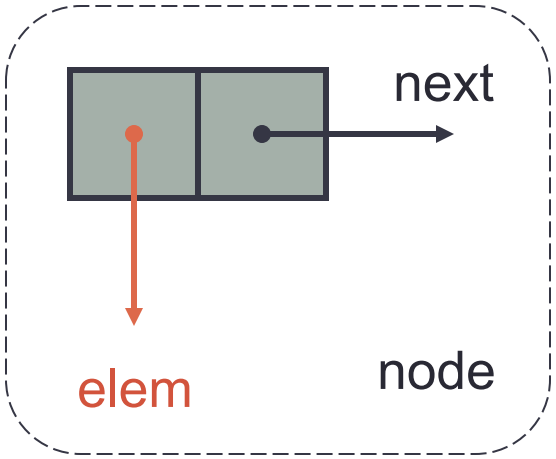
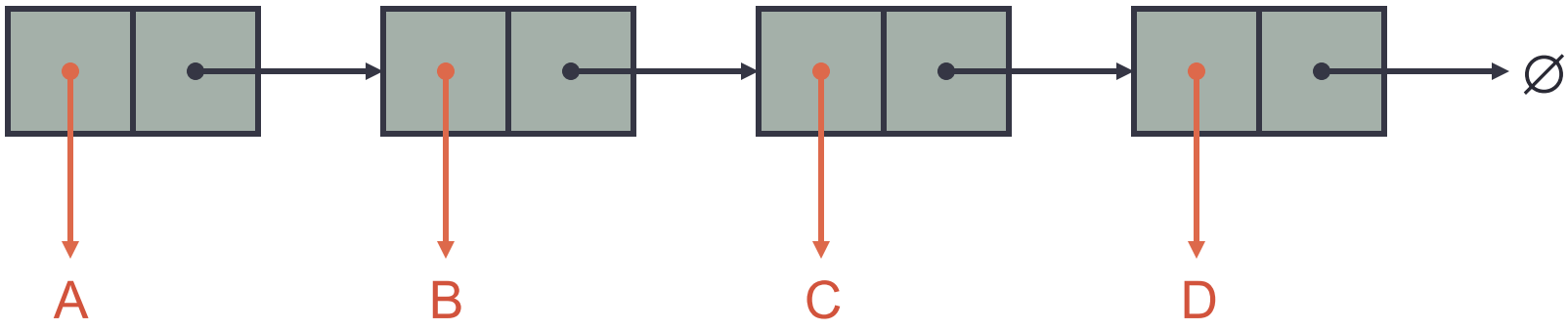
LINKED LISTS

EECS 2011

Singly Linked Lists (3.2)

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





"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;
    }
}
```

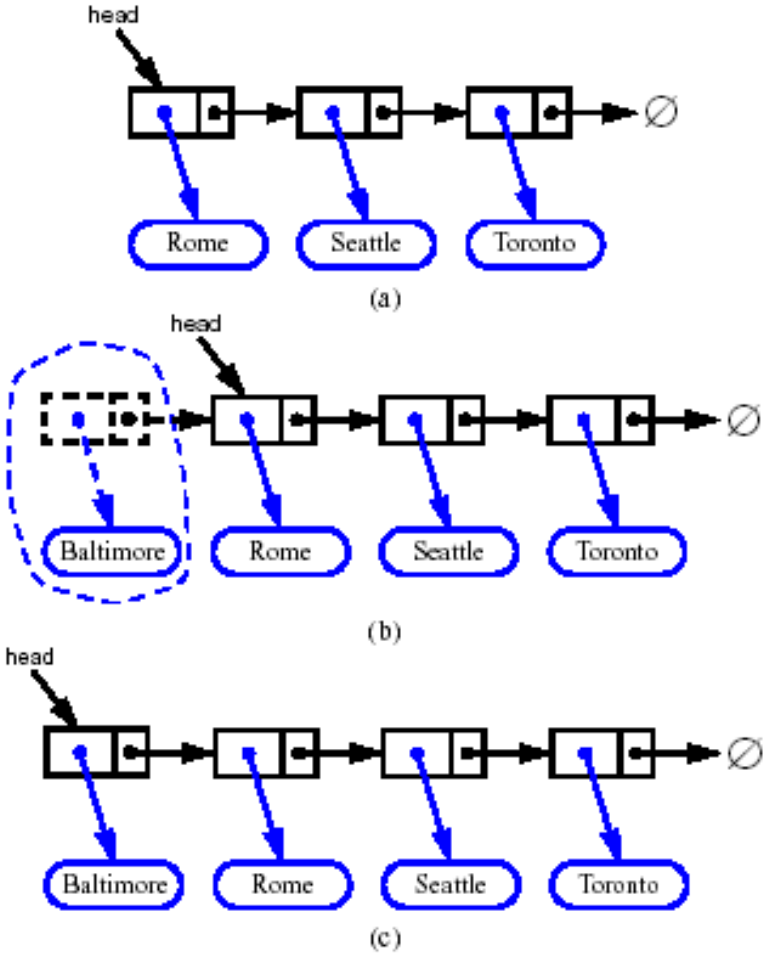
SLinkedList class

```
/** Singly linked list .*/
public class SLinkedList {
  private Node head;                    // head node of the list
  private int size;                     // number of nodes in the list
  /** Default constructor that creates an empty list */
  public SLinkedList() {
    head = null;
    size = 0;
  }

  // ... add, remove and search methods would go here ...
}
```

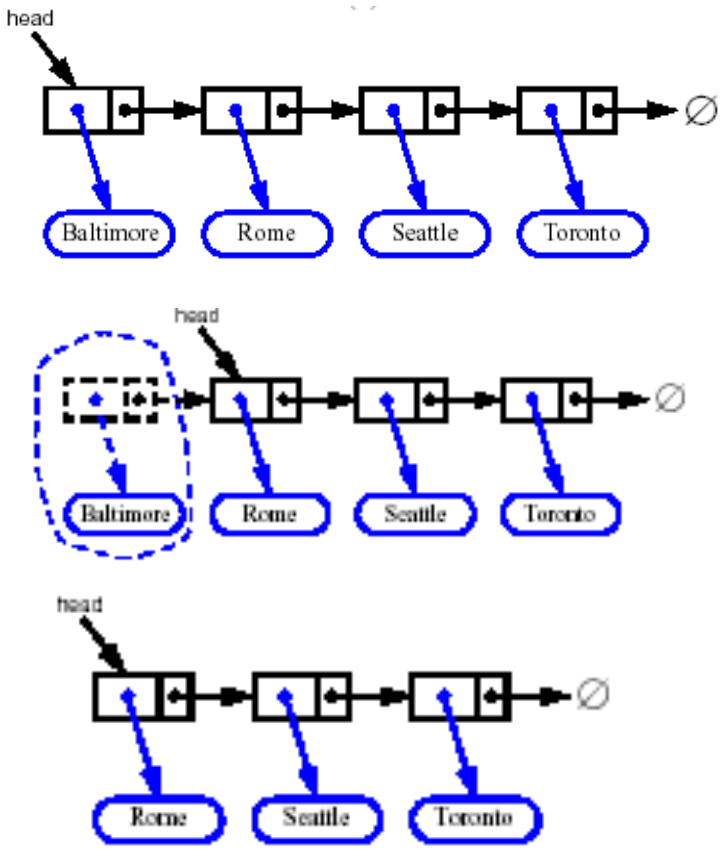
Inserting at the Head

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.



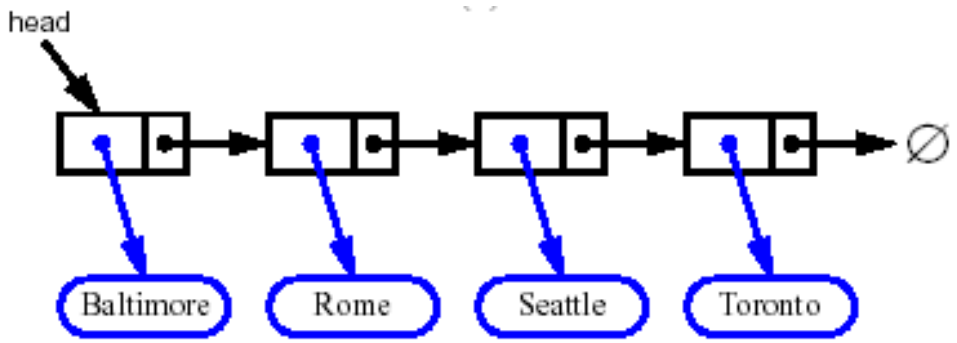
Removing at the Head

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 without a Tail Pointer

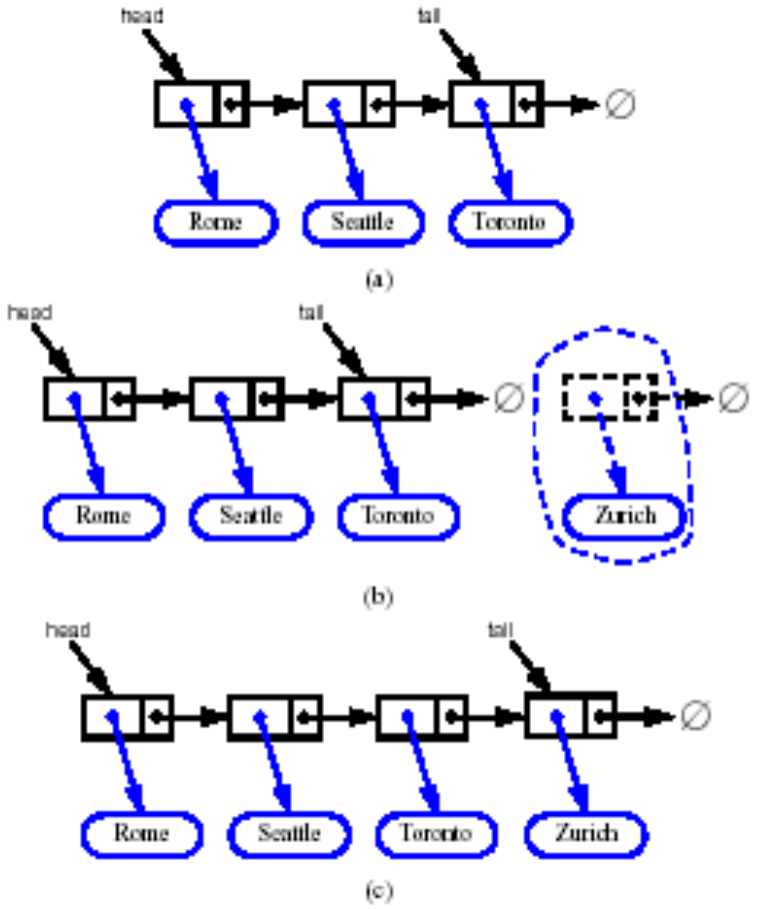
- Running time?



Inserting at the Tail

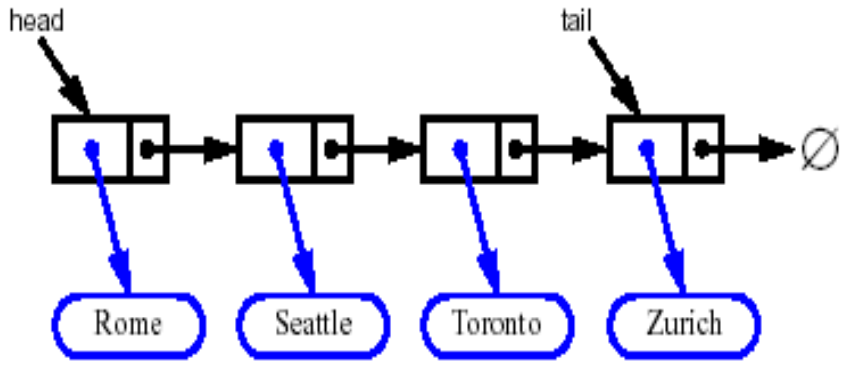
Assume that we keep a pointer to the last element of the list ("tail").

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



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.
- *Homework*: write Java code to remove at the tail of a singly linked list.



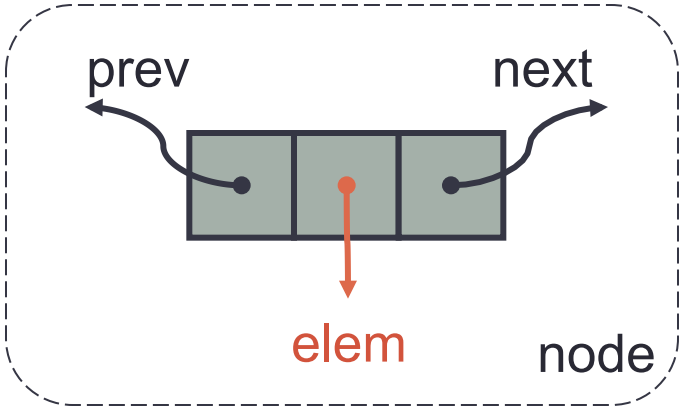
Singly Linked Lists: Summary

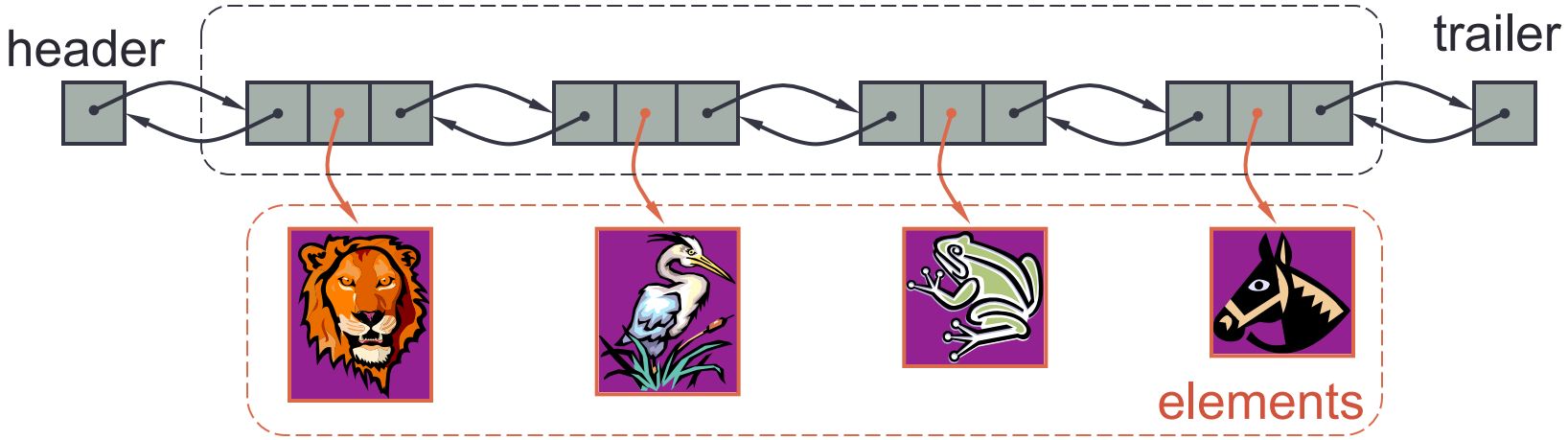
Algorithm	Running Time
Inserting at the head (*addFirst*)	O( )
Removing at the head (*removeFirst*)	O( )
Inserting at the tail (*addLast*)	O( )
Removing at the tail (*removeLast*)	O( )
Search for an element (*found*)	O( )

- Review Code Fragments 3.14 and 3.15
- Write methods *removeLast* and *found*.

Doubly Linked List (3.4)

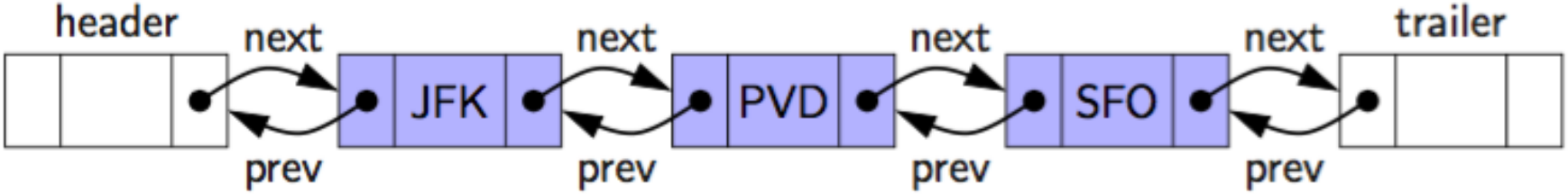
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Header and Trailer Sentinels

- Dummy nodes which do not store any elements.
- To simplify programming.
- Homework: implement the operations of a doubly linked list with NO header and trailer sentinels.
  - Compare the code of the 2 implementations.



"DNode" Class

```
/** Node of a doubly linked list of strings */
public class DNode {
  private String element;          // String
   element stored by a node
  private DNode prev, next;        // Pointers
   to previous and next nodes
  /** Constructor that creates a node with
   given fields */
  public DNode(String e, DNode p,
                          DNode n) {
    element = e;
    prev = p;
    next = n;
  }
```

```
  /** Returns the element of this node */
  public String getElement() { return
   element; }
  /** Returns the previous node of this
   node */
  public DNode getPrev() { return prev; }
  /** Returns the next node of this node */
  public DNode getNext() { return next; }
  /** Sets the element of this node */
  public void setElement(String newElem)
   { element = newElem; }
  /** Sets the previous node of this node */
  public void setPrev(DNode newPrev)
   { prev = newPrev; }
  /** Sets the next node of this node */
  public void setNext(DNode newNext)
   { next = newNext; }
}
```

"DList" Class

```
/** Doubly linked list with nodes of type DNode
   storing strings. */
public class DList {
  private int size;  // number of elements
  private DNode header, trailer;  // sentinels
  /** Constructor that creates empty list */
  public DList() {
   size = 0;
   header = new DNode(null, null, null);
   trailer = new DNode(null, header, null);
   header.setNext(trailer);       // make header
   and trailer point to each other
  }
... // Implementation of methods
}
```

Methods:

int size()

boolean isEmpty()

String first()

String last()

void addFirst(String e)

void addLast(String e)

void removeFirst()

Void removeLast()

void addBefore(String e, DNode v)

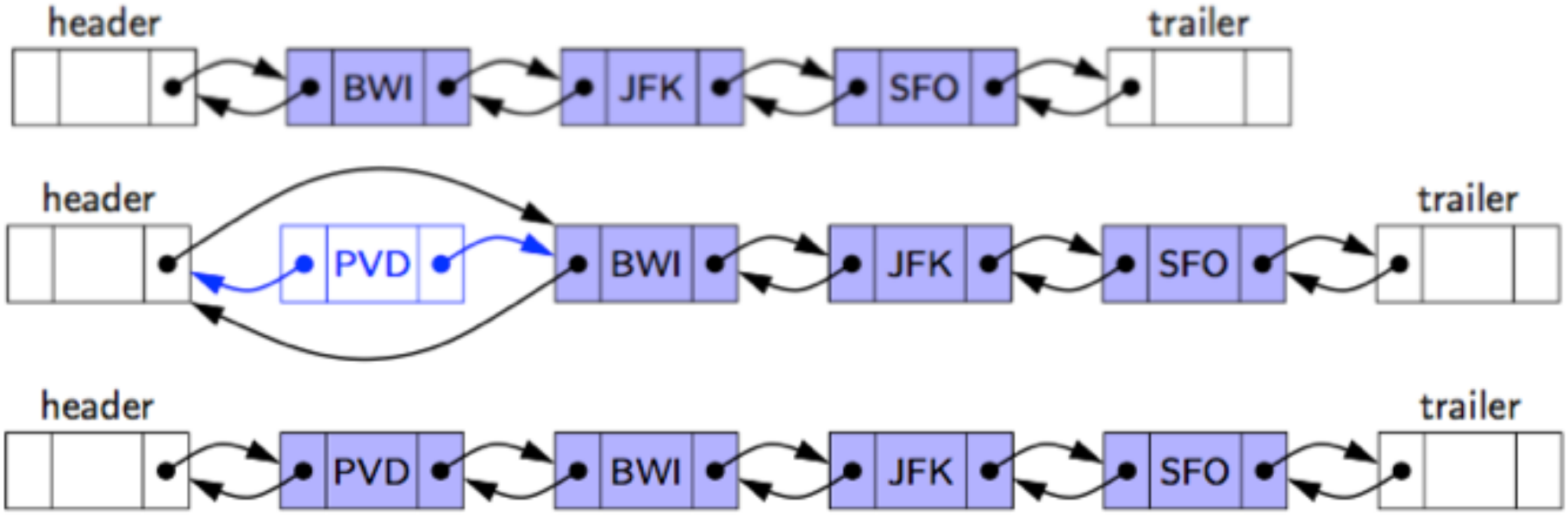
void addAfter(String e, DNode v)

void addBetween (String e, DNode u, DNode v)

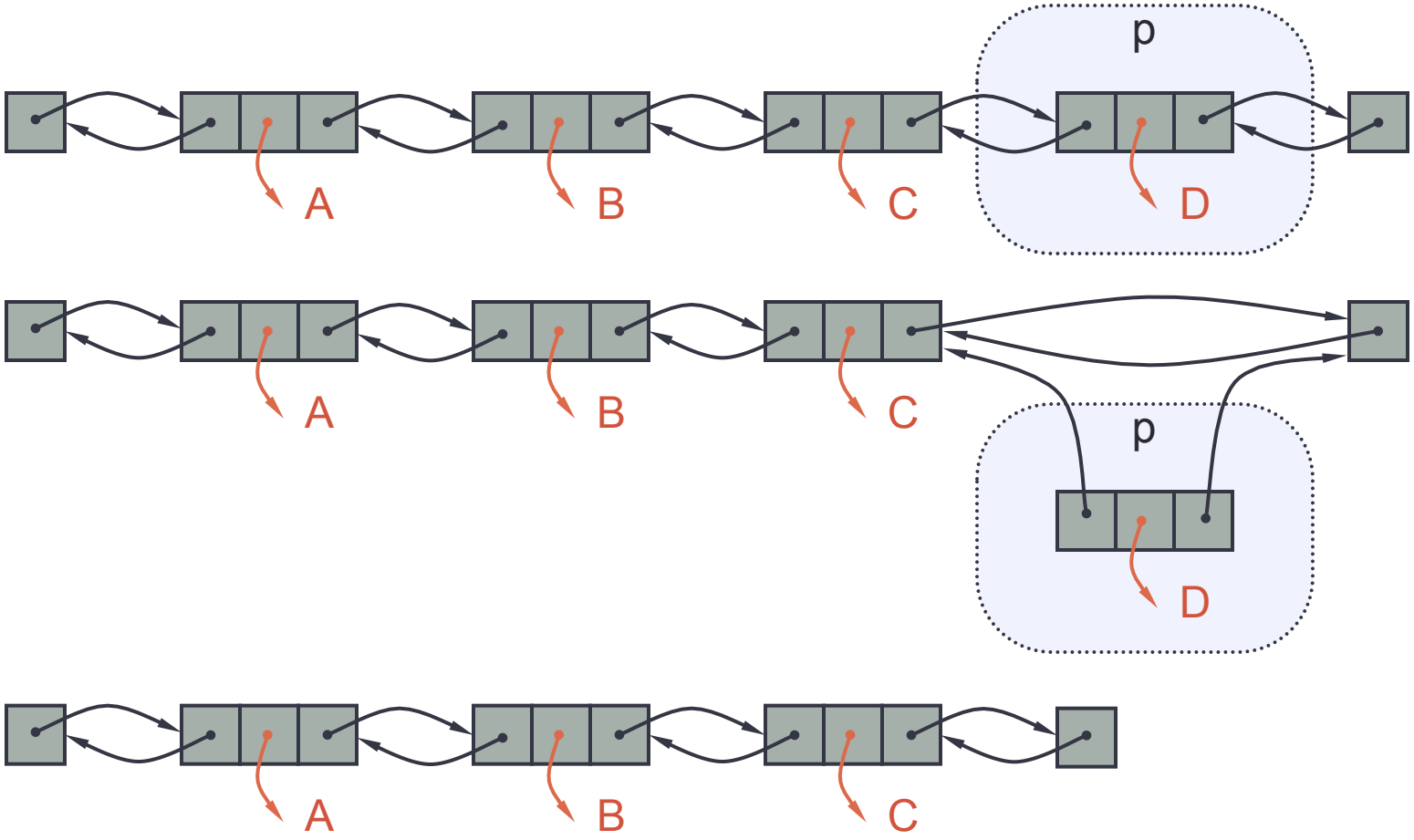
void remove(DNode v)

Adding or Removing at Either End

- Straightforward.
- Example: insert at the head (*addFirst*)



Removal at the Tail of the List (*removeLast*)

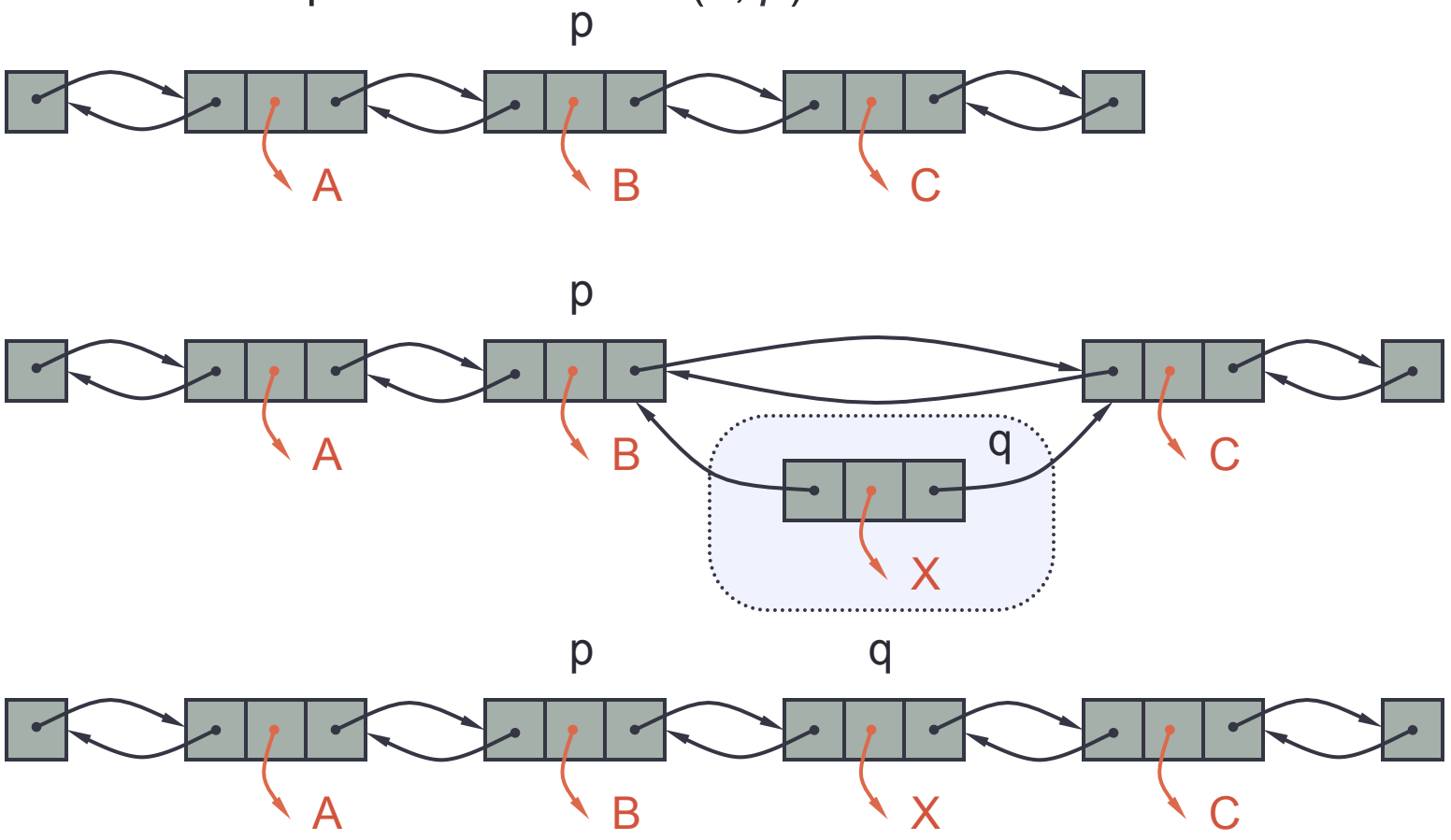


Removing at the Tail: Algorithm

```
Algorithm removeLast() {
if size == 0 then
   Indicate error “empty list”;
v = trailer.getPrev();  // last node
u = v.getPrev();        // node before last node
trailer.setPrev(u)
u.setNext(trailer);
v.setPrev(null);
v.setNext(null);
size = size – 1;
}
```

Insertion in the Middle of the List

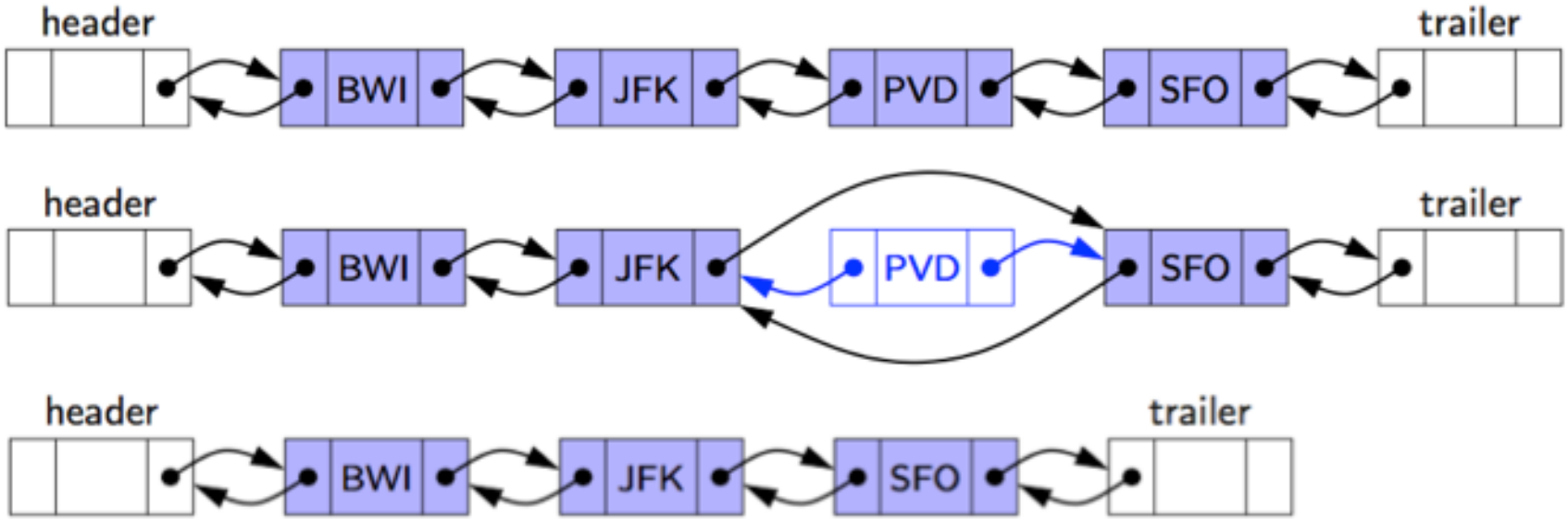
- We visualize operation *addAfter*(*e*, *p*).



Insertion Algorithm

```
Algorithm addAfter(e, p) {
Dnode q = new Dnode(e, null, null);
r = p.getNext;  // node after p
q.setPrev(p);   // link q to its predecessor, p
q.setNext(r);   // link q to its successor, r
r.setPrev(q) ); // link r to its new predecessor, q
p.setNext(q);   // link p to its new successor, q
size = size + 1;
}
```

Removal in the Middle of the List



Removal Algorithm

```
Algorithm remove(v) {
u = v.getPrev();    // node before v
w = v.getNext();    // node after v
w.setPrev(u);       // link out v
u.setNext(w);
v.setPrev(null);    // null out the fields of v
v.setNext(null);
size = size – 1;
}
```

Doubly Linked Lists: Summary

Algorithm	Running Time
Inserting at the head (*addFirst*)	O( )
Removing at the head (*removeFirst*)	O( )
Inserting at the tail (*addLast*)	O( )
Removing at the tail (*removeLast*)	O( )
Adding before, after, in between	O( )
Removing any element	O( )

- Review Code Fragments 3.17 and 3.18
- Homework: re-do the implementation without using the header and trailer sentinels.

Next lectures …

- Stacks (6.1)
- Queues (6.2)