**Binary tree and its balancing. Treap tree**

***Completed: Mishanya03***

***2023***

Оглавление

[**Класс бинарных деревьев** 3](#_Toc137758666)

[**Метод Add** 3](#_Toc137758667)

[**Методы SearchByKey и SearchPlace** 4](#_Toc137758668)

[**Методы NodeMin и NodeMax** 5](#_Toc137758669)

[**Методы вывода графа** 5](#_Toc137758670)

[**Метод IsBalanced** 7](#_Toc137758671)

[**Чем отличается данный класс бинарного дерева от AVL-дерева?** 8](#_Toc137758672)

[**Метод балансировки данного класса и его отличие от балансировки AVL-дерева** 9](#_Toc137758673)

[**Алгоритм создания сбалансированного списка ключей** 9](#_Toc137758674)

[**Тестирование балансировки** 12](#_Toc137758675)

[**Класс декартового дерева (дерамида)** 12](#_Toc137758676)

[**Заключение** 15](#_Toc137758677)

[**Список литературы** 17](#_Toc137758678)

# **Класс бинарных деревьев**

Класс BinaryTree<T> представляет собой бинарное дерево поиска, где каждый узел содержит ключ типа int и значение типа T, которое должно реализовывать интерфейс IComparable.

Класс имеет следующие публичные поля и методы:

* root - корневой узел дерева.
* Add(int key, T value) - добавляет новый узел в дерево с указанным ключом и значением. Если узел с таким ключом уже есть в дереве, то его значение обновляется.
* SearchByKey(int key) - ищет узел в дереве по заданному ключу и возвращает его. Если узел не найден, метод возвращает null.
* NodeMin() - находит узел с минимальным ключом в дереве и возвращает его. Если дерево пустое, метод возвращает null.
* NodeMax() - находит узел с максимальным ключом в дереве и возвращает его. Если дерево пустое, метод возвращает null.
* View(), View2(), View3(), ViewFromMax(), ViewFromMin() - методы для вывода дерева в консоль в различных форматах.
* DeleteNode(int key) - удаляет узел с указанным ключом из дерева. Если узла с таким ключом нет в дереве, метод ничего не делает.
* BalanceTree() - метод для балансировки дерева.
* IsBalanced() – метод для проверки на сбалансированность дерева

Для хранения узлов дерева используется класс Node<T>, который содержит ключ, значение и ссылки на левое, правое и родительское поддеревья.

В общем случае, бинарное дерево поиска является структурой данных, где каждый узел имеет не более двух потомков (левый и правый), а также выполняется условие: для любого узла все ключи в левом поддереве меньше его ключа, а все ключи в правом поддереве больше его ключа.

## **Метод Add**

public void Add(int key, T value)

Метод **Add** класса бинарных деревьев выполняет добавление нового узла в дерево по заданному ключу и значению.

Вначале метод проверяет, существует ли корень дерева. Если корня нет, то создается новый узел с заданным ключом и значением, и этот узел становится корнем дерева. Затем метод возвращает управление.

Если корень уже существует, то вызывается метод **SearchPlace**, который выполняет поиск места для вставки нового узла в дерево. Результат поиска сохраняется в переменной **currentNode**.

Если найденный узел **currentNode** имеет ключ, равный вставляемому ключу, то значит в дереве уже существует узел с таким ключом. В этом случае значение этого узла обновляется заданным значением **value**.

Если найденный узел **currentNode** не имеет ключа, равного вставляемому ключу, то создается новый узел с заданным ключом и значением. Затем происходит проверка ключа нового узла относительно ключа **currentNode**. Если ключ нового узла больше ключа **currentNode**, то новый узел становится правым потомком **currentNode**, а **currentNode** становится родительским узлом нового узла. Если ключ нового узла меньше или равен ключу **currentNode**, то новый узел становится левым потомком **currentNode**, а **currentNode** становится родительским узлом нового узла.

Таким образом, метод **Add** добавляет новый узел в бинарное дерево поиска в соответствии с его ключом и значением.

## **Методы SearchByKey и SearchPlace**

public Node<T> SearchByKey(int key)

private Node<T> SearchPlace(int key)

Метод SearchByKey(int key):

* + Принимает входной параметр "key" - ключ, по которому выполняется поиск.
  + Инициализирует переменную "currentNode" в корневым узлом дерева.
  + Запускает бесконечный цикл для поиска узла с указанным ключом.
  + В цикле проверяет текущий узел на равенство ключу:
    - Если ключ текущего узла равен заданному ключу, то возвращается текущий узел.
    - Если ключ текущего узла меньше заданного ключа и у текущего узла есть правый потомок, то текущим узлом становится правый потомок.
    - Если ключ текущего узла больше заданного ключа и у текущего узла есть левый потомок, то текущим узлом становится левый потомок.
    - Если ни одно из условий не выполняется, то возвращается null, так как узел с заданным ключом не найден.

Приватный метод SearchPlace(int key):

* + Принимает входной параметр "key" - ключ, для которого ищется место вставки.
  + Инициализирует переменную "currentNode" в корневым узлом дерева.
  + Запускает бесконечный цикл для поиска места вставки узла с указанным ключом.
  + В цикле проверяет текущий узел на равенство ключу:
    - Если ключ текущего узла равен заданному ключу, то возвращается текущий узел.
    - Если ключ текущего узла меньше заданного ключа и у текущего узла есть правый потомок, то текущим узлом становится правый потомок.
    - Если ключ текущего узла больше заданного ключа и у текущего узла есть левый потомок, то текущим узлом становится левый потомок.
    - Если ни одно из условий не выполняется, то возвращается текущий узел, что указывает на место вставки нового узла.

Оба метода осуществляют поиск в бинарном дереве по ключу и возвращают найденный узел или указывают на место вставки нового узла.

## **Методы NodeMin и NodeMax**

public Node<T> NodeMin()

public Node<T> NodeMax()

Метод **NodeMin()** возвращает узел с наименьшим ключом в бинарном дереве. Он начинает поиск с корневого узла и переходит к левому потомку до тех пор, пока не достигнет узла без левого потомка. Этот узел считается узлом с наименьшим ключом и возвращается в результате.

Метод **NodeMax()** возвращает узел с наибольшим ключом в бинарном дереве. Он начинает поиск с корневого узла и переходит к правому потомку до тех пор, пока не достигнет узла без правого потомка. Этот узел считается узлом с наибольшим ключом и возвращается в результате.

Оба метода проверяют, пустое ли дерево (если **root == null**) и возвращают **null**, если дерево пустое.

## **Методы вывода графа**

public void View() => ViewTree(root, 0);

public void View2() => PrintDepths(root, 0);

public void View3() => PrintTree(root, 0);

public void ViewFromMin()

public void ViewFromMax()

private void ViewTree(Node<T> currentNode, int level)

private void PrintDepths(Node<T> currentNode, int level)

private void PrintTree(Node<T> currentNode, int level)

**View()**: Этот метод вызывает метод **ViewTree(root, 0)** для вывода бинарного дерева. Метод **ViewTree** рекурсивно обходит дерево и выводит каждый узел вместе с его уровнем.

**View2()**: Этот метод вызывает метод **PrintDepths(root, 0)** для вывода бинарного дерева. Метод **PrintDepths** рекурсивно обходит дерево и выводит каждый узел вместе с его уровнем. Узлы выводятся в порядке обхода по глубине (depth-first).

**View3()**: Этот метод вызывает метод **PrintTree(root, 0)** для вывода бинарного дерева. Метод **PrintTree** рекурсивно обходит дерево и выводит каждый узел вместе с его уровнем. Узлы выводятся таким образом, чтобы структура дерева была наглядно представлена.

**ViewTree(Node<T> currentNode, int level)**: Это приватный метод, используемый методом **View()**. Он рекурсивно обходит дерево, начиная с указанного узла **currentNode** и выводит каждый узел вместе с его уровнем. Сначала обрабатываются левое поддерево, затем текущий узел и затем правое поддерево.

**PrintDepths(Node<T> currentNode, int level)**: Это приватный метод, используемый методом **View2()**. Он рекурсивно обходит дерево, начиная с указанного узла **currentNode** и выводит каждый узел вместе с его уровнем. Узлы выводятся в порядке обхода по глубине (depth-first). Сначала обрабатывается левое поддерево, затем правое поддерево.

**PrintTree(Node<T> currentNode, int level)**: Это приватный метод, используемый методом **View3()**. Он рекурсивно обходит дерево, начиная с указанного узла **currentNode** и выводит каждый узел вместе с его уровнем. Узлы выводятся таким образом, чтобы структура дерева была наглядно представлена. Сначала обрабатывается правое поддерево, затем текущий узел и затем левое поддерево.

Метод **ViewFromMin()** позволяет просмотреть элементы дерева, начиная с наименьшего элемента. Он начинает с поиска минимального элемента в дереве, вызывая метод **FindMin()** с передачей корневого узла **Root**. Затем происходит цикл, который выполняется до тех пор, пока следующий узел после **currentNode** не станет равным **null**. В каждой итерации цикла текущий узел выводится на консоль с помощью **Console.WriteLine()**, а затем устанавливается на следующий узел, вызывая метод **Next(currentNode)**. После выхода из цикла последний узел также выводится на консоль.

Метод **ViewFromMax()** работает аналогично, но позволяет просмотреть элементы дерева, начиная с наибольшего элемента. Он начинает с поиска максимального элемента в дереве, вызывая метод **FindMax()** с передачей корневого узла **Root**. Затем происходит цикл, который выполняется до тех пор, пока предыдущий узел перед **currentNode** не станет равным **null**. В каждой итерации цикла текущий узел выводится на консоль с помощью **Console.WriteLine()**, а затем устанавливается на предыдущий узел, вызывая метод **Prev(currentNode)**. После выхода из цикла последний узел также выводится на консоль.

**Методы DeleteNode**

public void DeleteNode(int key)

Метод **DeleteNode** класса бинарных деревьев выполняет удаление узла с заданным ключом из дерева. Вот описание его работы:

1. Поиск узла с заданным ключом в дереве с помощью метода **SearchByKey**. Если такой узел не найден, операция завершается.
2. Если удаляемый узел не имеет детей (является листом):
   * Если удаляемый узел является корнем дерева, то присваиваем **root** значение **null**, тем самым удаляя дерево.
   * Если удаляемый узел является листом дерева, то отсоединяем его от родительского узла путем установки соответствующего указателя (**left** или **right**) у родительского узла в значение **null**.
3. Если удаляемый узел имеет только одного ребенка:
   * Определяем единственного ребенка удаляемого узла.
   * Если удаляемый узел является корнем дерева, то делаем его ребенка новым корнем дерева.
   * Если удаляемый узел не является корнем, то заменяем указатель на удаляемый узел у его родителя указателем на его ребенка и обновляем указатель родителя у ребенка на родителя удаляемого узла.
4. Если удаляемый узел имеет двух детей, вызываем вспомогательный метод **DeleteTwoChild**, который выполняет удаление узла с двумя детьми.

Метод **DeleteTwoChild** выполняет удаление узла по заданному ключу м балансировкой его детей.

## **Метод IsBalanced**

public bool IsBalanced()

Метод **IsBalanced()** вызывает приватный метод **CheckBalanceNode()** для проверки сбалансированности корневого узла дерева. Если метод **CheckBalanceNode()** возвращает **true**, это означает, что дерево сбалансировано, и метод **IsBalanced()** также возвращает **true**. Если **CheckBalanceNode()** возвращает **false**, это означает, что дерево не сбалансировано, и метод **IsBalanced()** также возвращает **false**.

Метод **CheckBalanceNode()** рекурсивно проверяет сбалансированность каждого узла в дереве. Если текущий узел **currentNode** равен **null**, то метод возвращает **true**, так как пустое поддерево считается сбалансированным. Затем метод вычисляет высоты левого поддерева и правого поддерева текущего узла, с помощью метода **GetHeight()**. Вычисляется разница в высотах **heightDiff** между левым и правым поддеревьями. Если **heightDiff** больше 1, то дерево не сбалансировано, и метод возвращает **false**. В противном случае, метод вызывает себя рекурсивно для левого и правого поддерева текущего узла и возвращает логическое И (**&&**) результатов проверки сбалансированности левого и правого поддерева.

Метод **GetHeight()** рекурсивно вычисляет высоту узла **node**. Если узел **node** равен **null**, то метод возвращает 0, так как высота пустого поддерева равна 0. Затем метод вычисляет высоту левого поддерева и правого поддерева узла **node**, вызывая себя рекурсивно для каждого поддерева. Высота узла **node** равна максимальной высоте между левым и правым поддеревьями, увеличенной на 1.

Эти методы позволяют проверить сбалансированность дерева, определяя разницу в высотах между левым и правым поддеревьями каждого узла. Если разница превышает 1, дерево считается несбалансированным. Проверка сбалансированности дерева важна для обеспечения эффективности операций поиска, добавления и удаления узлов в дереве.

# **Чем отличается данный класс бинарного дерева от AVL-дерева?**

Данный класс бинарных деревьев не реализует балансировку, то есть не гарантирует, что дерево будет сбалансированным после добавления или удаления элементов. В результате дерево может иметь высоту O(n), что в худшем случае может привести к неэффективной работе с данными.

В отличие от него, дерево AVL - это самобалансирующееся бинарное дерево поиска, гарантирующее сбалансированность дерева после каждой операции вставки или удаления. Для этого AVL-дерево использует специальную технику балансировки, при которой разница в высоте между левым и правым поддеревьями узлов не превышает 1.

Это достигается путем автоматической ротации поддеревьев, чтобы сохранить баланс. Например, при вставке узла в AVL-дерево, проверяется балансировка каждого узла на пути от корня до вставленного узла, и в случае необходимости производится одна или несколько ротаций поддеревьев, чтобы вернуть дерево в сбалансированное состояние.

Таким образом, AVL-дерево гарантирует более эффективную работу со структурой данных, чем обычное бинарное дерево поиска, но за счет дополнительных затрат на балансировку при каждой операции вставки или удаления.

НО, можно реализовать класс так, чтобы при добавлении или удалении элементов тоже происходила балансировка.

# **Метод балансировки данного класса и его отличие от балансировки AVL-дерева**

Метод балансировки данного класса деревьев работает по следующему алгоритму:

1. Создается список, в котором будут храниться ключи всех узлов текущего поддерева.
2. Список сортируется в порядке возрастания.
3. Создается новый список, в который будут добавляться ключи в таком порядке, чтобы при их последовательной вставке в дерево, дерево получалось сбалансированным.
4. Создается новое сбалансированное дерево и заполняется ключами в порядке, определенном в новом списке.
5. Новое дерево вставляется на место текущего поддерева в исходном дереве.

Таким образом, метод балансировки данного класса деревьев не является самодостаточным и требует создания нового сбалансированного дерева. В отличие от метода балансировки AVL-дерева, который выполняет повороты узлов в существующем дереве, чтобы сохранить его сбалансированным.

# **Алгоритм создания сбалансированного списка ключей**

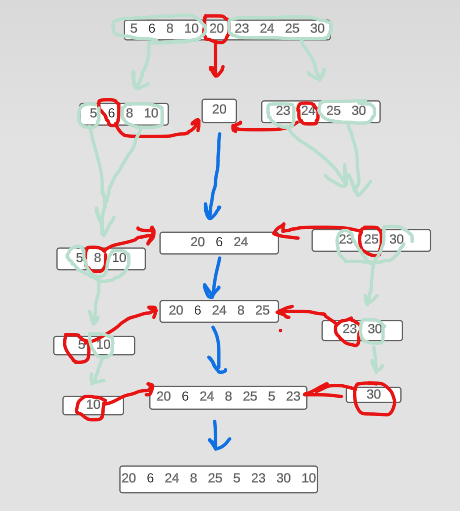
Сбалансированный список ключей – это список ключей, расположенных в таком порядке, чтобы при их последовательной вставке в дерево, дерево получалось сбалансированным.

При входе алгоритм создания сбалансированного списка ключей получает список ключей, отсортированный в порядке возрастания

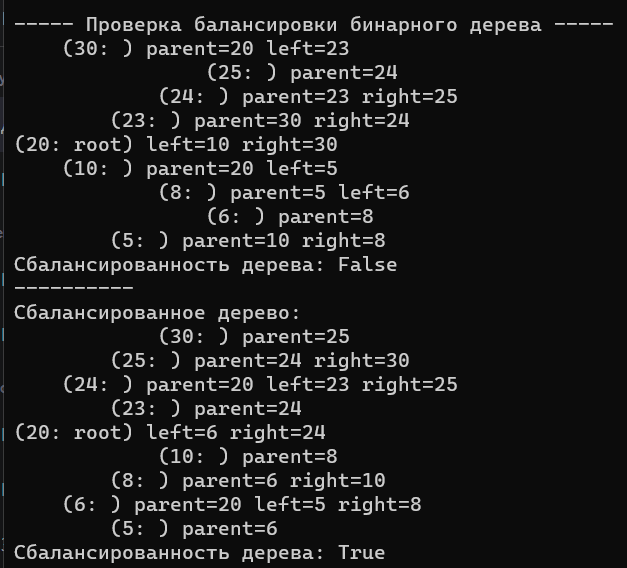
Алгоритм создания сбалансированного списка ключей:

1. Создание сбалансированного списка и вставка в него ключа, извлеченного из входящего списка, который находится в середине.
2. Создание списка правых и левых частей из входящего списка, методом деления входящего списка на две равные части.
3. Если в списке правых частей есть ключи, то извлекаем ключ, находящийся в середине списка, и добавляем в сбалансированный список.
4. Если в списке левых частей есть ключи, то извлекаем ключ, находящийся в середине списка, и добавляем в сбалансированный список.
5. Если список правых частей или список левых частей все еще имеют ключи, то возвращаемся к шагу 3.
6. Возвращаем сбалансированный список.





# **Тестирование балансировки**



# **Класс декартового дерева (дерамида)**

**Декартово дерево или дерамида** (англ. *Treap*) — это структура данных, объединяющая в себе [бинарное дерево поиска](https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0,_%D0%BD%D0%B0%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D1%80%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) и [бинарную кучу](https://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BA%D1%83%D1%87%D0%B0) (отсюда и второе её название: treap (tree + heap) и дерамида (дерево + пирамида), также существует название курево (куча + дерево).

Более строго, это бинарное дерево, в узлах которого хранятся пары (x,y), где x — это ключ, а y — это приоритет. Также оно является двоичным деревом поиска по x и пирамидой по y. Предполагая, что все x и все y являются различными, получаем, что если некоторый элемент дерева содержит (x0,y0), то у всех элементов в левом поддереве x<x0, у всех элементов в правом поддереве x>x0, а также и в левом, и в правом поддереве имеем: y<y0.

Дерамиды были предложены Сиделем (Siedel) и Арагон (Aragon) в 1996 г.

public void Add(int key, T value)

public void Add(int key, T value, int priority)

Метод **Add** добавляет новый узел в дерево. Он генерирует случайный приоритет для нового узла (второй вариант вводит значение приоритета) и создает экземпляр **TreapNode<T>**. Если дерево пустое (корень равен **null**), то новый узел становится корнем дерева. В противном случае происходит разделение дерева по ключу **key** с помощью метода **Split**, и новый узел становится корнем нового дерева с левым поддеревом **left** и правым поддеревом **right**.

public void Delete(int key)

Метод **Delete** удаляет узел из дерева по заданному ключу **key**. Он вызывает приватный рекурсивный метод **Delete**, передавая ему корень дерева и ключ для удаления. Результатом операции удаления является новый корень дерева.

public TreapNode<T> Find(int key)

Метод **Find** осуществляет поиск узла в дереве по заданному ключу **key**. Он вызывает приватный рекурсивный метод **FindNode**, передавая ему корень дерева и ключ для поиска. Результатом является найденный узел или **null**, если узел не найден.

public void View1() => PrintTree1(root, "");

public void View2() => PrintTree2(root, "");

Методы **View1** и **View2** отвечают за вывод дерева в горизонтальной и вертикальной форме соответственно.

Метод **View1** вызывает приватный метод **PrintTree1**, который рекурсивно проходит по дереву в префиксном порядке (обход в глубину). Для каждого узла дерева происходит вывод его значения с определенным отступом (**indent**), символами, обозначающими связи между узлами (**"└──"** или **"├──"**), и переход к следующим уровням дерева. Уровень каждого последующего узла имеет больший отступ, чтобы создать визуальную структуру дерева. При вызове рекурсивно метод **PrintTree1** передает левого и правого потомка текущего узла, а также обновляет отступ и флаг **last**, указывающий, является ли узел последним на текущем уровне.

Метод **View2** работает аналогично методу **View1**, но использует другие символы для обозначения связей между узлами и отличается визуальной структурой вывода. Здесь используются символы **"└─"** и **"├─"**, а также разные отступы (**" "** и **"│ "**). Рекурсивно вызывается приватный метод **PrintTree2**, который аналогично **PrintTree1** выводит дерево в префиксном порядке, но с измененными символами и отступами.

Оба эти метода позволяют визуализировать структуру дерева и увидеть связи между его узлами в удобном для чтения формате.

private TreapNode<T> Delete(TreapNode<T> node, int key)

Приватный метод **Delete** рекурсивно удаляет узел из дерева. Если текущий узел **node** равен **null**, то возвращается **null**. Если ключ **key** меньше ключа текущего узла, то рекурсивно вызывается **Delete** для левого поддерева узла **node**. Если ключ **key** больше ключа текущего узла, то рекурсивно вызывается **Delete** для правого поддерева узла **node**. Если ключи совпадают, то выполняется слияние левого и правого поддеревьев с помощью метода **Merge**, и результат возвращается.

private TreapNode<T> FindNode(TreapNode<T> node, int key)

Приватный метод **FindNode** рекурсивно выполняет поиск узла в дереве по заданному ключу **key**. Если текущий узел **node** равен **null** или ключ текущего узла равен **key**, то возвращается текущий узел. Если ключ **key** меньше ключа текущего узла, то рекурсивно вызывается **FindNode** для левого поддерева узла **node**. В противном случае, ключ **key** больше ключа текущего узла, и рекурсивно вызывается **FindNode** для правого поддерева узла **node**.

private void Split(TreapNode<T> node, int key, out TreapNode<T> left, out TreapNode<T> right)

Приватный метод **Split** разделяет дерево на два поддерева по ключу **key**. Если текущий узел **node** равен **null**, то левое и правое поддеревья устанавливаются в **null**. Если ключ **key** меньше ключа текущего узла, то рекурсивно вызывается **Split** для левого поддерева узла **node**. Результат разделения сохраняется в левом поддереве **left**, а правое поддерево **right** устанавливается равным текущему узлу **node**, а его левое поддерево **left** устанавливается во временную переменную **tempRight**. В противном случае, ключ **key** больше или равен ключу текущего узла, и рекурсивно вызывается **Split** для правого поддерева узла **node**. Результат разделения сохраняется в правом поддереве **right**, а левое поддерево **left** устанавливается равным текущему узлу **node**, а его правое поддерево **right** устанавливается во временную переменную **tempLeft**.

private TreapNode<T> Merge(TreapNode<T> left, TreapNode<T> right)

Приватный метод **Merge** объединяет два дерева **left** и **right** в одно дерево. Если левое поддерево **left** равно **null**, то возвращается правое поддерево **right**. Если правое поддерево **right** равно **null**, то возвращается левое поддерево **left**. Если приоритет левого поддерева **left** больше приоритета правого поддерева **right**, то рекурсивно вызывается **Merge** для правого поддерева **left.Right** и правого поддерева **right**. Результат объединения присваивается правому поддереву **left.Right**, и возвращается левое поддерево **left**. В противном случае, приоритет правого поддерева **right** больше или равен приоритету левого поддерева **left**, и рекурсивно вызывается **Merge** для левого поддерева **left** и левого поддерева **right.Left**. Результат объединения присваивается левому поддереву **right.Left**, и возвращается правое поддерево **right**.

private void PrintTree(TreapNode<T> node, int indent)

Приватный метод **PrintTree** рекурсивно выводит дерево на экран. Если текущий узел **node** не равен **null**, то сначала рекурсивно вызывается **PrintTree** для правого поддерева **node.Right** с увеличенным отступом **indent + 4**. Затем выводится строка с отступом **indent**, содержащая строковое представление текущего узла **node**, с помощью метода **ToString()**, и выводится на экран с помощью **Console.WriteLine**. После этого рекурсивно вызывается **PrintTree** для левого поддерева **node.Left** с увеличенным отступом **indent + 4**.

# **Заключение**

В данном классе бинарных деревьев реализованы основные операции, такие как добавление узла, поиск по ключу и удаление узла. Дерево представлено в виде класса **BinaryTree<T>**, где **T** - это тип данных, хранящийся в узлах дерева, и он должен быть сравнимым (**IComparable**).

Операция добавления узла (**Add**) осуществляет поиск места для нового узла в дереве и вставляет его на соответствующее место. Если узел с таким ключом уже существует, то его значение обновляется.

Операции поиска минимального (**NodeMin**) и максимального (**NodeMax**) ключей в дереве реализованы путем последовательного спуска влево и вправо от корня дерева.

Для удаления узла (**DeleteNode**) реализованы несколько случаев:

* Удаляемый узел не имеет детей: в этом случае удаляемый узел просто отсоединяется от родительского узла.
* Удаляемый узел имеет только одного ребенка: в этом случае ребенок заменяет удаляемый узел, и связи родительского узла соответственно перенаправляются на ребенка.
* Удаляемый узел имеет двух детей: в этом случае происходит сложная операция, включающая создание сбалансированного списка ключей, формирование нового дерева на основе этого списка и замена удаляемого узла новым деревом.

Для балансировки дерева (**BalanceTree**) используется алгоритм, который формирует список ключей в порядке возрастания и затем создает новое сбалансированное дерево на основе этого списка. Алгоритм балансировки вызывается как для всего дерева, так и для узла с двумя детьми при его удалении.

Для проверки сбалансированности дерева реализован метод IsBalanced, который проверяет сбалансированность корневого узла, вычисляя высоты поддеревьев и сравнивая их. Если разница высот превышает 1, то возвращает false.

Класс также предоставляет методы для вывода дерева на экран в различных форматах (**View**, **View2**, **View3**), а также методы для очистки дерева (**Clean**).

В целом, данный класс бинарных деревьев реализует основные операции и обладает функциональностью для удаления узлов и балансировки дерева, что позволяет эффективно управлять структурой дерева и обеспечивать его оптимальное состояние.

В данном классе **TreapTree<T>** реализована структура данных Treap (Tree + Heap), которая сочетает в себе свойства двоичного дерева поиска и бинарной кучи. Treap обладает следующими особенностями:

1. Дерево упорядочено по ключам: для каждого узла все ключи в левом поддереве меньше ключа узла, а все ключи в правом поддереве больше или равны ключу узла. Это свойство позволяет эффективно выполнять операции поиска, добавления и удаления узлов в дереве.
2. Дерево упорядочено по приоритетам: для каждого узла все приоритеты в левом поддереве больше или равны приоритету узла, а все приоритеты в правом поддереве меньше приоритета узла. Приоритеты используются для поддержания баланса в дереве и обеспечивают случайность при вставке новых узлов.

Класс **TreapTree<T>** имеет следующие методы:

* **Add(int key, T value)**: добавляет новый узел с заданным ключом и значением в дерево. При этом генерируется случайный приоритет для узла, и он вставляется в соответствующую позицию в дереве, учитывая и ключ, и приоритет.
* **Delete(int key)**: удаляет узел с заданным ключом из дерева. Метод осуществляет поиск узла по ключу и затем выполняет операцию удаления, сливая левое и правое поддеревья.
* **Find(int key)**: выполняет поиск узла по заданному ключу и возвращает найденный узел. Если узел не найден, возвращается **null**.
* **View()**: выводит содержимое дерева на экран в виде структурированного списка узлов. Для этого используется рекурсивный метод **PrintTree**, который обходит дерево в порядке справа-корень-лево и выводит каждый узел с соответствующим отступом.

Класс **TreapTree<T>** позволяет эффективно работать с большими объемами данных, обеспечивая быстрые операции поиска, добавления и удаления узлов в дереве. Также TreapTree поддерживает упорядоченность по ключам и случайность приоритетов, что способствует балансу дерева и предотвращает возникновение худшего случая при операциях.

# **Список литературы**

* C#. Алгоритмы и структуры данных. Н. А. Тюкачев, В. Г. Хлебостроев
* Вирт Н. АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ. М.Мир 1989
* "Алгоритмы. Построение и анализ" Дж. Кормена, Ч. Лейзерсон, Р. Ривест, К. Штайн.
* Алгоритмы. Построение и анализ. Андрей Левитин
* Балансировка | Алгоритмы на деревьях <https://ru.hexlet.io/courses/algorithms-trees/lessons/balancing/theory_unit>
* Бинарные деревья | Алгоритмы на деревьях <https://ru.hexlet.io/courses/algorithms-trees/lessons/binary/theory_unit>
* АВЛ-деревья / Хабр <https://habr.com/ru/articles/150732/>
* Декартово дерево: Часть 1. Описание, операции, применения / Хабр <https://habr.com/ru/articles/101818/>
* Декартово дерево — Викиконспекты <https://neerc.ifmo.ru/wiki/index.php?title=Декартово_дерево>