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 **ПРОГРАММНАЯ ИНЖЕНЕРИЯ (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.04** Программная инженерия

О Т Ч Е Т

По лабораторной работе № 7

Название: Деревья, хеш-таблицы

Дисциплина: Типы и структуры данных

Студент ИУ7-32Б Беляк С.С.

Преподаватель Барышникова М.Ю.

Москва, 2023

Описание условия задачи

Используя предыдущую программу (задача №6), сбалансировать полученное дерево. Вывести его на экран в виде дерева. Построить хеш-таблицу из слов текстового файла, задав размерность таблицы с экрана. Осуществить поиск введенного слова 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 задачи, реализуемой в программе

Цель работы - построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска, в хеш-таблицах и в файлах. Сравнить эффективность реструктуризации таблицы для устранения коллизий и поиска в ней с эффективностью поиска в исходной таблице.

Описание ТЗ.

Описание исходных данных

Пользователь выбирает опцию из меню, представленного в консоли.

Возможные варианты выбора:

Меню:

0. Выход
1. Считать дерево из файла
2. Вывести дерево
3. Найти слово в дереве
4. Сбалансировать дерево
5. Построить хеш-таблицу
6. Найти слово в хеш-таблице
7. Вывести хеш-таблицу
8. Оценка эффективности
9. Оценка эффективности реструктуризации

Пользователь может выбрать соответствующую опцию, введя число от 0 до 9 в консоли.

Имеются следующие ограничения:

Ограничение по памяти:

Необходимо обеспечить эффективное использование динамической памяти и предотвратить утечки.

Ограничение по времени:

Время выполнения программы может зависеть от множества факторов, включая объем обрабатываемых данных.

Ограничения на работу с файлами:

Ограничения на открытие и считывание файла могут привести к ошибкам, если файл не существует, недоступен для чтения или содержит данные, не соответствующие ожидаемому формату.

Ограничения бинарного дерева:

Реализация бинарного дерева может сталкиваться с ограничениями при обработке больших объемов данных, что может привести к неэффективной работе.

Ограничения хеш-таблиц:

Размер выделенной памяти для хеш-таблицы может быть ограничен. Это может привести к ограничениям на количество элементов, которые могут быть обработаны программой.

Ограничение по количеству операций ввода-вывода:

Программа взаимодействует с пользователем через стандартный ввод/вывод. Необходимо предусмотреть обработку ошибок при вводе и выводе данных.

Ограничения по вводу параметров:

Пользователь должен обеспечить корректность ввода данных. Программа должна корректно обрабатывать ввод пользователя и предоставлять информацию о возможных ошибках.

Описание результатов программы

0. Выход:

- Завершение работы программы.

1. Считать дерево из файла:

- Загрузка структуры дерева из внешнего файла. Формат файла предполагается заданным, содержащим информацию о структуре дерева.

2. Вывести дерево:

- Отображение содержимого текущего дерева в консольном интерфейсе.

3. Найти слово в дереве:

- Поиск заданного слова в текущем дереве и отображение результата.

4. Сбалансировать дерево:

- Процедура AVL-балансировки дерева с целью оптимизации его структуры и улучшения эффективности операций.

5. Построить хеш-таблицу:

- Создание хеш-таблицы на основе слов из файла. Разрешение коллизий и оптимизация размера таблицы.

6. Найти слово в хеш-таблице:

- Поиск заданного слова в текущей хеш-таблице и отображение результата.

7. Вывести хеш-таблицу:

- Отображение содержимого текущей хеш-таблицы.

8. Оценка эффективности:

- Измерение времени выполнения поиска в несбалансированном, сбалансированном дереве и хеш-таблице, а также объём памяти и количество сравнений при поиске.

9. Оценка эффективности реструктуризации:

- Измерение времени выполнения поиска после процедуры реструктуризации хеш-таблицы и сравнение с исходным временем поиска.

1. Способ обращения к программе

Способ обращения к программе пользователем происходит через исполняемый файл app.exe.

3. Описание возможных аварийных ситуаций и ошибок пользователя

CHOICE_ERROR (Ошибка выбора)

Эта ошибка возникает, когда пользователь вводит некорректное значение пункта меню программы. Например, если пользователь вводит символ или значение, которое не соответствует доступным опциям в меню. Пользователю будет предложено ввести корректное значение из доступных опций.

ALLOCATE (Ошибка выделения памяти)

Если при построении хеш-таблицы возникнет ошибка, например, из-за недостаточной памяти, программа должна корректно обработать ошибку и уведомить пользователя.

FILE_READ_ERROR (Ошибка чтения файла)

Если указанный файл не будет найден или не сможет быть прочитан, программа должна сообщить об ошибке и вернуть пользователя в главное меню.

OVERFLOW (Ошибка переполнения)

Возникает, когда система не может выделить достаточно памяти. В случае ошибки переполнения памяти, программа сообщит об этом пользователю.

5.Описание внутренних структур данных

```
// Структура для узла хеш-таблицы
struct Node_hash
{
    char word[100]; // Предполагается, что максимальная длина слова - 99 символов
    int occupied;   // Флаг, указывающий, занята ли ячейка
    struct Node_hash *next; // Указатель на следующий узел в случае коллизии
};
// Структура для хеш-таблицы
struct HashTable {
    struct Node_hash **table; // Указатель на массив указателей на узлы
    int currentTableSize;     // Текущий размер таблицы
};
// Структура для дерева
typedef struct Node
{
    char *data;          // Данные, хранимые в узле
    struct Node *left;   // Указатель на левое поддерево
    struct Node *right;  // Указатель на правое поддерево
    bool colour;         // Цвет узла
} Node;
```

- **Node:** Структура, представляющая узел дерева. Содержит указатели на левого и правого потомка, слово и цвет.
- **Node_hash:** Структура, представляющая узел в хеш-таблице. Содержит слово и указатель на следующий элемент в случае коллизии.
- **HashTable:** Структура, представляющая хеш-таблицу. Содержит указатель на массив указателей на узлы.

Алгоритм программы

Общий алгоритм программы:

1. **Инициализация:**
   - Создание структур данных: Node для дерева, Node_hash для хеш-таблицы и HashTable для управления хеш-таблицей, инициализация переменных.
2. **Цикл ввода команд пользователя:**
   - Вывод меню с доступными операциями.
   - Считывание введенной команды.
3. **Выполнение операций:**
   - **Считать дерево из файла:**
     - Запрос у пользователя имени файла.
     - Открытие файла и считывание слов в дерево.
   - **Вывести дерево:**
     - Отображение дерева в консоли.
   - **Найти слово в дереве:**
     - Запрос у пользователя слова для поиска.
     - Поиск слова в дереве и вывод соответствующего узла.
   - **Сбалансировать дерево:**
     - Процедура AVL-балансировки дерева.
   - **Построить хеш-таблицу:**
     - Запрос размера хеш-таблицы и имени файла с данными.
     - Создание и заполнение хеш-таблицы.
   - **Найти слово в хеш-таблице:**
     - Запрос у пользователя слова для поиска в хеш-таблице.
     - Поиск слова и вывод результата.
   - **Вывести хеш-таблицу:**
     - Отображение содержимого хеш-таблицы в консоли.

- **Оценка эффективности:**
  - Запрос имени файла с тестовыми данными.
  - Тестирование производительности поиска в сбалансированном, несбалансированном дереве и хеш-таблице.
- **Оценка эффективности реструктуризации (Опция 9):**
  - Проверка наличия хеш-таблицы.
  - Измерение эффективности реструктуризации хеш-таблицы.

4. **Освобождение памяти:**
   - Перед завершением программы освобождение памяти, выделенной для деревьев и хеш-таблицы.
5. **Завершение программы:**
   - Возврат нулевого значения, указывающего на успешное завершение программы.

Основная hash-функция:

Данная функция представляет собой реализацию хеш-функции для строковых ключей. Она принимает в качестве входного параметра указатель на строку const char *word и вычисляет хеш-значение для этой строки. Алгоритм хеширования основан на последовательном применении операции умножения на простое число к текущему хеш-значению и добавлении кода ASCII-символа строки word[i]. Таким образом, каждый символ строки влияет на формирование итогового хеш-значения. Этот метод хеширования прост в реализации и обеспечивает равномерное распределение значений.

Листинг hash-функции:

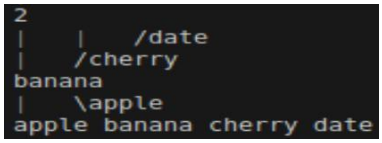
```
unsigned int hash_function(const char *word)
{
    unsigned int hash = 0;
    for (int i = 0; word[i] != '\0'; i++)
    {
        hash = hash * 31 + word[i];
    }
    return hash;
}
```

Пример работы программы:

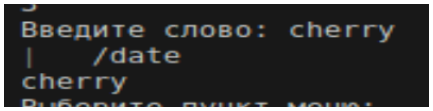
Выведем исходное дерево из файла:

```
2
|   |   |   /date
|   |   /cherry
|   /banana
apple
apple banana cherry date
```

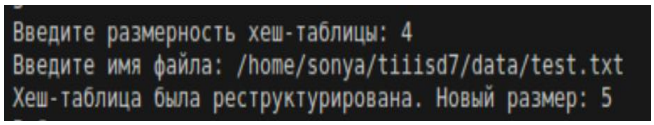
Сбалансируем дерево:



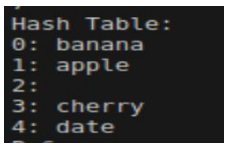
Найдем слово в сбалансированном дереве:



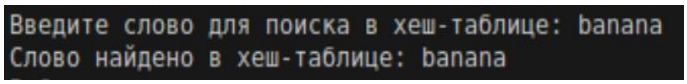
Создадим хеш-таблицу из файла. Введем размерность с консоли. Таблица реструктурирована.



Выведем хеш-таблицу:



Найдем слово в хеш-таблице:



Вывод по алгоритму работающей программы:

Программа реализует работу с набором данных, представленным в виде структуры дерева двоичного поиска и хеш-таблицы. Пользователь может выполнить различные операции, такие как считывание дерева из файла, балансировка дерева, создание хеш-таблицы, поиск узлов по словам, поиск слова в хеш-таблице, вывод дерева, вывод хеш-таблицы. Дополнительно, предусмотрена возможность проведения тестов для оценки эффективности структур.

Эффективность

Количество слов в файле: 10

Реализация	Время поиска (нс)	Количество сравнений	Память
Несбалансированное дерево	46	3	444
Сбалансированное дерево	34	3	444
Хеш - таблица	45	1	1324

Разность производительности относительно хеш-таблицы:

С несбалансированным деревом: 1 нс

С сбалансированным деревом: 11 нс

Хеш таблица быстрее на 3% от несбалансированного дерева.

Объём памяти хеш-таблицы больше объёма памяти дерева в 2.9 раз

Количество слов в файле: 100

Реализация	Время поиска (нс)	Количество сравнений	Память
Несбалансированное дерево	55	6	4249
Сбалансированное дерево	42	5	4249
Хеш - таблица	41	1	12643

Разность производительности относительно хеш-таблицы:

С несбалансированным деревом: 14 нс.

С сбалансированным деревом: 1 нс.

Хеш таблица быстрее на 26% от несбалансированного дерева.

Хеш таблица быстрее на 3% от сбалансированного дерева.

Объём памяти хеш-таблицы больше объёма памяти дерева в 2.9 раз

Количество слов в файле: 1000

Реализация	Время поиска (нс)	Количество сравнений	Память
Несбалансированное дерево	186	19	41535
Сбалансированное дерево	133	18	41535
Хеш - таблица	42	1	121929

Разность производительности относительно хеш-таблицы:

С несбалансированным деревом: 144 нс.

С сбалансированным деревом: 91 нс.

Хеш таблица быстрее на 78% от несбалансированного дерева.

Хеш таблица быстрее на 31% от сбалансированного дерева.

Объём памяти хеш-таблицы больше объёма памяти дерева в 2.9 раз

Количество слов в файле: 10000

Реализация	Время поиска (нс)	Количество сравнений	Память
Несбалансированное дерево	291	14	415928
Сбалансированное дерево	254	13	415928
Хеш - таблица	39	1	1216008

Разность производительности относительно хеш-таблицы:

С несбалансированным деревом: 252 нс.

С сбалансированным деревом: 215 нс.

Хеш таблица быстрее на 87% от несбалансированного дерева.

Хеш таблица быстрее на 85% от сбалансированного дерева.

Объём памяти хеш-таблицы больше объёма памяти дерева в 2.9 раз.

Реструктуризация

Если ключ пытается занять ту же ячейку в таблице при вставке нового значения в хеш-таблицу, мы применяем реструктуризацию методом цепочек. В случае, когда несколько различных значений ключа возвращают одинаковое значение хеш-функции, по этому адресу находится указатель на связанный список, содержащий все соответствующие значения. Поиск в этом списке выполняется путем простого перебора. Реструктуризация происходит путем увеличения размера таблицы на 15% до тех пор, пока не будут включены все слова из файла. Мы находим новое место для хранения ключей таким образом, чтобы каждому ключу соответствовало только одно значение, тем самым минимизируя возможность возникновения коллизий.

Реструктуризация в функции вставки (`insert_into_table`):

```
void insert_into_table(struct HashTable *hashtable, const char *word)
{
    ….
    if (load_factor(hashtable) > 0.75)
    {
        int new_size = hashtable->currentTableSize * GROWTH_FACTOR;
        new_size = (new_size < hashtable->currentTableSize + 1) ? new_size + 1 :
new_size;
        restructure_table(hashtable, new_size);
    }
}
```

Функция реструктуризации restructure_table(struct HashTable *hashtable, int new_size).

```
void restructure_table(struct HashTable *hashtable, int new_size)
{
    struct Node_hash **new_table = (struct Node_hash **)calloc(new_size,
sizeof(struct Node_hash *));
    if (new_table == NULL)
        exit(EXIT_FAILURE);
    for (int i = 0; i < hashtable->currentTableSize; i++)
    {
        struct Node_hash *current = hashtable->table[i];
        while (current != NULL)
        {
            struct Node_hash *next = current->next;
            unsigned int new_index = hash_function(current->word) % new_size;
            while (new_table[new_index] != NULL)
            {
                new_index = (new_index + 1) % new_size;
            }
            current->next = NULL;
            new_table[new_index] = current;

            current = next;
        }
    }
    free(hashtable->table);
    hashtable->table = new_table;
    hashtable->currentTableSize = new_size;
    printf("Хеш-таблица была реструктурирована. Новый размер: %d\n", new_size);
}
```

Если количество сравнений в поиске слова превышает 3 и 4, то появляется информационное сообщение о необходимости реструктуризации:

```
struct Node_hash *search_in_table(const struct HashTable *hashtable, const char *word, int
*comparisons)
{
    unsigned int index = hash_function(word) % hashtable->currentTableSize;
    int originalIndex = index;
    *comparisons = 0;
    while (hashtable->table[index] != NULL && hashtable->table[index]->occupied)
    {
        (*comparisons)++;
        if (strcmp(hashtable->table[index]->word, word) == 0)
        {
            if (*comparisons > 3)
                printf("Предупреждение: Количество сравнений при поиске элемента больше
3\n");
            return hashtable->table[index];
        index = (index + 1) % hashtable->currentTableSize;
        if (index == originalIndex)
```

```
            break;
    }
    if (*comparisons > 3)
        printf("Предупреждение: Количество сравнений при поиске элемента больше 3\n");
    return NULL;
}
```

Пример реструктуризации:

```
Введите размерность хеш-таблицы: 4
Введите имя файла: /home/sonya/tiiisd7/data/test.txt
Хеш-таблица была реструктурирована. Новый размер: 5
```

Проведем сравнение эффективности реструктуризации:

```
Хеш-таблица была реструктурирована. Новый размер: 150
Исходная таблица: Время поиска - 33 нс, Сравнения - 1
После реструктуризации: Время поиска - 23 нс, Сравнения - 1
Эффективность реструктуризации: 30.303030% ускорения в сравнен
ии с исходной таблицей
Выберите пункт меню:
| Пункт |               Действие                        |
|-------|-----------------------------------------------|
```

По полученным данным, эффективность времени реструктуризации составляет около 30%, однако количество сравнений совпадает, поиск выполнен за постоянное время O (1).

Вывод по реструктуризации: Реструктуризация хеш-таблицы с использованием метода цепочек предоставляет эффективный механизм для улучшения производительности при работе с большими объемами данных и эффективным управлением коллизиями.

Сравнение структур:

Для решения задачи поиска и хранения данных, целесообразно использовать различные структуры данных в зависимости от требований к операциям и объему данных.

Реализация	**Преимущества**	**Недостатки**	**Случаи применения**
Несбалансированное дерево	Простота реализации.	В случае несбалансированного дерева, операции могут иметь логарифмическую сложность, что может привести к неэффективной работе.	При небольшом объеме данных, когда не требуется частая балансировка.
Сбалансированное дерево	Гарантированная логарифмическая сложность поиска	На некоторые операции может быть небольшая дополнительная	При необходимости эффективного поиска и поддержания структуры в балансе

	Автоматическая балансировка	сложность из-за балансировки.	
Хеш - таблица	Константная сложность поиска в среднем случае. Эффективность при больших объемах данных.	Могут возникнуть коллизии, требующие дополнительных операций. Не гарантирована порядковая структура.	При работе с большим объемом данных и приоритете быстрого поиска.

Для подтверждения выбора структуры данных и сравнения их эффективности, были проведены тесты с использованием различных объемов данных и операций. Тесты показали, что хеш-таблицы быстрее деревьев на больших данных, однако занимают больший объём памяти в 2.9 раз. При малом количестве данных, по времени выигрывают сбалансированные деревья и хеш-таблицы. Сбалансированные деревья всегда быстрее, чем несбалансированные, т.к. дерево при поиске слова уже сбалансировано, следовательно, совершает меньшее количество сравнений, чем несбалансированное дерево.

Самое меньшее количество сравнений у хеш-таблиц, составляет 1 сравнение, это говорит об высокой эффективности поиска слово в хеш-таблицах. Если хеш-таблица превышает 3,4 сравнения, таблица реструктуризируется. Структуры сбалансированного дерева или хеш-таблицы наиболее целесообразны для решения поставленной задачи.

Особое внимание при тестировании следует уделить следующим моментам:

- Реструктуризация хеш-таблицы:
- Определение момента, когда реструктуризация становится целесообразной
- Предусмотреть вывод сообщения при необходимости реструктуризации хеш-таблицы;

Контрольные вопросы

1. Идеально сбалансированное дерево и AVL-дерево:

Идеально сбалансированное дерево (Perfectly Balanced Tree): Каждый уровень дерева полностью заполнен узлами, и высота дерева минимальна. Такие деревья обеспечивают оптимальное время выполнения операций, но в практике редко встречаются из-за ограничений на количество элементов в дереве.

AVL-дерево: это форма сбалансированного дерева двоичного поиска, в котором разница в высоте между левым и правым поддеревьями для каждого узла ограничена (высота различается не более чем на 1). Это обеспечивает быстрое выполнение операций вставки, удаления и поиска.

2. Поиск в AVL-дереве и дереве двоичного поиска:

В AVL-дереве поиск выполняется так же, как и в обычном дереве двоичного поиска. Разница заключается в том, что AVL-дерево поддерживает балансировку после каждой операции вставки или удаления, чтобы сохранять свою структуру сбалансированной.

3.Хеш-таблица и еѐ принцип построения:

Хеш-таблица — это структура данных, позволяющая эффективно выполнять операции вставки, удаления и поиска. Она использует хеш-функцию для преобразования ключа в индекс массива, где хранятся значения.

Принцип построения: Выбор хеш-функции. Выделение массива определенного размера.Разрешение коллизий (в случае, если два ключа хешируются в один и тот же индекс).

4. Коллизии и методы их устранения:

Коллизии возникают, когда два различных ключа хешируются в один и тот же индекс. Методы разрешения коллизий включают:

- Цепочки: Каждый индекс массива представляет собой связанный список.
- Открытое хеширование: при коллизии производится поиск следующего свободного слота в массиве.
- Двойное хеширование: используются две хеш-функции для определения следующего индекса при коллизии.

5. Неэффективность поиска в хеш-таблицах:

Поиск в хеш-таблицах становится неэффективным при большом количестве коллизий, что может привести к увеличению длины цепочек или увеличению размера открытого адреса.

6. Эффективность поиска:

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

Вывод

В ходе выполнения лабораторной работы была разработана программа для работы с деревьями и хеш-таблицей, предоставляющая пользователю удобный интерфейс для взаимодействия с данными. В результате тестирования эффективности программы на различных объемах данных (10, 100, 1000, 10000 слов), были получены временные показатели, сравнение по памяти, сравнение по количеству сравнений. Согласно результатам тестов, программа, использующая сбалансированное дерево, демонстрирует высокую временную эффективность по сравнению с несбалансированным деревом. Однако на больших размерах данных высокую эффективность по времени показывают хеш-таблицы, но объём памяти хеш-таблиц значительно превышает объем

памяти деревьев. В целом, разработанная программа успешно решает поставленную задачу и предоставляет эффективные средства для работы с бинарным деревом поиска, с балансированным AVL деревом и с хеш-таблицей.

Ответы на контрольные вопросы

1. Что такое дерево? Как выделяется память под представление деревьев?

Дерево - структура данных, состоящая из узлов, связанных между собой рёбрами. Один из узлов называется корнем, остальные разделяются на узлы и листья. Узлы, соединенные ребрами, образуют поддеревья. Память под представление деревьев обычно выделяется динамически. Каждый узел дерева содержит информацию и указатели на своих потомков (или нулевые указатели, если потомков нет). Для каждого узла память выделяется отдельно при добавлении новых узлов.

2. Какие бывают типы деревьев?

Дерево двоичное: Каждый узел имеет не более двух потомков.

Дерево двоичного поиска: Узлы упорядочены так, что для каждого узла все узлы в его левом поддереве меньше его, а в правом — больше.

N-арное дерево: Каждый узел может иметь произвольное количество потомков.

Распределенное дерево: используется в распределенных вычислениях и сетевых структурах.

AVL-дерево, красно-черное дерево: Сбалансированные бинарные деревья для эффективного поиска.

3. Какие стандартные операции возможны над деревьями?

Стандартные операции над деревьями включают:

Добавление узла: Вставка нового узла в дерево.

Удаление узла: Удаление существующего узла из дерева.

Поиск узла: Нахождение узла с определенным значением.

Обход дерева: Посещение всех узлов дерева в определенном порядке (прямой, обратный, симметричный).

Вывод дерева в виде строки: Представление дерева в текстовой или графической форме.

Изменение данных узла: Обновление значений в существующем узле.

4. Что такое дерево двоичного поиска?

Дерево двоичного поиска - бинарное дерево, в котором каждый узел имеет не более двух потомков. При этом для каждого узла выполнено следующее свойство: все узлы в левом поддереве меньше текущего узла, а все узлы в правом поддереве больше текущего узла. Это свойство делает дерево двоичного поиска эффективной структурой данных для поиска, вставки и удаления элементов, так как оно обеспечивает логарифмическую сложность этих операций в среднем случае.