

# Алгоритмы и структуры данных-1

## Лекция 11

Дата: 04.12.2023

---

Программная инженерия, 2 курс  
2023-2024 учебный год

**Нестеров Р.А.**, PhD, ст. преподаватель  
департамент программной инженерии ФКН

27 дней до  
Нового Года!

# План

---

Сбалансированные деревья поиска

2-3-4 деревья  $\leftrightarrow$  Красно-черные деревья.

Операции, нарушающие баланс

Ленивые вычисления. Ленивое удаление

AVL-деревья

ПО ВЫСОТЕ

БАЛАНС

Красно-черные  
деревья

по длине путей

$BB[\alpha]$ -деревья

по весу

# Сбалансированные деревья поиска

---

Высота AVL-дерева  $\log(n + 1) - 1 \leq h_{AVL} \leq 1.44 \log(n + 1) - 1.33$

Высота красно-черного дерева  $\log(n + 1) \leq h_{RB} \leq 2 \cdot \log(n + 1)$

# Сбалансированные деревья поиска

---

Высота AVL-дерева  $\log(n + 1) - 1 \leq h_{AVL} \leq 1.44 \log(n + 1) - 1.33$

Высота красно-черного дерева  $\log(n + 1) \leq h_{RB} \leq 2 \cdot \log(n + 1)$

*Стремимся к логарифмической  
высоте идеального бинарного дерева!*

# Почему красное и черное?

---

"A lot of people ask why did we use the name red-black. Well, we invented this data structure, this way of looking at balanced trees, at Xerox PARC which was the home of the personal computer and many other innovations that we live with today entering graphic user interfaces, ethernet and object-oriented programmings and many other things. But one of the things that was invented there was laser printing and we were very excited to have nearby color laser printer that could print things out in color and out of the colors the red looked the best. So, that's why we picked the color red to distinguish red links, the types of links, in three nodes. So, that's an answer to the question for people that have been asking."

*Robert Sedgewick, 2012*

# Почему красное и черное?

---

“A lot of people ask why did we use the name red–black. Well, we invented this data structure, this way of looking at balanced trees, at Xerox PARC which was the home of the personal computer and many other innovations that we live with today entering graphic user interfaces, ethernet and object-oriented programmings and many other things. But one of the things that was invented there was laser printing and we were very excited to have nearby color laser printer that could print things out in color and out of the colors the red looked the best. So, that’s why we picked the color red to distinguish red links, the types of links, in three nodes. So, that’s an answer to the question for people that have been asking.”

*Robert Sedgewick, 2012*

# Почему красное и черное?

---

"We had red and black pens for drawing the trees."

*Leonidas John Guibas, 2011*



# Почему красное и черное?

## A DICHROMATIC FRAMEWORK FOR BALANCED TREES

Leo J. Guibas  
Xerox Palo Alto Research Center,  
Palo Alto, California, and  
Carnegie-Mellon University

and

Robert Sedgwick\*  
Program in Computer Science  
Brown University  
Providence, R. I.

### ABSTRACT

In this paper we present a uniform framework for the implementation and study of balanced tree algorithms. We show how to imbed in this framework the best known balanced tree techniques and then use the framework to develop new algorithms which perform the update and rebalancing in one pass, on the way down towards a leaf. We conclude with a study of performance issues and concurrent updating.

### 0. Introduction

Balanced trees are among the oldest and most widely used data structures for searching. These trees allow a wide variety of operations, such as search, insertion, deletion, merging, and splitting to be performed in time  $O(\lg N)$ , where  $N$  denotes the size of the tree [AHH-], [K4]. (Throughout the paper  $\lg$  will denote log to the base 2.) A number of different types of balanced trees have been proposed, and while the related algorithms are often conceptually simple, they have proven cumbersome to implement in practice. Also, the variety of such trees and the lack of good analytic results describing their performance has made it difficult to decide which is best in a given situation.

In this paper we present a uniform framework for the implementation and study of balanced tree algorithms. The framework deals exclusively with binary trees which contain two kinds of nodes: internal and external. Each internal node contains a key (chosen from a linear order) and has two links to other nodes (internal or external). External nodes contain no keys and have null links. If such a tree is traversed in symmetric order [K4] then the internal nodes will be visited in increasing order of their key. A second defining feature of the framework is that it allows one bit per node, called the *color* of the node, to store balance information. We will use *red* and *black* as the two colors. In section 1 we further elaborate upon this dichromatic framework and show how to imbed in it the best known balanced tree algorithms. In doing so, we will discover surprising new and efficient implementations of these techniques.

In section 2 we use the framework to develop new balanced tree algorithms which perform the update and rebalancing in one pass, on

the way down towards a leaf. As we will see, this has a number of significant advantages over the older methods. We shall examine a number of variations on a common theme and exhibit full implementations which are notable for their brevity. One implementation is examined carefully, and some properties about its behavior are proved.

In both sections 1 and 2 particular attention is paid to practical implementation issues, and complete implementations are given for all of the important algorithms. This is significant because one measure under which balanced tree algorithms can differ greatly is the amount of code required to actually implement them.

Section 3 deals with the analysis of the algorithms. New results are given for the worst case performance, and a technique for studying the average case is described. While no balanced tree algorithm has yet satisfactorily submitted to an average case analysis, empirical results are given which show that the various algorithms differ only slightly in performance. One implication of this is that the top-down algorithms of section 2 can be recommended for most applications because of their simplicity.

Finally, in section 4, we discuss some other properties of the trees. In particular, a one-pass top down deletion algorithm is presented. In addition, we consider how to decouple the balancing from the updating operations and we explore parallel updating.

### 1. The Uniform Framework

In this section we present a uniform framework for describing balanced trees. We show how to embed in this framework the most widely used balanced tree schemes, namely B-trees [BMM-], and AVL trees [AVL]. In fact, this embedding will give us interesting and novel implementations of these two schemes.

We consider rebalancing transformations which maintain the symmetric order of the keys and which are local to a small portion of the tree for obvious efficiency reasons. These transformations will change the structure of the tree in the same way as the single and double rotations used by AVL trees [K4]. The difference between the various algorithms we discuss arises in the decision of *when* to rotate, and in the manipulation of the node colors.

For our first example, let us consider the implementation of 2-3 trees, the simplest type of B-tree. Recall that a 2-3 tree consists of 2-nodes, which have one key and two sons, 3-nodes, which have two

\* This work was done in part while this author was a Visiting Scientist at the Xerox Palo Alto Research Center and in part under support from the National Science Foundation, grant no. MCS75-23738.

## A dichromatic framework for balanced trees

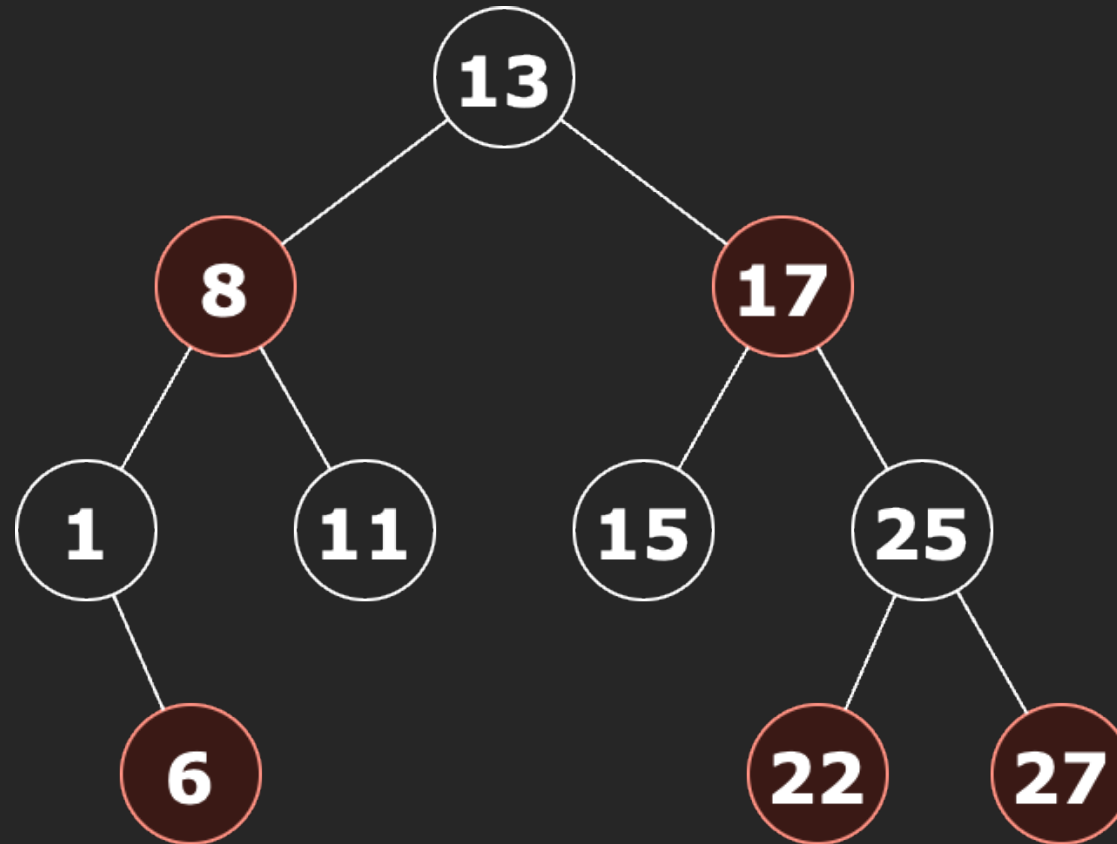
19th Annual Symposium  
on Foundations of Computer Science

IEEE, 1978

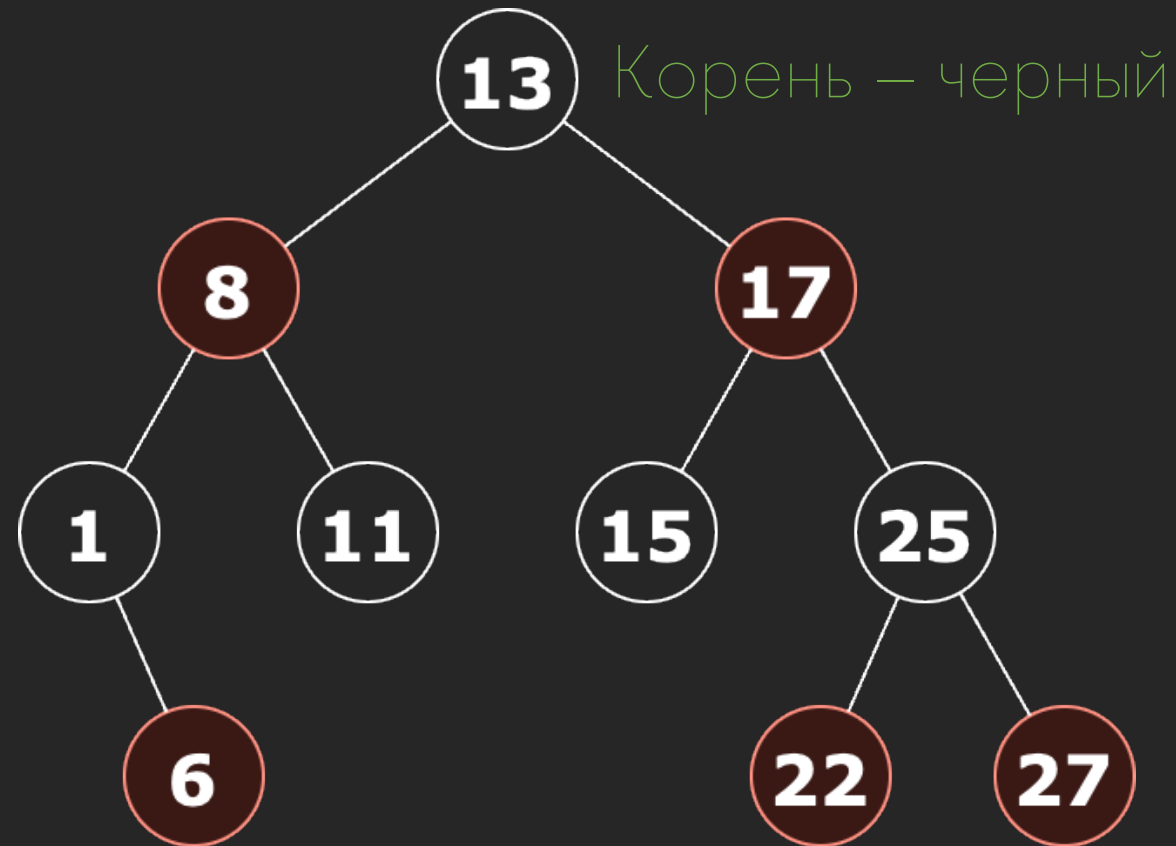
# Refresher

---

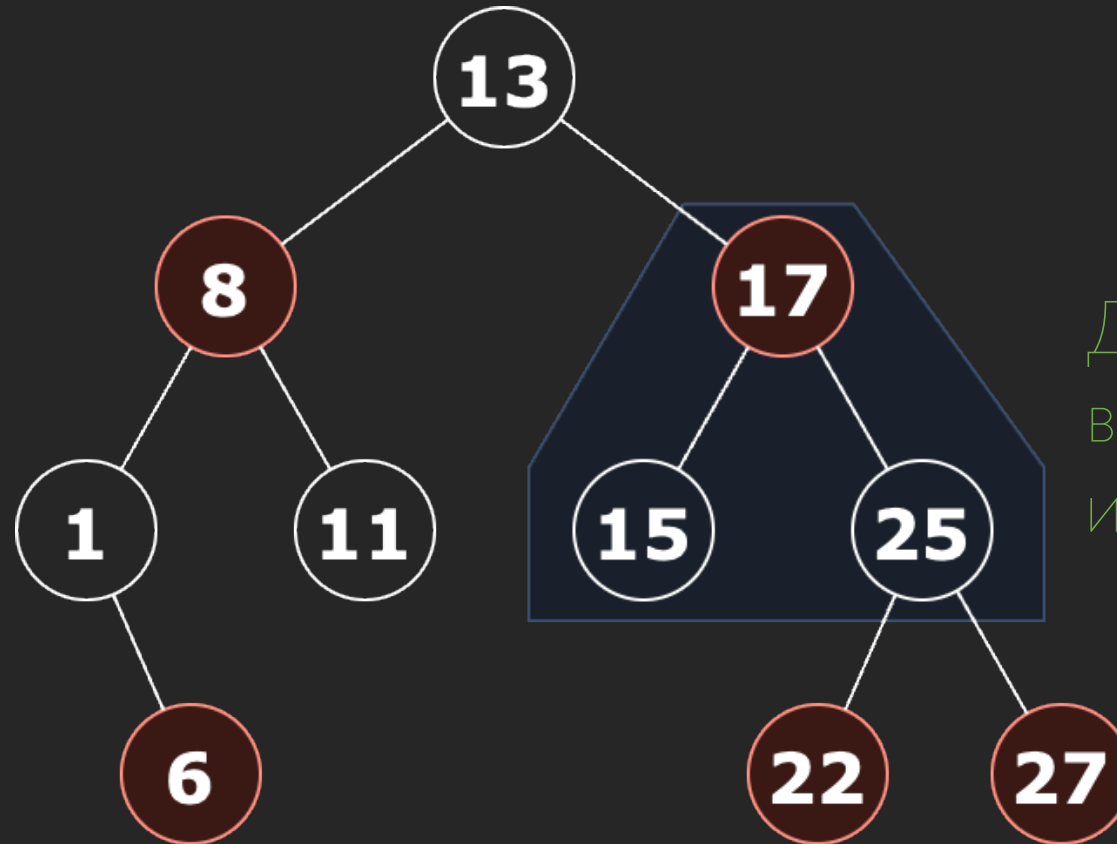
# Красно-черное дерево



# Красно-черное дерево

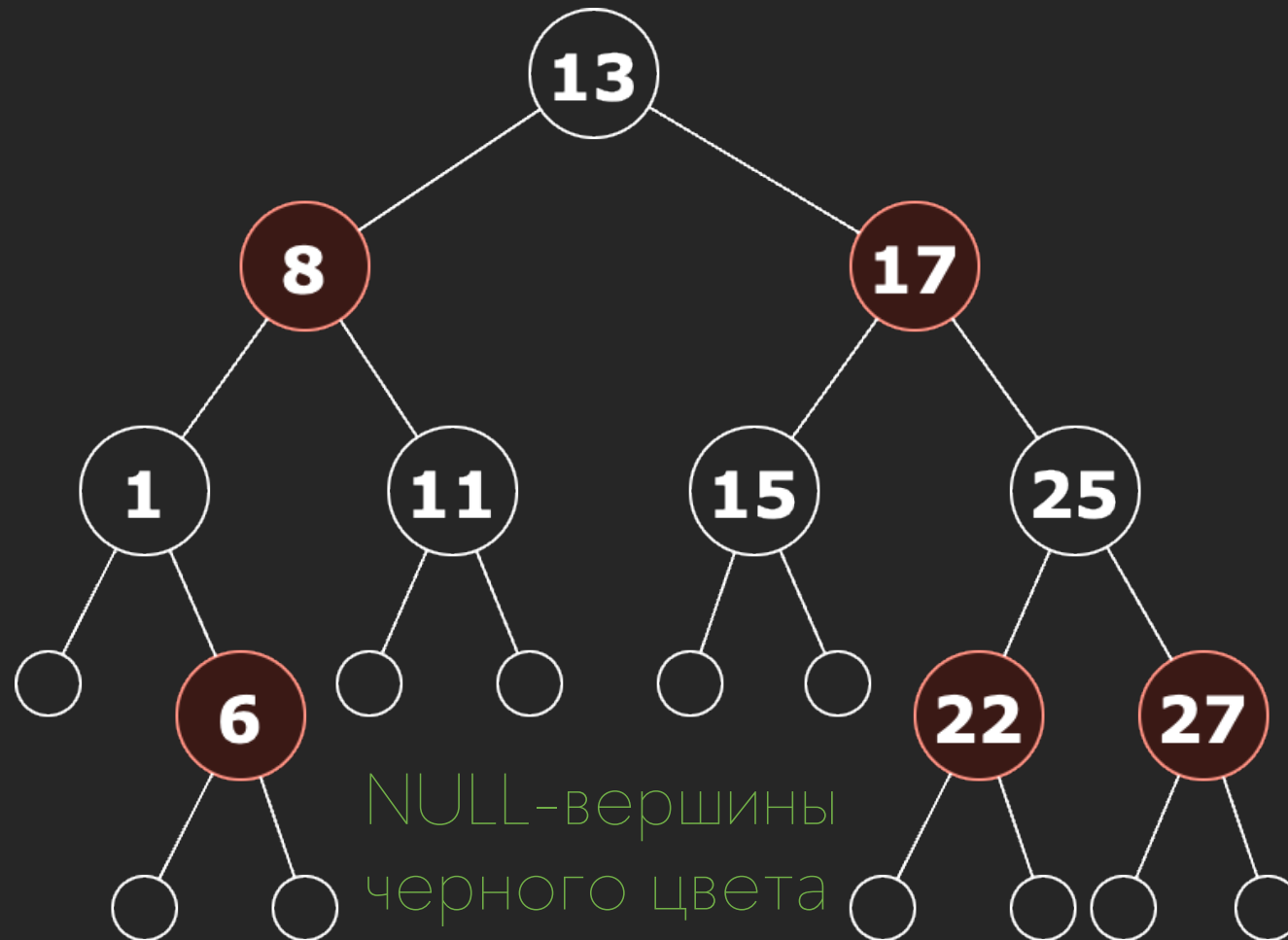


# Красно-черное дерево



Дети красной  
вершины всегда  
имеют черный цвет

# Красно-черное дерево



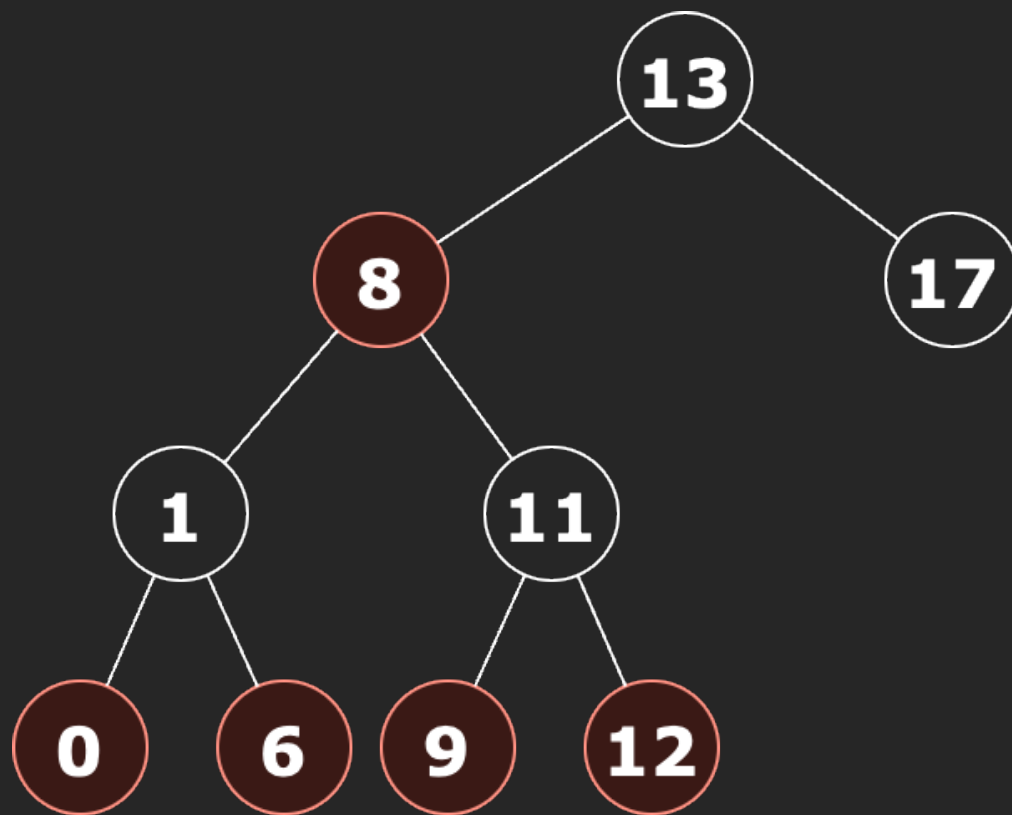
# Правила красно-черного дерева

---

1. Каждая вершина окрашена в один из цветов
2. Корень всегда окрашен в черный [правило корня]
3. NULL-вершины окрашены в черный
4. Потомки красной вершины – черные [правило красного]
5. Любой путь из вершины  $v$  до NULL-вершины содержит одинаковое число черных вершин [bh-правило]

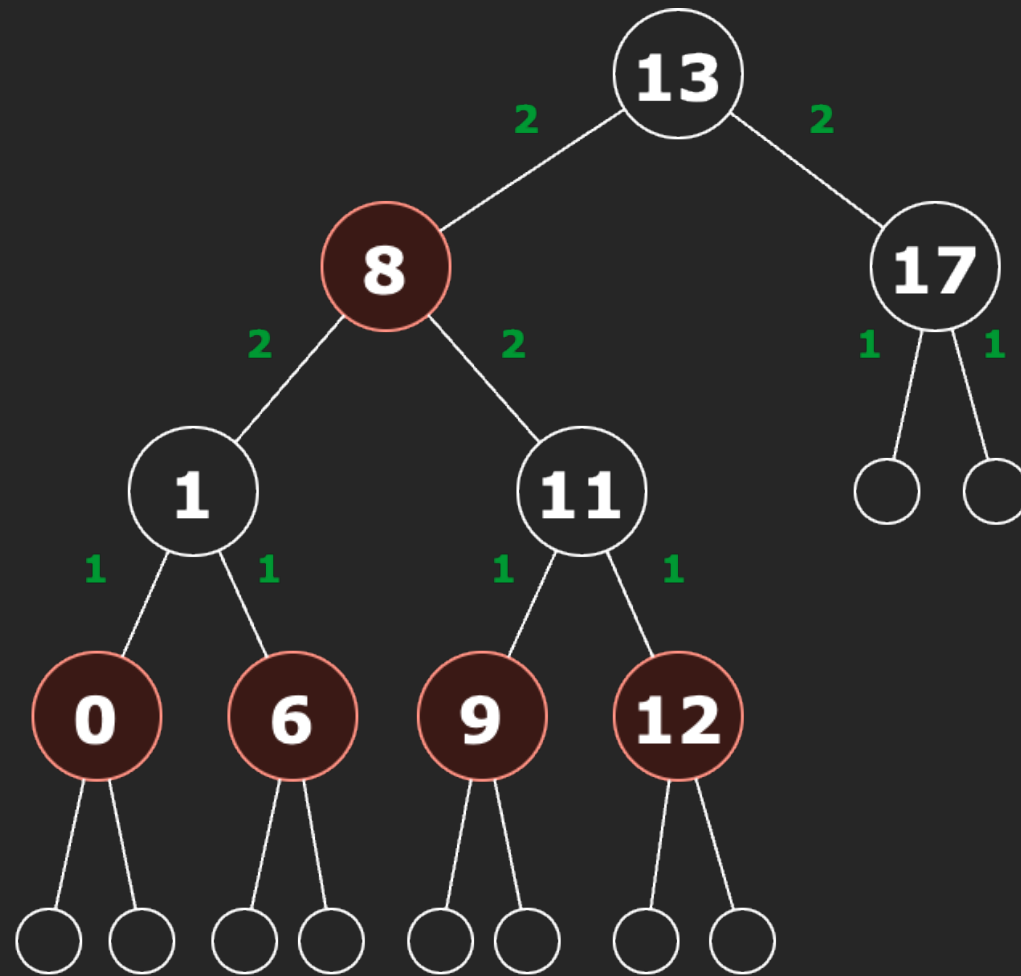
# Красно-черное дерево?

---





# Красно-черное дерево?



# Красно-черное дерево?

---

1. Условия баланса по длине путей в красно-черном дереве слабее, чем в AVL-дереве

# Красно-черное дерево?

---

1. Условия баланса по длине путей в красно-черном дереве слабее, чем в AVL-дереве
2. Любое AVL-дерево можно представить в виде красно-черного дерева

# Красно-черное дерево?

---

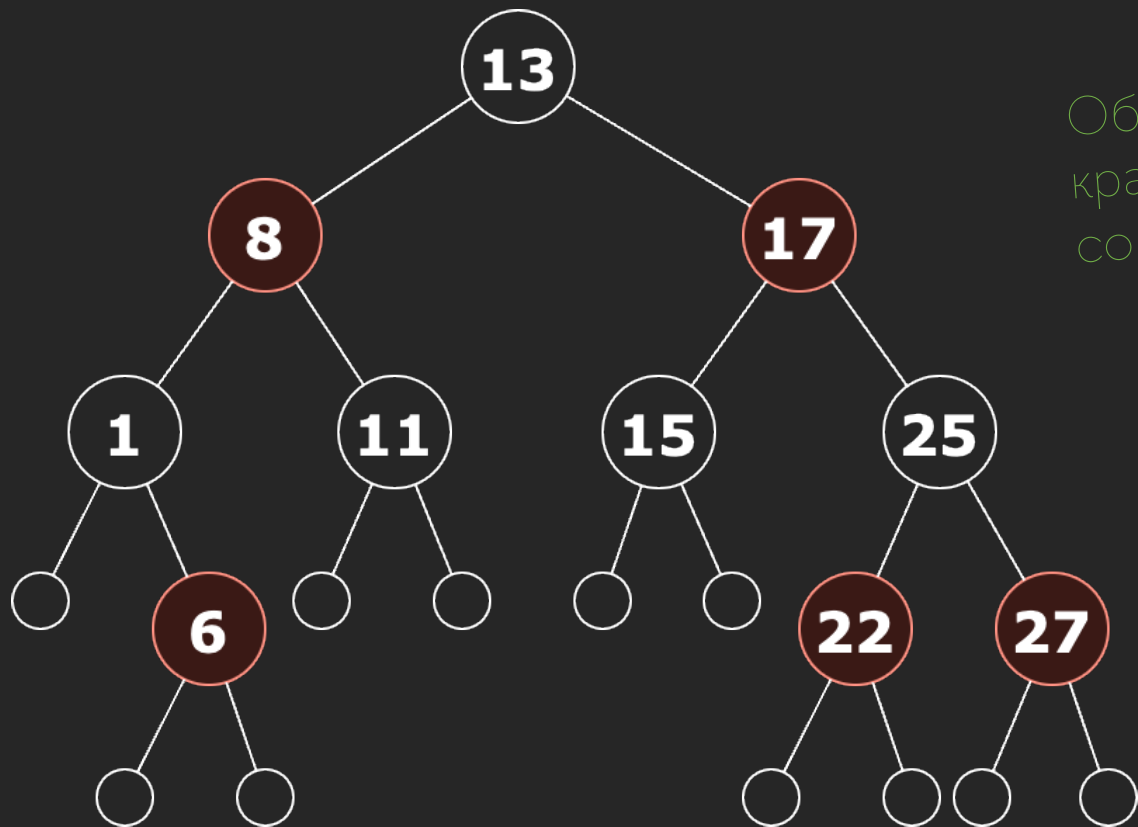
1. Условия баланса по длине путей в красно-черном дереве слабее, чем в AVL-дереве
2. Любое AVL-дерево можно представить в виде красно-черного дерева

AVL  $\Rightarrow$  RB  
RB  $\Rightarrow$  AVL

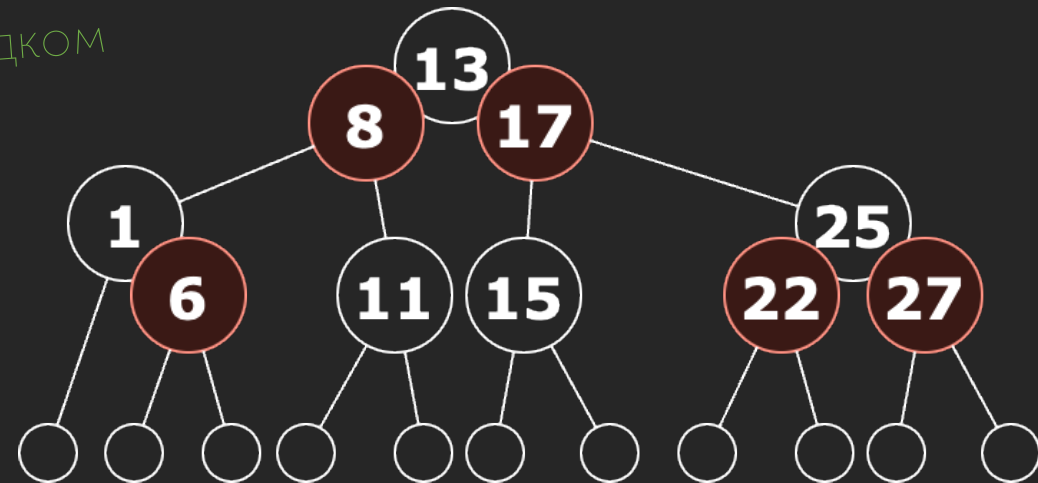
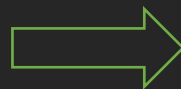
# Изометрия красно-черного дерева и 2-3-4 дерева

---

# КЧД $\leftrightarrow$ 2-3-4 дерево

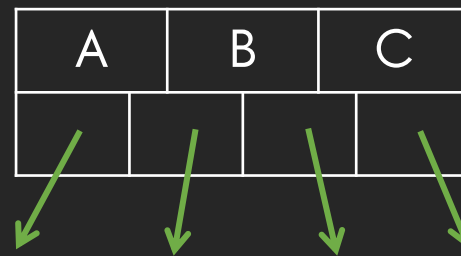
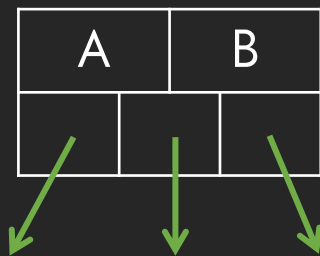
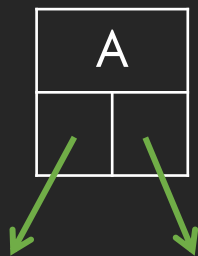


Объединим  
красные вершины  
со своим предком



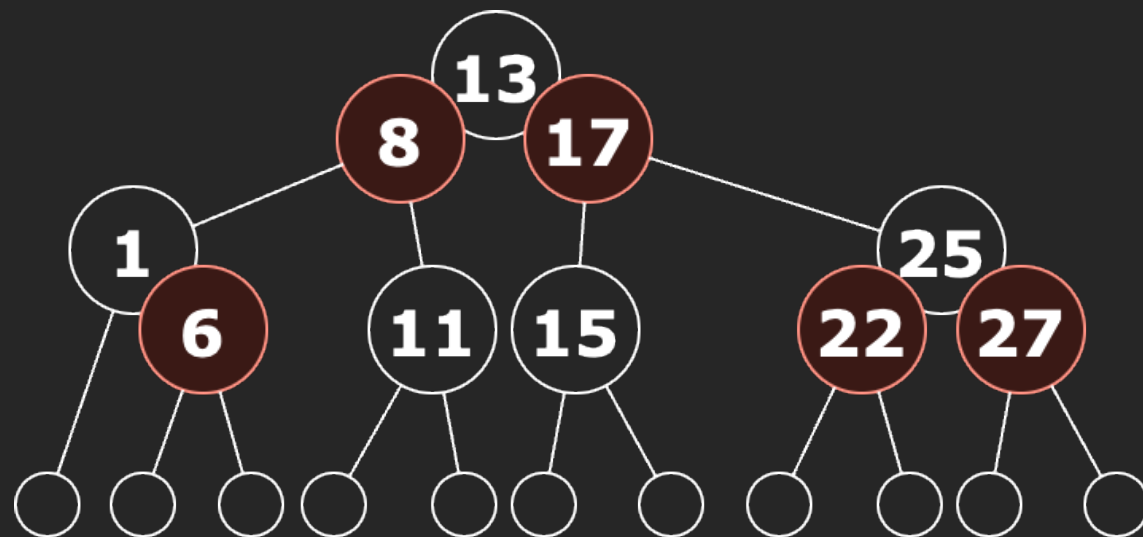
# 2-3-4 дерево

Много-проходное (ветвящееся) дерево, в вершинах которого может быть от одного до трех ключей – от двух до четырех потомков



# 2-3-4 дерево

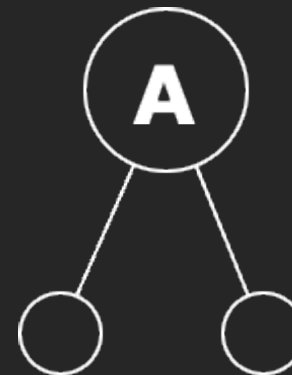
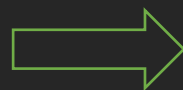
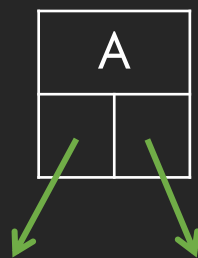
Много-проходное (ветвящееся) дерево, в вершинах которого может быть от одного до трех ключей – от двух до четырех потомков





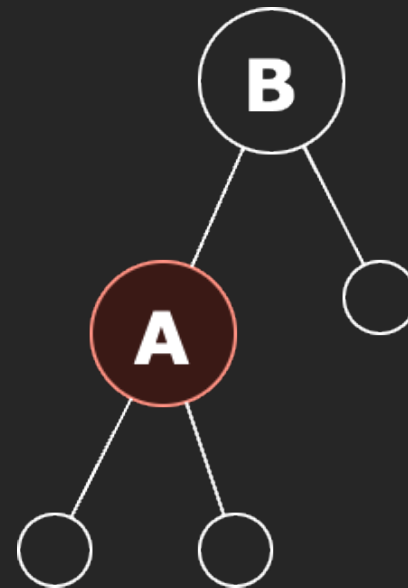
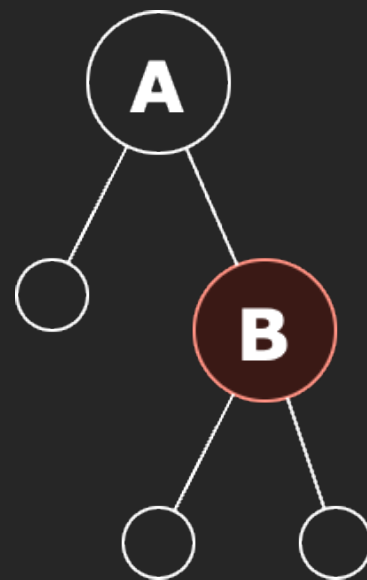
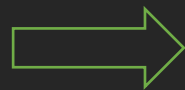
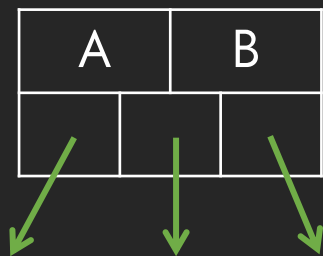
# 2-3-4 дерево. 2-вершина

КЧД является изометрией 2-3-4 дерева



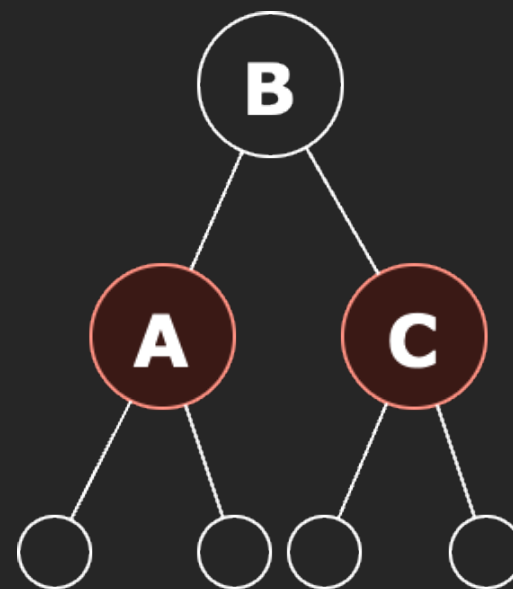
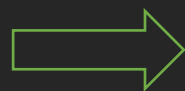
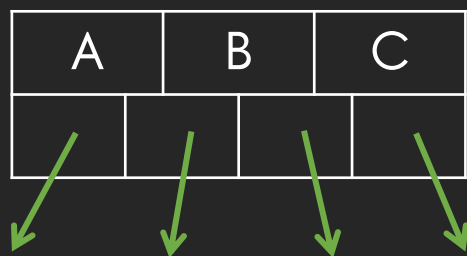
# 2-3-4 дерево. 3-вершина

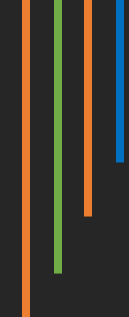
КЧД является изометрией 2-3-4 дерева



# 2-3-4 дерево. 4-вершина

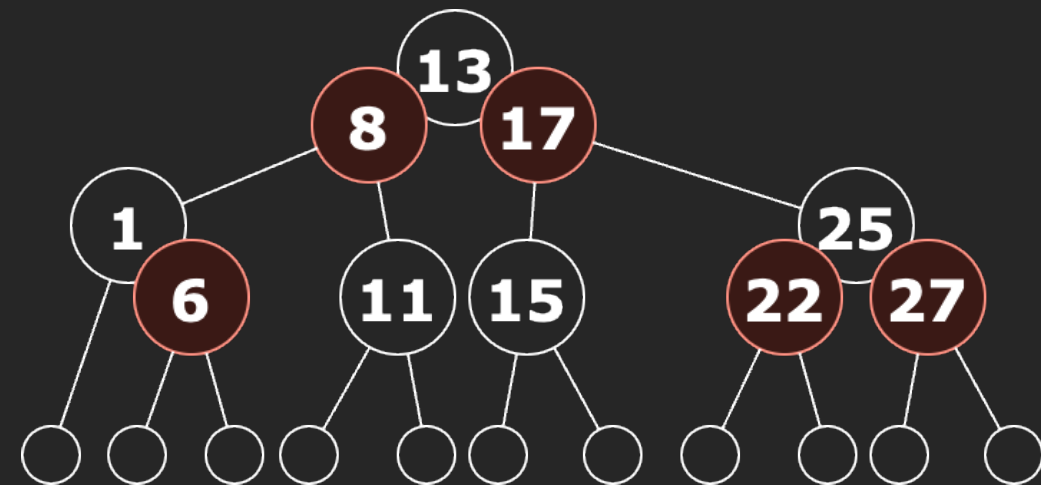
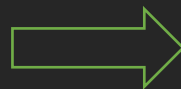
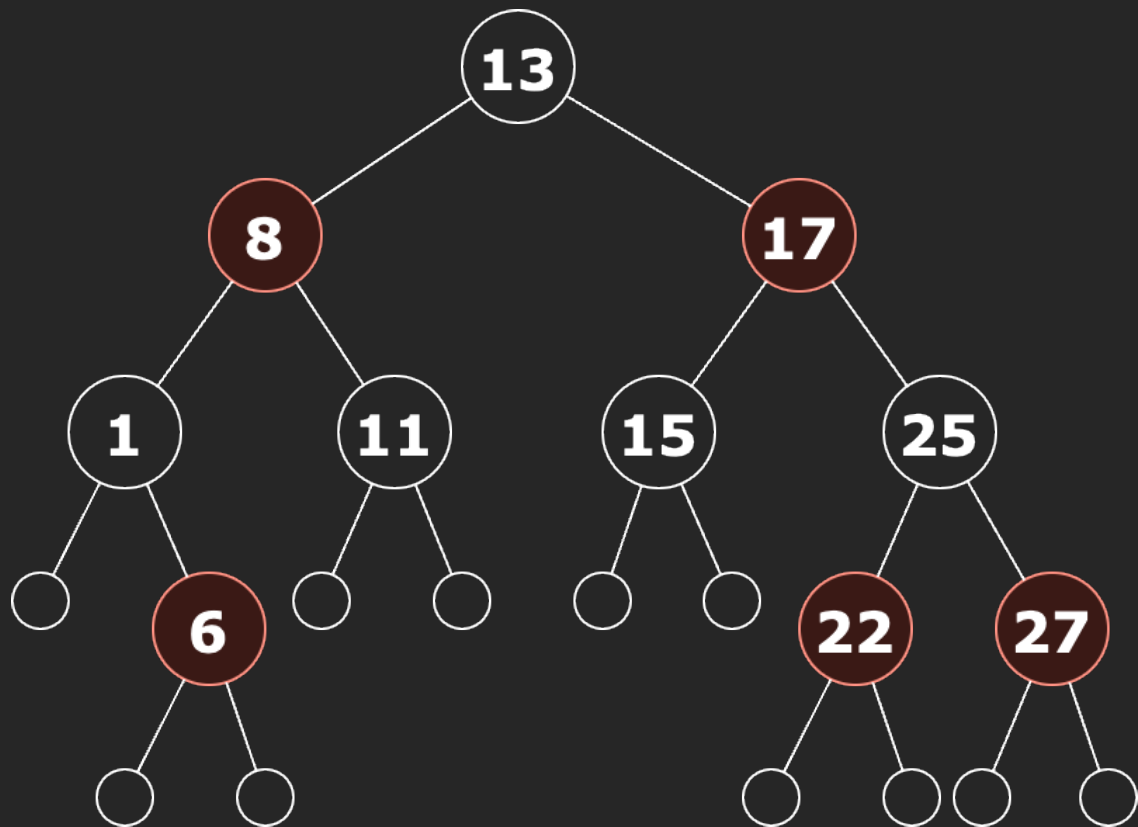
КЧД является изометрией 2-3-4 дерева



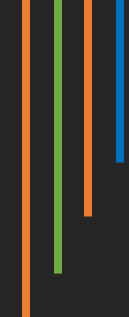


Цвет вершины нужен, чтобы из КЧД  
снова получить 2-3-4 дерево

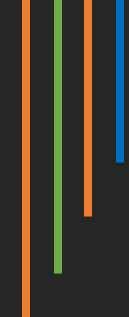
# КЧД $\leftrightarrow$ 2-3-4 дерево



Все листья на одном уровне!



2-3-4 дерево – член более  
широкого класса В-деревьев...



2-3-4 дерево – член более  
широкого класса В-деревьев...

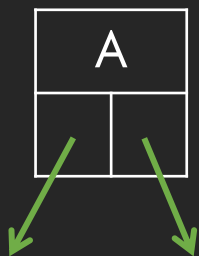
...которые сбалансированы по  
построению

# Вставка ключей в 2-вершину

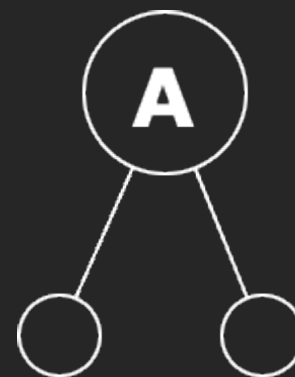
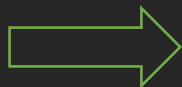
---



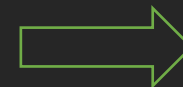
# Вставка в КЧД. 2-вершина



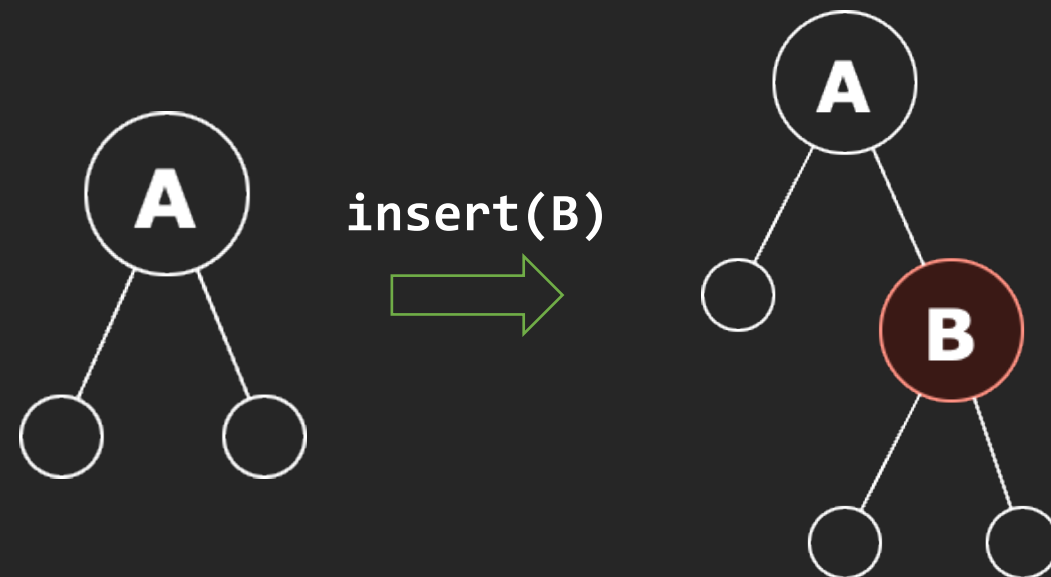
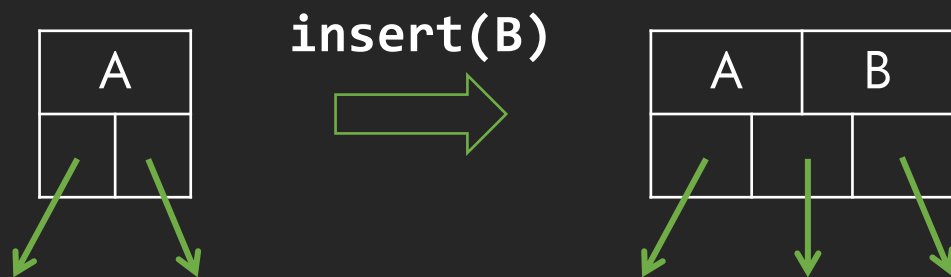
insert(B)



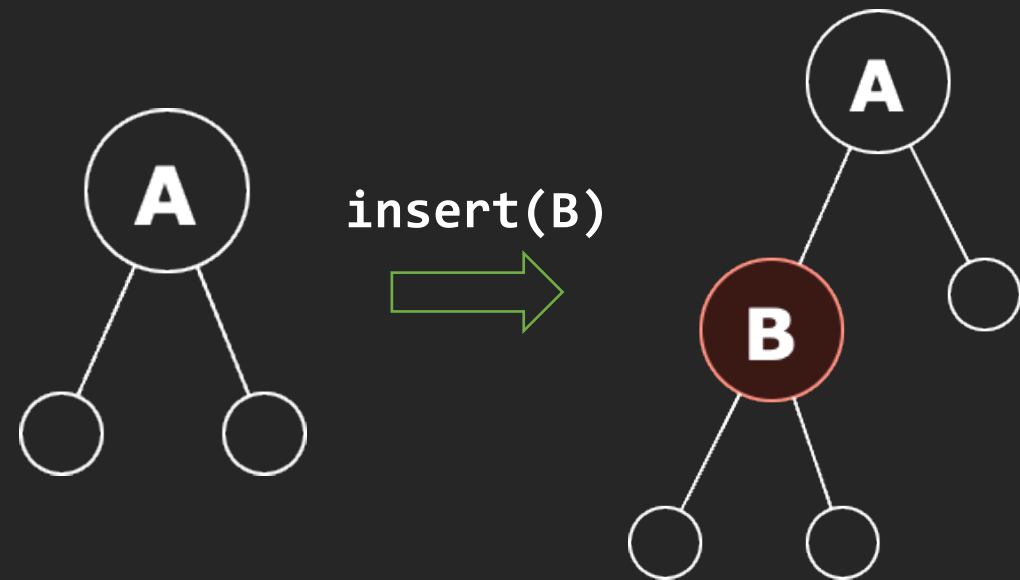
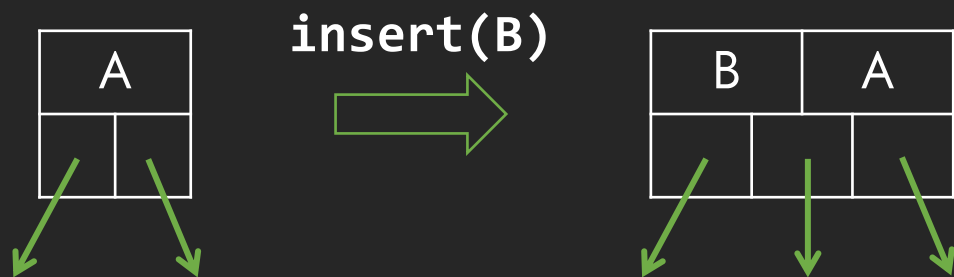
insert(B)



# Вставка в КЧД. 2-вершина



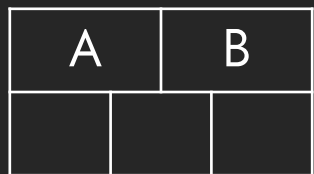
# Вставка в КЧД. 2-вершина



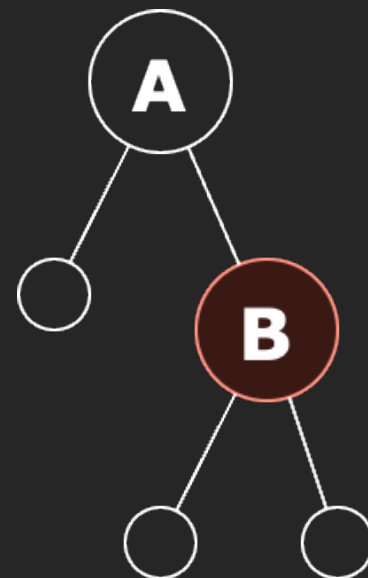
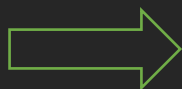
# Вставка ключей в 3-вершину

---

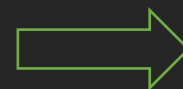
# Вставка в КЧД. 3-вершина



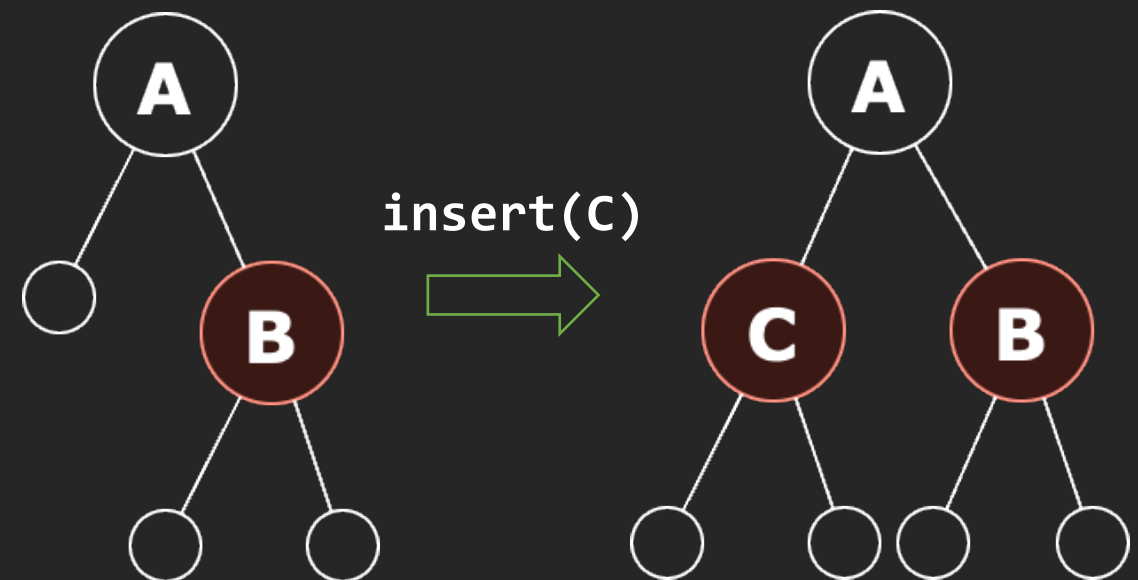
insert(C)



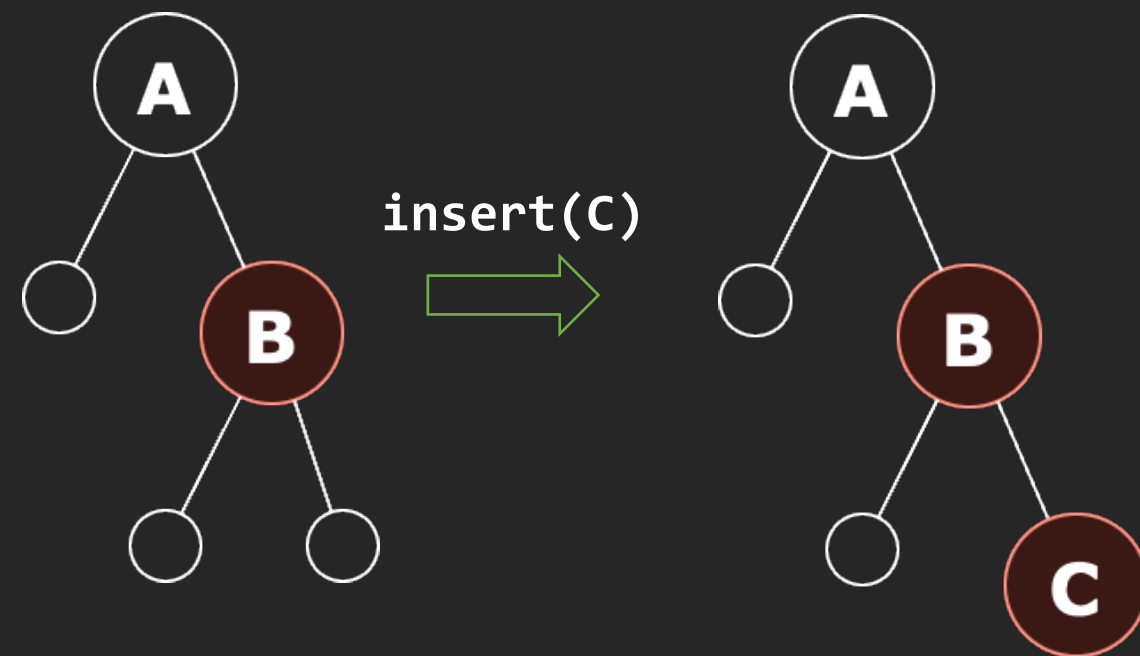
insert(C)



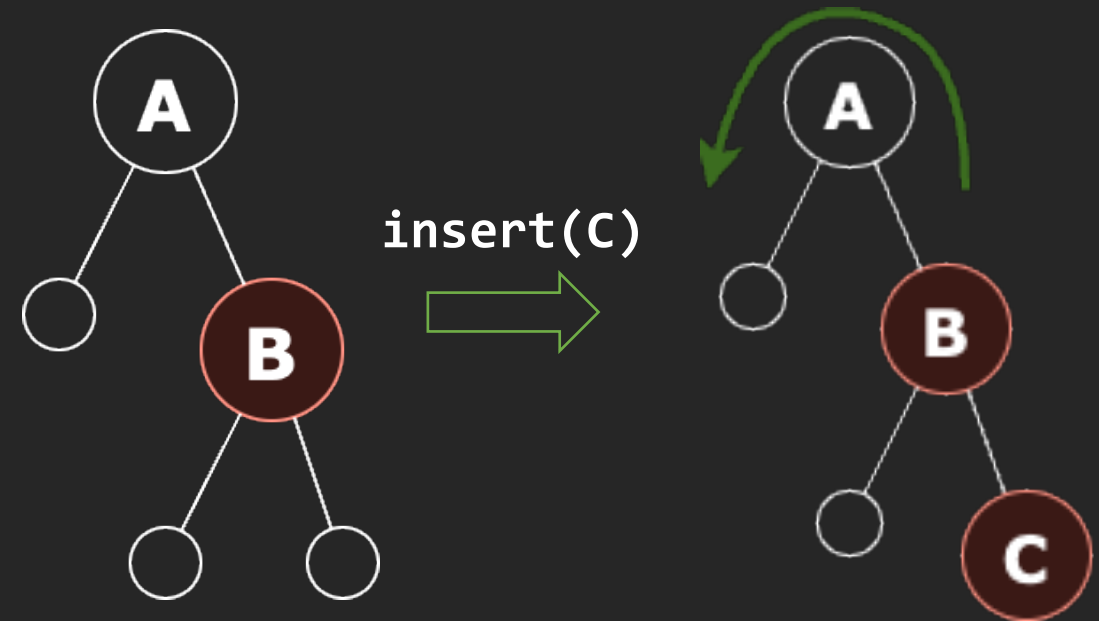
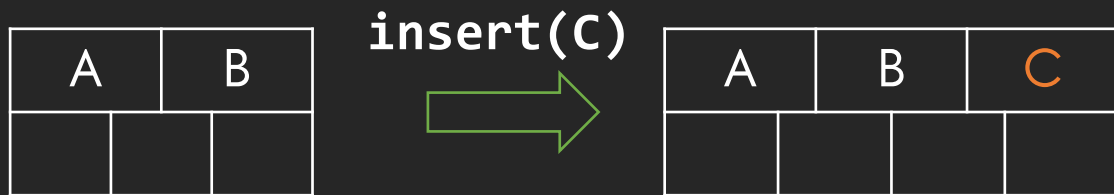
# Вставка в КЧД. 3-вершина



# Вставка в КЧД. 3-вершина



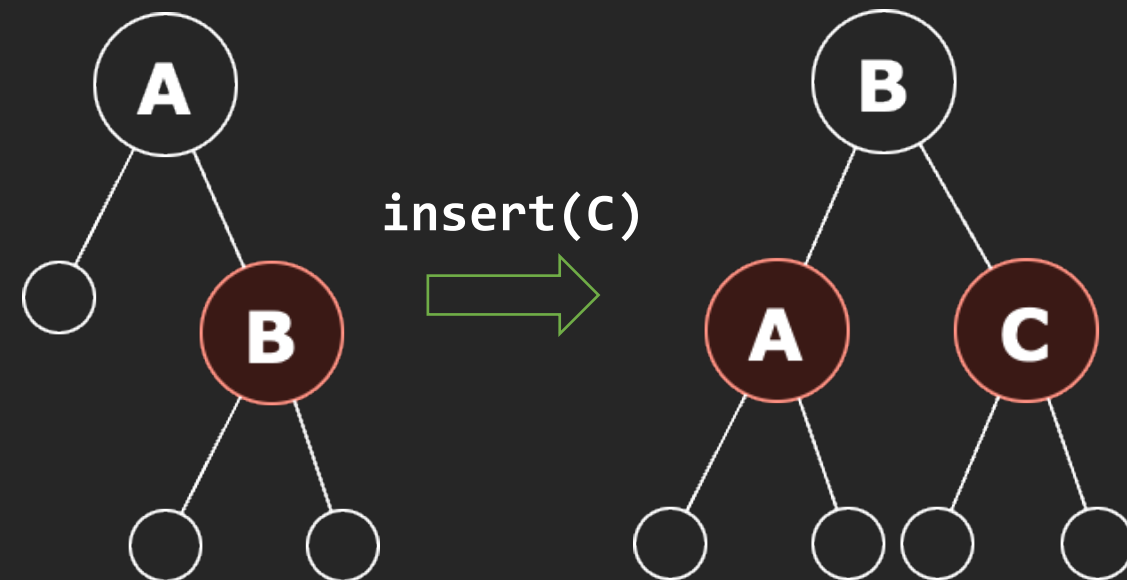
# Вставка в КЧД. 3-вершина



`leftRotate(B) + recolor(A,B)`

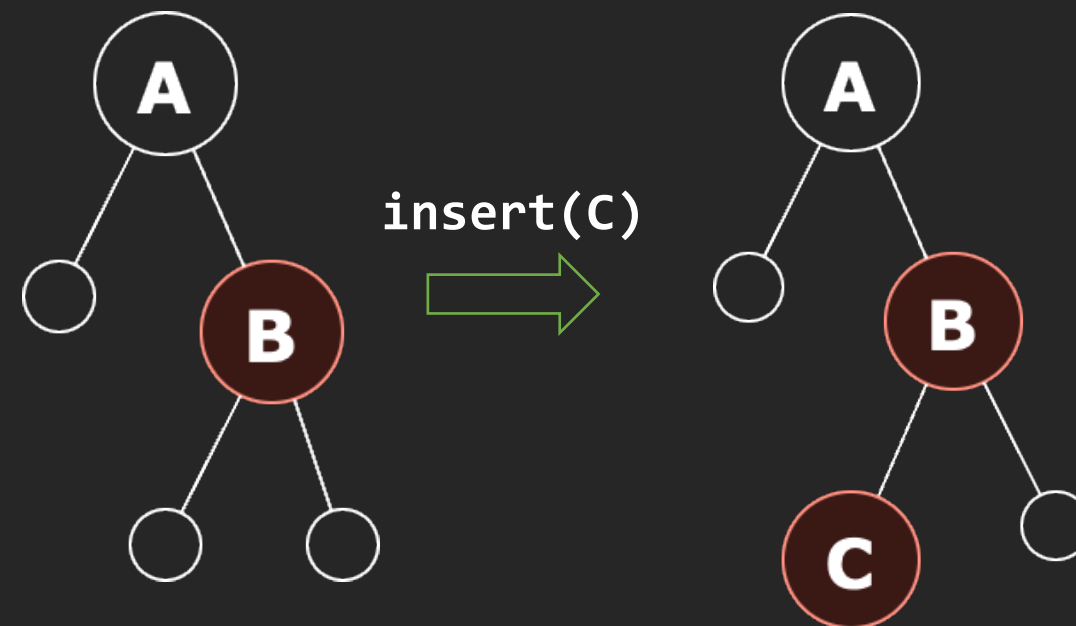


# Вставка в КЧД. 3-вершина

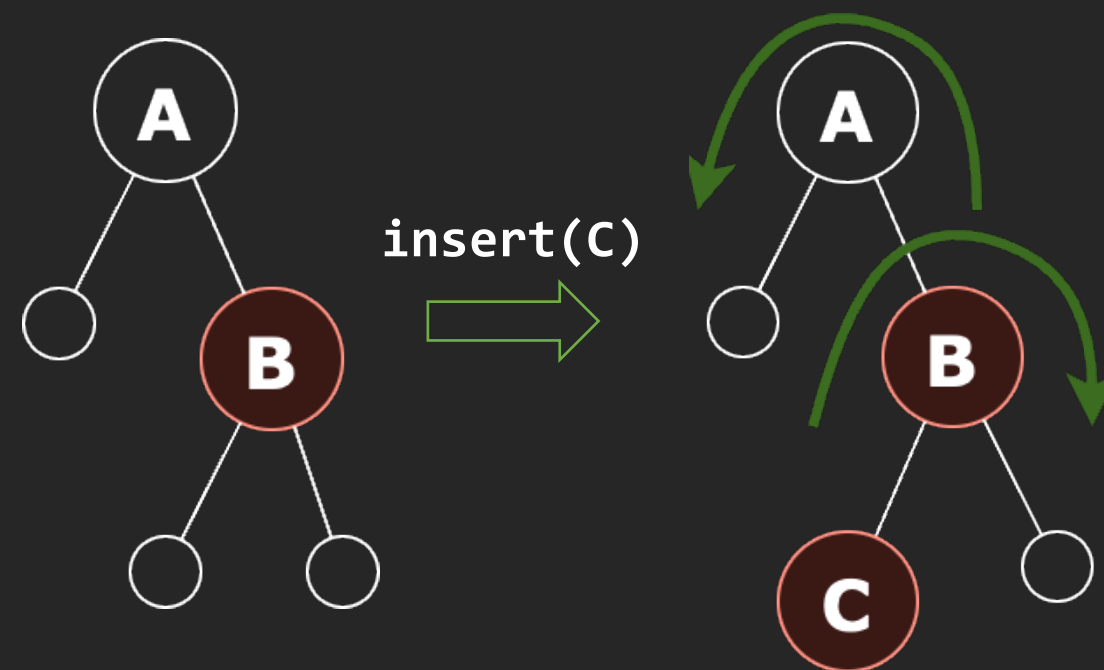


`leftRotate(B) + recolor(A,B)`

# Вставка в КЧД. 3-вершина

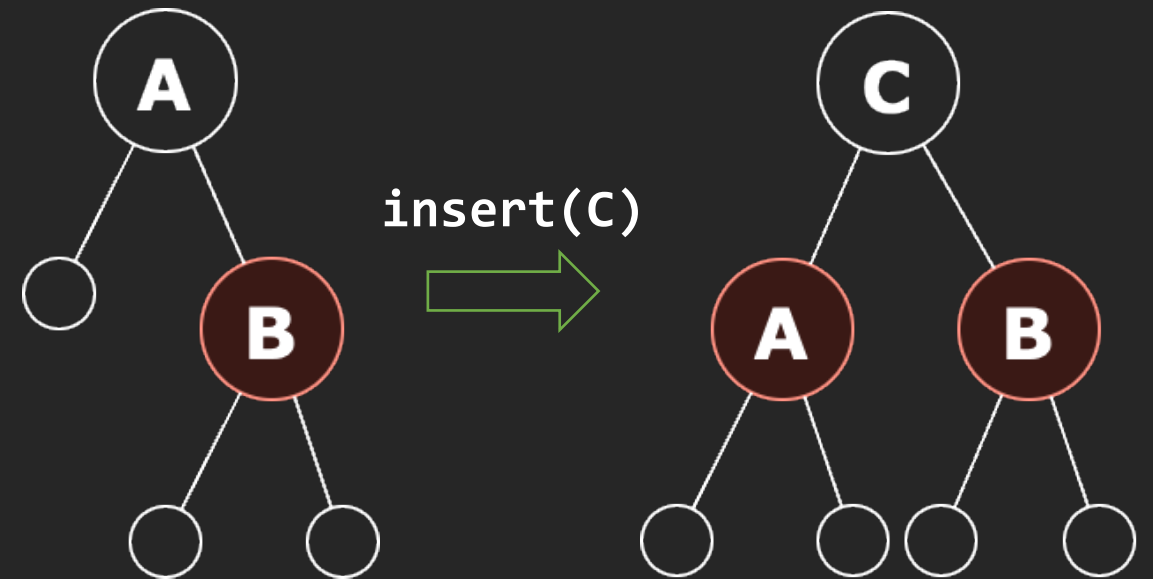
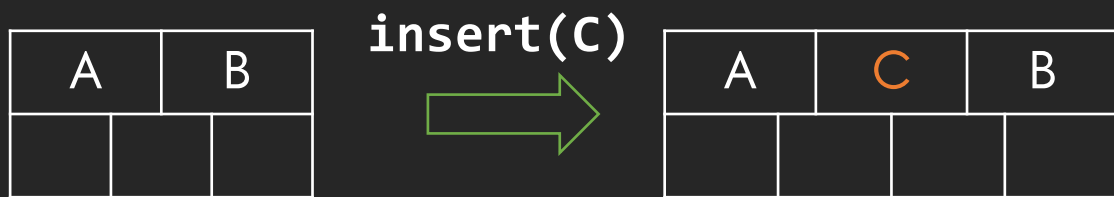


# Вставка в КЧД. 3-вершина

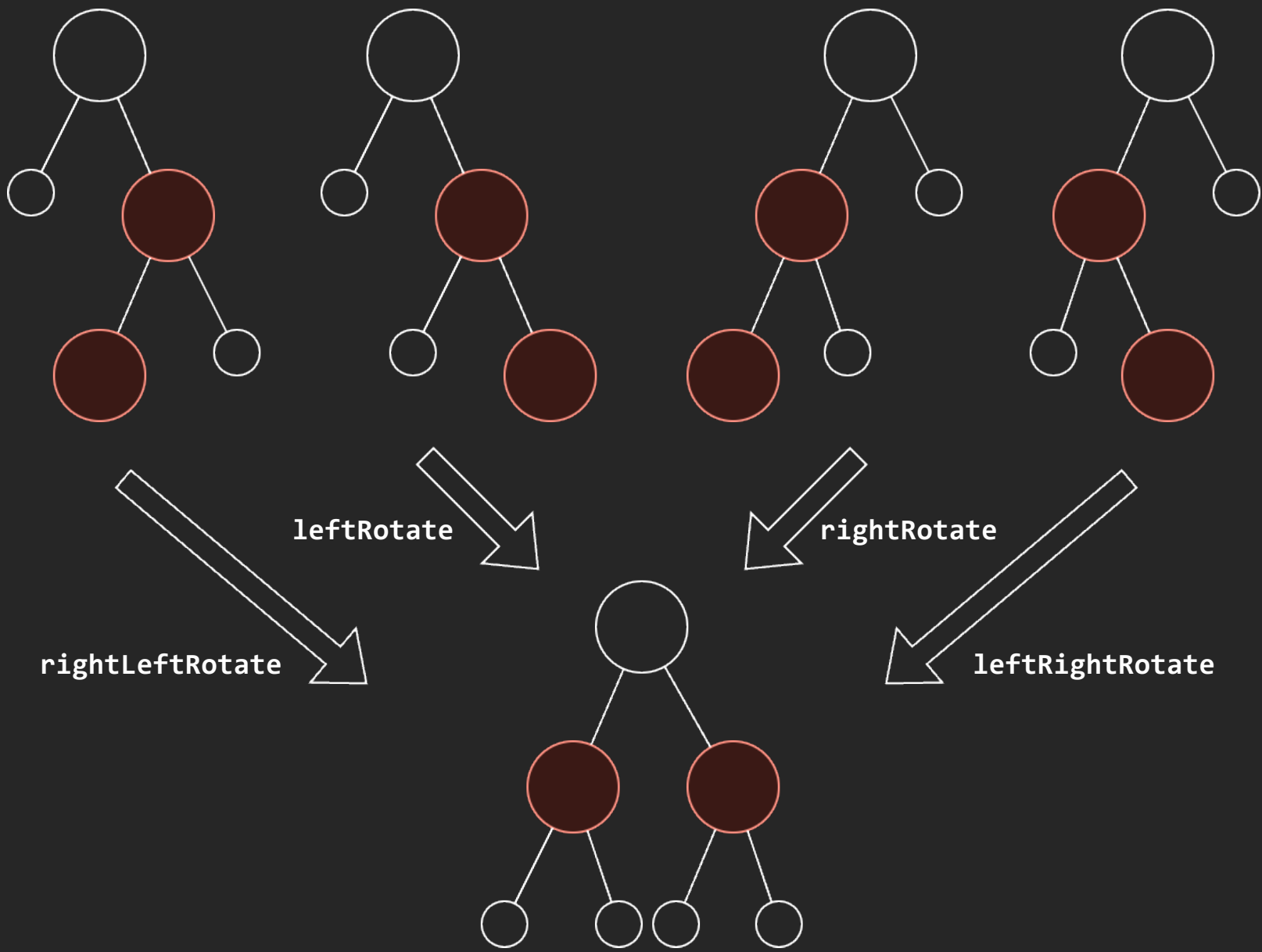


`rightLeftRotate(B) + recolor(B,C)`

# Вставка в КЧД. 3-вершина



`rightLeftRotate(B) + recolor(B,C)`



# Вставка в КЧД. 2- и 3-вершина

---

Итак...

- помимо базовых поворотов, имеем операцию перекраски вершины `recolor(Node* v)`
- 2 случая вставки ключа в 2-вершину
- 4+2 случая вставки ключа в 3-вершину

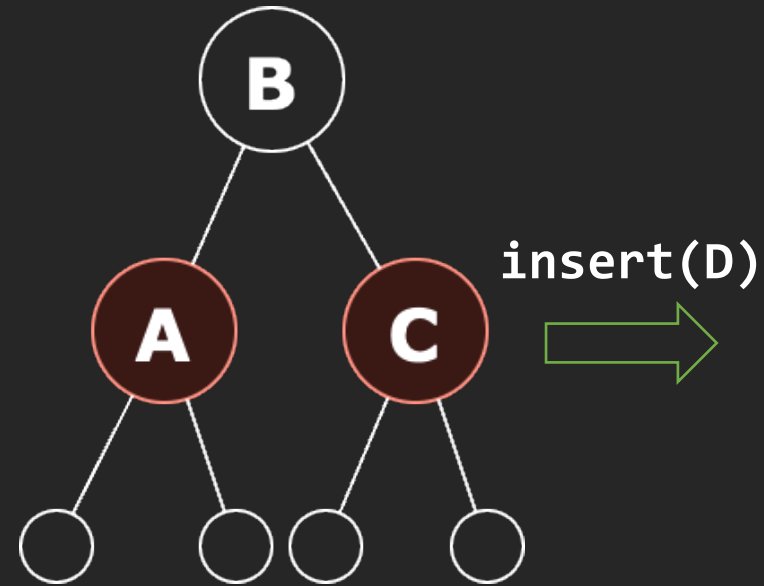
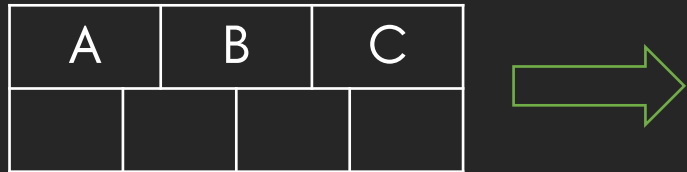
# Вставка ключей в 4-вершину без предка

---

# Вставка в КЧД. 4-вершина

PARENTS NOT  
ALLOWED!

insert(D)

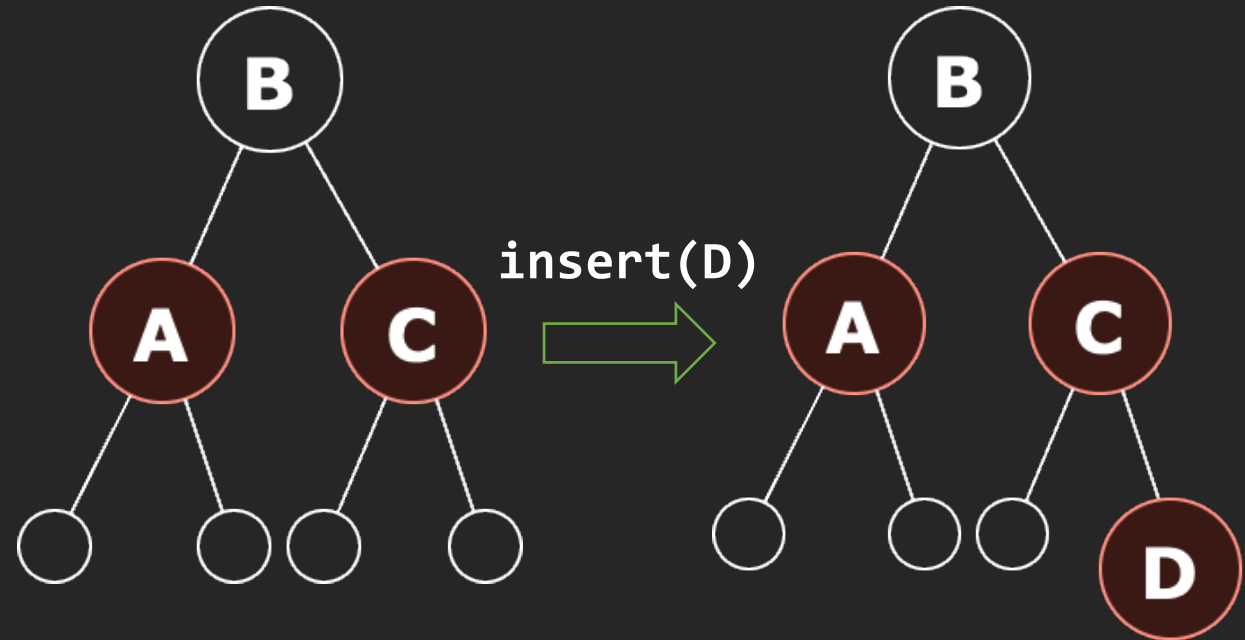
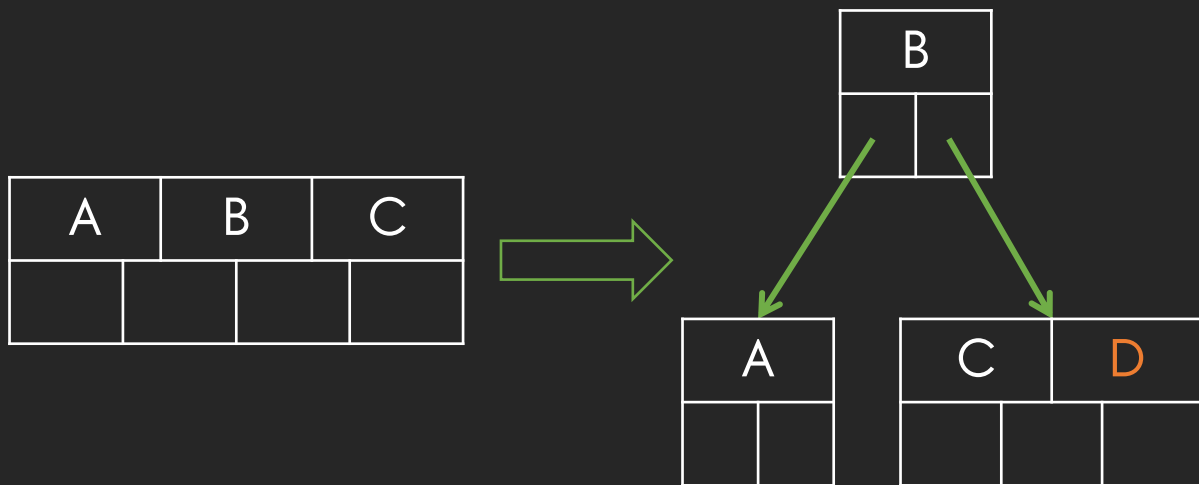




# Вставка в КЧД. 4-вершина

PARENTS NOT  
ALLOWED!

insert(D)

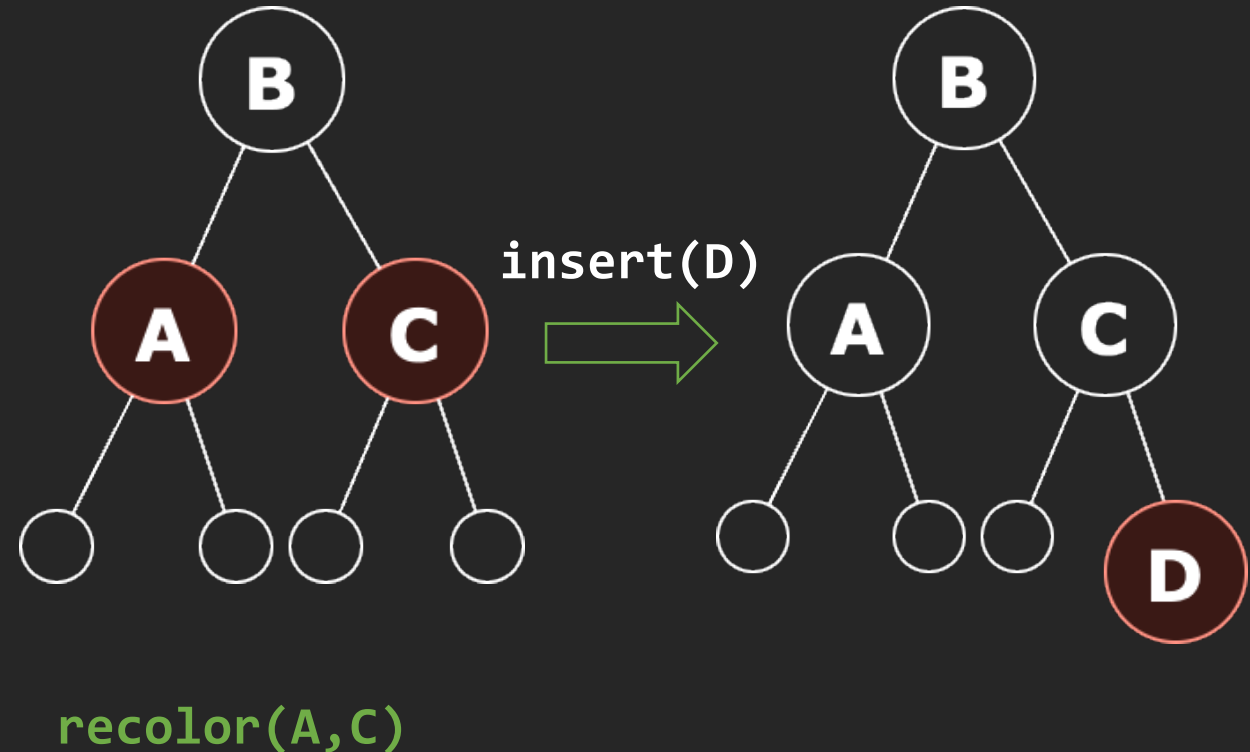
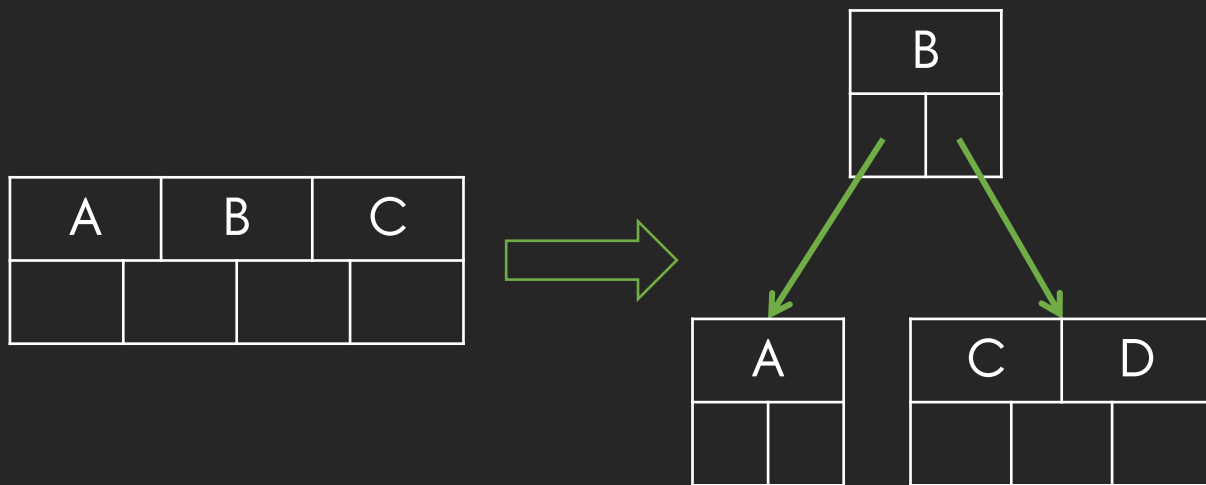


recolor(A,C)

# Вставка в КЧД. 4-вершина

PARENTS NOT  
ALLOWED!

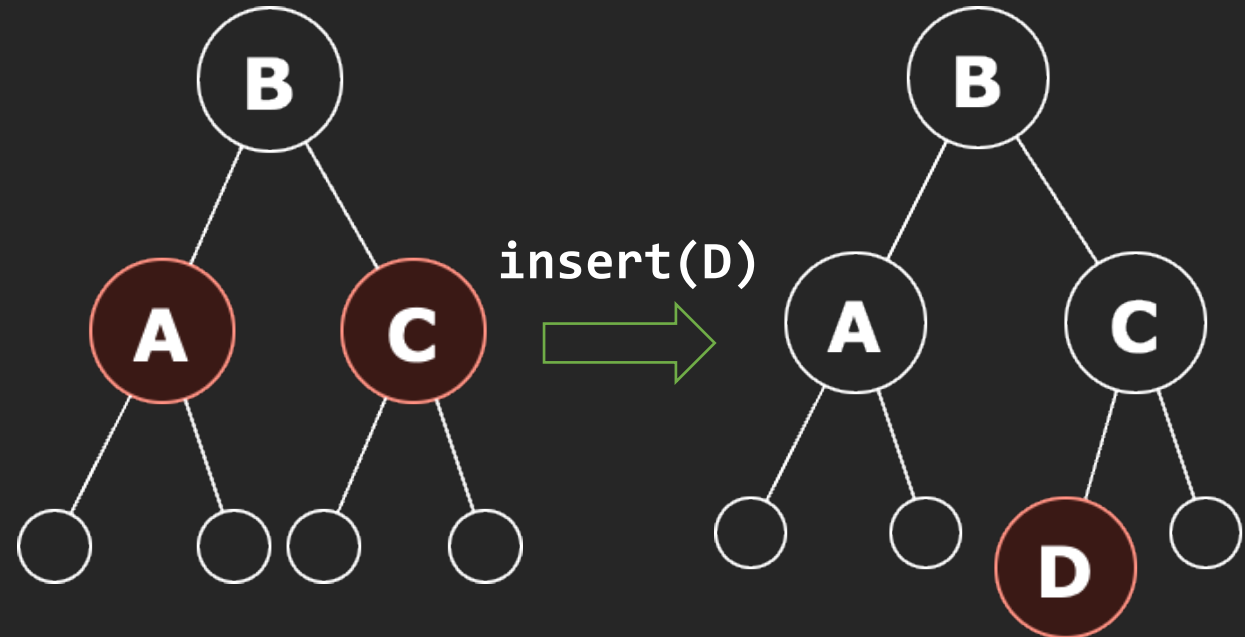
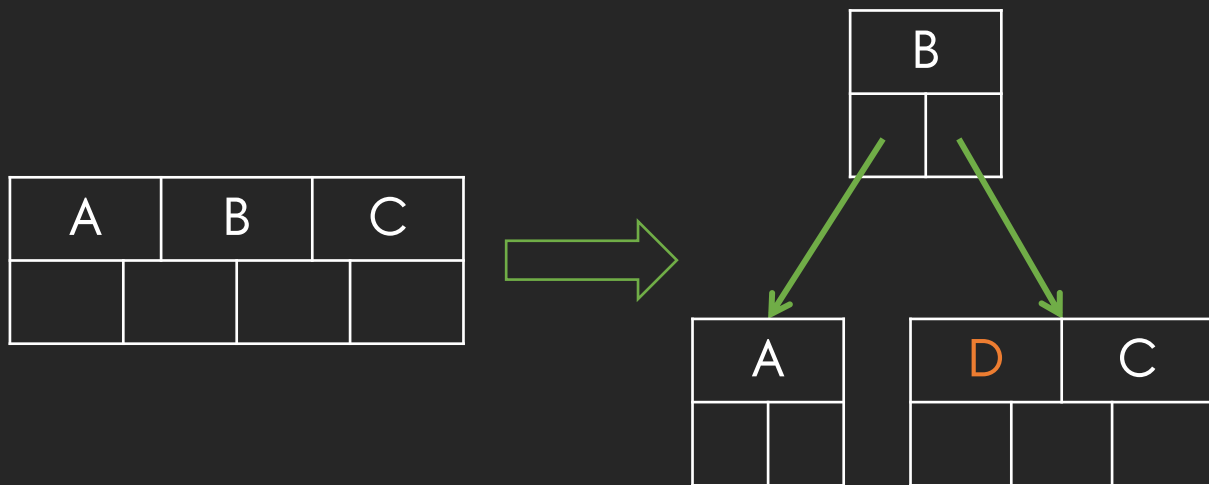
insert(D)



# Вставка в КЧД. 4-вершина

PARENTS NOT  
ALLOWED!

insert(D)

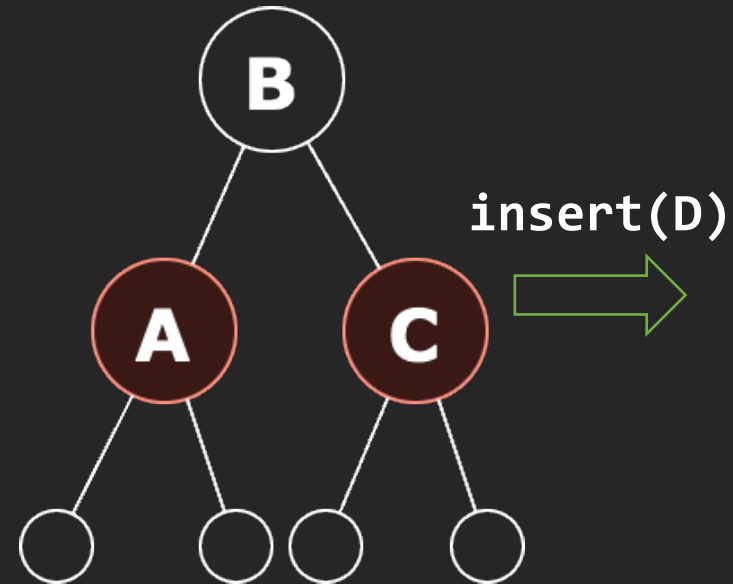
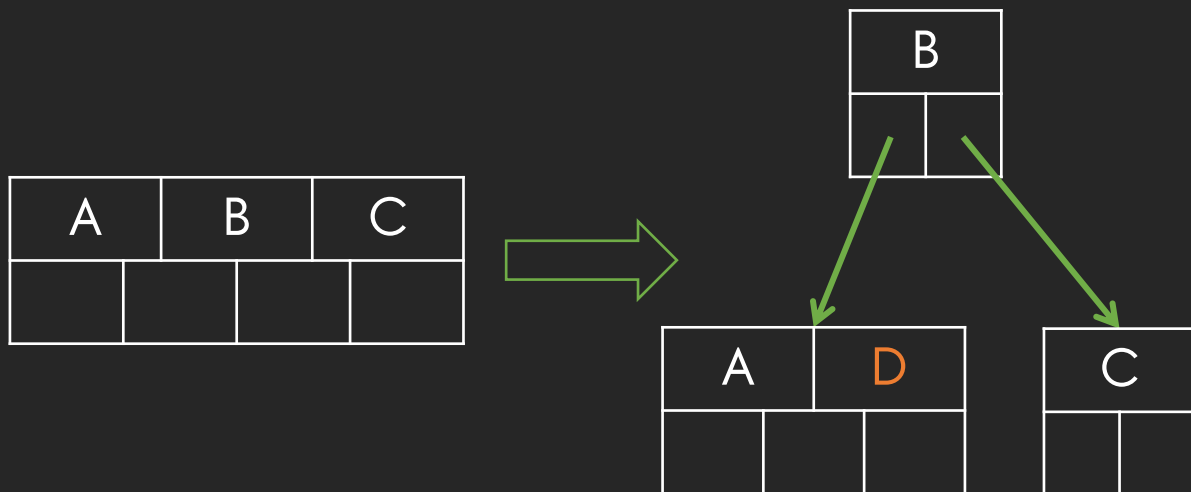


recolor(A,C)

# Вставка в КЧД. 4-вершина

PARENTS NOT  
ALLOWED!

insert(D)

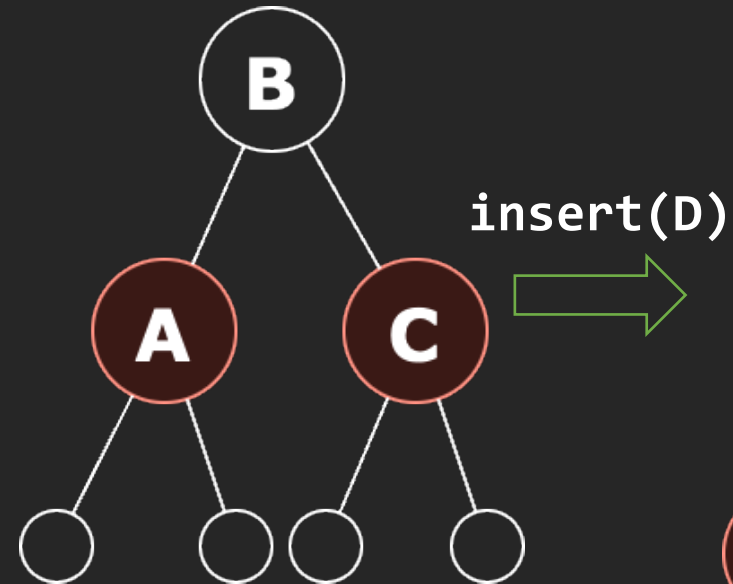
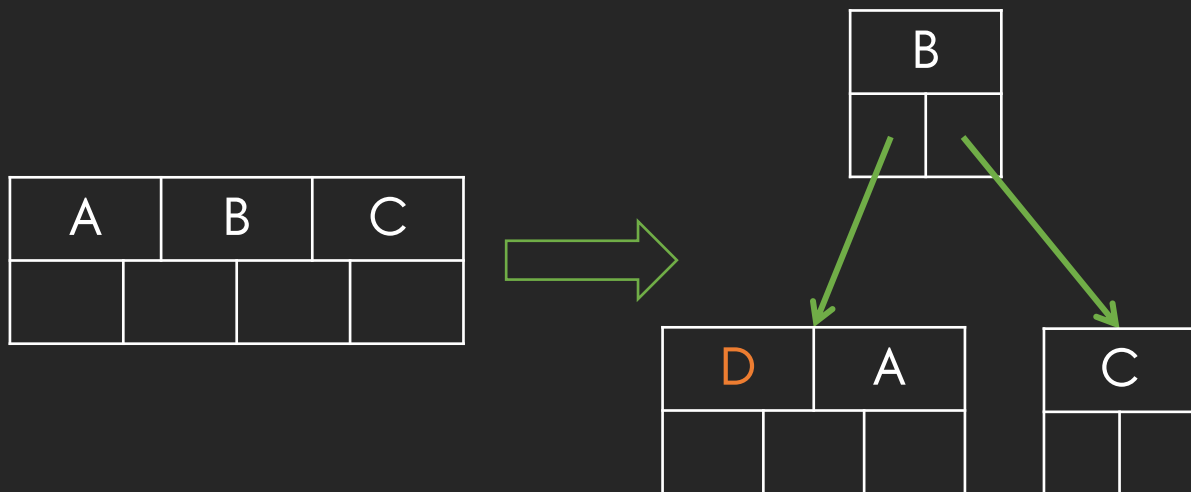


recolor(A,C)

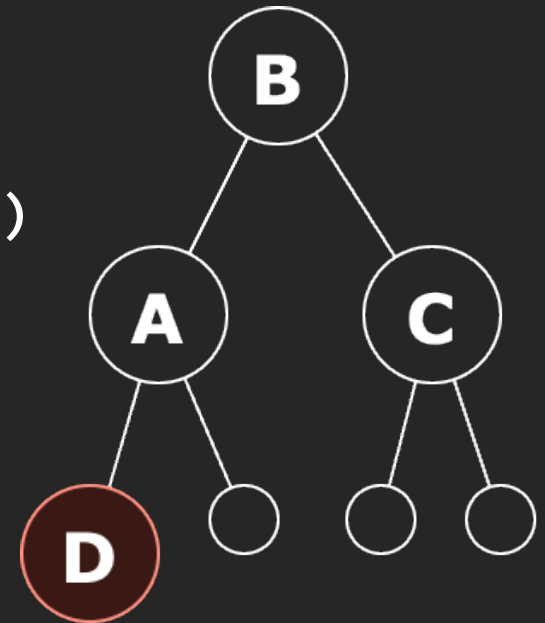
# Вставка в КЧД. 4-вершина

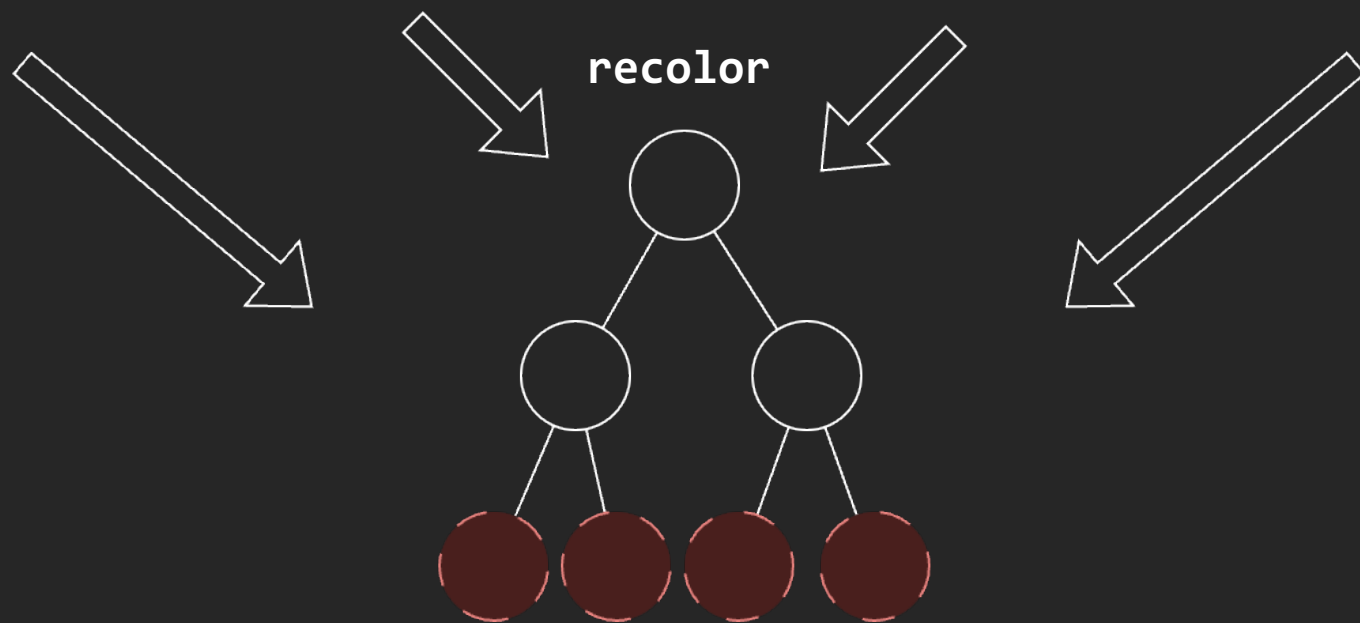
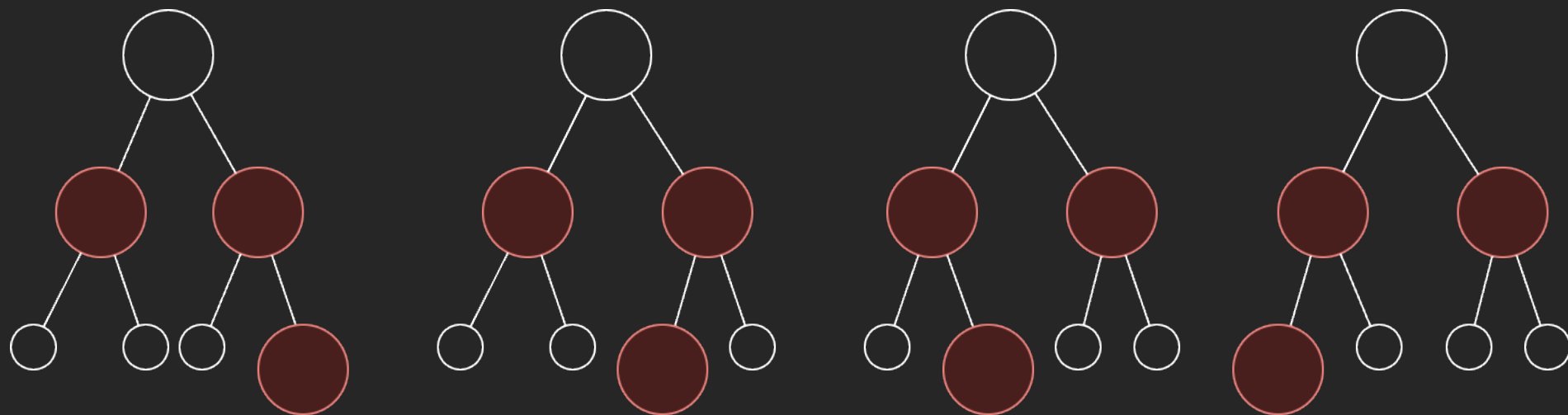
PARENTS NOT  
ALLOWED!

insert(D)



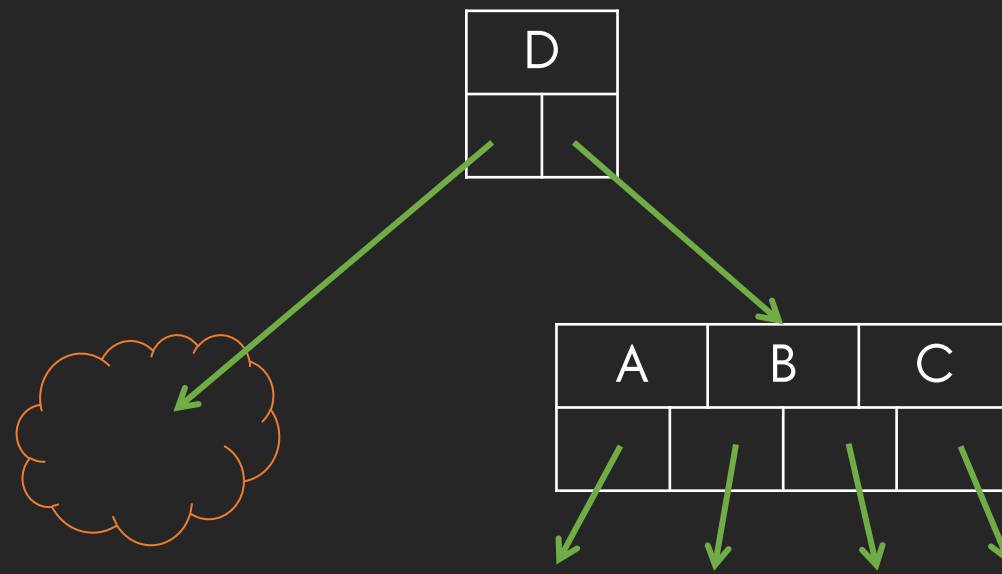
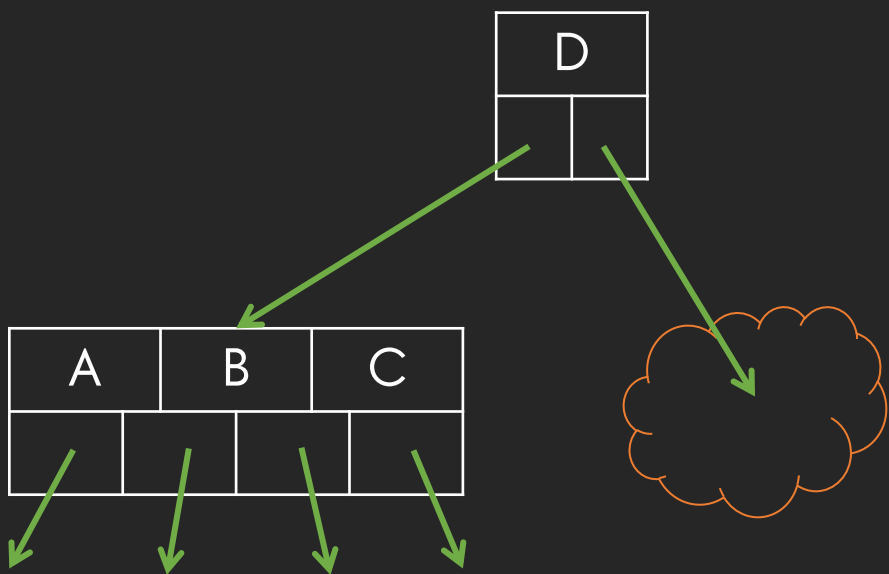
recolor(A,C)





# Вставка ключей в 4-вершину с предком – 2-вершиной

---

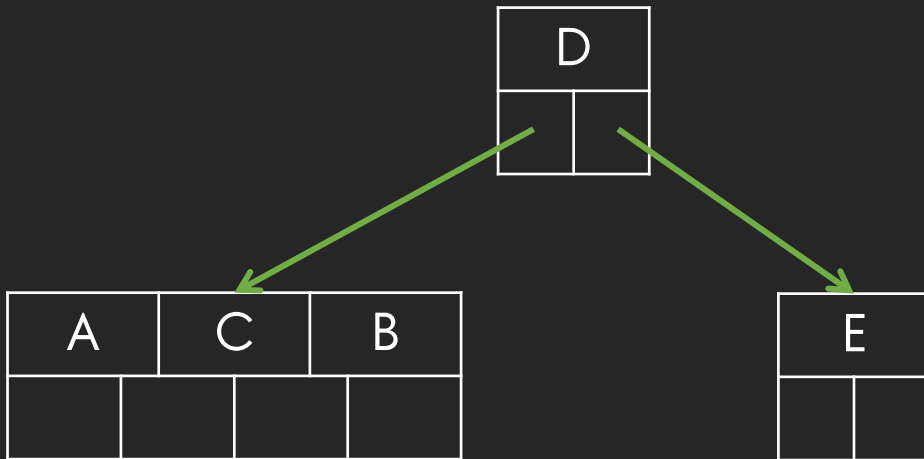


Всего **два способа** расположения 4-вершины  
среди потомков 2-вершины!



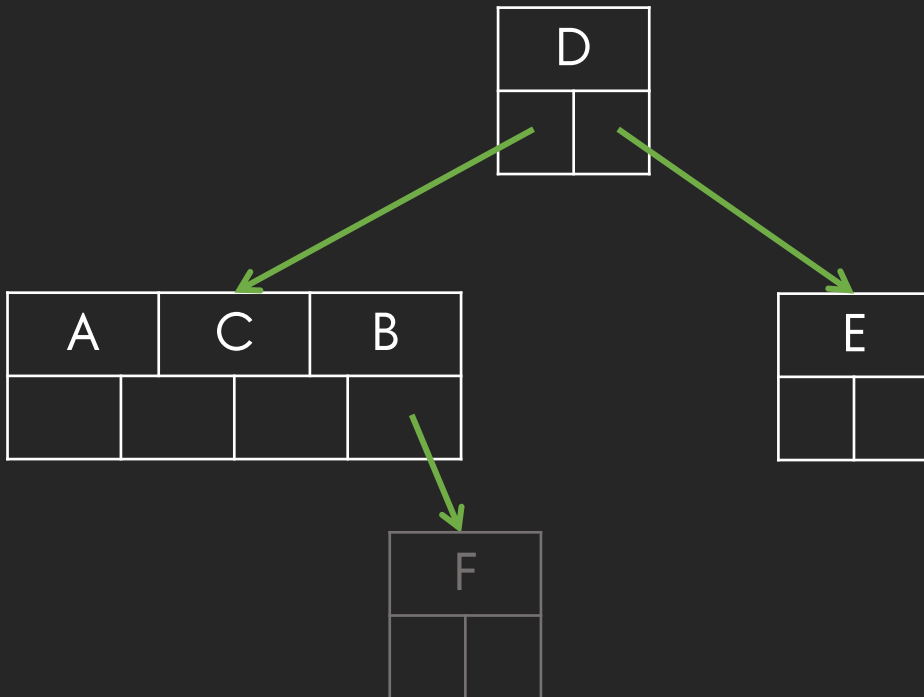
# Вставка в КЧД. 4-вершина

insert(F)  $F > B$



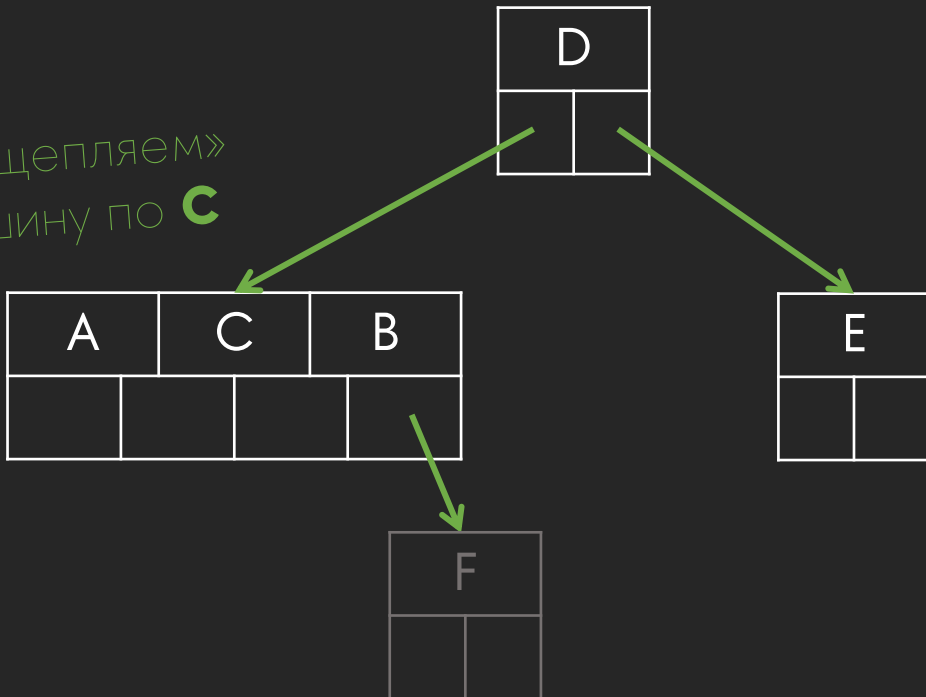
# Вставка в КЧД. 4-вершина

insert(F)  $F > B$



# Вставка в КЧД. 4-вершина

«Расщепляем»  
вершину по **C**

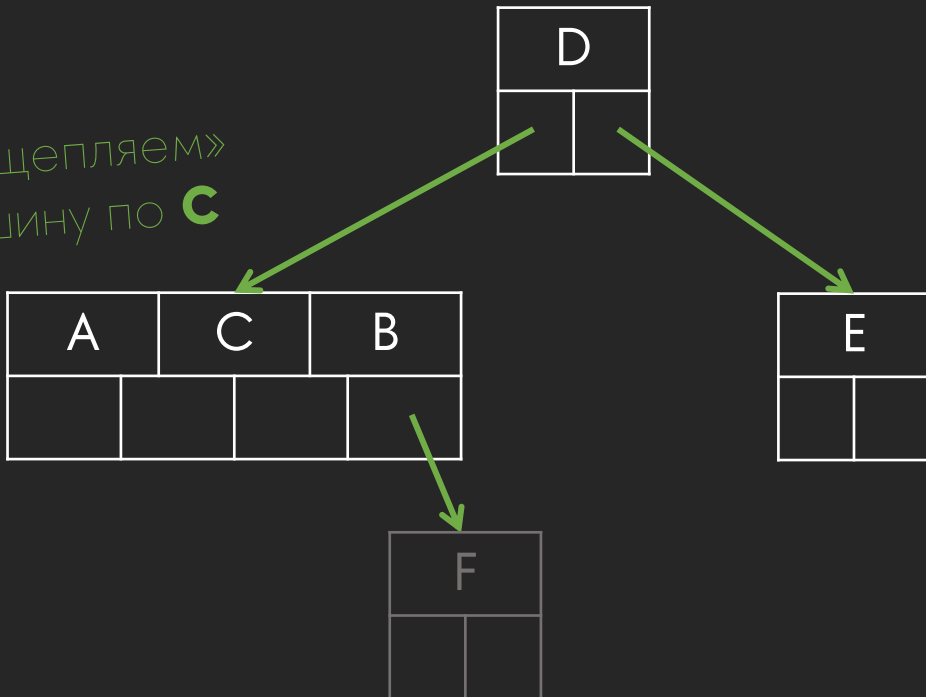


`insert(F)`  $F > B$

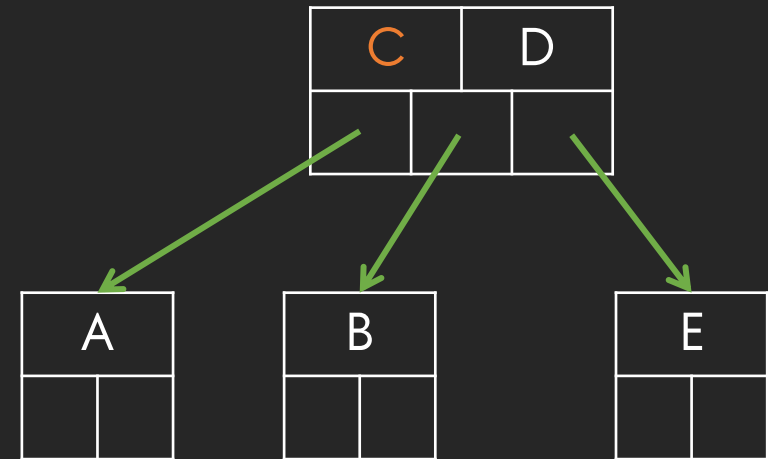


# Вставка в КЧД. 4-вершина

«Расщепляем»  
вершину по **C**

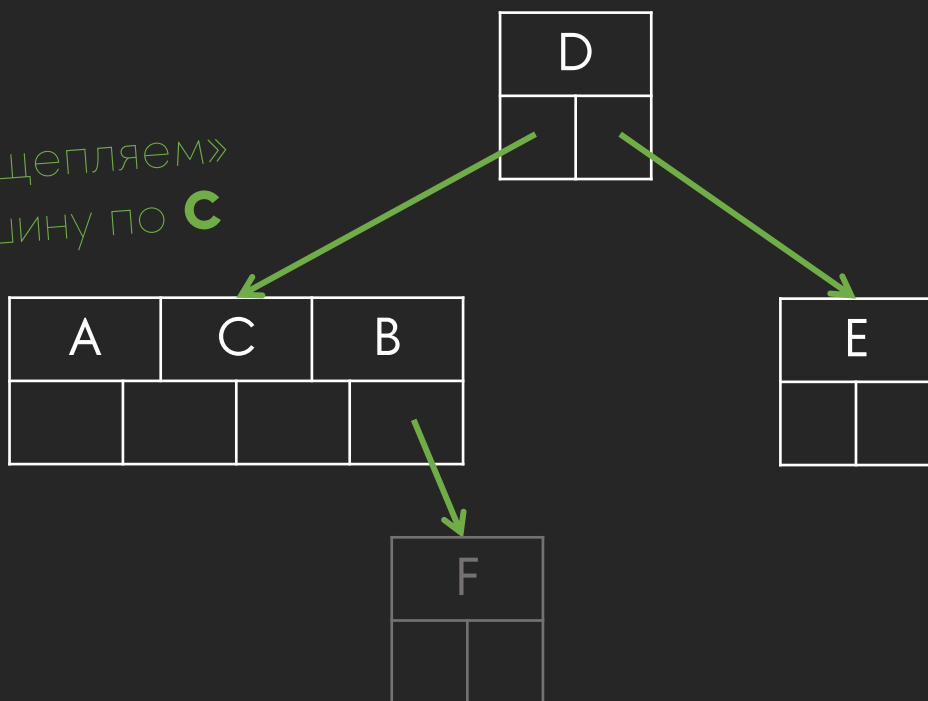


`insert(F)`  $F > B$

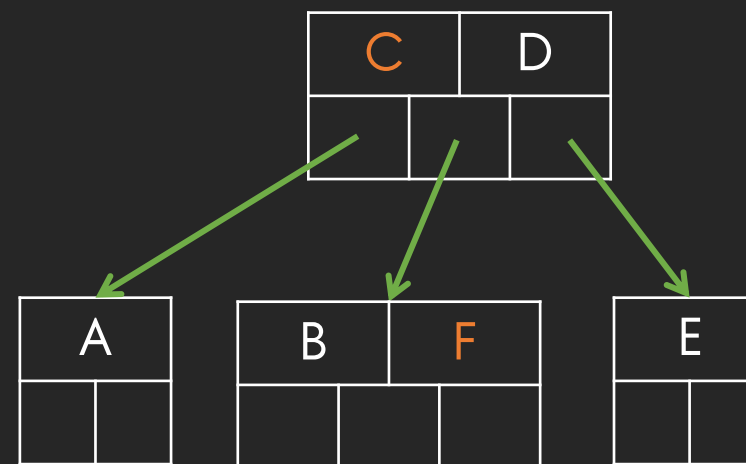


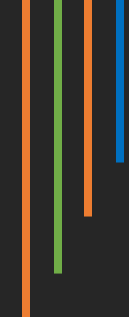
# Вставка в КЧД. 4-вершина

«Расщепляем»  
вершину по **C**



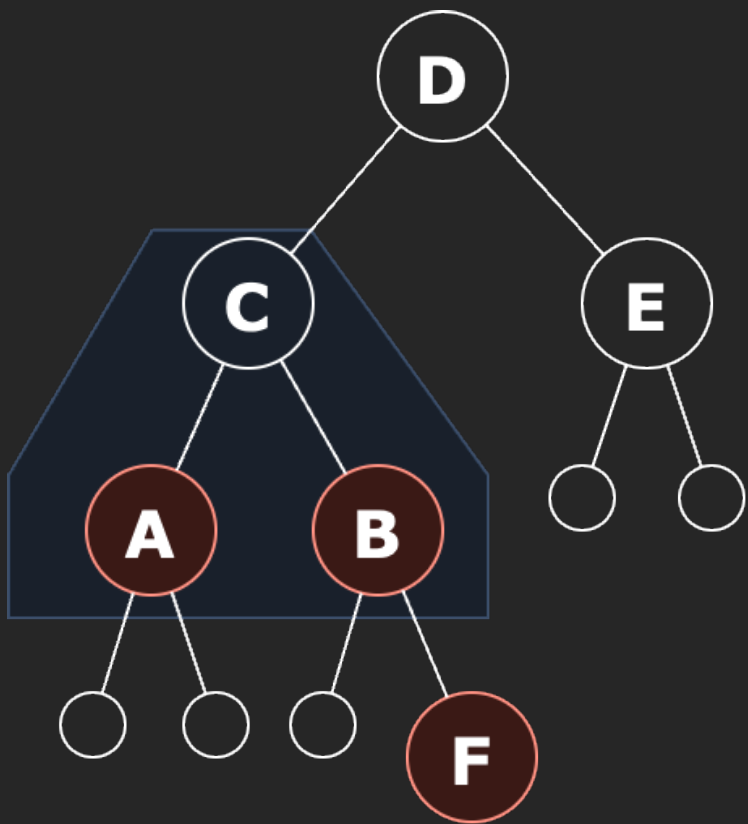
insert(F)  $F > B$



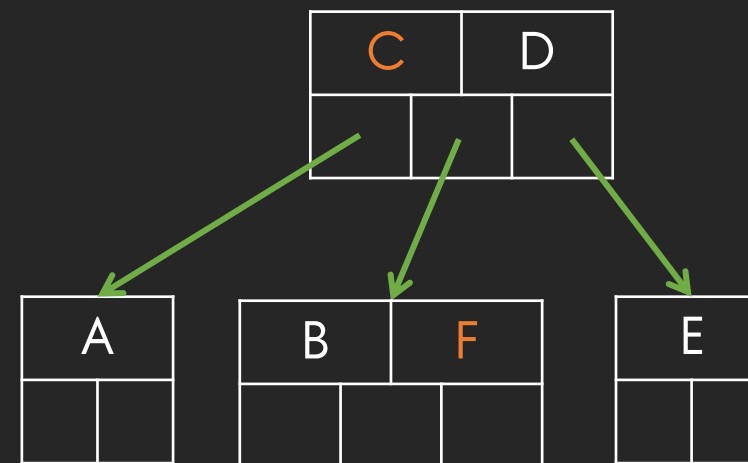


# Перейдем к КЧД-эквиваленту

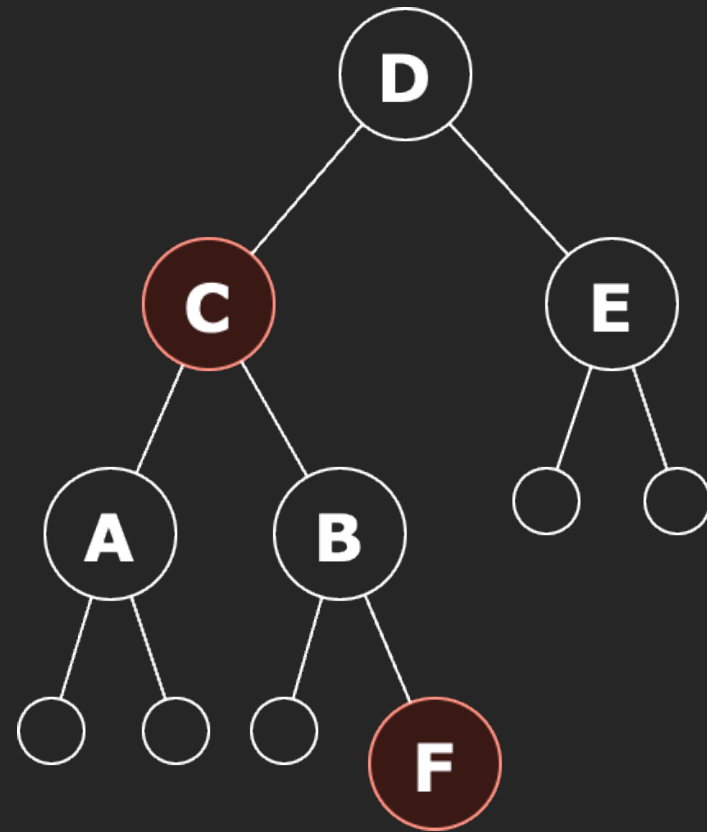
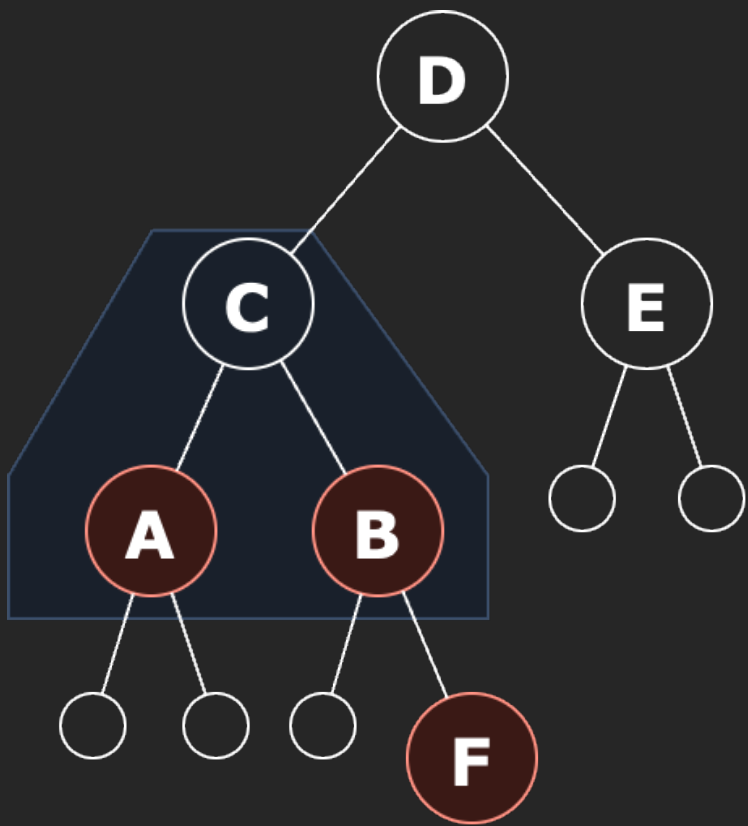
# Вставка в КЧД. 4-вершина



insert(F)  $F > B$

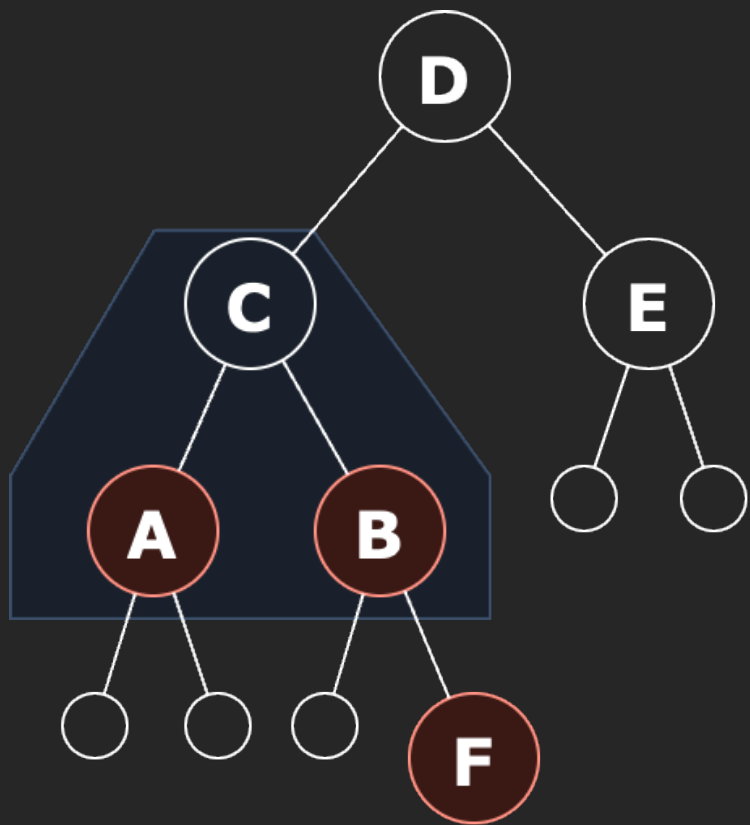


# Вставка в КЧД. 4-вершина

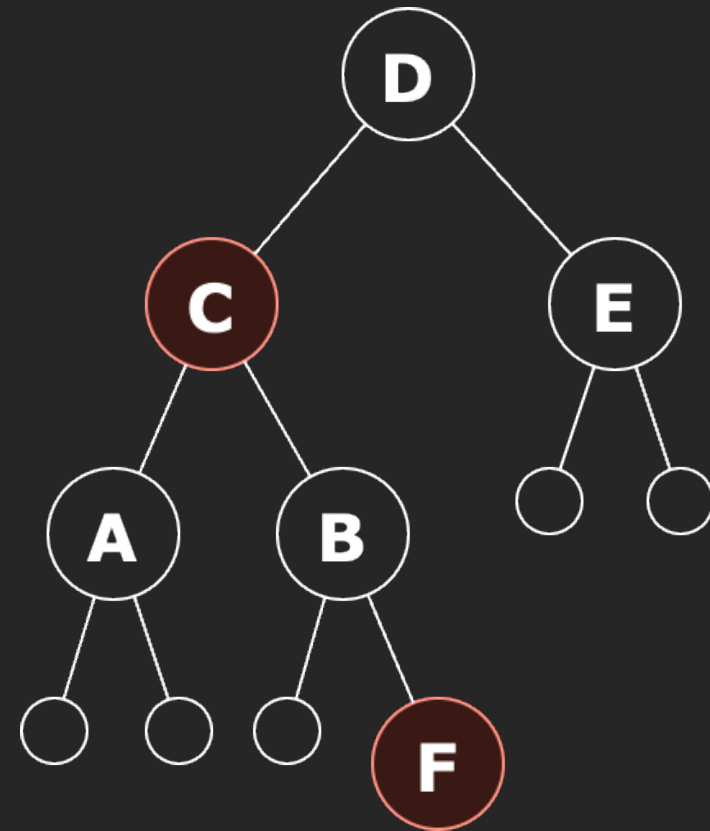
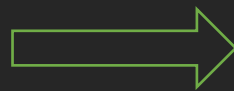




# Вставка в КЧД. 4-вершина



`recolor(A,B,C)`

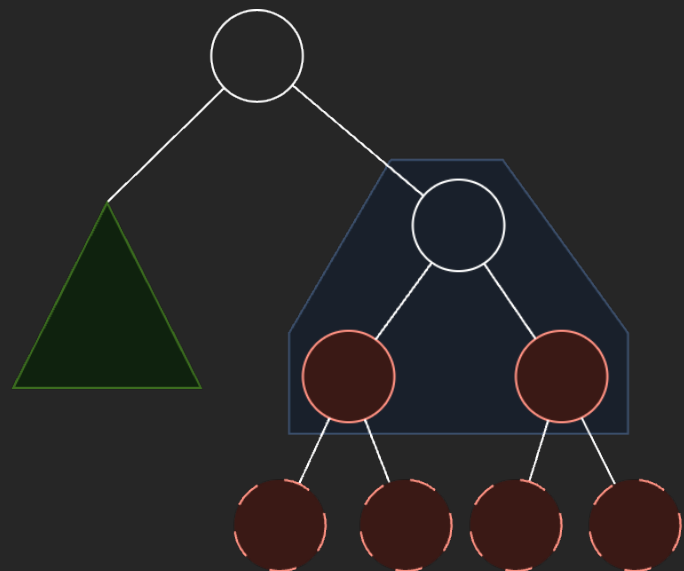


# Вставка в КЧД. 4-вершина

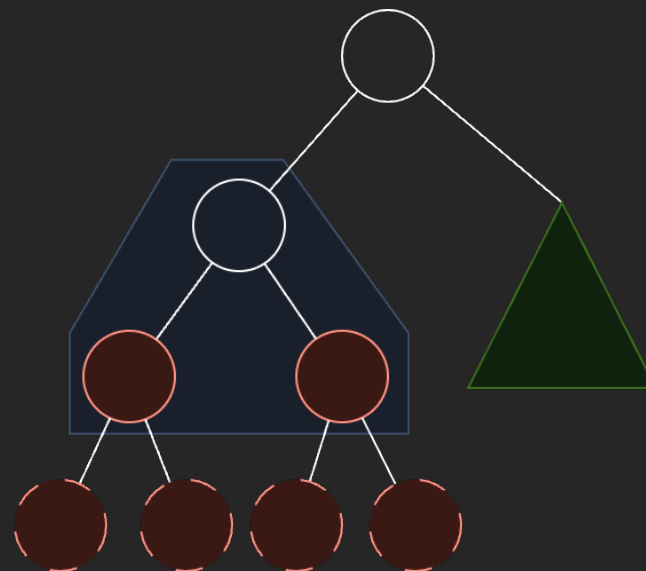
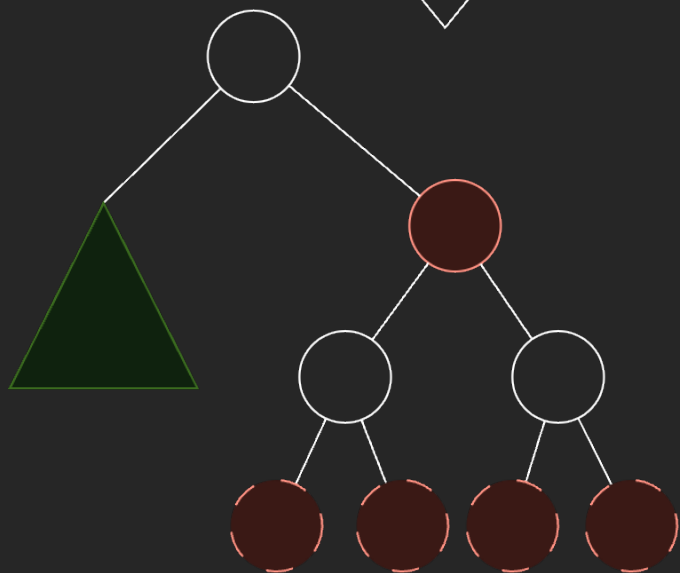
---

Вставка в 4-вершину, у которой предком является 2-вершина требует

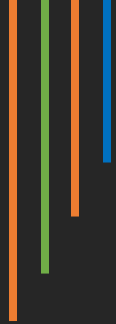
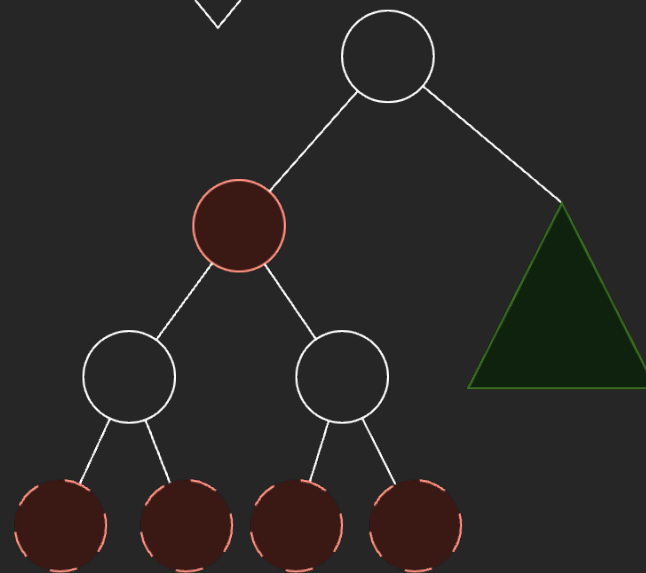
- перекраски родителя
- перекраски родственника родителя
- перекраски прародителя



recolor  
↓

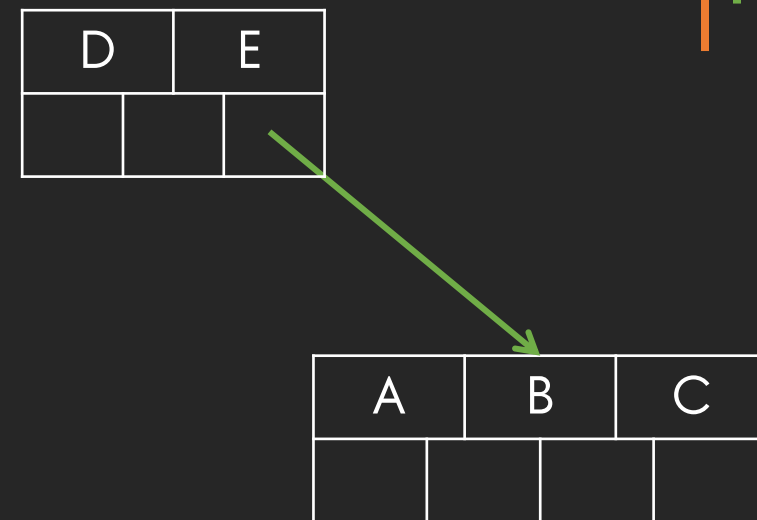
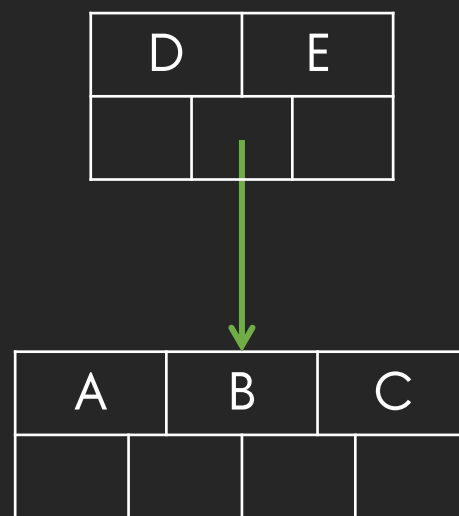
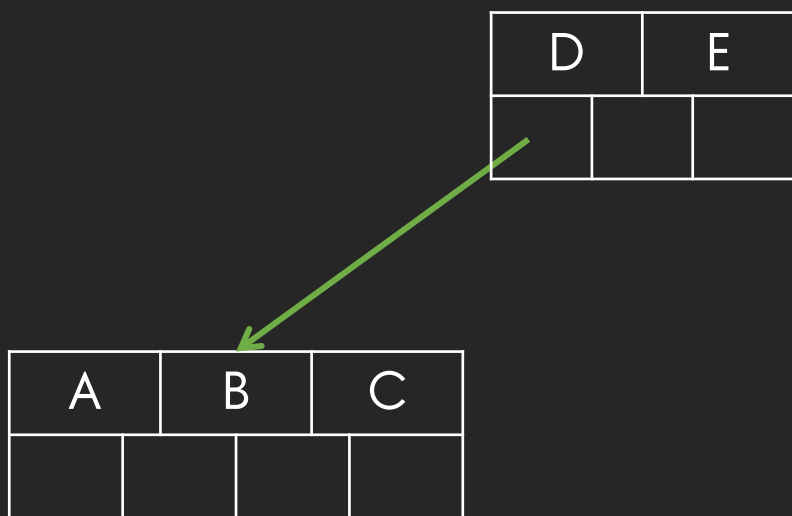


↓  
recolor



# Вставка ключей в 4-вершину с предком – 3-вершиной

---

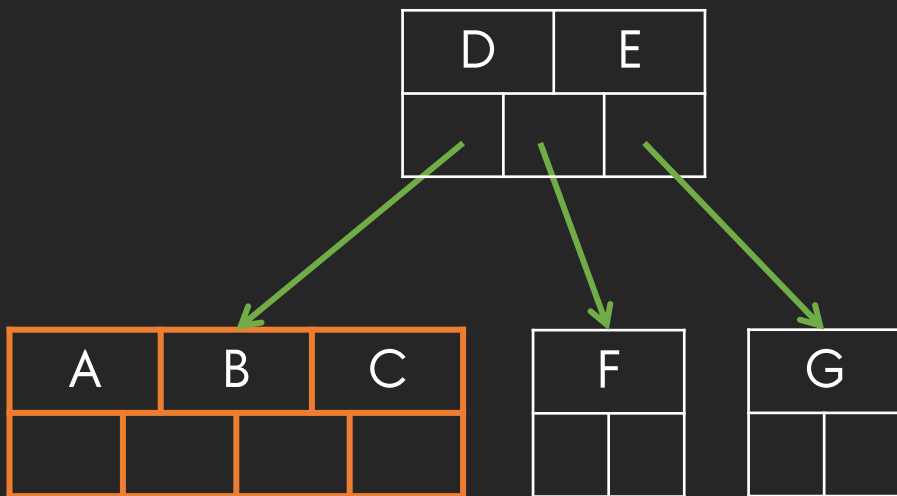


Всего **три способа** расположения 4-вершины  
среди потомков 3-вершины!

# Вставка в КЧД. 4-вершина

4-вершина – левый  
потомок 3-вершины

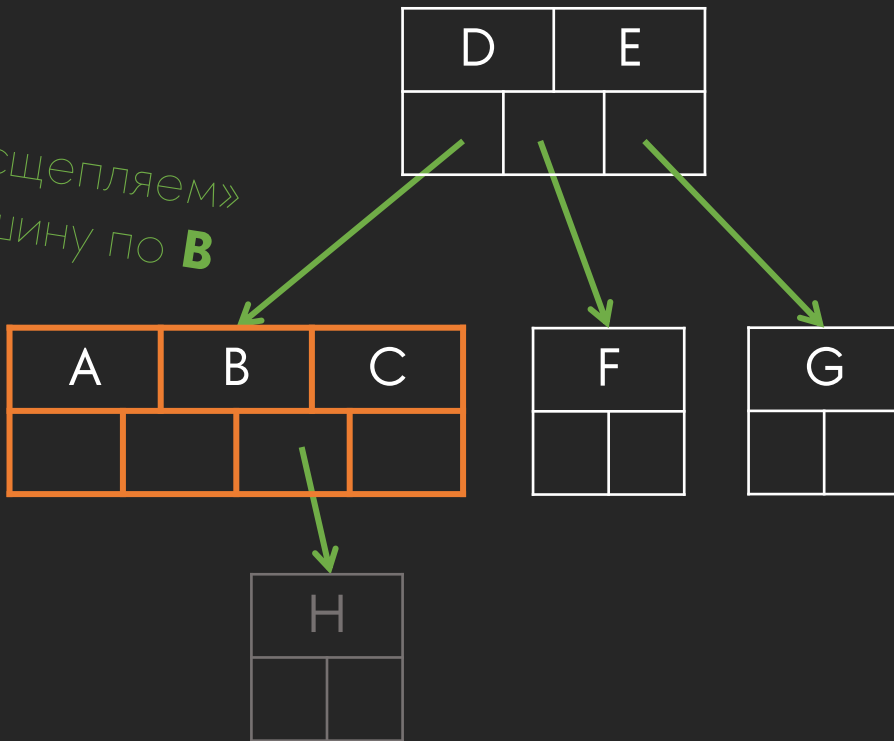
`insert(H)`     $H > B$  и  $H < C$



# Вставка в КЧД. 4-вершина

4-вершина – левый  
потомок 3-вершины

«Расцепляем»  
вершину по **B**

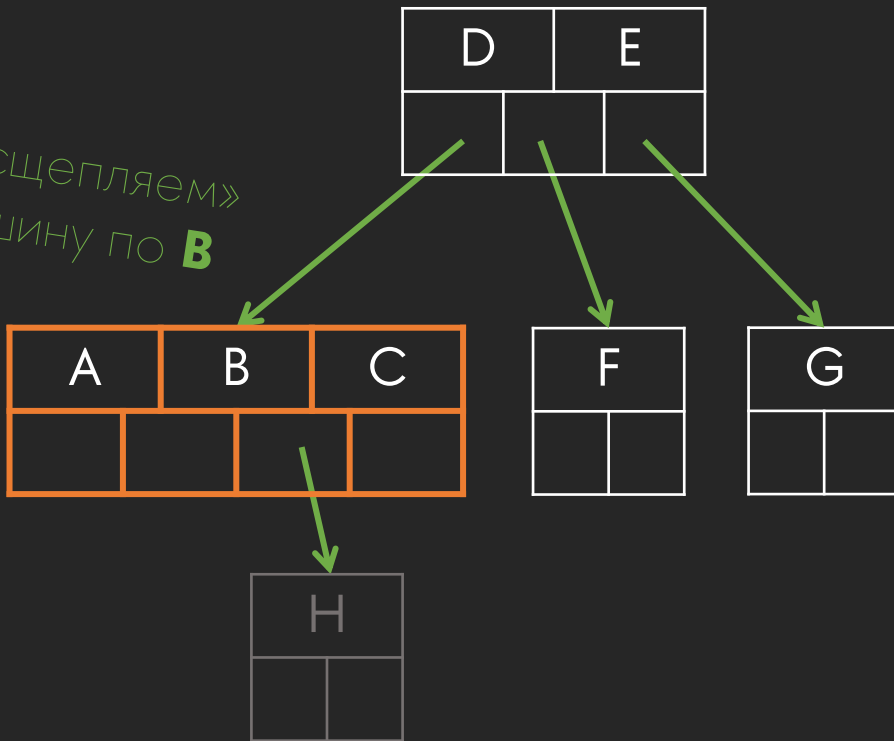


`insert(H)`     $H > B$  и  $H < C$

# Вставка в КЧД. 4-вершина

4-вершина – левый  
потомок 2-вершины

«Расцепляем»  
вершину по **B**



insert(H)

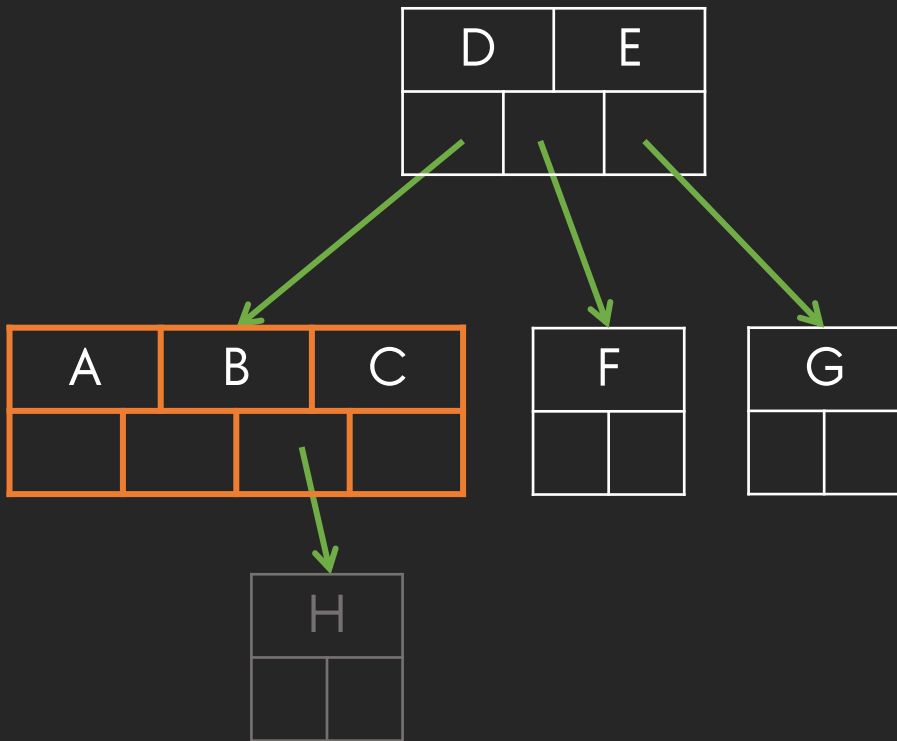
$H > B$  и  $H < C$



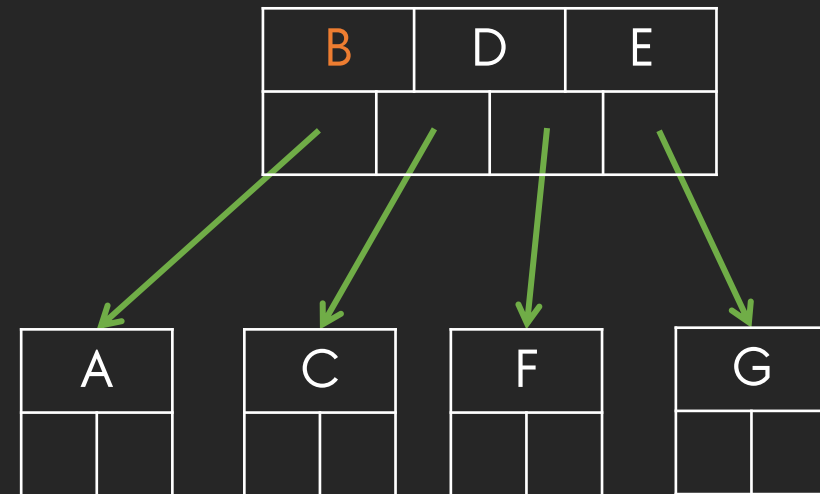


# Вставка в КЧД. 4-вершина

4-вершина – левый  
потомок 2-вершины

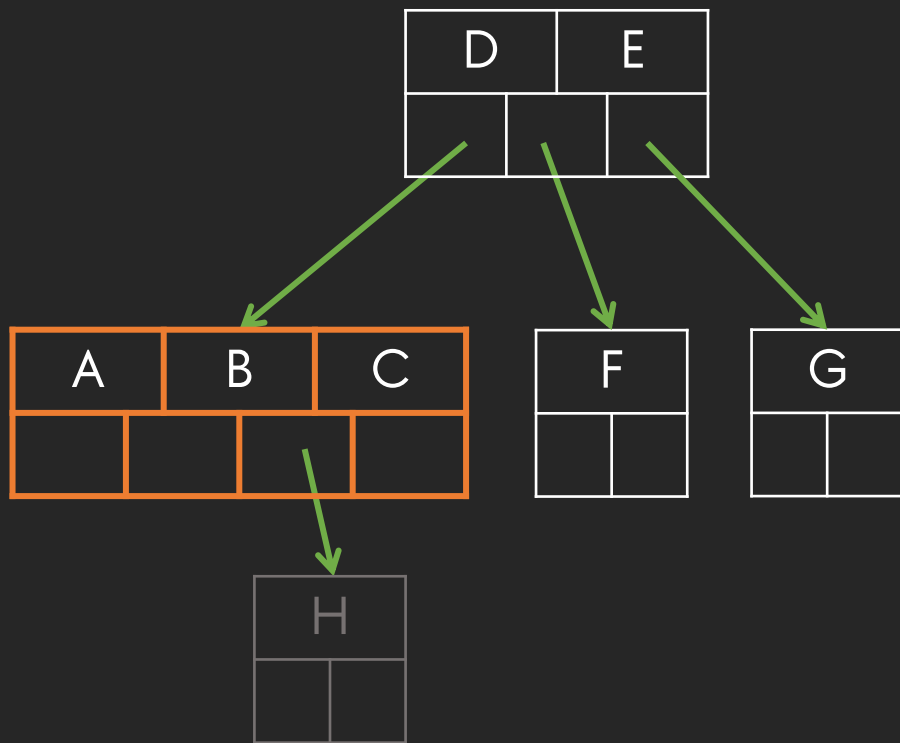


insert(H)  $H > B$  и  $H < C$

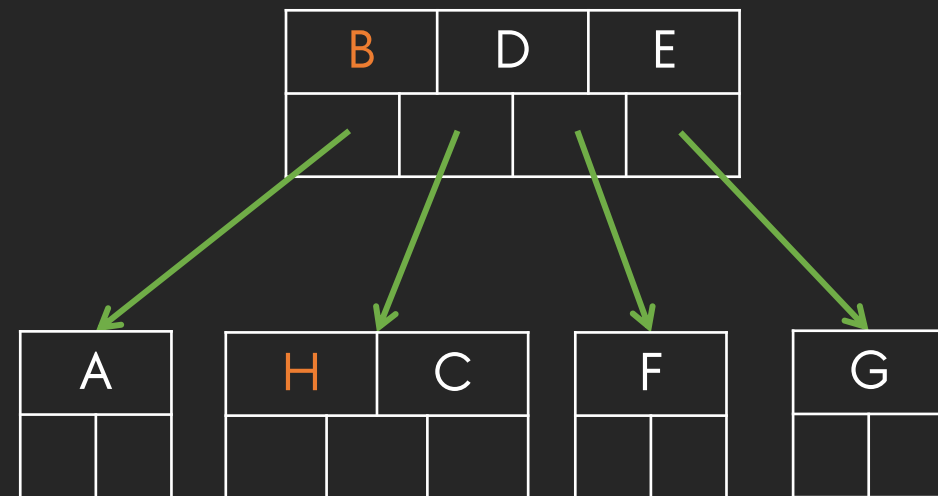


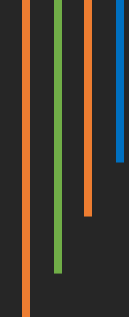
# Вставка в КЧД. 4-вершина

4-вершина – левый  
потомок 2-вершины



insert(H)  $H > B$  и  $H < C$

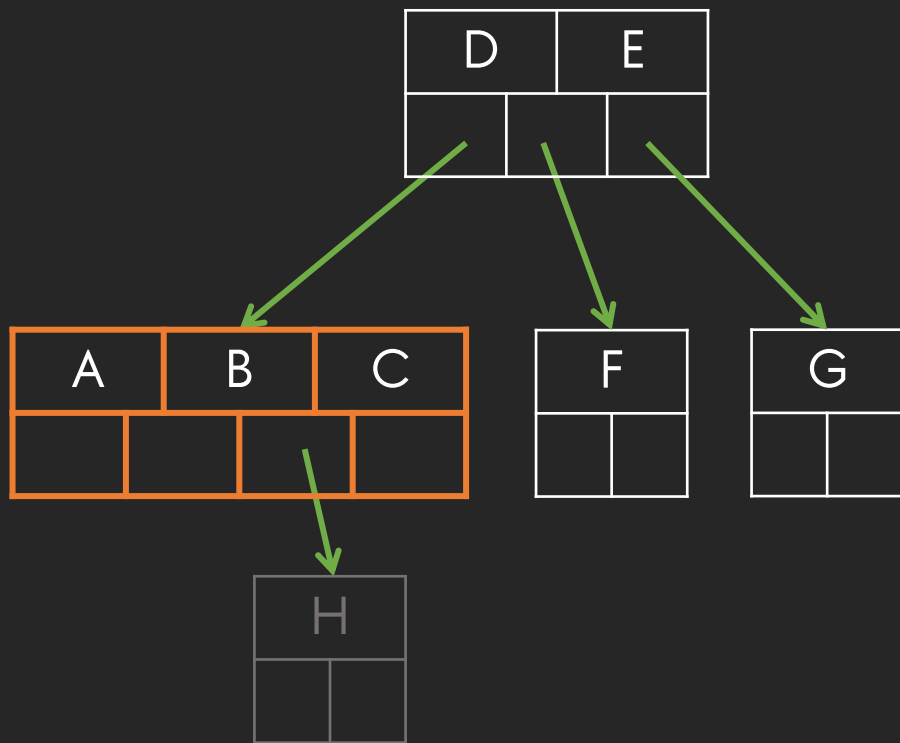




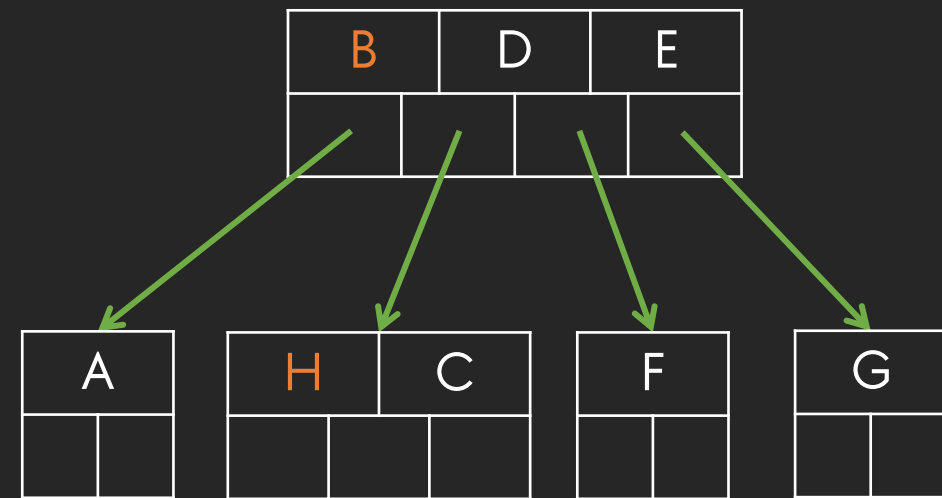
2-вершину в КЧД можно представить  
разными способами, один из которых  
мало интересен...

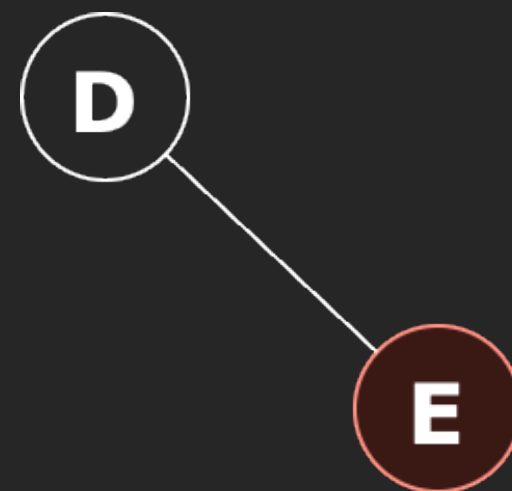
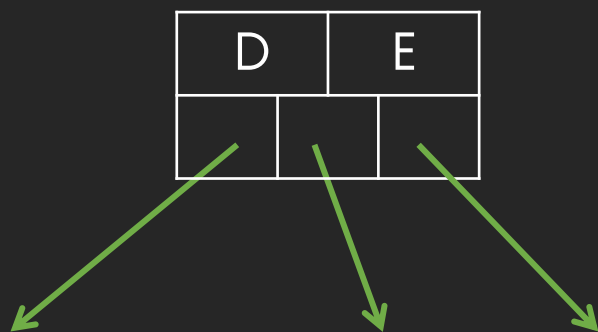
# Вставка в КЧД. 4-вершина

4-вершина – левый  
потомок 2-вершины



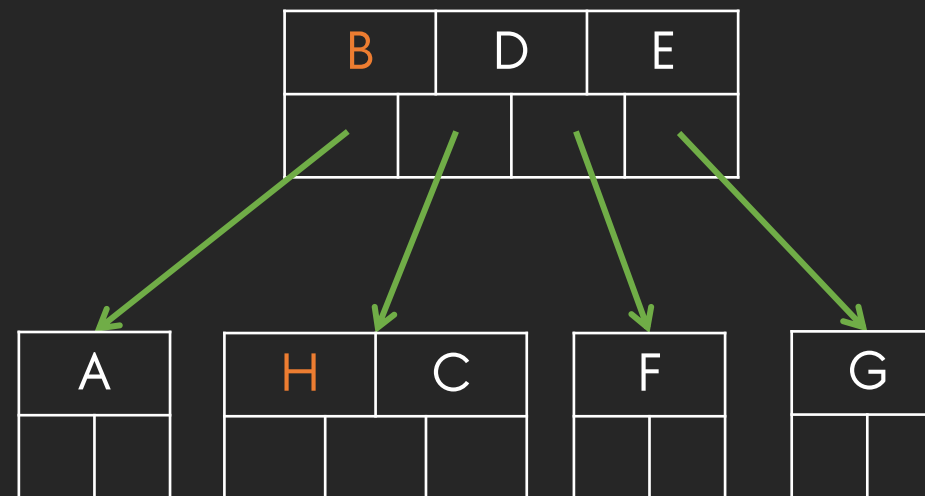
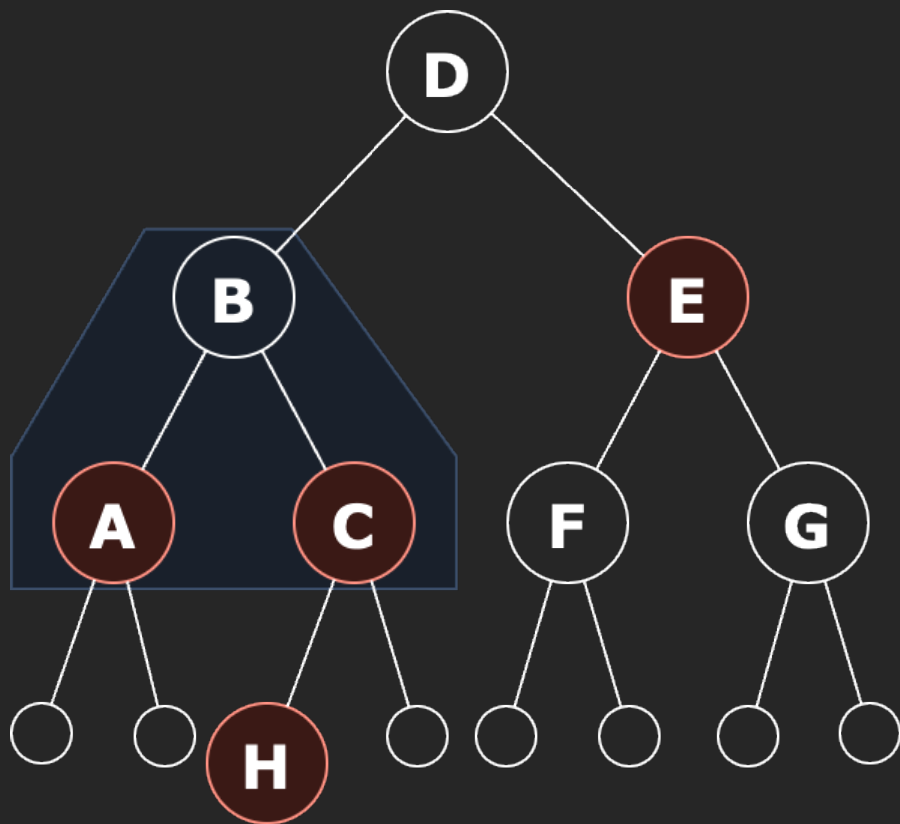
insert(H)  $H > B$  и  $H < C$





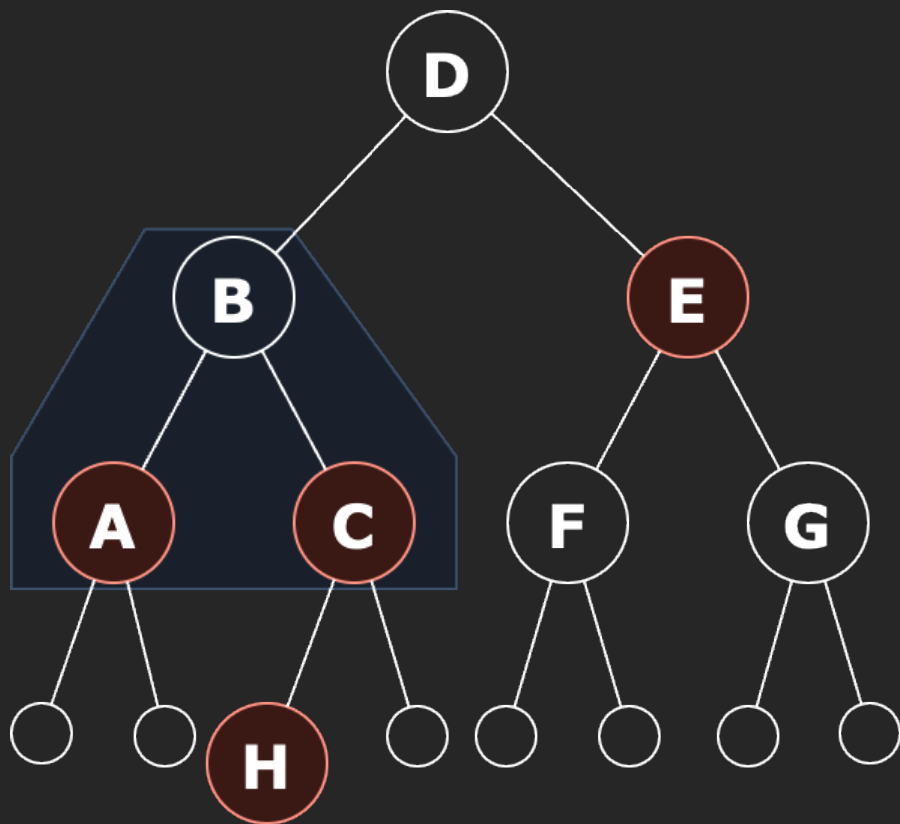
# Вставка в КЧД. 4-вершина

4-вершина – левый  
потомок 3-вершины

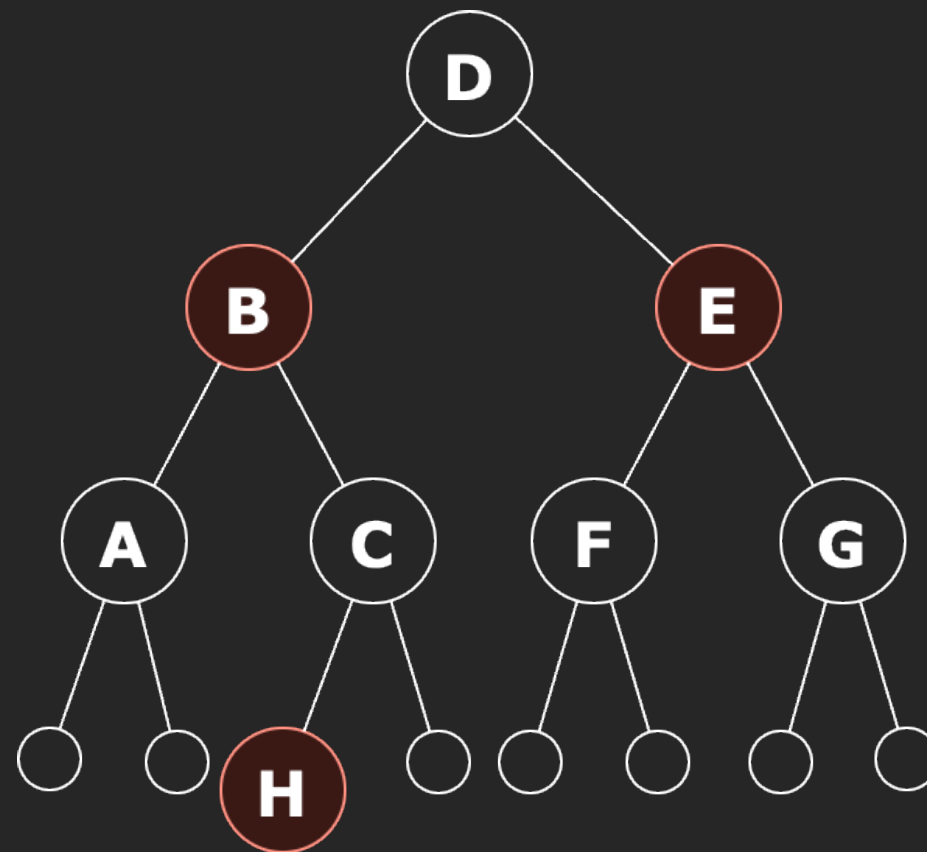
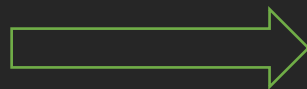


# Вставка в КЧД. 4-вершина

4-вершина – левый  
потомок 3-вершины

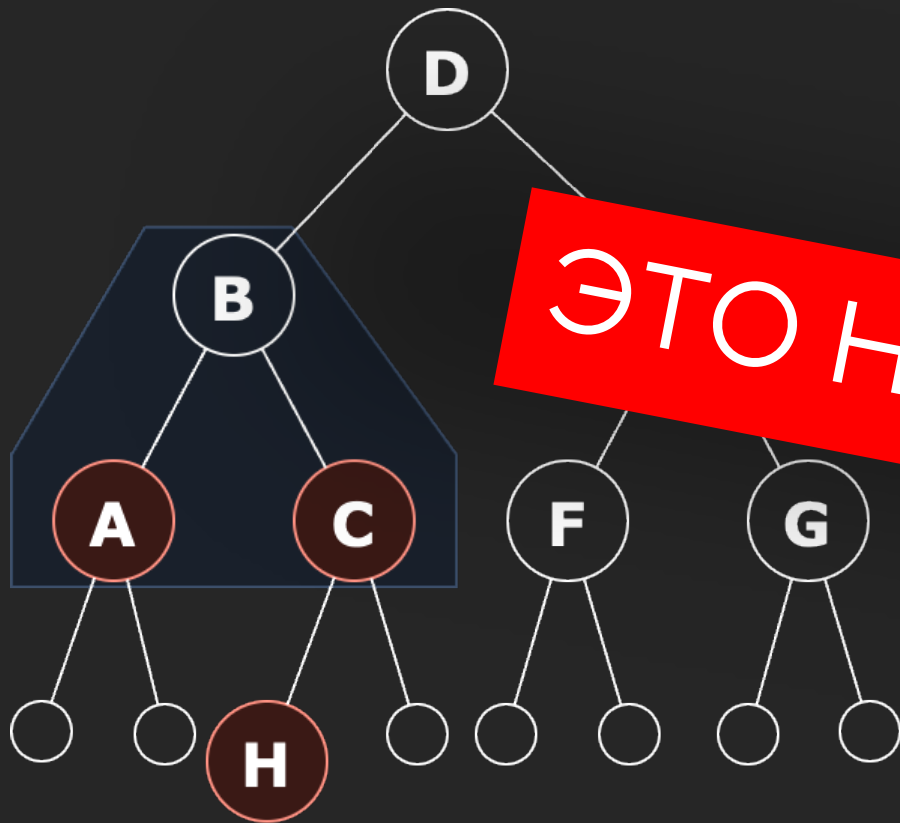


recolor(A,B,C)

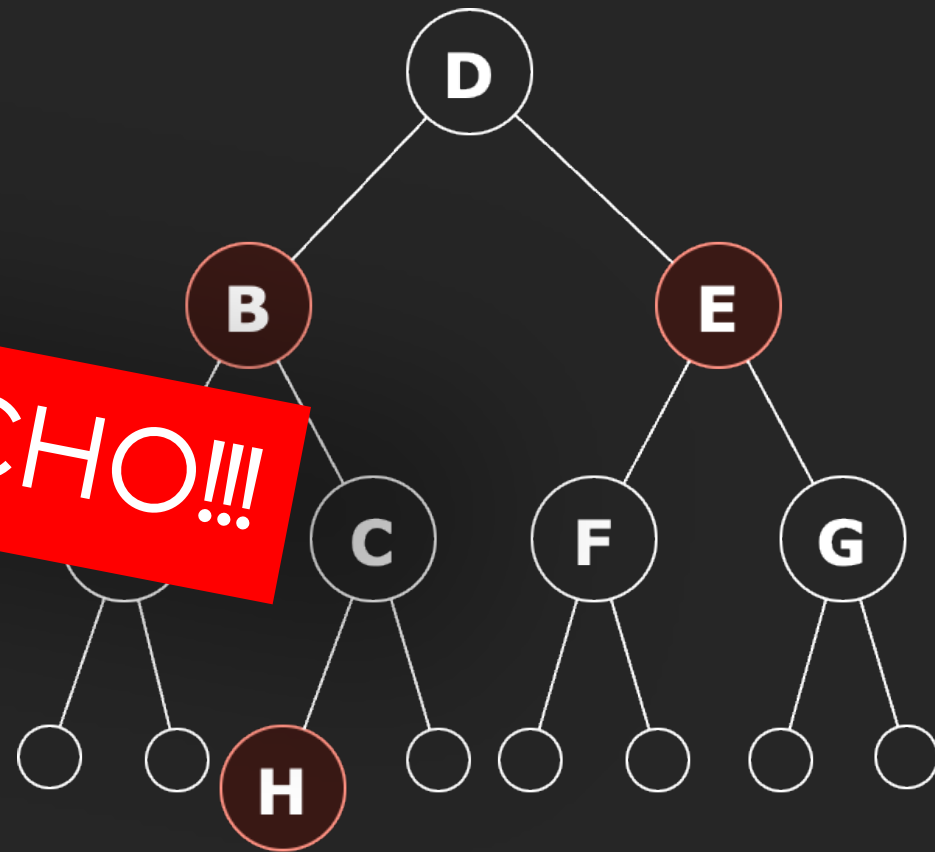


# Вставка в КЧД. 4-вершина

4-вершина – левый  
потомок 3-вершины



ЭТО НЕИНТЕРЕСНО!!!



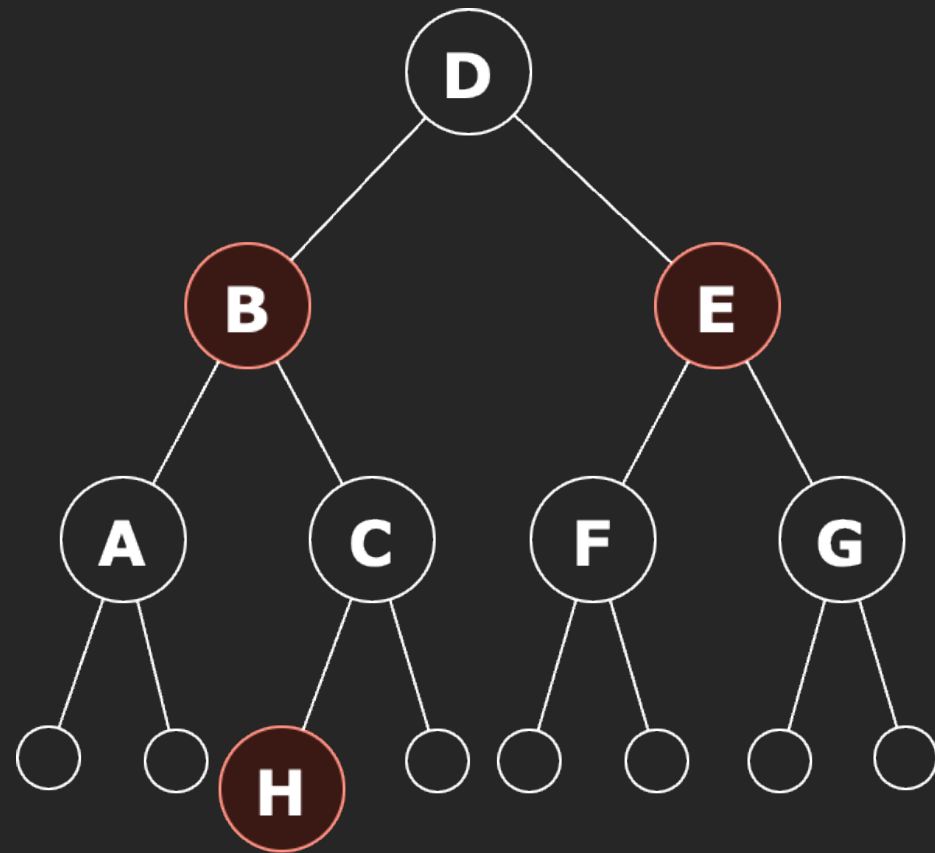
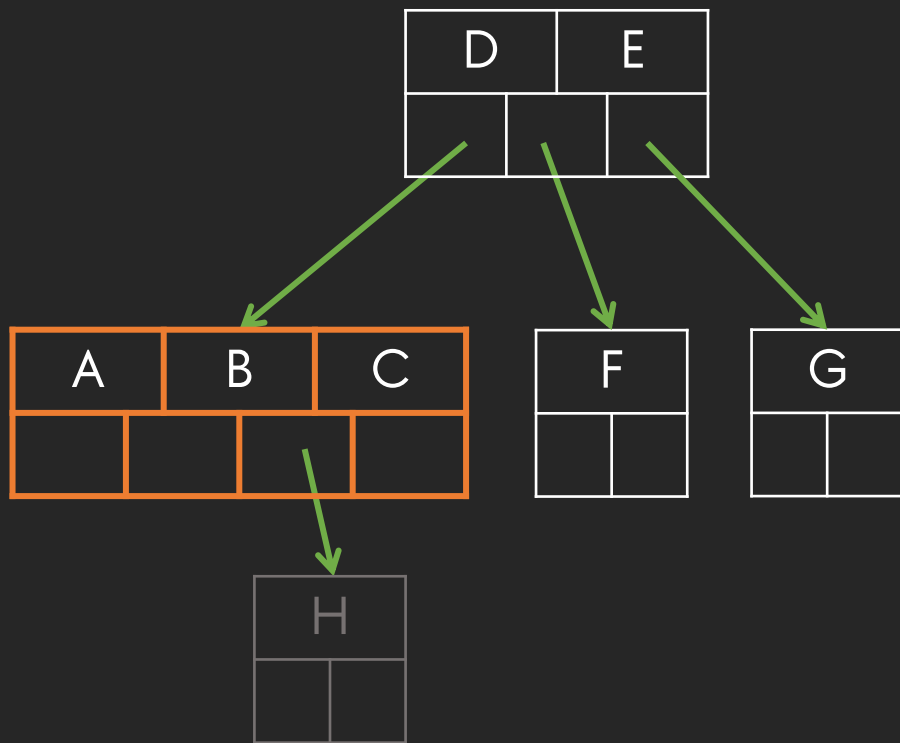


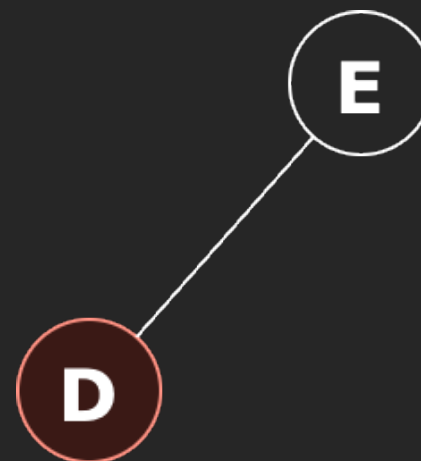
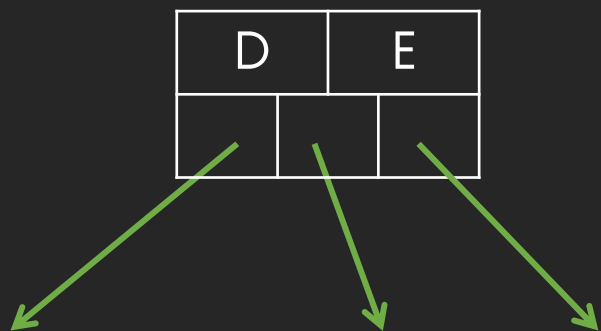


К более сложному случаю...

# Вставка в КЧД. 4-вершина

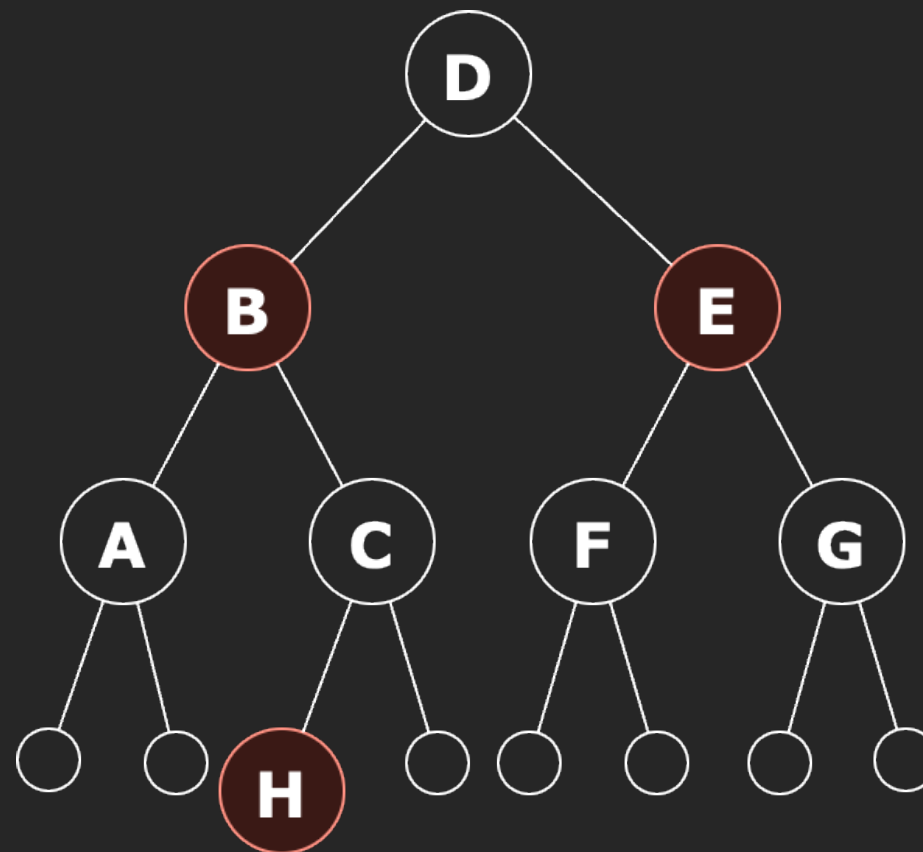
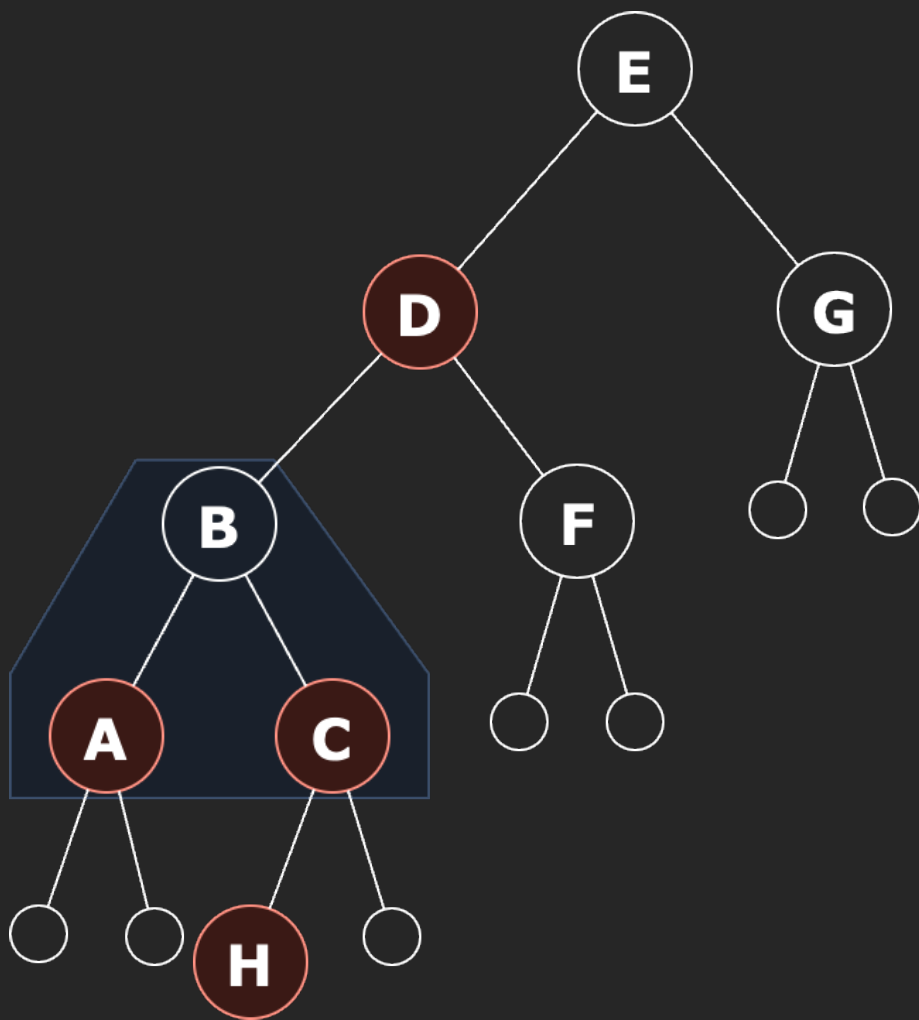
4-вершина – левый  
потомок 3-вершины





# Вставка в КЧД. 4-вершина

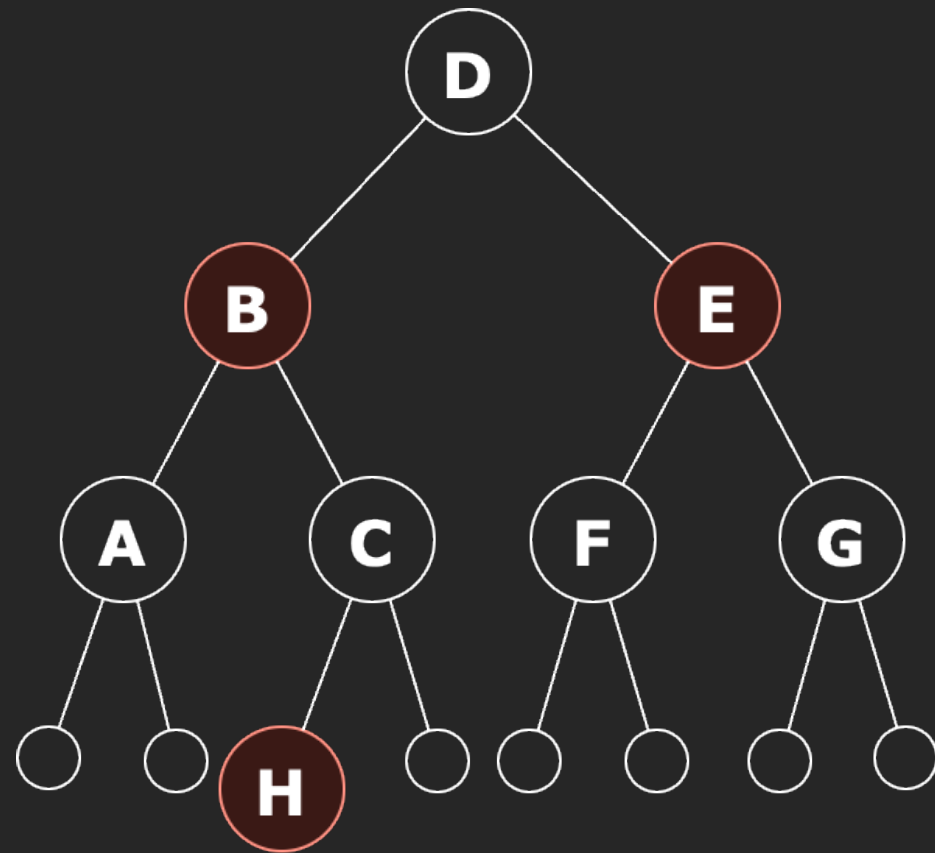
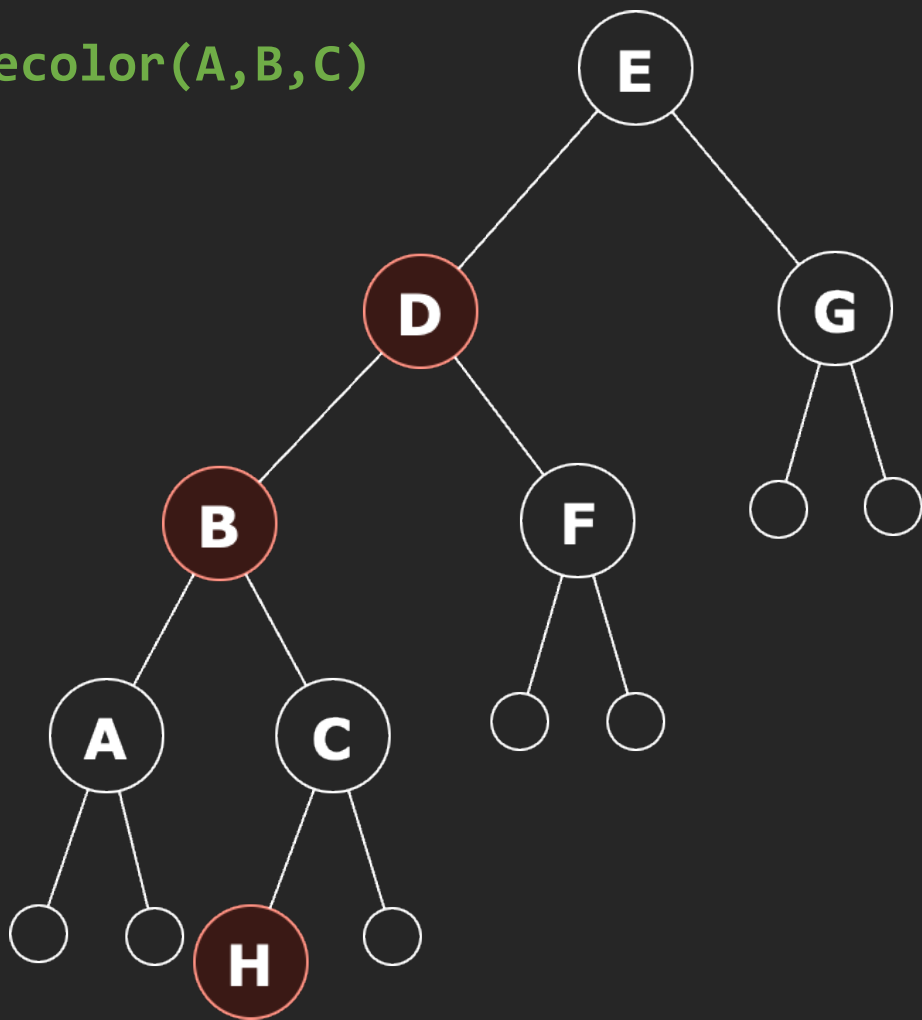
4-вершина – левый  
потомок 3-вершины

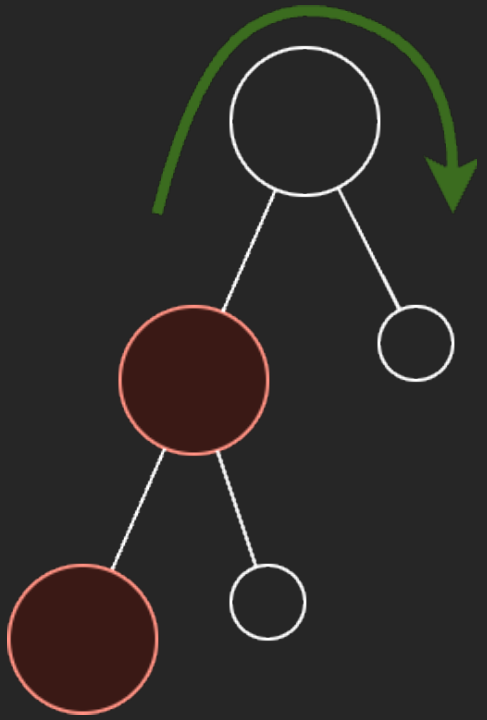


# Вставка в КЧД. 4-вершина

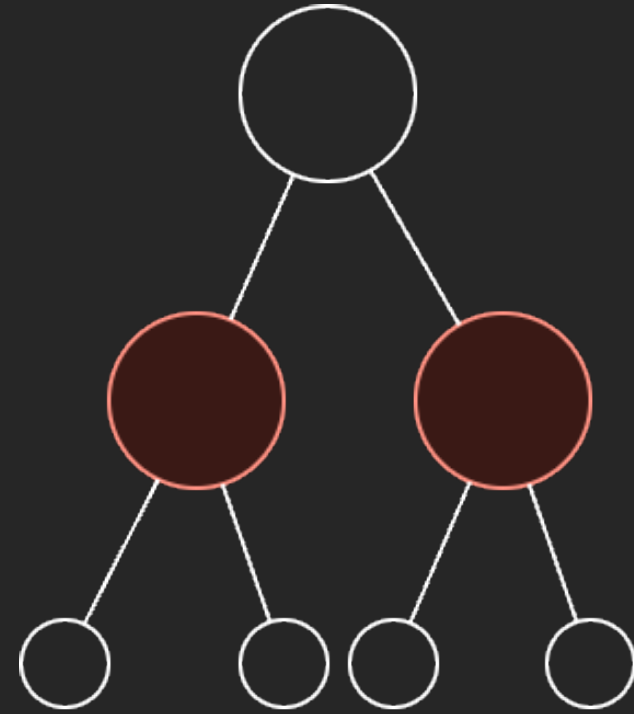
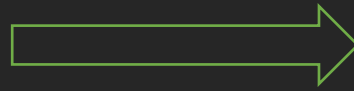
4-вершина – левый  
потомок 3-вершины

recolor(A,B,C)



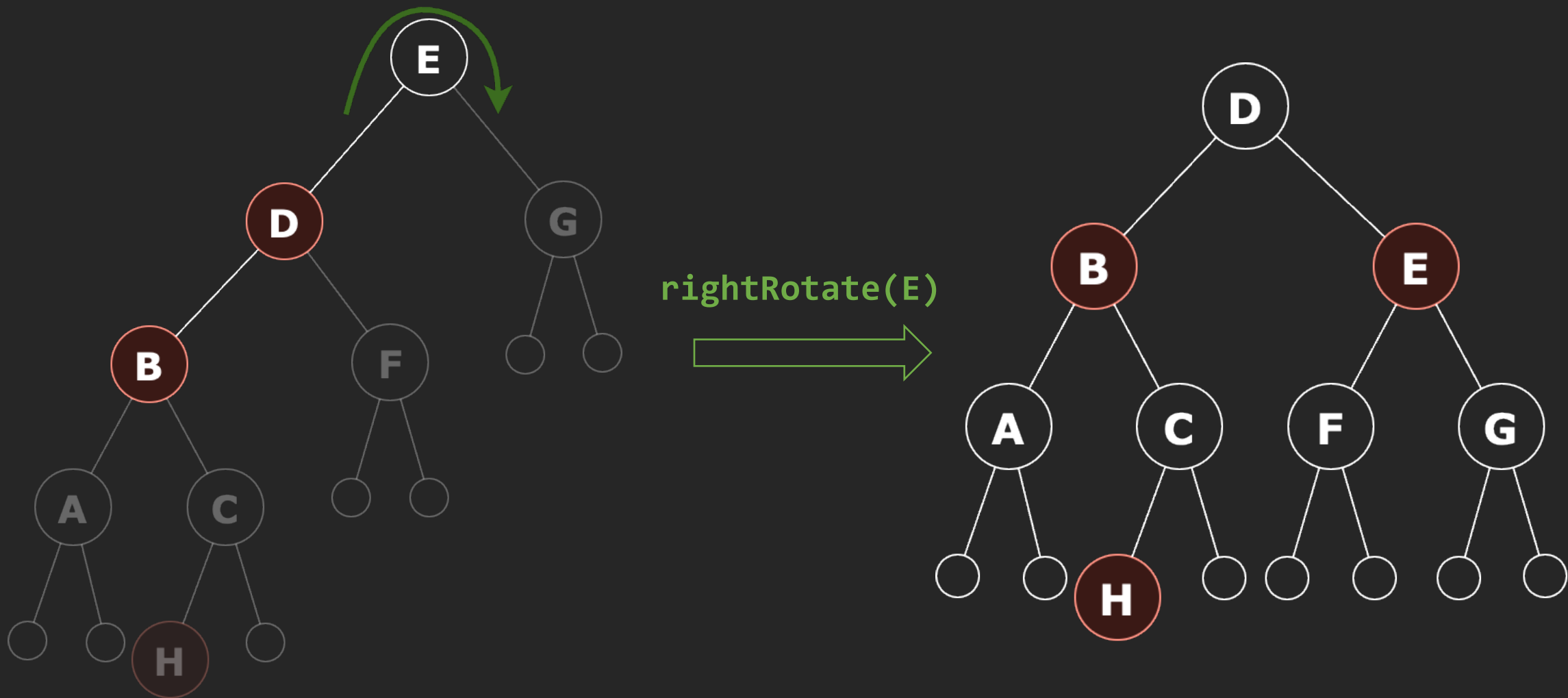


БАЗОВЫЙ СЛУЧАЙ!



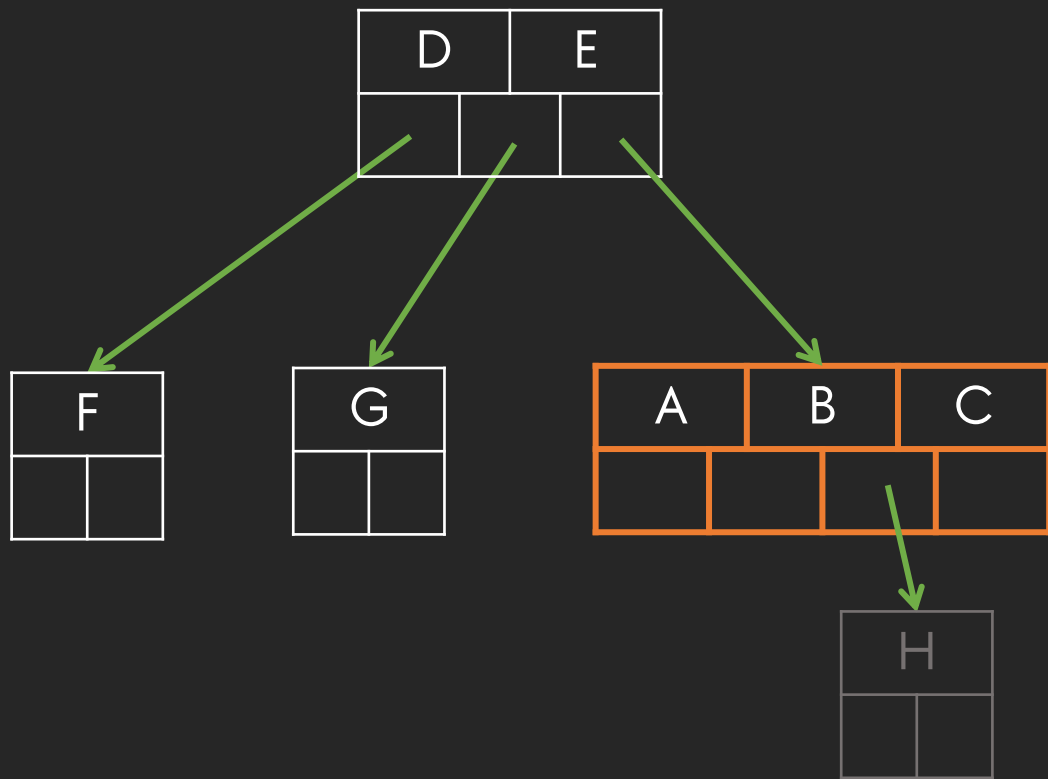
# Вставка в КЧД. 4-вершина

4-вершина – левый  
потомок 3-вершины



# Вставка в КЧД. 4-вершина

4-вершина – правый  
потомок 3-вершины

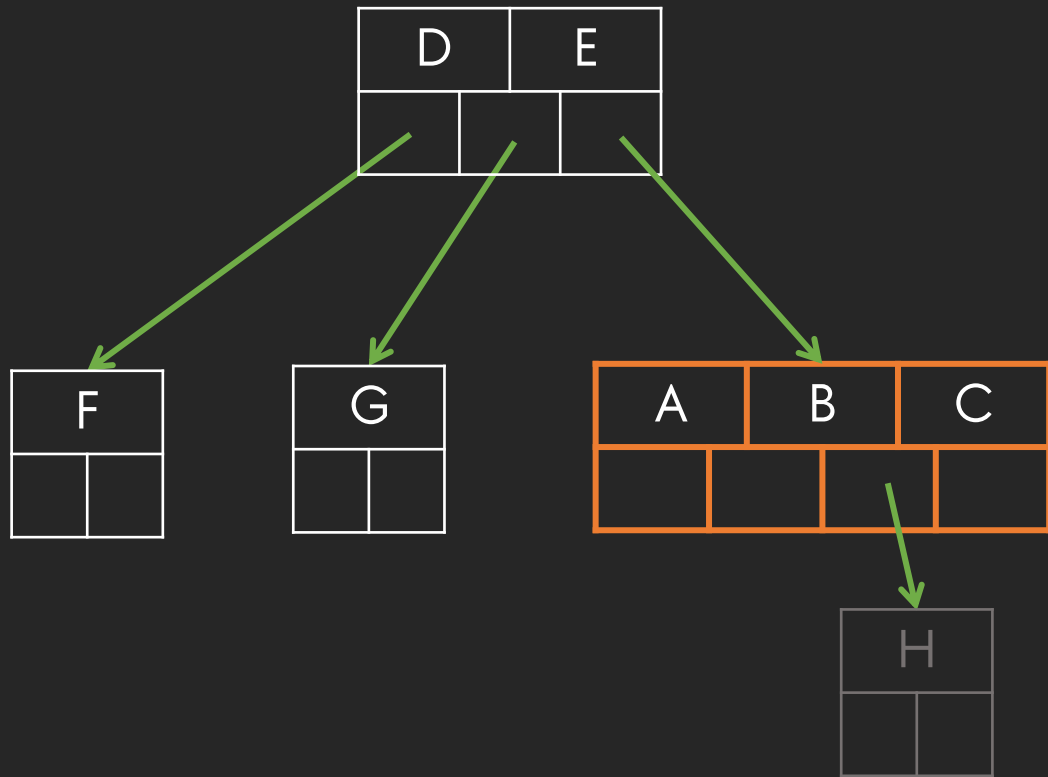


`insert(H)`     $H > B$  и  $H < C$

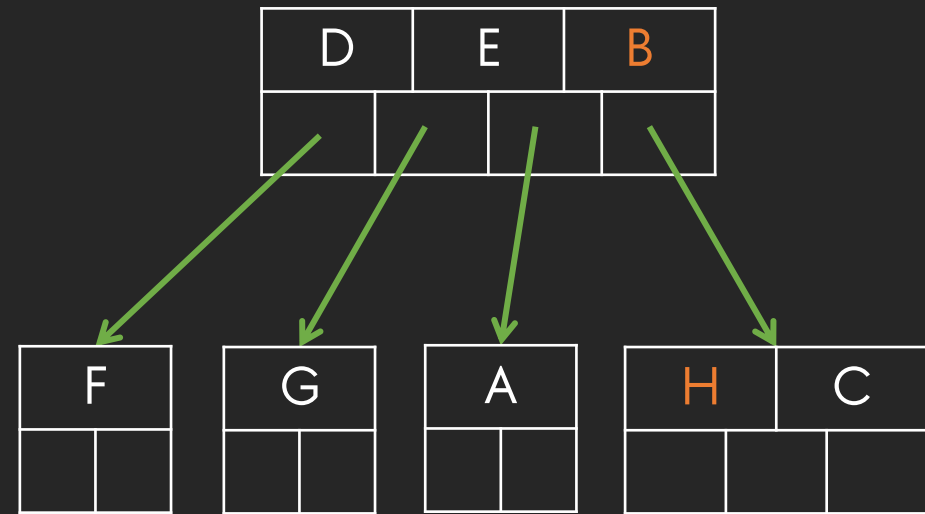


# Вставка в КЧД. 4-вершина

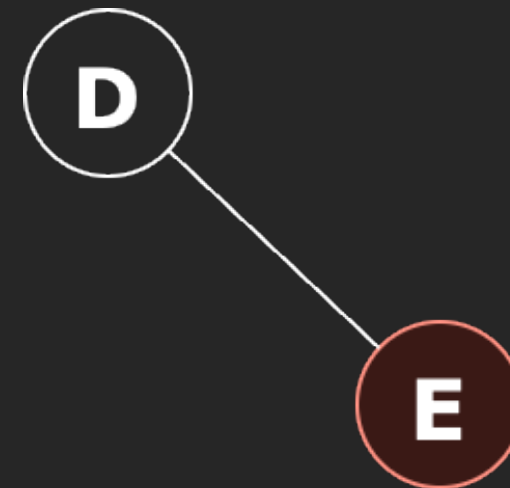
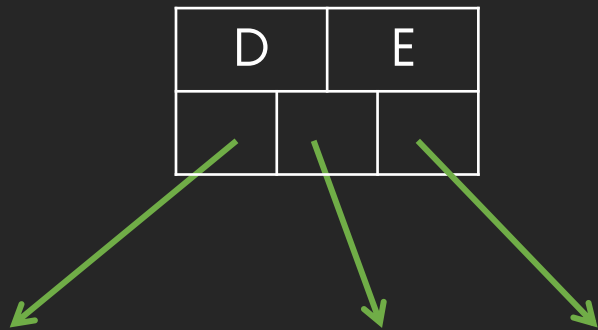
4-вершина – правый  
потомок 3-вершины



insert(H)  $H > B$  и  $H < C$

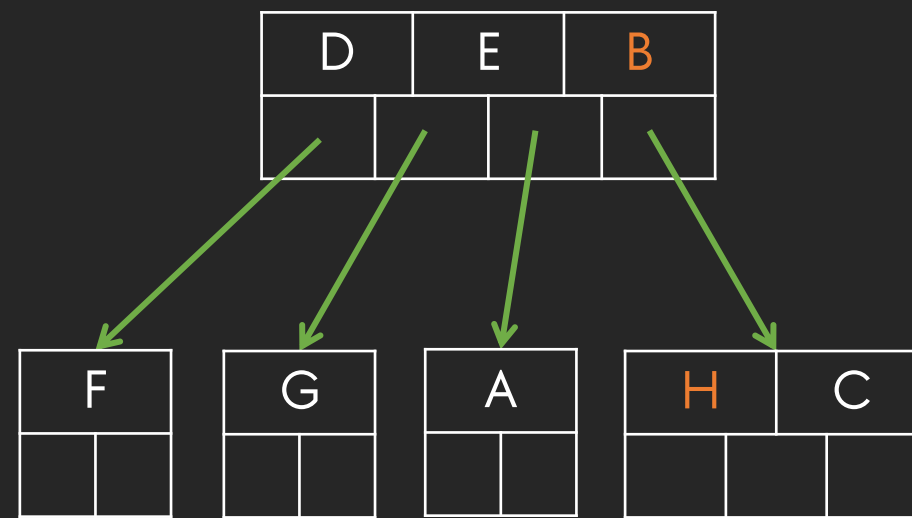
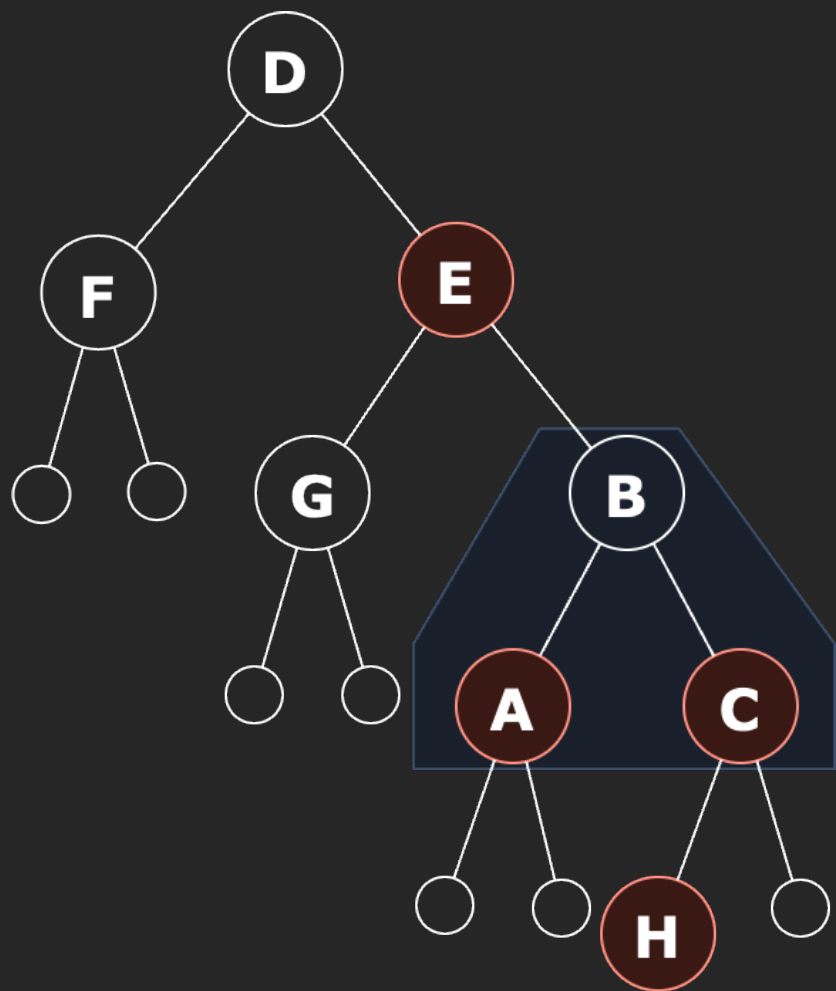


Сразу берем  
интересующее нас  
представление 2-вершины



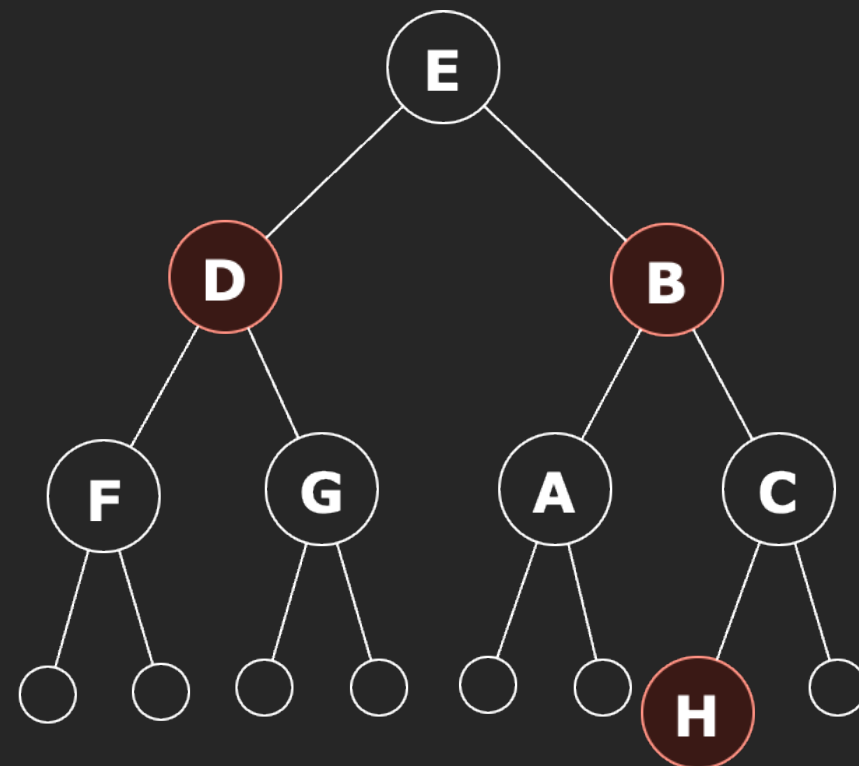
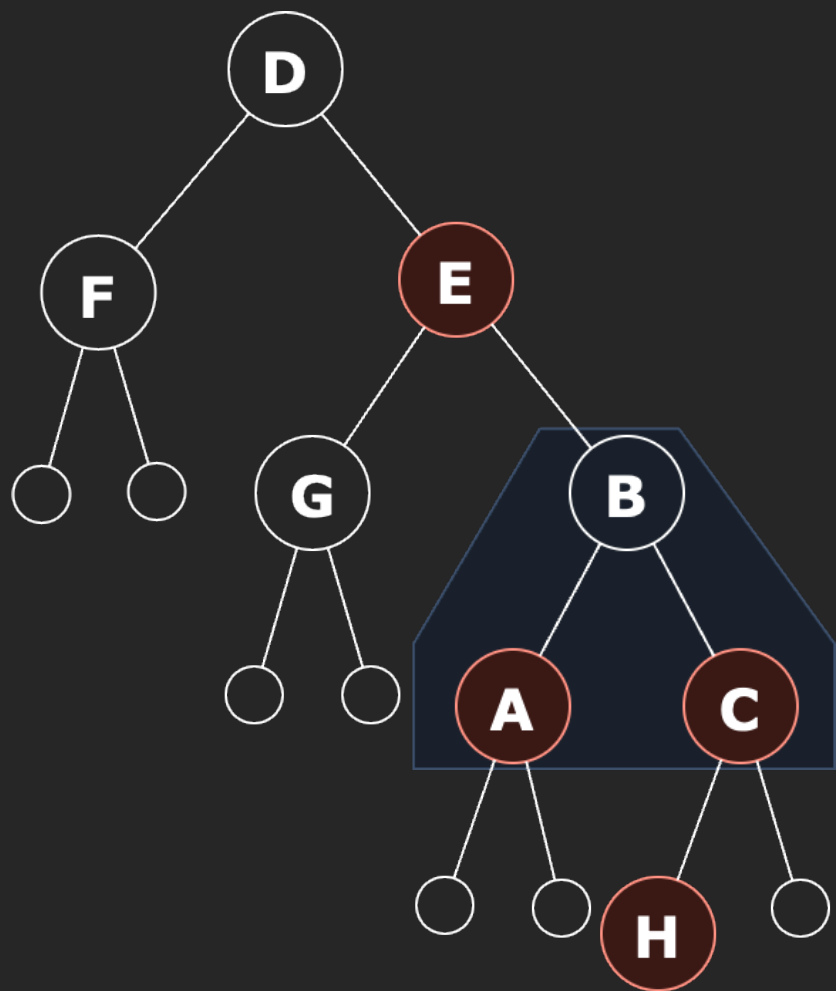
# Вставка в КЧД. 4-вершина

4-вершина – правый  
потомок 3-вершины



# Вставка в КЧД. 4-вершина

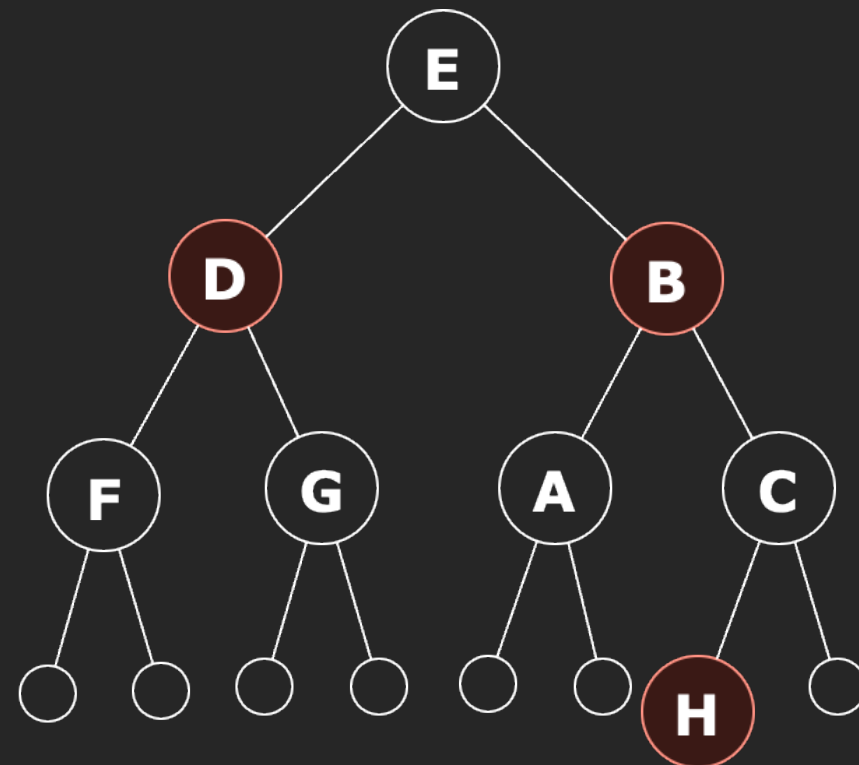
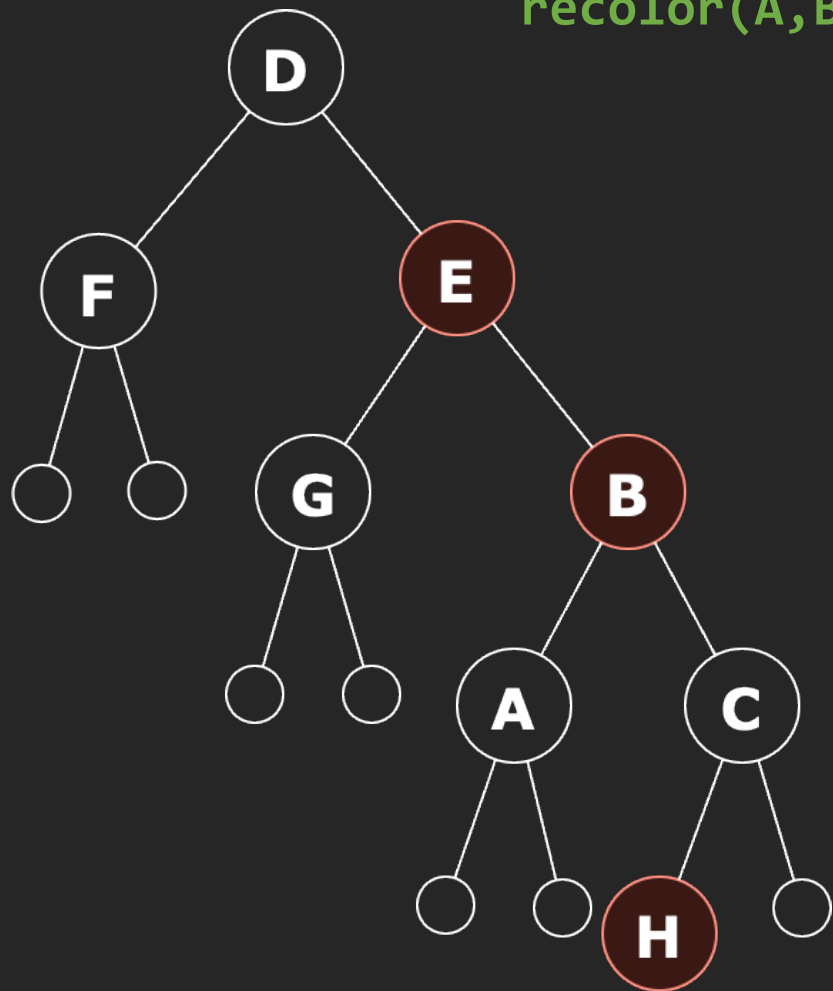
4-вершина – правый  
потомок 3-вершины

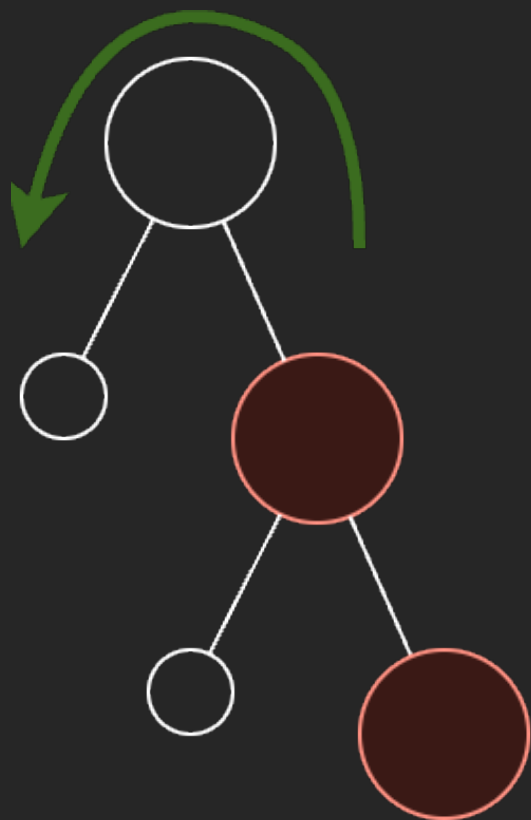


# Вставка в КЧД. 4-вершина

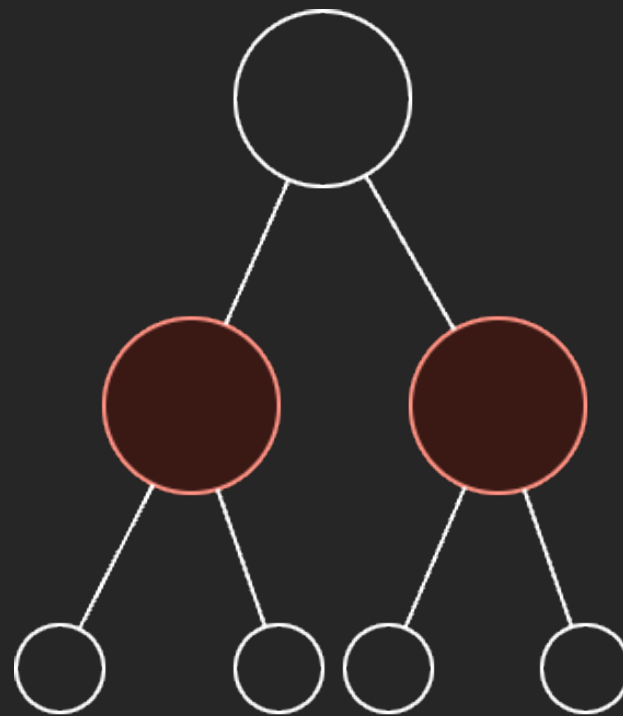
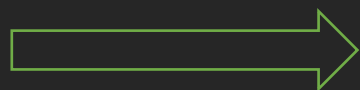
4-вершина – правый  
потомок 3-вершины

recolor(A,B,C)



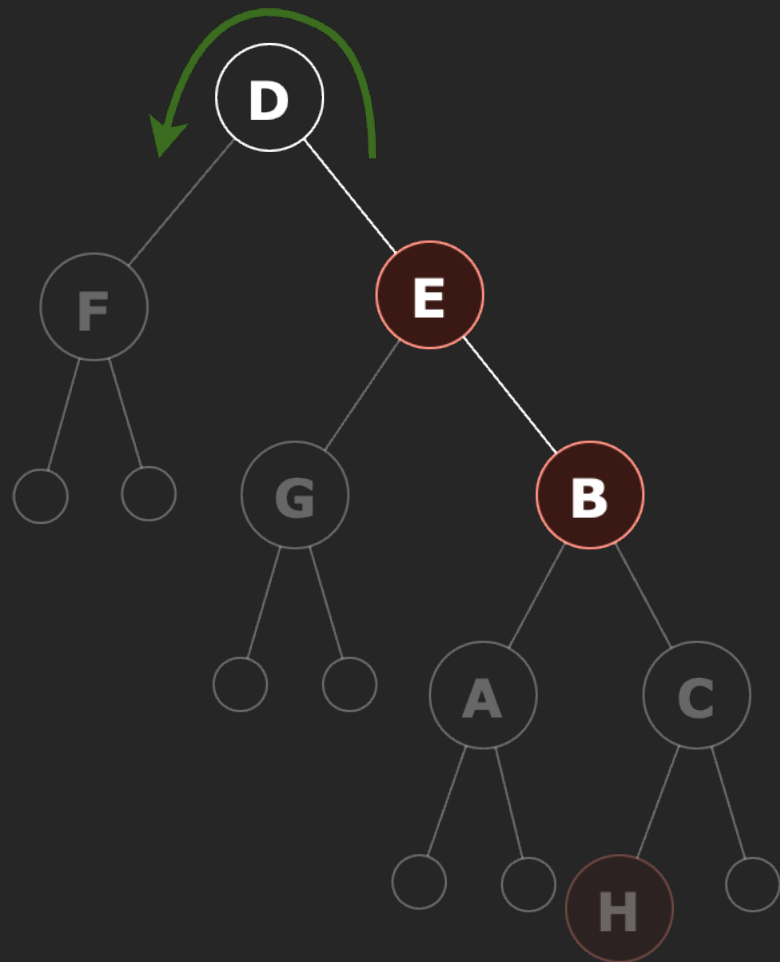


БАЗОВЫЙ СЛУЧАЙ!

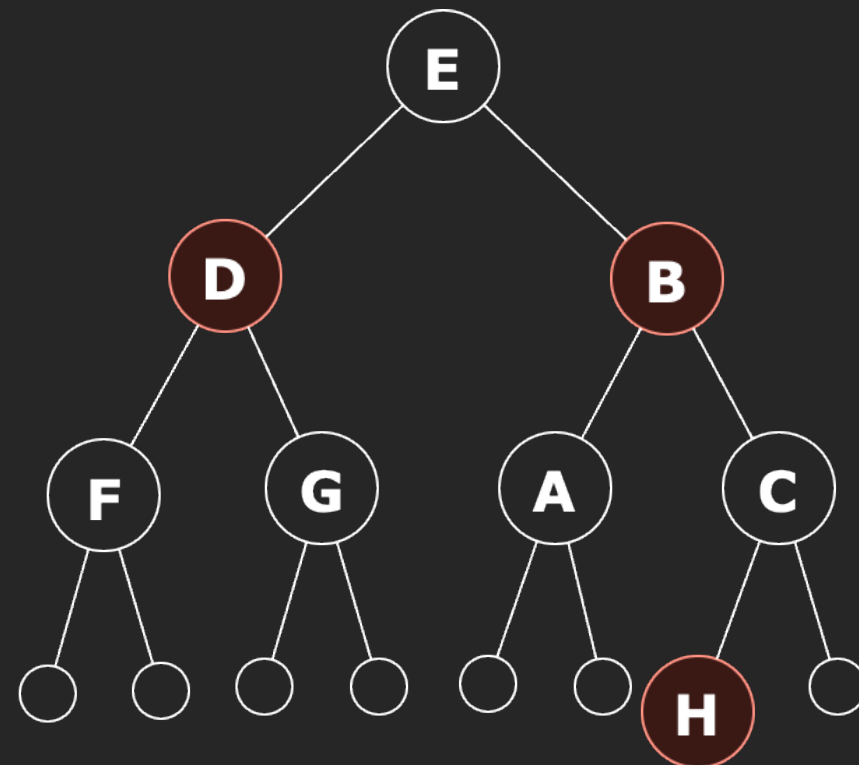
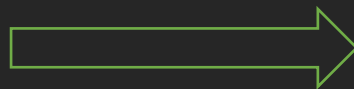


# Вставка в КЧД. 4-вершина

4-вершина – правый  
потомок 3-вершины



leftRotate(D)



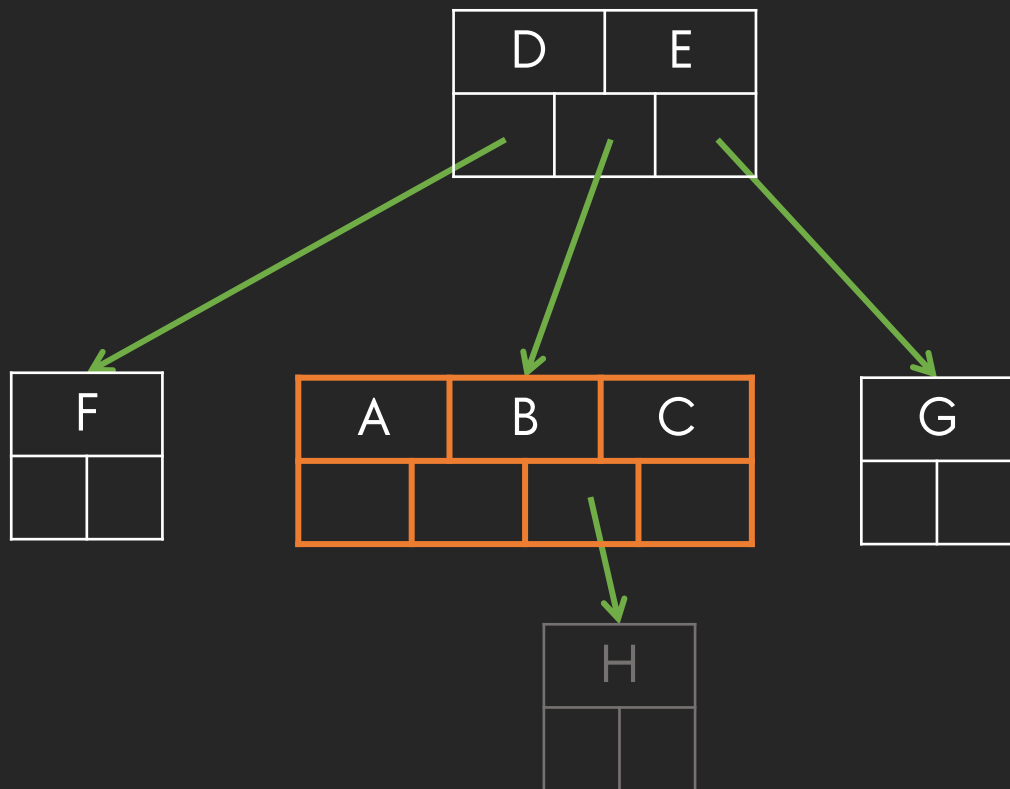


# Так-так-так...



# Вставка в КЧД. 4-вершина

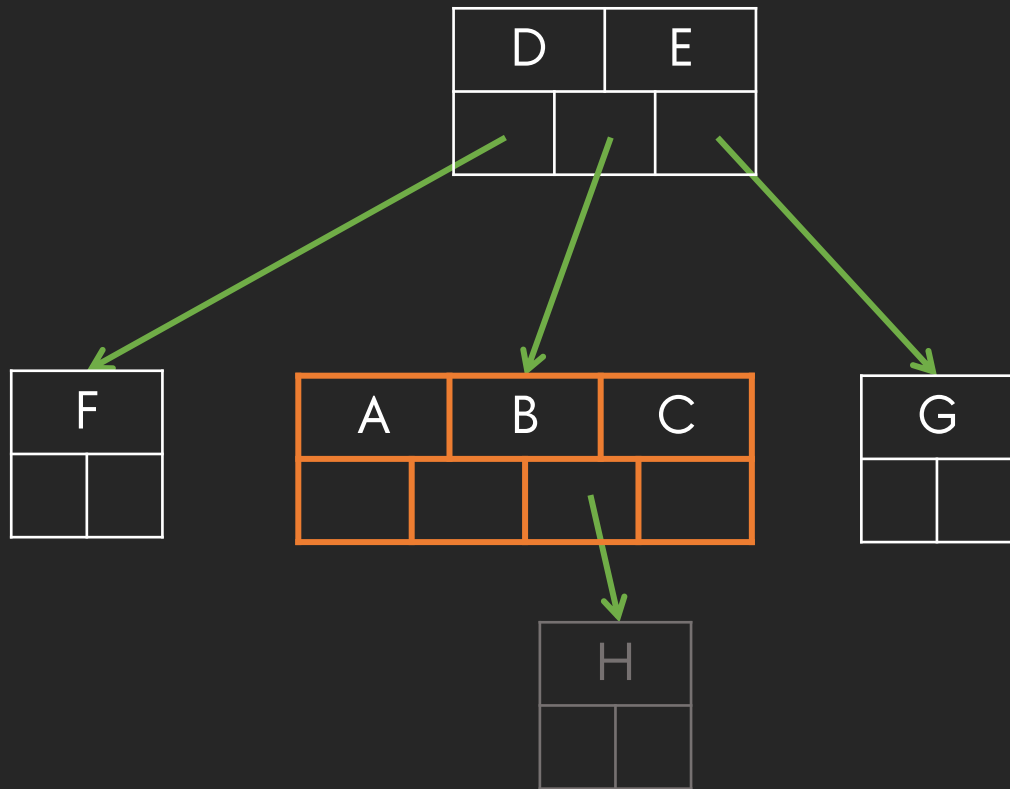
4-вершина – средний  
потомок 3-вершины



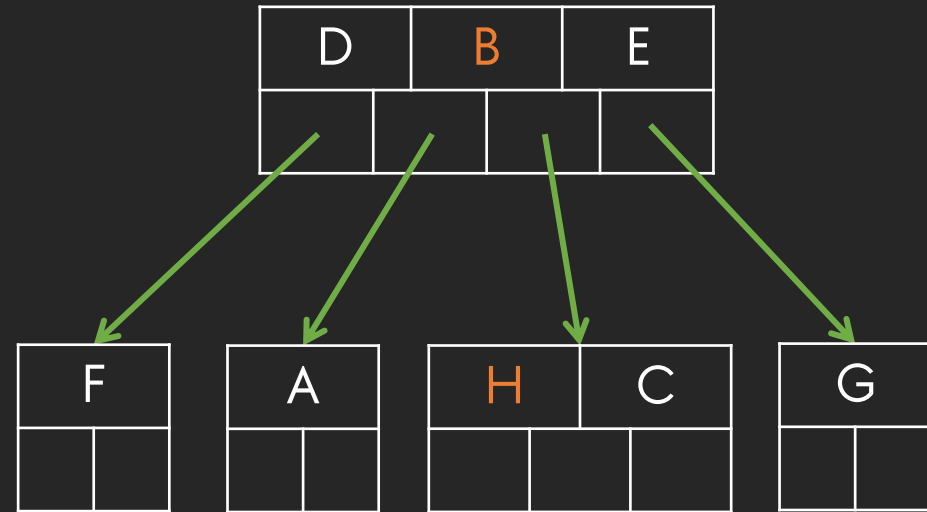
`insert(H)`     $H > B$  и  $H < C$

# Вставка в КЧД. 4-вершина

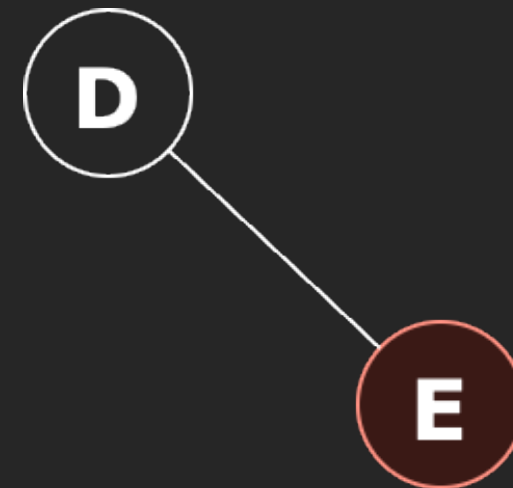
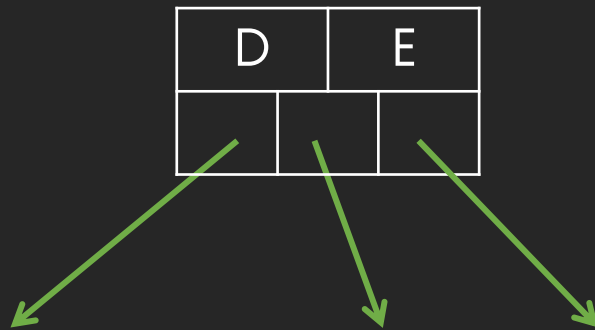
4-вершина – средний  
потомок 3-вершины



insert(H)  $H > B$  и  $H < C$

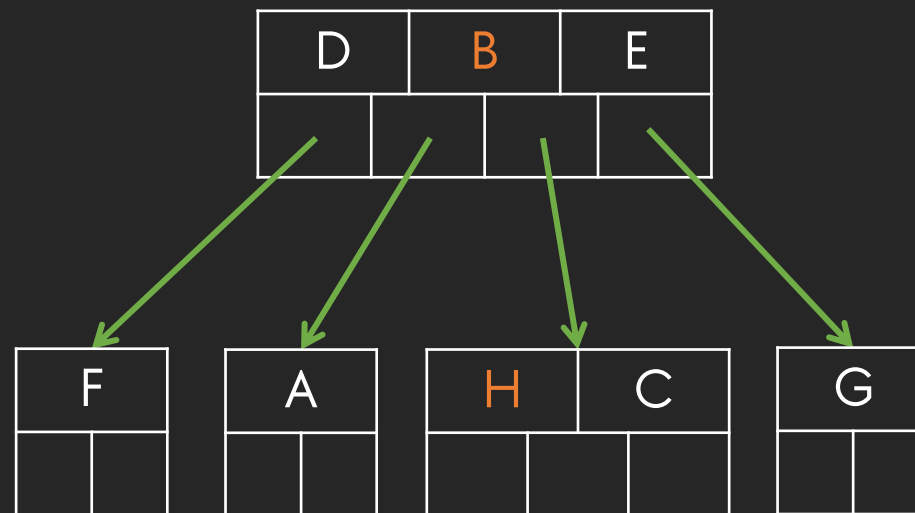
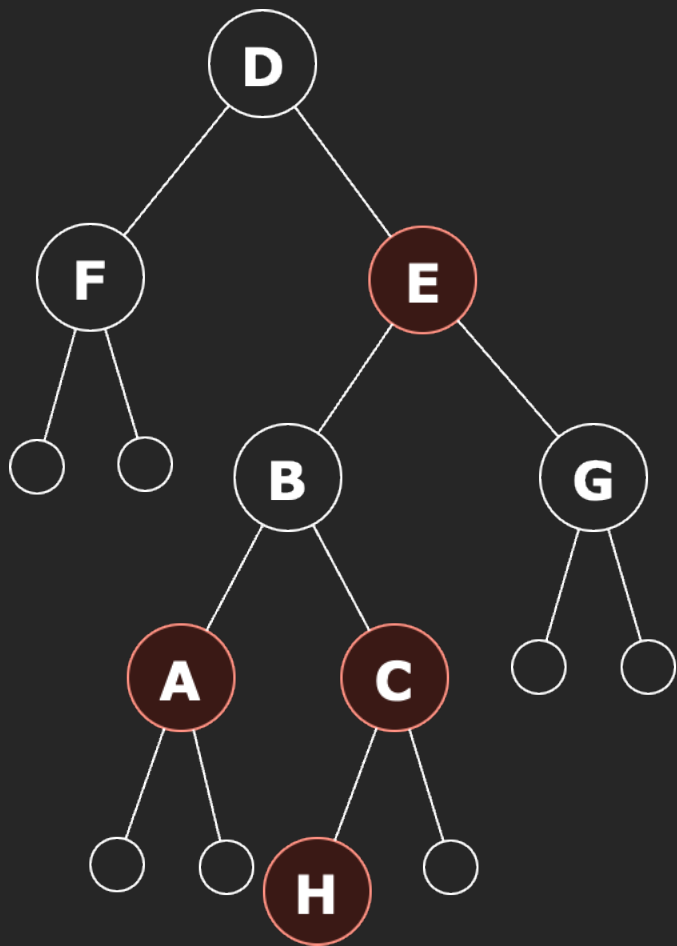


От выбора представления 2-  
вершины зависит результат!



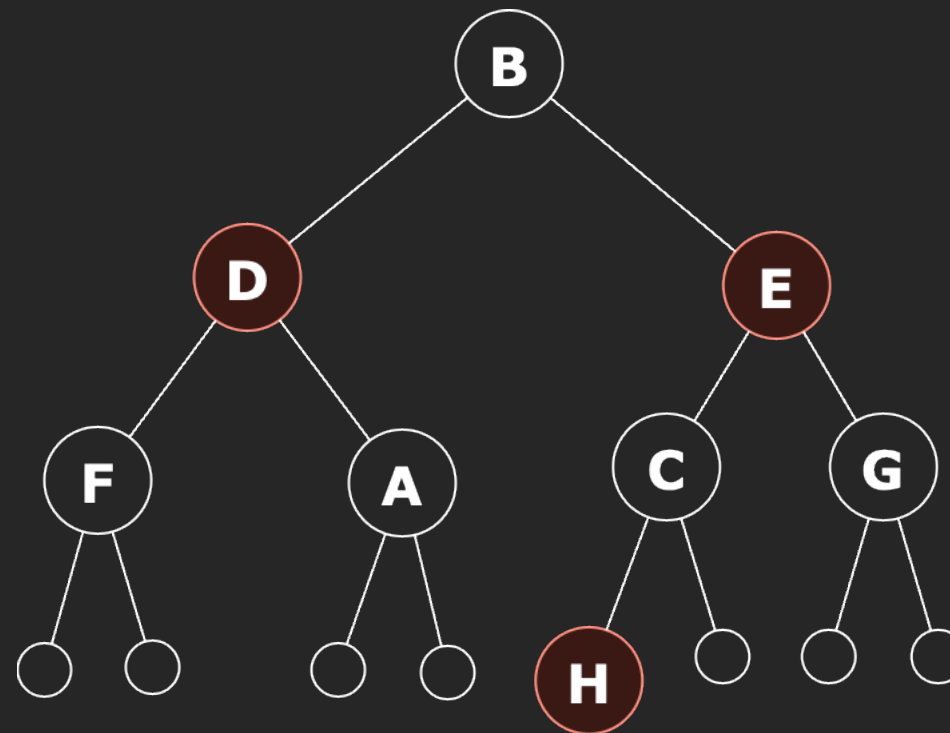
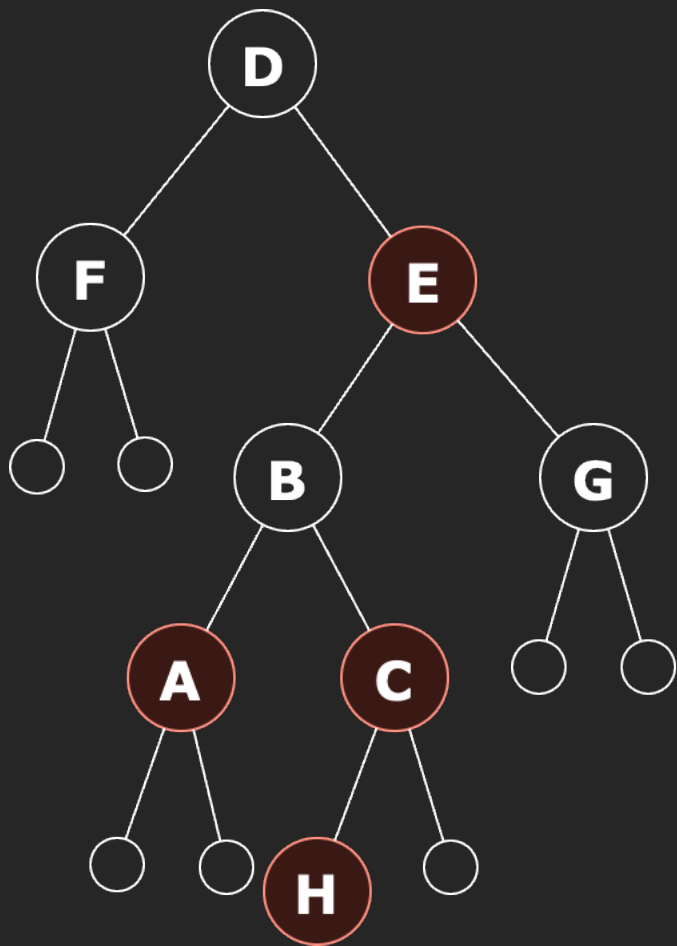
# Вставка в КЧД. 4-вершина

4-вершина – средний  
потомок 3-вершины



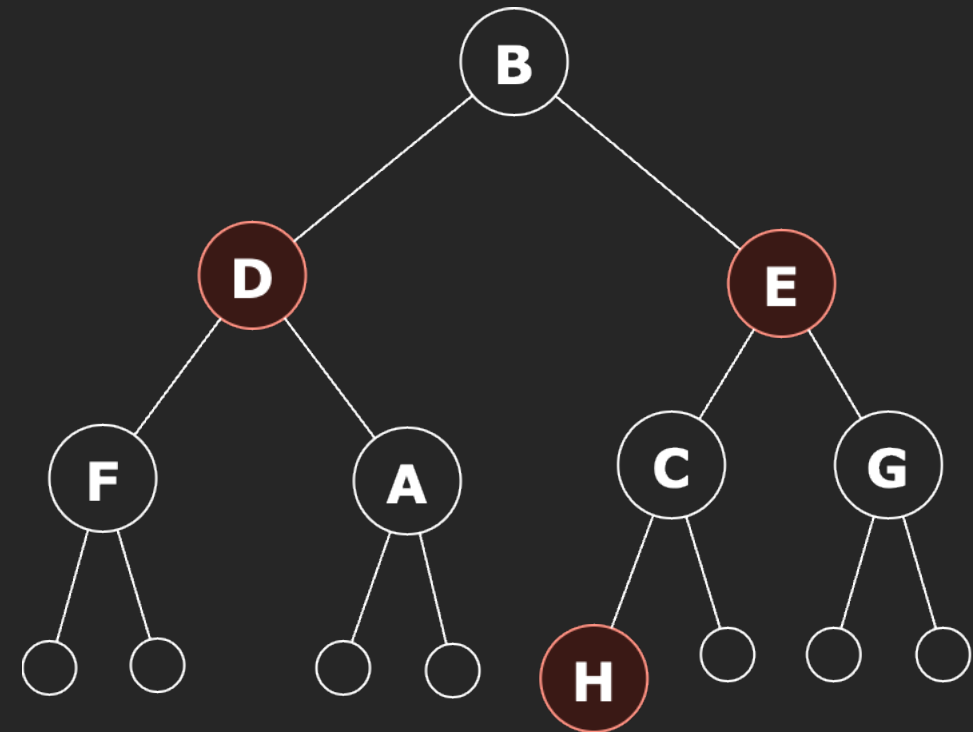
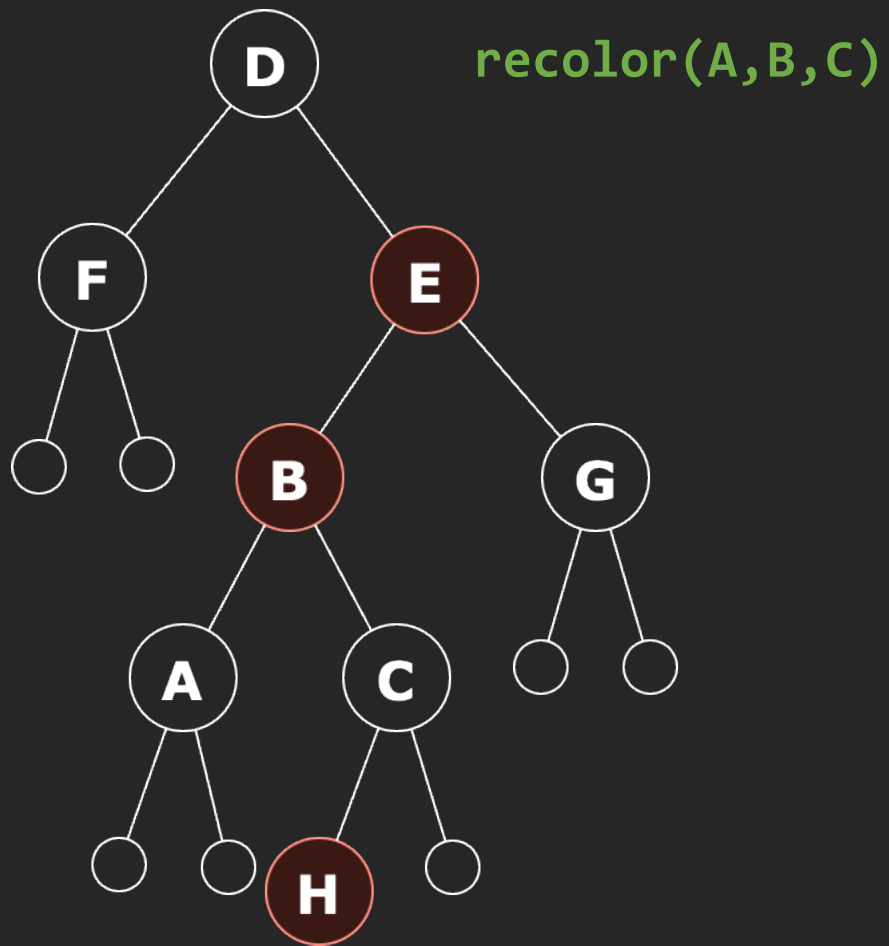
# Вставка в КЧД. 4-вершина

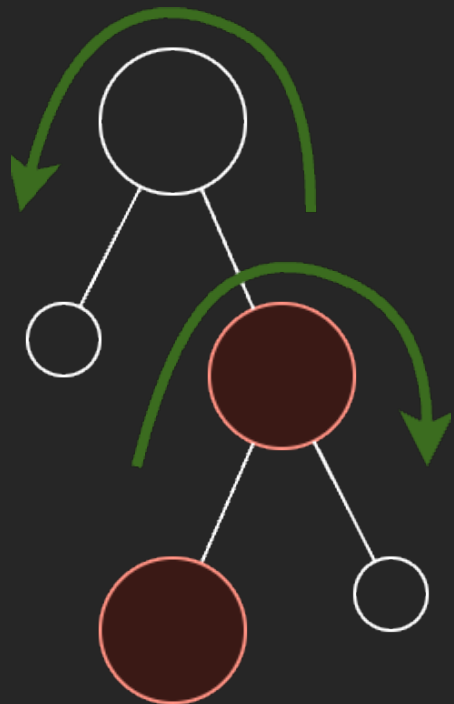
4-вершина – средний  
потомок 3-вершины



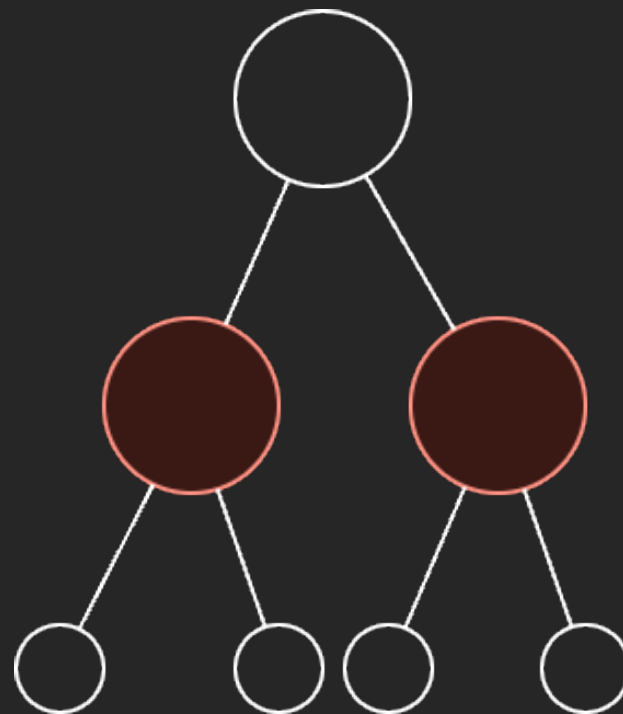
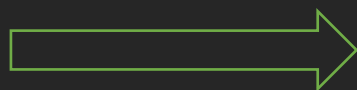
# Вставка в КЧД. 4-вершина

4-вершина – средний  
потомок 3-вершины



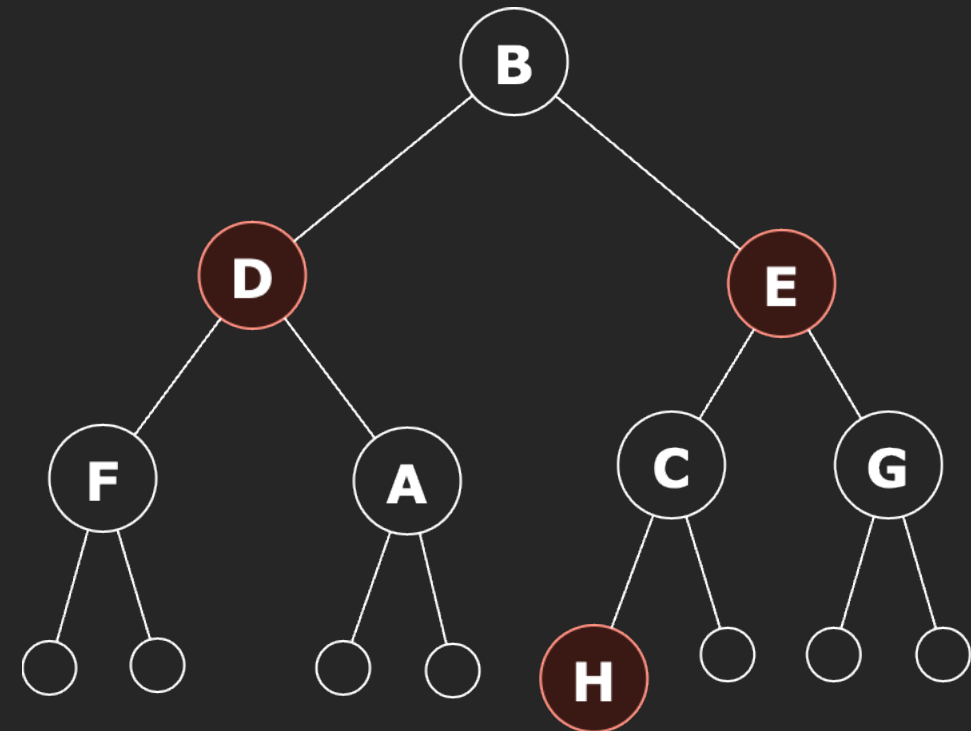
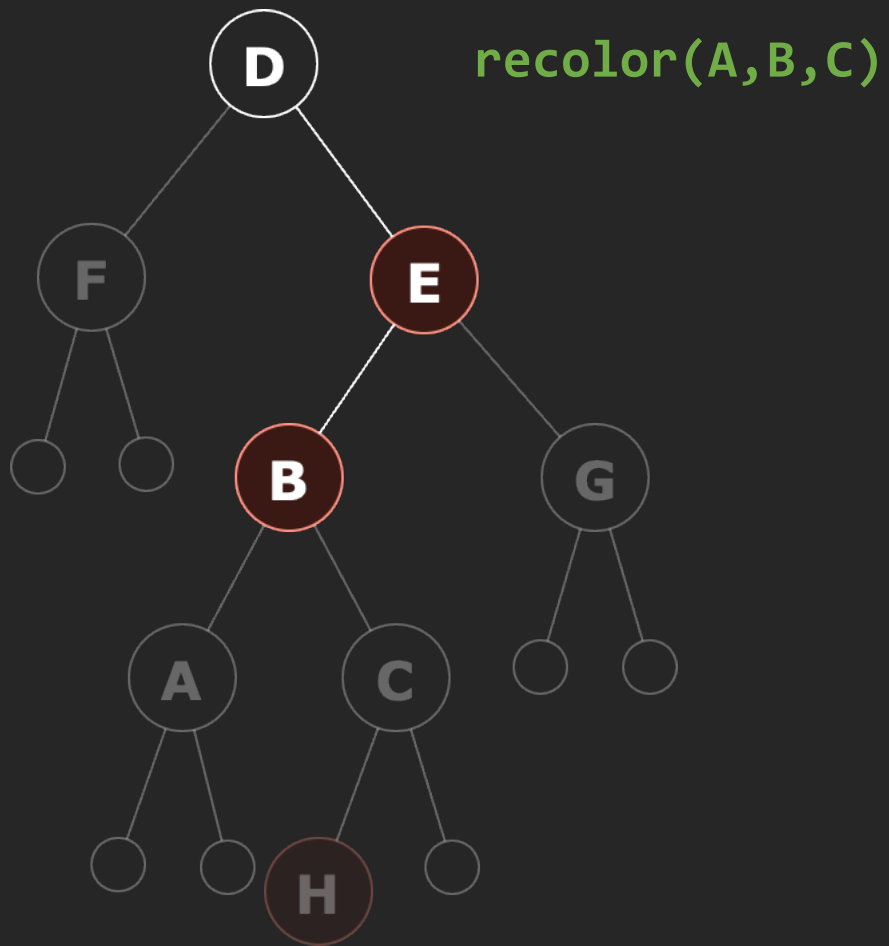


БАЗОВЫЙ СЛУЧАЙ!



# Вставка в КЧД. 4-вершина

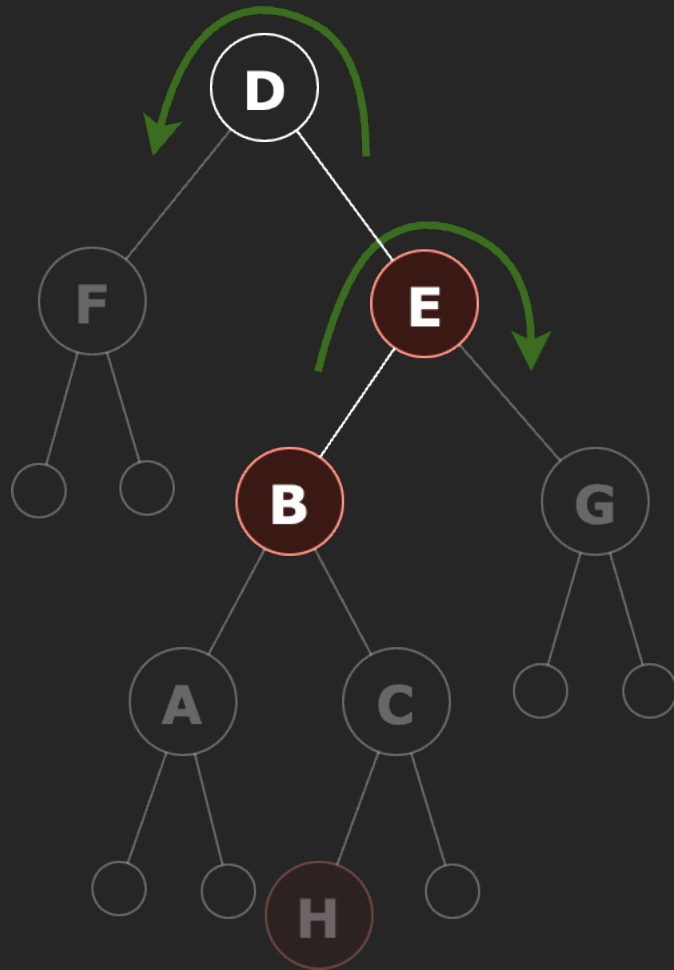
4-вершина – средний  
потомок 3-вершины



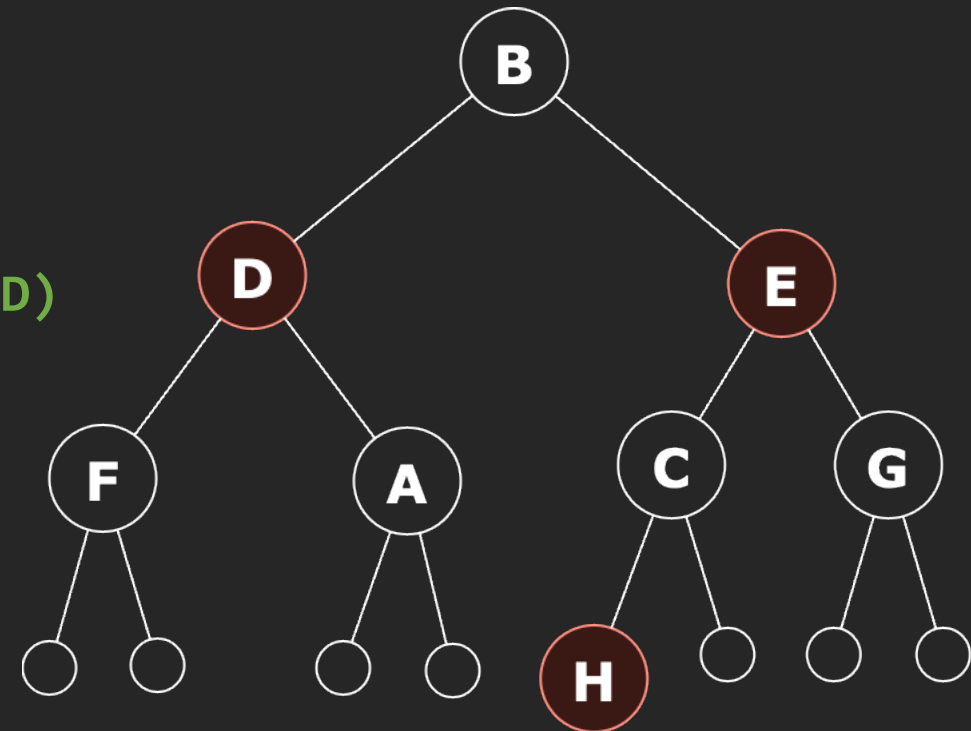
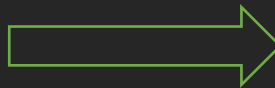


# Вставка в КЧД. 4-вершина

4-вершина – средний  
потомок 3-вершины

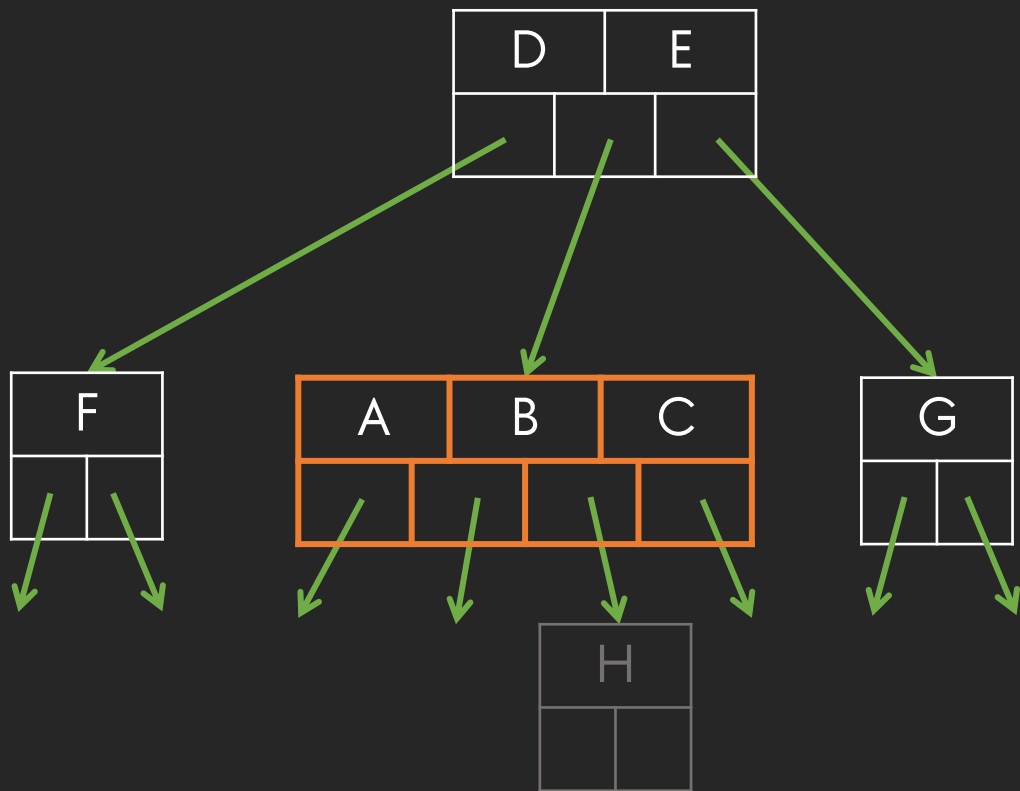


`rightLeftRotate(D)`

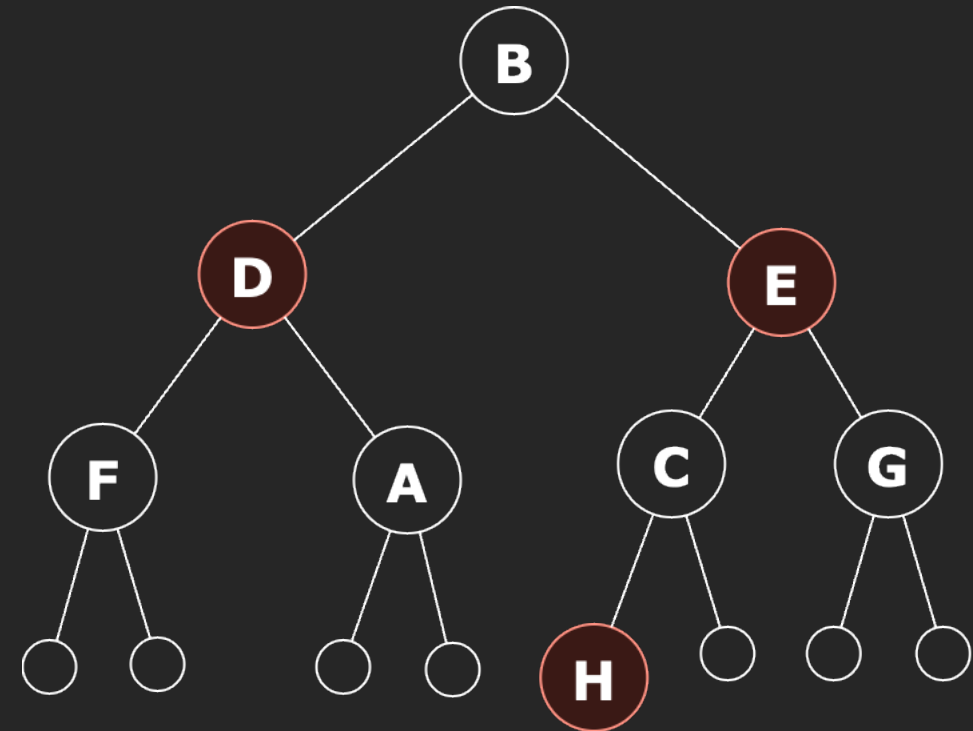


# Вставка в КЧД. 4-вершина

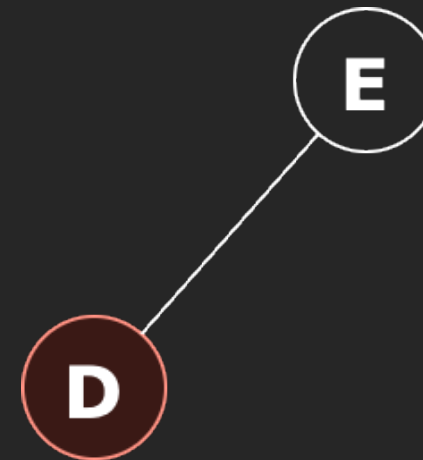
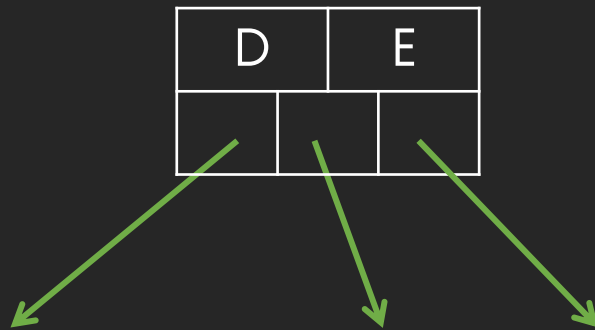
4-вершина – средний  
потомок 3-вершины



insert(H)  $H > B$  и  $H < C$

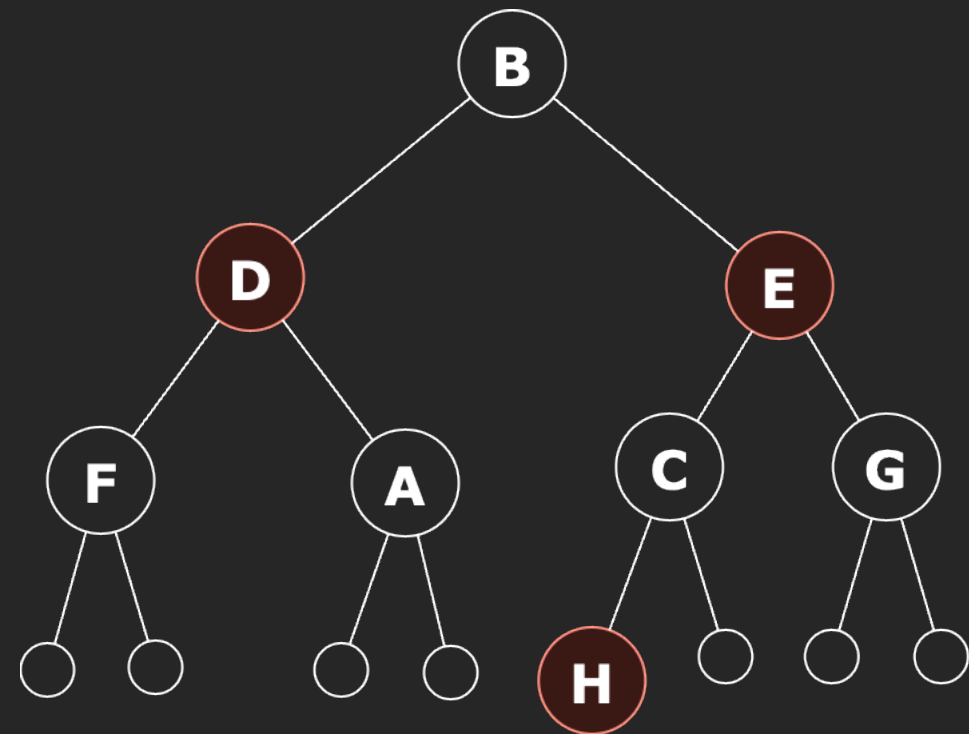
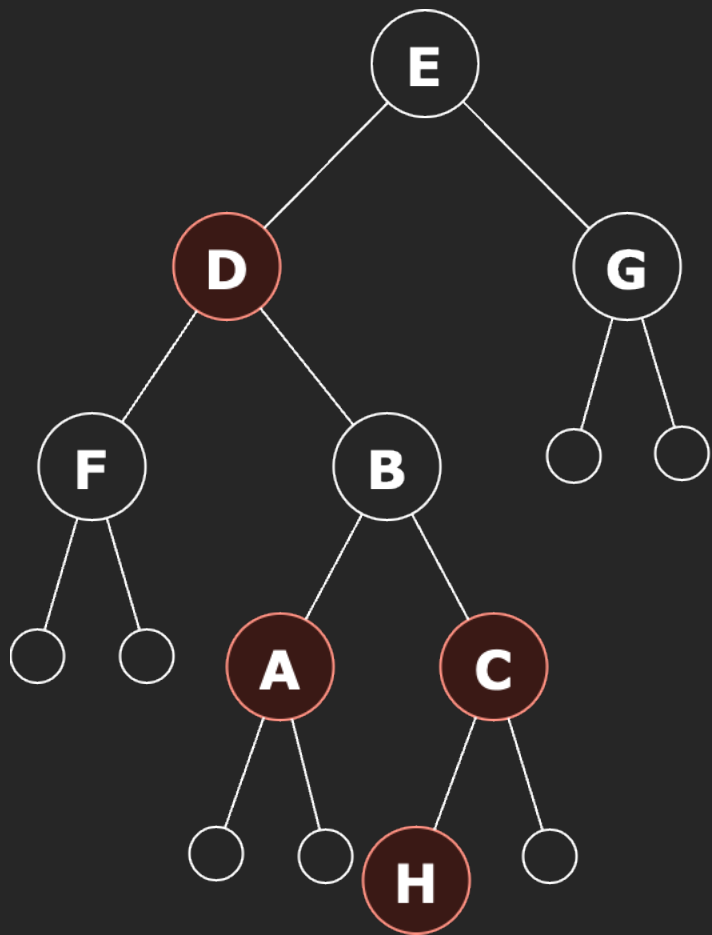


От выбора представления 2-  
вершины зависит результат!



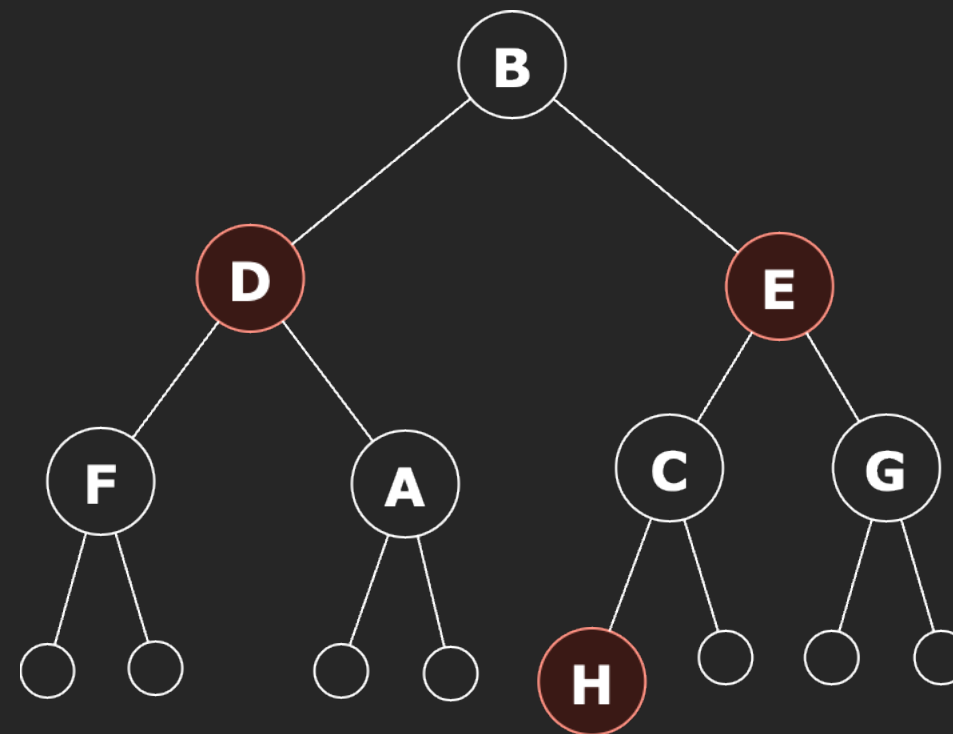
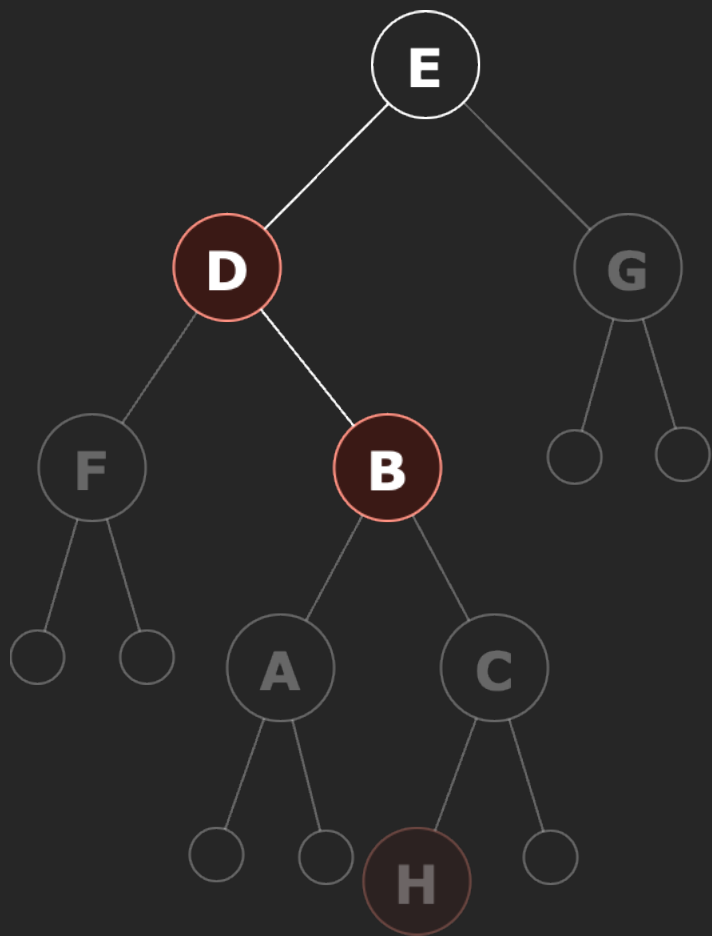
# Вставка в КЧД. 4-вершина

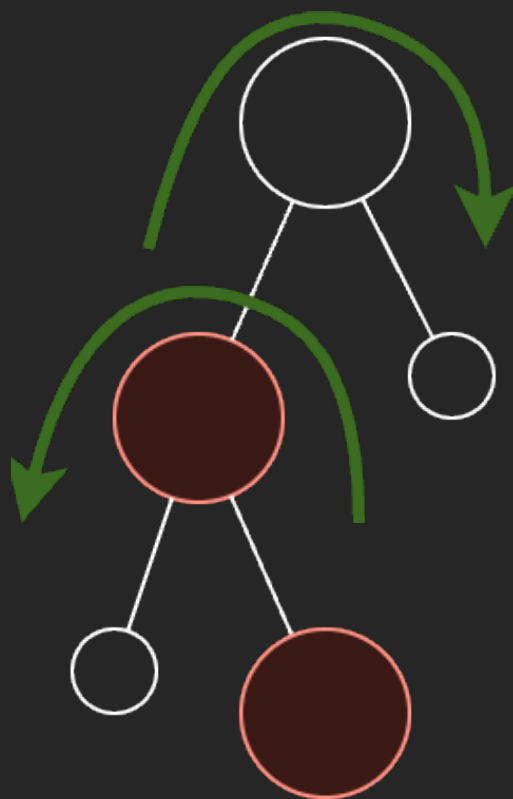
4-вершина – средний  
потомок 3-вершины



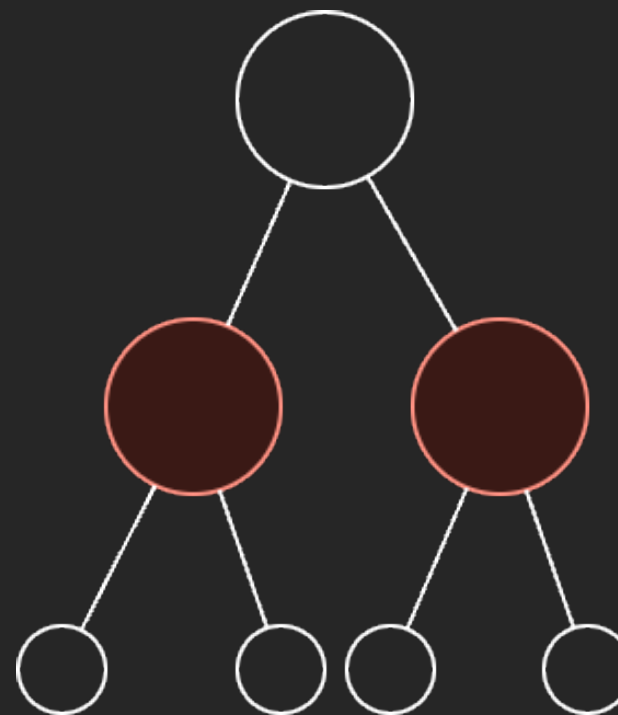
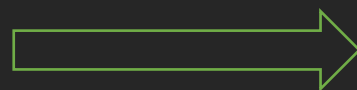
# Вставка в КЧД. 4-вершина

4-вершина – средний  
потомок 3-вершины



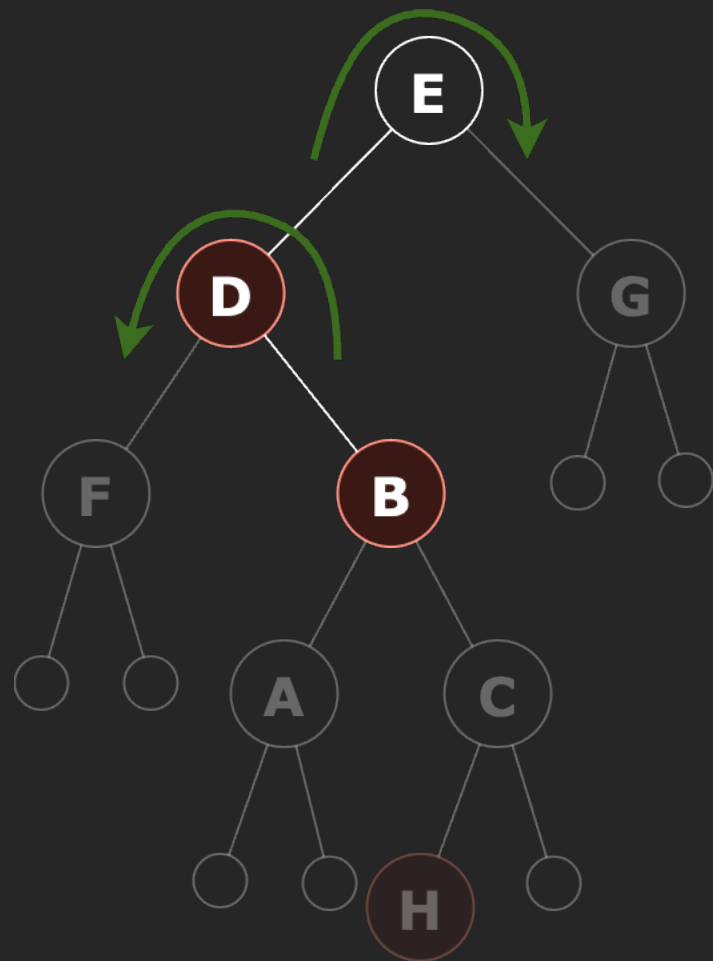


БАЗОВЫЙ СЛУЧАЙ!

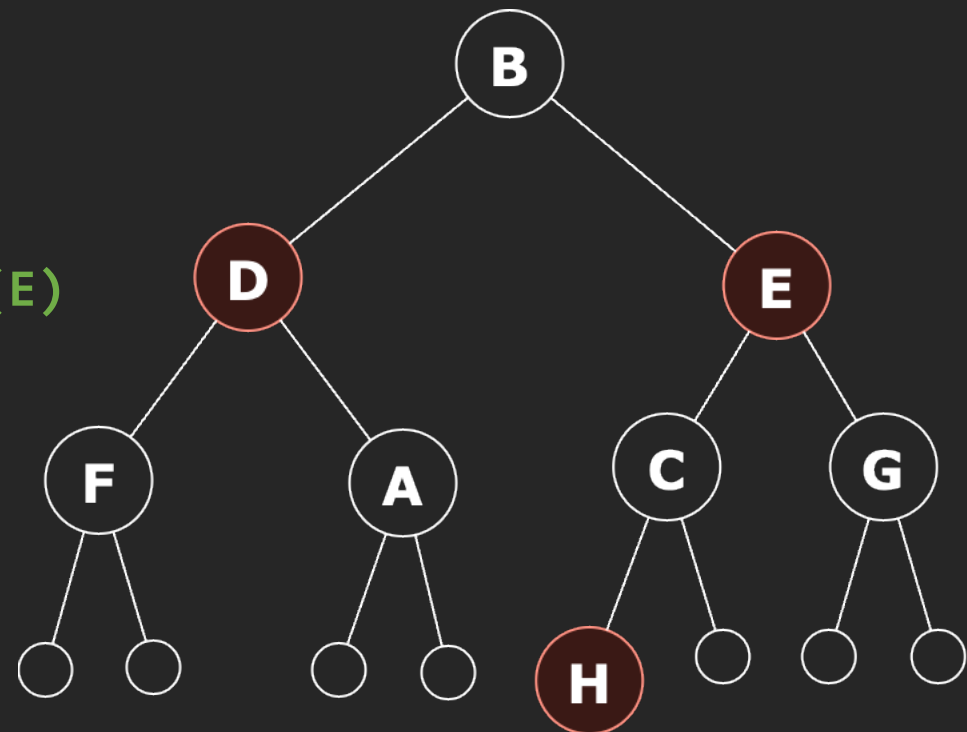
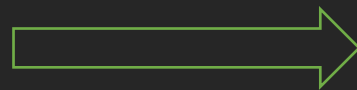


# Вставка в КЧД. 4-вершина

4-вершина – средний  
потомок 3-вершины



leftRightRotate(E)





Ну и зачем всё это?



# Вставка в КЧД. ИТОГО

---

Основные конфигурации вставки нового ключа в красно-черное дерево исходят из представления целевого сбалансированного 2-3-4 дерева

# Вставка в КЧД. ИТОГО

После того, как мы нашли место  $v$  для вставки  $x$ ...

3-вершина?

цвет  $v$  – красный  
родитель  $p$  – черный  
 $v$  – правый потомок  $p$

$x > v \rightarrow \text{data} \rightarrow \text{leftRotate}(p)$

$x < v \rightarrow \text{data} \rightarrow \text{rightLeftRotate}(p)$

перекрасить  
вовлеченные вершины

цвет  $v$  – красный  
родитель  $p$  – черный  
 $v$  – левый потомок  $p$

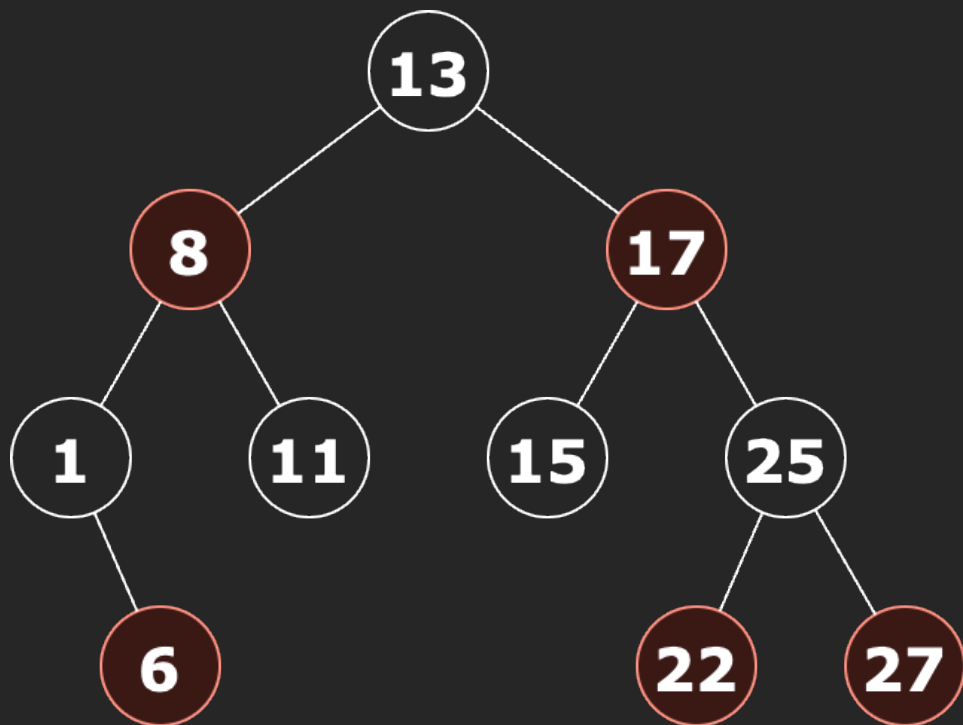
$x > v \rightarrow \text{data} \rightarrow \text{leftRightRotate}(p)$

$x < v \rightarrow \text{data} \rightarrow \text{rightRotate}(p)$

# Основные принципы удаления вершины из КЧД

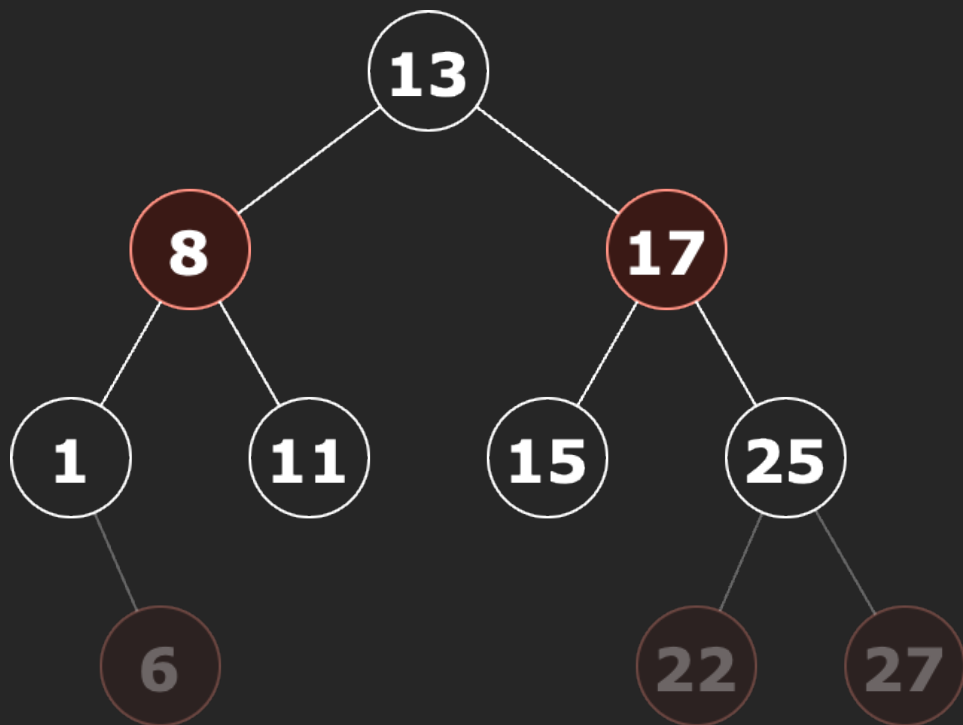
---

# Красный лист



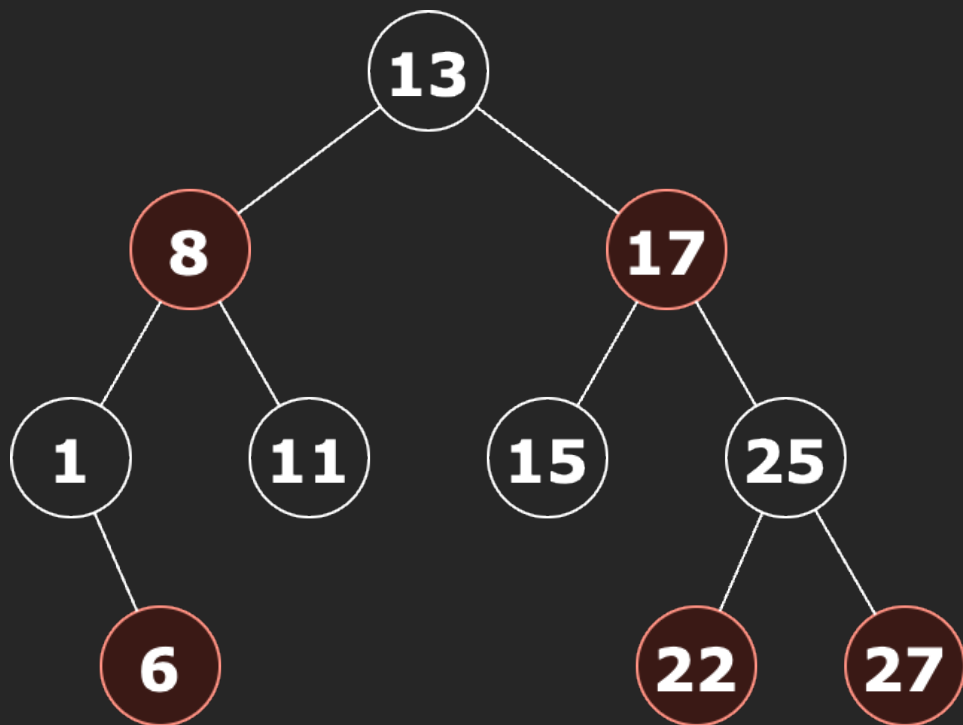
1. Не требует контроля соблюдения черной высоты
2. Может быть физически удален из дерева

# Красный ЛИСТ



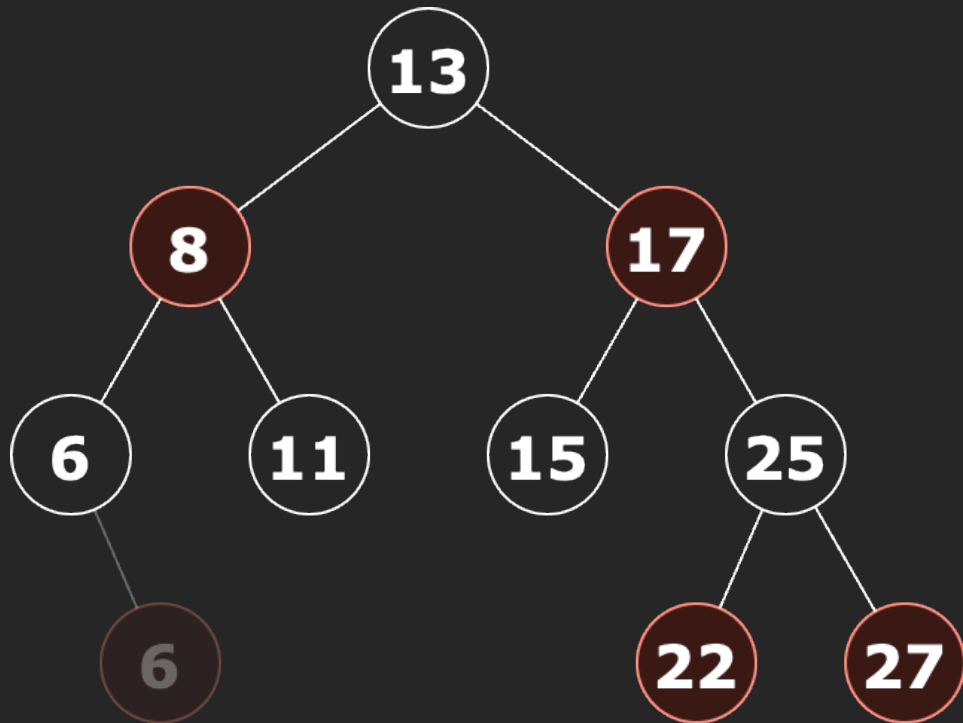
1. Не требует контроля соблюдения черной высоты
2. Может быть физически удален из дерева

# Черная вершина с **одним** ребенком



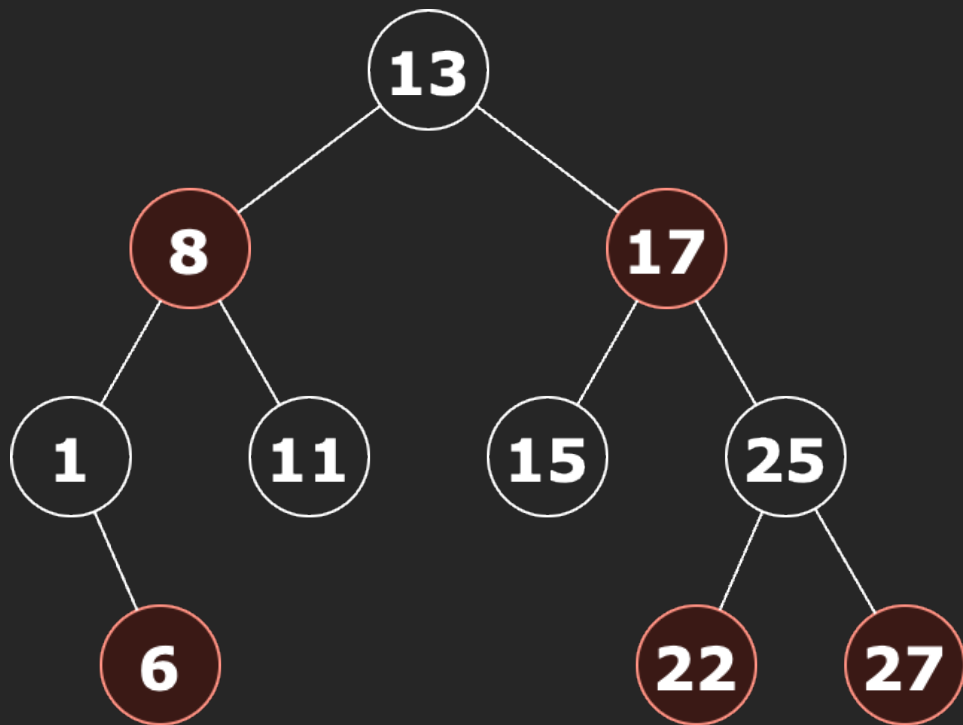
1. Ребенок может быть исключительно **красным**
2. Родитель принимает значение ребенка

# Черная вершина с **одним** ребенком



1. Ребенок может быть исключительно **красным**
2. Родитель принимает значение ребенка

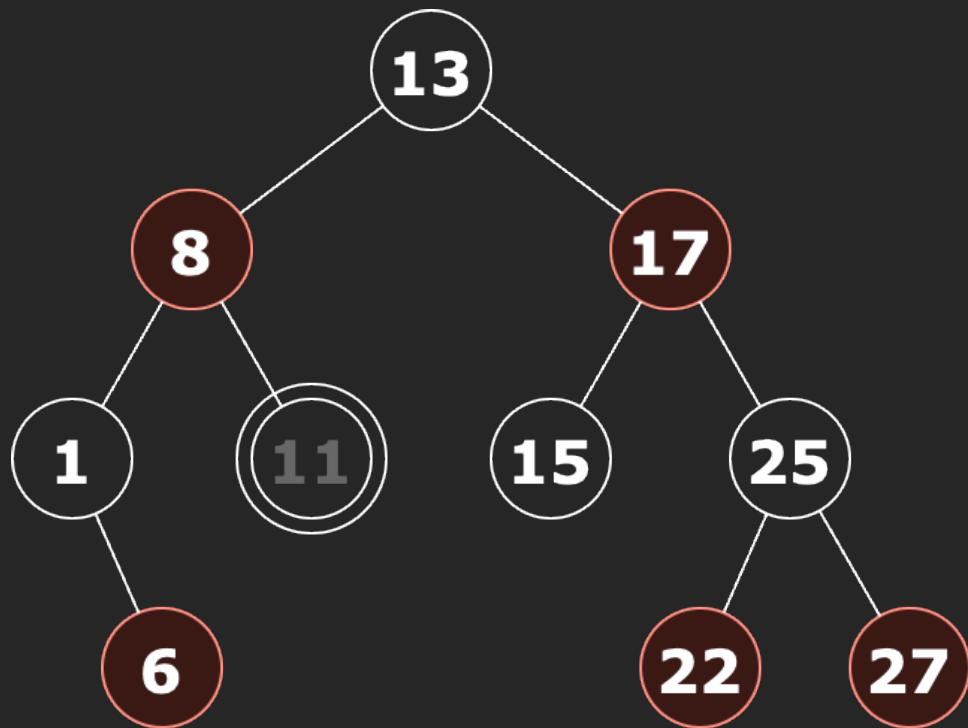
# Черный лист



1. Вершина принимает двойной черный цвет
2. Найти красную вершину *поблизости* для обмена цветов

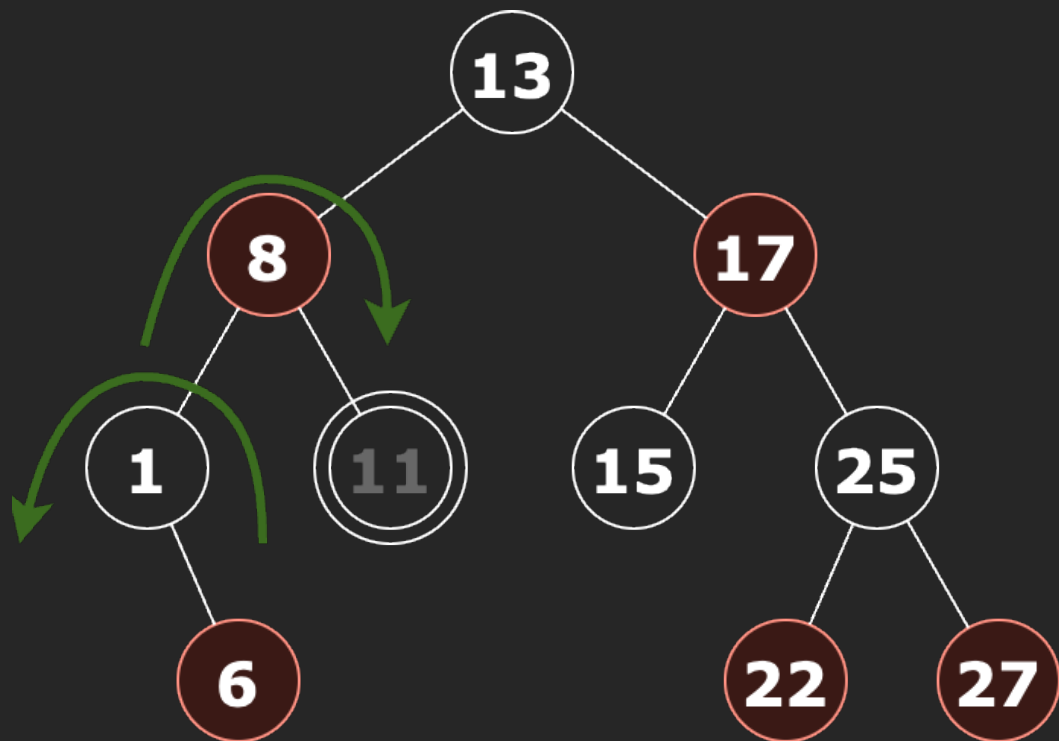


# Черный лист



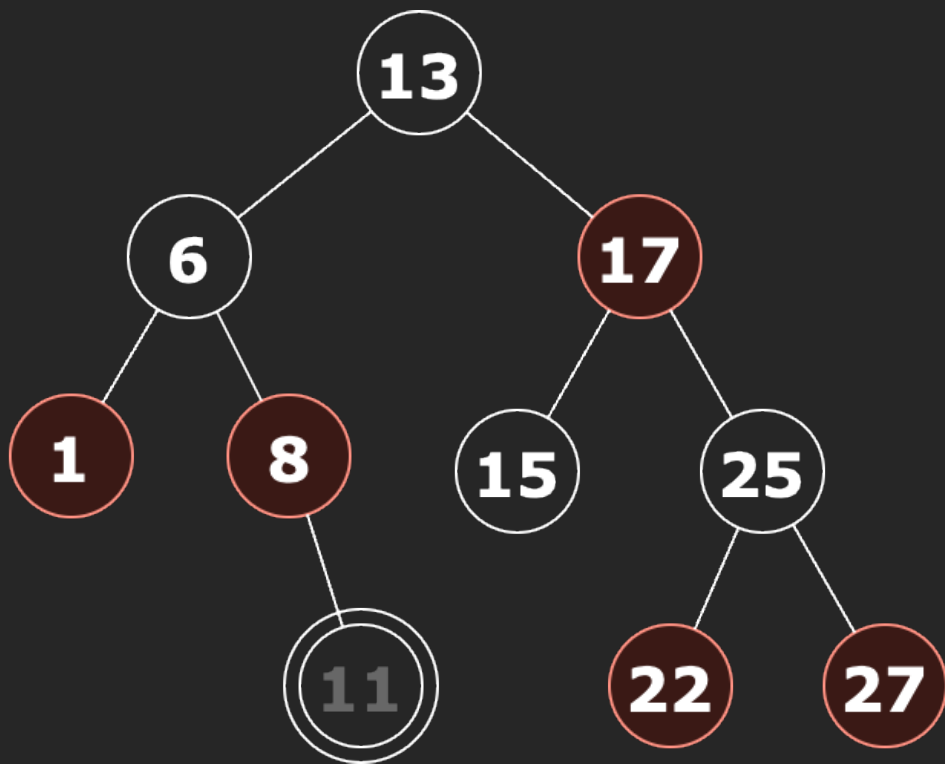
1. Вершина принимает двойной черный цвет
2. Найти красную вершину *поблизости* для обмена цветов

# Черный лист

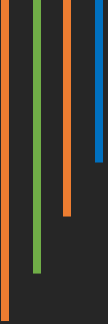


1. Вершина принимает двойной черный цвет
2. Найти красную вершину *поблизости* для обмена цветов

# Черный лист



1. Вершина принимает двойной черный цвет
2. Найти красную вершину *поблизости* для обмена цветов



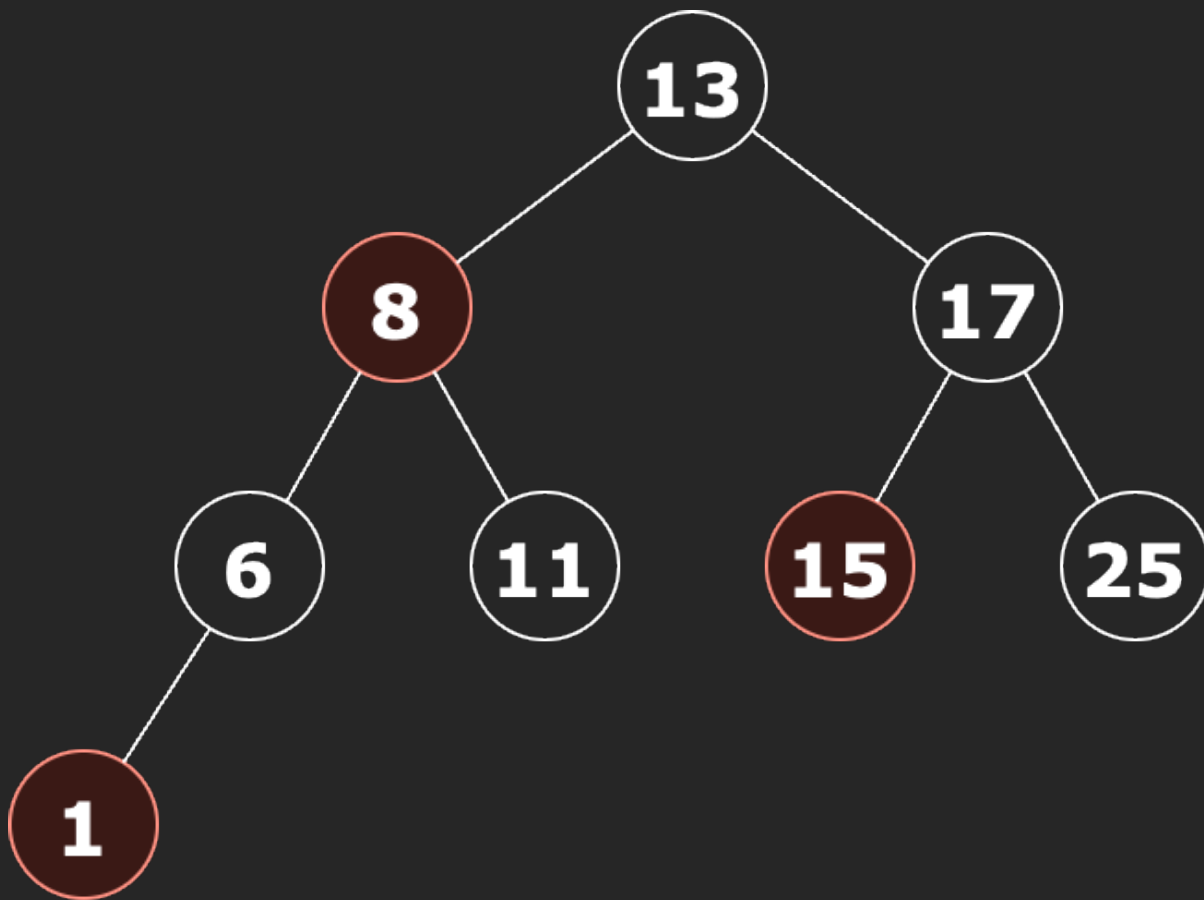
Всегда можно использовать изометрию  
с 2-3-4 деревом для поиска шагов  
к целевой конфигурации!

# Go beyond!

Left-Leaning Red-Black Tree

В 2008 году Роберт Седжвик предложил вариант красно-черного дерева, в котором красные узлы разрешаются только в качестве левых потомков черных вершин

# Go beyond!



## Left-leaning Red-Black Trees

Robert Sedgwick  
Department of Computer Science  
Princeton University  
Princeton, NJ 08544

### Abstract

The red-black tree model for implementing balanced search trees, introduced by Guibas and Sedgwick thirty years ago, is now found throughout our computational infrastructure. Red-black trees are described in standard textbooks and are the underlying data structure for symbol-table implementations within C++, Java, Python, BSD Unix, and many other modern systems. However, many of these implementations have sacrificed some of the original design goals (primarily in order to develop an effective implementation of the delete operation, which was incompletely specified in the original paper), so a new look is worthwhile. In this paper, we describe a new variant of red-black trees that meets many of the original design goals and leads to substantially simpler code for insert/delete, less than one-fourth as much code as in implementations in common use.

All red-black trees are based on implementing 2-3 or 2-3-4 trees within a binary tree, using red links to bind together internal nodes into 3-nodes or 4-nodes. The new code is based on combining three ideas:

- Use a recursive implementation.
- Require that all 3-nodes lean left.
- Perform rotations on the way up the tree (after the recursive calls).

Not only do these ideas lead to simple code, but they also unify the algorithms: for example, the left-leaning versions of 2-3 trees and top-down 2-3-4 trees differ in the position of one line of code.

All of the red-black tree algorithms that have been proposed are characterized by a worst-case search time bounded by a small constant multiple of  $\lg N$  in a tree of  $N$  keys, and the behavior observed in practice is typically that same multiple faster than the worst-case bound, close to the optimal  $\lg N$  nodes examined that would be observed in a perfectly balanced tree. This performance is also conjectured (but not yet proved) for trees built from random keys, for all the major variants of red-black trees. Can we analyze average-case performance with random keys for this new, simpler version? This paper describes experimental results that shed light on the fascinating dynamic behavior of the growth of these trees. Specifically, in a left-leaning red-black 2-3 tree built from  $N$  random keys:

- A random successful search examines  $\lg N - 0.5$  nodes.
- The average tree height is about  $2 \ln N$  (!)
- The average size of left subtree exhibits log-oscillating behavior.

The development of a mathematical model explaining this behavior for random keys remains one of the outstanding problems in the analysis of algorithms.

From a practical standpoint, left-leaning red-black trees (LLRB trees) have a number of attractive characteristics:

- Experimental studies have not been able to distinguish these algorithms from optimal.
- They can be implemented by adding just a few lines of code to standard BST algorithms.
- Unlike hashing, they support ordered operations such as *select*, *rank*, and *range search*.

Thus, LLRB trees are useful for a broad variety of symbol-table applications and are prime candidates to serve as the basis for symbol tables in software libraries in the future.

# Временные затраты КЧД

---

1. Вставка и удаление ключей требуют дополнительное время для перекраски и поворотов
2. В большинстве случаев, вставка нового ключа потребует не более одного поворота

# Применение КЧД

---

Контейнеры C++ `std::set`, `std::map`,  
`std::multiset`, `std::multimap`

Контейнеры Java `java.util.TreeMap`,  
`java.util.TreeSet`

Ядро Linux `linux/rbtree.h`



# Ресар

---

Исчерпывающая характеристика вставки нового значения в красно-черное дерево

Основные конфигурации восходят к базовым случаям исправления разбалансировки

# Teaser – Лекция 12

---

Саморасширяющееся *splay*-дерево. Подробнее об  
амортизационном анализе сложности

Итоги по сбалансированным деревьям поиска...