CH7.
LIST AND ITERATOR ADTS

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)

LISTS

LIST ADT

- A List is a general storage structure where items are accessed by index
- Main operations
  - **Element** `get(`**Index** `i)` – Returns the element of the list at index i.
  - **Element** `set(`**Index** `i,` **Element** `e)` – Replaces the element of the list at index i with element e, and returns the old element.
  - `add(`**Index** `i,` **Element** `e)` – Inserts a new element into the list so that it has index i, thus moving all subsequent elements to one index later in the list
  - **Element** `remove(`**Index** `i)` – Removes and returns the element at index i, thus moving all subsequent elements to one index earlier in the list.
- Auxiliary operations
  - `size()` – return the number of elements in the list
  - `isEmpty()` – return true if the list is empty
- An error condition occurs if any index is outside of the range $[0,$ `size()`$-1]$ (except for `add` which can add at index `size()`

EXAMPLE

- Follow along as we modify an initially empty list with the following operations

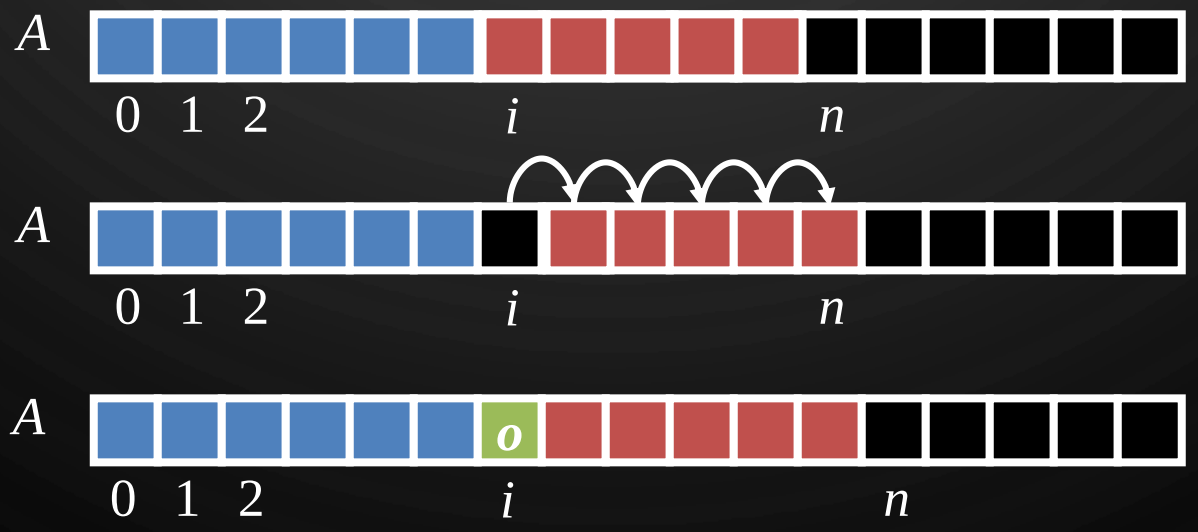
Operation	Return/Error	List Contents
`add(0, A)`	–	(A)
`add(0, B)`	–	(B, A)
`get(1)`	A	(B, A)
`set(2, C)`	Error	(B, A)
`add(2, C)`	–	(B, A, C)
`add(4, D)`	Error	(B, A, C)
`remove(1)`	A	(B, C)
`add(1, D)`	–	(B, D, C)
`add(1, E)`	–	(B, E, D, C)
`get(4)`	Error	(B, E, D, C)
`add(4, F)`	–	(B, E, D, C, F)
`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).

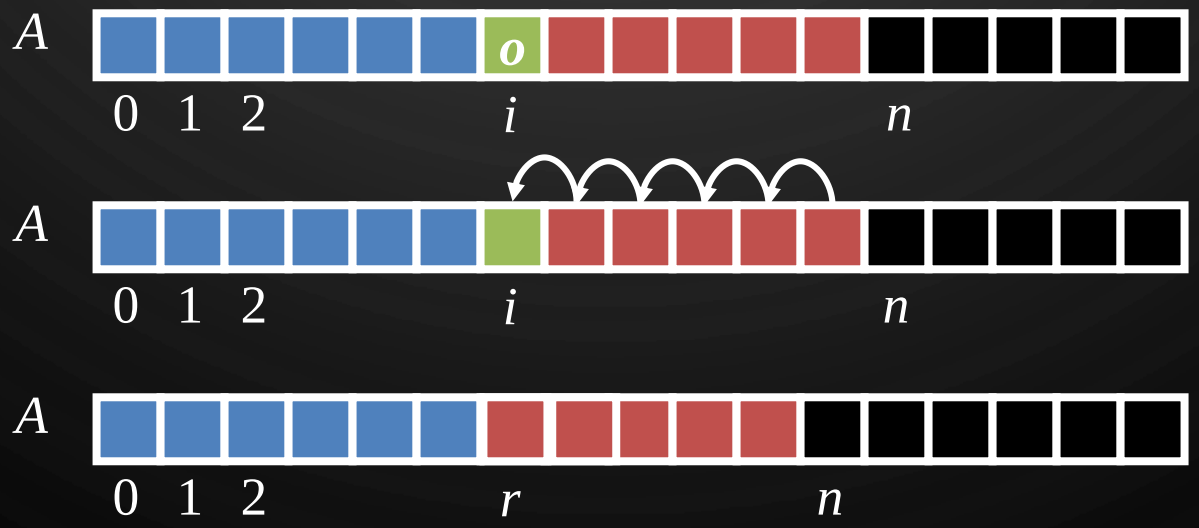
INSERTION

- 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 (`get`/`set`) 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 …

EXERCISE:

- Implement the Deque ADT with the List ADT
  - This means that you have an instance member List L, and are responsible for implementing (as pseudocode) the Deque methods
  - Deque ADT:
    - `first(), last(), addFirst(e), addLast(e), removeFirst(), removeLast(), size(), isEmpty()`
  - List functions:
    - `get(i), set(i, e), add(i, e), remove(i), size(), isEmpty()`

LIST SUMMARY

	Array Fixed-Size or Expandable	List Singly or Doubly Linked
`add(i, e)`, `remove(i)`	$O(1)$ Best Case ($i = n$) $O(n)$ Worst Case $O(n)$ Average Case	?
`get(i)`, `set(i, e)`	$O(1)$	?
`size()`, `isEmpty()`	$O(1)$	?

INTERVIEW QUESTION 1

- Implement a function to check if a list is a palindrome.

GAYLE LAAKMANN MCDOWELL, "CRACKING THE CODE INTERVIEW: 150 PROGRAMMING QUESTIONS AND SOLUTIONS", 5TH EDITION, CAREERCUP PUBLISHING, 2011.

INTERVIEW QUESTION 2

- Write code to partition a list around a value x, such that all nodes less than x come before all nodes greater than or equal to x.

GAYLE LAAKMANN MCDOWELL, "CRACKING THE CODE INTERVIEW: 150 PROGRAMMING QUESTIONS AND SOLUTIONS", 5TH EDITION, CAREERCUP PUBLISHING, 2011.

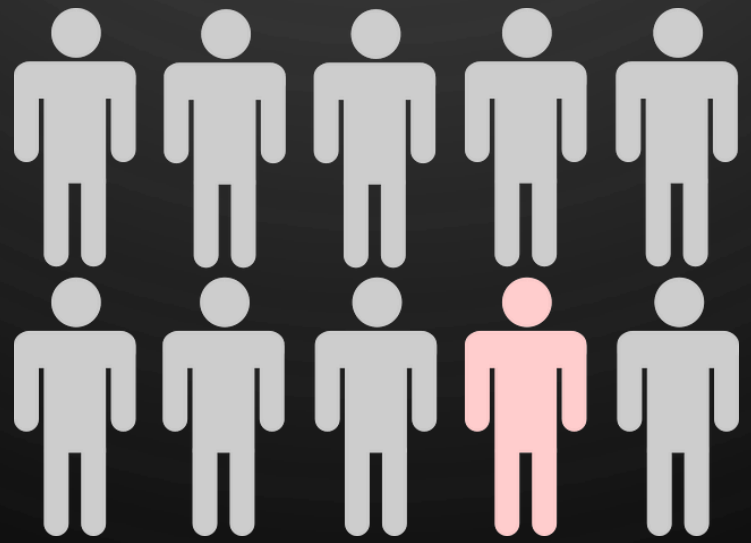
POSITIONAL LISTS

POSITIONAL LISTS

- A Positional List provides a general abstraction for a sequence of elements with the ability to identify the location of various elements
- A positional list operated with **positions** instead of indices
- A **position** is a location marker or token within the broader positional list, and is unaffected by changes elsewhere in the list

POSITION ADT

- Main (and only) operation
  - **`Element`** `getElement()` – access the element at this specific position in the data structure



AN ASIDE
SOLVING A PRACTICAL PROBLEM

- Recall Linked Structures, e.g., a linked-list
- Imagine a method returned a Node to a user of the object, and the following operations:

```
List l
Node n ← l.someNode()
n.next ← null
```

- What is the issue? Draw it out in memory.

- Positions provide a fool-proof pattern for giving the internals of a structure to a user, they are read-only. Compare with:

```
List l
Position p ← l.somePosition()
p.getElement()
{The only available operation}
```

POSITIONAL LIST ADT

- Accessor operations
  - **Position** `first()` – Returns the position of the first element (or null if empty).
  - **Position** `last()` – Returns the position of the last element (or null if empty).
  - **Position** `before(`**Position** `p)` – Returns the position immediately before a position p (or null if at the beginning of the list).
  - **Position** `after(`**Position** `p)` – Returns the position immediately after a position p (or null if at the end of the list).
- Auxiliary operations
  - `size()` – return the number of elements in the list
  - `isEmpty()` – return true if the list is empty

POSITIONAL LIST ADT

- Update operations
  - **Position** addFirst(**Element** e) – Inserts an element to the front of the list and returns the newly created position.
  - **Position** addLast(**Element** e) – Inserts an element to the end of the list and returns the newly created position.
  - **Position** addBefore(**Position** p, **Element** e) – Inserts an element immediately before position p and returns the newly created position.
  - **Position** addAfter(**Position** p, **Element** e) – Inserts an element immediately after position p and returns the newly created position.
  - **Element** set(**Position** p, **Element** e) – Replaces the element of the list at position p with element e, and returns the old element.
  - **Element** remove(**Position** p) – Removes and returns the element at position p.

EXAMPLE

- Follow along as we modify an initially empty list with the following operations

Operation	Return/Error	List Contents
`addLast(8)`	p	(p(8))
`first()`	p	(p(8))
`addAfter(p, 5)`	q	(p(8), q(5))
`before(q)`	p	(p(8), q(5))
`addBefore(q, 3)`	r	(p(8), r(3), q(5))
`r.getElement()`	3	(p(8), r(3), q(5))
`after(p)`	r	(p(8), r(3), q(5))
`before(p)`	null	(p(8), r(3), q(5))
`addFirst(9)`	s	(s(9), p(8), r(3), q(5))
`remove(last())`	5	(s(9), p(8), r(3))
`set(p, 7)`	8	(s(9), p(7), r(3))
`remove(q)`	Error	(s(9), p(7), r(3))

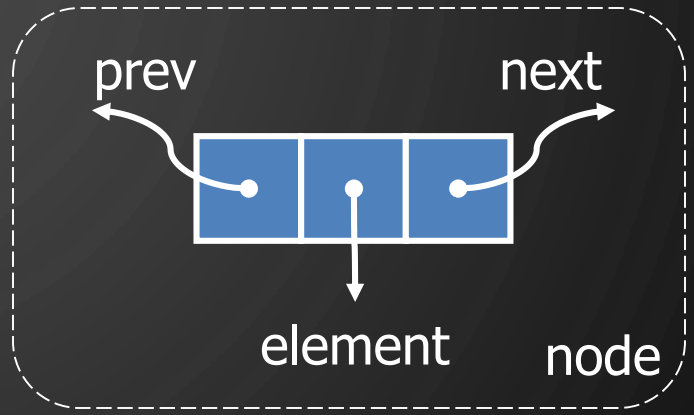
EXAMPLE
ITERATING THROUGH A POSITIONAL LIST

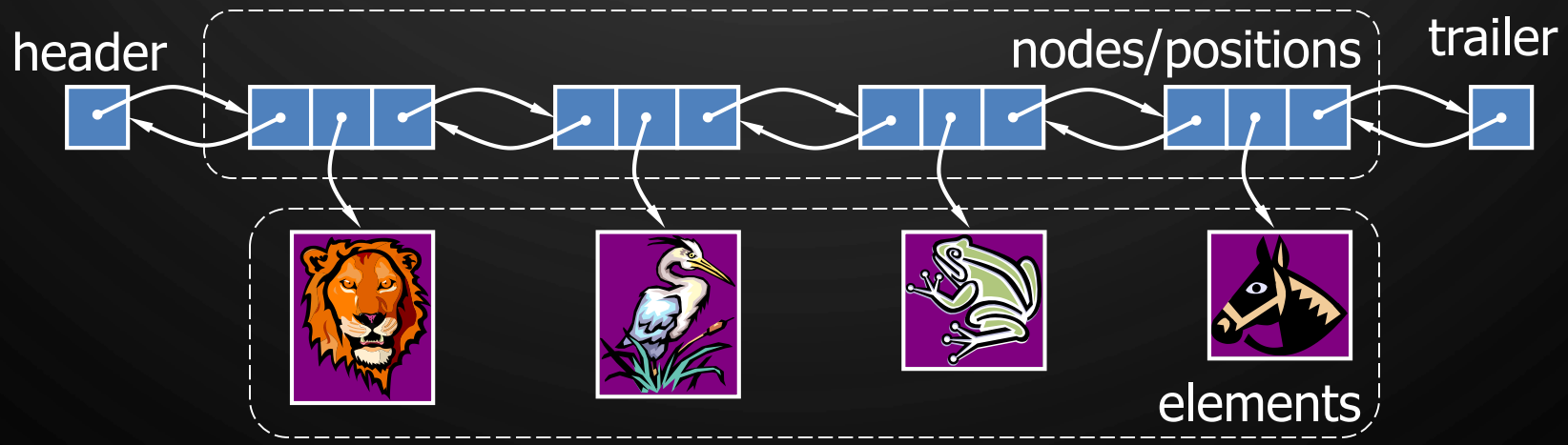
- Positional data structures are odd the first time you see them. Use this as a template for writing loops. The key is to always invoke list methods, as position has only one.

```
List l
Position p ← l.first()
while p ≠ null do
  p ← l.after(p)
```

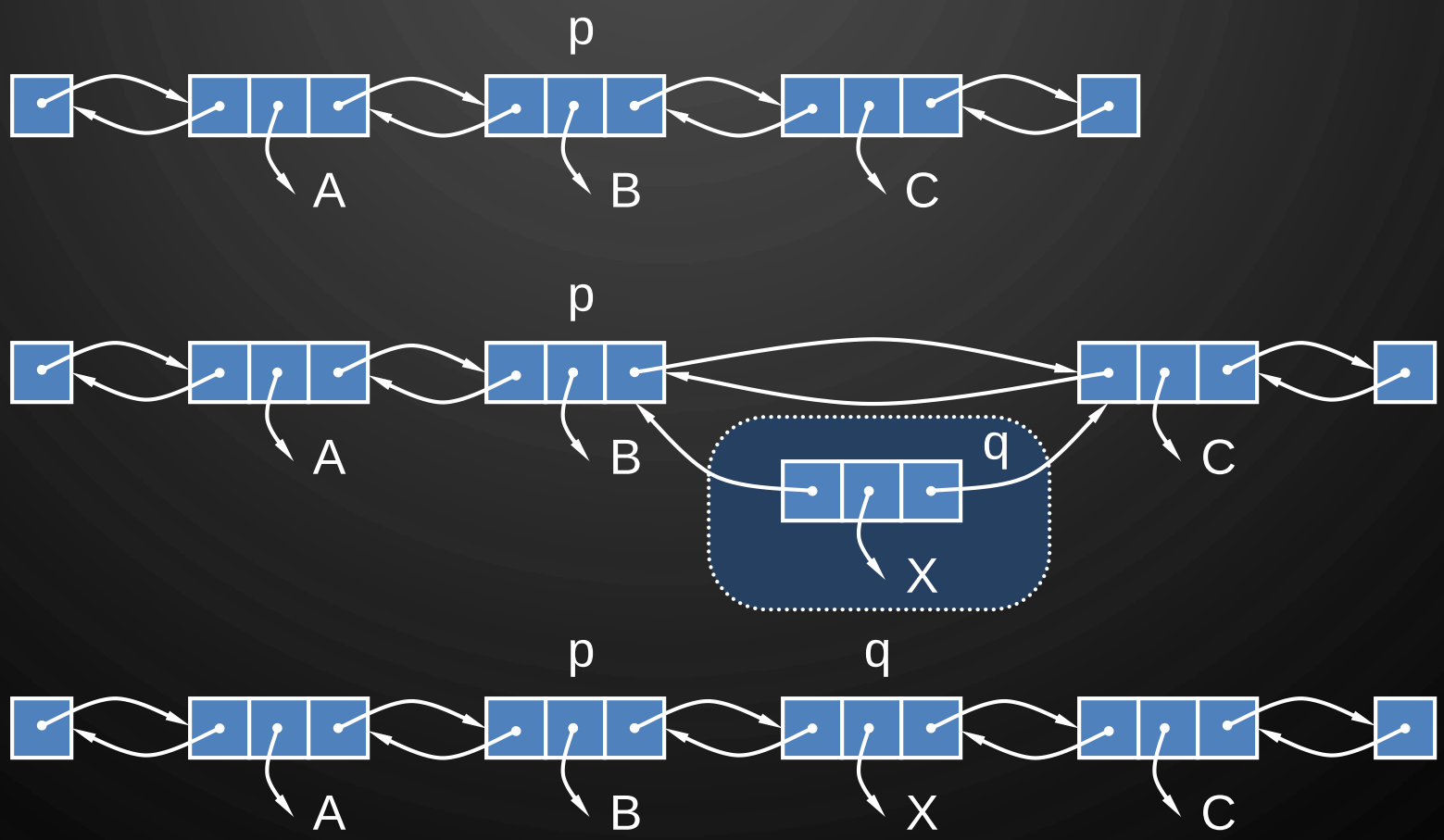
POSITIONAL LIST IMPLEMENTATION

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

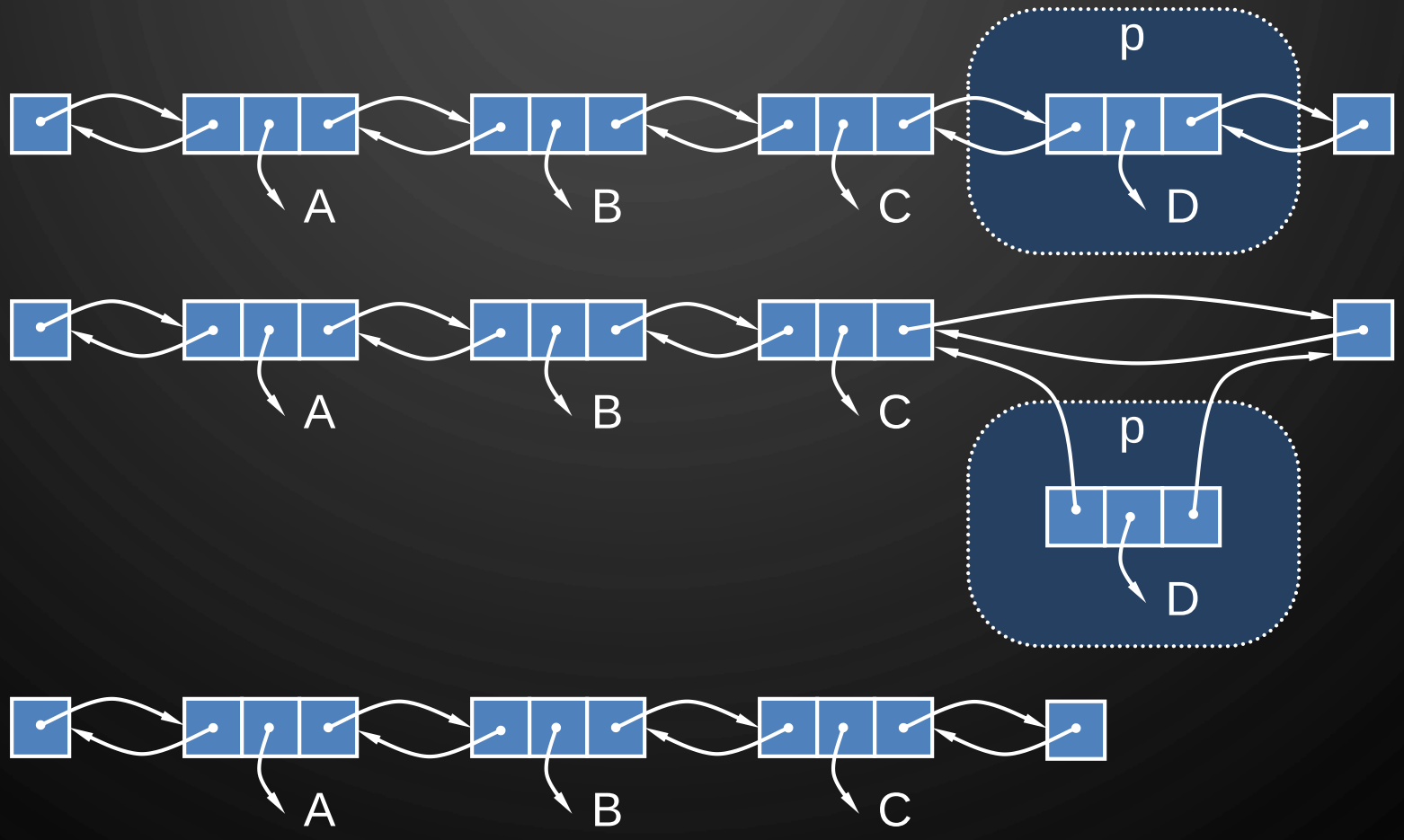




INSERTION, E.G., `ADDAFTER(P, E)`



REMOVE(P)



PERFORMANCE

- Assume doubly-linked list implementation of Positional List ADT
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the operations of the List ADT run in $O(1)$ time

POSITIONAL LIST SUMMARY

	List Singly-Linked	List Doubly- Linked
`first()`, `last()`, `addFirst()`, `addLast()`, `addAfter()`	$O(1)$	$O(1)$
`addBefore(p, e)`, `erase()`	$O(n)$ Worst and Average case $O(1)$ Best case	$O(1)$
`size( )`, `isEmpty()`	$O(1)$	$O(1)$

INTERVIEW QUESTION 3

- When Bob wants to send Alice a message M on the internet, he breaks M into n data packets, numbers the packets consecutively, and injects them into the network. When the packets arrive at Alice's computer, they may be out of order, so Alice must assemble the sequence of n packets in order before she can be sure she has the entire message. Using Positional Lists describe and analyze an algorithm for Alice to do this.
  - Can you do better with a regular List?

ITERATORS

ITERATORS

- An **iterator** is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time.
- Iterator ADT
  - **`Boolean`** `hasNext()` – returns true if there is at least one additional element in the sequence, false otherwise.
  - **`Element`** `next()` – Returns the next element in the sequence.
  - Some iterators offer a third operation: `remove()` to modify the data structure while scanning its elements

USES OF ITERATORS

- Abstracts a series or collection of elements
  - A container, e.g., List or PositionalList
  - A stream of data from a network or file
  - Data generated by a series of computations, e.g., random numbers
- Facilitate generic programming of algorithms to operate on any source of data, e.g., finding the minimum element in the data
- Why?
  - While it is true we could just reimplement minimum as many times as needed, it is better to use a trusted single implementation for: (1) correctness – no silly typos and (2) efficiency – professional libraries are often better than what you could implement on your own.

ITERABLE ADT

- An **Iterable** object is one which provides an iterator. It has a single operation:
  - **`Iterator`** `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.

EXAMPLE IN PSEUDOCODE

- The following algorithm will compute the minimum of an iterable collection:

```
Algorithm minimum
Input: Iterable collection I of comparable Elements
1. Iterator it ← I.iterator()
2. Element min ← null
3. while it.hasNext() do
4.    Element e ← it.next()
5.    if e.compareTo(min) < 0 then
6.       min ← e
7. return min
```

EXAMPLE IN JAVA

- The following code will compute the minimum of an Iterable collection:

```
1.  public static <E extends Comparable<E>> E minimum(
        Iterable<E> iterable) {
2.    Iterator<E> it = iterable.iterator();
3.    E min = null;
4.    while(it.hasNext()) {
5.      E e = it.next();
6.      if(e.compareTo(min) < 0)
7.        min = e;
8.    }
9.    return min;
10. }
```

EXERCISE

- Write an algorithm and a Java program using iterators to compute whether a collection contains only unique elements.
  - Test your generic method with both a Java ArrayList and a Java LinkedList

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) {
  // loop body
}
```

- Is equivalent to:

```
Iterator<ElementType> iter = collection.iterator();
while(iter.hasNext()) {
  ElementType variable = iter.next();
  // loop body
}
```

EXAMPLE IN PSEUDOCODE

- The following algorithm will compute the minimum of an iterable collection:

```
Algorithm minimum
Input: Iterable collection I of comparable Elements
1. Element min ← null
2. for all Element e ∈ I do
3.   if e.compareTo(min) < 0 then
4.     min ← e
5. return min
```

EXAMPLE IN JAVA

- The following code will compute the minimum of an Iterable collection:

```
1. public static <E extends Comparable<E>> E minimum(
       Iterable<E> iterable) {
2.   E min = null;
3.   for(E e : iterable) {
4.     if(e.compareTo(min) < 0)
5.       min = e;
6.   }
7.   return min;
8. }
```

EXERCISE

- Simplify your algorithm and Java program using the for-each loop construct to determine whether a collection contains only unique elements.

FOR-EACH VS ITERATORS

- For-each is not always a replacement for iterators
  - In fact it only replaces the most common use of iterators – iterating entirely through a collection
  - When you can't use a for-each loop, use iterators
    - Essentially, when you need more power, use more power
- Remember this is about generic programming. Iterators abstract the underlying collection. When you know your collection, you might be able to do something different.