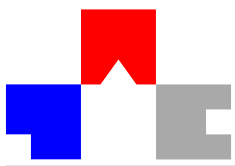


Each **node** of the list is a separate object:

```
class Node:
    def __init__(self, el, next=None):
        self.el = el
        self.next = next
```

```
>>> a = Node(13)
>>> b = Node(27, a)
>>> c = Node(99, b)
>>> c.el
99
>>> c.next.el
27
>>> c.next.next.el
13
```

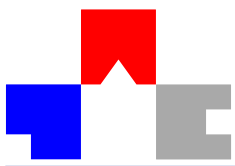


Let's make a class for a linked list. It contains a reference to the **front** of the list:

```
class LinkedList:
    def __init__(self):
        self._front = None

    def first(self):
        if self._front is None:
            raise EmptyListError
        return self._front

    def is_empty(self):
        return self._front is None
```



Add an element at the front of the linked list:

```
def prepend(self, el)
```

Remove the first element:

```
def remove_first(self)
```

Insert an element after node:

```
def insert_after(self, node, el)
```

Remove element after node:

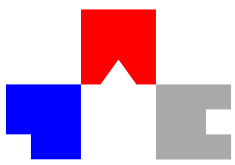
```
def remove_after(self, node)
```

Find element **before** node:

```
def before(self, node)
```

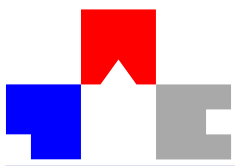
Find last node:

```
def last(self)
```



To compute the length of the linked list, or to display the list, we need to traverse the entire list:

```
def __len__(self):  
    if self.is_empty():  
        return 0  
    p = self._front  
    count = 0  
    while p is not None:  
        count += 1  
        p = p.next  
    return count
```



Appending to the list takes time linear in the length of the list.

To speed this up, the list needs to store a reference to both the **first** and **last node** of the list:

```
class LinkedList:
    def __init__(self):
        self._front = None
        self._rear = None
```

Now **append** is fast and easy:

```
def append(self, el)
```

But we have to be careful with all our other methods...

The **rear** field must be updated by every method of the **LinkedList** class.



---

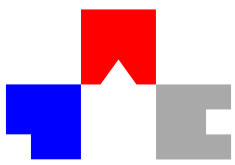
Remember the `Queue` ADT?

- `enqueue`
- `dequeue`
- `front`
- `is_empty`

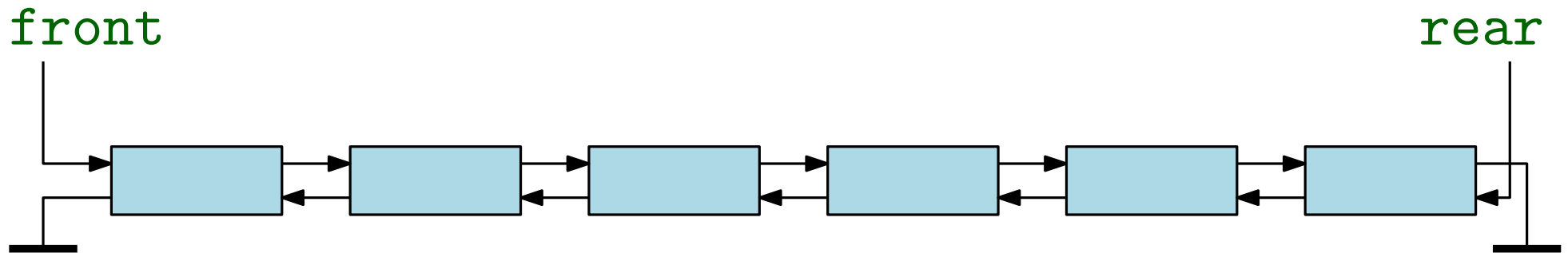
We can implement this as a linked list with fast append:

- `enqueue` appends at the rear of the list,
- `dequeue` removes from the front of the list.

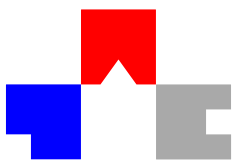
Now you know why the two ends of a queue are called front and rear...



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

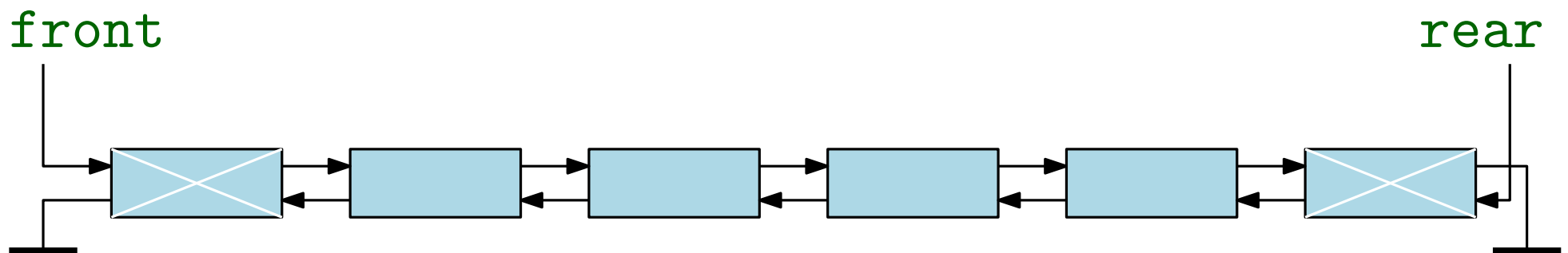


```
class Node:
    def __init__(self, el, next=None, prev=None):
        self.el = el
        self.next = next
        self.prev = prev
```

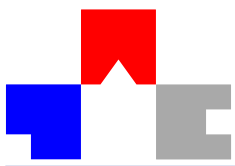


In doubly-linked lists, the code for inserting and removing elements needs to handle the case where the node is the first node and/or the last node separately.

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.



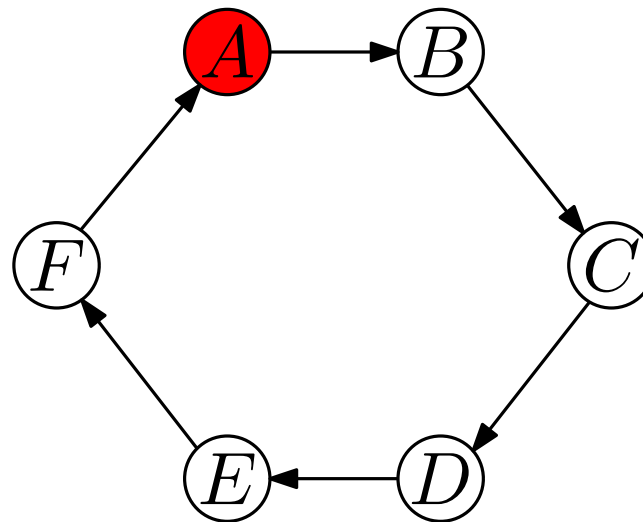


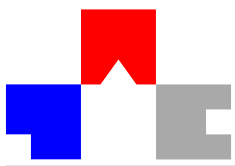


Given  $n$  player sitting in a circle, and a number  $m$ .

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.

$$n = 6, m = 2$$

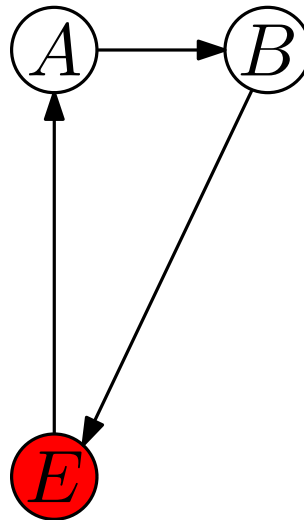


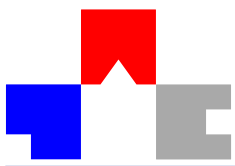


Given  $n$  player sitting in a circle, and a number  $m$ .

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.

$$n = 6, m = 2$$





---

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.