Основы программирования

ФИСТ 1 курс

Власенко
Олег
Федосович

Лекция 15

Бинарное дерево

Сравнение скорости работы структур данных.

Динамические структуры данных

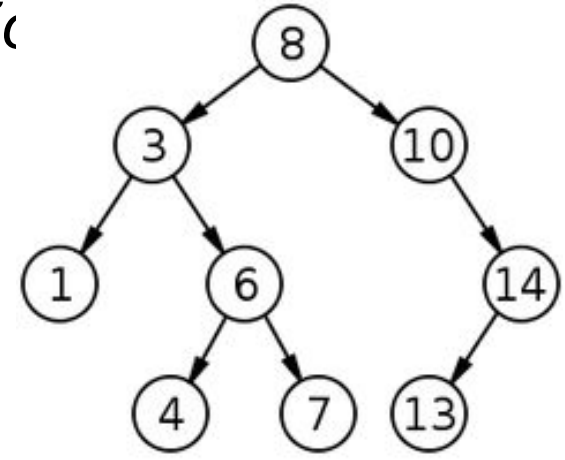
Список односвязный
Список двусвязный
Циклический список
Дерево
Двоичное дерево
Двоичное дерево поиска
Графы
…

Двоичное дерево поиска

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

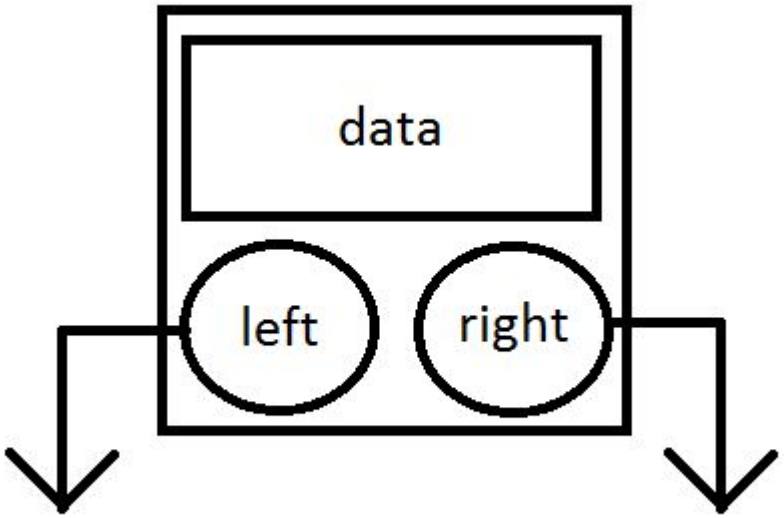
- Оба поддерева — левое и правое — являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных *меньше*, нежели значение ключа данных самого узла X.
- В то время, как значения ключей данных у всех узлов правого поддерева (того же узла X) *бо* значение ключа данных узла X.

https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE

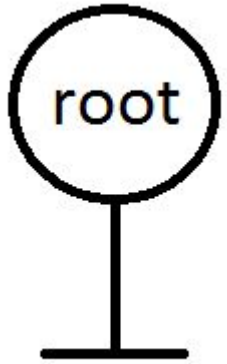


Структура узла дерева

```
struct NodeTree {
    int data;
    struct NodeTree * left;
    struct NodeTree * right;
};
```




```
struct NodeTree * root = NULL;
```



Отрабатываем навыки рисования

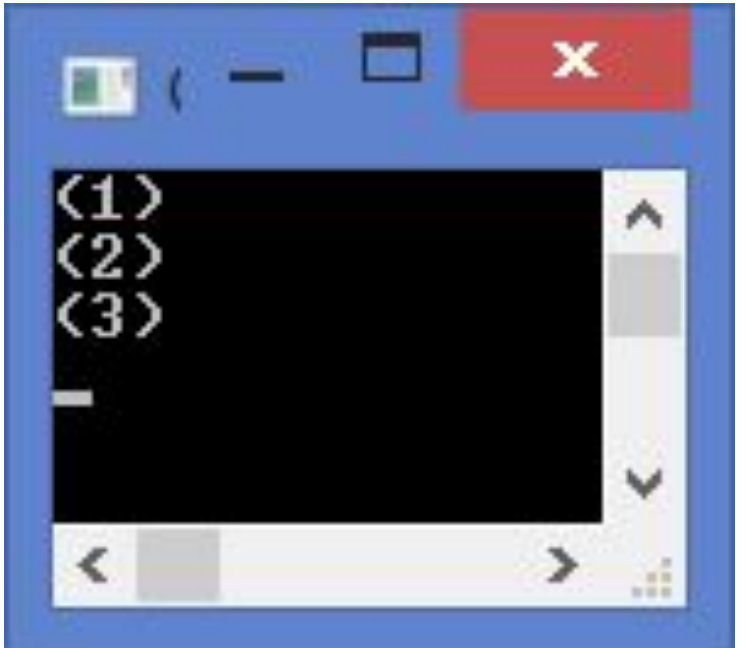
```
void main() {
    struct NodeTree node1 = { 1, NULL, NULL };
    struct NodeTree node2 = { 2, NULL, NULL };
    struct NodeTree node3 = { 3, NULL, NULL };

    root = &node2;
    node2.left = &node1;
    node2.right = &node3;

    printTree(root);
    printTreeShifted(root, 0);
}
```

Простейшая печать дерева

```
void printTree(struct NodeTree * p)
{
    if (p != NULL) {
        printTree(p->left);
        printf("(%d)\n", p->data);
        printTree(p->right);
    }
}
```



Печать дерева с отображением структуры


```
void printfShift(int shift) {
    int i;
    for (i = 0; i < shift; i++) {
        printf("  ");
    }
}
void printTreeShifted(struct NodeTree * p, int shift)
{
    if (p != NULL) {
        printTreeShifted(p->left, shift + 1);
        printfShift(shift);
        printf("(%d)\n", p->data);
        printTreeShifted(p->right, shift + 1);
    }
}
```

Добавление элемента в дерево

```
struct NodeTree * addElement(struct NodeTree *p, int value)
{
    if (p == NULL) {
        p = (struct NodeTree*)malloc(
                    sizeof(struct NodeTree));
        p->data = value;
        p->left = p->right = NULL;
    } else if (p->data == value)  {
        // НИЧЕГО НЕ ДЕЛАЕМ!!!
    } else if (value < p->data) {
        p->left = addElement(p->left, value);
    } else {
        p->right = addElement(p->right, value);
    }
    return p;
}
```

Добавление элемента в дерево

```
void main() {
    root = addElement(root, 60);
    root = addElement(root, 40);
    root = addElement(root, 20);
    root = addElement(root, 10);
    root = addElement(root, 30);
    root = addElement(root, 50);
    root = addElement(root, 90);
    root = addElement(root, 70);
    root = addElement(root, 80);

    printTreeShifted(root, 0);
}
```

Что будет выведено???

Добавление элемента в дерево

```
void main() {
    root = addElement(root, 60);
    root = addElement(root, 40);
    root = addElement(root, 20);
    root = addElement(root, 10);
    root = addElement(root, 30);
    root = addElement(root, 50);
    root = addElement(root, 90);
    root = addElement(root, 70);
    root = addElement(root, 80);

    printTreeShifted(root, 0);
}
```

Очистка дерева

```
void clearTree(struct NodeTree *p)
{
    if (p != NULL) {
        clearTree(p->left);
        clearTree(p->right);
        free(p);
    }
}


...
clearTree(root);
root = NULL;
...
```

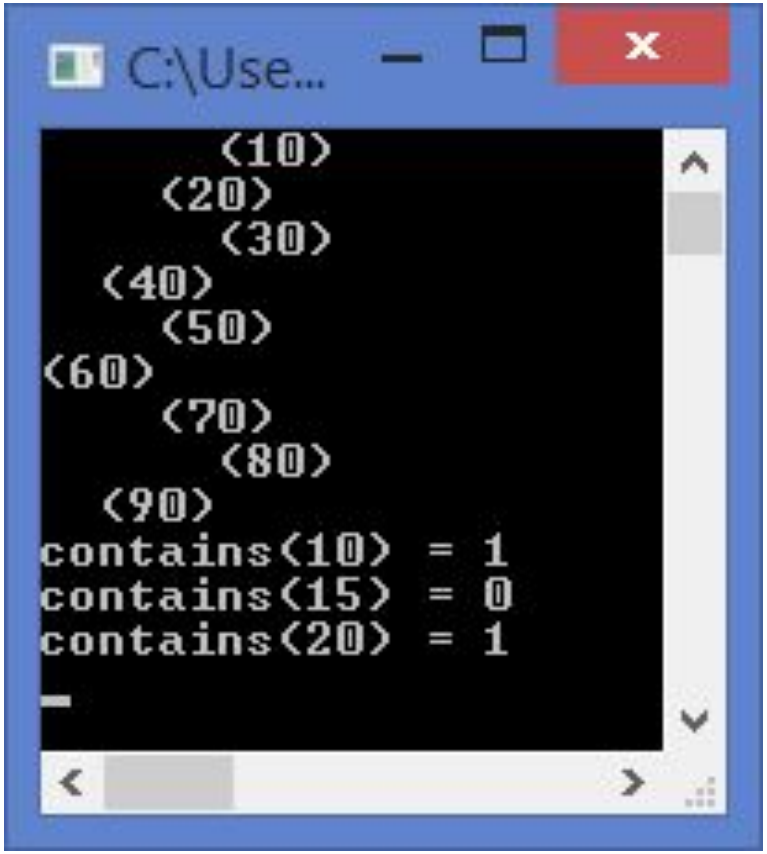
А такой элемент есть в дереве?

```
int contains(struct NodeTree * p, int value)
{
    if (p == NULL) {
        return 0;
    } else if (value == p->data) {
        return 1;
    } else if (value < p->data) {
        return contains(p->left, value);
    } else {
        return contains(p->right, value);
    }
}
```

А такой элемент есть в дереве? (2)

```
void main() {
    root = addElement(root, 60);
    root = addElement(root, 40);
    root = addElement(root, 20);
    root = addElement(root, 10);
    root = addElement(root, 30);
    root = addElement(root, 50);
    root = addElement(root, 90);
    root = addElement(root, 70);
    root = addElement(root, 80);
    printTreeShifted(root, 0);

    printf("contains(10) = %d\n", contains(root, 10));
    printf("contains(15) = %d\n", contains(root, 15));
    printf("contains(20) = %d\n", contains(root, 20));
}
```



Сравнение скорости работы структур данных

Задача конвертации текста

Задача из лекции 13.

На входе 2 файла:
Файл 1: Файл словаря – в каждой строке по 1 слову
Файл 2: Текстовый файл – большой текст (книга)
Оба файла содержат текст на английском языке.
Нужно создать третий файл - HTML файл, где есть весь текст из файла 2. Причем все слова, встречающиеся в файле 1, в файле 3 должны быть помечены как жирные.

Текст программы – решение на массиве (1)

```
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
#include <stdlib.h>
```

Текст программы – решение на массиве (2)

```
#define MAX_WORDS 10000
#define MAX_LEN 25

struct Dictionary {
    char words[MAX_WORDS][MAX_LEN];
    int cnt_words;
};

struct Dictionary * create();
void destroy(struct Dictionary * dict);
void addWord(struct Dictionary * dict, char * word);
int contains(struct Dictionary * dict, char * word);
```

Текст программы – решение на массиве (3)

```
struct Dictionary * create()
{
    struct Dictionary * dict = (struct Dictionary *)
                malloc(sizeof(struct Dictionary));
    dict->cnt_words = 0;
    return dict;
}

void destroy(struct Dictionary * dict) {
    free(dict);
}
```

Текст программы – решение на массиве (4)

```
void addWord(struct Dictionary * dict, char * word)
{
    if (dict->cnt_words < MAX_WORDS) {
        strncpy(dict->words[dict->cnt_words],
                word, MAX_LEN - 1);
        ++dict->cnt_words;
    }
}
```

Текст программы – решение на массиве (5)

```
int contains(struct Dictionary * dict, char * word)
{
    int i;
    for (i = 0; i < dict->cnt_words; ++i)
    {
        if (strcmp(word, dict->words[i]) == 0)
            return 1;
    }
    return 0;
}
```

Текст программы – решение на массиве (6)

```
int loadDictionary(struct Dictionary * dict,
                   char * filename) {
    // открыть файл
    FILE * fin;
    char s[MAX_LEN];

    fin = fopen(filename, "rt");
    if (fin == NULL)
    {
        return 0;
    }
```

Текст программы – решение на массиве (7)

```
    // в цикле для всех строк
    while (!feof(fin)) {
        // загрузить строку
        if (fgets(s, MAX_LEN - 1, fin) != NULL) {
            if (s[strlen(s) - 1] == '\n')
                s[strlen(s) - 1] = '\0';
            addWord(dict, s);
        }
    }
    // закрыть файл
    fclose(fin);
    return 1;
}
```

Текст программы – решение на массиве (8)

```
int convertTextToHtml(
        struct Dictionary * dict,
        char * text_in_filename,
        char * text_out_filename)
{
    char s[MAX_LEN];

    // открыть файлы
    FILE *fin = fopen(text_in_filename, "rt");
    if (fin == NULL)
    {
        return 0;
    }
```

Текст программы – решение на массиве (9)

```
FILE *fout = fopen(text_out_filename, "wt");
if (fout == NULL)
{
    fclose(fin);
    return 0;
}
fprintf(fout, "<!DOCTYPE html>");
fprintf(fout, "<html>");
fprintf(fout, "<head>");
fprintf(fout, "<meta http - equiv = \"Content-Type\"
content = \"text/html; charset=utf-8\" />");
fprintf(fout, "<title>HTML Document</title>");
fprintf(fout, "</head>");
fprintf(fout, "<body>");
```

Текст программы – решение на массиве (10)

```
char ch;
int is_letter = 0;
char word[81];
int word_len = 0;

while ((ch = getc(fin)) != EOF) {
    if (isalpha((unsigned char)ch)) {
        if (!is_letter) {
            word_len = 0;
        }
        is_letter = 1;
        word[word_len++] = ch;
    }
    else { //   if (!isalpha(ch)) {
```

Текст программы – решение на массиве (11)

```
else { //   if (!isalpha(ch)) {
    if (is_letter) {
        word[word_len] = '\0';
        if (contains(dict, word))
            fprintf(fout, "<b>%s</b> ", word);
        else
            fprintf(fout, "%s", word);
    }
    is_letter = 0;
    fprintf(fout, "%c", ch);
    if (ch == '\n')
        fprintf(fout, "<br>");
}
```

Текст программы – решение на массиве (12)

```
    } // while ((ch = getc(fin)) != EOF)
    fclose(fin);

    fprintf(fout, "</body>");
    fprintf(fout, "</html>");
    fclose(fout);
    return 1;
} //  convertTextToHtml- конец!!!
```

Текст программы – решение на массиве (13)

```
void main() {
    long t0, t1, t2;
    t0 = clock();
    printf("t0 = %f sec \n", t0 / (float)CLOCKS_PER_SEC);

    struct Dictionary *dict = create();
    loadDictionary(dict, "c:\\Temp\\lection16\\dict0.txt");

    t1 = clock();
    printf("t1 = %f sec \n", t1 / (float)CLOCKS_PER_SEC);

    convertTextToHtml(dict,
            "c:\\Temp\\lection16\\alice.txt",
"c:\\Temp\\lection16\\alice_out_array.html");
```

Текст программы – решение на массиве (14)

```
    destroy(dict);

    t2 = clock();
    printf("t2 = %f sec \n",
        t2 / (float)CLOCKS_PER_SEC);

    printf("Run time = t2 - t0 = %f sec \n",
        (t2 - t0) / (float)CLOCKS_PER_SEC);

}
```

Тестируем с массивом

Alice.txt – 142 800 байт
Tolkien.txt – 1 008 639 байт (Alice.txt x 7,06)
Tolkien2.txt – 5 043 195 байт (Tolkien.txt x 5)

dict0.txt – 12 слов
dict1.txt – 2960 слов (dict0 x 246,7)
dict2.txt – 9772 слов (dict1 x 3,3)

Время работы в секундах

	Dict0.txt	Dict1.txt	Dict2.txt
Alice.txt	**0,11**	**0,21**	**0,96**
Tolkien.txt	**0,57**	**1,77**	**8,53**
Tolkien2.txt	**2,62**	**8,55**	**27,21**

решение на списке (1)

```
struct Node {
    char * word;
    struct Node * next;
};

struct Dictionary {
    struct Node * first;
    int cnt_words;
};

struct Dictionary * create();
void destroy(struct Dictionary * dict);
void addWord(struct Dictionary * dict, char * word);
int contains(struct Dictionary * dict, char * word);
```

решение на списке (2)

```
struct Dictionary * create()
{
    struct Dictionary * dict = (struct Dictionary *)
                malloc(sizeof(struct Dictionary));
    dict->first = NULL;
    dict->cnt_words = 0;
    return dict;
}
```

решение на списке (3)

```
void addWord(struct Dictionary * dict, char * word)
{
    struct Node  * newNode = (struct Node*)
            malloc(sizeof(struct Node));

    newNode->next = dict->first;
    newNode->word = (char *)calloc(strlen(word) + 1,
                                   sizeof(char));
    strcpy(newNode->word, word);

    dict->cnt_words++;
    dict->first = newNode;
}
```

решение на списке (4)

```
void destroy(struct Dictionary * dict) {

    while (dict->first != NULL)
    {
        struct Node * delNode = dict->first;
        dict->first = dict->first->next;

        free(delNode->word);
        free(delNode);
    }
    free(dict);

}
```

решение на списке (5)

```
int contains(struct Dictionary * dict, char * word)
{
    struct Node * ptr = dict->first;
    while (ptr != NULL) {
        if (strcmp(ptr->word, word) == 0) {
            return 1;
        }
        ptr = ptr->next;
    }
    return 0;
}
```

решение на списке (6)

```
void main() {
	long t0, t1, t2;
	t0 = clock();
	printf("t0 = %f sec \n", t0 / (float)CLOCKS_PER_SEC);

	struct Dictionary *dict = create();
	loadDictionary(dict, "c:\\Temp\\lection16\\dict2.txt");

	t1 = clock();
	printf("t1 = %f sec \n", t1 / (float)CLOCKS_PER_SEC);

	convertTextToHtml(dict, "c:\\Temp\\lection16\\Tolkien2.txt",
		"c:\\Temp\\lection16\\Tolkien2_out_list.html");
	destroy(dict);

	t2 = clock();
	printf("t2 = %f sec \n", t2 / (float)CLOCKS_PER_SEC);
	printf("Run time = t2 - t0 = %f sec \n", (t2 - t0) / (float)CLOCKS_PER_SEC);
}
```

Тестируем со списком

Alice.txt – 142 800 байт
Tolkien.txt – 1 008 639 байт (Alice.txt x 7,06)
Tolkien2.txt – 5 043 195 байт (Tolkien.txt x 5)

dict0.txt – 12 слов
dict1.txt – 2960 слов (dict0 x 246,7)
dict2.txt – 9772 слов (dict1 x 3,3)

Время работы в секундах

	Dict0.txt	Dict1.txt	Dict2.txt
Alice.txt	**0,09**	**0,88**	**1,95**
Tolkien.txt	**0,71**	**6,29**	**12,47**
Tolkien2.txt	**3,04**	**30,51**	**63,79**

решение на дереве (1)

```
struct NodeTree {
    char * word;
    struct NodeTree * left;
    struct NodeTree * right;
};
struct Dictionary {
    struct NodeTree * root;
    int cnt_words;
};

struct Dictionary * create();
void destroy(struct Dictionary * dict);
void addWord(struct Dictionary * dict, char * word);
int contains(struct Dictionary * dict, char * word);
```

решение на дереве (2)

```
struct Dictionary * create()
{
    struct Dictionary * dict = (struct Dictionary *)
            malloc(sizeof(struct Dictionary));
    dict->root = NULL;
    dict->cnt_words = 0;
    return dict;
}
```

решение на дереве (3)

```
struct NodeTree * addElement(
                struct NodeTree *p,
                char* word)
{
    int cond;

    if (p == NULL) {
        p = (struct NodeTree*)malloc(
                sizeof(struct NodeTree));
        p->word = (char *)calloc(strlen(word) + 1,
                 sizeof(char));
        strcpy(p->word, word);
        p->left = p->right = NULL;
    }
```

решение на дереве (4)

```
    else if ((cond = strcmp(word, p->word)) == 0) {
        // вставляемое слово совпадает
        // с уже имеющимся  - ничего не делаем
    } else if (cond < 0) {
        // вставляемое слово меньше
        // корня поддерева
        p->left = addElement(p->left, word);
    } else {
        // вставляемое слово больше
        // корня поддерева
        p->right = addElement(p->right, word);
    }
    return p;
}
```

решение на дереве (5)

```
void addWord(struct Dictionary * dict, char * word)
{
    dict->root = addElement(dict->root, word);
    dict->cnt_words++;
}
```

решение на дереве (6)

```
void clearTree(struct NodeTree *p)
{
    if (p != NULL) {
        clearTree(p->left);
        clearTree(p->right);
        free(p->word);
        free(p);
    }
}

void destroy(struct Dictionary * dict) {
    clearTree(dict->root);
    free(dict);
}
```

решение на дереве (7)

```
int containElement(struct NodeTree * p, char *word)
{
    int cond;
    if (p == NULL) {
        return 0;
    }  else if ((cond = strcmp(word, p->word)) == 0) {
        return 1;
    }  else if (cond < 0) {
        return containElement(p->left, word);
    } else {
        return containElement(p->right, word);
    }
}
```

решение на дереве (8)

```
int contains(struct Dictionary * dict, char * word)
{
    return containElement(dict->root, word);
}
```

решение на дереве (9)

```
void main()  {
        long t0, t1, t2;
        t0 = clock();
        printf("t0 = %f sec \n", t0 / (float)CLOCKS_PER_SEC);

        struct Dictionary *dict = create();
        loadDictionary(dict, "c:\\Temp\\lection16\\dict1.txt");

        t1 = clock();
        printf("t1 = %f sec \n", t1 / (float)CLOCKS_PER_SEC);

        convertTextToHtml(dict, "c:\\Temp\\lection16\\Tolkien.txt",
                "c:\\Temp\\lection16\\Tolkien_out_tree.html");

        destroy(dict);

        t2 = clock();
        printf("t2 = %f sec \n", t2 / (float)CLOCKS_PER_SEC);
        printf("Run time = t2 - t0 = %f sec \n", (t2 - t0) / (float)CLOCKS_PER_SEC);
}
```

Тестируем с деревом

Alice.txt – 142 800 байт
Tolkien.txt – 1 008 639 байт (Alice.txt x 7,06)
Tolkien2.txt – 5 043 195 байт (Tolkien.txt x 5)

dict0.txt – 12 слов
dict1.txt – 2960 слов (dict0 x 246,7)
dict2.txt – 9772 слов (dict1 x 3,3)

Время работы в секундах

	Dict0.txt	Dict1.txt	Dict2.txt
Alice.txt	**0,10**	**0,14**	**0,17**
Tolkien.txt	**0,61**	**0,82**	**0,92**
Tolkien2.txt	**2,78**	**3,86**	**4,59**

Зависимость времени работы от длины текстового файла

Время работы в секундах

	Dict0.txt			Dict1.txt			Dict2.txt		
	List	Array	Tree	List	Array	Tree	List	Array	Tree
Alice.txt	**0,09**	**0,11**	**0,1**	**0,88**	**0,21**	**0,14**	**1,95**	**0,96**	**0,17**
Tolkien.txt	**0,71**	**0,57**	**0,61**	**6,29**	**1,77**	**0,82**	**12,47**	**8,53**	**0,92**
Tolkien2.txt	**3,04**	**2,62**	**2,78**	**30,51**	**8,55**	**3,86**	**63,79**	**27,21**	**4,59**

Alice.txt – 142 800 байт

Tolkien.txt – 1 008 639 байт (Alice.txt x **7,06**)

Tolkien2.txt – 5 043 195 байт (Tolkien.txt x **5**)

Во сколько раз больше время работы?

	Dict0.txt			Dict1.txt			Dict2.txt		
File2/file1	List	Array	Tree	List	Array	Tree	List	Array	Tree
T/A 7,06	**7,89**	**5,18**	**6,1**	**7,15**	**8,43**	**5,86**	**6,39**	**8,89**	**5,41**
T2/T 5	**4,28**	**4,60**	**4,56**	**4,85**	**4,83**	**4,71**	**5,12**	**3,19**	**4,99**

Вычислительная сложность алгоритма

https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C

«почистить ковёр пылесосом» требует время, линейно зависящее от его площади – O(N)

«найти имя в телефонной книге» требует всего лишь время, логарифмически зависящее от количества записей - $O(\log_2 N)$

Какая зависимость времени обработки от длины

	Dict0.txt			Dict1.txt			Dict2.txt		
File2/file1	List	Array	Tree	List	Array	Tree	List	Array	Tree
7,06	7,89	5,18	6,1	7,15	8,43	5,86	6,39	8,89	5,41
5	4,28	4,60	4,56	4,85	4,83	4,71	5,12	3,19	4,99

Вычислительная сложность алгоритма

Вопрос:

Какая зависимость времени обработки от длины файла?

	Dict0.txt			Dict1.txt			Dict2.txt		
File2/file1	List	Array	Tree	List	Array	Tree	List	Array	Tree
7,06	7,89	5,18	6,1	7,15	8,43	5,86	6,39	8,89	5,41
5	4,28	4,60	4,56	4,85	4,83	4,71	5,12	3,19	4,99

Ответ:

Линейная зависимость: O(N)

O(N) = «асимптотическая оценка сложности»

Вычислительная сложность поиска

Поиск элемента
– какая зависимость времени поиска от количества элементов в списке?
– какая зависимость времени поиска от количества элементов в массиве?
– какая зависимость времени поиска от количества элементов в дереве?

Варианты ответа:

- **O(1)**
- **O(N)**
- **$O(N^2)$**
- **O(log N)**
- **$O(2^N)$**

Вычислительная сложность поиска

- Поиск в списке: **O(N)**
- Поиск в массиве (неотсортированном) : **O(N)**
- Поиск в двоичном дереве поиска: **O(log N)**

- Поиск в отсортированном массиве (при использовании двоичного поиска) : **O(log N)**
- Поиск в хэш-таблице: **O(1)**

Источники информации

- http://c-spravochnik.ru/
- https://msdn.microsoft.com/ru-ru/default.aspx
- http://habrahabr.ru/
- https://www.google.ru/

- http://rsdn.ru/