|  |
| --- |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА – Российский технологический университет"**  **РТУ МИРЭА** |

Институт кибербезопасности и цифровых технологий

Кафедра КБ-2 «Институт кибербезопасности и цифровых технологий»

**ОТЧЕТ   
о выполнении домашней работе**

**«Реализация сортировки линейных структур данных»**

**по дисциплине   
«Технологии и методы программирования»**

**Вариант № 78**

Выполнил: студент 2 курса

группы БИСО-01-21

Русаков А.М.

шифр ИТ6-4387086

Москва 2022 г.

**Вариант № 78.**

**Тип дерева: Рандомизированное дерево двоичного поиска**

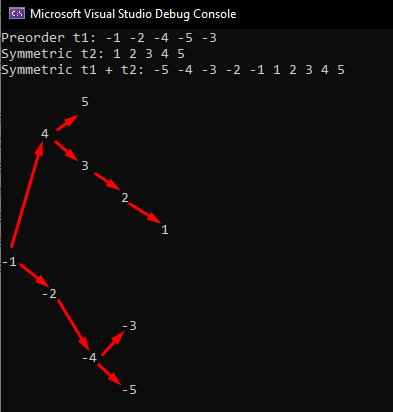
**Способ обхода дерева: Прямой и симметричный**

**Операция: Объединение с помощью симметричного обхода**

**Рандомизированное дерево двоичного поиска**

Рандомизированное дерево двоичного поиска – это вариант сбалансированного двоичного дерева поиска, при котором вставка в корень происходит с вероятностью 1/(n+1), а рекурсивная вставка в левое или правое поддерево с вероятностью 1-1/(n+1). Это возможно, т.к., при построении дерева из перемешанных ключей, его высота будет порядка 2log2n (вместо log2n для идеально сбалансированного дерева).

# Пример работы программы



Вывод дерева t1 в прямом порядке, вывод дерева t2 в симметричном порядке, вывод их объединения в симметричном порядке, а также схематический вывод объединения деревьев.

**ВЫВОДЫ**

Был реализован Абстрактный Тип Данных «Рандомизированное Бинарное Дерево Поиска», с помощью которого было выполнено задание варианта: вывод дерева в прямом и симметричном порядках и объединение двух деревьев путем симметричного обхода по второму дереву.

**Литература:**

1. Роберт Седжвик, Алгоритмы на C++, М.: Вильямс, 2011 г.

2. Martinez, Conrado; Roura, Salvador (1998), Randomized binary search trees, Journal of the ACM (ACM Press) 45 (2): 288–323

3. Reed, Bruce (2003), The height of a random binary search tree, Journal of the ACM 50 (3): 306–332

**Приложение 1 Листинг кода**

## Main.cpp

#include <iostream>

#include <random>

class Tree

{

struct Node

{

int key;

int size;

Node\* left;

Node\* right;

Node(int key, int size = 0, Node\* left = nullptr, Node\* right = nullptr)

:key(key), size(size), left(left), right(right) {}

};

public:

Tree()

:rnd(std::random\_device()()), root(nullptr) {}

Tree(const Tree& other)

:rnd(std::random\_device()())

{

root = new Node{ other.root->key, other.root->size };

other.preorder\_copy(other.root, root, copy\_func);

}

~Tree()

{

postorder\_delete(root);

}

Node\* find(int key) { return find(key, root); }

void insert(int key) { root = insert(key, root); }

void print() const { print(root, 0); }

void symmetric\_print() const { symmetric(root, print\_func); }

void preorder\_print() const { preorder(root, print\_func); }

void symmetric\_f(void(\*f\_ptr)(int)) const { symmetric(root, f\_ptr); }

void preorder\_f(void(\*f\_ptr)(int)) const { preorder(root, f\_ptr); }

// Добавление эл-тов в A из B в симм. порядке обхода

Tree operator+(const Tree& other)

{

Tree r\_tree{ \*this };

other.symmetric\_copy(other.root, &r\_tree, sym\_copy\_func);

return r\_tree;

}

private:

std::mt19937 rnd;

Node\* root;

Node\* find(int key, Node\* p)

{

if (!root)

return nullptr;

if (key == p->key)

return p;

if (key < p->key)

return find(key, p->left);

return find(key, p->right);

}

void fixsize(Node\* p)

{

int l\_size = 0;

int r\_size = 0;

if (p->left) l\_size = p->left->size;

if (p->right) r\_size = p->right->size;

p->size = l\_size + r\_size + 1;

}

Node\* rotate\_r(Node\* p)

{

Node\* q = p->left;

if (!q) return p;

p->left = q->right;

q->right = p;

q->size = p->size;

fixsize(p);

return q;

}

Node\* rotate\_l(Node\* q)

{

Node\* p = q->right;

if (!p) return q;

q->right = p->left;

p->left = q;

p->size = q->size;

fixsize(q);

return p;

}

Node\* insert\_root(int k, Node\* p)

{

if (!p) return new Node{ k };

if (k < p->key)

{

p->left = insert\_root(k, p->left);

return rotate\_r(p);

}

else

{

p->right = insert\_root(k, p->right);

return rotate\_l(p);

}

}

Node\* insert(int key, Node\* p)

{

if (!p) return new Node{ key, 1 };

if (rnd() % (p->size + 1) == 0)

return insert\_root(key, p);

if (p->key > key)

p->left = insert(key, p->left);

else

p->right = insert(key, p->right);

fixsize(p);

return p;

}

void print(Node\* p, int space) const

{

if (!p)

return;

space += 5;

print(p->right, space);

std::cout << "\n";

for (int i = 5; i < space; i++)

std::cout << " ";

std::cout << p->key << "\n";

print(p->left, space);

}

void preorder(Node\* p, void(\*f\_ptr)(int)) const

{

if (!p) return;

f\_ptr(p->key);

preorder(p->left, f\_ptr);

preorder(p->right, f\_ptr);

}

void symmetric(Node\* p, void(\*f\_ptr)(int)) const

{

if (!p) return;

symmetric(p->left, f\_ptr);

f\_ptr(p->key);

symmetric(p->right, f\_ptr);

}

void symmetric\_copy(Node\* p, Tree\* q, void(\*f\_ptr)(Node\*, Tree\*)) const

{

if (!p) return;

symmetric\_copy(p->left, q, f\_ptr);

f\_ptr(p, q);

symmetric\_copy(p->right, q, f\_ptr);

}

void preorder\_copy(Node\* p, Node\* q, void(\*f\_ptr)(Node\*, Node\*)) const

{

if (!p) return;

f\_ptr(p, q);

preorder\_copy(p->left, q->left, f\_ptr);

preorder\_copy(p->right, q->right, f\_ptr);

}

void postorder\_delete(Node\* p)

{

if (!p) return;

postorder\_delete(p->left);

postorder\_delete(p->right);

delete p;

}

static void print\_func(int x) { std::cout << x << " "; }

static void copy\_func(Node\* old\_r, Node\* new\_r)

{

if (old\_r->left)

{

new\_r->left = new Node{ old\_r->left->key, old\_r->left->size };

}

if (old\_r->right)

{

new\_r->right = new Node{ old\_r->right->key, old\_r->right->size };

}

}

static void sym\_copy\_func(Node\* r, Tree\* t)

{

t->insert(r->key);

}

};

int main()

{

srand(time(NULL));

Tree t1, t2;

for (int i = -5; i < 0; i++)

{

t1.insert(i);

}

for (int i = 1; i < 6; i++)

{

t2.insert(i);

}

std::cout << "Preorder t1: ";

t1.preorder\_print();

std::cout << "\n";

std::cout << "Symmetric t2: ";

t2.symmetric\_print();

std::cout << "\n";

const auto t3 = t1 + t2;

std::cout << "Symmetric t1 + t2: ";

t3.symmetric\_print();

std::cout << "\n";

t3.print();

}