COMP20010

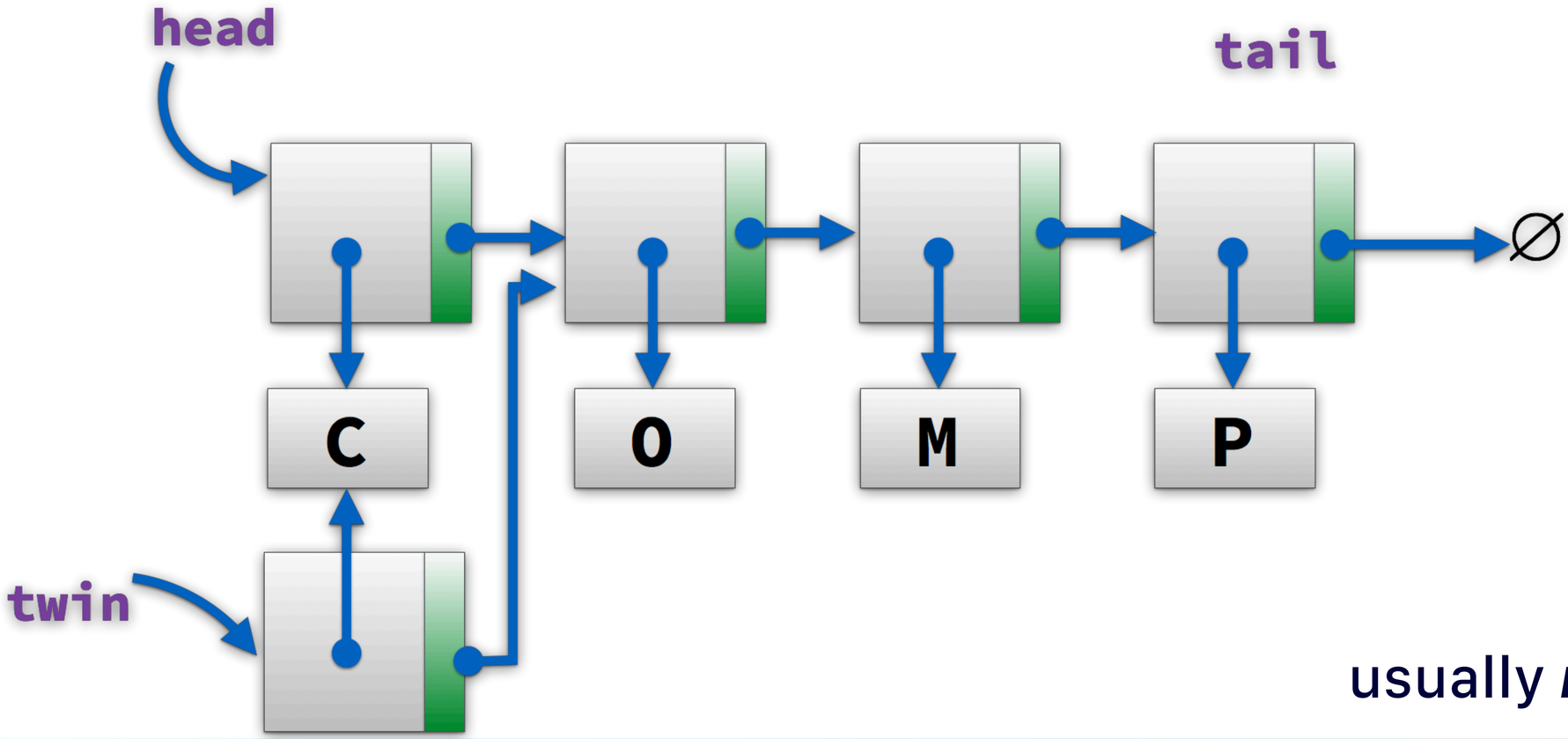
Data Structures and Algorithms I
03 - Linked Lists

Dr. Aonghus Lawlor
aonghus.lawlor@ucd.ie

Linked Lists II

LinkedList: Cloning

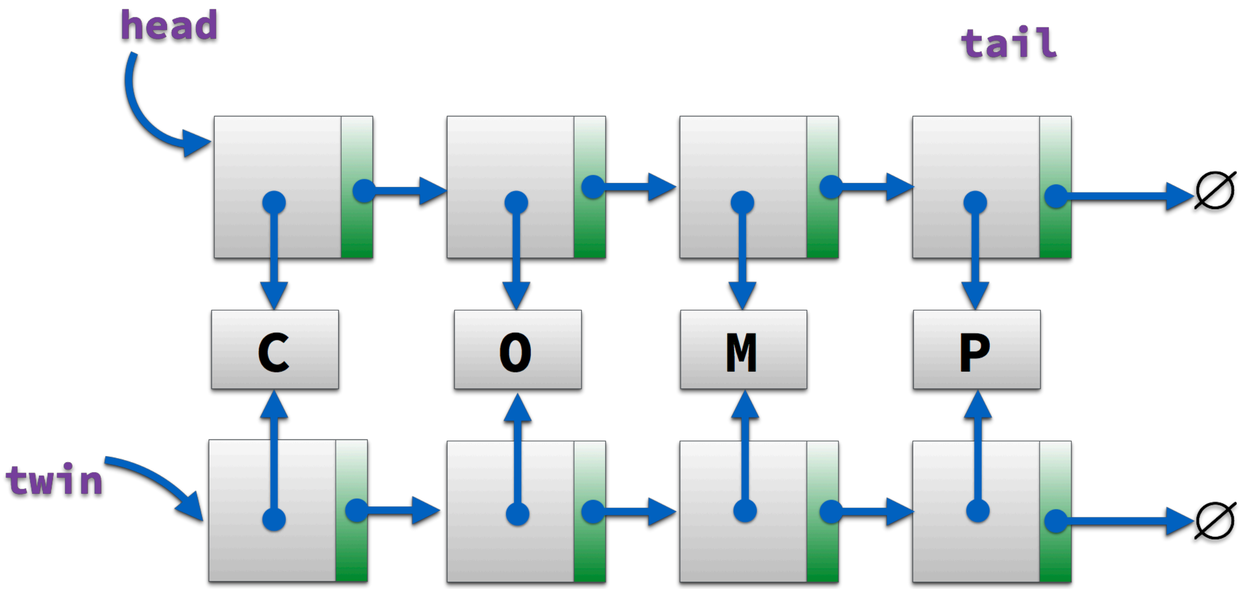
Like for any other objects, we need to learn how to clone linked lists. If we simply use the clone() method from the Object class, we will get the following structure called a "shallow" copy:



usually ***not*** what we want!

LinkedList: Cloning

Since our data is immutable it's ok to have data shared between two linked lists. There are a few ideas to implement linked list copying. The simplest one is to traverse the original list and copy each node by using the addLast() method:




```
public Object copy() {
  SinglyLinkedList<E> twin = new SinglyLinkedList<E>();
  Node<E> tmp = head;
  while (tmp != null) {
    twin.addLast(tmp.getElement());
    tmp = tmp.next;
  }
  return twin;
}
```

List ADT

- The List interface has the following methods:

size()	Returns the number of elements in the list
isEmpty()	Returns a boolean indicating whether the list is empty
get(i)	Returns the element of the list with index I, error condition occurs if i is outside the range [0, size()-1]
set(i, e)	Replaces the element at index i with e, and returns the old element that was replaced; an error occurs if i is not in range [0, size()-1]
add(i, e)	Inserts a new element e into the list at index i, moving all subsequent elements one index later in the list; an error occurs if i is not in range [0, size()-1]
remove(i)	Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error occurs if i is not in range [0, size()-1]

LinkedLists

- We can use the LinkedList to implement the List ADT

jdk8/java/util/ArrayList.java

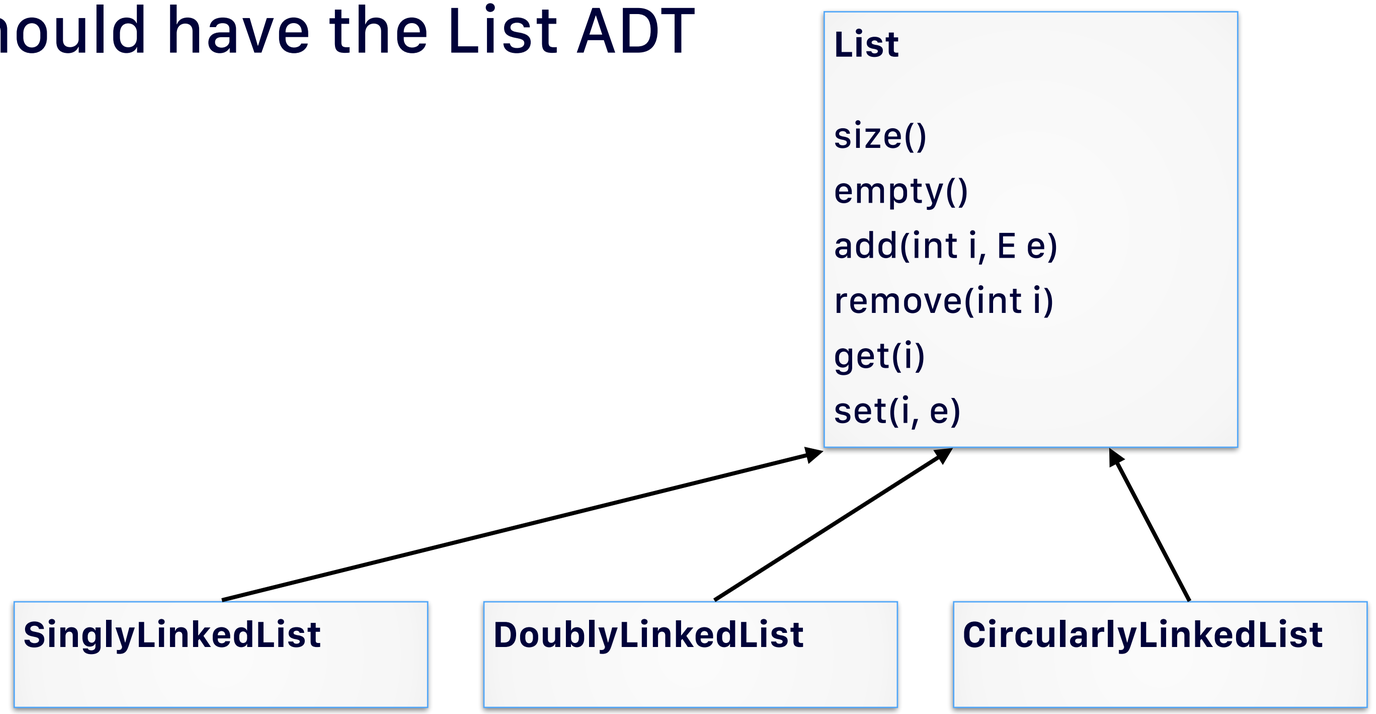
```
public class ArrayList<E> extends
AbstractList<E>
        implements List<E>, RandomAccess,
Cloneable, java.io.Serializable
{
```

jdk8/java/util/LinkedList.java

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>,
Cloneable, java.io.Serializable
{
```

List ADT

- A good strategy for implementing your own linked lists, is to have a base List class.
- The base class (or interface) should have the List ADT methods
- The subclasses will include
  - SinglyLinkedList
  - DoublyLinkedList
  - CircularlyLinkedList



LinkedLists

- When do we choose LinkedList or ArrayList>

Linked lists are preferable over arrays when:

a) you need constant-time insertions/ deletions from the list (such as in real-time computing where time predictability is absolutely critical)

b) you don't know how many items will be in the list. With arrays, you may need to re-declare and copy memory if the array grows too big

c) you don't need random access to any elements

d) you want to be able to insert items in the middle of the list (such as a priority queue)

Arrays are preferable when:

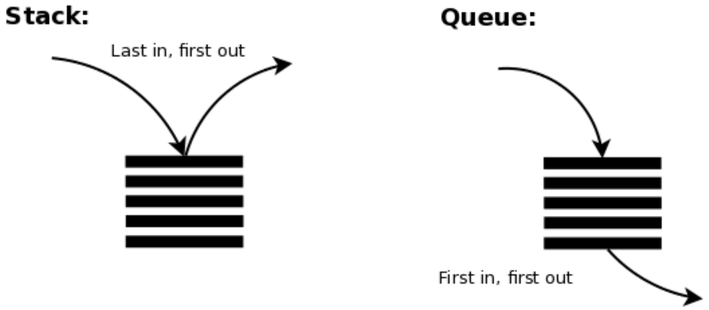
a) you need indexed/random access to elements

b) you know the number of elements in the array ahead of time so that you can allocate the correct amount of memory

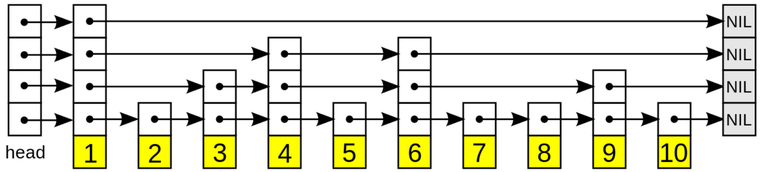
c) you need fast iteration (use pointer math).

d) Filled arrays take up less memory than linked lists. Each element in the array is just the data. Each linked list node requires the data as well as one (or more) pointers to the other elements in the linked list.

Linked Lists: Uses



stacks and **queues** are often implemented using linked lists, and simply restrict the type of operations which are supported.



skip lists are a linked-list-like structure which allows for fast search. It consists of a base list holding the elements, together with a tower of lists maintaining a linked hierarchy of subsequences, each skipping over fewer elements.

A hash table may use linked lists to store the chains of items that hash to the same position in the hash table.

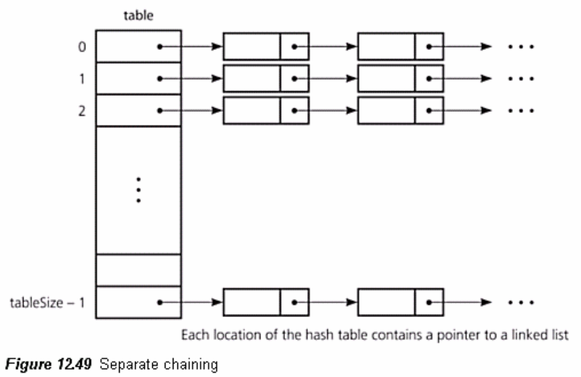
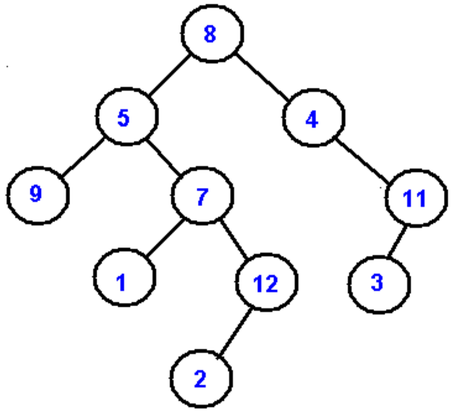


Figure 12.49 Separate chaining

A **binary tree** is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.



Variations on a Theme

Singly Linked List Limitations

Limitations of Singly-Linked Lists

- We can only move through it in one direction
- We need a pointer to the node before the spot where we want to insert and a pointer to the node before the node that needs to be deleted.
- Appending a value at the end requires that we step through the entire list to reach the end.

Doubly Linked List

A DLL is very similar to an SLL except that each node in a doubly linked list has references to both the node that follows it and the node that precedes it.

removeFront() is fast for a SLL, but removeLast() is not easy, since we do not have a quick way of accessing the node immediately preceding the one we want to remove (perhaps use a tail node).

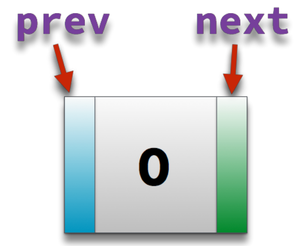
It would be nice to have a way of going both directions in a linked list.

The ***doubly linked*** list allows us to go in both directions—forward and reverse—in a linked list.

In addition to its element member, a node in a doubly linked list stores two pointers, a *next* link and a *prev* link, which point to the next and previous node in

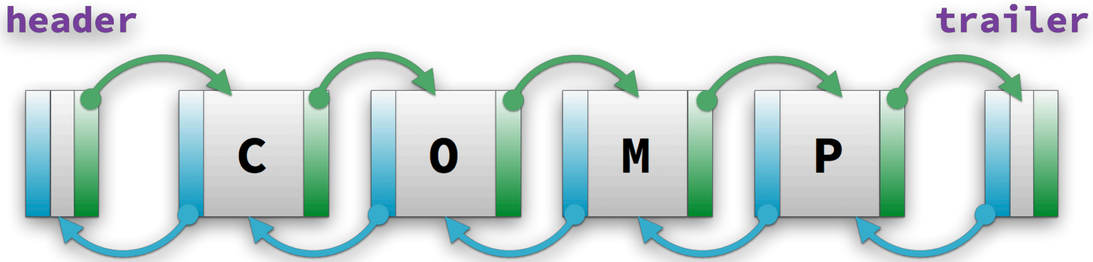
Doubly Linked List

To simplify programming, it is convenient to add special nodes at both ends of a doubly linked list: a ***header*** node just before the head of the list, and a ***trailer*** node just after the tail of the list.



These "dummy" or ***sentinel*** nodes do not store any elements.

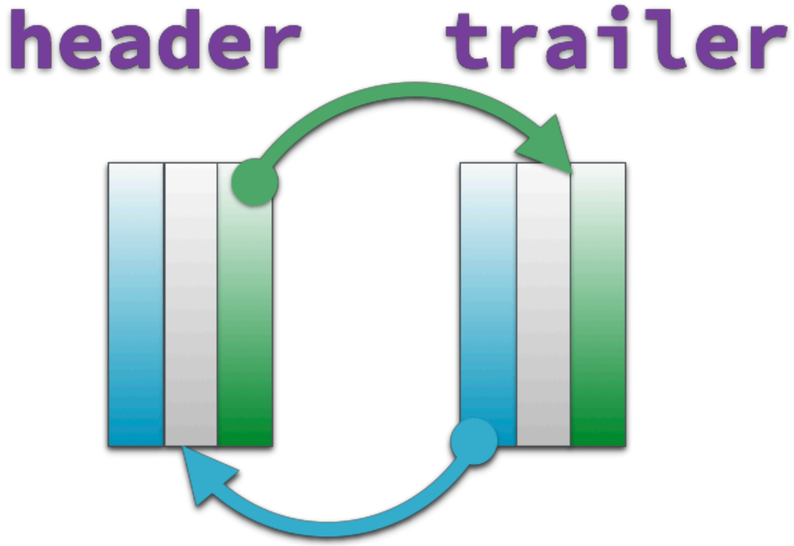
They provide quick access to the first and last nodes of the list. In particular, the header's *next* pointer points to the first node of the list, and the *prev* pointer of the trailer node points to the last node of the list.



Doubly Linked List

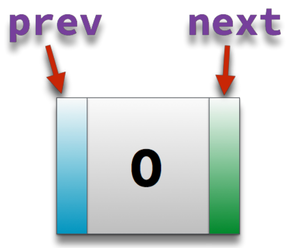
- Empty double linked list

- header == trailer



Doubly Linked List: Node

```
 private static class Node<E> {

    /** The element stored at this node */
    private E element;                // reference to the element stored at this
node

    /** A reference to the preceding node in the list */
    private Node<E> prev;             // reference to the previous node in the
list

    /** A reference to the subsequent 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;
    }
```



Doubly Linked List:

```
public class DoublyLinkedList<E> {

  /** Sentinel node at the beginning of the list */
  private Node<E> header;                     // header sentinel

  /** Sentinel node at the end of the list */
  private Node<E> trailer;                    // trailer sentinel

  /** Number of elements in the list (not including sentinels) */
  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
  }
```

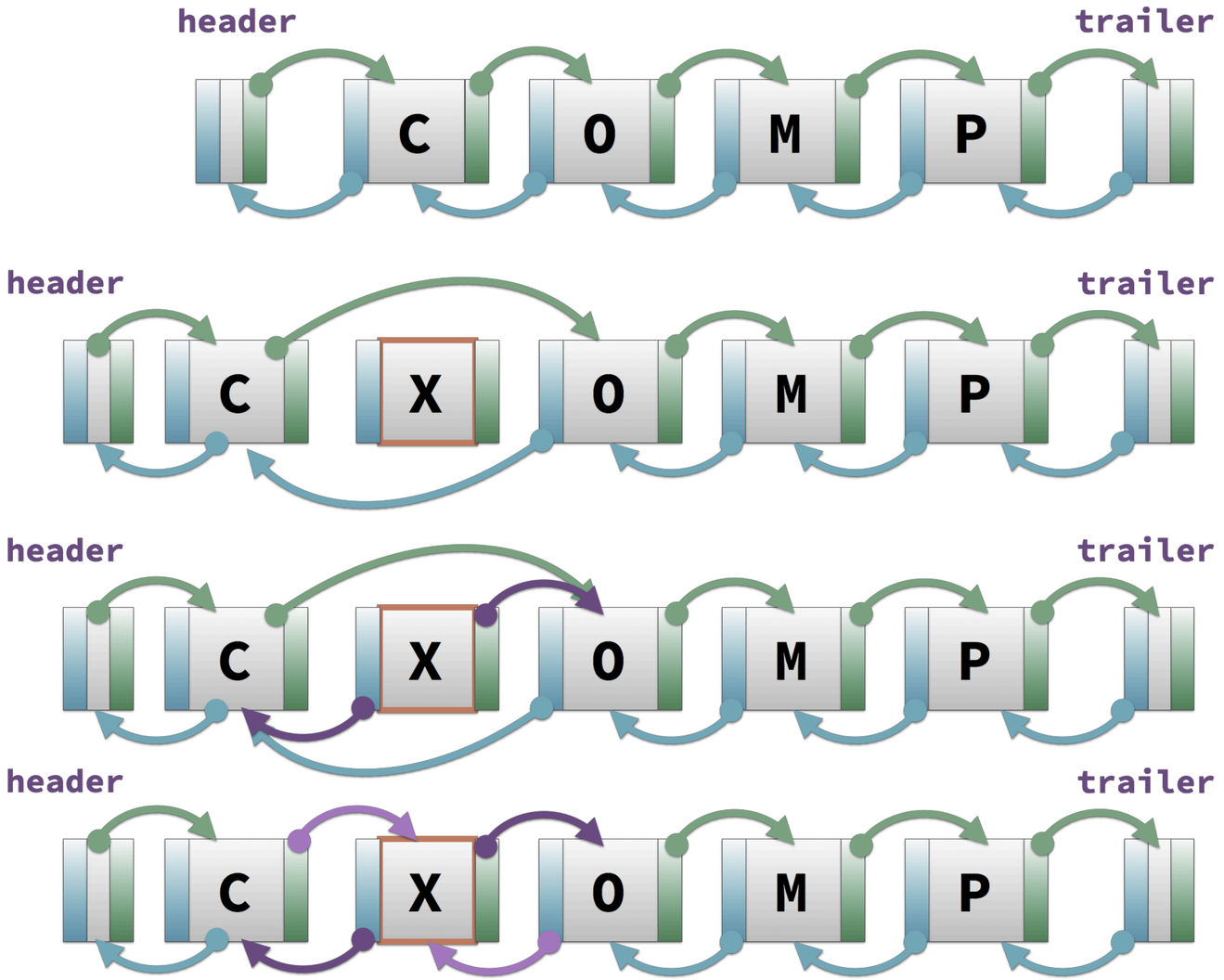
Doubly Linked List:

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

  /**
   * Tests whether the linked list is empty.
   * @return true if the linked list is empty,
false otherwise
   */
  public boolean isEmpty() { return size == 0; }

  /**
   * Returns (but does not remove) the first
element of the list.
   * @return element at the front of the list (or
null if empty)
   */
  public E first() {
    if (isEmpty()) return null;
    return header.getNext().getElement();
 }
```

```
  /**
   * Returns (but does not remove) the last
element of the list.
   * @return element at the end of the list (or
null if empty)
   */
  public E last() {
    if (isEmpty()) return null;
    return trailer.getPrev().getElement();
}
```

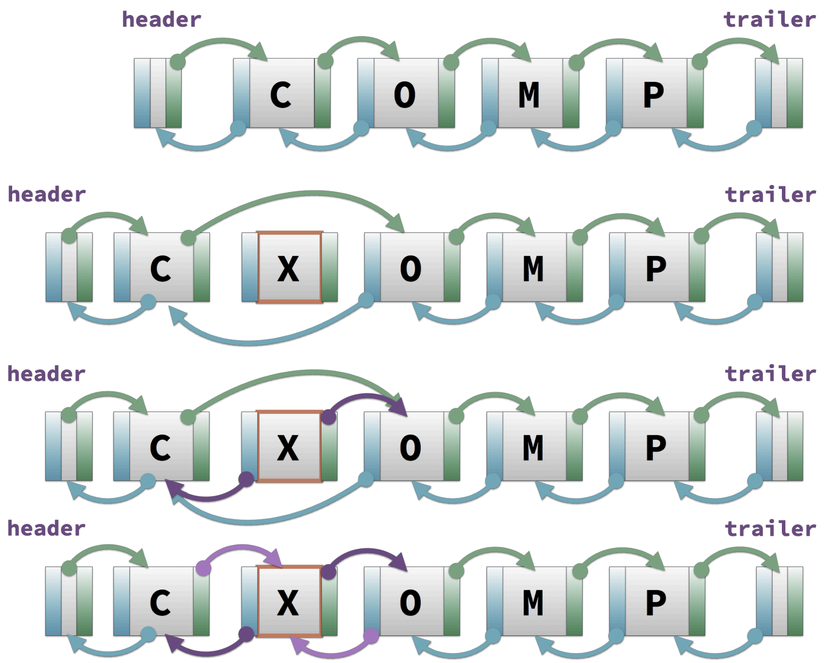
Doubly Linked List add

If we have a reference to a node in a DLList and we want to insert a node, then this is just a matter of setting the *next* and *prev* pointers, and then adjusting the adjacent pointers. Because of the dummy nodes, there is no need to worry about *next* or *prev* not existing.



Doubly Linked List add

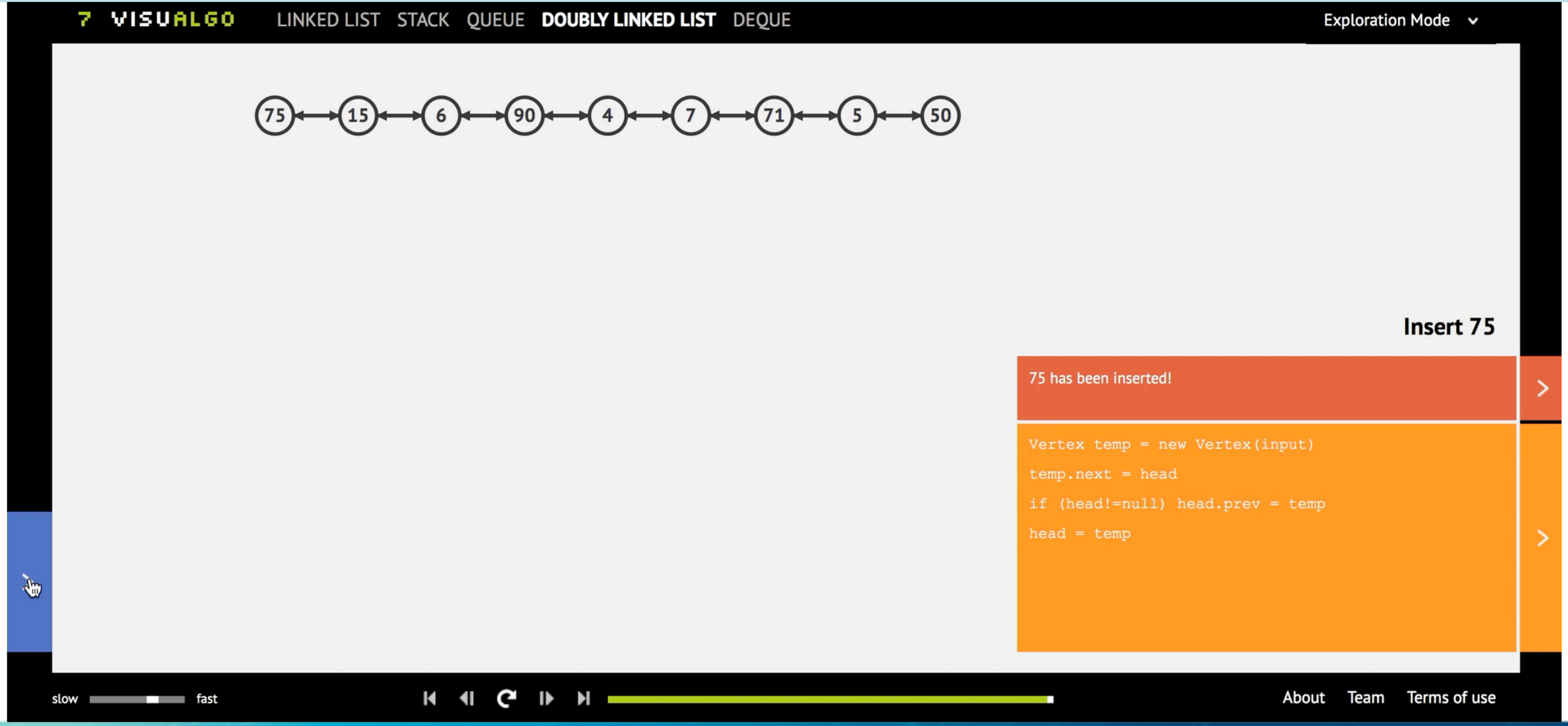
```
private void addBetween(E e,
        Node<E> predecessor,
        Node<E> successor) {
  // create and link a new node
  Node<E> newest = new Node<>(e, predecessor, successor);
  predecessor.setNext(newest);
  successor.setPrev(newest);
  size++;
}
```



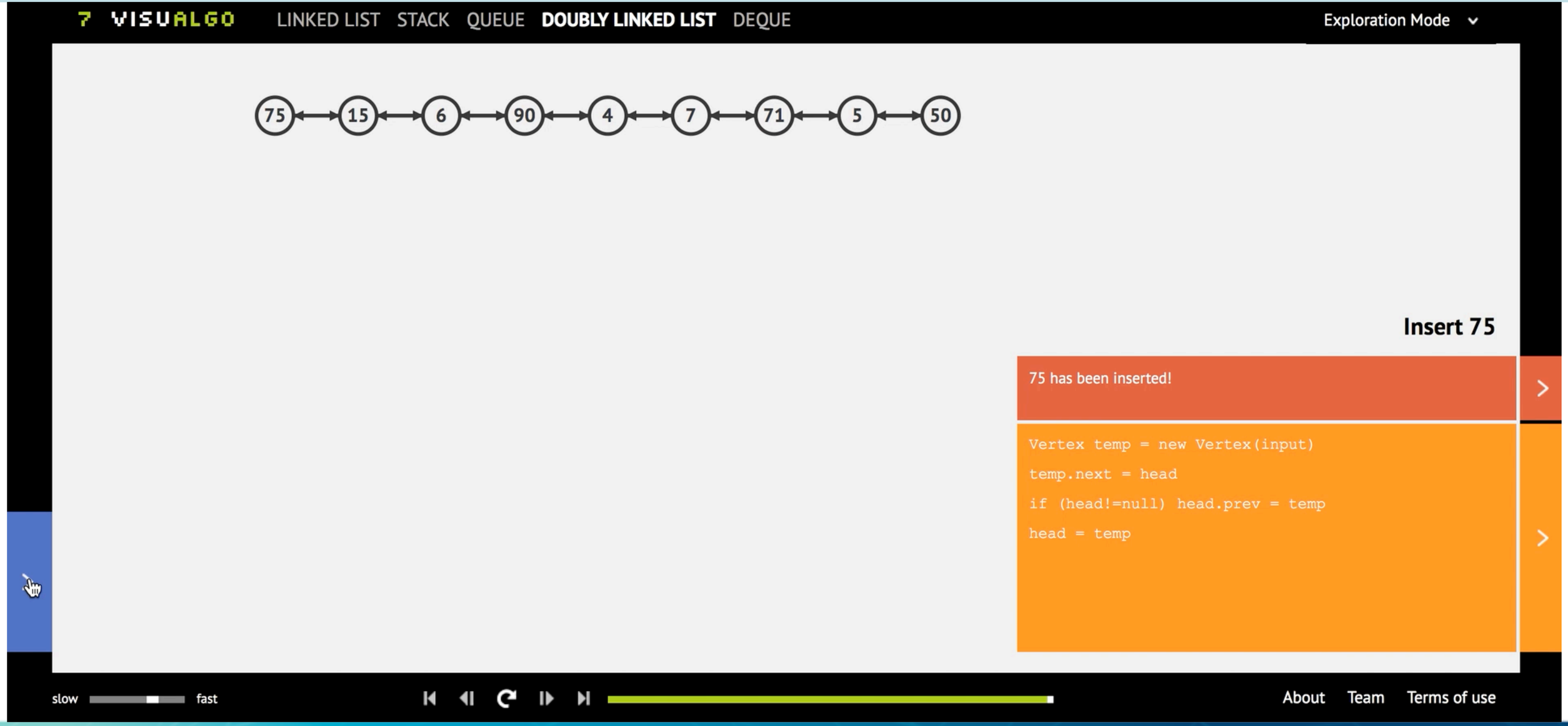
Doubly Linked List remove

```
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();
}
```

Doubly Linked List

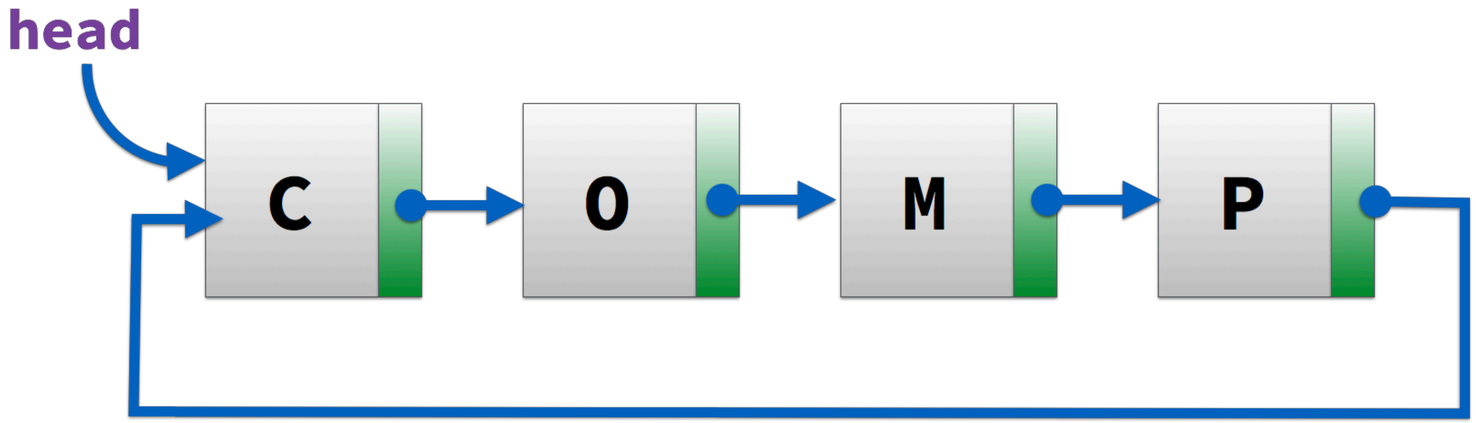


Doubly Linked List



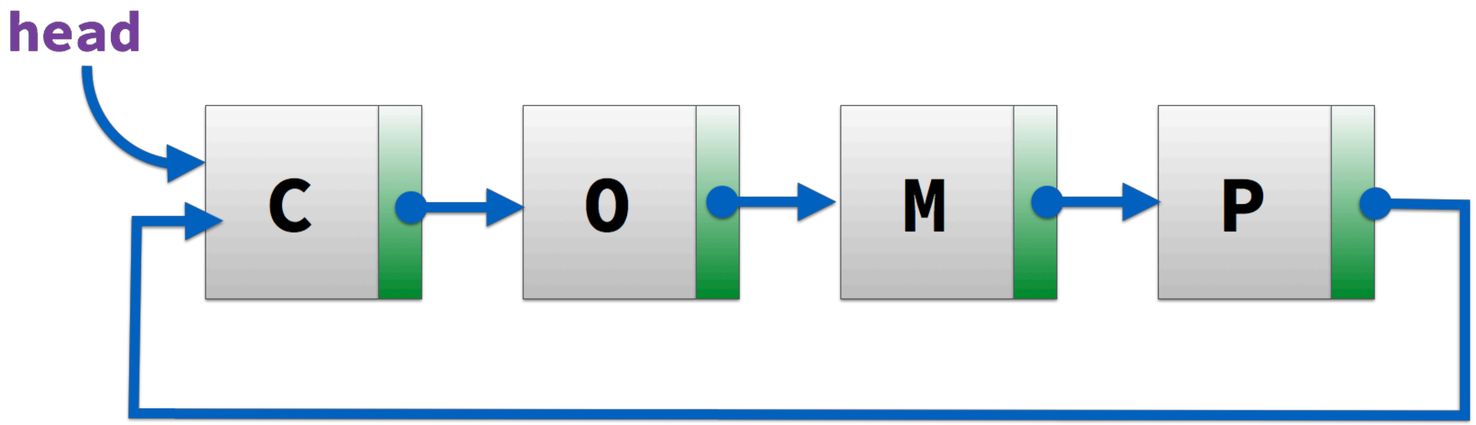
Circularly Linked List

- Almost
  identical to a Singly Linked List
- There is no pointer to null at the end
- Instead the end points back to the head
- there is no beginning
- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.



Circularly Linked List

- When do we use circularly linked lists:
- any application where a pointer to any node serves as a handle to the whole list
- arrays that are naturally circular, e.g. the corners of a polygon, a pool of buffers that are used and released in FIFO ("first in, first out") order, or a set of processes that should be time-shared in round-robin order..
- With a circular list, a pointer to the last node gives easy access also to the first node, by following one link- can be



Circularly Linked List

```
public class CircularlyLinkedList<E> {
 private static class Node<E> {
  // ...
 }

  // instance variables of the CircularlyLinkedList
  /** The designated cursor of the list */
  private Node<E> tail = null;                    // we store tail (but not head)

  /** Number of nodes in the list */
  private int size = 0;                           // number of nodes in the list

  /** Constructs an initially empty list. */
  public CircularlyLinkedList() { }               // constructs an initially empty list

  // access methods
  /**
   * Returns the number of elements in the linked list.
   * @return number of elements in the linked list
   */
  public int size() { return size; }

  public boolean isEmpty() { return size == 0; }
```

```
/**
   * Returns (but does not remove) the first element of the list
   * @return element at the front of the list (or null if empty)
   */
  public E first() {              // returns (but does not remove) the first element
    if (isEmpty()) return null;
    return tail.getNext().getElement();          // the head is *after* the tail
  }

  /**
   * Returns (but does not remove) the last element of the list
   * @return element at the back of the list (or null if empty)
   */
  public E last() {               // returns (but does not remove) the last element
    if (isEmpty()) return null;
    return tail.getElement();
  }
```

Circularly Linked List

```
// update methods
/**
 * Rotate the first element to the back of the list.
 */
public void rotate() {          // rotate the first element to the back of the list
  if (tail != null)                 // if empty, do nothing
    tail = tail.getNext();          // the old head becomes the new tail
}

/**
 * Adds an element to the front of the list.
 * @param e  the new element to add
 */
public void addFirst(E e) {                 // adds element e to the front of the list
  if (size == 0) {
    tail = new Node<>(e, null);
    tail.setNext(tail);                      // link to itself circularly
  } else {
    Node<E> newest = new Node<>(e, tail.getNext());
    tail.setNext(newest);
  }
  size++;
}

/**
 * Adds an element to the end of the list.
 * @param e  the new element to add
 */
public void addLast(E e) {                  // adds element e to the end of the list
  addFirst(e);              // insert new element at front of list
  tail = tail.getNext();    // now new element becomes the tail
}
```

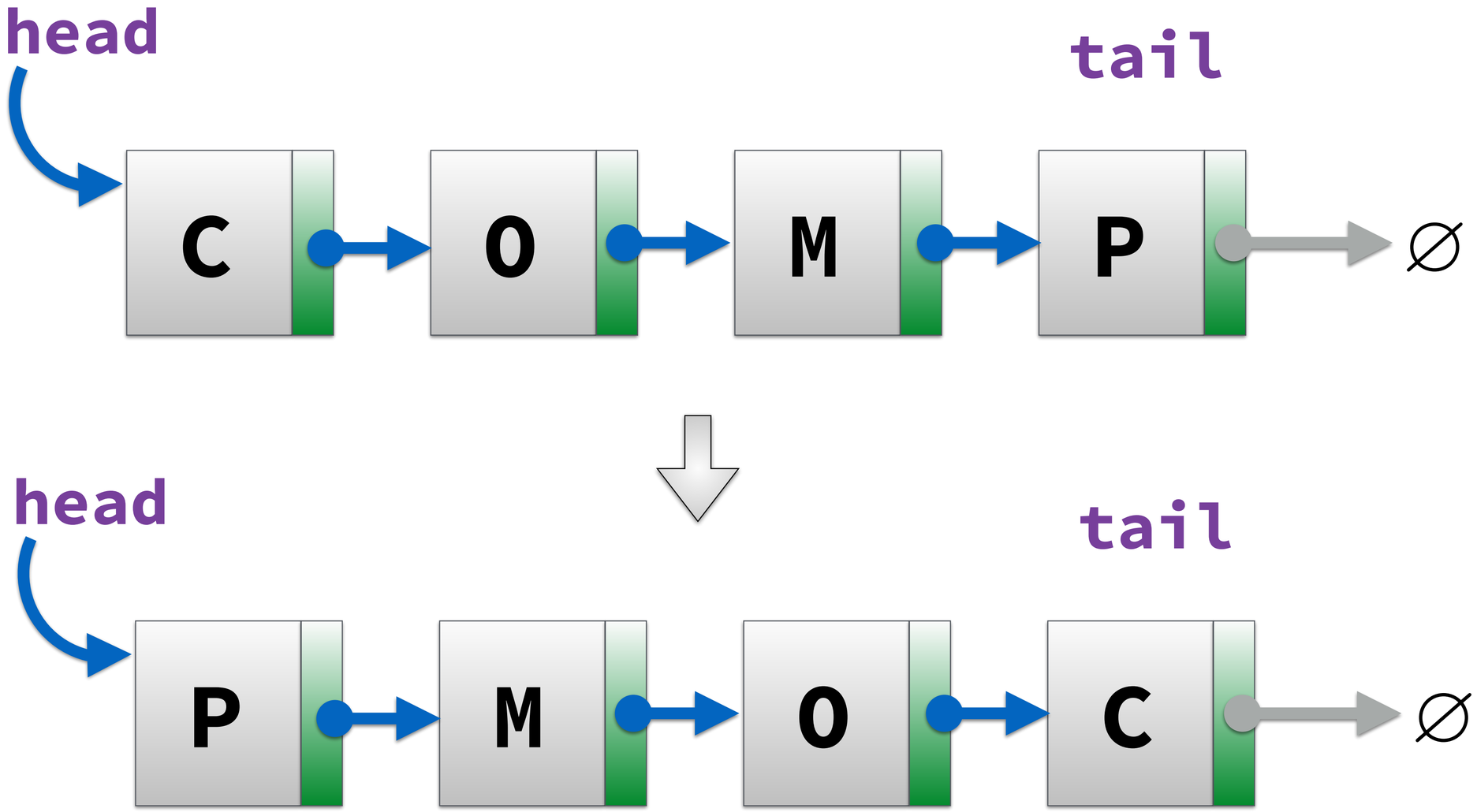
```
/**
  * Removes and returns the first element of the list.
  * @return the removed element (or null if empty)
  */
 public E removeFirst() {             // removes and returns the first element
   if (isEmpty()) return null;                // nothing to remove
   Node<E> head = tail.getNext();
   if (head == tail) tail = null;             // must be the only node left
   else tail.setNext(head.getNext());         // removes "head" from the list
   size--;
   return head.getElement();
 }
```

Linked List Algorithms

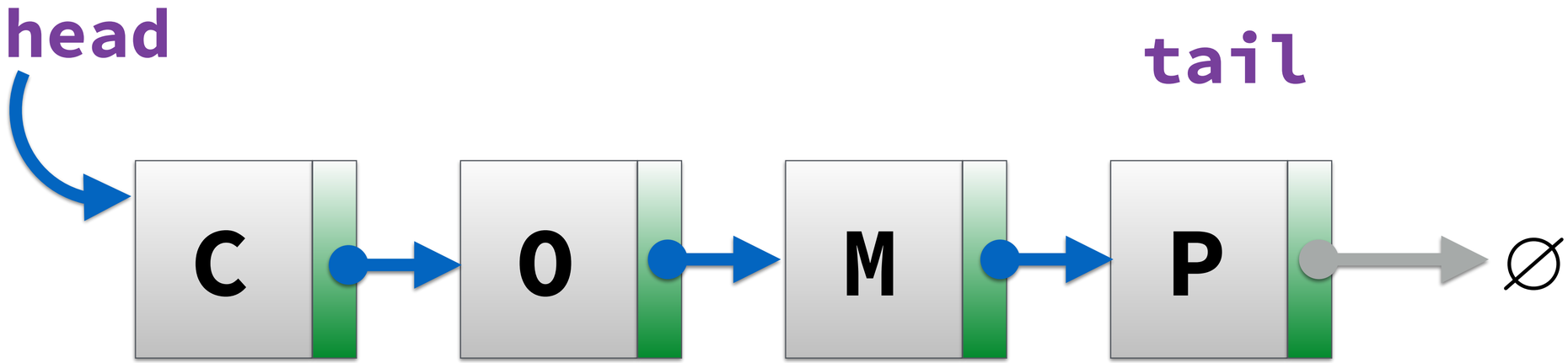
Reversing a Linked List

- The task is reverse the order of the list
- Not allowed to create new nodes
- There are well known iterative and recursive methods

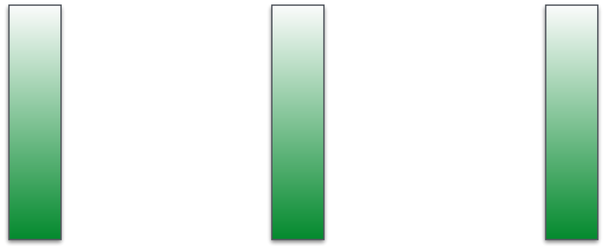
Reversing a Linked List



Reversing a Linked List

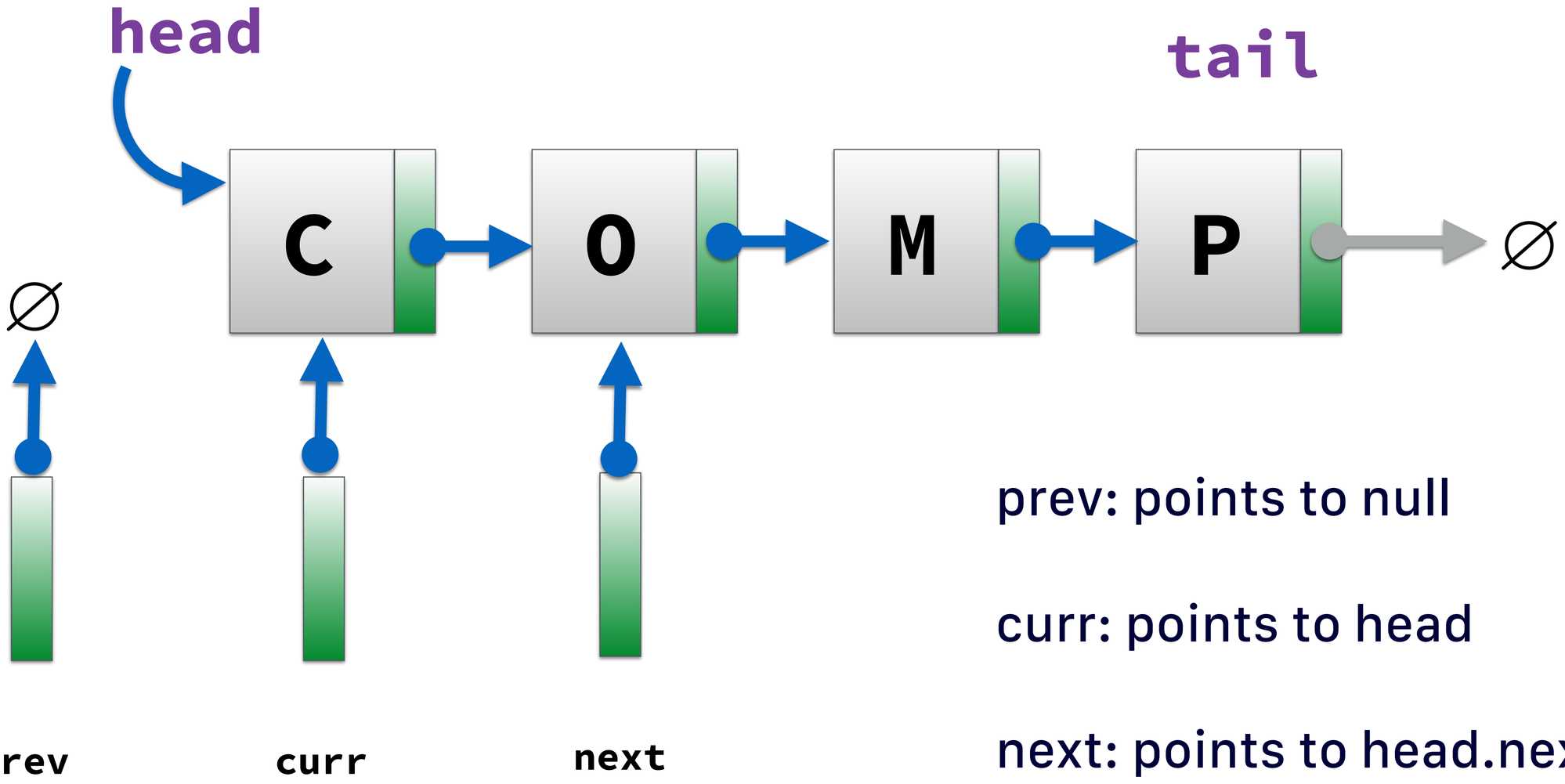


Define 3 new pointers:



`prev` `curr` `next`

Reversing a Linked List

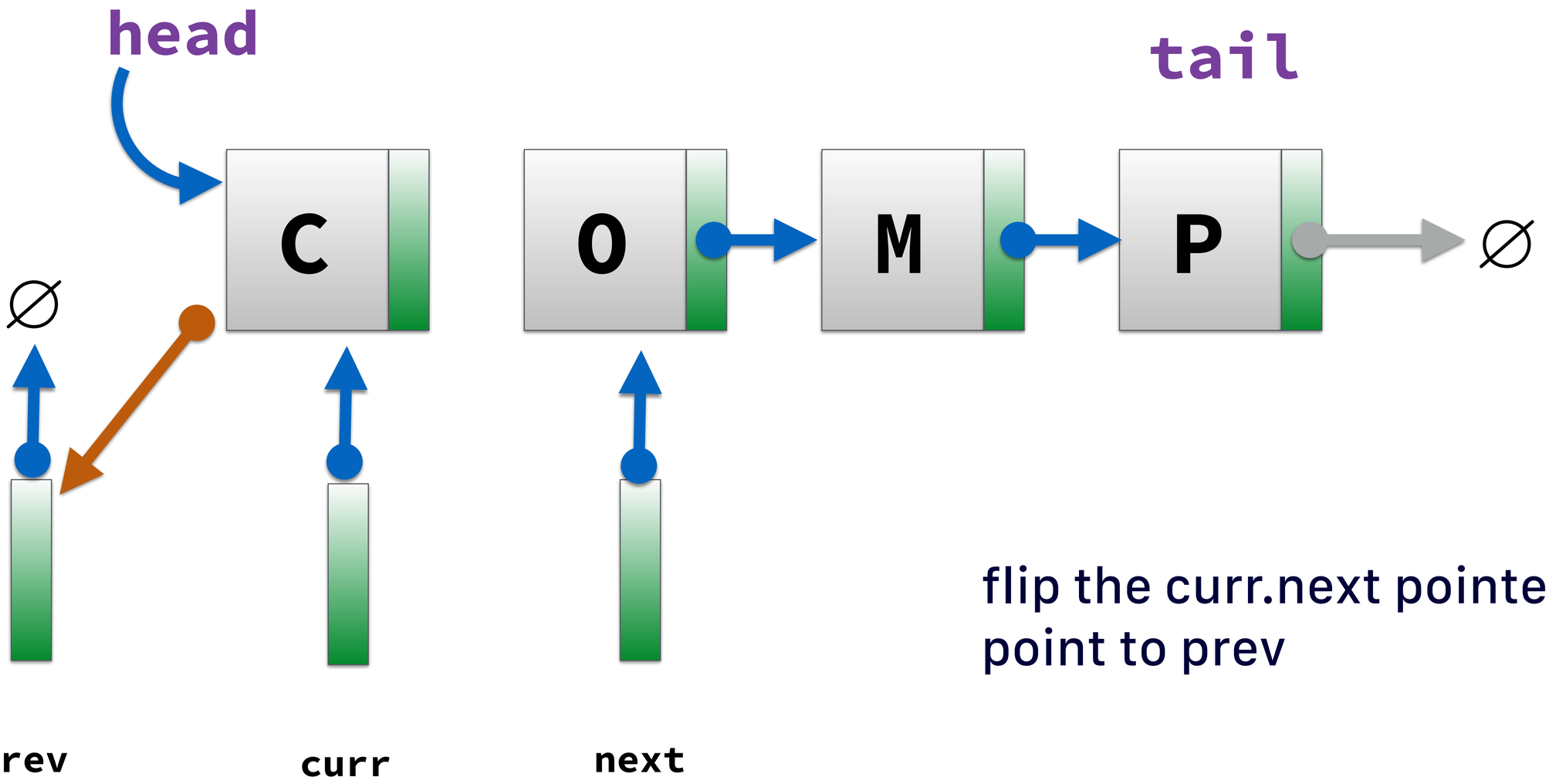


prev: points to null

curr: points to head

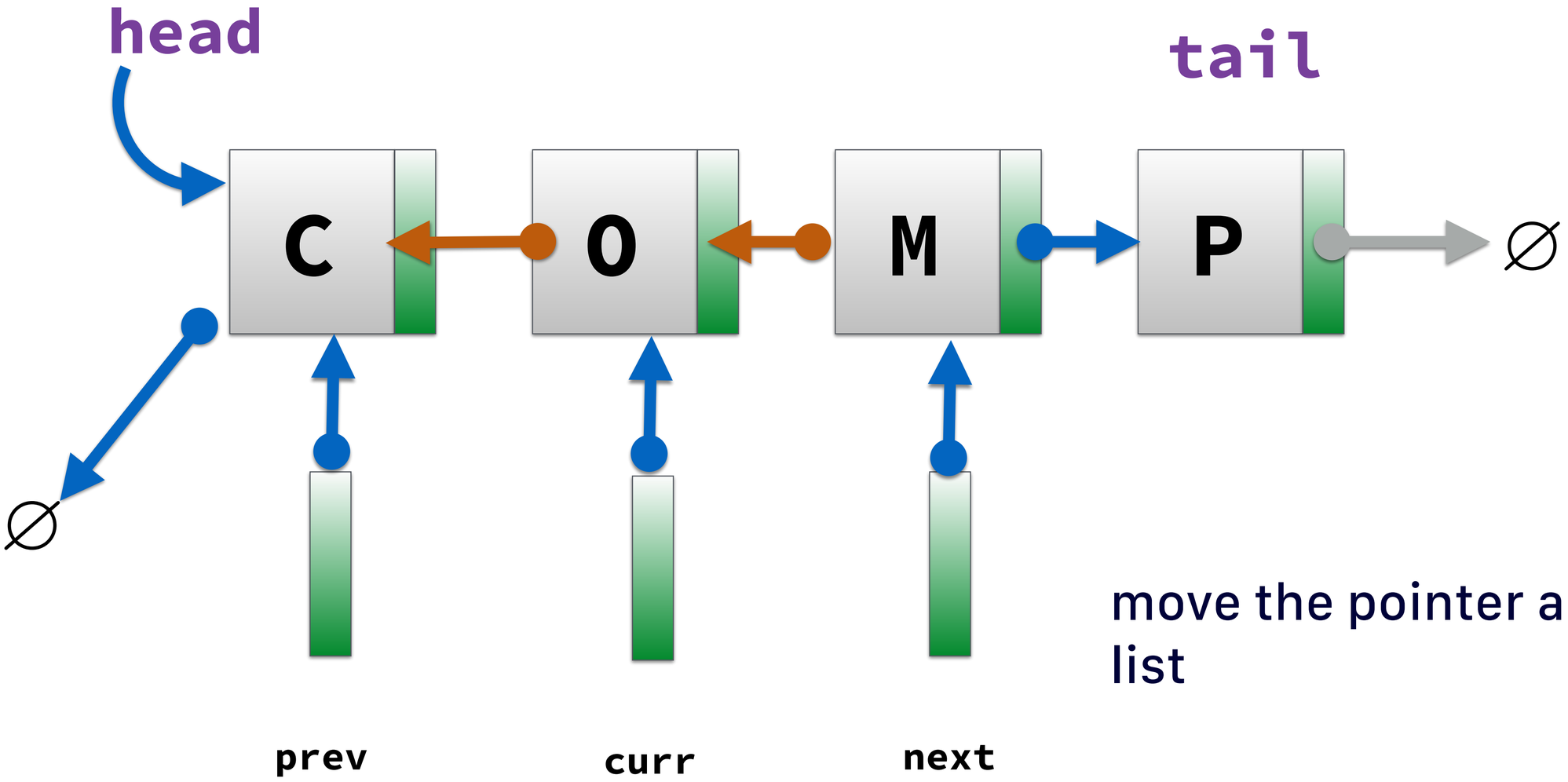
next: points to head.next

Reversing a Linked List



flip the curr.next pointer to point to prev

Reversing a Linked List



move the pointer along the list

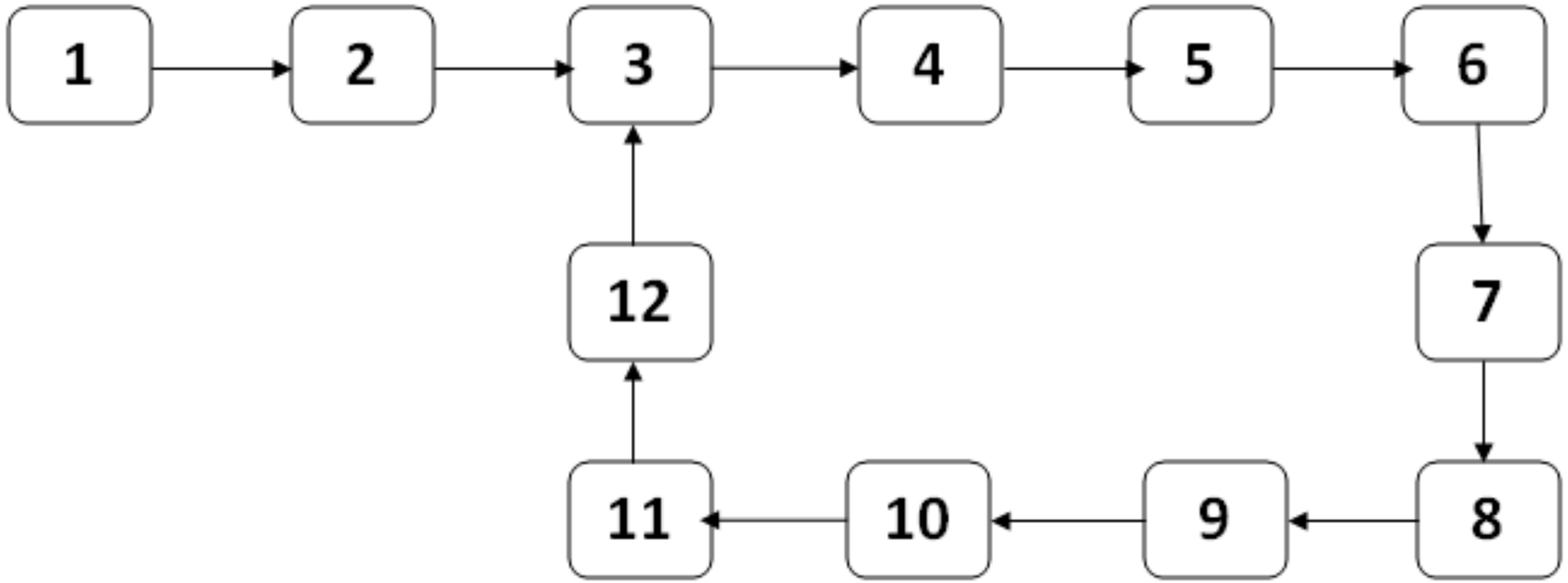
Reversing a Linked List

```
public void reverse() {
    Node<E> prev = null;
    Node<E> curr = head;
    Node<E> next;

    while(curr != null) {
        next = curr.getNext();
        current.setNext(prev);
        prev = curr;
        curr = next;
    }
    head = prev;
}
```

Detecting Loops in a Linked List

- Our linked list classes typically have a link to the first node (head).
- Common operations on a singly linked list are iterating through all the nodes, adding to the list, or deleting from the list. Algorithms for these operations require a well formed linked list.
- A linked list should *not* have loops or cycles in it.
- If a linked list has a cycle:
  - The malformed linked list has no end (no node ever has a null next_node pointer)
  - The malformed linked list contains two links to some node
  - Iterating through the malformed linked list will yield all nodes in the loop multiple times
  - A malformed linked list with a loop causes iteration over the list to fail because the iteration will never reach the end of the list. Therefore, it is desirable to be able to detect that a linked list is malformed before trying an iteration. This article is a discussion of various algorithms to detect a loop in a singly linked list.

Detecting Loops in a Linked List

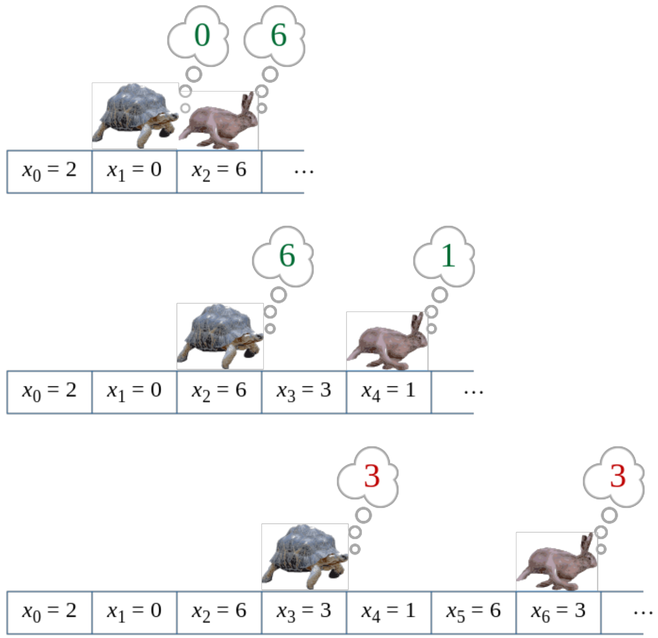


How can we detect loops like this in a linked list?

Can we do it efficiently?

Detecting Loops in a Linked List

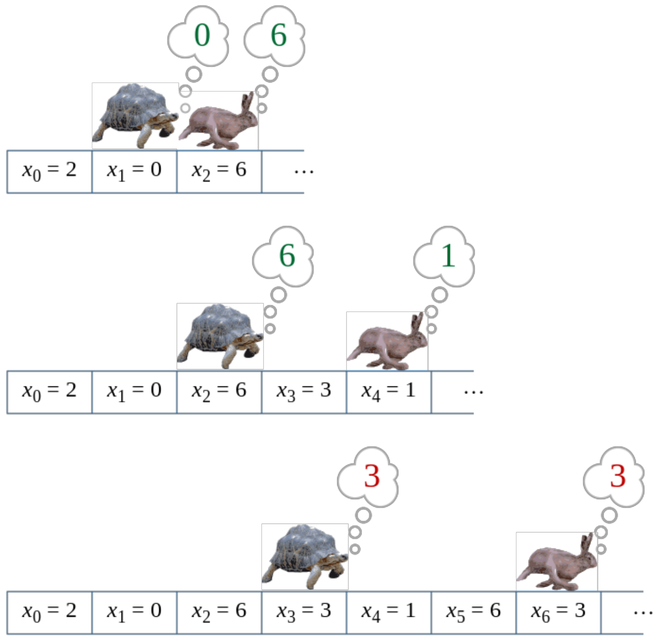
- You can detect a loop by simply running *two* pointers through the list
- The pointers move a different speeds
- this process is known as the tortoise and hare algorithm
- Also known as Floyd's cycle-finding algorithm.



1. Two pointers, the slow pointer and fast pointer are initially be set to the head of the linked list.
2. 'slow' moves at one node per iteration while 'fast' moves two nodes per iteration.
3. If at some step of the iteration, the two pointers 'fast' and 'slow' point to the same node, then the loop is detected.

Detecting Loops in a Linked List

- You can detect a loop by simply running *two* pointers through the list
- The pointers move a different speeds
- this process is known as the tortoise and hare algorithm
- Also known as Floyd's cycle-finding algorithm.




```
boolean detectLoop() {
    Node<E> slow = head;
    Node<E> fast = head;
    while (slow != null &&
           fast != null &&
           fast.getNext() != null) {
        slow = slow.getNext();
        fast = fast.getNext().getNext();
        if (slow == fast) {
            System.out.println("Found loop");
            return true;
        }
    }
    return false;
}
```