|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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» (ИУ) |
| Кафедра | «Программное обеспечение ЭВМ и информационные технологии» (ИУ7) |

**Лабораторная работа №7**

**“Графы”**

**Вариант №8**

|  |  |
| --- | --- |
| Студент: Князев Дмитрий Юрьевич, группа ИУ7-33Б | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись, дата) |
| Преподаватель:  Барышникова Марина Юрьевна | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись, дата) |

Оглавление

Описание задачи 3

Входные и выходные данные 4

Аварийные ситуации и особенности реализации 5

Описание структур данных 6

Функциональные тесты 11

Сравнение эффективности 18

Вывод 21

Контрольные вопросы 22

**Цель работы:**

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.

**Задание**:

Построить хеш-таблицу по указанным данным. Сравнить  
эффективность поиска в сбалансированном двоичном дереве, в двоичном дереве поиска и в хеш-таблице (используя открытую и закрытую адресацию). Вывести на экран деревья и хеш-таблицу. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если среднее количество сравнений больше указанного. Оценить эффективность использования этих структур (по времени и памяти) для поставленной задачи. Оценить эффективность поиска в хеш-таблице при различном количестве коллизий и при различных методах их разрешения.

**Задание по варианту:**

Используя предыдущую программу (задача №6), сбалансировать полученное дерево. Вывести его на экран в виде дерева. Построить хеш-таблицу из чисел файла. Осуществить поиск введённого целого числа в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Сравнить время поиска, объем памяти и количество сравнений при использовании различных структур данных.

**Входные данные:**

* Пункт меню (число от 0 до 6 включительно)
* В зависимости от выбранного пункта меню:
  + Файл с последовательностью целых чисел
  + Целое число

**Выходные данные (в зависимости от выбранного пункта меню):**

* Графическое изображение АВЛ дерева или ДДП в формате .svg файла в отдельном окне путём вызова xdg-open
* Содержимое хеш-таблиц в символьном представлении
* Информация об успешном или неудачном поиске, добавлении или удалении элемента в ДДП, АВЛ дереве, хеш-таблицах и массиве, о загрузке элементов из файла в АВЛ дерево, ДДП и в хеш-таблицы
* Результаты сравнения эффективности алгоритмов сортировки и поиска с использованием ДДП, АВЛ дерева, хеш-таблиц и массива

**Функции меню**

0. Выход

1. Загрузить числа из файла в АВЛ дерево, в ДДП и в хэш-таблицы

2. Вывести ДДП

3. Вывести АВЛ дерево

4. Вывести хэш-таблицы

5. Добавить элемент в ДДП, АВЛ дерево, хэш-таблицу и в файл

6. Удалить элемент из ДДП, АВЛ дерева, хэш-таблицы и из файла

7. Найти элемент в АВЛ дереве, в ДДП, в хэш-таблицах и в файле

8. Сравнить эффективность алгоритмов сортировки и поиска с использованием ДДП, АВЛ дерева и хэш-таблиц

Оценка эффективности производится в последнем пункте меню, предыдуще используются лишь для проверки работоспособности структур данных.

**Аварийные ситуации:**

1. Считываемый файл кроме чисел и переносов строк содержит другие символы

**Особенности реализации:**

1. Если файл содержит несколько одинаковых значений, они будут перезаписаны

2. Имена используемых файлов находятся в заголовочных файлах:

|  |
| --- |
| #define DATABASE\_FILENAME "database/numbers.txt" |

В файле presets.h

|  |
| --- |
| #define ITERATIONS 10  #define SMALL\_FILE\_NAME "database/numbers\_small.txt"  #define MEDIUM\_FILE\_NAME "database/numbers\_medium.txt"  #define BIG\_FILE\_NAME "database/numbers\_big.txt"  #define SORTED\_FILE\_NAME "database/numbers\_sorted.txt"  #define SORTED\_REVERSE\_FILE\_NAME "database/numbers\_sorted\_reverse.txt" |

В файле efficiency.h

**Описание структур данных:**

|  |
| --- |
| **typedef** **struct** tree\_node  {  **int** value;  **unsigned** **char** height;  **struct** tree\_node \*left;  **struct** tree\_node \*right;  } **tree\_node\_t**; |

Структура узла дерева

value — значение узла, информация пользователя дерева

height — высота узла, игнорируется в функциях для работы с деревом двоичного поиска

left — указатель на левое поддерево

right — указатель на левое поддерево

|  |
| --- |
| **typedef** **struct** hash\_node  {  **int** value;  **struct** hash\_node \*next;  } **hash\_node\_t**;  **struct** hashtable\_opened  {  **hash\_node\_t** \*\*data;  **bool** \*exists;  **size\_t** records\_amount;  **size\_t** max\_size;  };  **struct** hashtable\_closed  {  **int** \*data;  **bool** \*exists;  **size\_t** records\_amount;  **size\_t** max\_size;  }; |

**Структуры хеш-таблиц**

struct hash\_node — структура записи хеш-таблицы с закрытой адресацией

value — значение записи

next — указатель на следующую запись с таким же хешем

struct hashtable\_opened — структура хеш-таблицы с закрытой адресацией

data — указатель на массив записей таблицы (узлы односвязного списка)

struct hashtable\_closed — структура хеш-таблицы с открытой адресацией

data — указатель на массив значений таблицы (целые числа)

exists — указатель на массив ключей, определяющих существование элемента в таблице

records\_amount — количество записей в таблице

max\_size — размер таблицы

|  |
| --- |
| **typedef** **struct** array  {  **int** \*data;  **size\_t** size;  **size\_t** max\_size;  } **array\_t**; |

Структура массива чисел, используемая для обработки файла

data — указатель на динамический массив, хранящий числа из файла

size — текущий размер массива

max\_size — максимальный размер массива

**Константы, связанные с хешированием**

|  |
| --- |
| #define HASH\_OPENED\_COEFFICIENT 0.72  #define HASH\_CLOSED\_COEFFICIENT 1.5  #define HASHTABLE\_STEP\_DIVISION 10 |

В файле hashtable.h

Процесс выделения памяти заключается в нахождении количества элементов N, которые будут храниться в таблице. Затем N умножается на HASH\_OPENED\_COEFFICIENT в случае хеш-таблицы с открытым хешированием и на HASH\_CLOSED\_COEFFICIENT в случае хеш-таблицы с закрытым хешированием. Затем ищется простое число, превышающее полученное, которое будет размером новой хеш-таблицы, измеряемой в записях, к котором можно обратиться за O(1).

**Хеш-функции**

|  |
| --- |
| **size\_t**  **get\_hash\_opened**(**int** key,  **size\_t** max\_size)  {  **return** key % max\_size;  } |

Хеш-функция, используемая хеш-таблицей с закрытым хешированием, где ключом является сам элемент (целое число).

|  |
| --- |
| **size\_t**  **get\_hash\_closed**(**int** key,  **size\_t** max\_size,  **size\_t** step,  **size\_t** iterations)  {  **return** (key + iterations \* step) % max\_size;  } |

Хеш-функция, используемая хеш-таблицей с открытым хешированием

Параметр step определяет шаг, с которым происходит смещение элементов с одинаковым хешем (от хеш-функции указанной выше) в таблице. В качестве этого аргумента передаётся размер таблицы (поле max\_size структуры hashtable\_closed), делённое на коэффициент HASHTABLE\_STEP\_DIVISION, равный 10. Таким образом достигается равномерность распределения ключей в таблице.

**Реализация подсчёта сравнений**

Листинг кода для подсчёта количества сравнений на примере открытой хеш-таблицы (аналогичный код есть и в остальных структурах данных):

|  |
| --- |
| **size\_t hashtable\_comparisons;**  **size\_t**  **get\_hashtable\_comparisons**(**void**)  {  **return** hashtable\_comparisons;  }  **void**  **set\_hashtable\_comparisons**(**size\_t** new\_comparisons)  {  hashtable\_comparisons = new\_comparisons;  }  **int**  **hashtable\_opened\_find**(**hashtable\_opened\_t** \*hashtable,  **int** value)  {  **size\_t** hash;  **hash\_node\_t** \*curr\_node;  hash = get\_hash\_opened(value, hashtable->max\_size);  curr\_node = hashtable->data[hash];  **while** (curr\_node != NULL)  {  **hashtable\_comparisons++;**  **if** (curr\_node->value == value)  {  **return** EXIT\_SUCCESS;  }  curr\_node = curr\_node->next;  }  **return** ERR\_NOT\_FOUND;  } |

Здесь hashtable\_comparisons — статическая переменная, хранящая накопленное количество сравнений, она изменяется функцией поиска элемента в том же модуле, где и определена. Также её можно изменить и прочесть во внешних модулях с помощью функций get\_hashtable\_comparisons и set\_hashtable\_comparisons.

**Функциональные тесты**

|  |  |
| --- | --- |
| Ввод | Вывод |
| 1. Загрузить числа из файла в АВЛ дерево, в ДДП и в хеш-таблицы | Очистка АВЛ дерева...  Очистка ДДП...  Очистка открытой хеш-таблицы...  Очистка закрытой хеш-таблицы...  Открытие файла на чтение...  Запись в АВЛ дерево...  Запись в ДДП...  Запись в открытую хеш-таблицу...  Запись в закрытую хеш-таблицу...  Закрытие файла... |
| 2. Вывести ДДП  (После загрузки элементов из файла в структуры) |  |
| 3. Вывести АВЛ дерево  (После загрузки элементов из файла в структуры) |  |
| 4. Вывести хеш-таблицы  (После загрузки элементов из файла в структуры) |  |
| 5. Найти элемент в АВЛ дереве, в ДДП, в хеш-таблицах и в файле  (После загрузки элементов из файла в структуры)  Поиск числа 76, которое есть в файле  (Максимальное среднее количество сравнений равно 2) | Поиск элемента в файле...  Элемент найден: 76  Количество сравнений: 5  Поиск элемента в АВЛ дереве...  Элемент найден: 76  Количество сравнений: 5  Поиск элемента в двоичном дереве поиска...  Элемент найден: 76  Количество сравнений: 5  Поиск элемента в открытой хэш-таблице...  Элемент найден: 76  Количество сравнений: 1  Поиск элемента в закрытой хэш-таблице...  Элемент найден: 76  Количество сравнений: 2 |
| 0. Выход | Осуществляется выход из программы |
| Вставка элемента, который находится в файле, но не в остальных структурах | Вставка элемента в АВЛ дерево...  Успешно  Вставка элемента в ДДП...  Успешно  Вставка элемента в открытую хэш-таблицу...  Успешно  Вставка элемента в закрытую хэш-таблицу...  Успешно  Вставка элемента в файл...  Элемент уже находится в файле |
| Вставка элемента, который уже находится в файле и структурах | Вставка элемента в АВЛ дерево...  Элемент уже находится в АВЛ дереве  Вставка элемента в ДДП...  Элемент уже находится в ДДП  Вставка элемента в открытую хэш-таблицу...  Элемент уже находится в открытой хеш-таблице  Вставка элемента в закрытую хэш-таблицу...  Элемент уже находится в закрытой хеш-таблице  Вставка элемента в файл...  Элемент уже находится в файле |
| Удаление элемента, который находится во всех структурах | Удаление элемента из АВЛ дерева...  Успешно  Удаление элемента из ДДП...  Успешно  Удаление элемента из открытой хэш-таблицы...  Успешно  Удаление элемента из закрытой хэш-таблицы...  Успешно  Удаление элемента из файла...  Успешно |
| Удаление элемента, который не находится не в одной из структур | Удаление элемента из АВЛ дерева...  Элемент не найден в АВЛ дереве  Удаление элемента из ДДП...  Элемент не найден в ДДП  Удаление элемента из открытой хэш-таблицы...  Элемент не найден в открытой хэш-таблице  Удаление элемента из закрытой хэш-таблицы...  Элемент не найден в закрытой хэш-таблице  Удаление элемента из файла...  Элемент не найден в файле |
| Вывод хеш-таблиц до и после удаления элемента 72 |  |
| 2. Вывести ДДП  (До выполнения первого пункта меню) | Преобразование дерева в .dot файл...  Дерево не имеет ни одного узла |
| 3. Вывести АВЛ дерево  (До выполнения первого пункта меню) | Преобразование дерева в .dot файл...  Дерево не имеет ни одного узла |
| 4. Вывести хеш-таблицы  (До выполнения первого пункта меню) | Открытая хеш-таблица пуста  Закрытая хеш-таблица пуста |
| 5. Найти элемент в АВЛ дереве, в ДДП, в хеш-таблицах и в файле  (До выполнения первого пункта меню)  Поиск числа 123, которое есть в файле (из 12-ти элементов) | Поиск элемента в файле...  Элемент найден: 123  Количество сравнений: 12  Поиск элемента в АВЛ дереве...  Элемент не найден  Количество сравнений: 1  Поиск элемента в двоичном дереве поиска...  Элемент не найден  Количество сравнений: 1  Поиск элемента в открытой хэш-таблице...  Элемент не найден  Количество сравнений: 1  Поиск элемента в закрытой хэш-таблице...  Элемент не найден  Количество сравнений: 1 |

**Сравнение эффективности (на основе десяти замеров)**:

При каждом замере для поиска использовались случайные числа, находящиеся в соответствующих файлах.

Максимальное значение сравнений в данном сравнении, после которого происходит реструктуризация: **2,0**.

**Поиск (указано среднее количество сравнений)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Структура данных | Маленький файл (100 элементов) | Средний файл (500 элементов) | Большой файл (1000 элементов) | Отсортированный файл (500 элементов) | Файл, отсортированный в обратном порядке (500 элементов) |
| АВЛ дерево | 5.7 | 8.4 | 9.2 | 7.8 | 7.3 |
| ДДП | 6.3 | 14.9 | 8.9 | 284.7 | 217.3 |
| Массив | 48.3 | 203.4 | 346.5 | 284.7 | 217.3 |
| Открытая хеш-таблица | 1.3 | 1.0 | 1.4 | 1.1 | 1.0 |
| Закрытая хеш-таблица | 1.4 | 2.3 | 1.0 | 4.2 | 1.0 |
| Открытая хеш-таблица (реструктуризированная) | - | - | - | - | - |
| Закрытая хеш-таблица (реструктуризированная) | - | 1.0 | - | 1.0 | - |

**Поиск**

Так как время поиска прямо пропорционально количеству сравнений, проанализируем количество сравнений.

На файлах без с маленькой упорядоченностью данных ДДП и АВЛ деревья занимают примерно одинаковое количество сравнений. Однако в отсортированных файлах лучше себя показало АВЛ дерево, выигрывая до 37 раз по количеству сравнений у ДДП, которое по эффективности стало в точности как массив. В каждом случае деревья проигрывают хеш-таблицам в десятки (АВЛ дерево) и сотни (ДДП) раз.

Таблица с закрытой адресацией в среднем потребовала не больше 1,4 сравнений.

Таблица с открытой адресацией на некоторых файлах заняла большее количество сравнений, чем таблица с закрытой адресацией. Решить эту проблему помогла реструктуризация второй таблицы с увеличением её размера вдвое, поиск в таблице с увеличенным размером стал занимать в среднем 1 сравнение.

**Занимаемая память (в байтах)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Структура данных | Маленький файл (100 элементов) | Средний файл (500 элементов) | Большой файл (1000 элементов) | Отсортированный файл (500 элементов) | Файл, отсортированный в обратном порядке (500 элементов) |
| АВЛ дерево | 2400 | 12000 | 24000 | 12000 | 12000 |
| ДДП | 2400 | 12000 | 24000 | 12000 | 12000 |
| Массив | 424 | 2024 | 4024 | 2024 | 2024 |
| Открытая хеш-таблица | 2289 | 12455 | 26031 | 12455 | 12455 |
| Закрытая хеш-таблица | 787 | 3787 | 7587 | 3787 | 3787 |
| Открытая хеш-таблица (реструктуризированная) | - | - | - | - | - |
| Закрытая хеш-таблица (реструктуризированная) | - | 7587 | - | 7587 | - |

**Память**

Больше всего места заняла хеш-таблица с закрытой адресацией. Она занимает на 8% больше памяти, чем двоичное дерево поиска. Самой эффективной по памяти структурой данных является массив, так как он не требует никаких дополнительных полей, кроме, возможно, своего размера. Хеш-таблица с закрытой адресацией проигрывает массиву по памяти примерно в 6,5 раз.

Однако результаты намного лучше для хеш-таблицы с открытой адресацией. Она занимает примерно в 1,9 раз больше места, чем массив и примерно в 3,1 раза меньше места, чем двоичное дерево поиска.

Не потребовалась реструктуризация таблицы с закрытым хешированием, так как среднее количество сравнений при поиске не превысило 2,0.

Даже реструктуризированные таблицы с открытой адресацией занимают в среднем в 2 раза меньший общем памяти, чем таблицы с закрытой адресацией

**Вывод**

Хеш-таблица с закрытой адресацией в среднем показала себя лучше, чем таблица с открытой адресацией, но и потребовала значительно больше памяти.

Двоичные деревья поиска имеет смысл использовать на не слишком большом количестве элементов. Когда элементов много, выгоднее использовать хеш-таблицы. Однако сложность поиска в АВЛ дереве O(log2n), а в хеш-таблице — O(n), но это верно только для таблиц с большим количеством коллизий, в среднем поиск в хеш-таблице с малым числом коллизий занимает O(1).

Массив — самая маленькая структура данных, однако и требует больше всего сравнений при поиске.

Создание оптимальной по размеру хеш-таблицы с малым количеством коллизий представляет собой достаточно трудную задачу, так как при создании хеш-таблицы необходимо найти компромисс между её размером и количеством коллизий, а также правильный выбор хеш-функции.

**Контрольные вопросы:**

1. Чем отличается идеально сбалансированное дерево от АВЛ дерева?

В идеально сбалансированном дереве количество вершин в каждом поддереве различается не больше, чем на 1.

В АВЛ дереве для каждой его вершины высота двух её поддеревьев различается не более, чем на 1.

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

В АВЛ дереве поиск в среднем работает быстрее за счёт меньшей высоты поддеревьев, а следовательно и меньшего количества сравнений.

Однако возможны случаи, когда поиск в ДДП будет быстрее за счёт малой высоты его поддеревьев, в которых содержится искомое число (или не содержится, если данного числа нет).

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

Хеш-таблица — массив, заполненный в порядке, определенным хеш-функцией,

4. Что такое коллизии? Каковы методы их устранения.

Коллизия — ситуация, когда у разных элементов совпадают их хеш-значения.

Коллизии устраняются с помощью открытой адресации, при которой элементы с одинаковым хеш-значением образуют список, дерево, или другую структуру данных, по которой продолжается поиск, либо закрытой адресации, при которой хеш-значение продолжает вычисляться до тех пор, пока не будет найден искомый элемент или не найдётся пустая ячейка.

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

Поиск в хеш-таблицах становится неэффективен при большом количестве коллизий

6. Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска, в хеш-

таблицах и в файле.

Для ДДП: O(h), где h — высота дерева (в среднем), O(n) — худший случай

Для АВЛ: O(log2n)

хеш-таблица:

без коллизий: O(1)

с коллизиями: O(n)