ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«Национальный исследовательский университет ИТМО»**

**Факультет безопасности информационных технологий**

**Дисциплина:**

«Алгоритмы и структуры данных»

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4**

**Реализация сортировки кучей, формирующей кольцевую очередь на базе связного списка из красно-черного дерева**

**Выполнил:**

Студент группы N3350

Ахраров А. Р.

**Проверил:**

Ерофеев С.А.

# **ВВЕДЕНИЕ**

Цель работы: Разработать программу сортировки кучей, формирующей кольцевую очередь на базе связного списка из красно-черного дерева. В структуру узла кольцевой очереди внести цвет узла.

**ТЕХНИЧЕСКОЕ ЗАДАНИЕ (ПЕРЕЧЕНЬ ВХОДНЫХ, ВЫХОДНЫХ И ПРОМЕЖУТОЧНЫХ ДАННЫХ В ВИДЕ ТАБЛИЦЫ)**

|  |  |  |  |
| --- | --- | --- | --- |
| Название | Смысл | Тип данных | Ёмкость |
| largest | Корень | int | 4 байта |
| left | Левый потомок | int | 4 байта |
| right | Правый потомок | int | 4 байта |
| size | Размер массива | int | 4 байта |
| n | Размер массива | int | 4 байта |
| a | Массив для заполнения числами и их сортировки | int | 4 байта |
| parent | Указатель на родителя узла | Node\* | 4 байта |
| left | Указатель на левого ребенка узла | Node\* | 4 байта |
| right | Указатель на правого ребенка узла | Node\* | 4 байта |
| data | значение узла | int | 4 байта |
| color | Цвет узла (0 - черный, 1 - красный) | int | 4 байта |
| root | Корень КЧД | Node\* | 4 байта |
| TNULL | Лист КЧД | Node\* | 4 байта |
| data | Значение узла кольцевой очереди (КО) | int | 4 байта |
| next | Указатель на следующий узел в КО | QueueNode\* | 4 байта |
| color | Цвет узла КО (черный или красный) | int | 4 байта |
| tail | Указатель на конец КО | QueueNode\* | 4 байта |

**ОПИСАНИЕ АЛГОРИТМА**

1. Пирамидальная сортировка

Пирамидальная сортировка - это метод сортировки сравнением, основанный на двоичной куче.

Законченное двоичное дерево - это двоичное дерево, в котором каждый уровень, за исключением, возможно, последнего, имеет полный набор узлов, и все листья расположены как можно левее.

Сортировка основана именно на двоичной куче - это законченное двоичное дерево, в котором элементы хранятся в особом порядке: значение в родительском узле больше (или меньше) значений в его двух дочерних узлах.

В данной лабораторной реализуется первый вариант двоичной кучи - когда значение в родительском узле строго больше значений в дочерних узлах - max-heap.

Пирамидальная сортировка (по возрастанию) происходит по следующему алгоритму:

* Построить из входных данных max-heap
* После построения max-heap максимальное значение всегда будет находиться в корне кучи. Меняем местами корень и последний элемент в куче, уменьшаем размер кучи на 1 - теперь в конце массива будет наибольшее значение из входных данных
* Повторяем вышеперечисленные шаги до тех пор, пока размер кучи станет равным 1. Оставшийся элемент будет минимальным из входных данных
* Конец сортировки
  1. void Heap(int a[], int size, int i)

Функция построения двоичной кучи max-heap. В качестве параметров принимает массив входных данных, размер массива и индекс “корня”

* 1. void Sort(int a[], int n)

Функция пирамидальной сортировки. В качестве параметров принимает массив входных данных и размер массива

* 1. void Randomizer(int a[], int size)

Функция заполнения массива произвольными числами в диапазоне от 1000 до 5999. В качестве параметров принимает массив и его размер

1. Кольцевая очередь

Очередь (англ. queue) — это структура данных, добавление и удаление элементов в которой происходит путём операций enqueue и dequeue соответственно. Притом первым из очереди удаляется элемент, который был помещен туда первым, то есть в очереди реализуется принцип «первым вошел — первым вышел»

* 1. void enqueue(int data, int color)

Функция добавления нового узла в конец очереди. В качестве параметра принимает значение и цвет нового узла.

* 1. void dequeue()

Функция удаления узла с начала очереди.

1. Красно-чёрное дерево

Красно-чёрное дерево — один из видов самобалансирующихся двоичных деревьев поиска, позволяющее быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла.

Свойства красно-черных деревьев:

* Каждый узел окрашен либо в красный, либо в черный цвет
* Корень окрашен в черный цвет
* Листья (NULL-узлы) окрашены в черный цвет
* Каждый красный узел должен иметь два черных дочерних узла. Нужно отметить, что у черного узла могут быть черные дочерние узлы. Красные узлы в качестве дочерних могут иметь только черные
* Пути от узла к его листьям должны содержать одинаковое количество черных узлов (черная высота)

Вставка в красно-черное дерево начинается со вставки элемента. Элементы вставляются в позиции NULL-листьев. Вставленный элемент всегда окрашивается в красный цвет. Далее идет процедура проверки сохранения свойств красно-черного дерева

Для балансировки КЧД используется 3 операции:

* Левосторонний поворот
* Правосторонний поворот
* Свап цветов

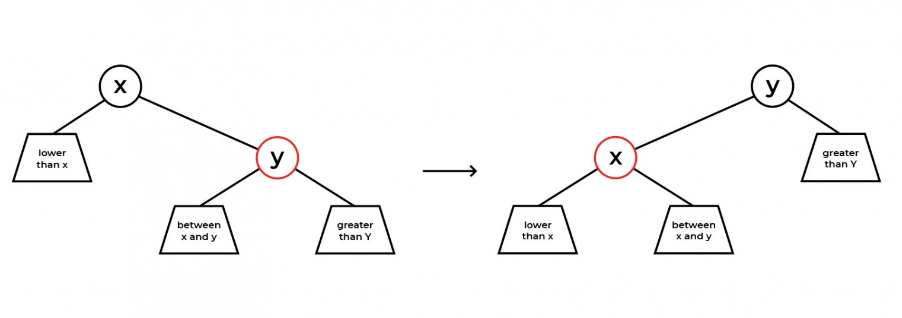


Рисунок 1 - иллюстрация левостороннего поворота

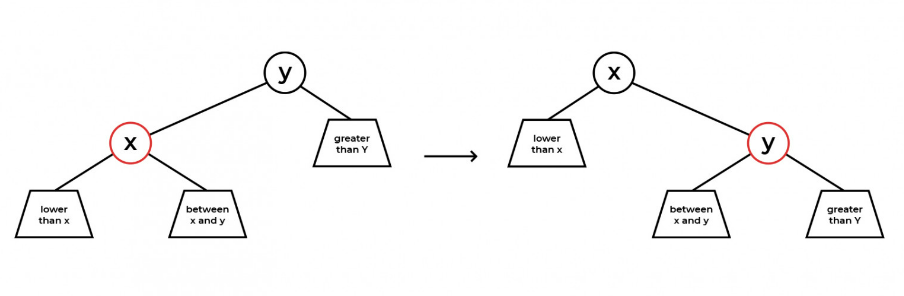


Рисунок 2 - иллюстрация правостороннего поворота

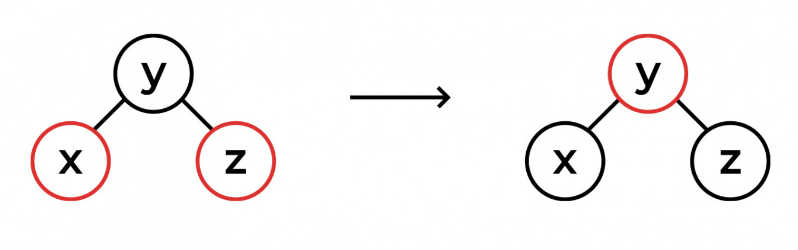


Рисунок 3 - иллюстрация свапа цветов

* 1. void leftRotate(NodePtr x)

Функция левостороннего поворота. Используется для балансировки дерева. В качестве параметра принимает родителя красного узла.

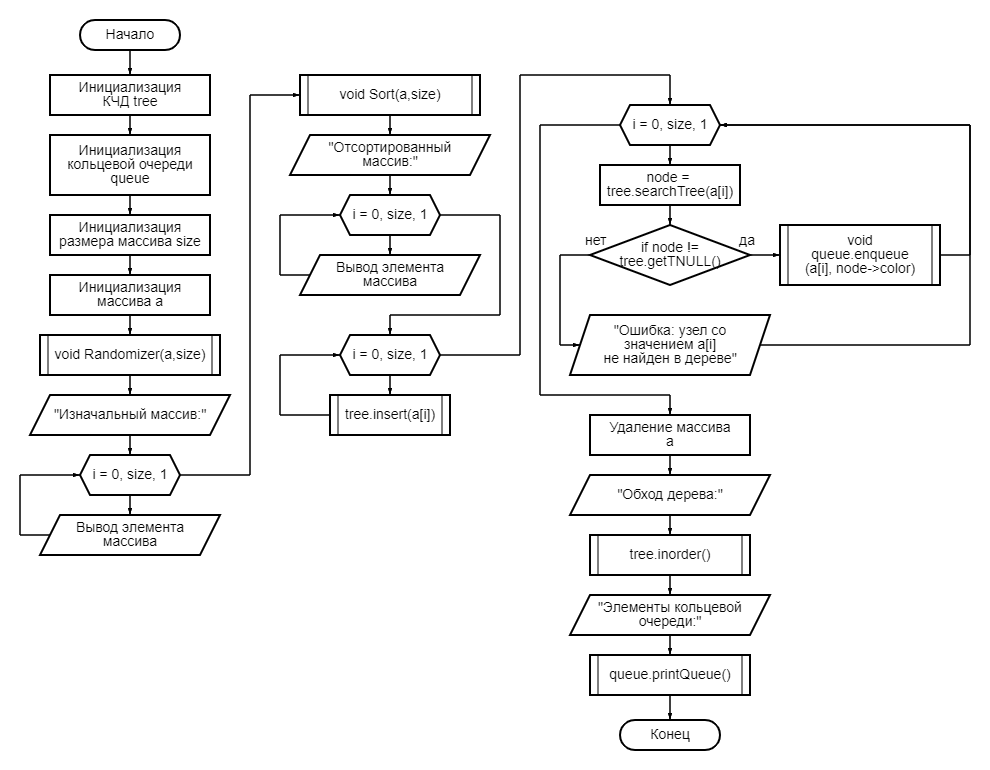
* 1. void rightRotate(NodePtr x)

Функция правостороннего поворота. Используется для балансировки дерева. В качестве параметра принимает родителя красного узла.

* 1. void insert(int key)

Функция вставки нового узла в КЧД. Устанавливаются указатели родителя, изначальный цвет - красный. В качестве параметра функция принимает значение нового узла. Новый узел вставляется в позицию NULL-листьев. Функция находит подходящее для узла место путем пробегания всего дерева. Меньшие значения узла-родителя являются его левым ребенком, большие - правам ребенком. Правый и левый ребенок нового узла являются TNULL-узлы - листья. После каждой вставки нового узла в КЧД выполняется балансировка для сохранения его свойств.

**БЛОК-СХЕМА**

****

**КОД ПРОГРАММЫ**

#include <iostream>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

using namespace std;

#define MAX\_SIZE 100

struct QueueNode {

int data;

QueueNode\* next;

int color; // Цвет узла (черный или красный)

};

// Класс для реализации кольцевой очереди с ограничением на количество узлов

class CircularQueue {

private:

QueueNode\* front; // Указатель на первый элемент (голову) очереди

QueueNode\* rear; // Указатель на последний элемент (хвост) очереди

int currentSize; // Текущее количество элементов в очереди

public:

CircularQueue() {

front = nullptr; // Изначально очередь пуста

rear = nullptr; // Изначально очередь пуста

currentSize = 0; // Текущий размер равен 0

}

// Функция для добавления элемента в кольцевую очередь

void enqueue(int data, int color) {

if (isFull()) {

// Если очередь полна, выводим сообщение об ошибке

cout << "Ошибка: Очередь переполнена, невозможно добавить элемент: " << data << endl;

return;

}

// Создаем новый узел

QueueNode\* newNode = new QueueNode;

newNode->data = data;

newNode->color = color;

newNode->next = nullptr;

if (front == nullptr) {

// Если очередь пуста, инициализируем первый элемент

front = newNode;

rear = newNode;

rear->next = front; // Указываем, что хвост ссылается на голову (образуем кольцо)

}

else {

// Добавляем новый узел после хвоста и обновляем указатели

rear->next = newNode;

rear = newNode;

rear->next = front; // Обновляем связь, чтобы кольцо сохранялось

}

currentSize++;

}

// Функция для удаления элемента из кольцевой очереди

void dequeue() {

if (isEmpty()) {

cout << "Очередь пуста, невозможно удалить элемент." << endl;

return;

}

QueueNode\* temp = front;

if (front == rear) {

// Если в очереди только один элемент, очищаем очередь

front = nullptr;

rear = nullptr;

}

else {

// Перемещаем `front` на следующий узел

front = front->next;

rear->next = front; // Обновляем связь хвоста с новым началом

}

delete temp;

currentSize--;

}

// Функция для проверки, пуста ли очередь

bool isEmpty() {

return front == nullptr;

}

// Функция для проверки, полна ли очередь

bool isFull() {

return currentSize == MAX\_SIZE;

}

// Функция для вывода элементов кольцевой очереди

void printQueue() {

if (isEmpty()) {

cout << "Очередь пуста" << endl;

return;

}

QueueNode\* current = front;

do {

cout << "Значение: " << current->data << " | Цвет: " << (current->color == 0 ? "Black" : "Red") << endl;

current = current->next;

} while (current != front);

}

};

// Функция Heapify для поддержания свойства кучи

void Heap(int a[], int size, int i) {

while (true) {

int largest = i; // Инициализируем наибольший элемент как корень

int left = 2 \* i + 1; // Левый потомок

int right = 2 \* i + 2; // Правый потомок

// Проверяем, является ли левый потомок больше корня

if (left < size && a[left] > a[largest]) {

largest = left;

}

// Проверяем, является ли правый потомок больше на данный момент

if (right < size && a[right] > a[largest]) {

largest = right;

}

// Если корень остается самым большим, прекращаем работу

if (largest == i) {

break;

}

// Меняем местами текущий элемент с наибольшим

int temp = a[i];

a[i] = a[largest];

a[largest] = temp;

// Переходим к следующему узлу

i = largest;

}

}

// Функция для выполнения сортировки кучей

void Sort(int a[], int n) {

// Построение кучи из массива

for (int i = n / 2 - 1; i >= 0; i--) {

Heap(a, n, i);

}

// Извлечение элементов из кучи один за другим

for (int i = n - 1; i > 0; i--) {

int temp = a[0];

a[0] = a[i];

a[i] = temp;

// Вызываем Heapify на уменьшенной куче

Heap(a, i, 0);

}

}

// Функция для заполнения массива случайными числами

void Randomizer(int a[], int size) {

std::srand(static\_cast<unsigned int>(std::time(nullptr))); // Инициализация генератора случайных чисел

for (int i = 0; i < size; i++) {

a[i] = std::rand() % 5000 + 1000; // Генерация чисел от 1000 до 5999

}

}

// Узел для красно-черного дерева

struct Node {

int data;

Node\* parent;

Node\* left;

Node\* right;

int color; // Цвет узла (0 - черный, 1 - красный)

};

typedef Node\* NodePtr;

// Класс для реализации красно-черного дерева

class RedBlackTree {

private:

NodePtr root;

NodePtr TNULL;

// Вспомогательная функция для обхода дерева inOrder

void inOrderHelper(NodePtr node) {

if (node != TNULL) {

inOrderHelper(node->left);

cout << "Значение " << node->data << " | Цвет: " << (node->color == 0 ? "Black" : "Red") << endl;

inOrderHelper(node->right);

}

}

// Вспомогательная функция для поиска узла по ключу

NodePtr searchTreeHelper(NodePtr node, int key) {

if (node == TNULL) {

return nullptr;

}

if (key == node->data) {

return node;

}

if (key < node->data) {

return searchTreeHelper(node->left, key);

}

return searchTreeHelper(node->right, key);

}

// Функция для балансировки дерева после удаления узла

void deleteFix(NodePtr x) {

NodePtr s;

while (x != root && x->color == 0) {

if (x == x->parent->left) {

s = x->parent->right;

if (s->color == 1) {

s->color = 0;

x->parent->color = 1;

leftRotate(x->parent);

s = x->parent->right;

}

if (s->left->color == 0 && s->right->color == 0) {

s->color = 1;

x = x->parent;

}

else {

if (s->right->color == 0) {

s->left->color = 0;

s->color = 1;

rightRotate(s);

s = x->parent->right;

}

s->color = x->parent->color;

x->parent->color = 0;

s->right->color = 0;

leftRotate(x->parent);

x = root;

}

}

else {

s = x->parent->left;

if (s->color == 1) {

s->color = 0;

x->parent->color = 1;

rightRotate(x->parent);

s = x->parent->left;

}

if (s->right->color == 0 && s->left->color == 0) {

s->color = 1;

x = x->parent;

}

else {

if (s->left->color == 0) {

s->right->color = 0;

s->color = 1;

leftRotate(s);

s = x->parent->left;

}

s->color = x->parent->color;

x->parent->color = 0;

s->left->color = 0;

rightRotate(x->parent);

x = root;

}

}

}

x->color = 0;

}

// Трансплантация узлов при удалении

void rbTransplant(NodePtr u, NodePtr v) {

if (u->parent == nullptr) {

root = v;

}

else if (u == u->parent->left) {

u->parent->left = v;

}

else {

u->parent->right = v;

}

v->parent = u->parent;

}

// Вспомогательная функция для удаления узла

void deleteNodeHelper(NodePtr node, int key) {

NodePtr z = TNULL;

NodePtr x, y;

while (node != TNULL) {

if (node->data == key) {

z = node;

}

if (node->data <= key) {

node = node->right;

}

else {

node = node->left;

}

}

if (z == TNULL) {

cout << "Key not found in the tree" << endl;

return;

}

y = z;

int y\_original\_color = y->color;

if (z->left == TNULL) {

x = z->right;

rbTransplant(z, z->right);

}

else if (z->right == TNULL) {

x = z->left;

rbTransplant(z, z->left);

}

else {

y = minimum(z->right);

y\_original\_color = y->color;

x = y->right;

if (y->parent == z) {

x->parent = y;

}

else {

rbTransplant(y, y->right);

y->right = z->right;

y->right->parent = y;

}

rbTransplant(z, y);

y->left = z->left;

y->left->parent = y;

y->color = z->color;

}

delete z;

if (y\_original\_color == 0) {

deleteFix(x);

}

}

// Балансировка дерева после вставки узла

void insertFix(NodePtr k) {

NodePtr u;

while (k->parent->color == 1) {

if (k->parent == k->parent->parent->right) {

u = k->parent->parent->left;

if (u->color == 1) {

u->color = 0;

k->parent->color = 0;

k->parent->parent->color = 1;

k = k->parent->parent;

}

else {

if (k == k->parent->left) {

k = k->parent;

rightRotate(k);

}

k->parent->color = 0;

k->parent->parent->color = 1;

leftRotate(k->parent->parent);

}

}

else {

u = k->parent->parent->right;

if (u->color == 1) {

u->color = 0;

k->parent->color = 0;

k->parent->parent->color = 1;

k = k->parent->parent;

}

else {

if (k == k->parent->right) {

k = k->parent;

leftRotate(k);

}

k->parent->color = 0;

k->parent->parent->color = 1;

rightRotate(k->parent->parent);

}

}

if (k == root) {

break;

}

}

root->color = 0;

}

public:

RedBlackTree() {

TNULL = new Node;

TNULL->color = 0;

TNULL->left = nullptr;

TNULL->right = nullptr;

TNULL->parent = nullptr;

root = TNULL;

}

void inorder() {

inOrderHelper(this->root);

}

NodePtr searchTree(int k) {

return searchTreeHelper(this->root, k);

}

NodePtr minimum(NodePtr node) {

while (node->left != TNULL) {

node = node->left;

}

return node;

}

NodePtr maximum(NodePtr node) {

while (node->right != TNULL) {

node = node->right;

}

return node;

}

NodePtr successor(NodePtr x) {

if (x->right != TNULL) {

return minimum(x->right);

}

NodePtr y = x->parent;

while (y != TNULL && x == y->right) {

x = y;

y = y->parent;

}

return y;

}

NodePtr predecessor(NodePtr x) {

if (x->left != TNULL) {

return maximum(x->left);

}

NodePtr y = x->parent;

while (y != TNULL && x == y->left) {

x = y;

y = y->parent;

}

return y;

}

void leftRotate(NodePtr x) {

NodePtr y = x->right;

x->right = y->left;

if (y->left != TNULL) {

y->left->parent = x;

}

y->parent = x->parent;

if (x->parent == nullptr) {

this->root = y;

}

else if (x == x->parent->left) {

x->parent->left = y;

}

else {

x->parent->right = y;

}

y->left = x;

x->parent = y;

}

void rightRotate(NodePtr x) {

NodePtr y = x->left;

x->left = y->right;

if (y->right != TNULL) {

y->right->parent = x;

}

y->parent = x->parent;

if (x->parent == nullptr) {

this->root = y;

}

else if (x == x->parent->right) {

x->parent->right = y;

}

else {

x->parent->left = y;

}

y->right = x;

x->parent = y;

}

void insert(int key) {

NodePtr node = new Node;

node->parent = nullptr;

node->data = key;

node->left = TNULL;

node->right = TNULL;

node->color = 1;

NodePtr y = nullptr;

NodePtr x = this->root;

while (x != TNULL) {

y = x;

if (node->data < x->data) {

x = x->left;

}

else {

x = x->right;

}

}

node->parent = y;

if (y == nullptr) {

root = node;

}

else if (node->data < y->data) {

y->left = node;

}

else {

y->right = node;

}

if (node->parent == nullptr) {

node->color = 0;

return;

}

insertFix(node);

}

NodePtr getRoot() {

return this->root;

}

void deleteNode(int data) {

deleteNodeHelper(this->root, data);

}

// Добавляем публичный метод для доступа к TNULL

NodePtr getTNULL() {

return TNULL;

}

};

int main() {

RedBlackTree tree;

CircularQueue queue;

int size = 100;

int\* a = new int[size];

Randomizer(a, size);

std::cout << "Изначальный массив:\n";

for (int i = 0; i < size; i++) {

std::cout << a[i] << " ";

}

std::cout << std::endl;

Sort(a, size);

std::cout << "Отсортированный массив:\n";

for (int i = 0; i < size; i++) {

std::cout << a[i] << " ";

}

std::cout << std::endl;

for (int i = 0; i < size; i++) {

tree.insert(a[i]);

}

for (int i = 0; i < size; i++) {

NodePtr node = tree.searchTree(a[i]);

if (node != tree.getTNULL()) {

queue.enqueue(a[i], node->color);

}

else {

std::cout << "Ошибка: узел со значением " << a[i] << " не найден в дереве." << std::endl;

}

}

delete[] a;

std::cout << "Обход дерева" << std::endl;

tree.inorder();

std::cout << std::endl;

std::cout << "Элементы кольцевой очереди:" << std::endl;

queue.printQueue();

return 0;

}

**РЕЗУЛЬТАТ РАБОТЫ ПРОГРАММЫ**

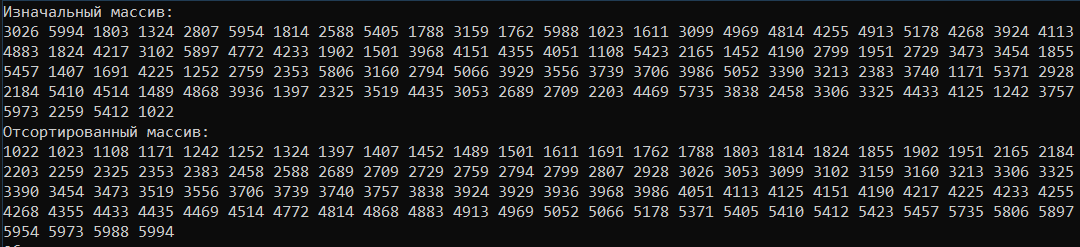
****

Рисунок 4 -вывод изначального и отсортированного массивов

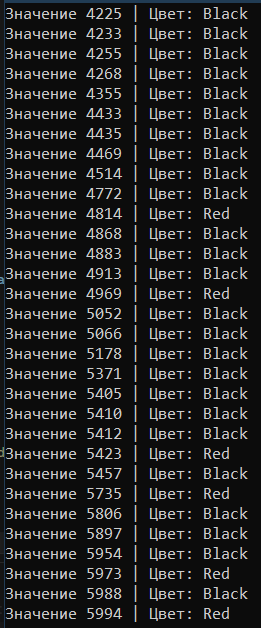
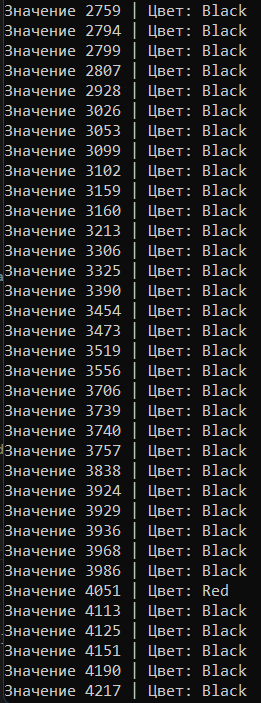
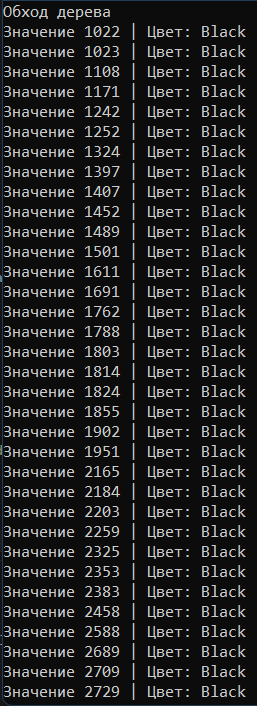
****

Рисунок 5 - 7 - обход дерева

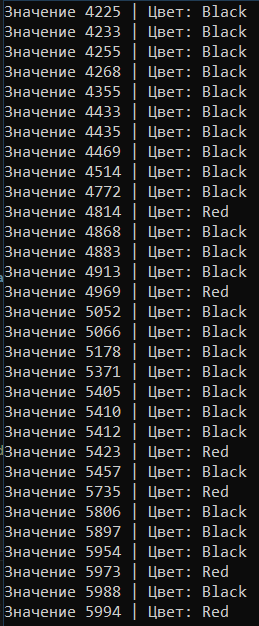
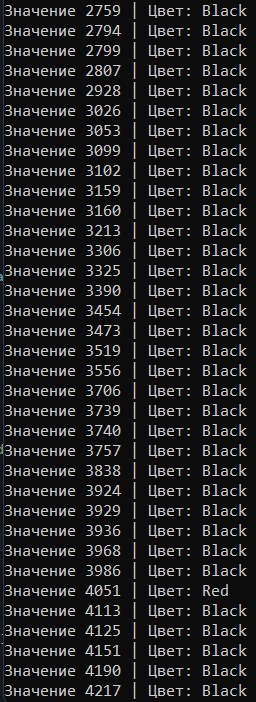
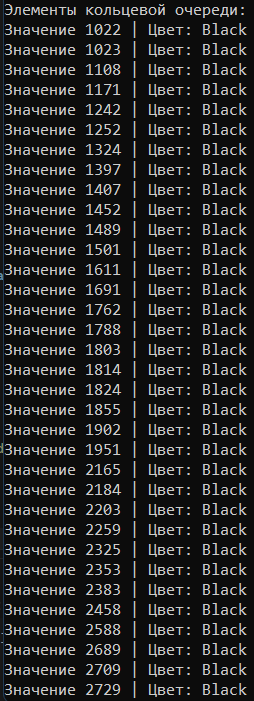
****

Рисунок 8 - 10 - элементы кольцевой очереди

**ВЫВОД**

Целью работы было разработать программу сортировки кучей, формирующей кольцевую очередь на базе связного списка из красно-черного дерева, а также в структуру узла кольцевой очереди внести цвет узла.. Задача была выполнена с использованием языка C++.

В ходе выполнения лабораторной работы была разработана программа, реализующая пирамидальную сортировку, составление красно-черного дерева и кольцевую очередь.

Были выполнены следующие задачи:

* Составлена блок-схема общего алгоритма программы
* Реализован код на языке C++
* Проведено тестирование программы

По итогам выполнения работы были проведены тесты, программа правильно сортирует числа с использованием красно-черного дерева и кольцевой очереди.