УДК 004.6 DOI: 10.31799/2077-5687-2025-2-47-54

ПРИМЕР РЕАЛИЗАЦИИ ЛОКАЛЬНОЙ БАЗЫ ДАННЫХ, ОСНОВАННОЙ НА B+ ДЕРЕВЬЯХ

А. В. Шахомиров, Н. Е. Шахомирова
Санкт-Петербургский государственный университет аэрокосмического приборостроения

В работе рассмотрена реализация локальной базы данных на основе B+ деревьев – эффективной структуры данных для хранения и быстрого поиска информации. Показано ключевое отличие B-деревьев от B+деревьев, где данные хранятся только в листьях, а внутренние узлы содержат ключи для навигации, что обеспечивает сбалансированность дерева, минимизирует количество операций ввода-вывода и ускоряет поиск, вставку и удаление элементов дерева. На примере реализации на языке программирования высокого уровня C++, рассмотрена структура B+дерева, включая метаданные, а также алгоритмы основных операций: поиска, вставки, удаления и обновления. Показаны механизмы разделения и слияния узлов при перебалансировке дерева.

Ключевые слова: базы данных, B+ деревья, сбалансированные деревья, пример реализации.

Для цитирования:
Шахомиров, А. В. Пример реализации локальной базы данных, основанной на B+ деревьях / А. В. Шахомиров, Н. Е. Шахомирова // Системный анализ и логистика. – 2025. – № 2(45). – с. 47-54. DOI: 10.31799/2077-5687-2025-2-47-54.

Введение

Системы хранения и обработки данных требуют эффективных структур для быстрого поиска, вставки и удаления информации. Одними из наиболее популярных структур данных, обеспечивающих высокую производительность, являются B-деревья [1]. В отличие от бинарных деревьев поиска, B-деревья имеют переменное количество ключей и потомков в узлах, что позволяет эффективно работать с блочно-ориентированными хранилищами. Особый интерес представляет B+ дерево, в котором данные хранятся только в листьях, а внутренние узлы содержат ключи для навигации. Такой подход обеспечивает хорошую производительность при последовательном доступе и часто используется в файловых системах и базах данных [2]. В данной статье рассматриваются: основные свойства и структура деревьев; алгоритмы поиска, вставки, удаления и обновления данных; пример практической реализации B+ дерева.

Что представляет собой B-дерево

B-дерево – это сбалансированное дерево поиска, в котором каждый узел содержит множество ключей и имеет более двух потомков [2, 3]. Количество ключей в узле и количество его потомков зависит от порядка B-дерева.

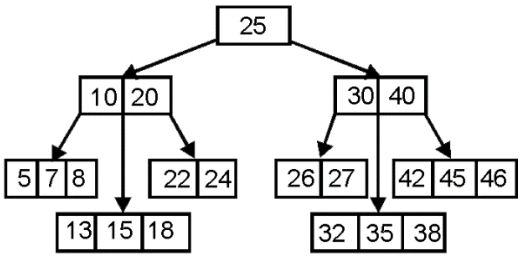

Рис. 1. Пример B-дерева.

B-дерево порядка *m* обладает следующими свойствами:

1) Глубина всех листьев одинакова.
2) Все узлы, кроме корня должны иметь как минимум *(m/2)-1* ключей и максимум *m-1* ключей.
3) Все узлы без листьев, кроме корня (все внутренние узлы), должны иметь минимум *m/2* потомков.
4) Если корень – это узел, не содержащий листьев, он должен иметь минимум 2 потомка.
5) Узел без листьев с n-1 ключами должен иметь n потомков.
6) Все ключи в узле должны располагаться в порядке возрастания их значений [4].

Что представляет собой B+ дерево

B+ дерево [5, 6] – это *m*-арное дерево с переменным, но часто большим числом потомков на узел. B+ дерево состоит из корня, внутренних узлов и листьев. Его рассматривают как разновидность B-дерева, в которой только конечные узлы (листья) содержат фактические значения ключей, промежуточные узлы B+-деревьев содержат значения указателя, которые используются только для управления поиском в дереве. В классических B-деревьях значения ключей хранятся как в конечных, так и в промежуточных узлах дерева.

B+ дерево порядка m обладает следующими свойствами:

1) B+ дерево является самобалансирующимся, что позволяет сохранить низкую высоту дерева, и уменьшить количество операций ввода-вывода.
2) Фактические данные содержатся только в конечных узлах нижнего уровня.

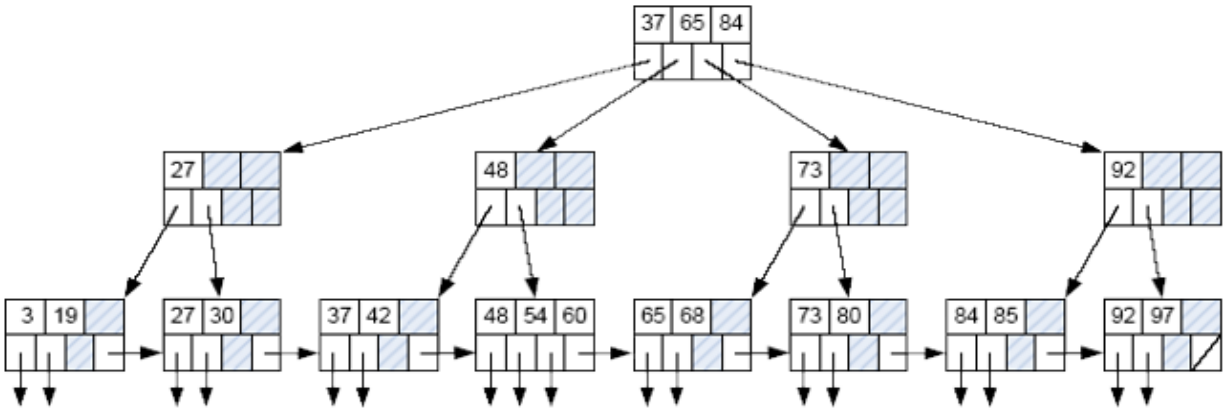

Рис. 2. Пример B+ дерева.

Основное применение B+ деревьев – хранение данных для эффективного поиска в блочно-ориентированных файловых системах. Например, файловая система ext4, широко используемая в операционных системах семейства Linux, хранит метаданные (inodes) в отдельных блоках, связанных структурой B+ дерева, что обеспечивает быструю навигацию и обновление файлов [5, 6, 7, 8]. Так же широкое применение B+ деревья нашли в структурах хранения в базах данных [7, 8].

Структура дерева

Чтобы упростить управление данными, операции с базой данных будем осуществлять с использованием самобалансирующегося B+ дерева [3].

Создадим структуру для хранения метаданных B+ дерева [3, 6, 7]:

```
void BPlusTree::initEmpty(){
       // инициализация метаданных. По умолчанию высота дерева равна 1.
       memset(&meta, 0, sizeof(meta_t));
       meta.order = BPLUS_ORDER;
       meta.value_size = sizeof(value_t);
       meta.key_size = sizeof(key_t);
       meta.height = 1;
       // инициализация корня дерева
       internal_t root;
       root.parent = 0;
       root.prev = root.next = 0;
       meta.root_offset = alloc(&root);
       // инициализация листьев
       leaf_t leaf;
       leaf.next = leaf.prev = 0;
       leaf.parent = meta.root_offset;
       root.children[0].child = alloc(&leaf);
       meta.leaf_offset = root.children[0].child;
}
```

Будем хранить данные в представлении «ключ-значение» [3, 6, 7]. Определим тип данных value_t для значения типа char name[256] (в общем случае value_t может представлять собой специальный тип, обозначающий числовое значение, строку, битовую маску или другое специфичные значения), int value (целое значение, введенное пользователем) и char data[256].

Структура meta_t будет хранить метаданные о базе данных: order (порядок), height (высота), key_size, value_size, internal_num (количество внутренних узлов), leaf_num (количество концевых вершин), root_offset, leaf_offset.

Структура index_t будет хранить индексы, содержащие ключ и дочерний элемент.

Структура internal_t будет хранить внутренние узлы, кроме корневых и конечных.

Структура leaf_t будет хранить конечные узлы, содержит parent (указательна родительский узел), next (указатель на следующий узел), prev (указатель на предыдущий узел), num (количество дочерних узлов), children[BPLUS_ORDER].

Структура record_t будет хранить записи в формате ключ-значение.

Выборка

Операции поиска значений в дереве реализуются как поиск по значению ключевого атрибута (индекса), поиск по значению в листе, поиск в диапазоне значений [3, 6, 7, 8].

```
// поиск по индексу
off_t BPlusTree::searchIndex(const key_t& key) const{
    off_t index = meta.root_offset;
    int height = meta.height;
    while(height > 1){
        internal_t node;
        map(&node, index);
        index_t *i = std::upper_bound(begin(node), end(node)-1, key);
        index = i -> child;
        height--;
    }
    return index;
}
// поиск листа
off_t BPlusTree::searchLeaf(off_t index, const key_t& key) const{
    internal_t node;
```

```
    map(&node, index);
    index_t *i = std::upper_bound(begin(node), end(node)-1, key);
    return i -> child;
}
//поиск в диапазоне
int BPlusTree::searchRange(key_t *left, const key_t& right, value_t *values, size_t max,
bool *next) const{
    if(left == NULL || right < *left) return -1;
    off_t off_left = searchLeaf(*left);
    off_t off_right = searchLeaf(right);
    off_t off = off_left;
    size_t i = 0;
    record_t *b_rec, *e_rec;
    leaf_t leaf;
    while(off != off_right && off != 0 && i < max){
        map(&leaf, off);
        b_rec = off_left == off ? find(leaf, *left) : begin(leaf);
        e_rec = leaf.children + leaf.num;
        for(; b_rec != e_rec && i < max; b_rec++, i++) values[i] = b_rec -> value;
        off = leaf.next;
    }
    if(i < max){
        map(&leaf, off_right);
        b_rec = find(leaf, *left);
        e_rec = std::upper_bound(begin(leaf), end(leaf), right);
        for(; b_rec != e_rec && i < max; b_rec++, i++) values[i] = b_rec -> value;
    }
    if(next != NULL){
        if(i == max && b_rec != e_rec){*next = true; *left = b_rec -> key;}
        else *next = false;
    }
    return i;
}
// поиск по значению
int BPlusTree::search(const key_t& key, value_t *value) const{
    leaf_t leaf;
    map(&leaf, searchLeaf(key));
    record_t *record = find(leaf, key);
    if(record != leaf.children + leaf.num){
        *value = record -> value;
        return keyCmp(record -> key, key);
    }
    else return -1;
}
off_t searchLeaf(const key_t &key) const{
    return searchLeaf(searchIndex(key), key);
}
```

`searchIndex` – это функция обхода индекса, которая, используя высоту, полученную из структуры meta, циклически сопоставляет каждый искомый индекс, и с помощью `upper_bound` находит и возвращает начало индекса (если не находит, то возвращается к концу последовательности и переходит к следующему циклу).

`searchLeaf` – это конечная функция поиска, аргументами являются индекс, полученный из `searchIndex`, и целевой ключ, который нужно выбрать, сопоставьте искомый индекс с узлом, снова используя `upper_bound`, чтобы найти и вернуть дочерний элемент конечного узла.

В функции `search` вызываем `searchLeaf(key)`, чтобы найти конечный узел, в котором находится ключ целевой записи. Поиск возвращает значение `keyCmp(record, key)`, если найденная запись не является последней. Если запись является последней, то искомое значение не найдено.

Вставка данных

В дереве B+ каждая запись (с данными) вставляется на нижний уровень. Вставка данных производится в порядке возрастания. Если в конечном узле есть переполнение, то необходимо разделить узел на два узла. Первый узел будет содержать (order-1)/2 значений. Если при этом и во внутренних узлах происходит переполнение, то их тоже надо разделить, и первый узел будет содержать m/2-1 значений [3, 7, 8].

```
int BPlusTree::insert(const key_t& key, value_t value){
    off_t parent = searchIndex(key);
    off_t offset = searchLeaf(parent, key);
    leaf_t leaf;
    if(std::binary_search(begin(leaf), end(leaf), key)) return 1; //уже есть такое
значение
    if(leaf.num == meta.order){
        // обнаружено переполнение. требуется разделение
        leaf_t new_leaf;
        createNode(offset, &leaf, &new_leaf);
        // ищем точку разделение
        size_t point = leaf.num / 2;
        bool place_right = leaf.children[point].key < key;
        if(place_right) point++;
        // разделяем узел
        std::copy(leaf.children + point, leaf.children + leaf.num, new_leaf.children);
        new_leaf.num = leaf.num - point;
        leaf.num = point;
        if(place_right) insertRecNoSplit(&new_leaf, key, value)
            else insertRecNoSplit(&leaf, key, value);
        insertKey(parent, new_leaf.children[0].key, offset, leaf.next);
    }else{
        insertRecNoSplit(&leaf, key, value);
    }
    return 0;
}
```

`insertRecNoSplit` используется для обработки операций вставки записей, которые не разделяют узлы. Эта вставка выполняет поиск точки вставки в целевом листе, выполняет операцию перезаписи, чтобы переместить узел после точки вставки для освобождения места для вставки новой записи.

`insertKey` используется для обработки операции вставки ключа. Операция insert делится на два вида операций вставки, одна из которых выполняется без разделения узла, другая – когда число количество узлов достигает определенного порядка (order). Сначала требуется найти точку разделения, создать новый внутренний узел, и нужно вызвать операции вставки и перебалансировки структуры данных, чтобы настроить родительский узел.

`insert` используется для вставки записей с предварительной проверкой на дублирование данных. Операция вставки делится на два случая, первый – если количество записей не достигает заданного порядка, то есть не требуется проводить разделение узла, и второй – когда количество записей достигает заданного порядка, и нужно выполнить разделение: создаем новый узел, определяем точку разделения, проводим разделение и обновляем информацию о дочерних элементах. В конце требуется дополнительно обновить родительские элементы для поддержания сбалансированной структуры дерева.

Удаление

Удаление элемента в дереве B+ состоит из трех основных действий: поиск в узле, удаление ключа, и перебалансировка [3, 6, 7, 8] дерева (если она требуется).

```
int BPlusTree::remove(const key_t& key){
    internal_t parent;
    leaf_t leaf;
    off_t parent_off = searchIndex(key);
    map(&parent, parent_off);
    index_t *where = find(parent, key);
    off_t offset = where -> child;
    if(!std::binary_search(begin(leaf), end(leaf), key)) return -1;
    size_t min = meta.leaf_num == 1 ? 0 : meta.order / 2;
    record_t *to_delete = find(leaf, key);
    std::copy(to_delete + 1, end(leaf), to_delete);
    leaf.num--;
    if(leaf.num < min){
        bool borrowed = false;
        if(leaf.prev != 0) borrowed = borrowKey(false, leaf);
        if(!borrowed && leaf.next != 0) borrowed = borrowKey(true, leaf);
        if(!borrowed){
            key_t index_key;
            leaf_t prev;
            index_key = begin(prev)->key;
            mergeLeaf(&prev, &leaf);
            removeNode(&prev, &leaf);
        }
        removeIndex(parent_off, parent, index_key);
    }
    return 0;
}
```

Удалить узел достаточно просто, отменить выделение памяти на узел (и обновить метаданные), перенести ссылку преемника удаленного узла на предыдущий узел, если удаленный узел не является последним узлом, перенести ссылку предыдущего узла на его предшественника [3, 6, 7, 8].

`removeIndex` используется для удаления индекса. `to_delete` – это индекс для удаления, если `to_delete` не является последним индексом, то надо перенести его дочерние элементы в последний индекс и перезаписать его. Если `to_delete` является корневым и есть только один узел, то надо удалите его напрямую и уменьшить высоту дерева, чтобы сбалансировать структуру. Если количество дочерних элементов узла, в котором находится `to_delete`, меньше `min`, необходимо выполнить заимствование и слияние узлов. Если предшественник данного листа не равен 0, то надо заимствовать слева, если последующий узел не равен 0, то заимствовать справа. Объединяем лист и его предшественника в один узел, удаляем лист, выполняем операцию слияния. После этого проводим удаление в индексе `removeIndex(parent_of, parent, index_key)`.

Обновление

Алгоритм обновления использует searchLeaf, чтобы найти целевой лист, если он не найден, то возвращает значение -1, в противном случае определяет, соответствует ли найденный ключ записи ключу из параметра функции, затем обновляет значение записи [3, 6, 7, 8].

```
int BPlusTree::update(const key_t& key, value_t value){
    off_t offset = searchLeaf(key);
    leaf_t leaf;
    record_t *record = find(leaf, key);
    if(record != leaf.children + leaf.num){
        record->value = value;
        return 0;
    }else return -1; // not found
}
```

Пользовательский интерфейс

Для упрощения разработки и выполнения тестирования необходимо разработать пользовательский интерфейс, который для упрощения реализации может быть консольным. Потребуется реализовать разбор вводимых команд. Пример такой реализации может быть следующим. Однако, заметим, что это не единственный возможный вариант.

```
int main() {
    while(1) {
        printf("> ");
        if (fgets(input, sizeof(input), stdin) == NULL) break; // Обработка EOF
        input[strcspn(input, "\n")] = '\0'; // Удаление символа новой строки
        // Разбор введённой строки
        if (sscanf(input, "%s %s", cmd, arg) < 1) continue; // Пустой ввод
        if (strcmp(cmd, "insert") == 0) insert(key, value);
        ...
        if (strcmp(cmd, "exit") == 0) printf("Exit...\n"); break;
    }
    return 0;
}
```

Заключение

В работе представлена реализация локальной базы данных на основе B+ дерева, демонстрирующая его возможности для хранения и поиска данных. Реализована система хранения данных по принципу «ключ-значение» с поддержкой различных типов. Реализован обход дерева с поддержкой точных и диапазонных запросов, автоматическое разделение узлов при переполнении, и перебалансировка дерева, модификация значений в листьях без нарушения структуры дерева.

СПИСОК ЛИТЕРАТУРЫ

1. *Comer, D.* The Ubiquitous B-Tree / D. Comer // ACM Computing Surveys. – 1979. – Vol. 11, № 2. – P. 121–137. – DOI: 10.1145/356770.356776.
2. Шенн, Дж. Проектирование и реализация баз данных / Дж. Шенн. – М.: ДМК Пресс, 2018. – 736 с.
3. Кормен, Т. Алгоритмы. Построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. – 3-е изд. – М. : Вильямс, 2022. – 1328 с.
4. B-Tree datastructure. BTech Smart Class [Электронный источник]. – URL: http://btechsmartclass.com/data_structures/b-trees.html (дата обращения 01.06.2025)
5. Kerttu Pollari-Malmi. B+-trees. Computer Science, Faculty of Science, University of Helsinki [Электронный источник]. – URL: https://www.cs.helsinki.fi/u/mluukkai/tirak2010/B-tree.pdf (дата обращения 01.06.2025)
6. Кнут, Д. Искусство программирования. Том 3. Сортировка и поиск / Д. Кнут. – 2-е изд. – М.: Вильямс, 2021. – 832 с.
7. Седжвик, Р. Алгоритмы на C++ / Р. Седжвик. – 4-е изд. – СПб.: Питер, 2019. – 1056 с.
8. Гарсия-Молина, Г. Системы баз данных. Полный курс / Г. Гарсия-Молина, Дж. Д. Ульман, Дж. Уидом. – М.: Диалектика, 2020. – 1088 с.

ИНФОРМАЦИЯ ОБ АВТОРАХ

Андрей Викторович Шахомиров
Доцент кафедры аэрокосмических компьютерных и программных систем, кандидат технических наук
Санкт-Петербургский государственный университет аэрокосмического приборостроения
Россия, 190000, Санкт-Петербург, ул. Большая Морская, д.67, лит. А
E-mail: shakhomirov@guap.ru

Наталья Евгеньевна Шахомирова
Ассистент кафедры аэрокосмических компьютерных и программных систем
Санкт-Петербургский государственный университет аэрокосмического приборостроения
Россия, 190000, Санкт-Петербург, ул. Большая Морская, д.67, лит. А
E-mail: n.shah2024@yandex.ru

Дата поступления: 25.03.2025
Дата принятия: 01.07.2025