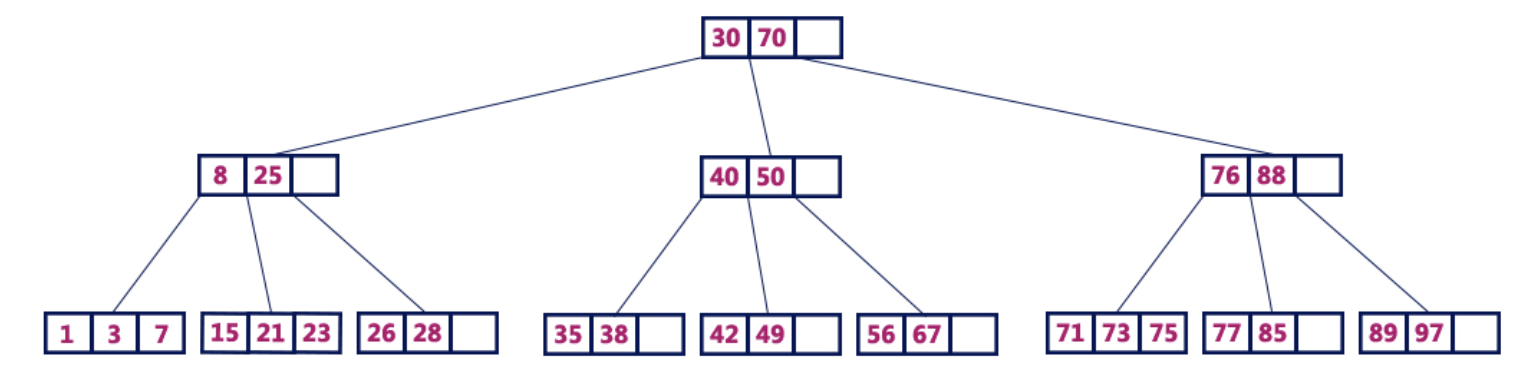
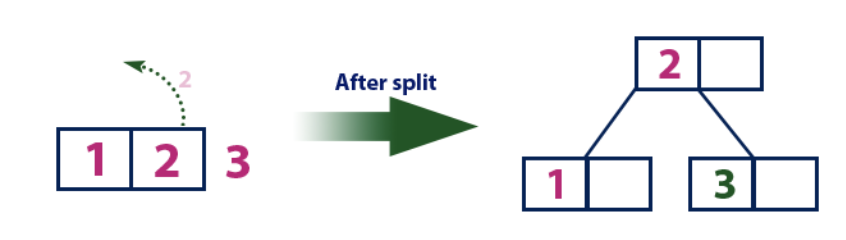
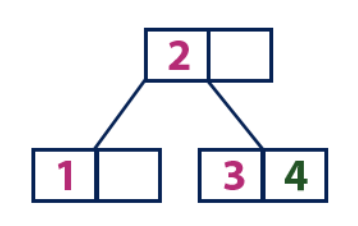
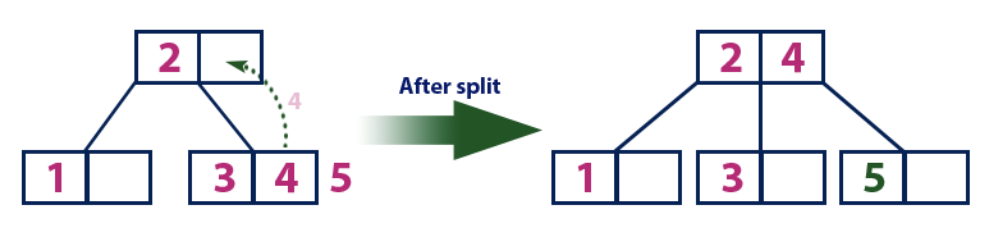
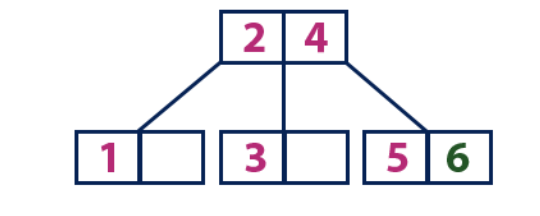
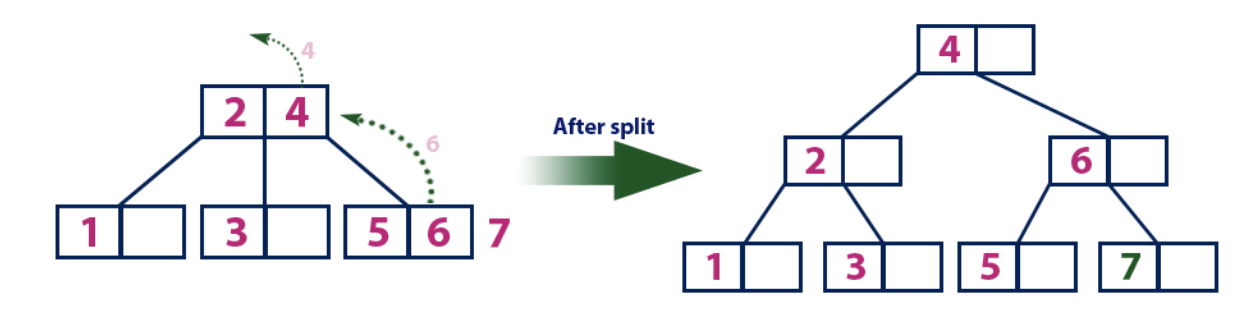
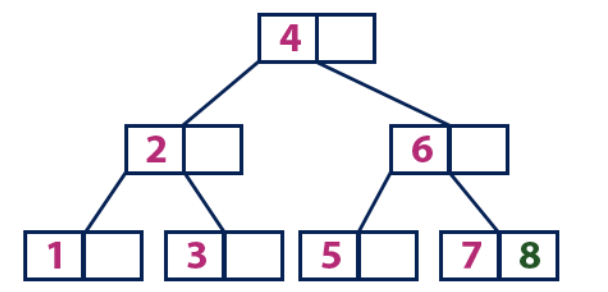
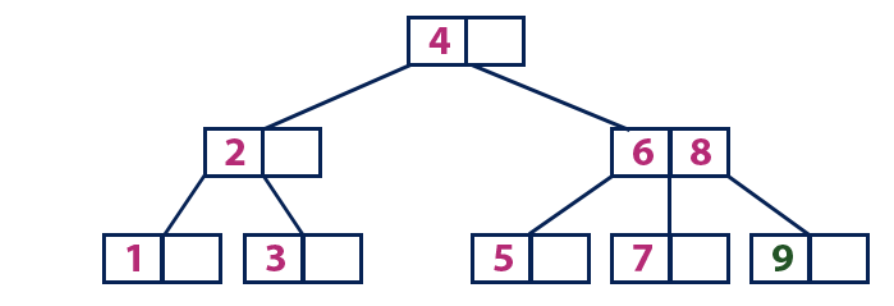
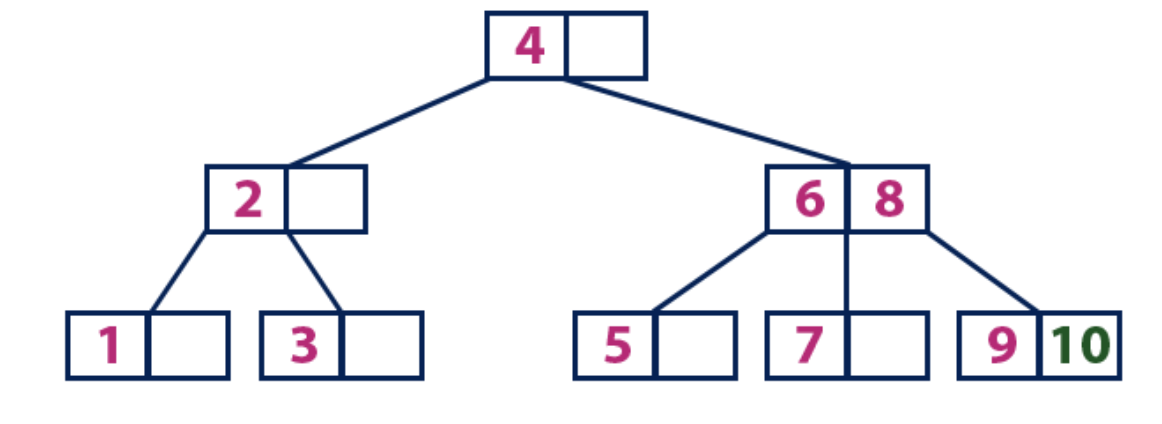
В деревьях поиска, таких как двоичное дерево поиска, AVL дерево, красно-чёрное дерево и т.п. каждый узел содержит только одно значение (ключ) и максимум двое потомков. Однако есть особый тип дерева поиска, который называется B-дерево (произносится как Би-дерево). В нем узел содержит более одного значения (ключа) и более двух потомков. B-дерево было разработано в 1972 году Байером и МакКрейтом и называлось *Сбалансированное по высоте дерево поиска порядка m (Height Balanced m-way Search Tree)*. Свое современное название B-дерево получило позже.  
  
B-дерево можно определить следующим образом:  
**B-дерево – это сбалансированное дерево поиска, в котором каждый узел содержит множество ключей и имеет более двух потомков.**  
  
Здесь количество ключей в узле и количество его потомков зависит от порядка B-дерева. Каждое B-дерево имеет порядок.  
  
B-дерево порядка **m** обладает следующими свойствами:  
  
*Свойство 1:* Глубина всех листьев одинакова.  
*Свойство 2:* Все узлы, кроме корня должны иметь как минимум ***(m/2) – 1*** ключей и максимум ***m-1*** ключей.  
*Свойство 3:* Все узлы без листьев, кроме корня (т.е. все внутренние узлы), должны иметь минимум ***m/2*** потомков.  
*Свойство 4:* Если корень – это узел не содержащий листьев, он должен иметь минимум **2** потомка.  
*Свойство 5:*Узел без листьев с **n-1** ключами должен иметь **n** потомков.  
*Свойство 6:* Все ключи в узле должны располагаться в порядке возрастания их значений.  
  
Например, B-дерево 4 порядка содержит максимум 3 значения ключа и максимум 4 потомка для каждого узла.  
  
  
*B-дерево 4 порядка*  
  
**Операции над B-деревом**  
  
Над B-деревом можно проводить следующие операции:

1. Поиск
2. Вставка
3. Удаление

**Поиск по B-дереву**  
  
Поиск по B-дереву аналогичен поиску по двоичному дереву поиска. В двоичном дереве поиска поиск начинается с корня и каждый раз принимается двустороннее решение (пойти по левому поддереву или по правому). В В-дереве поиск также начинается с корневого узла, но на каждом шаге принимается n-стороннее решение, где n – это общее количество потомков рассматриваемого узла. В В-дереве сложность поиска составляет **O(log n)**. Поиск происходит следующим образом:  
  
*Шаг 1:* Считать элемент для поиска.  
*Шаг 2:* Сравнить искомый элемент с первым значением ключа в корневом узле дерева.  
*Шаг 3:* Если они совпадают, вывести: «Искомый узел найден!» и завершить поиск.  
*Шаг 4:* Если они не совпадают, проверить больше или меньше значение элемента, чем текущее значение ключа.  
*Шаг 5:* Если искомый элемент меньше, продолжить поиск по левому поддереву.  
*Шаг 6:* Если искомый элемент больше, сравнить элемент со следующим значением ключа в узле и повторять Шаги 3, 4, 5 и 6 пока не будет найдено совпадение или пока искомый элемент не будет сравнен с последним значением ключа в узле-листе.  
*Шаг 7:* Если последнее значение ключа в узле-листе не совпало с искомым, вывести «Элемент не найден!» и завершить поиск.  
  
**Операция вставки в B-дерево**  
  
В В-дереве новый элемент может быть добавлен только в узел-лист. Это значит, что новая пара ключ-значение всегда добавляется только к узлу-листу. Вставка происходит следующим образом:  
  
*Шаг 1:* Проверить пустое ли дерево.  
*Шаг 2:* Если дерево пустое, создать новый узел с новым значением ключа и его принять за корневой узел.  
*Шаг 3:* Если дерево не пустое, найти подходящий узел-лист, к которому будет добавлено новое значение, используя логику дерева двоичного поиска.  
*Шаг 4:* Если в текущем узле-листе есть незанятая ячейка, добавить новый ключ-значение к текущему узлу-листу, следуя возрастающему порядку значений ключей внутри узла.  
*Шаг 5:* Если текущий узел полон и не имеет свободных ячеек, разделите узел-лист, отправив среднее значение родительскому узлу. Повторяйте шаг, пока отправляемое значение не будет зафиксировано в узле.  
*Шаг 6:* Если разделение происходит с корнем дерева, тогда среднее значение становится новым корнем дерева и высота дерева увеличивается на единицу.  
  
**Пример:**  
  
Давайте создадим B-дерево порядка 3, добавляя в него числа от 1 до 10.  
  
*Insert(1):*  
Поскольку «1» — это первый элемент дерева – он вставляется в новый узел и этот узел становится корнем дерева.  
  
  
  
*Insert(2):*  
Элемент «2» добавляется к существующему узлу-листу. Сейчас у нас всего один узел, следовательно он является и корнем и листом одновременно. В этом листе имеется пустая ячейка. Тогда «2» встает в эту пустую ячейку.  
  
  
  
*Insert(3):*  
Элемент «3» добавляется к существующему узлу-листу. Сейчас у нас только один узел, который одновременно является и корнем и листом. У этого листа нет пустой ячейки. Поэтому мы разделяем этот узел, отправляя среднее значение (2) в родительский узел. Однако у текущего узла родительского узла нет. Поэтому среднее значение становится корневым узлом дерева.  
  
  
  
*Insert(4):*  
Элемент «4» больше корневого узла со значением «2», при этом корневой узел не является листом. Поэтому мы двигаемся по правому поддереву от «2». Мы приходим к узлу-листу со значением «3», у которого имеется пустая ячейка. Таким образом, мы можем вставить элемент «4» в эту пустую ячейку.  
  
  
  
*Insert(5):*  
Элемент «5» больше корневого узла со значением «2», при этом корневой узел не является листом. Поэтому мы двигаемся по правому поддереву от «2». Мы приходим к узлу-листу и обнаруживаем, что он уже полон и не имеет пустых ячеек. Тогда мы делим этот узел, отправляя среднее значение (4) в родительский узел (2). В родительском узле есть для него пустая ячейка, поэтому значение «4» добавляется к узлу, в котором уже есть значение «2», а новый элемент «5» добавляется в качестве нового листа.  
  
  
  
*Insert(6):*  
Элемент «6» больше, чем элементы корня «2» и «4», который не является листом. Мы двигаемся по правому поддереву от элемента «4». Мы достигаем листа со значением «5», у которого есть пустая ячейка, поэтому элемент «6» помещаем как раз в нее.  
  
  
  
*Insert(7):*  
Элемент «7» больше, чем элементы корня «2» и «4», который не является листом. Мы двигаемся по правому поддереву от элемента «4». Мы достигаем узла-листа и видим, что он полон. Мы делим этот узел, отправляя среднее значение «6» вверх к родительскому узлу с элементами «2» и «4». Однако родительский узел тоже полон, поэтому мы делим узел с элементами «2» и «4», отправляя значение «4» родительскому узлу. Только вот этого узла еще нет. В таком случае узел с элементом «4» становится новым корнем дерева.  
  
  
  
*Insert(8):*  
Элемент «8» больше корневого узла со значением «4», при этом корневой узел не является листом. Мы двигаемся по правому поддереву от элемента «4» и приходим к узлу со значением «6». «8» больше «6» и узел с элементом «6» не является листом, поэтому двигаемся по правому поддереву от «6». Мы достигаем узла-листа с «7», у которого есть пустая ячейка, поэтому в нее мы помещаем «8».  
  
  
  
*Insert(9):*  
Элемент «9» больше корневого узла со значением «4», при этом корневой узел не является листом. Мы двигаемся по правому поддереву от элемента «4» и приходим к узлу со значением «6». «9» больше «6» и узел с элементом «6» не является листом, поэтому двигаемся по правому поддереву от «6». Мы достигаем узла-листа со значениями «7» и «8». Он полон. Мы делим этот узел, отправляя среднее значение (8) родительскому узлу. Родительский узел «6» имеет пустую ячейку, поэтому мы помещаем «8» в нее. При этом новый элемент «9» добавляется в узел-лист.  
  
  
  
Insert(10):  
Элемент «10» больше корневого узла со значением «4», при этом корневой узел не является листом. Мы двигаемся по правому поддереву от элемента «4» и приходим к узлу со значениями «6» и «8». «10» больше «6» и «8» и узел с этими элементами не является листом, поэтому двигаемся по правому поддереву от «8». Мы достигаем узла-листа со значением «9». У него есть пустая ячейка, поэтому туда мы помещаем «10».  
  
  
  
Предлагаем вам самостоятельно на практике понять, как устроены В-деревья, воспользовавшись [этой визуализацией](https://www.cs.usfca.edu/~galles/visualization/BTree.html).

## Что такое B-дерево?

**B Tree** – это самобалансирующаяся структура данных, основанная на определенном наборе правил для поиска, вставки и удаления данных более быстрым и эффективным способом памяти. Чтобы достичь этого, для создания B-дерева следуйте следующим правилам.

Реклама

B-дерево – это особый вид дерева в структуре данных. В 1972 году этот метод был впервые представлен McCreight, и Bayer назвал его m-way Search Tree с сбалансированной высотой. Это помогает сохранить отсортированные данные и позволяет выполнять различные операции, такие как вставка, поиск и удаление, за меньшее время.

**Правила для B-Tree**

Здесь важны правила создания B\_Tree

* Все листья будут созданы на одном уровне.
* B-Tree определяется числом степеней, которые также называют «порядком» (указанным внешним субъектом, например, программистом), называемым

m

и далее. Значение

m

зависит от размера блока на диске, на котором в первую очередь располагаются данные.

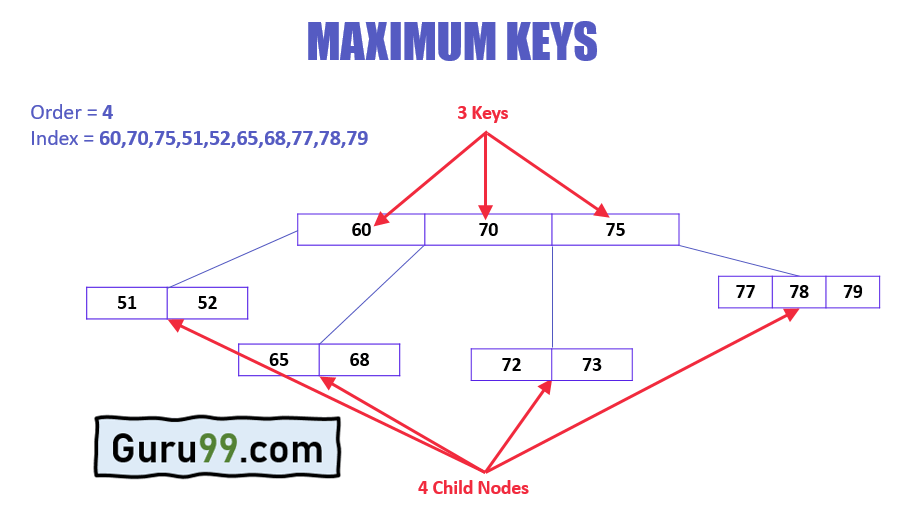
* Левое поддерево узла будет иметь меньшие значения, чем правая сторона поддерева. Это означает, что узлы также сортируются в порядке возрастания слева направо.
* Максимальное количество дочерних узлов, которые может содержать корневой узел, а также его дочерние узлы, рассчитывается по следующей формуле:

m – 1

Например:

m = 4

max keys: 4 – 1 = 3

[](https://coderlessons.com/wp-content/uploads/images/gur/d4f5cb103f8556f98c2509effa595012.png)

* Каждый узел, кроме root, должен содержать минимальные ключи

[m/2]-1

Например:

m = 4

min keys: 4/2-1 = 1

* Максимальное количество дочерних узлов, которое может иметь узел, равно его степени, которая равна

m

* Минимальное число дочерних элементов, которое может иметь узел, составляет половину порядка, то есть m / 2 (берется значение потолка).
* Все ключи в узле отсортированы в порядке возрастания.

**Зачем использовать B-Tree**

Вот причины использования B-Tree

* Уменьшает количество операций чтения на диске
* B Деревья могут быть легко оптимизированы для настройки их размера (то есть количества дочерних узлов) в соответствии с размером диска.
* Это специально разработанный метод для обработки большого количества данных.
* Это полезный алгоритм для баз данных и файловых систем.
* Хороший выбор для чтения и записи больших блоков данных.

**История B Tree**

* Данные хранятся на диске в блоках, эти данные, когда заносятся в основную память (или ОЗУ), называются структурой данных.
* В случае больших данных поиск одной записи на диске требует чтения всего диска; это увеличивает время и потребление основной памяти из-за высокой частоты доступа к диску и размера данных.
* Чтобы преодолеть это, создаются индексные таблицы, которые сохраняют ссылки на записи на основе блоков, в которых они находятся. Это значительно сокращает время и потребление памяти.
* Поскольку у нас огромные данные, мы можем создавать многоуровневые индексные таблицы.
* Многоуровневый индекс может быть разработан с использованием B Tree для сохранения сортировки данных в режиме самобалансировки.

**Операция поиска**

Операция поиска является самой простой операцией на B Tree.

Применяется следующий алгоритм:

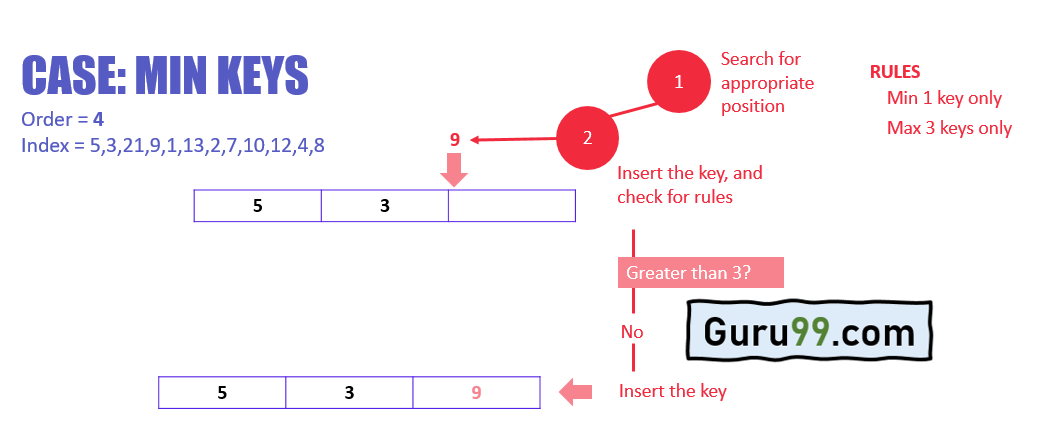
* Пусть ключ (значение) будет искать “k”.
* Начните поиск с корня и рекурсивно пройдите вниз.
* Если k меньше корневого значения, ищите левое поддерево, если k больше корневого значения, ищите правое поддерево.
* Если узел имеет найденное k, просто верните узел.
* Если k не найдено в узле, перейдите к дочернему элементу с большим ключом.
* Если k не найдено в дереве, мы возвращаем NULL.

**Операция вставки**

Поскольку B Tree является самобалансирующимся деревом, вы не можете принудительно вставить ключ в любой узел.

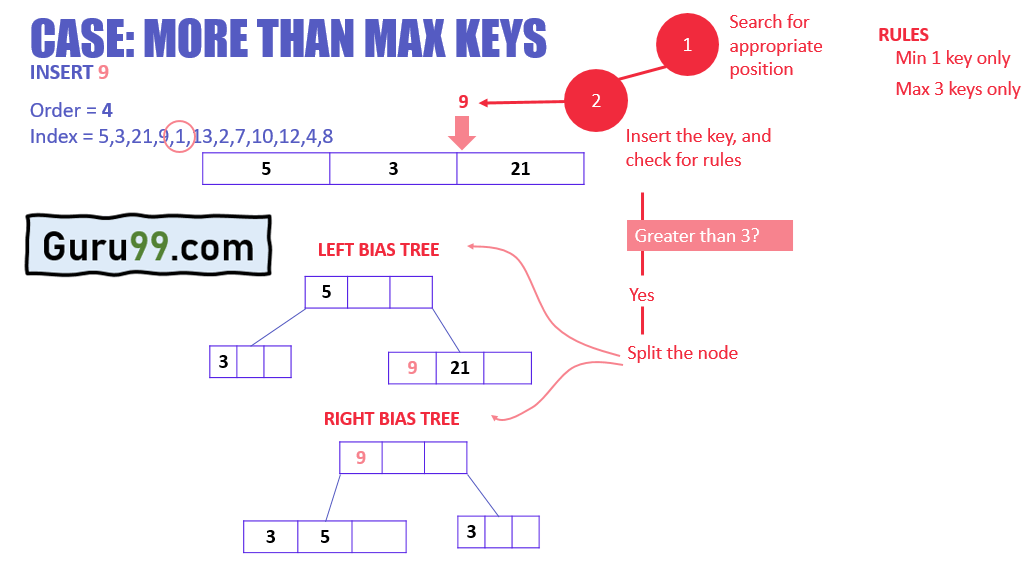
Применяется следующий алгоритм:

* Запустите операцию поиска и найдите подходящее место для вставки.
* Вставьте новый ключ в нужное место, но если у узла уже есть максимальное количество ключей:
* Узел вместе с новым вставленным ключом будет отделен от среднего элемента.
* Средний элемент станет родительским для двух других дочерних узлов.
* Узлы должны переставлять ключи в порядке возрастания.

[](https://coderlessons.com/wp-content/uploads/images/gur/175e08850eba55669197895418ec6e08.png)

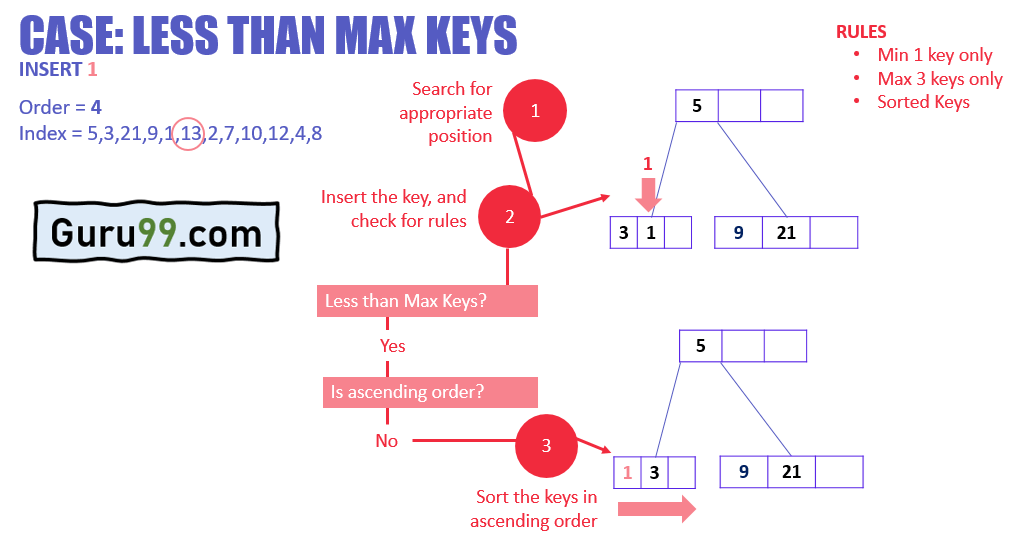
В приведенном выше примере:

* Поиск соответствующей позиции в узле для ключа
* Вставьте ключ в целевой узел и проверьте правила
* После вставки имеет ли узел более чем равный минимальному количеству ключей, которое равно 1? В этом случае, да, это так. Проверьте следующее правило.
* После вставки, у узла есть больше чем максимальное количество ключей, которое является 3? В этом случае нет, это не так. Это означает, что B-дерево не нарушает никаких правил, и вставка завершена.

[](https://coderlessons.com/wp-content/uploads/images/gur/64ac140398a55f35f825c9ea30425423.png)

В приведенном выше примере:

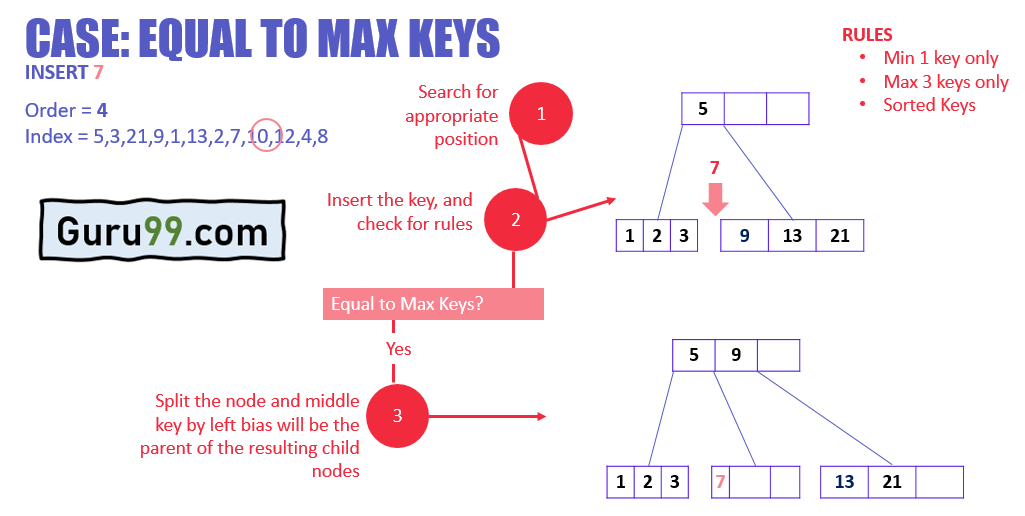
* Узел достиг максимального количества ключей
* Узел разделится, и средний ключ станет корневым узлом остальных двух узлов.
* В случае четного количества клавиш средний узел будет выбран смещением влево или вправо.

[](https://coderlessons.com/wp-content/uploads/images/gur/f1c45cb13f8a8e04caf0613e170ebe43.png)

В приведенном выше примере:

* В узле меньше макс ключей
* 1 вставлен рядом с 3, но правило в порядке возрастания нарушено
* Чтобы это исправить, ключи отсортированы

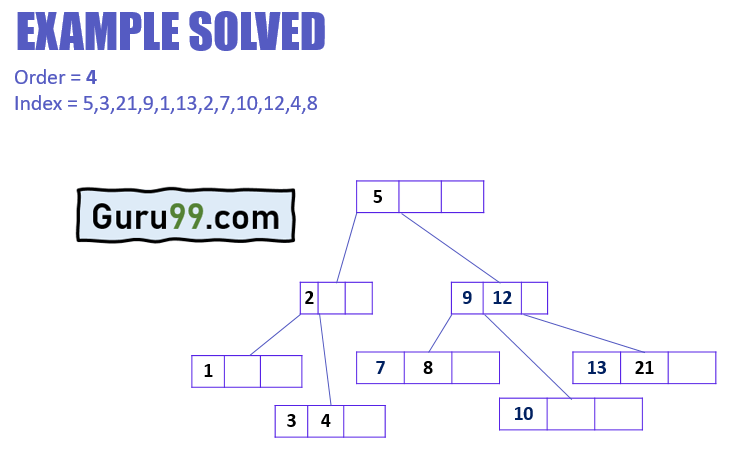
Точно так же 13 и 2 могут быть легко вставлены в узел, так как они удовлетворяют правилу «меньше ключей» для узлов.

[](https://coderlessons.com/wp-content/uploads/images/gur/bf571e6219e7540829110ea737daa145.png)

В приведенном выше примере:

* Узел имеет ключи, равные максимальным ключам.
* Ключ вставляется в целевой узел, но он нарушает правило максимальных ключей.
* Целевой узел разделен, и средний ключ по левому смещению теперь является родителем новых дочерних узлов.
* Новые узлы расположены в порядке возрастания.

Точно так же, основываясь на вышеуказанных правилах и случаях, остальные значения могут быть легко вставлены в B-дерево.

[](https://coderlessons.com/wp-content/uploads/images/gur/882484af87b59ce4bcb5c023e1a12af7.png)

**Удалить операцию**

Операция удаления имеет больше правил, чем операции вставки и поиска.

Применяется следующий алгоритм:

* Запустите операцию поиска и найдите целевой ключ в узлах
* Три условия применяются в зависимости от местоположения целевого ключа, как описано в следующих разделах

**Если целевой ключ находится в листовом узле**

* Цель находится в листовом узле, больше чем мин ключей.
  + Удаление этого не будет нарушать свойство B Tree
* Цель находится в листовом узле, она имеет минимальные ключевые узлы
  + Удаление этого нарушит свойство B Tree
  + Целевой узел может заимствовать ключ от непосредственного левого узла или непосредственного правого узла (одноуровневого)
  + Брат или сестра скажут « **да»,** если количество ключей превышает минимальное.
  + Ключ будет заимствован из родительского узла, максимальное значение будет передано родительскому узлу, максимальное значение родительского узла будет передано целевому узлу, а целевое значение будет удалено.
* Цель находится в листовом узле, но ни один из братьев и сестер не имеет более минимального количества ключей
  + Поиск ключа
  + Слияние с братьями и сестрами и минимум родительских узлов
  + Всего ключей будет больше минуты
  + Целевой ключ будет заменен минимумом родительского узла.

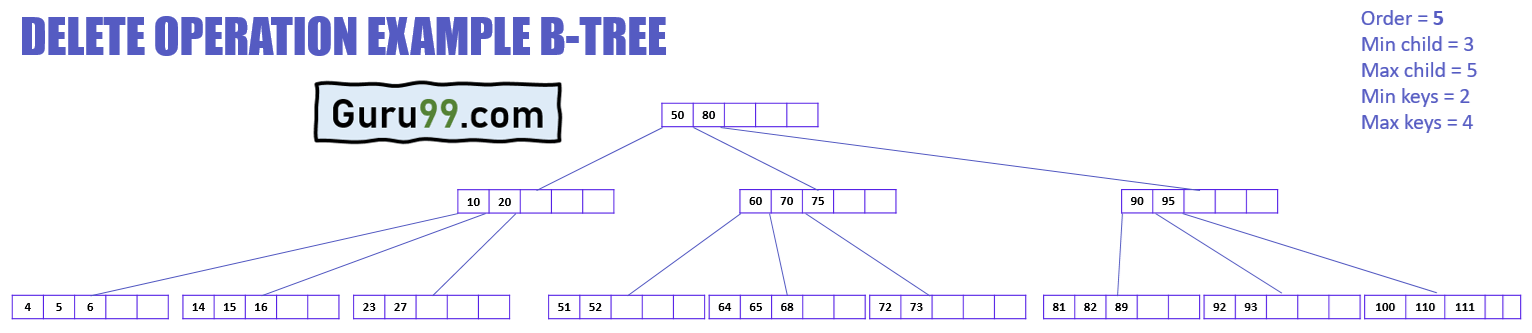
**Если целевой ключ находится во внутреннем узле**

* Либо выберите предшественник по порядку или преемник по порядку
* В случае предшествующего предшественника будет выбран максимальный ключ из его левого поддерева.
* В случае преемника по порядку будет выбран минимальный ключ из его правого поддерева
* Если предшествующий по порядку целевой ключ имеет больше, чем клавиши min, только тогда он может заменить целевой ключ с максимумом предшествующего по порядку ключа
* Если в предшествующем по порядку целевом ключе не больше, чем min ключей, ищите минимальный ключ в преемнике по порядку.
* Если предшествующий предшественник и преемник целевого ключа имеют ключи меньше чем min, объедините предшественник и преемник.

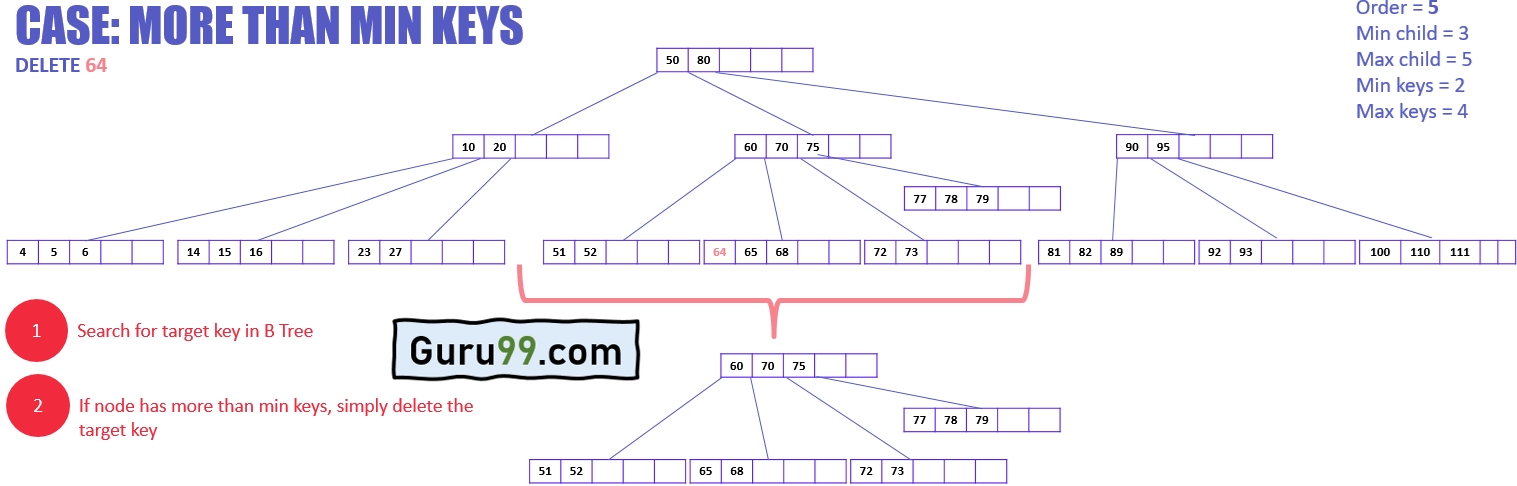
**Если целевой ключ находится в корневом узле**

* Заменить максимальным элементом в поддереве предшественника по порядку
* Если после удаления у цели меньше, чем мин ключей, то целевой узел заимствует максимальное значение у своего родного брата через родителя родного брата.
* Максимальное значение родителя будет взято целью, но с узлами максимального значения родного брата.

Теперь давайте разберемся с операцией удаления на примере.

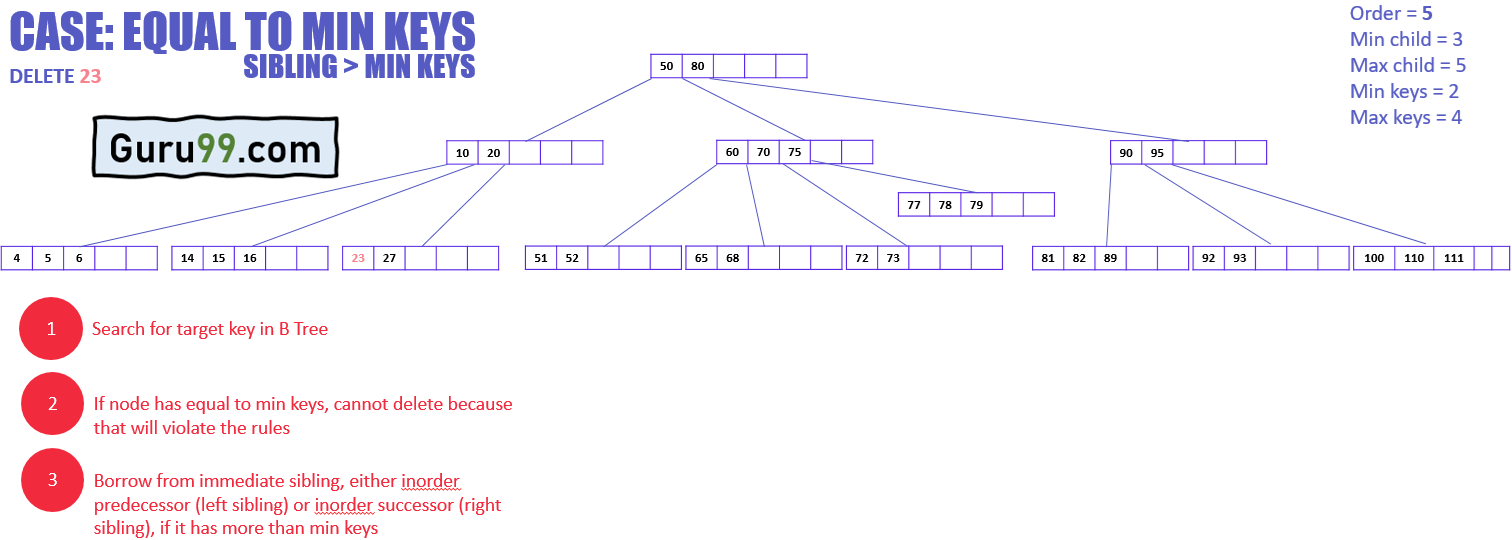
[](https://coderlessons.com/wp-content/uploads/images/gur/78ff0e8fa4ea4d494734cf1b7ccb4058.png)

На приведенной выше диаграмме показаны различные случаи операции удаления в B-дереве. Это B-дерево имеет порядок 5, что означает, что минимальное количество дочерних узлов, которое может иметь любой узел, равно 3, а максимальное количество дочерних узлов, которое может иметь любой узел, равно 5. При этом минимальное и максимальное количество ключей любого узла могут иметь 2 и 4 соответственно.

[](https://coderlessons.com/wp-content/uploads/images/gur/36c8265be5a9f3c34cca766447aca3c1.png)

В приведенном выше примере:

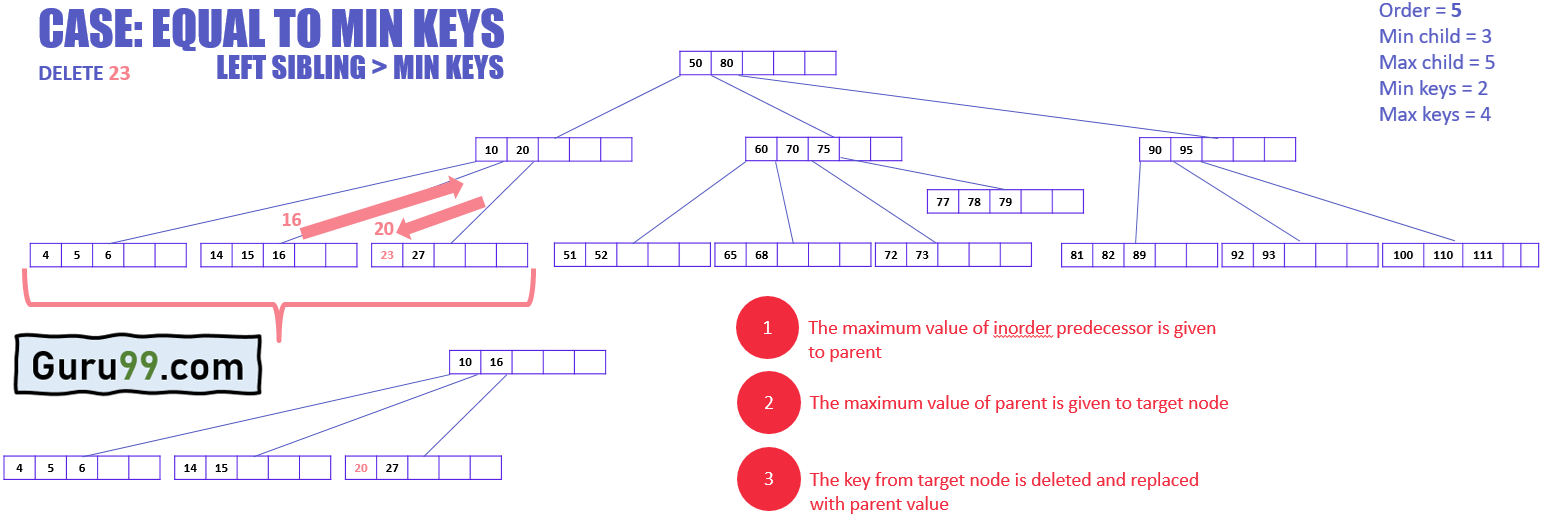
* Целевой узел имеет целевой ключ для удаления
* Целевой узел имеет ключи больше чем минимальные ключи
* Просто удалите ключ

[](https://coderlessons.com/wp-content/uploads/images/gur/07c6b4b48fef00ca8f150508cbe68ec8.png)

В приведенном выше примере:

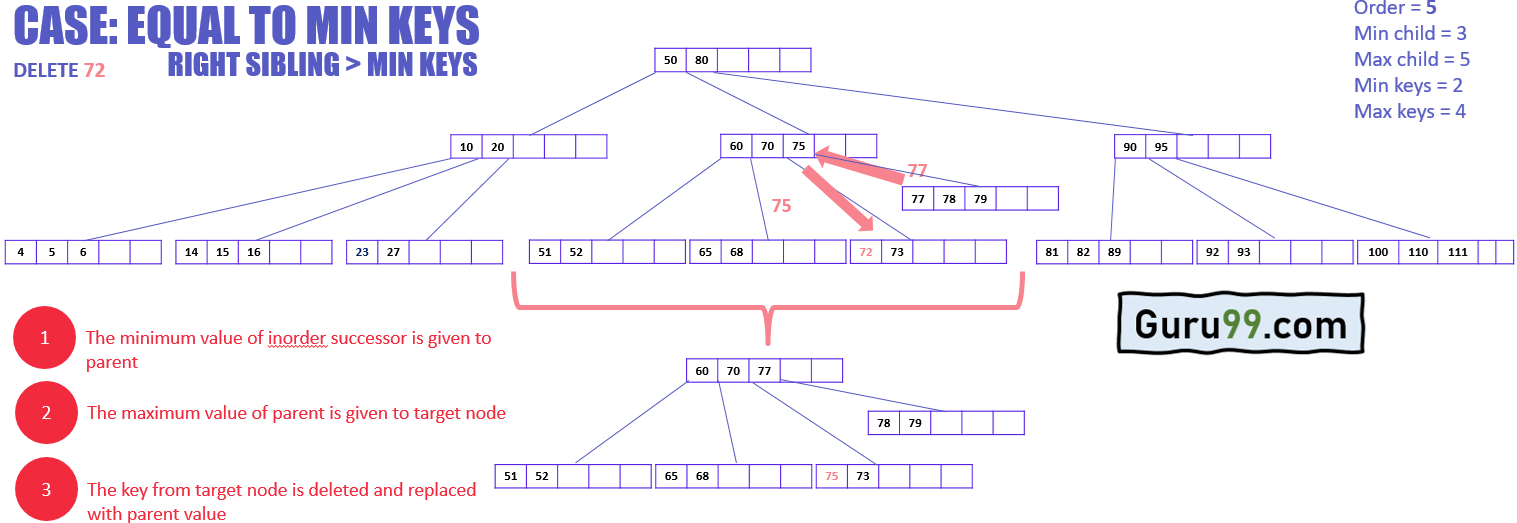
* Целевой узел имеет ключи, равные минимальным ключам, поэтому не может удалить его напрямую, так как это нарушит условия

Теперь следующая диаграмма объясняет, как удалить этот ключ:

[](https://coderlessons.com/wp-content/uploads/images/gur/7a8bd994f3f291cbe2699ca911244734.png)

* Целевой узел заимствует ключ у ближайшего родственника, в данном случае предшественника по порядку (левый брат), потому что у него нет преемника по порядку (правый брат)
* Максимальное значение предшествующего предшественника будет передано родительскому элементу, а родительское максимальное значение будет передано целевому узлу (см. Диаграмму ниже).

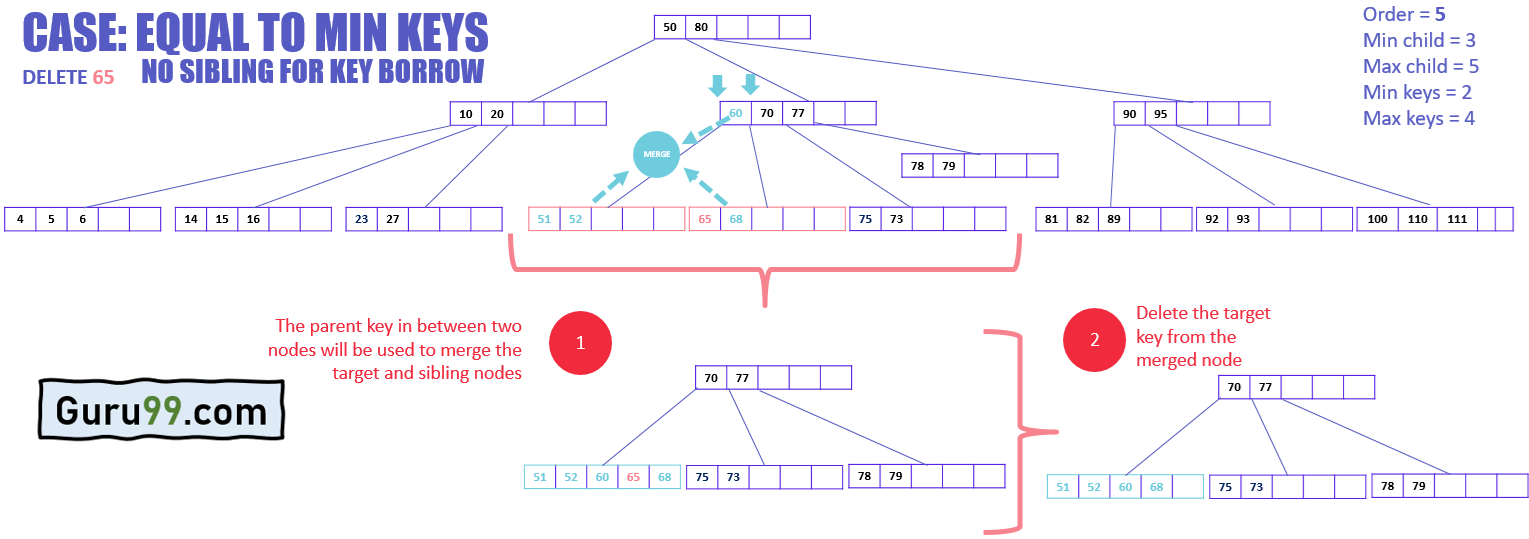
В следующем примере показано, как удалить ключ, которому необходимо значение, из его преемника по порядку.

[](https://coderlessons.com/wp-content/uploads/images/gur/1c5037fc8fcfb857e0fb55ba4701a975.png)

* Целевой узел заимствует ключ у ближайшего родственника, в данном случае преемника по порядку (правый брат), потому что его предшественник по порядку (левый брат) имеет ключи, равные минимальным ключам.
* Минимальное значение преемника по порядку будет передано родительскому элементу, а родительский элемент передаст максимальное значение целевому узлу.

В приведенном ниже примере целевой узел не имеет ни одного родного брата, который мог бы передать свой ключ целевому узлу. Поэтому слияние не требуется.

Смотрите процедуру удаления такого ключа:

[](https://coderlessons.com/wp-content/uploads/images/gur/8a88a1cea1f1c575eba89c9f1471fd16.png)

* Объединить целевой узел с любым из его ближайших братьев и сестер вместе с родительским ключом
  + Выбран ключ из родительского узла, который находится между двумя объединяющимися узлами.
* Удалить целевой ключ из объединенного узла

**Удалить псевдокод операции**

private int removeBiggestElement()

{

if (root has no child)

remove and return the last element

else {

answer = subset[childCount-1].removeBiggestElement()

if (subset[childCount-1].dataCount < MINIMUM)

fixShort (childCount-1)

return answer

}

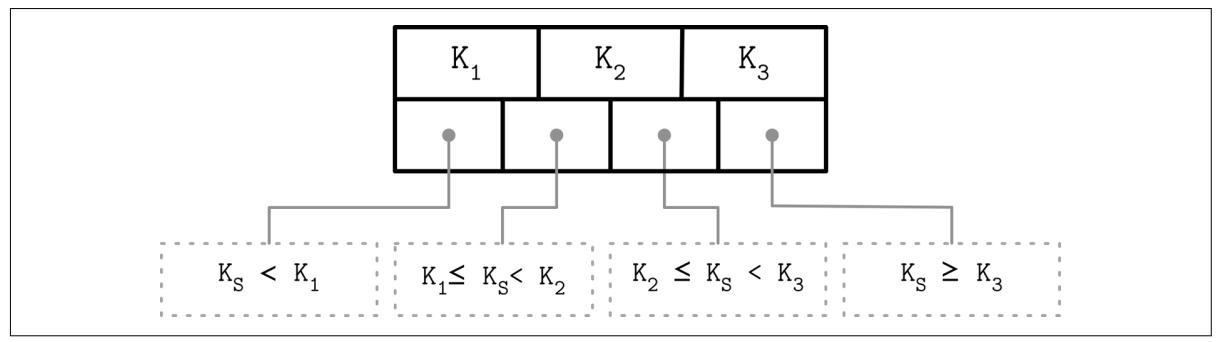
}

**Выход** :

Самый большой элемент удаляется из B-дерева.

**Резюме:**

* B Tree – это самобалансирующаяся структура данных для лучшего поиска, вставки и удаления данных с диска.
* B Дерево регулируется по степени, указанной
* B Ключи и узлы дерева расположены в порядке возрастания.
* Операция поиска B Tree – самая простая, которая всегда начинается с корня и начинает проверять, больше или меньше целевой ключ, чем значение узла.
* Операция вставки B-дерева довольно детальная, которая сначала находит подходящую позицию вставки для целевого ключа, вставляет ее, оценивает валидность B-дерева в различных случаях, а затем соответствующим образом реструктурирует узлы B-дерева.
* Операция удаления B Tree сначала ищет целевой ключ, который будет удален, удаляет его, оценивает допустимость на основе нескольких случаев, таких как минимальный и максимальный ключи целевого узла, братья и сестры и родительский элемент.



// Поиск ключа в B-дереве на C++

#include <iostream>

using namespace std;

class TreeNode {

int \*keys;

int t;

TreeNode \*\*C;

int n;

bool leaf;

public:

TreeNode(int temp, bool bool\_leaf);

void insertNonFull(int k);

void splitChild(int i, TreeNode \*y);

void traverse();

TreeNode \*search(int k);

friend class BTree;

};

class BTree {

TreeNode \*root;

int t;

public:

BTree(int temp) {

root = NULL;

t = temp;

}

void traverse() {

if (root != NULL)

root->traverse();

}

TreeNode \*search(int k) {

return (root == NULL) ? NULL : root->search(k);

}

void insert(int k);

};

TreeNode::TreeNode(int t1, bool leaf1) {

t = t1;

leaf = leaf1;

keys = new int[2 \* t - 1];

C = new TreeNode \*[2 \* t];

n = 0;

}

void TreeNode::traverse() {

int i;

for (i = 0; i < n; i++) {

if (leaf == false)

C[i]->traverse();

cout << " " << keys[i];

}

if (leaf == false)

C[i]->traverse();

}

TreeNode \*TreeNode::search(int k) {

int i = 0;

while (i < n && k > keys[i])

i++;

if (keys[i] == k)

return this;

if (leaf == true)

return NULL;

return C[i]->search(k);

}

void BTree::insert(int k) {

if (root == NULL) {

root = new TreeNode(t, true);

root->keys[0] = k;

root->n = 1;

} else {

if (root->n == 2 \* t - 1) {

TreeNode \*s = new TreeNode(t, false);

s->C[0] = root;

s->splitChild(0, root);

int i = 0;

if (s->keys[0] < k)

i++;

s->C[i]->insertNonFull(k);

root = s;

} else

root->insertNonFull(k);

}

}

void TreeNode::insertNonFull(int k) {

int i = n - 1;

if (leaf == true) {

while (i >= 0 && keys[i] > k) {

keys[i + 1] = keys[i];

i--;

}

keys[i + 1] = k;

n = n + 1;

} else {

while (i >= 0 && keys[i] > k)

i--;

if (C[i + 1]->n == 2 \* t - 1) {

splitChild(i + 1, C[i + 1]);

if (keys[i + 1] < k)

i++;

}

C[i + 1]->insertNonFull(k);

}

}

void TreeNode::splitChild(int i, TreeNode \*y) {

TreeNode \*z = new TreeNode(y->t, y->leaf);

z->n = t - 1;

for (int j = 0; j < t - 1; j++)

z->keys[j] = y->keys[j + t];

if (y->leaf == false) {

for (int j = 0; j < t; j++)

z->C[j] = y->C[j + t];

}

y->n = t - 1;

for (int j = n; j >= i + 1; j--)

C[j + 1] = C[j];

C[i + 1] = z;

for (int j = n - 1; j >= i; j--)

keys[j + 1] = keys[j];

keys[i] = y->keys[t - 1];

n = n + 1;

}

int main() {

BTree t(3);

t.insert(8);

t.insert(9);

t.insert(10);

t.insert(11);

t.insert(15);

t.insert(16);

t.insert(17);

t.insert(18);

t.insert(20);

t.insert(23);

cout << "B-дерево: ";

t.traverse();

int k = 10;

(t.search(k) != NULL) ? cout << endl

<< k << " найдено"

: cout << endl

<< k << " не найдено";

k = 2;

(t.search(k) != NULL) ? cout << endl

<< k << " найдено"

: cout << endl

<< k << " не найдено\n";

}