Лекции 13-14

Типы и структуры данных

3 семестр

Пчелинцева Наталья Ибрагимовна

Оглавление

Деревья отрезков

Деревья отрезков

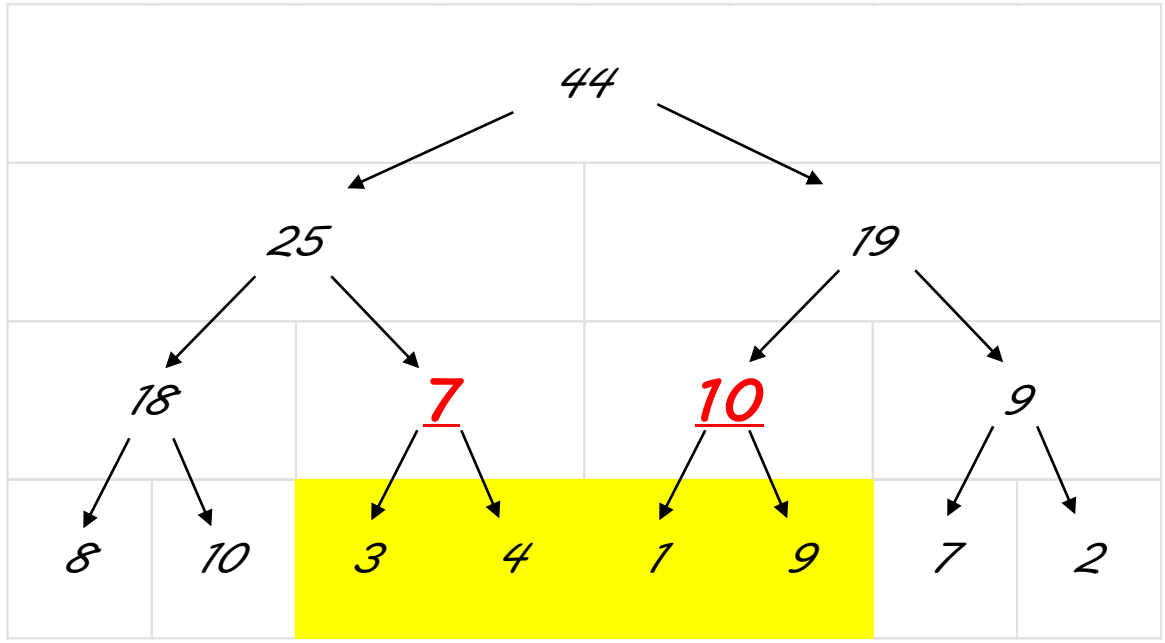
- **Дерево отрезков –** это структура данных, которая позволяет алгоритмически просто и логарифмически быстро находить сумму элементов массива на заданном отрезке.

- **Дерево отрезков –** полное бинарное дерево, в котором каждая вершина отвечает за некоторый отрезок в массиве.

Деревья отрезков

- Необходимо найти сумму чисел:
  - 8, 10, 3, 4, 1, 9, 7, 2
- Задача: **максимально быстро найти сумму любой последовательности из этих чисел**.
- Например:

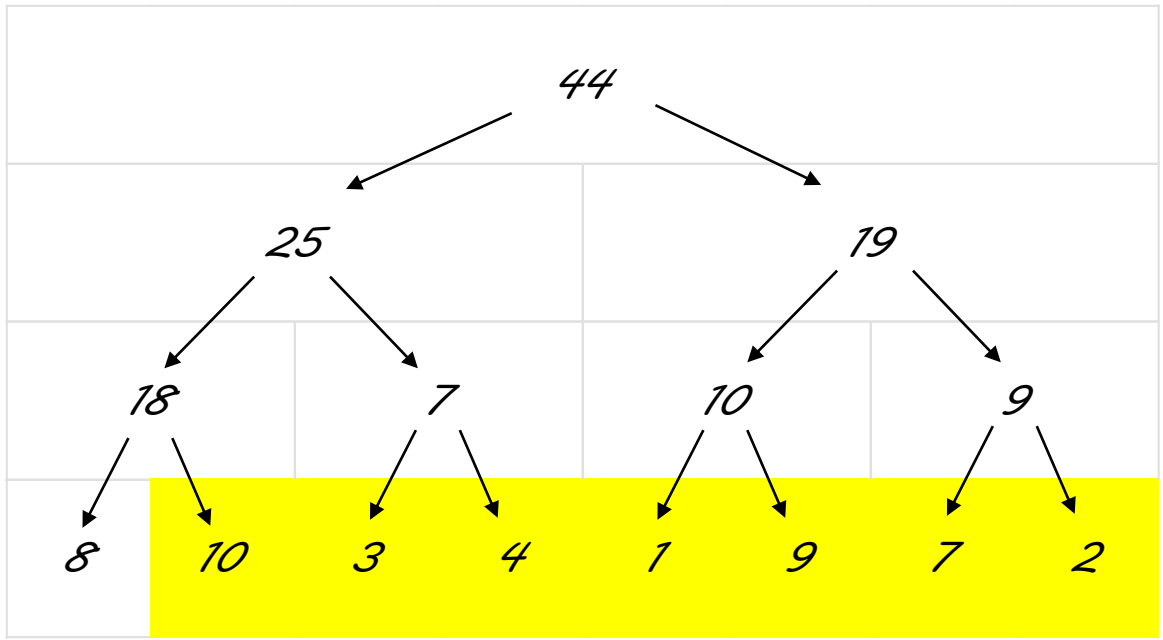
Деревья отрезков

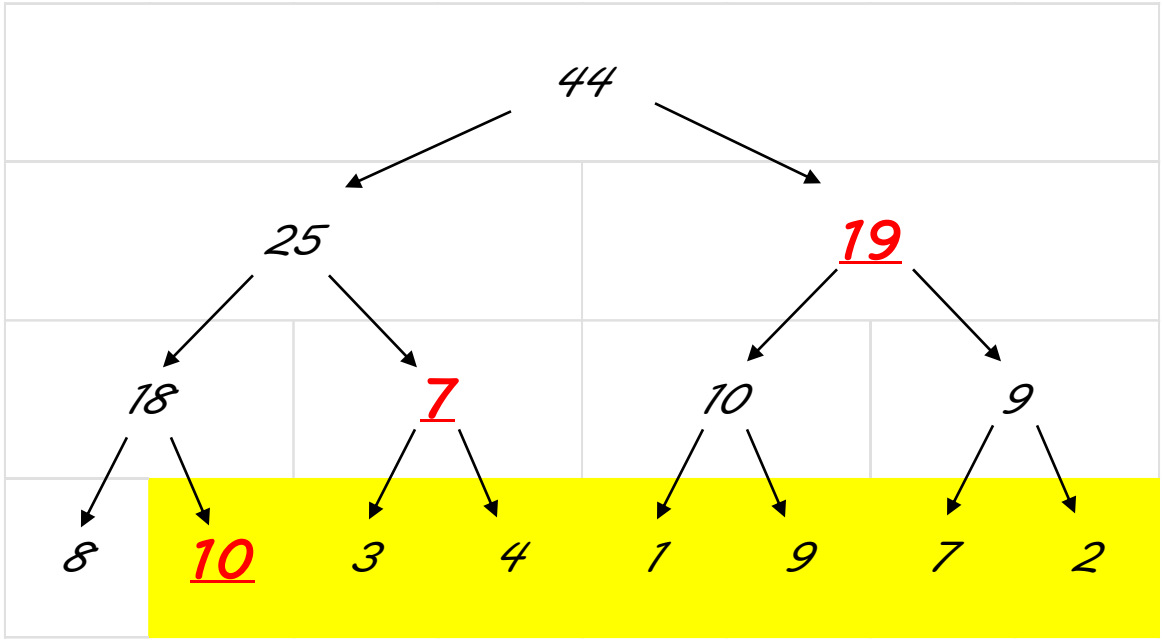


- В результате мы складываем только два числа вместо четырёх: 7 + 10 = 17

Деревья отрезков

- Усложняем задачу. Необходимо посчитать:





- Поучаем: = 19 + 7 + 10 = 36

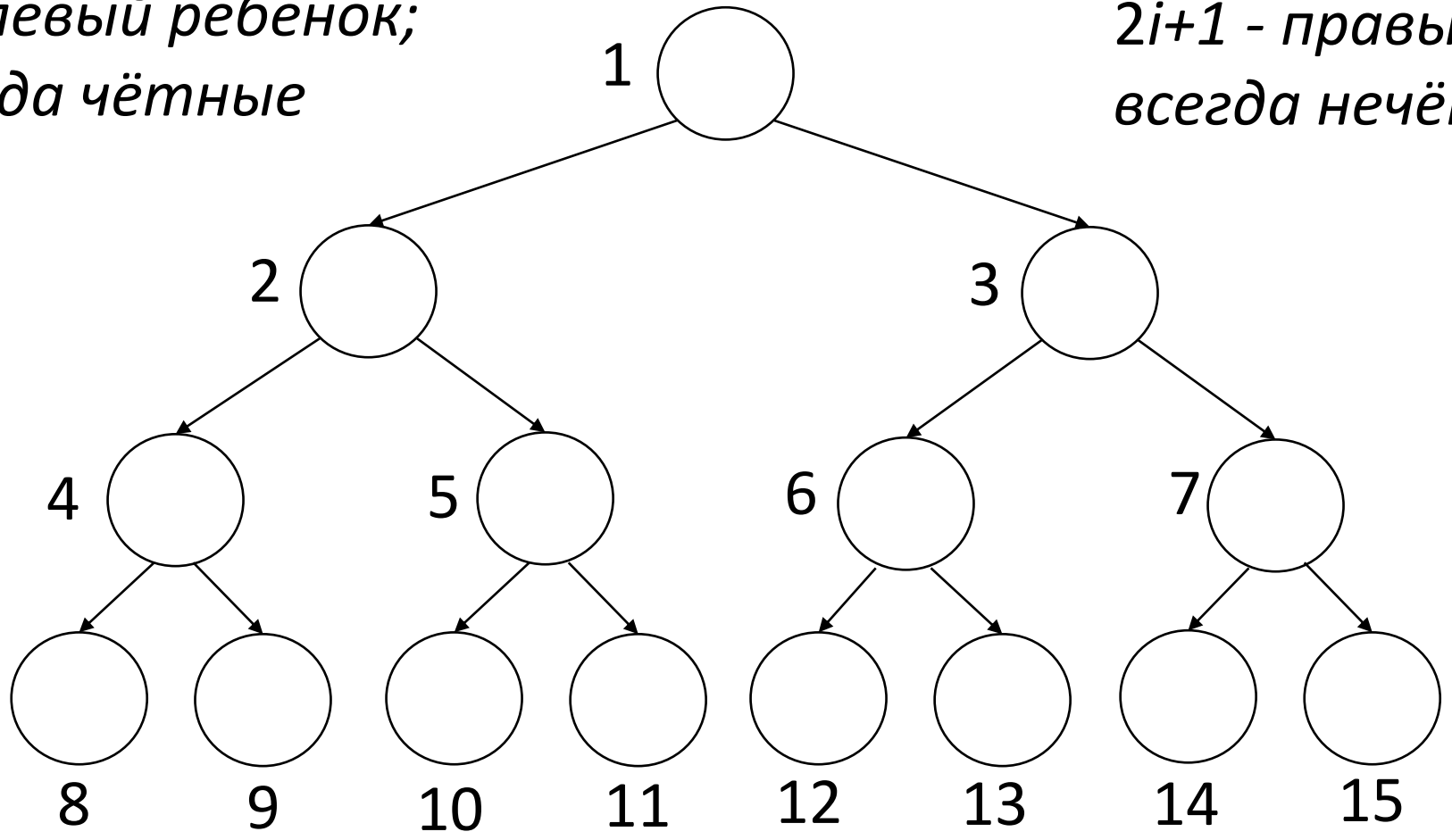
Деревья отрезков

- **Полное двоичное дерево** – это дерево, у каждого элемента которого есть ровно два дочерних элемента.
- Для работы с полным двоичным деревом можно и нужно использовать такую структуру данных, как массив.
- Нумеровать этот массив удобно с единицы. Пронумеруем каждый элемент двоичного дерева натуральными числами от 1 до 2n-1.

Деревья отрезков

2i - левый ребёнок;
всегда чётные

2i+1 - правый ребёнок;
всегда нечётные



для определения родителя ребёнка
используется целочисленное деление *i*/2

Деревья отрезков

- Сколько потребуется памяти для хранения двоичного дерева, внизу которого находятся эти элементы?
  - Ответ: 2n, если n является степенью двойки.
- С какого элемента необходимо разместить исходные числа в массиве полного двоичного дерева?
  - Ответ: у нас 8 элементов, всего в массиве будет 16 элементов, значит, первый элемент будет под номером 16 - 8 = 8. И начинать строить нужно будем слева-направо и снизу-вверх, начиная с 7 элемента, складывая значения у детей.

```
For elements:
8 10 3 4 1 9 7 2
Segment tree is:
44 25 19 18 7 10 9 8 10 3 4 1 9 7 2
```

Двоичные кучи

Двоичные кучи (binary heap)

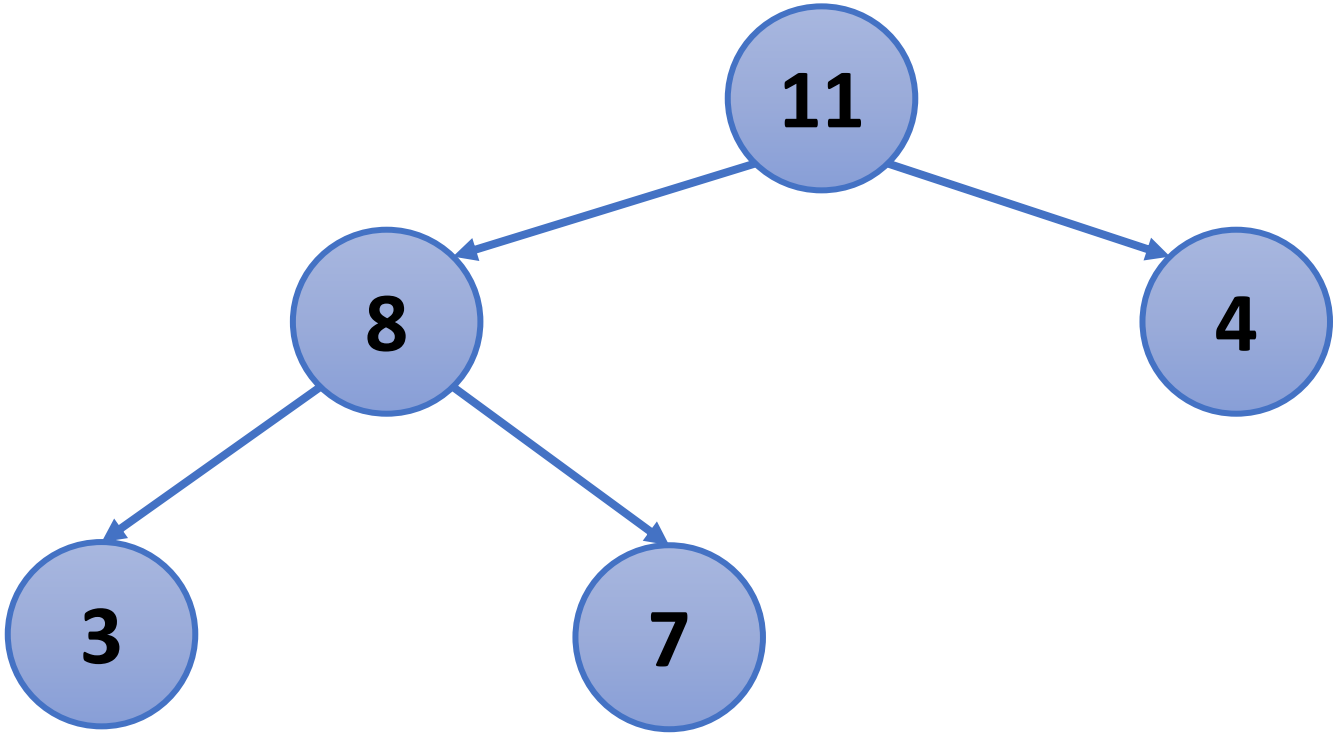
- Разновидность полного бинарного дерева.
- Двоичная куча – НЕ дерево поиска.
- Основное свойство: приоритет каждой вершины больше приоритета её потомков.
- В простейшем случае – приоритет каждой вершины можно считать равным её значению.
- В таком случае структура называется max-heap, поскольку корень дерева является максимумом из значений элементов поддерева.

Двоичные кучи

- Дерево называется полным бинарным, если у каждой вершины есть не более двух потомков, а заполнение уровней вершин идёт сверху вниз (в пределах одного уровня – слева-направо).
- Допускается противоположный вариант, в котором родитель меньше потомков (min heap).

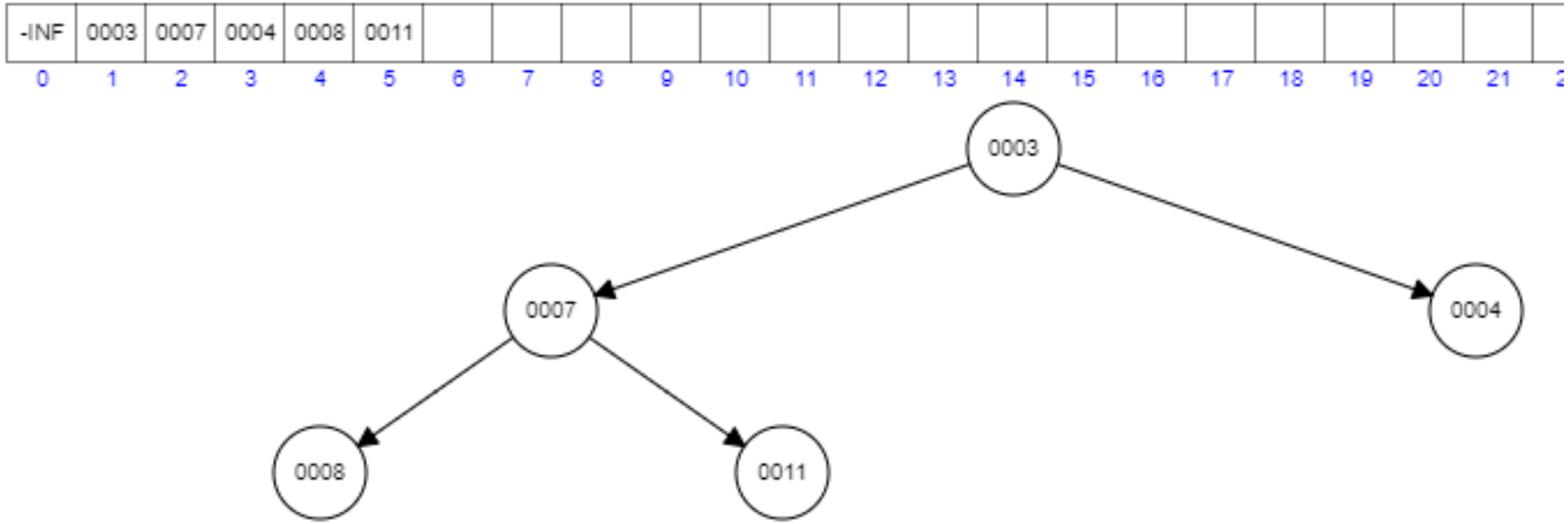
Двоичные кучи

- Пример корректной Max heap кучи (8, 3, 4, 7, 11):



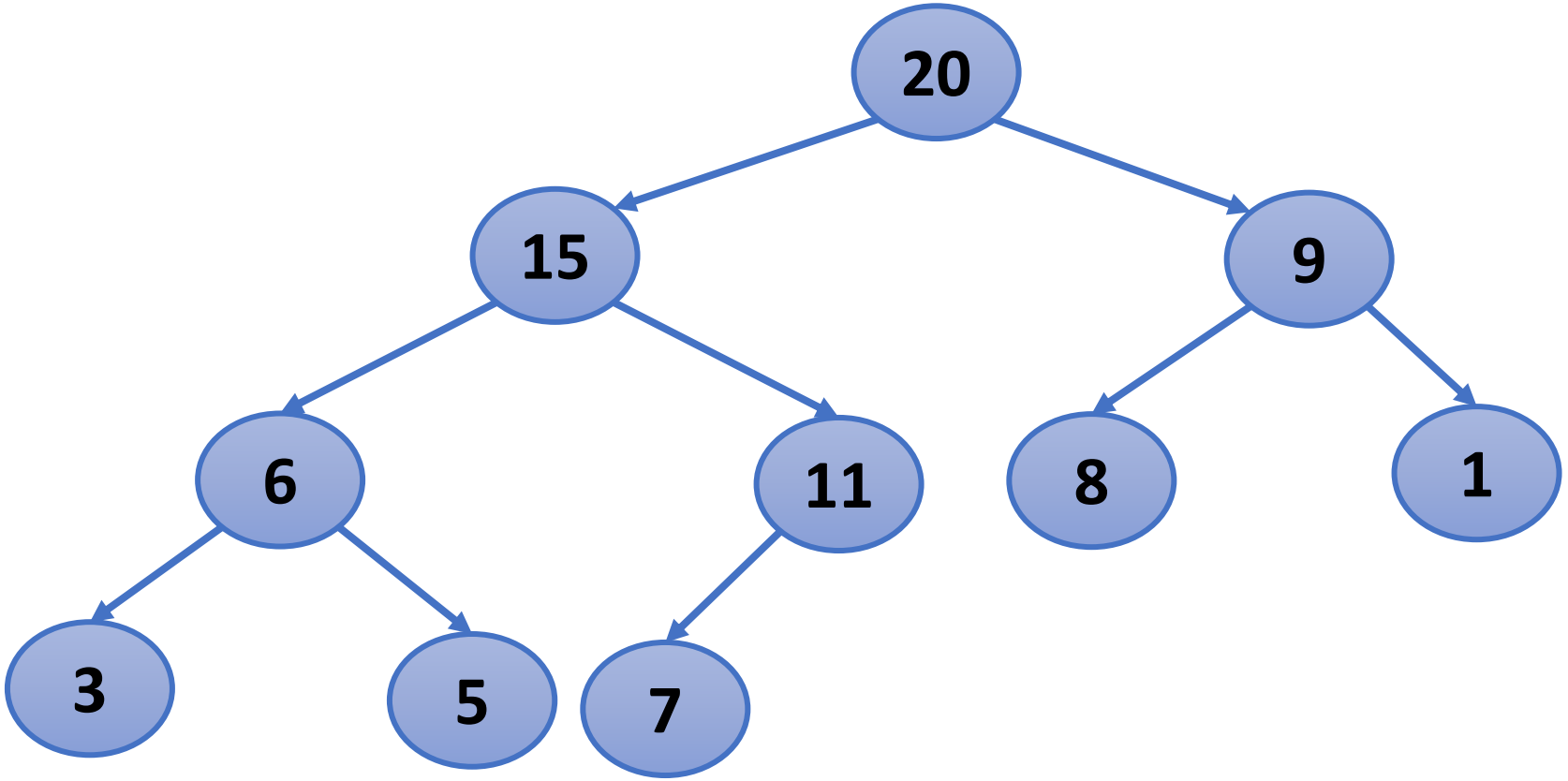
Двоичные кучи

- Пример корректной Min heap кучи (8, 3, 4, 7, 11):



Двоичные кучи

- Пример корректной кучи (6, 9, 11, 7, 15, 8, 1, 3, 5, 20):



20	15	9	6	11	8	1	3	5	7

Двоичные кучи

Добавление элемента

- Новый элемент добавляется на последнее место в массиве, то есть в позицию с индексом *heapSize*.
- Возможно, что при этом будет нарушено основное свойство кучи, так как новый элемент может быть больше родителя.
- В таком случае следует «поднимать» новый элемент на один уровень (менять с вершиной-родителем) до тех пор, пока не будет соблюдено основное свойство кучи.

Двоичные кучи

Корректирование кучи:

- Новый элемент «всплывает», «проталкивается» вверх, пока не займет свое место.

Упорядочение двоичной кучи:

- В ходе других операций с уже построенной двоичной кучей также может нарушиться основное свойство кучи: вершина может стать меньше своего потомка.
- Метод *heapify* восстанавливает основное свойство кучи для дерева с корнем в i-ой вершине при условии, что оба поддерева ему удовлетворяют.

Двоичные кучи

Построение двоичной кучи:

- Наиболее очевидный способ построить кучу из неупорядоченного массива – это по очереди добавить все его элементы.
- Временная оценка такого алгоритма O(N log2 N).
- Однако можно построить кучу еще быстрее за – O(N).
- Сначала следует построить дерево из всех элементов массива, не заботясь о соблюдении основного свойства кучи, а потом вызвать метод *heapify* для всех свойства кучи, а потом вызвать метод *heapify* для всех вершин, у которых есть хотя бы один потомок (так как поддеревья, состоящие из одной вершины без потомков, уже упорядочены).

- Визуализация построения двоичной кучи (Min heap): https://www.cs.usfca.edu/~galles/visualization/Heap.html

Хеш-таблицы
Фильтр Блума

Хеш-таблицы

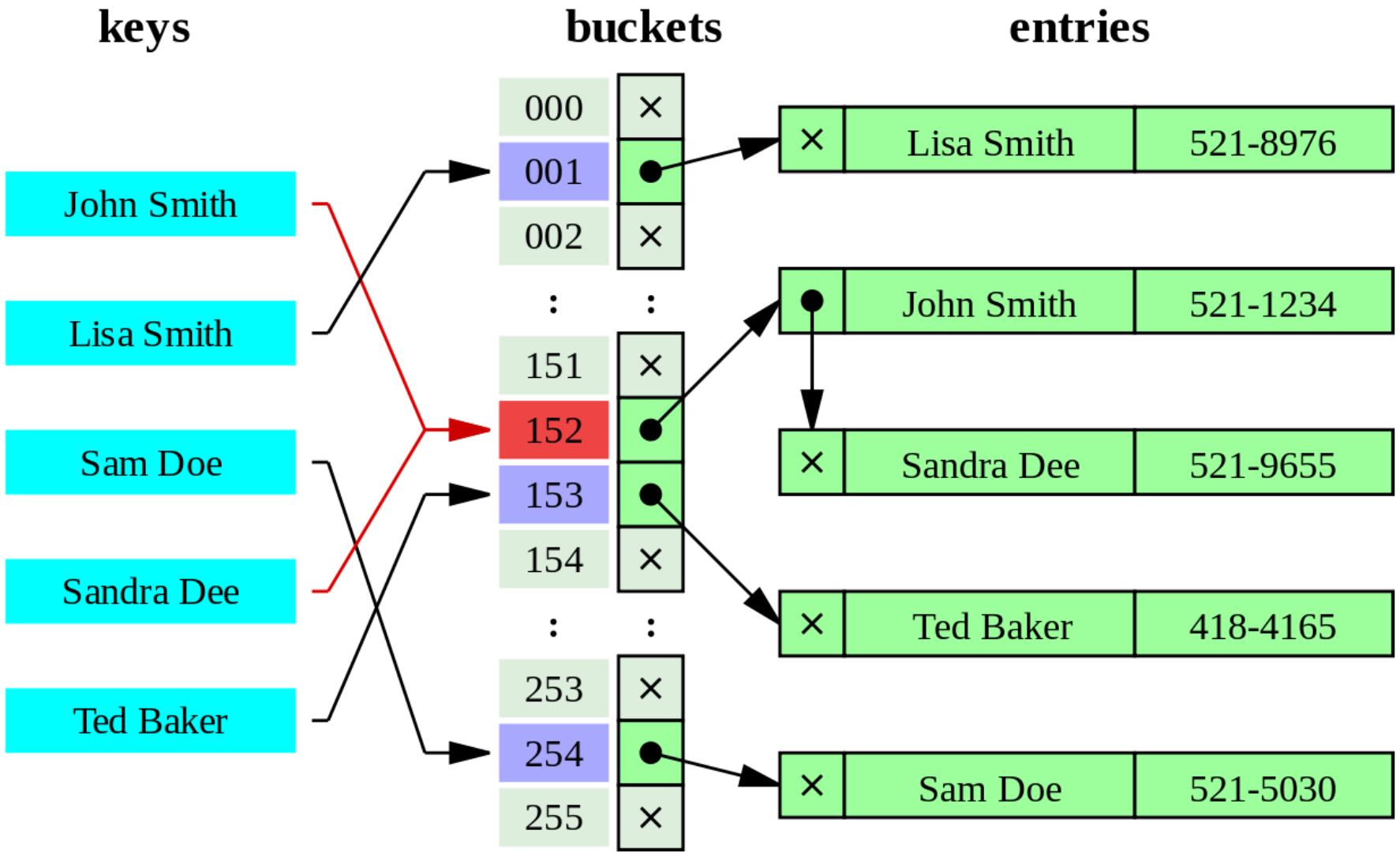
- Преимущества использования хеш-таблиц перед другими структурами данных:

Контейнер \ операция	insert	remove	find
Array	O(N)	O(N)	O(N)
List	O(1)	O(1)	O(N)
Sorted array	O(N)	O(N)	O(logN)
Бинарное дерево поиска	O(logN)	O(logN)	O(logN)
Хеш-таблица	O(1)	O(1)	O(1)

Хеш-таблицы

- Хеш-таблица – это контейнер, который используют быстрой работы операции вставки/удаления/нахождения (C++: unoredered_set и unordered_map, Python: set, dict).
- Хеш-функция – функция отображение значения (например, строки) на массив с получением натурального числа (индекса массива).
- Проблема коллизии хеш-функций: получение одинакового натурального числа для разных значений.
- Решение: выбор хеш-функции – метод двойного хеширования или метод цепочек.

Хеш-таблицы (решение коллизии методом открытых цепочек)



Хеш-таблицы (решение проблемы коллизии методом двойного хеширования)

- Используются две хеш-функции, возвращающие взаимопростые натуральные числа.
- Одна хеш-функция (при входе *g*) будет возвращать натуральное число *s*, которое будет для нас начальным. То есть первое, что мы сделаем, попробуем поставить элемент *g* на позицию *s* в нашем массиве.
- Но что, если это место уже занято? Именно здесь нам пригодится вторая хеш-функция, которая будет возвращать *t* – шаг, с которым мы будем в дальнейшем искать место, куда бы поставить элемент *g*.
- Мы будем рассматривать сначала элемент *s*, потом *s + t*, затем *s + 2*t* и т.д. Естественно, чтобы не выйти за границы массива, мы обязаны смотреть на номер элемента по модулю (остатку от деления на размер массива).

Хеш-таблицы

Для предотвращения коллизий необходимы:

- Низкий коэффициент заполнения массива;
- Хорошая хеш-функция.

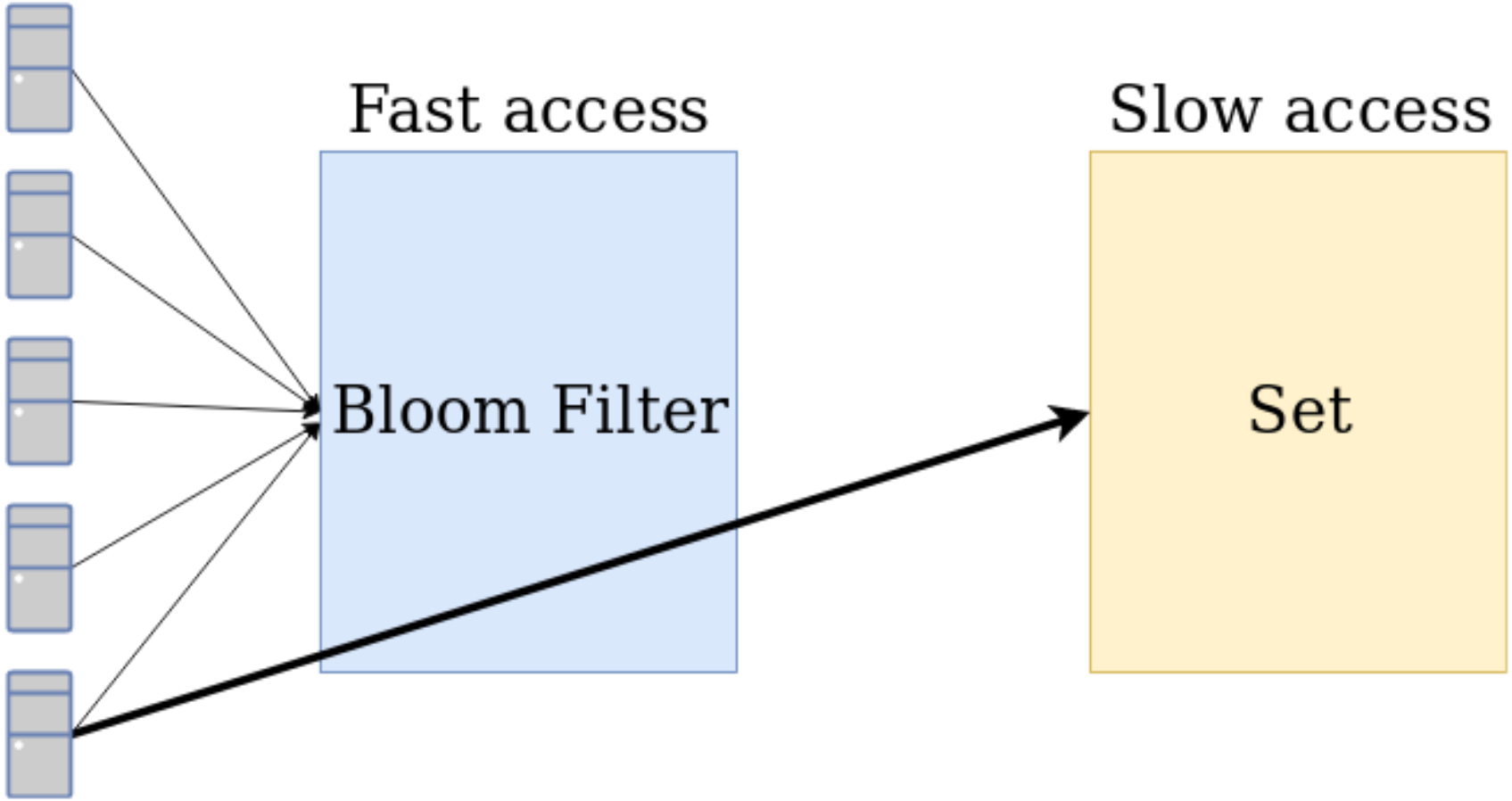
- Коэффициент заполнения – отношения количество заполненных элементов таблицы к общему количеству элементов.

- Визуализация хеш-таблицы методом открытых цепочек: https://www.cs.usfca.edu/~galles/visualization/OpenHash.html

Фильтр Блума

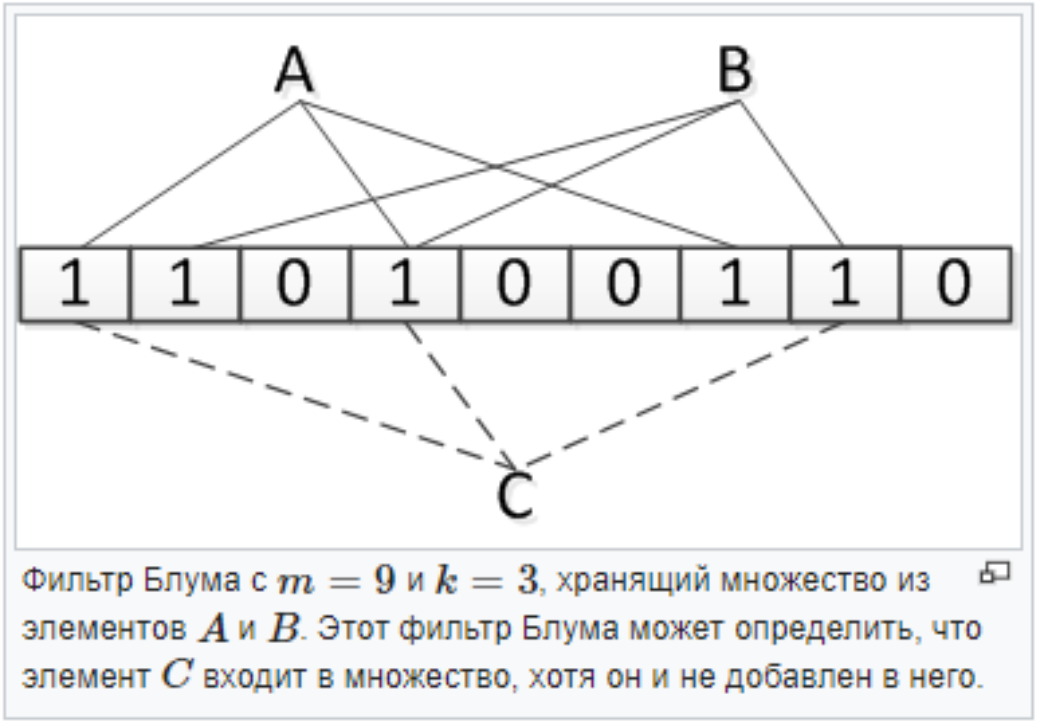
- Фильтр Блума – это структура данных, цель которой – быстро проверить, что элемент НЕ входит в множество (сложность вставки и проверки принадлежности элемента к множеству с помощью фильтра Блума – O(1)).
- Он может быть очень полезен для предотвращения излишнего выполнения задач, требующих интенсивных вычислений, просто проверяя, что элемент *совершенно точно не входит в множество*.
- Важно понимать, что фильтр Блума – это вероятностная структура данных: он может сказать вам со 100% вероятностью, что элемент отсутствует в наборе данных, но сказать со 100% вероятностью, что элемент находится в наборе, он не может (возможны ложно положительные результаты).

Фильтр Блума



Фильтр Блума

- Фильтр Блума представляет собой битовый массив из *m* бит и *k* различных хеш-функций *h1…hk*, равновероятно отображающих элементы исходного множества во множество *{0,1,…m−1}*, соответствующее номерам битов в массиве. Изначально, когда структура данных хранит пустое множество, все *m* бит обнулены.
- Для добавления элемента *e* необходимо записать единицы на каждую из позиций *h1(e)…hk(e)* битового массива.
- Чтобы проверить, что элемент *e* принадлежит множеству хранимых элементов, необходимо проверить состояние битов *h1(e)…hk(e)*.
- Если хотя бы один из них равен нулю, элемент не принадлежит множеству.
- Если все они равны единице, то структура данных сообщает, что элемент принадлежит множеству. При этом может возникнуть две ситуации: либо элемент действительно принадлежит к множеству, либо все эти биты оказались установлены при добавлении других элементов, что и является источником ложных срабатываний в этой структуре данных.
- По сравнению с хеш-таблицами, фильтр Блума может обходиться на несколько порядков меньшими объёмами памяти, жертвуя детерминизмом.



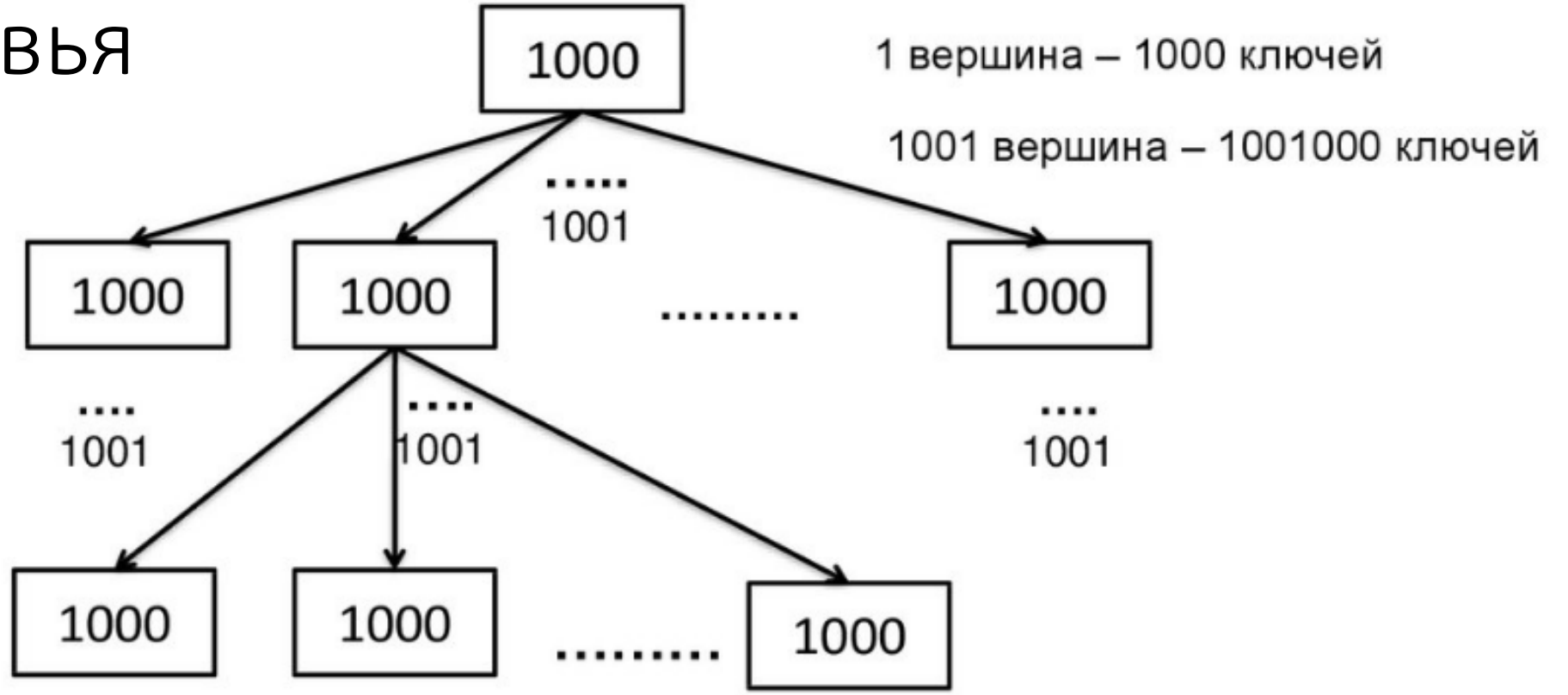
Фильтр Блума с $m = 9$ и $k = 3$, хранящий множество из элементов A и B. Этот фильтр Блума может определить, что элемент C входит в множество, хотя он и не добавлен в него.

B-деревья

B-деревья

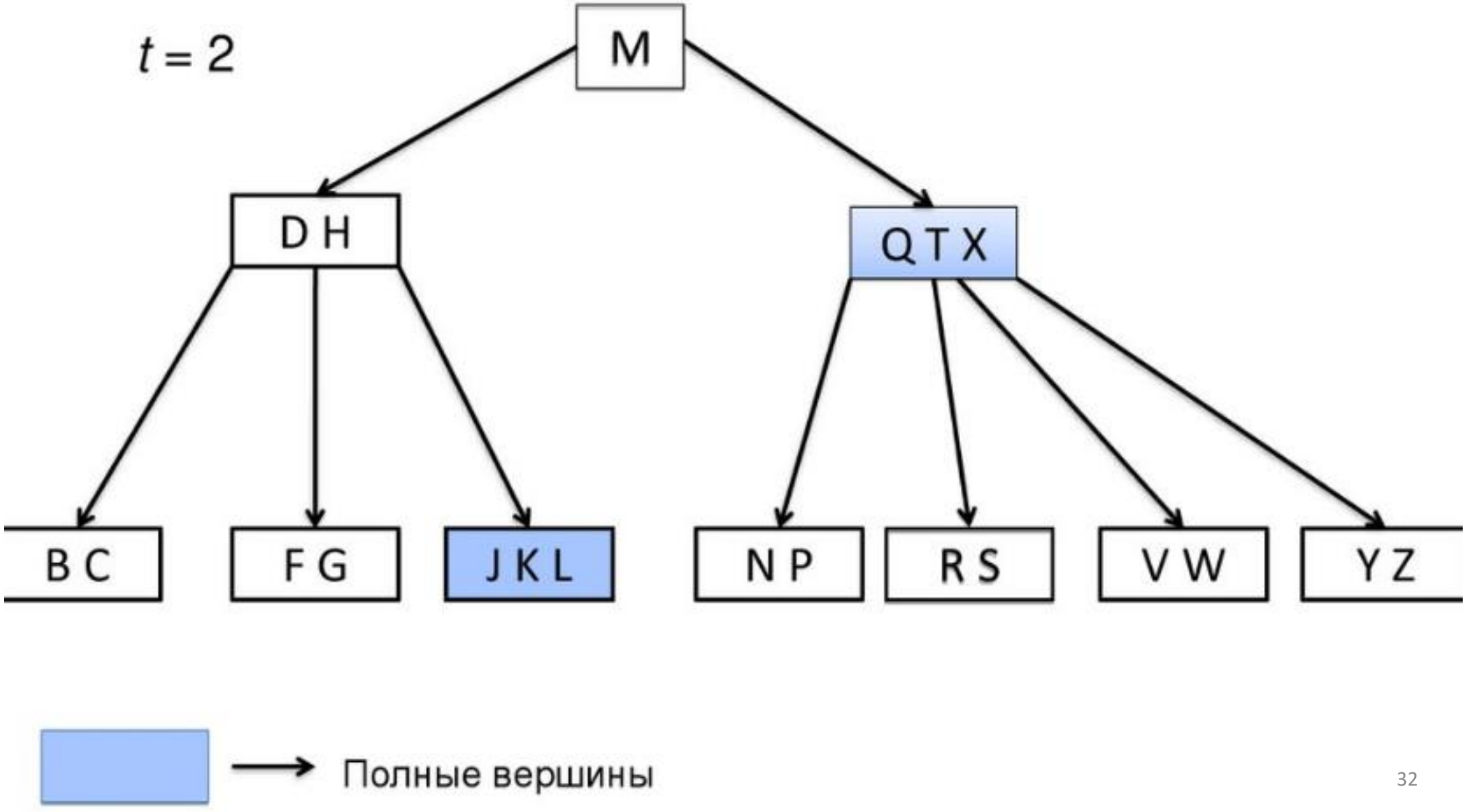
- B-дерево (B-tree) – это самобалансирующаяся древовидная структура данных, которая поддерживает отсортированные данные и позволяет осуществлять поиск, последовательный доступ, вставки и удаления в логарифмическом времени.
- B-дерево – это обобщение бинарного дерева поиска, в котором у узла может быть более двух дочерних элементов.
- В отличие от других самобалансирующихся бинарных деревьев поиска, B-дерево хорошо подходит для систем хранения, которые читают и записывают относительно большие блоки данных, такие как диски. Обычно используется в базах данных и файловых системах.

B-деревья



Б-дерево высоты 2 – содержит более миллиарда ключей.
Каждая вершина содержит 1000 ключей.
Более миллиона листьев на глубине 2.

B-деревья



B-деревья

У таких деревьев, как правило, только корень находиться в ОП, остальное дерево – на диске.

Диск разбит на сектора (дорожки на сектора).

Обычно записывают или считывают сектор целиком.

Время доступа, чтобы подвести головку к нужному месту на диске, может быть достаточно большим (до 20 миллисекунд).

Как только головка диска установлена, запись или чтение происходит довольно быстро.

Часто получатся, что обработка прочитанного занимает меньше времени, чем поиск нужного сектора.

Важно количество обращений к диску!

B-деревья

- В *B*-деревьях внутренние (неконечные) узлы могут иметь переменное число дочерних узлов в некотором предопределенном диапазоне. Когда данные вставляются или удаляются из узла, его количество дочерних узлов изменяется.
- Чтобы поддерживать предварительно определенный диапазон, внутренние узлы могут быть соединены или разделены. Поскольку диапазон дочерних узлов разрешен, B-деревья не нуждаются в перебалансировке так часто, как другие самобалансирующиеся деревья поиска, но могут тратить некоторое пространство, так как узлы не полностью заполнены.
- Нижняя и верхняя границы количества дочерних узлов обычно фиксируются для конкретной реализации. Например, в 2-3 B-дереве (часто просто называемом 2-3-деревом (2-3 tree)) каждый внутренний узел может иметь только 2 или 3 дочерних узла.

B-деревья

- Каждый внутренний узел B-дерева содержит несколько ключей. Ключи действуют как значения разделения, которые разделяют его поддеревья. Например, если внутренний узел имеет 3 дочерних узла (или поддерева), то он должен иметь 2 ключа: a1 и a2. Все значения в крайнем левом поддереве будут меньше, чем a1, все значения в среднем поддереве будут между a1 и a2, а все значения в крайнем правом поддереве будут больше, чем a2.
- Обычно количество ключей выбирается в зависимости от t до 2t, где t - минимальное количество ключей, а t+1 - минимальная степень или коэффициент ветвления дерева.

B-деревья

- Число ветвей (или дочерних узлов) от узла будет на один больше, чем количество ключей, хранящихся в узле.
- B-дерево сохраняется сбалансированным после вставки путем разбиения потенциального переполненного узла из 2t + 1 ключей на два дочерних элемента t-ключа и вставки ключа среднего значения в родитель. Глубина увеличивается только при разделении корня, сохраняя равновесие.

B-деревья

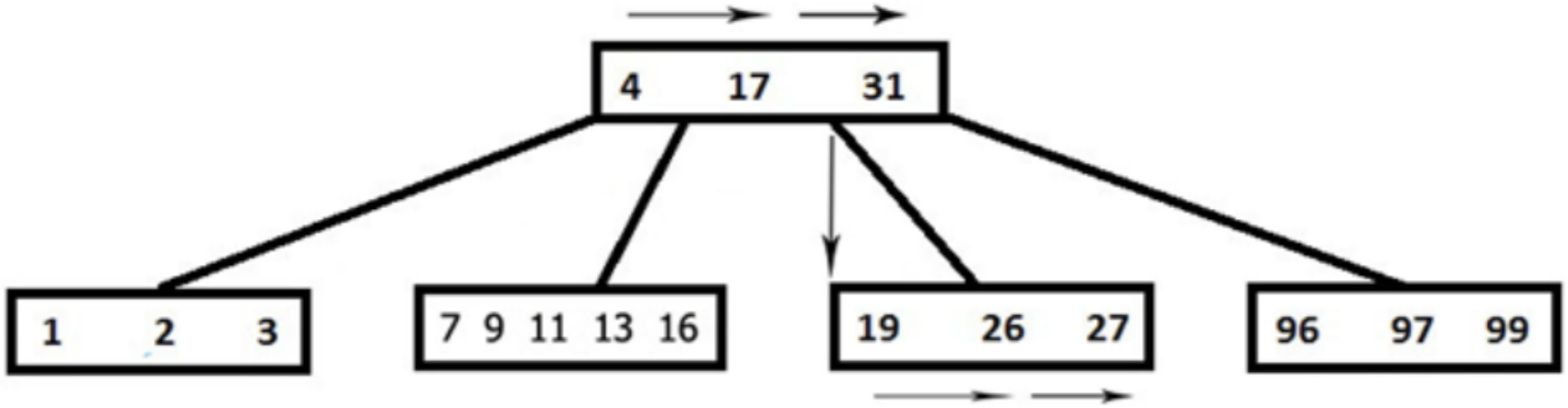
- Аналогично, B-дерево сохраняется сбалансированным после удаления за счет слияния или перераспределения ключей между братьями и сестрами, чтобы поддерживать минимум t-ключа для некорневых узлов. Слияние уменьшает количество ключей в родительском объекте, потенциально вынуждая его объединять или перераспределять ключи со своими братьями и сестрами и так далее.
- Единственное изменение в глубине происходит, когда корень имеет двух дочерних элементов, ключей t и (переходно) t-1, в этом случае два родных брата и родительский элемент объединяются, уменьшая глубину одним.

B-деревья

- B-деревья имеют существенные преимущества по сравнению с альтернативными реализациями, когда время доступа к данным узла значительно превышает время, затрачиваемое на обработку этих данных.
- Благодаря максимизации количества ключей в каждом внутреннем узле высота дерева уменьшается, а количество обращений к узлам уменьшается. Кроме того, ребалансировка дерева происходит реже.

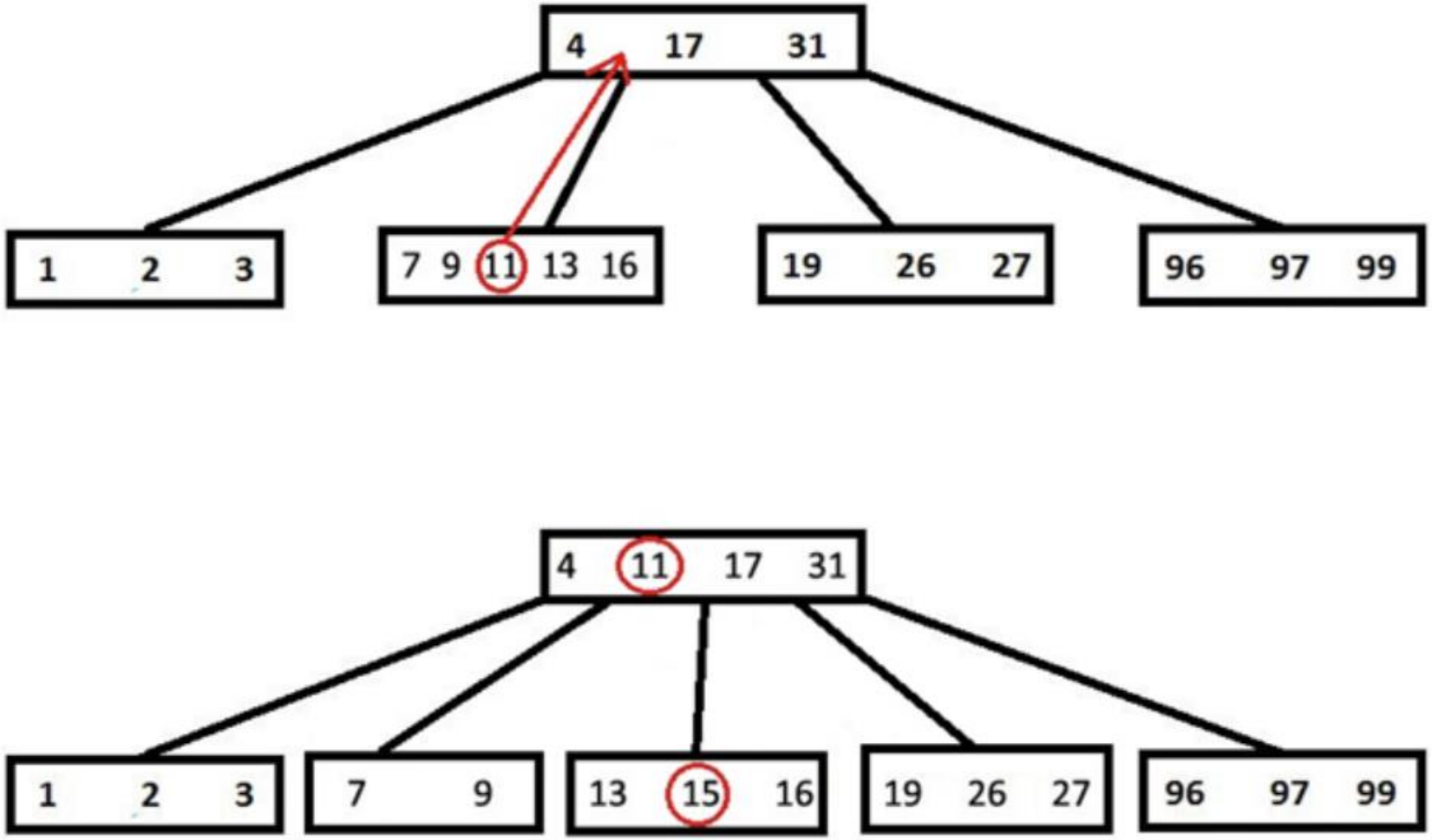
B-деревья

- Поиск 27 в B-дереве:
  - 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.



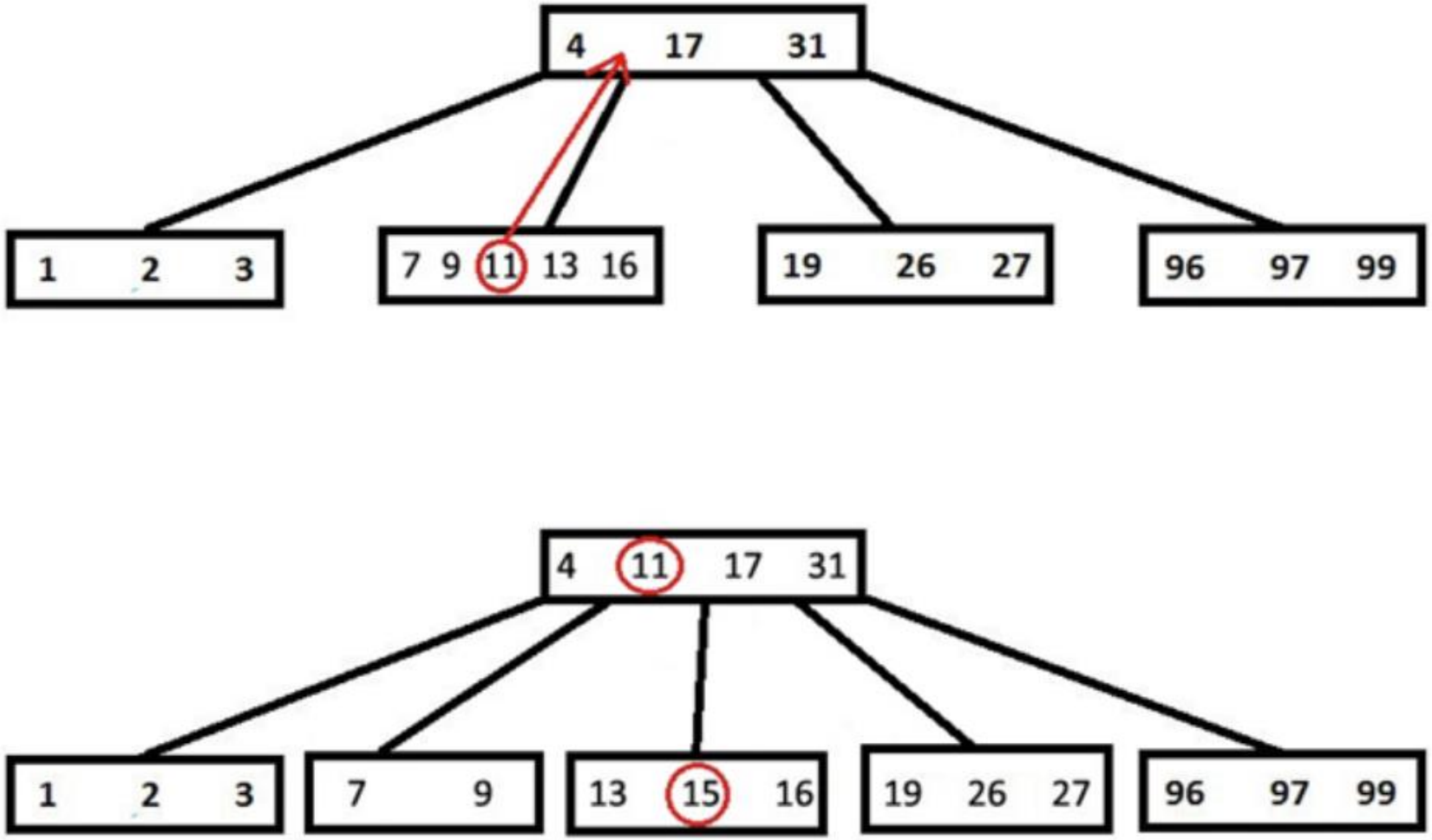
B-деревья

- Добавление ключа 15 в *B*-дерево:



B-деревья

- Добавление ключа 15 в *B*-дерево:



B-деревья

- B-деревья имеют существенные преимущества по сравнению с альтернативными реализациями, когда время доступа к данным узла значительно превышает время, затрачиваемое на обработку этих данных.
- Благодаря максимизации количества ключей в каждом внутреннем узле высота дерева уменьшается, а количество обращений к узлам уменьшается. Кроме того, ребалансировка дерева происходит реже.
- Визуализация построения B-дерева: https://www.cs.usfca.edu/~galles/visualization/BTree.html