

Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Бинарное сортирующее дерево

Выполнил студент гр. 3331506/10401

Гизатуллин Т.Р.

Преподаватель

Ананьевский М.С.

г. Санкт-Петербург

2024

Оглавление

1.Введение	3
2.Идея алгоритма.....	4
3.Эффективность	7
4.Результаты работы алгоритма	8
5.Заключение	10
6.Список литературы:	11
7.Приложение	12

1.Введение

В работе будет рассмотрен алгоритм сортировки с помощью бинарного сортирующего дерева - Binary Search Trees(BST). Данный вид сортировки относится к классу сортировок вставками. Это универсальный алгоритм сортировки, заключающийся в построении двоичного дерева поиска с последующей сборкой результирующего массива путём обхода узлов построенного дерева в необходимом порядке.

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.

Актуальность данного алгоритма обусловлена следующими факторами:

- 1) В современных системах часто требуется эффективная сортировка, поиск, вставка и удаление данных. BST предлагают логичный и удобный способ хранения данных, который обеспечивает быструю обработку.
- 2) BST используются в базах данных, компиляторах, алгоритмах поиска и сортировки, системах управления памятью и многих других приложениях.
- 3) Понимание принципов работы BST важно для разработки эффективных алгоритмов и структур данных. Это фундаментальная тема, изучаемая в курсе алгоритмов и структур данных.

Основная цель данной курсовой работы — реализация и исследование структуры данных бинарного сортирующего дерева. В курсовой работе также будут рассмотрены преимущества и недостатки алгоритм сортировки с помощью бинарного сортирующего дерева.

2.Идея алгоритма

Бинарное сортирующее дерево (BST) — это структура данных, которая организует элементы таким образом, что каждый узел имеет не более двух потомков. Основное свойство BST состоит в том, что для любого узла все значения в левом поддереве меньше значения этого узла, а все значения в правом поддереве больше или равны значению этого узла. Это свойство позволяет эффективно выполнять все основные операции.

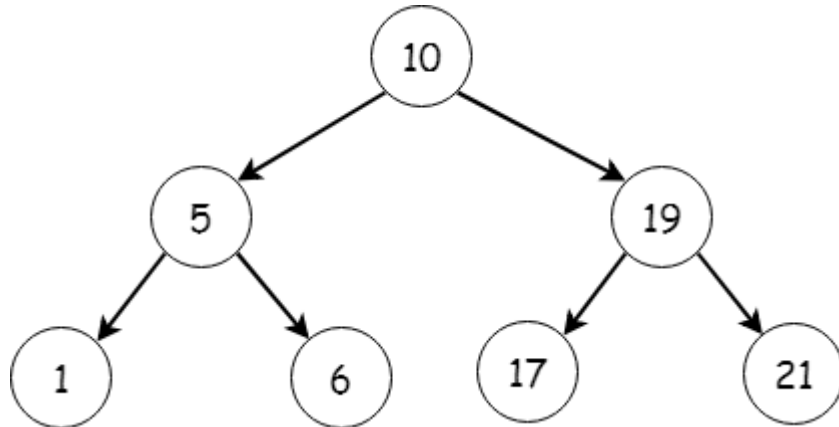


Рисунок 1 – Бинарное сортирующее дерево

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 поддеревьев. Вместе со свойствами бинарного дерева поиска центрированный обход даст отсортированный список узлов. Схема обхода представлена на рисунке 1.

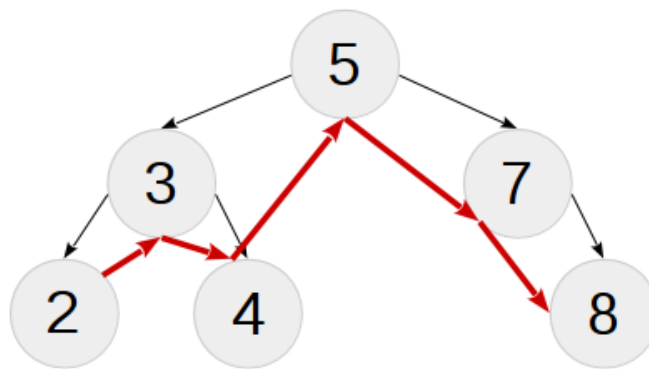


Рисунок 2 – Центрированный обход бинарного дерева

Рассмотрим код по частям:

1. Структура узла бинарного дерева `BinaryTreeNode`:

‘left’ и ‘right’ — указатели на левых и правых потомков, реализованы через умный указатель `shared_ptr`

‘key’ — значение, хранящееся в узле.

2. Класс бинарного дерева `BinaryTree`:

‘insert’ — метод для вставки нового узла в дерево.

‘Visitor’ — тип функции для обхода дерева.

‘visit’ — метод для обхода дерева с использованием переданной функции-обработчика.

‘insert_recursive’ и ‘visit_recursive’ — вспомогательные рекурсивные методы для вставки и обхода соответственно.

‘root’ — корневой узел дерева.

3. Методы вставки нового узла:

‘insert’ - создает новый узел и вставляет его в дерево.

‘insert_recursive’ - рекурсивно находит правильное место для нового узла.

4. Методы обхода:

‘visit’ - начинает обход дерева с корневого узла, если он не пуст.

‘visit_recursive’ - реализует рекурсивный обход дерева в порядке in-order (левый-посетить-правый).

5. Методы удаления:

‘remove’ - вызывает рекурсивную функцию ‘remove_recursive

‘remove_recursive’ – которая возвращает новое поддерево после

удаления узла. Если ключ меньше ключа текущего узла, рекурсивно удаляем из левого поддерева. Если ключ больше, рекурсивно удаляем из правого поддерева.

Если ключ совпадает, то:

Узел имеет только одного потомка или не имеет потомков -

возвращаем соответствующее поддерево. Узел имеет двух потомков -

находим минимальный узел в правом поддерева, заменяем ключ текущего узла на ключ минимального узла и рекурсивно удаляем минимальный узел из правого поддерева.

‘find_min’ - Находит и возвращает узел с минимальным ключом в данном поддерева, двигаясь по левым потомкам.

6. Методы поиска:

‘search’ – вызывает вспомогательный рекурсивный метод поиска.

‘search_recursive’ - рекурсивно ищет ключ в дереве.

Бинарное сортирующее дерево обладает несколькими преимуществами:

- Алгоритм бинарного сортирующего дерева относительно прост для понимания и реализации.
- Требуется меньше дополнительной памяти по сравнению с некоторыми другими алгоритмами сортировки, такими как сортировка слиянием или сортировка кучей.

- Поддержка динамических операций

Недостатки BST:

- Зависимость от удачного выбора корня: Производительность операций в бинарном дереве сильно зависит от выбора корня и структуры дерева.
- Сложность балансировки: Поддержание сбалансированности бинарного дерева может потребовать дополнительных усилий и ресурсов.

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, время выполнения для каждого случая (результаты представлены в таблице 1).

Таблица 1 – Результаты измерения

Элементы	Время, нс		Элементы	Время, нс
100	1805		2100	1300
200	982		2200	1920
300	662		2300	1694
400	839		2400	1581
500	1292		2500	3678
600	582		2600	1886
700	593		2700	3567
800	780		2800	2737
900	778		2900	2492
1000	909		3000	2380
1100	1116		3100	2874
1200	981		3200	3276
1300	1467		3300	2746
1400	1327		3400	3144
1500	1580		3500	3644
1600	1882		3600	3844
1700	2100		3700	3675
1800	1762		3800	4086
1900	1819		3900	3987
2000	1483		4000	4979

По таблице 1 построены графики зависимости времени выполнения от количества элементов (рисунок 3).

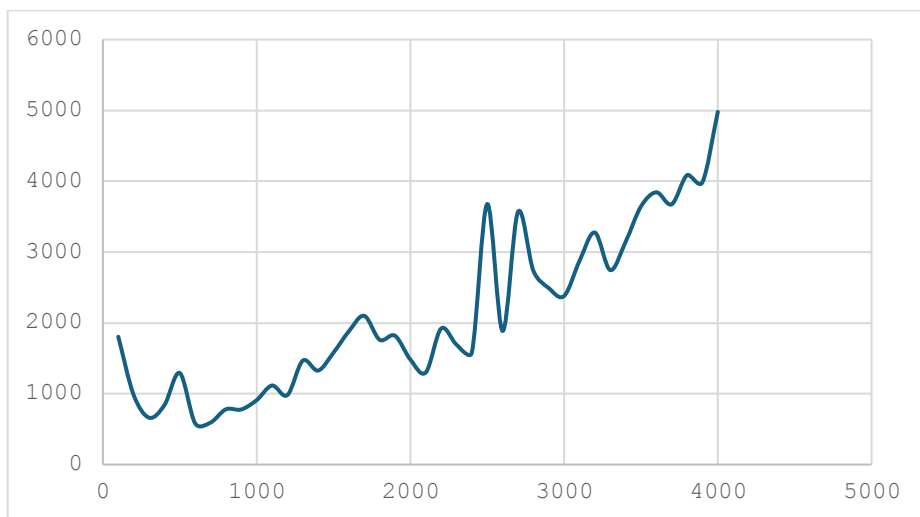


Рисунок 3 – График зависимости времени от количества элементов

7. Заключение

Бинарные деревья — это фундаментальная структура данных в информатике, используемая для эффективного хранения и обработки информации. Алгоритмы бинарных деревьев играют важную роль в реализации различных задач, от поиска и сортировки данных до машинного обучения и искусственного интеллекта. Код курсовой демонстрирует базовые операции с бинарным деревом поиска, включая вставку узлов и обход дерева в порядке возрастания. Это помогает понять, как данный алгоритм можно использовать для сортировки данных.

Основным преимуществом двоичного дерева поиска перед другими структурами данных является возможная высокая эффективность реализации основанных на нём алгоритмов поиска и сортировки.

Двоичное дерево поиска применяется для построения более абстрактных структур, так как множества, мультимножества, ассоциативные массивы.

6. 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

7. Приложение

```
#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);
    void remove(int key);
    bool search(int key) const;
    typedef function<void(int key)> Visitor;
    void visit(const Visitor& visitor) const;

private:
    void insert_recursive(const shared_ptr<BinaryTreeNode>& cur_node, const shared_ptr<BinaryTreeNode>&
node_to_insert);
    void visit_recursive(const shared_ptr<BinaryTreeNode>& cur_node, const Visitor& visitor) const;
    shared_ptr<BinaryTreeNode> root;
    shared_ptr<BinaryTreeNode> remove_recursive(shared_ptr<BinaryTreeNode> cur_node, int key);
    shared_ptr<BinaryTreeNode> find_min(shared_ptr<BinaryTreeNode> cur_node) const;
    bool search_recursive(const shared_ptr<BinaryTreeNode>& cur_node, int key) const;
};

void BinaryTree::insert(int key)
{
    auto node_to_insert = make_shared<BinaryTreeNode>();// new node
    node_to_insert->key = key;

    if (root == nullptr)
    {
        root = node_to_insert;
    }
    else
    {
        insert_recursive(root, node_to_insert);
    }
}

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;
        }
        else
        {
            insert_recursive(cur_node->left, node_to_insert);
        }
    }
    else
    {
        if (cur_node->right == nullptr)
```

```

    {
        cur_node->right = node_to_insert;
    }
    else
    {
        insert_recursive(cur_node->right, node_to_insert);
    }
}
}

```

```

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

```

```

void BinaryTree::visit_recursive(const shared_ptr<BinaryTreeNode>& cur_node, const Visitor& visitor) const
{
    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::remove(int key)
{
    root = remove_recursive(root, key);
}

```

```

shared_ptr<BinaryTreeNode> BinaryTree::remove_recursive(shared_ptr<BinaryTreeNode> cur_node, int key)
{
    if (cur_node == nullptr)
    {
        return cur_node;
    }

    if (key < cur_node->key)
    {
        cur_node->left = remove_recursive(cur_node->left, key);
    }
    else if (key > cur_node->key)
    {
        cur_node->right = remove_recursive(cur_node->right, key);
    }
    else
    {
        //1 or more children
        if (cur_node->left == nullptr)
        {
            return cur_node->right;
        }
        else if (cur_node->right == nullptr)
        {
            return cur_node->left;
        }
        //with 2 children
        shared_ptr<BinaryTreeNode> min_node = find_min(cur_node->right); //in the right subtree
    }
}

```

```

        cur_node->key = min_node->key;
        cur_node->right = remove_recursive(cur_node->right, min_node->key);
    }

    return cur_node;
}

shared_ptr<BinaryTreeNode> BinaryTree::find_min(shared_ptr<BinaryTreeNode> cur_node) const
{
    while (cur_node->left != nullptr)
    {
        cur_node = cur_node->left;
    }
    return cur_node;
}

bool BinaryTree::search(int key) const
{
    return search_recursive(root, key);
}

bool BinaryTree::search_recursive(const shared_ptr<BinaryTreeNode>& cur_node, int key) const
{
    if (cur_node == nullptr)
    {
        return false;
    }

    if (key < cur_node->key)
    {
        return search_recursive(cur_node->left, key);
    }
    else if (key > cur_node->key)
    {
        return search_recursive(cur_node->right, key);
    }
    else
    {
        return true;
    }
}

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;

    tree.remove(-3);
    tree.remove(4);

    tree.visit([](int visited_key) {
        cout << visited_key << " ";
    });
    cout << endl;

    int key_to_search = -5;

```

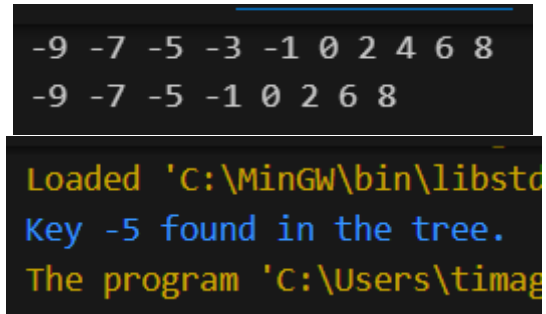
```

if (tree.search(key_to_search)) {
    cout << "Key " << key_to_search << " found in the tree." << endl;
} else {
    cout << "Key " << key_to_search << " not found in the tree." << endl;
}

return 0;
}

```

Результата работы программы:



```

-9 -7 -5 -3 -1 0 2 4 6 8
-9 -7 -5 -1 0 2 6 8
Loaded 'C:\MinGW\bin\libstdc++-6.dll'
Key -5 found in the tree.
The program 'C:\Users\timag...

```

Рисунок 4 – Пример работы программы