|  |  |
| --- | --- |
|  | **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

**ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №6**

Название предмета: Типы и структуры данных

Студент: Гриценко Алексей Михайлович

Группа: ИУ7-31Б

*2020г.*

1. **Описание условия задачи**

Построить дерево двоичного поиска, в вершинах которого находятся слова из текстового файла. Вывести его на экран в виде дерева. Сбалансировать полученное дерево и вывести его на экран. Построить хеш-таблицу из слов текстового файла. Использовать метод цепочек для устранения коллизий. Осуществить поиск введенного слова 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 (вводить), то произвести реструктуризацию таблицы, выбрав другую функцию.

1. **Техническое задание**
2. ***Описание исходных данных***
3. **Имя файла с деревом:** строка, содержащая имя файла.
4. **Максимальное значение допустимых коллизий:** целое число.
5. **Слово для поиска:** строка, содержащая слово, которое планируется найти в дереве
6. ***Описание результата программы***
7. Псевдографическое изображение бинарного дерева.
8. Псевдографическое изображение сбалансированного бинарного дерева.
9. Хеш-таблица, построенная по словам из файла.
10. Количественная характеристика выполнения моделирования (время поиска вершины в различных представлениях дерева, количество занимаемой деревом памяти в различных представлениях, количество сравнений для достижения результата (поиска) в различных представлениях, среднее количество сравнений (при поиске) в различных представлениях).

*.*

1. ***Задача, реализуемая программой***

Программа выполняет ряд функций в порядке:

1. Вывод бинарного дерева.
2. Вывод сбалансированного бинарного дерева.
3. Вывод хеш-таблицы.
4. Вывод количественной характеристики моделирования, указанного в секции выходных данных.
5. ***Обращения к программе***

Способ обращения к программе происходит через консоль.

1. ***Описание возможных аварийных ситуаций и ошибок пользователя***
2. Некорректный ввод имени файла.  
   На входе: имя файла, несуществующего в системе.   
   На выходе: сообщение «Неверное имя файла! Повторите попытку.»
3. Пустой файл в качестве аргумента программы.  
   На входе: пустой файл.  
   На выходе: сообщение «Файл пуст! Проверьте файл.»
4. Некорректный ввод максимального числа коллизий.  
   На входе: буква или, любой другой нечисловой символ или отрицательное число.  
   На выходе: сообщение «Введено недопустимое значение! Повторите попытку.»
5. **Описание внутренних структур данных**

Реализация линейного односвязного списка:

**typedef struct lst**

**{**

***char* \*value;**

***struct lst* \*next;**

**} lst\_t;**

Поля структуры:

* ***char* \*value** **–** указатель на массив символов;
* ***struct lst* \*next –** указатели на следующий элемент списка.

Реализация листа дерева:

**typedef struct tree\_node**

**{**

***char* \*value;**

***struct tree\_node* \*left;**

***struct tree\_node* \*right;**

**} tree\_node;**

Поля структуры:

* ***char* \*value *–*** указатель на массив символов;
* ***struct tree\_node* \*left *–***указатель на левого потомка;
* ***struct tree\_node* \*right *–***указатель на правого потомка.

Реализация динамического массива (для реализации хеш-таблицы):

**typedef struct**

**{**

***tree\_node* \*\*arr;**

***int* size;**

***int* mem\_size;**

**} dynarr\_t;**

Поля структуры:

* ***tree\_node* \*\*arr *–*** указатель на массив указателей листов дерева;
* ***int* size *–***фактическая ёмкость массива;
* ***int* mem\_size –** аллоцированная ёмкость массива.

В лучшем случае, для реализации сбалансированного дерева, нужно ввести такую структуру:

typedef struct BlncTNode {

word\_t \*data;

struct BlncTNode \*left;

struct BlncTNode \*right;

int height;

} balanced\_tree\_node\_t;

typedef struct

{

char value[MAXLEN];

} elem\_t;

word\_t – указатель на данные, хранимые в узле дерева

struct BlncTNode \*left – указатель на левый потомок

struct BlncTNode \*right – указатель на левый потомок

int height – высота узла (листья имеют высоту 0, их непосредственные родители – 1 и т.д.).

При этом слева находится элемент, меньший данного, а справа – больший. Кроме того, выполняется условие балансировки: для каждой вершины (узла) высота её двух поддеревьев различается не более чем на 1.

Данная структура лучше из-за главной особенности:

* за счёт балансировки дерево не вырождается в односвязный список в

случае отсортированных данных, его высота пропорциональна

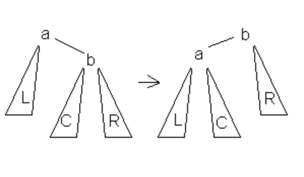
логарифму

1. **Описание алгоритма**
2. Пользователь вводит имя файла.
3. Выводится дерево из слов, сбалансированное дерево (исходное дерево вытягивается в отсортированный односвязный список, затем происходил рекурсивное построение АВЛ-дерева(методом деления полученного списка на половину)), и хеш-таблица, вместе с характеристиками (время, память, количество коллизий для хеш-таблиц).
4. Пользователь вводит максимальное количество коллизий, и таблица либо перестраивается, если количество меньше исходного, или нет, если количество больше и выводится характеристика моделирования
5. Происходит поиск слова в структурах данных, и выводится либо количественная характеристика поиска, либо сообщение, что слово не найдено.

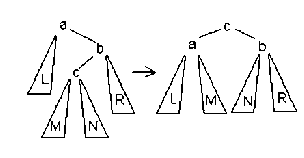
Алгоритм балансировки дерева **как надо**

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины. Используются 4 типа вращений:

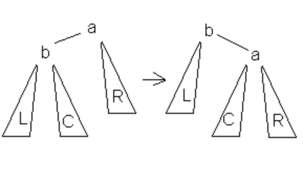
Малое левое вращения - Данное вращение используется тогда, когда (высота b-поддерева — высота L) = 2 и высота c-поддерева <= высота R.



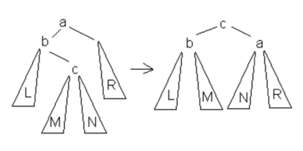
Большое левое вращение - Данное вращение используется тогда, когда (высота b-поддерева — высота L) = 2 и высота c-поддерева > высота R.



Малое правое вращение - Данное вращение используется тогда, когда (высота b-поддерева — высота R) = 2 и высота С <= высота L.



Большое правое вращение - Данное вращение используется тогда, когда (высота b-поддерева — высота R) = 2 и высота c-поддерева > высота L.



Балансировка реализуется с помощью переприсвоения указателей.

Алгоритм вставки **как надо**

Рекурсивный алгоритм:

1. Если на текущем шаге указатель на узел дерева пуст, то мы нашли место вставки, а значит можем создать новый узел и присвоить ему значение данных, переданных для вставки, а затем вернуть этот узел.

2. Если указатель на узел дерева не пуст, то сравниваем значение в этом узле с переданными данными: если значение узла больше, то продолжим поиск места вставки в левом поддереве, иначе в правом (случай равенства зависит от конкретной реализации, например, можно не включать узел, если такой уже есть).

3. Возврат текущего узла дерева (с изменённым левым или правым потомком). **При реализации сбалансированного дерева перед возвратом необходимо применить функцию балансировки к данному узлу**.

**Функции, использующиеся в программе**

int build\_hash\_table(lst\_t \*\*arr, int size, FILE \*f, int (\*hash)(char \*, int)) - функция создания хеш-таблицы

int file\_len(FILE \*f) – функция вычисления длины файла

void print\_hash\_table(lst\_t \*arr, int size) – функция печати хеш-таблицы

char \*\*fill\_words(FILE \*f) – функция чтения слов, содержащихся в файле

int search\_tree(char \*to\_find, tree\_node root) – функция поиска слова в дереве

int search\_hashtable(char \*to\_find, lst\_t \*hash\_lst, int n, int (\*hash)(char \*, int)) – функция поиска в хеш-таблице

int search\_file(char \*to\_find, FILE \*f) – функция поиска слова в файле

tree\_node \*balance\_tree(tree\_node \*root, int \*unique) – функция балансировки дерева

tree\_node \*insert(tree\_node \*node, char \*val) – функция вставки элемента в дерево

void print\_tree(tree\_node \*root, int space) – функция печати дерева

void free\_tree(tree\_node \*node) – функция чистки памяти для дерева

void free\_words(char \*\*words, unsigned long flen) – функция чистки памяти массива слов

int linsert(lst\_t \*arr, char \*buffer) – функция вставки элемента в хеш-таблицу

1. **Тестирование**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Тест | Пользовательский ввод | Результат |
| 1 | Некорректный ввод имени файла | tree07.txt (файл не существует) | Неверное имя файла! Повторите попытку. |
| 2 | Пустой файл | tree06.txt (пустой файл) | Файл пуст! Проверьте файл. |
| 3 | Некорректный ввод максимального количества коллизий | A | Введено недопустимое значение! Повторите поппытку |
| 4 | Некорректный ввод максимального количества коллизий | 0 | Введено недопустимое значение! Повторите поппытку |
| 5 | Некорректный ввод максимального количества коллизий | -1 | Введено недопустимое значение! Повторите поппытку |
| 6 | Ввод несуществующего слова | sdfg (слова нет) | Слово “sdfg” не найдено. |
| 7 | Ввод количества коллизий, большего, чем текущее максимальное | 3 (при максимальном 2) | Результат достижим за введенное количество коллизий.  Пересоздание таблицы не требуется. |
| 8 | Ввод количества коллизий, меньшего, чем текущее максимальное | 1 (при максимальном 2) | Пересоздание хеш-таблицы |
| 9 | Корректный ввод всех характеристик | Корректный файл, корректный ввод числа коллизий | Количественная характеристика моделирования |

1. **Оценка эффективности**

Измерения эффективности реализаций решения задачи на дереве будут производиться в единицах измерения – тактах процессора.

**Построение дерева (в тактах процессора):**

|  |  |  |
| --- | --- | --- |
| **Количество элементов дерева** | **Бинарное дерево** | **Сбалансированное бинарное дерево** |
| 500 | 816230 | 952208 |
| 1000 | 1668440 | 2741224 |

**Поиск слова (в тактах процессора)\*:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Количество элементов дерева** | **Бинарное дерево** | **Сбалансированное бинарное дерево** | **Хеш-таблица (без коллизий/80% коллизий от исходного числа)** | **Файл** |
| 16 | 1140 | 976 | 636/1867 | 7841 |
| 32 | 1563 | 1023 | 720/2093 | 11362 |
| 50 | 1767 | 1351 | 785/2341 | 21022 |
| 500 | 1893 | 1610 | 842/2473 | 43696 |
| 1000 | 2017 | 1724 | 902/2662 | 87986 |

\*в таблице указаны **средние** значения для поиска по всей структуре данных

**Объём занимаемой памяти (в байтах):**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Количество элементов дерева** | **Бинарное дерево** | **Сбалансированное бинарное дерево** | **Хеш-таблица (без коллизий/80% коллизий от исходного числа)** | **Файл** |
| 16 | 384 | 384 | 1432 / 488 | 119 |
| 32 | 768 | 768 | 2104 / 824 | 222 |
| 50 | 1200 | 1200 | 7384 / 1896 | 379 |
| 500 | 12000 | 12000 | 241944 / 11160 | 3737 |
| 1000 | 24000 | 24000 | 858232 / 22120 | 7624 |

**Среднее количество сравнений**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Количество элементов дерева** | **Бинарное дерево** | **Сбалансированное бинарное дерево** | **Хеш-таблица\*\*** | **Файл** |
| 16 | 2.750000 | 2.375000 | 2.000000 | 8.000000 |
| 32 | 5.031250 | 3.218750 | 2.000000 | 16.000000 |
| 50 | 6.000000 | 3.860000 | 2.000000 | 25.000000 |
| 500 | 9.898000 | 6.996000 | 2.500000 | 250.000000 |
| 1000 | 10.767000 | 7.987000 | 2.500000 | 500.000000 |

\*в таблице указаны **средние** значения для поиска по всей структуре данных

\*\* в моих файлах

1. **Выводы по проделанной работе**

Использование хеш-таблицы эффективно в зависимости от количества коллизий в ней, но в том случае, когда количество коллизий минимально, она занимает достаточно большой объем памяти, и наоборот, при большом времени обработки – требования к памяти будут уменьшаться. В случае деревьев, АВЛ дерево в основном выигрывает по времени поиска у несбалансированного дерева, и всегда выигрывает по **среднему** значению количества сравнений и **среднему** времени поиска по дереву. В целом, из моих исследований выше, я могу сделать несколько выводов. Во-первых при большом доступном количестве памяти выгоднее реализовывать хеш-таблицу, ведь время работы с ней выгоднее именно в таком случае. В ситуации, когда время работы совершенно не играет роли, и очень важна память, выгоднее использовать файл. Дерево же выгодно использовать при серединном случае, когда память частично важна, и частично важно время работы. Хочется сказать, что на создание АВЛ дерева в основном уходит больше времени чем на создание обычного дерева, и, чем больше элементов, тем больше разница. И, стоит отметить, что в моем случае, при добавлении или удалении узла из дерева невозможно узнать, осталось оно сбалансированным или нет, а следовательно, при любом таком действием с деревом, нужно его заново балансировать, что дополнительно сильно увеличивает время работы. В лучшем случае, нужно было использовать функцию балансировки, которая не должна была вытягивать дерево в список. Еще стоит отметить, что лучше использовать функции работы с деревом, после работы которых не приходилось каждый раз балансировать дерево

1. **Ответы на вопросы**

**1. Что такое дерево?**

Дерево – это рекурсивная структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим».

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

В виде связного списка — динамически под каждый узел.

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

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

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

Двоичное дерево поиска - двоичное дерево, для каждого узла которого сохраняется условие:  
левый потомок больше или равен родителю, правый потомок строго меньше родителя (либо наооборот).

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

У АВЛ дерева для каждой его вершины высота двух её поддеревьев различается не более чем на 1, а у идеально сбалансированного дерева различается количество вершин в каждом поддереве не более чем на 1.

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

Поиск в АВЛ дереве происходит быстрее, чем в ДДП.

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

Хеш-таблицей называется массив, заполненный элементами в порядке,

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

**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 открытом хешировании к ячейке по данному ключу прибавляется связанны список, при закрытом – новый элемент кладется в ближайшую свободную ячейку после данной.

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

Поиск в хеш-таблице становится неэффективен при большом числе коллизий –сложность поиска возрастает по сравнению с О(1). В этом случае требуется реструктуризация таблицы – заполнение её с использованием новой хеш-функции.

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

В хеш-таблице минимальное время поиска О(1). В АВЛ: О(log2n). В дереве двоичного поиска О(h), где h - высота дерева (от log2n до n)