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

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентрованное программирование»

Выполнил
студент группы 3331506/90401 _____ В. Ю. Толстых

Руководитель _____ М. С. Ананьевский

«__»_____2022 г

Санкт-Петербург
2022

Оглавление

1 Введение

В работе будет рассмотрен алгоритм бинарного дерева поиска

Бинарное дерево (рисунок 1.1) – динамическая структура данных, в которой каждый элемент имеет не более двух дочерних узлов

Состоит из корня (первого элемента) и узлов, крайние из которых называются листьями.

Основное назначение бинарных деревьев – поиск и сортировка данных.

Применяется при задачах, например, маршрутизации, где описывается соответствие между адресами назначения и интерфейсами, через которые следует отправить пакет данных до следующего маршрутизатора.. Маршрутизаторы с близкими значениями в итоге хранятся рядом.

Вопросы машинного обучения тоже затрагивают бинарные деревья (деревья решений), когда эмулируется процесс принятия решений нейросетью.

Администрирование баз данных может осуществляться с помощью деревьев так как позволяет быстро менять данные.

Бинарные деревья позволяют также сжимать данные, расставляя элементы по частоте их появления: чем чаще, тем ближе к корню дерева

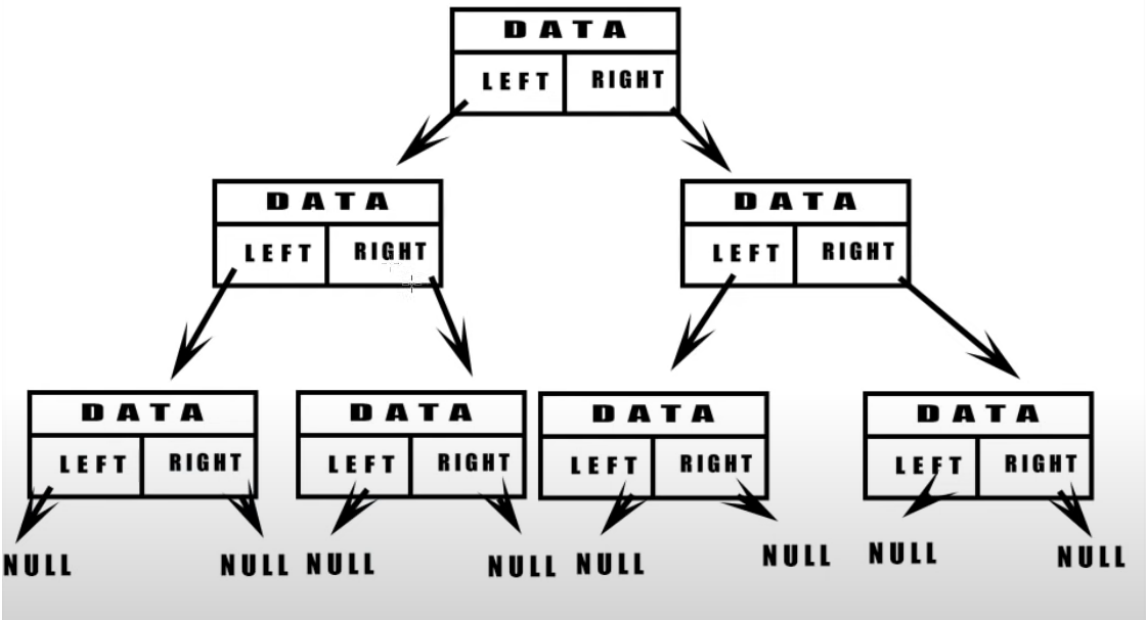

Рисунок 1.1 – Графическое представление бинарного дерева поиска

2 Описание алгоритма

Как уже было отмечено, максимальное количество дочерних элементов у узла – 2. Соответственно, каждый элемент дерева содержит какую-то информацию и указатели на дочерние узлы.

Алгоритм бинарного дерева поиска подразумевает сортировку входящих данных. Записываемое значение записывается в правый узел, если оно больше родительского, и в левый, если оно меньше его. Повторяющихся узлов при этом быть не должно.

Принцип проиллюстрирован на рисунке 2.1

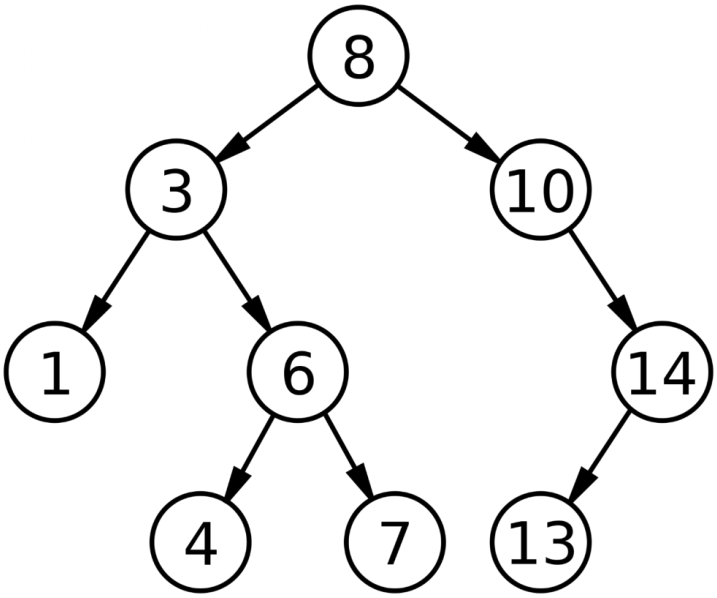

Рисунок 2.1 – Принцип заполнения бинарного дерева поиска

Двоичное дерево может быть логически разбито на уровни. Корень дерева является нулевым уровнем, потомки корня – первым уровнем, их потомки вторым, и т.д. Глубина дерева это его максимальный уровень.

Так как поддерево является той же самой структурой, то бинарное дерево поиска реализуется рекурсивным алгоритмом

С бинарным деревом поиска выполняются 3 базовые операции:

- Поиск элемента

- Добавление элемента

- Удаление узла

3 Эффективность

Благодаря сортированности, работа алгоритма по поиску, вставке, удалению в лучшем равна $O(\log_2(n))$, а в худшем будет O(n). В общем случае временная сложность O(h), где h – высота дерева.

Лучший случай подразумевает сбалансированное дерево (также называется AVL-дерево), когда каждый уровень имеет полный набор вершин (рисунок 3.1)

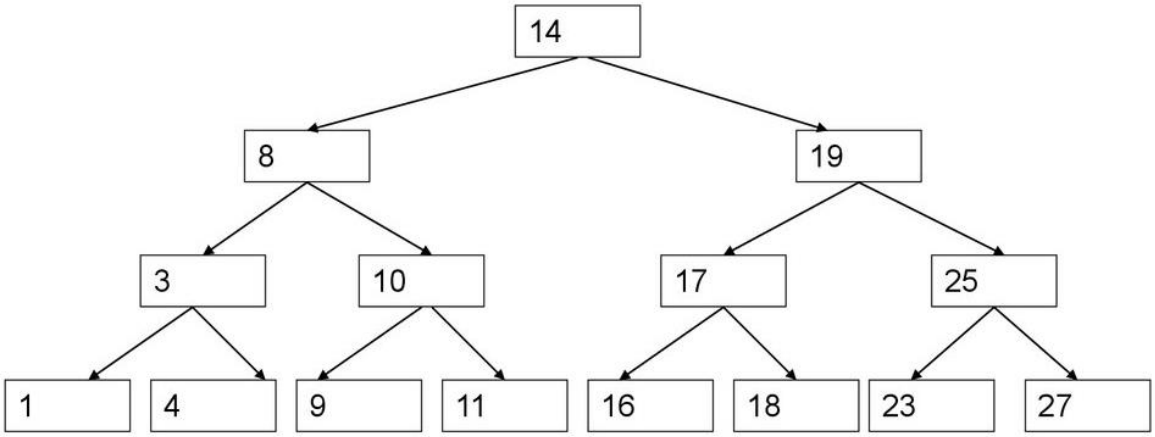

Рисунок 3.1 – Сбалансированное бинарное дерево поиска

Худший случай подразумевает, что в дерево был подан уже отсортированный список элементов, которые просто следуют друг за другом по порядку, от корня, к листу. В таком случае придется перебрать все элементы до нужного

Такой вид дерева называется вырожденным (рисунок 3.2)

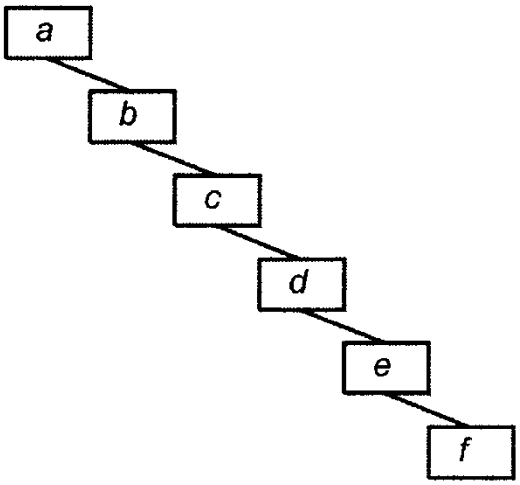

Рисунок 3.2 – Вырожденное бинарное дерево поиска

Основное преимущество бинарного дерева поиска в том, что путь к данным достаточно короткий. Предположим, у нас есть база данных в виде сбалансированного бинарного дерева на миллион элементов, на второй же итерации поиска алгоритм «откинет», если данные распределены равномерно по дереву, половину всех данных, потому что там точно нет искомого элемента. На следующей итерации будет отброшена ещё четверть и так далее.

Этим дерево выгодно отличается от, например, списков, где приходится проходится по всем элементам до нахождения позиции нужного, а от массивов тем, что не нужно записывать новые значения вместе с уже имеющимися в по-новой созданный динамический массив.

На рисунке 3.3 представлен график производительности работы одного из методов, insert, в зависимости от входящего числа элементов.

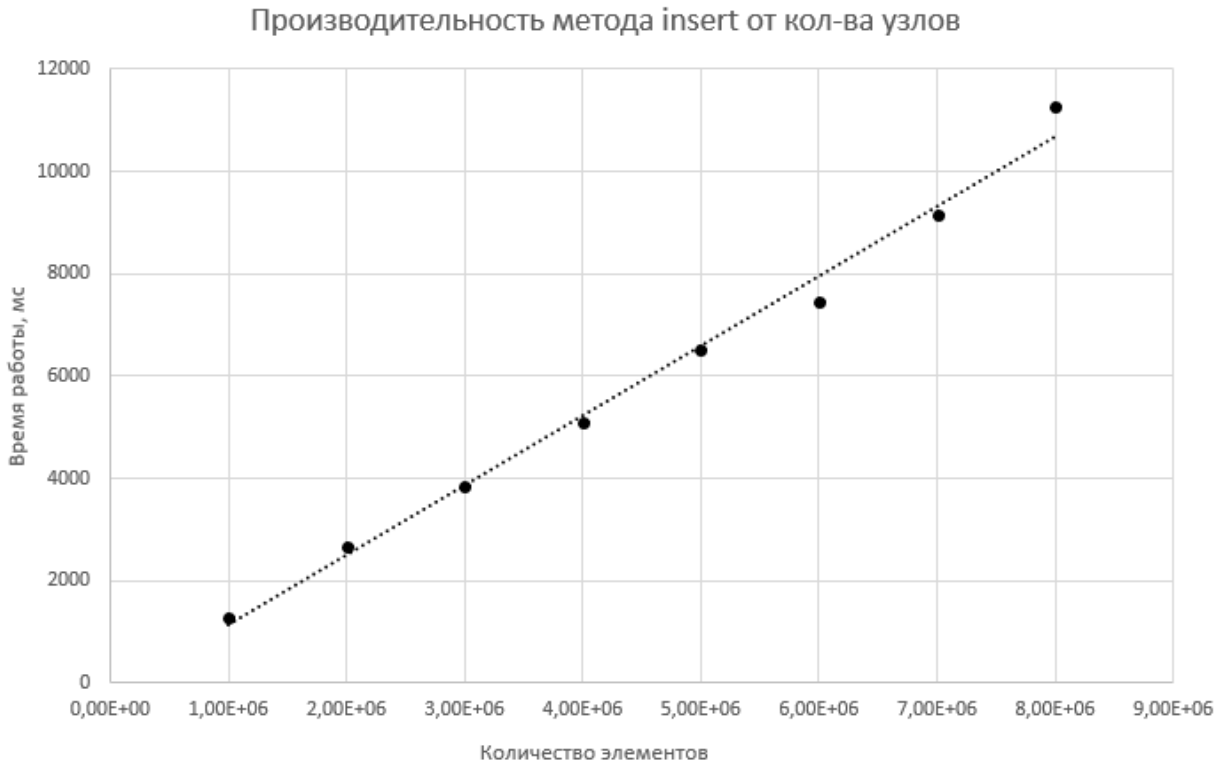

Рисунок 3.3 – График производительности insert

Результат оказался, как у вырожденного дерева, так как на первом тесте заполнение происходило значениями от 0 до 1 000 000. Математическое ожидание 500 000, а дисперсия очень велика. Поэтому несмотря на рандомное заполнение, дерево заполняется дисбалансно. Дальше разброс значений увеличивается пропорционально количеству узлов.

4 Заключение

В целом, бинарное дерево поиска является довольно эффективным алгоритмом работы с базами данных, причём, чем больше элементов содержится в дереве, тем эффективнее будет поиск в сравнении со списками.

Но есть проблемы с удалением элемента, так как удаляя элемент, мы удаляем и все потомственные узлы, поэтому бинарное дерево поиска больше приспособлено именно для поиска.

Метод хорошо подходит для реализации сортировки данных, операций с приоритетной очередью, а также алгоритм позволяет использовать возможности словаря, но только с одним значением к одному ключу.

5 Список литературы

1. Левитин А. Алгоритмы: введение в разработку и анализ – М.: Вильямс, 2006.

2. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ – М.: Вильямс, 2005

6 Приложение

```cpp
#include <iostream>
#include <vector>
using namespace std;

class Node {
    private:
        int data;
        Node* left;
        Node* right;
    public:
        Node(int data) :data(data), left(nullptr), right(nullptr) {}
        void insert(int a) {
            if (a > data && right) right->insert(a);
            else if (a > data && !right) right = new Node(a);
            else if (a < data && left) left->insert(a);
            else left = new Node(a);
        }
        void print() {
            if (left) left->print();
            cout << data << endl;
            if (right) right->print();
        }
        int find(int data) {
            if (this->data == data) {
                return data;
            }
            else if (this->data > data) {
                this->left->find(data);
            }
            else {
                this->right->find(data);
            }
        }
        void remove(int data) {
            if (this->data == data) {
                if (left) {
                    left->remove(data);
                }
                if (right) {
                    right->remove(data);
                }
                delete[] this;
            }
            else if (this->data > data) {
                this->left->find(data);
            }
            else {
                this->right->find(data);
            }
        }
    };
```

```
53  class Tree {
54  private:
55      Node* root;
56  public:
57      Tree() :root(nullptr) {}
58      void insert(int);
59      void print();
60      int find(int);
61      void remove(int);
62  };
63
64  void Tree::insert(int data) {
65      if (!root) root = new Node(data);
66      else root->insert(data);
67  }
68
69  void Tree::print() {
70      root->print();
71  }
72
73  int Tree::find(int data) {
74      return root->find(data);
75  }
76
77  void Tree::remove(int data) {
78      root->remove(data);
79  }
80
81  void test_insert_time(int nodes_count) {
82
83      vector<int> values(nodes_count);
84      for (int i = 0; i < values.size(); ++i) {
85          values[i] = rand() % nodes_count;
86      }
87      Tree* tree = new Tree();
88      int start = clock();
89      for (int i = 0; i < values.size(); i++) {
90          tree->insert(values[i]);
91      }
92      int end = clock();
93      cout << "nodes_count: " << nodes_count << ", milliseconds: ";
94      cout << (end - start) * 1000 / CLOCKS_PER_SEC << endl;
95  }
```