МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

Отчет

Практическая работа №9

Дисциплина Структуры и алгоритмы обработки данных

Тема. Алгоритмы поиска в таблице (массиве)

Выполнил студент Черномуров С. А.

Фамилия И.О.

Группа ИКБО-13-21

Номер группы

Москва 2022

Условие задания:

Разработать и реализовать алгоритмы поиска записей с заданным ключом в таблице с применением различных алгоритмов

Задание 1

Вариант №23

1. Требования к выполнению задачи:
    1. Таблица содержит записи, структура которых определена вариантом. Ключи уникальны в пределах таблицы.
    2. Разработать функцию линейного поиска (метод грубой силы).
    3. Разработать функцию поиска с барьером.
    4. Провести практическую оценку времени выполнения алгоритмов на таблицах объемом 100, 1000, 10 000 записей.
    5. Составить таблицу с указанием: времени выполнения алгоритма, его фактическую и теоретическую вычислительную сложность.
    6. Сделать выводы об эффективности алгоритмов.

Таблица 1. Сводная таблица результатов

n	**T(n)**	**$T_т$=f(C+M)**	**$T_п$=Cф+Mф**
100			
1000			
10000			
100000			
500000			

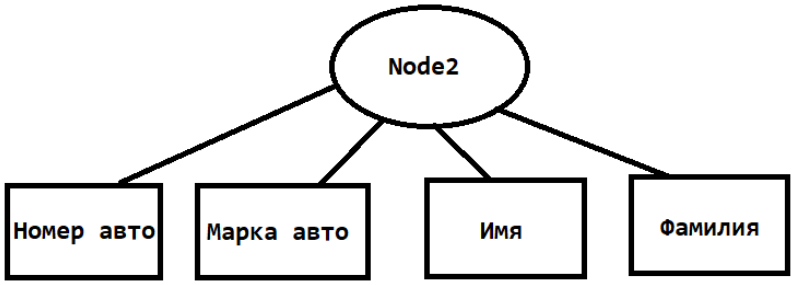
2. Постановка задачи:

    Дано. Таблица, представленная в виде вектора структур.

    Результат. Результат поиска элемента в таблице.
3. Модель решения:

    Метод линейного поиска состоит в полном переборе (с помощью цикла) всех элементов таблицы, пока не будет найден искомый элемент.

    Метод поиска с барьером является усовершенствованным линейным поиском, в цикле которого происходит на одно действие меньше, что и дает выигрыш по времени.
4. Структура записи таблицы:




```
struct Node2 {
	string number;
	string brand;
	string name;
	string surname;
};
```

Все поля в структуре записи имеют тип данных string, на который выделяется 28 байт, соответственно общий объем памяти, занимаемый структурой записи таблицы, составляет 28*4=112 байт.

5. Коды функций, реализующих алгоритмы поиска:
   Алгоритм линейного поиска:
   Предусловие. vec – вектор, содержащий записи, s – строка-ключ поиска.
   Постусловие. Кортеж, содержащий номер автомобиля, марку автомобиля, имя и фамилию владельца.
   ```
   tuple<string, string, string, string> linear_search(vector <Node2*>& vec, string s);
   ```

Тест функции:

Номер теста	Исходные данные	Ожидаемый результат
1	vec={(“num123”,”ford”,”oleg”,”tinkoff”)}, s=”num123”	(“num123”,”ford”,”oleg”,”tinkoff”)
2	vec={(“num123”,”ford”,”oleg”,”tinkoff”)}, s=”num321”	(“”,””,””,””)

```
tuple<string,string,string,string> linear_search(vector <Node2*>& vec, string s) {
	for (int i = 0; i < vec.size(); i++)
		if (vec[i]->number == s) return (make_tuple(vec[i]->number, vec[i]->brand, vec[i]->name, vec[i]->surname));
	return (make_tuple("", "", "", ""));
```

Алгоритм поиска с барьером:

Предусловие. vec – вектор, содержащий записи, s – структура, содержащая ключ поиска.
Постусловие. i – индекс искомого элемента.

```
size_t barrier_search(vector <Node2*>& vec, Node2* s);
```

Тест функции:

Номер теста	Исходные данные	Ожидаемый результат
1	vec={(“num123”,”ford”,”oleg”,”tinkoff”)}, s=(“num123”,”ford”,”oleg”,”tinkoff”)	i=0
2	vec={(“num123”,”ford”,”oleg”,”tinkoff”)}, s=(“num321”,”ford”,”oleg”,”tinkoff”)	i=-1

```
size_t barrier_search(vector <Node2*>& vec, Node2* s) {
	auto last = vec[vec.size() - 1];
	vec[vec.size() - 1] = s;
	size_t i = 0;
	for (i=0;vec[i]->number!=s->number;++i){}
	vec[vec.size() - 1] = last;
	if (i != (vec.size() - 1) || s == last) return i;
	return -1;
}
```

Время работы алгоритма линейного поиска (средний случай)

n	T(n), микросекунд	$T_т$=f(C+M), микросекунд	$T_п$=Cф+Mф, количество
100	13	6	200
1000	62	60	2000
10000	690	600	20000
100000	6900	6000	200000
500000	34343	30000	1000000

$T_т$(n)=n*1.2/20 (где 100 – количество операций, выполняемых C++ за одну микросекунду)

$T_п$(n)=2n

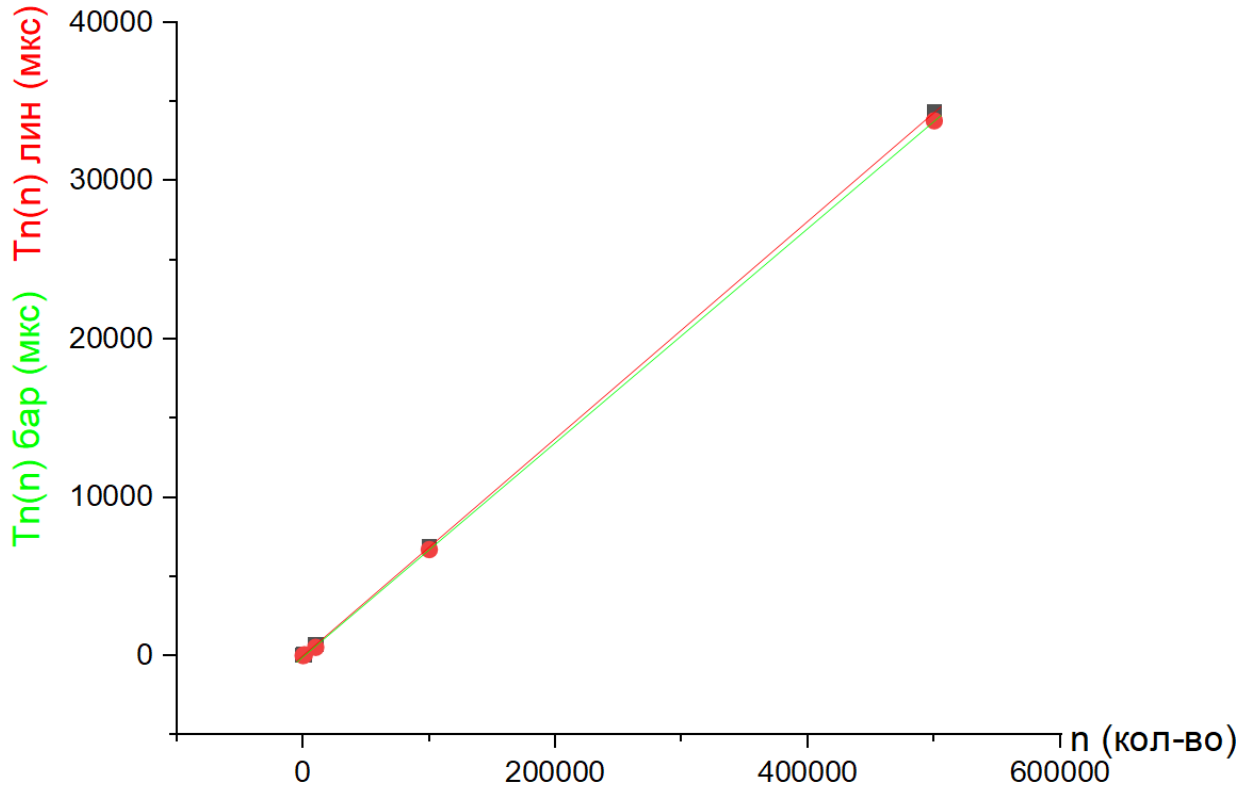
Время работы алгоритма поиска с барьером (средний случай)

n	T(n), микросекунд	$T_т$=f(C+M), микросекунд	$T_п$=Сф+Мф-количество
100	6	6	100
1000	58	60	1000
10000	527	600	10000
100000	6700	6000	100000
500000	33800	30000	500000

$T_т(n)=n*1.2/20$ (где 100 – количество операций, выполняемых C++ за одну микросекунду)

$T_п(n)=n$

Графики зависимости времени работы от количества элементов для алгоритмов линейного поиска и поиска с барьером



6. Выводы по эффективности работы алгоритмов:

Алгоритм линейной сортировки является простейшим алгоритмом поиска, который имеет простейшую реализацию и не накладывает никаких ограничений на массив данных, но за отсутствие ограничений приходится жертвовать скоростью поиска.
Алгоритм поиска с барьером является усовершенствованным вариантом линейной сортировки (на графике видно, что скорость его работы немного быстрее), но даже такое незначительное улучшение накладывает ограничение на данный массив, ведь для успешной работы алгоритмы требуется гарантия наличия искомого элемента в массиве.

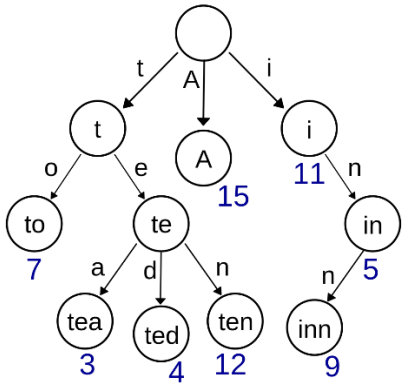
Название алгоритма	Асимптотическая сложность алгоритма		
	Наихудший случай	Наилучший случай	Средний случай
Линейный поиск	O(n)	O(1)	O(n)
Поиск с барьером	O(n)	O(1)	O(n)

Задание 2

Вариант №23

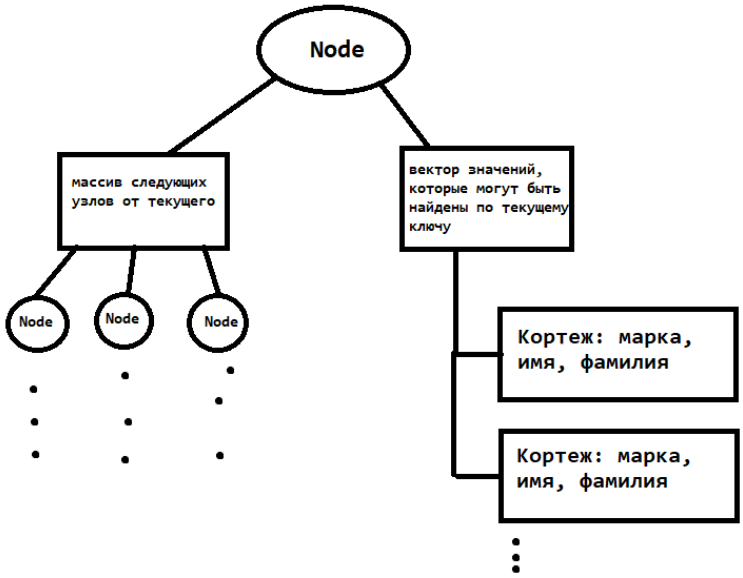
1. Требования к выполнению задачи:
    1. Таблица содержит записи, структура которых определена вариантом. Ключи уникальны в пределах таблицы.
    2. Разработать алгоритм поиска, определенный в варианте. Реализовать алгоритм функцией.
    3. Провести практическую оценку времени выполнения алгоритмов на таблицах объемом 100, 1000, 10 000 записей на случайно заполненных таблицах (худший случай). На таблицах с лучшим временем и средним.
    4. Составить таблицу с указанием: времени выполнения алгоритма, его фактическую и теоретическую вычислительную сложность.

1. Постановка задачи:
    Дано. Бор(trie).
    Результат. Результат нахождения элемента по ключу в боре.
2. Модель решения:
    Бор – структура данных для хранения набора строк, представляющая из себя подвешенное дерево с символами на рёбрах.



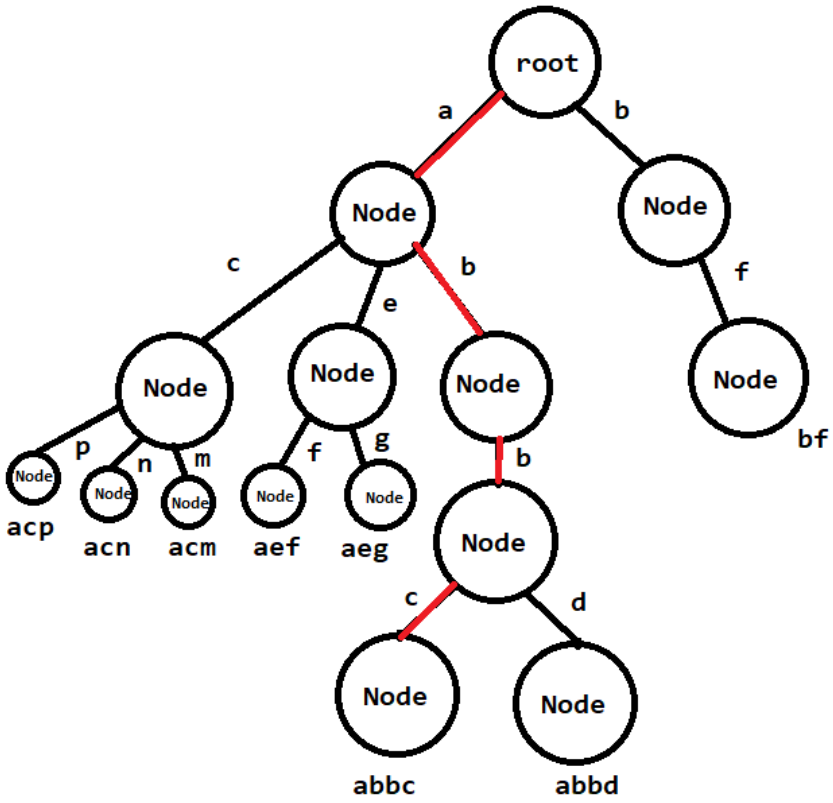
Поиск по бору осуществляется проходом по бору от корня по символам слова. Если в процессе прохода алгоритм пришел в несуществующий узел бора, то слова нет, если не пришел – то есть.

3. Структура узла бора:



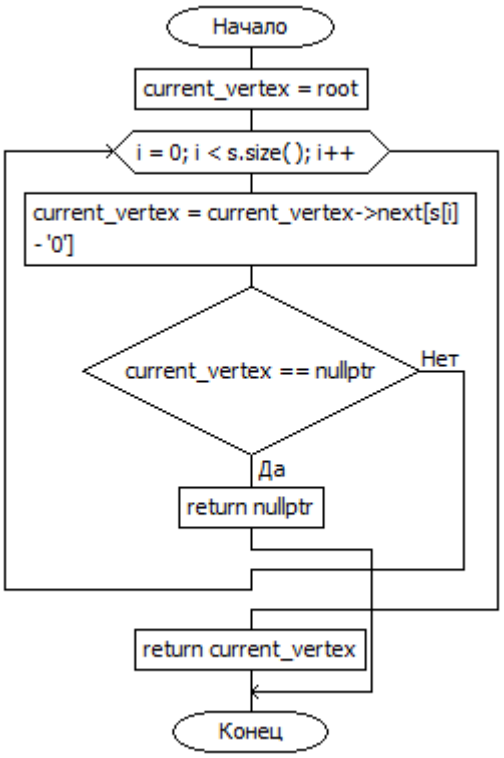

```
struct Node {
	Node* next[1000];

	vector <tuple<string, string, string>> info;

	Node() {
		for (int i = 0; i < 100; i++)
			next[i] = nullptr;

	}
};
```

4. Тестовый пример:
Нужные данные хранятся в боре, строка-ключ к ним – “abbc”. Чтобы получить данные по этому ключу, нужно попасть в узел, соответствующий этому ключу. Пройдем по бору по соответствующим ребрам(каждый проход по ребру соответствует одному символу в строке-ключе):



Проход по соответствующим ребрам бора показан красными линиями. Таким образом, мы попали в узел, доступный по ключу “abbc”, и теперь можем смотреть или изменять доступные по нему данные.

5. Блок-схема алгоритма поиска по бору:



6. **Поиск по бору:**

Предусловие. s – строка-ключ, root-указатель на корневой узел бора.

Постусловие. Ссылка на найденный по ключу узел current_vertex, в противном случае nullptr.

```
Node* is_found(string s, Node* root);
```

Тест функции:

Номер теста	Исходные данные	Ожидаемый результат
1	s=”abbc”, root=указатель на корень бора, содержащего ключ “abbc”	current_vertex от ключа abbc
2	s=”abbc”, root=указатель на корень бора, НЕ содержащего ключ “abbc”	nullptr

```
Node* is_found(string s, Node* root) {
	Node* current_vertex = root;
	for (int i = 0; i < s.size(); i++) {
		current_vertex = current_vertex->next[s[i] - '0'];
		if (current_vertex == nullptr) return nullptr;
	}
	return current_vertex;
}
```

Код программы для задания 2

```
void add_node(string s, string brand, string name, string surname, Node* root) {
	Node* current_vertex = root;

	for (int i = 0; i < s.size(); i++) {

		if (current_vertex->next[s[i] - '0'] == nullptr)
			current_vertex->next[s[i]- '0'] = new Node();

		current_vertex = current_vertex->next[s[i] - '0'];
	}

	current_vertex->info.push_back(make_tuple(brand, name, surname));
}

Node* is_found(string s, Node* root) {
	Node* current_vertex = root;
	for (int i = 0; i < s.size(); i++) {
		current_vertex = current_vertex->next[s[i] - '0'];
		if (current_vertex == nullptr) return nullptr;
	}

	return current_vertex;
}


void print_node(Node* v) {
	for (int i = 0; i < v->info.size(); i++) {
		cout << get<0>(v->info[i]) << " ";
		cout << get<1>(v->info[i]) << " ";
		cout << get<2>(v->info[i]) << "\n";
	}
}
```

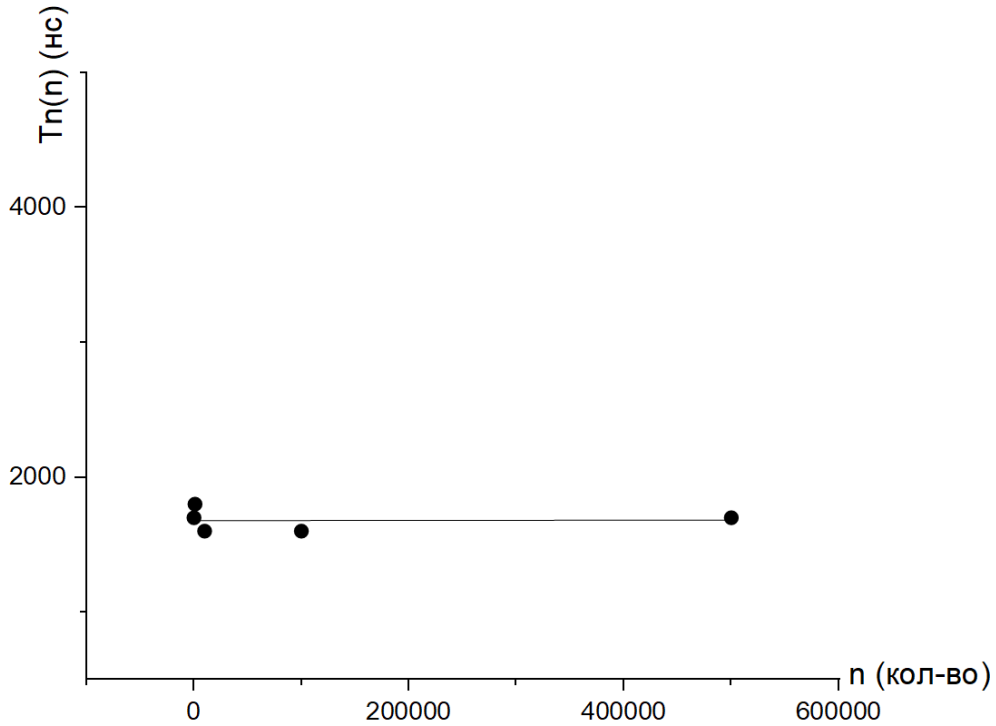
Время работы алгоритма для худшего, среднего и лучшего случаев(они все одинаковы, так как сложность поиска по бору зависит лишь от длины ключа, но не от размеров бора)

n	T(n), наносекунд	$T_т$=f(C+M), наносекунд	$T_п$=Сф+Мф-количество
100	1700	1120	8
1000	1800	1120	8
10000	1600	1120	8
100000	1600	1120	8
500000	1700	1120	8

$T_т(n)=28*4/100\ 000\ 000*10^9$, где (100 000 000 – примерное количество операций, выполняемых в C++ за секунду, 28 – размер строки в байтах, 4 – количество действий за одну итерацию цикла) (точный расчет невозможен, так как время поиска меняется при каждом перезапуске системы)

$T_п(n)=4*s$, где (s – длина ключа)

График зависимости



7. Выводы по эффективности работы алгоритма:

   Поиск по бору является очень эффективным алгоритмом поиска, который зависит не от размеров бора, а от длины ключа. Бор является достаточно легко модифицируемой структурой данных, и в зависимости от значений, хранимых в вершинах, и способа обхода с его помощью можно реализовать множество различных операций. Конкретная реализация бора варьируется от задачи к задаче.

   Бор также используется как вспомогательная структура данных для более сложных строковых алгоритмов, в частности, алгоритма Ахо-Корасик.

Название алгоритма	Асимптотическая сложность алгоритма		
	Наихудший случай	Наилучший случай	Средний случай
Поиск по бору	O(s)	O(s)	O(s)

   Асимптотическая сложность поиска по бору – O(s), где s – длина строки-ключа.

8. Сравнительный анализ алгоритмов линейного поиска, поиска с барьером и поиска по бору:

Название алгоритма	Асимптотическая сложность алгоритма		
	Наихудший случай	Наилучший случай	Средний случай
Линейный поиск	O(n)	O(1)	O(n)
Поиск с барьером	O(n)	O(1)	O(n)
Поиск по бору	O(s)	O(s)	O(s)

Полный код программы на языке C++

Файл main.cpp (основной алгоритм программы)

```
#include "functions.h"
#include <chrono>
#include <tuple>
using namespace std;
using namespace std::chrono;

int main() {
	setlocale(LC_ALL, "");
	srand(time(NULL));
	cout << "Лабораторная работа №9 ИКБО-13-21 Черномуров С.А. Вариант 23" << endl
<< endl;
	cout << "Выберите номер задания:\n1) Линейный поиск\n2) Поиск с барьером\n3)
Поиск по бору\n0) Закончить программу\n";

	int choice1;
```

```
	do {
		cin >> choice1;

		if (choice1 != 1 && choice1 != 2 && choice1 != 3 && choice1 != 0) cout <<
"Введено неверное значение, попробуйте снова.\n";
	} while (choice1 != 1 && choice1 != 2 && choice1 != 3 && choice1 != 0);

	Node* root=new Node();
	vector <Node2*> vec;

	cout << "Введите размер таблицы: ";
	int len;
	cin >> len;
	switch (choice1) {
	case 1: {
		for (int i = 1; i <= len; i++) {
			Node2* node = new Node2();
			node->number = to_string(i);
			vec.push_back(node);
		}
		auto start = steady_clock::now();
		auto t=linear_search(vec, to_string(len));
		auto end = steady_clock::now();
		cout << "Худший случай: " << duration_cast<microseconds>(end -
start).count()<<endl;

		 auto start2 = steady_clock::now();
		auto f = linear_search(vec, to_string(len/2));
		 auto end2 = steady_clock::now();
		 cout << "Средний случай: " << duration_cast<microseconds>(end2 -
start2).count()<<endl;

		 auto start3 = steady_clock::now();
		 auto fc = linear_search(vec, to_string(1));
		 auto end3 = steady_clock::now();
		 cout << "Лучший случай: " << duration_cast<microseconds>(end3 -
start3).count() << endl;
		break;
	}
	case 2:
	{
		for (int i = 1; i <= len; i++) {
			Node2* node = new Node2();
			node->number = to_string(i);
			vec.push_back(node);
		}
		Node2* node = new Node2();
		node->number = to_string(len);
		auto start = steady_clock::now();
		auto t = barrier_search(vec, node);
		auto end = steady_clock::now();
		cout << "Худший случай: " << duration_cast<microseconds>(end -
start).count() << endl;

		node = new Node2();
		node->number = to_string(len/2);
		auto start2 = steady_clock::now();
		auto f = barrier_search(vec, node);
		auto end2 = steady_clock::now();
		cout << "Средний случай: " << duration_cast<microseconds>(end2 -
start2).count() << endl;

		node->number = to_string(1);
		auto start3 = steady_clock::now();
		auto fc = barrier_search(vec, node);
```

```
            auto end3 = steady_clock::now();
            cout << "Лучший случай: " << duration_cast<microseconds>(end3 -
start3).count() << endl;
            break;
      }

      case 3: {
            for (int i = len; i <= 2 * len; i++) {
                  string s = to_string(i),r="";
                  ;
                  for (int j = 0; j < 3; j++)
                        r+=s[j];

                  add_node(r, "", "", "", root);
            }
            auto start = steady_clock::now();
            auto t = is_found(to_string(10), root);
            auto end = steady_clock::now();
            cout << "Худший случай: " << duration_cast<nanoseconds>(end -
start).count() << endl;

            /*auto start2 = steady_clock::now();
            auto f = is_found(to_string(10), root);
            auto end2 = steady_clock::now();
            cout << "Средний случай: " << duration_cast<nanoseconds>(end2 -
start2).count() << endl;

            auto start3 = steady_clock::now();
            auto fc = is_found(to_string(10), root);
            auto end3 = steady_clock::now();
            cout << "Лучший случай: " << duration_cast<nanoseconds>(end3 -
start3).count() << endl;*/
            break;
      }
      case 0:
            return 0;
      }

      main();
}
```

Файл functions.h (содержит прототипы функций)

```
#pragma once
#include <iostream>
#include <string>
#include <vector>
#include <tuple>
using namespace std;

struct Node {
      Node* next[100];

      vector <tuple<string, string, string>> info;

      Node() {
            for (int i = 0; i < 100; i++)
                  next[i] = nullptr;

      }
};

struct Node2 {
```

```
        string number;
        string brand;
        string name;
        string surname;
};

void add_node(string s, string brand, string name, string surname, Node* root);
Node* is_found(string s, Node* root);
void print_node(Node* v);

tuple<string, string, string, string> linear_search(vector <Node2*>& vec, string s);
size_t barrier_search(vector <Node2*>& vec, Node2* s);
```

Файл functions.cpp (содержит тела функций)

```
#include "functions.h"
#include <tuple>

void add_node(string s, string brand, string name, string surname, Node* root) {

        Node* current_vertex = root;

        for (int i = 0; i < s.size(); i++) {

                if (current_vertex->next[s[i] - '0'] == nullptr)
                        current_vertex->next[s[i]- '0'] = new Node();

                current_vertex = current_vertex->next[s[i] - '0'];
        }

        current_vertex->info.push_back(make_tuple(brand, name, surname));
}

Node* is_found(string s, Node* root) {
        Node* current_vertex = root;
        for (int i = 0; i < s.size(); i++) {
                current_vertex = current_vertex->next[s[i] - '0'];
                if (current_vertex == nullptr) return nullptr;
        }

        return current_vertex;
}


void print_node(Node* v) {
        for (int i = 0; i < v->info.size(); i++) {
                cout << get<0>(v->info[i]) << " ";
                cout << get<1>(v->info[i]) << " ";
                cout << get<2>(v->info[i]) << "\n";
        }
}

tuple<string,string,string,string> linear_search(vector <Node2*>& vec, string s) {
        for (int i = 0; i < vec.size(); i++)
                if (vec[i]->number == s) return (make_tuple(vec[i]->number, vec[i]-
>brand, vec[i]->name, vec[i]->surname));
        return (make_tuple("", "", "", ""));
}

size_t barrier_search(vector <Node2*>& vec, Node2* s) {
        auto last = vec[vec.size() - 1];
        vec[vec.size() - 1] = s;
        size_t i = 0;
```

```
        for (i=0;vec[i]->number!=s->number;++i){}
        vec[vec.size() - 1] = last;
        if (i != (vec.size() - 1) || s == last) return i;
        return -1;
}


//int main() {
//      root = new Node();
//
//      add_node("arab777", "ford", "Oleg", "Tinkoff");
//      add_node("arab777", "toyota", "Nikita", "4el");
//      Node* nadofind = is_found("arab777");
//      print_node(nadofind);
//
//      vector <Node2*> vec;
//
//
//      Node2* root2 = new Node2();
//      root2->number = "arab777";
//      root2->brand = "ford";
//      root2->name = "nikita";
//      root2->surname = "Tinkoff";
//      vec.push_back(root2);
//      auto t=linear_search(vec, "arab777");
//
//      cout << get<1>(t)<<" " << get<2>(t)<<" " << get<3>(t);
//
//      Node2* root3 = new Node2();
//      root2->number = "arab777";
//      root2->brand = "ford";
//      root2->name = "nikita";
//      root2->surname = "Tinkoff";
//      vec.push_back(root2);
//      auto i = barrier_search(vec, root2);
//
//      cout << vec[i]->brand << " " << vec[i]->name << " " << vec[i]->surname;
//}
```

Вывод

В ходе выполнения работы были получены навыки по применению алгоритмов поиска в таблицах данных, а также навыки по анализу эффективности этих алгоритмов:

Название алгоритма	Асимптотическая сложность алгоритма		
	Наихудший случай	Наилучший случай	Средний случай
Линейный поиск	O(n)	O(1)	O(n)
Поиск с барьером	O(n)	O(1)	O(n)
Поиск по бору	O(s)	O(s)	O(s)