Отчёт

Лабораторная работа 6 по предмету «Типы и структуры данных».
Керимов Ахмед, ИУ7-34Б.

Цель работы — получить навыки применения двоичных деревьев, реализовать основные операции над деревьями: обход деревьев, включение, исключение и поиск узлов; построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска и в хеш-таблицах.

Техническое задание

В текстовом файле содержатся целые числа. Построить ДДП из чисел файла. Вывести его на экран в виде дерева. Сбалансировать полученное дерево и вывести его на экран. Построить хеш-таблицу из чисел файла. Использовать метод цепочек для устранения коллизий. Осуществить удаление введённого целого числа в ДДП, в сбалансированном дерево, в хеш-таблицу и в файл. Сравнить время удаления, объем памяти и количество сравнений при использовании различных структур данных. Если количество сравнений в хеш-таблице больше указанного, то произвести реструктуризацию таблицы, выбрав другую функцию.

Аварийные ситуации

1. Неправильный ввод чего угодно. Будет выведено сообщение об ошибке.
2. Попытка удалить несуществующий элемент или добавить существующий. Будет выведено сообщение.

Структуры данных

```
template <typename T>
struct BinaryTreeNode {
    BinaryTreeNode* left;
    BinaryTreeNode* right;

    T key;
    int height;
};
```

```
template <typename T>
struct SLLNode {
    T key;
    SLLNode* next;
};
```

```
template <typename T>
struct HashTable {
    SLLNode<T>** table;
    int table_size;
};
```

Алгоритмы

Двоичное дерево

Вставка

```
template <typename T>
BinaryTreeNode<T>* insert(BinaryTreeNode<T>* node, T key) {
    if (!node)
        return new BinaryTreeNode<T>(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    node->height = std::max(height_of(node->left), height_of(node->right)) + 1;

    return node;
}
```

Удаление

```
template <typename T>
BinaryTreeNode<T>* remove(BinaryTreeNode<T>* node, T key, int& counter) {
    if (!node) {
        counter = -counter;
        return node;
    }

    if (++counter, (key < node->key))
        node->left = remove(node->left, key, counter);
    else if (++counter, (key > node->key))
        node->right = remove(node->right, key, counter);
    else {
        BinaryTreeNode<T>* tmp;
        if (!node->left) {
            tmp = node->right;
            delete node;
            return tmp;
        }
        else if (!node->right) {
            tmp = node->left;
            delete node;
            return tmp;
        }

        tmp = min_node(node->right);
```

```
        node->key = tmp->key;
        node->right = remove(node->right, tmp->key, counter);
    }

    node->height = std::max(height_of(node->left), height_of(node->right)) + 1;
    return node;
}
```

Удаление с балансировкой

```
// counter - количество сравнений, при попутке удаления несуществующего
// значения counter становится отрицательным
template <typename T>
BinaryTreeNode<T>* remove(BinaryTreeNode<T>* node, T key, int& counter) {
    if (!node) {
        counter = -counter;
        return nullptr;
    }
    if (++counter, (key == node->key)) {
        BinaryTreeNode<T>* left = node->left;
        BinaryTreeNode<T>* right = node->right;
        delete node;
        if (!right)
            return left;
        node = right->_find_min();
        node->right = right->_remove_min(counter);
        node->left = left;
    }
    else if (++counter, (key < node->key))
        node->left = remove(node->left, key, counter);
    else
        node->right = remove(node->right, key, counter);

    if (counter <= 0)
        return node->balance();
    return node->balance(counter);
}
```

балансировка

```
template <class T>
BinaryTreeNode<T>* BinaryTreeNode<T>::balance(int& counter) {
    update_fields();

    if (++counter, (balance_factor() == 2)) {
        if (++counter, (right->balance_factor() < 0))
            right = right->rotate_right();
        return rotate_left();
    }
    if (++counter, (balance_factor() == -2)) {
        if (++counter, (left->balance_factor() > 0))
            left = left->rotate_left();
        return rotate_right();
    }

    return this;
}
```

Хеш-таблица

```
int hash(const int &elem, int m) {
    return (elem % m + m) % m;
}

int get_next_simple(int n) {
    for (int i = n + 1;; ++i) {
        bool is_prime = true;
        for (int j = 2; j * j <= i; ++j) {
            if (!(i % j)) {
                is_prime = false;
                break;
            }
        }
        if (is_prime)
            return i;
    }
}

template <typename T>
bool HashTable<T>::add(const T& elem) {
    if (has(elem) > 0)
        return false;

    const int key = hash(elem, table_size);
    table[key] = new Node(elem, table[key]);

    return true;
}

template <typename T>
int HashTable<T>::remove(const T& elem) {
    int counter = 0;
    const int key = hash(elem, table_size);

    SLLNode* parent = table[key];
    if (!parent)
        return -counter;

    if (++counter, (parent->elem == elem)) {
        table[key] = parent->next;
        parent->next = nullptr;
        delete parent;
        if (counter > MAX_COMP_NUM)
            rehash();
        return counter;
    }

    SLLNode* node = parent->next;
    while (node) {
        if (++counter, (node->elem == elem)) {
            parent->next = node->next;
            delete node;
            if (counter > MAX_COMP_NUM)
                rehash();
            return counter;
```

```
        }
        parent = node;
        node = parent->next;
    }

    if (counter > MAX_COMP_NUM)
        rehash();

    return -counter;
}

template <typename T>
int HashTable<T>::has(const T& elem) {
    int counter = 0;
    const int key = hash(elem, table_size);
    SLLNode* node = table[key];
    while (node) {
        ++counter;
        if (node->elem == elem) {
            if (counter > MAX_COMP_NUM)
                rehash();
            return counter;
        }
        node = node->next;
    }
    if (counter > MAX_COMP_NUM)
        rehash();
    return -counter;
}

template <class T>
void HashTable<T>::rehash() {
    int old_size = table_size;
    table_size = get_next_simple(table_size);

    SLLNode** new_table = new SLLNode*[table_size];
    for (int i = 0; i != table_size; ++i)
        new_table[i] = nullptr;

    for (int i = 0; i != old_size; ++i) {
        SLLNode* node = table[i];
        while (node) {
            const int key = hash(node->elem, table_size);
            new_table[key] = new SLLNode(node->elem, new_table[key]);
            SLLNode* tmp = node;
            node = node->next;
            delete tmp;
        }
    }

    delete[] table;
    table = new_table;
}
```

Результаты измерения времени и памяти

	10	100	100 отсортированных

	элементов	элементов	элементов
кол-во сравнений (ДДП)	6.1	13.2	200
кол-во сравнений (АВЛ)	8.2	21.5	21.3
кол-во сравнений (ХТ)	1.6	2.1	3
время (ДДП), нс	124	1883	3188
время (АВЛ), нс	146	1588	1484
время (ХТ), нс	98	898	1595
память (ДДП)	164	1604	1604
память (АВЛ)	164	1604	1604
память (ХТ)	88	898	1595

Вопросы

1. *Что такое дерево?*

Дерево – это нелинейная структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим». Дерево с базовым типом T определяется рекурсивно либо как пустая структура (пустое дерево), либо как узел типа T с конечным числом древовидных структур этого же типа, называемых поддеревьями.

2. *Как выделяется память под представление деревьев?*

Выделение памяти под деревья определяется типом их представления. Это может быть таблица связей с предками (№ вершины - № родителя), или связный список сыновей. Оба представления можно реализовать как с помощью матрицы, так и с помощью списков. При динамическом представлении деревьев (когда элементы можно удалять и добавлять) целесообразнее использовать списки – т.е. выделять память под каждый элемент динамически.

3. *Какие стандартные операции возможны над деревьями?*

Основные операции с деревьями: обход (инфиксный, префиксный, постфиксный), поиск элемента по дереву, включение и исключение элемента из дерева.

4. *Что такое дерево двоичного поиска?*

Дерево двоичного поиска – это такое дерево, в котором все левые потомки моложе предка, а все правые – старше.

5. *Чем отличается идеально сбалансированное дерево от АВЛ 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. Идеальная балансировка даёт наименьшую высоту дерева. В АВЛ дереве высота левого и правого поддеревьев отличается не более, чем на единицу.

6. *Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска?*

Поиск в АВЛ дереве имеет трудоёмкость $O(\log n)$, в то время как в обычном ДДП может иметь $O(n)$. АВЛ дерево никогда не вырождается в линейный список (исключение – дерево из двух элементов), в то время как «внешний вид» ДДП может зависеть от того, в каком порядке в него добавлялись элементы.

7. *Что такое хеш-таблица, каков принцип её построения?*

Массив, заполненный в порядке, определенным хеш-функцией, называется хеш- таблицей. Принцип построения основывается на том, что добавляемый элемент помещается в ячейку, индекс которой совпадает со значением хеш-функции для этого элемента.

8. *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.

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

В случае большого количества коллизий, т. е. большого количества элементов с одинаковым хешем или сильной заполненности таблицы.

10. *Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска и в хеш-таблицах.*

В АВЛ деревьях поиск эл-та выполняется за $O(\log n)$, в дереве двоичного поиска за $O(n)$ — в худшем случае (если дерево вырождается в список), в хеш-таблицах при реализации методом цепочек в среднем за $O(1 + a)$, для открытой адресации $O(1/(1— a))$, где a — коэффициент заполнения таблицы.