МГТУ им. Н. Э. Баумана

ИУ 7 – 32

Отчет о лабораторной работе №8

«**Сбалансированные деревья, хеш –таблицы**»

Вариант №8

Исаев Д.С.

# Цель работы

# Построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска и в хеш-таблицах. Сравнить эффективность устранения коллизий при внешнем и внутренним хешировании.

# Условие задачи

Построить хеш-таблицу по указанным данным. Сравнить эффективность поиска в сбалансированном двоичном дереве, в двоичном дереве поиска и в хеш-таблице (используя открытую и закрытую адресацию). Вывести на экран деревья и хеш-таблицу. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если среднее количество сравнений больше указанного. Оценить эффективность использования этих структур (по времени и памяти) для поставленной задачи. Оценить эффективность поиска в хеш-таблице при различном количестве коллизий и при различных методах их разрешения.

Используя предыдущую программу (задача №7), сбалансировать полученное дерево. Вывести его на экран в виде дерева. Построить хеш-таблицу из чисел файла. Осуществить поиск введенного целого числа в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Сравнить время поиска, объем памяти и количество сравнений при использовании различных структур данных.

**Описание**

**Сбалансированное дерево**

Рассмотрим алгоритм построения двоичного дерева. Если при добавлении узлов в дерево мы будем их равномерно располагать слева и справа, то получится дерево, у которого число вершин в левом и правом поддеревьях отличается не более, чем на единицу. Такое дерево называется **идеально сбалансированным**.

Для построения идеально сбалансированного дереваиспользуется рекурсия. Для дерева из **n** узлов, где **nl** - количество узлов в левом поддереве,  **nr** - количество узлов в правом поддереве, алгоритм построения идеально сбалансированного дерева описывается следующим образом:

1. выбрать одну вершину в качестве корня;
2. рекурсивно построить левое поддерево с **nl = n div 2** узлами;
3. рекурсивно построить правое поддерево с **nr = n – nl 1** узлами.

Идеальная балансировка дает наименьшую высоту дерева, а так как высота дерева определяет длину пути поиска в нем, то, следовательно, и укорачивает поиск. Но поддержание идеальной сбалансированности дерева при включении или исключении элемента – это достаточно сложная процедура.

Адельсон-Вельский и Ландис сформулировали менее жесткий критерий сбалансированности таким образом: двоичное дерево называется сбалансированным, если у каждого узла дерева высота двух поддеревьев отличается не более чем на единицу. Такое дерево называется *АВЛ-деревом.*

Использование деревьев для поиска информации достаточно эффективно (трудоемкость – O(log2n))

Поиск в сбалансированном дереве, в АВЛ-дереве быстрее поиска в двоичном дереве и разность в скорости зависит от степени ветвления деревьев.

**Хеш-таблицы**

Можно ли создать еще более эффективную структуру или метод, позволяющий лучше осуществлять поиск информации? Для этого было бы хорошо по значению ключа сразу определять индекс элемента массива, в котором хранится информация. То есть необходимо создать такую функцию, по которой можно вычислить этот индекс. Такая функция называется *хеш-функцией*(от англ. to hash - крошить, рубить) и она ставит в соответствие каждому ключу ki индекс ячейки j, где расположен элемент с этим ключом, таким образом:

h(ki) = j, если j=(1,m),

где j принадлежит множеству от 1 до m, а m. – размерность массива.

Массив, заполненный в порядке, определенным хеш-функцией, называется *хеш-таблицей*. Минимальная трудоемкость поиска в хеш-таблице равна О(1)!

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.

Первый метод – внешнее (открытое) хеширование (метод цепочек)

В случае, когда элемент таблицы с индексом, который вернула хеш-функция, уже занят, к нему присоединяется связный список. Таким образом, если для нескольких различных значений ключа возвращается одинаковое значение хеш-функции, то по этому адресу находится указатель на связанный список, который содержит все значения. Поиск в этом списке осуществляется простым перебором, так как при грамотном выборе хеш-функции любой из списков оказывается достаточно коротким.

Другой путь решения проблемы, связанной с коллизиями – внутреннее (закрытое) хеширование (открытая адресация). Оно, состоит в том, чтобы полностью отказаться от ссылок. В этом случае, если ячейка с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку (с шагом 1), до тех пор, пока не будет найден ключ K или пустая позиция в таблице.

**Реализация**

**Открытое хеширование:**

Я применил двойное хеширование:

**Hash = (HashFunction1(Key) + Offset \* HashFunction2(Key)) % Size;**

Где Offset – текущее смещение относительно первоначальной ячейки, оно сначала равно 0 и увеличивается, если вычисленная ячейка занята.

При вычислении следующей ячейки имеем зависимость от 2 переменных, что позволяет достаточно равномерно распределить элементы по хеш-таблице.

Хеш-функции:

private int HashFunction1(Info Key)

        {

            int a = 3, b = 7;

            int p = 211;

            return ((a \* Key.Data + b) % p) % Size;

        }

private int HashFunction2(Info Key)

        {

            int a = 5, b = 11;

            int p = 199;

            return 1 + ((a \* Key.Data + b) % p) % (Size - 1);

        }

Все коэффициенты в них - простые числа, а сами функции являются универсальными функциями о очень равномерно отображают множество ключей в множество ячеек таблицы.

При коллизиях элементы идут на другие ячейки, которые вычисляются хеш-функцией.

**Закрытое хеширование:**

Применяется универсальное хеширование – функция:

private int HashFunction(int Key)

        {

            int a = 37, b = 101;

            int p = 211;

            return ((a \* Key + b) % p) % Size;

        }

Коллизии разрешаются благодаря спискам, т.е. если элементы попали в одну ячейку, они добавляются в список этой ячейки. Это может замедлить программу при большом числе коллизий.

**Дерево и сбалансированное дерево:**

Реализация дерева описана в предыдущей лабораторной работе.

Сбалансированное дерево отличается лишь порядком следования вершин, реализация одинакова. Была написана функция балансировки дерева.

**Результаты**

В результате работы программы были получены следующие данные:

Количество повторений = 1000000.

**Бинарное дерево**

-Время поиска = 0,45

-Занимаемая память = 156 байт

**Сбалансированное бинарное дерево**

-Время поиска = 0,30

-Занимаемая память = 156 байт

**Хеш-таблица с открытой адресацией**

-Время поиска = 0,18

-Занимаемая память = 124 байт

**Хеш-таблица с закрытой адресацией**

-Время поиска = 0,09

-Занимаемая память = 372 байт.

Из приведённых данных видно, что время сбалансированном дереве значительно меньше, чем в несбалансированном. А в хеш-таблице еще меньше и равно О(1). Причем поиск в таблице с закрытой адресацией быстрее из-за сложности вычисления хеш-функций в открытом хешировании.

Закрытое хеширование неэффективно при большом количестве коллизий, т.е. при сильно ограниченных размерах таблицы.

Со временем при наполнении дерева, если элементы поступают не в случайном порядке, дерево «дестабилизируется» и время поиска в нем стремится к времени поиска в обычном односвязном списке, поэтому важно иметь сбалансированное дерево, или алгоритмы балансировки.

**Вывод:**

Двоичные деревья поиска позволяют осуществить быстрый поиск, сбалансированные двоичные деревья – еще быстрее. Время поиска равно O(высота дерева). Самый быстрый поиск в хеш-таблицах – за время О(1).