

Лабораторная работа №7 по дисциплине «Типы и структуры данных»

Сбалансированные деревья, хеш –таблицы

Цель работы:

Цель работы – построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска и в хеш-таблицах. Сравнить эффективность устранения коллизий при внешнем и внутреннем хешировании.

Задание (Вариант 1):

Построить хеш-таблицу по указанным данным. Сравнить эффективность поиска в сбалансированном двоичном дереве, в двоичном дереве поиска и в хеш-таблице (используя открытую и закрытую адресацию). Вывести на экран деревья и хеш-таблицы. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если среднее количество сравнений больше указанного. Оценить эффективность использования этих структур (по времени и памяти) для поставленной задачи. Оценить эффективность поиска в хеш-таблице при различном количестве коллизий и при различных методах их разрешения.

Используя предыдущую программу (задача №6), сбалансировать полученное дерево. Вывести его на экран в виде дерева. Построить хеш-таблицу из чисел файла. Осуществить поиск введенного целого числа в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Сравнить время поиска, объем памяти и количество сравнений при использовании различных структур данных.

Исходные данные:

Программа предназначена для работы с текстовым файлом. Файл может содержать произвольное количество целых чисел.

Тесты

Если структура не содержит удаляемого числа будет выведено предупреждение об отсутствии числа.

В случае некорректного выбора меню, будет выведено предупреждение
"Команда не найдена"

Интерфейс программы:

1) Главное меню

Выберите одно из следующих действий:

- 0: Загрузить данные из файла
- 1: Работа с bst деревом
- 2: Работа с avl деревом
- 3: Работа с хэш таблицей с открытой адресацией
- 4: Работа с хэш таблицей с закрытой адресацией
- 5: Сравнение времени поиска
- 6: Закончить работу

2) Отображение меню дерева

Выберите одно из следующих действий:

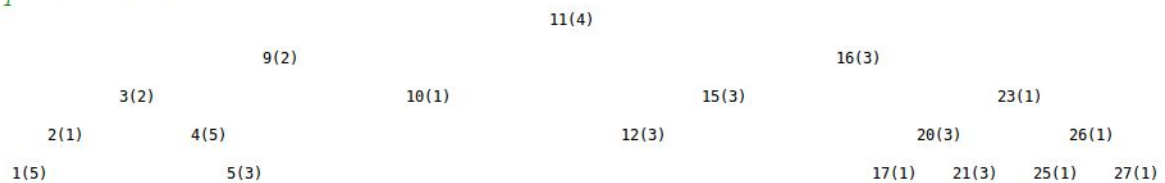
- 1: Отобразить дерево (1 способ)
- 2: Отобразить дерево (2 способ)
- 3: Добавить число в дерево
- 4: Удалить число из дерева
- 5: Удалить все вхождения числа в дереве
- 6: Поиск числа
- 7: Закончить работу

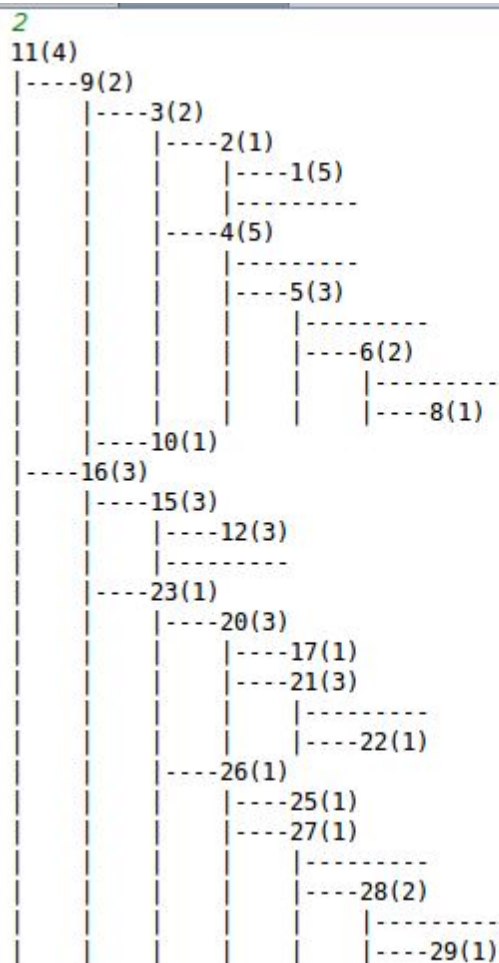
1

3) Отображение дерева

.. закончить работу

1





4) Отображение меню Хеш таблицы

Выберите одно из следующих действий:

- 1: Отобразить таблицу
- 2: Добавить число в таблицу
- 3: Удалить число из таблицы
- 4: Удалить все вхождения числа в таблице
- 5: Поиск числа
- 6: Закончить работу

5) Отображение Хеш таблицы

```

0:16(3)
1:1(5) 17(1)
2:2(1)
3:3(2)
4:20(3) 4(5)
5:5(3) 21(3)
6:6(2) 22(1)
7:23(1)
8:8(1)
9:9(2) 25(1)
10:26(1) 10(1)
11:11(4) 27(1)
12:28(2) 12(3)
13:29(1)
14:
15:15(3)

```

Внутренние структуры данных:

Структура AVL дерева

```
class AVL
{
private:
    int height(AVL_Node<T> *t);

    int difference_of_h(AVL_Node<T> *t);

    int cnt_value(AVL_Node<T> *t);

    T key_value(AVL_Node<T> *t);

    void right(AVL_Node<T> *&t);

    void left(AVL_Node<T> *&t);

    void balance(AVL_Node<T> *&t); // балансировка узла p
    void add_to_avl(AVL_Node<T> *&t, AVL_Node<T> *tmp);

    AVL_Node<T> *findmin(AVL_Node<T> *t);

    AVL_Node<T> *removemin(AVL_Node<T> *t);

    bool delete_from_avl(AVL_Node<T> *&t, T x);

    //int value_by_index(AVL_Node* t, int ind);
    //int index(AVL_Node* t, int x);
    void free_tree(AVL_Node<T> *&tmp);

    AVL_Node<T> *head = NULL;
    int count_of_element = 0;
public:
    AVL() {}

    int Memory();

    void Insert(T x);

    bool Remove(T x);

    bool Search(T x);

    ~AVL();

    template<typename X>
    friend void print(X *tmp, int deep, bool flag);

    template<typename X>
    friend void show_as_tree(X *tree);

    template<typename X>
    friend void show(X *tree);
};
```

Структура дерева поиска

```
class BST
{
private:
    element<T> *head = NULL;
    int count_of_element = 0;

    void delete_tree(element<T> *tmp);

    void operator_copy(element<T> **head, element<T> *tmp);

    void delete_remove(element<T> *prev, element<T> *tmp);

public:
    BST();

    BST(const BST &obj);

    BST<T> &operator=(const BST &obj);

    int Memory();

    void Insert(T x);

    bool Remove(T x);

    bool Search(T x);

    ~BST();

    template<typename X>
    friend void print(X *tmp, int deep, bool flag);

    template<typename X>
    friend void show_as_tree(X *tree);

    template<typename X>
    friend void show(X *tree);

};
```

Структура Хеш таблицы с открытой адресацией

```
class HashTableOpen
{
private:
    ElementTable<T> **table;
    bool *status;
    int tableSize;
    int count_of_elem;
public:
    HashTableOpen(int Start_size);

    ~HashTableOpen();

    void New_Table(int new_size);

    bool Insert(T k);

    bool Search(T x);

    int Memory();

    bool Delete_element(T x);

    void Show();

};
```

Структура Хеш таблицы с закрытой адресацией

```
class HashTableClose
{
private:
    T *table;
    short int *status;
    int tableSize;
    int count_of_elem;
public:
    HashTableClose(int Start_size);

    ~HashTableClose();

    void New_Table(int new_size);

    bool Insert(T k);

    bool Search(T x);

    int Memory();

    bool Delete_element(T x);

    void Show();
};
```

Особенности реализации:

Чтобы избавиться от накопления коллизий при превышении определенного количества, происходит перехэширование таблицы.

Для таблицы с открытой адресацией :если отношение кол-ва элементов к размеру таблицы равно 2.

Для закрытой адресации: отношение кол-ва элементов к размеру таблицы больше 3/4.

Сравнение поиска в различных структурах данных:

Введите число:

20

Время поиска

Бинарное дерево:

Время работы(мкс):5

Кол-во сравнений: 7

Объем памяти(байт): 552

Avl дерево:

Время работы(мкс):3

Кол-во сравнений: 7

Объем памяти(байт): 736

Таблица с открытой адресацией:

Время работы(мкс):2

Кол-во сравнений: 1

Объем памяти(байт): 368

Таблица с закрытой адресацией:

Время работы(мкс):2

Кол-во сравнений: 2

Объем памяти(байт): 256

Введите число:

16

Время поиска

Бинарное дерево:

Время работы(мкс):2

Кол-во сравнений: 3

Объем памяти(байт): 552

Avl дерево:

Время работы(мкс):0

Кол-во сравнений: 1

Объем памяти(байт): 736

Таблица с открытой адресацией:

Время работы(мкс):1

Кол-во сравнений: 1

Объем памяти(байт): 368

Таблица с закрытой адресацией:

Время работы(мкс):1

Кол-во сравнений: 2

Объем памяти(байт): 256

Введите число:

26

Время поиска

Бинарное дерево:

Время работы(мкс):1

Кол-во сравнений: 7

Объем памяти(байт): 552

Avl дерево:

Время работы(мкс):1

Кол-во сравнений: 3

Объем памяти(байт): 736

Таблица с открытой адресацией:

Время работы(мкс):1

Кол-во сравнений: 1

Объем памяти(байт): 368

Таблица с закрытой адресацией:

Время работы(мкс):1

Кол-во сравнений: 2

Объем памяти(байт): 256

Введите число:

100

Время поиска

Бинарное дерево:

Время работы(мкс):2

Кол-во сравнений: 14

Объем памяти(байт): 552

Avl дерево:

Время работы(мкс):2

Кол-во сравнений: 8

Объем памяти(байт): 736

Таблица с открытой адресацией:

Время работы(мкс):2

Кол-во сравнений: 2

Объем памяти(байт): 368

Таблица с закрытой адресацией:

Время работы(мкс):1

Кол-во сравнений: 8

Объем памяти(байт): 256

Таблицы дают относительно стабильное время поиска и кол-во сравнений. В дереве результат поиска зависит от расположения символа в дереве(AVL):

16 - корень дерева;

26 -2 уровень;

20- одна из листовых вершин;

1000 -отсутствует в дереве

Анализ использования памяти:

Деревья - структуры, память на которые выделяется по мере добавление элементов. Объем памяти прямо пропорционален количеству элементов.

Таблица с открытой адресацией требует больше памяти по сравнению с закрытой, так как таблица с открытой адресацией может содержать пустые указатели. В деревьях помимо значения необходимо хранить указатель на потомков.

Вывод:

Основным преимуществом деревьев является возможная высокая эффективность реализации основанных на нём алгоритмов поиска и сортировки.

Хеш таблицы используют меньше памяти, и требуют минимального количества операций сравнения при поиске. Таблицы также требуют качественной хеш-функции для избегания коллизий.

Контрольные вопросы

1. Чем отличается идеально сбалансированное дерево от AVL дерева?

Если при добавлении узлов в дерево располагать их равномерно слева и справа, то получится дерево, у которого число вершин в левом и правом поддеревьях отличается не более, чем на единицу. Такое дерево называется идеально сбалансированным.

В AVL дереве высота левого и правого поддеревьев отличается не более, чем на единицу.

2. Чем отличается поиск в AVL-дереве от поиска в дереве двоичного поиска?

Поиск в AVL дереве имеет сложность $O(\log_2 n)$, в то время как в обычном ДДП может иметь $O(n)$.

3. Что такое хеш-таблица, каков принцип ее построения?

Хеш-таблицей называется массив, заполненный элементами в порядке, определяемом хеш-функцией. Хеш-функция каждому элементу таблицы ставит в соответствие некоторый индекс.

4. Что такое коллизии? Каковы методы их устранения.

Коллизия – ситуация, когда разным ключам хеш-функция ставит в соответствие один и тот же индекс. Основные методы устранения коллизий: открытое и закрытое хеширование.

При открытым хешировании, конфликтующие ключи просто добавляются в список, находящийся по их общему индексу. Поиск по ключу сводится к определению индекса, а затем к поиску ключа в списке перебором.

При закрытом хешировании, конфликтующий ключ добавляется в первую свободную ячейку после «своего» индекса. Поиск по ключу сводится к определению начального приближения, а затем к поиску ключа методом перебора.

5. В каком случае поиск в хеш-таблицах становится неэффективен?

Поиск в ХТ становится неэффективен при большом числе коллизий – сложность поиска возрастает по сравнению с $O(1)$.

6. Эффективность поиска в AVL деревьях, в дереве двоичного поиска и в хеш-таблицах.

В хеш-таблице минимальное время поиска $O(1)$. В AVL: $O(\log_2 n)$. В дереве двоичного поиска $O(h)$, где h - высота дерева (от $\log_2 n$ до n).