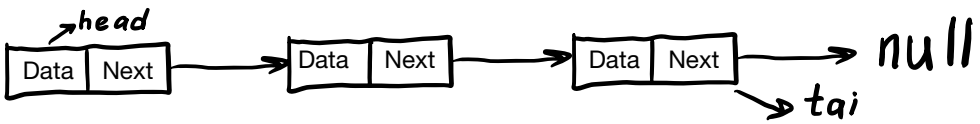
Связный список - это структура данных, в которой несколько значений хранятся линейно. Каждое значение содержит своё собственное значение узла, а также содержит данные вместе со ссылкой на следующий узел в списке. Ссылка - это указатель на другой объект узла или на null, если следующего узла нет. Если у каждого узла есть только один указатель на другой узел (чаще всего называется next), то этот список считается односвязный (singly linked list); тогда как если у каждого узла есть две ссылки (обычно previous и next), то он считается двусвязный (doubly linked list).



Поле data хранит в себе любые данные, а поле next хранит в себе ссылку на следующий элемент

```ts
type TNodeValue = string | number | { [key: string]: any };

class LinkedListNode {
    value: TNodeValue;
    next: null | LinkedListNode;

    constructor(value: TNodeValue, next: null | LinkedListNode = null) {
        this.value = value;
        this.next = next;
    }

    toString(callback?: (value: TNodeValue) => string) {
        return callback ? callback(this.value) : `${this.value}`;
    }
}
```

Это реализация класса одного элемента связанного списка (ноды)

```ts
class LinkedList {
    head: null | LinkedListNode = null;
    tail: null | LinkedListNode = null;
}
```

Это реализация самого списка. У списка всегда есть начало (head) и конец (tail)

Метод append - принимает значение и создаёт новый узел с этим значением, помещая его в конец связного списка;

```ts
/**
 * Принимает значение и создаёт новый узел с этим значением,
 * помещая его в конец связного списка
 */
append(value: TNodeValue) {
    const newNode = new LinkedListNode(value);

    if (!this.head || !this.tail) {
        this.head = newNode;
        this.tail = newNode;

        return this;
    }

    /**
     * Поле next у последнего элемента списка устанавливаем
     * равным новой ноде
     */
    this.tail.next = newNode;
    /**
     * Устанавливаем последний элемент равным новой ноде
     */
    this.tail = newNode;

    return this;
}
```

1) append('a');
head: {val: 'a'; next: null}; tail: {val: 'a'; next: null}

2) append('b');
this.tail.next = {val: 'b'; next: null}
head: {val: 'a'; next: {val: 'b'; next: null}}

Значение next у head поменялось, так как tail и head ссылаются на один и тот же блок в памяти. Мы поменяли next у tail, а следовательно и у head next тоже поменялся.

this.tail = {val: 'b'; next: null}

3) append('c')
new Node = {val: 'c'; next: null}
head: {val: a; next: {val: b; next: {val: c; next: null}}}
tail: {val: c; next: null}

Одним цветом помечены одни и тоже блоки памяти в компьютере

Метод prepend - метод принимает значение в качестве аргумента и создаёт новый узел с этим значением, помещая его в начало связного списка.

```
/**
 * Метод принимает значение в качестве аргумента
 * и создаёт новый узел с этим значением, помещая его в начало связного списка.
 */
prepend(value: TNodeValue) {
    /**
     * Создаём новый узел, который будет новым head,
     * при создании передаем второй аргумент, который указывает
     * что его "next" будет текущий head,
     * так как новый узел будет стоять перед текущем head.
     */
    const newNode = new LinkedListNode(value, this.head);

    this.head = newNode;

    /**
     * Если у списка еще не было tail, то устанавливаем  tail
     * равным новой ноде
     */
    if (!this.tail) this.tail = newNode;

    return this;
}
```

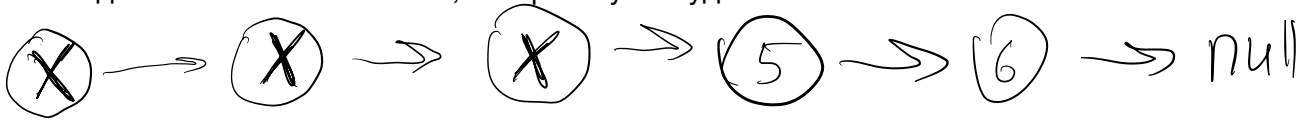
Метод find - метод принимает значение в качестве аргумента, находит первый узел с таким же значением и возвращает его.

```
/**
 * Метод принимает значение в качестве аргумента,
 * находит первый узел с таким же значением и возвращает его.
 */
find(value: TNodeValue): null | LinkedListNode {
    /** Если нет head значит список пуст. */
    if (!this.head) return null;

    let currentNode: null | LinkedListNode = this.head;

    /** Перебираем все узлы в поиске значения. */
    while (currentNode) {
        if (currentNode.value === value) return currentNode;
        currentNode = currentNode.next;
    }
    return null;
}
```

Метод delete - метод принимает значение в качестве аргумента, удаляет все узлы, которые имеют указаное значение и возвращает последний удалённый узел.

Алгоритм удаления:

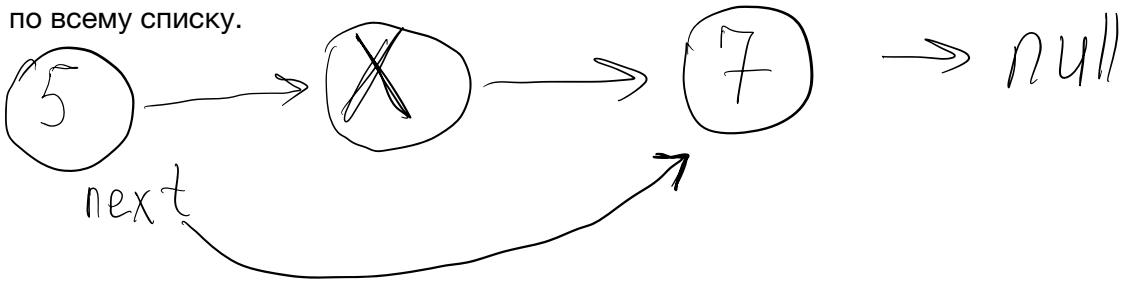
1. Когда элемент или элементы, которые нужно удалить являются head списка



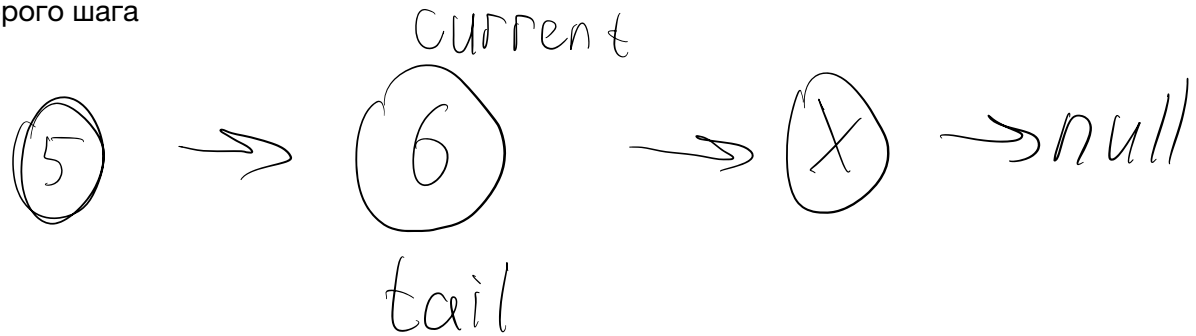
Крестом помечены те элементы, которые нужно удалить.

2. Далее устанавливает элемент равным head (после предыдущего шага head не может быть равно значению, которое нужно удалить. Если next у текущего элемента равен значению, которое нужно то перезаписываем next у текущего элемента на следующего next (через один или next.next) Иначе если next у текущего элемента не равен значению, которое нужно удалить, то перезаписываем текущий элемент на следующий(next). Так происходит итерация

по всему списку.

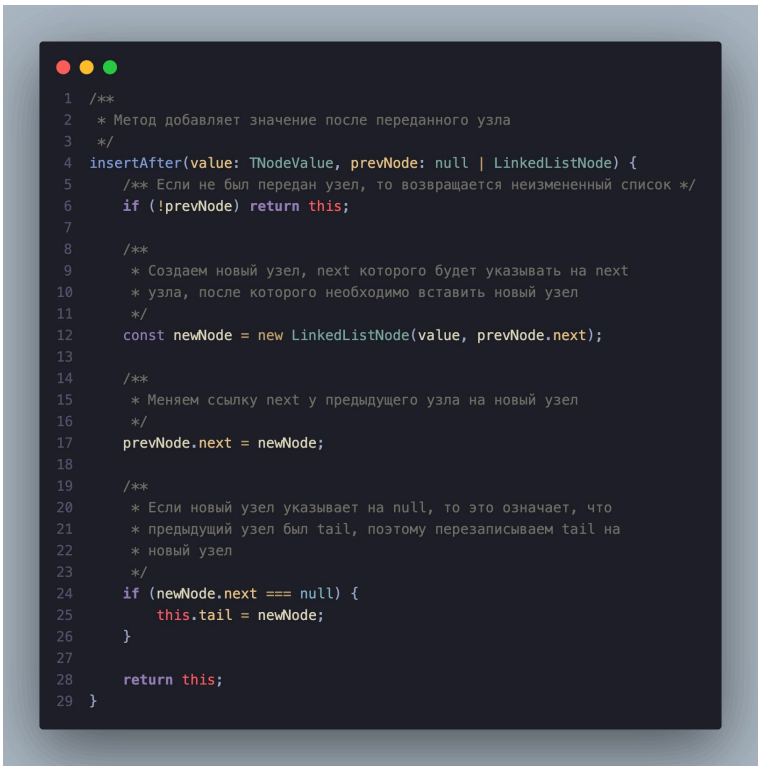


3. Если tail равен значению, которое нужно удалить, то нужно tail присвоить значение ноды из второго шага




```ts
/**
 * Метод принимает значение в качестве аргумента,
 * удаляет все узлы, которые имеют указанное значение и возвращает последний удалённый узел.
 */
delete(value: TNodeValue): null | LinkedListNode {
    let deletedNode = null;

    /**
     * Если head должен быть удален, то делаем следующий узел новым head.
     */
    while (this.head && this.head.value === value) {
        deletedNode = this.head;
        this.head = this.head.next;
    }

    let currentNode = this.head;

    /**
     * Если следующий узел должен быть удален,
     * то перезаписывает next у текущего значением через 1 узел
     * (next.next)
     * Иначе перезаписываем текущий узел на следующий для итерации
     */
    while (currentNode && currentNode.next) {
        if (currentNode.next.value === value) {
            deletedNode = currentNode.next;
            currentNode.next = currentNode.next.next;
        } else {
            currentNode = currentNode.next;
        }
    }

    /**
     * Если нужно удалить tail, то перезаписываем tail
     * на currentNode. Так как после второго шага
     * currentNode будет последним узлом, который не
     * нужно удалять
     */
    if (this.tail && this.tail.value === value) {
        this.tail = currentNode;
    }

    return deletedNode;
}
```

Метод insertAfter - Метод добавляет значение после переданного узла



```
/**
 * Метод добавляет значение после переданного узла
 */
insertAfter(value: TNodeValue, prevNode: null | LinkedListNode) {
    /** Если не был передан узел, то возвращается неизмененный список */
    if (!prevNode) return this;

    /**
     * Создаем новый узел, next которого будет указывать на next
     * узла, после которого необходимо вставить новый узел
     */
    const newNode = new LinkedListNode(value, prevNode.next);

    /**
     * Меняем ссылку next у предыдущего узла на новый узел
     */
    prevNode.next = newNode;

    /**
     * Если новый узел указывает на null, то это означает, что
     * предыдущий узел был tail, поэтому перезаписываем tail на
     * новый узел
     */
    if (newNode.next === null) {
        this.tail = newNode;
    }

    return this;
}
```

Метод deleteTail - Метод, который удаляет последний узел из списка и возвращает его.

```
/**
 * Метод, который удаляет последний узел из списка и возвращает его.
 */
deleteTail(): null | LinkedListNode {
    /** Если нет tail, то список пуст */
    if (!this.tail) return null;

    let deletedNode = this.tail;

    /**
     * Если head и tail равны, то список состоит из одного узла
     */
    if (this.head === this.tail) {
        this.head = null;
        this.tail = null;
        return deletedNode;
    }

    let current = this.head;

    /**
     * Если в списке много элементов, то находим предпоследним и
     * устанавливаем его next в null
     */
    while (current && current.next) {
        if (current.next.next === null) {
            current.next = null;
        } else {
            current = current.next;
        }
    }

    this.tail = current;

    return deletedNode;
}
```

Метод deleteHead - Метод, который удаляет из списка первый узел и возвращает его.

```
/**
 * Метод, который удаляет из списка первый узел и возвращает его.
 */
deleteHead(): null | LinkedListNode {
    /** Если нет head, то список пуст */
    if (!this.head) return null;

    const deletedNode = this.head;

    /**
     * Если у head next не равен null, то устанавливаем
     * новый head в значение next.
     * Иначе устанавливаем head и tail в null,
     * так как в списке всего 1 узел
     */
    if (this.head.next) {
        this.head = this.head.next;
    } else {
        this.head = null;
        this.tail = null;
    }

    return deletedNode;
}
```