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

Chapter 7 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)		
add(1, E)		
get(4)		
add(4, F)		
set(2, G)		
get(2)		

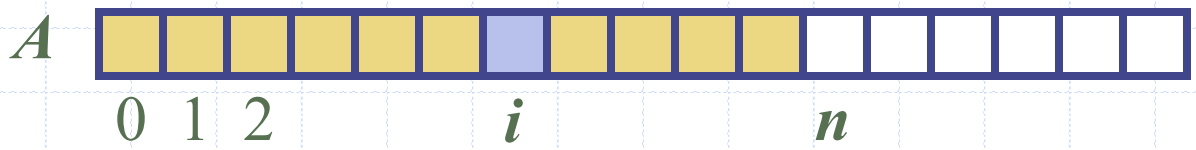
Example

- 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)

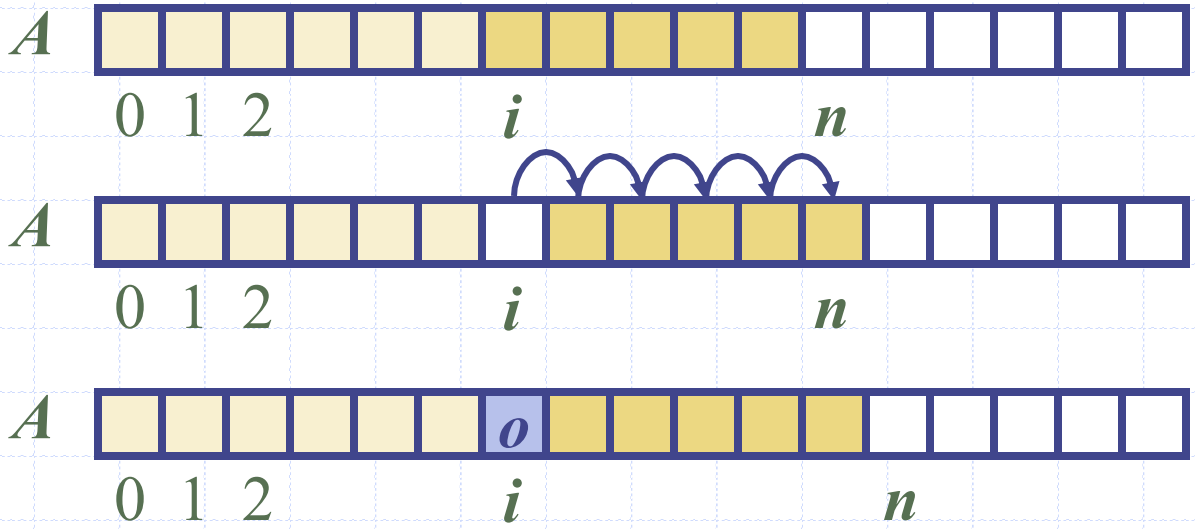
Array Lists

- 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).



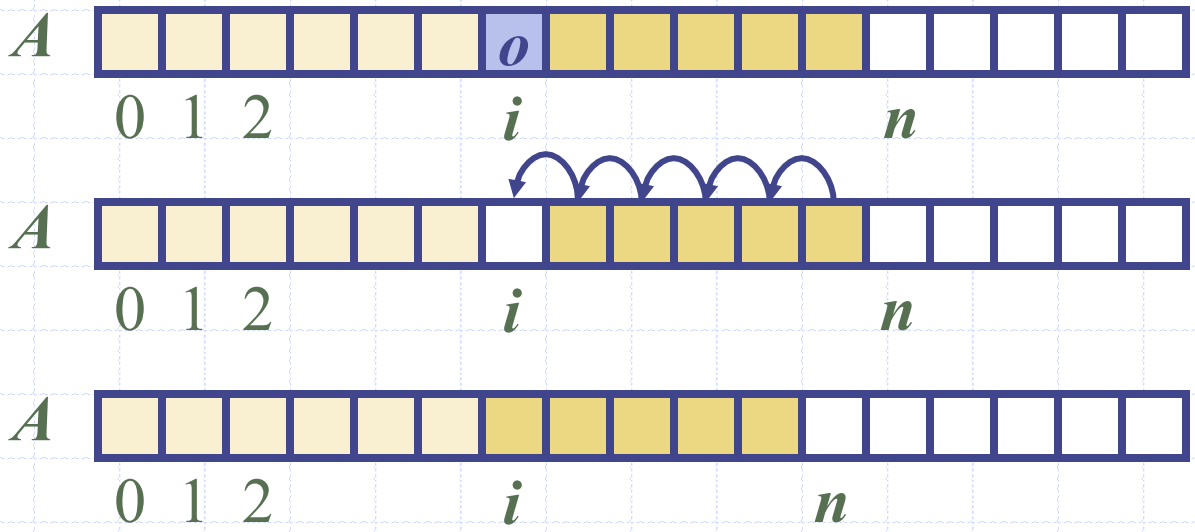
Insertion

- In an operation $add(i, o)$, we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- 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 $A[i + 1], \ldots, A[n - 1]$
- 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

Performance of Array-based List

- Performance of an array list with n elements realized by a fixed-capacity array.

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)	$O(n)$

Advantages of Array-based list

- it is straightforward to store or retrieve element with index (get(*i*) and set(*i*, *e*) methods).
- For array-based list, memory allocation is done once in the constructor.
- Node-based data structures, e.g., linked lists, require $\Theta(n)$ calls to new, which call the OS requesting a memory allocation.

Weakness of Array-Based List

- Methods add(*i*, *e*) and remove(*i*) are time consuming.
- A typical array has a fixed-size, not allowing to grow beyond maximum capacity.
- Without knowing the maximum size, there is risk of requesting too large or too small an array, causing waste of memory or fatal insertion error.

Adding to a full array…

- In an ***add*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one …

Solution to Array List: Dynamic Arrays

- An array list instance maintains an internal array that often has greater capacity than the current length of the list.
- This extra capacity makes it easy to add a new element to the end of the list by using the next available cell of the array.
- If reserved capacity is exhausted, the class requests a new, larger array from the system, and copies all references from the smaller array into the beginning of the new array.
- A **dynamic array** effectively allows an array-based list to have unbounded capacity.

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

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

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

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

- 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 $O(n)$

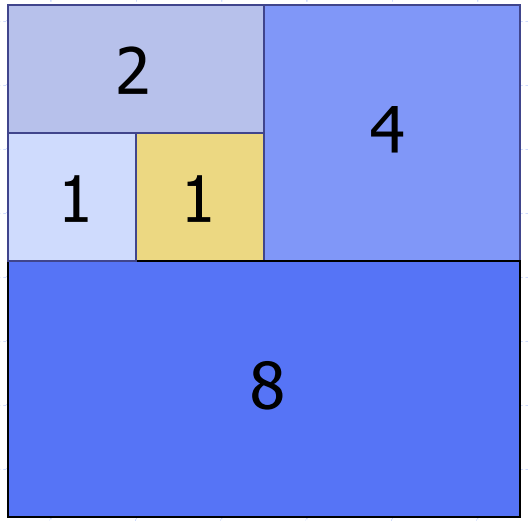
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- 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



Array Doubling

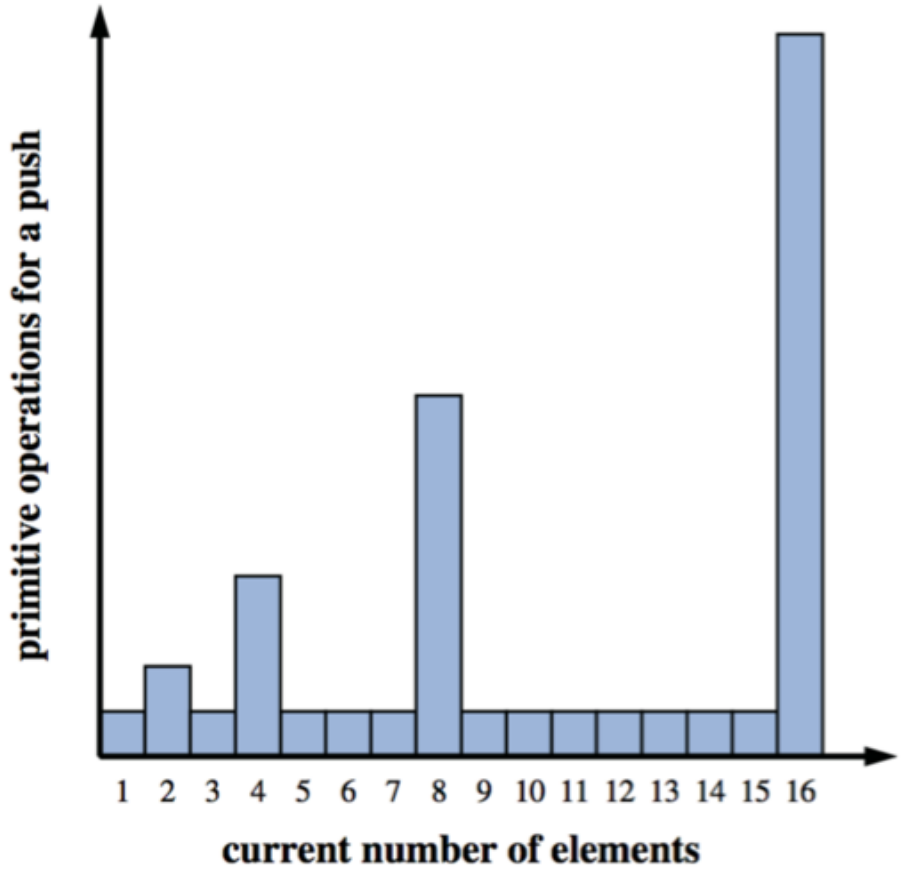


Figure 7.4: Running times of a series of push operations on a dynamic array.

Arithmetic Progression

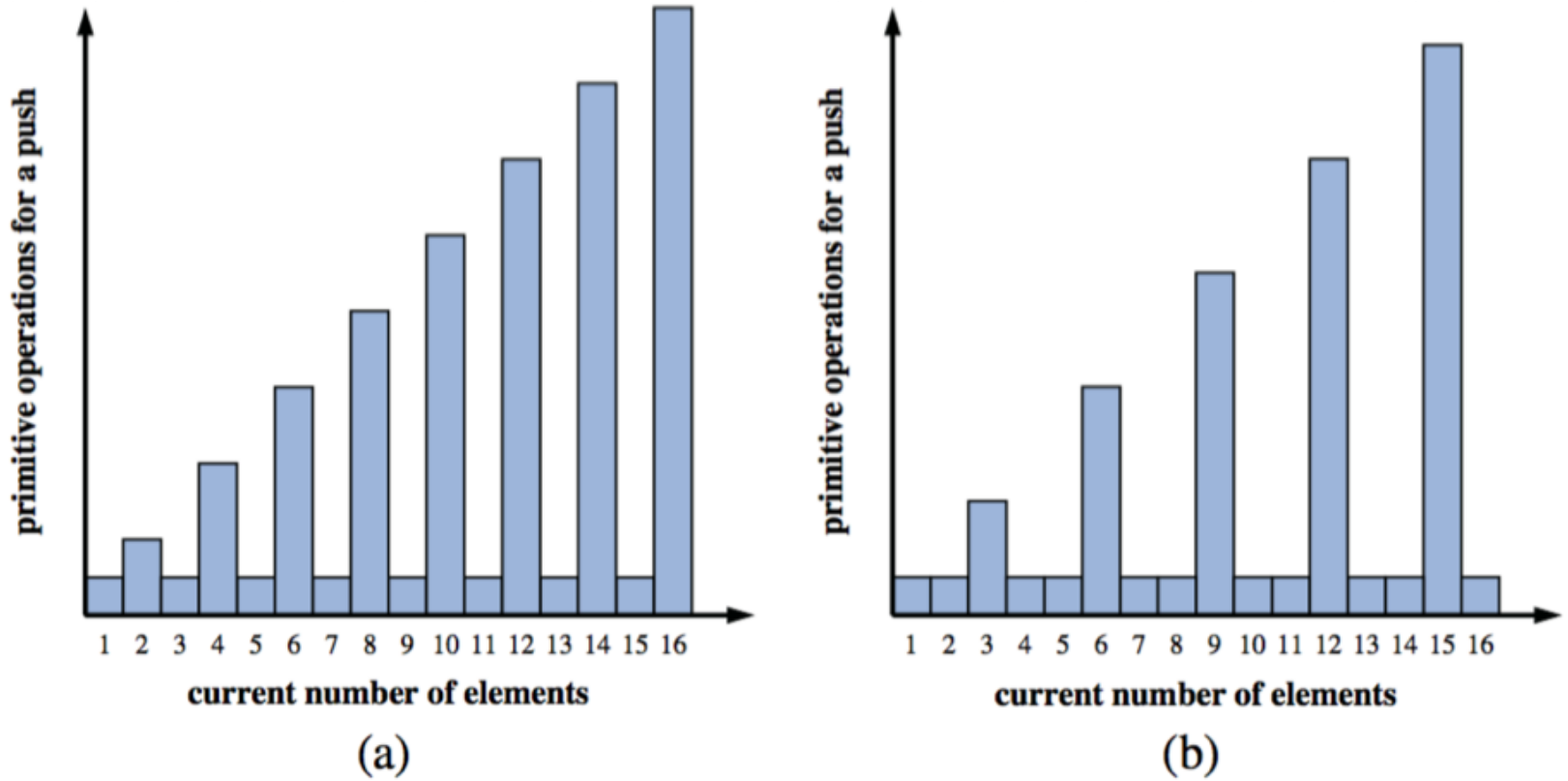


Figure 7.6: Running times of a series of push operations on a dynamic array using arithmetic progression of sizes. Part (a) assumes an increase of 2 in the size of the array, while part (b) assumes an increase of 3.

Deleting elements from Array List

- Array list uses $O(n)$ memory
- Space can get big with respect to n.
- We hope dynamic array guarantees $O(n)$ memory usage even after many elements are removed.
- Solution?

One Solution

- Shrink the array while maintaining the $O(1)$ amortized bound on individual operations.
- The array capacity is halved whenever the number of actual element falls below one-fourth of that capacity
  - when n decreases to n/4, shrink to half the size.
- In Exercise C-7.29 and C-7.31

Recaps

- Array List
- Dynamic Array

An example of dynamic array: constructing long strings in Java

- StringBuilder class represents a mutable string by storing characters in a dynamic array
- It guarantees that a series of append operations resulting in a string of length n execute in a combined time of $O(n)$

String VS StringBuilder (cont.)

```
1  /** Uses repeated concatenation to compose a String with n copies of character c. */
2  public static String repeat1(char c, int n) {
3    String answer = "";
4    for (int j=0; j < n; j++)
5      answer += c;
6    return answer;
7  }
8
9  /** Uses StringBuilder to compose a String with n copies of character c. */
10 public static String repeat2(char c, int n) {
11   StringBuilder sb = new StringBuilder();
12   for (int j=0; j < n; j++)
13     sb.append(c);
14   return sb.toString();
15 }
```

Code Fragment 4.2: Two algorithms for composing a string of repeated characters.

Running time of String and StringBuilder

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

Table 4.1: Results of timing experiment on the methods from Code Fragment 4.2.

Homework 3 question C-5.17

Write a short recursive Java method that takes a character string *s* and outputs its reverse. For example, the reverse of 'pots&pans' would be 'snap&stop'.

```
public String reverseString(String s){
        if (s.length() <= 1){
                return s;
        }
        return reverseString(s.substring(1)) + s.charAt(0);
}
```

Positional Lists

- A ***positional list*** is a general abstraction of a sequence of elements with the ability to identify the location of an element.

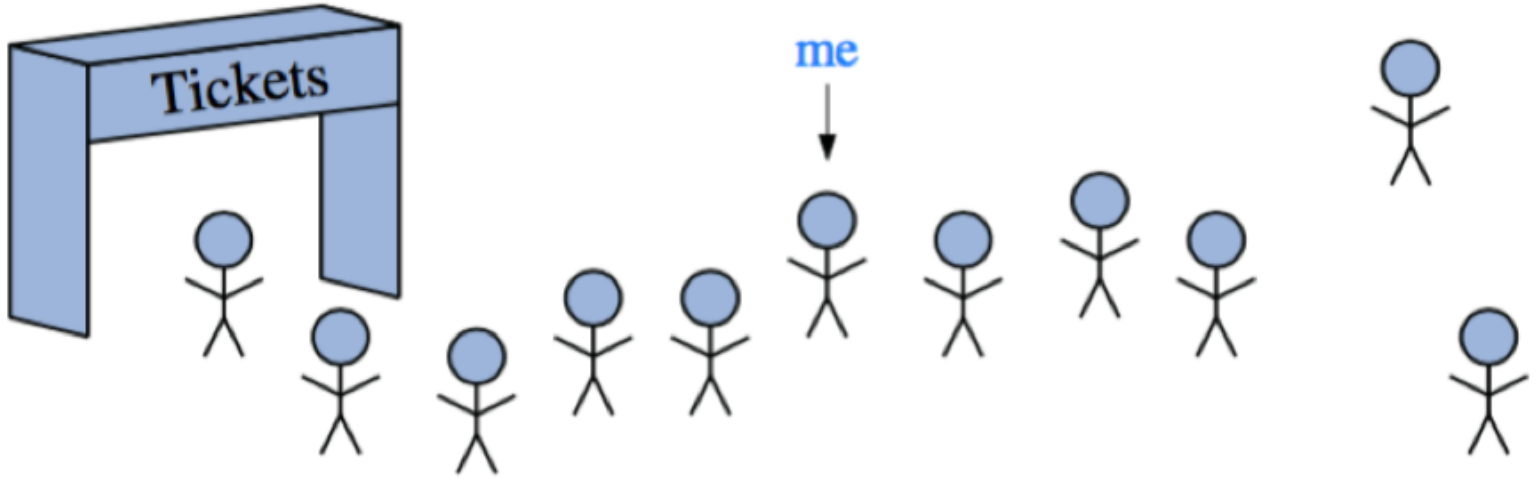


Figure 7.7: We wish to be able to identify the position of an element in a sequence without the use of an integer index. The label "me" represents some abstraction that identifies the position.

Positional Lists

- A ***positional list*** is a collection of positions, each of which stores an element.
- 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.

Traversal of a positional list

```
1  Position<String> cursor = guests.first( );
2  while (cursor != null) {
3    System.out.println(cursor.getElement( ));
4    cursor = guests.after(cursor);                // advance to the next position (if any)
5  }
```

Code Fragment 7.6: A traversal of a positional list.

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	
remove(last( ))	5	
set(p, 7)	8	
remove(q)	"error"	

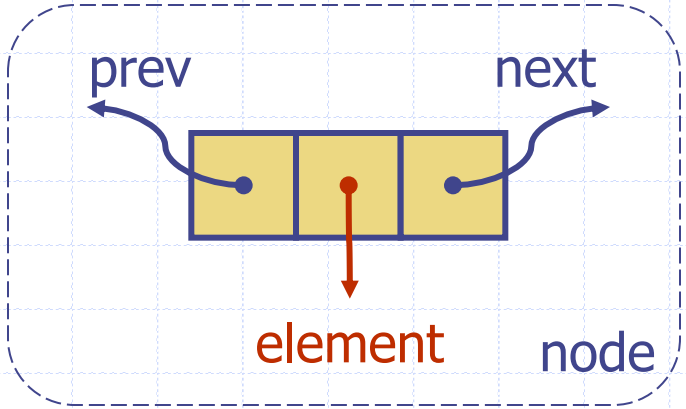
Example

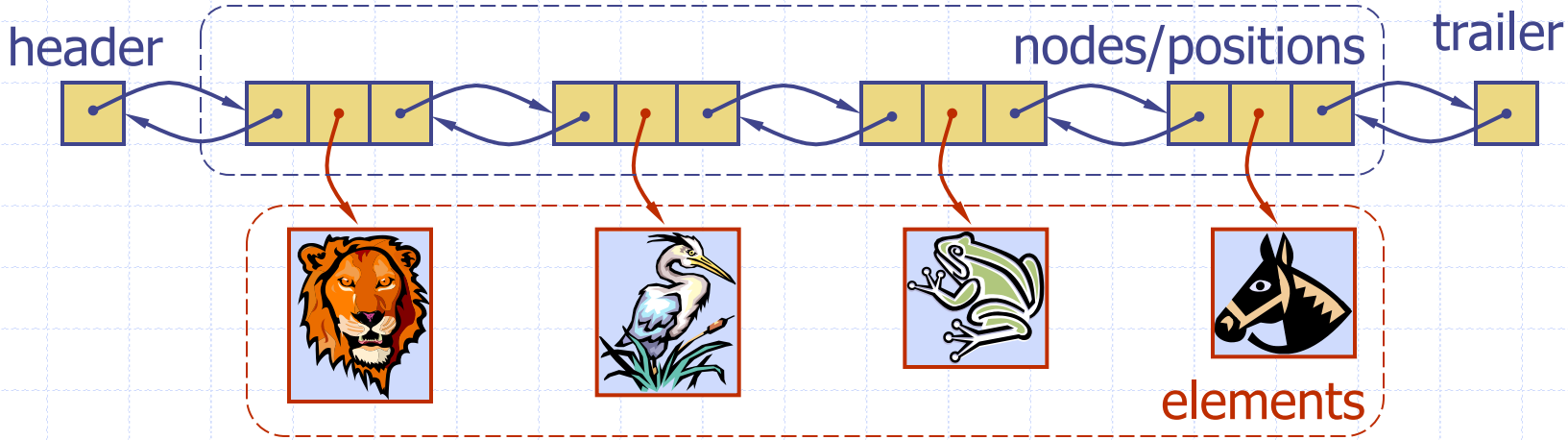
- A sequence of Positional List operations:

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

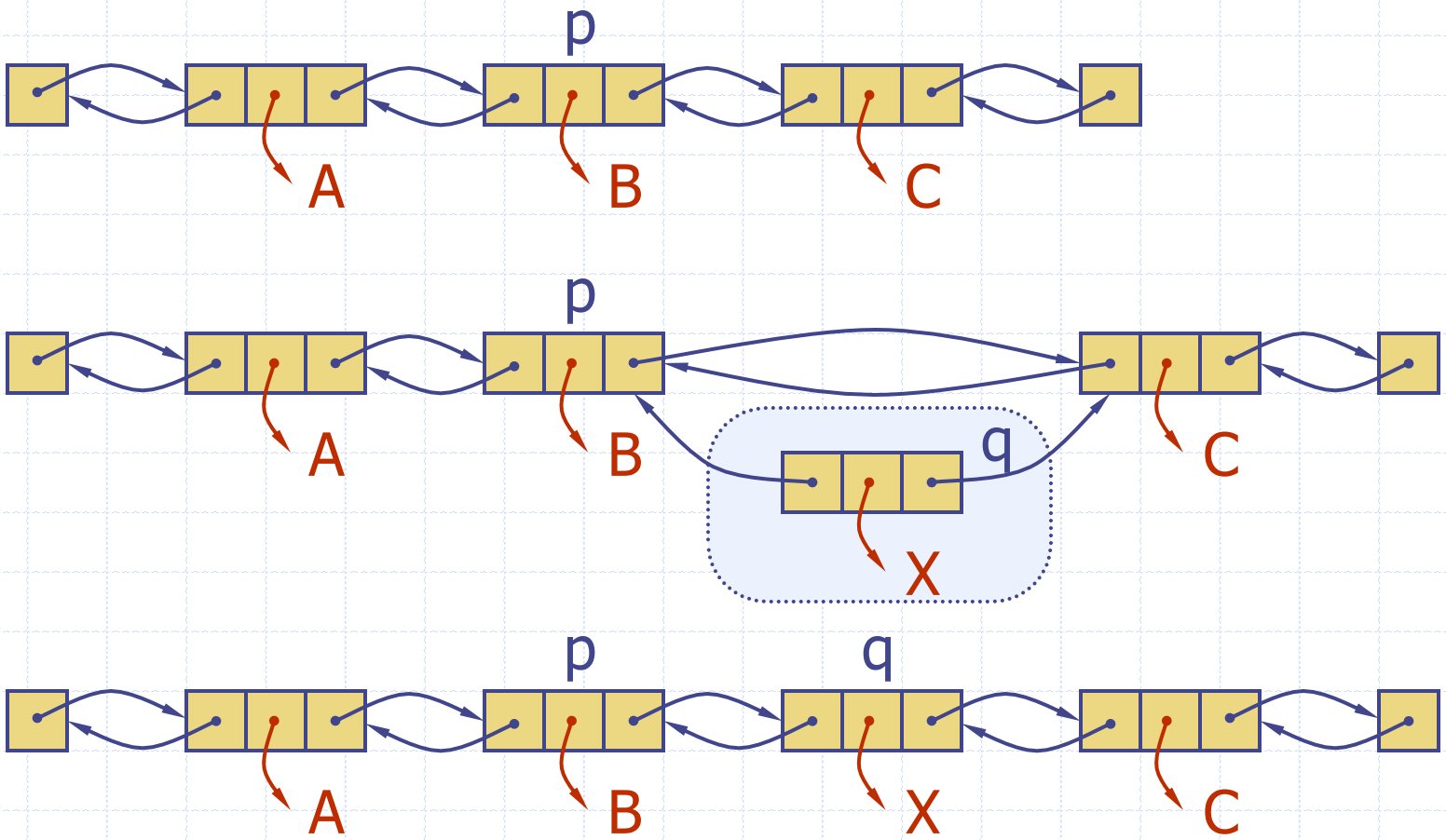
- The most natural way to implement a positional list is with a doubly-linked list.





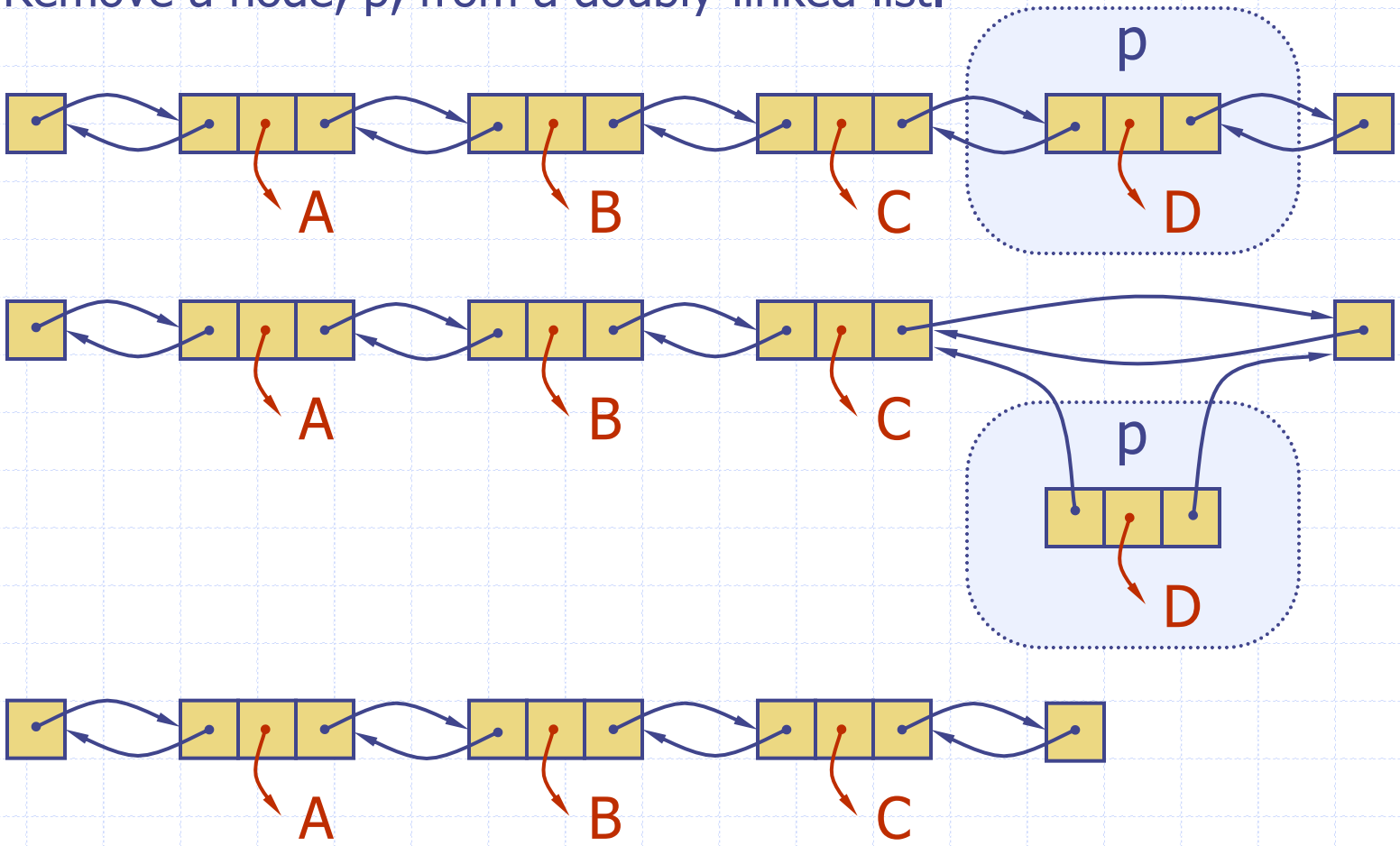
Insertion

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



Deletion

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



Performance of a Linked Positional List

Method	Running Time
size( )	$O(1)$
isEmpty( )	$O(1)$
first( ), last( )	$O(1)$
before(p), after(p)	$O(1)$
addFirst(e), addLast(e)	$O(1)$
addBefore(p, e), addAfter(p, e)	$O(1)$
set(p, e)	$O(1)$
remove(p)	$O(1)$

Table 7.2: Performance of a positional list with n elements realized by a doubly linked list. The space usage is $O(n)$.

Insertion Sort

6 5 3 1 8 7 2 4

Sorting a Positional List

- ***Insertion-sort*** algorithm
  - **Marker**: a variable representing the rightmost position of the currently sorted portion.
  - **Pivot**: the position just past the marker.

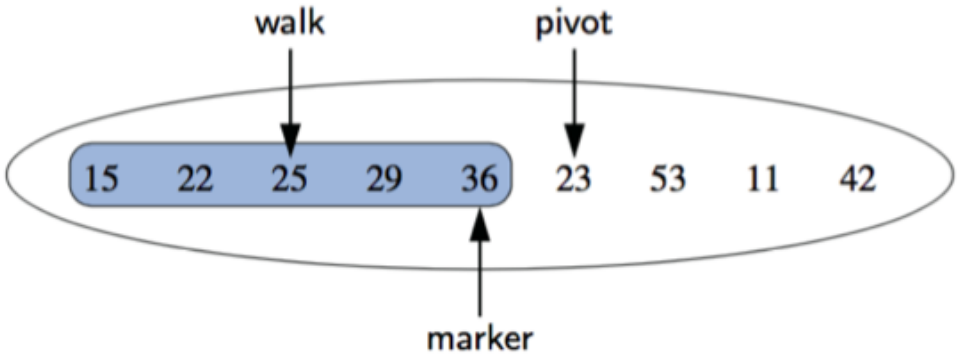
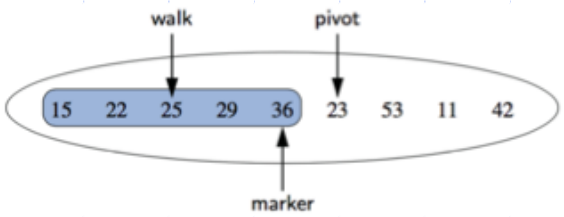


Figure 7.9: Overview of one step of our insertion-sort algorithm. The shaded elements, those up to and including marker, have already been sorted. In this step, the pivot's element should be relocated immediately before the walk position.

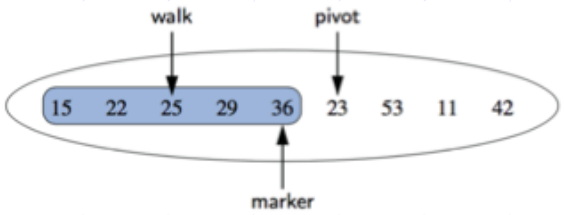
Implementation




```
1   /** Insertion-sort of a positional list of integers into nondecreasing order */
2   public static void insertionSort(PositionalList<Integer> list) {
3     Position<Integer> marker = list.first( );      // last position known to be sorted
4     while (marker != list.last( )) {
5       Position<Integer> pivot = list.after(marker);
6       int value = pivot.getElement( );            // number to be placed
7       if (value > marker.getElement( ))           // pivot is already sorted
8         marker = pivot;
9       else {                                      // must relocate pivot
10        Position<Integer> walk = marker;          // find leftmost item greater than value
11
12
13
14
15      }
16    }
17  }
```

Code Fragment 7.15: Java code for performing insertion-sort on a positional list.

Implementation




```
1  /** Insertion-sort of a positional list of integers into nondecreasing order */
2  public static void insertionSort(PositionalList<Integer> list) {
3    Position<Integer> marker = list.first( );      // last position known to be sorted
4    while (marker != list.last( )) {
5      Position<Integer> pivot = list.after(marker);
6      int value = pivot.getElement( );            // number to be placed
7      if (value > marker.getElement( ))           // pivot is already sorted
8        marker = pivot;
9      else {                                      // must relocate pivot
10       Position<Integer> walk = marker;          // find leftmost item greater than value
11       while (walk != list.first( ) && list.before(walk).getElement( ) > value)
12         walk = list.before(walk);
13       list.remove(pivot);                       // remove pivot entry and
14       list.addBefore(walk, value);              // reinsert value in front of walk
15     }
16   }
17 }
```

Code Fragment 7.15: Java code for performing insertion-sort on a positional 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.
- To unify the treatment and syntax for iterating objects in a way that is independent from a specific organization, Java defines the java.util.Iterator interface with the following two methods:

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.

Iterators (cont.)

- The combination of these two methods allows a general loop construct for processing elements of the iterator.
- For example, if we let variable, iter, denote an instance of the Iterator<String> type, then we can write the following:

```
while (iter.hasNext( )) {
  String value = iter.next( );
  System.out.println(value);
}
```

Iterators (cont.)

- The java.util.Iterator interface contains a third method, which is *optionally* supported by some iterators:

remove( ): Removes from the collection the element returned by the most recent call to next( ). Throws an IllegalStateException if next has not yet been called, or if remove was already called since the most recent call to next.

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"
}
```

Example

- We note that the iterator's remove method cannot be invoked when using the for-each loop syntax. Instead, we must explicitly use an iterator.
- As an example, the following loop can be used to remove all negative numbers from an ArrayList of floating-point values.

```
ArrayList<Double> data;   // populate with random numbers (not shown)
Iterator<Double> walk = data.iterator( );
while (walk.hasNext( ))
  if (walk.next( ) < 0.0)
    walk.remove( );
```

Implementing Iterators

- A **snapshot iterator** maintains its own private copy of the sequence of elements, which is constructed at the time the iterator object is created.
- A ***lazy iterator*** is one that does not make an upfront copy, instead performing a piecewise traversal of the primary structure only when the next() method is called to request another element.

Java collections framework

- The root interface in the Java collections framework is named **Collection**.
- The Collection interface includes many methods, including **iterator()**.
- It is a superinterface for other interfaces in the Java Collections Framework that can hold elements, including the java.util interfaces Deque, List, and Queue, and other subinterfaces including Set and Map

Java collections framework (cont.)

Class	Interfaces			Properties			Storage	
	Queue	Deque	List	Capacity Limit	Thread-Safe	Blocking	Array	Linked List
ArrayBlockingQueue	✓			✓	✓	✓	✓	
LinkedBlockingQueue	✓			✓	✓	✓		✓
ConcurrentLinkedQueue	✓				✓		✓	
ArrayDeque	✓	✓					✓	
LinkedBlockingDeque	✓	✓		✓	✓	✓		✓
ConcurrentLinkedDeque	✓	✓			✓			✓
ArrayList			✓				✓	
LinkedList	✓	✓	✓					✓

Table 7.3: Several classes in the Java Collections Framework.

List Iterators in Java

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

add(e):	Adds the element e at the current position of the iterator.
hasNext( ):	Returns true if there is an element after the current position of the iterator.
hasPrevious( ):	Returns true if there is an element before the current position of the iterator.
previous( ):	Returns the element e before the current position and sets the current position to be before e.
next( ):	Returns the element e after the current position and sets the current position to be after e.
nextIndex( ):	Returns the index of the next element.
previousIndex( ):	Returns the index of the previous element.
remove( ):	Removes the element returned by the most recent next or previous operation.
set(e):	Replaces the element returned by the most recent call to the next or previous operation with e.

Next

First midterm exam !