Linked lists

1 Introduction

A **linked list** is a linear collection of data elements, whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence.

Looking ahead, we compare complexity of main functions with same functions in dynamic arrays (tab. 1). Unlike arrays, lists do not provide random access to elements. For this reason, indexing and inserting in an arbitrary location work for linear time. Work with the end of the list depends on the presence of the pointer to the end. Another difference in the lists is the ability to quickly insert and delete at the beginning, which is important in the implementation of some data structures.

Table 1: Time complexity comparison for main functions.

Operation	**Linked lists**	**Dynamic arrays**
Indexing	$O(n)$	$O(1)$
Insert/delete at beginning	$O(1)$	$O(n)$
Insert/delete at end	$O(1)$ or $O(n)$	$O(1)$ amortized
Insert/delete in middle	$O(n)$	$O(n)$

Linked lists used in implementation of

- stack,
- queue,
- deque,
- associative arrays.

Depending on the use, linked lists are implemented in different ways. Let's look at the main implementations.

Note: we'll use dicts to represent nodes. Function get_node() will be used for creating a new node.

2 Singly linked list

A **singly linked list** is a linked list, where each node contains one pointer, a pointer to the next node in the list (fig. 1). The last node always refers to `NULL`. A general view of a singly linked list is shown in the figure 2.

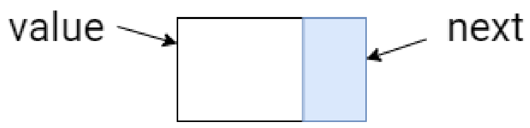

Figure 1: Singly linked list's node structure

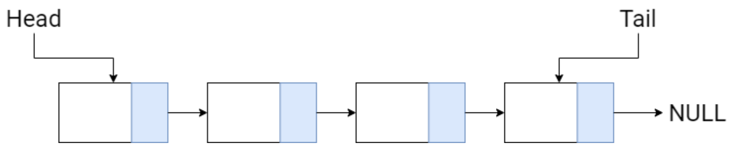

Figure 2: Singly linked list

Note that there are two pointers – `head` and `tail`. The first one refers to the beginning, the second one refers to the end. In some cases `tail` is not used and can be omitted. Let's write a singly linked list in context of stack and queue implementation.

2.1 Stack

A **stack** is an abstract data type that works by the LIFO (last in, first out) principle, i.e. allows operations only with top element. Stack supports following operations:

- push – add to stack,
- pop – delete from stack,
- peek – look at the top element,
- length – size of stack.

To implement the stack, it is sufficient to use a dynamic array or a singly linked list without a pointer to the end. To determine the stack size, you just need to support an additional variable that stores the current stack size.

```
def get_node(v):
    return {
        "value": v,
        "next": None
    }


def enqueue(head, tail, value):
    tmp = get_node(value)
    if head is None:
        return tmp, tmp
    tail["next"] = tmp
    return head, tmp


def dequeue(head, tail):
    if head is None:
        print("Queue is empty")
        return None, None, None
    if id(head) == id(tail):
        return None, None, head["value"]
    return head["next"], tail, head["value"]


def peek(head):
    if head is None:
        print("Queue is empty")
        return None
    return head["value"]
```

2.2 Queue

A **queue** is an abstract data type that works by the FIFO (first in, first out) principle, i.e. the first added element will be the first one to be removed. Queue supports following operations:

- enqueue – add to queue,
- dequeue – delete from queue,
- peek – show first element,
- length – queue size.

To implement the queue, it is sufficient to use a singly linked list with pointer to the end.

```
def get_node(v):
    return {
        "value": v,
        "next": None
    }


def push(head, value):
    tmp = get_node(value)
    tmp["next"] = head
    return tmp


def pop(head):
    if head is None:
        print("Stack is empty")
        return None, None
    return head["next"], head["value"]


def peek(head):
    if head is None:
        print("Stack is empty")
        return None
    return head["value"]
```

3 Doubly linked list

A **doubly linked list** is a linked list, where each node contains pointers to the next node and to the previous node (fig. 3). The last one and the first one refers to `NULL`. A general view of a doubly linked list is shown in the figure 4.

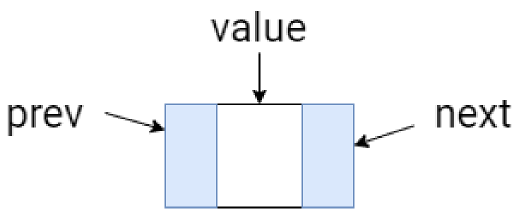

Figure 3: Doubly linked list's node structure

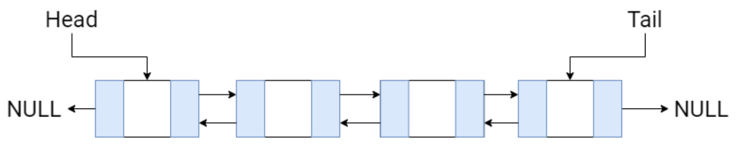

Figure 4: Doubly linked list

Doubly linked lists can be used to implement a stack and a queue, but this is not an optimal solution, since in this case, the presence of a backward link is a waste of memory. However, for the deque it is necessary.

3.1 Deque

A **double-ended queue** or **deque** is an abstract data type that generalizes queue. It provides access to the elements from both sides. Deque supports following operations:

- push front/back – add to the beginning/end,
- pop front/back – delete from the beginning/end,
- peek front/back – show the first/last element,
- length – deque size.

To implement the deck, you need a doubly linked list with a pointer to the end.

```python
def get_node(v):
    return {
        "value": v,
        "next": None,
        "prev": None
    }


def push_front(head, tail, value):
    tmp = get_node(value)
    if head is None:
        return tmp, tmp
    tmp["next"] = head
    head["prev"] = tmp
    return tmp, tail


def push_back(head, tail, value):
    tmp = get_node(value)
    if head is None:
        return tmp, tmp
    tmp["prev"] = tail
    tail["next"] = tmp
    return head, tmp


def pop_front(head, tail):
    if head is None:
        print("Deque is empty")
        return None, None, None
    if id(head) == id(tail):
        return None, None, head["value"]
    head["next"]["prev"] = None
    return head["next"], tail, head["value"]


def pop_back(head, tail):
    if head is None:
        print("Deque is empty")
        return None, None, None
    if id(head) == id(tail):
        return None, None, head["value"]
    tail["prev"]["next"] = None
    return head, tail["prev"], tail["value"]


def peek_front(head):
    if head is None:
        print("Deque is empty")
        return None
    return head["value"]


def peek_back(tail):
    if tail is None:
        print("Deque is empty")
        return None
    return tail["value"]
```

4 Circular linked list

A **circular linked list** is a linked list, where the last element linked with the first one. In fact, such a list has no end, and the beginning is determined only by the `head`. A circular list can be represented by either a singly linked (fig. 5) or a doubly linked (fig. 6) list.

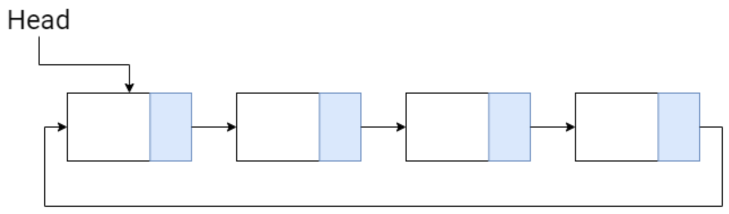

Figure 5: Circular linked list based on singly linked list

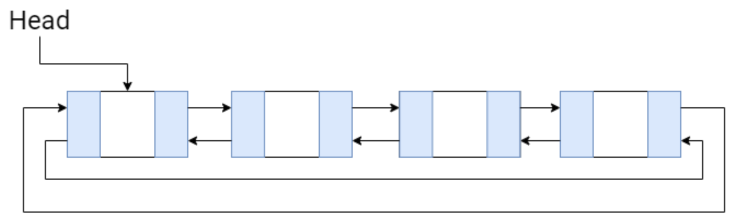

Figure 6: Circular linked list based on doubly linked list

The implementation of functions in the circular list is not much different from its non-cyclic counterparts. Circular lists themselves are rarely used. For example, to maintain the structure of a polygon when it is important to know the order of the vertices in it.