Алгоритмы и структуры данных-1
Лекция 9

Дата: 20.11.2023

Программная инженерия, 2 курс
2023-2024 учебный год

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

План

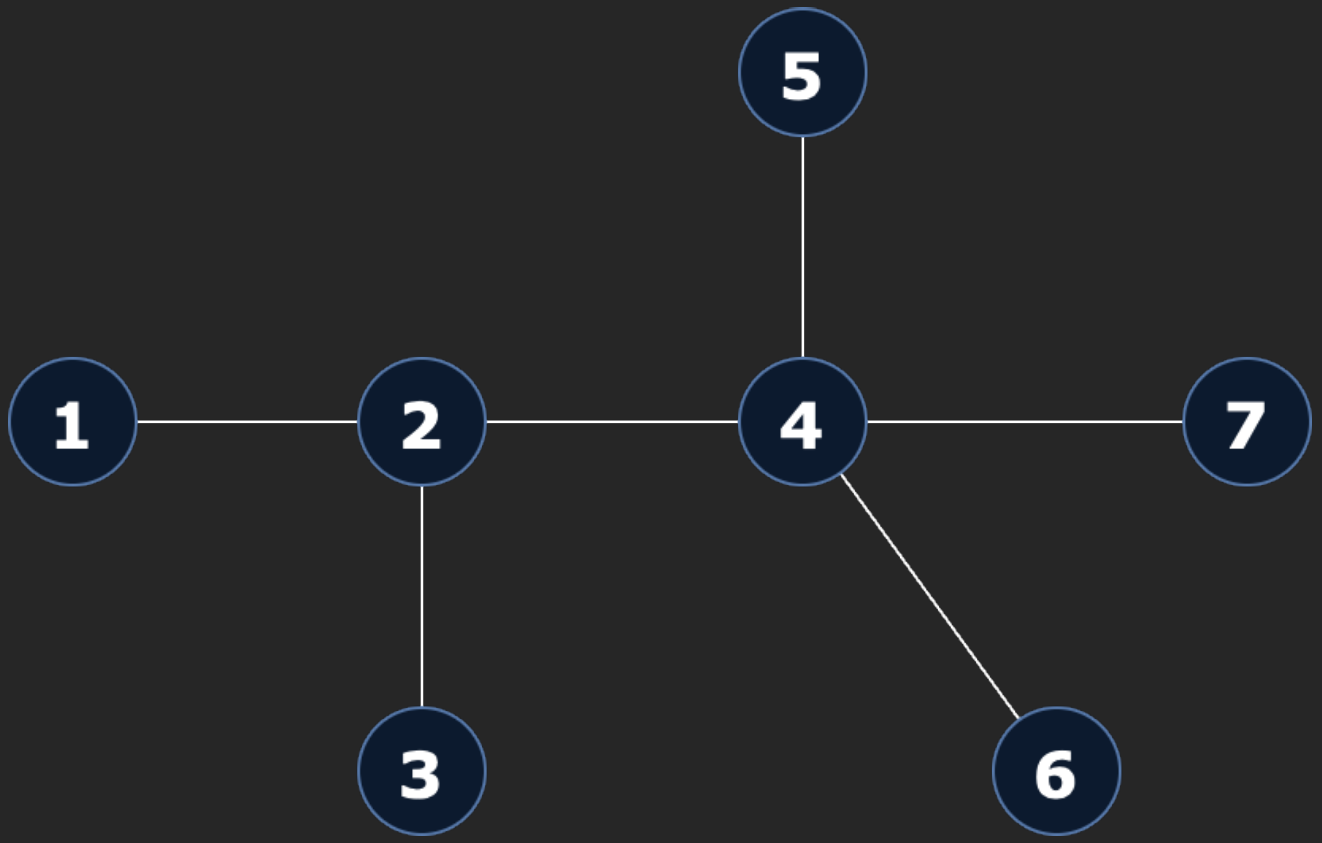
Бинарные деревья. Общие вопросы

Бинарные деревья поиска.
ADT «Отсортированный список»

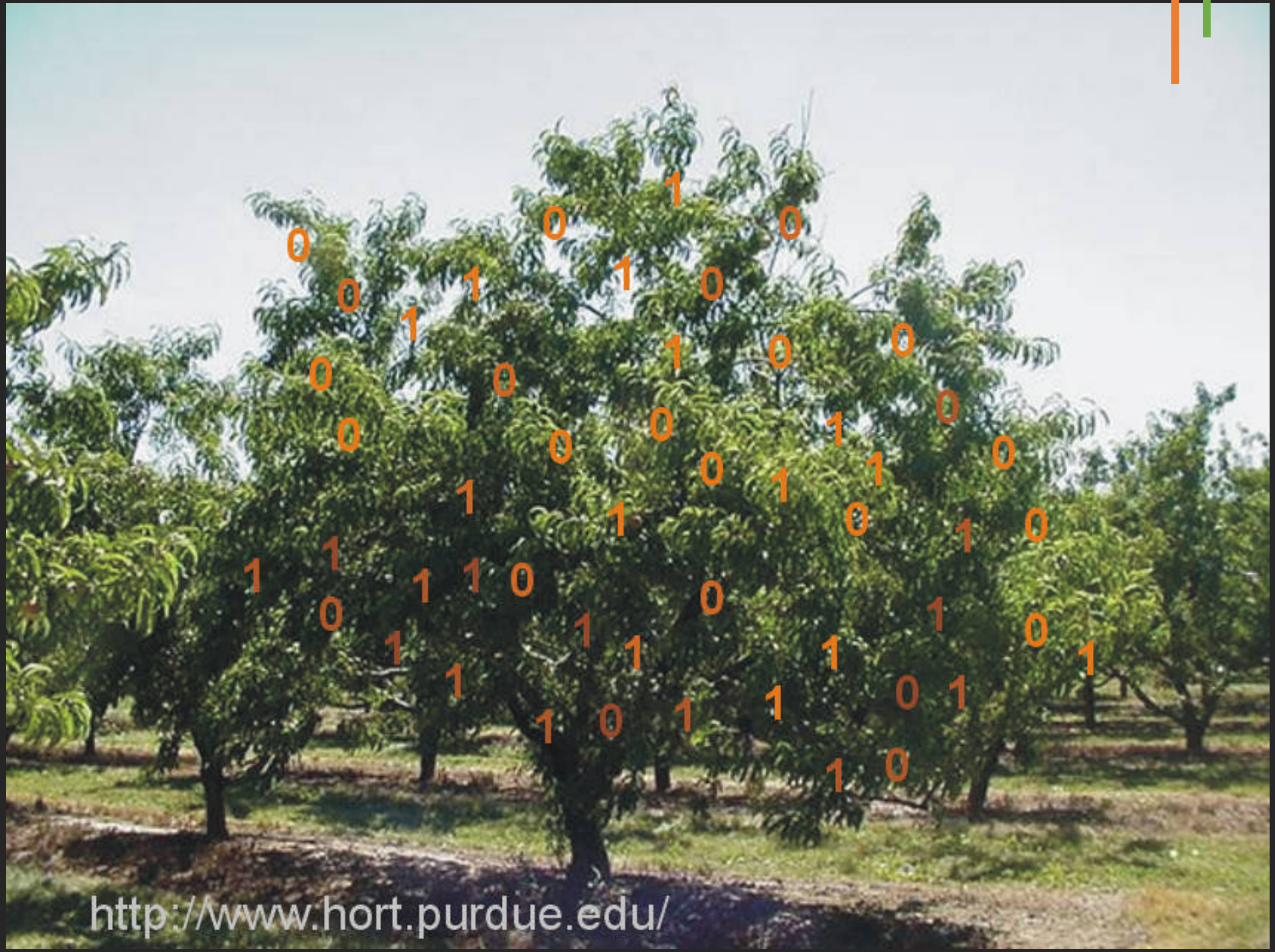
Вырождение бинарных деревьев поиска.
Проблема баланса

Дерево – это …

Односвязный граф без циклов



Это НЕ бинарное дерево…



Бинарное дерево

Произвольное количество вершин-потомков для простых деревьев редко встречается в реальных приложениях…

Бинарное дерево

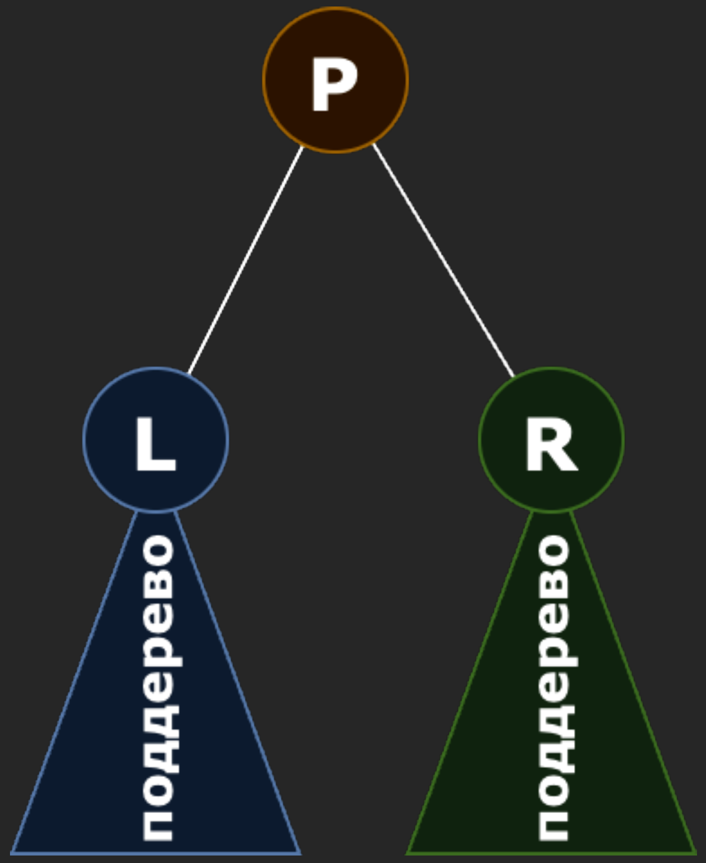
Произвольное количество вершин-потомков для простых деревьев редко встречается в реальных приложениях…

- разбор выражений с бинарными операторами
- алгоритмы кодирования без потерь
- генеалогические и филогенетические деревья

Бинарное дерево – рекурсивная структура

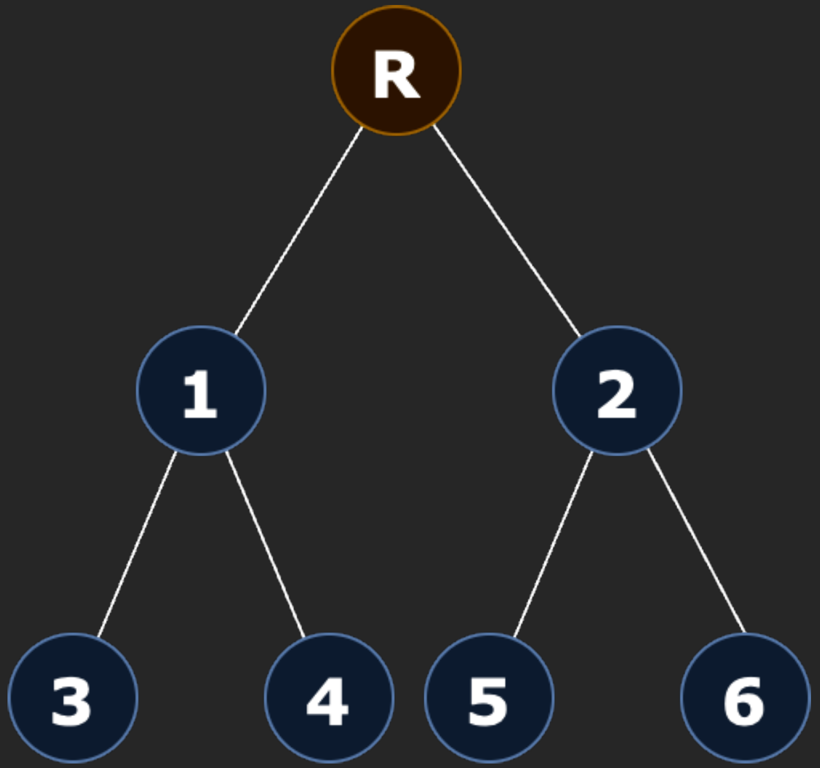
Каждая вершина имеет не более двух потомков (левый L и правый R)

Потомки могут определять целое поддерево

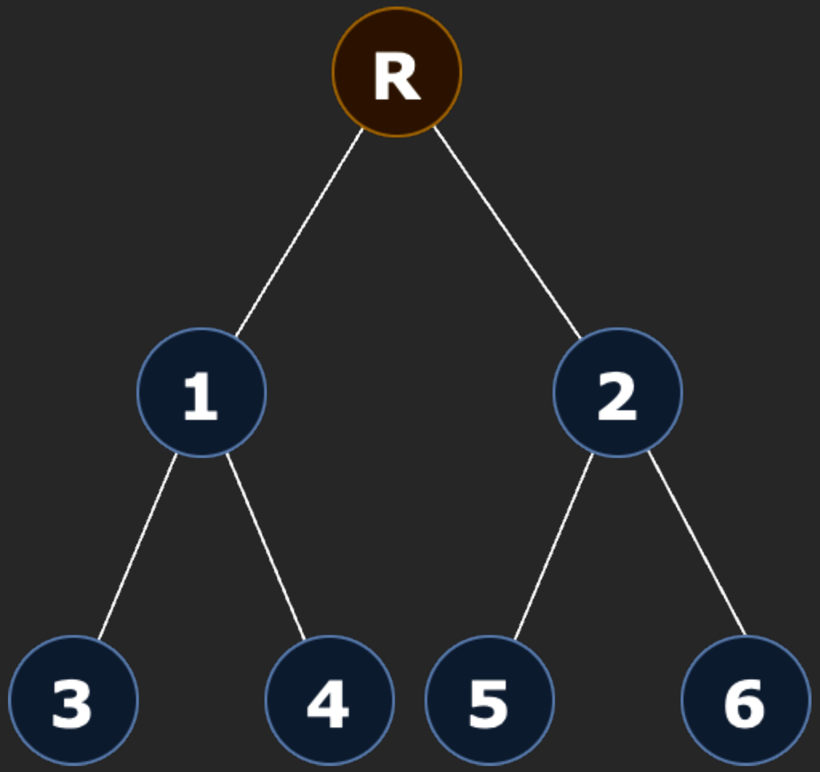


Классификация

Идеальные perfect бинарные деревья



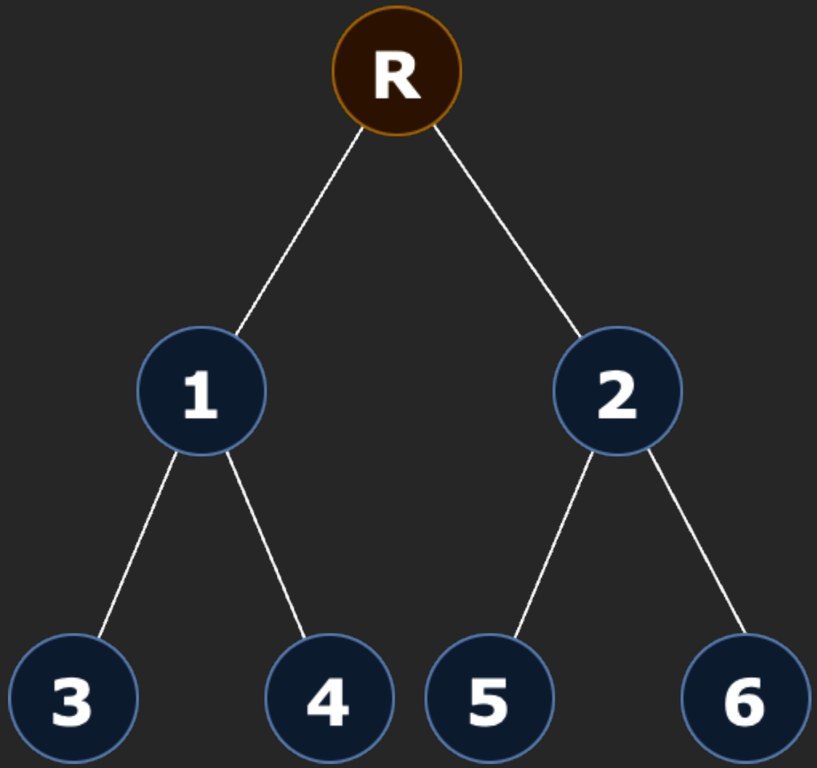
Идеальные perfect бинарные деревья



Все вершины, кроме листьев, имеют двух потомков

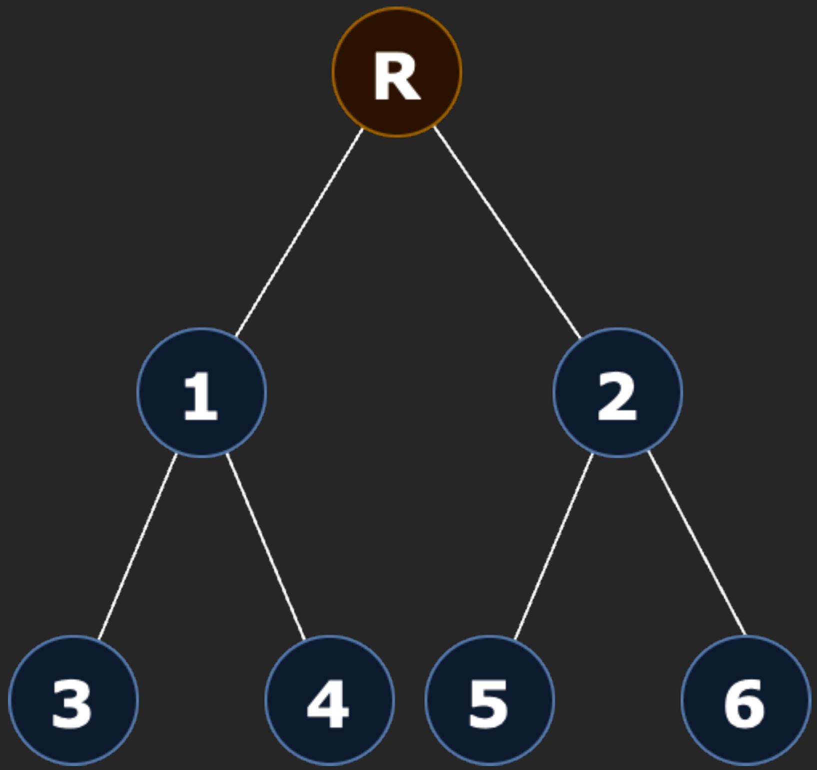
Все листья располагаются на одном уровне

Идеальные perfect бинарные деревья



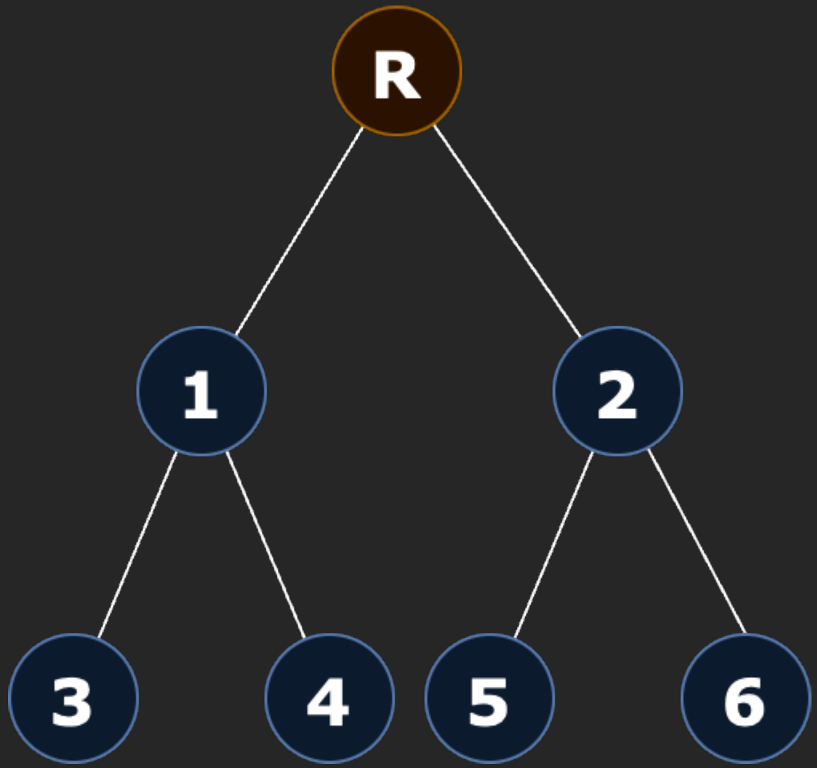
Высота идеального дерева с n вершинами – …

Идеальные perfect бинарные деревья



Высота идеального дерева с n вершинами – $\Theta(\log n)$

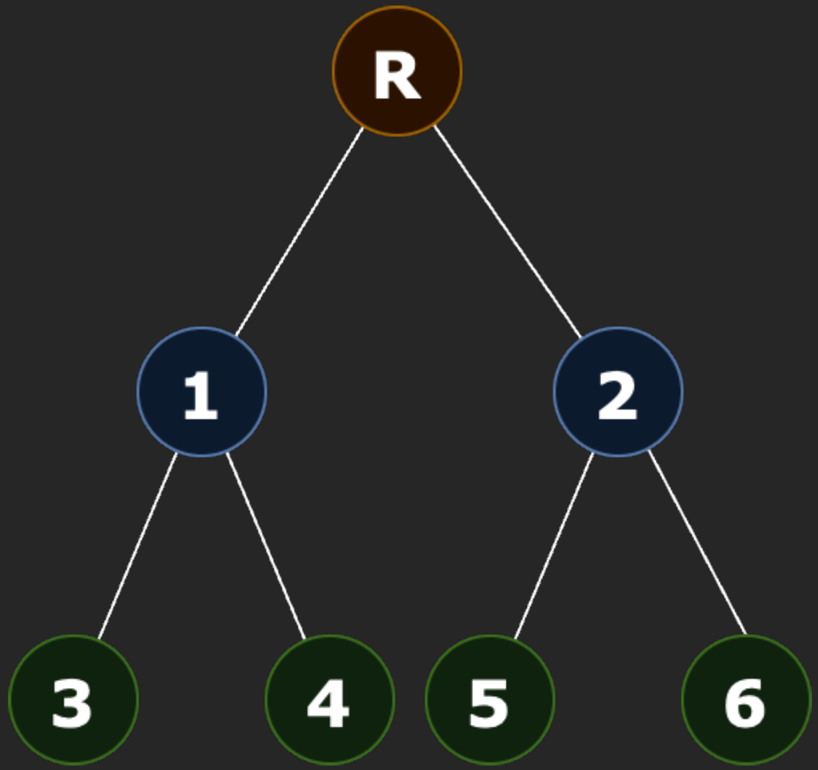
Идеальные perfect бинарные деревья



Высота идеального дерева с n вершинами – $\Theta(\log n)$

Идеальное дерево высоты h имеет $2^{h+1} - 1$ вершин

Идеальные perfect бинарные деревья

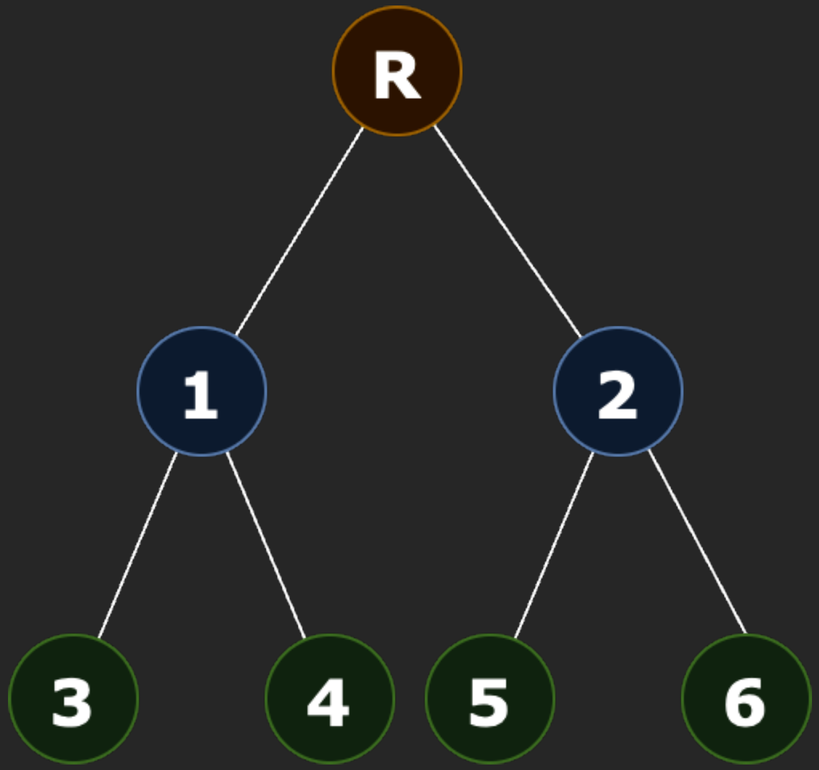


Высота идеального дерева с n вершинами – $\Theta(\log n)$

Идеальное дерево высоты h имеет $2^{h+1} - 1$ вершин

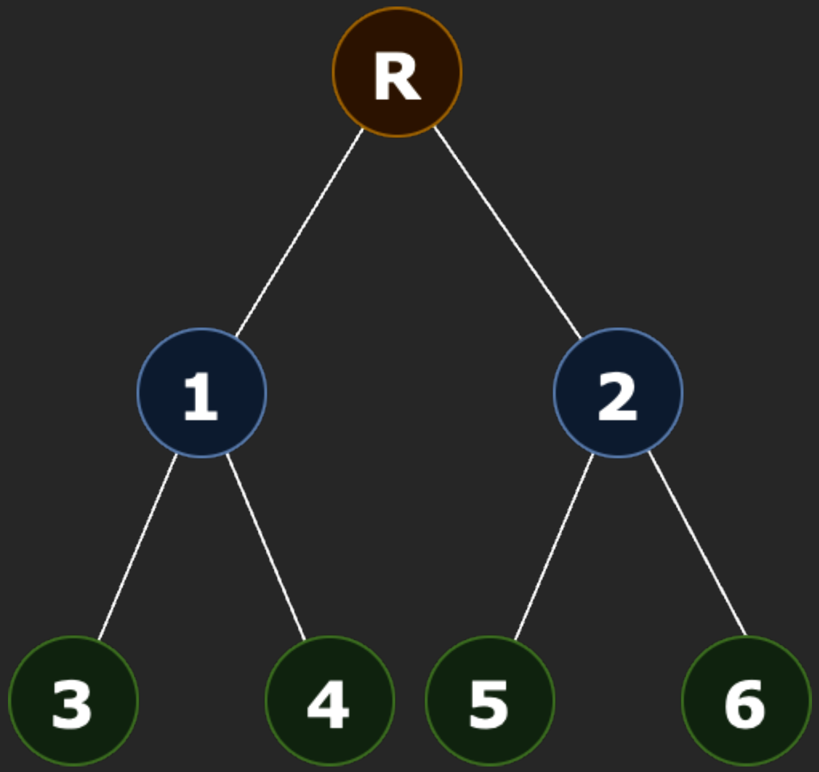
Идеальное дерево высоты h имеет 2^h листьев

Идеальные perfect бинарные деревья



Мы использовали идеальные деревья для анализа рекурсивных алгоритмов

Идеальные perfect бинарные деревья



Идеальные деревья использовались для анализа рекурсивных алгоритмов

А вообще, это то, к чему мы стремимся при работе с бинарными деревьями…

Полные complete бинарные деревья

Идеальное дерево имеет строго определенное количество вершин $n = 2^{h+1} - 1$ для $h = 0, 1, 2, 3, \dots$

$$1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots$$

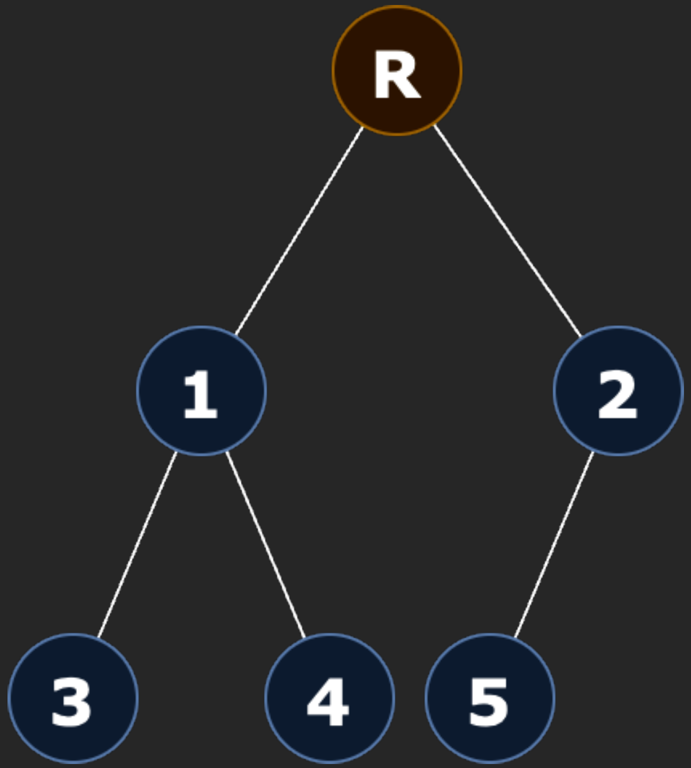
Полные complete бинарные деревья

Идеальное дерево имеет строго определенное количество вершин $n = 2^{h+1} - 1$ для $h = 0, 1, 2, 3, \dots$

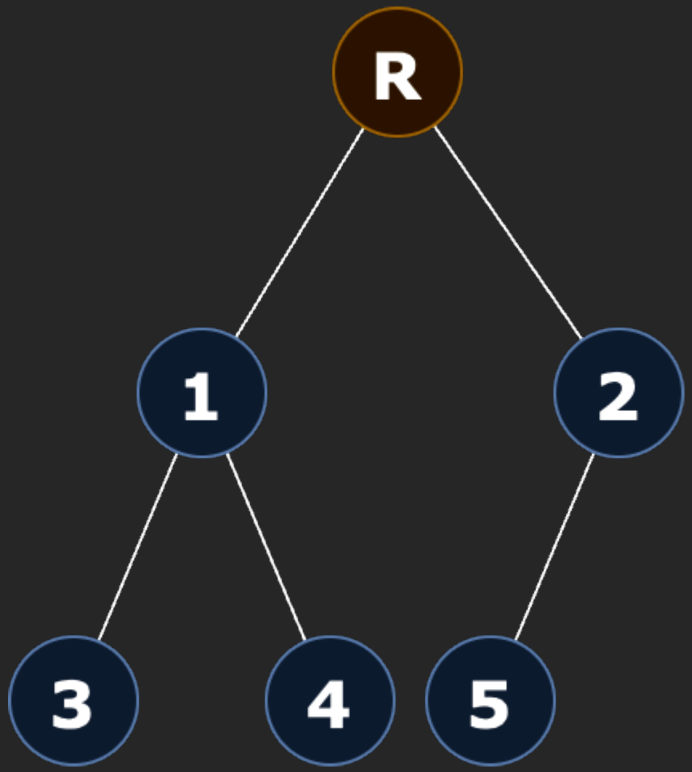
$$1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, \dots$$

Рассмотрим деревья, похожие на идеальные, но в которых количество вершин определено для всех n

Полные complete бинарные деревья



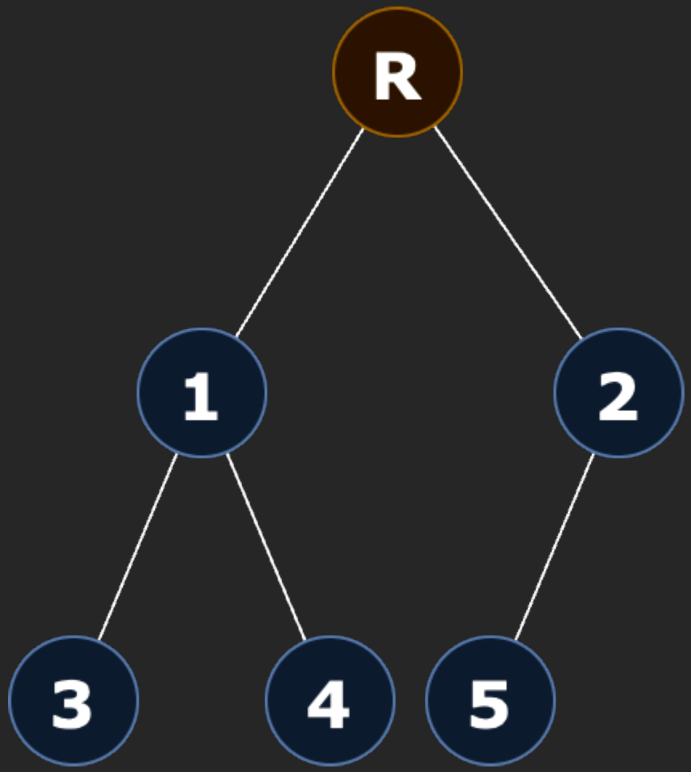
Полные complete бинарные деревья



Все уровни, кроме, м. б., последнего, заполнены

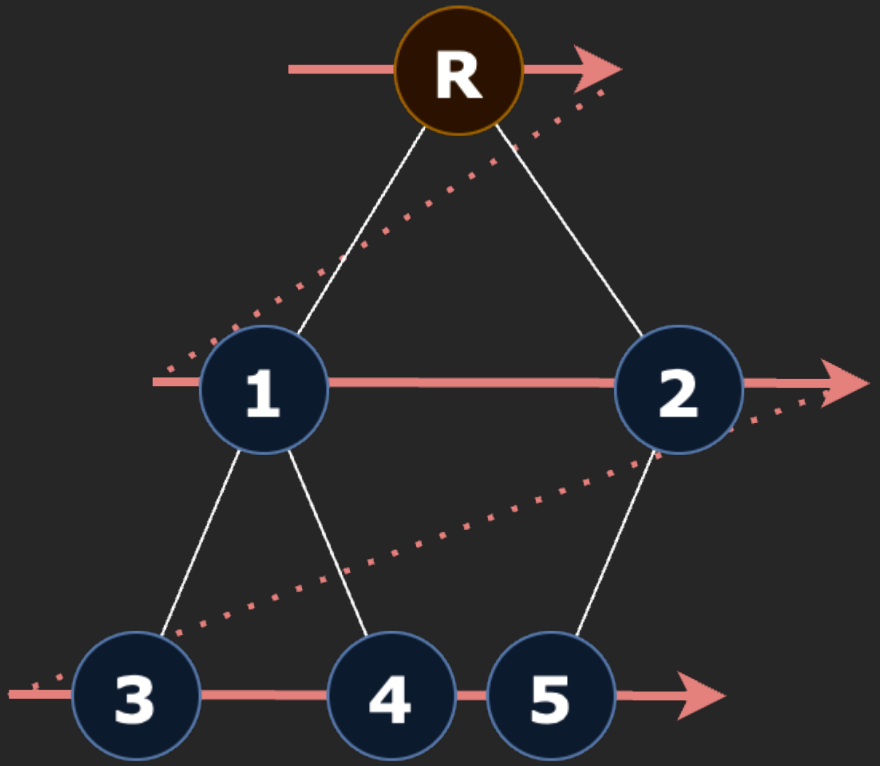
Заполнение уровней происходит слева направо

Полные complete бинарные деревья



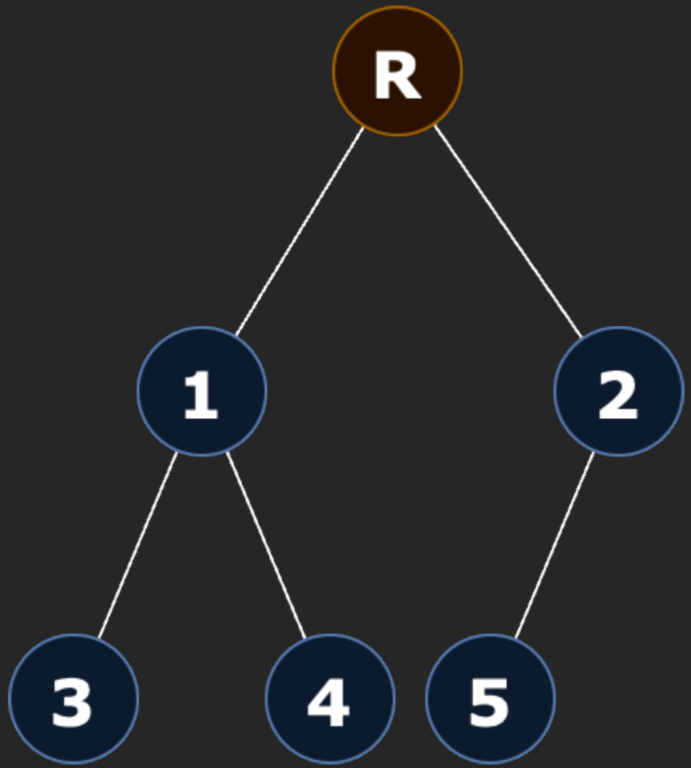
Высота полного дерева с n вершинами – $\lfloor \log n \rfloor$

Полные complete бинарные деревья



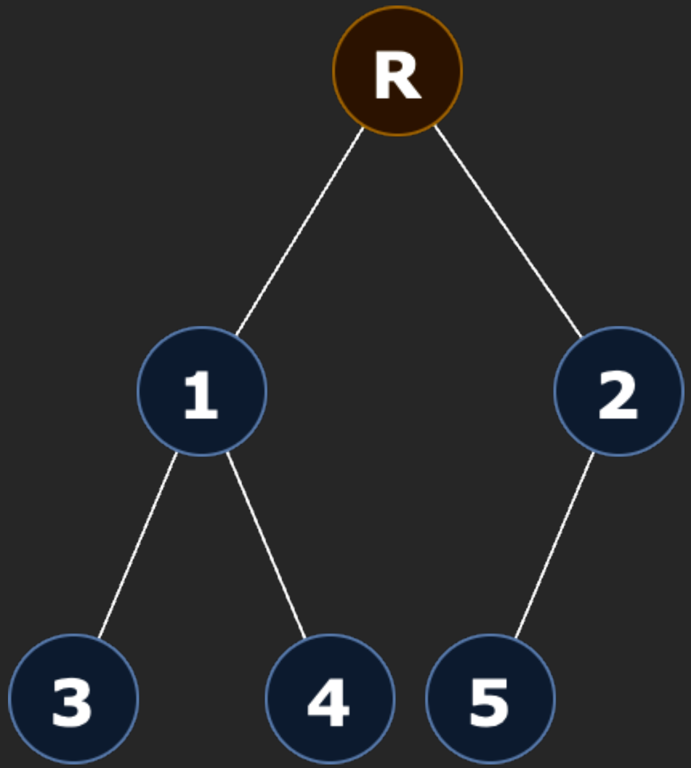
Такое дерево логичнее всего обрабатывать обходом в ширину

Полные complete бинарные деревья



Естественным образом индуцируется нумерация вершин по уровням, …

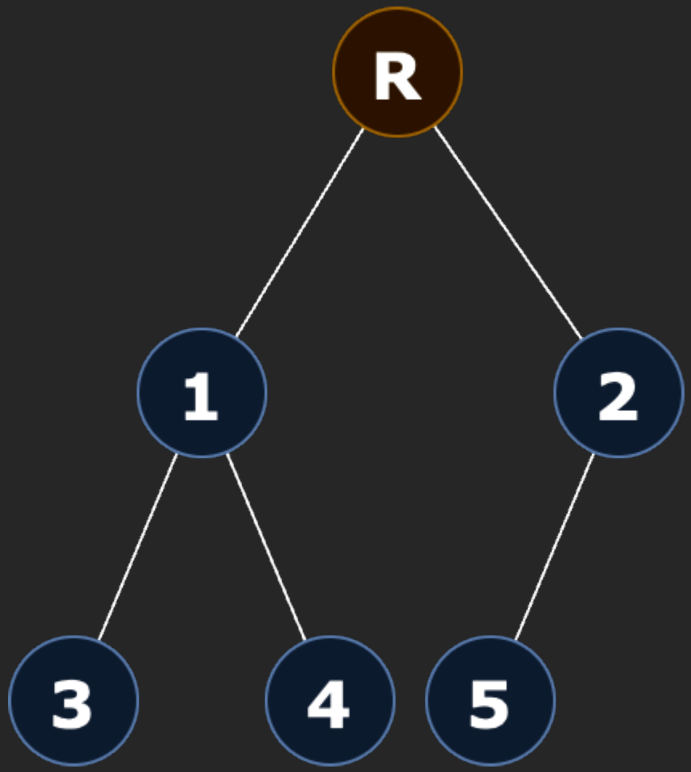
Полные (complete) бинарные деревья



Естественным образом индуцируется нумерация вершин по уровням, …

…что делает удобным хранение этого дерева в массиве

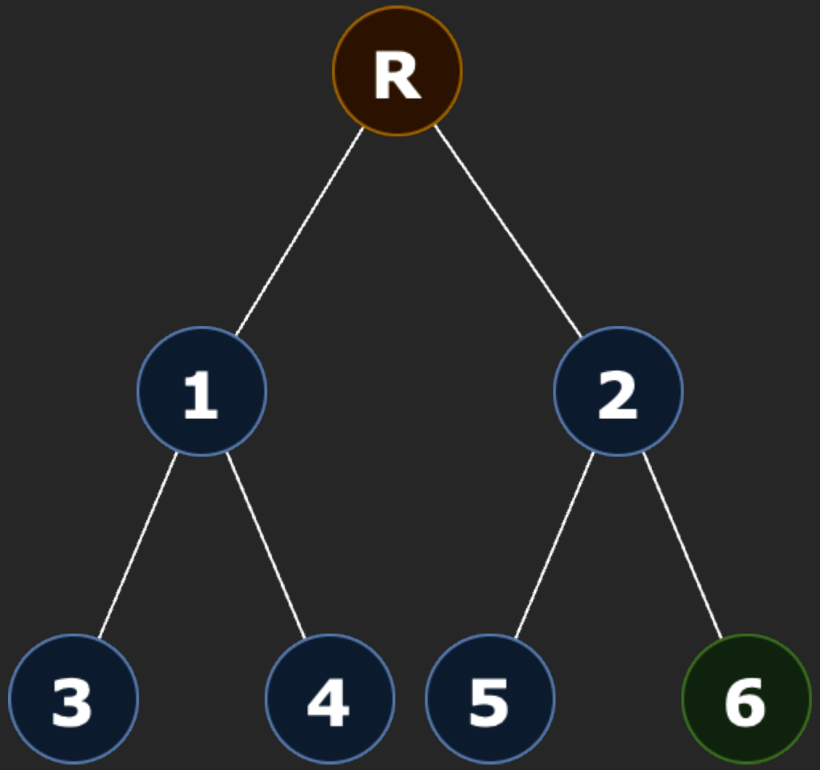
Полные (complete) бинарные деревья



	R	1	2	3	4	5	

Вставка сводится к помещению значения на первое свободное место

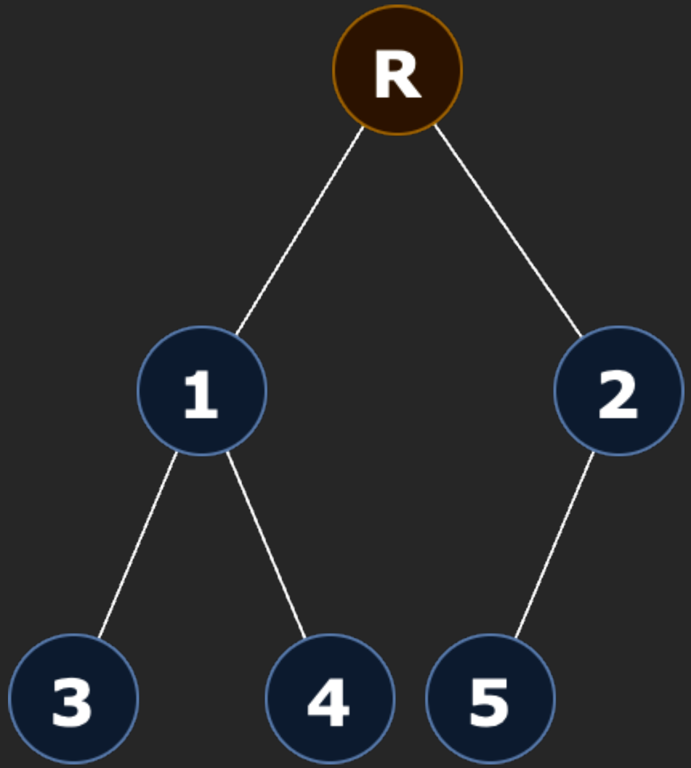
Полные (complete) бинарные деревья



	R	1	2	3	4	5	6

Вставка сводится к помещению значения на первое свободное место

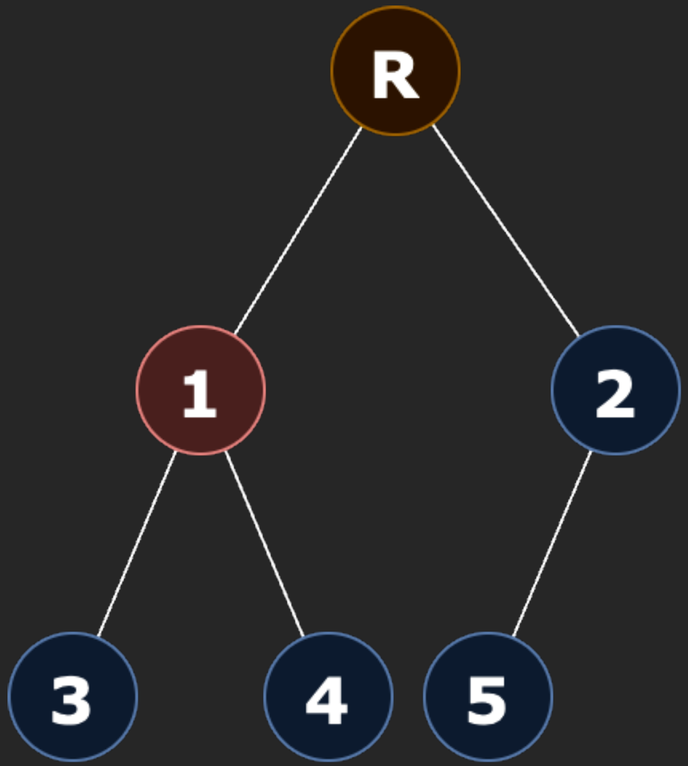
Полные (complete) бинарные деревья



	R	1	2	3	4	5	

Удаление сводится к обмену с последним значением в дереве

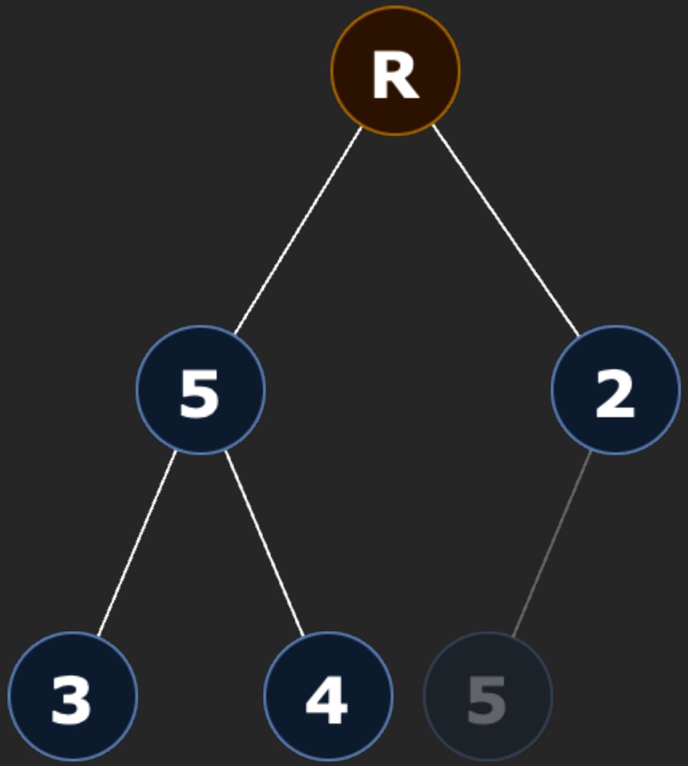
Полные (complete) бинарные деревья



	R	1	2	3	4	5	

Удаление сводится
к обмену с последним
значением в дереве

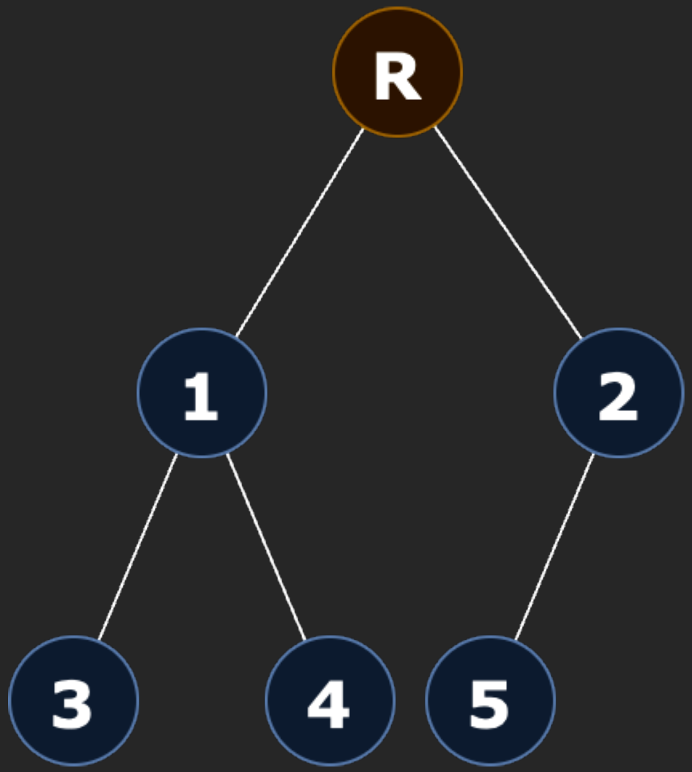
Полные (complete) бинарные деревья



	R	5	2	3	4		

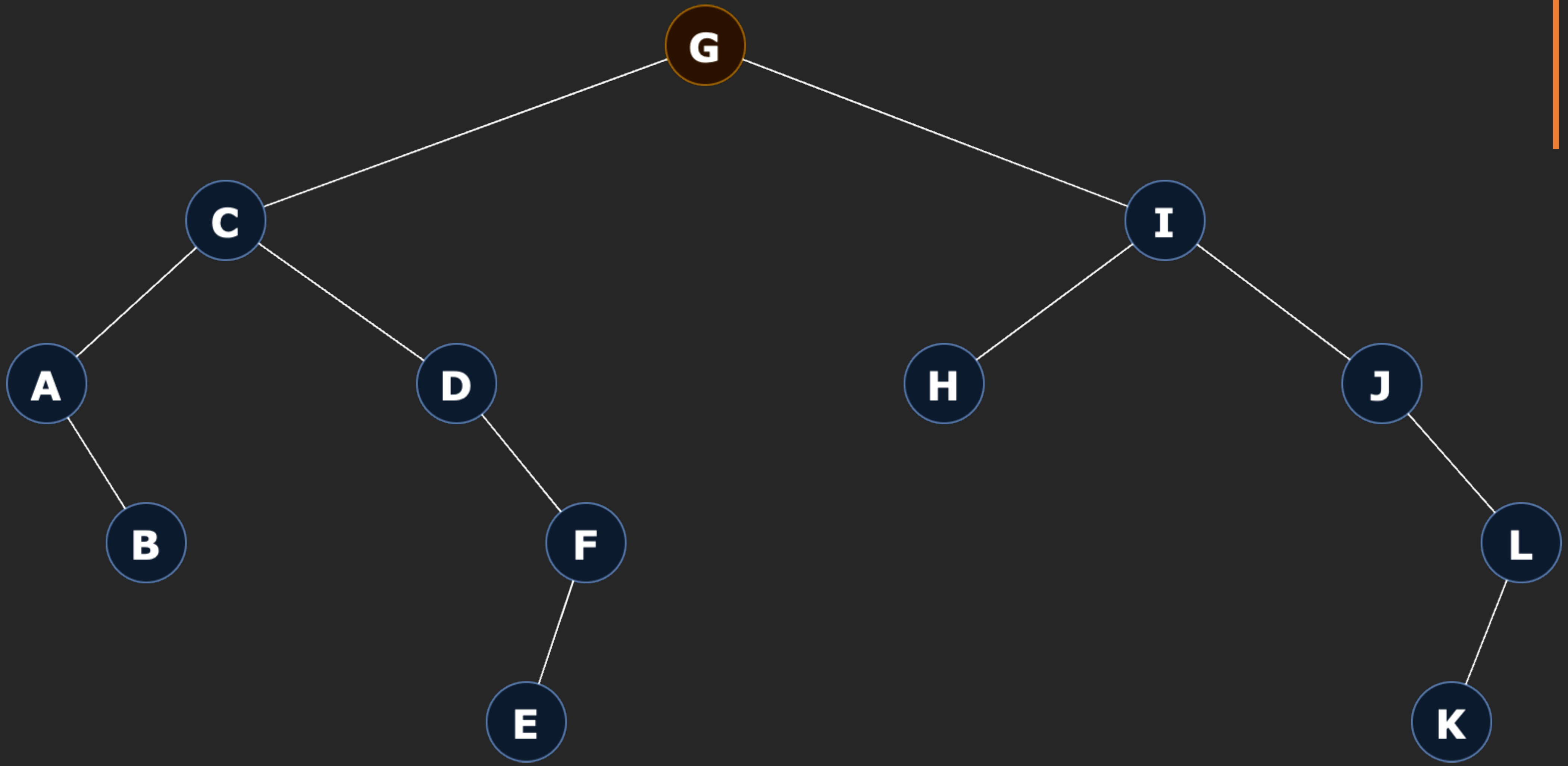
Удаление сводится к обмену с последним значением в дереве

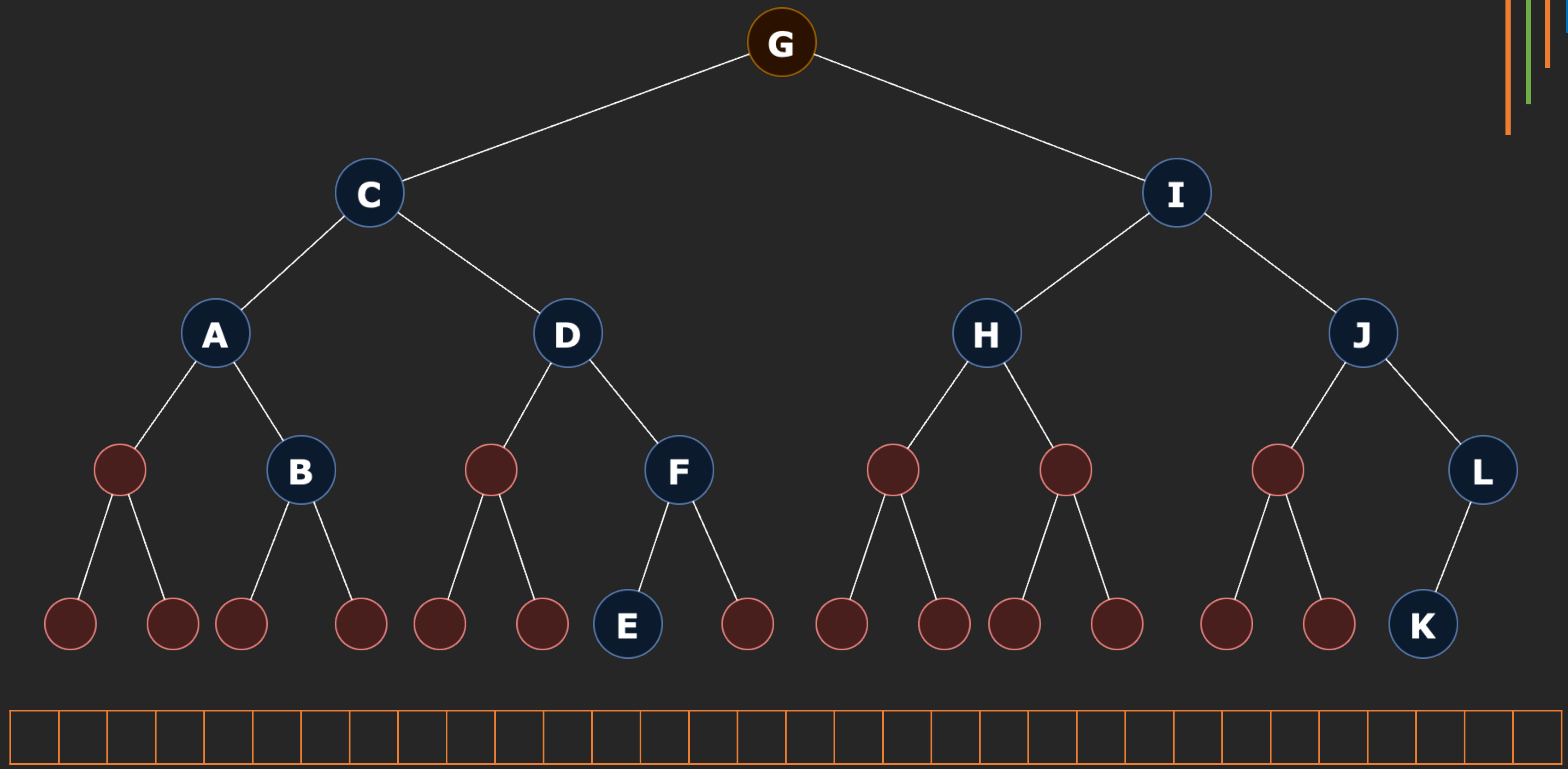
Полные (complete) бинарные деревья

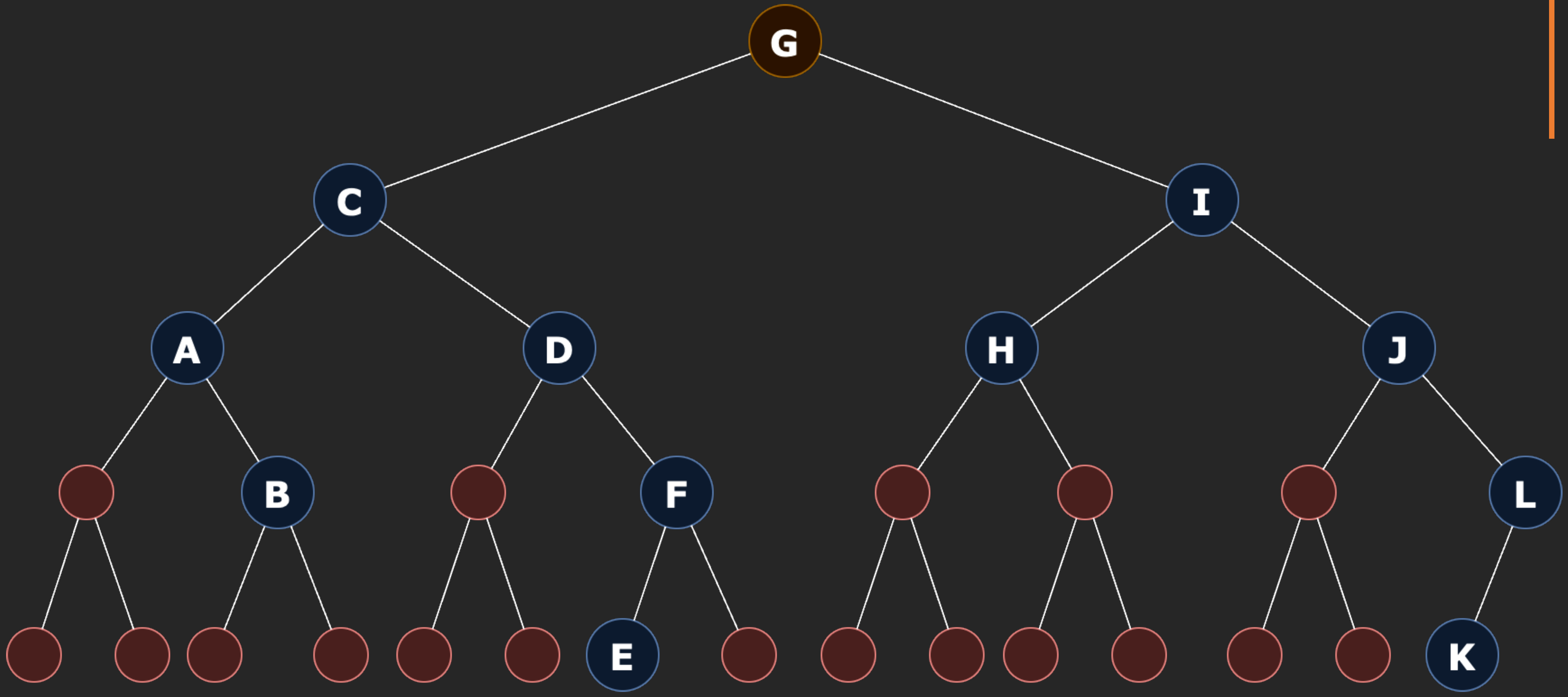


Куча является ярким примером полного бинарного дерева с доп. ограничениями

Почему бы не хранить любое бинарное дерево в массиве?

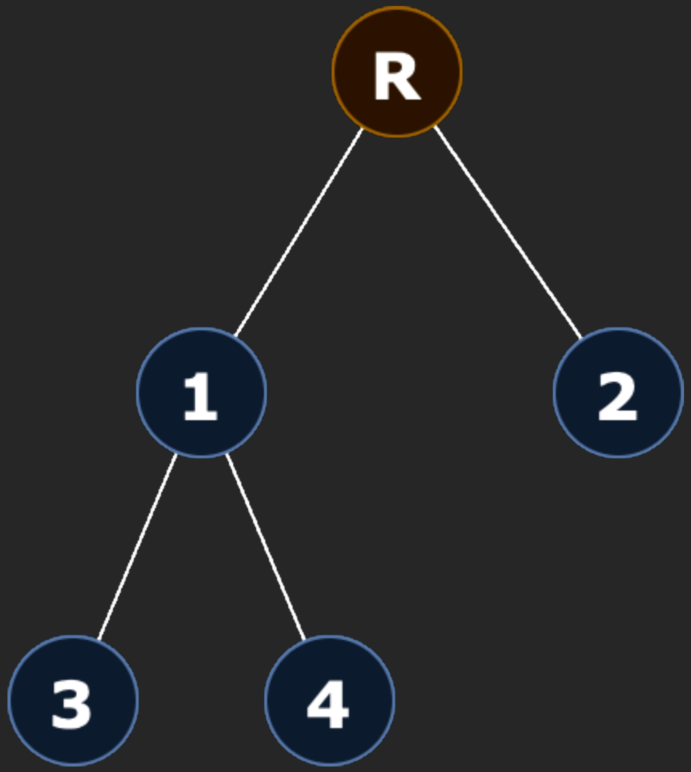

G
C
I
A
D
H
J
B
F
L
E
K


G
C
I
A
D
H
J
B
F
L
E
K

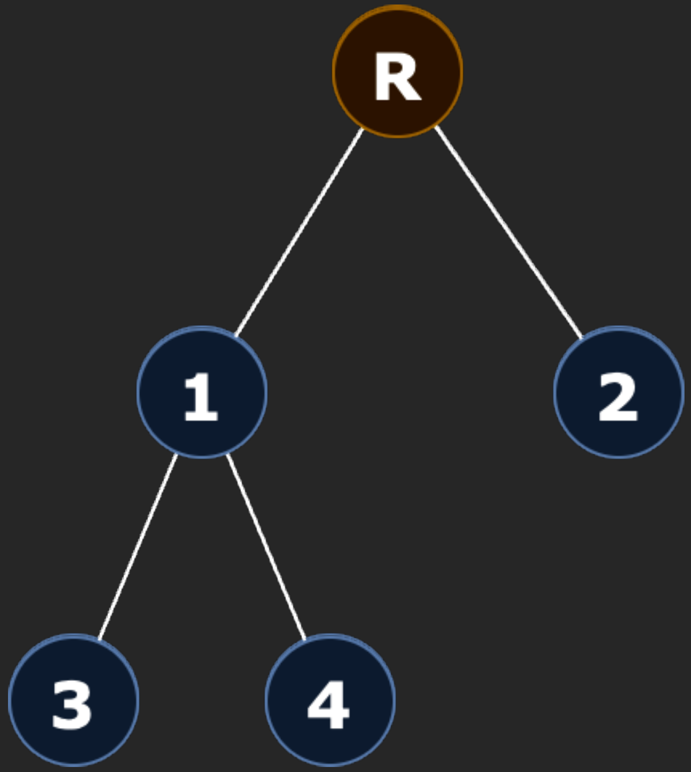


G	C	I	A	D	H	J		B		F				L							E								K		

Строгие full бинарные деревья

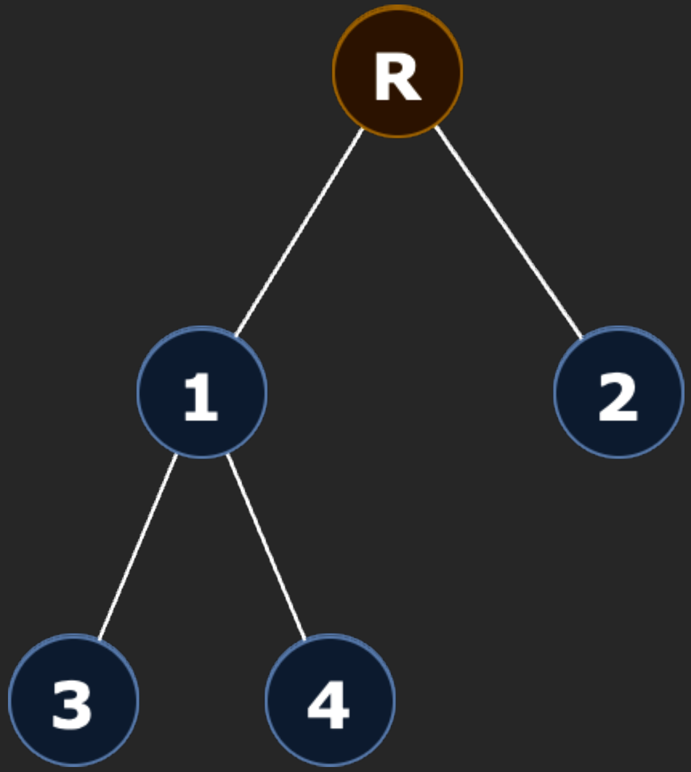


Строгие full бинарные деревья



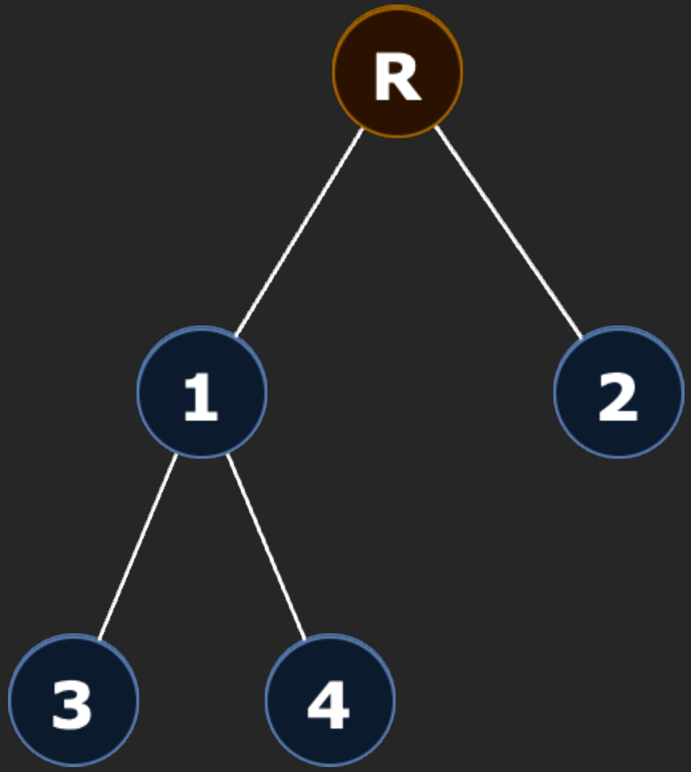
Каждая вершина, кроме листьев, имеет в точности по два потомка

Строгие full бинарные деревья



Пусть n – количество вершин с потомками, ...

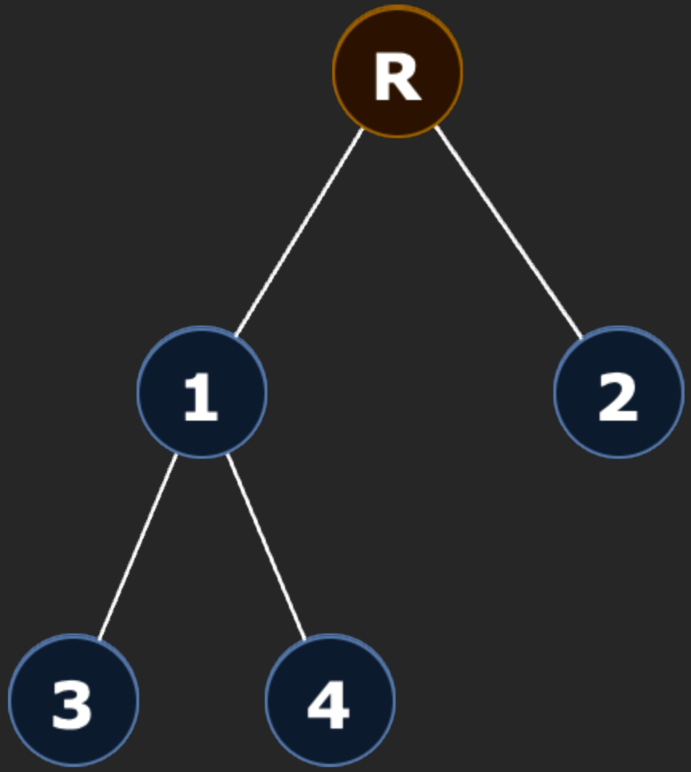
Строгие full бинарные деревья



Пусть n – количество вершин с потомками, …

тогда количество вершин без потомков – $n+1$

Строгие full бинарные деревья



Строгие деревья находят свое применение в

- кодировании Хаффмана
- синтаксическом разборе выражений

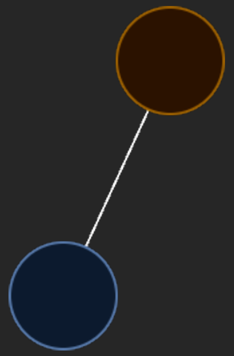
Синтаксический разбор выражений

Дерево выражения

(	a	+	2	)	*	b	

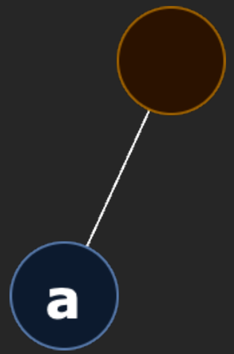
Дерево выражения

(	a	+	2	)	*	b	



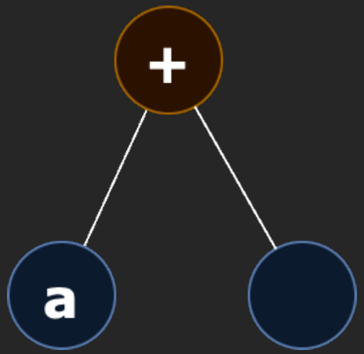
Дерево выражения

(	a	+	2	)	*	b	



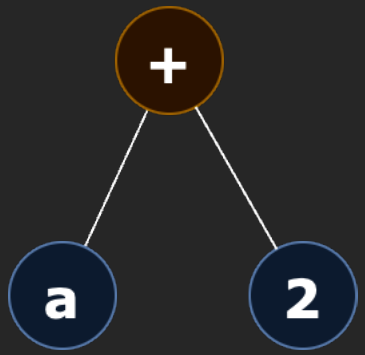
Дерево выражения

(	a	+	2	)	*	b	



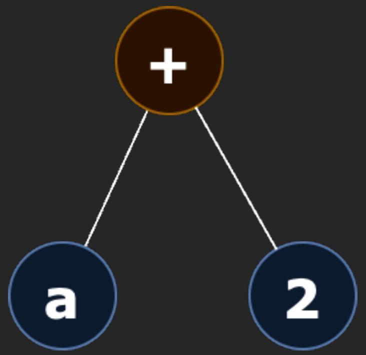
Дерево выражения

(	a	+	2	)	*	b	



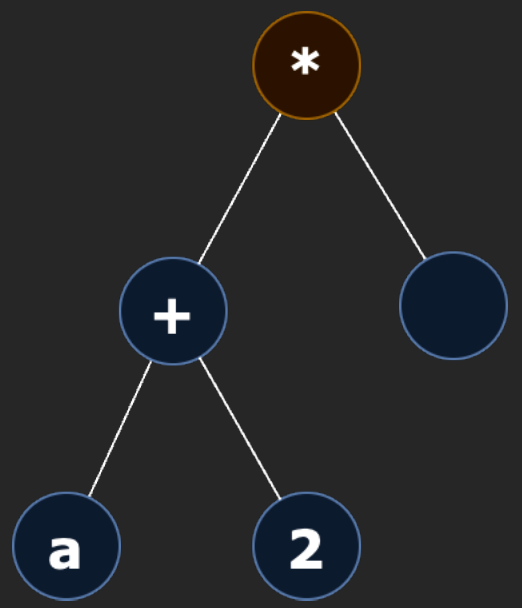
Дерево выражения

(	a	+	2	)	*	b	



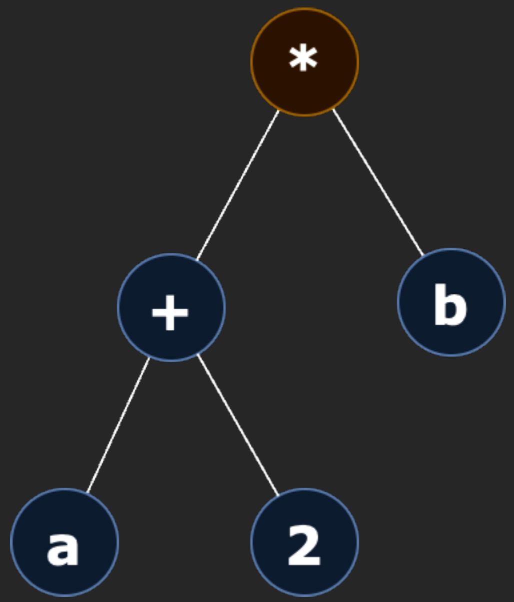
Дерево выражения

(	a	+	2	)	*	b	

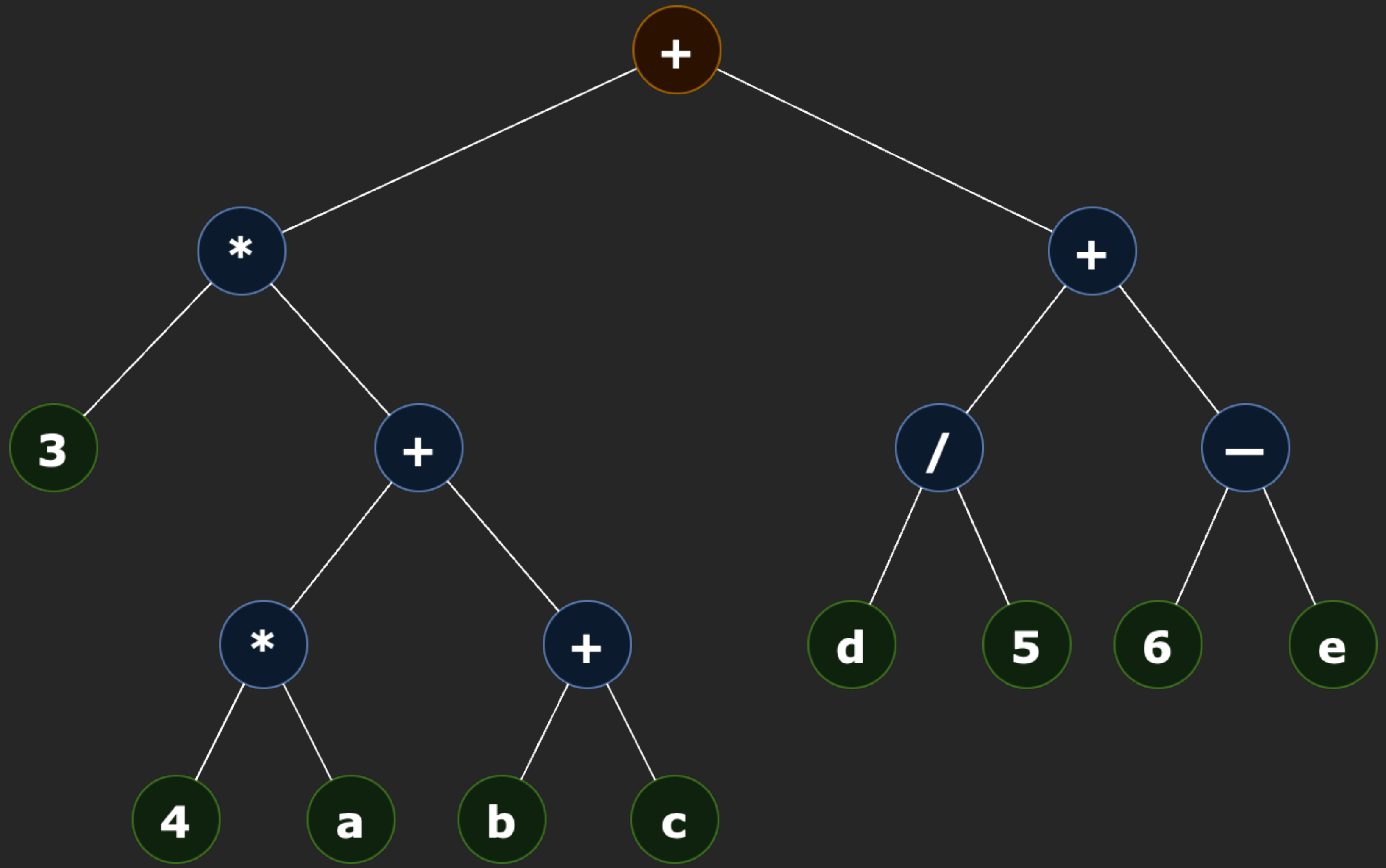


Дерево выражения

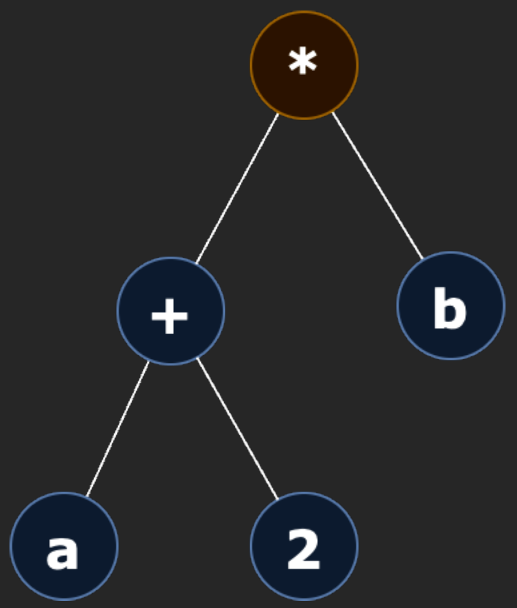
(	a	+	2	)	*	b	



Дерево выражения $3(4a + b + c) + {}^{d}/_{5} + (6 - e)$

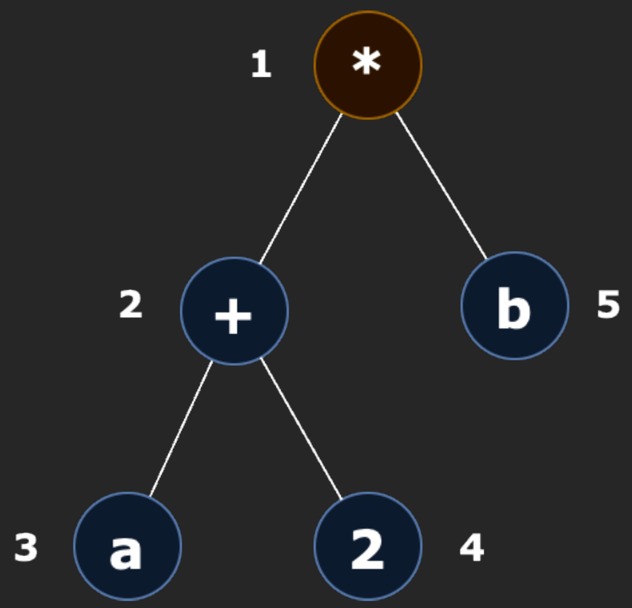


Дерево выражения. Прямой обход



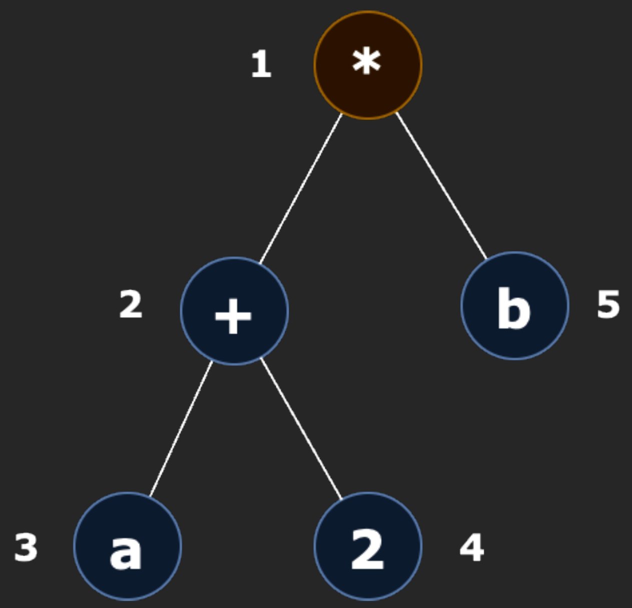

```
preOrder(Node* root)
if root != nullptr
    visit(root)
    preOrder(root->left)
    preOrder(root->right)
```

Дерево выражения. Прямой обход




```
preOrder(Node* root)
if root != nullptr
    visit(root)
    preOrder(root->left)
    preOrder(root->right)
```

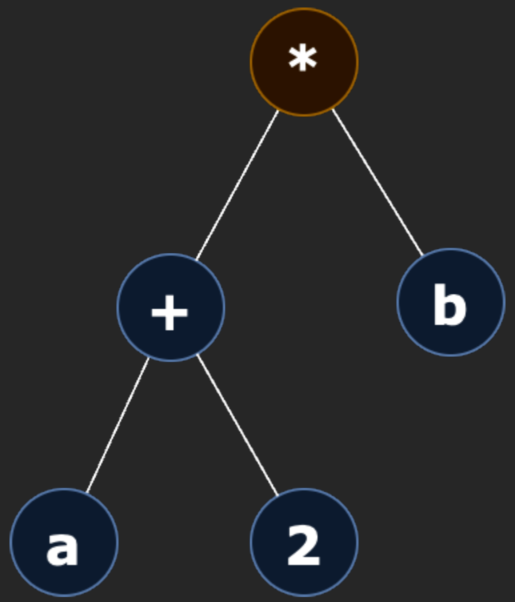
Дерево выражения. Прямой обход




```
preOrder(Node* root)
if root != nullptr
    visit(root)
    preOrder(root->left)
    preOrder(root->right)
```

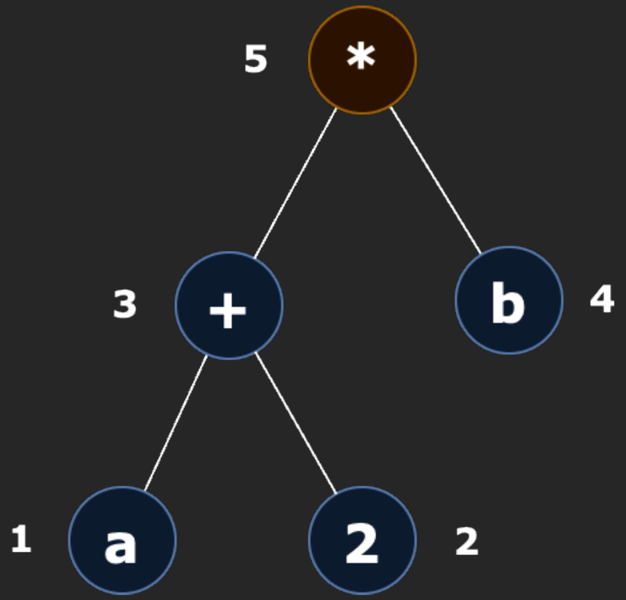
$$* + a\, 2\, b$$

Дерево выражения. Обратный обход



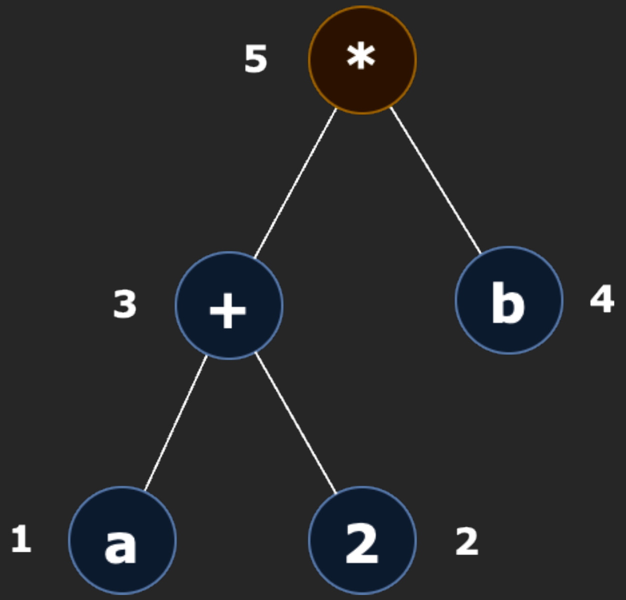

```
postOrder(Node* root)
if root != nullptr
    postOrder(root->left)
    postOrder(root->right)
    visit(root)
```

Дерево выражения. Обратный обход




```
postOrder(Node* root)
if root != nullptr
    postOrder(root->left)
    postOrder(root->right)
    visit(root)
```

Дерево выражения. Обратный обход

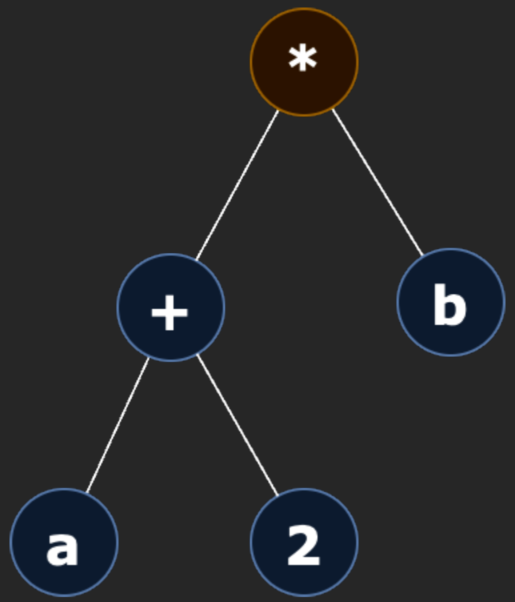



```
postOrder(Node* root)
if root != nullptr
    postOrder(root->left)
    postOrder(root->right)
    visit(root)
```

$$a\ 2 + b\ *$$

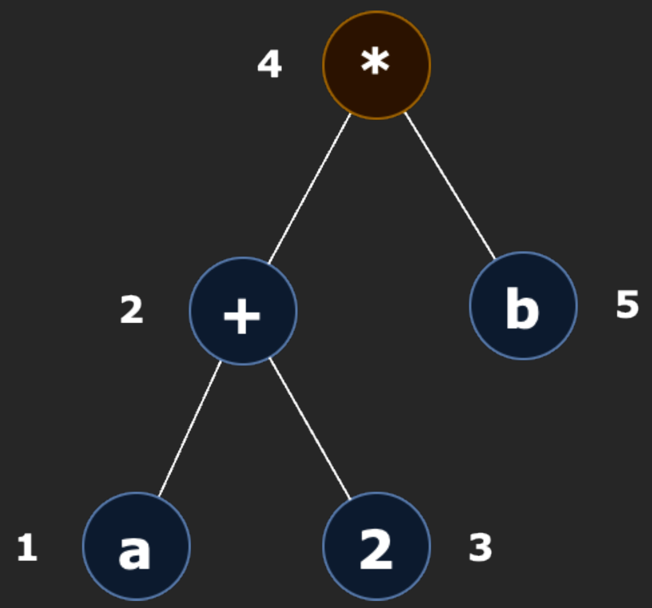
Префиксная и постфиксная запись не требует скобок

Дерево выражения. Симметричный обход



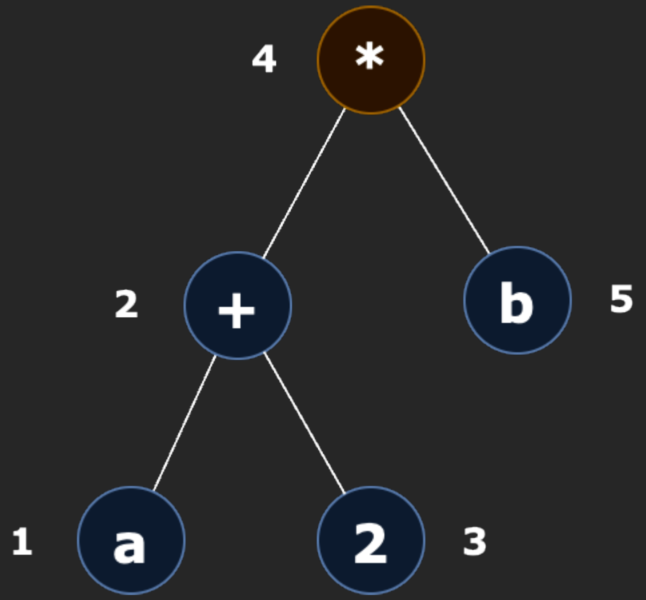

```
inOrder(Node* root)
if root != nullptr
    inOrder(root->left)
    visit(root)
    inOrder(root->right)
```

Дерево выражения. Симметричный обход




```
inOrder(Node* root)
if root != nullptr
    inOrder(root->left)
    visit(root)
    inOrder(root->right)
```

Дерево выражения. Симметричный обход




```
inOrder(Node* root)
if root != nullptr
    inOrder(root->left)
    visit(root)
    inOrder(root->right)
```

$$a + 2 * b$$

Нужны дополнительные действия для восстановления скобок

Обходы бинарного дерева

Три вариации поиска в глубину

Могут быть реализованы без рекурсии, но тогда потребуется хранить вершины дерева в стеке

Бинарное дерево поиска

ADT Отсортированный список

Ранее мы работали с реализациями ADT «Линейный контейнер», упорядочивание объектов в которых выполняется самим разработчиком

ADT Отсортированный список

Ранее мы работали с реализациями ADT «Линейный контейнер», упорядочивание объектов в которых выполняется самим разработчиком явно

В случае с ADT «Отсортированный список», объекты упорядочиваются **неявно**

ADT Отсортированный список

Операции `push_front`, `push_back` больше не имеют смысла

ADT Отсортированный список

Операции `push_front`, `push_back` больше не имеют смысла

Вместо них реализуется обобщенная операция вставки `insert`

ADT Отсортированный список

Поиск максимума и минимума

Поиск порядковых статистик

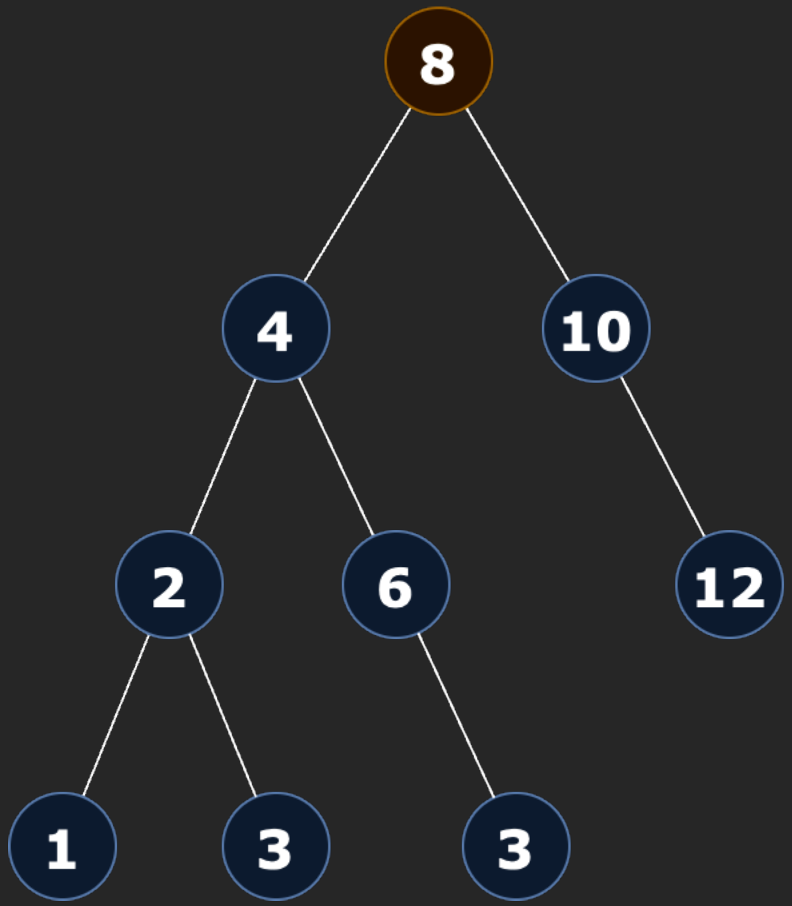
Поиск предыдущего и следующего значения

Итерация по объектам в заданном интервале $[a, b]$

…

В чем проблема реализации отсортированного списка на массиве?

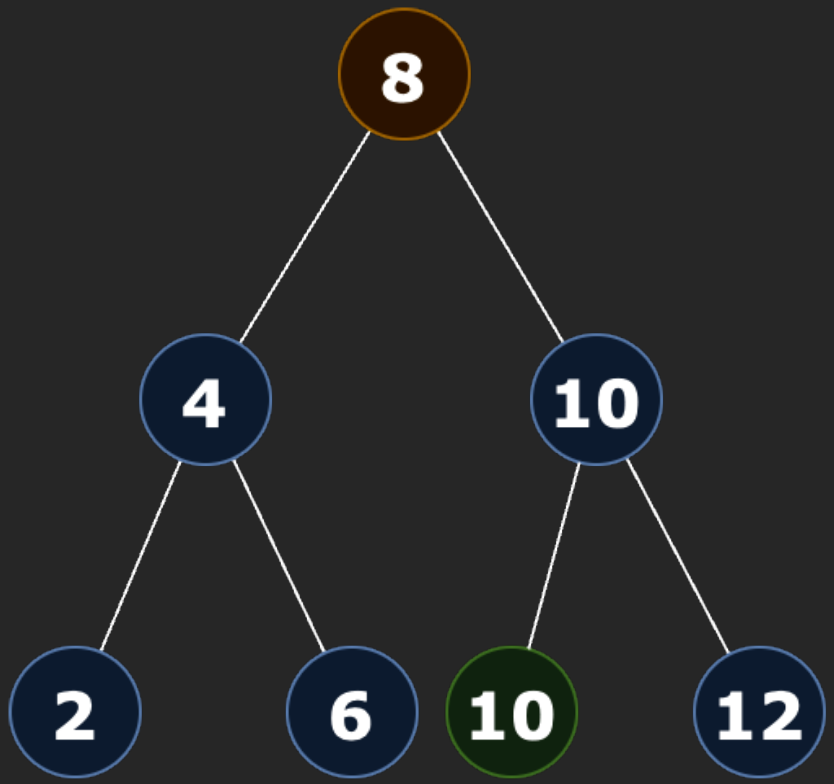
Бинарное дерево поиска – BST



Для любой вершины верно:

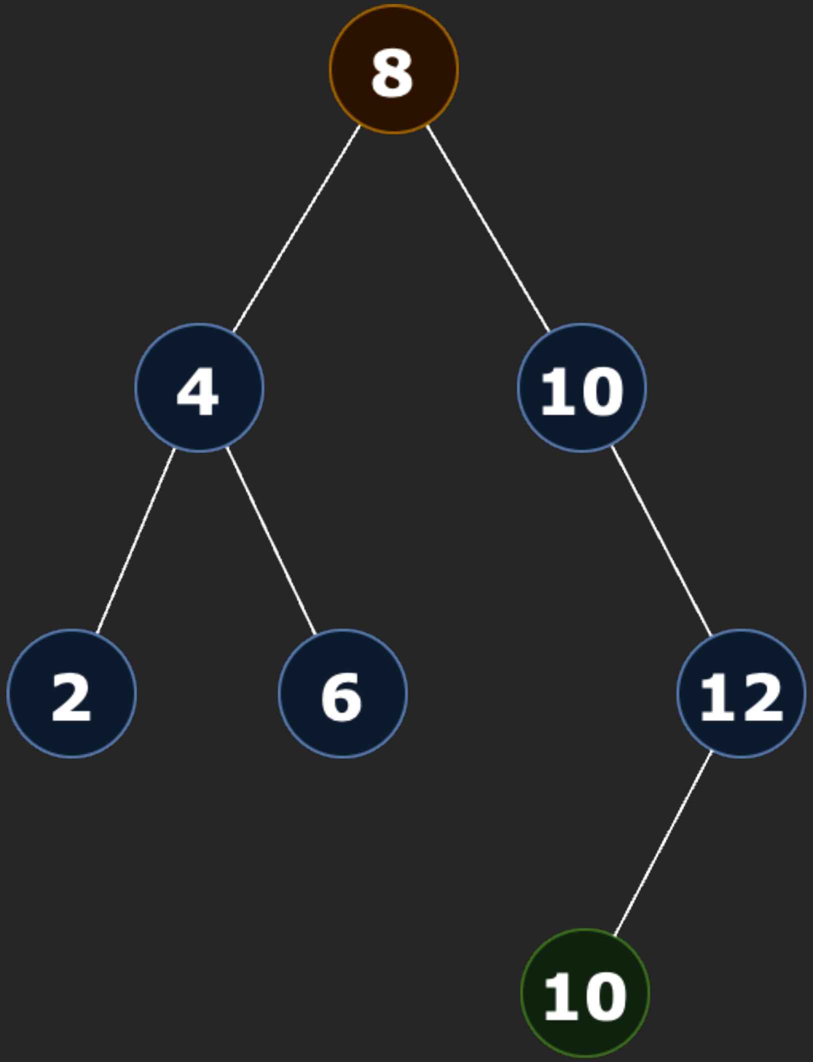
- ключи в левом поддереве меньше
- ключи в правом поддереве больше

BST и ключи-дубликаты



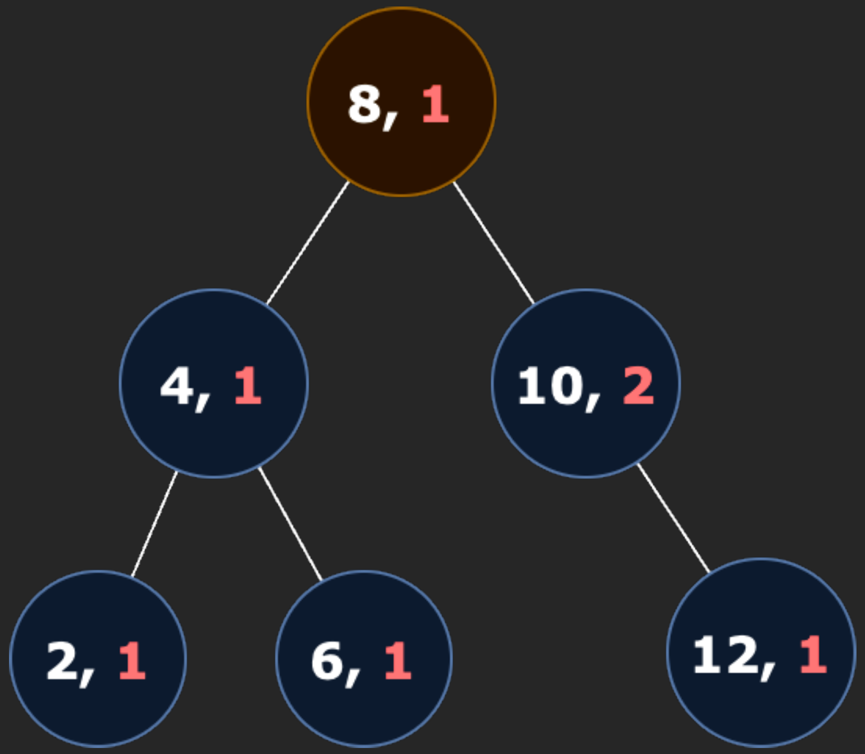
Делаем одно из условий сравнения нестрогим

BST и ключи-дубликаты



Делаем одно из условий сравнения нестрогим

BST и ключи-дубликаты



Дополнительно храним кратность для каждого ключа в дереве

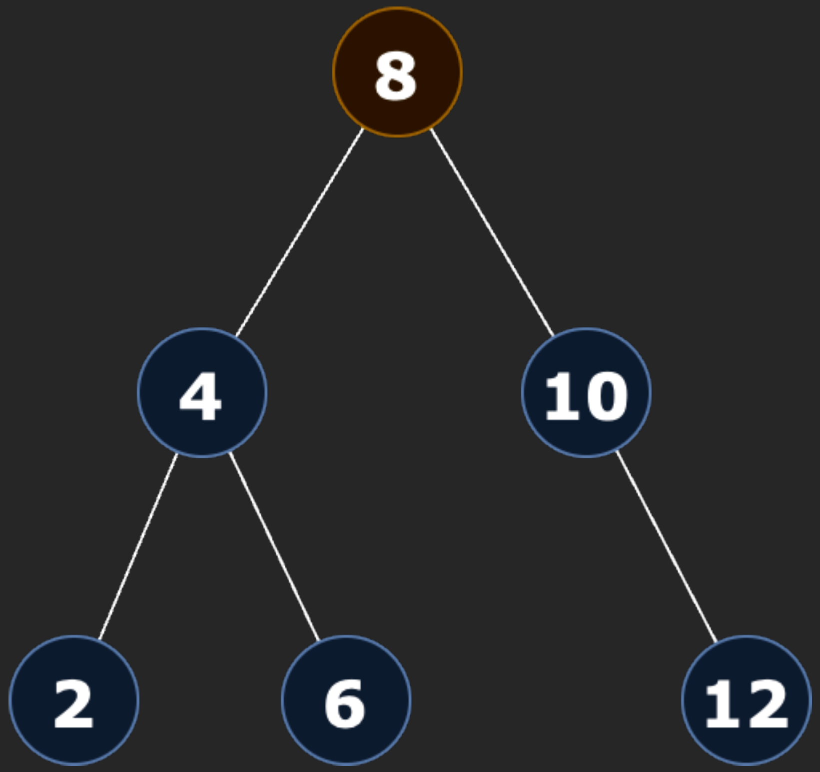
BST и ключи-дубликаты

Будем всегда рассматривать BST в случае обработки уникальных значений

На практике, дубликаты редко хранятся в виде отдельных записей

Бинарное дерево поиска. Вставка и удаление

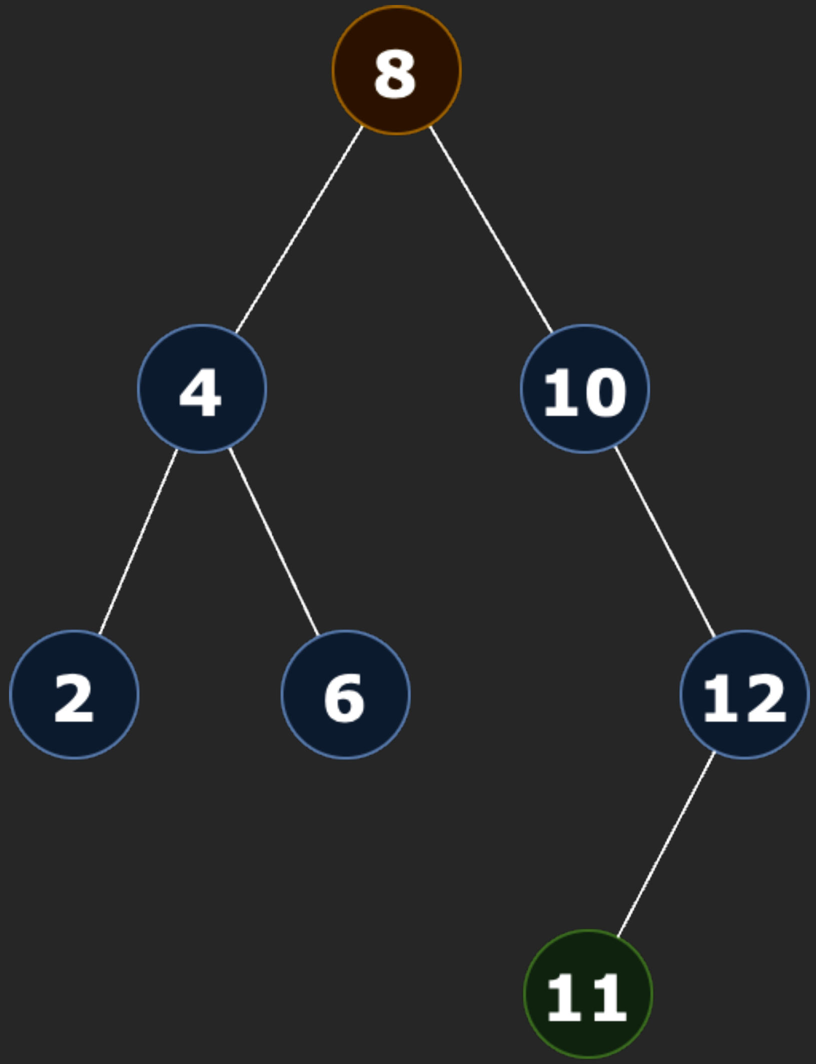
Вставка нового ключа в BST




```
insert 11
```

Место для нового ключа ищем последовательным спуском в правое или в левое поддерево

Вставка нового ключа в BST



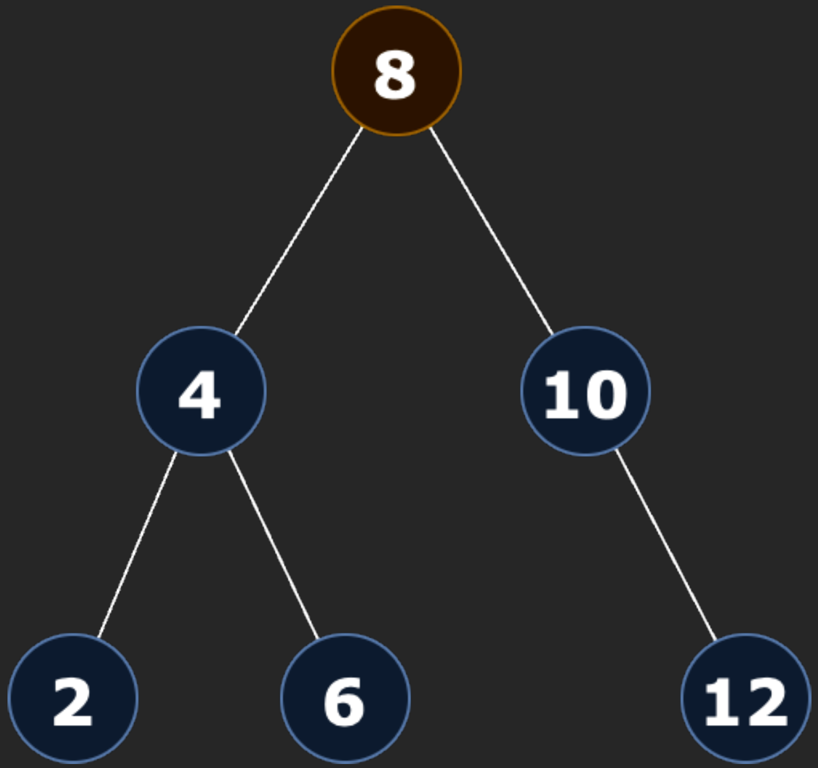
`insert 11`

Место для нового ключа ищем последовательным спуском в правое или в левое поддерево

Вставка нового ключа в BST

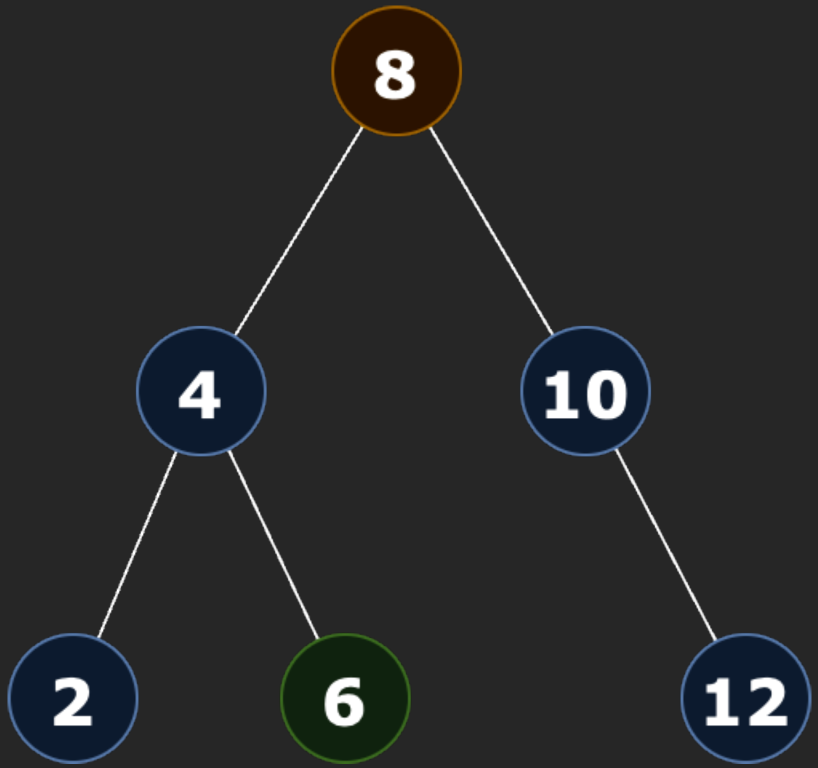
```
insert(Node* r, T key)
if (r == nullptr)
    return Node(key)
else if (key < r->data)
    r->left = insert(r->left, key)
else
    r->right = insert(r->right, key)
return r
```

Удаление ключа из BST. Лист




```
erase 6
```

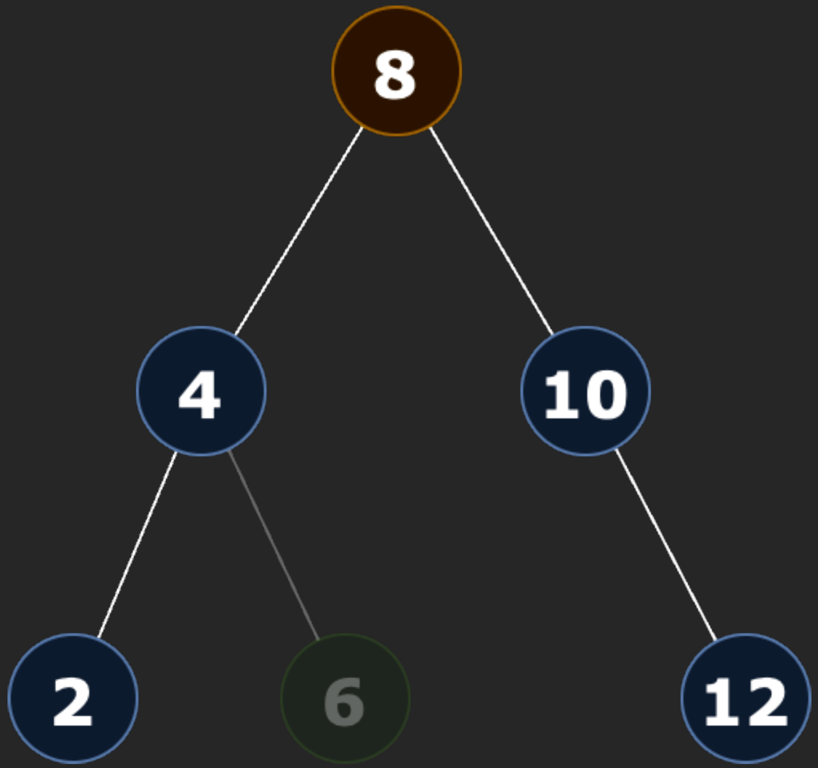
Удаление ключа из BST. Лист



`erase 6`

Найти ключ в дереве и установить предка

Удаление ключа из BST. Лист

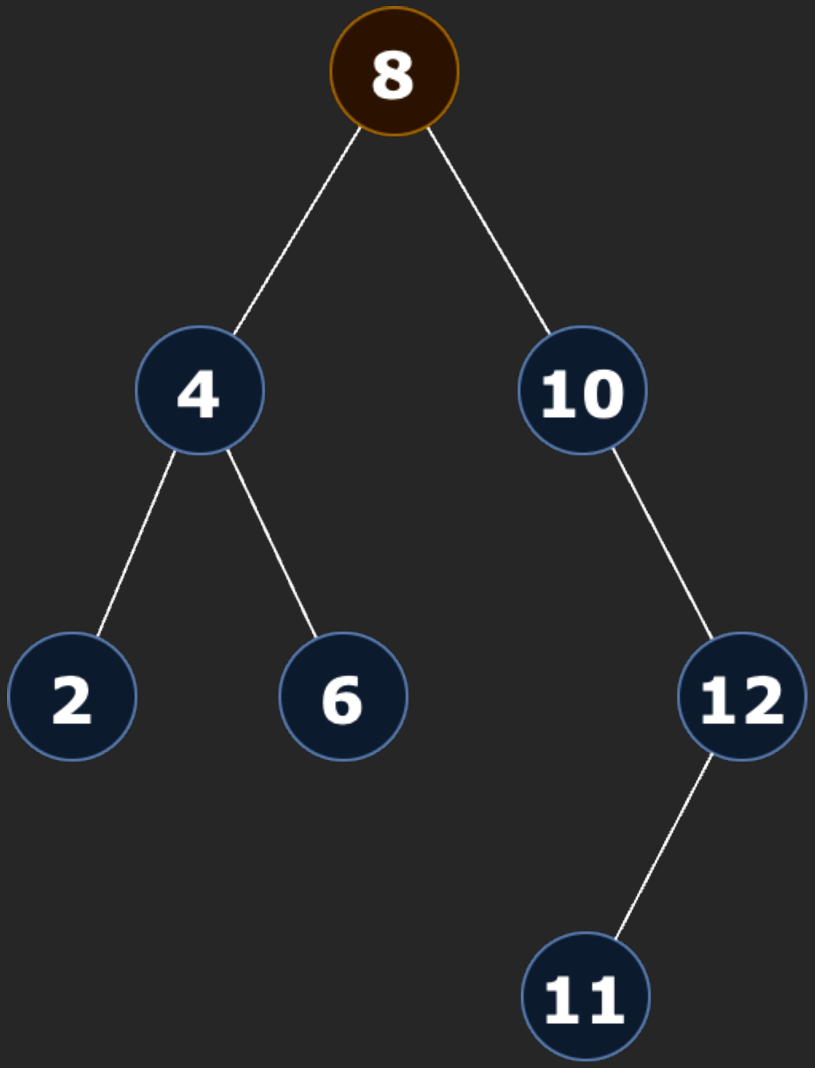


`erase 6`

Найти ключ в дереве и установить предка

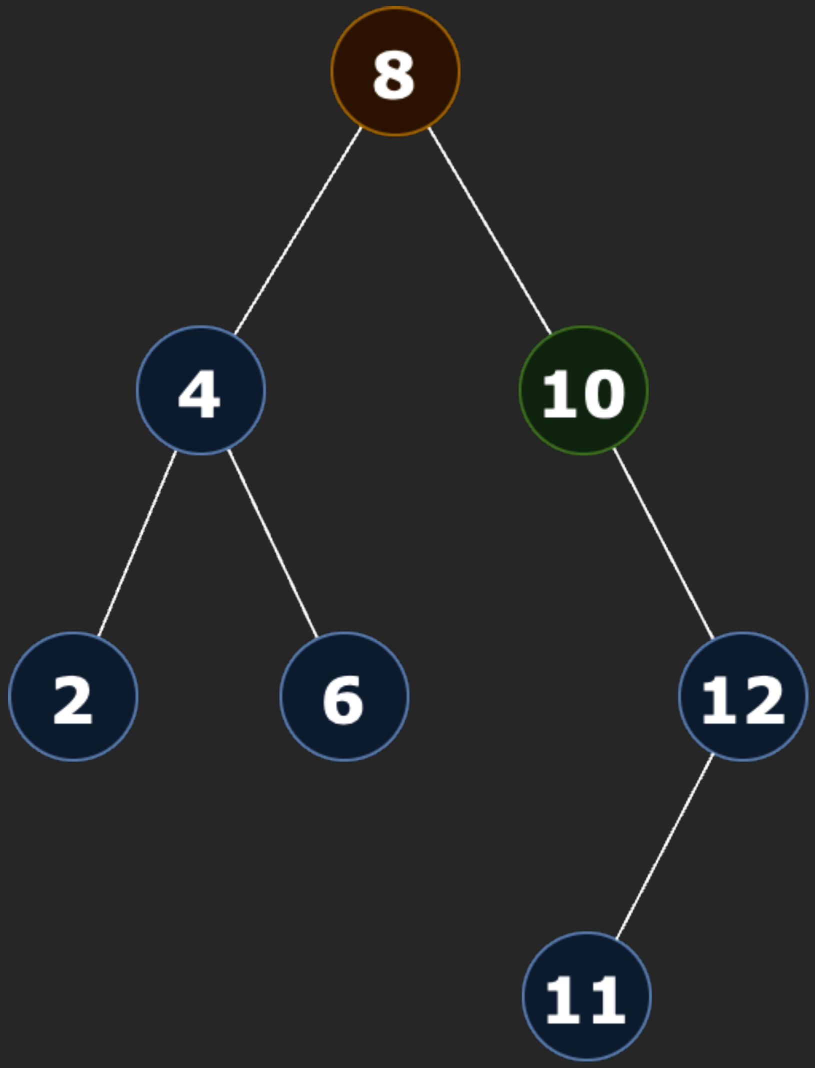
Освободить память и отвязать от предка

Удаление ключа из BST. Один потомок



`erase 10`

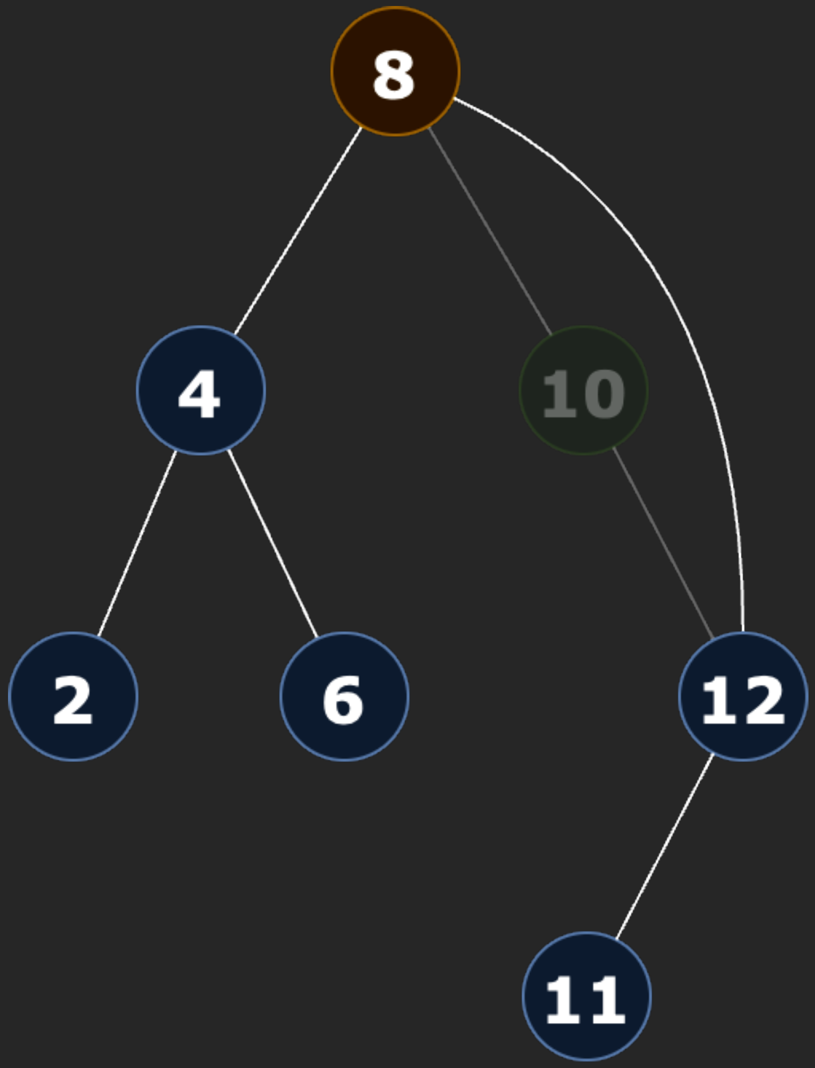
Удаление ключа из BST. Один потомок



`erase 10`

Удаление выполняется так же, как и из обычного односвязного списка

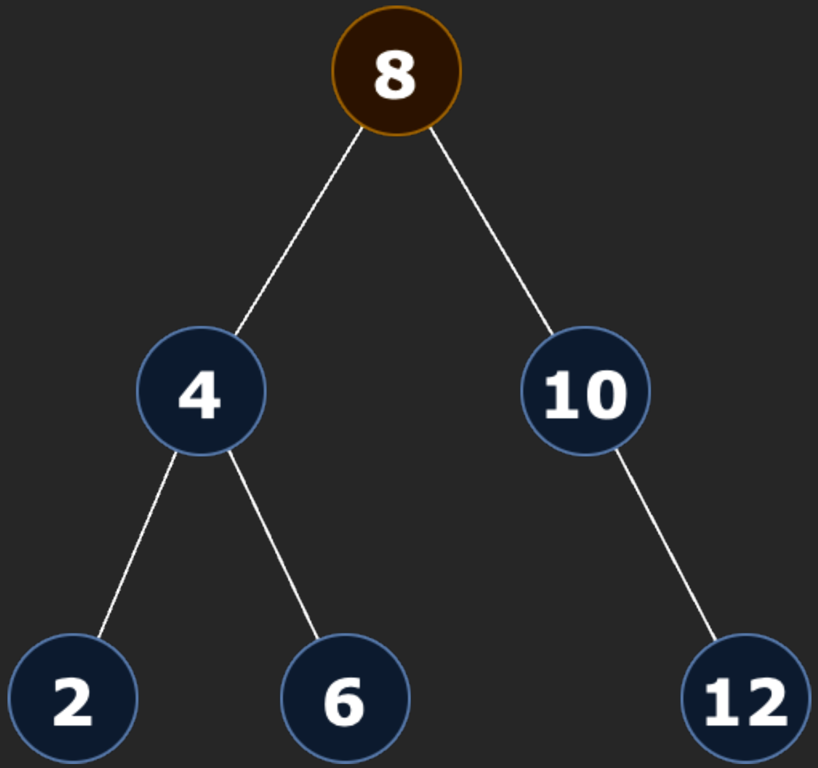
Удаление ключа из BST. Один потомок



`erase 10`

Удаление выполняется так же, как и из обычного односвязного списка

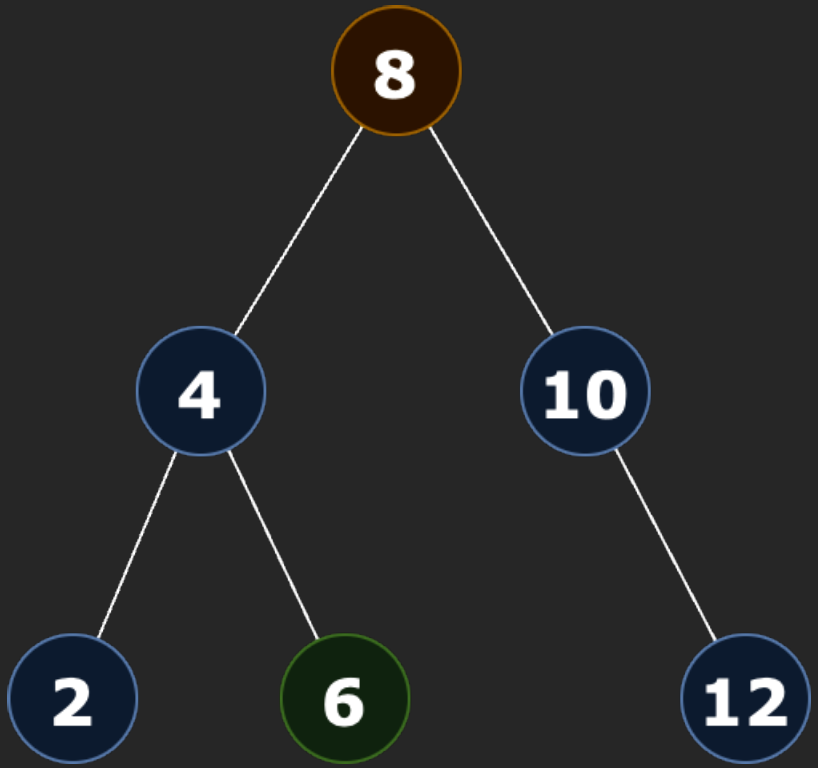
Удаление ключа из BST. Два потомка-1



`erase 8`

Замещаем удаляемый ключ на предыдущий/следующий

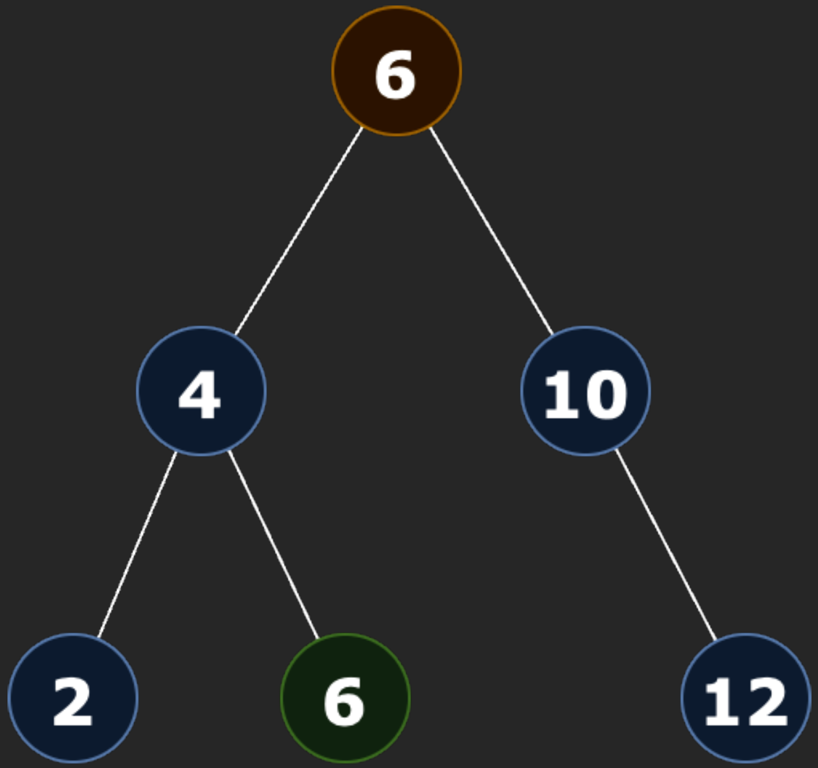
Удаление ключа из BST. Два потомка-1



`erase 8`

Замещаем удаляемый ключ на предыдущий

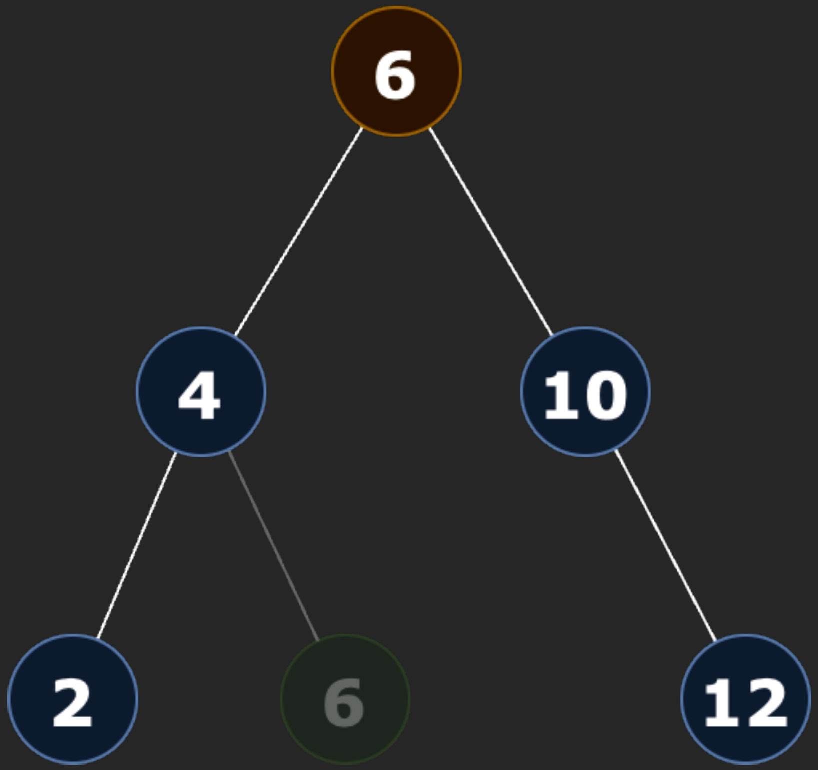
Удаление ключа из BST. Два потомка-1



`erase 8`

Замещаем удаляемый ключ на предыдущий

Удаление ключа из BST. Два потомка-1

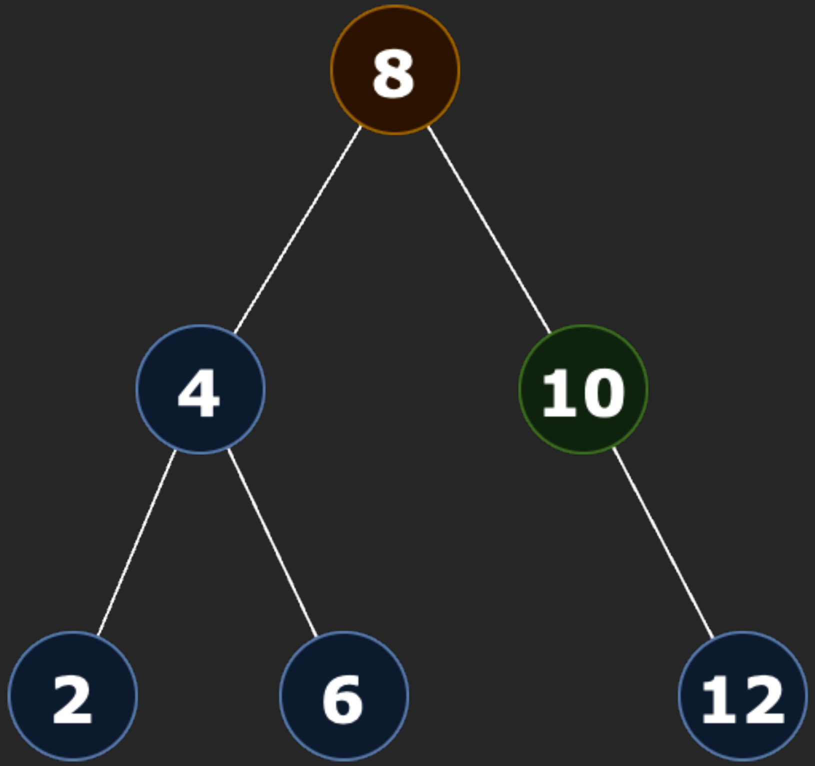


`erase 8`

Замещаем удаляемый ключ на предыдущий

Сводим к удалению листа

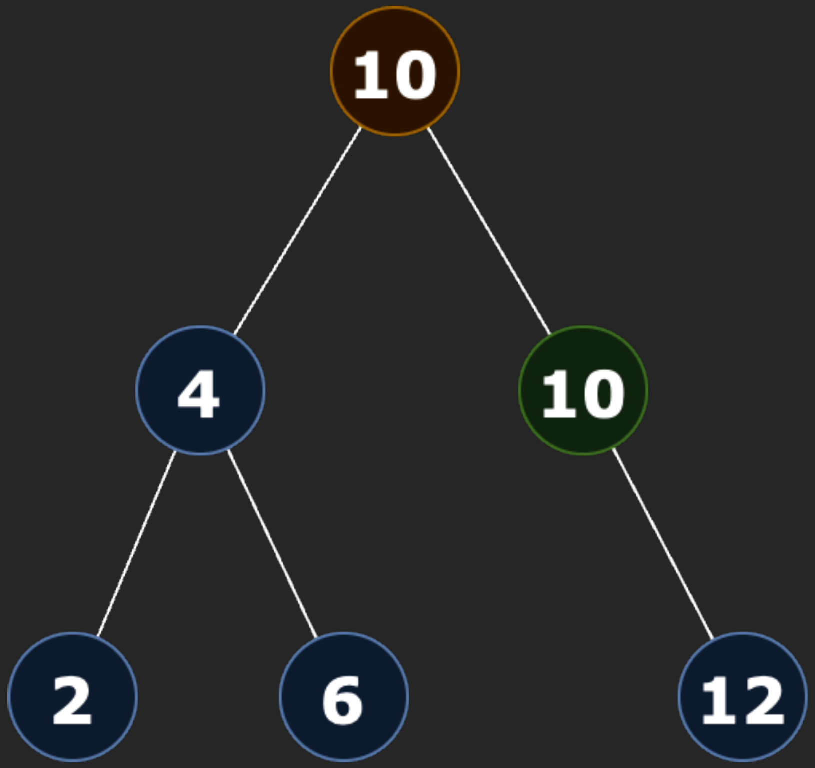
Удаление ключа из BST. Два потомка-1



`erase 8`

Замещаем удаляемый ключ на следующий

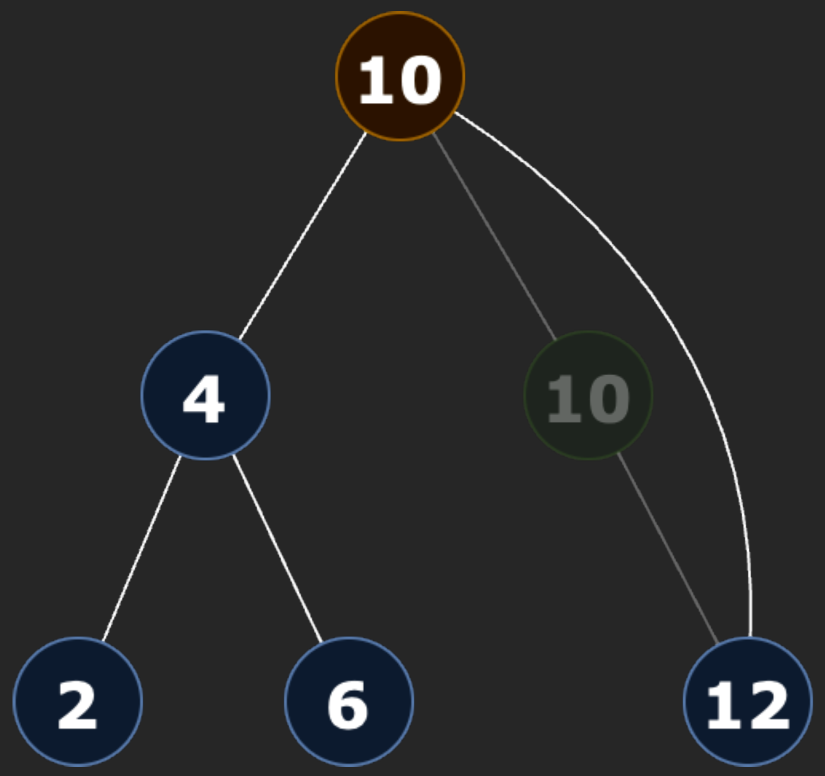
Удаление ключа из BST. Два потомка-1



`erase 8`

Замещаем удаляемый ключ на следующий

Удаление ключа из BST. Два потомка-1



`erase 8`

Замещаем удаляемый ключ на следующий

Сводим к удалению вершины с одним потомком

Предыдущее и следующее значение для некоторого ключа всегда находится в листе или вершине с одним потомком

Сложность основных операций с бинарным деревом поиска полностью определяется его высотой

Бинарное дерево поиска. Проблема баланса

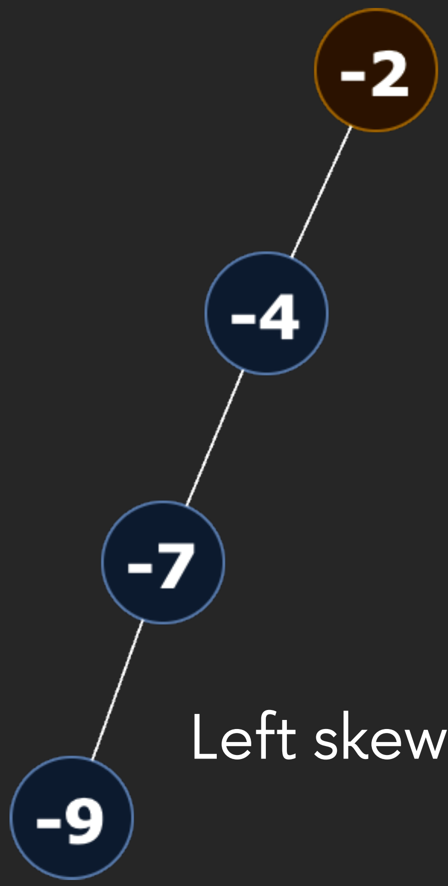
Вырождение BST

-2	-4	-7	-9

1	5	7	20

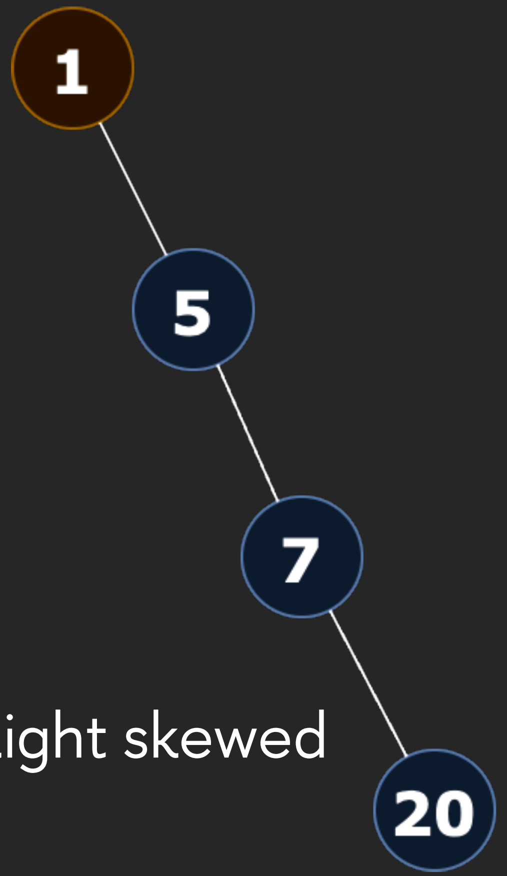
Вырождение BST

-2	-4	-7	-9



Left skewed

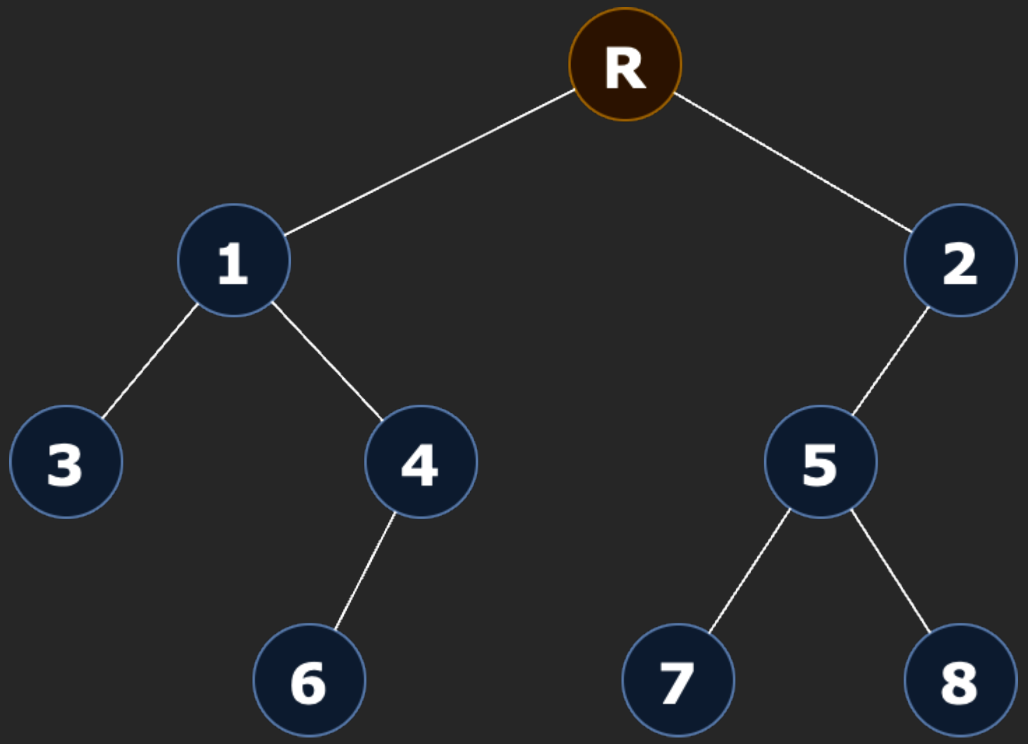
1	5	7	20



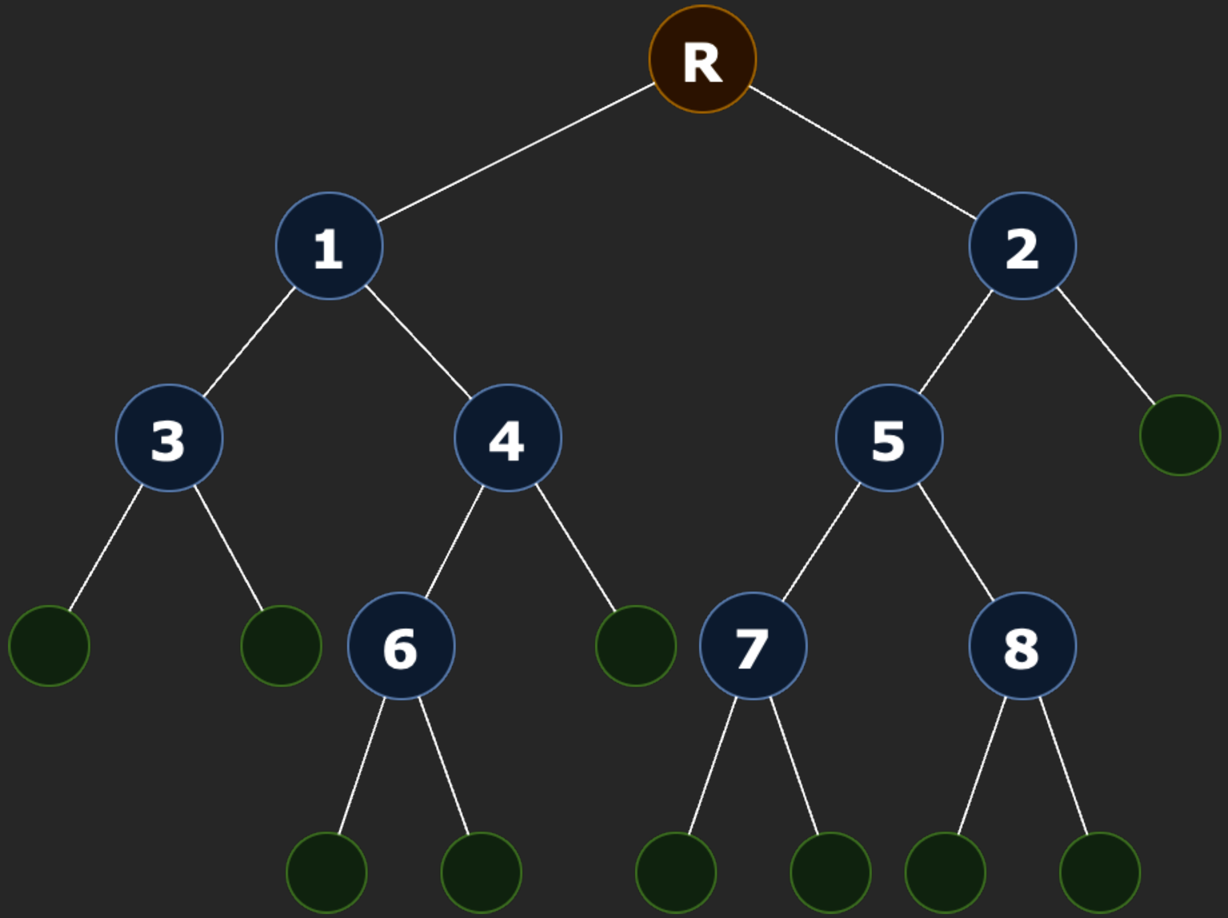
Right skewed

Стремимся к логарифмической высоте бинарного дерева поиска

Баланс по весу

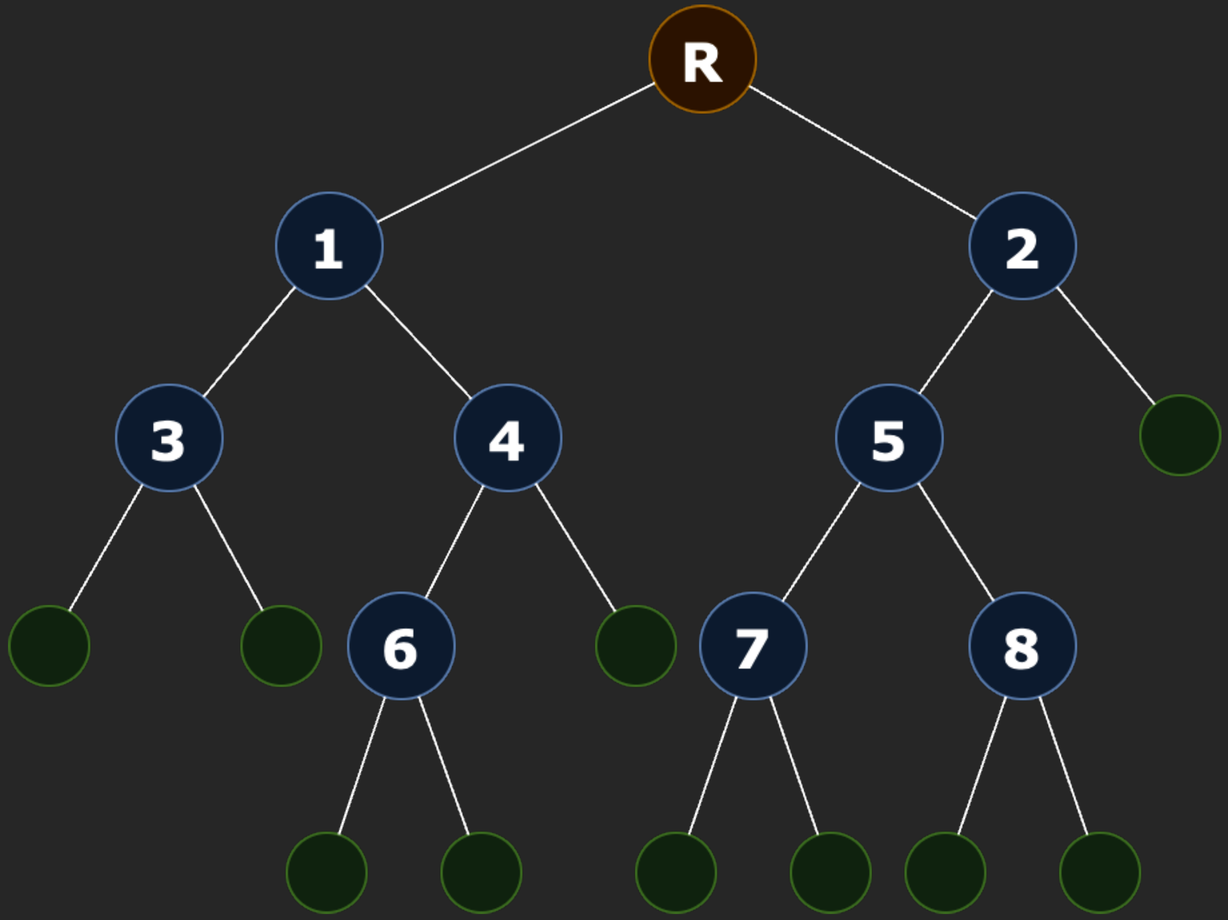


Баланс по весу



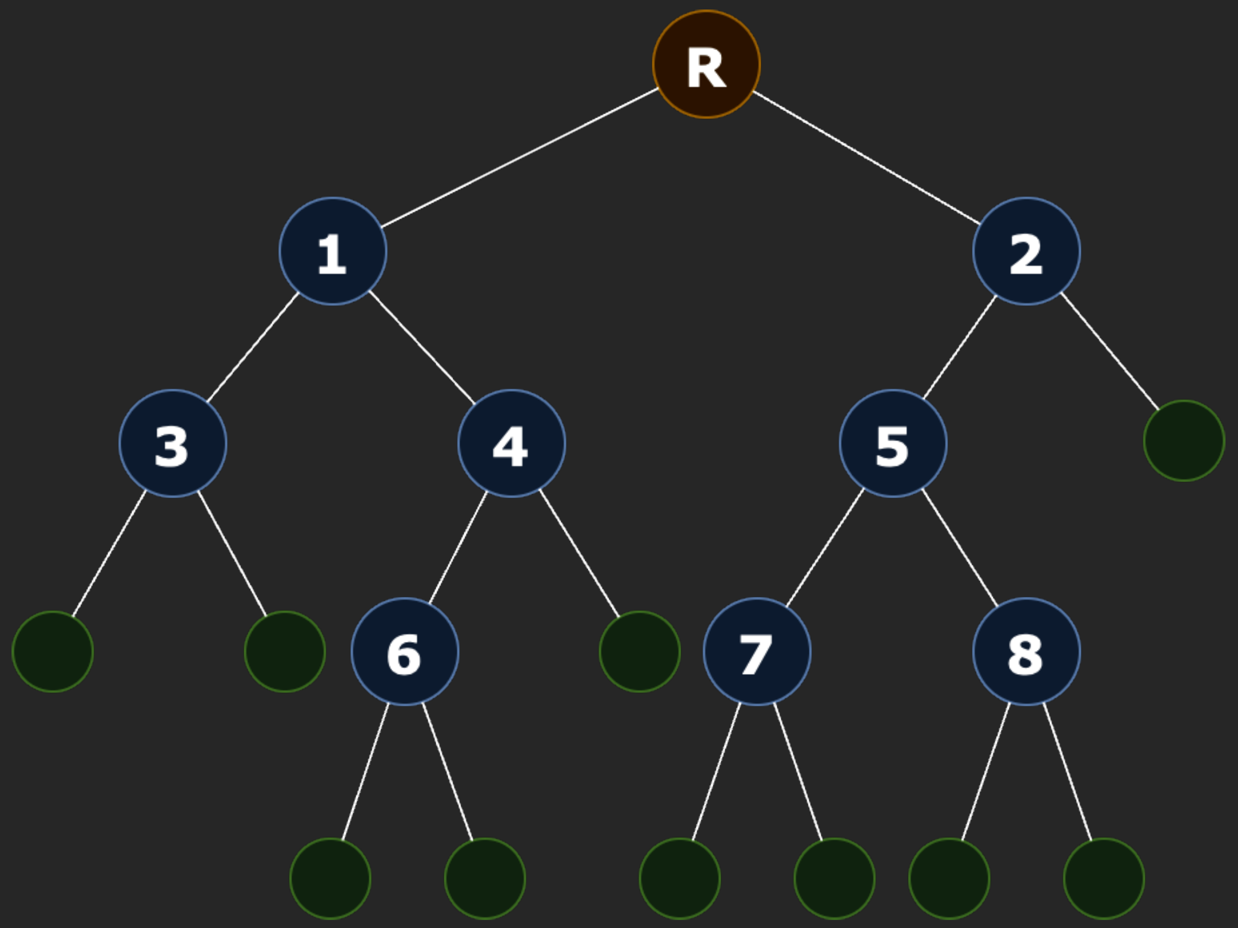
Введем null-вершины, которые будут заполняться при следующих вставках

Баланс по весу

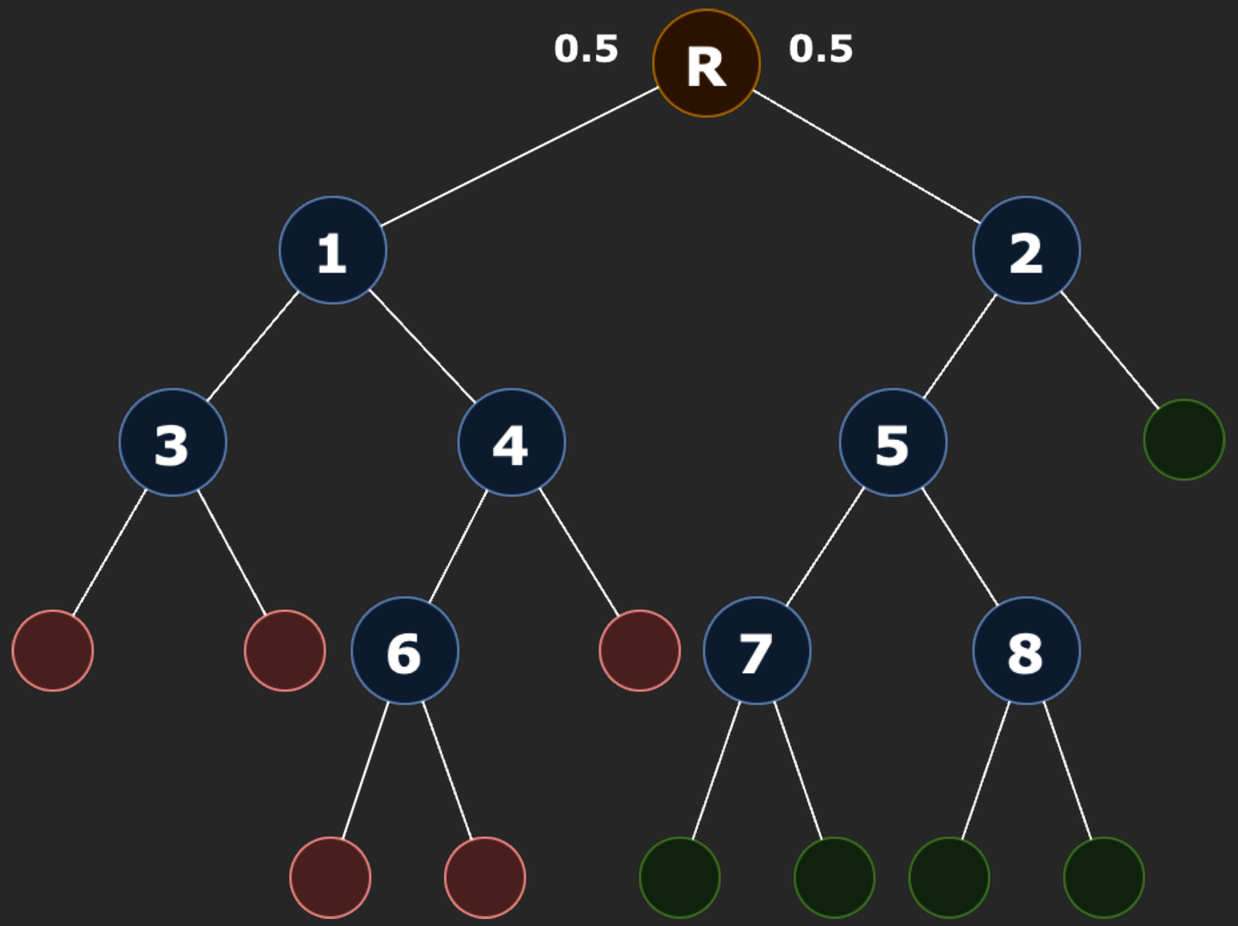


Отношение числа null-вершин в левом и правом поддеревьях к общему количеству

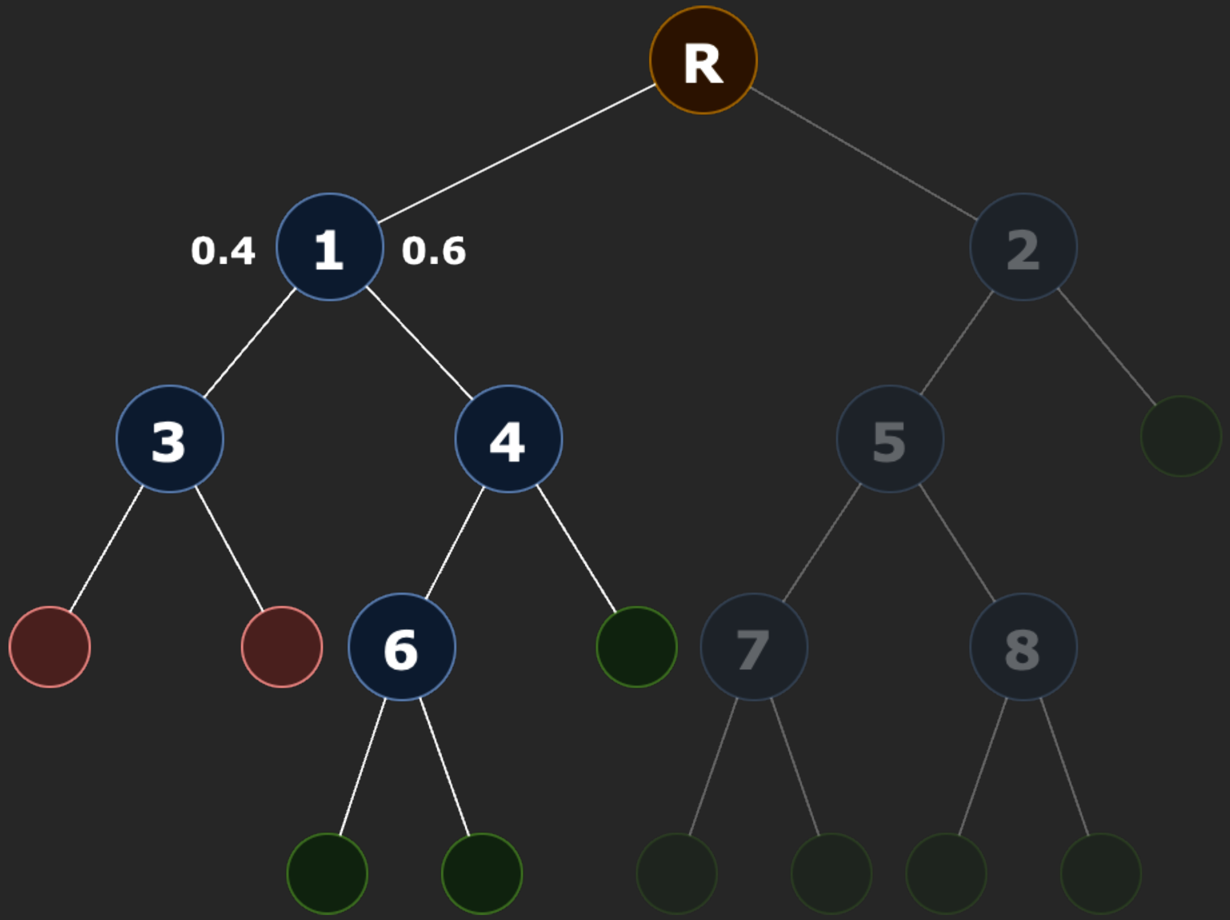
Баланс по весу



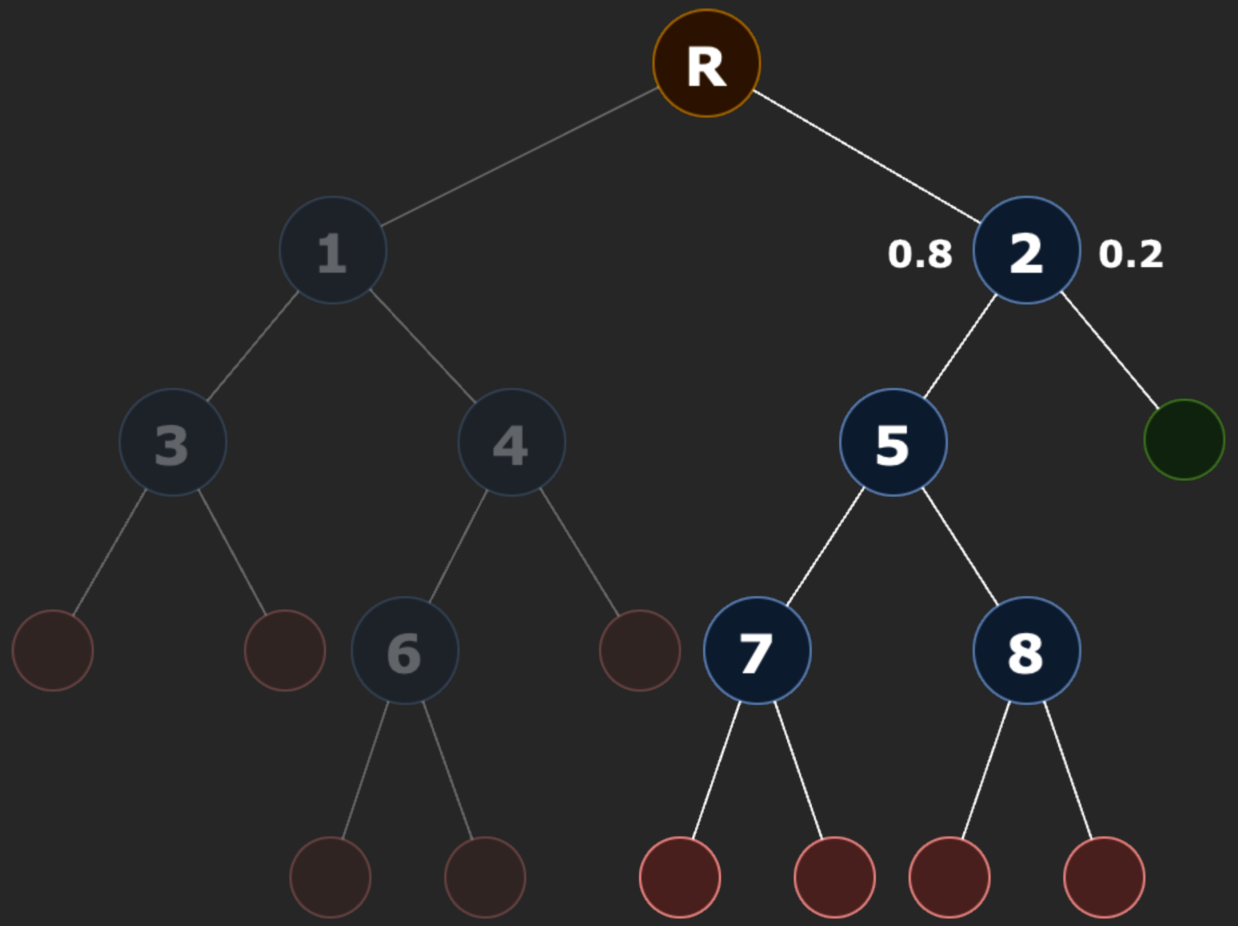
Баланс по весу



Баланс по весу



Баланс по весу

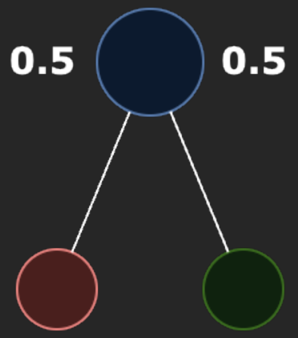


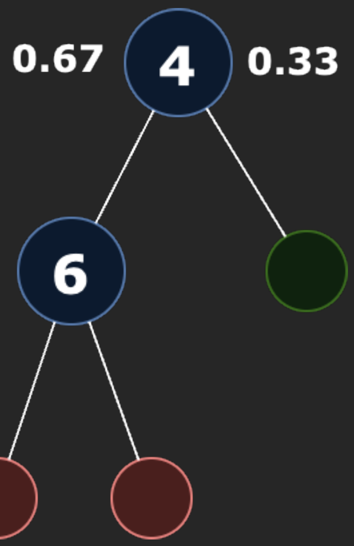
Баланс по весу. BB[α] деревья

Параметр $\alpha \in [0, {}^1/_3]$ определяет нижнюю границу доли null-вершин левого и правого поддерева для каждой вершины

Баланс по весу. BB[α] деревья

Параметр $\alpha \in [0, {}^{1}/_{3}]$ определяет нижнюю границу доли null-вершин левого и правого поддерева для каждой вершины



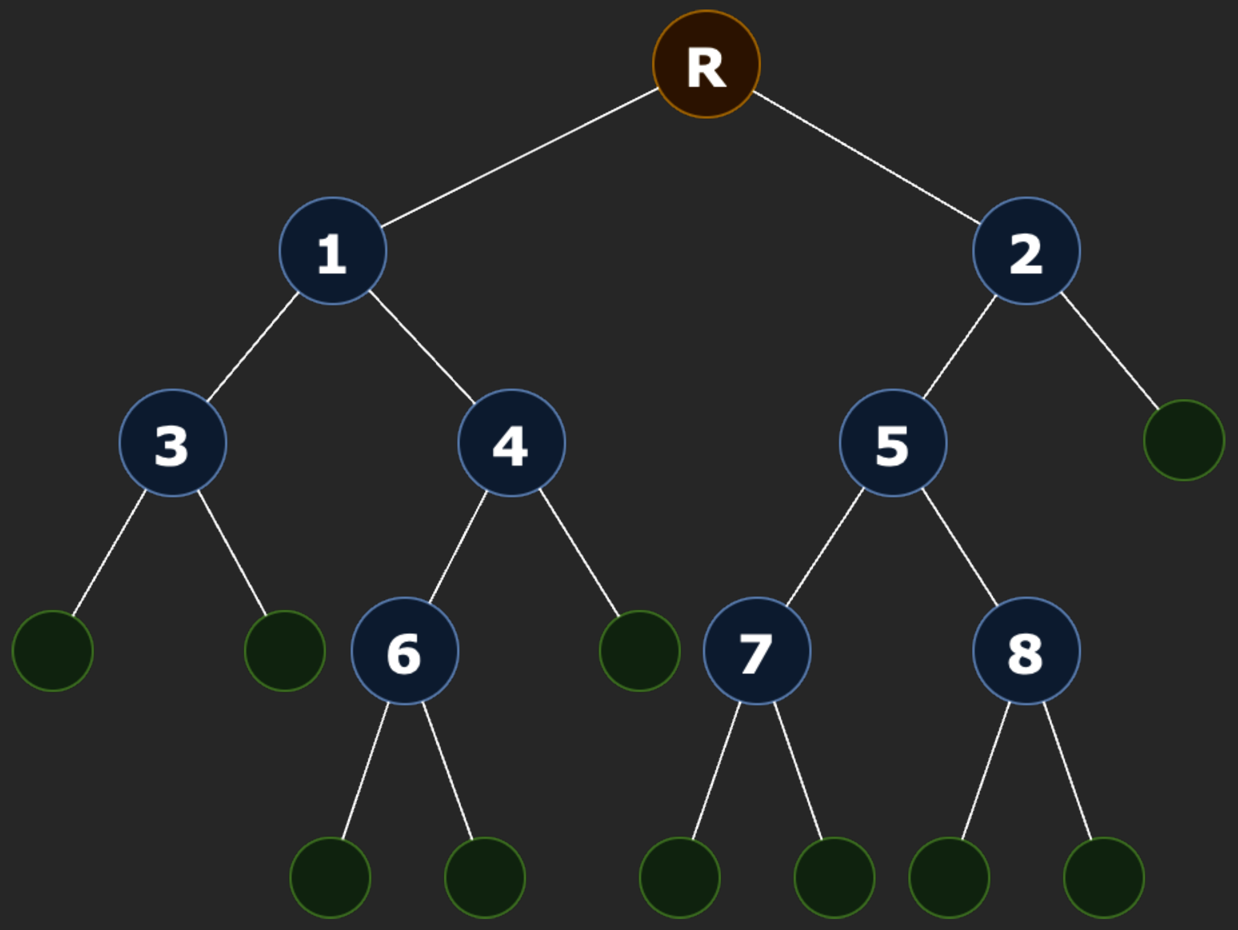


Баланс по весу. BB[α] деревья

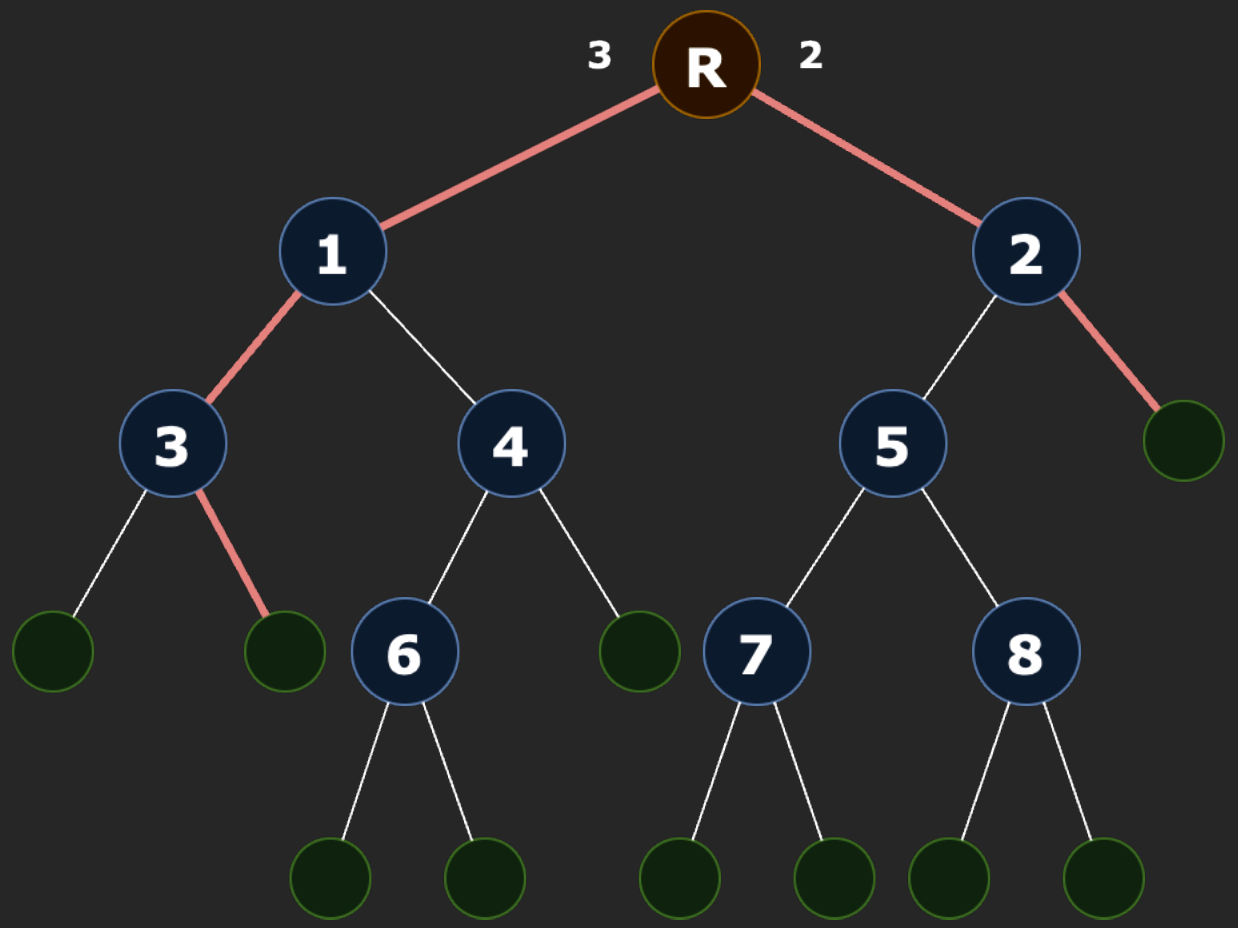
Параметр $\alpha \in [0, {}^{1}/_{3}]$ определяет нижнюю границу доли null-вершин левого и правого поддерева для каждой вершины

Логарифмическая сложность основных операций достигается при $\alpha \in \left[{}^{1}/_{2}, {}^{\sqrt{2}}/_{2}\right]$.

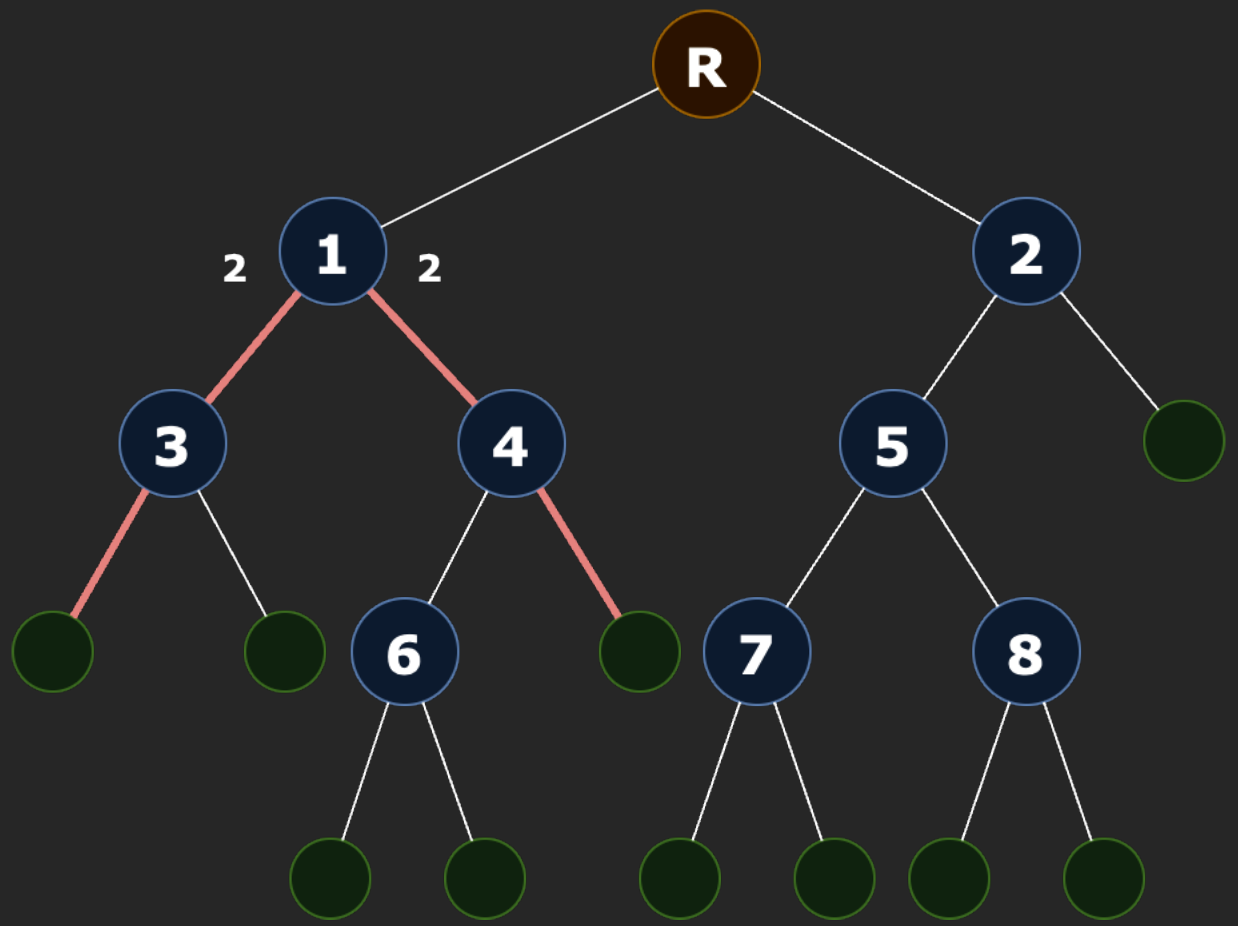
Баланс по длине путей



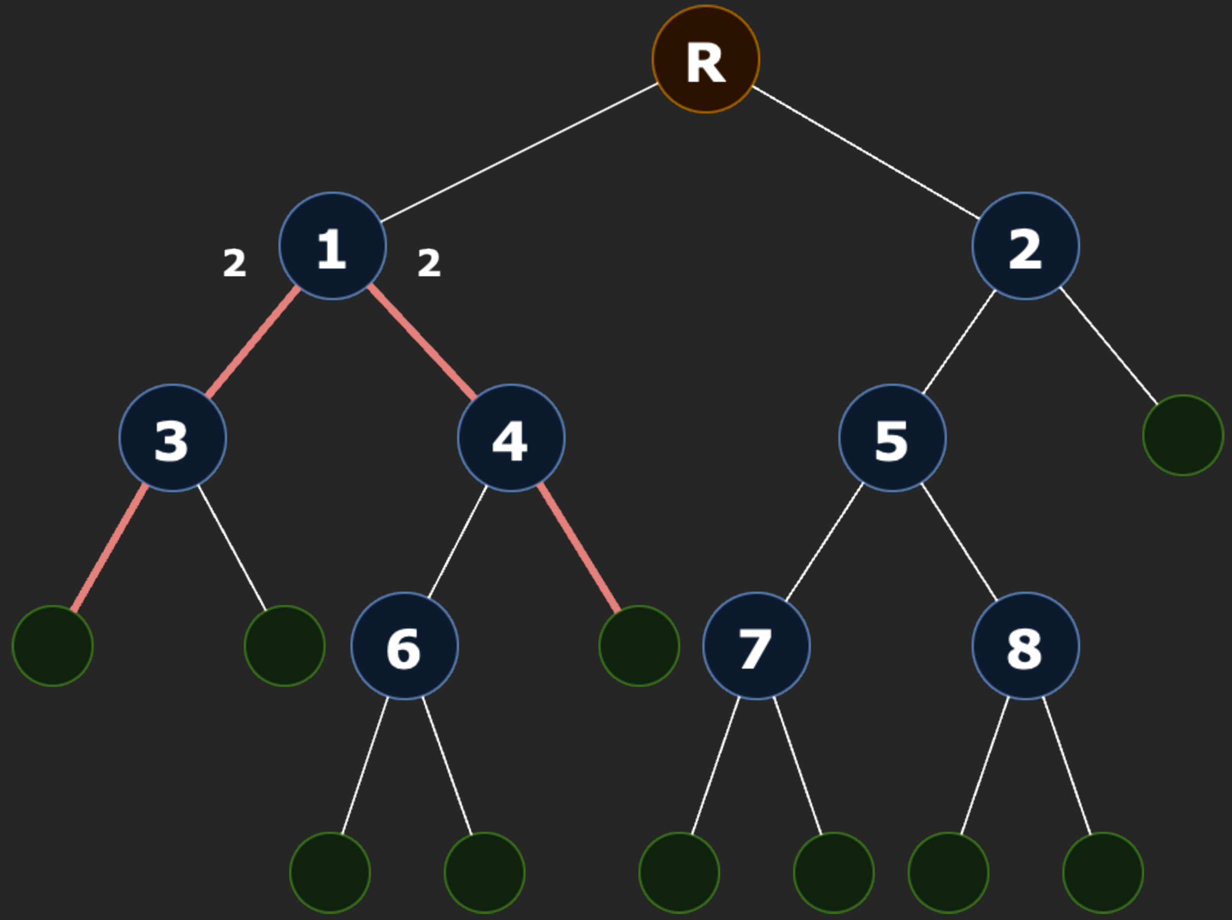
Баланс по длине путей



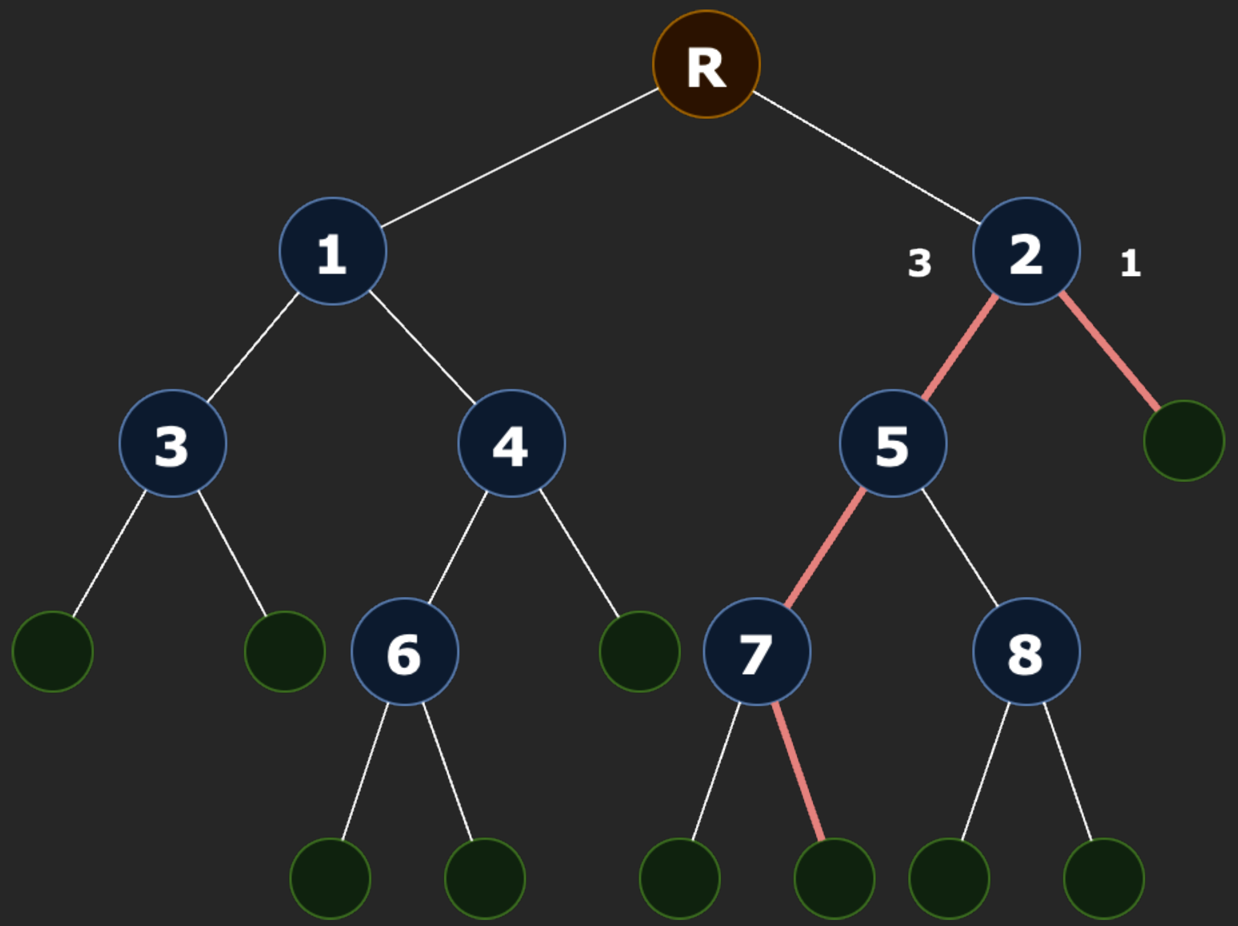
Баланс по длине путей



Баланс по длине путей



Баланс по длине путей



Баланс по длине путей. КЧД

В красно-черных деревьях следят за тем, чтобы длина кратчайших путей до null-вершин отличалась не более, чем вдвое для каждой вершины

Баланс по высоте. AVL

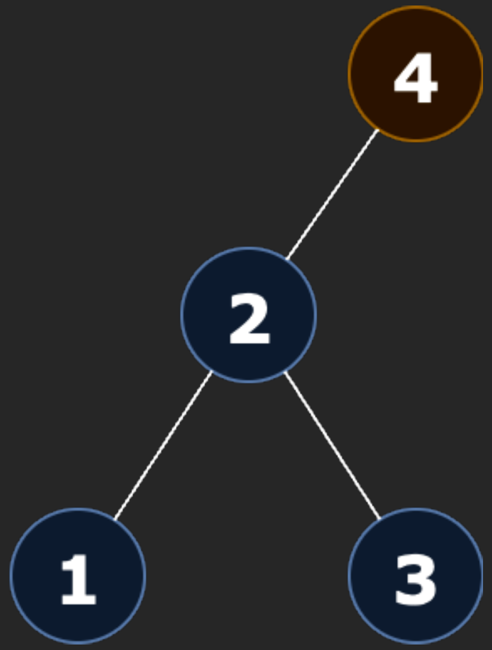
Прямое обеспечение баланса в дереве

В AVL-деревьях высоты поддеревьев не должны отличаться более, чем на 1

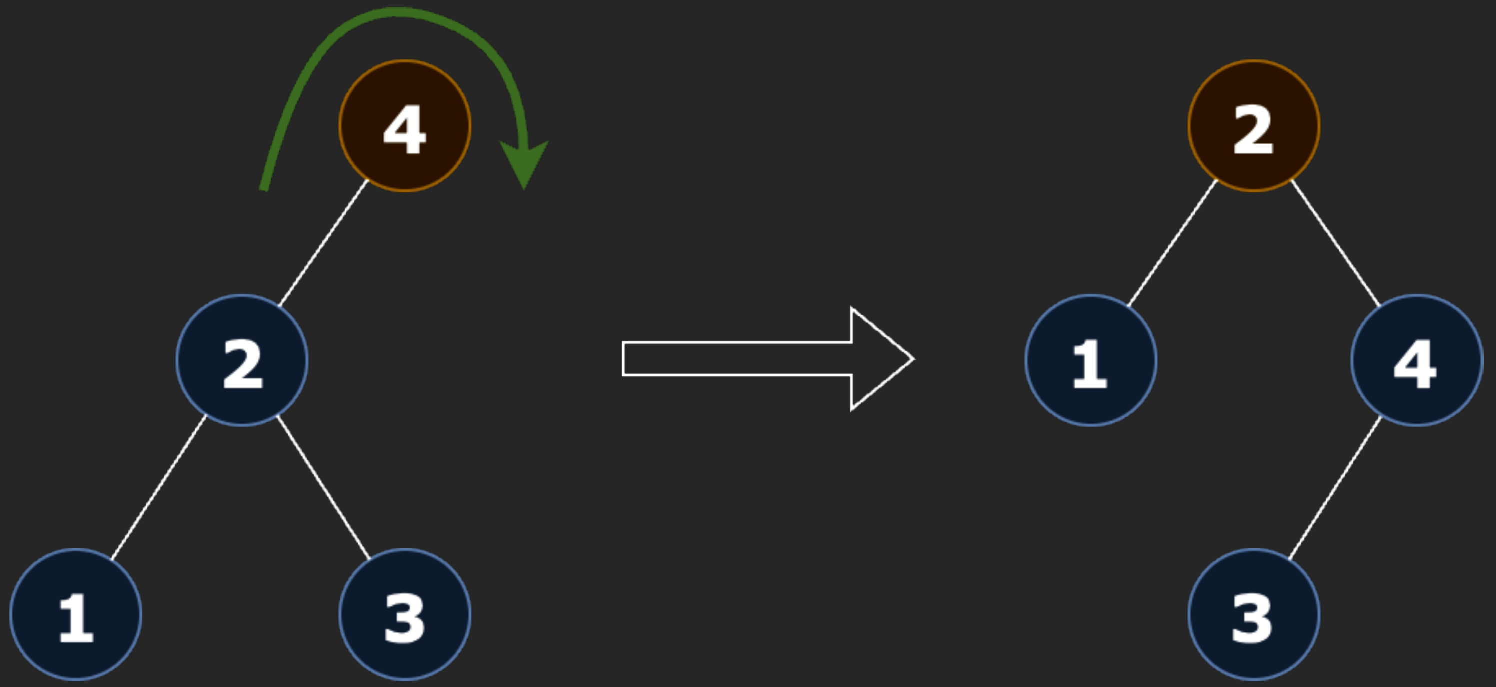
Как исправить разбалансировку?

…и не нарушить порядок

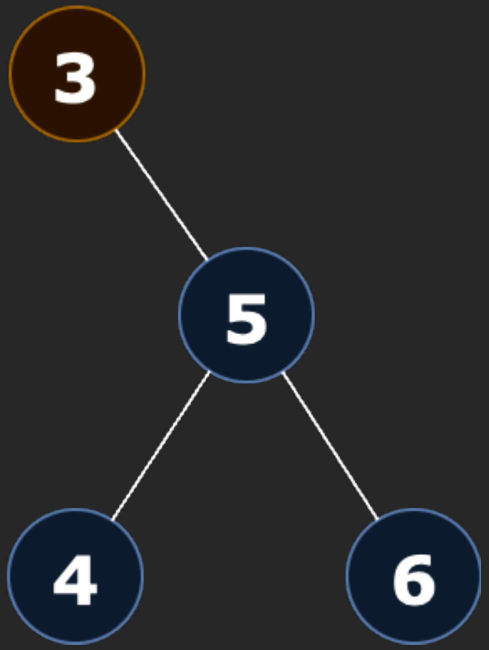
Правый поворот



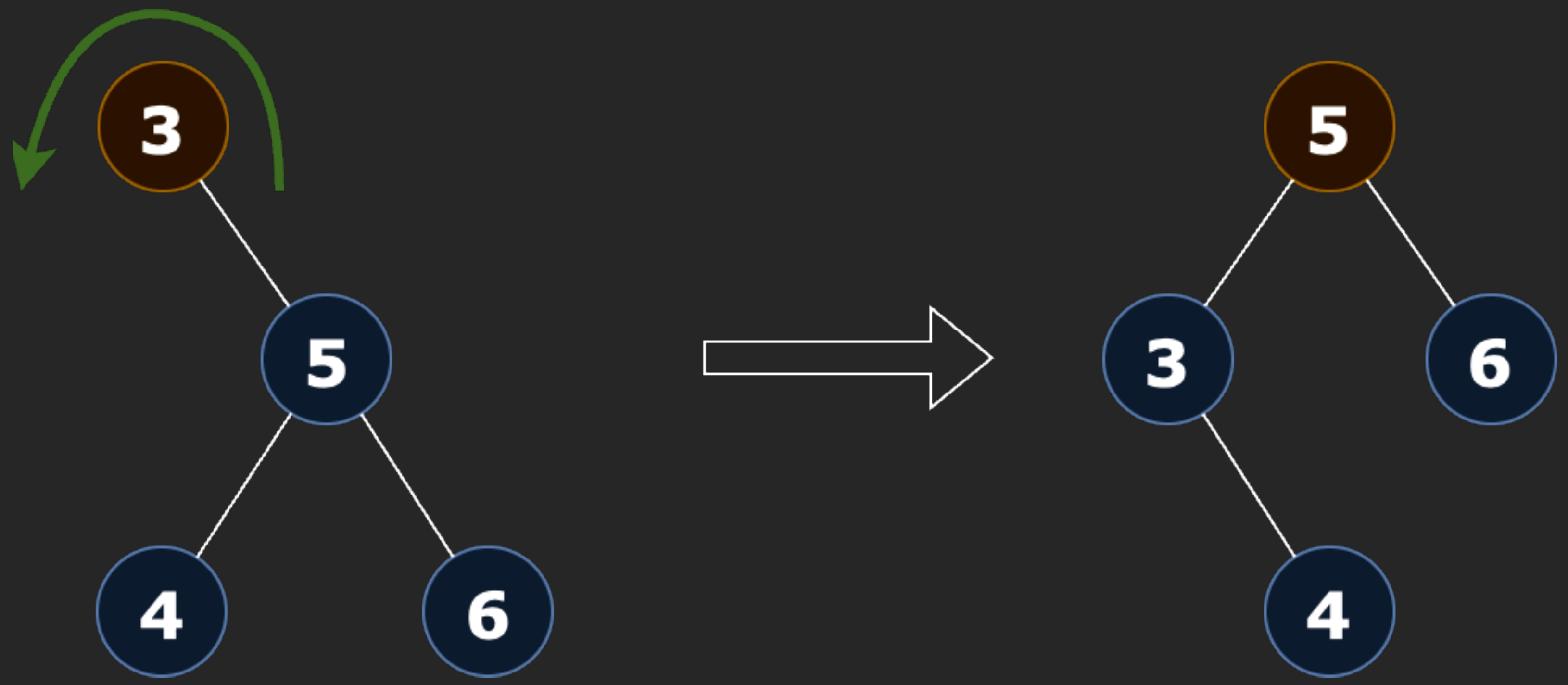
Правый поворот



Левый поворот



Левый поворот



Повороты дерева

Операции, направленные на исправление локальной разбалансировки

Выполняются при вставке и удалении ключей

Recap

Бинарное дерево. Классификация и анализ

Бинарное дерево поиска. Неявное упорядочивание ключей. ADT «Отсортированный список»

Прямой и косвенный анализ баланса бинарного дерева поиска

Teaser – Лекция 10

AVL-дерево. Баланс по высоте

Самобалансирующееся B-дерево. 2-3-4 дерево

Красно-черное дерево. Изометрия с B-деревом