Звіт

Автор: Момот Р. КІТ-119а

Дата: 11.05.2020

ЛАБОРАТОРНА РОБОТА № 14. ВИКОРИСТАННЯ ДЕРЕВ

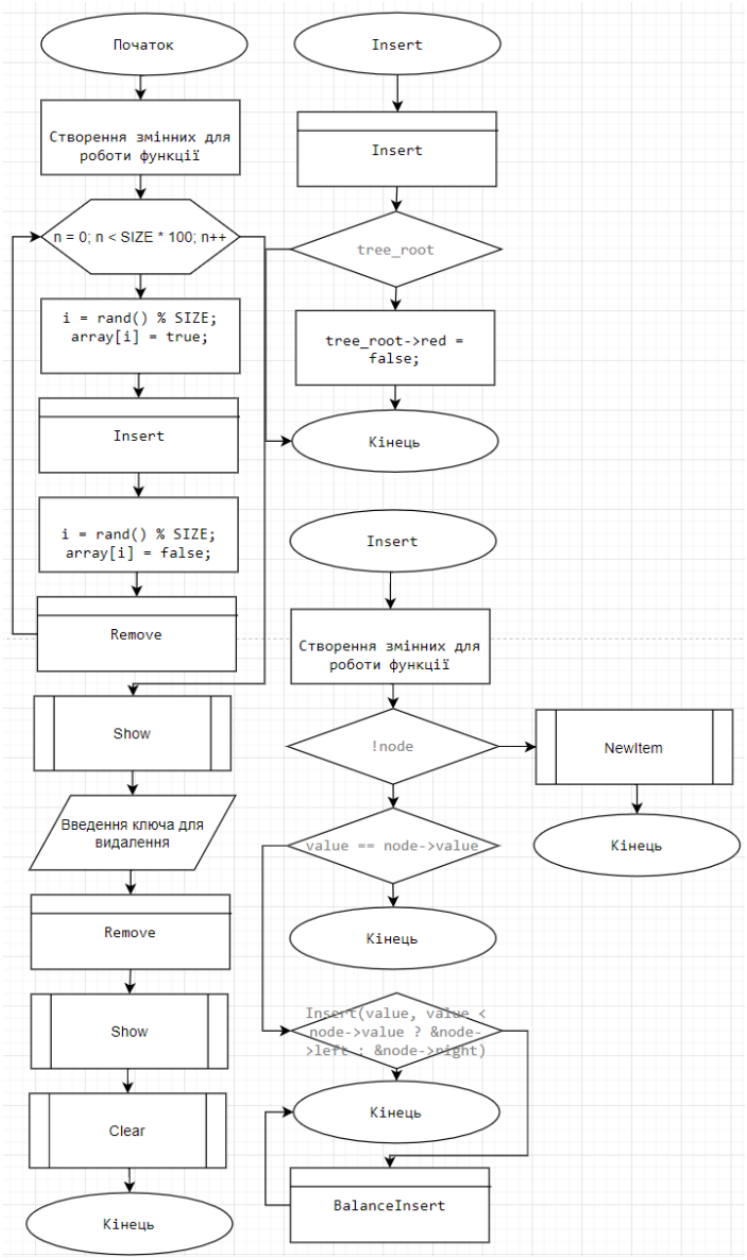
Мета: набути досвід практичної роботи вирішення задач з використанням бінарних дерев.

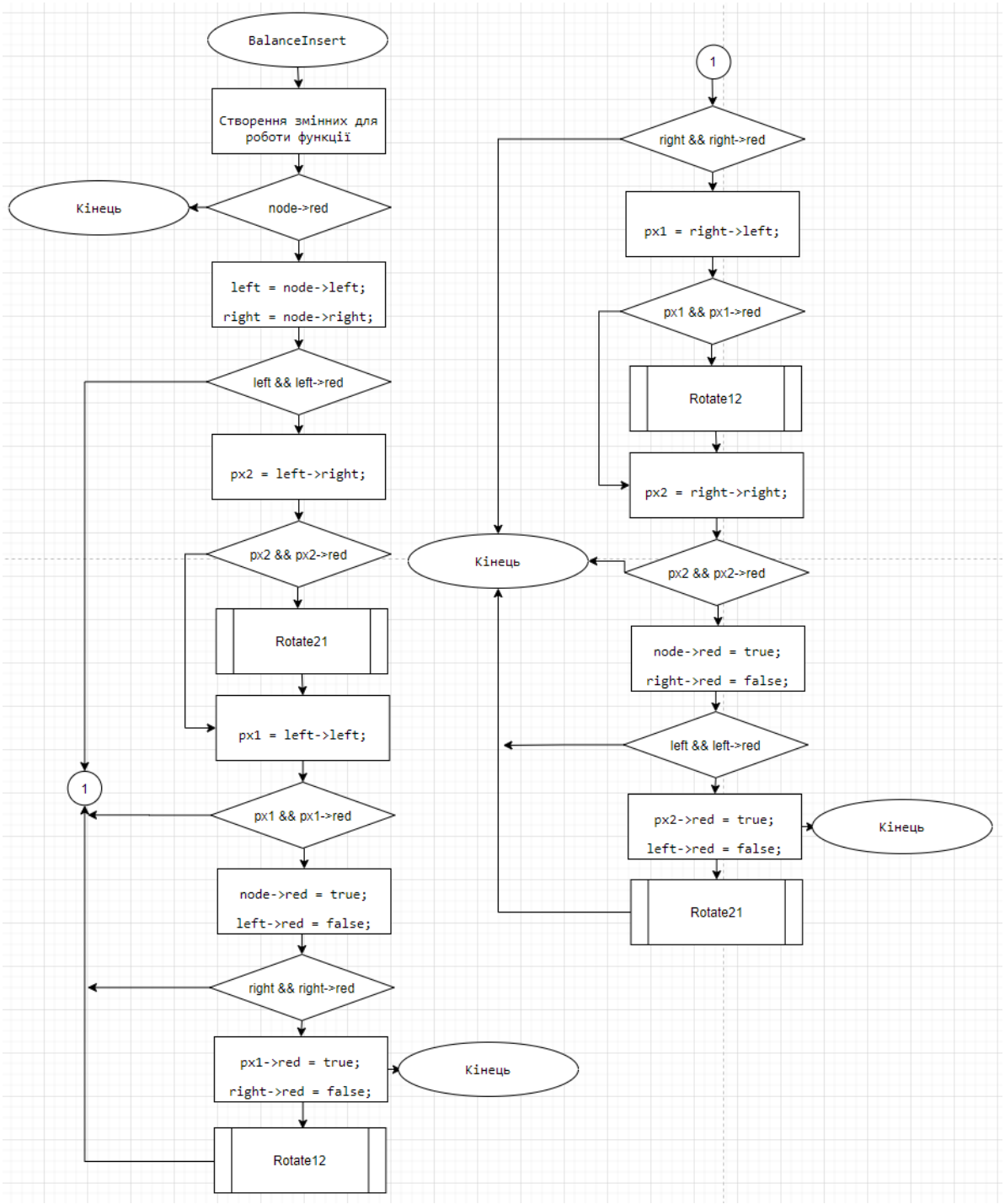
Індивідуальне завдання

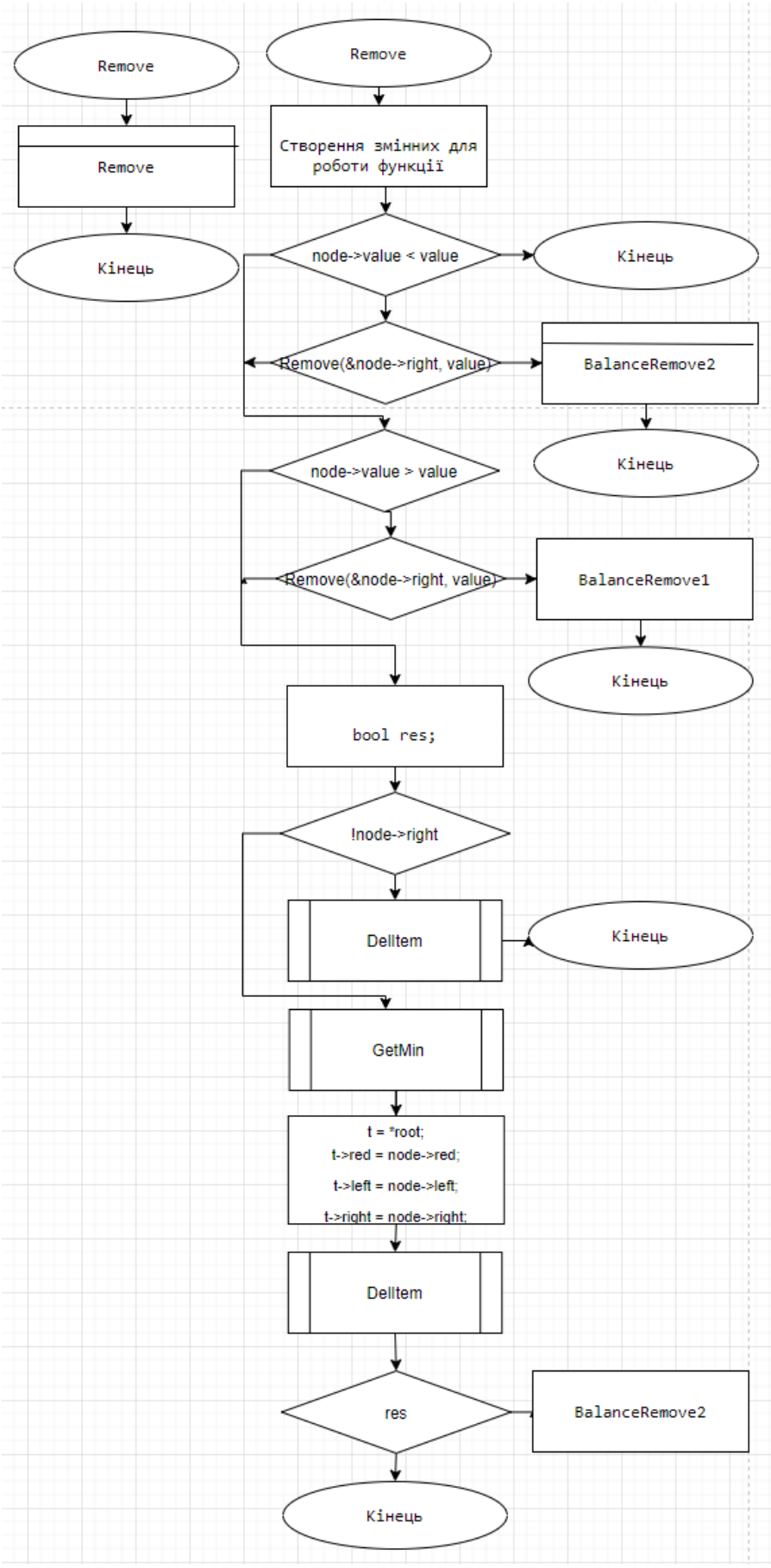
Розробити програму, що створює бінарне дерево та вирішує індивідуальне завдання. Видати вміст дерева та результати індивідуального завдання на екран.

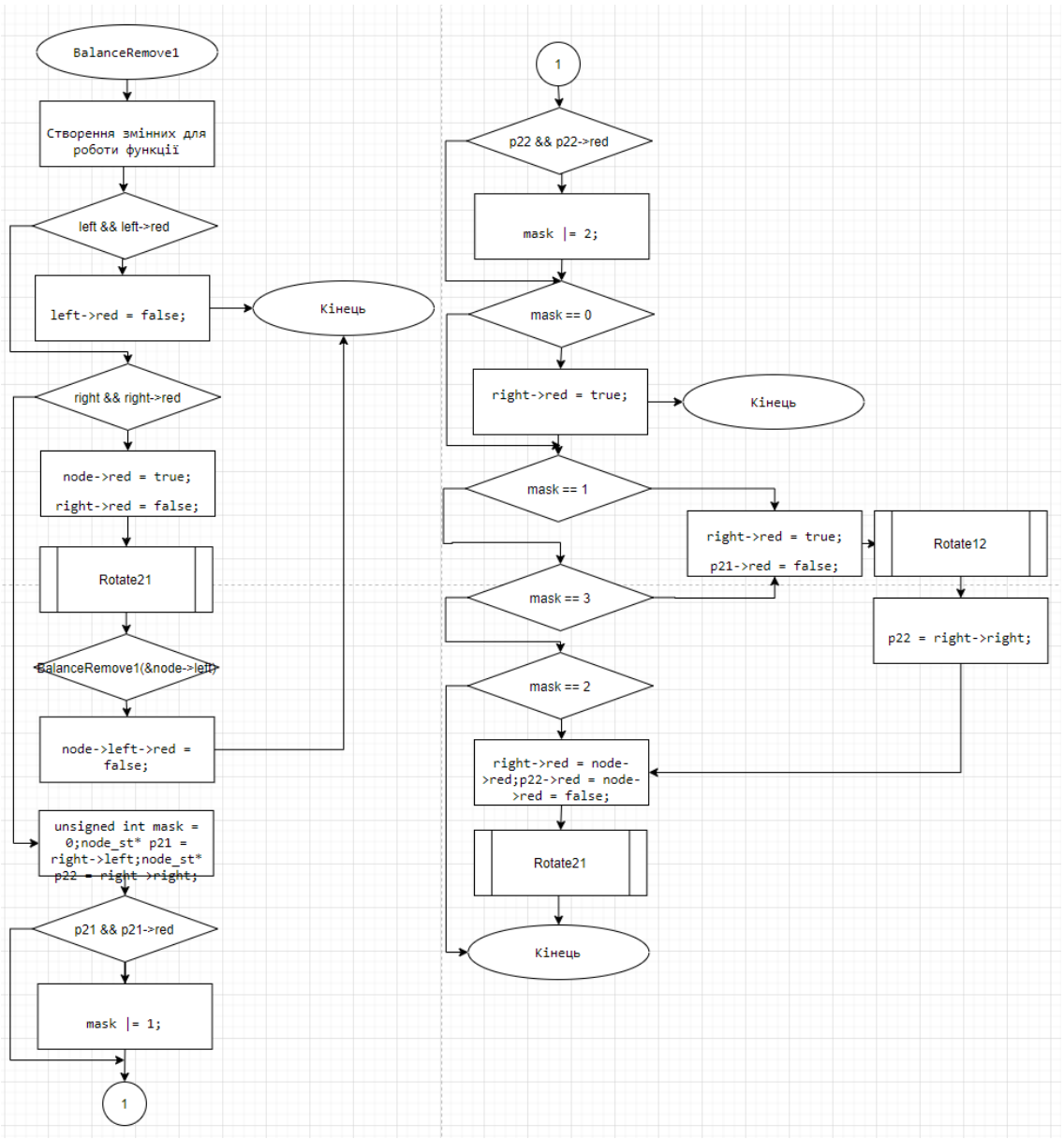
Індивідуальне завдання: розробити програму для побудови червоно-чорного дерева. Забезпечити видалення вузлів за заданим значенням.

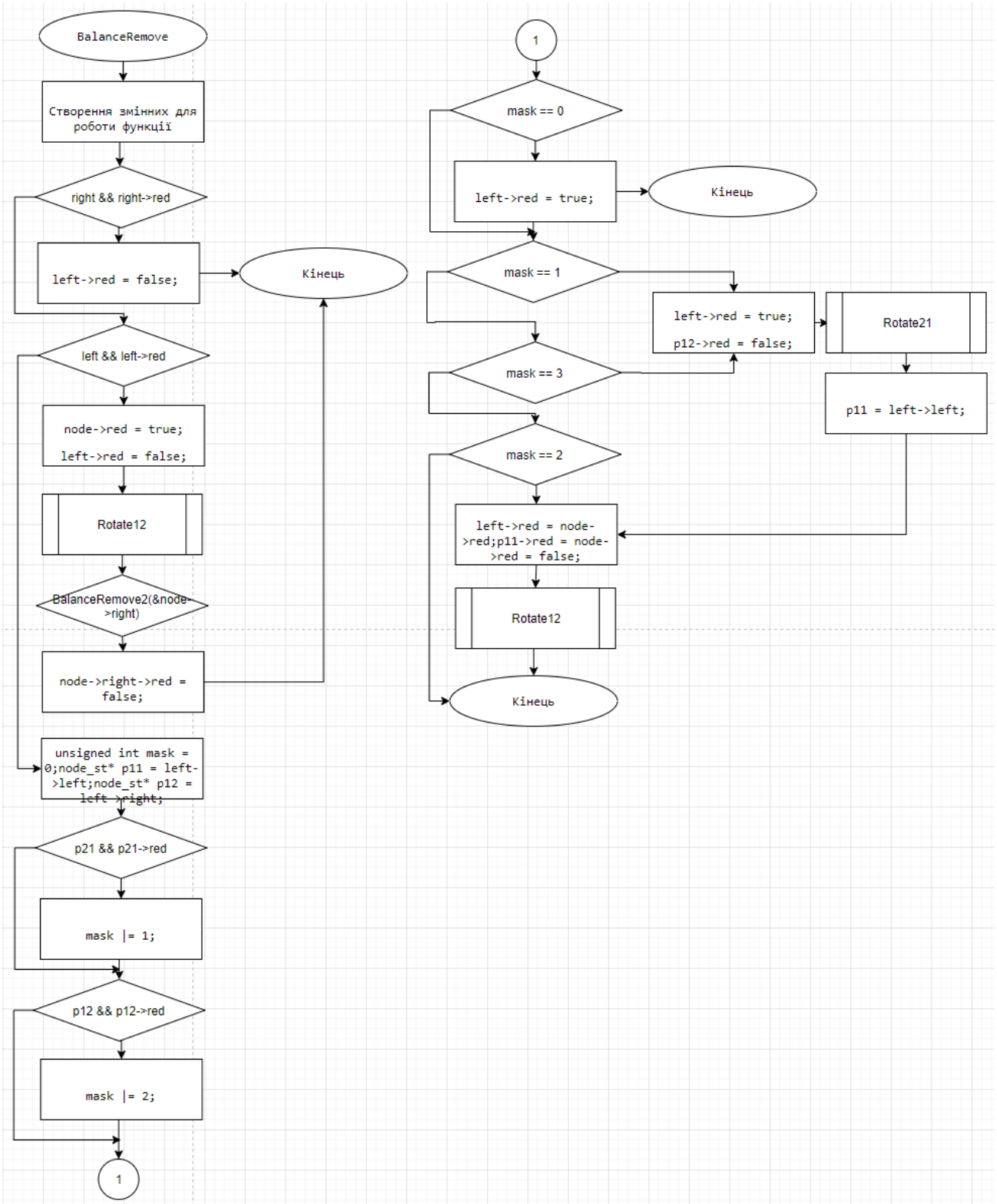
Блок-схеми алгоритмів програми




BalanceInsert
Створення змінних для роботи функції
node->red
Кінець
left = node->left;
right = node->right;
left && left->red
px2 = left->right;
px2 && px2->red
Rotate21
px1 = left->left;
1
px1 && px1->red
node->red = true;
left->red = false;
right && right->red
px1->red = true;
right->red = false;
Кінець
Rotate12
1
right && right->red
px1 = right->left;
px1 && px1->red
Rotate12
px2 = right->right;
Кінець
px2 && px2->red
node->red = true;
right->red = false;
left && left->red
px2->red = true;
left->red = false;
Кінець
Rotate21


Remove
Remove
Remove
Створення змінних для роботи функції
Кінець
node->value < value
Кінець
Remove(&node->right, value)
BalanceRemove2
Кінець
node->value > value
Remove(&node->right, value)
BalanceRemove1
Кінець
bool res;
!node->right
DelItem
Кінець
GetMin
t = *root;
t->red = node->red;
t->left = node->left;
t->right = node->right;
DelItem
res
BalanceRemove2
Кінець


BalanceRemove1
Створення змінних для роботи функції
left && left->red
left->red = false;
Кінець
right && right->red
node->red = true;
right->red = false;
Rotate21
BalanceRemove1(&node->left)
node->left->red = false;
unsigned int mask = 0;node_st* p21 = right->left;node_st* p22 = right->right;
p21 && p21->red
mask |= 1;
1
1
p22 && p22->red
mask |= 2;
mask == 0
right->red = true;
Кінець
mask == 1
right->red = true;
p21->red = false;
Rotate12
mask == 3
p22 = right->right;
mask == 2
right->red = node->red;p22->red = node->red = false;
Rotate21
Кінець


BalanceRemove
Створення змінних для роботи функції
right && right->red
left->red = false;
Кінець
left && left->red
node->red = true;
left->red = false;
Rotate12
BalanceRemove2(&node->right)
node->right->red = false;
unsigned int mask = 0;node_st* p11 = left->left;node_st* p12 = left->right;
p21 && p21->red
mask |= 1;
p12 && p12->red
mask |= 2;
1
1
mask == 0
left->red = true;
Кінець
mask == 1
left->red = true;
p12->red = false;
Rotate21
p11 = left->left;
mask == 3
mask == 2
left->red = node->red;p11->red = node->red = false;
Rotate12
Кінець

Текст програми

```
#define __CRTDBG_MAP_ALLOC
#include <crtdbg.h>
#define DEBUG_NEW new(_NORMAL_BLOCK, __FILE__, __LINE__)
#define new DEBUG_NEW

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <memory.h>
#include <locale.h>
using namespace std;

#define SIZE 100    //размер массива проверки

struct node_st                          // структура узла
{
	node_st* left, * right;    // left,right - левая правая ветка
	int value;                          // value - значение
	bool red;                           // red   - цвет (true - если красный)
};

node_st* tree_root = 0;    //корень

void Show();               //вывод дерева в консоль
void Clear();              //очистка дерева
void Insert(int);          //вставка элемента
void Remove(int);          //удаление элемента

int Check();                            //проверка дерева
int Check(node_st*, int, int);    //проверка дерева (рекурсивная часть)

bool TreeWalk(node_st*, bool*, int);     //обход дерева и сверка значений с массивом
(рекурсивная часть)
bool TreeWalk(bool*, int);                    //обход дерева и сверка значений с массивом

node_st* NewItem(int);                        //выделение новой вешины
void DelItem(node_st*);                       //удаление вершины
void Clear(node_st*);                         //снос дерева (рекурсивная часть)
void Show(node_st*, int, char);          //вывод дерева, рекурсивная часть
node_st* Rotate21(node_st*);             //вращение влево
node_st* Rotate12(node_st*);             //вращение вправо
void BalanceInsert(node_st**);           //балансировка вершины
bool BalanceRemove1(node_st**);
bool BalanceRemove2(node_st**);
bool Insert(int, node_st**);             //рекурсивная часть вставки
bool Insert(int, node_st**);             //рекурсивная часть вставки
bool GetMin(node_st**, node_st**);       //найти и убрать максимальный узел поддерева
bool Remove(node_st**, int);             //рекурсивная часть удаления

int main()
{
	setlocale(LC_ALL, "Rus");
	int n, i, value = 0;
	bool array[SIZE];
	srand(time(0));

	memset(array, false, sizeof(array));
	printf("Заполнение массива данных:\n");
	for (n = 0; n < SIZE * 100; n++) {
		printf("Проход: %d of %d\r", n + 1, SIZE * 100);
		i = rand() % SIZE; array[i] = true;  Insert(i);
		i = rand() % SIZE; array[i] = false; Remove(i);
```

```
	}
	putchar('\n');

	if (TreeWalk(array, SIZE))
		printf("*** context error\n\a");

	Show();

	printf("Введите число, которое надо удалить: ");
	scanf_s("%i", &value);
	Remove(value);
	printf("Результат:\n");
	Show();


	Clear();

	if (_CrtDumpMemoryLeaks())
		printf("\n\nУтечка памяти обнаружена.\n");
	else
		printf("\n\nУтечка памяти отсутствует.\n");

	return 0;
}

int Check()		//проверка дерева
{
	int d = 0;

	if (!tree_root)
		return 0;

	return Check(tree_root, d, 0);
}
int Check(node_st* tree, int b_d, int d) //проверка дерева (рекурсивная часть)
{
	int n;
	node_st* left = tree->left;
	node_st* right = tree->right;

	if (!tree->red)
		b_d++;

	n = Check(left, b_d, d + 1);

	if (n)
		return n;

	return Check(right, b_d, d + 1);
}
bool TreeWalk(node_st* node, bool* array, int size)		//обход дерева и сверка значений с
массивом (рекурсивная часть)
{
	if (!node)
		return false;

	int value = node->value;

	if (value < 0 || value >= size || !array[value])
		return true;

	array[value] = false;

	return TreeWalk(node->left, array, size) || TreeWalk(node->right, array, size);
}
bool TreeWalk(bool* array, int size)	//обход дерева и сверка значений с массивом
```

```
{
	if (TreeWalk(tree_root, array, size))
		return true;

	for (int n = 0; n < size; n++)
		if (array[n])
			return true;

	return false;
}
void Show()		//вывод дерева
{
	printf("\nДерево\n");
	Show(tree_root, 0, '*');
}
void Insert(int value)	//функция вставки
{
	Insert(value, &tree_root);

	if (tree_root)
		tree_root->red = false;
}
void Remove(int value)	//удаление узла
{
	Remove(&tree_root, value);
}
void Clear()  //снос дерева
{
	Clear(tree_root);
	tree_root = 0;
}

node_st* NewItem(int value)	//выделение новой вешины
{
	node_st* node = new node_st;
	node->value = value;
	node->left = node->right = 0;
	node->red = true;

	return node;
}
void DelItem(node_st* node)	//удаление вершины
{
	delete node;
}
void Clear(node_st* node)  //снос дерева (рекурсивная часть)
{
	if (!node)
		return;

	Clear(node->left);
	Clear(node->right);
	DelItem(node);
}
void Show(node_st* node, int depth, char dir)  //вывод дерева, рекурсивная часть
{
	int n;

	if (!node)
		return;

	for (n = 0; n < depth; n++)
		putchar(' ');

	printf("%c%d (%s)\n", dir, node->value, node->red ? "Красное" : "Чёрное");
	Show(node->left, depth + 2, '-');
```

```
	Show(node->right, depth + 2, '+');
}
node_st* Rotate21(node_st* node)  //вращение влево
{
	node_st* right = node->right;
	node_st* p21 = right->left;
	right->left = node;
	node->right = p21;

	return right;
}
node_st* Rotate12(node_st* node)  //вращение вправо
{
	node_st* left = node->left;
	node_st* p12 = left->right;
	left->right = node;
	node->left = p12;

	return left;
}

void BalanceInsert(node_st** root)	//балансировка вершины
{
	node_st* left, * right, * px1, * px2;
	node_st* node = *root;

	if (node->red)
		return;

	left = node->left;
	right = node->right;

	if (left && left->red)
	{
		px2 = left->right;

		if (px2 && px2->red)
			left = node->left = Rotate21(left);

		px1 = left->left;

		if (px1 && px1->red)
		{
			node->red = true;
			left->red = false;

			if (right && right->red)
			{
				px1->red = true;
				right->red = false;

				return;
			}

			*root = Rotate12(node);
		}
	}

	if (right && right->red)
	{
		px1 = right->left;

		if (px1 && px1->red)
			right = node->right = Rotate12(right);

		px2 = right->right;
```

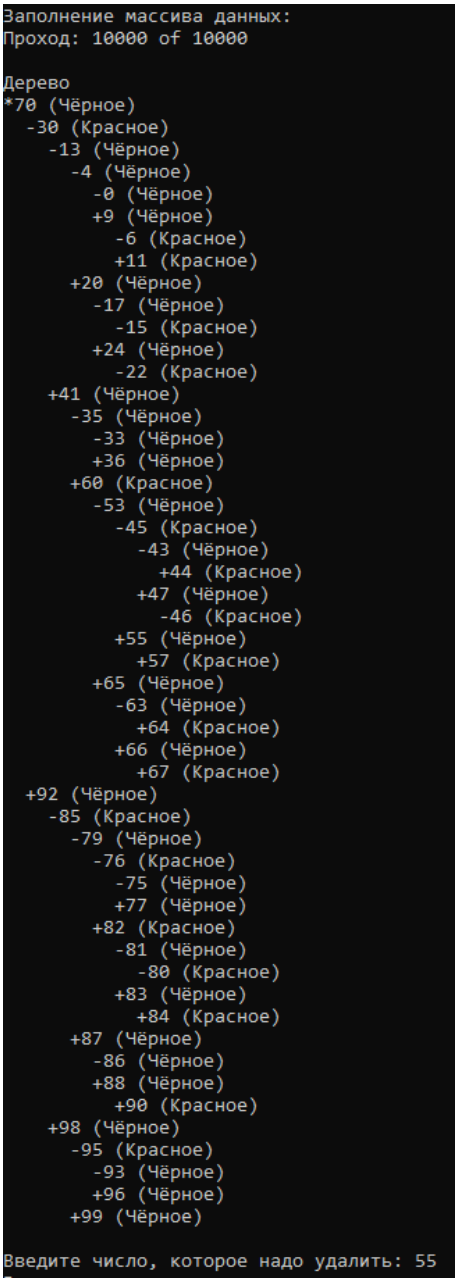
```
                if (px2 && px2->red)
                {
                        node->red = true;
                        right->red = false;

                        if (left && left->red)
                        {
                                px2->red = true;
                                left->red = false;

                                return;
                        }
                        *root = Rotate21(node);
                }
        }
}
bool BalanceRemove1(node_st** root)
{
        node_st* node = *root;
        node_st* left = node->left;
        node_st* right = node->right;

        if (left && left->red)
        {
                left->red = false;
                return false;
        }

        if (right && right->red)
        {
                node->red = true;
                right->red = false;
                node = *root = Rotate21(node);

                if (BalanceRemove1(&node->left))
                        node->left->red = false;

                return false;
        }

        unsigned int mask = 0;
        node_st* p21 = right->left;
        node_st* p22 = right->right;

        if (p21 && p21->red)
                mask |= 1;
        if (p22 && p22->red)
                mask |= 2;

        switch (mask)
        {
        case 0:
                right->red = true;
                return true;

        case 1:
        case 3:
                right->red = true;
                p21->red = false;
                right = node->right = Rotate12(right);
                p22 = right->right;

        case 2:
                right->red = node->red;
                p22->red = node->red = false;
```

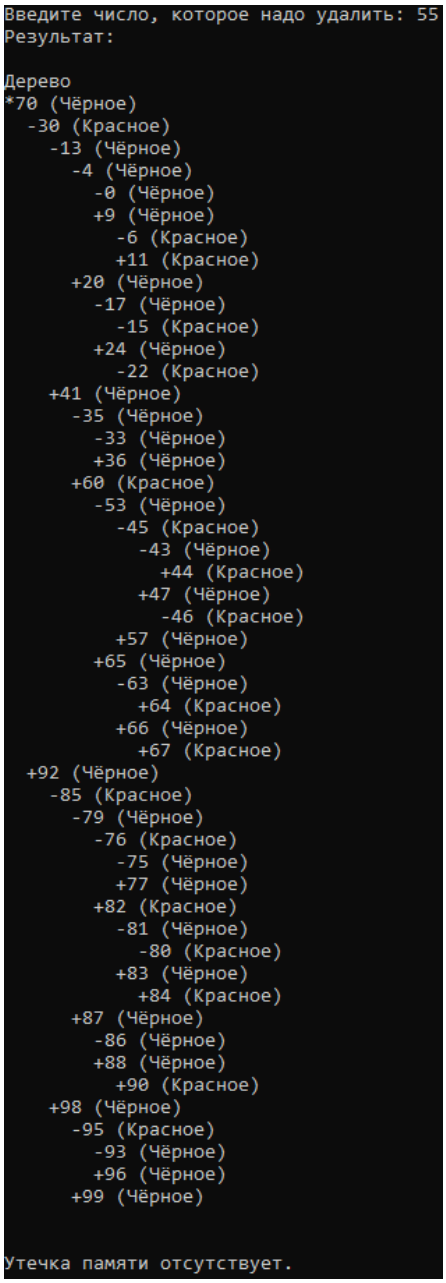
```
                *root = Rotate21(node);

        }

        return false;
}
bool BalanceRemove2(node_st** root)
{
        node_st* node = *root;
        node_st* left = node->left;
        node_st* right = node->right;

        if (right && right->red)
        {
                right->red = false;

                return false;
        }

        if (left && left->red)
        {
                node->red = true;
                left->red = false;
                node = *root = Rotate12(node);

                if (BalanceRemove2(&node->right))
                        node->right->red = false;

                return false;
        }

        unsigned int mask = 0;
        node_st* p11 = left->left;
        node_st* p12 = left->right;

        if (p11 && p11->red)
                mask |= 1;
        if (p12 && p12->red)
                mask |= 2;

        switch (mask)
        {
        case 0:
                left->red = true;
                return true;

        case 2:
        case 3:
                left->red = true;
                p12->red = false;
                left = node->left = Rotate21(left);
                p11 = left->left;

        case 1:
                left->red = node->red;
                p11->red = node->red = false;
                *root = Rotate12(node);

        }

        return false;
}
bool Insert(int value, node_st** root)
{
        node_st* node = *root;
        if (!node)
```

```
			*root = NewItem(value);
		else
		{
			if (value == node->value)
				return true;
			if (Insert(value, value < node->value ? &node->left : &node->right))
				return true;

			BalanceInsert(root);
		}

		return false;
}
bool GetMin(node_st** root, node_st** res)	//найти и убрать максимальный узел поддерева
{
		node_st* node = *root;

		if (node->left)
		{
			if (GetMin(&node->left, res))
				return BalanceRemove1(root);
		}
		else
		{
			*root = node->right;
			*res = node;

			return !node->red;
		}

		return false;
}
bool Remove(node_st** root, int value)	//рекурсивная часть удаления
{
		node_st* t, * node = *root;

		if (!node)
			return false;

		if (node->value < value)
		{
			if (Remove(&node->right, value))
				return BalanceRemove2(root);
		}
		else if (node->value > value)
		{
			if (Remove(&node->left, value))
				return BalanceRemove1(root);
		}
		else
		{
			bool res;

			if (!node->right)
			{
				*root = node->left;
				res = !node->red;
				DelItem(node);
				return res;
			}

			res = GetMin(&node->right, root);
			t = *root;
			t->red = node->red;
			t->left = node->left;
			t->right = node->right;
```

```
            DelItem(node);

            if (res)
                  res = BalanceRemove2(root);

            return res;
      }

      return 0;
}
```

Результат роботи про грами





Висновок

У результаті лабораторної роботи програми було розроблено програму, яка базується на червоно-чорному дереві. Були розроблені функції виведення дерева у консоль, вставки елемента у дерево, видалення елемента з дерева, видалення дерева, балансування дерева.

Відповіді на питання

1. Які дерева називають червоно-чорними. Які їх властивості?

Червоно-чорне дерево - це одне з самобалансующіхся довічних дерев пошуку, що гарантують логарифмичне зростання висоти дерева від числа вузлів і швидко виконує основні операції дерева пошуку: додавання, видалення і пошук вузла.

Збалансованість досягається за рахунок введення додаткового атрибута вузла дерева - «кольору». Цей атрибут може приймати одне з двох можливих значень - «чорний» або «червоний».

Червоно-чорне дерево - це бінарне дерево з наступними властивостями:

1). Кожен вузол пофарбований або в чорний, або в червоний колір. Корінь дерева - чорний.

2). Листям оголошуються NIL-вузли - це «віртуальні» вузли - спадкоємці вузлів, які зазвичай називають листям; на них «вказують» NULL покажчики. Листя пофарбовані в чорний колір.

3). Якщо вузол червоний, то обидва його нащадка чорні.

4). На всіх гілках дерева, що ведуть від його кореня до листя, число чорних вузлів однаково.

Найдовша гілка від кореня до листа не більше ніж удвічі довше будь-який інший галузі від кореня до листа.

2. Які дерева називають АВЛ-деревами. Які їх властивості?

АВЛ-дерева – це збалансовані дерева. Дерево називається збалансованим, якщо висоти лівих і правих піддерев кожної з її вершин відрізняються не більше, ніж на одиницю.

Висота h АВЛ-дерева з n ключами лежить в діапазоні від log2 (n + 1) до 1.44 log2 (n + 2) - 0.328. А так як основні операції над двійковими деревами пошуку (пошук, вставка і видалення вузлів) лінійно залежать від його висоти, то виходить гарантована логарифмічна залежність часу роботи цих алгоритмів від числа ключів, що зберігаються в дереві.

3. Які дерева називають ідеально збалансованими. Які їх властивості?

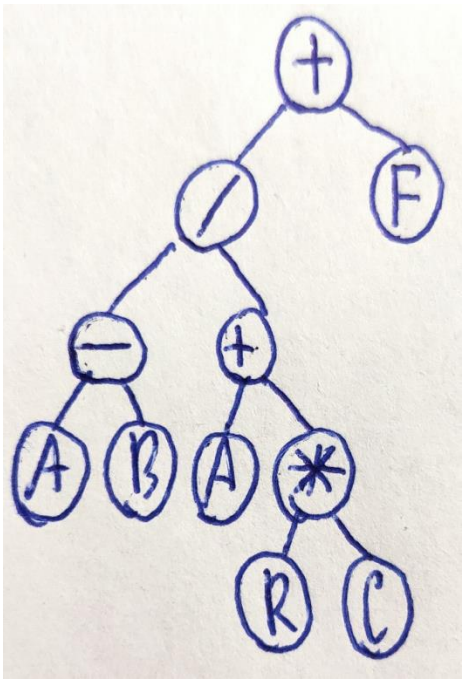
За визначенням Вірта дерево називається ідеально збалансованим, якщо число вершин в його лівих і правих піддерева відрізняється не більше, ніж на одну вершину. Таке дерево має мінімальну глибину (кількість рівнів). Мінімальна глибина досягається, якщо на всіх рівнях, крім останнього, міститься максимально можливе число вершин.

4. Яке призначення дерев Хаффмана?

В даний час кодування Хаффмана використовується в багатьох програмах стиснення даних в тому числі при стисненні фото- і відеозображень (JPEG, MPEG), в популярних архіваторах (PKZIP, LZH та ін.), В протоколах передачі даних HTTP.

5. Напишіть арифметичний вираз з кількістю операцій не менше 5. Побудуйте для нього бинарне дерево. Запишіть інфіксний, префіксний та постфіксний записи даного виразу.

((A - B) / (A + (R * C))) + F



Префіксна: +/-AB+A*RCF
Інфіксна: A-B/A+R*C+F
Постфіксна: AB-ARC*+/F+

6. Напишіть 5 символів та оберіть для них частоту появи у тексті. Побудуйте для них дерево Хаффмана. Побудуйте коди для заданих символів.

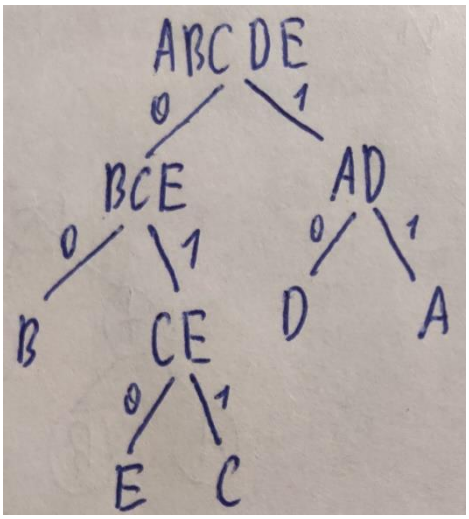
Символи: A,B,C,D,E;

Вираз: AADCBDEABCDA;

Частотність:

A	B	C	D	E
4 рази	2 рази	2 рази	3 рази	1 раз

Дерево Хаффмана:

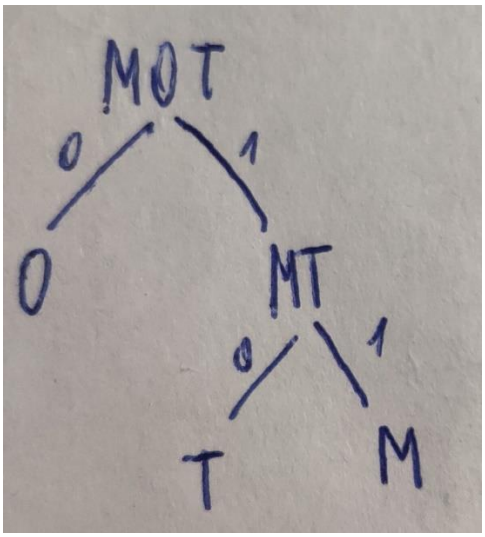


Коди символів:

Символ	Код
A	11
B	00
C	011
D	10
E	010

7. Напишіть ваше прізвище. Закодуйте його за допомогою дерева мінімального кодування.

Прізвище: Момот



M 11

O 0

T 10

Код: 11011010