Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Институт информационных технологий

Факультет компьютерных технологий

Лабораторная работа № 15

«В-ДЕРЕВЬЯ»

Выполнил: студент гр. 281073

Буйновский М.В.

Проверил: Потоцкий Д.С.

Минск 2023

**Цель работы:** изучить способы обработки данных с помощью В-деревьев.

**Задание:** ввести 10-15 целых чисел и построить из них В-дерево порядка 5. Реализовать операции поиска, вставки и удаления элементов В-дерева.

**Код программы:**

**b\_tree.hpp**

#pragma once

#include <cstddef>

#include <filesystem>

#include <fstream>

#include <iomanip>

#include <iostream>

#include <stdexcept>

#include <string>

#include <vector>

class Color

{

public:

inline static const std::string RED = "\033[31m";

inline static const std::string GREEN = "\033[32m";

inline static const std::string YELLOW = "\033[33m";

inline static const std::string CYAN = "\033[36m";

inline static const std::string RESET = "\033[0m";

};

class BTreeNode

{

private:

int \*keys;

BTreeNode \*\*children;

int size;

int order;

bool isLeaf;

void insertNonFull(int key);

void splitChild(int idx, BTreeNode \*y);

void removeFromLeaf(int idx);

void removeFromBranch(int idx);

int getPredecessor(int idx);

int getSuccessor(int idx);

void fill(int idx);

void borrowFromPrev(int idx);

void borrowFromNext(int idx);

void merge(int idx);

public:

friend class BTree;

BTreeNode(int order, bool isLeaf);

BTreeNode \*search(int k);

void traverse(bool isRoot = false);

int find(int key);

void remove(int key);

};

class BTree

{

BTreeNode \*root;

int order;

public:

BTree(int order)

{

root = nullptr;

this->order = order;

}

void traverse();

void insert(int key);

void remove(int key);

BTreeNode \*search(int k);

};

**b\_tree.hpp**

#include "b\_tree.hpp"

BTreeNode::BTreeNode(int order, bool isLeaf)

{

this->order = order;

this->isLeaf = isLeaf;

this->keys = new int[order - 1];

this->children = new BTreeNode \*[order];

this->size = 0;

}

int BTreeNode::find(int key)

{

int idx = 0;

while (idx < size && keys[idx] < key) {

++idx;

}

return idx;

}

void BTreeNode::remove(int key)

{

int idx = this->find(key);

if (idx < this->size && keys[idx] == key) {

if (this->isLeaf) {

this->removeFromLeaf(idx);

} else {

this->removeFromBranch(idx);

}

} else {

if (this->isLeaf) {

std::cout << Color::RED << "The key " << Color::YELLOW << key << Color::RED << " does not exist in the tree"

<< Color::RESET << std::endl;

return;

}

bool flag = idx == this->size;

if (this->children[idx]->size < order) {

this->fill(idx);

}

if (flag && idx > this->size) {

this->children[idx - 1]->remove(key);

} else {

this->children[idx]->remove(key);

}

}

}

void BTreeNode::removeFromLeaf(int idx)

{

for (int i = idx + 1; i < this->size; ++i) {

keys[i - 1] = keys[i];

}

this->size--;

}

void BTreeNode::removeFromBranch(int idx)

{

int key = keys[idx];

if (this->children[idx]->size >= this->order) {

int pred = this->getPredecessor(idx);

keys[idx] = pred;

this->children[idx]->remove(pred);

} else if (this->children[idx + 1]->size >= this->order) {

int succ = this->getSuccessor(idx);

keys[idx] = succ;

this->children[idx + 1]->remove(succ);

} else {

merge(idx);

this->children[idx]->remove(key);

}

}

int BTreeNode::getPredecessor(int idx)

{

BTreeNode \*cur = this->children[idx];

while (!cur->isLeaf) {

cur = cur->children[cur->size];

}

return cur->keys[cur->size - 1];

}

int BTreeNode::getSuccessor(int idx)

{

BTreeNode \*cur = this->children[idx + 1];

while (!cur->isLeaf) {

cur = cur->children[0];

}

return cur->keys[0];

}

void BTreeNode::fill(int idx)

{

if (idx != 0 && this->children[idx - 1]->size >= this->order) {

this->borrowFromPrev(idx);

} else if (idx != this->size && this->children[idx + 1]->size >= this->order) {

this->borrowFromNext(idx);

} else {

if (idx != this->size) {

this->merge(idx);

} else {

this->merge(idx - 1);

}

}

}

void BTreeNode::borrowFromPrev(int idx)

{

BTreeNode \*child = this->children[idx];

BTreeNode \*sibling = this->children[idx - 1];

for (int i = child->size - 1; i >= 0; --i) {

child->keys[i + 1] = child->keys[i];

}

if (!child->isLeaf) {

for (int i = child->size; i >= 0; --i) {

child->children[i + 1] = child->children[i];

}

}

child->keys[0] = keys[idx - 1];

if (!child->isLeaf) {

child->children[0] = sibling->children[sibling->size];

}

keys[idx - 1] = sibling->keys[sibling->size - 1];

child->size += 1;

sibling->size -= 1;

}

void BTreeNode::borrowFromNext(int idx)

{

BTreeNode \*child = this->children[idx];

BTreeNode \*sibling = this->children[idx + 1];

child->keys[child->size] = keys[idx];

if (!child->isLeaf) {

child->children[child->size + 1] = sibling->children[0];

}

keys[idx] = sibling->keys[0];

for (int i = 1; i < sibling->size; ++i) {

sibling->keys[i - 1] = sibling->keys[i];

}

if (!sibling->isLeaf) {

for (int i = 1; i <= sibling->size; ++i) {

sibling->children[i - 1] = sibling->children[i];

}

}

child->size += 1;

sibling->size -= 1;

}

void BTreeNode::merge(int idx)

{

BTreeNode \*child = this->children[idx];

BTreeNode \*sibling = this->children[idx + 1];

child->keys[this->order - 1] = keys[idx];

for (int i = 0; i < sibling->size; ++i) {

child->keys[i + this->order] = sibling->keys[i];

}

if (!child->isLeaf) {

for (int i = 0; i <= sibling->size; ++i) {

child->children[i + this->order] = sibling->children[i];

}

}

for (int i = idx + 1; i < this->size; ++i) {

keys[i - 1] = keys[i];

}

for (int i = idx + 2; i <= this->size; ++i) {

this->children[i - 1] = this->children[i];

}

child->size += sibling->size + 1;

this->size--;

delete (sibling);

}

void BTree::insert(int key)

{

if (root == nullptr) {

root = new BTreeNode(this->order, true);

root->keys[0] = key;

root->size = 1;

} else {

if (root->size == this->order - 1) {

BTreeNode \*node = new BTreeNode(this->order, false);

node->children[0] = root;

node->splitChild(0, root);

int i = 0;

if (node->keys[0] < key) i++;

node->children[i]->insertNonFull(key);

root = node;

} else

root->insertNonFull(key);

}

}

void BTreeNode::insertNonFull(int key)

{

int i = this->size - 1;

if (this->isLeaf == true) {

while (i >= 0 && keys[i] > key) {

keys[i + 1] = keys[i];

i--;

}

keys[i + 1] = key;

this->size = this->size + 1;

} else {

while (i >= 0 && keys[i] > key)

i--;

if (children[i + 1]->size == this->order - 1) {

splitChild(i + 1, this->children[i + 1]);

if (keys[i + 1] < key) {

i++;

}

}

this->children[i + 1]->insertNonFull(key);

}

}

void BTreeNode::splitChild(int idx, BTreeNode \*y)

{

BTreeNode \*z = new BTreeNode(y->order, y->isLeaf);

z->size = this->order - 1;

for (int j = 0; j < this->order - 1; j++) {

z->keys[j] = y->keys[j + this->order];

}

if (y->isLeaf == false) {

for (int j = 0; j < this->order; j++) {

z->children[j] = y->children[j + this->order];

}

}

y->size = this->order - 1;

for (int j = this->size; j >= idx + 1; j--) {

this->children[j + 1] = this->children[j];

}

this->children[idx + 1] = z;

for (int j = this->size - 1; j >= idx; j--) {

keys[j + 1] = keys[j];

}

keys[idx] = y->keys[this->order - 1];

this->size = this->size + 1;

}

void BTreeNode::traverse(bool isRoot)

{

int i;

for (i = 0; i < this->size; i++) {

if (this->isLeaf == false) this->children[i]->traverse();

std::cout << " " << (isRoot ? Color::GREEN : (this->isLeaf ? Color::CYAN : Color::YELLOW)) << keys[i]

<< Color::RESET;

}

if (this->isLeaf == false) {

this->children[i]->traverse();

}

}

void BTree::remove(int key)

{

if (!root) {

std::cout << Color::RED << "The tree is empty" << Color::RESET << std::endl;

return;

}

root->remove(key);

if (root->size == 0) {

BTreeNode \*tmp = root;

if (root->isLeaf) {

root = nullptr;

} else {

root = root->children[0];

}

delete tmp;

}

}

void BTree::traverse()

{

if (this->root != nullptr) {

this->root->traverse(true);

std::cout << std::endl;

}

}

BTreeNode \*BTreeNode::search(int key)

{

int idx = this->find(key);

if (keys[idx] == key) {

return this;

}

if (this->isLeaf == true) {

return nullptr;

}

return this->children[idx]->search(key);

}

BTreeNode \*BTree::search(int key)

{

return this->root == nullptr ? nullptr : this->root->search(key);

}

**main.cpp**

#include "b\_tree.hpp"

#include <cstdlib>

#include <iostream>

#include <random>

int main()

{

BTree btree(5);

while (true) {

std::cout << "Enter action (" << Color::CYAN << "traverse" << Color::RESET << ", " << Color::CYAN << "find"

<< Color::RESET << ", " << Color::CYAN << "add" << Color::RESET << ", " << Color::CYAN << "delete"

<< Color::RESET << "): " << Color::YELLOW;

std::string action;

std::cin >> action;

std::cout << Color::RESET;

if (action == "traverse") {

btree.traverse();

} else if (action == "find") {

std::cout << "Enter key to find: ";

int key;

std::cin >> key;

auto result = btree.search(key);

if (result != nullptr) {

std::cout << Color::GREEN << "Found" << Color::RESET << std::endl;

} else {

std::cout << Color::RED << "Not Found" << Color::RESET << std::endl;

}

} else if (action == "add") {

std::cout << "Enter value to add: ";

int value;

std::cin >> value;

btree.insert(value);

} else if (action == "delete") {

std::cout << "Enter key to delete: ";

int key;

std::cin >> key;

btree.remove(key);

} else {

std::cout << "Invalid action!" << std::endl;

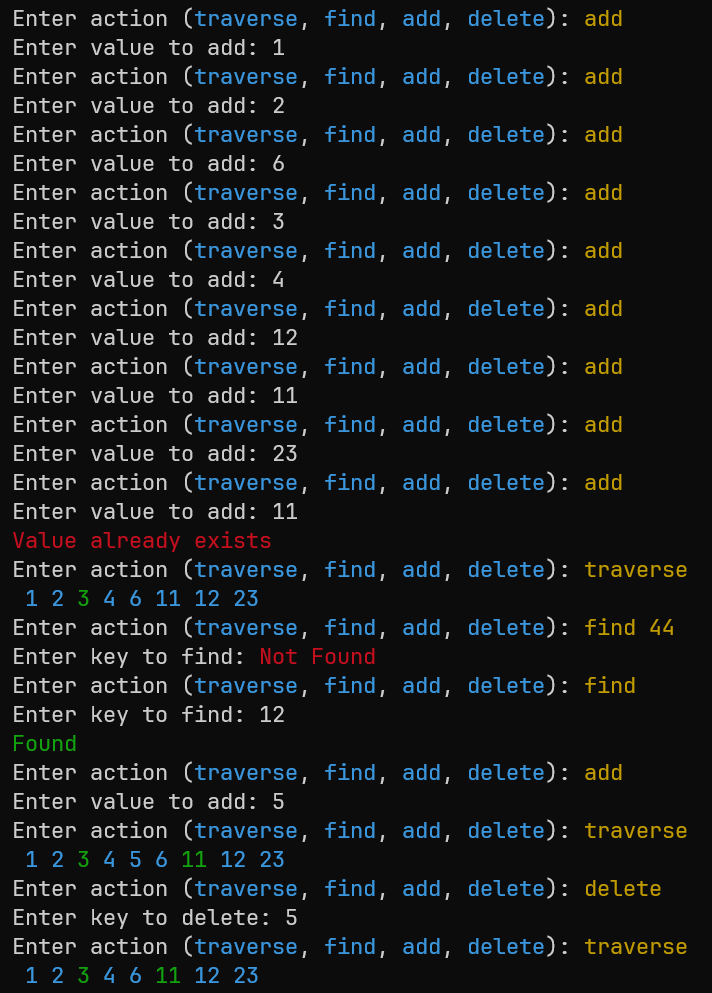
}

}

return EXIT\_SUCCESS;

}

**Скриншот работы программы, демонстрация результатов:**

****

**Контрольные вопросы:**

**1. В чем отличия между открытым и закрытым хешированием данных?**

В-дерево отличается от бинарного дерева несколькими основными характеристиками:

* Один узел B-дерева может содержать несколько ключей, в то время как узел бинарного дерева содержит только один ключ.
* Все листья B-дерева находятся на одном уровне, что обеспечивает более эффективный поиск и вставку элементов, чем в бинарном дереве.
* B-дерево может иметь переменное количество детей в узлах, в то время как у бинарного дерева каждый узел может иметь не более двух детей.
* B-дерево применяется в основном для работы с большим объемом данных и обеспечивает эффективное хранение, поиск и вставку элементов.
* B-дерево обычно используется в базах данных, файловых системах и других приложениях, где требуется эффективное управление большим объемом данных.

Это лишь некоторые из основных отличий между B-деревом и бинарным деревом. Все эти отличия позволяют B-дереву обеспечивать более эффективную работу с большими объемами данных и обеспечивать эффективное хранение и поиск элементов.

**2. Объясните суть коллизий при закрытом хешировании данных.**

Арность В-дерева выбирается на основании нескольких факторов, включая размер ключей и данных, доступ к данным и операции, выполняемые на дереве.

Одним из главных факторов является размер ключей и данных. Если ключи и данные имеют большой размер, то меньшая арность может быть предпочтительной, чтобы каждый узел В-дерева мог вместить больше информации. С другой стороны, если ключи и данные имеют относительно маленький размер, то большая арность может быть более эффективной, так как каждый узел будет содержать больше ключей и данных.

Также важно учитывать доступ к данным и операции, выполняемые на дереве. Если доступ к данным осуществляется через долгий сетевой канал или медленное хранилище, то меньшая арность может быть предпочтительной, чтобы уменьшить количество операций чтения и записи. С другой стороны, если доступ к данным осуществляется через быстрое хранилище или локальную память, то большая арность может быть более эффективной, так как уменьшается количество операций чтения и записи.

Наконец, арность В-дерева также зависит от требуемой производительности и использования памяти. Большая арность обычно требует меньше уровней дерева и, следовательно, меньше операций чтения и записи. Однако большая арность также требует больше памяти, так как каждый узел должен содержать больше ключей и данных. Меньшая арность требует больше уровней дерева и, следовательно, больше операций чтения и записи, но использует меньше памяти.

Итак, выбор арности В-дерева зависит от размера ключей и данных, доступа к данным и операций, производительности и использования памяти. В конечном итоге, выбор арности В-дерева зависит от конкретных требований и ограничений приложения.

**3. Перечислите способы борьбы с коллизиями.**

Ускорение обработки данных при использовании В-деревьев достигается за счет следующих факторов:

Быстрый доступ к данным: В-деревья обладают эффективной структурой данных, которая обеспечивает быстрый доступ к данным. Каждый узел В-дерева содержит множество ключей, и дерево разбивается на уровни, где каждый уровень содержит множество узлов. Благодаря этой структуре данных, поиск и вставка данных выполняются за время O(log n), где n - количество узлов в дереве.

* Эффективное использование памяти: В-деревья позволяют эффективно использовать память, так как узлы в дереве хранятся на диске или в оперативной памяти. Это позволяет обрабатывать большие объемы данных, не требуя большой объем памяти.
* Поддержка быстрой вставки и удаления: В-деревья поддерживают эффективные операции вставки и удаления данных. При вставке нового элемента или удалении существующего элемента В-дерево автоматически балансируется, чтобы сохранить оптимальное распределение данных.
* Поддержка диапазонных запросов: В-деревья позволяют выполнять диапазонные запросы, то есть находить все данные, которые удовлетворяют определенному условию. Это делает В-деревья особенно полезными для операций, связанных с поиском данных в заданном диапазоне.

В целом, использование В-деревьев позволяет эффективно ускорить обработку данных и обеспечить быстрый доступ к данным, что особенно важно при работе с большими объемами данных.

**Вывод:** в результате выполнения лабораторной работы освоили изучить способы обработки данных с помощью В-деревьев.