Listy odsyłaczowe

Jednokierunkowa lista odsyłaczowa

Implementacja struktury #1

(Implementacja obiektowa)

In [1]:
```python
class Node:
    def __init__(self, val=None):
        self.val = val
        self.next = None


class LinkedList:
    def __init__(self, values: 'iterable' = None):
        self.head = self.tail = None
        self.length = 0
        values and self.extend(values)  # The same as 'if values: self.extend(values)'

    def __iter__(self):
        curr = self.head
        while curr:
            yield curr.val
            curr = curr.next

    def __str__(self):
        return ' -> '.join(map(str, self))

    def __len__(self):
        return self.length

    def append(self, val: object):
        node = Node(val)
        if not self:
            self.head = self.tail = node
        else:
            self.tail.next = node
            self.tail = node
        self.length += 1

    def extend(self, values: 'iterable'):
        if values:
            iterator = iter(values)
            if not self:
                self.head = self.tail = Node(next(iterator))
            for val in iterator:
                self.tail.next = Node(val)
                self.tail = self.tail.next
            self.length += len(values)

    def appendleft(self, val: object):
        node = Node(val)
        if not self:
            self.head = self.tail = node
        else:
            node.next = self.head
            self.head = node
        self.length += 1

    def extendleft(self, values: 'iterable'):
        # Note that extendleft adds values in a reversed order so the first value of a linked
        # list will be the last value of the 'values' iterable (the same rule applies to the Python's
        # deque data structure, which is, in fact, a doubly linked list)
        if values:
            iterator = iter(values)
            if not self:
                self.head = self.tail = Node(next(iterator))
            for val in iterator:
                node = Node(val)
                node.next = self.head
                self.head = node
            self.length += len(values)

    def popleft(self) -> object:
        if not self:
            raise IndexError(f'pop from an empty {self.__class__.__name__}')
        removed = self.head.val
        if len(self) == 1:
            self.head = self.tail = None
        else:
            self.head = self.head.next
        self.length -= 1
        return removed

    def clear(self):
        self.length = 0
        self.head = self.tail = None

    def copy(self) -> 'LinkedList':
        return self.__class__(self)

    def count(self, val: object) -> int:
        total = 0
        for curr_val in self:
            if curr_val == val:
                total += 1
        return total

    def index(self, val: object) -> int:
        for i, curr_val in enumerate(self):
            if curr_val == val:
                return i
        raise ValueError(f'{val} not in {self.__class__.__name__}')

    def insert(self, idx: int, val: object):
        # Depending on the idx variable value, insert a new node in a correct position
        if idx <= 0:
            self.appendleft(val)
        elif idx >= len(self):
            self.append(val)
        else:
            # Look for the previous node after which a new node will be inserted
            prev_node = self._traverse_to_index(idx)
            self._insert_node_after(prev_node, Node(val))

    def remove(self, val: object):
        if self:
            if self.head.val == val:
                self.popleft()
                return
            else:
                prev_node = self.head
                while prev_node.next:
                    if prev_node.next.val == val:
                        self._remove_node_after(prev_node)
                        return
                    prev_node = prev_node.next
        raise ValueError(f'{self.__class__.__name__}.remove(val): val not in {self.__class__.__name__}')

    def reverse(self):
        # Modify a Linked List only if its length is greater than 1 as empty or single-element
        # Linked List will remain unchanged after having been reversed
        if len(self) > 1:
            curr_node = self.head
            next_node = curr_node.next
            self.tail = curr_node
            self.tail.next = None
            while next_node:
                temp = next_node.next
                next_node.next = curr_node
                curr_node = next_node
                next_node = temp
            self.head = curr_node

    def _traverse_to_index(self, idx: int) -> 'previous node object':
        if idx <= 0:  # We cannot return a node previous to the first one
            return None
        curr_node = self.head
        curr_idx = 1
        while curr_idx < idx:
            curr_node = curr_node.next
            curr_idx += 1
        return curr_node

    def _insert_node_after(self, prev_node: Node, curr_node: Node):
        curr_node.next = prev_node.next
        prev_node.next = curr_node
        self.length += 1

    def _remove_node_after(self, prev_node: Node):
        if prev_node.next is self.tail:
            self.tail = prev_node
        prev_node.next = prev_node.next.next
        self.length -= 1
```

Kilka testów

In [2]:
```python
a = LinkedList(range(5))
print(a)
a.append(5)
print(a)
a.appendleft(-1)
print(a)
a.clear()
print(a)
a.extend(range(5, 15, 3))
print(a)
b = a.copy()
b.extendleft(range(0, -15, -4))
print(b, a, sep='\t')
a.insert(-1, 5)
print(a)
a.insert(12345, 5)
print(a)
a.insert(4, 5)
print(a)
print(a.count(5))
print(a.index(14), a.index(5), a.index(8), sep='\t')
print(*(a.popleft() for _ in range(4)))
print(a)
a.remove(5)
print(a)
a.remove(5)
print(a)
a.reverse()
print(a)
print(b)
b.reverse()
print(b)
print(a.popleft())
# print(a.popleft())  # this is to check if an exception is properly raised
```

```
0 -> 1 -> 2 -> 3 -> 4
0 -> 1 -> 2 -> 3 -> 4 -> 5
-1 -> 0 -> 1 -> 2 -> 3 -> 4 -> 5

5 -> 8 -> 11 -> 14
-12 -> -8 -> -4 -> 0 -> 5 -> 8 -> 11 -> 14	5 -> 8 -> 11 -> 14
5 -> 5 -> 8 -> 11 -> 14
5 -> 5 -> 8 -> 11 -> 14 -> 5
5 -> 5 -> 8 -> 11 -> 5 -> 14 -> 5
4
5	0	2
5 5 8 11
5 -> 14 -> 5
14 -> 5
14
14
-12 -> -8 -> -4 -> 0 -> 5 -> 8 -> 11 -> 14
14 -> 11 -> 8 -> 5 -> 0 -> -4 -> -8 -> -12
14
```

Implementacja struktury #2

(Implementacja funkcyjna)

In [3]:
```python
class Node:
    def __init__(self, val=None):
        self.val = val
        self.next = None


def create_linked_list(values: 'iterable' = None) -> 'linked list head (sentinel)':
    curr = head = Node()  # A sentinel node
    if not values: return head
    for val in values:
        curr.next = Node(val)
        curr = curr.next
    return head


def print_linked_list(ll_head: 'linked list head (sentinel)'):
    curr = ll_head.next
    print(ll_head.val, end=' ')
    while curr:
        print('->', curr.val, end=' ')
        curr = curr.next
    print()

# This operation below is very inefficient (O(n)) compared to the same operation
# implemented using a LinkedList class above (which is O(1) in such a case)
def append_to_linked_list(ll_head: 'linked list head (sentinel)', val: object):
    # Store a tail node
    tail = _get_last_node(ll_head)
    # Append a node to the last one node
    tail.next = Node(val)

# Also a bit inefficient but traverses to the lat node only once na then appends
# values to the last node without traversing there again
def extend_linked_list(ll_head: 'linked list head (sentinel)', values : 'iterable'):
     # Store a tail node
    tail = _get_last_node(ll_head)
    # Appned values by one to the last node
    for val in values:
        tail.next = Node(val)
        tail = tail.next


def appendleft_to_linked_list(ll_head: 'linked list head (sentinel)', val: object):
    node = Node(val)
    node.next = ll_head.next
    ll_head.next = node


def extendleft_linked_list(ll_head: 'linked list head (sentinel)', values: 'iterable'):
    # Create a new linked list of the values specified
    new_head = new_curr = Node()  # Create a sentinel node
    for val in values:
        new_curr.next = Node(val)
        new_curr = new_curr.next
    # Link a new linked list to the one given as a function argument
    new_curr.next = ll_head.next
    ll_head.next = new_head.next


def popleft_from_linked_list(ll_head: 'linked list head (sentinel)') -> 'removed value':
    if not ll_head.next:
        raise IndexError('pop from an empty linked list')
    removed = ll_head.next.val
    ll_head.next = ll_head.next.next
    return removed


def clear_linked_list(ll_head: 'linked list head (sentinel)'):
    ll_head.next = None


def copy_linked_list(ll_head: 'linked list head (sentinel)') -> 'linked list copy head (sentinel)':
    copy_head = copy_curr = Node()  # A sentinel node
    curr = ll_head.next
    while curr:
        copy_curr.next = Node(curr.val)
        copy_curr = copy_curr.next
        curr = curr.next
    return copy_head


def count_value_in_linked_list(ll_head: 'linked list head (sentinel)', val: object) -> int:
    curr = ll_head.next
    total = 0
    while curr:
        if curr.val == val:
            total += 1
        curr = curr.next
    return total


def index_of_value_in_linked_list(ll_head: 'linked list head (sentinel)', val: object) -> int:
    idx = 0
    curr = ll_head.next
    while curr:
        if curr.val == val:
            return idx
        curr = curr.next
        idx += 1
    raise ValueError(f'{val} not in linked list')


def insert_in_linked_list(ll_head: 'linked list head (sentinel)', idx: int, val: object):
    if idx <= 0:
        appendleft_to_linked_list(ll_head, val)
    else:
        prev_node = _traverse_to_index(ll_head, idx)
        _insert_node_after(prev_node, Node(val))


def remove_from_linked_list(ll_head: 'linked list head (sentinel)', val: object):
    if ll_head.next:
        prev_node = ll_head
        while prev_node.next:
            if prev_node.next.val == val:
                _remove_node_after(prev_node)
                return
            prev_node = prev_node.next
    raise ValueError('linkedlist.remove(val): val not in linkedlist')


def reverse_linked_list(ll_head: 'linked list head (sentinel)'):
    # Reverse a linked list only if there are at least 2 values
    # (in other cases reversing is pointless)
    if ll_head.next and ll_head.next.next:
        curr_node = ll_head.next
        next_node = curr_node.next
        while next_node:
            temp = next_node.next
            next_node.next = curr_node
            curr_node = next_node
            next_node = temp
        tail_node = curr_node
        # Link a reversed linked list to a sentinel node
        ll_head.next.next = None
        ll_head.next = tail_node


def linked_list_to_list(ll_head: 'linked list head (sentinel)') -> list:
    values = []
    curr = ll_head.next
    while curr:
        values.append(curr.val)
        curr = curr.next
    return values

# If an index is too big (exceeds a number of elements in a linked list), the last node will be returned
def _traverse_to_index(ll_head: 'linked list head (sentinel)', idx: int) -> 'previous node object':
    curr = ll_head
    i = 0
    while curr.next:
        if i == idx:
            break
        curr = curr.next
        i += 1
    return curr


def _get_last_node(ll_head: 'linked list head (sentinel)') -> 'linked list tail':
    curr = ll_head
    while curr.next:
        curr = curr.next
    return curr


def _insert_node_after(prev_node: 'node after which a curr_node will be inserted', curr_node):
    curr_node.next = prev_node.next
    prev_node.next = curr_node


def _remove_node_after(prev_node: 'node after which a curr_node will be inserted'):
    prev_node.next = prev_node.next.next
```

Kilka testów

In [4]:
```python
a = create_linked_list(range(5))
print_linked_list(a)
append_to_linked_list(a, 5)
print_linked_list(a)
appendleft_to_linked_list(a, -1)
print_linked_list(a)
clear_linked_list(a)
print_linked_list(a)
extend_linked_list(a, range(5, 15, 3))
print_linked_list(a)
b = copy_linked_list(a)
extendleft_linked_list(b, range(0, -15, -4))
print_linked_list(b)
print_linked_list(a)
insert_in_linked_list(a, -1, 5)
print_linked_list(a)
insert_in_linked_list(a, 12345, 5)
print_linked_list(a)
insert_in_linked_list(a, 4, 5)
print_linked_list(a)
print(count_value_in_linked_list(a, 5))
print(index_of_value_in_linked_list(a, 14),
      index_of_value_in_linked_list(a, 5),
      index_of_value_in_linked_list(a, 8), sep='\t')
print(*(popleft_from_linked_list(a) for _ in range(4)))
print_linked_list(a)
remove_from_linked_list(a, 5)
print_linked_list(a)
remove_from_linked_list(a, 5)
print_linked_list(a)
reverse_linked_list(a)
print_linked_list(a)
print_linked_list(b)
reverse_linked_list(b)
print_linked_list(b)
print(popleft_from_linked_list(a))
# print(popleft_from_linked_list(a))  # this is to check if an exception is properly raised
print(linked_list_to_list(b))
```

```
None -> 0 -> 1 -> 2 -> 3 -> 4
None -> 0 -> 1 -> 2 -> 3 -> 4 -> 5
None -> -1 -> 0 -> 1 -> 2 -> 3 -> 4 -> 5
None
None -> 5 -> 8 -> 11 -> 14
None -> -12 -> -8 -> -4 -> 0 -> 5 -> 8 -> 11 -> 14
None -> 5 -> 8 -> 11 -> 14
None -> 5 -> 5 -> 8 -> 11 -> 14
None -> 5 -> 5 -> 8 -> 11 -> 14 -> 5
None -> 5 -> 5 -> 8 -> 11 -> 5 -> 14 -> 5
4
5	0	2
5 5 8 11
None -> 5 -> 14 -> 5
None -> 14 -> 5
None -> 14
None -> 14
None -> -12 -> -8 -> -4 -> 0 -> 5 -> 8 -> 11 -> 14
None -> 14 -> 11 -> 8 -> 5 -> 0 -> -4 -> -8 -> -12
14
[14, 11, 8, 5, -12, -8, -4, 0]
```

Dwukierunkowa lista odsyłaczowa

Poniższa implementacja jest bardzo podobna do implementacji listy jednokierunkowej (wersja obiektowa). Wynika to z faktu, że lista dwukierunkowa to taka "lista jednokierunkowa na sterydach", w której poza połączeniem węzłów z następnymi węzłami, występują również wskaźniki do poprzednich węzłów. Pozwala to na przeszukiwanie struktury "od tyłu", jeżeli np. wiemy, że dana wartość znajduje się bliżej końca lub musimy "się cofnąć" do poprzednich węzłów. Jedyną większą różnicą jest to, że usuwanie ostatniej wartości z listy dwukierunkowej (oraz także niekiedy wstawianie z użyciem metody $.insert()$) jest dużo szybsze ($O(1)$) niż w zwykłej jednokierunkowej liście odsyłaczowej ($O(n)$), ponieważ po usunięciu ostatniego węzła, możemy przypisać bez problemów przedostatni węzeł do atrybutu $tail$ bez konieczności iterowania przez całą listę od początku (dlatego też w implementacji listy jednokierunkowej powstrzymałem się od implementowania metody $.pop()$, bo jest ona niewydajna). Również metoda $.reverse()$ jest dużo łatwiejsza do zaimplementowania, ponieważ wystarczy zamieniać wskaźniki, a także można w wydajny sposób zaimplementować metodę $.rotate()$, która powoduje "przesunięcie" wszystkich wartości listy odsyłaczowej o 1 miejsce w prawo (dokł. przerzucenie wartości końcowej na początek).

Implementacja struktury #1

(Implementacja obiektowa)

In [5]:
```python
class Node:
    def __init__(self, val=None):
        self.val = val
        self.next = None
        self.prev = None


class DoublyLinkedList:
    def __init__(self, values: 'iterable' = None):
        self.head = self.tail = None
        self.length = 0
        values and self.extend(values) # The same as 'if values: self.extend(values)'

    def __iter__(self):
        curr = self.head
        while curr:
            yield curr.val
            curr = curr.next

    def __str__(self):
        return ' <-> '.join(map(str, self))

    def __len__(self):
        return self.length

    def append(self, val: object):
        node = Node(val)
        if not self:
            self.head = self.tail = node
        else:
            self.tail.next = node
            node.prev = self.tail
            self.tail = node
        self.length += 1

    def extend(self, values: 'iterable'):
        if values:
            iterator = iter(values)
            if not self:
                self.head = self.tail = Node(next(iterator))
            for val in values:
                node = Node(val)
                self.tail.next = node
                node.prev = self.tail
                self.tail = node
            self.length += len(values)

    def appendleft(self, val: object):
        node = Node(val)
        if not self:
            self.head = self.tail = node
        else:
            node.next = self.head
            self.head.prev = node
            self.head = node
        self.length += 1

    def extendleft(self, values: 'iterable'):
        # Note that extendleft adds values in a reversed order so the first value of a linked
        # list will be the last value of the 'values' iterable (the same rule applies to the Python's
        # deque data structure, which is, in fact, a doubly linked list)
        if values:
            iterator = iter(values)
            if not self:
                self.head = self.tail = Node(next(iterator))
            for val in iterator:
                node = Node(val)
                node.next = self.head
                self.head.prev = node
                self.head = node
            self.length += len(values)

    def pop(self) -> object:
        if not self:
            raise IndexError(f'pop from an empty {self.__class__.__name__}')
        removed = self.tail.val
        if len(self) == 1:
            self.head = self.tail = None
        else:
            self.tail = self.tail.prev
            self.tail.next = None
        self.length -= 1
        return removed

    def popleft(self) -> object:
        if not self:
            raise IndexError(f'pop from an empty {self.__class__.__name__}')
        removed = self.head.val
        if len(self) == 1:
            self.head = self.tail = None
        else:
            self.head = self.head.next
            self.head.prev = None
        self.length -= 1
        return removed

    def clear(self):
        self.length = 0
        self.head = self.tail = None

    def copy(self) -> 'LinkedList':
        return self.__class__(self)

    def count(self, val: object) -> int:
        total = 0
        for curr_val in self:
            if curr_val == val:
                total += 1
        return total

    def index(self, val: object) -> int:
        for i, curr_val in enumerate(self):
            if curr_val == val:
                return i
        raise ValueError(f'{val} not in {self.__class__.__name__}')

    def insert(self, idx: int, val: object):
        # Depending on the idx variable value, insert a new node in a correct position
        if idx <= 0:
            self.appendleft(val)
        elif idx >= len(self):
            self.append(val)
        else:
            center_idx = len(self) // 2
            node = Node(val)
            if idx < center_idx:
                # Look for the previous node after which a new node will be inserted
                prev_node = self._traverse_from_left(idx)
                self._insert_node_after(prev_node, node)
            else:
                # Look for the next node before which a new node will be inserted
                next_node = self._traverse_from_right(idx)
                self._insert_node_before(next_node, node)

    def remove(self, val: object):
        if self:
            if self.head.val == val:
                self.popleft()
                return
            else:
                prev_node = self.head
                while prev_node.next:
                    if prev_node.next.val == val:
                        self._remove_node_after(prev_node)
                        return
                    prev_node = prev_node.next
        raise ValueError(f'{self.__class__.__name__}.remove(val): val not in {self.__class__.__name__}')

    def reverse(self):
        # Modify a Linked List only if its length is greater than 1 as empty or single-element
        # Linked List will remain unchanged after having been reversed
        if len(self) > 1:
            curr_node = self.head

            while curr_node:
                next_node = curr_node.next
                # Swap pointers of the current node
                curr_node.next, curr_node.prev = curr_node.prev, curr_node.next
                curr_node = next_node

            # Swap a tail pointer with a head pointer
            self.head, self.tail = self.tail, self.head

    def rotate(self):
        if self:
            self.appendleft(self.pop())

    def _traverse_from_left(self, idx: int) -> 'previous Node object':
        if idx <= 0:  # We cannot return a node previous to the first one
            return None
        curr_node = self.head
        curr_idx = 1
        while curr_idx < idx:
            curr_node = curr_node.next
            curr_idx += 1
        return curr_node

    def _traverse_from_right(self, idx: int) -> 'next Node object':
        if idx >= len(self)-1:  # We cannot return a node next to the last one
            return None
        curr_node = self.tail
        curr_idx = len(self)-1
        while curr_idx > idx:
            curr_node = curr_node.prev
            curr_idx -= 1
        return curr_node

    def _insert_node_after(self, prev_node: Node, curr_node: Node):
        curr_node.next = prev_node.next
        curr_node.prev = prev_node
        prev_node.next = curr_node
        curr_node.next.prev = curr_node
        self.length += 1

    def _insert_node_before(self, next_node: Node, curr_node: Node):
        curr_node.next = next_node
        curr_node.prev = next_node.prev
        next_node.prev = curr_node
        curr_node.prev.next = curr_node
        self.length += 1

    def _remove_node_after(self, prev_node: Node):
        if prev_node.next is self.tail:
            self.tail = prev_node
        prev_node.next = prev_node.next.next
        if prev_node.next:
            prev_node.next.prev = prev_node
        self.length -= 1
```

Kilka testów

In [6]:
```python
a = DoublyLinkedList(range(5))
print(a)
a.append(5)
print(a)
a.appendleft(-1)
print(a)
a.clear()
print(a)
a.extend(range(5, 15, 3))
print(a)
b = a.copy()
b.extendleft(range(0, -15, -4))
print(b, a, sep='\t')
a.insert(-1, 5)
print(a)
a.insert(12345, 5)
print(a)
a.insert(4, 5)
print(a)
print(a.count(5))
print(a.index(14), a.index(5), a.index(8), sep='\t')
print(*(a.popleft() for _ in range(4)))
print(a)
a.remove(5)
print(a)
a.remove(5)
print(a)
a.reverse()
print(a)
print(b)
b.reverse()
print(b)
print(a.popleft())
# print(a.popleft())  # this is to check if an exception is properly raised
print(b)
b.rotate()
print(b)
```

```
0 <-> 0 <-> 1 <-> 2 <-> 3 <-> 4
0 <-> 0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5
-1 <-> 0 <-> 0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5

5 <-> 5 <-> 8 <-> 11 <-> 14
-12 <-> -8 <-> -4 <-> 0 <-> 5 <-> 5 <-> 5 <-> 8 <-> 11 <-> 14	5 <-> 5 <-> 8 <-> 11 <-> 14
5 <-> 5 <-> 5 <-> 8 <-> 11 <-> 14
5 <-> 5 <-> 5 <-> 8 <-> 11 <-> 14 <-> 5
5 <-> 5 <-> 5 <-> 8 <-> 11 <-> 5 <-> 14 <-> 5
5
6	0	3
5 5 5 8
11 <-> 5 <-> 14 <-> 5
11 <-> 14 <-> 5
11 <-> 14
11 <-> 14
-12 <-> -8 <-> -4 <-> 0 <-> 5 <-> 5 <-> 5 <-> 8 <-> 11 <-> 14
14 <-> 11 <-> 8 <-> 5 <-> 5 <-> 5 <-> 0 <-> -4 <-> -8 <-> -12
11
14 <-> 11 <-> 8 <-> 5 <-> 5 <-> 5 <-> 0 <-> -4 <-> -8 <-> -12
-12 <-> 14 <-> 11 <-> 8 <-> 5 <-> 5 <-> 5 <-> 0 <-> -4 <-> -8
```

Implementacja struktury #2

(Implementacja funkcyjna)

Warto mieć na uwadze, że w tym przypadku lista jest reprezentowana, przy pomocy dwóch wskaźników (jeden na początek i drugi na koniec)

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


def create_doubly_linked_list(values: 'iterable' = None) -> ('linked list head (sentinel)', 'linked list tail
    curr = head = Node()  # A head sentinel node
    tail = Node()  # A tail sentinel node
    if values:
        for val in values:
            node = Node(val)
            curr.next = node
            node.prev = curr
            curr = curr.next
    else:
        curr = head
    curr.next = tail
    tail.prev = curr
    return head, tail


def print_doubly_linked_list(ll_head: 'linked list head (sentinel)'):
    curr = ll_head.next
    print(ll_head.val, end=' ')
    while curr:
        print('<->', curr.val, end=' ')
        curr = curr.next
    print()


# This operation below is now efficient
def append_to_doubly_linked_list(ll_tail: 'linked list tail (sentinel)', val: object):
    node = Node(val)
    node.prev = ll_tail.prev
    ll_tail.prev.next = node
    node.next = ll_tail
    ll_tail.prev = node


# This operation is now efficient
def extend_doubly_linked_list(ll_tail: 'linked list tail (sentinel)', values : 'iterable'):
    for val in values:
        append_to_doubly_linked_list(ll_tail, val)


def appendleft_to_doubly_linked_list(ll_head: 'linked list head (sentinel)', val: object):
    node = Node(val)
    node.next = ll_head.next
    node.next.prev = node
    node.prev = ll_head
    ll_head.next = node


def extendleft_doubly_linked_list(ll_head: 'linked list head (sentinel)', values: 'iterable'):
    # Note that extendleft adds values in a reversed order so the first value of a linked
    # list will be the last value of the 'values' iterable (the same rule applies to the Python's
    # deque data structure, which is, in fact, a doubly linked list)
    for val in values:
        appendleft_to_doubly_linked_list(ll_head, val)


def popleft_from_doubly_linked_list(ll_head: 'linked list head (sentinel)') -> 'removed value':
    # ll_head.next will be a tail sentinel if a list is empty, we have to check ll_head.next.next
    if not ll_head.next.next:
        raise IndexError('pop from an empty doubly linked list')
    removed = ll_head.next.val
    ll_head.next = ll_head.next.next
    ll_head.next.prev = ll_head
    return removed


def pop_from_doubly_linked_list(ll_tail: 'linked list tail (sentinel)') -> 'removed value':
    # ll_tail.prev will be a head sentinel if a list is empty, we have to check ll_tail.prev.prev
    if not ll_tail.prev.prev:
        raise IndexError('pop from an empty doubly linked list')
    removed = ll_tail.prev.val
    ll_tail.prev = ll_tail.prev.prev
    ll_tail.prev.next = ll_tail
    return removed


def clear_doubly_linked_list(ll_head: 'linked list head (sentinel)', ll_tail: 'linked list tail (sentinel)'):
    ll_head.next = ll_tail
    ll_tail.prev = ll_head


def copy_doubly_linked_list(ll_head: 'linked list head (sentinel)') -> ('linked list copy head (sentinel)',
                                                                       'linked list copy tail (sentinel)'):
    copy_head = copy_curr = Node()  # A sentinel node
    copy_tail = Node()  # A sentinel node
    curr = ll_head.next
    while curr.next:
        node = Node(curr.val)
        copy_curr.next = node
        node.prev = copy_curr
        copy_curr = copy_curr.next
        curr = curr.next
    copy_tail.prev = copy_curr
    copy_curr.next = copy_tail
    return copy_head, copy_tail


def count_value_in_doubly_linked_list(ll_head: 'linked list head (sentinel)', val: object) -> int:
    curr = ll_head.next
    total = 0
    while curr:
        if curr.val == val:
            total += 1
        curr = curr.next
    return total


def index_of_value_in_doubly_linked_list(ll_head: 'linked list head (sentinel)', val: object) -> int:
    idx = 0
    curr = ll_head.next
    while curr.next:
        if curr.val == val:
            return idx
        curr = curr.next
        idx += 1
    raise ValueError(f'{val} not in linked list')


def insert_in_doubly_linked_list_from_left(ll_head: 'linked list head (sentinel)', idx_from_left: int, val: obj
    if idx_from_left <= 0:
        appendleft_to_doubly_linked_list(ll_head, val)
    else:
        prev_node = _traverse_from_left(ll_head, idx_from_left)
        _insert_node_after(prev_node, Node(val))


def insert_in_doubly_linked_list_from_right(ll_tail: 'linked list tail (sentinel)', idx_from_right: int, val: c
    if idx_from_right <= 0:
        appendright_to_doubly_linked_list(ll_tail, val)
    else:
        next_node = _traverse_from_right(ll_tail, idx_from_right)
        _insert_node_before(next_node, Node(val))


def remove_from_doubly_linked_list(ll_head: 'linked list head (sentinel)', val: object):
    if ll_head.next.next:
        prev_node = ll_head
        while prev_node.next.next:
            if prev_node.next.val == val:
                _remove_node_after(prev_node)
                return
            prev_node = prev_node.next
    raise ValueError('linkedlist.remove(val): val not in linkedlist')


def reverse_doubly_linked_list(ll_head: 'linked list head (sentinel)', ll_tail: 'linked list tail (sentinel)'):
    # Reverse a linked list only if there are at least 2 values
    # (in other cases reversing is pointless)
    if ll_head.next.next and ll_head.next.next.next:
        curr_node = ll_head.next.next
        # Swap sentinels' pointers
        temp = ll_head.next
        ll_head.next = ll_tail.prev
        ll_tail.prev = temp
        # Fix pointers of the nodes closest to the sentinels
        ll_head.next.next = ll_head.next.prev
        ll_head.next.prev = ll_head
        ll_tail.prev.prev = ll_tail.prev.next
        ll_tail.prev.next = ll_tail
        # Reverse pointers of each of the remaining nodes
        while curr_node.prev is not ll_head:
            curr_node.prev, curr_node.next = curr_node.next, curr_node.prev
            curr_node = curr_node.prev


def rotate_doubly_linked_list(ll_head: 'linked list head (sentinel)', ll_tail: 'linked list tail (sentinel)'):
    # Perform an operation only if there are at least 2 elements in a list
    if ll_head.next.next:
        appendleft_to_doubly_linked_list(ll_head, pop_from_doubly_linked_list(ll_tail))


def doubly_linked_list_to_list(ll_head: 'linked list head (sentinel)') -> list:
    values = []
    curr = ll_head.next
    while curr.next:
        values.append(curr.val)
        curr = curr.next
    return values


# If an index from left is too big (exceeds a number of elements in a linked list), the last node will be retur
def _traverse_from_left(ll_head: 'linked list head (sentinel)', idx_from_left: int) -> 'previous node object':
    curr = ll_head
    i = 0
    while curr.next.next:
        if i == idx_from_left:
            break
        curr = curr.next
        i += 1
    return curr


# If an index from right is too big (exceeds a number of elements in a linked list), the first node will be ret
def _traverse_from_right(ll_tail: 'linked list tail (sentinel)', idx_form_right: int) -> 'next node object':
    curr = ll_tail
    i = 0
    while curr.prev.prev:
        if i == idx_form_right:
            break
        curr = curr.prev
        i += 1
    return curr


def _insert_node_after(prev_node: 'node after which a curr_node will be inserted', curr_node):
    curr_node.next = prev_node.next
    curr_node.next.prev = curr_node
    curr_node.prev = prev_node
    prev_node.next = curr_node


def _insert_node_before(next_node: 'node before which a curr_node will be inserted', curr_node):
    curr_node.next = next_node
    curr_node.prev = next_node.prev
    next_node.prev = curr_node
    curr_node.prev.next = curr_node


def _remove_node_after(prev_node: 'node after which a curr_node will be inserted'):
    prev_node.next = prev_node.next.next
    prev_node.next.prev = prev_node
```

Kilka testów

```
a = create_doubly_linked_list(range(5))
a_head, a_tail = a
print_doubly_linked_list(a_head)
append_to_doubly_linked_list(a_tail, 5)
print_doubly_linked_list(a_head)
appendleft_to_doubly_linked_list(a_head, -1)
print_doubly_linked_list(a_head)
clear_doubly_linked_list(a_head, a_tail)
extend_doubly_linked_list(a_tail, range(5, 15, 3))
print_doubly_linked_list(a_head)
b_head, b_tail = copy_doubly_linked_list(a_head)
extendleft_doubly_linked_list(b_head, range(0, -15, -4))
print_doubly_linked_list(b_head)
print_doubly_linked_list(a_head)
print_doubly_linked_list(a_head)
insert_in_doubly_linked_list_from_left(a_head, -1, 5)
print_doubly_linked_list(a_head)
insert_in_doubly_linked_list_from_right(a_tail, 12345, 5)
print_doubly_linked_list(a_head)
insert_in_doubly_linked_list_from_left(a_head, 4 ,5)
print_doubly_linked_list(a_head)
print(count_value_in_doubly_linked_list(a_head, 5))
print(index_of_value_in_doubly_linked_list(a_head, 14),
      index_of_value_in_doubly_linked_list(a_head, 5),
      index_of_value_in_doubly_linked_list(a_head, 8),
      sep='\t')
print(*(popleft_from_doubly_linked_list(a_head) for _ in range(4)))
print_doubly_linked_list(a_head)
remove_from_doubly_linked_list(a_head, 5)
print_doubly_linked_list(a_head)
reverse_doubly_linked_list(a_head, a_tail)
print_doubly_linked_list(a_head)
print_doubly_linked_list(b_head)
reverse_doubly_linked_list(b_head, b_tail)
print_doubly_linked_list(b_head)
print(popleft_from_doubly_linked_list(a_head))
print_doubly_linked_list(a_head)
remove_from_doubly_linked_list(a_head, 11)
print_doubly_linked_list(a_head)
extendleft_doubly_linked_list(a_head, [123, 321])
rotate_doubly_linked_list(a_head, a_tail)
print_doubly_linked_list(a_head)
# print(popleft_from_doubly_linked_list(a_head))  # this is to check if an exception is properly raised
print_doubly_linked_list(b_head)
rotate_doubly_linked_list(b_head, b_tail)
print_doubly_linked_list(b_head)
print(doubly_linked_list_to_list(b_head))
```

```
None <-> 0 <-> 1 <-> 2 <-> 3 <-> 4 <-> None
None <-> 0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> None
None <-> -1 <-> 0 <-> 1 <-> 2 <-> 3 <-> 4 <-> 5 <-> None
None <-> 5 <-> 8 <-> 11 <-> 14 <-> None
None <-> -12 <-> -8 <-> -4 <-> 0 <-> 5 <-> 8 <-> 11 <-> 14 <-> None
None <-> 5 <-> 8 <-> 11 <-> 14 <-> None
None <-> 5 <-> 8 <-> 11 <-> 14 <-> None
None <-> 5 <-> 5 <-> 8 <-> 11 <-> 14 <-> None
None <-> 5 <-> 5 <-> 5 <-> 8 <-> 11 <-> 14 <-> None
None <-> 5 <-> 5 <-> 5 <-> 8 <-> 5 <-> 11 <-> 14 <-> None
4
6       0       3
5 5 5 8
None <-> 5 <-> 11 <-> 14 <-> None
None <-> 11 <-> 14 <-> None
None <-> 14 <-> 11 <-> None
None <-> -12 <-> -8 <-> -4 <-> 0 <-> 5 <-> 8 <-> 11 <-> 14 <-> None
None <-> 14 <-> 11 <-> 8 <-> 5 <-> 0 <-> -4 <-> -8 <-> -12 <-> None
14
None <-> 11 <-> None
None <-> None
None <-> 123 <-> 321 <-> None
None <-> 14 <-> 11 <-> 8 <-> 5 <-> 0 <-> -4 <-> -8 <-> -12 <-> None
None <-> -12 <-> 14 <-> 11 <-> 8 <-> 5 <-> 0 <-> -4 <-> -8 <-> None
[-12, 14, 11, 8, 5, 0, -4, -8]
```

Kolejka

UWAGA:

W poniższych przykładach znaduje się tylko implementacja zwykłej kolejki. Kolejka priorytetowa została umieszczona w pliku ze strukturami drzewiastymi, ponieważ opiera się ona na kopcach binarnych, które reprezentują kompletne drzewa binarne.

Implementacja struktury #1 (w oparciu o listę jednokierunkową)

(Implementacja obiektowa)

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


class Queue:
    def __init__(self, values: 'iterable' = None):
        self.head = self.tail = None
        self.length = 0
        values and self.enqueue_many(values)

    def __iter__(self):
        curr = self.head
        while curr:
            yield curr.val
            curr = curr.next

    def __str__(self):
        # An arrow indicates in which direction a queue moves
        return ' <- '.join(map(str, self))

    def __len__(self):
        return self.length

    def is_empty(self):
        return not bool(self)

    def peek(self):
        if self:
            return self.head.val

    def enqueue(self, val: object):
        node = Node(val)
        if not self:
            self.head = self.tail = node
        else:
            self.tail.next = node
            self.tail = node
        self.length += 1

    def dequeue(self) -> object:
        if not self:
            raise IndexError(f'dequeue from an empty {self.__class__.__name__}')
        removed = self.head.val
        if len(self) == 1:
            self.head = self.tail = None
        else:
            self.head = self.head.next
        self.length -= 1
        return removed

    def enqueue_many(self, values: 'iterable'):
        if values:
            iterator = iter(values)
            if not self:
                self.head = self.tail = Node(next(iterator))
            for val in iterator:
                self.tail.next = Node(val)
                self.tail = self.tail.next
            self.length += len(values)
```

Kilka testów

```
q = Queue()
for n in range(5):
    q.enqueue(n)
print(q)

for _ in range(3):
    print('Removed:', q.dequeue())
print(q)

for i in range(10, 20, 3):
    q.enqueue(i)
print(q)

print(q.dequeue())
print(q)
print(q.is_empty())
print(q.peek())
q.enqueue(20)
print(q.peek())
print(q)

while q:
    print(q.dequeue(), end='\t')
print()

print(q, q.is_empty(), q.peek())
```

```
0 <- 1 <- 2 <- 3 <- 4
Removed: 0
Removed: 1
Removed: 2
3 <- 4
3 <- 4 <- 10 <- 13 <- 16 <- 19
3
4 <- 10 <- 13 <- 16 <- 19
False
4
4
4 <- 10 <- 13 <- 16 <- 19 <- 20
4       10      13      16      19      20
 True None
```

Implementacja struktury #2 (w oparciu o listę dwukierunkową)

(Implementacja funkcyjna)

W przypadku funkcyjnej implementacji kolejki konieczne jest skorzystanie z listy dwukierunkowej, w celu umożliwienia dodawania elementów na koniec kolejki orazz zdejmowania z początku w czasie $O(1)$. Oczywiście można by za każdym razem po zaktualizowaniu kolejki zwracać w funkcji wskaźnik do nowego ostatniego lub nowego pierwszego elementu, lecz aktualizowanie zmiennych wskaźnikowych uważam za niewygodne i podatne na błędy. Z tego powodu preferuję powyższą implementację obiektową, lub, jeżeli jest konieczna funkcyjna, poniższą implementację.

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


def create_queue(values: 'iterable' = None) -> ('queue head (sentinel)', 'queue tail (sentinel)'):
    head = Node()  # A head sentinel node
    tail = Node()  # A tail sentinel node
    if values:
        head.next = curr = Node(values[0])
        curr.prev = head
        for i in range(1, len(values)):
            node = Node(values[i])
            curr.next = node
            node.prev = curr
            curr = curr.next
    else:
        curr = head
    curr.next = tail
    tail.prev = curr
    return head, tail


def print_queue(head: 'queue head (sentinel)'):
    curr = head.next
    print(head.val, end=' ')
    while curr:
        print('<->', curr.val, end=' ')
        curr = curr.next
    print()


def enqueue(tail: 'queue tail (sentinel)', val: object):
    node = Node(val)
    node.prev = tail.prev
    node.prev.next = node
    node.next = tail
    tail.prev = node


def dequeue(head: 'queue head (sentinel)') -> 'removed value':
    if not head.next.next:
        raise IndexError('dequeue from an empty queue')
    removed = head.next.val
    head.next = head.next.next
    head.next.prev = head
    return removed


def enqueue_many(tail: 'queue tail (sentinel)', values: 'iterable'):
    for val in values:
        enqueue(tail, val)


def is_empty(head: 'queue head (sentinel)') -> bool:
    return not head.next.next
```

Kilka testów

```
q = create_queue()
q_head, q_tail = q
print_queue(q_head)
enqueue_many(q_tail, range(0, 10, 2))
print_queue(q_head)
print(dequeue(q_head), dequeue(q_head), dequeue(q_head))
enqueue(q_tail, 12)
print(dequeue(q_head))
print_queue(q_head)
while not is_empty(q_head):
    dequeue(q_head)
print_queue(q_head)
```

```
None <-> None
None <-> 0 <-> 2 <-> 4 <-> 6 <-> 8 <-> None
0 2 4
6
None <-> 8 <-> 12 <-> None
None <-> None
```

Implementacja struktury #3 (z użyciem stosów)

(Implementacja obiektowa)

Przykładowe implementacje stosów znajdują się niżej. Ponieważ ta implementacja jest raczej przedstawiona tutaj jako ciekawostka a nie użyteczna wersja struktury, zdecydowałem się ją umieścić przed przedstawieniem wariantów implementacji stosu.

Omówienie działania struktury

Szczegółowe wyjaśnienie działania struktury znajduje się w poniższym wideo:

https://www.youtube.com/watch?v=Wg8IiY1LbII

Właściwa implementacja

```
class CrazyQueue:
    def __init__(self, values: 'iterable' = None):
        self.first_stack = [] # A stack for elements that are enqueued (last on a top)
        self.last_stack = []  # A stack for elements that are dequeued (first on a top)
        if values: self.first_stack = values[:]

    def __len__(self):
        # The length of a queue will always be a sum of stacks' lengths
        return len(self.first_stack) + len(self.last_stack)

    def __iter__(self):
        # In the priorted representation we assume that elements on the right side are added
        # before elements on the left side. Arrow show how a queue moves, and therefore, they
        # are pointing from the last to the first element (they are directed from the last to
        # the first element in a queue)
        yield from self.first_stack[::-1]
        yield from self.last_stack

    def __str__(self):
        # An arrow indicates in which direction a queue moves
        return ' <- '.join(map(str, self))

    def is_empty(self):
        return not bool(self)

    def peek(self):
        # Returns the first element of a queue
        if self:
            return self.first_stack[-1] if self.first_stack else self.last_stack[0]

    def enqueue(self, val: object):  # O(1)
        # This stack is only recieving pushes so we don't have to do any
        # extra work here
        self.last_stack.append(val)

    def dequeue(self):  # Amortized O(1)
        # If there is no element in a last_stack, we have to move all values
        # to the first_stack
        if not self.first_stack:
            self.__move_stack(self.last_stack, self.first_stack)
        # If there is still no first_stack, our queue must be empty
        if not self.first_stack:
            raise IndexError(f'dequeue from an empty {self.__class__.__name__}')
        # The only thing we have to do is to pop from teh first stack
        return self.first_stack.pop()

    def __move_stack(self, initial_stack, target_stack):
        while initial_stack:
            target_stack.append(initial_stack.pop())
```

Kilka testów

```
cq = CrazyQueue()
for n in range(5):
    cq.enqueue(n)
print(cq)

for _ in range(3):
    print('Removed:', cq.dequeue())
print(cq)

for i in range(10, 20, 3):
    cq.enqueue(i)
print(cq)

print(cq.dequeue())
print(cq)
print(cq.is_empty())
print(cq.peek())
cq.enqueue(20)
print(cq.peek())
print(cq)

while cq:
    print(cq.dequeue(), end='\t')
print()

print(cq, cq.is_empty(), cq.peek())
```

```
0 <- 1 <- 2 <- 3 <- 4
Removed: 0
Removed: 1
Removed: 2
3 <- 4
3 <- 4 <- 10 <- 13 <- 16 <- 19
3
4 <- 10 <- 13 <- 16 <- 19
False
4
4
4 <- 10 <- 13 <- 16 <- 19 <- 20
4       10      13      16      19      20
 True None
```

Implementacja struktury #4 (z użyciem tablicy)

(Implementacja obiektowa) (Pełzająca kolejka)

Implementacja kolejki z użyciem tablicy jest o tyle ciekawa, że w normalnej sytuacji niemożliwe jest zwiększanie ani zmniejszanie tablicy z żadnej ze stron. W gruncie rzeczy, w poniższej implementacji tworzymy kolejkę, która ma stały rozmiar (ale możliwe jest zaimplementowanie amortyzacji tablicy, a więc przepisywanie wartości do większej tablicy wtedy, gdy jest to potrzebne). Za początek kolejki przyjmujemy element, który znajduje się pod indeksem zapisanym jako początkowy. Warto zauważyć, że może zdarzyć się (nawet często) sytuacja, gdy koniec kolejki będzie wcześniej w tablicy niż początek. Z tego powodu taką kolejkę nazywa się często pełzającą kolejką, ponieważ jej początek i koniec wędrują po tablicy (tak naprawdę nie przemieszczamy elementów, a jedynie poruszamy indeksami).

```
class CrawlingQueue:
    def __init__(self, size):
        self.arr = [None] * size
        self.max_length = size    # A maximum length of a queue
        self.length = 0           # A current length of a queue
        self.first_idx = 0

    def __len__(self):
        return self.length

    def __repr__(self):
        return f'{self.__class__.__name__}({self.max_length})'

    def __iter__(self):
        for i in range(self.first_idx, self.first_idx + self.length):
            yield self.arr[i % self.max_length]

    def __str__(self):
        # An arrow indicates a direction in which a queue moves
        return ' <- '.join(map(str, self))

    def is_empty(self):
        return not bool(self)

    def peek(self):
        # Returns the first element of a queue
        if self:
            return self.arr[self.first_idx]

    def enqueue(self, val):
        if self.length == self.max_length:
            raise OverflowError(f'{self.__class__.__name__} has no space left.')
        new_idx = (self.first_idx + self.length) % self.max_length
        self.arr[new_idx] = val
        self.length += 1

    def dequeue(self):
        # If a queue is empty, raise an Exception
        if not self:
            raise IndexError(f'dequeue from an empty {self.__class__.__name__}')
        removed = self.arr[self.first_idx]
        self.first_idx = (self.first_idx + 1) % self.max_length
        self.length -= 1
        return removed
```

Kilka testów

```
cq = CrawlingQueue(10)
for n in range(5):
    cq.enqueue(n)
print(cq)

for _ in range(3):
    print('Removed:', cq.dequeue())
print(cq)

for i in range(10, 20, 3):
    cq.enqueue(i)
print(cq)

print('How an array looks like:')
print(cq.arr)

print(cq.dequeue())
print(cq)
print(cq.is_empty())
print(cq.peek())
cq.enqueue(20)
print(cq.peek())
print(cq)

while cq:
    print(cq.dequeue(), end='\t')
print()

print(cq, cq.is_empty(), cq.peek())
print('How an array looks like:')
print(cq.arr)

for i in range(10):
    cq.enqueue(i)
print(cq)
print('How an array looks like:')
print(cq.arr)

cq.dequeue()
cq.dequeue()
cq.dequeue()
cq.enqueue('new')
cq.enqueue('another new')
print(cq.arr)
while cq:
    print(cq.dequeue(), end=' ')
```

```
0 <- 1 <- 2 <- 3 <- 4
Removed: 0
Removed: 1
Removed: 2
3 <- 4
3 <- 4 <- 10 <- 13 <- 16 <- 19
How an array looks like:
[0, 1, 2, 3, 4, 10, 13, 16, 19, None]
3
4 <- 10 <- 13 <- 16 <- 19
False
4
4
4 <- 10 <- 13 <- 16 <- 19 <- 20
4       10      13      16      19      20
 True None
How an array looks like:
[0, 1, 2, 3, 4, 10, 13, 16, 19, 20]
0 <- 1 <- 2 <- 3 <- 4 <- 5 <- 6 <- 7 <- 8 <- 9
How an array looks like:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
['new', 'another new', 2, 3, 4, 5, 6, 7, 8, 9]
3 4 5 6 7 8 9 new another new
```

Implementacja struktury #5 (z użyciem tablicy)

(Implementacja obiektowa) (Amortyzowana pełzająca kolejka - modyfikacja powyższej)

```
class AmortizedQueue:
    # _amort_factor is an amortization factor value which indicates how many times will
    # a size of an array be increased (while creating a new one) if its length was exceeded
    def __init__(self, _amortization_factor=2):
        self.arr = [None]  # In other languages we don't have to initialize an array
        self.length = 0    # A current length of a queue
        self.first_idx = 0
        self.amort_factor = _amortization_factor

    def __len__(self):
        return self.length

    def __repr__(self):
        return f'{self.__class__.__name__}({len(self.arr)})'

    def __iter__(self):
        for i in range(self.first_idx, self.first_idx + self.length):
            yield self.arr[i % len(self.arr)]

    def __str__(self):
        # An arrow indicates a direction in which a queue moves
        return ' <- '.join(map(str, self))

    def is_empty(self):
        return not bool(self)

    def peek(self):
        # Returns the first element of a queue
        if self:
            return self.arr[self.first_idx]

    def enqueue(self, val):
        # Increase the size of an array if we exceeded its length
        if self.length == len(self.arr):
            self.__expand_array()
        new_idx = (self.first_idx + self.length) % len(self.arr)
        self.arr[new_idx] = val
        self.length += 1

    def dequeue(self):
        # If a queue is empty, raise an Exception
        if not self:
            raise IndexError(f'dequeue from an empty {self.__class__.__name__}')
        # Decrease a length of an array only if much empty space is left
        if self.length <= len(self.arr) // (self.amort_factor * 2):
            self.__shrink_array()

        removed = self.arr[self.first_idx]
        self.first_idx = (self.first_idx + 1) % len(self.arr)
        self.length -= 1
        return removed

    def __expand_array(self):
        self.arr = self.__new_array(len(self.arr) * self.amort_factor)
        # Reset the first_idx value as all the values are now placed at the beginning
        # of an array
        self.first_idx = 0

    def __shrink_array(self):
        # The same rule as above, we also assume it works in constant time
        self.arr = self.__new_array(len(self.arr) // self.amort_factor)
        # Reset the first_idx value as well
        self.first_idx = 0

    def __new_array(self, length):
        # We assume that initialization of a new array costs us constant time
        # (in Python it's impossible without extra modules which has arrays implemented)
        new_arr = [None] * length
        # Rewrite values from the first array to a new one
        for i in range(self.length):
            new_arr[i] = self.arr[(self.first_idx + i) % len(self.arr)]
        return new_arr
```

Kilka testów

```
aq = AmortizedQueue()
for n in range(5):
    aq.enqueue(n)
print(aq)

for _ in range(3):
    print('Removed:', aq.dequeue())
print(aq)

for i in range(10, 20, 3):
    aq.enqueue(i)
print(aq)

print('How an array looks like:')
print(aq.arr)

print(aq.dequeue())
print(aq)
print(aq.is_empty())
print(aq.peek())
aq.enqueue(20)
print(aq.peek())
print(aq)

while aq:
    print(aq.dequeue(), end='\t')
print()

print(aq, aq.is_empty(), aq.peek())
print('How an array looks like:')
print(aq.arr)

for i in range(9):
    aq.enqueue(i)
print(aq)
print('How an array looks like:')
print(aq.arr)

while aq:
    print(aq.dequeue(), aq)
    print(aq.arr)
```