B-деревья — это тип самобалансирующегося дерева, предназначенного для хранения данных в памяти внешних устройств, таких как жесткие диски. Они часто используются в базах данных и файловых системах. Вот основные концепции и характеристики B-деревьев:

1. **Структура**: B-дерево — это дерево поиска, в котором узлы могут иметь большее количество дочерних узлов по сравнению с двоичными деревьями. Каждый узел содержит ключи и указатели на дочерние узлы.
2. **Упорядочение**: Ключи в узле упорядочены. Все ключи в поддереве слева от ключа меньше этого ключа, а все ключи в поддереве справа — больше.
3. **Параметр B-дерева**: B-дерево определяется своим параметром *t*, который является минимальным количеством ключей, которые узел должен содержать. Каждый узел, кроме корня, должен содержать как минимум t-1 ключей. Корень должен содержать как минимум один ключ.
4. **Максимальное и минимальное количество ключей**: Каждый узел может содержать максимум 2t-1 ключей.
5. **Сбалансированность**: Все листовые узлы находятся на одном уровне, что обеспечивает сбалансированность и гарантирует, что операции поиска, вставки и удаления выполняются за логарифмическое время.
6. **Операции**:
   * **Поиск**: Похож на поиск в двоичном дереве поиска, но мы можем переходить сразу через несколько ключей в узле.
   * **Вставка**: Новый ключ вставляется в существующий узел. Если узел переполнен (имеет 2t ключей), он разделяется.
   * **Удаление**: Удаление ключа может требовать переструктуризации дерева для сохранения его свойств.
7. **Применение**: B-деревья чрезвычайно полезны в системах, где данные, с которыми происходят операции, не умещаются в основной памяти и должны часто читаться с или записываться на медленные внешние накопители.

B-деревья представляют собой эффективный и масштабируемый способ индексации данных, что делает их важным инструментом в области баз данных и файловых систем.

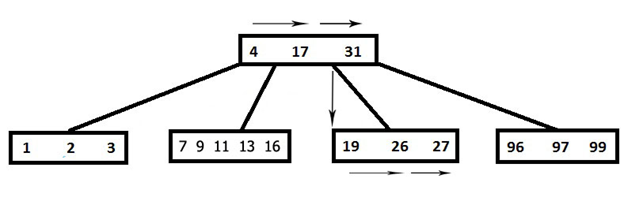
# Структура

При построении B-дерева применяется фактор t, который называется минимальной степенью. Каждый узел, кроме корневого, должен иметь, как минимум t – 1, и не более 2t – 1 ключей. Обозначается n[x] – количество ключей в узле x.  
  
Ключи в узле хранятся в неубывающем порядке. Если x не является листом, то он имеет n[x] + 1 детей. Если занумеровать ключи в узле x, как k[i], а детей c[i], то для любого ключа в поддереве с корнем c[i] (пусть k1), выполняется следующее неравенство – k[i-1] ≤k1≤k[i] (для c[0]: k[i-1] = -∞, а для c[n[x]]: k[i] = +∞). Таким образом, ключи узла задают диапазон для ключей их детей.  
  
Все листья B-дерева должны быть расположены на одной высоте, которая и является высотой дерева. Высота B-дерева с n ≥ 1 узлами и минимальной степенью t ≥ 2 не превышает logt(n+1). Это очень важное утверждение (почему – мы поймем чуть позже)!  
  
h ≤ logt((n+1)/2) — логарифм по основанию t.

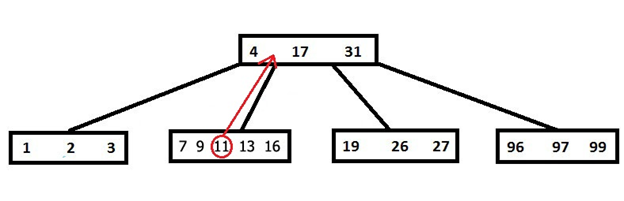
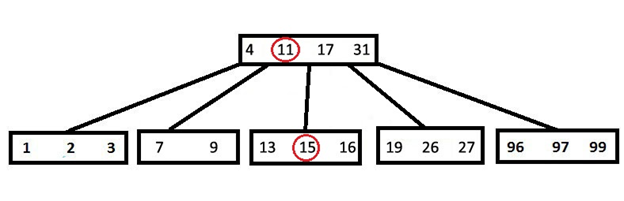
##### **Поиск**

Поиск в B-дереве очень схож с поиском в бинарном дереве, только здесь мы должны сделать выбор пути к потомку не из 2 вариантов, а из нескольких. В остальном — никаких отличий. На рисунке ниже показан поиск ключа 27. Поясним иллюстрацию (и соответственно стандартный алгоритм поиска):

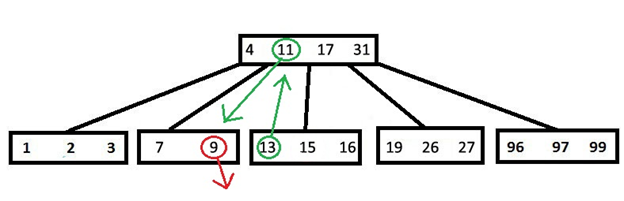
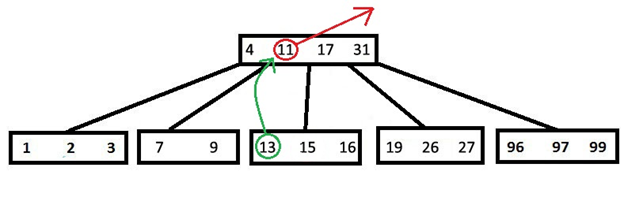
* Идем по ключам корня, пока меньше необходимого. В данном случае дошли до 31.
* Спускаемся к ребенку, который находится левее этого ключа.
* Идем по ключам нового узла, пока меньше 27. В данном случае – нашли 27 и остановились.

  
  
Операция поиска выполняется за время O(t logt n), где t – минимальная степень. Важно здесь, что дисковых операций мы совершаем всего лишь O(logt n)!

##### **Добавление**

В отличие от поиска, операция добавления существенно сложнее, чем в бинарном дереве, так как просто создать новый лист и вставить туда ключ нельзя, поскольку будут нарушаться свойства B-дерева. Также вставить ключ в уже заполненный лист невозможно => необходима операция разбиения узла на 2. Если лист был заполнен, то в нем находилось 2t-1 ключей => разбиваем на 2 по t-1, а средний элемент (для которого t-1 первых ключей меньше его, а t-1 последних больше) перемещается в родительский узел. Соответственно, если родительский узел также был заполнен – то нам опять приходится разбивать. И так далее до корня (если разбивается корень – то появляется новый корень и глубина дерева увеличивается). Как и в случае обычных бинарных деревьев, вставка осуществляется за один проход от корня к листу. На каждой итерации (в поисках позиции для нового ключа – от корня к листу) мы разбиваем все заполненные узлы, через которые проходим (в том числе лист). Таким образом, если в результате для вставки потребуется разбить какой-то узел – мы уверены в том, что его родитель не заполнен!  
  
На рисунке ниже проиллюстрировано то же дерево, что и в поиске (t=3). Только теперь добавляем ключ «15». В поисках позиции для нового ключа мы натыкаемся на заполненный узел (7, 9, 11, 13, 16). Следуя алгоритму, разбиваем его – при этом «11» переходит в родительский узел, а исходный разбивается на 2. Далее ключ «15» вставляется во второй «отколовшийся» узел. Все свойства B-дерева сохраняются!  
  
  
  
  
  
Операция добавления происходит также за время O(t logt n). Важно опять же, что дисковых операций мы выполняем всего лишь O(h), где h – высота дерева.

##### **Удаление**

Удаление ключа из B-дерева еще более громоздкий и сложный процесс, чем вставка. Это связано с тем, что удаление из внутреннего узла требует перестройки дерева в целом. Аналогично вставке необходимо проверять, что мы сохраняем свойства B-дерева, только в данном случае нужно отслеживать, когда ключей t-1 (то есть, если из этого узла удалить ключ – то узел не сможет существовать). Рассмотрим алгоритм удаления:  
1)Если удаление происходит из листа, то необходимо проверить, сколько ключей находится в нем. Если больше t-1, то просто удаляем и больше ничего делать не нужно. Иначе, если существует соседний лист (находящийся рядом с ним и имеющий такого же родителя), который содержит больше t-1 ключа, то выберем ключ из этого соседа, который является разделителем между оставшимися ключами узла-соседа и исходного узла (то есть не больше всех из одной группы и не меньше всех из другой). Пусть это ключ k1. Выберем ключ k2 из узла-родителя, который является разделителем исходного узла и его соседа, который мы выбрали ранее. Удалим из исходного узла нужный ключ (который необходимо было удалить), спустим k2 в этот узел, а вместо k2 в узле-родителе поставим k1. Чтобы было понятнее ниже представлен рисунок (рис.1), где удаляется ключ «9». Если же все соседи нашего узла имеют по t-1 ключу. То мы объединяем его с каким-либо соседом, удаляем нужный ключ. И тот ключ из узла-родителя, который был разделителем для этих двух «бывших» соседей, переместим в наш новообразовавшийся узел (очевидно, он будет в нем медианой).  
Рис. 1.  
  
  
2)Теперь рассмотрим удаление из внутреннего узла x ключа k. Если дочерний узел, предшествующий ключу k содержит больше t-1 ключа, то находим k1 – предшественника k в поддереве этого узла. Удаляем его (рекурсивно запускаем наш алгоритм). Заменяем k в исходном узле на k1. Проделываем аналогичную работу, если дочерний узел, следующий за ключом k, имеет больше t-1 ключа. Если оба (следующий и предшествующий дочерние узлы) имеют по t-1 ключу, то объединяем этих детей, переносим в них k, а далее удаляем k из нового узла (рекурсивно запускаем наш алгоритм). Если сливаются 2 последних потомка корня – то они становятся корнем, а предыдущий корень освобождается. Ниже представлен рисунок (рис.2), где из корня удаляется «11» (случай, когда у следующего узла больше t-1 ребенка).  
Рис.2.  
  
  
Операция удаления происходит за такое же время, что и вставка O(t logt n). Да и дисковых операций требуется всего лишь O(h), где h – высота дерева.  
  
Итак, мы убедились в том, что B-дерево является быстрой структурой данных (наряду с такими, как красно-черное, АВЛ). И еще одно важное свойство, которое мы получили, рассмотрев стандартные операции, – автоматическое поддержание свойства сбалансированности – заметим, что мы нигде не балансируем его специально.

Source: https://habr.com/ru/articles/114154/