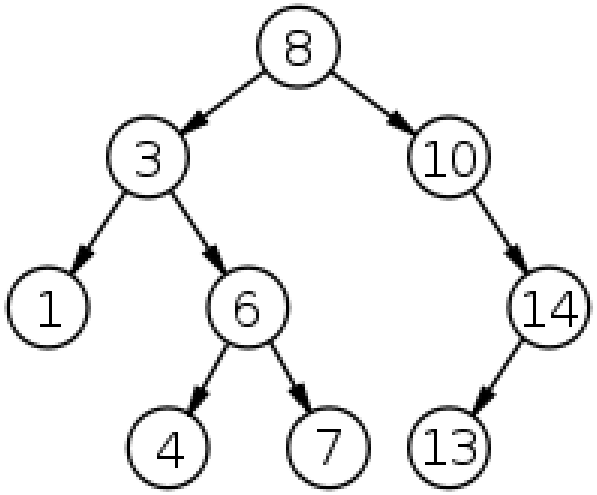
Двоичные деревья поиска

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.



Двоичное дерево поиска

Базовые операции

- Добавление узла.
- Поиск узла.
- Обход дерева.
- Удаление узла.

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(0);
    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);
}
```

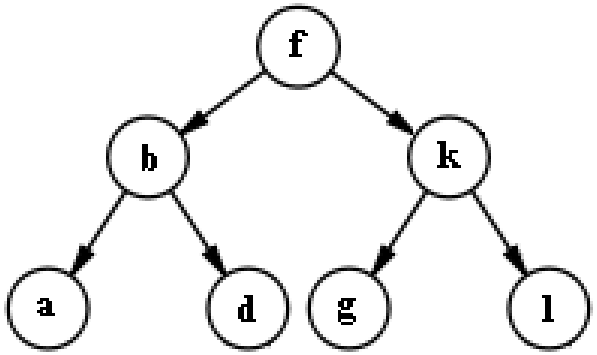
Поиск в дереве (2)

```
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

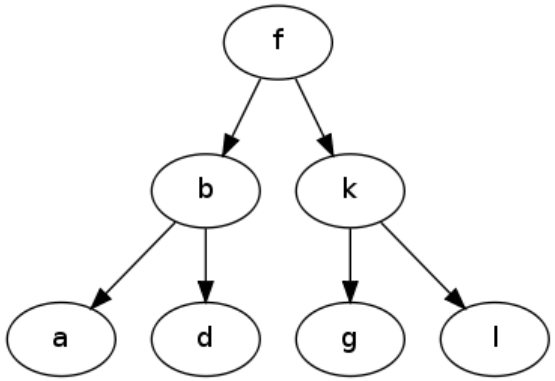
- DOT - язык описания графов.
- Граф, описанный на языке DOT, обычно представляет собой текстовый файл с расширением .gv в понятном для человека и обрабатывающей программы формате.
- В графическом виде графы, описанные на языке DOT, представляются с помощью специальных программ, например Graphviz.

В основном 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");
}
```