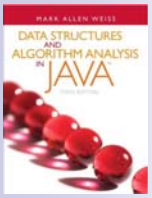


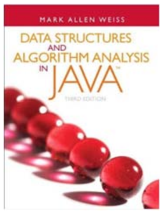
FLORIDA
INTERNATIONAL
UNIVERSITY

Lists, Stacks, and Queues (I)

Dr. Antonio L. Bajuelos

School of Computing & Information Sciences

Note: The most of the information of these slides was extracted and adapted from Weiss's book, "*Data Structures and Algorithm Analysis in Java*". They are provided for COP3530 students only. Not to be published or publicly distributed without permission by the publisher.

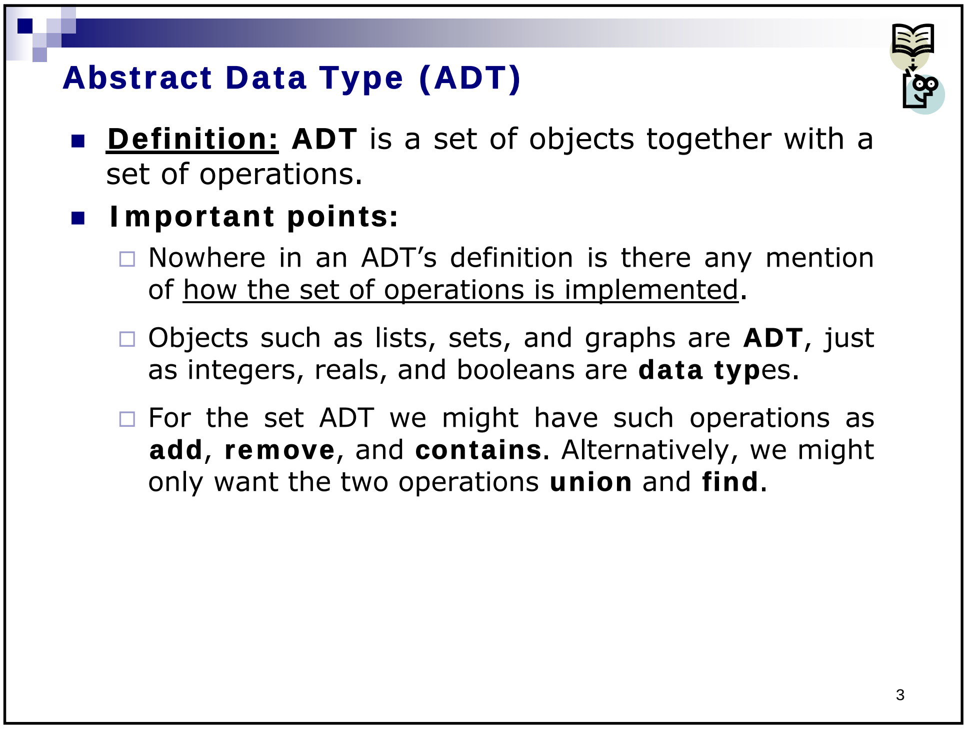


COP-3530 - Data Structures

Module #2: Lists, Stacks, and Queues (part I)

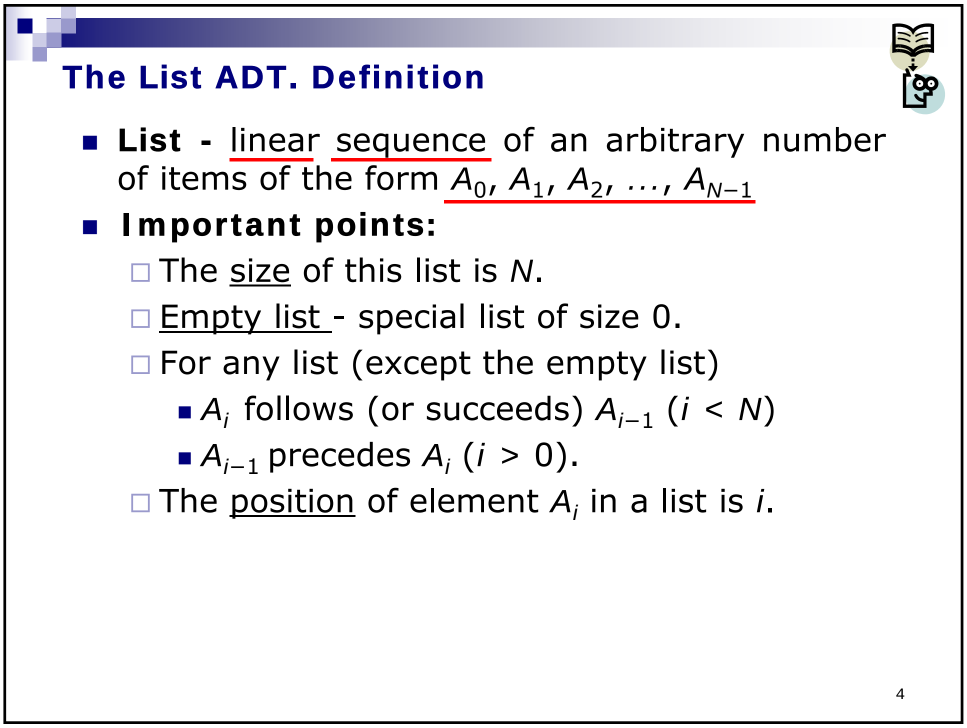
Outline:

- The List ADT.
- Array implementation of a List.
- Linked-List implementation of a List.
- Complexity analysis:
  - Array vs Linked List implementation.



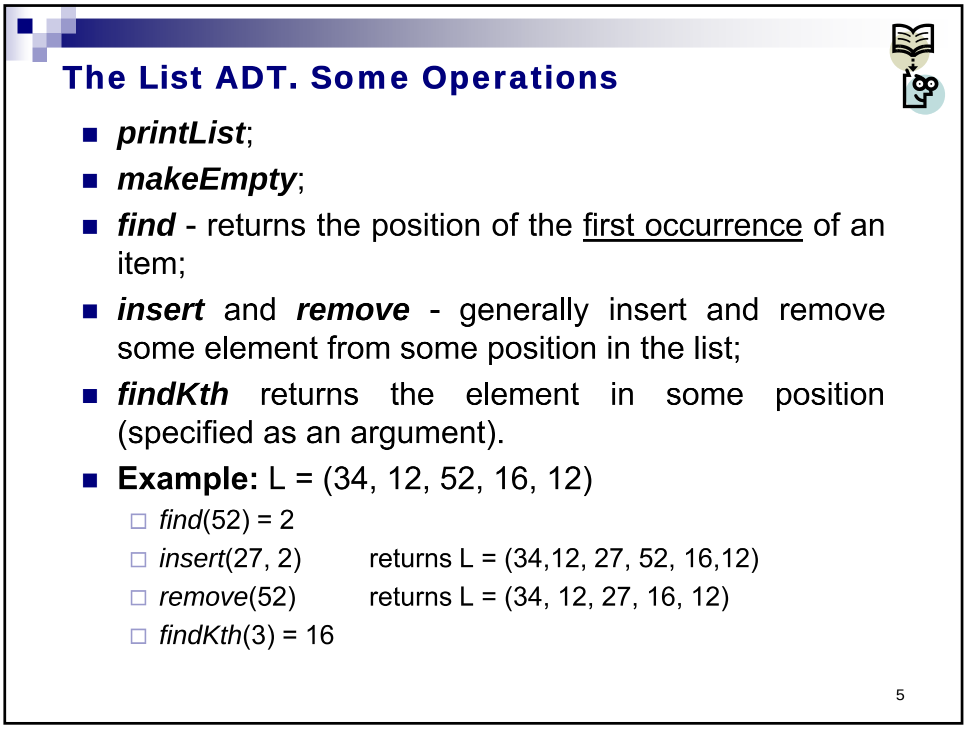
Abstract Data Type (ADT)

- **Definition:** **ADT** is a set of objects together with a set of operations.
- **Important points:**
  - □ Nowhere in an ADT's definition is there any mention of how the set of operations is implemented.
  - □ Objects such as lists, sets, and graphs are **ADT**, just as integers, reals, and booleans are **data typ**es.
  - □ For the set ADT we might have such operations as **add**, **remove**, and **contains**. Alternatively, we might only want the two operations **union** and **find**.



The List ADT. Definition

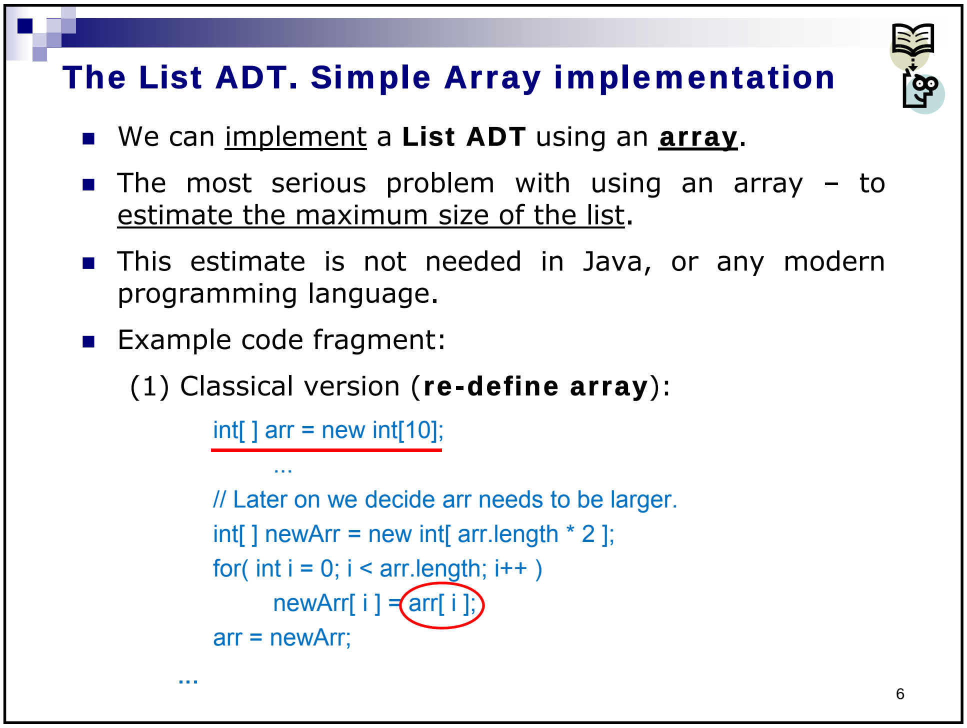
- **List** - linear sequence of an arbitrary number of items of the form $A_0, A_1, A_2, \ldots, A_{N-1}$
- **Important points:**
  - □ The size of this list is N.
  - □ Empty list - special list of size 0.
  - □ For any list (except the empty list)
    - A_i follows (or succeeds) A_{i-1} ($i < N$)
    - A_{i-1} precedes A_i ($i > 0$).
  - □ The position of element A_i in a list is i.



The List ADT. Some Operations

- ***printList***;
- ***makeEmpty***;
- ***find*** - returns the position of the first occurrence of an item;
- ***insert*** and ***remove*** - generally insert and remove some element from some position in the list;
- ***findKth*** returns the element in some position (specified as an argument).
- **Example:** L = (34, 12, 52, 16, 12)
  - □ *find*(52) = 2
  - □ *insert*(27, 2)   returns L = (34,12, 27, 52, 16,12)
  - □ *remove*(52)   returns L = (34, 12, 27, 16, 12)
  - □ *findKth*(3) = 16

5



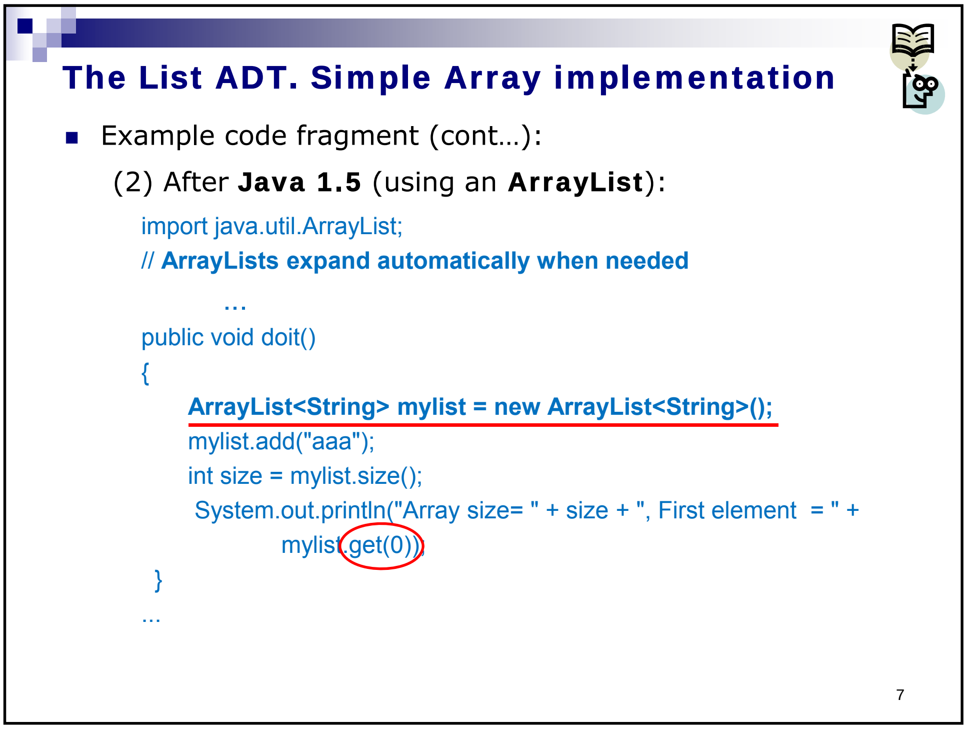
The List ADT. Simple Array implementation

- We can implement a **List ADT** using an **array**.
- The most serious problem with using an array – to estimate the maximum size of the list.
- This estimate is not needed in Java, or any modern programming language.
- Example code fragment:

(1) Classical version (**re-define array**):

```
int[ ] arr = new int[10];
        ...
// Later on we decide arr needs to be larger.
int[ ] newArr = new int[ arr.length * 2 ];
for( int i = 0; i < arr.length; i++ )
        newArr[ i ] = arr[ i ];
arr = newArr;
    ...
```

6



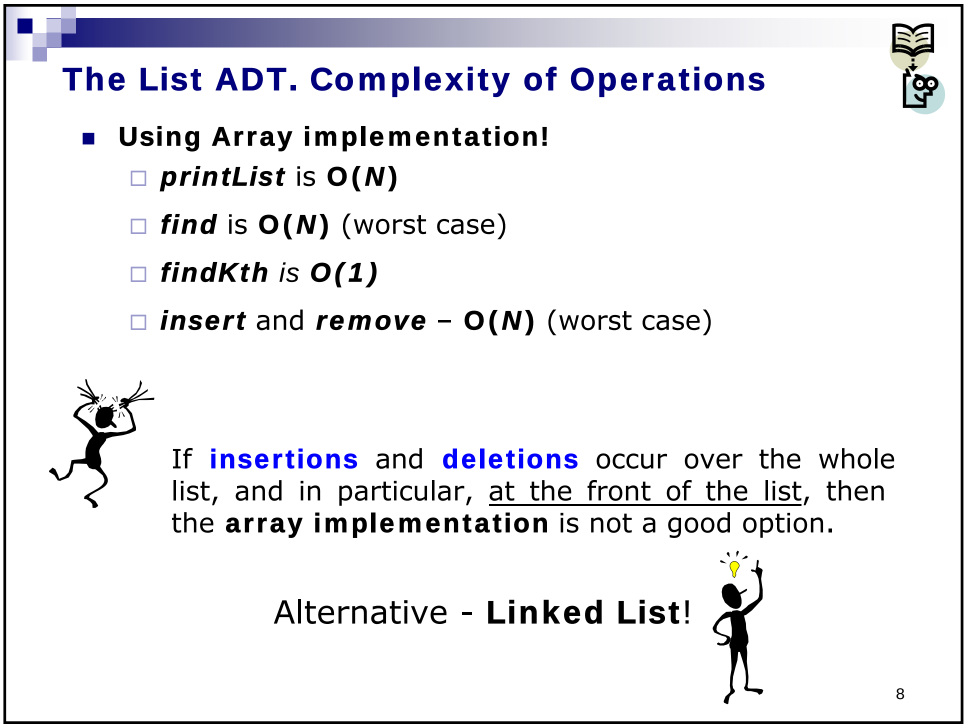
The List ADT. Simple Array implementation

- Example code fragment (cont…):

  (2) After **Java 1.5** (using an **ArrayList**):

```
import java.util.ArrayList;
// ArrayLists expand automatically when needed
        …
public void doit()
{
    ArrayList<String> mylist = new ArrayList<String>();
    mylist.add("aaa");
    int size = mylist.size();
     System.out.println("Array size= " + size + ", First element  = " +
            mylist.get(0));
 }
…
```

7



The List ADT. Complexity of Operations

- **Using Array implementation!**
  - □ ***printList*** is **O(*N*)**
  - □ ***find*** is **O(*N*)** (worst case)
  - □ ***findKth*** *is* ***O(1)***
  - □ ***insert*** and ***remove*** – **O(*N*)** (worst case)

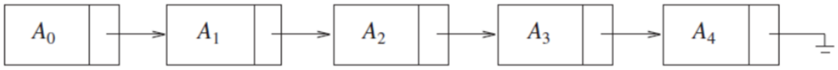
If **insertions** and **deletions** occur over the whole list, and in particular, at the front of the list, then the **array implementation** is not a good option.

Alternative - **Linked List**!

8

The Linked List. Definition

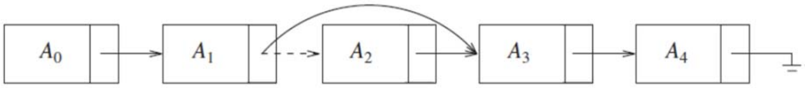
- **Linked List** - consists of a collection of **nodes**, which are not necessarily adjacent in memory.
- Each **node** contains the element (or data) and a link to a node containing its successor (next link). The last cell's next link references null.



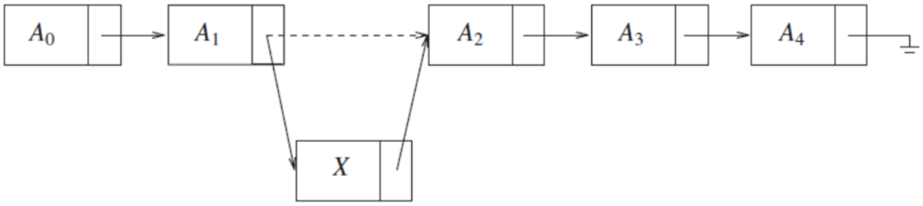
- **Important points:**
  - □ ***printList*** or ***find(x)*** is **O(*N*)** (start at the first node in the list and then traverse the list by following the next links);
  - □ ***findKth*** - is no longer quite as efficient as an array implementation;
  - □ ***findKth(i)*** takes "*O*(*i*)" time and works by traversing down the list in the obvious manner.

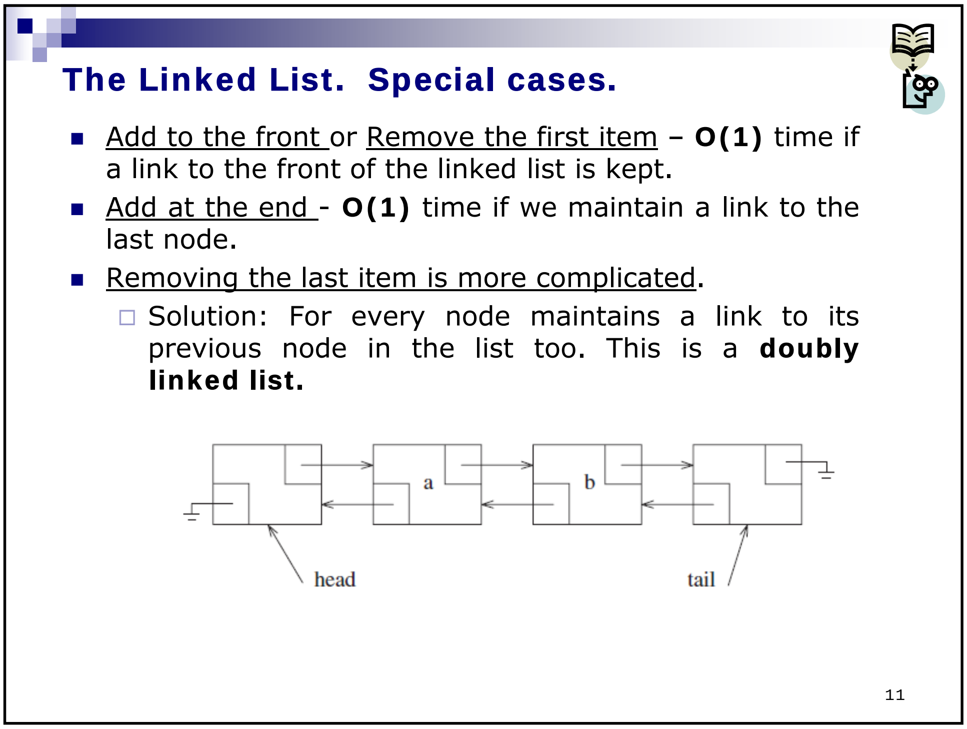
The Linked List. Remove & Insert

- The ***remove*** method can be executed in one next reference change.



- The ***insert*** method requires obtaining a new node from the system by using a new call and then executing two reference movements.

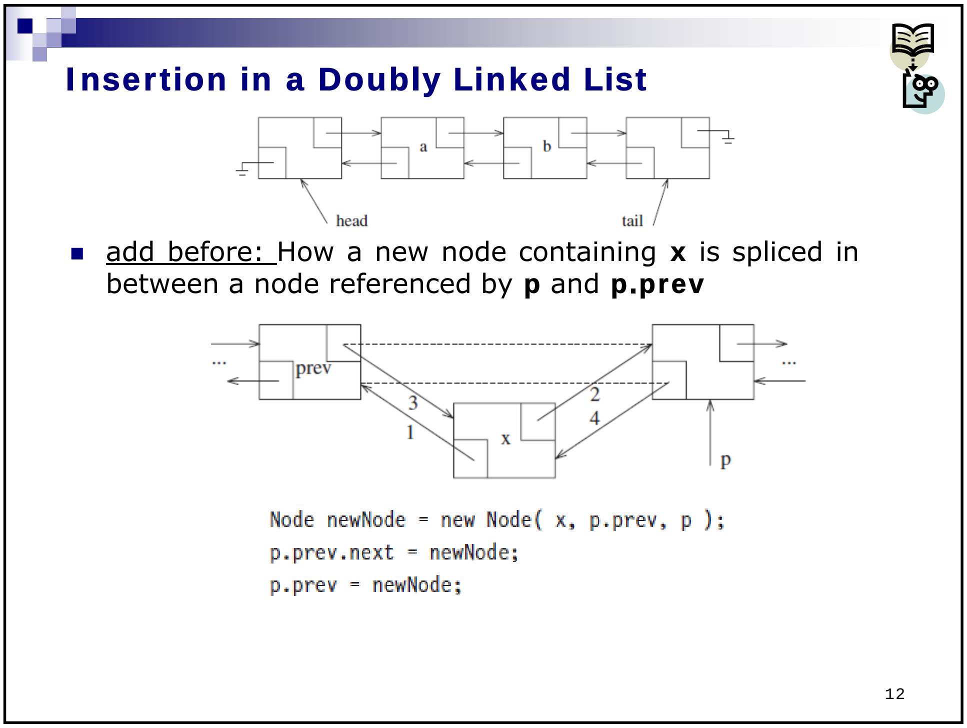




The Linked List. Special cases.

- Add to the front or Remove the first item – **O(1)** time if a link to the front of the linked list is kept.
- Add at the end - **O(1)** time if we maintain a link to the last node.
- Removing the last item is more complicated.
  - □ Solution: For every node maintains a link to its previous node in the list too. This is a **doubly linked list.**

a
b
head
tail



Insertion in a Doubly Linked List

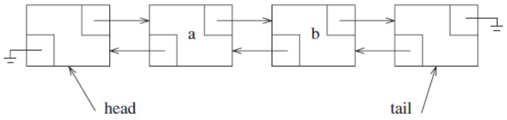
a
b
head
tail

- add before: How a new node containing **x** is spliced in between a node referenced by **p** and **p.prev**

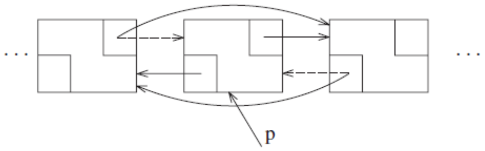
...
prev
...
3
1
x
2
4
p

```
Node newNode = new Node( x, p.prev, p );
p.prev.next = newNode;
p.prev = newNode;
```

Removing in a Doubly Linked List



- The logic of removing a node from a doubly linked list



```
p.prev.next = p.next;
p.next.prev = p.prev;
```