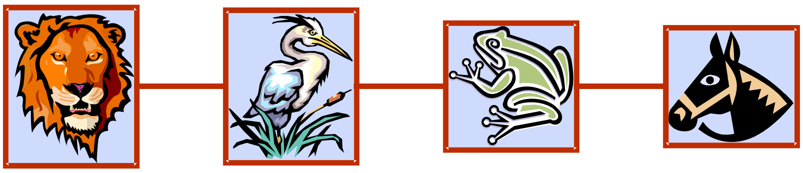
# Linked Lists

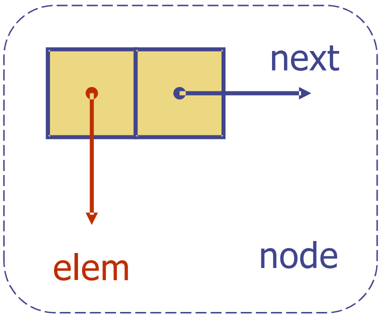


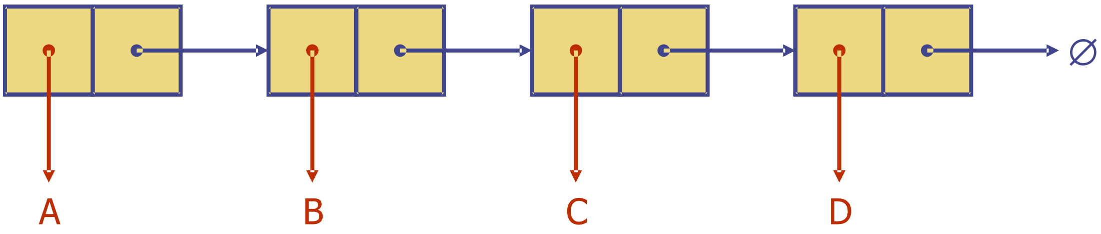
# List

- A list is an ordered collection of elements
  ➔ a sequence of elements
- By order, we mean "before and after" relationships between adjacent elements.
- Order is determined by the concept of "Positions".
- Array is one of possible implementations of List.

# Singly Linked List

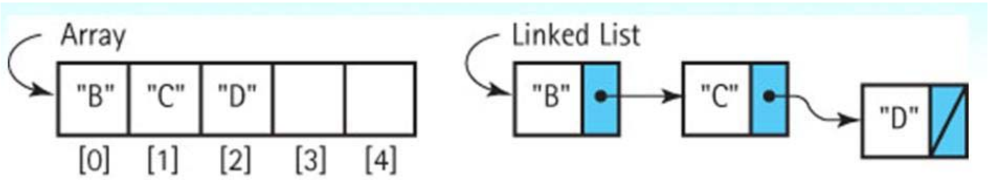
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node



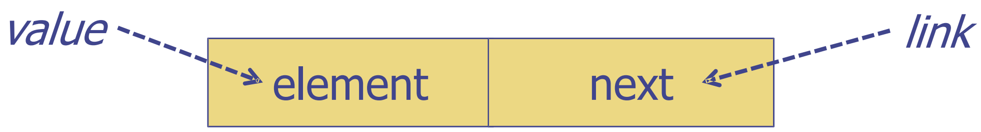


# Arrays *vs.* Linked Lists

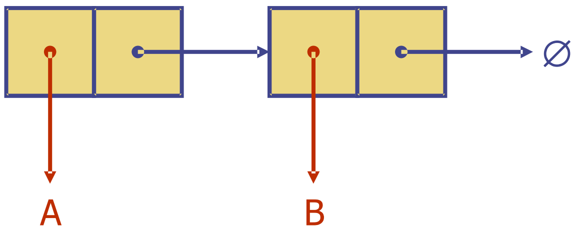
- In array, elements occupy contiguous memory locations.
- In linked list, elements occupy non-contiguous locations.



# Self-Reference Class



```
public class Node<E> {
   private  E element;
   private  Node<E> next;
   …
}
```



# Self-Reference Class

- To define a node in Java, define a ***self-reference*** class:

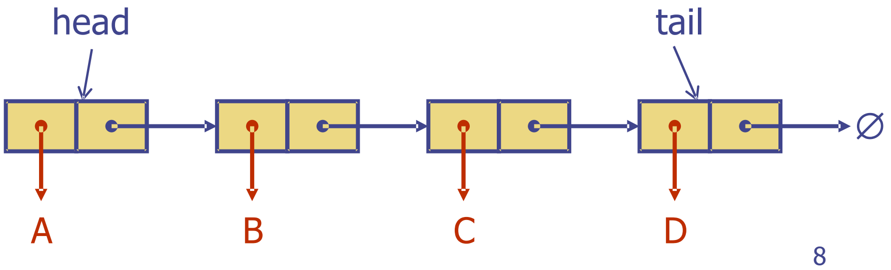
```
public class Node<E> {
   private  E element;
   private  Node<E> next;

  /** 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(E e,  Node<E> n)  {
        element  =  e;
        next  =  n;
  }
```

# The Node Class for List Nodes

```
    // Accessor methods:
     public E getElement() {
        return element;
     }
     public Node<E> getNext() {
        return next;
     }
     // Modifier methods:
     public void setElement(E newElem) {
          element = newElem;
     }
     public void setNext(Node<E> newNext) {
          next = newNext;
     }
}
```

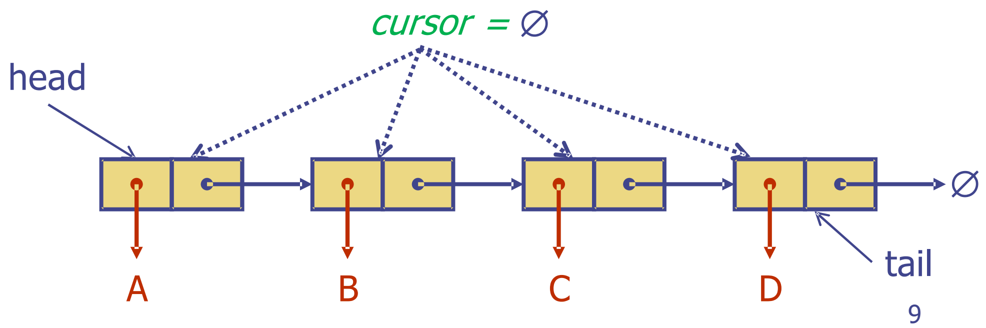
# Implementation of SLL

```
public class SLinkedList<E> {
   protected Node<E> head;  // head node of the list
   protected Node<E> tail;  // last node of the list (opt.)
   protected long size;     // # of nodes in the list (opt.)

   /** Default ctor that creates an empty list */
   public SLinkedList() {
      head = tail = null;
      size = 0;
   }
   …
}
```

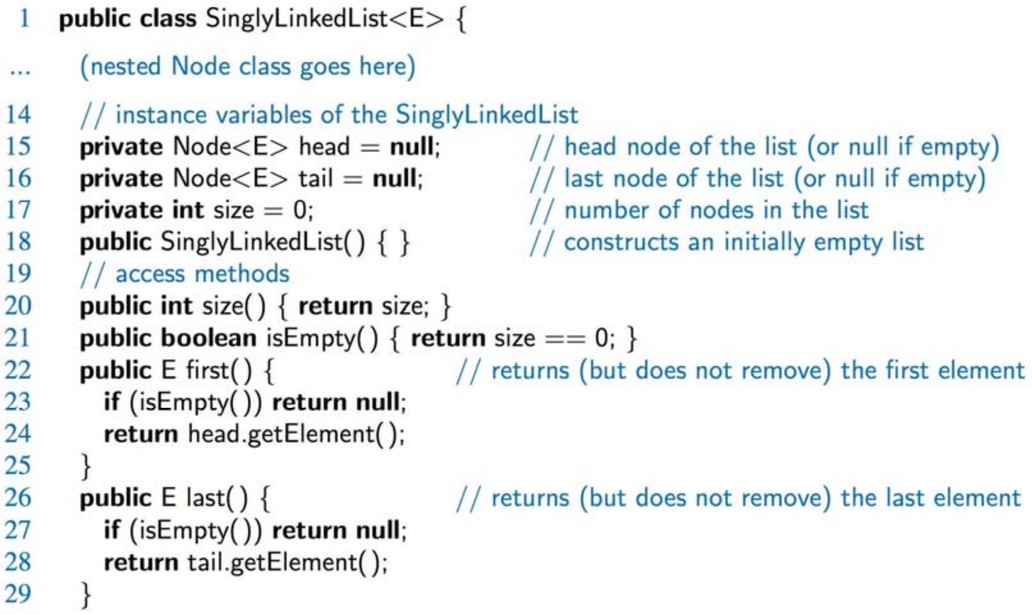


# Link Traversal

```
Void printAll() {
    Node<T> cursor = head;
    while (curor != null) {
        // do something …
        // ➔ System.out.println(cursor.getElement())
        cursor = cursor.getNext();
    }
}
```

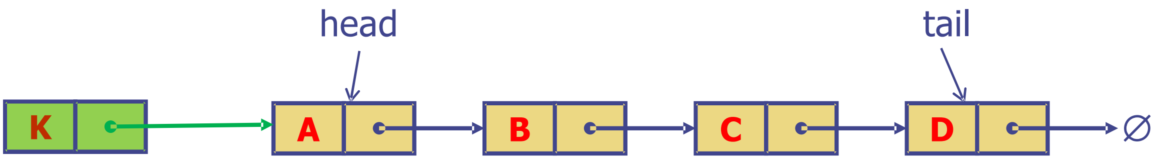


# Accessor Methods



```
public class SinglyLinkedList<E> {
...  (nested Node class goes here)
  // instance variables of the SinglyLinkedList
  private Node<E> head = null;              // head node of the list (or null if empty)
  private Node<E> tail = null;              // last node of the list (or null if empty)
  private int size = 0;                     // number of nodes in the list
  public SinglyLinkedList( ) { }            // constructs an initially empty list
  // access methods
  public int size( ) { return size; }
  public boolean isEmpty( ) { return size == 0; }
  public E first( ) {                  // returns (but does not remove) the first element
    if (isEmpty( )) return null;
    return head.getElement( );
  }
  public E last( ) {                   // returns (but does not remove) the last element
    if (isEmpty( )) return null;
    return tail.getElement( );
  }
```

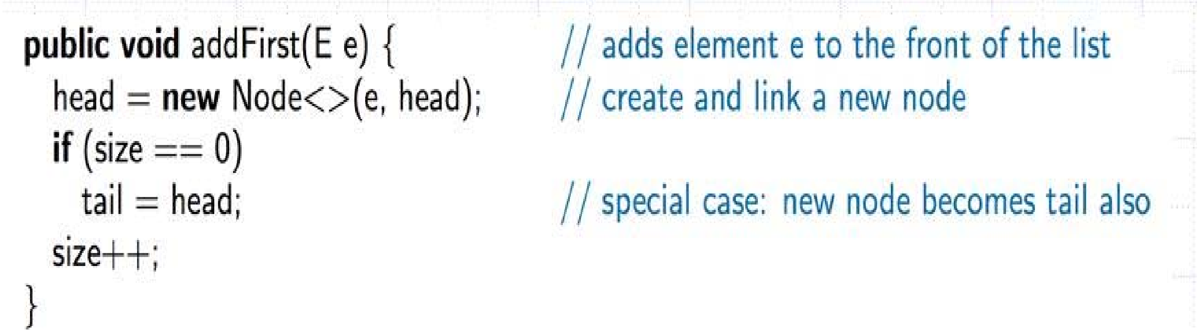
# Inserting at the Head



① Allocate new node
② Insert new element
③ Have new node point to old head
④ Update head to point to new node
⑤ Increment size by one

# Inserting at the Head

## addFirst(*E e*)



```
public void addFirst(E e) {            // adds element e to the front of the list
  head = new Node<>(e, head);          // create and link a new node
  if (size == 0)
    tail = head;                       // special case: new node becomes tail also
  size++;
}
```

```
public void  addFirst(E e) {
  Node<E> n = new Node<>();
  n.setElement(e);
  n.setNext(head);
  head = n
  if (size = 0) tail = head
  size++
}
```

① Allocate new node
② Insert new element
③ Have new node point to old head
④ Update head to point to new node
⑤ Increment size by one

# Inserting at the Tail

① Allocate a new node

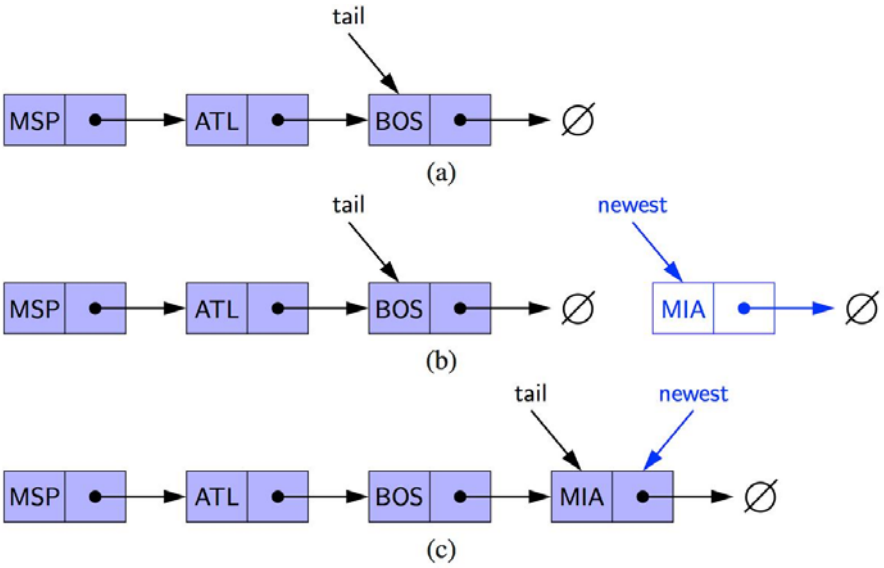
② Insert new element

③ Have new node point to null

④ Have old last node point to new node

⑤ Update tail to point to new node

⑥ Increment size



(a)

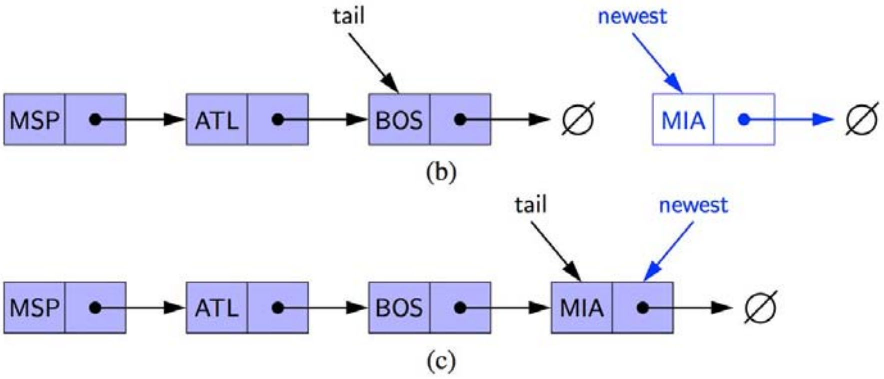
(b)

(c)

*Be careful about the order of step ④ and ⑤*

# Inserting at the Tail

```
public void addLast(E e) {                    // adds element e to the end of the list
  Node<E> newest = new Node<>(e, null);    // node will eventually be the tail
  if (isEmpty())
    head = newest;                            // special case: previously empty list
  else
    tail.setNext(newest);                     // new node after existing tail
  tail = newest;                              // new node becomes the tail
  size++;
}
```

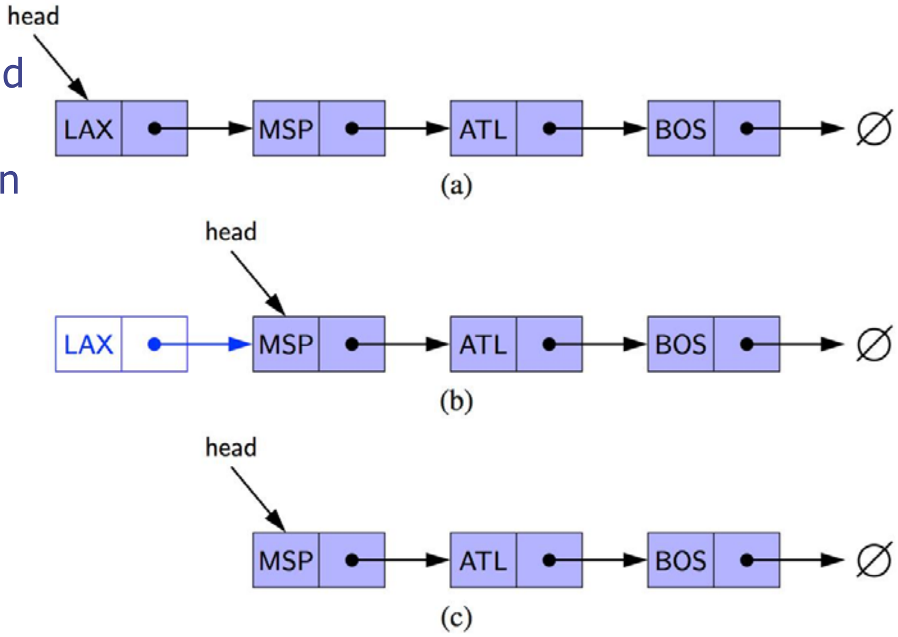


(b)

(c)

# Removing at the Head

- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node



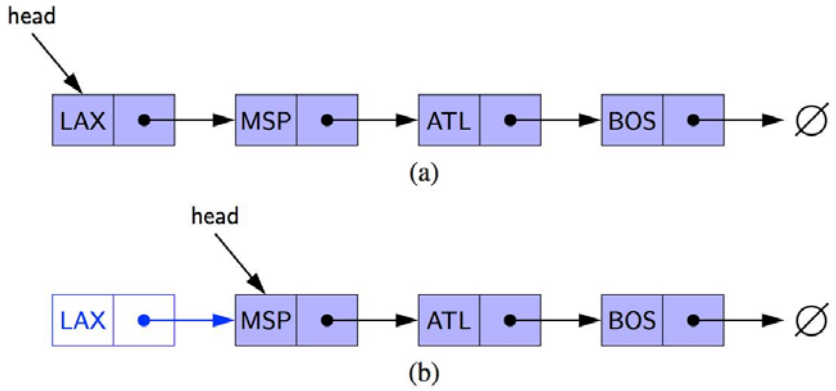
(a)

(b)

(c)

# Java Method

```
  public E removeFirst( ) {                    // removes and returns the first element
    if (isEmpty( )) return null;               // nothing to remove
    E answer = head.getElement( );
    head = head.getNext( );                    // will become null if list had only one node
    size−−;
    if (size == 0)
      tail = null;                             // special case as list is now empty
    return answer;
  }
}
```

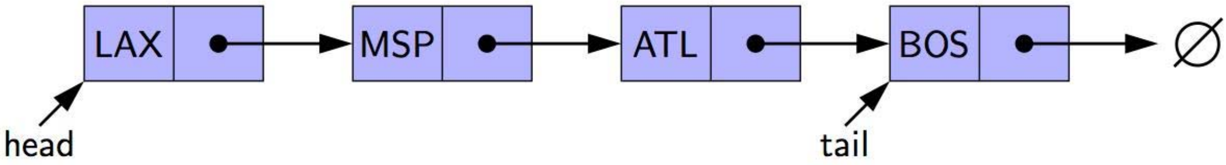


(a)

(b)

# Removing at the Tail

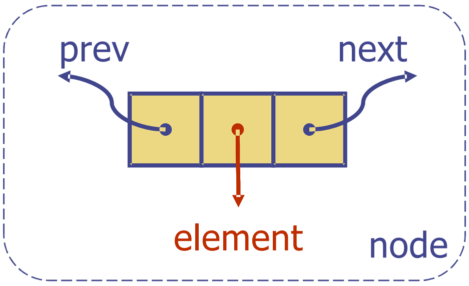
- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node

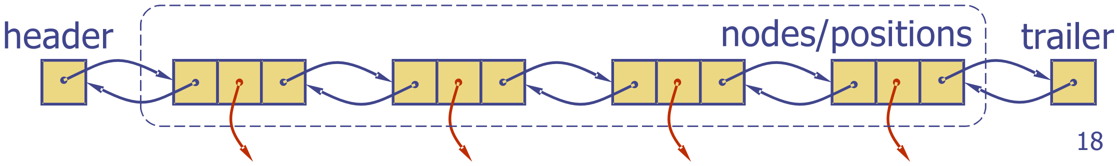


- It is time consuming to remove any node other than the head in a singly linked list. ➔ ***Any improvements?***
- How about insertion or deletion in the middle of the list?
  - ➔ Try for yourself!

# Doubly Linked List

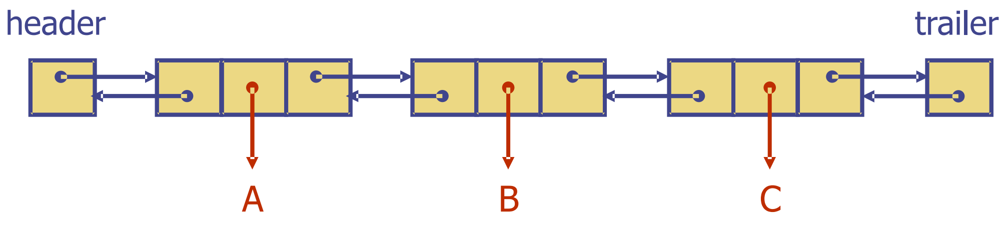
- The doubly linked list allows us to go in both directions – forward and reverse – in the list.
- Supports quick insertion and removal at both ends and in the middle.
- Each node stores
  - element
  - link to the next node
  - link to the previous node
- Special trailer and header nodes



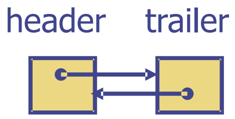


# Header and Trailer Sentinels

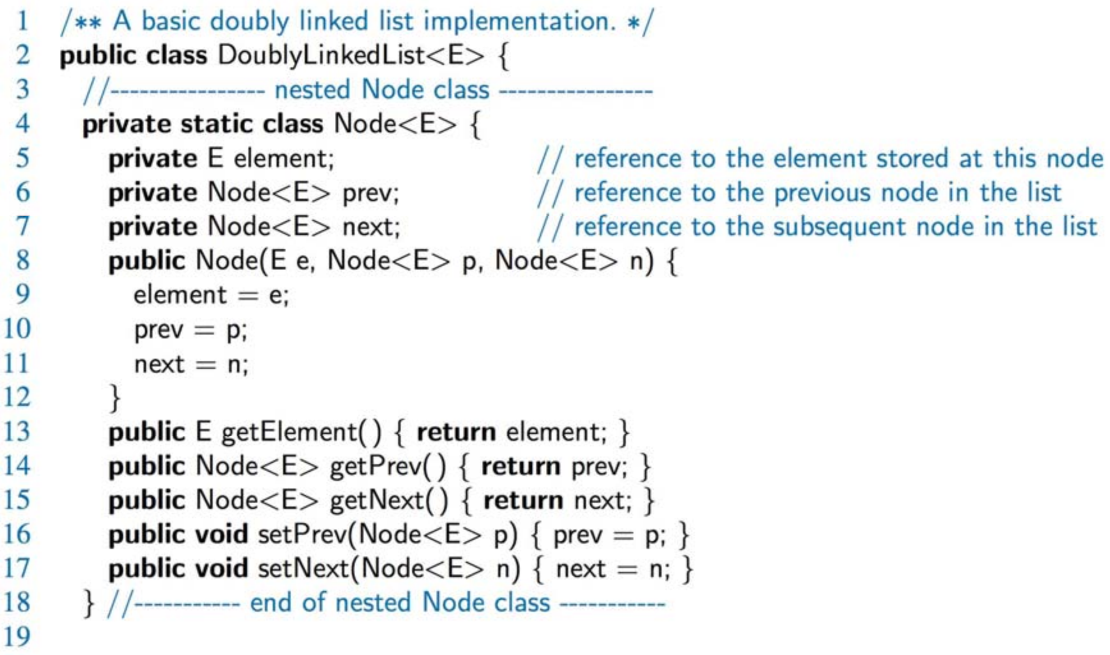
- To simplify programming, add special nodes at both ends of a DLL
  - ***Header*** node and ***trailer*** node



Empty list



# Doubly-Linked List in Java



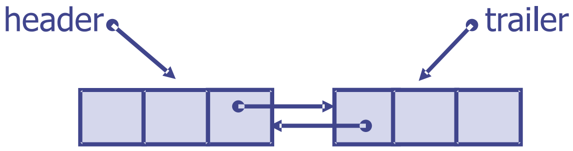
```
/** A basic doubly linked list implementation. */
public class DoublyLinkedList<E> {
  //---------------- nested Node class ----------------
  private static class Node<E> {
    private E element;               // reference to the element stored at this node
    private Node<E> prev;            // reference to the previous node in the list
    private Node<E> next;            // reference to the subsequent node in the list
    public Node(E e, Node<E> p, Node<E> n) {
      element = e;
      prev = p;
      next = n;
    }
    public E getElement( ) { return element; }
    public Node<E> getPrev( ) { return prev; }
    public Node<E> getNext( ) { return next; }
    public void setPrev(Node<E> p) { prev = p; }
    public void setNext(Node<E> n) { next = n; }
  } //----------- end of nested Node class -----------

```

# Doubly-Linked List in Java, 2

```
private Node<E> header;                         // header sentinel
private Node<E> trailer;                        // trailer sentinel
private int size = 0;                           // number of elements in the list
/** Constructs a new empty list. */
public DoublyLinkedList( ) {
  header = new Node<>(null, null, null);        // create header
  trailer = new Node<>(null, header, null);     // trailer is preceded by header
  header.setNext(trailer);                      // header is followed by trailer
}
/** Returns the number of elements in the linked list. */
public int size( ) { return size; }
/** Tests whether the linked list is empty. */
public boolean isEmpty( ) { return size == 0; }
/** Returns (but does not remove) the first element of the list. */
public E first( ) {
  if (isEmpty( )) return null;
  return header.getNext( ).getElement( );       // first element is beyond header
}
/** Returns (but does not remove) the last element of the list. */
public E last( ) {
  if (isEmpty( )) return null;
  return trailer.getPrev( ).getElement( );      // last element is before trailer
}
```

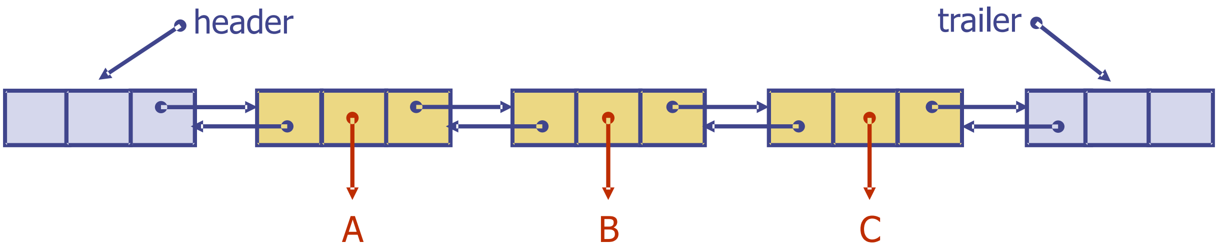
# Ctor of DLL

```
/** Constructs a new empty list. */
public DoublyLinkedList( ) {
  header = new Node<>(null, null, null);
  trailer = new Node<>(null, header, null);
  header.setNext(trailer);
}
```

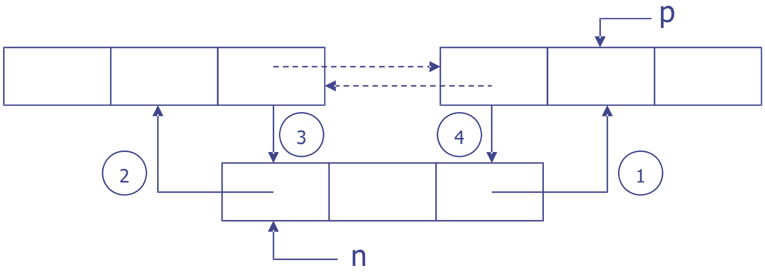


# fisrt() and last()

```
public E first( ) {
  if (isEmpty( )) return null;
  return header.getNext( ).getElement( );          // first element is beyond header
}
/** Returns (but does not remove) the last element of the list. */
public E last( ) {
  if (isEmpty( )) return null;
  return trailer.getPrev( ).getElement( );         // last element is before trailer
}
```

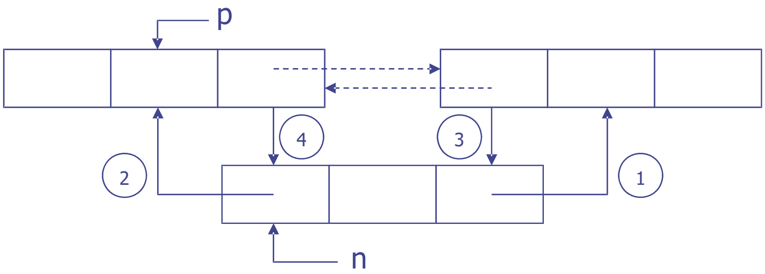


# Insertion before a Node



```
// Add new node n before node p
Algorithm addBefore(Node p, Node n)
   n.setNext(p);
   n.setPrev(p.getPrev());
   p.getPrev().setNext(n);
   p.setPrev(n);
   size ← size + 1;
```

# Insertion after a Node



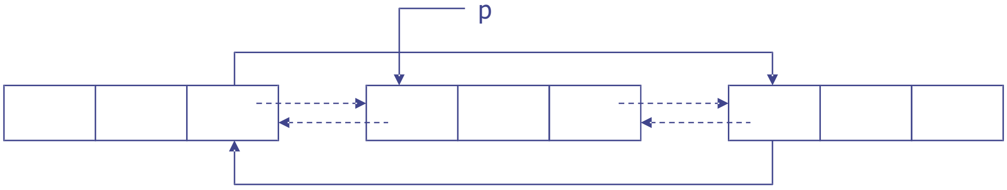
```
// Add new node n After node p
Algorithm addAfter(Node p, Node n)
    n.setNext(p.getNext());
    n.setPrev(p);
    p.getNext().setPrev(n);
    p.setNext(n);
    size ← size + 1;
```

# Insertion at the Front or Tail

```
Algorithm addFirst(DNode n)
    addAfter(header, n);
```

```
Algorithm addLast(DNode n)
    addBefore(trailer, n);
```

# Remove a Node

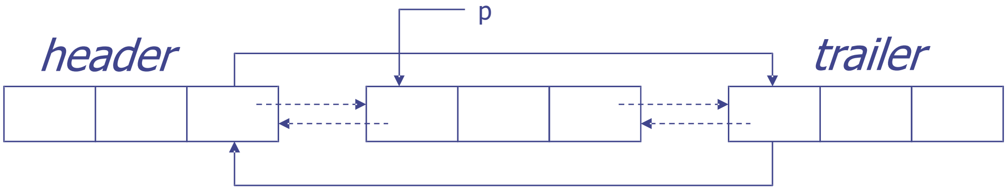


```
// Remove a node p
Algorithm remove(Node p)
    p.getPrev().setNext(p.getNext());
    p.getNext().setPrev(p.getPrev());
    p.setPrev(null);
    p.setNext(null);
    size ← size + 1;
```

# Removal at the Front or Tail

```
Algorithm removeFirst()
    remove(header.getNext());
```

```
Algorithm removeLast()
    remove(trailer.getPrev());
```



# Doubly-Linked List in Java, 3

```java
// public update methods
/** Adds element e to the front of the list. */
public void addFirst(E e) {
  addBetween(e, header, header.getNext());        // place just after the header
}
/** Adds element e to the end of the list. */
public void addLast(E e) {
  addBetween(e, trailer.getPrev(), trailer);      // place just before the trailer
}
/** Removes and returns the first element of the list. */
public E removeFirst() {
  if (isEmpty()) return null;                   // nothing to remove
  return remove(header.getNext());              // first element is beyond header
}
/** Removes and returns the last element of the list. */
public E removeLast() {
  if (isEmpty()) return null;                   // nothing to remove
  return remove(trailer.getPrev());             // last element is before trailer
}
```

# Doubly-Linked List in Java, 4

```java
  // private update methods
  /** Adds element e to the linked list in between the given nodes. */
  private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
    // create and link a new node
    Node<E> newest = new Node<>(e, predecessor, successor);
    predecessor.setNext(newest);
    successor.setPrev(newest);
    size++;
  }
  /** Removes the given node from the list and returns its element. */
  private E remove(Node<E> node) {
    Node<E> predecessor = node.getPrev();
    Node<E> successor = node.getNext();
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size--;
    return node.getElement();
  }
} //----------- end of DoublyLinkedList class -----------
```

# Circular Linked Lists

- Instead of having last node’s next pointer be null, it points back to the first node.