Lecture 13: Linked Lists

CS 62

Spring 2019

William Devanny & Alexandra Papoutsaki

Writing code

- This week's lab: Unit Testing with JUnit
  - No complex code ever works the first time.
  - Neither the "If I just fix this last thing…" attitude.
- Think about testing before you write the code.
  - Common interview question: how could a method break? Edge cases?
  - Comment before you write the code.
  - Avoid writing more than a method without testing it.
  - Once tests pass and you are satisfied, commit your work.

List ADT

```
public abstract class AbstractList<E>{
    public AbstractList() { }
    public boolean isEmpty() { return size() == 0; }
    public void addFirst(E value) {add(0,value); }
    public void addLast(E value) {add(size(),value);}
    public E getFirst() { return get(0);}
    public E getLast() {return get(size()-1);}
    public E removeFirst() { return remove(0); }
    public E removeLast() { return remove(size()-1); }
    public void add(E value) { addLast(value);}
    public E remove() { return removeLast();}
    public E get() { return getLast(); }
    public boolean contains(E value) {
        return -1 != indexOf(value); }
```

Linked List

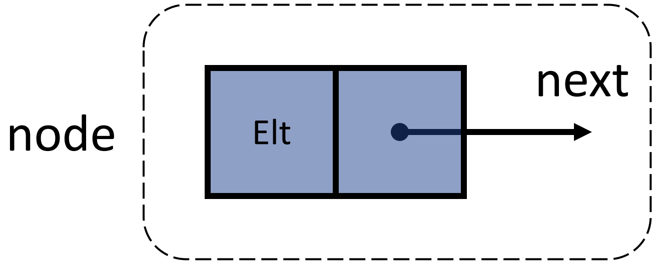
- A data structure consisting of a linear sequence of nodes
  - Think of them as snap-lock beads

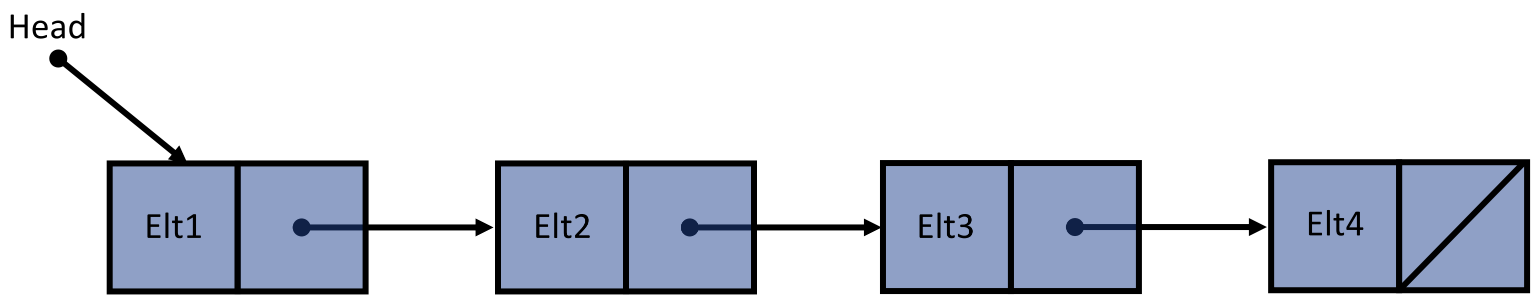
- Alternate implementation of an abstract list
  - `Vector` (ArrayList) in `structure5` also extends `AbstractList`

- Trade-offs in complexity:
  - In ArrayList, adding elements at beginning of list is expensive
  - For Linked lists, it is inexpensive to add elements in early positions
  - However, accessing the last element is slow

Singly Linked List

- A linked list consisting of a sequence of nodes, starting from a head reference
- Each node stores
  - Element
  - Link to the next node





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

        public Node(E v, Node<E> next) {
                data = v;
                nextElement = next; // construct the new head of a singly linked list
        }
        public Node(E v) {
                this(v,null); // constructs a new tail of a list with value v
        }
        public Node<E> next() {
                return nextElement; // returns reference to next value in list
        }
        public void setNext(Node<E> next) {
                nextElement = next; // sets reference to new next value
        }
        public E value() {
                return data; // returns value associated with this element
        }
        public void setValue(E value) {
                data = value; // sets value associated with this element
        }
```

SinglyLinkedList

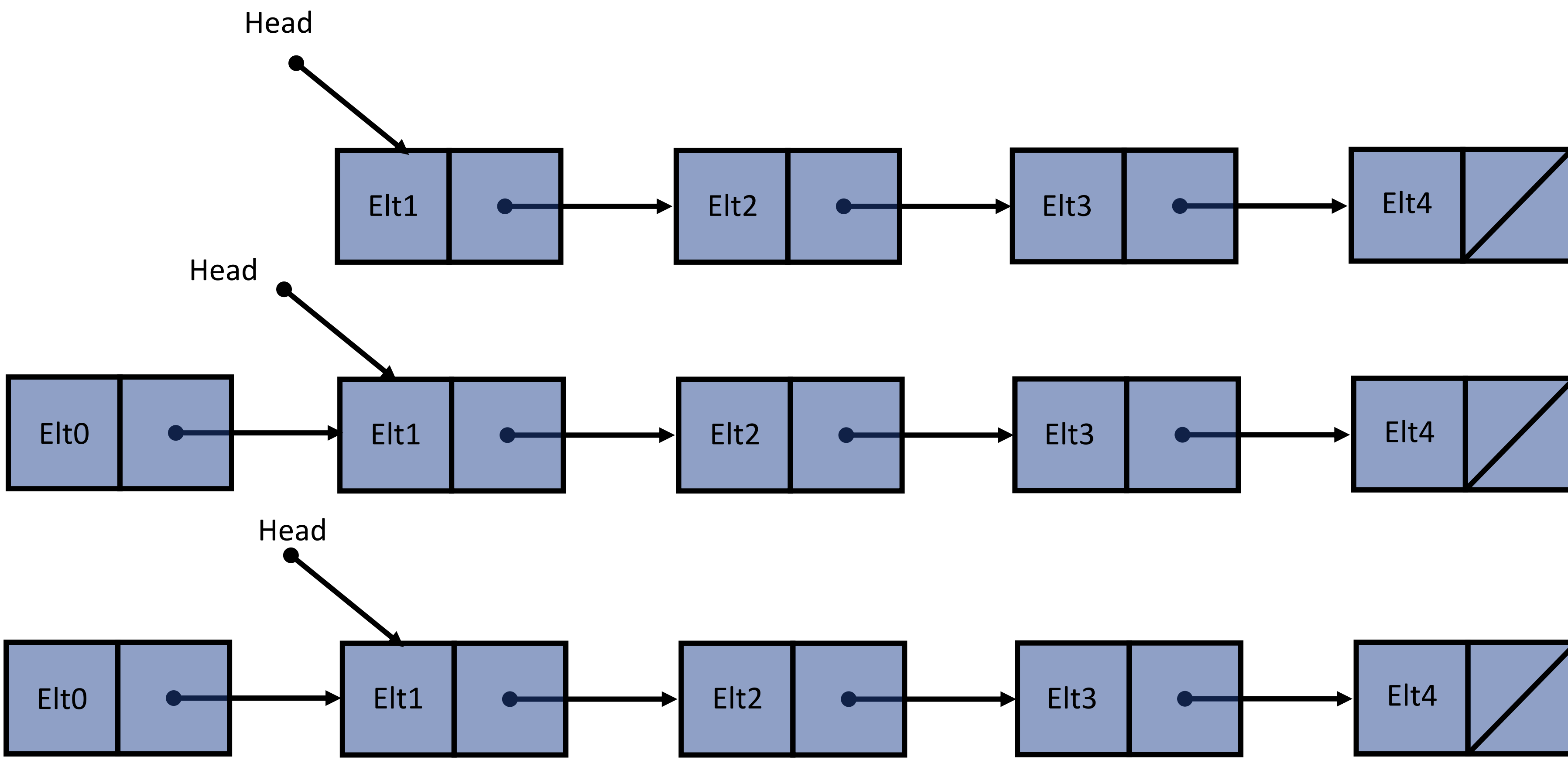
```
public class SinglyLinkedList<E> extends AbstractList<E> {
       protected int count; // list
       protected Node<E> head; // ref. to first element

       //construct an empty list
       public SinglyLinkedList() {
              head = null;
              count = 0;
        }
       public E getFirst() {
              return head.value(); //returns first value in list
       }
       public int size() {
              return count;
       }
```

Possible operations

A node can be added/removed:

- At the beginning of the list
- At the end of the list
- Between nodes
- In an empty list (addition only)

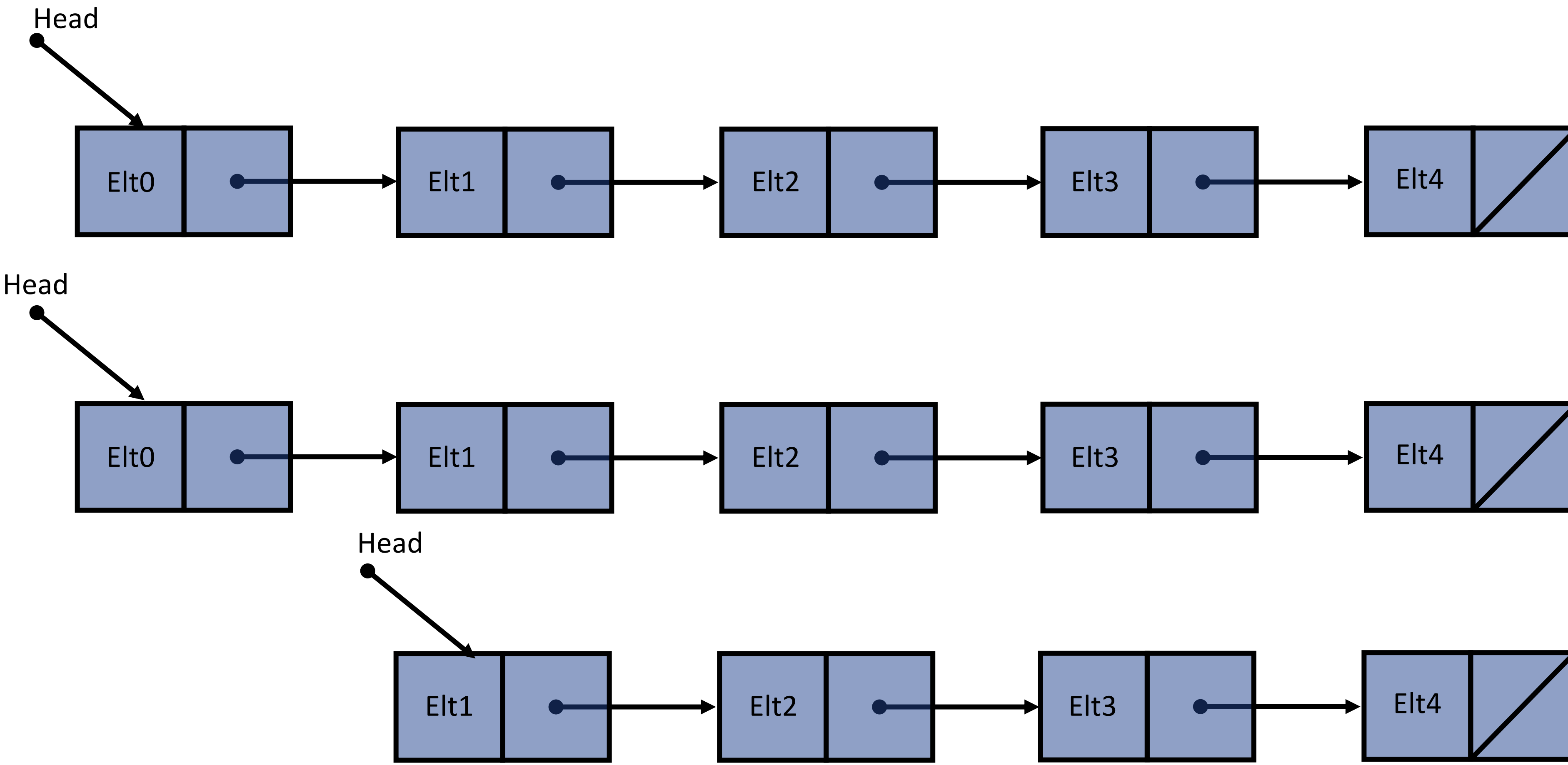
Adding at the head



Adding at the head is $O(1)$

```
public void addFirst(E value){
    // note order that things happen:
    // head is parameter, then assigned
    head = new Node<E>(value, head);
    count++;
}
```

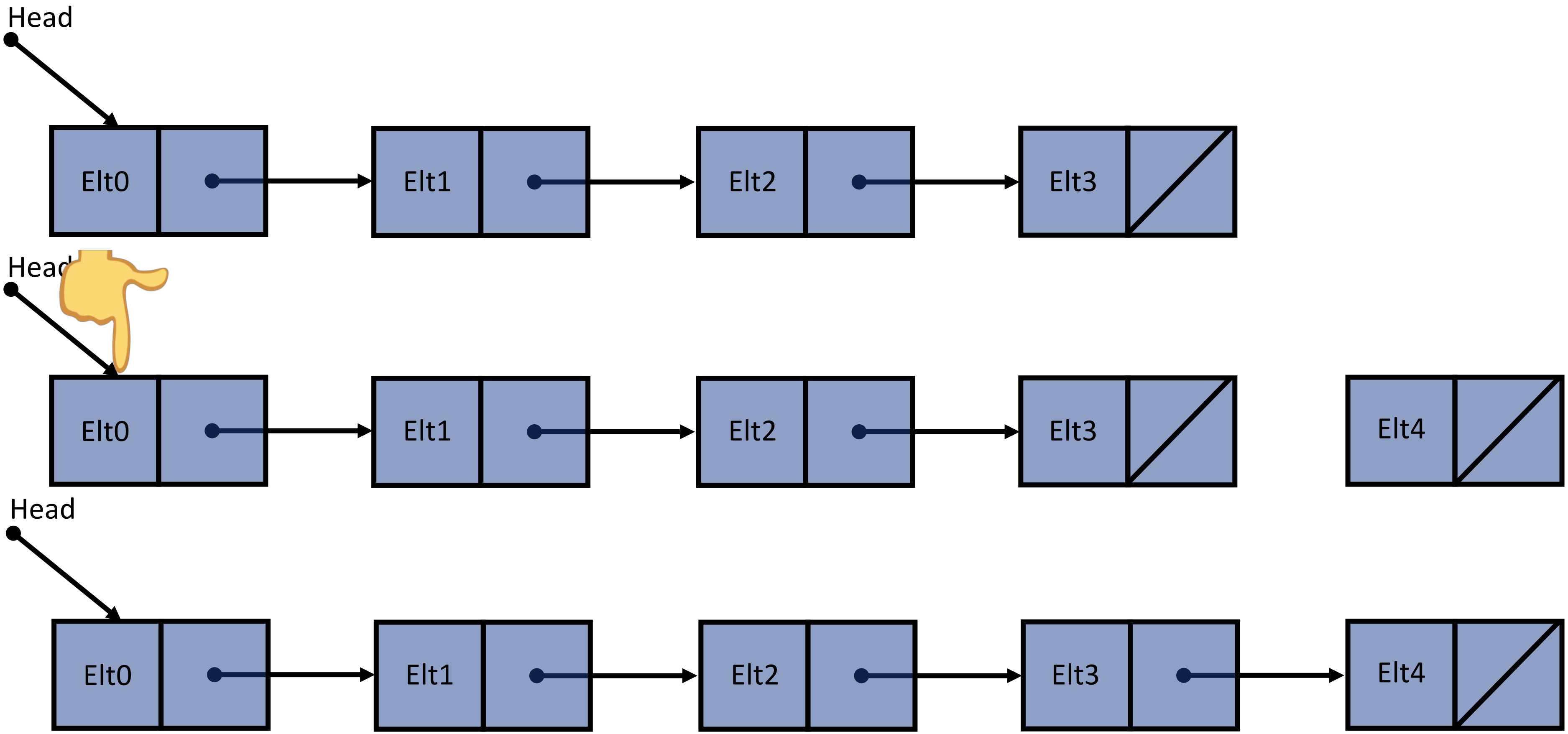
Removing at the head



Removing at the head is $O(1)$

```
public E removeFirst(){
    Node<E> temp = head;
    head = head.next(); // move head down list
    count--;
    return temp.value();
}
```

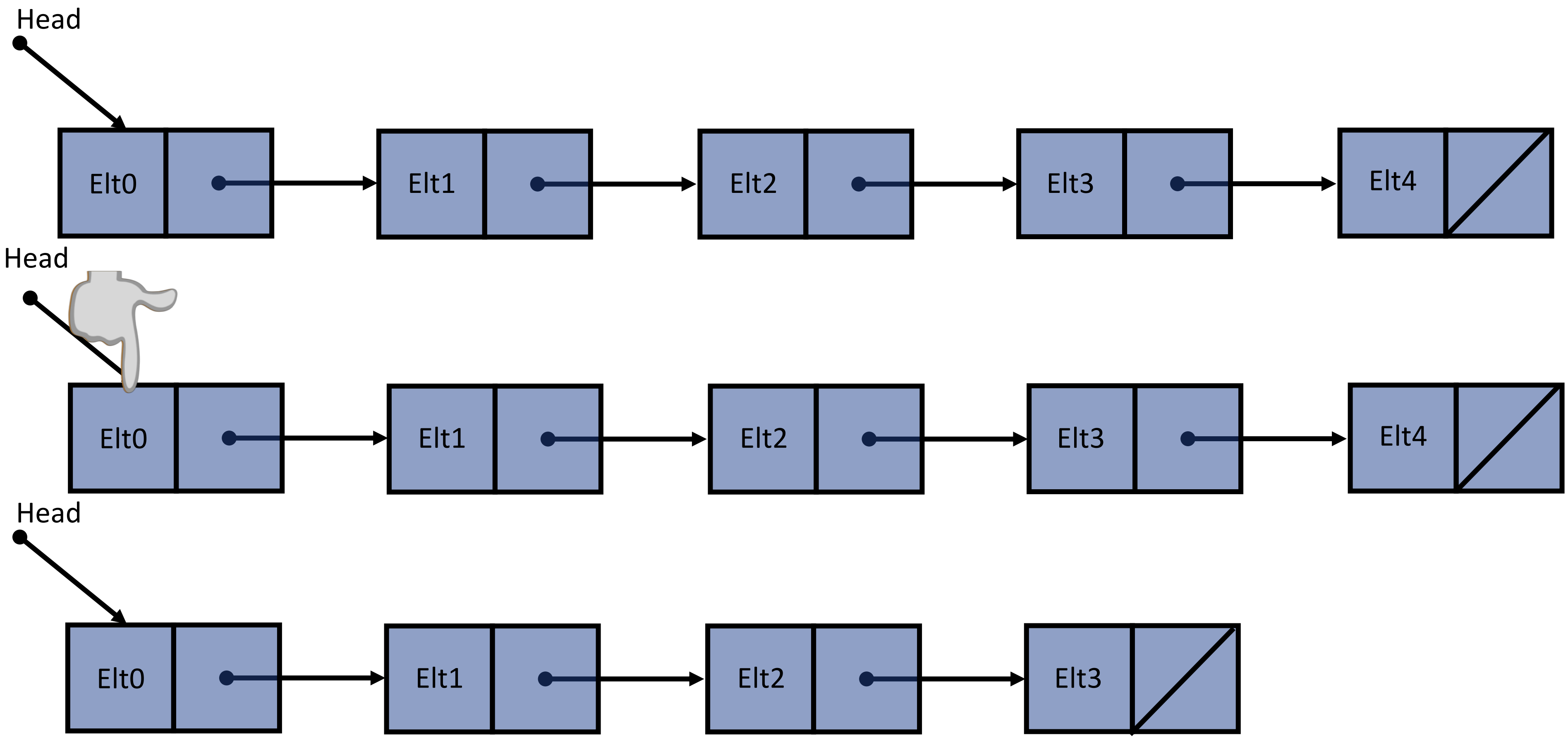
Adding at the end



Adding at the end is $O(n)$

```
public void addLast(E value) {
      // location for new value
      Node<E> temp = new Node<E>(value,null);
      if (head != null) {
            // pointer to possible tail
            Node<E> finger = head;
            while (finger.next() != null) {
                  finger = finger.next();
            }
           finger.setNext(temp);
      }
      else head = temp; //empty list
      count++;
}
```

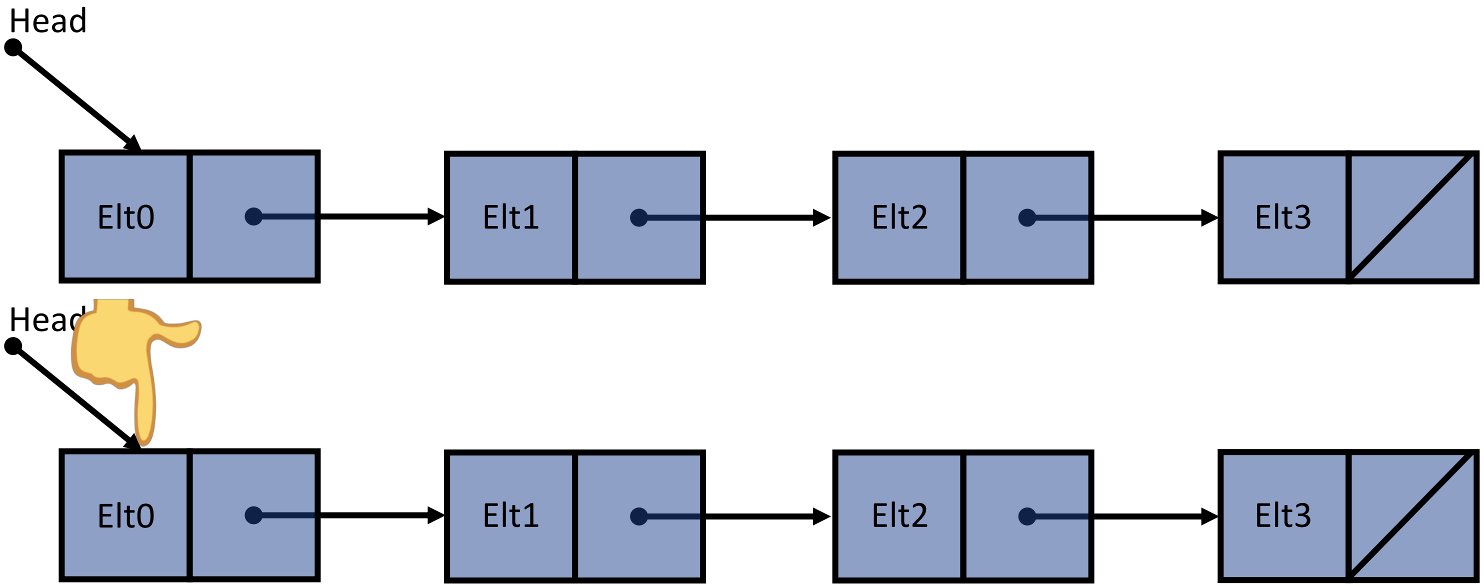
Removing from the end



Removing from the end is $O(n)$

```
public E removeLast() {
      Node<E> finger = head;
      Node<E> previous = null;
      //throw an exception if list is empty
      while (finger.next() != null)
      {
            previous = finger;      // find end of list
            finger = finger.next();
      }
      // finger is null, or points to end of list
      if (previous == null) {
            head = null;      // has exactly one element
      } else {
            previous.setNext(null); // pointer to last element is reset
      }
      count--;
      return finger.value();
}
```

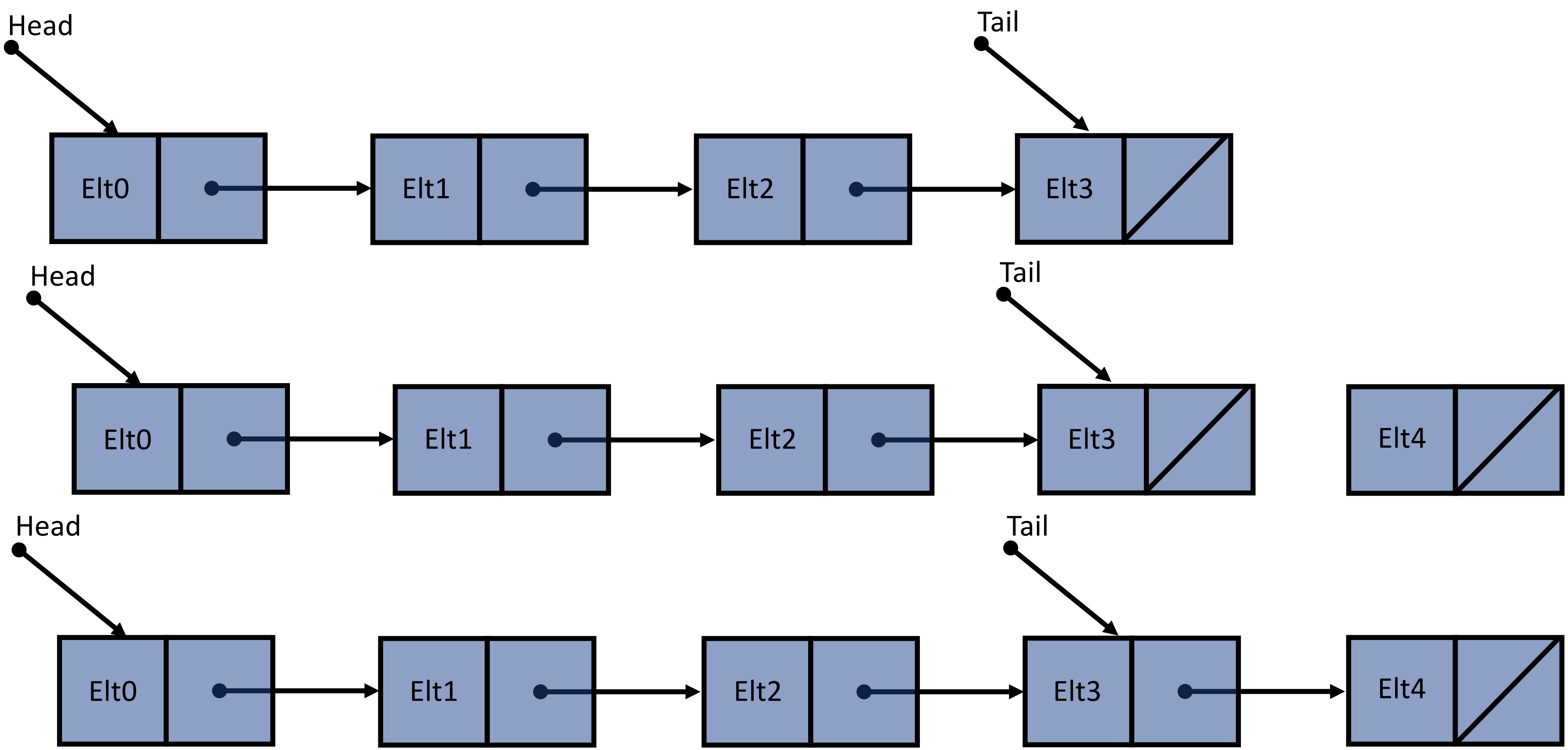
Getting the last element



Getting the last element is $O(n)$

```java
public E getLast() {
      Node<E> finger = head;
      //throw exception if list is empty
      while (finger != null && finger.next() != null)
      {
            finger = finger.next();
      }
      return finger.value();
}
```

Adding when having a tail pointer is $O(1)$

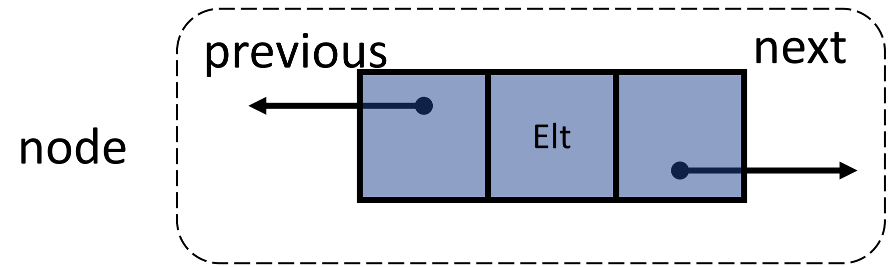


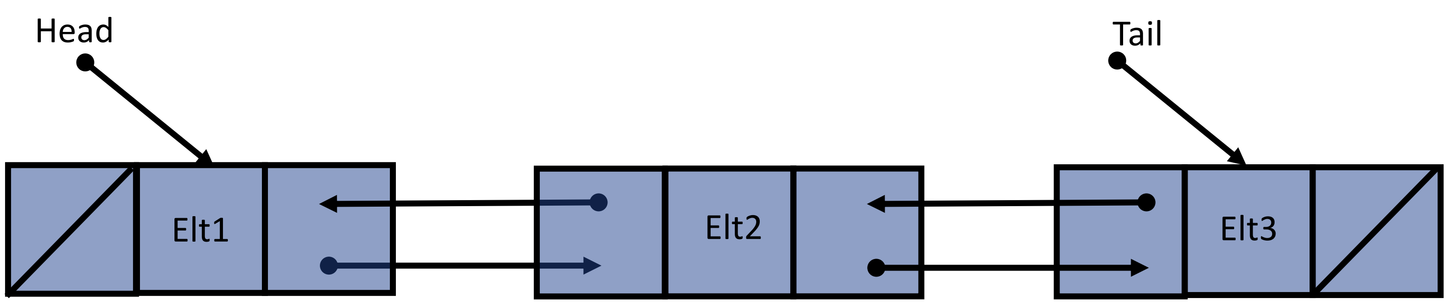
Removing at the tail is $O(n)$

- We still have to traverse the whole linked list – not efficient!

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





```java
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

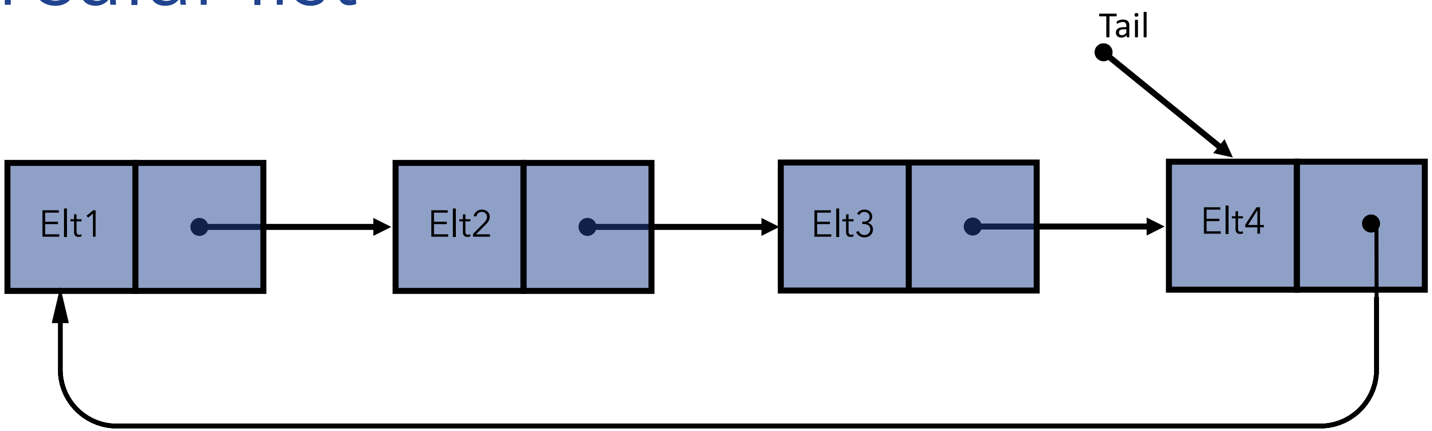
```
public class DoublyLinkedList<E> extends AbstractList<E> {
        protected int count; // number within list
        protected DoublyLinkedNode<E> head; // ref. to first element
        protected DoublyLinkedNode<E> tail; // ref. to last element

        //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;
        }
```

Assignment 4

```
for (int i = 0; i < myList.size(); i++) {
        myList.get(i);
}
```

- CurDoublyLinkedList
  - A doubly linked list with a head, a tail, and a cur
  - cur keeps track of most recently accessed element
  - Without cur above code is $O(n^2)$
  - With cur above code is $O(n)$
- Used to implement picture “compression”
- Think about edge cases and draw pictures of your linked lists!
  - Consider lists with 0, 1, or 2 elements.

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.