Глава 14. Таблицы

Ученье без размышления бесполезно, но и размышление без ученья опасно.

Конфуций, V век до н.э.

Таблица – структура данных, каждый элемент которой характеризуется определенным ключом. Доступ к элементам таблицы, а также включение и исключение элементов осуществляется по ключу.

Таблицы широко используются в задачах обработки данных. Рассмотрим использование этой структуры данных в так называемых задачах информационного поиска.

14.1. Задача информационного поиска

Кто ищет, тот всегда что-нибудь найдёт, но совсем не обязательно то, что искал.

Джон Толкин.

«Хоббит, или Туда и обратно»

Остерегайтесь найти то, что вы ищете.

Р.В. Хэмминг

Значительная часть всех вычислительных ресурсов сейчас тратится на поиск информации в различных информационный системах, базах данных, в сетевых ресурсах Интернета и т.д. Весь этот поиск можно условно разделить на две части, в зависимости от ожидаемых результатов.

Для некоторых поисковых запросов мы ожидаем лишь приблизительное соответствия между выданными данными и нашими интересами (обычно такое соответствие называется релевантностью между запросом и результатом поиска). Например, при запросе в системе Google «Опасность землетрясений», мы получаем ссылки на самые разнообразные сведения, от классификации силы землетрясений, до их предсказаний.

Для других запросов мы ожидаем точного ответа. Ясно, что нас совсем не удовлетворит, если в ответ на запрос о балансе счёта телефона мы получим в ответ «Кажется, у Вас маловато денег». Для таких случаев мы выдаем *формализованные* запросы и получаем точные (формализованные) ответы. Так работающие информационные системы и базы данных устроены по разному, имеют весьма сложную структуру, их изучение является темой отдельных курсов. В то же время, первоосновой таких систем служит относительно простая абстрактная структура данных, под названием таблица [1] (иногда используется немного более понятный термин «словарь»). К изучению этой структуры данных мы сейчас и приступим.

Абстрактная структура данных таблица является (возможно пустым) *неупорядоченным* набором элементов. Каждый элемент есть пара <ключ : значение> (key-value). В каждый момент времени все ключи в таблице уникальны (не повторяются). Ключи можно сравнивать между собой на равно и не равно, понятно, что только так и можно проверить уникальность ключей.

Таблицы являются частным случаем так называемых ассоциативных массивов. Доступ к элементам таких массивов производится не по их индексам, как обычно, а по содержащимся в элементах массива ключам. Это, собственно, и делает абстрактную структуры данных *таблица* частным случаем такого массива (в ассоциативном массиве, однако, не требуется уникальность ключей). Отметим, что так организованная память часто используется и в компьютерах, она так и называется ассоциативная память, в частности это особая очень быстрая кэш-память.

Для таблиц определены три стандартных операции:

1. Поиск в таблице элемента с заданным ключом.

---

[1] Не надо путать эту стуктуру данных с широко распространёнными электронными таблицами, которые по-существу являются простейшими базами данных.

2. Включение (добавление) в таблицу нового элемента. При совпадении ключа добавляемого элемента с одним из ключей в таблице действие операции не определено (зависит от реализации).
3. Исключение (удаление) из таблицы элемента с заданным ключом. При отсутствии такого ключа в таблице действие операции не определено (зависит от реализации).

Отметим, что таблица похожа на множество ключей, где похожие операции и каждый элемент может присутствовать в множестве не больше одного раза. Правда, при повторном включении элемента в множество это просто пустая операция. Кроме того, в отличие от множества, в каждом элементе таблицы, кроме ключа для поиска хранятся и некоторые (полезные) данные. В некоторых языках, например, в Python, эти множества и таблицы (словари) описываются и реализуются похожим способом.

Хорошо всем знакомыми примерами таблиц являются словари (например, англо-русский словарь). Ключами там выступают подлежащие переводу иностранные слова, а значениями – абзацы текста с возможными переводами этого слова на другой язык. Отсюда понятно, почему в некоторых языках программирования (например, в языке Python) таблицы так и называются словарями. Понятно, что все синонимы должны задавать в словаре один вход, а уж в самой статье описываются разные значения данного слова.

Разумеется, скажем, в современных базах данных реализуются более сложные таблицы. Каждый элемент (называемый кортéжем) может иметь не один, а несколько ключей. Например, в базе данных по легковым автомобилям можно искать машину по номеру, по владельцу и т.д. Правда, один из ключей считается основным, а остальные вспомогательными. Кроме того, как правило, это не одна, а несколько *взаимосвязанных* таблиц (обычно называемых отношениями). Но общая идея остаётся.

Познакомимся с простейшими реализациями этой абстрактной структуры данных.

14.2. Неупорядоченные и упорядоченные массивы

Только дурак нуждается в порядке – гений господствует над хаосом.
Альберт Эйнштейн

Простейшую реализацию таблиц на Паскале можно сделать с помощью массивов, где размер массива определяет максимальное количество элементов таблицы. Внешне такие массивы похожи на файлы, они заполняются сверху вниз и имеют указатель свободного места. Каждый элемент массива хранит элемент таблицы и является записью Паскаля <ключ : значение>. Показанный в виде стрелки → указатель установлен на первый свободный элемент массива, как показано справа.

	ключ	значение
	ключ	значение
	ключ	значение
	ключ	значение
	ключ	значение
→		

В неупорядоченном массиве ключи расположены в произвольном порядке. Обозначим буквой N количество элементов в таблице. Для поиска нужного ключа придётся просмотреть (от начала к концу) в среднем половину элементов массива, так что сложность операции поиска будет O(N).

На первый взгляд, операция вставки в таблицу нового элемента совсем простая, надо записать его на свободное место и сдвинуть вниз указатель. Это, однако, не так, сначала надо убедиться, что элемента с таким ключом в таблице уже нет. Это дополнительная операция поиска, так что сложность включения тоже будет O(N).

И, наконец, операция исключения элемента. Предварительно надо этот элемент найти (за O(N) операций), а потом все расположенные ниже элементы сдвинуть на одну позицию вверх, убирая в массиве «свободное место», так что сложность исключения тоже будет O(N).

Такие «нехорошие» оценки сложности операций можно исправить, если наложить на ключи дополнительное условие. Потребуем, чтобы их можно было сравнивать между собой не только на равно и не равно, но и на больше и меньше.

Теперь ключи уже не могут быть, например, комплексными числами или, скажем, матрицами. Без потери общности такие «упорядоченные» ключи можно считать неотрицательными целыми числами. Действительно, Вы уже должны понимать, что, например, для английского алфавита слова-ключи в словаре будут числами в системе счисления с основанием 26 (это число букв в алфавите).

Значения строк символов (например, типа `string`) будут «числами» в системе счисления с основанием 256 и т.д.

Для упорядоченных ключей можно реализовать таблицу в виде массива, в котором ключи идут по возрастанию (ну, или по убыванию). Такие массивы называются упорядоченными или отсортированными. В этих массивах возможен так называемый двоичный (или бинарный) поиск, известный Вам по детской игре «угадай число от 1 до 1000 за 10 вопросов.»

При таком поиске при каждой проверке ключа отбрасывается примерно половина ключей, следовательно, сложность операции поиска равна $O(\log_2(N))$. Алгоритм двоичного поиска либо находит элемент с заданным ключом, либо указывает на тот элемент, на месте которого он должен был бы находится. Это определяет алгоритм включения в таблицу нового элемента. Сначала производится поиск, если он завершается успешно, то такой ключ в таблице уже есть, в этом случае действие зависит от реализации. Когда ключа нет, то все элемента массива, начиная от найденного места хранения нового элемента, сдвигаются вниз на одну позицию, а в освободившееся место заноситься новый элемент. Итого сложность операции включения нового элемента равна $O(\log_2(N))$ (поиск) + $O(N)$ сдвиг, итого $O(N)$.

Аналогично реализуется операция исключения. Сначала находится позиция исключаемого элемента, по сложности это $O(\log_2(N))$. Когда исключаемый элемент отсутствует, то действие зависит от реализации, иначе все элементы массива сдвигаются вверх на одну позицию, итого общая сложность операции исключения элемента равно $O(N)$.

Итак, таблицы, реализованные как упорядоченные массивы, обеспечивают относительно быстрый поиск, но по-прежнему медленные операции включения и исключения элемента. Повысить скорость и этих операций нам поможет новая реализация таблицы под названием дерево двоичного поиска.

14.3. Деревья двоичного поиска

Удивительно, что мы видим деревья и больше не удивляемся им.

Ральф Уолдо Эмерсон, основатель философии трансцендентализма 🙄.

Деревья двоичного поиска (ДДП) (BST – bynary search tree), называемые также деревьями сравнений, являются двоичными деревьями особого вида. Каждый узел хранит один элемент таблицы, т.е. наряду со ссылками, там находится ключ и значение. Ключи, как и в упорядоченных таблицах, надо уметь сравнивать не только на равно и неравно, но и дополнительно на больше и меньше. На Паскале ДДП можно описать так (ссылка на родительский узел, как и раньше, не реализуется):

```
type Link=↑Node; { стрелка на Node }
     Node=record {Left, Right }
       L,R: Link; k: key; d: value;
     end;
     BST=Link; { Binary Search Tree }
var x,y: BST;
```

На распределения ключей в ДДП накладывается жёсткое ограничение. Требуется, чтобы для каждого узла все ключи из левого поддерева были меньше ключа в корне, а все ключи из правого поддерева были больше ключа в корне. Возможное ДДП для целочисленных ключей показано на рис. 14.1.

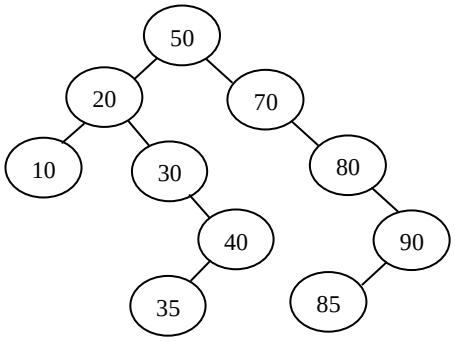

Рис. 14.1. Пример дерева двоичного поиска.

Реализуем операцию поиска в виде функции, которая получает как параметры ДДП и ключ и возвращает ссылку на узел с данным ключом, или **nil**, если такого ключа нет.

```
function SearchBST(x: BST; y: key): Link;
begin ❶{$B-}
  if (x=nil) or (x↑.k=y)
    then SearchBST:=x { пустое дерево или нашли }
    else                { продолжаем поиск }
      if x↑.k>y
{ 🡻🡻 ищем в левом поддереве }
        then SearchBST:=SearchBST(x↑.L,y)
{ 🡻🡻 ищем в правом поддереве }
        else SearchBST:=SearchBST(x↑.R,y)
  end
end;
```

В точке ❶ программа включает режим сокращённых вычислений логических выражений (нужно вспомнить, что это такое, впрочем, этот режим в языке Free Pascal и так включён по умолчанию), иначе при вычислении выражения `(x=nil) or (x↑.k=y)` возникнет ошибка при `x=nil`.Итак, для пустого дерева сразу возвращаем **nil**, иначе смотрим ключ в узле, если нашли, то тоже возвращаемся со ссылкой на корень. В противном случае ищем ключ либо в левом, либо в правом поддереве. Таким образом, за каждое сравнение ключа мы отбрасываем либо левое, либо правое поддерево как бесперспективное, спускаясь на один уровень вниз. Получается, что сложность этого алгоритма пропорциональна высоте дерева H, т.е. $O(H)$, таким образом для сбалансированных деревьев сложность равна $O(\log_2(N))$. Это совпадает со сложностью поиска в упорядоченном массиве, что не удивительно, потому что ДДП тоже в некотором смысле *линейная* структура данных, аналогичная векторам и спискам. Чтобы убедится в этом, рассмотрим такую процедуру:

```
function PrintKeys(x: BST);
begin
  if x<>nil then begin
    PrintKeys(x↑.L);
    Write(x↑.k:4);
    PrintKeys(x↑.R)
  end
end;
```

Эта процедура реализует так называемый левосторонний обход дерева. Проведите её трассировку для ДДП на рис. 14.1 и убедитесь, что она печатает все ключи этого дерева в возрастающем порядке

```
  10  20  30  35  40  50  70  80  85  90
```

Задание для самостоятельной работы. Измените эту процедуру так, чтобы она печатала все ключи в убывающем порядке (это просто) и по 10 ключей в строке (это сложно!).

Итак, для сбалансированных ДДП поиск это быстрая операция, а вот для «скособоченных» деревьев, у которых есть длинные ветви, операция будет работать медленнее. В предельном случае, когда ДДП вырождается в одну ветвь, сложность стремится к $O(N)$. Впрочем, на дереве с очень длинными ветвями, как мы уже разбирали, наша процедура вообще не работает.

Упорядоченность ДДП (как мы показали, оно подобно упорядоченному массиву!) делает возможным легко реализовать в языке Free Pascal операцию поиска не в рекурсивном виде, а с помощью цикла:

```
function BSTSearch(x: BST; y: key): Link;
begin ❶{$B-}
  while (x<>nil) and (x↑.k<>y) do begin
    if x↑.k>y
{k>y} then x:=x↑.L { ищем в левом поддереве }
{k<y} else x:=x↑.R { ищем в правом поддереве }
```

```
  end;
  BSTSearch:=x
end;
```

Понятно, что цикл работает значительно более эффективно, чем рекурсия. В точке ❶ программа, как уже говорилось выше, включает режим сокращённых вычислений логических выражений. Кроме того, мы воспользовались тем, что параметр x передан по значению, в функции его можно свободно менять, используя в качестве «обходчика» ДДП.

Задание для самостоятельной работы. Реализуйте с помощью цикла операцию поиска узла с максимальным ключом.

Рассмотрим теперь операцию включения в ДДП узла с новым ключом и значением. Что делать, если в ДДП уже есть узел с данным ключом? Давайте считать (это спецификация), что тогда производится модификация узла, т.е. старое *значение* (типа value в нашем описании) заменяется на новое. Идея нашего алгоритма в том, чтобы или найти узел, подлежащий модификации новым значением, либо найти тот узел, к которому должен быть прикреплён добавляемый узел в виде листа. Так как никакого небольшого результата наша операция не вырабатывает, то надо писать процедуру.

```
procedure AddBST(❶ var x: BST; y: key; z: value);
  var p,q: Link;
begin
  p:=x; q:=x;
  while (p<>nil) and (p↑.k<>y) do begin
    q:=p;
{ ⬇⬇ p спускается по дереву, q стоит на родительском узле }
    if p↑.k>y
      then p:=p↑.L     { ищем в левом поддереве }
      else p:=p↑.R     { ищем в правом поддереве }
  end;
{ ⬇⬇ p<>nil – нашли, p=nil – не нашли }
  if p<>nil then      { модификация }
    p↑.d:=z
  else begin          { делаем новый узел-лист }
    new(p); p↑.L:=nil; p↑.R:=nil;
    p↑.k:=y; p↑.d:=z;
    if q=nil then      { новый лист это корень }
      x:=p             { ❶ var x ! }
    else begin
      if q↑.k<y
        then q↑.R:=p  { новый узел – правый лист }
        else q↑.L:=p  { новый узел – левый лист }
    end
  end
end;
```

Здесь мы применили тот же метод, что и для вставки в односторонний список, где требовалось найти предыдущий (перед вставляемым) элемент списка. Аналогично, спускаясь по ДДП с помощью ссылочной переменной p, мы всё время держим ссылочную переменную q на родительском узле относительно указателя p. Значок ❶ показывает необходимость передачи ДДП x по ссылке. Сложность операции O(H).

Для простоты мы не обрабатывали ошибку исчерпания памяти для хранения динамических переменных, в этом случае оператор new вырабатывает исключительную ситуацию. В языке Free Pascal есть специальная функция, возвращающая размер доступной динамической памяти, её надо вызывать перед оператором new. В этом случае операцию включения в ДДП нового узла надо, например, делать в виде логической функции, возвращающей **false** при исчерпании памяти:

```
function AddBST(var x: BST; y: key; z: value): boolean;
```

Теперь перейдём к операции исключения из ДДП узла с заданным ключом. Для случая, когда такого ключа в ДДП уже нет, примем простое решение не считать это ошибкой и просто ничего не делать. Термин «исключение» будем понимать буквально, освобождая занимаемую узлом память с помощью операции `dispose`. Следовательно, надо реализовать эту операцию в виде процедуры. Заметим, что этот алгоритм удаления узла из ДДП предложил сам Н. Вирт.

Анализируя алгоритм удаления узла, мы видим три разные возможности. В самом простом случае удаляемый узле является листом, тогда просто удаляет его и ставим **`nil`** на нужном месте в родительском узле, или делая дерево пустым для листа, одновременно являющегося и корнем дерева. В более сложном случае у удаляемого узла только один потомок (левый иди правый) и надо просто сделать его потомком родительского узла (образно говоря, превратить внука в сына 🙂).

В самом сложном случае у удаляемого узла два потомка. Здесь мы поступим следующим образом: найдём в правом или левом поддереве удаляемого узла такой узел, который можно было бы подставить на место удаляемого, без нарушения структуры ДДП. Все три случая показаны на рис. 14.2 (удаляемый узел заштрихован, а возможные узлы для подстановки вместо удаляемого показаны жирным синим цветом).

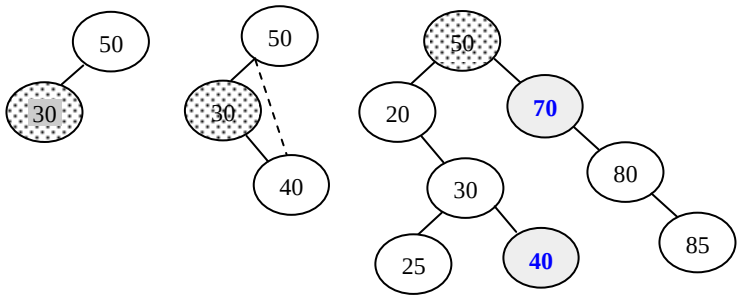

Рис. 14.2. Три случая удаления узла из ДДП.

Как видим, в третьем случае узел для подстановки вместо удаляемого можно выбрать даже двумя способами. Ключ подставляемого узла в правом поддереве минимальный, а в левом поддереве – максимальный. После того, как удаляемый и подставляемый на его место узлы поменять местами, задача сводится либо к первому, либо ко второму случаю. Заметим, что во всех случаев сложность операции, как и для случая добавления узла, пропорциональная высоте дерева O(H).

Процедура удаления получается хотя и быстрой, но достаточно сложной. Её можно немного упростить, если не «жадничать» и сохранить в каждом узле дерева дополнительную ссылку на родительский узел:

```
type Link=↑Node; { стрелка }
     Node=record
       L,R,U: Link; k: key; d: value
     end;
     BST=Link; { Binary Search Tree }
```

Надо понять, что операция поиска от этого не измениться, а операция включения узла изменится незначительно. Тогда получится такая процедура исключения узла:

```
procedure DelBST(❶var x: BST; y: key);
  var p,q: Link;
begin
  p:=x;
  while (p<>nil) and (p↑.k<>y) do
  begin
    if p↑.k>y
      then p:=p↑.L        { ищем в левом поддереве }
      else P:=p↑.R        { ищем в правом поддереве }
  end;
  if p<>nil then begin    { есть узел, удаление }
    if p↑.L=p↑.R then begin { это лист! }
```

```
      if p=x then begin { корень }
        dispose(x); ❶ x:=nil
      end else begin      { лист не корень }
        if p↑.U↑.L=p
          then p↑.U↑.L:=nil   { левый лист }
          else p↑.U↑.R:=nil;  { правый лист }
        dispose(p)
      end
    end { лист } else begin
    if (p↑.L=nil) or (p↑.R=nil) then begin
{ у исключаемого узла только один потомок }
      q:=p↑.L;                  { потомок левый }
      if q=nil then q:=p↑.R;  { потомок правый }
      if p↑.U↑.L=p then p↑.U↑.L:=q
                   else p↑.U↑.R:=q;
      dispose(p)
    end { один потомок } else begin
{ ================================= }
{ два потомка, ищем замену в левом поддереве }
      q:=p↑.L;
      while q↑.R<>nil do q:=q↑.R;
{ q на узле для замены, копия q в удаляемый узел  }
      p↑.k:=q↑.k; p↑.d:=q↑.d;
{ теперь надо удалить узел q }
      if q↑.L=q↑.R then begin { это правый лист }
        q↑.U↑.R:=nil; dispose(q)
      end { лист } else begin
{ узел q правый потомок, и у него один правый потомок }
      q↑.U↑.R:=q↑.R;
      dispose(q)
    end
  end
end;
```

Итак, для не очень разбалансированных ДДП все основные операции имеют сложность $O(\log_2(N))$. В то же время в процессе работы с ДДП могут производиться многочисленные операции включения и исключение узлов, и дерево может сильно разбалансироваться, как в этом случае быть?

Простейшим выходом является создание вспомогательной операции *балансировки* ДДП. Алгоритм балансировки требует дополнительного объёма памяти, равного объёму всех ключей и значений в ДДП. Сначала, используя, например, левосторонний обход ДДП, надо записать все значения по возрастанию ключей в массив типа

```
array BTS[1..N] of record k: key; d: Value end;
```

Другими словами, по ДДП мы построили таблицу в виде упорядоченного массива. Затем старое ДДП можно уничтожить, освобождая память, и начать построение нового, сбалансированного ДДП из упорядоченного массива. В корень надо записать ключ, расположенный в средине массива, при этом ключи, расположенные перед корневым, будут входить в левое поддерево, а ключи, расположенные после него – в правое. Далее с этими половинками массива надо поступить также: их середины взять за корни левого и правого поддерева и т.д. После построения дерева массив можно уничтожить. В результате получится идеально сбалансированное ДДП. Общая сложность алгоритма балансировки равна $O(N)$.

Другим решением проблемы разбансиро́вки является использование деревьев ДДП специального вида. В этих деревьях операции включения и исключения узлов будут дополнительно поддерживать балансировку дерева.

Часто используются так называемые AVL деревья (их создатели Г.М.Адельсон-Вельский и Е.М.Ландис). В этих деревьях каждый узел хранит также и баланс своих поддеревьев (`-1,0,+1`). После включения или исключения узла, если баланс выходит за эти пределы, то работают «хитрые» алгоритмы само балансировки.

Иногда используются *примерно* сбалансированные, так называемые красно-чёрные деревья (RB tree). В них допускается более сильный дисбаланс: для каждого узла высоты правого и левого поддеревьев должны различаться не более, чем в два раза. Для желающих разобраться по этой теме рекомендуется пособие «Сенюкова О.В. Сбалансированные деревья поиска. – М.: Издательский отдел факультета ВМК МГУ им. М.В. Ломоносова; МАКС Пресс, 2014, с. 68». Пособие в формате PDF можно загрузить из `https://algorithm.cs.msu.ru/semestr1`.

Сбалансированные ДДП обеспечивают для всех операций сложность $O(\log_2(N))$. В случае, когда нам потребуется выполнять эти операции ещё быстрее, придётся сделать другую реализацию абстрактной структуры данных таблица.

14.4. Хэш-таблицы

Тебе что-то непонятно? Перечитай и перепиши это несколько раз. Сначала непонятное станет привычным, а затем привычное – понятным.

Студенческая мудрость

Итак, нам бы хотелось делать все операции над таблицами максимально быстро, желательно, чтобы их сложность была $O(1)$. Это очень легко сделать, построив так называемую таблицу прямого отображения. Опишем нужные для этого типы на Паскале:

```
const N=1000000; { число элементов N }
type
  key=1..N;     { тип ключа }
  keys=0..N;    { «расширенный» ключ }
  Elem=record { элемент таблицы }
    k: keys; d: data;
  end;
  Tab=array[1..N] of Elem; { тип таблицы }
var x: Tab;
```

«Расширенный» ключ включает в себя дополнительное служебное нулевое значение, которое обозначает, что данный элемент в таблице отсутствует. Перед началом работы таблицу надо инициализировать (сделать пустой), это можно сделать с помощью цикла:

```
for i:=0 to N do x[i].k:=0
```

Функция поиска возвращает либо ноль, либо индекс нужного элемента `i` в `Tab`:

```
function TabSearch(var ❶x: Tab; y: key): keys;
begin TabSearch:=x[y].k end;
```

Объясните, почему нельзя передать таблицу по значению, убрав ❶ **var**.

Операцию вставки нового элемента реализуем как в ДДП, т.е. при наличии ключа в таблице будем просто заменять значение типа `data` на новое:

```
procedure TabAdd(var x: Tab; y: key; z: data);
begin x[y].k:=y; x[y].d:=z;
{ или «красиво» with x[y] do begin k:=y; d:=z end }
end;
```

И, наконец, операция исключения (исключение несуществующего ключа будем, как и для ДДП, считать пустой операцией):

```
procedure TabDel(var x: Tab; y: key);
begin x[y].k:=0 end;
```

Вот и всё! Все операции имеют сложность O(1), их текст умещается в одну строку и предельно понятен. Делайте так, и будет Вам счастье ⚠. Так, спрашивается, зачем же мы изучали другие реализации таблиц?

К сожалению, так реализованная таблица прямого отображения (скоро мы поймём, почему у неё такое название), оказывается практически бесполезной почти во всех задачах информационного поиска. Дело в том, что вид ключа в задаче информационного поиска определяет не программист, а заказчик информационной системы. Такие ключи существовали задолго до «компьютерной эры».

Например, на заводе в бухгалтерии у каждого рабочего есть такой ключ, обычно его называют *табельным номером* (или как-то похоже). Обычно этот номер имеет такую структуру:

<№ цеха><№ участка в цехе><№ бригады в участке><№ рабочего в бригаде>

Часто номера цехов, участков и бригад двузначные, а номер рабочего трёхзначный, например, у некоторого рабочего табельный номер 02 05 13 103 (пробелы добавлены для читабельности). Таким образом, число принципиально возможных номеров равно 10^9, в то время как номеров «настоящих» рабочих вряд ли больше нескольких тысяч. Ещё одним примером служит знакомые Вам «ключи» книг в библиотечной картотеке, например, УДК: 621.382.323 или ISBN 978-5-699-12014-7.

Такая ситуация часто встречается в задачах обработки данных. Хорошим примером служат так называемые разре́женные матрицы, они часто возникают при решении дифференциальных уравнений в частных производных. В таких матрицах «настоящих» (ненулевых) элементов много меньше процента, и возникает проблема, как такие матрицы хранить в памяти компьютера и обрабатывать.

В качестве ещё одного примера рассмотрим ключ в информационной системе «Студент» нашего университета. Этот ключ является номером пропуска и зачётной книжки студента, он имеет вид:

<№ факультета><год поступления><№ студента на факультете> [1]

Двузначный <№ факультета> нумерует факультеты в Москве и во всех филиалах, двузначного номера года поступления хватит где-то до 2070 года (информационная система «Студент» заработала в конце 70-х годов прошлого века). Под номер студента отведено 4 цифры. Таким образом, это 10^8 потенциальных ключей. Конечно, обучающихся к каждый момент студентов (на всех курсах и во всех филиалах) не более 50000, таким образом «настоящих» ключей много меньше 1%.[2]

Номер 01 присвоен Механико-математическому факультету (ничего не поделаешь, уже давно ректор нашего Университета именно с этого факультета ☺). Зато номер 02 присвоен факультету ВМК (систему делали наши люди!), номер 03 у физиков, 04 у химиков и т.д. Ну, а у разных филологов и экономистов уже двузначные номера.

Ясно, что такая таблица на 10^8 записей о студентах (≈ 6 Tb) не войдёт в память даже многих 64-битных ЭВМ. Да и на дисках хранить такую (практически пустую) таблицу никто не позволит. Надо что-то делать… Вот тут нам и помогут хэш-таблицы (hash-table). Для начала дадим следующие определения.

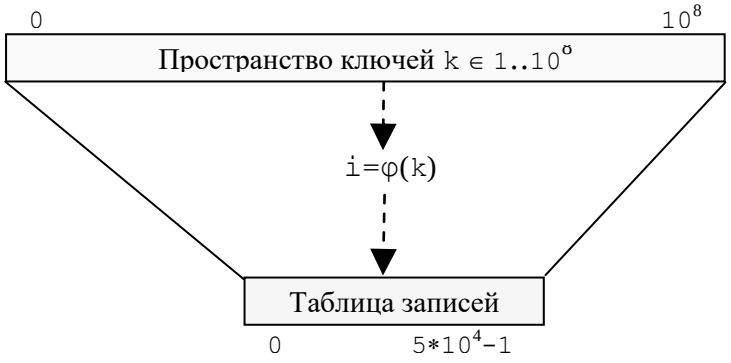

Рис. 4.3. Хэш-функция отображает ключи $k \in 0..10^8$ в индексы $i \in 0..5*10^4-1$.

Пространством ключей назовём диапазон всех возможных ключей, в нашем примере со студентами это $1..10^8$. Обычно туда добавляют некоторые служебные ключи, мы добавим ключи 0 и −1.

---

[1] На некоторых факультетах № студента разбивается на № отделения и № студента на этом отделении.

[2] Конечно, возможен архивный доступ к записям студентов прошлых лет, но там отдельные операции.

Пространством таблицы назовём индексы массива, в котором будут храниться все элементы таблицы, у нас это, например, будет массив `0..49999` записей о студентах. Конечно, предполагается, что число всех обучающихся сейчас студентов меньше `50000`.

Функция φ, отображающая пространство ключей на пространство таблицы, называется (первичной) хэш-функцией (hash function, см. рис 14.3). Так как пространство ключей много больше пространства таблицы, то φ задаёт сжимающее отображение: много разных ключей отображаются в одну позицию таблицы. А вот для рассмотренной ранее таблицы прямого отображения можно считать, что у неё тоже есть своя хэш-функция, но она задаёт тождественное (прямое) отображение (отсюда понятно название этой таблицы).

На языке Паскаль получатся следующие описания:

```
const Nk=100000000; { число ключей }
      Nt=50000;     { число студентов }
type  key=0..Nk-1;  { тип ключа }
      keys=-2..Nk-1; { расширенный тип ключа }
      ind=key;      { тип индекса таблицы }
      Elem=record   { элемент таблицы }
        k: keys; d: data;
      end;
      Tab=array[ind] of Elem; { таблица }
var T: Tab;
function fi(x: key): ind; forvard;
procedure InitTab(var x: Tab);
  var i: integer;
begin for i:=0 to Nt-1 do x[i].k:=-1 end;
```

Служебные ключи `-1` и `-2` означают отсутствие элемента в таблице, причём значение `-2` является признаком того, что раньше на этом месте таблицы находился элемент, который затем был исключён. Перед началом работы таблицу надо *инициализировать*, присвоив всем ключам служебное значение `-1`, это делает процедура `InitTab`. Хэш-функция φ(k) вычисляет предполагаемое место в таблице, где находится элемент с данным ключом. Так как функция сжимающая, то на этом месте таблицы может располагаться запись с другим ключом, такая ситуация называется коллизией. Операции над хэш-таблицами должны правильно обрабатывать коллизии.

Другими названиями хэш-функции являются функция расстановки и функция свёртки. Действительно, она «расставляет» все ключи по таблице или «сворачивает» длинные ключи в короткие индексы таблицы. Буквальный перевод будет «функция перемешивания» (to hash – крошить, перемешивать, делать фарш), она пытается перемешать ключи, чтобы они во возможности *равномерно* попадали в таблицу.

В качестве простой хэш-функции можно взять `φ(key)=key mod Nt` (у нас `key>=0`):

```
function fi(x: key):ind;
begin fi:=key mod Nt end;
```

Эта хэш-функция даёт очень много коллизий, убедитесь, что студенты с номером № из всех факультетов попадают на одно место таблицы. Теоретически, так как студентов меньше, чем размер таблицы, то можно написать *совершенную* хэш-функцию без коллизий, в математике такая функция называется инъективной (вспомним нашу таблицу прямого отображения). Практически, однако, сделать это почти всегда невозможно. Выбор хорошей хэш-функции является одной из основных задач разработчиков информационной системы. Можно, например, предположить, что хэш-функция $\varphi(key)=key^2 \bmod Nt$ будет давать значительно меньше коллизий.

Итак, если `T[φ(k)].k=k`, то мы нашли нужную запись о студенте и можем с ней работать. Ключ со служебным значением `-1` будут означать, что студента с таким ключом в таблице нет. В противном случае случилась коллизия и хэш-функция привела нас к *другому* студенту. Для устранения

коллизии в основном применяются два метода: вторичная хэш-функция [1] и списки записей с коллизиями, мы рассмотрим только первый из них.

Вторичная хэш-функция $\psi(i)\ i\in ind$ отображает индексы таблицы на себя и она *инъективна* на ind. Другими словами, требуется, чтобы $\forall j \in ind\ \exists\ i \in ind$, такой, что $\psi(i)=j$. Простейшим примером является функция $\psi(i)=(i+1)\ \mathbf{mod}\ Nt$, она последовательно движется по таблице, а когда дойдёт до последнего элемента, то появляется на нулевом элементе. Можно представить, что наша таблица как бы замкнута в кольцо. Такой поиск иногда называют индексно-последовательным: сначала по ключу с помощью (первичной) хэш-функции мы попадаем на некоторый индекс таблицы, а потом с помощью вторичной хэш-функции просматриваем все элементы таблицы «в круговую», двигаясь сверху вниз.

Напишем операцию поиска элемента по ключу в виде функции, она вырабатывает индекс таблицы, где находится нужный элемент, либо −1, если такого ключа в таблице нет. В качестве вторичной хэш-функции будем использовать описанную выше функцию ψ(i), которая делает полный круговой перебор записей таблицы:

```
function HashSearch(var x: Tab; y: key): keys;
  var i,j: ind; k: keys;
begin
  i:=fi(y); { i:=φ(y) } k:=x[i].k;
  j:=(i-1) mod Nt; { j это предыдущее (по кольцу) значение i }
  while (k<>y) and (k<>-1) and (i<>j) do begin
     i:=(i+1) mod Nt; { i:=ψ(i) }
     k:=x[i].k          { перебор по кольцу }
  end;
  HashSearch:=i;
  if k<>y then HashSearch:=-1
end;
```

Значение −1 наша функция вырабатывает, когда встречает при просмотре служебный нулевой ключ, либо безрезультатно просмотрев «в круговую» всю таблицу. Обратите внимание, что прекращает поиск только нахождение ключа −1, но не ключа −2 ⚠.

Какова сложность операции поиска? В первую очередь, конечно, сложность зависит от качества хэш-функции, т.е. от числа коллизий, которые она даёт. Видно, что на каждую коллизию тратится одно выполнение цикла **while**. Кроме того, число коллизий сильно зависит и от степени заполнения таблицы. Действительно, чем меньше в таблице остаётся свободных элементов (со служебными ключами −1 и −2), тем чаще будут происходить коллизии.

Моделирование на компьютере показало, что для «достаточно хороших» хэш-функций и при наличии 5-10% свободного места в таблице поиск занимает в среднем 1-3 сравнений ключей. При уменьшении свободного места в таблице сложность поиска стремиться к O(N). Следовательно, достаточно обеспечивать определённый запас свободного места в таблице, и сложность поиска будет O(1).

Рассмотрим теперь операцию включения в таблицу нового элемента. Как и прежде, сделаем спецификацию, что когда элемент с таким ключом уже есть, то это будет обозначать замену старых данных на новые. Реализуем операцию в виде логической функции, которая будет возвращать **false**, когда таблица переполнена, и включить новый элемент нельзя.

```
function HashAdd(var x: Tab; y: key;
                 ❶ var z: Value): boolean;
  var i,j: ind; k: keys;
begin
  i:=fi(y); { i:=φ(y) } k:=x[i].k; HashAdd:=true;
  j:=(i-1) mod Nt; { предыдущее (по кольцу) значение i }
```

[1] Таблицы со вторичной хэш-функцией «официально» называются таблицами с *открытой адресацией*, но это мало что проясняет.

```
  while (k<>y) and (k<0) and (i<>j) do begin
    i:=(i+1) mod Nt; { i:=ψ(i) }
    k:=x[i].k          { перебор по кольцу }
  end;
  if k=y then x[i].d:=z { модификация } else
  if i<>j then begin { k=-1 или k=-2 – новый элемент }
     x[i].k:=y; x[i].d:=z
  end else HashAdd:=false { переполнение }
end;
```

При включении нового элемента ищется свободная позиция таблицы, у неё ключ будет равен `-1` или `-2`. Необходимость передачи параметра `z` по ссылке ❶ **`var z`** определяется его размером: для небольших `z` его можно передавать и по значению.

Операцию исключения элемента таблицы реализуем в виде процедуры. Как и прежде, сделаем спецификацию, что исключение несуществующего ключа не является ошибкой и будет просто пустой операцией (иначе пришлось бы делать эту операцию в виде логической функции):

```
procedure HashDel(var x: Tab; y: key);
  var i,j: ind; k: keys;
begin
  i:=fi(y); { i:=φ(y) } k:=x[i].k;
  j:=(i-1) mod Nt; { предыдущее (по кольцу) значение i }
  while (k<>y) and (k<>-1) and (i<>j) do begin
    i:=(i+1) mod Nt; { i:=ψ(i) }
    k:=x[i].k          { перебор по кольцу }
  end;
  if k=y then x[i].k:=-2 { исключение }
end;
```

➕ С использованием уже написанной операции поиска можно переписать эту операцию так:

```
procedure HashDel(var x: Tab; y: key);
begin
  if HashSearch(x,y)<>-1 then
     x[i].k:=-2 { исключение }
end;
```

Задание для самостоятельной работы. Объясните, зачем понадобился служебный ключ `-2` и почему нельзя обойтись одним служебным ключом `-1`.

В качестве примера решения задач на хэш-таблицы рассмотрим задачу на экзамене по данному курсу на факультете Вычислительной математики и кибернетики МГУ.[1] Пусть задана хэш-таблица длиной 13 элементов:

```
type keys=-2..12;
     Tab=array[0..12] of Elem;
var  x: Tab;
```

Для этой таблицы задана первичная хэш-функция `φ(k)=k mod 13` и вторичная хэш-функция `ψ(i)=(i+4) mod 13`. В первоначально пустую таблицу `x` последовательно заносятся элементы с ключами 19, 27, 23, 6 и 18. Надо нарисовать заполненную таблицу `x`, в которой указаны только ключи, ответ показан ниже.

Индекс	0	1	2	3	4	5	6	7	8	9	10	11	12
Ключ	-1	**27**	-1	-1	-1	**6**	**19**	-1	-1	**18**	**23**	-1	-1

[1] Иванников В.П., Корухова Л.С., Пильщиков В.Н. Курс «Алгоритмы и алгоритмические языки»'. Варианты письменного экзамена. – М., МГУ, 2002, 48 с.pdf

Позиции занесённых в таблицу ключей обозначены жирным текстом, пустые позиции отмечены ключом `-1`. Обязательно поймите, что, например, при занесении ключа 6 были три коллизии. А вот если взять в качестве вторичной хэш-функции ψ`(i)=(i+1) mod 13`, то получится такая таблица:

Индекс	0	1	2	3	4	5	6	7	8	9	10	11	12
Ключ	-1	**27**	-1	-1	-1	**18**	**19**	**6**	-1	-1	**23**	-1	-1

Здесь возникла только одна коллизия при занесении ключа 6.

Итак, с использованием хэш-таблиц мы достигли теоретического минимума сложности всех операций O(1). Эти таблицы широко используются в самых разных задачах информационного поиска. Например, компиляторы с языков программирования используют хэш-таблицу для хранения имён пользователя, сами имена используются как ключи. Для уникальности ключей при совпадающих именах используются спецификаторы модулей, блоков или записей, куда входит это имя, например

```
program.X      MyProc.X      System.TextAttr
```

Это позволяет быстро искать вхождения имён и включать в таблицу новые имена. При этом легко определяются ошибки вида «неописанное имя» и «дважды описанное имя». Отметим, что и файловая система часто использует хеширование имён файлов на больших дисках для их быстрого поиска.

У Вас должен возникнуть закономерный вопрос: если это лучшая реализация, то зачем мы изучали реализации таблиц в виде массивов и ДДП? Ясно, что у хэш-таблиц должен быть какой-то существенный недостаток. Этот недостаток связан с самим принципом устройства хэш-таблицы. Вспомним, что ДДП является упорядоченной структурой данных, мы хорошо видели это на операциях левостороннего и правостороннего обхода дерева. Другими словами, для элемента с любым ключом легко находятся элементы со следующим и предыдущим ключами в ДДП.

А вот хэш-таблица является не упорядоченной структурой данных, сложность операции нахождения следующего существующего ключа в таблице эквивалентна полному перебору. Современные компиляторы редко предоставляют программисту опцию «выдать все имена программы в алфавитном порядке». Действительно, в языке Free Pascal пространство имён это примерно 40^{127} (26 букв + 10 цифр + особые «буквы»). Так что тем компиляторам, которые предоставляют такую опцию, приходится сначала извлекать все имена в массив, потом сортировать их, а лишь затем печатать.

Другим нехорошим качеством хэш-таблиц является то, что она является не динамической, а псевдо-динамической структурой данных, так как память должна резервироваться под максимальный объем хранимых данных (и даже с запасом в 5-10%). При переполнении хэш-таблицы надо делать её полную реорганизацию, переписывая все данные из старой таблицы в новую.

Итак, если нам не нужна в таблице упорядоченность, то хэш-таблица – наилучший выбор. Отдельно следует упомянуть такое полезное свойство хэш-таблиц, как масштабируемость. Легко реализовать таблицу, размер которой многократно превосходит оперативную память компьютера и располагается во внешней памяти («на дисках»). Нужно просто нарезать таблицы на куски (файлы), каждый из которых входит в память ЭВМ. По ключу хэш-функция сразу укажет на конкретную часть (файл) такой таблицы, надо считать его в память, а вторичное хеширование будет просто перебирать в нём элементы, пока не найдёт нужный. Ясно, что примерно так должны быть устроены «большие» системы, например, база ГИБДД по автомобильным номерам.

Вопросы и упражнения

Всякий человек имеет естественное желание знать.

Аристотель, IV век до н.э.

1. Чем задача информационного поиска отличается просто от поиска нужной информации, скажем, в Интернете?
2. Почему таблица является неупорядоченной абстрактной структурой данных?
3. Какие есть основные операции над таблицами и как оценивается сложность реализации этих операций?
4. Почему ключами в деревьях двоичного поиска (ДДП) не могут быть комплексные числа?
5. В каком смысле ДДП является линейной последовательностью узлов?
6. Какое двоичное дерево называется сбалансированным?
7. Какая сложность у операции балансировки ДДП?

8. Могут ли ключами в хэш-таблице быть комплексные числа?
9. Почему таблицу прямого отображения нельзя использовать в практических задачах?
10. Почему хэш-функцию называют функцией перемешивания?
11. Почему вторичная хэш-функция должна быть инъективной (и что это такое)?
12. Если хэш-таблица даёт лучшее время выполнения операций, то зачем может понадобиться использовать ДДП?
13. Почему хэш-таблица может обеспечить масштабируемость (и что это такое)?