ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту по дисциплине

«Структуры и алгоритмы обработки данных»

на тему

РАЗРАБОТКА И ИССЛЕДОВАНИЕ 2-3-ДЕРЕВА

|  |  |
| --- | --- |
| Выполнил студент |  |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы |  |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу принял |  | ст. преп. Кафедры ВС Д. М. Берлизов |
|  | подпись |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Защищена |  | Оценка |  |
|  |  |  |  |

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#__RefHeading___Toc4327_3153689509)

[1 Алгорим 2-3-Дерева 4](#__RefHeading___Toc4329_3153689509)

[1.1 Разработка основных операций 2-3-Дерева 4](#__RefHeading___Toc4331_3153689509)

[2 Экспериментальное исследование эффективности алгоритма 8](#__RefHeading___Toc4337_3153689509)

[ЗАКЛЮЧЕНИЕ 9](#__RefHeading___Toc4343_3153689509)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 10](#__RefHeading___Toc4345_3153689509)

[ПРИЛОЖЕНИЕ 11](#__RefHeading___Toc4347_3153689509)

[1 Исходный код программы 11](#__RefHeading___Toc4349_3153689509)

**ВВЕДЕНИЕ**

2-3-Дерево – это структура данных, 2-3 дерева были изобретены [Джоном Хопкрофтом](https://translated.turbopages.org/proxy_u/en-ru.ru.18823249-639756ae-a9d6252f-74722d776562/https/en.wikipedia.org/wiki/John_Hopcroft) в 1970 году, являющаяся [B-деревом](https://ru.wikipedia.org/wiki/B-дерево), каждый узел которого имеет либо два потомка и одно поле, либо три потомка и два поля. Листовые вершины являются исключением — у них нет детей, но есть одно или два поля. 2-3 деревья сбалансированы, то есть все листовые вершины находятся на одной высоте от корня дерева.

Где же может использоваться 2-3-Дерево? Начнем с того, где используется B-дерево: оно может применяться для структурирования информации на жёстком диске (как правило, метаданных). 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. 2-3-Дерево хорошо подходит для структурирования информации в ОЗУ, так как имеет невысокий коэффициент ветвления, нежели B-дерево.

В ходе данной работы будут реализованы основные операции для взаимодействия с 2-3-Деревом, а также вычислена асимптотическая сложность данных операций.

**Алгоритм 2-3-Дерева**

**Разработка основных операций 2-3-Дерева**

Нельзя начать разработку алгоритма без знания определенных свойств данного алгоритма, в нашем случае у 2-3-Дерева есть 4 свойства:

* Все не листовые вершины содержат одно поле и 2 поддерева или 2 поля и 3 поддерева.
* Все листовые вершины находятся на одном уровне (на нижнем уровне) и содержат 1 или 2 поля.
* Все данные отсортированы (по принципу двоичного дерева поиска).
* Не листовые вершины содержат одно или два поля, указывающие на диапазон значений в их поддеревьях. Значение первого поля строго больше наибольшего значения в левом поддереве и меньше или равно наименьшему значению в правом поддереве (или в центральном поддереве, если это 3-вершина); аналогично, значение второго поля (если оно есть) строго больше наибольшего значения в центральном поддереве и меньше или равно, чем наименьшее значение в правом поддереве. Эти не листовые вершины используются для направления функции поиска к нужному поддереву и, в конечном итоге, к нужному листу. (*прим. Это свойство не будет выполняться, если у нас есть одинаковые ключи. Поэтому возможна ситуация, когда равные ключи находятся в левом и правом поддеревьях одновременно, тогда ключ в не листовой вершине будет совпадать с этими ключами. Это никак не сказывается на правильности работы и производительности алгоритма.*).

Весь алгоритм, по требованию, будет реализован на языке программирования Си (C).

Начнем с вставления элемента **insert()**

* Если дерево пусто, то создать новую вершину, вставить ключ и вернуть в качестве корня эту вершину, иначе
* Если вершина является листом, то вставляем ключ в эту вершину и если получили 3 ключа в вершине, то разделяем её, иначе
* Сравниваем ключ key с первым ключом в вершине, и если key меньше данного ключа, то идем в первое поддерево и переходим к пункту 2, иначе
* Смотрим, если вершина содержит только 1 ключ (является 2-вершиной), то идем в правое поддерево и переходим к пункту 2, иначе
* Сравниваем ключ key со вторым ключом в вершине, и если key меньше второго ключа, то идем в среднее поддерево и переходим к пункту 2, иначе
* Идем в правое поддерево и переходим к пункту 2.

**Функция поиска элемента search()**

Поиск такой же простой, как и в бинарном дереве поиска:

* Ищем искомый ключ key в текущей вершине, если нашли, то возвращаем вершину, иначе
* Если key меньше первого ключа вершины, то идем в левое поддерево и переходим к пункту 1, иначе
* Если в дереве 1 ключ, то идем в правое поддерево (среднее, если руководствоваться нашим классом) и переходим к пункту 1, иначе
* Если key меньше второго ключа вершины, то идем в среднее поддерево и переходим к пункту 1, иначе
* Идем в правое поддерево и переходим к пункту 1.

**Функция удаления элемента из 2-3-Дерева remove\_elem()**

Удаление в 2-3-дереве, как и в любом другом дереве, происходит только из листа (из самой нижней вершины). Поэтому, когда мы нашли ключ, который нужно удалить, сначала надо проверить, находится ли этот ключ в листовой или не листовой вершине. Если ключ находится в не листовой вершине, то нужно найти эквивалентный ключ для удаляемого ключа из листовой вершины и поменять их местами. Для нахождения эквивалентного ключа есть два варианта: либо найти максимальный элемент в левом поддереве, либо найти минимальный элемент в правом поддереве. Я буду использовать второй вариант, искать минимальный в правом поддереве.

После того, как удалили ключ, у нас могут получиться концептуально 4 разные ситуации: 3 из них нарушают свойства дерева, а одна — нет. Поэтому для вершины, из которой удалили ключ, нужно вызвать функцию исправления **fix()**, которая вернет свойства 2-3 дерева. Случаи, которые описываются в функции, рассматриваются ниже.

Перейдем к возможным вариантам, которые могут появиться после удаления ключа. Для простоты будем рассматривать случаи, где глубина дерева равна 2. Приступим к частным случаям.

Случай 1:

Самый простой случай, также как и следующий, на которые хватит и одного предложения, чтобы понять: если дерево состоит из одной вершины (корень), которая имеет 1 ключ, то просто удаляем эту вершину.

Случай 2:

Если нужно удалить ключ из листа, где находятся два ключа, то мы просто удаляем ключ и на этом функция удаления закончена. Удалим из данного дерева ключ 4:

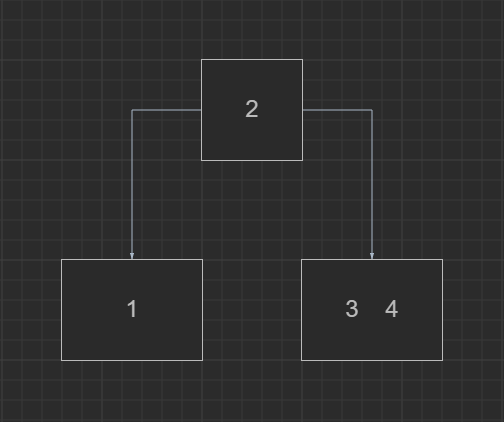
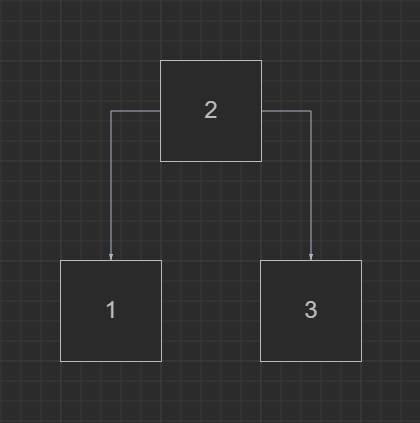
 

Рисунок . До удаления. Рисунок 2. После удаления.

Случай 3:

Мы удаляем ключ из вершины и вершина становится пустой. Если хотя бы у одного из братьев есть 2 ключа, то делаем простое правильное распределение и работа закончена. Под правильным распределением подразумевается, что при при цикличном сдвиге ключей между родителем и сыновьями также нужно будет перемещать и внуков родителя. Перераспределять ключи можно из любого брата, но удобнее всего из ближнего, который имеет 2 ключа, при этом мы циклично сдвигаем все ключи, например из примера ниже удалим ключ 1:

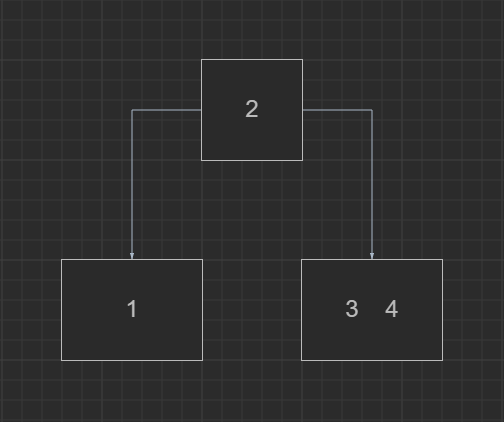
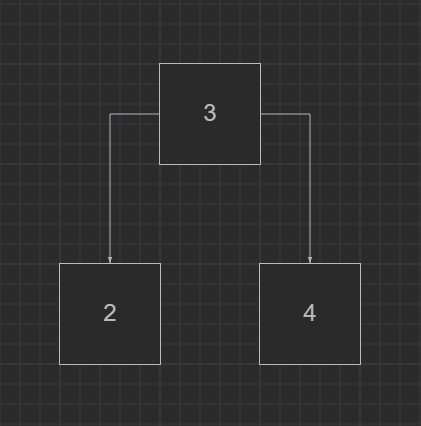
 

Рисунок 2. До удаления. Рисунок 2. После удаления.

Случай 4:

Самый сложный случай, так как после склеивания всегда обязательно идти по дереву вверх и опять применять операции либо **merge()**, либо **redistribute**. После **redistribute** алгоритм восстановления свойств 2-3-дерева после удаления ключа можно прекратить, так как вершины в этой операции не удаляются.

Удалим ключ 3 из вершины, родитель которой имеет только двух сыновей:

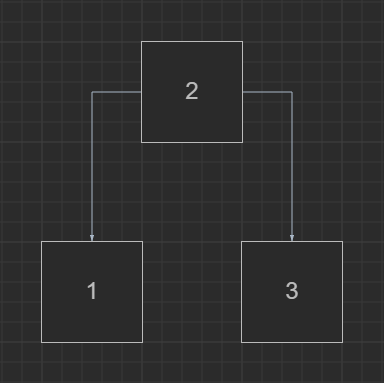
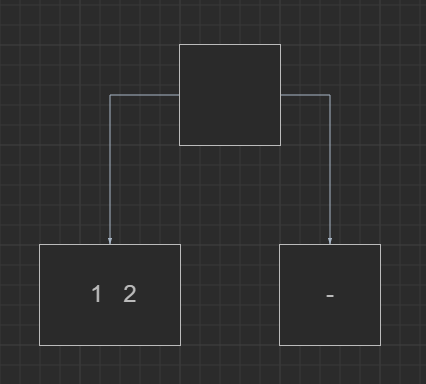
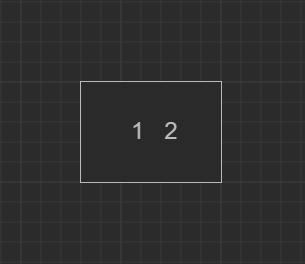
  

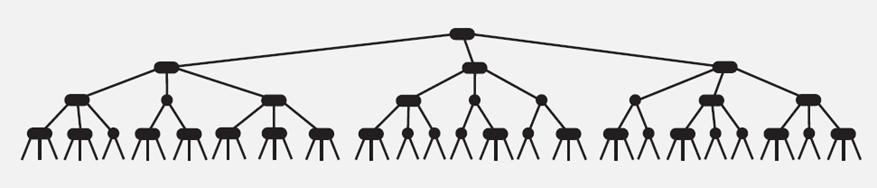
Рисунок 1. До удаления. Рисунок 2. Удалили ключ 3. Рисунок 3. Окончательный вариант.

Удалив ключ 3 после чего нужно перенести ключ из родителя в того сына, где ключ есть: в данном случае в левого сына. Затем нужно удалить вершину, из которой удалили ключ. И последний шаг — это проверить, если родитель был корнем дерева, то удалить этот корень и назначить новый корнем ту вершину, куда мы перенесли ключ, иначе придется вызывать функцию исправления **fix()** уже для родителя.

**Экспериментальное исследование эффективности алгоритма**

Эффективность алгоритмов взаимодействия с 2-3-Деревом тесно связана с высотой данного дерева, как и в BST, RB-Tree, B-Tree и других деревьях.

Полностью сбалансированное 2-3 дерево поиска показано ниже. Расстояние от каждого корневого узла до конечного узла одинаково:



В худшем случае, то есть все узлы являются узлами с двумя узлами, эффективность поиска равна , где N – количество элементов дерева.

Таблица теоретической временной эффективности основных операций:

| Алгоритм | Среднее время | Худшее время |
| --- | --- | --- |
| Поиск |  |  |
| Вставка |  |  |
| Удаление |  |  |

Для эксперимента буду вставлять по +50000 элементов до миллиона, и сравним эффективность вставки, удаления и поиска ключа.

Получаем такой график:

**Заключение**

В результате выполнения работы разработана и исследована структура данных 2-3-Дерево.

2-3-Дерево эффективнее бинарного дерева поиска (BST), но гораздо сложнее в реализации из-за разных типов вершин, а также дополнительных сравнений в вершине для того, чтобы перейти ниже. Плюс рекурсивный вызов функции **fix()** для того, чтобы восстановить свойства 2-3-Дерева. Улучшенное красно-черное дерево, основанное на дереве поиска 2-3, не только имеет более высокую эффективность, но и проще в реализации, чем дерево поиска 2-3.

Но 2-3-Дерево как более важная концепция и идея очень важны для красно-черного дерева и B-дерева.

Осуществлено моделирование разработанной структуры данных. Проведены экспериментальные исследования анализа эффективности, основных функций алгоритма такие как «Поиск», «Вставка» и «Удаление» - основных операций разработанной структуры данных. При проведении исследований получилось доказать, что время работы алгоритма соответствует теоретической сложности:

Поиск - , Вставка - , Удаление - .

**Список использованных источников**

1. Ахо А. В., Хопкрофт Д., Ульман Д. Д. Структуры данных и алгоритмы. – М.: Вильямс, 2001. – 384с.
2. Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К. Алгоритмы: построение и анализ. – 3-е изд. – М.: Вильямс, 2013. – 1328 с.
3. Курсносов М. Г., Берлизов Д. М.- Алгоритмы и структуры обработки информации. 2019.– 227 с.
4. Роберт Сикорд - Эффективный С. 2021.- 304 с.
5. Прат С., Уэит М., Мартин Д. - Язык программирования C. 6 изд. 1988.- 345 с.
6. Электронная энциклопедия:  
    - <https://ru.wikipedia.org/wiki/2-3-дерево>
7. Пособие по структуре:

- <https://www.geeksforgeeks.org/2-3-trees-search-and-insert/>

**Приложение**

2-3Tree.c

**struct** node \***insert**(**struct** node \*root, **int** k) // вставка ключа k в дерево с корнем root; всегда возвращаем корень дерева, т.к. он может меняться

{

**if**(!root){

root = malloc(**sizeof**(**struct** node));

root->size = **1**;

root->key[**0**] = k;

root->first = NULL;

root->second = NULL;

root->third = NULL;

root->fourth = NULL;

root->parent = NULL;

**return** root;

}

**if** (is\_leaf(root)) insert\_to\_node(root, k);

**else** **if** (k <= root->key[**0**]) insert(root->first, k);

**else** **if** ((root->size == **1**) || ((root->size == **2**) && k <= root->key[**1**])) insert(root->second, k);

**else** insert(root->third, k);

**return** split(root);

}

**struct** node \***split**(**struct** node \*item)

{

**if** (item->size < **3**) **return** item;

**struct** node \*x = malloc(**sizeof**(**struct** node)); // Создаем две новые вершины,

x->size = **1**;

x->key[**0**] = item->key[**0**];

x->first = item->first;

x->second = item->second;

x->third = NULL;

x->fourth = NULL;

x->parent = item->parent;

**struct** node \*y = malloc(**sizeof**(**struct** node)); // которые имеют такого же родителя, как и разделяющийся элемент.

y->size = **1**;

y->key[**0**] = item->key[**2**];

y->first = item->third;

y->second = item->fourth;

y->third = NULL;

y->fourth = NULL;

y->parent = item->parent;

**if** (x->first) x->first->parent = x; // Правильно устанавливаем "родителя" "сыновей".

**if** (x->second) x->second->parent = x; // После разделения, "родителем" "сыновей" является "дедушка",

**if** (y->first) y->first->parent = y; // Поэтому нужно правильно установить указатели.

**if** (y->second) y->second->parent = y;

**if** (item->parent) {

insert\_to\_node(item->parent, item->key[**1**]);

**if** (item->parent->first == item) item->parent->first = NULL;

**else** **if** (item->parent->second == item) item->parent->second = NULL;

**else** **if** (item->parent->third == item) item->parent->third = NULL;

// Дальше происходит своеобразная сортировка ключей при разделении.

**if** (item->parent->first == NULL) {

item->parent->fourth = item->parent->third;

item->parent->third = item->parent->second;

item->parent->second = y;

item->parent->first = x;

} **else** **if** (item->parent->second == NULL) {

item->parent->fourth = item->parent->third;

item->parent->third = y;

item->parent->second = x;

} **else** {

item->parent->fourth = y;

item->parent->third = x;

}

**struct** node \*tmp = item->parent;

free(item);

**return** tmp;

} **else** {

x->parent = item; // Так как в эту ветку попадает только корень,

y->parent = item; // то мы "родителем" новых вершин делаем разделяющийся элемент.

become\_node2(item, item->key[**1**], x, y);

**return** item;

}

}

**struct** node \***search**(**struct** node \*root, **int** k) // Поиск ключа k в 2-3 дереве с корнем root.

{

**if** (!root) **return** NULL;

**if** (find(root, k)) **return** root;

**else** **if** (k < root->key[**0**]) **return** search(root->first, k);

**else** **if** (((root->size == **2**) && (k < root->key[**1**])) || (root->size == **1**)) **return** search(root->second, k);

**else** **if** (root->size == **2**) **return** search(root->third, k);

**return** **0**;

}

**struct** node \***search\_min**(**struct** node \*root) // Поиск узла с минимальным элементов в 2-3-дереве с корнем root.

{

**if** (!root) **return** root;

**if** (!(root->first)) **return** root;

**else** **return** search\_min(root->first);

}

**struct** node \***remove\_elem**(**struct** node \*root, **int** k) // Удаление ключа k в 2-3-дереве с корнем p.

{

**struct** node \*item = malloc(**sizeof**(**struct** node));

item = search(root, k); // Ищем узел, где находится ключ k

**if** (!item) **return** root;

**struct** node \*min = NULL;

**if** (item->key[**0**] == k) min = search\_min(item->second); // Ищем эквивалентный ключ

**else** min = search\_min(item->third);

**if** (min) { // Меняем ключи местами

**int** z = (k == item->key[**0**] ? item->key[**0**] : item->key[**1**]);

swap(&z, &min->key[**0**]);

item = min; // Перемещаем указатель на лист, т.к. min - всегда лист

}

remove\_from\_node(item, k); // И удаляем требуемый ключ из листа

**return** fix(item); // Вызываем функцию для восстановления свойств дерева.

}

**struct** node \***fix**(**struct** node \*leaf)

{

**if** (leaf->size == **0** && leaf->parent == NULL) { // Случай 0, когда удаляем единственный ключ в дереве

free(leaf);

**return** NULL;

}

**if** (leaf->size != **0**) { // Случай 1, когда вершина, в которой удалили ключ, имела два ключа

**if** (leaf->parent) **return** fix(leaf->parent);

**else** **return** leaf;

}

**struct** node \*parent = leaf->parent;

**if** (parent->first->size == **2** || parent->second->size == **2** || parent->size == **2**) leaf = redistribute(leaf); // Случай 2, когда достаточно перераспределить ключи в дереве

**else** **if** (parent->size == **2** && parent->third->size == **2**) leaf = redistribute(leaf); // Аналогично

**else** leaf = merge(leaf); // Случай 3, когда нужно произвести склеивание и пройтись вверх по дереву как минимум на еще одну вершину

**return** fix(leaf);

}

**struct** node \***redistribute**(**struct** node \*leaf)

{

**struct** node \*parent = leaf->parent;

**struct** node \*first = parent->first;

**struct** node \*second = parent->second;

**struct** node \*third = parent->third;

**if** ((parent->size == **2**) && (first->size < **2**) && (second->size < **2**) && (third->size < **2**)) {

**if** (first == leaf) {

parent->first = parent->second;

parent->second = parent->third;

parent->third = NULL;

insert\_to\_node(parent->first, parent->key[**0**]);

parent->first->third = parent->first->second;

parent->first->second = parent->first->first;

**if** (leaf->first != NULL) parent->first->first = leaf->first;

**else** **if** (leaf->second != NULL) parent->first->first = leaf->second;

**if** (parent->first->first != NULL) parent->first->first->parent = parent->first;

remove\_from\_node(parent, parent->key[**0**]);

free(first);

} **else** **if** (second == leaf) {

insert\_to\_node(first, parent->key[**0**]);

remove\_from\_node(parent, parent->key[**0**]);

**if** (leaf->first != NULL) first->third = leaf->first;

**else** **if** (leaf->second != NULL) first->third = leaf->second;

**if** (first->third != NULL) first->third->parent = first;

parent->second = parent->third;

parent->third = NULL;

free(second);

} **else** **if** (third == leaf) {

insert\_to\_node(second, parent->key[**1**]);

parent->third = NULL;

remove\_from\_node(parent, parent->key[**1**]);

**if** (leaf->first != NULL) second->third = leaf->first;

**else** **if** (leaf->second != NULL) second->third = leaf->second;

**if** (second->third != NULL) second->third->parent = second;

free(third);

}

} **else** **if** ((parent->size == **2**) && ((first->size == **2**) || (second->size == **2**) || (third->size == **2**))) {

**if** (third == leaf) {

**if** (leaf->first != NULL) {

leaf->second = leaf->first;

leaf->first = NULL;

}

insert\_to\_node(leaf, parent->key[**1**]);

**if** (second->size == **2**) {

parent->key[**1**] = second->key[**1**];

remove\_from\_node(second, second->key[**1**]);

leaf->first = second->third;

second->third = NULL;

**if** (leaf->first != NULL) leaf->first->parent = leaf;

} **else** **if** (first->size == **2**) {

parent->key[**1**] = second->key[**0**];

leaf->first = second->second;

second->second = second->first;

**if** (leaf->first != NULL) leaf->first->parent = leaf;

second->key[**0**] = parent->key[**0**];

parent->key[**0**] = first->key[**1**];

remove\_from\_node(first, first->key[**1**]);

second->first = first->third;

**if** (second->first != NULL) second->first->parent = second;

first->third = NULL;

}

} **else** **if** (second == leaf) {

**if** (third->size == **2**) {

**if** (leaf->first == NULL) {

leaf->first = leaf->second;

leaf->second = NULL;

}

insert\_to\_node(second, parent->key[**1**]);

parent->key[**1**] = third->key[**0**];

remove\_from\_node(third, third->key[**0**]);

second->second = third->first;

**if** (second->second != NULL) second->second->parent = second;

third->first = third->second;

third->second = third->third;

third->third = NULL;

} **else** **if** (first->size == **2**) {

**if** (leaf->second == NULL) {

leaf->second = leaf->first;

leaf->first = NULL;

}

insert\_to\_node(second, parent->key[**0**]);

parent->key[**0**] = first->key[**1**];

remove\_from\_node(first, first->key[**1**]);

second->first = first->third;

**if** (second->first != NULL) second->first->parent = second;

first->third = NULL;

}

} **else** **if** (first == leaf) {

**if** (leaf->first == NULL) {

leaf->first = leaf->second;

leaf->second = NULL;

}

insert\_to\_node(first, parent->key[**0**]);

**if** (second->size == **2**) {

parent->key[**0**] = second->key[**0**];

remove\_from\_node(second, second->key[**0**]);

first->second = second->first;

**if** (first->second != NULL) first->second->parent = first;

second->first = second->second;

second->second = second->third;

second->third = NULL;

} **else** **if** (third->size == **2**) {

parent->key[**0**] = second->key[**0**];

second->key[**0**] = parent->key[**1**];

parent->key[**1**] = third->key[**0**];

remove\_from\_node(third, third->key[**0**]);

first->second = second->first;

**if** (first->second != NULL) first->second->parent = first;

second->first = second->second;

second->second = third->first;

**if** (second->second != NULL) second->second->parent = second;

third->first = third->second;

third->second = third->third;

third->third = NULL;

}

}

} **else** **if** (parent->size == **1**) {

insert\_to\_node(leaf, parent->key[**0**]);

**if** (first == leaf && second->size == **2**) {

parent->key[**0**] = second->key[**0**];

remove\_from\_node(second, second->key[**0**]);

**if** (leaf->first == NULL) leaf->first = leaf->second;

leaf->second = second->first;

second->first = second->second;

second->second = second->third;

second->third = NULL;

**if** (leaf->second != NULL) leaf->second->parent = leaf;

} **else** **if** (second == leaf && first->size == **2**) {

parent->key[**0**] = first->key[**1**];

remove\_from\_node(first, first->key[**1**]);

**if** (leaf->second == NULL) leaf->second = leaf->first;

leaf->first = first->third;

first->third = NULL;

**if** (leaf->first != NULL) leaf->first->parent = leaf;

}

}

**return** parent;

}

**struct** node \***merge**(**struct** node \*leaf)

{

**struct** node \*parent = leaf->parent;

**if** (parent->first == leaf) {

insert\_to\_node(parent->second, parent->key[**0**]);

parent->second->third = parent->second->second;

parent->second->second = parent->second->first;

**if** (leaf->first != NULL) parent->second->first = leaf->first;

**else** **if** (leaf->second != NULL) parent->second->first = leaf->second;

**if** (parent->second->first != NULL) parent->second->first->parent = parent->second;

remove\_from\_node(parent, parent->key[**0**]);

free(parent->first);

parent->first = NULL;

} **else** **if** (parent->second == leaf) {

insert\_to\_node(parent->first, parent->key[**0**]);

**if** (leaf->first != NULL) parent->first->third = leaf->first;

**else** **if** (leaf->second != NULL) parent->first->third = leaf->second;

**if** (parent->first->third != NULL) parent->first->third->parent = parent->first;

remove\_from\_node(parent, parent->key[**0**]);

free(parent->second);

parent->second = NULL;

}

**if** (parent->parent == NULL) {

**struct** node \*tmp = NULL;

**if** (parent->first != NULL) tmp = parent->first;

**else** tmp = parent->second;

tmp->parent = NULL;

free(parent);

**return** tmp;

}

**return** parent;

}