|  |  |
| --- | --- |
| **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)** |

**Лабораторная работа №6**

**По дисциплине «Типы и структуры данных»  
  
Тема: «Обработка деревьев и хеш-таблиц»**

**Вариант 3**

* + 1. Выполнил: студент группы ИУ7-31Б  
       Баринов Никита

Проверил:

Цель работы

Построить дерево в соответствии с заданным вариантом задания. Вывести его на экран в виде дерева. Реализовать основные операции работы с деревом: обход дерева, включение, исключение и поиск узлов. Сравнить эффективность алгоритмов сортировки и поиска в зависимости от высоты дерева и степени его ветвления. Построить хеш-таблицу по указанным данным. Вывести на экран деревья и хеш-таблицу. Сравнить эффективность поиска в двоичном дереве поиска, в сбалансированном дереве поиска и в хеш-таблице. Вывести на экран измененные структуры. 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.

Задание

Построить ДДП, в вершинах которого находятся слова из текстового файла. Вывести его на экран в виде дерева. Сбалансировать полученное дерево и вывести его на экран. Удалить указанное слово в исходном и сбалансированном дереве. Сравнить время удаления и объем памяти. Построить хеш-таблицу из слов текстового файла, задав размерность таблицы с экрана, используя метод цепочек для устранения коллизий. Вывести построенную таблицу слов на экран. Осуществить удаление введенного слова, вывести таблицу. Сравнить время удаления, объем памяти и количество сравнений при использовании ДДП, сбалансированных деревьев, хеш-таблиц и файла.

Описание программы

Программа позволяет работать по отдельности с ДДП, АВЛ деревом и хеш-таблицей. Одновременно добавлять и искать во всех структурах и сравнивать их производительность.

Входные данные

На вход выбранный пункт меню или слово для поиска или вставки и файл с данными.

Выходные данные

Программа выводит деревья, хеш-таблицу, результаты поиска и вставки и замеры времени и памяти при поиске и вставке.

Способ передачи входных данных и получения выходных данных

Программа получает входные данные через файл, название файла указывается как аргумент командной строки. Через консоль осуществляется управление программой и ввод слов для поиска и вставки.

Возможные аварийные ситуации

1. Некорректный ввод
2. Ошибка памяти
3. Пустой входной файл
4. Отсутствие входного файла

Данные

ДДП:

typedef struct tree\_node tree\_node\_t;

struct tree\_node

{

    char \*word;

    tree\_node\_t \*left;

    tree\_node\_t \*right;

    tree\_node\_t \*parent;

};

АВЛ дерево:

typedef struct avl\_tree\_node avl\_tree\_node\_t;

struct avl\_tree\_node

{

    char \*word;

    int height;

    avl\_tree\_node\_t \*left;

    avl\_tree\_node\_t \*right;

    avl\_tree\_node\_t \*parent;

};

Хеш-таблица:

typedef struct

{

    List \*\*bucket;

    int size;

}hash\_table\_t;

Список:

typedef struct List List;

struct List

{

    char \*word;

    List \*next;

};

Описание функций

ДДП:

*// Создание узла дерева*

tree\_node\_t \* create\_tree\_node(char \**word*);

*// Удаление узла дерева*

void destroy\_tree\_node(tree\_node\_t \**node*, void *delete\_content*(void \*));

*// Удаление всего дерева*

void destroy\_tree(tree\_node\_t \**tree*, void (\**delete\_content*)(void \*));

*// Вставка узла в дерево*

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

*// Поиск слова в дерево*

tree\_node\_t\* lookup\_tree(tree\_node\_t \**tree*, char \**word*, int \**n*);

*// удаление слова из дерева*

tree\_node\_t \*delete\_tree\_node(tree\_node\_t \**tree*, char \**word*, int \**n*);

*// Функции для вывода дерева на экран*

void bst\_print\_dot\_null(char \* *word*, int *nullcount*, FILE\* *stream*);

void bst\_print\_dot\_aux(tree\_node\_t\* *node*, FILE\* *stream*);

void bst\_print\_dot(tree\_node\_t\* *tree*, FILE\* *stream*);

АВЛ дерево:

*// Создание узла дерева*

avl\_tree\_node\_t \*create\_avl\_tree\_node(char \**word*);

*// Удаление узла дерева*

void destroy\_avl\_tree\_node(avl\_tree\_node\_t \**node*, void (\**delete\_content*)(void \*));

*// Удаление всего дерева*

void destroy\_avl\_tree(avl\_tree\_node\_t \**tree*, void (\**delete\_content*)(void \*));

*// Получение высоты узла*

int height(avl\_tree\_node\_t \**node*);

*// Расчет высоты узла*

void fix\_height(avl\_tree\_node\_t \**node*);

*// Вставка узла в дерево*

avl\_tree\_node\_t \*insert\_avl\_tree(avl\_tree\_node\_t \**tree*, avl\_tree\_node\_t \**node*);

*// Поиск слова в дереве*

avl\_tree\_node\_t \*lookup\_avl\_tree(avl\_tree\_node\_t \**tree*, char \**word*, int \**n*);

*// Обход дерева(обратный)*

void apply(avl\_tree\_node\_t \**tree*, void (\**f*)(avl\_tree\_node\_t \**node*));

*// Удаление слова из дерева*

avl\_tree\_node\_t \*delete\_avl\_tree\_node(avl\_tree\_node\_t \**tree*, char \**word*, int \**n*);

*// Нахождение разницы в высотах*

int bfactor(avl\_tree\_node\_t \**node*);

*// Правый поворот*

avl\_tree\_node\_t \*rotate\_right(avl\_tree\_node\_t \**node*);

*// Левый поворот*

avl\_tree\_node\_t \*rotateleft(avl\_tree\_node\_t \**node*);

*// Балансировка*

avl\_tree\_node\_t \*balance(avl\_tree\_node\_t \**node*);

*// Функции дл явывода дерева на экран*

void avl\_bst\_print\_dot\_null(char \* *word*, int *nullcount*, FILE\* *stream*);

void avl\_bst\_print\_dot\_aux(avl\_tree\_node\_t\* *node*, FILE\* *stream*);

void avl\_bst\_print\_dot(avl\_tree\_node\_t\* *tree*, FILE\* *stream*);

Хеш-таблица:

*// Хеш-функция*

int hash\_func(int *curr\_size*, const char \**value*);

*// Добавление слова в ХТ*

void add\_ht(hash\_table\_t \**table*, char \**word*);

*// Инициализация ХТ*

hash\_table\_t \*init\_ht(int *size*);

*// Поиск слова в ХТ*

List \*find\_ht(hash\_table\_t \**ht*, char \**word*, int \**n*);

*// Удаление слова из ХТ*

void delete\_from\_ht(hash\_table\_t \**ht*, char \**word*, int \**n*);

*// Очистка ХТ*

void destroy\_ht(hash\_table\_t \**ht*);

*// Вывод ХТ*

void print\_ht(hash\_table\_t \**ht*);

*// Реструктуризация ХТ*

void restruct(hash\_table\_t \**dst*, hash\_table\_t \**src*);

**Алгоритм**

Меню:

Выберите пункт меню: 0 - выход из программы 1 - работа с бинарными деревьями 2 - работа с хеш-таблицей 3 - оценка эффективности

Меню деревьев:

Выберите пункт меню: 0 - возврат в главное меню 1 - прочитать файл в дерево 2 - прочитать файл в avl-дерево 3 - добавить слово в дерево 4 - добавить слово в avl-дерево 5 - удалить слово из дерева + вывод кол-ва сравнений 6 - удалить слово из avl-дерева + вывод кол-ва сравнений 7 - вывести на экран дерево 8 - вывести на экран avl-дерево 9 - очистить дерево 10 - очистить avl-дерево

Меню ХТ:

Выберите пункт меню: 0 - возврат в главное меню 1 - прочитать файл в таблицу 2 - задать размер таблицы 3 - добавить слово в таблицу 4 - удалить слово из таблицы + вывод кол-ва сравнений 5 - вывести на экран таблицу 6 - очистить avl-дерево

Вставка в ДДП:

Сравниваем значение со значением текущего узла, если оно меньше, то переходи к левому потомку иначе к правому. Если при переходе к потомку обнаружилось, что его нет, то нашлось место для нового элемента. Выделяем память для узла и присваиваем нужное значение.

Вставка в АВЛ дерево:

Алгоритм вставки в АВЛ дерево идентичен вставке в ДДП, то после самой вставке происходит балансировка всех деревьев, корнями которых являются вершины, пройдённый во время поиска места для нового элемента, если она нужна.

Перед балансировкой сначала пересчитываются высоты поддеревьев. Если одна длиннее боле чем на 1, происходит балансировка.

Если правое дерево перевешивает на 2, то происходит левый поворот. Если при этом в правом поддереве текущего корня перевешивает левое поддерево, то предварительно происходит правый поворот в правом поддереве текущего корня. Если перевешивает левое поддерево на 2, алгоритм балансировки симметричен. Поворот, состоящий из 2 поворотов, называют большим.

Вставка в хеш-таблицу:

С помощью хеш-функции получаем индекс массива по значению ключа. При вставке в ячейку массива новый элемент добавляется в конец списка, указатель на который хранится в ячейке массива. При вводе нового размера таблицы все значения заново хешируются и таблица перестраивается.

Тесты

**Негативы**

Несуществующий файл

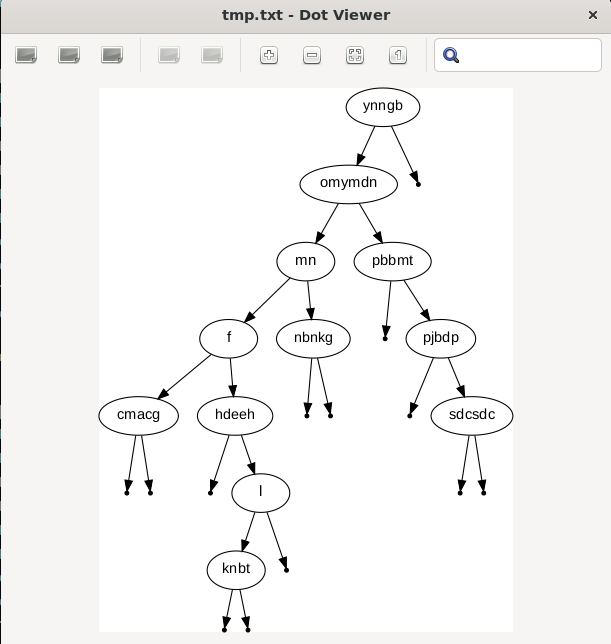
Пустой файл

Некорректный размер таблицы

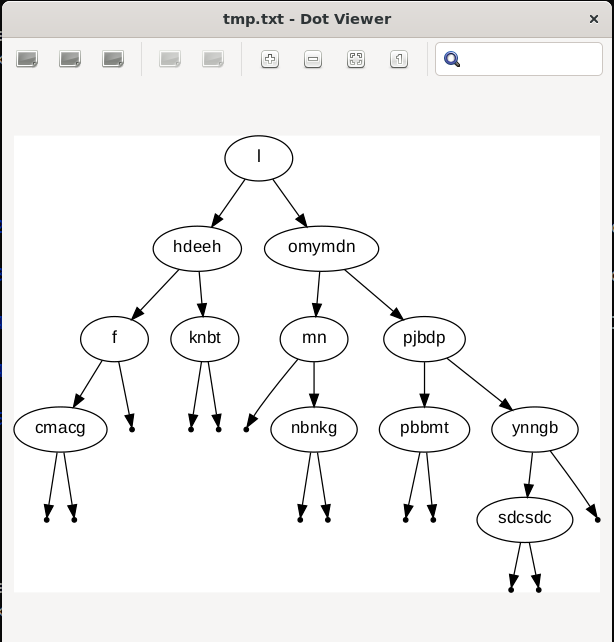
Вывод пустого дерева

**Позитивы**

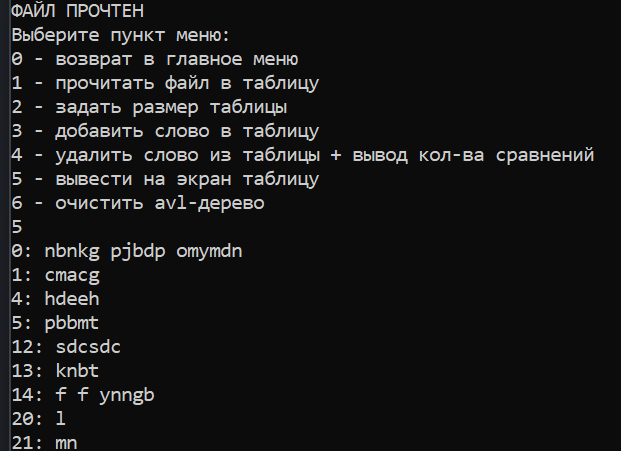
Вывод ДДП



Вывод АВЛ дерева



Вывод хеш-таблицы



Сравнение эффективности

Кол-во элементов: 400

Размер таблицы: 400 \* 1.25

Удаление

Время среднее, за 50 замеров

|  |  |  |
| --- | --- | --- |
| Структура | Время(мс) | Кол-во сравнений |
| ДДП | 1,11 | 9,7 |
| АВЛ-дерево | 3,59 | 18,3 |
| Хеш-таблица | 0,2 | 1,2 |
| Файл | 11,2 | 400 |

Память

|  |  |
| --- | --- |
| Структура | Память |
| ДДП | 12800 |
| АВЛ-дерево | 16000 |
| Хеш-таблица | 6400 |
| Файл | 4096 |

При удалении хеш-таблица быстрее ДДП примерно в 4,5 раза и АВЛ дерева примерно в 9 раз. ДДП быстрее АВД дерева в 2 раза. Файл сильно медленнее всех.

Файл занимает меньше всех памяти. ДДП занимает на 100% процентов больше памяти чем хеш-таблица. АВЛ дерево занимает на 25% больше памяти чем ДДП.

Вывод

В ходе лабораторной работы я написал ДДП, АВД дерево и хеш-таблицу и измерил их эффективность. Можно сделать вывод, что в быстрее всего работает хеш-таблица, но она обладает большим недостатком. В хеш-таблицах возникают коллизии, которые замедляют работу с таблицей. Для уменьшения коллизий нужно иметь хеш-функцию с хорошим распределением и реструктуризировать таблицу. Реструктуризация требует много ресурсов. Этот недостаток может сильно повлиять на быстродействие программы. Деревья работают медленнее, но они лишены такого недостатка. АВЛ деревья будет быстрее ДДП в задачах с частым поиском символов, наоборот, когда чаще поиска происходит вставка. По память самое эффективное решение это хеш-таблица. Файл лучше использовать, когда есть ограничения по память и не важно быстродействие. Выбор структуры данных сильно зависит от решаемой задачи.

Также, в моей реализации при удалении АВЛ-дерево работает дольше ДДП, связано это с тем, что при удалении происходит балансировка, которая и увеличивает время.

Контрольные вопросы

1. Что такое дерево?

Дерево – нелинейная структура данных с иерархическими связями, имеющих отношения один у многим.

2. Как выделяется память под представление деревьев?

В виде массива с номером родителя, в виде связного списка сыновей, в виды структуры данных с указателями на сыновей.

3. Какие стандартные операции возможны над деревьями?

Поиск, вставка, удаление, обход.

4. Что такое дерево двоичного поиска?

Двоичное дерево упорядоченно так, что все элементы левого поддерева меньше корня, а элементы правого поддерева больше. Для каждого узла тоже самое. Двоичное дерево – дерево, элемент которого имеет не более 2 сыновей.

5. Чем отличается идеально сбалансированное дерево от АВЛ дерева?

В идеально сбалансированном дереве кол-во элементов в правом и левом поддереве отличается не более чем на единицу. В АВЛ дереве высоты правого и левого поддерева отличается не более чем на единицу

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

Алгоритм одинаков.

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

Структура данных позволяющая получать по ключу элемент массива называется хеш-таблицей.

Для доступа по ключу используется хеш-функция. Она по ключу получает нужный индекс массива. Хеш-функция должна возвращать одинаковые значения для одного ключа и использовать все индексы с одинаковой вероятностью.

8. Что такое коллизии? Каковы методы их устранения.

Ситуация, когда из разных ключей хеш-функция выдаёт одни и тот же индекс, называется коллизией.

Метод цепочек – При коллизии элемент добавляется список элементов этого индекса.

Линейная адресация – при коллизии ищется следующая незаполненная ячейка.

Произвольная адресация - используется заранее сгенерированный список случайных чисел для получения последовательности.

Двойное хеширование – использовать разность 2 разных хеш-функций.

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

При большом количестве коллизий.

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

Скорость поиска в хеш-таблице зависит от числа коллизий. При небольшом числе коллизий для поиска элемента совершается мало сравнений и поиск быстрее чем в деревьях.

АВЛ дерево быстрее при поиске за счёт более равномерного распределения элементов чем в ДДП.