Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

# КУРСОВАЯ РАБОТА

По дисциплине «Объектно-ориентированное программирование»

**B - дерево**

(семестр VI)

Студент группы 3331506/20401 Я.Д. Лапкин
подпись, дата инициалы и фамилия

Оценка выполненной студентом работы:

Преподаватель, доцент, к.т.н. М. С. Ананьевский
подпись, дата инициалы и фамилия

Санкт-Петербург
2025

# ВВЕДЕНИЕ

B-дерево — это сбалансированная древовидная структура данных, которая широко используется для эффективного хранения и поиска информации, особенно при работе с большими объемами данных. Эта структура была разработана Рудольфом Байером и Эдвардом Маккремоном в 1972 году.

Цель данной курсовой работы — изучить структуру B-дерева и реализовать основные операции над ним: поиск, вставку и удаление ключей. В рамках работы будет рассмотрен алгоритм построения B-дерева, а также проведен анализ временной сложности указанных операций.

*Применение алгоритма*

Структура B-дерева применяется для организации индексов во многих современных системах управления базами данных.

B-дерево может применяться для структурирования информации на жёстком диске. Время доступа к произвольному блоку на жёстком диске очень велико (порядка миллисекунд), поскольку оно определяется скоростью вращения диска и перемещения головок. Поэтому важно уменьшить количество узлов, просматриваемых при каждой операции. Использование поиска по списку каждый раз для нахождения случайного блока могло бы привести к чрезмерному количеству обращений к диску вследствие необходимости последовательного прохода по всем его элементам, предшествующим заданному, тогда как поиск в B-дереве, благодаря свойствам сбалансированности и высокой ветвистости, позволяет значительно сократить количество таких операций.

*Словесное описание алгоритма***.**

B-дерево представляет из себя совокупность связанных узлов, удовлетворяющих следующим правилам:

- Ключи в каждом узле отсортированы;
- В корне содержится от 1 до 2t-1 ключей, где t – параметр дерева;
- Во всех остальных узлах содержится от t-1 до 2t-1 ключей;
- Листья – узлы, у которых нет потомков;
- Глубина всех листьев одинакова;
- Другие узлы, содержащие n ключей, содержат n+1 потомков. При этом:
    1. Первый потомок содержит ключи из интервала $(-\infty,K1)$;
    2. Каждый следующий потомок содержит ключи в интервале $(K_{i2},K_i)$;
    3. Последний потомок содержит ключи в интервале $(K_n,\infty)$.

# ОСНОВНАЯ ЧАСТЬ

*Операции над B-деревом*:

- Поиск
- Вставка
- Удаление

### *Поиск по B-дереву*

Поиск по B-дереву аналогичен поиску по двоичному дереву поиска. В двоичном дереве поиска поиск начинается с корня и каждый раз принимается двустороннее решение (пойти по левому поддереву или по правому). В B-дереве поиск также начинается с корневого узла, но на каждом шаге

принимается n-стороннее решение, где n – это общее количество потомков рассматриваемого узла. В B-дереве сложность поиска составляет **O(log n)**.

Поиск происходит следующим образом:

*1.* Считать элемент для поиска.

*2.* Сравнить искомый элемент с первым значением ключа в корневом узле дерева.

*3.* Если они совпадают, вывести: «Искомый узел найден!» и завершить поиск.

*4.* Если они не совпадают, проверить больше или меньше значение элемента, чем текущее значение ключа.

*5.* Если искомый элемент меньше, продолжить поиск по левому поддереву.

*6.* Если искомый элемент больше, сравнить элемент со следующим значением ключа в узле и повторять Шаги 3, 4, 5 и 6 пока не будет найдено совпадение или пока искомый элемент не будет сравнен с последним значением ключа в узле-листе.

*7.* Если последнее значение ключа в узле-листе не совпало с искомым, вывести «Элемент не найден!» и завершить поиск.

*Операция вставки в B-дерево*

В B-дереве новый элемент может быть добавлен только в узел-лист. Это значит, что новая пара ключ-значение всегда добавляется только к узлу-листу.

Вставка происходит следующим образом:

*1.* Проверить пустое ли дерево.

*2.* Если дерево пустое, создать новый узел с новым значением ключа и его принять за корневой узел.

*3.* Если дерево не пустое, найти подходящий узел-лист, к которому будет добавлено новое значение, используя логику дерева двоичного поиска.

*4.* Если в текущем узле-листе есть незанятая ячейка, добавить новый ключ-значение к текущему узлу-листу, следуя возрастающему порядку значений ключей внутри узла.

*5.* Если текущий узел полон и не имеет свободных ячеек, разделите узел-лист, отправив среднее значение родительскому узлу. Повторяйте шаг, пока отправляемое значение не будет зафиксировано в узле.

*6.* Если разделение происходит с корнем дерева, тогда среднее значение становится новым корнем дерева и высота дерева увеличивается на единицу.

*Пример*

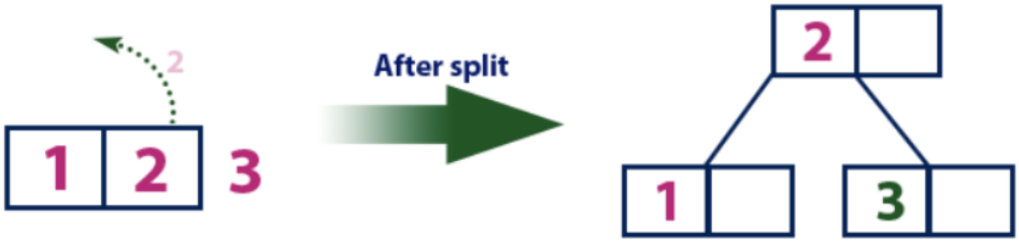
Давайте создадим B-дерево порядка 3, добавляя в него числа от 1 до 6.

1. Поскольку «1» — это первый элемент дерева – он вставляется в новый узел и этот узел становится корнем дерева.

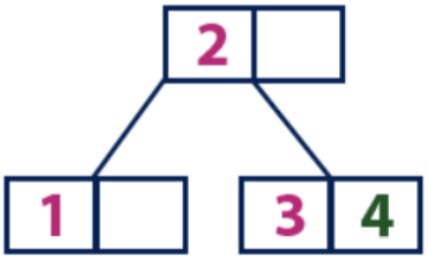

2. Элемент «2» добавляется к существующему узлу-листу. Сейчас у нас всего один узел, следовательно он является и корнем и листом одновременно. В этом листе имеется пустая ячейка. Тогда «2» встает в эту пустую ячейку.


3. Элемент «3» добавляется к существующему узлу-листу. Сейчас у нас только один узел, который одновременно является и корнем и

листом. У этого листа нет пустой ячейки. Поэтому мы разделяем этот узел, отправляя среднее значение (2) в родительский узел. Однако у текущего узла родительского узла нет. Поэтому среднее значение становится корневым узлом дерева.

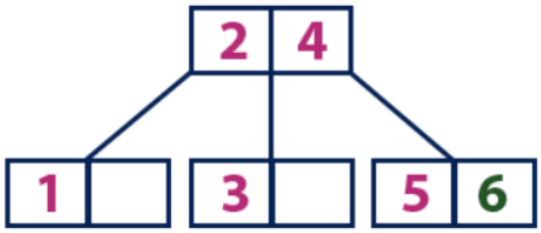



4. Элемент «4» больше корневого узла со значением «2», при этом корневой узел не является листом. Поэтому мы двигаемся по правому поддереву от «2». Мы приходим к узлу-листу со значением «3», у которого имеется пустая ячейка. Таким образом, мы можем вставить элемент «4» в эту пустую ячейку.




5. Элемент «5» больше корневого узла со значением «2», при этом корневой узел не является листом. Поэтому мы двигаемся по правому поддереву от «2». Мы приходим к узлу-листу и обнаруживаем, что он уже полон и не имеет пустых ячеек. Тогда мы делим этот узел, отправляя среднее значение (4) в родительский узел (2). В родительском узле есть для него пустая ячейка, поэтому значение «4» добавляется к узлу, в котором уже есть значение «2», а новый элемент «5» добавляется в качестве нового листа.


6. Элемент «6» больше, чем элементы корня «2» и «4», который не является листом. Мы двигаемся по правому поддереву от элемента «4». Мы достигаем листа со значением «5», у которого есть пустая ячейка, поэтому элемент «6» помещаем как раз в нее.




## РЕАЛИЗАЦИЯ АЛГОРИТМА

Для описания каждого узла была специально создана структура Node, в ней будем хранить указатель на родителя и ребенка, а также ключи и их количество.

Объявили общий класс Tree – где будем хранить все дерево.

Рассмотрим его приватные методы.

- сортировка ключей в узле, реализованная в методе tree::sortKeys – используем сортировку вставками.
- создание узла Tree::createNode
- создание нового корня Tree::createRootParent

- сортировка указателей на детей Tree::sortChildPointers

- Поиск узла, в котором содержится ключ: Tree::searchNode
- Поиск положения ключа в узле Tree::searchInNode
- Добавление ключа к узлу Tree::insertToNote
- Удаление ключа из узла: Tree::removeFromNode
- Удаление ключа из листа Tree::removeFromLeaf
- Слияние узлов Tree::mergeNodes

Рассмотрим его публичные методы:

- Поиск ключа в дереве: Tree::search

- Добавление ключа к дереву Tree::insert

- Удаление ключа из дерева Tree::remove

# АНАЛИЗ АЛГОРИТМА

1. Временная сложность алгоритма

Каждое действие при работе с B-деревом (поиск, удаление, добавление ключа) происходит за время O($t \cdot \log_t n$), n – количество узлов.

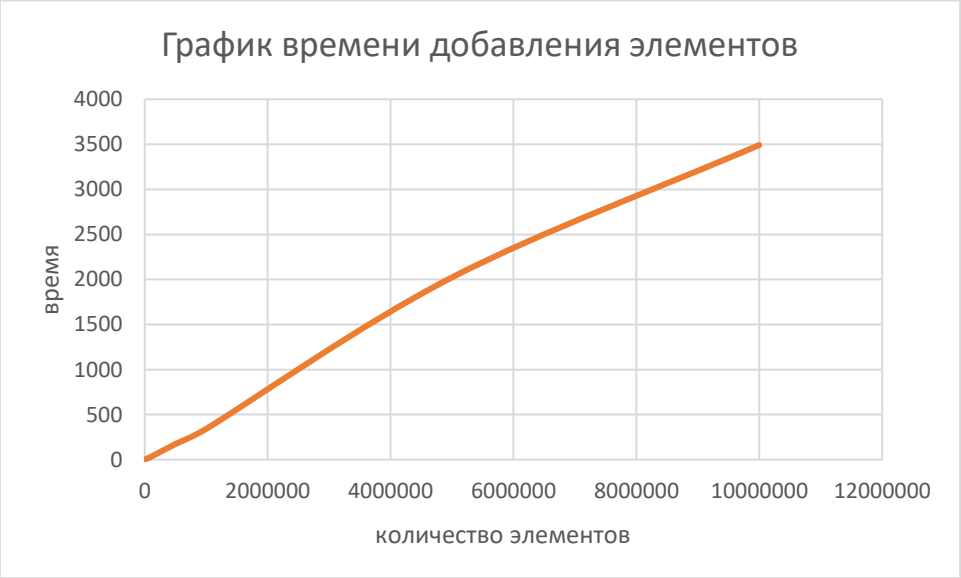
2. время выполнения алгоритма:

Таблица 1

результаты измерения времени работы программы

| количество элементов | операция | | |
|---|---|---|---|
| | insert | search | remove |
| 10000 | 4 | 2 | 4 |
| 50000 | 16 | 10 | 16 |
| 100000 | 31 | 24 | 34 |
| 500000 | 174 | 167 | 200 |
| 1000000 | 342 | 275 | 346 |
| 5000000 | 2023 | 1468 | 1873 |
| 10000000 | 3492 | 2993 | 3627 |

Графики времени работы алгоритма:



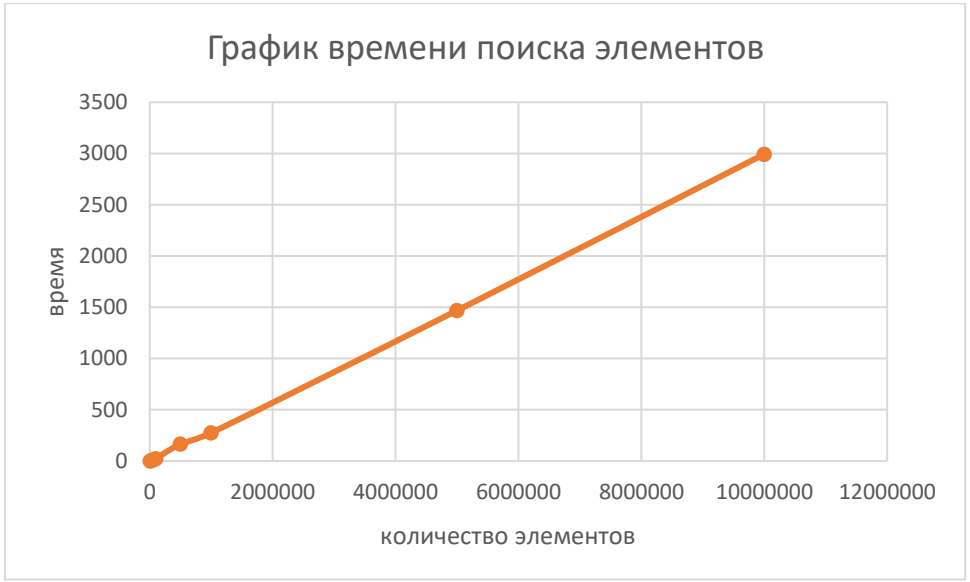


График времени поиска элементов
3500
3000
2500
2000
1500
1000
500
0
время
0
2000000
4000000
6000000
8000000
10000000
12000000
количество элементов

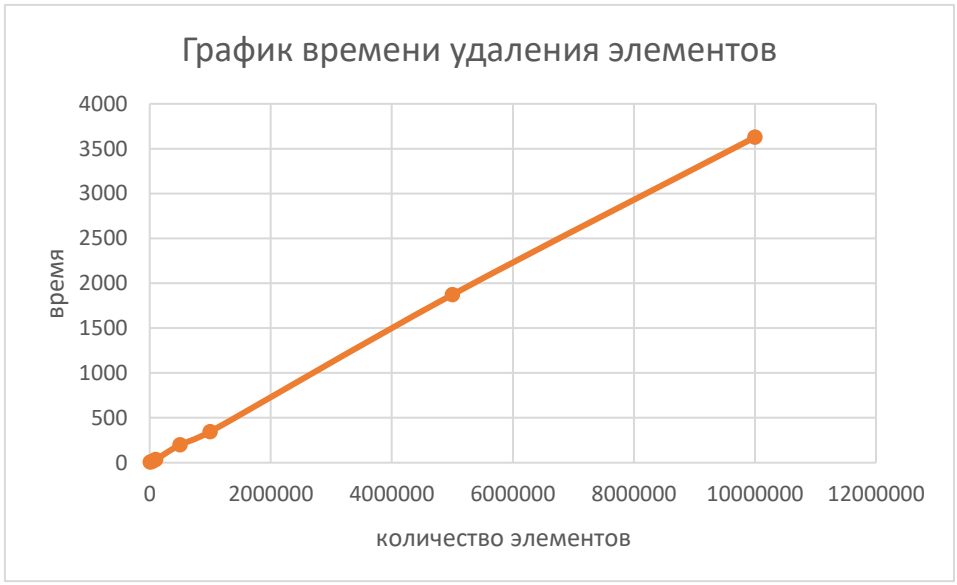


График времени удаления элементов
4000
3500
3000
2500
2000
1500
1000
500
0
время
0
2000000
4000000
6000000
8000000
10000000
12000000
количество элементов

# ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была успешно изучена структура B-дерева и реализованы основные операции над ним: поиск, вставка и удаление ключей. Реализация на языке C++ продемонстрировала эффективность B-дерева при работе с большими объемами данных, что соответствует его теоретическим свойствам. Экспериментальное тестирование и анализ временных характеристик подтвердили, что все операции выполняются за логарифмическое время $O(\log n)$, что делает B-дерево идеальным выбором для задач, требующих быстрого доступа и модификации данных.

*Анализ графиков и временных характеристик*

На основе проведенных измерений времени работы программы были построены графики, которые наглядно демонстрируют зависимость времени выполнения операций от количества элементов в дереве. Как видно из таблицы и графиков:

**Операция вставки (insert)**: Время выполнения растет пропорционально увеличению количества элементов, что соответствует теоретической сложности $O(\log n)$. Например, для 1 000 000 элементов время вставки составило 342 мкс, а для 10 000 000 элементов — 3492 мкс.

**Операция поиска (search)**: Показала наилучшие результаты благодаря эффективной структуре B-дерева, которая минимизирует количество сравнений. Для 1 000 000 элементов время поиска составило 275 мкс, а для 10 000 000 — 2993 мкс.

**Операция удаления (remove)**: Также демонстрирует логарифмическую сложность, хотя требует дополнительных шагов для поддержания сбалансированности дерева. Для 1 000 000 элементов время удаления составило 346 мкс, а для 10 000 000 — 3627 мкс.

Графики показывают, что все операции сохраняют предсказуемую и эффективную производительность даже при значительном увеличении объема данных. Это подтверждает, что B-дерево является надежной структурой для использования в системах, где критически важны скорость и стабильность работы, таких как базы данных и файловые системы.

# СПИСОК ЛИТЕРАТУРЫ


- https://ru.wikipedia.org/wiki/B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE
- https://habr.com/post/114154/
- *Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн.* Глава 18. B-деревья // Алгоритмы: построение и анализ = Introduction to Algorithms. — 2-е изд. — М.: Вильямс, 2006. — С. 515—536
- Структура данных B-дерево, Хабр: https://habr.com/ru/companies/otus/articles/459216/

# ПРИЛОЖЕНИЕ

```
struct Node {
    Node* parent;
    Node* children[2 * T];
    double keys[2 * T - 1];
    int keysCount;
};

class Tree {
public:
    Node* root;

    void insert(double value);
    double* search(double value);
    void remove(double value);

    Tree() {
        root = new Node;
        root->keysCount = 0;
        root->parent = nullptr;

        for (int i = 0; i < 2 * T; i++) {
            root->children[i] = nullptr;
        }
    }
```

```
private:
    void insertToNode(Node* currentNode, double value);
    Node* createNode(Node* parent);
    Node* searchNode(Node* currentNode, double value);
    double* searchInNode(Node* currentNode, double value);
    void removeFromNode(Node* currentNode, double value);
    void sortChildPointers(Node* currentNode);
    Node* createRootParent();
    void sortKeys(Node* currentNode);
    void removeFromLeaf(Node* currentNode, double value);
    void mergeNodes(Node* firstNode, Node* secondNode);
};

#endif // B_TREE_H
```

```cpp
#include "BTree.h"
#include <iostream>

Node* Tree::createNode(Node* parent) {
    Node* newNode = new Node;
    newNode->keysCount = 0;
    newNode->parent = parent;

    for (int i = 0; i < 2 * T; i++) {
        newNode->children[i] = nullptr;
    }

    for (int i = 0; i < 2 * T - 1; i++) {
        newNode->keys[i] = 0;
    }

    return newNode;
}

Node* Tree::createRootParent() {
    Node* newParent = new Node;
    root->parent = newParent;
    newParent->children[0] = root;
    newParent->keysCount = 0;
    root = newParent;
    newParent->parent = nullptr;

    for (int i = 1; i < 2 * T; i++) {
        newParent->children[i] = nullptr;
    }

    for (int i = 1; i < 2 * T - 1; i++) {
        newParent->keys[i] = 0;
    }

    return newParent;
}
```

```
void Tree::sortKeys(Node* currentNode) {
    int i = 1;
    int j = 2;
    double tempKey;
    Node* tempChild;

    while (i < currentNode->keysCount) {
        if (currentNode->keys[i] > currentNode->keys[i-1]) {
            i = j;
            j++;
        } else {
            tempKey = currentNode->keys[i];
            tempChild = currentNode->children[i];

            currentNode->keys[i] = currentNode->keys[i-1];
            currentNode->children[i] = currentNode->children[i-1];

            currentNode->keys[i-1] = tempKey;
            currentNode->children[i-1] = tempChild;

            i--;

            if (i == 0) {
                i = j;
                j++;
            }
        }
    }
}
```

```
void Tree::sortChildPointers(Node* currentNode) {
    int i = 1;
    int j = 2;
    Node* temp;

    while (i < currentNode->keysCount + 1) {
        if ((currentNode->children[i] == nullptr) ||
            (currentNode->children[i]->keys[0] > currentNode->children[i-1]->keys[0]))
            i = j;
            j++;
        } else {
            temp = currentNode->children[i];
            currentNode->children[i] = currentNode->children[i-1];
            currentNode->children[i-1] = temp;
            i--;

            if (i == 0) {
                i = j;
                j++;
            }
        }
    }
}
```

```
Node* Tree::searchNode(Node* currentNode, double value) {
    if (currentNode->keysCount == 0) return currentNode;

    if (value > currentNode->keys[currentNode->keysCount - 1]) {
        if (currentNode->children[0] == nullptr) {
            return currentNode;
        }
        return searchNode(currentNode->children[currentNode->keysCount], value);
    }

    for (int i = 0; i < currentNode->keysCount; i++) {
        if (value == currentNode->keys[i]) {
            return currentNode;
        }
        if (value < currentNode->keys[i]) {
            if (currentNode->children[i] == nullptr) {
                return currentNode;
            }
            return searchNode(currentNode->children[i], value);
        }
    }

    return nullptr;
}

double* Tree::searchInNode(Node* currentNode, double value) {
    for (int i = 0; i < currentNode->keysCount; i++) {
        if (value == currentNode->keys[i]) {
            return &currentNode->keys[i];
        }
    }
    return nullptr;
}
```

```
void Tree::insert(double value) {
    using namespace std;
    Node* targetNode = searchNode(root, value);
    double* position = searchInNode(targetNode, value);

    if (position != nullptr) {
        cout << value << " already exists at address " << search(value) << endl;
        return;
    }
    insertToNode(targetNode, value);
}
```

```
void Tree::insertToNode(Node* currentNode, double value) {
    if (currentNode->keysCount < 2 * T - 1) {
        currentNode->keys[currentNode->keysCount] = value;
        currentNode->keysCount++;
        sortKeys(currentNode);
    } else {
        if (currentNode->parent == nullptr) {
            createRootParent();
        }
        insertToNode(currentNode->parent, currentNode->keys[T-1]);

        double middleKey = currentNode->keys[T-1];
        Node* parent = currentNode->parent;
        parent->children[parent->keysCount] = createNode(parent);
        Node* rightChild = parent->children[parent->keysCount];

        int index = 0;
        for (int i = T; i < 2 * T - 1; i++) {
            rightChild->keys[index] = currentNode->keys[i];
            rightChild->keysCount++;
            currentNode->keys[i] = 0;
            currentNode->keysCount--;

            if (currentNode->children[i] != nullptr) {
                rightChild->children[index] = currentNode->children[i];
                currentNode->children[i]->parent = rightChild;
                currentNode->children[i] = nullptr;
            }
            index++;
        }

        if (currentNode->children[2 * T - 1] != nullptr) {
            rightChild->children[index] = currentNode->children[2 * T - 1];
            currentNode->children[2 * T - 1]->parent = rightChild;
            currentNode->children[2 * T - 1] = nullptr;
        }
```

```
        currentNode->keys[T-1] = 0;
        currentNode->keysCount--;

        if (value < middleKey) {
            currentNode->keys[currentNode->keysCount] = value;
            currentNode->keysCount++;
            sortKeys(currentNode);
        } else {
            rightChild->keys[rightChild->keysCount] = value;
            rightChild->keysCount++;
            sortKeys(rightChild);
        }

        sortKeys(parent);
        sortChildPointers(parent);
    }
}

double* Tree::search(double value) {
    Node* targetNode = searchNode(root, value);
    return searchInNode(targetNode, value);
}
```

```cpp
void Tree::remove(double value) {
    removeFromNode(searchNode(root, value), value);

    if (root->keysCount == 0 && root->children[0] != nullptr) {
        Node* newRoot = root->children[0];
        delete root;
        root = newRoot;
        root->parent = nullptr;
    }
}

void Tree::removeFromNode(Node* currentNode, double value) {
    if (currentNode->children[0] == nullptr) {
        removeFromLeaf(currentNode, value);
    } else {
        Node* targetNode = currentNode;
        int keyIndex;

        for (keyIndex = 0; keyIndex < currentNode->keysCount; keyIndex++) {
            if (value == currentNode->keys[keyIndex]) break;
        }

        currentNode = currentNode->children[keyIndex];
        if (currentNode->children[0] != nullptr) {
            do {
                currentNode = currentNode->children[currentNode->keysCount];
            } while (currentNode->children[0] != nullptr);
        }

        double replacementKey = currentNode->keys[currentNode->keysCount - 1];
        removeFromNode(currentNode, replacementKey);
        targetNode->keys[keyIndex] = replacementKey;
    }
}
```

```
void Tree::removeFromLeaf(Node* currentNode, double value) {
    for (int i = 0; i < currentNode->keysCount; i++) {
        if (value == currentNode->keys[i]) {
            for (int k = i; k < currentNode->keysCount; k++) {
                currentNode->keys[k] = currentNode->keys[k + 1];
            }
            currentNode->keysCount--;
            break;
        }
    }

    if (currentNode->parent == nullptr || currentNode->keysCount >= T - 1) {
        return;
    }

    int childIndex = -1;
    for (int i = 0; i <= currentNode->parent->keysCount; i++) {
        if (currentNode->parent->children[i] == currentNode) {
            childIndex = i;
            break;
        }
    }

    Node* leftSibling = (childIndex > 0) ? currentNode->parent->children[childIndex - 1] : nullptr;
    Node* rightSibling = (childIndex < currentNode->parent->keysCount) ?
                         currentNode->parent->children[childIndex + 1] : nullptr;

    if (rightSibling != nullptr && rightSibling->keysCount > T - 1) {
        // Заимствование у правого соседа
        insertToNode(currentNode, currentNode->parent->keys[childIndex]);

        if (currentNode->children[0] != nullptr) {
            currentNode->children[currentNode->keysCount] = rightSibling->children[0];
            rightSibling->children[0]->parent = currentNode;
        }
```

```
        currentNode->parent->keys[childIndex] = rightSibling->keys[0];
        removeFromLeaf(rightSibling, rightSibling->keys[0]);
        return;
    }

    if (leftSibling != nullptr && leftSibling->keysCount > T - 1) {
        // Заимствование у левого соседа
        insertToNode(currentNode, currentNode->parent->keys[childIndex - 1]);

        if (currentNode->children[0] != nullptr) {
            for (int i = currentNode->keysCount; i > 0; i--) {
                currentNode->children[i] = currentNode->children[i - 1];
            }
            currentNode->children[0] = leftSibling->children[leftSibling->keysCount];
            leftSibling->children[leftSibling->keysCount]->parent = currentNode;
        }

        currentNode->parent->keys[childIndex - 1] = leftSibling->keys[leftSibling->keysCount - 1];
        removeFromLeaf(leftSibling, leftSibling->keys[leftSibling->keysCount - 1]);
        return;
    }

    if (rightSibling != nullptr && rightSibling->keysCount <= T - 1) {
        mergeNodes(currentNode, rightSibling);
        return;
    }

    if (leftSibling != nullptr && leftSibling->keysCount <= T - 1) {
        mergeNodes(leftSibling, currentNode);
        return;
    }
}
```

```cpp
void Tree::mergeNodes(Node* leftNode, Node* rightNode) {
    int childIndex;
    for (childIndex = 0; childIndex <= leftNode->parent->keysCount; childIndex++) {
        if (leftNode->parent->children[childIndex] == leftNode) {
            break;
        }
    }

    insertToNode(leftNode, leftNode->parent->keys[childIndex]);

    int index = leftNode->keysCount;
    for (int i = 0; i < rightNode->keysCount; i++) {
        leftNode->keys[index] = rightNode->keys[i];
        leftNode->children[index] = rightNode->children[i];
        if (rightNode->children[i] != nullptr) {
            rightNode->children[i]->parent = leftNode;
        }
        index++;
    }
    leftNode->keysCount += rightNode->keysCount;

    // Последний ребенок правого узла
    leftNode->children[index] = rightNode->children[rightNode->keysCount];
    if (rightNode->children[rightNode->keysCount] != nullptr) {
        rightNode->children[rightNode->keysCount]->parent = leftNode;
    }

    // Удаляем правый узел из родителя
    delete leftNode->parent->children[childIndex + 1];
    leftNode->parent->children[childIndex + 1] = nullptr;

    // Сдвигаем оставшихся детей родителя
    for (int i = childIndex + 1; i < leftNode->parent->keysCount; i++) {
        leftNode->parent->children[i] = leftNode->parent->children[i + 1];
    }

    removeFromLeaf(leftNode->parent, leftNode->parent->keys[childIndex]);
}
```