Data Structures and Algorithms

Arrays and Linked Lists

Outline

- Arrays
- Singly Linked Lists
- Doubly Linked Lists

Arrays

- Most commonly used data structure.
- Built-in in most programming languages.
- Unordered arrays
- Ordered arrays

Arrays

- Example:
  - Unordered: attendance tracking
  - Ordered: high scorers
- Insertion
- Deletion
- Search

Arrays

	Unordered	Ordered
Insertion	O(1)	O(n)
Deletion	O(n)	O(n)
Search	O(n)	O(logn)

Arrays

- Advantages:
  - Simple
  - Fast random (index) access
- Disadvantages:
  - Inefficient: better data structures?
    - Slow deletion: Linked Lists
  - Fixed size (see Vector)

Abtract Data Type

- Variables have a **type**
- Type of a variable defines its possible values
- Defines the operations that can be performed on it
- Type can be defined independent of
  - the actual data structure
  - the programming language
  - the computer
- Hence called **ABSTRACT DATA TYPE**

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- Just one method:
  - object element(): returns the element stored at the position

List ADT

- The List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), isEmpty()

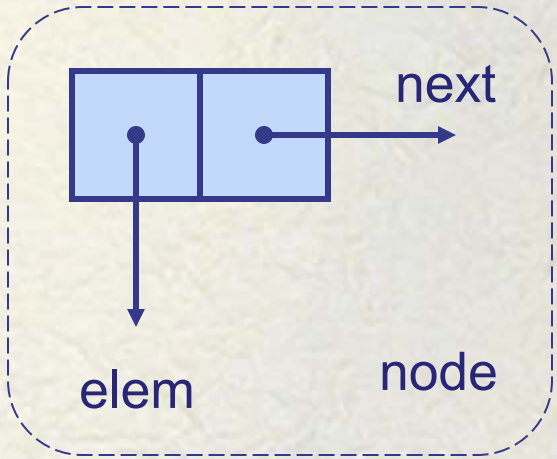
Accessor methods:

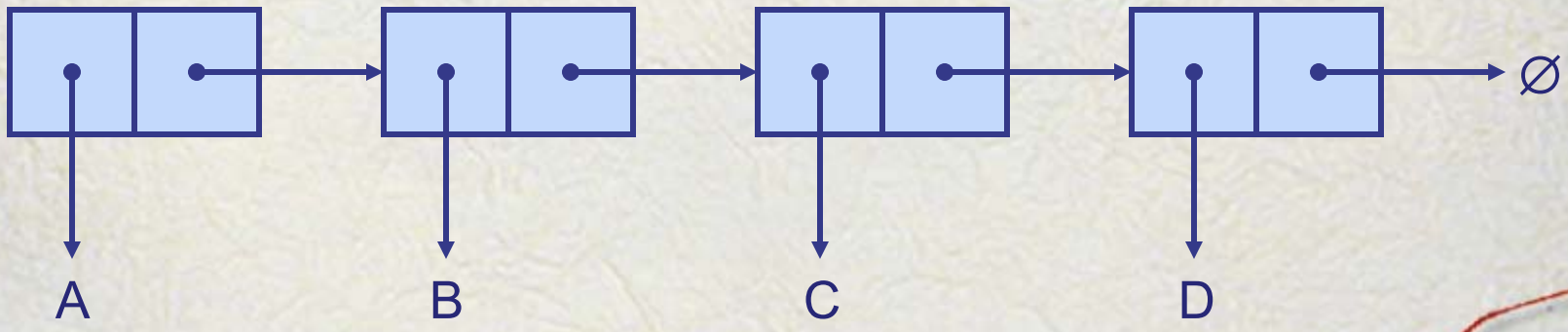
- first(), last()
- prev(p), next(p)

- Update methods:
  - replace(p, e)
  - insertBefore(p, e), insertAfter(p, e),
  - insertFirst(e), insertLast(e)
  - remove(p)

Singly Linked List

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





The Node Class for List Nodes

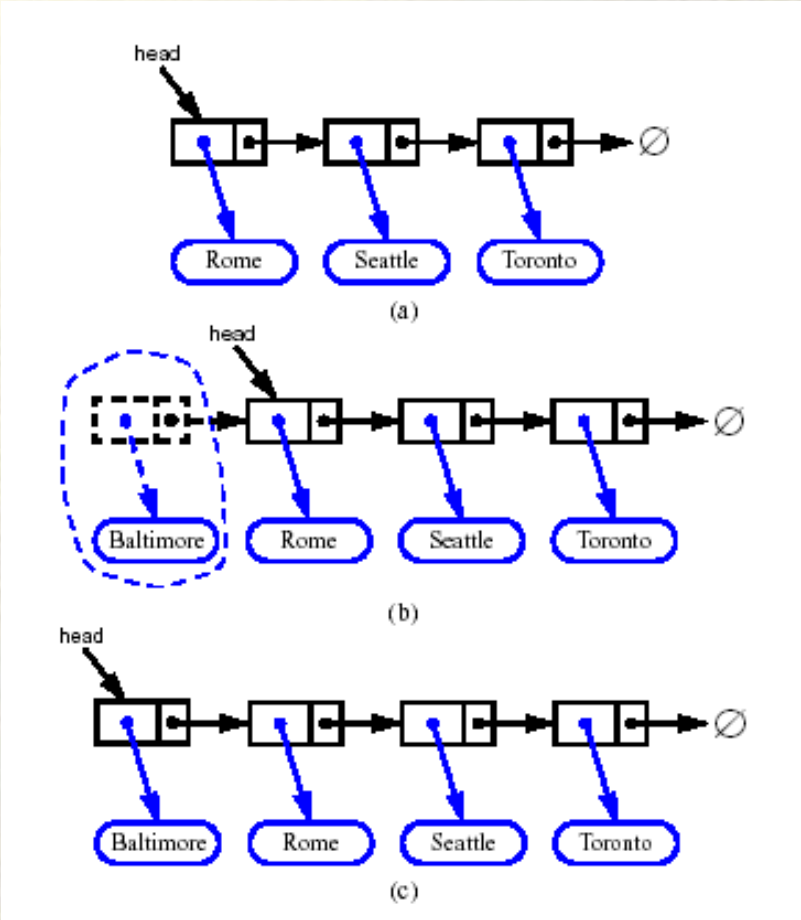
```
public class     Node              {
   // Instance variables:
   private  Object element;
   private  Node next;
   /** Creates a node with null references to its element and next node. */
   public  Node()                    {
     this(null,  null);
  }
   /** Creates a node with the given element and next node. */
   public  Node(Object e,  Node n)  {
       element  =  e;
       next  =  n;
  }
   // Accessor methods:
   public  Object getElement()  {
     return  element;
  }
   public  Node getNext()  {
     return  next;
  }
   // Modifier methods:
   public void  setElement(Object newElem)  {
       element  =  newElem;
  }
   public void  setNext(Node newNext)  {
       next  =  newNext;
  }
}
```

Singly Linked List Class

```
/** Singly linked list .*/
public class SLinkedList {
  protected Node head;// head node of the list
  protected long size;// number of nodes in the list
  /** Default constructor that creates an empty list */
  public SLinkedList() {
    head = null;
    size = 0;
  }
  // ... update and search methods would go here ...
}
```

Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node

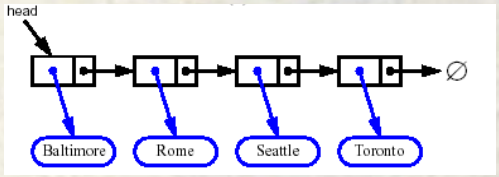


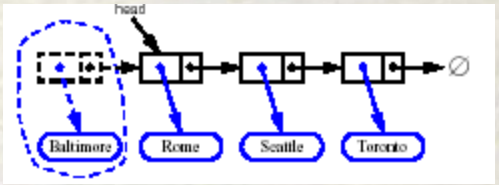
Algorithm addFirst

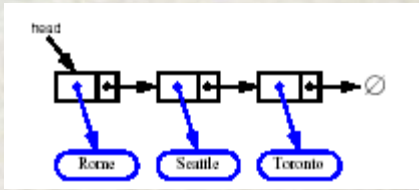
```
Algorithm addFirst(v):
    v.setNext(head)          {make v point to the old head node}
    head ← v                 {make variable head point to new node}
    size ← size + 1          {increment the node count}
```

Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node





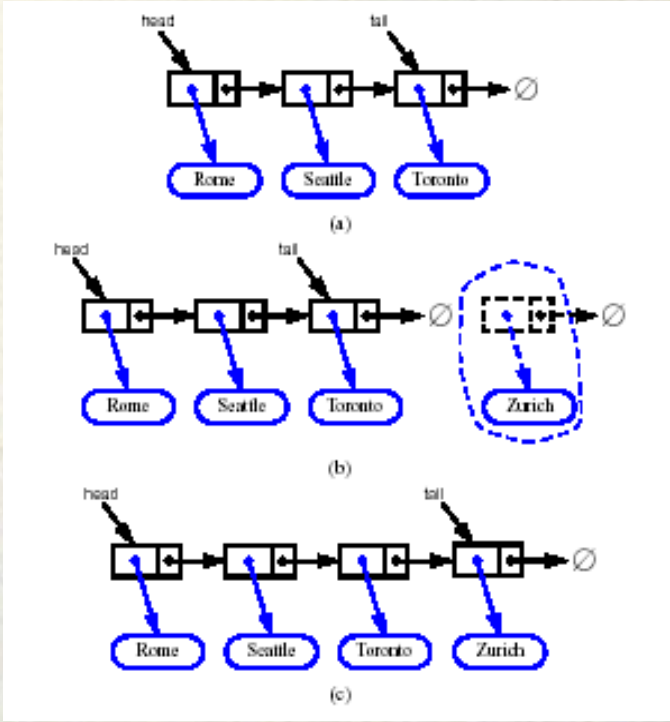


Algorithm removeFirst

```
Algorithm removeFirst():
    if head = null then
        Indicate an error: the list is empty.
    t ← head
    head ← head.getNext()          {make head point to next node (or null)}
    t.setNext(null)                {null out the next pointer of the removed node}
    size ← size − 1                {decrement the node count}
```

Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

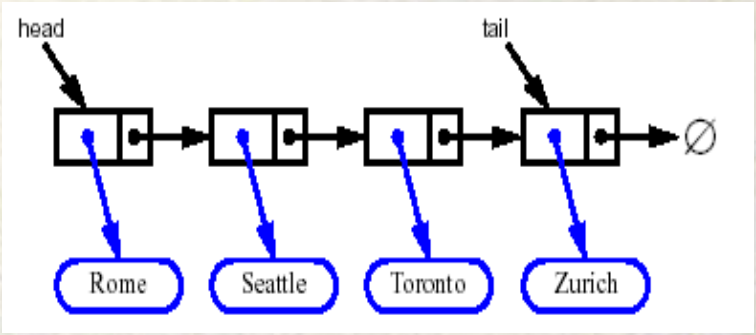


Algorithm addLast

```
Algorithm addLast(v):
    v.setNext(null)          {make new node v point to null object}
    tail.setNext(v)          {make old tail node point to new node}
    tail ← v                 {make variable tail point to new node.}
    size ← size + 1          {increment the node count}
```

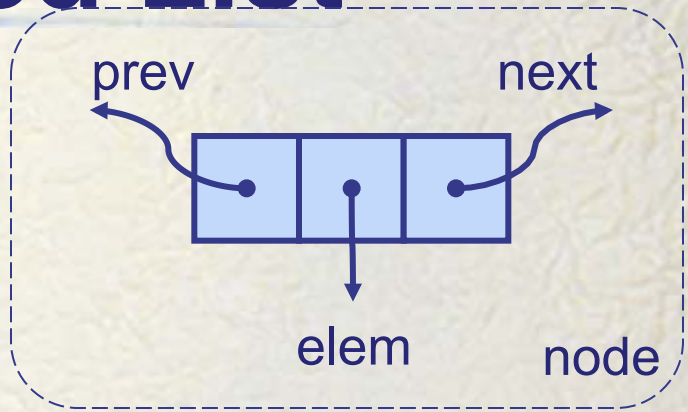
Removing at the Tail

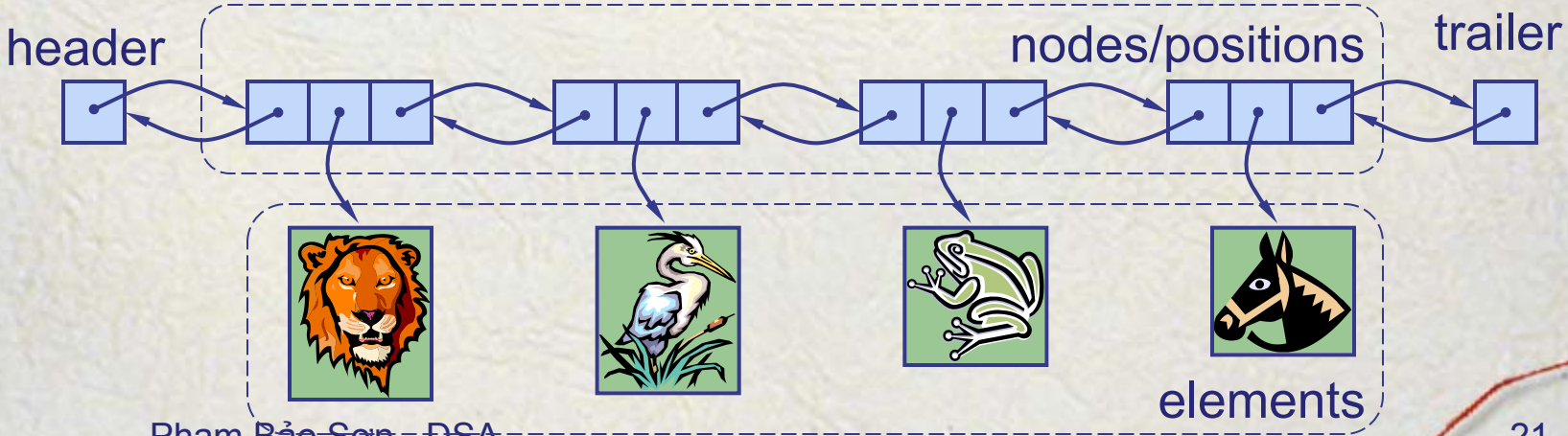
- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



Doubly Linked List

- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Node Class

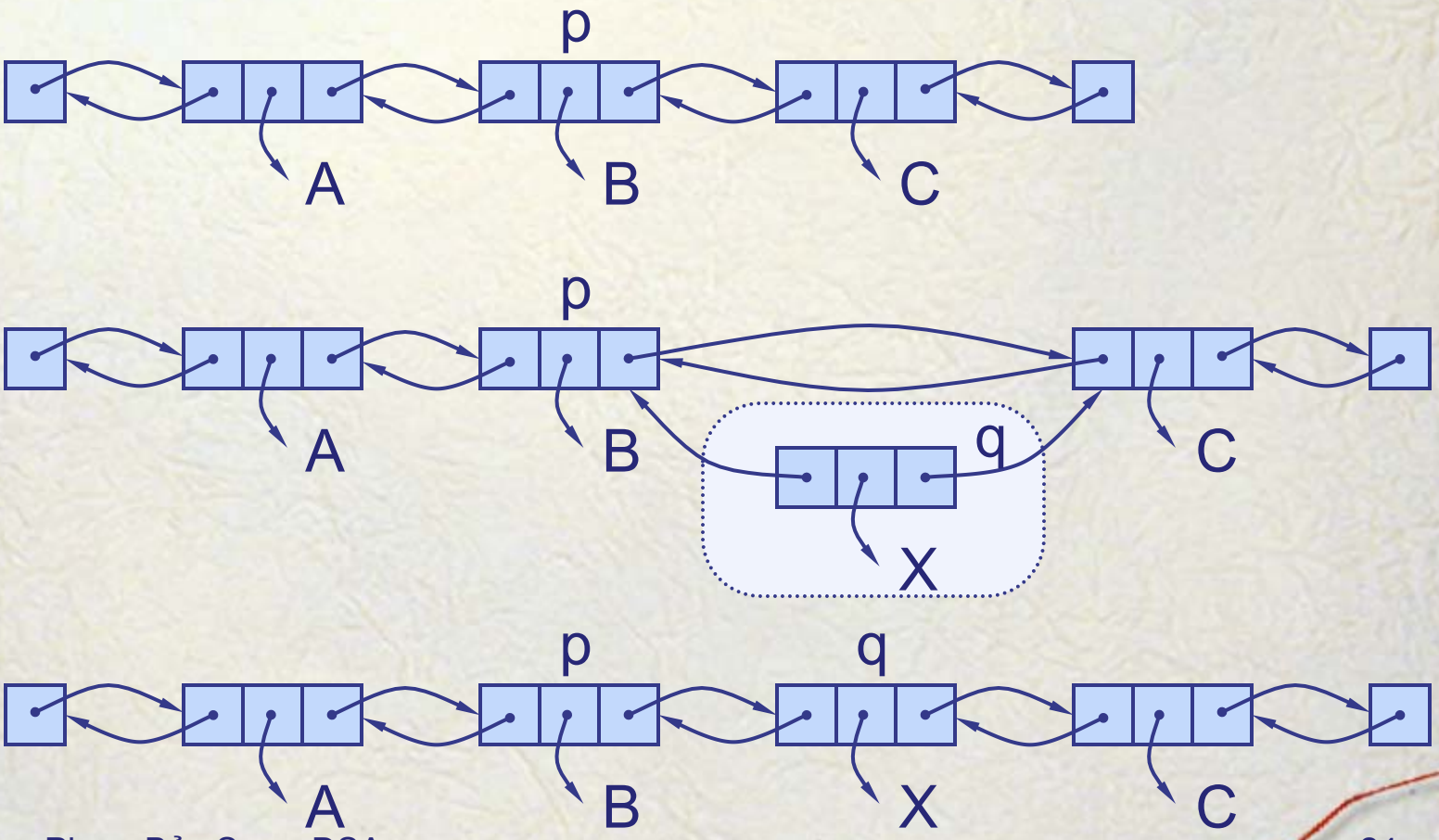
```
/** Node of a doubly linked list of strings */
public class DNode {
  protected String element;    // String element stored by a node
  protected DNode next, prev; // Pointers to next and previous nodes
  /** Constructor that creates a node with given fields */
  public DNode(String e, DNode p, DNode n) {
    element = e;
    prev = p;
    next = n;
  }
```

Dlink Class

```
/** Doubly linked list with nodes of type DNode storing strings. */
public class DList {
  protected int size;                    // number of elements
  protected DNode header, trailer;       // sentinels
  /** Constructor that creates an empty list */
  public DList() {
    size = 0;
    header = new DNode(null, null, null);      // create header
    trailer = new DNode(null, header, null);   // create trailer
    header.setNext(trailer);       // make header and trailer point to each other
  }
```

Insertion

- We visualize operation insertAfter(p, X), which returns position q



Insertion Algorithm

Algorithm insertAfter(*p*,*e*):

Create a new node *v*

v.setElement(*e*)

v.setPrev(*p*) {link *v* to its predecessor}

v.setNext(*p*.getNext()) {link *v* to its successor}

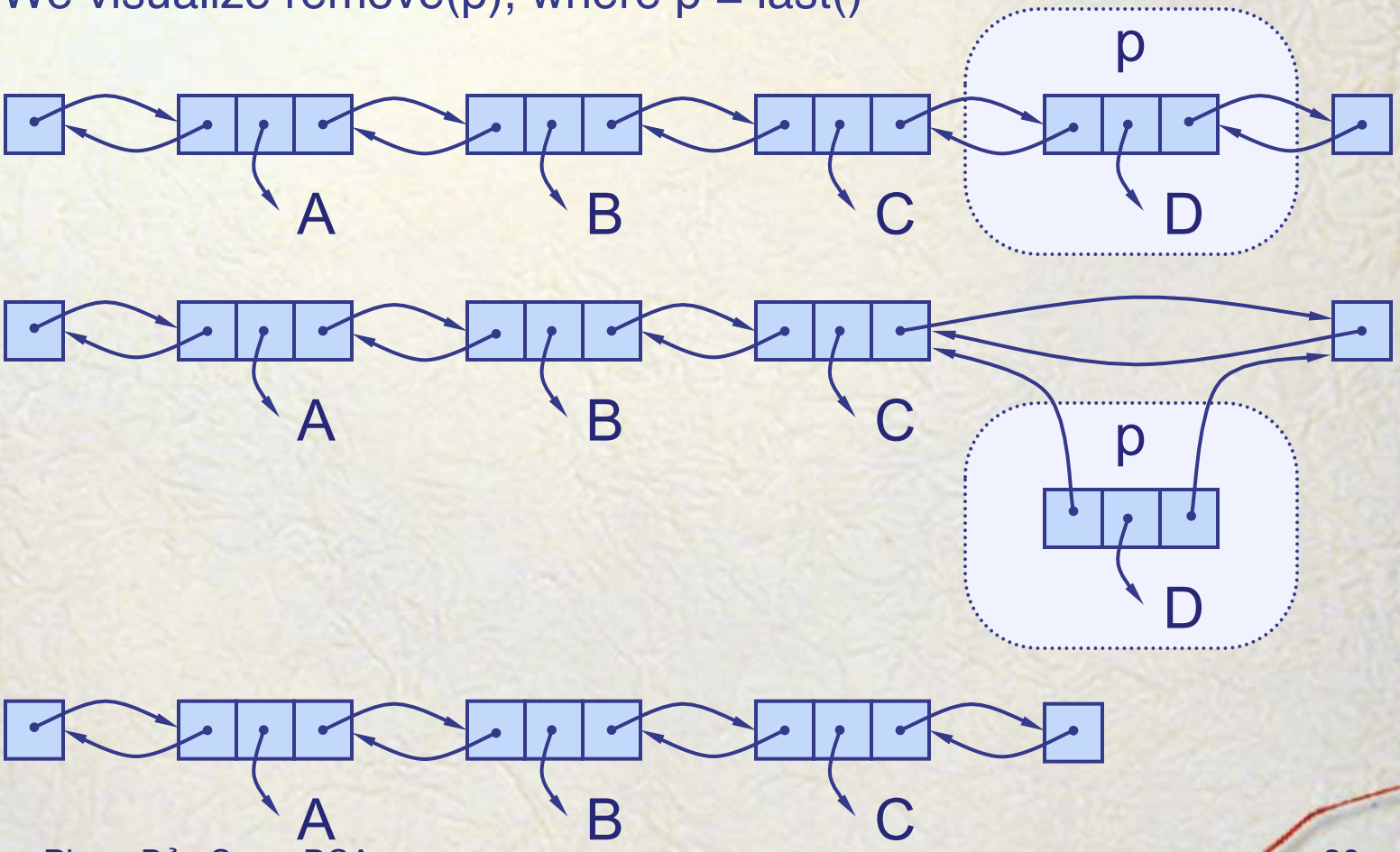
(*p*.getNext()).setPrev(*v*) {link *p*'s old successor to *v*}

p.setNext(*v*) {link *p* to its new successor, *v*}

return *v* {the position for the element *e*}

Deletion

- We visualize remove(p), where p = last()



Deletion Algorithm

Algorithm remove(*p*):
    t = *p*.element {a temporary variable to hold the return value}
    (*p*.getPrev()).setNext(*p*.getNext()) {linking out *p*}
    (*p*.getNext()).setPrev(*p*.getPrev())
    p.setPrev(**null**) {invalidating the position *p*}
    p.setNext(**null**)
    return *t*

Performance

- In the implementation of the List ADT by means of a doubly linked list
  - 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
  - Operation element() of the Position ADT runs in $O(1)$ time