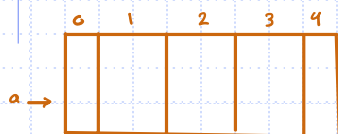


Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

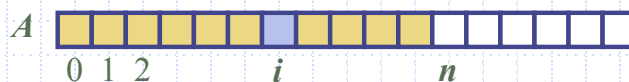
Arrays →

معان في الذائرة
صتابع المناوين



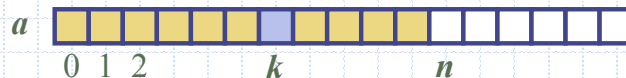
Array Definition

- An **array** is a sequenced collection of variables all of the same type. Each variable, or **cell**, in an array has an **index**, which uniquely refers to the value stored in that cell. The cells of an array, A , are numbered 0, 1, 2, and so on.
- Each value stored in an array is often called an **element** of that array.



Array Length and Capacity

- Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its **capacity**. *السعة*
 - In Java, the length of an array named *a* can be accessed using the syntax *a.length*. Thus, the cells of an array, *a*, are numbered 0, 1, 2, and so on, up through *a.length-1*, and the cell with index *k* can be accessed with syntax *a[k]*. *الطول التي تعرفه على array*
- how to refer to the last element in the array?*
نحوه لا-ال-element في ال-array



Declaring Arrays (first way)

- The first way to create an array is to use an assignment to a literal form when initially declaring the array, using a syntax as:

```
type arrayName = { initialValue0, initialValue1, ..., initialValueN-1 };
```

نحوه ال-value ونفس ال-type

- The *elementType* can be any Java base type or class name, and *arrayName* can be any valid Java identifier. The initial values must be of the same type as the array.

```
Ex: int [] a = { 5, 7, 9, 103 };
```



← العنصر → العنصر التي الاوبجكت العدد
Element TSPe [] a = new element TSPe [length]

Declaring Arrays (second way)

- The second way to create an array is to use the **new** operator.
 - However, because an array is not an instance of a class, we do not use a typical constructor. Instead we use the syntax:


```
new elementType[length]
```
- *length* is a positive integer denoting the length of the new array.
- The **new** operator returns a reference to the new array, and typically this would be assigned to an array variable.

© 2014 Goodrich, Tamassia, Goldwasser Arrays 5

Arrays of Characters or Object References

- An array can store **primitive** elements, such as characters.

S	A	M	P	L	E
0	1	2	3	4	5
- An array can also store references to objects.

String [] a = new String []

0 1 2 3 4 5

© 2014 Goodrich, Tamassia, Goldwasser Arrays 6

Java Example: Game Entries

- A game entry stores the name of a player and her best score so far in a game

```

1 public class GameEntry {
2     private String name;           // name of the person earning this score
3     private int score;            // the score value
4     /** Constructs a game entry with given parameters. */
5     public GameEntry(String n, int s) {
6         name = n;
7         score = s;
8     }
9     /** Returns the name field. */
10    public String getName() { return name; } → name الـ يرجع
11    /** Returns the score field. */
12    public int getScore() { return score; } → Score الـ يرجع
13    /** Returns a string representation of this entry. */
14    public String toString() {
15        return "(" + name + ", " + score + ")";
16    }
17 }
    
```

Java Example: Scoreboard

- Keep track of players and their best scores in an array, board
 - The elements of board are objects of class GameEntry
 - Array board is sorted by score

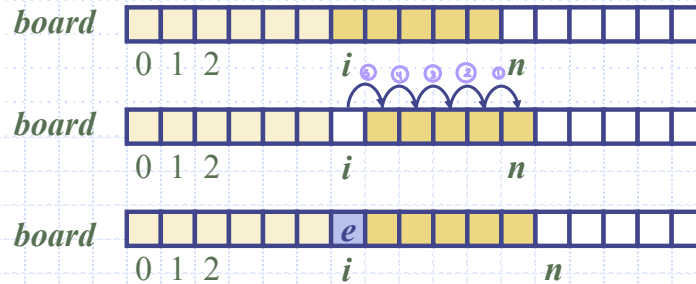
```

1 /** Class for storing high scores in an array in nondecreasing order. */
2 public class Scoreboard {
3     private int numEntries = 0; → بين ما عندني // number of actual entries
4     private GameEntry[] board; → اعدد بالذات // array of game entries (names & scores)
5     /** Constructs an empty scoreboard with the given capacity for storing entries. */
6     public Scoreboard(int capacity) {
7         board = new GameEntry[capacity];
8     }
9     // more methods will go here
36 }
    
```



Adding an Entry إضافة element جديد

- To add an entry *e* into array *board* at index *i*, we need to make إتاحة المكان room for it by shifting forward the *n - i* entries *board[i], ..., board[n - 1]*



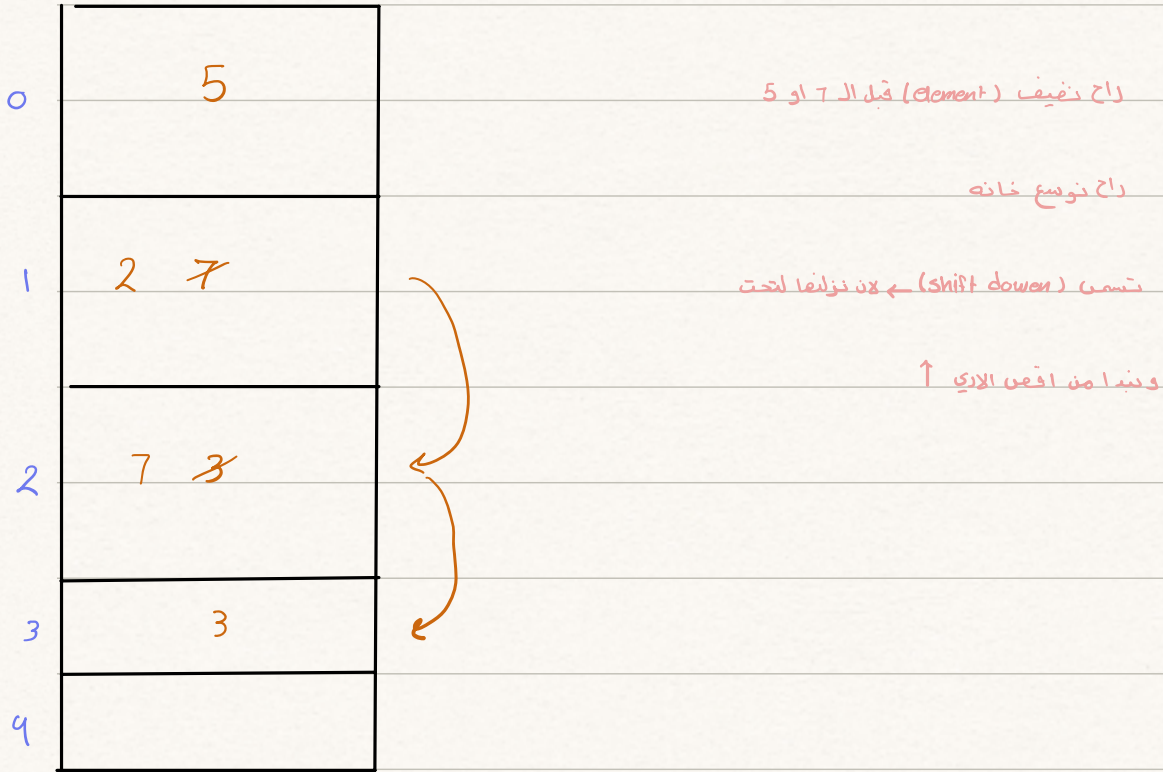
Java Example



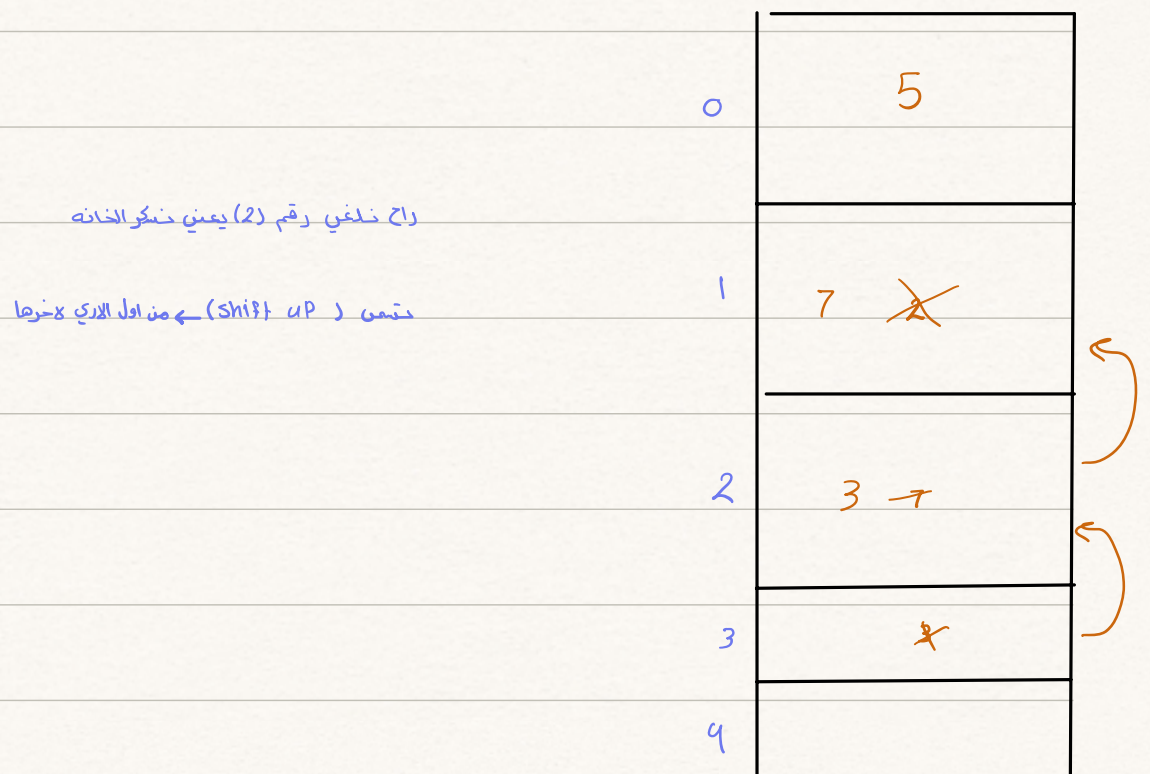
```

9  /** Attempt to add a new score to the collection (if it is high enough) */
10 public void add(GameEntry e) { → function
11     int newScore = e.getScore(); جيد ومرت على المكان
12     // is the new entry e really a high score? ↑ 100
13     if (numEntries < board.length || newScore > board[numEntries-1].getScore()) {
14         بعد ال element if (numEntries < board.length) // no score drops from the board
15             ال array numEntries++; يزيد واحد // so overall number increases
16         // shift any lower scores rightward to make room for the new entry
17         int j = numEntries - 1;
18         while (j > 0 && board[j-1].getScore() < newScore) {
19             board[j] = board[j-1]; // shift entry from j-1 to j
20             j--; // and decrement j
21         }
22         board[j] = e; // when done, add new entry
23     }
24 }
    
```

Adding an Entry:



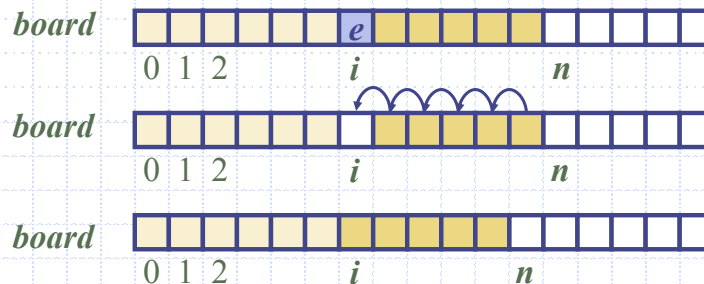
Removing an Entry:



②

Removing an Entry

- To remove the entry e at index i , we need to fill the hole left by e by shifting backward the $n - i - 1$ elements $board[i + 1], \dots, board[n - 1]$



Java Example

* دائماً يرجع لي ال element الـ طلبت عليه remove .

Function or delet

```

25  /** Remove and return the high score at index i. */
26  public GameEntry remove(int i) throws IndexOutOfBoundsException {
27      if (i < 0 || i >= numEntries)
28          throw new IndexOutOfBoundsException("Invalid index: " + i);
29      GameEntry temp = board[i];
30      for (int j = i; j < numEntries - 1; j++)
31          board[j] = board[j+1];
32      board[numEntries - 1] = null;
33      numEntries--;
34      return temp;
35  }
    
```

Handwritten annotations in Arabic and English explain the code:

- Line 27: *if (i < 0 || i >= numEntries)* - اول اكوبر متسمح الذي
- Line 28: *throw new IndexOutOfBoundsException("Invalid index: " + i);* - مسميتني راجع بالاسمي
- Line 29: *GameEntry temp = board[i];* - // save the object to be removed
- Line 30: *for (int j = i; j < numEntries - 1; j++)* - // count up from i (not down)
- Line 31: *board[j] = board[j+1];* - // move one cell to the left
- Line 32: *board[numEntries - 1] = null;* - // null out the old last score
- Line 33: *numEntries--;* - نقص لان فميت خانه ففيمعا
- Line 34: *return temp;* - // return the removed object
- Line 35: *}* - اقل بانديكس وحدة

* 3 - retrieve -> رجع لي الخانة

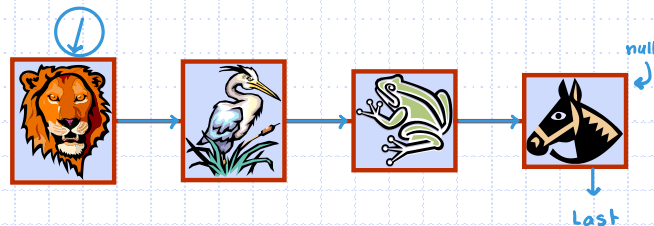
4 - traverse -> نلق على العناصر -> مرور على كل

For (int i=a; i<num.length(); i++) ← باستخدام الـ for loop (element)

board[i]

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

Singly Linked Lists

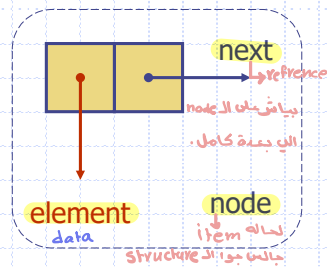


Singly Linked Lists

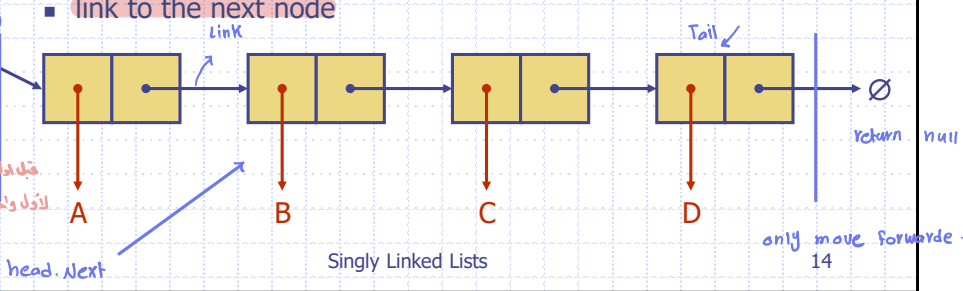
13

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
 - element
 - link to the next node



من الاول
الحركة بتابعة فبدا
في حالتنا نقدر
نحركه:
1- insert اوله
2- remove
الأول واحد



Singly Linked Lists

14

Java Methods

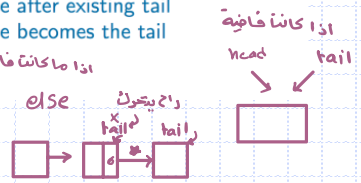


31 ← هنا الـ head ترك الشيء
 32 ← الـ head بتأخر عليه ، ودهج تبني لـ head (new node)
 33 ← ودخلني الـ head فيكي واشتري زي ما كانت اشتر
 34 ← قبل ما اشتر عليكي .
 35 ← بعد اخر (node)
 36 ← الإضافة فاضية .

```

31 public void addFirst(E e) { // adds element e to the front of the list
32     head = new Node<>(e, head); // create and link a new node
33     if (size == 0)
34         tail = head; // special case: new node becomes tail also
35     size++;
36 }
37 public void addLast(E e) { // adds element e to the end of the list
38     Node<E> newest = new Node<>(e, null); // node will eventually be the tail
39     if (isEmpty()) // Last tail
40         head = newest; // special case: previously empty list
41     else // is not Empty
42         tail.setNext(newest); // new node after existing tail
43     tail = newest; // new node becomes the tail
44     size++;
45 }
  
```

* I can't update the tail before step 2?



add element f, after (b) ?
 we creat temp

```

Node temp = head;
for ( ; temp != null; temp = temp.getNext())
    if (temp.getElement() == 'b')
    {
        Node new node = new Node('f', temp.getNext());
        size++;
        temp.setNext(new node);
        break;
    }
  
```

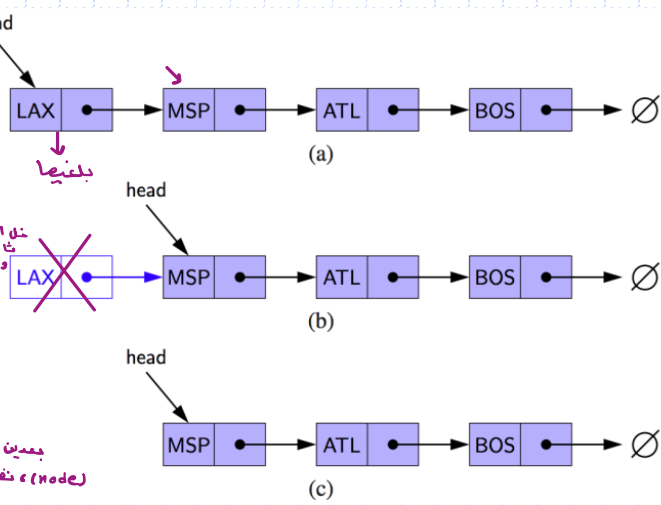
```

add f in middle
int mid = size() / 2;
Node temp = head;
for (int i = 0; i < mid; i++)
    temp = temp.getNext();
Node new node = new Node('f', temp.getNext());
size++;
temp.setNext(new node);
  
```

* temp.next = new node

Removing at the Head

- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node





1) remove head

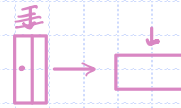
```
double old_head = head;
```

```
head = head->next;
```

```
old_head->next = null;
```

```
size--;
```

Java Method

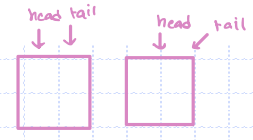


```

46 public E removeFirst() { // removes and returns the first element
47     if (isEmpty()) return null; // ماضية شيء -> nothing to remove
48     E answer = head.getElement(); // مذبذبة
49     head = head.getNext(); // will become null if list had only one node
50     size--; // طاعتت من ال list ->
51     if (size == 0) // (Node) واحد فقط ->
52         tail = null; // special case as list is now empty . ويتطلب .
53     return answer;
54 }
55 }
    
```

- 1- أول شيء واشتبه مع ال list -
- اذا فاضية اول
- 2- بتروح نخزن ال Node لـ
- الاول عندي بنحطها ونحتفظ فيها
- 3- بتحول ال head لتاني
- Node بنفس القرايش التاني
- وتمقص ال size

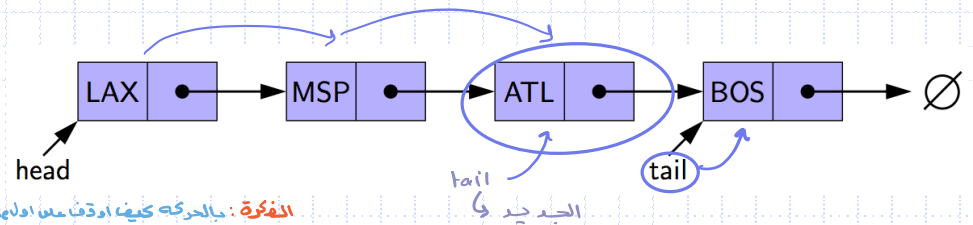
- 4- بصال ال size
- عل صوت Zero
- صماته : ما مدينا ال Node واحد
- والفيناها فكان ال tail
- بيوقف على null بظل عندي
- Node.



Singly Linked Lists

Removing at the Tail

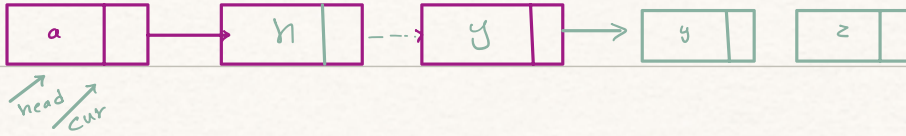
- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



الفكرة : بالحركة كيف اوقف على اول Node
 واتحرك للـ (Node) التي قبل الاخير متان اول
 للـ Node والعني (Pointer) الـ الـ

Singly Linked Lists

1) Remove the tail



1) $i < (\text{size}() - 1)$

```
for (Node cur = head ; cur.next.next != null ; cur = cur.next)
    int i = 1
```

2) Solution

```
tail = cur;
cur.next = null;
tail.next = null;
tail = cur;
size--;
size--;
```

Delete



```
Node before = temp = null;
```

```
Node temp = head;
```

```
while (temp != null)
```

```
{ if (temp.getelement == 'Sara')
```

```
{ before = temp;
  temp.next = temp.next;
  temp.next = null;
  size--;
  break; }
```

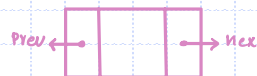
else

```
before = temp;
```

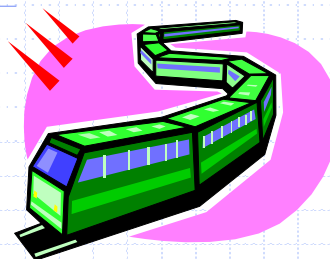
```
temp = temp.next;
```

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

Doubly Linked Lists

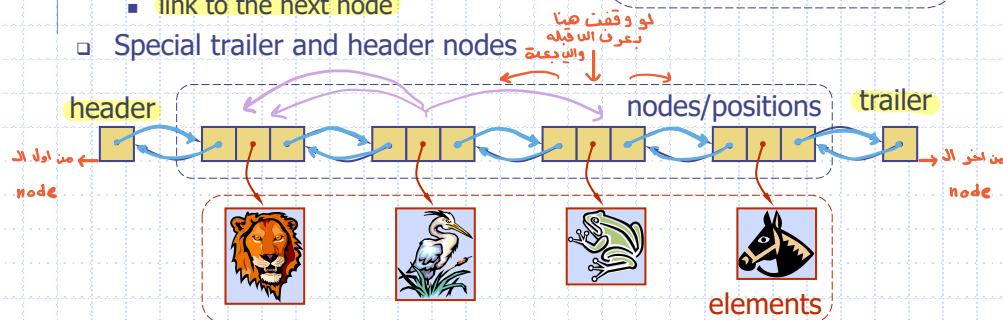
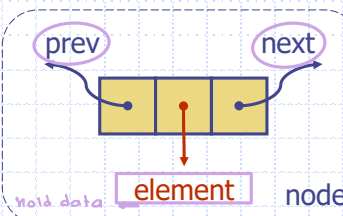


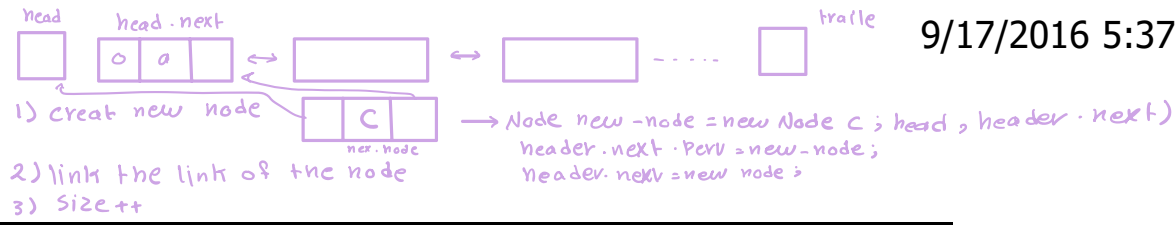
And 2 link:
Painter
 Prev : بيأثر على العقيلة
 next : بيأثر على العيلة



Doubly Linked List

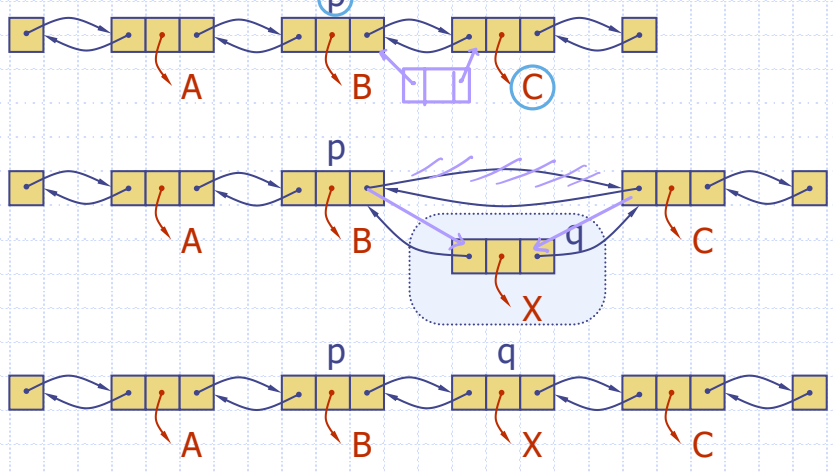
- A doubly linked list can be traversed forward and backward.
- Nodes store:
 - element
 - link to the previous node
 - link to the next node
- Special trailer and header nodes





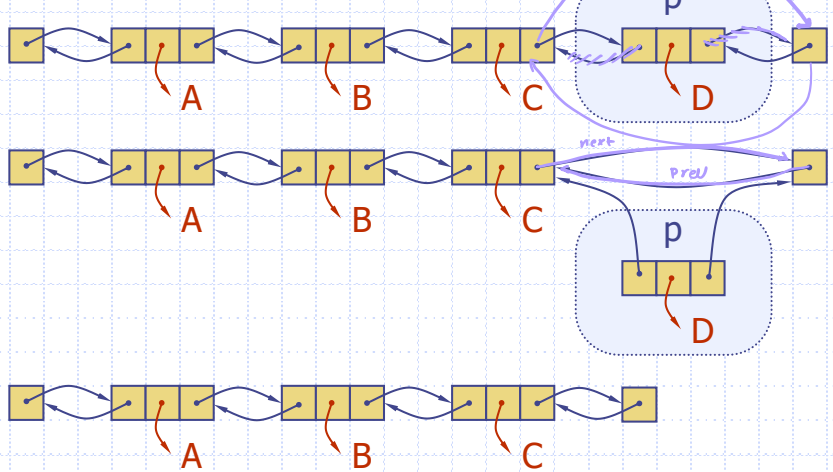
Insertion

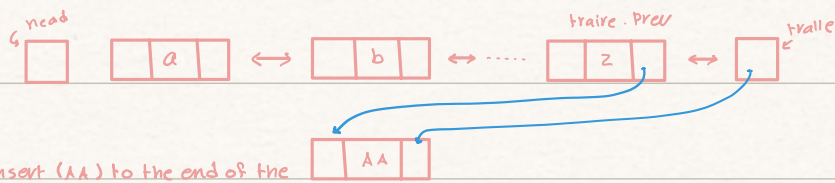
Insert a new node, q, between p and its successor.



Deletion

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





1. Insert (AA) to the end of the

create node and attach it at the list

```
Node new_node = new Node ('AA', trailer.Prev, trailer);
```

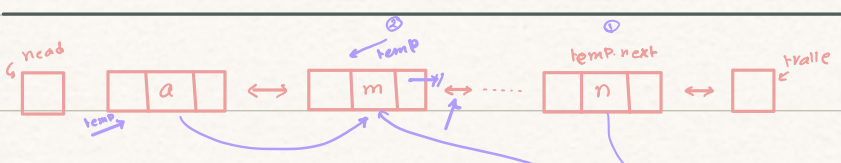
Element
Prev
next

link the list to the node

```
trailer.Prev.next = new_node;
```

update the size

```
size++;
```



1) Insert (M, after node containing 'm')

```
new_node = new node ('M', temp, temp.next);
```

Element
Prev
next

new_node

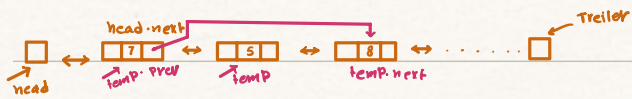
2) link the list to the node

```
temp.next.Prev = new_node;
```

```
temp.next = new_node;
```

3) size++;

Delete



1) search the element (5) and delete the node

```
for (Node temp = head.next; temp != trailer; temp = temp.next
```

```
if (temp.element == '5')
```

current node

```
temp.prev.next = temp.next;
```

```
temp.next.prev = temp.prev;
```

```
temp.prev = null;
```

```
size--;
```

Doubly-Linked List in Java

```

1  /** A basic doubly linked list implementation. */
2  public class DoublyLinkedList<E> {
3  //----- nested Node class -----
4  private static class Node<E> {
5  private E element; // reference to the element stored at this node
6  private Node<E> prev; // reference to the previous node in the list
7  private Node<E> next; // reference to the subsequent node in the list
8  public Node(E e, Node<E> p, Node<E> n) {
9  element = e;
10 prev = p;
11 next = n;
12 }
13 public E getElement() { return element; }
14 public Node<E> getPrev() { return prev; }
15 public Node<E> getNext() { return next; }
16 public void setPrev(Node<E> p) { prev = p; }
17 public void setNext(Node<E> n) { next = n; }
18 //----- end of nested Node class -----
19 }

```

اسم نفس اسم (const)
 يرجع لي قيمة ال element جو ال data
 يرجع ال prev
 يرجع ال next
 ما عندنا تغير لل data.
 لا يتم ال update
 لا يتم ال remove
 لا يتم ال add

Doubly-Linked List in Java, 2

```

21 private Node<E> header; // header sentinel
22 private Node<E> trailer; // trailer sentinel
23 private int size = 0; // number of elements in the list
24 /** Constructs a new empty list. */
25 public DoublyLinkedList() {
26 header = new Node<>(null, null, null);
27 trailer = new Node<>(null, header, null);
28 header.setNext(trailer);
29 }
30 /** Returns the number of elements in the linked list. */
31 public int size() { return size; }
32 /** Tests whether the linked list is empty. */
33 public boolean isEmpty() { return size == 0; }
34 /** Returns (but does not remove) the first element of the list. */
35 public E first() {
36 if (isEmpty()) return null;
37 return header.getNext().getElement();
38 }
39 /** Returns (but does not remove) the last element of the list. */
40 public E last() {
41 if (isEmpty()) return null;
42 return trailer.getPrev().getElement();
43 }

```

عدد ال element
 create header
 trailer is preceded by header
 header is followed by trailer
 method
 ال data لا يرجع
 ال data لا يرجع
 ال data لا يرجع
 ال data لا يرجع

Doubly-Linked List in Java, 2

عبارتين نودين

يأشرو عليهما بعين

header

trailer

Singly Linked List



* هما من جوا مش فاضين مش (Empty) او (null).

لمن بنعمل الإضافة او الحذف (العمل ذكوة):

1- انا ال (list) فاضية (null). X

2- اضيف قبل اول واحد (node). X

3- اضيف بعد اخر (node). X

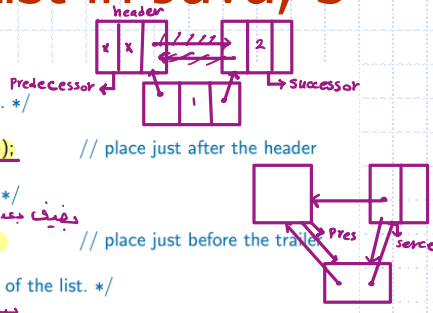
1- الإضافة بالوسط. ✓

2- الحذف من الوسط. ✓

Doubly-Linked List in Java, 3

```

44 // public update methods
45 /** Adds element e to the front of the list. */
46 public void addFirst(E e) {
47     addBetween(e, header, header.getNext()); // place just after the header
48 }
49 /** Adds element e to the end of the list. */
50 public void addLast(E e) {
51     addBetween(e, trailer.getPrev(), trailer); // place just before the trailer
52 }
53 /** Removes and returns the first element of the list. */
54 public E removeFirst() {
55     if (isEmpty()) return null; // nothing to remove
56     return remove(header.getNext()); // first element is beyond header
57 }
58 /** Removes and returns the last element of the list. */
59 public E removeLast() {
60     if (isEmpty()) return null; // nothing to remove
61     return remove(trailer.getPrev()); // last element is before trailer
62 }
    
```

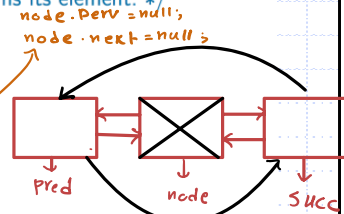
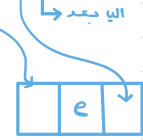


* جانب اول كم عدد الـ nodes التي في doubly linked ؟
 تكون حريصة في ترتيب زيادة الـ (header و Trailer)

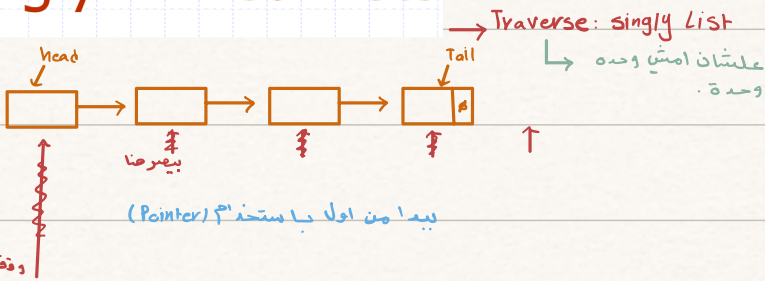
Doubly-Linked List in Java, 4

```

64 // private update methods
65 /** Adds element e to the linked list in between the given nodes. */
66 private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
67     // create and link a new node
68     Node<E> newest = new Node<>(e, predecessor, successor);
69     predecessor.setNext(newest);
70     successor.setPrev(newest);
71     size++;
72 }
73 /** Removes the given node from the list and returns its element. */
74 private E remove(Node<E> node) {
75     Node<E> predecessor = node.getPrev();
76     Node<E> successor = node.getNext();
77     predecessor.setNext(successor);
78     successor.setPrev(predecessor);
79     size--;
80     return node.getElement();
81 }
82 } //----- end of DoublyLinkedList class -----
    
```



Singly Linked Lists



```
Node < E > P = head;
```

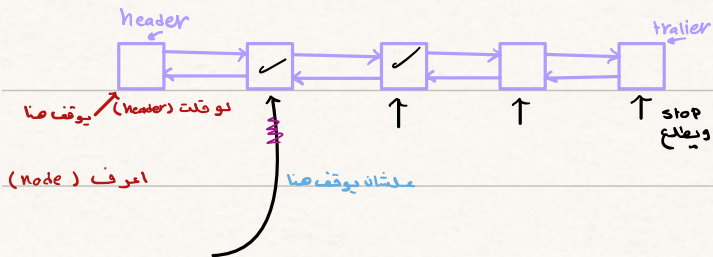
```
While ( P != null ) → جواز ال P لاصغري (null) يكون  
تمتتيا.
```

```
system.out.println ( P.getElement() );
```

```
P = P.getNext(); → داخ بتحرك ال P
```

3

Doubly Linked Lists → Traverse



```
Node < E > P = header.getNext();
```

```
While ( P != Trailer )  
ع
```

```
system.out.println ( P.getElement() );
```

```
P = P.getNext();
```

3

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

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 $[0, \text{size}() - 1]$.

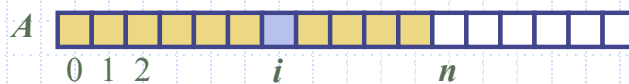
Example

- 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)	–	(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)

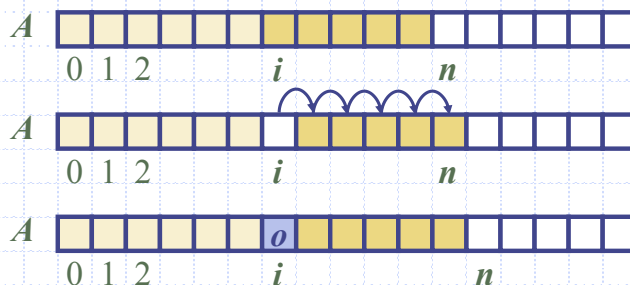
Array Lists

- 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).



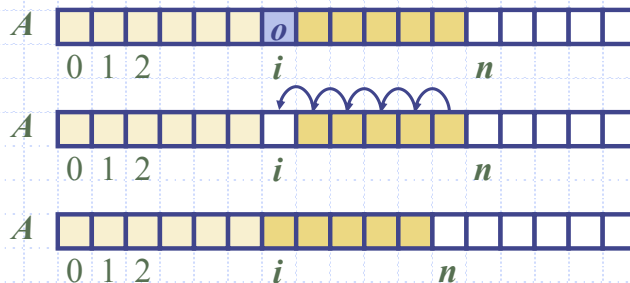
Insertion

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



not ended

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 ...

Java Implementation

```

11 // public methods
12 /** Returns the number of elements in the array list. */
13 public int size() { return size; }
14 /** Returns whether the array list is empty. */
15 public boolean isEmpty() { return size == 0; }
16 /** Returns (but does not remove) the element at index i. */
17 public E get(int i) throws IndexOutOfBoundsException {
18     checkIndex(i, size);
19     return data[i];
20 }
21 /** Replaces the element at index i with e, and returns the replaced element. */
22 public E set(int i, E e) throws IndexOutOfBoundsException {
23     checkIndex(i, size);
24     E temp = data[i];
25     data[i] = e;
26     return temp;
27 }

```

Java Implementation, 2

```

28  /** Inserts element e to be at index i, shifting all subsequent elements later. */
29  public void add(int i, E e) throws IndexOutOfBoundsException,
30         IllegalStateException {
31      checkIndex(i, size + 1);
32      if (size == data.length) // not enough capacity
33          throw new IllegalStateException("Array is full");
34      for (int k=size-1; k >= i; k--) // start by shifting rightmost
35          data[k+1] = data[k];
36      data[i] = e; // ready to place the new element
37      size++;
38  }
39  /** Removes/returns the element at index i, shifting subsequent elements earlier. */
40  public E remove(int i) throws IndexOutOfBoundsException {
41      checkIndex(i, size);
42      E temp = data[i];
43      for (int k=i; k < size-1; k++) // shift elements to fill hole
44          data[k] = data[k+1];
45      data[size-1] = null; // help garbage collection
46      size--;
47      return temp;
48  }
49  // utility method
50  /** Checks whether the given index is in the range [0, n-1]. */
51  protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
52      if (i < 0 || i >= n)
53          throw new IndexOutOfBoundsException("Illegal index: " + i);
54  }
55  }
    
```

© 2014 Goo

39

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

© 2014 Goodrich, Tamassia, Goldwasser

Lists and Iterators

40

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

$$\begin{aligned} n + c + 2c + 3c + 4c + \dots + kc &= \\ n + c(1 + 2 + 3 + \dots + k) &= \\ n + ck(k + 1)/2 & \end{aligned}$$

- 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

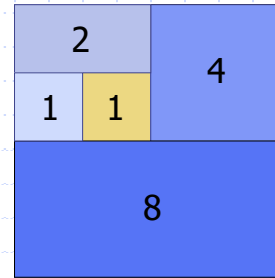
$$n + 1 + 2 + 4 + 8 + \dots + 2^k =$$

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

$$3n - 1$$

- $T(n)$ 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.

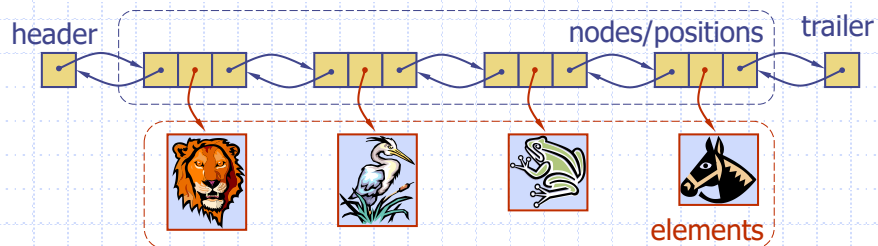
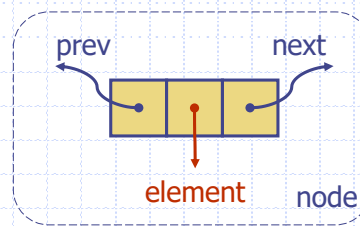
Example

□ A sequence of Positional List operations:

Method	Return Value	List Contents
addLast(8)	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)

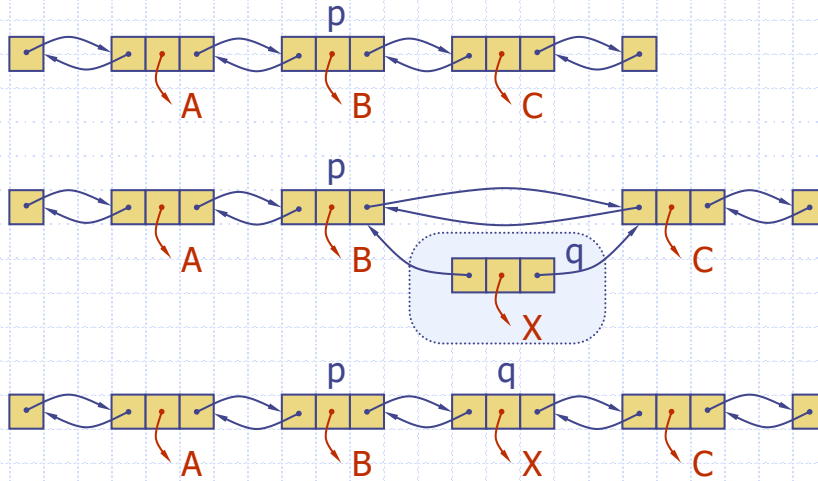
Positional List Implementation

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



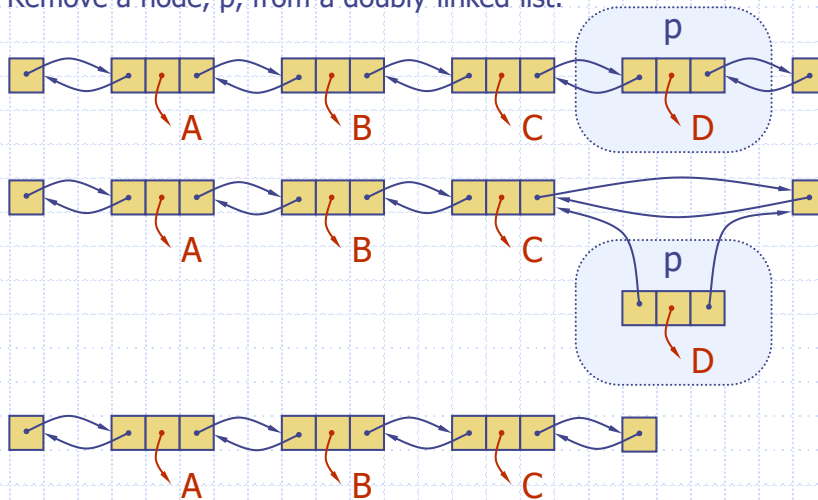
Insertion

- 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"  
}
```