COMP20010

Data Structures and Algorithms I
03 - Linked Lists

Dr. Aonghus Lawlor
aonghus.lawlor@ucd.ie

Outline

- List ADT
- Singly Linked Lists
- Doubly Linked Lists
- Circularly Linked Lists

List ADT

List ADT

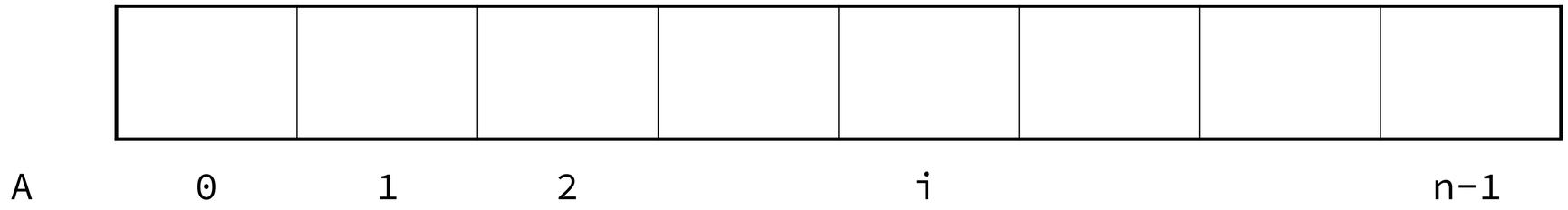
- The List interface has the following methods:

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]

Array List

- An obvious choice for implementing the list ADT is to use an array, **A**, where **A[i]** stores (a reference to) the element with index **i**.

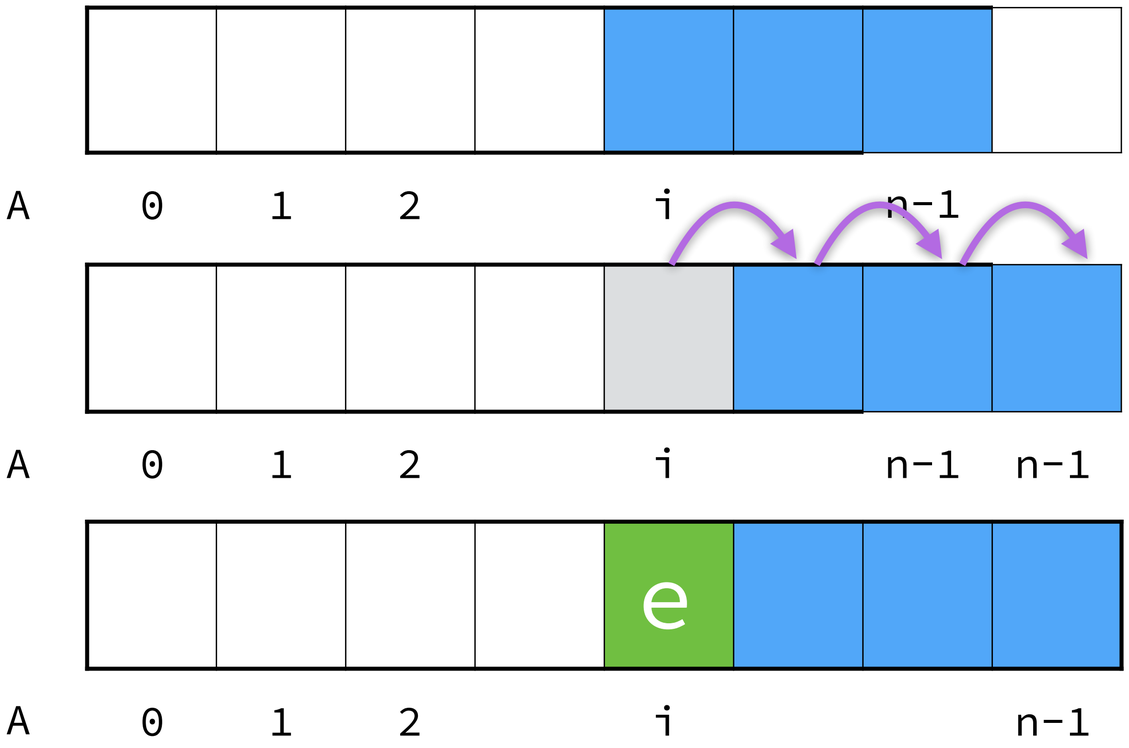
- With a representation based on an array **A**, the get(**i**) and set(**i**, **e**) methods are easy to implement by accessing **A[i]** (assuming **i** is a legitimate index).



Array List - Insertion

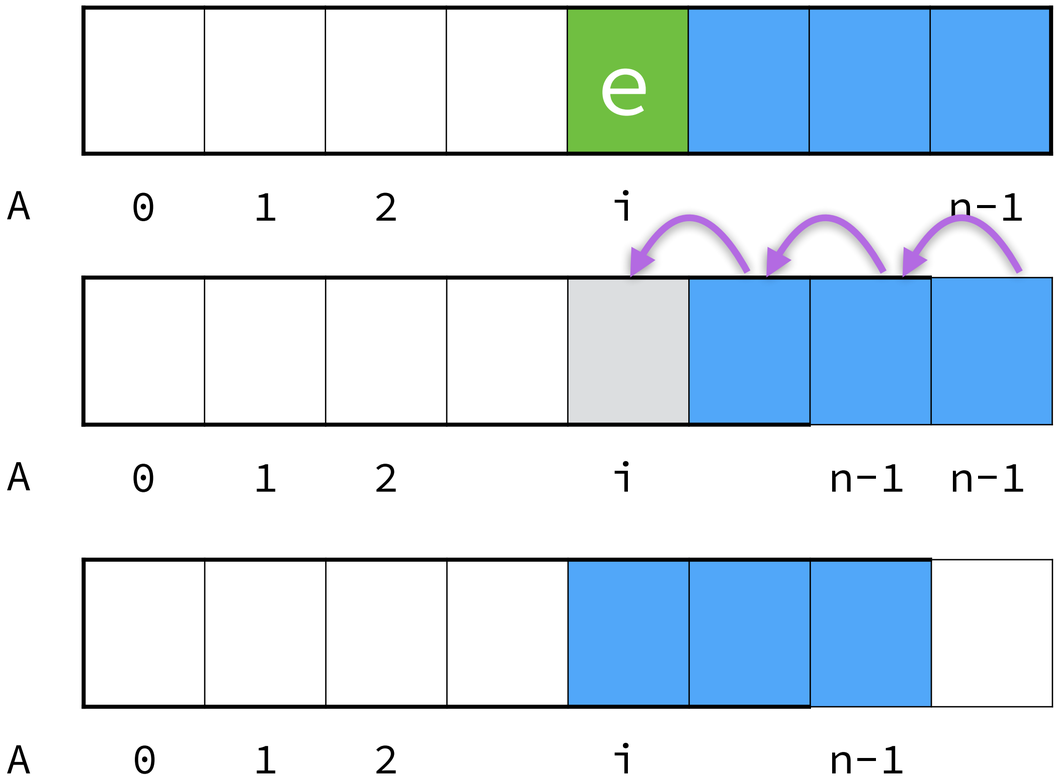
In an operation add(i, e), we need to make room for the new element by shifting forward the n - i elements A[i], ..., A[n - 1]

In the worst case (i = 0), this takes O(n) time



In an operation remove(i), we need to fill the space left by the removed element by shifting backwards the n - i -1 elements A[i+1], ..., A[n - 1]

In the worst case (i = 0), this takes O(n) time



Problems With Arrays

Rank is not the only way of referring to the place where an element appears in a sequence.

Problems: Fixed size of N and use of integer indices to access its contents.

If we were to use a linked list to implement a sequence, then it would be more natural to use elements (nodes) to describe the place where an element appears.

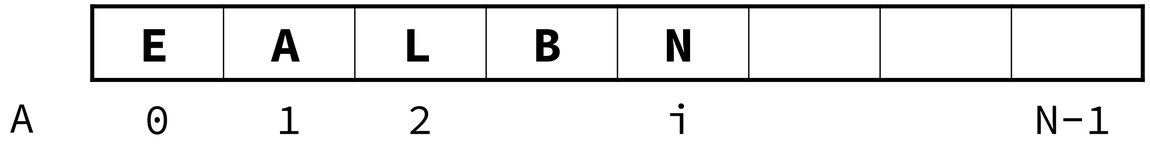
For example:

X should be inserted after Y.

Linked Structures

An array is a collection of variables stored sequentially in memory.

```
char[] A = new char[N];
char[0] = ‘E’;
...
```

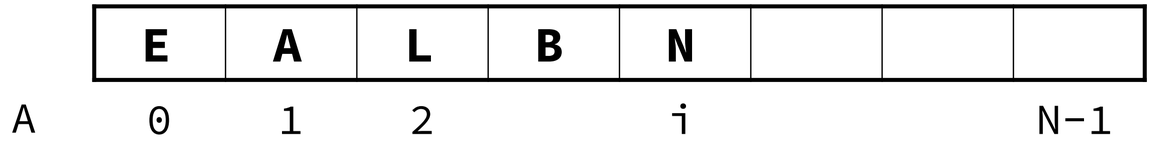


- arrays are sequential chunks of memory
- require contiguous space of size N*sizeof(object) to be available

Linked Structures

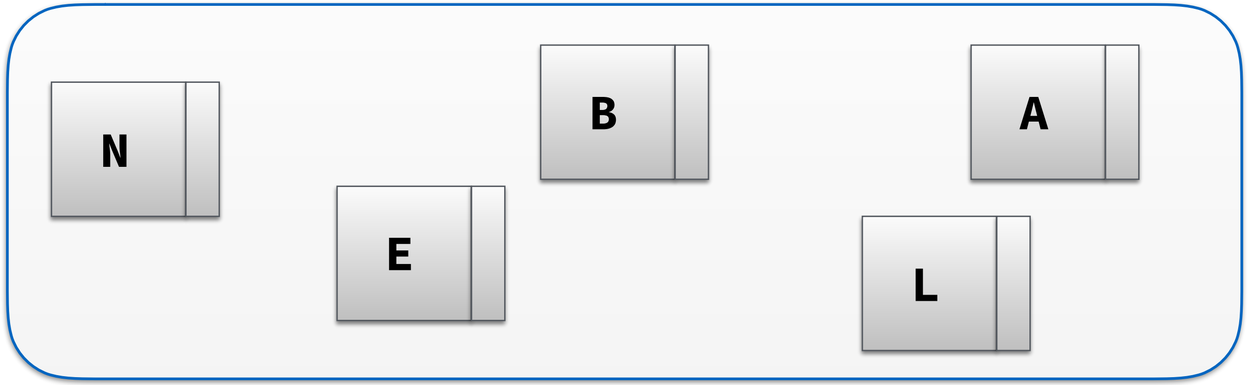
An array is a collection of variables stored sequentially in memory.

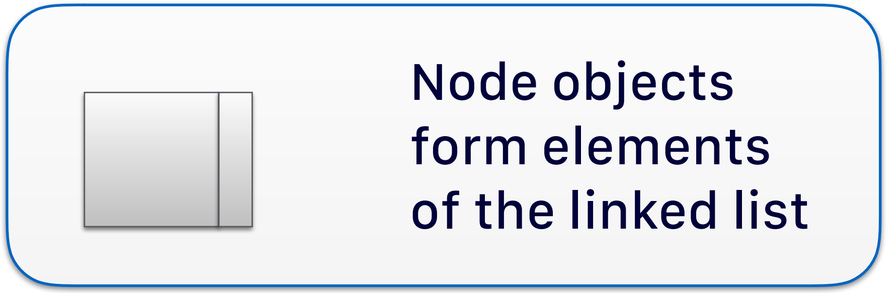
```
char[] A = new char[N];
char[0] = ‘E’;
...
```



- arrays are sequential chunks of memory
- require contiguous space of size N*sizeof(object) to be available

A Linked List is a collection of **elements** that together form a linear ordering. Each node stores a reference to an element and a reference, called (next) to another node.

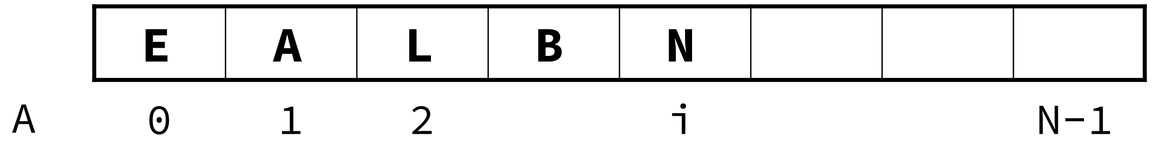




Linked Structures

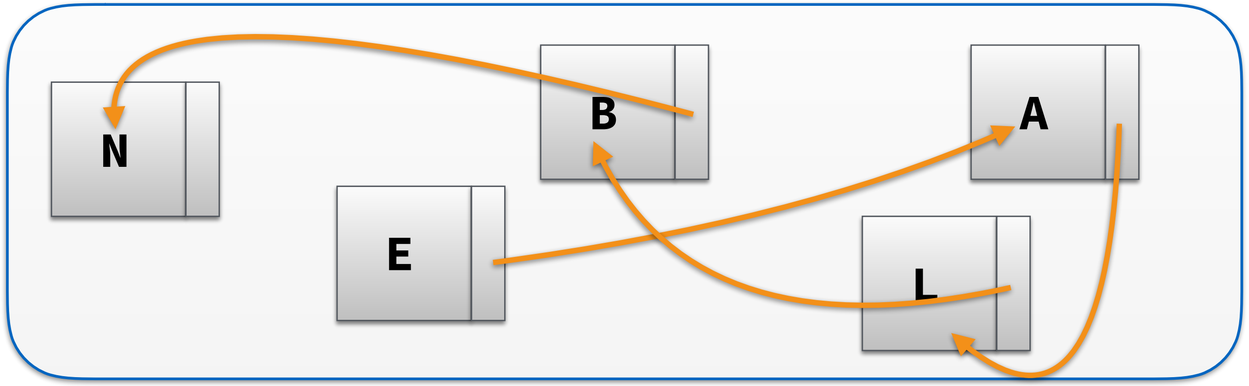
An array is a collection of variables stored sequentially in memory.

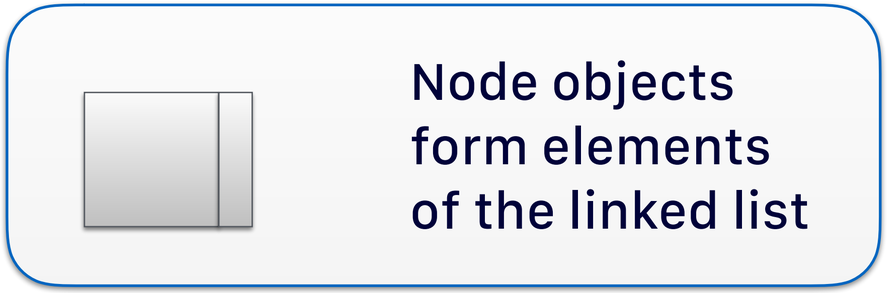
```
char[] A = new char[N];
char[0] = ‘E’;
...
```



- arrays are sequential chunks of memory
- require contiguous space of size N*sizeof(object) to be available

A Linked List is a collection of **elements** that together form a linear ordering. Each node stores a reference to an element and a reference, called (next) to another node.

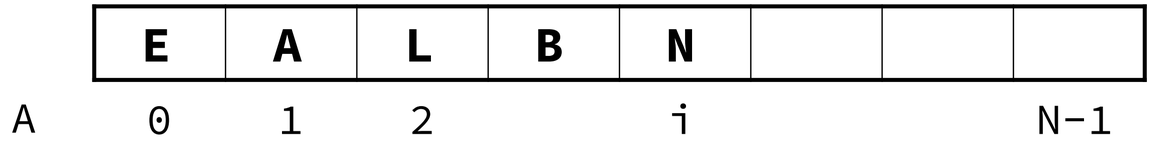




Linked Structures

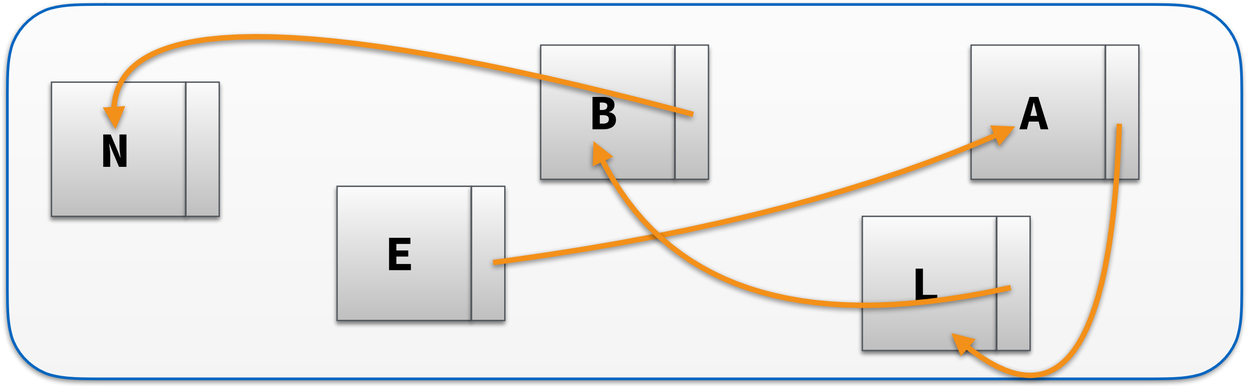
An array is a collection of variables stored sequentially in memory.

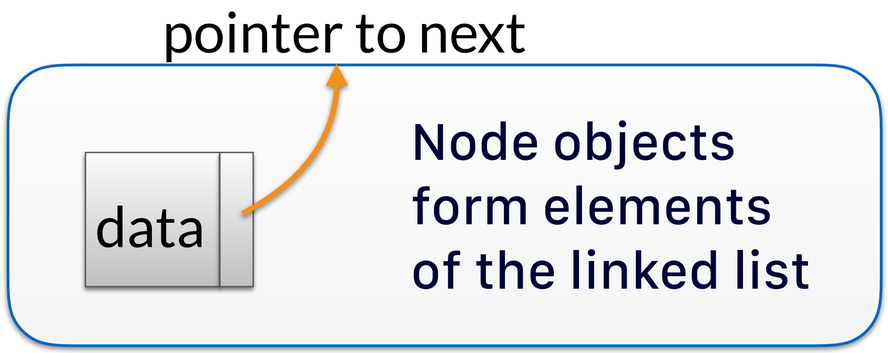
```
char[] A = new char[N];
char[0] = ‘E’;
...
```



- arrays are sequential chunks of memory
- require contiguous space of size N*sizeof(object) to be available

A Linked List is a collection of **elements** that together form a linear ordering. Each node stores a reference to an element and a reference, called (next) to another node.

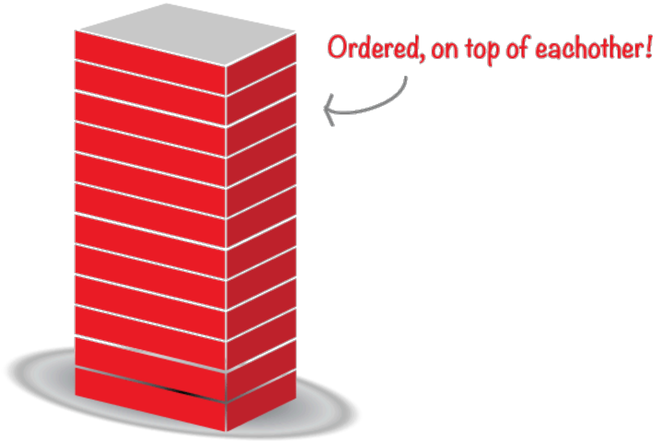




Stack and Heap Memory

Stack

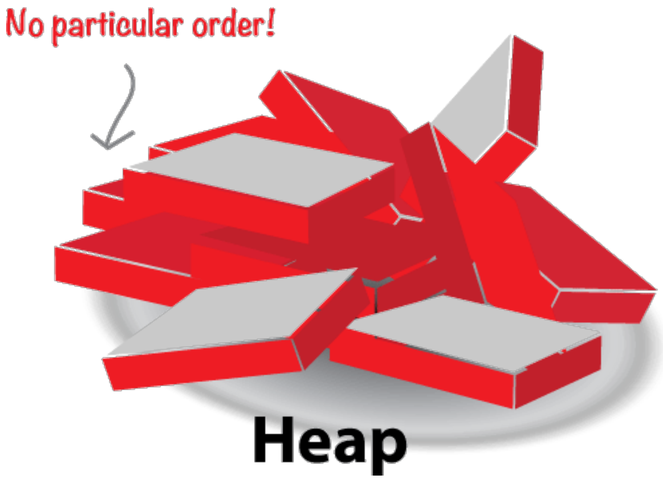
- Very fast access
- Don't have to explicitly de-allocate variables
- Space is managed efficiently by CPU, memory will not become fragmented
- Local variables only
- Limit on stack size (OS-dependent)
- Variables cannot be resized



Stack

Heap

- Variables can be accessed globally
- No limit on memory size
- (Relatively) slower access
- No guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
- Variables can be resized



Heap

Linked List

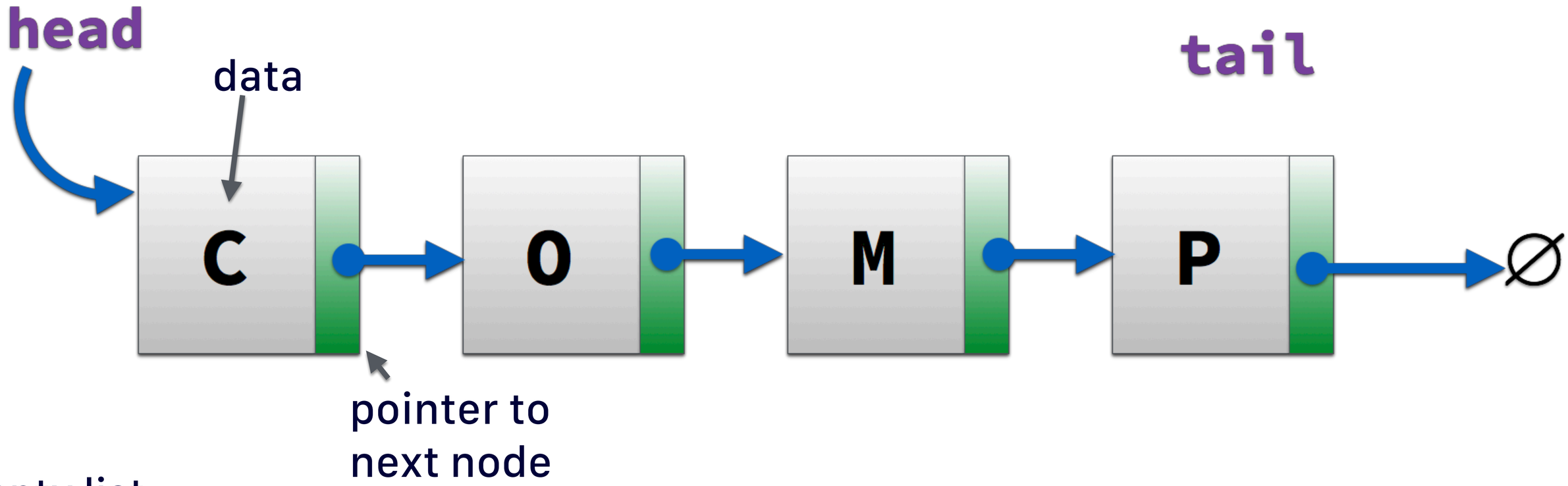
- A Linked List is a collection of ***nodes*** that together form a linear ordering. Each node stores a reference to an element and a reference, called *next* to another node.

- The *next* reference inside a node can be viewed as a ***link*** or ***pointer*** to another node.

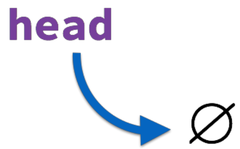
- Moving from one node to the another by following the *next* reference is known as ***link hopping*** or ***pointer hopping***.

- The first and last node of a linked list are called the ***head*** and ***tail*** of the list.

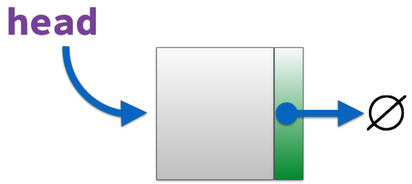
Linked Lists



empty list



head points to null



use a header node with next pointer to null

A linked list is a linear data structure where each element is a separate object.

Linked Lists

- Each element (or **node**) of a list is comprises two items:
  - the data
  - reference to the next node.
- The last node has a reference to null.
- The entry point into a linked list is called the **head** of the list.
- the head is not a separate node, but the reference to the first node.
- If the list is empty then the head is a null reference.

Linked Lists

A linked list is a dynamic data structure

- The number of nodes in a list is not fixed and can grow and shrink on demand.

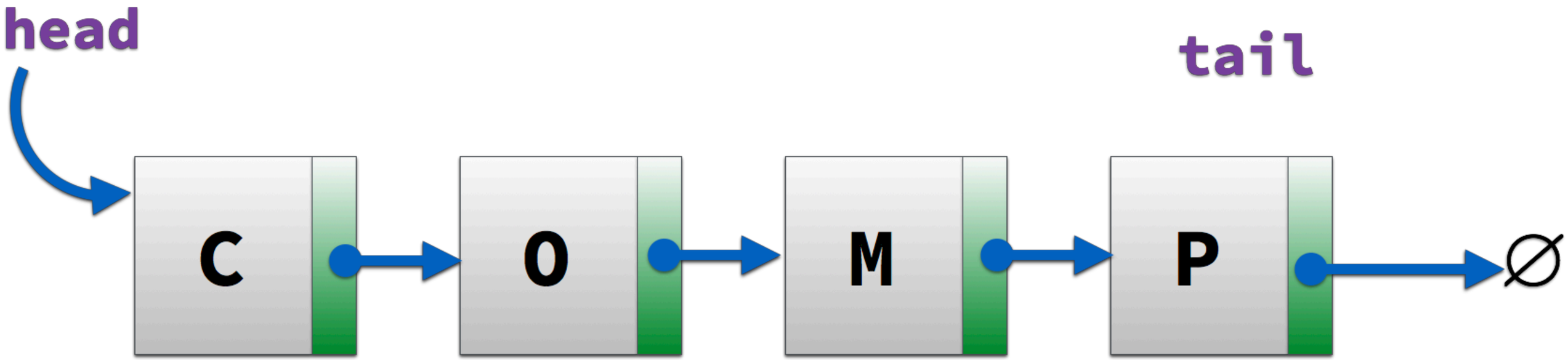
- An application which has to deal with an unknown number of objects will need to use a linked list.

One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements.

- If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

List Operations

Basic Structure



OO Design

- We need a class to hold the data -> ***Node***
- and one for the List itself -> ***SinglyLinkedList***

```
public class Node {

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

  /** A reference to the subsequent node in the list */
  private Node next; // reference to the subsequent node in the list

  /**
   * Creates a node with the given element and next node.
   *
   * @param e: the element to be stored
   * @param n: reference to a node that should follow the new node
   */
  public Node(String e, Node n) {
    element = e;
    next = n;
  }
}
```

Let's start with a non-generic Node class

Node.java

```
  /**
   * Returns the element stored at the node.
   *
   * @return the element stored at the node
   */
  public String getElement() {
    return element;
  }

  /**
   * Returns the node that follows this one (or null
if no such node).
   *
   * @return the following node
   */
  public Node getNext() {
    return next;
  }
```

```
  /**
   * Sets the node's next reference to point to Node
n.
   *
   * @param n
   *             the node that should follow this one
   */
  public void setNext(Node n) {
    next = n;
  }
}
```

LinkedList.java

```
public class SinglyLinkedList {

  /** The head node of the list */
  private Node head = null; // head node of the list (or null if empty)

  /** The last node of the list */
  private Node tail = null; // last node of the list (or null if empty)

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

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

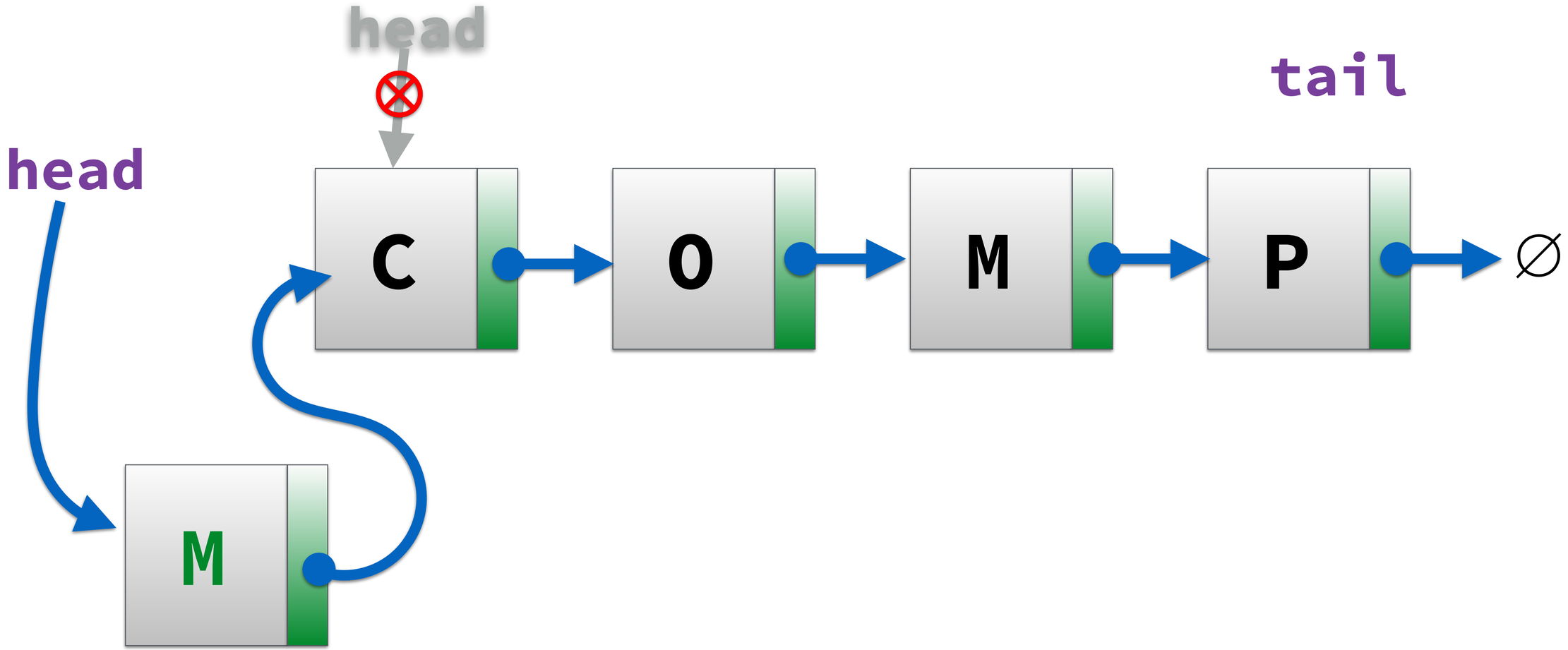
LinkedList.java

```
  /**
   * @return number of elements in the linked
list
   */
  public int size() {
    return size;
  }

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

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

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

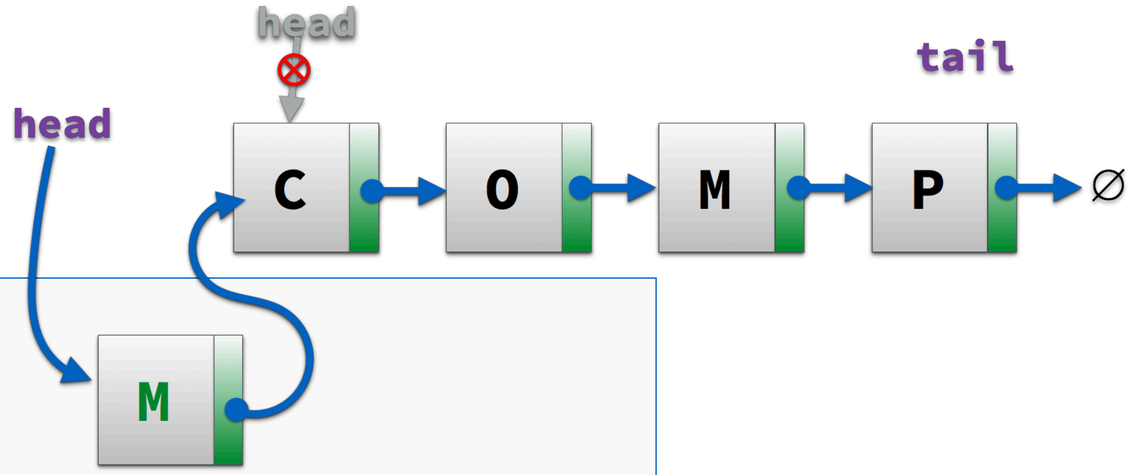
Linked List addFirst

The method creates a node and prepends it at the beginning of the list.



Linked List addFirst

The method creates a node and prepends it at the beginning of the list.

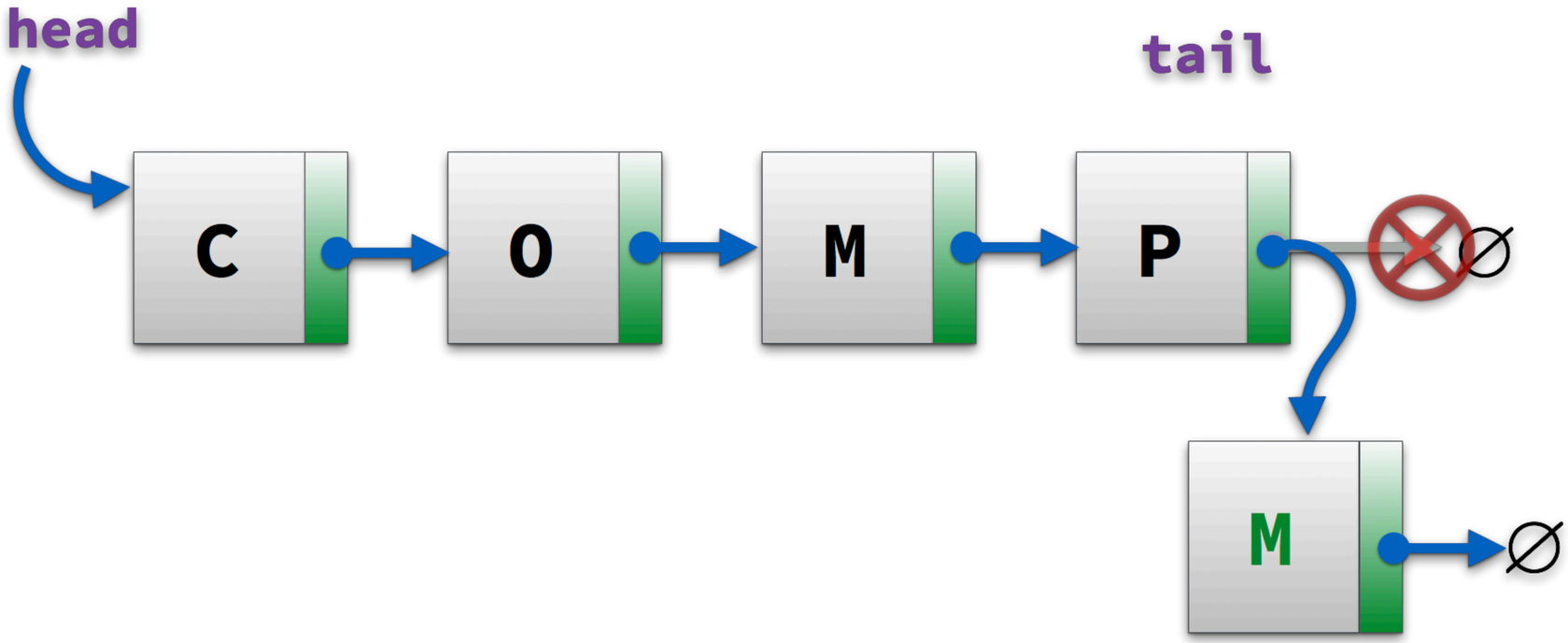



```
/**
   * Adds an element to the front of the list.
   *
   * @param e
   *             the new element to add
   */
  public void addFirst(String e) { // adds element e to the front of the list
    head = new Node(e, head); // create and link a new node
    if (size == 0)
      tail = head; // special case: new node becomes tail also
    size++;
  }
```

LinkedList addLast

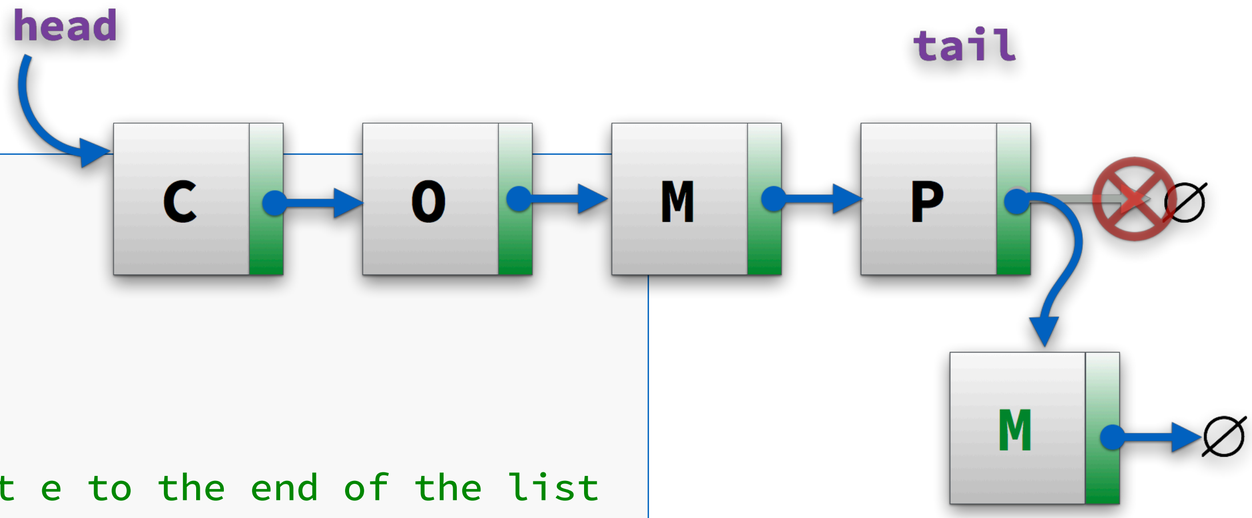
The method appends the node to the end of the list. This requires traversing, but make sure you stop at the last node.

Alternatively, we can store a reference to the last node, which makes the traversal unnecessary.



LinkedList addLast

The method appends the node to the end of the list. Uses a pointer to the last node.

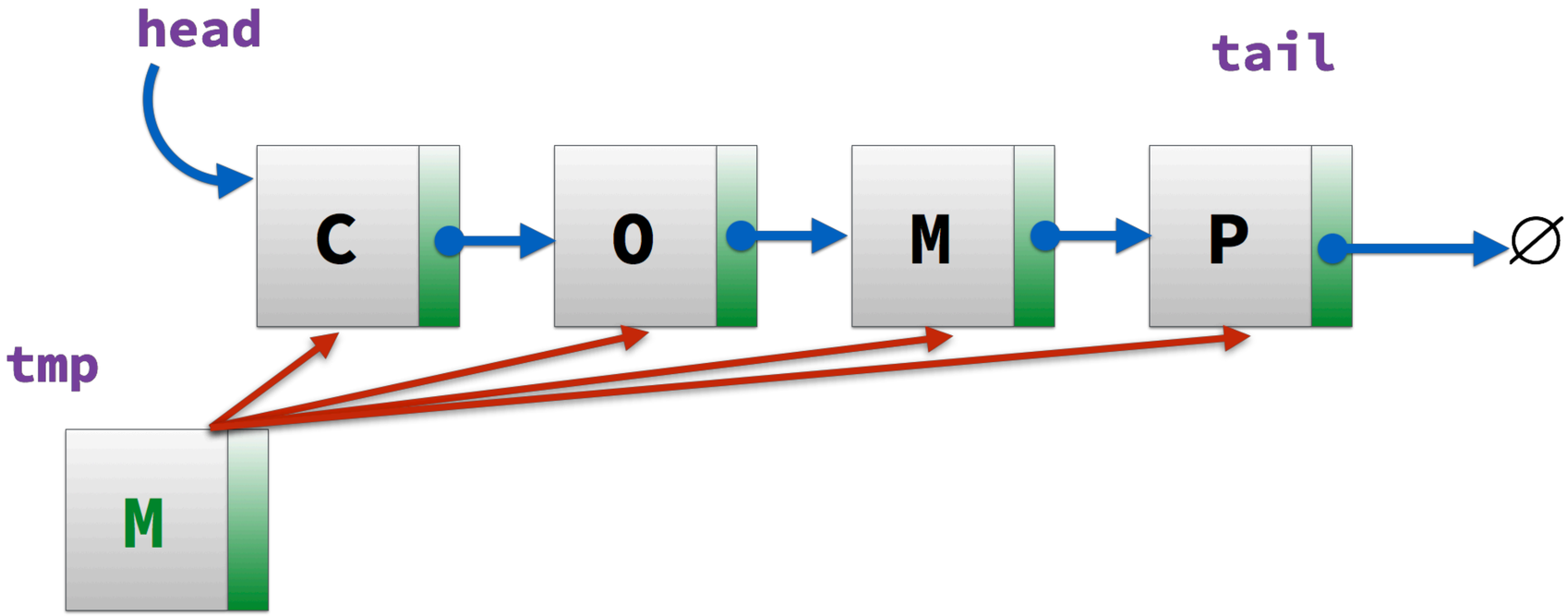



```
/**
 * Adds an element to the end of the list.
 *
 * @param e
 *             the new element to add
 */
public void addLast(String e) { // adds element e to the end of the list
  Node last = new Node(e, null); // node will eventually be the tail
  if (isEmpty())
    head = last; // special case: previously empty list
  else
    tail.setNext(last); // new node after existing tail
  tail = last; // new node becomes the tail
  size++;
```

LinkedList traverse

- Traversing the list

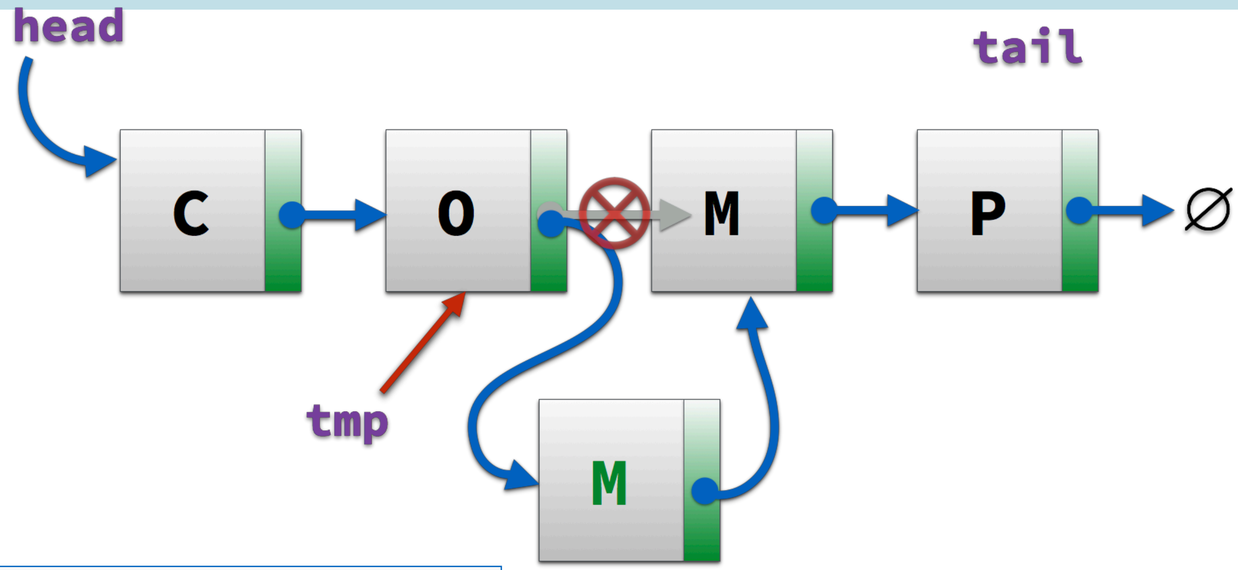
```
Node tmp = head;
while(tmp != null) {
  tmp = tmp.getNext();
}
```



LinkedList insertAfter

Find a node containing "key" and insert a new node ***after*** it.

We could use a positional index instead of a key.

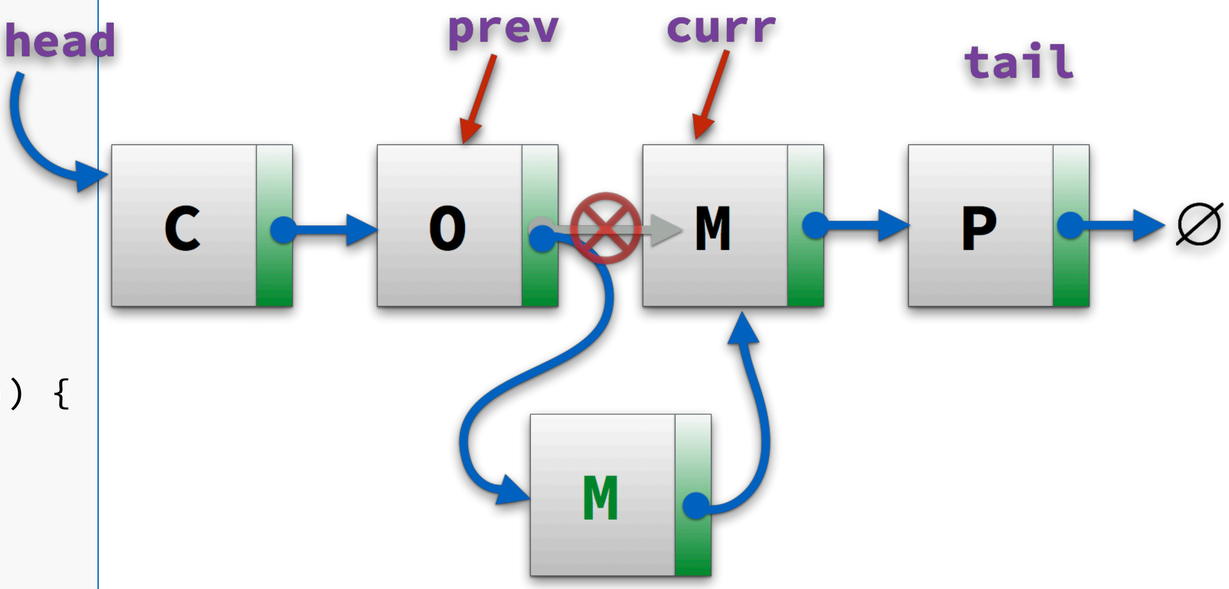



```
public void insertAfter(String key, String s) {
  Node tmp = head;
  while(tmp != null && !tmp.getElement().equals(key)) {
    tmp = tmp.next;
  }
  if(tmp != null) {
    tmp.setNext(new Node(s, tmp.getNext()));
  }
}
```

LinkedList insertBefore

```
public void insertBefore(String key, String s) {
   if (head == null) {
      return null;
   }
   if (head.getElement().equals(key)) {
      addFirst(toInsert);
      return;
   }

   Node prev = null;
   Node curr = head;

   while (curr != null && !curr.getElement().equals(key)) {
      prev = curr;
      curr = curr.getNext();
   }
   // insert between curr and prev
   if (curr != null) {
      prev.setNext(new Node(s, curr));
   }
}
```

Find a node containing "key" and insert a new node ***before*** it.



LinkedList deletion

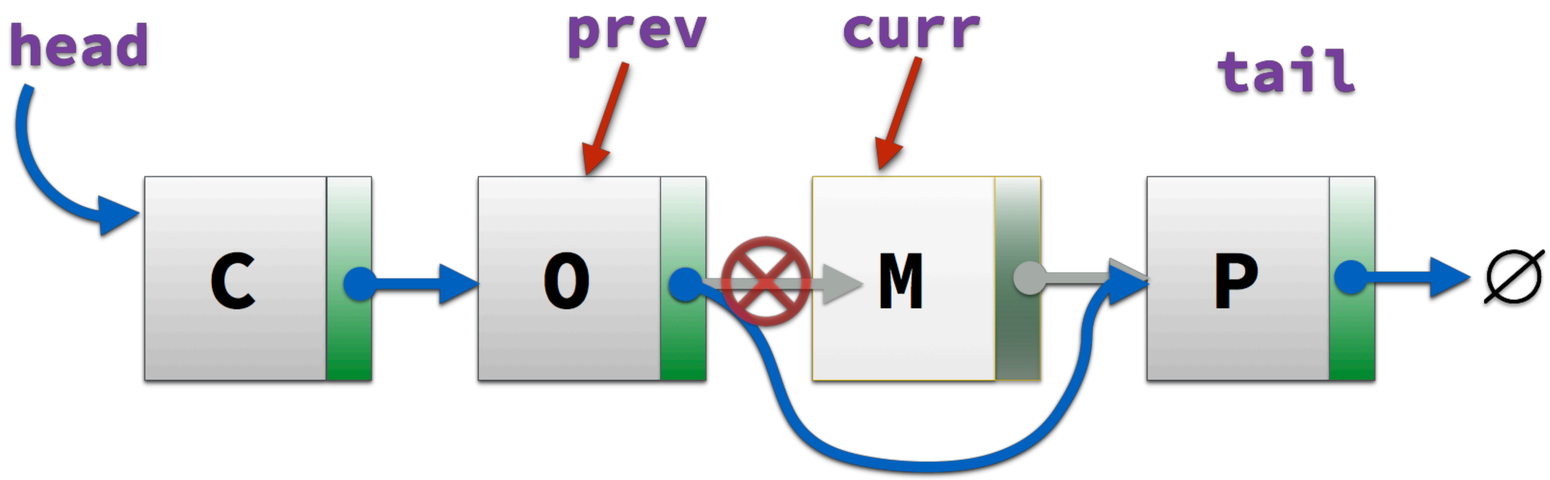
Find a node containing "key" and delete it.

The algorithm is similar to insertBefore() algorithm.

It is convenient to use two references `prev` and `curr`. When we move along the list we shift these two references, keeping `prev` one step before `cur`.

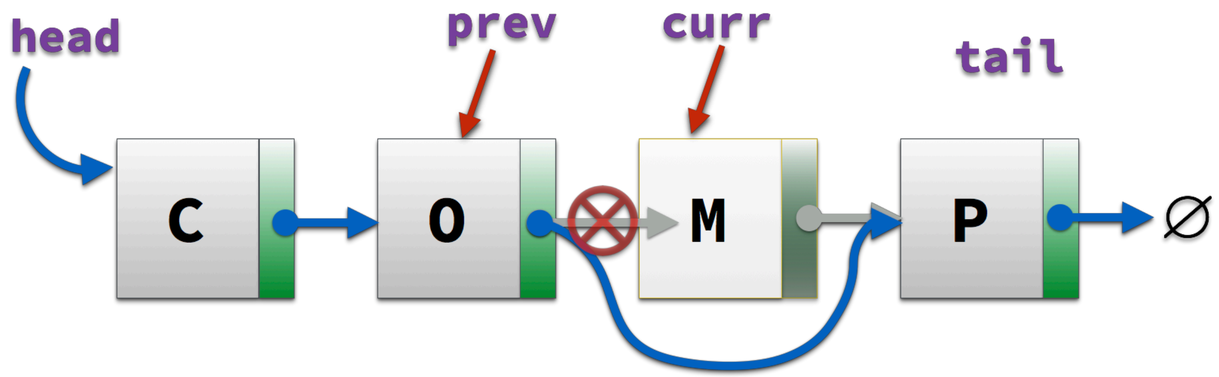
We continue until `cur` reaches the node which we need to delete. There are three exceptional cases, we need to take care of:

1. list is empty
2. delete the head node
3. node is not in the list



LinkedList deletion

```
public void remove(String key) {
   if (head == null) {
      throw new RuntimeException("cannot delete");
   }

   if (head.getElement().equals(key)) {
      head = head.next;
      return;
   }

   Node cur = head;
   Node prev = null;

   while (cur != null && !cur.getElement().equals(key)) {
      prev = cur;
      cur = cur.next;
   }

   if (cur == null) {
      throw new RuntimeException("cannot delete");
   }

   // delete cur node
   prev.next = cur.next;
}
```



List Implementation Details

Linked Lists in Java

- There are a few points about linked lists to note before we move on
- This might be a typical usage for our list:

```
// start with some String data
String [] data = {"one", "two", "three", "four", "five"};

// create the linked list
SinglyLinkedList ll = new SinglyLinkedList();

// add the data to the list
for(String s: data) {
  ll.addLast(s);
}

System.out.println(ll); // print out the linked list

// iterate over the list and print it out
for(String s : ll) {
  System.out.println(s);
}
```

- we really need to implement a generic class
- we never use the Node class explicitly outside the linked lists
- we would like to build a string representation, and to iterate over the elements of the

Node and List

- The first thing we usually do is make the Node a private static nested class of the LinkedList.

- Why?

Node and List

- If the class will not be used outside the enclosing class, we make it a private inner class
- A *nested* class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
- Static nested classes do not have access to other members of the enclosing class.
- There is no need for LinkedList.Node to be top-level class as it is only used by LinkedList. Since it does not need access to LinkedList's

LinkedList: nested Node class

```
public class SinglyLinkedList {

  private static class Node {

    private String element; // reference to the element stored at this node

    /** A reference to the subsequent node in the list */
    private Node next; // reference to the subsequent node in the list

    public Node(String e, Node n) {
      element = e;
      next = n;
    }
  }
```

LinkedList: generics

```
public class SinglyLinkedList<E> {

  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 subsequent node in the list */
    private Node<E> next; // reference to the subsequent node in the list

    /**
     * Creates a node with the given element and next node.
     *
     * @param e: the element to be stored
     * @param n: reference to a node that should follow the new node
     */
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
```

previously “String”

you could write the LinkedList class using a concrete type, like String, first, then convert it to a generic representation

LinkedList: iteration

old style iteration with an index

```
for(int i = 0; i < s.length; ++i) {
  //
}
```

Java for-each loop

no explicit index!

```
String[] data = {"one", "two", "three"};
for(String s : data) {
  System.out.println(data);
}
```

LinkedList: iteration

this for-each loop is effectively equivalent to:

```
for(T name : expression) {
  System.out.println(name);
}
```

```
for(Iterator<T> it = expression.iterator(); it.hasNext(); ) {
  T name = it.next();
  System.out.println(name);
}
```

Iterable Interface

Java defines a parameterised interface, named Iterable, that includes the following single method:

iterator( ): Returns an iterator of the elements in the collection.

An instance of a typical collection class in Java, such as an ArrayList, is iterable (but not itself an iterator); it produces an iterator for its collection as the return value of the iterator( ) method.

Each call to iterator( ) returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.

LinkedList iteration:

```
public class SinglyLinkedList<E> implements Iterable<E> {
  public Iterator<E> iterator() {
    return new ListIterator();
  }

  private class ListIterator implements Iterator<E> {
    Node curr;
    public ListIterator() {
      curr = head;
    }
    public boolean hasNext() {
      return curr != null;
    }

    @Override
    public E next() {
      E res = (E) curr.getElement();
      curr = curr.getNext();
      return res;
    }
  }
```

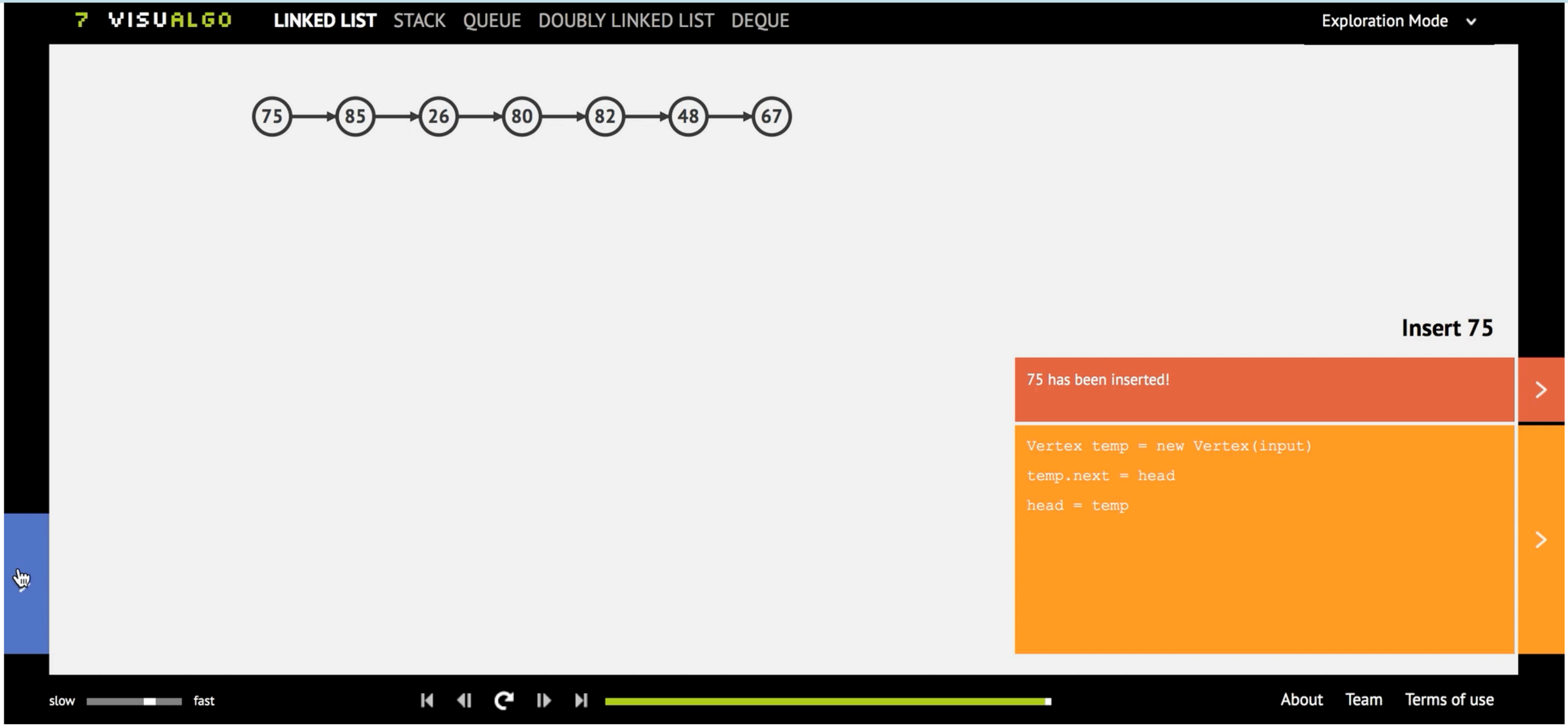
the SinglyLinkedList class implements the Iterable interface

The Iterable interface needs the generic parameter E

The iterator() function returns an object of ListIterator (which is another inner class)

The ListIterator controls how it traverses the internal elements of the list

Singly Linked List Animation



Singly Linked List Animation