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

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Бинарное сортирующее дерево, splay-tree

Выполнил студент гр. 3331506/20101 Хоменко В. С,
Преподаватель Ананьевский М. С.

«___»_____2025 г.

Санкт-Петербург
2025

Введение

В ходе работы будет рассмотрен один из алгоритмов, а именно бинарное (двоичное) дерево поиска.

Задачи:

1. Изучить принципы работы бинарного сортирующего дерева и его применение.
2. Реализовать и изучить структуру данных бинарного сортирующего дерева.

Бинарные деревья – это древовидная структура данных, в которой каждый узел имеет не более двух дочерних узлов. Дочерние узлы называются левым и правым дочерними узлами. Для него выполняются следующие дополнительные условия (свойства дерева поиска) пример представлен н рисунке 1:

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

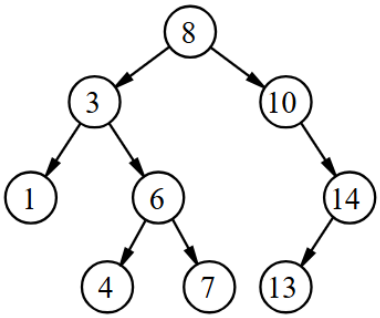

Рисунок 1 - Пример

Бинарные деревья могут применяться для поиска данных в специально построенных деревьях (базы данных), сортировки данных, вычислений арифметических выражений, кодирования (метод Хаффмана) и т.д.

Актуальность: бинарные сортирующие деревья и Splay-деревья играют важную роль в алгоритмах обработки данных. Понимание их сильных и слабых сторон позволяет выбирать оптимальную структуру для конкретных задач (например, приоритетные очереди, кэширование часто используемых элементов).

Основная часть

Дерево – это структура данных, представляющая собой совокупность элементов и отношений, образующих иерархическую структуру этих элементов. Каждый элемент дерева называется вершиной (узлом) дерева. Вершины дерева соединены направленными дугами, которые называют ветвями дерева. Начальный узел дерева называют корнем дерева, ему соответствует нулевой уровень. Листьями дерева называют вершины, в которые входит одна ветвь и не выходит ни одной ветви.

Бинарные деревья имеют два общих представления:

- Представление в виде массива: заполняет массив сверху вниз, слева направо на каждом уровне, и оставляет пустой слот для любого отсутствующего дочернего элемента
- Представление связанного списка: представляет объект узла с помощью данных узла, левого указателя и правого указателя

Сортировка по дереву — это алгоритм сортировки, основанный на структуре данных бинарного дерева поиска. Сначала он создаёт двоичное дерево поиска из элементов входного списка или массива, а затем выполняет обход созданного двоичного дерева поиска в прямом порядке, чтобы получить элементы в отсортированном порядке. То есть если мы будем выполнять операцию обхода дерева, то, записывая все встречающиеся элементы в массив, получим упорядоченное в порядке возрастания множество.

Все вершины, в которые входят ветви, исходящие из одной общей вершины, называются потомками, а сама вершина – предком. Для каждого предка может быть выделено несколько. Уровень потомка на единицу

превосходит уровень его предка. Корень дерева не имеет предка, а листья дерева не имеют потомков.

Высота (глубина) дерева определяется количеством уровней, на которых располагаются его вершины. Высота пустого дерева равна нулю, высота дерева из одного корня – единице. На первом уровне дерева может быть только одна вершина – корень дерева, на втором – потомки корня дерева, на третьем – потомки потомков корня дерева и т.д.

Обход дерева – это упорядоченная последовательность вершин дерева, в которой каждая вершина встречается только один раз.

Рассмотрим алгоритм удаления элемента в дереве поиска без учета баланса. Для удаления концевого элемента или элемента, имеющего только одного потомка, достаточно очевидным образом изменить ссылку от родительской вершины. В случае двух потомков удалить элемент так просто не удастся. Здесь уже имеются три ссылки (от родителя к вершине и от нее к двум потомкам), и их невозможно замкнуть друг на друга. Идея удаления состоит в том, что удаляемая вершина как структурный элемент дерева на самом деле остается на месте, но данные, хранящиеся в этой вершине, заменяются на другие. Поясним это на примере. Пусть мы хотим удалить элементы с ключами 5 и 12 из дерева, изображенного на рисунке 2.

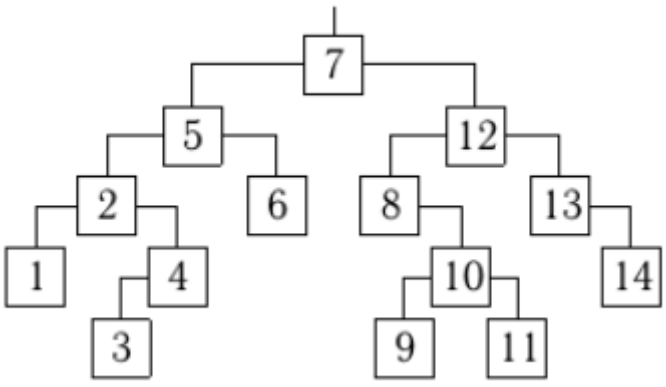

Рисунок 2 – Дерево до удаления

Найдем в дереве вершины, содержимое которых можно подставить вместо удаляемых данных так, чтобы дерево сохранило свойство упорядоченности (осталось деревом поиска). В нашем примере этими

вершинами, очевидно, будут вершины с ключами 4 (для 5) и 11 (для 12). Перенесем данные из этих вершин в «удаляемые» вершины (с ключами 5 и 12) замена значений, должны быть удалены помеченные вершины.

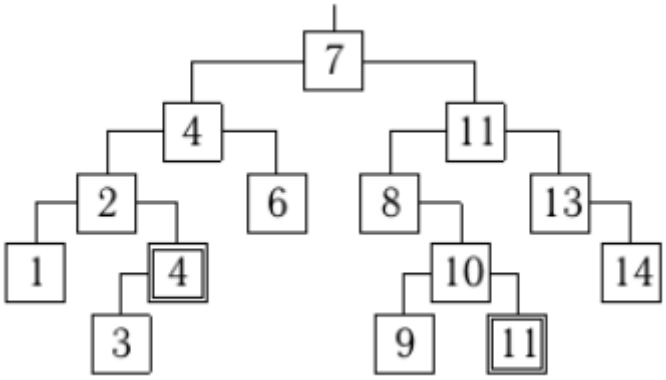

Рисунок 3 - Операция

Теперь можно удалить вершины, хранившие эти данные ранее, поскольку они имеют не более одного потомка.

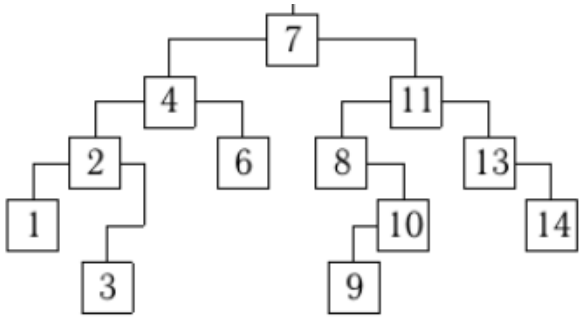

Рисунок 4 – После удаления

Анализируя рассмотренный пример, мы приходим к следующему неформальному алгоритму удаления.

1. Если удаляемая вершина является концевой (не имеет потомков), то достаточно освободить память, занимаемую этой вершиной, и обнулить соответствующую ссылку от родительской вершины

2. Если удаляемая вершина имеет ровно одного потомка, то после удаления вершины из памяти ссылка от родительской вершины переставляется на этого потомка.

3. В случае двух потомков сначала в левом поддереве удаляемой вершины ищется вершина с максимальным ключом. Заметим, что эта максимальная вершина обязательно является либо концевой, либо имеет

только одного потомка. Для того чтобы ее найти, достаточно спускаться по самой правой ветви левого поддерева до тех пор, пока не обнаружится нулевая ссылка направо. Найдя нужную максимальную» вершину, переносим данные из нее в удаляемую вершину. Теперь «максимальную> вершину можно удалить, воспользовавшись пунктом 1 или 2.

Основные термины:

- Корень (Root) — первый узел дерева. У него нет родителя.
- Лист (Leaf) — узел, у которого нет потомков.
- Родитель (Parent) — узел, имеющий потомков.
- Потомки (Children) — узлы, являющиеся дочерними по отношению к другому узлу.
- Глубина узла (Depth) — расстояние от корня до узла.
- Высота узла (Height) — расстояние от узла до самого дальнего листа.
- Поддерево (Subtree) — любое дерево, корнем которого является один из узлов исходного дерева.
- Размер дерева (Size) — общее количество узлов в дереве.

Основными операциями, осуществляемые с бинарными деревьями в коде, являются:

1. Создание бинарного дерева - struct Node;

2. Существование элемента в дереве - bool exist;

4. Вставка элемента в бинарное дерево - Node * insert;

6. Вычисление глубины бинарного дерева - int depth;

7. Удаление элемента бинарного дерева - Node* deleteNode.

Код для бинарного сортирующего дерева представлен в Приложении А.

Эффективность

Процедура добавления объекта в бинарное дерево имеет среднюю алгоритмическую сложность порядка O(log(n)). Соответственно, для n объектов сложность будет составлять O(nlog(n)), что относит сортировку с помощью двоичного дерева к группе «быстрых сортировок». Однако,

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

Наихудшую временную сложность сортировки по дереву можно улучшить, используя самобалансирующееся двоичное дерево поиска, например красно-чёрное дерево, дерево AVL. При использовании самобалансирующегося двоичного дерева сортировка по дереву в худшем случае займёт O(n log n) времени.

При физическом развёртывании древовидной структуры в памяти требуется не менее чем 4n ячеек дополнительной памяти (каждый узел должен содержать ссылки на элемент исходного массива, на родительский элемент, на левый и правый лист), однако, существуют способы уменьшить требуемую дополнительную память.

По написанному коду сгенерировала данные разного размера и посмотрела, сколько времени будет работать алгоритм на них.

Кол-во	t, мс
100	0
1000	0
5000	3
10000	5
15000	10
30000	18
35000	21
40000	24
45000	27
50000	30
55000	32
60000	32

График зависимости приведен на рисунке 3.

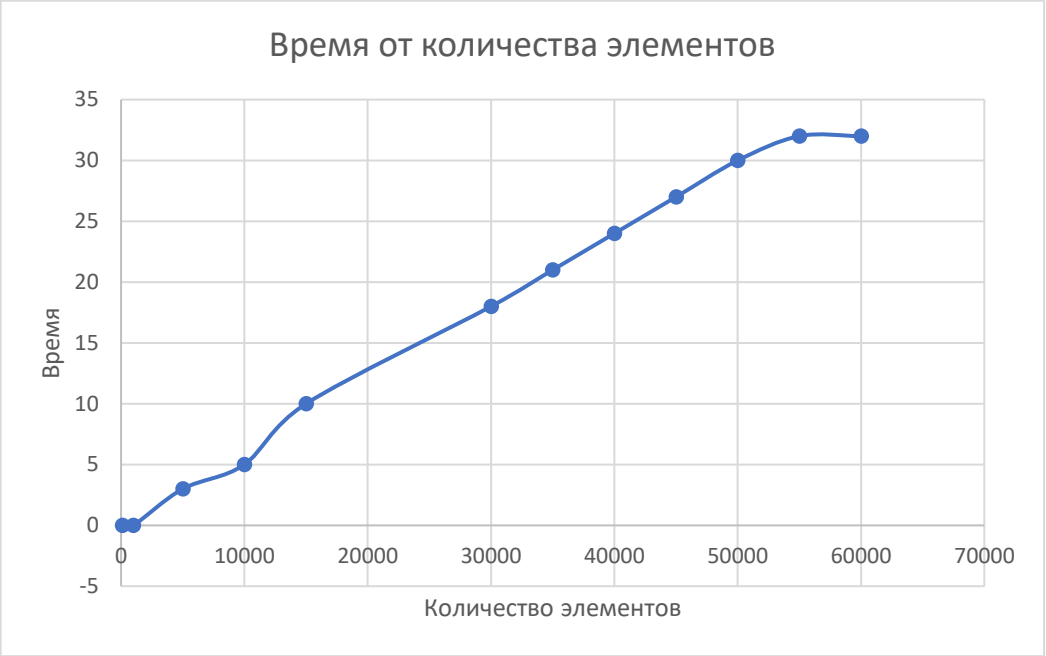

Рисунок 3 – График зависимости

Заключение

В данной курсовой работе был рассмотрен алгоритм - бинарное сортирующее дерево и его практического применения. Бинарное сортирующее дерево представляет собой эффективную структуру данных для организации и обработки упорядоченной информации.

Основное преимущество бинарного сортирующего дерева заключается в его способности поддерживать элементы в частично упорядоченном состоянии, что позволяет быстро находить и извлекать минимальный или максимальный элемент. Эта особенность делает его незаменимым при реализации алгоритмов сортировки, таких как пирамидальная сортировка (Heapsort).

Практическая часть работы включала реализацию бинарного сортирующего дерева и анализ его производительности на данных различного размера. Код курсовой демонстрирует базовые операции с бинарным деревом поиска, включая вставку узлов и обход дерева в порядке возрастания

Список литературы

1. Методы программирования в задачах и примерах на C/C++: учебное издание / В. Д. Валединский [и др.] – Москва: Изд-во Московского. Ун-та, 2023 – 413 с.

2. Introduction to Splay tree data structure. URL: https://www.geeksforgeeks.org/introduction-to-splay-tree-data-structure/ (дата обращения 09.03.2025).

3. C Splay Trees Deleting A Node. URL: https://www.demo2s.com/g/c/describe-the-steps-involved-in-deleting-a-node-from-a-splay-tree-in-c-and-discuss-how-the.html (дата обращения 25.03.2025)

4. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2011 — 1296 с. : ил. — Парал. тит. англ.

5. Алгоритмы. Теория и практическое применение / Род Стивенс – Москва : Издательство «Э», 2016. – 544 с.

6. Структуры данных и проектирование программ [Электронный ресурс] / Р. Л. Круз ; пер. с англ. — 2-е изд. (эл.). —М. : БИНОМ. Лаборатория знаний, 2014 — 765 с.

Приложение А

```
 #include <iostream>

using namespace std;
// создание структуры узла
struct Node {
    int x;
    Node *left; //указатель левого поддерева
    Node *right; //указатель правого поддерева

    Node (int key)
    {
        x = key;
        left = nullptr;
        right = nullptr;
    }
    ~Node()
    {
        delete left;
        delete right;
    }
};

//существование элемента в дереве
bool exist(Node * root, int y)
{
    while (root != nullptr) {
        if (root->x == y) return true;
        root = (y < root->x) ? root->left : root->right;
    }
```

```
    return false;
}

//добавление элемента в дерево
Node * insert (Node * root, int y)
{
    if (exist(root, y)) return root;
    if (root == nullptr) return new Node(y);
        if (y < root->x) {
        root->left = insert(root->left, y);
    }
        else if (y > root->x) {
        root->right = insert(root->right, y);
    }
    return root;
}

//расчет глубины дерева
int depth (Node * root)
{
    if (root == nullptr) return 0;
    return max(depth(root->right), depth(root->left)) + 1;
}

// удаление узла
Node* deleteNode(Node* root, int data) {
    if (root == nullptr) return root;
    // Поиск узла для удаления
    if (data < root->x) {
        root->left = deleteNode(root->left, data);
```

```
    }
    else if (data > root->x) {
        root->right = deleteNode(root->right, data);
    }
    else {
        // Узел с единственной дочерней нулевой ветви или лист
        if (root->left == nullptr) {
            Node *temp = root->right;
            root->right = nullptr;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node *temp = root->left;
            root->left = nullptr;
            delete root;
            return temp;
        }
          // У узла два дочерних узла, взять минимальное значение в правом поддереве
        Node *temp = root->right;
        while (temp && temp->left != nullptr) {
            temp = temp->left;
        }
        // заменить значение удаляемого узла значением найденного узла
        root->x = temp->x;
        // удалить узел
        root->right = deleteNode(root->right, temp->x);
    }
    return root;
}
```

```
void test_performance() {
    // Просто тестируем разных размера
    int sizes[] = {100, 1000, 5000, 10000, 15000, 30000,
        35000, 40000, 45000, 50000, 55000, 60000};

    cout << "Performance test:\n";

    // Цикл по 4 тестовым размерам (0-3)
    for(int i = 0; i < 12; i++) {
        // Создаем пустое дерево для теста
        Node* tree = nullptr;

        int size = sizes[i];
        // Запоминаем начальное время
        clock_t start = clock();

        // Вставляем элементы
        for(int j = 0; j < size; j++) {
            // Вставляем случайное число
            tree = insert(tree, rand() % size);
        }

        // Запоминаем конечное время
        clock_t end = clock();

        // Просто выводим результаты
        cout << size << " elements: "
             << (double)(end - start)/CLOCKS_PER_SEC * 1000
             << " ms\n";
```

```
        delete tree;
    }
}

//вывод
void print(Node * root)
{
    if (root == nullptr) return;
    print(root->left);
    cout << root->x << " ";
    print(root->right);
  }

int main(){
    Node* root = nullptr;

    //вставка элементво в дерево
    root = insert(root, 1344);
    root = insert(root, -20);
    root = insert(root, 44);
    root = insert(root, 17);
    root = insert(root, 9);
    root = insert(root, -1);
    root = insert(root, -110);
    root = insert(root, 912);
    root = insert(root, 2000);

    std::cout<<"listya ";
    print(root);
    cout<<endl;
```

```
    int tree_depth = depth(root);
    std::cout<<"depth "<< tree_depth << endl;

    root = deleteNode(root, 2000);
    std::cout << "";
    print(root);
    cout << endl;

    // Запуск теста производительности
    cout << "\nTest\n";
    test_performance();

    delete root;
    return 0;
}
```