

Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентированное программирование»

Выполнил

студент группы 3331506/80401

К. В. Зарубин

Руководитель

Е. М. Кузнецова

«__» _____ 2021 г

Санкт-Петербург

2021

Оглавление

1	ВВЕДЕНИЕ	3
2	ИДЕЯ АЛГОРИТМА.....	4
3	ЭФФЕКТИВНОСТЬ.....	5
4	РЕЗУЛЬТАТЫ РАБОТЫ АЛГОРИТМА	6
5	СПИСОК ЛИТЕРАТУРЫ.....	8
6	ПРИЛОЖЕНИЕ	9

1 ВВЕДЕНИЕ

В работе будет рассмотрен алгоритм сортировки с помощью бинарного дерева. Это универсальный алгоритм сортировки, заключающийся в построении двоичного дерева поиска с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке.

Данный вид сортировки относится к классу сортировок вставками.

Данная сортировка является оптимальной в случаях, когда:

- данные получаются путём непосредственного чтения из потока (файла, сокета или консоли);
- данные уже построены в дерево;
- данные можно считать непосредственно в дерево.

2 ИДЕЯ АЛГОРИТМА

Из элементов массива формируется бинарное дерево поиска, обладающее следующими свойствами:

- оба поддерева – левое и правое – являются двоичными деревьями поиска;
- у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X ;
- у всех узлов правого поддерева произвольного узла X значения ключей данных не меньше, нежели значение ключа данных самого узла X .

Первый элемент – корень дерева, остальные добавляются по следующему методу. Начиная с корня дерева, элемент сравнивается с узлами: если элемент меньше, чем узел, то спускаемся по левой ветке, иначе – по правой; спустившись до конца элемент сам становится узлом.

Построенное таким образом дерево можно легко обойти так, чтобы двигаться от узлов с меньшими значениями к узлам с большими. При этом получаем все элементы в возрастающем порядке. Такой обход называют центрированным (in-order traversal). При таком обходе корень дерева занимает место между результатами соответствующих обходов левого и правого поддерева. Вместе со свойствами бинарного дерева поиска центрированный обход даст отсортированный список узлов. Схема обхода представлена на рисунке 2.1.

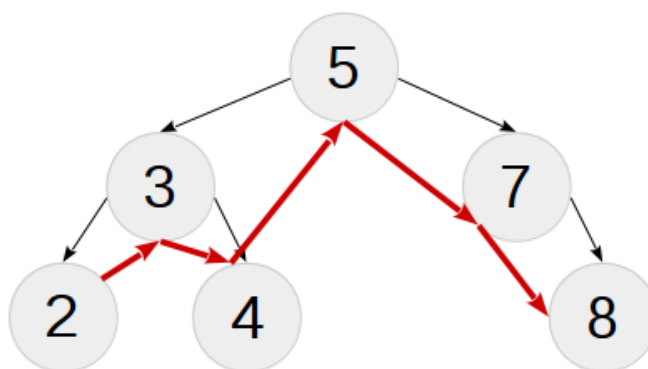


Рисунок 2.1 – Центрированный обход бинарного дерева

3 ЭФФЕКТИВНОСТЬ

Опишем скорость работы алгоритма в зависимости от количества входных данных.

Процедура добавления объекта в бинарное дерево имеет среднюю алгоритмическую сложность порядка $O(\log(n))$, так как для каждого элемента требуется $\log(n)$ сравнений. Соответственно, для n объектов сложность будет составлять $O(n \cdot \log(n))$, что относит сортировку бинарным деревом к группе быстрых сортировок.

Однако сложность добавления объекта в разбалансированное дерево может достигать $O(n)$, что может привести к общей сложности порядка $O(n^2)$.

Данные собраны в таблицу 1.

Таблица 1 – Алгоритмическая сложность сортировки бинарным деревом

Худшая	$O(n^2)$
Средняя	$O(n \cdot \log(n))$
Лучшая	$O(n \cdot \log(n))$

4 РЕЗУЛЬТАТЫ РАБОТЫ АЛГОРИТМА

В качестве входных данных будем подавать массивы разной длины, замеряем количество итераций и время выполнения для каждого случая (результаты представлены в таблице 2).

Таблица 2 – Результаты измерения

Элементы	Итерации	Время, нс	Элементы	Итерации	Время, нс
100	5049	1805	2100	2206049	1300
200	20099	982	2200	2421099	1920
300	45149	662	2300	2646149	1694
400	80199	839	2400	2881199	1581
500	125249	1292	2500	3126249	3678
600	180299	582	2600	3381299	1886
700	245349	593	2700	3646349	3567
800	320399	780	2800	3921399	2737
900	405449	778	2900	4206449	2492
1000	500499	909	3000	4501499	2380
1100	605549	1116	3100	4806549	2874
1200	720599	981	3200	5121599	3276
1300	845649	1467	3300	5446649	2746
1400	980699	1327	3400	5781699	3144
1500	1125749	1580	3500	6126749	3644
1600	1280799	1882	3600	6481799	3844
1700	1445849	2100	3700	6846849	3675
1800	1620899	1762	3800	7221899	4086
1900	1805949	1819	3900	7606949	3987
2000	2000999	1483	4000	8001999	4979

По таблице 2 построены графики зависимости времени выполнения от количества элементов (рисунок 4.1) и зависимости количества итераций от количества элементов (рисунок 4.2).

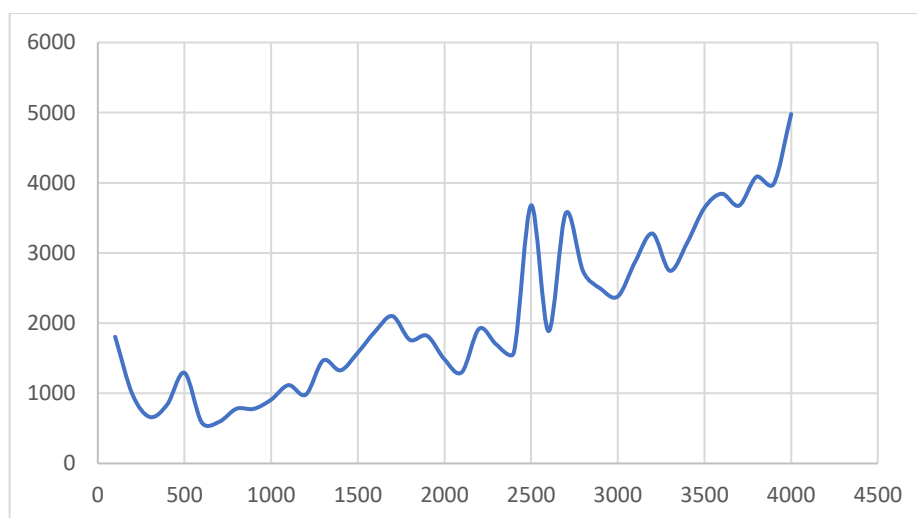


Рисунок 4.1 – График зависимости времени от количества элементов

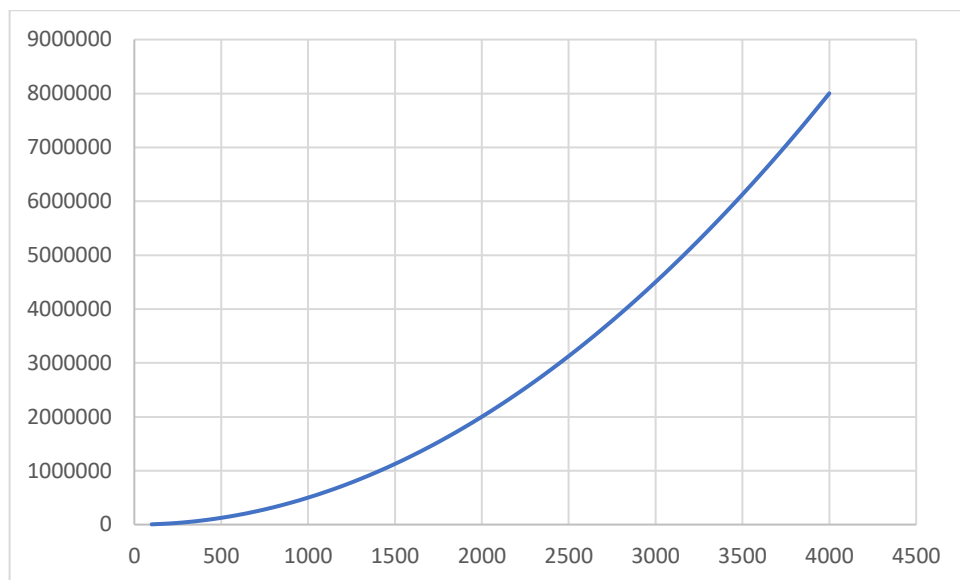


Рисунок 4.2 – График зависимости количества итераций от количества
элементов

5 СПИСОК ЛИТЕРАТУРЫ

- 1) Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ = Introduction to Algorithms. — 2-е. — М.: Вильямс, 2005. — 1296 с.
- 2) Готтшлинг П. Современный C++ для программистов, инженеров и ученых. Серия «C++ In-Depth» = Discovering Modern C++: A Concise Introduction for Scientists and Engineers (C++ In-Depth). — М.: Вильямс, 2016. — 512 с.
- 3) Вирт Н. Алгоритмы и структуры данных — М.: Мир, 1989. — 360 с.


```
#include <algorithm>
#include <vector>
#include <iostream>
#include <memory>
#include <functional>

using namespace std;

struct BinaryTreeNode
{
    shared_ptr<BinaryTreeNode> left, right;
    int key;
};

class BinaryTree
{
public:
    void insert(int key);
    typedef function<void(int key)> Visitor;
    void visit(const Visitor& visitor);

private:
    void visit_recursive(const shared_ptr<BinaryTreeNode> cur_node, const
Visitor visitor);
    shared_ptr<BinaryTreeNode> node_recursive;
    shared_ptr<BinaryTreeNode> m_root;
    void insert_recursive(const shared_ptr<BinaryTreeNode> cur_node, const
shared_ptr<BinaryTreeNode> node_to_insert);
};

// cur_node и node_to_insert не могут быть nullptr
void BinaryTree::insert_recursive(const shared_ptr<BinaryTreeNode> cur_node,
const shared_ptr<BinaryTreeNode> node_to_insert)
{
    if (node_to_insert->key < cur_node->key)
    {
        if (cur_node->left == nullptr)
        {
            cur_node->left = node_to_insert;
            return;
        }
        node_recursive = cur_node->left;
    }
    else
    {
        if (cur_node->right == nullptr)
        {
            cur_node->right = node_to_insert;
            return;
        }
        node_recursive = cur_node->right;
    }
    insert_recursive(node_recursive, node_to_insert);
}

void BinaryTree::insert(int key)
{

```

```

shared_ptr<BinaryTreeNode> node_to_insert(new BinaryTreeNode);
node_to_insert->key = key;

if(m_root == nullptr)
{
    m_root = node_to_insert;
    return;
}

insert_recursive(m_root, node_to_insert);
}

// узлы посещаем в порядке: левое поддерево -> текущий элемент -> правое
// поддерево
// cur_node не может быть nullptr
void BinaryTree::visit_recursive(const shared_ptr<BinaryTreeNode> cur_node,
const Visitor visitor)
{
    if(cur_node->left != nullptr)
        visit_recursive(cur_node->left, visitor);

    visitor(cur_node->key);

    if(cur_node->right != nullptr)
        visit_recursive(cur_node->right, visitor);
}

void BinaryTree::visit(const Visitor& visitor)
{
    if(m_root == nullptr)
        return;
    visit_recursive(m_root, visitor);
}

int main()
{
    BinaryTree tree;
    vector<int> data_to_sort = {0, -1, 2, -3, 4, -5, 6, -7, 8, -9};
    for(int value : data_to_sort)
    {
        tree.insert(value);
    }
    tree.visit([](int visited_key)
    {
        cout<<visited_key<<" ";
    });
    cout<<endl;

    return 0;
}

```