**Типы и Структуры Данных**

**Oтчет**

## *Работа № 6:* Обработка деревьев, хеш-функций

***Студент: Нгуен Фыок Санг***

***Группa: ИУ7И-36Б***

*2019*

1. **Oписание условия задачи:**

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

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 дерево, 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.

**II. Oписание ТЗ:**

1. **Исходные данные**: Файл содержит целые числа
2. **Результат**: результаты поиска элементов
3. **Задачи, реализуемой программой**:
   * Добавить узел в ДДП-дерево
   * Удалить узел из ДДП-дерева
   * Поиск узла в ДДП-дереве
   * Обход деревьев
   * Построение ДДП-дерева
   * Построить хэш-таблицу
   * Поиск в хеш-таблице
   * Построениe ***АВЛ*** дерево

**III. Описание СД:**

1. **Для ввода:** массив целых чисел размером n
2. **Для обработки:** 
   * Использовать 2 связанных списка (2 бинарныe дерева)

***структура узла:***

struct node

{

long int data; //Data element

struct node \* left; //Pointer to left node

struct node \* right;//Pointer to right node

};

**Структура хеш-таблицы**: массив с его начальным состоянием, содержащий все **-1** (пустой массив)

**IV. Oписание алгоритма:**

1. ***Добавить узел (P) в* ДДП*-дерево T*.**
   1. Если дерево пусто, заменить T 🡨 P и остановиться.
   2. Иначе сравнить P с T.
      1. Если P > T, рекурсивно добавить (P) в правое поддерево Т.
      2. Если P < T, рекурсивно добавить (P) в левое поддерево Т.
2. ***Удалить узел (P) из* ДДП*-дерева (T)***
   1. Если дерево T пусто, остановиться;
   2. Иначе сравнить P u T.
      1. Если P > T, рекурсивно удалить P из правого поддерева Т;
      2. Если P < T, рекурсивно удалить P из левого поддерева Т;
      3. Если P = T, то необходимо рассмотреть три случая.
         1. Если обоих детей нет, то удаляем текущий узел и обнуляем ссылку на него у родительского узла
         2. Если одного из детей нет, то значения полей ребёнка (S = P->next) ставим вместо соответствующих значений корневого узла, затирая его старые значения, и освобождаем память, занимаемую узлом S
         3. Если оба ребёнка присутствуют, то
            1. Найти самый левый узел правого поддерева (или самый правый узел левого поддерева) дерева T (S)
            2. Изменить значение двух узлов (P), (S).
            3. Удалить узел (S) (используя два вышеупомянутых случая)
3. ***Поиск узла (P) в* ДДП*-дереве (T)***
   1. Если дерево пусто, сообщить, что узел не найден, и остановиться.
   2. Иначе сравнить (P) со значением узла T.
      1. Если P = T, выдать ссылку на этот узел и остановиться.
      2. Если P > T, рекурсивно искать P в правом поддереве Т.
      3. Если P < T, рекурсивно искать P в левом поддереве Т.
4. ***Обход деревьев***
   1. INFIX\_TRAVERSE (T) — обойти всё дерево, следуя порядку (левое поддерево, **вершина**, правое поддерево).
      1. Если дерево пусто, остановиться.
      2. Иначе
         1. Рекурсивно обойти левое поддерево Т.
         2. Действие к началу
         3. Рекурсивно обойти правое поддерево Т.
   2. PREFIX\_TRAVERSE (T) — обойти всё дерево, следуя порядку (**вершина**, левое поддерево, правое поддерево).
   3. POSTFIX\_TRAVERSE (T) — обойти всё дерево, следуя порядку (левое поддерево, правое поддерево', **вершина**).
5. ***Построение* ДДП*-дерева:*** 
   1. Добавить все узлы в дерево
6. ***Построить хэш-таблицу:***
   1. Инициализировать массив со всеми элементами, равными -1
   2. Конвертировать элемент через хеш-функцию
   3. Сохранить элемент в хеш-таблицу
   4. ***Обработкa коллизии:*** найти новое место (по функции) для хранения ключей,
7. ***Поиск в хеш-таблице***
   1. Конвертировать элемент через хеш-функцию (hash-func)
   2. Пока (элемент не найден & элемент не пуст ( = -1):
      1. Перейти к следующей позиции (по функции: index = hash-func(index + h2)
      2. h 🡨 h + 1
8. ***Построениe АВЛ дерево***
   1. Обход деревьев INFIX\_TRAVERSE: Получить массив по возрастанию.
   2. Выбрать одну вершину в качестве корня;
   3. Рекурсивно построить левое поддерево с nl = n div 2 узлами;
   4. Рекурсивно построить правое поддерево с nr = n – nl 1 узлами.

* **Вывод:**
  + ***Использовать хеш-таблицу:***
    - Скорость поиска элементов быстрее.
    - Память используется неоптимально (большая часть памяти может не использоваться).
    - Подходит для поиска с точными значениями, трудно найти значения в диапазонах.
  + ***Дерево:***
    - Быстрая скорость поиска (хуже, чем хеш-таблица).
    - Может искать значения в диапазонах.
    - Оптимальное использование памяти (хотя для сохранения ссылки требуется больше памяти, требуется только достаточно памяти).
    - Поскольку дерево AVL короче дерева BST, в общем случае дерево AVL выполняет поиск быстрее, чем дерево BST.
    - Однако в некоторых случаях (небольшое количество), дерево BST ищет быстрее.
  + ***Файл:***
    - Скорость поиска медленная
    - Память не нужна для хранения

**V. Набор тестов:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Время работы** | | | | | | | | |
| **N =** | **10** | **100** | **500** | **1000** | **5000** | **10000** | **20000** | **50000** |
| BST | 1 | 2 | 9 | 7 | 18 | 14 | 20 | 12 |
| AVL | 1 | 5 | 6 | 6 | 9 | 12 | 8 | 7 |
| HASH | 1 | 1 | 1 | 0 | 1 | 2 | 1 | 1 |
| FILE | 6800 | 7100 | 7500 | 20500 | 14300 | 173200 | 12260 | 15080 |
| **Количество сравнений** | | | | | | | | |
| **N =** | **10** | **100** | **500** | **1000** | **5000** | **10000** | **20000** | **50000** |
| BST | 1 | 5 | 12 | 10 | 17 | 20 | 23 | 16 |
| AVL | 2 | 7 | 9 | 7 | 13 | 13 | 12 | 14 |
| HASH | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| FILE | 8 | 28 | 35 | 864 | 584 | 9543 | 7721 | 9091 |

**VI. Ответы на вопросы:**

1. ***Что такое дерево?*** 
   * *Дерево* – это нелинейная структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим».
2. ***Какие стандартные операции возможны над деревьями?***
   * Основные операции с деревьями:
     + Обход дерева
     + Поиск по дереву
     + Включение в дерево
     + Исключение из дерева.
3. ***Что такое дерево двоичного поиска?***
   * *Дерево двоичного поиска* – это такое дерево, в котором все левые потомки моложе предка, а все правые – старше.
4. ***Чем отличается идеально сбалансированное дерево от АВЛ дерева?***
   * Дерево идеально сбалансированное - двоичное дерево, у которого **число вершин** в левом и правом поддеревьях отличается не более, чем на единицу.
   * Авл-дерево - двоичное дерево, у **каждого узла дерева высота** двух поддеревьев отличается не более чем на единицу.
5. ***Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска?***
   * Деревья AVL более сбалансированы, чем деревья BST, и, следовательно, высота деревьев AVL будет меньше (или равна) деревьев BST, поэтому в общем случае поиск по деревьям AVL будет быстрее, чем в деревьях BST.
6. ***Что такое хеш-таблица, каков принцип ее построения?***
   * Массив, заполненный в порядке, определенным хеш-функцией, называется *хеш-таблицей*.
   * Принято считать, что хорошей является такая функция, которая удовлетворяет следующим условиям:
     + Функция должна быть простой с вычислительной точки зрения;
     + Функция должна распределять ключи в хеш-таблице наиболее равномерно.
7. ***Что такое коллизии? Каковы методы их устранения.***
   * Может возникнуть ситуация, когда разным ключам соответствует одно значение хеш-функции, то есть, когда

h(K1)=h(K2), в то время как K1 ≠ K2.

Такая ситуация называется ***коллизией***.

* + Существует несколько возможных вариантов разрешения коллизий, которые имеют свои достоинства и недостатки.
    - ***Внешнее*** (открытое) хеширование (метод цепочек).
    - ***Внутреннее*** (закрытое) хеширование (открытая адресация).

1. ***В каком случае поиск в хеш-таблицах становится неэффективен?***
   * Если для поиска элемента необходимо более ***3–4 сравнений***, то эффективность использования такой хеш-таблицы пропадает и ее следует реструктуризировать (т.е. найти другую хеш-функцию), чтобы минимизировать количество сравнений для поиска элемента.
2. ***Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска и в хештаблицах***
   * Эффективность поиска в АВЛ деревьях: log2 n
   * Эффективность поиска в дереве двоичного поиска: в худшем случае (дерево слева или справа): O(n), n - высота дерева
   * Эффективность в поиска хештаблицах: Зависит от хеш-функции и как обрабатывать коллизии, в лучшем случае: O(1)