| Изображение выглядит как герб, эмблема, символ, нашивка  Автоматически созданное описание | **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 ПО ЛАБОРАТОРНОЙ РАБОТЕ №7**

**ПО ДИСЦИПЛИНЕ:**

**ТИПЫ И СТРУКТУРЫ ДАННЫХ**

**Деревья, хеш-таблицы**

**Вариант 0**

Студент **Ильченко Е. А.**

Группа **ИУ7-34Б**

Название предприятия **НУК ИУ МГТУ им. Н. Э. Баумана**

| Студент | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ильченко Е. А.** |
| --- | --- |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Силантьева А. В.** |

**2024 г.**

# Описание условия задачи

Сбалансировать дерево (задача №6) после удаления повторяющихся букв. Вывести его на экран в виде дерева. Составить хеш-таблицу, содержащую буквы и количество их вхождений во введенной строке. Вывести таблицу на экран. Реализовать операции добавления и удаления введенной буквы во всех структурах. Осуществить поиск введенной буквы в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Сравнить время поиска, объем памяти и количество сравнений при использовании различных структур данных.

# Описание ТЗ

## 1. Описание исходных данных и результатов работы программы

**Входные данные:**

Пользовательская команда из доступных и необходимые аргументы определенного сценария:

1: Добавить строку в BST дерево

2: Добавить узел в BST дерево

3: Удалить узел из BST дереве

4: Вывести BST дерево

5: Вывести узел BST дерева

6: Удалить повторяющиеся буквы и сбалансировать

7: Добавить строку в AVL дерево

8: Добавить узел в AVL дерево

9: Удалить узел из AVL дереве

10: Вывести AVL дерево

11: Вывести узел AVL дерева

12: Добавить строку в хэш-таблицу

13: Добавить символ в хэш-таблицу

14: Удалить символ из хэш-таблицы

15: Вывести хэш-таблицу

16: Вывести символ из хэш-таблицы

17: Сравнить эффективность поиска в различных структурах

18: Очистить BST дерево

19: Очистить AVL дерево

20: Очистить хэш-таблицу

21: Вывести меню

0: Выход

**Выходные данные:**

BST дерево, AVL дерево, хэш-таблица, измененные в соответствии с выбранной операцией. Сравнение времени поиска, объема памяти и количество сравнений при использовании AVL дерева, BST дерева и хэш-таблицы.

## 2. Описание задачи, реализуемой в программе

Задача, реализуемая в программе, заключается в реализации операций работы с двоичным деревом поиска, AVL деревом и хэш-таблицей; сравнение эффективности поиска в сбалансированных (AVL) деревьях, в двоичных деревьях поиска и в хеш-таблицах.

## 3. Способ обращения к программе

Запуск исполняемого файла

./app.exe

Далее выбирается, какой пункт меню выполнить

## 4. Описание возможных аварийных ситуаций и ошибок пользователя

1. Неверный ввод строки для вставки в структуру: сообщение “Неверный ввод строки”
2. Ввод пустой строки для вставки в структуру: сообщение “Пустая строка”
3. Неверный ввод символы для вставки в структуру: сообщение “Ошибка ввода символа”
4. Вывод пустого BST, AVL дерева: сообщение “Пустое дерево”
5. Ошибка памяти при добавлении символа к хэш-таблицу: сообщение “Ошибка: не удалось добавить символ”
6. Вывод пустой хэш-таблицы: “Хэш-таблица пуста”
7. Неверный ввод пункта меню: сообщение “Неверная команда”

## 5. Описание внутренних структур данных

Узел BST дерева

typedef struct BSTNode

{

char data;

int count;

struct BSTNode \*left;

struct BSTNode \*right;

} BSTNode;

Узел AVL дерева

typedef struct AVLNode

{

char data;

int height;

int count;

struct AVLNode \*left;

struct AVLNode \*right;

} AVLNode;

Хэш-таблица

typedef struct HashTableNode

{

char data;

struct HashTableNode \*next;

} HashTableNode;

typedef struct HashTable

{

HashTableNode \*\*buckets;

size\_t size;

size\_t count;

} HashTable;

## 6. Описание функций

**BST дерево**

BSTNode \*create\_node\_bst(char data);

Создание узла дерева

BSTNode \*insert\_bst(BSTNode \*root, char data);

Вставить узел в дерево

BSTNode \*delete\_node\_bst(BSTNode \*root, char data);

Удалить узел из дерева

void free\_tree\_bst(BSTNode \*root);

Очистить дерево

void save\_to\_png\_from\_graphviz\_bst(BSTNode\* node, int option);

Сохранить дерево в png с помощью graphviz

BSTNode \*search\_bst(BSTNode \*root, char data, int \*cmp\_count);

Найти элемент в дереве

BSTNode \*delete\_duplicates(BSTNode \*root);

Удалить дубликаты

**AVL дерево**

AVLNode \*create\_node\_avl(char value)

Создание узла дерева

AVLNode \*rotate\_right(AVLNode \*y)

Поворот дерева направо

AVLNode \*rotate\_left(AVLNode \*x)

Поворот дерева налево

AVLNode \*insert\_avl(AVLNode \*root, char key);

Вставить узел в дерево

AVLNode \*delete\_node\_avl(AVLNode \*root, char data);

Удалить узел из дерева

AVLNode \*search\_avl(AVLNode \*root, char data, int \*cmp\_count);

Найти элемент в дереве

void free\_tree\_avl(AVLNode \*root);

Очистить дерево

AVLNode \*balance\_tree(BSTNode \*root);

Сбалансировать BST дерево

void save\_to\_png\_from\_graphviz\_avl(AVLNode \*node, int option);

Сохранить дерево в png с помощью graphviz

**Хэш-таблица**

HashTable \*create\_hash\_table(size\_t init\_size);

Создать хэш-таблицу

int insert\_hash\_table(HashTable \*hash\_table, char data);

Вставить элемент в хэш-таблицу

int search\_hash\_table(HashTable \*table, char data, int \*cmp\_count);

Найти элемент в хэш-таблице

void delete\_in\_hash\_table(HashTable \*table, char data);

Удалить элемент из хэш-таблицы

void print\_hash\_table(HashTable \*table);

Вывести хэш-таблицу

void free\_hash\_table(HashTable \*\*table);

Освободить хэш-таблицу

## 7. Описание алгоритмов

**AVL дерево**

**Алгоритм вставки элемента в AVL дерево**

Инициализация нового узла:

1. Если переданный корень поддерева пустой (то есть равен NULL), создается новый узел с заданным значением.

Рекурсивный поиск места для вставки:

1. Если дерево непустое, сравнивается значение вставляемого элемента с данными текущего узла:
   1. Если значение меньше данных текущего узла, осуществляется рекурсивный вызов функции вставки для левого поддерева.
   2. Если значение больше, вызов производится для правого поддерева.
   3. Если значение совпадает с данными текущего узла (например, символ уже существует в дереве), увеличивается счётчик повторений этого узла, и вставка на этом завершается.

Обновление высоты узла:

1. После того как новый элемент добавлен в поддерево, высота текущего узла пересчитывается. Высота определяется как 1 плюс максимум из высот левого и правого дочерних узлов. Это необходимо для обеспечения корректности структуры AVL-дерева.

Расчёт баланса текущего узла:

1. Вычисляется баланс — разность высот левого и правого поддерева текущего узла. Если баланс выходит за пределы диапазона [-1, 1], дерево становится несбалансированным, и требуется выполнить вращение.

Восстановление баланса:

1. Если левое поддерево стало слишком высоким (баланс > 1):
   1. Если вставленный элемент находится в левом поддереве левого дочернего узла, выполняется правый поворот.
   2. Если вставленный элемент находится в правом поддереве левого дочернего узла, сначала выполняется левый поворот у левого дочернего узла, а затем правый поворот у текущего узла.
2. Если правое поддерево стало слишком высоким (баланс < -1):
   1. Если вставленный элемент находится в правом поддереве правого дочернего узла, выполняется левый поворот.
   2. Если вставленный элемент находится в левом поддереве правого дочернего узла, сначала выполняется правый поворот у правого дочернего узла, а затем левый поворот у текущего узла.

Возврат текущего корня:

1. После всех операций узел возвращается как корень текущего поддерева.

**Алгоритм поиска узла с заданным значением в AVL дереве**

Инициализация поиска:

1. Начинаем с корня дерева. Передаём в алгоритм указатель на корневой узел, значение для поиска и переменную для подсчёта сравнений.

Проверка текущего узла:

1. Если текущий узел пустой, это означает, что элемент отсутствует в дереве, и поиск завершается.
2. Если значение текущего узла совпадает с искомым, элемент найден, и поиск завершается.

Сравнение искомого значения с текущим узлом:

1. Если искомое значение меньше данных текущего узла, двигаемся в левое поддерево.
2. Если значение больше, переходим в правое поддерево.
3. При каждом сравнении увеличиваем счётчик сравнений, фиксируя количество шагов, необходимых для поиска.

Рекурсивное продолжение:

1. Алгоритм повторяется для дочернего узла, пока не будет найден элемент или не достигнут конец дерева (пустой узел).

Результат поиска:

1. Если элемент найден, возвращается указатель на соответствующий узел.
2. Если поиск достиг пустого узла, возвращается индикатор отсутствия элемента (например, NULL).

**Алгоритм удаления узла с заданным значением в AVL дереве**

Инициализация удаления:

1. Начинаем с корня дерева. В функцию передаётся указатель на корень, значение удаляемого элемента.

Поиск узла для удаления:

1. Если текущий узел пустой, удаление невозможно, элемент отсутствует.
2. Если значение меньше данных текущего узла, продолжаем поиск в левом поддереве.
3. Если значение больше, переходим в правое поддерево.

Удаление узла:

1. Когда узел с искомым значением найден:
   1. Если у узла один или ноль детей:
      1. Если у узла нет детей, он просто удаляется.
      2. Если у узла один ребёнок, он заменяется этим ребёнком.
   2. Если у узла два ребёнка:
      1. Находим минимальный элемент в правом поддереве (или максимальный в левом).
      2. Копируем его значение в текущий узел.
      3. Рекурсивно удаляем этот минимальный элемент в правом поддереве.

Обновление высоты узлов:

1. После удаления узла обновляем высоты всех предков на пути вверх.

Проверка баланса:

1. Проверяем баланс текущего узла.
2. Если узел стал несбалансированным, выполняем соответствующие вращения:
   1. Левый дисбаланс:
      1. Если левый ребёнок сбалансирован или перегружен слева, выполняем правое вращение.
      2. Если перегрузка слева-справа, выполняем левое вращение левого ребёнка, затем правое вращение текущего узла.
   2. Правый дисбаланс:
      1. Если правый ребёнок сбалансирован или перегружен справа, выполняем левое вращение.
      2. Если перегрузка справа-слева, выполняем правое вращение правого ребёнка, затем левое вращение текущего узла.

Возврат результата:

1. Возвращается обновлённый корень дерева, чтобы сохранить связь между уровнями дерева.

**Хэш-таблица**

**Алгоритм вставки элемента в хэш-таблицу**

Вычисление индекса:

1. Для вставляемого элемента рассчитывается индекс в массиве хэш-таблицы с использованием хэш-функции. Индекс определяется как hash(data) % size, где size — размер массива.

Поиск в цепочке:

1. Переходим к соответствующему индексу в массиве и начинаем проверку элементов в цепочке.
2. Проходим по цепочке, подсчитывая её длину.

Проверка длины цепочки:

1. Если длина цепочки превышает допустимый порог (например, 3 элемента), это сигнал о необходимости перераспределения таблицы:
   1. Увеличиваем размер таблицы.
   2. Перерассчитываем индексы всех элементов.
   3. После увеличения таблицы пересчитываем индекс для текущего вставляемого элемента.

Создание нового узла:

1. Если цепочка допустимой длины, создаём новый узел для хранения данных:
   1. Выделяется память для нового узла.
   2. Записывается значение в новый узел.
   3. Указатель нового узла устанавливается на начало цепочки (текущий первый элемент).

Добавление узла в таблицу:

1. Новый узел становится первым элементом цепочки в соответствующем индексе массива.
2. Увеличивается общий счётчик элементов в хэш-таблице.

Завершение:

1. Функция возвращает статус операции:
   1. Успешная вставка – 0.
   2. Ошибка – 1.

**Алгоритм поиска элемента в хэш-таблице**

Вычисление хэш-индекса

1. Хэш-функция принимает значение data и размер хэш-таблицы size. Вычисляется индекс: index = hash(data) % size. Этот индекс указывает на список (цепочку), где может находиться элемент.

Начало обхода цепочки

1. Указатель current устанавливается на начало цепочки по адресу buckets[index].

Перебор элементов в цепочке

1. Выполняется последовательный просмотр узлов в цепочке:
   1. Для каждого узла сравнивается хранимое значение current->data с искомым data.
   2. Если найдено совпадение, функция завершает выполнение и возвращает положительный результат (элемент найден).
   3. При каждом сравнении увеличивается счётчик сравнений.

Достижение конца цепочки

1. Если узлы цепочки заканчиваются (указатель current становится равным NULL), это означает, что элемент отсутствует в хэш-таблице. В этом случае возвращается отрицательный результат.

**Алгоритм удаления элемента с заданным значением из хэш-таблицы**

Вычисление хэш-индекса

1. Используя хэш-функцию, вычисляется индекс: index = hash(data) % size. Этот индекс указывает на цепочку, где потенциально находится удаляемый элемент.

Инициализация указателей

1. Указатель current устанавливается на начало цепочки по адресу buckets[index]. Дополнительно создаётся указатель prev, который изначально равен NULL, для отслеживания предыдущего узла.

Перебор цепочки

1. Выполняется последовательный обход узлов в цепочке
   1. На каждом шаге проверяется, совпадает ли значение текущего узла current->data с удаляемым значением data.
   2. Если совпадение найдено:
      1. Если это первый узел в цепочке (prev == NULL), обновляется указатель начала цепочки: buckets[index] = current->next.
      2. Если это не первый узел, предыдущий узел prev перенаправляется на следующий за текущим: prev->next = current->next.
      3. Удаляется текущий узел current, а счётчик элементов таблицы уменьшается.

Продолжение или завершение

1. Если элемент найден и удалён, алгоритм завершает выполнение.
2. Если конец цепочки достигнут, значит, элемент отсутствует, и удаление не производится.

# Тесты

| **Тест** | **Входные данные** | **Выходные данные** |
| --- | --- | --- |
| Добавить элемент в AVL дерево | Вставить символ A в пустое дерево | Узел A добавлен, дерево: A |
| Удалить элемент из AVL дерева | Удалить узел C | Успешное удаление элемента из дерева |
| Проверить наличие элемента в AVL дереве | Проверить наличие символа А в дереве с элементами A, B, C | Символ А найден в дереве |
| Вывести пустое AVL дерево | Пустое дерево | Сообщение: “Пустое дерево” |
| Добавить элемент в хэш-таблицу | Вставить символ A в таблицу | Сообщение: “Символ А добавлен” |
| Удалить элемент из хэш-таблицы | Удалить символ А из таблицы | Успешное удаление элемента |
| Проверить наличие элемента из хэш-таблицы, который там есть | Проверить наличие символа А в хэш-таблице из элементов A, B, C | Сообщение: “Символ А найден в хэш-таблице” |
| Проверить наличие элемента из хэш-таблицы, которого там нет | Проверить наличие символа А в хэш-таблице из элементов B, C | Сообщение: “Символ А не найден в хэш-таблице” |
| Вывести хэш-таблицу | Хэш-таблица из символов ABCD | Контейнер 0: NULL  Контейнер 1: NULL  Контейнер 2: A -> NULL  Контейнер 3: B -> NULL  Контейнер 4: C -> NULL  Контейнер 5: D -> NULL  Контейнер 6: NULL |
| Вывести пустую хэш-таблицу | Пустая хэш-таблица | Сообщение: “Хэш-таблица пуста” |
| Ввод неверной команды | Команда: “а” | Сообщение: “Неверная команда” |
| Ввод пустой строки в структуру | Пустая строка | Сообщение: “Пустая строка” |
| Неверный ввод элемента в структуру | Ошибка ввода элемента | Сообщение: “Ошибка ввода символа” |
| Добавление уже имеющегося элемента в хэш-таблицу | Повторное добавление элемента А | Символ ‘А’ уже есть в хэш-таблице |

# Оценка эффективности

Эффективность по времени/памяти считалась путем замера 1000 раз методов поиска элементов для BST дерева, AVL дерева и хэш-таблицы и усреднения результатов

Измерения проводились на MacBook Pro 13 2019

| **Количество элементов** | **Структура данных** | **Время, нс** | **Память, байт** | **Количество сравнений** |
| --- | --- | --- | --- | --- |
| 4 | BST дерево | 189.00 | 208 | 2 |
| AVL дерево | 175.00 | 288 | 2 |
| Хэш-таблица | 145.00 | 112 | 1 |
| 8 | BST дерево | 201.00 | 416 | 3 |
| AVL дерево | 189.00 | 576 | 3 |
| Хэш-таблица | 167.00 | 224 | 1 |
| 16 | BST дерево | 221.00 | 832 | 4 |
| AVL дерево | 201.00 | 1152 | 4 |
| Хэш-таблица | 195.00 | 384 | 1 |
| 32 | BST дерево | 238.00 | 1664 | 5 |
| AVL дерево | 228.00 | 2304 | 5 |
| Хэш-таблица | 204.00 | 800 | 1 |
| 64 | BST дерево | 259.00 | 3328 | 9 |
| AVL дерево | 233.00 | 4608 | 6 |
| Хэш-таблица | 212.00 | 1632 | 1 |
| 128 | BST дерево | 287.00 | 5656 | 10 |
| AVL дерево | 267.00 | 7616 | 7 |
| Хэш-таблица | 222.00 | 4200 | 1 |
| 256 | BST дерево | 317.00 | 11312 | 13 |
| AVL дерево | 287.00 | 13121 | 8 |
| Хэш-таблица | 232.00 | 7144 | 1 |
| 512 | BST дерево | 358.00 | 18312 | 16 |
| AVL дерево | 303.00 | 21432 | 9 |
| Хэш-таблица | 254.00 | 12457 | 1 |
| 1024 | BST дерево | 405.00 | 32712 | 25 |
| AVL дерево | 356.00 | 37123 | 10 |
| Хэш-таблица | 286.00 | 24293 | 1 |

# Вывод

Тестирование эффективности программы на различных объемах данных позволило получить временные показатели для операций поиска с использованием разных структур. Результаты показали, что сбалансированное дерево работает быстрее обычного за счет оптимальной организации узлов, однако оно требует больше памяти для хранения информации о высоте узлов, что необходимо для поддержания сбалансированности. Хеш-таблица, в свою очередь, превосходит деревья по скорости и по памяти. Скорость доступа к элементам хеш-таблицы близка к O(1), но требуется хеширования элементов, что увеличивает затраты по времени.

Хеш-таблица использует более компактную структуру данных (массив), что способствует экономии памяти. При правильной настройке и минимальных коллизиях таблица требует памяти только для массива и значений, что подтверждает её эффективную реализацию и подбор хеш-функции. Даже если хеш-таблица занимает больше памяти, её структура позволяет сэкономить ресурсы на уровне общего использования.

В среднем хеш-таблица работает быстрее BST дерева на 40% и быстрее AVL дерева на 20%. По памяти хеш-таблица занимает в среднем на 50% меньше памяти, чем BST дерево и на 70% меньше памяти, чем AVL дерево.

# Ответы на контрольные вопросы

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

Идеально сбалансированное дерево и AVL-дерево оба стремятся поддерживать сбалансированность для эффективных операций.

* Идеально сбалансированное дерево имеет одинаковую высоту левого и правого поддеревьев для каждого узла, что дает минимальную высоту, но требует жесткой балансировки.
* AVL-дерево позволяет разницу в высоте поддеревьев до 1 (баланс-фактор от -1 до +1), что делает балансировку менее строгой, но более гибкой и эффективной.

В результате, AVL-дерево проще поддерживать, так как требует меньше операций при вставке и удалении элементов, чем идеально сбалансированное дерево.

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

Поиск в AVL-дереве осуществляется аналогично поиску в обычном дереве двоичного поиска. Однако основное отличие заключается в том, что AVL-дерево поддерживает балансировку после каждой операции вставки или удаления, чтобы сохранять сбалансированную структуру дерева.

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

Хеш-таблица – это структура данных, обеспечивающая эффективное выполнение операций вставки, удаления и поиска. Она использует хеш-функцию для преобразования ключа в индекс массива, где сохраняются значения.

Принцип работы включает: выбор хеш-функции, выделение массива определенного размера и разрешение коллизий (если два ключа хешируются в один и тот же индекс).

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

Коллизии происходят, когда два разных ключа хешируются в один и тот же индекс. Методы их разрешения включают:

* Цепочки: каждый индекс массива соответствует связанному списку.
* Открытое хеширование: при коллизии ищется следующий свободный слот в массиве.
* Двойное хеширование: для нахождения следующего индекса при коллизии используются две хеш-функции.

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

Поиск в хеш-таблицах становится менее эффективным при большом числе коллизий, что может привести к удлинению цепочек или росту размера области для открытого адреса.

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

* В AVL-деревьях и деревьях двоичного поиска поиск занимает время, пропорциональное логарифму числа элементов в дереве.
* В хеш-таблицах при эффективном хешировании поиск может быть выполнен за постоянное время O(1).