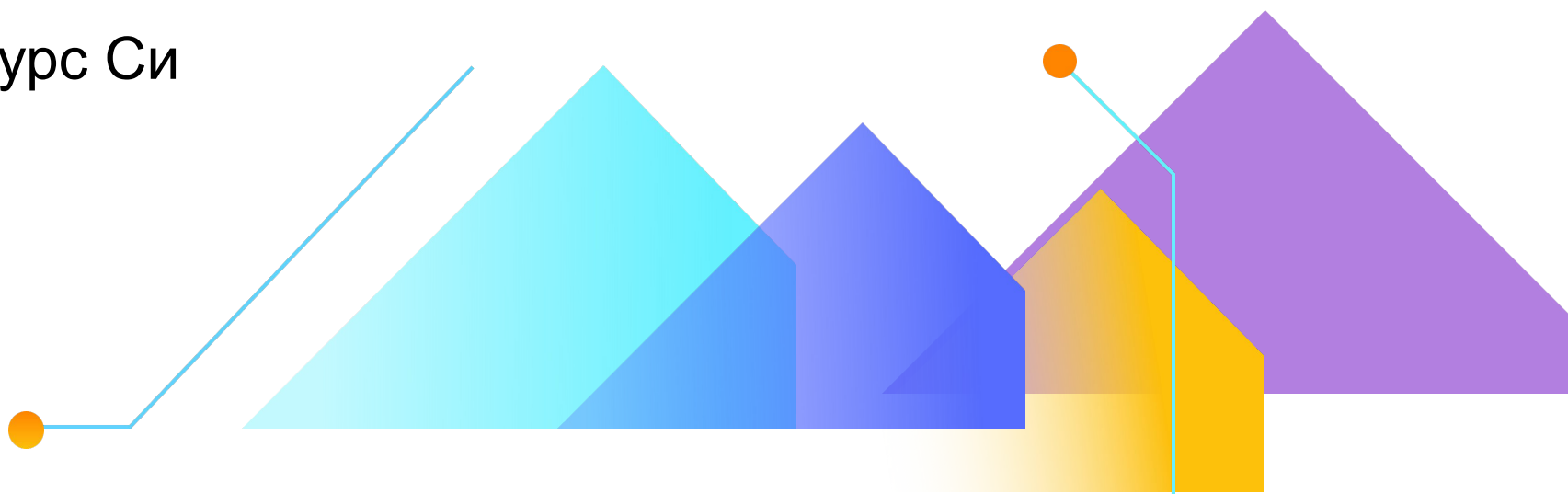




Лекция №7

Динамические структуры данных

Продвинутый курс Си






План курса

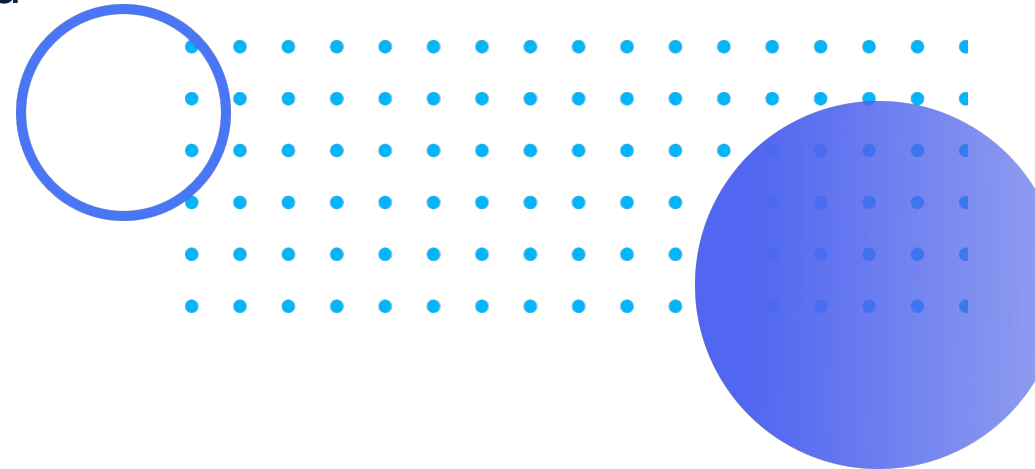
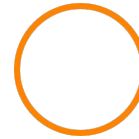
- Вводный урок
- Структуры. Динамические типы
- Библиотеки языка C
- Оптимизация кода
- Алгоритмы
- Компиляция и компиляторы
- Динамические структуры данных
- Курсовая работа



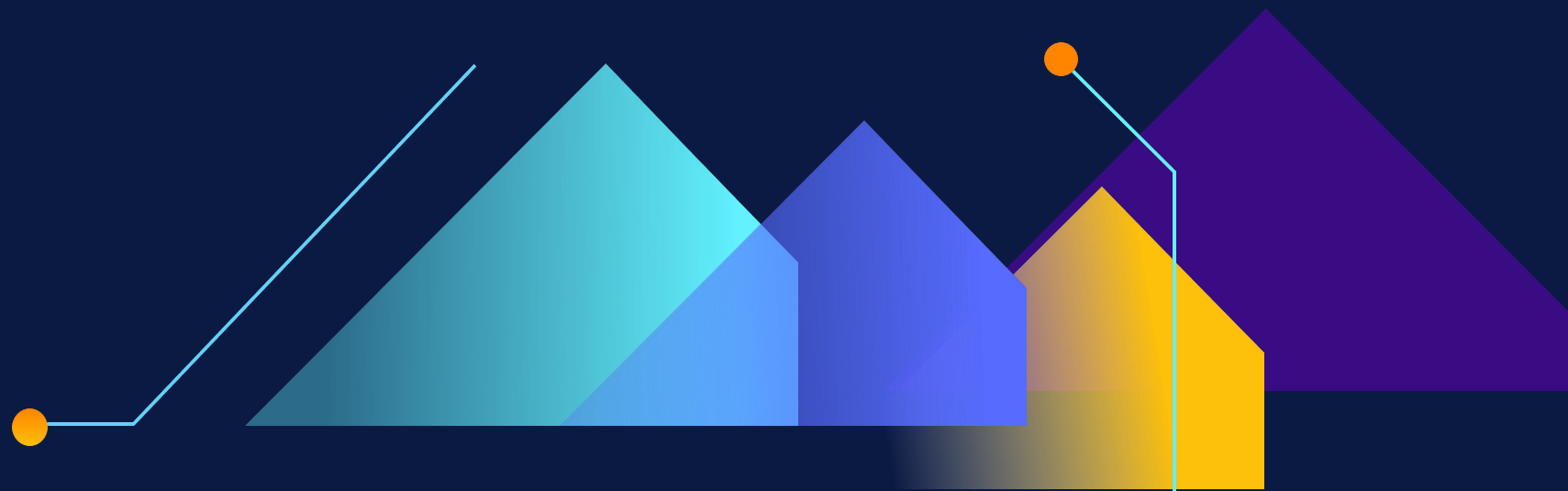
Маршрут

Динамические структуры данных

-  Изучим однонаправленные списки и двунаправленные списки
-  Познакомимся с двоичными деревьями
-  Узнаем что такое двоичные деревья поиска



Теория





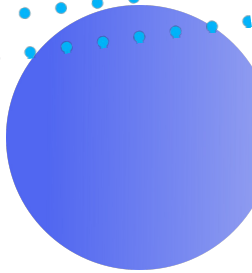
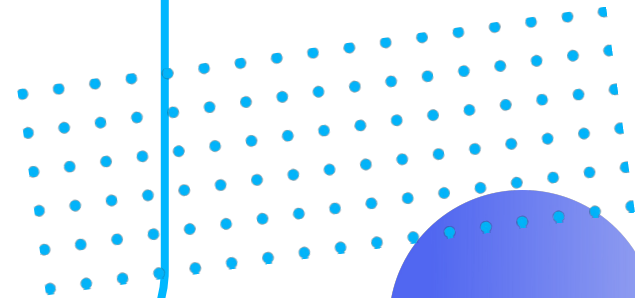
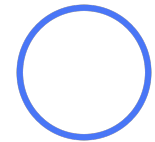
Динамические структуры данных

Динамические структуры данных или как их еще называют рекурсивными, используются для представления в памяти данных, размер которых заранее неизвестен.

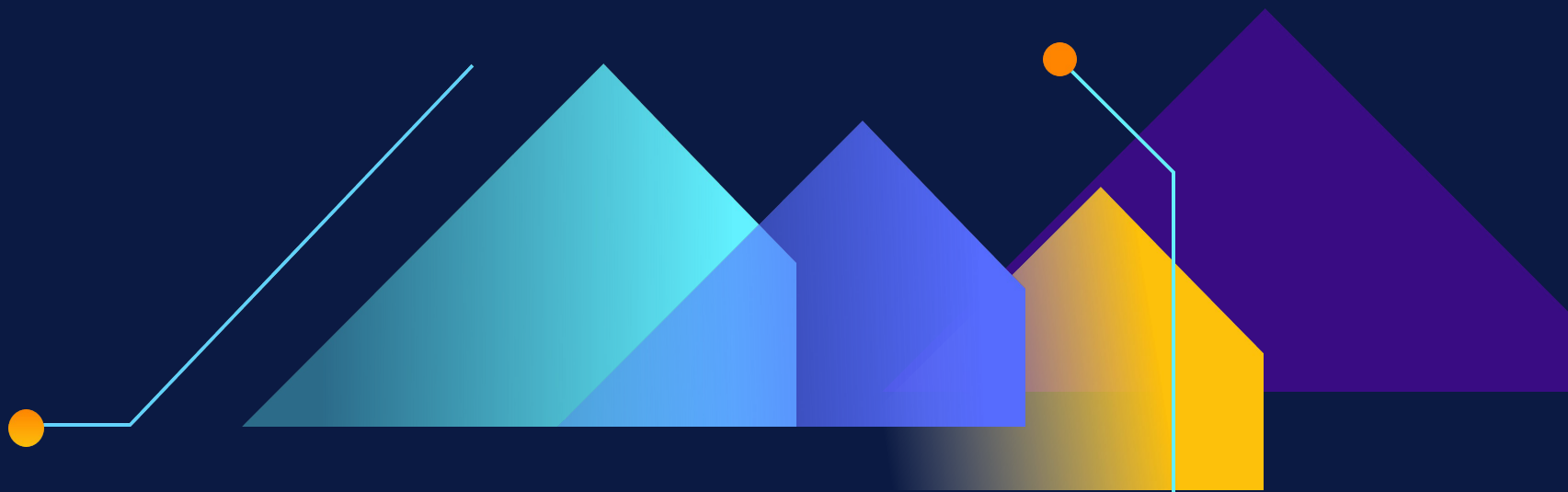
При реализации таких конструкций на языке Си для представления звеньев используются структурные типы, а для представления ссылок — **указатели**.

Память под звенья выделяется динамически (malloc) в куче и должна освобождаться (free) при удалении звеньев.

Работа с такими данными ведётся через указатели.



Списки



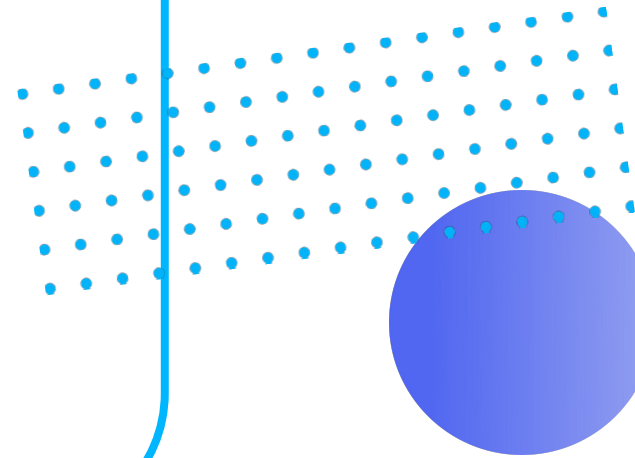
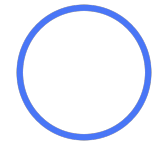


Динамические структуры данных

Типы рекурсивных данных, в которых содержится ссылка на следующий и/или предыдущий элемент называют списками.

Рассмотрим различные варианты таких списков:

- Однонаправленный список без заглавного элемента
- Однонаправленный список с заглавным элементом
- Двухнаправленный список без заглавного элемента
- Двухнаправленный список с заглавным элементом



Однонаправленный список

Это такой тип данных, в котором каждое звено содержит ссылку на следующее. При объявлении таких типов удобно использовать typedef.

```
typedef struct list {  
    uint32_t id;  
    struct list *next;  
} list;
```

Внимание! Тип структуры не может иметь поля типа самой этой структуры, однако допускаются указатели на тип самой структуры.

Однонаправленный список

Рассмотрим пример списка из трёх звеньев, содержащий числа 1, 2, 3. Для создания каждого элемента предпочтительнее использовать **calloc**, так как выделяемая память будет в этом случае инициализирована нулями и не потребуется заносить **NULL** в конечный элемент списка. Важно помнить, что **NULL** обычно служит признаком конца списка.

```
head = calloc(1, sizeof(list));
head->id = 1;
head->next = calloc(1, sizeof(list));
head->next->id = 2;
head->next->next =
calloc(1, sizeof(list));
head->next->next->id = 3;
```

```
head = malloc(sizeof(list));
head->id = 1;
head->next = malloc(sizeof(list));
head->next->id = 2;
head->next->next = malloc(sizeof(list));
head->next->next->id = 3;
/* Важно занести NULL в последний элемент */
head->next->next->next=NULL;
```

Пример реализации однонаправленного списка

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
typedef struct list {
    uint32_t id;
    struct list *next;
} list;
int main(void)
{
    list* head=NULL;
    head = calloc(1,sizeof(list));
    head->id = 1;
    head->next = calloc(1,sizeof(list));
    head->next->id = 2;
    head->next->next = calloc(1,sizeof(list));
    head->next->next->id = 3;
    printf("%d %d %d\n",head->id,
head->next->id, head->next->next->id);
    return 0;
}
```

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
typedef struct list {
    uint32_t id;
    struct list *next;
} list;
int main(void)
{
    list* head=NULL;
    head = malloc(sizeof(list));
    head->id = 1;
    head->next = malloc(sizeof(list));
    head->next->id = 2;
    head->next->next = malloc(sizeof(list));
    head->next->next->id = 3;
    /* Важно занести NULL в последний элемент */
    head->next->next->next=NULL;
    printf("%d %d %d\n",head->id, head->next->id,
head->next->next->id);
    return 0;
}
```

Печать однонаправленного списка

При работе со списком удобно описать соответствующие обслуживающие функции и в дальнейшем эксплуатировать их.

//Итерационная печать

```
void printListIteration(list *p)
{
    while(p)
    {
        printf("%d ",p->id);
        p = p->next;
    }
    printf("\n");
}
```

//Рекурсивная печать

```
void printListRecurs(list *p)
{
    if(p)
    {
        printf("%d ",p->id);
        printListRecurs(p->next);
    }
    printf("\n");
}
```

Добавление звена в начало списка без заглавного элемента

```
/* Без заглавного элемента */
void insert(list **head, int32_t
value) {
    list *new =
calloc(1, sizeof(list));
    new->id = value;
    new->next = *head;
    *head = new;
}
int main(void) {
    insert(&L, 100);
    return 0;
}
```

```
/* Без заглавного элемента и без
двойного указателя*/
list* insert2(list *head, int32_t
value) {
    list *new =
calloc(1, sizeof(list));
    new->id = value;
    new->next = head;
    return new;
}
int main(void) {
    L = insert2(L, 100);
    return 0;
}
```

Однонаправленный список с заглавным звеном

Часто для облегчения операций добавления и удаления звеньев в списки используют списки с заглавным звеном. В заглавном звене не хранят никаких данных.

Для списка с заглавным звеном указатель на его начало никогда не изменяется, это удобно при передаче такого списка в функцию в качестве указателя.

```
// Добавляем заглавное звено
list* L = calloc(1, sizeof(list));
// Эти данные не используются
L->id = -1;
```

Добавление звена в начало списка с заглавным элементом

```
/* С заглавным элементом, в этом случае двойной указатель не нужен*/
void insert3(list *head, int32_t value) {
    list *new = calloc(1, sizeof(list));
    new->id = value;
    new->next = head->next; //в списке точно есть хотя бы одно звено
    head->next = new;
}

int main(void) {
    L = calloc(1, sizeof(list)); // Добавляем заглавное звено
    L->id = -1; // Эти данные не используются
    insert3(L, 123);
    return 0;
}
```

Добавление звена в конец списка без заглавного элемента

```
/* Без заглавного элемента с использованием двойного указателя*/
void insert_end(list **head, int32_t value) {
    list *new = calloc(1, sizeof(list));
    new->id = value;
    if( *head == NULL ) { // пустой список
        *head = new; // изменяем голову списка
    } else {
        list *p = *head;
        while(p->next != NULL)
            p = p->next; // идем в конец списка
        p->next = new;
    }
}

int main(void) {
    insert_end(&L, 100);
    return 0;
}
```

Добавление звена в конец списка без заглавного элемента

```
/* Без заглавного элемента и без двойного указателя*/
list* insert_end2(list *head, int32_t value) {
    list *new = calloc(1, sizeof(list));
    new->id = value;
    if( head == NULL ) { // пустой список
        return new;
    } else {
        list *p = head;
        while(p->next != NULL)
            p = p->next; // идем в конец списка
        p->next = new;
        return head;
    }
}

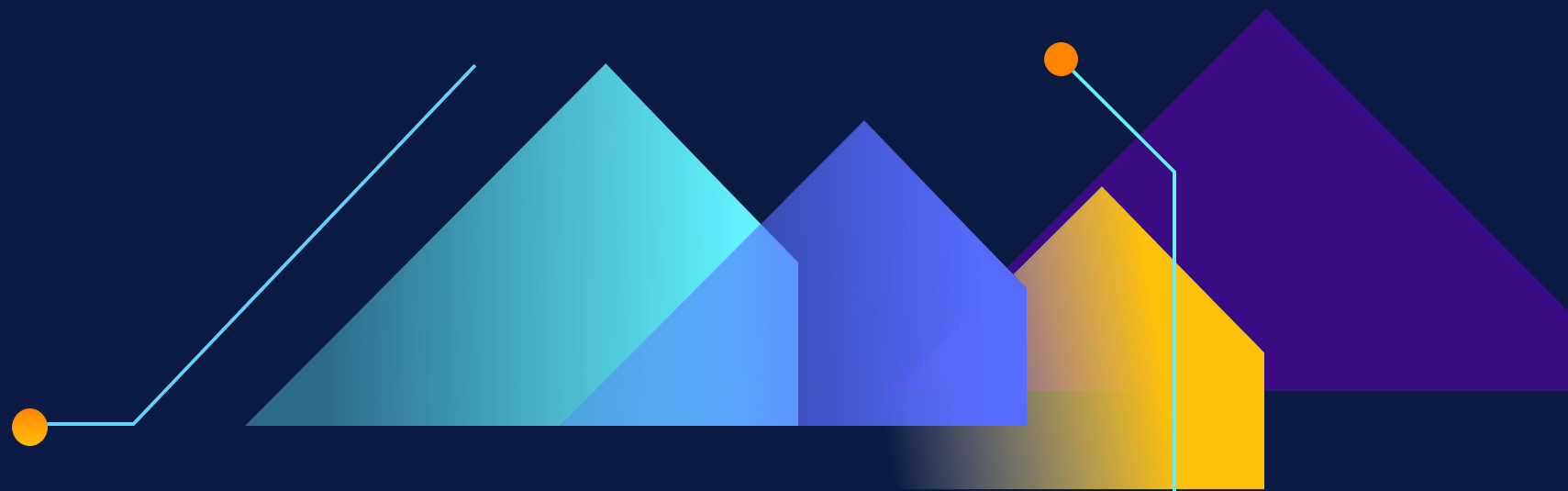
int main(void) {
    L = insert_end2(L, 100);
    return 0;
}
```


Рекурсивная реализация часто оказывается короче

```
/* Без заглавного элемента, рекурсивный вариант*/
void insert_end_recurs(list **head, int32_t value) {
    if(*head == NULL) {
        (*head) = calloc(1, sizeof(list));
        (*head)->id = value;
    } else {
        insert_end_recurs( &((*head)->next), value );
    }
}

int main(void) {
    insert_end_recurs(&L, 100);
    return 0;
}
```

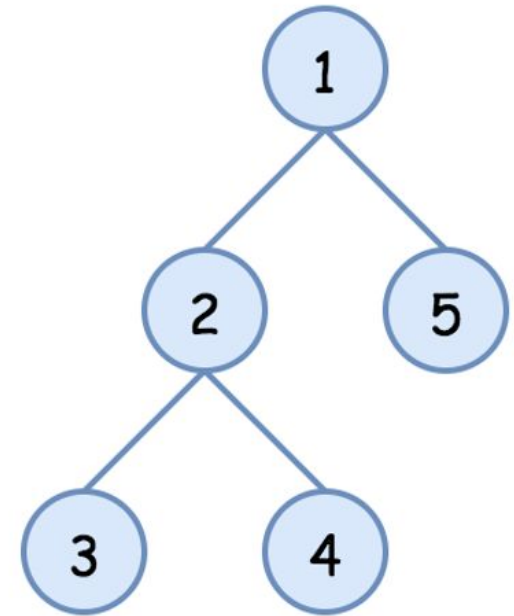
Двоичные деревья



Определение

Двоичное дерево — набор узлов, который:

- либо пуст (пустое двоичное дерево),
- либо разбит на три непересекающиеся части:
 - узел, называемый корнем
 - двоичное дерево, называемое левым поддеревом
 - двоичное дерево, называемое правым поддеревом



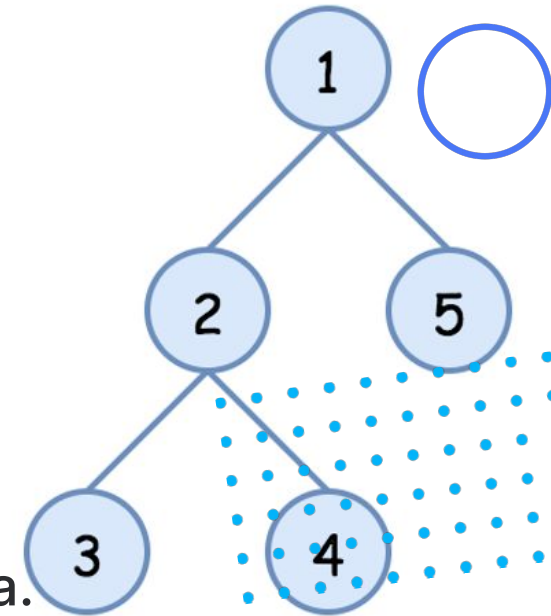


Двоичные деревья

Каждый узел двоичного дерева можно представить в виде следующих полей:

- Ключ и/или данные, по которому их можно идентифицировать
- Указатель на левое поддерево
- Указатель на правое поддерево
- Указатель на родителя (необязательное поле)

Внимание! Значение ключа уникально для каждого узла.

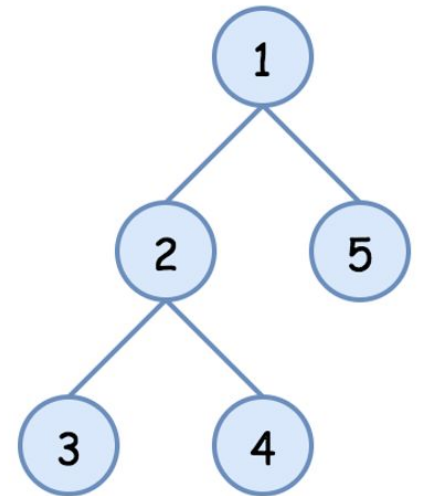


Двоичные деревья

Для организации хранения больших объёмов данных, быстрого доступа к ним, а также возможности модифицировать их существует способ организации хранилища в виде двоичного дерева поиска.

Эту структуру данных можно описать в виде структурного типа данных.

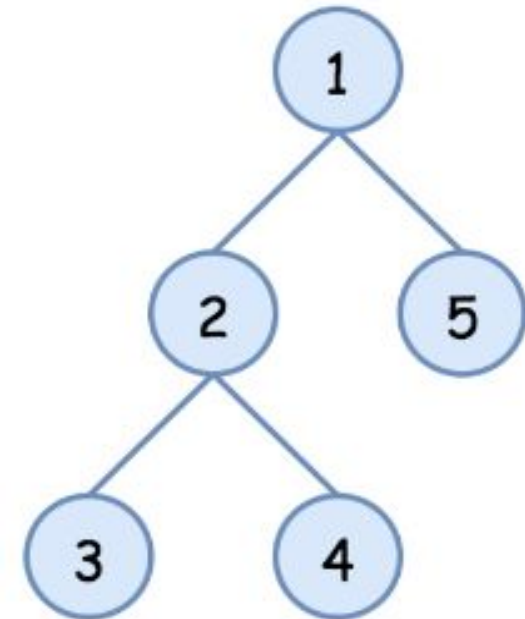
```
typedef struct tree {  
    datatype key;  
    struct tree *left, *right;  
    struct tree *parent; // необязательное поле  
} tree;
```



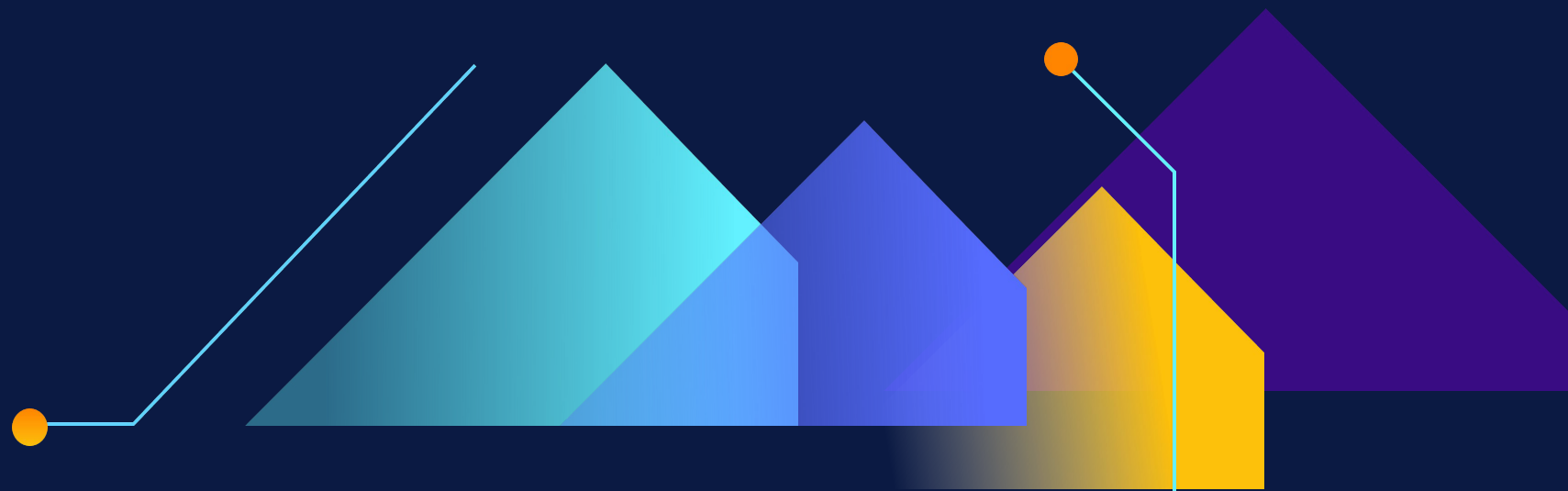
Пример двоичных деревьев

Рассмотрим пример создания дерева:

```
tree *tr = NULL;  
tr = calloc(1, sizeof(tree));  
tr->key = 1;  
tr->right = calloc(1, sizeof(tree));  
tr->right->key = 5;  
tr->left = calloc(1, sizeof(tree));  
tr->left->key = 2;  
tr->left->left = calloc(1, sizeof(tree));  
tr->left->left->key = 3;  
tr->left->right = calloc(1, sizeof(tree));  
tr->left->right->key = 4;
```



Обходы двоичных деревьев

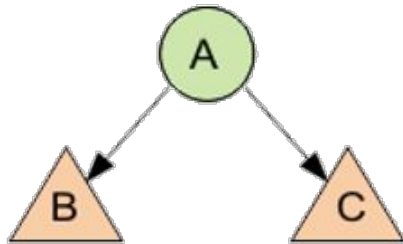


Обходы в глубину DFS (Depth First Traversals)

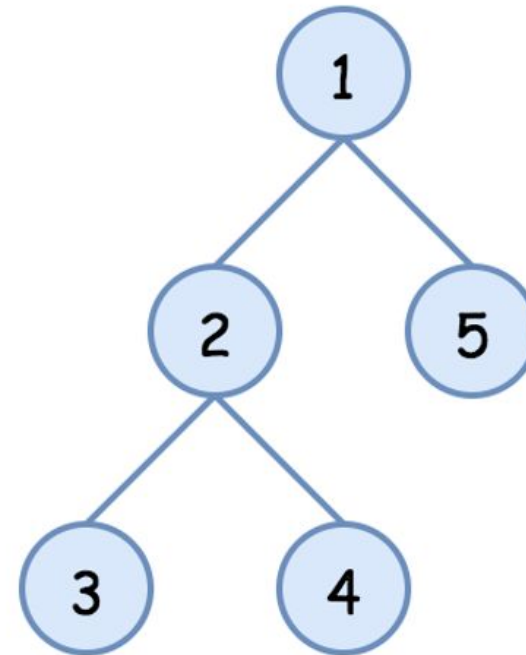
Существуют различные способы обхода двоичных деревьев.

Рассмотрим их на примере дерева:

- Preorder (Root, Left, Right) : 1 2 3 4 5
- Inorder (Left, Root, Right) : 3 2 4 1 5
- Postorder (Left, Right, Root) : 3 4 2 5 1



A, B, C
B, A, C
B, C, A



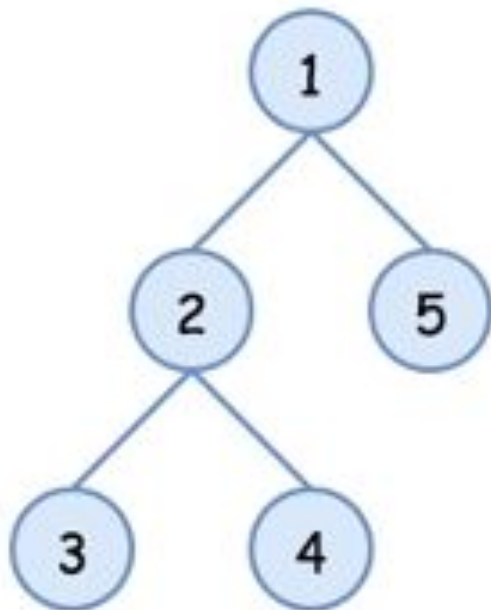
Сравнение обходов DFS

DFS Preorder
Node -> Left -> Right

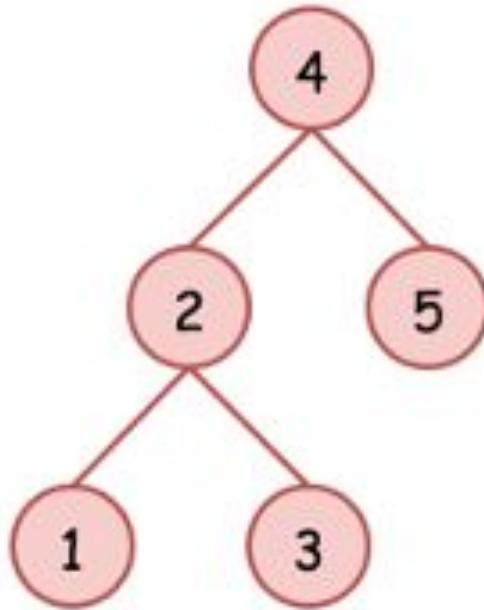
DFS Inorder
Left -> **Node** -> Right

DFS Postorder
Left -> Right -> **Node**

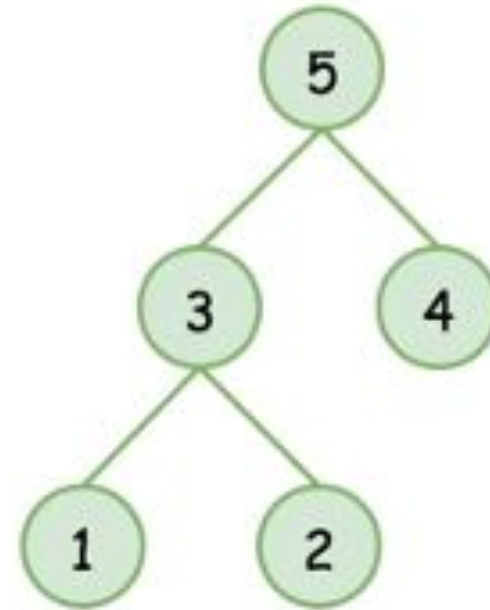
Traversal = [1, 2, 3, 4, 5]



(Root, Left, Right)
1 2 3 4 5



(Left, Root, Right)
3 2 4 1 5

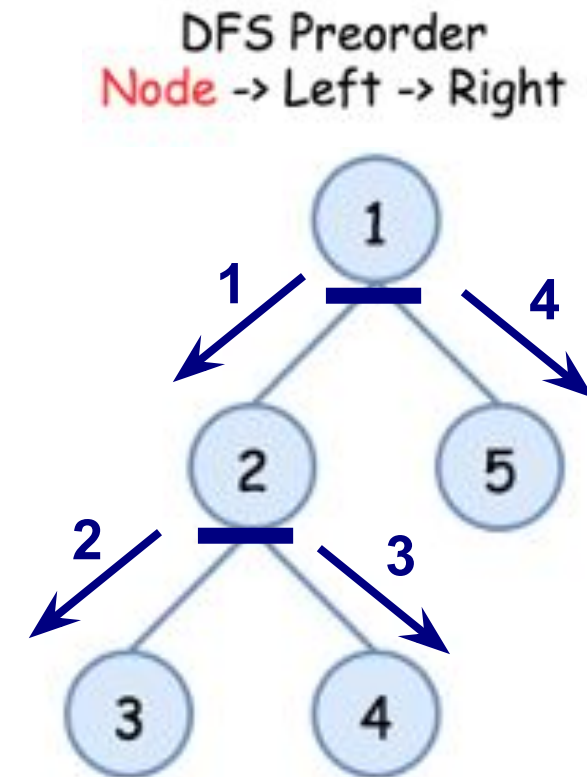


(Left, Right, Root)
3 4 2 5 1

DFS (Depth First Traversals) Preorder

→ Preorder (Root, Left, Right) : 1 2 3 4 5

```
void preorder(tree *root) {  
    if(root == NULL)  
        return;  
    printf("%d ", root->key);  
    if(root->left)  
        preorder(root->left);  
    if(root->right)  
        preorder(root->right);  
}  
...  
preorder(tr);
```

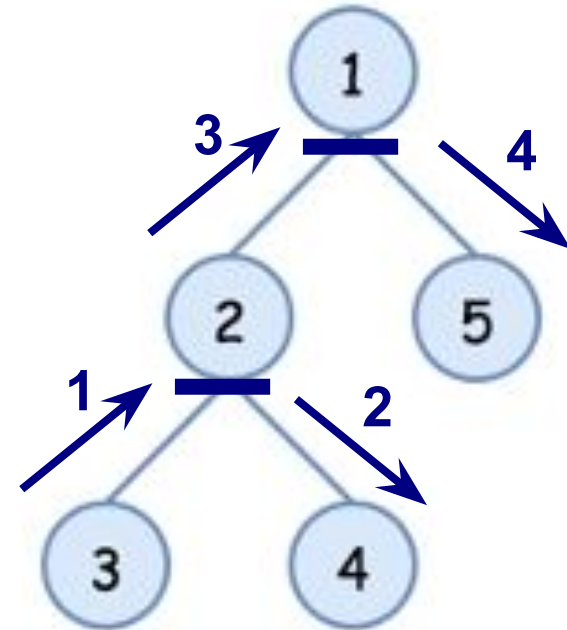


DFS (Depth First Traversals) Inorder

→ Inorder (Left, Root, Right) : 3 2 4 1 5

```
void inorder(tree *root) {  
    if(root == NULL)  
        return;  
    if(root->left)  
        inorder(root->left);  
    printf("%d ", root->key);  
    if(root->right)  
        inorder(root->right);  
}  
...  
inorder(tr);
```

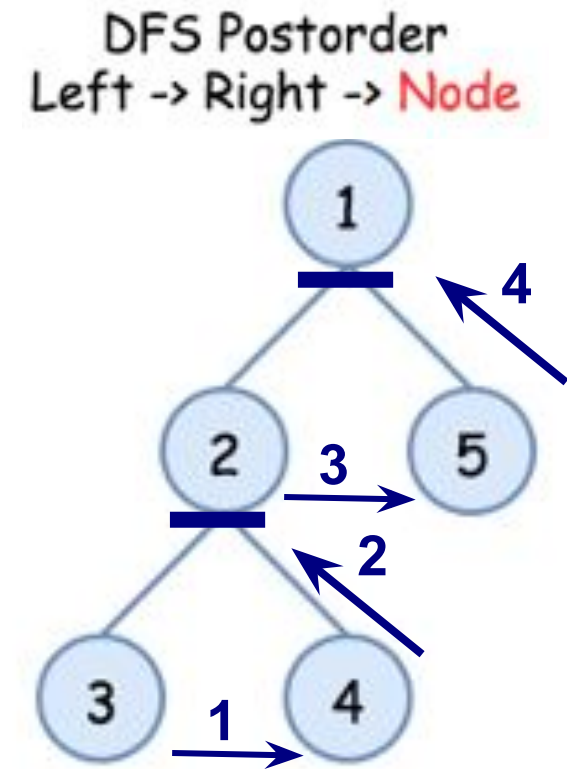
DFS Inorder
Left -> **Node** -> Right



DFS (Depth First Traversals) Postorder

→ Postorder (Left, Right, Root) : 3 4 2 5 1

```
void postorder(tree *root) {  
    if(root == NULL)  
        return;  
    if(root->left)  
        postorder(root->left);  
    if(root->right)  
        postorder(root->right);  
    printf("%d ", root->key);  
}  
...  
postorder(tr);
```



Обходы в глубину DFS (Depth First Traversals)

- Inorder (Left, Root, Right) : 3 2 4 1 5
- Preorder (Root, Left, Right) : 1 2 3 4 5
- Postorder (Left, Right, Root) : 3 4 2 5 1

```
void inorder(tree *root) {  
    if(root == NULL)  
        return;  
    if(root->left)  
        inorder(root->left);  
    printf("%d ", root->key);  
    if(root->right)  
        inorder(root->right);  
}
```

```
void preorder(tree *root) {  
    if(root == NULL)  
        return;  
    printf("%d ", root->key);  
    if(root->left)  
        preorder(root->left);  
    if(root->right)  
        preorder(root->right);  
}
```

```
void postorder(tree *root) {  
    if(root == NULL)  
        return;  
    if(root->left)  
        postorder(root->left);  
    if(root->right)  
        postorder(root->right);  
    printf("%d ", root->key);  
}
```

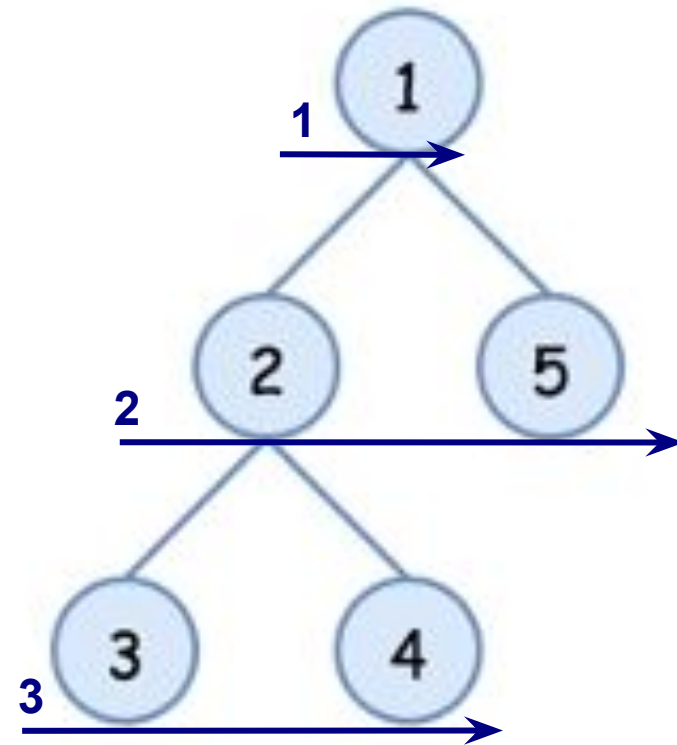
Обход в ширину BFS (Breadth First Traversal)

```
int heightTree(tree* p)
{
    //1. Вычисляем высоту дерева
}

void printCurrentLevel(tree* root, int level)
{
    //2. Печатаем все узлы на уровне level
}

void printBFS(tree* root)
{
    //3. Функция печати
}
```

BFS
Node -> Left -> Right



Обход в ширину BFS (Breadth First Traversal)

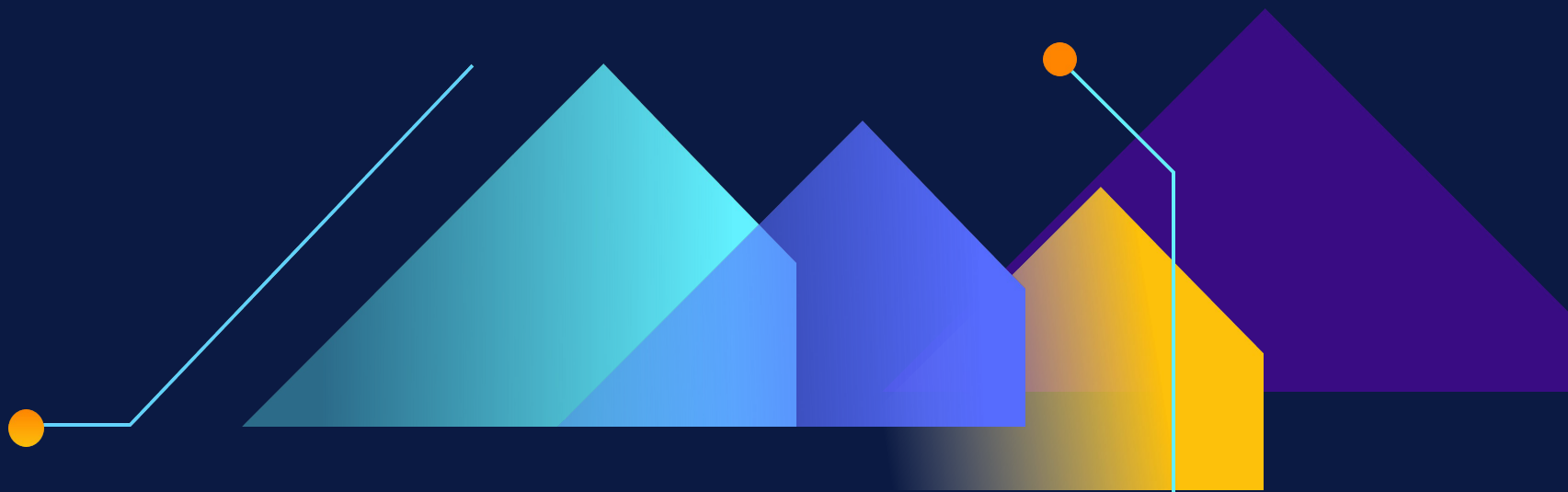
Для этого алгоритма удобно использовать две функции. Одна из которых печатает узлы на определённом уровне, вторая обходит все уровни дерева. Для исходного дерева вывод будет: 1 2 5 3 4.

```
int heightTree(tree* p){
    if (p == NULL)
        return 0;
    else {
        /* вычисляем высоту каждого поддерева */
        int lheight = heightTree(p->left);
        int rheight = heightTree(p->right);
        if (lheight > rheight)
            return (lheight + 1);
        else
            return (rheight + 1);
    }
}

void printCurrentLevel(tree* root, int level){
    if (root == NULL)
        return;
    if (level == 1)
        printf("%d ", root->key);
    else if (level > 1) {
        printCurrentLevel(root->left, level - 1);
        printCurrentLevel(root->right, level - 1);
    }
}
```

```
void printBFS(tree* root)
{
    int h = heightTree(root);
    int i;
    for (i = 1; i <= h; i++)
        printCurrentLevel(root, i);
}
```

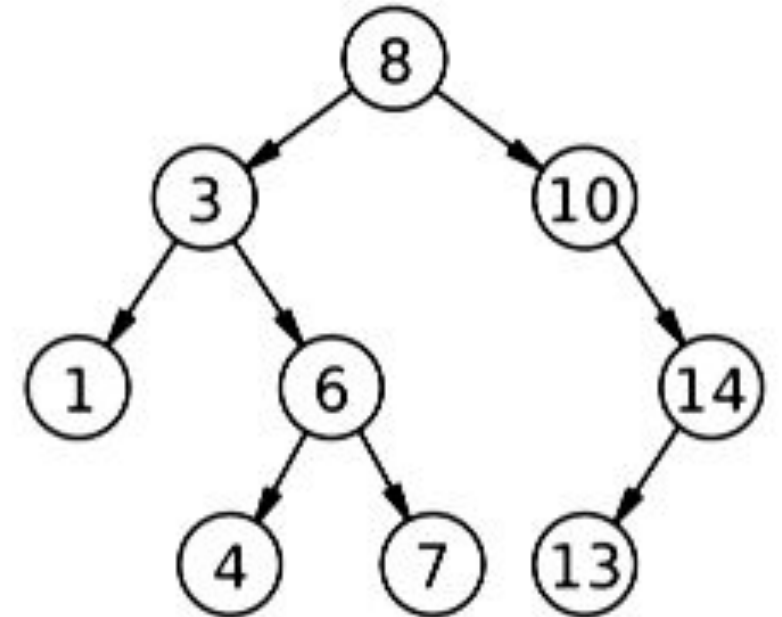
Дерево поиска



Дерево поиска

Дерево поиска — это двоичное дерево, в котором узлы упорядочены по значению ключей. Для любого узла верно следующее утверждение:

- Значения ключей всех узлов его левого поддерева меньше его значения
- Значения ключей всех узлов его правого поддерева больше его значения

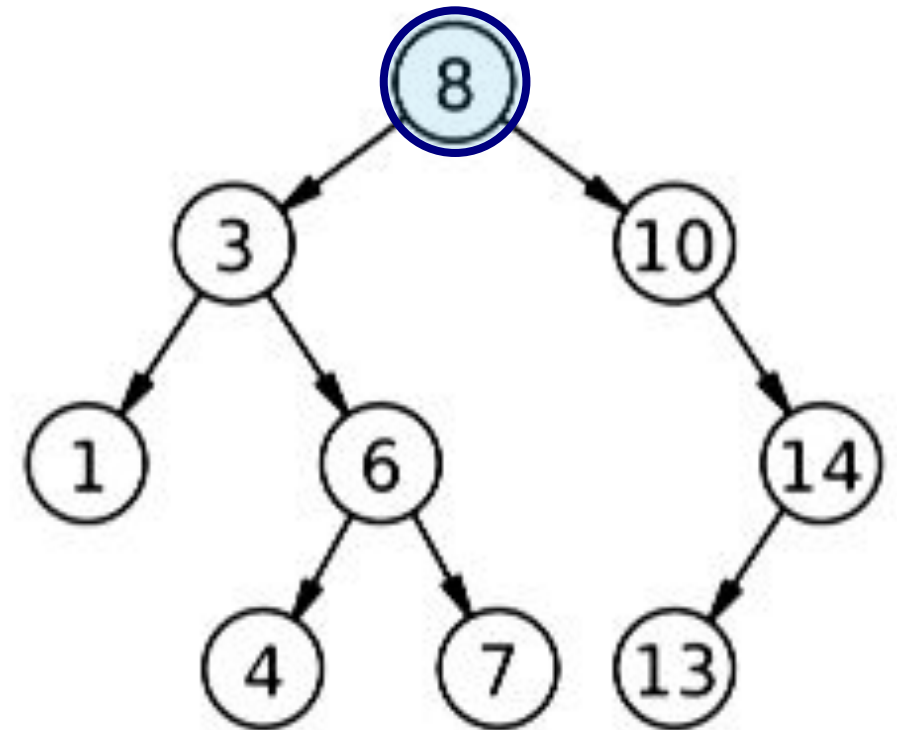


Поиск узла по ключу

Ключи в двоичном дереве поиска хранятся с соблюдением свойства упорядоченности.

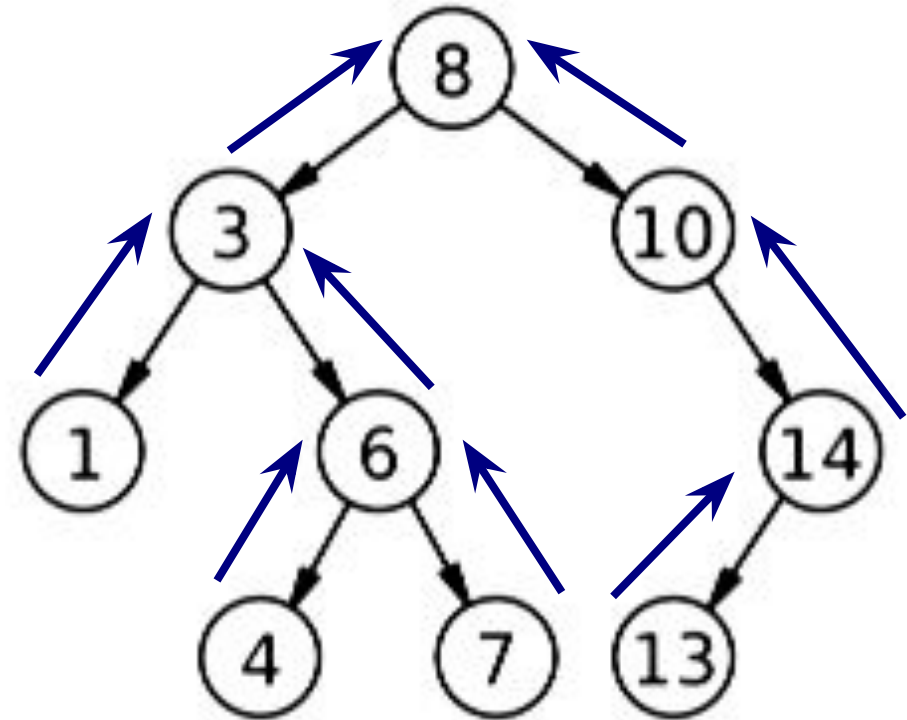
На каждом шаге значение искомого узла сравнивается со значением ключа в текущем узле.

- Если значение искомого узла меньше, то переходим к левому поддереву
- Если значение искомого узла больше, то переходим к правому поддереву
- Если значение равно, то ключ найден
- Если поддерево пусто, то не найден



Структура двоичного дерева поиска

```
struct tree{  
    int key;  
    struct tree *left;  
    struct tree *right;  
    struct tree *parent;  
    //позволяет двигаться вверх по  
    дереву  
};
```

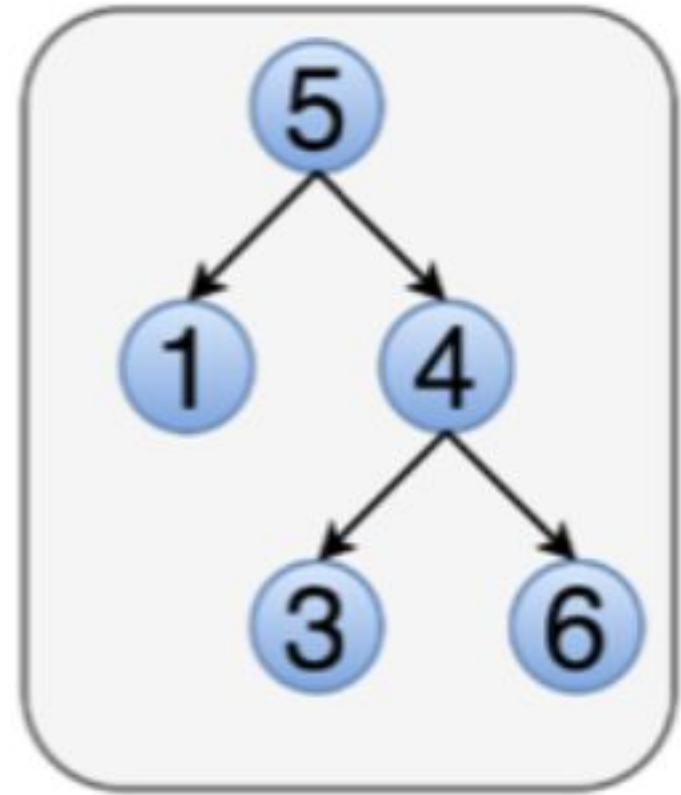


Вопрос: Является бинарное дерево бинарным деревом поиска?

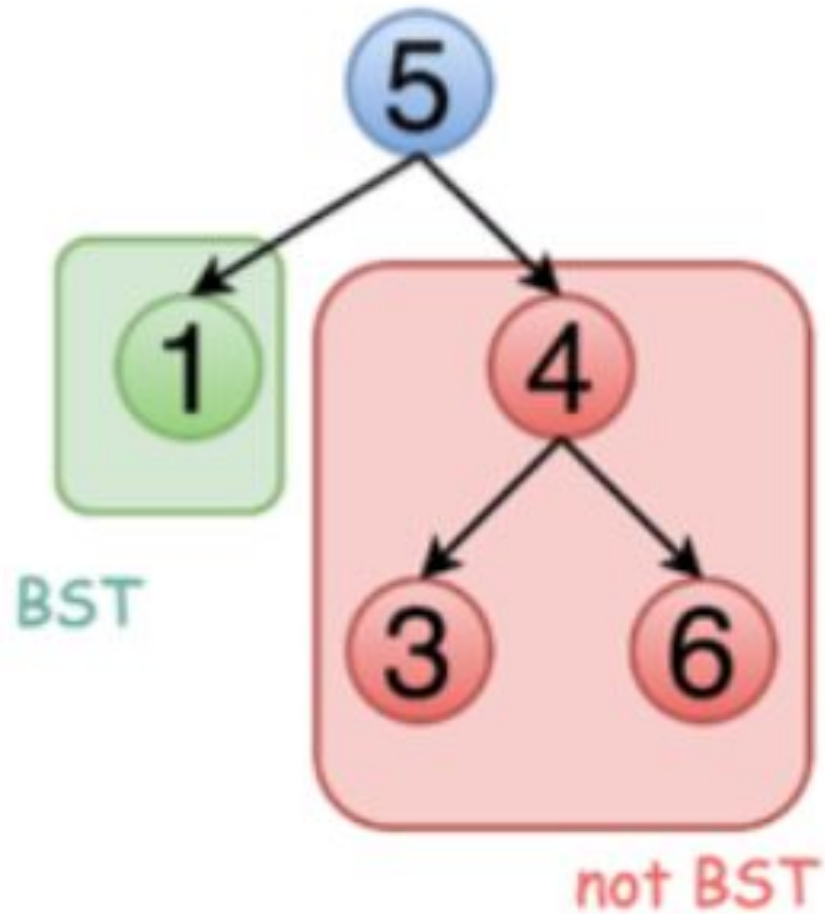
Дерево поиска — это двоичное дерево, в котором узлы упорядочены по значению ключей.

Для любого узла верно следующее утверждение:

- Значения ключей всех узлов его левого поддеревья меньше его значения
- Значения ключей всех узлов его правого поддеревья больше его значения



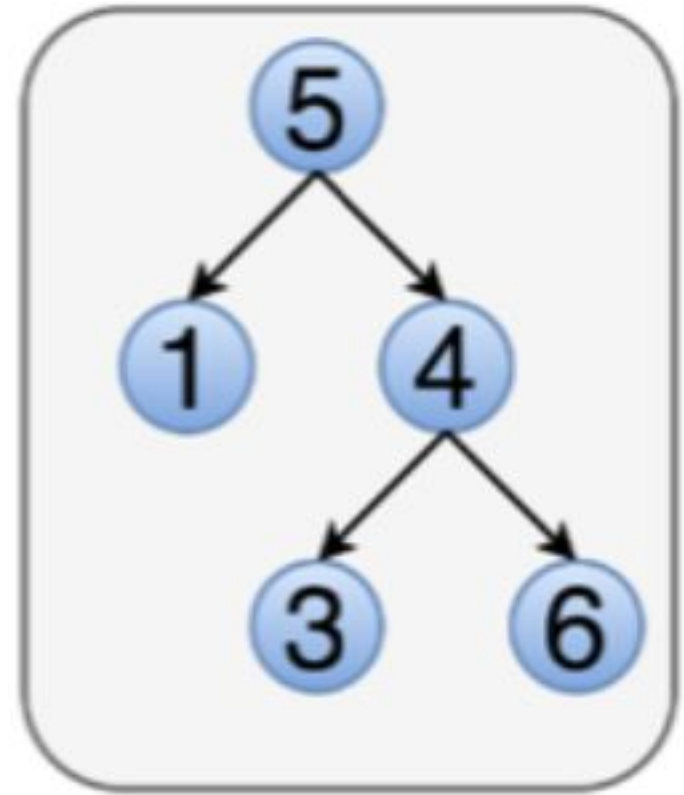
Ответ: Не является!



BST and not BST = not BST
return False

Программная реализация

```
typedef enum {false=0,true=1} bool;
bool isBST(tree* root)
{
    static tree *prev = NULL;
    // traverse the tree in inorder fashion and keep
    track of prev node
    if (root)
    {
        if (!isBST(root->left))
            return false;
        // Allows only distinct valued nodes
        if (prev != NULL && root->key <= prev->key)
            return false;
        prev = root;
        return isBST(root->right);
    }
    return true;
}
```

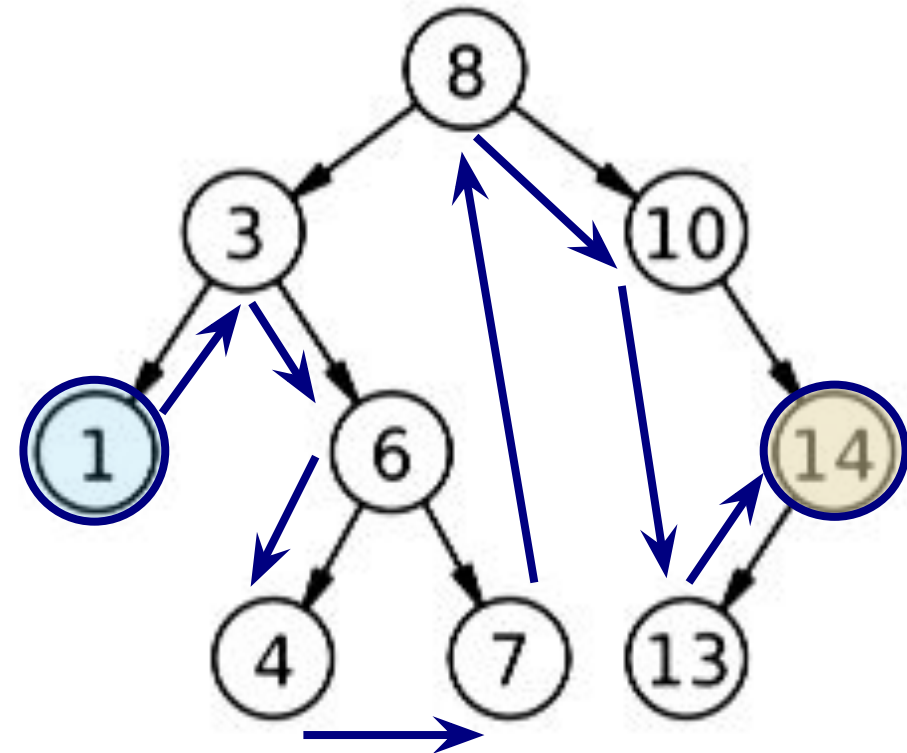


Зачем нужно двоичное дерево поиска?

Есть массив из чисел 8, 3, 10, 1, 6, 14, 4, 7, 13

- Крайний левый элемент всегда минимум, крайний правый максимум
- Если идти направо от минимума, то получим отсортированный массив по возрастанию
- Если идти налево от максимума, то получим отсортированный массив по убыванию

Количество сравнений нахождения элемента в дереве в самом худшем виде это высота дерева,
в массиве — длина массива



Вставка нового узла

Добавление нового ключа схоже с поиском ключа.

- Сначала надо найти место, куда вставить ключ
- Создать элемент
- Прикрепить его

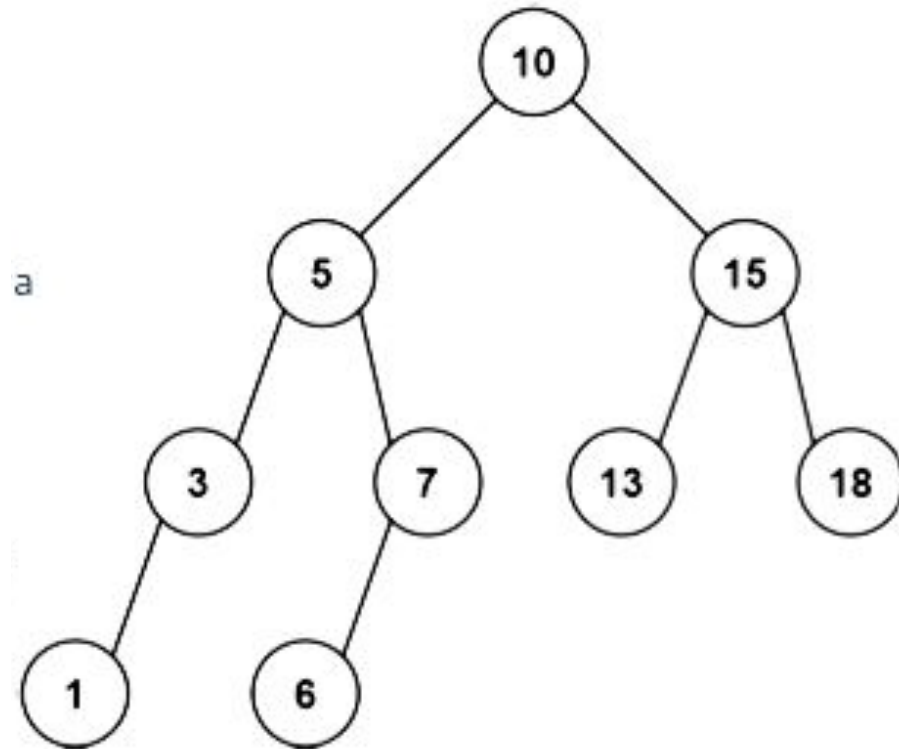
Для добавления нового узла удобно использовать двойной указатель, т.к может возникнуть ситуация, когда дерево ещё пустое, и требуется изменить его корень.

Сложность вставки аналогична поиску и оценивается как $O(h)$.

```
void insert(tree **root, int key, tree *pt)
{
    if(!(*root)) {
        // дерево пустое или дошли до нужного места
        *root=malloc(sizeof(tree));
        (*root)->key=key;
        (*root)->parent=pt;
        // с calloc строчка ниже не нужна
        (*root)->left=(*root)->right= NULL;
    }
    else if( key < (*root)->key)
        insert(&((*root)->left), key, *root);
    else
        insert(&((*root)->right), key, *root);
}
```

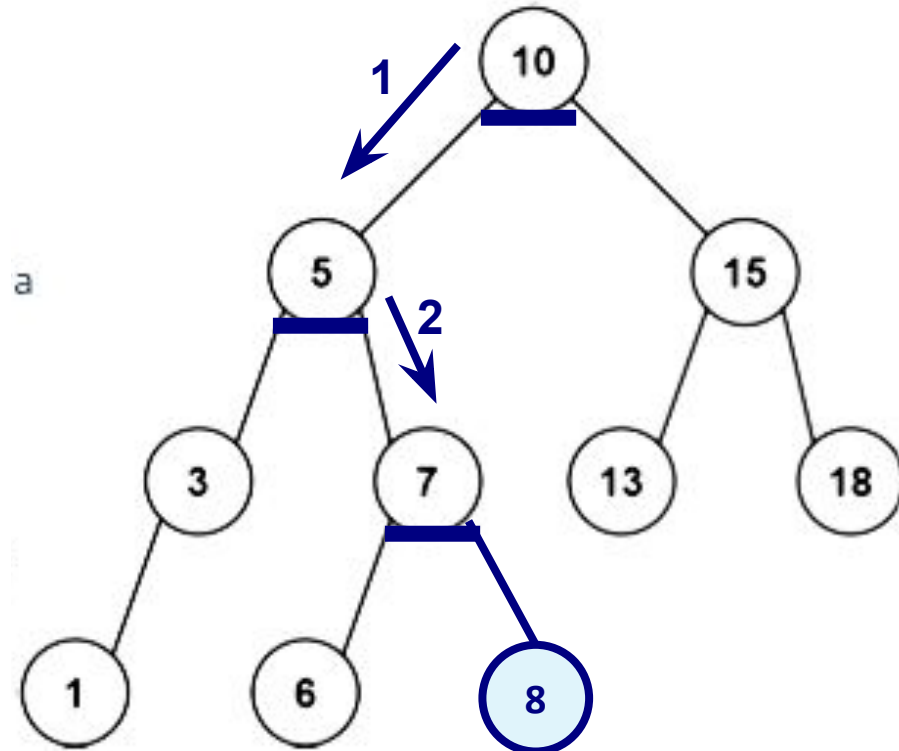

Двоичное дерево поиска

```
tree *tr = NULL;  
insert(&tr, 10, NULL);  
insert(&tr, 5, NULL);  
insert(&tr, 15, NULL);  
insert(&tr, 3, NULL);  
insert(&tr, 7, NULL);  
insert(&tr, 13, NULL);  
insert(&tr, 18, NULL);  
insert(&tr, 1, NULL);  
insert(&tr, 6, NULL);  
printBFS(tr);  
preorder(tr);  
inorder(tr);
```

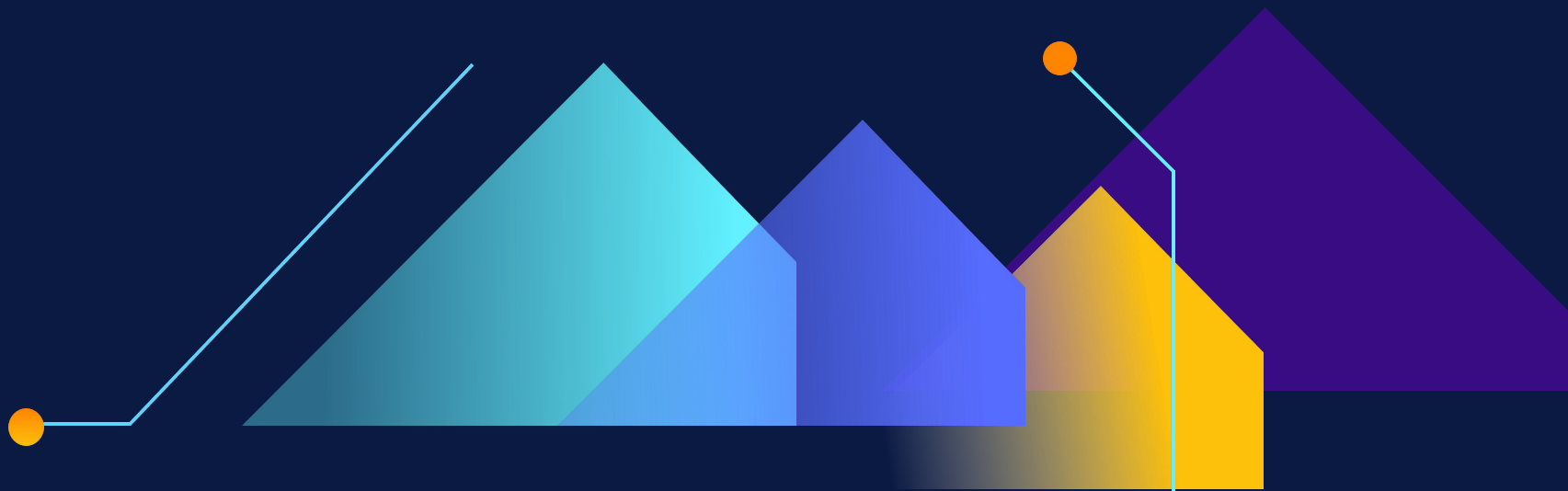


Добавление узла BST

```
tree *tr = NULL;  
insert(&tr, 10, NULL);  
insert(&tr, 5, NULL);  
insert(&tr, 15, NULL);  
insert(&tr, 3, NULL);  
insert(&tr, 7, NULL);  
insert(&tr, 13, NULL);  
insert(&tr, 18, NULL);  
insert(&tr, 1, NULL);  
insert(&tr, 6, NULL);  
printBFS(tr);  
printf("\n");  
insert(&tr, 8, NULL);  
printBFS(tr);  
printf("\n");
```



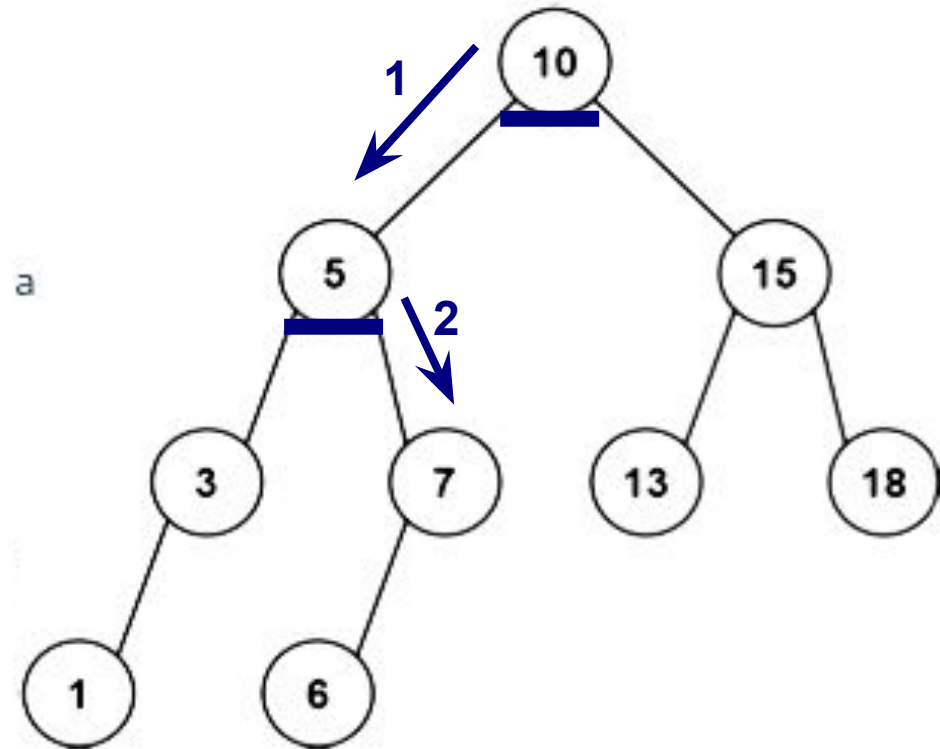
Поиск узла по ключу



Поиск узла в двоичном дереве

При поиске ключа в дереве, необходимо сравнивать его со значением в текущем узле:

- Если текущее поддерево пустое, то такого ключа нет
- Если значение ключа в текущем поддерево равно искомому, то ключ найден
- Если значение искомого ключа меньше то осуществлять переход в левое поддерево
- Если значение искомого ключа больше то осуществлять переход в правое поддерево

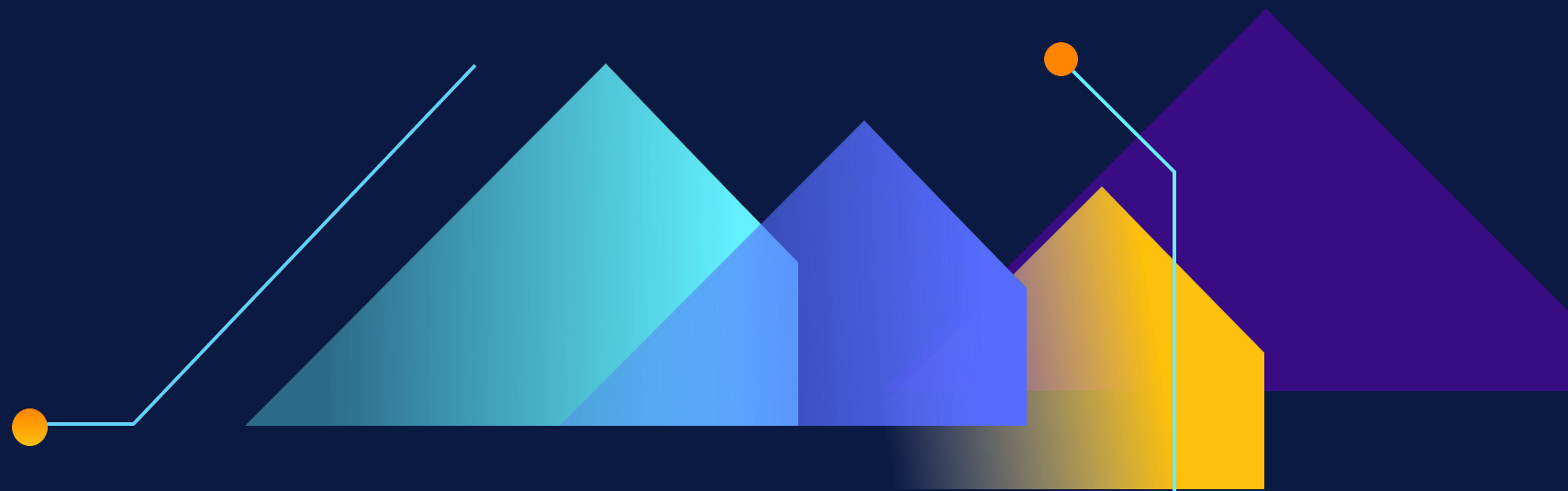


Поиск узла по ключу

```
/* Рекурсивная реализация */
tree* search_tree(tree *root,
int32_t key) {
    if(root==NULL || root->key ==
key)
        return root;
    else if(root->key > key)
        return
search_tree(root->left, key);
    else
        return
search_tree(root->right, key);
}
```

```
/* Итеративная версия */
tree* search_tree_i(tree *root,
int32_t key){
    tree *find=root;
    while(find && find->key!=key)
    {
        if( key < find->key )
            find = find->left;
        else
            find = find->right;
    }
    return find;
}
```

Удаление узла



Удаление узла

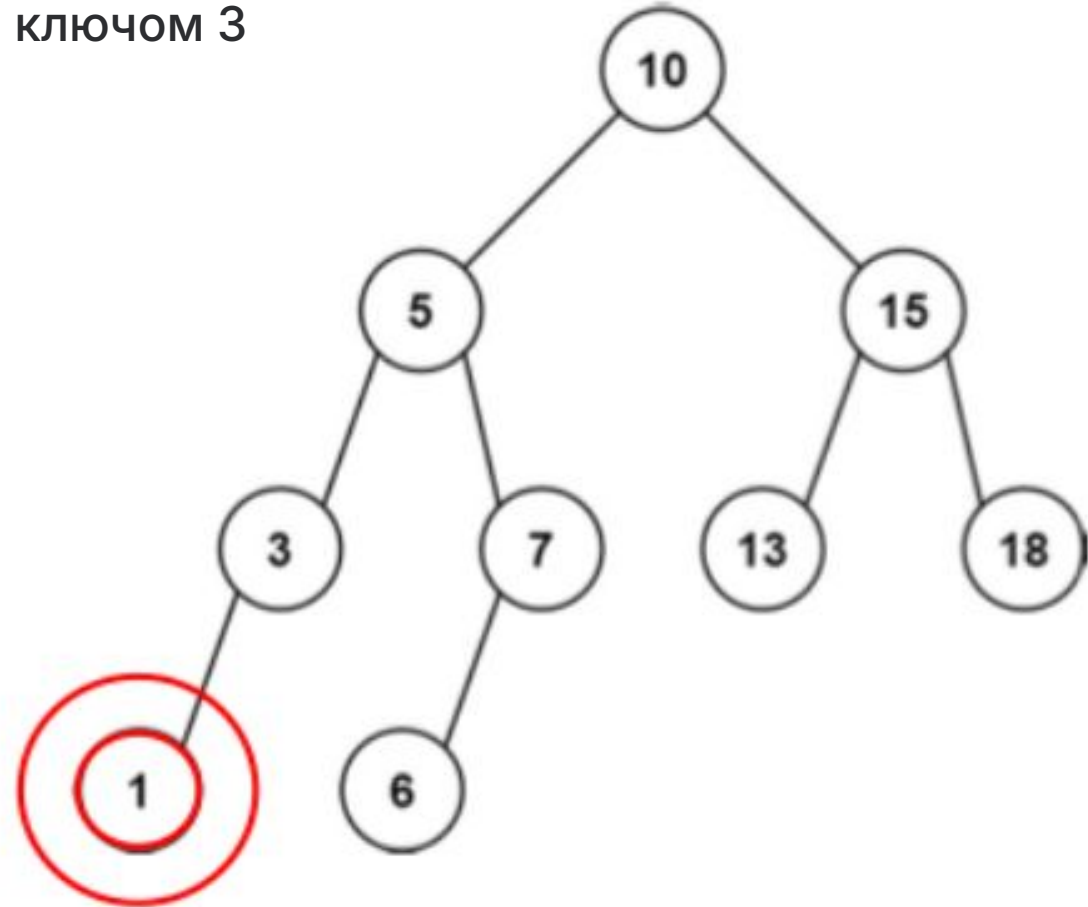
При удалении узла могут возникнуть три различных ситуации:

- Узел не имеет потомков. В этом случае происходит его удаление. В родительский элемент добавляется нулевой указатель
- Узел имеет одного потомка. В данном случае узел удаляется, а его потомок переходит к родителю
- Узел имеет два потомка. В этом случае необходимо найти минимальный элемент в правом поддереве

Сложность удаления элемента из дерева оценивается как $O(h)$, даже если при удалении придётся искать последователя удаляемого элемента.

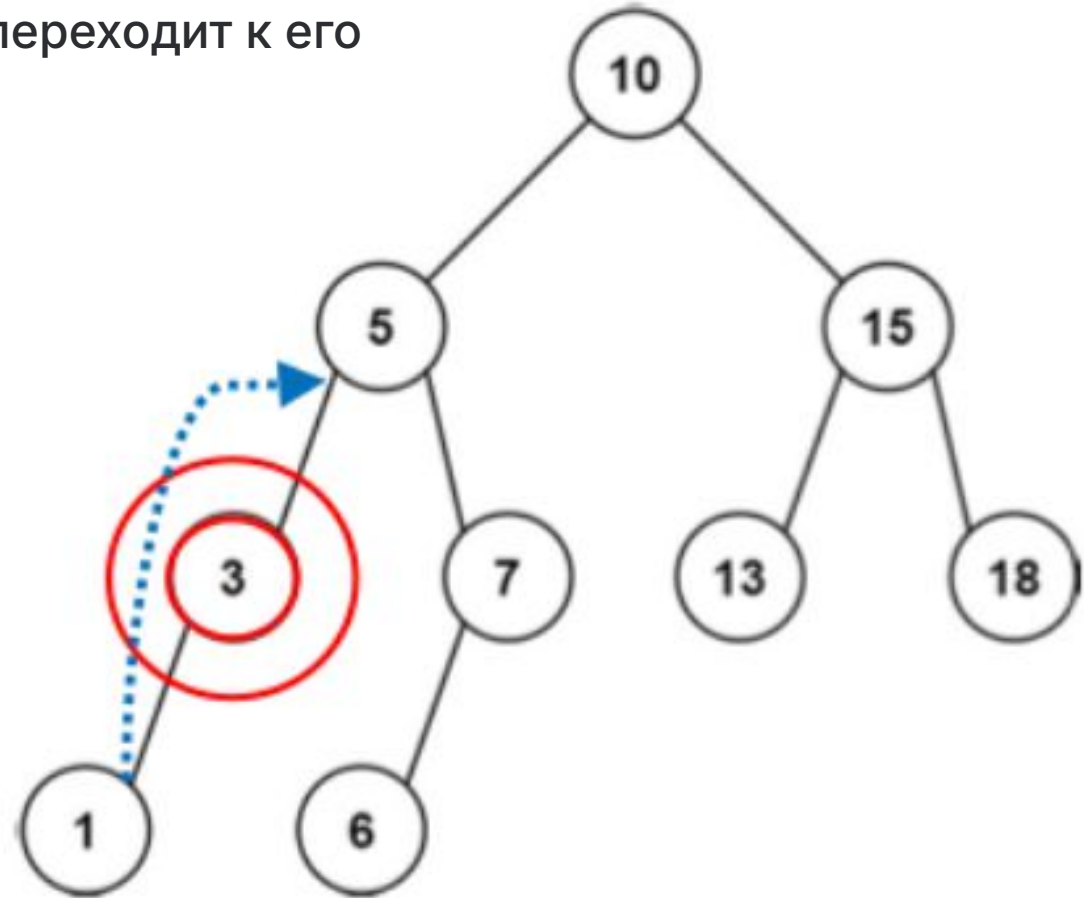
Узел не имеет потомков - узел является листом

В данном случае узел удаляется и узел с ключом 3 становится пустым.



Узел имеет только одного сына

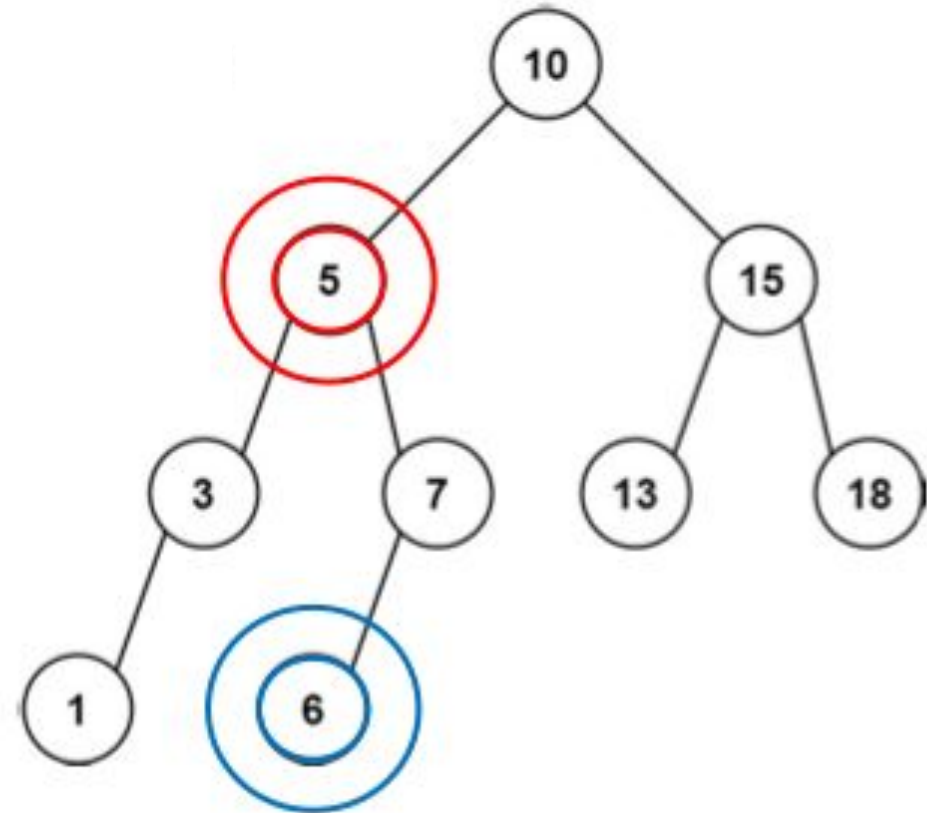
В данном случае сын удаляемого узла 3 переходит к его родителю - узлу 5.



Узел имеет два сына

Удаляем узел 5

- Необходимо найти минимальный элемент в правом дереве. Узел с ключом 6.
- Переносим ключ из узла 6 в узел с ключом 5.
- Сводим задачу к удалению узла с ключом 6.



Рефакторинг функции поиска узла в двоичном дереве и вспомогательная функция min_tree

```
tree* min_tree(tree *root) {  
    tree *find=root;  
    while(find && find->left){  
        find = find->left;  
    }  
    return find;  
}
```

```
tree * searchKey(tree *root, int  
key) {  
    if(!root)  
        return NULL;  
    if(root->key == key)  
        return root;  
    if(key < root->key )  
        return  
searchKey(root->left, key);  
    return searchKey(root->right,  
key);  
}
```

Поиск узла по ключу

Ещё одна процедура поиска, которая может потребоваться, — поиск узла со следующим по значению ключом, который имеется в дереве. Если правое поддереву текущего узла не пусто, то последователем узла будет самый левый узел его правого поддерева, потому что он имеет минимальный ключ среди всех ключей. (см. слайд 53)

```
tree* left_follower_key(tree *root) {
    if(root==NULL)
        return root;
    if(root->right)
        return min_tree(root->right);
    else {
        tree *y = root->parent;
        tree *x = root;
        while(y && x==y->right) {
            x = y;
            y = y->parent;
        }
        return y;
    }
}
```

Удаление узла

```
/* Удаление узла из дерева */
tree* delete(tree *root, tree* pt) {
    tree *remove = NULL, *remove_son;
    /* Нет потомков или один потомок.
    Удаляем сам узел */
    if(pt->left == NULL || pt->right ==
    NULL)
        remove = pt;
    else
        /* Два потомка. Удаляем его
        последователя. У последователя только один
        правый потомок, потому что это самый левый
        элемент правого поддерева.*/
        remove = left_follower_key(pt); //
    Всегда только один потомок
    /* Прикрепляем потомка удаляемого
    элемента к
    родителю, либо делаем этого потомка корнем
    дерева*/
```

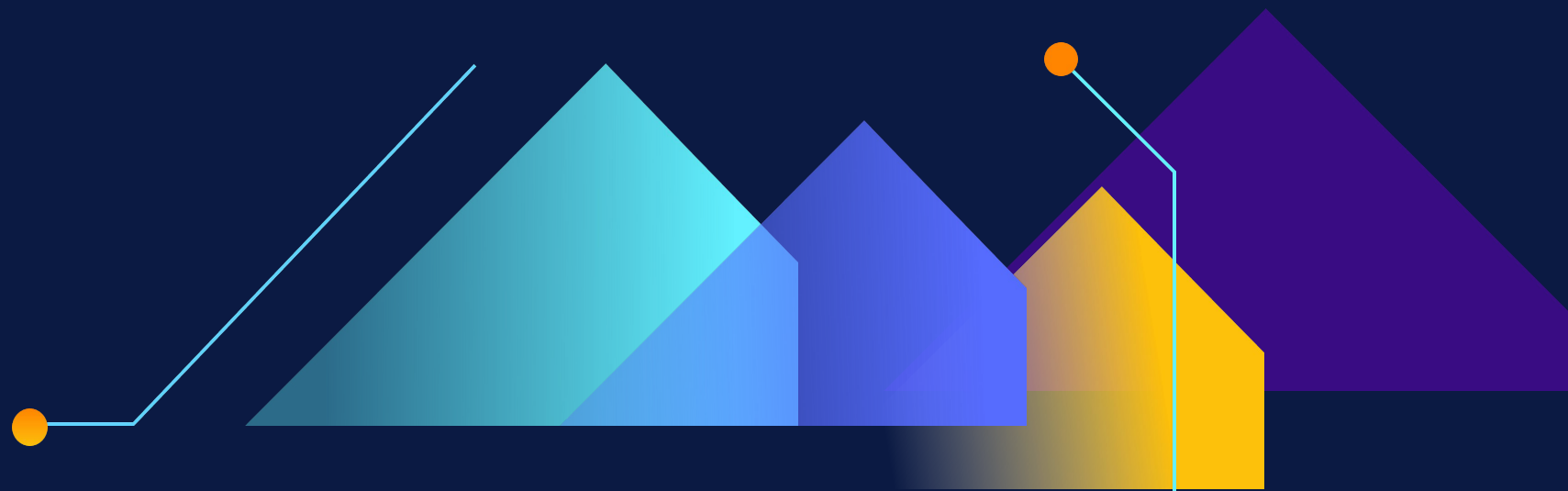
```
    if(remove -> left != NULL) // не
    сработает если это последователь pt
        remove_son = remove->left;
    else
        remove_son = remove->right;
    if( remove_son ) // обновляем родителя у
    удаляемого потомка
        remove_son->parent = remove->parent;
    if(remove->parent==NULL) // если удаляем
    корень
        root = remove_son;
    else if (remove == remove->parent->left)
        remove->parent->left = remove_son;
    else
        remove->parent->right = remove_son;
    if(pt != remove)
        pt->key = remove->key;
    return remove;
}
```

Пример удаления узла в двоичном дереве поиска

```
int main()
{
    tree *tr = NULL;
    insert(&tr, 10, NULL);
    insert(&tr, 5, NULL);
    insert(&tr, 15, NULL);
    insert(&tr, 3, NULL);
    insert(&tr, 7, NULL);
    insert(&tr, 13, NULL);
    insert(&tr, 18, NULL);
    insert(&tr, 1, NULL);
    insert(&tr, 6, NULL);
    printBFS(tr);
    printf("\n");
```

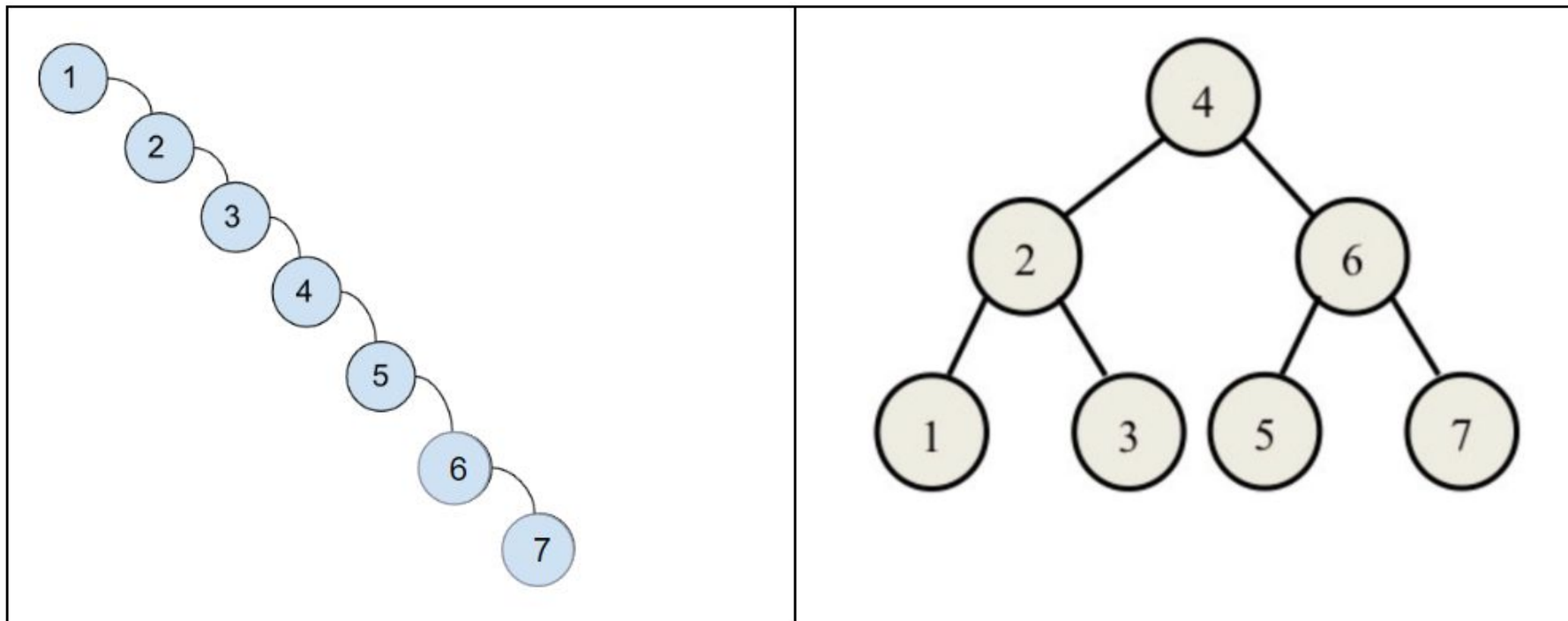
```
    preorder(tr);
    printf("\n");
    postorder(tr);
    printf("\n");
    inorder(tr);
    printf("\n");
    printBFS(searchKey(tr, 15));
    printf("\n");
    tree *del = NULL;
    del =
delete(tr, searchKey(tr, 15));
    free(del);
    inorder(tr);
    printf("\n");
    return 0;
}
```

Балансировка двоичных деревьев



Балансировка двоичных деревьев

Время выполнения базовых операций в дереве поиска линейно зависит от его высоты. Но из одного и того же набора ключей можно построить разные деревья поиска. Например, если мы имеем ключи: 1,2,3,4,5,6,7, то можно построить два таких дерева, причём оба будут корректными деревьями поиска.



Балансировки с известным набором ключей

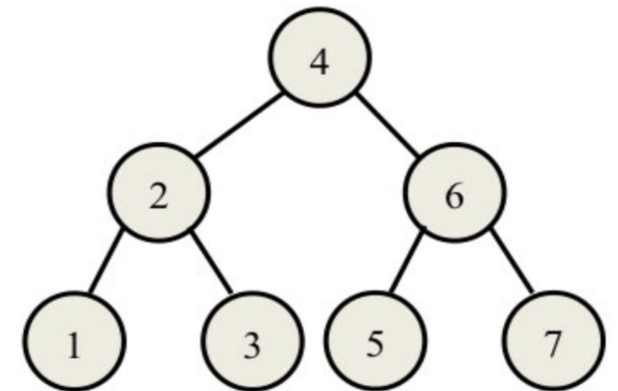
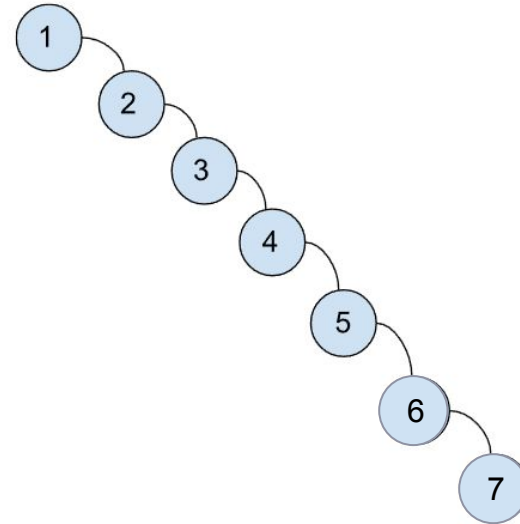
Для того чтобы построить дерево минимально возможной высоты, необходимо либо заранее знать, из каких ключей оно будет состоять, или выполнять балансировку дерева после добавления.

Балансировки с известным набором ключей

Если набор ключей известен заранее, то можно их упорядочить и каждый раз выбирать медиану. Например, для набора ключей:

1,2,3,4,5,6,7 в качестве корня выбираем 4, ключи меньше четырёх попадут в левое поддерево, ключи с большим корнем — в правое.

Для каждого из поддеревьев снова выбираем медиану из данного набора и строим соответствующие поддеревья.



Балансировки с известным набором ключей

Однако набор ключей не всегда известен заранее.

- Если ключи поступают по очереди, то построение дерева будет зависеть от порядка их поступления.
- Если при добавлении очередного узла количество узлов в левом и правом поддеревьях какого-либо узла дерева станет различаться более, чем на 1,
 - то дерево не будет являться идеально сбалансированным, и
 - его надо будет перестраивать, чтобы восстановить свойства идеально сбалансированного дерева поиска.

Виды сбалансированных деревьев поиска

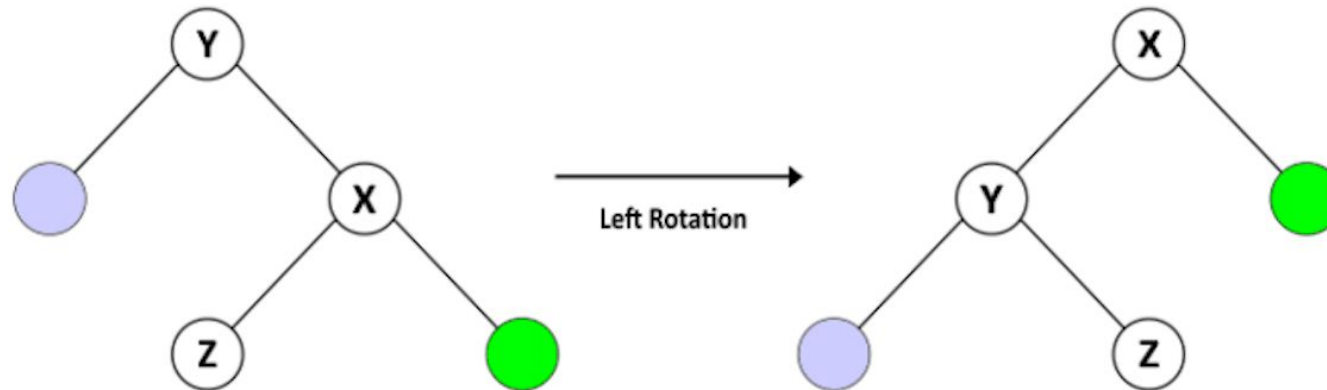
Рассмотрим некоторые виды сбалансированных деревьев поиска:

- **АВЛ-дерево.** Сбалансированность определяется разностью высот правого и левого поддеревьев любого узла. Если эта разность по модулю не превышает 1, то дерево считается сбалансированным.
- **Красно-черное дерево.** Каждый узел имеет дополнительное свойство — цвет, красный или черный.
- **SPLAY-дерево.** Иногда их называют самоперестраивающимся деревом. В отличие от двух предыдущих видов, у такого дерева нет никаких ограничений на расположение узлов, а сбалансированность в среднем достигается за счёт того, что каждый раз перед выполнением операции над узлом этот узел перемещается в корень дерева.

АВЛ-дерево

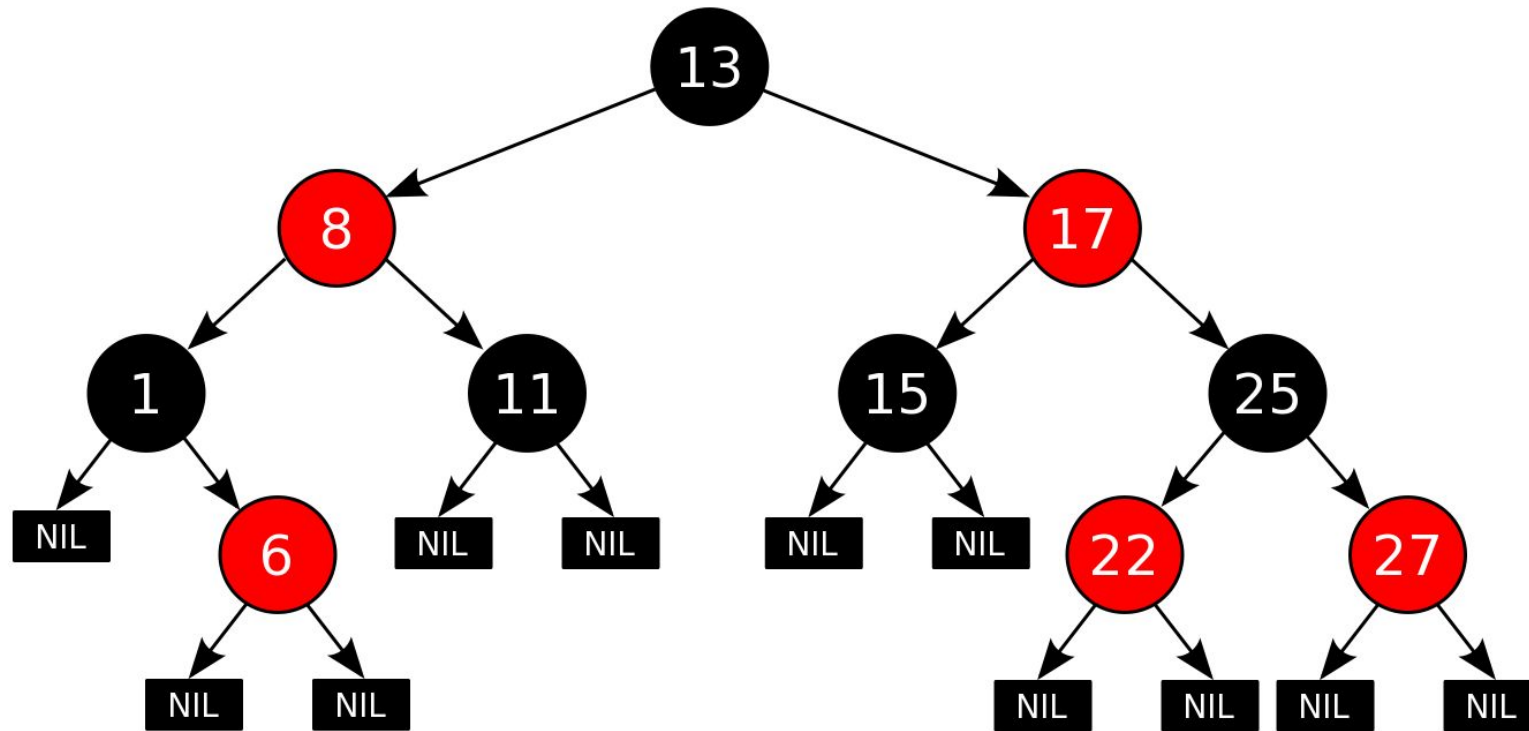
Сбалансированность определяется разностью высот правого и левого поддеревьев любого узла. Если эта разность по модулю не превышает 1, то дерево считается сбалансированным.

АВЛ — аббревиатура, образованная первыми буквами создателей (советских учёных) Адельсон-Вельского Георгия Максимовича и Ландиса Евгения Михайловича.



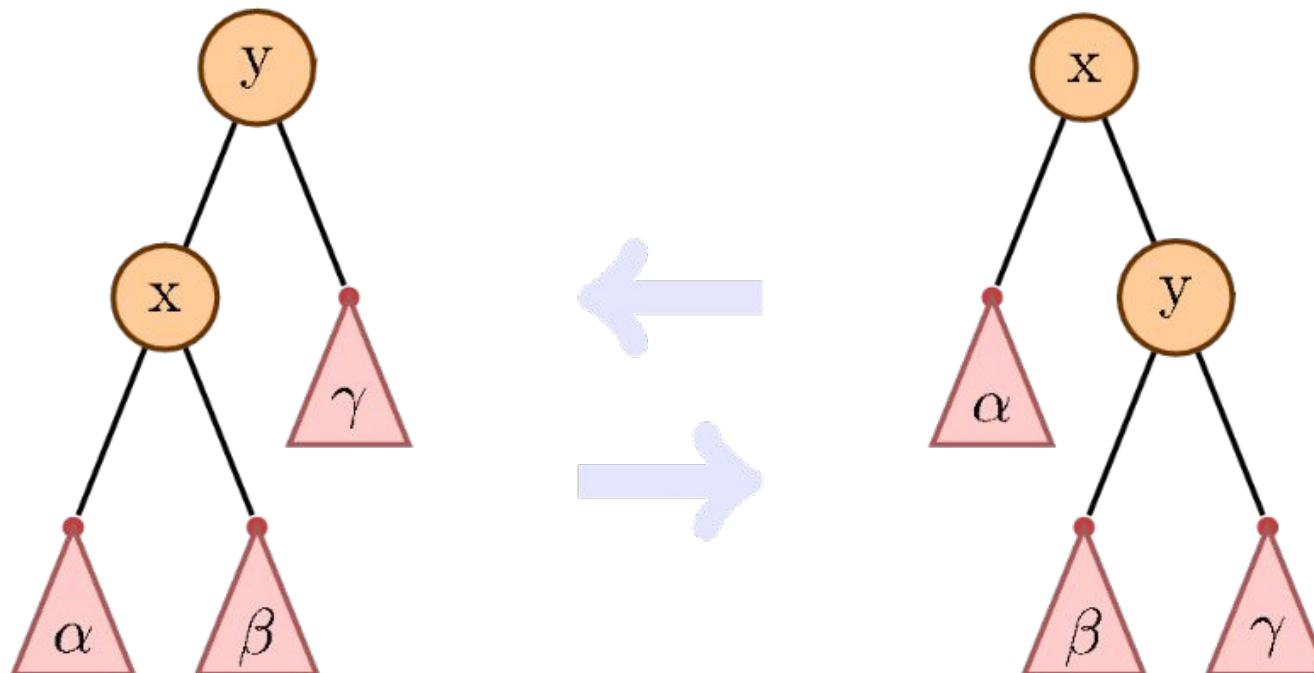
Красно-чёрное дерево

Каждый узел имеет дополнительное свойство — цвет, красный или чёрный. Корень и листья окрашены в чёрный цвет.

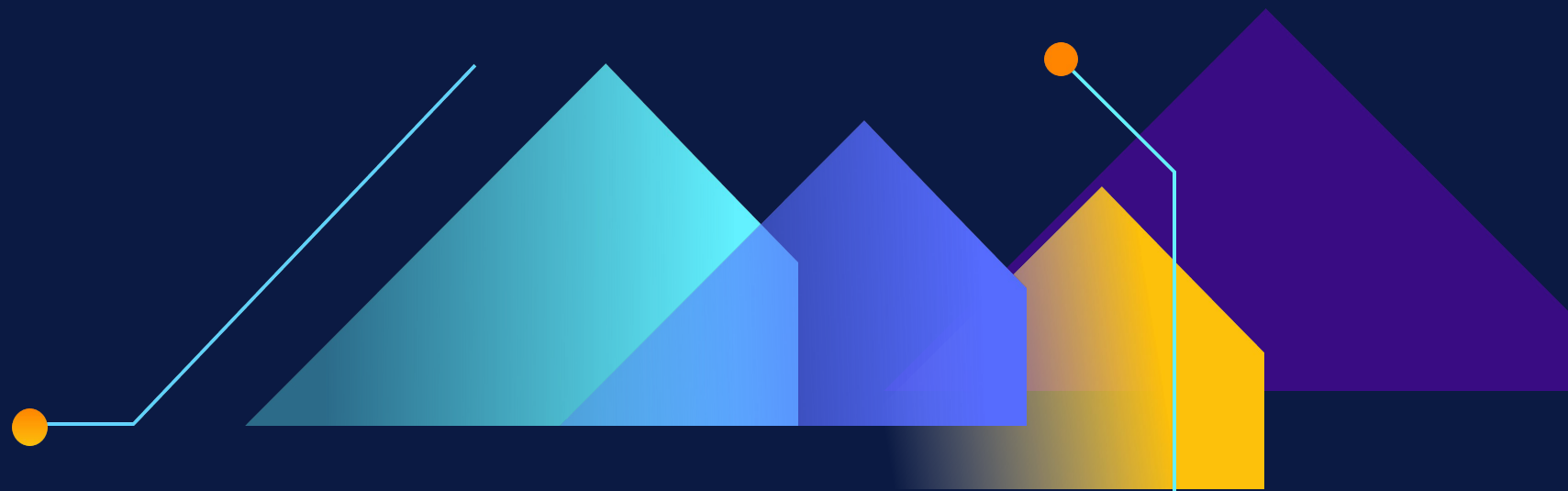


SPLAY-дерево

Иногда их называют самоперестраивающимся деревом. В отличие от двух предыдущих видов, у такого дерева нет никаких ограничений на расположение узлов, а сбалансированность в среднем достигается за счёт того, что каждый раз перед выполнением операции над узлом этот узел перемещается в корень дерева.



АВЛ-деревья



АВЛ-деревья

АВЛ-деревом называется такое дерево поиска, в котором для любого его узла высоты левого и правого поддеревьев отличаются не более, чем на единицу.

АВЛ-деревья

Для наглядности и простоты алгоритма будем хранить высоту в отдельном поле **height**.

*Традиционно, узлы АВЛ-дерева хранят не высоту, а разницу высот правого и левого поддеревьев (так называемый *balance factor*), которая может принимать только три значения -1, 0 и 1. В целом это не сильно влияет на объём занимаемых данных. Даже если хранить только баланс-фактор в однобайтовой переменной, то из-за выравнивания полей структуры по четырёхбайтной границе экономии не будет.*

```
struct node {  
    int key;  
    uint32_t height;  
    struct node *left;  
    struct node *right;  
};
```

Вставка узла

Вставка узла аналогична вставке в обычное дерево поиска:

```
struct node* insert(struct node *p, int k) {  
    if( p==NULL ) {  
        p=malloc(sizeof(struct node));  
        p->key=k;  
        p->height=0;  
        p->left = p->right=NULL;  
        return p;  
    }  
    if( k < p->key )  
        p->left = insert( p->left,k);  
    else  
        p->right = insert( p->right,k);  
    return balance(p);  
}
```

Пример

Для работы определим три вспомогательные функции:

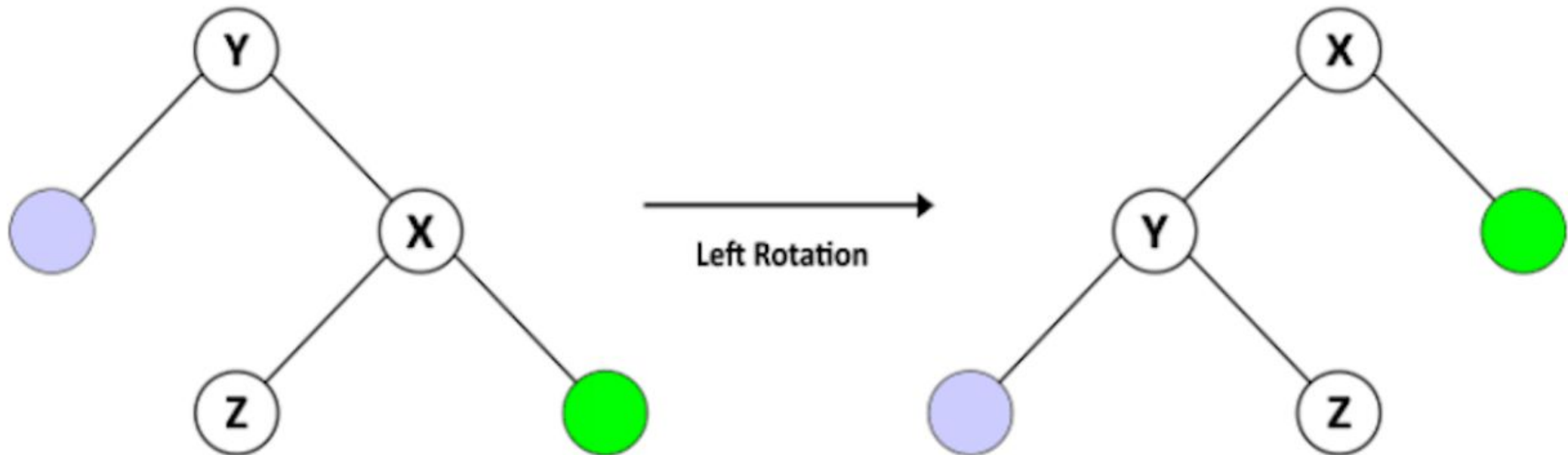
```
/* Высота поддерева */
uint32_t height(struct node* p)
{
    return p ? p->height : 0;
}
```

```
/* Вычисляем баланс фактор узла */
int bfactor(struct node* p) {
    return
    height(p->right) - height(p->left);
}
```

```
/* Восстанавливаем корректно значение высоты */
void fixheight(struct node* p)
{
    uint32_t hl = height(p->left);
    uint32_t hr = height(p->right);
    p->height = (hl > hr ? hl : hr) + 1;
}
```

Корректировка разбалансировки поддерева

В процессе добавления или удаления узлов в дереве возможно возникновение ситуации, когда balance factor некоторых узлов оказывается равными 2 или -2, т.е. возникает разбалансировка поддерева. Для корректировки ситуации применяются повороты вокруг узлов дерева. Простой поворот влево(вправо):



Простой поворот влево(вправо)

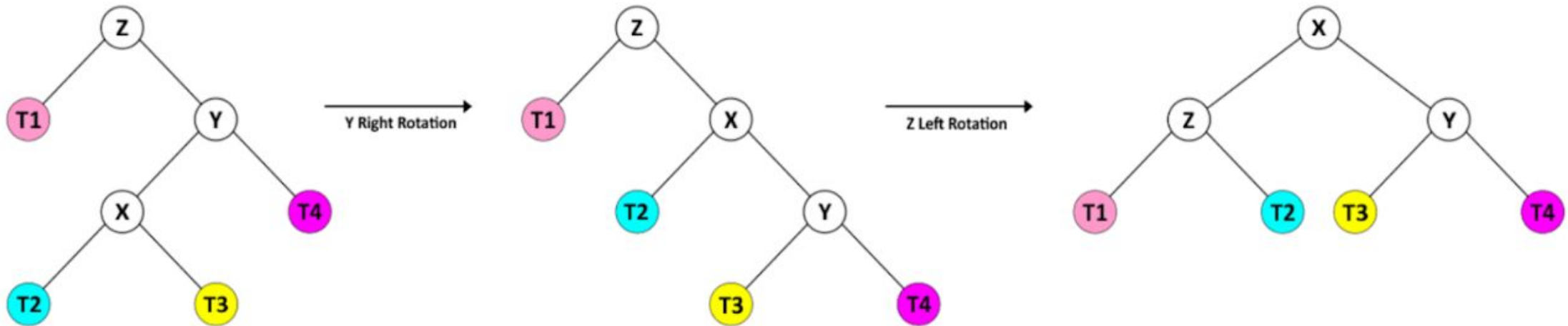
Функции поворотов и балансировки не содержат ни циклов, ни рекурсии, а значит выполняются за постоянное время, не зависящее от размера дерева. Код функций, реализующих данные повороты:

```
/* левый поворот вокруг Y */
struct node* rotateleft(struct
node* Y) {
    struct node* X = Y->right;
    Y->right = X->left;
    X->left = Y;
    fixheight(Y);
    fixheight(X);
    return X;
}
```

```
/* правый поворот вокруг X */
struct node* rotateright(struct
node* X) {
    struct node* Y = X->left;
    X->left = Y->right;
    Y->right = X;
    fixheight(X);
    fixheight(q);
    return Y;
}
```

Корректировка разбалансировки поддерева

Может возникнуть ситуация дисбаланса, когда высота правого поддерева узла Z на 2 больше высоты левого поддерева (обратный случай является симметричным и реализуется аналогично). $h(T1) - h(Y) = 2$.



В этом случае необходимо применить большой поворот вокруг Z. В данном случае он сводится к двум простым — сначала правый поворот вокруг Y, затем левый вокруг Z.

Балансировка узла

Балансировка реализуется следующим кодом:

```
/* балансировка узла p */
struct node* balance(struct node* p) {
    fixheight(p);
    if( bfactor(p)==2 ) {
        if( bfactor(p->right) < 0 )
            p->right = rotateright(p->right);
        return rotateleft(p);
    }
    if( bfactor(p)==-2 ) {
        if( bfactor(p->left) > 0 )
            p->left = rotateleft(p->left);
        return rotateright(p);
    }
    return p; // балансировка не нужна
}
```


Удаление элемента

Удаление элемента из дерева аналогично удалению из любого двоичного дерева поиска. Разница заключается в необходимости выполнять балансировку после различных удалений.

```
/* удаление ключа k из дерева p */
struct node *removenode(struct node *p, int k)
{
    /* ищем нужный ключ */
    if( !p ) return 0; // если не нашли
    if( k < p->key )
        p->left = removenode(p->left,k);
    else if( k > p->key )
        p->right = removenode(p->right,k);
    else // k == p->key
    {
        struct node* q = p->left;
        struct node* r = p->right;
        free(p);
    }
}
```

Удаление элемента

Удаление элемента из дерева аналогично удалению из любого двоичного дерева поиска. Разница заключается в необходимости выполнять балансировку после различных удалений.

```
    /* Если правого поддеревя нет то по свойству AVL-дерева
слева у этого узла может быть только один единственный дочерний узел
(дерево высоты 1), либо узел p вообще лист */
    if( !r ) return q;
    /* Находим последователя. Извлекаем его оттуда, слева к min
подвешиваем q, справа — то, что получилось из r, возвращаем min
после его балансировки. */
    struct node* min = findmin(r);
    min->right = removemin(r);
    min->left = q;
    return balance(min);
}
return balance(p);
}
```

Используемые источники

- The Art of Computer Programming, vol.1. Fundamental Algorithms.
- Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ
- [Статья про AVL-деревья](#)
- [Ещё одна статья про AVL-деревья](#)
- D. D. Sleator, R.E. Tarjan. Self-Adjusting Binary Search Trees