Федеральное агентство связи
ГОУ ВПО «Сибирский государственный университет телекоммуникаций и информатики»
Кафедра вычислительных систем

Практические задания по дисциплине «Программирование на языке высокого уровня»

семестр 2

2010/2011 учебный год

Преподаватели:
Старший преподаватель Кафедры ВС – Поляков Артём Юрьевич
Доцент Кафедры ВС – Молдованова Ольга Владимировна

Лабораторная работа №3

Структуры данных. Хеш-таблицы.

Цель работы:

Методические указания

Частой задачей, которая возникает при создании программного обеспечения (ПО), является формирование динамических множеств, поддерживающих только словарные операции (добавление, удаление и поиск элементов) [1]. Под динамическим множеством будем понимать набор элементов, количество и состав которого может постоянно изменяться в процессе работы программы. Каждый элемент динамического множества содержит специальное поле, называемое *ключом*, по которому осуществляется поиск. Одним из способов организации таких структур данных является хеширование, а соответствующие структуры данных называют *хеш-таблицами*.

Хеш-таблица — это структура данных, которая позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу. Хеш-таблица представляет собой массив $T[m]$, где m называют размером таблицы. Элементу, который описывается ключом $k \in U$ (U – множество всех возможных ключей), ставится в соответствие ячейка $T[h(k)]$ (рис. 1), где $h : U \rightarrow \{0, 1, \ldots, m - 1\}$ – функция, называемая *хеш-функцией* (*hash function*, англ. “to hash” – мелко порубить, помешивая). Число $h(k)$ называют *хеш-значением* (*hash value*).

Хеш-таблица позволяет создавать *ассоциативные массивы*, в которых аргументом операции индексации служит не целое число, а произвольный тип данных. Например, a[“hello”] = 10. В ряде языков (PHP, Shell script) существует изначальная поддержка таких массивов.

Так как количество элементов в множестве U обычно значительно больше m ($|U| >> m$), то существуют такие ключи $k_1, k_2 \in U$, для которых значения хеш-функций совпадают: $h(k_1) = h(k_2)$. Такая ситуация называется *коллизией* (*collision*) или *столкновением*. Для разрешения коллизий существует два подхода: с *применением цепочек* (линейных списков) и с использованием *открытой адресации*.

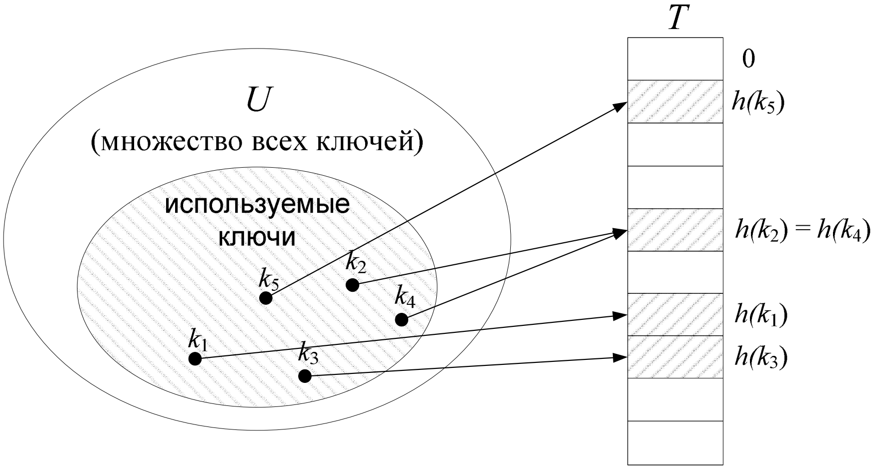

Рис. 1. Отображение ключей из множества U всех ключей на позиции хеш-таблицы T

Хеш-функции

Хорошая хеш-функция должна (приближенно) удовлетворять предположениям *равномерного хеширования*: для очередного ключа все m хеш-значений должны быть равновероятны. Однако на практике очень редко известна вероятность появления выбора ключа $k \in U$, а также редко можно считать ключи независимыми. Поэтому при выборе хеш-функции пользуются различными эвристиками, основанными на специфике задачи.

Обычно предполагают, что область определения хеш-функции – множество целых неотрицательных чисел. Если ключи не являются натуральными числами, их обычно можно преобразовать к целому виду. Например, последовательность символов можно интерпретировать как числа (ASCII-коды), записанные в системе счисления с подходящим основанием (количество различных ASCII-кодов). Пусть дана строка pt, символы имеют следующие коды: ‘p’ – 112, ‘t’ – 116. Количество различных кодов в одном байте – 256. Тогда строка pt может быть преобразована к числу 112*256 + 116 = 28788. В общем случае строка s длины l может быть преобразована к целому числу следующим образом:

```
int sum = 0;
for(i = 0; i < l; i++){
    sum += s[i]*D;
},
```

где D – целое положительное число. Если $D < 256$ то преобразование строки в число будет с потерями, т.е. для различных строк могут быть получены одинаковые значения *sum*. С другой стороны данный подход позволяет хранить числа, соответствующие строкам в ограниченных ячейках памяти, выделяемых для целого типа данных. Фактически данный подход может рассматриваться как хеширование.

Деление с остатком

Одной из распространенных хеш-функций является функция, использующая *деление с остатком*. В этом случае ключу k ставится в соответствие остаток от деления k на m (размер хеш-таблицы):

$$h(k) = k \bmod m.$$

Например, если $m = 12$, а $k = 100$, то хеш-значение $h(100) = 4$.

Для данной функции важным является выбор значения m. Например, выбор m числом, являющимся степенью двойки ($m = 2^p$), приводит к тому, что $h(k)$ – это просто набор младших битов числа k, однако часто комбинации младших битов встречаются с различной вероятностью, что не обеспечивает равномерного хеширования. Пусть, например, $m = 2^4 = 10000b$, тогда для чисел 5 (0101b), 21 (1 0101b), 37 (10 0101b), 53 (11 0101b), 69 (100 0101b) и т.д. значения $h(k)$ будут совпадать (символ b после числа обозначает представление в двоичном виде).

Статистически хорошие результаты получаются, если в качестве m выбирается простое число, далеко отстоящее от степеней двойки. Например при размещении 2000 в качестве m будет эффективно выбрать число 701, которое является простым и далеко (на равном расстоянии) отстоит от чисел, являющихся степенями двойки ($512 = 2^9 < 701 < 2^{10} = 1024$). При таком выборе m с высокой вероятностью количество элементов, отображающихся на одно хеш-значение (конфликтующих), будет в среднем равно 3 (2000/701 ~ 2,85).

Умножение

Построение хеш-функции методом умножения (multiplication method) состоит в следующем. Пусть количество хеш-значений равно m и дана константа $A \in (0,1)$, тогда

$$h(k) = [m \cdot (k \cdot A - [k \cdot A])],$$

где $[x]$ – ближайшее к x целое число такое, что $[x] \le x$ (например $[3{,}9888] = 3$, $[4{,}012] = 4$). Другими словами $[x]$ – это целая часть числа x.

Реализовать такую функцию на языке Си можно следующим образом:

```
int hash_multiply(int k, int m, float A){
    int tmp = k*A; // tmp = [k·A]
    int h = m*(k*A - tmp);
    return h;
}
```

Обратите внимание, что для реализации операции $[x]$ хорошо подходит преобразование вещественного числа в целое. Данную функцию можно записать одним выражением (альтернативная запись):

```
int hash_multiply(int k, int m, float A){
    return (int)(m*(k*A - (int)(k*A)));
}
```

Достоинством данного метода является то, что качество хеш-функции мало зависит от выбора m. Метод умножения работает при любом выборе константы A, однако некоторые значения могут быть лучше других. Оптимальность выбора A зависит от хешируемых данных. Например, в книге Кнута [2] рекомендуется использовать следующее значение:

$$A = (\sqrt{5} - 1)/2 = 0{,}6180339887\,.$$

Подходы к обработке коллизий

Как было сказано выше, существует два основных подхода к обработке коллизий, возникающих при добавлении элементов: с *применением цепочек* (линейных списков) и с использованием *открытой адресации*. Рассмотрим каждый из них подробнее.

Разрешение коллизий с помощью цепочек

Технология *сцепления элементов* (chaining) состоит в том, что элементы множества, которым соответствует одно и то же хеш-значение, связываются в цепочку-список (рис. 2).

Словарные операции в такой таблице реализуются следующим образом:

1. Добавление элемента выполняется путем его вставки первым в имеющийся список. Эта операция имеет константную вычислительную сложность – $O(1)$.

2. Поиск элемента по ключу k выполняется в списке, который соответствует хеш-значению $h(k)$ для искомого ключа (в списке $T[h(k)]$). Вычислительная сложность данного шага линейна ($O(l)$, где l – количество элементов в списке $T[h(k)]$). Результатом поиска является указатель x на найденный элемент или нулевой указатель (NULL) в случае его отсутствия.

3. Удаление элемента обычно выполняется с использованием указателя x на удаляемый элемент (который получается с помощью операции поиска). Вычислительная сложность операции удаления константна – $O(1)$ – для двусвязного списка и линейна – $O(l)$ – для односвязных списков (т.к. необходимо найти элемент, предшествующий x).

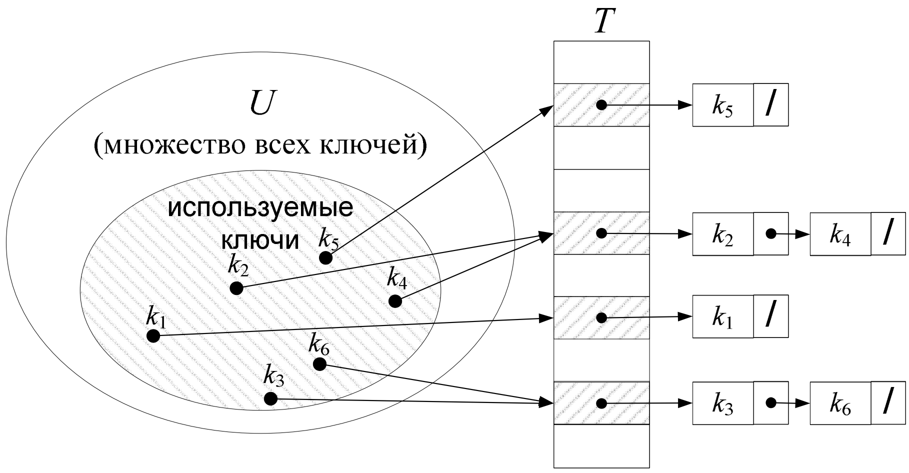

Рис. 2. Разрешение коллизий с помощью технологии цепочек. Каждый элемент $T[j]$ хранит указатель на список, хранящий множество элементов L_j, для каждого $k \in L_j$ справедливо $h(k) = j$. Например, $h(k_2) = h(k_4) = j'$, тогда $L_{j'} = \{k_2, k_4\}$.

Оценка хеш-функций

Для оценки качества распределения, которое обеспечивает хеш-функция, будем рассматривать распределение вероятностей получения цепочек заданной длины. В лабораторной работе предполагается экспериментальная оценка данного параметра. Например, пусть дана хеш-таблица, показанная на рис. 3.

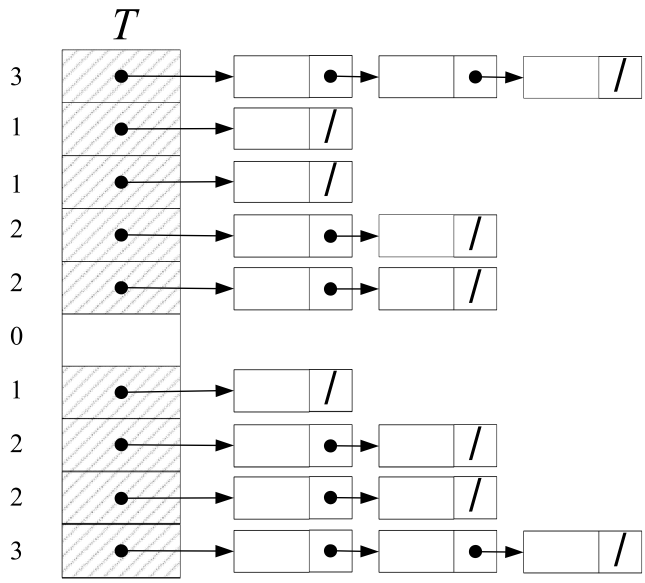

Рис. 3 Пример распределения элементов в хеш-таблице.

Статистическое распределение вероятностей $P(n)$ возникновения цепочек длины n, полученное для хеш-таблицы, приведенной на рис. 3, показано на рис. 4.

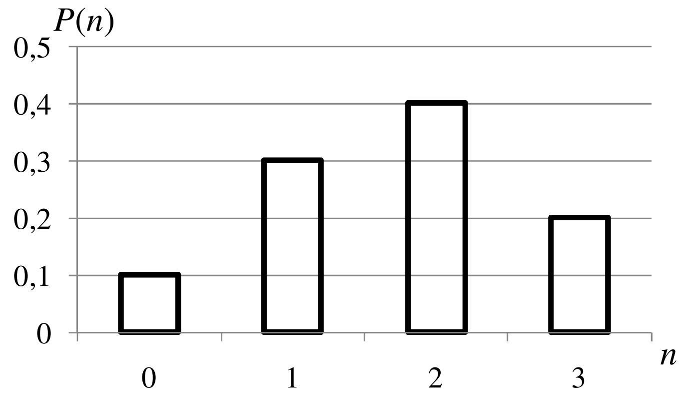

Рис. 4. Распределение вероятности $P(n)$ формирования цепочек длины n

Разрешение коллизий с помощью открытой адресации

При *открытой адресации* (open addressing) все записи хранятся в самой хеш-таблице: каждая ячейка содержит либо элемент множества, либо нулевой элемент NIL, отличный от любого из возможных ключей. Поиск заключается в том, чтобы организовать просмотр элементов таблицы в определенном порядке, пока не будет найден искомый или нулевой элемент. Количество хранимых элементов n не может превышать размер таблицы m. Преимуществом открытой адресации по сравнению с применением цепочек является отсутствие дополнительных накладных расходов памяти, связанных со служебными полями списков.

При добавлении нового элемента в таблицу она просматривается до тех пор, пока не будет найдено свободное место. В случае обычного массива данный подход потребует $O(n)$ просмотров, где n – количество элементов, присутствующих в T на момент вставки. В хеш-таблице с открытой адресацией поиск производится с помощью хеш-функции, которая принимает на вход не один, а два аргумента: ключ и номер попытки (нумерация с нуля). Последовательность проб для фиксированного ключа k выглядит следующим образом:

$$\langle\, h(k, 0), h(k, 1), \ldots, h(k, m-1) \,\rangle.$$

Функция h должна быть такой, чтобы каждое из чисел 0, 1, …, $(m-1)$ (хеш-значений таблицы T) встретилось в этой последовательности <u>ровно</u> один раз (для каждого ключа должны быть доступны все позиции таблицы).

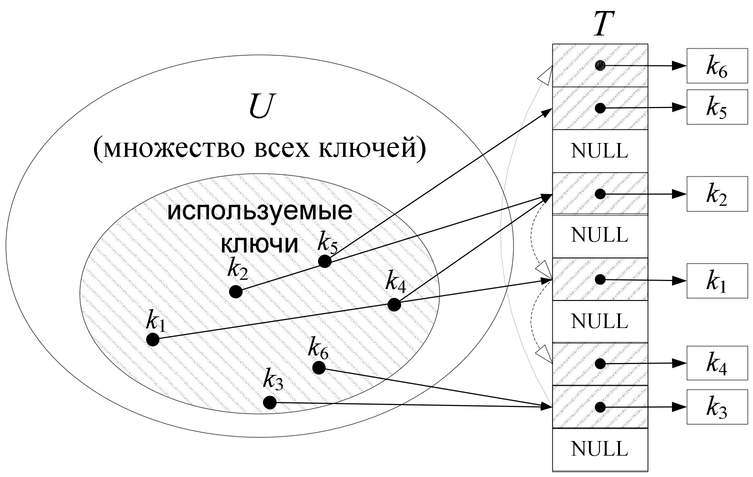

Рис. 5. Реализация хеш-таблицы с открытой адресацией

На рис. 5 показан пример реализации хеш-таблицы с открытой адресацией с использованием указателей. В качестве значения NIL выбран нулевой указатель (NULL). Видно, что при добавлении элемента k_4 успешной является только третья проба:

1) $h(k_4, 0) = h(k_2, 0)$, поэтому $T[h(k_4, 0)] \neq \text{NULL}$;

2) $h(k_4, 1) = h(k_1, 0)$, поэтому $T[h(k_4, 1)] \neq \text{NULL}$;

3) $h(k_4, 2)$ – уникален ($T[h(k_4, 2)]$ = NULL), поэтому k_4 записывается в позицию $T[h(k_4, 2)]$. Для k_6 успешной является уже вторая проба.

Поиск элемента k в такой таблице осуществляется аналогично включению k в T. Производится просмотр элементов таблицы T в следующем порядке:

$$\langle\, T[h(k, 0)], T[h(k, 1)], \ldots, T[h(k, i)] \,\rangle,$$

где $i < (m - 1)$. При этом для любых элементов $T[h(k, j)]$ таких, что $j < i$, справедливо $T[h(k, j)] \neq k$, а для элемента $T[h(k, i)]$ справедливо одно из трех предположений:

1) $T[h(k, i)]$ = NIL, в этом случае искомый элемент в таблице T <u>отсутствует</u>;

2) $T[h(k, i)] \neq$ NIL, $T[h(k, i)] \neq k$ и $i = (m - 1)$, т.е. было выполнено m проб, таблица T заполнена на 100% и <u>не содержит</u> искомый элемент;

3) $T[h(k, i)] = k$ – искомый элемент <u>найден</u> в позиции $h(k, i)$ после i проб.

Рассмотренная организация таблицы имеет один существенный недостаток – она не позволяет удалять элементы. Простое обнуление элемента T в соответствующей позиции приведет к тому, что элементы, которые были добавлены после удаляемого (т.е. когда его позиция была занята), станут недоступны. Например, при удалении элемента k_1 на рис. 5 элемент k_4 перестанет быть доступным, так как при его поиске для позиции элемента $T[h(k_4, 1)]$ выполнится условие $T[h(k_4, 1)]$ = NIL и поиск завершится неудачно в соответствии с предположением 1), изложенным выше.

Одним из возможных решений данной проблемы является запись на место удаленного элемента не NIL, а специализированного значения DELETED (удален). При добавлении элемента ячейка с данным значением рассматривается как свободная, а при поиске – как занятая, но не соответствующая искомому элементу (т.е. поиск будет продолжен).

Для рассмотренной выше реализации открытой адресации с помощью указателей может использоваться специальная ячейка, для которой выделяется отдельная память, и она не используется для хранения полезных данных (аналогично элементу-пустышке в линейных списках).

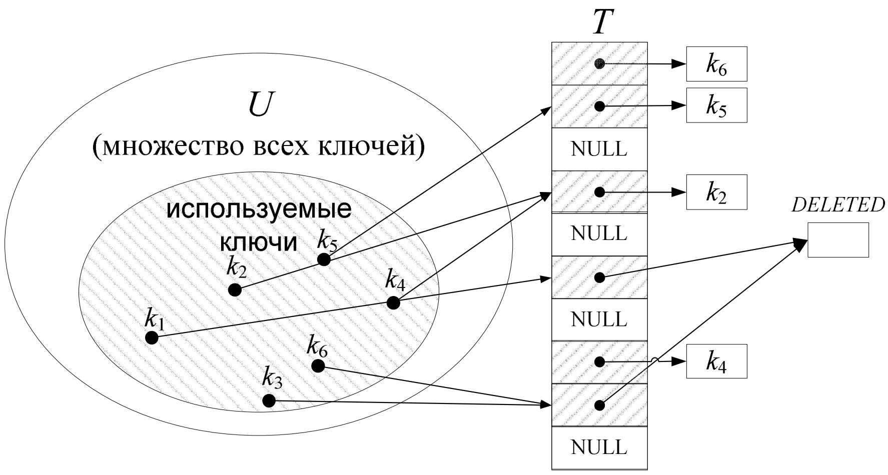

Рис. 4. Применение специального элемента для обозначения удаленных элементов.

На рис. 4 показан пример использования значения DELETED для обозначения того, что элементы k_1 и k_3 были удалены.

Наиболее распространенными являются следующие три способа вычисления последовательности проб: линейный, квадратичный и двойное хеширование. Рассмотрим каждый из них подробнее.

Линейная последовательность проб

Пусть h' – обычная хеш-функция (одна из рассмотренных в разделе «хеш-функции»). Хеш-функция, определяющая *линейную последовательность проб* (linear probing) задается следующим образом:

$$h(k, i) = (h'(k) + i) \bmod m$$

Такая функция обеспечивает переход к первой ячейке $T[h(k,0)] \sim T[h'(k)]$ по ключу k, а все остальные значения перебираются последовательно: $T[h(k, 1)] \sim T[h'(k) + 1]$, $T[h(k, 2)] \sim T[h'(k) + 2]$, … (после $T[m - 1]$ выполняется переход к $T[0]$).

Недостатком данного подхода является то, что он часто приводит к образованию кластеров (длинных последовательностей идущих подряд элементов), которые замедляют поиск в таблице.

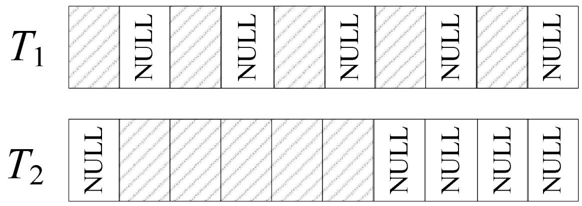

Рис. 6 Хеш-таблицы T_1 и T_2, содержащие одинаковое количество элементов. В T_2 наблюдается образование кластеров.

На рис. 6 приведен пример таблицы (T_2), содержащей один кластер. Поиск несуществующего элемента в T_2 потребует просмотра всех элементов, в то время как в T_1 потребуется просмотр только одного элемента.

Тенденция к образованию кластеров объясняется тем, что при вставке очередного элемента в область T, уже занятую другими элементами, он будет добавлен в первую свободную ячейку после непрерывной последовательности, тем самым удлиняя ее.

Квадратичная последовательность проб

Функция, определяющая *квадратичную последовательность проб* (quadratic probing), задается следующей формулой:

$$h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m,$$

где h' – обычная хеш-функция, а $c_1, c_2 \neq 0$ – некоторые константы. Как и при линейном методе пробы начинаются с ячейки $T[h'(k)]$, однако просмотр ячеек выполняется не подряд (номер исследуемой ячейки квадратично зависит от номера попытки). Для того чтобы обеспечить последовательный просмотр всех ячеек значения m, c_1, c_2 необходимо выбирать определенным образом. Например, одним из распространенных вариантов является использование в качестве m числа, являющегося степенью двойки ($m = 2^p$), и коэффициентов $c_1 = c_2 = 0{,}5$.

Использование данного подхода позволяет избежать тенденции к появлению кластеров, однако данный эффект проявляется в более мягкой форме образования *вторичных кластеров*.

Двойное хеширование

Двойное хеширование (double hashing) – один из лучших методов открытой адресации. Перестановки индексов, возникающие при двойном хешировании, обладают многими свойствами, присущими равномерному хешированию. Функция h при таком подходе имеет следующий вид:

$$h(k, i) = (\ h_1(k) + i \cdot h_2(k)\ ) \bmod m,$$

где h_1, h_2 – обычные хеш-функции. Таким образом, последовательность проб при работе с ключом k представляет собой арифметическую прогрессию по модулю m с первым членом $h_1(k)$ и шагом $h_2(k)$.

Чтобы последовательность проб покрывала всю таблицу, значение $h_2(k)$ должно быть взаимно простым с m. Простым способом добиться такого соотношения является выбрать $m = 2^p$, а функцию h_2 выбрать такой, чтобы она принимала только нечетные значения. Другим вариантом является выбор в качестве m простого числа, а в качестве h_2 – целых положительных чисел, меньших m:

$$h_1(k) = k \bmod m,$$
$$h_2(k) = 1 + (\ k \bmod m'\ ),$$

где m' – немного меньше m (например, $m' = m - 1$ или $m - 2$).
Например, если $m = 701$, $m' = 700$ и $k = 123456$, то $h_1(k) = 80$, $h_2(k) = 270$.

Литература

1. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Глава 11. Хеш-таблицы. // Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005. — 1296 с. — ISBN 5-8459-0857-4.

2. Кнут Д. Искусство программирования для ЭВМ. т.3. Сортировка и поиск: Пер. с англ. – М.: Мир, 1978. – 845 с., ил.

Задание на лабораторную работу

Обработка текстовой информации

При выполнении лабораторных для оценки и сравнения работы хеш-таблиц необходимо использовать тестовые данные, размещенные на странице предмета. Тестовые данные содержат тексты на английском языке из различных областей. Целью лабораторной работы является анализ частоты встречаемости слов во входном тексте с применением хеш-таблицы, соответствующей заданию.

Входной текст разбивается на слова, очищается от знаков препинания и помещается в хеш-таблицу. Каждый элемент хеш-таблицы – это слово и поле-счетчик, содержащее количество раз, которое данное слово встретилось в тексте. При вставке нового элемента в таблицу, если данный элемент уже присутствует в ней, необходимо увеличить соответствующий счетчик. В противном случае создается новый элемент. Программа должна выдавать 10 слов, которые встречаются чаще всего во входном тексте.

Также каждый вариант содержит дополнительное задание.

Операции вставки, поиска и удаления элементов должны быть реализованы в виде отдельных функций.

Для оценки времени работы некоторой функции func() использовать функцию gettimeofday(). Для успешной компиляции необходимо подключить файл time.h (#include <sys/time.h>). Данный файл содержит определение структуры timeval:

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

Вычисление времени работы func() определяется по аналогии со следующим листингом:

```
#include <stdio.h>
#include <sys/time.h>

void func()
{
  sleep(1);
}

int main()
{
  struct timeval tv1, tv2;
  float start, end, func_time;

  gettimeofday(&tv1,NULL);
  func();
  gettimeofday(&tv2,NULL);
  tv2.tv_sec -= tv1.tv_sec;
  tv1.tv_sec = 0;
  start = tv1.tv_sec + (float)tv1.tv_usec*1E-6;
  end = tv2.tv_sec + (float)tv2.tv_usec*1E-6;
  func_time = end - start;
  printf("func_time = %f\n",func_time);
}
```

Листинг 1. Определение времени работы функции func()

Варианты хеш-таблиц и дополнительные задания

Анализ хеш-функций

Использовать хеш-таблицы с применением цепочек для разрешения коллизий (на базе односвязных списков).

1. Используемая хеш-функция – деление с остатком. Построить статистическое распределение вероятности $P(n)$ формирования цепочек длины n для каждого из текстов при m = 512, 701, 1024, 1579, 2048.

2. Используемая хеш-функция – умножение. Построить статистическое распределение вероятности $P(n)$ формирования цепочек длины n для каждого из текстов при A = 0.23, 0.517, 0.618, 0.85 и m = 2048.

3. Используемая хеш-функция – умножение. Построить статистическое распределение вероятности $P(n)$ формирования цепочек длины n для каждого из текстов при $A = 0.618$ и m = 512, 701, 1024, 1579, 2048.

4. Реализовать две хеш-функции: деление с остатком и умножение. Построить и сравнить статистические распределения вероятности $P(n)$ формирования цепочек длины n для каждого из текстов. Параметры хеш-функций:
– деление с остатком: m = 1579;
– умножение: m = 1579, A = 0.618.

Анализ хеш-таблиц с применением цепочек

Реализовать хеш-таблицу, использующую технологию сцепления элементов для разрешения коллизий. Тип списка и хеш-функция определяются вариантом задания. Для реализованной хеш-таблицы необходимо определить среднее, максимальное и минимальное времена вставки, поиска и удаления элемента в реализованной таблице при m = 701, 1579 и 3083 для каждого из данных тестовых текстов. Определение времени работы операций определять по аналогии с Листингом 1.

Слова должны удаляться в порядке, обратном их поступлению. Для реализации поиска и удаления слов они должны сохраняться при вводе как в хеш-таблице, так и в динамически расширяемом массиве строк, откуда и будут выбираться в обратном порядке.

5. Реализовать хеш-таблицу на базе односвязного списка и хеш-функции деления с остатком.

6. Реализовать хеш-таблицу на базе односвязного списка и хеш-функции умножения.

7. Реализовать хеш-таблицу на базе двусвязного списка и хеш-функции деления с остатком.

8. Реализовать хеш-таблицу на базе двусвязного списка и хеш-функции умножения.

Анализ хеш-таблиц с открытой адресацией без возможности удаления элементов

Необходимо реализовать хеш-таблицу с открытой адресацией, удаление элементов в которой не предусмотрено. Хеш-функция и подход к вычислению последовательности проб выбирается в соответствии с вариантом. Необходимо определить производительность таблицы (среднее/максимальное/минимальное время вставки и поиска) при различных уровнях ее заполнения: до 20%, до 40%, до 60%, до 80%, до 100%. Определение времени работы операций определять по аналогии с Листингом 1.

9. Использовать линейный подход к вычислению последовательности проб на базе хеш-функции деления с остатком при $m = 1579, 3083$.

10. Использовать линейный подход к вычислению последовательности проб на базе хеш-функции умножения при $m = 3083, 4096$; $A = 0{,}618$.

11. Использовать квадратичный подход к вычислению последовательности проб на базе хеш-функции деления с остатком при $m = 1579, 3083$; $c_1 = c_2 = 0.5$.

12. Использовать квадратичный подход к вычислению последовательности проб на базе хеш-функции умножения при $m = 3083, 4096$; $A = 0{,}618$.

13. Использовать двойное хеширование для вычисления последовательности проб на базе хеш-функции деления с остатком при $m = 2048, 4096$; в качестве h_1 использовать хеш-функцию умножения ($A = 0{,}618$), в качестве h_2: $h_2(k) = ((k \bmod 3083) \operatorname{div} 2)\cdot 2 + 1$, где $(x \operatorname{div} y)$ – целая часть от деления x на y (в языке Си данная операция эквивалентна целочисленному делению: x/y).

14. Использовать двойное хеширование для вычисления последовательности проб на базе хеш-функции деления с остатком при $m = 1579, 3083$; в качестве h_1 и h_2 использовать следующие функции: $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$, где m' – немного меньше m (например $m' = m - 1$ или $m - 2$).