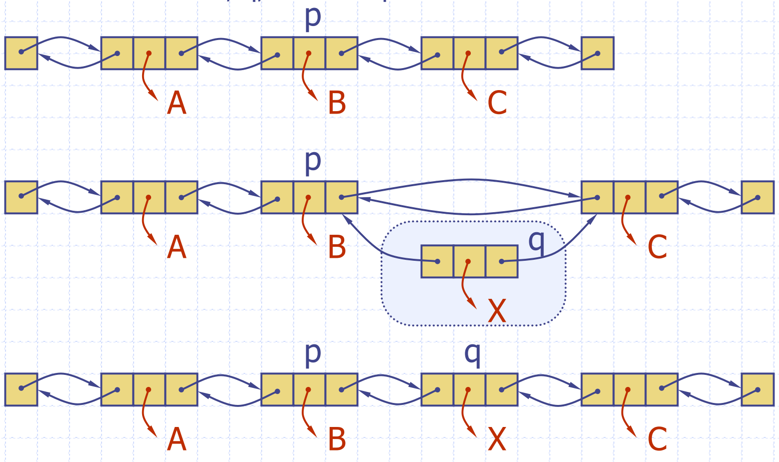


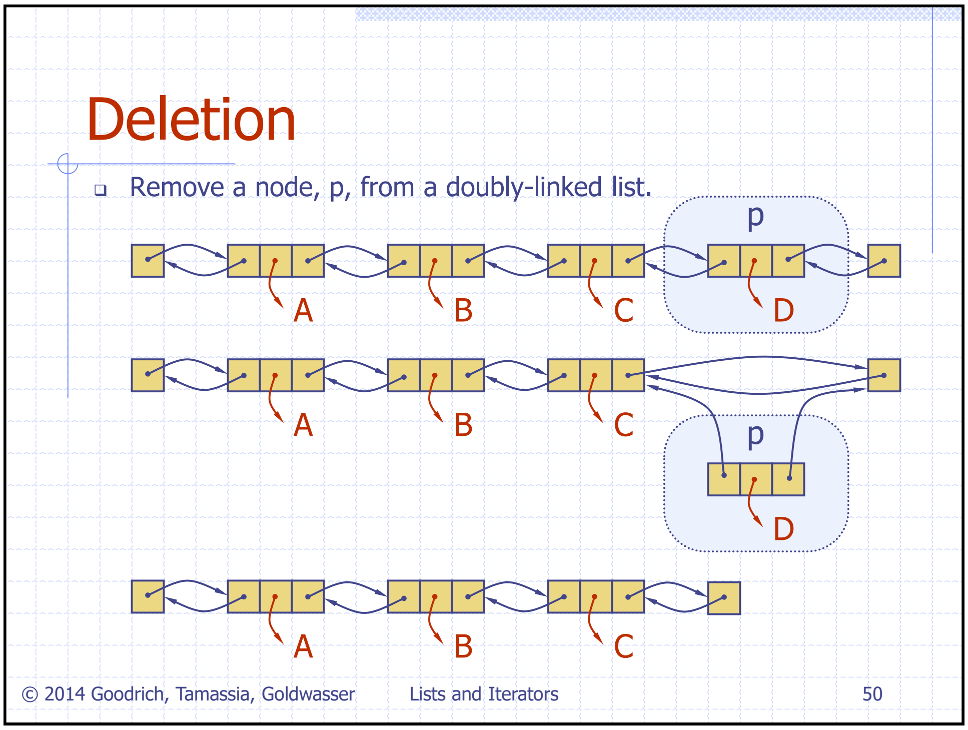




Insertion

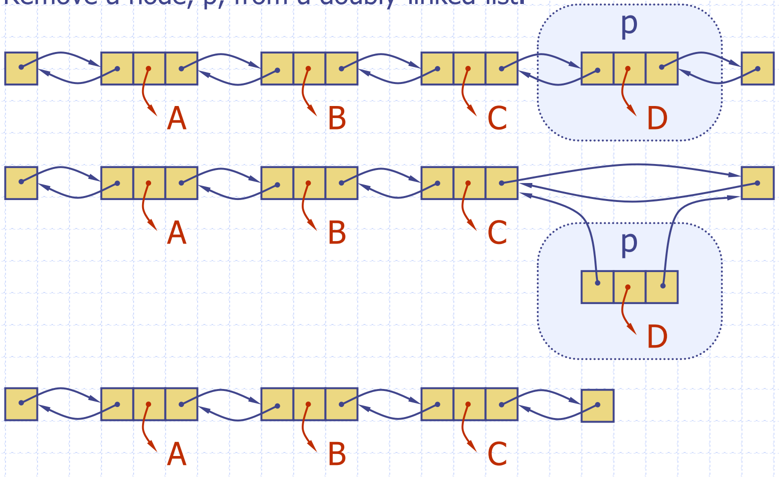
- Insert a new node, q, between p and its successor.

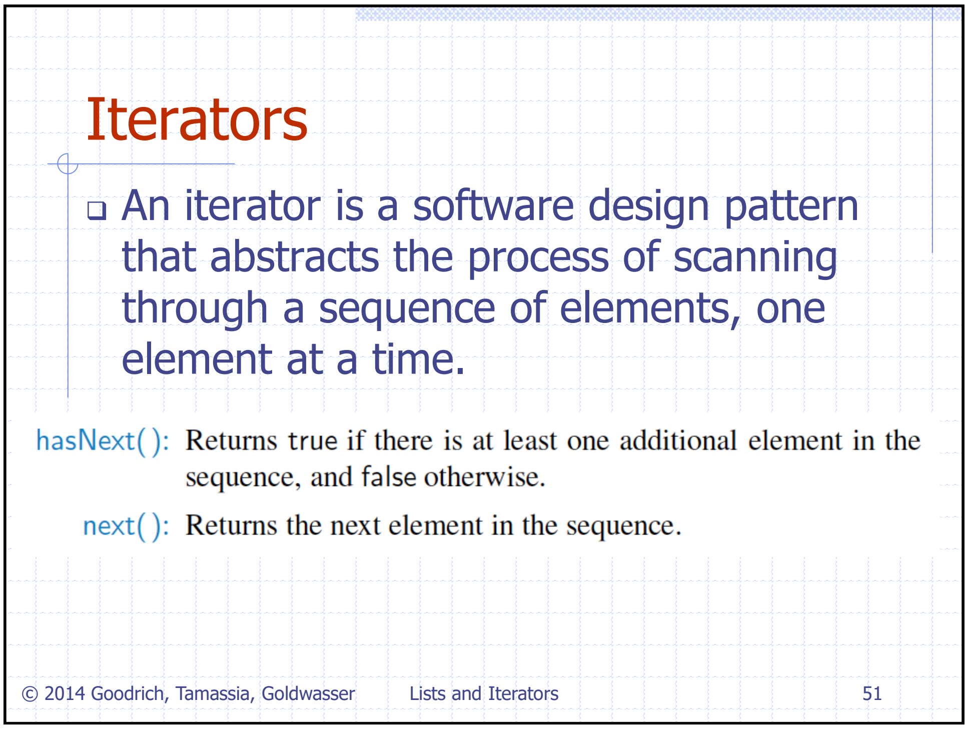




Deletion

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

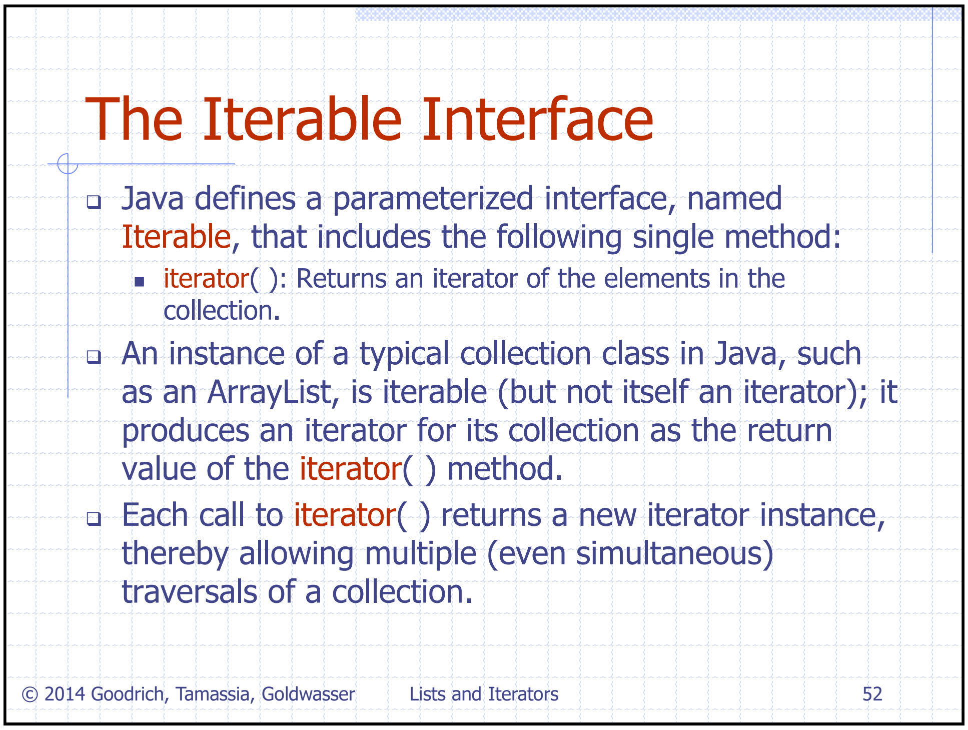




Iterators

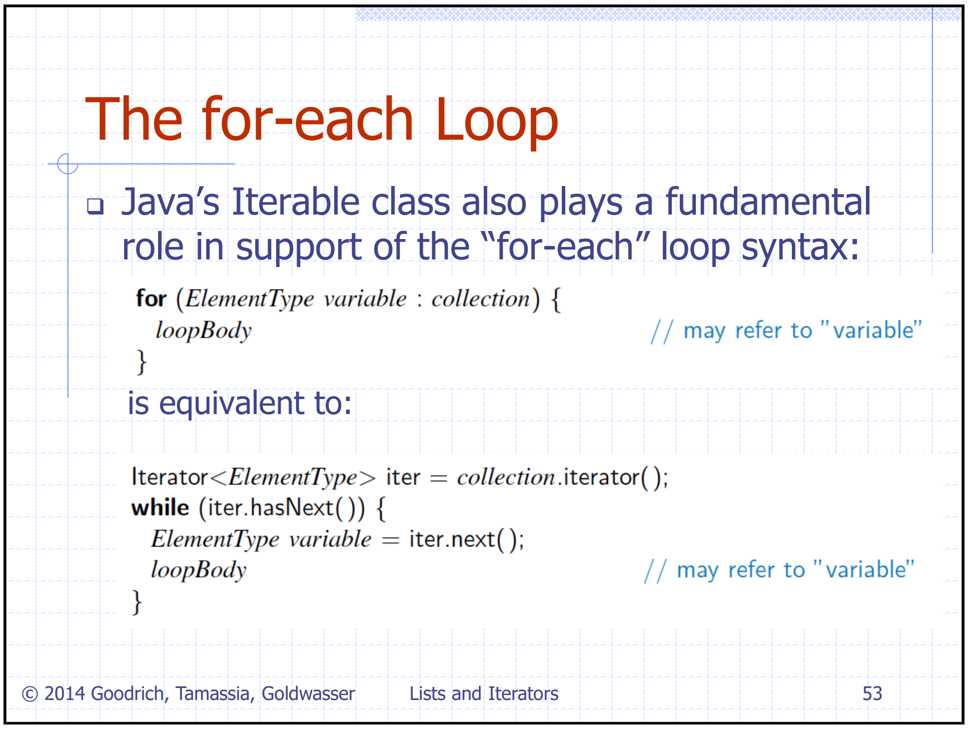
- An iterator is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time.

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 support 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"
}
```