

16.12.2024

Мультидеревья.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

N * Новосибирский
государственный
университет
***НАСТОЯЩАЯ НАУКА**

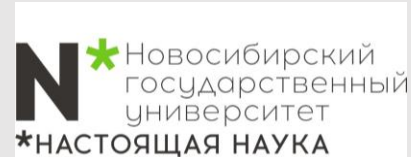


Давайте познакомимся



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 презентации

Мультидеревья

87 минут

**Итоги 1
семестра**


3 минуты

План презентации



Мультидеревья

87 минут



**Итоги 1
семестра**

3 минуты

Введение

Мы обсудили, что каждый узел в бинарном дереве поиска содержит одно значение и два указателя, левый и правый, которые указывают на левое и правое поддеревья узла соответственно. Та же концепция используется в M -канальном дереве поиска, которое имеет $M - 1$ значений на узел и M поддеревьев. В таком дереве M называется степенью дерева. Обратите внимание, что в бинарном дереве поиска $M = 2$, поэтому оно имеет одно значение и два поддерева. Другими словами, каждый внутренний узел M -way дерева поиска состоит из указателей на M поддеревья и содержит $M - 1$ ключей, где $M > 2$.

Указатель на левое поддерево	Значение или ключ узла	Указатель на правое поддерево
---------------------------------	---------------------------	----------------------------------

Структура узла бинарного дерева поиска

P_0	K_0	P_1	K_1	P_2	K_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	-------	-------	-------	---------	-----------	-----------	-------

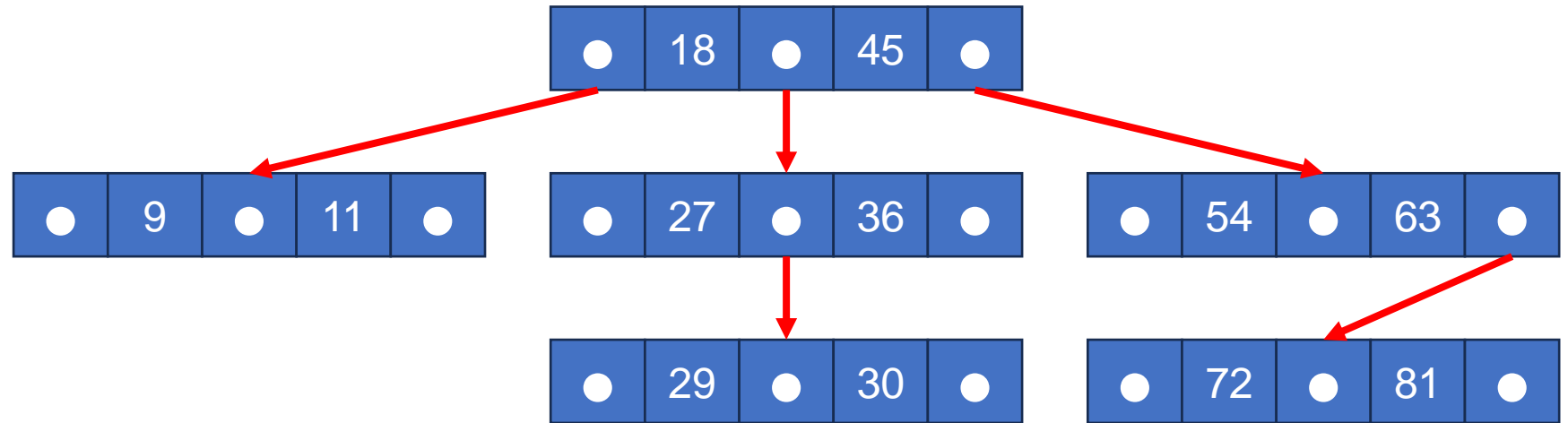
В показанной структуре $P_0, P_1, P_2, \dots, P_n$ являются указателями на поддеревья узла, а $K_0, K_1, K_2, \dots, K_{n-1}$ являются значениями ключей узла. Все значения ключей хранятся в порядке возрастания. То есть $K_i < K_{i+1}$ для $0 \leq i \leq n-2$.

Введение

В дереве поиска M-way не обязательно, чтобы каждый узел имел ровно $M-1$ значений и M поддеревьев. Вместо этого узел может иметь от 1 до $M-1$ значений, а количество поддеревьев может варьироваться от 0 (для листового узла) до $i + 1$, где i — количество значений ключей в узле. Таким образом, M — это фиксированный верхний предел, который определяет, сколько значений ключей может храниться в узле. Используя 3-стороннее дерево поиска, давайте изложим некоторые основные свойства M-стороннего дерева поиска.

- Все значения ключа в поддереве, указанном P_i , меньше K_i , где $0 \leq i \leq n-1$.
- Все значения ключей в поддереве, указанном P_i , больше K_{i-1} , где $0 \leq i \leq n-1$.

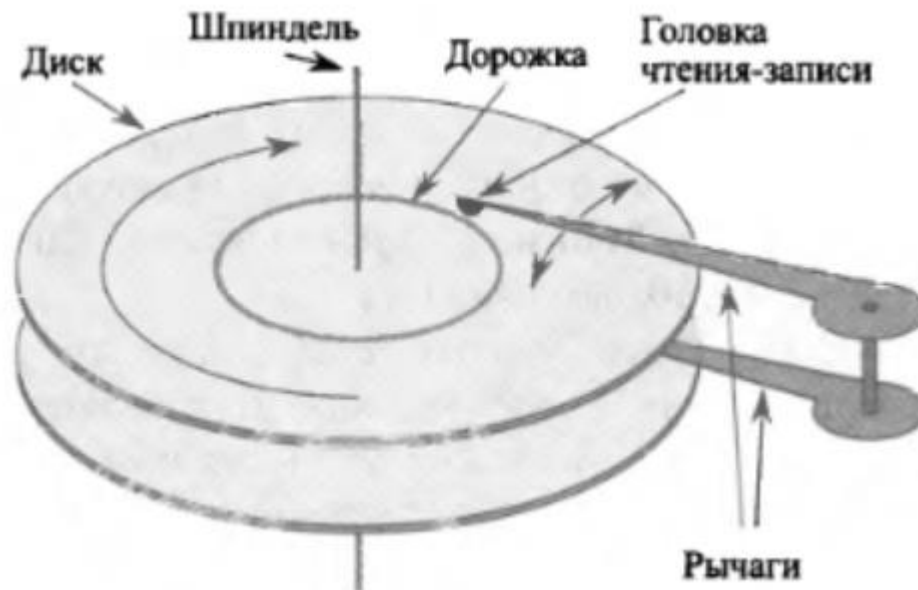
В M-way дереве поиска каждое поддерево также является M-way деревом поиска и следует тем же правилам.



Дерево поиска M-way порядка 3

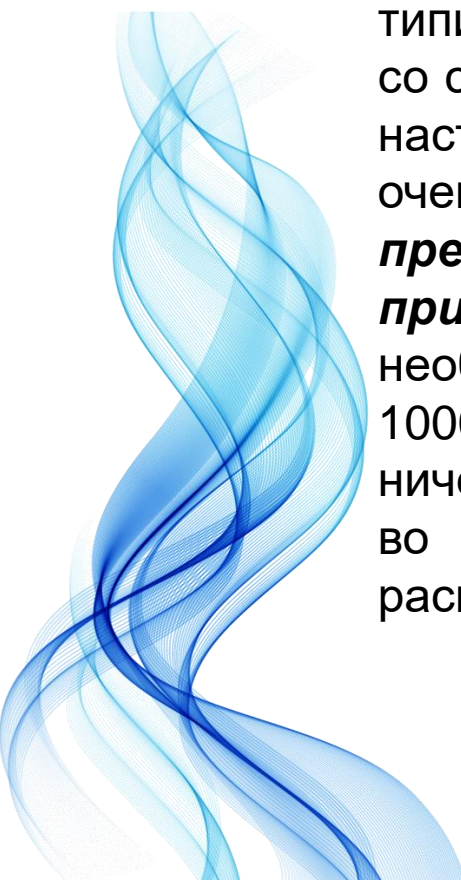
Структуры данных во вторичной памяти

Имеется несколько видов используемой компьютером памяти. Основная, или оперативная, память (primary, main memory) представляет собой специализированные микросхемы и обладает более высоким быстродействием и существенно более высокой ценой, чем магнитные носители, такие как магнитные диски или ленты. Большинство компьютеров, помимо первичной памяти, оснащено вторичной памятью (secondary storage) на базе магнитных дисков. Ее суммарный объем в типичной вычислительной системе как минимум на пару порядков превышает объем первичной памяти.



Типичный дисковый накопитель. Он состоит из одного или нескольких дисков (на рисунке их два), вращающихся на шпинделе. Каждый диск считывается и записывается с помощью головки на конце рычага. Рычаги вращаются на общей опорной оси. Дорожка представляет собой поверхность диска, проходящую под головкой, когда та находится в зафиксированном положении.

Структуры данных во вторичной памяти



Хотя диски существенно дешевле оперативной памяти и имеют высокую емкость, из-за механически движущихся частей они гораздо, гораздо медленнее оперативной памяти. Механическое движение головки относительно диска определяется двумя компонентами - перемещением головки по радиусу и вращением дисков. Когда были написаны эти строки, типичная **скорость вращения дисков** составляла **5400-15 000 об/мин (rpm)**. Обычно диски со скоростью 15 000 об/мин можно встретить в мощных серверах, скорость 7200 обычна для настольных систем, а 5400 - для переносных. Хотя скорость 7200 об/мин может показаться очень большой, **один оборот требует примерно 8.33 мс, что более чем на 5 порядков превышает время обращения к оперативной памяти (которое обычно составляет примерно 50 нс)**. Другими словами, пока мы ждем оборота диска, чтобы считать необходимые нам данные, из оперативной памяти мы могли бы получить эти данные почти 100000 раз! В среднем приходится ждать только половину оборота диска, но это практически ничего не меняет. Радиальное перемещение головок также требует времени. Одним словом, во время написания этих строк среднее время доступа к дисковой памяти для распространенных дисков составляло 8-11 мс.

Структуры данных во вторичной памяти

Для того чтобы сократить время ожидания, связанное с механическим перемещением, при обращении к диску выполняется обращение сразу к нескольким элементам, хранящимся на диске. Информация разделяется на несколько страниц (pages) одинакового размера, которые хранятся последовательно одна за другой в пределах одной дорожки, и каждая операция чтения или записи работает с одной или несколькими страницами. Для типичного диска **размер страницы может составлять 211-214 байт**. После того как головка позиционируется на нужную дорожку, а диск поворачивается так, что головка становится на начало интересующей нас страницы, операции чтения и записи выполняются очень быстро. Зачастую обработка прочитанной информации длится меньше времени, чем ее поиск и чтение с диска. По этой причине мы отдельно рассматриваем два компонента времени работы алгоритма:

- количество обращений к диску;
- время вычислений (процессорное время).

Количество обращений к диску измеряется в количествах страниц информации, которое должно быть считано с диска или записано на него. Заметим, что время обращения к диску не является постоянной величиной, поскольку зависит от расстояния между текущей дорожкой и дорожкой с интересующей нас информацией, а также от текущего угла поворота диска. Мы будем игнорировать это обстоятельство и в качестве первого приближения времени, необходимого для обращения к диску, будем использовать просто количество считываемых или записываемых страниц.

Структуры данных во вторичной памяти

В типичном приложении, использующем В-деревья, количество обрабатываемых данных так велико, что все они не могут одновременно разместиться в оперативной памяти. Алгоритмы работы с В-деревьями копируют в оперативную память с диска только некоторые выбранные страницы, необходимые для работы, и вновь записывают на диск те из них, которые были изменены в процессе работы. Алгоритмы работы с В-деревьями сконструированы таким образом, чтобы в любой момент времени обходиться только некоторым постоянным количеством страниц в основной памяти, так что ее объем не ограничивает размер В-деревьев, с которыми могут работать алгоритмы.

Пусть x - указатель на объект. Если объект находится в оперативной памяти компьютера, то мы обращаемся к его атрибутам обычным способом, например, как к $x.key$. Если же объект, к которому мы обращаемся посредством x , находится на диске, то мы должны выполнить операцию **DISK-READ(x)** для чтения объекта x в оперативную память перед тем, как будем обращаться к его полям. Аналогично для сохранения любых изменений в полях объекта x выполняется операция **DISK-WRITE(x)**.

Таким образом, типичный шаблон работы с объектом x имеет следующий вид:

x = a pointer to some object

Disk-Read(x)

Операции, обращающиеся к атрибутам x и/или изменяющие их

Disk-Write(x) // Не выполняется, если никакие атрибуты x не были изменены

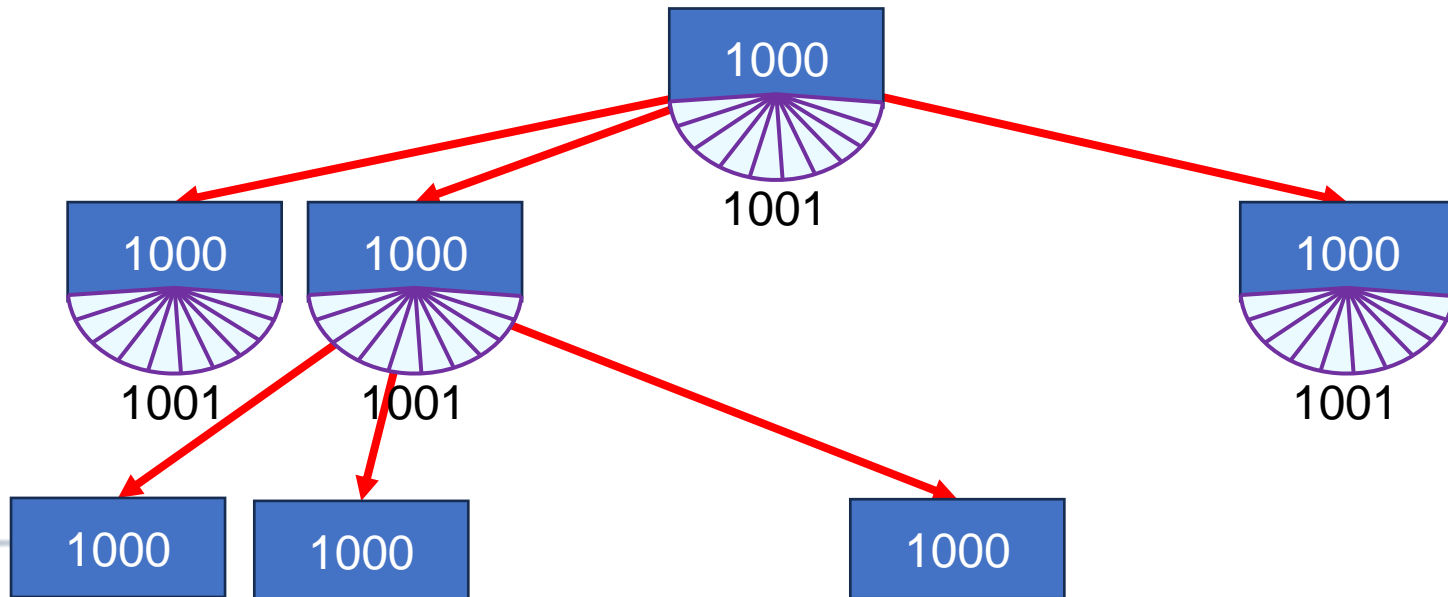
Прочие операции, обращающиеся к атрибутам x , но не изменяющие их

Структуры данных во вторичной памяти

Система в состоянии поддерживать в процессе работы в оперативной памяти только некоторое ограниченное количество страниц.

Поскольку в большинстве систем время выполнения алгоритма, работающего с В-деревьями, зависит, в первую очередь, от количества выполняемых операций с диском Disk-Read и Disk-Write, желательно минимизировать их количество и за один раз считывать и записывать как можно больше информации. Таким образом, размер узла В-дерева обычно соответствует дисковой странице. Количество потомков узла В-дерева, таким образом, ограничивается размером дисковой страницы.

Для больших В-деревьев, хранящихся на диске, степень ветвления обычно находится между 50 и 2000, в зависимости от размера ключа относительно размера страницы. Большая степень ветвления резко снижает как высоту дерева, так и количество обращений к диску для поиска ключа.



1 узел,
1000 ключей

1001 узел,
1 001 000 ключей

1002001 узел,
1002001000 ключей

В - деревья

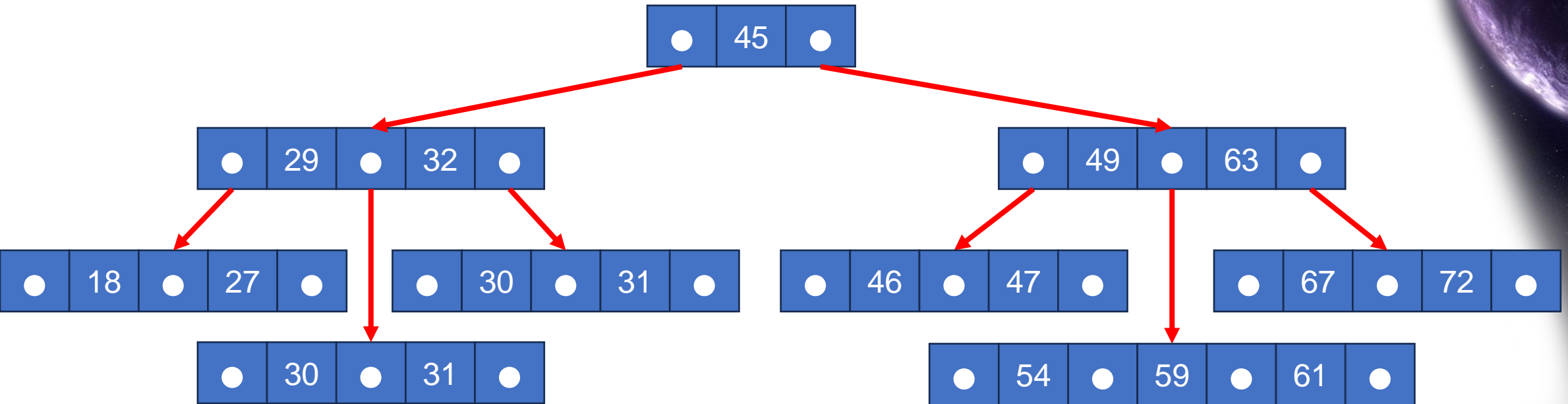
В-дерево — это специализированное М-way дерево, разработанное Рудольфом Байером и Эдом МакКрейтом в 1970 году, которое широко используется для доступа к диску. В-дерево порядка m может иметь максимум $m-1$ ключей и m указателей на свои поддеревья. В-дерево может содержать большое количество значений ключей и указателей на поддеревья. Хранение большого количества ключей в одном узле сохраняет высоту дерева относительно небольшой.

В-дерево предназначено для хранения отсортированных данных и позволяет выполнять операции поиска, вставки и удаления за логарифмическое амортизированное время. В-дерево порядка m (максимальное количество потомков, которое может иметь каждый узел) — это дерево со всеми свойствами М-путевого дерева поиска. Кроме того, оно обладает следующими свойствами:

1. Каждый узел в В-дереве имеет не более (максимум) m потомков.
2. Каждый узел в В-дереве, за исключением корневого узла и листовых узлов, имеет не менее (минимум) $m/2$ потомков.
3. Корневой узел имеет не менее двух потомков, если он не является конечным (листовым) узлом.
4. Все листовые узлы находятся на одном уровне.

В - деревья

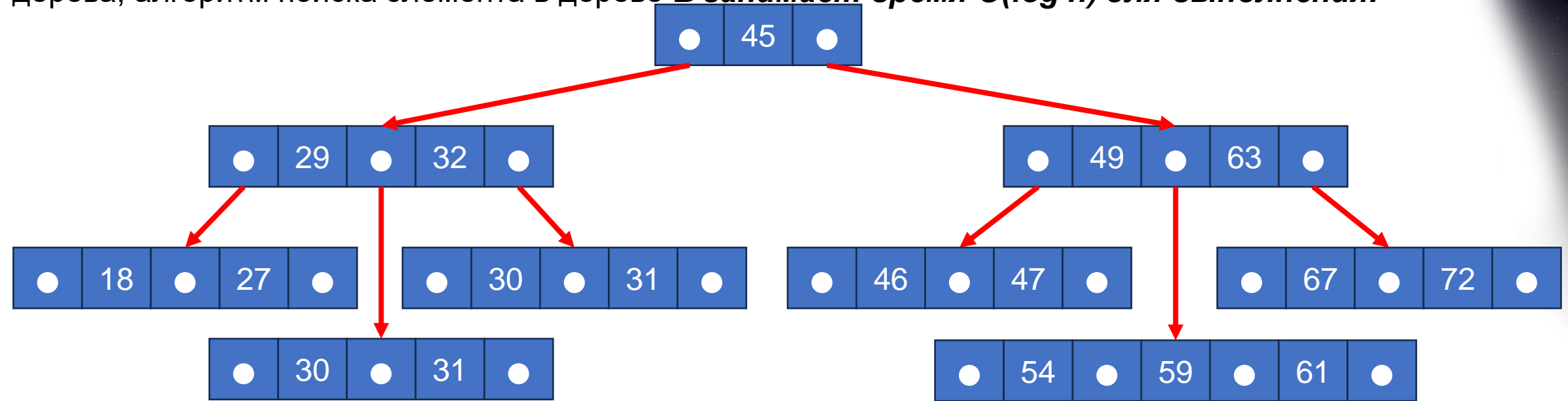
Внутренний узел в В-дереве может иметь n потомков, где $0 \leq n \leq m$. Не обязательно, чтобы каждый узел имел одинаковое количество потомков, но единственным ограничением является то, что узел должен иметь не менее $m/2$ потомков.



При выполнении операций вставки и удаления в В-дереве количество дочерних узлов может меняться. Поэтому, чтобы поддерживать минимальное количество дочерних узлов, внутренние узлы могут быть объединены или разделены.

Поиск элемента в В-дереве

Чтобы найти 59, мы начинаем с корневого узла. Корневой узел имеет значение 45, которое меньше 59. Поэтому мы перемещаемся по правому поддереву. Правое поддерево корневого узла имеет два ключевых значения, 49 и 63. Так как $49 \leq 59 \leq 63$, мы проходим правое поддерево 49, то есть левое поддерево 63. Это поддерево имеет три значения, 54, 59 и 61. При нахождении значения 59 поиск успешен. Возьмем другой пример. Если вы хотите найти 9, то мы проходим левое поддерево корневого узла. Левое поддерево имеет два ключевых значения, 29 и 32. Снова мы проходим левое поддерево 29. Мы обнаруживаем, что у него есть два ключевых значения, 18 и 27. Левое поддерево 18 нет, поэтому значение 9 не сохраняется в дереве. Поскольку время выполнения операции поиска зависит от высоты дерева, алгоритм поиска элемента в дереве **В занимает время $O(\log n)$ для выполнения.**



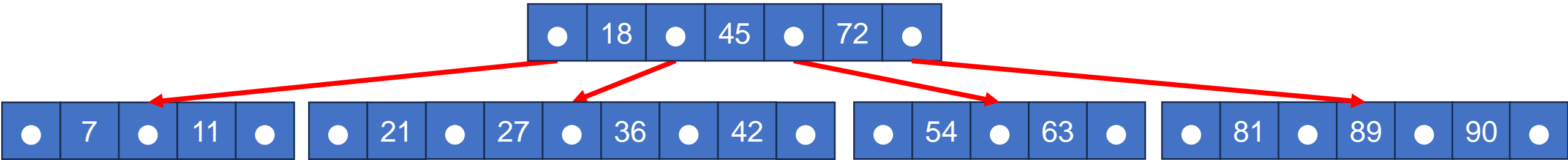
Вставка нового элемента в B-дерево

В B-дереве все вставки выполняются на уровне конечных узлов. Новое значение вставляется в B-дерево с использованием приведенного ниже алгоритма.

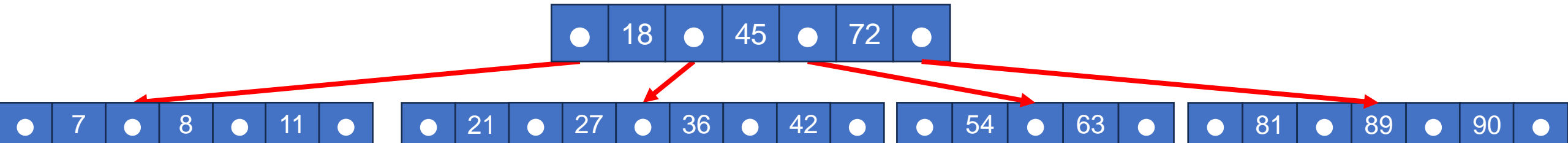
1. Найдите в B-дереве конечный узел, куда следует вставить новое значение ключа.
2. Если конечный узел не заполнен, то есть содержит менее $m-1$ значений ключа, то вставьте новый элемент в узел, сохраняя порядок элементов узла.
3. Если конечный узел заполнен, то есть уже содержит $m-1$ значений ключа, то
 - (a) вставьте новое значение по порядку в существующий набор ключей,
 - (b) разделите узел по его медиане на два узла (обратите внимание, что разделенные узлы наполовину заполнены), и
 - (c) подтолкните медианный элемент к его родительскому узлу. Если родительский узел уже заполнен, то разделите родительский узел, выполнив те же шаги.



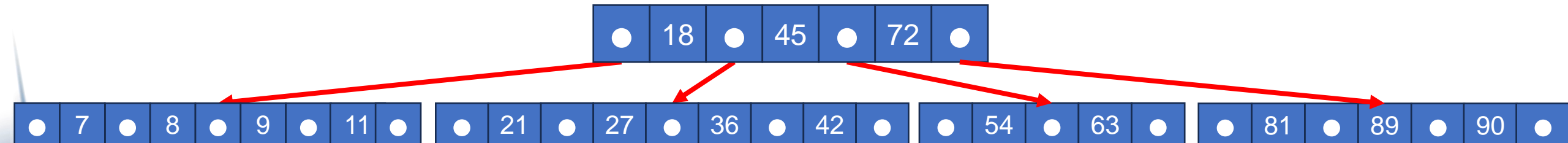
Вставка нового элемента в В-дерево



Шаг 1: Вставка 8

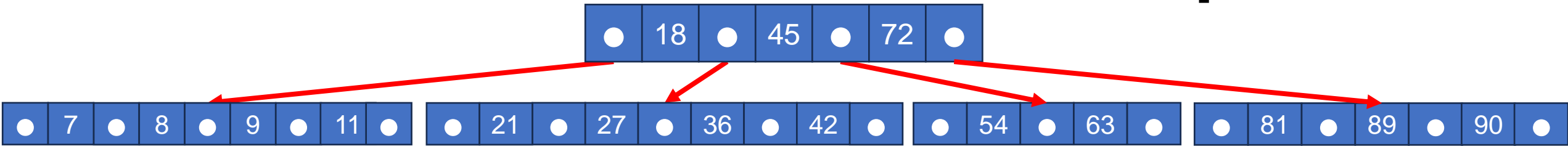


Шаг 2: Вставка 9



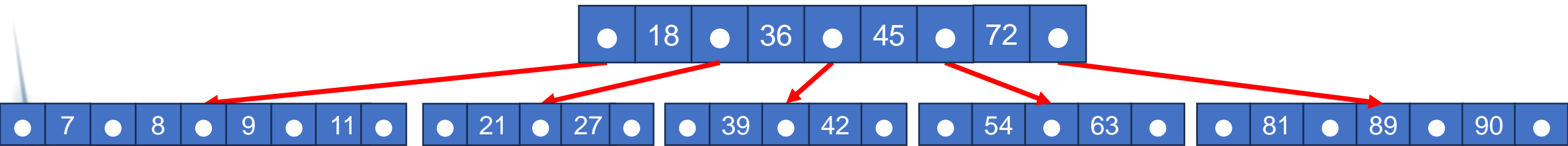
Посмотрите на В-дерево порядка 5 и вставим в него 8, 9, 39 и 4.

Вставка нового элемента в В-дерево



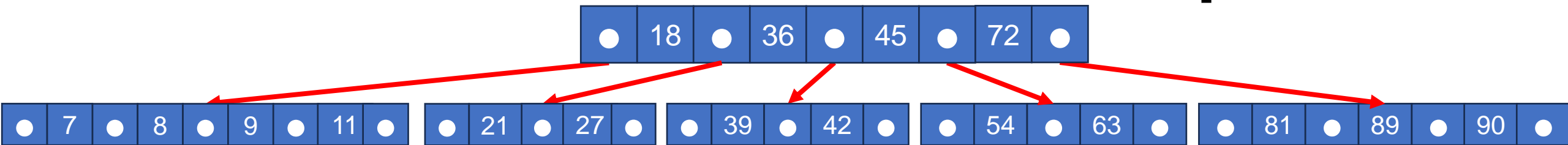
Шаг 3: Вставка 39

До сих пор мы легко вставляли 8 и 9 в дерево, потому что листовые узлы не были заполнены. Но теперь узел, в который должен быть вставлен 39, уже заполнен, так как он содержит четыре значения. Здесь мы разделяем узлы, чтобы сформировать два отдельных узла. Но перед разделением расположим ключевые значения по порядку (включая новое значение). Упорядоченный набор значений задан как 21, 27, 36, 39 и 42. Медианное значение равно 36, поэтому вставьте 36 в его родительский узел и разделите листовые узлы.



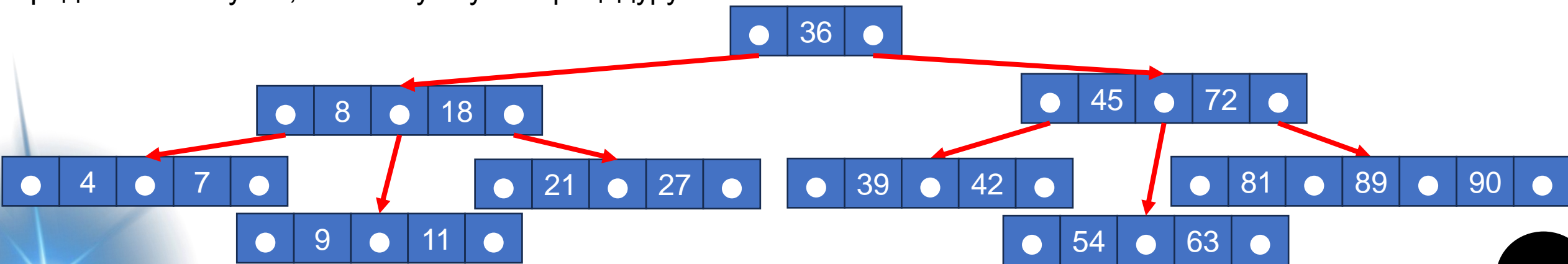
Посмотрите на В-дерево порядка 5 и вставим в него 8, 9, 39 и 4.

Вставка нового элемента в В-дерево



Шаг 4: Вставка 4

Теперь узел, в который должен быть вставлен 4, уже заполнен, так как он содержит четыре ключевых значения. Здесь мы разделяем узлы, чтобы сформировать два отдельных узла. Но перед разделением мы упорядочиваем ключевые значения (включая новое значение). Упорядоченный набор значений задан как 4, 7, 8, 9 и 11. Медианное значение равно 8, поэтому мы помещаем 8 в его родительский узел и разделяем листовые узлы. Но снова мы видим, что родительский узел уже заполнен, поэтому мы разделяем родительский узел, используя ту же процедуру.



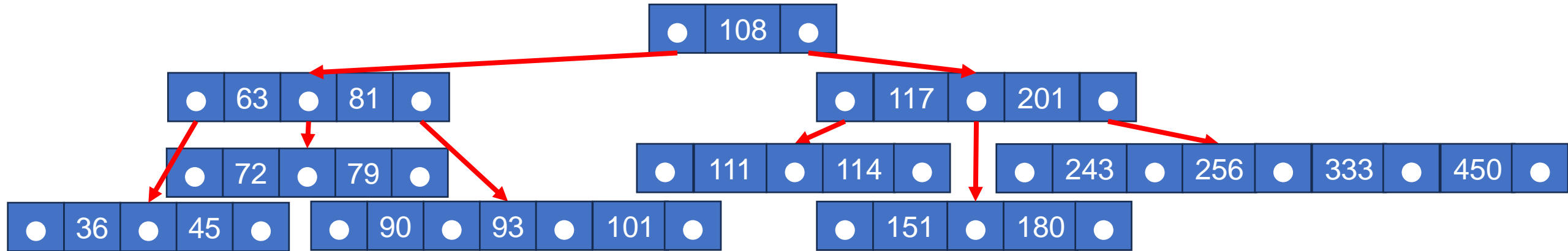
Посмотрите на В-дерево порядка 5 и вставим в него 8, 9, 39 и 4.

Удаление элемента из В-дерева

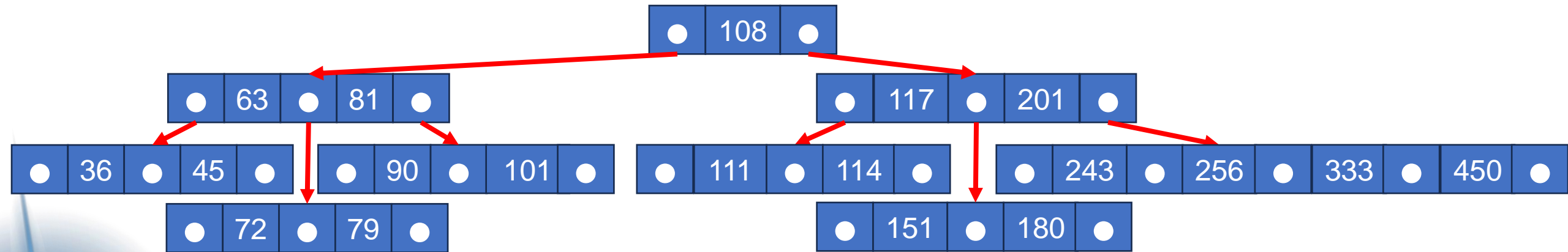
Подобно вставке, удаление также выполняется из листовых узлов. Существует два случая удаления. В первом случае листовой узел должен быть удален. Во втором случае должен быть удален внутренний узел. Давайте сначала рассмотрим шаги, необходимые для удаления листового узла.

1. Найдите листовой узел, который должен быть удален.
2. Если листовой узел содержит больше минимального количества ключевых значений (более $m/2$ элементов), то удалите значение.
3. Иначе, если листовой узел не содержит $m/2$ элементов, то заполните узел, взяв элемент либо из левого, либо из правого родственного узла.
 - (a) Если левый родственный узел имеет больше минимального количества ключевых значений, вставьте его наибольший ключ в родительский узел и вытяните промежуточный элемент из родительского узла в листовой узел, где ключ удаляется.
 - (b) Иначе, если правый сестринский узел имеет больше минимального количества значений ключей, поместите его наименьший ключ в его родительский узел и перетащите промежуточный элемент из родительского узла в конечный узел, где ключ удаляется.
4. Иначе, если и левый, и правый сестринский узел содержат только минимальное количество элементов, создайте новый конечный узел, объединив два конечных узла и промежуточный элемент родительского узла (убедившись, что количество элементов не превышает максимальное количество элементов, которое может иметь узел, то есть m). Если перетаскивание промежуточного элемента из родительского узла оставляет его с меньшим, чем минимальное количество ключей в узле, то распространите процесс вверх, тем самым уменьшив высоту В-дерева.

Удаление элемента из В-дерева

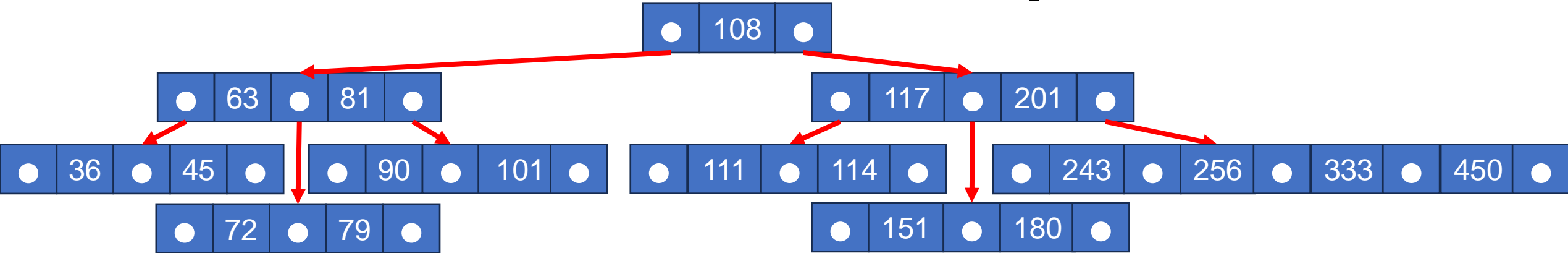


Шаг 1: Удаление 93

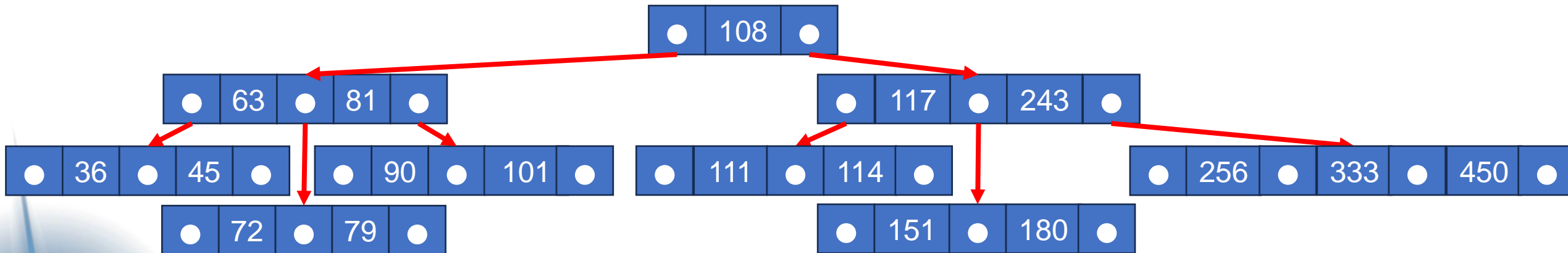


Рассмотрим дерево В порядка 5 и удалим из него значения 93, 201, 180 и 72

Удаление элемента из В-дерева

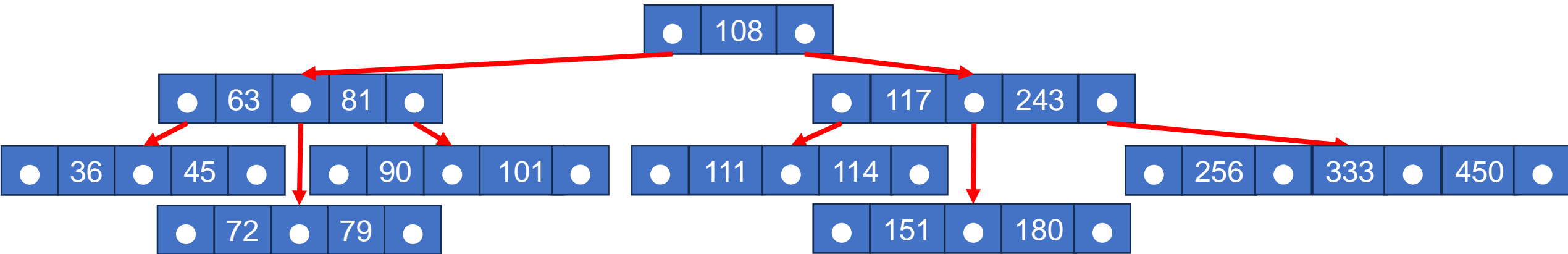


Шаг 2: Удаление 201

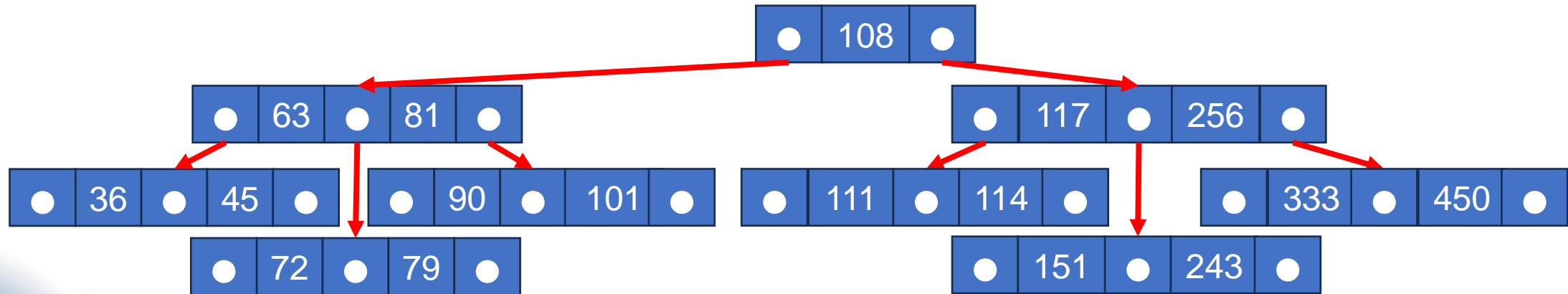


Рассмотрим дерево В порядка 5 и удалим из него значения 93, 201, 180 и 72

Удаление элемента из B-дерева

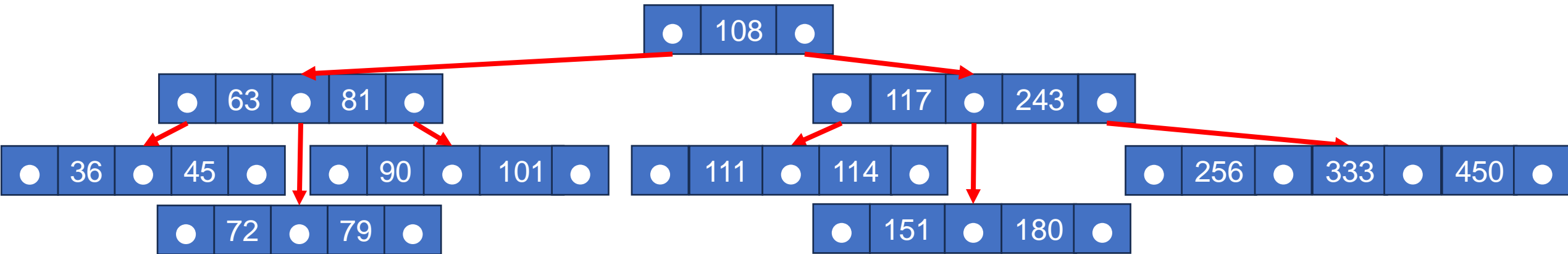


Шаг 3: Удаление 180

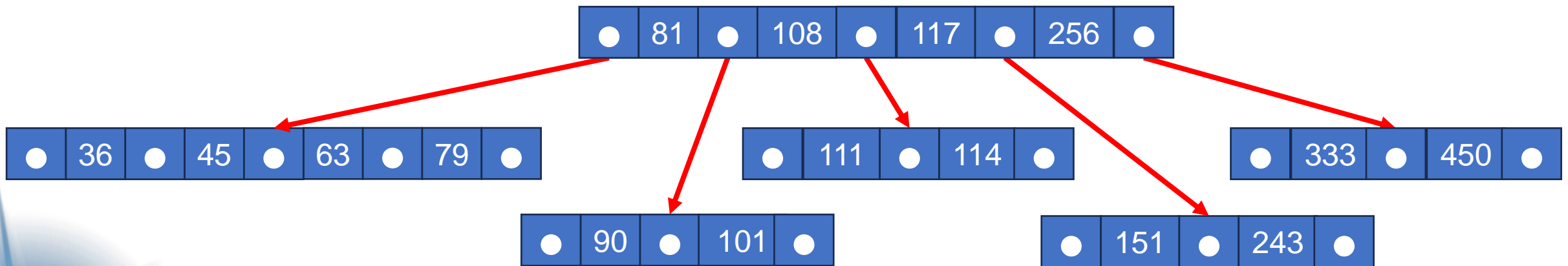


Рассмотрим дерево B порядка 5 и удалим из него значения 93, 201, 180 и 72

Удаление элемента из В-дерева

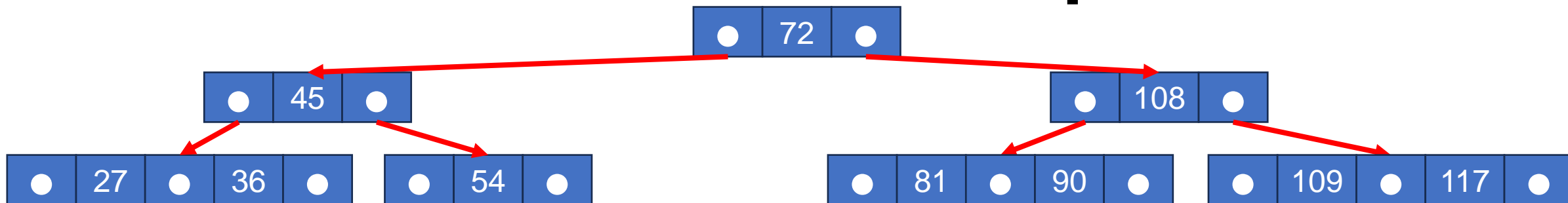


Шаг 4: Удаление 72

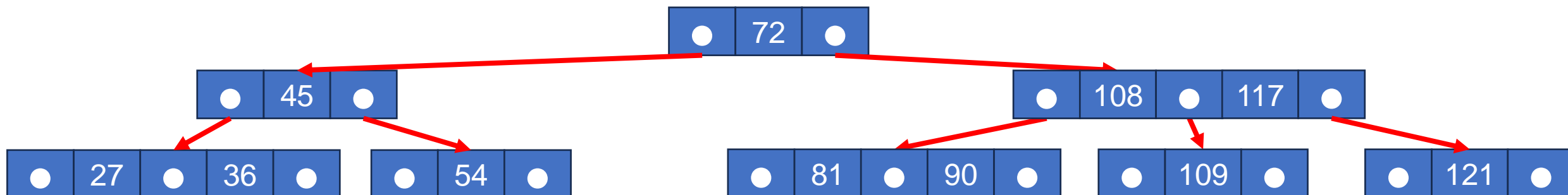


Рассмотрим дерево В порядка 5 и удалим из него значения 93, 201, 180 и 72

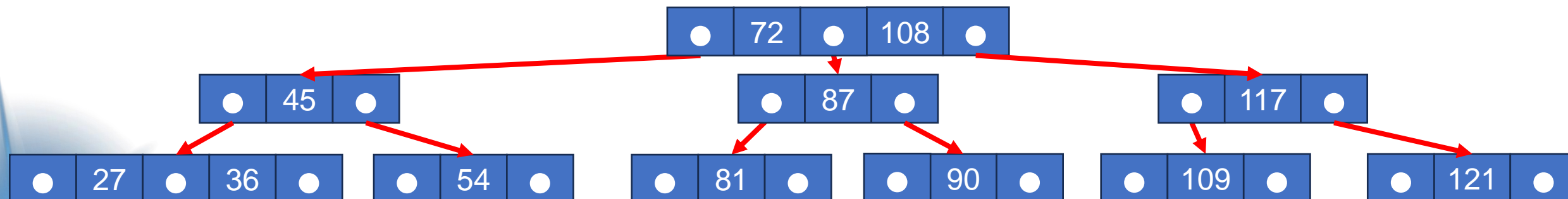
Удаление элемента из В-дерева



Шаг 1: Вставка 121

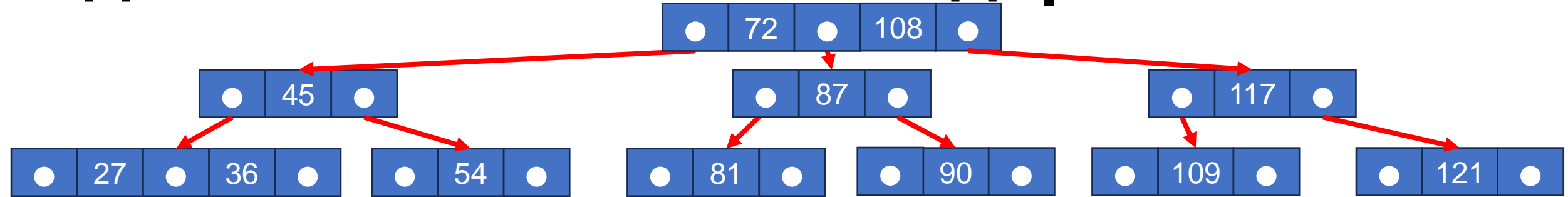


Шаг 2: Вставка 87

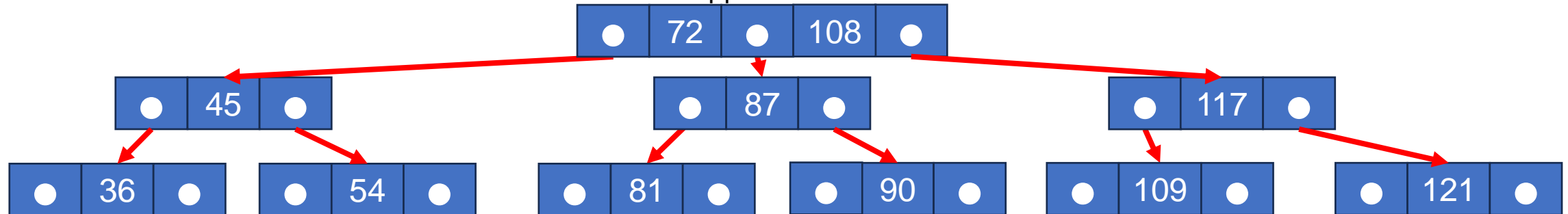


Рассмотрим В-дерево порядка 3 и выполним следующие операции: (а) вставим 121, 87, а затем (b) удалим 36, 109.

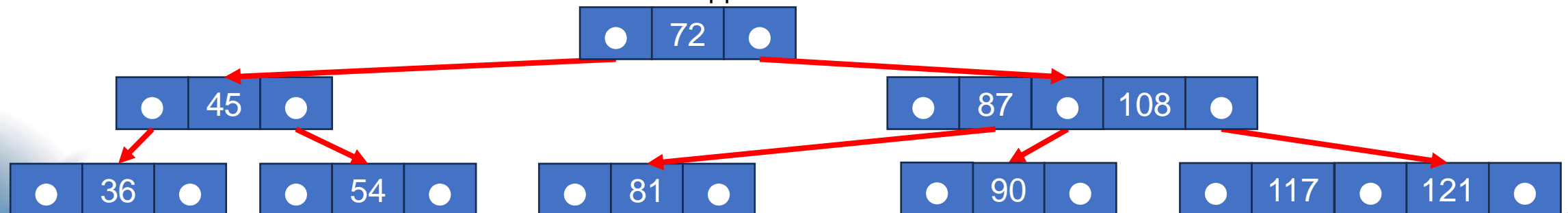
Удаление элемента из В-дерева



Шаг 3: Удалим 36



Шаг 4: Удалим 109



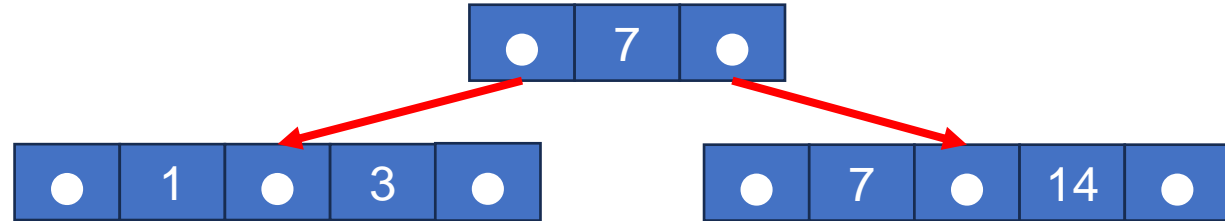
Рассмотрим В-дерево порядка 3 и выполним следующие операции: (а) вставим 121, 87, а затем (б) удалим 36, 109.

Удаление элемента из В-дерева

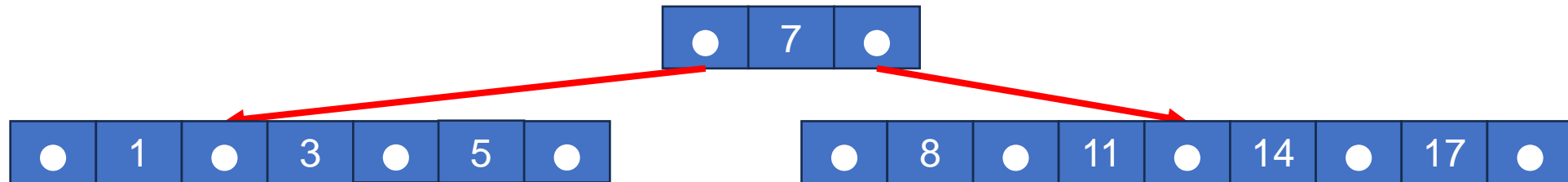
Шаг 1: Добавим 3, 14, 7, 1



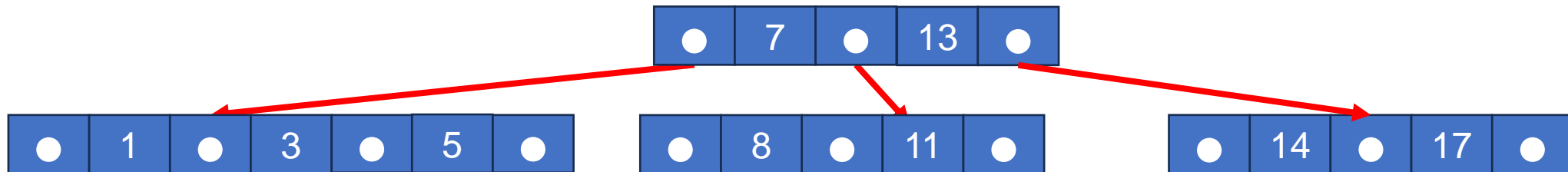
Шаг 2: Добавить 8



Шаг 3: Добавить 5, 11, 17

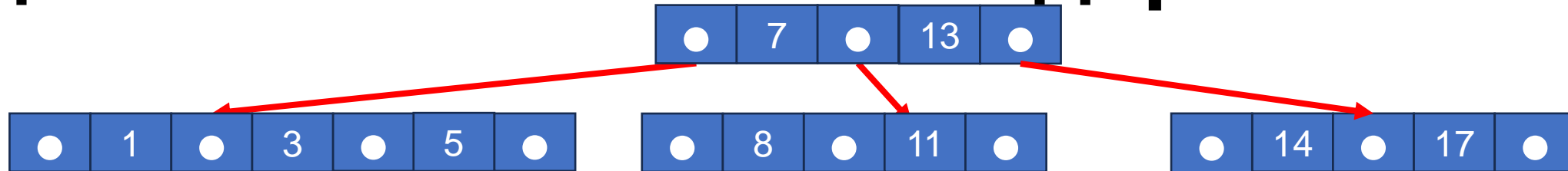


Шаг 4: Добавить 23

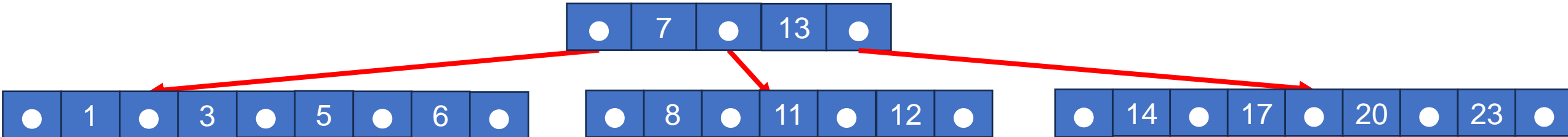


Создайте В-дерево порядка 5, вставив следующие элементы: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25 и 19.

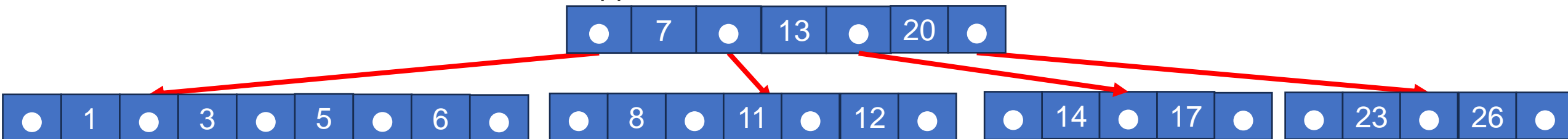
Удаление элемента из В-дерева



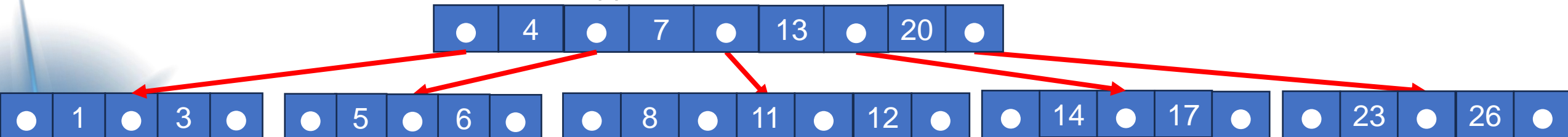
Шаг 5: Добавить 6, 23, 12, 20



Шаг 6: Добавить 26

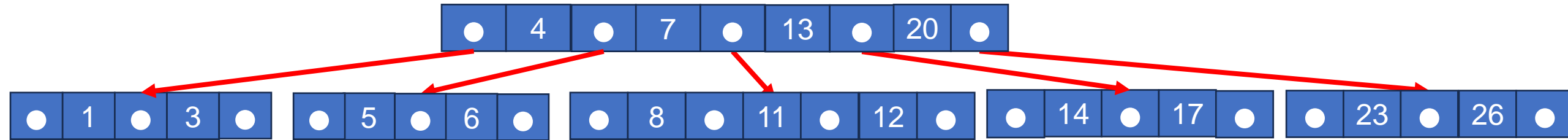


Шаг 7: Добавить 4

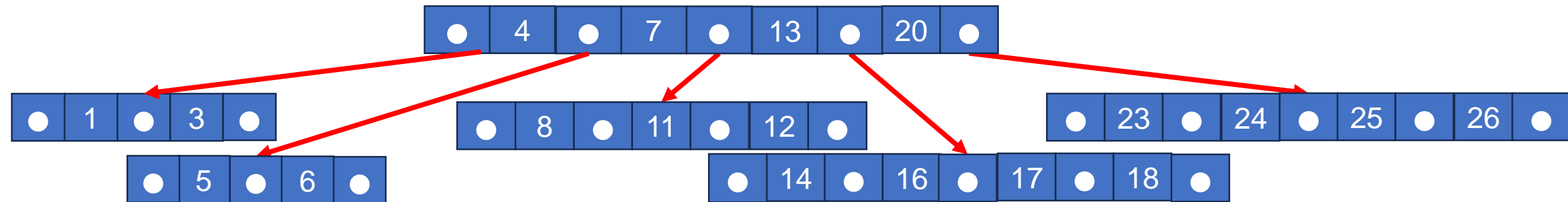


Создайте В-дерево порядка 5, вставив следующие элементы: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25 и 19.

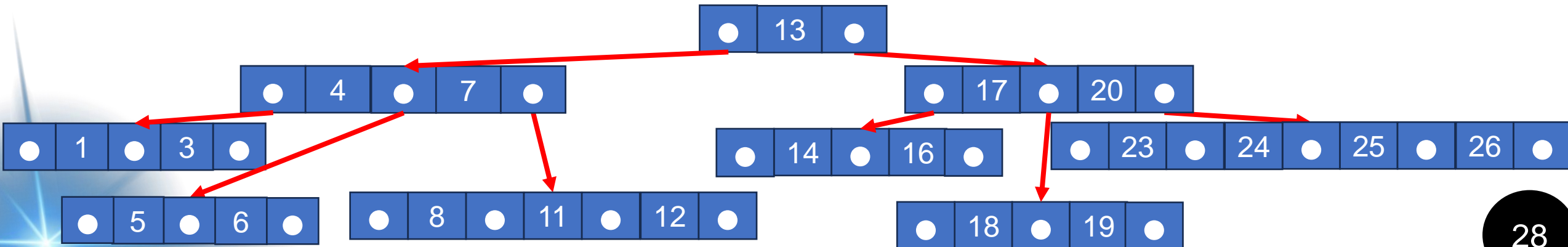
Удаление элемента из B-дерева



Шаг 8: Добавить 16,18,24,25



Шаг 9: Добавить 19



Применение В-деревьев

База данных — это набор связанных данных. Основная причина использования базы данных заключается в том, что она хранит организованные данные, чтобы облегчить пользователям обновление, извлечение и управление данными. Данные, хранящиеся в базе данных, могут включать имена, адреса, фотографии и номера. Например, учитель может захотеть вести базу данных всех учеников, которая включает имена, номера списков, дату рождения и оценки, полученные каждым учеником. В настоящее время базы данных используются в каждой отрасли для хранения сотен миллионов записей. В реальном мире не редкость, когда база данных хранит гигабайты и терабайты данных. Например, телекоммуникационная компания ведет базу данных счетов клиентов с более чем 50 миллиардами строк, содержащих терабайты данных. Мы знаем, что первичная память очень дорогая и способна хранить очень мало данных по сравнению с устройствами вторичной памяти, такими как магнитные диски. Кроме того, оперативная память по своей природе энергозависима, и мы не можем хранить все данные в первичной памяти. У нас нет другого выбора, кроме как хранить данные на вторичных устройствах хранения. Но доступ к данным с магнитных дисков в 10 000–1 000 000 раз медленнее, чем доступ к ним из основной памяти. Поэтому В-деревья часто используются для индексации данных и обеспечения быстрого доступа. Рассмотрим ситуацию, в которой нам нужно выполнить поиск в неиндексированной и несортированной базе данных, содержащей n значений ключей. Наихудшее время выполнения этой операции составит $O(n)$. Напротив, если данные в базе данных индексированы с помощью В-дерева, та же операция поиска будет выполняться за $O(\log n)$.

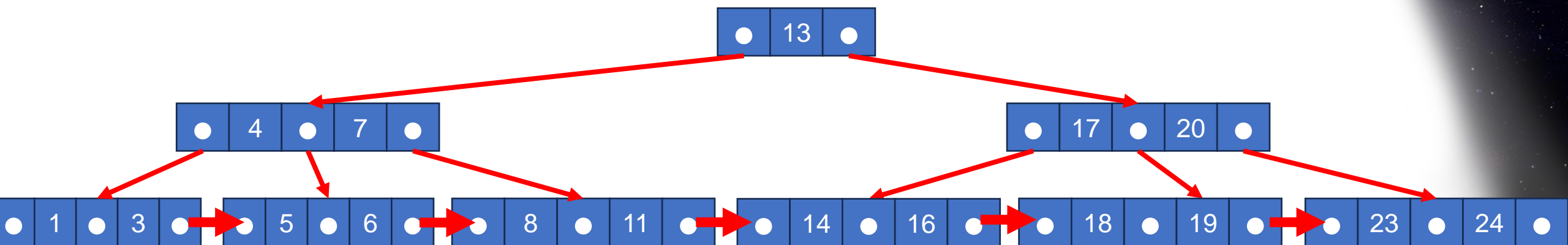
Применение В-деревьев

Например, поиск одного ключа в наборе из миллиона ключей потребует максимум 1 000 000 сравнений. Но если те же данные индексируются с помощью В-дерева порядка 10, то в худшем случае потребуется всего 114 сравнений. Следовательно, мы видим, что индексирование больших объемов данных может обеспечить значительный прирост производительности операций поиска. Когда мы используем В-деревья или обобщенные М-образные деревья поиска, значение m или порядок В-деревьев часто очень велики. Обычно они варьируются от 128 до 512. Это означает, что один узел в дереве может содержать 127–511 ключей и 128–512 указателей на дочерние узлы. Мы берем большое значение m в основном по трем причинам:

1. Доступ к диску очень медленный. Мы должны иметь возможность извлекать большой объем данных за один доступ к диску.
2. Диск — это блочно-ориентированное устройство. То есть данные организованы и извлекаются в терминах блоков. Поэтому при использовании В-дерева (обобщенного М-образного дерева поиска) используется большое значение m , чтобы один узел дерева мог занимать весь блок. Другими словами, m представляет собой максимальное количество элементов данных, которые могут быть сохранены в одном блоке. m максимизируется для ускорения обработки. Чем больше данных хранится в блоке, тем меньше времени требуется для их перемещения в основную память.
3. Большое значение минимизирует высоту дерева. Таким образом, операция поиска становится действительно быстрой.

B+ - деревья

Дерево B+ — это вариант дерева B, которое хранит отсортированные данные таким образом, что позволяет эффективно вставлять, извлекать и удалять записи, каждая из которых идентифицируется ключом. В то время как дерево B может хранить как ключи, так и записи во внутренних узлах, дерево B+, напротив, хранит все записи на уровне листьев дерева; во внутренних узлах хранятся только ключи. Листовые узлы дерева B+ часто связаны друг с другом в связанном списке. Это имеет дополнительное преимущество, делая запросы более простыми и эффективными. Обычно деревья B+ используются для хранения больших объемов данных, которые не могут быть сохранены в основной памяти. В деревьях B+ вторичное хранилище (магнитный диск) используется для хранения листовых узлов деревьев, а внутренние узлы деревьев хранятся в основной памяти.



B+ - деревья

Многие системы баз данных реализованы с использованием структуры дерева B+ из-за ее простоты. Поскольку все данные появляются в конечных узлах и упорядочены, дерево всегда сбалансировано и делает поиск данных эффективным.

Дерево B+ можно рассматривать как многоуровневый индекс, в котором листья составляют плотный индекс, а неконечные узлы составляют разреженный индекс. Преимущества деревьев B+ можно представить следующим образом:

1. Записи могут быть извлечены за равное количество обращений к диску
2. Его можно использовать для легкого выполнения широкого спектра запросов, поскольку листья связаны с узлами на верхнем уровне
3. Высота дерева меньше и сбалансирована
4. Поддерживает как случайный, так и последовательный доступ к записям
5. Ключи используются для индексации



Сравнение В и В+ деревьев

В деревья

- 1. Ключи поиска не повторяются
 - 2. Данные хранятся во внутренних или конечных узлах
 - 3. Поиск занимает больше времени, так как данные могут быть найдены в конечных или неконечных узлах
 - 4. Удаление неконечных узлов очень сложно
 - 5. Конечные узлы нельзя хранить с помощью связанных списков
 - 6. Структура и операции сложны
- Цифра

В+ деревья

- 1. Хранит избыточный ключ поиска
- 2. Данные хранятся только в конечных узлах
- 3. Поиск данных очень прост, так как данные могут быть найдены только в конечных узлах
- 4. Удаление очень просто, так как данные будут находиться в конечных узлах
- 5. Данные конечных узлов упорядочиваются с помощью последовательных связанных списков
- 6. Структура и операции просты



Вставка нового элемента в B+ дерево

Новый элемент просто добавляется в конечный узел, если для него есть место. Но если узел данных в дереве, куда необходимо вставить, заполнен, то этот узел разделяется на два узла. Это требует добавления нового значения индекса в родительский узел индекса, чтобы будущие запросы могли проводить арбитраж между двумя новыми узлами. Однако добавление нового значения индекса в родительский узел может привести к его разделению. Фактически, все узлы на пути от листа к корню могут разделиться, когда новое значение добавляется в листовой узел. Если корневой узел разделяется, создается новый листовой узел, и дерево увеличивается на один уровень.

Шаги по вставке нового узла в дерево B+:

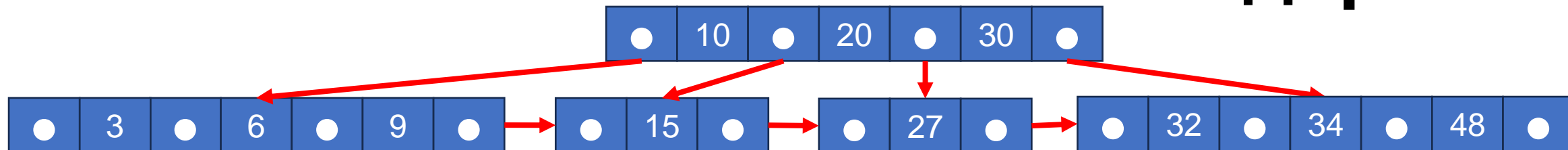
Шаг 1: Вставьте новый узел как листовой узел.

Шаг 2: Если листовой узел переполнен, разделите узел и скопируйте средний элемент в следующий индексный узел.

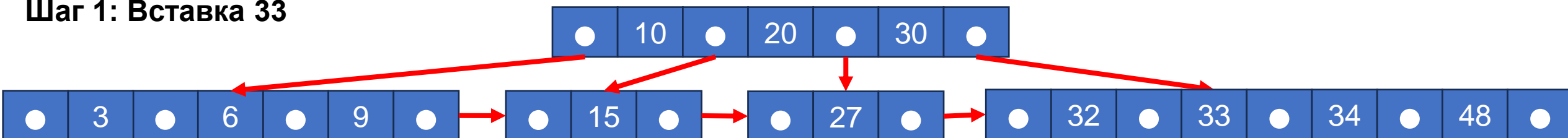
Шаг 3: Если индексный узел переполнен, разделите этот узел и переместите средний элемент на следующую страницу индекса.



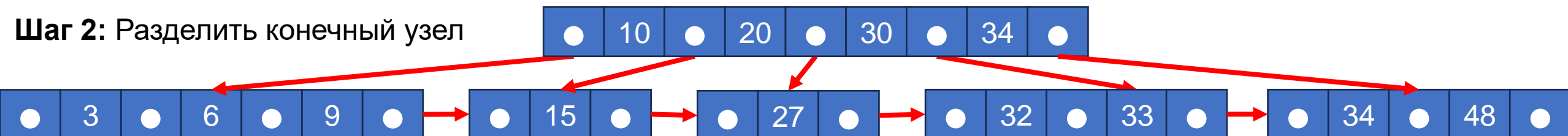
Вставка нового элемента в B+ дерево



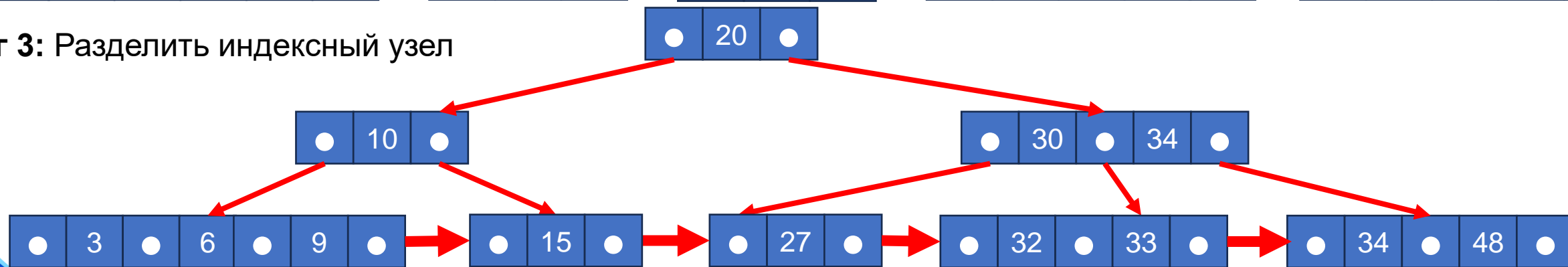
Шаг 1: Вставка 33



Шаг 2: Разделить конечный узел



Шаг 3: Разделить индексный узел



Вставка узла 33 в заданное дерево B+ порядка 4

Удаление элемента из дерева B+

Как и в деревьях B, удаление всегда выполняется из листового узла. Если удаление элемента данных оставляет этот узел пустым, то соседние узлы проверяются и объединяются с недозаполненным узлом. Этот процесс требует удаления значения индекса из родительского индексного узла, что, в свою очередь, может привести к его опустошению. Подобно процессу вставки, удаление может вызвать волну слияния-удаления, которая будет проходить от листового узла до корня. Это приводит к сжатию дерева на один уровень.

Шаги по удалению узла из дерева B+:

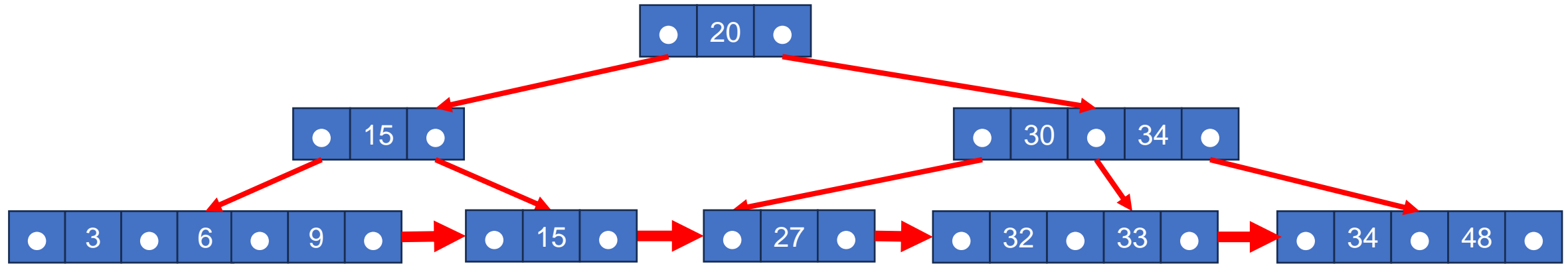
Шаг 1: Удалить ключ и данные из листьев.

Шаг 2: Если листовой узел переполняется, объединить этот узел с родственным узлом и удалить ключ между ними.

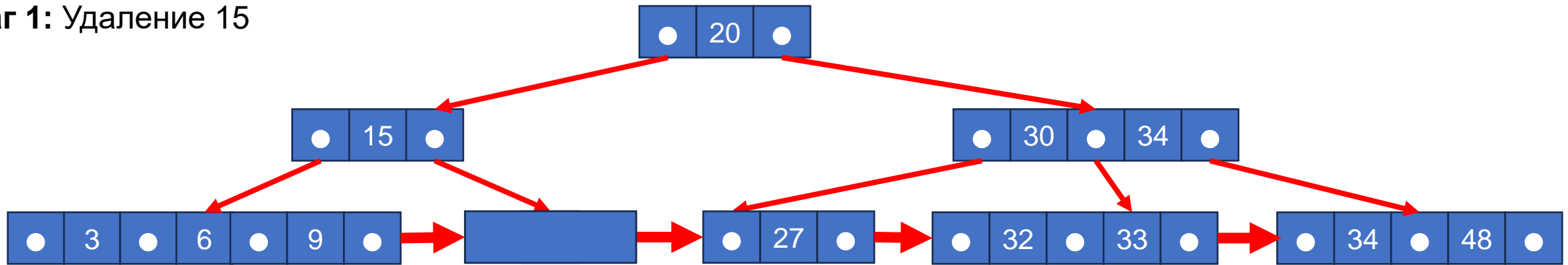
Шаг 3: Если индексный узел переполняется, объединить этот узел с родственным узлом и переместить ключ между ними вниз.



Вставка нового элемента в B+ дерево



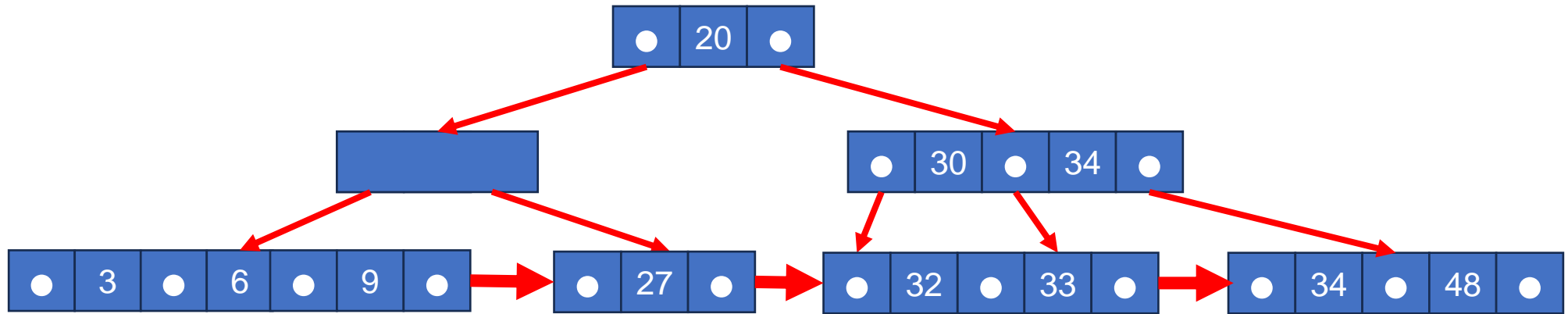
Шаг 1: Удаление 15



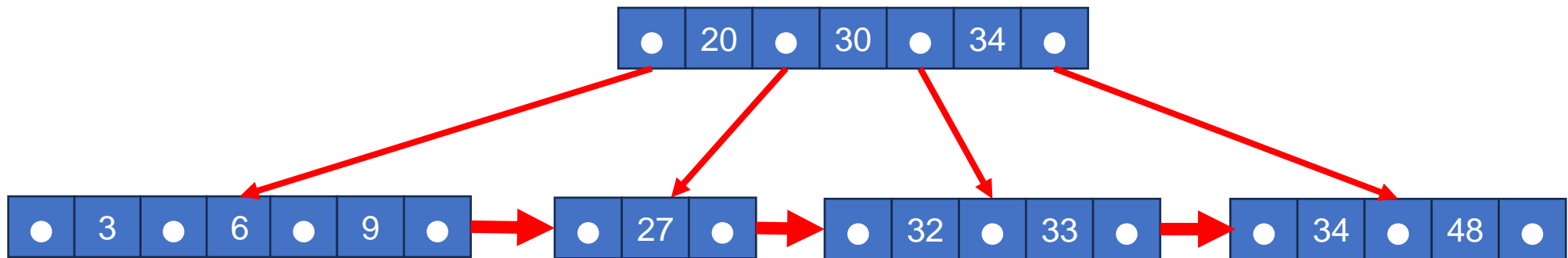
Шаг 2: Конечный узел переполняется, поэтому объединим с левым сестринским узлом и удалим ключ 15

Вставка нового элемента в B+ дерево

Шаг 2: Конечный узел переполняется, поэтому объединим с левым сестринским узлом и удалим ключ 15



Шаг 3: Теперь индексный узел переполняется, поэтому объединим с сестринским узлом и удалим узел



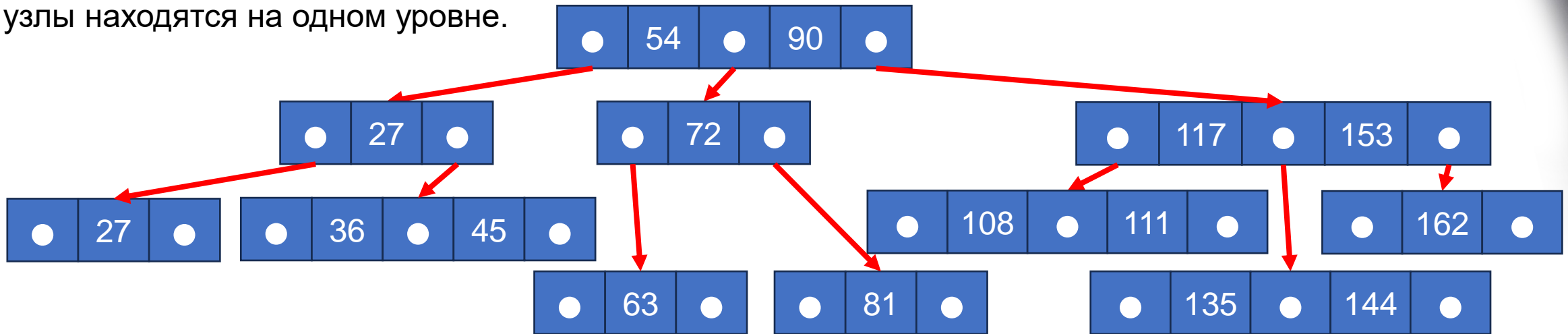
2-3 деревья

На прошлой лекции мы увидели, что для двоичных деревьев поиска среднее время для таких операций, как поиск/вставка/удаление, составляет $O(\log N)$, а худшее время составляет $O(N)$, где N — количество узлов в дереве. Однако сбалансированное дерево высотой $O(\log N)$ всегда гарантирует время $O(\log N)$ для всех трех методов. Типичные примеры сбалансированных по высоте деревьев включают деревья AVL, красно-черные деревья, B-деревья и 2-3 деревья.

В 2-3 дереве каждый внутренний узел имеет либо два, либо три потомка.

- Узлы с двумя потомками называются 2-узлами. 2-узлы имеют одно значение данных и два потомка
- Узлы с тремя потомками называются 3-узлами. 3-узлы имеют два значения данных и три детей (левый ребенок, средний ребенок и правый ребенок)

Это означает, что дерево 2-3 не является бинарным деревом. В этом дереве все конечные узлы находятся на одном уровне.



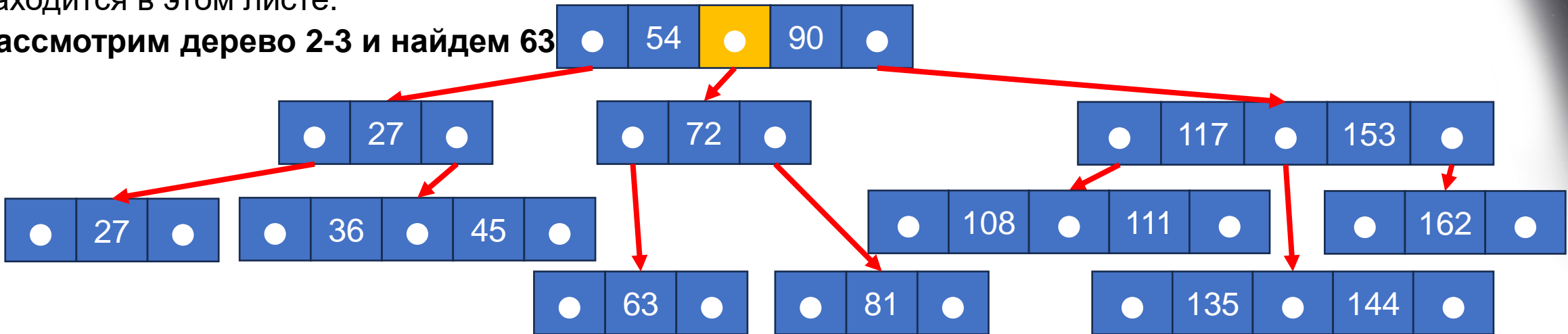
Поиск элемента в дереве 2-3

Операция поиска используется для определения того, присутствует ли значение данных x в дереве 2-3. Процесс поиска значения в дереве 2-3 очень похож на поиск значения в бинарном дереве поиска. Поиск значения данных x начинается с корня. Если k_1 и k_2 — два значения, хранящиеся в корневом узле, то

- если $x < k_1$, перейти к левому дочернему элементу.
- если $x \geq k_1$ и у узла только два дочерних элемента, перейти к правому дочернему элементу.
- если $x \geq k_1$ и у узла три дочерних элемента, то переходим к среднему дочернему элементу, если $x < k_2$, иначе к правому дочернему элементу, если $x \geq k_2$.

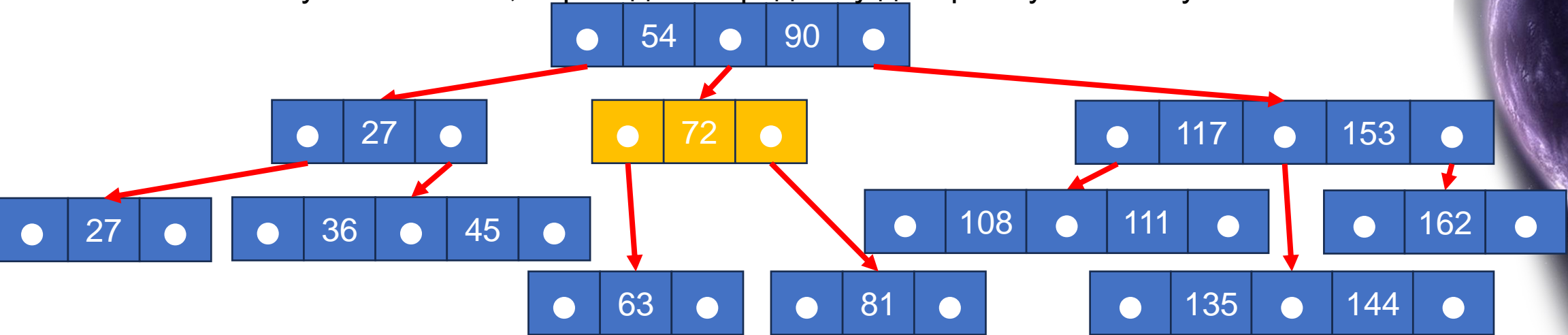
В конце процесса узел со значением данных x достигается тогда и только тогда, когда x находится в этом листе.

Рассмотрим дерево 2-3 и найдем 63

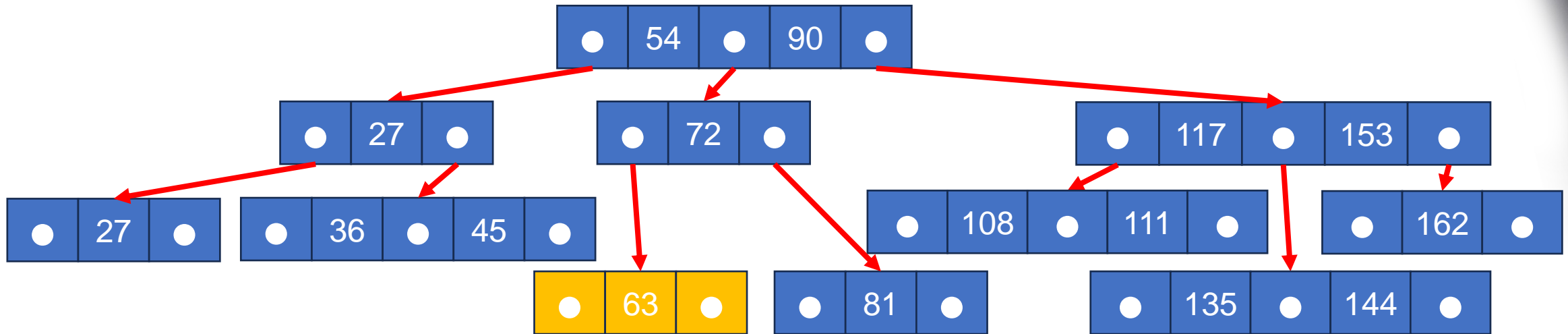


Поиск элемента в дереве 2-3

Шаг 1: Поскольку $54 < 63 < 90$, переходим к среднему дочернему элементу



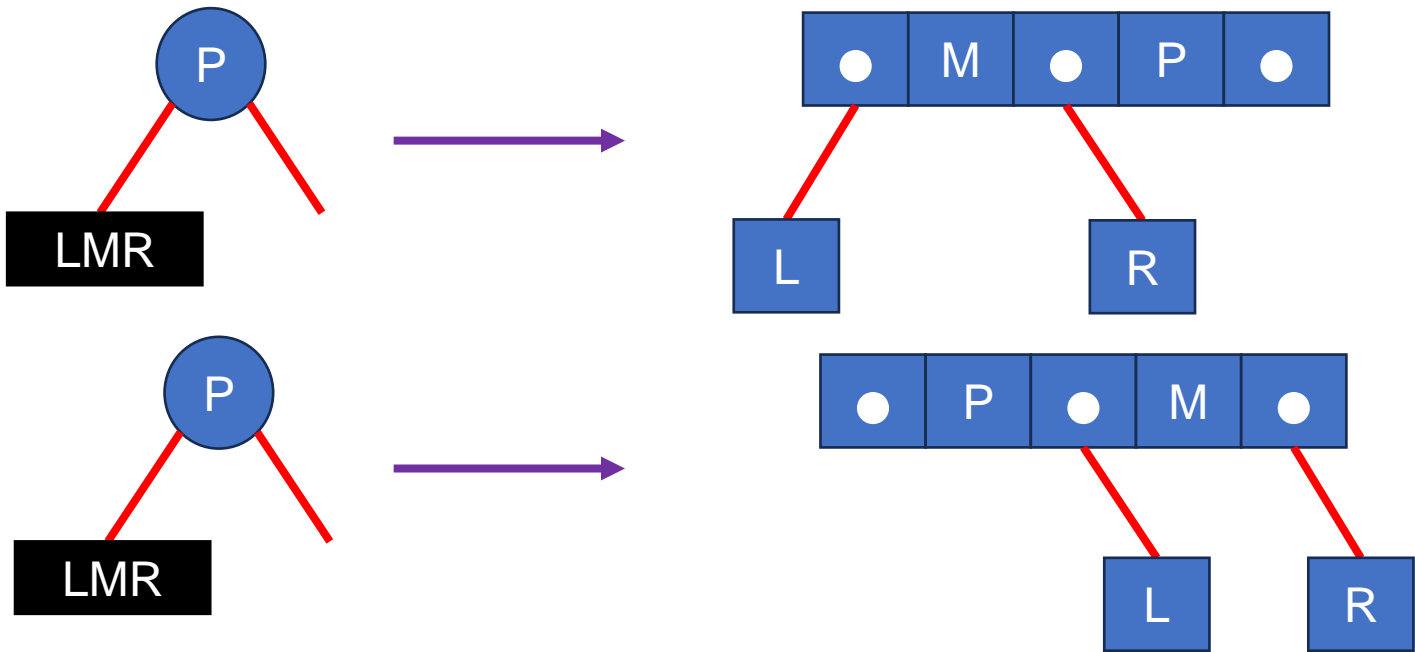
Шаг 2: Поскольку $63 < 72$, переходим к левому потомку.



Вставка нового элемента в дерево 2-3

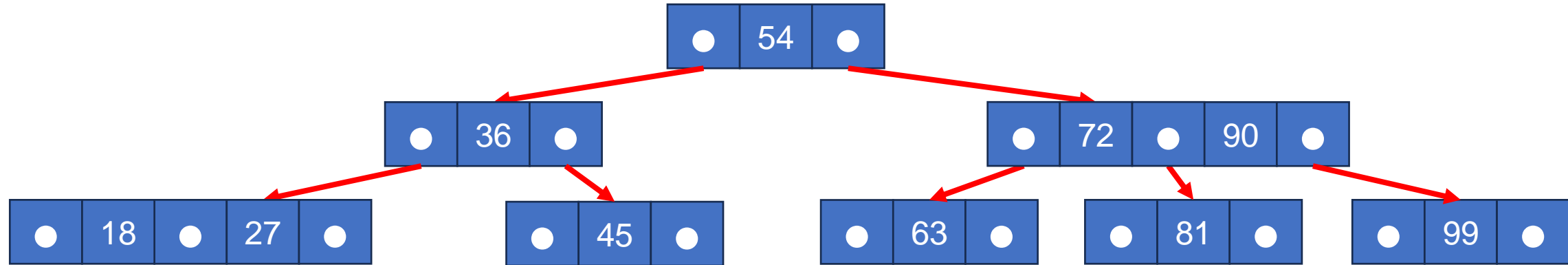
Чтобы вставить новое значение в дерево 2-3, соответствующая позиция значения находится в одном из листовых узлов. Если после вставки нового значения свойства дерева 2-3 не нарушаются, то вставка завершена. В противном случае, если какое-либо свойство нарушено, то нарушающий узел должен быть разделен.

Разделение узла Узел разделен, когда он имеет три значения данных и четыре дочерних элемента. Здесь P — родительский элемент, а L, M, R обозначают левого, среднего и правого дочерних элементов.



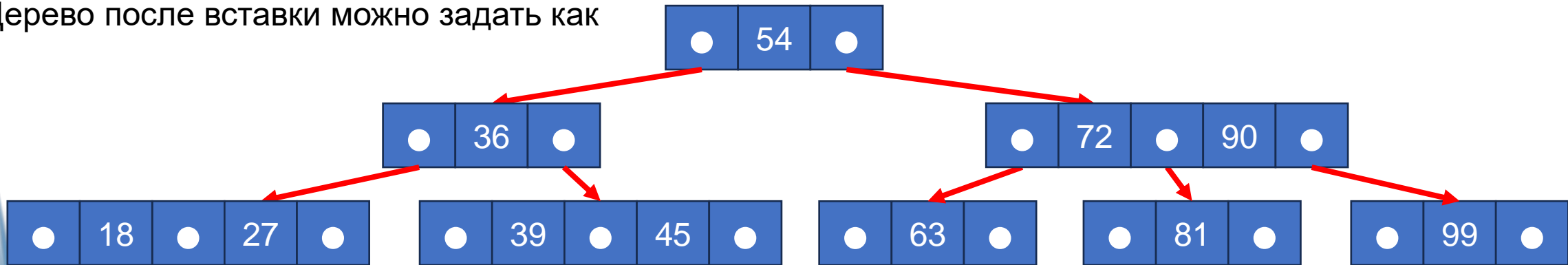
Вставка нового элемента в дерево 2-3

Рассмотрим приведенное ниже дерево 2-3 и вставим в него следующие значения данных: 39, 37, 42, 47.



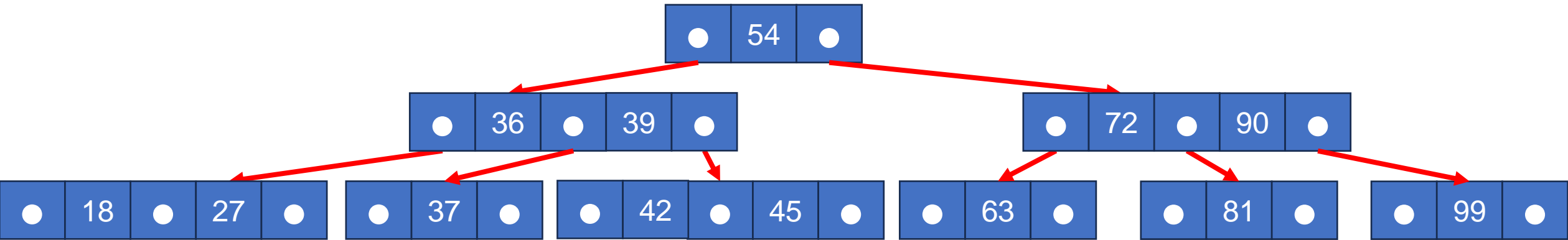
Шаг 1: Вставьте 39 в конечный узел

Дерево после вставки можно задать как

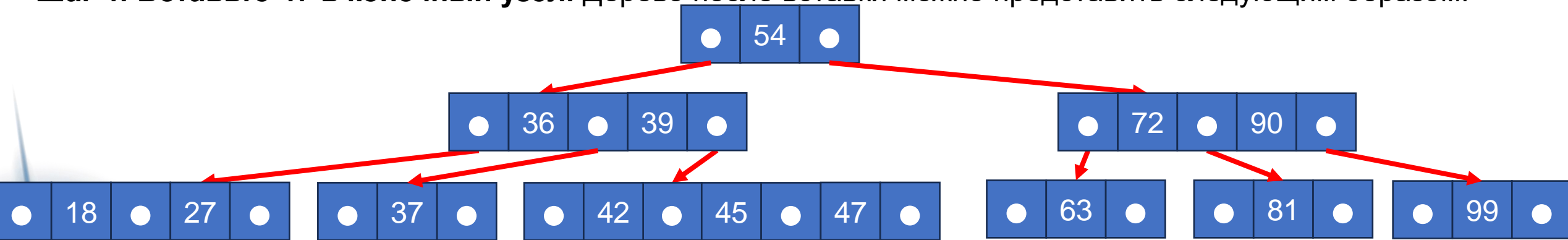


Вставка нового элемента в дерево 2-3

Шаг 3: Вставка 42 в конечный узел Дерево после вставки можно задать следующим образом.

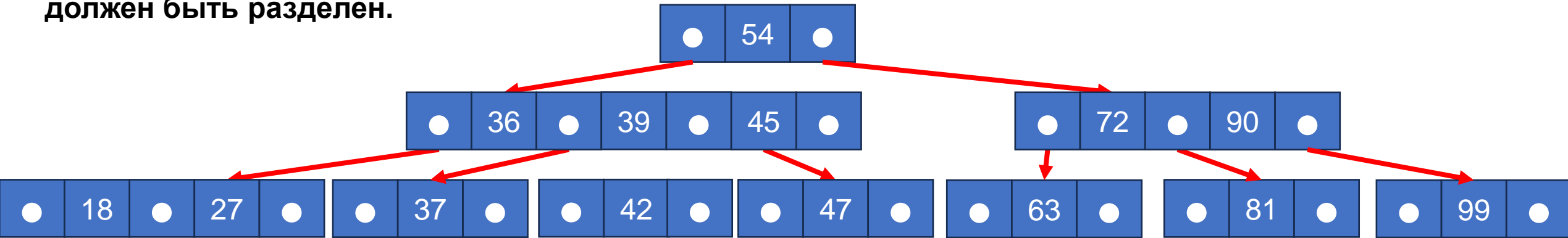


Шаг 4: Вставьте 47 в конечный узел. Дерево после вставки можно представить следующим образом.

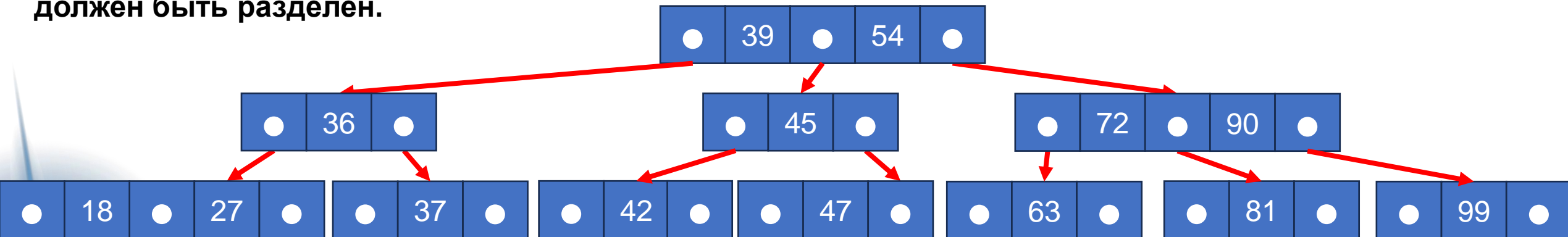


Вставка нового элемента в дерево 2-3

Конечный узел имеет три значения данных. Следовательно, узел нарушает свойства дерева и должен быть разделен.



Родительский узел имеет три значения данных. Следовательно, узел нарушает свойства дерева и должен быть разделен.



Удаление элемента из дерева 2-3

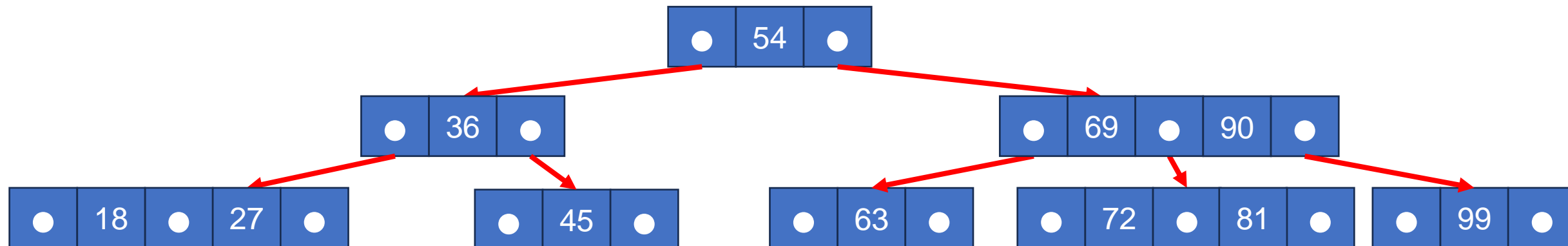
В процессе удаления указанное значение данных удаляется из дерева 2-3. Если удаление значения из узла нарушает свойство дерева, то есть если узел остается менее чем с одним значением данных, то два узла должны быть объединены вместе, чтобы сохранить общие свойства дерева 2-3.

При вставке новое значение должно быть добавлено в любой из конечных узлов, но при удалении не обязательно, чтобы значение было удалено из конечного узла. Значение может быть удалено из любого узла. Чтобы удалить значение x , оно заменяется его последовательным элементом, а затем удаляется. Если узел становится пустым после удаления значения, он затем объединяется с другим узлом, чтобы восстановить свойство дерева.



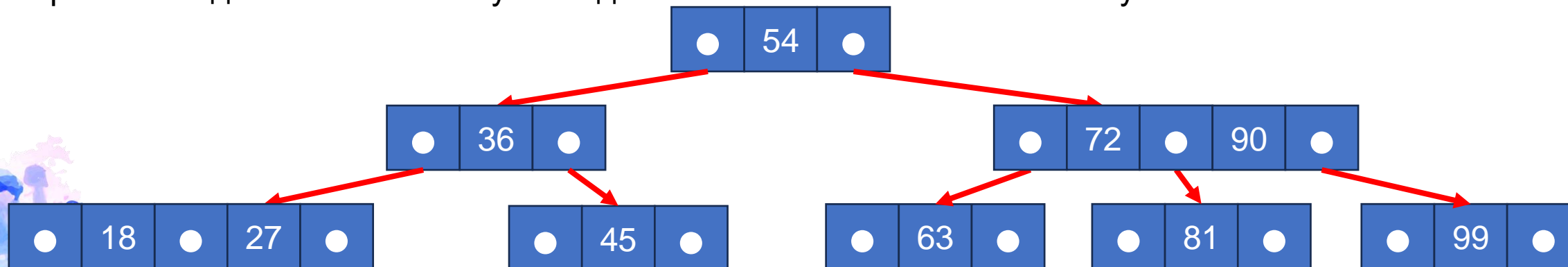
Удаление элемента из дерева 2-3

Рассмотрим дерево 2-3, приведенное ниже, и удалим из него следующие значения: 69, 72, 99, 81.



Чтобы удалить 69, поменяйте его местами с его последовательным элементом, то есть 72.

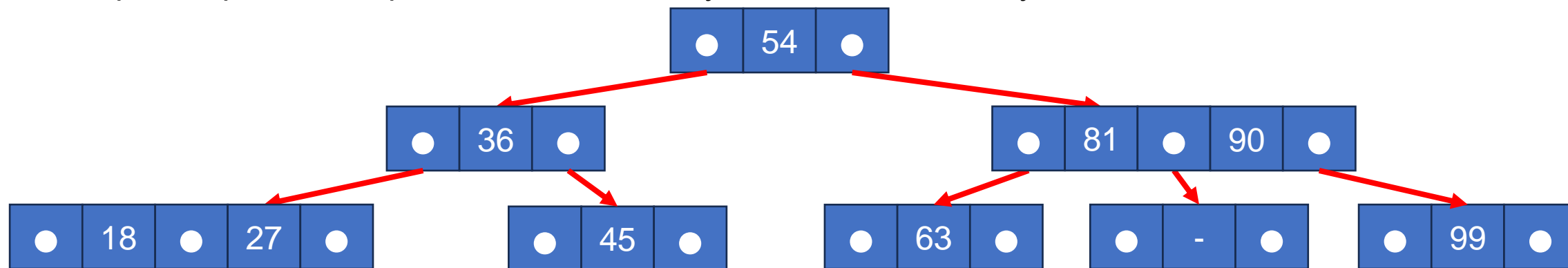
Теперь 69 находится в конечном узле. Удалите значение 69 из конечного узла.



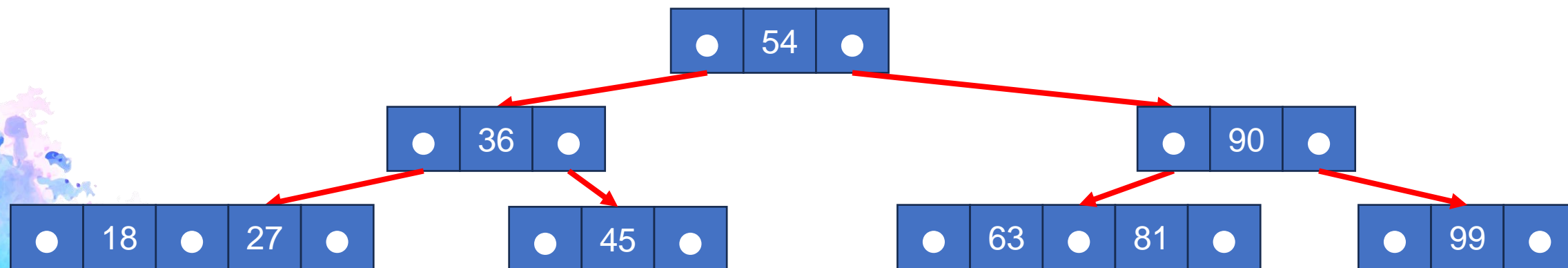
72 — это внутренний узел. Чтобы удалить это значение, поменяйте местами 72 с его последовательным преемником 81, так что 72 теперь станет листовым узлом. Удалите значение 72 из листового узла.

Удаление элемента из дерева 2-3

Рассмотрим дерево 2-3, приведенное ниже, и удалим из него следующие значения: 69, 72, 99, 81.



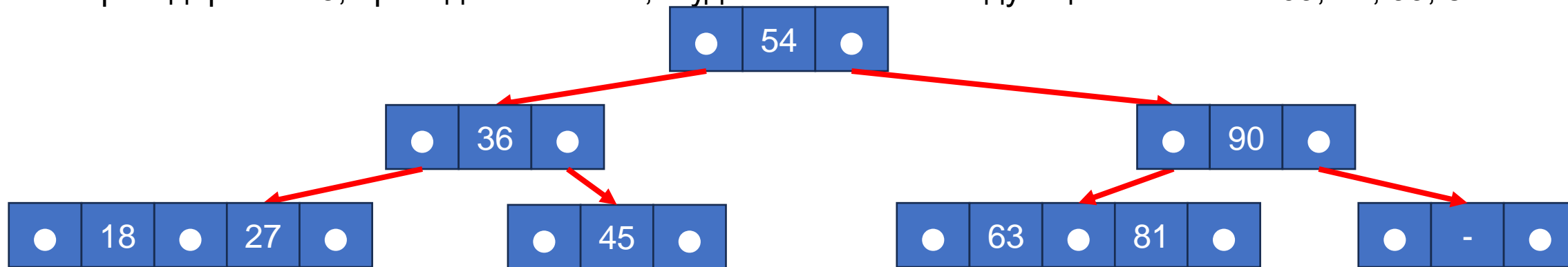
Теперь есть листовый узел, который имеет менее 1 значения данных, тем самым нарушая свойство дерева 2-3. Поэтому узел должен быть объединен. Чтобы объединить узел, потяните вниз наименьшее значение данных в родительском узле и объедините его с его левым братом.



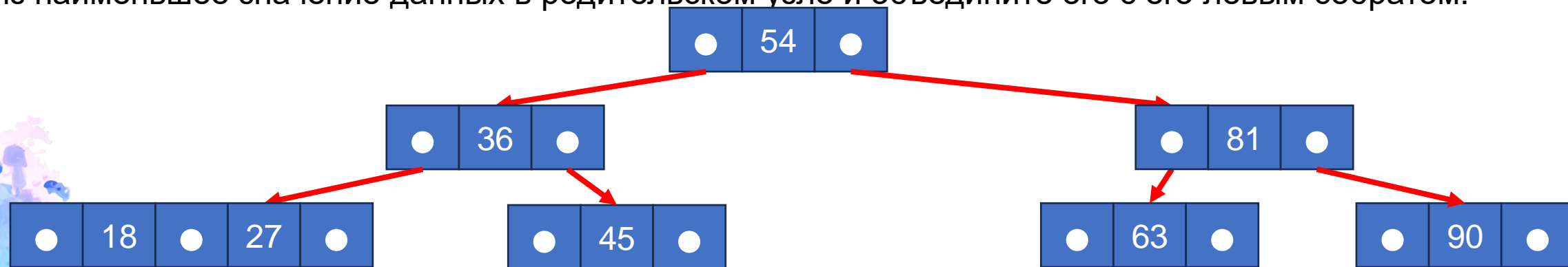
99 присутствует в листовом узле, поэтому значение данных можно легко удалить.

Удаление элемента из дерева 2-3

Рассмотрим дерево 2-3, приведенное ниже, и удалим из него следующие значения: 69, 72, 99, 81.



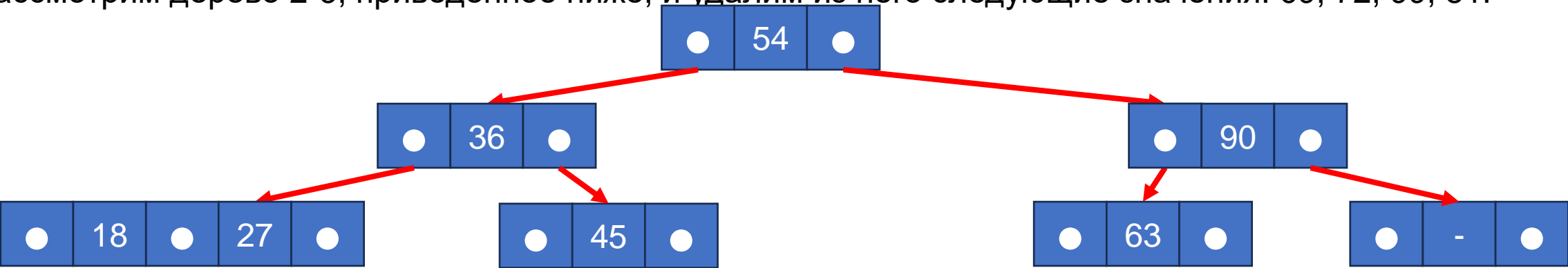
Теперь есть листовой узел, который имеет менее 1 значения данных, тем самым нарушая свойство дерева 2-3. Поэтому узел должен быть объединен. Чтобы объединить узел, потяните вниз наименьшее значение данных в родительском узле и объедините его с его левым собратом.



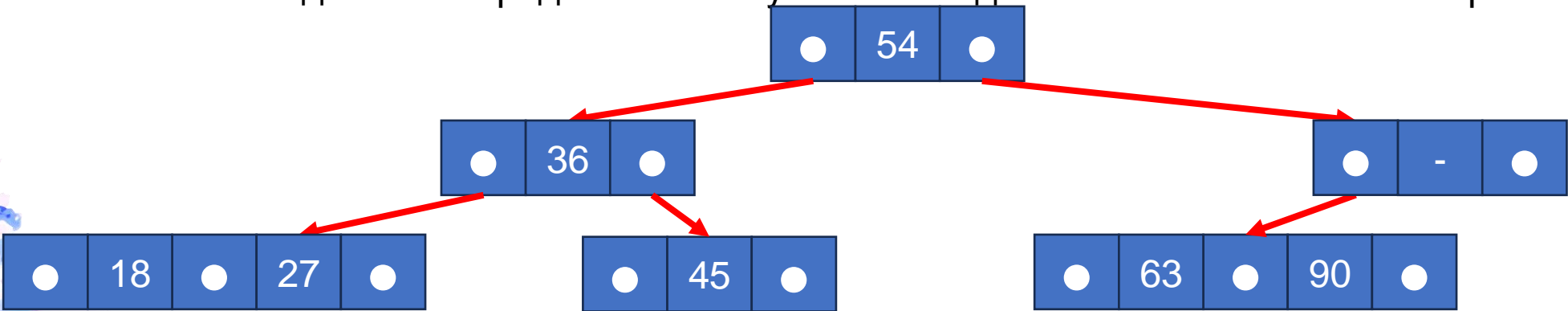
81 — внутренний узел. Чтобы удалить это значение, поменяйте местами 81 с его последовательным преемником 90, чтобы 81 теперь стал листовым узлом. Удалите значение 81 из листового узла.

Удаление элемента из дерева 2-3

Рассмотрим дерево 2-3, приведенное ниже, и удалим из него следующие значения: 69, 72, 99, 81.



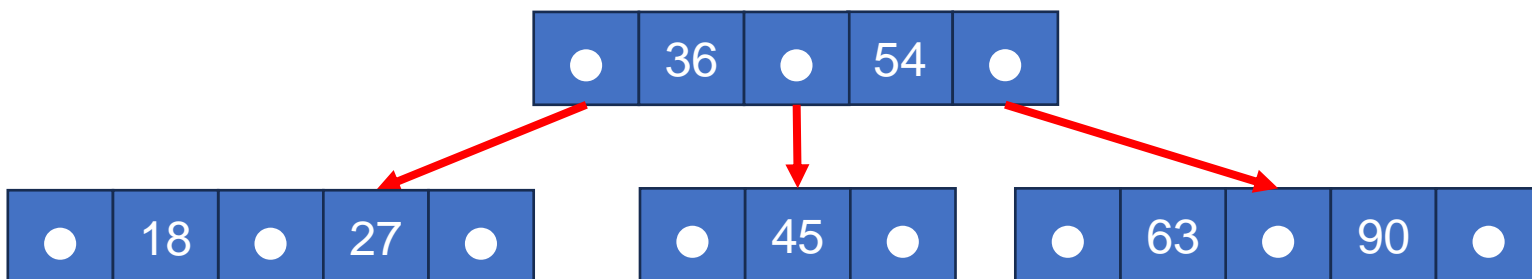
Теперь есть листовой узел, который имеет менее 1 значения данных, тем самым нарушая свойство дерева 2-3. Поэтому узел должен быть объединен. Чтобы объединить узел, извлеките наименьшее значение данных из родительского узла и объедините его с его левым собратом.



Внутренний узел не может быть пустым, поэтому теперь извлеките наименьшее значение данных из родительского узла и объедините пустой узел с его левым собратом.

Удаление элемента из дерева 2-3

Рассмотрим дерево 2-3, приведенное ниже, и удалим из него следующие значения: 69, 72, 99, 81.



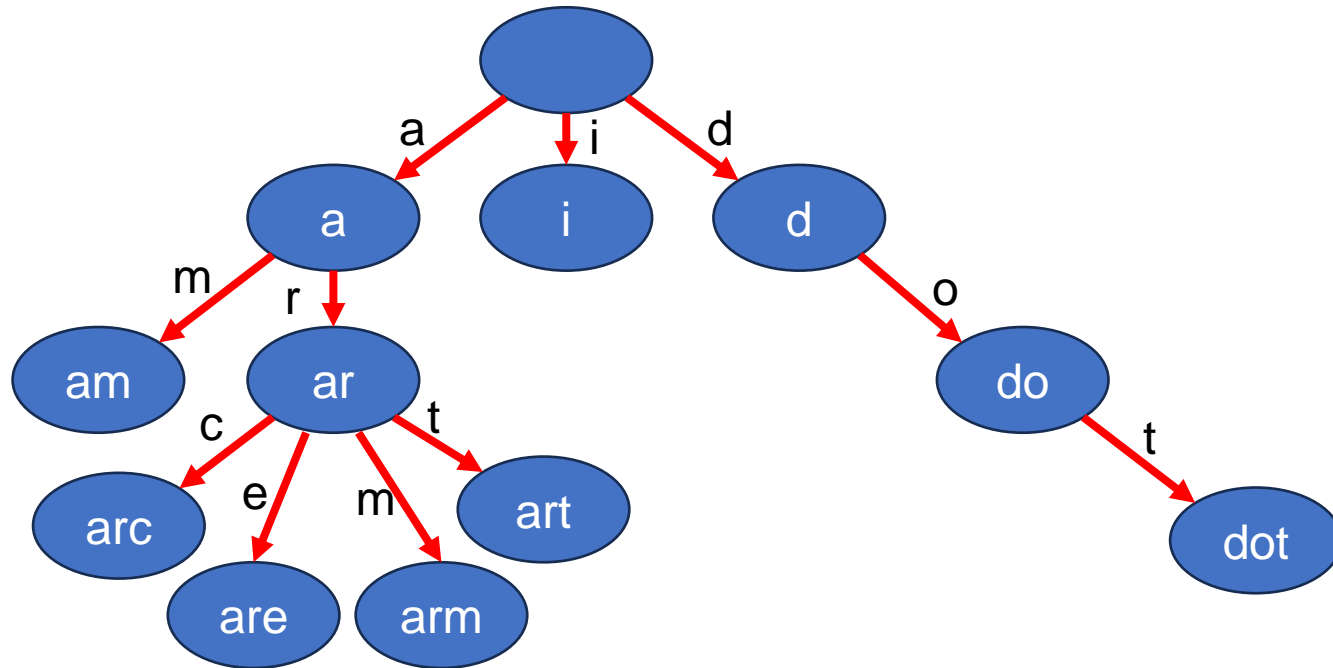
Удаление значений из заданного дерева 2-3



Префиксное дерево (trie)

Термин trie был взят из слова «извлечение». Trie — это упорядоченная древовидная структура данных, которая была введена в 1960-х годах Эдвардом Фредкиным. Trie хранит ключи, которые обычно являются строками. По сути, это k-арное позиционное дерево.

В отличие от двоичных деревьев поиска, узлы в trie не хранят ключи, связанные с ними. Вместо этого позиция узла в дереве представляет ключ, связанный с этим узлом. Все потомки узла имеют общий префикс строки, связанной с этим узлом, а корень связан с пустой строкой.



Префиксное дерево (trie)

Преимущества относительно дерева двоичного поиска

По сравнению с деревом двоичного поиска структура данных trie имеет следующие преимущества.

Более быстрый поиск

Поиск ключей выполняется быстрее, так как поиск ключа длиной m занимает $O(m)$ времени в худшем случае. С другой стороны, двоичное дерево поиска выполняет $O(\log n)$ сравнений ключей, где n — количество узлов в дереве. Поскольку время поиска зависит от высоты дерева, которая логарифмически зависит от количества ключей (если дерево сбалансировано), в худшем случае может потребоваться $O(m \log(n))$ времени. В дополнение к этому, m приближается к $\log(n)$ в худшем случае. Следовательно, структура данных trie обеспечивает более быстрый механизм поиска.

Меньше места

Trie занимает меньше места, особенно когда содержит большое количество коротких строк. Поскольку ключи не хранятся явно, а узлы совместно используются ключами с общими начальными подпоследовательностями, trie требует меньше места по сравнению с двоичным деревом поиска.

Самый длинный префикс-соответствие

Trie с сопоставлением самого длинного префикса облегчает сопоставление самого длинного префикса, что позволяет нам найти ключ, разделяющий самый длинный возможный префикс всех уникальных символов. Поскольку trie обеспечивает больше преимуществ, его можно рассматривать как хорошую замену двоичным деревьям поиска.

Префиксное дерево (trie)

Преимущества по сравнению с хэш-таблицей

- Trie также можно использовать для замены хэш-таблицы, поскольку он обеспечивает следующие преимущества:
- Поиск данных в trie в худшем случае выполняется быстрее, за время $O(m)$, по сравнению с несовершенной хэш-таблицей, в которой может быть множество коллизий ключей. Trie свободен от проблемы коллизий ключей.
- В отличие от хэш-таблицы, нет необходимости выбирать хэш-функцию или менять ее, когда в trie добавляется больше ключей.
- Trie может сортировать ключи, используя predetermined алфавитный порядок.

Недостатки

Недостатки наличия trie перечислены ниже:

- В некоторых случаях trie могут быть медленнее хэш-таблиц при поиске данных. Это справедливо в случаях, когда данные напрямую доступны на жестком диске или каком-либо другом вторичном устройстве хранения данных, которое имеет высокое время случайного доступа по сравнению с основной памятью.
- Все ключевые значения не могут быть легко представлены в виде строк. Например, одно и то же число с плавающей точкой может быть представлено в виде строки несколькими способами (1 эквивалентно 1,0, 1,00, +1,0 и т. д.).

Префиксное дерево (trie)

Приложения

Tries обычно используются для хранения словаря (например, на мобильном телефоне). Эти приложения используют возможность trie быстро искать, вставлять и удалять записи. Tries также используются для реализации алгоритмов приблизительного сопоставления, включая те, которые используются в программном обеспечении для проверки орфографии.



Фибоначчиевы пирамиды

Пирамиды Фибоначчи служат двум целям. Во-первых, они поддерживают набор операций, составляющих то, что известно под названием "объединяемые, или сливаемые, пирамиды" (mergeable heap). Во-вторых, некоторые операции в фибоначчиевых пирамидах выполняются за константное амортизированное время, что делает эту структуру данных хорошо подходящей для приложений, в которых часто применяются данные операции.



Объединяемые пирамиды

Объединяемой пирамидой является любая структура данных, поддерживающая следующие операции, в которых каждый элемент имеет ключ key .

Make-Heap() создает и возвращает новую пирамиду, не содержащую элементов.

INSERT(H, x) вставляет в пирамиду H элемент x , ключ key которого к этому моменту заполнен.

Minimum(H) возвращает указатель на элемент пирамиды H , ключ которого минимален.

Extract-Min(H) удаляет из пирамиды H элемент, ключ которого минимален, и возвращает указатель на этот элемент.

Union(H_1, H_2) создает и возвращает новую пирамиду, которая содержит все элементы пирамид H_1 и H_2 . Пирамиды H_1 и H_2 этой операцией "уничтожаются":

В дополнение к перечисленным выше операциям с объединяемыми пирамидами пирамиды Фибоначчи поддерживают еще две операции.

Decrease-Key(H, x, k) назначает элементу x в пирамиде H новое значение ключа k , которое не больше его текущего значения.

Delete(H, x) удаляет элемент x из пирамиды H .



Скорость операций

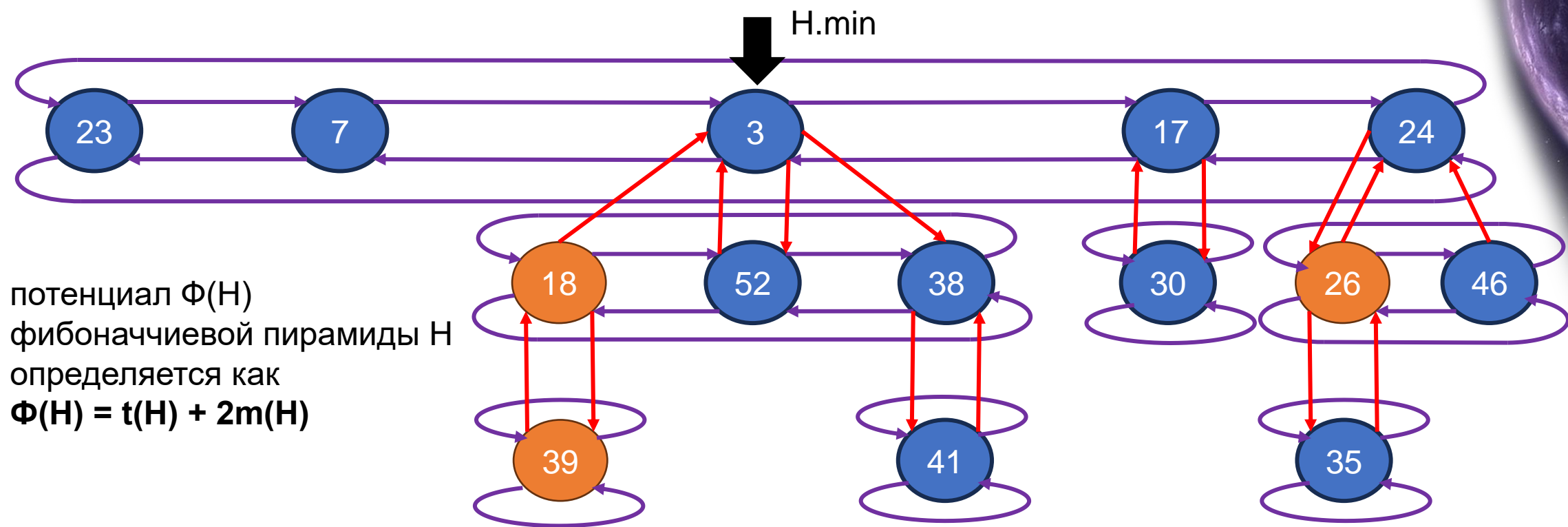
Процедура	Бинарная пирамида (наихудший случай)	Фибоначчиева пирамида (амортизированная))
Make-Heap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\lg n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
Extract-Min	$\Theta(\lg n)$	$O(\lg n)$
Union	$\Theta(n)$	$\Theta(1)$
Decrease-Key	$\Theta(\lg n)$	$\Theta(1)$
Delete	$\Theta(\lg n)$	$O(\lg n)$

Время выполнения операций в двух реализациях объединяемых пирамид. В момент выполнения операции количество элементов пирамиды (или пирамид) равно n .



Фибоначчиевы пирамиды

Фибоначчиева пирамида (Fibonacci heap) представляет собой набор корневых деревьев, упорядоченных в соответствии со свойством неубывающей пирамиды, т.е. каждое дерево подчиняется этому свойству: ключ узла не меньше ключа его родителя.



потенциал $\Phi(H)$
фибоначчиевой пирамиды H
определяется как
 $\Phi(H) = t(H) + 2m(H)$

Фибоначчиева пирамида, состоящая из пяти деревьев, упорядоченных в соответствии со свойством неубывающей пирамиды, и четырнадцати узлов. Помеченные узлы показаны оранжевым цветом. Потенциал данной конкретной фибоначчиевой пирамиды равен $5 + 2 \cdot 3 = 11$. (б). Указатели p (стрелки вверх), $child$ (стрелки вниз) и $left$ и $right$ (горизонтальные стрелки).

Деревья ван Эмде Боаса

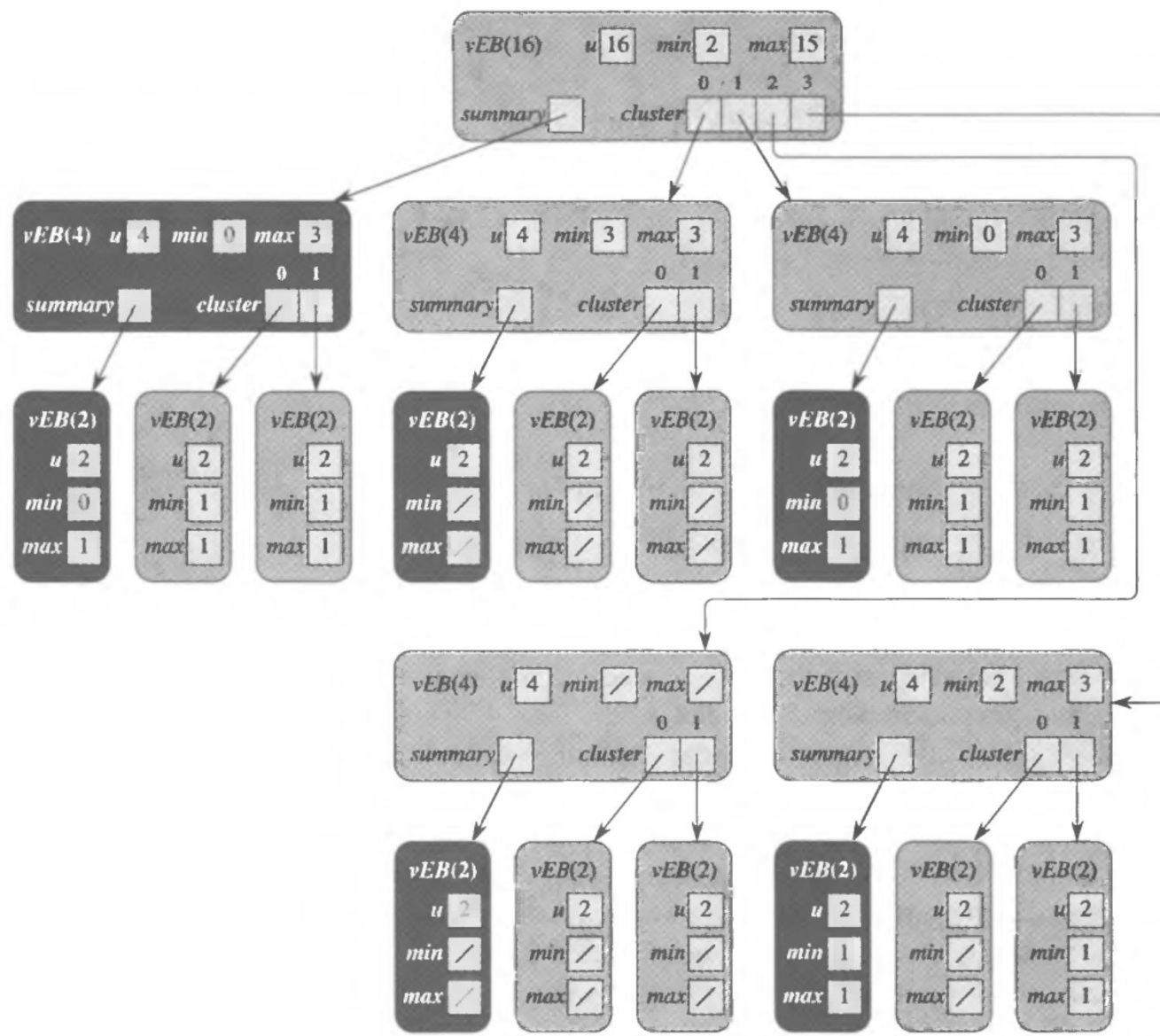
Мы познакомились со структурами данных, поддерживающими операции над очередями с приоритетами - бинарными пирамидами, красно-черными деревьями и фибоначчиевыми пирамидами. В каждой из этих структур данных как минимум одна важная операция выполняется за время $O(\lg n)$ - либо в худшем случае, либо в случае амортизированного времени. Фактически, поскольку каждая из указанных структур данных основывается на сравнении ключей, **нижняя граница времени сортировки $\Omega(n \lg n)$** говорит о том, что по меньшей мере одна из операций должна выполняться за время $\Omega(\lg n)$.

Почему? Если бы мы были способны выполнить и операцию INSERT, и операцию EXTRACT-MIN за время $o(\lg n)$, мы могли бы отсортировать n ключей за время $o(n \lg n)$, сначала выполнив операций INSERT, а затем операций EXTRACT-MIN.

Иногда можно воспользоваться дополнительной информацией о ключах, чтобы отсортировать их за меньшее время. В частности, сортировка подсчетом позволяет отсортировать n ключей, каждый из которых является целым числом в диапазоне от 0 до k , за время $\Theta(n + k)$, которое представляет собой $\Theta(n)$ при $k = O(n)$.

Поскольку нижняя граница сортировки $N(n \lg n)$ может быть превзойдена, когда ключи представляют собой целые числа из ограниченного диапазона, закономерно возникает вопрос - нельзя ли при том же условии выполнять каждую из операций очереди с приоритетами за время $o(\lg n)$? Деревья ван Эмде Боаса поддерживают операции очередей с приоритетами и несколько других операций, со временем работы каждой из них, в наихудшем случае равным $O(\lg \lg n)$. Условием для этого является то, что ключи должны быть целыми числами в диапазоне от 0 до $n - 1$ и не должно быть двух одинаковых ключей.

Деревья ван Эмде Боаса - спойлер

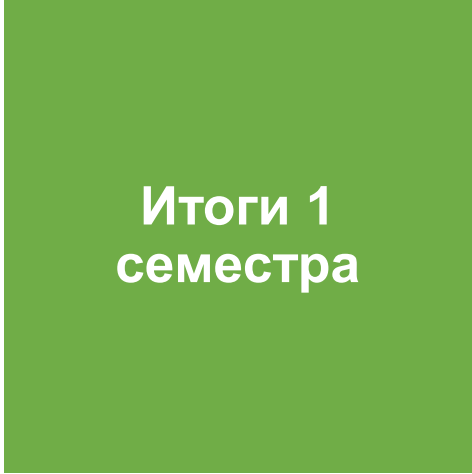


План презентации



Мультидеревья

87 минут



**Итоги 1
семестра**

3 минуты