IS2020 COMP 2540: Data Structures and Algorithms

Lecture 02: Linked List

Dr. Kalyani Selvarajah
kalyanis@uwindsor.ca

School of Computer Science
University of Windsor
Windsor, Ontario, Canada

May 27 & June 1, 2020

Use of a List: Motivation

- What are the advantages of arrays?
  - Arrays are simple and easy to use.
  - Faster access time to the element O(1).

- What are the disadvantages of arrays?
  - Pre-allocates memory up front (why is that bad thing?)
  - Fixed Size (Problem when inserting new element)
  - Contiguous memory allocation
  - Insertion and deletion are in general expensive.

Use of a List: Motivation

- C

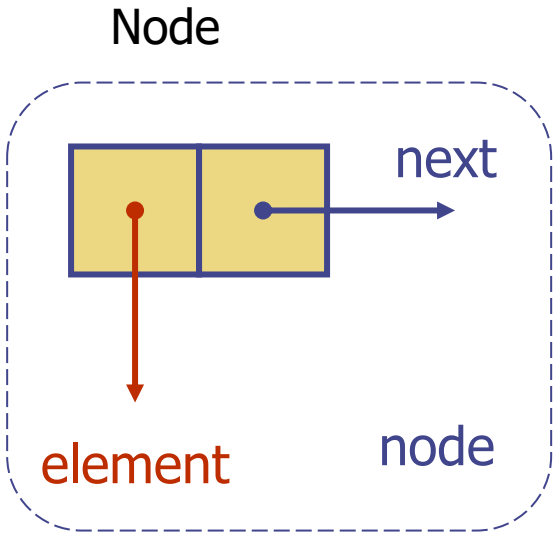
```
public class ArrayInsert {
    public static void main(String arg[]){
      int MAX=10;
      int [] my_array = new int[MAX];
        my_array[0]=10;
        my_array[1]=20;
        my_array[2]=30;
        my_array[3]=40;
        my_array[4]=50;
        my_array[5]=60;

// Insert an element in 3rd position of the array (index->2, value->5)
    int Index_position = 2;
    int newValue    = 5;

    System.out.println("Original Array : "+Arrays.toString(my_array));
    for(int i=my_array.length-1; i > Index_position; i--){
        my_array[i] = my_array[i-1];
    }
    my_array[Index_position] = newValue;
    //print the array after inserting an element
    System.out.println("New Array: "+Arrays.toString(my_array));

}
}
```

Use of a List: Motivation

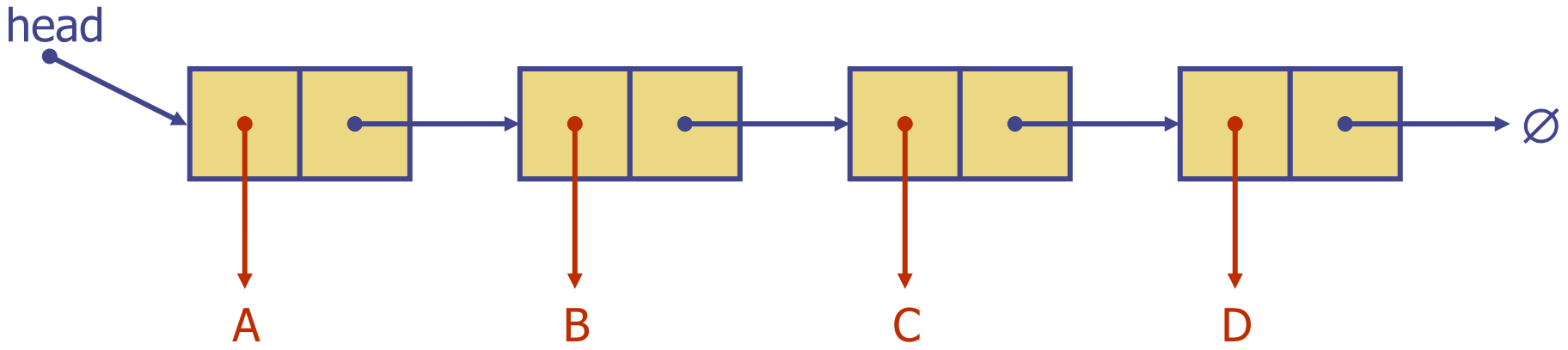
- Can we create dynamic arrays?
  - **No,** we need an efficient linear data structure.

Linked List

- **List** is one of the most basic types of data collection
  - For example, list of groceries, list of modules, list of friends, etc.
  - In general, we keep items of the same type (class) in one list



- **Definition:**
  - A collection of **nodes** where each node has an **element** and points to the **next** node in the list.



Linked List Properties

- Pointers connect successive elements.
- A Linked list has a **dynamic size**. It can grow or shrink during execution time.
- It can grow as much as we want or until the system run-out of memory.
- It does not waste memory space.
- The pointer of the last element in the list points to **null**
- It require a special pointer "**head**" that always point to the beginning of the list.

How a Linked List stored in memory?

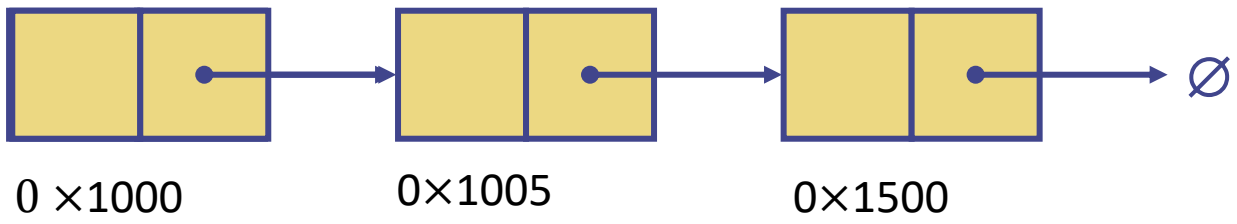
- Arrays in Memory: Continuous Allocation

Memory
0 ×1000
0 ×1001
0 ×1002
0×1003
0 ×1004

myArray

[0]	“A”
[1]	“P”
[2]	“P”
[3]	“L”
[4]	“E”

- Linked List in Memory: Random Allocation



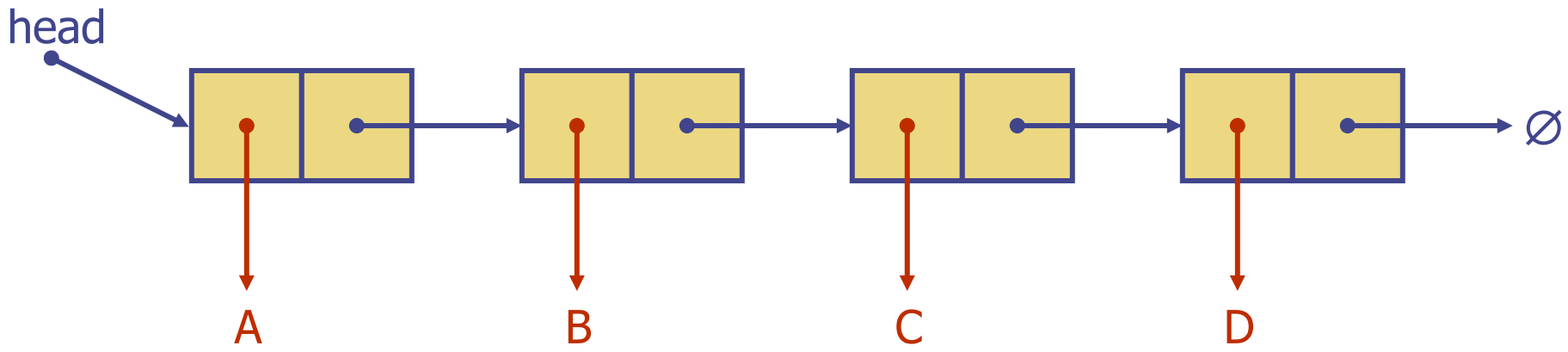
Linked List as ADT

- Linked List ADT = Linked List + Linked List Operations

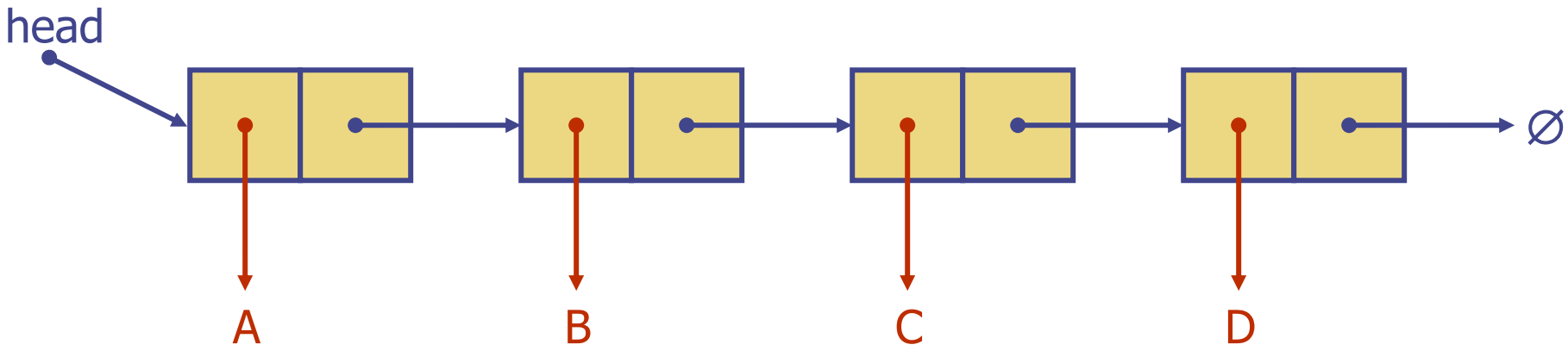
- Linked List Operations:
  - **Traverse**: visit each node (element) in the list in a sequential order.
  - **Insert**: add new node to the list
  - **Delete**: delete an existing node from the list
  - **Search:** find the index of a node with given data value
  - **Sort**: sort the nodes in the list based on some order
  - **Merge**: combine two or more list into one list

Singly Linked Lists

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer.



Singly Linked Lists




```
public class Node {
    Node next;  //pointer to the next node
    int data;   // value of the data

    //Constructor
    public Node(int data)
    {
        this.data=data;
    }
}
```

Linked List ADT Operations: Traversing

- Traversing a Linked List
  1. Start from the head pointer
  2. Follow the next pointers
  3. Process every node as required (e.g. print the node date, count the size of the node, find the node with maximum data value, etc)
  4. Stop when the pointer next points to NULL.

  The time complexity is O(n)

Singly Linked Lists

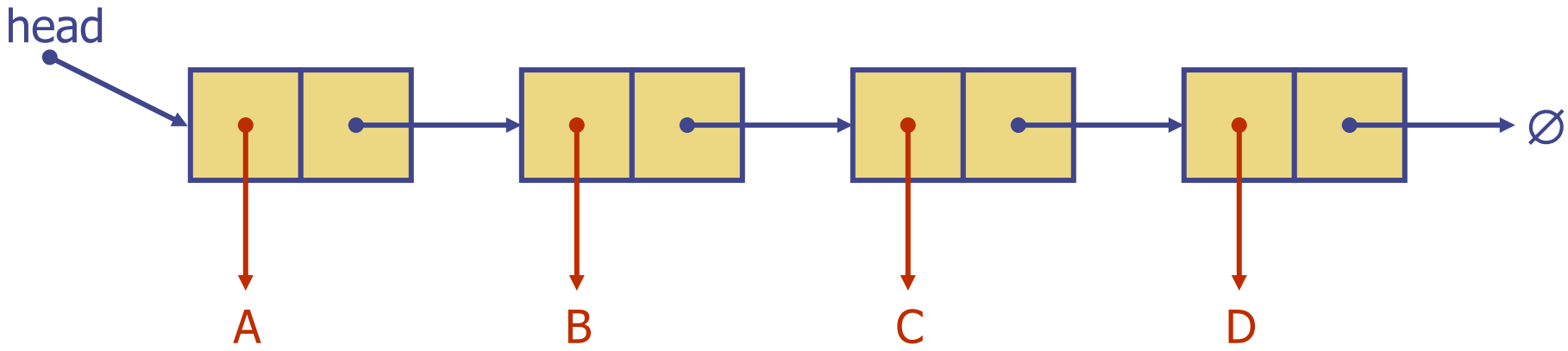
```
public class LinkedList {
        Node head; // head of the list - first element
        // Insert at the rear

        // print element in the linkedlist
        public void displayData()
        {
            Node current = head;
            while(current.next!=null)
            {
                System.out.println(current.data);
                current=current.next;
            }
            System.out.println(current.data);
        }
    }
```

Linked List ADT Operations: Inserting

- When we are inserting a node into a linked list we have three cases:
  1. Inserting a node at the beginning of the list
  2. Inserting a node at the end of the list
  3. Inserting a node at the middle of the list

Linked List ADT Operations: Inserting

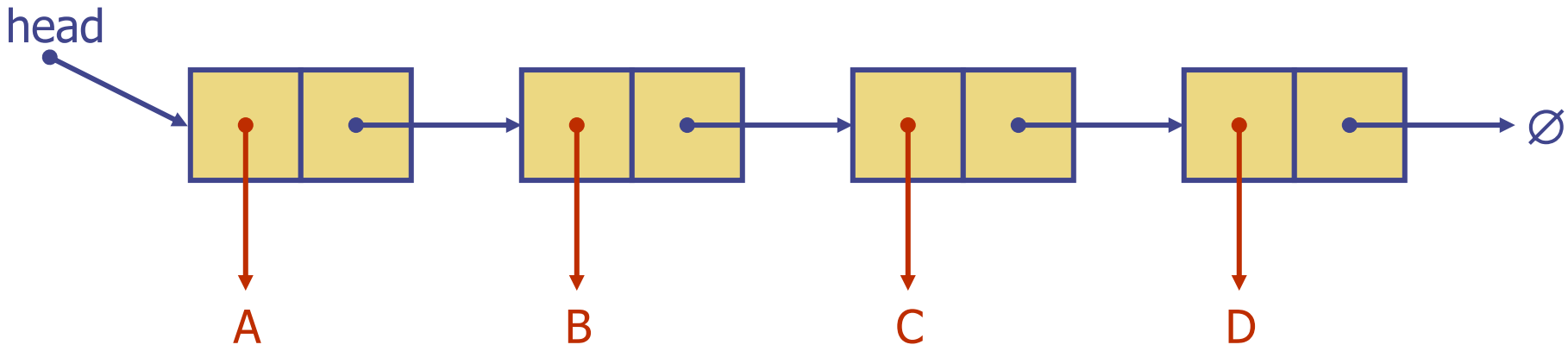
- CASE 1: Inserting at the beginning




```
public void insertFirst(int data)
{
     Node newHead=new Node(data);   // crate the new node
     newHead.next=head;          // assign the existing head at next
     head = newHead;            //  assign the new node at head
}
```

Linked List ADT Operations: Inserting

- CASE 2: Inserting a node at the end of the list




```
// Insert at the rear
public void insertData(int data){
    //check whether you are inserting to the head or node
    if(head==null) {
        head = new Node(data);
        return;
    }
    Node current = head;
    while(current.next!=null)    //check until last node
    {
        current=current.next;    // move pointer one by one
    }
    current.next=new Node(data); // when the pointer reaches the null pointer, insert the data.
}
```

Linked List ADT Operations: Inserting

- CASE 3: Inserting a node at the middle of the list

```
public void insertAt(int position,int data)
{
    if (head==null) return;
    Node newNode=new Node(data);  //create a new node
    Node previous=head;       // temporary node to point head
    int count=1;
    // find the position to insert data
    while(count<position-1)
    {
        previous=previous.next;
        count++;
    }

    Node current=previous.next;      // temporary node
    newNode.next=current;
    previous.next=newNode;
}
```

Linked List ADT Operations: Deleting

- When we are deleting a node into a linked list we have three cases:
  1. Delete from the beginning (first node)
  2. Delete from the end (delete the last node)
  3. Delete from the middle.

Linked List ADT Operations: Deleting

- CASE 1: Deleting at the beginning

Or

```
public void deleteFirst()
{
     head=head.next;
}
```

```
public void deleteFirst()
{
     Node current= head;
     head=head.next;
     current.next=null;
}
```

Linked List ADT Operations: Deleting

- CASE 2: Deleting at the end

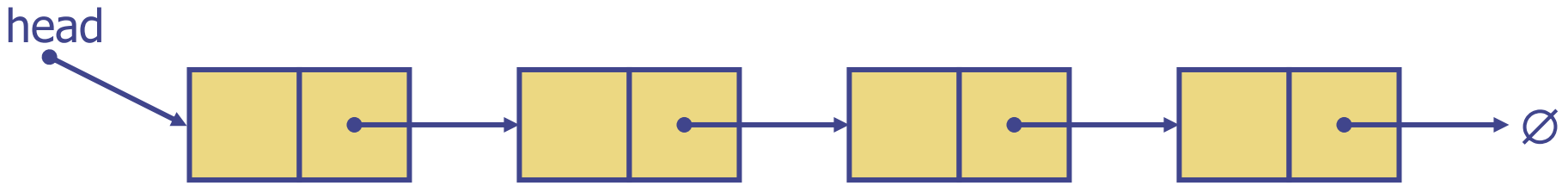
```
public void deleteData()
{
     Node current = head;
     Node previous=null;

     while(current.next!=null)
     {
          previous=current;
          current=current.next;
     }
     previous.next=null;

}
```

Linked List ADT Operations: Deleting

- CASE 3: Delete from the middle

```
public void deleteAt(int position)
{
    if (head==null) return;
    Node previous=head;       // temporary node to point head
    int count=1;
    // find the position to delete data
    while(count<position-1)
    {
        previous=previous.next;
        count++;
    }
    Node current=previous.next;      // temporary node
    previous.next=current.next;
    current.next=null;
}
```

Linked List ADT Operations: Searching




```
public boolean isContain(int data)
{
    Node current = head;
    while (current.next!=null)
    {
        if(current.data==data)
            return true;
        current=current.next;
    }
    return false;
}
```

Difference between Arrays and Linked List

Arrays	Linked List

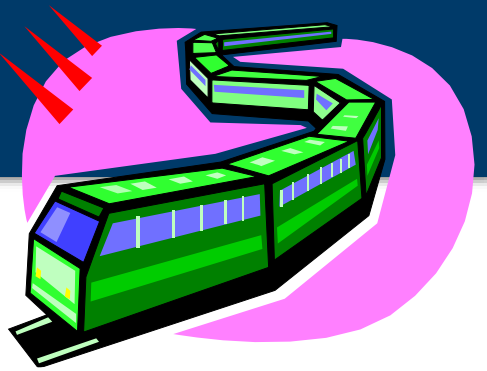
Time Complexity Linked List vs Array (Your turn!)

Operations	Arrays	Linked List (Singly)
Traversing		
Access an element		
Insert at the beginning		
Insert at the end		
Insert at the middle		
Delete at the beginning		
Delete at the end		
Delete at the middle		
Searching an element		

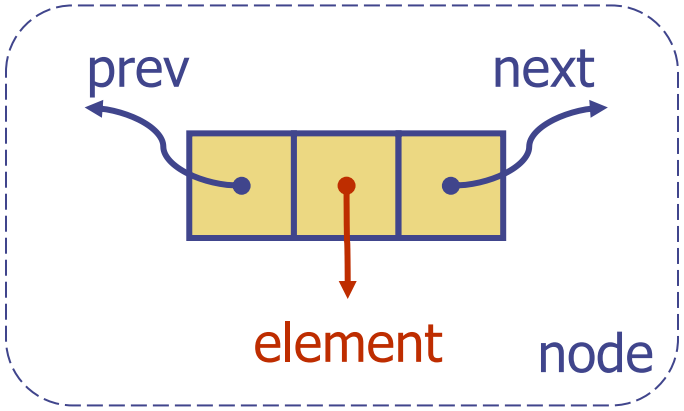
Other Variants

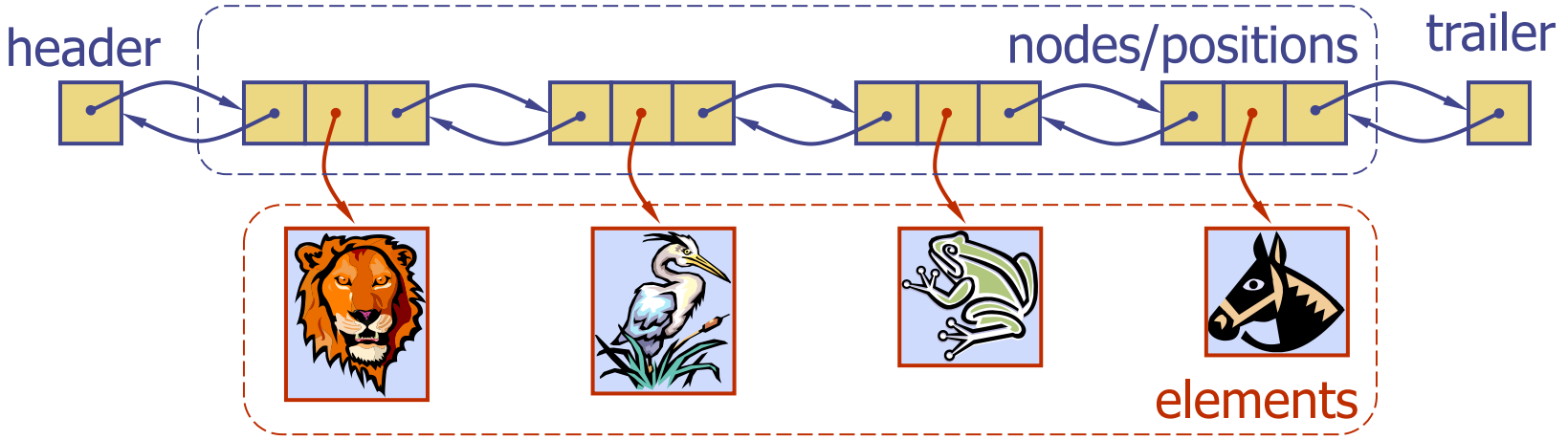
-Other Variants of Linked List

Doubly Linked List



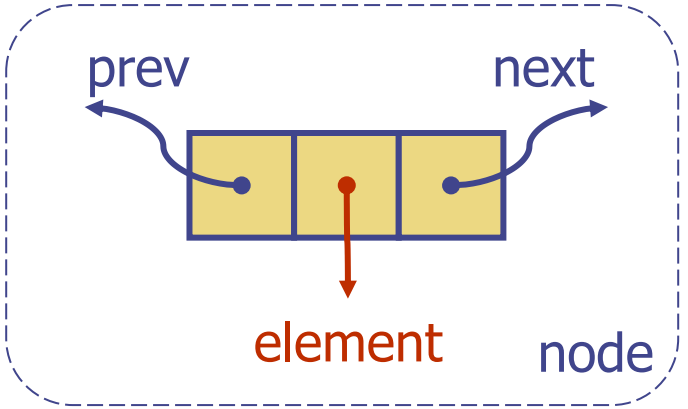
- A doubly linked list can be traversed forward and backward
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Doubly Linked List

```
public class Node {
        int data;
        Node previous;
        Node next;

        public Node(int data) {
            this.data = data;
        }
}
```

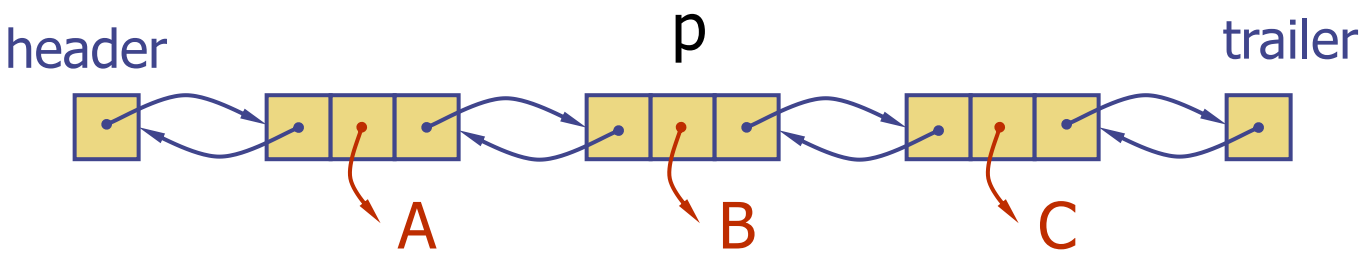


Advantages: From any node in the middle of the list we can navigate in any direction.

Disadvantages: Each node require extra memory space because the additional pointer. In addition, there are more pointer operations when inserting or deleting a node from the list

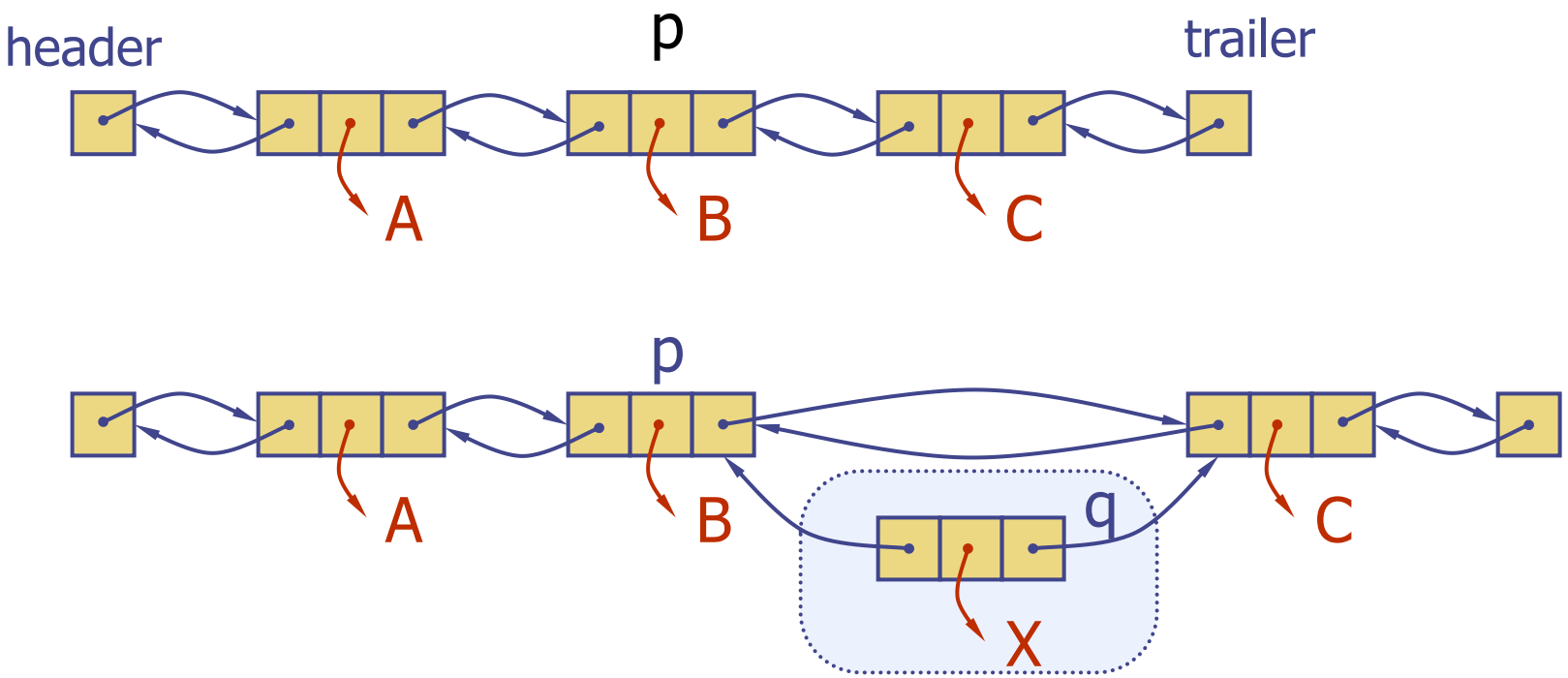
Doubly Linked List: Insertion (middle)

- Insert a node at a certain position. (let's say at p)



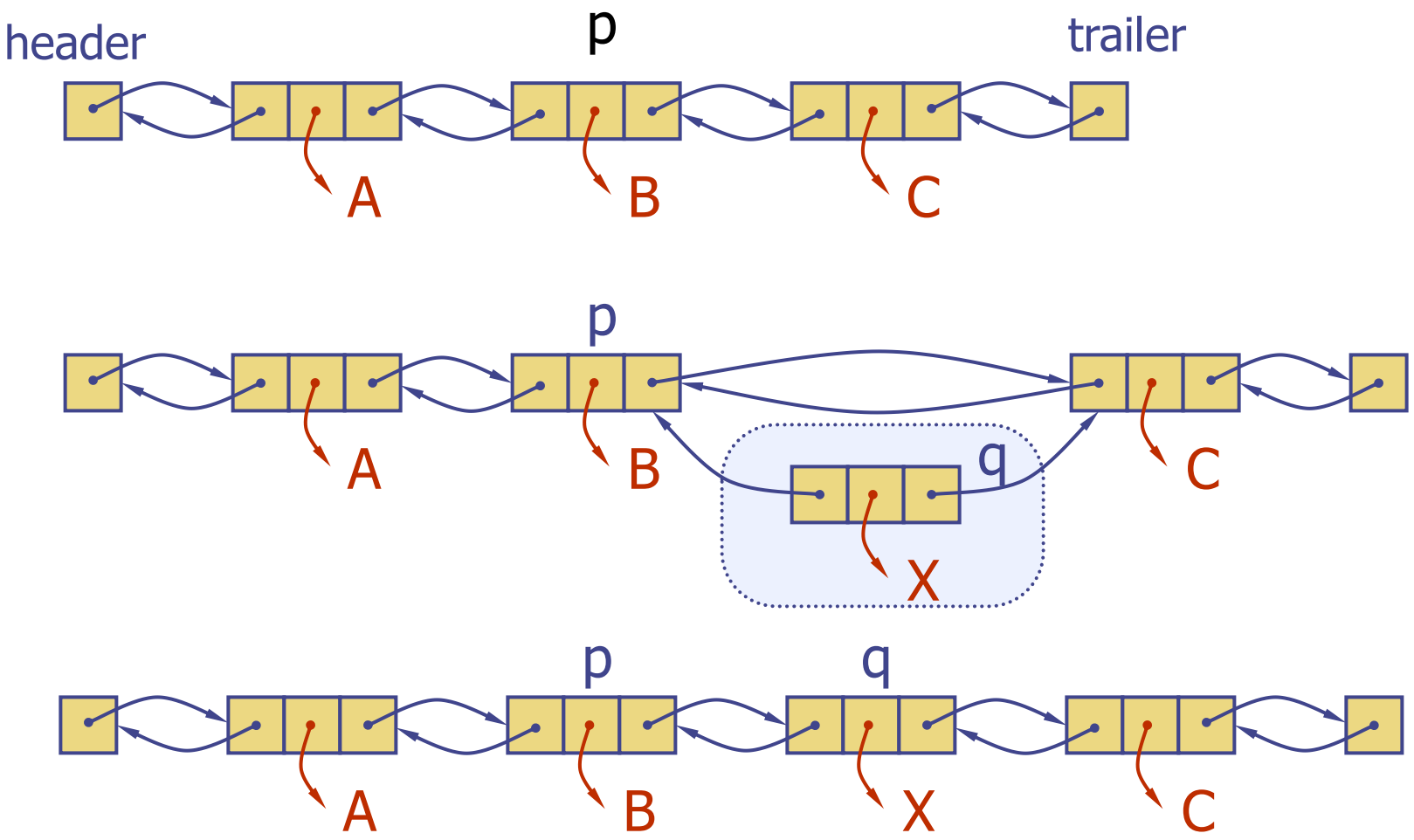
Doubly Linked List: Insertion (middle)

- Insert a node at a certain position. (let's say at p)



Doubly Linked List: Insertion (middle)

- Insert a node at a certain position. (let's say at p)



Doubly Linked List: Insertion at first

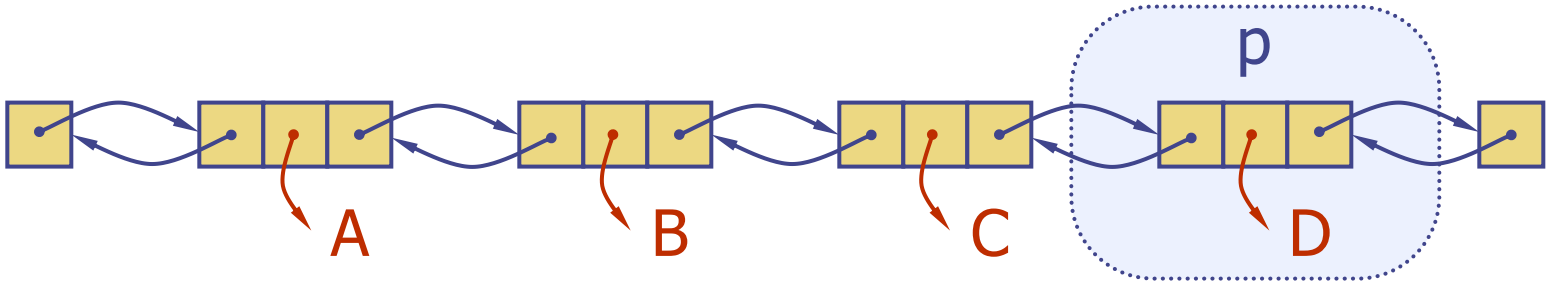
- Insert a node at first.

```
public void insertFirst(E data) {
    Node<E> newNode=new Node<E>(data);  //create a new node
    if(head==null)
        tail=newNode;              // in the empty list
    else
        head.previous=newNode;  // if the list is not empty
    newNode.next=head;
    head=newNode;
}
```

```
public class Main {
    public static void
main(String[] args)
    {
        DoublyLinkedList<Integer>
dblist=new
DoublyLinkedList<Integer>();

System.out.println(dblist.isEmpty(
));

        dblist.insertFirst(20);
        dblist.insertFirst(15);
        dblist.insertFirst(30);
        dblist.insertFirst(10);

        dblist.displayList();

    }
}
```

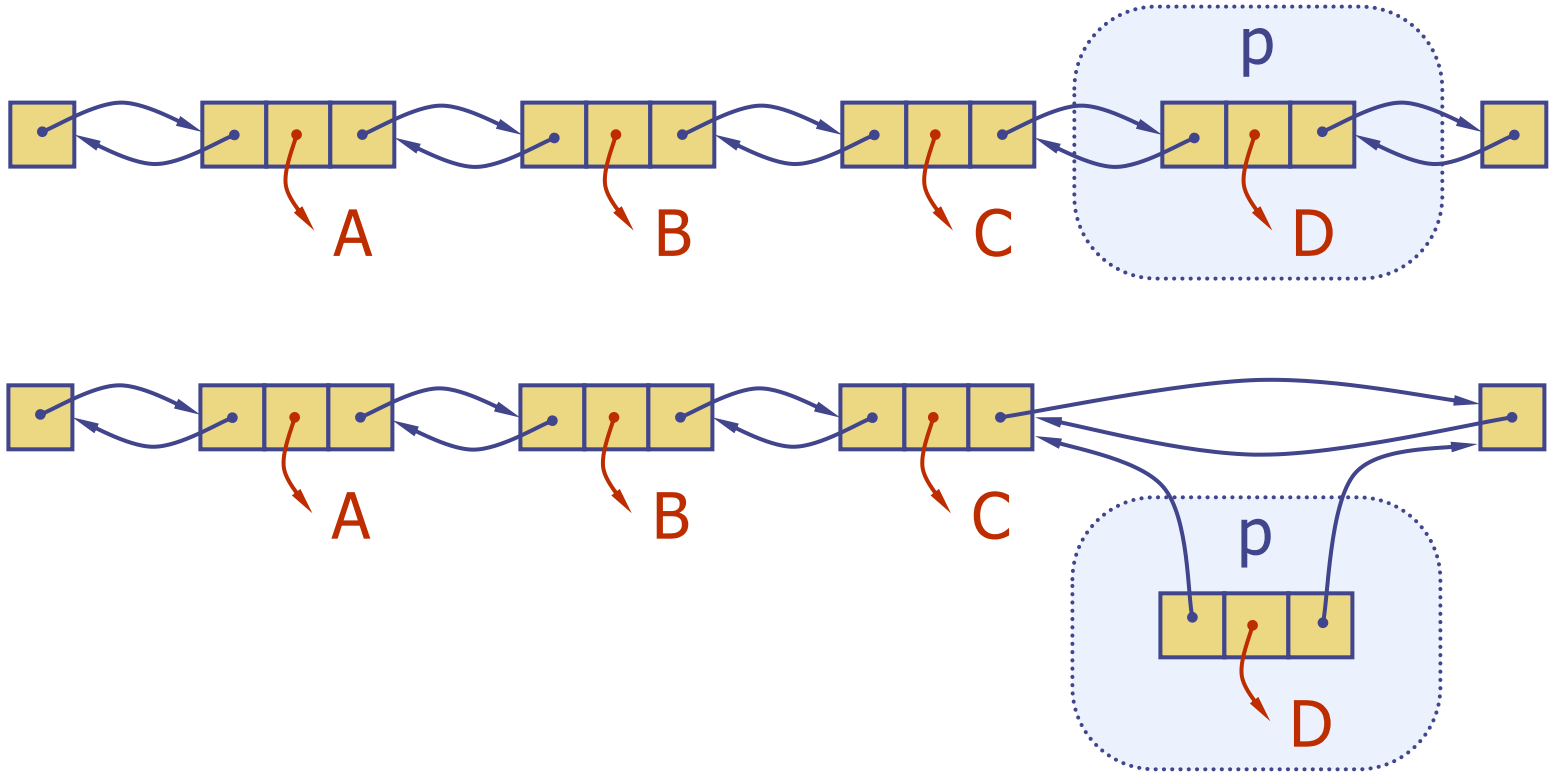
Doubly Linked List: Deletion

- Remove a node, p, from a doubly linked list.



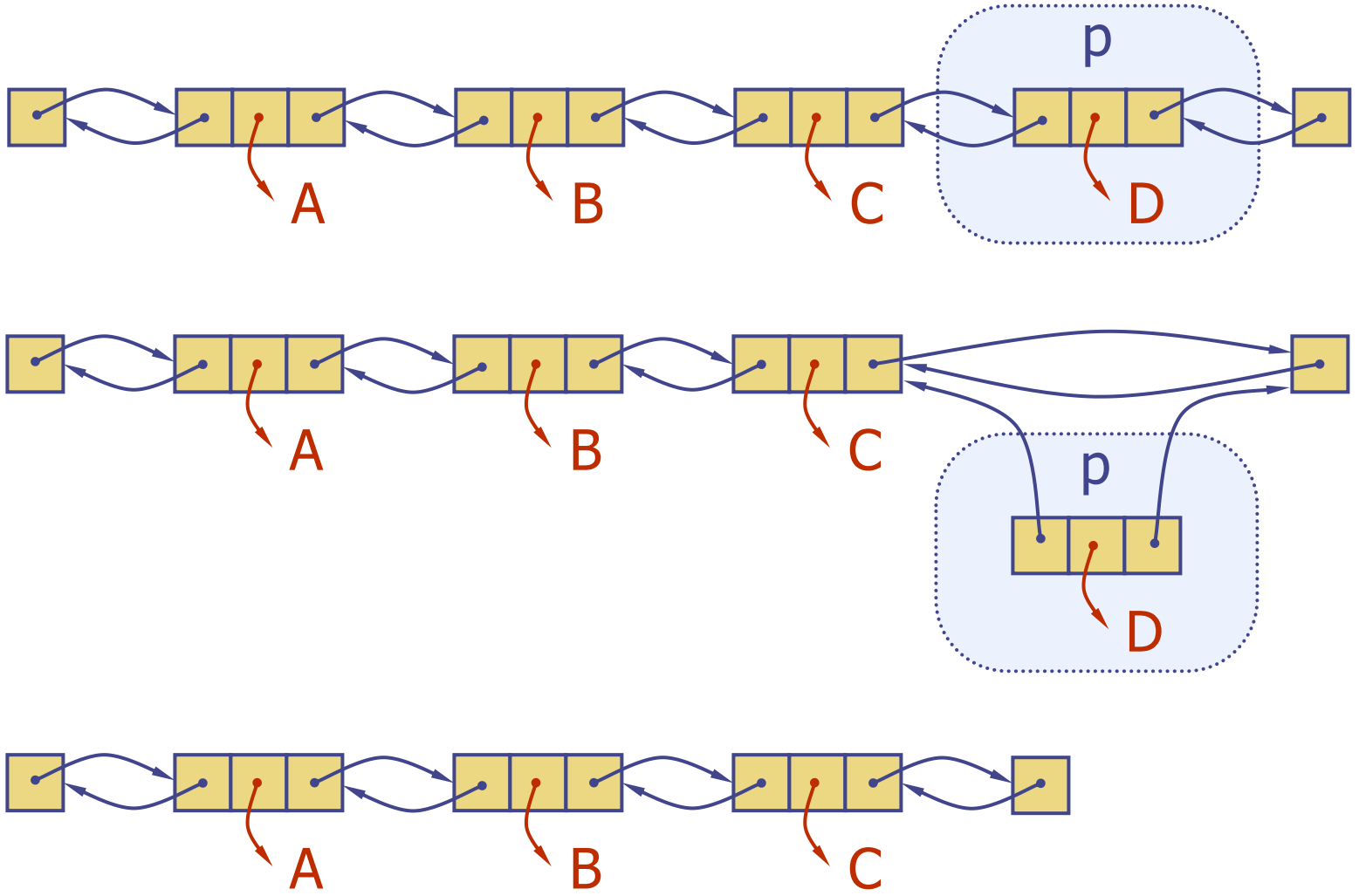
Doubly Linked List: Deletion

- Remove a node, p, from a doubly linked list.



Doubly Linked List: Deletion

- Remove a node, p, from a doubly linked list.



Doubly Linked List: Deletion at first

```
 public void deleteFirst()
    {
        if(head==null)
            return;

        Node<E> current = head;
//          head = head.next;
//          head.previous = null;

        head.next.previous=null;
        head=head.next;
        length--;
    }
```

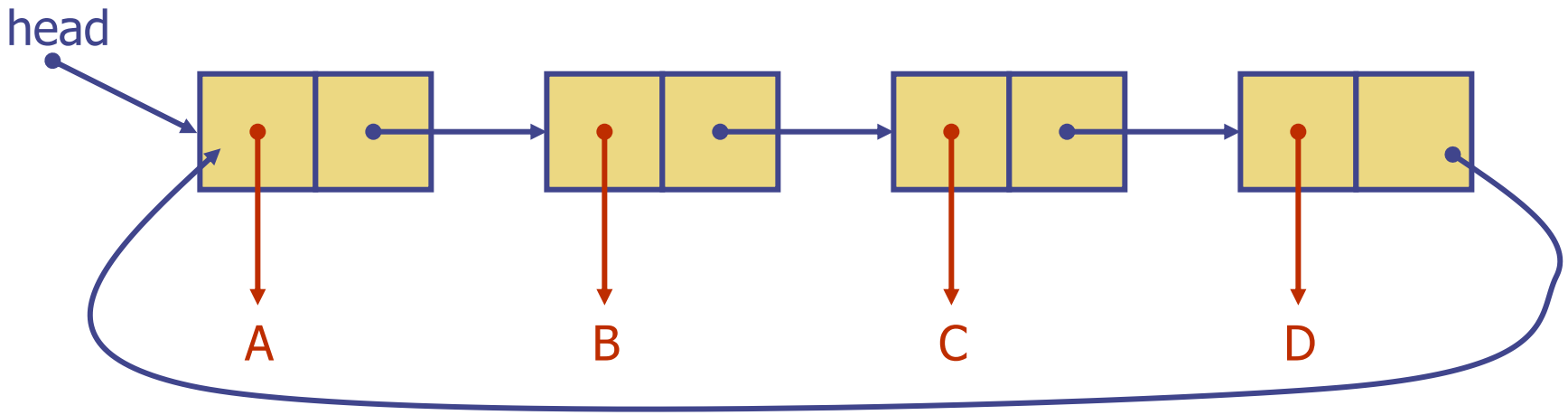
In class assessment (Work after lecture) (1/2)

- Implement the following operation for doubly Linked List:

```
public void insertAt(int position, E data){

}
public void insertEnd(E data){

}
public void deleteAt(int position) {

}
public void deleteEnd(){

}
```

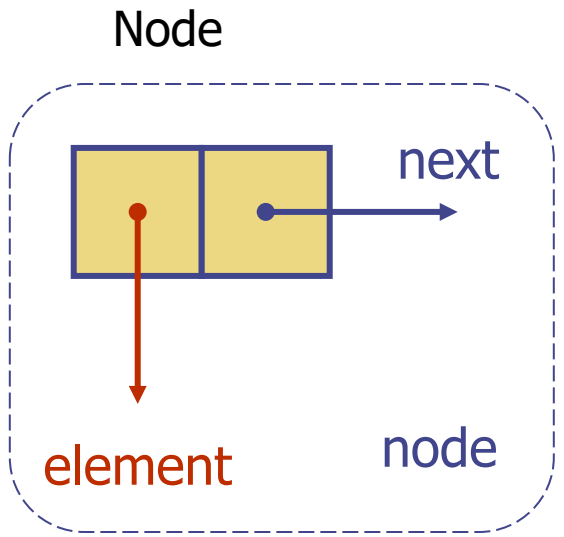
Circular (Singly) Linked List

A linked list where each node contains a pointer (next) pointing to the next node in the list. The pointer of the last node in the list point to the head node (first node in the list)



Circular (Singly) Linked List

```
public class Node {
    Node next;  //pointer to the next node
    int data;   // value of the data

    //Constructor
    public Node(int data)
    {
        this.data=data;
    }
}
```



Circular (Singly) Linked List: Traversing

```
// print element in the circular linkedlist
public void displayData()
{
    Node current = head;
    while(current.next!=head)
    {
        System.out.print(current.data+" --> ");
        current=current.next;
    }
    System.out.println(current.data);
}
```

Circular (Singly) Linked List: Inserting

```
public void insertEnd(int data) {
    Node newNode = new Node(data);

    if (head == null) {
        head = newNode;
    } else {
        tail.next = newNode;
    }
    tail = newNode;
    tail.next = head;
}
```

In class assessment (Work after lecture) (2/2)

- Implement the following operation for Circular Linked List:

```
public void insertAt(int position, E data){

}
public void insertFirst(E data){

}
public void deleteAt(int position) {

}
public void deleteEnd(){

}
```

Any Question

Sorting

- Many different way to sort.
  - You already learn some of them in COMP 141
  - We will learn in the future too.
  - You can use any sorting method that you are familiar with for the assignment. You can't use any in build functions.

Sorting in Array (Simple example)

```
import java.util.Arrays;
public class ArraysSorting {
    public static void main(String[] args) {
        int[] myArray = {4, 2, 1, 3, 5, 9, 6, 8, 7};
        System.out.println("Original Array:" +
Arrays.toString(myArray));

        for (int i = 0; i < myArray.length; i++) {
            for (int j = i + 1; j < myArray.length; j++) {
                if (myArray[i] > myArray[j]) {
                    int temp = myArray[i];
                    myArray[i] = myArray[j];
                    myArray[j] = temp;
                }
            }
        }
        System.out.println("After Sorting: "+Arrays.toString(myArray));
    }
}
```

Sorting

- Many different way to sort.
  - You already learn some of them in COMP 141
  - We will learn in Data Structures and Algorithms lectures in the future.
  - You can use any sorting method that you are familiar with for the assignment. You can't use any in build functions.

Sorting in linked List
(Simple example)

```
public void sortlist()
{
    if(length>0) {    // check whether the list has elements or not
        for (int i = 0; i < length; i++) {
            Node current = head;
            Node nextnode = head.next;
            int temp = 0;
            for (int j = 1; j < length; j++) {
                if (current.data > nextnode.data) {
                    temp = current.data;
                    current.data = nextnode.data;
                    nextnode.data = temp;
                }
                current = nextnode;
                nextnode = nextnode.next;
            }
        }
    }
}
```