Data Structures and Algorithms (02-24-00108)

Part # 3 Dynamic Memory Allocations

Course was created by
Dr. Adel A. El-Zoghabi

Professor of Computer Science & Information Technology
Faculty of Computer and Data Science
Alexandria University

Course Presentation by
Dr. Maged Abdelaty, and Dr. Doaa B. Ebaid

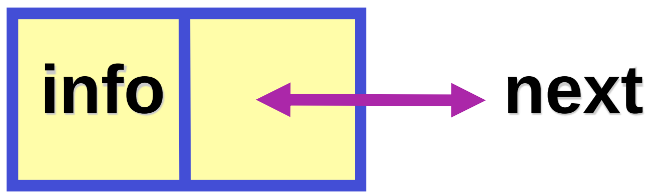
email: Adel.Elzoghby@alexu.edu.eg

Part # 3

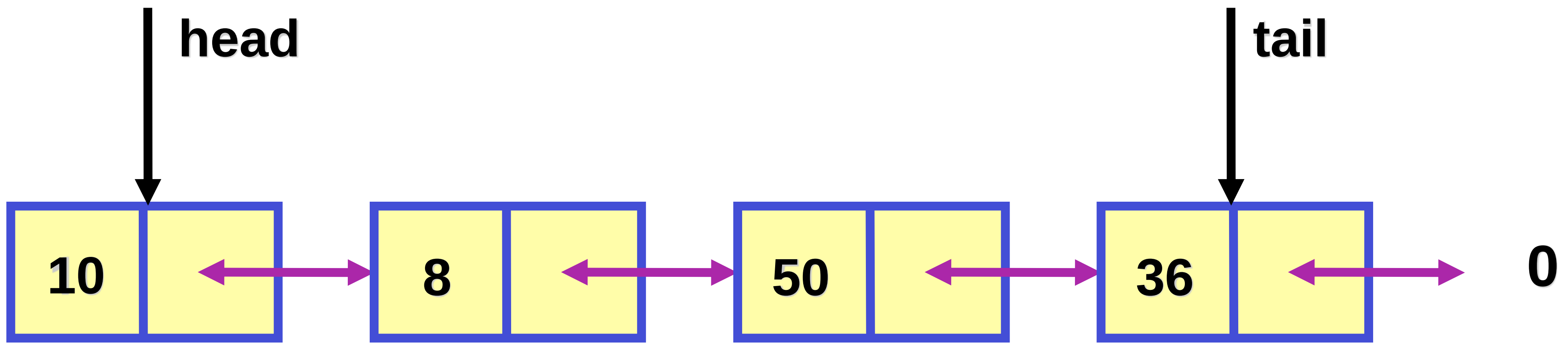
- **Dynamic Memory Allocation**
  1. **Singly Linked Lists,**
  2. **Doubly Linked Lists,**
  3. **Circular Lists,**
  4. **Applications:**
     - A. **Sparse Tables,**
     - B. **Other Applications**

Singly Linked Lists

- A singly linked list (SLL) is a concrete data structure consisting of a sequence of nodes,



- Each node stores:
  - A value called info of specific type, and
  - A pointer link to the next node, called next
- Two pointers of type node identify the list, these are: head and tail



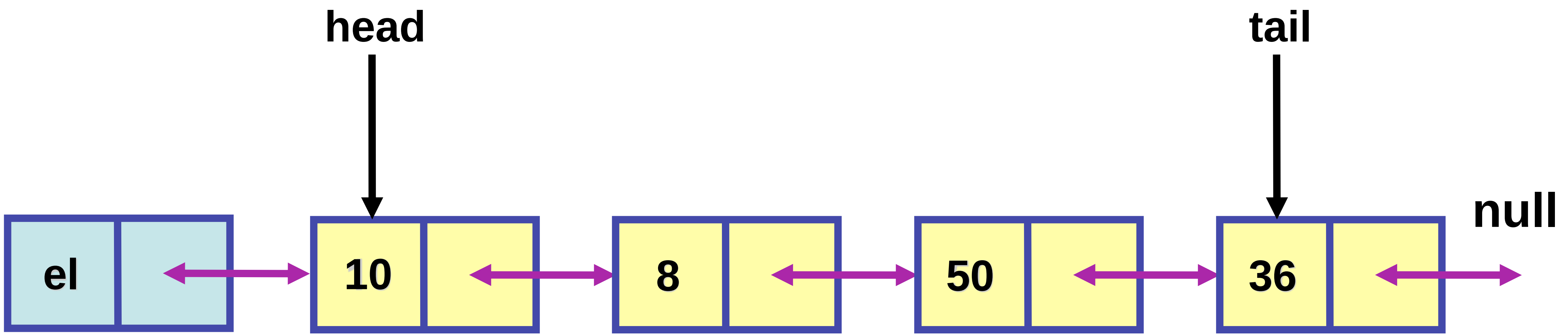
The SLL Node Class Definition

```
class Node {
    int info;
    Node next;
    Node() { info = 0; next = null; }
    Node(int el) {
            info = el;
            next = null;
    }
    Node(int el, Node n) {
            info = el;
            next = n;
    }
}
```

The SinglyLinkedList Class in Java

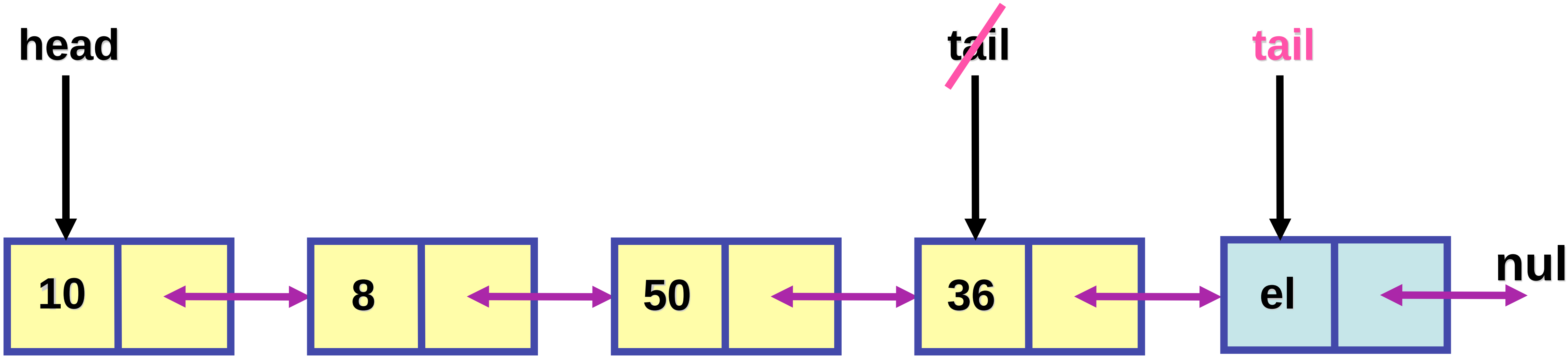
```
public class SinglyLinkedList {
    public static void main(String argv[]) {
        …
    }
    class Node {
        …
    }
    private Node head;
    private Node tail;
    SinglyLinkedList( ) {
        head = tail = null;
    }
    public void addToHead(int el)   {      …      }
    public void addToTail(int el)   {      …      }
    public int deleteFromHead()     {      …      }
    public int deleteFromTail()     {      …      }
    public int deleteNode()         {      …      }
    public boolean isInList(int el) {      …      }
    public void printList()         {      …      }
}
```

Method addToHead



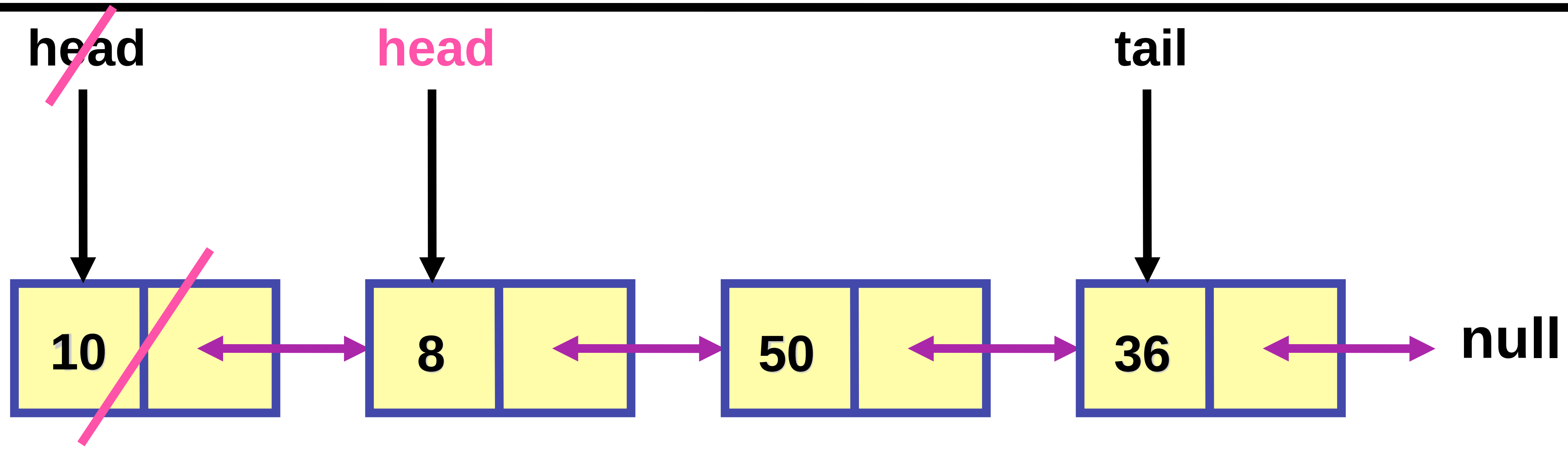

```
public void addToHead(int el) {
        if ((head == null) && (tail == null))
          { head = tail = new Node(el); }
        else
          { head = new Node(el, head); }
}
```

Method addToTail



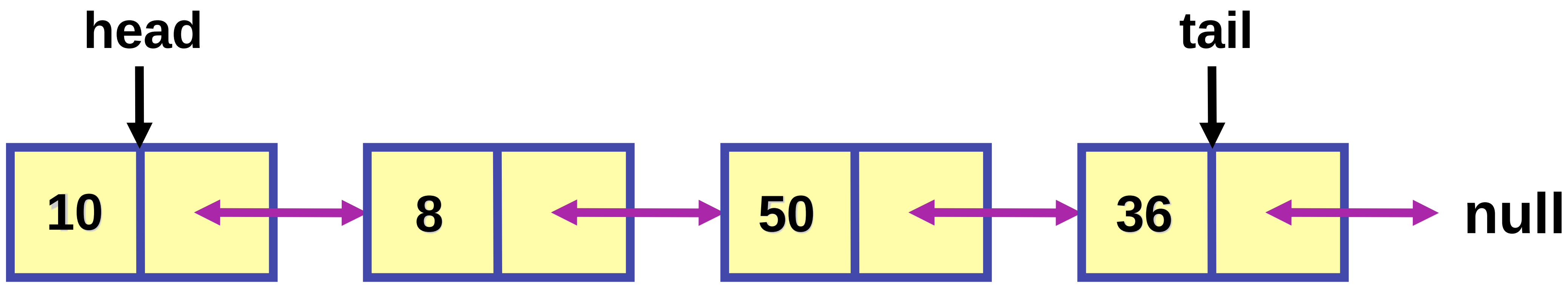

```
public void addToTail(int el) {
    if ((head == null) && (tail == null)) {
        head = tail = new Node(el);
    }
    else {
        tail.next = new Node(el);
        tail = tail.next;
    }
}
```

Method deleteFromHead




```
public int deleteFromHead() {
   if ( isEmpty() ) { return 0; } // Better to add a new method isEmpty
   else {
      int el = head.info;
      if (head == tail)    // if only one node in the list
         { head = tail = null; }
      else { head = head.next; }
      return el;
   }
}
```

Method deleteFromTail

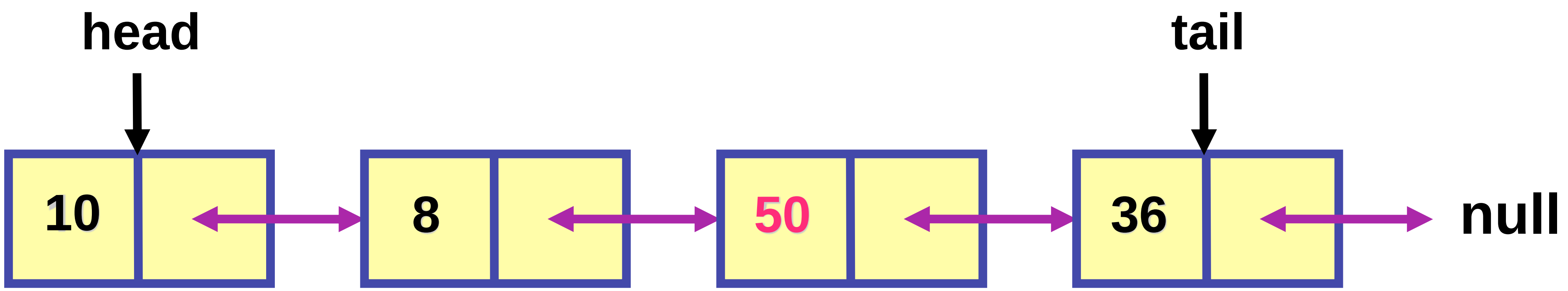



```
public int deleteFromTail() {
   if ( isEmpty() ) { return 0; }
   else {
      int el = tail.info;
      if (head == tail)  {  // if only one node in the list
         head = tail = null;
      }
      else { // if more than one node in the list, find the predecessor of tail
         Node tmp;
         for (tmp = head; tmp.next != tail; tmp = tmp.next);
         tail = tmp;    tail.next = null;
      }
      return el;
   }
}
```

Method deleteNode

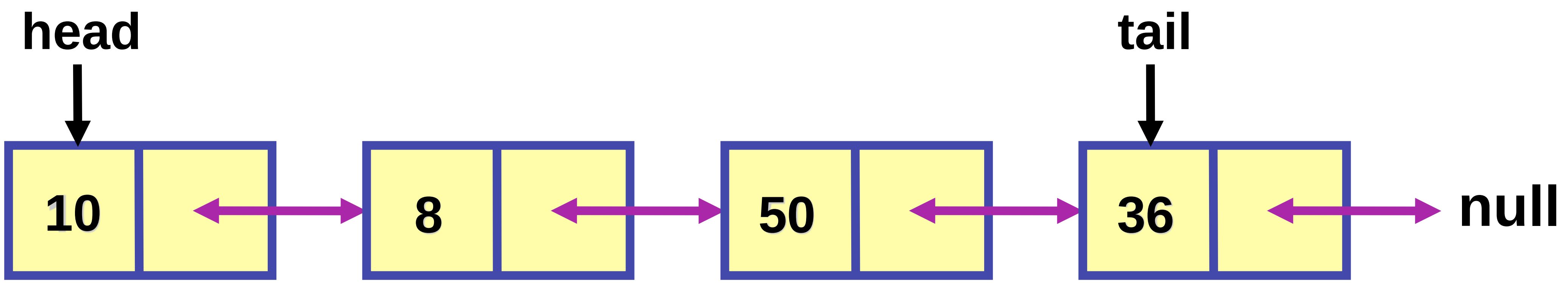
```
public void deleteNode(int el) {
    if ( !isEmpty() ) { // if not empty list
        if ((head == tail) && (el == head.info)) { // if only one node in the list
            head = tail = null;   // and el is the head
        }
        else if (el == head.info) { // if more than one in the list and el is head
            head = head.next;
        }
        else { // search for el starting at head, ending at tail
            Node pred, tmp;
            for (pred = head, tmp = head.next; (tmp != null) && !(tmp.info == el);
                                                    pred = pred.next, tmp= tmp.next);
            if (tmp != null) {
                pred.next = tmp.next;
                if (tmp == tail) tail = pred;
            }
        }
    }
}
```

Method isInList




```
public boolean isInList(int el) {
        Node tmp;
        for (tmp = head; tmp != null && !(tmp.info == el);
                                tmp = tmp.next);
        return (tmp != null);
}
```

Method printList




```
public void printList() {
    Node tmp = head;
    System.out.println("The Singly Linked List is:");
    while (tmp != null) {
        System.out.println(tmp.info);
        tmp = tmp.next;
    }
}
```

Testing the SinglyLinkedList Class

```
publix static void main() {
  SinglyLinkedList MyList = new SinglyLinkedList();
  MyList.addToHead(10);
  MyList.addToHead(20);
  MyList.printList();
  MyList.addToTail(100);
  MyList.addToTail(200);
  MyList.printList();
  System.out.println("Searching for 100: ", MyList.isInList(100));
  System.out.println("Deleted From Head: ", MyList.deleteFromHead());
  MyList.printList();
  System.out.println("Deleted From Tail: ", MyList.deleteFromTail());
  MyList.printList();
  System.out.println("Deleted Node 100: ");
  MyList.deleteNode(100);
  MyList.printList();
  System.out.println("Searching for 100: ", MyList.isInList(100));
}
```

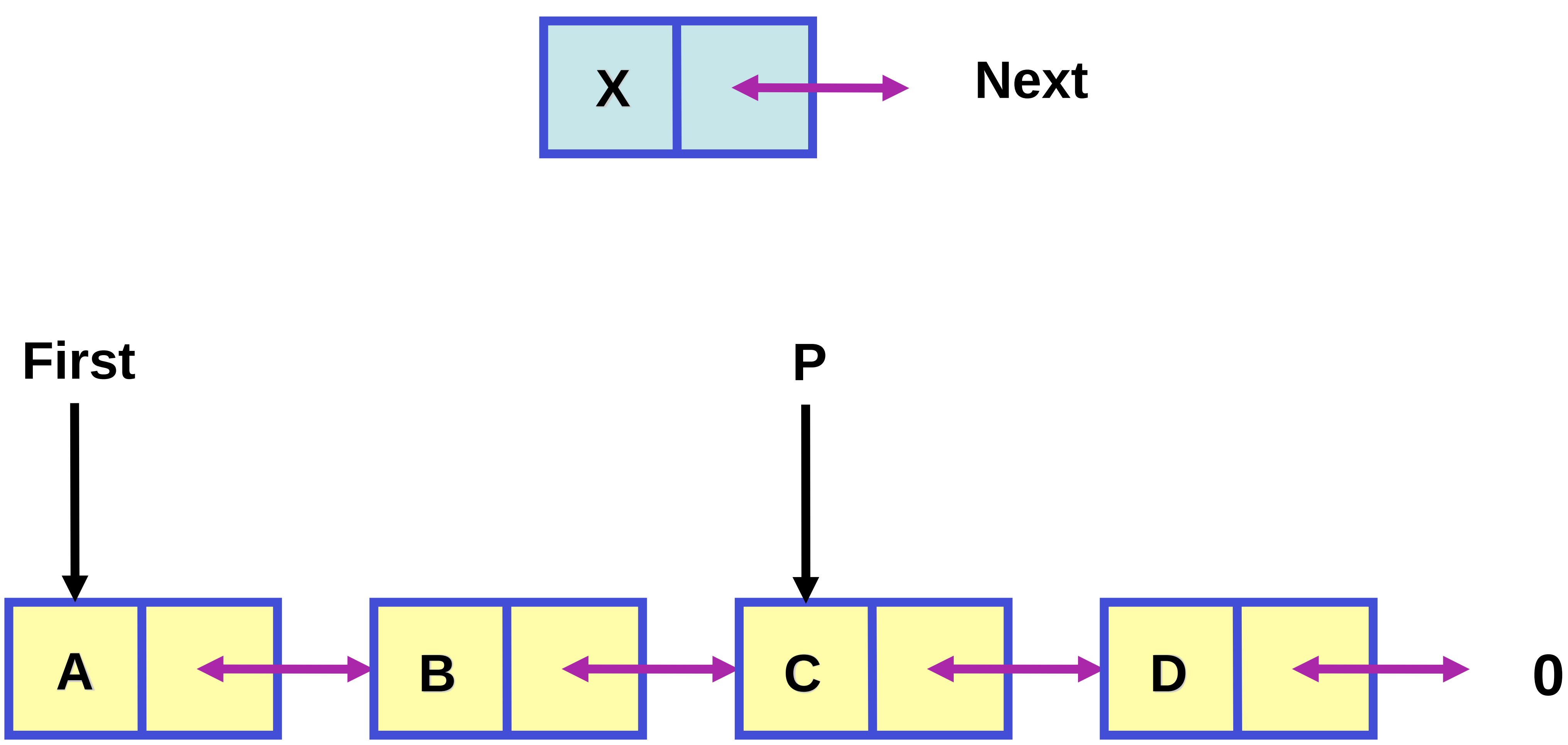
Testing the SinglyLinkedList Class

```
The Singly Linked List is:
20
10
The Singly Linked List is:
20
10
100
200
Searching for 100: true
Deleted From Head: 20
The Singly Linked List is:
10
100
200
```

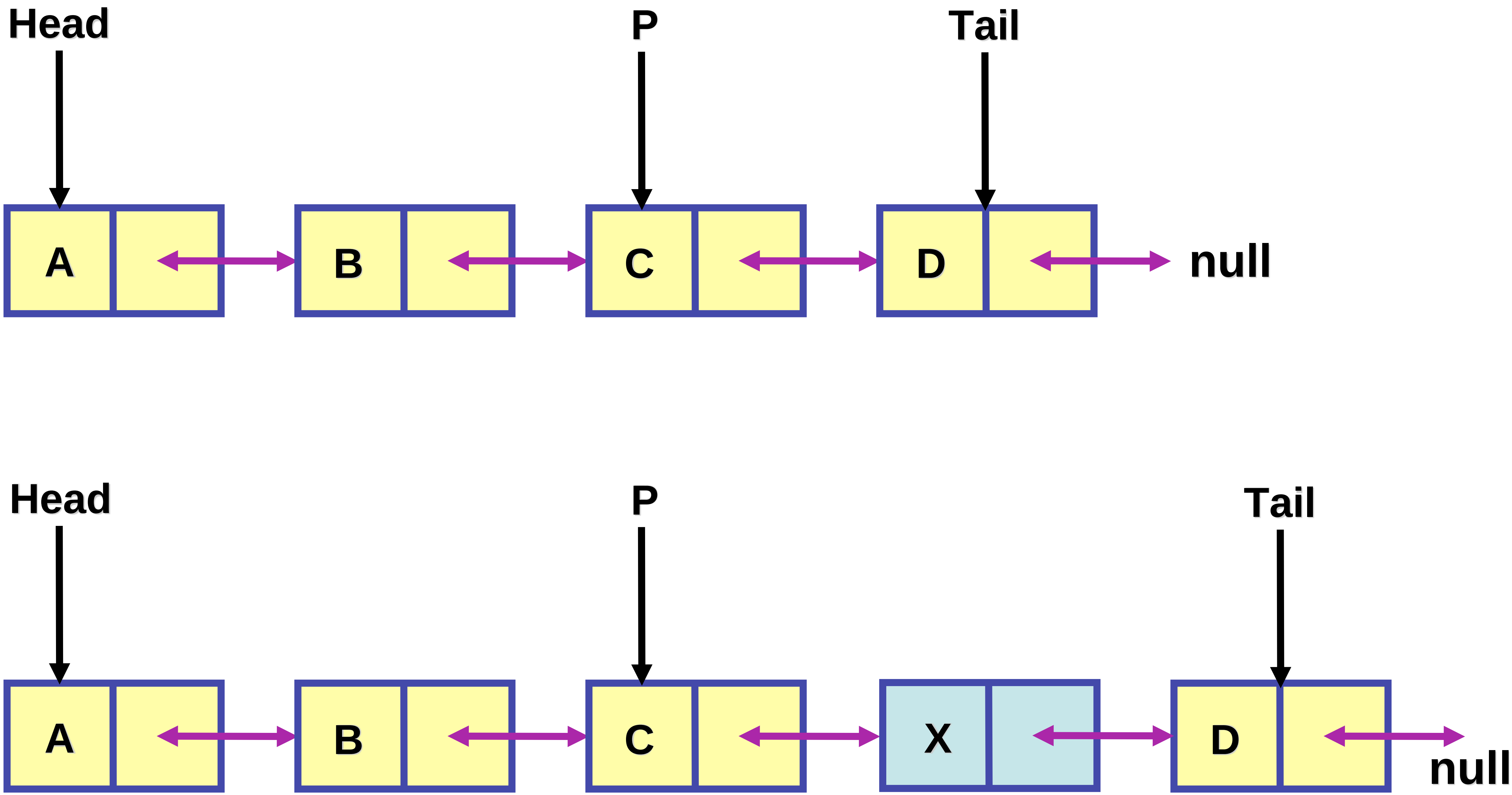
```
Deleted From Tail: 200
The Singly Linked List is:
10
100
Deleted Node 100:
The Singly Linked List is:
10
Searching for 100: false
```

Operations on a Singly Linked Lists

- **Inserting a node whose value is X after a node pointed to by position pointer P**

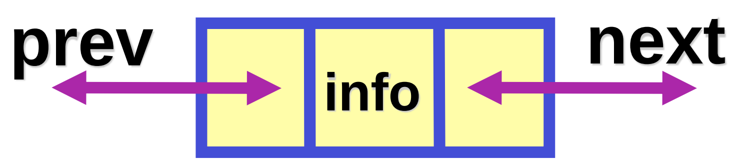


Singly Linked List: InsertAfter

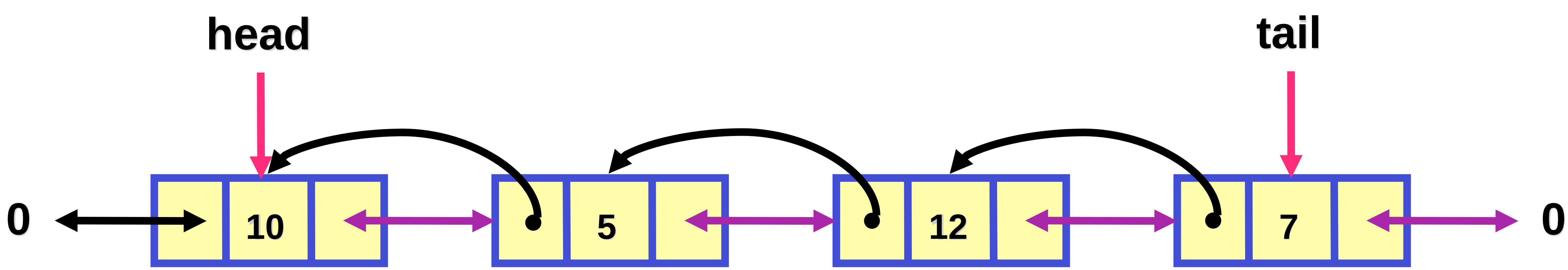


Doubly Linked Lists

- In a doubly linked, nodes store:
  - Element info,
  - Link to the previous node, and
  - Link to the next node



- Two pointers tail and head,
- In order to be able to design nodes with general type (int, real, char, …, etc.), we will use templates



The Node Class for Doubly Linked List

```
class Node {
        Object info;
        Node next, prev;
        Node() { info = null; next = null; prev = null; }
        Node(Object el) {
                info = el;
                next = null; prev = null;
        }
        Node(Object el, Node n, Node p) {
                info = el;
                next = n; prev = p;
        }
}
```

The DoublyLinkedList Class

```
public class DoublyLinkedList {
        public static void main(String argv[]) { … }
        class Node {
            …
        }
        private Node head;
        private Node tail;
        DoublyLinkedList( ) {
                head = tail = null;
        }
        public void addToTail(Object el) { … }
        public Object deleteFromTail() { … }
        public void printList() { … }
}
```

Method addToTail

```
public void addToTail(Object el) {
    if (tail != null) {
        tail = new Node(el, null, tail);
        tail.prev.next=tail;
    }
    else head = tail = new Node(el);
}
```

Method deleteFromTail

```
public Object deleteFromTail() {
    if (head  == null) return null;
    else {
        Object el = tail.info;
        if (head ==tail) { // if only one node in the list
            head = tail = null;
        }
        else {
            tail = tail.prev;
           tail.next = null;
        }
        return el;
    }
}
```

Method printList

```
public void printList() {
    Node tmp = head;
    System.out.println("The Doubly Linked List is:");
    while (tmp != null) {
        System.out.println(tmp.info);
        tmp = tmp.next;
    }
}
```

Testing the DoublyLinkedList Class

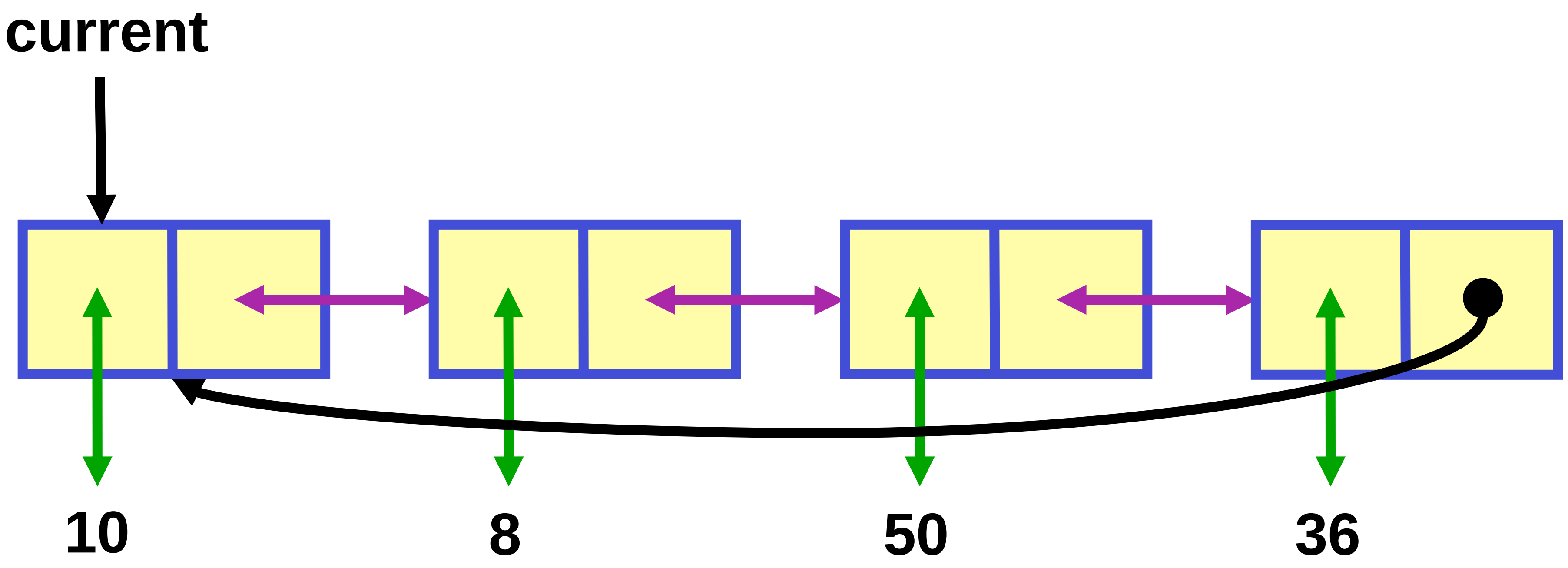
```
public static void main(String argv[]) {
    DoublyLinkedList MyList = new DoublyLinkedList();
    MyList.addToTail(new Integer(10));
    MyList.printList();
    MyList.addToTail(new Integer(20));
    MyList.printList();
    MyList.addToTail(new Integer(30));
    MyList.printList();
    System.out.println
            ("Deleted From Tail: "+MyList.deleteFromTail());
    MyList.printList();
}
```

Testing the DoublyLinkedList Class

```
The Doubly Linked List is:
10
The Doubly Linked List is:
10
20
The Doubly Linked List is:
10
20
30
Deleted From Tail: 30
The Doubly Linked List is:
10
20
```

Circular Lists

- In some situations, a circular list is needed in which nodes form a ring, or in other words, the list is finite and each node has a successor,
- Below is an example of circular singly linked list

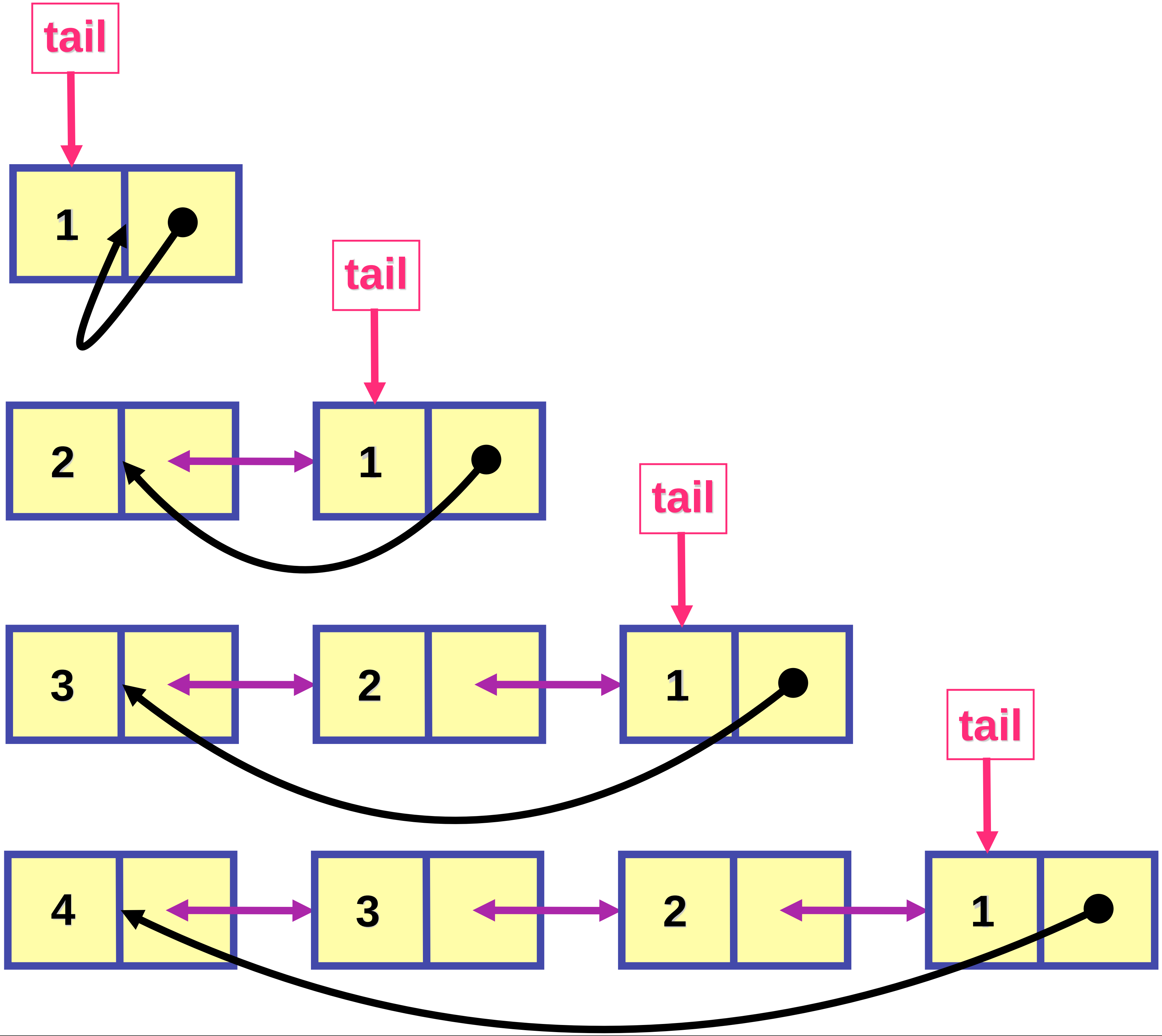


Circular Singly Linked Lists

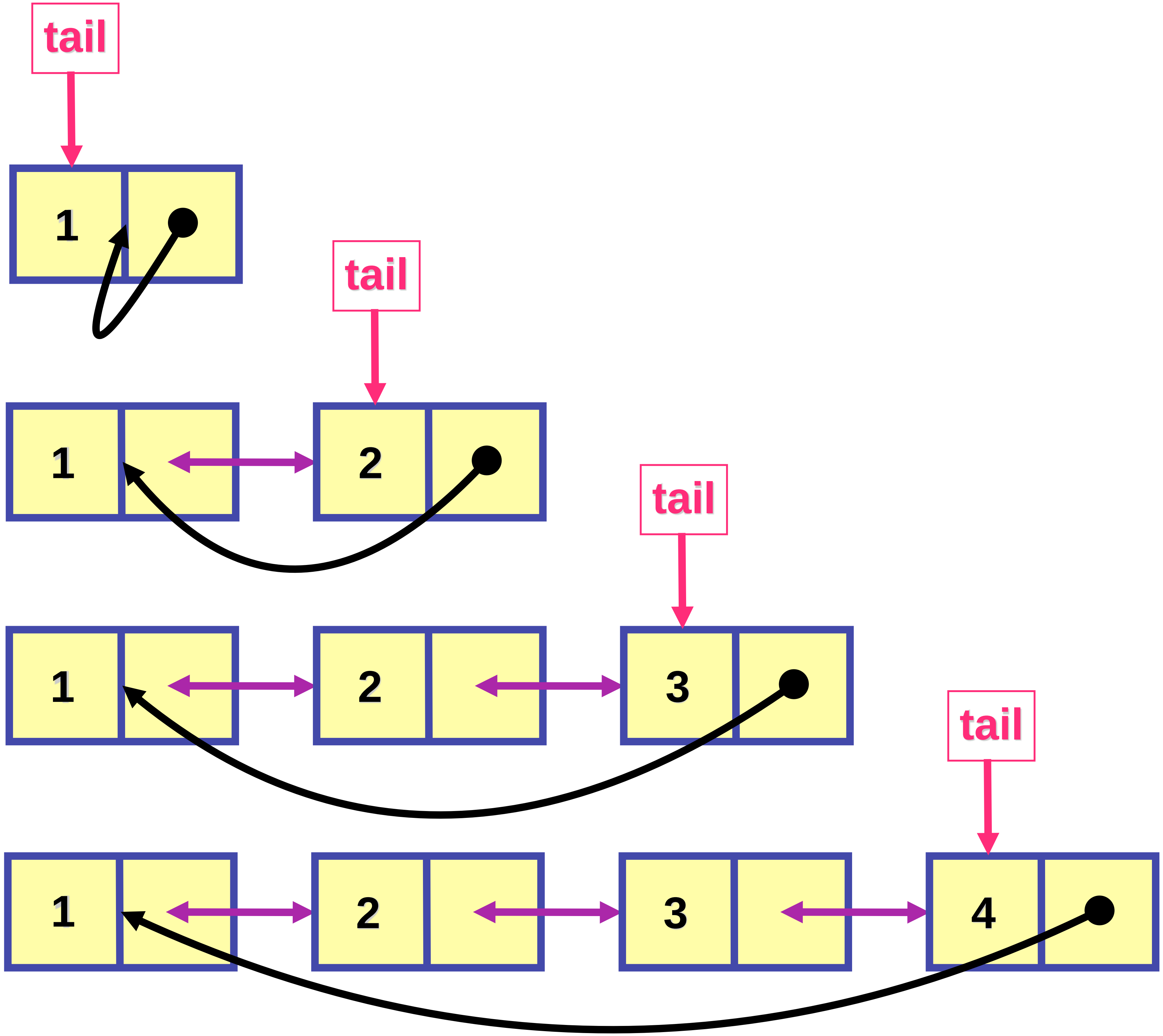
- **In an implementation of a circular singly linked list, we can use only one permanent pointer, tail, to the list even so operations on the list require access to the tail and its predecessor, the head,**

- **The next two slides show a sequence of insertions at the front and at the end of the circular list**

Inserting at the Front of a Circular SLL



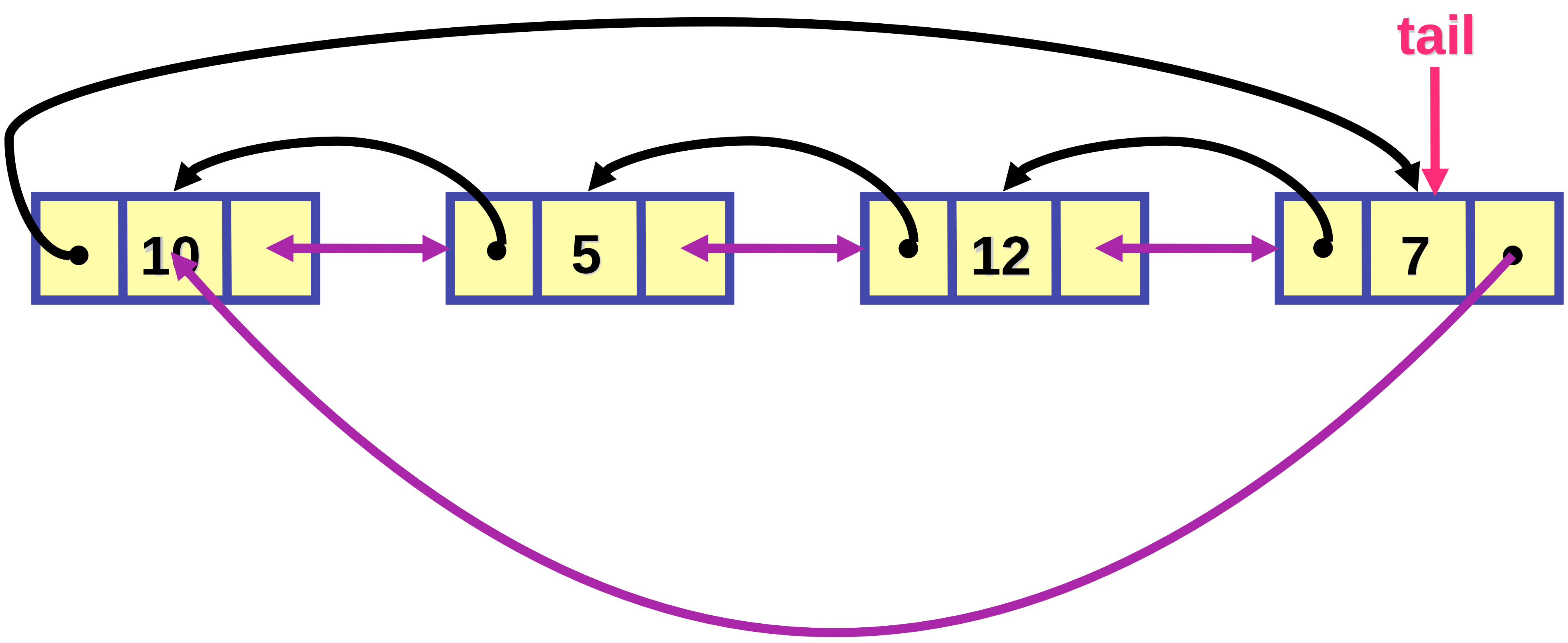
Inserting at the End of a Circular SLL



Circular Doubly Linked Lists

- The implementation of Circular SLL is not without problems,
- A member function for deleting of the tail node requires a loop so that tail can be set after delete to its predecessor,
- Moreover, processing data in the reverse order is not effective,
- To avoid these problems, A circular doubly linked list can be used

Circular Doubly Linked List



Applications: Sparse Tables

- In many real-world applications, the choice of a table seems to be the most natural one, but space consideration may preclude this choice,
- This is particularly true if only a small fraction of the table is actually used,
- A table of this type is called a sparse table since the table is populated sparsely by data and most of its cells are empty, so linked lists are better

Sparse Table Example

Students

1. Sheaver Geo
2. Weaver Henry
3. Shelton Mary

…

404. Crawford William
405. Lawson Earl

…

5206. Fulton Jenny
5207. Craft Donald
5208. Oates Key

…

Classes

1. Anatomy/Physiology
2. Microbiology

…

20 Advanced Writing
31 Chaucer

…

115 Data Structures
116 Cryptology
117 Computer Ethics

…

Grade Code

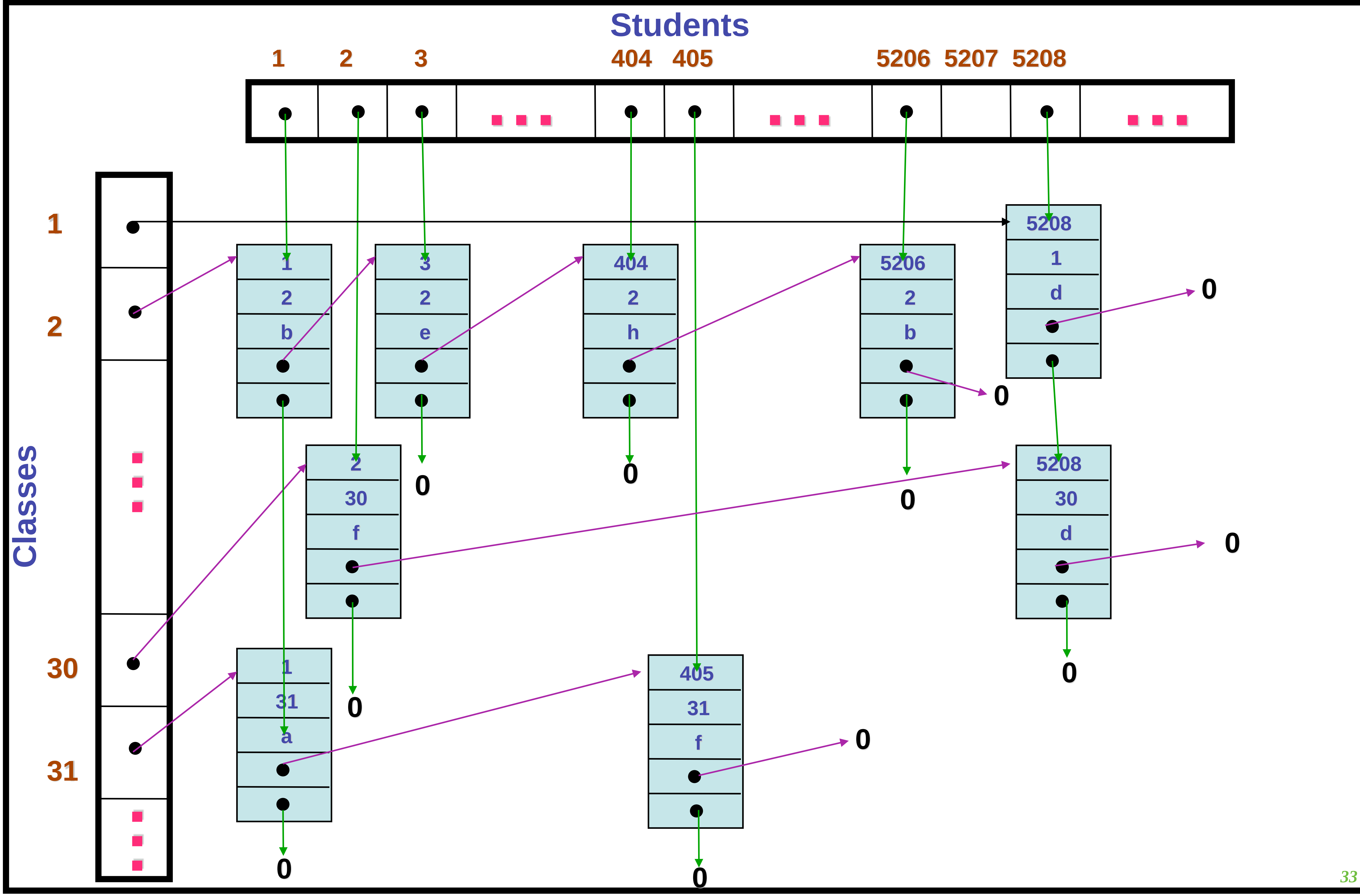
a	A
b	A-
c	B+
d	B
e	B-
f	C+
g	C
h	C-
i	D
j	F

Student

Class

	1	2	3	…	404	405	…	5206	5207	5208	…
1	8000									d	
2.	b		e		h			b			
…											
30.	f								d		
31.	a					f					
..											
200											

Implementation using Linked Lists



Other Applications

- **Various other applications presented and discussed in class including:**
  - **Dynamic Allocation Problems,**
  - **The File System in an Operating System,**
  - **Implementing the base class for other dynamic data structures, such as stacks, trees and graphs**

Thank You