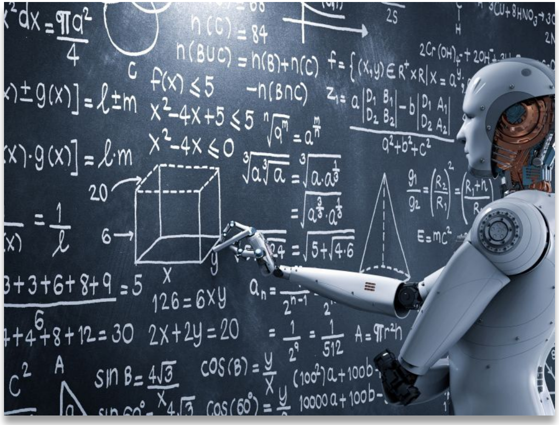
Python-9
Алгоритмы и структуры данных Python. Деревья. ХЕШ-таблицы.

Алгоритмы и структуры данных.

Алгоритм - это последовательность шагов или инструкций, предназначенных для решения конкретной задачи или выполнения определенной операции. *Например*, линейный/бинарный поиск, сортировка пузырьком/вставками.. и др.

Структуры данных - это способы организации и хранения данных для обеспечения эффективного доступа и модификации. *Например*, LIST/TUPLE/SET/DICT/... , и дополнительные структуры такие как - ХЕШ-таблицы, деревья, и др.

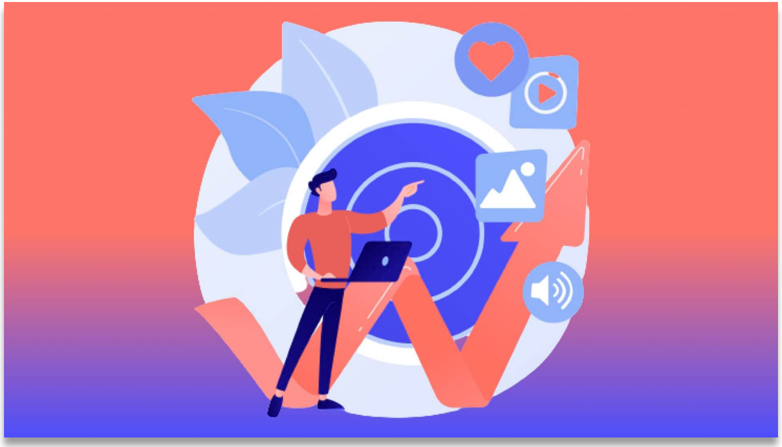


Эффективность Алгоритмов.

Значение эффективных алгоритмов и структур данных в программировании:
Эффективные алгоритмы и структуры данных играют ключевую роль в процессе разработки программного обеспечения, позволяя оптимизировать время выполнения программ, использовать ресурсы компьютера более эффективно и обеспечивать быстрый доступ к данным.

Какие задачи решают алгоритмы?

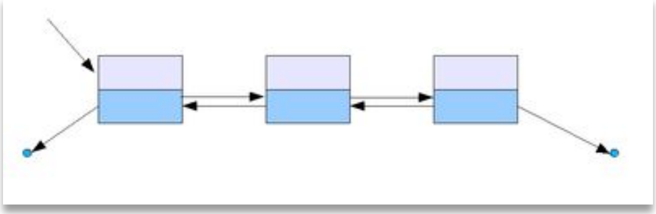
- Сортировка больших объемов данных.
- Поиск определенного элемента в списке.
- Обход графа для поиска кратчайшего пути.
- Управление памятью приложения с использованием различных структур данных.
- Поиск минимального/максимального элемента в списке.
- Линейный поиск в списке данных
- Бинарный поиск в списке данных
- ... и другие задачи.



Понимание основных концепций алгоритмов и структур данных является фундаментом для разработки эффективных программ.

Связные списки.

Связный список - это структура данных, представляющая собой последовательность элементов, в которой каждый элемент содержит ссылку на следующий элемент в цепочке. В отличие от обычных списков в Python, связные списки позволяют динамически изменять размер структуры данных. Связный список делится на два типа: односвязный и двусвязный списки.



Преимущества: Гибкость в изменении размера, не требуется выделение непрерывной области памяти.

Недостатки: Дополнительные указатели могут занимать дополнительное пространство.

Односвязный/Двусвязный списки.

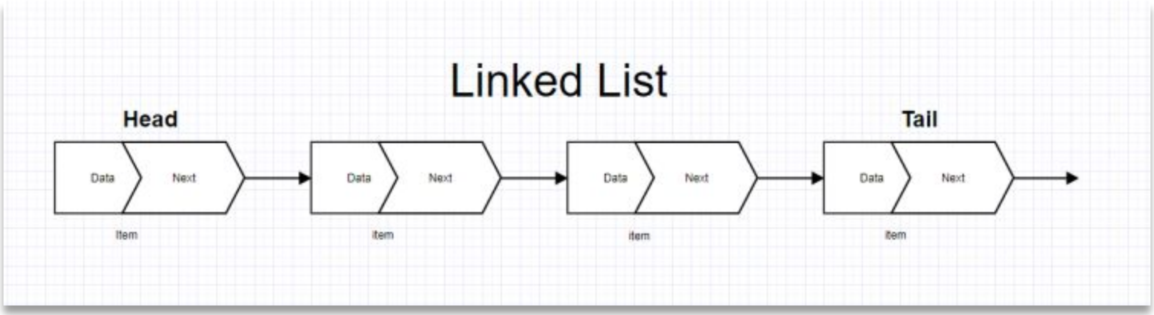
В **односвязном списке** каждый узел содержит данные и ссылку только на следующий узел в списке. Такая структура упрощает вставку и удаление элементов, но ограничивает обратный доступ к предыдущим элементам. При этом в **двусвязном списке** каждый узел содержит данные, ссылку на следующий узел и ссылку на предыдущий узел. Это позволяет более эффективно реализовывать операции вставки и удаления в середине списка, а также обеспечивает обратный доступ к предыдущим элементам.

Основными операциям являются:

- Вставка (Insertion): Добавление нового узла в список.
- Удаление (Deletion): Удаление узла из списка.
- Поиск (Search): Поиск узла с заданным значением.
- Обход (Traversal): Перебор всех узлов в списке.

Односвязный список.

Структура односвязного списка.

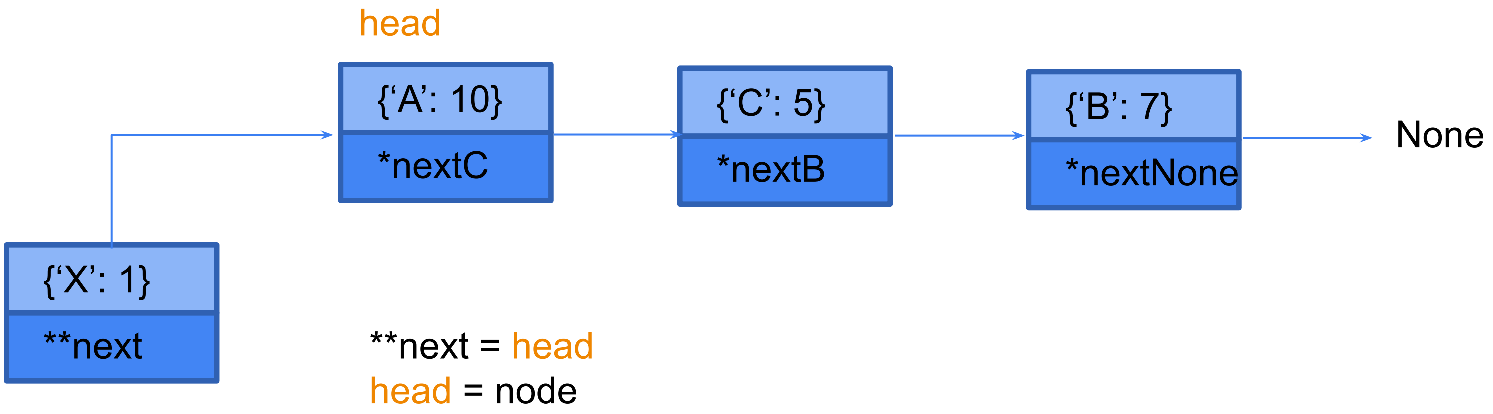


Data - данные узла, Next - указатель на следующий узел в списке

Head - голова списка (изначально она равна None)

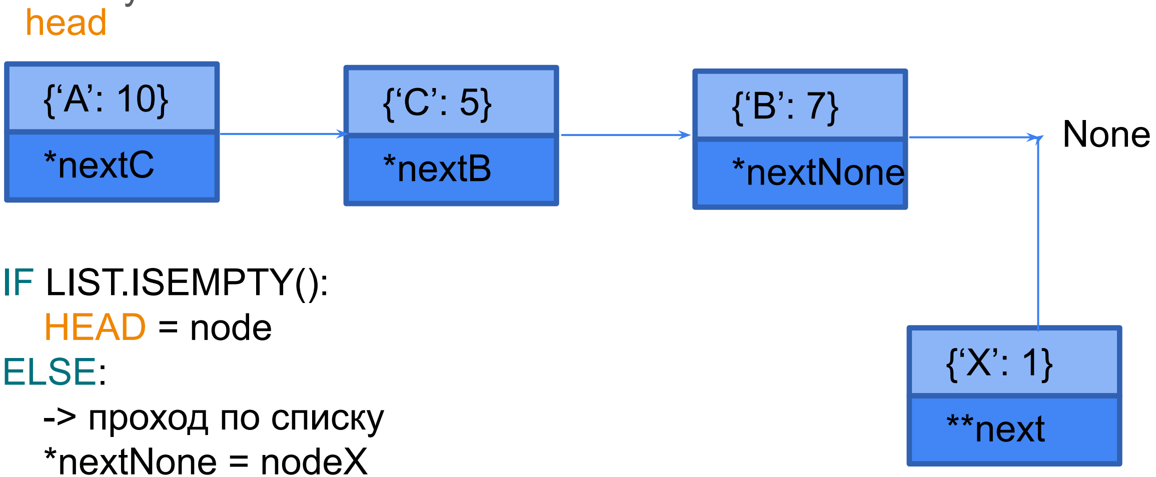
Вставка узла в начало (Add Node at Beginning)

- Создать новый узел с заданными данными.
- Присвоить указателю нового узла ссылку на текущий головной узел.
- Обновить указатель головного (начального) узла, чтобы он указывал на новый узел.



Вставка узла в конец (Add Node at End)

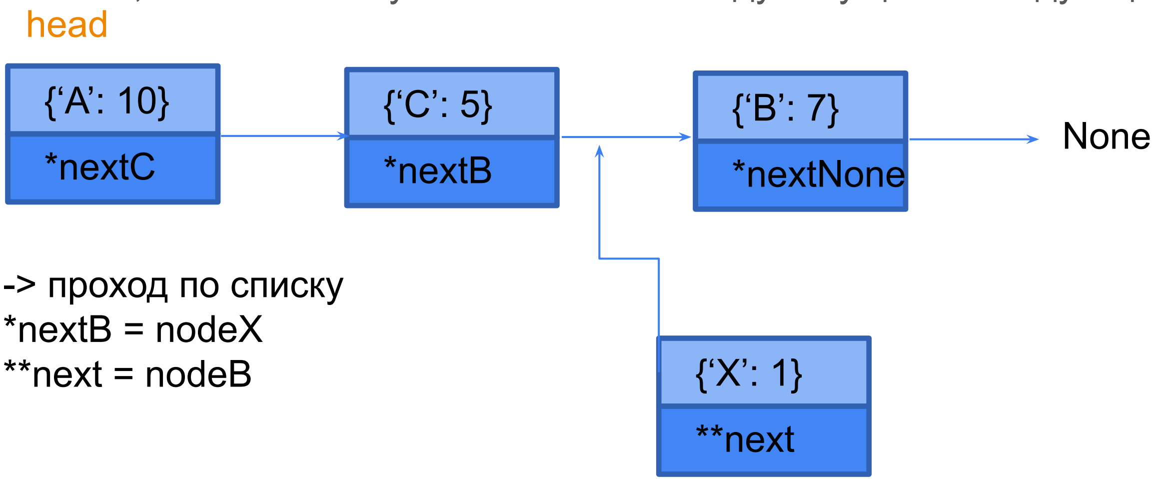
- Создать новый узел с заданными данными.
- Если список пуст (нет начального узла), сделать новый узел начальным.
- Иначе, пройти по списку до последнего узла и обновить ссылку последнего узла, чтобы указывать на новый узел.




```
IF LIST.ISEMPTY():
    HEAD = node
ELSE:
    -> проход по списку
    *nextNone = nodeX
    **next = None
```

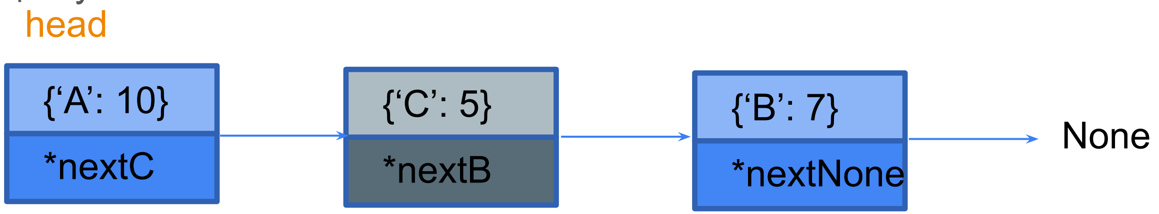
Вставка узла в середину (Add Node in the Middle)

- Начиная с начального узла, пройти по списку до узла, после которого нужно вставить новый узел.
- Создать новый узел с заданными данными.
- Обновить ссылки так, чтобы новый узел вставился между текущим и следующим узлами.



Удаление узла по значению (Delete Node by Value)

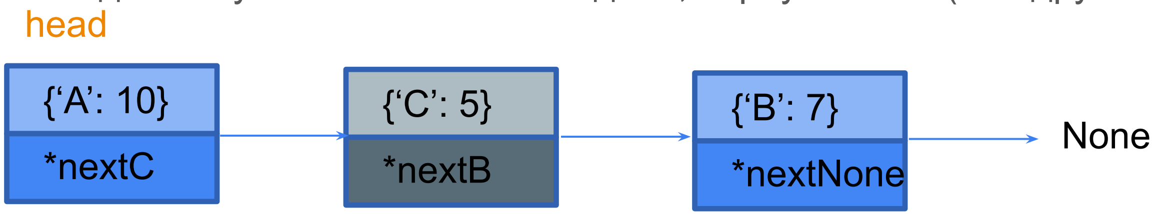
- Начиная с начального узла, пройти по списку, проверяя значение каждого узла.
- Если найдено значение для удаления, изменить ссылку предыдущего узла, чтобы она пропускала текущий узел.
- Удалить текущий узел.



-> проход по списку
IF VALUE == 5
    *nextC = nodeB

Поиск узла по значению (Search Node by Value)

- Начиная с начального узла, пройти по списку, сравнивая значение каждого узла с целевым значением.
- Если найдено значение, вернуть узел.
- Если конец списка достигнут и значение не найдено, вернуть None (или другой индикатор).



-> проход по списку
IF VALUE == 5
PRINT(VALUE)

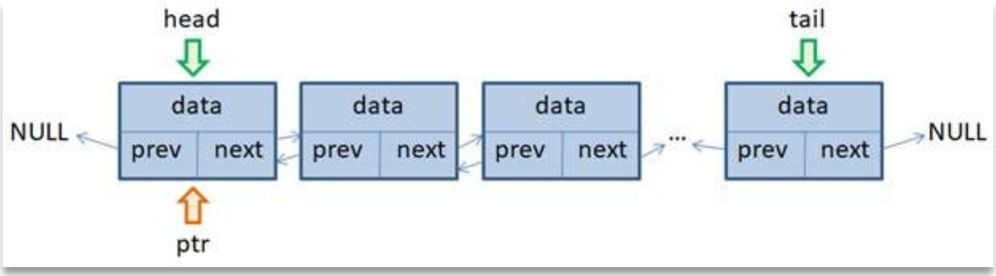
Программная реализация <односвязный список>

```python
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next_node = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def add_node(self, data):
        new_node = Node(data)
        new_node.next_node = self.head
        self.head = new_node

    def display_list(self):
        current_node = self.head
        while current_node:
            print(current_node.data, end=" -> ")
            current_node = current_node.next_node
        print("None")

singly_linked_list = SinglyLinkedList()
singly_linked_list.add_node(3)
singly_linked_list.add_node(7)
singly_linked_list.add_node(1)
singly_linked_list.display_list()
```

```
1 -> 7 -> 3 -> None
```

Двусвязный список.

Структура двусвязного списка.



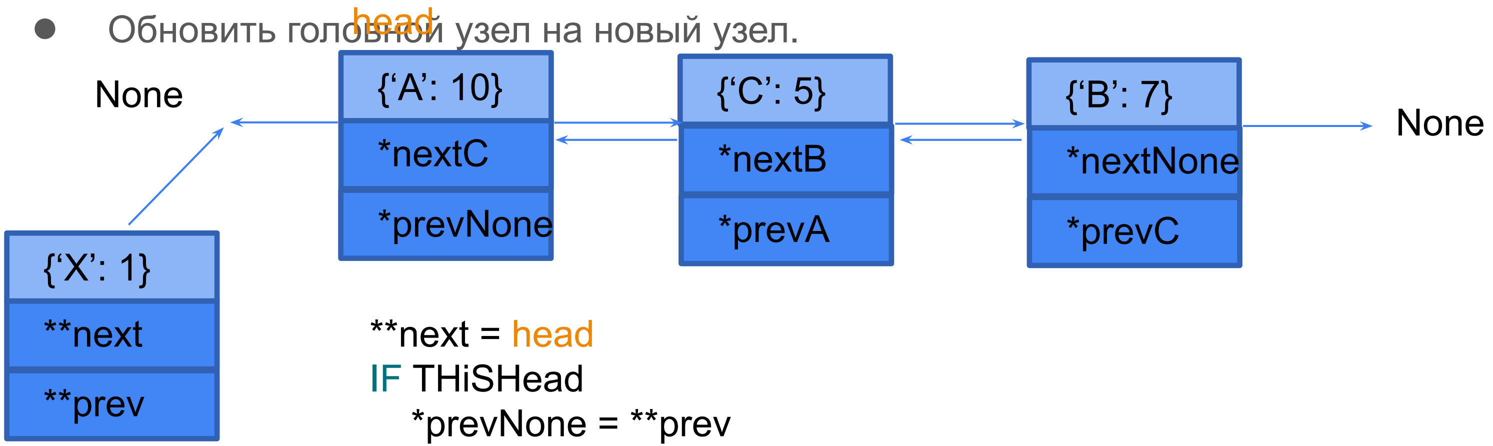
Data - данные узла, Next - указатель на следующий узел в списке, Prev - указатель на предыдущий узел в списке

Head = None; голова списка

**NULL = None (просто это стандартная структура для всех языков)*

Вставка узла в начало (Add Node at Beginning)

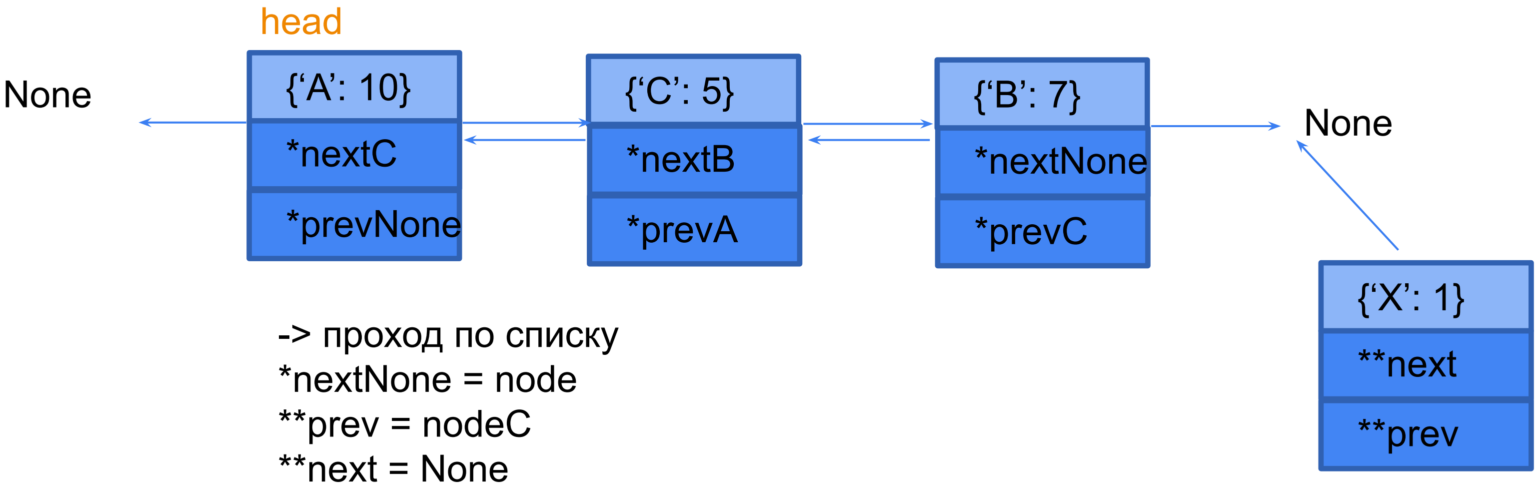
- Создать новый узел с заданными данными.
- Установить следующий узел нового узла на текущий головной узел.
- Если текущий головной узел не None, установить предыдущий узел головного узла на новый узел.
- Обновить головной узел на новый узел.




```
**next = head
IF THiSHead
    *prevNone = **prev
head = node
```

Вставка узла в конец

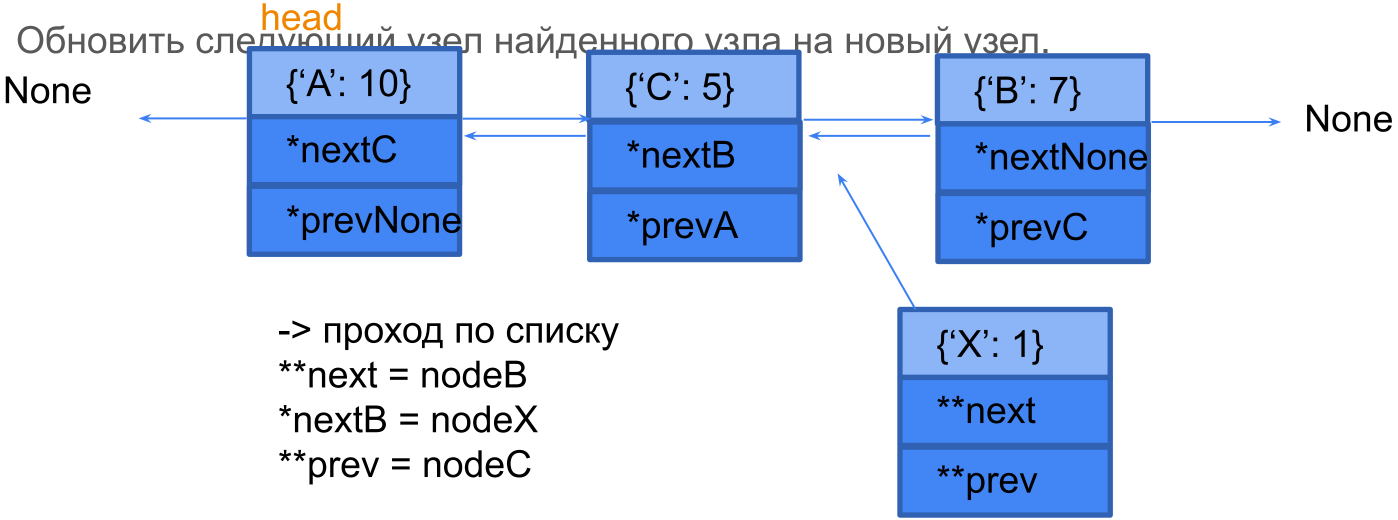
- Создать новый узел с заданными данными.
- Если список пуст (головной узел None), установить головной узел на новый узел.
- Иначе, пройти по списку до последнего узла и обновить следующий узел последнего узла на новый узел, а предыдущий узел нового узла на последний узел.



-> проход по списку
*nextNone = node
**prev = nodeC
**next = None

Вставка узла в середину

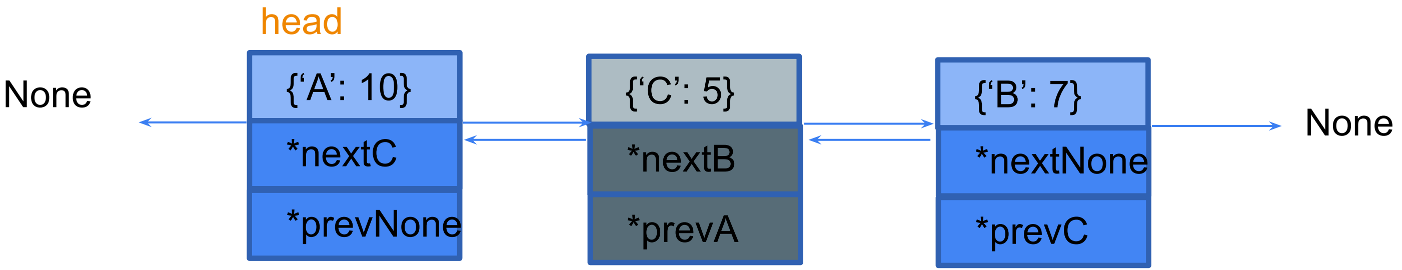
- Создать новый узел с заданными данными.
- Найти узел, после которого нужно вставить новый узел.
- Обновить следующий узел нового узла на узел, следующий за найденным узлом.
- Обновить предыдущий узел нового узла на найденный узел.
- Обновить следующий узел найденного узла на новый узел.



-> проход по списку
**next = nodeB
*nextB = nodeX
**prev = nodeC

Удаление узла

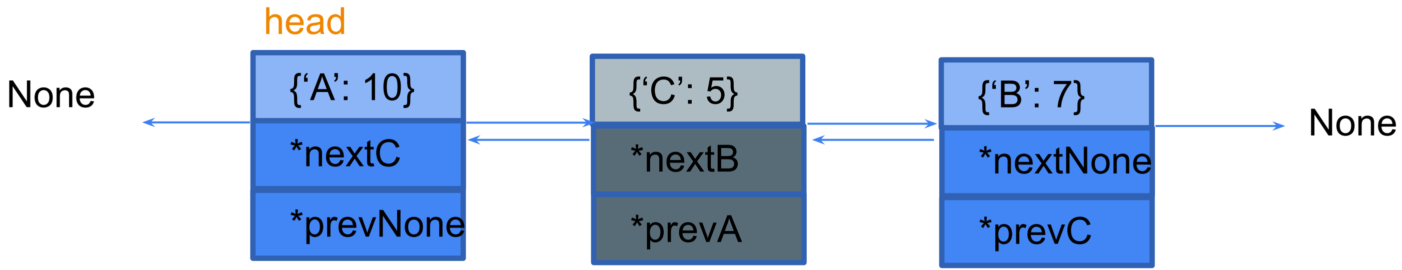
- Найти узел с заданным значением.
- Обновить связи предыдущего и следующего узлов, чтобы "обойти" удаляемый узел.
- Освободить память удаляемого узла.



-> проход по списку
если VALUE = 5;
    *nextC = nodeB
    *prevC = nodeA

Поиск узла по значению

- Пройти по списку и сравнивать значения узлов с искомым значением.
- Вернуть значение



-> проход по списку
если VALUE = 5;
PRINT(VALUE)

Программная реализация <двусвязный список>

```python
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next_node = None
        self.prev_node = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def add_node(self, data):
        new_node = Node(data)
        new_node.next_node = self.head
        if self.head:
            self.head.prev_node = new_node
        self.head = new_node

    def display_list(self):
        current_node = self.head
        while current_node:
            print(current_node.data, end=" <-> ")
            current_node = current_node.next_node
        print("None")

doubly_linked_list = DoublyLinkedList()
doubly_linked_list.add_node(3)
doubly_linked_list.add_node(7)
doubly_linked_list.add_node(1)
doubly_linked_list.display_list()
```

```
1 <-> 7 <-> 3 <-> None
```

ХЕШ-таблица

ХЕШ-таблица.

Хеш-таблица - это структура данных, которая реализует ассоциативный массив или словарь, где данные связаны с уникальными ключами (keys).

Основная идея заключается в использовании хеш-функции для преобразования ключа в индекс массива <обычного списка>, где будет храниться соответствующее значение. Это позволяет быстро выполнять операции вставки, поиска и удаления.

В Python DICT - это например встроенная хеш-таблица.

КЛЮЧИ. ХЕШ-функция.

Ключ - это уникальный идентификатор, который помогает найти значение в хеш-таблице. Например, если у нас есть хеш-таблица, представляющая телефонную книгу, то имена людей могут быть ключами, а их номера телефонов - значениями.

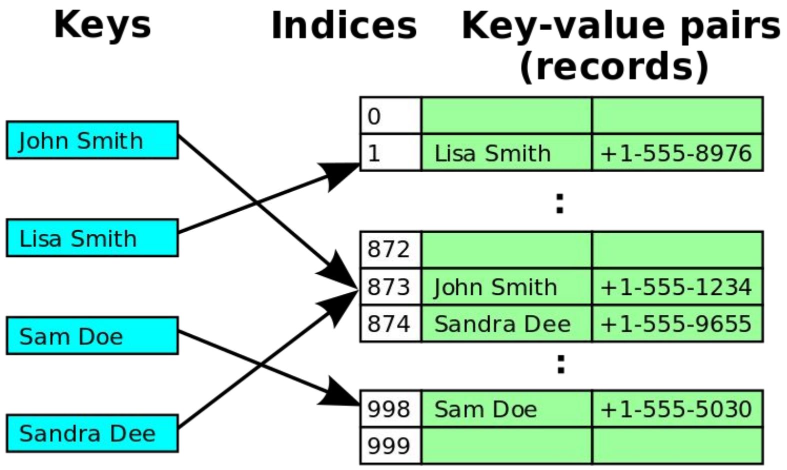
Хеш-функция - это функция, которая преобразует ключ в числовое значение (хеш-код). Хеш-код используется для быстрого поиска соответствующего значения. Хорошая хеш-функция должна быть быстрой и равномерно распределять хеш-коды для разных ключей.

Цель хеш-функции — равномерно распределить входные данные по всему диапазону хеш-кодов.

Запись в ХЕШ-таблицу.

- Вычислить хеш-код ключа.
- Перейти к соответствующей ячейке списка[].
- Вставить значение или обработать коллизию.

```
ind <-- hash(value)
data[ind] <-- value

ind <-- hash("John Smith") = 873
data[873] <-- { Name: "John Smith", Tel: "+1-555-1234" }
```



Коллизии.

Теперь о коллизиях. Коллизия происходит, когда два разных ключа дают одинаковый хеш-код. Например, если два человека имеют одинаковое имя в телефонной книге, у них будет одинаковый ключ, но разные номера телефонов. *Решение коллизий:*

Метод цепочек (Chaining) - метод заключается в том, чтобы каждую "ячейку" хеш-таблицы сделать как бы мини-списком (или другой структурой данных). Если есть коллизия, то новое значение просто добавляется в соответствующий мини-список. Поиск элемента осуществляется сначала по хеш-коду ключа, а затем внутри мини-списка.

Открытая адресация (Open Addressing) - здесь, если произошла коллизия, то новый элемент ищет другое место внутри самой хеш-таблицы на основе какого-либо правила (например, простое линейное пробирование - поиск следующей доступной ячейки). Этот процесс продолжается до тех пор, пока не будет найдено свободное место.

ХЕШ-таблица. Пример.

```
class HashTable:
    def __init__(self, size):
        self.size = size  # Размер хеш-таблицы
        self.table = [None] * size  # Создаем пустую таблицу заданного размера

    def _hash(self, key):
        """
        Приватный метод для вычисления хеш-кода ключа.
        """
        return hash(key) % self.size

    def insert(self, key, value):
        """
        Вставка элемента по ключу.
        """
        index = self._hash(key)
        if self.table[index] is None:
            self.table[index] = [(key, value)]
        else:
            # Если в данной ячейке уже есть элементы,
            # добавляем новый элемент в конец списка
            self.table[index].append((key, value))

hash_table = HashTable(10)  # Создаем хеш-таблицу размером 10
hash_table.insert("apple", 5)
hash_table.insert("banana", 2)
hash_table.insert("cherry", 8)
#print(hash_table.get("apple"))  # 5
#print(hash_table.get("cherry"))  # 8
#hash_table.remove("banana")
#print(hash_table.get("banana"))  # None, так как элемент удален
```

Динамический массив <список>

Динамический массив <список>

Динамический массив <список> - это по факту контейнер или коробка, в которой ты можешь хранить элементы (например, числа или строки), и этот-же контейнер умеет увеличиваться, когда ты добавляешь в него больше элементов. Две важные части динамического массива:

Capacity (емкость) - это, сколько элементов может вместить динамический массив, прежде чем ему нужно будет увеличить свой размер. Допустим, у нас есть коробка (динамический массив) с емкостью 6. Это значит, что мы можем в нее положить 6 элементов, но если мы попытаемся положить 7-й элемент, коробка автоматически увеличит свою емкость (на k-элементов), чтобы вместить его.

Size (размер) - это, сколько элементов действительно хранится в динамическом массиве. Например, если у нас есть коробка с емкостью 6 и внутри только 5 вещей, то размер равен 5.

Динамический массив <список>

Важно помнить, что динамический массив увеличивает свою емкость автоматически, когда ему не хватает места для новых элементов. Это удобно, так как мы можем добавлять элементы в массив, не беспокоясь о том, исчерпаем ли мы его емкость.

В Python, список (list) явл-ся примером динамического массива. Ты можешь добавлять элементы в список, и его емкость будет увеличиваться по мере необходимости. Это делает работу с данными более гибкой и удобной.

Реализация собственного динамич. списка

```python
class DynamicArray:
    def __init__(self):
        self.capacity = 2  # начальная емкость массива
        self.size = 0  # начальный размер массива
        self.array = [None] * self.capacity  # инициализация массива

    def resize(self, new_capacity):
        new_array = [None] * new_capacity
        for i in range(self.size):
            new_array[i] = self.array[i]
        self.array = new_array
        self.capacity = new_capacity

    def append(self, element):
        if self.size == self.capacity:
            # увеличиваем емкость, если массив заполнен
            self.resize(2 * self.capacity)

        self.array[self.size] = element
        self.size += 1

    def get(self, index):
        if 0 <= index < self.size:
            return self.array[index]
        else:
            raise IndexError("Индекс выходит за пределы размера массива")
```

```python
my_array = DynamicArray()
my_array.append(1)
my_array.append(2)
my_array.append(3)

print(my_array.get(0))  # Вывод: 1
print(my_array.get(1))  # Вывод: 2
print(my_array.get(2))  # Вывод: 3
```

Деревья.

Деревья. Виды деревьев.

Дерево - это структура данных, состоящая из узлов, связанных ребрами. Узлы в дереве имеют иерархическую структуру, где один из узлов выступает в качестве корня, а остальные узлы делятся на уровни и имеют родителей и потомков.

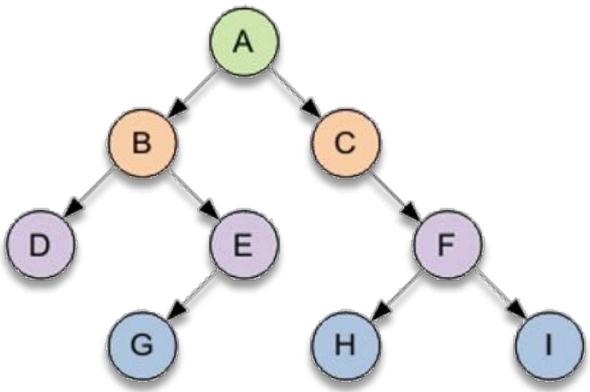
Виды деревьев:

1. **Двоичное дерево** - каждый узел имеет не более двух потомков - левого и правого. Важные типы двоичных деревьев включают бинарные деревья поиска и бинарные кучи.
2. **Двоичное дерево поиска (BST)** - узлы в таком дереве устроены так, что значение в левом поддереве меньше значения узла, а значение в правом поддереве больше. Это обеспечивает эффективный поиск, вставку и удаление элементов.
3. **AVL-дерево** - спец.тип двоичного дерева поиска, в котором высота каждого поддерева ограничена, чтобы гарантировать сбалансированность, что улучшает производительность операций.
4. Красно-черное дерево, N-арное дерево, Дерево отрезков, и др...

Двоичное дерево.

Бинарное дерево - это структура данных, которая состоит из узлов (вершин) и рёбер, связывающих эти узлы. В бинарном дереве каждый узел может иметь не более двух дочерних узлов: левого и правого. Это означает, что каждый узел может быть связан с максимум двумя другими узлами.

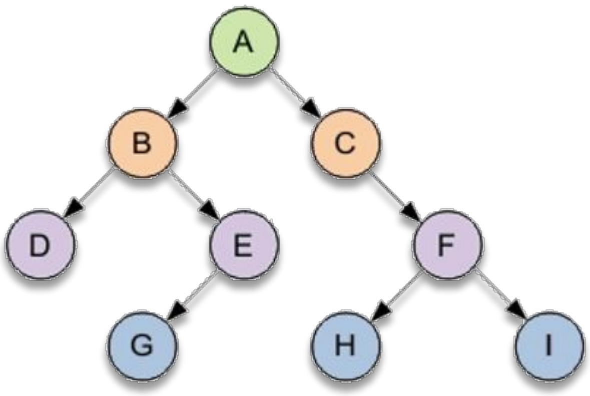
Основные компоненты бинарного дерева:
Корень (root) - это вершина дерева, из которой начинаются все пути вниз по дереву. У корня нет родителей.
Узлы (nodes) - это элементы дерева, которые содержат данные и имеют ноль, один или два дочерних узла.
Листья (leaves) - это узлы, которые не имеют дочерних узлов, то есть они находятся в самом нижнем уровне дерева.
Родители и дети - узел, из которого выходит ребро, называется "родителем", и узлы, соединенные этим ребром, называются "детьми".
Уровень (level) - уровень узла определяется количеством рёбер между корнем и этим узлом. Корень находится на уровне 0, его дети на уровне 1, и так далее.
Глубина (depth) - глубина узла - это количество рёбер от корня до этого узла.
Поддерево (subtree) - часть дерева, включая узел и всех его потомков, называется поддеревом.



*в данном примере **A** - root корень дерева. **B** и **C** поддеревья корня **A**.*

Двоичное дерево. Алгоритм вставки.

Начинаем с корня дерева.
Сравниваем значение, которое мы хотим вставить, с текущим узлом.
Если значение меньше текущего узла, переходим влево; если больше - вправо.
Если достигнут конечный узел (лист) или пустой узел, вставляем новое значение.
Рекурсия: Если узел занят, повторяем шаги 2-4 для поддерева.



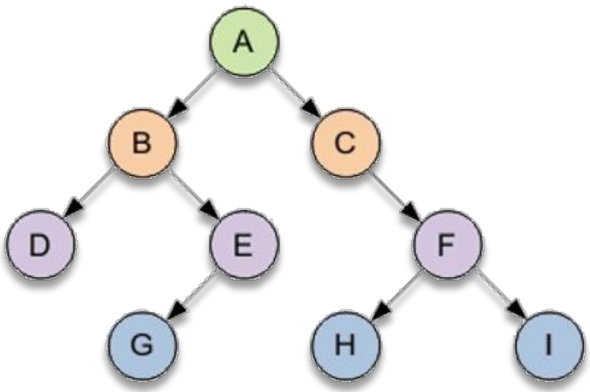

```python
class TreeNode:
    """класс узла дерева"""
    def __init__(self, key):
        self.left = None # левый потомок
        self.right = None # правый потомок
        self.val = key # значение в узле
```

```python
def insert(self, key):
    if self.root is None:
        self.root = TreeNode(key)
    else:
        self._insert_recursively(self.root, key)

def _insert_recursively(self, current_node, key):
    if key < current_node.val:
        if current_node.left is None:
            current_node.left = TreeNode(key)
        else:
            self._insert_recursively(current_node.left, key)
    else:
        if current_node.right is None:
            current_node.right = TreeNode(key)
        else:
            self._insert_recursively(current_node.right, key)
```

Двоичное дерево. Алгоритм удаления.

Находим узел, который содержит удаляемый элемент.
Удаление листьев: Если у узла нет потомков (лист), удаляем его.
Удаление узла с одним потомком: Если у узла есть только один потомок, заменяем узел его потомком.
Удаление узла с двумя потомками: Находим наименьший узел в правом поддереве или наибольший узел в левом поддереве, заменяем удаляемый узел этим узлом, а затем удаляем наименьший или наибольший узел.

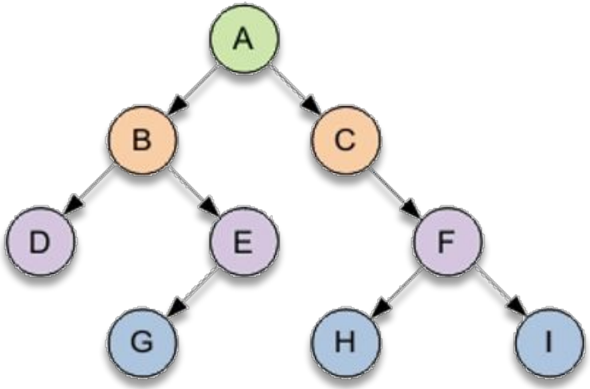
```
def deleteNode(root, key):
    if root is None:
        return root
    ...
```



Двоичное дерево. Алгоритм поиска.

Начинаем с корня дерева.
Сравниваем значение, которое мы ищем, с текущим узлом.
Если значение меньше текущего узла, переходим влево; если больше - вправо.
Если мы достигли узла с нужным значением, нашли. Если конечный узел достигнут и значение не найдено, элемент отсутствует.

```python
def search(self, key):
    return self._search_recursively(self.root, key)

def _search_recursively(self, current_node, key):
    if current_node is None or current_node.val == key:
        return current_node
    if key < current_node.val:
        return self._search_recursively(current_node.left, key)
    return self._search_recursively(current_node.right, key)
```



Двоичное дерево. Пример.

```python
class TreeNode:
    """класс узла дерева"""
    def __init__(self, key):
        self.left = None # левый потомок
        self.right = None # правый потомок
        self.val = key # значение в узле

class BinaryTree:
    """класс бинарное дерево"""
    def __init__(self):
        self.root = None # корень дерева

    def insert(self, key):
        if self.root is None:
            self.root = TreeNode(key)
        else:
            self._insert_recursively(self.root, key)

    def _insert_recursively(self, current_node, key):
        if key < current_node.val:
            if current_node.left is None:
                current_node.left = TreeNode(key)
            else:
                self._insert_recursively(current_node.left, key)
        else:
            if current_node.right is None:
                current_node.right = TreeNode(key)
            else:
                self._insert_recursively(current_node.right, key)

    def search(self, key):
        return self._search_recursively(self.root, key)

    def _search_recursively(self, current_node, key):
        if current_node is None or current_node.val == key:
            return current_node
        if key < current_node.val:
            return self._search_recursively(current_node.left, key)
        return self._search_recursively(current_node.right, key)
```

```python
# Пример использования
if __name__ == "__main__":
    tree = BinaryTree()
    tree.insert(5)
    tree.insert(3)
    tree.insert(8)
    tree.insert(1)
    tree.insert(4)

    # Поиск элемента в дереве
    result = tree.search(4)
    if result:
        print(result)
```

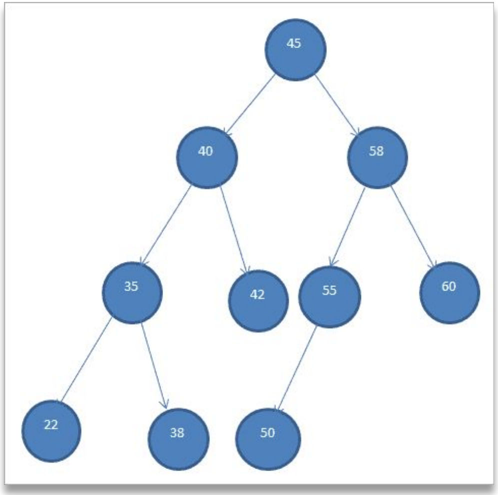
Бинарные деревья могут исп-ся для решения различных задач, таких поиск, сортировка и многие другие. Они также служат основой для дру видов деревьев, таких как двоичные поисковые деревья (Binary Search BST) и деревья куч (Binary Heaps), которые имеют свои собственные особенности и применения.

Двоичное дерево поиска (BST).

Бинарное дерево и бинарное дерево поиска (BST) - это две разные структуры данных, хотя обе они представляют собой деревья, состоящие из узлов, связанных между собой.

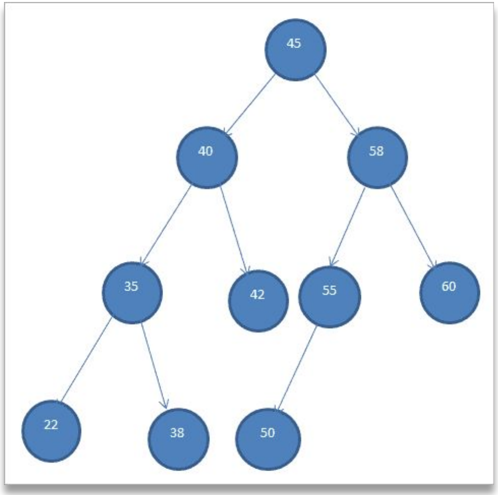
Бинарное дерево поиска также является бинарным деревом, но оно предназначено для хранения данных так, чтобы операции поиска, вставки и удаления были эффективными. Главной целью BST является упорядоченное хранение данных с возможностью быстрого поиска. В BST данные упорядочены таким образом, что для каждого узла все значения в левом поддереве меньше или равны значению узла, а все значения в правом поддереве больше. Это упорядочение обеспечивает эффективные операции поиска, вставки и удаления.

BST широко исп-ся для реализации структур данных, таких как словари, множества и ассоциативные массивы. Они позволяют эффективно выполнять операции поиска, вставки и удаления, и их упорядоченность данных имеет практическое применение.



BST. Алгоритм вставки.

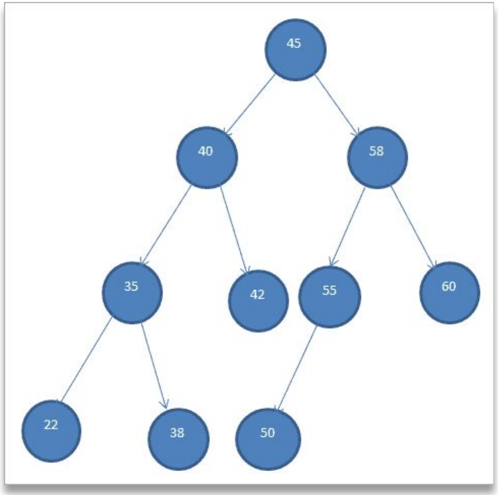
Начинаем с корня дерева.
Если значение, которое мы вставляем, меньше текущего узла, двигаемся влево.
Если значение больше, чем текущий узел, двигаемся вправо.
Повторяем шаги 2 и 3, пока не найдем подходящее место для вставки.
Создаем новый узел и вставляем его в найденное место.




```python
def insert(self, key):
    self.root = self._insert_recursively(self.root, key)

def _insert_recursively(self, root, key):
    if root is None:
        return TreeNode(key)
    if key < root.val:
        root.left = self._insert_recursively(root.left, key)
    else:
        root.right = self._insert_recursively(root.right, key)
    return root
```

BST. Алгоритм удаления.

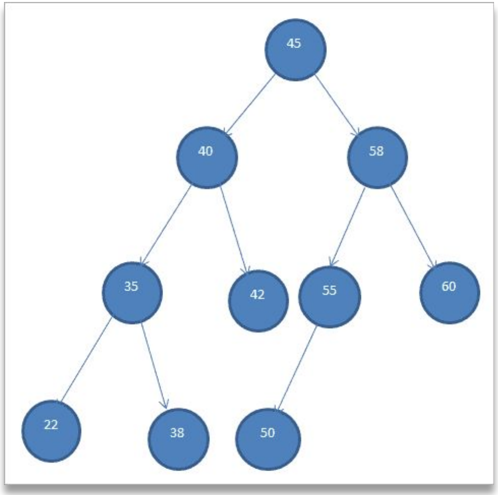
Находим узел, который нужно удалить.
Если у удаляемого узла нет потомков, удаляем его просто.
Если у узла есть один потомок, заменяем узел на своего потомка.
Если у узла два потомка, находим его преемника (например, минимальный узел в правом поддереве), копируем значение преемника в текущий узел, затем рекурсивно удаляем преемника.



BST. Алгоритм поиска.

Начинаем с корня дерева.
Сравниваем искомое значение с текущим узлом.
Если значение меньше текущего узла, двигаемся влево.
Если значение больше, чем текущий узел, двигаемся вправо.
Повторяем шаги 2-4, пока не найдем искомое значение или не дойдем до конца дерева.

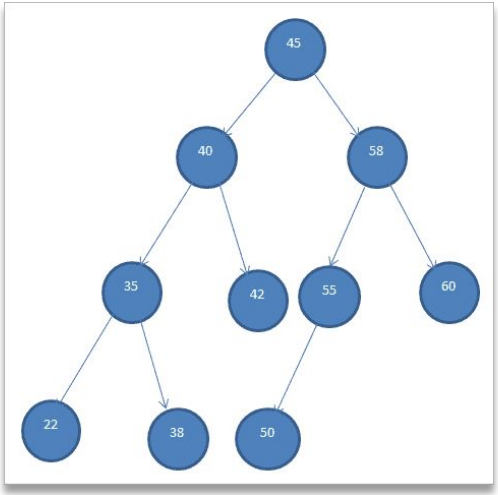
```python
def search(self, key):
    return self._search_recursively(self.root, key)

def _search_recursively(self, root, key):
    pass
```



BST. Алгоритм вывода дерева.

Начинаем с корня дерева.
Рекурсивно выводим левое поддерево.
Выводим значение текущего узла.
Рекурсивно выводим правое поддерево.

```
def print_tree(self):
    self._print_tree(self.root)

def _print_tree(self, ...):
    pass
```



BST. Пример.

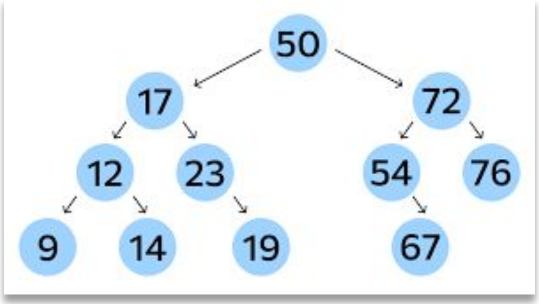
```python
class TreeNode:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self._insert_recursively(self.root, key)

    def _insert_recursively(self, root, key):
        if root is None:
            return TreeNode(key)
        if key < root.val:
            root.left = self._insert_recursively(root.left, key)
        else:
            root.right = self._insert_recursively(root.right, key)
        return root

    def search(self, key):
        return self._search_recursively(self.root, key)

    def _search_recursively(self, root, key):
        pass
```

```python
# остальные методы класса

if __name__ == "__main__":
    bst = BinarySearchTree()
    bst.insert(50)
    bst.insert(30)
    bst.insert(70)
    bst.insert(20)
    bst.insert(40)
    bst.insert(60)
    bst.insert(80)

    # Поиск элемента в дереве
    result = bst.search(60)
    if result:
        print("Элемент найден:", result.val)
    else:
        print("Элемент не найден")

    # Удаление элемента из дерева
    bst.delete(30)
    print("После удаления 30:")
    result = bst.search(30)
    if result:
        print("Элемент найден:", result.val)
    else:
        print("Элемент не найден")
```

AVL-дерево.

AVL-дерево (Adelson-Velsky и Landis) - это сбалансированное двоичное дерево поиска, где разница в высоте между левым и правым поддеревьями каждого узла ограничена значением, известным как баланс-фактор. Обычно баланс-фактор ограничен значениями -1, 0 и 1.

Сбалансированное дерево - это структура данных, представляющая собой дерево, в котором высота поддеревьев каждого узла различается не более чем на единицу. Такие деревья обеспечивают более эффективные операции вставки, удаления и поиска по сравнению с несбалансированными структурами данных.



AVL. Алгоритм вставки.

Вставка начинается как в обычном BST.
После вставки узла проверяется баланс-фактор.
Если баланс-фактор становится больше 1 или меньше -1, производится балансировка.
Возможны четыре случая небаланса: лево-лево, право-право, лево-право и право-лево.
Для каждого случая выполняются соответствующие вращения.

AVL. Алгоритм удаления.

Удаление начинается как в обычном BST.
После удаления узла проверяется баланс-фактор.
Если баланс-фактор становится больше 1 или меньше -1, производится балансировка.
Возможны те же четыре случая, что и при вставке, и также выполняются соответствующие вращения.

AVL. Алгоритм поиска.

Поиск выполняется так же, как и в обычном BST.
Благодаря сбалансированности AVL-дерева время поиска остается O(log n) в среднем.

```python
def _search(self, node, key):
    if node is None or node.key == key:
        return node
    if key < node.key:
        return self._search(node.left, key)
    return self._search(node.right, key)

def search(self, key):
    result = self._search(self.root, key)
    if result:
        return result.value
    else:
        return None
```

AVL. Пример.

```python
class AVLNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.height = 1
        self.left = None
        self.right = None


class AVLTree:
    def __init__(self):
        self.root = None

    def _height(self, node):
        if node is None:
            return 0
        return node.height

    def _balance(self, node):
        if node is None:
            return 0
        return self._height(node.left) - self._height(node.right)

    def _update_height(self, node):
        if node is not None:
            node.height = 1 + max(self._height(node.left), self._height(node.right))

    def _rotate_left(self, y):
        x = y.right
        T2 = x.left

        x.left = y
        y.right = T2

        self._update_height(y)
        self._update_height(x)

        return x
```

```python
def _rotate_right(self, x):
    y = x.left
    T2 = y.right
    y.right = x
    x.left = T2
    self._update_height(x)
    self._update_height(y)

    return y

def _insert(self, node, key, value):
    if node is None:
        return AVLNode(key, value)
    if key < node.key:
        node.left = self._insert(node.left, key, value)
    elif key > node.key:
        node.right = self._insert(node.right, key, value)
    else:
        # Duplicate keys are not allowed in this example
        return node
    self._update_height(node)
    balance = self._balance(node)
    # Left Heavy
    if balance > 1:
        if key < node.left.key:
            return self._rotate_right(node)
        else:
            node.left = self._rotate_left(node.left)
            return self._rotate_right(node)
    # Right Heavy
    if balance < -1:
        if key > node.right.key:
            return self._rotate_left(node)
        else:
            node.right = self._rotate_right(node.right)
            return self._rotate_left(node)
    return node

def insert(self, key, value):
    self.root = self._insert(self.root, key, value)
```

Сравнение деревьев BT/BST/AVL.

Двоичное дерево поиска (BST):

- Простая структура, где каждый узел имеет максимум два потомка: левого и правого.
- Не гарантирует сбалансированность, что может привести к неэффективному времени выполнения операций при неудачном выборе узлов.

AVL-дерево:

- Гарантирует сбалансированность, поддерживая ограничение на разницу высот поддеревьев.
- Операции вставки, удаления и поиска выполняются в среднем за O(log n) времени, где n - количество узлов.
- Обеспечивает стабильное время выполнения операций даже в худшем случае.

AVL-деревья предоставляют эффективные операции вставки, удаления и поиска за счет поддержания сбалансированной структуры. Однако, за счет дополнительных проверок и балансировок, они могут быть слегка менее эффективными в сравнении с обычными BST в случаях, где сбалансированность не является критичной.

Строкового представления деревьев

Строчное представление деревьев обычно используется для лаконичного представления структуры дерева в виде строки, где каждый элемент представляет узел дерева.

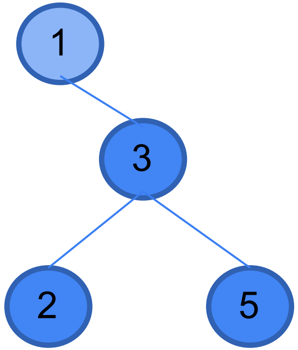
Рассмотрим строки представления для двоичного дерева, двоичного дерева поиска (BST) и AVL-дерева.

Двоичное дерево <string>

Простое двоичное дерево не обязательно сбалансированное.

Строковое представление:

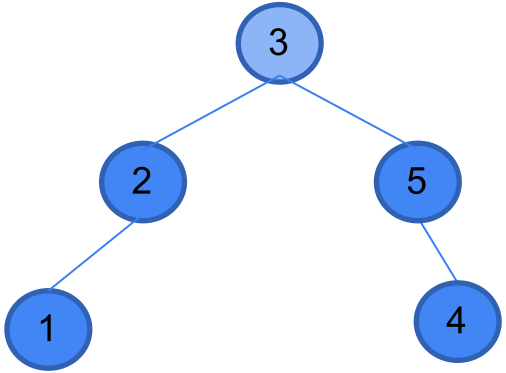
1,3,2,5 где (1) - root



BST <string>

Двоичное дерево, где для каждого узла левый потомок меньше, а правый потомок больше его значения.

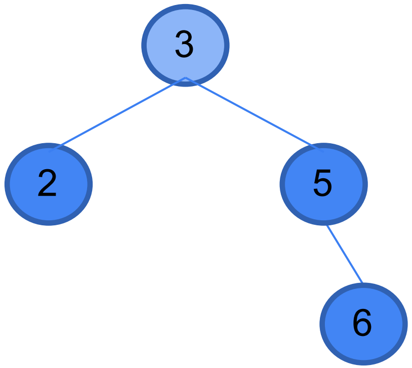
<u>*Строковое представление:*</u>
3,2,5,1,4 где (3) - root



AVL <string>

Строковое представление:

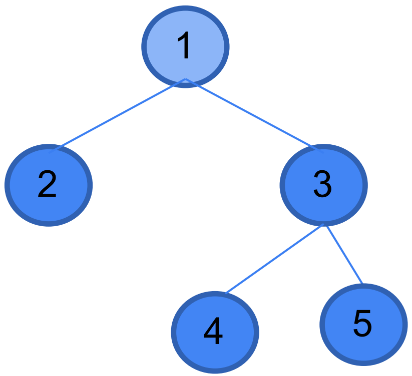
3,2,5,6 где (3) - root



Какое представление будет иметь это дерево? И что это за дерево?

Строковое представление:

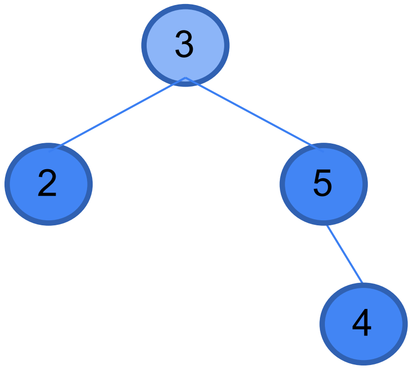
???



Какое представление будет иметь это дерево? И что это за дерево?

Строковое представление:

???



Графы.

Графы. Применение.

Граф – это абстрактная математическая структура, представляющая собой множество вершин (или узлов), соединенных ребрами (или дугами). Вершины могут представлять объекты, а ребра – отношения между этими объектами. Графы исп-ся для моделирования различных сценариев и отношений в различных областях, таких как социальные сети, транспортные системы, компьютерные сети и др.

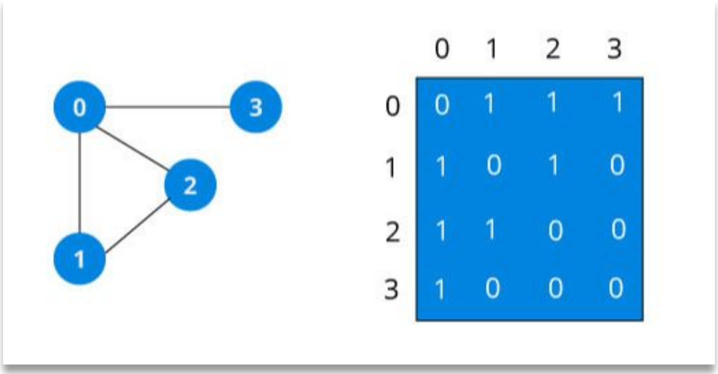
Применение:

- **Моделирование отношений:** Графы позволяют наглядно представить и анализировать связи и взаимодействия между объектами.
- **Решение задач навигации:** Графы используются для поиска кратчайших путей в транспортных системах, сетях и других областях.
- **Сетевое проектирование:** Графы широко применяются в проектировании и анализе сетей, таких как компьютерные сети, телекоммуникационные сети и др.
- **Алгоритмы машинного обучения:** Графы используются для представления и анализа данных, таких как графы схожести или графы знаний.

Графы. Списки смежности.

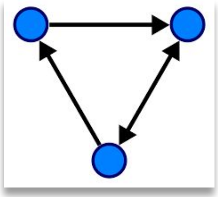
Список смежности - это способ представления графа в виде структуры данных, где для каждой вершины указываются все вершины, с которыми она соединена рёбрами. Это наиболее распространенный способ представления графов в программировании.

При использовании списка смежности для представления графа, для каждой вершины создается список, в который добавляются все вершины, соединенные с данной вершиной. В результате, весь граф представляется в виде словаря, где ключами являются вершины, а значениями — списки смежности.

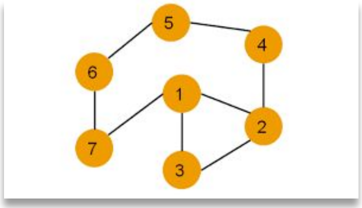


Виды графов.

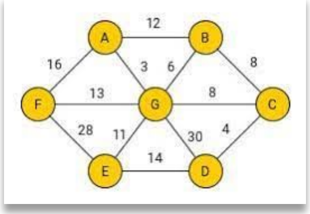
- **Ориентированный граф (орграф) -** граф, у которого ребра имеют направление, т.е., они представляют собой упорядоченные пары вершин.



- **Неориентированный граф -** граф, в котором ребра не имеют направления. Связи между вершинами двусторонние.



- **Взвешенный граф -** граф, в котором каждому ребру присвоено числовое значение (вес), представляющее стоимость или длину этого ребра.



Основные операции над графами.

- **Вставка вершины (узла):** Добавление новой вершины в граф.
- **Удаление вершины (узла):** Удаление вершины из графа и всех связанных с ней ребер.
- **Вставка ребра:** Добавление ребра между двумя вершинами.
- **Удаление ребра:** Удаление ребра между двумя вершинами.
- **Поиск вершины:** Поиск вершины в графе.
- **Вывод графа:** Печать графа в удобной форме для визуализации.

Граф. Алгоритм добавления вершины.

1. Определение вершины:

   Проверяем, существует ли уже вершина с таким именем в графе.

2. Добавление вершины:

- Если вершина не существует, создаем пустой список смежности для этой вершины.
- Добавляем вершину в структуру данных графа (например, в словарь, где ключи - имена вершин, а значения - списки смежности).

Граф. Алгоритм добавления ребра.

1. Определение существования вершин:

Проверяем, существуют ли вершины, соответствующие начальной и конечной вершинам ребра, в графе.

2. Добавление ребра:

- Если обе вершины существуют, добавляем конечную вершину в список смежности начальной вершины.
- В ориентированном графе это может означать добавление конечной вершины в список смежности начальной вершины.
- В неориентированном графе, где рёбра двунаправленные, добавляем каждую вершину в список смежности другой вершины.

Граф. Алгоритм удаления вершины.

1. Проверка существования вершины:

   Проверяем, существует ли вершина, которую мы хотим удалить.

2. Удаление вершины:

- Если вершина существует, удаляем ее из списка вершин графа.
- Затем проходим по всем остальным вершинам и удаляем все рёбра, содержащие удаленную вершину.

```
def remove_vertex(self, vertex):
    if vertex in self.graph:
        # Удаление вершины из списка вершин
        del self.graph[vertex]
        # Удаление вершины из списков смежности других вершин
        for v in self.graph:
            self.graph[v] = [adj for adj in self.graph[v] if adj != vertex]
```

Граф. Алгоритм вывода графа.

1. Вывод вершин:
   Проходим по всем вершинам графа.
   Для каждой вершины выводим ее имя и информацию о смежных вершинах (список смежности).

2.1 Вывод рёбер (неориентированный граф):

- При неориентированных графах рёбра выводятся только один раз для избежания дублирования.
- Проходим по всем вершинам графа.
- Для каждой вершины выводим рёбра, соединяющие ее с другими вершинами.

2.2 Вывод рёбер (ориентированный граф):

- В ориентированных графах рёбра направлены, поэтому выводим каждое ребро от начальной вершины к конечной.
- Проходим по всем вершинам графа.
- Для каждой вершины выводим исходящие рёбра.

Структура Графа пример кода.

```python
class Graph:
    def __init__(self):
        self.graph = {}

    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, start_vertex, end_vertex):
        if start_vertex in self.graph:
            self.graph[start_vertex].append(end_vertex)
        else:
            self.graph[start_vertex] = [end_vertex]

    def print_graph(self):
        for vertex in self.graph:
            neighbors = ', '.join(map(str, self.graph[vertex]))
            print(f"{vertex} -> {neighbors}")

g = Graph()
g.add_vertex(1)
g.add_vertex(2)
g.add_vertex(3)
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 1)
g.print_graph()
```

```
1 -> 2
2 -> 3
3 -> 1
```

Граф. Алгоритмы.

- **Поиск в глубину (Depth-First Search, DFS):** Алгоритм для обхода графа в глубину. Подходит для поиска в связных компонентах и топологической сортировки.
- **Поиск в ширину (Breadth-First Search, BFS):** Алгоритм для обхода графа в ширину. Используется для поиска кратчайших путей в невзвешенных графах.
- **Алгоритм Дейкстры (Dijkstra's Algorithm):** Используется для поиска кратчайших путей в графе с весами на ребрах.
- **Алгоритм Беллмана-Форда (Bellman-Ford Algorithm):** Решает задачу поиска кратчайших путей в графе, даже если есть рёбра с отрицательными весами.
- **Алгоритм Прима (Prim's Algorithm)** (мин-е остовое дерево) **Алгоритм Крускала** (Kruskal's Algorithm): др алгоритм для построения минимального остовного дерева на взвешенном графе.
- **Топологическая сортировка: -**
- **Алгоритм Флойда-Уоршелла (Floyd-Warshall Algorithm):** Находит кратчайшие пути между всеми парами вершин в графе (в том числе с отрицательными весами).

Алгоритм Дейкстры.

```python
def dijkstra(self, start_vertex):
    distances = {vertex: float('infinity') for vertex in self.graph}
    distances[start_vertex] = 0

    priority_queue = [(0, start_vertex)]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in self.graph[current_vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances
```

АИСД. Список задач. #1

- Связанные списки:

  Реализовать удаление дубликатов из односвязного списка.
  Написать функцию для нахождения среднего элемента в односвязном списке.
  Написать функцию нахождения цикла в списке (напр. односвязном) - задача уровня компании &APPLE.
  Реализовать функцию объединения двух односвязных списков в один.

- Хеш-таблицы:

  Реализовать хеш-таблицу с открытым методом адресации (например, методом цепочек).
  Написать функцию проверки наличия цикла в хеш-таблице.

- Двоичные деревья:

  Найти высоту бинарного дерева/BST.
  Найти минимальный/максимальный элемент в дереве
  Найти сумму всех элементов дерева (узлов)
  Найти сред. арифм всех узлов дерева
  Реализовать поиск элемента в бинарном дереве поиска (BST).
  Реализовать удаление элемента из AVL-дерева.

АИСД. Список задач. #2

- Очереди/деки:

  Реализовать очередь с использованием двусвязного списка.
  Написать функцию для обхода дека (двусторонней очереди) в обе стороны.
  Simple class очереди
  Simple class дека
  Поиск/вычисление коэф. загруженности очереди