Лабораторная работа № 6

Тема: *Деревья. Сбалансированные по высоте деревья (АВЛ-деревья). 2-3 деревья. Б-деревья. Красно-черные деревья. Практическое применение.*

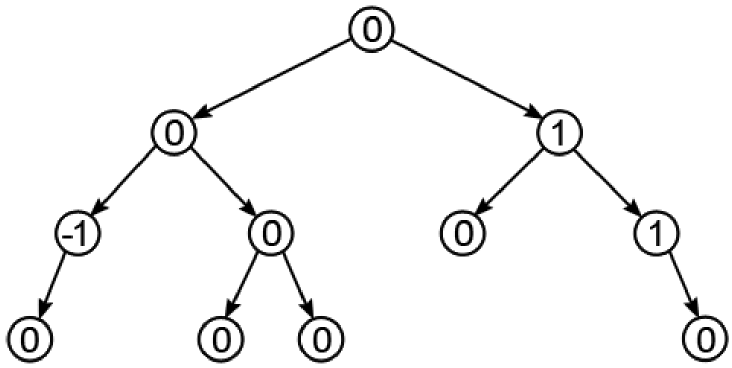
Цель: *ознакомиться с понятиями «Деревья», «АВЛ-деревья», «Б-деревья», «Красно-черные деревья», изучить основные алгоритмы их обработки, научиться применять полученные знания на практике.*

1 Краткая теория

1.1 АВЛ-ДЕРЕВЬЯ

Сбалансированным называется такое двоичное дерево поиска, в котором высота каждого из поддеревьев, имеющих общий корень, отличается не более чем на некоторую константу **k**, и при этом выполняются условия характерные для двоичного дерева поиска.

АВЛ-дерево – сбалансированное двоичное дерево поиска с k=1. Для его узлов определен **коэффициент сбалансированности** (balance factor). Balance factor – это разность высот правого и левого поддеревьев, принимающая одно значение из множества {-1, 0, 1}. Ниже изображен пример АВЛ-дерева, каждому узлу которого поставлен в соответствие его реальный коэффициент сбалансированности.



Положим B_i – коэффициент сбалансированности узла T_i (i – номер узла, отсчитываемый сверху вниз от корневого узла по уровням слева направо). Balance factor узла T_i рассчитывается следующим образом. Пусть функция h() с параметрами T_i и L возвращает высоту левого поддерева L узла T_i, а с T_i и R – правого. Тогда B_i=h(T_i, R)-h(T_i, L). Например, B_4=-1, так как h(T_4, R)-h(T_4, L)=0-1=-1.

Сбалансированное дерево эффективно в обработке, что следует из следующих рассуждений. Максимальное количество шагов, которое может потребоваться для обнаружения нужного узла, равно количеству уровней самого бинарного дерева поиска. А так как поддеревья сбалансированного дерева, «растущие» из произвольного корня, практически симметричны, то и его листья расположены на сравнительно невысоком уровне, т. е. высота дерева сводиться к оптимальному минимуму. Поэтому критерий баланса положительно сказывается на общей производительности. Но в процессе обработки АВЛ-дерева, балансировка может нарушиться, тогда потребуется осуществить операцию балансировки. Помимо нее, над АВЛ-деревом определены операции вставки и удаления элемента. Именно выполнение последних может привести к дисбалансу дерева.

Доказано, что высота АВЛ-дерева, имеющего **N** узлов, примерно равна $\mathbf{log_2N}$. Беря в виду это, а также то, то, что время выполнения операций добавления и удаления напрямую зависит от операции поиска, получим временную сложность трех операций для худшего и среднего случая – **O(logN)**.

Прежде чем рассматривать основные операции над АВЛ-деревом, определим структуру для представления его узлов, а также три специальные функции:

```
struct Node
{
int key;
char height;
Node *right;
Node *left;
Node(int k) { key=k; height=1; left=right=0; }
};
char height(Node *p)
{
if (p) return p->height;
else return 0;
}
int BF(Node *p)
{ return height(p->right)-height(p->left); }
void OverHeight(Node *p)
{
char hleft=height(p->left);
char hright=height(p->right);
p->height=(hleft>hright ? hleft : hright)+1;
}
```

Структура Node описывает узлы АВЛ-дерева. Ее поля right и left являются указателями на правое и левое поддеревья. Поле key хранит ключ узла, height – высоту поддерева. Функция-конструктор создает новый узел. Функции height и BF вычисляют коэффициент сбалансированности узла, а OverHeight – корректирует значение поля height, затронутое в процессе балансировки.

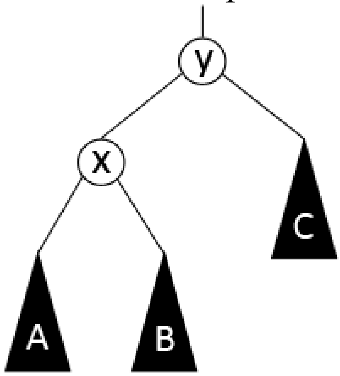
__Балансировка.__

Если после выполнения операции добавления или удаления, коэффициент сбалансированности какого-либо узла АВЛ-дерева становиться равен 2, т. е. $|h(T_i, R)-h(T_i, L)|=2$, то необходимо выполнить операцию балансировки. Она осуществляется путем вращения (поворота) узлов – изменения связей в поддереве. Вращения не меняют свойств бинарного дерева поиска, и выполняются за константное время. Всего различают 4 их типа:

1. малое правое вращение;
2. большое правое вращение;
3. малое левое вращение;
4. большое левое вращение.

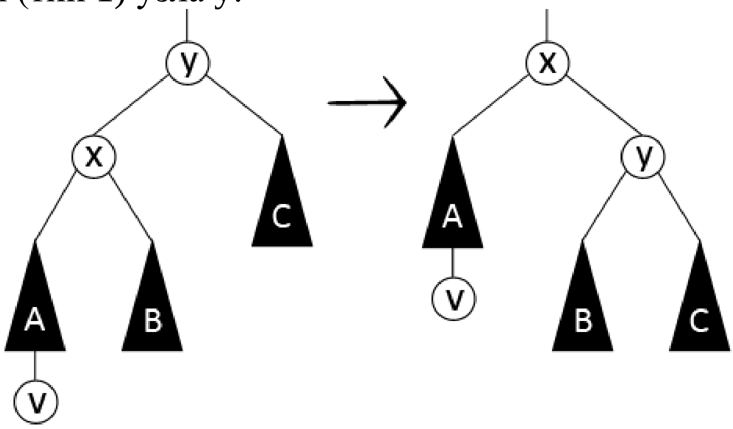
Оба типа больших вращений являются комбинацией малых вращений (право-левым или лево-правым вращением).

Возможны два случая нарушения сбалансированности. Первый из них исправляется 1 и 3 типом, а второй – 2 и 4. Рассмотрим первый случай. Пусть имеется следующее сбалансированное поддерево:



Поддерево. Случай 1

Здесь x и y – узлы, а A, B, C – поддеревья. После добавления к поддереву A узла v, баланс нарушиться, и потребуется балансировка. Она осуществляется правым поворотом (тип 1) узла y:

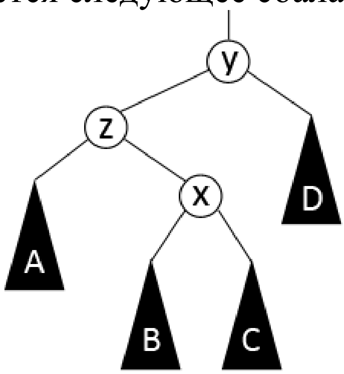


Малый правый поворот

Малое левое вращение выполняется симметрично малому правому. Следующие две функции выполняют малый правый и малый левый повороты.

```
Node* RightRotation(Node *x)
{
Node *y=x->left;
x->left=y->right;
y->right=x;
OverHeight(x);
OverHeight(y);
return y;
}
Node *LeftRotation(Node *y)
{
Node *x=y->right;
y->right=x->left;
x->left=y;
OverHeight(y);
OverHeight(x);
return x;
}
```

Второй случай дисбаланса исправляется большим правым или большим левым вращением. Пусть имеется следующее сбалансированное поддерево:



Большое левое вращение выполняется симметрично большому правому. Функция Balance выполняет балансировку узла путем вращения его поддеревьев:

```
Node *Balance(Node *x)
{
OverHeight(x);
if (BF(x)==2)
{
if (BF(x->right)<0) x->right=RightRotation(x->right);
return LeftRotation(x);
}
if (BF(x)==-2)
{
if (BF(x->left)>0) x->left=LeftRotation(x->left);
return RightRotation(x);
}
return x;
}
```

Данная функция проверяет условия, и в зависимости от результата балансирует узел x, применяя один из типов вращения.

<u>Добавление узлов</u>

Операция вставки нового узла в АВЛ-дерево выполняется рекурсивно. По ключу данного узла производится поиск места вставки: спускаясь вниз по дереву, алгоритм сравнивает ключ добавляемого узла со встречающимися ключами, далее происходит вставка нового элемента; по возвращению из рекурсии, выполняется проверка всех показателей сбалансированности узлов и, в случае необходимости, выполняется балансировка. Для осуществления балансировки следует знать, с каким из рассмотренных выше случаев дисбаланса имеем дело. Допустим, мы добавили узел **x** в левое поддерево, для которого выполнялось $h(T_i, R) < h(T_i, L)$, т. е. высота левого поддерева изначально превышала высоту правого. Если левое поддерево этого узла выше правого, то потребуется большое вращение, иначе – малое.

Функция добавления узла:

```
Node *Insert(Node *x, int k)
{
if (!x) return new Node(k);
if (k<x->key) x->left=Insert(x->left, k);
else x->right=Insert(x->right, k);
return Balance(x);
}
```

<u>Удаление узлов.</u>

Также как и вставку узла, его удаление удобно задать рекурсивно. Пусть **x** – удаляемый узел, тогда если **x** – лист (терминальный узел), то алгоритм удаления сводиться к простому исключению узла **x**, и подъему к корню с переопределением balance factor'ов узлов. Если же **x** не является листом, то он либо имеет правое поддерево, либо не имеет его. Во втором случае, из свойства АВЛ-дерева, следует, что левое поддерево имеет высоту 1, и здесь алгоритм удаления сводиться к тем же действиям, что и при терминальном узле. Остается ситуация когда у **x** есть правое поддерево. В таком случае нужно в правом поддереве отыскать следующий по значению за **x** узел **y**, заменить **x** на **y**, и рекурсивно вернуться к корню, переопределяя коэффициенты сбалансированности узлов. Из свойства двоичного дерева поиска следует, что узел **y** имеет наименьшее значение среди всех узлов правого поддерева узла **x**.

Для программной реализации операции удаления узла опишем функцию Delete:

```
Node *Delete(Node *x, int k)
{
if (!x) return 0;
if (k<x->key) x->left=Delete(x->left, k);
else if (k>x->key) x->right=Delete(x->right, k);
else
{
Node *y=x->left;
Node *z=x->right;
delete x;
if (!z) return y;
Node* min=SearchMin(z);
min->right=DeleteMin(z);
min->left=y;
return Balance(min);
}
return Balance(x);
}
```

Из нее вызываются вспомогательные функции: SearchMin и DeleteMin. Первая ищет минимальный элемент в правом поддереве, вторая удаляет его. Опишем эти вспомогательные функции:

```
Node *SearchMin(Node *x)
{
if (x->left) return SearchMin(x->left);
else return x;
}
Node *DeleteMin(Node *x)
{
if (x->left==0) return x->right;
x->left=DeleteMin(x->left);
return Balance(x);
}
```

Операция удаления реализуется определенно сложнее, чем операция добавления. Да и последствия ее выполнения могут потребовать поворота в каждом узле.

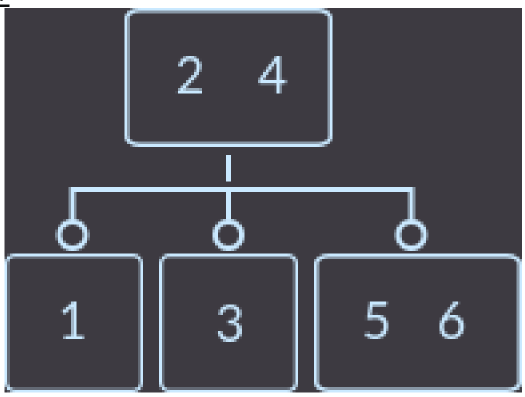
1.2 2-3 ДЕРЕВЬЯ

2-3-дерево — структура данных, являющаяся B-деревом Степени 1, страницы которого могут содержать только 2-вершины (вершины с одним полем и 2 детьми) и 3-вершины (вершины с 2 полями и 3 детьми). Листовые вершины являются исключением — у них нет детей (но может быть одно или два поля). 2-3-деревья сбалансированы, то есть, каждое левое, правое, и центральное поддерево имеет одну и ту же высоту, и, таким образом, содержат равные (или почти равные) объемы данных.

Свойства:

- Все нелистовые вершины содержат одно поле и 2 поддерева или 2 поля и 3 поддерева.
- Все листовые вершины находятся на одном уровне (на нижнем уровне) и содержат 1 или 2 поля.
- Все данные отсортированы (по принципу двоичного дерева поиска).
- Нелистовые вершины содержат одно или два поля, указывающие на диапазон значений в их поддеревьях. Значение первого поля строго больше наибольшего значения в левом поддереве и меньше или равно наименьшему значению в правом поддереве (или в центральном поддереве, если это 3-вершина); аналогично, значение второго поля (если оно есть) строго больше наибольшего значения в центральном поддереве и меньше или равно, чем наименьшее значение в правом поддереве. Эти нелистовые вершины используются для направления функции поиска к нужному поддереву и, в конечном итоге, к нужному листу. (*прим. Это свойство не будет выполняться, если у нас есть одинаковые ключи. Поэтому возможна ситуация, когда равные ключи находятся в левом и правом поддеревьях одновременно, тогда ключ в нелистовой вершине будет совпадать с этими ключами. Это никак не сказывается на правильности работы и производительности алгоритма.*).

<u>Пример 2-3-дерева:</u>



<u>Представление дерева в коде</u>

Вершины дерева представим в виде 4-вершин (это когда вершина может иметь 3 ключа и 4 сына). Данное решение было выбрано по нескольким причинам: во-первых, так легче сделать наивную (прямую) реализацию, а во-вторых, код и так сильно обрамлен if'ами и это решение позволило уменьшить количество проверок и упростить код.

Вот так выглядит представление вершины:

```
struct node {
private:
    int size;       // количество занятых ключей
    int key[3];
    node *first;    // *first <= key[0];
    node *second;   // key[0] <= *second < key[1];
    node *third;    // key[1] <= *third < key[2];
    node *fourth;   // kye[2] <= *fourth.
```

```
        node *parent; //Указатель на родителя нужен для того, потому что адрес корня может меняться при удалении
        bool find(int k) { // Этот метод возвращает true, если ключ k находится в вершине, иначе false.
            for (int i = 0; i < size; ++i)
                if (key[i] == k) return true;
            return false;
        }
        void swap(int &x, int &y) {
            int r = x;
            x = y;
            y = r;
        }
        void sort2(int &x, int &y) {
            if (x > y) swap(x, y);
        }
        void sort3(int &x, int &y, int &z) {
            if (x > y) swap(x, y);
            if (x > z) swap(x, z);
            if (y > z) swap(y, z);
        }
        void sort() { // Ключи в вершинах должны быть отсортированы
            if (size == 1) return;
            if (size == 2) sort2(key[0], key[1]);
            if (size == 3) sort3(key[0], key[1], key[2]);
        }
        void insert_to_node(int k) {  // Вставляем ключ k в вершину (не в дерево)
            key[size] = k;
            size++;
            sort();
        }
        void remove_from_node(int k) { // Удаляем ключ k из вершины (не из дерева)
            if (size >= 1 && key[0] == k) {
                key[0] = key[1];
                key[1] = key[2];
                size--;
            } else if (size == 2 && key[1] == k) {
                key[1] = key[2];
                size--;
            }
        }
        void become_node2(int k, node *first_, node *second_) {  // Преобразовать в 2-вершину.
            key[0] = k;
            first = first_;
            second = second_;
            third = nullptr;
            fourth = nullptr;
            parent = nullptr;
            size = 1;
        }
```

```
        bool is_leaf() { // Является ли узел листом; проверка используется при вставке и удалении.
            return (first == nullptr) && (second == nullptr) && (third == nullptr);
        }
    public:
        // Создавать всегда будем вершину только с одним ключом
        node(int k): size(1), key{k, 0, 0}, first(nullptr), second(nullptr),
                     third(nullptr), fourth(nullptr), parent(nullptr) {}
        node (int k, node *first_, node *second_, node *third_, node *fourth_, node *parent_):
                     size(1), key{k, 0, 0}, first(first_), second(second_),
                     third(third_), fourth(fourth_), parent(parent_) {}
        friend node *split(node *item); // Метод для разделение вершины при переполнении;
        friend node *insert(node *p, int k); // Вставка в дерево;
        friend node *search(node *p, int k); // Поиск в дереве;
        friend node *search_min(node *p); // Поиск минимального элемента в поддереве;
        friend node *merge(node *leaf); // Слияние используется при удалении;
        friend node *redistribute(node *leaf); // Перераспределение также используется при удалении;
        friend node *fix(node *leaf); // Используется после удаления для возвращения свойств дереву (использует merge или redistribute)
        friend node *remove(node *p, int k); // Собственно, из названия понятно;
    };
```

Вставка

Для того, чтобы вставить в дерево элемент с ключом key, нужно действовать по алгоритму:

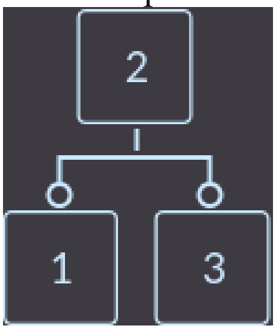
1. Если дерево пусто, то создать новую вершину, вставить ключ и вернуть в качестве корня эту вершину, иначе
2. Если вершина является листом, то вставляем ключ в эту вершину и если получили 3 ключа в вершине, то разделяем её, иначе
3. Сравниваем ключ key с первым ключом в вершине, и если key меньше данного ключа, то идем в первое поддерево и переходим к пункту 2, иначе
4. Смотрим, если вершина содержит только 1 ключ (является 2-вершиной), то идем в правое поддерево и переходим к пункту 2, иначе
5. Сравниваем ключ key со вторым ключом в вершине, и если key меньше второго ключа, то идем в среднее поддерево и переходим к пункту 2, иначе
6. Идем в правое поддерево и переходим к пункту 2.

Для примера вставим keys = {1, 2, 3, 4, 5, 6, 7}: При вставке key=1 мы имеем пустое дерево, а после получаем единственную вершину с единственным ключа key=1:

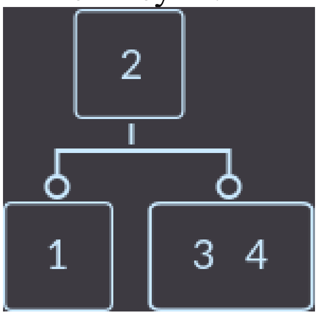
Дальше вставляем key=2:

Теперь вставляем key=3 и получаем вершину, содержащую 3 ключа:

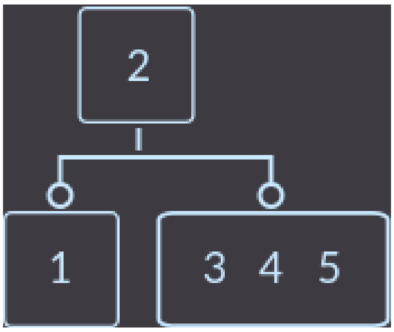
Так как эта вершина нарушает свойства 2-3-дерева, то мы должны с этим разобраться. А разберемся с этим вот таким разделением: ключ, который находится в середине (в нашем случае key=2), перенесем в родительскую вершину на соответствующее место, либо если у нас единственная вершина в дереве, то она соответственно является и корнем дерева — значит, мы создаем новую вершину и переносим туда средний ключ key = 2, а остальные ключи сортируем и делаем их сыновьями нового корня:



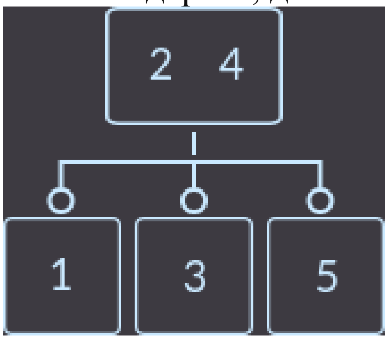
Дальше по алгоритму вставляем key=4:



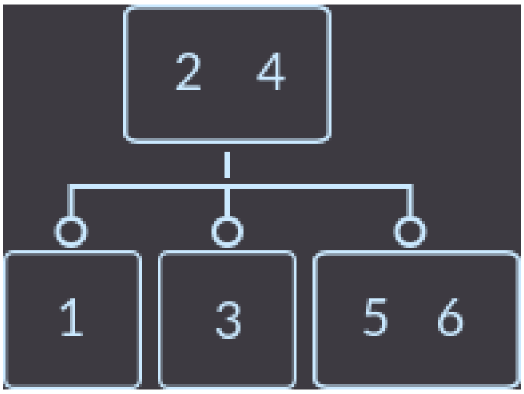
Key=5:



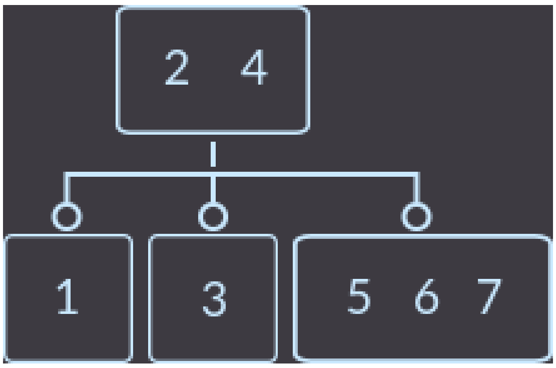
Снова нарушилось свойства 2-3 дерева, делаем разделение:



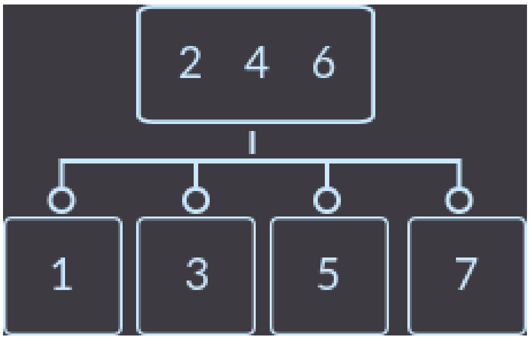
Key=6:



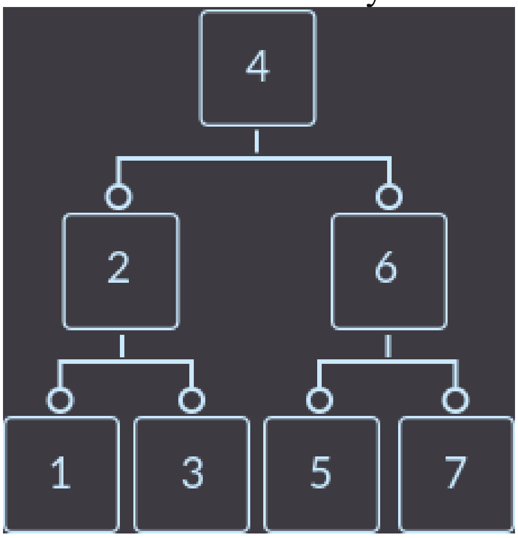
Key=7:



Теперь нам предстоит сделать два разделения, т.к. вершина, в которую вставили новый ключ теперь имеет 3 ключа (сначала разделим ее):



А теперь и корень имеет 3 вершины — разделим его и получим сбалансированное дерево, которое при таких входных данных с обычным двоичным деревом поиска мы бы не смогли получить:



Вставка ключа

```
node *insert(node *p, int k) { // вставка ключа k в дерево с корнем p; всегда возвращаем корень дерева, т.к. он может меняться
    if (!p) return new node(k); // если дерево пусто, то создаем первую 2-3-вершину (корень)

    if (p->is_leaf()) p->insert_to_node(k);
    else if (k <= p->key[0]) insert(p->first, k);
    else if ((p->size == 1) || ((p->size == 2) && k <= p->key[1])) insert(p->second, k);
    else insert(p->third, k);
    return split(p);
}
```

Разделение вершины

```
node *split(node *item) {
    if (item->size < 3) return item;
```

```
        node *x = new node(item->key[0], item->first, item->second, nullptr, nullptr, item->parent); // Создаем две новые вершины,
        node *y = new node(item->key[2], item->third, item->fourth, nullptr, nullptr, item->parent);   // которые имеют такого же родителя, как и разделяющийся элемент.
        if (x->first)  x->first->parent = x;    // Правильно устанавливаем "родителя" "сыновей".
        if (x->second) x->second->parent = x;    // После разделения, "родителем" "сыновей" является "дедушка",
        if (y->first)  y->first->parent = y;    // Поэтому нужно правильно установить указатели.
        if (y->second) y->second->parent = y;
        if (item->parent) {
            item->parent->insert_to_node(item->key[1]);
            if (item->parent->first == item) item->parent->first = nullptr;
            else if (item->parent->second == item) item->parent->second = nullptr;
            else if (item->parent->third == item) item->parent->third = nullptr;
            // Дальше происходит своеобразная сортировка ключей при разделении.
            if (item->parent->first == nullptr) {
                item->parent->fourth = item->parent->third;
                item->parent->third = item->parent->second;
                item->parent->second = y;
                item->parent->first = x;
            } else if (item->parent->second == nullptr) {
                item->parent->fourth = item->parent->third;
                item->parent->third = y;
                item->parent->second = x;
            } else {
                item->parent->fourth = y;
                item->parent->third = x;
            }
            node *tmp = item->parent;
            delete item;
            return tmp;
        } else {
            x->parent = item;   // Так как в эту ветку попадает только корень,
            y->parent = item;       // то мы "родителем" новых вершин делаем разделяющийся элемент.
            item->become_node2(item->key[1], x, y);
            return item;
        }
    }
```

Поиск

Поиск такой же простой, как и в бинарном дереве поиска:

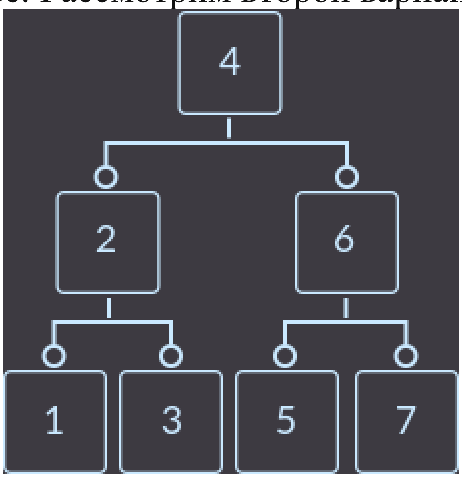
1. Ищем искомый ключ key в текущей вершине, если нашли, то возвращаем вершину, иначе
2. Если key меньше первого ключа вершины, то идем в левое поддерево и переходим к пункту 1, иначе
3. Если в дереве 1 ключ, то идем в правое поддерево (среднее, если руководствоваться нашим классом) и переходим к пункту 1, иначе
4. Если key меньше второго ключа вершины, то идем в среднее поддерево и переходим к пункту 1, иначе
5. Идем в правое поддерево и переходим к пункту 1.

Поиск ключа

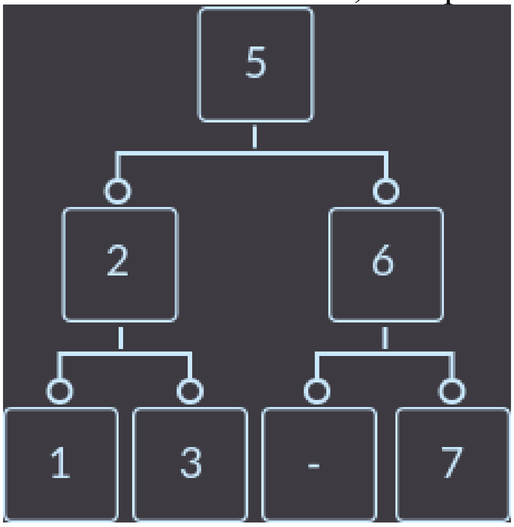
```
node *search(node *p, int k) { // Поиск ключа k в 2-3 дереве с корнем p.
   if (!p) return nullptr;
   if (p->find(k)) return p;
   else if (k < p->key[0]) return search(p->first, k);
   else if ((p->size == 2) && (k < p->key[1]) || (p->size == 1)) return search(p->second, k);
   else if (p->size == 2) return search(p->third, k);
}
```

Удаление

Удаление в 2-3-дереве, как и в любом другом дереве, происходит только из листа (из самой нижней вершины). Поэтому, когда мы нашли ключ, который нужно удалить, сначала надо проверить, находится ли этот ключ в листовой или нелистовой вершине. Если ключ находится в нелистовой вершине, то нужно найти эквивалентный ключ для удаляемого ключа из листовой вершины и поменять их местами. Для нахождения эквивалентного ключа есть два варианта: либо найти максимальный элемент в левом поддереве, либо найти минимальный элемент в правом поддереве. Рассмотрим второй вариант. Например:



Чтобы удалить из дерева ключ key=4, для начала нужно найти эквивалентный эму элемент и поменять местами: это либо key=3, либо key=5. Так выбран второй способ, то меняем ключи key=4 и key=5 местами и удаляем key=4 из листа («тире» будет обозначать место ключа, который удален):



Поиска минимального ключа

```
node *search_min(node *p) { // Поиск узла с минимальным элементов в 2-3-дереве с корнем p.
    if (!p) return p;
    if (!(p->first)) return p;
    else return search_min(p->first);
}
```

Удаление ключа

```
node *remove(node *p, int k) { // Удаление ключа k в 2-3-дереве с корнем p.
    node *item = search(p, k); // Ищем узел, где находится ключ k
    if (!item) return p;
    node *min = nullptr;
    if (item->key[0] == k) min = search_min(item->second); // Ищем эквивалентный ключ
    else min = search_min(item->third);
    if (min) { // Меняем ключи местами
        int &z = (k == item->key[0] ? item->key[0] : item->key[1]);
        item->swap(z, min->key[0]);
        item = min; // Перемещаем указатель на лист, т.к. min - всегда лист
    }
    item->remove_from_node(k); // И удаляем требуемый ключ из листа
    return fix(item); // Вызываем функцию для восстановления свойств дерева.
```

```
    }
```

После того, как удалили ключ, могут получиться концептуально 4 разные ситуации: 3 из них нарушают свойства дерева, а одна — нет. Поэтому для вершины, из которой удалили ключ, нужно вызвать функцию исправления fix(), которая вернет свойства 2-3 дерева. Случаи, которые описываются в функции, рассматриваются ниже.

__Исправление дерева после удаления ключа__

```
node *fix(node *leaf) {
    if (leaf->size == 0 && leaf->parent == nullptr) { // Случай 0, когда удаляем единственный ключ в дереве
        delete leaf;
        return nullptr;
    }
    if (leaf->size != 0) { // Случай 1, когда вершина, в которой удалили ключ, имела два ключа
        if (leaf->parent) return fix(leaf->parent);
        else return leaf;
    }
    node *parent = leaf->parent;
    if (parent->first->size == 2 || parent->second->size == 2 || parent->size == 2) leaf = redistribute(leaf); // Случай 2, когда достаточно перераспределить ключи в дереве
    else if (parent->size == 2 && parent->third->size == 2) leaf = redistribute(leaf); // Аналогично
    else leaf = merge(leaf); // Случай 3, когда нужно произвести склеивание и пройтись вверх по дереву как минимум на еще одну вершину
    return fix(leaf);
}
```

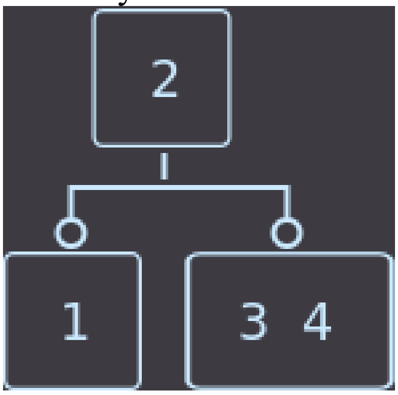
Теперь перейдем к возможным вариантам, которые могут появиться после удаления ключа. Для простоты будем рассматривать случаи, где глубина дерева равна 2. Общий случай — это дерево с глубиной равной трем. Тогда будет понятно, как справиться с удалением ключа в дереве с любой глубиной. Что мы и сделаем в итоговом примере для большинства ситуаций. А пока приступим к частным случаям.

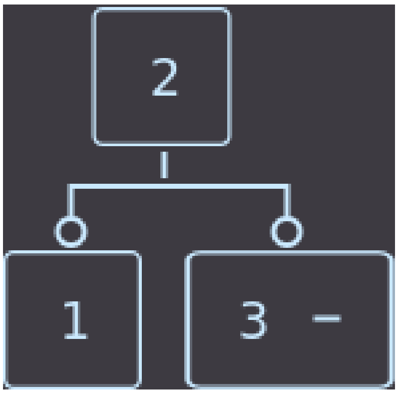
Случай 0:

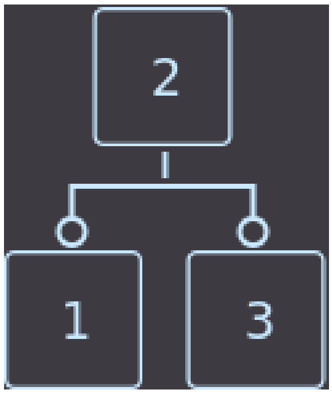
Самый простой случай, также как и следующий, на которые хватит и одного предложения, чтобы понять: если дерево состоит из одной вершины (корень), которая имеет 1 ключ, то просто удаляем эту вершину. The end.

Случай 1:

Если нужно удалить ключ из листа, где находятся два ключа, то мы просто удаляем ключ и на этом функция удаления закончена. Удалим key=4:

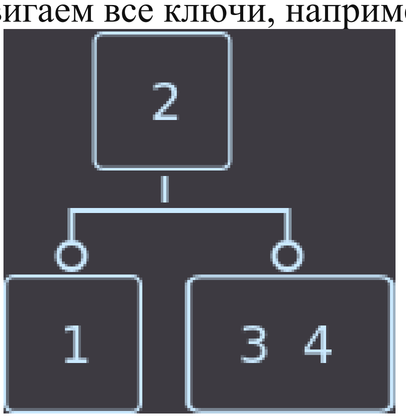


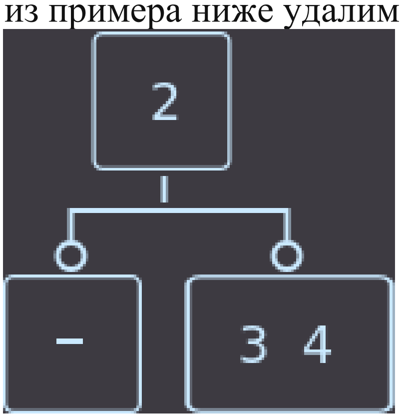
->


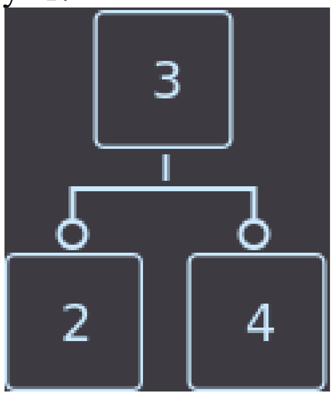
->


*Случай 2 (распределение или **redistribute**):*

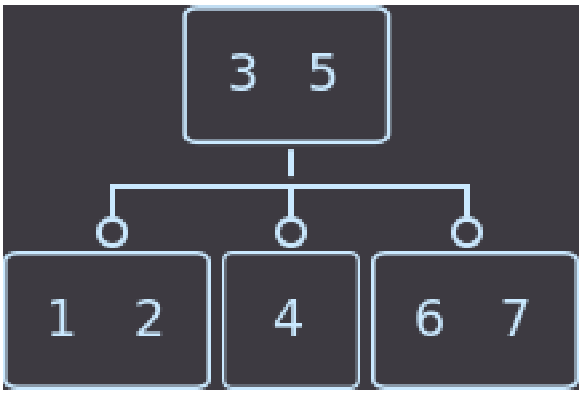
Мы удаляем ключ из вершины и вершина становится пустой. Если хотя бы у одного из братьев есть 2 ключа, то делаем простое правильное распределение и работа закончена. Под правильным распределением подразумевается, что при при цикличном сдвиге ключей между родителем и сыновьями также нужно будет перемещать и **внуков** родителя. Перераспределять ключи можно из любого брата, но удобнее всего из ближнего, который имеет 2 ключа, при этом мы циклично сдвигаем все ключи, например из примера ниже удалим key=1:

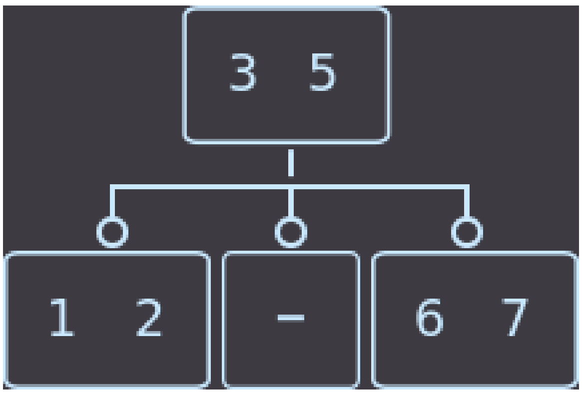


->


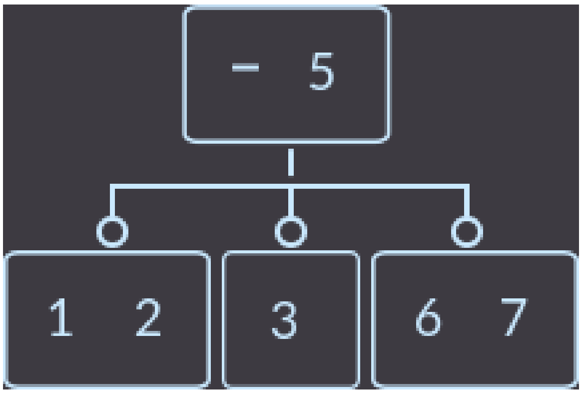
->


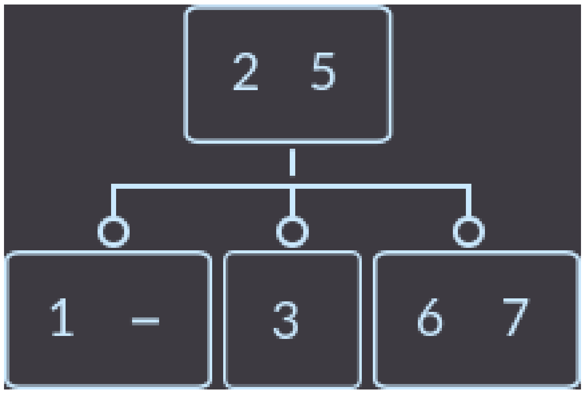
Или вот второй пример: у нас родитель имеет 2 ключа, соответственно 3 сына, и мы удалим key=4. Перераспределение в нашем примере можно сделать как из левого брата, так и из правого; Пример для левого:



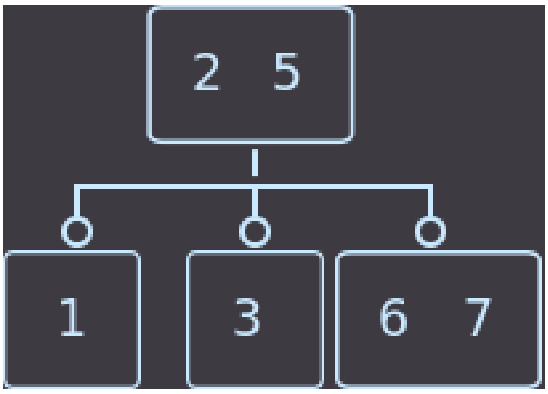
->


-

> 

-> 

->

-> 

Как видим, свойства дерева сохранены.

Перераспределение

```
    node *redistribute(node *leaf) {
        node *parent = leaf->parent;
        node *first = parent->first;
        node *second = parent->second;
        node *third = parent->third;
        if ((parent->size == 2) && (first->size < 2) && (second-
>size < 2) && (third->size < 2)) {
            if (first == leaf) {
                parent->first = parent->second;
                parent->second = parent->third;
                parent->third = nullptr;
                parent->first->insert_to_node(parent->key[0]);
                parent->first->third = parent->first->second;
                parent->first->second = parent->first->first;
                if (leaf->first != nullptr) parent->first->first =
leaf->first;
                else if (leaf->second != nullptr) parent->first-
>first = leaf->second;
                if (parent->first->first != nullptr) parent->first-
>first->parent = parent->first;
                parent->remove_from_node(parent->key[0]);
                delete first;
            } else if (second == leaf) {
                first->insert_to_node(parent->key[0]);
                parent->remove_from_node(parent->key[0]);
```

```
                if (leaf->first != nullptr) first->third = leaf-
>first;
                else if (leaf->second != nullptr) first->third =
leaf->second;
                if (first->third != nullptr) first->third->parent =
first;
                parent->second = parent->third;
                parent->third = nullptr;
                delete second;
            } else if (third == leaf) {
                second->insert_to_node(parent->key[1]);
                parent->third = nullptr;
                parent->remove_from_node(parent->key[1]);
                if (leaf->first != nullptr) second->third = leaf-
>first;
                else if (leaf->second != nullptr) second->third =
leaf->second;
                if (second->third != nullptr) second->third-
>parent = second;
                delete third;
            }
        } else if ((parent->size == 2) && ((first->size == 2) ||
(second->size == 2) || (third->size == 2))) {
            if (third == leaf) {
                if (leaf->first != nullptr) {
                    leaf->second = leaf->first;
                    leaf->first = nullptr;
                }
                leaf->insert_to_node(parent->key[1]);
                if (second->size == 2) {
                    parent->key[1] = second->key[1];
                    second->remove_from_node(second->key[1]);
                    leaf->first = second->third;
                    second->third = nullptr;
                    if (leaf->first != nullptr) leaf->first->parent
= leaf;
                } else if (first->size == 2) {
                    parent->key[1] = second->key[0];
                    leaf->first = second->second;
                    second->second = second->first;
                    if (leaf->first != nullptr) leaf->first->parent
= leaf;

                    second->key[0] = parent->key[0];
                    parent->key[0] = first->key[1];
                    first->remove_from_node(first->key[1]);
                    second->first = first->third;
                    if (second->first != nullptr) second->first-
>parent = second;
                    first->third = nullptr;
                }
            } else if (second == leaf) {
                if (third->size == 2) {
```

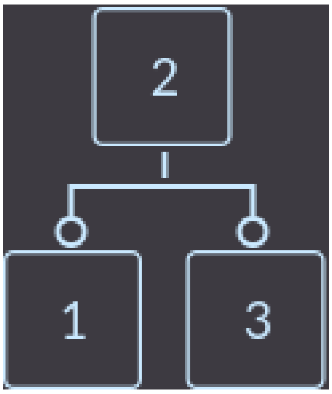
```
                    if (leaf->first == nullptr) {
                        leaf->first = leaf->second;
                        leaf->second = nullptr;
                    }
                    second->insert_to_node(parent->key[1]);
                    parent->key[1] = third->key[0];
                    third->remove_from_node(third->key[0]);
                    second->second = third->first;
                    if (second->second != nullptr) second->second-
>parent = second;
                    third->first = third->second;
                    third->second = third->third;
                    third->third = nullptr;
                } else if (first->size == 2) {
                    if (leaf->second == nullptr) {
                        leaf->second = leaf->first;
                        leaf->first = nullptr;
                    }
                    second->insert_to_node(parent->key[0]);
                    parent->key[0] = first->key[1];
                    first->remove_from_node(first->key[1]);
                    second->first = first->third;
                    if (second->first != nullptr) second->first-
>parent = second;
                    first->third = nullptr;
                }
            } else if (first == leaf) {
                if (leaf->first == nullptr) {
                    leaf->first = leaf->second;
                    leaf->second = nullptr;
                }
                first->insert_to_node(parent->key[0]);
                if (second->size == 2) {
                    parent->key[0] = second->key[0];
                    second->remove_from_node(second->key[0]);
                    first->second = second->first;
                    if (first->second != nullptr) first->second-
>parent = first;
                    second->first = second->second;
                    second->second = second->third;
                    second->third = nullptr;
                } else if (third->size == 2) {
                    parent->key[0] = second->key[0];
                    second->key[0] = parent->key[1];
                    parent->key[1] = third->key[0];
                    third->remove_from_node(third->key[0]);
                    first->second = second->first;
                    if (first->second != nullptr) first->second-
>parent = first;
                    second->first = second->second;
                    second->second = third->first;
                    if (second->second != nullptr) second->second-
>parent = second;
```

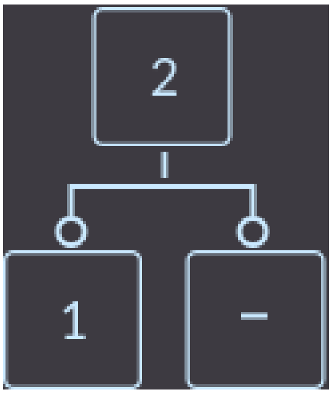
```
                        third->first = third->second;
                        third->second = third->third;
                        third->third = nullptr;
                    }
                }
            } else if (parent->size == 1) {
                leaf->insert_to_node(parent->key[0]);

                if (first == leaf && second->size == 2) {
                    parent->key[0] = second->key[0];
                    second->remove_from_node(second->key[0]);

                    if  (leaf->first  ==  nullptr)  leaf->first  =  leaf-
>second;

                    leaf->second = second->first;
                    second->first = second->second;
                    second->second = second->third;
                    second->third = nullptr;
                    if (leaf->second != nullptr) leaf->second->parent =
leaf;
                } else if (second == leaf && first->size == 2) {
                    parent->key[0] = first->key[1];
                    first->remove_from_node(first->key[1]);
                    if  (leaf->second  ==  nullptr)  leaf->second  =  leaf-
>first;
                    leaf->first = first->third;
                    first->third = nullptr;
                    if  (leaf->first  !=  nullptr)  leaf->first->parent  =
leaf;
                }
            }
            return parent;
        }
```

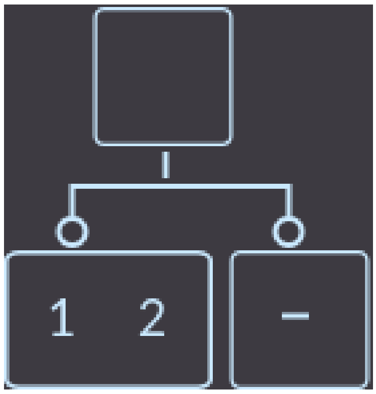
*Случай 3 (склеивание или **merge**):*

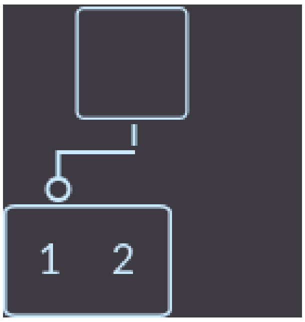
Пожалуй, самый сложный случай, так как после склеивания всегда обязательно идти по дереву вверх и опять применять операции либо *merge*, либо *redistribute*. После redistribute алгоритм восстановления свойств 2-3-дерева после удаления ключа можно прекратить, так как вершины в этой операции не удаляются.

Для начала посмотрим, как удалить ключ key=3 из вершины, родитель которой имеет только двух сыновей (~ 1 ключ):



->


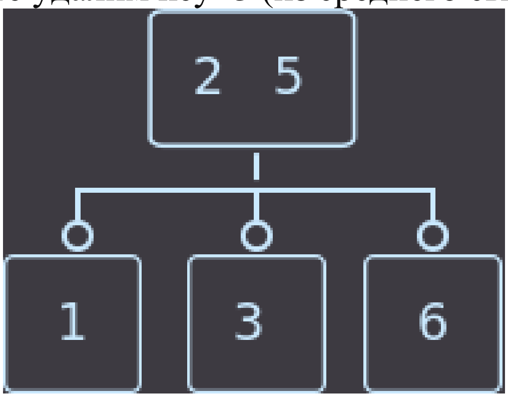
->


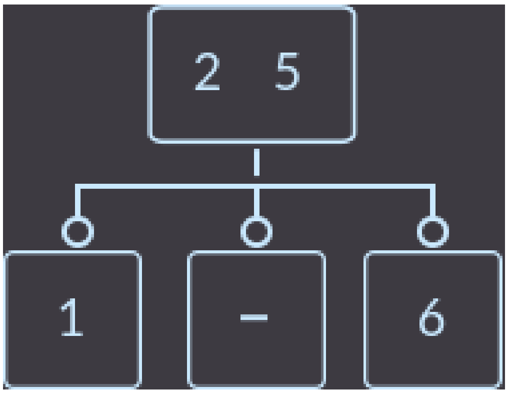
->


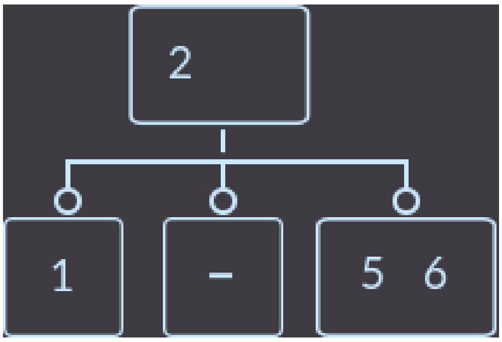
->

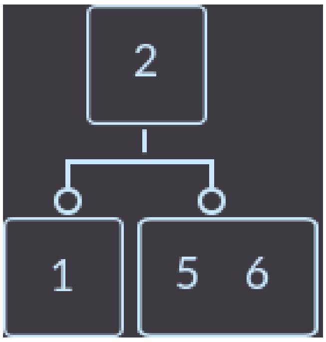

Что же мы сделали? Мы удалили ключ key=3. Затем нам нужно перенести ключ из родителя в того сына, где ключ есть: в данном случае в левого сына. Затем нужно удалить вершину, из которой удалили ключ. И последний шаг — это проверить, если родитель был корнем дерева, то удалить этот корень и назначить новый корнем ту вершину, куда мы перенесли ключ, иначе придется вызывать функцию исправления fix() уже для родителя. Выглядит легко.

Теперь рассмотрим два варианта, когда родитель имеет 2 ключа. В первом варианте удалим key=3 (из среднего сына):



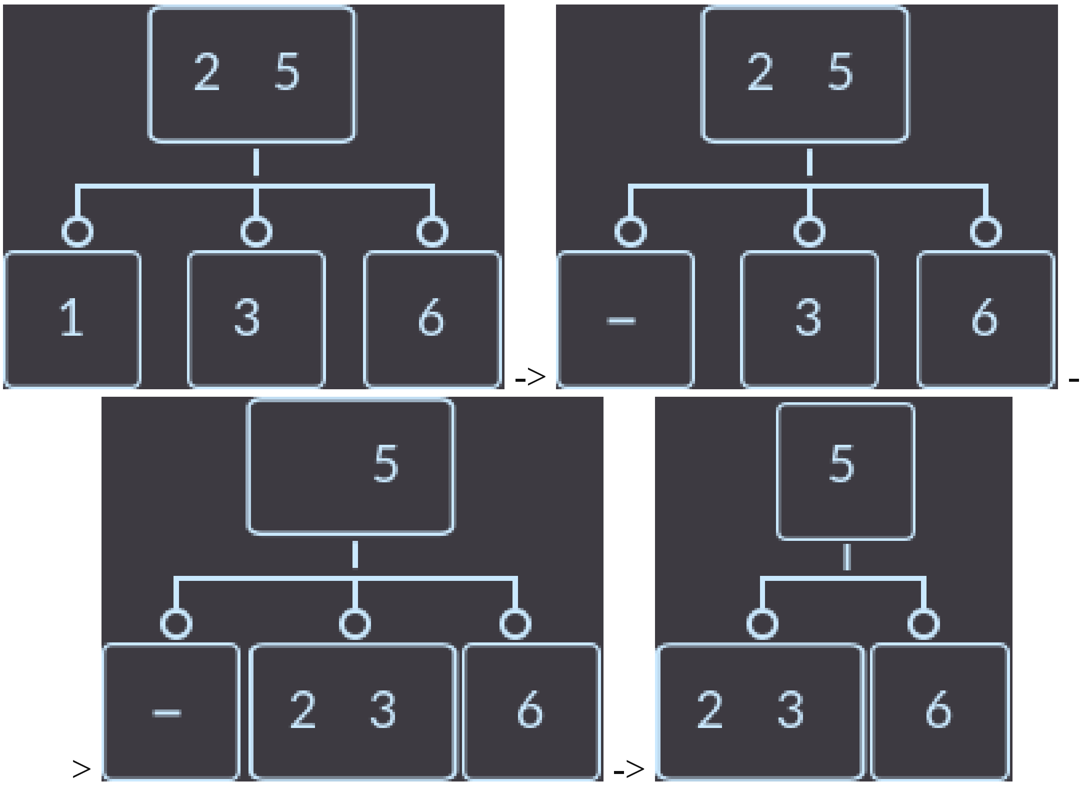
->


->


->


Что мы сделали на этот раз? Мы снова перенесли ключ родителя (уже один из двух) к сыну и удалили сына, который не имеет ключей. Так как родитель имел

2 ключа, то проверять, являлся ли родитель корнем, не нужно. В описанном выше случае алгоритм исправления такой: нужно перенести меньший ключ родителя в поддерево с меньшими ключами либо больший ключ в поддерево с большими ключами, и удалить вершину без ключей. Еще пример, удалим key=1:



Склеивание

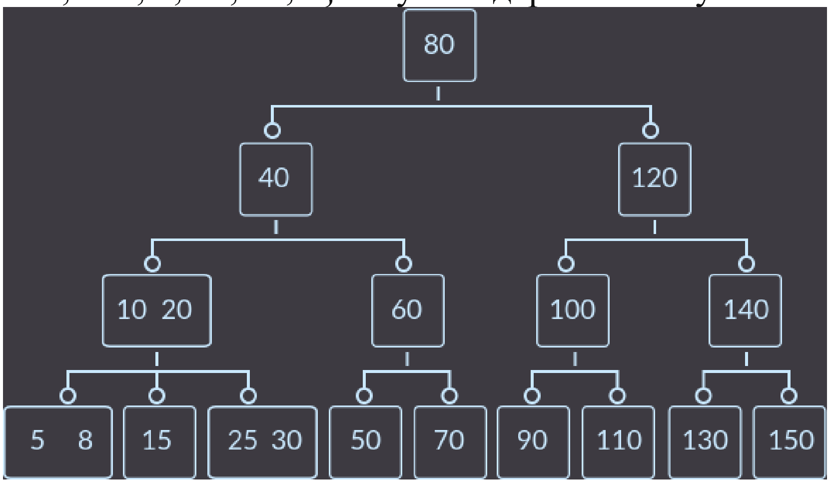
```
node *merge(node *leaf) {
    node *parent = leaf->parent;
    if (parent->first == leaf) {
        parent->second->insert_to_node(parent->key[0]);
        parent->second->third = parent->second->second;
        parent->second->second = parent->second->first;
        if (leaf->first != nullptr) parent->second->first = leaf->first;
        else if (leaf->second != nullptr) parent->second->first = leaf->second;
        if (parent->second->first != nullptr) parent->second->first->parent = parent->second;
        parent->remove_from_node(parent->key[0]);
        delete parent->first;
        parent->first = nullptr;
    } else if (parent->second == leaf) {
        parent->first->insert_to_node(parent->key[0]);
        if (leaf->first != nullptr) parent->first->third = leaf->first;
        else if (leaf->second != nullptr) parent->first->third = leaf->second;
```

```
            if (parent->first->third != nullptr) parent->first->third->parent = parent->first;
            parent->remove_from_node(parent->key[0]);
            delete parent->second;
            parent->second = nullptr;
        }
        if (parent->parent == nullptr) {
            node *tmp = nullptr;
            if (parent->first != nullptr) tmp = parent->first;
            else tmp = parent->second;
            tmp->parent = nullptr;
            delete parent;
            return tmp;
        }
        return parent;
    }
```

! Важно! При склеивании или перераспределении нелистовой вершины нужно будет также склеивать и/или перераспределять сыновей братьев.

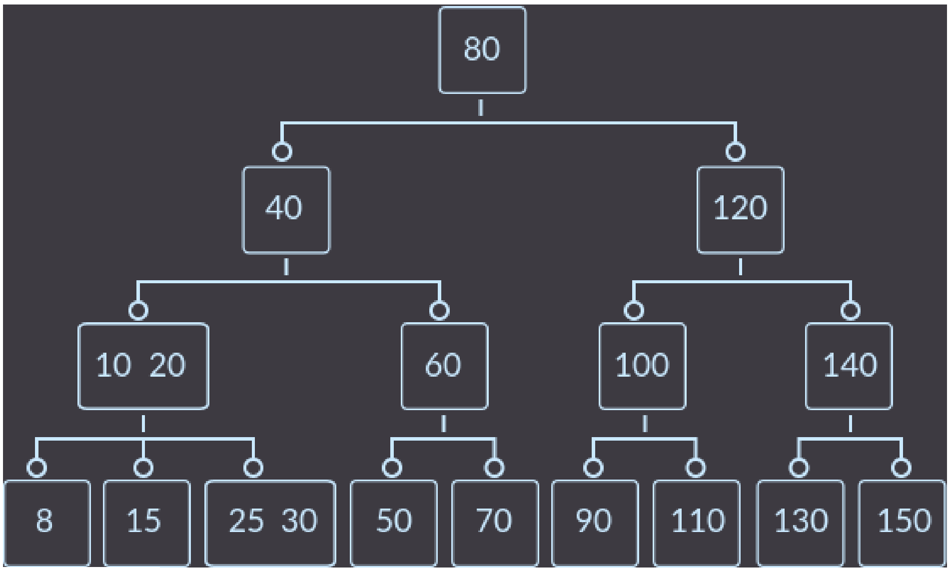
Итоговый пример удаления ключей:

Подобрал пример такой, чтобы можно было увидеть все основные случаи (кроме *случая 0*). Для начала вставим ключи keys={10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 5, 15, 25, 8} в пустое дерево и получим:

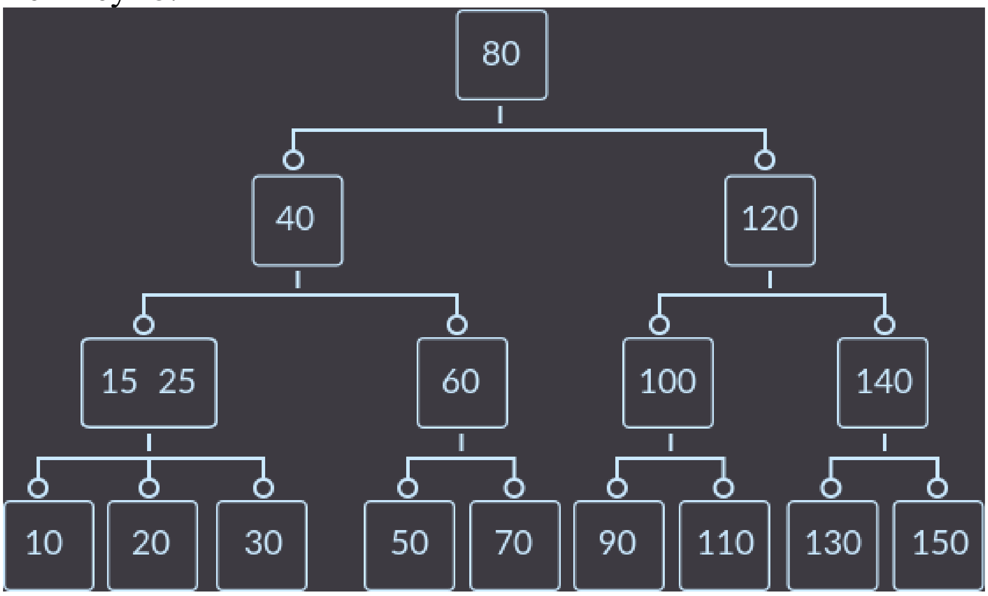


Теперь удалим по порядку ключи keys={5, 8, 10, 30, 15}.

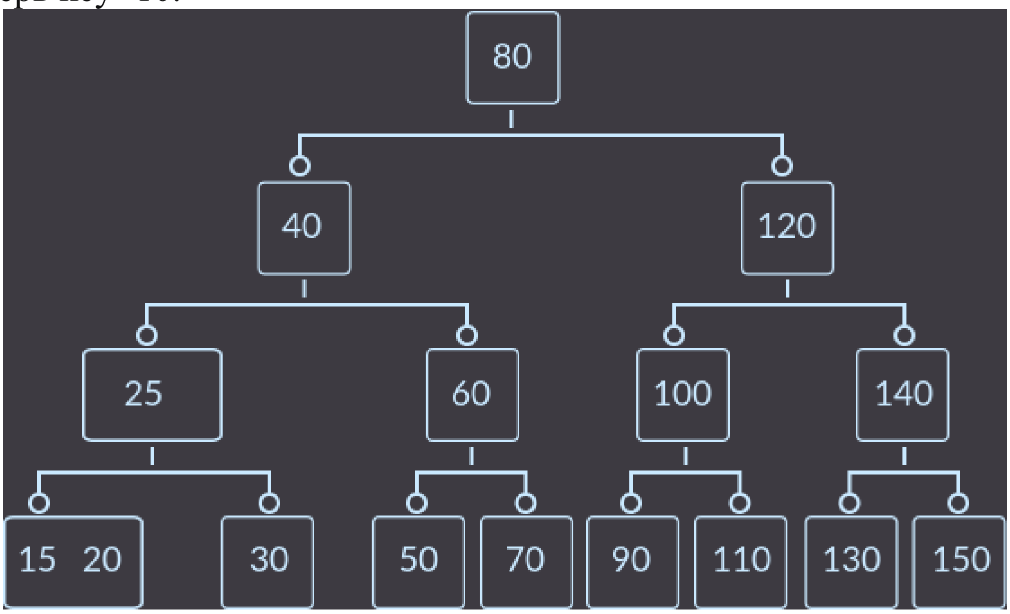
После удаления ключа key=5 получаем:



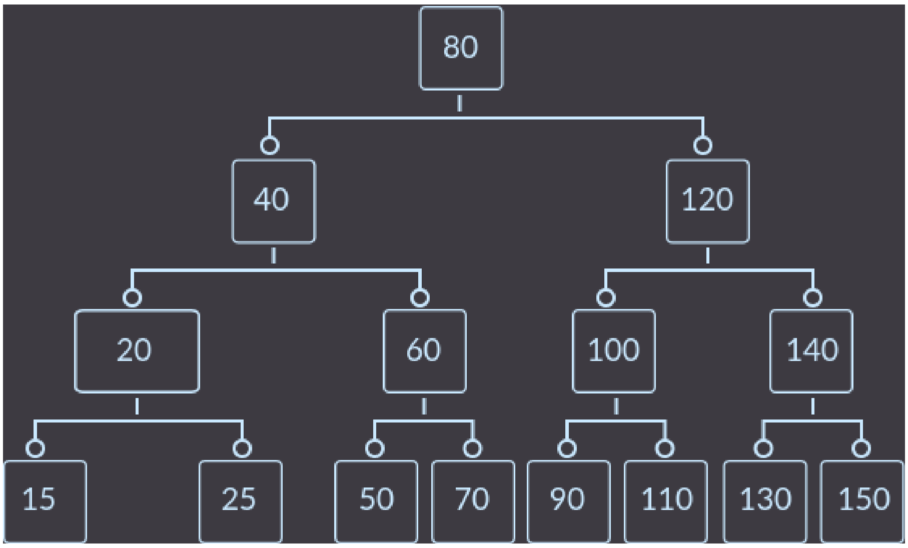
Удаляем key=8:



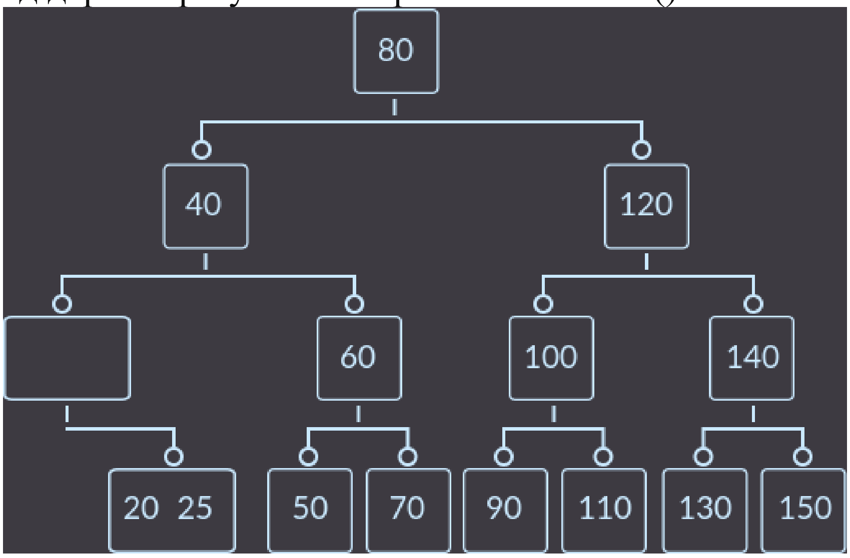
Теперь key=10:



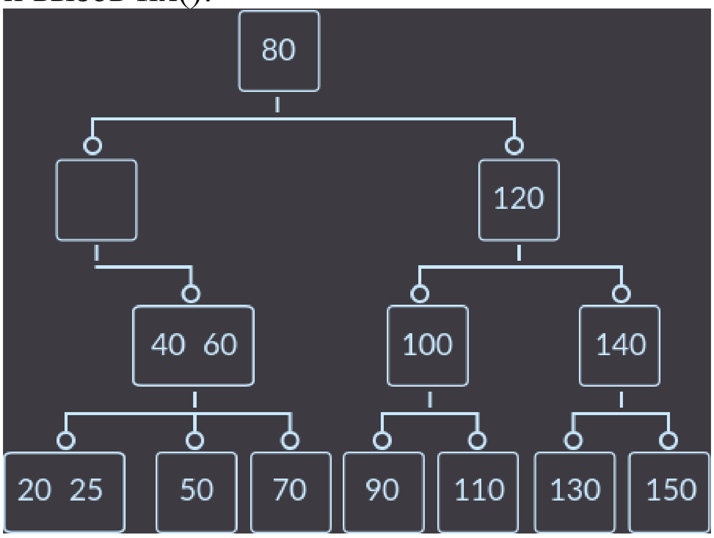
Key=30:



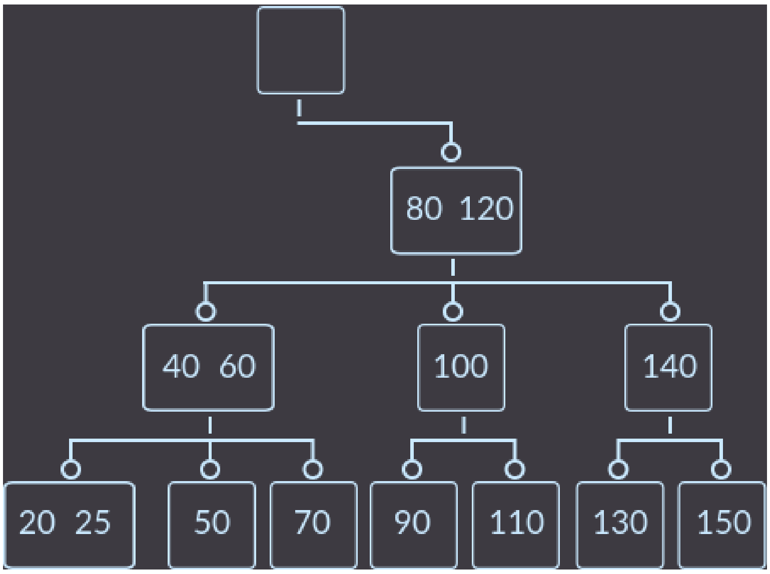
И, наконец, key=15. Так как тут происходит при исправлении дерева операция merge, то посмотрим на все шаги.
Шаг 1. Вид дерева сразу после первого вызова fix():



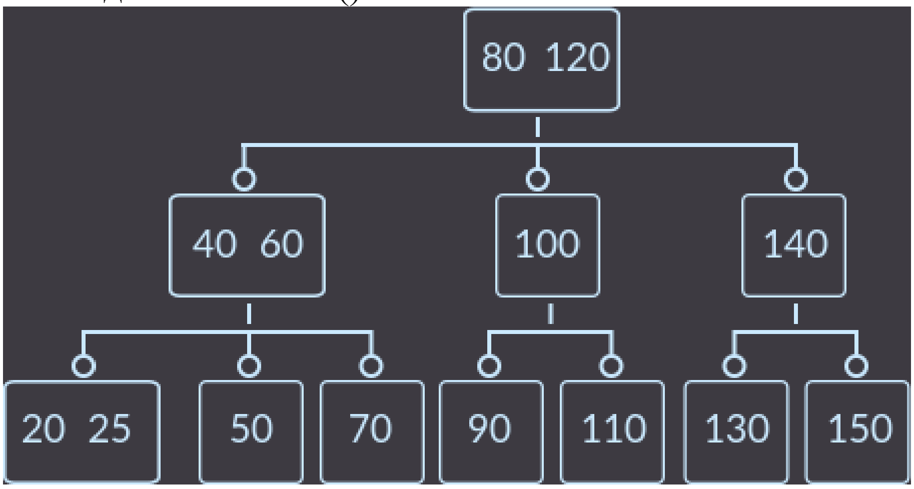
Шаг 2. Второй вызов fix():



Шаг 3. Третий вызов fix():



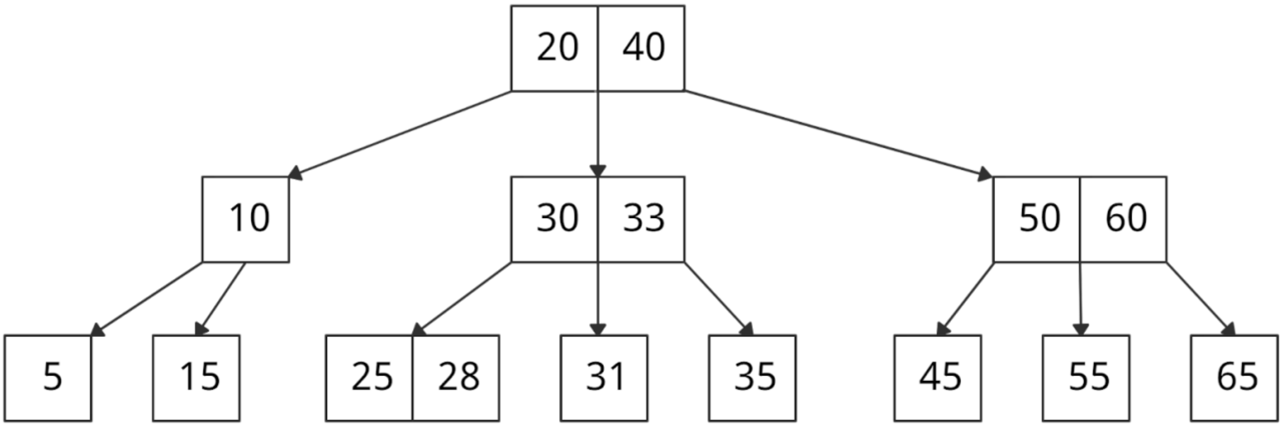
Шаг 4. Последний вызов fix():



Вот так мы удалили ключ key=15 и дерево осталось с теми свойствами, с которыми и должно быть.

1.3 Б-ДЕРЕВЬЯ

B-дерево (читается как Би-дерево) — это особый тип сбалансированного дерева поиска, в котором каждый узел может содержать более одного ключа и иметь более двух дочерних элементов. Из-за этого свойства B-дерево называют *сильноветвящимся*.



Зачем нужно

Вторичные запоминающие устройства (жесткие диски, SSD) медленно работают с большим объемом данных. Людям захотелось сократить время доступа к физическим носителям информации, поэтому возникла потребность в таких структурах данных, которые способны это сделать.

Двоичное дерево поиска, АВЛ-дерево, красно-черное дерево и т. д. могут хранить только один ключ в одном узле. Если нужно хранить больше, высота деревьев резко начинает расти, из-за этого время доступа сильно увеличивается.

С B-деревом все не так. Оно позволяет хранить много ключей в одном узле и при этом может ссылаться на несколько дочерних узлов. Это значительно уменьшает высоту дерева и, соответственно, обеспечивает более быстрый доступ к диску.

Свойства

1. Ключи в каждом узле x упорядочены по неубыванию.
2. В каждом узле есть логическое значение x.leaf. Оно истинно, если x — лист.
3. Каждый узел, кроме корня, содержит не менее t-1 ключей, а каждый внутренний узел имеет как минимум t дочерних узлов, где t — *минимальная степень* B-дерева.
4. Все листья находятся на одном уровне, т. е. обладают одинаковой глубиной, равной высоте дерева.
5. Корень имеет не менее 2 дочерних элементов и содержит не менее 1 ключа.

Операции с B-деревом

Поиск элемента

Средняя временная сложность: Θ(log n)
Худшая временная сложность: Θ(log n)

Поиск ключа в B-дереве работает так же, как и в двоичном дереве поиска.

1. Сравниваем k с первым ключом узла, начиная с корня. Если k = первый ключ узла, возвращаем узел и индекс.
2. Если k.leaf = true, возвращаем NULL. Элемент не найден.

3. Если k < первый ключ корня, рекурсивно ищем левый дочерний элемент этого ключа.

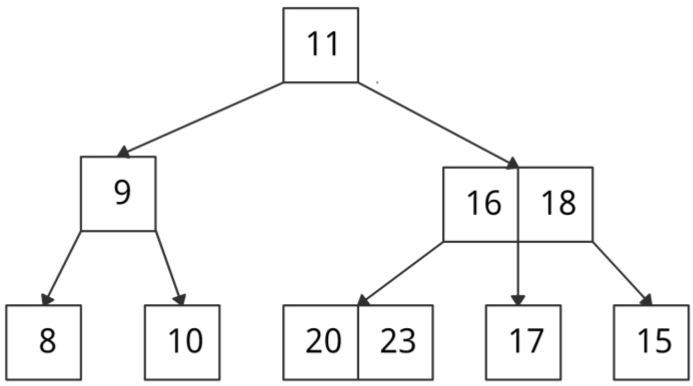
4. Если в текущем узле более одного ключа и k > первый ключ, сравниваем k со следующим ключом в узле. Если k < следующий ключ, ищем левый дочерний элемент этого ключа (k находится между первым и вторым ключами). Иначе иначе ищем правый дочерний элемент ключа.

5. Повторяем шаги с 1 по 4, пока не дойдем до листа.

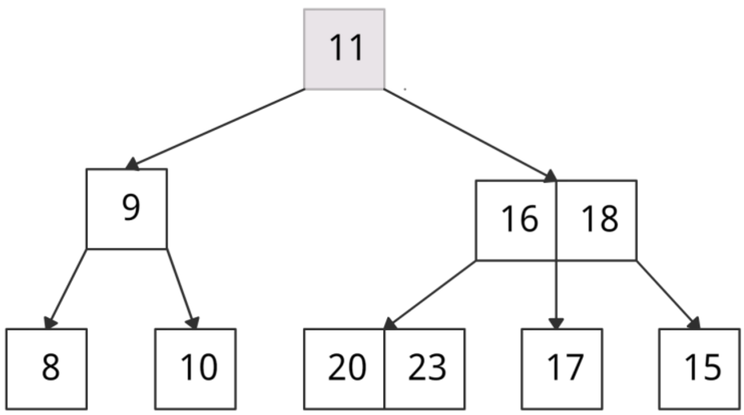
Алгоритм:

```
BtreeSearch(x, k)
 i = 1
 while i ≤ n[x] and k ≥ keyi[x]        // n[x] — количество ключей в узле x
   do i = i + 1
if i  n[x] and k = keyi[x]
   then return (x, i)
if leaf [x]
   then return NIL
else
   return BtreeSearch(ci[x], k)
```

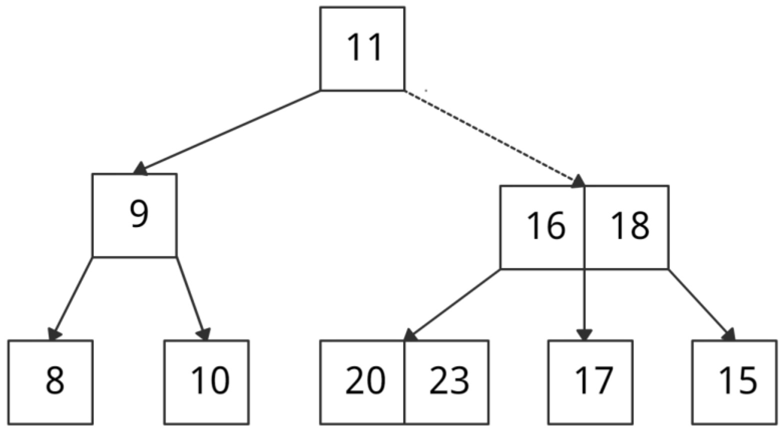
<u>*Применим на примере*</u>



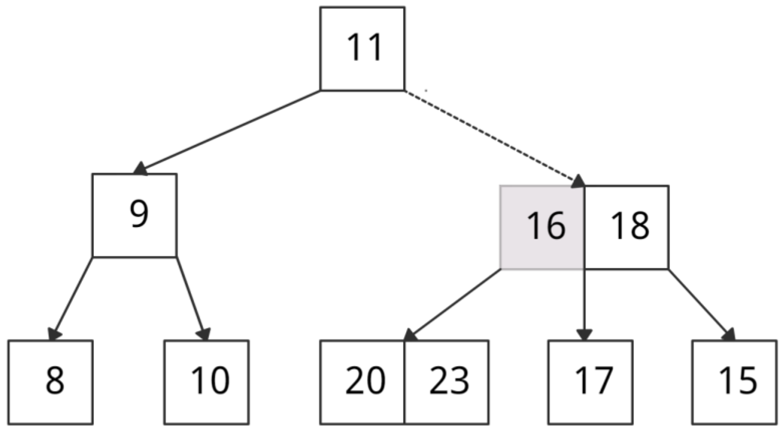
• Хотим найти ключ k = 17 в этом дереве.



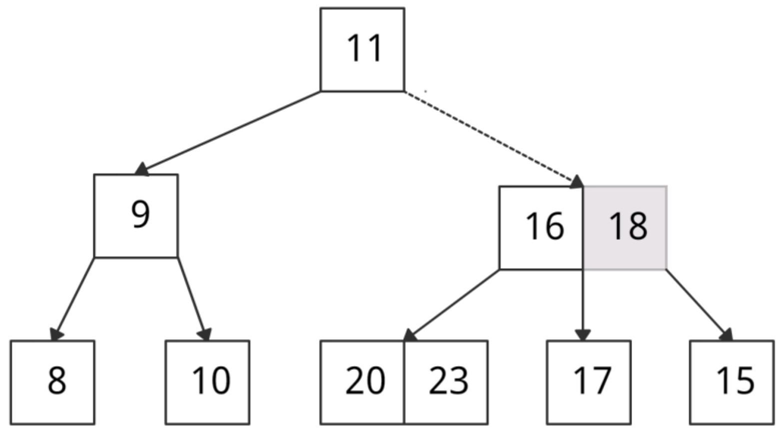
• k нет в корне → сравниваем k с ключом корня.



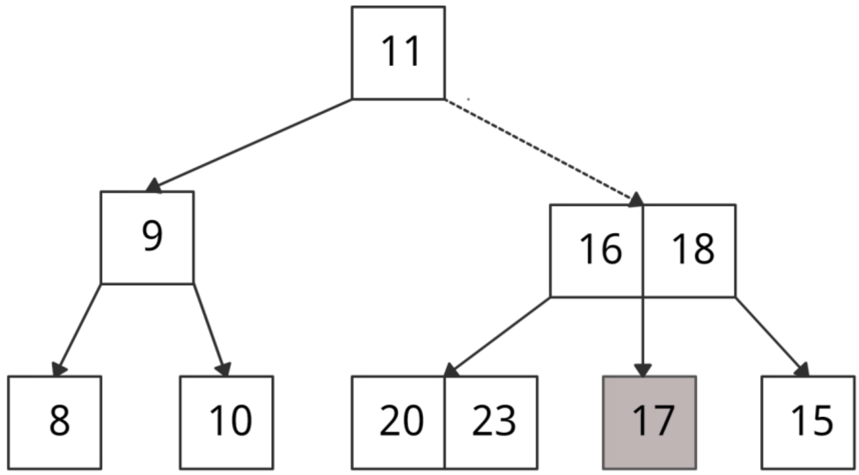
• k > 11 → идем через правого «ребенка».



• Сравниваем k с первым ключом узла: k > 16 → сравниваем k со вторым ключом узла.



• k > 18 → k лежит между 16 и 18. Ищем либо в правом «ребенке» 16, либо в левом «ребенке» 18.



• Нашли 17.

Реализация на языках программирования

Python

```
# Поиска ключа в B-дереве на Python
    # Создаем узел
    class BTreeNode:
        def __init__(self, leaf=False):
            self.leaf = leaf
            self.keys = []
            self.child = []
    class BTree:
        def __init__(self, t):
            self.root = BTreeNode(True)
            self.t = t
        # Выводим дерево на экран
        def print_tree(self, x, l=0):
            print("Уровень", l, " ", len(x.keys), end=":")
            for i in x.keys:
                print(i, end=" ")
            print()
            l += 1
            if len(x.child) > 0:
                for i in x.child:
                    self.print_tree(i, l)
        # Поиск ключа
        def search_key(self, k, x=None):
            if x is not None:
                i = 0
                while i < len(x.keys) and k > x.keys[i][0]:
                    i += 1
                if i < len(x.keys) and k == x.keys[i][0]:
                    return (x, i)
                elif x.leaf:
```

```
                return None
else:
return self.search_key(k, x.child[i])
else:
return self.search_key(k, self.root)
# Вставка ключа
def insert_key(self, k):
root = self.root
if len(root.keys) == (2 * self.t) - 1:
temp = BTreeNode()
self.root = temp
            temp.child.insert_key(0, root)
            self.split(temp, 0)
            self.insert_non_full(temp, k)
        else:
            self.insert_non_full(root, k)
    # Вставка ключа k в узел x, который должен быть
    # незаполненным при вызове
    def insert_non_full(self, x, k):
        i = len(x.keys) - 1
        if x.leaf:
            x.keys.append((None, None))
            while i >= 0 and k[0] < x.keys[i][0]:
                x.keys[i + 1] = x.keys[i]
                i -= 1
            x.keys[i + 1] = k
        else:
            while i >= 0 and k[0] < x.keys[i][0]:
                i -= 1
            i += 1
            if len(x.child[i].keys) == (2 * self.t) - 1:
                self.split(x, i)
                if k[0] > x.keys[i][0]:
                    i += 1
            self.insert_non_full(x.child[i], k)
    # Разбиение узла
    def split(self, x, i):
        t = self.t
        y = x.child[i]
        z = BTreeNode(y.leaf)
        x.child.insert_key(i + 1, z)
        x.keys.insert_key(i, y.keys[t - 1])
        z.keys = y.keys[t: (2 * t) - 1]
        y.keys = y.keys[0: t - 1]
        if not y.leaf:
            z.child = y.child[t: 2 * t]
            y.child = y.child[0: t - 1]
```

```
def main():
    B = BTree(3)
    for i in range(10):
        B.insert_key((i, 2 * i))
    B.print_tree(B.root)
    if B.search_key(8) is not None:
        print("\nНайдено")
    else:
        print("\nНе найдено")
if __name__ == '__main__':
    main()
```

Java

```
/ Поиск ключа в B-дереве на Java
public class BTree {
  private int T;
  // Создаем узел
  public class Node {
    int n;
    int key[] = new int[2 * T - 1];
    Node child[] = new Node[2 * T];
    boolean leaf = true;
    public int Find(int k) {
      for (int i = 0; i < this.n; i++) {
        if (this.key[i] == k) {
          return i;} }
      return -1;}; }
 public BTree(int t) {
    T = t;
    root = new Node();
    root.n = 0;
    root.leaf = true; }
  private Node root;
  // Поиск ключа
  private Node Search(Node x, int key) {
    int i = 0;
    if (x == null)
      return x;
    for (i = 0; i < x.n; i++) {
      if (key < x.key[i]) {
        break;  }
      if (key == x.key[i]) {
        return x;
      }}
    if (x.leaf) {
      return null;
    } else {
```

```
      return Search(x.child[i], key);
    }}
  // Разбиение узла
  private void Split(Node x, int pos, Node y) {
    Node z = new Node();
    z.leaf = y.leaf;
    z.n = T - 1;
    for (int j = 0; j < T - 1; j++) {
      z.key[j] = y.key[j + T];   }
    if (!y.leaf) {
      for (int j = 0; j < T; j++) {
        z.child[j] = y.child[j + T];} }
    y.n = T - 1;
    for (int j = x.n; j >= pos + 1; j--) {
      x.child[j + 1] = x.child[j]; }
    x.child[pos + 1] = z;
    for (int j = x.n - 1; j >= pos; j--) {
      x.key[j + 1] = x.key[j]; }
    x.key[pos] = y.key[T - 1];
    x.n = x.n + 1; }
  // Вставка значения
  public void Insert(final int key) {
    Node r = root;
    if (r.n == 2 * T - 1) {
      Node s = new Node();
      root = s;
      s.leaf = false;
      s.n = 0;
      s.child[0] = r;
      Split(s, 0, r);
      insertValue(s, key);
    } else {
      insertValue(r, key); }}
  // Вставка узла
  final private void insertValue(Node x, int k) {
    if (x.leaf) {
      int i = 0;
      for (i = x.n - 1; i >= 0 && k < x.key[i]; i--) {
        x.key[i + 1] = x.key[i]; }
      x.key[i + 1] = k;
      x.n = x.n + 1;
    } else {
      int i = 0;
      for (i = x.n - 1; i >= 0 && k < x.key[i]; i--) {
   }  ;
      i++;
      Node tmp = x.child[i];
```

```
      if (tmp.n == 2 * T - 1) {
        Split(x, i, tmp);
        if (k > x.key[i]) {
          i++;  } }
      insertValue(x.child[i], k); } }
  public void Show() {
    Show(root); }
  // Вывод на экран
  private void Show(Node x) {
    assert (x == null);
    for (int i = 0; i < x.n; i++) {
      System.out.print(x.key[i] + " "); }
    if (!x.leaf) {
      for (int i = 0; i < x.n + 1; i++) {
        Show(x.child[i]); }} }
  // Проверка, содержится ли ключ
  public boolean Contain(int k) {
    if (this.Search(root, k) != null) {
      return true;
    } else {
      return false; } }
  public static void main(String[] args) {
    BTree b = new BTree(3);
    b.Insert(8);
    b.Insert(9);
    b.Insert(10);
    b.Insert(11);
    b.Insert(15);
    b.Insert(20);
    b.Insert(17);
    b.Show();
    if (b.Contain(12)) {
      System.out.println("\nнайдено");
    } else {
      System.out.println("\nне найдено"); }  ; }}
```

C

```
// Поиск ключа в В-дереве на С
#include <stdio.h>
#include <stdlib.h>
# MAX 3
# MIN 2
struct BTreeNode {
  int val[MAX + 1], count;
  struct BTreeNode *link[MAX + 1];};
struct BTreeNode *root;
// Создаем узел
```

```
struct BTreeNode *createNode(int val, struct BTreeNode
*child) {
  struct BTreeNode *newNode;
  newNode = (struct BTreeNode *)malloc(sizeof(struct
BTreeNode));
  newNode->val[1] = val;
  newNode->count = 1;
  newNode->link[0] = root;
  newNode->link[1] = child;
  return newNode;}
// Вставка узла
void insertNode(int val, int pos, struct BTreeNode
*node,
        struct BTreeNode *child) {
  int j = node->count;
  while (j > pos) {
    node->val[j + 1] = node->val[j];
    node->link[j + 1] = node->link[j];
    j--;}
  node->val[j + 1] = val;
  node->link[j + 1] = child;
  node->count++;}
// Разбиение узла
void splitNode(int val, int *pval, int pos, struct
BTreeNode *node,
        struct BTreeNode *child, struct BTreeNode
**newNode) {
  int median, j;
  if (pos > MIN)
    median = MIN + 1;
  else
    median = MIN;
  *newNode = (struct BTreeNode *)malloc(sizeof(struct
BTreeNode));
  j = median + 1;
  while (j <= MAX) {
    (*newNode)->val[j - median] = node->val[j];
    (*newNode)->link[j - median] = node->link[j];
    j++; }
  node->count = median;
  (*newNode)->count = MAX - median;
  if (pos <= MIN) {
    insertNode(val, pos, node, child);
  } else {
    insertNode(val, pos - median, *newNode, child); }
  *pval = node->val[node->count];
  (*newNode)->link[0] = node->link[node->count];
```

```
  node->count--;}
// Устанавливаем значение
int setValue(int val, int *pval,
             struct BTreeNode *node, struct BTreeNode
**child) {
  int pos;
  if (!node) {
    *pval = val;
    *child = NULL;
    return 1; }
  if (val < node->val[1]) {
    pos = 0;
  } else {
    for (pos = node->count;
      (val < node->val[pos] && pos > 1); pos--)  ;
    if (val == node->val[pos]) {
      printf("Повторения недопустимы\n");
      return 0;     } }
  if (setValue(val, pval, node->link[pos], child)) {
    if (node->count < MAX) {
      insertNode(*pval, pos, node, *child);
    } else {
      splitNode(*pval, pval, pos, node, *child, child);
      return 1; } }
  return 0;}
// Вставка значения
void insert(int val) {
  int flag, i;
  struct BTreeNode *child;
  flag = setValue(val, &i, root, &child);
  if (flag)
    root = createNode(i, child);}
// Поиск узла
void search(int val, int *pos, struct BTreeNode *myNode)
{ if (!myNode) {
    return; }
  if (val < myNode->val[1]) {
    *pos = 0; } else {
    for (*pos = myNode->count;
      (val < myNode->val[*pos] && *pos > 1); (*pos)--)
  ;
    if (val == myNode->val[*pos]) {
      printf("%d is found", val);
      return;    }  }
  search(val, pos, myNode->link[*pos]);
  return;}
// Обход узлов
```

```
void traversal(struct BTreeNode *myNode) {
  int i;
  if (myNode) {
    for (i = 0; i < myNode->count; i++) {
      traversal(myNode->link[i]);
      printf("%d ", myNode->val[i + 1]); }
    traversal(myNode->link[i]); }}
int main() {
  int val, ch;
  insert(8);
  insert(9);
  insert(10);
  insert(11);
  insert(15);
  insert(16);
  insert(17);
  insert(18);
  insert(20);
  insert(23);
  traversal(root);
  printf("\n");
  search(11, &ch, root);}
```

C++

```
// Поиск ключа в B-дереве на C++
#include <iostream>
using namespace std;
class TreeNode {
  int *keys;
  int t;
  TreeNode **C;
  int n;
  bool leaf;
   public:
  TreeNode(int temp, bool bool_leaf);
  void insertNonFull(int k);
  void splitChild(int i, TreeNode *y);
  void traverse();
  TreeNode *search(int k);
  friend class BTree;};
class BTree {
  TreeNode *root;
  int t;
   public:
  BTree(int temp) {
    root = NULL;
```

```
    t = temp;}
  void traverse() {
    if (root != NULL)
      root->traverse();}
  TreeNode *search(int k) {
    return (root == NULL) ? NULL : root->search(k); }
  void insert(int k);};
TreeNode::TreeNode(int t1, bool leaf1) {
  t = t1;
  leaf = leaf1;
  keys = new int[2 * t - 1];
  C = new TreeNode *[2 * t];
  n = 0;}
void TreeNode::traverse() {
  int i;
  for (i = 0; i < n; i++) {
    if (leaf == false)
      C[i]->traverse();
    cout << " " << keys[i]; }
  if (leaf == false)
    C[i]->traverse();}
TreeNode *TreeNode::search(int k) {
  int i = 0;
  while (i < n && k > keys[i])
    i++;
  if (keys[i] == k)
    return this;
  if (leaf == true)
    return NULL;
  return C[i]->search(k);}
void BTree::insert(int k) {
  if (root == NULL) {
    root = new TreeNode(t, true);
    root->keys[0] = k;
    root->n = 1;
  } else {
    if (root->n == 2 * t - 1) {
      TreeNode *s = new TreeNode(t, false);
      s->C[0] = root;
      s->splitChild(0, root);
      int i = 0;
      if (s->keys[0] < k)
        i++;
      s->C[i]->insertNonFull(k);
      root = s;
    } else
      root->insertNonFull(k); }}
```

```
void TreeNode::insertNonFull(int k) {
  int i = n - 1;
  if (leaf == true) {
    while (i >= 0 && keys[i] > k) {
      keys[i + 1] = keys[i];
      i--; }
    keys[i + 1] = k;
    n = n + 1;
  } else {
    while (i >= 0 && keys[i] > k)
      i--;
    if (C[i + 1]->n == 2 * t - 1) {
      splitChild(i + 1, C[i + 1]);
      if (keys[i + 1] < k)
        i++; }
    C[i + 1]->insertNonFull(k); }}
void TreeNode::splitChild(int i, TreeNode *y) {
  TreeNode *z = new TreeNode(y->t, y->leaf);
  z->n = t - 1;
  for (int j = 0; j < t - 1; j++)
    z->keys[j] = y->keys[j + t];
  if (y->leaf == false) {
    for (int j = 0; j < t; j++)
      z->C[j] = y->C[j + t]; }
  y->n = t - 1;
  for (int j = n; j >= i + 1; j--)
    C[j + 1] = C[j];
  C[i + 1] = z;
  for (int j = n - 1; j >= i; j--)
    keys[j + 1] = keys[j];
  keys[i] = y->keys[t - 1];
  n = n + 1;}
int main() {
  BTree t(3);
  t.insert(8);
  t.insert(9);
  t.insert(10);
  t.insert(11);
  t.insert(15);
  t.insert(16);
  t.insert(17);
  t.insert(18);
  t.insert(20);
  t.insert(23);
  cout << "B-дерево: ";
  t.traverse();
  int k = 10;
```

```
  (t.search(k) != NULL) ? cout << endl
                   << k << " найдено"
               : cout << endl
                   << k << " не найдено";
  k = 2;
  (t.search(k) != NULL) ? cout << endl
                   << k << " найдено"
               : cout << endl
                   << k << " не найдено\n";}
```

Где используется

- В базах данных и файловых системах.
- Для хранения блоков данных (вторичные носители).
- Для многоуровневой индексации.

1.4 КРАСНО-ЧЕРНЫЕ ДЕРЕВЬЯ

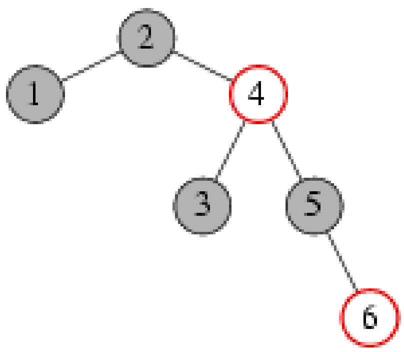
В красно-черных деревьях для балансировки используются цвета вершин. Каждая вершина красно-черного дерева окрашена либо в черный, либо в красный цвет, причем выполняются следующие условия:

- корень дерева окрашен в черный цвет;
- у красной вершины дети черные (если они есть);
- всякий путь от корня дерева к внешней вершине (листу) содержит одно и то же число черных вершин.

Последнее свойство означает сбалансированность красно-черного дерева по черным вершинам. Количество черных вершин на пути от корня дерева к внешней вершине иногда называют *черной высотой*красно-черного дерева, последнее свойство означает, что определенная таким образом черная высота не зависит от конкрентого пути.

Для удобства описания алгоритмов мы будем считать также, что все внешние вершины окрашены в черный цвет, а заголовочная вершина дерева (та, которая является родительской для корня дерева) — в красный.

Пример красно-черного дерева приведен на рисунке. Черные вершины изображаются кружками серого цвета, красные вершины — белого.



Пусть h_b — черная высота красно-черного дерева, т.е. количество черных вершин в произвольном пути от корня дерева к внешней вершине (не включая саму внешнюю вершину). По определению красно-черного дерева, эта величина определена корректно (не зависит от пути). Для дерева, изображенного на рисунке, h_b=2. Поскольку любой путь от корня к внешней вершине начинается с черной вершины (в красно-черном дереве корень всегда черный) и не может содержать двух красных вершин подряд (у красной вершины дети обязательно черные), то длина любого пути не превосходит $2h_b$. С другой стороны, длина минимального пути к внешней вершине не меньше чем h_b. Таким образом, в случае красно-черного дерева для длин m_0 и m_1 минимального и максимального путей к внешним вершинам выполняются неравенства

$m_1 \le 2\ h_b,$

$m_0 \ge h_b,$

откуда вытекает неравенство

$m_1 \le 2\ m_0.$

Таким образом, красно-черное дерево является почти сбалансированным с баланс-фактором 2. Из предложения 5 вытекает логарифмическая оценка на высоту h красно-черного дерева, содержащего n вершин:

$h \le 2\log_2(n+1).$

Поиск элемента в красно-черном дереве осуществляется за логарифмическое время:

$t = O(\log_2 n),$

где константа в представлении О-большого равна двум.

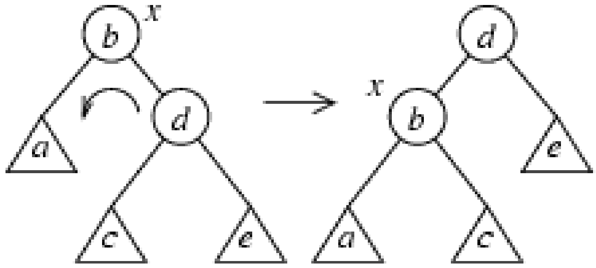
Добавление элементов в красно-черное дерево

При добавлении нового элемента к красно-черному дереву мы сначала применяем обычный алгоритм добавления вершины к дереву поиска, который был рассмотрен выше (алгоритм insert). Новая вершина (она в рассмотренном алгоритме всегда является терминальной) окрашивается в красный цвет. Если родительская вершина для добавленной имеет черный цвет, то все свойства красно-черного дерева при этом сохраняются, и никаких дополнительных действий не требуется. Однако, если родительская вершина красная, то нарушается требование того, что дети красной вершины должны быть черными, и нужно выполнить процедуру восстановления структуры красно-черного дерева. Обычно эта процедура называется восстановлением баланса или ребалансировкой. В алгоритме ребалансировки с деревом совершаются локальные действия, не меняющие упорядоченности его вершин. Действия бывают двух типов:

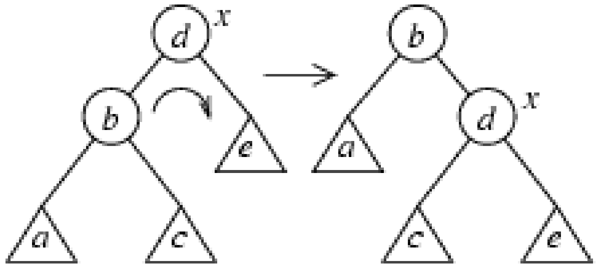
- перекрашивание вершин дерева,
- вращение вершины дерева влево или вправо.

Операция вращения вершины дерева

Определим преобразования вращения. Левое вращение вершины x :



Правое вращение вершины *x* :



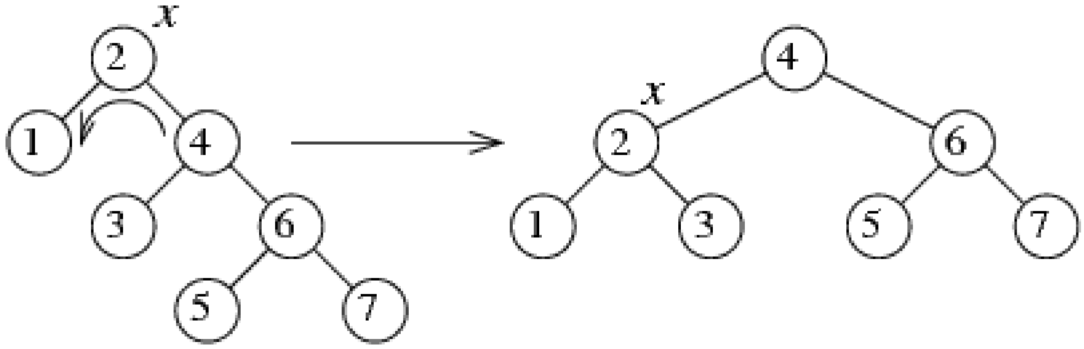
По рисункам видно, что оба преобразования не меняют упорядоченности вершин дерева. Преобразования носят локальный характер, при их выполнении меняется лишь фиксированное число ссылок в вешинах дерева вблизи вершины, которая "вращается" влево или вправо. Запишем на псевдокоде процедуру вращения вершины *x* влево.

```
void rotateLeft(
    Вход: TreeNode* x
)
начало алгоритма
    TreeNode* y = x->right; // y -- правый сын x
    утв: y != 0

    // Узлы x и y меняются местами (подчиненный "y" бывшего
    // начальника "x" становится новым начальником)
    // Делаем y корнем модифицированного поддерева
    TreeNode* p = x->parent;
    y->parent = p;
    если (x == p->left) // x -- левый сын своего отца
        p->left = y;
    иначе               // x -- правый сын своего отца
        p->right = y;
    конец если

    // Бывший левый сын узла "y" становится правым сыном узла
"x"
    x->right = y->left;
    если (y->left != 0)
        y->left->parent = x;
    конец если

    // Бывший начальник "x" становится подчиненным нового
начальника "y"
    y->left = x;
    x->parent = y;
```

конец алгоритма

Правое вращение вершины *x* реализуется аналогично.

Операция левого или правого вращения вершины во многих случаях позволяет "исправить" балансировку дерева, как показывает следующий пример. В нем мы применяем вращение вершины *x* влево.

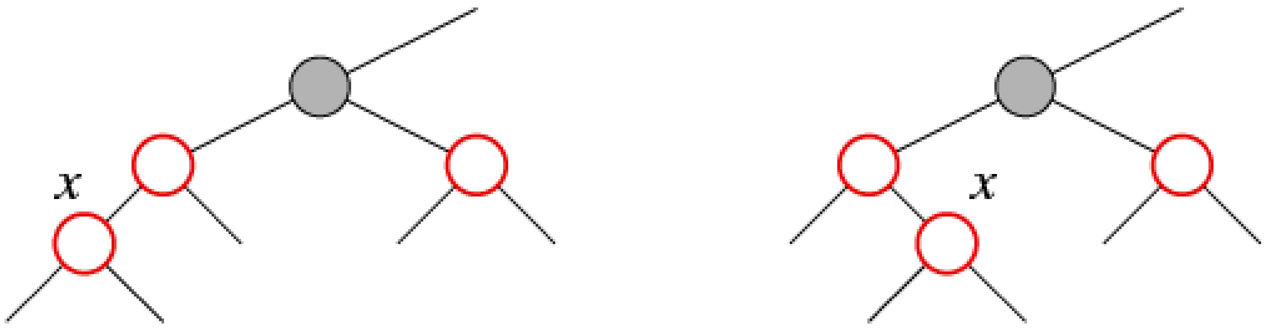


Восстановление структуры красно-черного дерева после добавления вершины (ребалансировка)

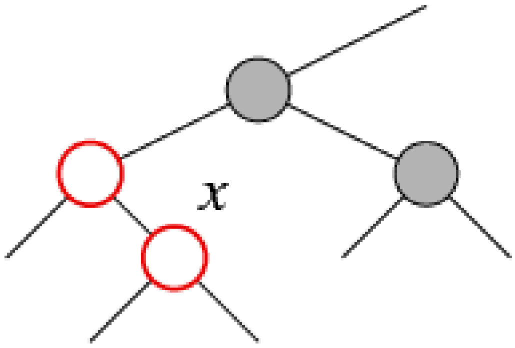
При добавлении вершины в дереве поиска новая вершина добавляется как терминальная после выполнения операции поиска. В красно-черном дереве новая вершина окрашивается изначально в красный цвет. Если ее родительская вершина черная, то все свойства красно-черного дерева выполняются и алгоритм добавления на этом заканчивается. Если же родительская вершина красная, то нарушается второе свойство в определении красно-черного дерева: у красной вершины дети должны быть черными. В этом случае выполняется процедура ребалансировки (восстановления структуры красно-черного дерева). Процедура ребалансировки носит итеративный характер. В ней рассматривается текущий узел *x* красного цвета, родительский узел (отец) которого тоже красный. В процессе ребалансировки метка *x* может перемещаться вверх по дереву, так что время восстановления не превосходит фиксированной константы, умноженной на высоту дерева, т.е. равно $O(\log n)$.

В процедуре ребалансировки рассматриваются 6 различных случаев. В случаях 1-3 отец узла *x* является **левым сыном** своего отца, т.е. деда *x*.

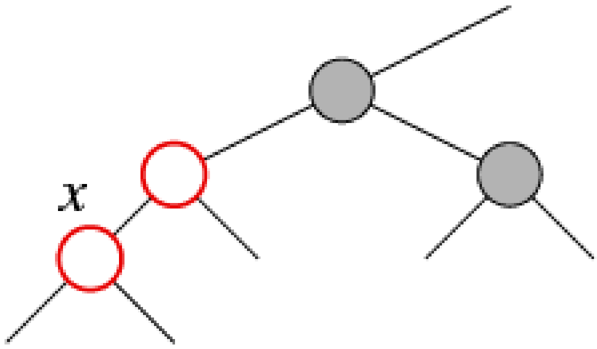
1. У узла *x* есть красный "дядя" (брат отца *x*). При этом *x* может быть левым или правым сыном своего отца — случаи эти рассматриваются аналогично.



2. У узла *x* "дядя" (брат отца) либо черный, либо его вообще нет (т.е. сыном является лист или внешний узел, который мы также считаем черным), и узел *x* является **правым сыном** своего отца.



3. У узла *x* "дядя" (брат отца) либо черный, либо его вообще нет, и узел *x* является **левым сыном** своего отца.



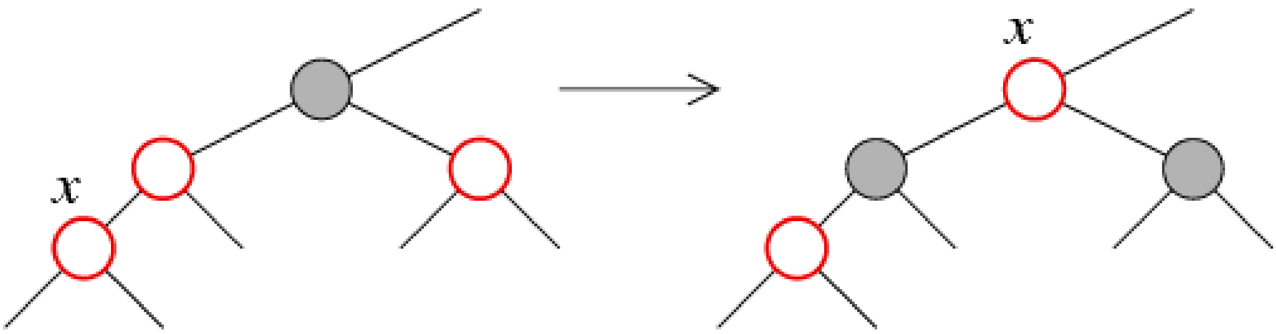
Случаи 4-6 зеркально симметричны случаям 1-3, в них отец узла *x* является **правым сыном** своего отца, т.е. деда *x*. Рассматриваются они аналогично случаям 1-3 заменой слова "левый" на "правый" и обратно, так что мы их описывать не будем.

Укажем, какие действия выполняются в каждом из случаев 1-3 для восстановления балансировки дерева.

Случай 1 (красный дядя)

Этот случай наиболее прост:

- перекрашиваем отца и дядю узла *x* в черный цвет;
- перекрашиваем деда узла *x* в красный цвет;
- перемещаем метку *x* вверх по дереву на деда узла *x*: и переходим в цикле к восстановлению нового узла *x* (деда предыдущего *x*) *x* = *x*->parent->parent

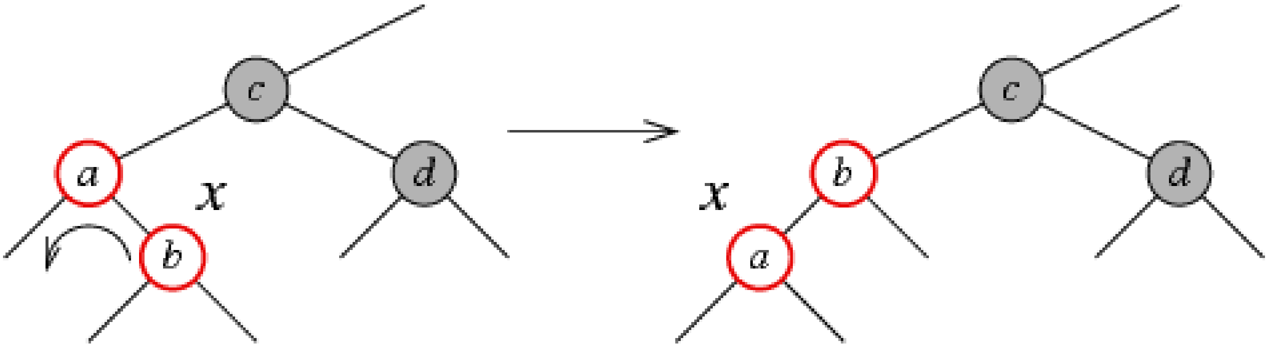


Цикл завершается, либо когда мы достигли корня дерева (если корень был перекрашен в красный цвет, то надо перекрасить его обратно в черный), либо когда отец узла *x* черный.

Случай 2 (дядя черный или его вообще нет, узел *x* является правым сыном своего отца)

Этот случай сводится к случаю 3 путем следующих преобразований:

- перемещаем метку *x* на вверх по дереву: *x* = *x*->parent;
- левое вращение *x*.

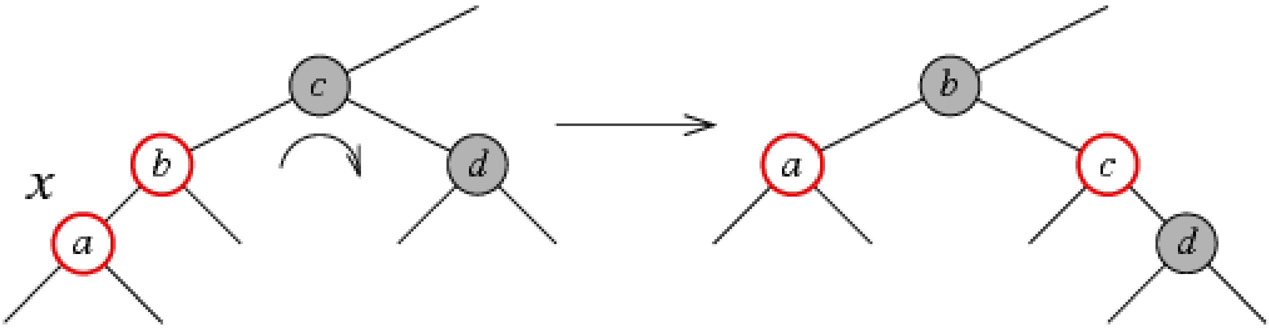


Случай 3 (дядя черный или его вообще нет, узел *x* является левым сыном своего отца)

В этом случае выполняются следующие действия:

- перекрашиваем отца узла *x* в черный цвет;
- перекрашиваем деда (т.е. отца отца) *x* в красный;
- выполняем правое вращение деда *x*.

На этом алгоритм восстановления балансировки заканчивается (в случае 3 цикл завершается).



Отметим, что всего в процедуре восстановления структуры красно-черного дерева (ребалансировки) может выполняться не более $O(h)$ операций; поскольку для красно-черного дерева $h \leq 2\log_2(n+1)$, получаем, что время выполнения ребалансировки равно $O(\log_2 n)$. В этой процедуре выполняются перекрашивание узлов дерева и операции вращения, причем общее число вращений — не больше двух (одно вращение в случае 3 и два в случае 2).

! Дополнительный материал !

1. АВЛ-деревья:

https://rsdn.org/article/alg/bintree/avl.xml

2. Структуры данных. АВЛ-дерево: *https://medium.com/@dimko1/структуры-даных-avl-дерево-7f8739e8faf9*
3. Красно-черное дерево: *https://intellect.icu/krasno-chernoe-derevo-derevo-poiska-4612*
4. Красно-черное дерево: *https://www.epaperpress.com/sortsearchRussian/rbt.html*
5. Алгоритмы, обход дерева: *https://medium.com/@dimko1/алгоритмы-обход-дерева-ed54848c2d47*

2 Задания

Обязательное 1*: Написать программу, которая создает АВЛ-дерево и реализует следующие операции:*

- *добавление нового узла*
- *поиск минимального элемента*
- *поиск максимального элемента*
- *поиск по значению*
- *удаление элемента*
- *обход в глубину (pre-order)*
- *обход в глубину (in-order)*
- *обход в глубину (post-order)*
- *вывод высоты дерева*

(Пример задания в файле АиСД_лаба6_пример)

! Контрольные вопросы !

1. Определение понятия АВЛ-дерево.
2. Как осуществляется операция балансировки дерева?
3. Типы вращений при балансировке.
4. Определение понятия 2-3 дерево.
5. Свойства 2-3 дерева.
6. Опишите алгоритм вставки в 2-3дерево элемента с ключом.
7. Определение понятия Б-дерево.
8. Где используются Б-деревья?
9. Условия для балансировки красно-черных деревьев.

Содержание отчёта

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Ответы на контрольные вопросы.
6. Выводы.