Lecture 14: Doubly Linked Lists

CS 62

Spring 2019

William Devanny & Alexandra Papoutsaki

Feedback Summary

- Assignments – good for applying lecture content in depth (11)
- Want more Java review (10)
- Office hours, mentor hours, and Piazza are helpful (8)
- But want more hours and to more effective OH/MH time (8)

Feedback Summary

- Make pace of course faster (4)
- Make pace of course slower (3)

Changes

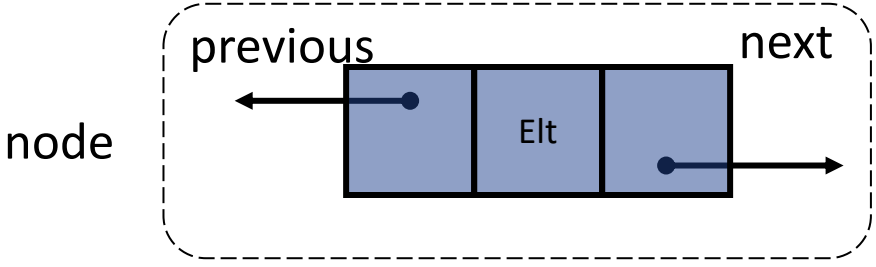
- Assignment alignment:
  - Extension for Assignment 4
    - Now due Monday, March 4th
  - Assignment 5 is now extra credit
- Mentor hours:
  - Mentors will have a timer to make sure they balance their time
  - Mentor hour increases (see Piazza)
  - Extra credit questions have lower priority
- Office hours:
  - Office hour increases (see Piazza)

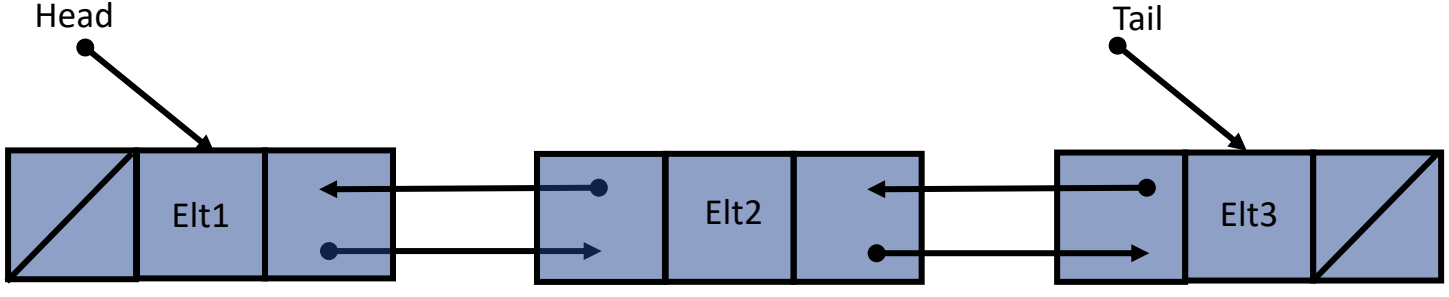
Coding Quick Tip: Debugging

- End to end test
  - For assignment 3, test going from input file to final sorted output
- Rubber ducky debugging
  - Explain your code to a rubber ducky
- Unit tests
  - Create simple tests to test functionality of individual methods
- The Eclipse debugger
  - Powerful tool to step through code line by line
  - Explored in next week's lab

Doubly Linked List

- A linked list consisting of a sequence of nodes, starting from a head pointer and ending to a tail
- Each node stores
  - Element
  - Link to the previous node
  - Link to the next node





```
public class DoublyLinkedNode<E>{
        protected E data; // value stored in this element
        protected DoublyLinkedNode<E> nextElement; // ref to next
        protected DoublyLinkedNode<E> previousElement; // ref to previous

        public DoublyLinkedNode(E v, DoublyLinkedNode<E> next, DoublyLinkedNode<E> previous) {
                data = v;
                nextElement = next;
                if (nextElement != null)
                        nextElement.previousElement = this;
                previousElement = previous;
                if (previousElement != null)
                         previousElement.nextElement = this;
                }
        }
        public DoublyLinkedNode(E v) {
                this(v,null,null); // constructs a single element
        }
        //setters and getters
}
```

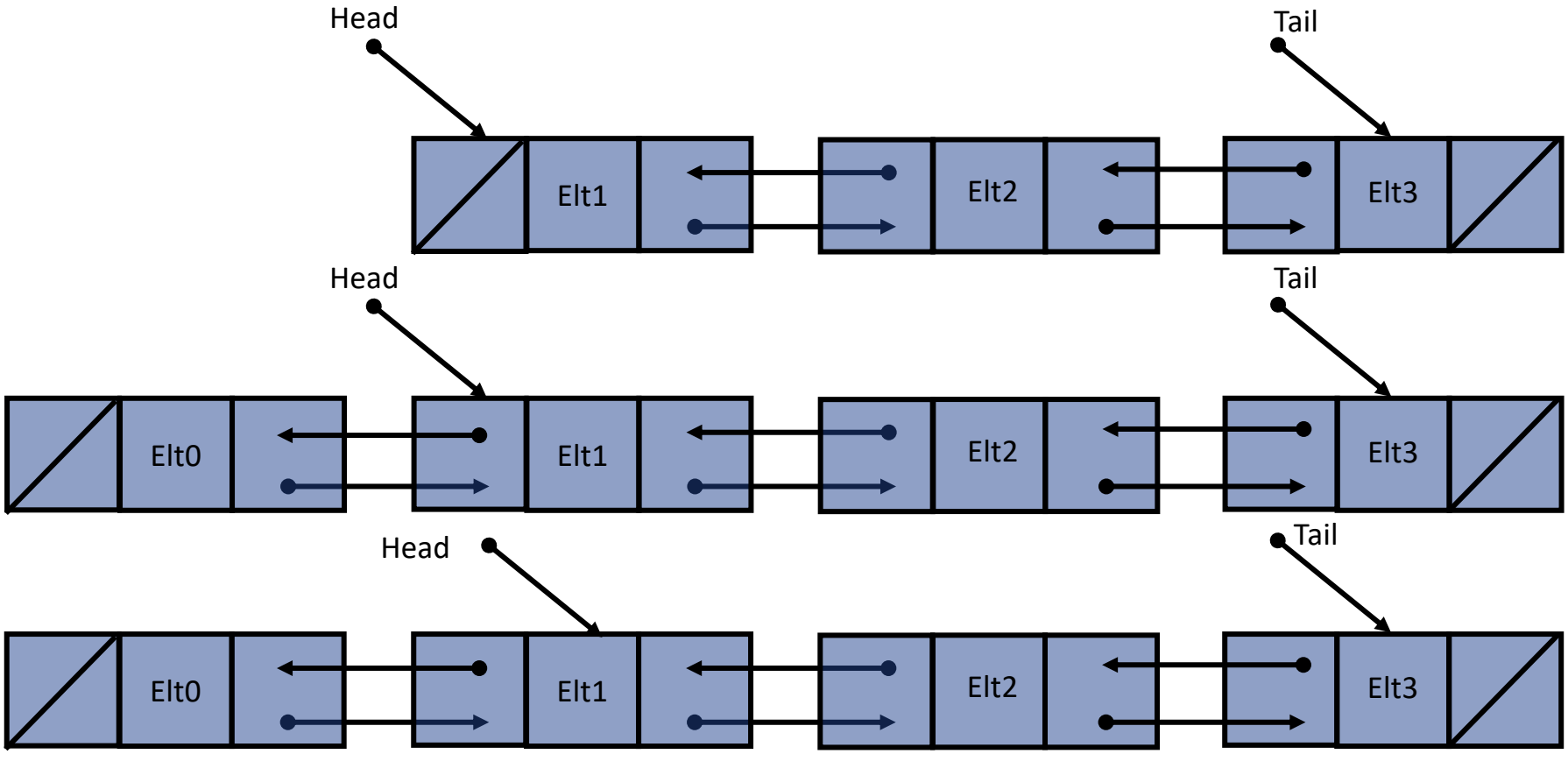
DoublyLinkedList

```
public class DoublyLinkedList<E> extends AbstractList<E> {
        protected int count; // number within list
        protected DoublyLinkedNode<E> head; // ref. to first element
        protected DoublyLinkedNode<E> tail; // ref. to last element


        //construct an empty list
        public DoublyLinkedList() {
                head = null;
                tail = null;
                count = 0;
         }
        public E getFirst() {
                return head.value(); //returns first value in list
        }
        public E getLast() {
                return tail.value(); //returns lastvalue in list
        }


        public int size() {
                return count;
        }
```

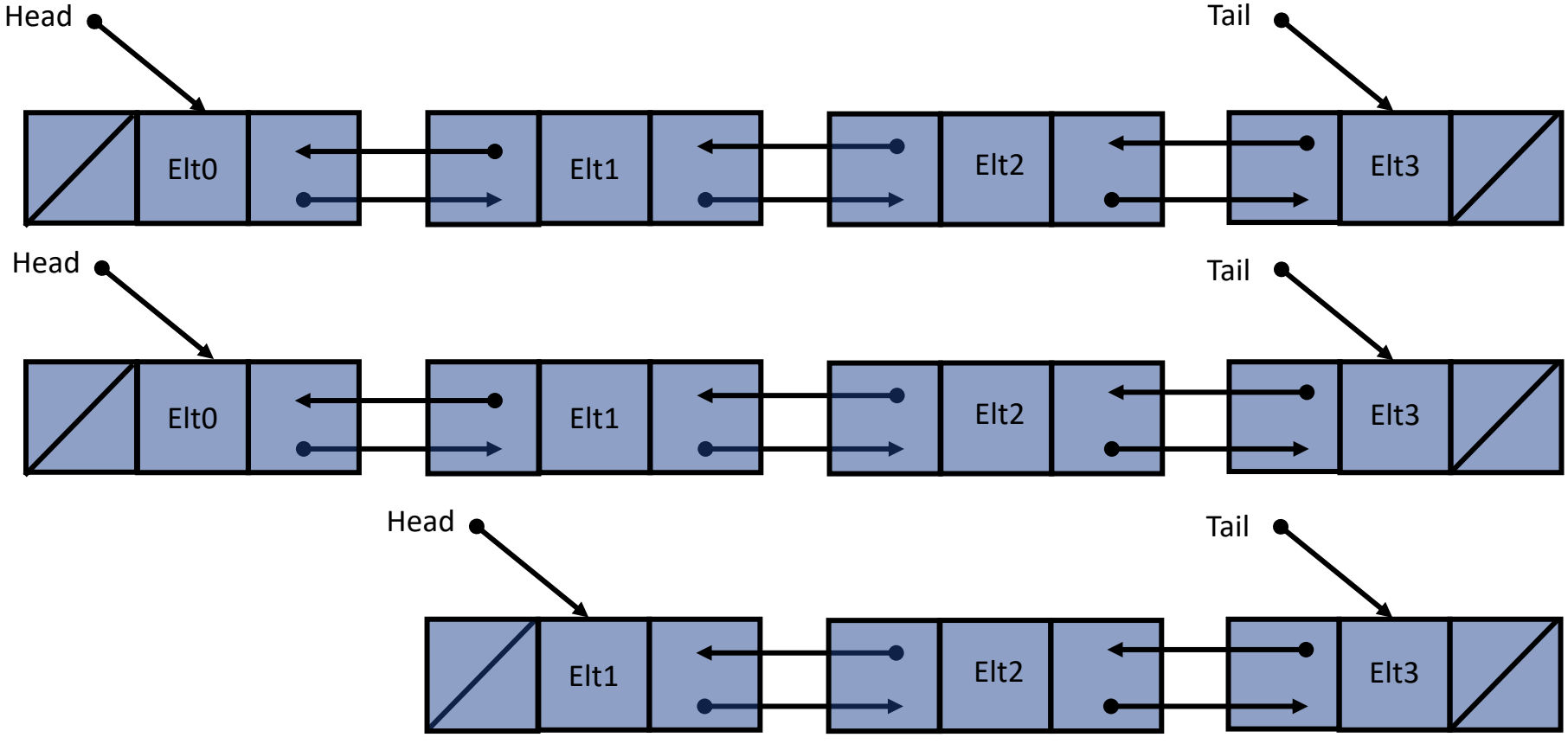
Adding at the head



Adding at the head is $O(1)$

```
public void addFirst(E value){
      // construct a new node making it head
      head = new DoublyLinkedNode<E>(value, head, null);
      if(tail == null) //list was empty
            tail = head;
      count++;
}
```

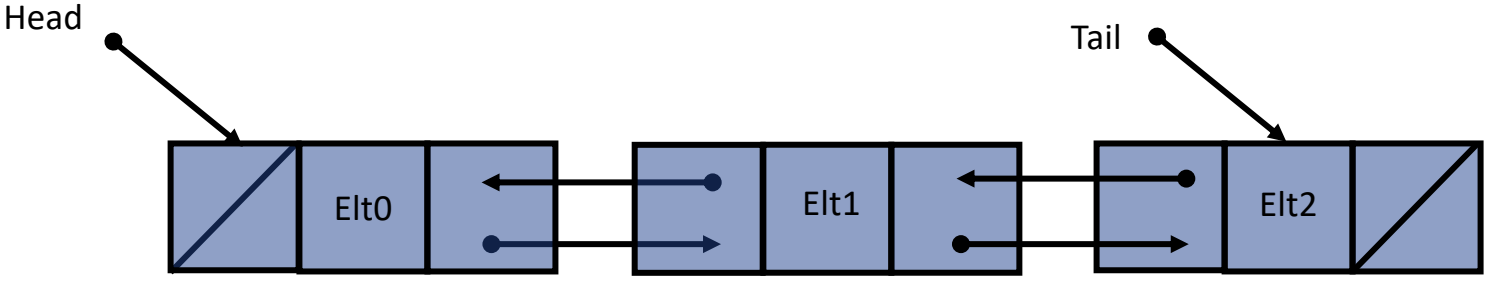
Removing at the head

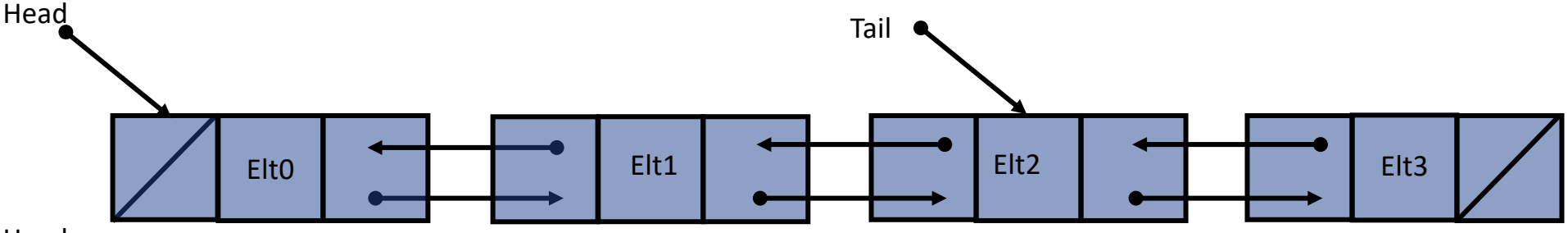


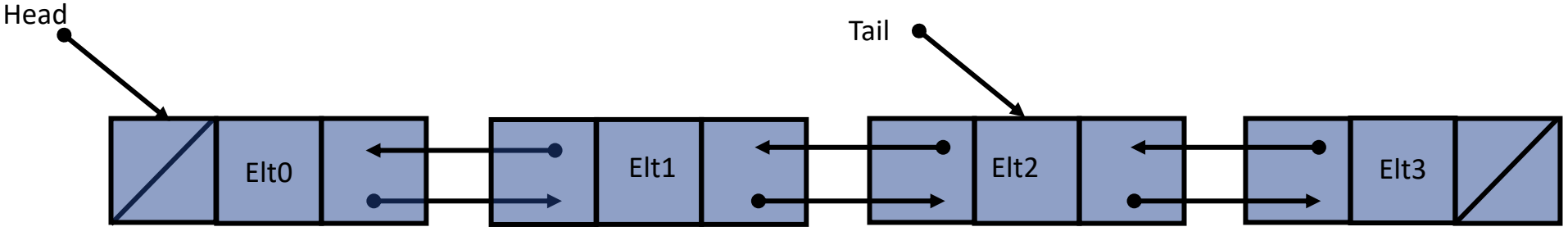
Removing at the head is $O(1)$

```
public E removeFirst(){
     //check that list is not empty
     DoublyLinkedNode<E> temp = head;
     head = head.next(); // move head down list
     if (head != null)
          head.setPrevious(null);
     else
          tail = null; //remove final value
     count--;
     return temp.value();
}
```

Adding at the tail



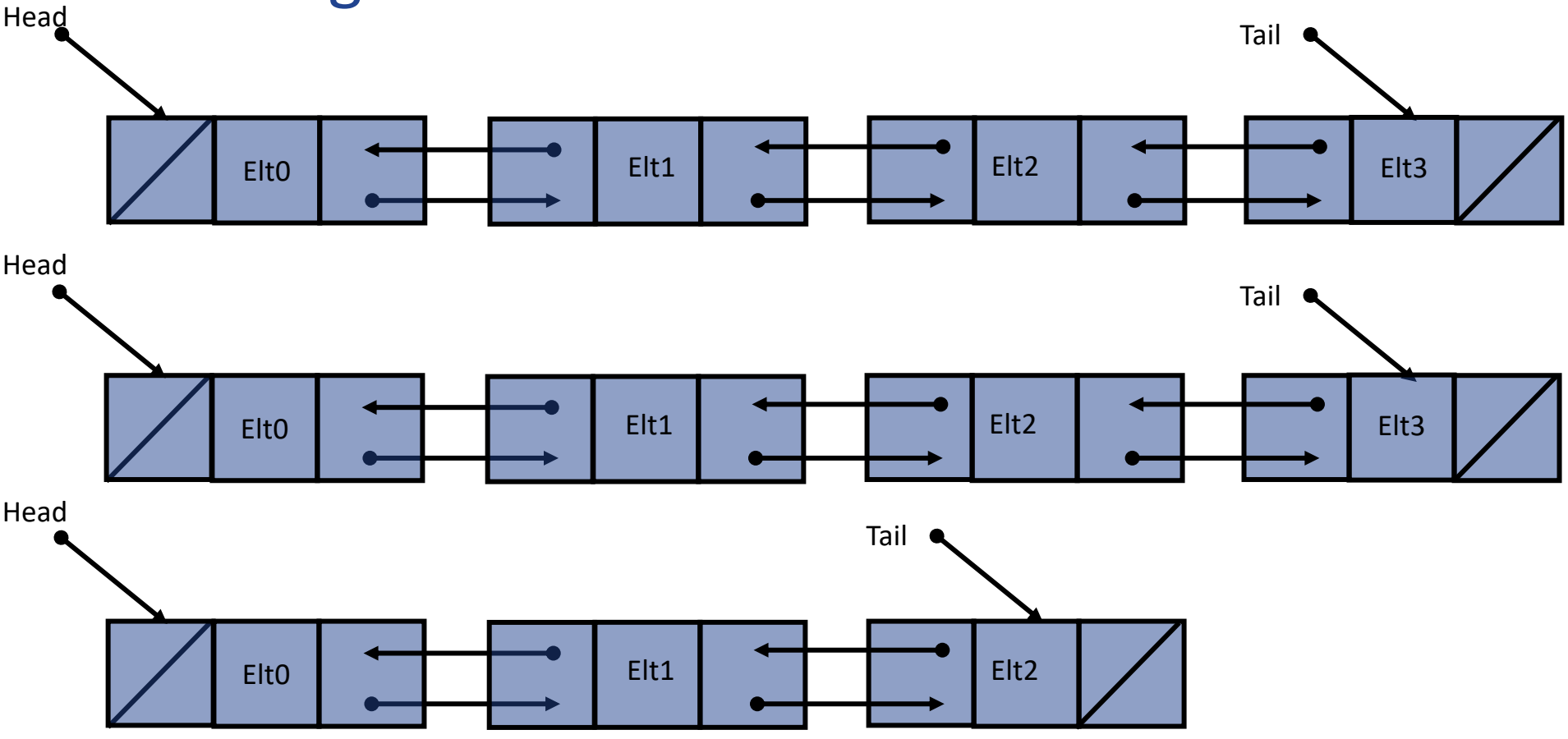




Adding at the tail is $O(1)$

```
public void addLast(E value) {
      tail = new DoublyLinkedNode<E>(value, null, tail);
      if (head == null)
            head = tail; //fix up head, it's the first node
      count++;
}
```

Removing from the tail

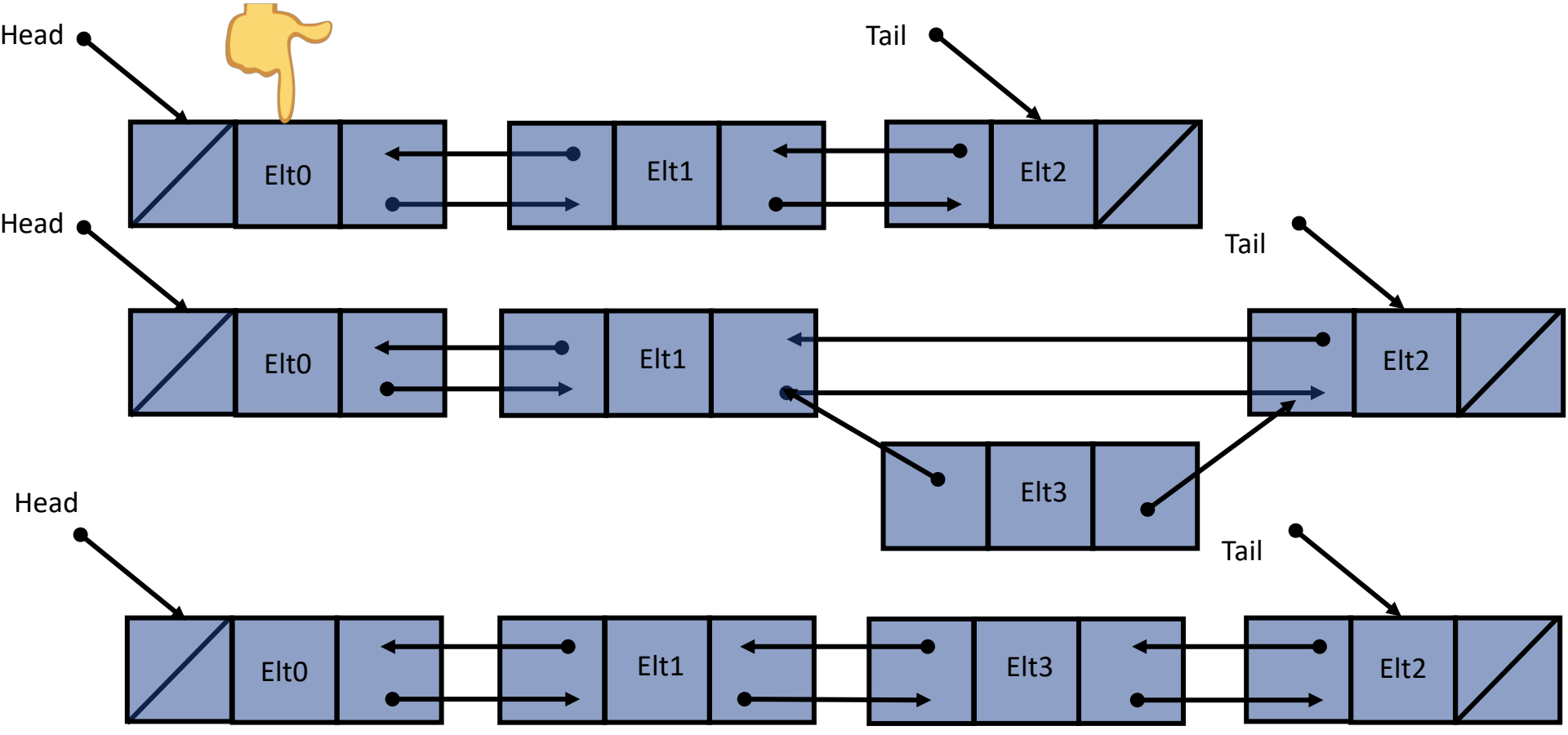


Removing from the end is $O(1)$

```
public E removeLast() {
     //check that the list is not empty
     DoublyLinkedNode<E> temp = tail;
     tail = tail.previous();
     if (tail == null)
          head = null; //empty list
     else
          tail.setNext(null);
     count--;
     return temp.value();
}
```

Adding between nodes

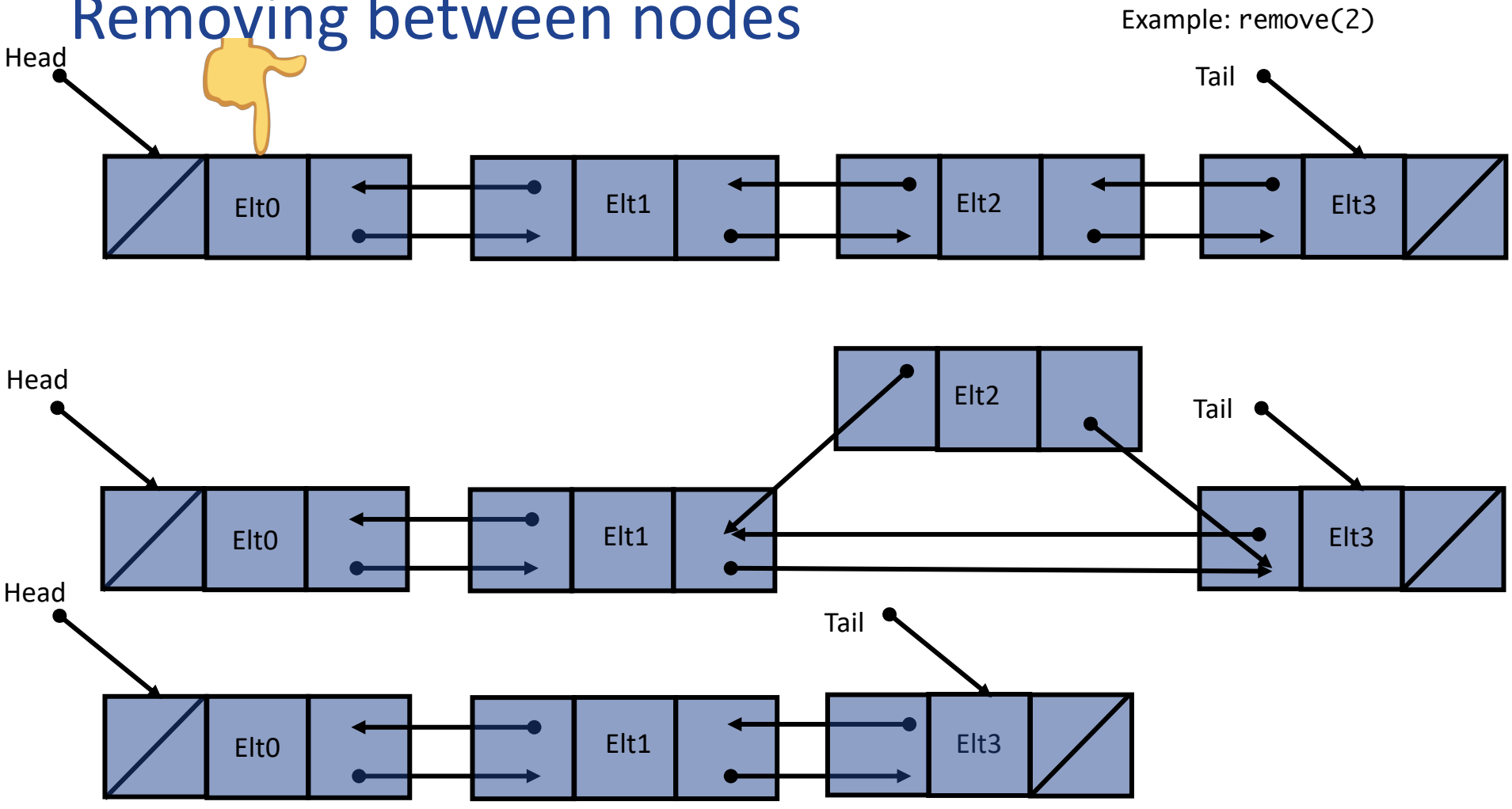
Example: `add(2,Elt3)`



Adding between nodes is $O(n)$

```
public void add(int i, E o) {
	//check that i in range. If i==0 call addFirst,
	//if i==size() call addLast()
	DoublyLinkedNode<E> before = null;
	DoublyLinkedNode<E> after = head;
	while (i > 0) { // search for ith position, or end of list
		before = after;
		after = after.next();
		i--;
	}
	// create new value to insert in correct position
	DoublyLinkedNode<E> current = new DoublyLinkedNode<E>(o,after,before);
	count++; // make after and before value point to new value
	before.setNext(current);
	after.setPrevious(current);
}
```

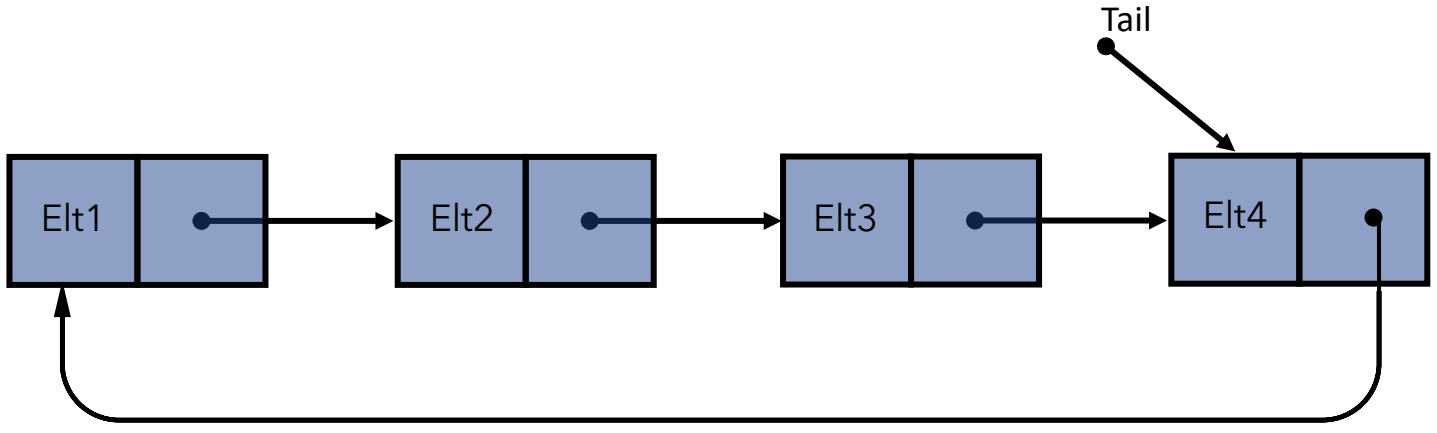
Removing between nodes



Removing between nodes is $O(n)$

```
public E remove(int i) {
        //check that i in range. If i==0 call removeFirst,
        //if i==size() call removeLast()
        DoublyLinkedNode<E> previous = null;
        DoublyLinkedNode<E> finger= head;
        while (i > 0) { // search for ith position, or end of list
                previous = finger;
                finger = finger.next();
                i--;
        }
        previous.setNext(finger.next());
        finger.next().setPrevious(previous);
        count--;
        return finger.value();
}
```

Circular list



- Accessing/modifying the head or the tail is $O(1)$.
- Circular lists are as space-efficient as singly linked lists but tail-related operations are less costly.

More on Linked Lists

`structure5` provides iterators both for singly and doubly linked lists
→ Check the code!

java.util.LinkedList implements a Doubly Linked List

Doubly linked lists are often represented as circular: