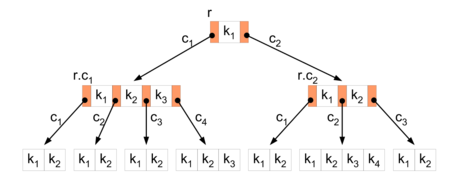
**1. B-дерево** — структура данных, дерево поиска. С точки зрения внешнего логического представления, сбалансированное, сильно ветвистое дерево.

Использование B-деревьев впервые было предложено Р. Бэйером и Э. МакКрейтом в 1970 году.

* *Сбалансированность* означает, что длина любых двух путей от корня до листьев различается не более, чем на единицу.
* *Ветвистость дерева* — это свойство каждого узла дерева ссылаться на большое число узлов-потомков.

С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц памяти, то есть каждому узлу дерева соответствует блок памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.

Пример b-дерева со степенью 3



**2.** 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)**.

**h ≤ logt((n+1)/2)** — логарифм по основанию **t**.

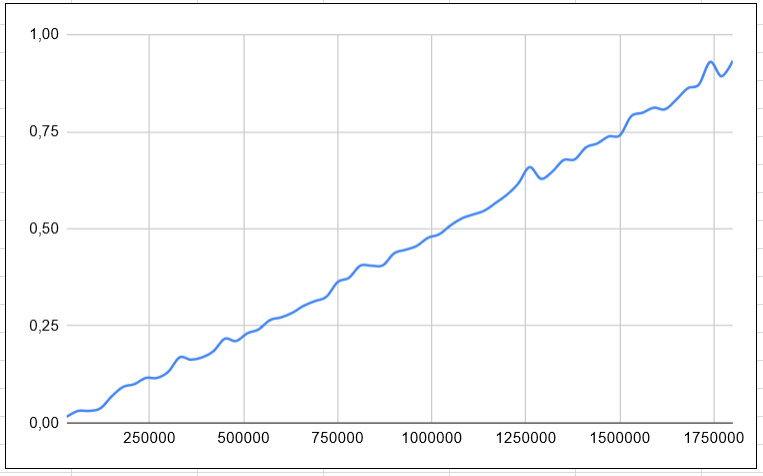
**3.**

* 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)**!
* Операция добавления происходит также за время **O(t logt n)**. Опять же, дисковых операций мы выполняем всего лишь **O(h)**, где h – высота дерева.
* Операция удаления происходит за такое же время, **O(t logt n)**. Да и дисковых операций требуется всего лишь **O(h)**, где **h** – высота дерева.

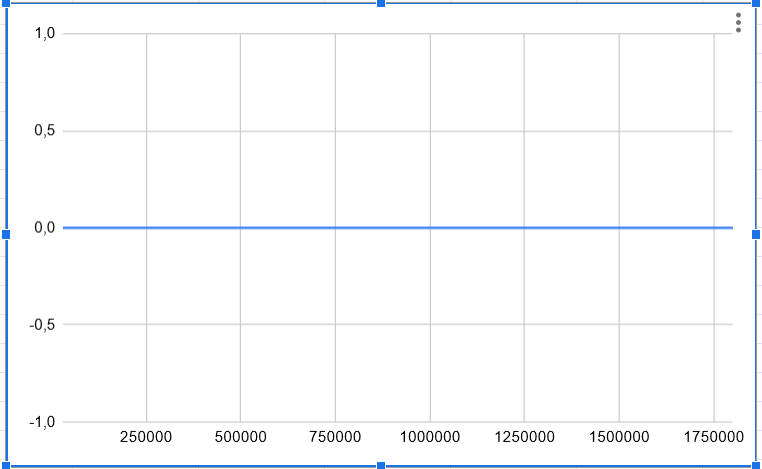
**4.** см. Приложение

**5.** Графики:

* Вставка:



* Поиск:



**6.** Выводы:

Итак, мы убедились в том, что B-дерево является быстрой структурой данных. И еще одно важное свойство – автоматическое поддержание свойства сбалансированности.

Проанализировав, вместе со скоростью выполнения, количество проведенных операций с дисковой памятью, мы можем сказать, что B-дерево несомненно является более выгодной структурой данных для случаев, когда мы имеем большой объем информации: пусть у нас есть миллиард ключей, и t=1001. Тогда нам потребуется всего лишь 3 дисковые операции для поиска любого ключа! При этом учитываем, что корень мы можем хранить постоянно. Теперь видно, на сколько это мало!

Также, мы читаем не отдельные данные с разных мест, а целыми блоками. Перемещая узел дерева в оперативную память, мы перемещаем выделенный блок последовательной памяти, поэтому эта операция достаточно быстро работает.

Соответственно, мы имеем минимальную нагрузку на сервер, и при этом малое время ожидания. Эти и другие описанные преимущества позволили B-деревьям стать основой для индексов, базирующихся на деревьях в СУБД.

**7.** Литература:

* <https://habr.com/ru/post/114154/>
* <https://google.com>
* <https://ru.wikipedia.org/wiki/B-дерево>

**8.**

class BTreeNode {  
 int \*keys;  
 int t;  
 BTreeNode \*\*C;  
 int n;  
 bool leaf;  
  
public:  
 BTreeNode(int \_t, bool \_leaf);  
  
 void traverse();  
  
 BTreeNode \*search(int k);  
  
 int findKey(int k);  
  
 void insertNonFull(int k);  
  
 void splitChild(int i, BTreeNode \*y);  
  
 void remove(int k);  
  
 void removeFromLeaf(int idx);  
  
 void removeFromNonLeaf(int idx);  
  
 int getPred(int idx);  
  
 int getSucc(int idx);  
  
 void fill(int idx);  
  
 void borrowFromPrev(int idx);  
  
 void borrowFromNext(int idx);  
  
 void merge(int idx);  
  
 friend class BTree;  
};  
  
class BTree {  
 BTreeNode \*root;  
 int t;  
public:  
  
 BTree(int \_t) {  
 root = nullptr;  
 t = \_t;  
 }  
  
 void traverse() {  
 if (root != nullptr) root->traverse();  
 }  
  
 BTreeNode \*search(int k) {  
 if (root == nullptr) {  
 return nullptr;  
 } else {  
 return root->search(k);  
 }  
 }  
  
 void insert(int k);  
  
 void remove(int k);  
};  
  
BTreeNode::BTreeNode(int t1, bool leaf1) {  
 t = t1;  
 leaf = leaf1;  
  
 keys = new int[2 \* t - 1];  
 C = new BTreeNode \*[2 \* t];  
  
 n = 0;  
}  
  
int BTreeNode::findKey(int k) {  
 int idx = 0;  
 while (idx < n && keys[idx] < k)  
 ++idx;  
 return idx;  
}  
  
void BTreeNode::remove(int k) {  
 int idx = findKey(k);  
  
 if (idx < n && keys[idx] == k) {  
 if (leaf)  
 removeFromLeaf(idx);  
 else  
 removeFromNonLeaf(idx);  
 } else {  
 if (leaf) {  
 cout << "key " << k << " not found\n";  
 return;  
 }  
  
 bool flag = (idx == n);  
  
 if (C[idx]->n < t)  
 fill(idx);  
 if (flag && idx > n)  
 C[idx - 1]->remove(k);  
 else  
 C[idx]->remove(k);  
 }  
}  
  
void BTreeNode::removeFromLeaf(int idx) {  
 for (int i = idx + 1; i < n; ++i)  
 keys[i - 1] = keys[i];  
  
 n--;  
}  
  
void BTreeNode::removeFromNonLeaf(int idx) {  
  
 int k = keys[idx];  
 if (C[idx]->n >= t) {  
 int pred = getPred(idx);  
 keys[idx] = pred;  
 C[idx]->remove(pred);  
 } else if (C[idx + 1]->n >= t) {  
 int succ = getSucc(idx);  
 keys[idx] = succ;  
 C[idx + 1]->remove(succ);  
 } else {  
 merge(idx);  
 C[idx]->remove(k);  
 }  
}  
  
int BTreeNode::getPred(int idx) {  
 BTreeNode \*cur = C[idx];  
 while (!cur->leaf)  
 cur = cur->C[cur->n];  
  
 return cur->keys[cur->n - 1];  
}  
  
int BTreeNode::getSucc(int idx) {  
 BTreeNode \*cur = C[idx + 1];  
 while (!cur->leaf)  
 cur = cur->C[0];  
  
 return cur->keys[0];  
}  
  
void BTreeNode::fill(int idx) {  
 if (idx != 0 && C[idx - 1]->n >= t)  
 borrowFromPrev(idx);  
  
 else if (idx != n && C[idx + 1]->n >= t)  
 borrowFromNext(idx);  
  
 else {  
 if (idx != n)  
 merge(idx);  
 else  
 merge(idx - 1);  
 }  
}  
  
void BTreeNode::borrowFromPrev(int idx) {  
  
 BTreeNode \*child = C[idx];  
 BTreeNode \*sibling = C[idx - 1];  
  
 for (int i = child->n - 1; i >= 0; --i)  
 child->keys[i + 1] = child->keys[i];  
  
 if (!child->leaf) {  
 for (int i = child->n; i >= 0; --i)  
 child->C[i + 1] = child->C[i];  
 }  
  
 child->keys[0] = keys[idx - 1];  
  
 if (!child->leaf)  
 child->C[0] = sibling->C[sibling->n];  
  
 keys[idx - 1] = sibling->keys[sibling->n - 1];  
  
 child->n += 1;  
 sibling->n -= 1;  
}  
  
void BTreeNode::borrowFromNext(int idx) {  
  
 BTreeNode \*child = C[idx];  
 BTreeNode \*sibling = C[idx + 1];  
  
 child->keys[(child->n)] = keys[idx];  
  
 if (!(child->leaf))  
 child->C[(child->n) + 1] = sibling->C[0];  
  
 keys[idx] = sibling->keys[0];  
  
 for (int i = 1; i < sibling->n; ++i)  
 sibling->keys[i - 1] = sibling->keys[i];  
  
 if (!sibling->leaf) {  
 for (int i = 1; i <= sibling->n; ++i)  
 sibling->C[i - 1] = sibling->C[i];  
 }  
  
 child->n += 1;  
 sibling->n -= 1;  
}  
  
void BTreeNode::merge(int idx) {  
 BTreeNode \*child = C[idx];  
 BTreeNode \*sibling = C[idx + 1];  
  
 child->keys[t - 1] = keys[idx];  
  
 for (int i = 0; i < sibling->n; ++i)  
 child->keys[i + t] = sibling->keys[i];  
  
 if (!child->leaf) {  
 for (int i = 0; i <= sibling->n; ++i)  
 child->C[i + t] = sibling->C[i];  
 }  
  
 for (int i = idx + 1; i < n; ++i)  
 keys[i - 1] = keys[i];  
  
 for (int i = idx + 2; i <= n; ++i)  
 C[i - 1] = C[i];  
  
 child->n += sibling->n + 1;  
 n--;  
  
 delete (sibling);  
}  
  
void BTree::insert(int k) {  
 if (root == nullptr) {  
  
 root = new BTreeNode(t, true);  
 root->keys[0] = k;  
 root->n = 1;  
 } else {  
 if (root->n == 2 \* t - 1) {  
  
 BTreeNode \*s = new BTreeNode(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 BTreeNode::insertNonFull(int k) {  
 int i = n - 1;  
  
 if (leaf) {  
 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 BTreeNode::splitChild(int i, BTreeNode \*y) {  
 BTreeNode \*z = new BTreeNode(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) {  
 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;  
}  
  
void BTreeNode::traverse() {  
 int i;  
 for (i = 0; i < n; i++) {  
 if (!leaf)  
 C[i]->traverse();  
 cout << " " << keys[i];  
 }  
  
 if (!leaf)  
 C[i]->traverse();  
}  
  
BTreeNode \*BTreeNode::search(int k) {  
 int i = 0;  
 while (i < n && k > keys[i])  
 i++;  
  
 if (keys[i] == k)  
 return this;  
  
 if (leaf)  
 return nullptr;  
  
 return C[i]->search(k);  
}  
  
void BTree::remove(int k) {  
 if (!root) {  
 cout << "tree is empty\n";  
 return;  
 }  
  
 root->remove(k);  
  
 if (root->n == 0) {  
 BTreeNode \*tmp = root;  
 if (root->leaf)  
 root = nullptr;  
 else  
 root = root->C[0];  
  
 delete tmp;  
 }  
}

}

**9.** Приложение

Таблица для поиска: <https://docs.google.com/spreadsheets/d/1s_x3irUqwroNOlDE5rQn5z7wFX-2AgnG9jjIoBPdKwU/edit?usp=sharing>

Таблица для вставки: <https://docs.google.com/spreadsheets/d/1f3GkOUaVCBn-4X5aI6-Eu4RhTw7X7qAs8_QThFHq3yQ/edit?usp=sharing>

Репозиторий с данными для тестирования: <https://github.com/AvanNorth/pskda_iip>