Динамически расширяемые массивы. Списки. Двоичные деревья поиска.

Функция realloc

```
void* realloc(void *ptr, size_t size);
```

- ptr == NULL && size != 0

  Выделение памяти (как malloc)

- ptr != NULL && size == 0
  - Освобождение памяти аналогично free(). Результат можно (но не обязательно!) передать во free().

- ptr != NULL && size != 0

  Перевыделение памяти. В худшем случае:
  - выделить новую область
  - скопировать данные из старой области в новую
  - освободить старую область

Ошибки при использовании realloc

Неправильно

```
int *p = malloc(10 * sizeof(int));

p = realloc(p, 20 * sizeof(int));
// А если realloc вернула NULL?
```

Правильно

```
int *p = malloc(10 * sizeof(int)), *tmp;

tmp = realloc(p, 20 * sizeof(int));
if (tmp)
    p = tmp;
else
    // обработка ошибки
```

Ошибки при использовании realloc

```
int* select_positive(const int *a, int n, int *k)
{
    int m = 0;
    int *p = NULL;

    for (int i = 0; i < n; i++)
        if (a[i] > 0)
        {
            m++;
            p = realloc(p, m * sizeof(int));    // !!!
            p[m-1] = a[i];
        }

    *k = m;
    return p;
}
```

Динамически расширяемые массивы

- Для уменьшение потерь при распределении памяти изменение размера должно происходить относительно крупными блоками.
- Для простоты реализации указатель на выделенную память должен храниться вместе со всей информацией, необходимой для управления динамическим массивом.

Динамически расширяемый массив

```
struct dyn_array
{
    int len;
    int allocated;
    int step;
    int *data;
};

#define INIT_SIZE 1

void init_dyn_array(struct dyn_array *d)
{
    d->len = 0;
    d->allocated = 0;
    d->step = 2;
    d->data = NULL;
}
```

Добавление элемента

```
int append(struct dyn_array *d, int item)
{
    if (!d->data)
    {
        d->data = malloc(INIT_SIZE * sizeof(int));
        if (!d->data)
            return -1;
        d->allocated = INIT_SIZE;
    }
    else
        if (d->len >= d->allocated)
        {
            int *tmp = realloc(d->data,
                        d->allocated * d->step * sizeof(int));
            if (!tmp)
                return -1;
            d->data = tmp;
            d->allocated *= d->step;
        }
    d->data[d->len] = item;
    d->len++;
    return 0;
}
```

Динамически расширяемые массивы: особенности реализации

- Удвоение размера массива при каждом вызове realloc сохраняет средние «ожидаемые» затраты на копирование элемента.
- Поскольку адрес массива может измениться, программа должна обращаться к элементами массива по индексам.
- Благодаря маленькому начальному размеру массива, программа сразу же «проверяет» код, реализующий выделение памяти.

Удаление элемента

```
int delete(struct dyn_array *d, int index)
{
    if (index < 0 || index >= d->len)
        return -1;

    memmove(d->data + index, d->data + index + 1,
                      (d->len - index - 1) * sizeof(int));
    d->len--;

    return 0;
}
```

Удаление элемента: на что обратить внимание

- Важен ли порядок элементов в массиве?
  - Нет: на место удаляемого записать последний.
  - Да: сдвинуть элементы за удаляемым вперед.
- for, memcpy или memmove?
  - for
  - memcpy НЕЛЬЗЯ (как и strcpy), memmove надежнее.
- А нужно ли удалять элементы?

Достоинства и недостатки массивов

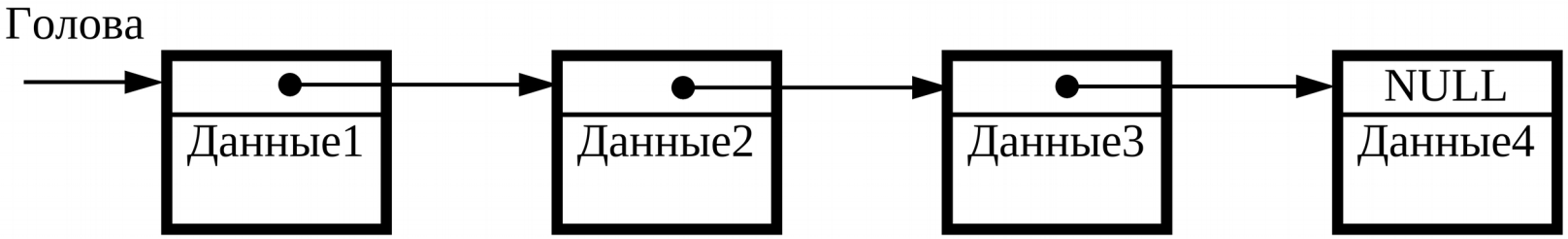
«+»

- Простота использования.
- Константное время доступа к любому элементу.
- Не тратят лишние ресурсы.
- Хорошо сочетаются с двоичным поиском.

«-»

- Хранение меняющегося набора значений.

Линейный односвязный список



Отличия списка от массива

- Размер массива фиксирован, а списка нет.
- Списки можно переформировывать, изменяя несколько указателей.
- При удалении или вставки нового элемента в список адрес остальных не меняется.

Элемент списка

```
struct person
{
    char *name;
    int born_year;

    struct person *next;
};

struct person* create_person(char *name, int born_year)
{
    struct person *pers = malloc(sizeof(struct person));

    if (pers)
    {
        pers->name = name;
        pers->born_year = born_year;
        pers->next = NULL;
    }

    return pers;
}
```

Добавление элемента в список

```
struct person* add_front(struct person *head,
                                    struct person *pers)
{
    pers->next = head;
    return pers;
}
```

NB: функции, изменяющие список, должны возвращать указатель на новый первый элемент.

```
head = add_front(head, pers);
```

Добавление элемента в список

```
struct person* add_end(struct person *head,
                                        struct person *pers)
{
    struct person *cur = head;

    if (!head)
        return pers;
    for ( ; cur->next; cur = cur->next)
        ;
    cur->next = pers;
    return head;
}
```

Добавление элемента в конец нашего простого списка – операция порядка O(N). Чтобы добиться времени O(1), можно завести отдельный указатель на конец списка.

Поиск элемента в списке

```
struct person* lookup(struct person *head,
                                        const char *name)
{
    for ( ; head; head = head->next)
        if (strcmp(head->name, name) == 0)
            return head;

    return NULL;
}
```

Поиск занимает время порядка O(N) и эту оценку не улучшить.

Обработка всех элементов списка

```
void apply(struct person *head,
             void (*f)(struct person*, void*),
                                        void* arg)
{
    for ( ; head; head = head->next)
        f(head, arg);
}
```

- **head**: список
- **f**: указатель на функцию, которая применяется к каждому элементу списка
- **arg**: аргумент функции f

Обработка всех элементов списка

```
// печать информации из элемента списка
void print_person(struct person *pers, void *arg)
{
    char *fmt = arg;
    printf(fmt, pers->name, pers->born_year);
}

// apply(l1, print_person, "l1: %s %d\n");

// подсчет количества элементов списка
void count(struct person *pers, void *arg)
{
    int *counter = arg;
    (*counter)++;
}

// apply(l2, count, &n); // где int n = 0;
```

Освобождение списка

```
void free_all(struct person *head)
{
    struct person *next;

    for ( ; head; head = next)
    {
        next = head->next;
        free(head);
    }
}
```

```
for ( ; head; head = head->next)
    free(head);
```

Наша функция free_all не освобождает память из поля name (см. create_person).

Удаление элемента по имени

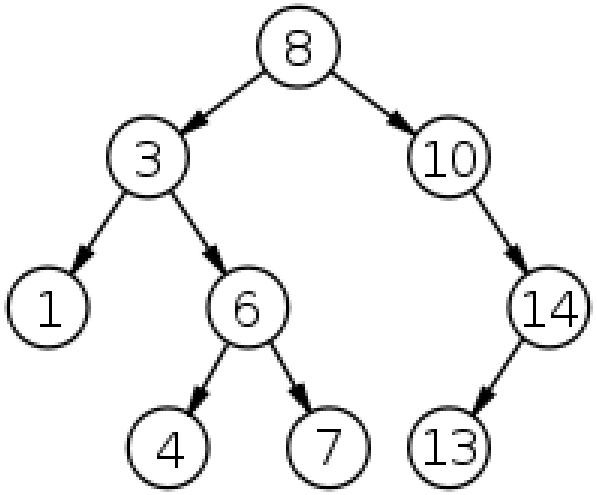
```
struct person* del_by_name(struct person *head,
                                        const char *name)
{
    struct person *cur, *prev = NULL;

    for (cur = head; cur; cur = cur->next)
    {
        if (strcmp(cur->name, name) == 0)
        {
            if (prev)
                prev->next = cur->next;
            else
                head = cur->next;
            free(cur);
            return  head;
        }
        prev = cur;
    }

    return NULL;
}
```

Списки: дальнейшее развитие

- Представление элемента списка
  - Универсальный элемент (void*).
- Двусвязные списки
  - Требует больше ресурсов.
  - Поиск последнего и удаление текущего – операции порядка O(1).

Двоичное дерево поиска

- *Дерево* - это связный ациклический граф.
- *Двоичным деревом поиска* называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (назовём их левым и правым), и все вершины, кроме корня, имеют родителя.



Элемент дерева

```
struct tree_node
{
    const char *name;

    // меньшие
    struct tree_node *left;
    // большие
    struct tree_node *right;
};

struct tree_node* create_node(const char *name)
{
    struct tree_node *node = malloc(sizeof(struct tree_node));
    if (node)
    {
        node->name = name;
        node->left = NULL;
        node->right = NULL;
    }

    return node;
}
```

Добавление элемента в дерево

```
struct tree_node* insert(struct tree_node *tree,
                                        struct tree_node *node)
{
    int cmp;

    if (tree == NULL)
        return node;

    cmp = strcmp(node->name, tree->name);
    if (cmp == 0)
        assert(1);
    else if (cmp < 0)
        tree->left = insert(tree->left, node);
    else
        tree->right = insert(tree->right, node);

    return tree;
}
```

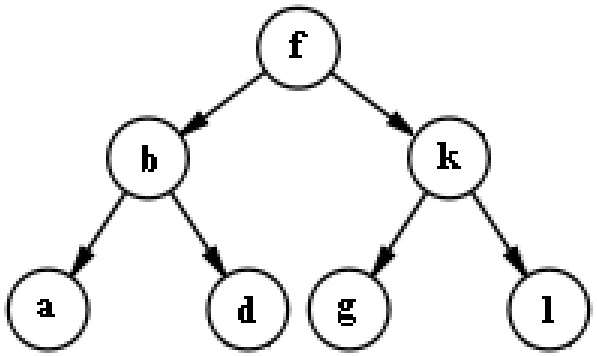
Поиск в дереве (1)

```
struct tree_node* lookup_1(struct tree_node *tree,
                                                const char *name)
{
    int cmp;

    if (tree == NULL)
        return NULL;

    cmp = strcmp(name, tree->name);
    if (cmp == 0)
        return tree;
    else if (cmp < 0)
        return lookup_1(tree->left, name);
    else
        return lookup_1(tree->right, name);
}
```

Поиск в дереве (2)

```
struct tree_node* lookup_2(struct tree_node *tree,
                                                const char *name)
{
    int cmp;

    while (tree != NULL)
    {
        cmp = strcmp(name, tree->name);
        if (cmp == 0)
            return tree;
        else if (cmp < 0)
            tree = tree->left;
        else
            tree = tree->right;
    }

    return NULL;
}
```

Обход дерева



- Прямой (pre-order)
  - **f b a d k g l**
- Фланговый или поперечный (in-order)
  - **a b d f g k l**
- Обратный (post-order)
  - **a d b g l k f**

Обход дерева

```
void apply(struct tree_node *tree,
                  void (*f)(struct tree_node*, void*),
                                                  void *arg)
{
    if (tree == NULL)
        return;

    // pre-order
    // f(tree, arg);
    apply(tree->left, f, arg);
    // in-order
    f(tree, arg);
    apply(tree->right, f, arg);
    // post-order
    // f(tree, arg);
}
```

DOT

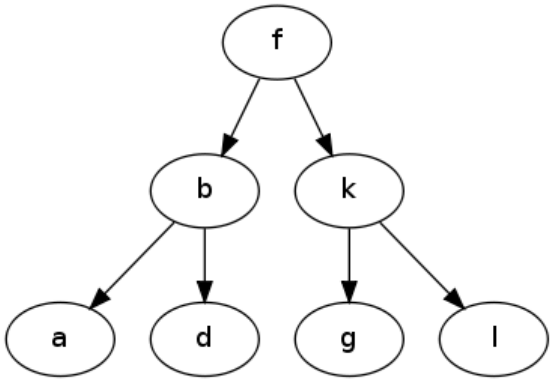
- DOT — язык описания графов.
- Граф, описанный на языке DOT, обычно представляет собой текстовый файл с расширением .gv в понятном для человека и обрабатывающей программы формате.
- В графическом виде графы, описанные на языке DOT, представляются с помощью специальных программ, например Graphviz.

Wiki (c)

DOT

```
// Описание дерева на DOT
digraph test_tree {
f -> b;
f -> k;
b -> a;
b -> d;
k -> g;
k -> l;
}
```

```
// Оформление на странице Trac
{{{
#!graphviz
digraph test_tree {
f -> b;
f -> k;
b -> a;
b -> d;
k -> g;
k -> l;
}
}}}
```



Edit this page | Attach file | Rename page

trac POWERED | Powered by Trac 0.12.2 By Edgewall Software.

DOT

```
void to_dot(struct tree_node *tree, void *param)
{
    FILE *f = param;

    if (tree->left)
        fprintf(f, "%s -> %s;\n", tree->name, tree->left->name);

    if (tree->right)
        fprintf(f, "%s -> %s;\n", tree->name, tree->right->name);
}

void export_to_dot(FILE *f, const char *tree_name,
                                        struct tree_node *tree)
{
    fprintf(f, "digraph %s {\n", tree_name);

    apply_pre(tree, to_dot, f);

    fprintf(f, "}\n");
}
```