|  |  |
| --- | --- |
| **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 И ХЕШ-ТАБЛИЦ»**

Студент Ву Минь Куанг

Группа ИУ7И – 34Б

Преподаватель Силантьева А. В.

*2021 г.*

**Условие**

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 поиск введенного целого числа в ДДП, в сбалансированном дереве, в хеш-таблице и в файле. Сравнить время поиска, объем памяти и количество сравнений при использовании различных (4-х) структур данных. Если количество сравнений в хеш-таблице больше указанного, то произвести реструктуризацию таблицы, выбрав другую функцию.

**Наименование**

Программа для хранения и поиска чисел и оценки эффективности разных структур данных для хранения и поиска.

**Область применения**

Хранение большого объема данных для быстрого их поиска, организация СУБД.

**Назначение разработки**

Общая концепция системы – консольное приложение.

Описание функционала системы:

Ввод и вывод производится пользователем вручную через консоль с помощью меню.

В меню пользователь может загрузить информацию из заранее подготовленного файла, вывести деревья, добавить и удалить элементы, найти элемент и провести сравнение в различных структурах, подсчитать среднее количество сравнений для поиска, распечатать текущее дерево в формате .png (данная возможность все еще дорабатывается, но программа умеет сохранять файлы в формате .png).

**Требования к программе**

Программа написана полностью на языке Си.

При неправильном вводе пользователем какого-либо значения программа завершает свою работу, выдавая сообщение об ошибке.

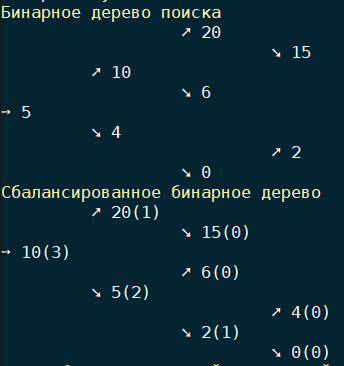
**Входные данные**

Программа строит сбалансированное двоичное дерево, двоичное дерево поиска и хеш-таблицу по информации, полученной из файла (числа разделены в файле пробелом) и добавленной пользователем через консоль (пункт №3 в меню).

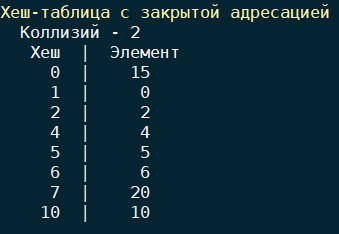
**Выходные данные**

Данная программа выводит на экран:

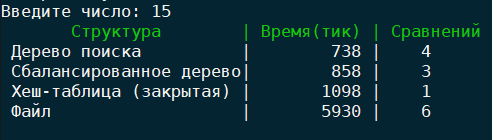
1. Деревья (двоичное сбалансированное дерево и двоичное дерево поиска) в графическом виде. Голова дерева – элемент, перед которым стоит символ «➙». Верхний потомок («➚») больше родителя, нижний потомок («➘») меньше родителя. Для каждой вершины сбалансированного дерева в скобках указана высота данной вершины.



1. Хеш-таблица с закрытой адресацией. При выводе хеш-таблицы также выводится количество коллизий. Так как мной используется хеш-таблица с закрытой индексацией, то во второй колонке находится одно число.



1. Для поиска элемента, введенного пользователем, в различных структурах выводится таблица со временем поиска и количеством сравнений.



**Работа с программой**

Работа с программой осуществляется в консоли с помощью меню:

[1] – загружает информацию из текстового файла. Генерирует дерево двоичного поиска, сбалансированное двоичное дерево и две хеш-таблицы на основе этой информации.

[2] – выводит на экран деревья (двоичного поиска и сбалансированное) и хеш-таблицы (с открытой и закрытой адресацией).

[3] – запрашивает у пользователя число, которое нужно добавить к структурам. Если до выбора этого пункта информация из файла загружена не была, генерирует два новых дерева и две хеш-таблицы с введенным элементом.

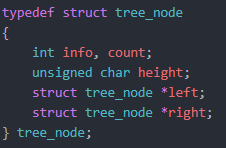
[4] – запрашивает у пользователя число, которое нужно удалить из структур.

[5] – запрашивает у пользователя число, которое нужно найти в структурах. При его наличии в структурах на экран выводится время, за которое был совершен поиск и количество сравнений.

[0] – выход из программы.

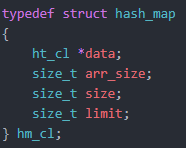
**Структуры данных и функции**

Узел дерева представляет собой структуру:



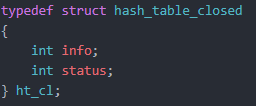
info – число, count – количество встреч, height – высота узла, left – левый потомок, right – правый потомок.

Оболочка хеш-таблицы с закрытой адресацией представляет собой структуру:



data – хеш-таблица, arr\_size – количество элементов, size – размер таблицы, limit – количество коллизий.

Элементы хеш-таблицы представляет собой структуру:



info - число, status – статус.

В программе есть следующие функции:

**Функции необходимые для работы с хеш-таблицей:**

**unsigned int hash(int info, int key);**

**hm\_cl \*new\_hm\_cl(size\_t size);**

**ht\_cl \*new\_ht\_cl(size\_t size);**

**hm\_cl \*resize\_ht\_cl(hm\_cl \*table);**

**hm\_cl \*add\_ht\_cl(hm\_cl \*table, int info);**

**int search\_ht\_cl(hm\_cl \*table, int info);**

**hm\_cl \*delete\_ht\_cl(hm\_cl \*table, int info);**

**void print\_ht\_cl(const hm\_cl \*table);**

**hm\_cl \*load\_ht\_cl\_from\_file(FILE \*file);**

**void free\_ht\_cl(hm\_cl \*table);**

**hm\_cl \*restruct\_ht\_cl(hm\_cl \*table);**

**Функции необходимые для работы с деревьями:**

**void print\_tree(tree\_node \*head, int down, int lr, int flag);**

**unsigned long long int tick();**

**tree\_node \*create\_node(int info);**

**tree\_node \*find\_node(tree\_node \*tree, int info);**

**void zero\_count\_made(void);**

**void show\_count(void);**

**int get\_count\_made(void);**

**void apply\_in(tree\_node \*tree, void (\*f)(tree\_node\*, void\*), void \*arg);**

**void apply\_post(tree\_node \*tree, void (\*f)(tree\_node\*, void\*), void \*arg);**

**void print\_node(tree\_node \*tree, void \*arg);**

**void apply\_pre(tree\_node \*tree, void (\*f)(tree\_node\*, void\*), void \*arg);**

**void free\_node(tree\_node \*tree, void \*param);**

**void free\_tree(tree\_node \*tree);**

**tree\_node \*insert(tree\_node \*tree, tree\_node \*node);**

**tree\_node \*binary\_tree\_insert(tree\_node \*tree, tree\_node \*node, int \*count);**

**tree\_node \*read\_tree\_from\_file(FILE \*file);**

**tree\_node \*read\_binary\_tree\_from\_file(FILE \*file, int \*count);**

**tree\_node \*find\_min(tree\_node \*p);**

**tree\_node \*remove\_min(tree\_node \*p);**

**tree\_node \*remove\_tree(tree\_node \*p, int k);**

**tree\_node \*binary\_tree\_remove(tree\_node \*p, int k);**

**void level\_apply(int \*levels, tree\_node \*tree, int index);**

**int \*count\_levels(tree\_node \*tree, int \*n);**

**void print\_array(int \*arr, int n);**

**unsigned char height(tree\_node \*p);**

**int bfactor(tree\_node \*p);**

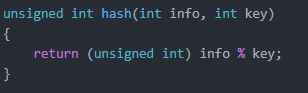
**void fix\_height(tree\_node \*p);**

**tree\_node \*rotate\_right(tree\_node \*p);**

**tree\_node \*rotate\_left(tree\_node \*q);**

**tree\_node \*balance(tree\_node \*p);**

**Хеш-функция:**



info – число, key – размер таблицы.

**Алгоритм**

Ввод: choice

Вывод: -

Считать choice

Пока choice != 0

Если choice = 0

Выход из программы

Если choice = 1

Чтение данных из файла

Если choice = 2

Вывод деревьев и таблицы на экран

Если choice = 3

Добавление элемента в деревья и в хеш-таблицу

Если choice = 4

Удаление элемента из деревьев и из хеш-таблицы

Если choice = 5

Поиск элемента в различных структурах и их сравнение

Все если

**Все пока**

**Модули и тесты**

**Функциональные тесты**

|  |  |  |
| --- | --- | --- |
| **Входные данные** | **Вывод** | **Класс эквивалентности** |
| Меню: y | Сообщение об ошибке, завершение программы | Некорректный ввод в меню |
| Меню: 3 | Успешный ввод | Корректный ввод в меню |
| Ввод значения: 15 | Успешный ввод | Корректный ввод значения |
| Ввод значения: F | Сообщение об ошибке, завершение программы | Некорректный ввод значения |
| Поиск: ввод существующего элемента | Вывод информации о том, как искался элемент (количество сравнений, время) | Существующий элемент |
| Поиск: ввод несуществующего элемента | Сообщение о том, что такой элемент не найден | Несуществующий элемент |

**Выводы**

В ходе данной работы я научился работать с такими структурами данных, как дерево двоичного поиска, сбалансированное дерево двоичного поиска и хеш-таблица, а также осуществлять балансировку дерева, построение и реструктуризацию хеш-таблицы, поиск элемента в этих структурах. Также я сравнил эффективность поиска по этим структурам и по файлу.

Результаты сравнения эффективности:

Для наглядности будем использовать файл из 40 числа

1. Слово для поиска количество коллизий = 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Данные | ДДП | АВЛ | ХЕШ | ФАЙЛ |
| Среднее время поиска каждого | 1497 | 1847 | 1469 | 5053 |
| Среднее количество сравнений | 4.24 | 3.36 | 1.54 | 1.00 |

1. для поиска находится в середине файла, количество коллизий = 1:

Занимаемая память в байтах (количество коллизий в таблице = 1):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | ДДП | АВЛ | ХЕШ | ФАЙЛ |
| 10 | 240 | 320 | 376 | 48 |
| 20 | 480 | 640 | 2104 | 97 |
| 30 | 744 | 992 | 2104 | 142 |
| 40 | 936 | 1248 | 3864 | 190 |
| 50 | 1152 | 1536 | 5976 | 242 |

Наиболее эффективна хеш-таблица (наименьшее количество сравнений, зависит от выбранной хеш-функции). Сбалансированное дерево эффективнее несбалансированого.

Мы можем увидеть из результатов тестирования, что использовать двоичное дерево поиска для поиска данных намного эффективнее, чем использовать файл. Также эффективность еще повышается, если дерево сбалансировать. Наиболее эффективно осуществлять поиск в хеш-таблице, так как она требует минимальное число сравнений, но нужно знать хеш элемента. Хеш-таблица перестает быть эффективной при большом количестве коллизий, поэтому ее приходится реструктурировать. По память оба вида ДДП и хеш-таблица занимают примерно одинаковый объем память, в то время как файл занимает намного меньше памяти. Из всего вышеперечисленного следует, что для хранения данных удобнее использовать файл, в то время как для поиска наиболее эффективна хеш-таблица в случае, если есть возможность быстро вычислить хеш, иначе эффективнее дерево двоичного поиска.

# **Ответы на контрольные вопросы**

**1. Что такое дерево?**

Дерево – рекурсивная структура данных, используемая для представления иерархических связей, имеющих отношение “один ко многим”.

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

В виде связанного списка. Каждый лепесток – узел.

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

Обход дерева, поиск по дереву, включение элемента в дерево, исключение элемента из дерева.

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

Двоичное дерево поиска – двоичное дерево, для каждого узла которого выполняется условие: левый потомок больше или равен родителю, правый потомок строго меньеш родителя (также можно наоборот).

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

В АВЛ дереве для каждой его вершины высота её 2-х поддеревьев отличается не более, чем на 1. В идеально сбалансированном дереве количество вершин в каждом поддереве различается не более, чем на 1.

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

В АВЛ дереве поиск происходит быстрее, чем в дереве двоичного поиска.

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

Хеш-таблица – массив, заполненный элементами в порядке, который опредляется хеш-функцией

Хеш-функция ставит каждому элементу таблицы в соответствие определенный индекс. Функция должна быть простой для вычисления, распределять ключи в таблице равномерно и давать в результате минимум коллизий.

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

Коллизия – ситуация, при которой разным ключам хем-таблицы ставится в соответствие один и тот же индекс.

Основные методы устранения коллизий: открытое хеширование, закрытое хеширование.

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

При большом количестве коллизий поиск становится менее эффективным, сложность возрастает относительно О(1). В таком случае требуется реструктуризация таблицы, то есть перезаполнение таблицы с использованием новой хеш-функции.

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

Минимальное время поиска для хеш-таблицы: О(1) Минимальное время поиска для АВЛ дерева: О(log2n)

Минимальное время поиска а Двоичном дереве поиска О(h), где h – высота дерева (от log2n до n)