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.

Формат входного файла: целые числа, записанные через произвольное количество пробельных символов.

Результат

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

Выходные данные

Выходными данными являются четыре структуры данных: дерево двоичного поиска, АВЛ-дерево, хеш-таблица и файл, а также временные показатели обработки этих структур данных.

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 вводе неверного имени файла программа выводит сообщение о неверном имени файла и завершает работу.

При неверном формате входного файла программа сообщает об этом и завершает работу.

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.
- **Хеширование Фибоначчи.** Из результата умножения золотого сечения на ключ выделяется 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 (T);
    иначе
      T := правый малый поворот (T);
    конец если;
  иначе если T.diff > 1:
    если T.right не пусто и T.right.left не пусто:
      T := левый большой поворот (T);
    иначе
      T := левый малый поворот (T);
  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	Неверное имя файла	a	Сообщение о неверном имени

			файла, завершение работы
2	Неверный формат файла	`a.txt`	Сообщение о неверном содержании файла и завершение работы
3	Пустой входной файл	`empty.txt`	Создание пустых структур данных, нормальное выполенение программы
4	Неверная команда	`data.txt` `swhow bst`	Ожидание повторного ввода команды
5	Вставка существующего ключа	`data.txt` `insert 0`	Сообщение о том, что данный ключ уже есть
6	Вставка ключа при полной заполненности хеш-таблицы	`data.txt` `insert 163`	Сообщение о том, что ключ будет добавлен только в деверья и в файл

Тесты эффективности по памяти

Результаты измерения требуемых объемов памяти для хранения различных структур данных:

Число ключей	Объем дерева	Объем хеш-таблицы	Объем файла
1	32 байт	72 байт	2 байт
32	1024 байт	320 байт	94 байт
128	4096 байт	1088 байт	430 байт

Самым выгодным по хранению данных оказалась файловая структура, так как она не содержит в себе дополнительную информацию для быстрого поиска ключей, а только сами ключи.

Следующей структурой по объёму является хеш-таблица. Хеш-таблица имеет структуру обычного массива, но из-за своей специфики и выбранной хеш-функции чем больше данных записывается в таблицу, тем более вероятнее появления коллизий, которые снижают эффективность таблицы. Значит, в таблице всегда должно оставаться место для новых ключей, а это приводит к значительному увеличению размера структуры.

Последней и самой требовательной к памяти структурой является двоичное дерево. Выбранный метод хранения дерева является динамическим, и не фиксирован по своему размеру. В дерево всегда можно добавить новый уникальный ключ, но из-за большого числа полей-указалетей увеличивается размер структуры, но также уменьшается время доступа к ключам в дереве.

Тесты эффективности по времени

Результаты тестирования эффективности по времени операции добавления ключа в различные структуры данных:

```
>>> show
ДДП:
Размер структуры: 0 байт.
Среднее число сравнений в структуре: 0.00.

AVL:
Размер структуры: 0 байт.
Среднее число сравнений в структуре: 0.00.

Хеш-таблица:
index: |    0|    1|    2|    3|    4|    5|    6|    7|    8|    9|   10|
key:   |     |     |     |     |     |     |     |     |     |     |     |
hash:  |     |     |     |     |     |     |     |     |     |     |     |
Размер структуры: 112 байт.
Среднее число сравнений в структуре: 0.00.
Число коллизий в таблице: 0.
Используемая хеш-функция: хеширование Фибоначчи.

Файл:

Размер структуры: 1 байт.
Среднее число сравнений в структуре: 0.00.
```

```
>>> insert 15
Добавлен ключ 15.

  Структура  |     Время вставки    | Размер структуры | Ср. число сравнений
     ДДП     |          806 тактов  |       32 байт    |       1.0
     AVL     |         1116 тактов  |       32 байт    |       1.0
 Хеш-таблица |        17424 тактов  |      112 байт    |       1.0
    Файл     |       228452 тактов  |        4 байт    |       1.0
>>> insert -23
Добавлен ключ -23.

  Структура  |     Время вставки    | Размер структуры | Ср. число сравнений
     ДДП     |         1022 тактов  |       64 байт    |       1.5
     AVL     |          618 тактов  |       64 байт    |       1.5
 Хеш-таблица |          732 тактов  |      112 байт    |       1.0
    Файл     |       115200 тактов  |        8 байт    |       1.5
>>> insert 56
Добавлен ключ 56.

  Структура  |     Время вставки    | Размер структуры | Ср. число сравнений
     ДДП     |         1766 тактов  |       96 байт    |       1.7
     AVL     |         1500 тактов  |       96 байт    |       1.7
 Хеш-таблица |         1360 тактов  |      112 байт    |       1.0
    Файл     |       109056 тактов  |       11 байт    |       2.0
```

Как видим, файл требует наибольшего времени для добавления нового ключа. Время добавления ключа в дерево двоичного поиска возрастает так как среднее число сравнений постоянно увеличивается. Тоже самое можно сказать и о АВЛ-дереве (на трёх ключах разницу заметить сложно, поэтому необходимо посмотреть результаты вставки с большим числом ключей). Время вставки в хеш-таблицу очень сильно колеблется. Время добавления ключа в хеш-таблицу обуславливается в первую очередь алгоритмической сложностью выбранной хеш-функции и количеством колиизий в таблице.

Средние времена добавления ключей в двоичные деревья и в хеш-таблицу прямопропорциональны среднему числу сравнений в них. Чем больше сравнений нужно сделать, тем больше будет время вставки и/или поиска ключа. Рассмотрим пример добавления ключа в структуры, в которых уже имеется 200 ключей:

```
>>> insert 2000
Добавлен ключ 2000.

  Структура  |     Время вставки    | Размер структуры | Ср. число сравнений
     ДДП     |       14868 тактов   |      6432 байт   |        8.2
     AVL     |        3906 тактов   |      6432 байт   |        6.8
 Хеш-таблица |        4038 тактов   |      1664 байт   |       92.0
    Файл     |      166804 тактов   |       688 байт   |      101.0
```

Выводы по проделанной работе

Деревья, хеш-таблицы и файлы, как структуры данных удобны для хранения большого объема данных, когда наиболее важным фактором является скорость произвольного доступа к этим данным.

Дерево двоичного поиска целесообразно использовать в случаях, когда данные поступают в дерево в равномерно распределённом виде (нет частично отсортированных последовательностей или их количество ничтожно мало), в следствие чего, оно разрастается равномерно во всех направлениях и не требует дополнительной балансировки.

АВЛ-деревья удобны в ситуациях, когда исходный объем данных может быть частично сортирован. Тогда, в отличие от дерева двоичного поиска, оно может быть вовремя сбалансировано, что приведёт к ускорению доступа к ключам и уменьшею среднего числа сравнений при поиске в структуре.

Хеш-таблицы используются в случаях, когда приоритетным фактором является скорость доступа к элементам структуры. В таких ситуациях не важен факт частичной или полной упорядоченности входных данных, в таблице каждый ключ уже имеет собственное место, которое не зависит от других ключей. Однако у такого способа хранения могут возникнуть сложности в случае обнаружения коллизий, но это уже зависит от выбранной хеш-функции и алгоритма хеширования.

Файл, как структура для хранения данных подходит очень даже хорошо, но вот для обработки не очень. Поиск по файлу происходит последовательно, из-за чего он сильно уступает остальным структурам по

времени обработки данных. Однако в нём хранятся только исходные данные, что делает его выбор наиболее эффективным решением по памяти.

Контрольные вопросы

1. *Что такое дерево?*

Деверо - это нелинейная структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим». Эта структура данных описывается рекурентно как узел, у которого есть указатели на два других узла (левое и правое поддеревья).

Деревья используются при построении организационных диаграмм, анализе электрических цепей, для представления синтаксических структур в компиляторах программ, для представления структур математических формул, организации информации в СУБД и, кроме того, для более эффективного извлечения данных.

2. *Как выделяется память под представление деревьев?*

Память для представления в виде связного списка выделяется динамически в момент добавления новых ключей.

3. *Какие стандартные операции возможны над деревьями?*

Стандартные операции над деревьями включают в себя вставку узла в дерево, поиск узла, балансировка дерева. Также возможно отделить поддерево в отдельное дерево.

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.

Данное свойство позволяет производить более быстрый доступ к узлам дерева.

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 высота несбалансированного дерева как правило превосходит высоту того же сбалансированного дерева.

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

Хеш-таблица это структура для данных с произвольным доступом к ним. Принцип построения хеш-таблицы основан на особой функции, называемой хеш-функцией, которая сопоставляет уникальный ключ с его

местом в таблице. Идеальная хеш-функция - это инъекция множества ключей во множество мест в таблице.

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

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

Коллизии могут возникать на этапе "упаковки" расчитанного большого хеша в размерность таблицы. То есть хеш-значения разных ключей могут быть разными, но при упаковке они получают одно и то же место в таблице. Такого рода коллизии могут быть устранены изменением размерности таблицы.

Другой случай колиизий - полное совпадение хешей двух различных ключей. Данный вид коллизий возникает по причине неидеальности выбранной хеш-функции и может быть устранён только с помощью её замены на другую.

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

Поиск в хеш-таблице может становиться неэффективным в случаях большого числа коллизий, из-за которых нужно будет производить дополнительный последовательный поиск по ключам, имеющим одинаковый хеш.