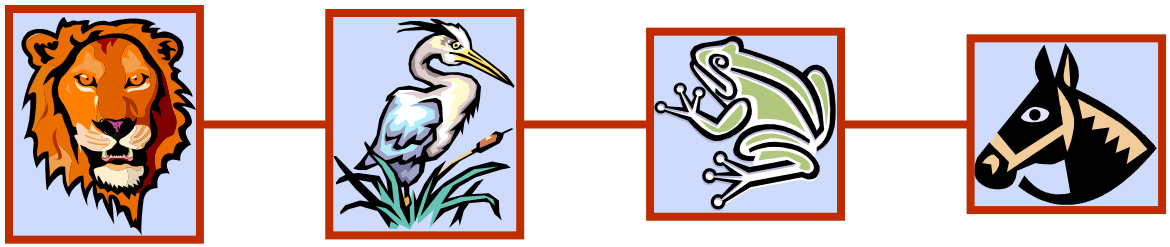
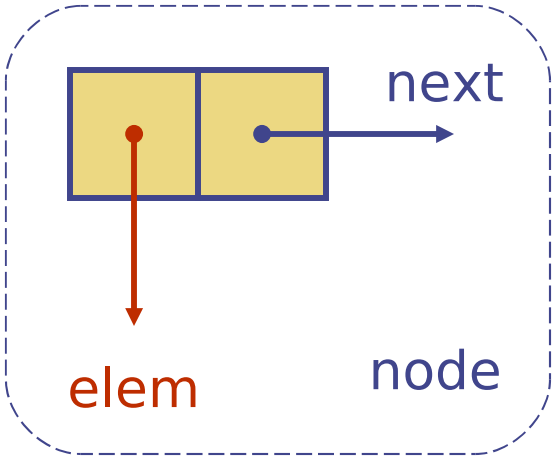
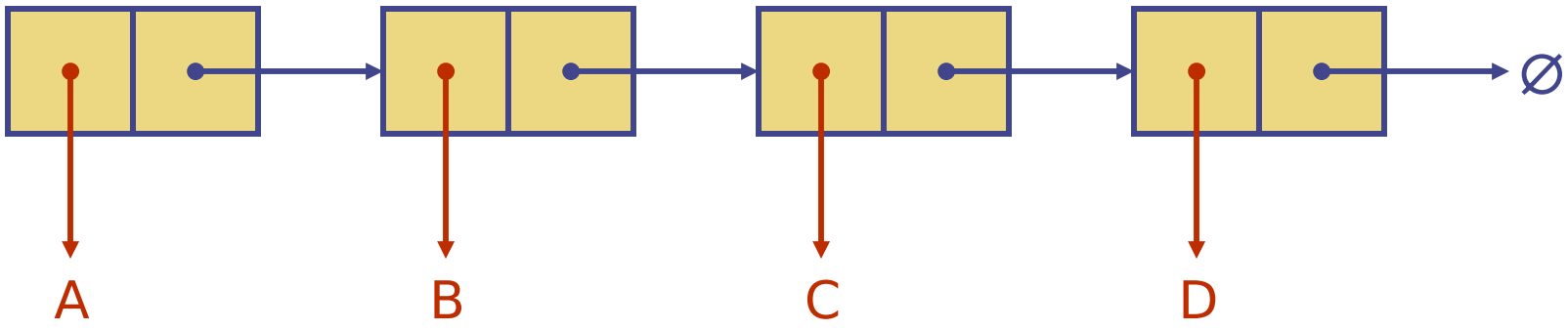
Linked Lists



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

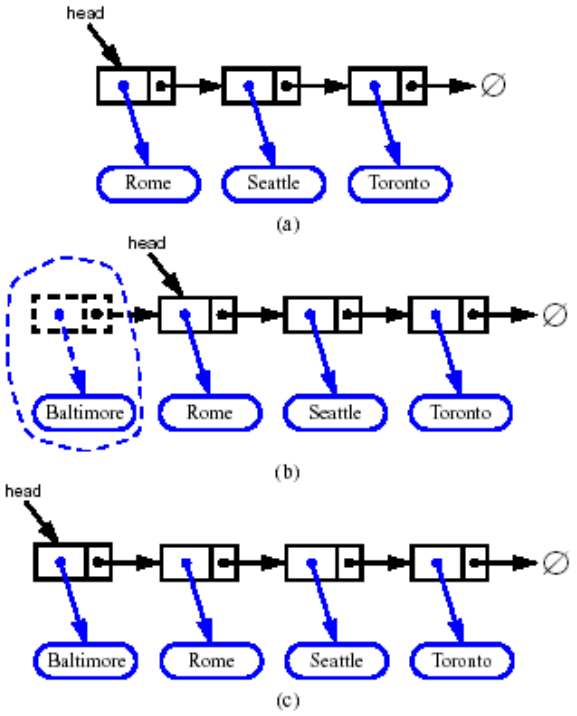
```
public class Node {
  // Instance variables:
  private Object element;
  private Node next;
  /** Creates a node with null references to its element and next node. */
  public Node() {
    this(null, null);
  }
  /** Creates a node with the given element and next node. */
  public Node(Object e, 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

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

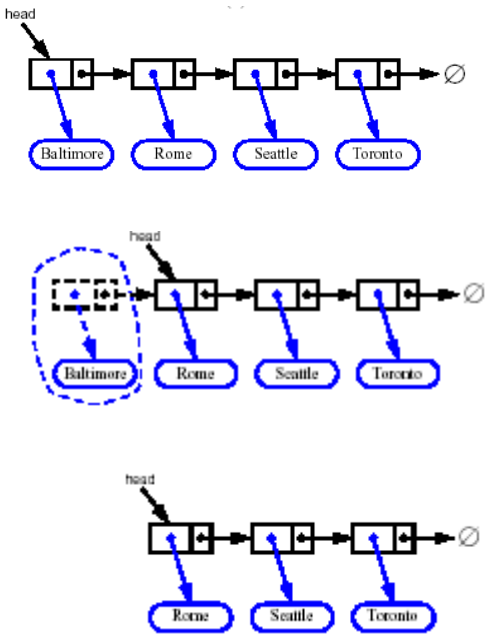
Inserting at the Head

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



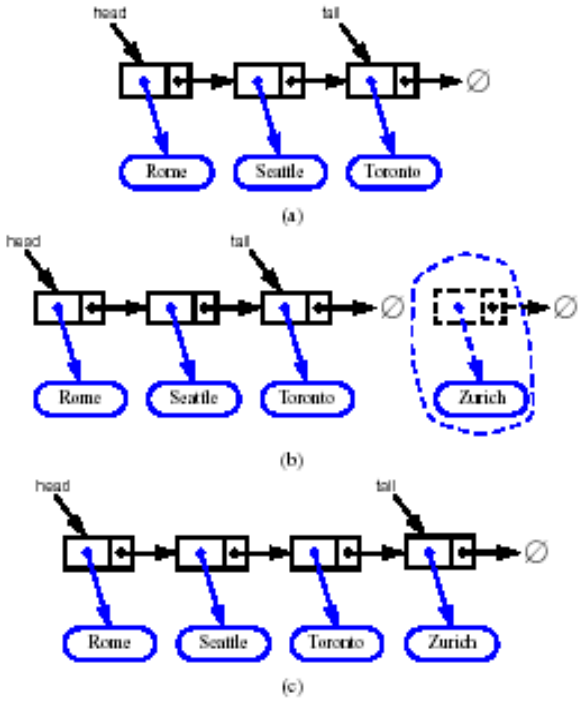
Removing at the Head

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



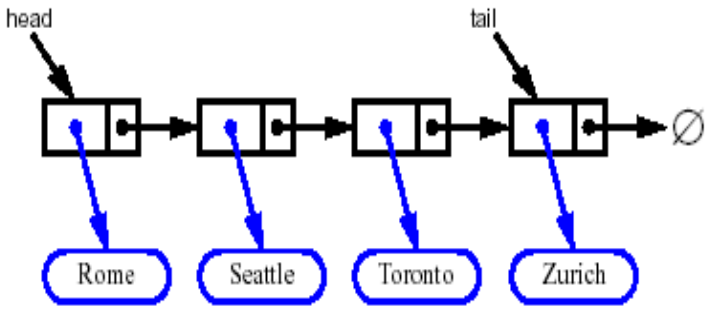
Inserting at the Tail

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



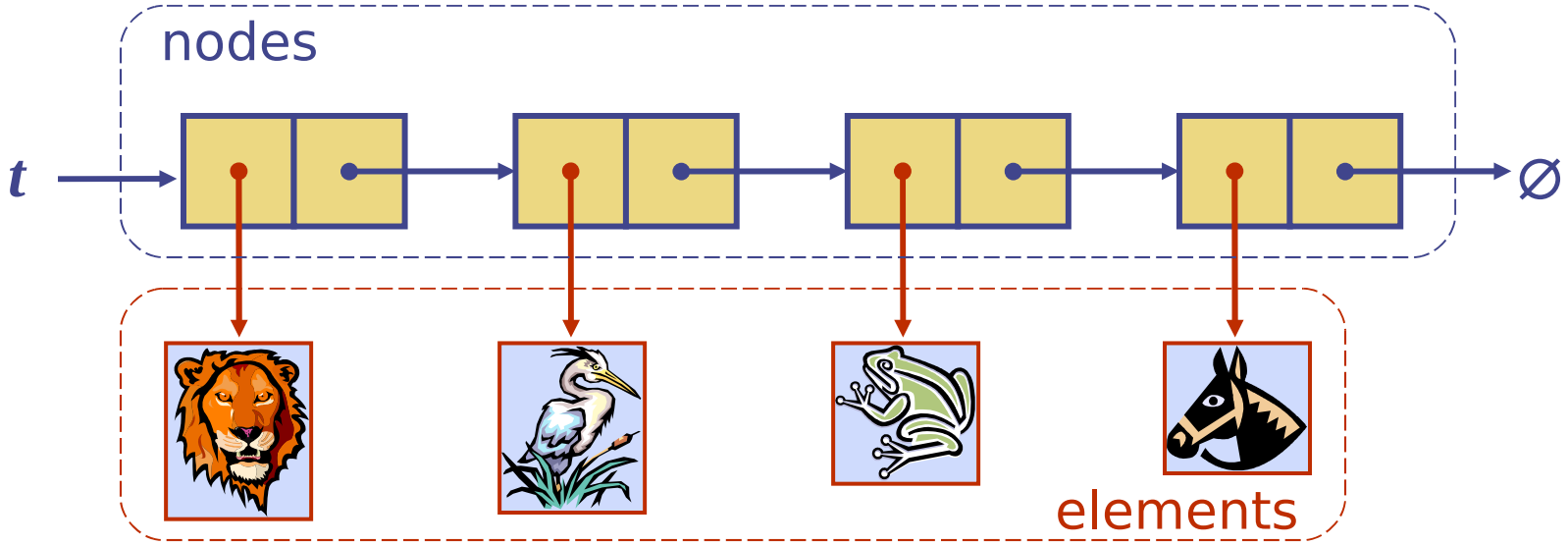
Removing at the Tail

- 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



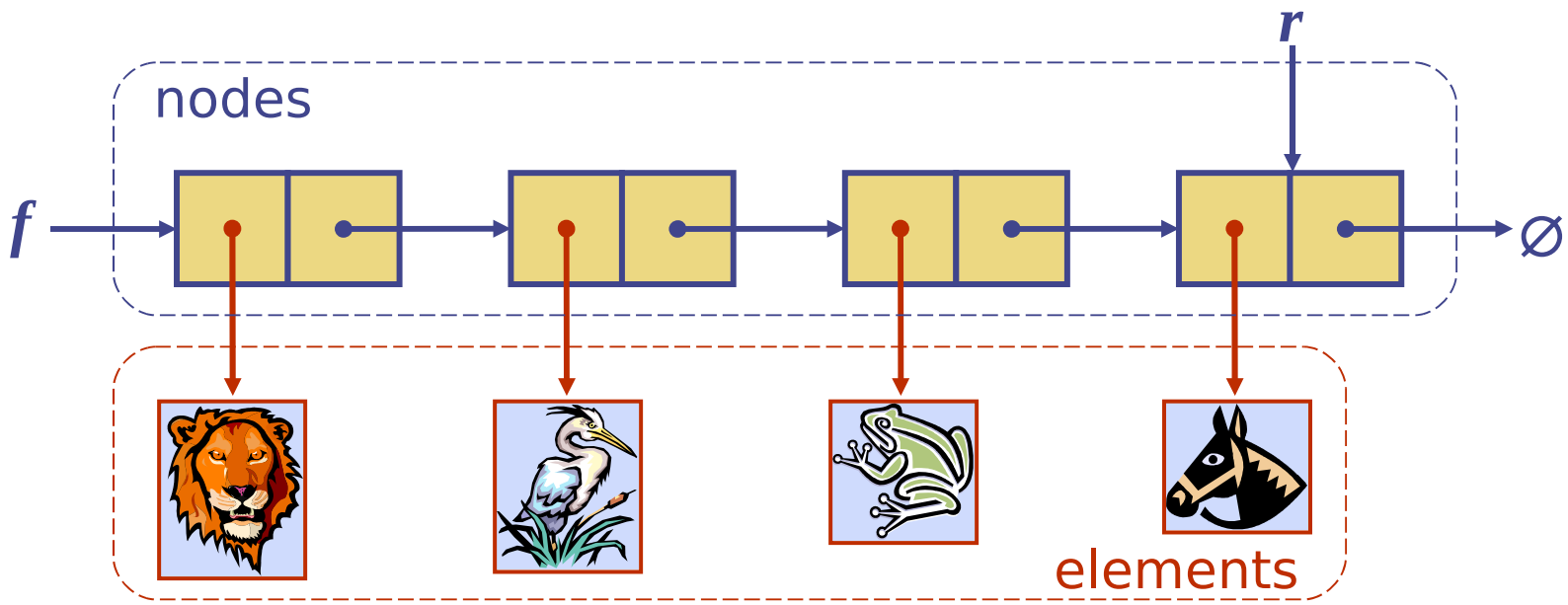
Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



Queue with a Singly Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time



Linked Lists with Header and Trailer Nodes

- Simplify insertion and deletion by never inserting an item before first or after last item and never deleting first node
- Set a header node at the beginning of the list containing a value smaller than the smallest value in the data set
- Set a trailer node at the end of the list containing a value larger than the largest value in the data set

Linked Lists with Header and Trailer Nodes

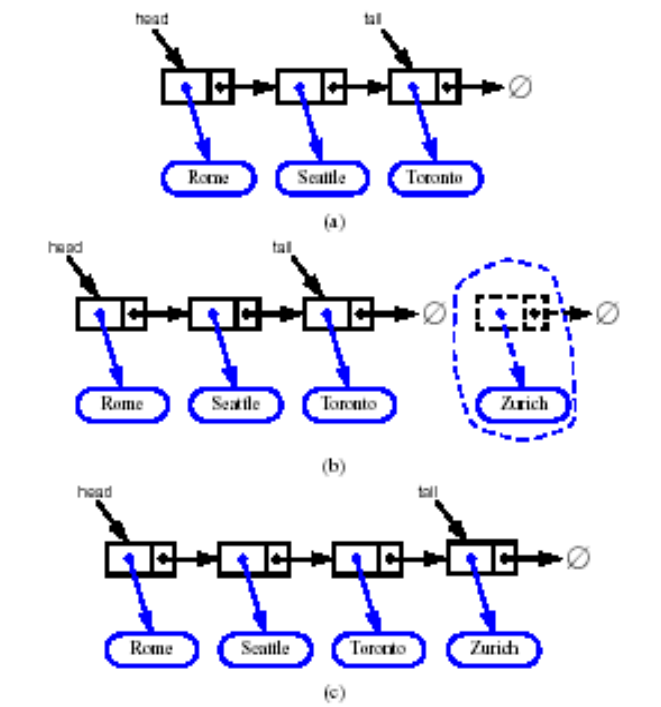
- These two nodes, header and trailer, serve merely to simplify the insertion and deletion algorithms and are not part of the actual list.
- The actual list is between these two nodes.

Singly linked list with ‘tail’ sentinel

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

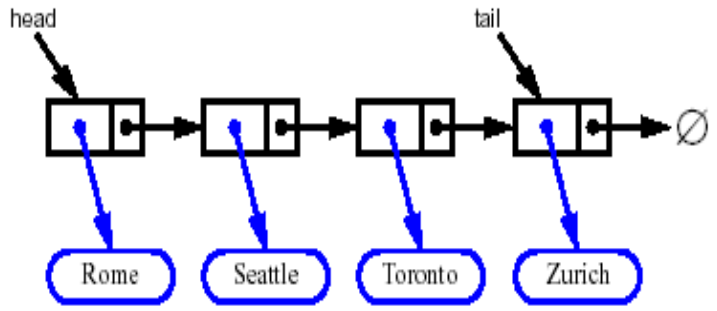
Inserting at the Tail

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



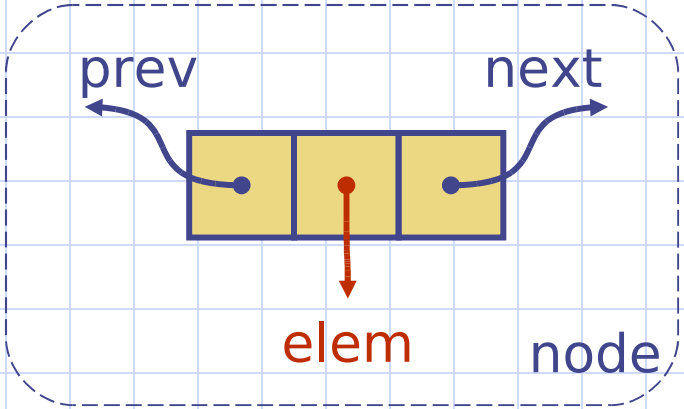
Removing at the Tail

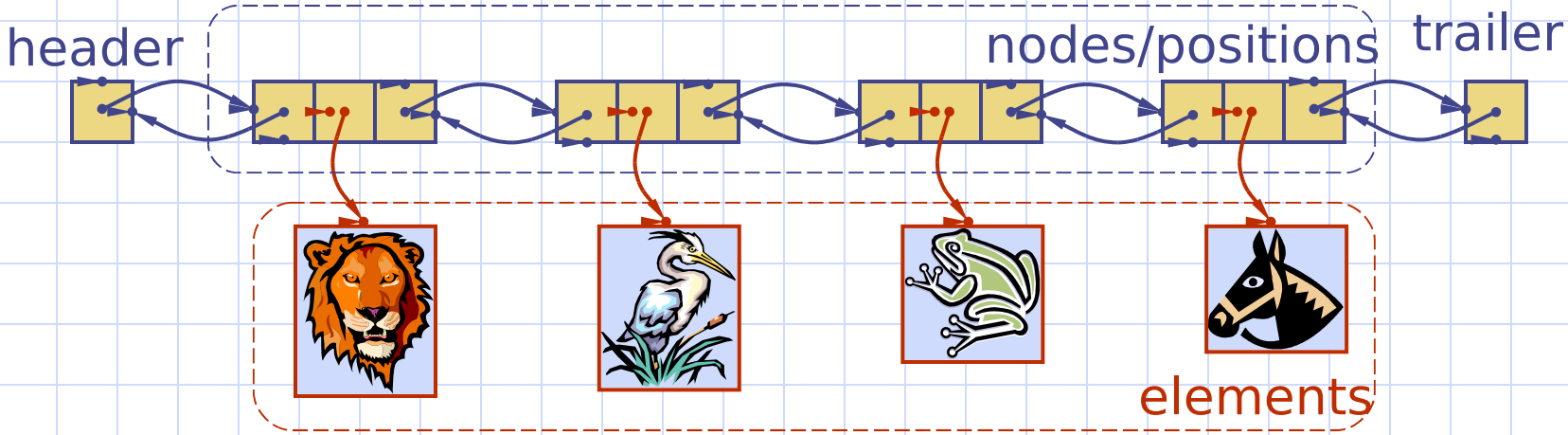
- Removing at the tail of a singly linked list cannot be efficient!
- There is no constant-time way to update the tail to point to the previous node



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

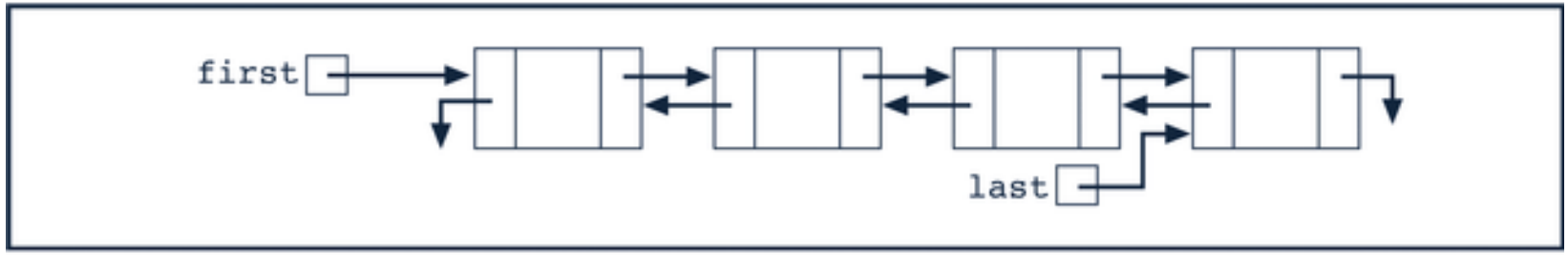




Doubly Linked List

- Every node:
  - has a next reference variable and a back reference variable
  - (except the last node) contains the address of the next node
  - (except the first node) contains the address of the previous node
- Can be traversed in either direction

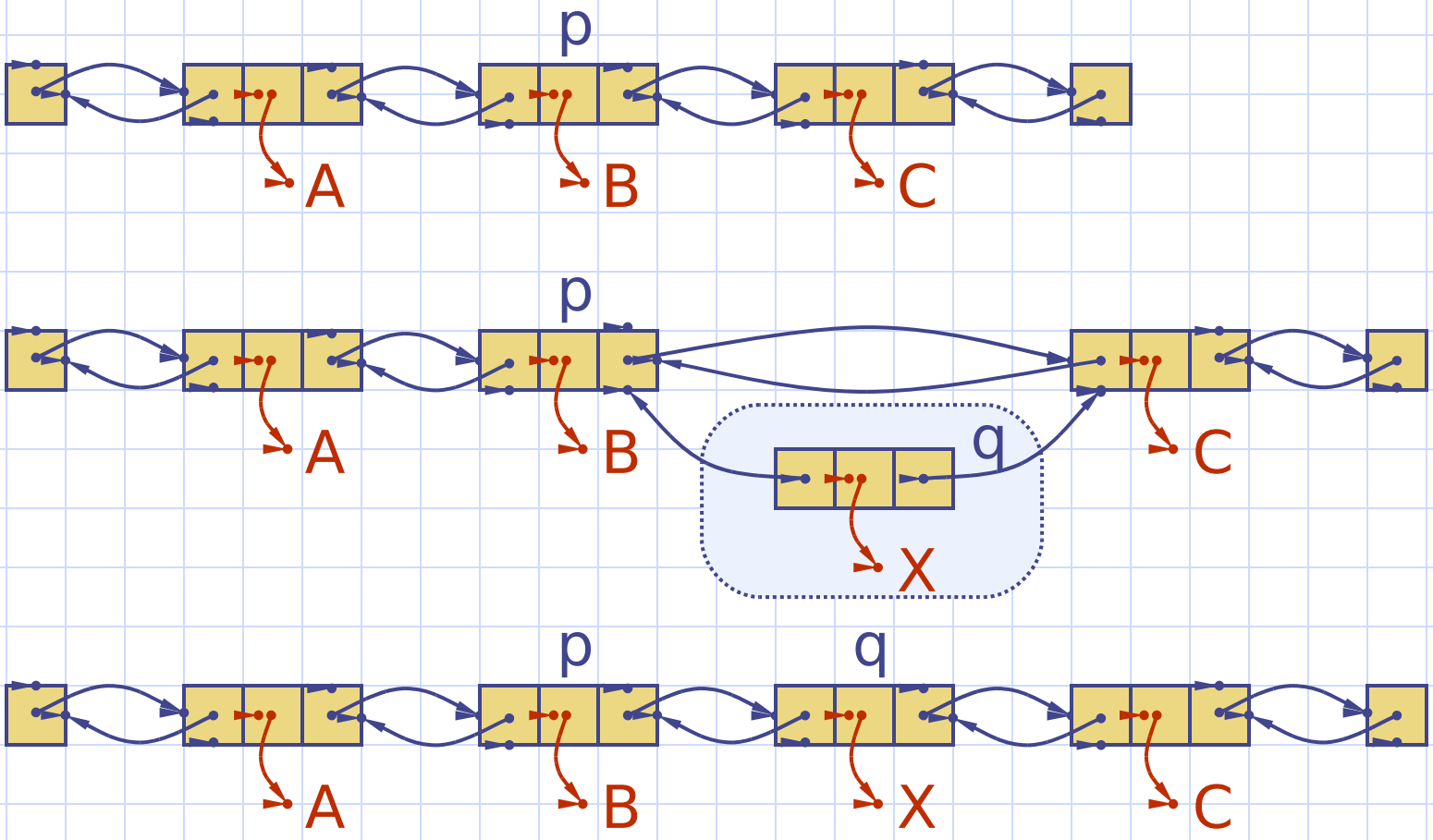
Doubly Linked List



Doubly linked list

Insertion

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

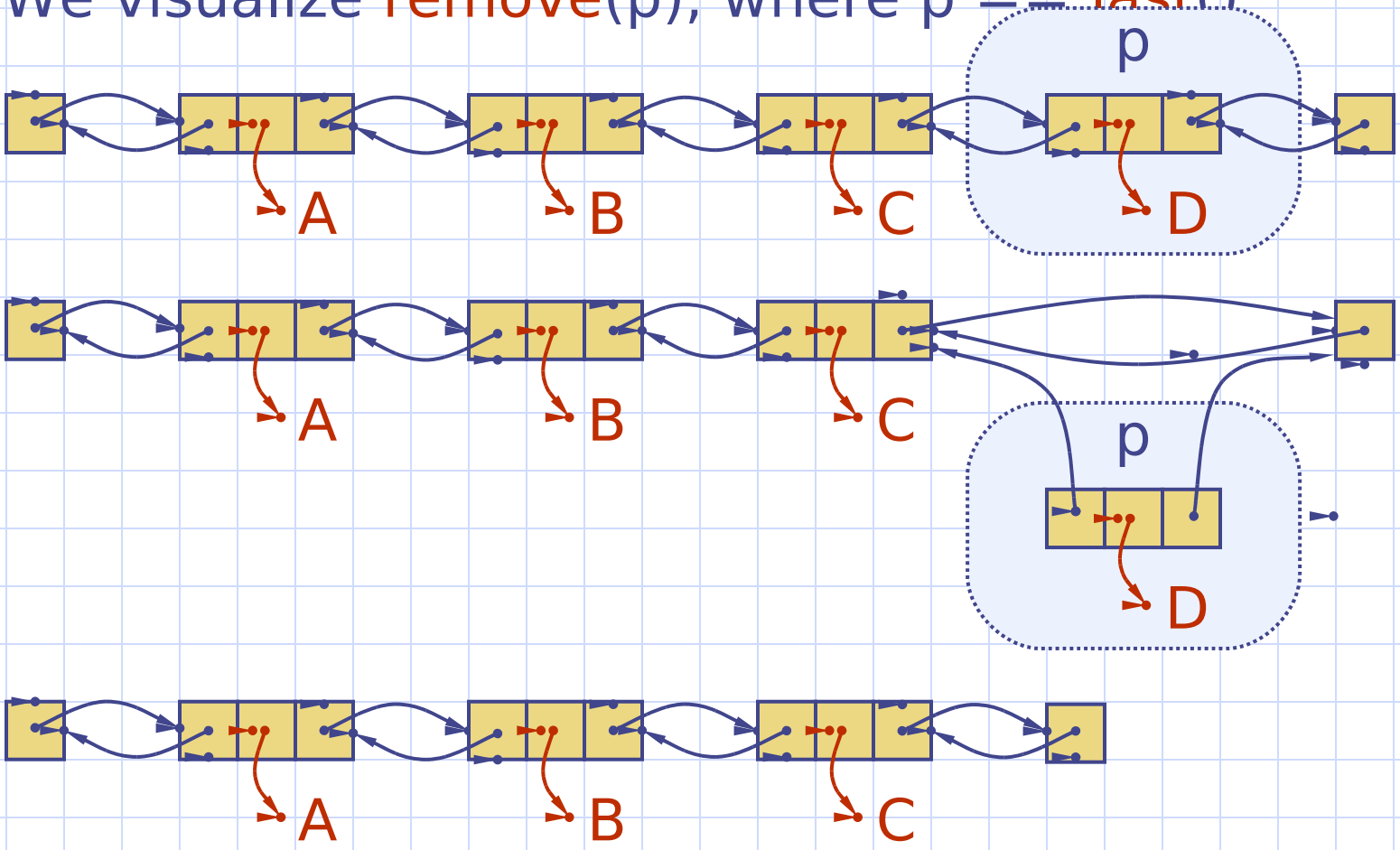


Insertion Algorithm

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*

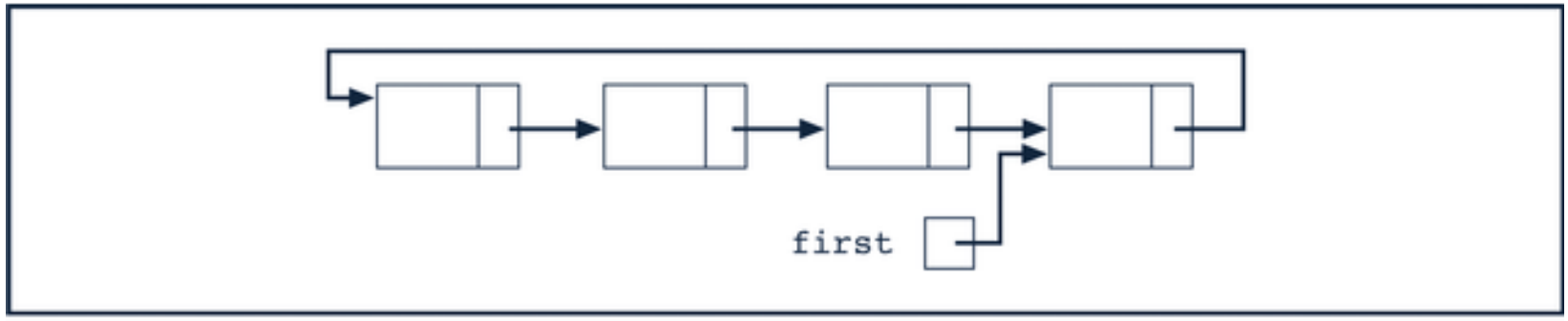
Worst-cast running time

- 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)

Circular Linked List

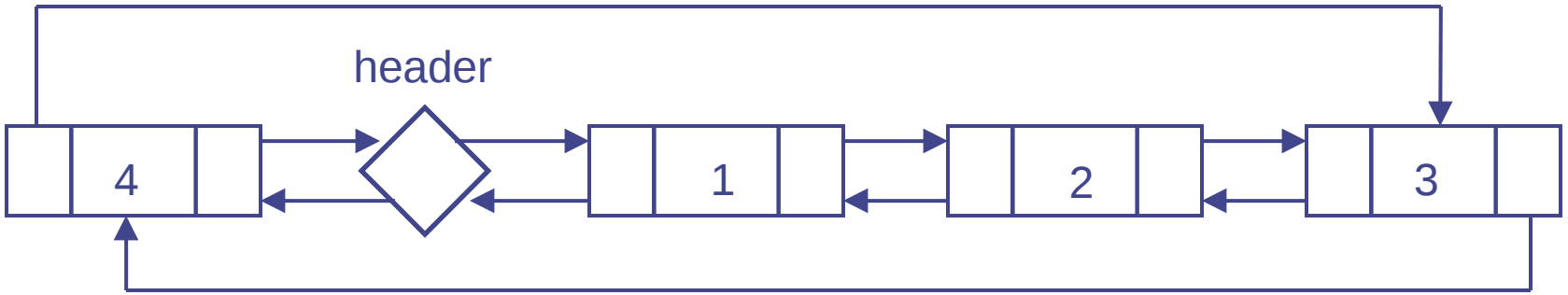
- A linked list in which the last node points to the first node is called a circular linked list
- In a circular linked list with more than one node, it is convenient to make the reference variable first point to the last node of the list

Circular Linked List



Circular linked list with more than one node

Circular Doubly Linked List



- Contains a *sentinel node* called *header* whose **value is never used**.
- The declaration of the list begins with the declaration of the header.