INFO1105/1905/9105
Data Structures
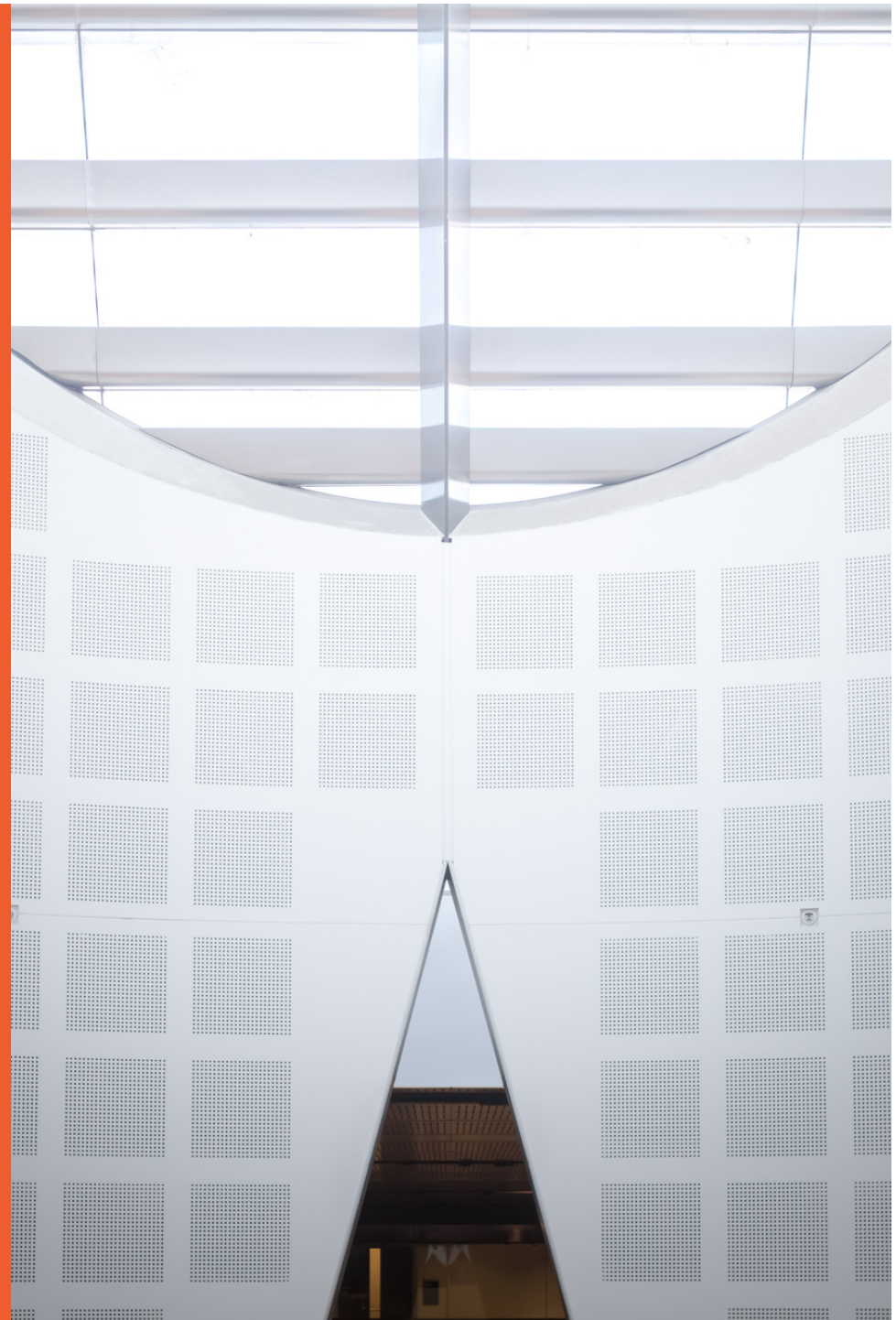## Week 2: Lists

Professor Alan Fekete
Professor Seokhee Hong
School of Information Technologies

Some content is taken from material provided by the textbook publisher J. Wiley

THE UNIVERSITY OF
SYDNEY

Copyright warning

Reflection!

What did you achieve from the lab of Week 1 (including what you did up to the submission deadline today)?

- completed it all?
- attempted it all and completed the assessed task?
- completed the assessed task but there was something you didn't attempt?
- attempted the assessed task but didn't complete it?
- did not attempt the assessed task?

Change your approach immediately,
or drop the unit until you are willing to do the work!

Get help from your tutor, or make appointment with coordinator

Java Generics

Coding technique that allows collections to have a type parameter, indicating what type the entries have (sec 2.5.2)

Generics

Java includes support for writing generic classes and methods that can operate on a variety of data types while often **avoiding the need for explicit casts**.

The generics framework allows us to define a class in terms of a set of formal type parameters, which can then be used as the declared type for variables, parameters, and return values within the class definition.

Those formal type parameters are later specified when using the generic class as a type elsewhere in a program.

© 2014 Goodrich, Tamassia, Goldwasser

Syntax for Generics

Types can be declared using generic names:

```
1   public class Pair<A,B> {
2      A first;
3      B second;
4      public Pair(A a, B b) {                    // constructor
5         first = a;
6         second = b;
7      }
8      public A getFirst() { return first; }
9      public B getSecond() { return second;}
10  }
```

They are then instantiated using actual types:

```
Pair<String,Double> bid;
```

© 2014 Goodrich, Tamassia, Goldwasser

Typed versus generic

```
public class DataNode {
    private Integer data;
    public DataNode( Integer newValue) {
        data = newValue;
    }

    public Integer getData() {
        return data;
    }
    public void setData( Integer newValue) {
        data = newValue;
    }
}
```

```
public class DataNode<E> {
    private E data;
    public DataNode( E newValue) {
        data = newValue;
    }

    public E getData() {
        return data;
    }
    public void setData( E newValue) {
        data = newValue;
    }
}
```

© 2014 Goodrich, Tamassia, Goldwasser

Using the generic data node

➤ An integer data node

```
Integer d = new Integer(7);
DataNode<Integer> myNode =
 new DataNode<Integer>( d ) ;
```

➤ A string data node

```
DataNode<String> myNode =
 new DataNode<String>("John") ;
```

```java
public class DataNode<E> {
    private E data;
    public DataNode( E newValue) {
        data = newValue;
    }

    public E getData() {
        return data;
    }
    public void setData( E newValue) {
        data = newValue;
    }
}
```

© 2014 Goodrich, Tamassia, Goldwasser

ADT programming: an example

Linear collection types

Many data collections have an ordering which means that they can be arranged in a row or sequence

attempts to a tutorial question

students on the waitlist for a class

unread email

log of events

Often, a collection contains a sequence of objects of the same type (e.g. sequence of Student objects for a waitlist)

Traditional programming

The enrollment system might have a field (instance variable) done with an array

```
Student[] waitlist;
int numwaiting;
```

And code that manipulates these

```
if (numwaiting>0) {
    Student firstwaiting = waitlist[0];
}
```

ADT programming

*Define an interface List
*Provide an implementation of this interface by a library class (or use an existing library)
The interface has well-chosen methods that manipulate the collection (get particular entries, change the order, insert new entry, etc.)

In the enrollment system, declare a field (instance variable) `List<Student> waitlist;`
and write code that manipulates this through its operations

```
Student firstwaiting = waitlist.get(0);
```

Benefits of ADT approach

*Code is easier to understand if different issues are separated into different places

   Client can be considered at a higher, more abstract, level

*Many different systems can use the same library

   only code (and test!) tricky manipulations once, rather than in every client system

*There can be choices of implementations with different performance tradeoffs, and the client doesn't need to be rewritten extensively to change which implementation it uses

List ADT

An index-accessed List ADT (sec 7.1)

Variations on the ADT for ordered sequences

There are several different ways that are commonly used for defining an ADT for a collection that has an order

One central issue: how to refer to locations or entries within the collection

`java.util.List` has a mixture of several ways!
We begin by considering a simple ADT we will call List, that is accessed very much like an array: by referring to entries based on their index

the first element is at index 0
the nth element is at index $n-1$

List ADT (with Index access)

size()   (int) number of elements in the store
isEmpty()   (boolean) whether or not the store is empty

get(i)   return element at index i

set(i, e)   replace element at index i with element e,
              and return element that was replaced

add(i, e)   insert element e at index i
              existing elements with index >= i are shifted up

remove(i)   remove and return the element at index i
              existing elements with index > i are shifted down

List ADT: Self-check

Given

mylist = (3, 4, 8, 1)

What is the result of doing the following sequence of operations, one after the other?:

mylist.size()
mylist.isEmpty()
mylist.get(2)
mylist.set(1, 5)
mylist.add(1, 7)
mylist.size()
mylist.remove(3)

List ADT: Self-check answers

Given

mylist = (3, 4, 8, 1)

What is the result of doing the following sequence of operations, one after the other?:

mylist.size()	4
mylist.isEmpty()	false
mylist.get(2)	8
mylist.set(1, 5)	4, mylist = (3, 5, 8, 1)
mylist.add(1, 7)	void, mylist = (3, 7, 5, 8, 1)
mylist.size()	5
mylist.remove(3)	8, mylist = (3, 7, 5, 1)

Implementing List with array

An array-based implementation
of List ADT (sec 7.2)

Recall: Java arrays provide a linear structure (sec 1.3)

Creating an array allocates memory to store a collection of elements

```
int[] someIntegers = new int[10];
```

The **type** of all elements is declared and fixed
Java generics framework can handle unspecified type

The amount of **memory** allocated is declared and fixed
an instance variable may indicate how many of the allocated slots are currently in use with meaningful contents

Retrieve an element by index in constant time (we say "O(1)")

```
int eighth = someIntegers[7];
```

Internals of the Array

Cells in an array are associated with particular pieces of allocated memory

allocated memory
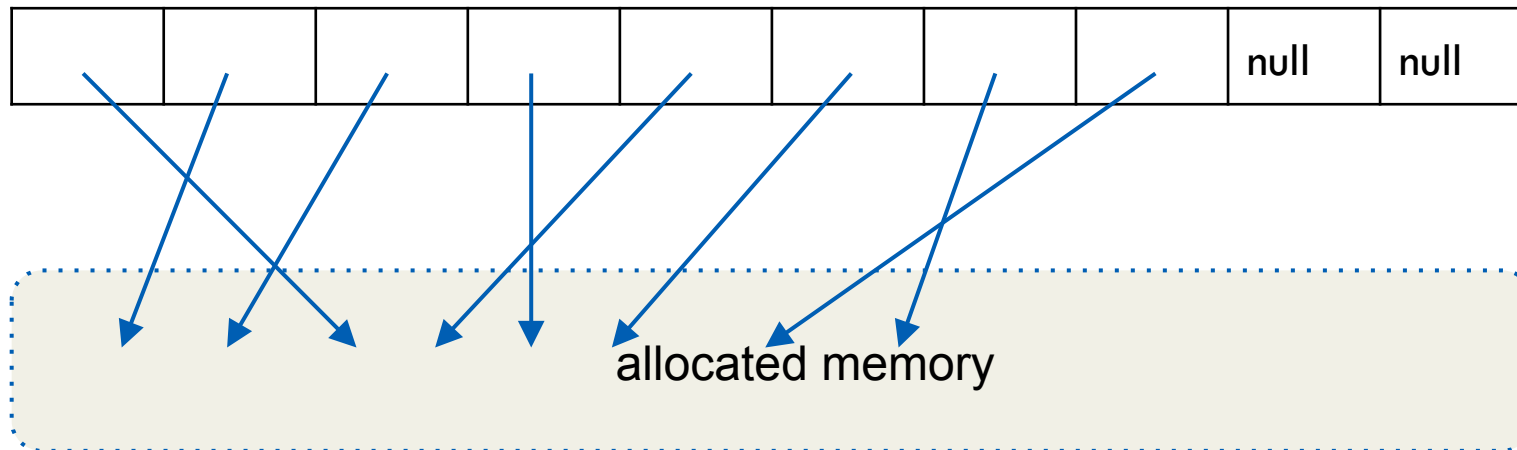
Retrieve an element by its index *in constant time*

```
int eighth = someIntegers[7];
```

ArrayList

The ArrayList class implements the List interface

An instance variable of ArrayList is an array
- another instance variable *size* says how many entries are currently meaningful
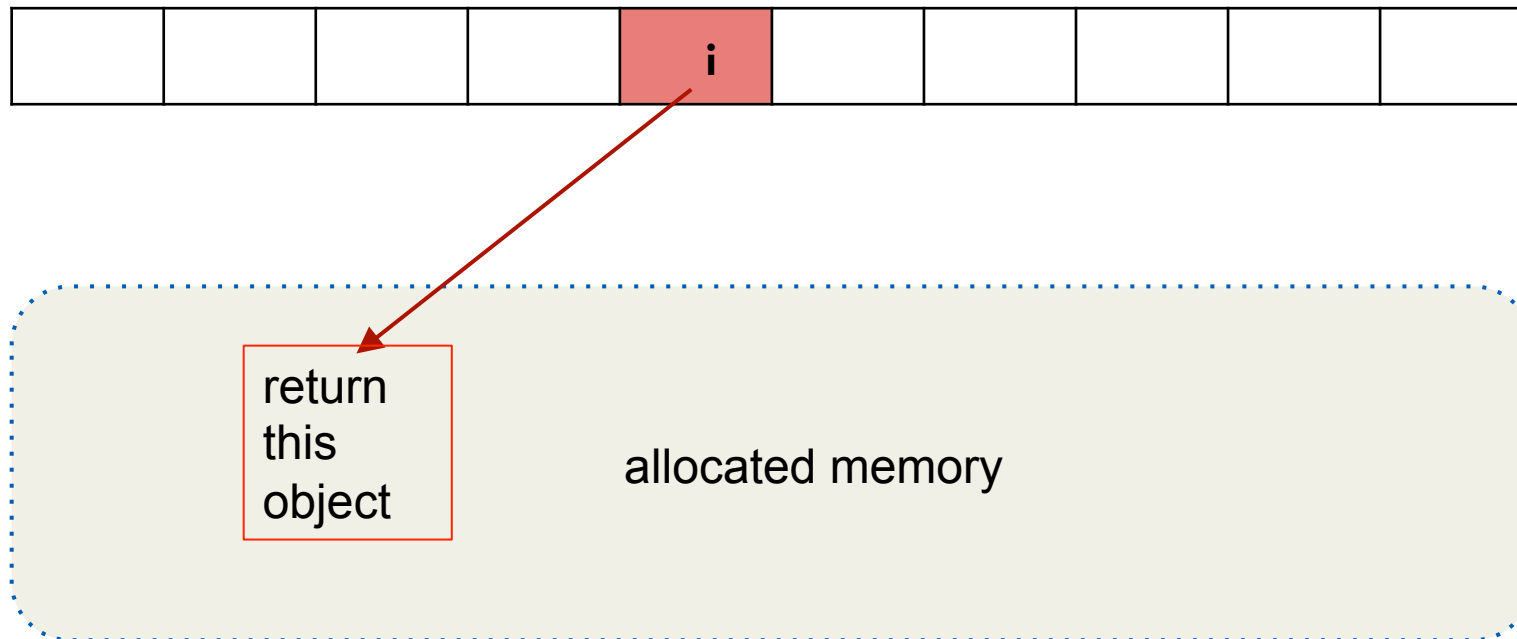
Methods of ArrayList manipulate the array
- See Code fragments 7.2, 7.3

Code excerpts (simplified from Code 7.2)

```
public class ArrayList<E> implements List<E> {
    public static final int CAPACITY = 16;
    private E[] data;
    private int size;

    public ArrayList() {
        data = new E[CAPACITY];
        size = 0;
    }

    // methods required in the interface
}
```

ArrayList.get(i)

Retrieve value stored in memory with index i

code fragment (over-simplified from Code 7.3)

Essence is:
```
public E get(int i){
    return data[i];
}
```

Actual code needs to deal with exceptional cases
```
public E get(int i) throws IndexOutOfBoundsException {
  if (i < 0 || i >= size)
    throw
 new IndexOutOfBoundsException("Illegal index: " + i);
  return data[i];
}
```

ArrayList.set(i, e)

Update the value in memory with index i so it now points to e, return previous value at this index

previous reference in index i
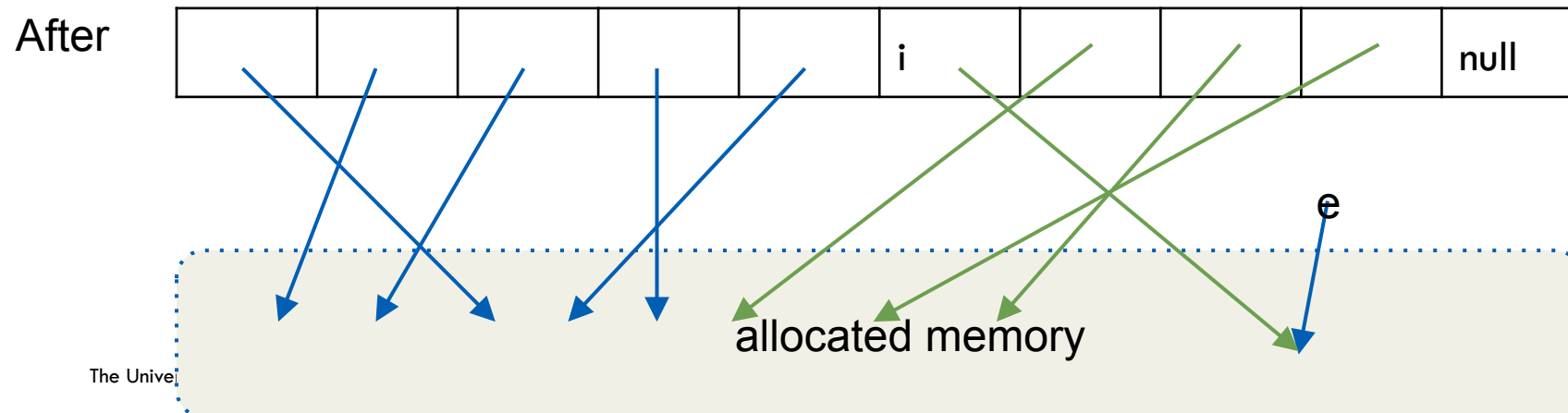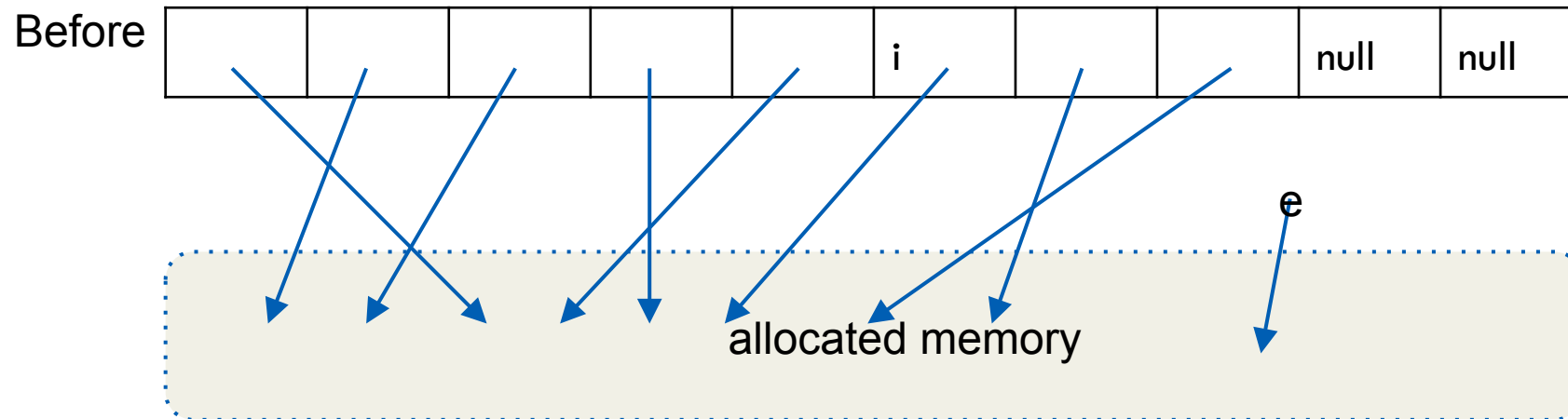
e

return this object

allocated memory
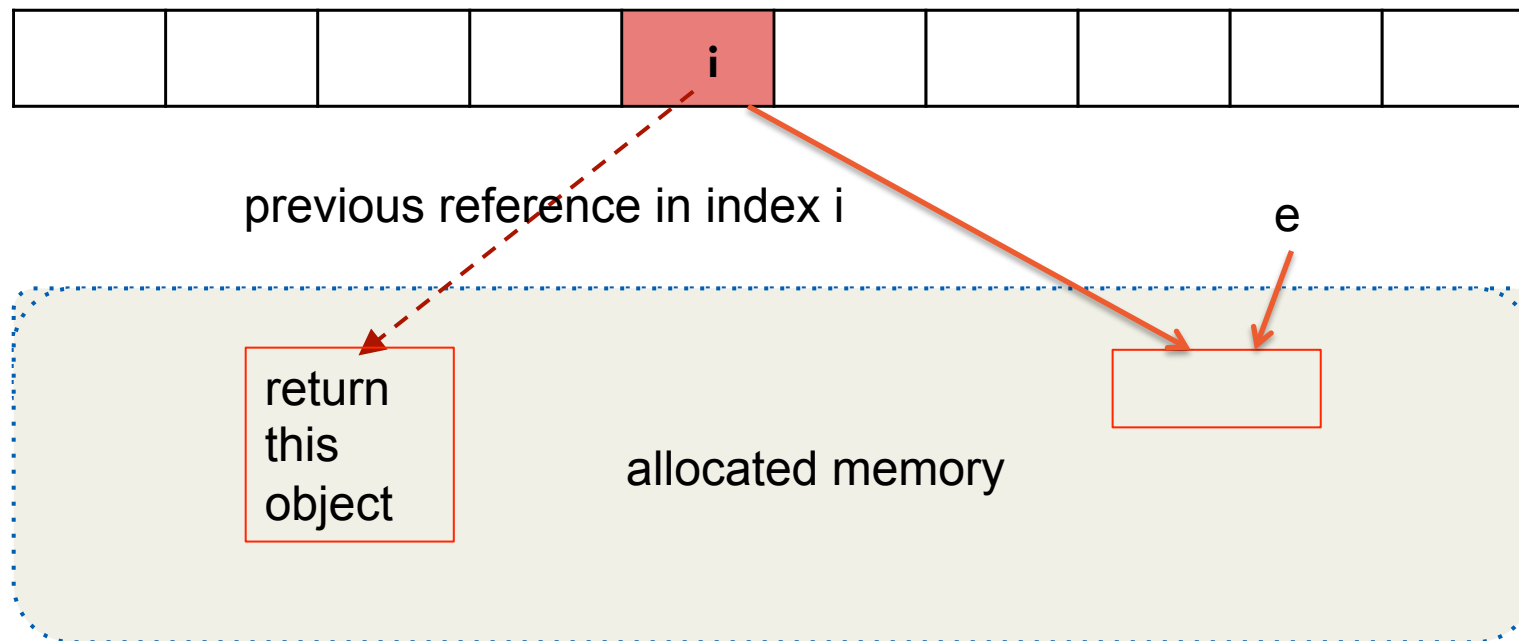
code fragment (over-simplified from Code 7.3)

Essence is:

```
public E set(int i, E e){
    E temp = data[i];
    data[i] = e;
    return temp;
}
```

ArrayList.add(i, e)

insert element e at index I, existing elements with index >= i are shifted up

Before

					i			null	null

allocated memory

e

After

					i				null

allocated memory

e

WRONG! WRONG! WRONG!

```
public void add(int i, E e) {
    data[i] = e;
    for (int j = i, j < size, j++) {
        data[j+1] = data[j];
    }
    size++;
}
```

This does ensure that e is the value at index i, but it doesn't shift the other elements correctly!

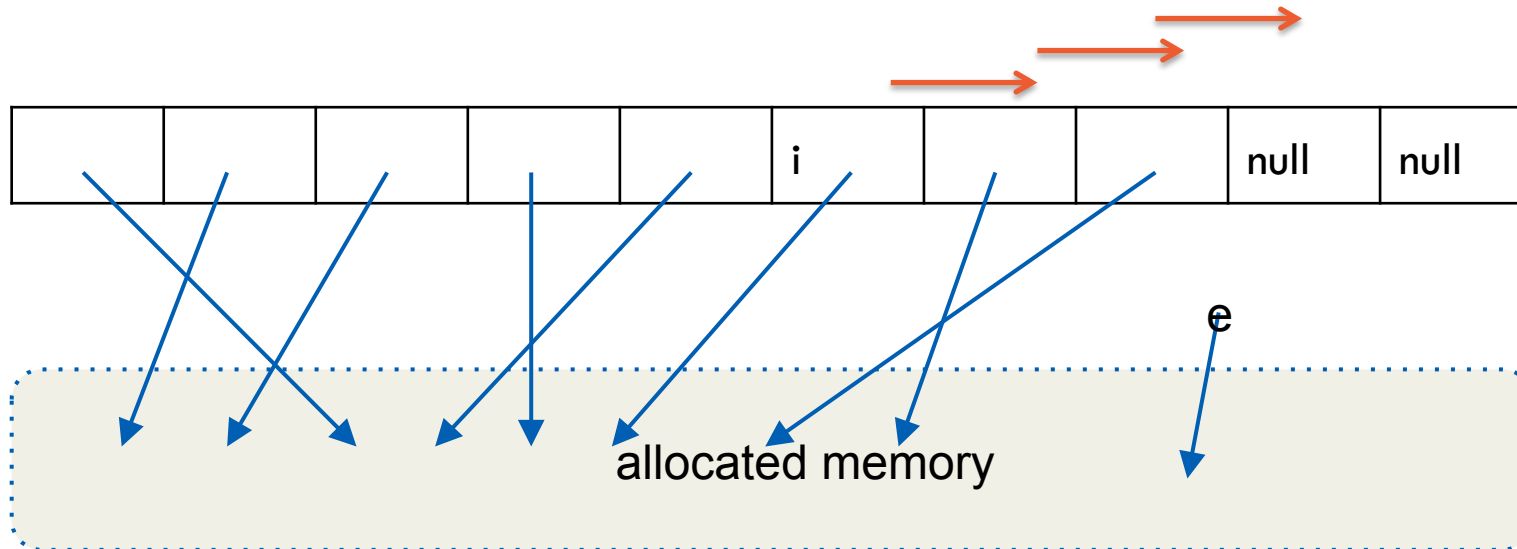# ArrayList.add(i, e)

Iterate over indices >= i, updating values
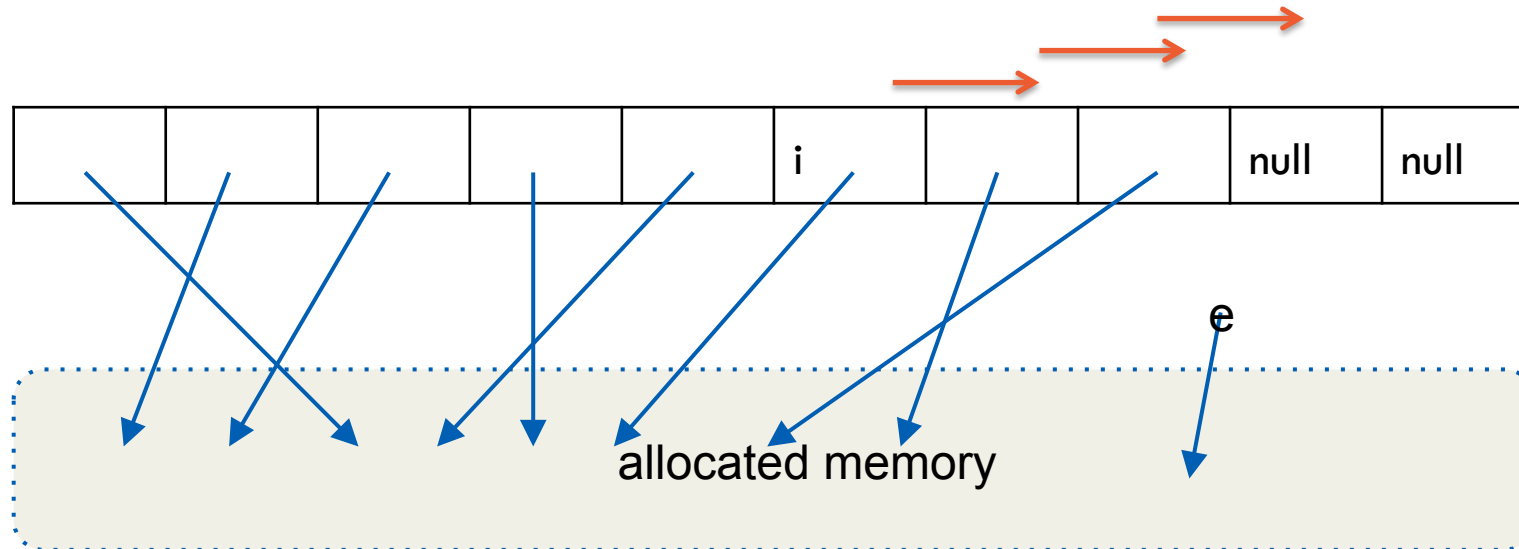
What is the most time consuming scenario?

What is the time complexity of ArrayList.add(i, e)?

ArrayList.add(i, e)

Iterate over indices >= i, updating values

What is the most time consuming scenario?
add at the start of the array (that is, i=0)
What is the time complexity of ArrayList.add(i, e)?
O(n)

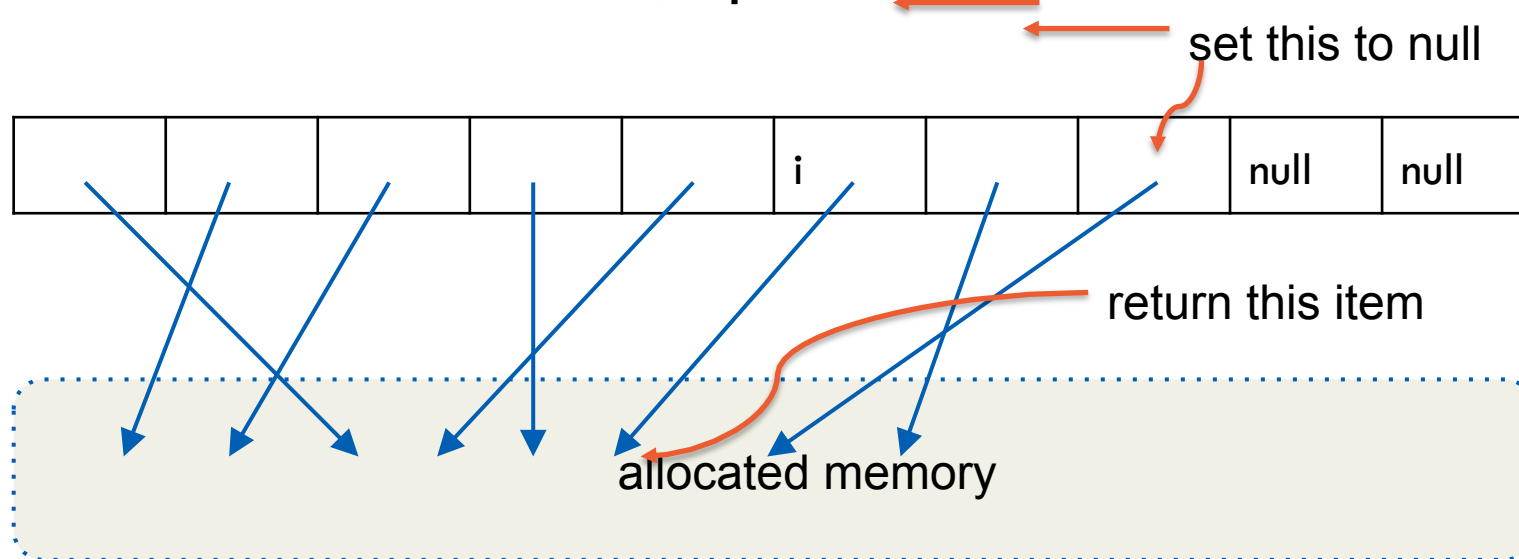# Essence of the code fragment (from Code 7.3)

```
public void add(int i, E e) {
   for (int k = size-1, k >= i, k--) {
        data[k+1] = data[k];
   }
   data[i] = e;
   size++;
}
```

Also, need to deal with exceptional situations, such as inappropriate index or array already full.

ArrayList.remove(i)

Iterate over indices >= i, updating values

set this to null

					i			null	null

return this item

allocated memory

What is the most time consuming scenario?
remove at the start of the array (that is, i=0)
What is the time complexity of ArrayList.remove(i)?
O(n)

Code fragment (simplified from code 7.3)

```
public E remove(int i) {
    E temp = data[i];
    for (int k = i, k<size-1;k++) {
        data[k+1] = data[k];
    }
    data[size-1] = null;
    size--;
    return temp;
}
```

Dynamic array

What if we want to insert an entry, and the internal array is full?

- simple approach: throw an exception (see code 7.3)

Instead, there is a more useful and general approach: get a larger array for the internal instance variable, and relocate all the existing entries to there

- Details in sec 7.2.1, 7.2.2

This makes the add operation sometimes much slower, but worst-case time is still O(n)

ArrayList implements List

Space complexity is O(n), but we often allocate more space than needed

    memory allocated via Array construction

Time complexity of simple List ADT operations is O(1)

    size()   (int) number of elements in the store
    isEmpty()   (boolean) whether or not the store is empty
    get(i)   return element at index i
    set(i, e)   replace element at index i with element e,
                and return element that was replaced

Time complexity of state altering ops: worst case O(n)

    add(i, e)   insert element e at index i
    remove(i) remove and return the element at index i

Position-based Lists

A variant ADT with access by Position (sec 7.3), and implementations with singly-linked or doubly-linked lists (sec 3.2, 3.4)

Position ADT

*Position models the abstract notion of place where a single object is stored within a container data structure
*Unlike index, this keeps referring to the same entry even after insertion/deletion happens elsewhere in the collection

*Position gives a unified view of diverse ways of storing data

a **cell** of an array implementation of a List

a **node** in a linked implementation of a List

*Position offers just one method:

element() return the element stored at the position instance

PositionalList ADT
(some of the ops, see 7.3.2 for more)

size()   (int) number of elements in the store

isEmpty()   (boolean) whether or not the store is empty

first()   return position of first element

last()   return position of last element

before(p)   return position immediately before p

after(p)   return position immediately after p

set(p, e)   replace element at position p with element e

remove(p)   remove and return the element at position

addBefore(p,e)   insert e in front of the element at position p

addAfter(p,e)   insert e following the element at position p

addFirst(e)   insert element e at front

addLast(e)   insert element e at end

Singly Linked List

Concrete data structure
A sequence of Nodes each with reference to the
next node
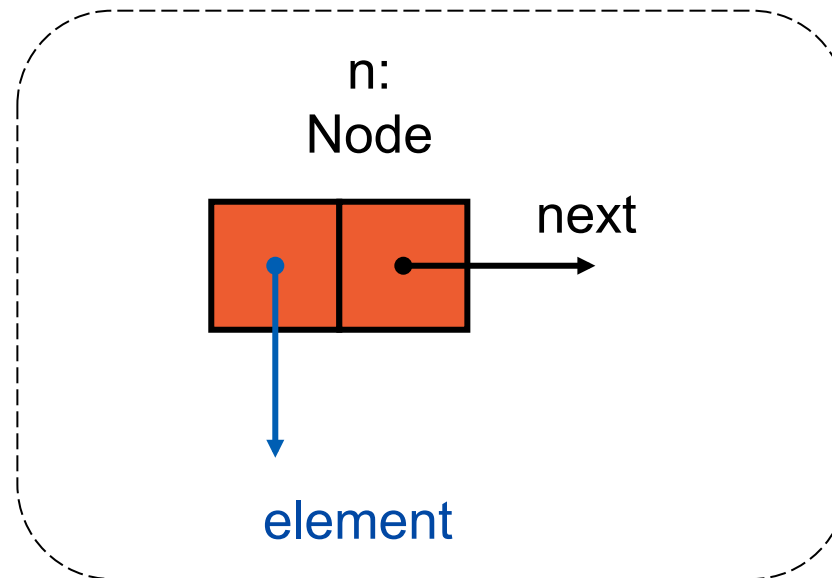List captured by reference to the first Node
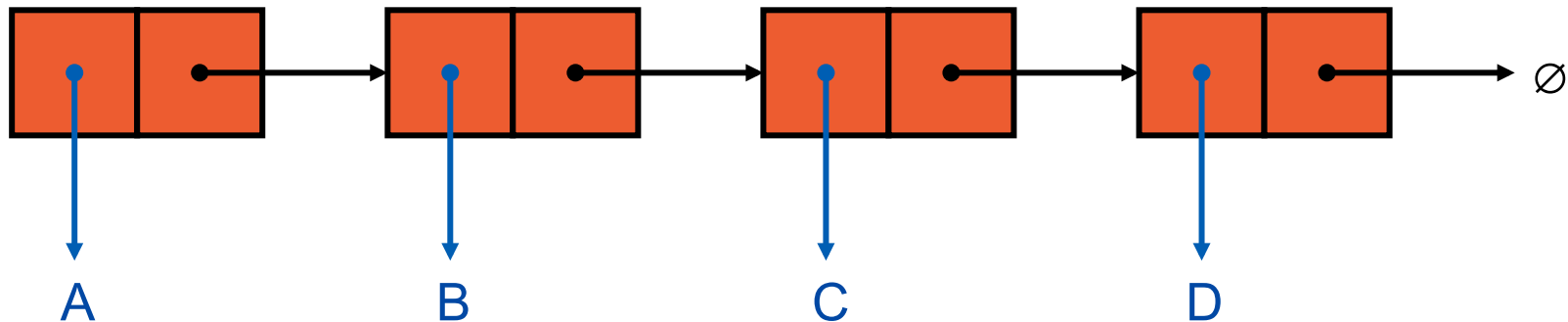
Node implements Position

Each Node in a Singly Linked List stores

- its element
- a link to the next node

Advice on working with linked structures

Draw the diagram showing the state

Show a location where you place carefully each of the instance variables (including references to nodes)
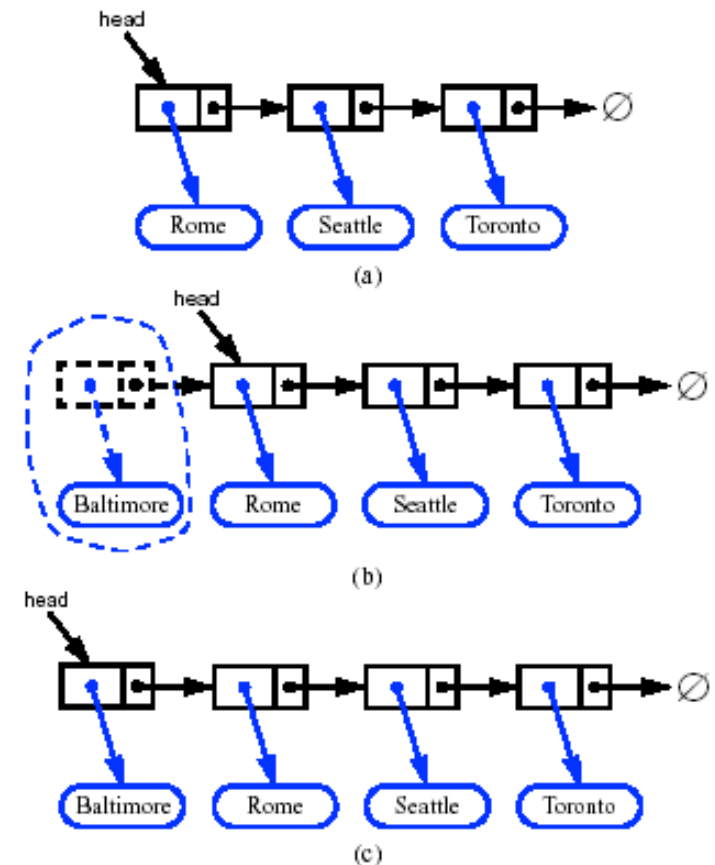
Be careful to step through dotted accesses
    e.g. `p.next.next`

Be careful about assignments to fields
    e.g. `p.next = q;    p.next.next = r;`

LinkedList.addFirst(e)

1. Instantiate a new node, n
2. Set e as element of n
3. Set n.next to point to old head
4. Update list's head to point to n
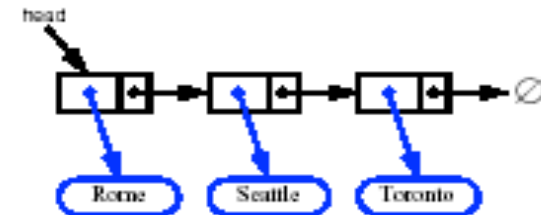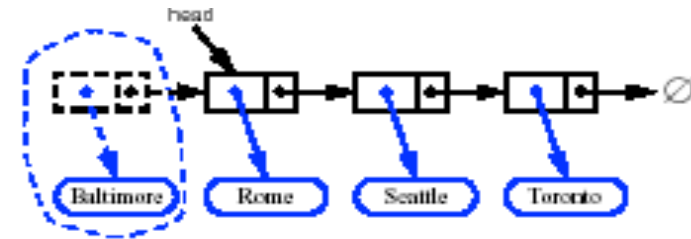
What is the time complexity?

LinkedList.addFirst(e)

1. Instantiate a new node, n
2. Set e as element of n
3. Set n.next to point to old head
4. Update list's head to point to n

What is the time complexity?
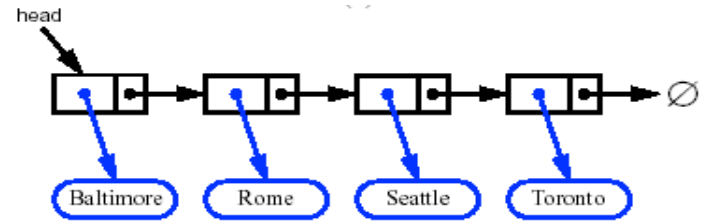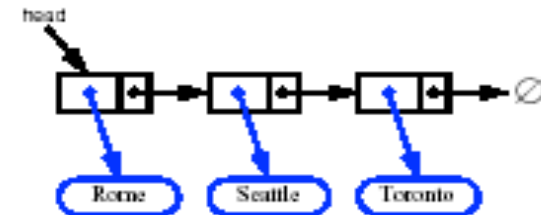
O(1)
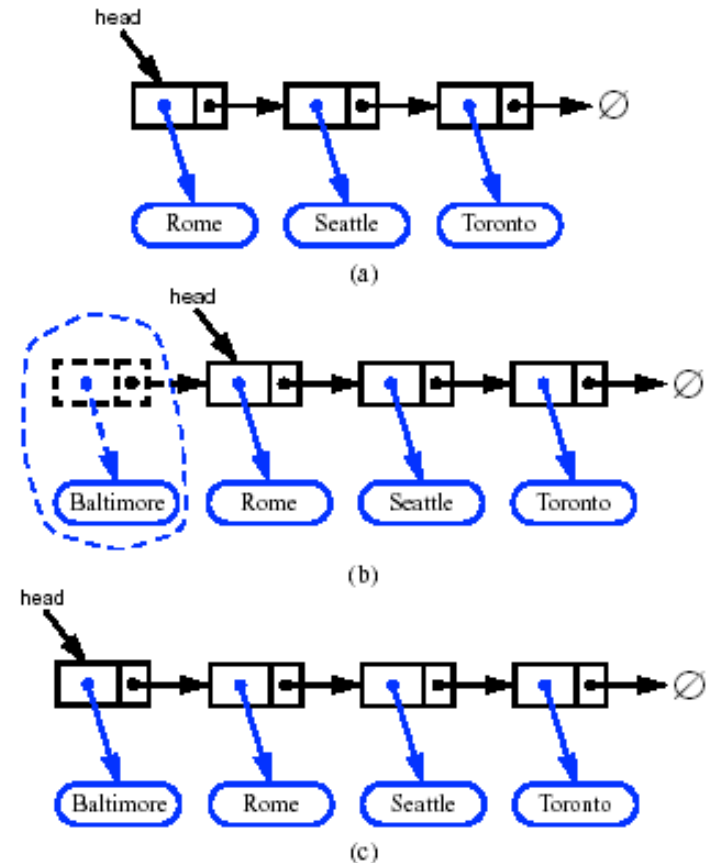
LinkedList.removeFirst()

1. Update list's head to point to next node

2. Allow garbage collection to reclaim the former first node

What is the time complexity?
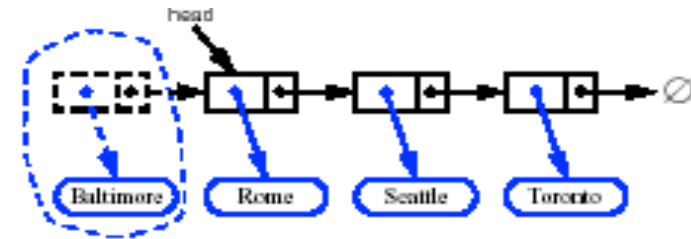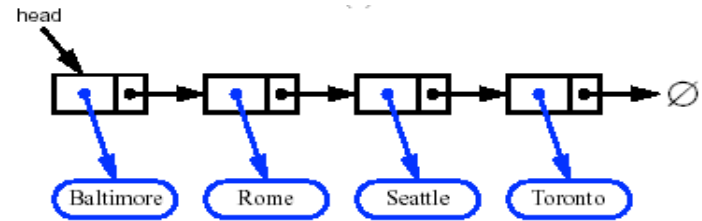
LinkedList.removeFirst()

1. Update list's head to point to next node

2. Allow garbage collection to reclaim the former first node
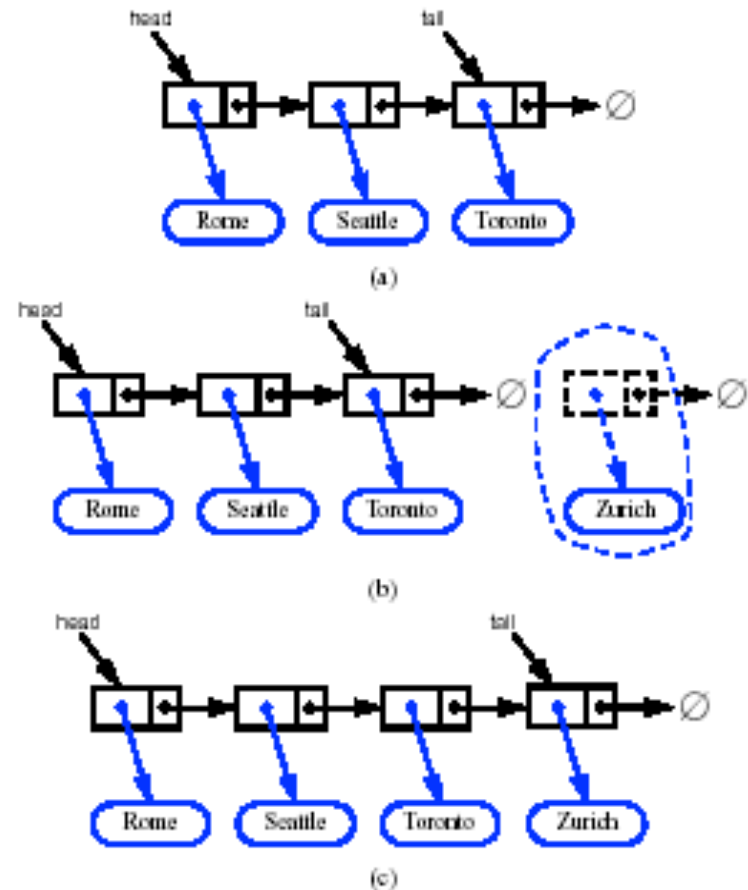
What is the time complexity?

O(1)

LinkedList.addLast(e)

1. Instantiate a new node, n
2. Set e as element of n
3. Set n.next to point to null
4. Set old tail to point to n

How do we find the tail?

LinkedList.last()

to find the tail in a singly-linked list: Follow links
from the list's head

What is the time complexity of this?

LinkedList.last()

to find the tail in a singly-linked list: Follow links from the list's head

What is the time complexity of this?
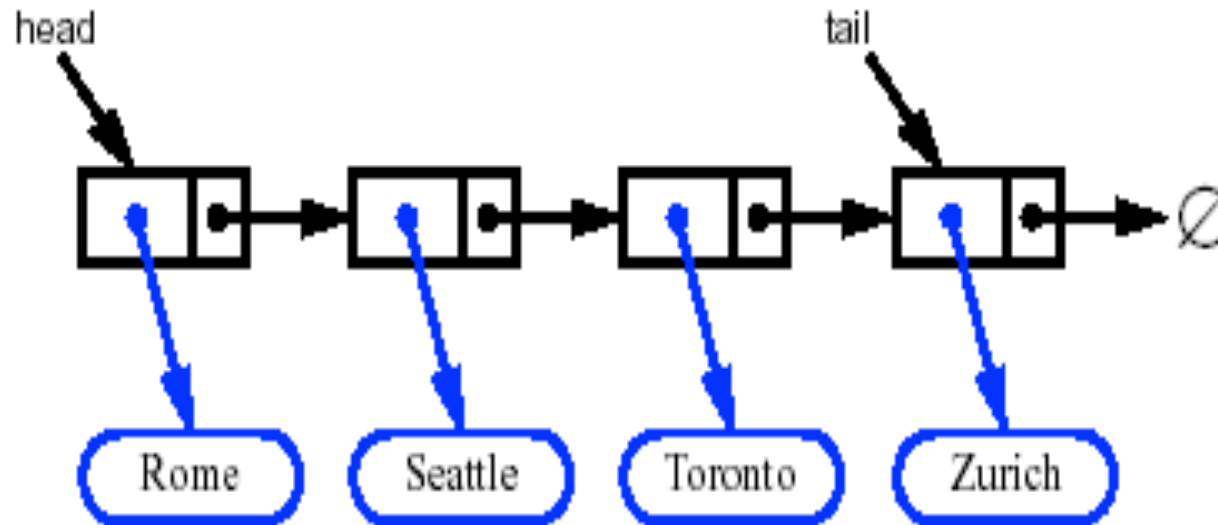
answer: O(n)

Faster: include a tail pointer in your implementation

Can anyone see the problem with this?

LinkedList.removeLast()

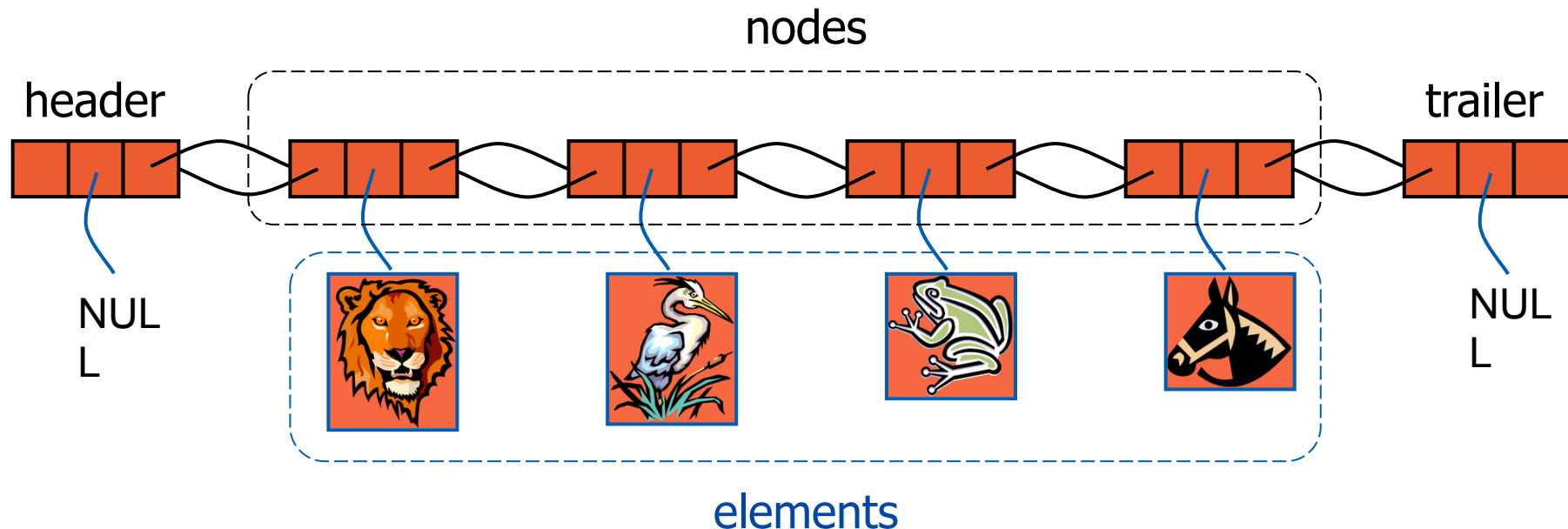There is no constant-time way to update the tail of a Singly Linked List to point to the new node

Doubly Linked List

Concrete data structure
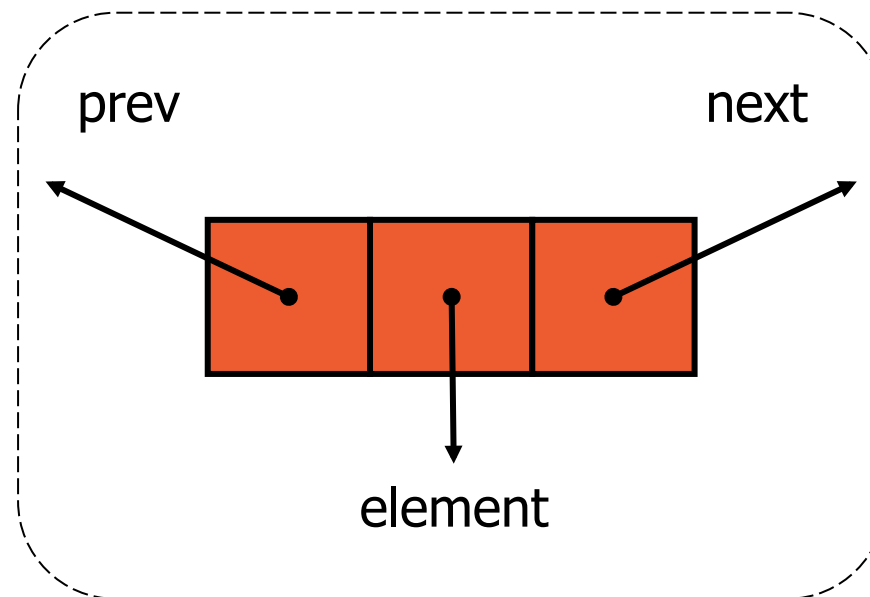A sequence of Nodes, each with reference to prev
and to next
List captured by reference to its Sentinel Nodes

Node implements Position

Each Node in a Doubly Linked List stores
- its element
- a link to the previous and next nodes

Putting it together: Insertion

Algorithm addAfter(p,e):
    create a new node n
    n.setElement(e)
    n.setPrev(p)   // link n to its predecessor
    n.setNext(p.getNext())   // link n to its successor
    (p.getNext()).setPrev(n) // link p's old successor to n
    p.setNext(n)   // link p to its new successor, n
    return n   // the Position for the element e

What is the time complexity of Insertion?

LinkedList.addAfter(p, e) [first steps]

p

A   B   C

Instantiate new Node n with element set to e
Update n.previous to point to $Node_A$ at p and n.next to point to $Node_C$

A   B   C

e
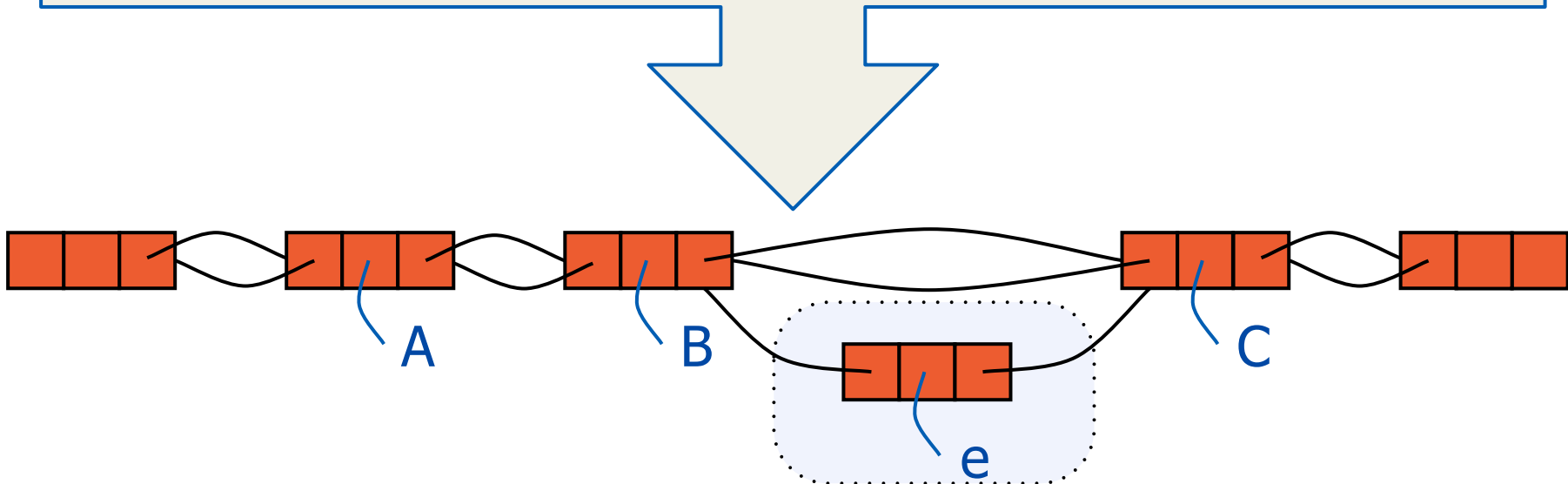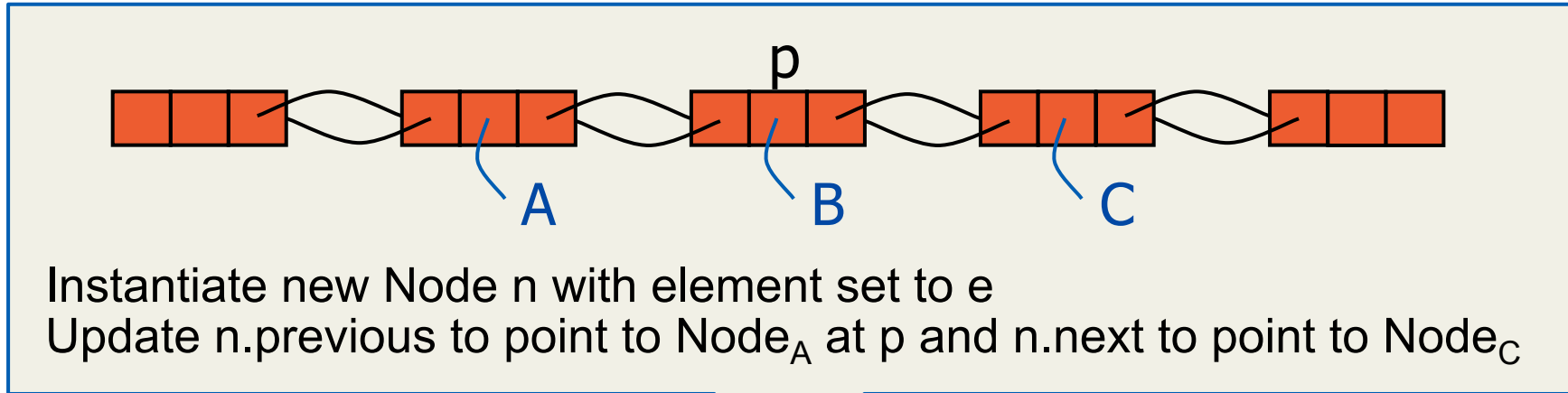
LinkedList.addAfter(p, e) [more steps]

Update Node$_B$.next and Node$_C$.previous to point to n
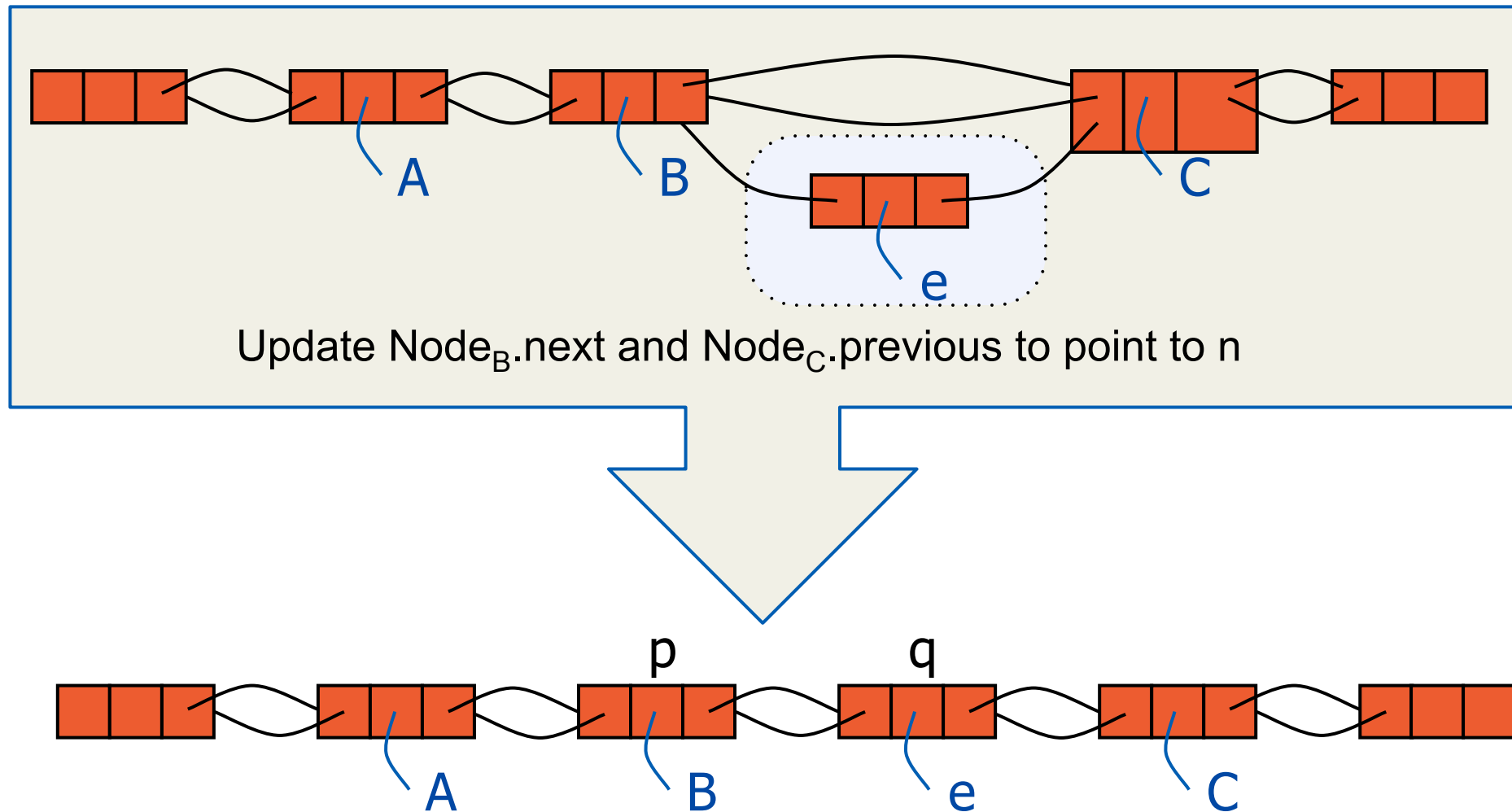
Putting it together: Deletion

Algorithm remove(p):

    t = p.element   // a temporary variable to hold the return value

    (p.getPrev()).setNext(p.getNext())   // linking out p

    (p.getNext()).setPrev(p.getPrev())

    p.setPrev(**null**)   // invalidating the position p

    p.setNext(**null**)

    return t

What is the time complexity of Deletion?

LinkedList.remove(p) [first steps]

Update Node$_C$.next to n.next and trailer.previous to n.previous
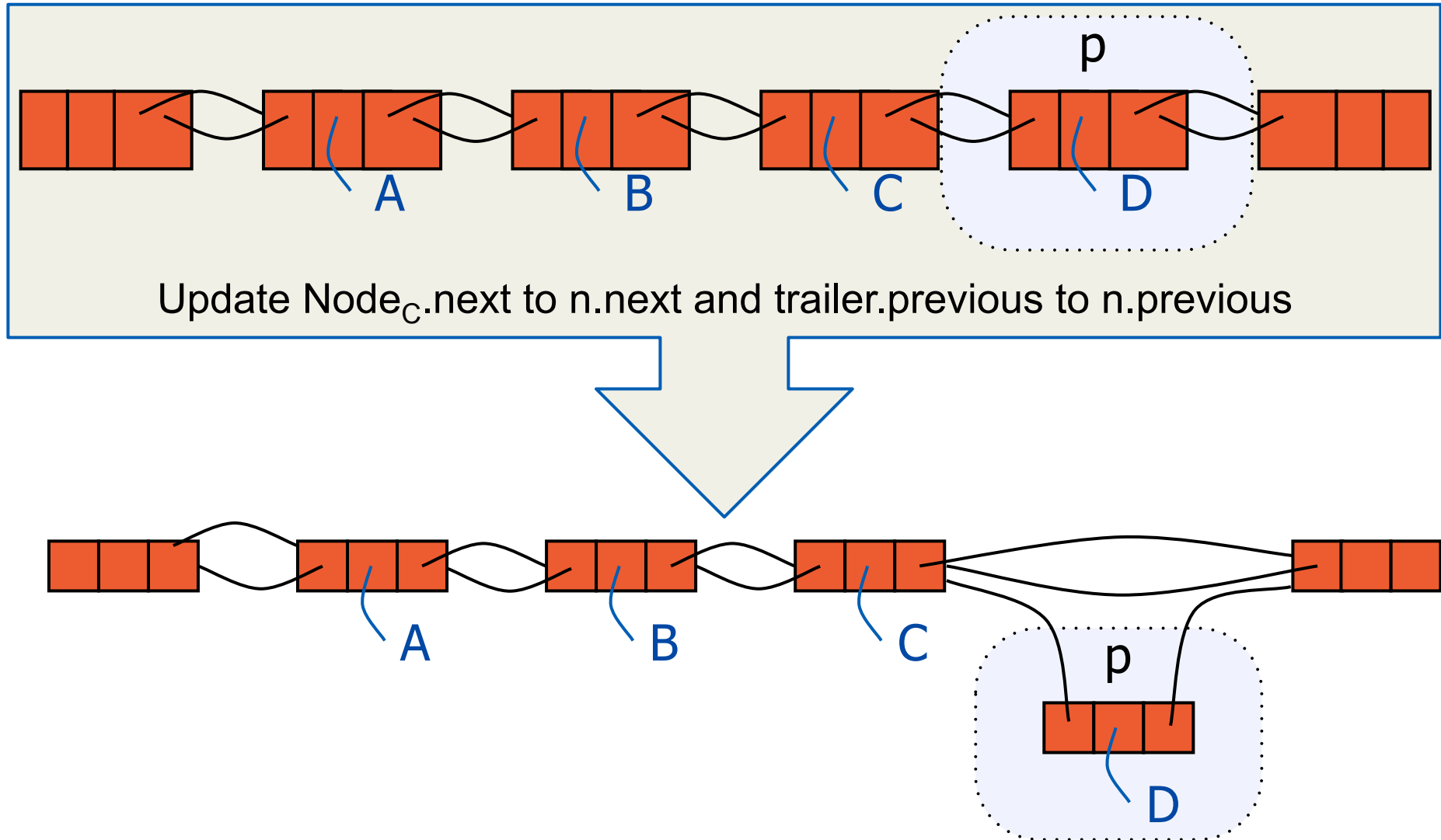
LinkedList.remove(p) [more steps]

Update Node$_D$.previous and Node$_D$.next to null

DoublyLinkedList implements PositionalList

Space complexity is O(n)

    memory need not be contiguous

Time complexity of all these (and other) PositionalList ADT operations is O(1)
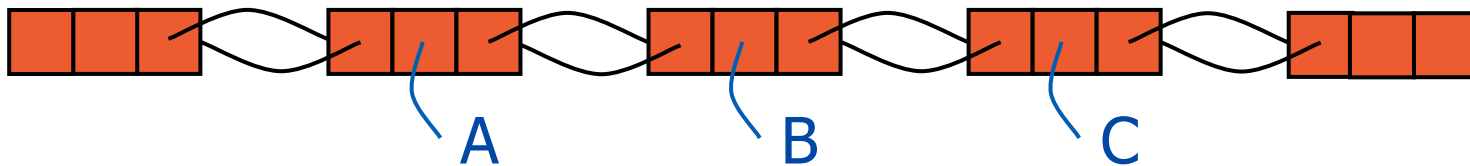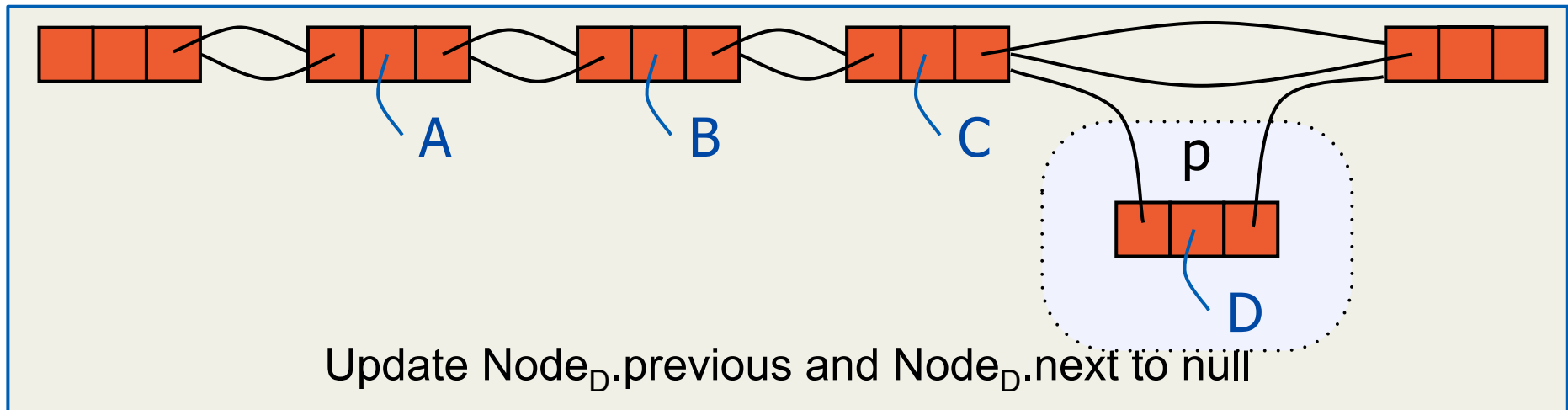
size()

isEmpty()   (boolean) whether or not the store is empty

first()   return position of first element

last()   return position of last element

before(p)   return position immediately before p

after(p)   return position immediately after p

set(p, e)   replace element at position p with element e

remove(p)   remove and return the element at position

addBefore(p,e) insert e in front of the element at position p

addAfter(p,e)   insert e following the element at position p

Array or Linked List implementation?

Linked List

- efficient insertion and deletion
- simpler behaviour as collection grows
- modifications can be made as collection iterated over
- space not wasted by list not having maximum capacity

Arrays

- caching makes traversal fast in practice
- no extra memory needed to store pointers
- allow random access (retrieve element by index)

Iterators

Yet another approach to accessing a sequence; this is how java.util.Collections are designed (sec 7.4)

Iterators

Abstracts the process of stepping through a collection of elements one at a time

Extends the concept of position by adding a **traversal** capability

Implemented by maintaining a **cursor** which points to the "current" element

Two notions of iterator:
- **snapshot** freezes the contents of the data structure
- **dynamic** follows changes to the data structure

Iterator ADT

hasNext()   (boolean) checks whether the collection is non-empty and the cursor is not after the last element

next()   returns the element after the cursor (cursor advances)

```
public interface Iterator<E> {
    public boolean hasNext();
    public E next();
}
```

Iterator ADT

hasNext()   (boolean) checks whether the collection is non-empty and the cursor is not after the last element

next()   returns the element after the cursor (cursor advances)

client code looks like this:

```
while (iterator.hasNext()){
    type value = iterator.next();
    // do something using value
}
```

© Goodrich and Tamassia

Iterable types

Iterables are objects with an iterator() method

```java
public interface Iterable<E> {
    public Iterator<E> iterator();
    }
```

An iterable ADT in Java can be represented as an interface that extends the Iterable interface

```java
public interface PositionList<E> extends Iterable<E> {
    // ...all the other methods of the list ADT ...
    /** Returns an iterator of all the elements in the list. */
    public Iterator<E> iterator();
}
```

For-Each

Java has a convenient syntax for using the iterator associated with an Iterable class

```java
List<Integer> values;
int sum = 0;
for (Integer i : values) {
  sum += i;    // implicit conversion
}
```

Java's **ListIterator ADT**

add(e)　　add e at the current cursor position

set(e)　　replace the element returned by last next()/previous method with e

remove()　remove the element returned by the last next()/previous method

next()　　return the element e after the cursor and move cursor to after e

hasNext()　(boolean) whether there is an element after the cursor

previous()　return the element e before the cursor and move cursor to before e

hasPrevious() (boolean) whether there is an element before the cursor

Implementing a List Iterator

This is implemented by our List methods plus:

Array implementation

    cursor position tracked by index i

Linked list implementation

    cursor position tracked by a pointer p to node containing
    the last element visited (start position is header sentinel)

Summary with textbook references

*Clients using an interface (ADT) rather than directly manipulating a data structure

*Four variant ADTs for ordered collections

    index access List (sec 7.1)

    position access PositionalList (sec 7.3.1, 7.3.2)

    iterator access (sec 7.4.1)

    JCF java.util.List (sec 7.5)

*Implementations of these ADTs by specific data structures

    array (simple or dynamic) (sec 3.1.1, 7.2.1, 7.2.2)

    singly-linked list (sec 3.2)

    doubly-linked list (sec 3.4, 7.3.3, 7.4.2)

*Run-time costs of the implementations

    notation for $O(1)$ and $O(n)$ (more in sec 4.1, 4.2, 4.3)

Self-check from 2016 final exam

You should now have *seen* the knowledge involved, to be able to answer these questions:

Q1(i)

Q1(ii)

Q3(a)

Q4(b) {note that you don't know enough yet for the efficient solutions that get full marks}

If you struggle with these questions now, don't worry, you still have to get the practice (from week 2 lab, and indeed more practice from later labs and from assignments too)