|  |  |
| --- | --- |
| **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)** |

**Дисциплина “Типы и структуры данных”**

**Лабораторный практикум №6**

**по теме: «обработка деревьев, хеш-таблиц»**

Выполнил студент: \_\_*Клименко Алексей Константинович\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*фамилия, имя, отчество*

Группа: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*ИУ7-35Б*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил, к.п.н.: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*подпись, дата*

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Дата \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*2020 г.*

**Цель работы**

Получить навыки применения двоичных деревьев, реализовать основные операции над деревьями: обход деревьев, включение и поиск узлов; построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска и в хеш-таблицах.

**описание условия задачи**

Создать программу для работы с деревьями и хеш-таблицами. Считать данные для заполенения структур из файла. Добавлять числа в структуры по требованию пользователя. Произвести реструктуризацию хеш-таблицы, если среднее число сравнений в ней превысит определённый порог.

Сравнить времена добавления нового ключа, поиска ключа для разных структур данных, а также занимаемый объем памяти.

**Техническое задание**

**Исходные данные**

Исходными данными являются целые числа, считанные из файла.

**Результат**

Результатом работы программы является обработка и отображение деревьев (двоичного поиска и АВЛ-дерева).

**Выходные данные**

Выходными данными являются четыре структуры данных: дерево двоичного поиска, АВЛ-дерево, хеш-таблица и файл.

**Способы обращения к программе**

Для запуска программы необходимо запустить файл **app.exe**. Далее необходимо указать имя файла, в котором содержатся целые числа.

**Возможные аварийные ситуации и ошибки пользователя**

При неверном вводе команды программа попросит ввести команду снова.

При полном заполнении хеш-таблицы новое вводимый ключ будет добавлен во все структуры данных, кроме хеш-таблицы.

**Структуры данных**

Реализация структуры для хранения дерева:

**struct tree**

**{**

**int depth; // глубина от данного узла**

**int diff;  // -1 - левое глубже, 0 - равны, 1 - правое глубже**

**int data; // данные для хранения**

**struct tree \*left; // левое поддерево**

**struct tree \*right; // правое поддерево**

**};**

Реализация структуры хеш-таблицы:

**typedef unsigned int (\*hash\_func\_t)(int); // хеш-функция**

**typedef struct ht\_data**

**{**

**int key; // уникальный ключ**

**bool valid; // флаг валидности ключа**

**} ht\_data\_t;**

**struct hash\_table**

**{**

**unsigned int size; // глубина от данного узла**

**unsigned int step; // шаг для открытого хеширования**

**ht\_data\_t \*data; // массив для хранения ключей**

**hash\_func\_t func; // используемая хеш-функция**

**};**

Для сравнения эффективности хеш-таблицы в зависимости от хеш-функции используются различные хеш-функции:

* **Сумматор.** Возвращает сумму цифр ключа.
* **Сумматор с солью.** Возвращает сумму цифр числа, образованного умножением ключа на большое простое число.
* **Хеширование Фибоначчи.** От результата умножения золотого сечения на ключ отсекается дробная часть и умножается на максимальное значение индекса в таблице (а после - округляется).

**Набор функций**

Для обработки деревьев используются следующие функции:

**// Создаёт новое дерево (пустое).**

**struct tree \*tree\_create(void);**

**// Очищает память и опустошает дерево.**

**void tree\_destroy(struct tree \*\*tree);**

**// возвращает -1 если элемента нет, и 0 - если успешно удалён.**

**int tree\_insert(struct tree \*\*tree, int key);**

**// возвращает -1 если элемента нет, и 0 - если успешно удалён.**

**int tree\_remove(struct tree \*tree, int key);**

**// NULL - если ключ не был найден.**

**struct tree \*tree\_find(struct tree \*tree, int key);**

Для обработки хеш-таблицы используются следующие функции:

**// Создаёт новую хеш-таблицу.**

**struct hash\_table ht\_create(unsigned int size, unsigned int step, hash\_func\_t func);**

**// Удаляет хеш-таблицу.**

**void ht\_destroy(struct hash\_table \*ht);**

**// 0 - успешно добавлен ключ, -1 - своб. место не найдено**

**int ht\_insert(struct hash\_table \*ht, int key);**

**// 0 - ключ есть, -1 - ключа нет**

**int ht\_find(struct hash\_table \*ht, int key);**

**Описание алгоритмов обработки данных**

**Алгоритмы балансировки АВЛ-дерева**

**{ общий головной алгоритм }**

**T: tree - балансируемое дерево**

**начало**

**если T.diff < -1:**

**если T.left не пусто и T.left.right не пусто:**

**T = правый большой поворот (Т);**

**иначе**

**T = правый малый поворот (Т);**

**конец если;**

**иначе если T.diff > 1:**

**если T.right не пусто и T.right.left не пусто:**

**T = левый большой поворот (Т);**

**иначе**

**Т = левый малый поворот (Т);**

**конец если;**

**конец.**

**{ правый большой поворот }**

**T: tree - вращаемое дерево**

**начало**

**x: tree = T;**

**y: tree = x.left;**

**z: tree = y.right;**

**sub\_left: tree = z.left;**

**sub\_right: tree = z.right;**

**z.left = y;**

**z.right = x;**

**x.left = sub\_right;**

**x.left = sub\_left;**

**T = z;**

**конец.**

**{ правый малый поворот }**

**T: tree - вращаемое дерево**

**начало**

**x: tree = T;**

**y: tree = x.left;**

**sub: tree = y.right;**

**y.right = x;**

**x.left = sub;**

**T = y;**

**конец.**

**Алгоритм добавления ключа в хеш-таблицу**

**T: hash\_table - хеш-таблица**

**key: int - вставляемый ключ**

**начало**

**{ вычисление хеша ключа key }**

**h = T.func(key);**

**h = h % T.size;**

**пока T.data[h] занято другим ключом:**

**h = (h + T.step) % T.size;**

**конец пока;**

**T.data[h] = key;**

**конец.**

**Набор функциональных тестов**

|  |  |  |  |
| --- | --- | --- | --- |
| **№** | **Описание теста** | **Входные данные** | **Выходные данные** |
| 1 | Некорректная команда | a | Сообщение о неверной команде. Возврат в меню |
| 2 | Запуск моделирования с настройками по-умолчанию | run A L | Вывод информации в процессе моделирования на экран |
| 3 | Изменение параметров по-умолчанию | config  set a 1.0 2.0  set b 3 3  show | Вывод измененных параметров на экран |
| 4 | Сброс настроек | config  set a 1.0 2.0  set b 3 3  reset  show | Вывод настроек по-умолчанию |
| 5 | Установка неверного промежутка времени | config  set a 20 18 | Вывод сообщении о неверном временном промежутке |
| 6 | Установка нулевого промежутка времени | config  set a 0 0 | Вывод сообщении о неверном временном промежутке |
| 7 | Переход в ручной режим | manual  show | Вывод пустых очередей |
| 8 | Удаление элемента из пустых очередей | manual  pop | Вывод сообщении о том, что нечего удалять |
| 9 | Превышение максимального числа элементов в очередях | manual  push 1  ...  push 11 | Вывод сообщении о переполнении очередей |

**Тесты эффективности по памяти**

Результаты тестирования эффективности по времени операций добавления и удаления из очереди в виде таблицы:

|  |  |  |  |
| --- | --- | --- | --- |
| Операция | время массив | время список | эффективность массива |
| push\_back | 17 тактов | 178 тактов | 90,4 % |
| pop\_front | 18 тактов | 2076 тактов | 99,1 % |

**Тесты эффективности по времени**

Результаты тестирования эффективности по времени операций добавления и удаления из очереди в виде таблицы:

**Выводы по проделанной работе**

Исп.

**Контрольные вопросы**

1. *Что такое дерево?*

Деверо - это структура данных, описываемая рекурентно как узел, у которого есть указатели на два других узла (левое и правое поддеревья).

2. *Как выделяется память под представление деревьев?*

П

3. *Какие стандартные операции возможны над деревьями?*

Стандартные операции над деревьями включают в себя вставку узла в дерево, поиск узла и удаление.

4. *Что такое дерево двоичного поиска?*

Дерево двоичного поиска - это дерево, в котором для каждого узла задано отношение порядка таким образом, что этот узел меньше одного своего поддерева, но больше другого поддерева.

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

Идеально сбалансированное дерево определяется как дерево двоичного поиска, в котором у каждого узла ***количество узлов*** в обоих его поддеревьях отличается не более чем на единицу.

В АВЛ деревьях это требование ослаблено. В них у каждого узла ***высоты*** обоих его поддеревьев отличаются не более чем на единицу.

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

Поиск в сбалансированном дереве зачастую происходит быстрее, так как

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

Хеш-таблица это структура данных

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

Коллизии - это ситуации, когда для разных ключей выбранная хеш-функция возвращает одно и то же значение.

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

При

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

При