**Лабораторная работа №6**

**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.

**Сроки исполнения**

2 недели, начиная с 26.11.2018

**Основание для разработки**

Заказчик – Силантьева Александра Васильевна

Исполнитель – Ильясов Идрис Магомет-Салиевич, группа ИУ7-33Б

Основание для разработки – лабораторная работа №6

**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.

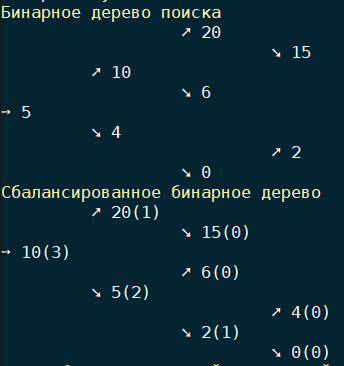
**Входные данные**

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).

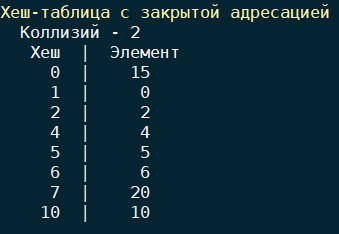
**Выходные данные**

Данная программа выводит на экран:

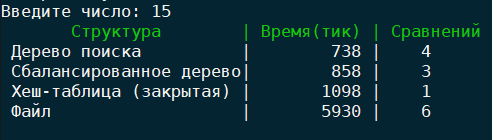
* Деревья (двоичное сбалансированное дерево и двоичное дерево поиска) в графическом виде. Голова дерева – элемент, перед которым стоит символ «➙». Верхний потомок («➚») больше родителя, нижний потомок («➘») меньше родителя. Для каждой вершины сбалансированного дерева в скобках указана высота данной вершины.



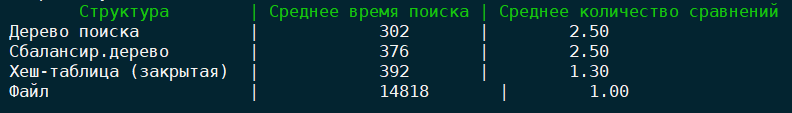
* Хеш-таблица с закрытой адресацией. При выводе хеш-таблицы также выводится количество коллизий. Так как мной используется хеш-таблица с закрытой индексацией, то во второй колонке находится одно число.



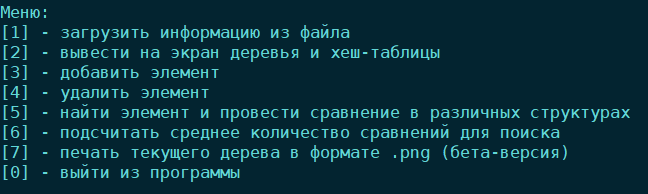
* Для поиска элемента, введенного пользователем, в различных структурах выводится таблица со временем поиска и количеством сравнений.



* Таблица, содержащая среднее время поиска элементов и среднее количество 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.

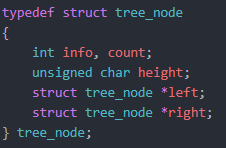
[6] – ищет в дереве N элементов (т. е. процедура поиска N раз). Выводит на экран количество затраченного времени / N (среднее время поиска) и количество сравнений / N (среднее количество сравнений) для каждой структуры данных.

[7] – выводит текущее дерево в виде картинки в файл .png.

[0] – выход из программы.

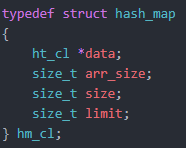
**Структуры данных и функции**

Узел дерева представляет собой структуру:



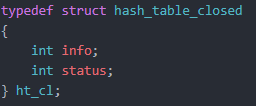
info – число, count – количество встреч, height – высота узла, left – левый потомок, right – правый потомок.

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);**

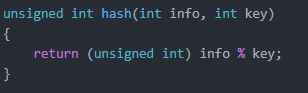
**void fix\_height(tree\_node \*p);**

**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.

**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 |

**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

Если choice = 6

Вывод дерева в файл (сохранение в виде картинки)

Все если

Все пока

**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:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | Среднее время, мкс  Среднее кол-во сравнений | | | |
| ДДП | СДДП | Хеш-таблица | Файл |
| 100 | 36  6 | 31  4 | 18  1 | 1340  50 |
| 1000 | 632  9 | 463  6 | 135  1 | 72231  500 |
| 10000 | 9231  17 | 7694  11 | 3410  1 | 5341623  5000 |

Занимаемая память:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов | ДДП | СДДП | Хеш-таблица | Файл |
| 100 | 5200 | 5200 | 5326 | 1100 |
| 1000 | 52000 | 52000 | 53260 | 11000 |
| 10000 | 520000 | 520000 | 532608 | 110000 |

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.

**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.

1. *Как выделяется память под представление деревьев?*

Способ выделения памяти под деревья определяется способом их представления в программе. Например, с 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.

1. *Какие стандартные операции возможны над деревьями?*

Основные операции над деревьями - обход дерева (инфиксный, префиксный, постфиксный), поиск элемента в дереве, добавление и удаление элемента в дерево.

1. *Что такое дерево двоичного поиска?*

Дерево двоичного поиска - дерево, в котором все левые потомки “моложе” предка, а все правые потомки - “старше” предка. Это свойство выполняется для любого узла, включая корень.

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

Если при добавлении узлов в дерево мы будем их равномерно располагать 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. Адельсон-Вельский и Ландис сформулировали менее жесткий критерий сбалансированности таким образом: двоичное дерево называется сбалансированным, если у каждого узла дерева высота двух поддеревьев отличается не более чем на единицу. Такое дерево называется АВЛ-деревом.

1. *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).

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

Массив, заполненный в порядке, определенным хеш-функцией, называется хеш-таблицей. Функцию, 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.

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

Может возникнуть ситуация, когда разным ключам соответствует одно значение хеш-функции, то есть, когда h(K1) = h(K2), в то время как K1 ≠ K2. Такая ситуация называется коллизией. 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. Поиск в этом списке осуществляется простым перебором, так как при грамотном выборе хеш-функции любой из списков оказывается достаточно коротким. Другой путь решения проблемы, связанной с коллизиями – 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. При этом, если индекс следующего просматриваемого элемента определяется добавлением какого-то постоянного шага (от 1 до n), то данный способ разрешения коллизий называется линейной адресацией.

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

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

1. *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).