Лекция 5. Методы быстрого поиска информации в таблицах

5.1. Таблицы

Таблица – это набор элементов, содержащих *ключ* - отличительный признак для поиска элемента, и *тело* (сопутствующую информацию).

Примеры таблиц.

а) *Таблица значений функции*: ключ элемента - аргумент X, тело – значение функции f(X);

б) *Словарь*: ключ - слово, тело - перевод слова;

в) *Телефонный справочник*: ключ - имя владельца, тело - номер телефона;

г) *Таблица имен* транслятора: ключ - имя какого-либо объекта транслируемой программы, тело - его характеристики (тип, размерность, адрес, значение переменной и т. п.).

Везде, где есть поиск информации, используются таблицы. Трансляторы тратят на работу с таблицами до 50 % времени.

Основные операции над таблицами:

1. *Инициализация* (подготовка к работе);
2. *Поиск* элемента по ключу - основная операция (входит в другие операции);
3. *Включение* в таблицу данного элемента;
4. *Исключение* из таблицы элемента с данным ключом;
5. *Изменение* в таблице тела элемента с данным ключом;
6. *Перебор* (например, вывод) элементов таблицы в порядке, определяемом ключами.

Виды таблиц: *статическая* (постоянная) и *динамическая* (меняющаяся при работе программы), *внутренняя* (в оперативной памяти) и *внешняя* (во внешней памяти). Таблицу размещают во внешней памяти, если ее необходимо сохранять после работы программы или она слишком велика для ОЗУ. Рассмотрим основные методы организации внутренних таблиц.

5.2. Линейные таблицы

Линейной (последовательной) называют таблицу, в которой производится *линейный поиск* (последовательный перебор элементов). Линейные таблицы бывают упорядоченными или неупорядоченными в виде вектора или списка. Рассмотрим две из них на примерах.

Таблица в виде вектора

Пример 5.1. ***Неупорядоченная векторная таблица.*** Задача: составить программу подсчета количества повторений каждого слова входного текста. Результат вывести в лексикографическом порядке.

Пример работы:

Вход: to be or not to be

Выход:

Слово	Кол-во
be	2
not	1
or	1
to	2

Для решения задачи нужна таблица, ключом в которой служит слово, тело - количество его повторений. Очередное слово ищется в таблице. Если его там нет, оно включается в таблицу, а если есть, то увеличивается число повторений. Требуемые операции: инициализация, поиск, включение, изменение и вывод.

Для примера используем *неупорядоченную линейную таблицу в виде вектора* (рис. 5.1). Ключи размещены в ней в порядке поступления. Тогда перед распечаткой потребуется еще сортировка (упорядочивание) таблицы.

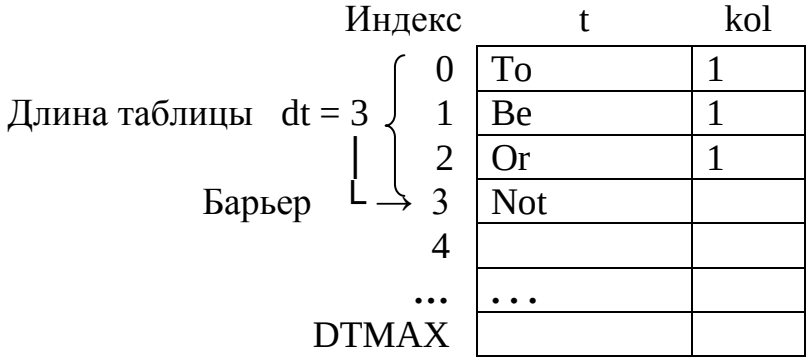


Рис. 5.1. Неупорядоченная таблица в виде вектора с барьером
(при поиске слова not входного текста: to be or not to be)

Ниже показана *нисходящая разработка* программы на псевдокоде. Сначала алгоритм записывается укрупненно с использованием неформальных обозначений данных и операций, например:

Заполнение таблицы t;

Если данные или операции достаточно сложны и не описаны в другом разделе пособия, они затем детализируются через более мелкие данные и операции в одном из фрагментов программы с соответствующим комментарием:

/* Заполнение таблицы t */.

Так продолжается, пока программа не станет достаточно подробной. Этапы разработки отделены горизонтальными линиями.

Алгоритм 5.1. Подсчет количества повторений слов текста с применением неупорядоченной линейной таблицы в виде вектора

```
#include <stdio.h>
void main ()
{   1. Инициализация таблицы;
    2. Чтение текста и заполнение таблицы слов;
    3. Сортировка таблицы по алфавиту;
    4. Вывод таблицы;
}
```

Определение данных:

```
#define DTMAX 1001          /* Max колич-во разных слов + 1 (для барьера)  */
#define DSLMAX 21           /* Максимальная длина слова + 1 (для'\0')      */
   /* Последовательная таблица в виде вектора: t,kol,dt                    */
   /* (представлена параллельными массивами t, kol)                        */
   char t[DTMAX][DSLMAX];       /* Слова                                   */
   int kol[DTMAX];              /* Количество повторений                   */
   int dt;                      /* Длина таблицы (количество слов)         */
```

```
dt = 0;                         /* 1. Инициализация таблицы               */
```

```
int sim;                        /* Текущий символ входного текста         */
char sl[DSLMAX];                /* Текущее слово входного текста          */
        …
/*      2. Чтение текста и заполнение таблицы слов                         */
sim = getchar();                /* 1-й символ текста                      */
while (sim != EOF)
{   Чтение слова sl (и 1-го символа следующего слова);
    Корректировка таблицы для прочитанного слова;
}
```

```
int j;                          /* Индекс символа в слове                 */
        …
/*              Чтение слова sl (и 1-го символа следующего слова)          */
    for (j=0; sim!=' ' && sim!=',' && sim!='.' && sim!='\n'
            && sim!=EOF; sim=getchar())
        if (j < DSLMAX-1)    sl[j++] = sim;
    sl[j] = '\0';                           /* Признак конца слова        */
     /*                 Пропуск разделителей слов                         */
    while (sim==' ' || sim==',' || sim=='.' || sim=='\n')
            sim = getchar();
```

```
int i;                          /* Индекс текущего элемента таблицы     */
    /* Корректировка таблицы для прочитанного слова                     */
     /* Линейный ***поиск*** с барьером слова sl в таблице t            */
    strcpy (t[dt], sl);         /* Создание барьера == sl               */
    for (i=0; strcmp(sl,t[i]); i++);
    if (i < dt)                 /* Нашли t[i] == sl                     */
        kol[i]++;               /* ***Изменение*** количества повторений sl */
    else                        /* Не нашли                             */
        /* ***Включение*** слова sl в таблицу                           */
        if (dt < DTMAX-1)       /* Есть место                           */
            kol[dt++] = 1;
        else                    /* Таблица переполнена                  */
        {   fprintf (stderr,
                    "\nОшибка: разных слов больше %d"
                    "\nслово '%s' не учитывается\n", DTMAX-1, sl);
        }
```

Ускорить поиск, избежав проверок в цикле на конец таблицы, можно с помощью "*барьера*" - элемента в конце таблицы, содержащего искомый ключ. Плата за ускорение - память для барьера.

```
int L;                          /* Длина просмотра сортировки           */
    /* 3. ***Сортировка*** таблицы по алфавиту методом обмена           */
    for (L=dt-1; L>0; L--)
        /* Просмотр элементов t[0]...t[l]                               */
        for (i=0; i<L; i++)
            /* Сортировка пары t[i] и t[i+1]                            */
            if (strcmp(t[i],t[i+1]) > 0)
            {   /*     Обмен t[i] <--> t[i+1]                           */
                strcpy (sl, t[i]);   strcpy (t[i], t[i+1]);   strcpy (t[i+1], sl);
                /*    Обмен kol[i] <--> kol[i+1]                        */
                j = kol[i];    kol[i] = kol[i+1];    kol[i+1] = j;
             }
```

```
    /* 4. ***Вывод*** (сортированной) таблицы                           */
    printf ("\n%*s%s", DSLMAX, "Слово ", "Кол-во\n");
    for (i=0; i<dt; i++)
        printf ("%-*s %5d\n", DSLMAX, t[i], kol[i]);
```

Таблицы в виде списка

Пример 5.2. ***Упорядоченная линейная таблица в виде списка*** (другой метод решения задачи из примера 5.1). Подсчитаем количество повторений

слов текста с помощью упорядоченной линейной таблицы в виде списка с элементами переменной длины, организованного с помощью указателей.

При нисходящей разработке программы на псевдокоде приведем только фрагменты, отличающиеся от примера 5.1. Упорядоченность таблицы устраняет необходимость ее сортировки (алгоритм зависит от структуры данных!) – алг. 5.2:

Алгоритм 5.2. Подсчет количества повторений слов текста с применением упорядоченной линейной таблицы в виде списка.

```
#include <stdio.h>
 #include <stdlib.h>
void main ()
{
    1. Инициализация таблицы;
    2. Чтение текста и заполнение таблицы слов;
    3. Вывод таблицы;
}
```

Чтение слов входного текста выполняется как в примере 5.1. Отличаются лишь операции с таблицей. На рис. 5.2 показана структура упорядоченной таблицы в виде списка.

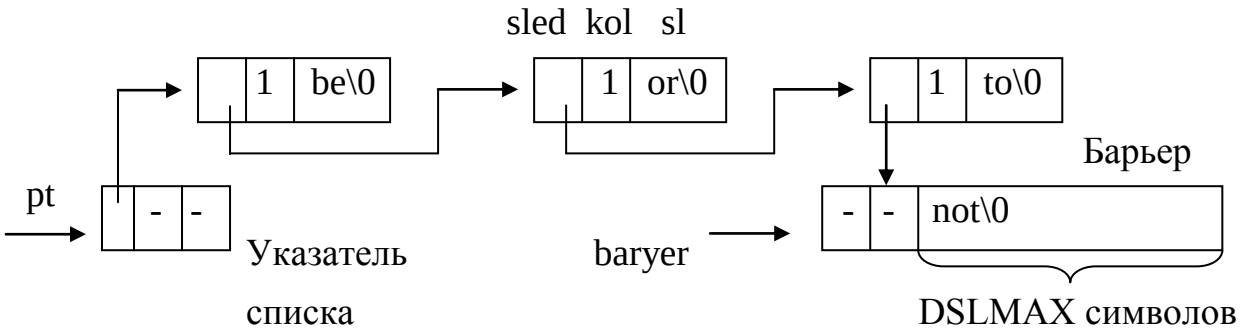


Рис. 5.2. Упорядоченная списковая таблица
с барьером и элементами переменной длины
(при поиске слова not входного текста: to be or not to be)

Первый фиктивный элемент списка содержит указатель фактического начала списка. Хранение указателя в фиктивном элементе списка позволяет включать элемент в начало списка так же, как в его середину - упрощает операцию включения (за счет памяти).

Последний фиктивный элемент списка используется как *барьер* для устранения проверок на конец списка в цикле поиска, подобно примеру 5.1. Это упрощает и ускоряет поиск (тоже за счет памяти).

Здесь у барьера адрес постоянный и, чтобы не копировать туда каждое слово, текущее слово расположено в барьере! Изменяется описание переменой sl, но не программа чтения слова!

Поскольку ключи упорядочены, и при встрече ключа, превышающего искомый, поиск прекращается, в качестве барьера можно было использовать не

только искомый ключ, но и код, состоящий из единиц и заведомо превышающий любой ключ.

Чтобы обеспечить постоянство смещений адресов полей внутри элемента списка, в начале элемента размещены поля фиксированной длины (sled и kol), а затем - поле переменной длины sl (для языка высокого уровня это может делать и сам компилятор).

Определение данных для таблицы:

```
struct el_tab                 /* Элемент таблицы (списка)                    */
{   struct el_tab *sled;      /* Указатель следующего элемента               */
    int kol;                  /* Количество повторений                       */
    char sl[];                /* Слово                                       */
};
struct el_tab *pt;            /* Указатель фиктивного начала таблицы         */
struct el_tab *baryer;        /* Указатель барьера                           */
char *sl;                     /* Адрес текущего входного слова (в барьере)   */
```

Пустая таблица состоит из фиктивного элемента и барьера.

```
/*          1. Инициализация таблицы                                          */
baryer = malloc (sizeof(struct el_tab)
                          + DSLMAX);      /* Место для слова и '\0'          */
sl = baryer->sl;
pt = malloc (sizeof(struct el_tab));
pt->sled = baryer;            /* Пустая таблица                              */
```

```
struct el_tab *i,             /* Указатели текущего и                        */
              ipr;            /*  предыдущего элементов таблицы              */
 /* Корректировка таблицы для прочитанного слова                             */
 /* Линейный поиск с барьером слова sl в таблице                             */
 i = pt->sled; ipr = pt;
 while (strcmp(sl,i->sl) > 0)
 {   ipr=i; i = i->sled;      /* К следующему элементу списка                */
 }
 if (i!=baryer && strcmp(sl,i->sl)==0)   /*Нашли i->sl==sl                   */
       i->kol ++;                  /* Изменение количества повторений sl     */
 else                              /* Не нашли                               */
 {   /* Включение слова sl в таблицу                                         */
     i = malloc (sizeof(struct el_tab)
                      + strlen(sl)+1);    /* Место для sl и '\0'             */
     if (i != NULL)                       /* Есть место                      */
     {   /* Создание нового элемента                                         */
         strcpy (i->sl, sl);              /* Слово sl - в новый элемент      */
         i->kol = 1;
```

```
            /* Включение элемента в список после *ipr                    */
            i->sled = ipr->sled; ipr->sled = i;
        }
        else                                  /* Таблица переполнена     */
        {  fprintf (stderr,
           "\nОшибка: слишком много разных слов, "
           "слово '%s' не учитывается\n", sl);
        }
    }
```

```
/* 3. Вывод таблицы */
printf ("\n%*s%s", DSLMAX, "Слово ", "Кол-во\n");
i = pt->sled;              /* Указатель 1-го не фиктивного элемента списка  */
while (i != baryer)        /* Не дошли до барьера                           */
{   printf ("%-*s %5d\n", DSLMAX, i->sl, i->kol);
    i = i->sled;           /* К следующему элементу списка                  */
}
```

5.3. Длина поиска

Основная характеристика способа организации таблицы - средняя длина поиска элемента, пропорциональная среднему времени поиска.

Длина поиска D - количество просматриваемых при поиске элементов таблицы. D - случайная величина с возможными значениями $D_1=1$, $D_2=2$, ... , $D_m=m$, где m - количество элементов таблицы.

Из теории вероятностей известно, что среднее арифметическое значение (математическое ожидание) случайной величины X с возможными значениями X_1, ..., X_m равно

$$X_{cp} = \sum_{i=1}^{m} X_i * P_i \qquad (5.1)$$

где P_i - вероятность того, что $X = X_i$;

причем $$\sum_{i=1}^{m} P_i = P_1 + ... + P_m = 1 \qquad (5.2)$$

По формуле (5.1) *средняя длина поиска* равна

$$D_{cp} = \sum_{i=1}^{m} D_i * P_i = \sum_{i=1}^{m} i * P_i = P_1 + 2*P_2 + 3*P_3 + ... + m*P_m \qquad (5.3)$$

где P_i - вероятность того, что длина поиска D=i, т.е. искомый элемент имеет в таблице порядковый номер i (i=1..m-1); P_m - сумма вероятностей того, что

искомый элемент имеет номер m, и того, что он отсутствует в таблице (безуспешный поиск требует m шагов).

В частном случае, когда элементы таблицы отыскиваются одинаково часто (равновероятно), и такова же вероятность безуспешного поиска, из формулы (5.2) получим

$$P_1 = P_2 = ... = P_{m-1} = P_m/2 = 1/(m+1)$$

Тогда из (5.3) следует, что

$$D_{ср}^{лин} = (1/(m+1))*(1+2+...+m) = (1/(m+1))*m*(m+1)/2 = m/2$$

т.е. средняя длина линейного поиска - половина длины таблицы:

$$D_{ср}^{лин} = m/2 \qquad (5.4)$$

Из формулы (5.3) видно, что длина поиска уменьшится, если элементы таблицы упорядочить по убыванию частоты обращения к ним (ближе размещать то, что чаще приходится искать), чтобы соблюдались неравенства

$$P_1 \geq P_2 \geq ... \geq P_m.$$

5.4. Двоичный поиск (делением пополам)

Двоичный поиск (*дихотомия*, деление пополам) используется в векторе, упорядоченном по возрастанию или убыванию ключей. Каждое сравнение позволяет определить, в какой части таблицы находится искомый ключ при неравенстве ключей: до или после сравниваемого с ним ключа. Если на каждом шаге делить область поиска пополам, максимальная длина поиска из m элементов равна

$$D_{max}^{дих} = \log_2 m + 1 \qquad (5.5)$$

Алгоритм 5.3. Дихотомический поиск ключа kl в упорядоченном по возрастанию векторе t (t[i-1] ≤ t[i] для i=1, ..., m-1)

```
L = 0;  R = m;              /* Индексы левой и правой границы поиска      */
while (L < R)
{   /* (t[k] < kl  для k=0, ..., L) && (t[k] >= kl  для k = R ,..., m-1)  */
      i = (L+R) / 2;   if (t[i] < kl)  L = i+1;   else  R = i;
}
if (R < m && t[R] == kl) ...          /* Нашли                            */
```

Доказательство правильности алгоритма 5.3:

а) *Инвариант цикла*:

(t[k]<kl для k=0, ..., L) && (t[k]>=kl для k=R, ..., m-1)

б) *Конечность цикла* следует из того, что R - L убывает при каждом повторении цикла и обязательно станет нулем, т. к.: перед телом цикла L<R; средний индекс L ≤ i < R; на каждом шаге либо L увеличивается до i+1, либо R уменьшается до i. При L = R цикл заканчивается.

в) При R=m - *не нашли* (т. к. t[m] вне вектора!), иначе надо проверить t[R], поскольку он не участвует в сравнениях.

Алгоритм 5.3а. Дихотомический поиск ключа kl в упорядоченном по возрастанию векторе t (t[i-1] ≤ t[i] для i=1, ..., m-1) с использованием указателей L, R, j (быстрее, чем 5.3)

```
L = &t[0];  R = &t[m];            /* Адреса левой и правой границы поиска */
while (L < R)
{    j = L + (R - L) / 2;         /* Нельзя складывать ссылки: j=(L+R)/2    */
     if (*j < kl)   L = j + 1;   else  R = j;
}
if (R < &t[m] && *R == kl) ...          /* нашли                          */
```

Примечание. Разность адресов равна числу ячеек между ними, а сумма адресов бессмысленна. Поэтому в языке С запрещено сложение указателей, и среднее приходится вычислять через разность.

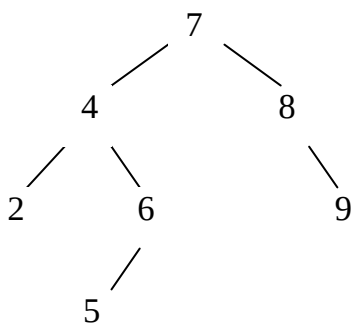
Двоичный поиск намного быстрее линейного, но требует, чтобы ключи образовали упорядоченный вектор. Обычно он применяется для постоянных таблиц или в случаях, когда сначала таблица заполняется, затем упорядочивается и потом уже используется только для поиска.

5.5. Древовидные таблицы

Древовидная таблица (*дерево поиска*) – это бинарное дерево, в котором вершины соответствуют элементам таблицы; причем ключ каждой вершины больше ключей ее левого поддерева и меньше ключей правого поддерева (рис. 5.3). Ключи могут быть любого типа (сравниваются их числовые коды).

а) Поступающие ключи: 7, 8, 4, 9, 2, 6, 5

б) Дерево поиска



в) Представление дерева параллельными массивами

Индекс	Ключ kl	Ссылки men	bol
	. . .		
13	7	15	14
14	8	-1	16
15	4	17	18
16	9	-1	-1
17	2	-1	-1
18	6	19	-1
19	5	-1	-1
	. . .		

Рис. 5.3. Древовидная таблица

Первый поступающий элемент образует корень дерева. Очередной ключ ищется в таблице, начиная с корня. После сравнения с ключом текущей вершины поиск продолжается в ее левом или правом поддереве: область поиска делится на две части, подобно дихотомии. Если нужное поддерево отсутствует (соответствующая ссылка пустая), искомого ключа нет в таблице. При включении новый элемент размещается в свободном месте памяти и подвешивается к найденной пустой ссылке, роль которой играет -1 (рис. 5.3).

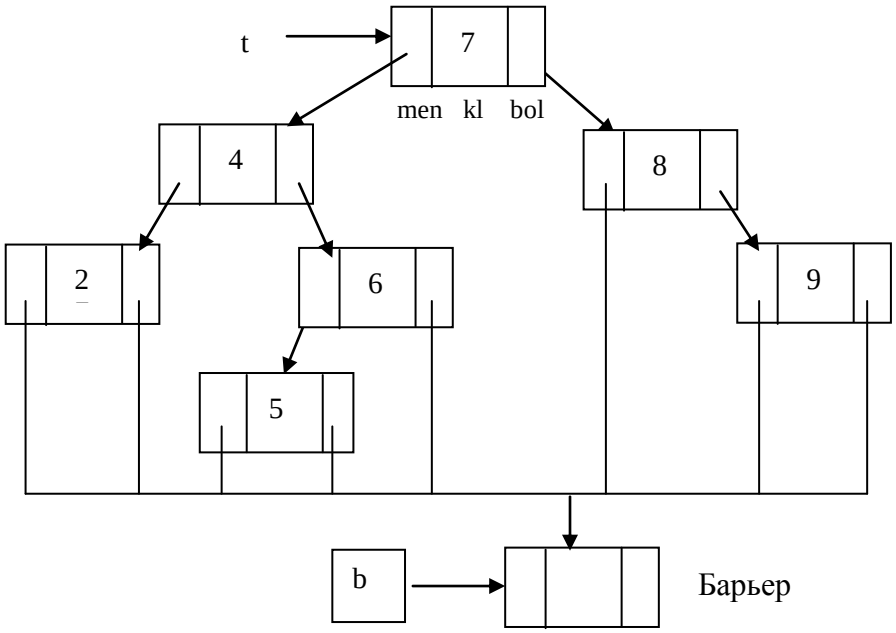


Рис. 5.4. Древовидная таблица с барьером

Для ускорения поиска как и в последовательной таблице удобно использовать *барьер* с искомым ключом. Барьером заканчиваются все пути дерева: все пустые ссылки заменяются ссылками на барьер (рис. 5.4). Барьер избавляет от проверки пустоты ссылок в цикле поиска (алг. 5.4, 5.5).

Алгоритм 5.4. Создание древовидной таблицы t с барьером b

```
/*                    Описание таблицы                              */
struct el_tab                       /* Элемент таблицы             */
{       Тип_ключа kl;               /* Ключ                        */
        Тип_инф inf;                /* Тело элемента               */
        struct el_tab *men, *bol;  /      * Ссылки влево и вправо   */
};
struct el_tab *t;                   /* Указатель корня дерева      */
struct el_tab *b;                   /* Указатель барьера           */
        . . .
/*                  Инициализация пустой таблицы                    */
/* Создание элемента для барьера                                    */
b = malloc (sizeof(struct el_tab)); t = b;  /* "Пустая" ссылка на корень дерева */
```

Алгоритм 5.5. Нерекурсивный поиск и включение элемента {kl, inf} в древовидной таблице t с барьером b

```
Тип_ключа kl;          /* Ключ нового элемента                          */
Тип_инф   inf;         /* Тело нового элемента                          */
struct el_tab  *i, *ip;  /* Указ-ли текущего и предыд-го эл-в           */
                          . . .
/*              Поиск с барьером ключа kl в таблице t                   */
b->kl = kl;                          /* Создание барьера = kl           */
for (i = t; kl != i->kl;)
{  ip = i;
   if (kl < i -> kl)   i = i-> men;
   else                i = i-> bol;
}
if (i == b)                          /* Не нашли (наткнулись на барьер) */
{ /*          Включение элемента с ключом kl в таблицу t                */
   i = malloc (sizeof (struct el_tab));
   if (i == NULL) ...                /* Переполнение таблицы            */
   else                              /* Есть место для нового элемента  */
   {    i->kl=kl; i->inf=inf;        /* Создание нового элемента        */
        i->men = i->bol = b;         /* "Пустые" ссылки на барьер       */
        /* Прицепление нового элемента к дереву                         */
        if (t == b)                  /* Дерево пусто */
            t = i;                   /* Новый  элемент - корень дерева  */
        else                         /* Прицепление  нового  элемента к *ip */
            if (kl < ip->kl)  ip -> men = i;  else  ip->bol = i;
   }
}
else          . . .                  /* Нашли     i->kl == kl           */
```

Алгоритм 5.6. ***Вывод*** по возрастанию ключей древовидной таблицы t с барьером b (рекурсивный обход бинарного дерева слева направо)

```
 void pech_tab (struct el_tab *t)
{    if (t != b)                     /* b - "пустая" ссылка на барьер   */
     {  pech_tab (t -> men);
        Вывод t->kl, t -> inf;
        pech_tab (t -> bol);
     }
}
```

Примечания. 1. Если нет барьера, вместо b - пустая ссылка NULL.

2. Можно использовать фиктивный корень с ключом равным 0.

Исключение элемента из древовидной таблицы показано на рис. 5.5. При исключении элемента, имеющего менее двух потомков, например, элемента с ключом 6, он уничтожается. Ссылка на него заменяется ссылкой на его потомка (она может быть и пустой).

Некоторые трудности создает исключение элемента с двумя потомками, например, элемента с ключом 7. В этом случае, чтобы не нарушить структуру дерева поиска, в исключаемый элемент пересылаются данные из элемента с ближайшим меньшим (или большим) ключом, который затем уничтожается. В алг. 5.7 это - самый правый элемент (с ключом 6) левого поддерева исключаемого элемента.

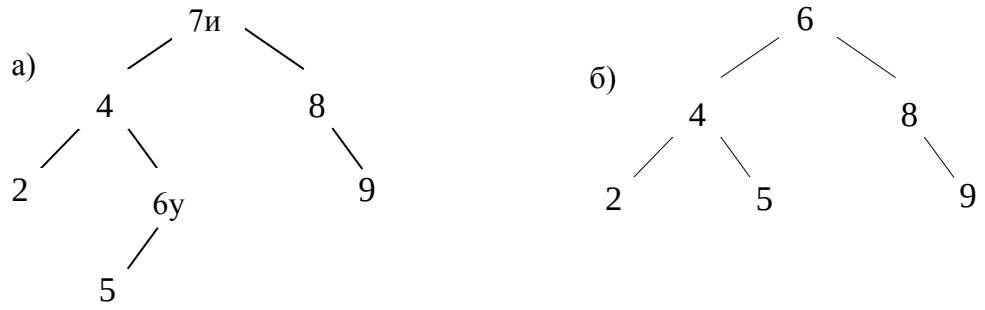


Рис. 5.5. Исключение элемента с ключом 7 из древовидной таблицы
а) до исключения; б) после исключения; "и", "у" - исключаемый и уничтожаемый элементы

Ссылка на уничтожаемый элемент заменяется ссылкой на его левое поддерево (правого поддерева у него нет).

Алгоритм 5.7. ***Исключение*** элемента из древовидной таблицы с барьером b

```
struct el_tab      **i,        /* Указатель ссылки на исключаемый эл-т      */
                   *u,         /* Указатель уничтожаемого элемента          */
                   **r;        /* Указатель ссылки на уничтож-мый эл-т      */
                    . . .
u = *i;                        /* Указатель исключаемого элемента           */
if (u->men == b)    *i = u->bol;      /* Нет левого потомка                   */
else  if (u->bol == b)    *i = u->men;   /* Нет правого потомка              */
else   /* Исключаемый элемент имеет двух потомков */
{      /* Поиск правого элемента левого поддерева исключаемого элемента    */
       r=&(u->men);  u=u->men;
       while (u->bol != b)        { r=&(u->bol); u=u->bol; }
       /* Пересылка уничтожаемого элемента в исключаемый элемент           */
       (*i)->kl=u->kl; (*i)->inf=u->inf;
       *r=u->men;                 /* Ссылка в обход уничтожаемого элемента  */
}
free (u);                      /* Уничтожение элемента *u                   */
```

Длина поиска в древовидной таблице зависит от конфигурации дерева, которая определяется порядком поступления ключей в таблицу. В худшем случае, когда ключи поступают по возрастанию или убыванию, дерево вырождается в одну ветку, длина поиска как в последовательной таблице:

$$D_{ср}^{посл} = m / 2, \qquad (m - количество элементов таблицы).$$

В среднем по всем конфигурациям деревьев (считая их равновероятными) средняя длина поиска равна:

$$D_{ср}^{древ} = 1.39 \log_2 m \qquad (5.6)$$

Для ускорения поиска после каждого включения и исключения элемента дерево балансируют (перестраивают). Рассмотрим два вида сбалансированных деревьев: выровненное и подравненное (рис. 5.6).

Выровненным (идеально сбалансированным) называют дерево, в котором незаполненным может быть только последний уровень (рис. 5.5а). Максимальная длина поиска в таком дереве, как при дихотомии:

$$D_{max}^{вып} = \log_2 m + 1 \qquad (5.7)$$

Российские математики Г.М. Адельсон-Вельский и Е.М. Ландис предложили более удобный для практики критерий сбалансированности: *подравненное дерево* (*АВЛ-дерево*) - такое дерево, в котором высоты двух поддеревьев каждой вершины отличаются не более чем на 1 (рис. 5.5 б). Выровненное дерево является частным случаем АВЛ-дерева. Алгоритм балансировки АВЛ-дерева значительно проще, а максимальная длина поиска всего на 44% больше чем у выровненного дерева и приблизительно равна средней длине поиска в древовидной таблице:

$$D_{max}^{АВЛ} = 1.44 \log_2 m \qquad (5.8)$$

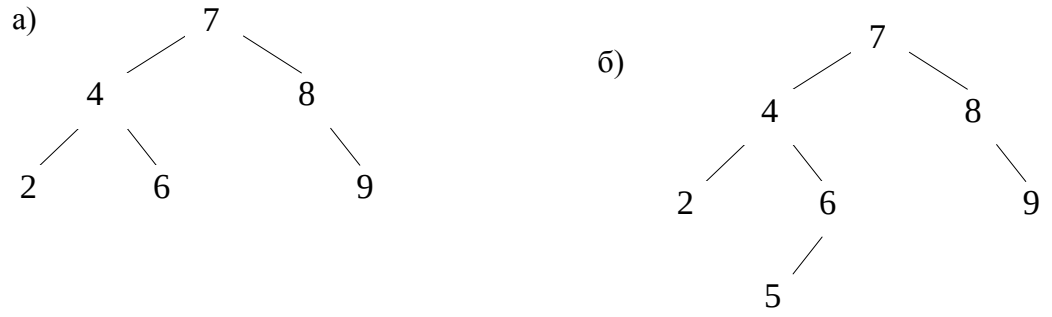


Рис. 5.6. Сбалансированные деревья
а) Выровненное дерево б) Подравненное дерево (АВЛ-дерево)

5.6. Таблицы с вычисляемым адресом

Таблица с вычисляемым адресом (*преобразованием ключа* в адрес) представляет собой вектор из N ключей с индексами от 0 до N-1. Поиск элемента начинается с индекса i, вычисляемого из его ключа kl по формуле *хеширования* (hash - перемешать, мешанина):

$$i = H(kl) \qquad (5.9)$$

где H - *функция расстановки* (*хеш-функция*) с целочисленными значениями, удовлетворяющими условию

$$0 \le H(kl) \le N-1 \qquad (5.10)$$

В частном случае, когда разным ключам всегда соответствуют разные значения функции расстановки:

$$kl1 \neq kl2 \implies H(kl1) \neq H(kl2) \qquad (5.11)$$

получается таблица *с прямым доступом*. В *перемешанных таблицах* может возникать *конфликт*, когда разные ключи претендуют на одно и то же место:

$$kl1 \neq kl2 \;\&\&\; H(kl1) = H(kl2) \qquad (5.12)$$

5.6.1. Таблицы с прямым доступом

В таблице с прямым доступом индекс элемента определяется по формуле (5.9) только его ключом (ключи поэтому можно не хранить в таблице), длина поиска всегда 1:

$$D^{прям} = 1 \qquad (5.13)$$

Таблицы с прямым доступом дают максимально быстрый поиск, но трудно подобрать бесконфликтную функцию расстановки, удовлетворяющую условию (5.11). Это удается лишь в редких случаях, например, для некоторых постоянных таблиц и табуляции функций числового аргумента. В большинстве практических случаев конфликты неизбежны, т. к. мощность множества ключей превышает размер таблицы.

Пример 5.3. Таблица перекодировки символов

Ключ x - код символа (x = от 0 до 255), тело t[i] - новый код символа, количество элементов таблицы 256, индекс элемента – от 0 до 255.

Отсюда: функция расстановки: i = H(x) = x; новый код символа

x = t[i] = t[H(x)] = t[x]

```
/* Новые коды символов:    0,   1, ... , 255          */
char   x,   t[256]              = { 45, 19, ... , 13 };
. . .
x  =  t[x];                     /* Перекодировка символа x   */
```

Пример 5.4. Таблица функции f(x) (x = 2, 2.01, ..., 2.99)

Ключ x - аргумент, тело элемента t[i] = f(x), количество элементов таблицы 100, индекс элемента 0..99.

Отсюда: функция расстановки: i = H(x) = 100*(x-2);

f(x) = t[i] = t[H(x)] = t[100*(x-2)].

```
/* Таблица функции  f(x)        (x = 2, 2.01, ... , 2.99):     */
/*                        f(2), f(2.01), ..., f(2.99)                */
float   t[100]  =  { 3.51, 3.47,      ..., -0.74 };
```

5.6.2. Перемешанные таблицы

Другие названия: ***рандомизированные таблицы*** (таблицы со случайным перемешиванием), *рассеянные таблицы*, *хеш-таблицы*.

Пример 5.5. Таблица с открытым перемешиванием (рис. 5.7)

Ключи - названия вузов; на рис. 5.7 а - значения функции расстановки; на рис. 5.7 б - поступающие ключи. Новый ключ помещается в позицию с индексом i = H(kl), а если она занята другим ключом (конфликт), то таблица просматривается циклически (i увеличивается на 1 по модулю N) до ключа, равного искомому, или до ближайшей свободной позиции (рис. 5.7 в). Обнаружение свободной позиции означает отсутствие в таблице поступившего ключа, и этот ключ можно поместить в найденную позицию.

а) Функция расстановки

kl	H(kl)
ВГУ	2
КАИ	6
КГУ	7
ЛГУ	4
МГУ	0
НГУ	3
ТГУ	6
ТПИ	3

. . .

б) Поступающие ключи:
КАИ, МГУ, ТПИ, КГУ, ТГУ, НГУ

в) Расстановочное поле для ключей

0	МГУ
1	ТГУ
2	
3	ТПИ
4	НГУ
5	
6	КАИ
7	КГУ

Рис. 5.7. Таблица с открытым перемешиванием

Свободные позиции таблицы содержат особое значение, отличающее их от ключей (алг. 5.8, 5.9).

Алгоритм 5.8. Создание перемешанной таблицы t

```
/*          Описание данных для перемешанной таблицы          */
#define     N ...       /* Длина таблицы                      */
#define     SVOB ...    /* Свободный элемент                  */
Тип_ключа t[N];         /* Расстановочное поле                */
int     ksv;            /* Кол-во позиций для включения элементов */
        . . .
/*      Инициализация пустой перемешанной таблицы             */
int     i;
        . . .
ksv  =  N-1;                    /* Одна свободная позиция для поиска */
for (i=0; i<N; i++) t[i] = SVOB;
```

Алгоритм 5.9. Поиск с включением ключа kl в перемешанной таблице t (линейные пробы с шагом 1)

```
Тип_ключа kl;                       /* Ключ нового элемента      */
int i;                              /* Текущий индекс в таблице  */
/*      Поиск ключа kl в таблице t                                */
i = H(kl);                          /* H - функция расстановки   */
while (t[i]!=kl && t[i]!=SVOB)
   if (i < N-1)  i++;  else  i=0;   /* i = (i+1) % N;            */
if (t[i] != kl)                     /* Не нашли                  */
{   /* Включение элемента с ключом kl в таблицу t                */
    if (ksv==0)  . . .              /* Переполнение таблицы      */
    else                            /* Есть место для нового элемента */
    {   ksv--;                      /* Занимаем SVOB             */
        t[i]  =  kl;                /* Создание нового элемента  */
    }
} else . . .                        /* Нашли t[i]  ==  kl        */
```

Слово "открытый" означает, что каждая позиция таблицы открыта для любого ключа, независимо от его функции расстановки. Чтобы не считать при поиске количество просмотренных позиций, хотя бы одну позицию оставляют свободной.

Нельзя просто заменять значением "свободный" удаляемые из таблицы ключи. Например, если “удалить” таким образом ключ КГУ из таблицы на рис. 5.7 в, то при последующем поиске будет сделан неверный вывод, что ключа ТГУ нет в таблице. Чтобы не прерывать пути поиска, на место удаляемого элемента перемещается элемент, путь поиска которого проходит через позицию удаляемого элемента. Затем таким же образом освобождается старая позиция

перемещенного элемента и т. д. (алг. 5.10). Это удается сделать только при включении элементов по алг. 5.9 для линейного рехеширования.

Алгоритм 5.10. Исключение элемента t[i] из перемешанной таблицы t, получаемой по алгоритму 5.9

```
int     j;                          /* Индекс удаляемого эл-та        */
int     i;                          /* Текущий индекс в таблице       */
int     r;                          /* H(t[i])                        */
do      /*      Исключение элемента t[i]                              */
{   j=i;   t[j]=SVOB;
    /* Поиск элемента, перемещаемого в освобождаемую позицию          */
    do
    {    if (i<N-1) i++;  else  i=0;
         if (t[i]!=SVOB)  r=H(t[i]);
    } while (t[i] != SVOB &&        /* r циклически между j и i :     */
          (j<r && r<=i || i<j && (r<=i || j<r)));
    if (t[i] != SVOB)  t[j] = t[i]; /* Перемещение элемента           */
} while (t[i] != SVOB);
ksv++;                              /* Кол-во свободных позиций       */
```

Удаляемые ключи можно заменять другим особым значением "удаленный", которое при поиске трактуется как занятая позиция, а при включении - как свободная. Эта идея оправдана только при редких удалениях, т. к. занятая позиция уже не может стать свободной. После ряда включений и удалений останется одна свободная позиция и безуспешный поиск будет долгим: N проб.

Теоретические и экспериментальные исследования выявили удивительный факт: длина поиска в перемешанной таблице очень мала, и в отличие от всех других методов зависит не от размера таблицы, а только от степени ее заполнения!

Для таблицы с открытым перемешиванием и линейными пробами (как в алг. 5.9) при равномерном распределении значений функции расстановки от 0 до N-1 средняя длина поиска приблизительно равна

$$D_{ср}^{перем} = (2 - s) / (2 - 2*s) \qquad (для\ s \le 0.85) \qquad (5.14)$$

где s = m / N - коэффициент заполнения таблицы,
N - длина расстановочного поля,
m - количество элементов таблицы (занятых позиций).

Например, при s=0.8 Dcp =3, т. е. в заполненной на 80% перемешанной таблице, независимо от того, сколько в ней элементов: 100 или 100000, поиск в среднем требует всего трех шагов! Это намного быстрее других методов.

При неравномерном заполнении таблицы длина поиска увеличивается на практике в 2-4 и более раз (все равно это быстро).

Две основные проблемы перемешанных таблиц - выбор функции расстановки и метода преодоления конфликтов.

Для функции расстановки обязательно условие (5.10) и желательно достаточно быстрое вычисление функции и, по возможности, равномерное рассеивание ключей по таблице для уменьшения числа конфликтов. Часто применяют функцию:

H(kl) = (Числовой код kl) % N

При N равном степени двух функцию H(kl) легко вычислять, но она плохо рассеивает символьные ключи, отличающиеся одним-двумя символами, типа: XA, XB, XC. В этом случае в качестве N рекомендуется простое число (делящееся только на себя и на 1). Деление часто заменяют умножением на обратную величину (это быстрее). Если ключ занимает несколько машинных слов, их предварительно можно сложить в одно слово арифметически или поразрядно по модулю 2 (исключающее или, в Си обозначается как ^).

Конфликт возникает, когда позиция i=H(kl) содержит ключ, не равный kl. Для разрешения конфликтов можно связать ссылками в список все конфликтующие ключи с одинаковыми значениями функции расстановки. Элементы списков размещаются в расстановочном поле или в отдельной области переполнения (тогда расстановочное поле может содержать только указатели списков).

Средняя длина поиска последовательным перебором в списке меньше, чем по формуле (5.13), т. к. списки короткие ввиду малого числа конфликтов. При наличии области переполнения размер расстановочного поля уже не ограничивает число элементов таблицы, а влияет лишь на количество конфликтов и скорость поиска. Недостатки метода - дополнительная память для ссылок и необходимость системы управления памятью. Другой метод разрешения конфликтов - открытая адресация (повторное перемешивание, рехеширование). В этом случае для поиска элемента пробуются позиции расстановочного поля i_0, i_1, i_2, ... , где

i_0 = H(kl),
i_k = (i_0+G(k)) % N при k>0, (5.15)
H(kl) - функция первичной расстановки,
G(k) - функция вторичной расстановки.

В простейшем случае G(k) = d*k, G линейно зависит от k. Это называется *линейным рехешированием* (опробованием) с шагом d (в алг. 5.9 шаг d=1). Его недостаток - скопления ключей в местах конфликтов.

Меньшие скопления возникают при *квадратичном рехешировании*, например, G(k) = k^2. При этом можно избежать умножения благодаря рекуррентным соотношениям

$$G(k+1) = (k+1)^2 = k^2+2*k+1 = G(k)+d(k),$$

где d(k) = 2*k+1;

Отсюда: d(k+1) = 2*(k+1)+1 = d(k)+2.

Они реализуются в цикле поиска (как в алг. 5.9) операторами

```
i = i + d;    if(i >= N) i = i - N;    d = d + 2;
```

с начальными значениями i = H(kl); d=1;

Небольшой недостаток - при квадратичном рехешировании пробуются не все, а лишь несколько более половины позиций таблицы (если N - простое число), и при наличии свободных мест для включения они могут не найтись, но обычно это бывает очень редко, когда таблица почти заполнена.

Для более равномерного вторичного рассеивания ключей функцию G подобно H делают зависимой и от ключа kl или используют псевдослучайные числа, но это усложняет алгоритм.

5.7. Сравнение методов организации таблиц

В малых таблицах средняя длина поиска приблизительно одинакова для всех методов (табл. 5.1, столбец m=8).

Поэтому для малого числа элементов предпочтительнее линейные таблицы (алгоритм проще, быстрее по времени). Для удаления элементов удобнее списки.

В больших таблицах самый быстрый поиск дают методы хеширования. Если они неудобны по каким-то причинам (требуется упорядоченная распечатка, частые удаления, трудно найти подходящую функцию расстановки или предугадать размер таблицы), используют древовидные таблицы, возможно с балансировкой.

Таблица 5.1

Сравнение таблиц по длине поиска

Вид таблицы	Средняя длина поиска (m – количество элементов)			
	Формула для D_{cp}	m = 8	m = 64	m= 1024
Линейная	m / 2	4	32	512
Древовидная	$1.39 \log_2 m$	4.2	8.4	13.9
Перемешанная при N=1.25 m, s=0.8	(2-s) / (2-2*s) $D_{cp} = 3$	3	3	3

В хеш-таблицах мала средняя длина поиска, но максимальная длина поиска в наихудшем случае велика - равна m. Поэтому они могут оказаться неприемлемыми и в особо ответственных системах, например, для управления транспортом. Сбалансированные деревья дают гарантии достаточно быстрого поиска и в наихудшем случае.

Граница между малыми и большими таблицами зависит от ЭВМ, длины ключа и других факторов и соответствует обычно значению m от 30 до 80.

5.8. Задачи

5.1. Составить полные программы по алгоритмам, приведенным в примерах 5.1, 5.2.

5.2. Составить фрагменты программ поиска без использования барьера для линейных таблиц в виде вектора и списка и древовидных таблиц.

5.3. Оформить приведенные в разделе 5 операции над таблицами в виде подпрограмм.

5.4. Написать программу подсчета количества повторений в данном тексте каждой цифры: 0, 1, ... , 9.

5.5. В пустую таблицу поступают ключи в следующем порядке: МГУ, ЛГУ, КАИ, ТГУ, КГПУ, ТПИ, НГТУ. Изобразить результирующее дерево поиска и представление в памяти этого дерева с помощью параллельных массивов. Является ли полученное дерево выровненным, АВЛ-деревом? Указать последовательность ключей при обходе этого дерева сверху, слева направо и снизу (см. раздел 4). Как изменится таблица после удаления ключа ЛГУ?

5.6. В пустую таблицу поступают ключи в следующем порядке: МГУ, ЛГУ, КАИ, ТГУ, КГПУ, ТПИ, НГТУ. Для хеш-таблицы с открытым перемешиванием показать содержимое отображающего вектора (расстановочного поля), если его длина равна 9, шаг перемешивания равен 2. Хеш-функция определена таблично:

Ключ	КАИ	КГПУ	ЛГУ	МГУ	НГТУ	ТГУ	ТПИ
Значение хеш-функции	6	0	7	5	3	5	6

Как изменится таблица после удаления ключа ЛГУ?

5.7. Древовидная таблица содержит ключи (в порядке поступления): МГУ, ЛГУ, КАИ, ТГУ, КГПУ, ТПИ, НГТУ. Найти среднюю длину поиска для дерева полученной конфигурации, считая, что вероятности поиска всех ключей одинаковы и равны 0.1, а вероятность безуспешного поиска (отсутствующего ключа) равна 0.3. Считать, что безуспешный поиск с равной вероятностью заканчивается в любой из пустых ссылок.

Для сравнения оценить длину поиска в таблице в среднем по всем конфигурациям дерева, считая их равновероятными, по общей формуле:

$$D_{ср}^{древ} = 1.39 \log_2 m,$$

где m - количество элементов таблицы.

5.8. Таблица содержит ключи (в порядке поступления): МГУ, ЛГУ, КАИ, ТГУ, КГПУ, ТПИ, НГТУ.

Даны вероятности:

- безуспешного поиска - 0.1;
- поиска ключей: КАИ - 0.3, КГПУ - 0.1, ЛГУ - 0.1, МГУ - 0.15, НГТУ - 0.1, ТГУ - 0.1, ТПИ - 0.05 .

Найти среднюю длину поиска для разных методов организации таблиц:

1) линейная таблица с расположением ключей в порядке их поступления;
2) линейная таблица с расположением ключей в лексикографическом порядке (как в словарях);
3) линейная таблица с расположением ключей в порядке убывания вероятностей их поиска;
4) древовидная таблица с расположением ключей в порядке их поступления;
5) перемешанная таблица из задачи 4.3 (рис. 4.2а);
6) перемешанная таблица из 7 элементов с расстановочным полем длиной 9 и равномерно распределенной функцией расстановки (как на рис. 4.2а, только без учета конкретных ключей и их вероятностей).

5.9. Составить фрагменты программы (подпрограммы) включения и исключения элемента в упорядоченной по возрастанию ключей линейной таблице в виде вектора. Ключ элемента - вещественное число.

5.10. Составить фрагменты программы (подпрограммы) инициализации и поиска с включением в древовидной таблице без использования барьера для представления дерева

а) ссылочными данными;

б) параллельными массивами.

5.11. Для перемешанной таблицы составить фрагменты программы (подпрограммы)

а) поиска с включением открытым перемешиванием с квадратичным рехешированием;

б) инициализации, поиска, включения и удаления для разрешения конфликтов с помощью списков конфликтующих ключей.

5.12. Для перемешанной таблицы длиной $n = 2^k$ составить подпрограмму вычисления индекса p повторного перемешивания, используя псевдослучайное число r с начальным значением 1, по следующему методу:

r = младшие к+2 бита 5*r;

p = r, сдвинутое на 2 бита вправо.

5.13. Оценить необходимый объем памяти для организации таблиц в задачах: 5.4 - 5.6, 5.8 - 5.11. Недостающие детали представления уточнить самостоятельно.