	Министерство образования и науки 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 (ИУ)

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии (ИУ7)

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №6
"Деревья, хеш –таблицы"

Студент группы ИУ7-31Б
Костев Дмитрий

Москва 2020

Цель работы

Получить навыки применения двоичных деревьев, реализовать основные операции над деревьями: обход деревьев, включение, исключение и поиск узлов; построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска и в хеш-таблицах.

Описание условия задачи

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. **Добавить** указанное слово, если его нет в дереве (по желанию пользователя) в исходное и сбалансированное дерево. Сравнить время добавления 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 **добавление** введенного слова, вывести таблицу. Сравнить время добавления, объем памяти и количество сравнений при использовании ДДП, сбалансированных деревьев, хеш-таблиц и файла.

Техническое задание:

Входные данные:

0. Файл для построения деревьев
    - каждый новый элемент записан с новой строки
    - не пустой
    - имеет LF окончания строк

1. Номер команды - целое число в диапазоне от 0 до 15 включительно.

2. Командно-зависимые данные:

    - строка - элемент дерева. Не нулевой.

    - тип хэш функции - чило из списка

        1. Сумма
        2. XOR сумма
        3. Хеширование Пирсона
        4. Хеширование DJB2
    - максимальное кол-во коллизий - число большее нуля

    - размерности таблицы - число большее нуля

Выходные данные:

В зависимости от выбранного действия результатом работы программы могут являться:
1. Графическое представление
    - АВЛ дерева
    - ДДП
    - хэш-таблицы

2. Статистика по времени выполнения, объему занимаемой памяти и кол-ва сравнений при
    - добавлении элемента
    - при поиске
3. Информация о состоянии хэш таблицы
    - функция хэширования
    - максимальное кол-во коллизий
    - размерность таблицы
    - кол-во элементов в таблице

Интерфейс программы:

1. Перестроить деревья из файла
===== ДДП
2. показать
3. добавить элемент
===== АВЛ дерево
4. показать
5. добавить элемент
===== Хеш-таблица
6. создать из файла
7. добавить элемент
8. показать
9. состояние
===== Переструктурировать
10. самостоятельно
11. автоматически
===== Все структуры
12. добавить элемент
13. поиск элемента
===== Анализ
14. добавления элемента
15. поиска элемента

Аварийные ситуации:

1. Некорректный ввод номера команды.
   - **На входе**: число, большее, чем максмальный индекс команды или меньшее, чем минимальный.
   - **На выходе**: Сообщение об ошибке
2. Ошибка открытия файла
   - **На входе**: отсутствие файла, запрещен доступ
   - **На выходе**: Сообщение об ошибке
3. Некорректный ввод типа хэш функции
   - **На входе**: ввод, отличный от указанного в ТЗ
   - **На выходе**: Сообщение об ошибке
4. Некорректный ввод строки - элемента дерева
   - **На входе**: ввод, отличный от указанного в ТЗ
   - **На выходе**: Сообщение об ошибке
5. Некорректный ввод максимальнго кол-ва коллизий
   - **На входе**: ввод, отличный от указанного в ТЗ
   - **На выходе**: Сообщение об ошибке
6. Некорректный ввод размерности таблицы

- **На входе**: ввод, отличный от указанного в ТЗ
- **На выходе**: Сообщение об ошибке

Обращение к программе

Исполняемый файл app.exe. Запускается из терминала при помощи комманды ./app.exe filename, где filename - имя файла с данными.

Используемые структуры

Двоичное дерево поиска

Для расширяемости структуры данных используются следующие замены.

```
typedef char *btree_type_t; // Тип элементотв ДДП
typedef int (*btree_cmp_t)(btree_type_t, btree_type_t); // Компаратор для этого типа
```

Дескриптор ДДП

```
typedef struct tree
{
	btree_node_t *head; // Указатель на главный узел
	btree_cmp_t cmp; // Используемый компаратор
} btree_t;
```

Узел ДДП

```
typedef struct btree_node
{
	btree_type_t data; // Хранимые данные
	struct btree_node *left; // Указатель на левое поддерево
	struct btree_node *right; // Указатель на правое поддерево
} btree_node_t;
```

АВЛ дерево

Для расширяемости структуры данных используются следующие замены.

```
typedef char *avl_tree_type_t; // Тип элементов
typedef int (*avl_tree_cmp_t)(avl_tree_type_t, avl_tree_type_t); // Компаратор для этого типа
```

Узел АВЛ дерева

```
typedef struct avl_tree_node
{
	avl_tree_type_t data; // Хранимые данные
	struct avl_tree_node *left; // Указатель на левое поддерево
	struct avl_tree_node *right; // Указатель на правое поддерево
	size_t height; // Высота текущего поддерева
} avl_tree_node_t;
```

Дескриптор АВЛ дерева

```
typedef struct avl_tree
{
  avl_tree_node_t *head;  // Указатель на главный узел
  avl_tree_cmp_t cmp;  // Используемый компаратор
} avl_tree_t;
```

Хэш таблица

Для расширяемости структуры данных используются следующие замены.

```
typedef char *hash_type_t;  // Тип элементов
typedef int (*hash_func_t)(hash_type_t); // Хэш-функция
typedef int (*hash_cmp_t)(hash_type_t, hash_type_t); // Компаратор для этого
типа
```

Узел цепочки таблицы

```
typedef struct hash_table_node
{
      hash_type_t data;  // Хранимые данные
      struct hash_table_node *next;  // Указатель на следующий узел
} hash_table_node_t;
```

Дескриптор хэш-таблицы

```
typedef struct hash_table {
      size_t max_collision_num; // Максимальное кол-во достигнутых коллизий
      size_t elements_count; // Кол-во элементов таблицы
      hash_func_t hasher; // Хэш функция
      hash_cmp_t cmp; // Компаратор
      hash_table_node_t **table; // Указатели на начало цепочек
      size_t table_size; // Размерность таблицы
} hash_table_t;
```

Алгоритм

1. На экран пользователю выводится меню

2.Пользователь вводит номер команды

3.Выполняется действие согласно номеру команды

ДДП

- поиск узла
    - рекурсивно двигаться от корня в левое или правое поддерево в зависимости от значения ключа узла, пока не встретится нужный элемент
- включения элемента в дерево
    - поиск корня для добавления нового узла
    - включение узла в левое или правое поддерев

АВЛ дерево

- включения элемента в дерево
    - поиск корня для добавления нового узла
    - включение узла в левое или правое поддерев
    - балансировка дерева

Хэш таблица

- поиск элемента
    - вычисление хэша и переход к нужной цепочке
    - линейный поиск элемента в цепочке

Основные функции

- `btree_t *init_btree(btree_cmp_t cmp)` - Создание ДДП
- `void btree_free(btree_t *tree)` - Удаление ДДП
- `bool add_btree(btree_t *tree, btree_type_t element)` - добавление элемента
- `bool has_element_btree(btree_t *tree, btree_type_t element)` - поиск элемента
- `bool add_avl_tree(avl_tree_t *tree, avl_tree_type_t element)` - добавление элемента
- `avl_tree_t *init_avl_tree(avl_tree_cmp_t cmp)` - Создание АВЛ дерева
- `bool has_avl_tree(avl_tree_t *tree, avl_tree_type_t element)` - поиск элемента
- `avl_tree_node_t *rotate_left_avl_tree(avl_tree_node_t *node)` - Правый поворот
- `avl_tree_node_t *rotate_right_avl_tree(avl_tree_node_t *node)` - Левый поворот
- `void free_avl_tree(avl_tree_t *tree)` - Удаление АВЛ дерева
- `hash_table_t *init_hash_table(size_t, hash_func_t, hash_cmp_t);` - Создание хэш-таблицы
- `bool add_element_hash_table(hash_table_t *table, hash_type_t element);` - добавление элемента
- `bool has_element_hash_table(hash_table_t *table, hash_type_t element);` - поиск элемента
- `int restruct_hash_table(hash_table_t **, size_t new_table_size, hash_func_t);` - реструктуризация хэш таблицы
- `void free_hash_table(hash_table_t *table);` - Удаление хэш-таблицы
- `int hash_sum(char *str)` - хэш-функция суммы ASCI кодов символов строки
- `int hash_xor(char *str)` - хэш-функция суммы операции логического или ASCI кодов символов строки и случайного постоянного числа
- `int hash_pearson(char *str)` - 4-х байтная реализация хэш-функции Пирсона

Тесты

Ввод	Результат	Причина
-1	Сообщение об ошибке	Неверный пункт меню
16	Сообщение об ошибке	Неверный пункт меню
.app.exe	Сообщение об ошибке	Отсутствует файл
.app.exe undefined_file.txt	Сообщение об ошибке	Отсутствует файл
Ввод пустого элемента	Сообщение об ошибке	Ввод пустого элемента

Оценка эффективности

В таблицах приведены средние числа для 1000 повторений

Добавление элемента

Размер СД	Двоичное дерево поиска			АВЛ дерево			Хэш таблица		
	Время, тики	Кол-во сравнений	Объём СД, Байты	Время, тики	Кол-во сравнений	Объём СД, Байты	Время, тики	Кол-во сравнений	Объём СД, Байты
16	0.177	4.250	400	0.282	8.250	528	0.183	1.625	408
32	0.150	5.500	784	0.266	11.250	1040	0.128	1.531	792
64	0.195	6.266	1552	0.387	14.578	2064	0.143	1.656	1544
128	0.265	7.641	3088	0.513	17.320	4112	0.198	2.297	2968
256	0.338	8.812	6160	0.596	20.262	8208	0.230	2.520	5928
512	0.324	10.072	12304	0.559	23.180	16400	0.219	3.797	11752
1024	0.375	11.496	24592	0.656	26.379	32784	0.279	5.929	23448

Из таблицы можно сделать выводы, что АВЛ дерево имеет самую низкую скорость вставки, это обусловлено процессом балансировки. В данной таблице видим, что самую высокую скорость генерации имеет хэш-таблица.

Поиск элемента

Размер СД	Двоичное дерево поиска			АВЛ дерево			Хэш таблица			Файл		
	Время, тики	Кол-во сравнений	Объём СД, Байты	Время, тики	Кол-во сравнений	Объём СД, Байты	Время, тики	Кол-во сравнений	Объём СД, Байты	Время, тики	Кол-во сравнений	Объём СД, Байты
16	0.122	5.312	400	0.086	3.438	528	0.098	1.625	408	1.057	8.500	0
32	0.100	6.531	784	0.063	4.312	1040	0.068	1.531	792	1.118	16.500	0
64	0.106	7.281	1552	0.077	5.250	2064	0.061	1.656	1544	1.621	32.500	0
128	0.182	8.648	3088	0.134	6.281	4112	0.097	2.297	2968	2.853	64.500	0
256	0.198	9.816	6160	0.149	7.238	8208	0.094	2.520	5928	4.479	128.500	0
512	0.232	11.074	12304	0.169	8.244	16400	0.111	3.797	11752	8.428	256.500	0
1024	0.276	12.497	24592	0.198	9.296	32784	0.142	5.929	23448	18.258	512.500	0

Из таблицы мы можем сделать выводы, что последовательный поиск в файле имеет самую меньшую скорость (причем тестирование производилось на твердотельном диске). Следовательно, необходимо использовать АСД для поиска элементов. Лучше всего себя показала хэш-таблица, как по памяти, так и по скорости работы.

Контрольные 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 родителю, правый потомок строго меньше родителя.

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, благодаря сбалансированности дерева.

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.

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:
  - при открытом хешировании к ячейке поданному 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 O(1). В этом случае требуется реструктуризация таблицы – заполнение её с использованием новой хеш-функции или расширение таблицы.

10. Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска и в хеш-таблицах.
- В хеш-таблице минимальное время поиска O(1).
- В АВЛ дереве: O(log_2n).

- В ДДП O(h), где h - высота дерева (от log_2n до n).

Вывод

В задачах связанных с быстром поиском случайных данных лучше всего подходит АСД Хэш-таблица с правильно подобранной хэш-функцией для данного типа данных. АВЛ дерево разумно использовать при большом кол-ве коллизий в хэш-таблице (т.е. если подобрать хэш-функцию не удалось).