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**

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

- 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)

示例执行解析

- add(0, A) → 在索引 0 处插入 A，列表变为 [A]。
- add(0, B) → 在索引 0 处插入 B，原 A 后移，列表变为 [B, A]。
- get(1) → 返回 A。
- set(2, C) → **错误（索引 2 超出范围）**。
- add(2, C) → 在索引 2 处插入 C，列表变为 [B, A, C]。
- remove(1) → 删除索引 1 的 A，列表变为 [B, C]。
- set(2, G) → 修改索引 2 处的 C 为 G，列表变为 [B, E, G, C, F]。

总结

- **列表（List）是有序的数据结构**，支持插入、删除、更新、访问等操作。
- **索引超出范围时，get()、set() 和 add() 会抛出异常。**
- **删除操作 remove(index) 之后，后续元素会前移**，确保索引连续。
- **add(index, element) 会导致后续元素右移**，而 set(index, element) 只是修改值，不影响结构。

✅ **列表广泛应用于数据管理、动态数据存储、数据缓存等场景。**

关键操作说明

1. **add(index, element)**
   - 在 index 位置**插入**元素 element，**并将原有元素后移。**
   - 如果 index 超出范围，**会抛出异常。**
2. **get(index)**
   - 返回索引 index 处的元素。
   - 若 index 超出范围，**会抛出异常。**
3. **set(index, element)**
   - **替换**索引 index 处的元素为 element。
   - 若 index 超出范围，**会抛出异常。**
4. **remove(index)**
   - **删除**索引 index 处的元素，**并将后续元素前移。**

How to implement the list ADT?

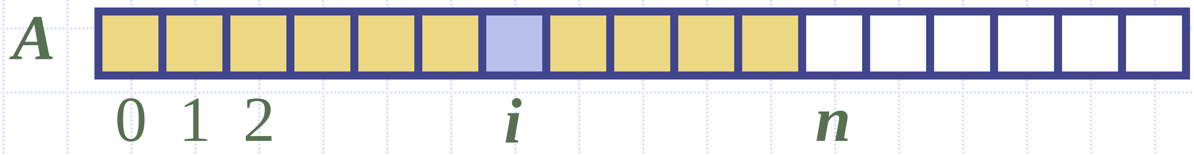
- Using an array?
- Using a linked list?

What is the complexity of implemented methods?

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

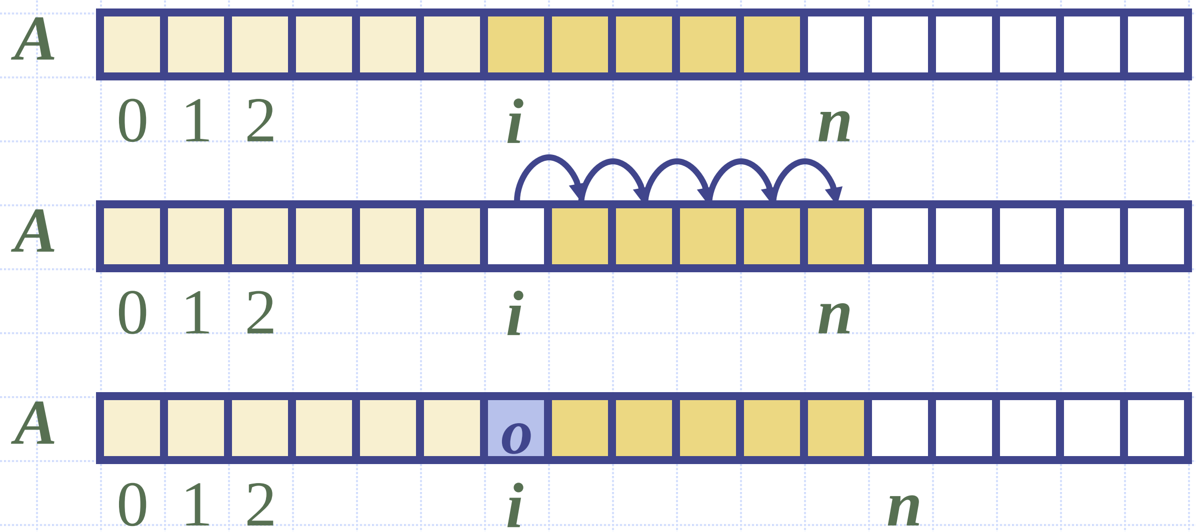
Array Lists

- 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).



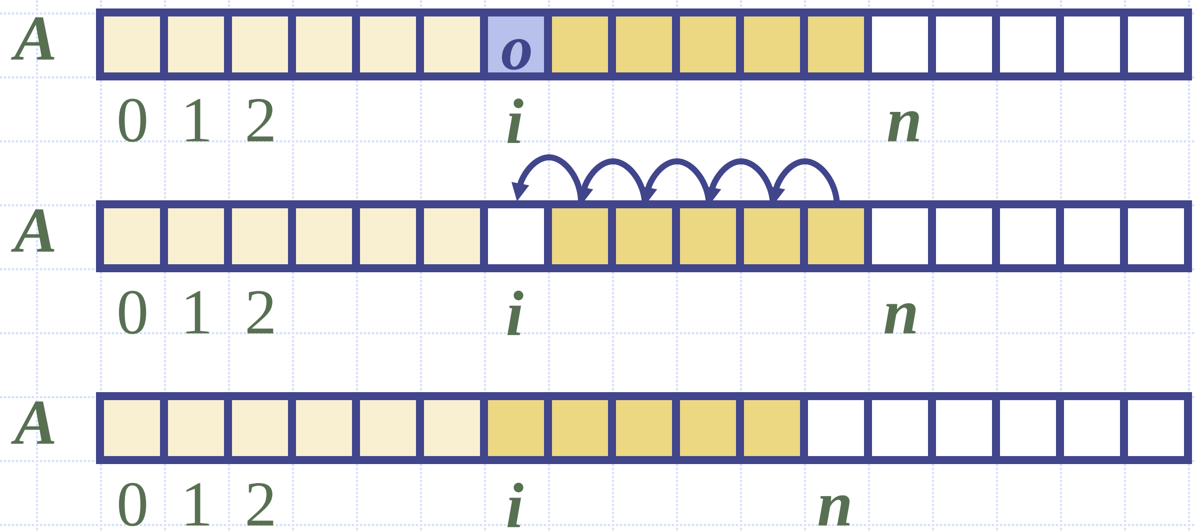
Insertion

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

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

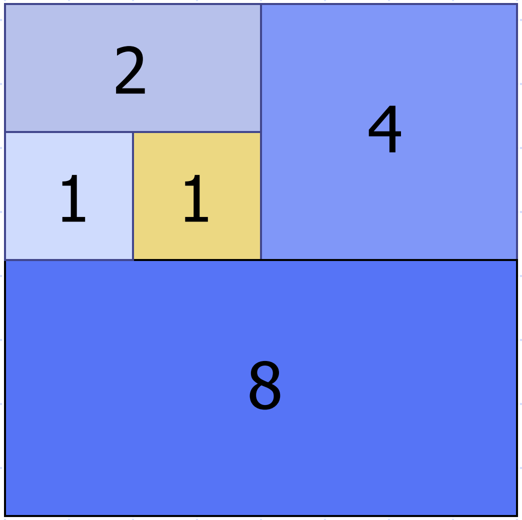
- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

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

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

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

geometric series



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

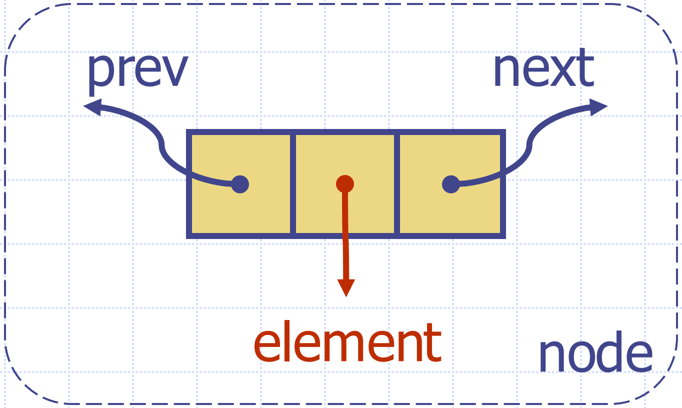
Example

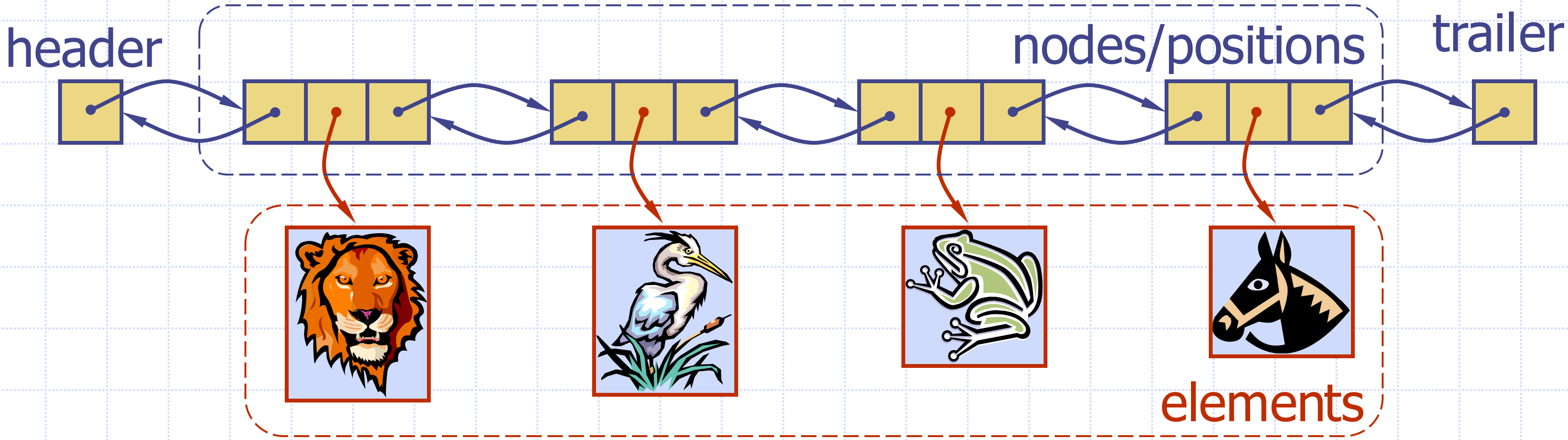
- A sequence of Positional List operations:

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

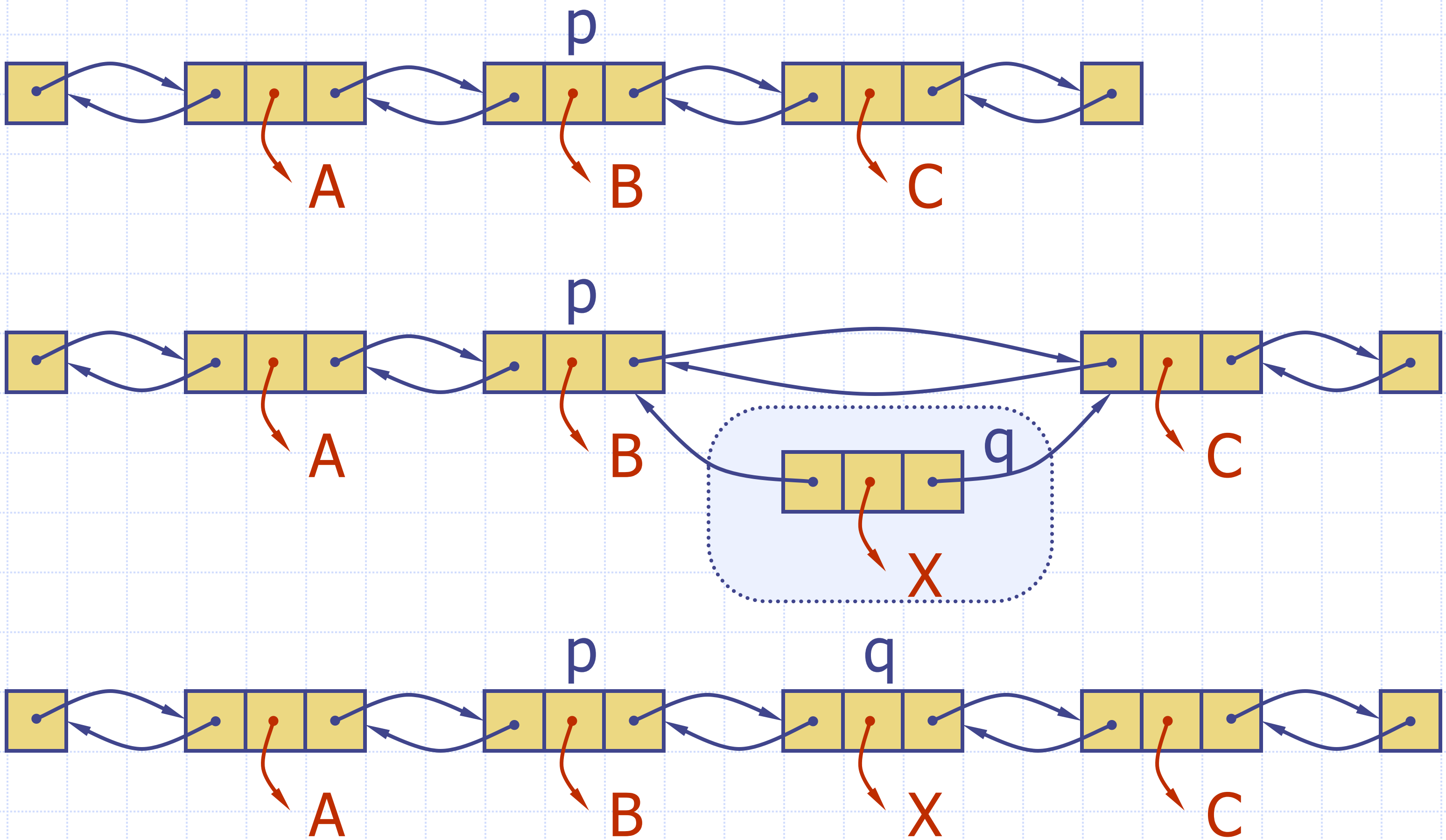
- The most natural way to implement a positional list is with a doubly-linked list.





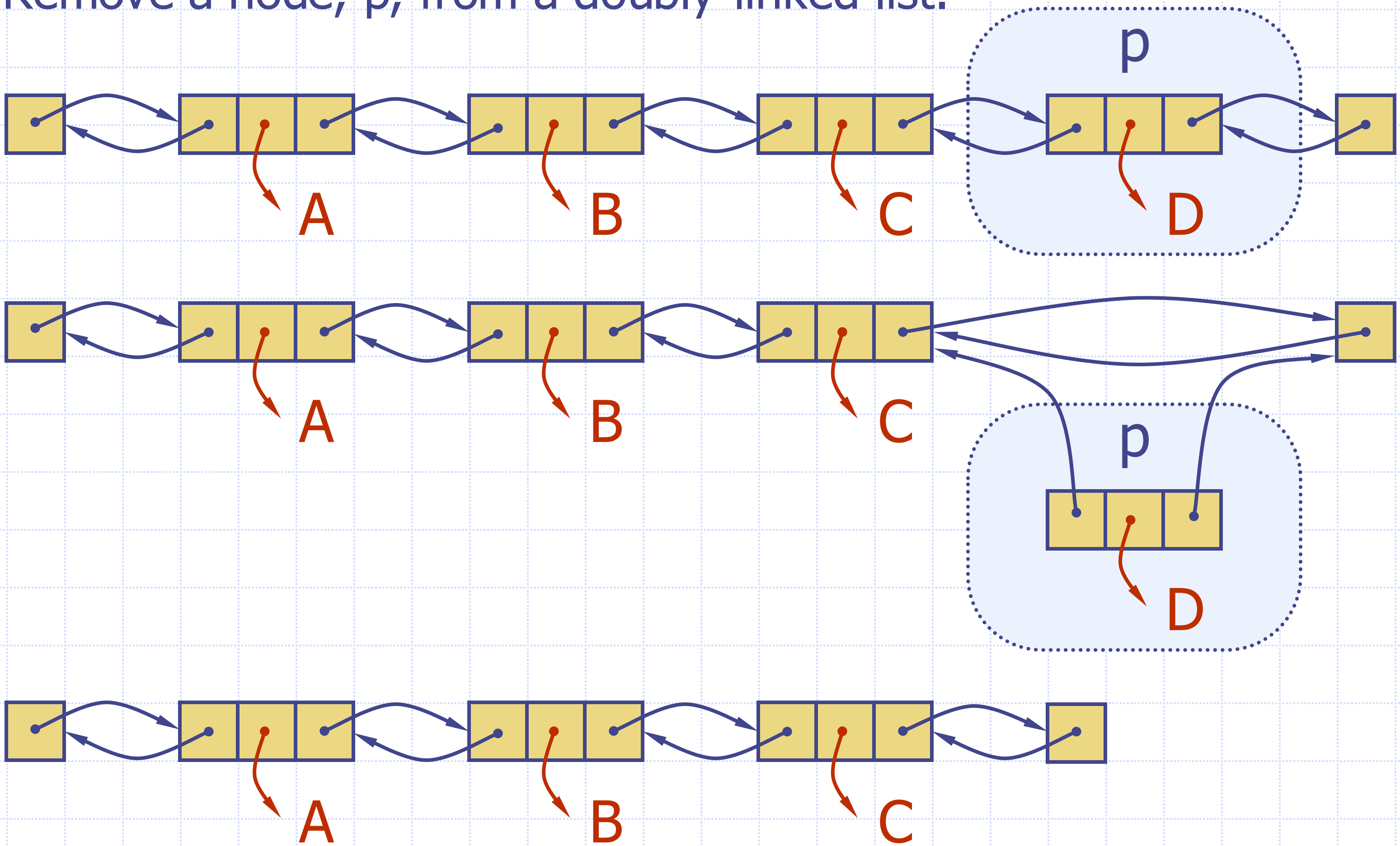
Insertion

- 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"
}
```

Reading

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

- **Chapter 7. List Abstractions**