Министерство науки и высшего образования

Пензенский государственный университет

Кафедра «Вычислительная техника»

ОТЧЕТ

по лабораторной работе №4

по дисциплине: "Логика и основы алгоритмизации в инженерных задачах"

на тему: "Бинарное дерево поиска"

Выполнили:

студенты группы 23ВВВ4

Королёв Д.В.

Алешин К.А.

Приняли:

Юрова О.В.

Деев М.В.

Пенза, 2024

Общие сведения

Бинарные деревья – это деревья, у каждого узла которого возможно наличие только двух сыновей. Двоичные деревья являются упорядоченными.

Задание 1. - Алгоритм поиска

Листинг

```
#include <iostream>
#include <cstdint>
#include <cstdlib>


struct Node
{
int data;
Node* left;
Node* right;

};

struct Node* root;

struct Node* CreateTree(struct Node* root, struct
Node* r, int data)
{
if (r == NULL)
{
r = (struct Node*)malloc(sizeof(struct Node));
if (r == NULL)
{
printf("Error allocate memory");
exit(0);
}

r->left = NULL;
r->right = NULL;
r->data = data;
```

```
if (root == NULL) return r;

if (data > root->data) root->left = r;
else root->right = r;
return r;
}

if (data > r->data)
CreateTree(r, r->left, data);
else
CreateTree(r, r->right, data);

return root;
}

void print_tree(struct Node* r, int l)
{

if (r == NULL)
{
return;
}

print_tree(r->right, l + 1);
for (int i = 0; i < l; i++)
{
printf(" ");
}

printf("%d\n", r->data);
print_tree(r->left, l + 1);
}

bool find_value(Node* root, int value)
{
if (!root)
{
return false;
}
else if (root->data == value)
```

```
    {
    return true;
    }

    return find_value(root->right, value) ||
find_value(root->left, value);
    }


    int main()
    {
    setlocale(LC_ALL, "");
    int D, start = 1;

    root = NULL;
    printf("-1 - for break\n");
    while (start)
    {
    printf("Enter number: ");
    scanf_s("%d", &D);
    if (D == -1)
    {
    printf("created tree finished\n\n");
    start = 0;
    }
    else
    root = CreateTree(root, root, D);

    }
    print_tree(root, 0);
    std::cout << "Enter value for search: ";
    int value;
    std::cin >> value;
    std::cout << std::endl;

    if (find_value(root, value))
    {
    std::cout << "True\n";
    }
    else
```

```
{
std::cout << "False\n";
}

//scanf_s("%d", &D);
return 0;
}
```

Результат работы программы

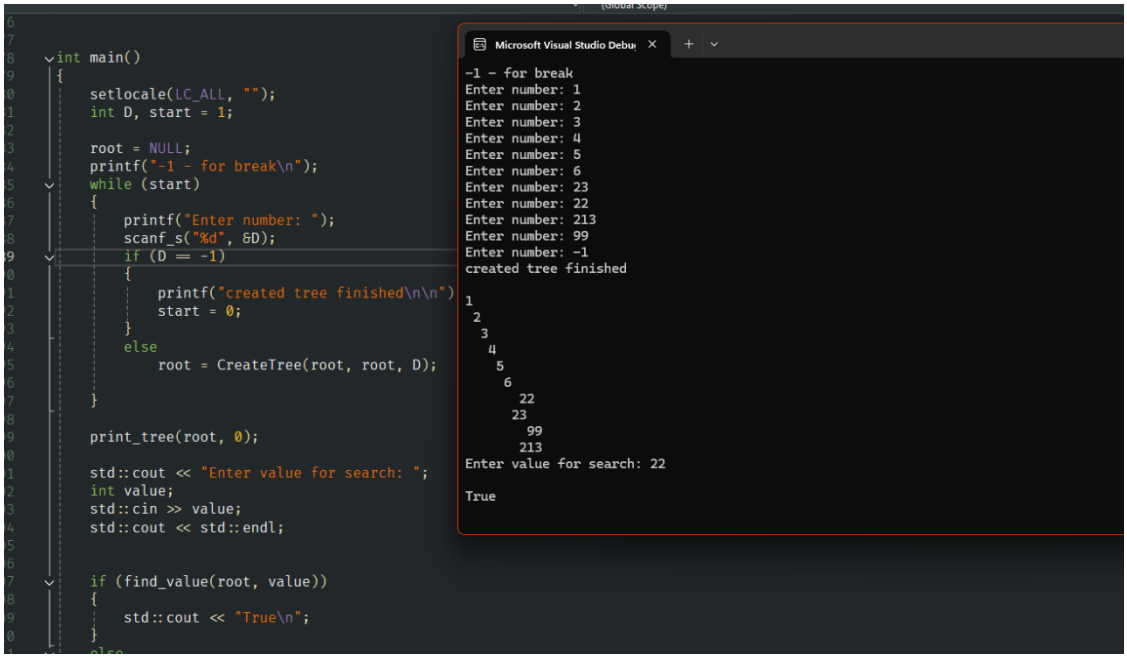


Рисунок № 1

Задание 2. - Алгоритм посчета вхождения элемента в дерево.

Листинг

```
#include <iostream>
#include <cstdint>
#include <cstdlib>


struct Node
{
int data;
Node* left;
Node* right;
```

```
};

struct Node* root;

struct Node* CreateTree(struct Node* root, struct Node* r,
int data)
{
if (r == NULL)
{
r = (struct Node*)malloc(sizeof(struct Node));
if (r == NULL)
{
printf("Ошибка выделения памяти");
exit(0);
}

r->left = NULL;
r->right = NULL;
r->data = data;
if (root == NULL) return r;

if (data > root->data) root->left = r;
else root->right = r;
return r;
}

if (data > r->data)
CreateTree(r, r->left, data);
else
CreateTree(r, r->right, data);

return root;
}

void print_tree(struct Node* r, int l)
{

if (r == NULL)
{
return;
```

```
}

print_tree(r->right, l + 1);
for (int i = 0; i < l; i++)
{
printf(" ");
}

printf("%d\n", r->data);
print_tree(r->left, l + 1);
}

int find_count_value(Node* root, int value, int counter =
0)
{

if (!root)
{
return counter;
}
else if (root->data == value)
{
counter++;
}
counter = find_count_value(root->right, value, counter);
counter = find_count_value(root->left, value, counter);

return counter;
}


int main()
{
setlocale(LC_ALL, "");
int D, start = 1;

root = NULL;
printf("-1 - окончание построения дерева\n");
while (start)
{
```

```
printf("Введите число: ");
scanf_s("%d", &D);
if (D == -1)
{
printf("Построение дерева окончено\n\n");
start = 0;
}
else
root = CreateTree(root, root, D);

}

print_tree(root, 0);

std::cout << "Введите число для поска: ";
int value;
std::cin >> value;
std::cout << std::endl;


std::cout << find_count_value(root, value);

return 0;
}
```

Результат работы программы

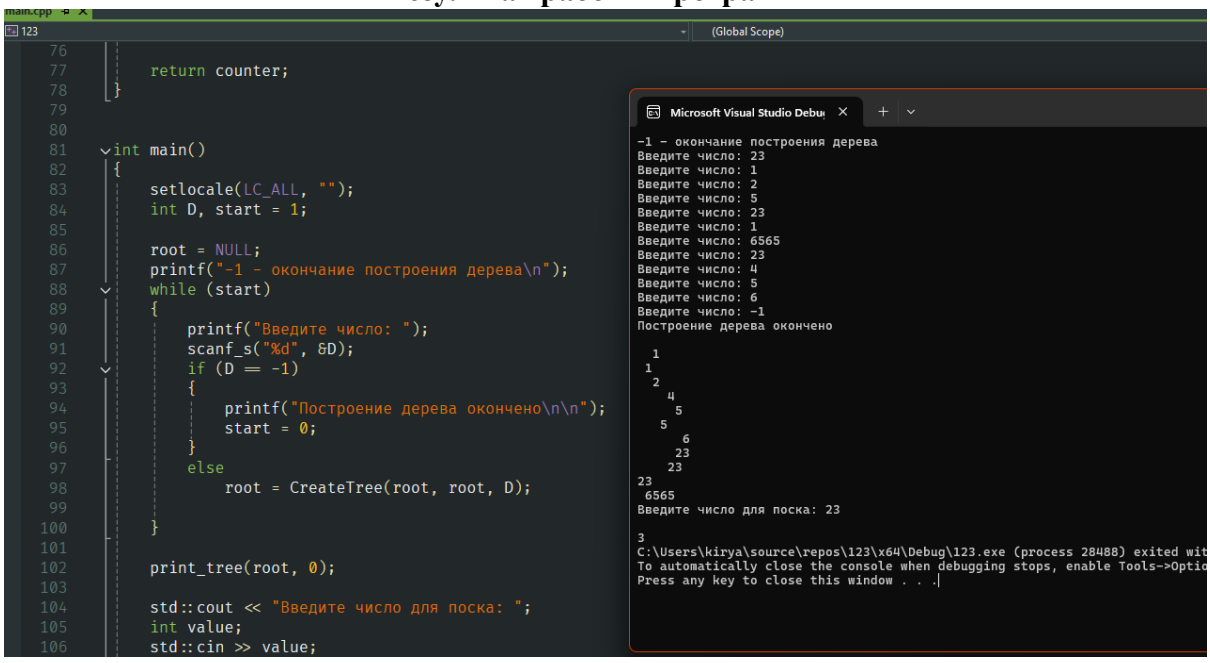


Рисунок 2)

Задание 3. - процедура добавления не повторяющихся элементов.

Листинг

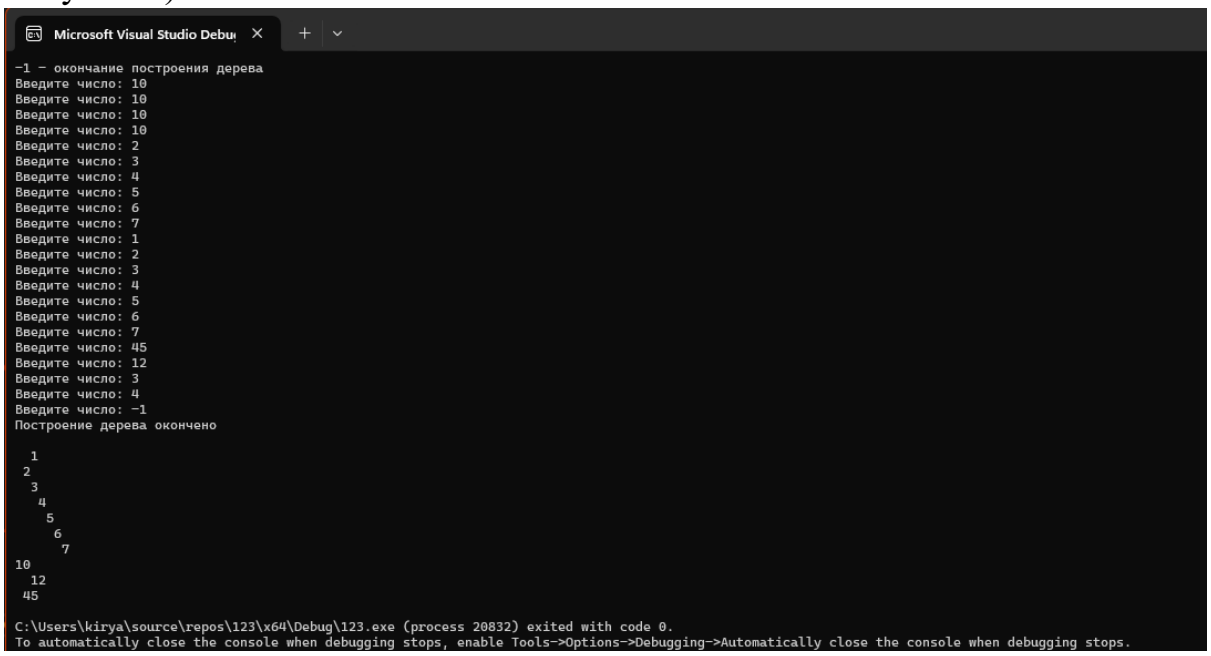
```
#include <iostream>
#include <cstdint>
#include <cstdlib>

struct Node
{
int data;
Node* left;
Node* right;

};


struct Node* root;
struct Node* CreateTree(struct Node* root, struct Node* r,
int data)
{
if (r == NULL)
{
r = (struct Node*)malloc(sizeof(struct Node));
if (r == NULL)
{
printf("Ошибка выделения памяти");
exit(0);
}

r->left = NULL;
r->right = NULL;
r->data = data;
if (root == NULL) return r;

if (data > root->data) root->left = r;
else root->right = r;
return r;
```

```
}

if (data == r->data) { return root; }
else if (data > r->data)
CreateTree(r, r->left, data);
else
CreateTree(r, r->right, data);

return root;
}

void print_tree(struct Node* r, int l)
{

if (r == NULL)
{
return;
}

print_tree(r->right, l + 1);
for (int i = 0; i < l; i++)
{
printf(" ");
}

printf("%d\n", r->data);
print_tree(r->left, l + 1);
}


int main()
{
setlocale(LC_ALL, "");
int D, start = 1;

root = NULL;
printf("-1 - окончание построения дерева\n");
while (start)
{
printf("Введите число: ");
scanf_s("%d", &D);
```

```
if (D == -1)
{
printf("Построение дерева окончено\n\n");
start = 0;
}
else
root = CreateTree(root, root, D);

}

print_tree(root, 0);

return 0;
}
```

Результат работы программы

Рисунок 3)



Задание 4. - Оценка сложности процедуры поиска по значению

Сложность процедуры поиска по значению в бинарном дереве поиска в среднем случае составляет $O(\log n)$. В этом случае дерево сбалансировано, и высота дерева приблизительно равна $\log n$, где n— количество узлов в дереве.

Однако в наихудшем случае, когда дерево становится вырожденным (например, все узлы — правые потомки одного перешагиваемого узла), сложность поиска может увеличиться до $O(n)$
. Это происходит, когда бинарное дерево поиска теряет свою сбалансированность и становится похожим на связный список.

Для поддержания дерева в сбалансированном состоянии часто используют самобалансирующиеся деревья, такие как красно-черные деревья или AVL-деревья, в которых сложность поиска остаётся $O(\log n)$ в худшем случае.

Вывод - были получены навыки реализации бинарного дерева, также процедур обхода дерева, нахождения элементов, подсчет элементов, оценка сложности алгоритма и его процедур.