Тема 5.
ПРИМЕНЕНИЕ НЕКОТОРЫХ АБСТРАКТНЫХ ТИПОВ ДАННЫХ ДЛЯ РЕАЛИЗАЦИИ МНОЖЕСТВ

Множество является той базовой структурой, которая лежит в основании всей математики. При разработке алгоритмов множества используются как основа многих важных абстрактных типов данных, и многие технические приемы и методы разработаны для реализации абстрактных типов данных, основанных именно на множествах.

В этой теме мы рассмотрим основные операторы, выполняемые над множествами, и опишем некоторые простые реализации множеств.

Мы представим "словарь" и "очередь с приоритетами" - два АТД, основанных на модели множеств.

Введения в множества

Множеством называется некая совокупность элементов, каждый элемент множества или сам является множеством, или является примитивным элементом, называемым **атомом**.

Далее будем предполагать, что все элементы любого множества различны, т.е. в любом множестве нет двух копий одного и того же элемента.

Когда множества используются в качестве инструмента при разработке алгоритмов и структур данных, то атомами обычно являются целые числа, символы, строки символов и т.п., и все элементы одного множества, как правило, имеют одинаковый тип данных.

Введения в множества

Мы часто будем предполагать, что атомы линейно упорядочены с помощью отношения, обычно обозначаемого символом "<" и читаемого как "меньше чем" или "предшествует".

Линейно упорядоченное множество ***S*** удовлетворяет следующим двум условиям:

1) Для любых элементов ***a*** и ***b*** из множества ***S*** может быть справедливым только одно из следующих утверждений: $a<b$, $a=b$ или $b<a$.
2) Для любых элементов ***a***, ***b*** и ***c*** из множества ***S*** таких, что $a<b$ и $b<c$, следует $a<c$ (свойство транзитивности).

Введения в множества

Множества целых и действительных чисел, символов и символьных строк обладают естественным линейным порядком, для проверки которого можно использовать оператор отношения $<$ языка Pascal.

Понятие линейного порядка можно определить для объектов, состоящих из множеств упорядоченных объектов. Формулировку такого определения сделайте в качестве упражнения.

Для множеств в дискретной математике принята соответствующая система обозначений. Над множествами определен ряд стандартных операций. Основные из них:

- объединения множеств
- пересечения множеств
- разности множеств

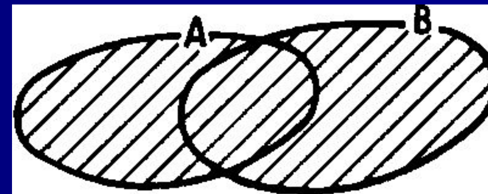
Объединение множеств

Объединением двух множеств является третье множество, содержащее элементы обоих множеств (**A** ∪ **B**).

Например:

Значение ***A***	Значение ***B***	Выражение	Результат
[1, 2, 3]	***[1, 4, 5]***	***A+B***	***[1, 2, 3, 4, 5]***
[‘A’..’D’]	***[‘E’..’Z’]***	***A+B***	***[‘A’..’Z’]***
[ ]	***[ ]***	***A+B***	***[ ]***

Графическая интерпретация:



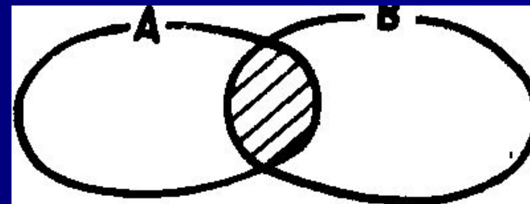
Пересечение множеств

Пересечением двух множеств является третье множество, которое содержит элементы, входящие одновременно в оба множества (**A**∩**B**).

Например:

Значение ***A***	Значение ***B***	Выражение	Результат
[1, 2, 3]	***[1, 4, 2, 5]***	***A * B***	***[1, 2]***
[‘A’..’Z’]	***[‘B’..’R’]***	***A * B***	***[‘B’..’R’]***
[ ]	***[ ]***	***A * B***	***[ ]***

Графическая интерпретация:



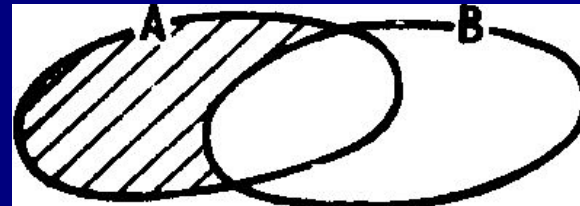
Разность множеств

Разностью двух множеств является третье множество, которое содержит элементы первого множества, не входящие во второе множество (**A \ B**).

Например:

Значение ***A***	Значение ***B***	Выражение	Результат
[1, 2, 3, 4]	***[3, 4, 1]***	***A – B***	***[2]***
[‘A’..’Z’]	***[‘D’..’Z’]***	***A – B***	***[‘A’..’C’]***
[X1, X2, X3, X4]	***[X4, X1]***	***A – B***	***[X2, X3]***

Графическая интерпретация:



Операторы АТД, основанные на множествах

1. Процедуры ***UNTON(A, B, C)*** имеет "входными" аргументами множества ***A*** и ***B***, а в качестве результата - "выходное" множество ***C***, равное $A \cup B$.

2. Процедуры ***INTERSECTION(A, B, C)*** имеет "входными" аргументами множества ***A*** и ***B***, а в качестве результата - "выходное" множество ***C***, равное $A \cap B$.

3. Процедуры ***DIFFERENCES(A, B, C)*** имеет "входными" аргументами множества ***A*** и ***B***, а в качестве результата - "выходное" множество ***C***, равное $A \setminus B$.

Операторы АТД, основанные на множествах

4. Иногда используетсяь оператор, который называется слияние (merge), или объединение непересекающихся множеств. Этот оператор (обозначается ***MERGE***) не отличается от оператора объединения двух множеств, но здесь предполагается, что множества-операнды не пересекаются (т.е. не имеют общих элементов). Процедура ***MERGE(A, B, C)*** присваивает множеству ***C*** значение $A \cup B$, но результат будет не определен, если $A \cap B \neq \varnothing$, т.е. в случае, когда множества ***A*** и ***B*** имеют общие элементы.

Операторы АТД, основанные на множествах

5. Функция ***MEMBER(x, A)*** имеет аргументами множество ***A*** и объект ***x*** того же типа, что и элементы множества ***A***, и возвращает булево значение ***true*** (истина), если $x \in A$, и значение ***false*** (ложь), если $x \notin A$.

6. Процедура ***MAKENULL(A)*** присваивает множеству ***A*** значение пустого множества.

Операторы АТД, основанные на множествах

7. Процедура ***INSERT(x, A)***, где объект ***x*** имеет тот же тип данных, что и элементы множества ***A***, делает ***x*** элементом множества ***A***. Другими словами, новым значением множества ***A*** будет $A \cup \{x\}$. Отметим, что в случае, когда элемент ***x*** уже присутствует в множестве ***A***, это множество не изменяется в результате выполнения данной процедуры.

8. Процедура ***DELETE(x, A)*** удаляет элемент ***x*** из множества А, т.е. заменяет множество ***A*** множеством $A \setminus \{x\}$. Если элемента ***x*** нет в множестве ***A***, то это множество не изменяется.

Операторы АТД, основанные на множествах

9. Процедура ***ASSIGN(A, B)***, присваивает множеству ***A*** в качестве значения множество ***B***.

10. Функция ***MIN(A)*** возвращает наименьший элемент множества ***A***. Для применения этой функции необходимо, чтобы множество ***A*** было параметризировано и его элементы были линейно упорядочены. Например, ***MIN(*{**2, 3, 1**}*)*** = 1 и ***MIN(('a', 'b', 'c'})*** ='a'. Подобным образом определяется функция ***MAX***.

Операторы АТД, основанные на множествах

11. Функция ***EQUAL(A, B)*** возвращает значение ***true*** тогда и только тогда, когда множества ***A*** и ***B*** состоят из одних и тех же элементов.

12. Функция ***FIND(x)*** оперирует в среде, где есть набор непересекающихся множеств. Она возвращает имя (единственное) множества, в котором есть элемент ***x***.

АТД с операторами множеств

Начнем с определения АТД для математической модели "множество" с определенными тремя основными теоретико-множественными операторами объединения, пересечения и разности. Сначала приведем пример такого АТД и покажем, как его можно использовать, а затем обсудим некоторые простые реализации этого АТД.

Реализация множеств посредством двоичных векторов

- Наилучшая конкретная реализация абстрактного типа данных ***SET*** (Множество) выбирается исходя из набора выполняемых операторов и размера множества.
- Если все рассматриваемые множества будут подмножествами небольшого универсального множества целых чисел **1**, ..., ***N*** для некоторого фиксированного ***N***, тогда можно применить реализацию АТД ***SET*** посредством двоичного (булева) вектора.
- В этой реализации множество представляется двоичным вектором, в котором ***i***-й бит равен **1** (или ***true***), если ***i*** является элементом множества.
- Главное преимущество этой реализации состоит в том, что здесь операторы ***MEMBER***, ***INSERT*** и ***DELETE*** можно выполнить за фиксированное время (независимо от размера множества) путем прямой адресации к соответствующему биту.
- Но операторы ***UNION***, ***INTERSECTION*** и ***DIFFERENCE*** выполняются за время, пропорциональное размеру универсального множества.

Реализация множеств посредством двоичных векторов

Если универсальное множество так мало, что двоичный вектор занимает не более одного машинного слова, то операторы ***UNION***, ***INTERSECTION*** и ***DIFFERENCE*** можно выполнить с помощью простых логических операций (конъюнкции и дизъюнкции), выполняемых над двоичными векторами.

В языке Pascal для представления небольших множеств можно использовать встроенный тип данных ***set***.

Максимальный размер таких множеств зависит от конкретного применяемого компилятора и поэтому не формализуем.

Реализация множеств посредством двоичных векторов

Однако при написании программ мы не хотим быть связаны ограничением максимального размера множеств по крайней мере до тех пор, пока наши множества можно трактовать как подмножества некоего универсального множества {**1**, ..., ***N***].

Поэтому в данной теме будем придерживаться представления множества в виде булевого массива ***A***, где ***A[i] = true*** тогда и только тогда, когда ***i*** является элементом множества.

С помощью объявлений языка Pascal АТД ***SET*** можно определить следующим образом:

```
const   N = { подходящее числовое значение };
type
    SET = array[1..N]  of  boolean;
```

Листинг 5.2. Реализация оператора UNION

```
procedure   UNION ( A, B: SET; var  C: SET );
var   i: integer;
begin
      for  i:= 1   to   N  do
            C[i]:= A[i] or B[i]
end;
```

Реализация множеств посредством двоичных векторов

Для реализации операторов *INTERSECTION* и *DIFFERENCE* надо в листинге 5.2 заменить логический оператор *or* на операторы *and* и *and not* соответственно.

Реализация множеств посредством связанных списков

Представление посредством связанных списков является более общим, поскольку здесь множества не обязаны быть подмножествами некоторого конечного универсального множества.

В отличие от представления множеств посредством двоичных векторов, в данном представлении занимаемое множеством пространство пропорционально размеру представляемого множества, а не размеру универсального множества.

Реализация множеств посредством связанных списков

При реализации оператора ***INTERSECTION*** (Пересечение) в рамках представления множеств посредством связанных списков есть несколько альтернатив.

Если универсальное множество ***линейно упорядочено***, то в этом случае множество можно представить в виде сортированного списка, т.е. предполагая, что все элементы множества сравнимы посредством отношения "**<**", можно ожидать, что эти элементы в списке будут находиться в порядке e_1, e_2, e_3, ..., e_n, когда $e<e_2<e_3<...<e_n$.

Преимущество отсортированного списка заключается в том, что для нахождения конкретного элемента в списке нет необходимости просматривать весь список.

Реализация множеств посредством связанных списков

Элемент будет принадлежать пересечению списков $\mathbf{L_1}$ и $\mathbf{L_2}$ тогда и только тогда, когда он содержится в обоих списках.

В случае ***несортированных списков*** мы должны сравнить каждый элемент списка $\mathbf{L_1}$ с каждым элементом списка $\mathbf{L_2}$, т.е. сделать порядка $O(n^2)$ операций при работе со списками длины ***n***.

Для ***сортированных списков*** операторы пересечения и некоторые другие выполняются сравнительно просто:

- если надо сравнить элемент е списка $\mathbf{L_1}$ с элементами списка $\mathbf{L_2}$, то надо просматривать список $\mathbf{L_2}$ только до тех пор, пока не встретится элемент **е** или больший, чем ***е***. В первом случае будет совпадение элементов, второй случай показывает, что элемента **е** нет в списке $\mathbf{L_2}$.

Реализация множеств посредством связанных списков

Более интересна ситуация, когда мы знаем элемент ***d***, который в списке $\mathbf{L_1}$ непосредственно предшествует элементу **e**.

Тогда для поиска элемента, совпадающего с элементом **e**, в списке $\mathbf{L_2}$ можно сначала найти такой элемент ***f***, что $d<f$, и начать просмотр списка $\mathbf{L_2}$ с этого элемента.

Используя этот прием, можно найти совпадающие элементы в списках $\mathbf{L_1}$ и $\mathbf{L_2}$ за один проход этих списков, продвигаясь вперед по спискам в прямом порядке, начиная с наименьшего элемента.

Реализация множеств посредством связанных списков

В листингах основных операций множества представлены связанными списками ячеек, чей тип определяется следующим образом:

```
type
    celltype = record
        element: elementtype;
        next: ^celltype
    end ;
```

В листинге 5.3 предполагается, что ***elementtype*** - это тип целых чисел, которые можно упорядочить посредством обычного оператора сравнения <.

Если ***elementtype*** представлен другим типом данных, то надо написать функцию, которая будет определять, какой из двух заданных элементов предшествует другому элементу.

Листинг 5.3. Процедура INTERSECTION, использующая связанные списки

procedure INTERSECTION (ahead, bhead: ^celltype;
var pc: ^celltype) ;

{ Вычисление пересечения сортированных списков ***A*** и ***B*** с ячейками заголовков ***ahead*** и ***bhead***, результат - сортированный список, на чей заголовок указывает ***pc*** }

var acurrent, bcurrent, ccurrent: ^celltype; {текущие ячейки списков ***A*** и ***B*** и последняя ячейка дополнительного списка ***C***}

begin

(1) new(pc); { создание заголовка для списка ***C*** }

(2) acurrent:= ahead^ .next;

(3) bcurrent:= bhead^ .next;

(4) ccurrent:= pc;

Листинг 5.3.

```
(5)   while (acurrent <> nil) and {bcurrent <> nil)  do  begin
          { сравнение текущих элементов списков A и B }
(6)       if  acurrent^.element = bcurrent^.element  then  begin
              { добавление элемента в пересечение }
(7)           new(ccurrentt.next) ;
(8)           ccurrent:= ccurrent^.next;
(9)           ccurrent^.element:= acurrent^.element;
(10)      acurrent:= acurrent^ .next;
(11)      bcurrent:= bcurrent^ .next
          end
```

Листинг 5.3.

```
        else      { элементы неравны }
(12)    if  acurrent^.element < bcurrent^.element  then
(13)          acurrent:= acurrent^.next;
(14)     else     bcurrent:= bcurrent^.next
      end
(15)  ccurrent^.next:= nil
end;    { INTERSECTION }
```

Реализация множеств посредством связанных списков

Связанные списки в листинге 5.3 в качестве заголовков имеют пустые ячейки, которые служат как указатели входа списков. При желании можно написать эту программу в более общей абстрактной форме с использованием примитивов списков.

Но программа листинга 5.3 может быть более эффективной, чем абстрактная программа.

Например, в листинге 5.3 используются указатели на отдельные ячейки вместо "позиционных" переменных, указывающих на предыдущую ячейку. Так можно сделать вследствие того, что элементы добавляются в список ***C***, а списки ***A*** и ***B*** только просматриваются без вставки или удаления в них элементов.

Реализация множеств посредством связанных списков

Процедуру *INTERSECTION* листинга 5.3 можно легко приспособить для реализации операторов *UNION* и *DIFFERENCE*.

Для выполнения оператора *UNION* надо все элементы из списков ***A*** и ***B*** записать в список ***C***. Поэтому, когда элементы не равны (строки 12-14 в листинге 5.3), наименьший из них заносится в список ***C***, так же, как и в случае их равенства. Элементы заносятся в список ***C*** до тех пор, пока не исчерпаются оба списка, т.е. пока логическое выражение в строке 5 не примет значение ***false***.

В процедуре *DIFFERENCE* в случае равенства элементов они не заносятся в список ***C***. Если текущий элемент списка ***A*** меньше текущего элемента списка ***B***, то он (текущий элемент списка ***A***) заносится в список ***C***. Элементы заносятся в список ***C*** до тех пор, пока не исчерпается список ***A*** (логическое условие в строке 5).

Реализация множеств посредством связанных списков

Оператор ***ASSIGN(A, B)*** копирует список ***A*** в список ***B***. Этот оператор нельзя реализовать простым переопределением заголовка списка ***B*** на заголовок списка ***A***, поскольку при последующих изменениях в списке ***B*** надо будет делать аналогичные изменения в списке ***A***, что, естественно, может привести к нежелательным коллизиям.

Оператор ***MIN*** реализуется легко - просто возвращается первый элемент списка.

Операторы ***DELETE*** и ***FIND*** можно реализовать, применив общие методы поиска заданного элемента в списках. Для оператора ***DELETE*** найденный элемент (точнее, ячейка, в которой он находится) удаляется.

Реализация множеств посредством связанных списков

Реализовать оператор вставки нового элемента в список также несложно, но он должен стоять не в произвольной позиции в списке, а в "правильной" позиции, учитывающей взаимный порядок элементов.

В листинге 5.4 представлен код процедуры ***INSERT*** (Вставка), которая в качестве параметров имеет вставляемый элемент и указатель на ячейку заголовка списка, куда вставляется элемент.

Листинг 5.4. Процедура вставки элемента

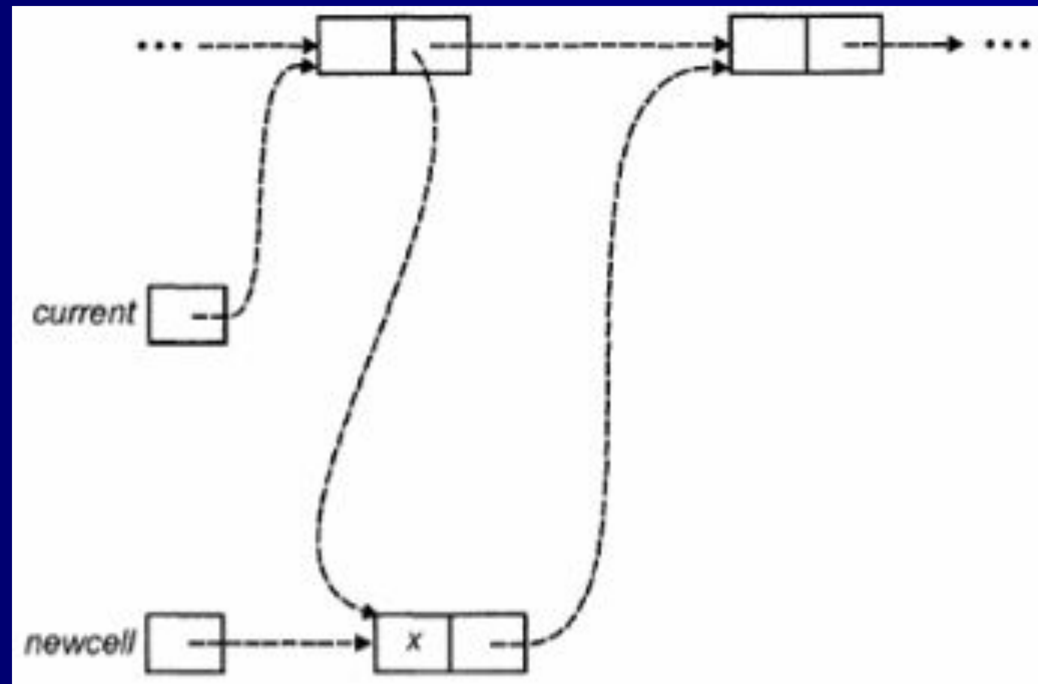
```
procedure   INSERT ( x: elementtype; p: ^celltype );
var   current, newcell: ^celltype;
begin
    current:= p;
    while   current^.next <> nil   do  begin
        if  current^.next^.element=x  then
            goto  fin;          { элемент x уже есть в списке }
        if  current^.next^.element > x  then
            goto  add;          { далее останов процедуры }
        current:= current^.next
    end;
```

Листинг 5.4.

```
    add:    {здесь current - ячейка, после которой надо вставить x}
    new(newcell);
    newcell^.element:= x;
    newcell^.next:= current^.next;
    current^.next:= newcell ;
fin:
end;    { INSERT }
```

Реализация множеств посредством связанных списков

На рисунке показаны ключевые ячейки и указатели до и после вставки элемента (старые указатели обозначены сплошными линиями, а новые - пунктирными).



Словари

Применение множеств при разработке алгоритмов не всегда требует таких мощных операторов, как операторы объединения и пересечения. Часто достаточно только хранить в множестве "текущие" объекты с периодической вставкой или удалением некоторых из них. Время от времени также возникает необходимость узнать, присутствует ли конкретный элемент в данном множестве.

Абстрактный тип множеств с операторами ***INSERT***, ***DELETE*** и ***MEMBER*** называется ***DICTIONARY*** (Словарь).

Мы также включим оператор ***MAKENULL*** в набор операторов словаря - он потребуется при реализации АТД для инициализации структур данных.

Словари

Пример 5.2. Общество защиты тунцов (ОЗТ) имеет базу данных с записями результатов самого последнего голосования законодателей по законопроектам об охране тунцов. База данных состоит из двух списков (множеств) имен законодателей, которые названы ***goodguys*** (хорошие парни) и ***badguys*** (плохие парни). ОЗТ прощает законодателям их прошлые "ошибки", но имеет тенденцию забывать своих "друзей", которые ранее голосовали "правильно". Например, после голосования по законопроекту об ограничении вылова тунца в озере Эри все законодатели, проголосовавшие за этот законопроект, заносятся в список ***goodguys*** и удаляются из списка ***badguys***, тогда как над оппонентами этого законопроекта совершается обратная процедура. Законодатели, не принимавшие участие в голосовании, остаются в тех списках, в которых они были ранее.

Словари

Для управления описываемой базы данных при вводе имен законодателей будем применять односимвольные команды, за символом команды будет следовать 10 символов с именем законодателя. Каждая команда располагается в отдельной строке.

Используем следующие односимвольные команды:

- **F** (законодатель голосовал "правильно").
- **U** (законодатель голосовал "неправильно").
- **?** (надо определить статус законодателя).

Будем использовать символ '**E**' для обозначения окончания процесса ввода списка законодателей.

В листинге 5.5 показан эскиз программы ***tuna*** (тунец), написанный в терминах пока не определенного АТД ***DICTIONARY*** (Словарь), который в данном случае можно представить как множество символьных строк длиной 10.

Листинг 5.5. Программа управления базой данных ОЗТ

```
program   tuna;
{ База данных законодателей (legislator) }
type   nametype = array[1..10]  of  char;
var
      command: char;
      legislator: nametype;
      goodguys, badguys: DICTIONARY;

procedure  favor (friend: nametype);
{ заносит имя friend  в список goodguys и вычеркивает из списка badguys}
begin
      INSERT(friend, goodguys);
      DELETE(friend, badguys)
end;     { favor }
```

Листинг 5.5.

```
procedure unfavor ( foe: nametype );
{заносит имя foe (враг) в список badguys и вычеркивает из списка goodguys}
begin
    INSERT(foe, badguys);
    DELETE(foe, goodguys)
end;    { unfavor }
```

Листинг 5.5.

```
procedure   report ( subject: nametype );
{печать имени subject с соответствующей характеристикой}
begin
    if  MEMBER(subject, goodguys)  then
        Writeln (subject, ' - это друг')
    else  if  MEMBER(subject, badguys)  then
            writeln{subject, ' - это враг')
        else
            writeln('Нет данных о ', subject,)
end;    { report }
```

Листинг 5.5.

```
begin        { основная программа }
    MAKENULL(goodguys) ;        MAKENULL(badguys);
    read(command);
    while  command <> ' E'  do  begin
        readln(legislator);
        if  command = 'F'  then  favor(legislator)
        else  if  command = 'U'  then  unfavor(legislator)
            else  if  command = '?'  then  report(legislator)
                else  report('Неизвестная команда')
        read(command)
    end
end;    { tuna }
```

Реализации словарей

- Словари можно представить посредством сортированных или несортированных связанных списков.
- Другая возможная реализация словарей использует двоичные векторы, предполагая, что элементы данного множества являются целыми числами **1**, ..., ***N*** для некоторого ***N*** или элементы множества можно сопоставить с таким множеством целых чисел.
- Третья возможная реализация словарей использует массив фиксированной длины с указателем на последнюю заполненную ячейку этого массива. Эта реализация выполнима, если мы точно знаем, что размер множества не превысит заданную длину массива.

Реализации словарей

Эта реализация проще реализации посредством связанных списков, но имеет следующие недостатки:

- ♦ множества могут расти только до определенной фиксированной величины;
- ♦ медленно выполняются операции удаления элементов из множества (так как требуется перемещение оставшихся элементов массива)
- ♦ и невозможность эффективно организовать пространство массивов (особенно если множества имеют различные размеры).

Так как мы рассматриваем реализации именно словарей (и вследствие последнего приведенного недостатка), то не будем затрагивать возможности выполнения в реализации посредством массивов операций объединения и пересечения множеств.

В листинге 5.6 приведены объявления и процедуры, являющиеся необходимым дополнением программы листинга 5.5.

Листинг 5.6. Объявления типов и процедуры реализации словаря посредством массива

```
const   maxsize ={некое число, максимальный размер массива}
type   DICTIONARY = record
                last: integer;
                data: array[1..maxsize]  of  nametype
            end;

procedure  MAKENULL ( var A: DICTIONARY );
begin
      A.last:= 0
end;     { MAKENULL }
```

Листинг 5.6.

```
function  MEMBER ( x: nametype; var A: DICTIONARY ): boolean;
var   i: integer;
begin
     for   i:= 1   to  A.last   do
          if  A.data[i] = x   then  MEMBER:=true;
     MEMBER:=false              { элемент х не найден }
end;     { MEMBER }
```

Листинг 5.6.

```
procedure  INSERT ( x: nametype; var A: DICTIONARY );
begin
     if   not  MEMBER(x, A)    then
          if   A.last < maxsize  then  begin
               A.last:= A.last + 1;
               A. data [A. last] := x
          end
          else  error('База данных заполнена')
end;      { INSERT }
```

Листинг 5.6.

```
procedure   DELETE ( x: nametype; var A: DICTIONARY );
var   i: integer;
begin
    if   A.last > 0  then  begin
        i:= 1;
        while (A.data[i] <> x) and (i < A.last)   do
                i:= i + 1;
        if   A.data[i] = x   then   begin
            A.data[i] = A.data[A.last];
            { перемещение последнего элемента на место элемента x;
            если i=A.last, то удаление x происходит на следующем шаге }
            A.last:= A.last - 1
        end
    end
end;    { DELETE }
```

Структуры данных, основанные на хеш-таблицах

В реализации словарей с помощью массивов выполнение операторов ***INSERT***, ***DELETE*** и ***MEMBER*** требует в среднем ***O(N)*** выполнений элементарных инструкций для словаря из ***N*** элементов.

Подобной скоростью выполнения операторов обладает и реализация с помощью списков.

При реализации словарей посредством двоичных векторов эти три оператора выполняются за фиксированное время независимо от размера множеств, но в этом случае мы ограничены множествами целых чисел из некоторого небольшого конечного интервала.

Структуры данных, основанные на хеш-таблицах

Существует еще один полезный и широко используемый метод реализации словарей, который называется ***хешированием***.

Этот метод требует фиксированного времени (в среднем) на выполнение операторов и снимает ограничение, что множества должны быть подмножествами некоторого конечного универсального множества.

В самом худшем случае этот метод для выполнения операторов требует времени, пропорционального размеру множества, так же, как и в случаях реализаций посредством массивов и списков.

Но при тщательной разработке алгоритмов мы можем сделать так, что вероятность выполнения операторов за время, большее фиксированного, будет как угодно малой.

Структуры данных, основанные на хеш-таблицах

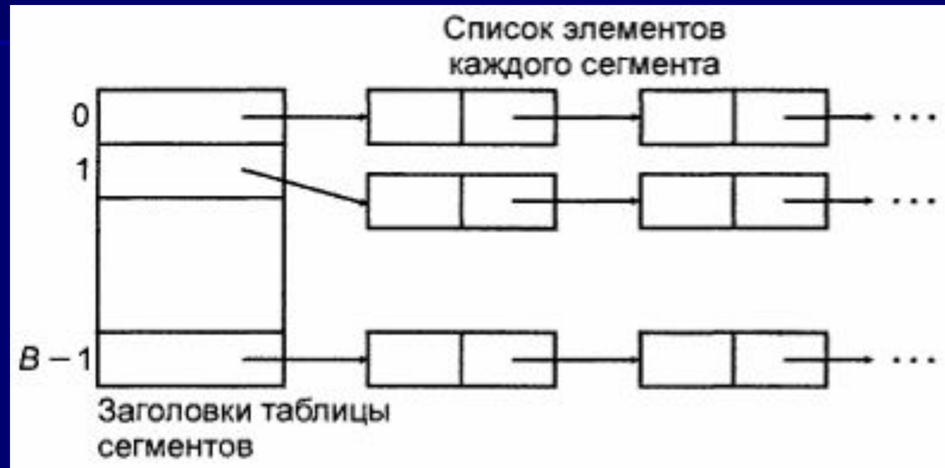
Мы рассмотрим две различные формы хеширования.

Одна из них называется ***открытым***, или ***внешним***, ***хешированием*** (частный случай *расширенного хеширования*) и позволяет хранить множества в потенциально бесконечном пространстве, снимая тем самым ограничения на размер множеств.

Другая называется ***закрытым***, или ***внутренним*** или ***прямым***, ***хешированием*** и использует ограниченное пространство для хранения данных, ограничивая таким образом размер множеств.

Открытое хеширование

Организация структуры данных при открытом хешировании:



Основная идея заключается в том, что потенциальное множество (возможно, бесконечное) разбивается на конечное число классов.

Для ***B*** классов, пронумерованных от **0** до ***B*-1**, строится хеш-функция ***h*** такая, что для любого элемента ***x*** исходного множества функция ***h(x)*** принимает целочисленное значение из интервала **0**, **...**, ***B*-1**, которое соответствует классу, которому принадлежит элемент ***x***.

Элемент ***x*** часто называют ***ключом***, ***h(x)*** - ***хеш-значением x***, а "классы" - ***сегментами***. Мы будем говорить, что элемент ***x*** принадлежит сегменту ***h(x)***.

Открытое хеширование

Массив, называемый ***таблицей сегментов*** и проиндексированный номерами сегментов **0**, **1**, ..., ***B*–1**, содержит заголовки для ***B*** списков. Элемент ***x i***-го списка - это элемент исходного множества, для которого ***h(x)=i***.

Если сегменты примерно одинаковы по размеру, то в этом случае списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из ***N*** элементов, тогда средняя длина списков будет ***N/B*** элементов.

Если можно оценить величину ***N*** и выбрать ***B*** как можно ближе к этой величине, то в каждом списке будет 1-2 элемента. Тогда время выполнения операторов словарей будет малой постоянной величиной, зависящей от ***N*** (или, что эквивалентно, от ***B***).

Открытое хеширование

Массив, называемый ***таблицей сегментов*** и проиндексированный номерами сегментов **0**, **1**, ..., ***B*-1**, содержит заголовки для ***B*** списков. Элемент ***x*** ***i***-го списка - это элемент исходного множества, для которого ***h(x)=i***.

Если сегменты примерно одинаковы по размеру, то в этом случае списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из ***N*** элементов, тогда средняя длина списков будет ***N/B*** элементов.

Если можно оценить величину ***N*** и выбрать ***B*** как можно ближе к этой величине, то в каждом списке будет 1-2 элемента. Тогда время выполнения операторов словарей будет малой постоянной величиной, зависящей от ***N*** (или, что эквивалентно, от ***B***).

Не всегда ясно, как выбрать хеш-функцию ***h*** так, чтобы она примерно поровну распределяла элементы исходного множества по всем сегментам.

Ниже показан простой способ построения функции ***h***, причем ***h(x)*** будет "случайным" значением, почти независящим от ***x***.

Открытое хеширование

Здесь же мы введем хеш-функцию (которая будет "хорошей", но не "отличной"), определенную на символьных строках.

Идея построения этой функции заключается в том, чтобы представить символы в виде целых чисел, используя для этого машинные коды символов. В языке Pascal есть встроенная функция ***ord(c)***, которая возвращает целочисленный код символа ***c***.

Таким образом, если ***x*** - это ключ, тип данных ключей определен как array[1..10] of char (выше этот тип данных назван ***nametype***), тогда можно использовать хеш-функцию, код которой представлен в листинге 5.7.

В этой функции суммируются все целочисленные коды символов, результат суммирования делится на ***B*** и берется остаток от деления, который будет целым числом из интервала от **0** до ***B*-1**.

Листинг 5.7. Простая хеш-функция

```
Function  h ( x: nametype ): 0..B-1;
var   i, sum: integer;
begin
      sum:= 0;
      for  i:= 1  to  10  do
            sum:= sum + ord( x[i] );
      h:= sum  mod  B
end;                  { h }
```

Открытое хеширование

В листинге 5.8 показаны объявления структур данных для открытой хеш-таблицы и процедуры, реализующие операторы, выполняемые над словарем.

Тип данных элементов словаря - ***nametype*** (здесь - символьный массив), поэтому данные объявления можно непосредственно использовать в листинге 5.5.

Отметим, что в листинге 5.8 заголовки списков сегментов сделаны указателями на ячейки, а не "настоящими" ячейками.

Это сделано для экономии пространства, занимаемого данными: если заголовки таблицы сегментов будут ячейками массива, а не указателями, то под этот массив необходимо столько же места, сколько и под списки элементов.

Но за такую экономию пространства надо платить: код процедуры ***DELETE*** должен уметь отличать первую ячейку от остальных.

Листинг 5.8. Реализация словарей посредством открытой хеш-таблицы

```
const  B = { подходящая константа };
type  celltype = record
            elemet: nametype;
            next: ^celltype
      end;
      DICTIONARY = array[0..B-1]  of  ^celltype;

procedure  MAKENULL ( var A: DICTIONARY );
var   i: integer;
begin
      for  i:= 0  to  B - 1  do
      A[i]:= nil
end;          { MAKENULL }
```

Листинг 5.8.

```
function  MEMBER ( x: nametype; var A: DICTIONARY ): boolean;
var   current: ^celltype;
begin
      current:= A[h(x)];
      { начальное значение current равно заголовку сегмента,
      которому принадлежит элемент x }
      while  current <> nil  do
            if  current^.element = x   then  MEMBER:=true
            else  current:= current^.next;
      MEMBER:=false              { элемент x не найден }
end;            { MEMBER }
```

Листинг 5.8.

```
procedure  INSERT ( x: nametype; var A: DICTIONARY );
var   bucket: integer;    { для номера сегмента }
      oldheader: ^celltype;
begin
      if  not MEMBER(x, A)  then  begin
            bucket:= h(x);
            oldheader:= A[bucket];
            new( A[bucket] );
            A[bucket] ^.element: = x;
            A[bucket] ^.next:= oldheader
      end
end;            { INSERT }
```

Листинг 5.8.

```
procedure  DELETE ( x: nametype; var A: DICTIONARY );
var  bucket: integer;     current: ^celltype;  f: boolean;
begin
    bucket:= h(x);   f:= true;
    if  A[bucket] <> nil  then  begin
        if A[bucket] ^.element = x  then      { x в первой ячейке }
            A[bucket]:= A[bucket] ^.next   { удаление x из списка }
        else   begin       { x находится не в первой ячейке }
                current:= A[bucket];
                { current указывает на предыдущую ячейку }
                while  (current^.next <> nil ) and  f  do
                    if  current^.next^.element = x  then  begin
                     current^.next: = current^.next^.next;
                     { удаление x из списка }
                     f:= false { останов }  end
                    else   { x пока не найден }  current:= current^.next
        end
    end
end;          { DELETE }
```

Закрытое хеширование

При закрытом хешировании в таблице сегментов хранятся непосредственно элементы словаря, а не заголовки списков. Поэтому в каждом сегменте может храниться только один элемент словаря.

При закрытом хешировании применяется методика ***повторного хеширования***.

Если мы попытаемся поместить элемент ***x*** в сегмент с номером ***h(x)***, который уже занят другим элементом (такая ситуация называется ***коллизией***), то в соответствии с методикой повторного хеширования выбирается последовательность других номеров сегментов $h_1(x)$, $h_2(x)$, ..., куда можно поместить элемент ***x***.

Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное.

Если свободных сегментов нет, то, следовательно, таблица заполнена и элемент ***x*** вставить нельзя.

Закрытое хеширование. Пример

Предположим, что *B*=8 и ключи ***a***, ***b***, ***c*** и ***d*** имеют хеш-значения ***h(a)***=3, ***h(b)***=0, ***h(c)***=4 и ***h(d)***=3.

Применим простую методику, которая называется ***линейным хешированием***.

При линейном хешировании

$$h_i(x)=(h(x)+i) \bmod B.$$

Например, если мы хотим вставить элемент ***d***, а сегмент 3 уже занят, то можно проверить на занятость сегменты 4, 5, 6, 7, 0, 1 и 2 (именно в таком порядке).

Закрытое хеширование. Пример

Мы предполагаем, что вначале вся хеш-таблица пуста, т. е. в каждый сегмент помещено специальное значение ***empty*** (пустой), которое не совпадает ни с одним элементом словаря.

Теперь последовательно вставим элементы ***a***, ***b***, ***c*** и ***d*** в пустую таблицу:

- элемент ***a*** попадет в сегмент 3,
- элемент ***b*** - в сегмент 0,
- а элемент ***c*** - в сегмент 4.

Для элемента ***d*** ***h(d)***=3, но сегмент 3 уже занят.

Применяем функцию h_1: $h_1(d)$=4, но сегмент 4 также занят.

Далее применяем функцию h_2: $h_2(d)$=5, сегмент 5 свободен, помещаем туда элемент ***d***.

0	*b*
1	
2	
3	*a*
4	*c*
5	*d*
6	
7	

Закрытое хеширование

При поиске элемента ***x*** (например, при выполнении оператора ***MEMBER***) необходимо просмотреть все местоположения $h(x)$, $h_1(x)$, $h_2(x)$, ..., пока не будет найден ***x*** или пока не встретится пустой сегмент.

Чтобы объяснить, почему можно остановить поиск при достижении пустого сегмента, предположим, что в словаре не допускается удаление элементов. И пусть, для определенности, $h_3(x)$ - первый пустой сегмент. В такой ситуации невозможно нахождение элемента ***x*** в сегментах $h_4(x)$, $h_5(x)$ и далее, так как при вставке элемент ***x*** вставляется в первый пустой сегмент, следовательно, он находится где-то до сегмента $h_3(x)$.

Но если в словаре допускается удаление элементов, то при достижении пустого сегмента мы, не найдя элемента ***x***, не можем быть уверенными в том, что его вообще нет в словаре, так как сегмент может стать пустым уже после вставки элемента ***x***.

Закрытое хеширование

Поэтому, для увеличения эффективности данной реализации необходимо в сегмент, который освободился после операции удаления элемента, поместить специальную константу, которую назовем ***deleted*** (удаленный).

Важно различать константы ***deleted*** и ***empty*** - последняя находится в сегментах, которые никогда не содержали элементов.

При таком подходе выполнение оператора ***MEMBER*** не требует просмотра всей хеш-таблицы. Кроме того, при вставке элементов сегменты, помеченные константой ***deleted***, можно трактовать как свободные и использовать их повторно.

Но если невозможно непосредственно сразу после удаления элементов пометить освободившееся сегменты, то следует предпочесть закрытому хешированию схему открытого хеширования.

Закрытое хеширование. Пример

Предположим, что надо определить, есть ли элемент ***e*** в множестве на рис.. Если ***h(e)***=4, то надо проверить еще сегменты 4, 5 и затем сегмент 6.

Сегмент 6 пустой, в предыдущих просмотренных сегментах элемент ***e*** не найден, следовательно, этого элемента в данном множестве нет.

Допустим, мы удалили элемент ***c*** и поместили константу ***deleted*** в сегмент 4.

В этой ситуации для поиска элемента ***d*** мы начнем с просмотра сегмент ***h(d)***=3, затем просмотрим сегменты 4 и 5 (где и найдем элемент ***d***), при этом мы не останавливаемся на сегменте 4 - хотя он и пустой, но не помечен как ***empty***.

0	b
1	
2	
3	a
4	c
5	d
6	
7	

Закрытое хеширование

В листинге 5.9 представлены объявления типов данных и процедуры операторов для АТД ***DICTIONARY*** с элементами типа ***nametype*** и реализацией, использующей закрытую хеш-таблицу.

Здесь используется хеш-функция ***h*** из листинга 5.7, для разрешения коллизий применяется методика линейного хеширования.

Константа ***empty*** определена как строка из десяти пробелов, а константа ***deleted*** - как строка из десяти символов "*", в предположении, что эти строки не совпадают ни с одним элементом словаря.

Процедура ***INSERT(x, A)*** использует функцию ***locate*** (местонахождение) для определения, присутствует ли элемент ***x*** в словаре ***A*** или нет, а также специальную функцию ***locatel*** для определения местонахождения элемента ***x***. Последнюю функцию также можно использовать для поиска констант ***deleted*** и ***empty***.

Листинг 5.9. Реализация словаря посредством закрытого хеширования

```
const   empty='          ';          { 10 пробелов }
        deleted='**********' ;       { 10 символов * }
type   DICTIONARY = array[0..B-1]  of  nametype;

procerude   MAKENULL ( var A: DICTIONARY );
var   i: integer;
begin
     for  i:= 0  to  B - 1   do
          A[i]:= empty
end;     { MAKENULL }
```

Листинг 5.9.

```
function  locate ( x: nametype; A: DICTIONARY ): integer;
{ Функция просматривает A начиная от сегмента h(x) до тех пор, пока не
      будет найден элемент x или не встретится пустой сегмент или пока
      не будет достигнут конец таблицы (в последних случаях
      принимается, что таблица не содержит элемент x). Функция
      возвращает позицию, в которой остановился поиск. }
var   initial, i: integer;
begin
    initial:= h(x) ;    i:= 0;
    while (i < B) and (A[initial + i) mod B] <> x) and
              (A[(initial + i) mod B] <> empty)  do
           i:= i + 1;
    locate:=(initial + i) mod B
end;     { locate }
```

Листинг 5.9.

```
function  locatel ( x: nametype; A: DICTIONARY ): integer;
{ То же самое, что и функция locate, но останавливается и при
        достижении значения deleted }
end;     { LOCATEL }

function  MEMBER ( x: nametype; var A: DICTIONARY ):
                              boolean;
begin
      if  A[locate(x)] = x  then
            MEMBER:=true
      else
            MEMBER:=false
end;     { MEMBER }
```

Листинг 5.9.

```
procedure  INSERT ( x: nametype; var A: DICTIONARY ) ;
var   bucket: integer;
begin
     if  A[locate(x)] <> x  then  begin;      { x нет в A }
          bucket: = locatel(x);
          if  (A[bucket] = empty) or (A[bucket) = deleted)  then
               A[bucket]:= x
          else  error('Операция INSERT невозможна: таблица
               полна‘)
     end
end;     { INSERT }
```

Листинг 5.9.

```
procedure  DELETE ( x: nametype; var A: DICTIONARY );
var   bucket: integer;
begin
      bucket:= locate(x);
      if   A[locate(x)] = x   then
            A[bucket]:= deleted
end;     { DELETE }
```

Оценка эффективности хеш-функций

Оценим среднее время выполнения операторов словарей для случая открытого хеширования.

Если есть ***B*** сегментов и **N** элементов, хранящихся в хеш-таблице, то каждый сегмент в среднем будет иметь ***N/B*** элементов и мы ожидаем, что операторы ***INSERT***, ***DELETE*** и ***MEMBER*** будут выполняться в среднем за время ***O(*1*+N/B)***. Здесь константа **1** соответствует поиску сегмента, а ***N/B*** - поиску элемента в сегменте.

Если ***B*** примерно равно ***N***, то время выполнения операторов становится константой, независящей от ***N***.

Оценка эффективности хеш-функций

Предположим, что есть программа, написанная на языке программирования, подобном Pascal, и мы хотим все имеющиеся в этой программе идентификаторы занести в хеш-таблицу.

После обнаружения объявления нового идентификатора он вставляется в хеш-таблицу после проверки, что его еще нет в хеш-таблице. На этапе проверки естественно предположить, что идентификатор с равной вероятностью может быть в любом сегменте.

Таким образом, на процесс заполнения хеш-таблицы с N элементами потребуется время порядка $O(N(1 + N/B))$.

Если положить, что B равно N, то получим время $O(N)$.

Оценка эффективности хеш-функций

На следующем этапе анализа программы просматриваются идентификаторы в теле программы.

После нахождения идентификатора в теле программы, чтобы получить информацию о нем, его же необходимо найти в хеш-таблице. Какое время потребуется для нахождения идентификатора в хеш-таблице?

Если время поиска для всех элементов примерно одинаково, то оно соответствует среднему времени вставки элемента в хеш-таблицу.

Чтобы увидеть это, достаточно заметить, что время поиска любого элемента равно времени вставки элемента в конец списка соответствующего сегмента.

Таким образом, время поиска элемента в хеш-таблице составляет $O(1+N/B)$.

Оценка эффективности хеш-функций

В приведенном выше анализе предполагалось, что хеш-функция распределяет элементы по сегментам равномерно. Но существуют ли такие функции?

Рассмотрим функцию, код которой приведен в листинге 5.7, как типичную хеш-функцию:

- эта функция преобразует символы в целочисленный код,
- суммирует коды всех символов
- и в качестве результата берет остаток от деления этой суммы на число ***B***.

Следующий пример оценивает работу этой функции.

Оценка эффективности хеш-функций. Пример

Предположим, что функция из листинга 5.7 применяется для занесения в таблицу со 100 сегментами 100 символьных строк А0, А1, ... А99.

Обратите внимание, что здесь "А" - буква английского алфавита и что приведенное распределение элементов по сегментам справедливо только для записанных выше символьных строк. Для другой буквы (или других символьных строк) получим другое распределение элементов по сегментам, но также, скорее всего, далекое от равномерного.

Принимая во внимание, что ***ord*(0)**, ***ord*(1)**, ..., ***ord*(9)** образуют арифметическую прогрессию (это справедливо для всех таблиц кодировок, где цифры 0, ..., 9 стоят подряд, например для кодировки ASCII), легко проверить, что эти элементы займут не более 29 сегментов из ста (отметим, что строки А2 и А20 не обязательно должны находиться в одном сегменте, но А23 и А41, например, будут располагаться в одном сегменте).

Наибольший сегмент (сегмент с номером 2) будет содержать элементы А18, А27, А36, ..., А90, т.е. девять элементов из ста.

Оценка эффективности хеш-функций. Пример

Используя тот факт, что для вставки ***i***-го элемента требуется ***i*+1** шагов, легко подсчитать, что в данном случае среднее число шагов, необходимое для вставки всех 100 элементов, равно 395.

Для сравнения заметим, что оценка ***N*(1 + *N/B*)** предполагает только 200 шагов.

Оценка эффективности хеш-функций

В приведенном примере хеш-функция распределяет элементы исходного множества по множеству сегментов не равномерно. Но возможны "более равномерные" хеш-функции.

Для построения такой функции можно воспользоваться хорошо известным методом возведения числа в квадрат и извлечения из полученного квадрата нескольких средних цифр.

Например, если есть число ***n***, состоящее из 5 цифр, то после возведения его в квадрат получим число, состоящее из 9 или 10 цифр. "Средние цифры" - это цифры, стоящие, допустим, на позициях от 4 до 7 (отсчитывая справа). Их значения, естественно, зависят от числа ***n***.

Если ***B***=100, то для формирования номера сегмента достаточно взять две средние цифры, стоящие, например, на позициях 5 и 6 в квадрате числа.

Оценка эффективности хеш-функций

Этот метод можно обобщить на случай, когда ***B*** не является степенью числа 10.

Предположим, что элементы исходного множества являются целыми числами из интервала 0, 1, ..., ***K***.

Введем такое целое число ***C***, что BC^2 примерно равно K^2.

Тогда функция

$$h(n) = [n^2/C] \bmod B,$$

где ***[x]*** обозначает целую часть числа ***x***, эффективно извлекает из середины числа n^2 цифры, составляющие число, не превышающее ***B***.

Оценка эффективности хеш-функций. Пример

Если K=1000 и B=8, то можно выбрать C=354.

Тогда

$$h(456) = [207936/354]\ mod\ 8 = 587\ mod\ 8 = 3.$$

Оценка эффективности хеш-функций

Для применения к символьной строке описанной хеш-функции надо сначала в строке сгруппировать символы справа налево в блоки с фиксированным количеством символов, например по 4 символа, добавляя при необходимости слева пробелы.

Каждый блок трактуется как простое целое число, из которого путем конкатенации (сцепления) формируется двоичный код символов, составляющих блок.

Например, основная таблица ASCII кодировки символов использует 7-битовый код (для представления латиницы), поэтому символы можно рассматривать как "цифры" по основанию 2^7 или 128.

Таким образом, символьную строку ***abed*** можно считать целым числом

$$128^3 a + 128^2 b + 128^1 c + d.$$

После преобразования всех блоков в целые числа они суммируются, а затем выполняется вышеописанный процесс возведения в квадрат и извлечения средних цифр.

Анализ закрытого хеширования

В этом случае скорость выполнения вставки и других операций зависит не только от равномерности распределения элементов по сегментам хеш-функцией, но и от выбранной методики повторного хеширования для разрешения коллизий, связанных с попытками вставки элементов в уже заполненные сегменты.

Например, методика линейного хеширования для разрешения коллизий - не самый лучший выбор. Не приводя полного решения этой проблемы, мы ограничимся следующим анализом.

Анализ закрытого хеширования

Как только несколько последовательных сегментов будут заполнены (образуя группу), любой новый элемент при попытке вставки в эти сегменты будет вставлен в конец этой группы, увеличивая тем самым длину группы последовательно заполненных сегментов.

Другими словами, для поиска пустого сегмента в случае непрерывного расположения заполненных сегментов необходимо просмотреть больше сегментов, чем при случайном распределении заполненных сегментов.

Отсюда также следует очевидный вывод, что при непрерывном расположении заполненных сегментов увеличивается время выполнения вставки нового элемента и других операторов.

Анализ закрытого хеширования

Определим, сколько необходимо сделать проб (проверок) на заполненность сегментов при вставке нового элемента, предполагая, что в хеш-таблице, состоящей из ***B*** сегментов, уже находится ***N*** элементов и все комбинации расположения ***N*** элементов в ***B*** сегментах равновероятны.

Далее получим формулы, оценивающие "стоимость" (количество проверок на заполненность сегментов) вставки нового элемента, если сегменты выбираются случайным образом.

Наконец, мы рассмотрим некоторые методики повторного хеширования, дающие "случайное" (равномерное) распределение элементов по сегментам.

Анализ закрытого хеширования

Вероятность коллизии равна ***N/B***.

Предполагая осуществление коллизии, на первом этапе повторного хеширования "работаем" с ***B*-1** сегментом, где находится ***N*-1** элемент.

Тогда вероятность возникновения двух подряд коллизий равна

$$N(N-1)/(B(B-1)).$$

Аналогично, вероятность по крайней мере ***i*** коллизий равна

$$N(N-1)\ldots(N-i+1)\,/\,(B(B-1)\ldots(B-i+1)).$$

Если значения ***B*** и ***N*** большие, то эта вероятность примерно равна

$$(N/B)^{i}.$$

Таким образом, обобщив рассуждения, для случая полного заполнения таблицы требуется в среднем ***ln B*** проб на сегмент, или всего ***BlnB*** проб. Но для заполнения таблицы на 90% требуется примерно **2.56*B*** проб.

Анализ закрытого хеширования

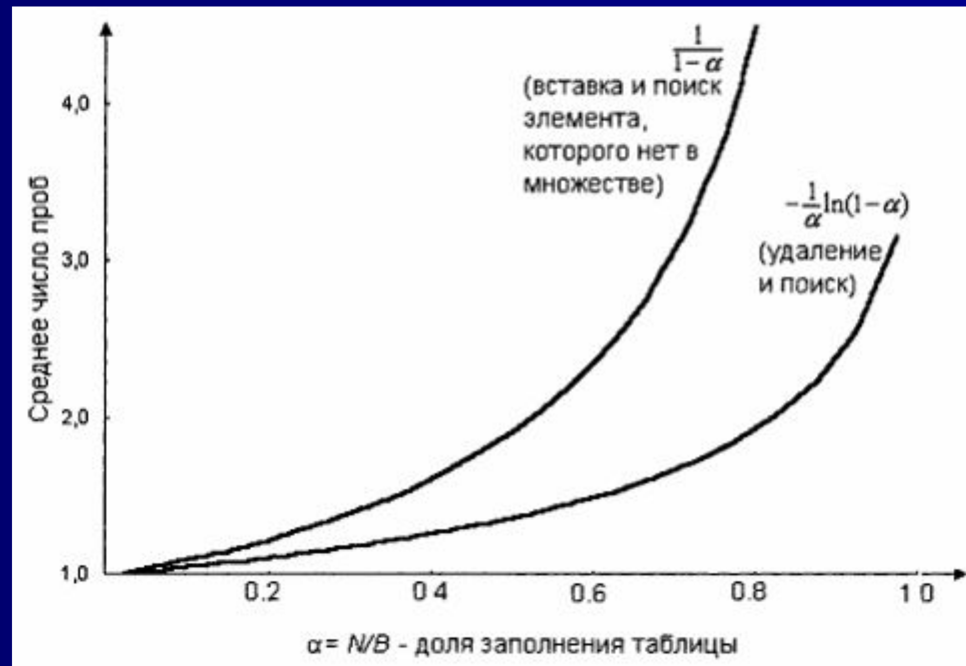
При проверке на принадлежность исходному множеству элемента, которого заведомо нет в этом множестве, требуется в среднем такое же число проб, как и при вставке нового элемента при данном заполнении таблицы.

Но проверка на принадлежность элемента, который принадлежит множеству, требует в среднем столько же проб, сколько необходимо для вставки всех элементов, сделанных до настоящего времени.

Удаление требует в среднем примерно столько же проб, сколько и проверка элемента на принадлежность множеству. Но в отличие от схемы открытого хеширования, удаление элементов из закрытой хеш-таблицы не ускоряет процесс вставки нового элемента или проверки принадлежности элемента множеству.

Анализ закрытого хеширования

Необходимо особо подчеркнуть, что средние числа проб, необходимые для выполнения операторов словарей, являются константами, зависящими от доли заполнения хеш-таблицы.



Графики средних чисел проб, необходимые для выполнения операторов, в зависимости от доли заполнения хеш-таблицы.

„Случайные" методики разрешения коллизий

Методика линейного повторного хеширования приводит к группированию заполненных сегментов в большие непрерывные блоки.

Можно предложить хеш-функции с более "случайным" поведением, например, ввести целочисленную константу $c>1$ и определить

$$h_i(x) = (h(x) + c^i) \bmod B.$$

В этом случае для $B=8$, $c=3$ и $h(x)=4$ получим "пробные" сегменты в следующем порядке: 4, 7, 2, 5, 0, 3, 6 и 1.

Конечно, если B и c имеют общие делители (отличные от единицы), то эта методика не позволит получить все номера сегментов, например при $B=8$ и $c=2$.

„Случайные" методики разрешения коллизий

Но даже если ***B*** и ***c*** взаимно простые числа, то все равно существует проблема "группирования", как и при линейном хешировании, хотя здесь разделяются блоки заполненных сегментов, соответствующие различным константам ***c***.

Этот феномен увеличивает время выполнения операторов словарей (как и при линейном хешировании), поскольку попытка вставить новый элемент в заполненный сегмент приводит к просмотру цепочки заполненных сегментов, различных для различных ***c***, и длина этих цепочек при каждой вставке увеличивается на единицу.

„Случайные" методики разрешения коллизий

Фактически любая методика повторного хеширования, где очередная проба зависит только от предыдущей (например, как зависимость от числа предыдущих "неудачных" проб, от исходного значения ***h(x)*** или от самого элемента ***x***), обнаруживает группирующие свойства линейного хеширования.

Возможна простейшая методика, для которой проблема "группирования" не существует:

для этого достаточно положить

$$h_i(x) = (h(x) + d_i) \bmod B.$$

где $d_1, d_2, \ldots, d_{B-1}$ - "случайные" перестановки чисел 1, 2, ..., ***B***-1.

Такой набор чисел $d_1, d_2, \ldots, d_{B-1}$ должен использоваться при реализации всех операторов, выполняемых над словарями, а "случайное" перемешивание целых чисел должно быть сделано (выбрано) еще при разработке алгоритма хеширования.

„Случайные" методики разрешения коллизий

Существуют сравнительно простые методы получения последовательности "случайных" чисел путем "перемешивания" целых чисел из заданного интервала.

При наличии такого генератора случайных чисел можно воспроизводить требуемую последовательность d_1, d_2, ..., d_{B-1} при каждом выполнении операторов, работающих с хеш-таблицей.

„Случайные" методики разрешения коллизий

Одним из эффективных методов "перемешивания" целых чисел является метод "***последовательных сдвигов регистра***".

Пусть ***B*** является степенью числа 2 и ***k*** - константа из интервала от 1 до ***B***-1. Не для каждого значения ***k*** можно получить все "перемешанные" значения от 1 до ***B***-1, иногда некоторые сгенерированные числа повторяются. Поэтому для каждого значения ***B*** необходимо подобрать свое значение ***k***, которое будет "работать".

Начнем с некоторого числа d_1, взятого из интервала от 1 до ***B***-1.

Далее генерируется последовательность чисел d_i путем удвоения предыдущего значения до тех пор, пока последнее значение не превысит ***B***.

Тогда для получения следующего числа d_i из последнего значения отнимается число ***B*** и результат *суммируется побитово по модулю 2* с константой ***k***.

„Случайные" методики разрешения коллизий

Сумма по модулю 2 чисел ***x*** и ***y*** (***x*** $\oplus$ ***y***) определяется следующим образом:

- числа ***x*** и ***y*** записываются в двоичном виде с возможным приписыванием ведущих нулей так, чтобы числа имели одинаковую длину;
- результат формируется по правилу логической операции "исключающего ИЛИ", т.е. 1 в какой-либо позиции результата будет стоять тогда и только тогда, когда только в одной аналогичной позиции слагаемых стоит 1, но не в обеих.

Пример. 25 $\oplus$ 13 вычисляется следующим образом:

$$25 = 11001$$

$$13 = 01101$$

$$25 \oplus 13 = 10100$$

Такое "суммирование" возможно с помощью обычного двоичного суммирования, если игнорировать перенос разрядов.

Реструктуризация хеш-таблиц

При использовании открытых хеш-таблиц среднее время выполнения операторов возрастает с ростом параметра ***N/B*** и особенно быстро растет при превышении числа элементов над числом сегментов.

Подобным образом среднее время выполнения операторов также возрастает с увеличением параметра ***N/B*** и для закрытых хеш-таблиц (но превышения ***N*** над ***B*** здесь невозможно).

Реструктуризация хеш-таблиц

Чтобы сохранить постоянное время выполнения операторов, которое теоретически возможно при использовании хеш-таблиц, предлагается при достижении ***N*** достаточно больших значений, например,

- при $N > 0.9B$ для закрытых хеш-таблиц
- и $N > 2B$ - для открытых хеш-таблиц,

просто создавать новую хеш-таблицу с удвоенным числом сегментов.

Перезапись текущих элементов множества в новую хеш-таблицу в среднем займет меньше времени, чем их ранее выполненная вставка в старую хеш-таблицу меньшего размера.

Кроме того, затраченное время на перезапись компенсируется более быстрым выполнением операторов словарей.

Очереди с приоритетами

Название "очередь с приоритетами" происходит от того вида упорядочивания (сортировки), которому подвергаются данные этого АТД.

Слово "очередь" предполагает, что люди (или входные элементы) ожидают некоторого обслуживания, а слова "с приоритетом" обозначают, что обслуживание будет производиться не по принципу "первый пришел - первым получил обслуживание", как происходит с АТД ***QUEUE*** (Очередь), а на основе приоритетов всех персон, стоящих в очереди.

Например, в приемном отделении больницы сначала принимают пациентов с потенциально фатальными диагнозами, независимо от того, как долго они или другие пациенты находились в очереди.

Очереди с приоритетами

Термин "***очередь с приоритетами***" подразумевает, что на множестве элементов задана функция приоритета (***priority***), т.е. для каждого элемента ***a*** множества можно вычислить функцию ***p(a)***, приоритет элемента ***a***, которая обычно принимает значения из множества действительных чисел, или, в более общем случае, из некоторого линейно упорядоченного множества.

Очередь с приоритетами - это АТД, основанный на модели множеств с операторами ***INSERT*** и ***DELETEMIN***, а также с оператором ***MAKENULL*** для инициализации структуры данных.

Оператор ***INSERT*** для очередей с приоритетами понимается в обычном смысле, тогда как ***DELETEMIN*** является функцией, которая возвращает элемент с наименьшим приоритетом и в качестве побочного эффекта удаляет его из множества. Таким образом, ***DELETEMIN*** является комбинацией операторов ***DELETE*** и ***MIN***, которые были описаны выше в этой теме.

Очереди с приоритетами. Пример

Очередь с приоритетами, которая возникает среди множества процессов, ожидающих обслуживания совместно используемыми ресурсами компьютерной системы.

Обычно системные разработчики стараются сделать так, чтобы короткие процессы выполнялись незамедлительно (на практике "незамедлительно" может означать одну-две секунды), т.е. такие процессы получают более высокие приоритеты, чем процессы, которые требуют (или уже израсходовали) значительного количества системного времени.

Процессы, которые требуют нескольких секунд машинного времени, не выполняются сразу - рациональная стратегия разделения ресурсов откладывает их до тех пор, пока не будут выполнены короткие процессы.

Однако нельзя переусердствовать в применении этой стратегии, иначе процессы, требующие значительно больше времени, чем "средние" процессы, вообще никогда не смогут получить кванта машинного времени и будут находиться в режиме ожидания вечно.

Очереди с приоритетами. Пример

Один возможный путь удовлетворить короткие процессы и не заблокировать большие состоит в задании процессу ***P*** приоритета, который вычисляется по формуле

$$100t_{исп}(P) - t_{нач}(P),$$

где параметр $t_{исп}(P)$ равен времени, израсходованному процессом ***P*** ранее, а $t_{нач}(P)$ - время, прошедшее от начала инициализации процесса, отсчитываемое от некоего "нулевого времени".

В общем случае приоритеты будут большими отрицательными целыми числами, если, конечно, $t_{нач}(P)$ не отсчитывается от "нулевого времени" в будущем. Число **100** в приведенной формуле пришло из практики и не поддается четкому логическому обоснованию; оно должно быть несколько больше числа ожидаемых активных процессов.

Очереди с приоритетами. Пример

Легко увидеть, что если всегда сначала выполняется процесс с наименьшим приоритетным числом и если в очереди немного коротких процессов, то в течение некоторого (достаточно продолжительного) времени процессу, который не закончился за один квант машинного времени, будет выделено не менее 1% машинного времени.

Если этот процент надо увеличить или уменьшить, следует заменить константу **100** в формуле вычисления приоритета.

Очереди с приоритетами

Представим процесс в виде записи, содержащей поле ***id*** идентификатора процесса и поле ***priority*** со значением приоритета, т.е. тип процесса ***processtype*** определим следующим образом:

```
type   processtype = record
            id: integer;
            priority: integer
       end;
```

Значение приоритета определено как целое число.

Функцию определения приоритета (но не его вычисления) можно определить так:

```
function  p ( a: processtype ): integer;
begin
    p:=a.priority
end;
```

Очереди с приоритетами

Для того чтобы для выбранных процессов зарезервировать определенное количество квантов машинного времени, компьютерная система поддерживает очередь с приоритетом ***WAITING*** (Ожидание), состоящую из элементов типа ***processtype*** и использующую две процедуры:

- ***initial*** (инициализация)
- и ***select*** (выбирать).

Очередь ***WAITING*** управляется с помощью операторов ***INSERT*** и ***DELETEMIN***.

При инициализации нового процесса вызывается процедура ***initial***, которая указывает записи, соответствующей новому процессу, место в очереди ***WAITING***.

Очереди с приоритетами

Процедура ***select*** вызывается тогда, когда система хочет выделить квант машинного времени какому-либо процессу. Запись для выбранного процесса удаляется из очереди ***WAITING***, но сохраняется посредством ***select*** для повторного ввода в очередь (если процесс не закончился за выделенное время) с новым приоритетом - приоритет увеличивается на 100 при пересчете времени $t_{исп}$ (когда $t_{исп}$ увеличивается на единицу).

Можно использовать функцию ***currenttime*** (которая возвращает текущее машинное время) для вычисления временных интервалов, отводимых системой процессам; обычно эти интервалы измеряются в микросекундах.

Будем также использовать процедуру ***execute(P)*** для вызова процесса с идентификатором ***P*** на исполнение в течение одного кванта времени. В листинге 5.11 приведены коды процедур ***initial*** и ***select***.

Листинг 5.11. Выделение процессам машинного времени

```
procedure  initial ( P: integer );
{initial указывает процессу с идентификатором P место в очереди}
var   process: processtype;
begin
      process.id:= P;
      process.priority:= - currenttime;
      INSERT(process, WAITING)
end;           { initial }
```

Листинг 5.11. Выделение процессам машинного времени

```
procedure  select;
{select выделяет квант времени процессу с наивысшим приоритетом}
var   begintime, endtime: integer;
      process: processtype;
begin
     process:= ^DELETEMIN(WAITING);
     {DELETEMIN возвращает указатель на удаленный элемент}
     begintime:= currenttime;
     execute(process.id);
     endtime:= currenttime;
     process.priority: =process.priority+100*(endtime-begintime);
     { пересчет приоритета }
     INSERT(process, WAITING)
     { занесение процесса в очередь с новым приоритетом }
end;          { select }
```

Реализация очередей с приоритетами

За исключением хеш-таблиц, те реализации множеств, которые рассмотрены ранее, можно применить к очередям с приоритетами.

Хеш-таблицы следует исключить, поскольку они не имеют подходящего механизма нахождения минимального элемента. Т.е., применение хеширования привносит дополнительные сложности, которых лишены, например, связанные списки.

При использовании связанных списков можно выбрать вид упорядочивания элементов списка или оставить его несортированным.

Если список отсортирован, то нахождение минимального элемента просто - это первый элемент списка. Но вместе с тем вставка нового элемента в отсортированный список требует просмотра в среднем половины элементов списка.

Если оставить список неупорядоченным, упрощается вставка нового элемента и затрудняется поиск минимального элемента.

Реализация очередей с приоритетами

- В листинге 5.12 показана реализация функции ***DELETEMIN*** для несортированного списка элементов, имеющих тип ***processtype***, описанный выше. Также приведены объявления для типа ячеек списка и для АТД ***PRIORITYQUEUE*** (Очередь с приоритетами).
- Реализация операторов ***INSERT*** и ***MAKENULL*** (как для отсортированных, так и для несортированных списков) не вызывает затруднений, и может быть выполнена в качестве упражнения.

Листинг 5.12. Реализация очереди с приоритетами связанным списком

```
type  celltype = record
          element: processtype;
          next: ^celltype
       end;
       PRIORITYQUEUE = ^celltype;  { ячейка ук-ет на заголовок списка }

function  DELETEMIN ( var A: PRIORITYQUEUE ): ^celltype;
var   current: ^celltype; {ук-ет на ячейку, кот. будет проверена сл-щей }
       lowpriority: integer; { содержит ранее найденный наим. приоритет }
       prewinner: ^celltype; { ук-ет на ячейку, сод-щую элемент с наим.
                                           приоритетом }
```

Листинг 5.12

```
begin
    if A^.next = nil   then
        error('Нельзя найти минимум в пустом списке')
    else  begin
        lowpriority:= p(A^.next^. element) ;
        { функция p возвращает приоритет первого элемента. A указывает на
            ячейку заголовка, которая не содержит элемента }
        prewinner:= A;
        current:= A^.next;
        while  current^.next <> nil  do  begin
            { сравнение текущего наим. приоритета с приоритетом сл.
                элемента }
            if  p(current^.next^.element)<lowpriority  then  begin
                prewinner:= current;
                lowpriority:= p(current^.next^.element)
            end;
            current:= currentT. next
        end;
```

Листинг 5.12

```
        DELETEMIN:= prewinner^.next;
        { возвращает указатель на найденный элемент }
        prewinner^. next: = prewinner^. next^. next
        { удаляет найденный элемент из списка }
    end
end;    { DELETEMIN }
```

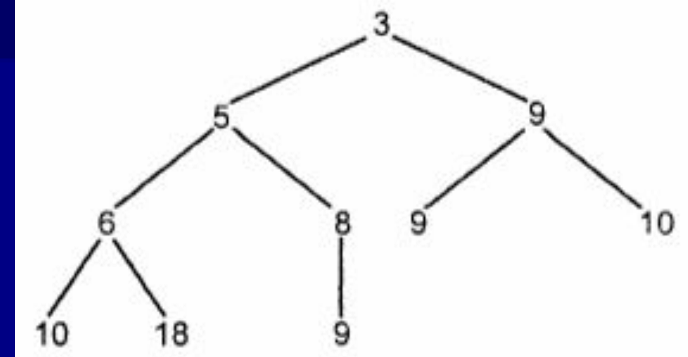
Реализация очереди с приоритетами частично упорядоченными деревьями

Для представления очередей с приоритетами посредством списков требуется затратить время, пропорциональное ***n*** для выполнения операторов ***INSERT*** и ***DELETEMIN*** на множестве размера ***n***.

Существует другая реализация очередей с приоритетами, в которой на выполнение этих операторов требуется порядка ***O(log n)*** шагов.

Реализация очереди с приоритетами частично упорядоченными деревьями

Основная идея такой реализации заключается в том, чтобы организовать элементы очереди в виде ***сбалансированного*** (по возможности) ***двоичного дерева***.



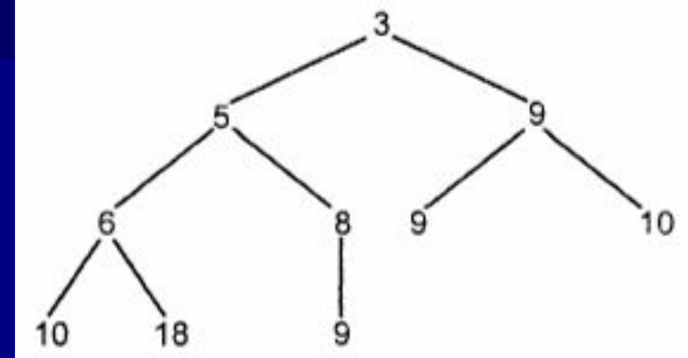
Сбалансированность в данном случае конструктивно можно определить так: листья возможны только на самом нижнем уровне или на предыдущем, но не на более высоких уровнях.

Другими словами, максимально возможная сбалансированность двоичного дерева здесь понимается в том смысле, чтобы дерево было как можно "ближе" к полному двоичному дереву.

На нижнем уровне, где некоторые листья могут отсутствовать, требуется, чтобы все отсутствующие листья в принципе могли располагаться только справа от присутствующих листьев нижнего уровня.

Реализация очереди с приоритетами частично упорядоченными деревьями

Более существенно, что дерево ***частично упорядочено***. Это означает, что приоритет узла ***v*** не больше приоритета любого сына узла ***v***, где приоритет узла - это значение приоритета элемента, хранящегося в данном узле.



Из рисунка видно, что малые значения приоритетов не могут появиться на более высоком уровне, где есть большие значения приоритетов.

Например, на третьем уровне располагаются приоритеты 6 и 8, которые меньше приоритета 9, расположенного на втором уровне. Но родитель узлов с приоритетами 6 и 8, расположенный на втором уровне, имеет (и должен иметь) по крайней мере не больший приоритет, чем его сыновья.

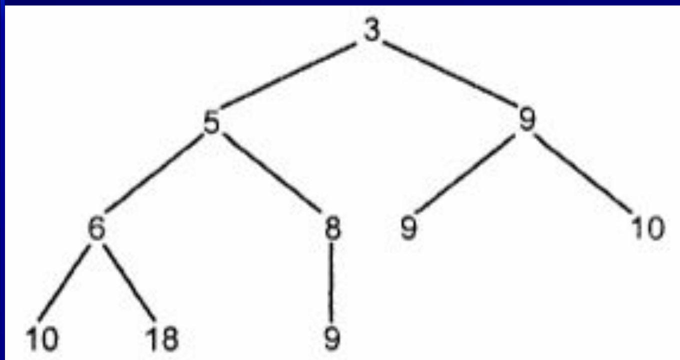
Реализация очереди с приоритетами частично упорядоченными деревьями

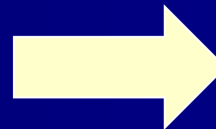
При выполнении функции ***DELETEMIN*** возвращается элемент с минимальным приоритетом, который, очевидно, должен быть корнем дерева. Но если просто удалить корень, то тем самым разрушится дерево.

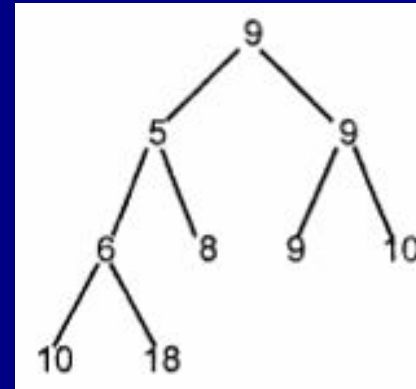
Чтобы не разрушить дерево и сохранить частичную упорядоченность значений приоритетов на дереве после удаления корня, необходимо выполнить следующее: сначала находим на самом нижнем уровне самый правый узел и временно помещаем его в корень дерева.

Реализация очереди с приоритетами частично упорядоченными деревьями

На рисунке показаны изменения, сделанные на дереве после удаления корня.



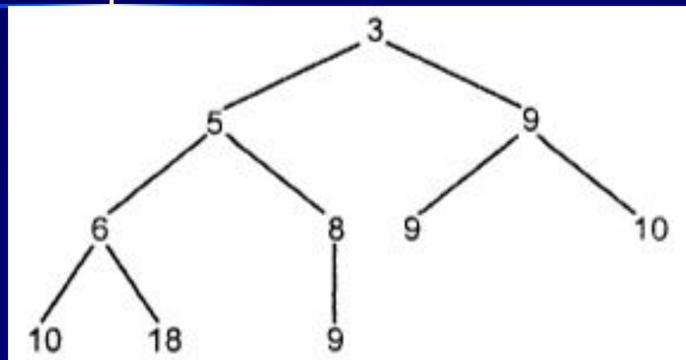


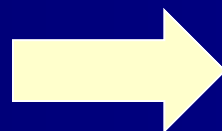


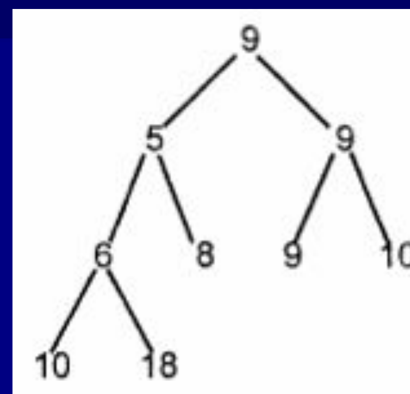
Затем этот элемент спускаем по ветвям дерева вниз (на более низкие уровни), по пути меняя его местами с сыновьями, имеющими меньший приоритет, до тех пор, пока этот элемент не станет листом или не встанет в позицию, где его сыновья будут иметь по крайней мере не меньший приоритет.

Т.о., надо поменять местами корень и его сына, имеющего меньший приоритет, равный 5.

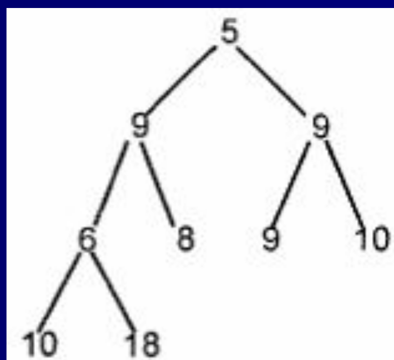
Реализация очереди с приоритетами частично упорядоченными деревьями







В результате получим дерево:

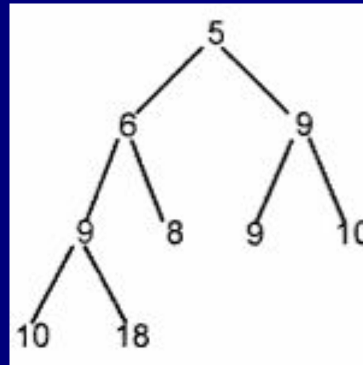


Наш элемент надо спустить еще на более низкий уровень, так как его сыновья сейчас имеют приоритеты 6 и 8.

Меняем его местами с сыном, имеющим наименьший приоритет 6.

Реализация очереди с приоритетами частично упорядоченными деревьями

Получено в результате такого обмена новое дерево:



Это дерево уже является частично упорядоченным и его дальше не надо менять.

Реализация очереди с приоритетами частично упорядоченными деревьями

Если при таком преобразовании узел ***v*** содержит элемент с приоритетом ***a*** и его сыновьями являются элементы с приоритетами ***b*** и ***c***, из которых хотя бы один меньше ***a***, то меняются местами элемент с приоритетом ***a*** и элемент с наименьшим приоритетом из ***b*** и ***c***. В результате в узле ***v*** будет находиться элемент с приоритетом, не превышающим приоритеты своих сыновей.

Таким образом, описанный процесс спуска элемента по дереву приводит к частичному упорядочиванию двоичного дерева.

Реализация очереди с приоритетами частично упорядоченными деревьями

Рассмотрим, как на частично упорядоченных деревьях работает оператор ***INSERT***.

Сначала поместим новый элемент в самую левую свободную позицию на самом нижнем уровне, если же этот уровень заполнен, то следует начать новый уровень.

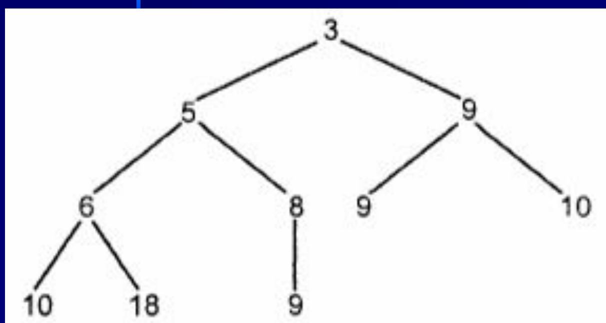
Если новый элемент имеет меньший приоритет, чем у его родителя, то они меняются местами. Таким образом, новый элемент теперь находится в позиции, в которой у его сыновей больший приоритет, чем у него.

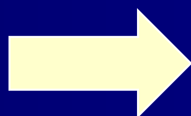
Но возможно, что у его нового родителя приоритет больше, чем у него. В этом случае они также меняются местами.

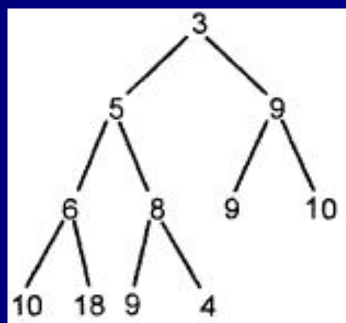
Этот процесс продолжается до тех пор, пока новый элемент не окажется в корне дерева или не займет позицию, где приоритет родителя не будет превышать приоритет нового элемента.

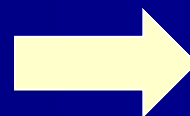
Реализация очереди с приоритетами частично упорядоченными деревьями

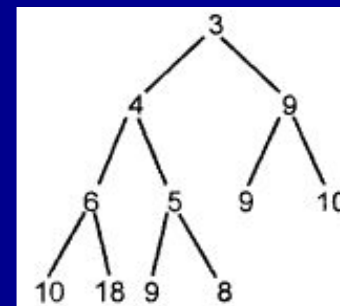
Этапы перемещения нового элемента:











Реализация частично упорядоченных деревьев посредством массивов

Исходя из того, что рассматриваемые нами деревья являются двоичными, по возможности сбалансированными и на самом нижнем уровне все листья "сдвинуты" влево, можно применить для этих деревьев необычное представление, которое называется ***куча***.

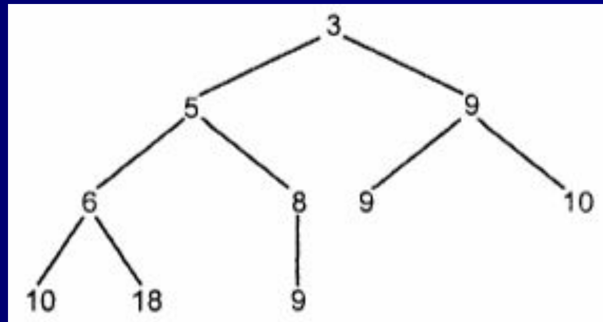
В этом представлении используется массив, назовем его ***A***, в котором первые ***n*** позиций соответствуют ***n*** узлам дерева.

A[1] содержит корень дерева. Левый сын узла ***A[i]***, если он существует, находится в ячейке ***A[2i],*** а правый сын, если он также существует, - в ячейке ***A[2i+1]***.

Обратное преобразование: если сын находится в ячейке ***A[i]***, ***i***>1, то его родитель - в ячейке ***A[i div 2]***.

Отсюда видно, что узлы дерева заполняют ячейки ***A[1]***, ***A[2]***, ..., ***A[n]*** последовательно уровень за уровнем, начиная с корня, а внутри уровня - слева направо.

Реализация частично упорядоченных деревьев посредством массивов



Это дерево будет представлено в массиве следующей последовательностью своих узлов:

3, **5**, **9**, **6**, **8**, **9**, **10**, **10**, **18**, **9**.

Реализация частично упорядоченных деревьев посредством массивов

В данном случае можно объявить АТД *PRIORITYQUEUE* (Очередь с приоритетами) как записи с полем *contents* для массива элементов типа, например, *processtype*, и полем *last* целочисленного типа, значение которого указывает на последний используемый элемент массива.

Если также ввести константу *maxsize*, равную количеству элементов очереди, то получим следующее объявление типов:

```
type  PRIORITYQUEUE = record
      contents: array[ 1. .maxsize]  of  processtype;
      last: integer
end;
```

Листинг 5.13. Реализация очереди с приоритетами посредством массива

```
procedure  MAKENULL ( var A: PRIORITYQUEUE );
begin
        A.last:= 0
end;     { MAKENULL }
```

Листинг 5.13

```
procedure  INSERT ( x: processtype; var A: PRIORITYQUEUE );
var   i: integer;  temp: processtype;
begin
     if  A.last >= maxsize  then  error('Очередь заполнена')
     else  begin
          A.last:= A.last + 1;
          A.contents[A.last]:= x;
          i:= A.last;   { i — индекс текущей позиции x }
          while (i>1) and (p(A.contents[i])<p(A.contents[i div 2])) do  begin
               { перемещение x вверх по дереву путем обмена
                         местами с родителем, имеющим больший приоритет }
               temp:= A.contents[i];
               A.contents[i]:= A.contents[i div 2];
               A.contents[i div 2]:= temp;
               i:= i div 2
     end; end
end;          { INSERT }
```

Листинг 5.13

```
function  DELETEMIN ( var A: PRIORITYQUEUE ): ^processtype;
var  i, j: integer;      temp: processtype;
       minimum: ^processtype;
begin
     if  A.last=0  then  error('Очередь пуста')
     else  begin
          new(minimum);
          minimum ^:= A.contents[1];
          A.contents[1J:= A.contents[A.last];
          A.last:= A.last-1;
          i:=1;
          while  i <= A.last div 2  do  begin
               { перемещение старого последнего элемента вниз по дереву }
               if  (p(A.contents[2*i]) < p(A.contents[2*i+1]))
                         or  B*i=A.last)  then   j:=2*i
               else   j:=2*i+1;
               { j будет сыном i с наименьшим приоритетом или, если 2*i=A.last,
                   будет просто сыном i }
```

Листинг 5.13

```
            if  p(A.contents[i]) > p(A.contents[j])  then  begin
                { обмен старого последнего элемента с сыном,
                имеющим наименьший приоритет }
                temp:= A.contents[i];
                A.contents[i]:= A.contents[j];
                A.contents[j]:= temp;
                i:= j;
            end
            else   DELETEMIN ^:=minimum
            { дальше перемещение элемента невозможно }
        end;
        DELETEMIN ^:=minimum    { элемент дошел до листа }
    end
end;    { DELETEMIN }
```