

Отчёт

Дисциплина: Типы и структуры данных
Лабораторная работа №6
по теме: «Деревья, хэш-таблицы»

Работу выполнил:
студент группы ИУ7-32Б
Иванов Всеволод

Описание условия задачи

Реализовать основные операции работы с деревом: обход дерева, включение, исключение и поиск узлов, сбалансировать дерево, сравнить эффективность алгоритмов сортировки и поиска в зависимости от высоты деревьев и степени их ветвления.

Построить хеш-таблицу и вывести ее на экран, устранить коллизии, если они достигли указанного предела, выбрав другую хеш-функцию и реструктуризировав таблицу; сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска (ДДП) и в хеш-таблицах. Сравнить эффективность реструктуризации таблицы для устранения коллизий с эффективностью поиска в исходной таблице.

Описание технического задания

Программа для обработки чисел, хранящихся в текстовом файле. Программа должна построить дерево двоичного поиска, сбалансировать его в AVL дерево, а также построить хэш-таблицу из исходных данных. После чего добавить в данные структуры и файл введенное число.

Область применения – файл с целыми числами

Сроки выполнения – 2 недели

Основой для разработки программы послужила потребность в реализации методов быстрого поиска. Выполняемой задачей было поставлено построение дерева, AVL дерева и хэш-таблицы на основе целых чисел из текстового файла, а также изучить эффективность поиска, добавления и удаления элементов из данных структур.

Для устранения коллизий в хэш-таблице был выбран метод закрытого хэширования.

Программа должна:

- Содержать указание формата и диапазона данных при вводе
- Содержать указание операций, производимых программой
- Контролировать правильность ввода данных
- Иметь пояснения при выводе результата
- Обеспечить вывод дерева в графическом виде
- Обеспечить вывод хэш-таблицы
- Вывести время и количество сравнений при поиске одних и тех же данных при различных структурах данных.

Этап постановки

Исходными данными являются целые числа, хранящиеся в текстовом файле или вводимые с консоли. Результатом является также файл с целыми числами, а также сравнение времени и памяти добавления элементов в различные структуры данных.

Обращение к программе происходит выбором функций программы путём введения их кодов в консоли. Вывод результатов работы программы также производится в консоль или в файл.

Возможными ошибками являются:

- Ввод символов в поле команды
- Ввод несуществующей команды
- Отсутствие входного файла
- Пустой файл или некорректный формат входного файла
- Попытка удаления элемента из пустой структуры

Особых требований для технических средств, аппаратной совместимости, программных средств не имеется.

Описание СД

Для хранения элементов дерева и АВЛ дерева была создана структура `node_t`:

- `val (int)` – хранимое число
- `bal(int)` – баланс левого и правого поддерева (используется только при обработке АВЛ дерева)
- `left, right (указатель на tree_t)` – указатели на левого и правого потомка данного узла

```
// Tree element type
typedef struct node_tag
{
    int val;
    int bal;
    struct node_tag *left;
    struct node_tag *right;
} node_t;
typedef node_t *tree_t;
```

Для хранения хэш-таблицы была создана структура `table_t`:

- `size (int)` – размер таблицы
- `count (int)` – количество элементов таблицы
- `hash_n (int)` – номер используемой хэш-функции
- `func (функция int, принимающая 2 числа int)` – указатель на хэш-функцию
- `arr (динамический массив int)` – массив элементов
- `is_busy (динамический массив int)` – массив состояний ячеек таблицы (занято/не занято)

```
typedef int* array_t;
typedef struct
{
    int size;
    int count;
    int hash_n;
    int (*func) (int, int);
    array_t arr;
    array_t is_busy;
} table_t;
```

Описание алгоритма

Из файла поэлементно считываются элементы и вставляются в бинарное дерево. Вставка происходит рекурсивно в левое поддерево, если элемент меньше элемента корня, и в правое, если оно больше.

После этого данное дерево балансируется в АВЛ дерево. Сначала рекурсивно балансируются левое и правое поддерево, после чего происходит балансировка дерева, с использованием поворотов. В процессе поворота меняются местами узлы дерева и изменяются балансы вершин деревьев. Так

как балансировка происходит с уже готовым деревом, а не в процессе добавления новых элементов, в случае осуществления поворота требуется повторить операцию балансировки для данного дерева. Это вызвано тем, что в таком дереве после поворота может появиться дисбаланс уже в одном из поддеревьев.

Далее происходит поэлементное считывание элементов файла в хэш-таблицу. Позиции вставляемых элементов определяются хэш-функцией. В случае, если таблица излишне заполняется, происходит её реструктуризация, путём создания более большой таблицы и переносом в неё элементов из старой. В случае, если количество сравнений поиска в хэш-таблице превышает заданное, хэш-функция меняется на следующую в заданном массиве хэш-функций и происходит вышеописанная реструктуризация.

В программе были использованы следующие хэш-функции:

```
int hash_f0(int size, int num)
{
    return abs(num % size);
}

int hash_f1(int size, int num)
{
    int hash;
    num *= num;
    hash = num << 24;
    hash ^= num >> 24;
    num = num >> 8;
    return abs((num ^ hash) % size);
}

int hash_f2(int size, int num)
{
    return abs(size * ((GOLDEN_RATIO * num) - (int)(GOLDEN_RATIO * num)));
}

int hash_f3(int size, int num)
{
    num += num << 3;
    num ^= num >> 11;
    num += num << 15;
    return abs(num % size);
}
```

Набор тестов

Негативные тесты

Ввод	Проверяемый случай	Вывод
50	Использование несуществующей команды	Ошибка: команда не существует
Afa	Ввод символов в поле команды	Ошибка: неправильный формат ввода
afa	Ввод символов вместо чисел	Ошибка: неправильный формат ввода
da.exe	Некорректный или несуществующий входной файл	Ошибка: файл нечитаемый
data.txt: 2 afaf 2	Некорректное содержание файла	Ошибка: неправильный формат файла
вызов удаления элемента	Попытка удаления элемента из пустого дерева	Ошибка: дерево пусто
вызов удаления элемента	Попытка удаления элемента из пустого AVL дерева	Ошибка: AVL дерево пусто
вызов удаления элемента	Попытка удаления элемента из пустой таблицы	Ошибка: таблица пуста
вызов поиска элемента	Попытка поиска элемента в пустой структуре	Ошибка: структура пуста
вызов подсчёта среднего количества сравнений	Попытка подсчёта среднего количества сравнений в пустой структуре	Ошибка: структура пуста
вызов вывода обходов	Попытка вывести обход пустого дерева	Ошибка: дерево пусто

Тепличные тесты

Ввод	Проверяемый случай	Вывод
начало программы Команда 2 data.txt 2	Работа с файлом	*Вывод дерева, AVL дерева и хэш-таблицы из чисел файла* *Вывод дерева, AVL дерева и хэш-таблицы с новым элементом* *завершение программы*
начало программы Команда 1	Запуск тестового режима	*Переход к выбору тестируемой структуры*

начало программы Команда: 3	Запуск анализа	Данные по времени и занимаемой памяти для проведения установленного количества добавлений в структуры *завершение работы программы*
тестовый режим Команда: 1	Запуск режима тестирования бинарного дерева	*Переход к меню тестирования дерева*
тестовый режим Команда: 2	Запуск режима тестирования AVL дерева	*Переход к меню тестирования AVL дерева*
тестовый режим Команда: 3	Запуск режима тестирования хэш-таблицы	*Переход к меню тестирования хэш-таблицы*
тестовый режим Команда: 21	Удаление элемента из очереди на массиве	Вывод удалённого элемента Вывод стека
тестовый режим Команда: 30	Сравнение времени и памяти при моделировании с помощью очереди на списке и массиве	Время и память, затраченные на моделирование работы ОА Сравнение в процентах эффективности очередей
основное меню Команда: 0	Переход в тестовый режим	*запуск тестового режима*
основное меню Команда: 1	Моделирование ОА с использованием очереди на массиве	*вывод информации о результатах моделирования* *завершение программы*
основное меню Команда: 2	Моделирование ОА с использованием очереди на списке	*вывод информации о результатах моделирования* *завершение программы*

Общее для тестовых режимов дерева, AVL дерева и хэш-таблицы		
Ввод	Проверяемый случай	Вывод
Команда 0	Выход	Выход из программы
Команда 1 2	Добавление элемента	Элемент добавлен
Команда 2 2	Удаление существующего элемента	Элемент удалён
Команда 2 3	Удаление несуществующего элемента	Элемент не в структуре
Команда 3 2	Поиск существующего элемента	Элемент найден за n сравнений
Команда 3 3	Поиск существующего элемента	Элемент найден после n сравнений


Команда 4 data.txt	Считывание файла	Файл считан *вывод построенной структуры*
Команда 5	Подсчёт среднего количества сравнений для поиска в структуре	Среднее количество сравнений

Граничные тесты

Ввод	Проверяемый случай	Вывод
удаление элемента	Удаление единственного элемента структуры	Пустая структура

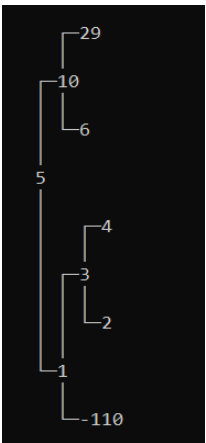
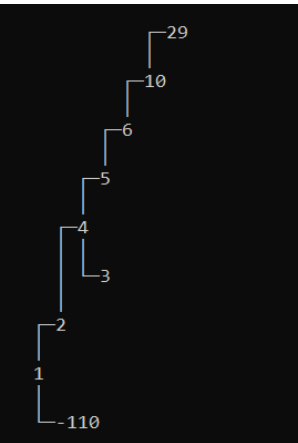
Пример работы программы

Для входного файла data.txt строятся следующие структуры:

 data – Блокнот

Файл Правка Формат Вид Справка
1 2 4 5 6 10 3 | -110 29

Бинарное дерево поиска, АВЛ дерево и хеш-таблица:



#	Value	Hash
0	-110	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	29	7
8	---	--
9	---	--
10	10	10

Обходы данного АВЛ дерева:

Postfix traversal: -110 1 3 4 2 6 29 10 5

Prefix traversal: 5 2 1 -110 4 3 10 6 29

Infix traversal: -110 1 2 3 4 5 6 10 29

Также для данных структур программа вычисляет среднее количество сравнений при поиске:

Бинарное дерево – 3.78, АВЛ дерево – 2.78, Хеш-таблица – 1.0

Сравнение эффективностей

1. для хэш-таблицы проводились замеры также и с динамическим расширением размера, но в сравнении они не указаны, т.к. в основном хэш-таблица не изменяет свой размер
2. добавление в файл осуществляется в конец, из-за чего числа в нём неупорядочены
3. Хеш-функции – побитовые сдвиги

Добавление 100 000 элементов:

```
>>> Comparing time and memory of adding 100000 and searching for 1000 int numbers
```

Struct	Time add	Memory	Time find	Find cmp	
Tree	37.00 ms.	2343.750 Kb	0.270 ms.	21247 cmp	
AVL Tree	41.00 ms.	2343.750 Kb	0.210 ms.	15778 cmp	
Hash table	27.00 ms.	1121.820 Kb	0.030 ms.	2009 cmp	(resize)
Hash table	6.00 ms.	937.586 Kb	0.030 ms.	2712 cmp	(full size)
File	33.00 ms.	585.938 Kb	5711.00 ms.	46802864 cmp	

Структура	Время добавления	Память
Дерево	100%	100%
АВЛ дерево	125%	100%
Хэш-таблица	80%	48%
Файл	90%	25%

Добавление 300 000 элементов:

```
>>> Comparing time and memory of adding 300000 and searching for 1000 int numbers
```

Struct	Time add	Memory	Time find	Find cmp	
Tree	138.00 ms.	7031.250 Kb	0.310 ms.	22338 cmp	
AVL Tree	159.00 ms.	7031.250 Kb	0.240 ms.	17371 cmp	
Hash table	87.00 ms.	3350.008 Kb	0.020 ms.	1879 cmp	(resize)
Hash table	15.00 ms.	2812.555 Kb	0.040 ms.	2660 cmp	(full size)
File	65.00 ms.	1757.813 Kb	18870.00 ms.	145099297 cmp	

Структура	Время добавления	Память
-----------	------------------	--------

Дерево	100%	100%
АВЛ дерево	115%	100%
Хэш-таблица	25%	48%
Файл	60%	25%

Добавление 1 000 000 элементов:

```
>>> Comparing time and memory of adding 1000000 and searching for 1000 int numbers
```

Struct	Time add	Memory	Time find	Find cmp	
Tree	778.00 ms.	23437.500 Kb	0.420 ms.	25417 cmp	
AVL Tree	828.00 ms.	23437.500 Kb	0.390 ms.	19248 cmp	
Hash table	346.00 ms.	10003.180 Kb	0.030 ms.	2284 cmp	(resize)
Hash table	86.00 ms.	9375.055 Kb	0.050 ms.	2624 cmp	(full size)
File	189.00 ms.	5859.375 Kb	71603.00 ms.	482844861 cmp	

Структура	Время добавления	Память
Дерево	100%	100%
АВЛ дерево	115%	100%
Хэш-таблица	20%	48%
Файл	35%	25%

Ответы на вопросы

1. Что такое дерево?

Дерево – нелинейная структура данных для представления иерархических связей «один ко многим»

2. Как выделяется память под представление деревьев?

Для представления деревьев память выделяется отдельно под каждый узел дерева. Объём памяти одного узла равен `sizeof(node_t)`, а общий - `sizeof(node_t) * общее количество узлов`

3. Какие стандартные операции возможны над деревьями?

Для деревьев возможны операции добавления, удаления, поиска элемента и измерения степени и глубины дерева

4. Что такое дерево двоичного поиска?

Дерево двоичного поиска – дерево, в котором у узла существует два потомка, при этом все левые потомки меньше данного узла, а все правые больше. Таким образом, поиск в дереве существенно проще простого дерева, так как основываясь на данном свойстве можно найти элемент за один спуск от корня до листа

5. Чем отличается идеально сбалансированное дерево от AVL дерева?

В идеально сбалансированном дереве количество элементов в левом и правом поддереве отличаются не более, чем на 1 для любого узла

В AVL дереве не более, чем на 1 отличаются глубины левых и правых поддеревьев

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

В AVL дереве средняя длина поиска примерно равна длине поиска в идеально сбалансированном дереве ($\approx \log_2 n$). В дереве двоичного поиска

время поиска зависит от данных и порядка их добавления в дерева. Время поиска в нём варьируется от $\log_2 n$ до n .

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

Хэш-таблица — это массив, порядок размещения в котором определяется связанным с ним хэш-функцией.

8. Что такое коллизии? Каковы методы их устранения.

Коллизии в хэш-таблицах возникают, когда два разных элемента в соответствии с хэш-функцией должны располагаться на одной позиции. Существуют методы внешнего хеширования (создание односвязных списков в каждой ячейке массива, хранящие все элементы с данным ключом) и внутреннего хеширования (поиск незанятой ячейки таблицы, и вставки элемента на найденную позицию)

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

В случае, когда для поиска элемента требуется более 3-4 сравнений

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

Среднее время поиска в ДДП варьируется от $\log_2 n$ до n .

В АВЛ время поиска стремится к $\log_2 n$

В хэш-таблице время постоянно (в случае, если она не переполнена коллизиями)

Выводы по проделанной работе

Из проведённого анализа следует вывод, о том, в каких случаях следует применять данные структуры. Хэш-таблицы с умеренным количеством коллизий показывает лучшее время по поиску и добавлению элементов, при этом занимая меньшую память, по сравнению с деревьями.

Преимущество деревьев проявляется при частом изменении данных, так как они лучше приспособлены к динамике. АВЛ деревья при этом обладают качеством устойчивости времени поиска вне зависимости от входных данных. АВЛ деревья затрачивают меньшее время на поиск, по сравнению с обычными бинарными деревьями. При этом время добавления и удаления элементов уступает бинарному дереву лишь на 20%.

Добавление элементов в файл занимает наименьшую память и время, меньше деревьев, но лишь при условии, что данные файле будут неупорядоченными, что приводит к линейному времени поиска в них.

Экспериментально подтверждено, что время поиска в хэш-таблице не зависит от её размера, в деревьях стремится к логарифмической сложности, а в неупорядоченном файле к линейной.