**ЛАБОРАТОРНАЯ РАБОТА №3**

**«ИССЛЕДОВАНИЕ Б-ДЕРЕВЬЕВ ПРИ ДОСТУПЕ К ДАННЫМ ВО ВНЕШНЕЙ ПАМЯТИ»**

**Цель работы**

Исследовать возможности применения нелинейных структур данных – Б-деревьев, для хранения и поиска информации. Приобрести практические навыки использования Б-деревьев для реализации эффективного поиска и доступа к данным. Произвести оценку эффективности использования Б-деревьев для организации хранения данных.

**Вариант задания**

Вариант 8

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ключевое поле | N1 | N2 | N3 | N4 | N5 |
| Абонент | 30 | 500 | 1900 | 4000 | 7500 |

**Ход выполнения работы**

Была написана программа на языке C++, которая реализует классы для работы со списком и с бинарным деревом поиска. Текст программы представлен в приложении А.

Было произведено добавление 30 элементов в Б-дерево со степенью дерева 4. Результаты по скорости выполнения операций изображены на рисунке 1.

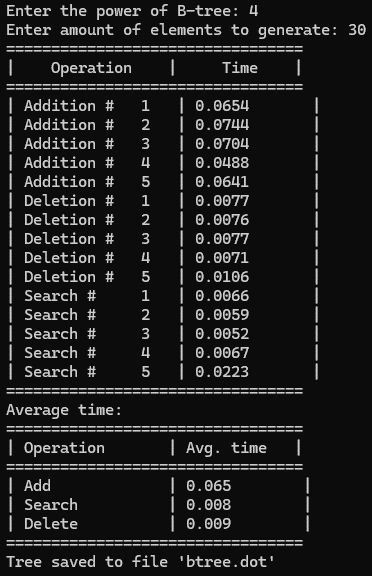


Рисунок 1 – Результаты операций добавления, удаления и поиска для Б-дерева при 50 элементов

Было произведено добавление 500 элементов в Б-дерево со степенью дерева 5. Результаты по скорости выполнения операций изображены на рисунке 2.

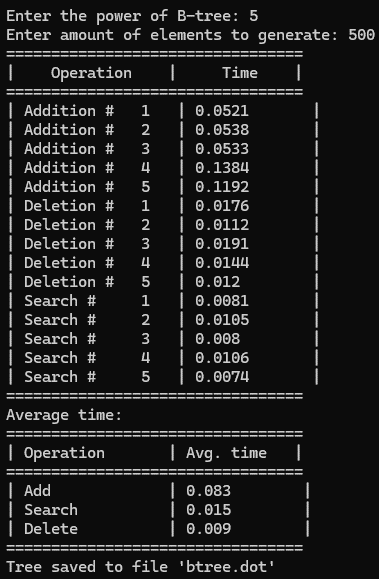


Рисунок 2 – Результаты операций добавления, удаления и поиска для Б-дерева при 500 элементов

Было произведено добавление 1900 элементов в Б-дерево со степенью дерева 6. Результаты по скорости выполнения операций изображены на рисунке 3.

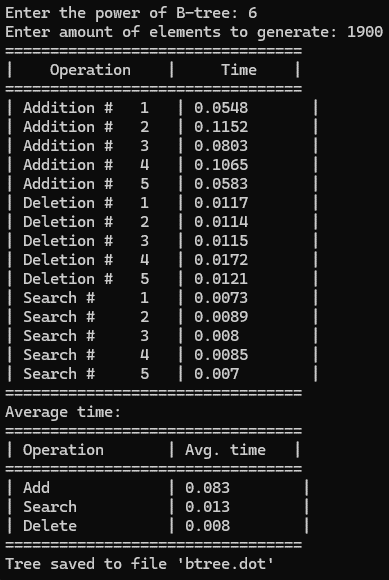


Рисунок 3 – Результаты операций добавления, удаления и поиска для Б-дерева при 1600 элементов

Было произведено добавление 4000 элементов в Б-дерево со степенью дерева 7. Результаты по скорости выполнения операций изображены на рисунке 4.

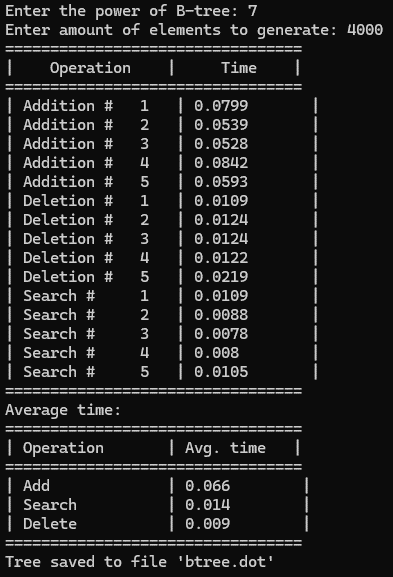


Рисунок 4 – Результаты операций добавления, удаления и поиска для Б-дерева при 4000 элементов

Было произведено добавление 7500 элементов в Б-дерево со степенью дерева 8. Результаты по скорости выполнения операций изображены на рисунке 5.

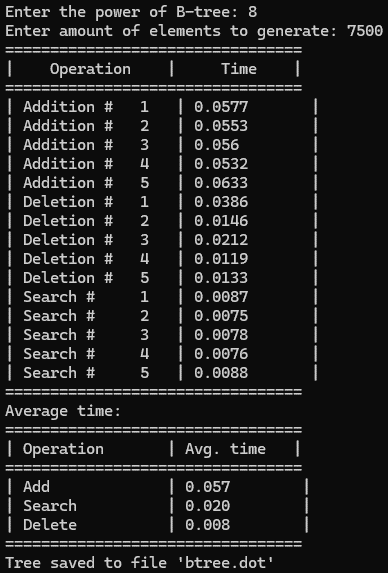


Рисунок 5 – Результаты операций добавления, удаления и поиска для Б-дерева при 7500 элементов

Было проведено добавление 100000 элементов в Б-дерево со степенью Б-дерева 10. Результаты изображены на рисунке 6.

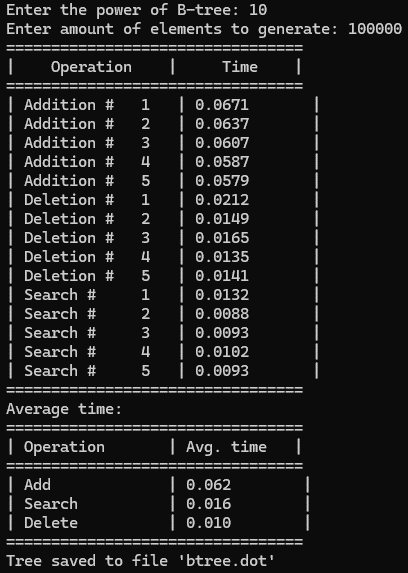


Рисунок 6 – Результаты операций добавления, удаления и поиска для Б-дерева при 100000 элементов

При помощи пакета утилит для визуализации графов была выведена структура Б-дерева на 50 элементов со степенью 4. Это изображено на рисунке 7.

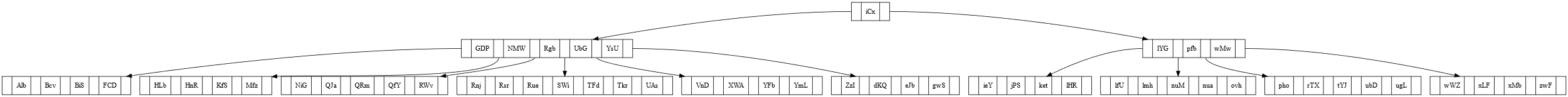


Рисунок 7 – Дерево из 50 элементов

**Вывод**

В ходе выполнения лабораторной работы была написана программа, реализующая построение B-дерева и выводящая таблицу с временем, требуемым на выполнение операций поиска, удаления и добавления. На основе информации, полученной из предыдущих работ, сделан вывод о высокой эффективности этой структуры данных по сравнению с AVL-деревом или бинарным деревом.

**ПРИЛОЖЕНИЕ А**

Листинг 1 - Текст программы с реализацией классов для работы с Б-деревом.

#include <iostream>

#include <Windows.h>

#include <string>

#include <vector>

#include <ctime>

#include <random>

#include <cstdlib>

#include <fstream>

#include <chrono>

#include <iomanip>

using namespace std;

using namespace std::chrono;

struct Record {

string subscriber;

string name;

string address;

string phone;

double account;

Record(string subscriber, string name, string address, string phone, double account)

: subscriber(subscriber), name(name), address(address), phone(phone), account(account) {}

};

class TreeNode {

vector<Record> keys;

int t;

TreeNode\*\* C;

int n;

bool leaf;

public:

TreeNode(int temp, bool bool\_leaf);

void insertNonFull(Record k);

void splitChild(int i, TreeNode\* y);

void traverse();

void remove(string k);

TreeNode\* search(string k);

int findKey(string k);

void removeFromLeaf(int idx);

void removeFromNonLeaf(int idx);

Record getPredecessor(int idx);

Record getSuccessor(int idx);

void fill(int idx);

void borrowFromPrev(int idx);

void borrowFromNext(int idx);

void merge(int idx);

void saveAsDot(ofstream& out);

void generateDotCode(ofstream& out);

friend class BTree;

};

class BTree {

public:

TreeNode\* root;

int t;

BTree(int temp) {

root = NULL;

t = temp;

}

void traverse() {

if (root != NULL)

root->traverse();

}

TreeNode\* search(string k) {

return (root == NULL) ? NULL : root->search(k);

}

void insert(Record k);

void remove(string k);

};

TreeNode::TreeNode(int t1, bool leaf1) {

t = t1;

leaf = leaf1;

C = new TreeNode \* [2 \* t];

n = 0;

}

void TreeNode::traverse() {

int i;

for (i = 0; i < n; i++) {

if (leaf == false)

C[i]->traverse();

cout << " " << keys[i].subscriber;

}

if (leaf == false)

C[i]->traverse();

}

TreeNode\* TreeNode::search(string k) {

int i = 0;

while (i < n && k > keys[i].subscriber)

i++;

if (i < n && keys[i].subscriber == k)

return this;

if (leaf == true)

return NULL;

return C[i]->search(k);

}

void BTree::insert(Record k) {

if (root == NULL) {

root = new TreeNode(t, true);

root->keys.push\_back(k);

root->n = 1;

}

else {

if (root->n == 2 \* t - 1) {

TreeNode\* s = new TreeNode(t, false);

s->C[0] = root;

s->splitChild(0, root);

int i = 0;

if (s->keys[0].subscriber < k.subscriber)

i++;

s->C[i]->insertNonFull(k);

root = s;

}

else

root->insertNonFull(k);

}

}

void TreeNode::insertNonFull(Record k) {

int i = n - 1;

if (leaf == true) {

keys.push\_back(Record("", "", "", "", 0.0));

while (i >= 0 && keys[i].subscriber > k.subscriber) {

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

i--;

}

keys[i + 1] = k;

n = n + 1;

}

else {

while (i >= 0 && keys[i].subscriber > k.subscriber)

i--;

if (C[i + 1]->n == 2 \* t - 1) {

splitChild(i + 1, C[i + 1]);

if (keys[i + 1].subscriber < k.subscriber)

i++;

}

C[i + 1]->insertNonFull(k);

}

}

void TreeNode::splitChild(int i, TreeNode\* y) {

TreeNode\* z = new TreeNode(y->t, y->leaf);

z->n = t - 1;

for (int j = 0; j < t - 1; j++)

z->keys.push\_back(y->keys[j + t]);

if (y->leaf == false) {

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;

keys.push\_back(Record("", "", "", "", 0.0));

for (int j = n - 1; j >= i; j--)

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

keys[i] = y->keys[t - 1];

n = n + 1;

}

int TreeNode::findKey(string k) {

int idx = 0;

while (idx < n && keys[idx].subscriber < k)

++idx;

return idx;

}

void TreeNode::remove(string k) {

int idx = findKey(k);

if (idx < n && keys[idx].subscriber == k) {

if (leaf)

removeFromLeaf(idx);

else

removeFromNonLeaf(idx);

}

else {

if (leaf) {

return;

}

bool flag = ((idx == n) ? true : false);

if (C[idx]->n < t)

fill(idx);

if (flag && idx > n)

C[idx - 1]->remove(k);

else

C[idx]->remove(k);

}

return;

}

void TreeNode::removeFromLeaf(int idx) {

for (int i = idx + 1; i < n; ++i)

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

n--;

return;

}

void TreeNode::removeFromNonLeaf(int idx) {

Record k = keys[idx];

if (C[idx]->n >= t) {

Record pred = getPredecessor(idx);

keys[idx] = pred;

C[idx]->remove(pred.subscriber);

}

else if (C[idx + 1]->n >= t) {

Record succ = getSuccessor(idx);

keys[idx] = succ;

C[idx + 1]->remove(succ.subscriber);

}

else {

merge(idx);

C[idx]->remove(k.subscriber);

}

return;

}

Record TreeNode::getPredecessor(int idx) {

TreeNode\* cur = C[idx];

while (!cur->leaf)

cur = cur->C[cur->n];

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

}

Record TreeNode::getSuccessor(int idx) {

TreeNode\* cur = C[idx + 1];

while (!cur->leaf)

cur = cur->C[0];

return cur->keys[0];

}

void TreeNode::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);

}

return;

}

void TreeNode::borrowFromPrev(int idx) {

TreeNode\* child = C[idx];

TreeNode\* 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;

return;

}

void TreeNode::borrowFromNext(int idx) {

TreeNode\* child = C[idx];

TreeNode\* 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;

return;

}

void TreeNode::merge(int idx) {

TreeNode\* child = C[idx];

TreeNode\* 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);

return;

}

void BTree::remove(string k) {

if (!root) {

cout << "Tree is empty\n";

return;

}

root->remove(k);

if (root->n == 0) {

TreeNode\* tmp = root;

if (root->leaf)

root = NULL;

else

root = root->C[0];

delete tmp;

}

return;

}

void TreeNode::saveAsDot(ofstream& out) {

out << "digraph BTree {" << endl;

out << "node [shape=record];" << endl;

generateDotCode(out);

out << "}" << endl;

}

void TreeNode::generateDotCode(ofstream& out) {

out << "node" << this << " [label=\"";

for (int i = 0; i < n; ++i) {

out << "<f" << i << ">|" << keys[i].subscriber << "|";

}

out << "<f" << n << ">\"];" << endl;

for (int i = 0; i <= n; ++i) {

if (!leaf && C[i]) {

out << "node" << this << ":f" << i << " -> node" << C[i] << ";" << endl;

C[i]->generateDotCode(out);

}

}

}

vector<string> firstNames = {

"Liam", "Emma", "Noah", "Olivia", "William", "Ava", "James", "Isabella",

"Oliver", "Sophia", "Benjamin", "Mia", "Elijah", "Charlotte", "Lucas", "Harper"

};

vector<string> lastNames = {

"Smith", "Johnson", "Williams", "Brown", "Davis", "Miller", "Wilson", "Moore",

"Taylor", "Anderson", "Thomas", "Jackson", "White", "Harris", "Martin", "Thompson"

};

string generateRandomName() {

static random\_device rd;

static mt19937 gen(rd());

static uniform\_int\_distribution<> firstNameDist(0, firstNames.size() - 1);

static uniform\_int\_distribution<> lastNameDist(0, lastNames.size() - 1);

string firstName = firstNames[firstNameDist(gen)];

string lastName = lastNames[lastNameDist(gen)];

return firstName + " " + lastName;

}

string generateRandomString(int length) {

string chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

string result;

for (int i = 0; i < length; ++i) {

result += chars[rand() % chars.length()];

}

return result;

}

Record generateRandomRecord() {

string subscriber = generateRandomString(3);

string name = generateRandomName();

string address = "st. " + generateRandomString(8) + ", h. " + to\_string(rand() % 100);

string phone = "+7" + to\_string(9000000000 + rand() % 1000000000);

double account = static\_cast<double>(rand() % 100000) / 100.0;

return Record(subscriber, name, address, phone, account);

}

int main() {

SetConsoleOutputCP(65001);

srand(static\_cast<unsigned int>(time(0)));

int t;

cout << "Enter the power of B-tree: ";

cin >> t;

BTree tree(t);

int numRecords;

cout << "Enter amount of elements to generate: ";

cin >> numRecords;

for (int i = 0; i < numRecords; ++i) {

Record record = generateRandomRecord();

tree.insert(record);

}

cout << "=================================" << endl;

cout << "| Operation | Time |" << endl;

cout << "=================================" << endl;

string name\_list[5] = {};

double addTime = 0.0, deleteTime = 0.0, searchTime = 0.0;

for (int i = 0; i < 5; ++i) {

auto start = chrono::high\_resolution\_clock::now();

Record temp\_record = generateRandomRecord();

string temp\_name = temp\_record.name;

name\_list[i] = temp\_name;

tree.insert(temp\_record);

auto end = chrono::high\_resolution\_clock::now();

double oneElementAddTime = chrono::duration<double, milli>(end - start).count();

cout << left << setw(15) << "| Addition #" << i + 1 << " | " << setw(13) << setprecision(4) << oneElementAddTime << "|" << endl;

addTime += oneElementAddTime;

}

for (int i = 0; i < 5; ++i) {

auto start = chrono::high\_resolution\_clock::now();

tree.remove(name\_list[i]);

auto end = chrono::high\_resolution\_clock::now();

double oneElementSearchTime = chrono::duration<double, milli>(end - start).count();

cout << left << setw(15) << "| Deletion #" << i + 1 << " | " << setw(13) << setprecision(4) << oneElementSearchTime << "|" << endl;

searchTime += oneElementSearchTime;

}

for (int i = 0; i < 5; ++i) {

auto start = chrono::high\_resolution\_clock::now();

tree.search(name\_list[i]);

auto end = chrono::high\_resolution\_clock::now();

double oneElementDeleteTime = chrono::duration<double, milli>(end - start).count();

cout << left << setw(15) << "| Search #" << i + 1 << " | " << setw(13) << setprecision(4) << oneElementDeleteTime << "|" << endl;

deleteTime += oneElementDeleteTime;

}

cout << "=================================" << endl;

cout << "Average time:" << endl;

cout << "=================================" << endl;

cout << "| Operation | Avg. time |" << endl;

cout << "=================================" << endl;

cout << "| Add | " << fixed << setprecision(3) << setw(12) << addTime / 5 << " |" << endl;

cout << "| Search | " << fixed << setprecision(3) << setw(12) << searchTime / 5 << " |" << endl;

cout << "| Delete | " << fixed << setprecision(3) << setw(12) << deleteTime / 5 << " |" << endl;

cout << "=================================" << endl;

ofstream outputFile("btree.dot");

if (outputFile.is\_open()) {

tree.root->saveAsDot(outputFile);

outputFile.close();

cout << "Tree saved to file 'btree.dot'" << endl;

}

else {

cout << "Error" << endl;

}

system("pause");

return 0;

}