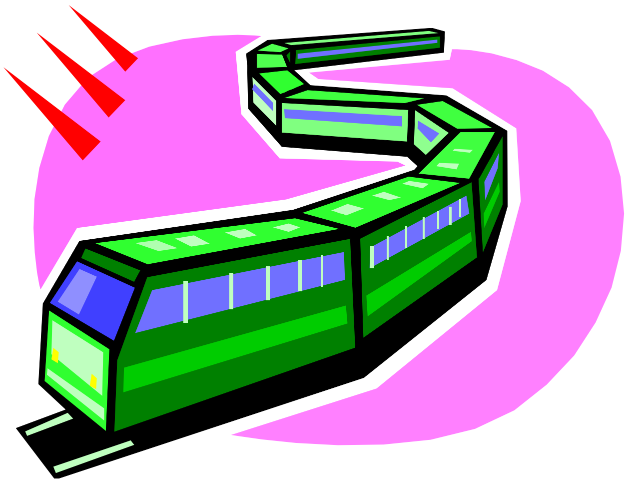
# Arrays and Linked Lists

Data Structures and Algorithms



Acknowledgement:
These slides are adapted from slides provided with *Data Structures and Algorithms in C++*
Goodrich, Tamassia and Mount (Wiley, 2004)

# Outline

- Arrays
- Singly Linked Lists
- Doubly Linked Lists

# Arrays

- Built-in in most programming languages
- Two kinds (programmer responsibility):
  - Unordered: attendance tracking
  - Ordered: high scorers
- Operations:
  - Insertion
  - Deletion
  - Search

# Arrays

| Operation | Unordered | Ordered |
|---|---|---|
| Insertion | O(1) | O(n) |
| Deletion | O(n) | O(n) |
| Search | O(n) | O(logn) |

Pluses & minuses

+ Fast element access; simple

-- Impossible to resize; slow deletion

- Many applications require resizing!
- Required size not always immediately available.

# 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

# Abstract Data Types - Example

An ADT modeling a simple stock trading system

- Data stored: buy/sell orders
- Operations on the data:
  - 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 * getElement(): 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:

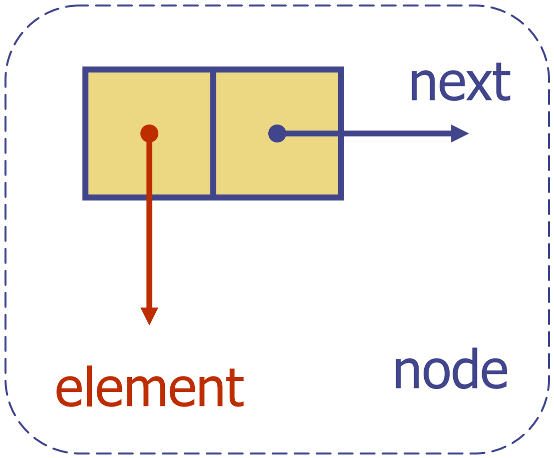
- first(), last()
- prev(p), next(p)

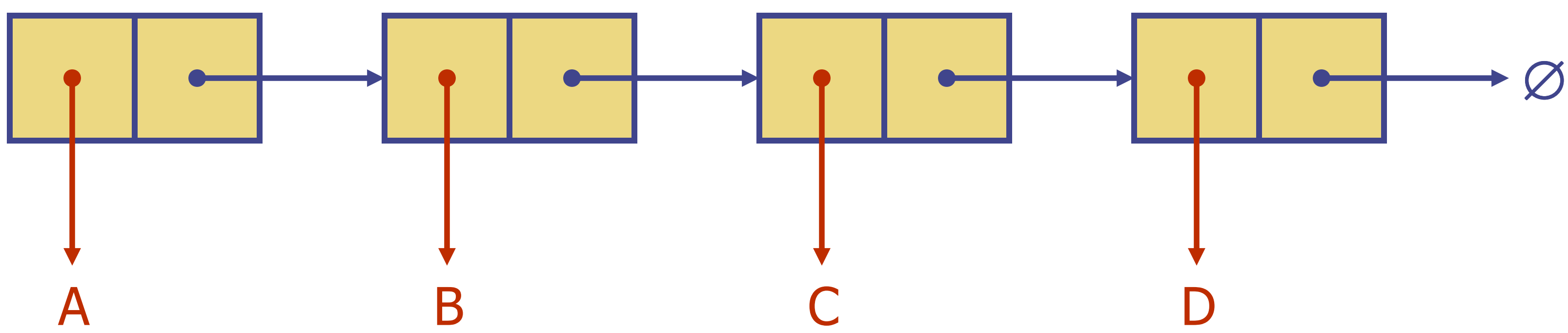
Update methods:

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

# Singly Linked Lists

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

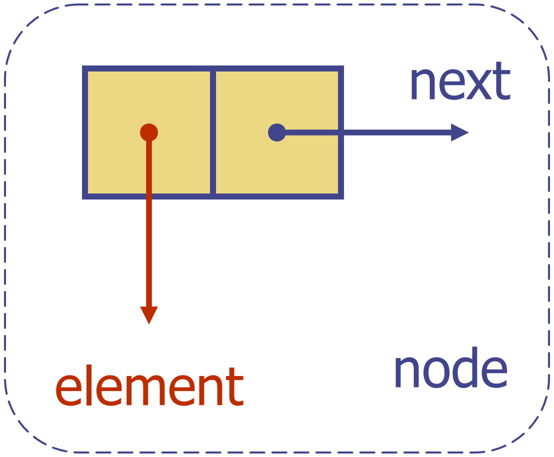


# Node struct

```
struct Node {
   // Instance variables
   Object* element;
   Node* next;

   // Initialize a node
   Node() {
      this(null,  null);
   }
   Node(Object* e, Node* n)  {
      element = e;
      next = n;
   }
}
```

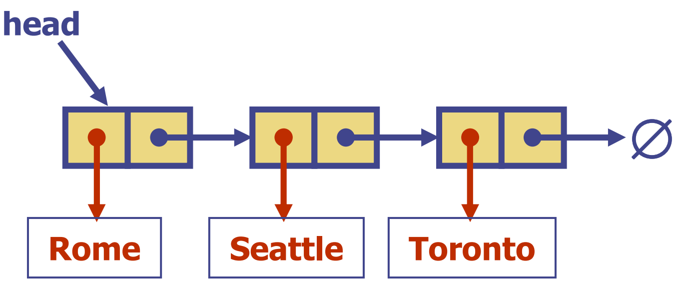


# Singly linked list

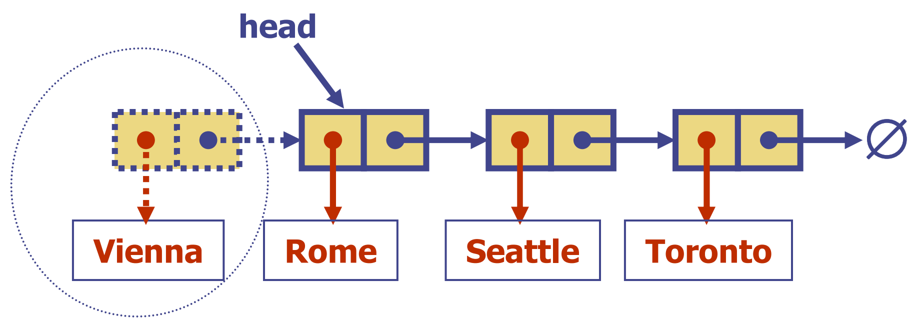
```
struct SLinkedList {
    Node* head;     // head node of the list
    long size;             // number of nodes in the list

    /* Default constructor that creates an empty list */
    SLinkedList() {
        head = null;
        size = 0;
    }
    // ... update and search methods would go here ...
    boolean isEmpty() {return head ==null; }
    void insertAfter(Node * node, Object* element) {...}
    ...
};
```

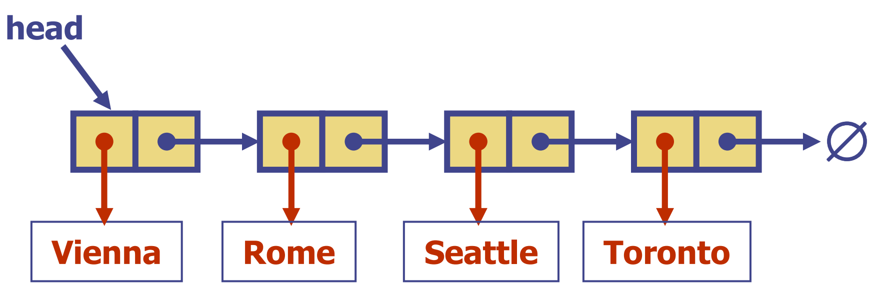
# Inserting at the Head




1. Allocate a new node
2. Insert new element
3. Make new node point to old head




4. Update head to point to new node

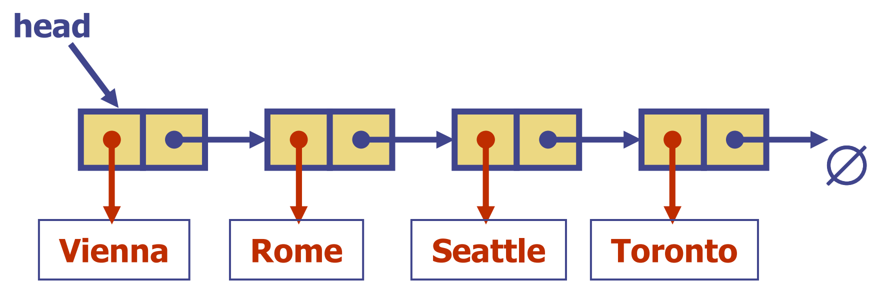


# Algorithm addFirst

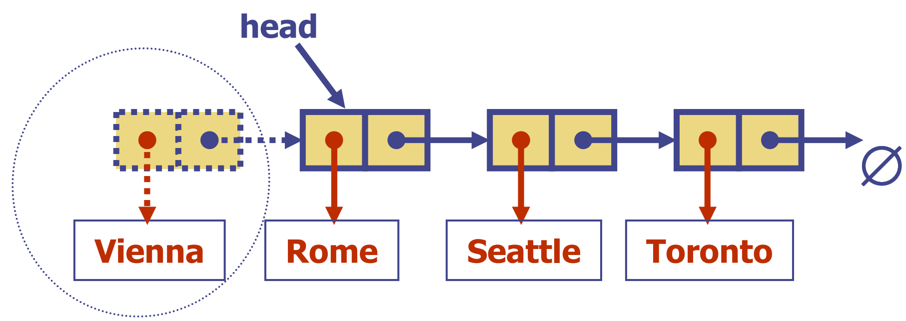
**Algorithm** *addFirst*($v$)
  **Input** node $v$ to be added to the beginning of the list
  **Output**

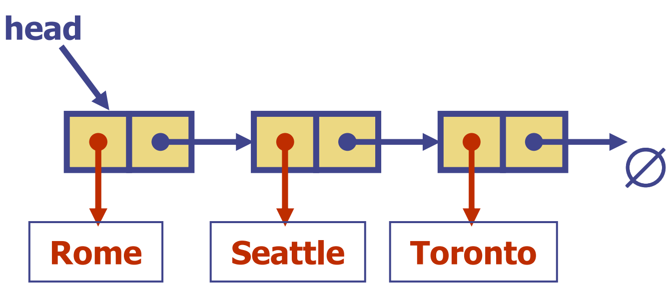
| | |
|---|---|
| $v.setNext\,(head)$ | {make v point to the old head node} |
| $head \leftarrow v$ | {make variable head point to new node} |
| $size \leftarrow size + 1$ | {increment the node count} |

# Removing at the Head




1. Update head to point to next node in the list
2. Disallocate the former first node
   - the garbage collector to reclaim (Java), or
   - the programmer does the job (C/C++)



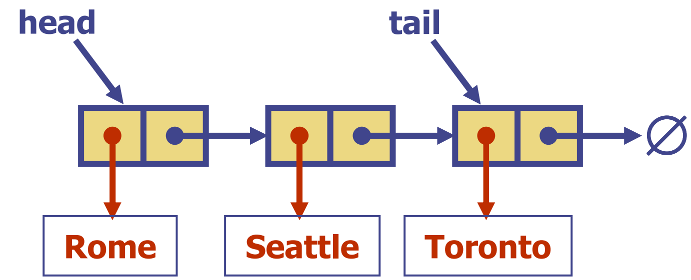



# Algorithm removeFirst

**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}
Disallocate node *t*   {null out t's next pointer or free t's memory}
*size* ← *size* - 1   {decrement the node count}

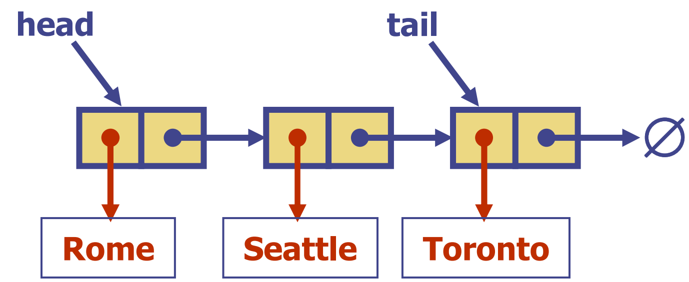
# Singly linked list with 'tail' sentinel



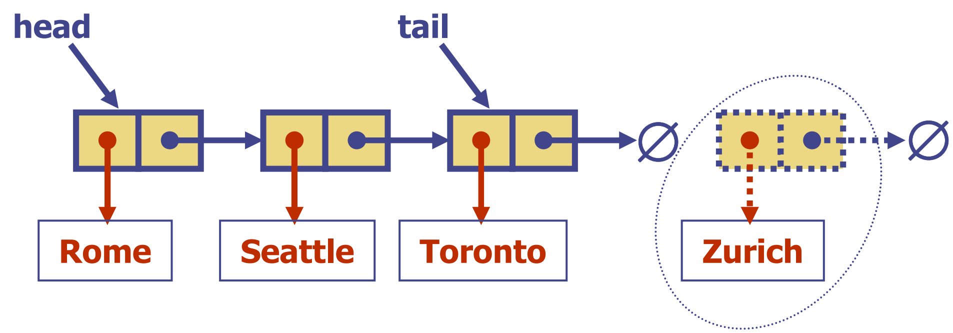

```
struct SLinkedListWithTail {
    Node* head; // head node
    Node* tail; // tail node of the list

     /* Default constructor that creates an empty list */
    SLinkedListWithTail() {
        head = null;
        tail = null;
    }
    // ... update and search methods would go here ...
}
```

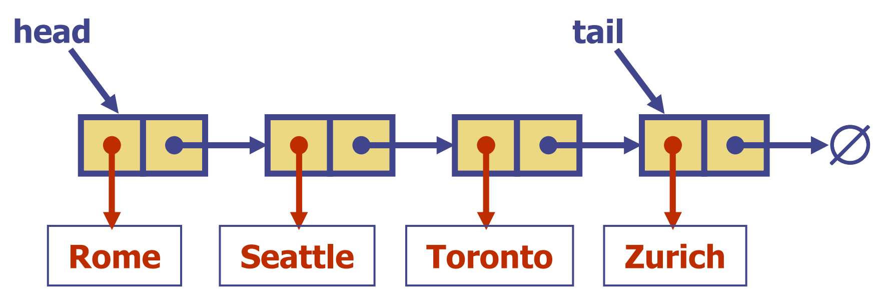
# Inserting at the Tail




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



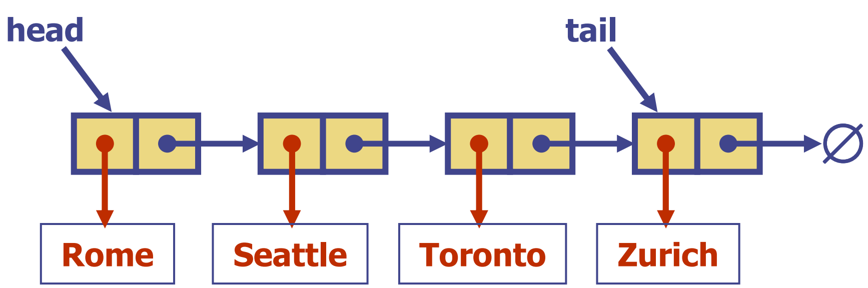
# Algorithm addLast

**Algorithm** *addLast*(*v*)
**Input** node *v* to be added to the end of the list
**Output**

*v.setNext* (NULL) {make new node v point to null object}
*tail.setNext(v)* {make old tail node point to new node}
*tail* ← *size;* {make variable tail node point to new node}
*size* ← *size* + 1 {increment the node count}

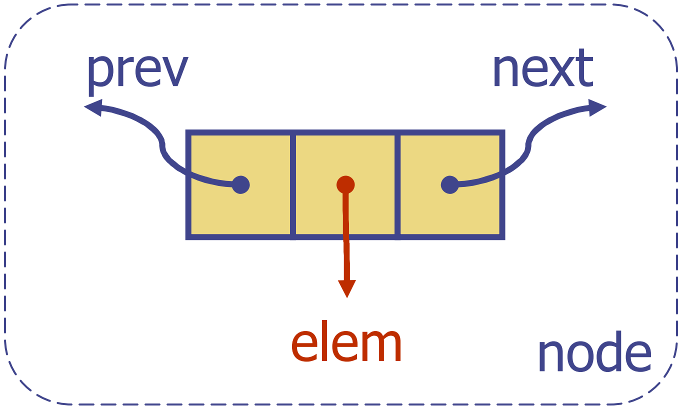
# Removing at the Tail

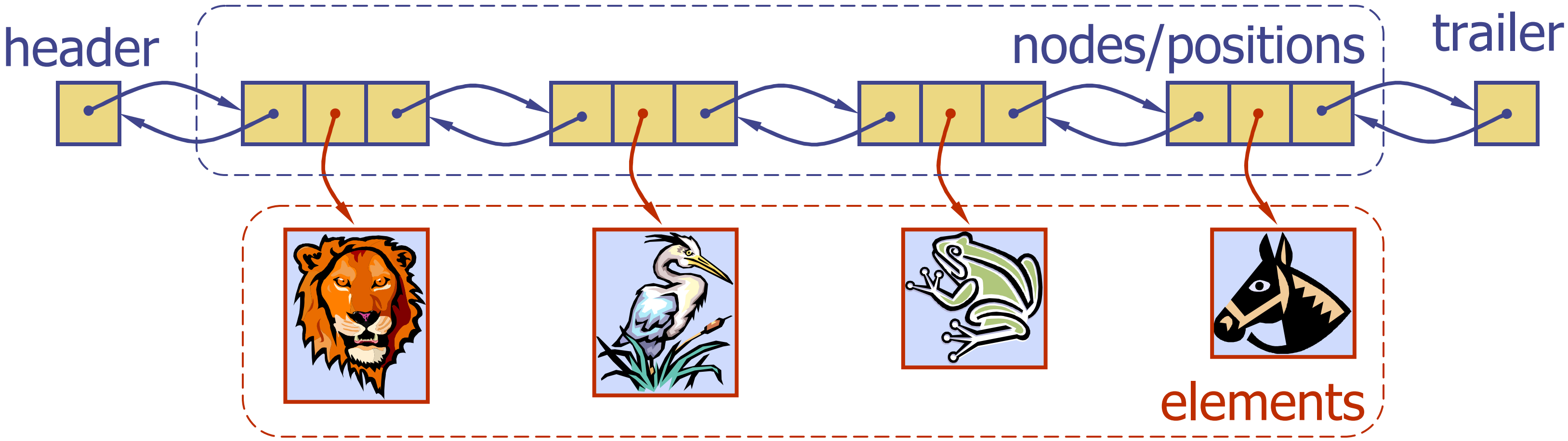
- Removing at the tail of a singly linked list cannot be efficient!
- There is no constant-time way to update the tail to point to the previous node



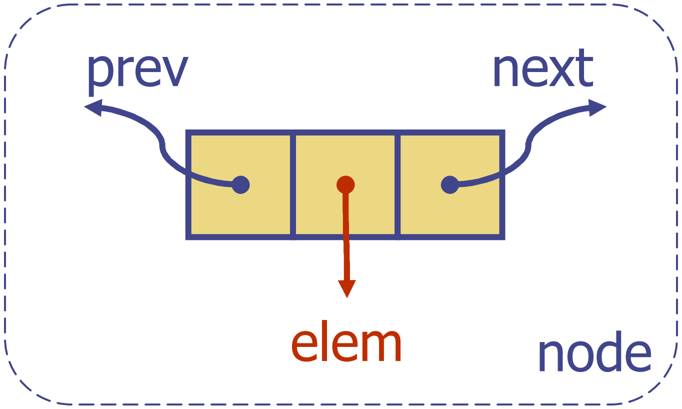
# Doubly Linked List

- A doubly linked list is often more convenient!
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes



# Node struct




```
/* Node of a doubly linked list of strings */
struct DNode {
   string* element;
   DNode *next, *prev; // Pointers to next and previous node

   /* Initialize a node. */
   DNode(string* e, DNode* p, DNode *n)  {
      element = e;
      prev = p;
      next = n;
   }
   string* getElement() { return element; }
};
```

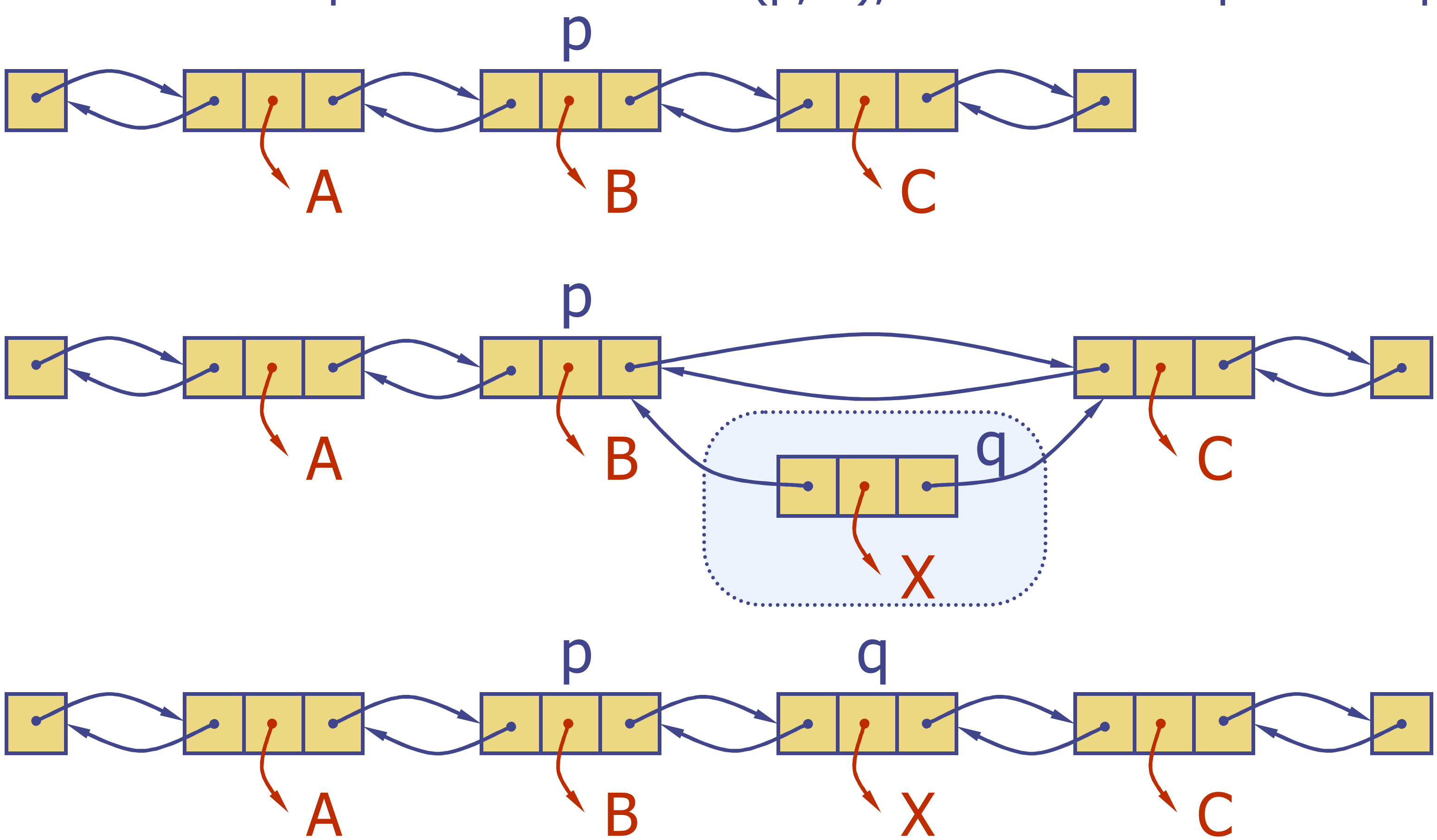
# Class for doubly linked list

```
struct DList {
    DNode* header, *trailer;     // sentinels
    long size;              // number of nodes in the list

    /* Default constructor that creates an empty list */
    DList() {
        header = new DNode(NULL, NULL, NULL);
        trailer = new DNode(NULL, header, NULL);
        header->next = trailer;
        size = 0;
    }
    // ... update and search methods would go here ...
};
```

# Insertion

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



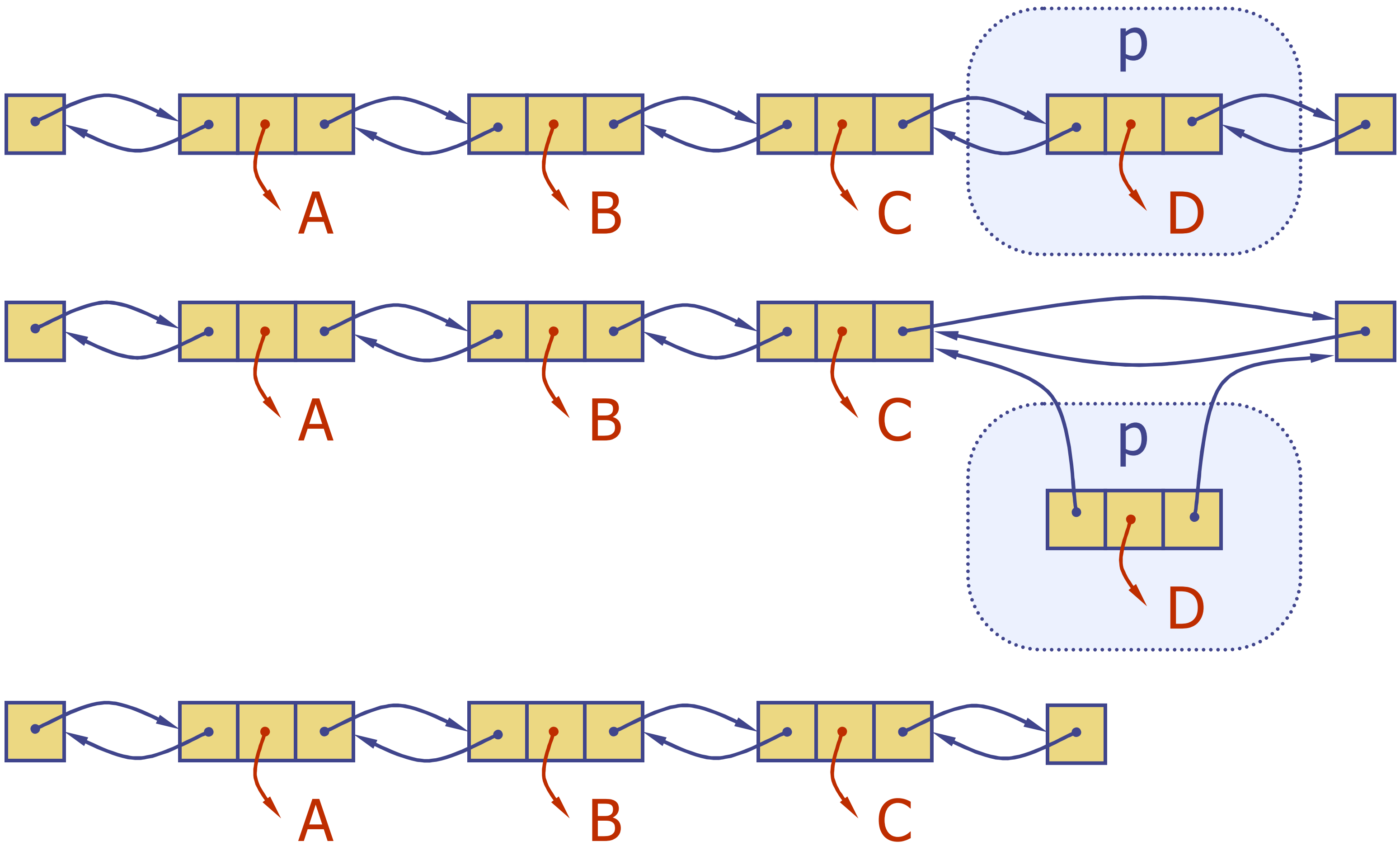
# Insertion Algorithm

**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***  {temporary variable to hold the return value}

   (***p.getPrev***()).***setNext***(***p.getNext***())  {linking out *p*}

   (***p.getNext***()).***setPrev***(***p.getPrev***())

   Disallocate node ***p***  {invalidating the position *p*}

   **return** ***t***

# Worst-case running time

- In a doubly linked list

  + insertion at head or tail is in O(1)

  + deletion at either end is on O(1)

  -- element access is still in O(n)