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: «2-3 TREE»

|  |  |
| --- | --- |
| Выполнил студент | Шиндель Эдуард Дмитриевич |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИВ-823 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу принял |  | преподаватель Кафедры ВС Д. М. Берлизов |
|  | подпись |  |

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

Новосибирск – 2019

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#__RefHeading___Toc4327_3153689509)

[1 2-3 деревья и его свойства 4](#__RefHeading___Toc4329_3153689509)

[2 Операции с 2-3 деревьями](#__RefHeading___Toc4337_3153689509) 5

[2.1 Поиск](#__RefHeading___Toc4339_3153689509) 5

[2.2 Добавление](#__RefHeading___Toc4341_3153689509) 5

[2.3 Поиск минимального ключа](#__RefHeading___Toc4341_3153689509) 9

[2.4 Удаление](#__RefHeading___Toc4341_3153689509) 9

[3 Экспериментальное исследование эффективности алгоритма](#__RefHeading___Toc4337_3153689509) 12

[ЗАКЛЮЧЕНИЕ](#__RefHeading___Toc4343_3153689509) 12

[ПРИЛОЖЕНИЕ](#__RefHeading___Toc4347_3153689509) 13

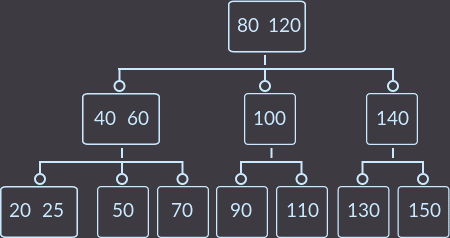
[1 Исходный код программы](#__RefHeading___Toc4349_3153689509) 13

ВВЕДЕНИЕ

Большинству программистов (и не только) известно такое дерево как бинарное дерево поиска. Деревья поиска используют тогда, когда нужно очень часто выполнять операцию поиск. В обычном дереве поиска есть очень большой недостаток: когда на вход получаем отсортированные данные, наше дерево становится обычным массивом. И тогда операция поиск будет осуществляться за такую же сложность, как и в массиве, — за O(n), где n — количество элементов в массиве. Есть несколько способов, как можно обычное дерево сделать сбалансированным (поиск имеет сложность O(log n) и 2-3 деревья это один из них.

1.  2-3 деревья и его свойства

2-3 дерево (англ. 2-3 tree) — структура данных, являющаяся B-деревом, представляющая собой сбалансированное дерево поиска, такое что из каждого узла может выходить две или три ветви, и глубина всех листьев одинакова.



Свойства 2-3 дерева:

1. Все нелистовые вершины содержат одно поле и 2 поддерева или 2 поля и 3 поддерева.
2. Все листовые вершины находятся на одном уровне (на нижнем уровне) и содержат 1 или 2 поля.
3. Все данные отсортированы (по принципу двоичного дерева поиска).
4. Нелистовые вершины содержат одно или два поля, указывающие на диапазон значений в их поддеревьях.
5. Высота 2-3 дерева O(log n), где n — количество элементов в дереве.

2. Операции с 2-3 деревьями

Я в своей курсовой работе реализовал такие операции с 2-3 деревьями как:

* Поиск элемента
* Добавление
* Поиск минимального элемента
* Удаление

2.1 Поиск

Операцию поиска можно описать при помощи простого алгоритма:

Поиск начинаем с корня дерева.

1. Ищем искомый ключ key в текущей вершине, если нашли, то возвращаем вершину, иначе
2. Если key меньше первого ключа вершины, то идем в левое поддерево и переходим к пункту 1, иначе
3. Если в дереве 1 ключ, то идем в правое поддерево и переходим к пункту 1, иначе
4. Если key меньше второго ключа вершины, то идем в среднее поддерево и переходим к пункту 1, иначе
5. Идем в правое поддерево и переходим к пункту 1.

2.2 Добавление

Также как и для поиска, операция добавления имеет определённый алгоритм.

Для того, чтобы вставить в дерево элемент с ключом key, нужно следовать пунктам.

1. Если дерево пусто, то создать новую вершину, вставить ключ и вернуть в качестве корня эту вершину, иначе
2. Если вершина является листом, то вставляем ключ в эту вершину и если получили 3 ключа в вершине, то разделяем её, иначе
3. 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
4. Смотрим, если вершина содержит только 1 ключ (является 2-вершиной), то идем в правое поддерево и переходим к пункту 2, иначе
5. 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
6. Идем в правое поддерево и переходим к пункту 2.

Для примера вставим keys = {1, 2, 3, 4, 5, 6, 7}:

При вставке key=1 мы имеем пустое дерево, а после получаем единственную вершину с единственным ключа key=1:

image

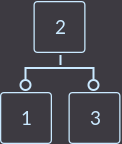
Дальше вставляем key=2:

image

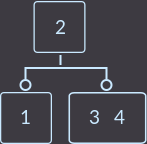
Теперь вставляем key=3 и получаем вершину, содержащую 3 ключа:

image

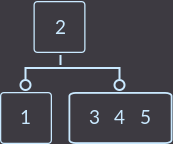
Теперь эта вершина нарушает свойства 2-3 дерева, поэтому вызовется функция разделения вершины split



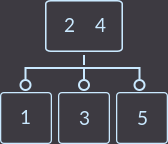
Дальше по алгоритму вставляем key=4:



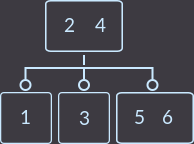
Key=5:



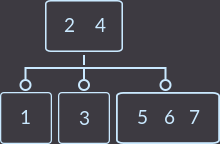
Снова нарушилось свойства 2-3 дерева, делаем разделение:



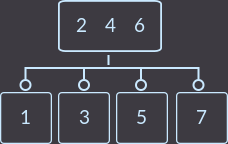
Key=6:



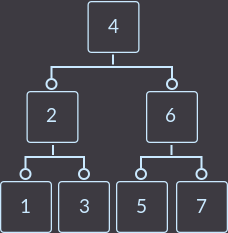
Key=7:



Теперь нам предстоит сделать два разделения, т.к. вершина, в которую вставили новый ключ теперь имеет 3 ключа (сначала разделим ее):



А теперь и корень имеет 3 вершины — разделим его и получим сбалансированное дерево, которое при таких входных данных с обычным двоичным деревом поиска мы бы не смогли получить:



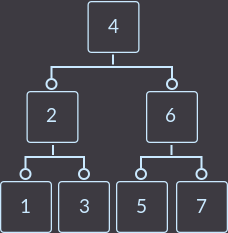
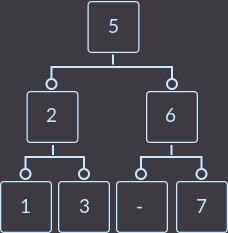
2.3 Поиск минимального элемента

Поиск минимального элемента происходит точно также, как и в обычном бинарном дереве поиска: спускаемся в самый левый нижний узел и его левый ключ и будет минимальным.

2.4 Удаление

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 нелистовой вершине. Если ключ находится в нелистовой 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.

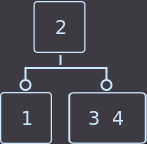
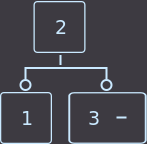
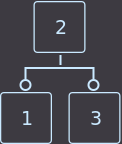
Чтобы удалить из дерева ключ key=4, для начала нужно найти эквивалентный эму элемент и поменять местами: это либо key=3, либо key=5. Так как я выбрал второй способ, то меняю ключи key=4 и key=5 местами и удаляю key=4 из листа

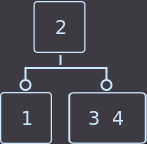
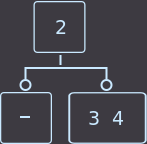
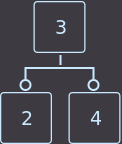
После того, как удалили ключ, у нас могут получиться концептуально 4 разные ситуации: 3 из них нарушают свойства дерева, а одна — нет. Поэтому для вершины, из которой удалили ключ, нужно вызвать функцию исправления fix(), которая вернет свойства 2-3 дерева.

Случай 0: самый простой случай: если дерево состоит из одной вершины (корень), которая имеет 1 ключ, то просто удаляем эту вершину.

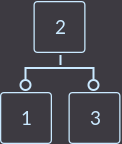
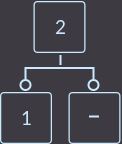
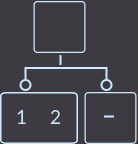
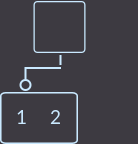
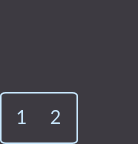
Случай 1: если нужно удалить ключ из листа, где находятся два ключа, то мы просто удаляем ключ и на этом функция удаления закончена.

Случай 2 (распределение или redistribute): 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.

Случай 3 (склеивание или merge): пожалуй, самый сложный случай, так как после склеивания всегда обязательно идти по дереву вверх и опять применять операции либо merge, либо redistribute.

[3 Экспериментальное исследование эффективности алгоритма](#__RefHeading___Toc4337_3153689509)

В практической части было проведено 2 исследования: 1) зависимость времени выполнения функции добавления в 2-3 дереве и обычном бинарном дереве поиска от количества элементов, у которых случайные значения (график 1); 2) также зависимость времени выполнения функции добавления в 2-3 дереве и бинарном дереве поиска от количества элементов, у которых значения только возрастают (худший случай бинарного дерева поиска) (график 2).

График 1. Зависимость времени выполнения функции от кол-ва элементов со случайными значениями.

График 2. Зависимость времени выполнения функции от кол-ва элементов с возрастающими значениями.

Данное исследование подтвердило, что 2-3 деревья, в отличии от бинарного дерева поиска не имеют худшего случая и сложность их операций всегда равна О(log n), где n – количество элементов.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы было реализовано 2-3 дерево и проверено наглядно, что, действительно, его высота не превышает log n, где n – это количество узлов, что делает его самобалансирующимся.

ПРИЛОЖЕНИЕ

1 Исходный код программы

#include <iostream>

#include <stdlib.h>

#include <conio.h>

#define COUNT 10

using namespace std;

struct node {

public:

int size; // количество занятых ключей

int key[3]; // ключи

node \*first; // левый дочерний узел

node \*second; // средний

node \*third; // правый

node \*fourth; // дополнительный

node \*parent; // родитель

bool find(int k) { // Этот метод возвращает true, если ключ k находится в вершине, иначе false.

for (int i = 0; i < size; ++i)

if (key[i] == k) return true;

return false;

}

void swap(int &x, int &y) {

int r = x;

x = y;

y = r;

}

void sort2(int &x, int &y) {

if (x > y) swap(x, y);

}

void sort3(int &x, int &y, int &z) {

if (x > y) swap(x, y);

if (x > z) swap(x, z);

if (y > z) swap(y, z);

}

void sort() { // Ключи в вершинах должны быть отсортированы

if (size == 1) return;

if (size == 2) sort2(key[0], key[1]);

if (size == 3) sort3(key[0], key[1], key[2]);

}

void insert\_to\_node(int k) { // Вставляем ключ k в вершину (не в дерево)

key[size] = k;

size++;

sort();

}

void remove\_from\_node(int k) { // Удаляем ключ k из вершины (не из дерева)

if (size >= 1 && key[0] == k) {

key[0] = key[1];

key[1] = key[2];

size--;

} else if (size == 2 && key[1] == k) {

key[1] = key[2];

size--;

}

}

void become\_node2(int k, node \*first\_, node \*second\_) { // Преобразовать в 2-вершину.

key[0] = k;

first = first\_;

second = second\_;

third = nullptr;

fourth = nullptr;

parent = nullptr;

size = 1;

}

bool is\_leaf() { // Является ли узел листом; проверка используется при вставке и удалении.

return (first == nullptr) && (second == nullptr) && (third == nullptr);

}

void print(node \*p, int space) {

if (!p) return;

space += COUNT;

print(p->third, space);

cout << endl;

print(p->second, space);

for (int i = COUNT; i < space; i++) cout << " ";

if (p->key[0] != 0) {

if ((p->key[0] != p->key[1]) && (p->key[1] != 0) && (p->key[0] < p->key[1])) cout << p->key[0] << " " << p->key[1] << " ";

else cout << p->key[0] << " ";

}

cout << endl;

print(p->first, space);

}

// Создавать всегда будем вершину только с одним ключом

node(int k): size(1), key{k, 0, 0}, first(nullptr), second(nullptr),

third(nullptr), fourth(nullptr), parent(nullptr) {}

node (int k, node \*first\_, node \*second\_, node \*third\_, node \*fourth\_, node \*parent\_):

size(1), key{k, 0, 0}, first(first\_), second(second\_),

third(third\_), fourth(fourth\_), parent(parent\_) {}

friend node \*split(node \*item); // Метод для разделение вершины при переполнении;

friend node \*insert(node \*p, int k); // Вставка в дерево;

friend node \*search(node \*p, int k); // Поиск в дереве;

friend node \*search\_min(node \*p); // Поиск минимального элемента в поддереве;

friend node \*merge(node \*leaf); // Слияние используется при удалении;

friend node \*redistribute(node \*leaf); // Перераспределение также используется при удалении;

friend node \*fix(node \*leaf); // Используется после удаления для возвращения свойств дереву

friend node \*remove(node \*p, int k); // удаление из дерева;

};

node \*insert(node \*p, int k) { // вставка ключа k в дерево с корнем p

if (!p || p->key[0] == 0) return new node(k); // если дерево пусто, то создаем первую 2-3-вершину (корень)

if (p->is\_leaf()) p->insert\_to\_node(k);

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

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

else insert(p->third, k);

return split(p);

}

node \*split(node \*item) {

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

node \*x = new node(item->key[0], item->first, item->second, nullptr, nullptr, item->parent);

node \*y = new node(item->key[2], item->third, item->fourth, nullptr, nullptr, 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) {

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

if (item->parent->first == item) item->parent->first = nullptr;

else if (item->parent->second == item) item->parent->second = nullptr;

else if (item->parent->third == item) item->parent->third = nullptr;

// Дальше происходит своеобразная сортировка ключей при разделении.

if (item->parent->first == nullptr) {

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 == nullptr) {

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

item->parent->third = y;

item->parent->second = x;

} else {

item->parent->fourth = y;

item->parent->third = x;

}

node \*tmp = item->parent;

delete 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.

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

return item;

}

}

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

if (!p) return nullptr;

if (p->find(k)) return p;

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

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

if (p->size == 2) return search(p->third, k);

return nullptr;

}

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

if (!p) return p;

if (!(p->first)) return p;

else return search\_min(p->first);

}

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

node \*item = search(p, k); // Ищем узел, где находится ключ k

if (!item) return p;

node \*min = nullptr;

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]);

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

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

}

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

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

}

node \*fix(node \*leaf) {

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

delete leaf;

return nullptr;

}

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;

}

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);

}

node \*redistribute(node \*leaf) {

node \*parent = leaf->parent;

node \*first = parent->first;

node \*second = parent->second;

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 = nullptr;

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

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

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

if (leaf->first != nullptr) parent->first->first = leaf->first;

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

if (parent->first->first != nullptr) parent->first->first->parent = parent->first;

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

delete first;

} else if (second == leaf) {

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

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

if (leaf->first != nullptr) first->third = leaf->first;

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

if (first->third != nullptr) first->third->parent = first;

parent->second = parent->third;

parent->third = nullptr;

delete second;

} else if (third == leaf) {

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

parent->third = nullptr;

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

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

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

if (second->third != nullptr) second->third->parent = second;

delete third;

}

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

if (third == leaf) {

if (leaf->first != nullptr) {

leaf->second = leaf->first;

leaf->first = nullptr;

}

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

if (second->size == 2) {

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

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

leaf->first = second->third;

second->third = nullptr;

if (leaf->first != nullptr) 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 != nullptr) leaf->first->parent = leaf;

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

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

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

second->first = first->third;

if (second->first != nullptr) second->first->parent = second;

first->third = nullptr;

}

} else if (second == leaf) {

if (third->size == 2) {

if (leaf->first == nullptr) {

leaf->first = leaf->second;

leaf->second = nullptr;

}

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

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

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

second->second = third->first;

if (second->second != nullptr) second->second->parent = second;

third->first = third->second;

third->second = third->third;

third->third = nullptr;

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

if (leaf->second == nullptr) {

leaf->second = leaf->first;

leaf->first = nullptr;

}

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

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

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

second->first = first->third;

if (second->first != nullptr) second->first->parent = second;

first->third = nullptr;

}

} else if (first == leaf) {

if (leaf->first == nullptr) {

leaf->first = leaf->second;

leaf->second = nullptr;

}

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

if (second->size == 2) {

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

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

first->second = second->first;

if (first->second != nullptr) first->second->parent = first;

second->first = second->second;

second->second = second->third;

second->third = nullptr;

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

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

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

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

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

first->second = second->first;

if (first->second != nullptr) first->second->parent = first;

second->first = second->second;

second->second = third->first;

if (second->second != nullptr) second->second->parent = second;

third->first = third->second;

third->second = third->third;

third->third = nullptr;

}

}

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

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

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

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

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

if (leaf->first == nullptr) leaf->first = leaf->second;

leaf->second = second->first;

second->first = second->second;

second->second = second->third;

second->third = nullptr;

if (leaf->second != nullptr) leaf->second->parent = leaf;

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

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

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

if (leaf->second == nullptr) leaf->second = leaf->first;

leaf->first = first->third;

first->third = nullptr;

if (leaf->first != nullptr) leaf->first->parent = leaf;

}

}

return parent;

}

node \*merge(node \*leaf) {

node \*parent = leaf->parent;

if (parent->first == leaf) {

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

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

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

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

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

if (parent->second->first != nullptr) parent->second->first->parent = parent->second;

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

delete parent->first;

parent->first = nullptr;

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

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

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

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

if (parent->first->third != nullptr) parent->first->third->parent = parent->first;

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

delete parent->second;

parent->second = nullptr;

}

if (parent->parent == nullptr) {

node \*tmp = nullptr;

if (parent->first != nullptr) tmp = parent->first;

else tmp = parent->second;

tmp->parent = nullptr;

delete parent;

return tmp;

}

return parent;

}

int main() {

int i = 1, key;

node \*root = new node(0);

node \*find = new node(0);

while (1) {

system("cls");

cout << "1. Insert\n";

cout << "2. Lookup\n";

cout << "3. Min\n";

cout << "4. Delete\n";

cout << "5. Print\n";

cout << "0. Quit\n";

cout << " :";

cin >> i;

switch(i) {

case 1:

system("cls");

cout << "Enter key to insert: ";

cin >> key;

root = insert(root, key);

break;

case 2:

system("cls");

cout << "Enter key to lookup: ";

cin >> key;

find = search(root, key);

if (find == nullptr) cout << "Key " << key << " not found\n";

else cout << "Key " << key << " found\n";

getch();

break;

case 3:

system("cls");

find = search\_min(root);

if (!find || find->key[0] == 0) cout << "No elements\n";

else cout << "Min = " << find->key[0];

getch();

break;

case 4:

system("cls");

cout << "Enter key to delete: ";

cin >> key;

root = remove(root, key);

break;

case 5:

system("cls");

root->print(root, 0);

getch();

break;

case 0:

return 0;

}

}

}