02.12.2024

Деревья. Часть 1.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

План лекции

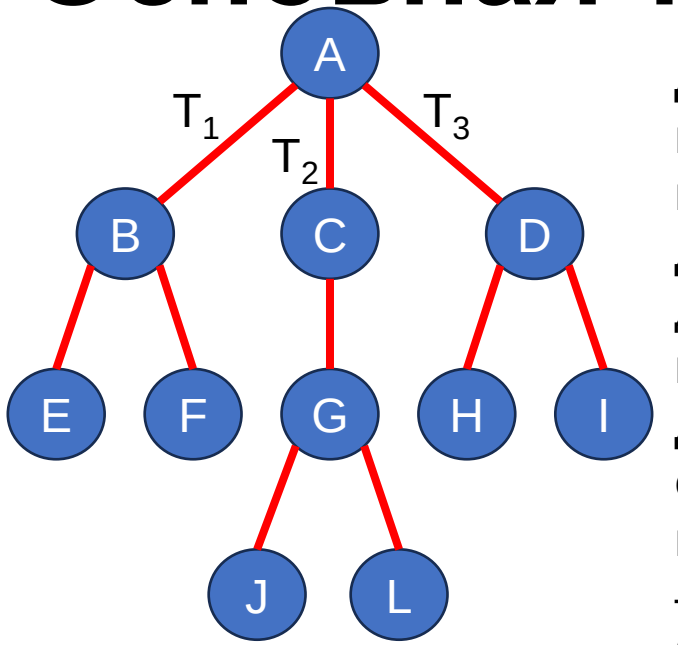
Деревья. Часть 1.

90 минут

Введение

Деревья поиска представляют собой структуры данных, которые поддерживают многие операции с динамическими множествами, включая Search, Minimum, Maximum, Predecessor, Successor, Insert и Delete. Таким образом, дерево поиска может использоваться и как словарь, и как очередь с приоритетами.
Основные операции в бинарном дереве поиска выполняются за время, пропорциональное его высоте. Для полного бинарного дерева с n узлами эти операции выполняются за время O(lgn) в наихудшем случае. Однако, если дерево представляет собой линейную цепочку из n узлов, те же операции выполняются в наихудшем случае за время O(n). Математическое ожидание высоты построенного случайным образом бинарного дерева равно O(lgn), так что все основные операции над динамическим множеством в таком дереве выполняются в среднем за время O(lgn).
На практике мы не всегда можем гарантировать случайность построения бинарного дерева поиска, однако имеются версии деревьев, в которых гарантируется хорошее время работы в наихудшем случае. Одна из таких версий, а именно - красно-черные деревья, высота которых составляет O(lgn). Существуют также B-деревья, которые особенно хорошо подходят для баз данных, хранящихся во вторичной памяти с произвольным доступом (на дисках).

Основная терминология



Дерево — особый вид сети, или графа, поэтому по аналогии ветви, соединяющие вершины, иногда называют ссылками или линиями.

Дерево состоит из ***вершин (узлов),*** которые содержат данные и соединяются ***ветвями***. У каждой вершины, за исключением корневой, есть одна родительская вершина.

Две вершины с общим родителем иногда называют узлами-сестрами, дочерние вершины дочерних вершин — потомками, а родительские вершины родительских вершин — предками. Все эти термины, описывающие связи, очень близки к генеалогии.

В зависимости от типа дерева у вершины может быть разное количество потомков. Это число называется ***степенью вершины***. Его максимальное значение определяет степень дерева в целом.

Уровень (глубина) вершины — это расстояние от вершины до корня. У корня оно будет равно 0.

Высота вершины — это длина нисходящего пути от текущей вершины до листовой, то есть до низа дерева. Высота дерева равна высоте корневой вершины.

Поддеревом дерева T с корнем R является вершина R со всеми ее потомками.

В ***упорядоченном дереве*** расположение дочерних деревьев имеет значение. Многие алгоритмы по-разному рассматривают левую и правую дочерние вершины.

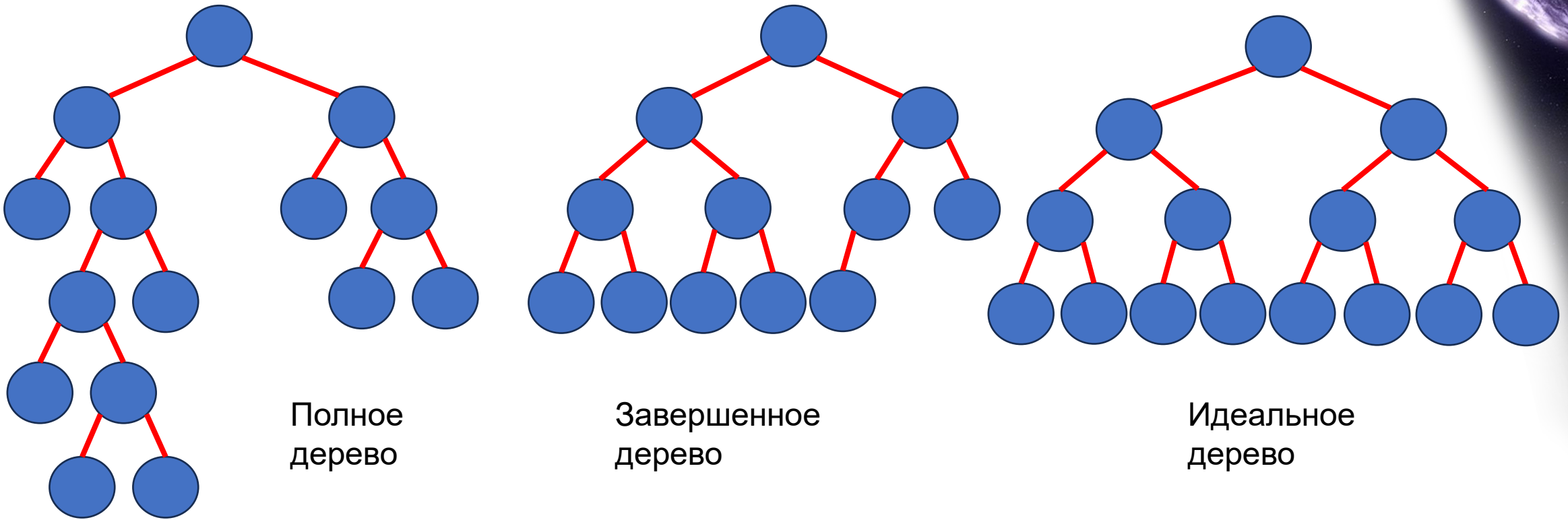
В ***неупорядоченном*** расположение дочерних деревьев не играет роли.

Основная терминология

В ***полном дереве*** вершина не имеет дочерних вершин либо их количество равно степени дерева. Например, в полном бинарном дереве у каждого узла может быть два дочерних или ни одного.

У ***завершенного дерева*** каждый уровень является полным, за исключением, возможно, нижнего, где все вершины сдвигаются влево.

Идеальное дерево — полное дерево, все листья которого находятся на одном уровне. Другими словами, в нем присутствуют все вершины, допустимые для его, высоты.



Полное дерево

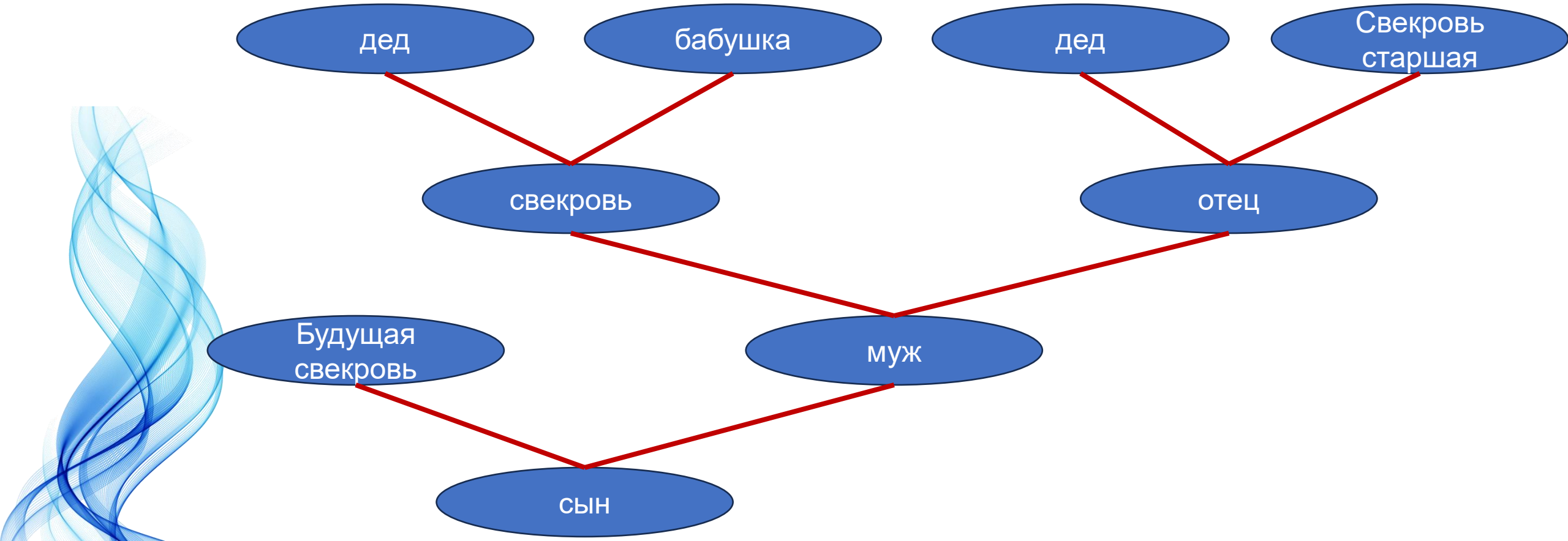
Завершенное дерево

Идеальное дерево

Общие деревья

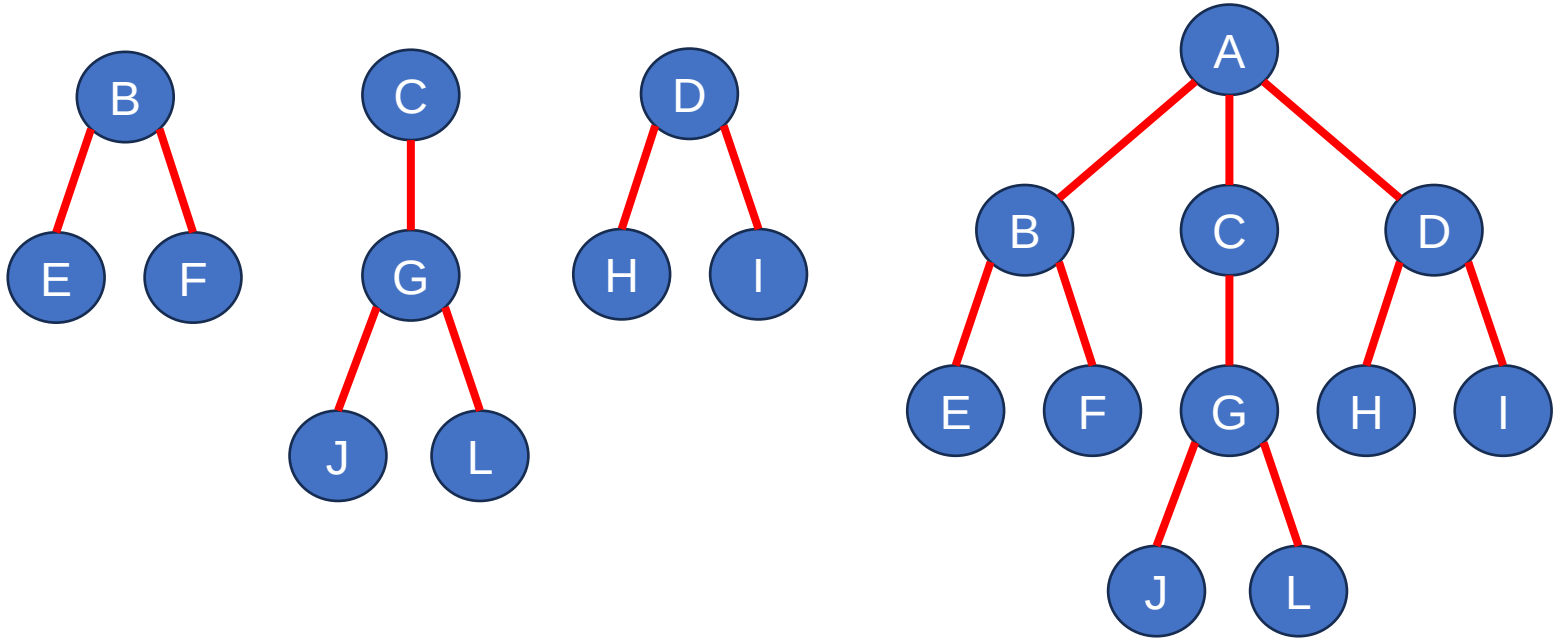
Общие деревья — это структуры данных, которые хранят элементы иерархически. Верхний узел дерева — это корневой узел, и каждый узел, кроме корня, имеет родителя. Узел в общем дереве (кроме листовых узлов) может иметь ноль или более поддеревьев. Общие деревья, которые имеют 3 поддерева на узел, называются тернарными деревьями. Однако количество поддеревьев для любого узла может быть переменным. Например, узел может иметь 1 поддерево, тогда как какой-либо другой узел может иметь 3 поддерева. Хотя общие деревья можно представить в виде АТД, всегда возникает проблема, когда к узлу, к которому уже прикреплено максимальное количество поддеревьев, добавляется еще одно поддерево. Даже алгоритмы поиска, обхода, добавления и удаления узлов становятся намного сложнее, поскольку для любого узла существует не просто две возможности, а несколько возможностей. Чтобы преодолеть сложности общего дерева, его можно представить в виде структуры данных графа (будет обсуждаться позже), тем самым теряя многие преимущества древовидных процессов. Поэтому лучшим вариантом является преобразование общих деревьев в бинарные деревья.

Пример – родственное дерево



Леса

Лес — это несвязное объединение деревьев. Набор несвязных деревьев (или лесов) получается путем удаления корня и ребер, соединяющих корневой узел с узлами на уровне 1. Мы уже видели, что каждый узел дерева является корнем некоторого поддерева. Поэтому все поддеревья, расположенные непосредственно под узлом, образуют лес.

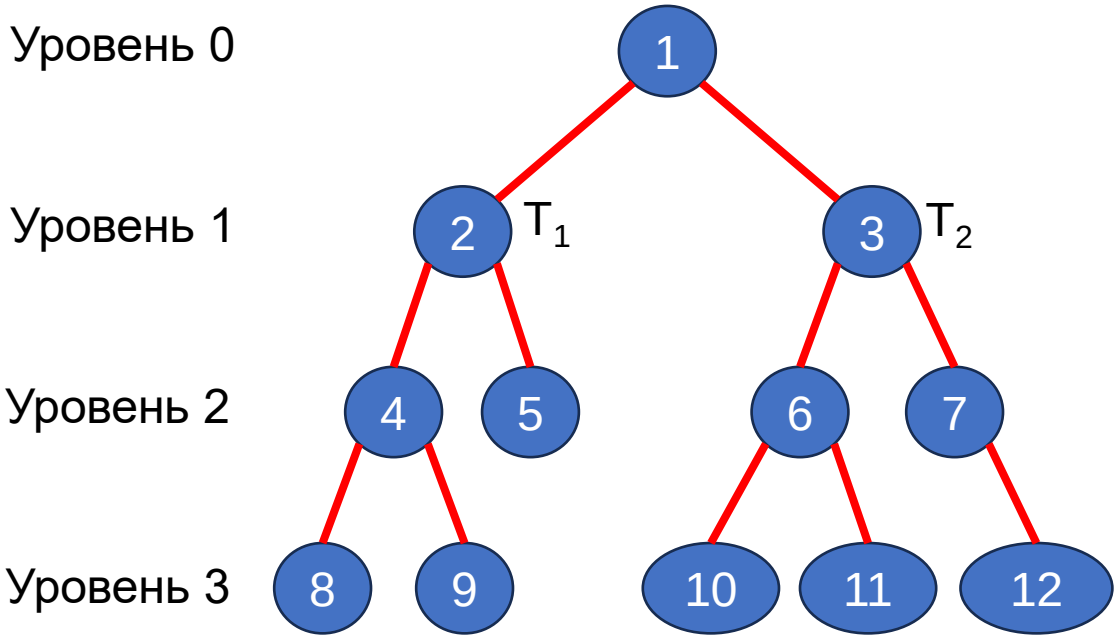


Лес и соответствующее ему дерево

Двоичные деревья

Двоичное дерево — это структура данных, которая определяется как набор элементов, называемых узлами. В бинарном дереве самый верхний элемент называется корневым узлом, и каждый узел имеет 0, 1 или максимум 2 дочерних элемента. Узел, не имеющий дочерних элементов, называется листовым узлом или конечным узлом. Высота полного двоичного дерева Tn, имеющего ровно n узлов, задается как:

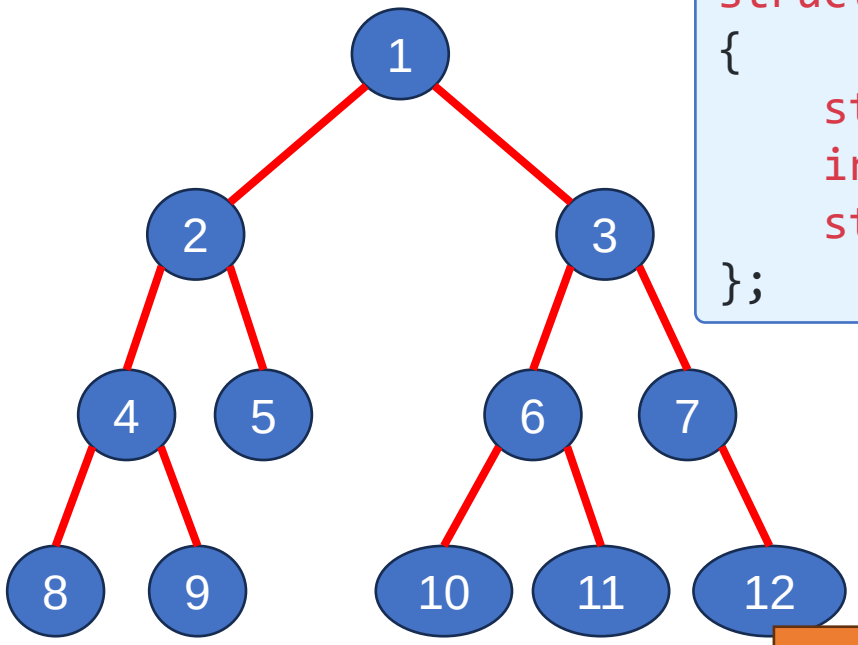
$H_n = | \log_2 (n + 1) |$

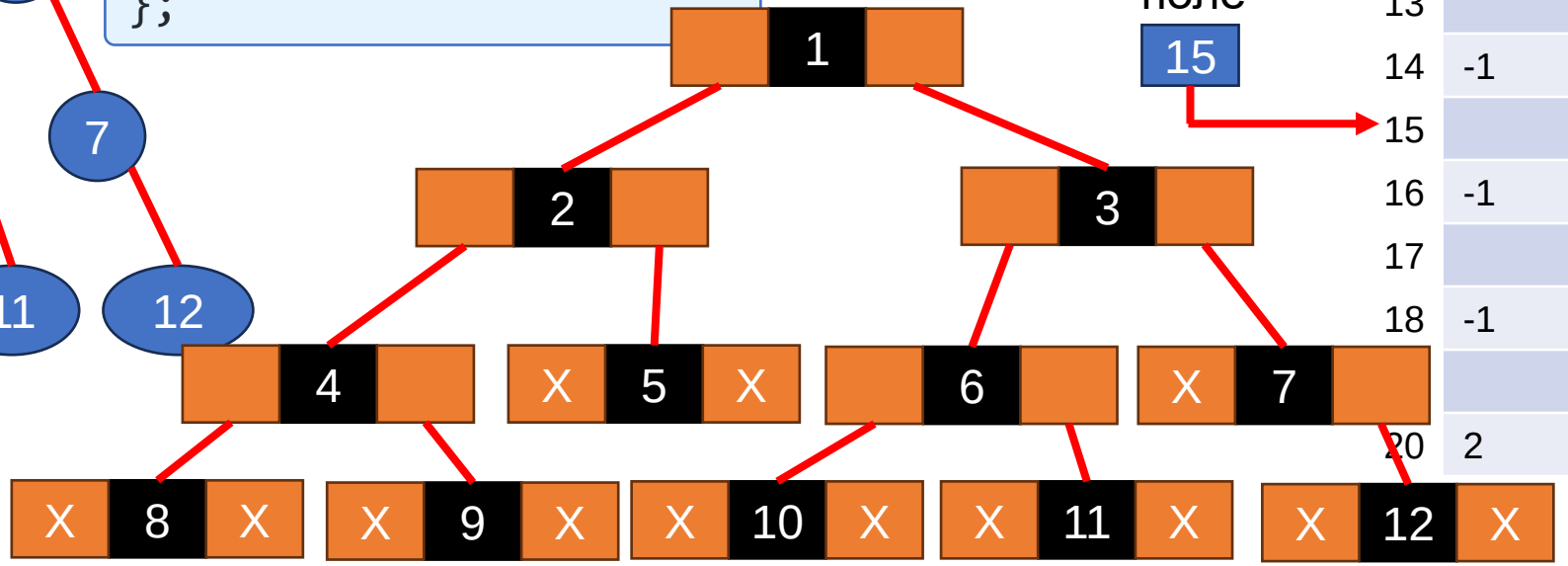


Связанное представление двоичных деревьев

В связанном представлении двоичного дерева каждый узел будет иметь три части: элемент данных, указатель на левый узел и указатель на правый узел.

```
struct node
{
    struct node *left;
    int data;
    struct node *right;
};
```





Корень

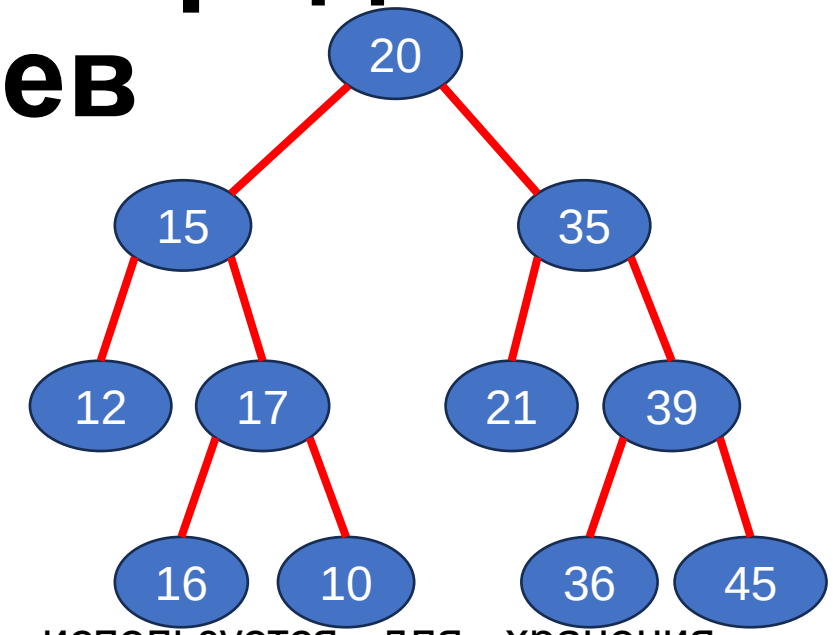
3

Доступное поле

15

	левый	дата	правый
1	-1	8	-1
2	-1	10	-1
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	-1	7	18
12	-1	9	-1
13			
14	-1	5	-1
15			
16	-1	11	-1
17			
18	-1	12	-1
20	2	6	16

Последовательное представление двоичных деревьев



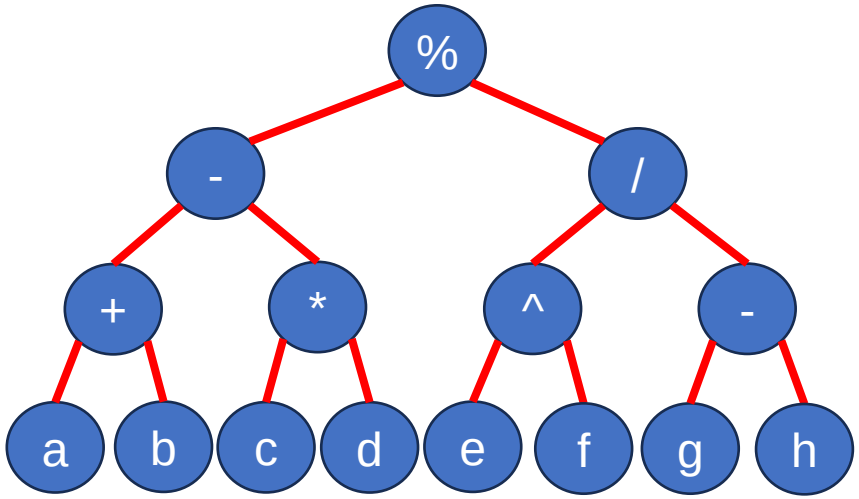
1	20
2	15
3	35
4	12
5	17
6	21
7	39
8	
9	
10	16
11	18
12	
13	
14	36
15	45

Последовательное представление деревьев выполняется с использованием одномерных или одномерных массивов. Хотя это самый простой метод представления в памяти, он неэффективен, так как требует много памяти. Последовательное двоичное дерево следует следующим правилам:

- Одномерный массив, называемый TREE, используется для хранения элементов дерева.
- Корень дерева будет храниться в первом месте. То есть, TREE[1] будет хранить данные корневого элемента.
- Потомки узла, хранящегося в месте K, будут храниться в местах (2 × K) и (2 × K+1).
- Максимальный размер массива TREE задается как (2h–1), где h — высота дерева.
- Пустое дерево или поддерево указывается с помощью NULL. Если TREE[1] = NULL, то дерево пустое.

Пример –дерево выражений

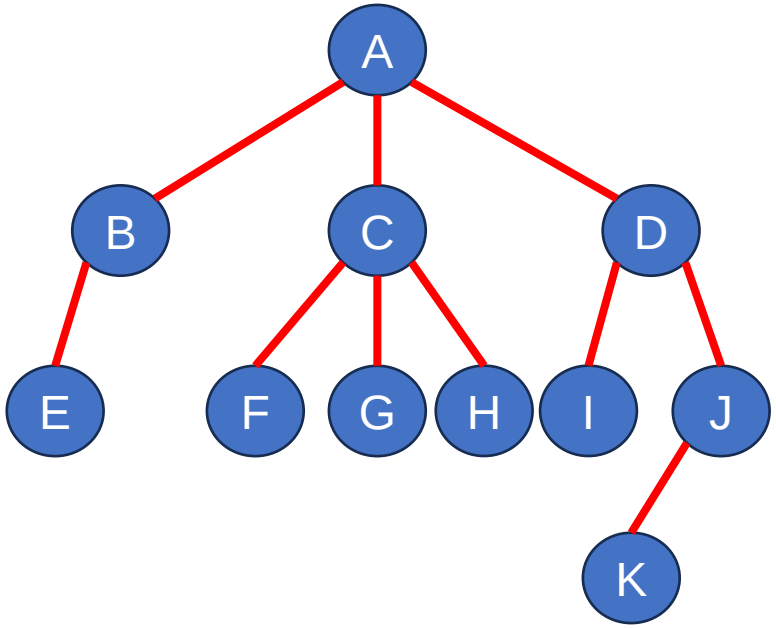
Дано выражение Exp = ((a + b) – (c * d)) % ((e ^ f) / (g – h)), постройте соответствующее бинарное дерево.



Пример – турнирные деревья (НХЛ 23/24)

1/4 финала конференции	1/2 финала конференции	Финал конференции	Финал
Нью-Йорк Рейнджерс 4	Нью-Йорк Рейнджерс 4	Нью-Йорк Рейнджерс 2	Флорида Пантерз 4
Вашингтон Кэпиталз 0	Каролина Харрикейнз 2	Флорида Пантерз 4	Эдмонтон Ойлерз 3
Каролина Харрикейнз 4	Флорида Пантерз 4	Даллас Старз 2	
Нью-Йорк Айлендерс 1	Бостон Брюинз 2	Эдмонтон Ойлерз 4	
Флорида Пантерз 4	Даллас Старз 4		
Тампа-Бэй Лайтнинг 1	Колорадо Эвеланш 2		
Бостон Брюинз 4	Ванкувер Кэнакс 3		
Торонто Мэйпл Лифс 3	Эдмонтон Ойлерз 4		
Даллас Старз 4			
Вегас Голден Найтс 3			
Виннипег Джетс 1			
Колорадо Эвеланш 4			
Ванкувер Кэнакс 4			
Нэшвилл Предаторз 2			
Эдмонтон Ойлерз 4			
Лос-Анджелес Кингз 1			

Создание двоичного дерева из общего дерева



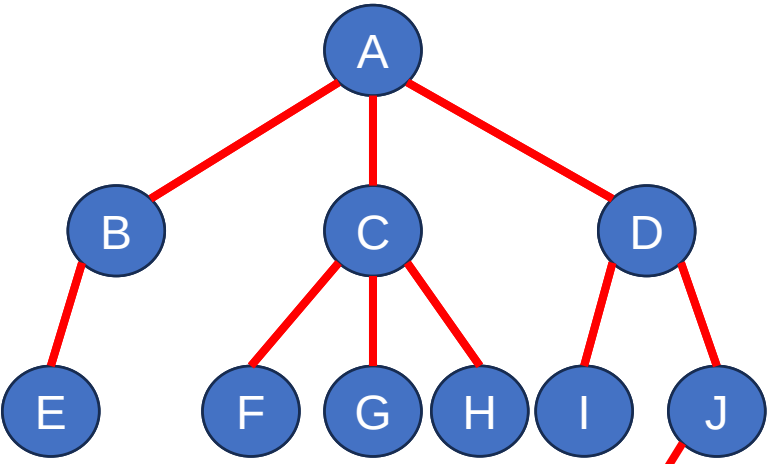
Начинаем с корня по уровням вниз

Правило 1: Корень бинарного дерева = Корень общего дерева

Правило 2: Левый потомок узла = Самый левый потомок узла в бинарном дереве в общем дереве

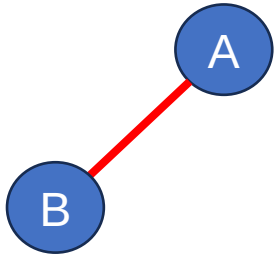
Правило 3: Правый потомок узла в бинарном дереве = Правый брат узла в общем дереве

Создание двоичного дерева из общего дерева

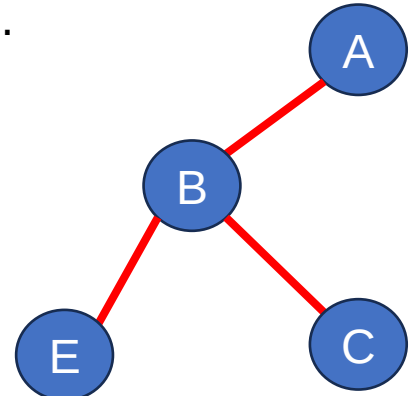


Шаг 1: Узел A является корнем общего дерева, поэтому он также будет корнем бинарного дерева.

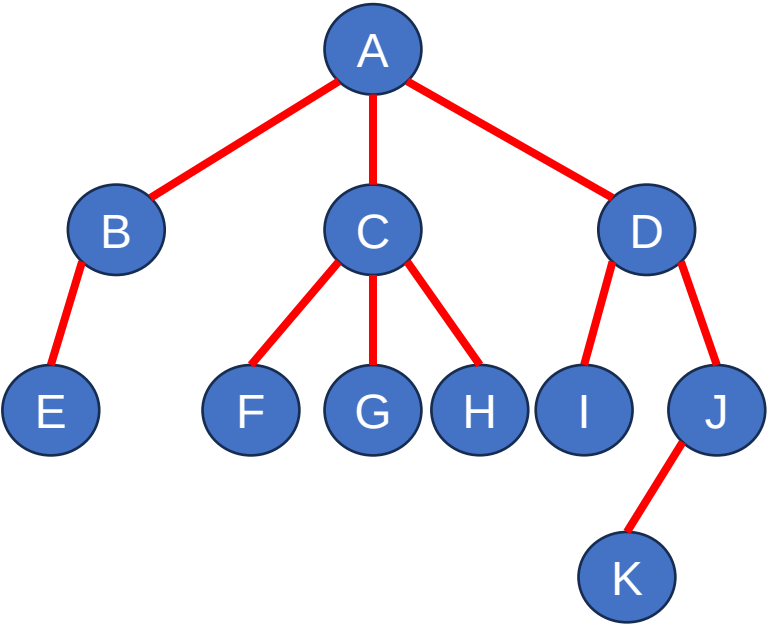
Шаг 2: Левый потомок узла A является самым левым потомком узла A в общем дереве, а правый потомок узла A является правым братом узла A в общем дереве. Поскольку у узла A нет правого брата в общем дереве, у него нет правого брата в бинарном дереве.



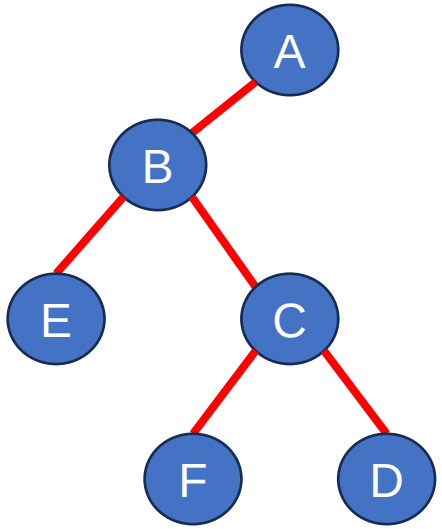
Шаг 3: Теперь обработайте узел B. Левый потомок B — это E, а его правый потомок — это C (правый брат в общем дереве).

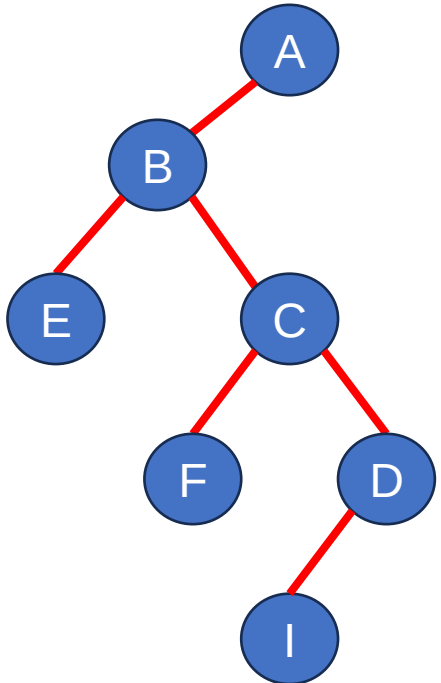


Создание двоичного дерева из общего дерева



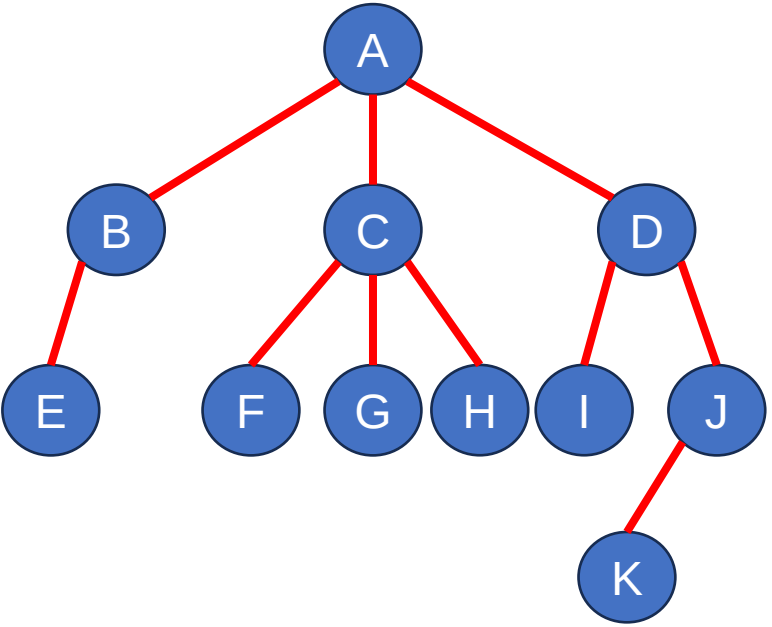
Шаг 4: Теперь обработайте узел C. Левый потомок C — это F (самый левый потомок), а его правый потомок — это D (правый брат в общем дереве).

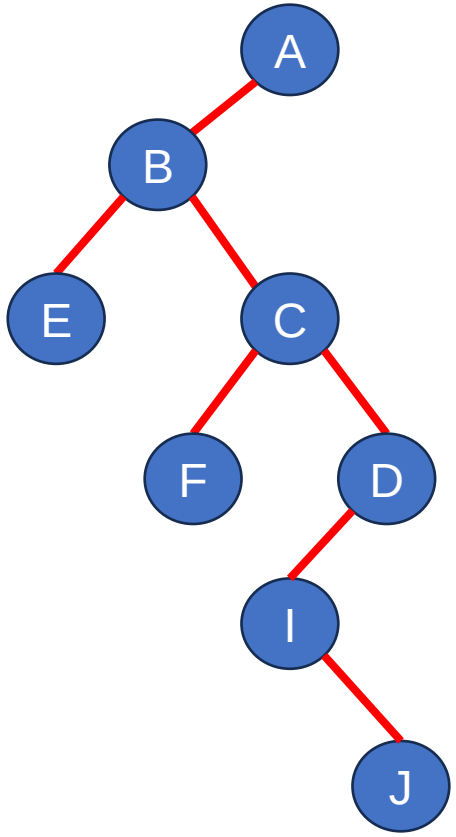




Шаг 5: Теперь обработайте узел D. Левый потомок D — это I (самый левый потомок). Правый потомок D не будет существовать, поскольку у него нет правого брата в общем дереве.

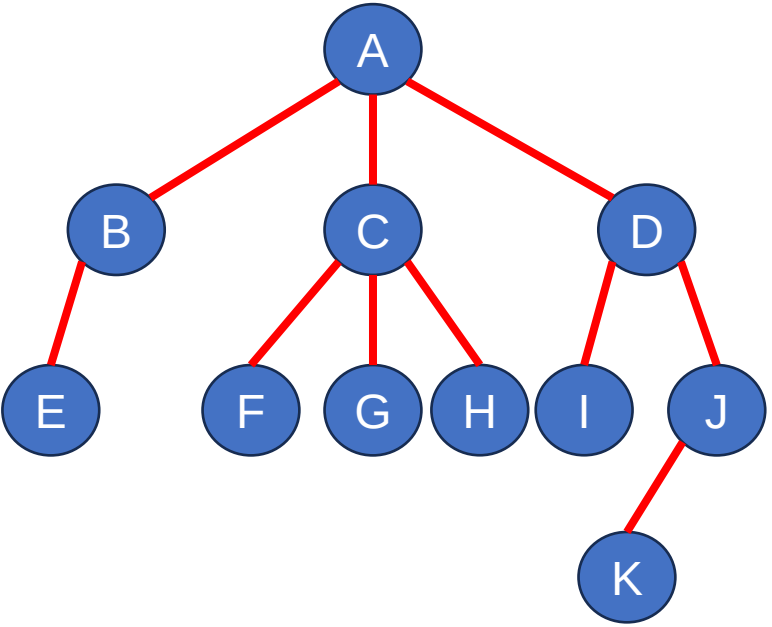
Создание двоичного дерева из общего дерева

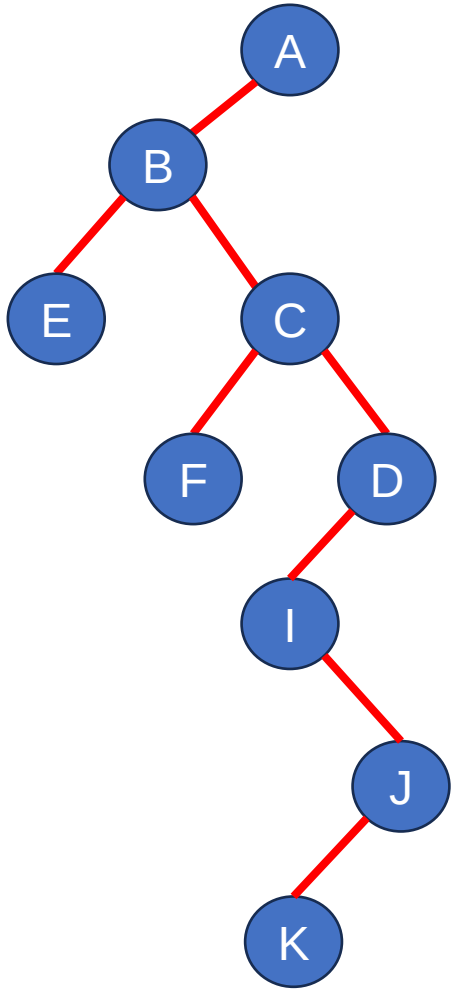




Шаг 6: Теперь обработайте узел I. Левого потомка I в бинарном дереве не будет существовать, поскольку у I нет левого брата в общем дереве. Однако у I есть правый брат J, поэтому он будет добавлен как правый потомок I.

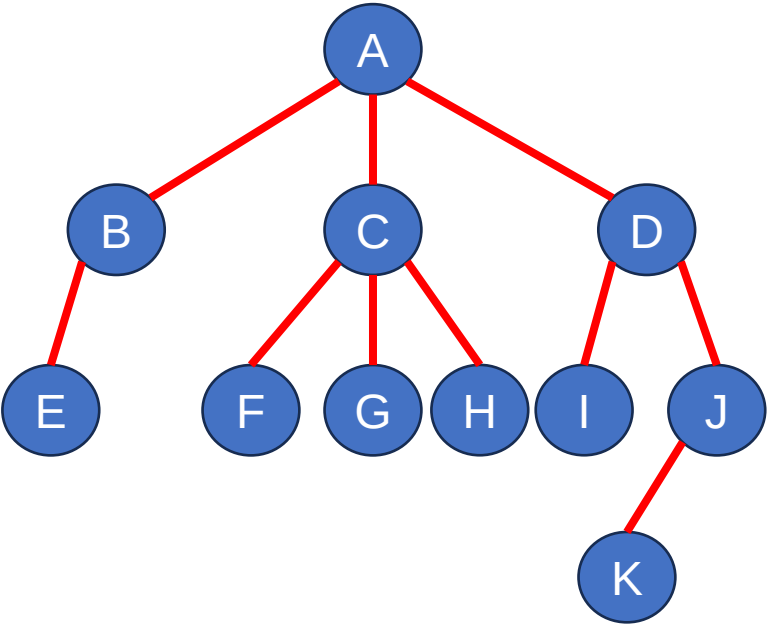
Создание двоичного дерева из общего дерева

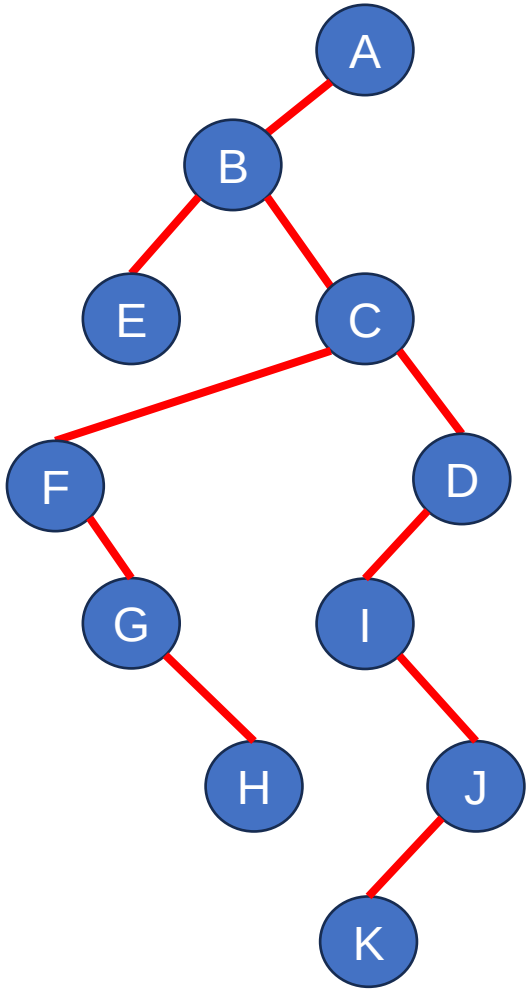




Шаг 7: Теперь обработайте узел J. Левый потомок J — это K (самый левый потомок). Правый потомок J не будет существовать, поскольку у него нет правого брата в общем дереве.

Создание двоичного дерева из общего дерева





Шаг 8: Теперь обработайте все необработанные узлы (E, F, G, H, K) таким же образом, поэтому результирующее бинарное дерево может быть задано следующим образом.

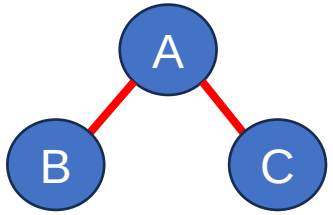
Обход двоичного дерева

Обход бинарного дерева — это процесс посещения каждого узла в дереве ровно один раз систематическим образом. В отличие от линейных структур данных, в которых элементы обходят последовательно, дерево — это нелинейная структура данных, в которой элементы можно обходить множеством различных способов. Существуют различные алгоритмы обхода дерева:

1. Префиксный (в прямом порядке)
2. Инфиксный (симметричный)
3. Постфиксный (в обратном порядке)
4. Поуровневый (в ширину)

Эти алгоритмы различаются порядком посещения узлов.

Префиксный обход



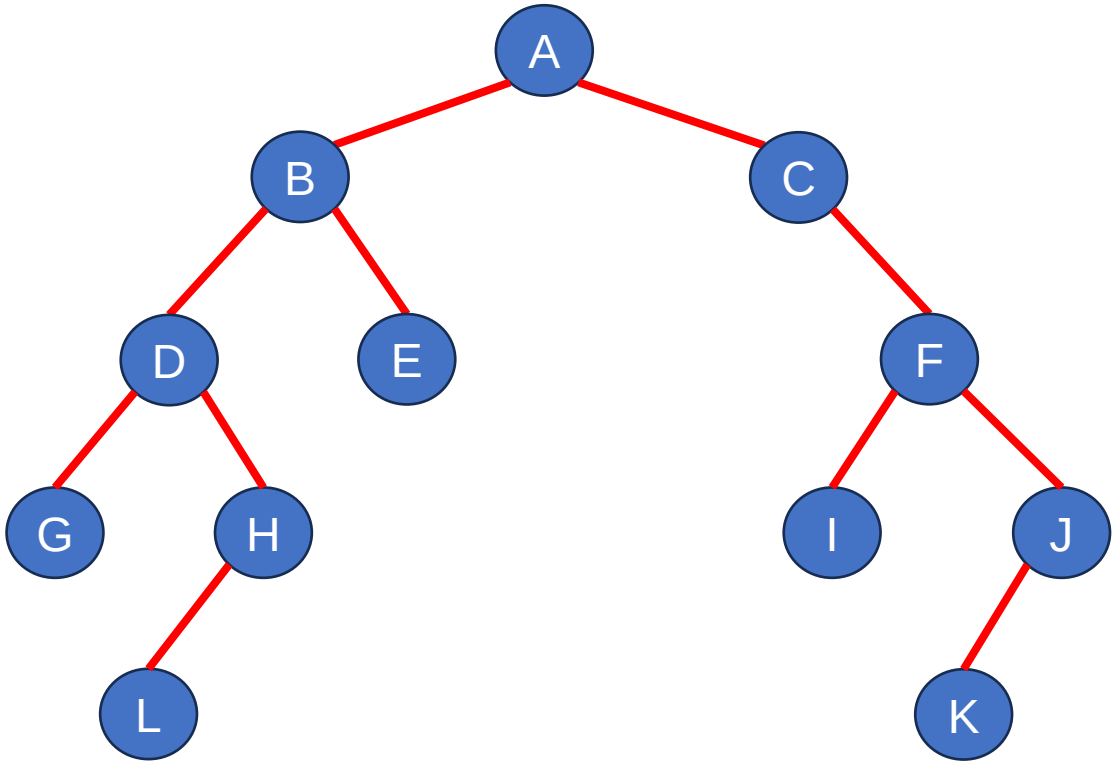
Чтобы обойти непустое двоичное дерево в прямом порядке, в каждом узле рекурсивно выполняются следующие операции. Алгоритм работает следующим образом:

1. Посещение корневого узла,
2. Обход левого поддерева и, наконец,
3. Обход правого поддерева.

Алгоритм для предварительного порядка обхода:

Шаг 1: Повторите шаги 2–4, пока TREE != NULL
Шаг 2: Запишите TREE -> DATA
Шаг 3: PREORDER(TREE -> LEFT)
Шаг 4: PREORDER(TREE -> RIGHT)
[END OF LOOP]
Шаг 5: END

Префиксный обход

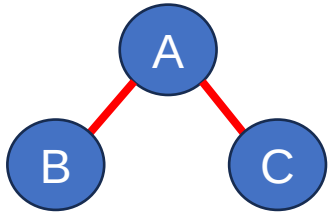


ПОРЯДОК ОБХОДА: A, B, D, G, H, L, E, C, F, I, J и K

Когда мы обходим элементы дерева с помощью алгоритма обхода в прямом порядке, выражение, которое мы получаем, является префиксным выражением.

^ + / a b * c d / % f g – h i

Инфиксный обход



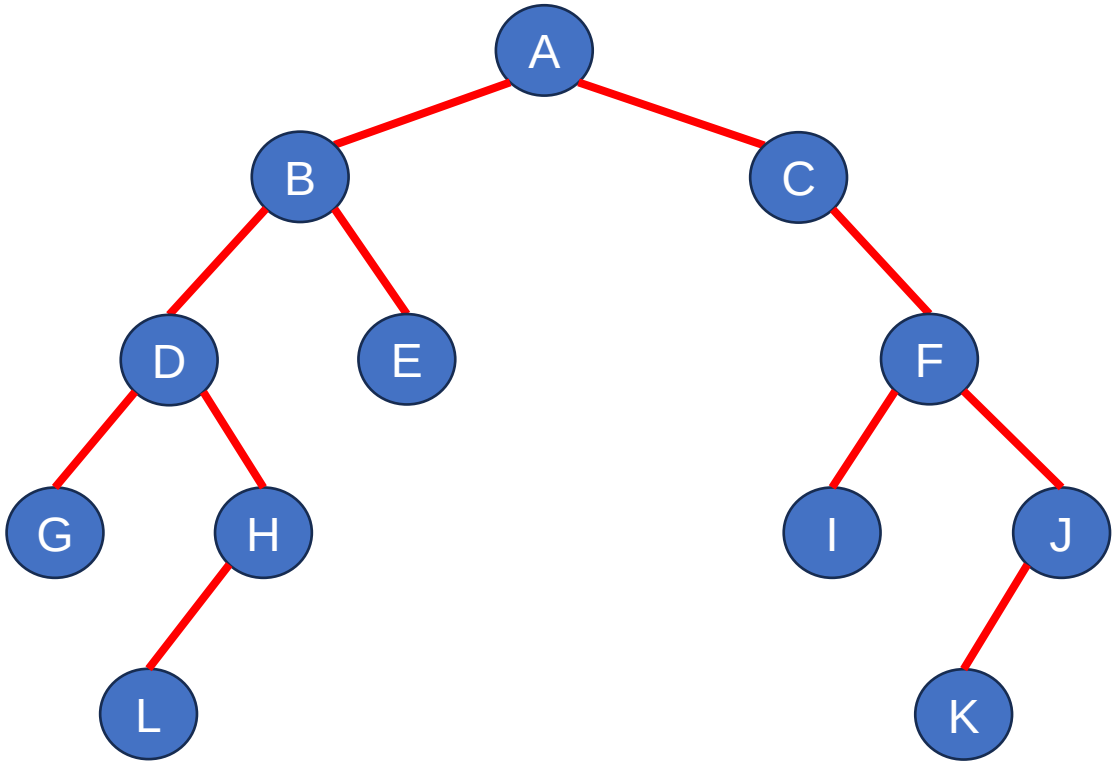
Чтобы обойти непустое двоичное дерево в прямом порядке, в каждом узле рекурсивно выполняются следующие операции. Алгоритм работает следующим образом:

1. Посещение корневого узла,
2. Обход левого поддерева и, наконец,
3. Обход правого поддерева.

Алгоритм для предварительного порядка обхода:

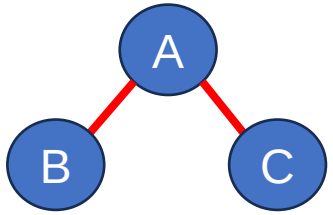
Шаг 1: Повторите шаги 2–4, пока TREE != NULL
Шаг 2: Запишите TREE -> DATA
Шаг 3: PREORDER(TREE -> LEFT)
Шаг 4: PREORDER(TREE -> RIGHT)
[END OF LOOP]
Шаг 5: END

Инфиксный обход



ПОРЯДОК ОБХОДА: G, D, L, H, B, E, A, C, I, F, K и J

Постфиксный обход

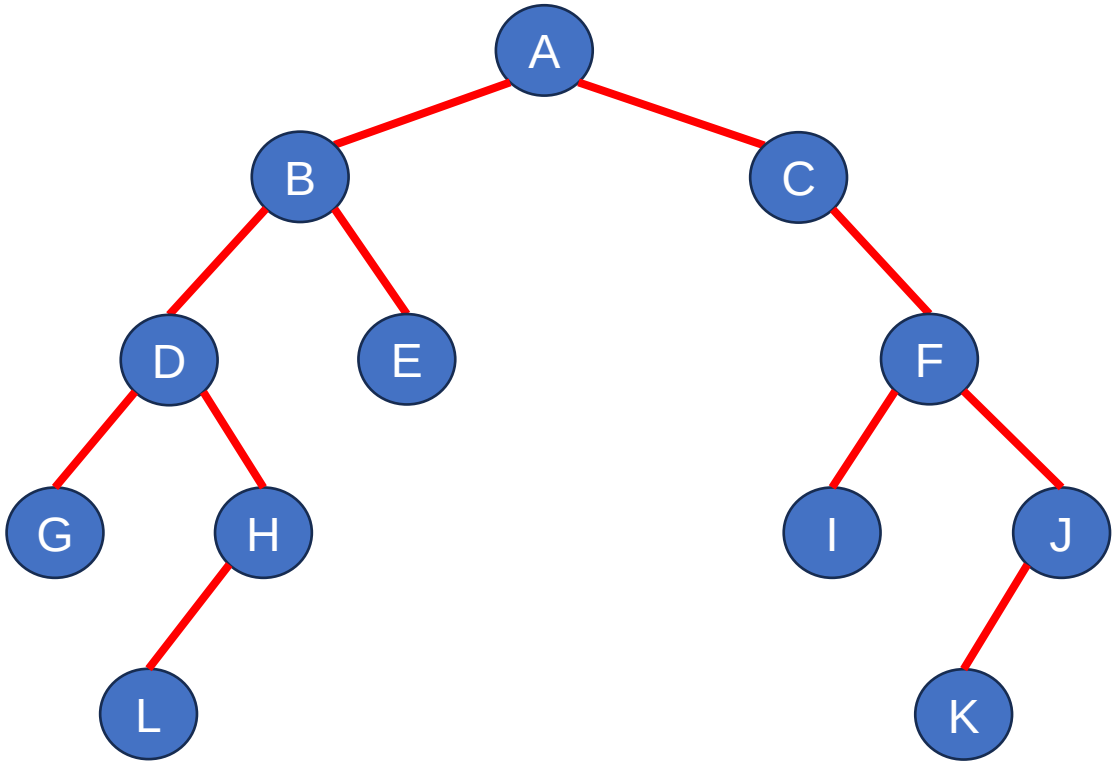


Для обхода непустого бинарного дерева в прямом порядке следующие операции выполняются рекурсивно в каждом узле. Алгоритм работает следующим образом:

1. Обход левого поддерева,
2. Обход правого поддерева и, наконец,
3. Посещение корневого узла.

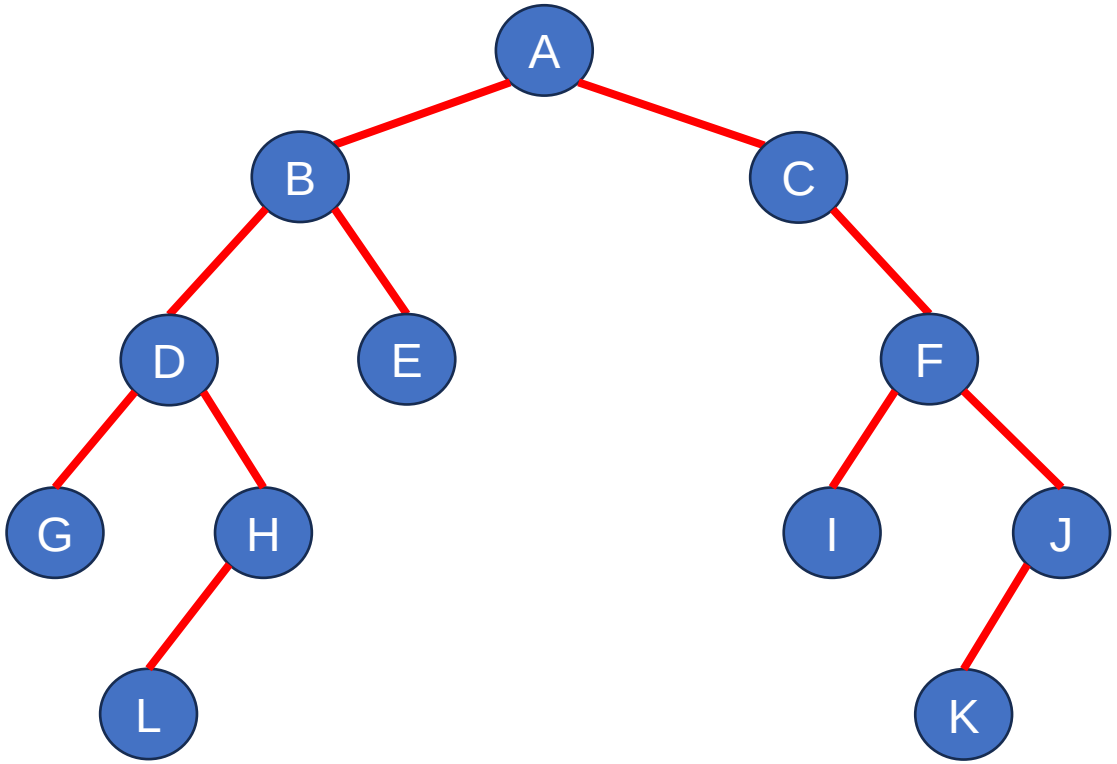
Шаг 1: Повторите шаги 2–4 while TREE != NULL
Шаг 2: POSTORDER(TREE-> LEFT)
Шаг 3: POSTORDER(TREE-> RIGHT)
Шаг 4: Запись TREE-> DATA
[КОНЕЦ ЦИКЛА]
Шаг 5: КОНЕЦ

Постфиксный обход



ПОРЯДОК ОБХОДА: G, L, H, D, E, B, I, K, J, F, C и A

Обход в ширину



ПОРЯДОК ОБХОДА: A, B, C, D, E, F, G, H, I, J, L и K

Обход в ширину

```
DepthTree(struct node *tree, struct queue *q)
{
    // Помещаем корень в очередь.
    q = insert(q, tree);
    // Обрабатываем очередь, пока она не станет пустой.
    struct node *node = tree;
    while (q->front != NULL)
    {
        // Получаем следующую вершину в очереди.
        struct queue *p = delete_element(q);
        // Обрабатываем вершину.
        //....
        // Добавляем дочернюю вершину в очередь.
        if(p->front->node->left != NULL)
            q = insert(q, p->front->node->left);
        if (p->front->node->right != NULL)
            q = insert(q, p->front->node->right);
    }
}
```

Построение бинарного дерева из результатов обхода

Мы можем построить бинарное дерево, если нам даны по крайней мере два результата обхода. Первый обход должен быть инфиксным, а второй может быть либо префиксным, либо постфиксным. Результат инфиксного обхода будет использоваться для определения левого и правого дочерних узлов, а префиксный/постфиксный может использоваться для определения корневого узла.

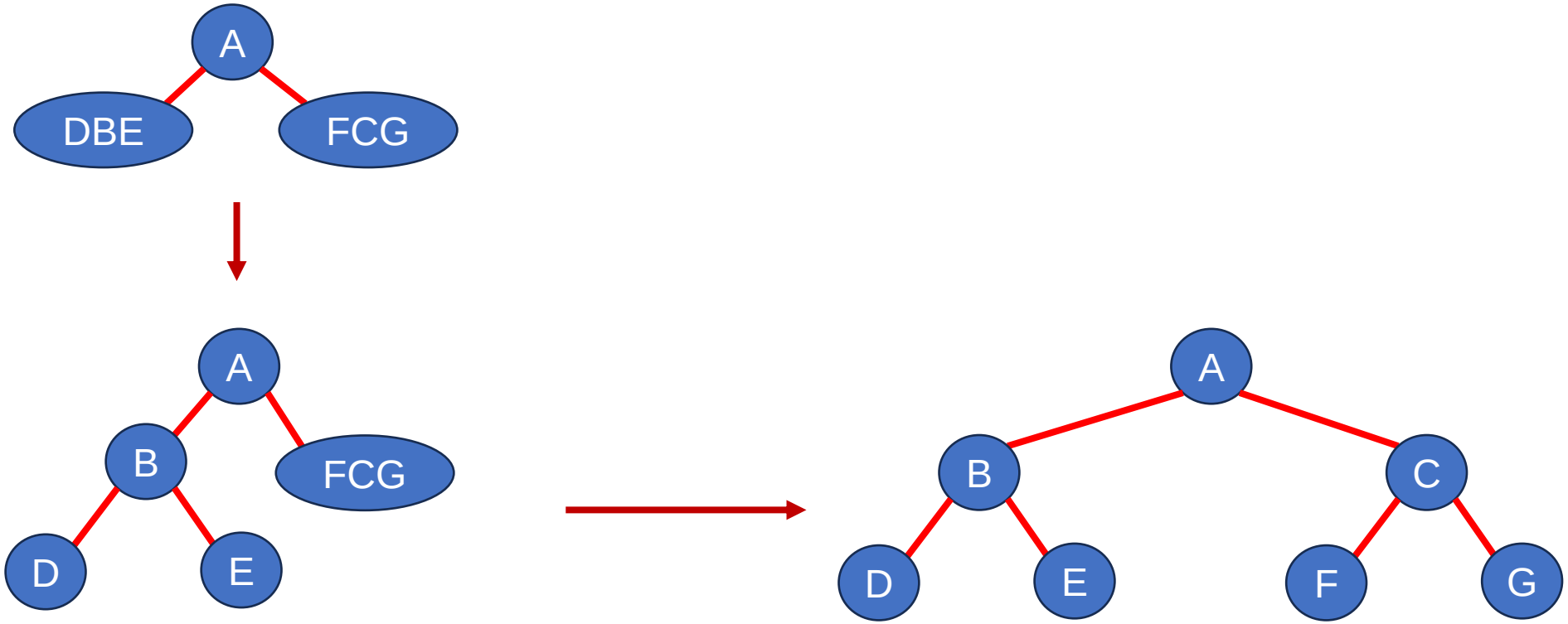
Шаг 1 Используйте последовательность в префиксном /постфиксном порядке для определения корневого узла дерева. Первым элементом будет корневой узел.

Шаг 2 Элементы с левой стороны корневого узла в последовательности обхода в инфиксном порядке образуют левое поддерево корневого узла. Аналогично элементы с правой стороны корневого узла в последовательности обхода в инфиксном порядке образуют правое поддерево корневого узла.

Шаг 3 Рекурсивно выбираем каждый элемент из последовательности обхода в прямом порядке и создаем его левое и правое поддеревья из последовательности обхода в прямом порядке.

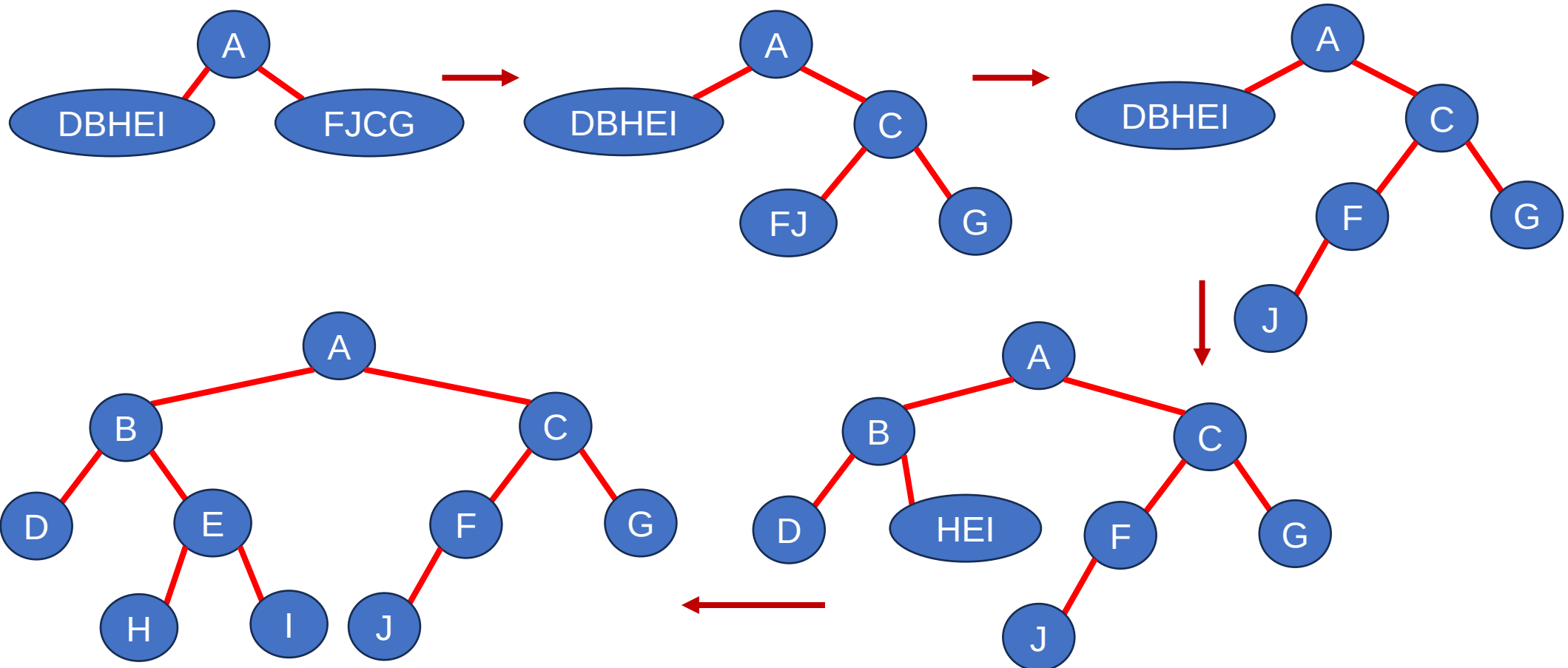
Построение бинарного дерева из результатов обхода - пример

Обход в инфиксном порядке: D B E A F C G
Обход в префиксном порядке: A B D E C F G



Построение бинарного дерева из результатов обхода – пример 2

Обход в инфиксном порядке: D B H E I A F J C G
Обход в постфиксном порядке: D H I E B J F G C A



Время выполнения обхода

Алгоритмы, выполняющие обход дерева симметрично, в прямом и обратном порядках, следуют вниз до терминальных вершин, возвращаясь вверх к корню по мере разворачивания рекурсивных вызовов. После того как алгоритм прошел по вершине и вернулся к родителю, он больше к ней не обращается, то есть посещает ее всего один раз. Таким образом, для дерева из N вершин время работы этих алгоритмов составит O(N).

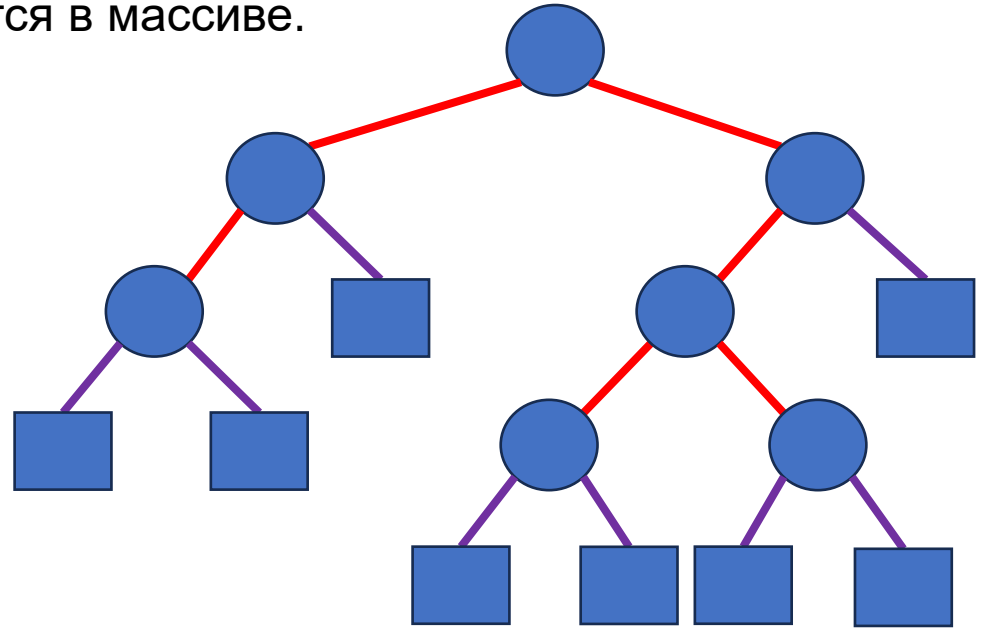
Упомянутые три обхода не требуют дополнительного пространства на диске, поскольку они используют структуру дерева для отслеживания своего местоположения в нем. Тем не менее их глубина рекурсии равна высоте дерева, и если дерево окажется очень высоким, вероятно переполнение стека.

При обходе в ширину вершины обрабатываются по мере их постановки в очередь. Поскольку каждая вершина помещается туда единожды, для дерева из N вершин время работы алгоритма будет равным O(N).

Этот алгоритм не использует рекурсию, а значит, не зависит от ее глубины, но нуждается в дополнительном пространстве для построения очереди. В наихудшем случае, если дерево является идеальным бинарным, то есть на его нижнем уровне находится практически половина всех вершин, в очереди окажется O(N/2) = O(N) вершин.

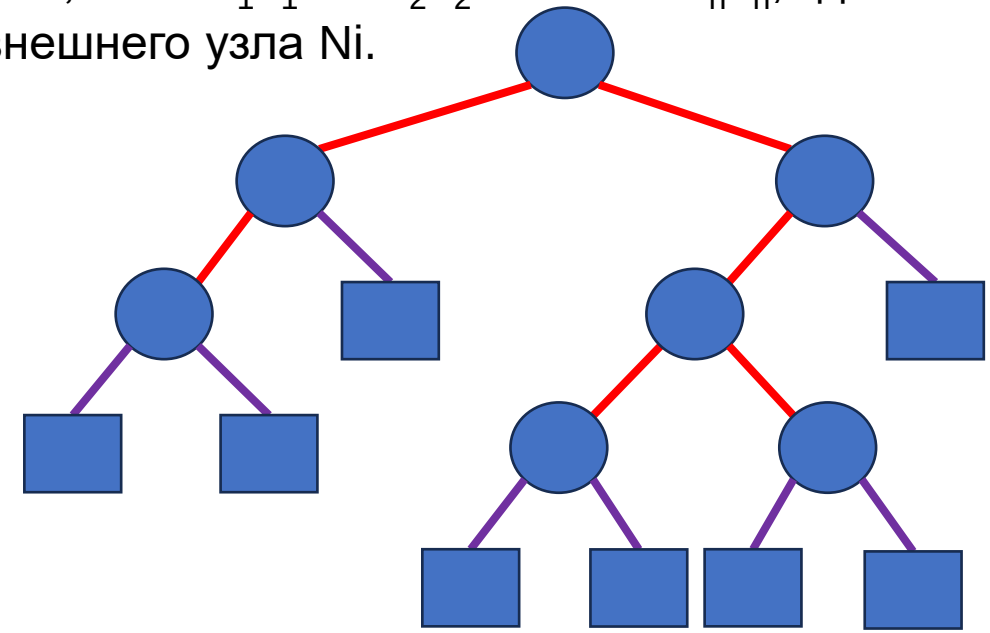
Дерево Хаффмана

Кодирование Хаффмана — это алгоритм энтропийного кодирования, разработанный Дэвидом А. Хаффманом, который широко используется ***как метод сжатия данных без потерь***. Алгоритм кодирования Хаффмана использует таблицу кодов переменной длины для кодирования исходного символа, где таблица кодов переменной длины выводится на основе предполагаемой вероятности появления исходного символа. Основная идея алгоритма Хаффмана заключается в том, что он кодирует наиболее распространенные символы, используя более короткие строки битов, чем те, которые используются для менее распространенных исходных символов. Алгоритм работает путем создания двоичного дерева узлов, которые хранятся в массиве.

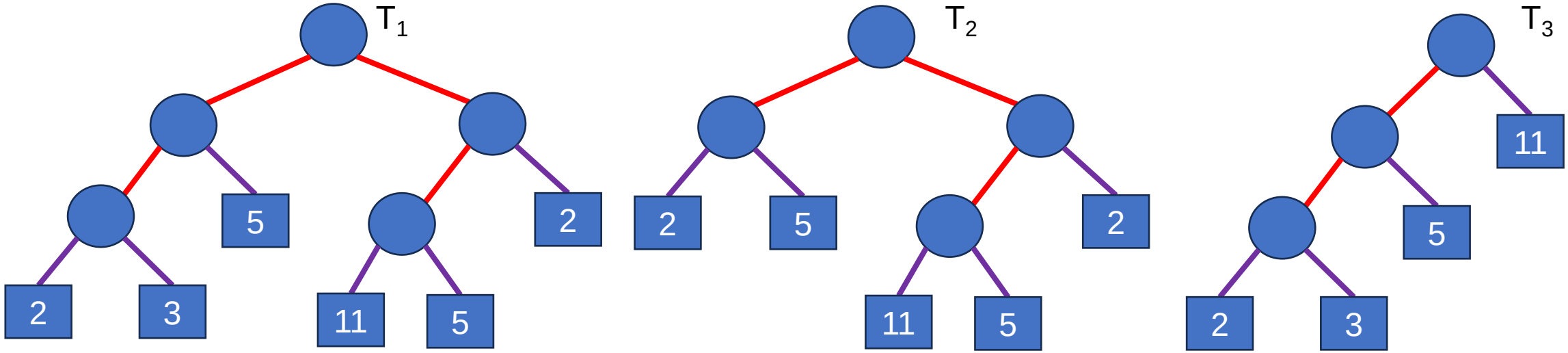


Дерево Хаффмана

Внутренняя длина пути, $L_I = 0 + 1 + 2 + 1 + 2 + 3 + 3 = 12$
Внешняя длина пути, $L_E = 2 + 3 + 3 + 2 + 4 + 4 + 4 + 4 = 26$
Обратите внимание, что $L_I + 2 * n = 12 + 2 * 7 = 12 + 14 = 26 = L_E$
Таким образом, $L_I + 2n = L_E$, где n — количество внутренних узлов.
Теперь, если дерево имеет n внешних узлов и каждому внешнему узлу назначен вес, то взвешенная длина пути P определяется как сумма взвешенных длин путей.
Следовательно, $P = W_1L_1 + W_2L_2 + \ldots. + W_nL_n$, где Wi и Li — вес и длина пути внешнего узла Ni.



Дерево Хаффмана



Взвешенная длина внешнего пути $\mathbf{T_1}$ может быть задана как,
$\mathbf{P_1} = 2 \cdot 3 + 3 \cdot 3 + 5 \cdot 2 + 11 \cdot 3 + 5 \cdot 3 + 2 \cdot 2 = 6 + 9 + 10 + 33 + 15 + 4 = 77$
Взвешенная длина внешнего пути $\mathbf{T_2}$ может быть задана как,
$\mathbf{P_2} = 5 \cdot 2 + 7 \cdot 2 + 3 \cdot 3 + 4 \cdot 3 + 2 \cdot 2 = 10 + 14 + 9 + 12 + 4 = 49$
Взвешенная длина внешнего пути $\mathbf{T_3}$ может быть задана как,
$\mathbf{P_3} = 2 \cdot 3 + 3 \cdot 3 + 5 \cdot 2 + 11 \cdot 1 = 6 + 9 + 10 + 11 = 36$

Метод

При наличии n узлов и их весов алгоритм Хаффмана используется для поиска дерева с минимальной длиной пути. Процесс по сути начинается с создания нового узла, дочерними узлами которого являются два узла с наименьшим весом, так что вес нового узла равен сумме весов дочерних узлов. То есть два узла объединяются в один узел. Этот процесс повторяется до тех пор, пока в дереве не останется только один узел. Такое дерево с одним узлом известно как дерево Хаффмана. Алгоритм Хаффмана может быть реализован с использованием очереди приоритетов, в которой все узлы размещаются таким образом, что узлу с наименьшим весом присваивается наивысший приоритет.

Шаг 1: Создание конечных узлов для каждого символа. Добавьте символ и его вес или частоту появления в очередь приоритетов.

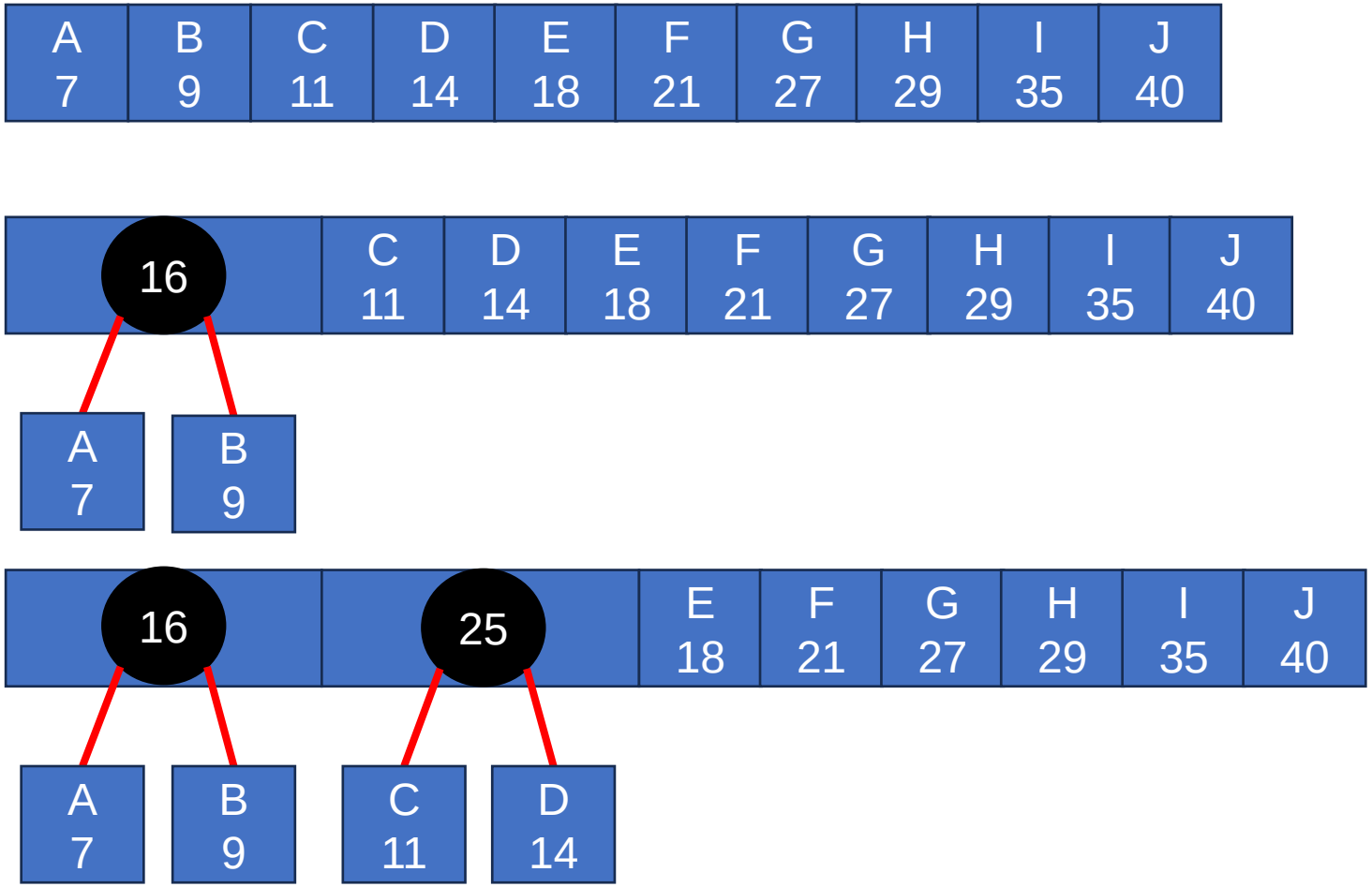
Шаг 2: Повторите шаги 3–5, пока общее количество узлов в очереди больше 1.

Шаг 3: Удалите два узла с самым низким весом (или самым высоким приоритетом).

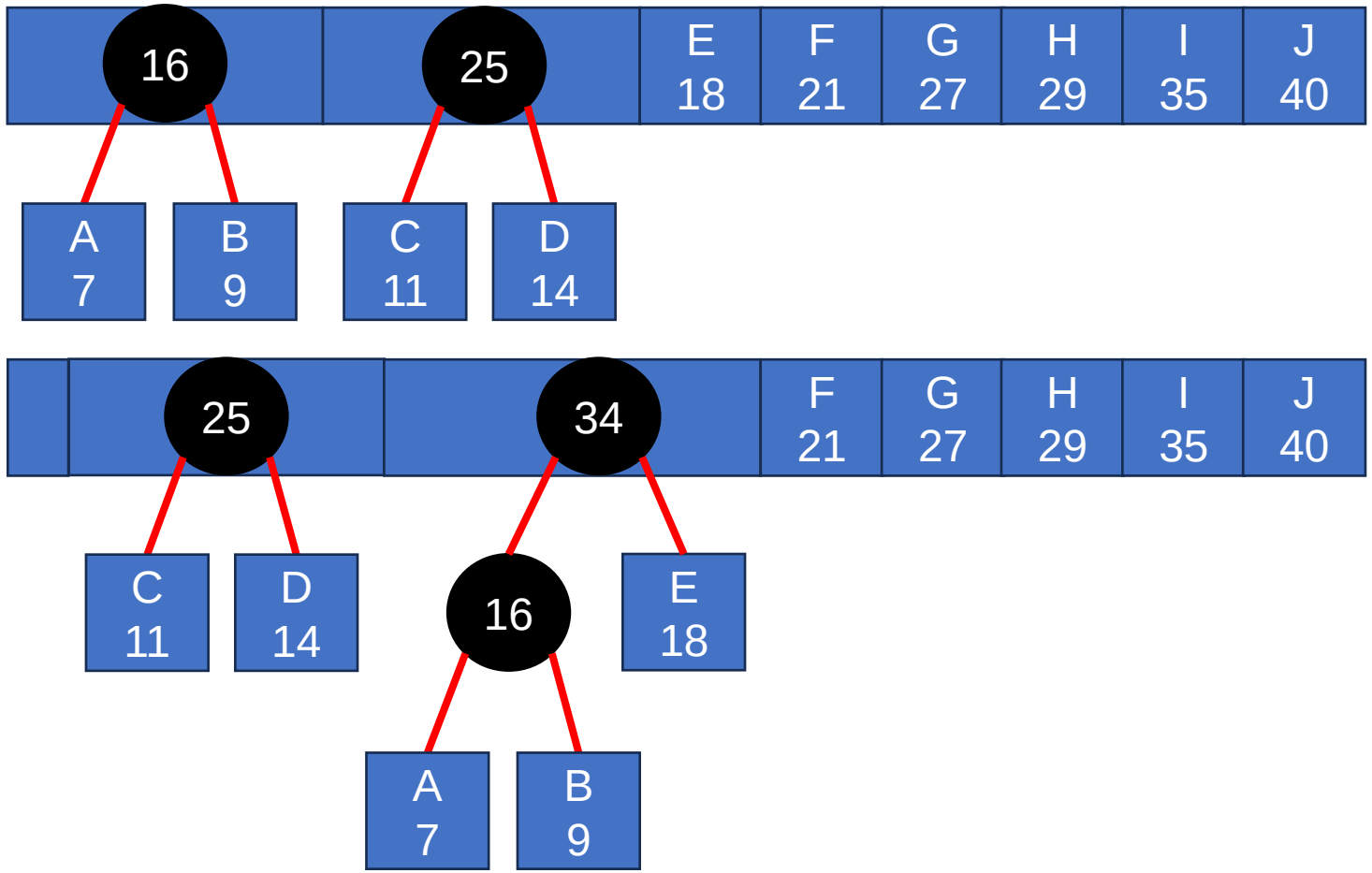
Шаг 4: Создайте новый внутренний узел, объединив эти два узла как дочерние и с весом, равным сумме весов двух узлов.

Шаг 5: Добавьте вновь созданный узел в очередь.

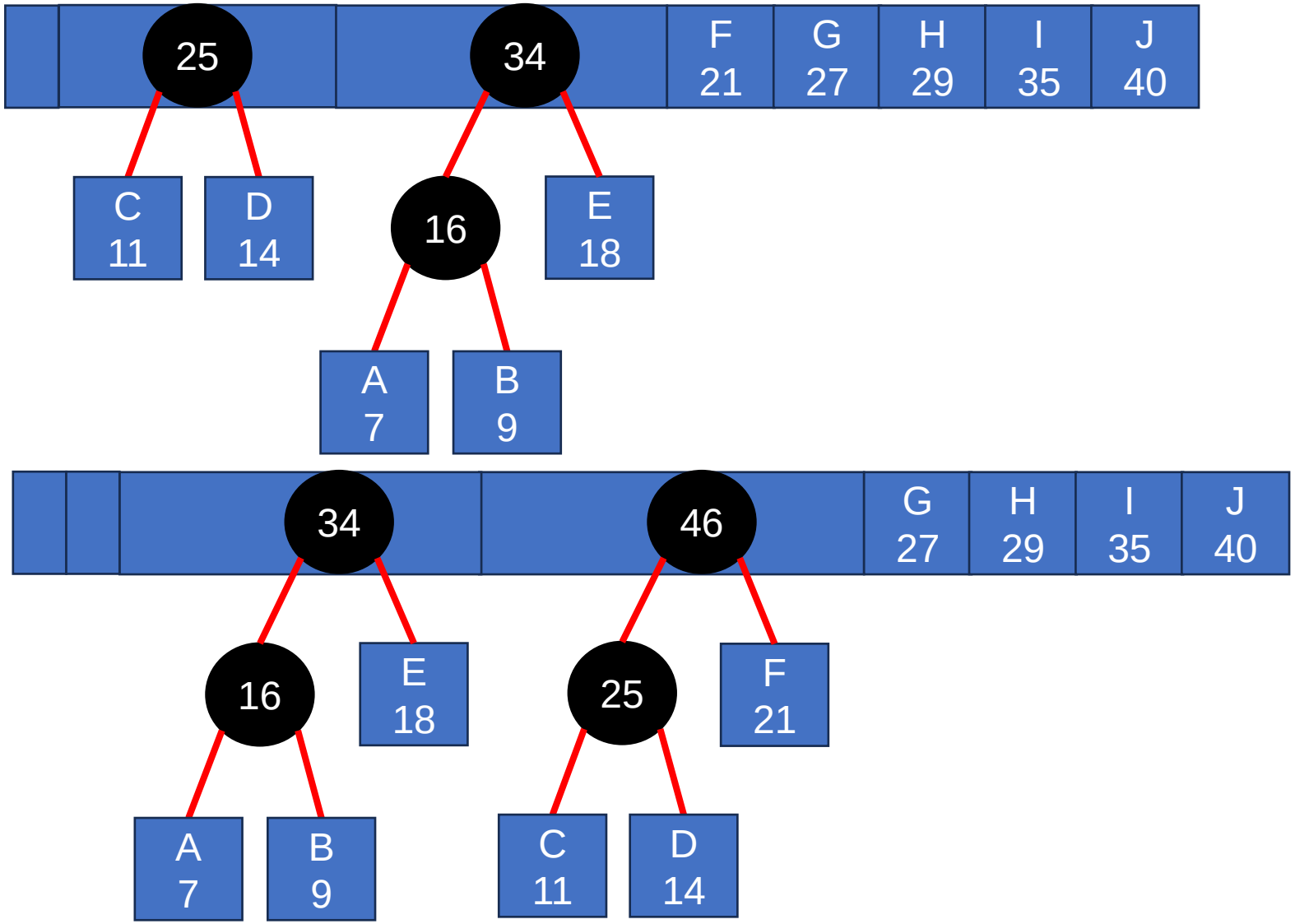
Метод (пример)



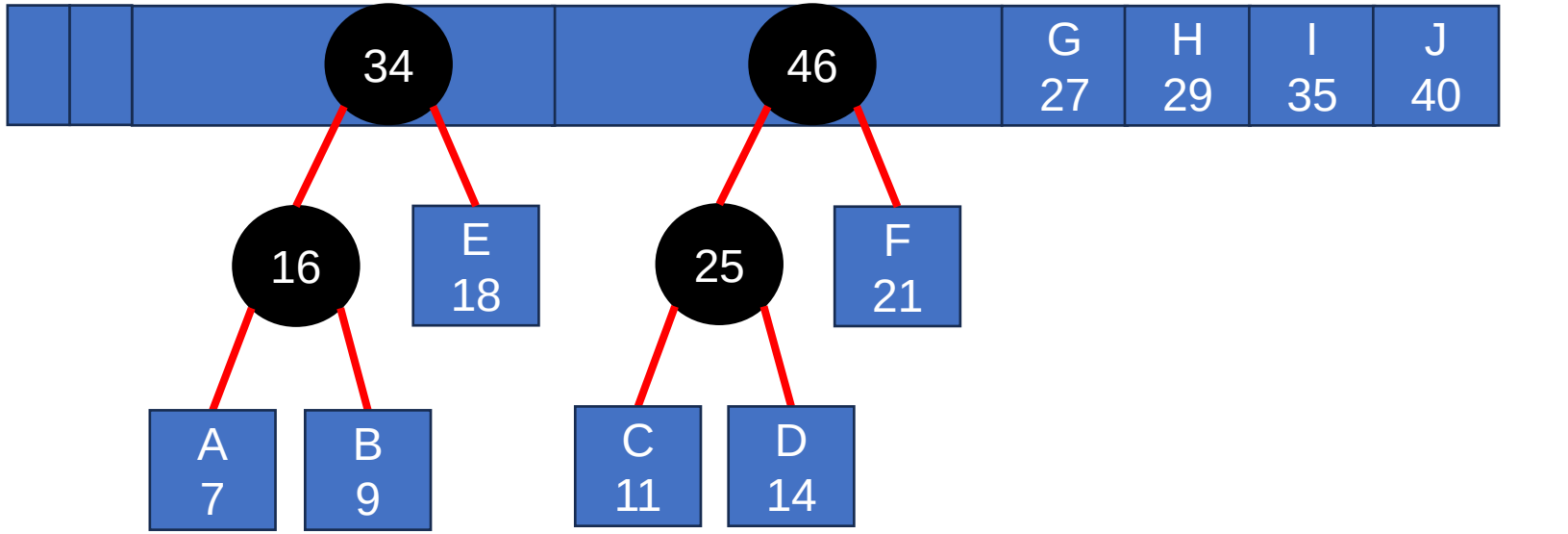
Метод (пример)

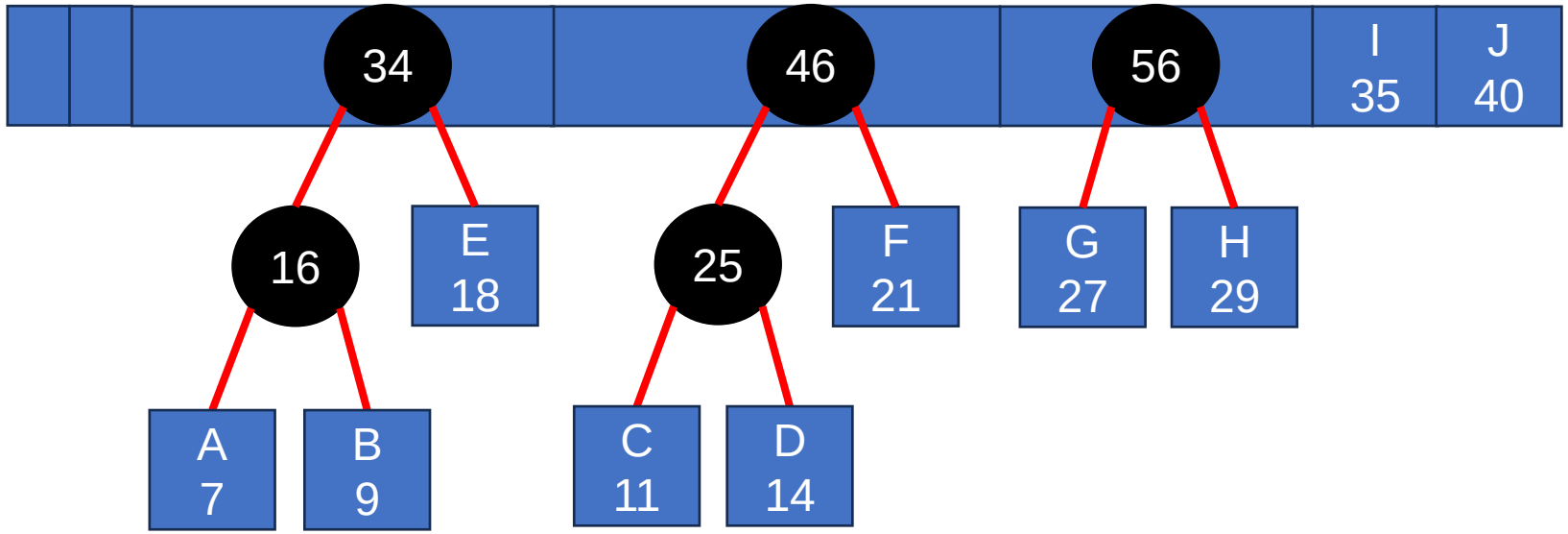


Метод (пример)

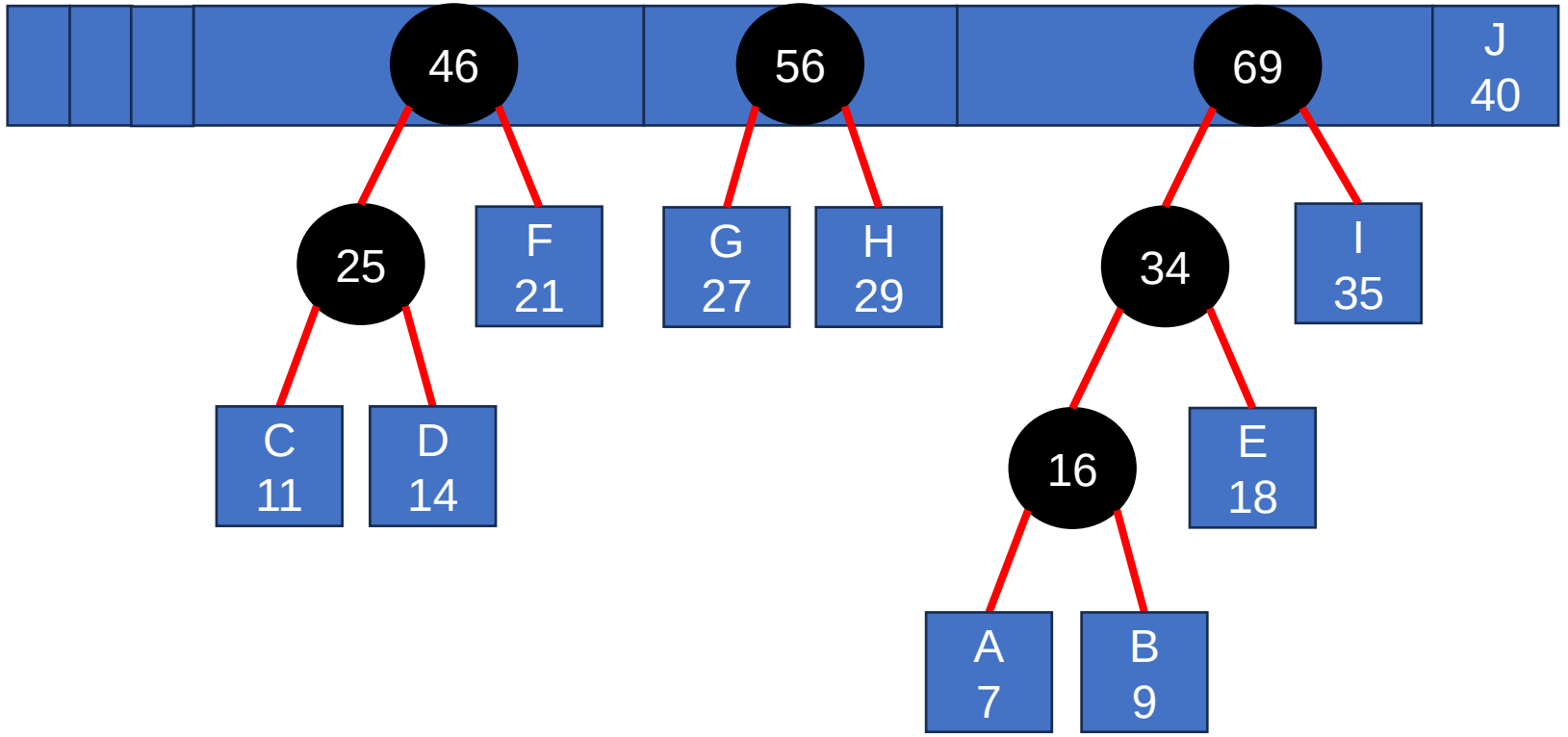


Метод (пример)

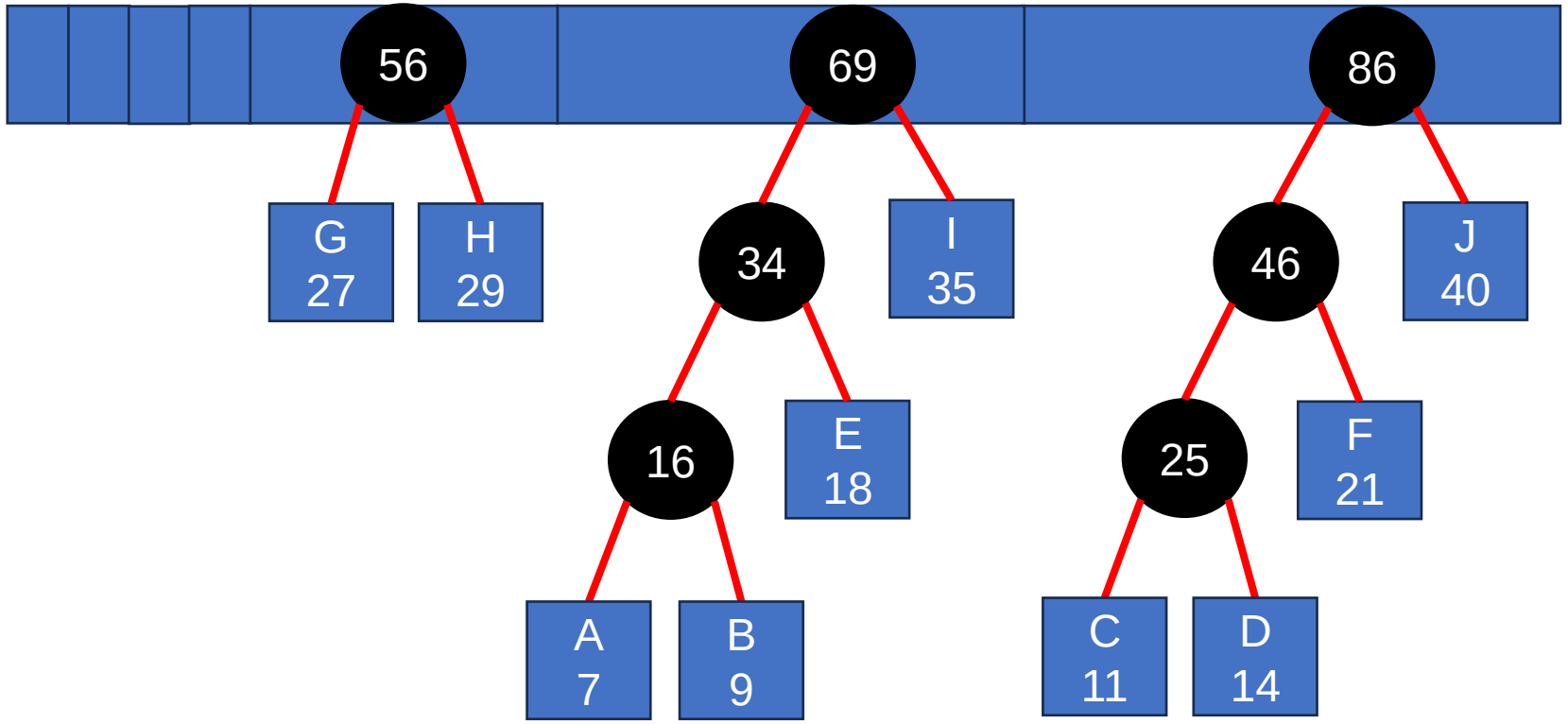




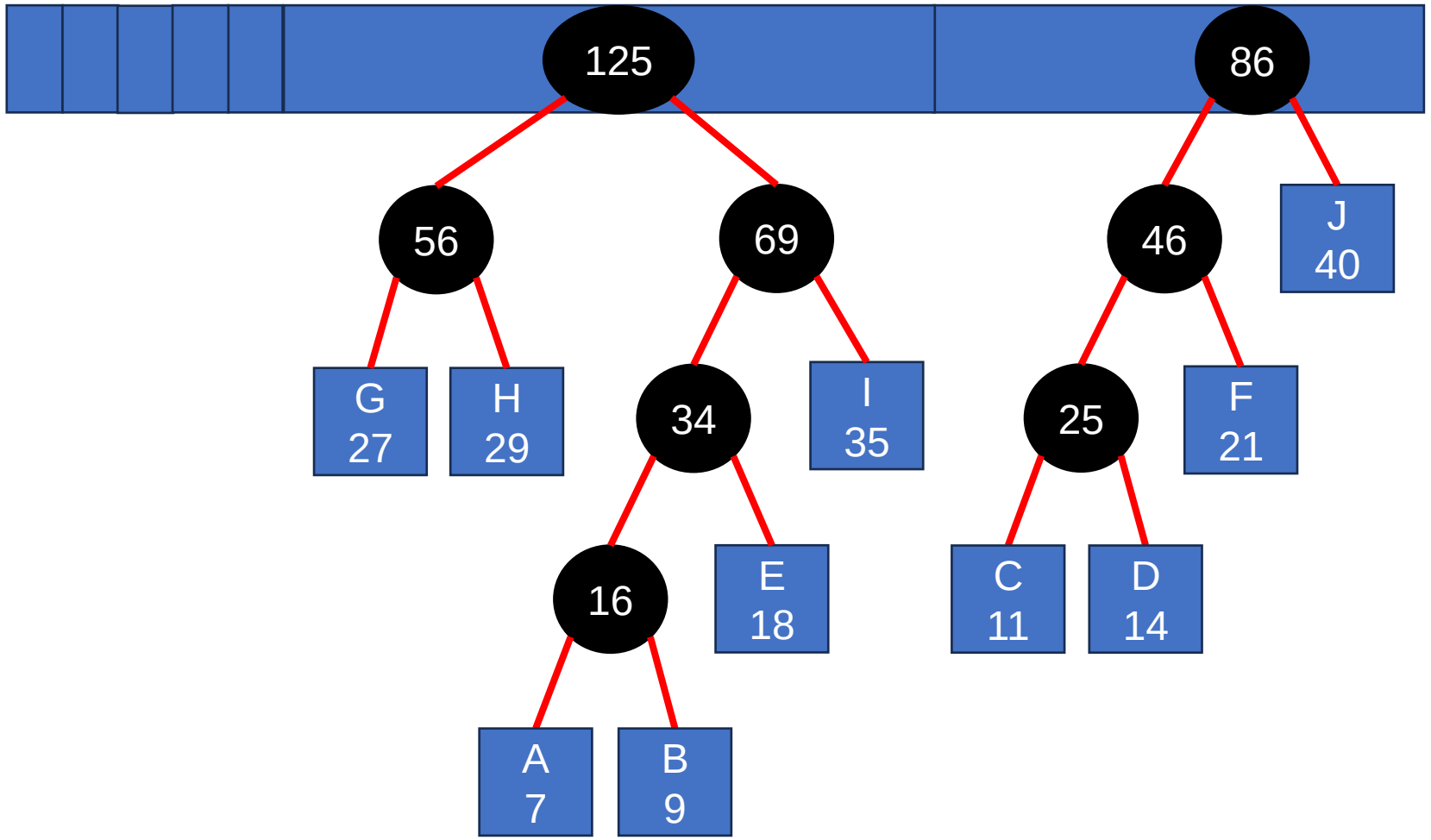
Метод (пример)



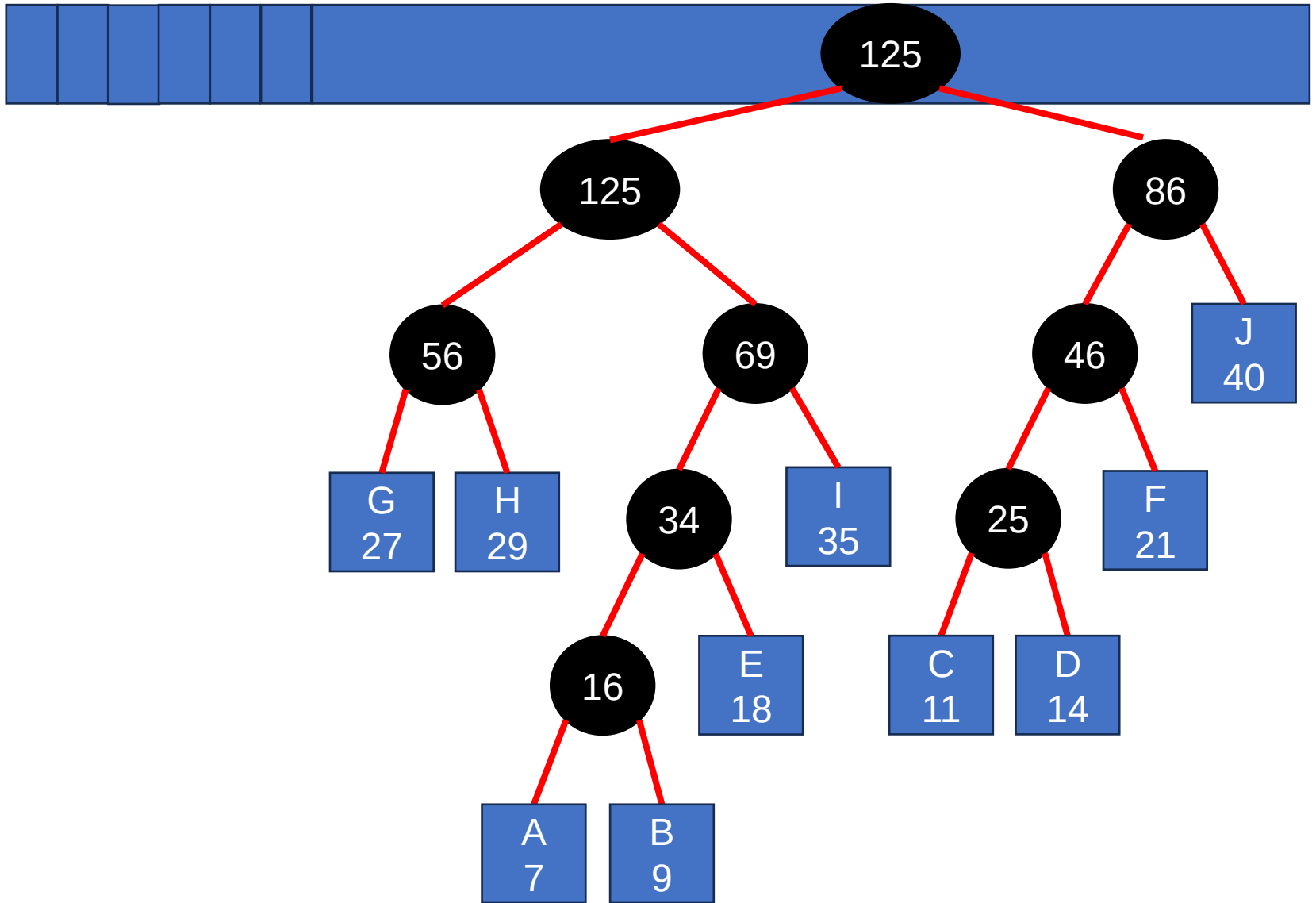
Метод (пример)



Метод (пример)



Метод (пример)



Кодирование данных

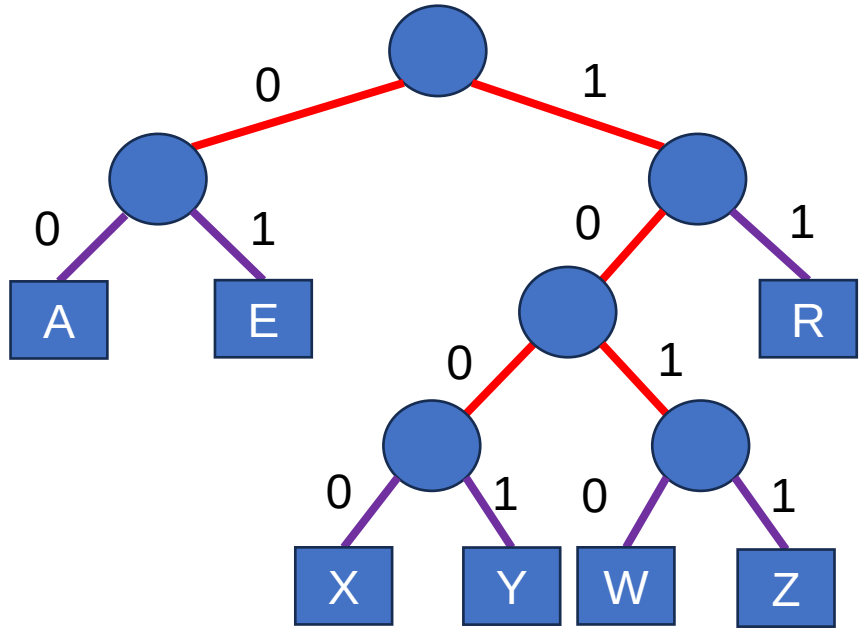
Код	Символ
000	A
001	B
010	C
011	D
100	E
101	F
110	G
111	H

Когда мы хотим закодировать наши данные (символ) с помощью битов, то мы используем r битов для кодирования 2^r символов. Например, если r=1, то можно закодировать два символа. Если эти два символа — A и B, то A можно закодировать как 0, а B можно закодировать как 1 и наоборот.

Теперь, если нам нужно закодировать строку данных ABBBBBBAAAACDEFGGGGH, то соответствующий код будет следующим:
00000100100100100100100000000000010011100101110110110111

Кодирование данных - пример

Символ	Код
A	00
E	01
R	11
W	1010
X	1000
Y	1001
Z	1011



Таким образом, мы видим, что частые символы имеют более короткий код, а редкие символы имеют более длинный код.

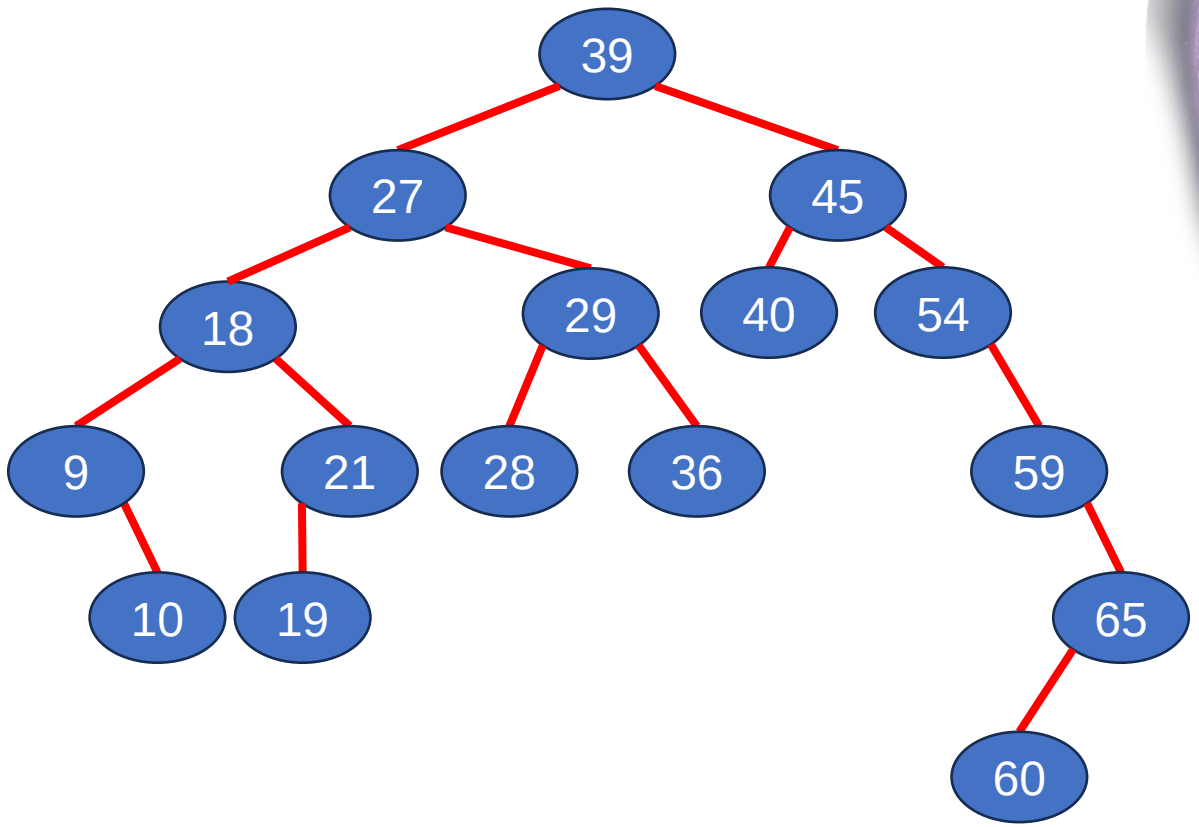
Применение деревьев

Деревья используются для хранения как простых, так и сложных данных. Здесь простое означает целочисленное значение, символьное значение, а сложные данные означают структуру или запись.

- Деревья часто используются для реализации других типов структур данных, таких как хэш-таблицы, наборы и карты.
- Самобалансирующееся дерево, красно-черное дерево, используется в планировании ядра, чтобы вытеснить использование многопроцессорной операционной системы компьютера.
- Другая разновидность дерева, B-деревья, широко используются для хранения древовидных структур на диске. Они используются для индексации большого количества записей.
- B-деревья также используются для вторичных индексов в базах данных, где индекс облегчает операцию выбора для ответа на некоторые критерии диапазона.
- Деревья являются важной структурой данных, используемой для построения компилятора.
- Деревья также используются при проектировании баз данных.
- Деревья используются в каталогах файловой системы.
- Деревья также широко используются для хранения и поиска информации в таблицах символов.

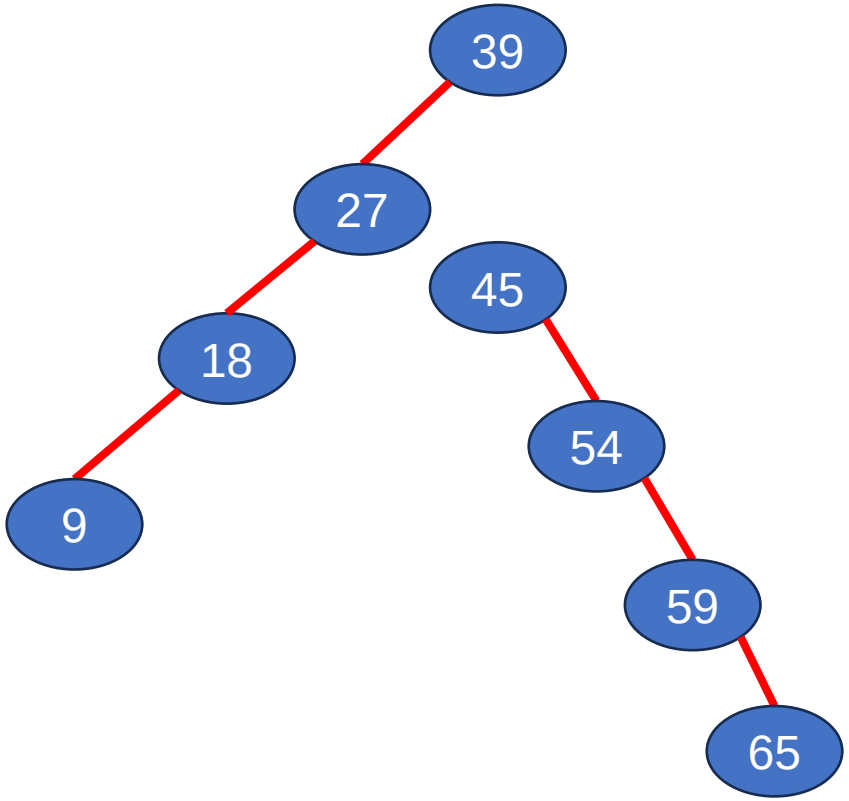
Двоичные деревья поиска

Двоичное дерево поиска, также известное как упорядоченное бинарное дерево, является вариантом бинарных деревьев, в котором узлы расположены в определенном порядке. В бинарном дереве поиска все узлы в левом поддереве имеют значение, меньшее, чем у корневого узла. Соответственно, все узлы в правом поддереве имеют значение, равное или большее, чем корневой узел. Это же правило применимо к каждому поддереву в дереве. (Обратите внимание, что бинарное дерево поиска может содержать или не содержать повторяющиеся значения в зависимости от его реализации.)



Таким образом, среднее время выполнения операции поиска составляет O(log2n), так как на каждом шаге мы исключаем половину поддерева из процесса поиска.

Двоичные деревья поиска

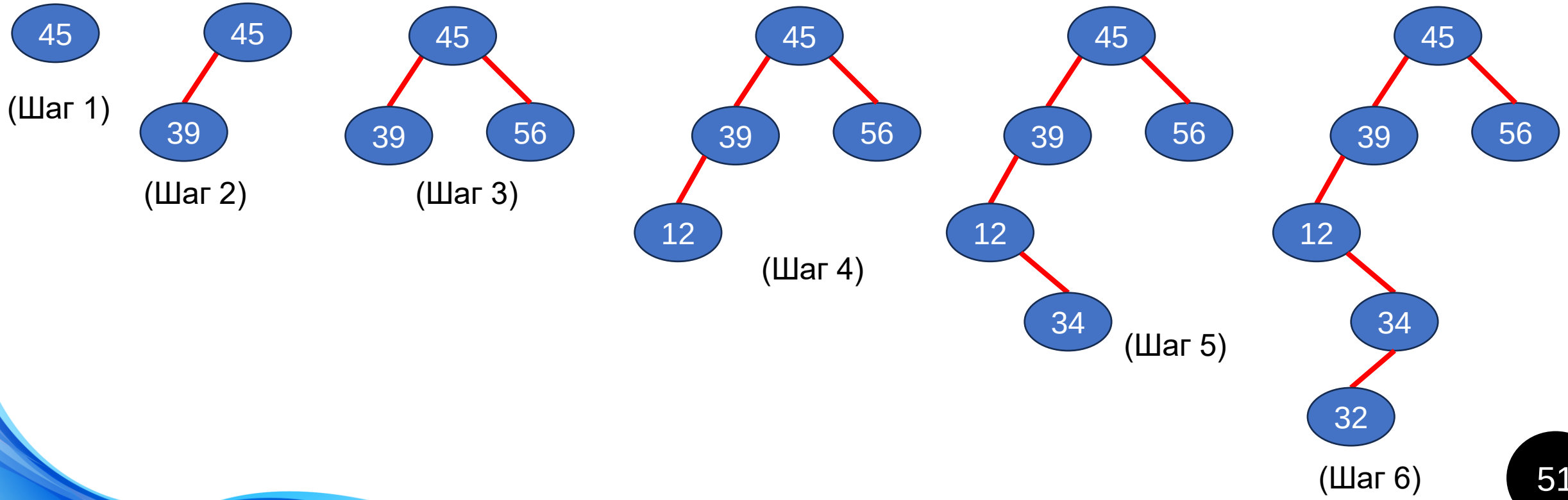


Лево-скошенные и право-скошенные бинарные деревья поиска

В худшем случае бинарное дерево поиска будет тратить O(n) времени на поиск элемента. Худший случай возникнет, когда дерево представляет собой линейную цепочку узлов. Подводя итог, можно сказать, что бинарное дерево поиска — это бинарное дерево со следующими свойствами:
Левое поддерево узла N содержит значения, которые меньше значения N.
•Правое поддерево узла N содержит значения, которые больше значения N.
И левое, и правое бинарные деревья также удовлетворяют этим свойствам и, таким образом, являются бинарными деревьями поиска.

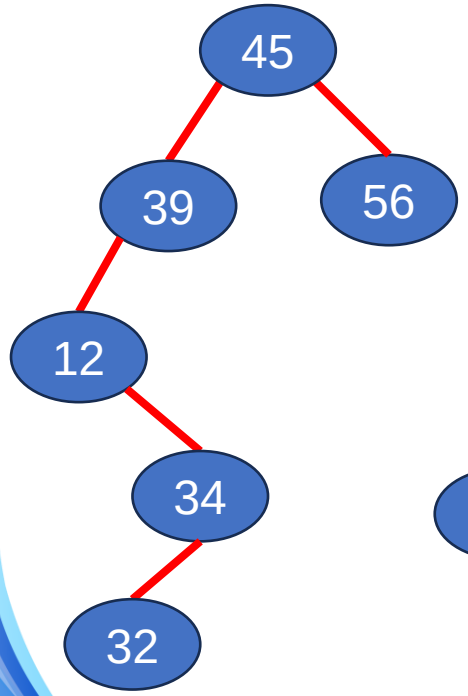
Двоичные деревья поиска – пример создания

Создание двоичного дерева поиска с использованием следующих элементов данных: 45, 39, 56, 12, 34, 32, 10, 78, 89, 54, 67, 81

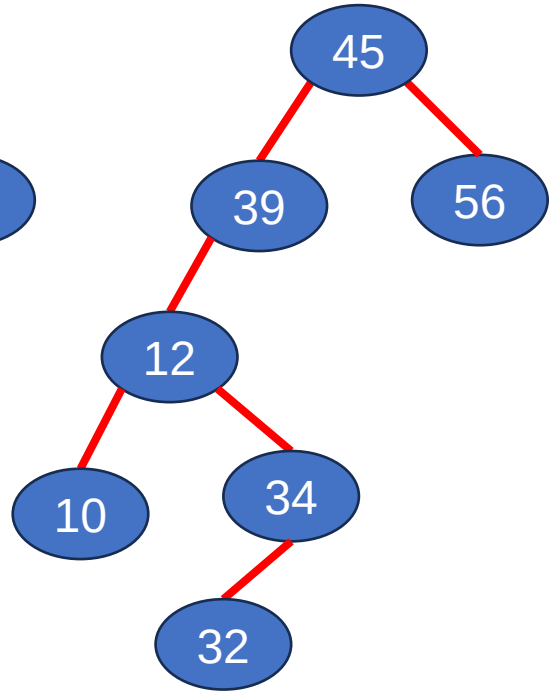


Двоичные деревья поиска – пример создания

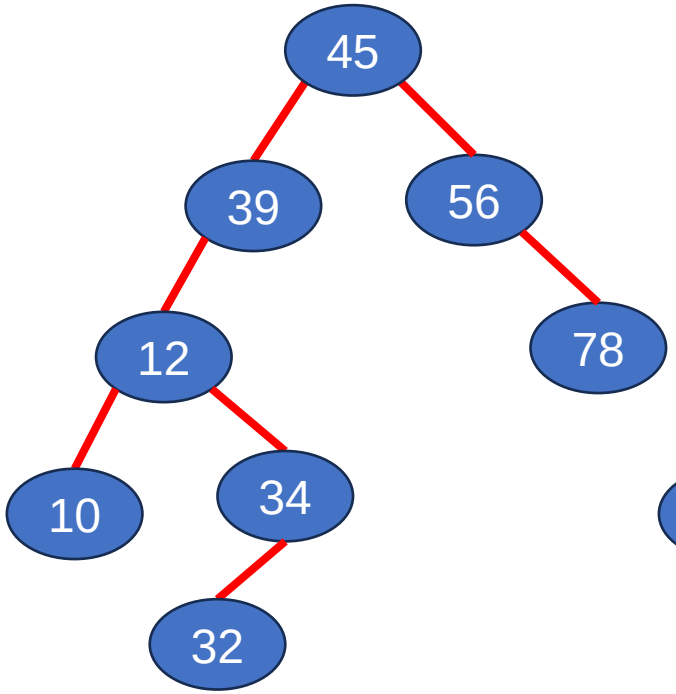
Создание двоичного дерева поиска с использованием следующих элементов данных: 45, 39, 56, 12, 34, 32, 10, 78, 89, 54, 67, 81



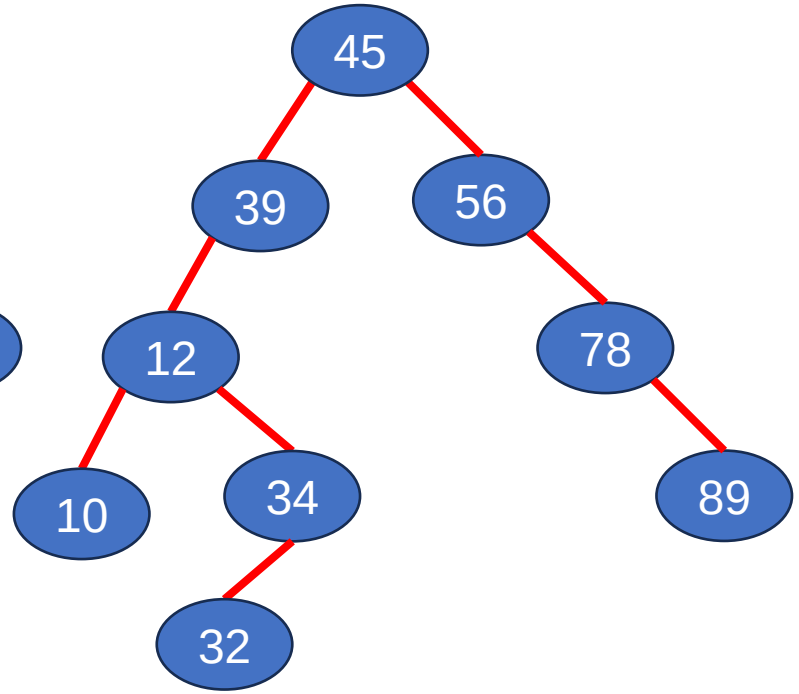
(Шаг 6)



(Шаг 7)



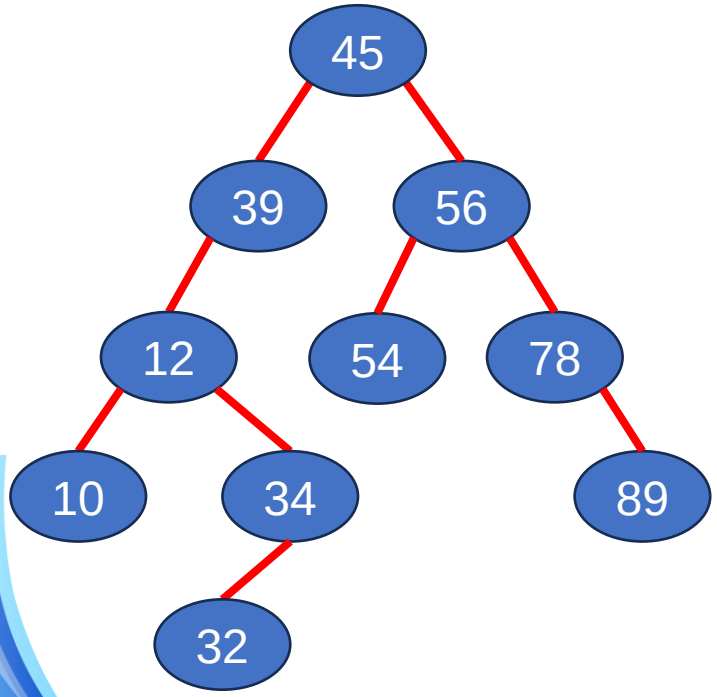
(Шаг 8)



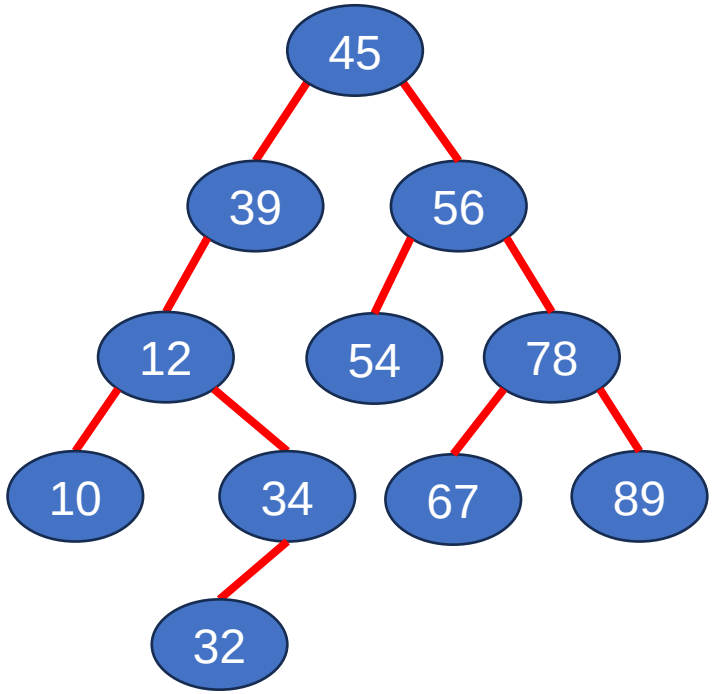
(Шаг 9)

Двоичные деревья поиска – пример создания

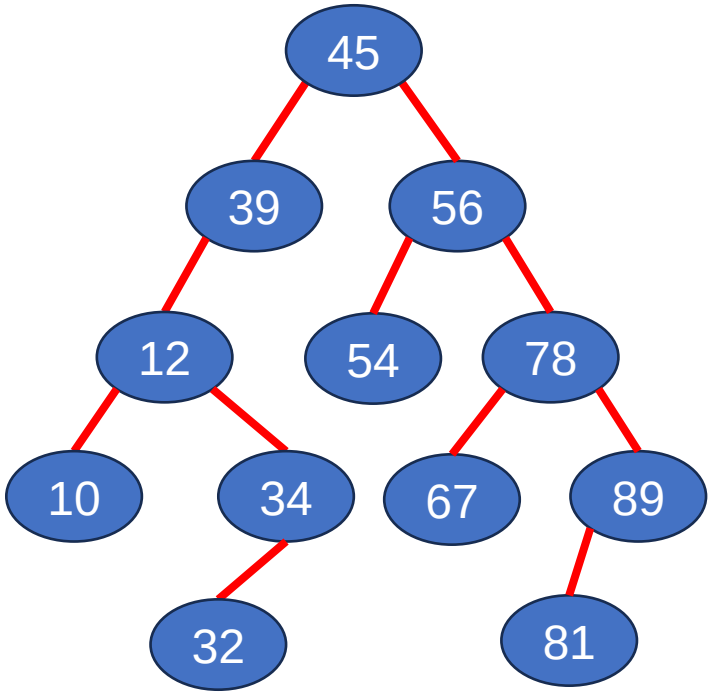
Создание двоичного дерева поиска с использованием следующих элементов данных: 45, 39, 56, 12, 34, 32, 10, 78, 89, 54, 67, 81



(Шаг 10)



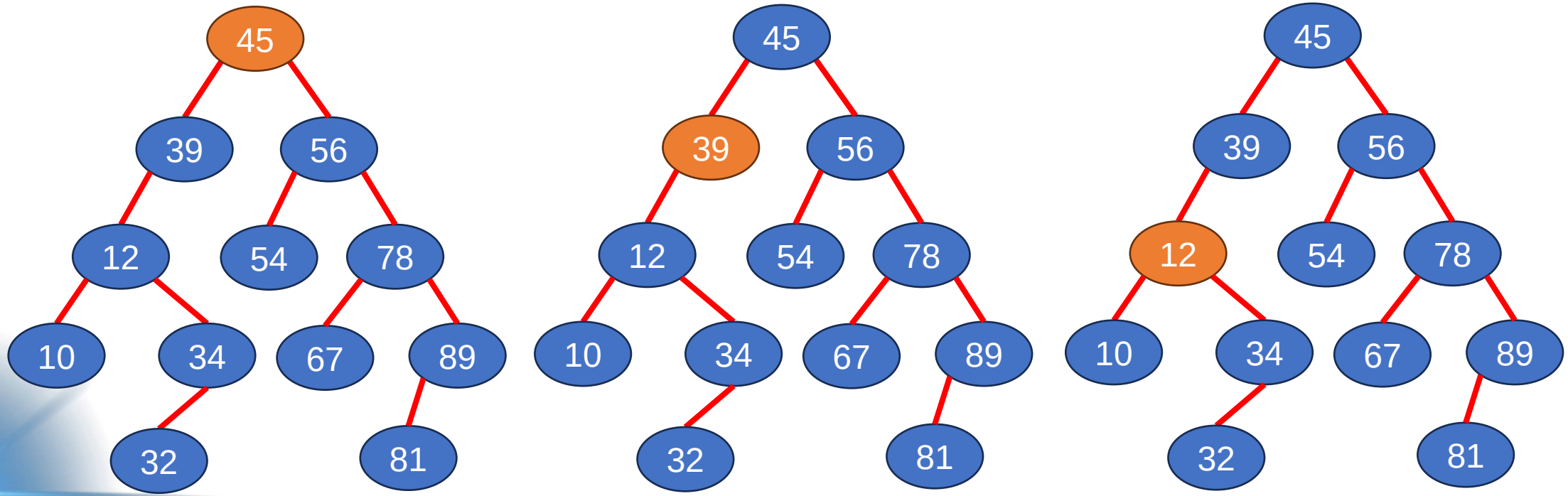
(Шаг 11)



(Шаг 12)

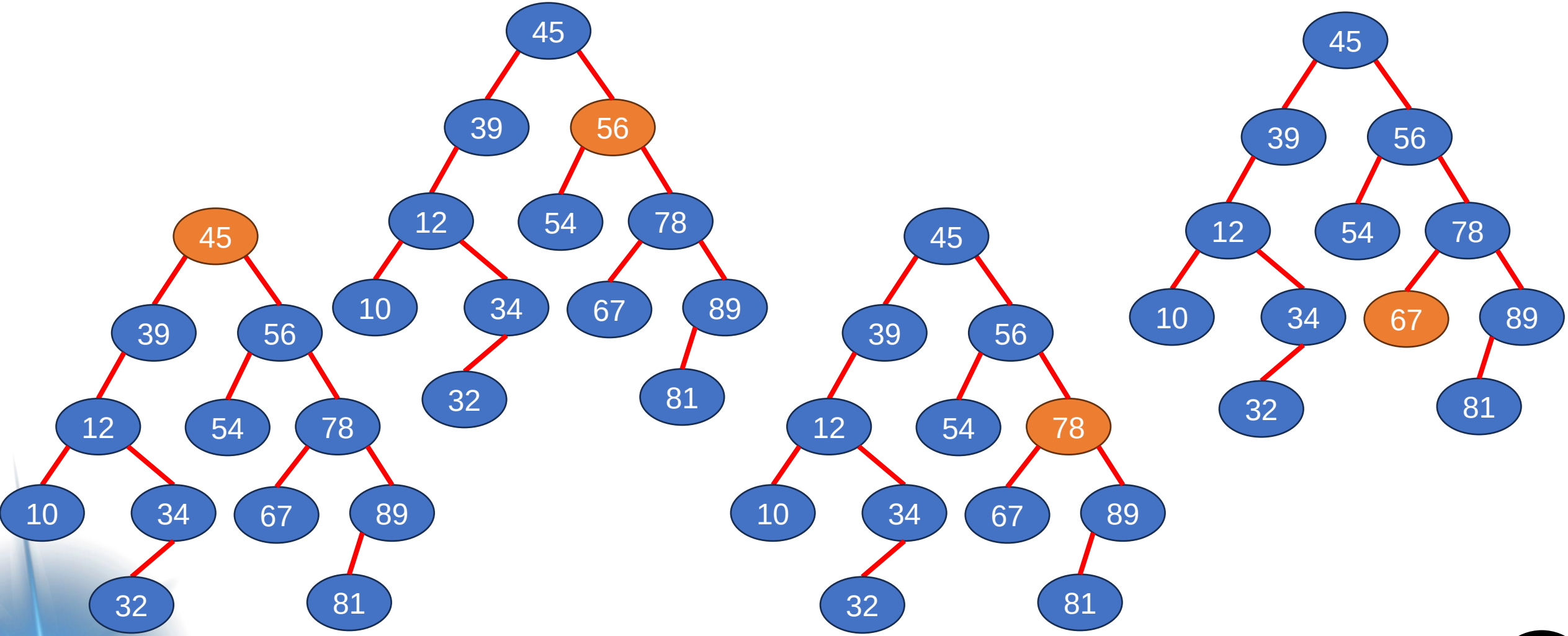
Поиск узла в бинарном дереве поиска

Функция поиска используется для определения того, присутствует ли заданное значение в дереве или нет. Процесс поиска начинается с корневого узла. Сначала функция проверяет, пусто ли бинарное дерево поиска. Если оно пусто, то искомое значение отсутствует в дереве. Поэтому алгоритм поиска завершается выводом соответствующего сообщения. Однако если в дереве есть узлы, то функция поиска проверяет, равно ли ключевое значение текущего узла искомому значению. Если нет, она проверяет, меньше ли искомое значение значения текущего узла, и в этом случае ее следует рекурсивно вызвать на левом дочернем узле. В случае, если значение больше значения текущего узла, ее следует рекурсивно вызвать на правом дочернем узле.



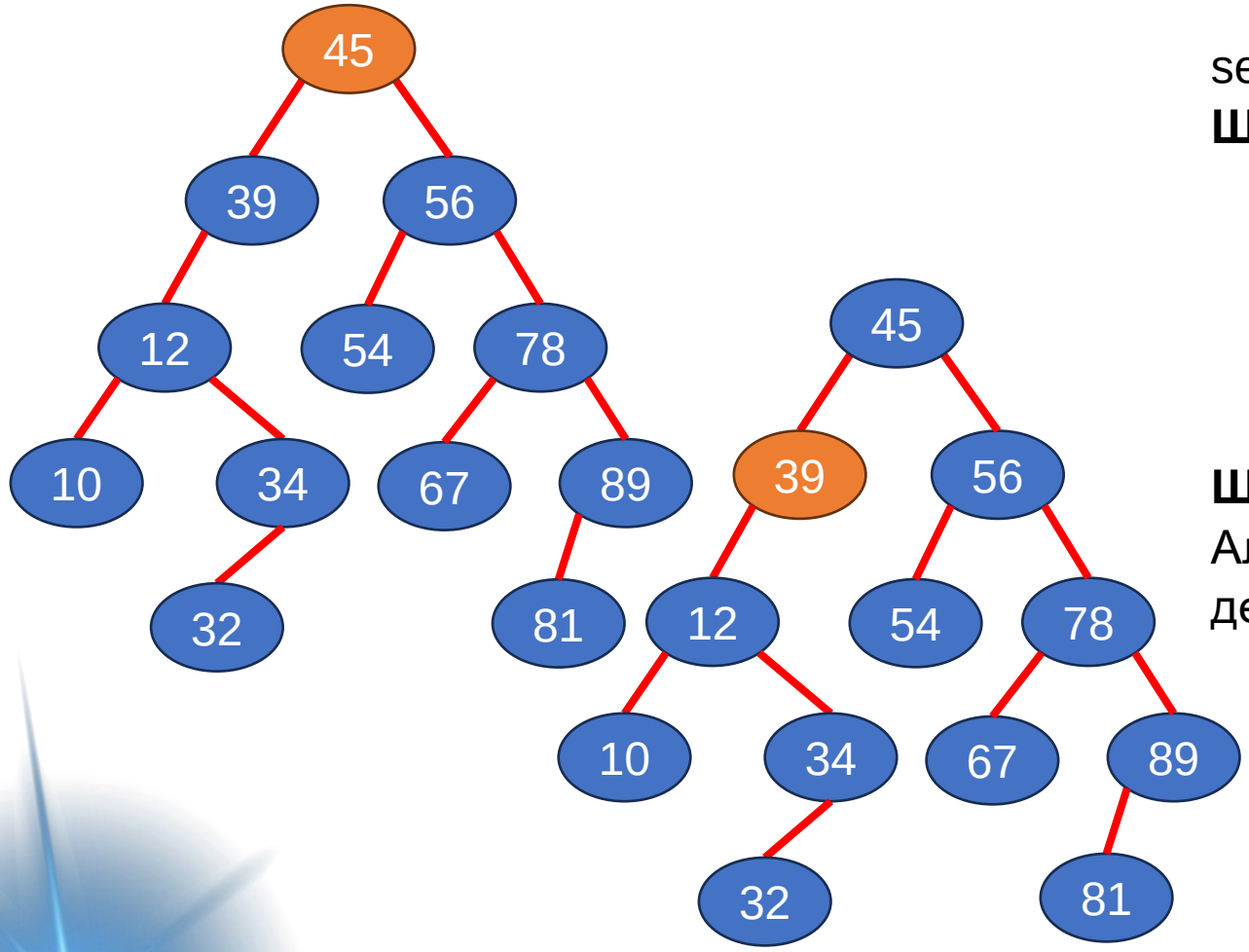
Поиск узла со значением 12 в заданном двоичном дереве поиска

Поиск узла в бинарном дереве поиска



Поиск узла со значением 67 в заданном двоичном дереве поиска

Поиск узла в бинарном дереве поиска



Поиск узла со значением 40 в заданном бинарном дереве поиска

searchElement (TREE, VAL)

Шаг 1: *IF* TREE->DATA =VAL OR TREE =NULL
RETURN TREE
ELSE IF VAL < TREE->DATA
RETURN searchElement(TREE->LEFT, VAL)
ELSE
RETURN searchElement(TREE->RIGHT, VAL)

Шаг 2: END

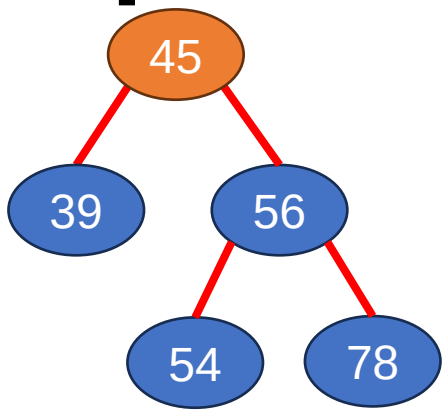
Алгоритм поиска заданного значения в бинарном дереве поиска

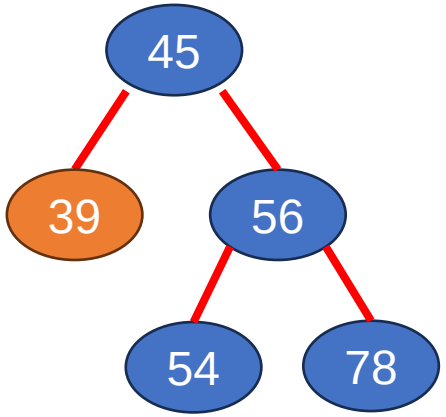
Вставка нового узла в бинарное дерево поиска

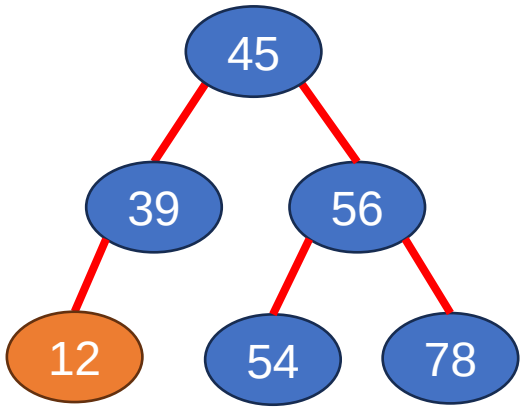
Функция вставки используется для добавления нового узла с заданным значением в правильной позиции в бинарном дереве поиска. Добавление узла в правильной позиции означает, что новый узел не должен нарушать свойства бинарно дерева поиска.

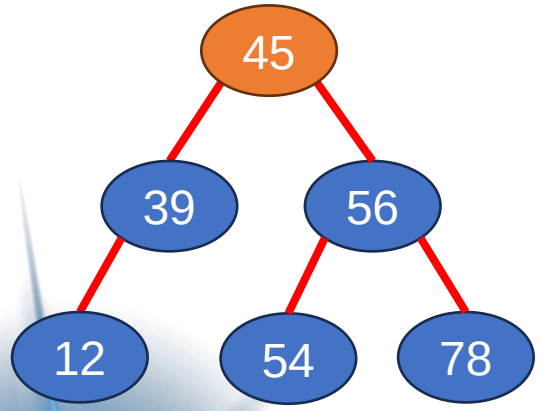
```
Insert (TREE, VAL)
Шаг 1:  IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE IF VAL < TREE -> DATA
            Insert(TREE -> LEFT, VAL)
        ELSE
            Insert(TREE -> RIGHT, VAL)
Шаг 2: END
```

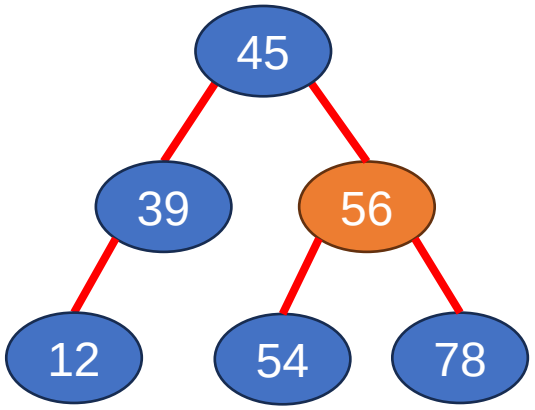
Вставка нового узла в бинарное дерево поиска

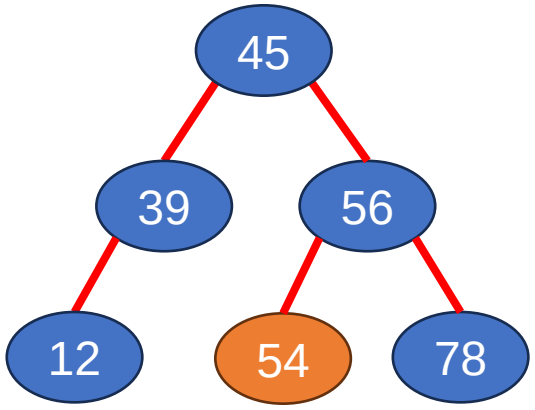


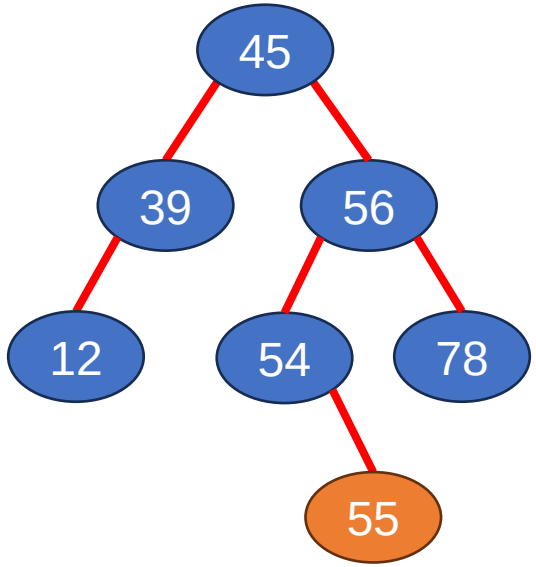






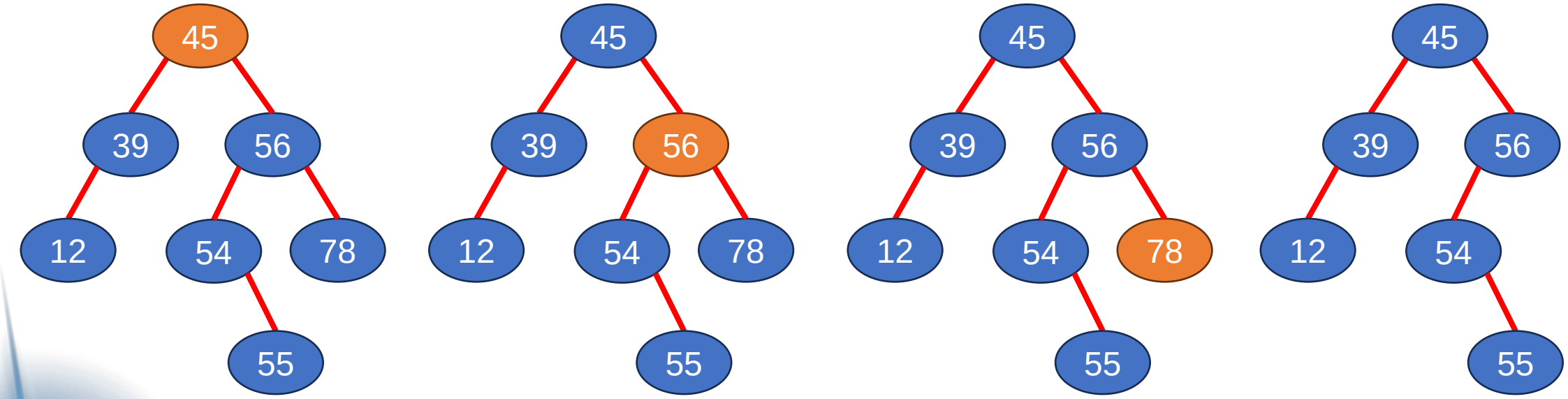






Удаление узла из двоичного дерева поиска

Случай 1: Удаление узла, не имеющего потомков
Если нам нужно удалить узел 78, мы можем просто удалить этот узел без каких-либо проблем. Это простейший случай удаления.

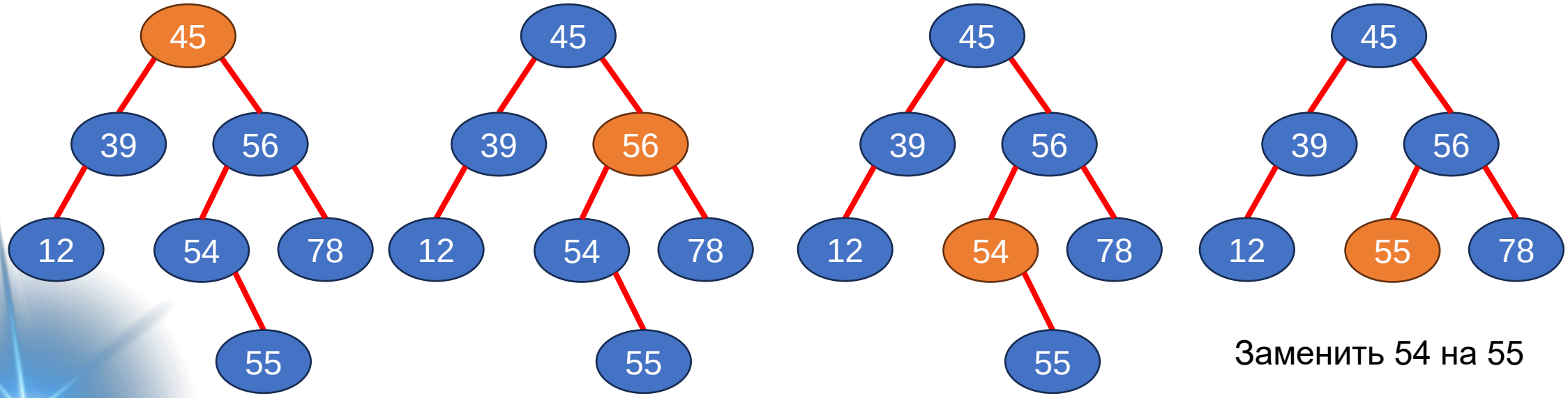


Удалить конечный узел 78

Удаление узла из двоичного дерева поиска

Случай 2: Удаление узла с одним потомком

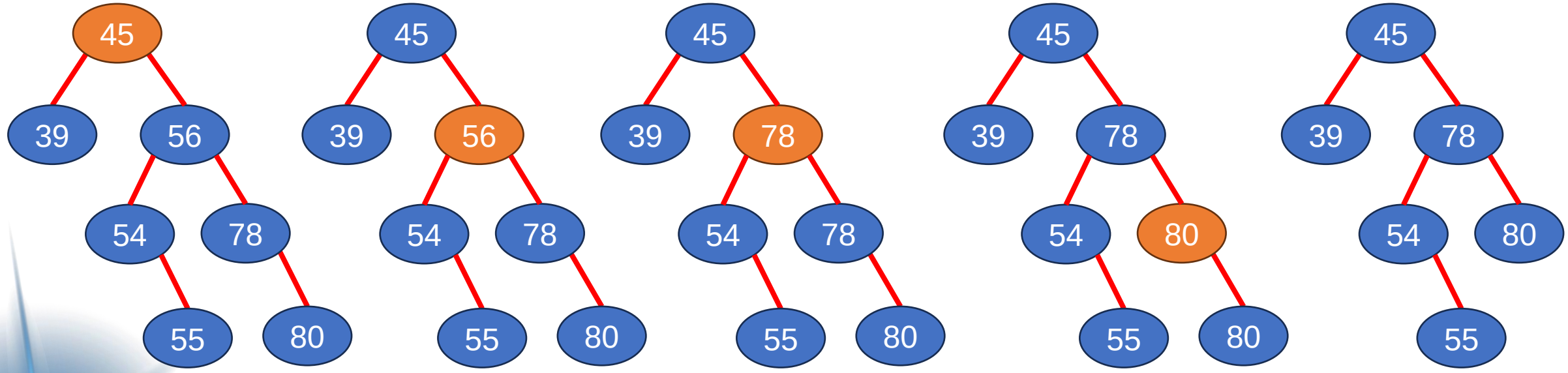
Для обработки этого случая потомок узла устанавливается как потомок родителя узла. Другими словами, замените узел его потомком. Теперь, если узел является левым потомком своего родителя, потомок узла становится левым потомком родителя узла. Соответственно, если узел является правым потомком своего родителя, потомок узла становится правым потомком родителя узла. Посмотрите, как обрабатывается удаление узла 54.



Удаление узла из двоичного дерева поиска

Случай 3: Удаление узла с двумя потомками

Это удаление также можно выполнить, заменив узел 56 его последовательной заменой последователей

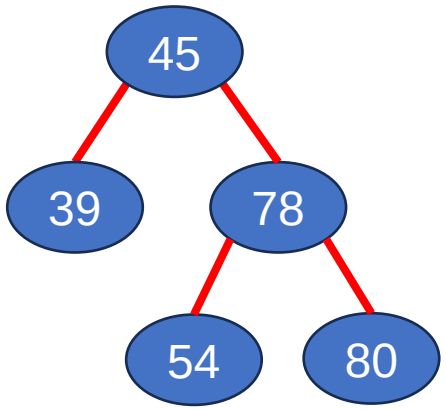


Удаление узла из двоичного дерева поиска

```
Delete (TREE, VAL)
Шаг 1:  IF TREE = NULL
            Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
            Delete(TREE -> LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
            Delete(TREE RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
            SET TEMP = findLargestNode(TREE -> LEFT)
            SET TREE -> DATA = TEMP -> DATA
            Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
            SET TEMP = TREE
            IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
                SET TREE = NULL
            ELSE IF TREE -> LEFT != NULL
                SET TREE = TREE -> LEFT
            ELSE
                SET TREE = TREE -> RIGHT
            FREE TEMP
Шаг 2: END
```

Определение высоты двоичного дерева поиска

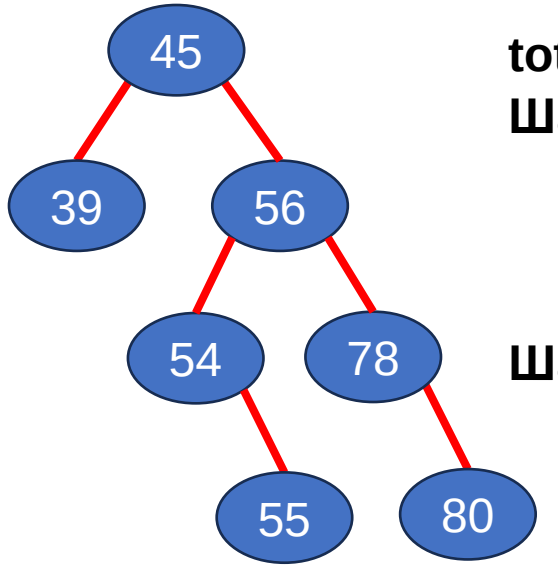
Чтобы определить высоту двоичного дерева поиска, мы вычисляем высоту левого поддерева и правого поддерева. Какая бы высота ни была больше, к ней добавляется 1. Например, если высота левого поддерева больше, чем высота правого поддерева, то к левому поддереву добавляется 1, в противном случае к правому поддереву добавляется 1.



Height (TREE)
Шаг 1: *IF* TREE = NULL
Return 0
ELSE
SET LeftHeight = Height(TREE -> LEFT)
SET RightHeight = Height(TREE -> RIGHT)
IF LeftHeight > RightHeight
Return LeftHeight + 1
ELSE
Return RightHeight + 1
Шаг 2: END

Определение количества узлов

Определение количества узлов в двоичном дереве поиска аналогично определению его высоты. Чтобы вычислить общее количество элементов/узлов в дереве, мы подсчитываем количество узлов в левом поддереве и правом поддереве.
Количество узлов = totalNodes(left sub–tree) + totalNodes(right sub–tree) + 1

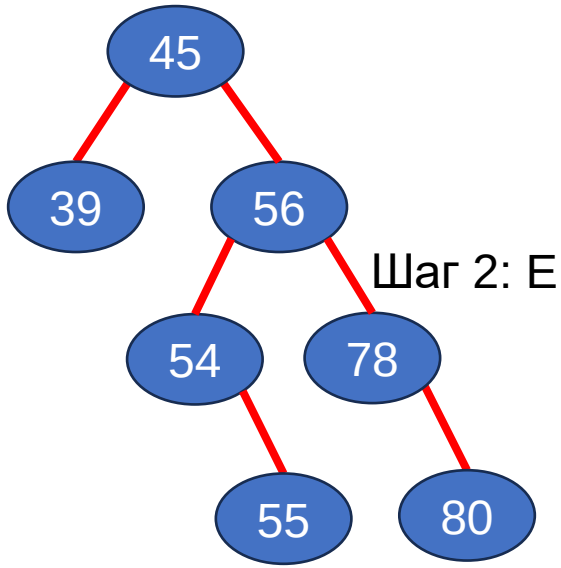


totalNodes(TREE)
Шаг 1: *IF* TREE = NULL
Return 0
ELSE
Return totalNodes(TREE -> LEFT) + totalNodes(TREE -> RIGHT) + 1
Шаг 2: END

Определение количества внутренних узлов

Чтобы рассчитать общее количество внутренних узлов или нелистовых узлов, мы подсчитываем количество внутренних узлов в левом поддереве и правом поддереве и добавляем к нему 1 (1 добавляется для корневого узла).

```
totalInternalNodes(TREE)
Шаг 1:  IF TREE = NULL
            Return 0
        ELSE IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
            Return 0
        ELSE
            Return totalInternalNodes(TREE LEFT) + totalInternalNodes(TREE RIGHT) + 1
Шаг 2: END
```

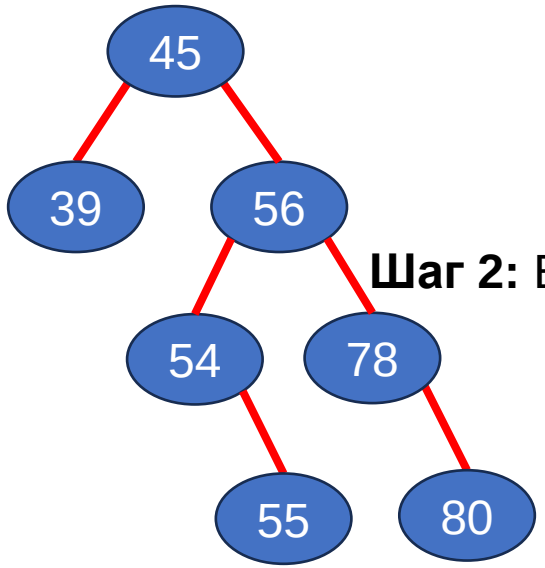


Общее количество внутренних узлов = 4

Определение количества внутренних узлов

Чтобы рассчитать общее количество внутренних узлов или нелистовых узлов, мы подсчитываем количество внутренних узлов в левом поддереве и правом поддереве и добавляем к нему 1 (1 добавляется для корневого узла).

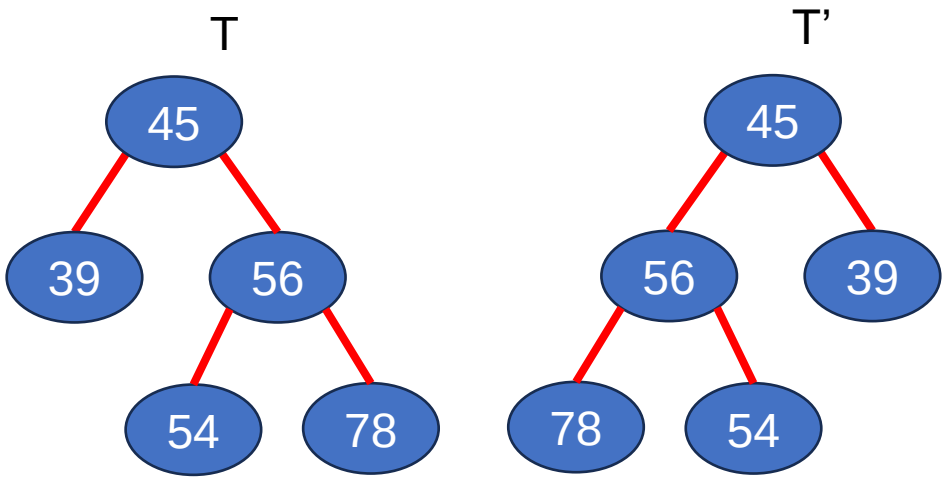
totalExternalNodes(TREE)
Шаг 1: *IF* TREE = NULL
Return 0
ELSE IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
Return 1
ELSE
Return totalExternalNodes(TREE -> LEFT) + totalExternalNodes(TREE -> RIGHT)
Шаг 2: END



Общее количество внешних узлов дерева = 3

Нахождение зеркального отображения двоичного дерева поиска

Зеркальное отображение двоичного дерева поиска получается путем замены левого поддерева правым поддеревом в каждом узле дерева.



MirrorImage(TREE)
Шаг 1: *IF* TREE != NULL
MirrorImage(TREE -> LEFT)
MirrorImage(TREE -> RIGHT)
SET TEMP = TREE -> LEFT
SET TREE -> LEFT = TREE -> RIGHT
SET TREE -> RIGHT = TEMP
Шаг 2: END

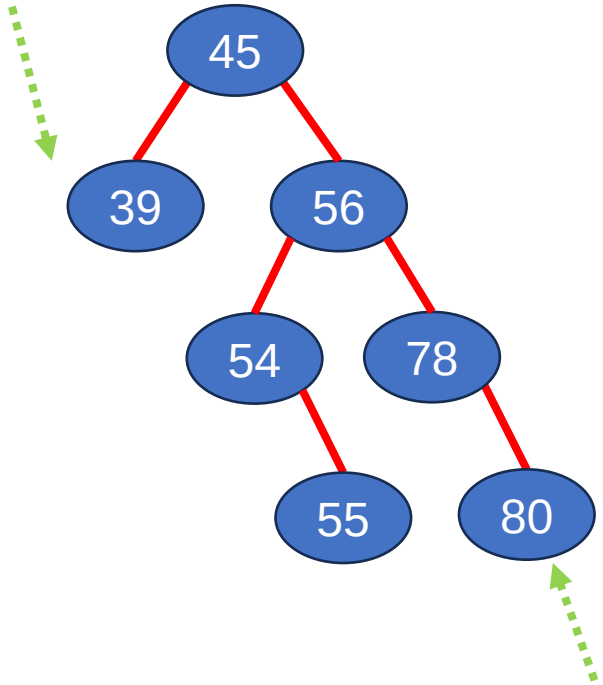
Удаление бинарного дерева поиска

Чтобы удалить/удалить все бинарное дерево поиска из памяти, мы сначала удаляем элементы/узлы в левом поддереве, а затем удаляем узлы в правом поддереве.

deleteTree(TREE)
Шаг 1: *IF* TREE != NULL
deleteTree (TREE -> LEFT)
deleteTree (TREE -> RIGHT)
Free (TREE)
Шаг 2: END

Поиск наименьшего и наибольшего узла в двоичном дереве поиска

Наименьший узел (самый левый потомок левого поддерева)



Наибольший узел (самый правый потомок правого поддерева)

findSmallestElement(TREE)
Шаг 1: *IF* TREE = NULL OR TREE -> LEFT = NULL
Return TREE
ELSE
Return findSmallestElement(TREE -> LEFT)
Шаг 2: END

findLargestElement(TREE)
Шаг 1: *IF* TREE = NULL OR TREE -> RIGHT = NULL
Return TREE
ELSE
Return findLargestElement(TREE -> RIGHT)
Шаг 2: END

Случайное построение бинарных деревьев поиска

Все базовые операции с бинарными деревьями поиска имеют время выполнения O(h), где h - высота дерева. Однако при вставке и удалении элементов высота дерева меняется. Если, например, все элементы вставляются в дерево в строго возрастающей последовательности, то такое дерево вырождается в цепочку высотой n - 1. С другой стороны, $h \geq [\lg n]$. Как и в случае быстрой сортировки, можно показать, что поведение алгоритма в среднем случае гораздо ближе к наилучшему случаю, чем к наихудшему.

К сожалению, в ситуации, когда при формировании бинарного дерева поиска используются и вставки, и удаления, о средней высоте образующихся деревьев известно мало, так что мы ограничимся анализом ситуации, когда дерево строится только с использованием вставок, без удалений. Определим случайно построенное бинарное дерево gоиска (randomly built binary search tree) с n ключами как дерево, которое возникает при вставке ключей в изначально пустое дерево в случайном порядке, когда все n! перестановок входных ключей равновероятны.

Случайное построение бинарных деревьев поиска

Теорема

Математическое ожидание высоты случайно построенного бинарного дерева поиска с n различными ключами равно O(lgn).

Доказательство. Начнем с определения трех случайных величин, которые помогут определить высоту случайного бинарного дерева поиска. Обозначая высоту случайного бинарного дерева поиска с n ключами как X_n, определим ***экспоненциальную высоту*** $Y_n = 2^{X^n}$. При построении бинарного дерева поиска с n ключами мы выбираем один из них в качестве корня. Обозначим через R_n случайную величину, равную рангу корневого ключа в множестве из всех n ключей, т.е. Rn содержит позицию, которую бы занимал ключ, если бы множество было отсортировано. Значение R_n с равной вероятностью может быть любым элементом множества {1,2,…,n}. Если R_n = i, то левое поддерево корня представляет собой случайно построенное бинарное дерево поиска с i - 1 ключами, а правое – с n - i ключами. Поскольку высота бинарного дерева на единицу больше наибольшей из высот поддеревьев корневого узла, экспоненциальная высота бинарного дерева в два раза больше экспоненциальной высоты наивысшего из поддеревьев корневого узла.

Случайное построение бинарных деревьев поиска

Если мы знаем, что $R_n = i$, то $Y_n = 2 \cdot \max(Y_{i-1}, Y_{n-i})$
В качестве базового случая мы имеем $Y_1 = 1$, поскольку экспоненциальная высота дерева с одним узлом составляет $2^0 = 1$, и для удобства мы определим $Y_0 = 0$.
Далее мы определим индикаторные случайные величины $Z_{n,1}, Z_{n,2},\ldots, Z_{n,n}$, где $Z_{n,i} = I\{R_n = i\}$.
Поскольку R_n с равной вероятностью может быть любым элементом множества $\{1,2,\ldots,n\}$, $Pr\{R_n = i\} = 1/n$ для $i = 1,2, \ldots,n$, а следовательно, мы имеем

$$E[Z_{n,i}] = \frac{1}{n}$$

для $i = 1,2,\ldots n$. Поскольку ровно одно значение $Z_{n,i}$ равно 1, а все прочие равны 0, мы также имеем

$$Y_n = \sum_{i=1}^{n} Z_{n,i} \, (2 \cdot max(Y_{i-1}, Y_{n-1}))$$

Мы покажем, что $E[Y_n]$ полиномиально зависит от n, что неизбежно приводит к выводу, что $E[X_n] = O(\lg n)$.

Случайное построение бинарных деревьев поиска

Мы утверждаем, что индикаторная случайная переменная $Z_{n,i} = I\{R_n = i\}$ не зависит от значений Yi-1 и Yn-i. Если Rn = i, то левое поддерево, экспоненциальная высота которого равна Yi-1, случайным образом строится из i - 1 ключей, ранги которых меньше i. Это поддерево ничем не отличается от любого другого случайного бинарного дерева поиска из i - 1 ключей. Выбор $R_n = i$ никак не влияет на структуру этого дерева, а влияет только на количество содержащихся в нем узлов. Следовательно, случайные величины Y_{i-1} и $Z_{n,i}$ независимы. Аналогично правое поддерево, экспоненциальная высота которого равна Y_{n-i}, строится случайным образом из n - i ключей, ранги которых больше i. Структура этого дерева не зависит от R_n, так что случайные величины Y_{n-i} и Z_{n-i} независимы. Следовательно, мы имеем

$$E[Y_n] = E\left[\sum_{i=1}^{n} Z_{n,i}\,(2 \cdot max(Y_{i-1}, Y_{n-i}))\right] = \sum_{i=1}^{n} E\left[Z_{n,i}\,(2 \cdot max(Y_{i-1}, Y_{n-i}))\right]$$

$$= \sum_{i=1}^{n} E[Z_{n,i}]\,E[(2 \cdot max(Y_{i-1}, Y_{n-i}))] = \sum_{i=1}^{n} \frac{1}{n}\,E[(2 \cdot max(Y_{i-1}, Y_{n-i}))] = \frac{2}{n}\sum_{i=1}^{n} E[(max(Y_{i-1}, Y_{n-i}))]$$

$$\leq \frac{2}{n}\sum_{i=1}^{n} (E[Y_{i-1}] + E[Y_{n-i}])$$

Случайное построение бинарных деревьев поиска

Поскольку каждый член $E[Y_0]$, $E[Y_1]$,…, $E[Y_{n-1}]$ появляется в последней сумме дважды, как $E[Y_{i-1}]$ и как $E[Y_{n-i}]$, мы получаем следующее рекуррентное соотношение:

$$E[Y_n] \leq \frac{4}{n}\sum_{i=0}^{n-1} E[Y_i]$$

Используя метод подстановок, покажем, что для всех натуральных n рекуррентное соотношение имеет следующее решение:

$$E[Y_n] \leq \frac{1}{4}\binom{n+3}{n}$$

При этом мы воспользуемся тождеством

$$\sum_{i=0}^{n}\binom{i+3}{3} = \binom{n+3}{4}$$

(данное тождество предлагается доказать самостоятельно.)

Случайное построение бинарных деревьев поиска

Заметим, что для базовых случаев границы $0 = Y_0 = E[Y_0]\,(1/4)\binom{3}{3} = ¼$ и $1 = Y_1 = E[Y_0]\,(1/4)\binom{1+3}{3} = 1$ справедливы. По индукции имеем

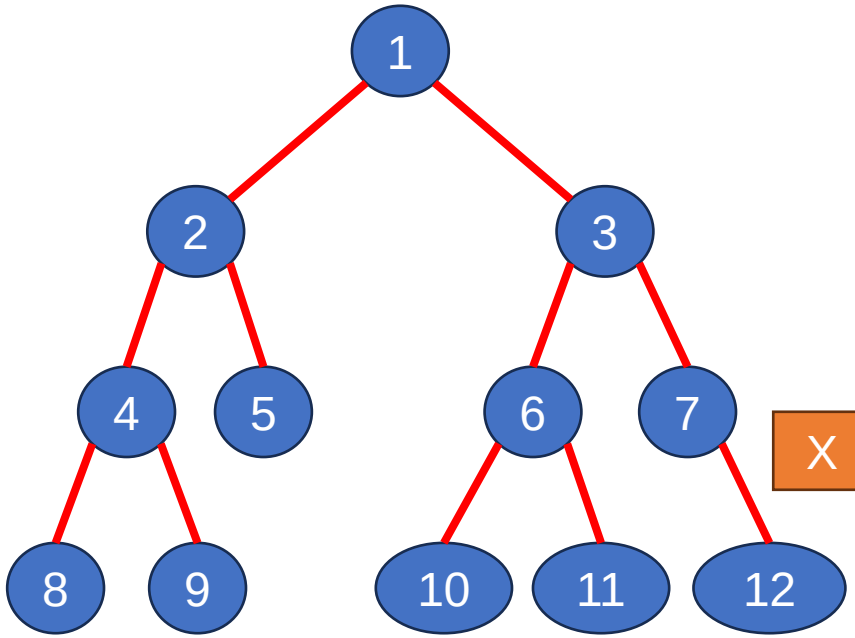
$$E[Y_n] \le \frac{4}{n}\sum_{i=0}^{n-1} E[Y_i] \le \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}\binom{i+3}{3} = \frac{1}{n}\sum_{i=0}^{n-1}\binom{i+3}{3} = \frac{1}{n}\binom{n+3}{4} = \frac{1}{n}\frac{(n+3)!}{4!\,(n-1)!} = \frac{1}{4}\frac{(n+3)!}{3!\,n!} = \frac{1}{4}\binom{n+3}{n}$$

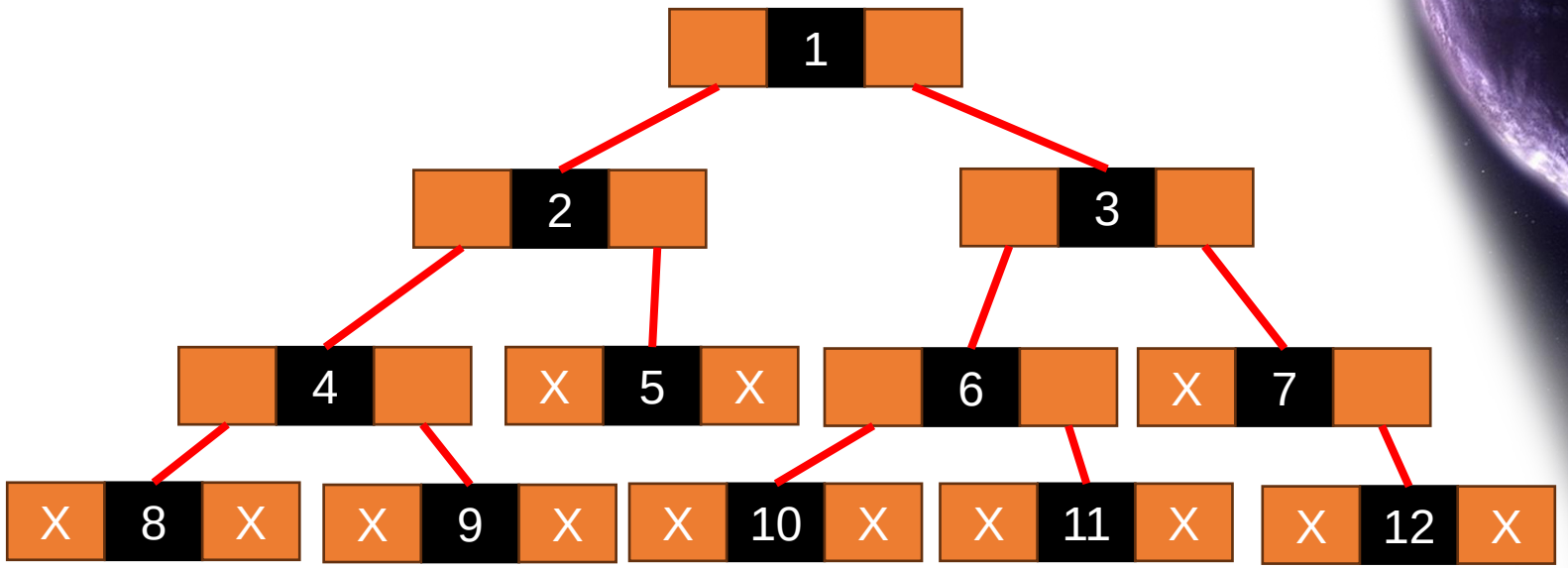
Мы получили границу для E [Y_n], но наша конечная цель - найти границу E[Xn]. Так как функция f(x) = 2^x выпуклая вниз. Таким образом, мы можем применить неравенство Йенсена , которое гласит, что $2^{E[Xa]} < E^{[2Xn]} = E[Yn]$,

Взятие логарифма от обеих частей дает нам $E[X_n] = O(lgn)$.

Связные двоичные деревья

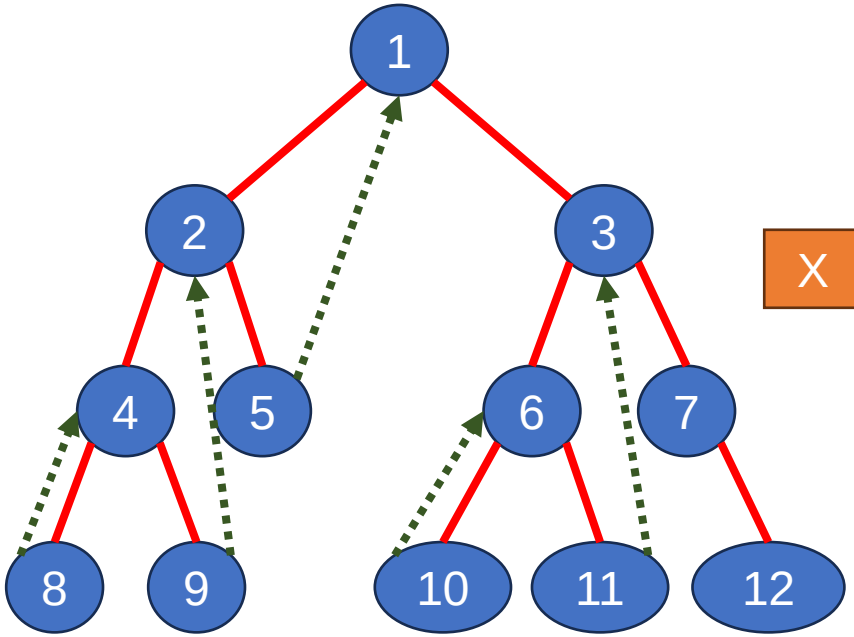
Связное двоичное дерево такое же, как и двоичное дерево, но с разницей в хранении указателей NULL.

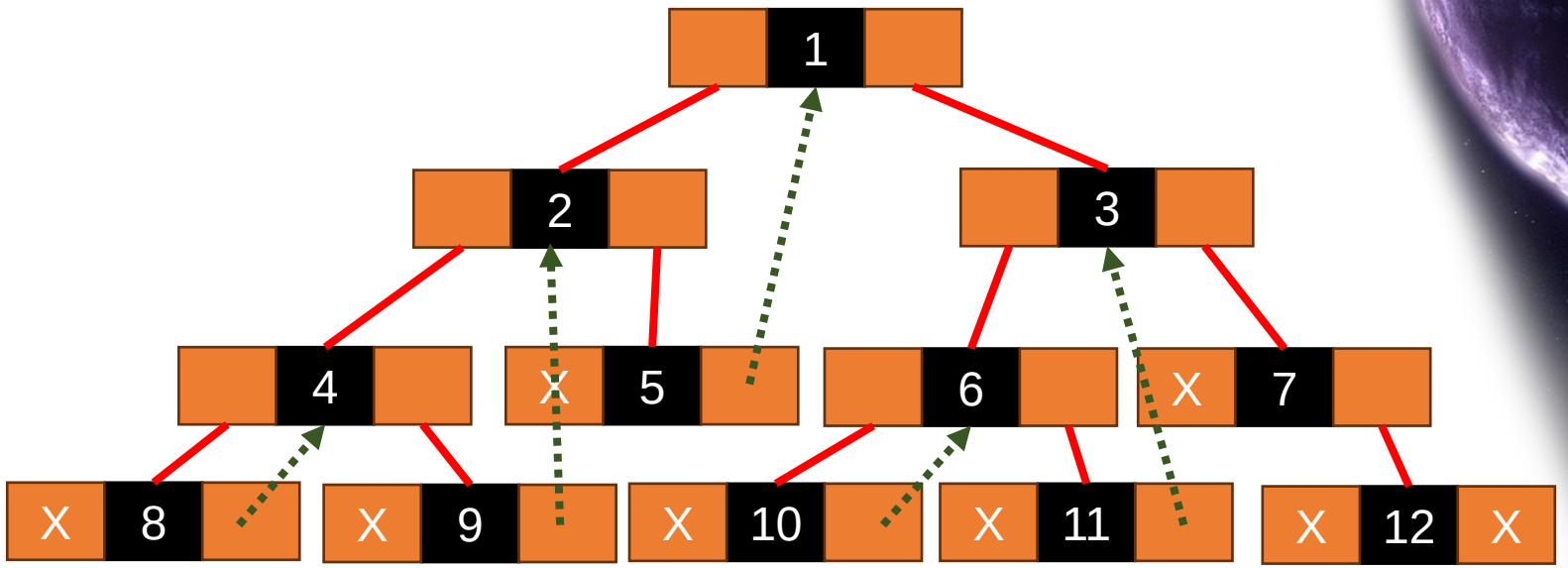




Одностороннее связывание

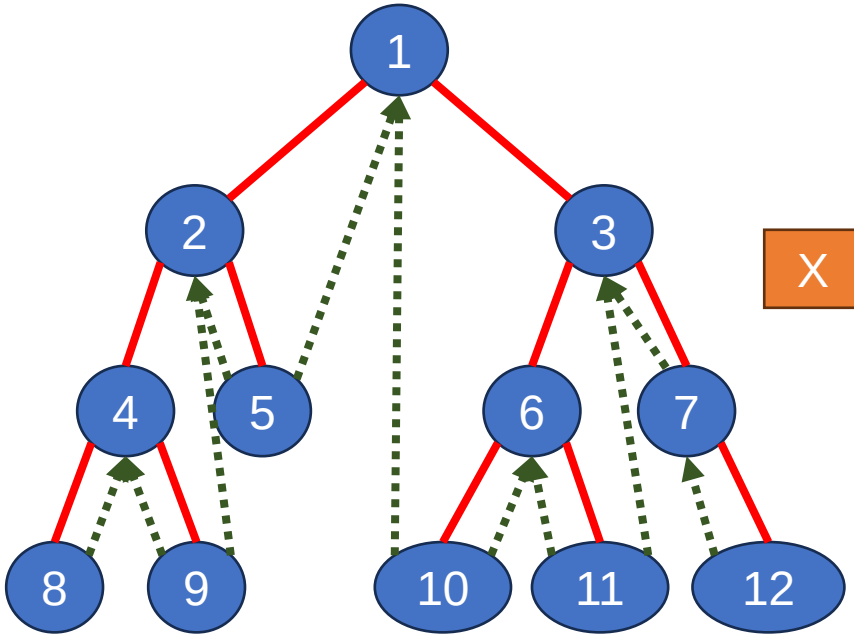
Связное двоичное дерево такое же, как и двоичное дерево, но с разницей в хранении указателей NULL.

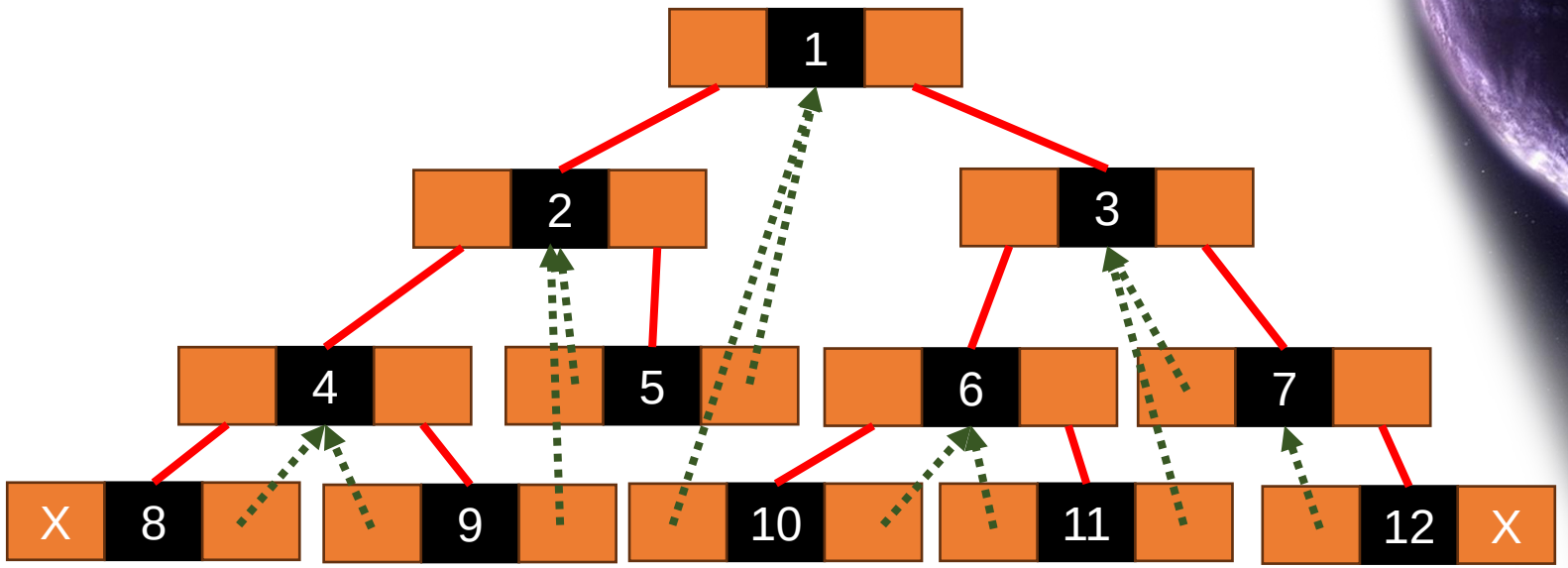




Двухстороннее связывание

Связное двоичное дерево такое же, как и двоичное дерево, но с разницей в хранении указателей NULL.



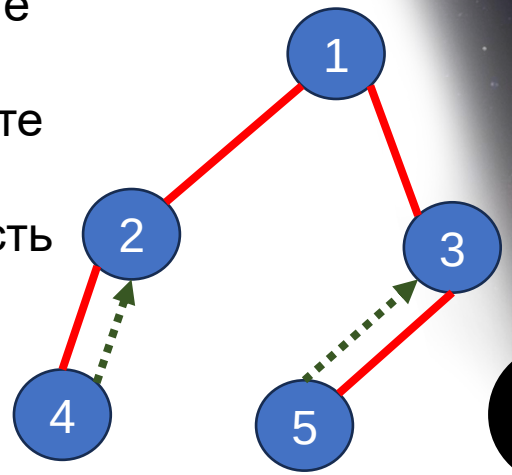


Обход связного бинарного дерева

Для каждого узла сначала посетите левое поддерево, если оно существует и не было посещено ранее. Затем за самим узлом (корнем) следует посещение его правого поддерева (если оно существует). В случае отсутствия правого поддерева проверьте наличие нитевидной ссылки и сделайте нитевидный узел текущим рассматриваемым узлом.
Шаг 1: проверьте, есть ли у текущего узла левый дочерний узел, который не был посещен. Если левый дочерний узел существует, который не был посещен, перейдите к шагу 2, в противном случае перейдите к шагу 3.
Шаг 2: добавьте левого дочернего узла в список посещенных узлов. Сделайте его текущим узлом и перейдите к шагу 6.
Шаг 3: Если текущий узел имеет правого потомка, перейдите к шагу 4, иначе перейдите к шагу 5.
Шаг 4: Сделайте этого правого потомка текущим узлом и перейдите к шагу 6. Шаг 5: выведите узел и, если есть узел с нитью, сделайте его текущим узлом.
Шаг 6: Если все узлы посещены, то END, иначе перейдите к шагу 1.

Обход связного бинарного дерева

1. Узел 1 имеет левого потомка, т. е. 2, который не был посещен. Поэтому добавьте 2 в список посещенных узлов, сделайте его текущим узлом.
2. Узел 2 имеет левого потомка, т. е. 4, который не был посещен. Итак, добавьте 4 в список посещенных узлов, сделайте его текущим узлом.
3. Узел 4 не имеет левого или правого потомка, поэтому выведите 4 и проверьте его связанную ссылку. У него есть связанная ссылка на узел 2, поэтому сделайте узел 2 текущим узлом.
4. Узел 2 имеет левого потомка, который уже был посещен. Однако у него нет правого потомка. Теперь выведите 2 и перейдите по его связанной ссылке на узел 1. Сделайте узел 1 текущим узлом.
5. Узел 1 имеет левого потомка, который уже был посещен. Поэтому выведите 1. Узел 1 имеет правого потомка 3, который еще не был посещен, поэтому сделайте его текущим узлом.
6. Узел 3 имеет левого потомка (узел 5), который не был посещен, поэтому сделайте его текущим узлом.
7. Узел 5 не имеет левого или правого потомка. Итак, выведите 5. Однако у него есть нить ссылки, указывающая на узел 3. Сделайте узел 3 текущим узлом.
8. Узел 3 имеет левого потомка, который уже был посещен. Итак, выведите 3.
9. Теперь не осталось ни одного узла, поэтому мы заканчиваем здесь. Последовательность выведенных узлов — 4 2 1 5 3.



Преимущества связного двоичного дерева

- ✓ Он обеспечивает линейный обход элементов в дереве.
- ✓ Линейный обход исключает использование стеков, которые, в свою очередь, потребляют много памяти и машинного времени.
- ✓ Он позволяет находить родителя заданного элемента без явного использования родительских указателей.
- ✓ Поскольку узлы содержат указатели на упорядоченных предшественника и преемника, нитевидное дерево обеспечивает прямой и обратный обход узлов, как задано упорядоченным способом.

Таким образом, мы видим, что основное различие между двоичным деревом и потоковым двоичным деревом заключается в том, что в двоичных деревьях узел хранит указатель NULL, если у него нет потомка, и поэтому нет возможности вернуться назад.