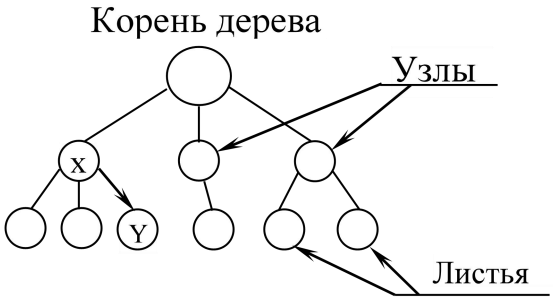
Дисциплина «Алгоритмы решения прикладных задач»
Рабочая тетрадь 1.
Структуры данных. Дерево. Двоичное дерево поиска.

Теоретический материал

Дерево

Дерево состоит из элементов, называемых узлами (вершинами). Узлы соединены между собой направленными дугами. В случае X→Y вершина X называется родителем, а Y – потомком.



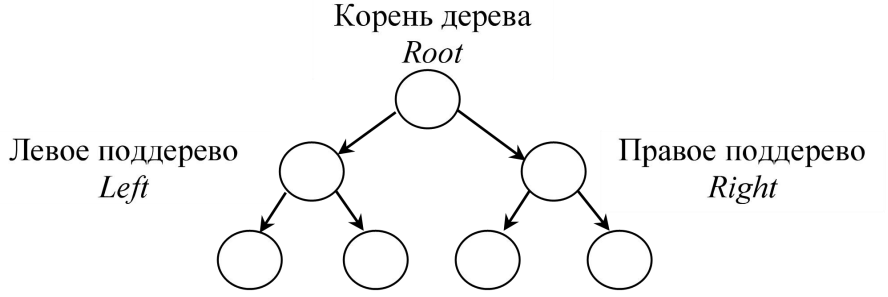
Дерево имеет единственный узел, не имеющий родителей (указателей на этот узел), который называется ***корнем***. Любой другой узел имеет ровно одного родителя, т.е. на каждый узел дерева имеется ровно один указатель.

Узел, не имеющий потомков, называется ***листом***.

Внутренний узел – это узел, не являющийся ни листом, ни корнем. ***Порядок узла*** равен количеству его узлов-потомков. ***Степень дерева*** – максимальный порядок его узлов. ***Высота (глубина) узла*** равна числу его родителей плюс один. ***Высота дерева*** – это наибольшая высота его узлов.

Двоичное дерево

Двоичное дерево – это иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками



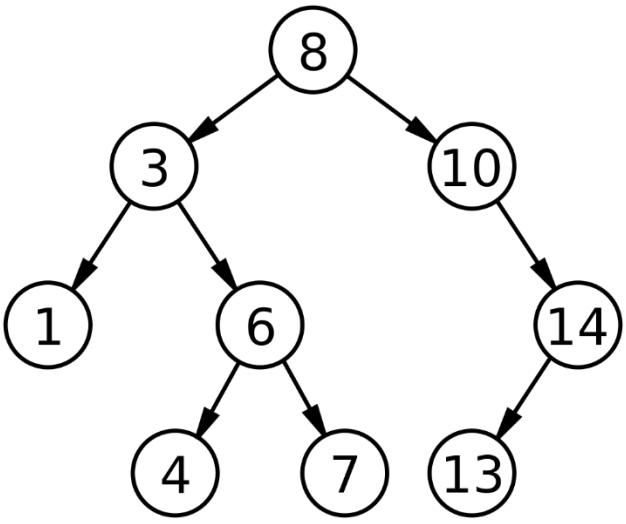
Дерево по своей организации является рекурсивной структурой данных, поскольку каждое его поддерево также является деревом. В связи с этим действия с такими структурами чаще всего описываются с помощью рекурсивных алгоритмов.

Если дерево организовано таким образом, что для каждого узла все ключи (значения узлов) его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева – больше, оно называется **двоичным деревом поиска**. Одинаковые ключи здесь не допускаются.

Двоичное дерево поиска (BST)

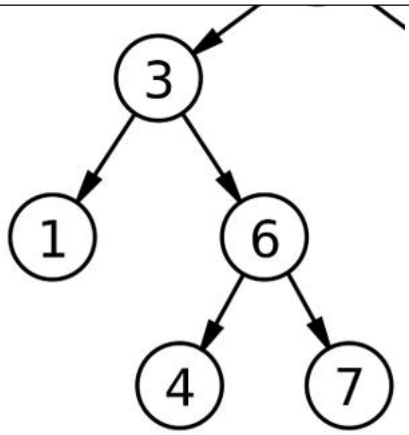
Двоичное дерево поиска (англ. *binary search tree*, BST) — двоичное дерево, для которого выполняются следующие дополнительные условия (*свойства дерева поиска*):

- оба поддерева — левое и правое — являются двоичными деревьями поиска;
- у всех узлов *левого* поддерева произвольного узла X значения ключей данных *меньше либо равны*, нежели значение ключа данных самого узла X;
- у всех узлов *правого* поддерева произвольного узла X значения ключей данных *больше*, нежели значение ключа данных самого узла X.

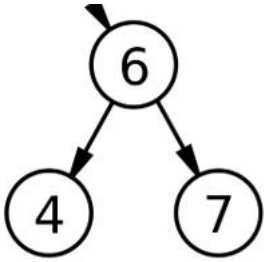


Например, чтобы добавить узел со значением «5», процедура будет следующей:

1. Сравниваем 5 и 8. 5 меньше 8, поэтому идем в левое поддерево.

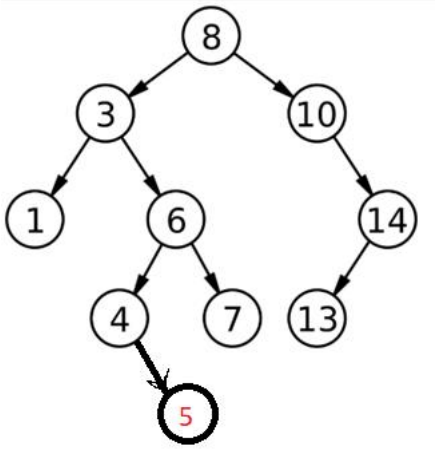


2. Сравниваем 5 и 3. 5 больше, чем 3, поэтому идем в правое поддерево.



3. Сравниваем 5 и 6. 5 меньше, чем 6, поэтому идем в левое поддерево

4. 4 – это лист. Сравниваем 5 и 4. 5 больше, чем 4. Поэтому 5 делаем правым потомком относительно 4. Итоговое дерево имеет вид:



Если в двоичное дерево поиска необходимо добавить данное, которое там уже есть, новый узел не добавляют, а увеличивают значение специальной служебной переменной – количество экземпляров узлов с определенным значением ключа (count)

Структура бинарного дерева построена из узлов. Как и в связанном списке эти узлы содержат поля данных и указатели на другие узлы в коллекции. Узел дерева содержит поле данных и два поля с указателями, которые называются левым и

правым указателями. Значение nullptr является признаком пустого поддерева.

Дерево, по сути, является рекурсивной структурой данных. В результате множество операций будут реализованы через рекурсивные функции. У таких функций будет обязательный служебный параметр - адрес узла текущего уровня.

Структура данных, описывающих дерево, имеет вид:

```
struct Node
{
int value; //Значение узла (ключ), данные любого типа
int count; //Количество экземпляров узла с данным значением (в дереве)
Node * left; //
Node * right; //
};
```

Перечислим основные рекурсивные функции для работы с деревом (названия являются условными):

• addNode() - добавление нового узла в дерево.

• printTree() - обход и печать данных дерева.

• depthTree() - вычисление глубины (высоты) дерева.

• searchNode() - поиск узла в дереве.

• delTree() - удаление дерева.

• delNode() - удаление определенного узла в дереве.

Добавление нового узла в двоичное дерево поиска

Наиболее ответственная операция: добавление в дерево нового узла. Суть алгоритма добавления в следующем: мы начинаем работу с корня всего дерева. Если дерево пустое (корень нулевой), то тогда сразу создается новый узел и его адрес возвращается в качестве корня дерева. Если корень не пустой, то по результату сравнения вставляемых данных и данных в узле дерева мы идем либо в левое, либо в правое поддерево. Далее, либо мы достигаем листа и добавляем новый узел в качестве его потомка, либо находим узел, значение данных в котором совпадает с добавляемым значением. В таком случае добавлять узел не надо, а только увеличить счетчик в найденном узле.

Обход и печать данных дерева

Существует несколько методов прохождения дерева для доступа к его элементам. К ним относятся **прямой, обратный и симметричный**. При прохождении дерева используется рекурсия, поскольку каждый узел является корнем своего поддерева. Каждый алгоритм выполняет в узле три действия: заходит в узел, рекурсивно спускается по левому и по правому поддереву. Спуск прекращается при достижении пустого поддерева (нулевой указатель).

- Порядок действий при прямом обходе:
  1. Обработка данных узла.
  2. Прохождение левого поддерева.
  3. Прохождение правого поддерева.
- Порядок действий при обратном обходе:
  1. Прохождение левого поддерева.
  2. Прохождение правого поддерева.
  3. Обработка данных узла.
- Порядок действий при симметричном обходе:
  1. Прохождение левого поддерева.
  2. Обработка данных узла.
  3. Прохождение правого поддерева.

Поиск конкретного элемента в дереве

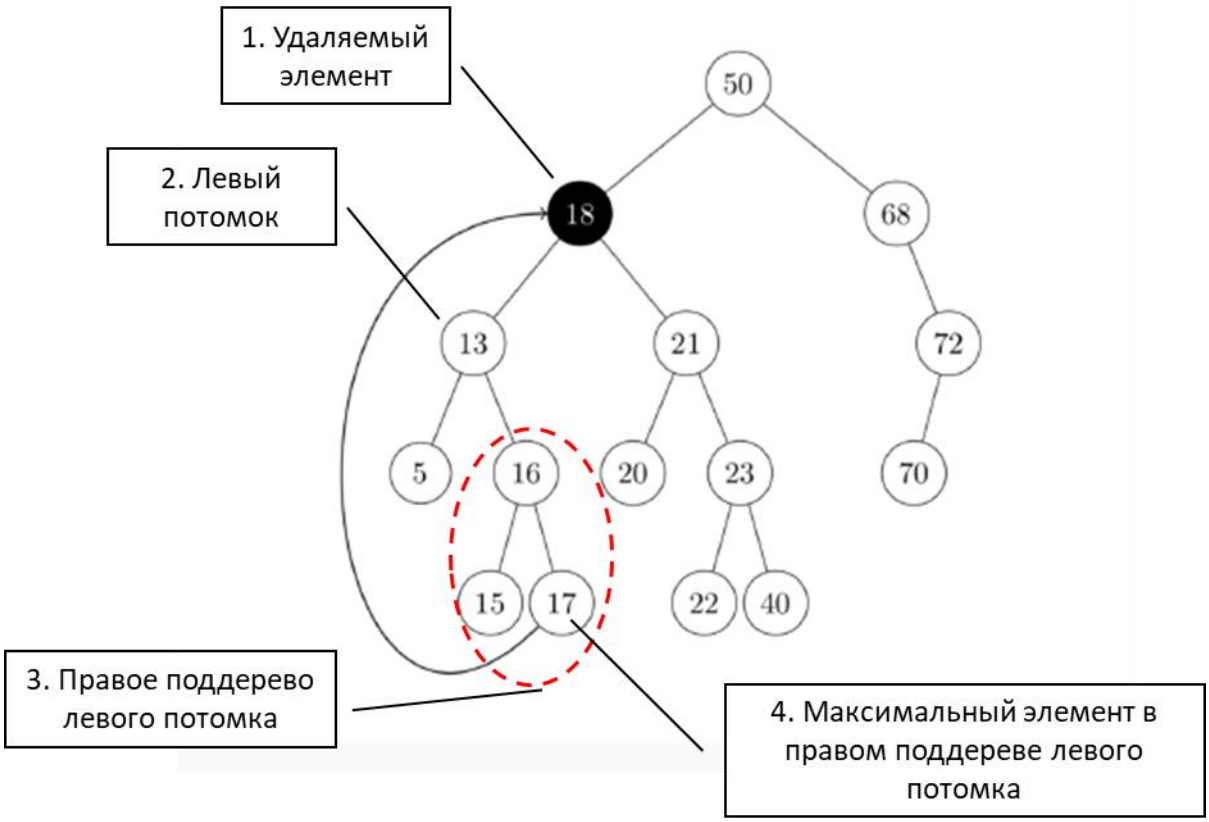
Известно, что слева от узла располагается элемент, который меньше чем текущий узел. Из чего следует, что если у узла нет левого наследника, то он является минимумом в дереве. Таким образом, можно найти минимальный элемент дерева.

Поиск нужного узла по значению выполняется следующим образом. Если искомое значение больше узла, то продолжаем поиск в правом поддереве, если меньше, то продолжаем в левом. Если узлов уже нет, то элемент не содержится в дереве.

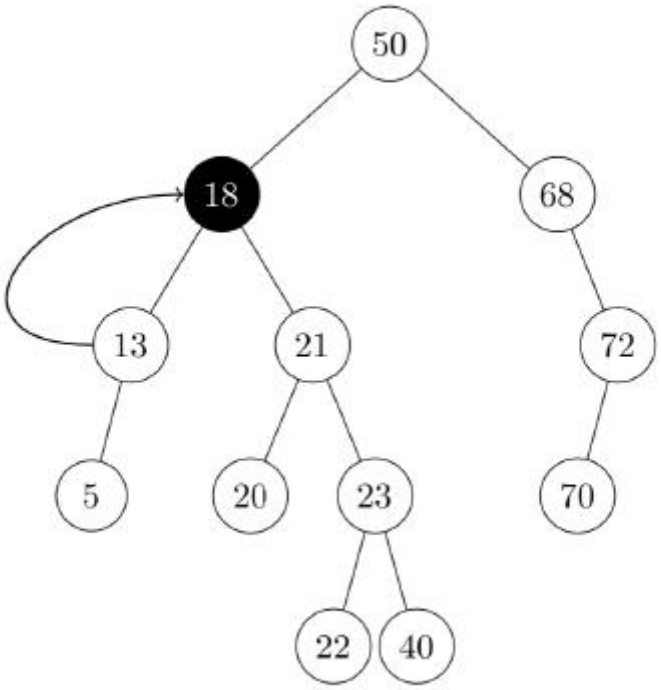
Удаление узла с определенными данными

Данная операция – наиболее сложная, поскольку при удалении произвольного узла должна сохраняться упорядоченность оставшихся элементов дерева. Существует 4 возможных ситуации при удалении узла дерева:

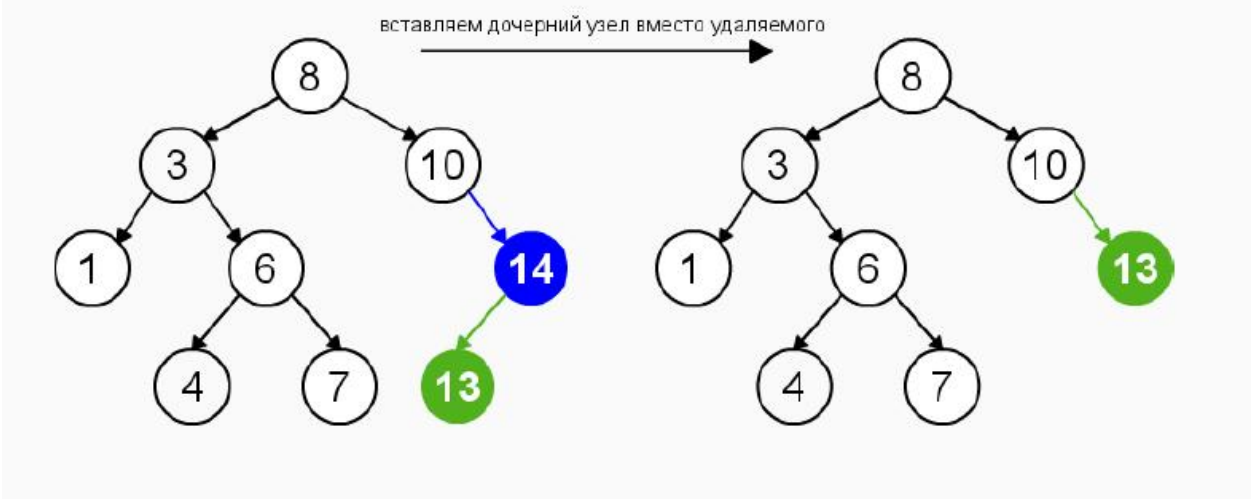
1. У удаляемого узла нет потомков. Тогда мы можем освободить память, занимаемую узлом, а у его родителя выставить nullptr в указателе на потомка.
2. Удаляемый узел имеет двух потомков, причем у левого потомка есть свое правое поддерево. В этом случае нужно найти в этом правом поддереве наибольший элемент и вставить его вместо удаляемого узла.



3. Удаляемый узел имеет двух потомков, причем у левого потомка нет правого поддерева. В этом случае элемент заменяется на корень левого поддерева:



4. Удаляемый узел имеет одного потомка (левого или правого). В этом случае мы присваиваем адрес потомка указателю нашего родителя, вместо адреса текущего узла:



Задание 1

Задача:

Создать двоичное дерево поиска (в узлах хранятся целые положительные числа). Программа должна запрашивать количество элементов дерева, далее значения, хранящиеся в элементах, создаются генератором случайных чисел. Для готового дерева реализовать операции:

а) добавление нового узла в дерево;

б) обход дерева (прямой, обратный или симметричный – по выбору) и печать элементов дерева на экран;

в) вычисление глубины (высоты) дерева;

г) поиск конкретного элемента в дереве

д*) удаление определенного узла в дереве;

Решение:

```
#include <iostream>

using namespace std;


struct node{

    int data;

    int count;

    int level;

    node *left;

    node *right;



    node(int _data) : data(_data), count(1), level(1), left(nullptr), right(nullptr) {}

};



struct pare{

    node *parent;

    node *child;

};



struct tree{

    node *root;

    int height;



    tree(): root(nullptr), height(0) {}
```

```
private:
    void add(node *data_ptr, node *root_){
        if (root_->data > data_ptr->data) {
            if (root_->left == nullptr) {
                data_ptr->level = root_->level + 1;
                root_->left = data_ptr;
            }
            else add(data_ptr, root_->left);
        }
        else if (root_->data < data_ptr->data) {
            if (root_->right == nullptr) {
                data_ptr->level = root_->level + 1;
                root_->right = data_ptr;
            }
            else add(data_ptr, root_->right);
        }
        else (root_->count)++;
    }

    void print_(node *root_){
        if (root_ != nullptr){
            cout << root_->data << ' ';
            print_(root_->left);
            print_(root_->right);
        }
    }

    node * search_(int num, node *temp){
        if (temp != nullptr){
            if (temp->data == num) return temp;
            else{
                if (temp->data > num) search_(num, temp->left);
                else search_(num, temp->right);
            }
        }
```

```
        else return nullptr;
    }

    pare searchDel(int num, node *temp, node *parent){
        if (temp != nullptr){
            if (temp->data == num) {
                pare exp;
                exp.parent = parent;
                exp.child = temp;
                return exp;
            }
            else{
                if (temp->data > num) searchDel(num, temp->left, temp);
                else searchDel(num, temp->right, temp);
            }
        }
        else{
            pare exp;
            exp.parent = nullptr;
            exp.child = nullptr;
            return exp;
        }

    }

    node * findMax(node *root_){
        if (root_->right != nullptr) findMax(root_->right);
        else return root_;
    }

public:
    void addNode(int num){
        node *data_ptr = new node(num);
        if (root == nullptr) root = data_ptr;
        else add(data_ptr, root);
        if (height < data_ptr->level) height = data_ptr->level;
```

```
}


void print() {
    print_(root);
    cout << "\n";
}


void depth(){
    cout << height << "\n";
}


node *search(int num){
    node* temp = root;
    return search_(num, temp);
}


void delet(int num){
    pare exp = searchDel(num, root, root);
    node *root_ = exp.child, *parent = exp.parent;
    if (root_->left == nullptr){
        if (root_->right == nullptr) root_ = nullptr;
        else root_=root_->right;
    }
    else{
        if (root_->right == nullptr) root_ = root_->left;
        else{
            if (root_->left->right == nullptr) root_ = root_->left;
            else{
                node *max = findMax(root_->left);
                root_->data = max->data;
                max = nullptr;
            }
        }
    }
    if (parent->data < root_->data) parent->right = root_;
    else  parent->left = root_;
```

```
    }


};


int main(){
    tree trial;
    int i, a;
    for (i=0;i<10;i++){
        a = rand()%10;
        trial.addNode(a);
    }


    cout << "Print all elements: ";
    trial.print();
    cout << "Depth: ";
    trial.depth();
    cout << "Enter number you want to find: ";
    cin >> a;
    cout << "Number adress: " << trial.search(a) << "\n";
    cout << "Enter number you want to delete: ";
    cin >> a;
    trial.delet(a);
    cout << "Print all elements after deleting: ";
    trial.print();
    return 0;
}
```

Ответ:

```
Print all elements: 1 0 7 4 2 9 8
Depth: 4
Enter number you want to find:0
 Number adress: 0x243169b5b20
Enter number you want to delete:1
 Print all elements after deleting: 0 7 4 2 9 8
```

Задание 2*

Задача:

На основе двоичного дерева поиска реализовать консольное приложение «Телефонная книга». Двоичное дерево поиска в данном случае – это хранилище записей (имя человека, его телефон) с операциями поиска и удаления записей по имени человека и операцией добавления новой записи. При этом у одного и того же человека может быть несколько номеров телефона.

Решение:

```
#include <iostream>
#include <vector>
using namespace std;

struct node{
    string name;
    int count;
    vector<string>numbers;
    node *left;
    node *right;

    node(string _name) : name(_name), count(1), left(nullptr), right(nullptr) {}
};

struct pare{
    node *parent;
    node *child;
};

struct tree{
    node *root;
    int height;

    tree(): root(nullptr), height(0) {}

private:

    void add(node *name_ptr, node *root_, string number){
        if (root_->name == name_ptr->name) {
            (root_->count)++;
            (root_->numbers).push_back(number);
        }
        else if (root_->name > name_ptr->name) {
            if (root_->left == nullptr) {
                (name_ptr->numbers).push_back(number);
                root_->left = name_ptr;
            }
            else add(name_ptr, root_->left, number);
        }
        else{
            if (root_->right == nullptr) {
                (name_ptr->numbers).push_back(number);
                root_->right = name_ptr;
            }
            else add(name_ptr, root_->right, number);
        }
    }

    void print_(node *root_){
        if (root_ != nullptr){
            cout << root_->name << "\n";
            for (int i=0;i<root_->count;i++) cout << root_->numbers[i] << ' ';
```

```
            cout << "\n";
            print_(root_->left);
            print_(root_->right);
        }
    }

    node * search_(string num, node *temp){
        if (temp != nullptr){
            if (temp->name == num) return temp;
            else{
                if (temp->name > num) return search_(num, temp->left);
                else return search_(num, temp->right);
            }
        }
        else return nullptr;
    }

    node * findMax(node *root_){
        if (root_->right != nullptr) findMax(root_->right);
        return root_;
    }

    pare searchDel(string num, node *temp, node *parent){
        if (temp != nullptr){
            if (temp->name == num) {
                pare exp;
                exp.parent = parent;
                exp.child = temp;
                return exp;
            }
            else{
                if (temp->name > num) return searchDel(num, temp->left, temp);
                else return searchDel(num, temp->right, temp);
            }
        }
        else {
            pare exp;
            exp.parent = nullptr;
            exp.child = nullptr;
            return exp;
        }
    }

public:

    void addNode(string data){
        node *name_ptr = new node(data);
        string number;
        cout << "Enter phone number: ";
        cin >> number;
        if (number == "X") return;
        if (root == nullptr) {
            root = name_ptr;
            (root->numbers).push_back(number);
        }
        else add(name_ptr, root, number);
    }

    void print() {
        print_(root);
    }

    void search(string num){
        node* temp = root;
```

```
        temp = search_(num, temp);
        if (temp == nullptr) cout << "no such name in the telephone book\n";
        for (int i=0;i<temp->count;i++) cout << temp->numbers[i] << ' ';
        cout << "\n";
    }

    void delet(string num){
        node * del;
        if (root->name == num){
            del = root;
            if (root->left == nullptr){
                if (root->right == nullptr) cout << "!" << "\n";
                else root = root->right;
            }
            else{
                if (root->right == nullptr) root = root->left;
                else{
                    if (root->left->right == nullptr) {
                        root = root->left;
                        root->right = del->right;
                    }
                    else{
                        node *max = findMax(root->left);
                        root->name = max->name;
                        max = nullptr;
                    }
                }
            }
        }
        else{
            pare exp = searchDel(num, root, root);
            node *root_ = exp.child, *parent = exp.parent;
            if (root_ == nullptr)  cout << "no such name in the telephone book\n";
            del = root_;
            if (root_->left == nullptr){
                if (root_->right == nullptr) root_ = nullptr;
                else root_=root_->right;
            }
            else{
                if (root_->right == nullptr) root_ = root_->left;
                else{
                    if (root_->left->right == nullptr) {
                        root_ = root_->left;
                        root_->right = del->right;
                    }
                    else{
                        node *max = findMax(root_->left);
                        root_->name = max->name;
                        max = nullptr;
                    }
                }
            }
            if (parent->name < num) parent->right = root_;
            else parent->left = root_;
        }
        delete del;
    }
};

void help(){
    int i;
    for (i=0;i<20;i++) cout << '-';
    cout <<
         "\nEnter\n"<<
```

```
        "H - to get the list of the commands\n"<<
        "N - to add contact\n"<<
        "S - to find a contact\n" <<
        "D - to delete contact\n" <<
        "P - to print the list of the contacts\n" <<
        "E - to exit and finish program running\n" <<
        "X - to exit the current step\n";
    for (i=0;i<20;i++) cout << '-'; cout << "\n";
}

void request_handler(char command, tree * Telephone_book){
    string a;
    switch (command)
    {
        case 'H': help(); break;
        case 'N':
            cout << "Enter a name of the contact: ";
            cin >> a;
            if (a == "X") break;
            (*Telephone_book).addNode(a);
            break;
        case 'S':
            cout << "Enter a name of the contact you want to find: ";
            cin >> a;
            if (a == "X") break;
            cout << "The list of the numbers of " << a << ": ";
            (*Telephone_book).search(a);
            break;
        case 'D':
            cout << "Enter name you want to delete: ";
            cin >> a;
            if (a == "X") break;
            (*Telephone_book).delet(a);
            cout << "\n--The contact was deleted successfully--\n";
            break;
        case 'P':
            cout << "\n--Print all contacts--\n";
            (*Telephone_book).print();
            break;
        default:
            break;
    }
}

int main(){
    tree Telephone_book;
    string a;
    char command = 'H';

    while (command != 'E'){
        request_handler(command, &Telephone_book);
        cout << "Enter command: ";
        cin >> command;
    }

    return 0;
}
```

Ответ:

```
--------------------
Enter
H - to get the list of the commands
N - to add contact
S - to find a contact
D - to delete contact
P - to print the list of the contacts
E - to exit and finish program running
X - to exit the current step
--------------------
Enter command: N
Enter a name of the contact: Jane
Enter phone number: +7(999)-999-99-99
Enter command: N
Enter a name of the contact: Andrew
Enter phone number: +3(500)-664-44-22
Enter command: N
Enter a name of the contact: Jane
Enter phone number: +7(000)-000-00-00
Enter command: N
Enter a name of the contact: John
Enter phone number: 8(800)-555-35-35
Enter command: N
Enter a name of the contact: Jane
Enter phone number: +7(888)-888-88-88
Enter command: S
Enter a name of the contact you want to find: Jane
The list of the numbers of Jane: +7(999)-999-99-99 +7(000)-000-00-00 +7(888)-888-88-88
Enter command: D
Enter name you want to delete: X
Enter command: P
```

```
Enter command: P

--Print all contacts--
Jane
+7(999)-999-99-99 +7(000)-000-00-00 +7(888)-888-88-88
Andrew
+3(500)-664-44-22
John
8(800)-555-35-35
Enter command: D
Enter name you want to delete: Andrew

--The contact was deleted successfully--
Enter command: P

--Print all contacts--
Jane
+7(999)-999-99-99 +7(000)-000-00-00 +7(888)-888-88-88
John
8(800)-555-35-35
Enter command: E
```