МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «Полоцкий государственный университет»

Факультет информационных технологий

Кафедра технологий программирования

**Отчёт по лабораторной работе № 6 по курсу**

**«Алгоритмы и структуры данных»**

Деревья. Сбалансированные по высоте деревья (АВЛ-деревья). 2-3

деревья. Б-деревья. Красно-черные деревья. Практическое применение.

Вариант №4

ВЫПОЛНИЛА студентка группы 21-ИТ-3

Демчишина А.Ю.

ПРОВЕРИЛ преподаватель

Виноградова А.Д.

Полоцк, 2022 г.

**ЦЕЛЬ РАБОТЫ:** ознакомиться с понятиями «Деревья», «АВЛ-деревья», «Б-деревья», «Красно-черные деревья», изучить основные алгоритмы их обработки, научиться применять полученные знания на практике.

**ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ:**

1. Определение понятия АВЛ-дерево.

АВЛ-дерево – сбалансированное двоичное дерево поиска с k=1.

1. Как осуществляется операция балансировки дерева?

Если после выполнения операции добавления или удаления, коэффициент

сбалансированности какого-либо узла АВЛ-дерева становиться равен 2, т. е. |h(Ti,R)-h(Ti, L)|=2, то необходимо выполнить операцию балансировки. Она осуществляется путем вращения (поворота) узлов – изменения связей в поддереве. Вращения не меняют свойств бинарного дерева поиска, и выполняются за константное время.

1. Типы вращений при балансировке.

Всего различают 4 их типа:

1. малое правое вращение;

2. большое правое вращение;

3. малое левое вращение;

4. большое левое вращение.

4. Определение понятия 2-3 дерево.

2-3-дерево — структура данных, являющаяся B-деревом Степени 1, страницы которого могут содержать только 2-вершины (вершины с одним полем и 2 детьми) и 3-вершины (вершины с 2 полями и 3 детьми).

1. Свойства 2-3 дерева.

Свойства:

• Все нелистовые вершины содержат одно поле и 2 поддерева или 2

поля и 3 поддерева.

• Все листовые вершины находятся на одном уровне (на нижнем

уровне) и содержат 1 или 2 поля.

• Все данные отсортированы (по принципу двоичного дерева поиска).

• Нелистовые вершины содержат одно или два поля, указывающие на

диапазон значений в их поддеревьях.

1. Опишите алгоритм вставки в 2-3дерево элемента с ключом.

Для того, чтобы вставить в дерево элемент с ключом key, нужно

действовать по алгоритму:

1. Если дерево пусто, то создать новую вершину, вставить ключ и

вернуть в качестве корня эту вершину, иначе

2. Если вершина является листом, то вставляем ключ в эту вершину и

если получили 3 ключа в вершине, то разделяем её, иначе

3. Сравниваем ключ key с первым ключом в вершине, и если key меньше

данного ключа, то идем в первое поддерево и переходим к пункту 2, иначе

4. Смотрим, если вершина содержит только 1 ключ (является 2-

вершиной), то идем в правое поддерево и переходим к пункту 2, иначе

5. Сравниваем ключ key со вторым ключом в вершине, и если key

меньше второго ключа, то идем в среднее поддерево и переходим к пункту 2, иначе

6. Идем в правое поддерево и переходим к пункту 2.

7. Определение понятия Б-дерево.

B-дерево (читается как Би-дерево) — это особый тип сбалансированного

дерева поиска, в котором каждый узел может содержать более одного ключа и

иметь более двух дочерних элементов.

8. Где используются Б-деревья?

Вторичные запоминающие устройства (жесткие диски, SSD) медленно работают с большим объемом данных. Людям захотелось сократить время доступа к физическим носителям информации, поэтому возникла потребность в таких структурах данных, которые способны это сделать.

9. Условия для балансировки красно-черных деревьев.

Каждая вершина красно-черного дерева окрашена либо в черный, либо в красный цвет, причем выполняются следующие условия:

• корень дерева окрашен в черный цвет;

• у красной вершины дети черные (если они есть);

• всякий путь от корня дерева к внешней вершине (листу) содержит одно и то же число черных вершин.

**ОПИСАНИЕ ПРОДЕЛАННОЙ РАБОТЫ:**

При выполнении работы я пользовался IDE Visual Studio Code. В результате выполнения работы был написан следующий код:

#include <iostream>

using namespace std;

// стркуктура узла

struct node

{

int data;

unsigned char height;

node \*left;

node \*right;

node(int n\_data)

{

data = n\_data;

left = right = 0;

height = 1;

}

};

// Возвращает height узла если он существует

unsigned char height(node \*p\_node)

{

return p\_node ? p\_node->height : 0;

}

// Возвращает фактор баланса между левым и правым

int balanceFactor(node \*p\_node)

{

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

}

// Исправляет высоту

void fix\_height(node \*p\_node)

{

unsigned char height\_left = height(p\_node->left);

unsigned char height\_right = height(p\_node->right);

p\_node->height = (height\_left > height\_right ? height\_left : height\_right) + 1;

}

// Меняе узлы местами -> малый поворот

node \*rotateRight(node \*p\_node)

{

node \*temp\_node = p\_node->left;

p\_node->left = temp\_node->right;

temp\_node->right = p\_node;

fix\_height(p\_node);

fix\_height(temp\_node);

return temp\_node;

}

// Меняе узлы местами -> малый поворот

node \*rotateLeft(node \*p\_node)

{

node \*temp\_node = p\_node->right;

p\_node->right = temp\_node->left;

temp\_node->left = p\_node;

fix\_height(p\_node);

fix\_height(temp\_node);

return temp\_node;

}

// Большие повороты

node \*balance(node \*p\_node)

{

fix\_height(p\_node);

if (balanceFactor(p\_node) == 2)

{

if (balanceFactor(p\_node->right) < 0)

{

p\_node->right = rotateRight(p\_node->right);

}

return rotateLeft(p\_node);

}

if (balanceFactor(p\_node) == -2)

{

if (balanceFactor(p\_node->left) > 0)

{

p\_node->left = rotateLeft(p\_node->left);

}

return rotateRight(p\_node);

}

return p\_node;

}

// Добавление узла

node \*insert(node \*p\_node, int data)

{

if (!p\_node)

return new node(data);

if (data < p\_node->data)

{

p\_node->left = insert(p\_node->left, data);

}

else

{

p\_node->right = insert(p\_node->right, data);

}

return balance(p\_node);

}

//прямой обход бинарного дерева

void pre\_order(node \*Node)

{

if (Node != NULL)

{

printf("%3ld", Node->data);

pre\_order(Node->left);

pre\_order(Node->right);

}

}

//обратный обход бинарного дерева

void post\_order(node \*Node)

{

if (Node != NULL)

{

post\_order(Node->left);

post\_order(Node->right);

printf("%3ld", Node->data);

}

}

//симметричный обход бинарного дерева

void symmetric\_order(node \*Node)

{

if (Node != NULL)

{

post\_order(Node->left);

printf("%3ld", Node->data);

post\_order(Node->right);

}

}

node \*findMin(node \*p) // поиск узла с минимальным ключом в дереве p

{

return p->left ? findMin(p->left) : p;

}

node \*removeMin(node \*p) // удаление узла с минимальным ключом из дерева p

{

if (p->left == 0)

return p->right;

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

return balance(p);

}

// Идем влево до упора что-бы найти минимальное

node \*find\_Min(node \*p\_node)

{

if (!p\_node)

{

cout << "No node";

return (p\_node);

}

else

{

while (p\_node->left != NULL)

{

p\_node = p\_node->left;

}

return p\_node;

}

}

// Идем вправо до упора что-бы найти максимальное

node \*findMax(node \*p\_node)

{

if (!p\_node)

{

cout << "No node";

return (p\_node);

}

else

{

while (p\_node->right != NULL)

{

p\_node = p\_node->right;

}

return p\_node;

}

}

node \*removeNode(node \*p\_node, int data)

{

if (!p\_node)

return 0;

if (data < p\_node->data)

{

p\_node->left = removeNode(p\_node->left, data);

}

else if (data > p\_node->data)

{

p\_node->right = removeNode(p\_node->right, data);

}

else

{

node \*q = p\_node->left;

node \*r = p\_node->right;

delete p\_node;

if (!r)

return q;

node \*min = findMin(r);

min->right = removeMin(r);

min->left = q;

return balance(min);

}

return balance(p\_node);

}

int main()

{

system("CLS");

system("Color 09");

int x = 0;

cout << "Input first node: ";

cin >> x;

node \*root = new node(x);

int task = -1;

while (task != 0)

{

cout << "\n~-~Menu~-~\n";

cout << "1. Insert node\n";

cout << "2. Delete node\n";

cout << "3. Output\n";

cout << "4. Min. and max.\n";

cout << "5. Height.\n";

cout << "0. Exit\n";

cin >> task;

switch (task)

{

case 1:

system("CLS");

cin >> x;

insert(root, x);

break;

case 2:

system("CLS");

cin >> x;

removeNode(root, x);

break;

case 3:

system("CLS");

cout << "\ninOrder ->\n";

pre\_order(root);

cout << "\ninPostOrder ->\n";

post\_order(root);

cout << "\ninPreOrder ->\n";

symmetric\_order(root);

break;

case 4:

system("CLS");

cout << "Min. -> " << find\_Min(root)->data << "\n";

cout << "Max. -> " << findMax(root)->data << "\n";

break;

case 5:

system("CLS");

fix\_height(root);

cout << height(root);

break;

case 0:

system("CLS");

system("Color 07");

break;

default:

cout << "!\* Check your input\n";

break;

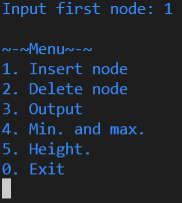
}

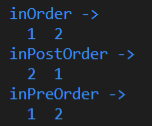
}

}

**РЕЗУЛЬТАТЫ ВЫПОЛЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ**:

Предыдущий код выдает следующие результаты:

 Главное меню. Содержит функции добавления элемента, удаления элемента, вывода дерева, нахождения максимального и минимального элемента, вывода высоты дерева и выход из программы.

****

Вывод элементов дерева. По 3 типам обходов: inOrder, inPostOrder, inPreOrder.

****Вывод минимального и максимального элементов дерева.

**Вывод:** в результате выполнения лабораторной работы было реализовано АВЛ-дерево. Также я научилась работать с деревьями на ЯП C++.