List and Positional List

Aim and Learning Objectives

- To be able to understand and describe the list ADT and the positional list ADT.
- To be able to implement the list ADT and the positional list ADT, and analyze the complexity of implemented methods.
- To be able to apply (positional) lists to solve problems.

Aim and Learning Objectives

- To be able to understand and describe `growable array-based array list' or `dynamic arrays'.
- To be able to describe and compare two `growing' strategies: incremental strategy and doubling strategy.
- To be able to perform some simple *amortized analysis*.

Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th Edition, 2014.

- **Chapter 7. List Abstractions**

size()：返回列表中元素的数量。
isEmpty()：返回一个布尔值，表示列表是否为空。
get(i)：返回列表中索引为 i 的元素。如果索引 i 不在有效范围内（即不在 [0, size() - 1] 之间），则会发生错误。
set(i, e)：替换列表中索引为 i 的元素，新的元素是 e，并返回被替换的旧元素
。如果索引 i 无效（即不在 [0, size() - 1] 之间），则会发生错误。
add(i, e)：在索引为 i 的位置插入一个新的元素 e，并将索引 i 及之后的所有元素向后移动一个位置。如果索引 i 无效（即不在 [0, size()] 之间），则会发生错误。
remove(i)：移除并返回列表中索引为 i 的元素，并将索引 i 之后的所有元素向前移动一个位置。如果索引 i 无效（即不在 [0, size() - 1] 之间），则会发生错误。

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)		
add(0, B)		
get(1)		
set(2, C)		
add(2, C)		
add(4, D)		
remove(1)		
add(1, D)		
add(1, E)		
get(4)		
add(4, F)		
set(2, G)		
get(2)		

Example

add(i,e): 在i的位置加入元素e，其中i从0开始
get(i): 获取i位置的元素
set(i,e): 在i的位置上替换原来的元素成新元素e，如果没有就error
remove(i): 移除i位置的元素，没有就error

- 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)

实现 List ADT 的方法：
使用数组：数组实现列表时，元素存储在连续的内存空间中。优点：可以通过索引快速访问元素。缺点：添加或删除元素时需要移动大量元素，特别是对于数组前面或中间的元素。
使用链表：链表使用节点存储数据，每个节点包含元素数据和指向下一个节点的指针。优点：可以高效地插入和删除元素。缺点：随机访问元素较慢，必须从头节点开始逐个遍历。

How to implement the list ADT?

- Using an array?
- Using a linked list?

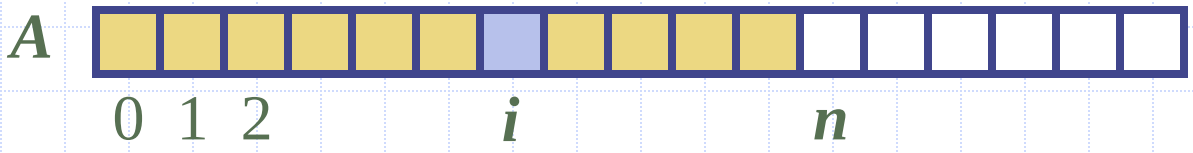
实现方法的复杂度：
O(1)：常数时间复杂度，表示某些操作如在数组末尾插入或删除元素，或者在链表的头部插入或删除元素，可以在固定时间内完成。
O(n)：线性时间复杂度，表示操作的执行时间随着数据量的增加而线性增加。例如，在链表中查找某个元素需要遍历整个链表。
O(n²)：平方时间复杂度，表示操作的执行时间随着数据量的增加而以平方的速度增长。例如，某些排序算法的复杂度是O(n²)。

What is the complexity of implemented methods?

- $O(1)? O(n)? O(n^2)?$ ...

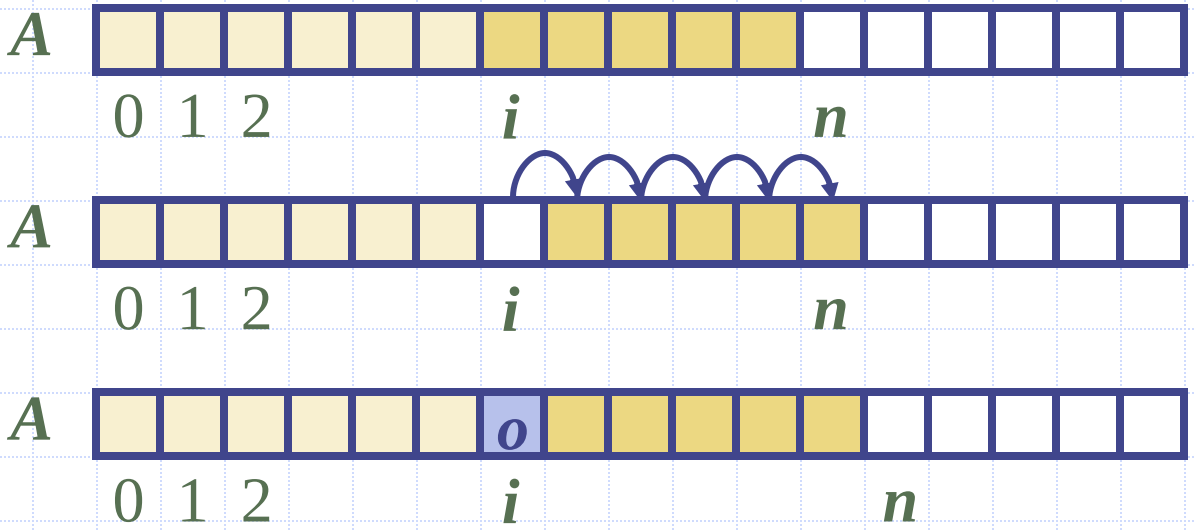
Array Lists

- 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).



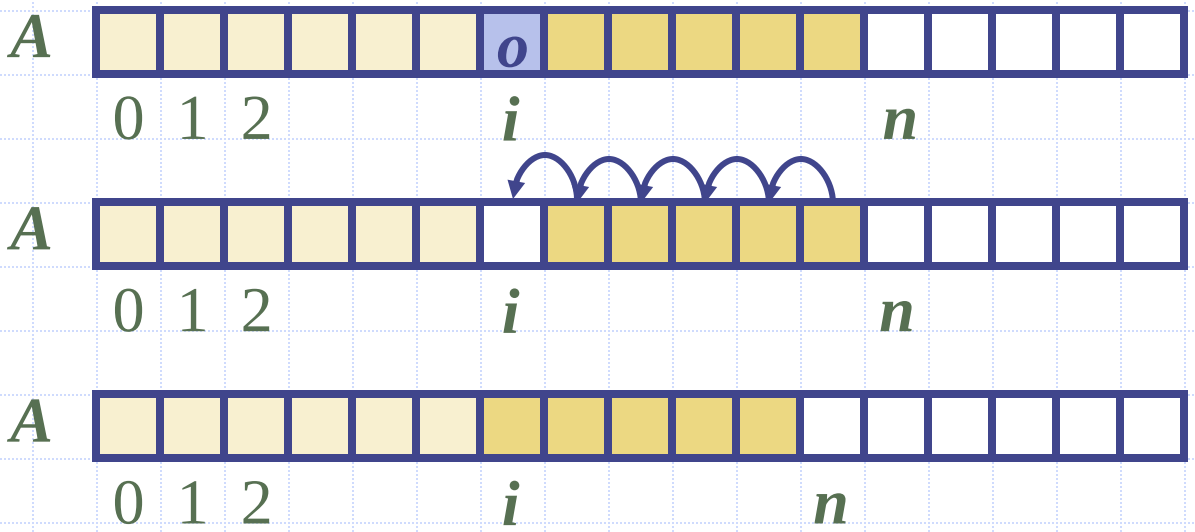
Insertion

- 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



空间复杂度：在基于数组实现的动态列表中，数据结构所占用的空间是 O(n)，其中 n 是列表中元素的数量。
索引操作：对于数组中的元素，使用索引访问元素 i 的时间复杂度是 O(1)。这意味着访问任何一个元素是常数时间操作。**插入和删除操作**：插入（add）和删除（remove）操作的时间复杂度为 O(n)。在这些操作中，最坏情况下需要移动数组中的元素，例如插入时可能需要移动所有后面的元素，删除时也可能需要移动所有前面的元素。

Performance

扩展数组：在 add 操作中，如果数组已满，通常会抛出异常。然而，一种常见的处理方式是将数组替换为一个更大的数组，而不是抛出异常。这种做法可以确保在动态扩展时不会丢失元素。

- 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* …
- *How to replace the array with a larger one?*

进一步思考：
如何替换为更大的数组：在实现上，当数组已满时，可以创建一个新的、更大的数组（通常是原数组大小的两倍），然后将旧数组中的元素复制到新数组中。这种操作的时间复杂度是 O(n)，因为需要复制所有元素。
总结：基于数组的动态列表在插入和删除时可能面临性能瓶颈，尤其是当操作涉及到元素的移动时。然而，索引操作非常高效。为了避免容量限制问题，通常会在数组满时进行扩展。

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**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$

增量策略（Incremental strategy）：按一个常量值 c 增加数组的大小。例如，如果当前数组大小为 n，则增加一个固定值 c，得到新的数组大小。
翻倍策略（Doubling strategy）：将数组的大小翻倍，即新的数组大小是当前大小的两倍。

策略比较：我们通过分析执行一系列 n 次 push 操作所需的总时间 T(n) 来比较增量策略和翻倍策略的效率。
假设：我们假设从一个空列表开始，列表用一个大小为 1 的可增长数组表示。
摊销时间（Amortized time）：
摊销时间指的是一系列操作中每个操作的平均时间。它是通过总时间 T(n) 除以操作次数 n 来计算的，即 T(n)/n。

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., $T(n)/n$

推送操作次数与数组替换：在执行 n 次 push 操作的过程中，我们将数组替换 k = n/c 次，其中 c 是一个常数。
总时间的计算：进行 n 次推送操作所需的总时间 T(n) 与以下表达式成比例：n + c + 2c + 3c + 4c + \dots + kc = n + c(1 + 2 + 3 + \dots + k)这里 n 是初始的时间复杂度，c 是常数，k 是推送操作中的数组替换次数。
总时间的简化：将和式展开并简化得到：n + c \cdot \frac{k(k + 1)}{2}这里的 k 是随着推送操作增加的，因此可以表示为逐步增加的时间。
最终的时间复杂度：因为 c 是常数，最终的时间复杂度 T(n) 为：O(n + k^2) \quad 即 \quad O(n^2)所以每次 push 操作的摊销时间为 O(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 $O(n)$

替换数组的次数：在倍增策略下，数组在执行了 k = logn 次时会被替换一次。这里，n 表示操作的数量，k 是数组替换的次数。
总时间的计算：执行 n 次 push 操作时，总时间 T(n) 需要完成以下操作：n + 1 + 2 + 4 + 8 + \dots + 2^k = n + 2^{k+1} - 1这里，k 是指数，n 是 push 操作的次数。通过求和，得到的时间复杂度是：T(n) = 3n - 1
时间复杂度：最终的时间复杂度 T(n) 是线性的，O(n)。即使在数组扩展时，操作的时间也是线性增长的。
摊销时间（Amortized Time）：每次 push 操作的摊销时间为 O(1)，意味着虽然某些 push 操作可能会涉及到数组扩展，但这些操作的平均时间仍然是常数级别的。

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

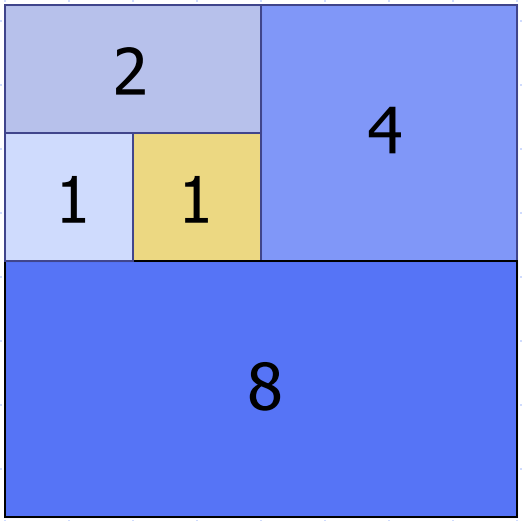
$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

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

等比数列

$$S = a_1 \times \frac{r^{(k+1)} - 1}{r - 1}$$

geometric series



位置列表（Positional List）定义：位置列表是一种抽象数据类型（ADT），用来表示一个元素序列，并能够标识每个元素的位置。
位置作为标记：位置在列表中充当标记或指示符，指示元素在整个位置列表中的位置。
位置的稳定性：位置本身不受列表其他部分变化的影响。除非显式执行删除命令，否则位置实例不会失效。
位置实例：位置实例本质上是一个简单的对象，它支持一个方法：p.getElement()。该方法返回存储在位置 p 上的元素。

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.

first()：返回列表 L 中第一个元素的位置。如果列表为空，则返回 null。
last()：返回列表 L 中最后一个元素的位置。如果列表为空，则返回 null。
before(p)：返回位置 p 前一个元素的位置。如果 p 是第一个位置，则返回 null。**after(p)**：返回位置 p 后一个元素的位置。如果 p 是
最后一个位置，则返回 null。
isEmpty()：返回布尔值，指示列表 L 是否为空。
size()：返回列表 L 中元素的数量。

addFirst(e)：在列表的开头插入一个新元素 e，并返回新元素的位置。
addLast(e)：在列表的末尾插入一个新元素 e，并返回新元素的位置。
addBefore(p, e)：在位置 p 之前插入一个新元素 e，并返回新元素的位置。
addAfter(p, e)：在位置 p 之后插入一个新元素 e，并返回新元素的位置。
set(p, e)：替换位置 p 处的元素为新元素 e，并返回被替换的元素。
remove(p)：删除位置 p 处的元素，并返回该元素，同时使该位置无效。

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)		
first( )		
addAfter(p, 5)		
before(q)		
addBefore(q, 3)		
r.getElement( )		
after(p)		
before(p)		
addFirst(9)		
remove(last( ))		
set(p, 7)		
remove(q)		

Example

addLast(8)：在列表的末尾插入元素 8，返回的位置对象是 p，表示 8 的位置。
first()：获取列表中第一个元素的位置，返回位置对象 p，因为 p 是第一个元素的位置。
addAfter(p, 5)：在 p 所在的位置后插入元素 5，返回位置对象 q，即 5 插入后的位置。
before(q)：获取 q 位置之前的元素位置，返回位置对象 p，即 q 前的那个元素的位置。
addBefore(q, 3)：在 q 之前插入元素 3，返回位置对象 r，即新插入的 3 的位置。

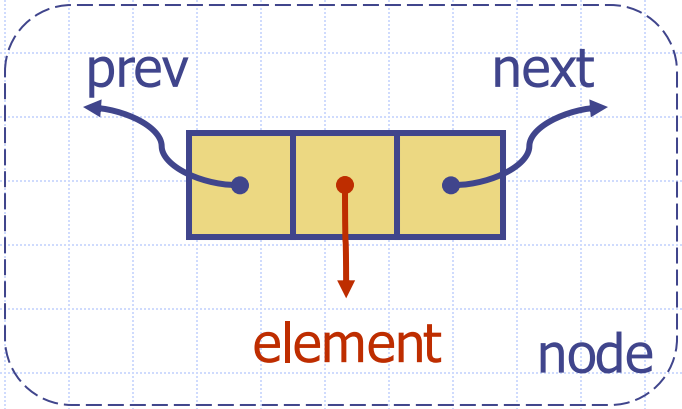
A sequence of Positional List operations:

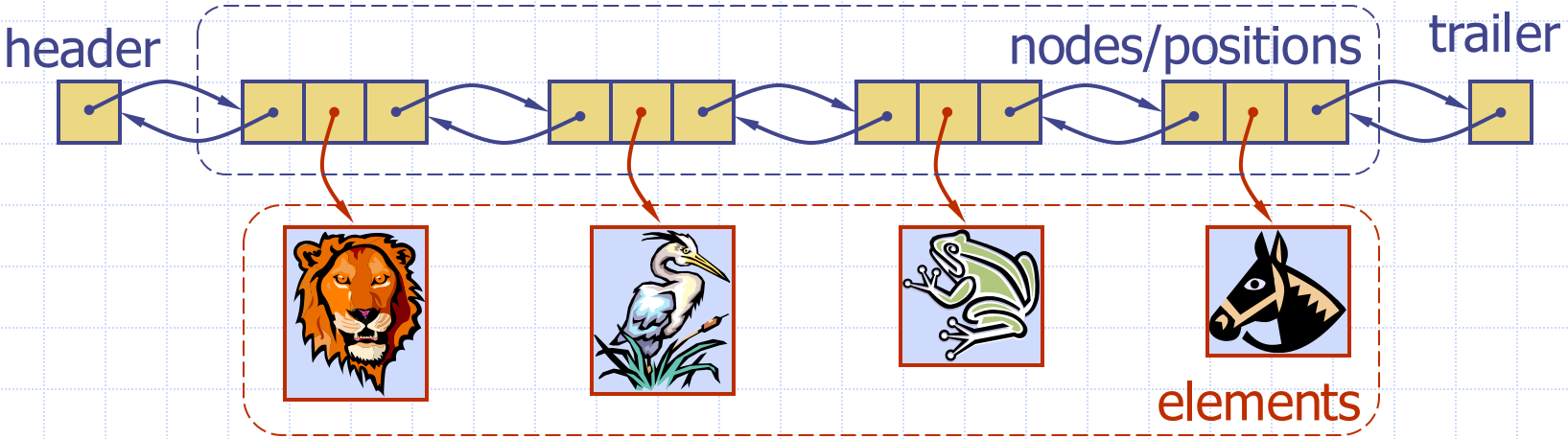
结论：
这些位置标识符（如 p、q、r、s）是用于跟踪和表示元素在列表中的位置。在进行插入、删除、查询等操作时，程序通过这些位置对象来定位操作对象的精确位置。

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

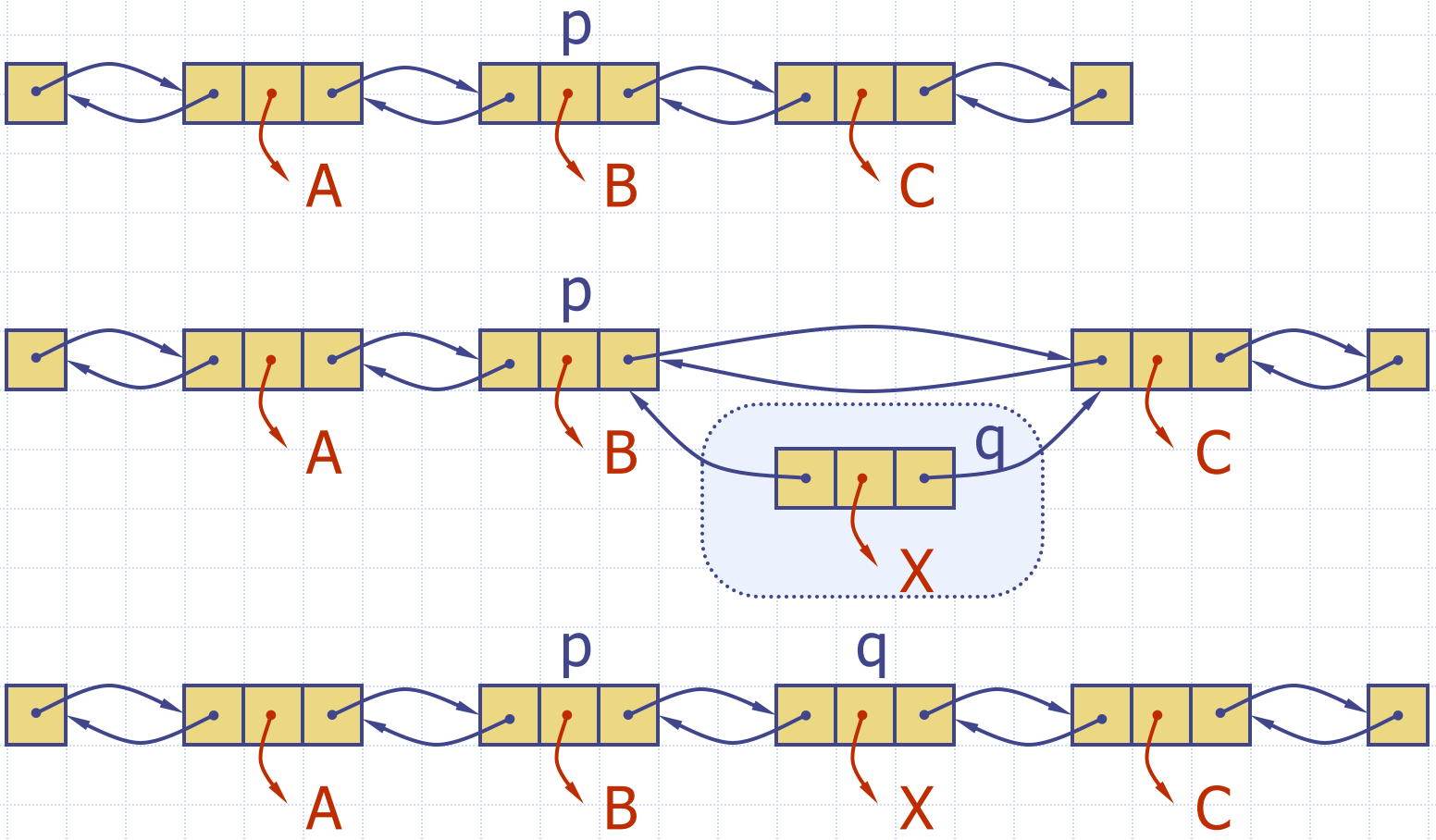
- The most natural way to implement a positional list is with a doubly-linked list.





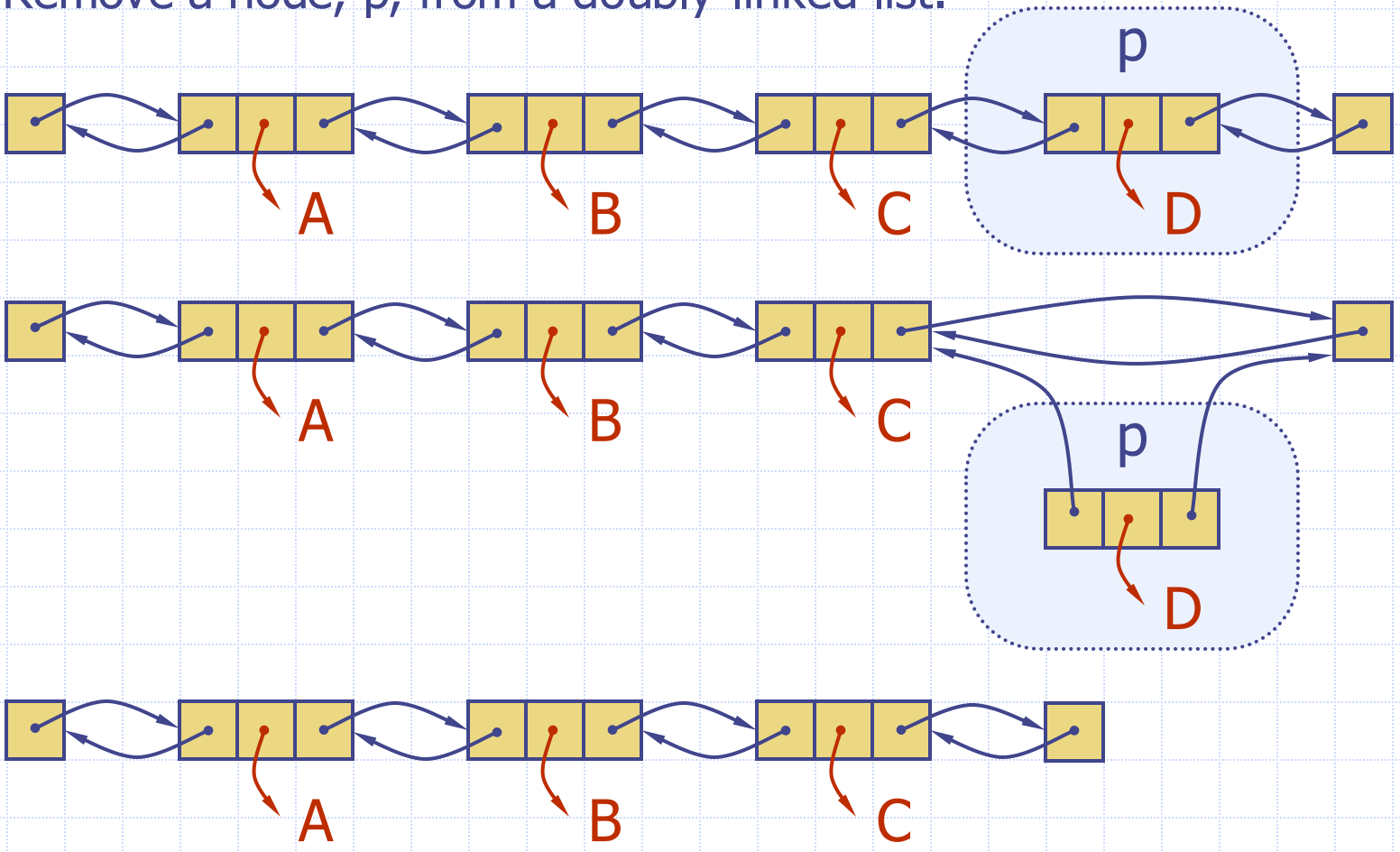
Insertion

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



Deletion

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



Iterators

hasNext():**作用**：检查序列中是否还有下一个元素。**返回值**：如果序列中还有至少一个元素，返回 true，否则返回 false。
next():**作用**：返回序列中的下一个元素。**返回值**：返回当前元素，并移动到下一个元素。

- 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"
}
```

Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th Edition, 2014.

- **Chapter 7. List Abstractions**