Linked lists

Each node of the list is a separate object:

```
class Node[T] (val el: T, var next: Node)
```

The object representing the entire list contains only a reference to the front of the list:

```
class LinkedList[T] {
  private var front: Node[T] = null

  def first: Node[T] =
    if (front== null) throw NoSuchElementException
    else front
}
```


List with fast append

The list object contains a reference to the first and last element of the list:

```
class LinkedList[T] {
  private var front: Node[T] = null
  private var rear: Node[T] = null
  // ...
}
```

Now append is fast and easy:

```
def append(el: T)
```

But we have to be careful with all our other methods...
The `rear` field must be updated by every method of the `LinkedList` class.

Basic list methods

Add an element at the front of the list:

```
def prepend(el: T)
```

Remove the first element:

```
def removeFirst()
```

Add an element after `node`:

```
def addAfter(node: Node[T], el: T)
```

Remove element after `node`:

```
def removeAfter(node: Node[T])
```

Find element before `node`:

```
def before(node: Node[T]): Node[T]
```

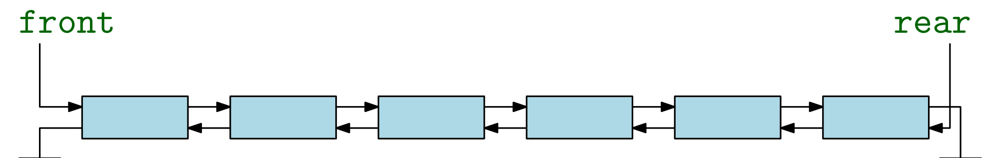
Find last node:

```
def last: Node[T]
```


Doubly-linked lists

If we want to be able to quickly search a list both forward and backward, we need a doubly-linked list.

front	rear

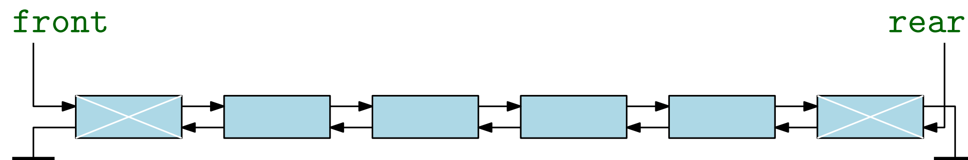


```
class Node[T] (val el: T, var next: Node[T],
               var prev: Node[T])
```

Sentinels

In doubly-linked lists, the code for inserting and removing elements needs to handle the case where the node is the first node and/or the last node separately.

We can simplify the code by using sentinel nodes. Sentinel nodes are "guarding" the two ends of the list, so that no special handling is necessary. Sentinel nodes do not contain elements. When we create an empty list, we automatically create the two sentinel nodes, which cannot be deleted.

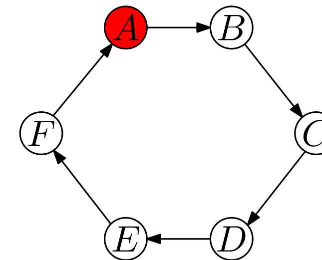


The Josephus Game

Given n player sitting in a circle, and a number m.

A hot potato starts at player 1, and is passed around m times. The player holding the potato then is eliminated, the next player gets the potato, and the game continues until only one player is left.

$n = 6, \ m = 2$

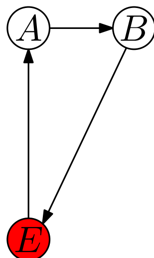


The Josephus Game

Given n player sitting in a circle, and a number m.

A hot potato starts at player 1, and is passed around m times. The player holding the potato then is eliminated, the next player gets the potato, and the game continues until only one player is left.

$n = 6, \ m = 2$



What data structure to use?

We need a data structure to store the circle of people.

Required methods:

- Pass the potato to the next person.
- Delete the person holding the potato.

A doubly linked list does it all. A circular linked list would be even better, but we can simulate that easily.