ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(национальный исследовательский университет)»

Факультет №3 «Системы управления, информатика и электроэнергетика»

Кафедра № 304 «Вычислительные машины, системы и сети»

Отчёт по лабораторной работе

по дисциплине: «Структуры и алгоритмы обработки данных»

на тему: «Бинарные деревья поиска»

Выполнили:

студенты группы М3О-210Б-23
Фомин В. А.

Миронов А. Д.

Проверила:

Дмитриева Е. А.

Москва 2024 г.

Задание (вариант 7)

Лабораторная работа «Бинарные деревья поиска»

Задание

- Случайным образом сгенерировать массив размерностью 20-25 элементов, повторные значения не допустимы.
- Реализовать функции вставки, поиска, удаления узла, обхода дерева, вывода дерева на экран, нахождения высоты дерева и количества узлов.
- Реализовать дополнительно функцию в соответствии с вариантом: *T* – тип ключей, *D* – диапазон изменения значений ключей.
- Для набора значений из пункта 1 построить рандомизированное дерево, сравнить высоты бинарного и рандомизированного дерева.

№	*T*	*D*	Функция
1	`int`	[100; 200]	Подсчет суммы длин путей от корня до каждого из узлов, содержащих четные числа
2	`char`	[a..z, A..Z]	Подсчет количества гласных в листьях
3	`int`	[0; 100]	Подсчет количества нечетных чисел в узлах, имеющих ровно два поддерева
4	`int`	[-50; 50]	Подсчет суммы четных отрицательных чисел в узлах, поддеревья которых содержат не более 4 узлов
5	`int`	[0; 100]	Определить сумму четных чисел
6	`char`	[a..z, A..Z]	Подсчет количества согласных в узлах, высота поддеревьев которых одинакова
7	`char`	[a..z, A..Z]	Определить, каких букв в дереве больше - гласных или согласных
8	`char`	[a..z, A..Z]	Определить число узлов в левом и правом поддеревьях
9	`int`	[1; 90]	Определить два минимальных элемента, два максимальных элемента.
10	`int`	[-100; 100]	Определить, каких чисел больше - положительных или отрицательных.
11	`int`	[-50; 50]	Определить сумму элементов, кратных 5

Код программы

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cctype>

using namespace std;

// Структура узла дерева
struct TreeNode {
   char data;           // Данные узла (символ)
   TreeNode* left;      // Указатель на левое поддерево
   TreeNode* right;     // Указатель на правое поддерево
   TreeNode* p;         // Указатель на родительский узел
};

// Структура бинарного дерева
struct BST {
   TreeNode* root;      // Указатель на корневой узел дерева
};

// Проверяет, является ли символ гласной
bool isVowel(char ch);

// Возвращает узел с минимальным значением в поддереве
TreeNode* TreeMinimum(TreeNode* node);

// Поиск узла с заданным ключом в дереве
TreeNode* TreeSearch(BST* tree, char key);

// Вставка нового узла в бинарное дерево поиска
void TreeInsert(BST* T, char data);

// Замена одного поддерева другим
void Transplant(BST* T, TreeNode* u, TreeNode* v);

// Удаление узла из бинарного дерева поиска
void TreeDelete(BST* T, TreeNode* node);

// Прямой обход дерева (Preorder)
void PreorderTraversal(TreeNode* node);

// Симметричный обход дерева (Inorder)
void InorderTraversal(TreeNode* node);

// Обратный обход дерева (Postorder)
void PostorderTraversal(TreeNode* node);

// Печать дерева с отступами, визуализация структуры
void PrintTree(TreeNode* node, int depth = 0);
```

```
// Подсчет количества гласных и согласных в дереве
void CountVowelsAndConsonants(TreeNode* node, int& vowels, int& consonants);

// Генерация массива случайных символов (a-z)
void RandomChars(char* arr, int size);

// Поворот дерева вправо для рандомизированного дерева поиска
TreeNode* RotateRightRST(TreeNode* y);

// Поворот дерева влево для рандомизированного дерева поиска
TreeNode* RotateLeftRST(TreeNode* y);

// Получение размера поддерева (количество узлов)
int GetSizeRST(TreeNode* node);

// Вставка узла в рандомизированное дерево поиска
TreeNode* TreeInsertRST(TreeNode* node, char key);

// Вставка узла в рандомизированное дерево поиска (функция-обертка)
void InsertRST(BST* T, char key);

// Получение высоты дерева
int GetTreeHeight(TreeNode* node);

// Печать высоты дерева
void PrintTreeHeight(TreeNode* root);

// Отображение меню для выбора действий
void ShowMenu();

int main() {
   system("chcp 65001"); // Установка кодировки для корректного вывода на
русском
   srand(time(NULL)); // Инициализация генератора случайных чисел

   BST tree = {nullptr}; // Инициализация пустого дерева
   BST RST = {nullptr};  // Инициализация пустого рандомизированного дерева

   int size = 25; // Размер массива случайных символов
   char StartArr[size];
   RandomChars(StartArr, size); // Заполнение массива случайными уникальными
символами

   // Вставка символов в оба дерева
   for (int i = 0; i < size; i++) {
       InsertRST(&RST, StartArr[i]);   // Рандомизированное дерево
       TreeInsert(&tree, StartArr[i]); // Обычное дерево
   }

   // Печать рандомизированного дерева и его высоты
   cout << "Рандомизированное дерево:" << endl;
   PrintTree(RST.root);
   cout << "Высота RST: ";
```

```
PrintTreeHeight(RST.root);

int option = -1; // Выбор пользователя

while (option != 0) {
    ShowMenu(); // Отображение меню
    cout << "Введите номер операции: ";
    cin >> option;

    switch (option) {
        case 1: { // Печать дерева
            if (tree.root == nullptr) {
                cout << "Дерево пустое." << endl;
            } else {
                PrintTree(tree.root);
            }
            break;
        }
        case 2: { // Прямой обход дерева (Preorder)
            if (tree.root == nullptr) {
                cout << "Дерево пустое." << endl;
            } else {
                cout << "Прямой обход (Preorder): ";
                PreorderTraversal(tree.root);
                cout << endl;
            }
            break;
        }
        case 3: { // Симметричный обход дерева (Inorder)
            if (tree.root == nullptr) {
                cout << "Дерево пустое." << endl;
            } else {
                cout << "Симметричный обход (Inorder): ";
                InorderTraversal(tree.root);
                cout << endl;
            }
            break;
        }
        case 4: { // Обратный обход дерева (Postorder)
            if (tree.root == nullptr) {
                cout << "Дерево пустое." << endl;
            } else {
                cout << "Обратный обход (Postorder): ";
                PostorderTraversal(tree.root);
                cout << endl;
            }
            break;
        }
        case 5: { // Удаление узла
            if (tree.root == nullptr) {
                cout << "Дерево пустое. Удаление невозможно." << endl;
            } else {
                cout << "Введите символ узла для удаления: ";
```

```
            char data;
            cin >> data;
            TreeNode* node = TreeSearch(&tree, data); // Поиск узла
            if (node == nullptr) {
                cout << "Узел не найден." << endl;
            } else {
                TreeDelete(&tree, node); // Удаление узла
                cout << "Узел удалён." << endl;
                PrintTree(tree.root); // Вывод дерева после удаления
            }
        }
        break;
    }
    case 6: { // Поиск узла
        if (tree.root == nullptr) {
            cout << "Дерево пустое." << endl;
        } else {
            cout << "Введите символ для поиска: ";
            char data;
            cin >> data;
            TreeNode* node = TreeSearch(&tree, data); // Поиск узла
            if (node == nullptr) {
                cout << "Узел не найден." << endl;
            } else {
                cout << "Узел найден: " << node->data << endl;
            }
        }
        break;
    }
    case 7: { // Вставка узла
        cout << "Введите символ для вставки: ";
        char data;
        cin >> data;
        TreeInsert(&tree, data); // Вставка узла
        cout << "Узел добавлен." << endl;
        break;
    }
    case 8: { // Подсчет гласных и согласных
        if (tree.root == nullptr) {
            cout << "Дерево пустое." << endl;
        } else {
            int vowels = 0, consonants = 0;
            CountVowelsAndConsonants(tree.root, vowels, consonants);
            cout << "Количество гласных: " << vowels << endl;
            cout << "Количество согласных: " << consonants << endl;
        }
        break;
    }
    case 9: { // Вывод высоты дерева
        PrintTreeHeight(tree.root);
        break;
    }
    case 0: // Выход
```

```
            cout << "Выход из программы..." << endl;
            break;
        default: // Неверный ввод
            cout << "Неверный выбор. Пожалуйста, попробуйте снова." << endl;
    }
}

return 0;
}

bool isVowel(char ch){
    ch = tolower(ch);
    return (ch == 'a') || (ch == 'e') ||(ch == 'i') ||(ch == 'o') ||(ch == 'u');
}

TreeNode* TreeMinimum(TreeNode* node){
    while(node->left!= nullptr){
        node = node->left;
    }
    return node;
}

TreeNode* TreeSearch(BST* tree, char key) {
    TreeNode* temp = tree->root;
    while (temp != nullptr) {
        if (key == temp->data) {
            return temp; // Узел найден
        } else if (key < temp->data) {
            temp = temp->left; // Идем в левое поддерево
        } else {
            temp = temp->right; // Идем в правое поддерево
        }
    }
    return nullptr; // Узел не найден
}

void TreeInsert(BST* T, char data) {
    TreeNode* newNode = new TreeNode{data, nullptr, nullptr, nullptr};
    TreeNode* parent = nullptr;
    TreeNode* current = T->root;

    while (current != nullptr) {
        parent = current;
        current = (data < current->data) ? current->left : current->right;
    }

    newNode->p = parent;
    if (parent == nullptr) {
        T->root = newNode; // Дерево было пустым
    } else if (data < parent->data) {
        parent->left = newNode;
    } else {
        parent->right = newNode;
```

```
    }
}

void Transplant(BST* T, TreeNode* u, TreeNode* v) {
    if (u->p == nullptr) { // u — корень дерева
        T->root = v;
    } else if (u == u->p->left) { // u — левый дочерний узел
        u->p->left = v;
    } else { // u — правый дочерний узел
        u->p->right = v;
    }
    if (v != nullptr) {
        v->p = u->p; // Обновление родителя для v
    }
}

void TreeDelete(BST* T, TreeNode* node) {
    if (node->left == nullptr) {
        // Если нет левого поддерева
        Transplant(T, node, node->right);
    } else if (node->right == nullptr) {
        // Если нет правого поддерева
        Transplant(T, node, node->left);
    } else {
        // Найти минимальный элемент в правом поддереве
        TreeNode* y = TreeMinimum(node->right);
        if (y->p != node) {
            // Если y — не непосредственный потомок, переместить его
            Transplant(T, y, y->right);
            y->right = node->right;
            if (y->right != nullptr) y->right->p = y;
        }
        Transplant(T, node, y);
        y->left = node->left;
        if (y->left != nullptr) y->left->p = y;
    }
    delete node;
}

// Прямой обход (Preorder): Узел -> Левое -> Правое
void PreorderTraversal(TreeNode* node){
    if(node == nullptr) return;
    cout << node->data << " ";
    PreorderTraversal(node->left);
    PreorderTraversal(node->right);
}

// Симметричный обход (Inorder): Левое -> Узел -> Правое
void InorderTraversal(TreeNode* node){
    if(node == nullptr) return;
    InorderTraversal(node->left);
    cout << node->data << " ";
    InorderTraversal(node->right);
```

```
}

// Обратный обход (Postorder): Левое -> Правое -> Узел
void PostorderTraversal(TreeNode* node){
   if(node == nullptr) return;
   PostorderTraversal(node->left);
   PostorderTraversal(node->right);
   cout << node->data << " ";
}

void PrintTree(TreeNode* node, int depth){
   if(node!= nullptr){
       PrintTree(node->right, depth+1);
       for(int i = 0; i < depth*4; i++){
           cout << " ";
       }
       cout << node->data << endl;
       PrintTree(node->left, depth+1);
   }
}

void CountVowelsAndConsonants(TreeNode* node, int& vowels, int& consonants){
   if(node == nullptr) return;
   if(isalpha(node->data)) {
       if(isVowel(node->data)){
           vowels++;
       } else{
           consonants++;
       }
   }
   CountVowelsAndConsonants(node->left, vowels, consonants);
   CountVowelsAndConsonants(node->right, vowels, consonants);
}

void RandomChars(char* arr, int size) {
   char min = 'a';
   char max = 'z';
   bool used[26] = {false};

   int count = 0;
   while (count < size) {
       char randomChar = min + rand() % (max - min + 1);
       if (!used[randomChar - min]) {
           used[randomChar - min] = true;
           arr[count++] = randomChar;
       }
   }
}

TreeNode* RotateRightRST(TreeNode* y) {
   TreeNode* x = y->left;
   if (x == nullptr) return y;
```

```
    y->left = x->right;
    if (x->right) x->right->p = y;

    x->right = y;
    x->p = y->p;
    y->p = x;

    return x;
}

TreeNode* RotateLeftRST(TreeNode* y) {
    TreeNode* x = y->right;
    if (x == nullptr) return y;

    y->right = x->left;
    if (x->left) x->left->p = y;

    x->left = y;
    x->p = y->p;
    y->p = x;

    return x;
}


int GetSizeRST(TreeNode* node){
    if(node == nullptr) return 0;
    return 1 + GetSizeRST(node->left) + GetSizeRST(node->right);
}

TreeNode* TreeInsertRST(TreeNode* node, char key) {
    if (node == nullptr) {
        return new TreeNode{key, nullptr, nullptr, nullptr};
    }
    if (rand() % (GetSizeRST(node) + 1) == 0) {
        // Вставка в корень
        if (key < node->data) {
            node->left = TreeInsertRST(node->left, key);
            if (node->left) node->left->p = node;
            return RotateRightRST(node);
        } else {
            node->right = TreeInsertRST(node->right, key);
            if (node->right) node->right->p = node;
            return RotateLeftRST(node);
        }
    }
    // Обычная вставка
    if (key < node->data) {
        node->left = TreeInsertRST(node->left, key);
        if (node->left) node->left->p = node;
    } else {
        node->right = TreeInsertRST(node->right, key);
        if (node->right) node->right->p = node;
```

```
    }
    return node;
}

void InsertRST(BST* T, char key) {
    T->root = TreeInsertRST(T->root, key);
    if (T->root) T->root->p = nullptr; // Корень дерева не имеет родителя
}

int GetTreeHeight(TreeNode* node) {
    if (node == nullptr) {
        return 0; // Базовый случай: высота пустого поддерева равна 0
    }
    int leftHeight = GetTreeHeight(node->left);  // Высота левого поддерева
    int rightHeight = GetTreeHeight(node->right); // Высота правого поддерева

    return 1 + max(leftHeight, rightHeight); // Добавляем текущий уровень
}

void PrintTreeHeight(TreeNode* root) {
    if (root == nullptr) {
        cout << "Дерево пустое, высота равна 0." << endl;
    } else {
        int height = GetTreeHeight(root);
        cout << "Высота дерева: " << height << endl;
    }
}

void ShowMenu() {
    cout << "\nМеню:" << endl;
    cout << "1. Печать дерева" << endl;
    cout << "2. Прямой обход дерева" << endl;
    cout << "3. Симметричный обход дерева" << endl;
    cout << "4. Обратный обход дерева" << endl;
    cout << "5. Удалить узел" << endl;
    cout << "6. Поиск узла" << endl;
    cout << "7. Вставка узла" << endl;
    cout << "8. Сравнение числа согласных и гласных" << endl;
    cout << "9. Вывод высоты дерева" << endl;
    cout << "0. Выход" << endl;
}
```

Результаты:

```
Рандомизированное дерево:
            y
        t
            s
    r
k
        j
            i
    f
            e
        b
Высота RST: Высота дерева: 4
```

Бинарное:

```
          y
     t
               s
          r
k
          j
     i
          f
               e
                    b
```

```
Высота дерева: 5
```

```
Введите номер операции:4
 4
Обратный обход (Postorder): b e f j i s r y t k
```

```
Введите номер операции:3
 3
Симметричный обход (Inorder): b e f i j k r s t y
```

```
Введите номер операции:2
 2
Прямой обход (Preorder): k i f e b j t r s y
```

```
Введите символ узла для удаления:i
 i
Узел удалён.
        y
    t
            s
        r
k
    j
        f
            e
                b
```

Добавление узла p

```
 1
          y
     t
                s
          r
                p
k
     j
          f
                e
                     b
```

Вывод

В ходе работы были разработаны основные функции работы с бинарными и рандомизированными деревьями, такие как вставки, поиска, удаления узла, обхода дерева, вывода дерева на экран, нахождения высоты дерева и количества узлов. Также установлено, что в случае заполнения бинарного дерева случайно сгенерированной последовательностью, и заполнения рандомизированного дерева той же последовательностью, их высоты почти не отличаются.