

Язык С. Введение в алгоритмы и структуры данных.

Лекция 11 Списки. Деревья.

Сергей Леонидович Бабичев

План лекции

- 1 Абстракция *хранилище*
- 2 Структура данных «список».
- 3 Односвязные и двусвязные списки.
- 4 Деревья. Обход деревьев.

Абстракция хранилище.

Абстракция хранилище

- Хранилище содержит *ключи и значения*.
- Помимо операций создания и уничтожения хранилища реализуются операции:
 - ▶ create — создать новую пару;
 - ▶ read — найти пару по ключу;
 - ▶ update — изменить значение по ключу;
 - ▶ delete — удалить пару по ключу.

Структуры данных, реализующие эту абстракцию называются CRUD-структурами.

Структура данных «список».

Список — структура данных, которая реализует абстракции:

- `insertAfter` — добавление элемента за текущим.
- `insertBefore` — добавление элемента перед текущим.
- `insertToFront` — добавление элемента в начало списка.
- `insertToBack` — добавление элемента в конец списка
- `find` — поиск элемента
- `size` — определение количества элементов

Списки: реализация

Для реализации списков обычно требуется явное использование указателей.

```
typedef struct linkedListNode_s {  
    someType data;  
    struct linkedListNode *next;  
} linkedListNode;
```

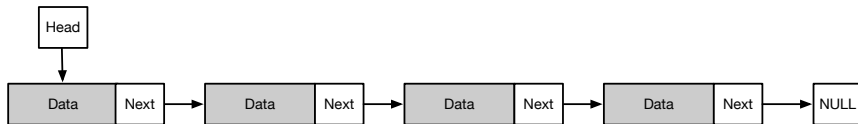
Внутренние операции создания элементов — через malloc, calloc.

```
...  
linkedListNode *item = new_linkedListNode();  
item->data = myData;  
...
```

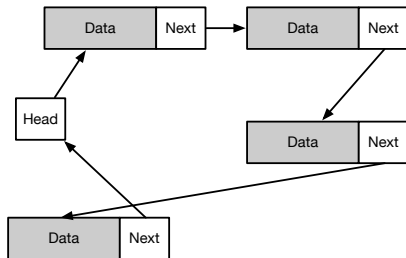
Списки: представления

Различные варианты представлений:

В линейном виде:



В виде кольца:



Списки: создание

```
typedef double myData;
```

```
linkedListNode *list_createNode(myData data) {  
    linkedListNode *ret = malloc(sizeof (linkedListNode));  
    ret->data = data;  
    ret->next = NULL;  
    return ret;  
}
```

Создание списка из одного элемента:

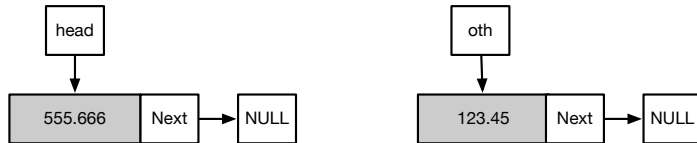
```
linkedListNode *head = list_createNode(555.666);
```



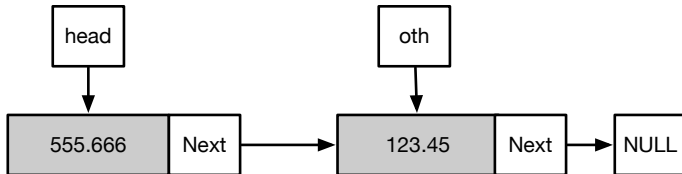
Списки: добавление

Добавление элемента в хвост списка:

```
linkedListNode *oth = list_createNode(123.45);
```

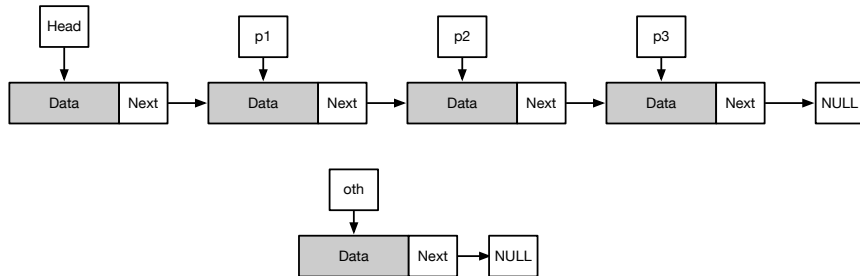


```
head->next = oth;
```



Списки: добавление

Добавление элемента в хвост списка, состоящего из нескольких элементов:

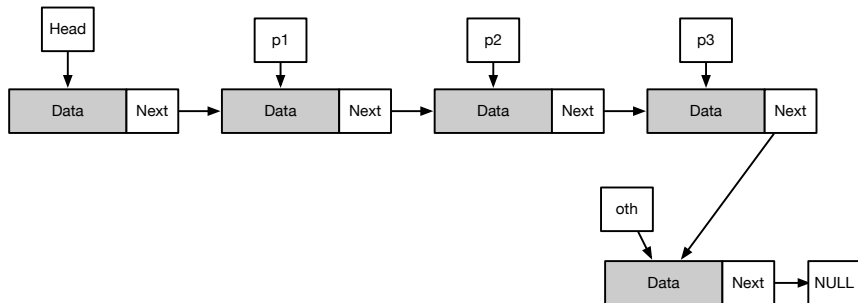


Проход по элементам до нужного (traversal, walk):

```
linkedListNode *ptr = head;
while (ptr->next != NULL) {
    ptr = ptr->next;
}
ptr->next = oth;
```

Списки: добавление

Заключительное состояние после вставки.

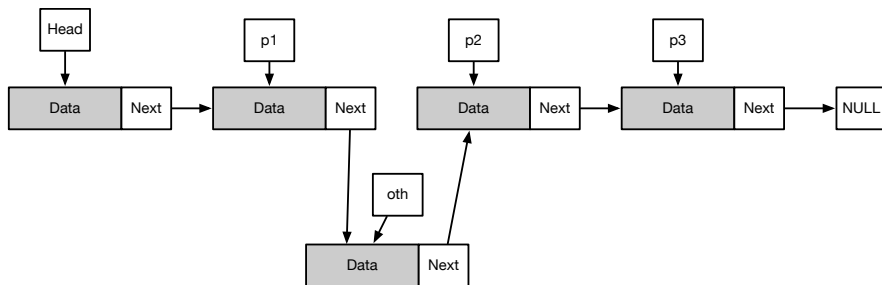


Сложность операции — $O(N)$

Списки: добавление

Вставка `insertAfter` ЗА конкретным элементом `p1` примитивна.

```
oth->next = p1->next;  
p1->next = oth;
```



Списки: добавление

Вставка *ПЕРЕД* известным элементом p2 сложнее:

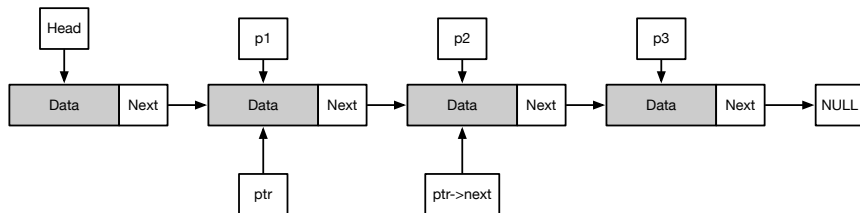
```
linkedListNode *ptr = head;
while (ptr->next != p2) {
    ptr = ptr->next;
}
oth->next = p2;
ptr->next = oth;
```

Списки: удаление

Удаление элемента p2 — непростая операция.

- Нужно найти удаляемый элемент и его предшественника:

```
linkedListNode *ptr = head;  
while (ptr->next != p2) {  
    ptr = ptr->next;  
}  
// ptr - prev to p2
```

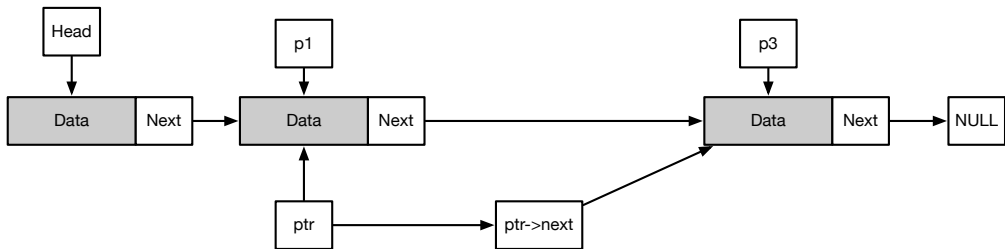


Списки: удаление

Удаление элемента из списка.

- Переместить указатели.

```
ptr->next = p2->next;  
free(p2);
```



Списки: размер

Операция size — две возможности:

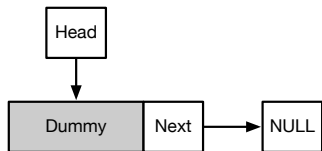
- Через операцию walk до NULL:

```
linkedListNode *ptr = head;
int size = 0;
while (ptr != NULL) {
    ptr = ptr->next;
    size++;
}
return size;
```

- Вести размер списка в структуре данных. Потребуется изменить все методы вставки/удаления.

Списки: альтернативное представление

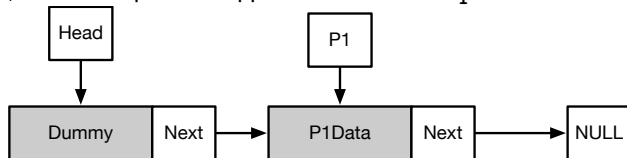
- При нашем представлении требуется всегда различать, работаем ли мы с головой списка или с другим элементом. При смене головы списка приходится заменять все указатели в программе.
- Существуют различные способы представления списков.
- Для абстрактного типа данных удобнее иметь список с неизменной головой.



- Это — пустой список, содержащий ноль элементов.

Списки: альтернативное представление

- Список, состоящий из одного элемента $p1$.



- Такое представление упрощает реализацию за счёт одного дополнительного элемента.

Списки: сложность

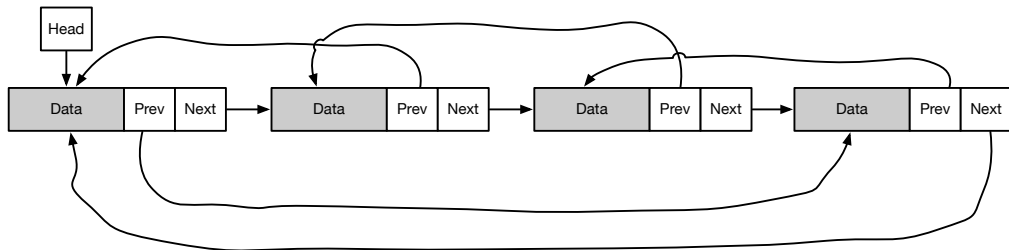
Ещё раз оценим сложность основных операций:

- Вставка элемента в голову списка — $O(1)$
- Вставка элемента в хвост списка — $O(N)$
- Поиск элемента — $O(N)$
- Удаление известного элемента — $O(N)$
- Вставка элемента ЗА известным — $O(1)$
- Вставка элемента ПЕРЕД известным — $O(N)$

Можно ли улучшить худшие случаи?

Списки: двусвязные списки

Худшие случаи можно улучшить, если заметить, что операция «слева-направо» более эффективно реализуется, чем «справа-налево» и восстановить симметрию.



Списки: двусвязные списки: сложность

Для двусвязного списка сложность такая:

- Вставка элемента в голову списка — $O(1)$
- Вставка элемента в хвост списка — $O(1)$
- Поиск элемента — $O(N)$
- Удаление известного элемента — $O(1)$
- Вставка элемента ЗА известным — $O(1)$
- Вставка элемента ПЕРЕД известным — $O(1)$

Списки: двусвязные списки: вставка

Операции вставки и удаления усложняются:

Для вставки элемента `oth` после элемента `p1`:

- 1 Подготавливаем вставляемый элемент.
- 2 Сохраняем указатель `s = p1->next`
- 3 `oth->prev = p1`
- 4 `oth->next = s`
- 5 `s->prev = oth`
- 6 `p1->next = oth`

Списки: двусвязные списки: удаление

Для удаления элемента $p1$ из списка:

- 1 Сохраняем указатель $s = p1 \rightarrow next$
- 2 $s \rightarrow prev = p1 \rightarrow prev$
- 3 $p1 \rightarrow prev \rightarrow next = s$
- 4 Освобождаем память элемента $p1$

Списки: использование

Когда используют списки? Когда нужно представлять быстро изменяющееся множество объектов.

- Пример из математического моделирования: множество машин при моделировании автодороги. Они:
 - ▶ появляются на дороге (вставка в начало списка)
 - ▶ покидают дорогу (удаление из конца списка)
 - ▶ перестраиваются с полосы на полосу (удаление из одного списка и вставка в другой)
- Пример из системного программирования: представление множества исполняющихся процессов, претендующих на процессор. Представление множества запросов ввода/вывода. Важная особенность: лёгкий одновременный доступ от множества процессоров.

Списки: использование: менеджер памяти

Одна из реализаций выделения/освобождения динамической памяти (`calloc/new/free/delete`).

- Вначале свободная память описывается пустым списком.
- Память в операционной системе выделяется *страницами*.
- При заказе памяти:
 - ▶ если есть достаточный свободный блок памяти, то он разбивается на два подблока, один из которых помечается занятым и возвращается в программу;
 - ▶ если нет достаточной свободной памяти, запрашивается несколько страниц у системы и создаётся новый элемент в конце списка (или изменяется старый).

На практике применяется несколько списков, в зависимости от размера заявки.

Связные списки как хранилище

- Сложность операций:
 - ▶ *Create* — $O(1)$
 - ▶ *Read* — $O(N)$
 - ▶ *Update* — $O(N)$
 - ▶ *Delete* — $O(N)$

Структура данных «дерево».

Деревья: особенности

Основная особенность деревьев — наличие нескольких наследников.
По максимальному числу наследников деревья делятся на

- двоичные (бинарные)
- троичные (тернарные)
- N-ричные

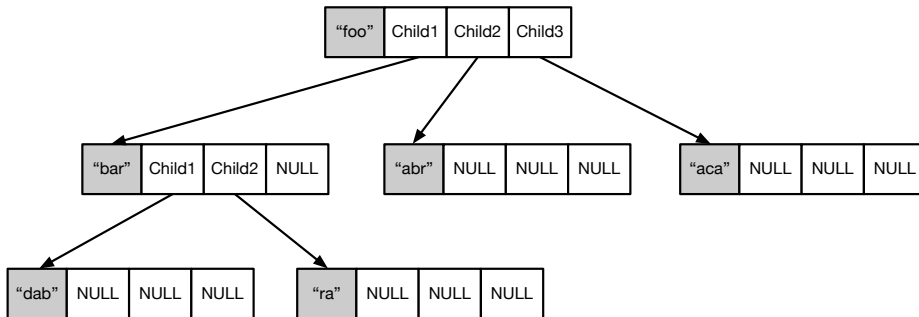
```
struct tree {  
    struct tree *children[3];  
    myType data;  
    ...  
};
```

Деревья: соглашения

- Любое N -ричное дерево может представлять деревья меньшего порядка.
- Соглашение: если наследника нет, соответствующий указатель равен `NULL`.
- Деревья 1-ричного порядка существуют (списки).

Деревья: троичное дерево

Пример дерева троичного дерева или дерева 3-порядка.



Деревья: классификация

- Условно все элементы дерева делят на две группы:
 - ▶ **Вершины**, не содержащие связей с потомками.
 - ▶ **Узлы**, содержащие связи с потомками.
- Второй вариант — все элементы дерева называют **узлами**, а **вершина** — частный случай **узла**, **терминальный узел**.
- Ещё термины:
 - ▶ **Родитель** (*parent*)
 - ▶ **Дети** (*children*)
 - ▶ **Братья** (*sibs*)
 - ▶ **Глубина** (*depth*)

$$D_{node} = D_{parent} + 1$$

Деревья: создание узла

- Добавим метод создания элемента (узла) дерева:

```
typedef struct tree_s {
    char *data;
    struct tree_s *child[3];
} tree;

tree *new_tree(const char *data) {
    tree *t = malloc(sizeof(tree));
    t->data = strdup(data);
    t->children[0]=t->children[1]=t->children[2]=NULL;
    return t;
}
```

Деревья: пример построения

- Дерево на примере строится, например, так:

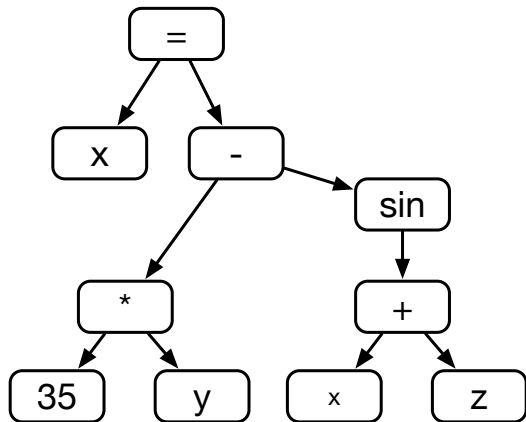
```
tree *root = new_tree("foo");  
root->child[0] = new_tree("bar");  
root->child[1] = new_tree("abr");  
root->child[2] = new_tree("aca");  
root->child[0]->child[0] = new_tree("dab");  
root->child[0]->child[1] = new_tree("ra");
```

Деревья: пример использования

Использование деревьев:

- Для представления выражений в языках программирования.

$$x = 35y - \sin(x+z);$$



Деревья: обход

- Алгоритмы работы с деревьями часто рекурсивны.
- Всего существует $6=3!$ способов обхода бинарного дерева.
- На практике применяют четыре основных варианта рекурсивного обхода:
 - ▶ Прямой
 - ▶ Симметричный
 - ▶ Обратный
 - ▶ Обратно симметричный

Деревья: обход

Бинарное дерево.

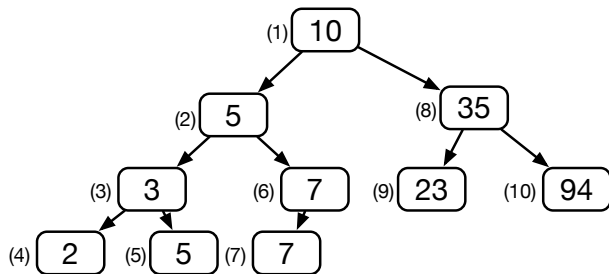
```
typedef struct tree_s {
    char *data;
    struct tree_s *left, *right;
} tree;

tree *new_tree(const char *data) {
    tree *t = malloc(sizeof(tree));
    t->data = strdup(data);
    t->left = t->right = NULL;
    return t;
};
```

Деревья: обход

Прямой способ обхода. preOrder

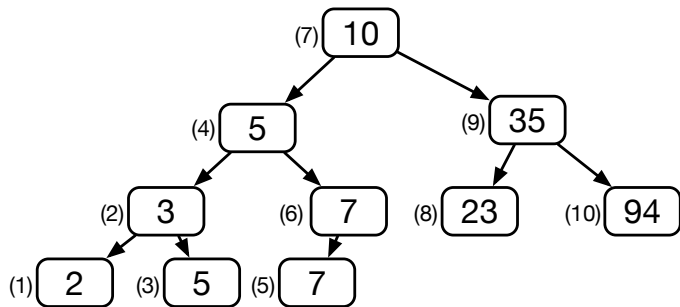
```
void walk(tree *t) {  
    work(t);  
    if (t->left != NULL) walk(t->left);  
    if (t->right != NULL) walk(t->right);  
}
```



Деревья: обход

Симметричный способ обхода. inOrder.

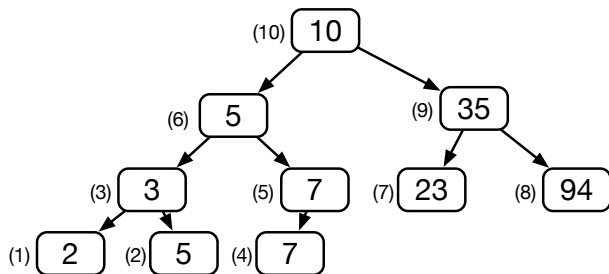
```
void walk(tree *t) {  
    if (t->left != NULL) walk(t->left);  
    work(t);  
    if (t->right != NULL) walk(t->right);  
}
```



Деревья: обход

Обратный способ обхода. postOrder

```
void walk(tree *t) {  
    if (t->left != NULL) walk(t->left);  
    if (t->right != NULL) walk(t->right);  
    work(t);  
}
```

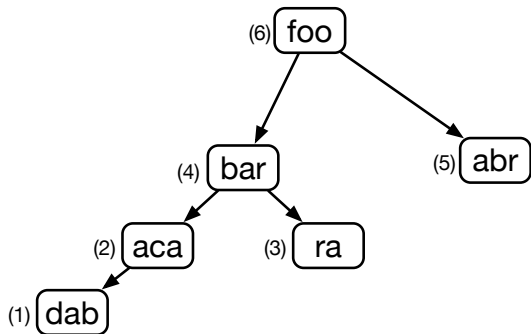


Деревья: обход

Функция обработки может быть параметром.

```
typedef void (*walkFunction)(tree *);
void walk(tree *t, walkFunction wf) {
    if (t->left != NULL) walk(t->left, wf);
    if (t->right != NULL) walk(t->right, wf);
    wf(t);
}
void printData(tree *t) {
    printf("t[%p]='%s'\n", t, t->data);
}
int main() {
    tree *root = new_tree("foo");
    root->left = new_tree("bar");
    root->right = new_tree("abr");
    root->left->left = new_tree("aca");
    root->left->left->left = new_tree("dab");
    root->left->right = new_tree("ra");
    walk(root, printData);
}
```

Деревья: обход



```
t[0x600001fe9180]='dab'  
t[0x600001fe9160]='aca'  
t[0x600001fe91a0]='ra'  
t[0x600001fe9120]='bar'  
t[0x600001fe9140]='abr'  
t[0x600001fe9100]='foo'
```

Деревья: обход

Вывод генеалогического дерева (обратно симметричный обход):

```
typedef void (*walkFunction)(tree *i, int lev);

void walk(tree *t, walkFunction wf, int lev) {
    if (t->right != NULL) walk(t->right, wf, lev+1);
    wf(t, lev);
    if (t->left != NULL) walk(t->left, wf, lev+1);
}

void printData(tree *t, int lev) {
    for (int i = 0; i < lev; i++) printf(" ");
    printf("%s\n", t->data);
}

int main() {
    ...
    walk(root, printData, 0);
}
```

Деревья: обход

Вывод программы:

```
abr
foo
  ra
bar
  aca
  dab
```

Деревья: обход

- При использовании динамических структур данных для некоторых операций важно выбрать верный обход дерева.
- Как теперь освободить заказанную память?

Деревья: обход

- Заказ памяти под поддеревья происходил динамически.
- Имелся узел, от которого шло построение дерева.
- Так как в данном дереве не хранится информация о том, кто является предком узла, корневой узел — центр всего построения.
- При операции освобождения памяти узла исказятся значения подузлов.

Деревья: динамическая память

- Создание:

```
tree *new_tree(const char *init) {  
    tree *t = malloc(sizeof(tree));  
    t->data = strdup(data);  
    t->left = t->right = NULL;  
    return t;  
};
```

- 1 Система выделяет память из кучи, достаточную для хранения всех полей структуры.
- 2 После этого выполняется инициализация полей (написанный нами код).

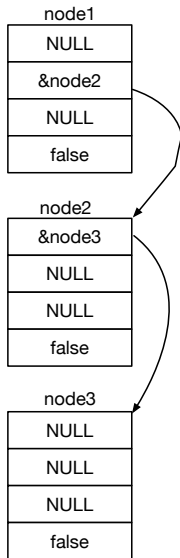
Деревья: динамическая память

- Освобождение: [плохой вариант]

```
void delete_tree(tree *t) {  
    free (t->data);  
    free(t0;  
}
```

- 1 Исполняется написанный нами код.
- 2 Система освобождает занятую память.
- 3 Обращение к освобождённой памяти приводит к ошибкам.

Деревья: динамическая память



Деревья: динамическая память

- Удаление корневого узла приводит к тому, что остальные узлы останутся недоступны.
- Такие недоступные узлы называются *висячими ссылками* (*dangling pointers*).
- Ситуация, возникшая в программе называется *утечкой памяти* (*memory leak*).
- Чтобы не было утечки памяти, удаление узлов нужно производить с терминальных.

Деревья: освобождение памяти

```
void destroy(node *n) {
    if (n->left != NULL) {
        destroy(n->left);
    }
    if (n->right != NULL) {
        destroy(n->right);
    }
    delete_tree(n);
}
```

Деревья: свойства

Свойства деревьев:

- Позволяют использовать быстро изменяющиеся структуры данных.
- Есть надежда, что операции вставки и удаления окажутся быстрыми (быстрее $O(N)$).
- Есть надежда, что операции поиска окажутся быстрыми (быстрее $O(N)$).