|  |  |
| --- | --- |
| **Gerb-BMSTU_01** | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №6**

**«ОБРАБОТКА ДЕРЕВЬЕВ, ХЕШ-ФУНКЦИЙ»**

Студент Егорова Полина Александровна

Группа ИУ7 – 34Б

Преподаватель Барышникова Марина Юрьевна

2021 г.

# **Цель работы**

Цель работы – построить дерево, вывести его на экран в виде дерева, реализовать основные операции работы с деревом: обход дерева, включение, исключение и поиск узлов, сбалансировать дерево, сравнить эффективность алгоритмов сортировки и поиска в зависимости от высоты деревьев и степени их ветвления; построить хеш-таблицу и вывести ее на экран, устранить коллизии, если они достигли указанного предела, выбрав другую хеш-функцию и реструктуризировав таблицу; сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска (ДДП), в хеш-таблицах и в файлах. Сравнить эффективность реструктуризации таблицы для устранения коллизий и поиска в ней с эффективностью поиска в исходной таблице.

# **Описание условия задачи**

Построить ДДП, сбалансированное двоичное дерево (АВЛ) и хеш-таблицу по указанным данным. Сравнить эффективность поиска в ДДП в АВЛ дереве, в хеш-таблице (используя открытую или закрытую адресацию) и в файле. Вывести на экран деревья и хеш-таблицу. 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.

**Описание технического задания**

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 закрытое хеширование для устранения коллизий. Осуществить добавление введенного целого числа, если его там нет, в ДДП, в сбалансированное дерево, в хеш-таблицу и в файл. Сравнить время добавления, объем памяти и количество сравнений при использовании различных (4-х) структур данных. Если количество сравнений в хеш-таблице больше указанного(вводить), то произвести реструктуризацию таблицы, выбрав другую функцию.

Входные данные:

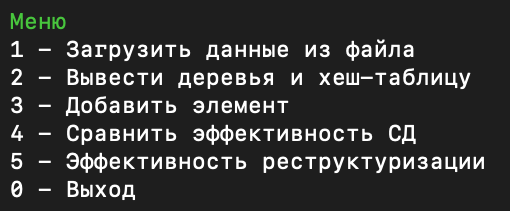
1. Целое число, представляющее собой номер команды:

целое число в диапазоне от 0 до 4.

**Выходные данные:**

1. Результат выполнения команды.
2. Сообщение об ошибке.

Функции программы:



Обращение к программе:

Запуск через терминал (./app.exe)

Аварийные ситуации:

* 1. Некорректный ввод номера команды.

На вход: число, большее чем 5 или меньшее, чем 0.

На выход: *«Неверная команда.»*

# **Описание структуры данных**

**Структуры**

// узел дерева  
typedef struct node  
{  
 int data; // данные, хранящие в узлах  
 int height; // высота дерева в узле  
 struct node\* left; // указатель на левую ветку  
 struct node\* right; // указатель на правую ветку  
} node\_t;

// ячейка хэш-таблицы  
typedef struct  
{  
 int flag; // показатель, что ячейка занята  
 int value; // данные ячейки  
} cell\_t;

// хэш-таблица  
typedef struct   
{  
 cell\_t \*data; // ячейка  
 int limit; // количество коллизий  
 int size; // текущий размер таблицы  
 int max\_size; // максимальный размер таблицы  
} hash\_t;

**Макрозамены**

#define TRUE 1  
#define FALSE 0  
#define LIMIT 15  
#define SIZE 15

**Хэш-функция:**

// остаток от деления числа на размер таблицы  
int hash\_function(int key, int table\_size)  
{  
 key > 0 ? (key) : (key \*= -1);  
 return key % table\_size;  
}

# **Набор тестов**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Название теста** | **Пользователь вводит** | **Вывод** |
| 1 | Некорректный ввод команды | 5  &^% | Неверная команда. |
| 2 | Выбор из пункта 1 | 3 | Неверно выбор файл. |
| 3 | Вызов пункта 2 до вызова 1 |  | Данные для построения дерева не были загружены. Выберите пункт 1. |
| 4 | Вызов пункта 3 до вызова 1 |  | Данные для построения дерева не были загружены. Выберите пункт 1. |
| 5 | Вызов пункта 4 до вызова 1 |  | Данные для построения дерева не были загружены. Выберите пункт 1. |
| 6 | Ввод добавляемого числа (пункт 3) | У  .,:^% | Ошибка ввода. |
| 7 | Ввод количества генерируемых чисел в файле (пункт 1.2) | У  .,:^% | Неверное количество генерируемых чисел. |
| 8 | Ввод добавляемого числа (пункт 4) | У  .,:^% | Ошибка ввода. |
| 9 | Попытка открыть несуществующий файл (пункт 1) |  | Ошибка открытия файла. |

# 

# **Оценка эффективности**

Оценивается время (в тактах), количество сравнений и память (в байтах) при добавлении элемента в различные структуры данных.

**Количество элементов: 10**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Время** | **Сравнения** | **Память** |
| **ДДП** | 3 | 4 | 240 |
| **АВЛ** | 2 | 4 | 240 |
| **Хэш-таблица** | 2 | 1 | 112 |
| **Файл** | 12 | 11 | 35 |

**Количество элементов: 50**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Время** | **Сравнения** | **Память** |
| **ДДП** | 4 | 3 | 1200 |
| **АВЛ** | 3 | 4 | 1200 |
| **Хэш-таблица** | 2 | 1 | 432 |
| **Файл** | 14 | 51 | 175 |

**Количество элементов: 100**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Время** | **Сравнения** | **Память** |
| **ДДП** | 5 | 5 | 2400 |
| **АВЛ** | 5 | 6 | 2400 |
| **Хэш-таблица** | 2 | 1 | 832 |
| **Файл** | 30 | 101 | 360 |

# **Оценка эффективности реструктуризации**

Были созданы три файла, содержащие числа, кратные максимально возможному размеру таблицы (15). Первый файл содержит 12 чисел, второй – 20, третий – 50. Необходимо сравнить эффективность по времени реструктуризации таблицы. Реструктуризация не будет происходить, если максимально возможное количество сравнений будет превышать количество чисел, кратных размеру таблицы (в нашем случае, она может быть >51).

Для каждого из файлов производится замер времени без реструктуризации и с ней - при трех максимально возможных количествах сравнений: 10, 5, 2. Время измеряется в тактах.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Сравнений** | **12 элементов** | | **20 элементов** | | **50 элементов** | |
| **Реструктуризация** | **-** | **+** | **-** | **+** | **-** | **+** |
| 10 | 24 | 40 | 19 | 131 | 57 | 347 |
| 5 | 13 | 58 | 26 | 124 | 59 | 988 |
| 2 | 37 | 113 | 30 | 236 | 63 | 1020 |

# 

# **Контрольные вопросы**

***1.Что такое дерево?***

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

***2.Как выделяется память под представление деревьев?***

Способ выделения памяти под деревья определяется способом их представления в программе. C помощью матрицы или списка может быть реализована таблица связей с предками или связный список сыновей. Целесообразно использовать списки для упрощенной работы с данными, когда элементы требуется добавлять и удалять, т. е. выделять память под каждый элемент отдельно. При реализации матрицей память выделяется статически.

***3.Какие стандартные операции возможны над деревьями?***

Основные операции с деревьями: обход дерева, поиск по дереву, включение в дерево, исключение из дерева. Обход вершин дерева можно осуществить следующим образом:

* сверху вниз (префиксный обход)
* слева направо (инфиксный обход)
* снизу вверх (постфиксный обход)

***4.Что такое дерево двоичного поиска?***

Дерево двоичного поиска – дерево, в котором все левые потомки моложе предка, а все правые – старше. Это свойство называется характеристическим свойством дерева двоичного поиска и выполняется для любого узла, включая корень. С учетом этого свойства поиск узла в двоичном дереве поиска можно осуществить, двигаясь от корня в левое или правое поддерево в зависимости от значения ключа поддерева.

***5.Чем отличается идеально-сбалансированное дерево от АВЛ дерева?***

Узлы при добавлении в идеально сбалансированное дерево располагаются равномерно слева и справа. Получается дерево, у которого число вершин в левом и правом поддеревьях отличается не более, чем на единицу. В то время как АВЛ-дерево – сбалансированное двоичное дерево, у каждого узла которого высота двух поддеревьев отличается не более чем на единицу.

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

Временная сложность поиска элемента в АВЛ дереве – О(log2n)

Временная сложность поиска элемента в дереве двоичного поиска –от О(log2n) до O(n).

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

Массив, заполненный в порядке, определенном хеш-функцией, называется хеш-таблицей. Функция, по которой можно вычислить этот индекс, называется хеш-функцией. Принято считать, что хорошей является такая функция, которая удовлетворяет следующим условиям:

* функция должна быть простой с вычислительной точки зрения;
* функция должна распределять ключи в хеш-таблице наиболее равномерно.
* функция должна минимизировать число коллизий

***8.Что такое коллизии? Каковы методы их устранения.***

Коллизия - ситуация, когда разным ключам соответствует одно значение хеш-функции, то есть, когда h(K1) = h(K2), в то время как K1 ≠ K2. Существует два метода разрешения этой проблемы.

Первый метод – внешнее(открытое) хеширование (метод цепочек). В случае, когда элемент таблицы с индексом, который вернула хеш-функция, уже занят, к нему присоединяется связный список. Таким образом, если для нескольких различных значений ключа возвращается одинаковое значение хеш-функции, то по этому адресу находится указатель на связанный список, который содержит все значения.

Второй метод - внутреннее (закрытое) хеширование (открытая адресация). Оно состоит в том, чтобы полностью отказаться от ссылок. В этом случае, если ячейка с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку (с шагом 1), до тех пор, пока не будет найден ключ K или пустая позиция в таблице.

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

Поиск в хеш-таблицах становится менее эффективен, если наблюдается большое число коллизий. Тогда вместо ожидаемой сложности О(1) получим сложность O(n).

В случае открытого хеширования (цепочки) поиск в списке осуществляется простым перебором, так как при грамотном выборе хеш-функции любой из списков оказывается достаточно коротким. Если же хеширование закрытое, необходимо просматривать все ячейки, если есть много коллизий.

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

Хеш-таблица - от О(1) до O(n)

АВЛ-дерево - О(log2n)

Дерево двоичного поиска – от О(log2n) до O(n).

# **Вывод**

Основным преимуществом деревьев является возможная высокая эффективность реализации основных на ней алгоритмов поиска и сортировки. При удалении или добавлении элемента необходимо корректировать балансировку, тем самым это занимает время.

Хеш-таблицы используют меньше памяти, и для них требуется меньшее количество операций сравнения при добавлении. Так же таблицы требуют качественной хеш-функции, чтобы избежать большого количества коллизий.

Из переведенной выше оценки эффективности можно сделать вывод, что лучше всего и по памяти, и по времени работает хеш-таблица. Это объясняется тем, что для того, чтобы в сбалансированное бинарное дерево добавить элемент, необходимо так же сделать балансировку, что занимает время. Так же, когда мы храним данные в таблице, мы не используем указатели, как в случае с деревьями, поэтому память у хеш-таблицы меньше.

Так же можно заметить, что сбалансированное дерево не всегда выигрывает у несбалансированного. Проигрывает во времени, так как порядок вершин всегда меняется, но выигрывает в сравнении (по среднему количеству сравнений добавления), так как высота сбалансированного дерева будет меньше или такой же, как и у несбалансированного, поэтому чтобы узнать месторасположение элемента, приходится меньше ходить по дереву.