

**Министерство образования Республики Беларусь
Учреждение образования «Полоцкий государственный
университет»**

Кафедра технологий
программирования

Алгоритмы и структуры данных
Отчет по лабораторной работе №6
Вариант 11

Выполнил

Ланцев Евгений Николаевич.
21-ИТ-1, ФИТ

Проверил

преподаватель
Виноградова А.Д.

Полоцк
2022 г.

Лабораторная работа № 6

“Деревья. Сбалансированные по высоте деревья (АВЛ-деревья). 2-3 деревья. Б-деревья. Красно-черные деревья. Практическое применение.”

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 (ответы на контрольные вопросы):

1. Определение понятия АВЛ-дерево.

АВЛ-дерево – сбалансированное двоичное дерево поиска с $k=1$. Для его узлов определен коэффициент сбалансированности (balance factor).

2. Как осуществляется операция балансировки дерева?

Она осуществляется путем вращения (поворота) узлов – изменения связей в поддереве. Вращения не меняют свойств бинарного дерева поиска, и выполняются за константное время.

3. Типы вращений при балансировке.

- a. малое правое вращение;
- b. большое правое вращение;
- c. малое левое вращение;
- d. большое левое вращение;

4. Определение понятия 2-3 дерево.

2-3-дерево — структура данных, являющаяся В-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).

5. Свойства 2-3 дерева.

- 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.

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

- a. Если дерево пусто, то создать новую вершину, вставить ключ и вернуть в качестве корня эту вершину, иначе
- b. Если вершина является листом, то вставляем ключ в эту вершину и если получили 3 ключа в вершине, то разделяем её, иначе
- c. Сравниваем ключ key с первым ключом в вершине, и если key меньше данного ключа, то идем в первое поддереву и переходим к пункту b, иначе
- d. Смотрим, если вершина содержит только 1 ключ (является 2-вершиной), то идем в правое поддереву и переходим к пункту b, иначе
- e. Сравниваем ключ key со вторым ключом в вершине, и если key меньше второго ключа, то идем в среднее поддереву и переходим к пункту b, иначе
- f. Идем в правое поддереву и переходим к пункту b. Для примера вставим $keys = \{1, 2, 3, 4, 5, 6, 7\}$: При вставке $key=1$ мы имеем пустое дерево, а после получаем единственную вершину с единственным ключа $key=1$

7. Определение понятия Б-дерево.

В-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. Из-за этого свойства В-дерево называют сильно ветвящимся.

8. Где используются Б-деревья?

- В базах данных и файловых системах.
- Для хранения блоков данных (вторичные носители).
- Для многоуровневой индексации.

9. Условия для балансировки красно-черных деревьев.

- 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.

Вариант 11

jejikeh, 3 weeks ago | 1 author (jejikeh)

```
struct node {
    int data;
    unsigned char height;
    node* left;
    node* right;
    node(int n_data) {
        data = n_data;
        left = right = 0;
        height = 1;
    }
};
```

```
10 }
11
12 class Node {
13     constructor(data) {
14         this.data = data;
15         this.height = 1;
16         this.left = 0;
17         this.right = 0;
18     }
19 }
20
21 function Height(node) {
```

Рисунок 1 - Класс узла бинарного дерева

```
// Возвращает height узла если он существует
char height(node *p_node){
    return p_node ? p_node->height : 0;
}
```

```
// Возвращает фактор баланса между левым и правым
int balance_factor(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);
}
```

// Меняю узлы местами → малый поворот

```
node* rotate_right(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;
}
```

// Меняю узлы местами → малый поворот

```
node* rotate_left(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(balance_factor(p_node) == 2){
        if ( balance_factor(p_node->right) < 0){
            p_node->right = rotate_right(p_node->right);
        }
        return rotate_left(p_node);
    }
    if(balance_factor(p_node) == -2){
        if(balance_factor(p_node->left) > 0){
            p_node->left = rotate_left(p_node->left);
        }
        return rotate_right(p_node);
    }
    return p_node;
}
```

```
19 }
20
21 function Height(node) {
22     if (node) return node.height;
23     else return 0;
24 }
25
26 function BF(node) {
27     return Height(node.right) - Height(node.left);
28 }
29
30 function OverHeight(node) {
31     height_left = Height(node.left);
32     height_right = Height(node.right);
33     node.height = height_left > height_right ? height_left : height_right;
34 }
35
36 function RightRotation(node) {
37     y = node.left;
38     y.left = x.right;
39     y.right = node;
40     OverHeight(node);
41     OverHeight(y);
42     return y;
43 }
44
45 function LeftRotation(node) {
46     y = node.right;
47     y.right = x.left;
48     y.left = node;
49     OverHeight(node);
50     OverHeight(y);
51     return y;
52 }
53
54 function Balance(node) {
55     OverHeight(node);
56     if (BF(node) == 2) {
57         if (BF(node.right) < 0) {
58             node.right = RightRotation(node.right);
59         }
60         return LeftRotation(node);
61     }
62     if (BF(node) == -2) {
63         if (BF(node.left) > 0) {
64             node.left = LeftRotation(node.left);
65         }
66         return RightRotation(node);
67     }
68     return node;
69 }
70
71 function Insert(node, data) {
```

```

// если два, то вызов этой же функции для элемента
rl.question(
    `Выберете к какому элементу добавить узел, к ${currentNode.children[0].data}
    (index) => {
        if (index < 2) {
            this.add(data, currentNode.children[index]);
        } else {
            tasks();
        }
    }
);
}
}
display(currentNode) {
    console.log("-----");
    console.log("Корень : " + currentNode.data);
    for (let i = 0; i < currentNode.children.length; i++) {
        console.log("Ребенок : " + currentNode.children[i].data);
    }
    for (let i = 0; i < currentNode.children.length; i++) {
        this.display(currentNode.children[i]);
    }
}
}

```

```

// 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);
}

void in_order(node *p_node){
    if(p_node){
        in_order(p_node->left);
        std::cout << p_node->data << "\t";
        in_order(p_node->right);
    }
}

void in_preorder(node *p_node){
    if(p_node){
        std::cout << p_node->data << "\t";
        in_preorder(p_node->left);
        in_preorder(p_node->right);
    }
}

void in_postorder(node *p_node){
    if(p_node){

```

```

71 function Insert(node, data) {
72     if (!node) return new Node(data);
73     if (data < node.data) {
74         node.left = Insert(node.left, data);
75     } else {
76         node.right = Insert(node.right, data);
77     }
78     return Balance(node);
79 }
80
81 function SearchMin(node) {
82     return node.left ? SearchMin(node.left) : node;
83 }
84
85 function InOrder(node) {
86     if (node) {
87         InOrder(node.left);
88         console.log(node.data);
89         InOrder(node.right);
90     }
91 }
92
93 function InPreOrder(node) {
94     if (node) {
95         console.log(node.data);
96         InPreOrder(node.right);
97         InPreOrder(node.left);
98     }

```

```

void in_postorder(node *p_node){
    if(p_node){
        in_postorder(p_node->left);
        in_postorder(p_node->right);
        std::cout << p_node->data << "\t";
    }
}

node* findmin(node* p) // поиск узла с минимальным ключом
{
    return p->left?findmin(p->left):p;
}

node* removemin(node* p) // удаление узла с минимальным
{
    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){
        std::cout << "No node";
        return(p_node);
    }else {
        while(p_node->left)
            p_node = p_node->left;
    }
}

```

```

101 function InPostOrder(node) {
102     if (node.data) {
103         InPostOrder(node.left);
104         InPostOrder(node.right);
105         console.log(node.data);
106     }
107 }
108
109 function DeleteMin(node) {
110     if (node.left == 0) return node.right;
111     node.left = DeleteMin(node.left);
112     return Balance(node);
113 }
114
115 function Delete(node, data) {
116     if (!node) return 0;
117     if (data < node.data) {
118         node.left = Delete(node.left, data);
119     } else if (data > node.data) {
120         node.right = Delete(node.right, data);
121     } else {
122         y = node.left;
123         z = node.right;
124         delete node;
125         if (!z) return y;
126         min = SearchMin(z);
127         min.right = DeleteMin(z);
128         min.left = y;
129         return Balance(min);
130     }
131 }

```

```

node* find_max(node* p_node){
    if(!p_node){
        std::cout << "No node";
        return(p_node);
    }else {
        while(p_node→right ≠ NULL){
            p_node = p_node→right;
        }
        return p_node;
    }
}

node* remove_node(node* p_node, int data){
    if(!p_node) return 0;
    if(data < p_node→data){
        p_node→left = remove_node(p_node→left,data);
    }else if( data > p_node→data){
        p_node→right = remove_node(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);
}

```

Рисунок 2 - Базовые методы для работы и балансировка бинарного дерева

4-Find min and max
Task -> 1

Add node -> 24
1-Add node
2-Delete node
3-Print list
4-Find min and max
Task -> 3

IN_ORDER ->
3
IN_POSTORDER ->
3
IN_PREORDER ->
3
1-Add node
2-Delete node
3-Print list
4-Find min and max
Task -> 5
1-Add node
2-Delete node
3-Print list
4-Find min and max
Task -> 4
MIN -> 3
MAX -> 3
1-Add node
2-Delete node
3-Print list
4-Find min and max
Task -> ■

Рисунок 3 - Результат работы программы