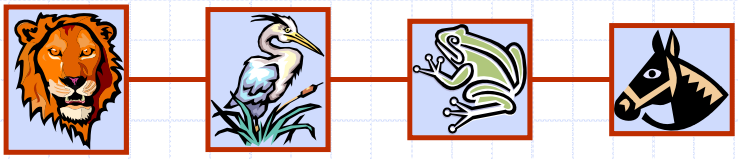
Linked Lists




```
class IntBox {
  int x;
  IntBox(int v) {
    x = v;
  }
  int getValue(){
    return x;
  }
  void setValue(int v) {
    x = v;
  }
}
```

```
class DoubleBox {
  double x;
  DoubleBox(double v) {
    x = v;
  }
  double getValue(){
    return x;
  }
  void setValue(double v) {
    x = v;
```

```
class StringBox {
  String x;
  StringBox(String v) {
    x = v;
  }
  String getValue(){
    return x;
  }
  void setValue(String v) {
    x = v;
  }
}
```

```
IntBox ibox = new IntBox(1234);
int ivalue = ibox.getValue();

StringBox sbox = new StringBox("Hello");
String svalue = sbox.getValue();
```

```
ibox = sbox.getValue();
```

```
ibox = (int) sbox.getValue();
```



```
class IntBox {
  int x;
  IntBox(int v) {
    x = v;
  }
  int getValue(){
    return x;
  }
  void setValu
    x = v;
  }
}
```

```
class DoubleBox {
  double x;
  DoubleBox(double v) {
    x = v;
  }
  double getValue(){
```

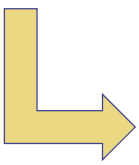
```
(double v) {
```

```
class Box {
  Object x;
  Box(Object v) {
    x = v;
  }
  Object getValue(){
    return x;
  }
  void setValue(Object v) {
    x = v;
  }
}
```

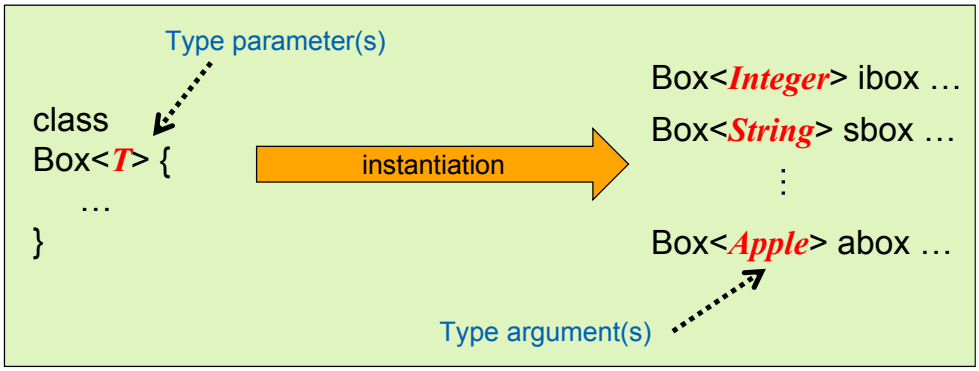
```
    x = v;
  }
}
```

```
Box ibox = new Box(1234);
int ivalue = (Integer) ibox.getValue();

Box sbox = new Box("Hello");
String svalue = (String) sbox.getValue();


svalue = (String) ibox.getValue();

System.out.println(svalue.length());
```


```
class Box {
  Object x;
  Box(Object v) {
    x = v;
  }
  Object getValue(){
    return x;
  }
  void setValue(Object v) {
    x = v;
  }
}
```



```
class Box<T> {
  T x;
  Box(T v) {
    x = v;
  }
  T getValue(){
    return x;
  }
  void setValue(T v) {
    x = v;
  }
}
```




```
class Box<Integer> {
  Integer x;
  Box(Integer v) {
    x = v;
  }
  Integer getValue(){
    return x;
  }
  void setValue(Integer v) {
    x = v;
  }
}
```

```
class Box<String> {
  String x;
  Box(String v) {
    x = v;
  }
  String getValue(){
    return x;
  }
  void setValue(String v) {
    x = v;
  }
}
```

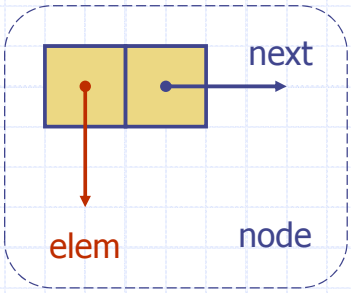


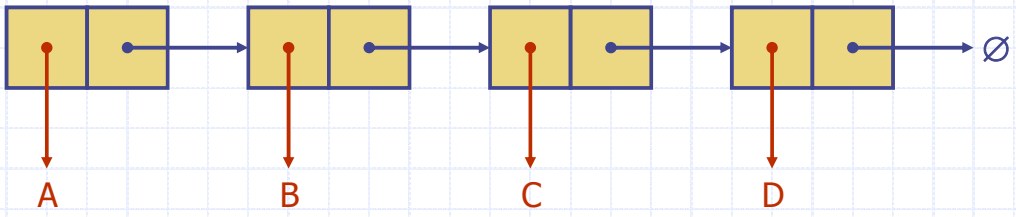
```
Box ibox = new Box(1234);
int ivalue = (Integer) ibox.getValue();

Box sbox = new Box("Hello");
String svalue = (String) sbox.getValue();


svalue = (String) ibox.getValue();

System.out.println(svalue.length());
```

```
Box<Integer> ibox = new Box<>(1234);
int ivalue = ibox.getValue();

Box<String> sbox = new Box<>("Hello");
String svalue = sbox.getValue();


svalue = ibox.getValue();
svalue = (String) ibox.getValue();

System.out.println(svalue.length());
```

Singly Linked List

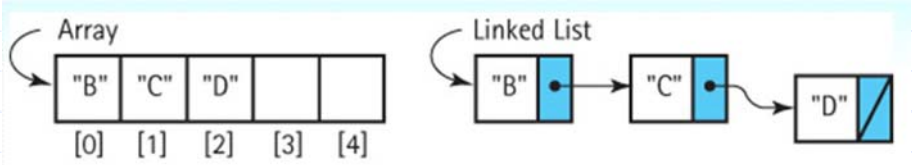
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node



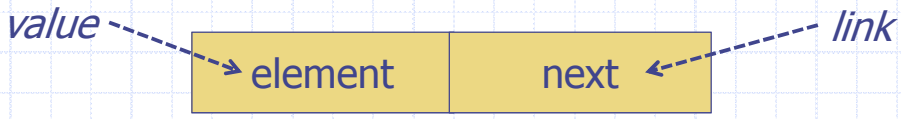


Arrays *vs.* Linked Lists

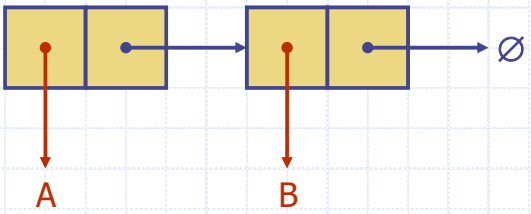
- In array, elements occupy contiguous memory locations.
- In linked list, elements occupy non-contiguous locations.



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:

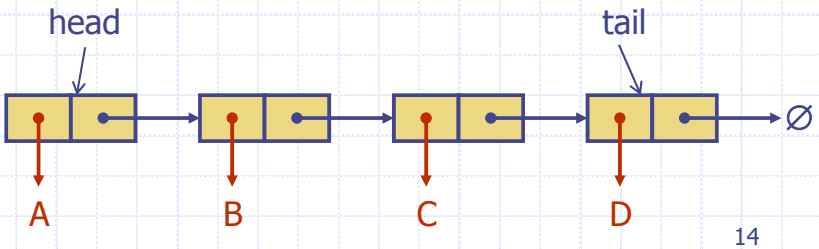
```
public class Node<E> {
   private  E element;
   private  Node<E> next;

   /** 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;
 }
}
```

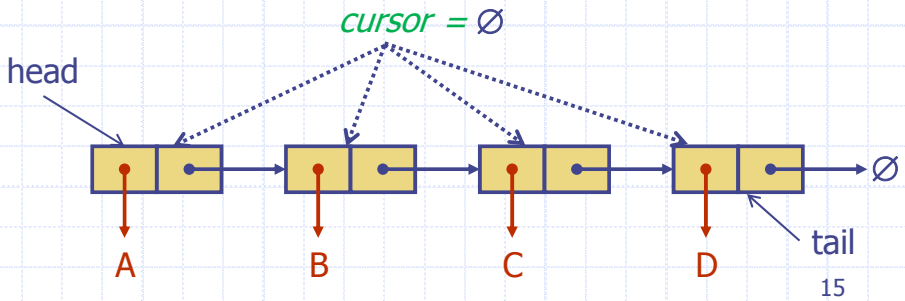

Implementation of SLL

```
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

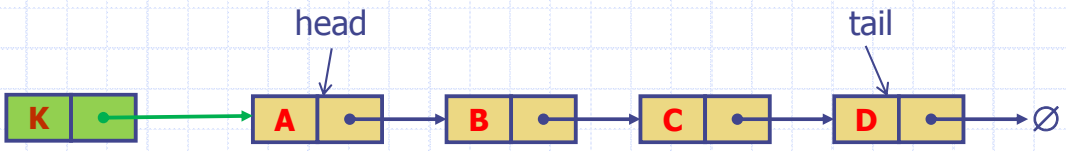
```
Void printAll() {
    Node<T> cursor = head;
    while (curor != null) {
        // do something …
        // ➔ System.out.println(cursor.getElement())
        cursor = cursor.getNext();
    }
}
```



Accessor Methods

```
1   public class SinglyLinkedList<E> {
...     (nested Node class goes here)
14      // instance variables of the SinglyLinkedList
15      private Node<E> head = null;          // head node of the list (or null if empty)
16      private Node<E> tail = null;          // last node of the list (or null if empty)
17      private int size = 0;                 // number of nodes in the list
18      public SinglyLinkedList( ) { }        // constructs an initially empty list
19      // access methods
20      public int size( ) { return size; }
21      public boolean isEmpty( ) { return size == 0; }
22      public E first( ) {                   // returns (but does not remove) the first element
23        if (isEmpty( )) return null;
24        return head.getElement( );
25      }
26      public E last( ) {                    // returns (but does not remove) the last element
27        if (isEmpty( )) return null;
28        return tail.getElement( );
29      }
```

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*)

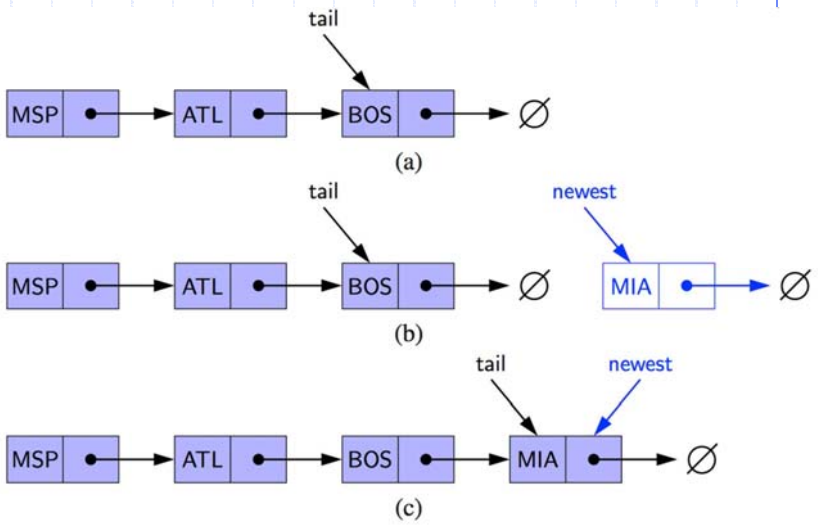
```
public void addFirst(E e) {          // adds element e to the front of the list
  head = new Node<>(e, head);        // create and link a new node
  if (size == 0)
    tail = head;                     // special case: new node becomes tail also
  size++;
}
```

```
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

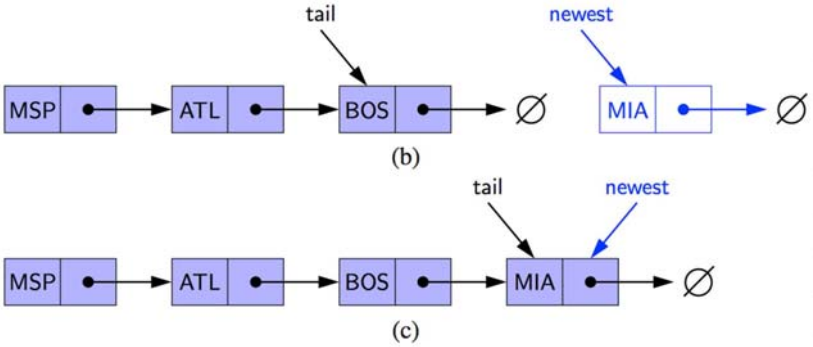
① Allocate a new node
② Insert new element
③ Have new node point to null
④ Have old last node point to new node
⑤ Update tail to point to new node
⑥ Increment size



Be careful about the order of step ④ and ⑤

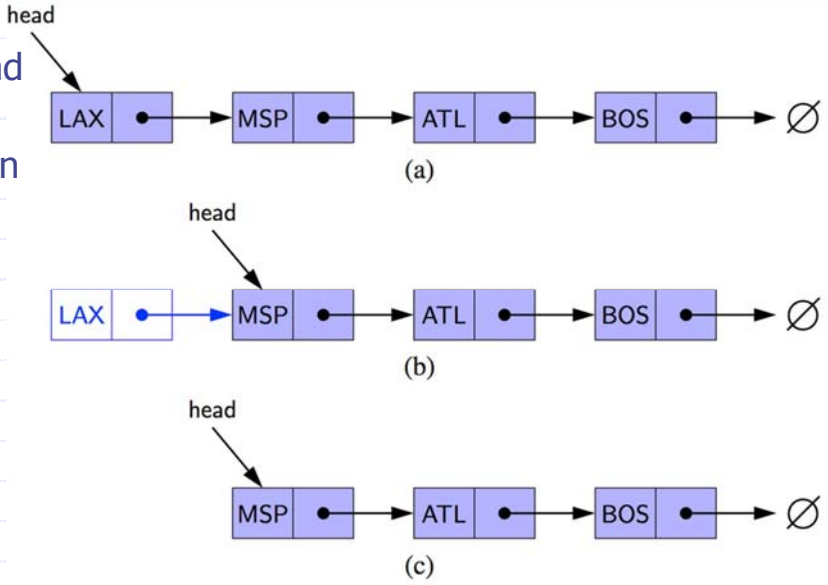
Inserting at the Tail

```
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++;
}
```



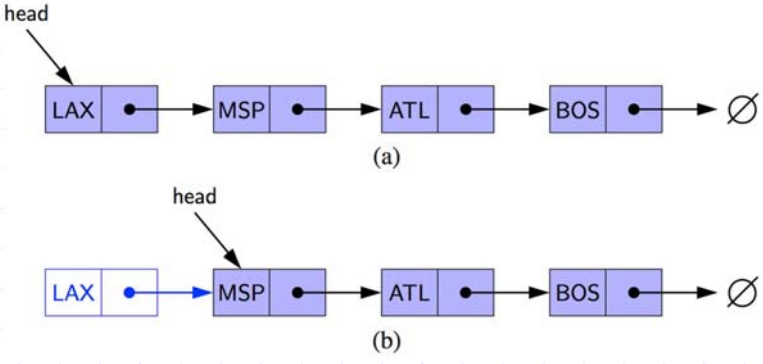
Removing at the Head

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



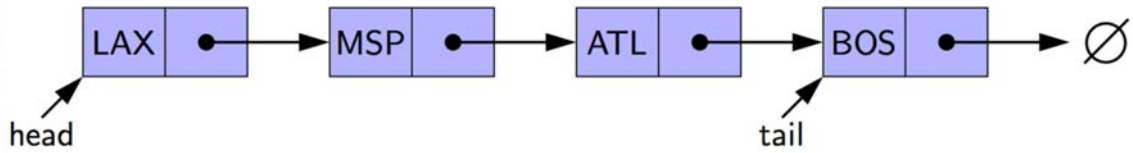
Java Method

```
46    public E removeFirst( ) {                  // removes and returns the first element
47      if (isEmpty( )) return null;             // nothing to remove
48      E answer = head.getElement( );
49      head = head.getNext( );                  // will become null if list had only one node
50      size−−;
51      if (size == 0)
52        tail = null;                           // special case as list is now empty
53      return answer;
54    }
55  }
```



Removing at the Tail

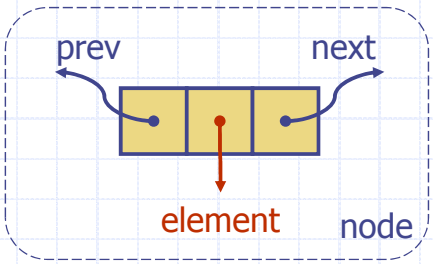
- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node

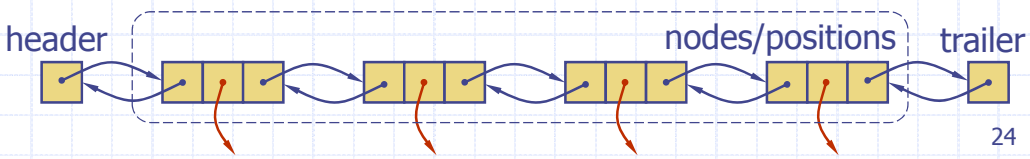


- It is time consuming to remove any node other than the head in a singly linked list. ➔ ***Any improvements?***
- How about insertion or deletion in the middle of the list?
  - ➔ Try for yourself!

Doubly Linked List

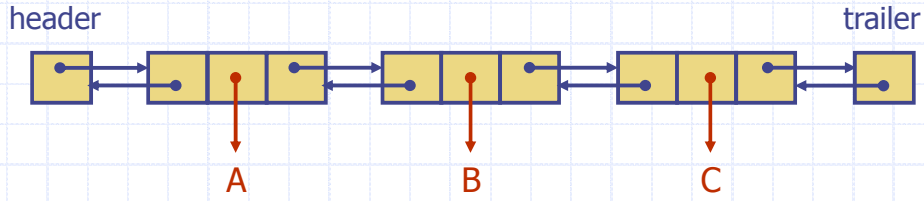
- The doubly linked list allows us to go in both directions – forward and reverse – in the list.
- Supports quick insertion and removal at both ends and in the middle.
- Each node stores
  - element
  - link to the next node
  - link to the previous node
- Special trailer and header nodes



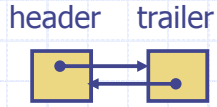


Header and Trailer Sentinels

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



Empty list



Doubly-Linked List in Java

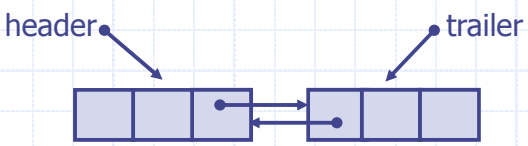
```
1  /** A basic doubly linked list implementation. */
2  public class DoublyLinkedList<E> {
3    //--------------- nested Node class ---------------
4    private static class Node<E> {
5      private E element;                // reference to the element stored at this node
6      private Node<E> prev;             // reference to the previous node in the list
7      private Node<E> next;             // reference to the subsequent node in the list
8      public Node(E e, Node<E> p, Node<E> n) {
9        element = e;
10       prev = p;
11       next = n;
12     }
13     public E getElement( ) { return element; }
14     public Node<E> getPrev( ) { return prev; }
15     public Node<E> getNext( ) { return next; }
16     public void setPrev(Node<E> p) { prev = p; }
17     public void setNext(Node<E> n) { next = n; }
18   } //---------- end of nested Node class ----------
19
```

Doubly-Linked List in Java, 2

```
21   private Node<E> header;                          // header sentinel
22   private Node<E> trailer;                         // trailer sentinel
23   private int size = 0;                            // number of elements in the list
24   /** Constructs a new empty list. */
25   public DoublyLinkedList( ) {
26     header = new Node<>(null, null, null);         // create header
27     trailer = new Node<>(null, header, null);      // trailer is preceded by header
28     header.setNext(trailer);                       // header is followed by trailer
29   }
30   /** Returns the number of elements in the linked list. */
31   public int size( ) { return size; }
32   /** Tests whether the linked list is empty. */
33   public boolean isEmpty( ) { return size == 0; }
34   /** Returns (but does not remove) the first element of the list. */
35   public E first( ) {
36     if (isEmpty( )) return null;
37     return header.getNext( ).getElement( );        // first element is beyond header
38   }
39   /** Returns (but does not remove) the last element of the list. */
40   public E last( ) {
41     if (isEmpty( )) return null;
42     return trailer.getPrev( ).getElement( );       // last element is before trailer
43   }
```

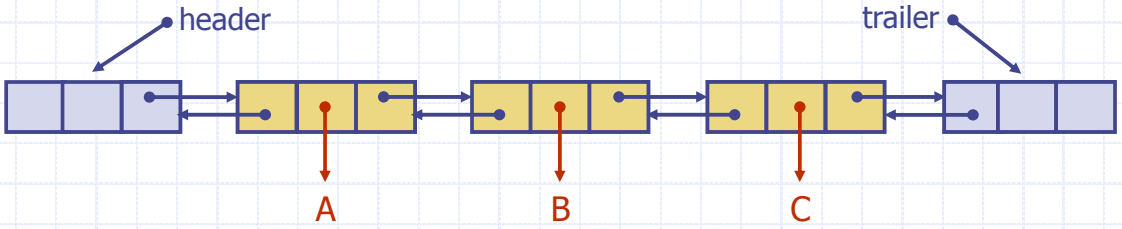

Ctor of DLL

```
24   /** Constructs a new empty list. */
25   public DoublyLinkedList( ) {
26     header = new Node<>(null, null, null);
27     trailer = new Node<>(null, header, null);
28     header.setNext(trailer);
29   }
```

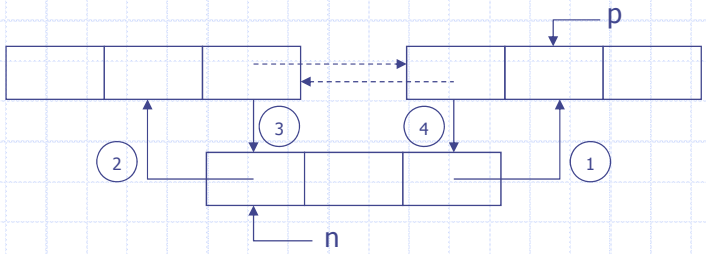


fisrt() and last()

```
35   public E first( ) {
36     if (isEmpty( )) return null;
37     return header.getNext( ).getElement( );          // first element is beyond header
38   }
39   /** Returns (but does not remove) the last element of the list. */
40   public E last( ) {
41     if (isEmpty( )) return null;
42     return trailer.getPrev( ).getElement( );         // last element is before trailer
43   }
```

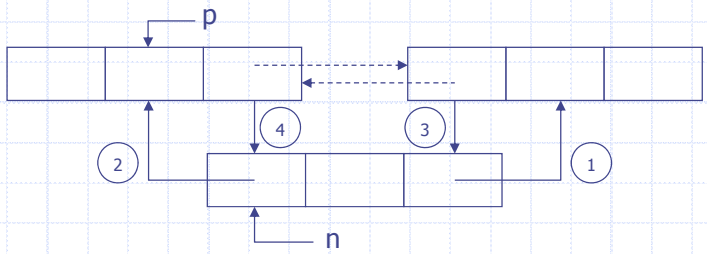


Insertion before a Node




```
// 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



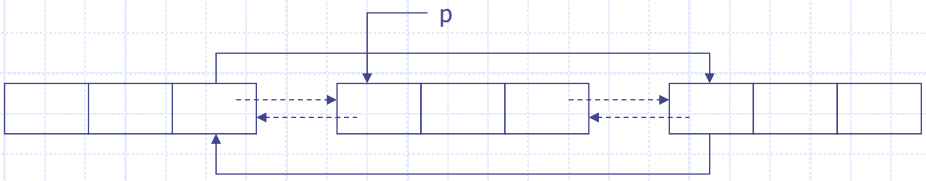

```
// Add new node n After node p
Algorithm addAfter(Node p, Node n)
    n.setNext(p.getNext());
    n.setPrev(p);
    p.getNext().setPrev(n);
    p.setNext(n);
    size ← size + 1;
```


Insertion at the Front or Tail

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

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

Remove a Node

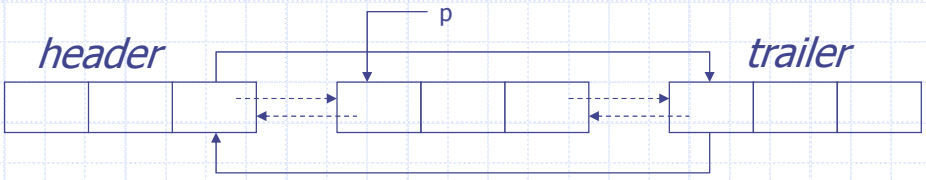



```
// Remove a node p
Algorithm remove(Node p)
    p.getPrev().setNext(p.getNext());
    p.getNext().setPrev(p.getPrev());
    p.setPrev(null);
    p.setNext(null);
    size ← size + 1;
```


Removal at the Front or Tail

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

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



Doubly-Linked List in Java, 3

```
44   // public update methods
45   /** Adds element e to the front of the list. */
46   public void addFirst(E e) {
47     addBetween(e, header, header.getNext());          // place just after the header
48   }
49   /** Adds element e to the end of the list. */
50   public void addLast(E e) {
51     addBetween(e, trailer.getPrev(), trailer);        // place just before the trailer
52   }
53   /** Removes and returns the first element of the list. */
54   public E removeFirst() {
55     if (isEmpty()) return null;                       // nothing to remove
56     return remove(header.getNext());                  // first element is beyond header
57   }
58   /** Removes and returns the last element of the list. */
59   public E removeLast() {
60     if (isEmpty()) return null;                       // nothing to remove
61     return remove(trailer.getPrev());                 // last element is before trailer
62   }
```


Doubly-Linked List in Java, 4

```
64   // private update methods
65   /** Adds element e to the linked list in between the given nodes. */
66   private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
67     // create and link a new node
68     Node<E> newest = new Node<>(e, predecessor, successor);
69     predecessor.setNext(newest);
70     successor.setPrev(newest);
71     size++;
72   }
73   /** Removes the given node from the list and returns its element. */
74   private E remove(Node<E> node) {
75     Node<E> predecessor = node.getPrev();
76     Node<E> successor = node.getNext();
77     predecessor.setNext(successor);
78     successor.setPrev(predecessor);
79     size--;
80     return node.getElement();
81   }
82 } //---------- end of DoublyLinkedList class ----------
```

Circular Linked Lists

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