МГТУ им. Баумана

Типы и структуры данных

Лабораторная работа №6. "Деревья, Хеш-таблицы"

Студент Ивахненко Дмитрий - ИУ7-36Б

Описание условия задачи

В текстовом файле содержатся целые числа.

- Построить ДДП из чисел файла.
- Вывести его на экран в виде дерева.
- Сбалансировать полученное дерево и вывести его на экран.
- Построить хеш-таблицу из чисел файла.
- Использовать закрытое хеширование для устранения коллизий.
- Осуществить удаление введенного целого числа в ДДП, в сбалансированном дереве, в хеш-таблице и в файле.
- Сравнить время удаления, объем памяти и количество сравнений при использовании различных (4-х) структур данных.
- Если количество сравнений в хеш-таблице больше указанного, то произвести реструктуризацию таблицы, выбрав другую функцию

Техническое задание

Входные данные:

1. Номер команды - целое число в диапазоне от `0` **до** `25` включительно.
2. **Командно-зависимые данные:**
    - Добавить/удалить/найти элемент
        - значение элемента - целое число.
    - Считать из фалйа
        - имя файла - строка.
    - Сгенерировать файл
        - Название файла - строка
        - Кол-во чисел, границы / шаг.

Выходные данные:

В зависимости от выбранного действия результатом работы программы могут являться:

1. Текущее состояние структуры данных (ДДП, АВЛ-дерева, Хеш-таблицы)

2. Статистика по времени выполнения и объему занимаемой памяти при обработке структур данных.
3. Результат поиска/удаления/добавление эл-та в СД. Информация о времени и кол-ве сравнений.

Команды программы

```
======Преобразовать======
1. ДДП -> АВЛ-дерево
=======АВЛ-дерево=======
2. Добавить
3. Удалить
4. Найти
5. Текущее состояние
6. Разрушить
7. Считать из файла
==========ДДП==========
8. Добавить
9. Удалить
10. Найти
11. Текущее состояние
12. Разрушить
13. Считать из файла
=========Хеш-таблица========
14. Добавить
15. Удалить
16. Найти
17. Текущее состояние
18. Сброс
19. Проверить необходимость реструктуризации
20. Реструктурировать
21. Считать из файла
=========Дополнительно========
22. Сгенерировать файл со случайными числами
23. Сгенерировать файл с упорядоченными числами
24. Считать из файла во все СД
25. Анализ
```

Обращение к программе:

Запускается из терминала при помощи комманды `./bin/app`.

Аварийные ситуации:

1. Некорректный ввод номера команды.

   На входе: число, большее, чем максмальный индекс команды или меньшее, чем минимальный.

   На выходе: `[Ошибка] ::Некорректный ввод пункта меню::`

2. Некорректный ввод ключа при поиске/удаления/добавлении элемента.

   На входе: ввод, отличный от указанного в ТЗ.

   На выходе: `[Ошибка] ::Некорректный ввод::`

3. Некорректный ввод названия файла при открытии

   На входе: ввод, отличный от указанного в ТЗ

   На выходе: `[Ошибка] ::Не удалось открыть файл`

4. Некорректный ввод параметров при генерации файла.

   На входе: ввод, отличный от указанного в ТЗ

   На выходе: `[Ошибка] ::Некорректный ввод::`

5. Переполнение хеш-таблицы

   На входе: Попытка добавить ключ в заполненную таблицу

   На выходе: `[Ошибка] ::Таблица полностью заполнена!::`

6. Попытка добавить дубликат

   На входе: добавление ключа, уже содержащегося в дереве / таблице

   На выходе: `[Ошибка] ::Дерево / Таблица уже содержит данный элемент::`

Структуры данных

Двоичное дерево поиска:

```
typedef struct {
    NodeBST *root;          // Указатель на корень дерева
    size_t size;            // Размер дерева (кол-во элементов в дереве)
} TreeBST;
```

Узел ДДП:

```
typedef int treeType_t;     // Элементы дерева - целые числа
struct NodeBST {
    treeType_t key;         // элемент дерева
    NodeBST *right;         // указатель на правый дочерний элемент
    NodeBST *left;          // указатель на левый дочерний элемент
};
```

АВЛ-дерево:

```
typedef struct {
    NodeAVL *root;          // Указатель на корень дерева
    size_t size;            // Размер дерева (кол-во элементов в дереве)
} TreeBST;
```

Узел АВЛ-дерева:

```
struct NodeAVL {
    treeType_t key;         // элемент дерева
    unsigned char height;   // высота данного узла
    NodeAVL *right;         // указатель на правый дочерний элемент
    NodeAVL *left;          // указатель на левый дочерний элемент
};
```

Хеш-таблица:

```
typedef size_t(*hashFunc_t)(HashTable *, htItem_t);     // Указатель на
хеш-функцию
struct Hashtable {
    htItem_t *items;        // буфер с элементами таблицы
    size_t capacity;        // полный размер таблицы
    size_t count;           // текущее кол-во элементов а таблице
    hashFunc_t hashFunc;    // хеш-функция, используемая на данный момент
};
```

Описание основных функций

```
/* Создание СД*/
TreeBST *CreateTreeBST();
TreeAVL *CreateTreeAVL();
HashTable *CreateTable(size_t capacity);

/* Удаление СД*/
void DeleteTreeBST(TreeBST *tree);
void DeleteTreeAVL(TreeAVL *tree);
void FreeTable(HashTable *table);

/* Поиск в СД*/
bool SearchInTreeBST(NodeBST *node, const treeType_t key, size_t
*__numOfCmp__);
bool SearchInTreeAVL(NodeAVL *node, const treeType_t key, size_t
*__numOfCmp__);
ssize_t SearchInTable(HashTable *table, htItem_t item, size_t
*__numOfCmp__);
```

```
/* Вставка в СД*/
NodeBST *InsertBST(NodeBST *node, const treeType_t key, bool *added);
NodeAVL *InsertAVL(NodeAVL *node, const treeType_t key, bool *added);
ssize_t InsertToTable(HashTable *table, htItem_t item);

/* Удаление из СД*/
NodeBST *RemoveBST(NodeBST *node, const treeType_t key, bool *found, size_t
*__numOfCmp__);
NodeAVL *RemoveAVL(NodeAVL *node, const treeType_t key, bool *found, size_t
*__numOfCmp__);
ssize_t RmFromTable(HashTable *table, htItem_t item, size_t *__numOfCmp__);

/* Текущее состояние СД*/
void PrintTable(FILE *ostream, HashTable *table);
void ExportToDotAVL(FILE *stream, const char *treeName, NodeAVL *root);
void ExportToDotBST(FILE *stream, const char *treeName, NodeBST *root);


/* Сменить хеш-функцию*/
int ChangeHashFunc(HashTable *table);

/*Реструктуризировать таблицу*/
void RestructTable(HashTable **table, size_t newCapcacity);

/* Хеш-функции */
size_t hashFuncMod(HashTable *table, htItem_t item) {
    return (size_t)(item % table->capacity);
}

size_t JenkinsHashFunc(HashTable *table, htItem_t item) {
    unsigned int x = item >= 0 ? item : -item;
    x = ((x >> 16) ^ x) * 0x45d9f3b;
    x = ((x >> 16) ^ x) * 0x45d9f3b;
    x = (x >> 16) ^ x;

    return (size_t)(x % table->capacity);
}

size_t hash32shift(HashTable *table, htItem_t item) {
    unsigned int x = item >= 0 ? item : -item;
    int c2 = 0x27d4eb2d;
    x = (x ^ 61) ^ (x >> 16);
    x = x + (x << 3);
    x = x ^ (x >> 4);
    x = x * c2;
    x = x ^ (x >> 15);
    return (size_t)(x % table->capacity);
}
```

Алгоритм

1. На экран пользователю выводится меню

2. Пользователь вводит номер команды
3. Выполняется действие согласно номеру команды

1. Деревья

Обход

Основной рекурсивный подход для обхода (непустого) бинарного дерева: Начиная с узла N делаем следующее:

- (L) Рекурсивно обходим левое поддерево. Этот шаг завершается при попадании опять в узел N.
- (R) Рекурсивно обходим правое поддерево. Этот шаг завершается при попадании опять в узел N.
- (N) Обрабатываем сам узел N.

Эти шаги могут быть проделаны в любом порядке:

- сверху вниз: N,L,R.
- слева направо: L,N,R
- снизу вверх: L,R,N

Вставка

1. Если на текущем шаге указатель на узел дерева пуст, то мы нашли место вставки, а значит можем создать новый узел и присвоить ему значение данных, переданных для вставки, а затем вернуть этот узел.

2. Если указатель на узел дерева не пуст, то сравним значение в этом узле с переданными данными: если значение узла больше, то продолжим поиск места вставки в левом поддереве, иначе в правом (случай равенства зависит от конкретной реализации, например, можно не включать узел, если такой уже есть).

3. Возврат текущего узла дерева (с изменённым левым или правым потомком). При реализации сбалансированного дерева перед возвратом необходимо применить функцию балансировки к данному узлу

Удаление

Идея следующая: находим узел p с заданным ключом k (если не находим, то делать ничего не надо), в правом поддереве находим узел min с наименьшим ключом и заменяем удаляемый узел p на найденный узел min.

1. Если на текущем шаге указатель на узел дерева пуст, то мы пришли в потомок листа, а значит надо вернуть нулевой указатель.

2. Если указатель на узел дерева не пуст, то сравним значение в этом узле с переданными данными: если значение узла больше, то продолжим поиск элемента для удаления в левом поддереве, иначе в правом. Выйдя из рекурсии, вернём указатель на текущий узел дерева (с изменённым левым или правым потомком). При реализации сбалансированного дерева перед возвратом необходимо применить функцию балансировки к данному узлу.

3. Если значение узла равно переданным данным, то мы нашли элемент, который необходимо удалить. Сохраним указатели на потомков и удалим данный узел.
    - Если правого потомка не существует, то вернём указатель на левого потомка (для ДДП это очевидный шаг, а для АВЛ это справедливо благодаря его свойству: поскольку правый потомок отсутствует, то левый потомок либо вообще не существует, либо является листом).
4. Если же правый существует, то найдём минимум в правом поддереве (очевидно нужно двигаться по левым потомкам). Извлекаем минимум и присваиваем его правому потомку указатель на правого потомка исходного узла, который может быть получен после извлечения минимума (для АВЛ в процессе получения надо также выполнять условие балансировки). Левому потомку минимума присваиваем указатель на левый потомок исходного узла. Возвращаем минимум. При реализации сбалансированного дерева перед возвратом необходимо применить функцию балансировки к данному узлу.
5. Возврат текущего узла дерева (с изменённым левым или правым потомком). При реализации сбалансированного дерева перед возвратом необходимо применить функцию балансировки к данному узлу.

Балансировка (для АВЛ)

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины.

`Используются 4 типа вращений:`

- Малое левое вращение
- Большое левое вращение
- Малое правое вращение
- Большое правое вращение

Балансировка реализуется с помощью переприсвоения указателей.

2. Хеш-таблица

Выбор размера таблицы

- На старте по умолчанию установлен размер 67 элементов (простое число)
- В случае считывания таблицы из файла, она будет очищена, а ее размер будет выбран в соответствии с кол-вом чисел в файле.

1. Происходит подсчет кол-ва чисел в файле `(N)`
2. Находится первое простое число, большее чем `1.2 x N` - это и будет размером таблицы.

Выбор размера таблицы "с запасом" необходим для уменьшения коллизий. По той же причине выбирается простое число - хеш-функция, основанная на взятии остатка, дает меньше коллизий, если размер таблицы - простое число.

Реструктуризация

По запросу пользователя возможна реструктуризация таблицы. Для того, чтобы понять, нужно ли делать реструктуризацию, пользователь может выбрать соответствующий пункт меню.

Программа подсчитает среднее кол-во сравнений для поиска элемента в таблице на данный момент. Если это кол-во превысит 4 - программа *порекомендует* произвести реструктуризацию.

Реструктуризация происходит следующим образом:

0. В зависимости от решения пользователя возможна замена хеш-функции (Всего их в программе 3, они меняются по кольцу)
1. Создается новая хеш-таблица, размером на 20% больше текущей
2. На основе текущей хеш-функции (см. п. 0) производится перестановка всех элементов в новую таблицу.
3. После завершения переноса элементов, старая таблица удаляется, память из под нее высвобождается. Старая таблица заменяется на новую.

Тесты

# Описание теста	Вввод	Вывод
Неправильный номер команды	-1 `<Enter>`	Сообщение об ошибке
Неправильный ввод ключа	2 `<Enter>` f `<Enter>`	Сообщение об ошибке
Добавить элемент	2 `<Enter>` 9 `<Enter>`	Сообщение об успешном добавлении
Попытка считать данные из несуществующего файла	7 `<Enter>` WrongPath `<Enter>`	Сообщение об ошибке
Неправильный ввод параметров генерации файла	WrongPath `<Enter>` 5 100 0 `<Enter>`	Сообщение об ошибке
Провести сравнительный анализ	25 `<Enter>`	Таблица-результат анализа

Оценка эффективности

Сравнительный анализ 4-ех структур данных `(Поиск/Удаление элемента)` Для каждого измерения взято среднее значение по времени для **100** повторений.

`Сокращения:`

- **АВЛ** - АВЛ-дерево
- **ДДП** - Дерево двоичного поиска
- **Ф** - Файл

- **ХТ1** - Хеш-таблица с использованием хеш-функции #1 (Остаток от деления)
- **ХТ2** - Хеш-таблица с использованием хеш-функции #2 (Хеш-функция Дженкинсона)

Единицы измерения:

- Время - **нс**
- Память - **байты**

Время

Неупорядоченные данные

Исходные данные - N случайных чисел. Числа генерируются заново на каждом повторении.

Деревья

Размер	Сравнения (АВЛ)	Поиск (АВЛ)	Удаление (АВЛ)	Сравнения (ДДП)	Поиск (ДДП)	Удаление (ДДП)
512	8.2/10.2	143.595	551.430	10.7/12.1	169.550	243.176
1024	9.2/11.2	147.256	604.530	12.0/13.5	194.418	274.718

Хеш-таблица

Размер	Сравнения (ХТ1)	Поиск (ХТ1)	Удаление (ХТ1)	Сравнения (ХТ2)	Поиск (ХТ2)	Удаление (ХТ2)
512	3.29	95.907	108.633	3.26	117.048	120.151
1024	3.43	97.559	110.840	3.30	119.289	124.982

Файл

Размер	Сравнения (Ф)	Поиск (Ф)	Удаление (Ф)
512	512.50	87738	318318
1024	1024.50	162345	599629

Упорядоченные данные

Исходные данные - Отсортированные числа. Числа генерируются заново на каждом повторении.

Деревья

Размер	Сравнения (АВЛ)	Поиск (АВЛ)	Удаление (АВЛ)	Сравнения (ДДП)	Поиск (ДДП)	Удаление (ДДП)
512	8.0/10.0	104.376	399.370	256.5/256.5	1510.416	2432.190

Размер	Сравнения (АВЛ)	Поиск (АВЛ)	Удаление (АВЛ)	Сравнения (ДДП)	Поиск (ДДП)	Удаление (ДДП)
1024	9.0/11.0	110.973	490.915	512.5/512.5	3170.113	6217.681

Хеш-таблица

Размер	Сравнения (ХТ1)	Поиск (ХТ1)	Удаление (ХТ1)	Сравнения (ХТ2)	Поиск (ХТ2)	Удаление (ХТ2)
512	3.31	97.918	108.164	3.28	117.568	121.891
1024	3.40	100.104	110.060	3.43	120.196	126.085

Файл

Сравнения (Ф)	Поиск (Ф)	Удаление (Ф)
512.50	81473	295952
1024.50	153361	570042

Память

Упаковка	Размер	АВЛ	ДДП	Хеш-таблица
Без упаковки	1024	24600	24600	4955
С упаковкой	1024	21528	20504	4955

Вывод

Исходя из полученных экспериментальных данных, оценим эффективность использования структур 4-ех данных для поставленной задачи (удаление элемента).

Хеш-таблица

В среднем хеш-таблица быстрее всех СД *(Замечание: для хеш-таблицы взят лучший результат из двух хеш-функций)*

Так при размерности **512** элементов она выигрывает по времени

- На неотсортированных данных
  - ДДП: `на ~120%` или `в ~2,2 раза`
  - АВЛ-дерево: `на ~410%` или `в ~5,1 раза`
  - Файл: `в ~3000 раз`
- На отсортированных данных
  - ДДП: `на ~2150%` или `в ~22,5 раз !`
  - АВЛ-дерево: `на ~270%` или `в ~3.7 раз`
  - Файл: `в ~5700 раз`

(Поскольку оценка хеш-таблицы в лучшем случае близка к `O(1)`, на б**о**льших данных таблица все также эффективна, в отличие от других СД)

Однако нужно учитывать, что данные результаты справделивы только в случае, если кол-во коллизий минимально (в среднем 3-4 сравнения), то есть таблица имеет достаточный размер и верно подобранную хеш-функцию.

В случае б**о**льшего кол-ва коллизий эффективность хеш-таблицы будет уменьшаться. В этом случае ее придется реструктурировать путем увеличиения размера и/или выбора другой хеш-функции, на что будут тратиться ресурсы, поскольку придется перевыделять память и тратить время на перераспределение элементов.

Таким образом, хеш-таблица будет особо эффективна, когда заранее известно примерное количество элементов и их структура. В этом таком случае можно заранее подобрать нужный размер таблицы и подходящую хеш-функцию.

Кроме того, хеш-таблица является самой эффективной по памяти (эффективнее, чем АВЛ-дерево или ДДП `в ~5 раз`), поскольку при использовании закрытой адресации она, по сути, является одномерным динамическим массивом, где каждая ячейка хранит только значение, в отличие от деревьев. *(Однако, здесь кроется и недостаток: при слишком большом размере таблицы, может не найтись подходящего участка памяти и ОС откажет в выделении)*

Деревья

ДДП эффективнее АВЛ-дерева по удалению, но только на неотсортированных данных.

- Так при размерности `512` элемента удаление элемента из ДДП происходит быстрее, чем в АВЛ-дереве `на ~130%` или `в ~2,3 раза`
- Однако на отсортированных данных при той же размерности удаление эл-та из АВЛ-дерева происходит `на ~510%` или `в ~6,1 раза` быстрее.

АВЛ-дерево требует балансировку, на что тратится время, однако, в то же время, благодаря балансировке, АВЛ-дерево гарантирует одинаково эффективный поиск на любых данных, в отличие от ДДП, которое резко теряет свою эффективность на отсортированных данных. Поэтому, предпочтительнее выбирать АВЛ-дерево.

По памяти немного более эффективным (`на ~5%`) должно быть ДДП, поскольку оно не хранит высоту узла, необходимую для балансировки. Однако из-за выравнивания, мы получим одинаковую эффективность. (См. значения без упаковки и с упаковкой).

Файл

Если же сравнивать ДДП, АВЛ-дерево, хеш-таблицу и файл, то мы видим, что файл сильно (более чем `в 1000 раз`) проигрывает другим структурам данных по времени, поскольку происходит обращение к внешнему устройству, однако это позволяет практически не ограничиваться по памяти (так как размер ОП, отведённый программе сильно меньше, чем количество памяти на внешнем устройстве).

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?

Если дерево имеет списочную структуру, то память выделяется под каждый узел отдельно, а если дерево представлено массивом, то один раз на какой-то фиксированный размер.

3. Какие стандартные операции возможны над деревьями?

Обход вершин, поиск по дереву, включение узла в дерево, удаление узла.

4. Что такое дерево двоичного поиска?

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

5. Чем отличается идеально сбалансированное дерево от АВЛ дерева?

- ИДС - дерево, у которого количество вершин в левом и правом поддеревьях отличается не более, чем на 1.
- АВЛ дерево - двоичное дерево, у которого высота двух поддеревьев каждого узла дерева отличается не более чем на 1.

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

- Из-за условия балансировки кол-во сравнений в АВЛ-дереве меньше, чем в ДДП.

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

Это структура данных, в основе которой лежит массив, но индекс, по которому располагается элемент, зависит непосредственно от значения элемента.

Функция, которая реализует отображение из множества значений элементов в множество индексов называется хеш-функцией.

8. Что такое коллизии? Каковы методы их устранения.

Коллизия - ситуация, когда разным ключам (ключ вычисляется из значения элемента) соответствует одно значение хеш-функции. Для устранения или минимизации числа коллизий можно попробовать подобрать другую хешфункцию. Если коллизия всё же возникла, то используется открытое или закрытое хеширование: при открытом для каждого индекса выстраивается цепочка из элементов, ключ которых соответствует данному индексу (то есть элементы помещаются в список, а указатель на голову хранится в хештаблице); при закрытом - если ячейка с вычисленным индексом занята, то просматриваются следующие записи таблицы по порядку (с шагом 1), до тех пор, пока не будет найден ключ или пустая позиция в таблице. При этом, если индекс следующего просматриваемого элемента определяется добавлением какого-то постоянного шага (от 1 до n), то данный способ разрешения коллизий называется линейной адресацией, существует также квадратичная адресация, при которой для вычисления шага применяется формула: h=h+a^2, где a – это номер попытки поиска ключа.

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

При большом количестве коллизий (в частности, когда количество элементов меньше размера таблицы)

10. Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска и в хеш-таблицах См. «Описание алгоритма»