**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе №4**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: «**ДЕРЕВЬЯ ДВОИЧНОГО ПОИСКА»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 7307 | Шалугин Е.Д. Васильев А.В. |  |
| Преподаватель | Колинько П.Г. |  |

Санкт-Петербург

2019

Оглавление

[Цель работы 3](#__RefHeading___Toc188_4150899979)

[Задание 3](#__RefHeading___Toc190_4150899979)

[Обоснование по выбору размера хэш-таблицы и коэффициентов хэш-функции 3](#__RefHeading___Toc302_2376616454)

[Оценка временной сложности 3](#__RefHeading___Toc967_306870553)

[Результаты работы программы 3](#__RefHeading___Toc969_306870553)

[Код программы 6](#__RefHeading___Toc306_2376616454)

[Вывод 9](#__RefHeading___Toc308_2376616454)

# **Цель работы**

Научиться работать с деревьями двоичного поиска.

**Задание**

# Переделать программу, составленную по теме «Хеш-таблицы», под использование деревьев двоичного поиска, используя АВЛ-деревья.

Формула для вычислений: (A ∩ B) ⊕ C ∪ D \ E

средняя мощность множества: 32.

# **Оценка временной сложности. Описание АВЛ-д.**

# **Определение.**

# Коэффициентом сбалансированности называют некоторую константу k, на которую могут отличаться высоты левого и правого поддерева любого произвольного узла X. Таким образом АВЛ-дерево – это двоичное дерево поиска, для которого определен коэффициент сбалансированности k = 1.

# **Разница высот. Высота.**

# Из определения АВЛ-дерева, следует, что разница высот левого и правого поддерева любого произвольного узла Х лежит в диапозоне {-1, 0, 1}.

# Исходя из этого высота h АВЛ-дерева, содержащего n узлов может быть рассчитана по формуле:

# Таким образом можно сказать, что АВЛ-деревья являются наиболее эффективными в обработке ввиду того, что максимальное количество шагов, для обнаружения искомого узла равно высоте дерева, которая является минимальной среди всех существующих двоичных деревьев поиска и максимально приближена к высоте идеально сбалансированного дерева.

# **Время нахождения узла.**

# Время Т нахождения произвольного узла X АВЛ-дереве, содержащем n элементов может быть рассчитана по формуле:

# **Операции вставки и удаления.**

# Время выполнения операций вставки и удаления узлов напрямую зависит от скорости поиска узла по дереву и следовательно является минимальным по сравнению с другими видами двоичных деревьев поиска.

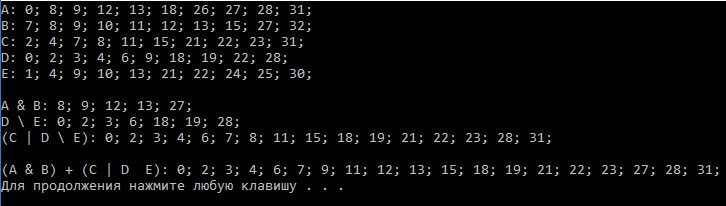
# **Возникающие сложности и способы их решения.**

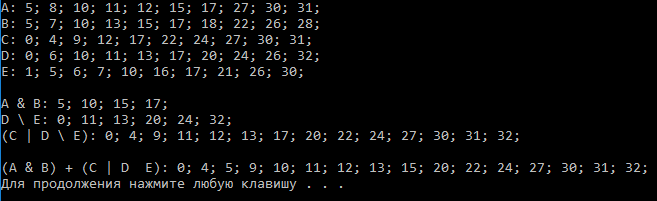
# В следствии выполнения операций вставки и удаления элементов может произойти разбалансировка дерева, т.е. для некоторого узла X высота левого и правого узла станет отличаться на 2, что приведет к потере свойств АВЛдерева. Для того, чтобы не допустить этого над АВЛ-деревом определена операция балансировки. Время выполнения операции балансировки фиксировано и она осуществляется путем поворота узлов и изменению связей в дереве.

# **Вывод.**

# Таким образом можно сказать, что АВЛ-деревья являются наиболее эффективными в обработке ввиду того, что максимальное количество шагов, для обнаружения искомого узла равно высоте дерева, которая является минимальной среди всех существующих двоичных деревьев поиска и максимально приближена к высоте идеально сбалансированного дерева.

# **Результаты работы программы**





# **Код программы**

#include "pch.h"

#include "Bst.h"

#include "Generator.h"

#include "LogicalOperation.h"

#include <iostream>

#include <ctime>

#include <windows.h>

using std::cin;

using std::cout;

using std::endl;

int main()

{ /\*

Bst A = generator::GenerateSet(100, 100);

cout << "Root is " << A.root->value << endl << endl << endl;

A.print\_Tree(A.root, 0);

\*/

std::srand(unsigned(std::time(0)));

Bst A, B, C, D, E, R, R1, R2, R3;

A = generator::GenerateSet(10, 32);

B = generator::GenerateSet(10, 32);

C = generator::GenerateSet(10, 32);

D = generator::GenerateSet(10, 32);

E = generator::GenerateSet(10, 32);

cout << "A: ";

A.displayForward();

cout << endl;

cout << "B: ";

B.displayForward();

cout << endl;

cout << "C: ";

C.displayForward();

cout << endl;

cout << "D: ";

D.displayForward();

cout << endl;

cout << "E: ";

E.displayForward();

cout << endl;

cout << endl;

// (A \ B \ C) + D v E

// 1 2 3 4

//(A & B) + (C | D \ E)

// 1 4 3 2

R1 = A & B;

cout << "A & B: ";

R1.displayForward();

cout << endl;

R2 = D - E;

cout << "D \\ E: ";

R2.displayForward();

cout << endl;

R2 = R2 | C;

cout << "(C | D \\ E): ";

R2.displayForward();

cout << endl;

cout << endl;

R = R1 + R2;

cout << "(A & B) + (C | D \ E): ";

R.displayForward();

//R = A - B;

//cout << "A \\ B: ";

//R.displayForward();

//cout << endl;

//R = R - C;

//cout << "A \\ B \\ C: ";

//R.displayForward();

//cout << endl;

//R = R ^ D;

//cout << "(A \\ B \\ C) + D: ";

//R.displayForward();

//cout << endl;

//cout << endl;

//R = R | E;

//cout << "(A \\ B \\ C) + D v E: ";

//R.displayForward();

cout << endl;

system("pause");

return 0;

1. }

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#pragma once

enum inWidthTask

{

conj,

disj,

sub,

copy

};

class LogicalOperation

{

public:

// Îáõîä äåðåâà À â øèðèíó, ñ âûïîëíåíèåì ñîîòâåòñâóþùåé îáðàáîòêè

static void inWidth(Node\* a, Bst\* b, Bst\* c, inWidthTask task)

{

if (a != nullptr)

{

inWidth(a->left, b, c, task);

switch (task)

{

case conj:

if ((\*b)[a->value])

c->root = c->insert(c->root, a->value);

break;

case disj:

case sub:

if (!(\*b)[a->value])

c->root = c->insert(c->root, a->value);

break;

case copy:

c->root = c->insert(c->root, a->value);

default:

break;

}

inWidth(a->right, b, c, task);

}

}

};

Bst operator&(Bst a, Bst b)

{

Bst\* c = new Bst();

LogicalOperation::inWidth(a.root, &b, c, conj);

return \*c;

}

Bst operator|(Bst a, Bst b)

{

Bst\* c = new Bst();

LogicalOperation::inWidth(b.root, nullptr, c, copy); // Êîïèðîâàòü b â c

LogicalOperation::inWidth(a.root, &b, c, disj);

return \*c;

}

Bst operator-(Bst a, Bst b)

{

Bst\* c = new Bst();

LogicalOperation::inWidth(a.root, &b, c, sub);

return \*c;

}

Bst operator+(Bst a, Bst b)

{

return (a | b) - (a & b);

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#pragma once

#include "Bst.h"

#include <cstdlib>

#include <iostream>

class generator

{

public:

static Bst GenerateSet(int size, int maxValue)

{

++maxValue;

int\* raw = new int[size];

for (int i = 0; i < size; ++i)

{

do

{

raw[i] = std::rand() % maxValue;

} while (repeatExists(raw, i, raw[i]));

}

// Maybe (size - 1)

return Bst(raw, size);

}

private:

static bool repeatExists(int\* arr, int size, int value)

{

for (int i = 0; i < size; ++i)

{

if (arr[i] == value) return true;

}

return false;

}

};

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

#pragma once

#include <iostream>

struct Node

{

Node \*left, \*right;

unsigned char height;

int value;

Node() : value(0), left(nullptr), right(nullptr), height(0) {}

Node(int value) : value(value), left(nullptr), right(nullptr), height(1) {}

};

class Bst

{

private:

enum DisplayDirection

{

forward,

backward

};

void inWidthDisplay(Node \*root, DisplayDirection dir)

{

if (dir == forward)

if (root != nullptr)

{

inWidthDisplay(root->left, forward);

std::cout << root->value << "; ";

inWidthDisplay(root->right, forward);

}

if (dir == backward)

if (root != nullptr)

{

inWidthDisplay(root->right, backward);

std::cout << root->value << "; ";

inWidthDisplay(root->left, backward);

}

}

public:

Node\* root;

Bst() {}

Bst(int\* data, int arraylength)

{

root = nullptr;

build(data, arraylength);

}

void build(int\* data, int arraylength) //Ñáîðêà ÄÄÏ èç ìàññèâà óçëîâ data

{

for (int i = 0; i < arraylength; ++i)

root = insert(root, data[i]);

}

unsigned char height(Node\* p)

{

return p ? p->height : 0;

}

int bfactor(Node\* p)

{

return height(p->right) - height(p->left);

}

void fixheight(Node\* p)

{

unsigned char hl = height(p->left);

unsigned char hr = height(p->right);

p->height = (hl > hr ? hl : hr) + 1;

}

Node\* rotateright(Node\* p) // ïðàâûé ïîâîðîò âîêðóã p

{

Node\* q = p->left;

p->left = q->right;

q->right = p;

fixheight(p);

fixheight(q);

return q;

}

Node\* rotateleft(Node\* q) // ëåâûé ïîâîðîò âîêðóã q

{

Node\* p = q->right;

q->right = p->left;

p->left = q;

fixheight(q);

fixheight(p);

return p;

}

Node\* balance(Node\* p) // áàëàíñèðîâêà óçëà p

{

fixheight(p);

if (bfactor(p) == 2)

{

if (bfactor(p->right) < 0)

p->right = rotateright(p->right);

return rotateleft(p);

}

if (bfactor(p) == -2)

{

if (bfactor(p->left) > 0)

p->left = rotateleft(p->left);

return rotateright(p);

}

return p; // áàëàíñèðîâêà íå íóæíà

}

Node\* insert(Node\* p, int k) // âñòàâêà êëþ÷à k â äåðåâî ñ êîðíåì p

{

if (!p) return new Node(k);

if (k < p->value)

p->left = insert(p->left, k);

else

p->right = insert(p->right, k);

return balance(p);

//return p;

}

void print\_Tree(Node \* p, int level)

{

if (p)

{

print\_Tree(p->right, level + 1);

for (int i = 0; i < level; i++) std::cout << " ";

std::cout << p->value << std::endl;

print\_Tree(p->left, level + 1);

}

}

bool operator[] (int value)

{

return search(this->root, value);

}

bool search(Node\* n, int value)

{

if (n == nullptr) return false;

if (n->value == value) return true;

if (n->value < value)

return search(n->right, value);

else

return search(n->left, value);

}

void displayForward()

{

inWidthDisplay(root, forward);

}

void displayBackward()

{

inWidthDisplay(root, backward);

}

};

# **Вывод**

1. В данной работе мы изучили АВЛ-деревья и методы работы с ними.

АВЛ-деревья являются наиболее эффективными в обработке ввиду того, что максимальное количество шагов, для обнаружения искомого узла равно высоте дерева, которая является минимальной среди всех существующих двоичных деревьев поиска и максимально приближена к высоте идеально сбалансированного дерева.