

Лекция 5

В-деревья (B-trees)

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

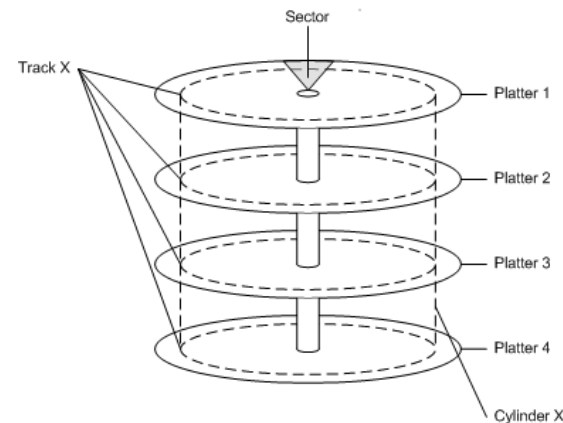
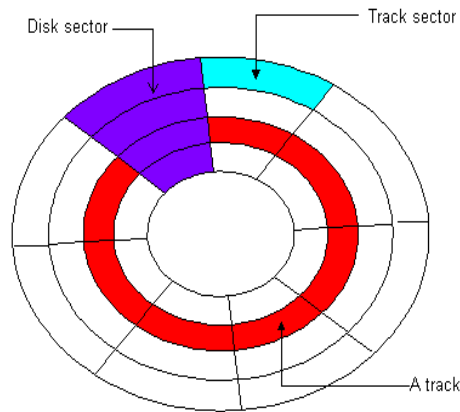
WWW: www.mkurnosov.net

Курс «Структуры и алгоритмы обработки данных»

Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

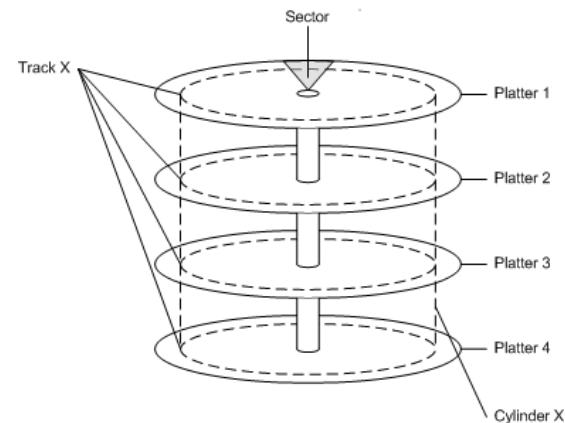
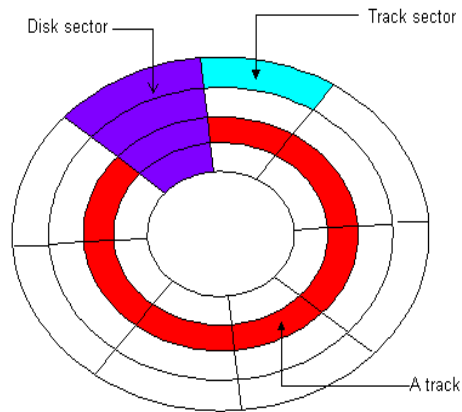
Осенний семестр, 2015

Организация дисковой памяти



- **Жёсткий диск (hard disk drive, HDD)** – это совокупность пластин (platters), которые хранят данные в концентрических окружностях – **дорожках (tracks)**
- Данные считываются/записываются головками (heads)
- Пластины и головки приводятся в движение двигателями
- **Сектор (Sector)** – часть дорожки, минимально адресуемая единица жесткого диска
- **Кластер (Cluster)** – это совокупность секторов дорожки
- **Цилиндр (Cylinder)** – множество дорожек, размещенных на одном и том же месте нескольких пластин

Организация дисковой памяти



- Время доступа к сектору (seek time) зависит от скорости вращения пластин

Количество вращений в минуту (Revolutions per minute, RPM)	Время доступа (Rotational latency)
5400 rpm	0.011 сек.
7200 rpm	0.008 сек.
15 000 rpm	0.004 сек.

- Время доступа к оперативной памяти (≈ 10 нс) в 10^5 раз меньше времени доступа к диску

Твердотельные накопители (SSD)

- **Твердотельный накопитель (Solid state drive, flash drive, SSD) –** это немеханическое запоминающее устройство на основе микросхем памяти
- **Время поиска блока на SSD (seek time) ≈ 0 сек**



Организация дисковой памяти

- Для сокращения времени работы с диском данные читают и записывают блоками – буферизованный ввод/вывод
- В алгоритмах для внешней памяти необходимо учитывать *количество обращений к диску*

Структуры данных для внешней памяти

- **Имеется бинарное дерево поиска (binary search tree), в котором каждый узел содержит:**
 - ☐ **ключ (key)** – строка из 32 символов (char[32], 32 байта)
 - ☐ **значение (value)** – целое число (int, 4 байта)
 - ☐ **указатели left и right** (по 8 байт)
- **Имеется сервер с 16 GiB оперативной памяти**
- **Сколько узлов дерева поместится в оперативную память?**

Структуры данных для внешней памяти

- Имеется бинарное дерево поиска (binary search tree), в котором каждый узел содержит:
 - ❑ **ключ** (key) – строка из 32 символов (char[32], 32 байта)
 - ❑ **значение** (value) – целое число (int, 4 байта)
 - ❑ **указатели** left и right (по 8 байт)
- Имеется сервер с 16 GiB оперативной памяти
- Сколько узлов дерева поместится в оперативную память?
- Узел дерева занимает $32 + 4 + 8 + 8 = 52$ байта
(не учитывая выравнивание)
- Пусть нам доступны все 16 GiB (оценка сверху):

$$16 * 2^{30} / 52 \approx \mathbf{330\ 382\ 099\ \text{узлов}}$$

Структуры данных для внешней памяти

- Как хранить словарь из **1 000 000 000** записей?
- Количество пользователей Facebook (лето 2014):
1 350 000 000
- Количество пользователей Google Gmail (май 2015):
900 000 000
- Количество пользователей ВКонтакте (февраль 2015):
219 000 000

**Решение: использовать внешнюю память —
HDD/SSD-диски, сетевые хранилища**

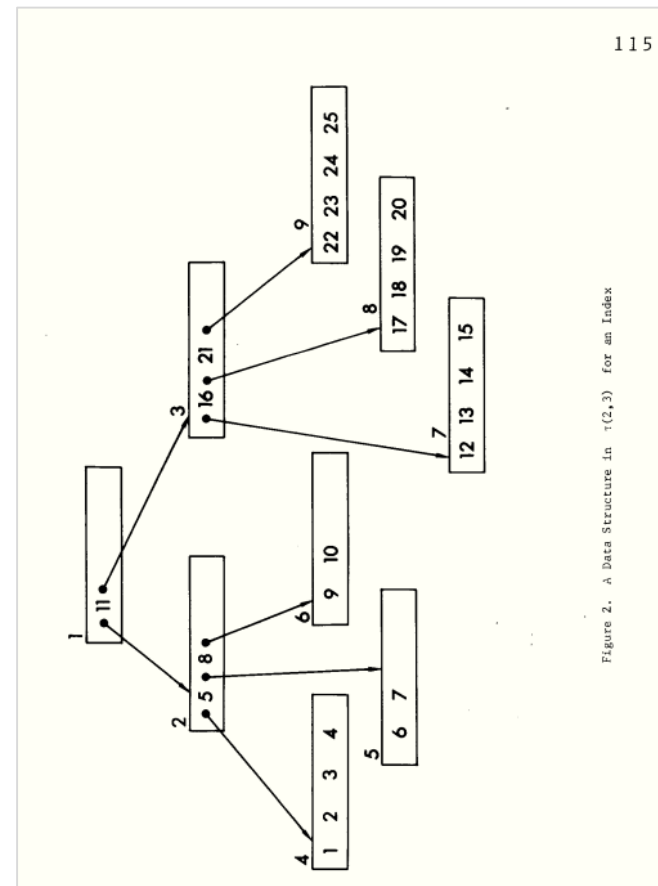
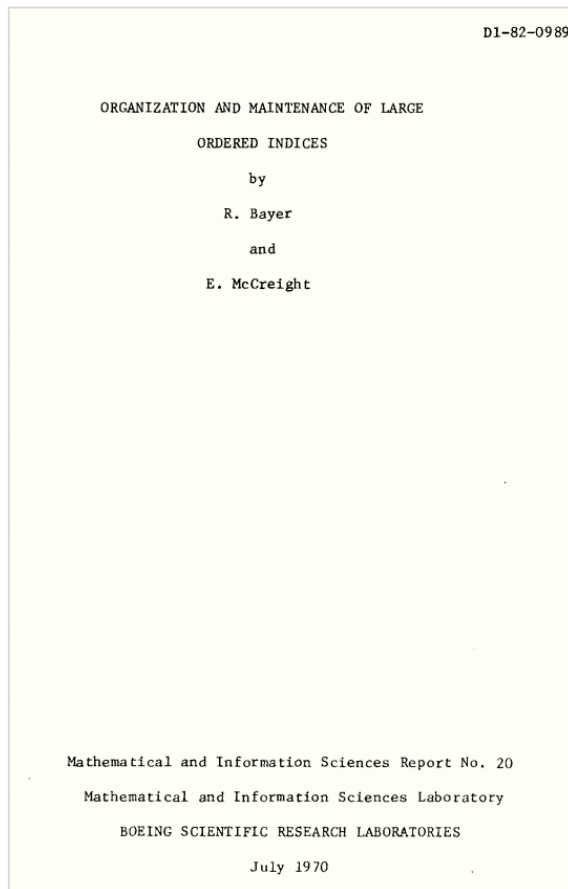
В-дерево (B-tree)

- **В-дерево (B-tree)** – это сбалансированное дерево поиска, узлы которого хранятся во внешней памяти
- В любой момент времени в оперативной памяти находится лишь часть В-дерева (размер дерева может значительно превышать объем оперативной памяти)
- В-деревья используются в файловых системах и системах управления базами данных (СУБД)
- **Авторы: Rudolf Bayer и Edward M. McCreight**
Boeing Research Labs, USA, 1971
- Буква «В» в названии В-деревьев: Boeing + Balanced + Bayer

*Ed McCreight answered a question on B-tree's name by Martin Farach-Colton saying (2013):
"Bayer and I were in a lunch time where we get to think a name. And we were, so, B, we were thinking... B is, you know... We were working for **Boeing** at the time, we couldn't use the name without talking to lawyers. So, there is a B. It has to do with **balance**, another B. **Bayer** was the senior author, who did have several years older than I am and had many more publications than I did. So there is another B. And so, at the lunch table we never did resolve whether there was one of those that made more sense than the rest. What really lives to say is: the more you think about what the B in B-trees means, the better you understand B-trees."*

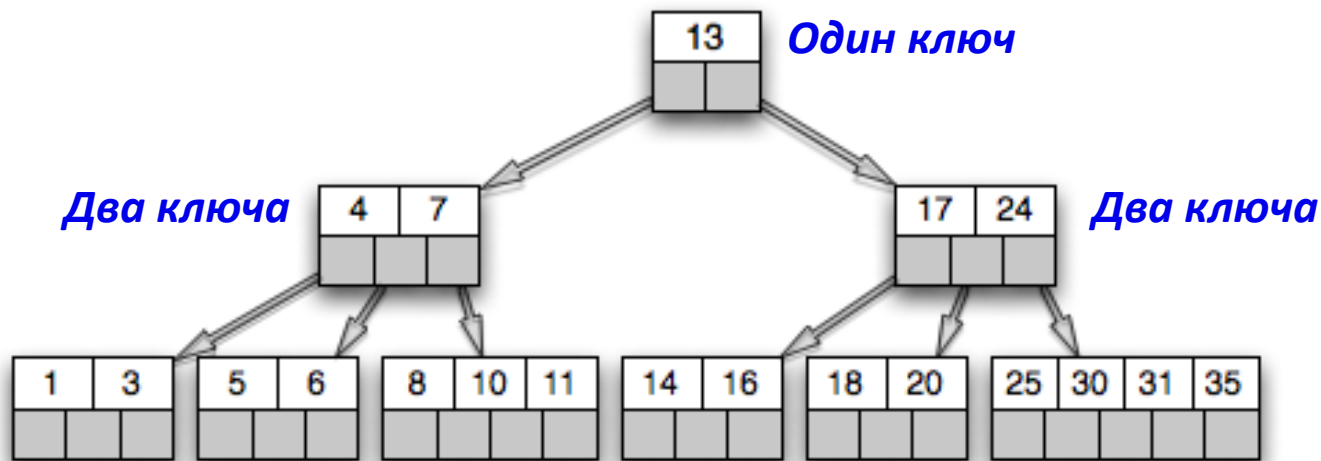
В-дерево (B-tree)

- Bayer R., McCreight E. **Organization and Maintenance of Large Ordered Indices** // Mathematical and Information Sciences Report No. 20, Boeing Scientific Research Laboratories, 1970.
- Bayer R. **Binary B-Trees for Virtual Memory** // Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, 1971. – pp. 219-235.

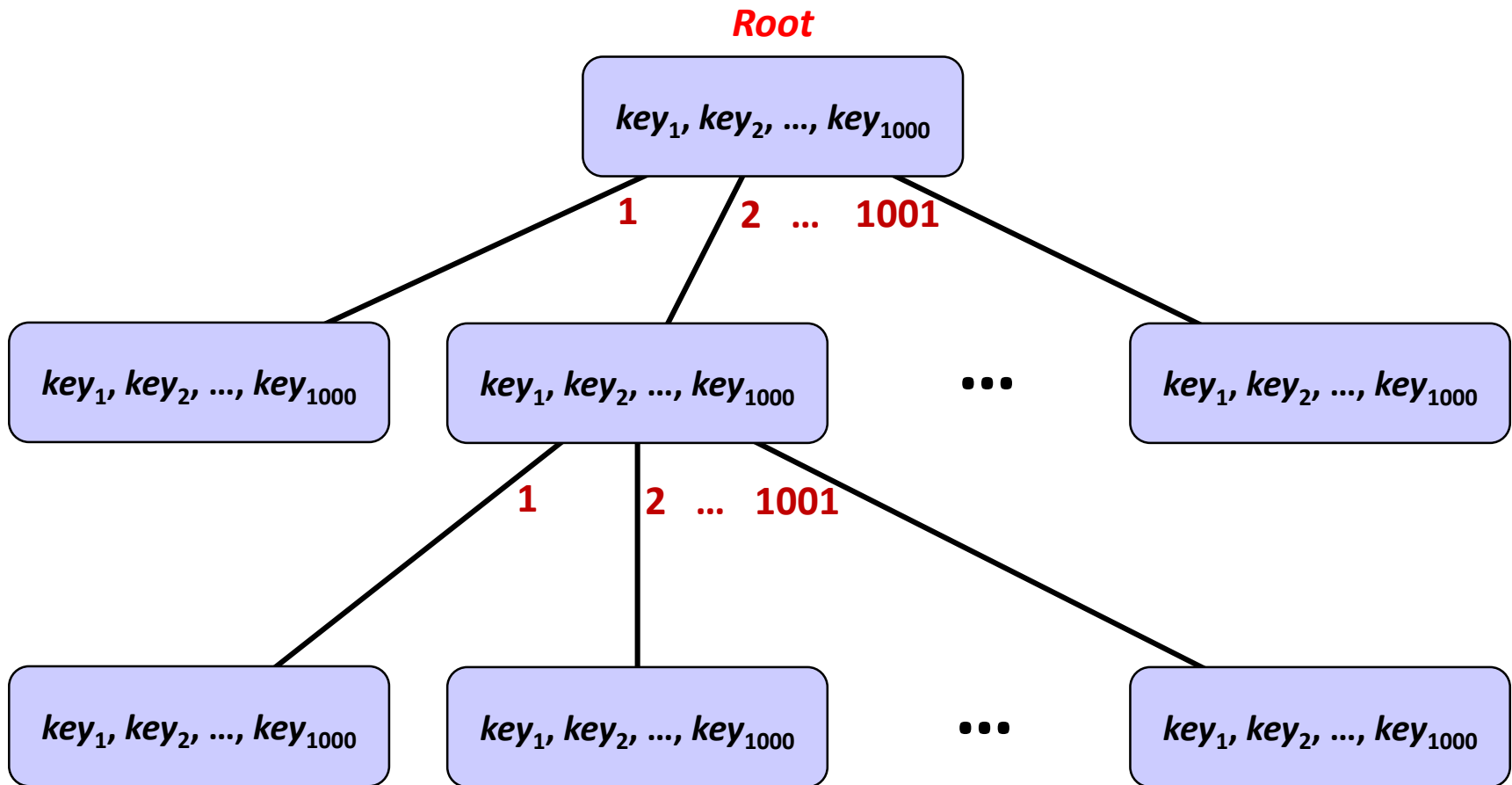


В-дерево (B-tree)

- Высота В-дерева не превышает $O(\log n)$, где n – количество узлов в дереве
- Каждый узел В-дерева может содержать *больше 1 ключа*
- Каждый узел В-дерева может иметь больше двух дочерних вершин (до тысяч в сильно ветвящихся деревьях)
- Если внутренний узел содержит k ключей, то у него $k + 1$ дочерних вершин



В-дерево (B-tree)



3 уровня

$$1 + 1001 + 1001 * 1001 = 1\ 003\ 003 \text{ узлов}$$

$$1000 + 1001 * 1000 + 1001 * 1001 * 1000 = 1\ 003\ 003\ 000 \text{ ключей}$$

В-дерево (B-tree)

- **В-дерево (B-tree)** – это корневое дерево, обладающее следующими свойствами:

1) Каждый узел содержит поля:

- количество n ключей, хранящихся в узле
- ключи $key_1, key_2, \dots, key_n$, упорядоченные по убыванию $key_1 < key_2 < \dots < key_n$
- флаг *leaf*, равный *true*, если узел является листом
- внутренний узел содержит $n + 1$ указателей c_1, c_2, \dots, c_{n+1} на дочерние узлы (листья не имеют дочерних узлов)

В-дерево (B-tree)

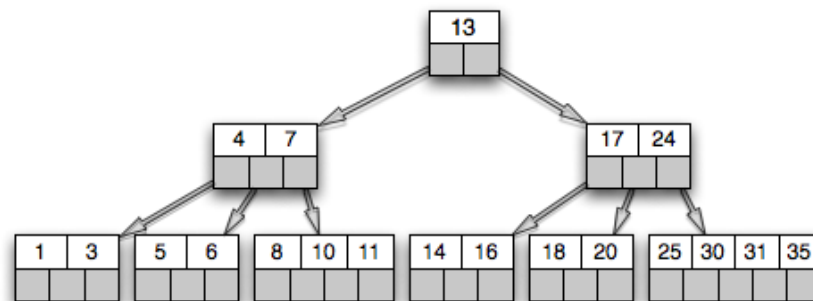
- **В-дерево (B-tree)** – это корневое дерево, обладающее следующими свойствами:

2) Ключи разделяют диапазоны ключей, хранящихся в поддеревьях:

если k_i – произвольный ключ в поддереве c_i , то

$$k_1 \leq \text{key}_1 \leq k_2 \leq \text{key}_2 \leq \dots \leq \text{key}_n \leq k_{n+1}$$

3) Все листья расположены на одинаковой глубине, равной высоте h дерева



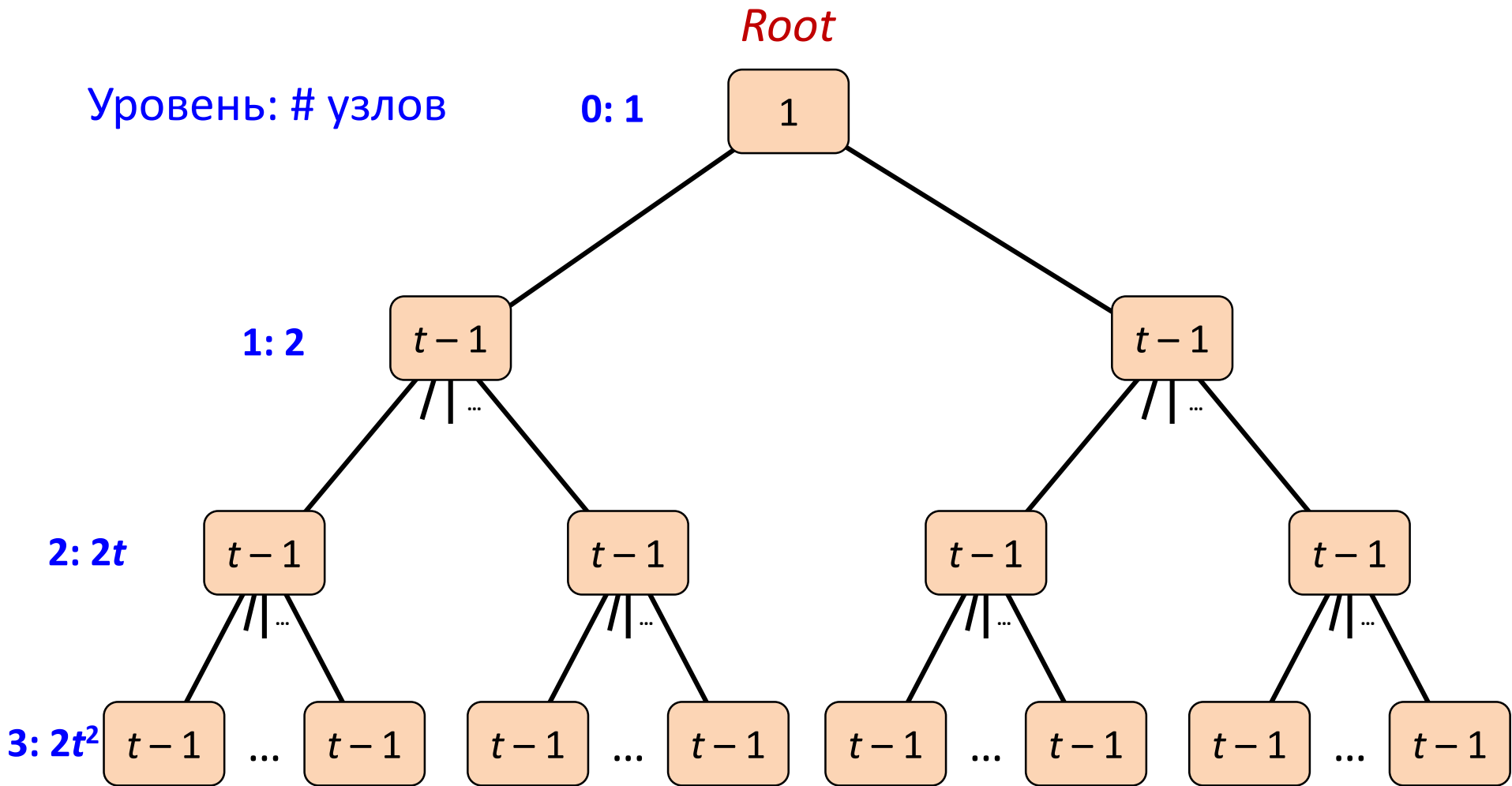
В-дерево (B-tree)

- **В-дерево (B-tree)** – это корневое дерево, обладающее следующими свойствами:

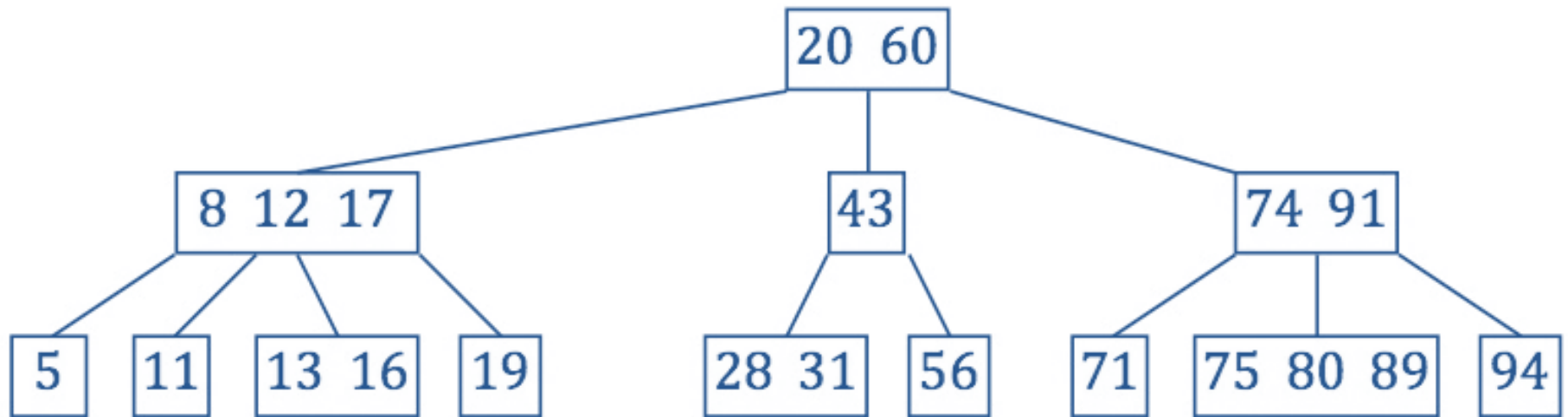
4) Имеются нижняя и верхняя границы количества ключей, хранящихся в узле – минимальная степень t (minimum degree) В-дерева

- корневой узел содержит от 1 до $2t - 1$ ключей
- каждый *внутренний* узел содержит минимум $t - 1$ ключей и минимум t дочерних узлов
- каждый узел содержит максимум $2t - 1$ ключей и максимум $2t$ дочерних узлов (за исключением листьев)
- узел *заполнен* (full), если он содержит $2t - 1$ ключей

В-дерево (B-tree)



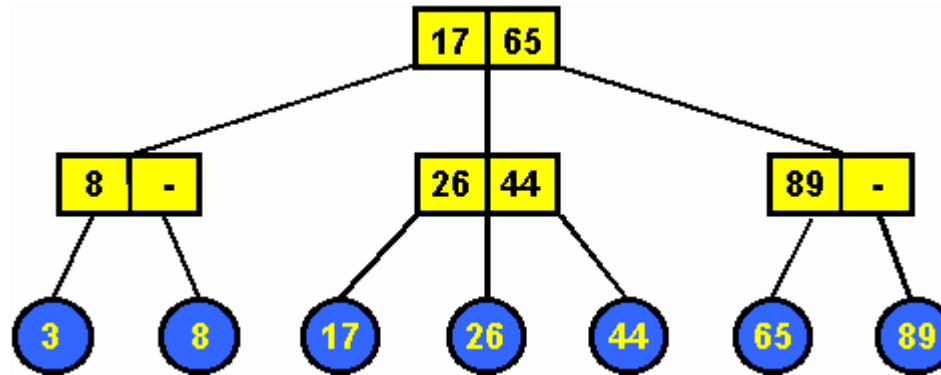
2-3-4-дерево (2-3-4 tree)



2-3-4-дерево (2-3-4 tree)

$t = 2$, каждый узел может иметь 2, 3 или 4 дочерних узла

2-3 дерево (2-3 tree)



http://www.cs.waikato.ac.nz/Teaching/COMP317B/Week_3/

Автор J. Е. Нормсroft, 1970
(работа не была опубликована)

В-деревья – обобщение 2-3 деревьев

Высота В-дерева

- **Утверждение.** *Высота В-дерева с $n \geq 1$ ключами и минимальной степенью $t \geq 2$ в худшем случае не превышает $\log_t((n + 1) / 2)$*
- **Доказательство**
- Обозначим высоту В-дерева через h .
- Рассмотрим максимально высокое В-дерево:
в корне такого дерева хранится 1 ключ, а в остальных узлах по $t - 1$ ключу (минимально возможное количество)
- На уровне 0 размещен один узел (корень) с 1 ключом
- На уровне 1: 2 узла (у каждого по $t - 1$ ключу)
- На уровне 2: $2t$ узлов
- На уровне 3: $2t^2$ узлов
- ...
- На уровне h : $2t^{h-1}$ узлов

Высота В-дерева

- **Утверждение.** *Высота В-дерева с $n \geq 1$ ключами и минимальной степенью $t \geq 2$ в худшем случае не превышает $\log_t((n + 1) / 2)$*

- **Доказательство (продолжение)**

- Тогда, общее количество ключей есть

$$\begin{aligned} n &= 1 + (t - 1)(2 + 2t + 2t^2 + 2t^3 + \dots + 2t^{h-1}) = \\ &= 1 + 2(t - 1)(1 + t + t^2 + \dots + t^{h-1}). \end{aligned}$$

- Сумма h первых членов геометрической прогрессии

$$S_h = \frac{b_1(q^h - 1)}{q - 1}$$

Высота В-дерева

- **Утверждение.** *Высота В-дерева с $n \geq 1$ ключами и минимальной степенью $t \geq 2$ в худшем случае не превышает $\log_t((n + 1) / 2)$*

- **Доказательство (продолжение)**

- Следовательно,

$$n = 1 + 2(t - 1) \frac{t^h - 1}{t - 1} = 2t^h - 1,$$

$$\frac{n+1}{2} = t^h,$$

$$h = \log_t \left(\frac{n+1}{2} \right).$$

- Утверждение доказано.

Операции с B-деревьями

- Во всех операциях предполагаем следующее:
 - Корень B-дерева всегда находится в оперативной памяти
 - Для чтения и записи данных на диск используются процедуры DiskRead и DiskWrite
- B-дерево минимизирует количество обращений к внешней памяти – минимум операций DiskRead, DiskWrite

Поиск элемента (Lookup)

- 1) Поиск начинаем с корня дерева. Для заданного ключа k среди ключей текущего узла

$$key_1 < key_2 < \dots < key_n$$

отыскиваем первый ключ key_i , такой что $k \leq key_i$

- 2) Если $k = key_i$, то прекращаем поиск (узел найден)
- 3) Иначе, загружаем с диска (DiskRead) дочерний узел c_i и рекурсивного обследуем его ключи
- 4) Если достигли листа (поле $leaf = true$), завершаем работу – ключ не найден

Поиск элемента (Lookup)

```
function BTreeLookup(node, key)
    i = 1
    while i < node.n AND key > node.key[i] do
        i = i + 1
    end while
    if i <= node.n AND key = node.key[i] then
        return (node, i)
    if node.leaf != TRUE then
        child = DiskRead(node.c[i])
        return BTreeLookup(child, key)
    end if
    return NULL
end function
```


Поиск элемента (Lookup)

```
function BTreeLookup(node, key)
  i = 1
  while i < node.n AND key > node.key[i] do
    i = i + 1
  end while
  if i <= node.n AND key = node.key[i] then
    return (node, i)
  if node.leaf != TRUE then
    child = DiskRead(node.c[i])
    return BTreeLookup(child, key)
  end if
  return NULL
end function
```

} $O(t)$

- Количество чтений с диска: $T_{Read} = O(h) = O(\log_t n)$
- Вычислительная сложность: $T = O(th) = O(t \log_t n)$

Поиск элемента (Lookup)

```
function BTreeLookup(node, key)
  i = 1
  while i < node.n AND key > node.key[i] do
    i = i + 1
  end while
```

} $O(t)$

Как ускорить поиск ключа в узле?

```
if node.key[i] < key then
  child = DiskRead(node.c[i])
  return BTreeLookup(child, key)
end if
return NULL
end function
```

- Количество чтений с диска: $T_{Read} = O(h) = O(\log_t n)$
- Вычислительная сложность: $T = O(th) = O(t \log_t n)$

Поиск элемента (Lookup)

```
function BTreeLookup(node, key)
```

```
  i = 1
```

```
  while i < node.n AND key > node.key[i] do
```

```
    i = i + 1
```

```
  end while
```

} $O(t)$

Как ускорить поиск ключа в узле?

Двоичный поиск (Binary search) за время $O(\log_2 t)$

$$T = O(\log_2 t \cdot \log_t n)$$

```
end function
```

- Количество чтений с диска: $T_{Read} = O(h) = O(\log_t n)$
- Вычислительная сложность: $T = O(th) = O(t \log_t n)$

Создание пустого B-дерева

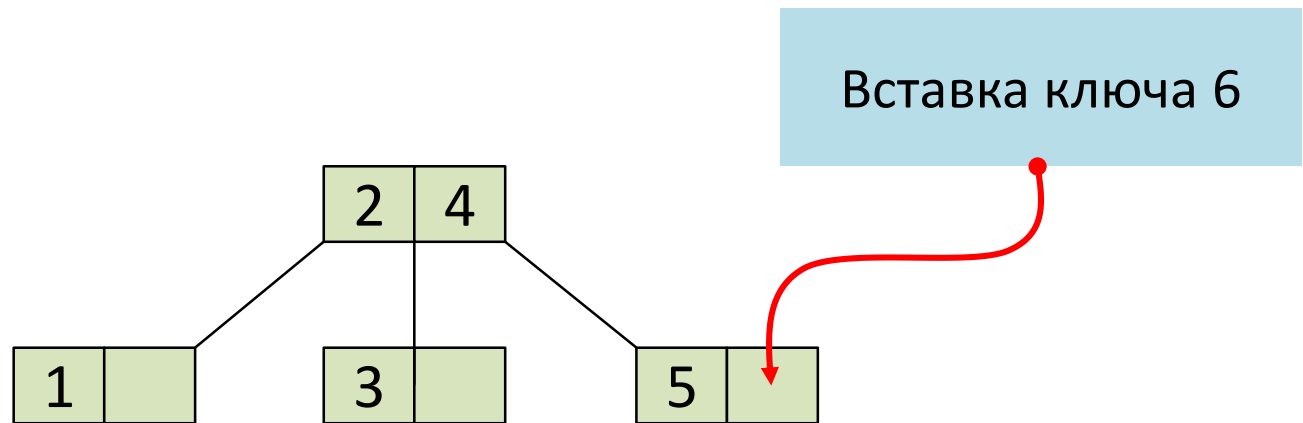
- 1) Выделить на диске место для корневого узла
- 2) Заполнить поля корневого узла

```
function BTreeCreate()  
    node = DiskAllocateNode()  
    node.leaf = TRUE  
    node.n = 0  
    DiskWrite(node)  
end function
```

- Количество дисковых операций: $T_{Disk} = O(1)$
- Вычислительная сложность: $T = O(1)$

Добавление ключа (Insert)

- 1) Для заданного ключа *key* **находим лист** для вставки нового элемента
- 2) Если **лист не заполнен** (ключей меньше или равно $2t - 1$), то вставляем ключ в узел (сохраняя упорядоченность ключей)

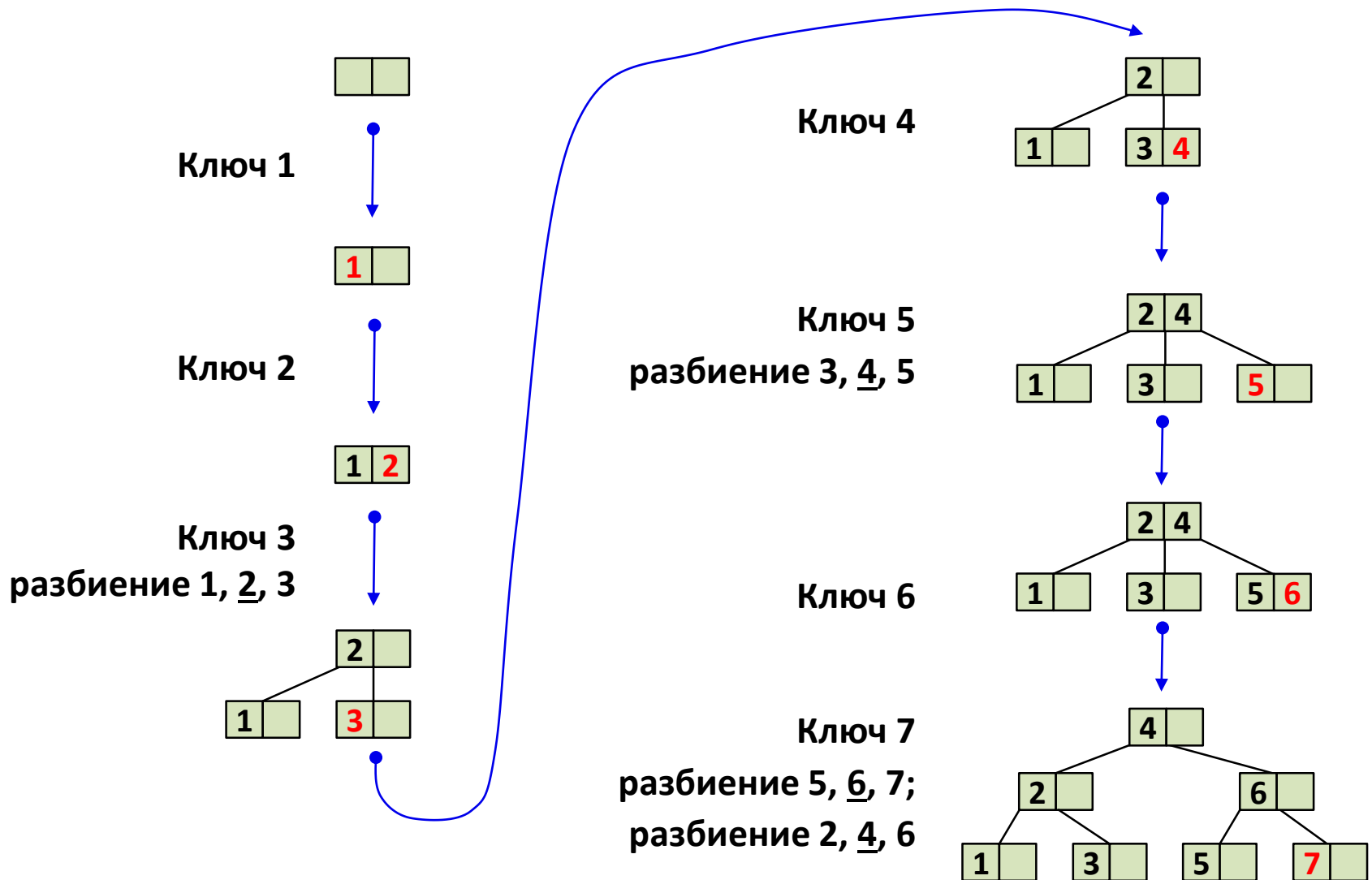


В-дерево

Добавление ключа (Insert)

- 3) Если **лист заполнен** (содержит $2t - 1$ ключей), разбиваем его (split) на 2 листа по $t - 1$ ключей: среди ключей листа $key_1 < key_2 < \dots < key_n$ и ключа key отыскиваем медиану (median key) – средний ключ разделитель
- 4) Ключ **медиана вставляется в родительский узел**.
Если родительский узел заполнен, то он разбивается и процедура повторяется по описанной выше схеме.
Подъем вверх по дереву может продолжаться до корня

Добавление ключа (Insert) в 2-3 B-дерево

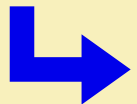


Добавление ключа (Insert)

- При проходе от корня дерева к искомому листу разбиваем (split) все заполненные узлы, через которые проходим (включая лист).
- Это гарантирует, что если понадобится разбить узел, то его родительский узел не будет заполнен

Разбиение узла (Split)

```
function BTreeSplitNode(node, parent, i)
    z = DiskAllocateNode()
    z.leaf = node.leaf
    z.n = t - 1
    /* Половина ключей перемещаются в новый узел */
    for j = 1 to t - 1 do
        z.key[j] = node.key[t + j]
    end for
    if node.leaf = FALSE then
        for j = 1 to t do
            z.c[j] = node.c[t + j]
        end for
    end if
end if
```



Разбиение узла (Split)

```
/* В node остается меньшая половина ключей */
node.n = t - 1
/* Вставляем ключ в родительский узел */
for j = parent.n + 1 downto i + 1 do
    parent.c[j + 1] = parent.c[j]
end for
parent.c[i + 1] = z
for j = parent.n downto i do
    parent.key[j + 1] = parent.key[j]
end for
parent.key[i] = node.key[t]
parent.n = parent.n + 1
DiskWrite(node)
DiskWrite(z)
DiskWrite(parent)
end function
```

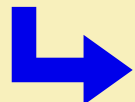
$$T_{SplitNode} = O(t)$$
$$T_{Disk} = O(1)$$

Добавление ключа (Insert)

```
function BTreeInsert(root, key)
  if root.n = 2t - 1 then
    newroot = DiskAllocateNode()
    newroot.leaf = FALSE
    newroot.n = 0
    newroot.c[1] = root
    BTreeSplitNode(root, newroot, 1)
    return BTreeInsertNonfull(newroot, key)
  end if
  return BTreeInsertNonfull(root, key)
end function
```

Добавление ключа (Insert)

```
function BTreeInsertNonfull(node, key)
    i = node.n
    if node.leaf = TRUE then
        while i > 1 AND key < node.key[i] do
            node.key[i + 1] = node.key[i]
            i = i - 1
        end while
        node.key[i + 1] = key
        node.n = node.n + 1
        DiskWrite(node)
    else
        while i > 1 AND key < node.key[i] do
            i = i - 1
        end while
        i = i + 1
        DiskRead(node.c[i])
```



Добавление ключа (Insert)

```
    if node.c[i].n = 2t - 1 then
        BTreeSplitNode(node.c[i], node, i)
        if key > node.key[i] then
            i = i + 1
        end if
    end if
    node = BTreeInsertNonfull(node.c[i], key)
end if
return node
end function
```

- Вычислительная сложность вставки ключа в B-деревов худшем случае (разбиваем узлы на каждом уровне):

$$T = O(th) = O(t \log_t n)$$

- Количество дисковых операций:

$$T_{Disk} = O(h) = O(\log_t n)$$

Удаление ключа (Delete)

- Находим узел, содержащий искомый ключ *key*
- Если ключ *key* находится в листе, то удаляем ключ из него
- Если количество ключей стало меньше $t - 1$, выполняем восстановление свойств B-дерева
- ...

Изучить самостоятельно [CLRS 3ed, C. 536]

Виды В-деревьев

- **В⁺-дерево (B⁺-tree)** – это В-дерево, в котором только листья содержат информацию, а внутренние узлы хранят только ключи
 - ❑ индексация метаданных в файловых системах Btrfs, NTFS, ReiserFS, NSS, XFS, JFS
 - ❑ индексы таблиц в СУБД IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASE, SQLite

Одно из первых упоминаний В⁺-деревьев:

[Douglas Comer. *The Ubiquitous B-Tree* // ACM Computing Surveys 11\(2\), 1979. – pp. 121–137](#)

- **В^{*}-дерево (B^{*}-tree)** – это В-дерево, в котором каждый узел (за исключением корня) должен содержать не менее $2/3$ ключей (а не $1/2$ как в В-дереве)

Реализация 2-3-4 B-tree

```
/* Minimum degree of B-tree */
#define T 2                      /* 2-3-4 B-tree */

struct btree {
    int leaf;
    int nkeys;
    int *key;
    int *value;
    struct btree **child;
};
```


Реализация B-tree

```
struct btree *btree_create()
{
    struct btree *node;

    node = malloc(sizeof(*node));
    node->leaf = 1;
    node->nkeys = 0;
    node->key = malloc(sizeof(*node->key) * 2 * T - 1);
    node->value = malloc(sizeof(*node->value) * 2 * T - 1);
    node->child = malloc(sizeof(*node->child) * 2 * T);
    return node;
}
```

Реализация B-tree

```
void btree_lookup(struct btree *tree, int key,
                  struct btree **node, int *index)
{
    int i;

    for (i = 0; i < tree->nkeys && key > tree->key[i]; ) {
        i++;
    }
    if (i < tree->nkeys && key == tree->key[i]) {
        *node = tree;
        *index = i;
        return;
    }
    if (!tree->leaf) {
        /* Disk read tree->child[i] */
        btree_lookup(tree, key, node, index);
    } else {
        *node = NULL;
    }
}
```

Реализация B-tree

```
struct btree *btree_insert(struct btree *tree,
                           int key, int value)
{
    struct btree *newroot;

    if (tree == NULL) {
        tree = btree_create();
        tree->nkeys = 1;
        tree->key[0] = key;
        tree->value[0] = value;
        return tree;
    }
    if (tree->nkeys == 2 * T - 1) {
        newroot = btree_create(); /* Create empty root */
        newroot->leaf = 0;
        newroot->child[0] = tree;
        btree_split_node(tree, newroot, 0);
        return btree_insert_nonfull(newroot, key, value);
    }
    return btree_insert_nonfull(tree, key, value);
}
```

Реализация B-tree

```
void btree_split_node(struct btree *node,
                     struct btree *parent, int index)
{
    struct btree *z;
    int i;

    z = btree_create();
    z->leaf = node->leaf;
    z->nkeys = T - 1;
    for (i = 0; i < T - 1; i++) {
        z->key[i] = node->key[T + i];
        z->value[i] = node->value[T + i];
    }
    if (!node->leaf) {
        for (i = 0; i < T; i++)
            z->child[i] = node->child[i + T];
    }
    node->nkeys = T - 1;
```

Реализация B-tree

```
/* Insert median key into parent node */
for (i = parent->nkeys; i >= 0 && i <= index + 1; i--)
    parent->child[i + 1] = parent->child[i];
parent->child[index + 1] = z;
for (i = parent->nkeys - 1; i >= 0 && i <= index; i--) {
    parent->key[i + 1] = parent->key[i];
    parent->value[i + 1] = parent->value[i];
}
parent->key[index] = node->key[T - 1];
parent->value[index] = node->value[T - 1];
parent->nkeys++;
/* Write to disk: node, z, parent */
}
```

Реализация B-tree

```
struct btree *btree_insert_nonfull(  
    struct btree *node, int key, int value)  
{  
    int i;  
  
    i = node->nkeys;  
    if (node->leaf) {  
        for (i = node->nkeys - 1; i > 0 &&  
            key < node->key[i]; i--)  
        {  
            node->key[i + 1] = node->key[i];  
        }  
        node->key[i + 1] = key;  
        node->nkeys++;  
    } else {
```

Реализация B-tree

```
    for (i = node->nkeys - 1; i > 0 &&
         key < node->key[i]; )
    {
        i--;
    }
    i++;
    if (node->child[i]->nkeys == 2 * T - 1) {
        btree_split_node(node->child[i], node, i);
        if (key > node->key[i])
            i++;
    }
    node = btree_insert_nonfull(node->child[i],
                                key, value);
}
return node;
}
```

Реализация B-tree

```
int main()
{
    struct btree *tree;

    tree = btree_insert(NULL, 3, 0);
    tree = btree_insert(tree, 12, 0);
    tree = btree_insert(tree, 9, 0);
    tree = btree_insert(tree, 18, 0);

    return 0;
}
```


Внешняя сортировка слиянием

- **Внешняя сортировка (External sorting)** – это класс алгоритмов сортировки, которые оперируют данными размещенными на внешней памяти
- Как отсортировать 300 GiB данных на жестком диске имея 2 GiB оперативной памяти?

Внешняя сортировка слиянием

- **Внешняя сортировка (External sorting)** – это класс алгоритмов сортировки, которые оперируют данными размещенными на внешней памяти
- Как отсортировать 300 GiB данных на жестком диске имея 2 GiB оперативной памяти?
 1. Загружаем блок размером 2 GiB в оперативную память
 2. Сортируем блок (MergeSort, QuickSort, CountingSort, ...) и сохраняем на внешнюю память (диск)
 3. Повторяем шаги 1 и 2, пока не получим $300 / 2 = 150$ отсортированных блоков на внешней памяти
 4. Создаем в оперативной памяти 151 буфер по 13 KiB ($2 \text{ GiB} / 151 \approx 13 \text{ KiB}$)
 5. Загружаем в 150 буферов по 13 KiB из отсортированных блоков, 151-й буфер – это выходной блок
 6. В выходной блок сливаем (merge) данные из 150 буферов и записываем его на внешнюю память
 7. Повторяем шаги 5 и 6 пока не сольем все 150 блоков

Внешняя сортировка слиянием

k-way merge sort

$$k = 300 \text{ GiB} / 2 \text{ GiB} = 150$$

150-way merge sort

1. Загружаем блок размером 2 GiB в оперативную память
2. Сортируем блок (MergeSort, QuickSort, CountingSort, ...) и сохраняем на внешнюю память (диск)
3. Повторяем шаги 1 и 2, пока не получим $300 / 2 = 150$ отсортированных блоков на внешней памяти
4. Создаем в оперативной памяти 151 буфер по 13 KiB ($2 \text{ GiB} / 151 \approx 13 \text{ KiB}$)
5. Загружаем в 150 буферов по 13 KiB из отсортированных блоков, 151-й буфер – это выходной блок
6. В выходной блок сливаем (merge) данные из 150 буферов и записываем его на внешнюю память
7. Повторяем шаги 5 и 6 пока не сольем все 150 блоков

Задания

- Рассмотреть применение В-деревьев в файловых системах – что содержит поле «ключ» (key) и поле «значение» (value)
- Привести пример значения t , используемого в файловой системе Btrfs
- Изучить применение В-деревьев в системах управления базами данных (MySQL, PostgreSQL, Microsoft SQL Server, IBM DB2 и др.): какие задачи они решают, что хранится в поле «ключ» (key) и поле «значение» (value)