Отчет по 6 лабораторной работе

По дисциплине «Типы и структуры данных»

Подготовил Жабин Дмитрий

Группа ИУ7-34Б

Вариант 2

**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.

**Задание**

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. 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.

**Внешняя спецификация программы.**

**Входные данные:** имя текстового файла со словами (строка), максимально допустимое количество коллизий (целое число), слово для поиска (строка).

**Выходные данные:** псевдографическое изображение ДДП и АВЛ деревьев, хеш-таблица, информация для сравнения эффективности используемых структур данных по времени и памяти.

**Задача программы:**

Вывод ДДП, АВЛ-дерева и хеш-таблицы, вывод информации об эффективности используемых структур данных по времени и памяти.

**Способ обращения к программе:** Запуск программы в консоли с аргументом в виде имени файла со словами (./app.exe filename.txt).

**Аварийные ситуации**

1) Некорректный ввод имени файла.

2) Пустой файл в качестве аргумента программы.

3) Некорректный ввод максимально допустимого числа коллизий.

Во всех указанных случаях программа завершает работу с соответствующим сообщением.

**Структуры данных.**

Реализация узла линейного односвязного списка:

typedef struct list

{

char \*value; //значение узла (строка)

struct list \*next; // указатель на следующий элемент списка

} list\_t;

Реализация узла дерева:

typedef struct tree\_node

{

char \*val; //значение узла (строка)

struct tree\_node \*left; // указатель на левого потомка

struct tree\_node \*right; // указатель на правого потомка

} tree\_node;

Реализация динамического массива (использовался при балансировке дерева):

typedef struct

{

tree\_node \*\*arr; // массив указателей на узлы дерева

int size; // фактический размер массива

int mem\_size; // аллоцированный размер массива

} dynarr\_t;

**Функции**

char \*\*fill\_words(FILE \*f); - функция чтения слов из файла в массив

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

int search\_hashtable(char \*to\_find, list\_t \*hash\_list, int n, int (\*hash)(char \*, int)); - функция поиска слова в хеш-таблице

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

tree\_node \*create\_node(char \*val); - функция создания узла дерева

tree\_node \*insert(tree\_node \*node, char \*val); - функция вставки узла дерева

void fill\_tree(tree\_node \*\*root, FILE \*f); - функция заполнения дерева

void print\_tree(tree\_node \*root, int space); - функция вывода дерева на экран

tree\_node \*build\_tree(dynarr\_t \*arr, int start, int end); - функция построения сбалансированного дерева

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

int simple\_hash(char \*s, int n); - простая хеш-функция

int compl\_hash(char \*s, int n); - улучшенная хеш-функция

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

void print\_hash\_table(list\_t \*arr, int size); -функция вывода хеш-таблицы на экран

**Алгоритм**

В данной реализации для построения сбалансированного дерева указатели на узлы ДДП добавляются в отсортированный массив, затем по нему рекурсивно строится АВЛ-дерево (массив многократно “делится пополам” для взятия очередного узла).

После этого по данным строится хеш-таблица с помощью простой хеш-функции (возвращает остаток от деления суммы кодов символов строки на размер таблицы), изначальный размер таблицы выбирается как наименьшее простое число, превосходящее количество слов в файле. Используется метод открытого хеширования - элементы с равными хеш-значениями связываются в цепочку-список. В случае, если максимальное количество коллизий превосходит допустимое, таблица реструктуризируется на основе улучшенной хеш-функции (в ней добавлен побитовый сдвиг). При этом размер таблицы увеличивается до каждого следующего простого числа, пока условие максимально допустимого количества коллизий не будет выполнено.

Поиск элемента в дереве начинается с корня дерева. Значение искомого элемента сравнивается со значением в очередном узле, если значение в узле больше, то поиск продолжается в его левом поддереве, если меньше - в правом поддереве. Процесс продолжается до тех пор, пока элемент не будет найден или же будет достигнут узел, не имеющий нужного потомка.

Для поиска элемента в хеш-таблице с помощью хеш-функции вычисляется его хеш-значение (индекс в таблице), и значение элемента поочередно сравнивается со значениями элементов цепочки-списка, находящейся по этому индексу в таблице.

При поиске в файле значение искомого элемента сравнивается со значениями элементов в файле в порядке их чтения. Процесс продолжается до тех пор, пока элемент не будет найден или же будет достигнут конец файла.

**Правильный алгоритм построения сбалансированного дерева**

В структуре для хранения узла АВЛ-дерева добавляется показатель сбалансированности узла - разность высот его правого и левого поддеревьев.

Построение АВЛ-дерева по указанным данным происходит за счет балансировки его узлов (в случае нарушения сбалансированности) при включении очередного узла.

**Процесс включения нового узла состоит из следующих задач**:

1) проход по пути поиска (чтобы убедиться, что элемента с таким ключом в дереве нет);

2) включение узла и определение показателя сбалансированности;

3) возврат по пути поиска, с проверкой показателей сбалансированности для каждой вершины, и если необходимо, то проведение балансировки.

(показатель сбалансированности каждой вершины, пройденной при поиске места включения нового узла, корректируется: уменьшается на 1, если мы спускались в её левое поддерево, увеличивается на 1 - если в правое)

**Балансировка вершины (возможны 2 случая):**

1) если **показатель сбалансированности вершины равен 2**, то:

1.1. Если показатель сбалансированности правого потомка данной вершины отрицателен, то осуществляется большое левое вращение - сначала малое правое вращение правого потомка, затем малое левое вращение самой вершины

1.2. Если же показатель сбалансированности правого потомка данной вершины неотрицателен, то осуществляется малое левое вращение самой вершины

2) если **показатель сбалансированности вершины равен -2**, то:

2.1. Если показатель сбалансированности левого потомка данной вершины положителен, то осуществляется большое правое вращение - сначала малое левое вращение левого потомка, затем малое правое вращение самой вершины

2.2. Если же показатель сбалансированности левого потомка данной вершины неположителен, то осуществляется малое правое вращение самой вершины

**При малом левом вращении вершины** на место данной вершины встает её правый потомок, новым правым потомком данной вершины становится левый потомок её старого правого потомка, а сама вершина становится новым левым потомком её старого правого потомка. При этом обновляются показатели сбалансированности данной вершины и вершины-её старого правого потомка в соответствии с таблицей:

+1,-1,-2,+0

+1,+0,-1,+0

+1,+1,-1,-1

+2,+0,-1,+1

+2,+1,+0,+0

+2,+2,+0,-1

где 1 и 4 столбец содержат значения показателя сбалансированности данной вершины до и после поворота соответственно, а 2 и 3 - значения показателя сбалансированности вершины-её старого правого потомка до и после поворота соответственно.

**При малом правом вращении вершины** на место данной вершины встает её левый потомок, новым левым потомком данной вершины становится правый потомок её старого левого потомка, а сама вершина становится новым правым потомком её старого левого потомка. При этом обновляются показатели сбалансированности данной вершины и вершины-её старого левого потомка в соответствии с таблицей:

-1,-1,+1,+1

-1,+0,+1,+0

-1,+1,+2,+0

-2,-1,+0,+0

-2,-2,+0,+1

-2,+0,+1,-1

где 1 и 4 столбец содержат значения показателя сбалансированности данной вершины до и после поворота соответственно, а 2 и 3 - значения показателя сбалансированности вершины-её старого левого потомка до и после поворота соответственно.

Использованный в программе алгоритм хуже вышеописанного, так как время построения АВЛ-дерева в данном случае складывается из времен построения ДДП, помещения указателей на его узлы в массив в нужном порядке, рекурсивного построения дерева по данным из массива. К тому же под этот массив требуется дополнительная память.

**Тесты**

|  |  |  |
| --- | --- | --- |
| Тест | Ввод пользователя | Результат |
| Некорректный ввод имени файла | tree0.txt (файл не существует) | Неверное имя файла! |
| Пустой файл | empty.txt (файл пуст) | Файл пуст! |
| Некорректный ввод максимального числа коллизий | 0/abc/-5 | Введено некорректное значение! |
| Поиск несуществующего слова | Hello (такого слова нет в дереве) | Слово "Hello" не найдено. |
| Ввод количества коллизий, большего, чем текущее максимальное | 3 (при текущем максимальном 2) | Максимальное количество коллизий не больше введенного.  Реструктуризация таблицы не требуется. |
| Ввод количества коллизий, меньшего, чем текущее максимальное | 1 (при текущем максимальном 2) | Реструктуризация  хеш-таблицы |
| Корректный ввод всех характеристик | Корректный файл, корректный ввод числа коллизий | Информация об эффективности использования структур данных |

**Оценка эффективности**

Поиск слова (в тактах процессора):

Указаны средние значения для поиска по всей структуре данных.

При поиске рассматривались хеш-таблицы без коллизий.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество узлов дерева | ДДП | АВЛ-дерево | Хеш-таблица | Файл |
| 20 | 399 | 357 | 355 | 13628 |
| 50 | 528 | 390 | 372 | 16101 |
| 100 | 552 | 471 | 409 | 24033 |
| 500 | 1019 | 605 | 460 | 81688 |

Среднее количество сравнений при поиске элемента

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество узлов дерева | ДДП | АВЛ-дерево | Хеш-таблица | Файл |
| 20 | 4.3 | 3.7 | 1 | 10.5 |
| 50 | 7 | 4.86 | 1 | 25.5 |
| 100 | 7.04 | 5.8 | 1 | 50.5 |
| 500 | 10.898 | 7.996 | 1 | 250.5 |

Объем занимаемой памяти (в байтах):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество узлов дерева | ДДП | АВЛ-дерево | Хеш-таблица  (до/после реструктуризации) | Файл |
| 20 | 480 | 656 | 548/1624 | 160 |
| 50 | 1200 | 1616 | 1236/5016 | 428 |
| 100 | 2400 | 3216 | 2582/27864 | 831 |
| 500 | 12000 | 16016 | 15474/348376 | 4236 |

**Вывод**

Использование хеш-таблицы особенно эффективно по времени при отсутствии коллизий, но для этого зачастую требуется реструктуризация таблицы с увеличением её размера, из-за чего хеш-таблица начинает проигрывать другим структурам данных по памяти (требуется память под каждый хеш, а при полном устранении коллизий многие из них не используются). Использование файла, наоборот, наиболее эффективно по памяти и наименее эффективно по времени. Если учитывать эффективность по памяти и по времени вместе, то более оптимальной структурой данных для данной задачи оказались деревья. АВЛ дерево не всегда выигрывает по времени поиска (конкретного элемента) у несбалансированного дерева, так как порядок вершин при балансировке меняется, но всегда выигрывает по среднему количеству сравнений и среднему времени поиска.

**Контрольные вопросы**

**1. Что такое дерево?**

Дерево – это нелинейная 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?**

Дерево двоичного поиска – это такое дерево, в котором все левые потомки моложе предка, а все правые – старше.

**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 количество вершин в каждом поддереве различается не более чем на 1.

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

Поиск в АВЛ дереве в среднем происходит быстрее, чем в ДДП, т.к. путь до некоторых узлов оказывается короче.

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

Хеш-таблица — это 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: открытое и закрытое хеширование. При открытом хешировании к ячейке по данному ключу прибавляется связный 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.

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

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