COMP2054-ADE

ADE Lec06
Linked Lists

Lecturer: Andrew Parkes

http://www.cs.nott.ac.uk/~pszajp/

Note

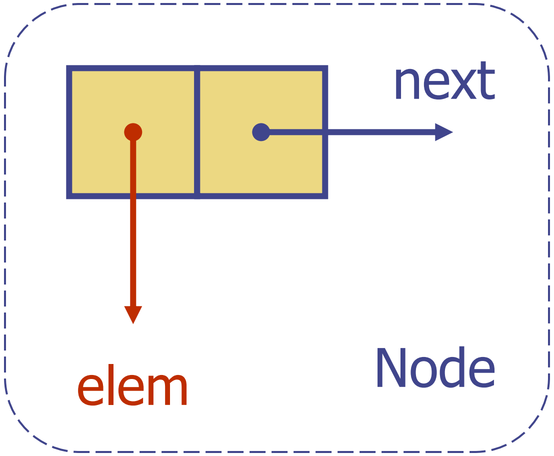
- This short set of slides is just a quick introduction to linked lists, and not everything you need to know.
- Hopefully, most of it is revision from 1st year (or before).

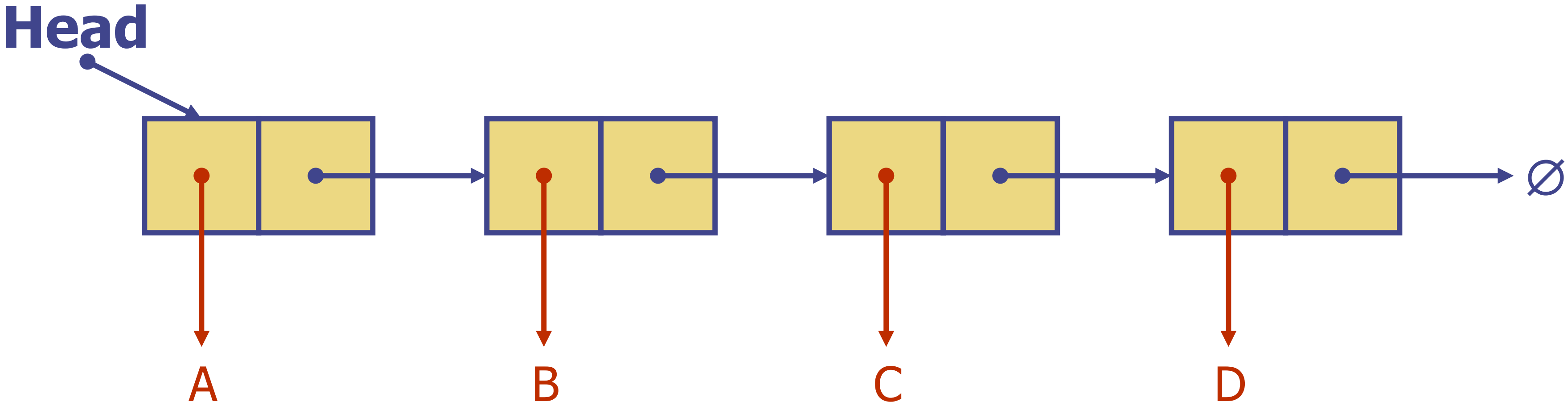
Relevance

- It is generally assumed that you will have done singly and doubly linked lists
    - Please revise them, if not then these slides just cover the basic ideas
- Our goal in this module is to be more careful about their efficiency, that is, to observe the complexity, O(1) versus O(n), of the various standard operations
    - Where 'n' is the length of the list
- Also, they are relevant to "simple sorting" algorithms, and also to "stacks" and "queues"

Recap: Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element e.g.
    - Reference to an Object
    - A primitive date type (int,…)
  - "link": a reference (pointer) to the next node





A Node Class for List Nodes

The relevant code usually looks like:

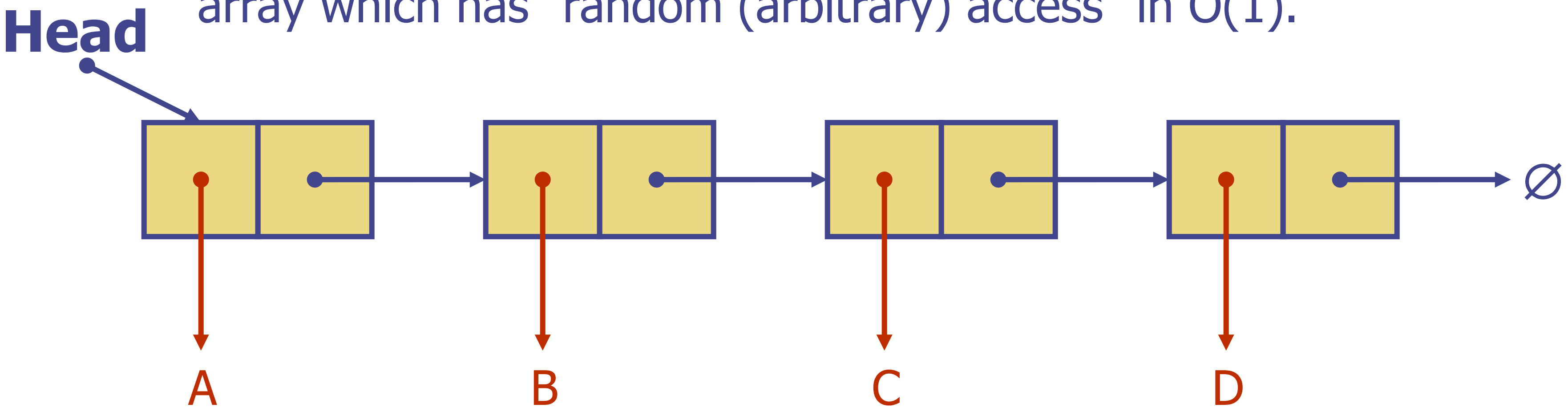
```
public class Node  {
   // Instance variables:
   private  Object element;
   private  Node next; // reference (“pointer”) to a ‘Node’

   /** Creates a node with the given element and next
   node. */
   public  Node(Object e,  Node n)  {
       element  =  e;
       next  =  n;
  }
```

The Node Class for List Nodes

```
  // 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;
  }
} // end of class Node
```

Usage?

- In a simple linked list, all the data is accessible from the head by just "walking along the list"
  - Hence, it is clear (hopefully) that all "standard" operations (insert/delete etc) are implementable
- The key question is what operations are implementable **efficiently**!?
  - Need O(.) of operations in terms of the length n of the list
    - Note: Typically the overall length of a list (or other data structure) is stored in an auxiliary element to allow fast access as it is generally used a lot.
  - Note: There is no direct access to the middle of a list - unlike an array which has "random (arbitrary) access" in O(1).

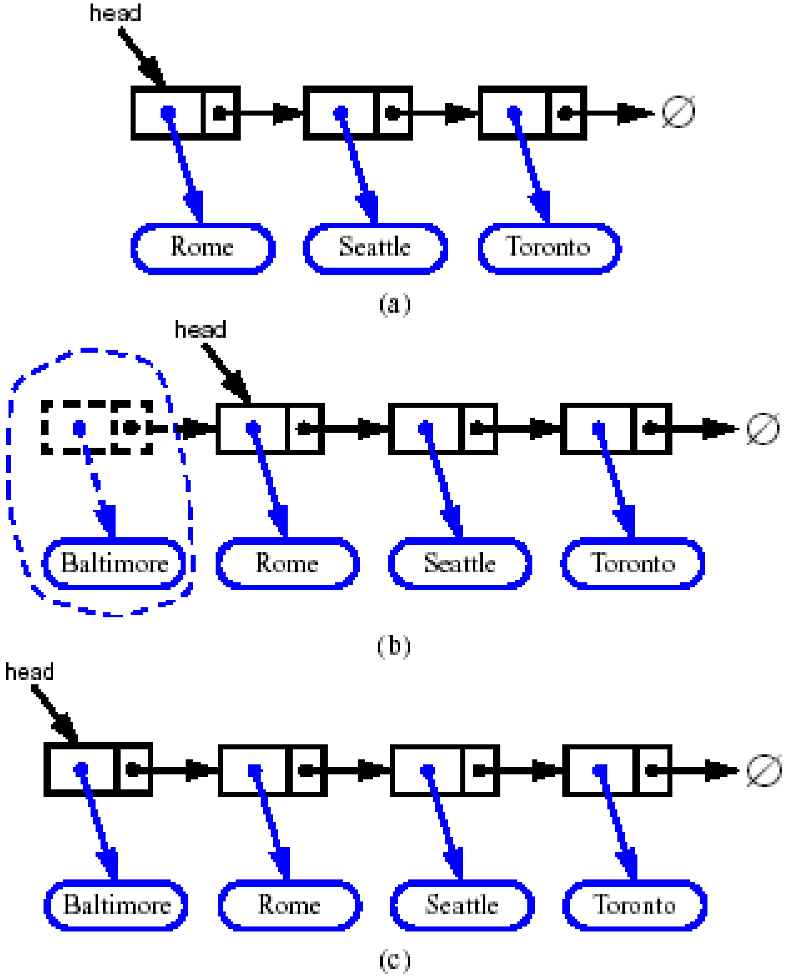


Inserting at the Head

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

What is the complexity (with n elements in list)?

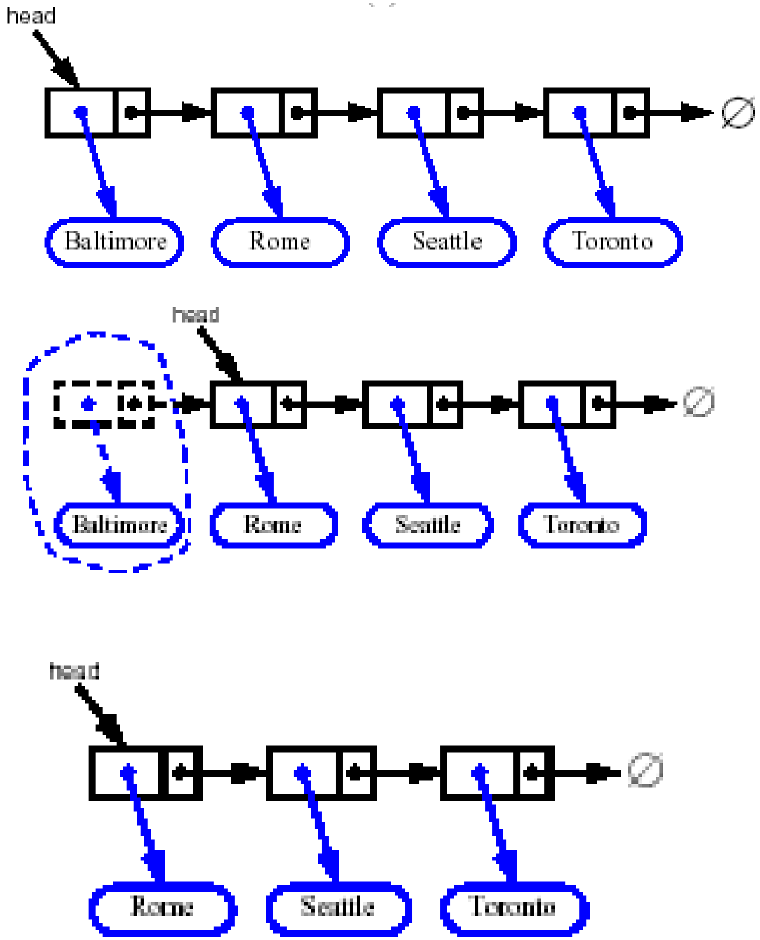
- Answer: O(1)
- Very efficient!



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
    1. Or do explicit free in C/C++

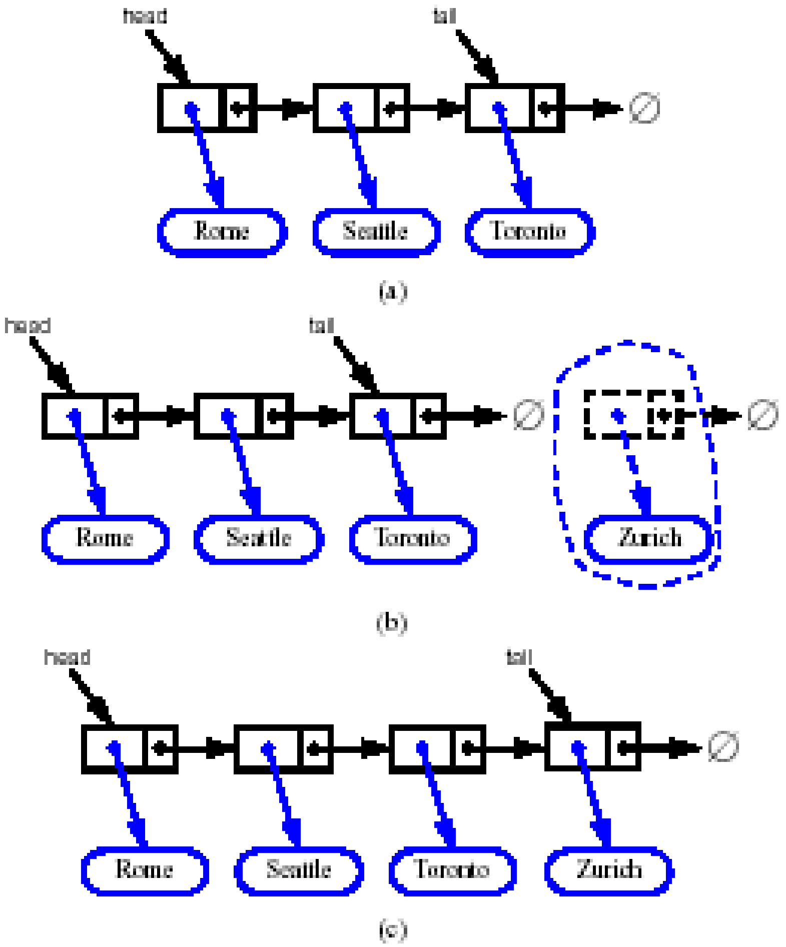
Again, the operation is O(1), and so efficient.



Inserting at the Tail

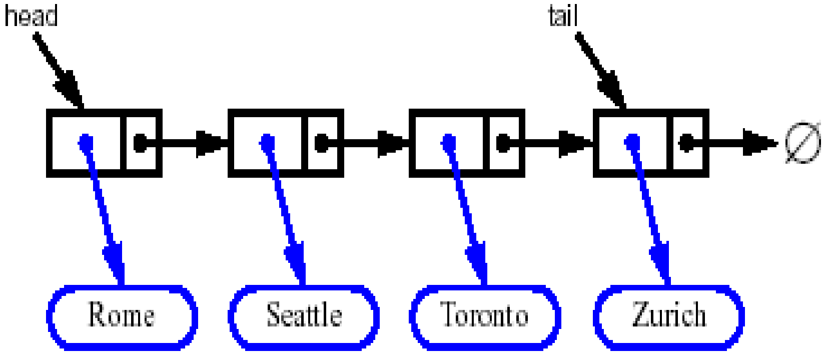
1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

Complexity: O(1)



Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- To find new tail we must walk the list from the head
  - There is no constant-time way to update the tail to point to the previous node
  - Exercise: Why not keep a "pre-tail" pointing to one before the "tail"? ("Toronto" here)
  - Advanced optional: look up "skip lists"
- Complexity: O(n)

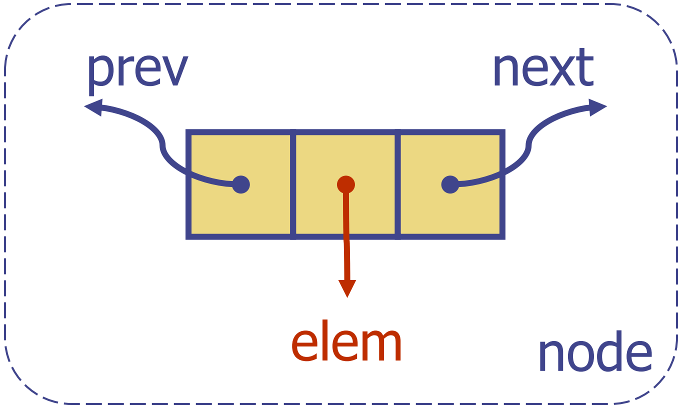


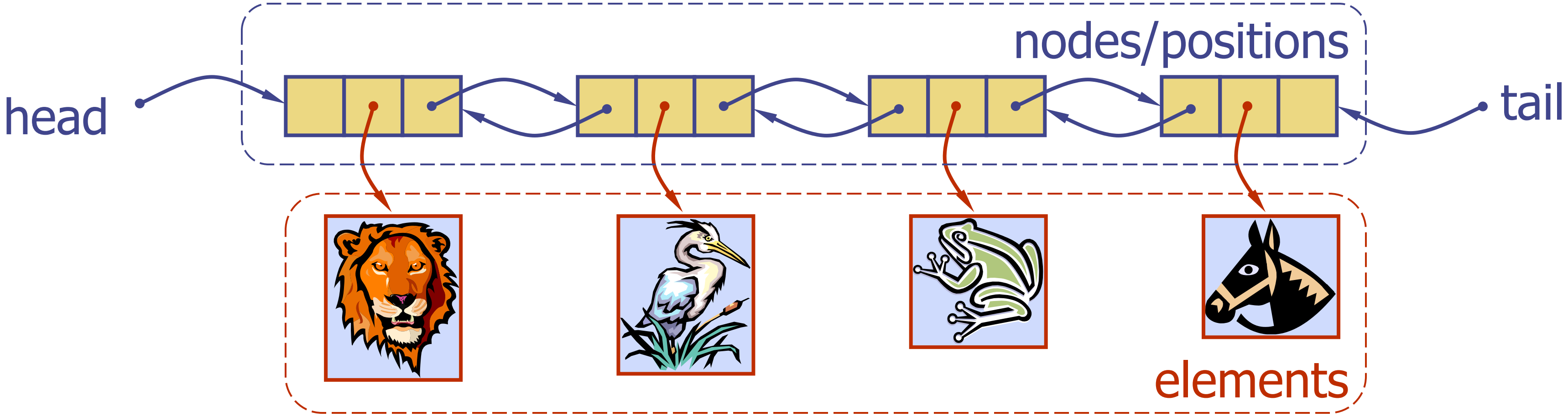
A SinglyLinkedList Class

```
class SinglyLinkedList {
    private Node head;

    public SinglyLinkedList() {
    } // head automatically set to null by Java

    public void insertAtHead(Object newElem)  {
        Node newHead = new Node(newElem,head);
        head = newHead;
    }
// etc
}
```

Doubly Linked List

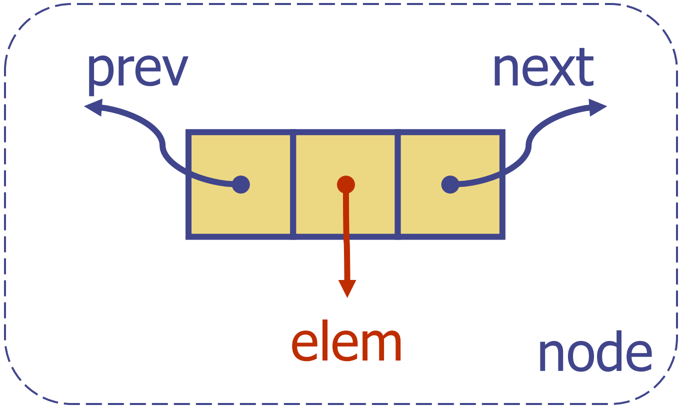
- A doubly linked list provides a natural extension of a singly linked list
- Nodes store:
    - element
    - link to the next node
    - **link to the previous node**
- **Deletion at the tail is now O(1)**
- **But uses more memory**

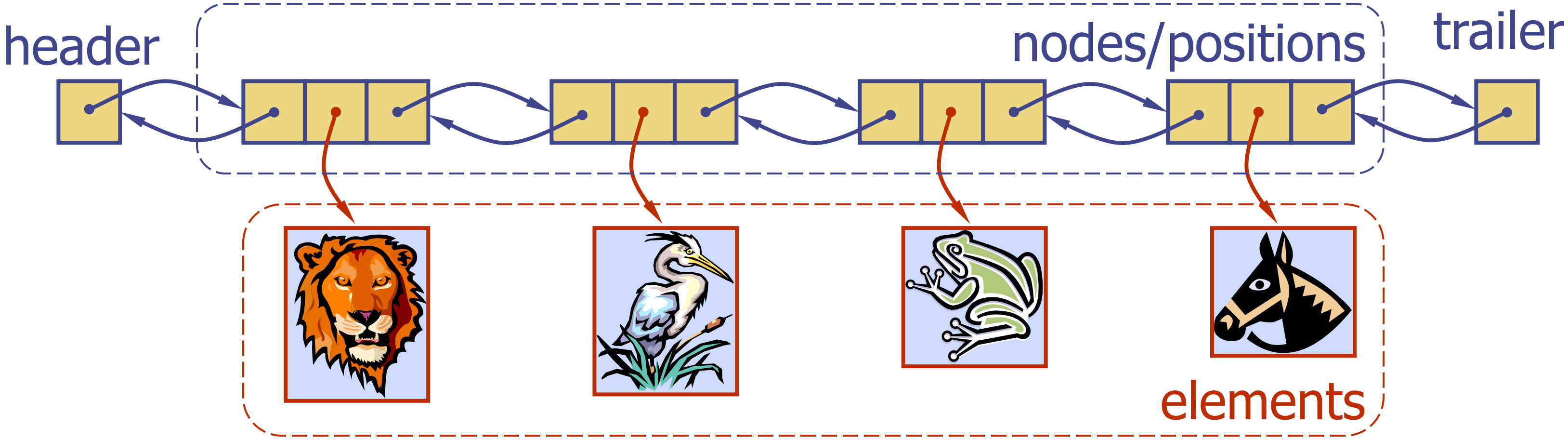




Doubly Linked List (version 2)

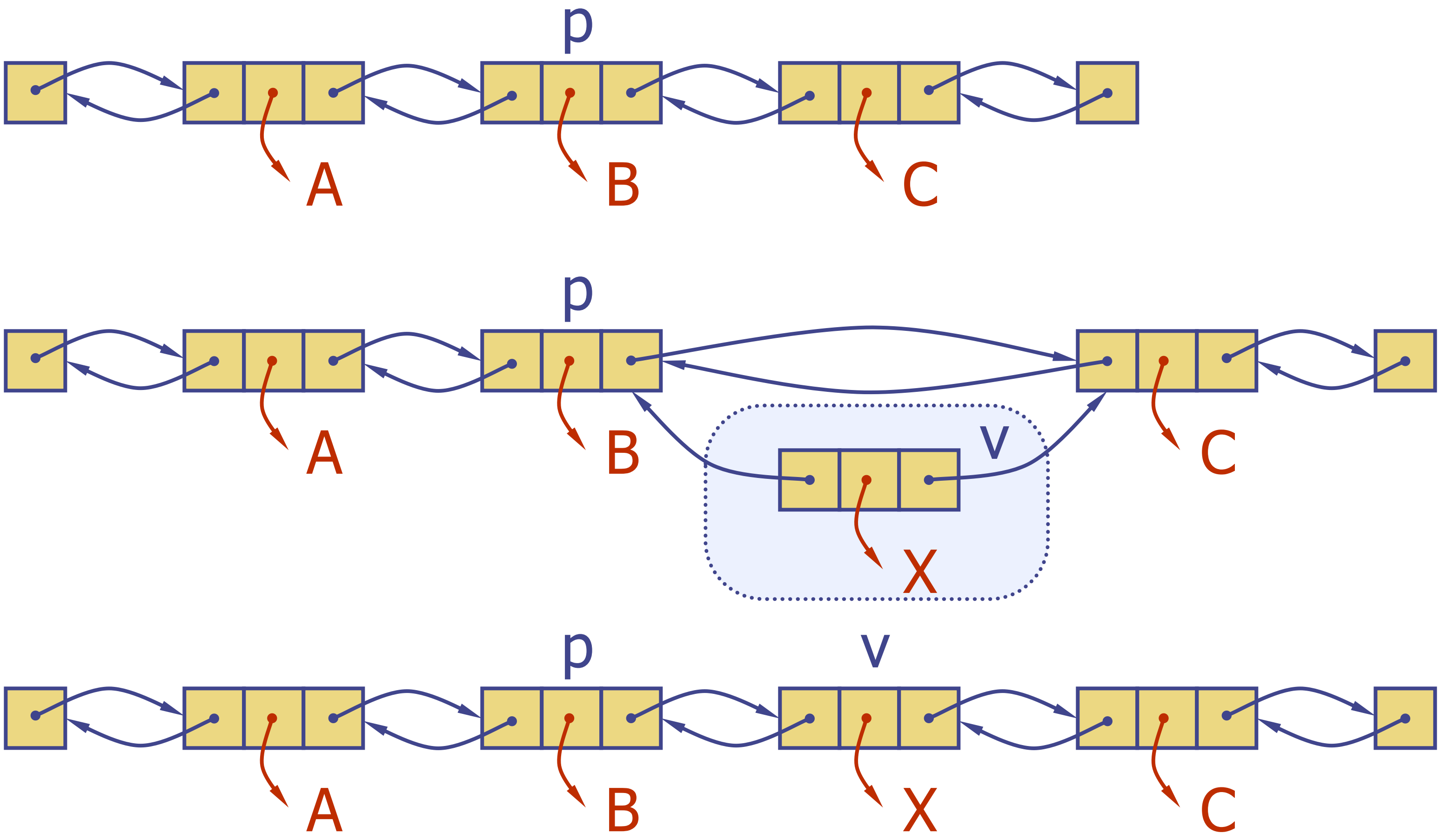
- A doubly linked list provides a natural implementation of a List
- Nodes implement "Position" and store:
  - element
  - **link to the previous node**
  - link to the next node
- Special trailer and header nodes for convenience (an alternative)





Insertion

- We visualize O(1) operation addAfter(p, X), which returns position v



Insertion Algorithm

```
Algorithm addAfter(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}
```

Minimum Expectations

- It is generally assumed that you will be familiar with singly and doubly linked lists
- You should be aware of the complexity, O(1) versus O(n), of the various standard operations, and how to implement them.