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
Лабораторная работа №7
«Сбалансированные деревья, хэш-таблицы»

Группа ИУ7-34Б

Дисциплина Типы и структуры данных

Вариант 14

Студент Козлитин Максим Александрович

Преподаватель Силантьева Александра Васильевна

2022г.

Цель работы

Построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска и в хеш-таблицах. Сравнить эффективность устранения коллизий при внешнем и внутреннем хешировании.

Условие задачи

1. Построить хеш-таблицу по указанным данным.
2. Сравнить эффективность поиска в сбалансированном двоичном дереве, в двоичном дереве поиска и в хеш-таблице.
   Вывести на экран деревья и хеш-таблицу.
3. Подсчитать среднее количество сравнений для поиска данных в указанных структурах.
4. Произвести реструктуризацию хеш-таблицы, если среднее количество сравнений больше указанного.
5. Оценить эффективность использования этих структур (по времени и памяти) для поставленной задачи.
6. Сбалансировать дерево.
7. Сравнить время поиска, объем памяти.

Описание исходных данных

- Выбор действия:
  Целое число от 0 до 10.
- Строка:
  Набор букв максимальным размером 1000 символов.
- Буква:
  Один символ, который является буквой.

Способ обращения к программе

Работа с программой осуществляется с помощью консоли.

Программа запускается с помощью команды: **./app.exe**

Далее пользователь выполняет взаимодействие с помощью меню.
Меню выводится в начале и, далее, каждые 3 введенные команды.

Описания возможных аварийных ситуаций и ошибок пользователя

1. Ввод не совпадающий с форматом входных данных (описано в Описание исходных данных).
2. Ввод целых чисел превышающих максимальное допустимое значение типа данных int.

Описание внутренних структур данных

Бинарное дерево, узел дерева:

```
typedef struct btree_node_t btree_node_t;


struct btree_node_t
{
    char data; // содержимое узла
    size_t cnt; // количество элементов
    size_t weight; // вес поддерева с корнем в данной вершине
    btree_node_t *left; // левый сын
    btree_node_t *right; // правый сын
};


typedef struct
{
    btree_node_t *root; // корень дерева
} binary_tree_t; // бинарное дерево
```

size_t *используется для задания размеров наших массивов, так как является особым макросом, значение которого позволяет адресовать массив любой теоретически возможной длины для данной машины.*

char *- символьный тип данных.*

Массив узлов:

```
typedef struct
```

```
{
    size_t size; // размер массива узлов
    btree_node_t *data; // содержимое массива узлов
} btree_nodes_t; // массив узлов
```

Массив узлов *используется для конвертации дерева в массив, позволяя сохранить содержимое узлов в него.*

size_t *используется для задания размеров наших массивов, так как является особым макросом, значение которого позволяет адресовать массив любой теоретически возможной длины для данной машины.*

Сбалансированное дерево, узел дерева:

```
typedef struct scapegoat_node_t scapegoat_node_t;


struct scapegoat_node_t
{
    char data; // содержимое узла
    size_t cnt; // количество элементов
    size_t weight; // вес поддерева с корнем в данной вершине
    scapegoat_node_t *left; // левый сын
    scapegoat_node_t *right; // правый сын
};


typedef struct
{
    scapegoat_node_t *root; // корень дерева
} scapegoat_tree_t; // массив узлов
```

size_t *используется для задания размеров наших массивов, так как является особым макросом, значение которого позволяет адресовать массив любой теоретически возможной длины для данной машины.*

char *- символьный тип данных.*

Массив узлов (Сбалансированное дерево):

```
typedef struct
{
    size_t size; // размер массива узлов
    scapegoat_node_t *data; // содержимое массива узлов
} scapegoat_nodes_t; // массив узлов
```

Массив узлов *используется для конвертации дерева в массив, позволяя сохранить содержимое узлов в него.*

size_t *используется для задания размеров наших массивов, так как является особым макросом, значение которого позволяет адресовать массив любой теоретически возможной длины для данной машины.*

Хэш-таблица:

```
#define HASH_TABLE_OPEN_DEFAULT_DIVISOR 151 // стандартный модуль хэш-таблицы


typedef struct
{
    list_t *data; // содержимое хэш таблицы
    int divisor; // текущий модуль хэш таблицы
} hash_table_open_t; // хэш-таблица
```

Описание алгоритма

Балансировка - **`scapegoat_balance`**:

1. Ищем вершину которая нарушает балансировку дерева (вес левого и правого поддеревьев отличается больше чем на 1).
2. Получаем inorder-обход дерева, таким образом по свойству бинарного дерева на выходе будет отсортированный массив узлов.
3. Берем медиану и ставим в корень, рекурсивно повторяем ту же операцию для других вершин поддерева найденной вершины.

Добавление элемента в хэш-таблицу - **`hashtableopen_add`**:

1. Получаем хэш элемента.
2. Добавляем в список данного хэша элемент.

Добавление элемента в хэш-таблицу - **`hashtableopen_del`**:

1. Получаем хэш элемента.
2. Удаляем из списка данного хэша элемент.

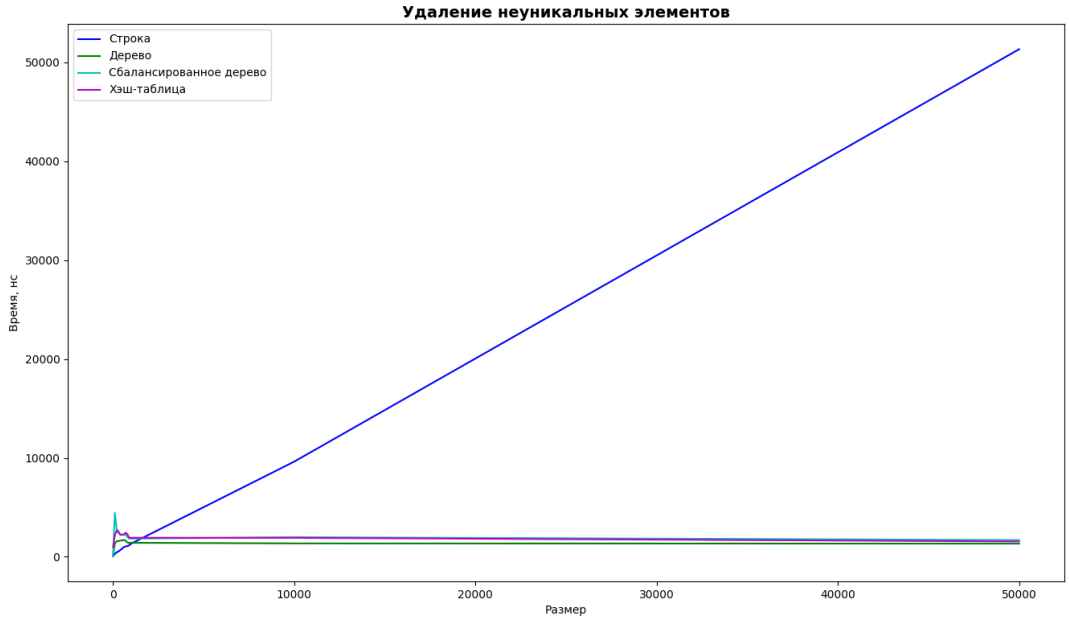
Оценка эффективности

Алгоритм измерения времени:

1. Измеряется текущее время - начало.
2. Запускается выполнение функции.
3. Измеряется текущее время - конец.
4. Вычисляется количество наносекунд прошедших от начала и до конца.
5. Полученное значение добавляется в сумму.
6. Данное измерение производится 100 раз, вычисляя итоговую сумму всех запусков.
7. Ответ - среднее арифметическое.

Удаление неуникальных:

Размер	Строка, нс	Дерево, нс	Сбалансированное дерево, нс	Хэш-таблица, нс
1	79	86	72	988
100	281	1351	4437	2171
200	448	1553	2783	2659
300	540	1598	2594	2429
400	669	1627	2197	2247
500	820	1653	2214	2244
600	973	1699	2216	2258
700	1046	1598	2220	2437
800	1082	1436	1984	2263
900	1172	1394	1851	1887
1000	1324	1412	1835	1921
10000	9624	1348	1976	1913
50000	51341	1325	1691	1546

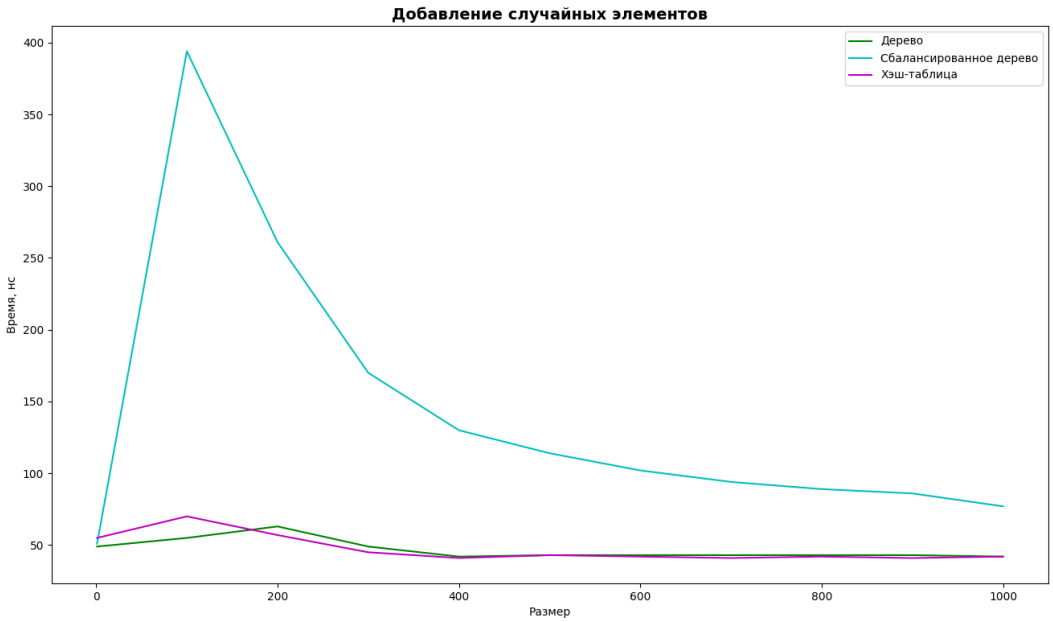


По данным можем заметить, что данный алгоритм быстрее всего выполняется с использованием структуры данных бинарное дерево поиска.
Сбалансированное дерево уступает, непосредственно, из-за балансировки.

Хэш-таблица незначительно уступает по эффективности, из-за того что в бинарном дереве поиска, мы обходим только интересующие нас элементы, нежели все возможные значения хэш-таблицы.

Добавление

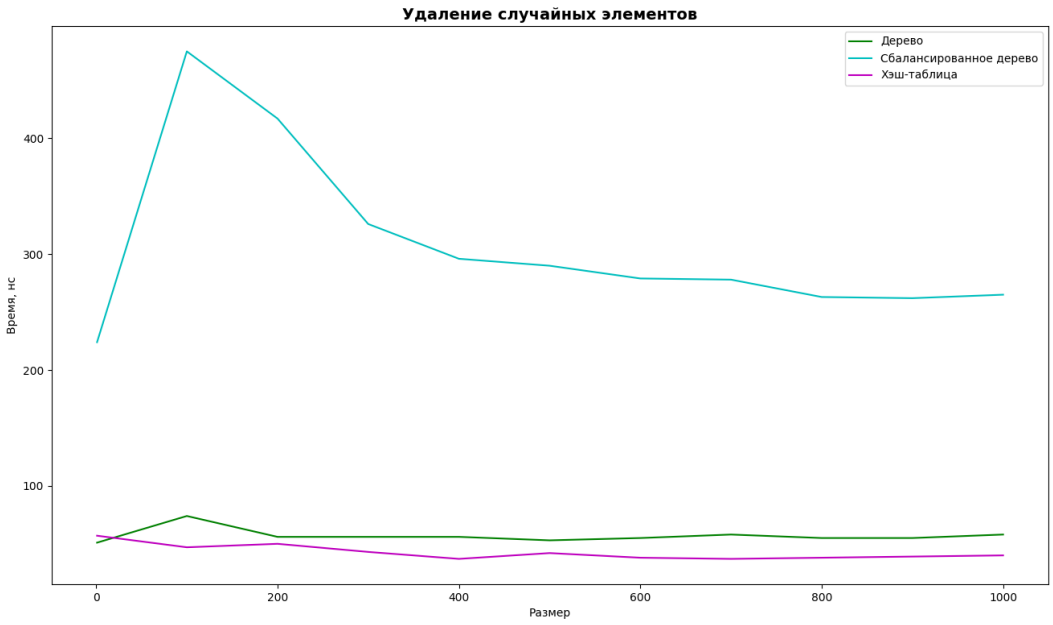
	Время, нс		
Размер	**Дерево**	**Сбалансированное дерево**	**Хэш-таблица**
1	51	224	57
100	74	475	47
200	56	417	50
300	56	326	43
400	56	296	37
500	53	290	42
600	55	279	38
700	58	278	37
800	55	263	38
900	55	262	39
1000	58	265	40



Добавление элемента примерно одинаковое для бинарного дерева поиска и хэш-таблицы, но балансировка сильно ухудшает эффективность, что можно заметить по графику.

Удаление

	Время, нс		
Размер	**Дерево**	**Сбалансированное дерево**	**Хэш-таблица**
1	49	51	55
100	55	394	70
200	63	261	57
300	49	170	45
400	42	130	41
500	43	114	43
600	43	102	42
700	43	94	41
800	43	89	42
900	43	86	41
1000	42	77	42



Удаление элемента примерно одинаковое для бинарного дерева поиска и хэш-таблицы, но балансировка сильно ухудшает эффективность, что можно заметить по графику.

Поиск

	Время, нс		
Размер	**Дерево**	**Сбалансированное дерево**	**Хэш-таблица**
1	34	33	33
100	51	64	48
200	51	58	36
300	50	51	36
400	52	51	35
500	57	49	35
600	52	53	38
700	53	53	37
800	54	53	35
900	60	55	35
1000	55	57	36

Поиск элемента приблизительно одинаков для данных структур, но стоит сказать, что каждая из них имеет свои приемущества в зависимости от набора данных.
При отсортированной изначальной строке поиск в дерево будет менее эффективен чем в сбалансированном дереве и хэш-таблице.
При этом хэш-таблица отличается эффективностью от противопоставляемых структур данных, уменьшить эффективность хэш-таблицы может лишь более сложная хэш-функция. Но при этом сравнения будут происходить быстрее, чем при бинарных деревьях поиска.

Память

	Память	
Структура данных	**Структура**	**1 элемент**
Строка	50001	50001
Дерево	8	40
Сбалансированное дерево	8	40
Хэш-таблица	2432	40

Можем заметить, что наиболее эффективным по памяти структурами данных являются деревья.
Строка требует много памяти из-за большого объема статичных данных, учитывая, что строка хранит повторяющиеся буквы, то использование памяти не оптимизировано.
Деревьям же требуется хранить только узлы и саму структуру, что позволяет минимизировать использование памяти.
Хэш-таблица требует много памяти на хранение самой таблицы, нам необходимо выделять массив размером модуля таблицы, из-за этого получается такой объем требуемой памяти.

Вывод

В ходе проделанной работы, удалось построить и обработать хеш-таблицы и сбалансированные деревья, сравнить эффективность данных структур данных для конкретной задачи.

По результатам, проведенной оценки эффективности, было выявлено, что использование сбалансированного бинарного дерева поиска не необходимо, поэтому имеет смысл в пользу эффективности отказать от балансировки.
Бинарное дерево поиска в сравнение с хэш-таблицей показывает практически идентичную скорость работы, но объем требуемой памяти у хэш-таблицы гораздо больше.
Суммируя вышесказанное, можно сказать, что наиболее эффективная структура данных для данной задачи - бинарное дерево поиска.
Стоит отметить, что вывод вывод в виде дерева, при данной структуре получается весьма громоздки, в отличии от сбалансированного бинарного дерева поиска.

Подробные результаты и методы оценки эффективности программы можно изучить в пункте «Оценка эффективности».

Контрольные вопросы

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

   Идеально сбалансированное дерево - дерево, у которого вес левого и правого поддерева отличается не более, чем на единицу.
   АВЛ дерево - дерево, у которого высота левого и правого поддерева

отличается не более, чем на единицу.

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

   Сам процесс поиска ничем не отличается, но из-за того что АВЛ-дерево балансирует высоту дерева, то и поиск происходит более стабильно.

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

   Хеш-таблица - структура данных, которая позволяет по значению ключа сразу определять индекс элемента массива, в котором хранится информация.
   Для этого необходимо создать такую функцию, по которой можно вычислить этот индекс. Такая функция называется хеш-функцией и она ставит в соответствие каждому ключу индекс ячейки

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

   Коллизия - ситуация, при которой два разных ключа дают одинаковое значение хэш-функции, т. е. одинаковый индекс.

   Различают два основных метода разрешения коллизий: открытое и закрытое хеширование.
   Открытое (метод цепочек) - каждому индексу соответствует связный список значений.
   Закрытое (открытая адресация) - при возникновение коллизии элемент ставится выполняется поиск, пока не найдется данный элемент или свободная ячейка таблицы.

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

   Поиск в хэш-таблицах становится неэффективен при достаточном количестве коллизий (3-4). Так как для поиска элемента требуется производить больше сравнений.

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

   Эффективность поиска зависит от поставленной задачи, в каких-то случаях можно ограничиваться деревья в пользу других преимуществ, а в каких-то стоит отдать предпочтение хэш-таблицей.
   Данная структура является наиболее эффективной для поиска.