Аргов Д.И.

Структуры данных

учебное пособие

Рыбинск, 2016 г.

Оглавление:

- Линейный список и операции над ним
- Стек и операции над ним
- Анализ корректности скобочной структуры
- Очередь и операции над ней
- Динамический тип. Указатели
- Динамический стек
- Динамическая очередь
- Разреженные матрицы
- Конечные автоматы
- ХЕШ-функции
- Фибоначчиев поиск
- Отображения
- Множество
- Словари
- Дерево поиска
- Красно-черные деревья
- Деревья отрезков
- Дерево Фенвика
- Поиск мостов
- Суффиксное дерево (Суффиксное дерево (gusfield.djvuСуффиксное дерево (gusfield.djvu)
- Игры двух лиц
- Длинная арифметика
- Очередь с приоритетом на основе бинарной кучи
- Алгоритм сжатия информации методом Хаффмана

Линейный список и операции над ним

- Линейный список – это способ организации хранения информации, при котором все элементы равноправны и хранятся последовательно. В реальной жизни мы часто встречаем линейные списки, например, классный журнал в школе – фамилии учеников и их оценки и есть линейный список. Вспомним, какие операции допустимы над таким списком:
- **формирование списка** – занесение в него первоначальной информации;
- **вывод списка** – просмотр его содержимого;
- **добавление нового элемента**;
- **удаление элемента**;
- **поиск информации в списке**;
- **сортировка списка** – упорядочивание элементов, например по алфавиту.
- Рассмотрим эти операции на примере использования статического массива.

Формирование списка

Обычный массив в паскале обладает рядом недостатков:

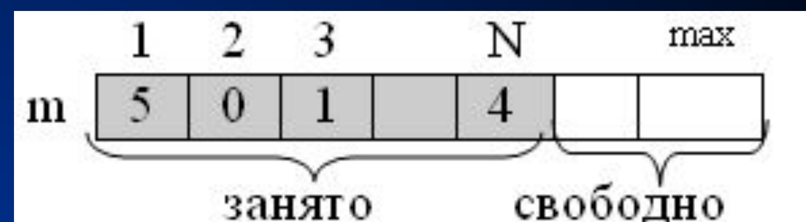
- его размер нужно указать до начала работы программы, когда размер списка еще не известен;
- в ходе работы программы длина списка может меняться, а массива – нет;
- при удалении элемента списка нельзя удалить элемент массива.

Для решения этих проблем размер массива берут равным максимальному размеру списка, а используют только N первых ячеек (как в записной книжке).


```
Const Max=100;
Type tList=array[1..Max] of тип элементов;
Var  m:tList; {список}
   N:integer; {его размер}
{Прежде, чем вводить список, определим его размер (N)}
  Write('Введите размер списка');
  Readln(N);
  For i:=1 to N do
   Begin
    Write('Введите ',I',' элемент списка');
    Readln(m[i]);
   End;
```

Вывод списка

- Перебираем все N элементов и выводим их на экран. Существуют два способа вывода элементов:



- а) в столбец

```
For i:=1 to N do
    Writeln(m[i]);
```

- б) в строку

```
For i:=1 to N do
    Write(m[i],' ');
```

Добавление элемента в список

- Существует единственное место, куда можно разместить новый элемент, не нарушая принципов построения линейного списка.
- Действительно: с 1 по N элементы заняты, туда нельзя, с N+1 по Max – свободны, но можно нарушить принцип последовательного расположения элементов, то есть остается единственный вариант – N+1ый элемент.

```
N:=N+1;{увеличим размер списка}
Write('Введите элемент списка');
Readln(m[N]);
```

Удаление элемента из списка

- Существуют два способа удаления:
- а) удаление с нарушением порядка следования элементов.
- Проще всего удалить последний элемент – достаточно уменьшить N, а что делать, если надо удалить не последний? Пусть i номер удаляемого элемента. Скопируем последний элемент (с номером N) в ячейку с удаляемым (номер i). Теперь у нас в списке два последних элемента, поэтому уменьшим размер списка на 1, избавляясь от «дублера».

```
M[i]:=m[N];
N:=N-1;
```

Достоинство: огромная скорость работы. Скорость работы алгоритма нельзя определить в единицах времени, так как время определяется мощностью процессора. Поэтому эффективность алгоритма измеряют в операциях. Данный алгоритм вообще не зависит от размера списка, поэтому его скорость равна 1 и обозначается O(1), где O – это верхняя оценка скорости работы алгоритма.

Недостаток: нарушается порядок следования элементов, поэтому данный алгоритм не применим в упорядоченных списках.

N

Удаление элемента из списка

- удаление элемента с сохранением порядка следования элементов.
- Рассмотрим картинку. Пусть i номер удаляемого элемента. Когда мы удалим 3, то ее место должна занять 7, а ее место – 5 и так далее. То есть все последующие элементы должны сместиться влево на 1 позицию.

```
For k:=I to N-1
   m[k]:=m[k+1];
N:=N-1;
```

	1	2	3	4	5	6	7	8	9	10
m	5	3	7	5	4	3				

i: 2 N: 5

Достоинство: сохраняется порядок следования элементов.
Недостаток: худшая скорость – O(N), средняя – O(N/2) сдвигов.

Поиск элемента в списке

- Пожалуй, самая важная операция, так как используется чаще всего. От ее эффективности зависит скорость работы программы в целом. Ключом поиска (key) называется искомый элемент. Задачей поиска является обнаружить место расположения ключа или сообщить, что его нет. Рассмотрим несколько вариантов поиска и оценим их эффективность:
- а) «тупой» полный перебор.
- Так как мы заранее не знаем, где может быть расположен ключ, то мы последовательно перебираем элементы и сравниваем их с ключом.

```
Write('Введите искомый элемент');
Readln(key); k:=0;
For i:=1 to N do
  If m[i]=key Then k:=i;
If k=0
Then Write('Искомого нет')
Else Write(k);
```

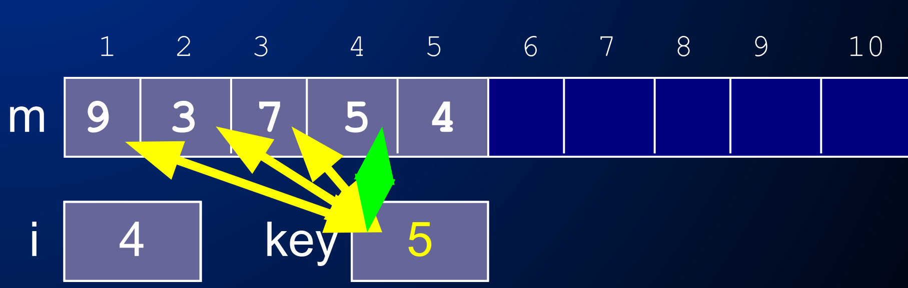
Этот вариант решения является самым неэффективным, так как, обнаружив искомый элемент, алгоритм продолжит поиск. Поэтому его средняя скорость равняется худшей и равняется O (2N). Почему так? Всего у нас N элементов, на каждый приходится 2 проверки: m[i]=key и i>N.

Поиск элемента в списке

- б) полный перебор.
- Заменим цикл for на другой, зачем нам искать, если уже нашли?
- Этот вариант решения является более эффективным, так как, обнаружив искомый элемент, алгоритм прекращает поиск. Поэтому его худшая скорость равняется O(2N), а средняя – O(2N/2)=O(N). Почему так? Всего у нас N элементов, искомый равновероятно может находиться в начале и в конце. Тогда среднее количество операций сравнения будет равно:
.
- На каждую операцию приходится 2 проверки.

$$\frac{1+2+3+...+N}{N} \approx \frac{N}{2}$$

```
Write('Введите искомый элемент');
Readln(key); i:=1;
While (i<n)and(m[i]<>key) do
   I:=i+1;
If m[i]<>key
Then Write('Искомого нет')
Else Write(i);
```



Поиск элемента в списке

- в) <u>поиск «с барьером».</u>
- Попробуем ускорить предыдущий алгоритм в 2 раза.
- Для этого надо избавиться от одной проверки. Их две:
- сравнение с ключом (m[i]<>key), от этой проверки избавиться нельзя – как найти не сравнивая?
- Дошли ли до конца (I<N)?
- Придется убирать ее, так как другого варианта нет. За что она отвечает? Если алгоритм не обнаружит искомый, то он остановится, дойдя до конца. Если убрать эту проверку, то алгоритм в этом случае станет бесконечным. Что же делать? А мы сделаем так, чтобы алгоритм всегда находил искомый! Для этого поместим ключ в первый свободный элемент – это и будет барьер. Теперь наш алгоритм поиска всегда найдет искомый: либо там, где он был, либо там, куда мы его поместили.

```
Write('Введите искомый элемент');
Readln(key); i:=1;
m[N+1]:=key;
While (m[i]<>key) do
   I:=i+1;
If i>N
Then Write('Искомого нет')
Else Write(i);
```

Этот вариант решения является самым эффективным, так как, его средняя скорость равняется O(N/2), а худшая – O(N).

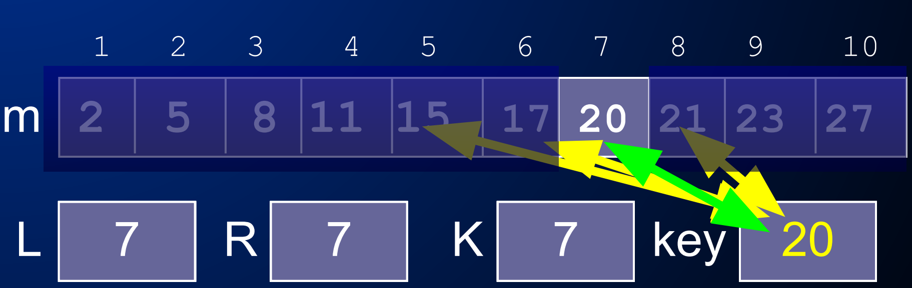
Поиск элемента в списке

г) бинарный поиск.

Кажется, что в предыдущем алгоритме мы достигли максимальной скорости, но что Вы скажите про алгоритм, который для 1000 элементов в худшем случае потратит 10 сравнений, для 1000 000 – 20 сравнений, для 1 000 000 000 – 30 сравнений? Невероятно? Реально!

Заведем два указателя на левую и правую границы поиска: L=1, R=N. Мы будем искать key=20. Найдем середину списка k=(L+R) div 2=(1+10) div 2=5 . m[5]=15. Заметим, что 20>15, следовательно, левее числа 15 не может быть 20, поэтому мы уменьшаем область поиска в 2 раза – L:=k+1=6. Повторим процесс: k=(L+R) div 2=(6+10) div 2=8 . m[8]=21. Заметим, что 20<21, следовательно, правее числа 21 не может быть 20 – R:=k-1=7. И так далее. Левая и правая граница сближаются, каждый раз уменьшая область поиска в 2 раза. Это логарифмическая скорость: O ($\log_2(N)$).

```
Write('Введите искомый элемент');
Readln(key); L:=1; R:=N;
Repeat
 K:=(L+R) div 2;
 If Key<m[k] Then R:=k-1
 Else L:=k+1
until (m[k]=key)or(L>R);
If m[k]<>key
Then Write('Искомого нет')
Else Write(k);
```



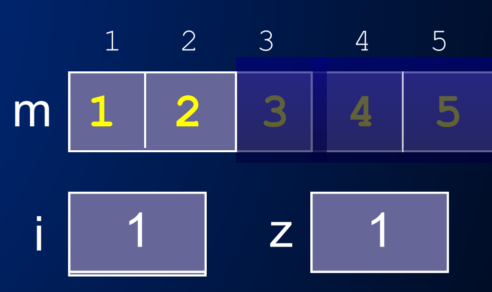
Сортировка элементов списка

Существуют десятки различных алгоритмов сортировки, они отличаются по сложности и скорости:

- простые алгоритмы, скорость $O(N^2)$;
- сложные алгоритмы, скорость $O(N*\log_2 N)$.
- а) сортировка «Пузырек».

Данный алгоритм является наиболее эффективным среди простых. Пусть имеется список целых чисел. Перебираем его и сравниваем два соседних, если предыдущий элемент оказывается больше последующего, то меняем их местами. После такого просмотра самый большой элемент сместиться в конец списка, никакой другой элемент не сможет его оттуда сдвинуть. Это позволяет нам уменьшить размер неотсортированной части на 1. Процесс просмотра списка повторяем вновь, пока при очередном просмотре не будет сделано ни одной перестановки

```
z:=N;
repeat
 f:=true;
 For i:=1 to z-1 do
  If m[i]>m[i+1]
  then Begin
         b:=m[i]; m[i]:=m[i+1];
         m[i+1]:=b; f:=false
       End;
 z:=z-1
until f
```

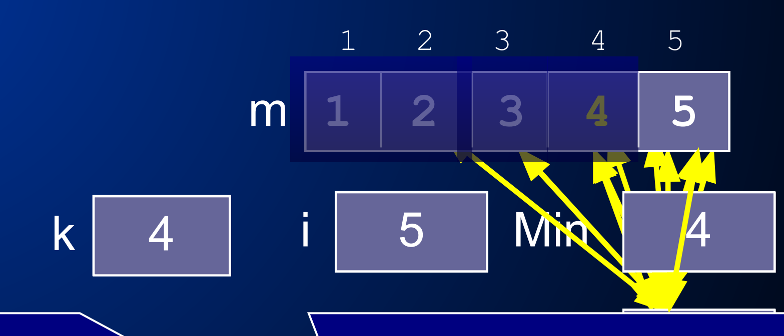


Скорость работы программы в худшем случае $O(N^2/2)$. Цикл for будет выполняться N-1, N-2, N-3,…1 раз. Цикл repeat будет выполняться в худшем случае N раз.

Сортировка элементов списка

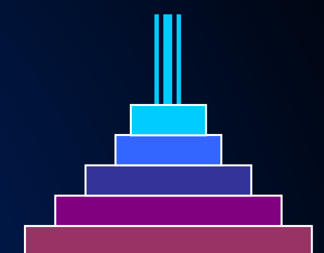
- б) сортировка «Метод простого выбора».
- Разделим список на две части: отсортированную и неотсортированную. В неотсортированной части ищем наименьший элемент и переставляем его в конец отсортированной части, при этом изменяем размеры этих частей. Процесс повторяем N-1 раз.

```
For k:=1 to N-1 do
 Begin
   Min:=k
   For i:=k+1 to N do
    If m[i]<m[min] then min:=I;
   b:=m[min];
   m[min]:=m[k];
   m[k]:=b;
 end;
```



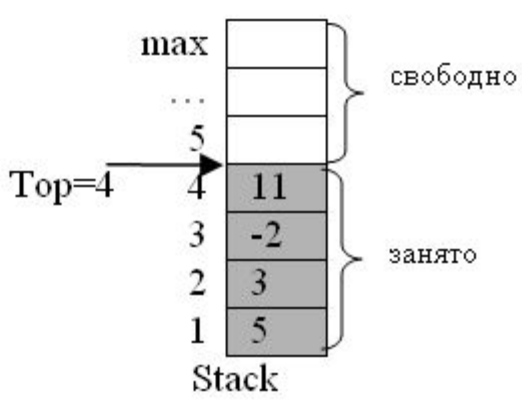
Скорость работы программы всегда $O(N^2/2)$. В отличие от метода «Пузырька», сортировка не может завершиться досрочно даже, если список уже упорядочен

Стек и операции над ним



- Стек (Stack) – это очередь особого вида, постановка и извлечение элементов в которой осуществляется с одного конца. Над стеком возможны две операции: Push – поместить элемент в стек и Pop – извлечь элемент из стека. Моделью стека является детская пирамидка. Стек часто называю очередью типа LIFO (Last In – First Out или последний пришел – первым уйдет). Можно заметить, что в пирамидке нижнее кольцо было положено первым, а вот снять его можно только последним. На механизм стека опираются подпрограммы и используют многие алгоритмы. Паскаль имеет собственный стек, но пользователь не имеет к нему доступа, поэтому собственный стек придется создавать самим. Для этой цели прекрасно подойдет массив.

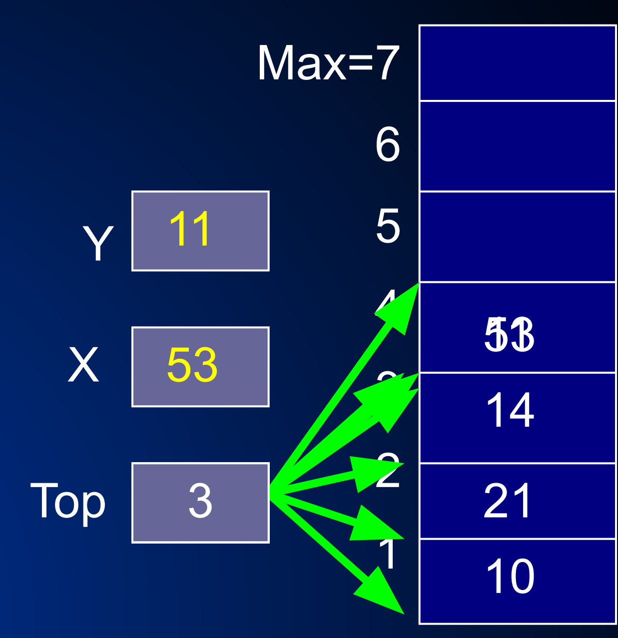
- Const Max=100;
- Type tStack=array[1..Max] of тип элементов стека;
- Var Stack:tStack;
- Top:integer;{вершина стека}
- При помещении элемента в стек (операция Push), его вершина Top увеличивается на 1 и в свободную ячейку массива помещается нужный элемент. Извлечение (операция Pop) происходит обратным образом.



- Рассмотрим соответствующие процедуры:

```
• Procedure Push(x:Тип элемента);
• Begin
• If Top=max{Стек переполнен?}
• Then writeln(‘Ошибка! Стек переполнен’)
• Else begin
•        Inc(top);{увеличить на 1 вершину стека}
•        Stack[top]:=x{поместить в стек элемент}
•      end
• End;
• Function Pop:Тип элемента);
• Begin
•   If Top=0 {Стек пуст?}
•   Then writeln(‘Ошибка! Стек пуст’)
•   Else begin
•        Pop:=Stack[top];{извлечь верхний элемент из стека}
•        Dec(top);{уменьшить на 1 вершину стека}
•       end
• End;
```

```
Procedure Push(x:Тип элемента);
Begin
If Top=max
Then writeln('Ошибка! Стек переполнен')
Else begin
        Inc(top);
        Stack[top]:=x
      end
End;
```




```
Function Pop:Тип элемента);
Begin
  If Top=0
  Then writeln('Ошибка!Стек пуст')
  Else begin
        Pop:=Stack[top];
        Dec(top);
       end
End;
```

```
Push(10);
Push(21);
Push(14);
Push(53);
X:=pop
Push(11);
Y:=pop;
```

Анализ корректности скобочной структуры

- Пусть имеется некоторое арифметическое выражение, например, (a+c)*(c-d). Необходимо, не учитывая операнды и операции, проверить правильность скобочного выражения. Мы рассмотрим два алгоритма.
- **Анализ скобочного выражения с использованием ранга.**
- Рассмотрим несколько вариантов скобочных структур:
- ()()(()) – корректная, (() – некорректная, ())) – некорректная.
- *Гипотеза: скобочное выражение корректно, если количество левых (открывающих) и правых (закрывающих) скобок равно.*
- Контр пример:)(. Количество скобок равно, но выражение некорректно. То есть, наша гипотеза является необходимым, но недостаточным условием корректности скобочного выражения.
- Ранг – числовая характеристика корректности скобочного выражения. В начале он равен 0. Когда мы встречаем левую скобку, то увеличиваем его значение на 1, когда правую – уменьшаем на 1. Если в процессе анализа значение ранга стало отрицательным, то скобочная структура некорректна. Если после завершения анализа значение ранга не равно 0, то скобочная структура некорректна.

Анализ скобочного выражения с использованием ранга

r 1 — ((a+b)*c+((x-3)-6)

Выражение некорректно

r -1 — ( )) (( )

Выражение некорректно

r 0 — (( ) (( )))

Выражение корректно

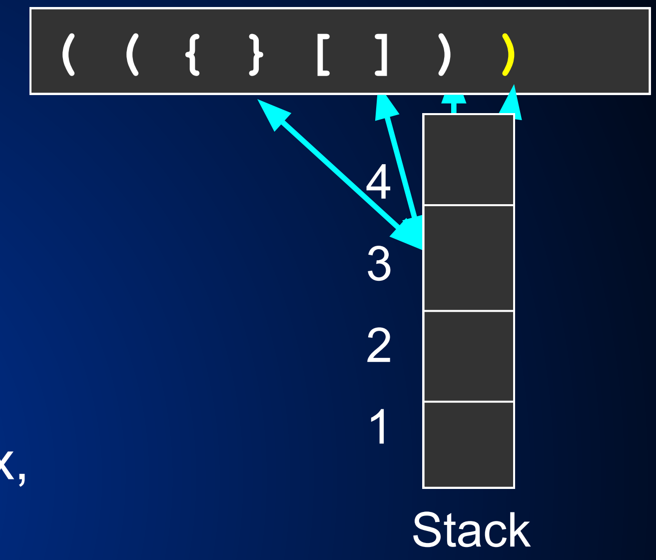
```
Var    s:string;
   R,i:integer;
Begin
  Readln(s); {ввод выражения}
  I:=0;r:=0;
  Repeat
   inc(i);
   if s[i]='(' then inc(r);
   if s[i]=')' then dec(r);
  Until (r<0)or(i>=length(s));
  If r=0
  Then writeln('скобочное выражение корректно')
  Else writeln('скобочное выражение
   некорректно')
End.
```

Анализ скобочного выражения с использованием стека

Предположим, что используется три типа скобок: (), {}, []. Необходимо проверить корректность выражения. Рассмотрим несколько вариантов скобочных структур:

( ){ }[ ] – корректно, [} – некорректно, [{] } – некорректно.

Для решения задачи заведем стек символов. Когда встречаем левую скобку, то помещаем ее в стек. Когда встречаем правую скобку, то извлекаем из стека верхнюю и сравниваем их, если скобки не парные, то скобочная структура некорректна. Если в процессе анализа была попытка извлечения из пустого стека, то скобочная структура некорректна. Если после завершения анализа стек не пуст, то скобочная структура некорректна.



(({ } [ ]))

4
3
2
1
Stack

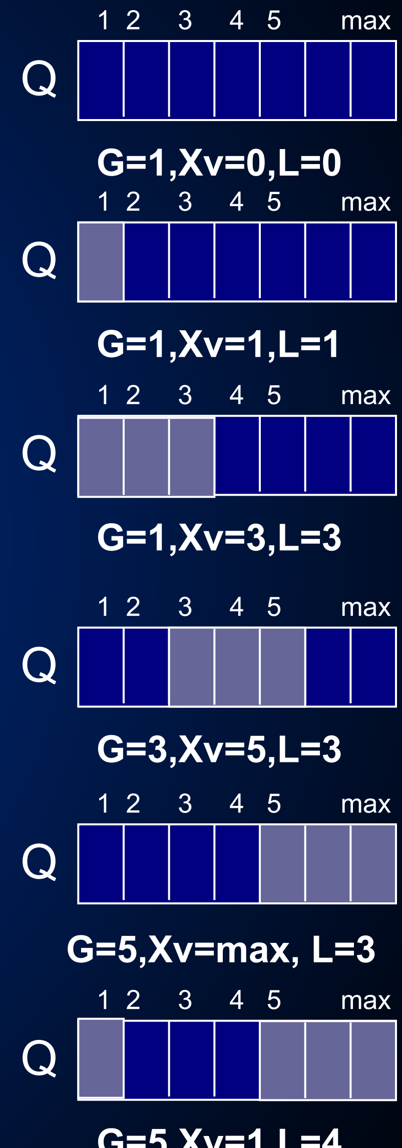
```
Var   s:string; f:Boolean;
  i:integer;
Begin
  Readln(s); {ввод выражения}
  I:=0;f:=true;{предположим, что она корректна}
  Repeat
   inc(i);
   case s[i] of
   '(`,'{`,'[`: Push(s[i]);
   `)': if (top=0) or (pop<>'(`) then f:=false;
   `]': if (top=0) or (pop<>'[`) then f:=false;
   `}': if (top=0) or (pop<>'{`) then f:=false;
   End; {case}
  Until (not f)or(i>=length(s));
  If f and(top=0)
  Then writeln(`скобочное выражение корректно')
  Else writeln(`скобочное выражение некорректно')
End.
```

Очередь и операции над ней

Очередь (Queue) – это линейный список особого вида, помещение элементов в который осуществляется с одного конца (хвоста), а извлечение элементов – с другого конца (головы). Над очередью возможны две операции: PutQ – поместить элемент в очередь и GetQ – извлечь элемент из очереди. Моделью очереди является очередь за пирожками в столовой. Очередь часто называю FIFO (First In – First Out или первый пришел – первым уйдет).

```
Const Max=100;
Type    tQueue=array[1..Max] of тип элементов стека;
Var Q: tQueue;
    G,Xv,L:integer;{голова, хвост, длина}
```

При помещении элемента в очередь, ее хвост увеличивается на 1 и в свободную ячейку массива помещается нужный элемент. Когда хвост достигнет Max, то есть правой границы массива, то он смещается к первому элементу. Извлечение элемента из очереди происходит аналогично, но со стороны головы. В начале программы очередь необходимо инициализировать: Xv:=0;L:=0;G:=1;



```
Procedure PutQ(x:Тип элемента);
Begin
  If L=max
  Then writeln('Очередь переполнена')
  Else begin
         Inc(Xv)
         If Xv>max then Xv:=1;
         Q[Xv]:=x;
         Inc(L)
   end
End;
```

```
Procedure GetQ(var x:Тип элемента);
Begin
  If L=0
  Then writeln('Очередь пуста)
  Else begin
         X:=Q[G];
         Inc(G);
         If G>max then G:=1;
         dec(L)
   end
End;
```

	1	2	3	4	5	6	7	8	max
Q	J	K					G	H	I

G 3   Xv 4   L 2

X A   y B

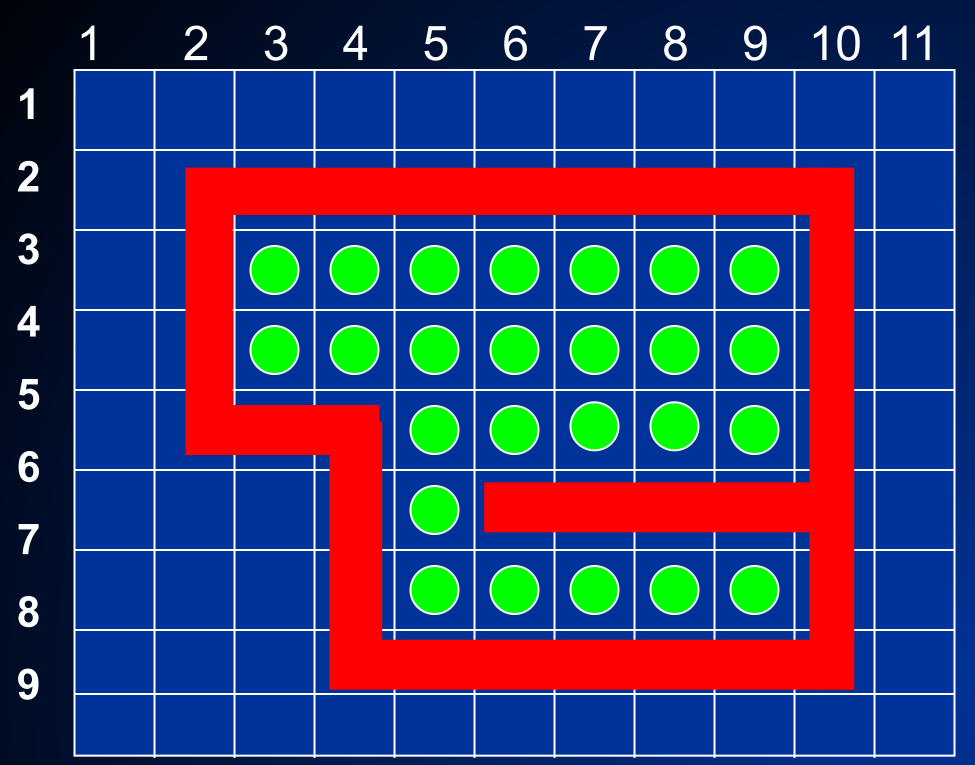
```
G:=1;Xv:=0;L:=0;
PutQ('A');
PutQ('B');
PutQ('C');
GetQ(x);
GetQ(y);
PutQ('D');
```

Хвост очереди достиг правой границы, при добавлении элемента сработает проверка и хвост станет равен 1

Волновой алгоритм. Закраска замкнутых областей

- Пусть имеется некоторая замкнутая область, граница которой имеет не 0 цвет, а внутренняя часть – 0 цвет. Пусть (x, y) – координаты любой внутренней точки области. Покрасим данную точку в нужный цвет и рассмотрим четырех ее соседей: сверху, снизу, слева, справа. Если соседняя точка имеет черный цвет, то есть, не покрашена, то покрасим ее и запомним ее координаты, чтобы повторить этот процесс для ее соседей. В результате цветная волна начнет разливаться по экрану. Наткнувшись на точку границы или уже покрашенную, процесс в данном месте прервется. Для хранения положения соседей воспользуемся очередью – массивом записей.

```
Type El=record
          i,j:integer;
        End;
tQueue=array[1..20000] of El;
Var   Q: tQueue;
  G,Xv,L:integer;{голова, хвост, длина}
```



Процесс продолжается аналогично для всех точек, находящихся в очереди. «Наткнувшись» на границу или уже покрашенную область, цветная волна останавливается, так как координаты точки не попадут в очередь

Q												
	4	4	5	4	3	4	5	3				
	7	8	7	6	7	9	8	8				

```
Нарисовать фигуру;    задать координаты внутренней точки
putPixel(x,y,цвет);{закрасить точку}   PutQ(x,y); {поместим ее в очередь}
While L>0 do Begin{пока очередь не пуста}
        GetQ(x,y);{извлекаем из очереди очередную точку}
    If (x>1)and(GetPixel(x-1,y)=0) {если сосед слева существует}
    Then  begin  {и не покрашен, то}
            PutPixel(x-1,y,цвет);{покрасить его}
            PutQ(x-1,y){поместить его координаты в очередь}
    End;
    If (x<639)and(GetPixel(x+1,y)=0){если сосед справа существует}
    Then begin  {и не покрашен, то}
           PutPixel(x+1,y,цвет);{покрасить его}
           PutQ(x+1,y){поместить его координаты в очередь}
   End;
     If (y>1)and(GetPixel(x,y-1)=0) {если сосед сверху существует}
      Then begin  {и не покрашен, то}
             PutPixel(x,y-1,цвет);{покрасить его}
             PutQ(x,y-1){поместить его координаты в очередь}
   End;
     If (y<479)and(GetPixel(x,y+1)=0){если сосед снизу существует}
      Then begin  {и не покрашен, то}
            PutPixel(x,y+1,цвет);{покрасить его}
            PutQ(x,y+1){поместить его координаты в очередь}
    End;
  End;
End.
```

Волновой алгоритм. Поиск пути в лабиринте

Пусть имеется лабиринт, представленный матрицей поля. A[i,j]=0, если клетка свободна и A[i,j]=255, если клетка содержит непроходимое препятствие. Пусть объект хочет найти путь из точки с координатами Si, Sj в точку с координатами Fi, Fj.

1. Поместить в очередь координаты выхода PutQ(Fi,Fj), пометить эту точку в матрице числом 1 (A[Fi, Fj]:=1).

2. Пока очередь не пуста делать

    2.1 извлечь координаты очередной точки (x,y);

    2.2 рассматриваем все соседние точки (сверху, снизу, слева, справа);

    2.3 Если сосед существует, и еще не помечен, то

        Пометить его числом на 1 больше (C[y,x]+1);

        поместить его координаты в очередь;

Для восстановления пути по заполненной матрице, объект «смотрит» вокруг себя и смещается в клетку с наименьшим числовым значением.

Q												
	5	5	5	4	3	3	3					
	8	9	7	9	9	8	7					

```
Нарисовать фигуру;    задать координаты внутренней точки
A[Fi,Fj]:=1;{пометим точку}   PutQ(Fi,Fj); {поместим ее в очередь}
While L>0 do Begin{пока очередь не пуста}
        GetQ(y,x);{извлекаем из очереди очередную точку}
        If (x>1)and(A[y,x-1]=0) {если сосед слева существует}
        Then  begin  {и не помечен, то}
             A[y,x-1]= A[y,x]+1;{пометим его}
                     PutQ(y,x-1){поместить его координаты в очередь}
        End;
        If (x<MaxX)and(A[y,x+1]=0) {если сосед справа существует}
        Then  begin  {и не помечен, то}
             A[y,x+1]= A[y,x]+1;{пометим его}
                     PutQ(y,x+1){поместить его координаты в очередь}
        End;
         If (y>1)and(A[y-1,x]=0) {если сосед сверху существует}
        Then  begin  {и не помечен, то}
             A[y-1,x]= A[y,x]+1;{пометим его}
                     PutQ(y-1,x){поместить его координаты в очередь}
        End;
        If (y<MaxY)and(A[y+1,x]=0) {если сосед снизу существует}
        Then  begin  {и не помечен, то}
             A[y+1,x]= A[y,x]+1;{пометим его}
                     PutQ(y+1,x){поместить его координаты в очередь}
        End;
   End;
End.
```

Динамический тип. Указатели

- Обычные переменные (глобальные или локальные) представляют собой ячейку памяти, которая хранит значение. Переменные создаются в момент компиляции (вызова подпрограммы) и существую до конца ее работы, даже, если необходимость в них отпала. Такая ситуация приводит к нерациональному использованию памяти. Вспомним главный недостаток массива – его размер должен быть заранее определен и не может меняться в ходе работы программы, то есть мы либо резервируем лишнюю память, которая не используется, либо сталкиваемся с проблемой нехватки места. Для решения таких проблем был создан динамический тип данных.
- Указателем называется особая переменная, которая хранит не значение, а адрес того места в ОЗУ, где хранится значение. Указатель будем обозначать кружком, а переменную прямоугольником.
- Type pInt=^integer; {тип указатель на переменную целого типа}
- Var   p,q,t:pInt; {переменные указатели}   p
-    X:integer; {переменная целого типа} x

- Указатель не может хранить значение, как обычная переменная, но в любой момент работы программы, программист может создать или ликвидировать объект у указателя. Над объектом допустимы любые операции, разрешенные над данным типом. Рассмотрим операции над указателями:
- **создание объекта у указателя**.
  New(p); p

- **присвоение значения объекту**: p:=10; указателю нельзя присвоить значение. Для того чтобы обратиться к объекту, необходимо использовать специальный знак – стрелку:
- p^:=10

- **уничтожение объекта.** Когда необходимость в динамической переменной отпала, ее можно ликвидировать, освободив тем самым место в ОЗУ. Для этого используется команда Dispose(p). Она уничтожает объект, освобождая занятую им память для повторного использования, но адрес в указателе сохраняется. В результате программист не сможет определить, есть у данного указателя объект или нет. Для решения этой проблемы используется специальная пустая ссылка nil.

- присвоение указателей. Указателю можно присвоить либо другой указатель, либо пустую ссылку nil. Рассмотрим пример:
- New(p);
- q:=p;
- p^:=10;
- write(q^); {10}Теперь у одного объекта имеется два имени p и q и одно значение 10.

p 10 q

- сравнивание указателей. Два указателя одного типа можно сравнить на = и на <> между собой и с константой nil. Например:
- if p=q then writeln(‘один объект’) else writeln (‘разные’)
- While p<>nil do …

Типичные ошибки при работе с указателями

- **обращение к несуществующему объекту**, то есть программист, не выполнив команду new(p);, пытается присвоить объекту значение p^:=10. Это может привести к катастрофическим последствиям, начиная от зависания ПК, до появления плавающей ошибки, которая появляется в случайное время. Ошибку очень сложно обнаружить, так как значение локальных указателей не равно nil.
- **потери памяти**. Когда ненужный объект не ликвидируют командой Dispose(p), он продолжает занимать память. Например, New(p)… New(p); Теперь существуют два объекта, но к первому доступ потерян.
- **создание ненужного объекта**. Очень часто нам нужен указатель для хранения адреса чужого объекта («бегать» по динамической цепочке), в этом случае не надо у данного указателя создавать объект.

С клавиатуры вводятся серии по 1000 натуральных чисел, последняя серия -1000 нулей. Вывести на экран наибольшую серию (максимальную по сумме элементов).

Классический вариант решения.

```
Const max=1000
Type mass=array[1..max]of word;
Var  t,mMax:mass;
     I:integer; s,smax:longint;
Begin
      Smax:=0; {максимальная сумма равна 0}
       Repeat
          S:=0; {текущая сумма равна 0}
          For i:=1 to max do
            Begin {считаем серию и найдем сумму ее элементов}
              Readln(t[i])s:=s+t[i]
            End;
         If s>sMax{если нашли большую серию, то}
        Then begin
                 sMax:=s; {запомним новую сумму}
                 For i:=1 to max do {переписываем серию в mMax}
              mMax[i]:=t[i]
         end
      Until s=0;
     For i:=1 to max do  write(mMax[i],' ')
End. {выведем лучшую серию}
```

```
Const max=1000
Type       pMass=^mass;
mass=array[1..max]of word;
Var  t,p,mMax:pmass;
     I:integer; s,smax:longint;
Begin
  Smax:=0; {максимальная сумма равна 0}
  New(t);New(mMax); {создать два динамических массива}
  Repeat
     S:=0; {текущая сумма равна 0}
     For i:=1 to max do
       Begin {считаем серию и найдем сумму ее элементов}
         Readln(t^[i])s:=s+t^[i]
       End;
    If s>sMax{если нашли большую серию, то}
    Then begin
           sMax:=s; {запомним новую сумму}
           {вместо копирования элементов массива, перекинем указатели}
      p:=t; t:=mMax;
            mMax:=p;
          end
   Until s=0;
   For i:=1 to max do  write(mMax^[i],' ')
End. {выведем лучшую серию}
```

t
mMax
p

Динамический линейный однонаправленный список

Основными проблемами классического статического списка на основе массива являются:

- неэффективное распределение памяти (резервирование лишней);
- невозможность увеличить размер списка в ходе программы;
- медленные операции вставки и удаления элементов без нарушения порядка их следования (сдвиг элементов в среднем O(n/2) штук).
- От этих проблем избавлен динамический список. Он формируется по мере необходимости путем добавления (вставки) нового звена. Рассмотрим основные операции над динамическим списком:

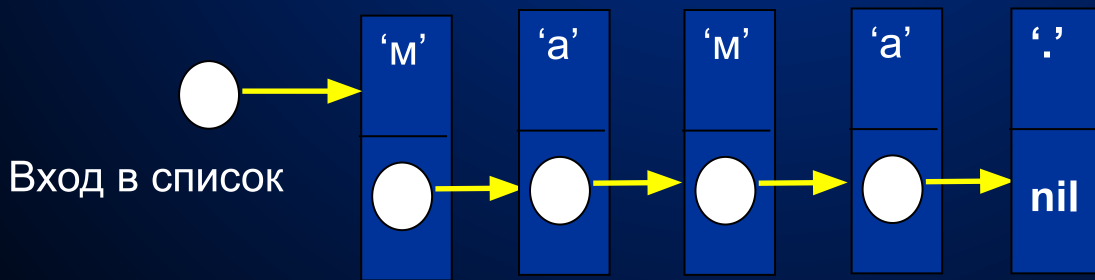
Формирование списка

Рассмотрим структуру данных. Для примера в списке будем хранить литеры (буквы), причем признаком конца ввода выберем символ ‘.’, это условность, формально можно взять любой другой символ.

```
Type    ref=^Node;{указатель на звено}
    Node=record{звено}
        Next:Ref; {указатель на следующее звено}
        Lit:char {информация звена (символ)}
    End;
```

Каждое звено хранит один информационный символ и указатель на следующее звено.

Сам список будет выглядеть так:



```
Procedure CreateList(var inz:ref);
Var tz:ref; a:char;
Begin
  New(inz); tz:=Inz;
  Read(a);tz^.Lit:=a;
 {дополнительный указатель tz потребовался, так как inz
   должен хранить адрес первого звена}
  While a<>'.' Do
    Begin
       New(tz^.next);{создадим новое звено}
       Tz:=tz^.Next;{перейдем к следующему звену}
       Read(a); tz^.lit:=a
    End;
  Tz^.Next:=nil; ;{ пометим конец списка}
  Readln ;{ удалим enter из буфера клавиатуры}
End;
```

tz

Inz

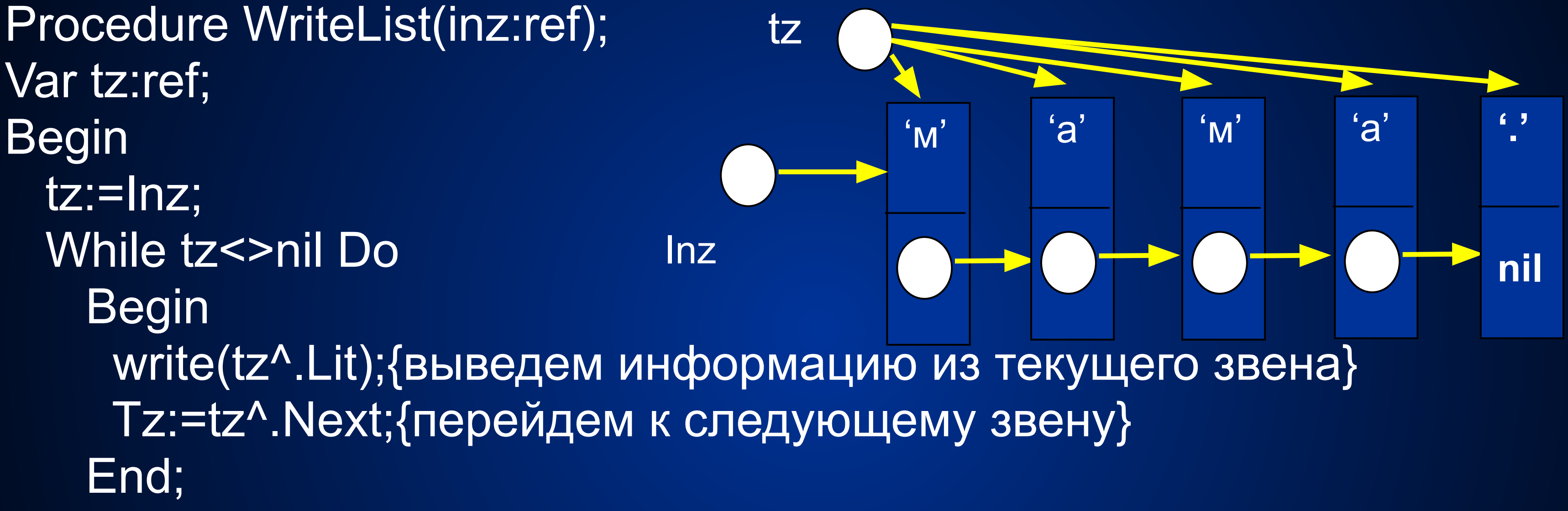
'm' 'a' 'm' 'a' '.'

nil

•Вывод списка.

Встаем на начало списка (inz), двигаемся по нему, переходя к следующему элементу (поле Next) и выводим информацию (поле lit).

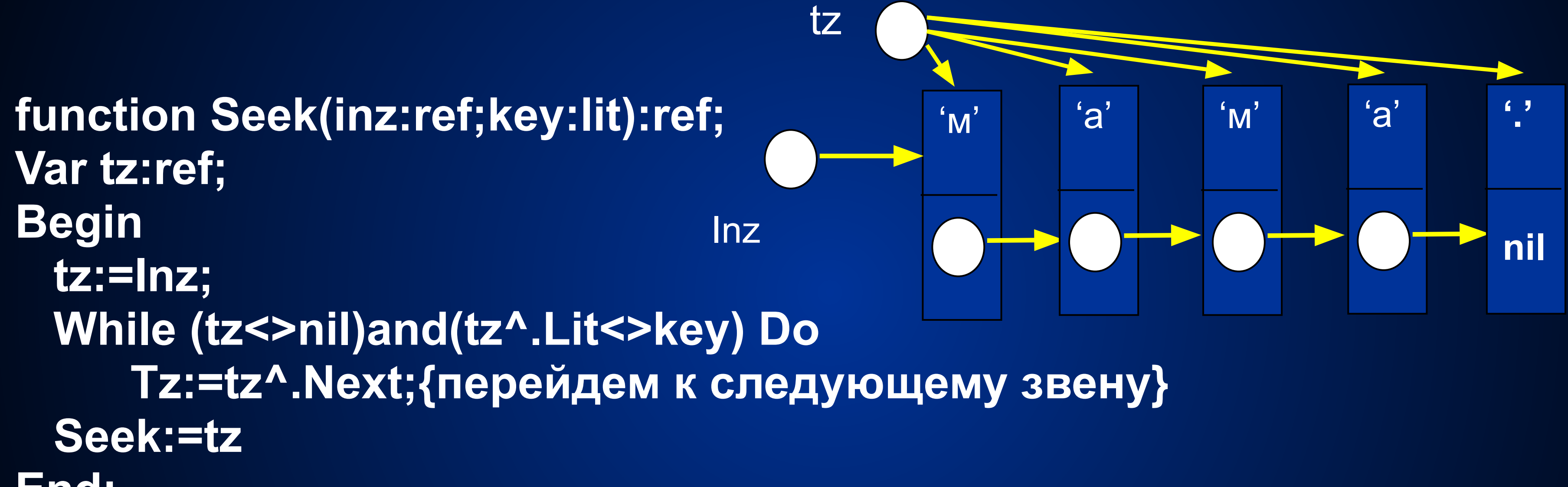
```
Procedure WriteList(inz:ref);
Var tz:ref;
Begin
   tz:=Inz;
   While tz<>nil Do
      Begin
        write(tz^.Lit);{выведем информацию из текущего звена}
        Tz:=tz^.Next;{перейдем к следующему звену}
      End;
End;
```



Признаком конца списка мы используем не символ ‘.’, а пустую ссылку nil. Это позволяет оторваться от конкретного списка и создать универсальную процедуру вывода списка, которая не зависит от того, что в нем содержится.

•Поиск элемента в списке.

Одна из важнейших операций в программировании. Встаем на начало списка (inz), двигаемся по нему, переходя к следующему элементу (поле Next), пока не найдем искомый элемент или пока не дойдем до конца списка (nil).




```
function Seek(inz:ref;key:lit):ref;
Var tz:ref;
Begin
   tz:=Inz;
   While (tz<>nil)and(tz^.Lit<>key) Do
         Tz:=tz^.Next;{перейдем к следующему звену}
   Seek:=tz
End;
```

Мы имеем два сравнения на каждый элемент, поэтому худшая скорость будет O(2n), а средняя – O(2n/2)=O(n). Обратите внимание, что в цикле while используется логическая операции and, а не or! Это условие продолжения, а не окончания!

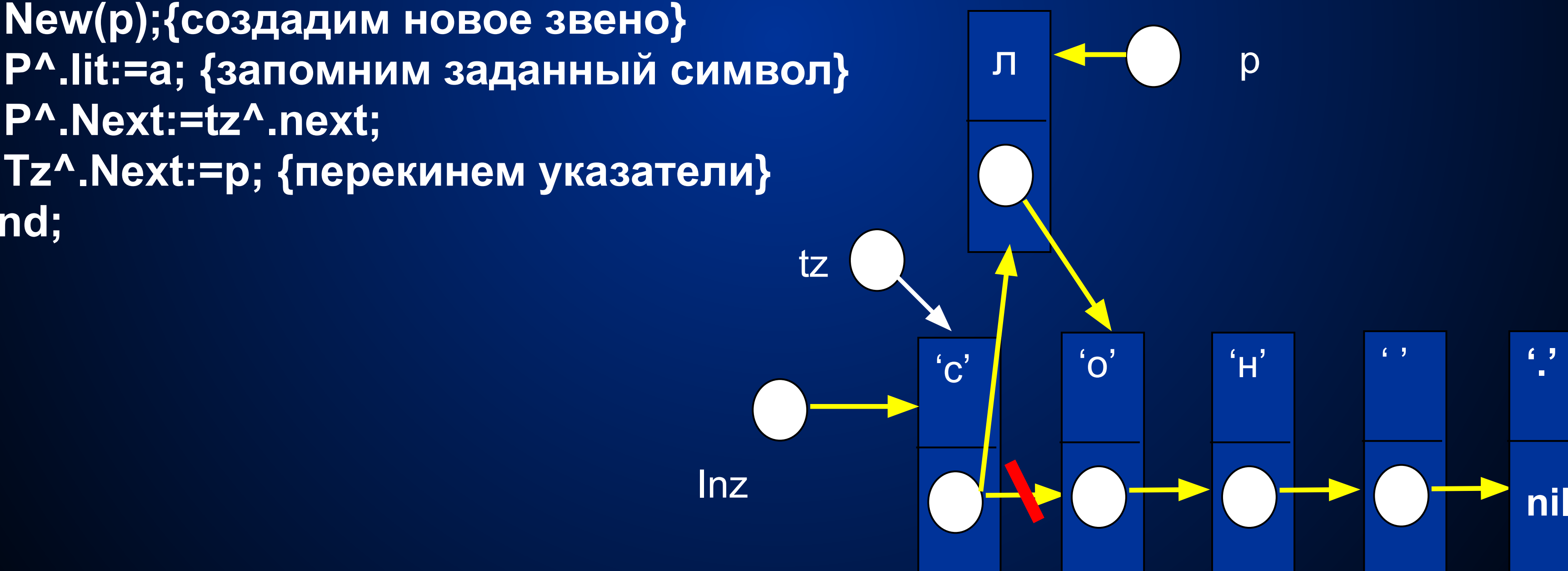
•Вставка элемента в список.

В отличие от статического списка, в котором для вставки элемента необходимо сместить все правые элементы на одну ячейку вправо, что требует в среднем порядка O(n/2) операций копирования, в динамическом – вставка осуществляется за O(1). Существуют два варианта вставки:

а) вставка после текущего.

Пусть tz указывает на некоторый элемент списка, необходимо за ним разместить новое звено с заданным символом.

```
Procedure InsAfter(tz:ref;a:char);
Var p:ref;
Begin
   New(p);{создадим новое звено}
   P^.lit:=a; {запомним заданный символ}
   P^.Next:=tz^.next;
   Tz^.Next:=p; {перекинем указатели}
End;
```



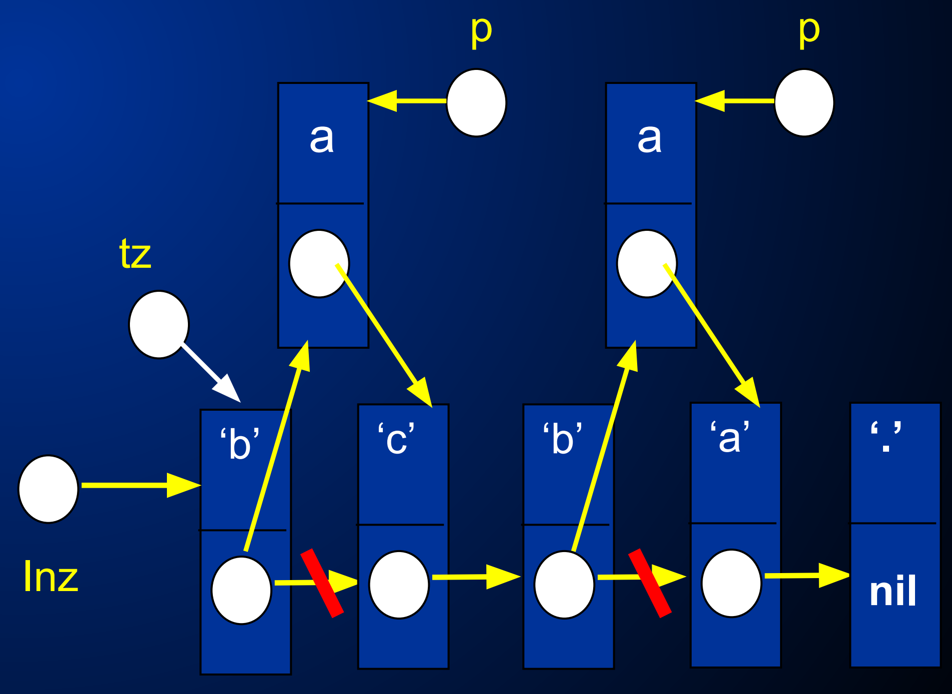
б) вставка перед текущим.

Пусть tz указывает на некоторый элемент списка, перед которым необходимо разместить новое звено с заданным символом. Кажется, что это невозможно, так как мы не имеем доступа к предыдущим элементам и, следовательно, не сможем изменить связи в цепочке. Однако, если нельзя, но очень надо, то можно! ☺ Для этого вставим новое звено после текущего, а значение текущего звена скопируем во вновь созданное. В результате вставляемый символ окажется перед символом, который был текущим, то есть задача выполнена!

```
Procedure InsBefore(tz:ref;a:char);
Var p:ref;
Begin
   New(p);{создадим новое звено}
   P^.lit:=tz^.lit;{запомним текущий символ}
   P^.Next:=tz^.next;
   Tz^.Next:=p; {перекинем указатели}
   Tz^.Lit:=a
End;
```

Дан линейный динамический список, вставить в нем букву ‘a’ после буквы ‘b’.

```
Procedure InsAfter(tz:ref;a:char);
Var p:ref;
Begin
   New(p);
   P^.lit:=a;
   P^.Next:=tz^.next;
   Tz^.Next:=p;
End;
Procedure InsA(var inz:Ref;
               a,b:char);
Vat tz:Ref;
Begin
  tz:=Inz;
  While tz<>nil do begin
    if tz^.Lit=b
    then InsAfter(tz,a);
    tz:=tz^.Next
  end
End;
```



Удаление элемента из списка.

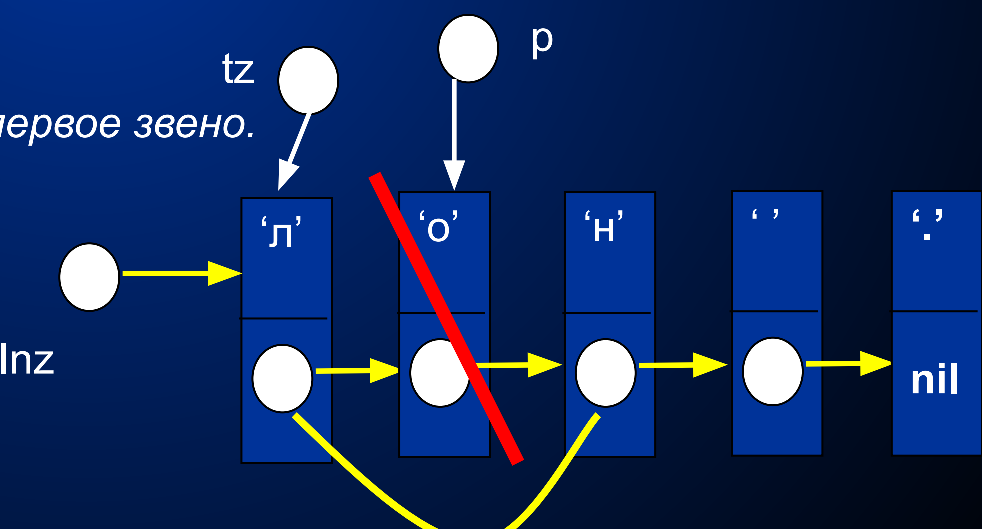
В отличие от статического списка, в котором для удаление элемента необходимо сместить все правые элементы на одну ячейку влево, что требует в среднем порядка O (n/2) операций копирования, в динамическом – удаление осуществляется за O(1). Существуют два варианта удаления:

а) удаление после текущего.

Пусть tz указывает на некоторый элемент списка, необходимо удалить следующий за ним элемент. Воспользуемся дополнительным указателем p. Установим его на следующее звено, перекинем связь Next минуя удаляемый элемент. **‘с’‘л’‘о’‘н’‘.’**

```
Procedure DelAfter(tz:ref);
Var p:ref;
Begin
   P:=tz^.Next;{p:= след.звено}
   If p<> nil;{если след. есть, то}
   Then begin
              tz^.Next:=p^.Next;
              Dispose(p){перекинем связи и удалим объект}
            end
End;
```

Недостаток метода: нельзя удалить первое звено.



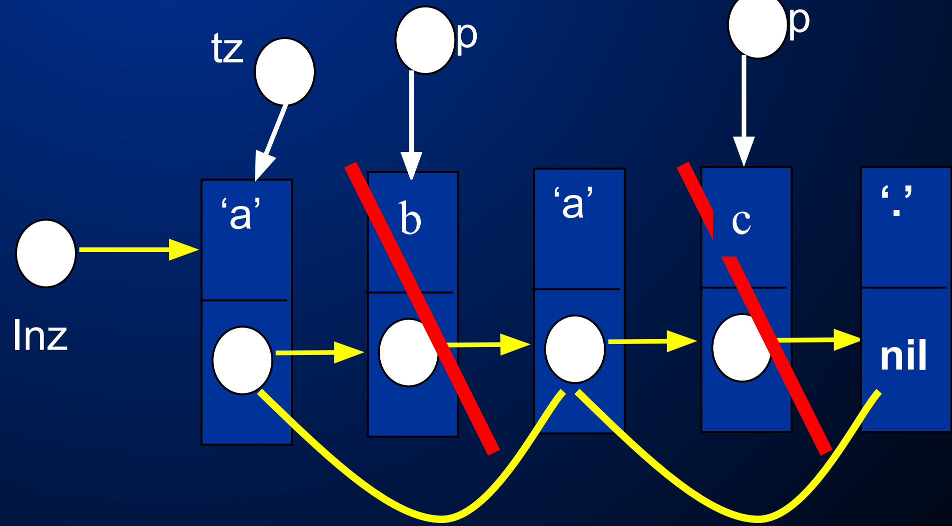
б) удаление текущего.

Пусть tz указывает на некоторый элемент списка, который нам необходимо удалить. Опять кажется, что это невозможно, так как мы не имеем доступа к предыдущим элементам и, следовательно, не сможем изменить связи в цепочке. Однако, воспользуемся предыдущей идеей – удалим следующее звено после текущего, предварительно скопировав из него информацию в текущее.

```
Procedure DelCur(tz:ref);
Var p:ref;
Begin
    P:=tz^.Next;{p:= след.звено}
    If p<> nil;{если след. есть, то}
    Then begin
            tz^.Next:=p^.Next;
            tz^.Lit:=p^.lit;
            Dispose(p){перекинем связи и удалим объект}
  end
End;
```

Дан линейный динамический список, удалить в нем все буквы ‘а’.

```
Procedure DelCur(tz:ref);
Var p:ref;
Begin
   P:=tz^.Next;
   If p<> nil;
   Then begin
         tz^.Next:=p^.Next;
         tz^.Lit:=p^.lit;
         Dispose(p)
       end
End;
Procedure DelA(var inz:Ref;
               a:char);
Vat tz:Ref;
Begin
  tz:=Inz;
  While tz<>nil do
    if tz^.Lit=a
    then DelCur(tz)
    else tz:=tz^.Next
End;
```

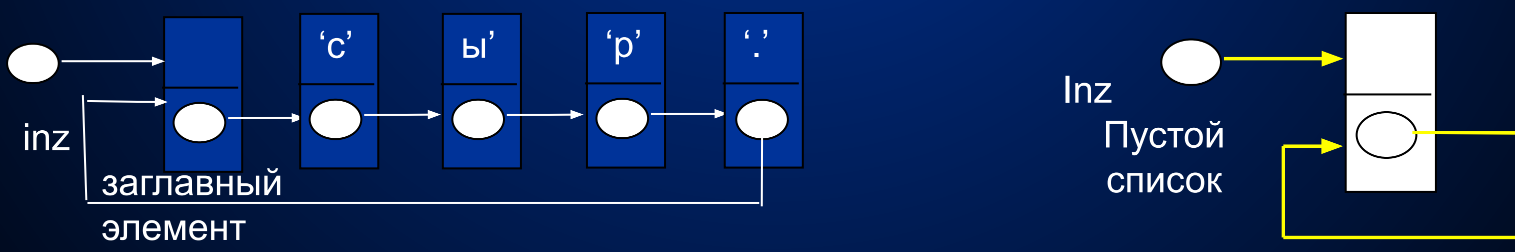


Динамический линейный однонаправленный кольцевой список с заглавным элементом

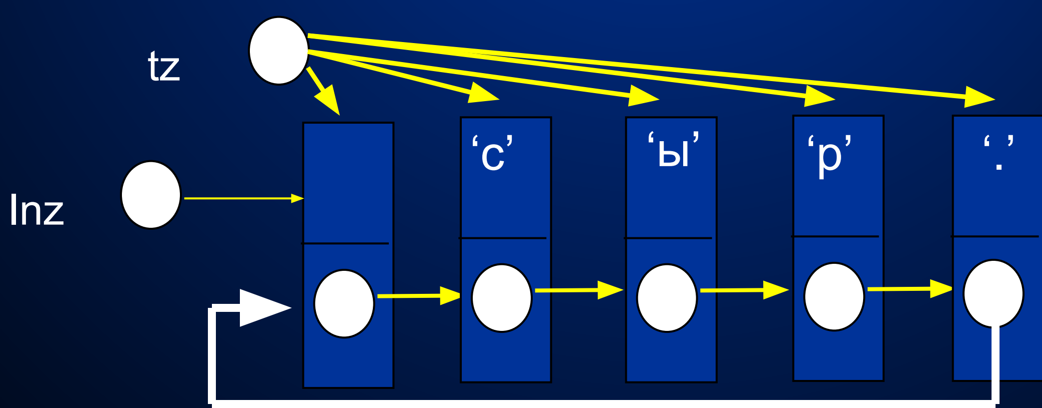
Основные проблемы классического динамического линейного списка:

- наличие нескольких частных случаев списка: пустой, один элемент, несколько элементов. Каждый случай требует отдельного рассмотрения;
- при удалении последнего оставшегося звена и получении пустого списка требуется изменение входного указателя inz, а это потребует усложнение процедур удаления элементов;
- трудности с удалением крайних элементов списка;
- гигантские проблемы с пустым списком, вставкой, удалением и так далее.

От большинства этих недостатков избавлен кольцевой список с заглавным элементом. Заглавный элемент не содержит информации, его задача избавиться от частного случая – пустой список, в котором inz=nil. Кольцо – позволяет замкнуть последнюю связь на заглавный элемент, что в принципе позволяет добраться до любого элемента.



```
Procedure CreateRing(var inz:ref);
Var tz:ref; a:char;
Begin
      {создадим заглавное звено}
      New(inz); tz:=Inz;
      repeat
        New(tz^.next);{создадим новое звено}
        Tz:=tz^.Next;{перейдем к следующему звену}
         Read(a); tz^.lit:=a
       Until a=’.’
      Tz^.Next:=inz; ;{замкнем конец списка на его начало}
      Readln ;{удалим enter из буфера клавиатуры}
End;
```



Вывод списка

Встаем на начало списка (inz), двигаемся по нему, переходя к следующему элементу (поле Next) и выводим информацию (поле lit).

```
Procedure WriteList(inz:ref);
Var tz:ref;
Begin
     tz:=Inz^.Next;{пропустим заглавный элемент}
    While tz<>Inz Do
      Begin
        write(tz^.Lit);{выведем информацию из текущего звена}
       Tz:=tz^.Next;{перейдем к следующему звену}
    End;
End;
```

Признаком конца списка мы используем не символ '.'и пустую ссылку nil, а входной указатель inz.

Поиск элемента в кольцевом списке с заглавным элементом.

Наличие кольца и заглавного элемента позволит нам увеличить скорость поиска в 2 раза. Для этого избавимся от проверки на достижении конца списка, следовательно, процесс остановится только при нахождении ключа. А что делать, если его нет в списке? Разместим искомый в заглавном элементе – он всё равно пустой! Встаем на начало списка (inz), двигаемся по нему, переходя к следующему элементу (поле Next), пока не найдем искомый элемент. Если мы его нашли в заглавном, то в списке искомого элемента не было.

```
function Seek(inz:ref;key:lit):ref;
Var tz:ref;
Begin
    tz:=Inz^.Next; Inz^.Lit:=key;{поместим искомый в заглавный}
    While (tz^.Lit<>key) Do
        Tz:=tz^.Next;{перейдем к следующему звену}
    If tz<>inz then Seek:=tz else Seek:=nil
End;
```

Мы имеем одно сравнение на каждый элемент, поэтому худшая скорость будет O(n), а средняя – O(n/2).

Остальные операции реализуются аналогично.

Динамический линейный двунаправленный список

Используется в тех случаях, когда необходимо просто и быстро путешествовать в обоих направлениях, например, текстовый редактор – курсор может перемещаться к предыдущей и последующей строкам. Отличие данного списка состоит в наличии двух связей Left и Right.

```
Type Ref=^Node;
     Node=record
          Left,Right:Ref;
          Lit:char;
       End;
Procedure CreateDouble(var inz:ref);
Var tz:ref; a:char;
Begin
   New(inz);tz:=inz; tz^.Left:=nil;
   Read(a); tz^.lit:=a;
   While a<>’.’ Do
    Begin
      New(tz^.Right); {создать новое звено справа}
      tz^.Right^.Left:=tz; {создать «левую» связь от нового звена}
      tz:=tz^.Right; {перейти к следующему звену}
      Read(a); tz^.lit:=a; {считать и заполнить информацию}
    End;
  Tz^.Right:=nil;readln
End;
```

Остальные операции реализуются аналогично.

inz

‘м’ nil

‘а’

‘м’

‘а’

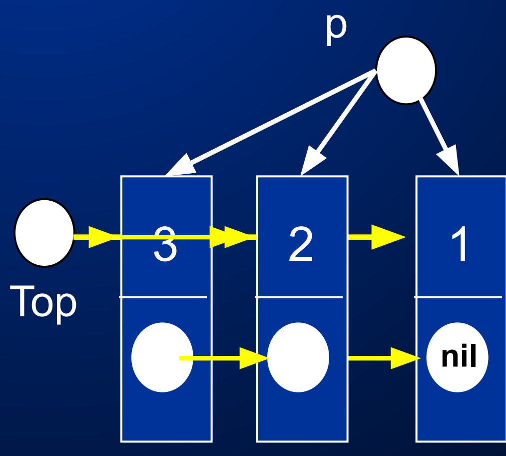
‘.’ nil

Динамический стек

Стек – это очередь особого вида, помещение элементов в которую и их извлечение осуществляется с одного конца. У стека есть указатель top – вершина стека. Стек на основе массива имеет один недостаток – он ограничен в размерах и всегда есть риск его переполнения. Динамический стек растет по мере необходимости, поэтому переполниться не может. Зададимся вопросом: какой вид динамического списка мы будем использовать? Однонаправленный или двунаправленный? Подумайте!

```
Procedure Push(x:тип);
Var p:Ref;
Begin
  new(p);  p^.key:=x;
  p^.Next:=top;
  top:=p
End;
```

```
function Pop:тип;
Var p:Ref;
Begin
 Pop:=top^.key;
 p:=top; top:=p^.next;
 Dispose(p);
End;
```



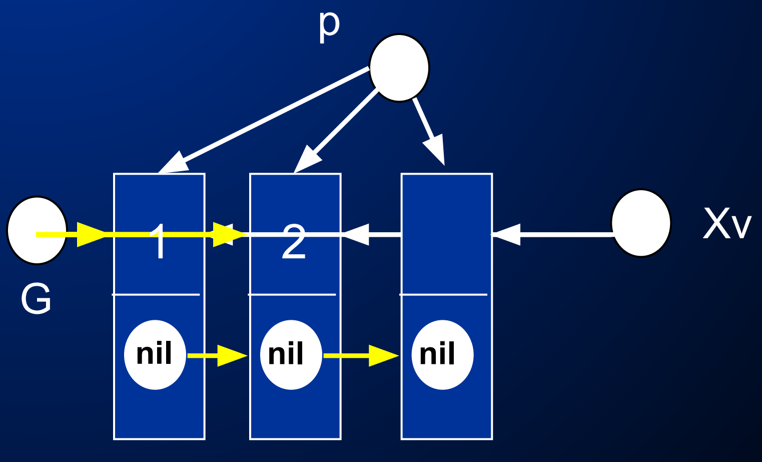
Динамическая очередь

Очередь – это список особого вида, помещение элементов в который осуществляется с одного конца (хвоста), а их извлечение – со стороны головы. У очереди есть два указателя G – голова очереди и Xv - хвост. Зададимся вопросом: какой вид динамического списка мы будем использовать? Однонаправленный или двунаправленный? Подумайте!

Воспользуемся однонаправленным списком с заглавным элементом.

```
Procedure PutQ(x:тип);
Var p:Ref;
Begin
  new(p); Xv^.key:=x;
  Xv^.Next:=p;
  P^.Next:=nil; Xv:=p
End;
```

```
function GetQ:тип;
Var p:Ref;
Begin
  GetQ:=G^.key;
  p:=G; G:=G^.Next;
  Dispose(p)
End;
```

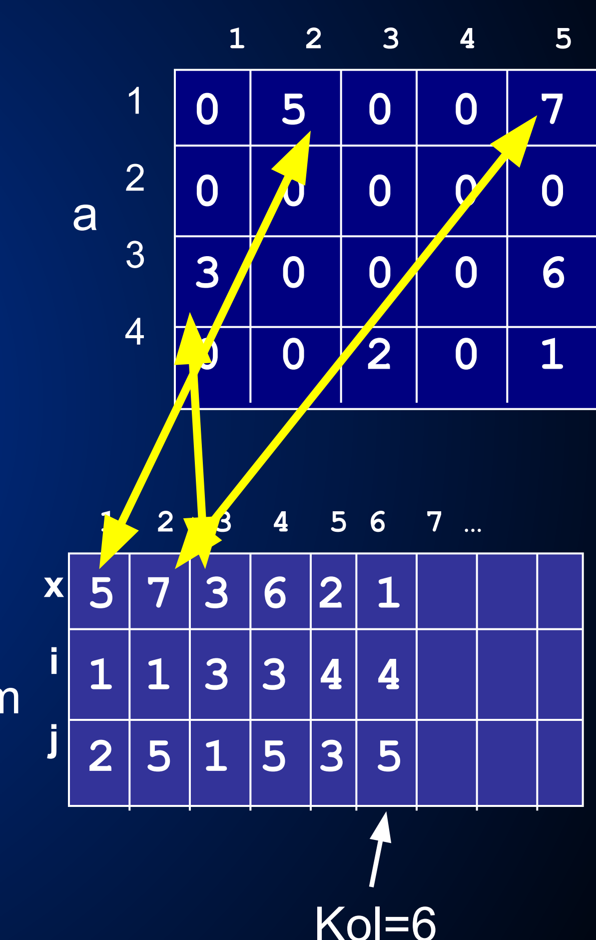


Разреженные матрицы

Разреженной матрицей называется матрица, в которой присутствует подавляющее большинство 0. Хранение такой матрицы в виде массива крайне неэффективно, поэтому возникает идея – хранить не все элементы, а только координаты и значения ненулевых элементов. Можно предложить несколько вариантов:

1) хранение координат и значений ненулевых элементов.

```
Type elem= record
            i, j: integer;
            x:real;
         End;
tMass=array[1..max] of elem;
Var m: tMass;
     Kol: integer; {количество не нулевых элементов}
Function a(I,j:integer):real;
Var k:integer;
Begin
A:=0;
For k:=1 to Kol do
   If (m[k].i=i)and(m[k].j=j)   Then a:=m[k].x
End;
```



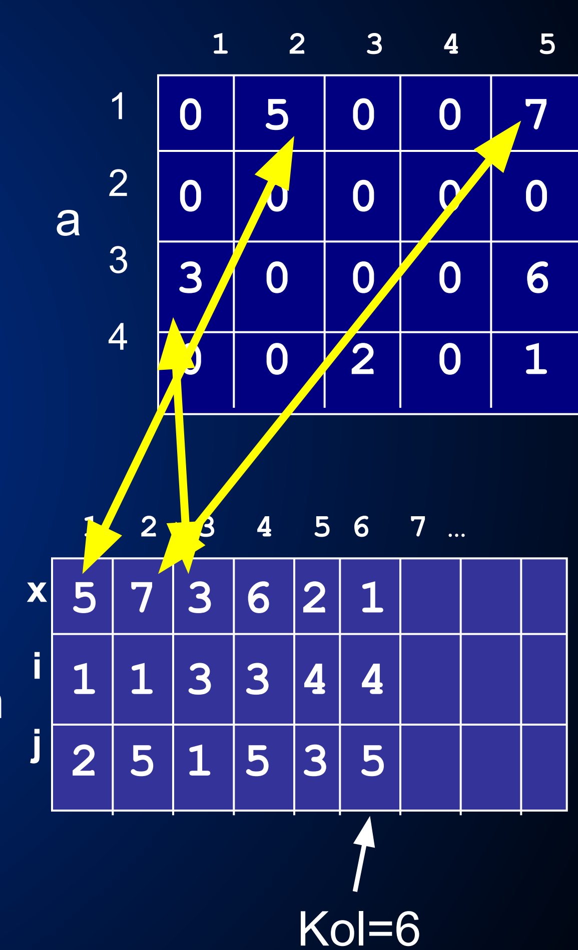
Решение не самое эффективное по скорости, т.к. даже обнаружив искомый элемент, цикл **for** продолжит свою работу. В результате средняя скорость равна худшей и равна O(3*Kol). Цикл выполняется Kol раз, в нем 3 проверки (две в условии **If**, одна в заголовке цикла **for**). Можно несколько улучшить решение, заменив цикл **for** на **While**.

```
Function a(i, j:integer):real;
Var k:integer;
Begin
  k:=1;
  While (not ((m[k].i=i)and(m[k].j=j))and(k< Kol) do
      inc(k)
  if (m[k].i=i)and(m[k].j=j)   then a:=m[k].x else a:=0
End;
```

Средняя скорость увеличилась в два раза, худшая не изменилась.

Добавление или удаление элемента в матрицу достаточно простое:

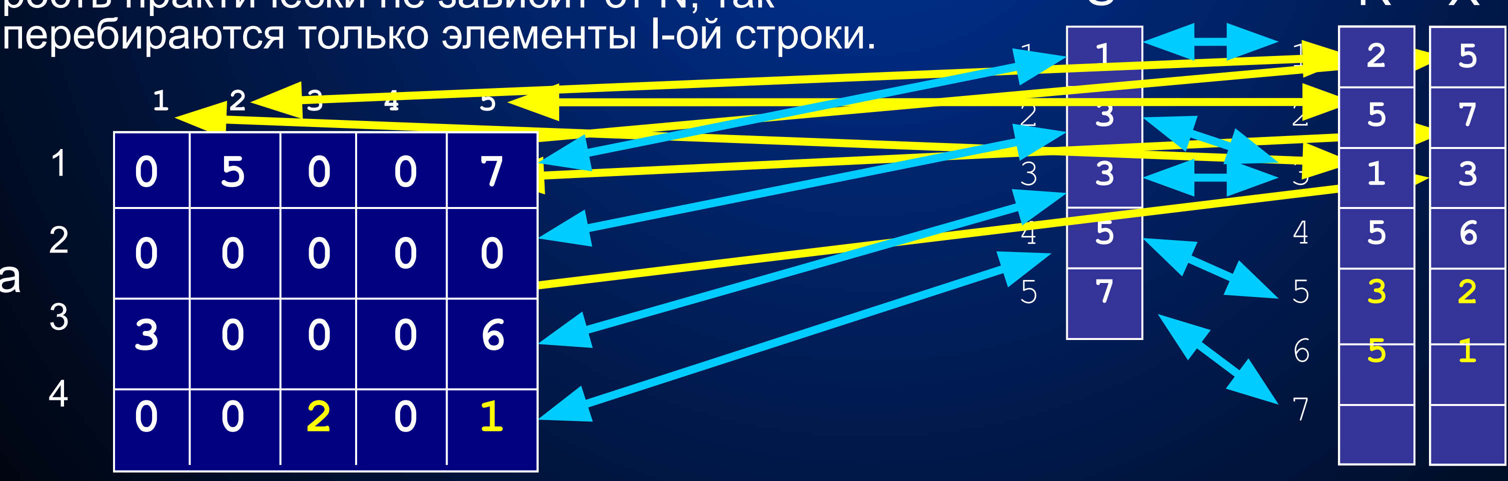
```
Function addEl(i, j:integer; el:real):real;
Var k:integer;
Begin
  k:=1;
  While (not ((m[k].i=i)and(m[k].j=j))and(k< Kol) do
      inc(k)
  if (m[k].i=i)and(m[k].j=j)   then m[k].x:=El
  else begin
         inc(Kol); m[Kol].x:=El; m[Kol].i:=i; m[Kol].j:=j
       end
End;
```

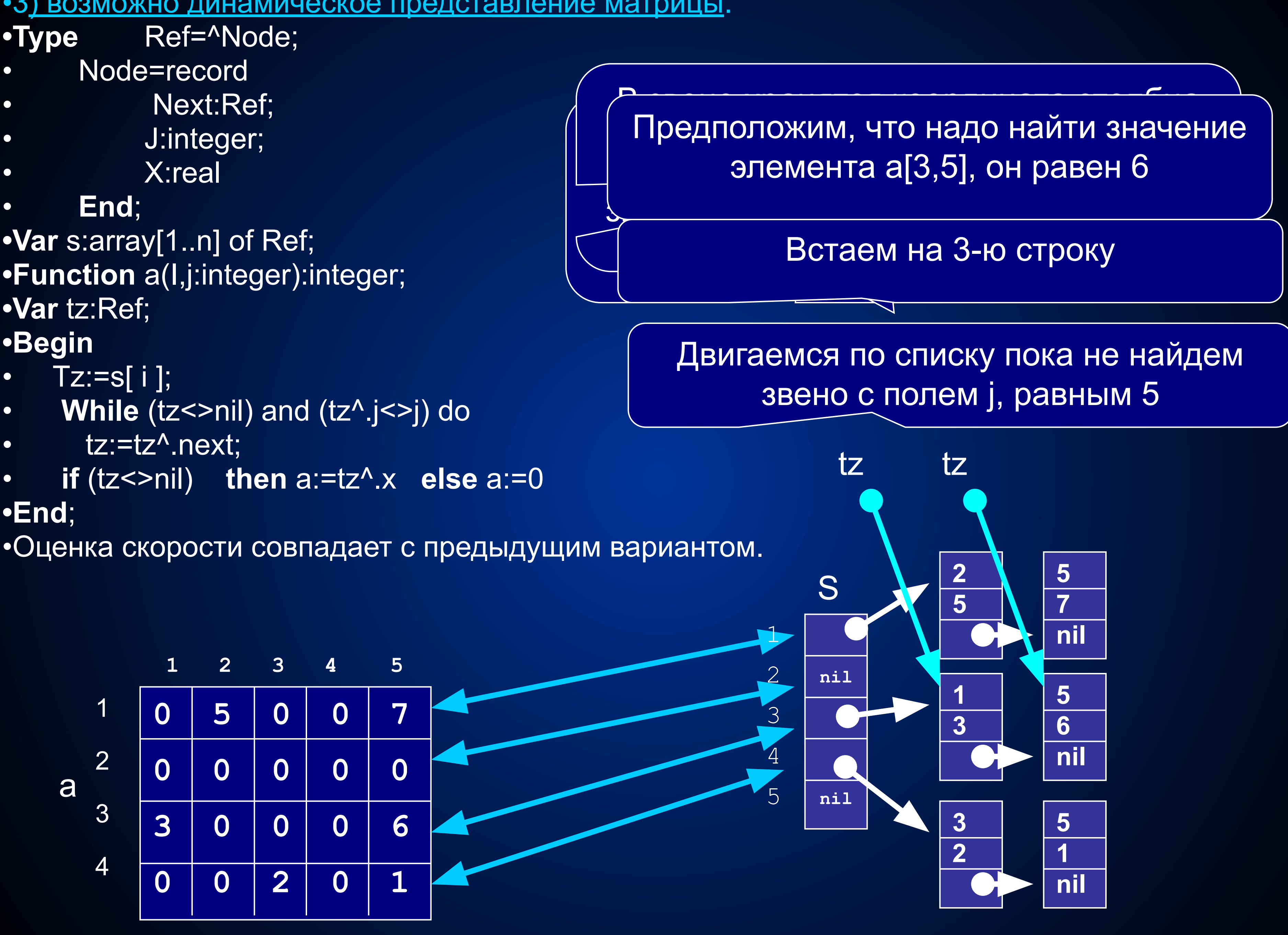


2) Специальная организация хранения координат.

- В массиве X хранятся значения ненулевых элементов.
- Массив R хранит номера столбцов ненулевых элементов,
- массив S хранит ссылки на массив R. Каждый элемент S соответствует строке исходной матрицы A. S[1]=1 – это означает, что ненулевые элементы первой строки матрицы A в массиве R начинаются с 1 элемента, S[2]=3 – это означает, что ненулевые элементы второй строки начинаются с 3 элемента, следовательно, 1 строка заканчивается на один элемент раньше.
- 1ая строка в массиве R с S[1] по s[2]-1
- 2ая строка в массиве R с S[2] по s[3]-1
- 3ая строка в массиве R с S[3] по s[4]-1
- iая строка в массиве R с S[i] по s[i+1]-1
- **Function** a(I,j:integer):integer;
- **Var** k:integer;
- **Begin** A:=0;
- **For** k:=s[i] to s[i+1]-1 do
- **If** (r[k]= j) **Then** a:=x[k]
- **End**;
- Скорость практически не зависит от N, так как перебираются только элементы I-ой строки.

Попробуйте реализовать подпрограмму, которая вернет значение элемента матрицы A по его координатам i, j

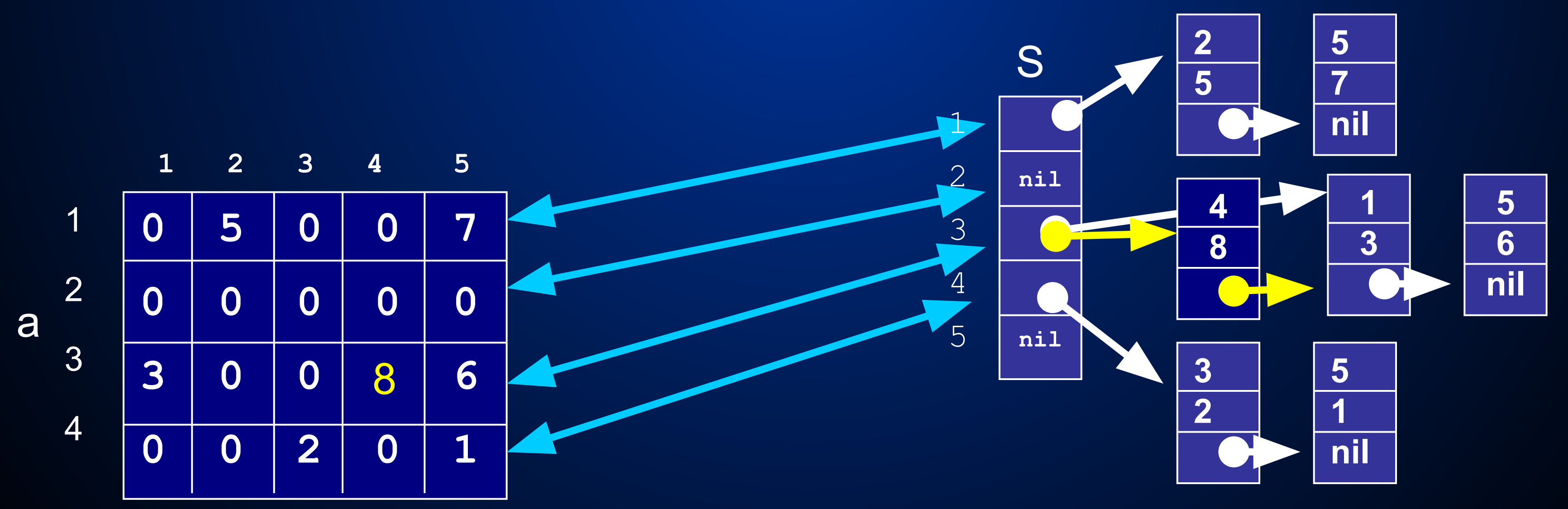



•3) возможно динамическое представление матрицы.
•Type Ref=^Node;
• Node=record
• Next:Ref;
• J:integer;
• X:real
• End;
•Var s:array[1..n] of Ref;
•Function a(I,j:integer):integer;
•Var tz:Ref;
•Begin
• Tz:=s[i];
• While (tz<>nil) and (tz^.j<>j) do
• tz:=tz^.next;
• if (tz<>nil) then a:=tz^.x else a:=0
•End;
•Оценка скорости совпадает с предыдущим вариантом.
Предположим, что надо найти значение элемента a[3,5], он равен 6
Встаем на 3-ю строку
Двигаемся по списку пока не найдем звено с полем j, равным 5
tz
tz
S
a
1 2 3 4 5
1 2 3 4
0 5 0 0 7
0 0 0 0 0
3 0 0 0 6
0 0 2 0 1
1 2 3 4 5
nil
nil
2 5
5 7 nil
1 3
5 6 nil
3 2
5 1 nil

- **Procedure** AddEl(i, j:integer; El:real);
- **Var** tz:Ref;
- **Begin**
-     new(Tz); tz^.j:=j; tz^.x:=El;
-     Tz^.Next:=s[i]; s[i]:=tz
- **End**;
- Оценка скорости O(1).

Создадим новое звено, заполним его поля, вставим его в НАЧАЛО списка строки с номером i

Добавим в матицу новый элемент 8 с координатами 3, 4



Конечные автоматы

Конечный автомат представляет собой особый способ описания алгоритма, который характеризуется набором из 5 элементов: К - конечный (ограниченный) набор состояний автомата, А - конечный алфавит, S - начальное состояние автомата, F - множество заключительных состояний автомата, D – отображение: откуда/куда.

Говорят, что конечный автомат допускает цепочку, если при ее анализе, начиная с начального состояния, функция D определена на каждом шаге и последнее состояние является заключительным.

Конечный автомат не допускает входную цепочку, если:

1) на каком-то шаге не определена функция D;

2) последнее состояние не является заключительным.

Пример. Конечный автомат, распознающий идентификатор.

K={0,1} множество состояний

A={' ', 'A'..'Z','a'..'z','0'..'9'} алфавит

S=0 начальное состояние

F={1} конечное состояние

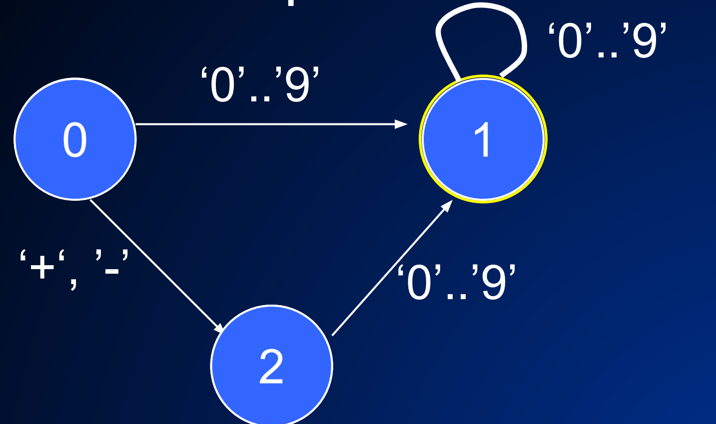
Конечный автомат можно задавать не только таблицей, но и диаграммой переходов.

входные символы

D:

	пробел	буква	цифра
0	0	1	Error
1	Error	1	1

- Описать конечный автомат, распознающий запись целого числа в десятичном виде.



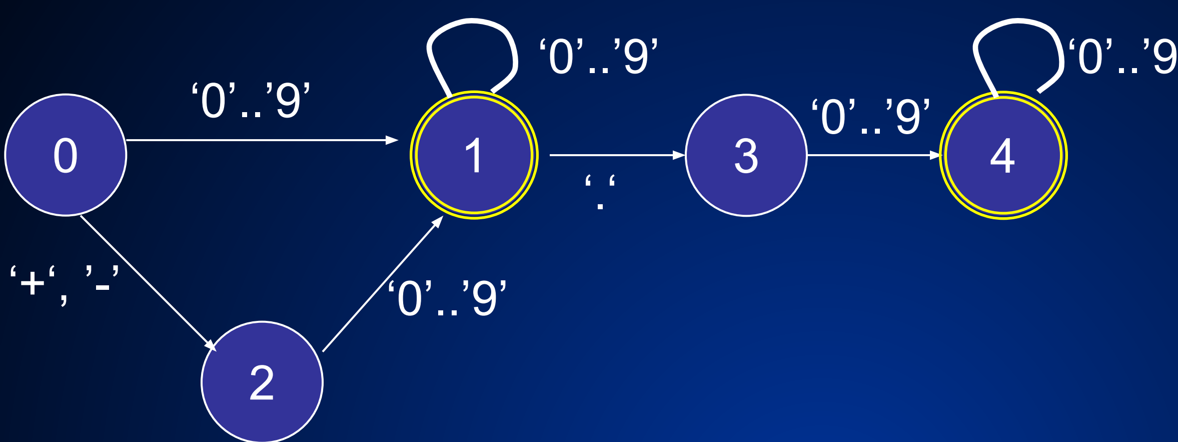
В чем проблема данного автомата?

Посмотрим, как работает автомат для числа s=‘-15’

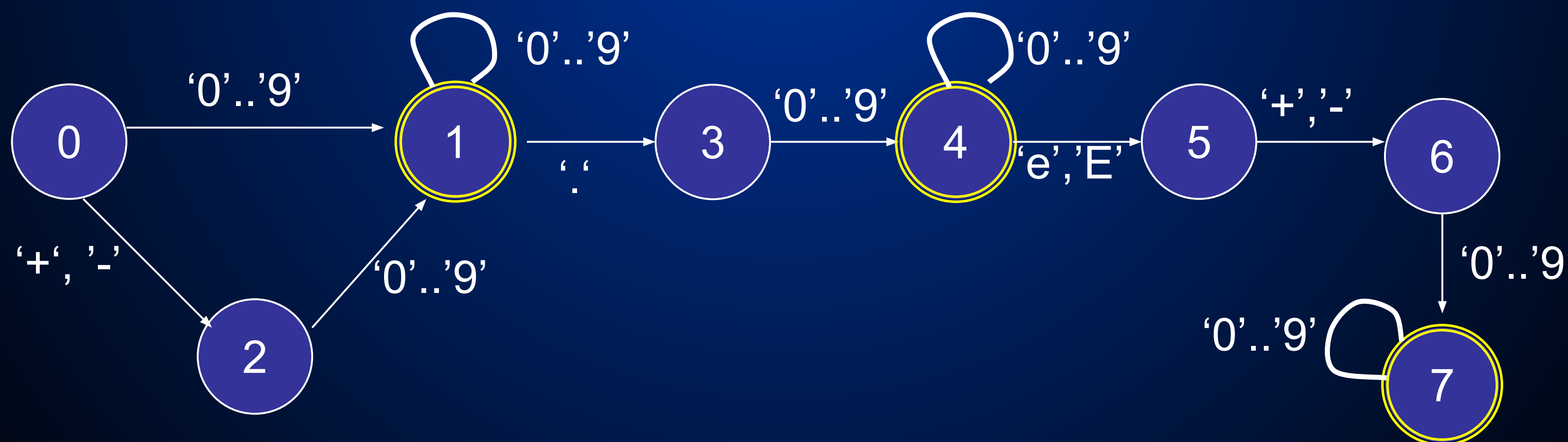
```
Function IntA(s:string):boolean;
Var i, q :integer; f:boolean;
Begin
   q:=0; i:=0; f:=true;
   repeat
    inc(i);
    case q of
     0: if s[i] in ['0'..'9']
         then q:=1
         else if s[i] in ['+','-']
              then q:=2 else f:=false;
           2: if s[i] in ['0'..'9']
               then q:=1 else f:=false;
           1: if not (s[i] in ['0'..'9'])
               then f:=false
         end
   until (not f)or(i>=length(s));
   IntA:=f and (q=1)
End;
```

S	-15
q	1

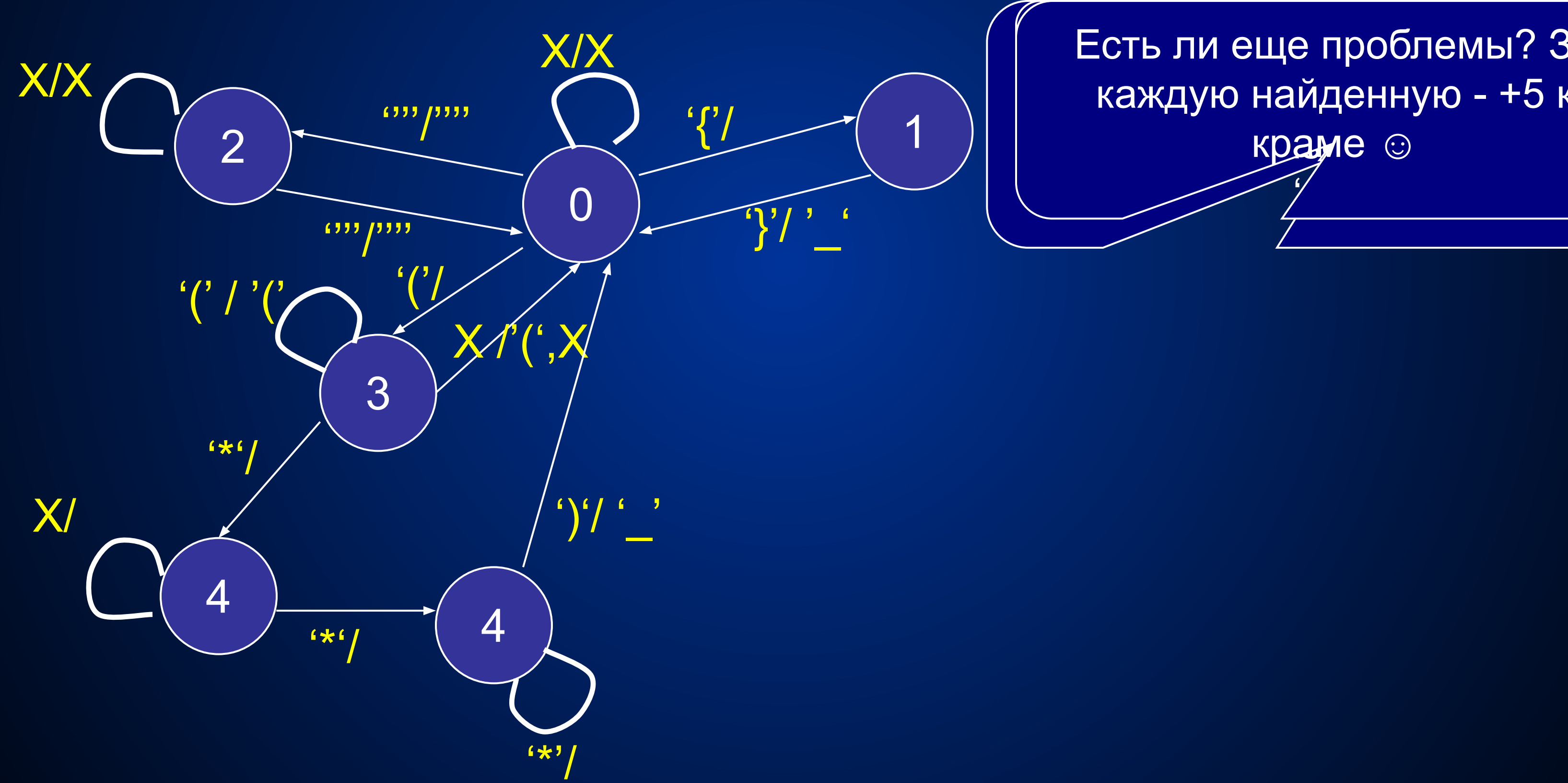
- Описать конечный автомат, распознающий запись дробного числа в десятичном виде.



- Описать конечный автомат, распознающий запись дробного числа типа real.

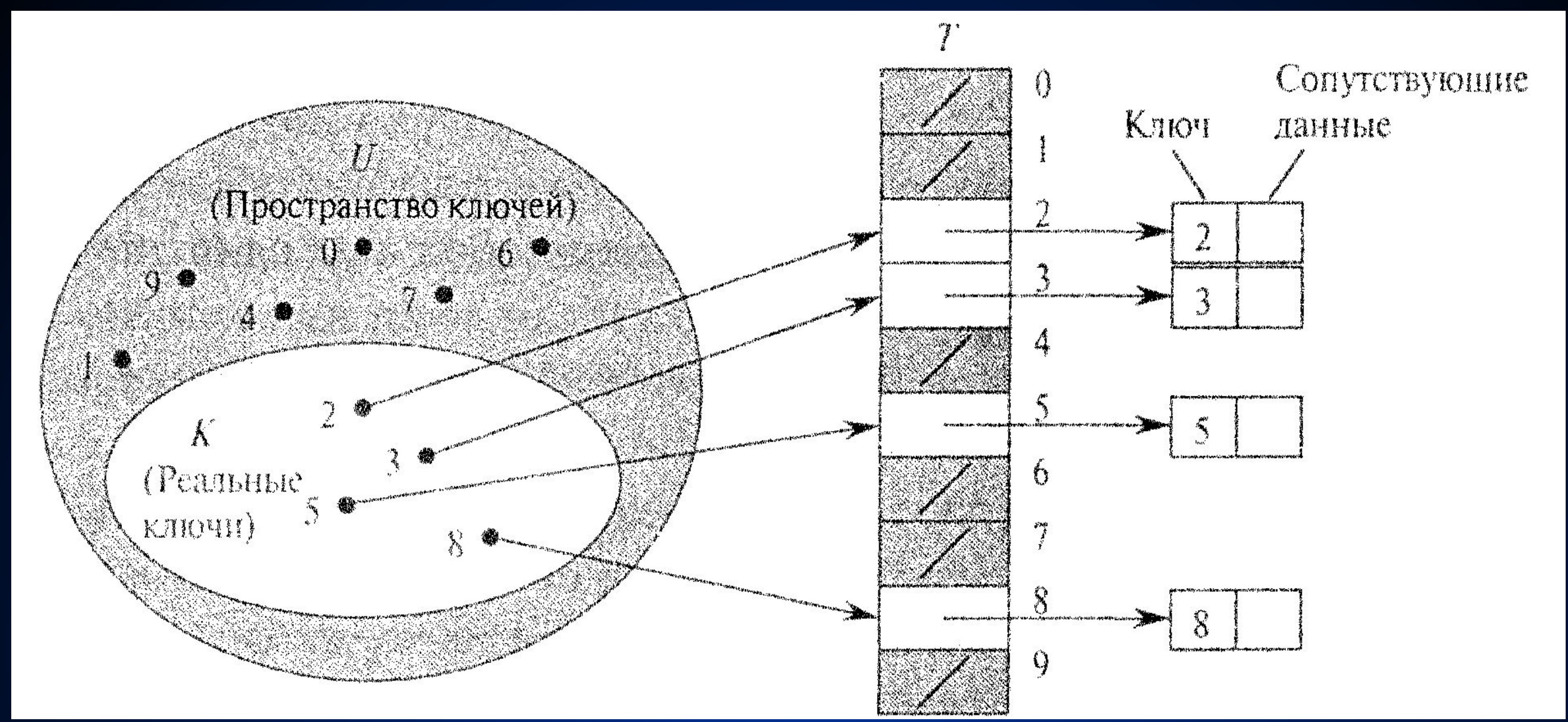


- Дана работоспособная программа на Паскале. Необходимо удалить из нее комментарии так, чтобы сохранить ее функции:
- А) разрешается использовать комментарии только одного типа. Все, что заключено в фигурные скобки ‘{‘, ‘}’ считается комментарием. Комментарии не могут быть вложены друг в друга.
- Б) разрешается использовать комментарии только двух типов. Все, что заключено в фигурные скобки ‘{‘, ‘}’ или (*. *) считается комментарием. Комментарии разного типа могут быть вложены друг в друга.



ХЕШ- таблицы с прямой адресацией

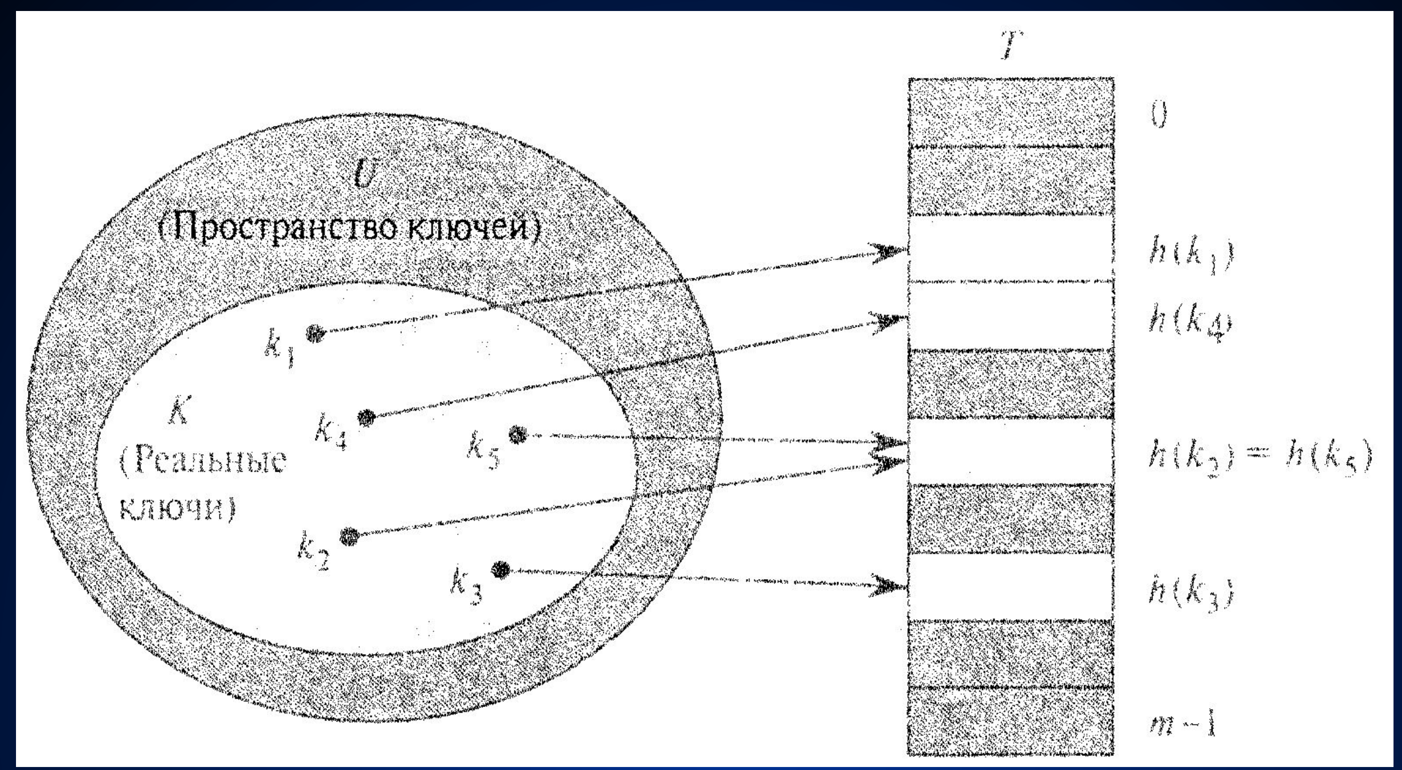
- Прямая адресация представляет собой простейшую технологию, которая хорошо работает для небольших множеств ключей. Предположим, что приложению требуется динамическое множество, каждый элемент которого имеет ключ из множества U = {0,1,..., m — 1}, где m не слишком велико. Кроме того, предполагается, что никакие два элемента не имеют одинаковых ключей. Для представления динамического множества мы используем массив, или таблицу с прямой адресацией, который обозначим как T [0..m — 1], каждая по-позиция, или ячейка (position, slot), которого соответствует ключу из пространства ключей U. Ячейка K указывает на элемент множества с ключом к. Если множество не содержит элемента с ключом K, то T [к] = NIL. На рисунке каждый ключ из пространства U = {0,1,..., 9} соответствует индексу таблицы. Множество реальных ключей К = {2,3,5,8} определяет ячейки таблицы, которые содержат указатели на элементы. Остальные ячейки (закрашенные темным цветом) содержат значение nil.



- Direct_Address_Search(T, к) return T[k]
- Direct_Address_Insert(T, x) T[кey[x]] <- x
- Direct_Address_Delete(T, x) T[кey[x]]<- NIL
- Недостаток прямой адресации очевиден: если пространство ключей U велико, хранение таблицы T размером |U| непрактично, а то и вовсе невозможно — в зависимости от количества доступной памяти и размера пространства ключей. Кроме того, множество К реально сохраненных ключей может быть мало по сравнению с пространством ключей U, а в этом случае память, выделенная для таблицы T, в основном расходуется напрасно.

Хеш-таблицы

- Когда множество К хранящихся в словаре ключей гораздо меньше пространства возможных ключей U, хеш-таблица требует существенно меньше места, чем таблица с прямой адресацией. Точнее говоря, требования к памяти могут быть снижены до Θ(|K|), при этом время поиска элемента в хеш-таблице остается равным O(1). Надо только заметить, что это граница среднего времени поиска, в то время как в случае таблицы с прямой адресацией эта граница справедлива для наихудшего случая. В случае прямой адресации элемент с ключом к хранится в ячейке к. При хешировании этот элемент хранится в ячейке h (к), т.е. мы используем хеш-функцию h для вычисления ячейки для данного ключа к. Функция h отображает пространство ключей U на ячейки хеш-таблицы T [0..m — 1]: h:U-> {0,1,...,m-1}. Мы говорим, что элемент с ключом к хешируется в ячейку h (k); величина h (к) называется хеш-значением ключа к. Цель хеш-функции состоит в том, чтобы уменьшить рабочий диапазон индексов массива, и вместо |U| значений мы можем обойтись всего лишь m значениями. Соответственно снижаются и требования к количеству памяти.



- Однако здесь есть одна проблема: два ключа могут быть хешированы в одну и ту же ячейку. Такая ситуация называется коллизией. К счастью, имеются эффективные технологии для разрешения конфликтов, вызываемых коллизиями.

- Важнейшей задачей программиста является организация быстрого поиска информации. Существуют следующие способы поиска (в порядке увеличения скорости):
- Полный перебор вариантов O(2N)
- Поиск с барьером O(N)
- Бинарный поиск $O(\log_2 N)$
- Дерево поиска $O(\log_2 N)$
- ХЕШ-функции O(1)

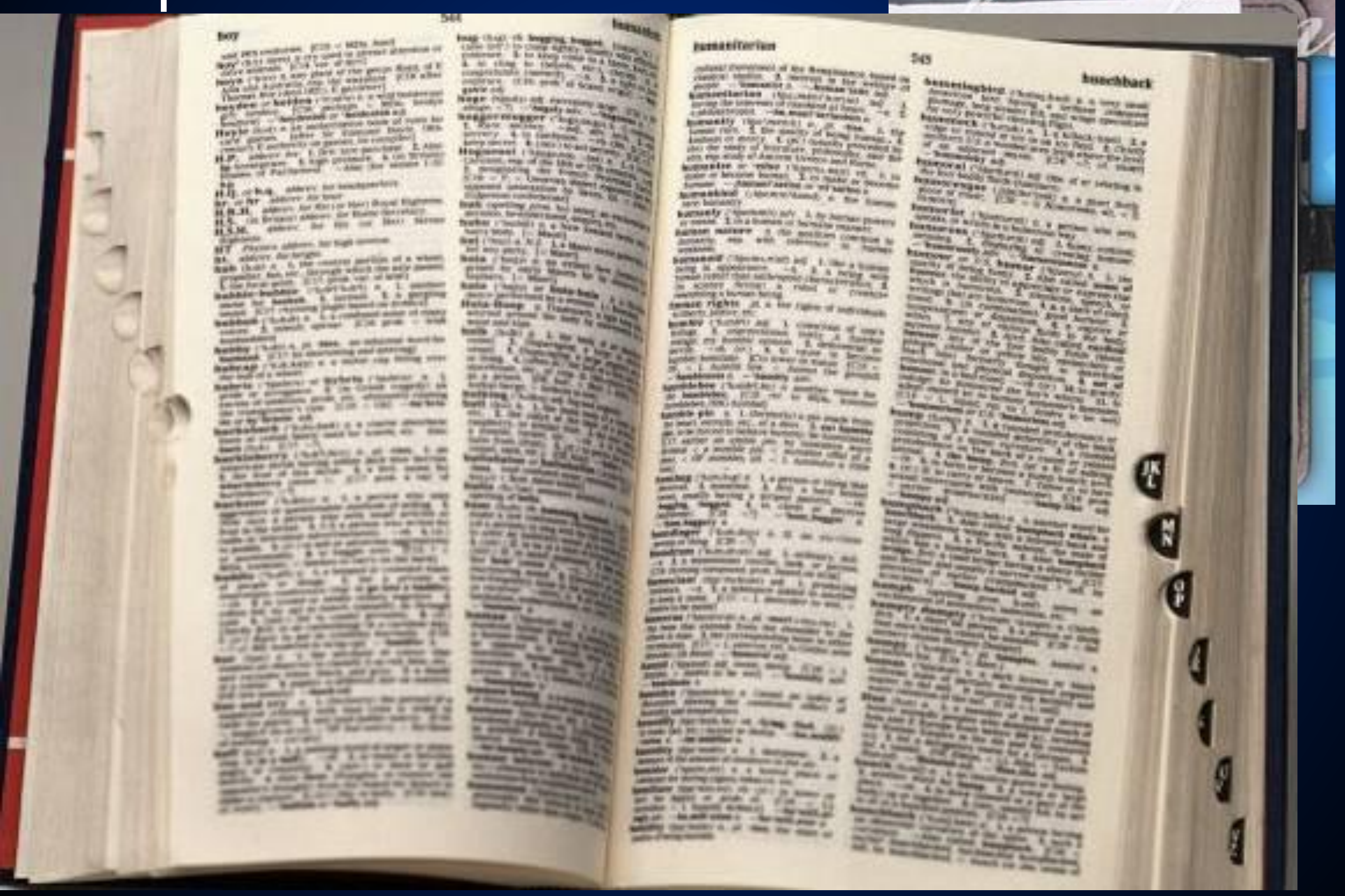
	0	1	2	3	4	5	6	7	8	N-1
m	5	3	1	-7	21	4	11	8	10	2

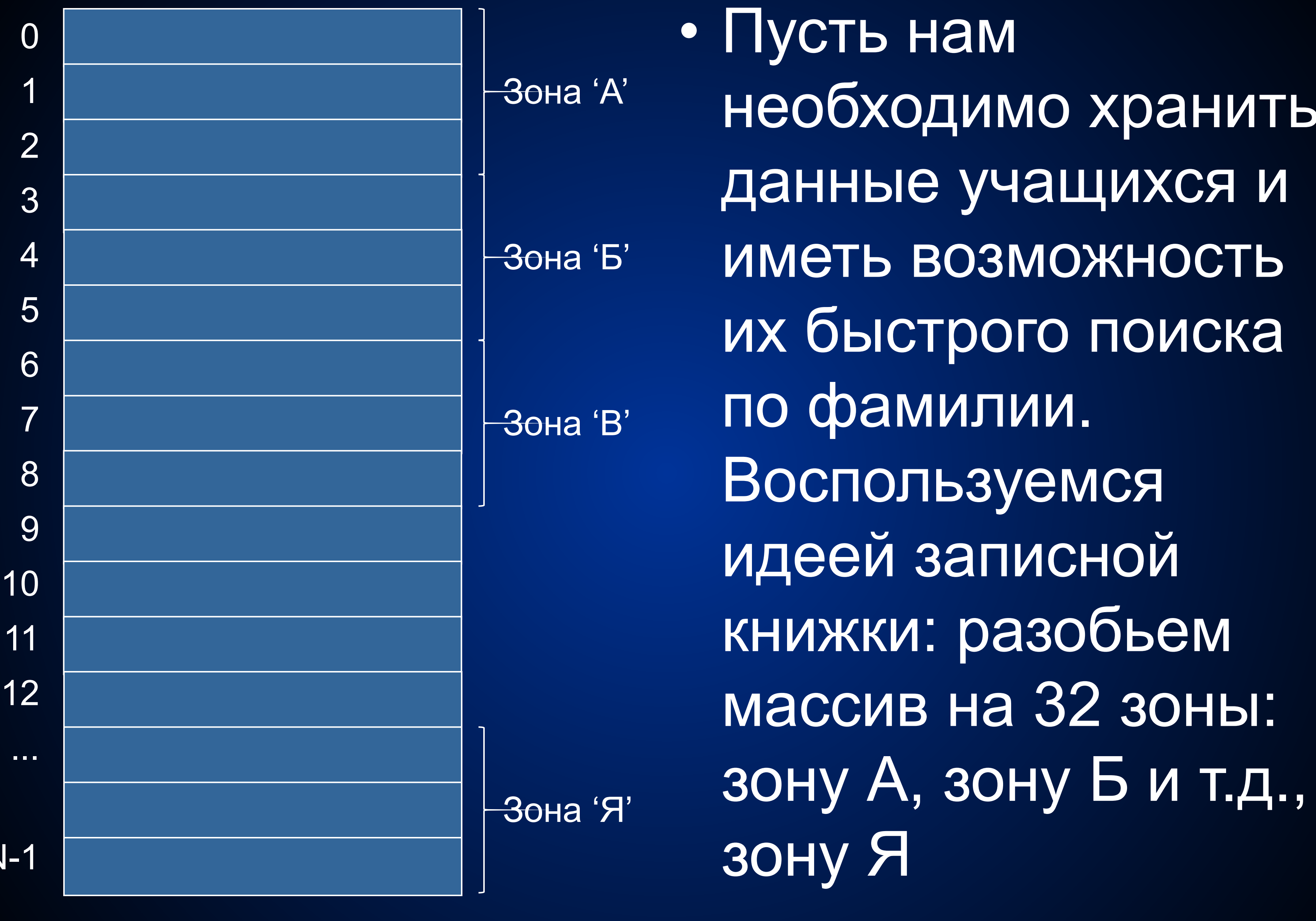
- Пусть имеется массив m, в котором хранится инф… кция h, которая по зн… предположит… расположени…
- i1:=h(key1)
- i21:=h(key2)
- i1<>i2 при key1<>key1 (… не обязательно)
- Ситуация, когда … **коллизией**

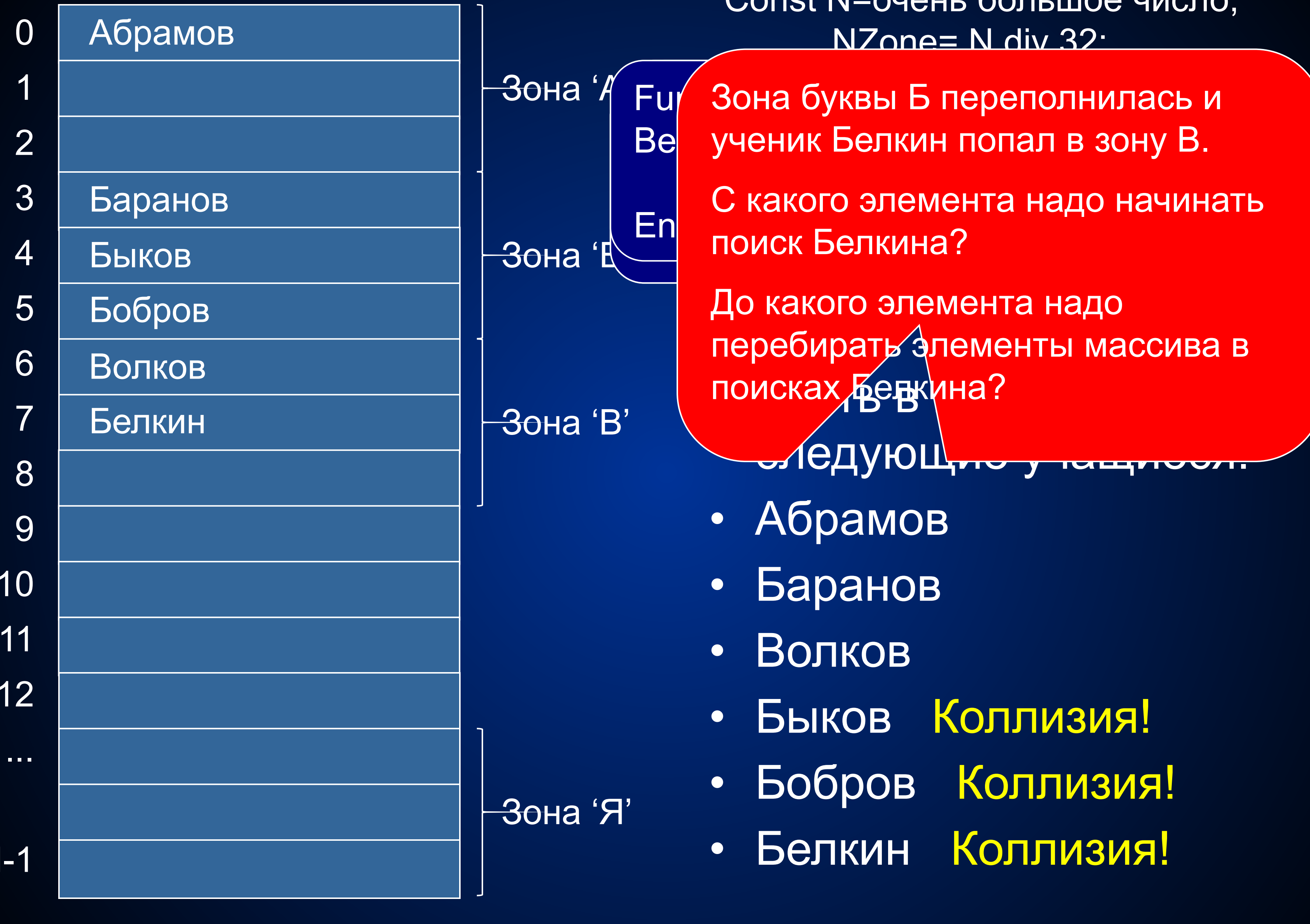
Например,
i1:=h(10) = 8
i2:=h(-7)=3

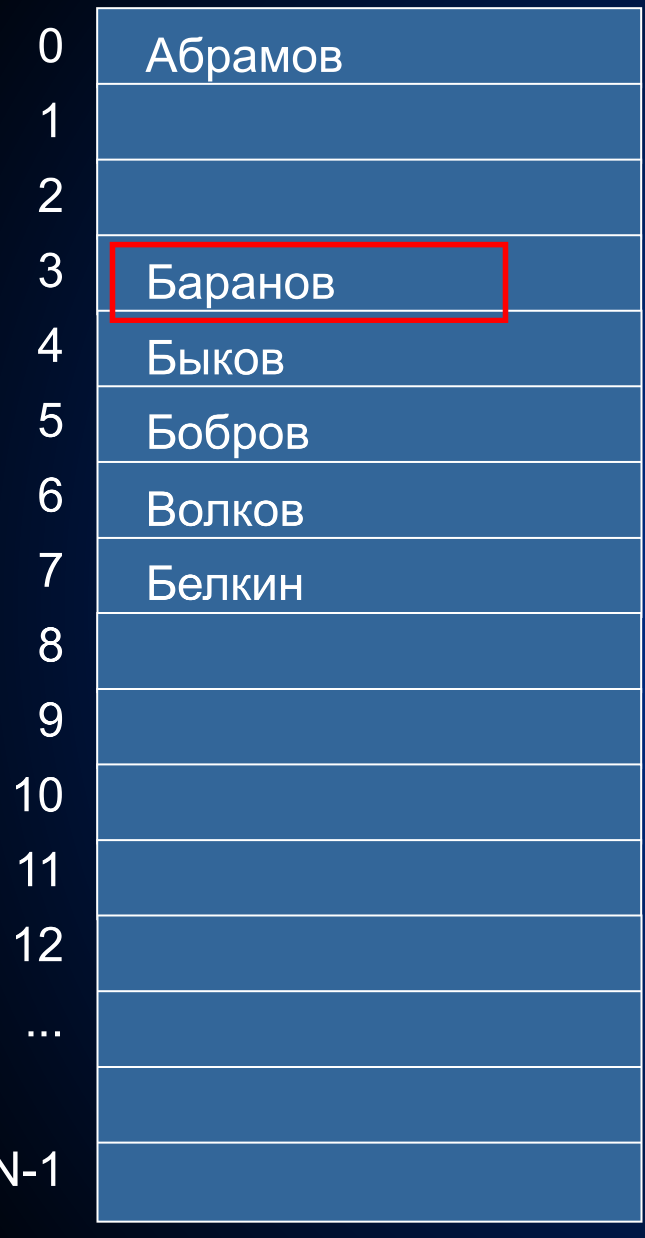
ХЕШ-функцией называется функция, которая по значению ключа поиска возвращает предполагаемое место расположения этого ключа

- Где мы встречались с ХЕШ-функциями в реальной жизни?




0
1
2
3
4
5
6
7
8
9
10
11
12
...
N-1
Зона 'А'
Зона 'Б'
Зона 'В'
Зона 'Я'
• Пусть нам необходимо хранить данные учащихся и иметь возможность их быстрого поиска по фамилии. Воспользуемся идеей записной книжки: разобьем массив на 32 зоны: зону А, зону Б и т.д., зону Я


Const N=очень большое число;
NZone= N div 32;
0
1
2
3
4
5
6
7
8
9
10
11
12
...
N-1
Абрамов
Баранов
Быков
Бобров
Волков
Белкин
Зона 'В'
Зона 'Я'
Зона буквы Б переполнилась и ученик Белкин попал в зону В.
С какого элемента надо начинать поиск Белкина?
До какого элемента надо перебирать элементы массива в поисках Белкина?
Абрамов
Баранов
Волков
Быков Коллизия!
Бобров Коллизия!
Белкин Коллизия!




```
Function Seek(s:string; var k:word):boolean;
begin
   k:=h(s);
   While (m[k]<>s) and (m[ k ]<>’’) do
      (k:=k+1) mod N;
   Seek:=(m[ k ]=s)
end;
BEGIN
…
   if seek(‘Бобров’,k)
   Then writeln(‘Есть!’, k)
   if seek(‘Буйволов’,k)
   Then writeln(‘Есть!’, k)
```

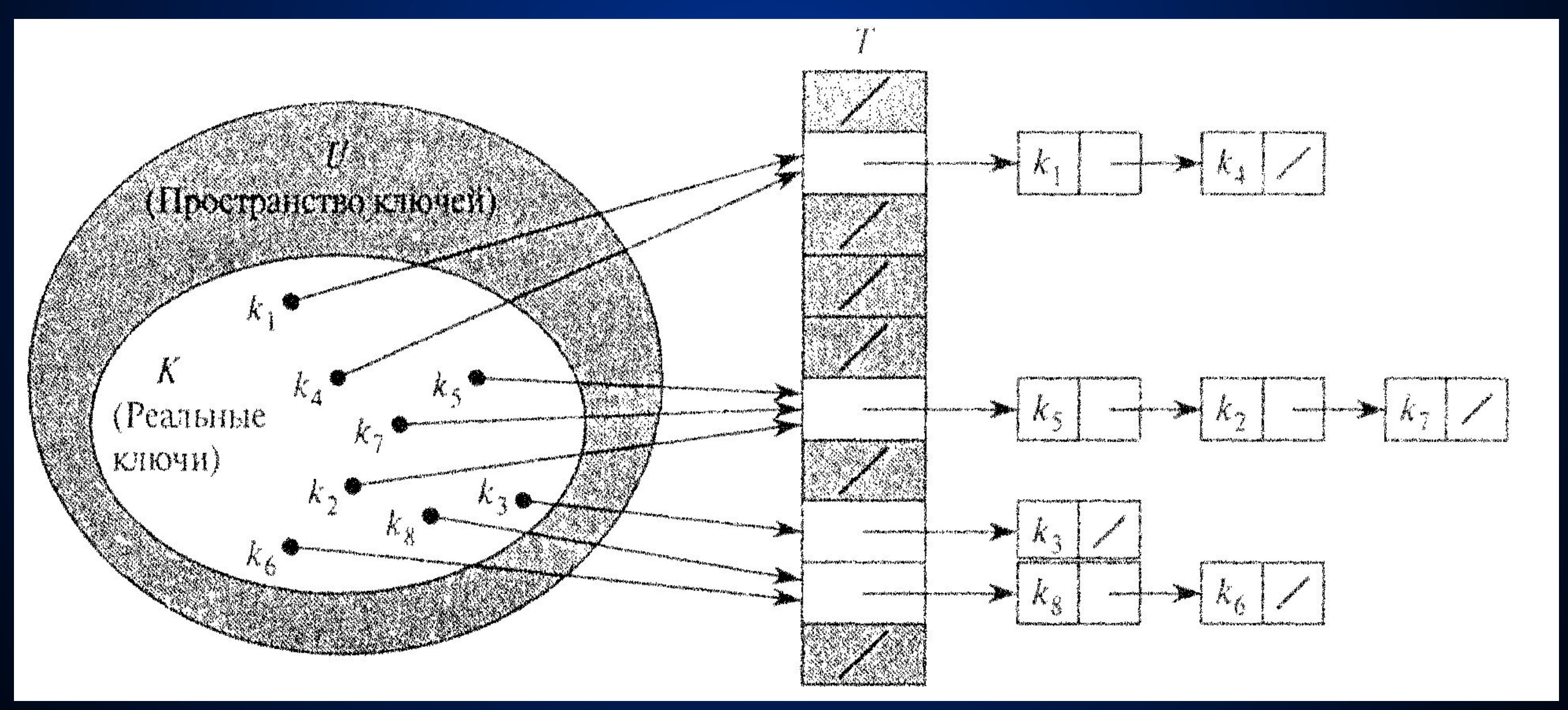
Какие проблемы можно заметить при работе с таким списком?

Что будет, если из класса уйдет ученик Бобров? Как это отразится на поиске Белкина?

- Задание:
- Реализуйте на ПК программу работы с таким списком:
- Добавление нового ученика в список;
- Удаление ученика из списка;
- Поиск ученика в списке;
- Вывод списка на экран.

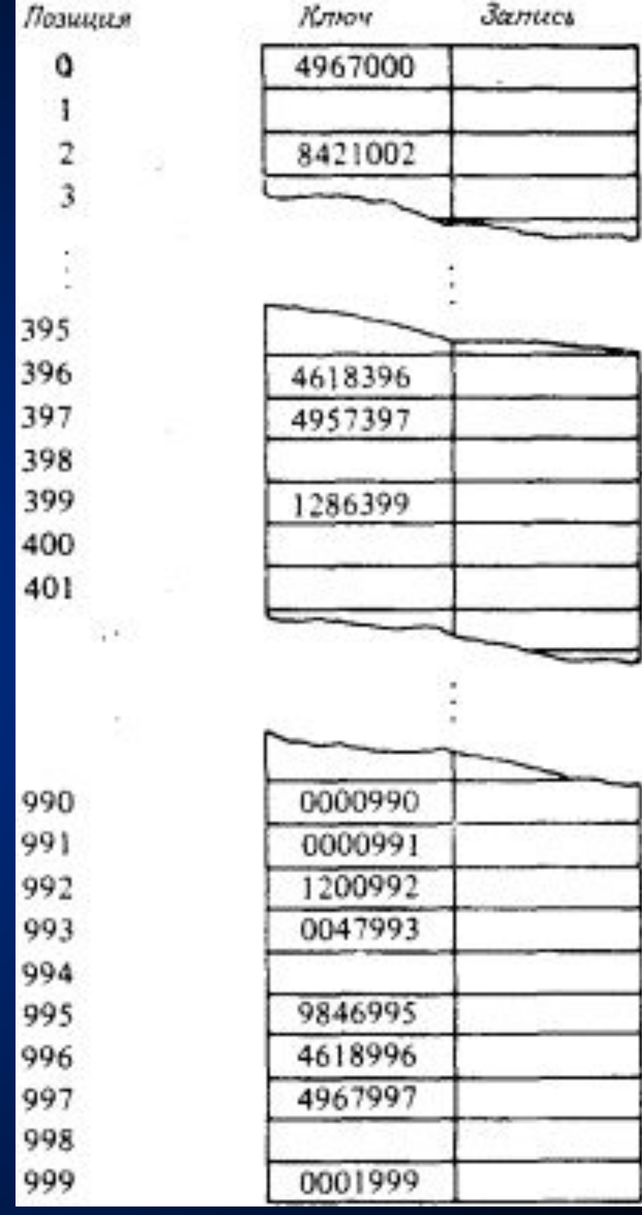
Разрешение коллизий при помощи цепочек

- При использовании данного метода мы объединяем все элементы, хешированные в одну и ту же ячейку, в связанный список, как показано на рис.Ячейка j содержит указатель на заголовок списка всех элементов, хеш-значение ключа которых равно j; если таких элементов нет, ячейка содержит значение NIL. На рисунке показано разрешение коллизий, возникающих из-за того, что h(k1) = h (K4), h (K5) = h (K2) = h (K7) и h (K8) = h (K6).



Если каждый ключ должен быть извлечен за один доступ, то положение записи внутри такой таблицы может зависеть только от данного ключа. Оно не может зависеть от расположения других ключей, как это имеет место в дереве. Наиболее эффективным способом организации такой таблицы является массив.
Предположим, что фирма некоторая фирма выпускает детали и кодирует их семизначными цифрами. Для применения прямой индексации с использованием полного семизначного ключа потребовался бы массив из 100 млн. элементов. Ясно, что это привело бы к потере неприемлемо большого пространства, поскольку совершенно невероятно, что какая-либо фирма может иметь больше чем тысяча наименований изделий. Поэтому необходим некоторый метод преобразования ключа в какое-либо целое число внутри ограниченного диапазона.

Тогда для хранения всего файла будет достаточно массива из 1000 элементов. Этот массив индексируется целым числом в диапазоне от 0 до 999 включительно. В качестве индекса записи об изделии в этом массиве используются три последние цифры номера изделия.



Позиция	Ключ	Запись
0	4967000	
1		
2	8421002	
3		
⋮	⋮	
395		
396	4618396	
397	4957397	
398		
399	1286399	
400		
401		
	⋮	
990	0000990	
991	0000991	
992	1200992	
993	0047993	
994		
995	9846995	
996	4618996	
997	4967997	
998		
999	0001999	

Отметим, что два ключа, которые близки друг к другу как числа (такие как 4618396 и 4618996), могут располагаться дальше друг от друга в этой таблице, чем два ключа,. которые значительно различаются как числа (такие как 0000991 и 9846995). Это происходит из-за того, что для определения позиции записи используются только три последние цифры ключа.

Хеширование - это способ сведения хранения одного большого множества к более меньшему.

Функция, которая трансформирует ключ в некоторый индекс в таблице, называется хеш-функцией.

В данном случае h(key):= key mod 1000;

Хеш-таблица - это обычный массив с необычной адресацией, задаваемой хеш-функцией.{см. рисунок}

Этот метод имеет один недостаток. Давайте добавим в таблицу запись с ключом 0596397. Увидим, что данная ячейка уже занята.

Ситуация, когда два или более ключа ассоциируются с одной и той же ячейкой называется коллизией при хешировании.

Следует отметить, однако, что хорошей хеш-функцией является такая функция, которая минимизирует коллизии и распределяет записи равномерно по всей таблице.

Совершенная хеш-функция - эта функция, которая не порождает коллизий.

Разрешить коллизии при хешировании можно 2 методами:

методом открытой адресации

методом цепочек

Разрешение коллизий при хешировании методом открытой адресации

Посмотрим, что произойдет, если мы захотим ввести в таблицу некоторый новый номер изделия 0596397. Используя хеш-функцию h(key):=key mod 1000, мы найдем, что h (0596397) =397 и что запись для этого изделия должна находиться в позиции 397 в массиве. Однако позиция 397 уже занята, поскольку там находится запись с ключом 4957397. Следовательно, запись с ключом 0596397 должна быть вставлена в таблицу в другом месте.

Самым простым методом разрешения коллизий при хешировании является помещение данной записи в следующую свободную позицию в массиве. Например, запись с ключом 0596397 помещается в ячейку 398, которая пока свободна, поскольку 397 уже занята. Когда эта запись будет вставлена, другая запись, которая хешируется в позицию 397 (с таким ключом, как 8764397) или в позицию 398 (с таким ключом, как 2194398), вставляется в следующую свободную позицию, которая в данном случае равна 400.

Если ячейка массива h(key) уже занята некоторой записью с другим ключом, то функция rh применяется к значению h(key) для того, чтобы найти другую ячейку, куда может быть помещена эта запись. Если ячейка rh(h(key)) также занята, то хеширование выполняется еще раз и проверяется ячейка rh(rh(h(key))). Этот процесс продолжается до тех пор, пока не будет найдена пустая ячейка. Rh - это функция повторного хеширования, которая воспринимает один индекс в массиве и выдает другой индекс.

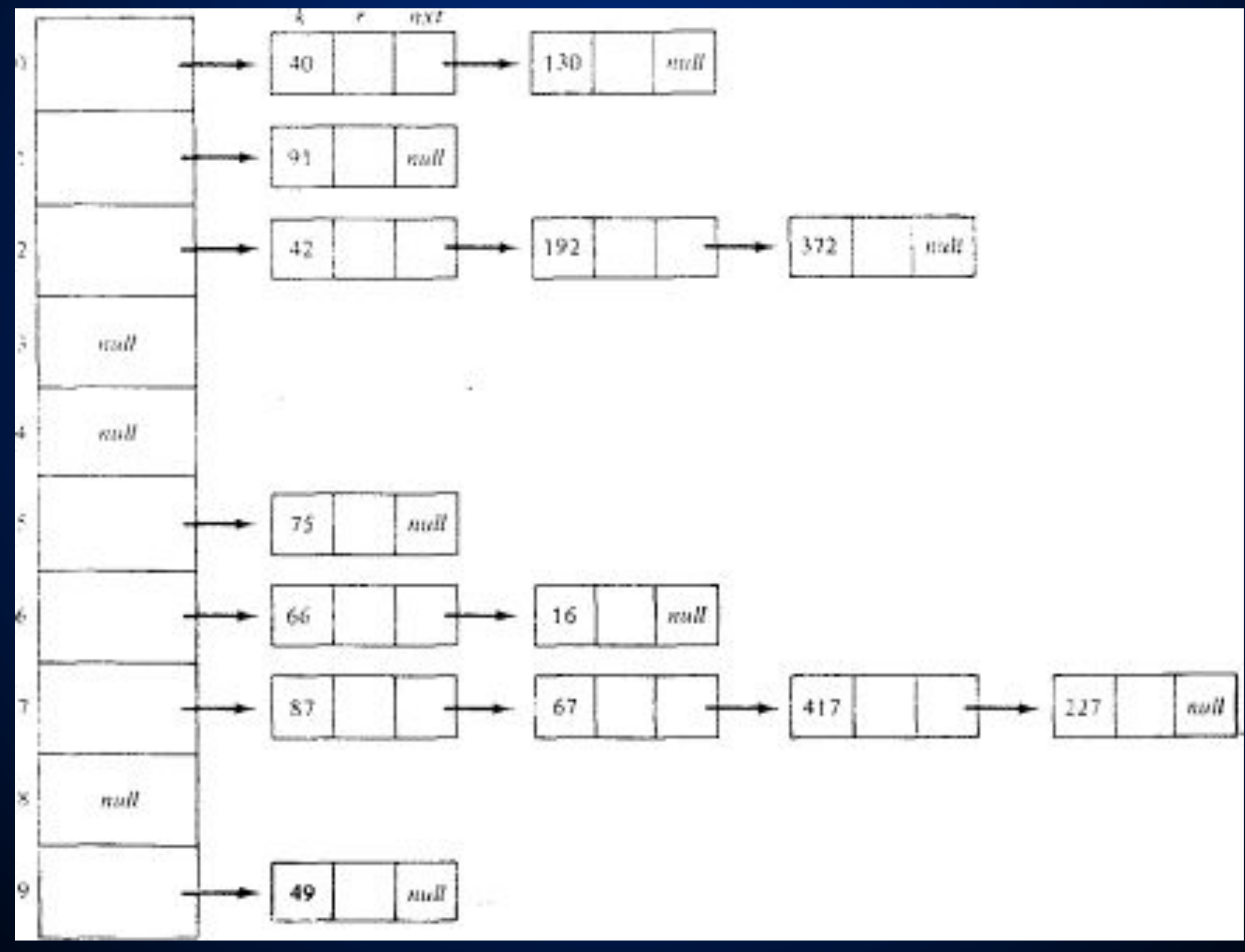
- Var

```
   K: array [0...999] of integer;

Function h(key: integer): integer;
Begin
   h := key mod 1000;
End;

Function rh(i: integer): integer;
Begin
   rh:=i+1 mod 1000;
End;

Procedure insert(key: integer);
Var
   I: integer;
begin
  I := h(key); {хешируем ключ}
 while ((k(i)< >key) and (k(i)< >0)) do
     i := rh(i); {мы должны выполнить повторное хеширование}
 if k(i) =0 then {вставляем запись в пустую позицию}
             k(i)=key
end;
```

- Недостатки метода.
- Во-первых, он предполагает фиксированный размер таблицы. Если число записей превысит этот размер, то их невозможно вставлять без выделения таблицы большего размера и повторного вычисления значений хеширования для ключей всех записей, находящихся уже в таблице, используя новую хеш-функцию.
- Во -вторых, из такой таблицы трудно удалить запись.

Разрешение коллизий при хешировании методом цепочек

- Он представляет собой организацию связанного списка из всех записей, чьи ключи хешируются в одно и то же значение.
- 75 66 42 192 91 40 49 87 67 16 417 130 372 227



```
type
link = ^node;
node = record
        key: integer;
        st: string;
        next: link;
      end;

var
mas: array[0..9] of link;
```

- ```
  function h(key: integer): integer;
  begin
     h:=key mod 10;
  end;

  function search(key1: integer; st1: string): link;
  var i: integer; q, p, s: link;
  begin
     i:= h(key1);
     q:=nil;  p:=mas[i];
  while p <> nil do  begin
    if p^.key = key1
  ```
- ```
    then  begin
            search:=p;    exit;
          end;
  q := p; p := p^.link;
  end;
  {Если ключ не найден, вставляем новую запись}
  new(s);
  s^.key:=key1;s^.st:=st1;
  s^.next:=nil;
  if q = nil then  mas[i]:=s
  else  q^.next:=s;
  search:=s;
  end;
  ```

Чем определяется качество хеш-функции?

- Качественная хеш-функция удовлетворяет (приближенно) предположению простого равномерного хеширования: для каждого ключа равновероятно помещение в любую из га ячеек, независимо от хеширования остальных ключей. К сожалению, это условие обычно невозможно проверить, поскольку, как правило, распределение вероятностей, в соответствии с которым поступают вноси- вносимые в таблицу ключи, неизвестно; кроме того, вставляемые ключи могут не быть независимыми. Иногда распределение вероятностей оказывается известным. Например, если известно, что ключи представляют собой случайные действительные числа, равномерно распределенные в диапазоне 0 < к < 1, то хеш-функция h(к) = |k m| удовлетворяет условию простого равномерного хеширования.
  - **Интерпретация ключей как целых неотрицательных чисел**
- Для большинства хеш-функций пространство ключей представляется множеством целых неотрицательных чисел N = {0,1,2,...}. Если же ключи не являются целыми неотрицательными числами, то можно найти способ их интерпретации как таковых. Например, строка символов может рассматриваться как целое число, записанное в соответствующей системе счисления. Так, идентификатор pt можно рассматривать как пару десятичных чисел 112,116), поскольку в ASCII-наборе символов p = 112 и t = 116. Рассматривая pt как число в системе счисления с основанием 128, мы находим, что оно соответствует значению 112 • 128 +116 = 14452.

Метод деления

- Построение хеш-функции методом деления состоит в отображении ключа к в одну из ячеек путем получения остатка от деления к на rа, т.е. хеш-функция имеет вид h(k) = к mod m. Например, если хеш-таблица имеет размер m = 12, а значение ключа к = 100, то h (к) = 4. Поскольку для вычисления хеш-функции требуется только одна операция деления, хеширование методом деления считается достаточно быстрым. При использовании данного метода мы обычно стараемся избегать некоторых значений m. Например, m не должно быть степенью 2, поскольку если $m = 2^P$, то h (к) представляет собой просто р младших битов числа к. Если только заранее не известно, что все наборы младших р битов ключей равновероятны, лучше строить хеш-функцию таким образом, чтобы ее результат зависел от всех битов ключа. Зачастую хорошие результаты можно получить, выбирая в качестве значения m простое число, достаточно далекое от степени двойки. Предположим, например, что мы хотим создать хеш-таблицу с разрешением коллизий методом цепочек для хранения n = 2000 символьных строк, размер символов в которых равен 8 битам. Нас устраивает проверка в среднем трех элементов при неудачном поиске, так что мы выбираем размер таблицы равным m = 701. Число 701 выбрано как простое число, близкое к величине 2000/3 и не являющееся степенью 2. Рассматривая каждый ключ к как целое число, мы получаем искомую хеш-функцию: h(k) = k mod 701.

Метод умножения

- Построение хеш-функции методом умножения выполняется в два этапа. Сначала мы умножаем ключ к на константу 0 < A < 1 и получаем дробную часть полученного произведения. Затем мы умножаем полученное значение наши при-применяем к нему функцию “mod" т.е. h(k)= ⌊m(kA mod i) ⌋, где выражение "кА mod 1" означает получение дробной части произведения кА, т.е. величину к А — ⌊кА⌋. Достоинство метода умножения заключается в том, что значение m перестает быть критичным. Обычно величина m из соображений удобства реализации функции выбирается равной степени 2.

Выбор хеш-функции

- Обратимся теперь к вопросу о том, как выбрать хорошую хеш-функцию. Ясно, что эта функция должна создавать как можно меньше коллизий при хешировании, т.е. она должна равномерно распределять ключи на имеющиеся индексы в массиве. Конечно, нельзя определить, будет ли некоторая конкретная хеш-функция распределять ключи правильно, если эти ключи заранее не известны. Однако, хотя до выбора хеш-функции редко известны сами ключи, некоторые свойства этих ключей, которые влияют на их распределение, обычно известны.
- метод деления. Некоторый целый ключ делится на размер таблицы и остаток от деления берется в качестве значения хеш-функции. Эта хеш-функция обозначается h (key) := key mod m.
- метод середины квадрата. Ключ умножается сам на себя и в качестве индекса используется несколько средних цифр этого квадрата.
-
```
Function h(key: integer): integer;
Begin
    Key:=key*key; {Возвести в квадрат}
    Key:=key shl 11;{Отбросить 11 младших бит}
    H:= key mod 1024;{Возвратить 10 младших бит}
End;
```

Аддитивный метод для строк

- 3) Аддитивный метод для строк (размер таблицы равен 256). Для строк вполне разумные результаты дает сложение всех символов и возврат остатка от деления на 256.
- Function h(st: string): integer;
  Var Sum: longint;   I: integer;
  Begin
    For i:=0 to length(st) do
        Sum := sum + ord(st[i]);
    H:=sum mod 256;
  End;

Исключающее ИЛИ для строк

- 4) Исключающее ИЛИ для строк (размер таблицы равен 256). Этот метод аналогичен аддитивному, но успешно различает схожие слова и анаграммы (аддитивный метод даст одно значение для XY и YX). Метод заключается в том, что к элементам строки последовательно применяется операция "исключающее или". В алгоритме добавляется случайная компонента, чтобы еще улучшить результат.
- Var rand8: array[0..255] of integer;

```
Var   rand8: array[0..255] of integer;
procedure init;
var   i: integer;
begin    randomize;
  for i:=0 to 255 do
      rand8[i]:=random(255);
end;

function h(st: string): integer;
Var   Sum: longint;    I: integer;
Begin
  For i:=0 to length(st) do
         Sum := sum + ord(st[i]) xor rand8[i];
  H:=sum mod 256;
end;
```

Открытая адресация

- При использовании метода открытой адресации все элементы хранятся непосредственно в хеш-таблице, т.е. каждая запись таблицы содержит либо элемент динамического множества, либо значение NIL. При поиске элемента мы систематически проверяем ячейки таблицы до тех пор, пока не найдем искомый элемент или пока не убедимся в его отсутствии в таблице. Здесь, в отличие от метода цепочек, нет ни списков, ни элементов, хранящихся вне таблицы. Таким образом, в методе открытой адресации хеш-таблица может оказаться заполненной, делая невозможной вставку новых элементов; коэффициент заполнения а не может превышать 1. Конечно, при хешировании с разрешением коллизий методом цепочек можно использовать свободные места в хеш-таблице для хранения связанных списков, но преимущество открытой адресации заключается в том, что она позволяет полностью отказаться от указателей. Вместо того чтобы следовать по указателям, мы вычисляем последовательность проверяемых ячеек. Дополнительная память, освобождающаяся в результате отказа от указателей, позволяет использовать хеш-таблицы большего размера при том же общем коли- количестве памяти, потенциально приводя к меньшему количеству коллизий и более быстрой выборке.

- Для выполнения вставки при открытой адресации мы последовательно проверяем, или исследуем (probe), ячейки хеш-таблицы до тех пор, пока не находим пустую ячейку, в которую помещаем вставляемый ключ. Вместо фиксированного порядка исследования ячеек 0,1,..., m — 1 (для чего требуется Θ (n) времени), последовательность исследуемых ячеек зависит от вставляемого в таблицу ключа. Для определения исследуемых ячеек мы расширим хеш-функцию, включив в нее в качестве второго аргумента номер исследования (начинающийся с 0). В результате хеш-функция становится следующей:

  h : U x {0,1,..., m - 1} -> {0,1,..., m - 1} .
  В методе открытой адресации требуется, чтобы для каждого ключа к последовательность исследований

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

  представляла собой перестановку множества (0,1,..., m — 1), чтобы в конечном счете могли быть просмотрены все ячейки хеш-таблицы. В приведенном далее псевдокоде предполагается, что элементы в таблице T представляют собой ключи без сопутствующей информации; ключ к тождественен элементу, содержащему ключ к. Каждая ячейка содержит либо ключ, либо значение nil (если она не заполнена):

```
Hash_Insert(T, k)
i<-0
repeat
  j <— h(k, i)
  if T[j] = NIL
  then T[j] <- к; return j
  else inc(i)
until i = m
error "Хеш-таблица переполнена"
```

Алгоритм поиска ключа к исследует ту же последовательность ячеек, что и алгоритм вставки ключа к. Таким образом, если при поиске встречается пустая ячейка, поиск завершается неуспешно, поскольку ключ к должен был бы быть вставлен в эту ячейку в последовательности исследований, и никак не позже нее.

```
Hash_Search(T, к)
i<-0
repeat
  J <- h(k,I)
  If T[J]=K
  then RETURN J
  INC(I)
until T[ J ]=nil or i=m
return nil
```

- Процедура удаления из хеш-таблицы с открытой адресацией достаточно сложна. При удалении ключа из ячейки i мы не можем просто пометить ее значением NIL. Поступив так, мы можем сделать невозможным выборку ключа k, в процессе вставки которого исследовалась и оказалась занятой ячейка i. Одно из решений состоит в том, чтобы помечать такие ячейки специальным значением deleted вместо nil. При этом мы должны слегка изменить процедуру Hash_Insert, с тем чтобы она рассматривала такую ячейку как пустую и могла вставить в нее новый ключ. В процедуре Hash_Search никакие изменения не требуются, поскольку мы просто пропускаем такие ячейки при поиске и исследуем следующие ячейки в последовательности. Однако при использовании специального значения deleted время поиска перестает зависеть от коэффициента заполнения a, и по этой причине, как правило, при необходимости удалений из хеш-таблицы в качестве метода разрешения коллизий выбирается метод цепочек.

Линейное исследование

- Пусть задана обычная хеш-функция h : U -> {0,1,..., m — 1}, которую мы будем в дальнейшем именовать вспомогательной хеш-функцией (auxiliary hash function). Метод линейного исследования для вычисления последовательности исследований использует хеш-функцию
- h (k, i) = (h` (k) + i) mod m,
- где i принимает значения от 0 до m— 1 включительно. Для данного ключа к первой исследуемой ячейкой является T [h` (k)], т.е. ячейка, которую дает вспомогательная хеш-функция. Далее мы исследуем ячейку T [h` (к) + 1] и далее последовательно все до ячейки T [m — 1], после чего переходим в начало таблицы и последовательно исследуем ячейки T [0], T [1], и так до ячейки T [h` (к) — 1]. Поскольку начальная исследуемая ячейка однозначно определяет всю последовательность исследований целиком, всего имеется m различных последовательностей.
- Линейное исследование легко реализуется, однако с ним связана проблема первичной кластеризации, связанной с созданием длинных последовательностей занятых ячеек, что, само собой разумеется, увеличивает среднее время поиска.

Квадратичное исследование

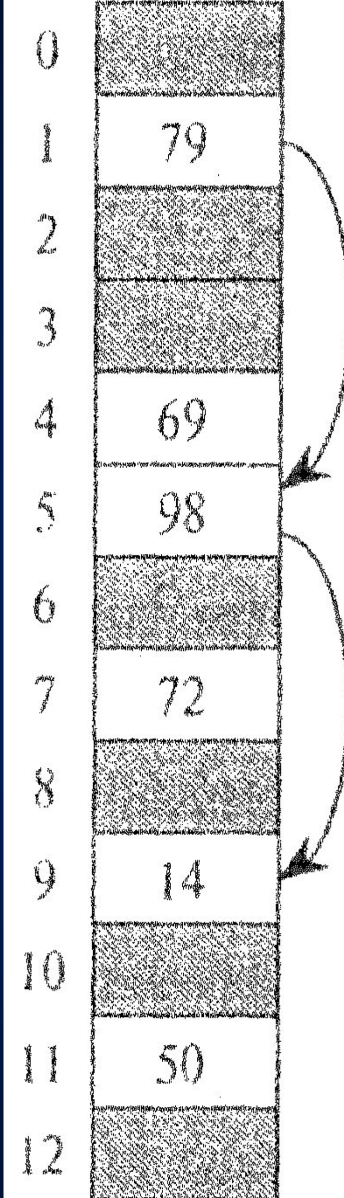
- Квадратичное исследование использует хеш-функцию вида $h(к, i) = (h`(к) + c_1 i + c_2 i^2) \bmod m$,
- где h` — вспомогательная хеш-функция, $c_1$ и $c_2 <> 0$ — вспомогательные константы, а i принимает значения от 0 до m — 1 включительно. Начальная исследуемая ячейка — T [h` (к)]; остальные исследуемые позиции смещены относительно нее на величины, которые описываются квадратичной зависимостью от номера исследования i. Этот метод работает существенно лучше линейного исследования, но для того, чтобы исследование охватывало все ячейки, необходим выбор специальных значений c1 c2 и m.

Двойное хеширование

- Двойное хеширование представляет собой один из наилучших способов использования открытой адресации, поскольку получаемые при этом перестановки обладают многими характеристиками случайно выбираемых перестановок. Двойное хеширование использует хеш-функцию вида $h(k)\ i) = (h_1(k) + ih_2(k)) \bmod m$, где h1 и h2 — вспомогательные хеш-функции. Начальное исследование выполняется в позиции T [h1 (K)], а смещение каждой из последующих исследуемых ячеек относительно предыдущей равно h2 (k) по модулю m. Следовательно, в отличие от линейного и квадратичного исследования, в данном случае последовательность исследования зависит от ключа к по двум параметрам — в плане выбора начальной исследуемой ячейки и расстояния между соседними исследуемыми ячейками, так как оба эти параметра зависят от значения ключа.

Вы видите хеш-таблицу размером 13 ячеек, в которой используются вспомогательные хеш-функции h1 (к) = к mod 13 и h2 (к) = 1 + (к mod 11).Так как 14 = 1 (mod 13) и 14 = 3 (mod 11), ключ 14 вставляется в пустую ячейку 9, после того как при исследовании ячеек 1 и 5 выясняется, что эти ячейки заняты. Для того чтобы последовательность исследования могла охватить всю таблицу, значение h2 (к) должно быть взаимно простым с размером хеш-таблицы m.

Удобный способ обеспечить выполнение этого условия состоит в выборе числа m, равного степени 2, и разработке хеш-функции h2 таким образом, чтобы она возвращала только нечетные значения. Еще один способ состоит в использовании в качестве m простого числа и построении хеш-функции h2 такой, чтобы она всегда возвращала натуральные числа, меньшие m. Например, можно выбрать простое число в качестве m, а хеш-функции такими: hi (к) = к mod m, h2 (к) = 1 + (к mod m`) , где m' должно быть немного меньше m (например, m — 1). Скажем, если к = = 123456, m = 701, а m' = 700, то h1 (к) = 80 и h2 (к) = 257,



Поиск хешированием

В основе поиска лежит переход от исходного множества к множеству хеш-функций h(k). Хеш-функция имеет следующий вид:

h(k) = к mod m, где к — ключ; m — целое число; mod — остаток от целочисленного деления.

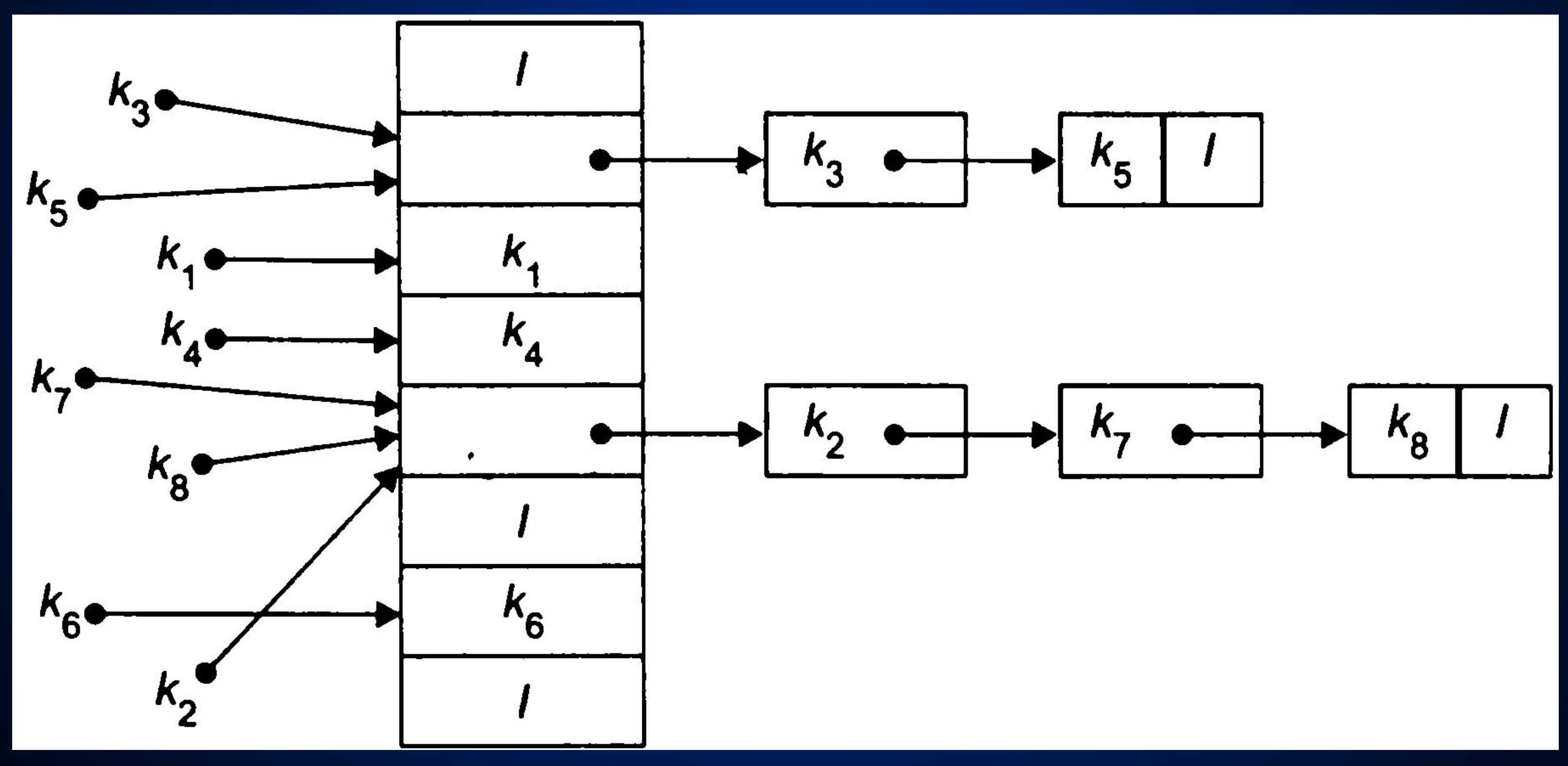
Например, пусть дано множество {9, 1, 4, 10, 8, 5}. Определим для него хеш-функцию h{k) = к mod m.

- Пусть m = 1, тогда h(k) = {0, 0, 0, 0, 0, 0}. Множество хеш-функций состоит из нулей.
- Пусть m = 20, тогда h(к) = {9, 1, 4, 10, 8, 5}. Множество хеш-функций повторяет исходное множество.
- Пусть m равно половине максимального ключа m = [Kmax/2],тогда m= [10/2] = 5; h(k) = {4, 1, 4, 0, 3, 0}.

Хеш-функция указывает адрес, по которому следует отыскивать ключ. Для разных ключей хеш-функция может принимать одинаковые значения, такая ситуация называется коллизией.

Пример 1

- Дано множество ключей {7, 13, 6, 3, 9, 4, 8, 5}. Найти ключ К = 27.Хеш-функция равна h(k) = k mod m; m = [13/2] = 6 (так как 13 —максимальный ключ); h(k) = {1, 1, 0, 3, 3, 4, 2, 5}.



Алгоритмы хэширования в задачах на строки

- Алгоритмы хеширования строк помогают решить очень много задач. Но у них есть большой недостаток: что чаще всего они не 100%-ны, поскольку есть множество строк, хэши которых совпадают. Другое дело, что в большинстве задач на это можно не обращать внимания, поскольку вероятность совпадения хэшей всё-таки очень мала.
- Один из лучших способов определить хэш-функцию от строки S следующий:
- h(S) = S[0] + S[1] * P + S[2] * P^2 + S[3] * P^3 + ... + S[N] * P^N
- где P - некоторое число. Разумно выбирать для P простое число, примерно равное количеству символов во входном алфавите. Например, если строки предполагаются состоящими только из маленьких латинских букв, то хорошим выбором будет P = 31. Если буквы могут быть и заглавными, и маленькими, то, например, можно P = 53.Во всех кусках кода в этой статье будет использоваться P = 31. Само значение хэша желательно хранить в самом большом числовом типе - int64, он же long long. Очевидно, что при длине строки порядка 20 символов уже будет происходить переполнение значение. Ключевой момент - что мы не обращаем внимание на эти переполнения, как бы беря хэш по модулю 2^64.
- Пример вычисления хэша, если допустимы только маленькие латинские буквы:

```
const  p:longint = 31;
          hash:longint = 0, p_pow:longint = 1;
for i=0 to  length(s) do
begin   // желательно отнимать 'a' от кода буквы
           // единицу прибавляем, чтобы у строки вида 'aaaaa' хэш был ненулевой
        hash := hash+ (s[i] - 'a' + 1) * p_pow;
        p_pow =p_pow* p;
End;
```

- В большинстве задач имеет смысл сначала вычислить все нужные степени P в каком-либо массиве.

Поиск одинаковых строк

- Уже теперь мы в состоянии эффективно решить такую задачу. Дан список строк S[1..N], каждая длиной не более M символов. Допустим, требуется найти все повторяющиеся строки и разделить их на группы, чтобы в каждой группе были только одинаковые строки. Обычной сортировкой строк мы бы получили алгоритм со сложностью O (N M log N), в то время как используя хэши, мы получим O (N M + N log N).
- Алгоритм. Посчитаем хэш от каждой строки, и отсортируем строки по этому хэшу.

```
•// ... считывание строк ...
•// считаем все степени p, допустим, до 10000 - максимальной длины строк
•const  p:longint = 31;
•p_pow:array[0..10000] of longint;
•…
•p_pow[0] := 1;
•for  i:=1 to размер p_pow do {рассчитаем заранее все степени числа P}
•       p_pow[i] := p_pow[i-1] * p;
•Хсчитаем хэши от всех строк в массиве храним значение хэша и номер строки в массиве sЪ
•for i:=0 to n-1 do
• begin
•       hash := 0;
•       for  j :=0 to length(s[i])-1 do
•           hash := hash +(s[i] [j] - 'a' + 1) * p_pow[ j ];
•       hashes[i] = make_pair (hash, i);
•End;
•// сортируем по хэшам
•sort (hashes.begin(), hashes.end());
•// выводим ответ
```

Хэш подстроки и его быстрое вычисление

Предположим, нам дана строка S, и даны индексы I и J. Требуется найти хэш от подстроки S[I..J].

По определению имеем:

H[I..J] = S[I] + S[I+1] * P + S[I+2] * P^2 + ... + S[J] * P^(J-I)

откуда:

H[I..J] * P[I] = S[I] * P[I] + ... + S[J] * P[J],

H[I..J] * P[I] = H[0..J] - H[0..I-1]

Полученное свойство является очень важным.

Действительно, получается, что, зная только хэши от всех префиксов строки S, мы можем за O (1) получить хэш любой подстроки. Единственная возникающая проблема - это то, что нужно уметь делить на P[I]. На самом деле, это не так просто. Поскольку мы вычисляем хэш по модулю 2^64, то для деления на P[I] мы должны найти к нему обратный элемент в поле (например, с помощью Расширенного алгоритма Евклида), и выполнить умножение на этот обратный элемент. Впрочем, есть и более простой путь. В большинстве случаев, вместо того чтобы делить хэши на степени P, можно, наоборот, умножать их на эти степени. Допустим, даны два хэша: один умноженный на P[I], а другой - на P[J]. Если I < J, то умножим перый хэш на P[J-I], иначе же умножим второй хэш на P[I-J]. Теперь мы привели хэши к одной степени, и можем их спокойно сравнивать.

Например, код, который вычисляет хэши всех префиксов, а затем за O (1) сравнивает две подстроки:

```
S: string; i1, i2, len: integer; // входные данные
// считаем все степени p
const p:longint = 31;
p_pow: array[0.. length(S)] of int64;
p_pow[0] = 1;
for i=1to length(s) do
	p_pow[i] = p_pow[i-1] * p;
// считаем хэши от всех префиксов
h:array[0..length(s)] of int 64;
for (i=0 to length(s) do
begin
	h[i] = (s[i] - 'a' + 1) * p_pow[i];
	if i>0 then h[i] := h[i]+h[i-1];
End;
// получаем хэши двух подстрок
  h1 = h[i1+len-1];
  if i1>0 then  h1 :=h1- h[i1-1];
  h2 = h[i2+len-1];
  if i2>0 then  h2 :=h2- h[i2-1];
// сравниваем их
 if (i1 < i2) and (h1 * p_pow[i2-i1] == h2) or
    (i1 > i2) and (h1 == h2 * p_pow[i1-i2])
Then writeln (‘equal’)
else  writeln(‘different’)
```

Определение количества различных подстрок

- Пусть дана строка S длиной N, состоящая только из маленьких латинских букв. Требуется найти количество различных подстрок в этой строке.Для решения переберём по очереди длину подстроки: L = 1 .. N. Для каждого L мы построим массив хэшей подстрок длины L, причём приведём хэши к одной степени, и отсортируем этот массив. Количество различных элементов в этом массиве прибавляем к ответу.

```
•S:string; // входная строка  n:longint=length(s);
•const p = 31; p_pow:array[0..length(s)] of int 64;
•p_pow[0] = 1;    // считаем все степени p
•for i=1 to length(s)-1 do
•        p_pow[i] := p_pow[i-1] * p;
•H: array[0..length(s] of int 64;   // считаем хэши от всех префиксов
•for i=0 to length(s)do begin
•        h[i] = (s[i] - 'a' + 1) * p_pow[i];
•        if (i)  h[i] += h[i-1];
•End;
•result = 0;   // перебираем длину подстроки
•for l=1 to n do begin
•        // ищем ответ для текущей длины, получаем хэши для всех подстрок длины l
•       for i=0 to n do begin
•            cur_h := h[i+l-1];
•            if i>0  cur_h := cur_h- h[i-1];           // приводим все хэши к одной степени
•                cur_h: = cur_h* p_pow[n-i-1];
•                hs[i] = cur_h;
•        end;     // считаем количество различных хэшей
•        sort (hs.begin(), hs.end());
•        hs.erase (unique (hs.begin(), hs.end()), hs.end());
•        result :=result+ hs.size();
•end
•Writeln(result);
```

Фибоначчиев поиск

- В этом поиске анализируются элементы, находящиеся в позициях, равных числам Фибоначчи. Числа Фибоначчи получаются по следующему правилу: каждое последующее число равно сумме двух предыдущих чисел, например:{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...}. Поиск продолжается до тех пор, пока не будет найден интервал между двумя ключами, где может располагаться отыскиваемый ключ. Фибоначчиев поиск предназначается для поиска аргумента К среди расположенных в порядке возрастания ключей К1 < К2 < ...... Kn
- Для удобства описания предполагается, что n +1 есть число Фибоначчи Kn+1. Подходящей начальной установкой данный метод можно сделать пригодным для любого n.

Алгоритм

1. [Начальная установка.] Установить i := Fk; P:=Fk-1; Q:=Fk-2 (В алгоритме p и q обозначают последовательные числа Фибоначчи.)
2. [Сравнение.] Если К < Ki то перейти k п.3; если К > Ki, то перейти к п.4; если К =Ki, то алгоритм заканчивается удачно.
3. [Уменьшение i] Если q =0, то алгоритм заканчивается неудачно. Если q<>0, то установить i := i - q, заменить (p, q) на (q, p-q) и вернуться на п.2.
4. [Увеличение i] Если p = 1, алгоритм заканчивается неудачно. Если p <> 1, установить i := i + q, p := p - q, q := q — p и вернуться к п.2.

Дано исходное множество ключей
{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}.
{1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18}

Пусть отыскиваемый ключ равен 42 (К = 42).Последовательное сравнение отыскиваемого ключа будет проводиться с элементами исходного множества, расположенными в позициях, равных числам Фибоначчи: 1, 2, 3, 5, 8,13, 21,...

Шаг 1. К ~ К1, 42 > 3, отыскиваемый ключ сравнивается с ключом, стоящим в позиции, равной числу Фибоначчи.

Шаг 2. К~ К2, 42 > 5, сравнение продолжается с ключом, стоящим в позиции, равной следующему числу Фибоначчи.

Шаг 3. К ~ К3, 42 > 8, сравнение продолжается.

Шаг 4. К ~ К5, 42 > 11, сравнение продолжается.

Шаг 5. К ~ К8, 42 > 19, сравнение продолжается.

Шаг 6. К ~ К13, 42 > 35, сравнение продолжается.

Шаг 7. К ~ К21=К18, 42 < 52, найден интервал, в котором находится отыскиваемый ключ, т.е. отыскиваемый ключ может находиться в исходном множестве между позициями 13 и 21, т.е. {35, 37, 42, 45,48, 52}.В найденном интервале поиск вновь ведется в позициях, равных числам Фибоначчи.

Отображения

Отображение – это функция, определенная на множестве элементов одного типа и принимающая значения из множества элементов другого множества. Допустимы следующие основные операции:

- MakeNull(M) – делает отображение M пустым
- Assign(M, d, r) – делает M(d) равным r
- Compute(M, d, r) возвращает true и r:=M(d), если значение M(d) уже определено, возвращает false – в противном случае.

Реализация отображений при помощи массивов

- procedure MAKENULL (var M: MAPPING) ;
- var i: domaintype;
- begin
-   for i;= firstvalue to Lastvalue do
-     M[i] : = неопределен
- end; {MAKENULL}

- procedure ASSIGN (var M: MAPPING; d: domain type; r: rangetype);
- begin
-     M[d] := r
- end; {ASSIGN }

- function COMPUTE (var M: MAPPING; d: domaintype; r: rangetype):boolean;
- begin
- if M[d] = неопределен then COMPUTE:= false
- else begin
-         r:= M[d] ; COMPUTE:= true
-     end
- end; { COMPUTE }

Реализация отображений при помощи списков

```
type
elementtype = record
            domain: domaintype;
            range: rangetype
end;
procedure ASSING ( var M: MAPPING; d: domaintype; r: range type ),
var  x: elementtype; p: position;
begin
    x.domain:= d; x. range: = r; p: = FIRST (M) ;
    whi.ll p <> END(M) do
       if RETRIEVE (p, M).domain = d
       then   DELETE (p, M) {удаляется элемент со значением d в поле domain}
       else  p: = NEXT (p, M) ;
INSERT (x, FIRST(M), M) {вставляем элемент (d, r) в начало списка}
end; {ASSIGN}
function COMPUTE ( var M: MAPPING; d: domaintype; var r: rangetype ): boolean;
var  p: position;
begin
p: = FIRST (M)
whill p <> END (M) do begin
  if RETRIEVE (P. M) . domain = d
  then begin
            r:= RETRIEVE (p, M) . range;           COMPUTE :=true;    exit
           end;
  p:= NEXT(p, M)
 end;
COMPUTE:=(false)
end; { COMPUTE }
```

Множество

- Множеством называется некая совокупность элементов, каждый элемент которой в свою очередь либо является множеством, либо примитивным элементом (атомом).
- В некоторых случаях элементы множества (атомы) считаются упорядоченными. Основные операции над множеством:
- UNION(A, B, C) (объединение), INTERSECTION(A, B, C) (пересечение), DIFFERENCE(A, B, C) (разность) – A и B – входные множества, C – результирующее.
- MERGE(A, B, C) – объединение множеств с учетом того, что первоначально они не пересекались.
- MEMBER(x, A) – проверяет наличие элемента x в множестве A.
- MAKENULL(A) – присваивает множеству A значение пустого множества.
- INSERT(x, A) – вставляет атом x в множество A.
- INSERT(x, A) – удаляет элемент x из A.
- ASSIGN(A, B) – присваивает множеству A значение множества B.
- MIN(A) – возвращает значение наименьшего элемента A.
- EQUAL(A, B) – возвращает true, если элементы A и B совпадают.

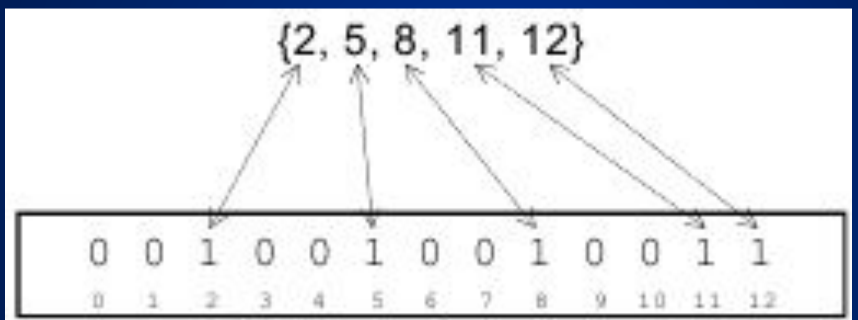
Реализация множеств с использованием двоичных векторов

Множество представляет собой набор атомов. Каждый атом может либо присутствовать в множеств, либо – нет. Представим множество двоичным числом: i-ый бит равен 1, если i-ый атом присутствует в множестве и 0 – в противном случае. Достоинство – очень быстро.

```
const  N= {количество атомов в множестве}
type SET = packed array[I..N] of boolean;
Var A: SET;
A[i] = true, если атом i присутствует в множестве.
procedure UNION ( A, B: SET; var C: SET );
var i: integer;
begin
for i: = 1 to N do
  C[i]:= A[ij or Bfi]
End;
```

Битовые вектора (BitSet)

- Это один из самых простых способов реализации множеств. Множество реализуется на основании набора битов. Каждый бит соответствует своему элементу (номер бита соответствует элементу, которому этот бит сопоставлен). Если бит равен единице, то данный элемент содержится в множестве. Если нет – не содержится. На рисунке показан битовый вектор для множества {2, 5, 8, 11, 12}.



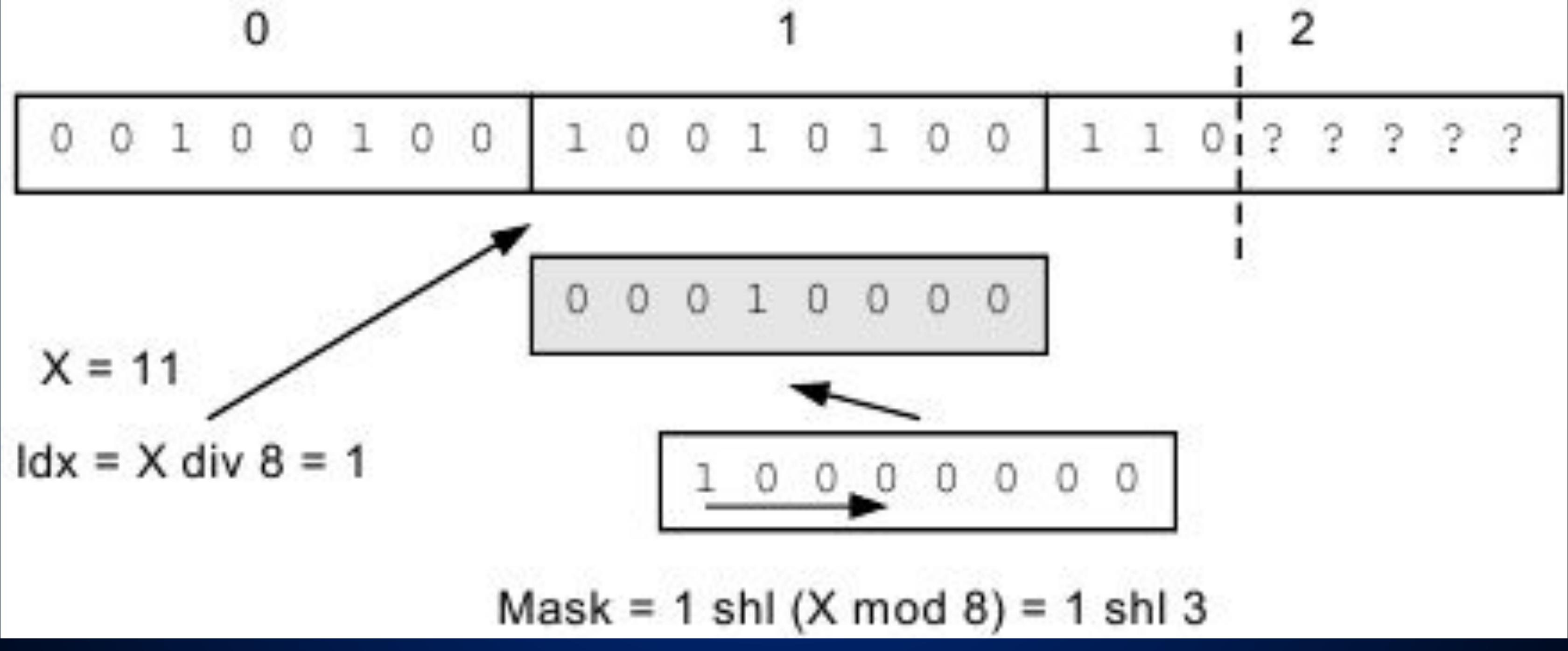
- На рисунке показано соответствие между элементами множества и отдельными битами. Без утраты общности, будем полагать, что все наши элементы множества – натуральные целые числа, включая нуль. Если это не так, то элементы можно соответствующим образом закодировать (например, если среди элементов встречаются отрицательные числа, то вычтем из всех элементов минимально-допустимый элемент).

- Основой реализации станет массив байтов Base[] (для ускорения могут быть выбраны более крупные единицы, соответствующие размеру слова для данной ЭВМ). Пусть элементы множества принадлежат промежутку [0…n], тогда нам потребуется n+1 бит, или (n div 8 + 1) байт. В данном случае мы воспользовались тем свойством, что

$$\left\lceil \frac{x}{a} \right\rceil = \left\lfloor \frac{x+a-1}{a} \right\rfloor$$

- В каждом байте будут лежать биты, соответствующие восьми элементам. Для доступа к биту будем использовать понятие индекса и маски. Индекс – индекс элемента массива, где хранится нужный бит. Маска – байт, содержащий единственный единичный бит в той позиции, где расположен целевой бит.
- Во всех следующих алгоритмах считается, что обрабатываемый элемент x является корректным $0 \le x \le n$.
- Начнем с операции получения по значению элемента x соответствующей маски и индекса элемента. В этой операции (да и в других тоже) младшие 3 бита номера используются как индекс бита внутри байта, а оставшиеся как индекс самого байта.

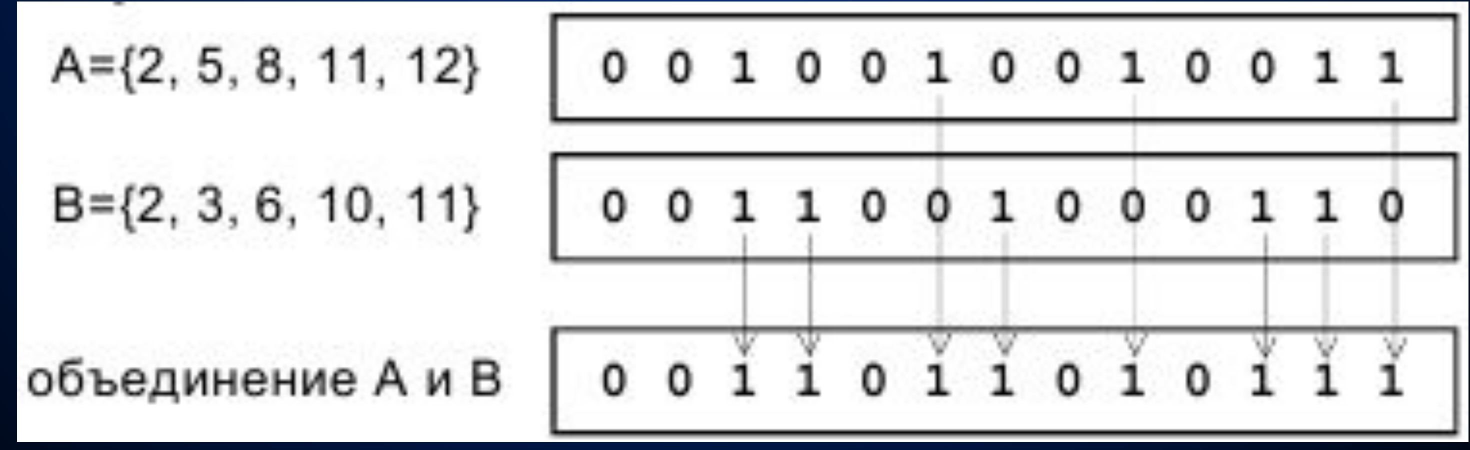
- BitSet.BitAddr(X)
- begin
-   return (<X div 8, 1 shl (X mod 8)>);
-   {return (<X shr 3, 1 shl (X and 7)>)}
- end
-    В скобках приведен вариант с битовыми операциями.



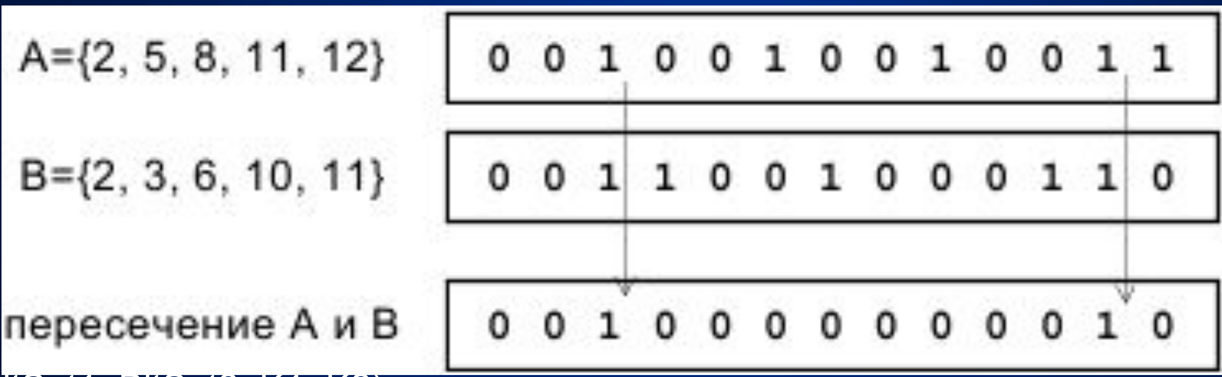
Легко заметить, что младшие 3 бита элемента x используются для вычисления маски по формуле $2^{(x \bmod 8)}$, а старшие биты являются номером элемента в массиве Base[]. Операция выполняется за O(1).

- Операция очистки множества является простой – нужно обнулить все байты массива Base[].
- BitSet.Clear(K)
- begin
- {K – размер массива Base[]}
- for I ← 0 to K do
- Base[I] ← 0;
- end Алгоритм требует времени O(n).
- Алгоритм включения (include) элемента сводится к установке нужного бита.
- BitSet.Include(X)
- begin {Получаем позицию} <Idx, Mask> ← BitSet.BitAddr(X);
- {Устанавливаем бит} Base[Idx] ← Base[Idx] or Mask;
- end
- Исключение бита происходит похожим образом, только бит сбрасывается.
- BitSet.Exclude(X)
- begin {Получаем позицию} <Idx, Mask> ← BitSet.BitAddr(X);
- {Сбрасываем бит} Base[Idx] ← Base[Idx] and (not Mask);
- end
- Также выглядит операция проверки принадлежности – проверяется установлен ли соответствующий бит.
- BitSet.∈(X)
- begin {Получаем позицию}
- <Idx, Mask> ← BitSet.BitAddr(X);
- {Проверяем бит, накладывая маску и сравнивая с нулем}
- return ((Base[Idx] and Mask) ≠ 0);
- end Все эти операции требуют времени O(1).

- Все остальные операции над битовым вектором могут быть произведены за время O(n).
- Операция получения мощности (количества элементов) множества сводится к подсчету количества единичных битов. Для этого, нужно заранее подсчитать количество единиц во всех числах от 0 до 255, пометив их в отдельный массив. Затем следует обработать все элементы массива Base[]. Пусть количество единичных битов в числе a составляет BitCount[a], тогда алгоритм определения мощности множества будет выглядеть так:
- BitSet.|…|(K)
- begin {K – размер массива Base[]}
- Cnt ← 0;
- for I ← 0 to K do
- Cnt ← Cnt + BitCount[Base[I]];
- return (Cnt);
- end
- Операция объединения сводится к тому, чтобы получить новое множество, которое содержит те биты, которые установлены в первом или втором битовом векторе. Для реализации данной операции просто нужно сложить битовые вектора с помощью побитовой операции «или».

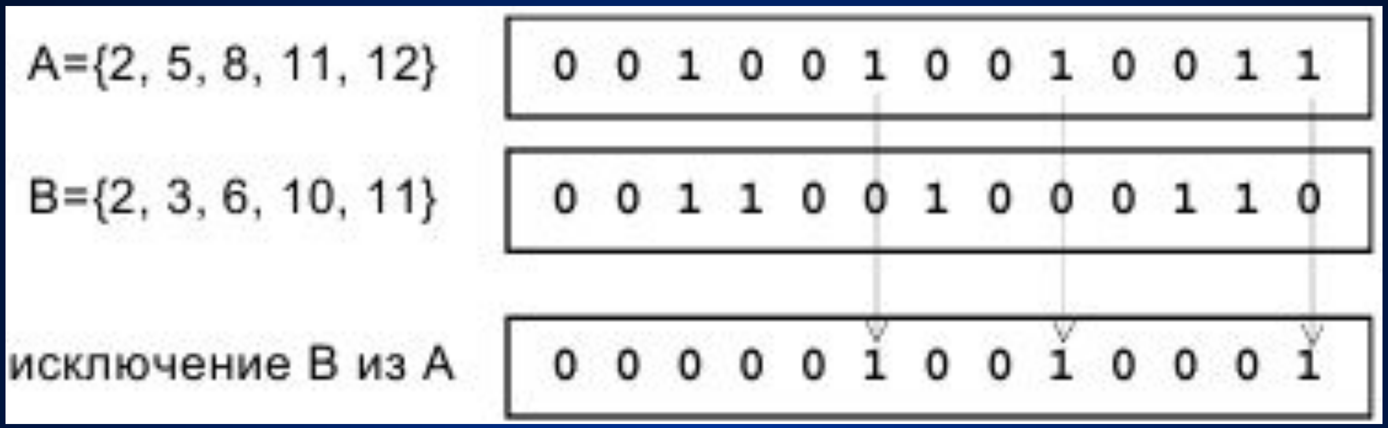


- Алгоритм такой:
- ∪(BitSet1, BitSet2, K1, K2)
- begin
- {K1, K2 – размеры массивов Base[] у множеств}
- {В Base[] множества BitRes содержится max(K1, K2) элементов}
- BitRes.Clear;
- for I ← 0 to min(K1, K2) do
- BitRes.Base[I] ← BitSet1.Base[I] or BitSet2.Base[I];
- return (BitRes);
- End
- Операция пересечения сводится к побитовой операции «и», так как результат содержит элементы, содержащиеся и в том и в другом множестве.



- ∩(BitSet1, BitSet2, K1, K2)
- begin
- {K1, K2 – размеры массивов Base[] у множеств}
- {В Base[] множества BitRes содержится max(K1, K2) элементов}
- BitRes.Clear;
- for I ← 0 to min(K1, K2) do
- BitRes.Base[I] ← BitSet1.Base[I] and BitSet2.Base[I];
- return (BitRes);
- end

- Операция исключения «\» более сложная – требуется оставить биты, которые есть только в первом множестве, но сбросить те, которые есть во втором – перед тем как произвести операцию побитового «и» второй аргумент нужно инвертировать.



- Для инвертирования битов просто используем операцию not.
- \(BitSet1, BitSet2, K1, K2)
- begin
- {K1, K2 – размеры массивов Base[] у множеств}
- {В Base[] множества BitRes содержится max(K1, K2) элементов}
- BitRes.Clear;
- for I ← 0 to min(K1, K2) do
- BitRes.Base[I] ← BitSet1.Base[I] and (not BitSet2.Base[I]);
- return (BitRes);
- end
- Все эти операции выполняются за время O(n).

- Следующие операции предназначены для перебора элементов множества. Простейшая реализация приведена ниже.
- BitSet.Succ(X, N)
- begin
-   {N – максимально возможный элемент множества}
-   repeat
-     {Переходим к следующему}
-     X ← X + 1;
-   until (X > N) or BitSet.∈(X);
-   return (X);
- end
-
- BitSet.Prev(X)
- begin
-   {Минимальный элемент множества – 0}
-   repeat
-     {Переходим к следующему}
-     X ← X – 1;
-   until (X < 0) or BitSet.∈(X);
-   return (X);
- end
- Признаком окончания множества служит возвращение результата большего n в первом случае, и меньшего нуля во втором.

- Для перечисления всех элементов можно просто воспользоваться циклом:
- I ← -1; Т{очно меньший всех элементов множества}
- repeat {Получаем очередной элемент}
- I ← BitSet.Succ(I);
- {Если элемент принадлежит множеству, то обрабатываем}
- if I ≤ N then begin
- {Обработка очередного элемента}
- ...
- end;
- until I > N;
- Сравнение множеств на равенство сводится к поэлементному сравнению массивов Base[].
- =(BitSet1, BitSet2)
- begin {K1, K2 – размеры массивов Base[] у множеств}
- I ← 0; {Сравниваем поэлементно пока один из массивов не кончится}
- while I ≤ min(K1, K2) do begin
- if BitSet1.Base[I] ≠ BitSet2.Base[I]
- then return (FALSE);
- I ← I + 1;
- end;
- {Теперь проверяем хвост более длинного – он должен быть пуст}
- while I ≤ K1 do begin {Первое множество}
- if BitSet1.Base[I] ≠ 0 then return (FALSE);
- I ← I + 1;
- end;
- while I ≤ K2 do begin {Второе множество}
- if BitSet2.Base[I] ≠ 0 then return (FALSE);
- I ← I + 1;
- end;
- {Если до сих пор не вышли, то множества равны}
- return (TRUE);
- end Алгоритм имеет сложность O(n).

Массивы и мультимножества

- Реализация мультимножеств с помощью массивов используется, когда требуется две быстрые операции – добавление элемента и перечисление в произвольном порядке.
- Основная идея данной структуры в том, чтобы поместить все элементы в последовательные ячейки массива, и поддерживать счетчик занятых ячеек (мы с таким уже встречались при обсуждении реализации списка в виде массива).
- Операции принадлежности элемента и извлечения элемента выполняются за O(n), где n – мощность множества. Операции пересечения и объединения выполняются за O(nlog2n) с учетом требования предварительной сортировки массивов. Рассмотрим операции над множествами. Массив будет называться Base[], а количество элементов в нем будет обозначаться Count.
- ArraySet.Clear(K)
- begin
- Count ← 0;
- end
- Добавления элемента в мультимножество просто добавляет элемент в конец массива:
- ArraySet.Include(X)
- begin
- {Получаем позицию}
- Idx ← Count ← Count + 1;
- {Устанавливаем бит}
- Base[Count] ← X;
- end
- Сложность вставки элемента O(1).

- Сложность удаления больше, так как требуется предварительно найти удаляемый элемент в множестве. Для этого используем следующий алгоритм линейного поиска:
- ArraySet.Seek(X)
- begin {Если нашли вернем индекс, если нет - 0}
- for I ← 1 to Count do
- if Base[I] = X then return (I);
- return (0);
- end
- Сложность алгоритма O(n). Исключение приобретает вид:
- ArraySet.Exclude(X)
- begin
- Idx ← ArraySet.Seek(X);
- if Idx > 0
- then begin
- {Перепишем элемент последним}
- Base[Idx] ← Base[Count];
- Count ← Count – 1;
- end;
- end
- Проверка наличия элемента будет выглядеть как проверка наличия элемента в массиве:
- ArraySet.∈(X)
- begin
- return (ArraySet.Seek(X) ≠ 0);
- end

- Получение мощности осуществляется за O(1) простым обращением к счетчику элементов.
- ArraySet.|…|()
- begin
-   return (Count);
- end
- Остальные алгоритмы требуют предварительной сортировки массивов и требуют времени $O(n\log_2 n)$. Сравнение реализуется как сравнение предварительно отсортированных массивов:
- =(ArraySet1, ArraySet2)
- begin {Сравниваем длины}
-   if ArraySet1.Count ≠ ArraySet2.Count
-   then return (FALSE);
-   {Сортируем массивы}
-   QuickSort(ArraySet1.Base[], 1, ArraySet1.Count);
-   QuickSort(ArraySet2.Base[], 1, ArraySet2.Count);
-   {Сравниваем элементы}
-   for I ← 1 to ArraySet1.Count do
-     if ArraySet1.Base[I] ≠ ArraySet2.Base[I]
-     then retrun (FALSE);
-   {Нет несовпадающих элементов}
-   return (TRUE);
- end

- Пересечение множеств также требует предварительной сортировки, но только одного из массивов, логично выбрать меньший из двух:
- ∩(ArraySet1, ArraySet2)
- begin
- {Предполагаем, что ArraySet1.Count < ArraySet2.Count !}
- {Результат пересечения}
- Result.Clear;
- {Сортируем массив}
- QuickSort(ArraySet1.Base[], 1, ArraySet1.Count);
- {Ищем элементы второго в первом}
- for I ← 1 to ArraySet2.Count do begin
- X ← ArraySet2.Base[I];
- {Применяем двоичный поиск}
- Idx ← ArrayBinarySearch(ArraySet1.Base[], ArraySet1.Count);
- if (Idx ≤ ArraySet1.Count) and (ArraySet1.Base[Idx] = X)
- then Result.Include(X);
- end;
- return (Result);
- end
- Пусть мощность меньшего множества n, а мощность большего – m. Сортировка займет O(nlog2n) шагов, а стадия поиска O(mlog2n), получим оценку O((m+n)log2n). Теперь должно быть понятно, почему сортировать нужно меньший массив – сумма включает длину обоих массивов, а логарифм – только одного из них. Если предварительно отсортировать оба массива, то оценка будет хуже – $O(n\log_2 n + m\log_2 m)$, а так как $n \le m$, то и $\log_2 n \le \log_2 m$.

- Объединение множеств не требует сортировки, а заключается просто в переписывании всех элементов подряд.
- ∪ (ArraySet1, ArraySet2)
- begin
- Result.Clear;
- for I ← 1 to ArraySet1.Count do
- Result.Include(ArraySet1.Base[I]);
- for I ← 1 to ArraySet2.Count do
- Result.Include(ArraySet2.Base[I]);
- return (Result);
- end   Алгоритм имеет сложность O(n+m).
- Исключение подмножества из множества может быть осуществлено за время $O((n+m)\log_2 m)$. Для достижения этой оценки следует отсортировать массив с исключаемыми элементами и использовать двоичный поиск.
- \(ArraySet1, ArraySet2)
- begin   {Результат исключения}
- Result.Clear;
- {Сортируем массив}
- QuickSort(ArraySet2.Base[], 1, ArraySet2.Count);
- {Ищем элементы второго в первом}
- for I ← 1 to ArraySet1.Count do begin
- X ← ArraySet1.Base[I];
- {Применяем двоичный поиск}
- Idx ← ArrayBinarySearch(ArraySet2.Base[], ArraySet2.Count);
- if (Idx > ArraySet1.Count) or (ArraySet1.Base[Idx] ≠ X)
- then Result.Include(X);
- end;
- return (Result);
- end   Сложность алгоритма $O(n+m\log_2 m)$.

Массивы и вектора

- Данная структура совмещает в себе битовое множество и массив, что улучшает оценки для многих алгоритмов, но за счет дополнительной памяти.
- В этом разделе как n будем обозначать мощность области значений элементов множества, и как m мощность самого множества. В случае, когда $m << n$ применение данной структуры данных можно считать оправданным, несмотря на дополнительный расход памяти $O(m)$.
- Опишем основные операции, используя внутреннее множество BitSet, массив элементов Items[], и счетчик элементов Count. Будем использовать для BitSet только операции включения, исключения и проверки принадлежности, поэтому предположим, что BitSet реализовано как битовый вектор.

```
MixSet.Clear()
begin
  Count ← 0;   BitSet.Clear;   {BitSet ← ∅}
end
```

- Операция добавления и удаления элементов по логике совпадают с реализацией для массива, но упрощены за счет использования битового вектора.

```
MixSet.Include(X)
begin   {Проверим, нет ли уже такого}
  if not BitSet.∈(X)     {X ∉ BitSet}
  then begin
    {Такого элемента еще нет}
    BitSet.Include(X);   {BitSet ← BitSet ∪ {X}}
    Count ← Count + 1;
    Items[Count] ← X;
  end;
end
```

- Поиск элемента немного упрощается для тех элементов, которых в множестве нет.
- MixSet.Seek(X)
- begin
- if BitSet.∈(X) {X ∈ BitSet}
- then begin
- for I ← 1 to Count do
- if Items[I] = X
- then return (X);
- end;
- return (0);
- end Легко заметить, что поиск по прежнему имеет оценку O(m) для элементов присутствующих в множестве, и O(1) для отсутствующих.
- Исключение элемента выполняется в два этапа – поиск удаляемого элемента, и исключение его из массива и множества.
- MixSet.Exclude(X)
- begin
- Idx ← MixSet.Seek(X);
- if Idx > 0 then begin
- {Исключаемый элемент есть}
- BitSet.Exclude(X); { BitSet ← BitSet \ {X}}
- Items[Idx] ← Items[Count]; Count ← Count – 1;
- end;
- end Исключение элемента выполняется за O(m).

- Исключение элемента с известным номером происходит за O(1).
- MixSet.Exclude(Idx)
- begin
- if BitSet.∈(Items[Idx])
- then begin
- {Исключаемый элемент есть}
- BitSet.Exclude(X); { BitSet ← BitSet \ {X}}
- Items[Idx] ← Items[Count];
- Count ← Count – 1;
- end;
- end
- Принадлежность определяется за O(1) с помощью битового вектора.
- MixSet.∈(X)
- begin
- return (BitSet. ∈(X)); {X ∈ BitSet}
- end
- Мощность множества вычисляется за O(1) с помощью счетчика элементов.
- MixSet.|…|
- begin
- return (Count);
- end

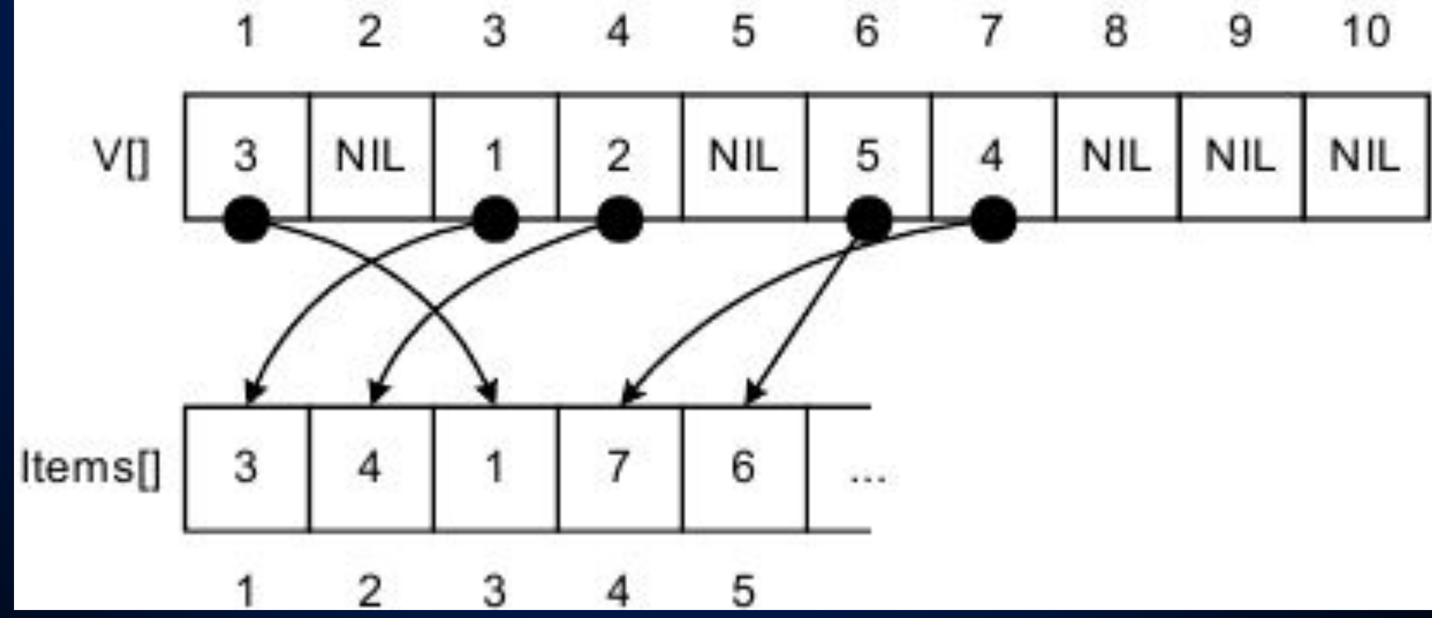
- Операции пересечения и объединения значительно ускоряются за счет ускорения операции проверки принадлежности элемента. При реализации этих операций используется уже не внутренние особенности структуры данных, а число логический подход, основанный на теории множеств.
- ∪(MixSet1, MixSet2)
- begin {К элементам MixSet1 добавляем элементы MixSet2}
- Result ← MixSet1;
- for I ← 1 to MixSet2.Count do
- Result.Include(MixSet2.Items[I]);
- return (Result);
- end
- Стадия копирования первого множества займет O(m1), где m1 – мощность первого множества, а стадия добавления элементов второго O(m2), так что общая оценка операции объединения будет O(m1+m2).
- Пересечение, по сравнению с простым массивом, тоже ускоряется:
- ∩(MixSet1, MixSet2)
- begin
- Result.Clear;
- if MixSet1.Count < MixSet2.Count {Выбираем меньшее из множеств}
- then begin
- {Копируем только те элементы MixSet1 которые есть в MixSet2}
- for I ← 1 to MixSet1.Count do
- if MixSet2.∈(MixSet1.Items[I]) then Result.Include(MixSet1.Items[I]);
- end
- else begin {Копируем только те элементы MixSet2 которые есть в MixSet1}
- for I ← 1 to MixSet2.Count do
- if MixSet1.∈(MixSet2.Items[I]) then Result.Include(MixSet2.Items[I]);
- end;
- return (Result);
- end Такой алгоритм будет работать за O(min(m1, m2)).

- Алгоритм исключения подмножества использует похожий принцип:
- \(MixSet1, MixSet2)
- begin
- Result.Clear;
- {Копируем те элементы MixSet1 которых нет в MixSet2}
- for I ← 1 to MixSet1.Count do
- if not MixSet2.∈(MixSet1.Items[I])
- then Result.Include(MixSet1.Items[I]);
- return (Result);
- end Сложность алгоритма O(m1).

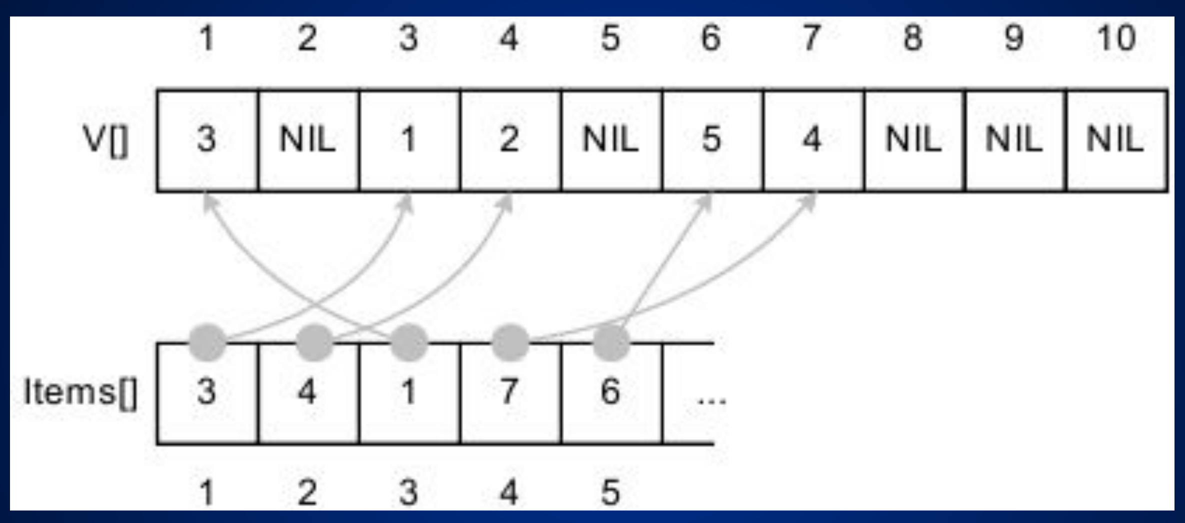
- Сравнение множества на равенство сводится к проверке условия – каждый элемент из первого множества содержится во втором, и мощности множеств равны.
- =(MixSet1, MixSet2)
- begin {Проверяем равенство мощностей}
- if MixSet1.Count ≠ MixSet2.Count
- then return (FALSE);
- {Копируем только те элементы MixSet1 которые есть в MixSet2}
- for I ← 1 to MixSet1.Count do
- if not MixSet2.∈(MixSet1.Items[I]) then return (FALSE);
- return (TRUE);
- end Сложность алгоритма O(m).

Cross-массивы

- Это структура данных, которая получается из описанной в предыдущей главе, если заменить в ней битовый вектор на массив ссылок. То есть для элемента i заводится ячейка V[i], в которой хранится указатель на позицию элемента i в массиве Items[]. Если в нашем множестве элемент i отсутствует, то V[i] хранит пустой указатель, то есть NIL. Например, для подмножества 1, 3, 4, 6, 7 множества 1..10 описываемая структура может выглядеть, например, так.



- С другой стороны, элементы массива Items[] в свою очередь можно рассматривать как ссылки на элементы массива V[], которые содержат ссылки, отличные от NIL. Эту интерпретацию легко представить, если на предыдущем рисунке развернуть все стрелки в противоположную сторону



Понятно, что структура множества на cross-массивах будет содержать по одному элементу массивов V[] и Items[], ссылающемуся на другой, а все множество окажется разбитым на пары указывающих друг на друга элементов (отсюда и «cross» в названии). Математически, эти отношения можно выразить двумя следующими утверждениями (мы будем пользоваться ими при составлении алгоритмов): Items[V[x]] = x
V[Items[x]] = x

- Заметьте, что такая структура не подходит для хранения мультимножества, так как существует только одна ссылка из массива V[]. Конечно, ничего не мешает нам добавить массив Next[] и связать все одинаковые элементы массива Items[] в односвязный список. Однако это сильно усложнит операцию удаления элемента и сведет на нет все получаемые нами преимущества.
- Теперь опишем основные алгоритмы для работы с множествами, реализованными на основе cross-массивов. Начнем как всегда с операции инициализации (очистки). Эта операция сводится к обнулению счетчика элементов массива Items[] и заполнению массива V[] пустыми значениями NIL.

```
CrossSet.Clear()
begin
  Count ← 0;
  for I ← Low to High do
    V[I] ← NIL;
end
```

- Здесь Low и High означают диапазон значений, которые могут обслуживаться множеством. В дальнейшем, для упрощения кода, мы будем считать, что все обрабатываемые нами значения лежат именно в интервале [Low, High], за исключением значения NIL. Это значение следует выбрать как раз таким способом, чтобы оно не попадало в интервал [Low, High].

- Операция добавления элемента в множество сводится к проверке наличия добавляемого элемента, и ели его нет, то добавлению его к концу массива Items[].
- CrossSet.Include(X)
- begin
-   if V[X] ≠ N IL   {Элемента X еще нет в множестве}
-   then begin
-     {Добавляем новый элемент и сохраняем ссылкку на него}
-     V[X] ← Count ← Count + 1;
-     Items[Count] ← X;
-   end;
- end
-   Поиск элемента x в множестве сводится к простой проверке ячейки V[x], и может быть описан таким кодом:
- CrossSet.Seek(X)
- begin
-   return(V[X]);
- end
-   Аналогично, операция проверки принадлежности элемента множеству может быть выражена очень лаконично:
- CrossSet.∈(X)
- begin
-   return (V[X] ≠ NIL);
- end

- Исключение элемента x из множества тоже значительно упрощается за счет использования массива V[], только нужно не забыть заполнить ячейку V[x] значением NIL.
- CrossSet.Exclude(X)
- begin
- Idx ← V[X];
- if Idx ≠ N IL {X принадлежит множеству}
- then begin {Перезаписываем на это место последний элемент Items[]}
- Items[Idx] ← Items[Count];
- {Коректируем ссылку в V[] – поддерживаем свойства}
- V[Items[Idx]] ← Idx;
- V[X] ← NIL; {Обнуляем ссылку на удаленный элемент}
- Count ← Count – 1; {Один элемент был удален}
- end;
- end
- Мощность множества тоже может быть вычислена одной строкой – через поле Count:
- CrossSet.|…|
- begin return (Count);
- end
- Очевидно, сложность всех описанных алгоритмов, кроме очистки, составляет O(1). Операции объединения, пересечения, сравнения и прочих будут совершенно аналогичны этим операциям, описанным в предыдущем разделе и будут иметь ту же самую сложность.

Реализация множеств с использованием связных списков

```
type celltype = record
        element: elementtype; {значение атома}
        next: ^celltype {указатель на следующий элемент}
    end
INTERSECTION – ищет пересечение списков. А и В считаются упорядоченными.
procedure INTERSECTION ( ahead, bhead: Tcelltype; var pc: Tcelltype );
var acurrent, bcurrent, ccurrent: Tcelltype;
begin
(1) new (pc) ; { создание заголовка списка }
(2) acurrent:= ahead^.next; (3) bcurrent:= bhead^.next; (4) ccurrent:= pc;
(5) while (acurrent <> nil) and (bcurrent <> nil) do begin
{ сравнение текущих элементов списков А и В }
(6) if acurrent^.element = bcurrent^.element
    then begin { если оба совпали, то добавим атом в пересечение}
      (7) new(ccurrent^.next);     (8) ccurrent:= ccurrent^.next;
      (9) ccurrent^.element:= acurrent^.element;
      (10) acurrent:= acurrent^.next;    (11) bcurrent:= bcurrenti. next
    end
   else { элементы не равны}
         if acurrent^.element < bcurrent^.element
         then acurrent:= acurrent^.next;
         else  bcurrent:= bcurrent^.next
  end;
(15) ccurrent^. next: = nil
end; { INTERSECTION }
```

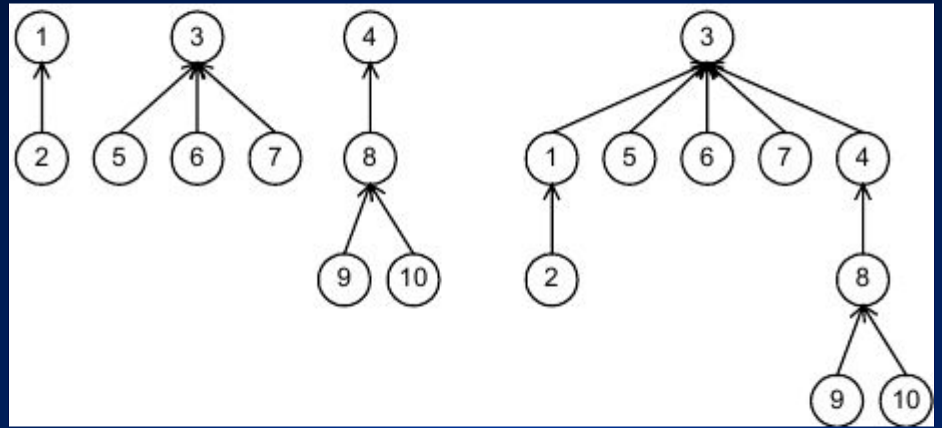
Вставка элемента в множество

```
procedure INSERT ( x: elementtype; p: tcelltype );
var current, newcell: Tcelltype;
begin
current: = p;
while currenti. next <> nil do begin
   if currentt.nextt. element = x
   then exit; {элемент уже есть}
   if current^.next^. element> x
   then begin {добавление нового элемента}
           new(newcell) ; newcell^. element := x;
           newcell^.next:= current^.next;
           current^.next;= newcell; exit
        End;
   current;= current^.next
end;
end; { INSERT }
```

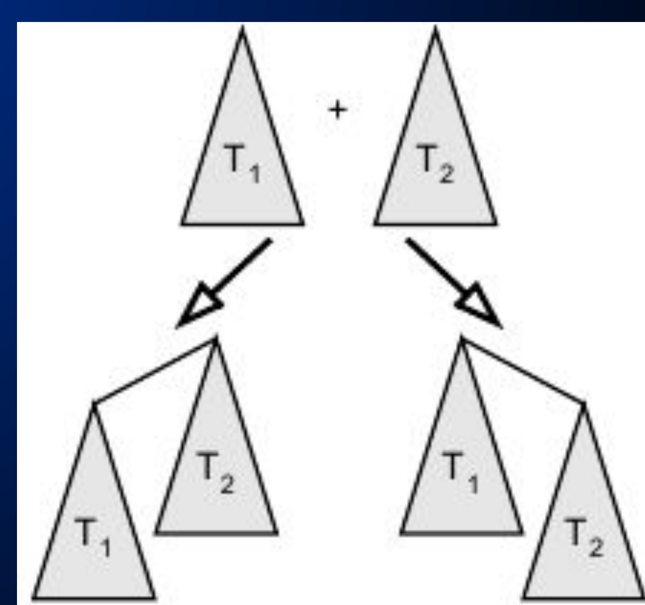
Системы непересекающихся множеств

- Во многих алгоритмах понадобятся системы множеств, которые не имеют общих элементов. Над этими множествами будут выполнятся только две основные операции – определение, какому множеству принадлежит данный элемент и слияние (объединение) множеств. Так же возможна операция проверки принадлежности двух элементов одному и тому же множеству.
- Изначально, каждый элемент содержится в множестве с самим собой в качестве единственного элемента. Операция добавления элемента не используется – считается что все элементы добавляются на стадии инициализации структуры. В процессе работы множества будут объединяться.
- Такие множества реализуются с помощью корневых деревьев. Корень дерева называется представителем множества.
- Пусть у нас есть изначально есть n элементов, пронумерованных числами от 1 до n. Для представления множеств в виде корневых деревьев нам потребуется массив ссылок на предков Parent[], из n элементов. Parent[i] хранит номер вершины-предка для узла i, если узел i не имеет предка (является представителем), то Parent[i] = i. Кроме этого, введем массив W[] из n элементов, такой что W[i] хранит количество вершин в поддереве с корнем i.
- Процедура инициализации сводится к тому, чтобы разложить все элементы по поддеревьям, состоящим из одного элемента.

```
RTreeSet.Init(N)
begin
  for I ← 1 to N do begin
    {Изначально все узлы явлются представителями самих себя}
    Parent[I] ← I;
    W[I] ← 1;
  end;
end
```

- Слияние подмножеств сводится к тому, что представитель одного из них делается потомком представителя другого, т.е. одно дерево присоединяется к другому.
- 

- Очевидно, что при слиянии двух множеств можно их сливать двумя способами: присоединить в качества поддерева первое или второе.



- Если мы всегда будем присоединять деревья одинаковым способом, то может получиться вырожденное дерево (вытянутое в список). Для того, чтобы избежать данной ситуации и хранится количество узлов поддерева W[i] в вершине-представителе (корне поддерева). Мы всегда будем добавлять меньшее дерево к большему, тогда слияние двух множеств, заданных их представителями, описывается так:
- RTreeSet.MergeRep(R1, R2)
- begin
- {Проверим, что R1 и R2 - представители}
- if (Parent[R1] ≠ R1) or (Parent[R2] ≠ R2)
- then ◊ Один из элементов не является представителем;
- {Выбираем представителя с меньшим весом}
- if W[R1] < W[R2]
- then begin
- {Подвешиваем R1 к R2}
- Parent[R1] ← R2;
- W[R2] ← W[R2] + W[R1];
- end else begin
- {Подвешиваем R2 к R1}
- Parent[R2] ← R1;
- W[R1] ← W[R1] + W[R2];
- end;
- end

Нахождение представителя для данного элемента сводится к нахождения корня дерева, в котом расположен элемент. Так как структура самого дерева нам не важна, то целесообразно применить описанный ранее алгоритм сжатия путей. С учетом этого алгоритма, получение представителя описывается следующим алгоритмом.

```
RTreeSet.GetRep(I)
begin   {Проверим, не является ли она потомком корня}
  {если нет – используем сжатие путей}
  if Parent[I] ≠ I   then Parent[I] ← GetRep(Parent[I]);
  {Теперь вершина точно потомок корня}
  return (Parent[I]);
end
```

Заметим, что процедура сжатия путей не поддерживает массив W[] в актуальном состоянии, но значения W[] для корней верны, а так как слияние производится только для корней, то процедуру можно использовать. У элементов одного множества один единственный представитель. Теперь можно написать процедуру определения являются ли элементы членом одного подмножества.

```
RTreeSet.SameSet(I1, I2)
begin   return (GetRep(I1) = GetRep(I2));
end
```

Процедура слияния для двух произвольных элементов также использует такую проверку и нахождение представителей.

```
RTreeSet.Merge(I1, I2)
begin   {Получаем представителей}
  R1 ← GetRep(I1);   R2 ← GetRep(I2);
  {И слияние, если они разные}   if R1 ≠ R2   then MergeRep(R1, R2);
end
```

Оценивать данные алгоритмы довольно сложно, поэтому здесь не будет приводиться получение оценки – только результат.

Применение систем непересекающихся множеств

- Одним из классических применений систем непересекающихся множеств является построение классов эквивалентности по отношениями между отдельными элементами.
- Напомним, что эквивалентностью (≡) называется отношение, обладающее свойствами транзитивности, рефлективности и симметричности. Например, типичным отношением эквивалентности является отношение равенства. Отношение «принадлежат к одному множеству» также является эквивалентностью.
- Транзитивность это свойство, утверждающее, что если $a \equiv b$ и $b \equiv c$, то и $a \equiv c$. Именно это свойство отношения «принадлежат к одному множеству» и используется при разбиении на классы эквивалентности.
- Любое множество S разбивается на непересекающиеся классы
- эквивалентности $S = S1 \cup S2 \cup \ldots \cup Sk$. Элементы любого класса Si эквивалентны между собой (это гарантирует транзитивность), и неэквивалентны ни с одним другим элементом. То, что классы Si и Sj ($i \neq j$) не пересекаются ($Si \cap Sj = \varnothing$) гарантирует та же транзитивность, ведь если существует такой элемент $a \in Si$ и $a \in Sj$, то и все элементы Si и Sj эквивалентны между собой.
- По сути, отношениями эквивалентности являются очень многие свойства. Например, эквивалентностью являются такие распространенные отношения как отношения равенства, сравнимости по модулю или, например, отношения «одного цвета» и «принадлежат одному множеству».

- Задача разбиения на классы эквивалентности формулируется так: дано n различных объектов, и отношение эквивалентности этих объектов «≡». На вход алгоритму подаются пары эквивалентных объектов. На выходе необходимо получить множества эквивалентных между собой объектов.
- Для решения данной задачи, потребуется использовать систему непересекающихся множеств. На начальном этапе все объекты эквивалентны только сами себе. Предъявление пары эквивалентных объектов ведет к слиянию множеств, в которых расположены сливаемые объекты. Алгоритм выглядит так:

```
EquSplit(N, Pare[], M)
begin   {Массив Pare[] – M пар номеров эквивалентных элементов}
  {Инициируем множество}
  RTreeSet.Init(N);
    for I ← 1 to M do begin {Обрабатываем массив}
    <I1, I2> ← Pare[I];
    RTreeSet.Merge(I1, I2);
  end;
K ← 0; {Теперь нумеруем элементы. S[i] – номер класса элемента i}
 for I ← 1 to N do
    S[I] ← 0 ;   {Класс еще не пронумерован}
  for I ← 1 to N do begin
    {Получаем представителя для элемента I}
    R ← RTreeSet.GetRep(I);
    if S[R] = 0    {Пронумерован ли этот класс?}
    then S[R] ← K ← K + 1;
    {Номер элемента совпадает с номером представителя}
    S[I] ← S[R];
  end;
  return (S[]);
end
```

- На выходе алгоритм выдает массив S[], элемент S[i] содержит номер класса эквивалентности, к которому принадлежит элемент.

Словари

- Словари поддерживают извлечение по содержанию, а не по положению, что делают стеки и очереди. Словари поддерживают три основные операции:
- Insert(x,d) - вставить объект x в словарь d;
- Delete(x,d) - удалить объект x (или объект, на который указывает x) из словаря d;
- Search(k,d) - вернуть объект с ключом к, если таковой имеется в словаре d.
- В направленных структурах данных можно предложить десятки способов реализации словарей, включая отсортированные/несортированные связанные списки, отсортированные/несортированные массивы, и целый лес, полный случайных, однонаправленных (AVL) и черно-красных деревьев; не говоря обо всех вариациях хеширования. Основным объектом анализа алгоритмов является производительность, точнее, достижение лучшего возможного компромисса между этими тремя операциями. Но на практике мы обычно хотим получить простейший путь решения проблемы, удовлетворяющий ограничениям по времени. Корректный выбор реализации зависит от того, насколько сильно меняется содержимое вашего словаря в процессе выполнения.

- Статические словари. Эти структуры строятся один раз и никогда не меняются. Таким образом, они должны поддерживать поиск, но не вставку и удаление. Правильным выбором для статического словаря обычно является массив. Единственным важным вопросом будет вопрос о том, нужно ли держать его отсортированным, чтобы использовать бинарный поиск для быстрой обработки запросов. Если у вас нет жестких временных ограничений, то, вероятнее всего, не имеет смысла использовать бинарный поиск до тех пор, пока n не превысит 100 или около того. Возможно, вы сумеете обойтись последовательным поиском до n = 1000 или более при условии, что вам не нужно будет проводить слишком много поисков.

- Полудинамические словари. Эти структуры поддерживают поиск и вставку, но не удаление. Если мы знаем верхний предел числа элементов, которые могут быть вставлены, мы можем использовать массив, иначе мы должны использовать связанные структуры. Хеш-таблицы являются превосходными структурами данных для словарей, в особенности если не требуется поддерживать удаление. Идея состоит в применении функции к поисковому ключу, так что мы можем определить, где объект появится в массиве, не просматривая остальные объекты. Чтобы создать таблицу разумного размера, мы должны учесть коллизии, когда два различных ключа привязаны к одной ячейке. Два компонента хеширования - это
- А) определение хеш-функции, которая привяжет ключи к целым числам в определенном диапазоне, и
- В) создание массива, чей размер соответствует этому диапазону, так чтобы значение хеш-функции означало индекс. Простая хеш-функция превращает ключ в целое число и берет значение, равное целочисленному остатку от деления этого числа на размер хеш-таблицы. Выбор простого числа в качестве размера таблицы (или, по крайней мере, отказ от выбора очевидных составных чисел, таких, как 1000) помогает избежать проблем.

- Строки могут быть переведены в целые числа, если использовать буквы алфавита в качестве цифр системы счисления с основанием, равным длине алфавита. Чтобы перевести слово «steve» в число, заметим, что e - это 5-я буква алфавита, s - это 19-я буква, t - это 20-я буква и v - это 22-я буква. Таким образом, «steve» $=26^4 x19+26^3 x20+26^2 x5+26^1 x22+26^0 x5=9\ 038\ 021$. Первые, последние или средние 10 символов или около того, вероятно, подойдут для хорошего индекса.
- Отсутствие необходимости удаления делает открытую адресацию простым, удобным способом для разрешения конфликтов. При открытой адресации мы используем простое правило для решения того, куда положить новый объект, если желаемое место уже занято. Пусть мы всегда кладем его в следующую незанятую ячейку. При поиске данного объекта мы идем в предназначенное место и начинаем последовательный поиск. Если мы обнаруживаем пустую ячейку до того, как обнаруживаем объект, то он не существует в таблице. Удаление в схеме с открытой адресацией неприемлемо, поскольку удаление одного элемента может сломать цепочку вставок, сделав некоторые элементы недоступными. Ключ к эффективности лежит в выборе достаточно большой таблицы, в которой будет много свободного места. Не жадничайте, когда выбираете размер таблицы, иначе потом придется платить дороже.

- Полностью динамические словари. Хеш-таблицы также удобны для реализации полностью динамических словарей при условии, что мы используем формирование цепочки данных при разрешении конфликтов. В данном случае с каждой позицией в хеш-таблице мы связываем связанный список, так что задачи вставки, удаления и запросов сводятся к аналогичным задачам для связанных списков. Если хеш-функция работает хорошо, то m ключей будут равномерно распределены по таблице размера n, так что каждый список окажется достаточно коротким и поиск будет быстрым.

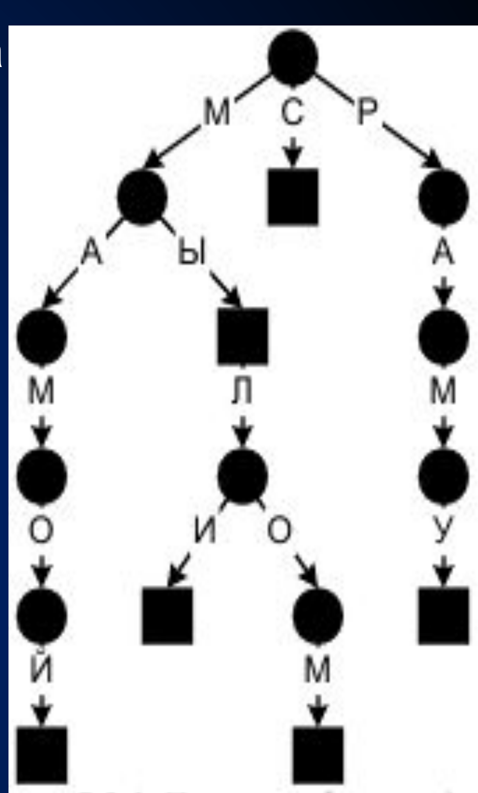
Словари

- Словари могут быть реализованы по средством упорядоченного или неупорядоченного списка, двоичных массивов (предполагая, что элементами являются числа 1..N). Третий вариант – использование статического линейного списка на основе массива.
- const
- maxsize = { максимальный размер словаря }
- type DICTIONARY=record
-                     last: integer;
-                     data: array[1 .. maxsize] of nametype
- end;
- procedure MAKENULL (var A: DICTIONARY);
- begin
-     A.last:=0
- end; {MAKENULL }

```
function MEMBER ( x: nametyp,; var A: DICTIONARY ): boolean;
var i: integer;
begin
   for i:= 1 to A.last do
       if A.data[i] = x then begin MEMBER :=true; exit end;
   MEMBER :=false { элемент не найден}
end; {MEMBER}
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(' словарь заполнен')
end; {INSERT }
procedure DELETE ( x: name type ; 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] ; {удаление с нарушением порядка следования элементов}
       A.last:= A.last - l
    end {while}
  end {then}
end; ( DELETE }
```

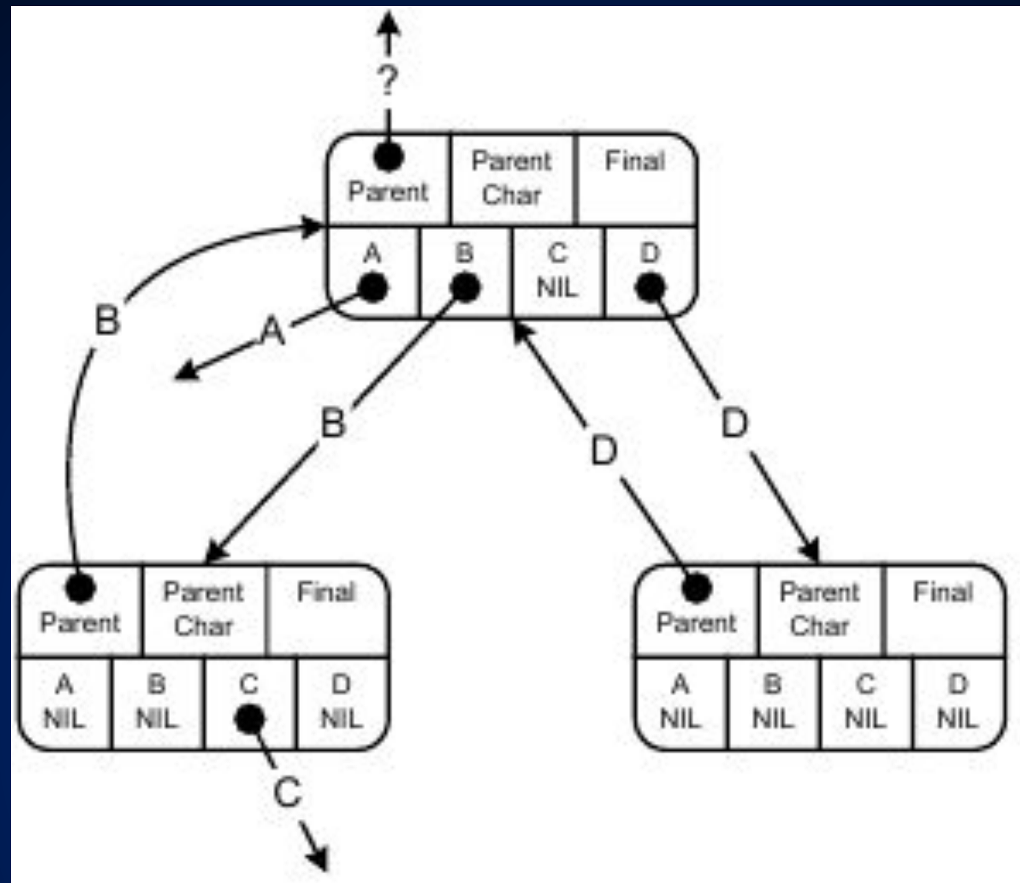
Цифровые деревья (Digit tree, Radix tree)

- Чаще всего цифровые деревья используются для организации различных словарей. Принцип организации узла такого дерева очень прост, и напоминает записную книжку с алфавитными вырезами.
- Пусть нам нужно построить цифровое дерево для множества слов. Под словом будем понимать последовательность знаков ai, принадлежащих некоторому алфавиту A = {ai}. Этими символами мы будем маркировать дуги, исходящие из узла дерева. Слово в цифровом дереве будет представлять собой путь от вершины дерева до некоторого узла. На рисунке показан пример цифрового дерева для слов из фразы «мы с мамой мыли раму мылом».
- Будем говорить, что вершина цифрового дерева соответствует некоторому слову, если метки на путь от корня до этой вершины образуют это слово. Узлы цифрового дерева, которым не соответствуют слова из словаря, называются фиктивными. На рисунке они круглые.
- Действительными или конечными (не путать с листовыми) узлами называют узлы, путь до которых из корня представляет собой некоторое слово из словаря. На рисунке конечные вершины – квадратные. Заметьте, что любой лист дерева – конечная вершина, но не любая конечная вершина – лист. Например, узел соответствующий слову «мы» является концевым, но не является листом.



Простейшая реализация цифрового дерева

- При простейшей реализации цифрового дерева, в каждой вершине отводится столько ссылок на потомков, сколько букв в алфавите исходном алфавите. Обычно ссылки индексируются этими же буквами. Таким образом, вместо массивов ссылок Left[] и Right[] используется массив массивов ссылок Childs[][].
- RadixTree.Root Индекс корневой вершины
- Final[] Признак того, является этот узел дерева конечным или нет
- Childs[][] Массив массивов ссылок на потомков. Первый индекс номер узла в дереве, второй – буква алфавита, которой соответствует ссылка на потомка
- Parent[] Ссылка на предка в цифровом дереве
- ParentChar[] Буква алфавита, которой помечена ссылка из предка в данный узел цифрового дерева
- Структура узла такого дерева показана на следующем рисунке. На примере предполагается, что алфавит состоит из 4 букв A, B, C, D.



- Методы работы с цифровым деревом указаны в следующей таблице.
- RadixTree.Init Инициализирует цифровое дерево
- RadixTree.NewNode Создает и инициализирует новую вершину в дереве
- RadixTree.AddWord(W) Добавляет в дерево новое слово W
- RadixTree.Delete(W) Удаляет из дерева слово W
- RadixTree.Search(W) Ищет в дереве вершину соответствующую слову W
- RadixTree.Word(Idx) Составляет слово по вершине

Во всех наших алгоритмах, мы будем обозначать как A{} множество всех символов алфавита, с которым работает наше дерево. Все массивы мы будем полагать глобальными, и доступными для реализации нескольких цифровых деревьев одновременно. Как всегда мы не будем здесь описывать процедуры удаления и добавления самих узлов – для этого можно воспользоваться той же техникой, что и при работе с обычными списками (списки пустых блоков и т.п.). В описанных алгоритмах используется просто Прежде всего, опишем алгоритм создания новой вершины. При создании вершины требуется только заполнить ее ссылки на потомков пустыми (NIL).

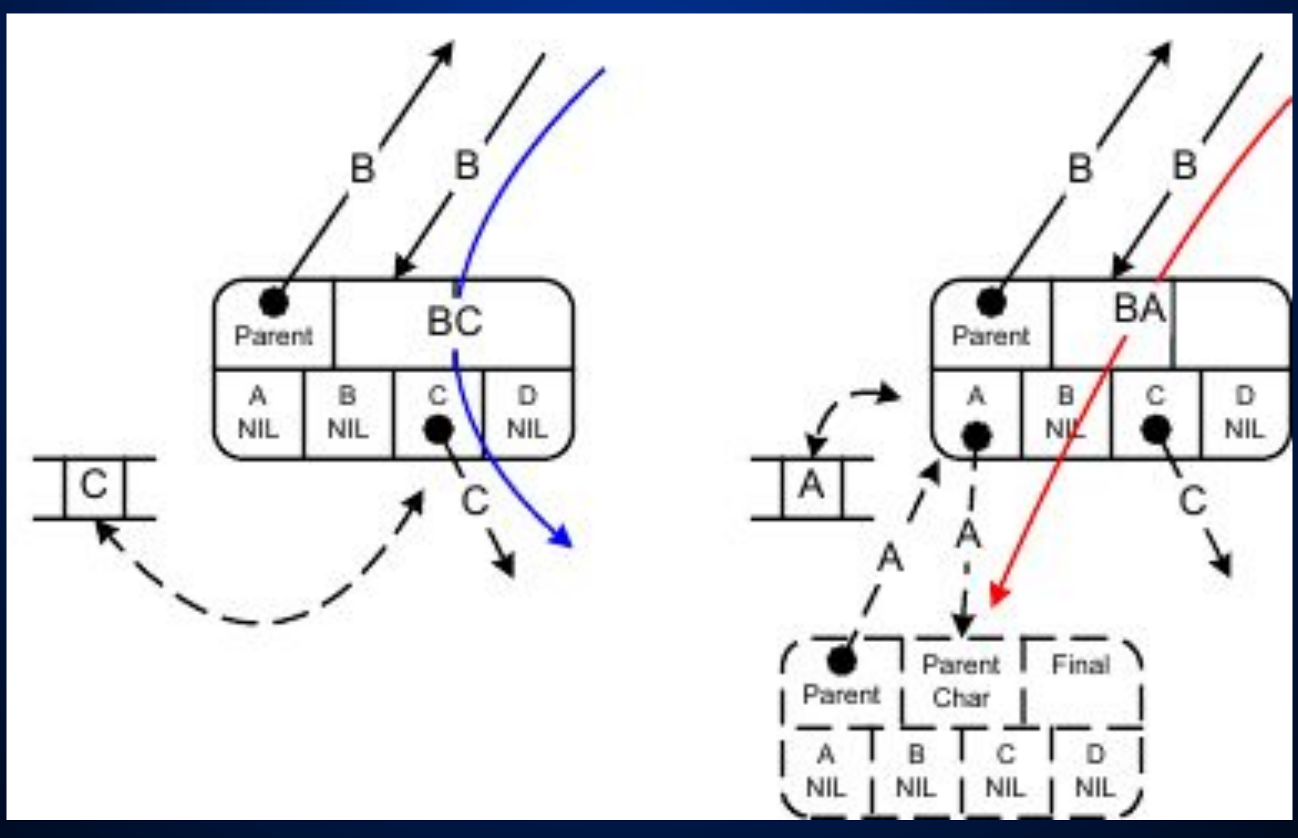
```
RadixTree.NewNode()
begin   I ← Новый индекс (в простом случае Count ← Count + 1)
  for Ch ∈ A{} do
    Childs[I][Ch] ← NIL;
  Final[I] ← FALSE;
  return (I);
end
```

Процедура инициализации дерева выглядит проще всего. Для этого нужно создать корневую вершину и сделать все ее ссылки пустыми.

```
RadixTree.Init()
begin
  Root ← NewNode();   Н{овый блок для корня}
  Parent[Root] ← NIL;  {У корня нет предка}
  Final[Root] ← TRUE;
end
```

Заметьте, что, так как корневая вершина всегда присутствует в дереве, то дерево всегда содержит пустое слово. Мы обозначаем этот факт, взводя для корня признак концевой вершины. Это небольшое изменение никак не повлияет на алгоритмы добавления, но несколько упростит удаление из дерева – признак концевой вершины у корня сыграет роль барьера.

- Теперь опишем алгоритм добавления нового слова к цифровому дереву. Алгоритм очень прост – нужно бежать по слову от начала к концу и переходить по ссылкам пока они есть.
- Алгоритм (как впрочем, и все остальные) оперирует понятиями текущего узла и текущей буквы. Текущие буквы изменяются слева направо по слову, а текущие узлы – сверху вниз по дереву. При рассмотрении каждой буквы выбирается некоторый (возможно новый) узел, и делается шаг вниз.
- При спуске по цифровому дереву возможны всего две ситуации: из данного узла есть ссылка, помеченная обрабатываемой буквой, или такой ссылки нет (соответствующее поле содержит NIL). Если ссылки нет, то нужно добавить новый узел – потомка текущего, и пройти в него. Если ссылка уже есть – просто пройти по ней. Обе эти ситуации показаны на рисунке.



- RadixTree.AddWord(W#)
- begin
-   C ← R oot;   {C – текущий узел}
-   for I ← 1 to Length(W#) do
-     if Childs[C][W[I]] ≠ NIL {Ссылка есть}
-     then C ← Childs[C][W[I]]
-     else begin
-       {Ссылки нет – нужно создать новую вершину}
-       J ← NewNode();
-       {И заполнить ее для символа W[I]}
-       Parent[J] ← Cur;
-       ParentChar[J] ← W[I];
- Childs[C][W[I]] ← J;
-       C ← J;
-     end;
-   Final[C] ← T RUE;   Новая конечная вершина}
-   return (C);
- end
-    Сложность алгоритма добавления в цифровое древо пропорциональна O(LW*|A{}|), где LW – длина слова W. Мощность алфавита появляется из-за того, что необходимо инициализировать каждую вершину пустыми ссылками.

- Процедуру добавления можно несколько ускорить, если заметить, что после того как добавлен первый новый узел, в дальнейшем нет смысла проверять существование ссылок – в новых вершинах они всегда пусты. Это позволяет избавиться от некоторого числа
- сравнений и немного ускорить добавление. Модифицированный алгоритм добавления будет выглядеть следующим образом.
- RadixTree.AddWord(W#)
- begin C ← Root; {Текущий узел}
- I ← 1;
- {Пропускаем префикс слова, который уже есть в дереве}
- while (I ≤ Length(W#)) and (Childs[C][W[I]] ≠ NIL) do begin
- I ← I + 1; C{ледующая буква}
- C ← Childs[C][W[I]]; {Вниз по дереву}
- end;
- while I ≤ Length(W#) do begin
- {Ссылки нет – нужно создать новую вершину}
- J ← NewNode();
- {И заполнить ее для символа W[I]}
- Parent[J] ← C; ParentChar[J] ← W[I]; Childs[C][W[I]] ← J;
- C ← J;
- end;
- Final[C] ← T RUE; {Новая конечная вершина}
- return (C);
- end
- Сложность алгоритма добавления остается той же самой, но работа несколько ускоряется из-за удаления лишнего сравнения – при добавлении множества узлов экономия может выглядеть достаточно значительной.

- Процедура поиска в цифровом дереве очень проста – просто следует идти по ссылкам, помеченным буквами искомого слова от корня.
- Алгоритм очень похож на добавление новой вершины. Единственное различие в том, что если вершина не находится, то алгоритм прекращает свою работу, а не пытается добавить новую вершину.
- RadixTree.Search(W#)
- begin
- C ← Root;
- I ← 1;
- {Идем по дереву, пока не кончилось слово и ссылки в дереве}
- while (I ≤ Length(W#)) and (Childs[C][W[I]] ≠ NIL) do begin
- I ← I + 1;
- C ← Childs[C][W[I]];
- end;
- if (I > Length(W#)) and Final[C]   Н{ашли слово?}
- then return (C)   {Да, нашли}
- else return (-1);   {Не нашли}
- end
- Сложность алгоритма поиска в цифровом дереве составляет O(LW). Именно такой быстрый поиск является существенным преимуществом цифровых деревьев – при поиске в словаре, каждая буква просматривается ровно по одному разу. Алгоритм возвращает индекс узла дерева соответствующего слову, если слово найдено, и
- –1 в противном случае.

- Процедура удаления слова из цифрового дерева основывает на специальном вспомогательном алгоритме – этот алгоритм определяет, является ли узел дерева листом. Этот алгоритм просто просматривает ссылки на потомков. Если он находит хотя бы одну непустую ссылку, то узел листом не является.
- RadixTree.IsLeaf(I)
- begin
- for Ch ∈ A{} do
- if Childs[I][Ch] ≠ NIL
- then return (FALSE);
- return (TRUE);
- end
- Если процедуры удаления из цифрового дерева встречаются часто, то имеет смысл добавить в каждую вершину специальный счетчик потомков – он позволит обойтись без сканирования ссылок в вершине.
- Алгоритм удаления слова из цифрового дерева достаточно прост. Сначала с помощью алгоритма поиска находится узел дерева, соответствующий удаляемому слову. Затем производится подъем по дереву с удалением всех узлов, являющихся листами. Здесь есть один тонкий момент – промежуточные узлы дерева, являющиеся концевыми. Удалять эти узлы нельзя, даже если они являются листьями дерева. Эта тонкость достаточно часто служит источником ошибок при кодировании удаления из цифрового дерева.

- RadixTree.Delete(W#)
- begin
- I ← Search(W#); {Ищем слово W#}
- if (I ≠ -1) and (I ≠ Root) {Слово найдено, и оно не пустое}
- then begin
- Final[I] ← FALSE; {Снимаем признак концевой вершины}
- {Так как Final[Root] = TRUE, то экономим сравнение}
- while (not Final[I]) and IsLeaf(I) do begin
- {Удаляем из дерева текущий узел, если он лист}
- P ← Parent[I];
- Childs[P][ParentChar[I]] ← NIL;
- {Теперь узел с индексом I можно освободить}
- ...
- I ← P ; {Шаг по дереву вверх, к корню}
- end;
- end;
- end
- Заметьте, что эта процедура не может удалить корень, а ее сложность составляется O(LW*|A{}|). Множитель |A{}| появился из-за использования алгоритма определения листа. Если в каждый узел добавить счетчик потомков, то можно улучшить алгоритм, превратив оценку в O(LW).

- Последний алгоритм, который нам нужно описать – это получение слова по заданной вершине. Алгоритм заключается в проходе начиная с данного узла до корня и левой конкатенации (присоединении) меток проходимых дуг.
- RadixTree.Word(Idx)
- begin
- W# ← «»;
- while Idx ≠ Root do begin
- W# ← ParentChar[Idx] + W#;
- Idx ← Parent[Idx];
- end;
- return (W#)
- end
- Сложность этого алгоритма будет O(LW), если реализация строк позволяет эффективную левую конкатенацию. Если реализация позволяет только правую конкатенацию, то нужно использовать ее, а в конце просто перевернуть результат – так как это можно сделать за все тоже O(LW), общая оценка не ухудшится.

Реализация словарей посредством ХЕШ-таблиц

```
const  B = { подходящая константа };
type celltype = record
                  element: nametype;
                  next: Icelltype
end;
DICTIONARY = array[0..B-1] of Tcelltype;
procedure MAKENULL ( var A: DICTIONARY );
var i: integer;
begin
  for i:= 0 to B-1 do
    A [ i] := nil
end; {MAKENULL}

function MEMBER ( x: nametype; var A: DICTIONARY): boolean;
var current: tcelltype;
begin
   current: = A[h(x)]; {}
   while current <> nil do
     if currenti.element = x then begin MEMBER :=true; exit end
     else current: = current^.next;
  MEMBER :=false
end; {MEMBER}
```

```
procedure INSERT (var x: name type ; var A: DICTIONARY );
var     bucket: integer;       oldheader: tcelltype;
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 }
procedure DELETE (var x: nametype; var A: DICTIONARY);
Var bucket: integer; current:: tcelltype;
begin
   bucket:= h(x);
   if A[bucket] <> nil then begin
        if A [bucket]^.element = x then A[bucket]:= A[bucket]^.next
       else begin
                 current:= A[bucket];
                 while current^.next <> nil do
                     if current^. nexti . element = x then begin
                                             current^.next:= current^.next^.next;
                                             exit
                                        end
                 else current:= current^.next
            end {else}
     end {then}
end; { DELETE }
```

```
procedure ASSIGN ( var A: MAPPING; d: domaintype; r: rangetype );
var  bucket: integer;  current: tcelltype;
begin
   bucket:= h(d);  current:= A[bucket];
   while current <> nil do
      if current^. domainelement = d
      then begin current^.range:=r{замена старого значения d}
                        exit
                end
     else current:= current^.next;
{ d не найден в списке}
  current:= A[bucket];
  new(A[bucket]) ; A[bucket] ^. domainelement:=d;
  A[bucket]^.range:= r;
  A[bucket]^.next:= current;
end; { ASSIGN }
```

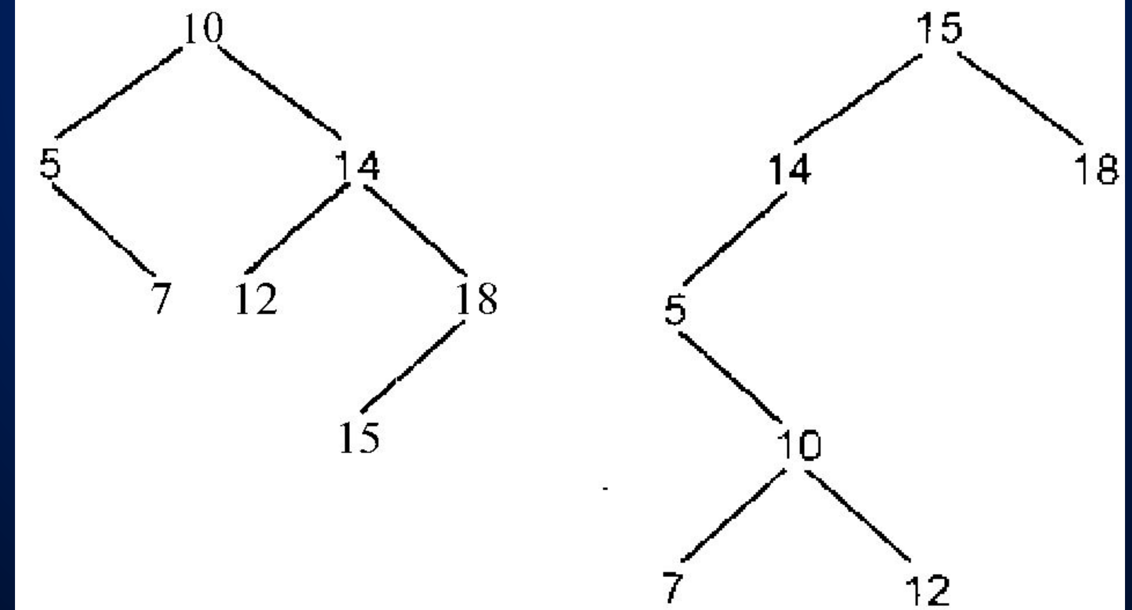
Реализация словарей посредством закрытого хеширования

```
const empty = '                    ';{10 пробелов}
        deleted = '**********' ;  {10 *}
type DICTIONARY = array[0. .B-l] of nametype;
procerude MAKENULL ( var A: DICTIONARY);
var i: integer;
begin
for i:= 0 to B-1 do
  A[i] : = empty
End; {MAKENULL }

function locate ( x: nametype; A: DICTIONARY ): integer;
var initiall : integer;
begin
  initiall:= 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 }
```

```
function locate1( x: name type ; A: DICTIONARY ): inte:rer;
{то же самое, что и locate, но останавливается при достижении deleted}
function MEMBER ( x: nametype; var A: DICTIONARY ): boolean;
begin
   if A[locate(x)] = x then MEMBER := true     else MEMBER :=false
end; {MEMBER}
procedure INSERT (var x: nametype; var A: DICTIONARY);
var       backet: integer;
begin
 if A[locate(x)] x then exit; {x и так есть в A }
  bucket:= locate1(x);
  if (A[bucket] = empty) or (A (bucket] = deleted) then A[bucket]:= x
  else error(‘таблица переполнена'}
end; { INSERT }
procedure DELETE (var x: nametype; var A: DICTIONARY );
Var bucket: integer;
begin
   bucket:= locate(x);
   if A[locate(x)] = x then A[bucket]:= deleted
end; { DELETE }
```

Дерево поиска

- Дерево поиска устроено так: элементы, меньшие корня, хранятся в левом поддереве, а большие корня – в правом.



```
function MEMBER ( x: elementtype; A: SET ): boolean;
{ возвращает true, если элемент x принадлежит множеству A,
    false – в противном случае}
begin
  if A = nil then MEMBER:= false) { x не принадлежит A}
  else if x = A^. element then MEMBER :=true
         else if x < A^. element
                then MEMBER :=MEMBER (x, A^.leftchild)
               else {x > AI. element }
                       MEMBER :=MEMBER (x, A^.rightchild)
end; { MEMBER }
procedure INSERT ( x: element type; var A: SET);
begin
  if A = nil
  then begin
        new(A) ; A^. element: = x; A^.leftchild:= nil; A^. rightchild: = nil
       end
  else if x < A^. element then INSERT (x, A^.leftchild)
         else if x > A^.element then INSERT (x, A^.rightchild)
end; { INSERT }
```

```
function DELETEMIN ( var A: SET ): elementtype;
begin
   if A^.leftchild = nil
   then begin {A указывает на наименьший элемент}
          DELETEMIN:= A^.element;
          A:= A^.rightchild
      end
  else {узел имеет левого сына}
        DELETEMIN:= DELETEMIN (A^. leftchild)
end; { DELETEMIN }

procedure DELETE ( x: elementtype; var A: SET );
begin
   if A< > nil
   then if x < A^. element
          then DELETE (x, A^.leftchild)
          else if x > A^. element
                  then DELETE (x, A^.rightchild)
                  else if (A^ .leftchild= nil) and (A^. rightchild= nil)
                         then A: = nil { удалени листа, содержащего x }
                         else if A^.leftchild = nil
                                 then A:= A^.rightchild
                                 else if A^.rightchild = nil
                                         then A: = A^. leftchild
                                         else { у узла есть оба сына}
                                               A^.element:= DELETEMIN(A^.rightchild)
end; DELETE}
```

Красно-черные деревья

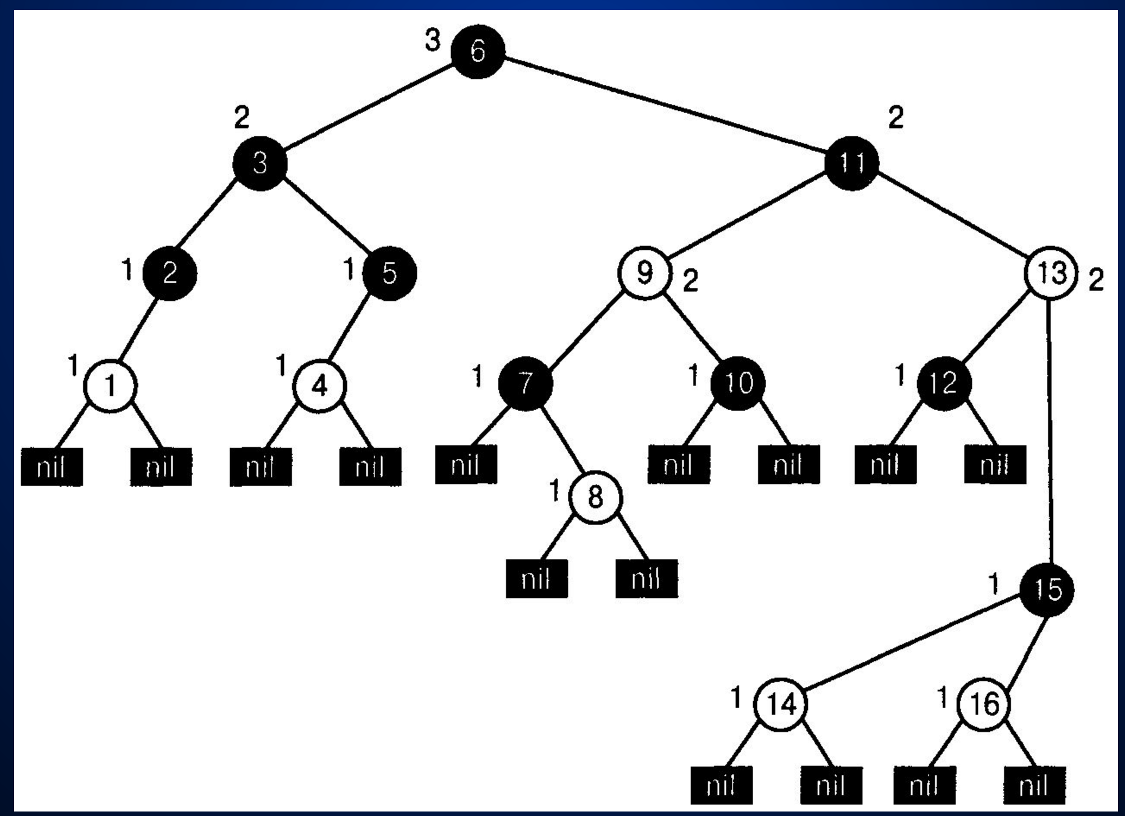
Как упоминалось в предыдущем разделе, обычное бинарное дерево поиска в наихудшем случае представляет собой одномерную цепочку узлов, а высота Такого дерева становится равной n. Такое дерево образуется, например, при внесении в него возрастающей последовательности значений. Однако путем достаточно небольших модификаций можно гарантировать, что даже в наихудшем случае высота бинарного дерева поиска будет равна O(log n). Заметим, что операции поиска минимального, максимального, последующего и предшествующего элементов, а также поиска элемента с заданным значением (соответственно, и реализация итераторов) зависят только от свойства бинарного дерева поиска и, таким образом, остаются неизменными для любых бинарных деревьев поиска - будь то рассмотренные выше простейшие бинарные деревья поиска или рассматриваемые далее красно-черные деревья. Однако того же нельзя сказать об операциях вставки и удаления, поскольку именно при их выполнении нарушается свойство сбалансированности дерева, которое и подлежит восстановлению.

Красно-черные деревья гарантируют, что ни один путь в дереве от корня к вершине не отличается от другого по длине более чем в два раза, так что красно-черное дерево является приближенно сбалансированным и имеет высоту не более, чем $2\log_2(n + 1)$. Каждый узел красно-черного дерева содержит, помимо указателей, дополнительное поле цвета, который может быть либо красным, либо черным (откуда и происходит название данного дерева). Если дочерний или родительский по отношению к данному узел не существует, соответствующий указатель принимает специальное значение nil. Эти значения ni! можно рассматривать как указатели на внешние узлы (листья) бинарного дерева поиска. При этом все «нормальные»- узлы, содержащие поле значения, становятся внутренними узлами дерева.

Бинарное дерево поиска является красно-черным деревом, если оно удовлетворяет следующим красно-черным свойствам.

1. Каждый узел является красным или черным.
2. Корень дерева является черным.
3. Каждый лист дерева (nil) является черным.
4. Если узел - красный, то оба его дочерних узла - черные.
5. Для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов.

На рис. 1,9, а представлен пример красно-черного дерева. Рядом с узлами стоят цифры, указывающие количество черных узлов на пути от листьев к данному узлу.



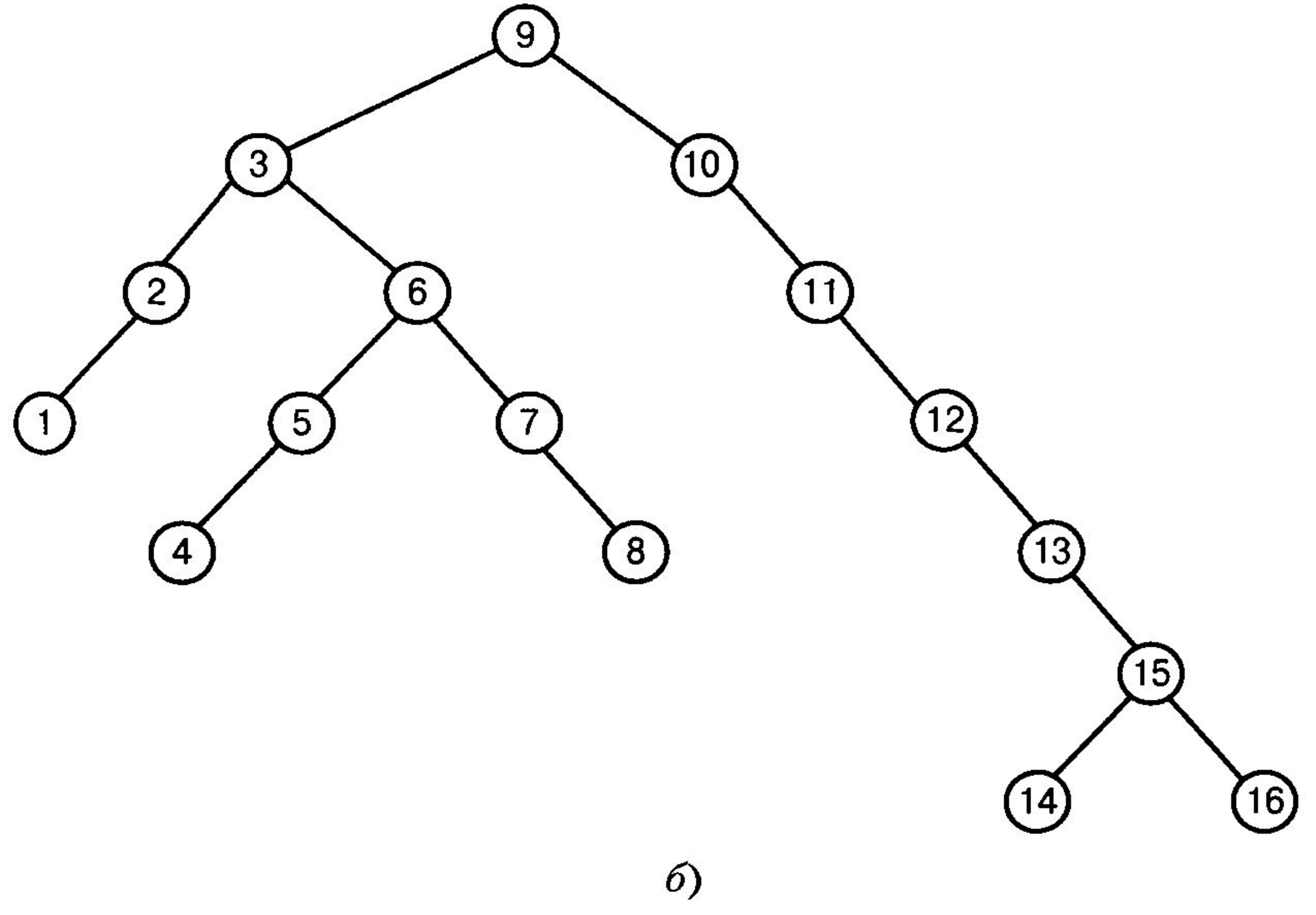


Рис. 1.9. *а) красно-черное дерево и б) бинарное дерево поиска для входных данных 9, 10, 3, 2, 6, 7, 5, 8, 11, 1, 4, 12, 13, 15, 14, 16*

Можно доказать, что высота красно-черного дерева с n узлами не превышает 2 $\log_2 (n+1)$, так что все операции поиска минимального, максимального, последующего и предшествующего элементов, а также поиска элемента с заданным значением обладают эффективностью O(log n) в наихудшем случае. Для удобства работы с красно-черным деревом все листья заменяются единым ограничивающим узлом, представляющим значение nil. Этот узел - черный (значения прочих полей не имеют значения). Кроме того, поскольку, как уже говорилось, в случае отсутствия родительского узла соответствующий указатель также принимает значение nil, этот узел-ограничитель выполняет функции родительского узла по отношению к корню красно-черного дерева. Процедуры вставки узла в красно черное дерево и удаления из него требуют определенных модификаций. Дело в том, что, если применять рассмотренные ранее процедуры TreeInsert и TreeRemove, корректно вставляя и удаляя узлы, будут нарушены свойства красно-черного дерева. Для исправления ситуации используется процедура, именуемая поворотом, которая представляет собой локальную операцию в бинарном дереве поиска, сохраняющую его свойство. На рис. 1.10 показаны два типа поворотов - левый и правый (здесь a, b и c - произвольные поддеревья).

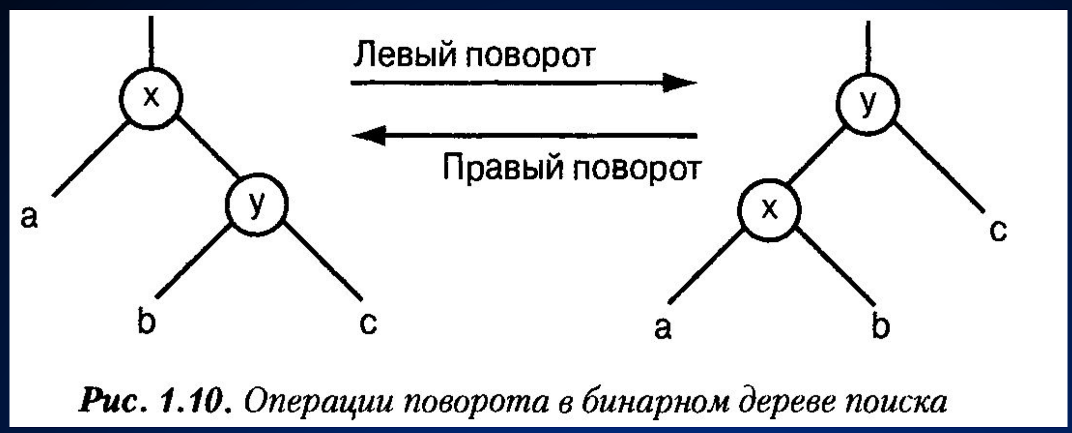


Рис. 1.10. Операции поворота в бинарном дереве поиска

- При выполнении левого поворота в узле x предполагается, что его правый дочерний узел y не является листом nil. Левый поворот выполняется вокруг связи между x и y, делая y новым корнем поддерева, левым дочерним узлом которого становится x, а бывший левый потомок узла y - правым потомком x.
- Ниже приведен псевдокод левого поворота (правый поворот полностью симметричен левому). В псевдокоде процедуры LeftRotate предполагается, что right[x] <> nil, и что nil также является родителем корневого узла.

LeftRotate(root,x)
// Входные данные:
{узел x, вокруг которого выполняется левый поворот в дереве с корневым узлом root }
Выходные данные: {дерево с выполненным поворотом} //}

```
y = right [x] {Присваивание y }
right[x] = left[y] {Левое поддерево y становится правым поддеревом x }
 parent[left[y]] := x;
 parent[y] = parent[x] // Перенос родителя x в y
 if parent[x] = nil then root = y
 else begin
         if x = left[parent[x]] then left [parent [x] ]:= y
         else right [parent [x] ]:= y
       end;
 left [y] := x {x - левый дочерний y }
 parent [x]:= y
```

Очевидно, что данная процедура (как и процедура правого поворота) выполняется за время 0(1),

Вставка узла в красно-черное дерево с n узлами выполняется, как и в обычное бинарное дерево поиска, за время O(log n). Для вставки узла в красно-черное дерево используется модифицированная версия процедуры TreeInsert, которая вставляет узел в дерево, как если бы это было обычное бинарное дерево поиска (только вместо ну левых указателей теперь используется значение nil), а затем окрашивает его в красный цвет. Для того чтобы вставка сохраняла красно-черные свойства дерева, после нее вызывается вспомогательная процедура RBInsertFixup, которая перекрашивает узлы и выполняет повороты.

RBInsert(root,z)

Входные данные:{узел z, добавляемый в дерево с корневым узлом root }

Выходные данные: {красно-черное дерево с добавленным в него узлом z }

```
 y := nil; x := root ;
while x <>nil do begin
    y:= X;
    if value[z] < value[x]
    then x :=left [x] else x:=right [x];
End;
parent [z] := y;
if y = nil then root := z
else begin
        if value[z] < value[y] then left[y] := z else right [y] := Z;
    end
left [z]:=nil;  right [z] :=nil; color[z]:= RED;
RBInsertFixup(root, z);
```

RBInsertFixup(root,z)

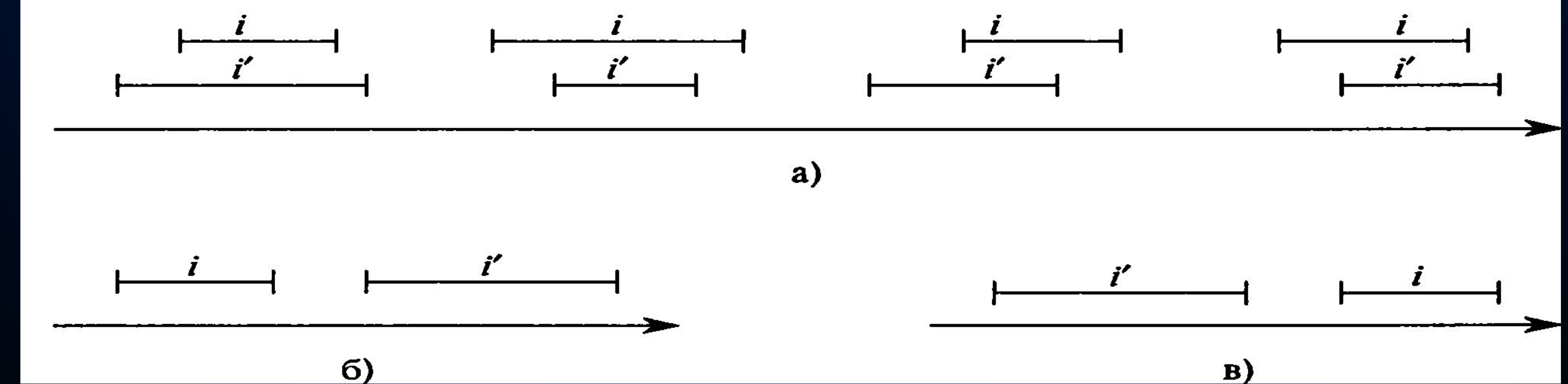
// Входные данные: узел z, добавленный в дерево с корневым узлом root

Выходные данные: красно-черное дерево с восстановленными после добавления узла z красно-черными свойствами

```
while color[parent[z]] = RED do begin
    if parent[z] = left[parent[parent[z]]]
    then begin
            y := right [parent [parent [z] ] ];
            if color[y] = RED
            then begin color[parent[z]] := BLACK;
                    color[У] := BLACK
                    color[parent[parent[z]]]:=RED;
                    z := parent [parent [z] ]
                end
            else begin
                    if z = right [parent [z] ]
                    then begin z := parent [z];
                            LeftRotate(root,z)
                         end;
                    color[parent[z]] := BLACK; color[parent[parent[z]]] := RED;
                    RightRotate(root, parent [parent [z]])
                end
  else begin Здесь код такой же, как и в части ‘then’, но все “left" в нем заменяются на
      "right" (включая направления поворотов) и наоборот
        end
 end;
color[root] = BLACK
```

Деревья отрезков

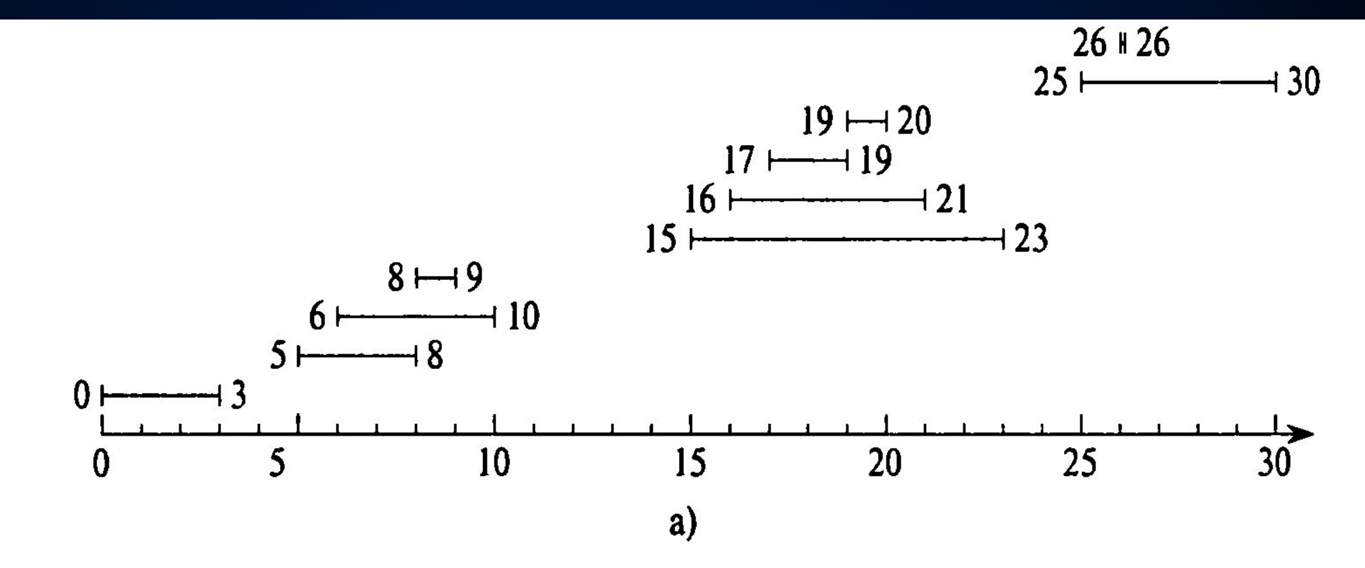
- Отрезком называется упорядоченная пара действительных чисел [t1,t2]> таких что t1<=t2/ Отрезок [t1,t2] представляет множество {t∈R : t1<=t<=t2}. Интервал (t1,t2) представляет собой отрезок без конечных точек, т.е. множество {t∈R : t1<=t<=t2}, а полуинтервалы [t1,t2) и (t1,t2] образуются из отрезка при удалении из него одной из конечных точек. В случае, когда принадлежность концов несущественна, обычно говорят о промежутках. Отрезки удобны для представления событий, которые занимают некоторый промежуток времени. Мы можем представить отрезок [t1, t2] в виде объекта i с полями low [i] = t1 (левый, или нижний, конец отрезка) и high [i] = t2 (правый, или верхний, конец). Мы говорим, что отрезки i и i' перекрываются (overlap), если i ∩ i' <> 0, т.е. если low [i] < high [i`] и low [i`] < high [i]. Для любых двух отрезков i и i` выполняется только одно из трех свойств (трихотомия отрезков):
- а) i и i` перекрываются;
- б) i находится слева от i1 (т.е. high [i] < low [i`]);
- в) i находится справа от i` (т.е. high [i`] < low [i]).



Дерево отрезков представляет собой красно-черное дерево, каждый элемент которого содержит отрезок int[x]. Деревья отрезков поддерживают следующие операции.

- Interval_Insert(T, x), которая добавляет в дерево отрезков T элемент x, поле int которого содержит некоторый отрезок.
- Interval_Delete(T, x), которая удаляет элемент x из дерева отрезков T.
- Interval_Search(T, i), которая возвращает указатель на элемент x из дерева отрезков T, такой что int [x] перекрывается с отрезком г (либо ограничитель nil [T], если такого элемента в дереве нет).

- Шаг 1. Выбор базовой структуры данных. В качестве базовой структуры данных мы выбираем красно-черное дерево, каждый узел x которого содержит отрезок int [x], а ключом узла является левый конец отрезка low [int[x]]. Таким образом, центрированный обход дерева приводит к перечислению отрезков в порядке сортировки по их левым концам.
- Шаг 2. Дополнительная информация. В дополнение к самим отрезкам, каждый узел x содержит значение max [x], которое представляет собой максимальное значение всех конечных точек отрезков, хранящихся в поддереве, корнем которого является x.
- Шаг 3. Поддержка информации. Мы должны убедиться, что вставка и удаление в дереве с n узлами могут быть выполнены за время O(log n). Определить значение поля max в узле x можно очень просто с использованием полей max дочерних узлов: max [x] = max (high [int [x]], max [left [x]], max [right [x]]). Таким образом, вставка в дерево отрезков и удаление из него может быть выполнена за время O (log n).



a)

б)

Шаг 4. Разработка новых операций.

Единственная новая операция, которую мы хотим разработать, — это INTERVAL_SEARCH(T, i), которая осуществляет поиск отрезка в дереве T, который перекрывается с данным. Если такого отрезка в дереве нет, процедура возвращает указатель на ограничитель nil [T]:

```
Interval_Search(T, i)
x <- root[T]
while x <> nil[T] и i не перекрывается с int[x] do
   if left[x] <> nil[T] и max[ left [x] ] > low[i]
   then x <- left[x]
   else x <- right[x]
return x
```

Для поиска отрезка, который перекрывается с i, мы начинаем с присвоения указателю x корня дерева и выполняем спуск по дереву. Спуск завершается когда мы находим перекрывающийся отрезок или когда x указывает на ограничитель nil [T]. Поскольку каждая итерация основного цикла выполняется за время O(1), а высота красно-черного дерева с n узлами равна O (log n), время работы процедуры Interval_Search равно O (log n).

Подсчитаем и запомним где-нибудь сумму элементов всего массива, т.е. отрезка a[0..n-1]. Также посчитаем сумму на двух половинках этого массива: a[0..n/2] и a[n/2+1..n-1]. Каждую из этих двух половинок в свою очередь разобьём пополам, посчитаем и сохраним сумму на них, потом снова разобьём пополам, и так далее, пока текущий отрезок не достигнет длины 1. Иными словами, мы стартуем с отрезка a[0..n-1] и каждый раз делим текущий отрезок надвое (если он ещё не стал отрезком единичной длины), вызывая затем эту же процедуру от обеих половинок; для каждого такого отрезка мы храним сумму чисел на нём.

Можно говорить, что эти отрезки, на которых мы считали сумму, образуют дерево: корень этого дерева — отрезок a[0..n-1], а каждая вершина имеет ровно двух сыновей (кроме вершин-листьев, у которых отрезок имеет длину 1). Отсюда и происходит название — "дерево отрезков" (хотя при реализации обычно никакого дерева явно не строится, но об этом ниже в разделе реализации).

Итак, мы описали структуру дерева отрезков. Сразу заметим, что оно имеет линейный размер, а именно, содержит менее 2n вершин. Понять это можно следующим образом: первый уровень дерева отрезков содержит одну вершину (отрезок a[0..n-1]), второй уровень — в худшем случае две вершины, на третьем уровне в худшем случае будет четыре вершины, и так далее, пока число вершин не достигнет n . Таким образом, число вершин в худшем случае оценивается суммой $n+n/2+n/4+n/8+\ldots+1<2n$.

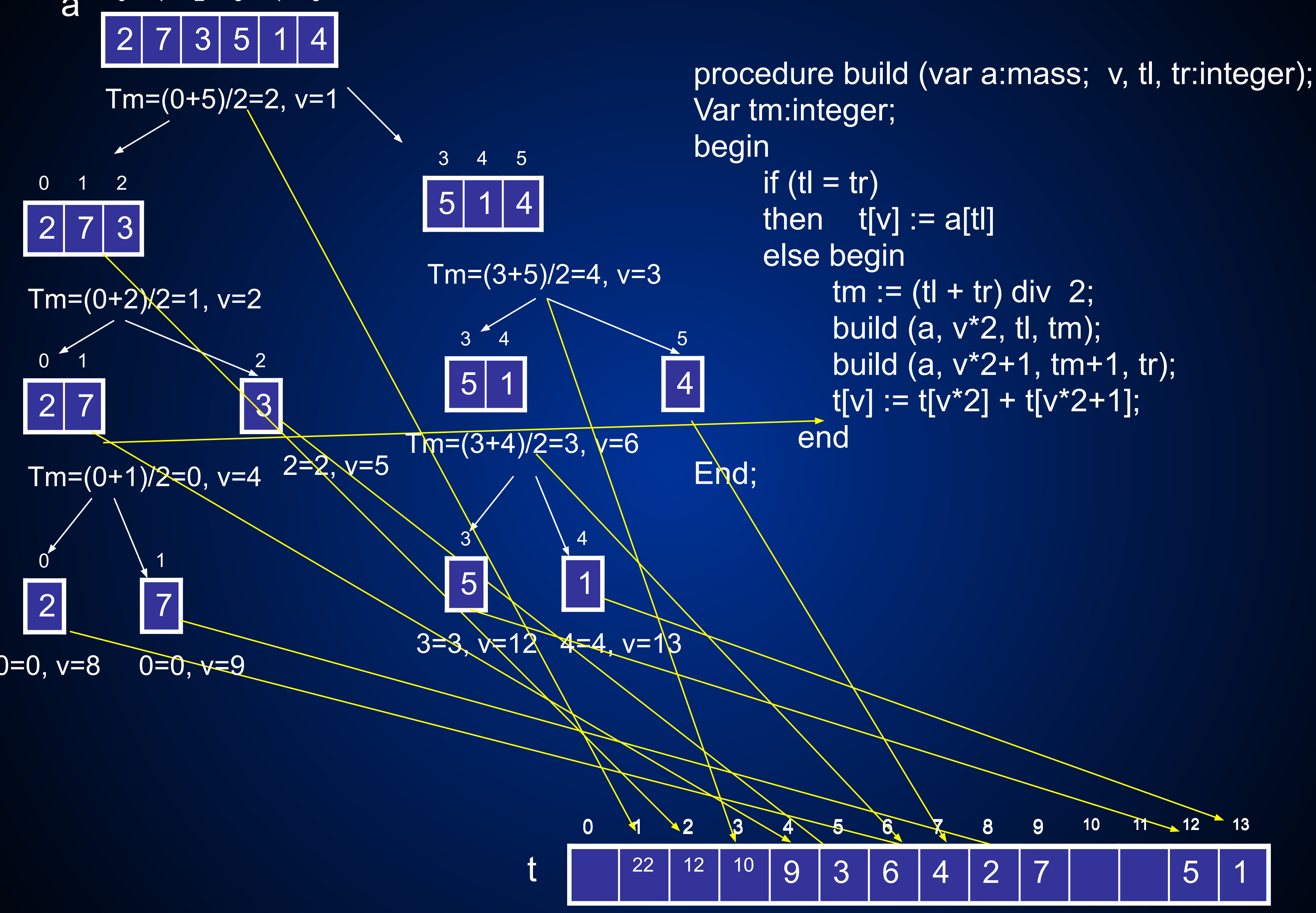
Построение

- Процесс построения дерева отрезков по заданному массиву   можно делать эффективно следующим образом, снизу вверх: сначала запишем значения элементов a[i] в соответствующие листья дерева, затем на основе них посчитаем значения для вершин предыдущего уровня как сумму значений в двух листьях, затем аналогичным образом посчитаем значения для ещё одного уровня, и т.д. Удобно описывать эту операцию рекурсивно: мы запускаем процедуру построения от корня дерева отрезков, а сама процедура построения, если её вызвали не от листа, вызывает себя от каждого из двух сыновей и суммирует вычисленные значения, а если её вызвали от листа — то просто записывает в себя значение этого элемента массива. Асимптотика построения дерева отрезков составит, таким образом, O(n) .
- Основной реализационный момент — это то, как хранить дерево отрезков в памяти. В целях простоты мы не будем хранить дерево в явном виде, а воспользуемся таким трюком: скажем, что корень дерева имеет номер 1, его сыновья — номера 2 и 3, их сыновья — номера с 4 по 7, и так далее. Легко понять корректность следующей формулы: если вершина имеет номер i, то пусть её левый сын — это вершина с номером 2i, а правый — с номером 2i+1.Такой приём значительно упрощает программирование дерева отрезков, — теперь нам не нужно хранить в памяти структуру дерева отрезков, а только лишь завести какой-либо массив для сумм на каждом отрезке дерева отрезков. Стоит только отметить, что размер этого массива при такой нумерации надо ставить не 2n, а 4n. Дело в том, что такая нумерация не идеально работает в случае, когда n не является степенью двойки — тогда появляются пропущенные номера, которым не соответствуют никакие вершины дерева (фактически, нумерация ведёт себя подобно тому, как если бы n округлили бы вверх до ближайшей степени двойки). Это не создаёт никаких сложностей при реализации, однако приводит к тому, что размер массива надо увеличивать до 4n. Итак, дерево отрезков мы храним просто в виде массива , размера вчетверо больше размера   входных данных:

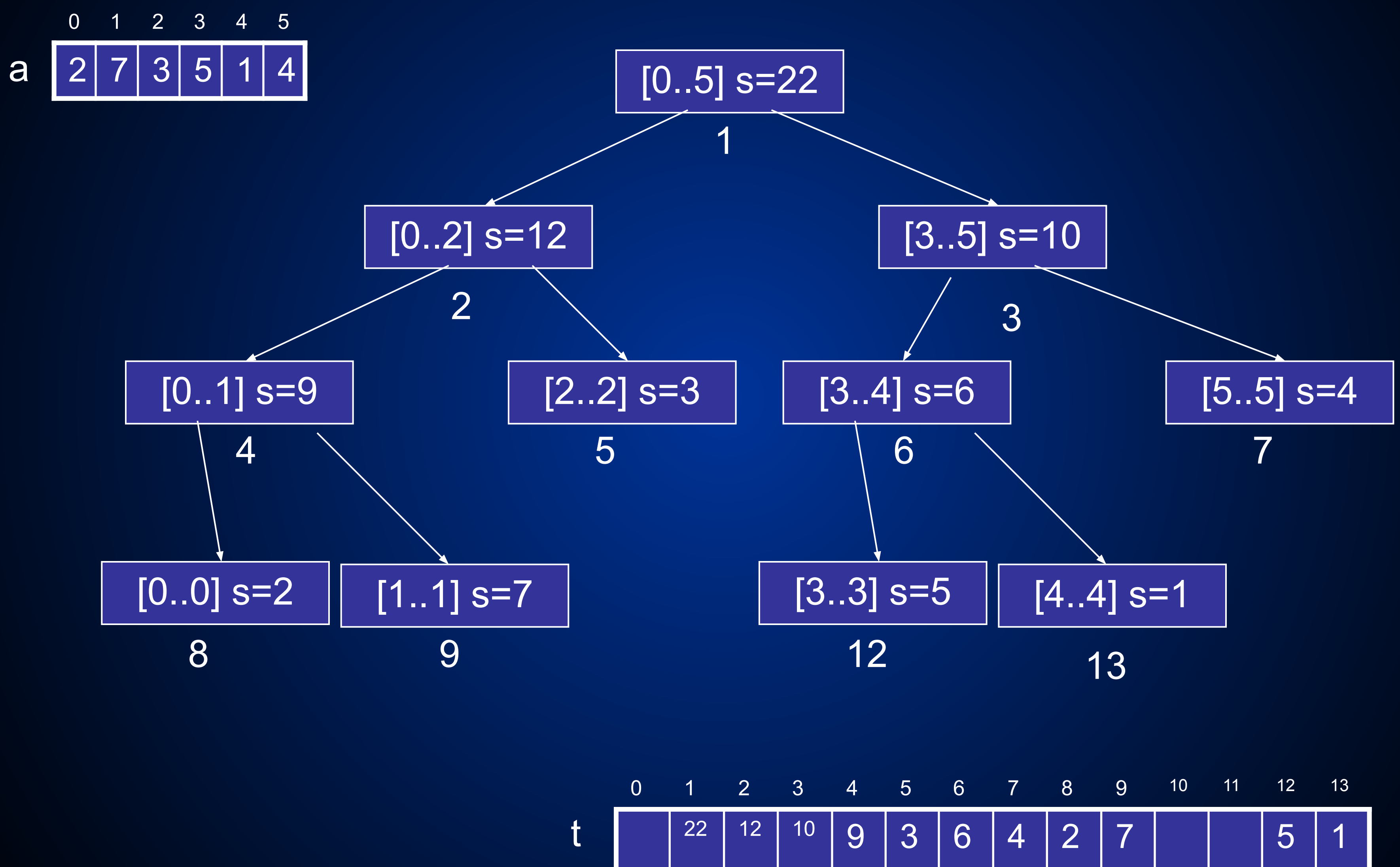
Процедура построения дерева

- выглядит следующим образом:
- это рекурсивная функция, ей передаётся сам массив a[], номер v текущей вершины дерева, и границы tl и tr отрезка, соответствующего текущей вершине дерева. Из основной программы вызывать эту функцию следует с параметрами v=1, tl=0, tr=n-1.

```
procedure build (var a:mass;  v, tl, tr:integer);
Var tm:integer;
begin
      if (tl = tr)
      then    t[v] := a[tl]
      else begin
            tm := (tl + tr) div  2;
            build (a, v*2, tl, tm);
            build (a, v*2+1, tm+1, tr);
            t[v] := t[v*2] + t[v*2+1];
         end
End;
```


a
0 1 2 3 4 5
2 7 3 5 1 4
Tm=(0+5)/2=2, v=1
0 1 2
2 7 3
3 4 5
5 1 4
Tm=(0+2)/2=1, v=2
Tm=(3+5)/2=4, v=3
0 1
2 7
2
3
3 4
5 1
5
4
Tm=(0+1)/2=0, v=4
2=2, v=5
Tm=(3+4)/2=3, v=6
0
2
1
7
3
5
4
1
0=0, v=8
0=0, v=9
3=3, v=12
4=4, v=13
procedure build (var a:mass; v, tl, tr:integer);
Var tm:integer;
begin
if (tl = tr)
then t[v] := a[tl]
else begin
tm := (tl + tr) div 2;
build (a, v*2, tl, tm);
build (a, v*2+1, tm+1, tr);
t[v] := t[v*2] + t[v*2+1];
end
End;
t
0 1 2 3 4 5 6 7 8 9 10 11 12 13
22 12 10 9 3 6 4 2 7 5 1

Дерево отрезков



Запрос суммы

- Рассмотрим теперь запрос суммы. На вход поступают два числа l и r, и мы должны за время O(log n) посчитать сумму чисел на отрезке a[l,r]. Для этого мы будем спускаться по построенному дереву отрезков, используя для подсчёта ответа посчитанные ранее суммы на каждой вершине дерева. Изначально мы встаём в корень дерева отрезков. Посмотрим, в какие из двух его сыновей попадает отрезок запроса [l..r] (напомним, что сыновья корня дерева отрезков — это отрезки [0..n/2] и [n/2+1..n-1]). Возможны два варианта: что отрезок [l..r] попадает только в одного сына корня, и что, наоборот, отрезок пересекается с обоими сыновьями.
- Первый случай прост: просто перейдём в того сына, в котором лежит наш отрезок-запрос, и применим описываемый здесь алгоритм к текущей вершине.
- Во втором же случае нам не остаётся других вариантов, кроме как перейти сначала в левого сына и посчитать ответ на запрос в нём, а затем — перейти в правого сына, посчитать в нём ответ и прибавить к нашему ответу. Иными словами, если левый сын представлял отрезок [l1..r1], а правый — отрезок [l2..r2] (заметим, что l2=r1+1), то мы перейдём в левого сына с запросом [l,r1], а в правого — с запросом [l2,r].
- Итак, обработка запроса суммы представляет собой рекурсивную функцию, которая всякий раз вызывает себя либо от левого сына, либо от правого (не изменяя границы запроса в обоих случаях), либо от обоих сразу (при этом деля наш запрос на два соответствующих подзапроса). Однако рекурсивные вызовы будем делать не всегда: если текущий запрос совпал с границами отрезка в текущей вершине дерева отрезков, то в качестве ответа будем возвращать предвычисленное значение суммы на этом отрезке, записанное в дереве отрезков.

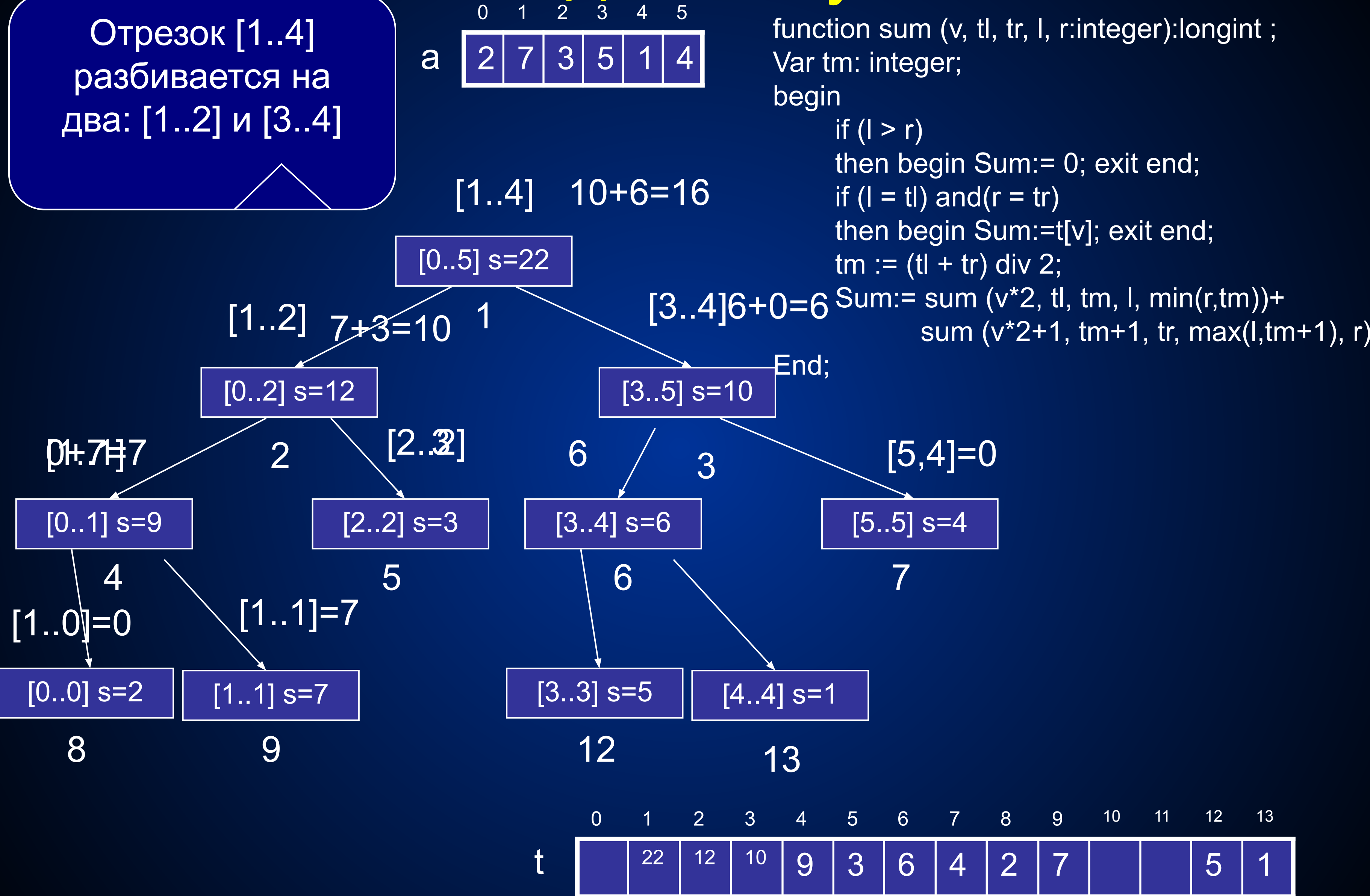
Функция для запроса суммы

представляет из себя также рекурсивную функцию, которой таким же образом передаётся информация о текущей вершине дерева (т.е. числа v, tl, tr, которым в основной программе следует передавать значения 1, 0, n-1 соответственно), а помимо этого — также границы l и r текущего запроса.

В целях упрощения кода эта функция всегда делает по два рекурсивных вызова, даже если на самом деле нужен один — просто лишнему рекурсивному вызову передастся запрос, у которого , что легко отсекается дополнительной проверкой в самом начале функции.

```
function sum (v, tl, tr, l, r:integer):longint ;
Var tm: integer;
begin
    if (l > r)
    then begin Sum:= 0; exit end;
  if (l = tl) and(r = tr)
  then begin Sum:=t[v]; exit end;
    tm := (tl + tr) div 2;
    Sum:= sum (v*2, tl, tm, l, min(r,tm))+ sum (v*2+1, tm+1, tr, max(l,tm+1), r);
End;
```

Нахождение суммы



Запрос обновления

Напомним, что запрос обновления получает на вход индекс i и значение x, и перестраивает дерево отрезков таким образом, чтобы оно соответствовало новому значению a[i]=x. Этот запрос должен также выполняться за время O(log n). Это более простой запрос по сравнению с запросом подсчёта суммы. Дело в том, что элемент a[i] участвует только в относительно небольшом числе вершин дерева отрезков: а именно, в O(log n) вершинах — по одной с каждого уровня. Тогда понятно, что запрос обновления можно реализовать как рекурсивную функцию: ей передаётся текущая вершина дерева отрезков, и эта функция выполняет рекурсивный вызов от одного из двух своих сыновей (от того, который содержит позицию в своём отрезке), а после этого — пересчитывает значение суммы в текущей вершине точно таким же образом, как мы это делали при построении дерева отрезков (т.е. как сумма значений по обоим сыновьям текущей вершины).

```
procedure update (v, tl, tr, pos, new_val: integer);
Var tm: integer;
begin
     if (tl = tr)
     then  t[v] := new_val;
     else begin
          tm := (tl + tr) / 2;
          if (pos <= tm)
          then  update (v*2, tl, tm, pos, new_val);
          else  update (v*2+1, tm+1, tr, pos, new_val);
          t[v] := t[v*2] + t[v*2+1];
       end
end
```

Обновление на отрезке

- Выше рассматривались только задачи, когда запрос модификации затрагивает единственный элемент массива. На самом деле, дерево отрезков позволяет делать запросы, которые применяются к целым отрезкам подряд идущих элементов, причём выполнять эти запросы за то же время $O(\log_2 N)$.
  - Прибавление на отрезке
- Начнём рассмотрение деревьев отрезков такого рода с самого простого случая: запрос модификации представляет собой прибавление ко всем числам на некотором подотрезке a[l..r] некоторого числа x. Запрос чтения — по-прежнему считывание значения некоторого числа a[i]. Чтобы делать запрос прибавления эффективно, будем хранить в каждой вершине дерева отрезков, сколько надо прибавить ко всем числам этого отрезка целиком. Например, если приходит запрос "прибавить ко всему массиву a[0..n-1] число 2, то мы поставим в корне дерева число 2. Тем самым мы сможем обрабатывать запрос прибавления на любом подотрезке эффективно, вместо того чтобы изменять все O(n) значений. Если теперь приходит запрос чтения значения того или иного числа, то нам достаточно спуститься по дереву, просуммировав все встреченные по пути значения, записанные в вершинах дерева.

```
procedure build (a: array[0..n-1] of integer; v,  tl, tr: integer);
Var tm: integer;
begin
        if (tl = tr) then   t[v] = a[tl];
        else begin
                tm := (tl + tr) div 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
             end
End;
  proceduer  update (v,  tl,  tr,  l,  r,  add: integer);
   Var tm: integer;
   begin
    if (l > r) then exit;
    if (l = tl) and (tr = r) then t[v] := t[v] + add;
    else begin
                tm := (tl + tr) div 2;
                update (v*2, tl, tm, l, min(r, tm), add);
                update (v*2+1, tm+1, tr, max(l, tm+1), r, add);
           end
End;
 function get (v, tl, tr, pos: integer):integer;
Var tm: integer;
begin
        if (tl = tr) then get:=t[v];
        else begin   tm := (tl + tr) div 2;
                        if (pos <= tm) then get:=t[v] + get (v*2, tl, tm, pos);
                      else  get:=t[v] + get (v*2+1, tm+1, tr, pos);
              end
end
```

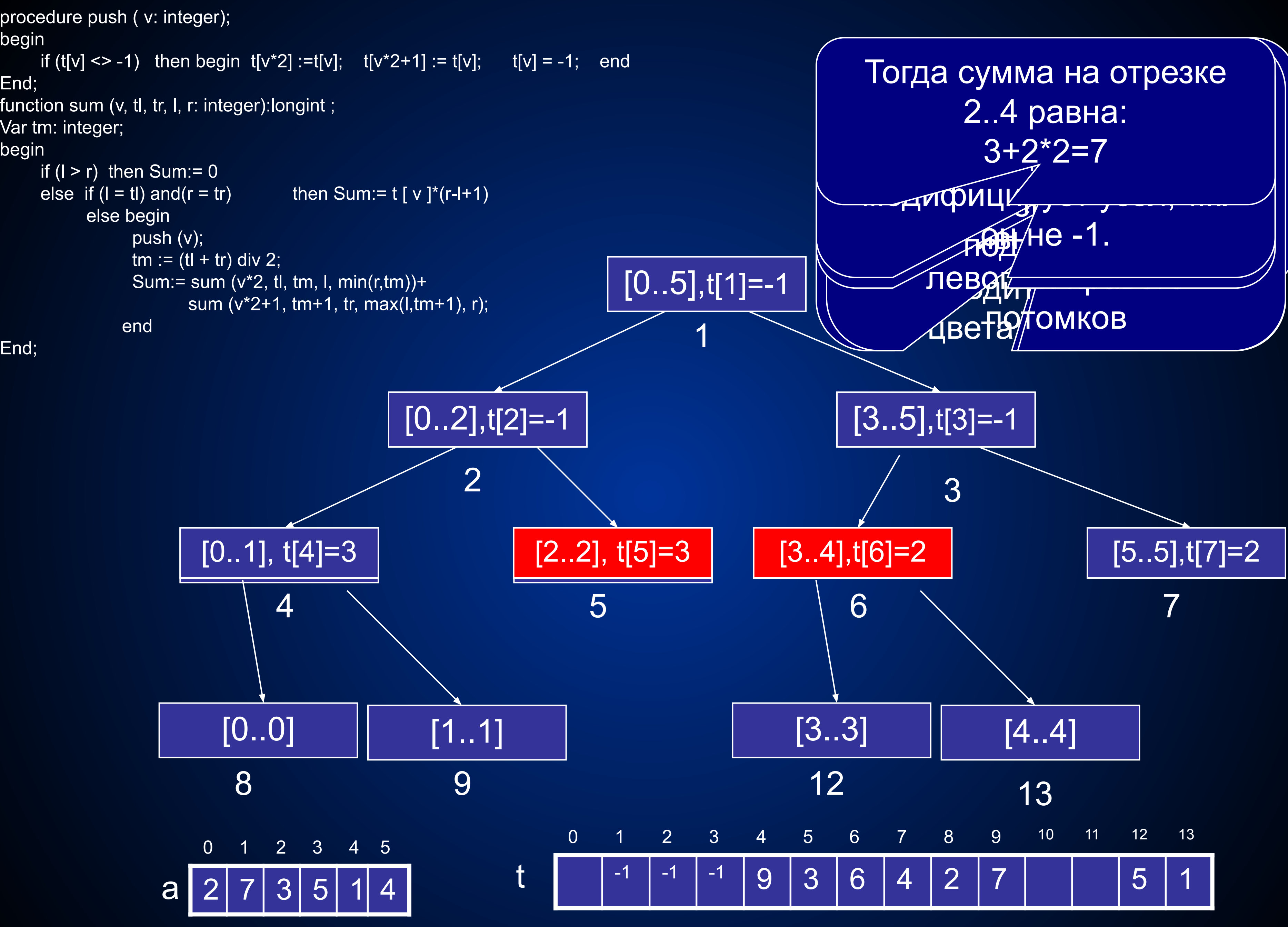
Присвоение на отрезке

- Пусть теперь запрос модификации представляет собой присвоение всем элементам некоторого отрезка a[l..r] некоторого значения p. В качестве второго запроса будем рассматривать считывание значения массива a[i]. Чтобы делать модификацию на целом отрезке, придётся в каждой вершине дерева отрезков хранить, покрашен ли этот отрезок целиком в какое-либо число или нет (и если покрашен, то хранить само это число). Это позволит нам делать "запаздывающее" обновление дерева отрезков: при запросе модификации мы, вместо того чтобы менять значения во множестве вершин дерева отрезков, поменяем только некоторые из них, оставив флаги "покрашен" для других отрезков, что означает, что весь этот отрезок вместе со своими подотрезками должен быть покрашен в этот цвет.
- Итак, после выполнения запроса модификации дерево отрезков становится, вообще говоря, неактуальным — в нём остались недовыполненными некоторые модификации.
- Например, если пришёл запрос модификации "присвоить всему массиву a[0..n-1] какое-то число", то в дереве отрезков мы сделаем единственное изменение — пометим корень дерева, что он покрашен целиком в это число. Остальные же вершины дерева останутся неизменёнными, хотя на самом деле всё дерево должно быть покрашено в одно и то же число.

- Предположим теперь, что в том же дереве отрезков пришёл второй запрос модификации — покрасить первую половину массива a[0..n/2] в какое-либо другое число. Чтобы обработать такой запрос, мы должны покрасить целиком левого сына корня в этот новый цвет, однако перед тем как сделать это, мы должны разобраться с корнем дерева. Тонкость здесь в том, что в дереве должно сохраниться, что правая половина покрашена в старый цвет, а в данный момент в дереве никакой информации для правой половины не сохранено.
- Выход таков: произвести проталкивание информации из корня, т.е. если корень дерева был покрашен в какое-либо число, то покрасить в это число его правого и левого сына, а из корня эту отметку убрать. После этого мы можем спокойно красить левого сына корня, не теряя никакой нужной информации.
- Обобщая, получаем: при любых запросах с таким деревом (запрос модификации или чтения) во время спуска по дереву мы всегда должны делать проталкивание информации из текущей вершины в обоих её сыновей. Можно понимать это так, что при спуске по дереву мы применяем запаздывающие модификации, но ровно настолько,
- насколько это необходимо (чтобы не ухудшить асимптотику с $O(\log_2 n)$).
- При реализации это означает, что нам надо сделать функцию push , которой будет передаваться вершина дерева отрезков, и она будет производить проталкивание информации из этой вершины в обоих её сыновей. Вызывать эту функцию следует в самом начале функций обработки запросов (но не вызывать её из листьев, ведь из листа проталкивать информацию не надо, да и некуда).

- procedure push (v: integer);
- begin
- if (t[v] <> -1)
- then begin t[v*2] :=t[v]; t[v*2+1] := t[v]; t[v] = -1; end
- End;
- procedure update (v, tl, tr, l, r, color: integer);
- var tm: integer;
- begin
- if (l > r) then exit;
- if (l = tl) and (tr = r) then t[v] := color
- else begin
- push (v); tm := (tl + tr) div 2;
- update (v*2, tl, tm, l, min(r,tm), color);
- update (v*2+1, tm+1, tr, max(l,tm+1), r, color);
- end
- End;
- function get (v, tl, tr, pos: integer):integer;
- var tm: integer;
- begin
- if (tl = tr) then get:=t[v]
- else begin
- push (v); tm := (tl + tr) div 2;
- if (pos <= tm) then get:=get (v*2, tl, tm, pos)
- else get:=get (v*2+1, tm+1, tr, pos);
- end;
- End;

```
procedure push ( v: integer);
begin
    if (t[v] <> -1)
     then begin
            t[v*2] :=t[v];     t[v*2+1] := t[v];      t[v] = -1;
          end
End;

function sum (v, tl, tr, l, r: integer):longint ;
Var tm: integer;
begin
    if (l > r)
    then Sum:= 0
    else  if (l = tl) and(r = tr)
        then if Sum:= t [ v ]*(r-l+1) {всего r-l+1 одинаковых чисел, равных t[v]}
        else begin
                push (v);{перед нахождением суммы проталкиваем цвет из текущей}
                tm := (tl + tr) div 2; {вершины в два ее потомка, затем находим сумму}
                Sum:= sum (v*2, tl, tm, l, min(r,tm))+
                     sum (v*2+1, tm+1, tr, max(l,tm+1), r);
              end
End;
```



procedure push (v: integer);
begin
if (t[v] <> -1) then begin t[v*2] :=t[v]; t[v*2+1] := t[v]; t[v] = -1; end
End;
function sum (v, tl, tr, l, r: integer):longint ;
Var tm: integer;
begin
if (l > r) then Sum:= 0
else if (l = tl) and(r = tr) then Sum:= t [v]*(r-l+1)
else begin
push (v);
tm := (tl + tr) div 2;
Sum:= sum (v*2, tl, tm, l, min(r,tm))+
sum (v*2+1, tm+1, tr, max(l,tm+1), r);
end
End;
Тогда сумма на отрезке
2..4 равна:
3+2*2=7
[0..5],t[1]=-1
1
[0..2],t[2]=-1
2
[3..5],t[3]=-1
3
[0..1], t[4]=3
4
[2..2], t[5]=3
5
[3..4],t[6]=2
6
[5..5],t[7]=2
7
[0..0]
8
[1..1]
9
[3..3]
12
[4..4]
13
0 1 2 3 4 5
a 2 7 3 5 1 4
0 1 2 3 4 5 6 7 8 9 10 11 12 13
t -1 -1 -1 9 3 6 4 2 7 5 1

Поиск минимума/максимума

- Немного изменим условие задачи, описанной выше: вместо запроса суммы будем производить теперь запрос минимума/максимума на отрезке. Тогда дерево отрезков для такой задачи практически ничем не отличается от дерева отрезков, описанного выше. Просто надо изменить способ вычисления t[v] в функциях build и update, а также вычисление возвращаемого ответа в функции sum (заменить суммирование на минимум/максимум).
- Поиск минимума/максимума и количества раз, которое он встречается
- Задача аналогична предыдущей, только теперь помимо максимума требуется также возвращать количество его вхождений. Эта задача встаёт естественным образом, например, при решении с помощью дерева отрезков такой задачи: найти количество наидлиннейших возрастающих подпоследовательностей в заданном массиве. Для решения этой задачи в каждой вершине дерева отрезков будем хранить пару чисел: кроме максимума количество его вхождений на соответствующем отрезке. Тогда при построении дерева мы должны просто по двум таким парам, полученным от сыновей текущей вершины, получать пару для текущей вершины.
- Объединение двух таких пар в одну стоит выделить в отдельную функцию, поскольку эту операцию надо будет производить и в запросе модификации, и в запросе поиска максимума.

```
Type pair=record
            first, second:integer
        end;
Var t:array [0..4*MAXN] of pair;
function combine (a,b :pair):pair;
begin
       if (a.first > b.first)
       then combine :=a
       else if (b.first > a.first)
              then combine:=b;
              else combine :=make_pair (a.first, a.second + b.second);
End;
Procedure build (var a:mass; v, tl, tr: integer) ;
Var tm:integer;
begin
       if (tl = tr)
       then  t[v] := make_pair (a[tl], 1);
       else begin tm := (tl + tr) div 2;
              build (a, v*2, tl, tm);
              build (a, v*2+1, tm+1, tr);
              t[v] = combine (t[v*2], t[v*2+1]);
            end
End;
```

```
function get_max (v, tl, tr, l, r: integer):pair;
Var tm: integer;
begin
        if (l > r)
        then get_max :=make_pair (-INF, 0)
        else
        if (l= tl) and( r = tr)
        then get_max :=t[v]
        else begin tm = (tl + tr) div 2;
                get_max :=combine (get_max(v*2, tl, tm, l, min(r,tm)),
                                        get_max(v*2+1, tm+1, tr, max(l,tm+1), r)    );
               end
End;
procedure update (v, tl, tr, pos, new_val: integer);
Var tm: integer;
begin
        if (tl = tr)
        then t[v] = make_pair (new_val, 1);
        else begin
                tm = (tl + tr) div 2;
                if (pos <= tm)
                then update (v*2, tl, tm, pos, new_val);
                else  update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = combine (t[v*2], t[v*2+1]);
              end
End;
```

Подсчёт количества нулей, поиск K-го нуля

- В этой задаче мы хотим научиться отвечать на запрос количества нулей в заданном отрезке массива, а также на запрос нахождения k-го нулевого элемента. Снова немного изменим данные, хранящиеся в дереве отрезков: будем хранить теперь в массиве количество нулей, встречающихся в соответствующих отрезках массива. Понятно, как поддерживать и использовать эти данные в функциях build, sum, update, — тем самым мы решили задачу о количестве нулей в заданном отрезке массива.
- Теперь научимся решать задачу о поиске позиции k-го вхождения нуля в массиве. Для этого будем спускаться по дереву отрезков, начиная с корня, и переходя каждый раз в левого или правого сына в зависимости от того, в каком из отрезков находится искомый k-ый ноль. В самом деле, чтобы понять, в какого сына нам надо переходить, достаточно посмотреть на значение, записанное в левом сыне: если оно больше либо равно k, то переходить надо в левого сына (потому что в его отрезке есть как минимум k нулей), а иначе — переходить в правого сына.
- При реализации можно отсечь случай, когда k-го нуля не существует, ещё при входе в функцию, вернув в качестве ответа, например, -1.

Подсчёт количества нулей, поиск K-го нуля

```
Function find_kth (v, tl, tr, k:integer):integer;
Var tm:integer;
begin
        if (k > t[v])
        then find_kth:= -1
        else if (tl = tr)
                then find_kth:= tl
            else begin
                        tm = (tl + tr) div 2;
                        if (t[v*2] >= k)
                        then find_kth := find_kth (v*2, tl, tm, k);
                        else find_kth:= find_kth (v*2+1, tm+1, tr, k - t[v*2]);
                      end
End;
```

Поиск подотрезка с максимальной суммой

По-прежнему на вход даётся массив a[0..n-1], и поступают запросы (l,r), которые означают: найти такой подотрезок a[l`,r`], что l<l`, r`<r, и сумма этого отрезка a[l`,r`] максимальна. Запросы модификации отдельных элементов массива допускаются. Элементы массива могут быть отрицательными (и, например, если все числа отрицательны, то оптимальным подотрезком будет пустой — на нём сумма равна нулю).Это весьма нетривиальное обобщение дерева отрезков получается следующим образом. Будем хранить в каждой вершине дерева отрезков четыре величины: сумму на этом отрезке, максимальную сумму среди всех префиксов этого отрезка, максимальную сумму среди всех суффиксов, а также максимальную сумму подотрезка на нём. Иными словами, для каждого отрезка дерева отрезков ответ на нём уже предпосчитан, а также дополнительно ответ посчитан среди всех отрезков, упирающихся в левую границу отрезка, а также среди всех отрезков, упирающихся в правую границу.

Как же построить дерево отрезков с такими данными? Снова подойдём к этому с рекурсивной точки зрения: пусть для текущей вершины все четыре значения в левом сыне и в правом сыне уже подсчитаны, посчитаем их теперь для самой вершины. Заметим, что ответ в самой вершине равен:

- либо ответу в левом сыне, что означает, что лучший подотрезок в текущей вершине целиком помещается в отрезок левого сына,
- либо ответу в правом сыне, что означает, что лучший подотрезок в текущей вершине целиком помещается в отрезок правого сына,
- либо сумме максимального суффикса в левом сыне и максимального префикса в правом сыне, что означает, что лучший подотрезок лежит своим началом в левом сыне, а концом — в правом.

Значит, ответ в текущей вершине равен максимуму из этих трёх величин.

- Приведём реализацию функции combine, которой будут передаваться две структуры l, r , содержащие в себе данные о левом и правом сыновьях, и которая возвращает данные в текущей вершине.

- Type data=record
-             sum, pref, suff, ans:integer
-          end;
- Function combine (l, r: data):data ;
- Var Res:data;
- begin
-          res.sum = l.sum + r.sum;
-          res.pref = max (l.pref, l.sum + r.pref);
-          res.suff = max (r.suff, r.sum + l.suff);
-          res.ans = max (max (l.ans, r.ans), l.suff + r.pref);
-          combine:=res;
- End;

```
Function  make_data (int val) :data
Var  res: data
begin    res.sum = val;   res.pref = res.suff = res.ans = max (0, val);  make_data :=res;
End;
Procedure  build (a:mass, v, tl, tr: integer);
Var tm:integer;
begin
        if (tl = tr) then  t[v] = make_data (a[tl])
        else begin
                tm = (tl + tr) div 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                t[v] = combine (t[v*2], t[v*2+1]);
          end
End;
Procedure update (v, tl, tr, pos, new_val: integer) ;
Var tm:integer;
begin
        if (tl = tr) then     t[v] = make_data (new_val);
        else begin
                tm = (tl + tr) div 2;
                if (pos <= tm) then update (v*2, tl, tm, pos, new_val);
                else update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = combine (t[v*2], t[v*2+1]);
          end
End;
```

Осталось разобраться с ответом на запрос. Для этого мы так же, как и раньше, спускаемся по дереву, разбивая тем самым отрезок запроса [l, r] на несколько подотрезков, совпадающих с отрезками дерева отрезков, и объединяем ответы в них в единый ответ на всю задачу. Тогда понятно, что работа ничем не отличается от работы обычного дерева отрезков, только надо вместо простого суммирования/минимума/максимума значений использовать функцию combine. Приведённая ниже реализация немного отличается от реализации запроса : она не допускает случаев, когда левая граница l запроса превышает правую границу r (иначе возникнут неприятные случаи — какую структуру data возвращать, когда отрезок запроса пустой?..).

```
Function query (v, tl, tr, l, r: integer):data;
Var tm: integer;
begin
    if (l = tl) and (tr = r) then query :=t[v]
    else begin
             tm = (tl + tr) / 2;
            if (r <= tm) then query :=query (v*2, tl, tm, l, r)
            else  if (l > tm) then query :=query (v*2+1, tm+1, tr, l, r)
                   else query := combine (query (v*2, tl, tm, l, tm),
                                          query (v*2+1, tm+1, tr, tm+1, r)  );
         end
End;
```

Дерево Фенвика

Дерево Фенвика - это структура данных, дерево на массиве, обладающее следующими свойствами:

1) позволяет вычислять значение некоторой обратимой операции G на любом отрезке [L; R] за время O (log N);

2) позволяет изменять значение любого элемента за O (log N);

3) требует O (N) памяти, а точнее, ровно столько же, сколько и массив из N элементов;

4) легко обобщается на случай многомерных массивов.

Наиболее распространённое применение дерева Фенвика - для вычисления суммы на отрезке, т.е. функция G (X1, ..., Xk) = X1 + ... + Xk.

Описание:

Для простоты описания мы предполагаем, что операция G, по которой мы строим дерево, - это сумма. Пусть дан массив A[0..N-1]. Дерево Фенвика - массив T[0..N-1], в каждом элементе которого хранится сумма некоторых элементов массива A:

Ti = сумма Aj для всех F(i) <= j <= i,

где F(i) - некоторая функция, которую мы определим несколько позже.

Теперь мы уже можем написать псевдокод для функции вычисления суммы на отрезке [0; R] и для функции изменения ячейки:

```
Function  sum (r:integer):integer;
Var result:integer;
Begin
        result = 0;
        while r >= 0 do begin
                result :=result+ t[r];
                r := f(r) - 1;
        end;
        sum:= result;
End;
procedure inc (i, delta:integer);
begin
        для всех j, для которых F(j) <= i <= j
                t[j] := t[j] + delta;
End;
```

Функция sum работает следующим образом. Вместо того чтобы идти по всем элементам массива A, она движется по массиву T, делая "прыжки" через отрезки там, где это возможно. Сначала она прибавляет к ответу значение суммы на отрезке [F(R); R], затем берёт сумму на отрезке [F(F(R)-1); F(R)-1], и так далее, пока не дойдёт до нуля. Функция inc движется в обратную сторону - в сторону увеличения индексов, обновляя значения суммы Tj только для тех позиций, для которых это нужно, т.е. для всех j, для которых F(j) <= i <= j. Очевидно, что от выбора функции F будет зависеть скорость выполнения обеих операций. Сейчас мы рассмотрим функцию, которая позволит достичь логарифмической производительности в обоих случаях.

Определим значение F(X) следующим образом. Рассмотрим двоичную запись этого числа и посмотрим на его младший бит. Если он равен нулю, то F(X) = X. Иначе двоичное представление числа X оканчивается на группу из одной или нескольких единиц. Заменим все единицы из этой группы на нули, и присвоим полученное число значению функции F(X). Этому довольно сложному описанию соответствует очень простая формула: F(X) = X & (X+1),

где & - это операция побитового логического "И".

Нетрудно убедиться, что эта формула соответствует словесному описанию функции, данному выше. Нам осталось только научиться быстро находить такие числа j, для которых F(j) <= i <= j. Однако нетрудно убедиться в том, что все такие числа j получаются из i последовательными заменами самого правого (самого младшего) нуля в двоичном представлении. Например, для i = 10 мы получим, что j = 11, 15, 31, 63 и т.д. Как ни странно, такой операции (замена самого младшего нуля на единицу) также соответствует очень простая формула: H(X) = X | (X+1),

где | - это операция побитового логического "ИЛИ".

Реализация дерева Фенвика для суммы для одномерного случая

```
T:array [0..n-1] of integer;
Function  sum (r:integer):integer;
Var result:integer;
begin
        result = 0;
        while r >= 0 do begin
            r := (r and (r+1)) - 1);
            result := result + t[r];
        end;
        Sum:=result;
End;
Procedure  inc (i, delta:integer);
begin
        while i < n do begin
             i := (i or (i+1)))
             t[i] := t[i] + delta;
        end;
end
Function  sum2 (l, r: integer):integer;
Begin sum2:=sum (r) - sum (l-1);
End;
Procedure  init (var a:mass);
begin
        fillchar(a,sizeOf(a),0);
        for i := 0 to n-1 do
                inc (i, a[i]);
End;
```

Реализация дерева Фенвика для минимума для одномерного случая

Следует сразу заметить, что, поскольку дерево Фенвика позволяет найти значение функции в произвольном отрезке [0;R], то мы никак не сможем найти минимум на отрезке [L;R], где L > 0. Далее, все изменения значений должны происходить только в сторону уменьшения (опять же, поскольку никак не получится обратить функцию min). Это значительные ограничения.

```
const INF = 1000*1000*1000;
Function  getmin (r:integer):integer;
Var result:integer;
begin  result := INF;
       while r >= 0 do begin
           r := (r and (r+1)) - 1)
           result := min (result, t[r]);
        end;
    getmin:=result;
End;
Procedure update (i, new_val: integer);
begin
       while i < n do begin
             i := (i or (i+1)));
             t[i] := min (t[i], new_val);
       end;
End;
Procedure  init (var a:mass);
begin
       fillchar(a,sizeOf(a),0);
       for i := 0 to n-1 do
         update (i, a[i]);
End;
```

Реализация дерева Фенвика для суммы для двумерного случая

Как уже отмечалось, дерево Фенвика легко обобщается на многомерный случай.

```
Function sum (x, y: integer):integer;
Var result:integer;
Begin    result := 0;    i := x;
        while i >= 0 do begin
              i := (i and (i+1)) - 1);
              j = y;
              while j >= 0 do begin
                   j := (j and (j+1)) - 1);
                  result := result + t[i][j];
              end
      end;
   sum:= result;
End;
Procedure  inc (x, y, delta:integer);
Begin  i = x;
        while i < n do begin
              i := (i or (i+1))); j = y;
              while j < m do begin
                   j := (j or (j+1)));
                   t[i][j] := t[i][j] + delta;
              end
     end
End;
```

Поиск мостов

Пусть дан неориентированный граф. Мостом называется такое ребро, удаление которого делает граф несвязным (или, точнее, увеличивает число компонент связности). Требуется найти все мосты в заданном графе. Неформально эта задача ставится следующим образом: требуется найти на заданной карте дорог все "важные" дороги, т.е. такие дороги, что удаление любой из них приведёт к исчезновению пути между какой-то парой городов. Ниже мы опишем алгоритм, основанный на поиске в глубину, и работающий за время , где — количество вершин, — рёбер в графе. Заметим, что на сайте также описан онлайновый алгоритм поиска мостов — в отличие от описанного здесь алгоритма, онлайновый алгоритм умеет поддерживать все мосты графа в изменяющемся графе (имеются в виду добавления новых рёбер).

Алгоритм

Запустим обход в глубину из произвольной вершины графа; обозначим её через . Заметим следующий факт (который несложно доказать):

- Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины . Тогда, если текущее ребро таково, что из вершины и из любого её потомка в дереве обхода в глубину нет обратного ребра в вершину или какого-либо её предка, то это ребро является мостом. В противном случае оно мостом не является.

(В самом деле, мы этим условием проверяем, нет ли другого пути из в , кроме как спуск по ребру дерева обхода в глубину.) Теперь осталось научиться проверять этот факт для каждой вершины эффективно. Для этого воспользуемся "временами входа в вершину", вычисляемыми алгоритмом поиска в глубину.

- Итак, пусть tin[v] — это время захода поиска в глубину в вершину v. Теперь введём массив fup[v], который и позволит нам отвечать на вышеописанные запросы. Время fup[v] равно минимуму из времени захода в саму вершину tin[v], времён захода в каждую вершину p, являющуюся концом некоторого обратного ребра (v,p) , а также из всех значений fup[t_0] для каждой вершины t_0 , являющейся непосредственным сыном v в дереве поиска:

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$

- "tree edge" — ребро дерева)
- Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын t_0, что fup[t_0]<=tin[v], если fup[t_0]=tin[v], то это означает, что найдётся обратное ребро, приходящее точно в v; если же fup[t0]<tin[v], то это означает наличие обратного ребра в какого-либо предка вершины v.)Таким образом, если для текущего ребра (v,t_0) (принадлежащего дереву поиска) выполняется fup[t_0]>tin[v], то это ребро является мостом; в противном случае оно мостом не является.

Реализация

Если говорить о самой реализации, то здесь нам нужно уметь различать три случая: когда мы идём по ребру дерева поиска в глубину, когда идём по обратному ребру, и когда пытаемся пойти по ребру дерева в обратную сторону. Это, соответственно, случаи:

- — критерий ребра дерева поиска;
- — критерий обратного ребра;
- — критерий прохода по ребру дерева поиска в обратную сторону.

Таким образом, для реализации этих критериев нам надо передавать в функцию поиска в глубину вершину-предка текущей вершины.

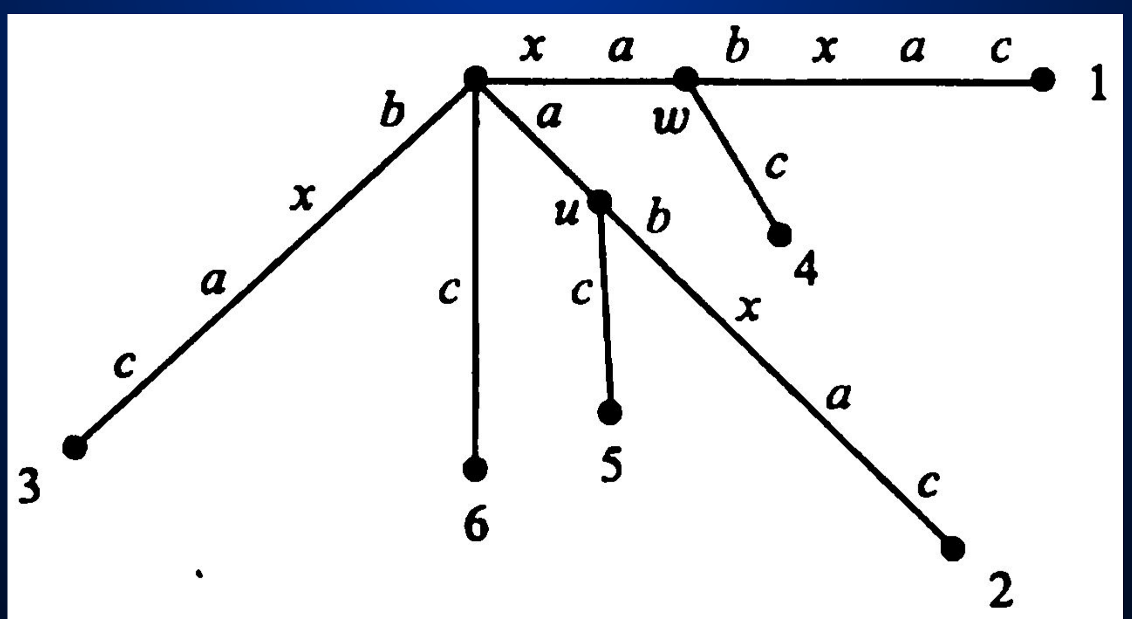
Здесь основная функция для вызова — это findbridges — она производит необходимую инициализацию и запуск обхода в глубину для каждой компоненты связности графа. При этом is_bridges(a,b) — это некая функция, которая будет реагировать на то, что ребро (a,b) является мостом, например, выводить это ребро на экран. Константе MaxN в самом начале кода следует задать значение, равное максимально возможному числу вершин во входном графе. Стоит заметить, что эта реализация некорректно работает при наличии в графе кратных рёбер: она фактически не обращает внимания, кратное ли ребро или оно единственно. Разумеется, кратные рёбра не должны входить в ответ, поэтому при вызове можно проверять дополнительно, не кратное ли ребро мы хотим добавить в ответ. Другой способ — более аккуратная работа с предками, т.е. передавать в не вершину-предка, а номер ребра, по которому мы вошли в вершину (для этого надо будет дополнительно хранить номера всех рёбер).

```
tin, fup :array[0..MaxN] of integer;     g :array[0..MaxN,0..MaxN] of integer;
Used: array [0..[MAXN] of boolean;  Timer: integer;
procedure dfs (v, p :integer);
Var to:integer;
begin
        used[v] := true; p:=-1;
        tin[v] := fup[v] := timer; inc(timer);
        for i:=0 to размер(g[v]) do begin
                to := g[v][i];
                if (to <> p)  then begin
                   if (used[to])
                   then   fup[v] := min (fup[v], tin[to])
                   else begin
                            dfs (to, v);
                            fup[v] := min (fup[v], fup[to]);
                            if (fup[to] > tin[v]) thenIS_BRIDGE(v,to);
                       end
        end
End;
 procedure find_bridges() ;
Var i:integer;
begin
        timer := 0;       for i:=0to n do used[i] = false;
        for i:=0 to n do
                if (not used[i]) then dfs (i);
End;
```

Суффиксное дерево (gusfield.djvu)

- Суффиксное дерево — это структура данных, которая выявляет внутреннее строение строки более глубоко, чем основной препроцессинг. Суффиксные деревья можно использовать для решения задачи о точных совпадениях . За линейное время (достигая той же границы для наихудшего случая, как в алгоритмах Кнута-Морриса-Пратта и Бойера-Мура), но их подлинное превосходство - . становится ясным при решении за линейное время многих строковых задач, более • Сложных, чем точные совпадения. Более того, суффиксные деревья наводят мост между задачами точного совпадения и неточного совпадения.
- Классическим приложением для суффиксных деревьев служит задача о подстроке. Пусть задан текст T длины m. За препроцессное время O (m), т.е. линейное, нужно приготовиться к тому, чтобы, получив неизвестную строку S длины n, за время O(n) либо найти вхождение S в T, либо определить, что S в T не входит. Это значит, что допустим препроцессинг со временем, пропорциональным длине текста, но после этого поиск строки S должен выполняться за время, пропорциональное длине S, независимо от длины T. Эти границы достигаются применением суффиксного дерева. Оно строится для текста за время O(т) в препроцессной стадии; а затем, используя это суффиксное дерево, алгоритм, получив строку длины n на входе, ищет S за время O(n).

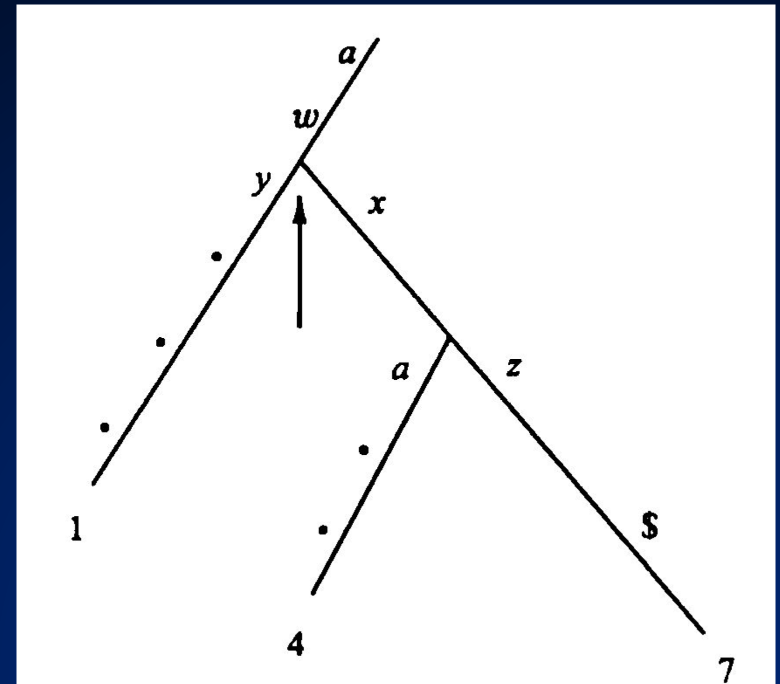
• Суффиксное дерево Г для m-символьной строки S — это ориентированное дерево с корнем, имеющее ровно m листьев, занумерованных от 1 до m. Каждая внутренняя вершина, отличная от корня, имеет не меньше двух детей, а каждая дуга помечена непустой подстрокой строки S (дуговой меткой). Никакие две дуги, выходящие из одной и той же вершины, не могут иметь пометок, начинающихся с одного и того же символа. Главная особенность суффиксного дерева заключается в том, что для каждого листа i конкатенация меток дуг на пути от корня к листу i в точности составляет (произносит) суффикс строки S, который начинается в позиции i. То есть, этот путь произносит S[i..m]. Например, суффиксное дерево для строки xabxac показано на рисунке. Путь из Корня к листу 1 произносит полную строку S = xabxac, тогда как путь до листа 5 произносит суффикс ac, который начинается в позиции 5 строки S



- Как уже констатировалось, определение суффиксного дерева для S не гарантирует, что такое дерево действительно существует для любой строки S. Трудность в том, что если один суффикс совпадает с префиксом другого суффикса, то построить суффиксное дерево, удовлетворяющее данному выше определению, невозможно, поскольку путь для первого суффикса не сможет закончиться в листе. Например, если удалить последний символ из строки xabxac, образовав строку xabxa, то суффикс xa будет префиксом суффикса xabxa, так что путь, произносящий xa, не будет заканчиваться в листе.
- Во избежание этой трудности предположим (это предположение выполняется на рис.), что последний символ S нигде больше в строку не входит. При такому условии никакой суффикс строки не сможет быть префиксом другого суффикса. Чтобы обеспечить это на практике, мы может добавить в конце S какой-либо символ, не входящий в основной алфавит (из символов которого составлена строка). Здесь в качестве такого "завершающего" символа мы используем $.

Пример

- Прежде чем вдаваться в детали методов построения суффиксных деревьев, посмотрим, как суффиксное дерево для строки используется при решении задачи о точных совпадениях. Задан образец P длины n и текст T длины m, нужно найти все вхождения P в T за время O(n + m). Мы уже видели несколько решений этой задачи. Суффиксные деревья дают другой подход. Построим суффиксное дерево Г для текста T за время O (m). Затем будем искать совпадения для символов из P вдоль единственного пути в Г до тех пор, пока либо P не исчерпается либо очередное совпадение не станет невозможным. Во втором случае P в T не входит. В первом случае каждый лист в поддереве, идущем из точки последнего совпадения, имеет своим номером начальную позицию P в T, а каждая начальная позиция P в T нумерует такой лист.
- Ключом к пониманию первого случая (когда все символы из P совпали с путем в T) служит такое наблюдение: P входит в T, начиная с позиции j, в том и только том случае, если P является префиксом T[j..m]. Но это происходит тогда и только тогда, когда P помечает начальную часть пути от корня до листа j. Именно по нему и проходит алгоритм сравнения. Этот совпадающий путь единственен, так как никакие две дуги, выходящие из одной вершины, не имеют меток, начинающихся с одного и того же символа. И поскольку мы предположили, что алфавит конечен, работа в каждой вершине занимает константное время, а значит, время на проверку совпадения P с путем пропорционально длине P.



- На рис. показан фрагмент суффиксного дерева для строки T = awyawxawxz. Образец P = aw входит в T три раза, начинаясь в позициях 1, 4 и 7. Образец P совпадает с путем вниз до вершины, указанной стрелкой, и, как полагается, листья ниже этой точки имеют номера 1, 4 и 7. Если P полностью совпадает с некоторым путем в дереве, алгоритм может найти все начальные позиции P в T проходом по поддереву вниз от конца совпавшего пути. При этом просмотре нужно просто собрать все номера встретившихся листьев. Таким образом, все вхождения P в T могут быть найдены за время O(n+m). Ранее предложенные алгоритмы тратили время O(n) на препроцессинг P, а затем время O(m) на поиск. Напротив, при использовании суффиксного дерева тратится время O(m) на препроцессинг и O(n + к) на поиск, где к — число вхождений P в T.

- Чтобы собрать к начальных позиций Р, обойдем поддерево из конца совпадающего пути, используя обход за линейное время (скажем, обход в глубину), и отметим i, встретившиеся номера листьев. Так как каждая внутренняя вершина имеет не меньше двух детей, число встретившихся листьев пропорционально числу пройденных дуг, так что время обхода равно О(к), даже несмотря на то, что общая строковая глубина этих О(к) дуг может быть сколь угодно больше к. Если требуется найти только одно вхождение Р, то можно чуть-чуть изменить предварительную обработку так, что поисковое время уменьшится с O(n + к) до O(n). Идея в том, чтобы записать в каждой вершине один номер (скажем, наименьший) листа в его поддереве. Это можно сделать на препроцессной фазе за время 0(m) обходом в глубину дерева Г. После такой подготовки в фазе поиска число, записанное в вершине или ниже конца совпадения, и дает начальную позицию Р в Т.

Наивный алгоритм построения суффиксного дерева

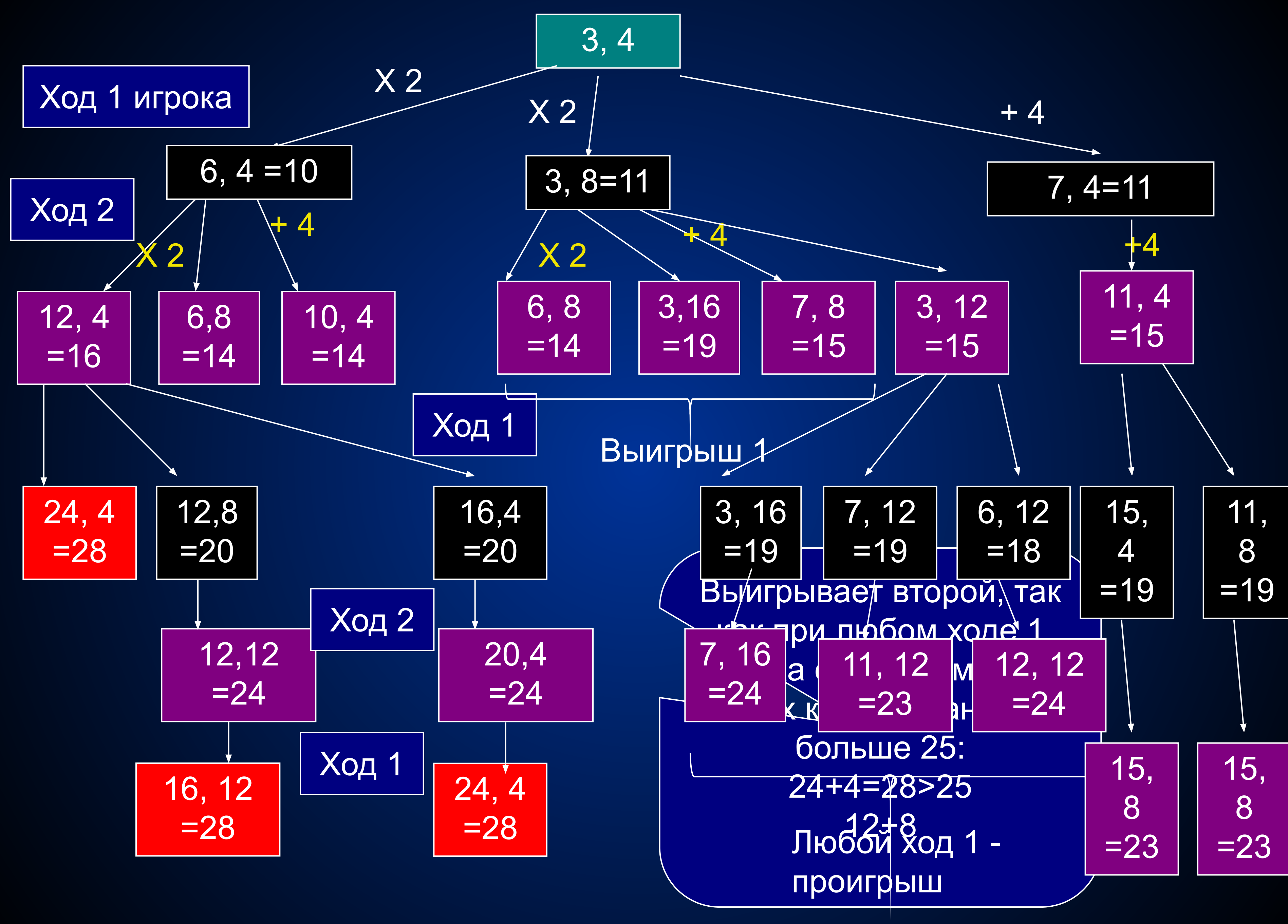
- Чтобы еще подкрепить определение суффиксного дерева продемонстрируем алгоритм непосредственного построения суффиксного дерева для строки S. Этот наивный метод сначала помещает в дерево простую дугу для суффикса S[1..m]$ (для всей строки); затем последовательно вводит в растущее дерево суффиксы S[i..m]$ для i от 2 до m. Пусть N, обозначает промежуточное дерево, в которое входят все суффиксы от 1 до i. S Подробнее, дерево Ni состоит из одной дуги, идущей от корня дерева к вершине, с номером 1. Эта дуга имеет метку S$. Дерево N i+1 строится из дерева N i, следующим образом. Начиная из корня Ni, найти самый длинный путь от корня, метка которого совпадает с префиксом строки S[i + 1..m]$. Этот путь можно отыскать последовательным сравнением и поиском совпадений символов суффикса S[i + 1..m]$ с символами вдоль единственного пути из корня до тех пор, пока не обнаруживается место, где очередного совпадения нет. Совпадающий путь единственен, так как никакие две дуги, выходящие из одной вершины, не могут иметь одинакового символа в начале их меток. В какой-то точке совпадения прекратятся, так как никакой суффикс S$ не является префиксом другого суффикса S$. Когда такая точка будет достигнута, то алгоритм останавливается либо в вершине, скажем, w, либо в середине какой-то дуги. Если в середине дуги, скажем; (u, v), то он разбивает- дугу (u, v) на две дуги, вводя новую вершину, которую мы обозначим через w, сразу после последнего совпадающего символа S[i + 1..m] и перед первым несовпадающим символом. Новая дуга (u, w) имеет меткой часть метки (u, v), которая совпадает с частью S[i + l..m], а новая дуга (w, v) помечена оставшейся частью метки (u, v). Далее (независимо от того, создана вершина w заново или она уже существовала) алгоритм создает новую дугу (w, i + 1), идущую из w в новый лист с номером i + 1, и эта дуга помечается частью суффикса S[i + l..m]$, не нашедшей совпадения.
- Теперь дерево содержит единственный путь из корня к листу i + 1, и этот путь имеет метку S[i + 1..m]$. Заметим, что все дуги, выходящие из новой вершины w, имеют метки, отличающиеся первыми символами, и отсюда мы по индукции получаем, что никакие две дуги, выходящие из одной вершины, не имеют меток с одинако- одинаковыми первыми символами.

Игры двух лиц

- Игры, в которых участвуют два игрока, знакомы нам с детства
- от крестиков-ноликов на поле 3х3 до шашек и шахмат. Как правило, такие игры являются антагонистическими - выигрыш одного игрока означает проигрыш другого. Игры также бывают с полной и с неполной информацией в зависимости от того, все ли известно игроку о позиции своей и соперника. Примеры игр с полной информацией варианты крестиков ноликов, шахматы или шашки, с неполной домино или карточные игры, в которых соперник скрывает, что у него "на руках".
- В процессе игры ее участники, как правило, по очереди совершают ходы, благодаря которым образуются позиции в игре. Варианты процесса любой игры обычно можно представить корневым ориентированным деревом (иногда
- направленным графом). Узлы в этом дереве представляют позиции (корневой узел - исходную позицию, узлы листья - заключительные позиции, из которых нельзя сделать ход). Дуги дерева представляют ходы.

Демонстрационный вариант ЕГЭ 2011 г.

- *Два игрока играют в следующую игру. Перед ними лежат две кучки камней, в первой из которых 3, а во второй 4 камня.*
  У каждого игрока неограниченно много камней. Игроки ходят по очереди. Ход состоит в том, что игрок или удваивает число камней в какой-то кучке или добавляет 4 камня в какую-то кучку. Игрок, после хода которого общее число камней в двух кучках становится больше 25, проигрывает.
- *Кто выигрывает при безошибочной игре обоих игроков – игрок, делающий первый ход, или игрок, делающий второй ход? Каким должен быть первый ход выигрывающего игрока? Ответ обоснуйте.*

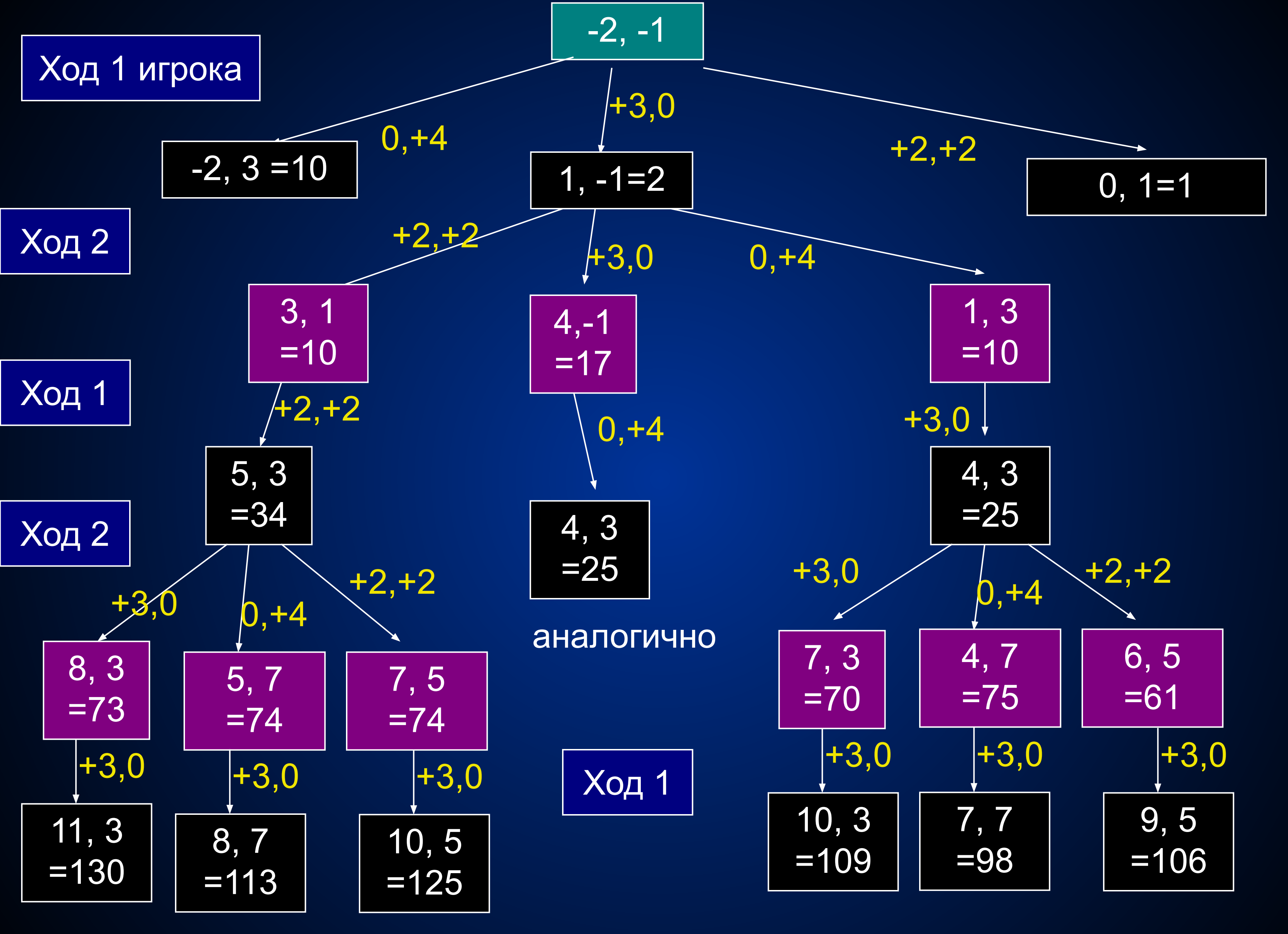


3, 4
Ход 1 игрока
X 2
X 2
+ 4
6, 4 =10
3, 8=11
7, 4=11
Ход 2
X 2
+ 4
X 2
+ 4
+4
12, 4 =16
6,8 =14
10, 4 =14
6, 8 =14
3,16 =19
7, 8 =15
3, 12 =15
11, 4 =15
Ход 1
Выигрыш 1
24, 4 =28
12,8 =20
16,4 =20
3, 16 =19
7, 12 =19
6, 12 =18
15, 4 =19
11, 8 =19
Выигрывает второй, так
Ход 2
12,12 =24
20,4 =24
7, 16 =24
11, 12 =23
12, 12 =24
больше 25:
24+4=28>25
12+8
Любой ход 1 - проигрыш
Ход 1
16, 12 =28
24, 4 =28
15, 8 =23
15, 8 =23

- Выигрывает второй игрок.
- Для доказательства рассмотрим неполное дерево игры, где в каждой ячейке записаны пары чисел, разделенные запятой. Эти числа соответствуют количеству камней на каждом этапе игры в первой и второй кучках соответственно. Чтобы доказать победу второго игрока необходимо доказать, что второй игрок побеждает вне зависимости от хода первого, то есть рассмотреть все возможные ходы первого игрока и в каждом из них доказать, что победит второй.
- Рассмотрим ход 1 игрока (6,4). У второго игрока есть три варианта ответа ((12,4), (6,8), (10,4)). Второй игрок играет безошибочно, следовательно не будет делать ходы, которые ведут его к поражению (дают выигрыш 1). Поэтому он сходит (12,4), удвоив количество камней в первой кучке. Далее ходит 1 игрок: ход (24,4) сразу приводит его к поражению, а два хода (16, 4) и (12, 8) откладывают поражение на 1 ход.
- Рассмотрим ход 1 игрока (3,8). У второго игрока есть четыре варианта ответа, но только один ведет к гарантированной победе. Поэтому он сходит (3,12), увеличив количество камней во второй кучке на 4. Далее ходит 1 игрок: на любой его ход из 4, у второго игрока есть выигрышный ответ.
- Рассмотрим ход 1 игрока (7,4). У второго игрока есть четыре варианта ответа, но только один ведет к гарантированной победе. Поэтому он сходит (11,4), увеличив количество камней в первой кучке на 4. Далее ходит 1 игрок: на любой его ход из 4, у второго игрока есть выигрышный ответ.

Демонстрационный вариант ЕГЭ 2010 г.

- Два игрока играют в следующую игру. На координатной плоскости стоит фишка. В начале игры фишка находится в точке с координатами (–2,–1).
- Игроки ходят по очереди. Ход состоит в том, что игрок перемещает фишку из точки с координатами (x, y) в одну из трех точек: (x+3,y), (x,y+4), (x+2,y+2). Игра заканчивается, как только расстояние от фишки до начала координат превысит число 9. Выигрывает игрок, который сделал последний ход. Кто выигрывает при безошибочной игре – игрок, делающий первый ход, или игрок, делающий второй ход? Каким должен быть первый ход выигрывающего игрока? Ответ обоснуйте.


-2, -1
Ход 1 игрока
0,+4
+3,0
+2,+2
-2, 3 =10
1, -1=2
0, 1=1
Ход 2
+2,+2
+3,0
0,+4
3, 1
=10
4,-1
=17
1, 3
=10
Ход 1
+2,+2
0,+4
+3,0
5, 3
=34
4, 3
=25
4, 3
=25
Ход 2
+3,0
0,+4
+2,+2
аналогично
+3,0
0,+4
+2,+2
8, 3
=73
5, 7
=74
7, 5
=74
7, 3
=70
4, 7
=75
6, 5
=61
+3,0
+3,0
+3,0
Ход 1
+3,0
+3,0
+3,0
11, 3
=130
8, 7
=113
10, 5
=125
10, 3
=109
7, 7
=98
9, 5
=106

- Выигрывает первый игрок, своим первым ходом он должен поставить фишку в точке с координатами (1,-1). Для доказательства рассмотрим неполное дерево игры, для доказательства победы 1 игрока достаточно найти одну ветку, приводящую к победе, рассматривать остальные не обязательно.
- Рассмотрим всевозможные ходы 2 игрока и докажем, что при любом его ходе первый игрок сможет выиграть.
- Рассмотрим ход 2 игрока (3,1). У первого игрока есть три варианта ответа ((6,1), (3,5), (5,3)). Первый игрок играет безошибочно, следовательно не будет делать ходы, которые ведут его к поражению (дают выигрыш 2). Поэтому он сходит (5,3), увеличив на 2 обе координаты. Далее ходит 2 игрок: любой его ход не позволяет ему перейти расстояние в 9 единиц $9^2=81$, зато второй игрок выигрывает следующим ходом.
- Рассмотрим ход 2 игрока (1,3). У первого игрока есть три варианта ответа, но только один ведет к гарантированной победе. Поэтому он сходит (4,3), увеличив на 3 координату х. Далее ходит 1 игрок: на любой его ход из 3, у второго игрока есть выигрышный ответ.
- Рассмотрим ход 2 игрока (4,-1). Первый игрок ходит в позицию (4,3) и играет аналогично рассмотренному.

Игра Ним

Формулировка

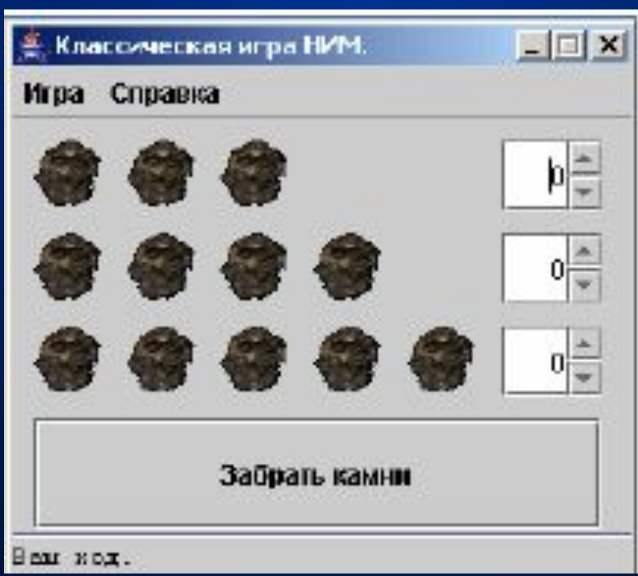
- **Ним** — математическая игра, в которой два игрока по очереди берут предметы, разложенные на несколько кучек. За один ход может быть взято любое количество предметов (большее нуля) из одной кучки. Выигрывает игрок, взявший последний предмет.

Правила игры Ним

Ним — игра для двух игроков, каждый из которых по очереди делает ход. Перед игроками располагается поле с камнями. Известны различные варианты игры Ним. В данном случае правила игры таковы:

- фишки раскладываются в несколько рядов;
- игроки по очереди забирают камни из любого ряда;
- не разрешается за один ход брать камни из нескольких рядов;
- за один ход игрок должен взять хотя бы один камень;
- выигрывает тот, кто возьмет последний камень.

В наиболее известном варианте игры 12 фишек раскладываются так, как показано на рис. 1



Каждую комбинацию фишек (камней) он назвал либо опасной, либо безопасной. Если позиция, создавшаяся после очередного хода игрока, гарантирует ему победу, то она называется безопасной. В противном случае, позиция опасная.
Ч. Бутон строго доказал, что любую опасную позицию всегда можно превратить в безопасную с помощью соответствующего хода. С другой стороны, если перед очередным ходом игрока уже сложилась безопасная позиция, то любой его ход превращает позицию в опасную. Таким образом, оптимальная стратегия игрока состоит в том, чтобы каждым ходом опасную позицию превращать в безопасную и заставлять соперника превращать позицию в опасную. Использование оптимальной стратегии гарантирует победу игроку тогда и только тогда, когда он ходит первым и начальная позиция фишек опасна, или он когда он ходит вторым, а начальная позиция безопасна.
Для того, чтобы определить, опасна ли позиция, или она безопасна, требуется количество фишек в каждом ряду записать в двоичной системе счисления и вычислить сумму чисел в каждом столбце (разряде). Если эта сумма четна, то позиция безопасна. Если сумма хотя бы в одном разряде нечетна, то позиция опасна. Эквивалентный, но более простой способ оценки позиции состоит в том, чтобы представить количество фишек в каждой кучке (в каждом ряду) в виде суммы степеней двойки, а затем вычеркнуть все пары одинаковых степеней и просуммировать оставшиеся степени. В результате получается так называемая «ним-сумма» для данной позиции. Иногда это число называют также «числом Ганди», или «числом Спрэга-Ганди» в честь Р. Спрэга и П. Ганди, которые независимо друг от друга разработали общую теорию такого рода игр, основанную на численных оценках каждой игровой позиции.

Предположим, например, что в начале игры имеются три кучки – из трех, пяти и семи фишек. Запишем эти числа в следующем виде.

3 = 2 + 1;

5 = 4 + 1;

7 = 4 + 2 + 1.

Вычеркнем, как показано выше, соответствующие пары четверок, двоек и единиц. Сумма того, что осталось равна единице – это и есть ним-сумма для данной позиции. Позиция безопасна в том и только в том случае, если ним-сумма для нее равна нулю. В противном случае позиция оказывается опасной (как в рассмотренном примере).

Для того, чтобы обеспечить свой выигрыш, имея перед собой опасную позицию, игроку следует превратить ее в безопасную. В данном случае если взять одну фишку из любой кучки, то ним-сумма позиции уменьшится до нуля. Как и во всех играх такого рода, в игре Ним можно играть «наоборот», когда тот игрок который забирает последнюю фишку считается проигравшим. Для многих игр со взятием фишек стратегия игры «наоборот» чрезвычайно сложна, однако для игры «Ним» в этом случае требуется ввести в стратегию лишь достаточно тривиальные изменения, касающиеся только конца партии. В самом деле, для того, чтобы выиграть требуется просто придерживаться обычной стратегии, причем таким образом, чтобы оставить нечетное число кучек, состоящих из одной фишки.

Как же определять кто победит при оптимальной игре обеих соперников? Об этом нам говорит теорема Бутона:
Пусть имеется N кучек нима, количество камешков в каждой из них x1,x2,.. xn. Тогда текущая позиция проигрышная если $x1 \otimes x2 \otimes x3 \otimes \ldots \otimes xn$, где $\otimes$ — значок побитовой суммы по модулю 2 (xor). Доказательство теоремы строится на том, что из любой позиции с нулевой ним-суммой перейти можно только в позицию с ненулевой (за исключением пустых кучек), и из любой позиции с ненулевой ним-суммой всегда можно перейти в позицию с нулевой. Значит после правильной последовательности ходов игрок с ненулевой ним-суммой всегда побеждает, так как после своих ходов оставляет нулевую ним-сумму, строго уменьшая количество камней, следовательно рано или поздно именно он заберет последние камни.
Посмотрим как работает эта теорема на примере, пусть мы имеем три кучки с количеством 3, 8, 15 камней в каждой. Запишем эти числа в столбик в двоичном представлении и вычислим xor, он равен нулю, значит второй игрок, назовем его Y, находится в выигрышной позиции. Его стратегия состоит в том, чтобы любым своим ходом оставлять позицию с нулевой ним-суммой.

- Рассмотрим возможные ходы, где Y придерживается своей стратегии и в итоге побеждает:

111
1000
[1111]
0000
→ I X
111
[1000]
1111
→ II Y
111
[111]
000
→ III X
[111]
100
011
→ IV Y

→ V X
100
[100]
000
→ VI Y
[1 0]
1
1 1
→ VII X
[1]
1
0
→ VIII Y
[1]
1
→ IX X
P

Получается, стратегия игры в ним проста, еще проще можно определить победителя при оптимальной стратегии, теперь перейдем к нашей задаче мизерного нима. Правила у него те же, за исключением одного: теперь тот кто забирает последний камень проигрывает. Наша задача состоит в том чтобы определить по размерам кучек кто победит при оптимальной стратегии. На этот раз игра мизерная, значит её не всегда можно свести к ниму, у нас нет четких алгоритмов и подходов для решения этой задачи, их придется придумывать с нуля. В предложенном мною решении мы попытаемся свести мизерный ним к обычному, в данном случае это возможно, но после некоторых умозаключений.

- Легко видеть что в первом случае чтобы выдать ответ достаточно проверить четность количества кучек. В случае четности побеждает X, в случае нечетности Y. Теперь рассмотрим второй случай. Можно заметить что оптимальная стратегия тут будет заключаться в том чтобы свести игру к указанной нами позиции с K единичными кучками и X0 кучкой размера больше одного, назовем эту позицию A. Причем как бы не проходила игра, позиция A будет рано или поздно достигнута одним из игроков. Кто попадет в эту позицию, тот победит, причем как в мизерном ниме, так и в обычном, а из этого следует то, что мы можем проверить начальную позицию на выигрышность, как будто бы играем в обычный ним, вычислив xor всех кучек. Если xor нулевой, то первым в позицию A попадет второй игрок, где ему ничего не будет мешать сделать выигрышный в мизерном ниме ход. Иначе этот ход достанется иксу.

Еще вариант игры Ним

Ним — одна из самых старых и занимательных математических игр. Играют в нее вдвоем. Дети используют для игры камешки или клочки бумаги, взрослые предпочитают раскладывать монетки на стойке бара. В наиболее известном варианте нима 12 монет раскладывают в три ряда так, как показано на рисунке.

Правила нима просты. Игроки по очереди забирают по одной или нескольку монет из любого ряда. Выигрывает тот, кто возьмет последнюю монету. Можно играть и наоборот: считать того, кто возьмет последнюю монету, проигравшим. Хороший игрок вскоре обнаружит, что и в том и в другом варианте можно добиться победы, если после его хода останется два одинаковых ряда монеток (то есть с одним и тем же числом монет в каждом ряду), причем в каждом ряду будет находиться более одной монетки. Выиграть можно и в том случае, если в первом ряду останется одна, во втором — две и в третьем — три монетки. Тот, кто открывает игру, наверняка побеждает, если первым ходом он забирает две монетки из верхнего ряда, а затем рационально продолжает игру.

Казалось, что анализ столь простой игры не может привести к каким-либо неожиданностям, однако в начале века было сделано удивительное открытие. Обнаружилось, что ним допускает обобщение на любое число рядов с любым числом фишек в каждом ряду и что с помощью до смешного простой стратегии, используя двоичную систему счисления, любой желающий может стать непобедимым игроком. Полный анализ и доказательство существования оптимальной стратегии впервые опубликовал в 1901 году Чарлз Л. Бутон, профессор математики Гарвардского университета. Бутон и назвал игру «ним» от устаревшей формы английских глаголов «стянуть», «украсть».

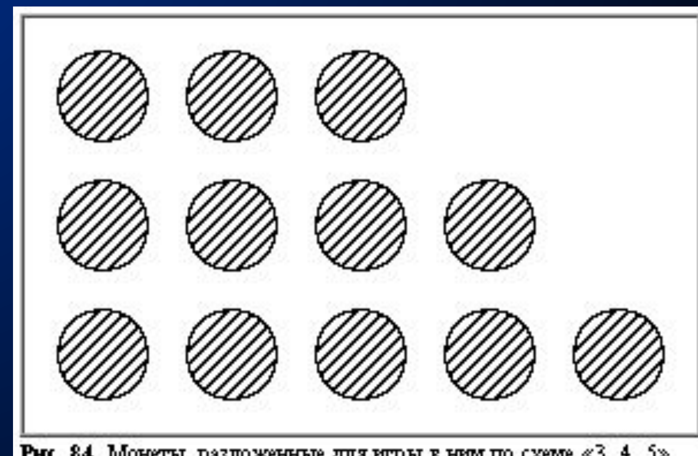


Рис. 84. Монеты, разложенные для игры в ним по схеме «3, 4, 5».

- Каждую комбинацию фишек в обобщенной игре Бутон назвал либо «опасной», либо «безопасной». Если позиция, создавшаяся после очередного хода игрока, гарантирует ему выигрыш, она называется безопасной; в противном случае позиция называется опасной. Так, при игре в ним по описанной выше схеме «3, 4, 5» (рис. 84) первый игрок окажется в безопасной позиции, взяв две монетки из верхнего ряда. Любую опасную позицию, сделав соответствующий ход, всегда можно превратить в безопасную. Каждая безопасная позиция становится опасной после любого хода. Следовательно, рациональная игра заключается в том, чтобы каждый раз превращать опасную позицию в безопасную.
- Чтобы определить, опасна или безопасна данная позиция, число фишек в каждом ряду нужно записать в двоичной системе. Если сумма чисел в каждом столбце (разряде) равна нулю или четна, то позиция безопасна. Если же сумма нечетна хотя бы в одном разряде, то позиция опасна.

Двоичные числа для игры в ним.

	16	8	4	2	1		16	8	4	2	1
1											
2					1	11		1	0	1	1
3				1	0	12		1	1	0	0
4				1	1	13		1	1	0	1
5			1	0	0	14		1	1	1	0
6			1	0	1	15		1	1	1	1
7			1	1	0	16	1	0	0	0	0
8			1	1	1	17	1	0	0	0	1
9		1	0	0	0	18	1	0	0	1	0
10		1	0	0	1	19	1	0	0	1	1
		1	0	1	0	20	1	0	1	0	0

В двоичной системе нет ничего сверхъестественного. Это всего лишь способ записи чисел в виде суммы степеней двойки. В помещенной здесь таблице приведена двоичная запись чисел от 1 до 20. Обратите внимание на то, что, двигаясь справа налево, вы каждый раз попадаете в столбец, отвечающий большей степени двойки, чем предыдущий (то есть переходите ко все более старшим двоичным разрядам). Так, двоичная запись 10 101 говорит нам, что к 16 нужно прибавить 4 и 1, а это дает десятичное число 21. Записывая в двоичной системе число фишек в каждом ряду, расставленных по схеме «3, 4, 5», мы получим

	4	2	1
3		1	1
4	1	0	0
5	1	0	1
Итого	2	1	2

Сумма цифр в среднем столбце равна 1 — нечетному числу, что свидетельствует об опасности данной позиции. Поэтому первый игрок может сделать ее безопасной. Как уже объяснялось, именно это он и делает, когда забирает из верхнего ряда две монетки. В результате в верхнем ряду остается лишь 1 монетка (двоичное число также 1) и нечетное число в последовательности сумм чисел по столбцам пропадает. Перепробовав остальные ходы, читатель убедится в том, что только указанный ход может сделать исходную позицию безопасной.

Если в каждом ряду стоит не более 31 фишки, то любую позицию легко проанализировать, использовав в качестве вычислительной машины (работающей в двоичной системе!) пальцы левой руки. Предположим, что в начальной позиции в первом ряду стоит 7, во втором 13, в третьем — 24 и в четвертом — 30 фишек. Вы должны сделать первый ход. Опасна или безопасна исходная позиция? Поверните левую руку с растопыренными пальцами ладонью к себе. Большой палец будет означать коэффициент при 16, указательный — коэффициент при 8, средний — при 4, безымянный — при 2 и мизинец — коэффициент при 1. Для того чтобы ввести в вашу вычислительную машину число 7, прежде всего нужно загнуть палец, соответствующий наибольшей степени двойки, входящей в 7. Такой степенью является 4, поэтому вы загибаете средний палец. Продолжая двигаться направо, добавляйте степени двойки до тех пор, пока вы в сумме не получите 7. Для этого вам придется загнуть средний, безымянный пальцы и мизинец. Три остальных числа — 13, 24 и 30 — вводятся в вашу вычислительную машину точно так же, но, поскольку вам требуется вычислить сумму чисел, стоящих в столбцах при одной и той же степени двойки, вы, дойдя до согнутого пальца, который вам нужно согнуть еще раз, просто разгибаете его.

Независимо от количества рядов позиция безопасна, если по окончании работы вашей вычислительной машины на левой руке не останется ни одного загнутого пальца. Это означает, что любым ходом вы наверняка сделаете положение опасным и заведомо проиграете, если ваш противник знает о ниме столько же, сколько и вы. В приведенном нами примере большой и указательный пальцы останутся согнутыми. Это говорит о том, что позиция опасна и что, сделав правильный ход, вы сможете выиграть. Поскольку опасных позиций больше, чем безопасных, у первого игрока при случайном выборе начальной позиции гораздо больше шансов выиграть, чем проиграть.

Итак, вы знаете, что позиция 7,13, 20, 30 опасна. Как найти ход, превращающий ее в безопасную? На пальцах найти нужный ход довольно трудно, поэтому лучше всего записать четыре двоичных числа в последовательности

	16	8	4	2	1
7			1	1	1
13		1	1	0	1
24	1	1	0	0	0
30	1	1	1	1	0
Итого	2	3	3	2	2

Найдем самый левый столбец с нечетной суммой цифр. Изменив любой ряд с единицей в этом столбце, мы сможем превратить позицию в безопасную. Предположим, что вы хотите взять одну или несколько фишек из второго ряда. Замените ту единицу, которая вам мешала, нулем, а остальные цифры, расположенные правее ее, подберите так, чтобы ни в одном столбце сумма цифр не была нечетной. Единственный способ сделать это состоит в том, чтобы выбрать в качестве второго двоичного числа единицу. Иначе говоря, вы должны взять либо четыре фишки из третьего ряда, либо двенадцать фишек из последнего, четвертого ряда.

Полезно помнить, что для верного выигрыша фишек в двух рядах должно оставаться поровну. Поэтому при очередном ходе вы должны уравнивать число фишек в каких-нибудь двух рядах. И это правило и тот способ анализировать позиции с помощью двоичных чисел, о котором мы рассказали выше, пригодны при обычной игре, когда победителем считается тот, кто забирает последнюю фишку. К счастью, для того чтобы приспособить эту стратегию к игре «наоборот», достаточно внести лишь довольно тривиальное изменение в правило. Когда в игре «наоборот» наступит такой момент (а он непременно наступит), что только в одном ряду число фишек будет больше 1, вы должны взять из этого ряда либо все фишки, либо оставить одну фишку, чтобы число рядов, состоящих из одной-единственной фишки, стало нечетным. Например, если фишки расставлены по схеме 1, 1, 1, 3, вы должны взять все фишки, стоящие в последнем ряду. Если бы фишки были расставлены по схеме 1, 1, 1, 1, 1, 8, то из последнего ряда следовало бы взять семь фишек. Необходимость в изменении стратегии возникает лишь в самом конце игры, когда хорошо видно, что следует делать для того, чтобы добиться выигрыша.

Мизер

В этом варианте игрок, взявший последний объект, проигрывает.

Выигрышная стратегия совпадает с выигрышной стратегией обычной игры до того момента, когда в результате хода игрока на столе должно остаться некоторое количество кучек с единственным предметом в каждой из них. В случае мизера игрок должен оставить нечётное количество кучек, тогда как выигрышная стратегия обычной игры требует оставить чётное количество кучек, чтобы ним-сумма равнялась нулю.

Мультиним

- Более общий случай игры Ним был предложен Муром (Eliakim Moore). В игре Nim_i игрокам разрешается брать предметы из максимум i кучек. Легко видеть, что обычная игра ним является Nim_1 .
- Решение такой задачи удивительно просто. Запишем размеры каждой кучки в двоичной системе счисления. Затем просуммируем эти числа в i-ичной системе счисления без переносов разрядов. Если получилось число 0, то текущая позиция проигрышная, иначе — выигрышная (и из неё есть ход в позицию с нулевой величиной).

Анализ позиций и выбор хода

- Задача. Касса содержит С копеек. Два игрока поочередно забирают из кассы от 1 до М копеек. Выигрывает игрок, после хода которого касса станет пустой. Программа должна читать положительные числа С и М (от 1 до 10^4) выбирать право первого хода и выигрывать.
- Пример. При С= 17 и М=9 последовательность ходов (первый 5, второй 6, первый 6) означает выигрыш первого игрока, поскольку 17 - 5 – 6- 6= 0.
- Анализ и решение задачи. В любой игре присутствует такое понятие, как позиция. Неформально говоря, это совокупность текущих параметров игры, которую изменяют игроки своими ходами и по которой в конце концов определяется выигрыш, кроме того, позиция должна однозначно задавать возможные ходы из нее. В данной задаче позицией естественно считать остаток в кассе R и номер игрока, имеющего право хода. Предположим, что сейчас наш ход. Начнем с ситуаций, в которых остаток R мал. Если R=0, мы уже проиграли. Если R=1, R=2, ..., R=M, У нас есть победный ход число R, после которого сумма станет равной 0. Если R = M+ 1, то любой наш ход сделает R равным одному из чисел 1, 2, ..., М, и тогда победный ход сделает соперник. Итак, остатки R=0 и R=M+1 характеризуют проигрышную позицию. Каким бы ни был наш ход, соперник выиграет (конечно, если не допустит ошибки). В тоже время, в позициях с R=1,2, ... или М есть победный ход. Продолжим анализ. Если R = M+2, M+3, ... или M+M+ 1, то наш соответствующий ход 1, 2, ..., М приведет соперника к проигрышной позиции С R=M+l. Значит, в этих позициях можно выиграть. Но если R=2M+2, любой наш ход приведет в позицию с R в пределах от M+2 до 2M+ 1, начиная с которой выиграет соперник...
- По этим наблюдениям нетрудно догадатъся, что позиции с R=0, M+1, 2(M+1), 3(M+1), ... являются проигрышными, а во всех остальных существуют ходы, "загоняющие" соперника в одну из ЭТИХ проигрышных позиций, Т.е. остальные позиции выигрышные.
- Итак, при С mod (M+1)<>0 исходная позиция является выигрышной, и наш первый ход число С mod (M+1). Иначе позиция проигрышная, и пусть начинает соперник. На каждый ход А соперника нужно отвечать ходом M+1-A.

- program SimpleGame;
- var c, m : integer; { исходная сумма, максимальный ход}
- r, { текущий остаток }move integer; { ХОД } m1: integer; { m + 1 }
- begin
-   write('Введите сумму и максимальный ХОД (integer»');
-   readln(c, m);
-   r := c; m1 := m+1; { инициализация }
- writeln('МаКс. ХОД: ‘, m, 'остаток=’, r);
- if r mod m1 <> 0
- then begin
-             move := r mod ml;
-             dec (r, move);
-             writeln('Мой ХОД>', move);
-             writeln('МаКс. ХОД: ‘, m, 'остаток=’, r);
-         end;
- while r > 0 do begin
-     write('Ваш ХОД>,); readln(move);
-     dec(r, move);
-     writeln('МаКс. ХОД: ‘, m, 'остаток=’, r);
-     move := m1 - move;
-     dec (r, move);
-      writeln('my move>', move);
-      writeln('МаКс. ХОД: ‘, m, 'остаток=’, r);
- end;
- writeln('Простите, вы проиграли. ');
- end.

Выигрышная позиция это позиция, начиная с которой можно, играя правильно, гарантированно выиграть при любой игре соперника. Проигрышная позиция, начиная с которой выиграть невозможно (если соперник не допустит ошибки). Иными словами, в выигрышной позиции либо игрок уже выиграл, либо хотя бы один ход приводит в проигрышную позицию, обеспечивая выигрыш при любой игре соперника. В проигрышной же позиции либо игра уже проиграна,

либо нет ни одного хода, обеспечивающего выигрыш (при правильной игре соперника), Т.е. любой ХОД приводит В выигрышную позицию. (Наконец, рассмотрим общую схему, которую может иметь программа или подпрограмма для любой антагонистической игры с известными выигрышными и проигрышными позициями.

```
создать начальную позицию;
отобразить позицию;
if позиция выигрышная then begin
      найти и выполнить свой ход;
      отобразить ход;
      отобразить позицию;
end;
while позиция незаключительная do begin
   получить ход от игрока;
   выполнить ход игрока;
   отобразить позицию;
   найти и выполнить свой ход;
   отобразить свой ход;
   отобразить позицию;
end;
отобразить сообщение о своей победе.
```

Золотое сечение

- Задача. Есть две кучки спичек. Два игрока берут из них спички ПО очереди. За один ход игрок берет из наименьшей кучки ненулевое число спичек, кратное числу спичек в другой кучке. Выигрывает тот, который взял последнюю спичку в одной из кучек. Программа должна реализовывать выигрышную стратегию, в частности, решать, кто должен ходить первым.
- Пример. Если в кучках 1 и 3 спички, выигрывает первый игрок, забрав 3 спички. Если в кучках 2 и 3 спички, то первый игрок может взять только 2 спички из второй кучки, после чего в позиции (2,1) второй игрок забирает 2 спички и выигрывает..
- Вход u выход. На клавиатуре задаются два положительных целых чисел (типа integer), обозначающих количества спичек в кучках. Ход задается числом, кратным меньшему из чисел в текущей позиции. После каждого хода выводится полученная позиция (два числа).
- Анализ задачи. Поищем способ определения, является ли заданная позиция выигрышной. Пусть в позиции (a, b) первое число не меньше второго. Если b = 0, то позиция является проигрышной, иначе, если a кратно b - выигрышной.
- Пусть a не кратно b, т. е. a = kb+r, rде r = a mod b > 0. Из позиции (a, Ь) можно перейти в позиции (r, b), (b+r, b), (2b+r, b), ..., (k-1)b+r, b).
- По определению, позиция (a, b) является выигрышной тогда и только тогда, Когда хотя бы одна из этих позиций проигрышная. Но это можно определить рекурсивно! Поскольку во всех этих позициях первое число строго меньше a, рекурсия обязательно достигнет "дна", на котором одно из чисел равно 0 или одно из них кратно другому. Однако такая рекурсия очень неэффективна из за многократных рекурсивных вызовов. Можно оптимизировать время работы за счет памяти, используя табличную технику. Но для этой игры есть способ получше. Докажем, что позиции (2b+r, b), ..., ((k+1)b+r, b) обязательно выигрышные, т.е. их можно не анализировать.

- Предположим, что одна из указанных позиций, скажем, (l*b+r,b), где 2<=l<k и r<b, проигрышная. Но из проигрышной позиции любой ход ведет в выигрышную позицию. Значит, и (r, b), и (b+r, b) являются выигрышными. Но тогда из выигрышной позиции (b+r, b) единственный допустимый ход b приводит в выигрышную же позицию (r, b), что невозможно по определению выигрышной позиции. Полученное противоречие доказывает, что позиция вида (l* b+r, b) при l>=2 является выигрышной. Отсюда следует: при a/b > 2 позиция (a, b) - выигрышная. Кроме того, из двух позиций (r, b) и (b+r, b) одна, и только одна, является выигрышной.
- Итак, если Ь<a<2b (неравенства строгие), то достаточно только рекурсивно выяснить, является ли позиция (r, b), где r=a mod b>0, проигрышной. Если является, то позиция (a, b) выигрышная, иначе проигрышная.
- Нужно еще правильно выбрать ход при a/ b>=2. Если позиция (r, b) проигрышная, то в нее ведет ход a- a mod b, иначе проигрышной обязательно будет позиция (b+r, b), в которую ведет ход a - a mod b - b.
- Эти соображения избавляют от повторных рекурсивных вызовов и существенно сокращают работу.

Решение задачи

- Оформим решение задачи с помощью трех подпрограмм. Процедура makeMove реализует собственно изъятие спичек из большей кучки. функция detWin получает при вызове количества а и Ь спичек в кучках, возвращает признак того, что данная ПОЗИЦИЯ выигрышная, и в параметре переменной сохраняет выигрышный ход (или наименьший возможный ход, если позиция проигрышная). Процедура game получает исходную позицию и вызывает функцию detWin, чтобы определить, является ли эта позиция выигрышной. Если это так, она делает первый ход. Далее в цикле соперник и программа делают по одному ходу. Поскольку соперник все время загоняется в проигрышную позицию, в которой у него есть только один допустимый ход, дальнейшие вызовы detWin не нужны.

- var a, b : integer; { исходные числа }
- procedure makeMove(var a, b, c : integer);
- Begin
- if a > Ь then dec(a, c) else dec(b, c);
- End;

- function detWin(a, b: integer; var c: integer) boolean;
- var t: integer;
- Begin
- if a < Ь then begin t := a; a := b; b := t end;
- { гарантированно a >= Ь }
- if b = 0 then begin
- detWin := false; c := 0
- end
- else if (a mod b = 0) then begin detWin := true; c := a end
- else if (a div b >= 2) then begin detWin:=true;
- if not detWin(b, a mod Ь , c)
- then c := a a mod b else c := a a mod b - b
- end
- else if not detWin(b, a mod b , c)
- then begin
- c := a - a mod b; detWin:=true
- end
- else begin
- c := b; detWin:=false;
- end;
- End;

- procedure game(a, Ь integer);
- var c : integer;
- Begin
- writeln('Позиция: ‘, a,’ ‘, b);
- if detWin(a, Ь, c) then begin
- makeMove(a, Ь, c);
- writeln (' Мой ход>', c);
- writeln ('Позиция: ‘, a, ‘ ‘, b) ;
- end;
- while (a <> 0) and (Ь <> 0) do begin
- write('Ваш ход>,); readln(c); { Здесь нужно добавить проверку правильности хода }
- makeMove(a, b, c);
- writeln ('Позиция: ‘, a, ‘ ‘, b) ;
- if detWin(a, Ь, c)
- then begin
- makeMove(a, b, c);
- writeln (' Мой ход>', c);
- writeln ('Позиция: ‘, a, ‘ ‘, b) ;
- end
- end;
- writeln(‘Простите, вы проиграли.');
- End;
- Begin
- readln(a, b); game(a, b);
- End.

- Задача. На столе лежат несколько кучек камешков. Два игрока берут из них камешки по очереди. За один ход игрок выбирает любую кучку и берет из нее любое ненулевое число камешков. Выигрывает тот, кто взял самый последний камешек. Программа должна реализовывать выигрышную стратегию, в частности, решать, кто должен ходить первым.
- Пример. Если в кучках 1 и 3 камешка, выиграет первый игрок. Он берет 2 камешка из второй кучки и оставляет позицию (1,1). Второй игрок может взять из любой кучки только 1 камешек, после чего второй заберет последний камешек из другой и выиграет. Если в кучках по 2 камешка, то выиграет второй игрок. Если первый возьмет 2 камешка из какой-нибудь КУЧКИ, то второй заберет 2 из другой и выиграет. Если же первый возьмет 1 камешек из какой-нибудь кучки, то второй - 1 камешек из другой КУЧКИ, а с позицией (1, 1) уже все ясно.
- Вход и выход: с клавиатуры вводится количество кучек n, 1<= n<= 10, затем n положительных чисел (типа integer) количества камешков в кучках. Ход задается номером кучки и количеством забираемых камешков. После каждого хода выводится полученная позиция (п чисел).
- Анализ задачи. Анализ позиций с двумя равными кучками прост: сколько бы камешков ни брал первый игрок из какой-либо КУЧКИ, второй будет брать столько же из другой кучки и в конце концов выиграет. Если же кучки не равны, то цель ясна своим ходом сделать их равными. В этой ситуации выигрывает первый игрок первым ходом он заберет из большей кучки разницу количеств камешков и оставит второму игроку две равные кучки. В общем случае решение связано с двоичным представлением чисел. Заметим: если числа равны, то их двоичные представления одинаковы, иначе они отличаются хотя бы в одном разряде. Поэтому поразрядная сумма по модулю 2 ("исключающее или") двух равных чисел имеет во всех разрядах 0, т. е. равна 0, а разных чисел - отлична от 0.
- Результат поразрядного сложения по модулю 2 для краткости будем называть НИМ-СУММОЙ, а само это сложение ним-сложением. В языке Turbo Pasca1 оно реализовано операцией xor: 10 xor 5 = 15, 5 xor 6 = 3. Итак, позиция с двумя числами a1 и a2 является выигрышной тогда и только тогда, когда a1 xor a2 <>0. НО если чисел не два, а больше, то этот критерий остается в силе! Позиция с числами a1, a2, ..., an является выигрышной тогда и только тогда, КОгда a1 xor a2 xor ... xor an<>0.

Рассмотрим примеры. Позиция (1,2, 3) проигрышная двоичные представления чисел 01, 10, 11 дают в каждом разряде четное число единиц, из-за чего их ним-сумма равна 0. Позиция (5, 2, 4) - выигрышная эти числа с двоичными представлениями 101, 010 , 100 дают ним-сумму 3 с представлением 011.

Итак, чтобы определить, является ли позиция выигрышной, достаточно вычислить ним-сумму чисел в позиции и убедиться, что она не равна 0. НО нужно еще определить, из какой кучки и сколько камешков взять, чтобы получилось проигрышная позиция.

Вначале найдем, сколько камешков нужно взять. Если выбрать одно из чисел, скажем, a1 и ним-прибавить к нему ним-сумму всех чисел S = a1 xor a2 xor … xor an, то ним-сумма (a1 xor S) xor a2 xor .,. xor an, очевидно, равна 0. Значит, чтобы новая ним-сумма стала равной 0, из a1 камешков должно остаться a1 xor S. Например, в позиции (5,2,4) с ним-суммой 3 выберем число 2; 2 xor 3 = 1 - это количество камешков, которые должны остаться во второй кучке. Значит, возьмем из нее 2 -1=1 камешек и получим позицию (5, 1, 4) с ним-суммой 0. Но как выбрать число для ним-сложения с S? Вообще говоря, ним-прибавление S может увеличить число, например 4 xor 3 = 7 или 5 xor 3 =6. Однако число должно уменьшиться ведь камешки нужно не добавлять к кучке, а брать из нее! Но оказывается, что среди чисел a1, a2 , ..., an обязательно найдется хотя бы одно, которое при ним-прибавлении S уменьшится. Убедимся в этом.

Заметим: если двоичные представления двух ним-слагаемых имеют 1 в одном и том же разряде, то в их ним-сумме этот разряд равен 0. Например, 2 xor 3 = 1 (2, 3 и 1 имеют двоичные представления 10,11 и 01 соответственно).

Найдем разряд со старшей единицей ним-суммы S. Тогда, по построению S, среди a1, a2 , ..., an. есть нечетное количество чисел, у которых в этом разряде 1. Обозначим любое из НИХ через x. В результате ним-прибавления S к x в указанном разряде будет 0, старшие разряды x, если есть, останутся без изменений, а возможные изменения в младших разрядах гарантированно "не перекроют" уменьшения в указанном разряде. Следовательно, ним-сумма x и S окажется меньше x.

Итак, искомым может быть любое из чисел, имеющих 1 в том разряде, в котором находится старшая 1 ним-суммы S. Чтобы найти его, переберем числа a1, a2 , ..., an пока не найдем ai для которого
ai xor S < ai .

```
function detWin(var num : aInt; n: byte;
            var nHeap : byte; var decrem: integer) : boolean;
var sum, numCopy : integer; k : byte;
begin
   Sum := 0;
   for k := 1 to n do
          sum:=sum xor num[k] ;
   detwin := (sum <> 0);
   k := 1;
   while num[k] xor Sum >= num[k] do
       inc(k);
  nHeap ;= k;
  numCopy := num[nHeap] ;
  num [nHeap] : = num [nHeap] xor Sum;
  decrem := numCopy -  num[nHeap];
end;
```

- Задача. Касса содержит C копеек (C>= 3). Два игрока по очереди берут из кассы по целому числу копеек. За ход можно взять не меньше одной копейки и не больше удвоенного числа копеек, взятых соперником на предыдущем ходу. Первый ХОД одна или две копейки. Выигрывает ТОТ, кто своим ходом опустошит кассу. Программа должна читать C (от 1 до 300), выбирать право первого хода и выигрывать.
- Пример. При C= 3 выигрывает второй игрок - если первый взял 1 копейку, второй берет 2, если первый взял 2 копейки, второй берет 1. При C=4 выиграет первый игрок взяв 1 копейку, он оставит в кассе 3 копейки перед ходом второго игрока.
- Анализ задачи.
- Ясно, что позиция в данной задаче это не просто остаток в кассе, а пара (остаток, последний ход). Обозначим эти величины соответственно r и m.
- Позиция (r, m) является выигрышной, если существует ход k (от 1 до 2m) в проигрышную позицию (r-k, k), и является проигрышной, если при любом k от 1 до 2m позиция (r-k, k) выигрышная.. Это рекурсивное определение имеет "дно": позиции (1, k) и (2, k) при любом возможном k ЯВЛЯЮТСЯ выигрышными, (0, k) - проигрышными.
- Чтобы решить, является ли текущая позиция выигрышной или проигрышной, нетрудно реализовать это определение "в лоб" с помощью рекурсивной функции.
- Однако из-за кратных рекурсивных вызовов такое решение неэффективно. Рекурсия с запоминанием может улучшать его, но проще один раз заполнить таблицу вариантов ходов и использовать ее во время игры.
- Используем таблицу tab, строки которой индексированы остатками, столбцы последними ходами. Значением элемента, соответствующего выигрышной позиции, является один из возможных ходов, которые ведут к выигрышу. Если позиция проигрышная, значением элемента должно быть ТО, что не может быть ходом положим его равным 0. Тогда 0 будет признаком проигрышной позиции, а положительное значение - признаком выигрышной.

- Построение таблицы. Очевидно, все элементы первой строки должны иметь значение 1, второй 2. Заполним остальные строки, инициализировав все их элементы значением 0. Пусть теперь n>=3 и m>=. Если n<=2m, то в позиции возможен ход n, опустошающий кассу, поэтому tab [n, m] = n. При n > 2m переберем все возможные ходы k от 1 до 2m и используем элементы предыдущих строк таблицы. Если tab [n-k, k] = 0 хотя бы для одного значения k, значит, ход k приводит В проигрышную позицию (n-k, k); тогда tab [n, m] = k. При этом естественно выбрать максимальное значение k, ускоряющее игру. Если же все tab[n-k, k]<>0 при всех k от 1 до 2m tab [n, m] представляет проигрышную позицию и остается равным 0.
- Выбор хода с помощью таблицы. Предположим, текущей является ПОЗИЦИЯ (r,m). Если r<=2m, то победный ход r является допустимым, и для его определения таблица вообще не нужна. При r>2m, если tab [r, m]>0, ходом будет tab [r, m], иначе - 1 (такой ход в действительности программа никогда не выполняет). Если оформить выбор хода в виде функции, возвращающей булев признак выигрышности позиции, то в первых двух ситуациях возвращается true, в третьей false.
- Ширина таблицы. Остается уточнить, какой должна быть ширина таблицы, т. е. каким может быть максимальный ход. Заметим: если последний ход равен т, ТО все выполненные ходы уменьшили исходную сумму С не . менее чем на 2m-1 копеек (касса уменьшается в точности на 2m-1, если первый ход равен 1 и каждый следующий ход вдвое больше предыдущего). Отсюда, если достигнута позиция (r,m), то r<= C-2m+1l. Но при r<=2m для определения следующего хода (одного из чисел от 1 до 2m) таблица не нужна. Но таблица нужна, поэтому r>2m. Тогда 2m<C-2m+1, или 2m<(C+1)/2. Итак, достаточно таблицы шириной (C+1)/2.

Оценивание позиций: максимальная сумма

- Задача 14.5. N золотых слитков с различными ценами разложены в ряд. Два игрока по очереди берут несколько слитков в левом конце ряда. Первый ход один или два слитка, а далее за каждый ход можно взять не меньше одного слитка и не больше удвоенного числа слитков, взятых перед этим соперником. Цель каждого игрока получить как можно большую сумму цен взятых слитков. Программа должна вычислять, какие суммы могут обеспечить себе первый и второй игроки, как бы ни играл их соперник. Затем она должна играть за игрока, максимальная сумма которого больше, и набирать слитки с суммой не меньше этой максимальной.
- Вход. В строке текста записано число N (1<=N<=200), затем N цен слитков от
- 1 до 200 в порядке их расположения в ряду. Числа разделены пробелами.
- Примеры. Вход: 4 1 2 4 8. Первый игрок не позволит набрать второму сумму больше 6, если сделает первый ход 1. Тогда второй игрок возьмет два слитка ценой 2 и 4. Первый игрок заберет последний слиток и получит сумму 9. Если первый игрок сделает первый (ошибочный!) ход 1 2, то второй заберет слитки 4 8. Итак, играя правильно, первый набирает не меньше 9, второй не меньше 6, поэтому программа должна ходить первой и набирать сумму не меньше 9.
- Вход: 4 1 1 1 1. Каждый игрок может набрать не меньше 2, поэтому не важно, за какого игрока будет играть программа.
- Анализ задачи. В данной задаче главной является сумма, которую можно набрать, исходя из позиции. Сумма зависит как от оставшихся слитков, так и от последнего хода, поэтому под позицией будем понимать пару (r, m), где r номер первого из оставшихся слитков, m - последний ход. Максимум суммы, которую можно набрать, начиная с позиции (r, m), обозначим через S(r, m), а сумму цен всех слитков, начиная с r-го - T(r). Оба игрока стремятся увеличить свою сумму, поэтому игрок получит все, что не сможет забрать его соперник. Значит, ХОДИТЬ нужно так, чтобы соперник Мог набрать как можно меньше. Иными словами, первый ход k, где 1<=k<=2m, приводящий к позиции (r+k, k), нужно выбрать так, чтобы минимизировать максимум суммы, набираемой начиная с позиции (r+k, k).
- S(r, m) = T(r) – min(S(r+k, k))
- для 1<=k<=2m
- а ХОД - это значение k, при котором достигается min S(r+k, k) для всех к от 1 до 2m

- Как видим, максимальная сумма S(r, т) рекурсивно выражается с помощью максимальной суммы с большим аргументом r+k. На "дне" рекурсии находятся ситуации, в КОТОРЫХ игрок может взять все оставшиеся слитки. Естественно, он должен их взять, иначе часть возьмет соперник и уменьшит сумму игрока, поэтому S(r,m)=T(r) при 2m >=N+1-r.
- Первым ходом первого игрока может быть 1 или 2, поэтому можно считать, что он начинает в позиции (1, 1). Вначале нужно определить, кто делает первый ход, для этого нужно знать S(1,1) и T(1). Если S(1, 1)>= T(1)/2, программа будет ходить первой, иначе отдаст это право сопернику.
- Решим задачу, не используя рекурсию. Значения T(r) для r = 1,2, ..., N вычислим сразу и запомним в массиве T, поскольку они понадобятся в дальнейшем. для вычисления
S(1, 1) используем таблицу s размерами N x L, где L ширина таблицы, которую определим ниже. Заполним таблицу построчно, начав с последней строки все значения в ней равны цене последнего слитка T (N). Затем в очередной строке r для каждого возможного значения m, если 2m<N+1-r, то S(r, m) вычисляется по формуле
S(r, m) = T(r) – min(S(r+k, k)), иначе S(r, m)=T(r).
- Определив S(1, 1) и право первого хода, программа должна играть, т. е. для каждой позиции (r, m) находить значение k, при котором достигается минимум S(r+k,k) по k от 1 до 2m. для решения этой задачи достаточно построенной таблицы S.

Длинная арифметика

- Для того чтобы рассматривать арифметику требуется принять некоторые соглашения относительно представления положительных и отрицательных чисел. Можно конечно хранить для каждого числа знак, и соответственно обрабатывать его при каждой операции, но это приводит к необоснованному усложнению алгоритмов. Гораздо более изящным приемом будет использование K-дополненного кода, по аналогии с тем, как это делается для двоичных чисел. Вы наверно уже отметили, что наши числа почему-то имеют размер N + 1 разрядов, так вот этот лишний, самый старший, разряд мы потратим на хранение «знака» числа.
- Теперь немного о том, как работает наше K-дополнение. Обратим внимание, что с помощью нашего длинного числа можно представлять числа из диапазона $0 \ldots K^{N+1}$. Пожертвуем одним старшим разрядом (уже отмечалось, что он изначально предназначался для хранения знака) и будем представлять числа из диапазона $-K^N \ldots +K^N$. Таким образом, все арифметические операции окажутся представленными по модулю K^N. Для того чтобы представить отрицательное число, добавим к нему K^{N+1} – согласно законам арифметики $K^{N+1} \bmod K^N = 0$, поэтому такая добавка не повлияет на результаты вычислений, и увеличенное число будет вести себя подобно исходному отрицательному. Очевидно, что для положительного числа такая добавка не нужна, и самый старший, знаковый, разряд будет нулевым, а вот для отрицательного числа этот знак будет в точности равен K–1 (вспомните заем при вычитании столбиком).

Смена знака числа (K-дополнение)

- Для лучшего понимания механизма K-дополнения зафиксируем K равным 10. Теперь рассмотрим процесс смены знака. Смена знака заключается для N-разрядного числа в вычитании этого числа из 10N+1. Воспользуемся маленькой хитростью. Заметим, что вычитание любого разряда из девятки не приводит к займу из более старшего разряда, а число 10N+1 легко представить как число из N + 1 девятки плюс 1: $10^{n+1}=\underbrace{9999\ldots9}_{N+1}+1$
- N+1
- Теперь, вместо того, чтобы реализовывать полноценное вычитание с займами (подобно вычитанию в столбик) можно просто каждый разряд числа (в том числе и знаковый) вычесть из девятки, а затем к результату прибавить единицу. Добавление единицы также процесс очень простой – нужно просто бежать с самого младшего разряда числа, и до тех пор, пока встречаются девятки заменять их нулями, а первый разряд не равный девятке увеличить на единицу.
- Теперь легко обобщить процесс на основание K и написать общий алгоритм. Роль девятки будет, очевидно, выполнять K–1.

Длинный inc()

Так как алгоритм увеличения числа на 1 имеет самостоятельную ценность, то сначала опишем его.

```
Procedure LongInc (A[])
begin
  I ← 0;
  {Заменяем все хвостовые значения K-1 на нули}
  while (I <= N) and (A[I] = K - 1) do begin
    A[I] ← 0;
    I ← I + 1;
  end;
  {Увеличиваем найденный разряд}
  A[I] ← A[I] + 1;
  return (A[]);
end
```

N+1	6	5	4	3	2	1	0
0	0	2	2	0	0	0	0

Заменяем все

Увеличиваем найденный разряд

Заметим, что решение затратить дополнительный разряд на хранение знака сильно упростило код – если бы лишнего разряда не было, то проверки пришлось бы усложнять, чтобы не допустить ситуации выхода за границу массива. Также заметим, что ситуация когда все разряды числа равны K–1 также обрабатывается «корректно» – переполнение приводит к смене знака числа, как и при машинных командах, что легко отследить.

Смена знака Neg()

```
Procedure LongNeg'(A[])
begin
  {Вычитаем каждый разряд из девятки}
  for I ← 0 to N do
    A[I] ← (K – 1) - A[I];
  {Теперь увеличиваем число на единицу}
  LongInc(A[]);
  return (A[]);
end
```

Если рассмотреть данный алгоритм подробнее, то можно заметить, что в худшем случае каждый элемент массива обрабатывается дважды. Для повышения эффективности алгоритма желательно избавиться от избыточного прохода. Заметим, что все нулевые завершающие разряды в результате двойного прохода не изменяются, а вот первый не нулевой (если считать от младшего) вычитается как бы не из K–1 а из K. Теперь можно написать более эффективный вариант смены знака у числа.

```
Procedure LongNeg (A[])
begin
  I ← 0;
  {Пропускаем нулевые разряды}
  while (I <= N) and (A[I] = 0) do
    I ← I + 1;
  {Вычитаем из K первый не нулевой разряд,
   учитывая что разряд может быть нулевым}
  A[I] ← (K - A[I]) mod K;
  I ← I + 1;
  {Оставшиеся разряды дополняем вычитанием из K-1}
  while I < N + 1 do
    A[I] ← K – 1 – A[I];
  return (A[]);
end
```

Заметим, что даже в ситуации, когда все не знаковые разряды равны нулю, алгоритм работает корректно, так как такое число может означать только нуль, а нуль не имеет знака и его знаковый разряд тоже должен быть равен нулю. В таком случае алгоритм ничего не меняет, а оставляет число нулем.

Длинное сложение

	N+1	6	5	4	3	2	1	0
A	0	0	0	1	2	9	7	1
B	0	0	1	9	5	0	8	9
A'	0	0	2	0	8	0	6	0

Carry-> 0

Реализация сложения в точности совпадает со школьным алгоритмом сложения в столбик. В нем только более четко прописан процесс переноса.

```
Procedure LongAdd (A[], B[])
begin
  {Carry сохраняет значение переноса, может быть 0 или 1}
  Carry ← 0;
  for I ← 0 to N do begin
    {Складываем разряды I и перенос от сложения I-1-ых}
    Temp ← A[I] + B[I] + Carry;
    {Берем от результата только один разряд}
    A[I] ← Temp mod K;
    {Temp всегда меньше 2K, и перенос можно вычисли
     простым целочисленным делением}
    Carry ← Temp div K;
  end;
  return (A);
end
```

Начинаем с «конца» числа (младших разрядов), складываем две цифры и перенос из предыдущего разряда

Добавление одной цифры к числу

В некоторых случаях требуется добавить к длинному числу один разряд (например, при считывании текстовой записи числа). Такой алгоритм несколько отличается от исходного, за счет хитрого использования начального значения переноса:

```
Procedure LongAddDigit (A[], D)
begin
  {Начальное значение рассматриваем как перенос}
  Carry ← D;   I ← 0;
  {Проверяем, не достигли ли мы нулевого переноса, так как по достижении Carry = 0, алгоритм ничего не изменяет в более старших разрядах}
  while (I <= N) and (Carry > 0) do begin
    {Добавляем перенос к очередному разряду}
    Temp ← A[I] + Carry;
    A[I] ← Temp mod K;
    {В действительности, операция div позволяет нам корректно
     обрабатывать ситуацию, даже если D > K (!)}
    Carry ← Temp div K;
    I ← I + 1;
  end;
  return (A[]);
end
```

Длинное вычитание

Длинное вычитание может быть реализовано либо с помощью обращения знака и последующего сложения:

```
Procedure LongSub’ (A[], B[])
begin
  C[] ← LongNeg(B[]);
  D[] ← LongAdd(A[], C[]);
  return (D[]);
end
```

Теперь, если подсчитать количество операций, в любом случае потребует не менее 2N операций – N на смену знака числа, и N на сложение.

Легко убедиться, что гораздо эффективнее выглядит реализация школьного вычитания в столбик:

```
Procedure LongSub (A[], B[])
begin
  {Carry будет хранить заем, и может принимать значения 0 и -1}
  Carry ← 0;
  for I ← 0 to N do begin
    Temp ← A[I] – B[I] + Carry;
    {Значения разряда всегда должно быть положительно,
     если оно отрицательно, то формируем заем из более старшего}
    A[I] ← (K + Temp) mod K;
    {Предполагаем, что целочисленное вычитание не изменяет
     знак числа, и обеспечиваем возникновение десятка}
    Carry ← (Temp – K + 1) div K;
  end;
  return (A[]);
end
```

Данный алгоритм совершит ровно N итераций цикла, и, следовательно, является более эффективным.

Умножение числа на один разряд

Легко заметить, что умножение числа на один разряд может быть проведено по той же схеме, что и предыдущие алгоритмы сложения и вычитания – действительно, перемножение двух K-ричных разрядов дает максимальное число, равное

$(K - 1) * (K - 1) = K^2 - 2 * K + 1 < K^2$,

следовательно, оно содержит не больше двух разрядов. Таким образом, перенос не превосходит K, и можно сформулировать следующий алгоритм:

```
Procedure LongMulDigit (A[], B)
begin
  Carry ← 0;
  for I ← 0 to N do begin
    {Производим обработку для очередного разряда}
    Temp ← A[I] * B + Carry;
    A[I] ← Temp mod K;
    Carry ← Temp div K;
  end;
  return (A[]);
End
```

- Заметим, что случай отрицательных чисел не требует дополнительного рассмотрения, так как операция умножения на положительный разряд не изменяет знака числа – поэтому «знаковый» разряд оставляется алгоритмом без изменения, если не случается переполнения. С другой стороны, изменение знакового разряда однозначно сигнализирует о возникшем переполнении.

Длинное сложение со сдвигом

Теперь, сформулировав алгоритм для умножения числа на один разряд, мы можем легко реализовать школьный алгоритм умножения в столбик. Так как требуется сложение специального вида (со сдвигом), то целесообразно сформулировать такой алгоритм.

```
Procedure LongAddShift (*A[], B[], Shift)
begin
  {Carry сохраняет значение переноса, может быть 0 или 1}
  Carry ← 0;
  for I ← Shift to N do begin
    {Складываем разряды I и перенос от сложения I-1-ых}
    Temp ← A*[I] + B[I] + Carry;
    A*[I] ← Temp mod K;
    Carry ← Temp div K;
  end;
end
```

Данный алгоритм отличается в двух пунктах: суммирование начинается только с разряда с номером Shift (это соответствует сдвигу второго слагаемого на Shift разрядов вправо), массив A[] передается по ссылке – это необходимо для того, чтобы избежать лишнего присваивания, которое для длинных чисел имеет сложность O(N).

Длинное умножение чисел

- Теперь можно сформулировать полный алгоритм:
- Procedure LongMul (A[], B[])
- begin
- {В качестве начального значения берем длинный нуль}
- C[] ← [0];
- for I ← 0 to N do begin
- {Умножаем на следующий разряд}
- Temp[] ← LongMulDigit(A[], B[I], K);
- {Переполнение возникает, если Temp[N] <> Temp[N - I]}
- {Добавляем очередное слагаемое (со сдвигом I)}
- LongAddShift(C[], Temp[], I);
- end;
- return (C[]);
- end

Длинное сравнение

Длинное сравнение чисел легко реализовать на основе вычитания, надо только договориться, как возвращать результат. В основном, при сравнении, нас интересуют три случая: равны ли числа, первое число больше второго, первое число меньше второго. Будем обозначать эти исходы целыми числами 0, +1 и –1 соответственно.

```
Procedure LongCompare (A[], B[])
begin
 {Вычисляем A - B}
  Temp[] ← LongSub(A[], B[]);
  {Проверяем знак – если число отрицательное – возвращаем -1}
  if Temp[N] <> 0
  then return (-1)
  else begin
    {Проверяем равенство нулю}
    for I ← 0 to N – 1 do
      if Temp[I] <> 0
      then return(1);
    return (0);
  end;
End
```

- Легко заметить, что сложность данного алгоритма в худшем случае равна сложности вычитания плюс O(N). Вычитание включает в себя цикл, который, в принципе, легко совместить с циклом проверки результата на нуль.

- Таким образом, улучшенный алгоритм будет выглядеть так:
- Procedure LongCompare (A[], B[])
- begin
-   {Carry будет хранить заем, и может принимать значения 0 и -1}
-   Carry ← 0;
-   {Это флажок, который хранит TRUE, если результат нулевой}
-   Zero ← TRUE;
-   for I ← 0 to N do begin
-     Temp ← A[I] – B[I] + Carry;
-     {Проверяем, что результат вычитания - нуль}
-     Zero ← Zero and ((Temp mod K) = 0);
-     {Предполагаем, что целочисленное деление не изменяет
-      знак числа, и обеспечиваем возникновение десятка}
-     Carry ← (Temp – K + 1) div K;
-   end;
-   {Теперь проверяем перенос – если он есть, то однозначно A < B,
-    если нет, то проверяем Zero}
-   if Carry <> 0
-   then return (-1)
-   else if Zero
-     then return(0)
-     else return(1);
- end
- Данный алгоритм прокручивает цикл один раз, и поэтому эффективней предыдущего.

Длинное деление

- Для того, чтобы реализовать длинное деление, вспомним определение деления: под делением числа a на число b понимается нахождение такой пары чисел r и q, что: $a = b * q + r, 0 \le r < b$.
- Рассмотрим теперь деление числа на 2 * b:
- $a = 2 * b * q + r, 0 \le r < 2 * b$.
- оно распадается на две альтернативы:
- $0 \le r < b$, $a = b * (2 * q) + r$;
- $b \le r < 2 * b$, $a = b * (2 * q + 1) + (r – b)$.
- Отсюда немедленно следует рекурсивный алгоритм длинного деления, который в качестве примитивов использует только сложение, вычитание, а также умножение и деление на два:

- Procedure LongDivRec (A[], B[])
- begin
-   {Если A < B, то выход из рекурсии}
-   if LongCompare(A[], B[]) = -1
-   then return (<[0], A[]>)
-   else begin
-     {Получим результат деления A на 2B}
-     <Q[], R[]> ← LongDivRec(A[], LongMulDigit(B[], 2));
-     {Так как в первом и втором случае фигурирует 2Q
-      сразу получим его}
-     Q[] ← LongMulDigit(Q[], 2);
-     {Получим R - B для второго случая}
-     Temp[] ← LongSub(R[], B[]);
-     {Проверим знак R – B, если разность отрицательная, то первый
-      вариант, иначе - второй}
-     if Temp[N] <> 0
-     then return(<Q[], R[]>)
-     else return(<LongInc(Q[]), Temp[]>)
-   end;
- end

```
Procedure LongDivTable (A[], B[])
begin
  {Сначала добиваемся A < B – уход в глубину рекурсии}
  I ← 0;   TableB[0] ← B;
  while LongCompare(A[], TableB[I]) ≥ 0 do begin
    TableB[I + 1] ← LongMulDigit(TableB[I], 2);
    I ← I + 1;
  end;
  {Теперь возврат}
  Q[] ← [0];   R[] ← A[];
  while I > 0 do begin
    {Теже действия, что и при рекурсивном алгоритме}
    Q[] ← LongMulDigit(Q[], 2);
    Temp[] ← LongSub(R[], TableB[I - 1]);
    if Temp[N] = 0
    then begin
      Q[] ← LongInc(Q[]);
      R[] ← Temp[];
    end;
    I ← I – 1;
  end;
  return (<Q[], R[]>);
end
```

Более компактное представление длинных чисел

- 30!= 265252859812191058636308480000000
- Представим в виде:
- $30!=2*(10^4)^9+6525*(10^4)^8+2859*(10^4)^7+8121*(10^4)^6+9105*(10^4)^5+8636*(10^4)^3+3084*(10^4)^2+8000*(10^4)^1+0000*(10^4)^0$
- Это представление наталкивает на мысль о массиве

Номер элемента в массиве А	0	1	2	3	4	5	6	7	8	9
Значение	9	0	8000	3084	8636	9105	8121	2859	6525	2

- Возникают вопросы. Что за 9 в A[0], почему число хранится задом наперед»?
- Первая задача. Ввести число из файла. Но прежде описание данных.
- Const MaxDig=1000;
- {*Максимальное количество цифр — четырехзначных.*}
- Osn=10000;{Основание нашей системы счисления, в элементах массива храним четырехзначные числа.*}
- Type TLong=Array[0..MaxDig] Of Integer;
- {* Вычислите максимальное количество десятичных цифр в нашем числе.*}

- Прежде чем рассмотреть процедуру ввода, приведем пример. Пусть в файле записано число 23851674 и основанием (Osn) является 1000 (храним по три цифры в элементе массива A). Изменение значений элементов массива A в процессе ввода (посимвольного — переменная ch) отражено в таблице.

A[0]	A[1]	A[2]	A[3]	ch	Примечание
3	674	851	23	-	Конечное состояние
0	0	0	0	2	Начальное состояние
1	2	0	0	3	1-й шаг
1	23	0	0	8	2-й шаг
1	238	0	0	5	3-й шаг
2	385	2	0	1	4-й шаг
2	851	23	0	6	5-й шаг
2	516	238	0	7	6-й шаг
3	167	385	2	4	7-й шаг
3	674	851	23		

- Итак, в A[0] храним количество задействованных (ненулевых) элементов массива А — это уже очевидно. И при обработке каждой очередной цифры входного числа старшая цифра элемента массива с номером i становится младшей цифрой числа в элементе i+1, а вводимая цифра будет младшей цифрой числа из А[1].
- For i:=A[0] DownTo 1 Do Begin
-   A[i+1] :=A[i+l] + (LongInt (A[i]) *10) Div Osn;
-   A[i] : = (LongInt (A[i]) *10) Mod Osn;
- End;
- Пусть мы вводим число 23851674 и первые 6 цифр уже разместили «задом наперед» в массиве А. В символьную переменную ch считали очередную цифру многоразрядного числа — это «7». По нашему алгоритму эта цифра «7» должна быть размещена младшей цифрой в А[1]. Выписанный фрагмент программы освобождает место для этой цифры. В таблице отражены результаты работы этого фрагмента.

i	A[1]	A[2]	A[3]	ch
2	516	238	0	7
2	516	380	2	
1	160	385	2	

- Procedure ReadLong(Var A:TLong);
- Var ch:Char;i:Integer;
- Begin
- FillChar (A,SizeOf(A) ,0) ;
- Repeat
- Read (ch) ;
- Until ch In ['0'..'9'] {*Пропуск не цифр в начале файла. *}
- While ch In ['0'..'9'] Do Begin
- For i:=A[0] DownTo 1 Do Begin
- {*"Протаскивание« старшей цифры в числе из A[i] в младшую
- цифру числа из A[i+1]. *}
- A[i+1]:=A[i+l]+(LongInt(A[i])*10) Div Osn;
- A[i] := (LongInt (A[i]) *10) Mod Osn;
- End;
- A[l] :=A[1]+Ord(ch)-Ord('0');{*Добавляем младшую цифру к числу из A[1] . *}
- If A[A[0] +1]>0 Then Inc (A[0]);{ *Изменяем длину, число задействованных элементов массива А. *}
- Read (ch);
- End; {While}
- End;

Вывод многоразрядного числа

- Казалось бы, нет проблем — выводи число за числом. Однако в силу выбранного нами представления числа необходимо всегда помнить, что в каждом элементе массива хранится не последовательность цифр числа, а значение числа, записанного этими цифрами. Пусть в элементах массива хранятся четырехзначные числа. И есть число, например, 128400583274. При выводе нам необходимо вывести не 58, а 0058, иначе будет потеря цифр. Итак, нули также необходимо выводить. Процедура вывода имеет вид:

```
Procedure WriteLong(Const A:TLong) ;
Var ls,s:String; i: Integer;
Begin
    Str(Osn Div 10,ls);
    Write(A[A[0]]);{*Выводим старшие цифры числа. *}
    For i:=A[0]-1 DownTo 1 Do Begin
        Str(A[i],s) ;
        While Length (s)<Length (ls) Do s:='0'+s;
        {*'Дополняем незначащими нулями. *}
        Write (s) ;
     End;
  WriteLn;
End;
```

- Алгоритм имитирует привычное сложение столбиком, начиная с младших разрядов. И именно для простоты реализации арифметических операций над многоразрядными числами используется машинное представление «задом наперед». Ниже приведен текст процедуры сложения двух чисел.
- Procedure SumLongTwo(Const A,B:TLong;Var C:TLong) ;
- Var i,k:Integer;
- Begin
-   FillChar (C,SizeOf (C) , 0) ;
-   If A[0]>B[0] Then k:=A[0] Else k:=B[0];
-   For i:=1 To к Do Begin
-     C[i+1]:=(C[i]+A[i]+B[i]) Div Osn;
-     C[i]:=(C[i]+A[i]+B[i]) Mod Osn; {*Есть ли в этих опера торах ошибка?*}
-    End;
-   If C[k+1]=0 Then C[0]:=k Else C[0]:=k+1;
- End;

- Четвертая задача. Реализация операций сравнения чисел (A=B, A<B, A>B, A=<B, A>=B).
- Функция A=B имеет вид.
- Function Eq(Const A,B:TLong):Boolean;
- Var i:Integer;
- Begin
-   Eq:=False;
-   If A[0]=B[0] Then Begin
-     i : =1 ;
-     While (i<=A[0]) And (A[i]=B[i]) Do Inc(i);
-     Eq:=(i=A[0]+l) ;
-   End;
- End;
- Реализация функции A>B также прозрачна.
- Function More (A,B: TLong): Boolean;
- Var i:Integer;
- Begin
-   If A[0]<B[0] Then More:=False
-   Else If A[0]>B[0] Then More:=True
-     Else Begin
-       i:=A[0];
-       While (i>0) And (A[i]=B[i]) DoDec(i);
-       If i=0 Then More:=False
-       Else If A[i]>B[i] Then More:=True Else More:=False;
-     End;
- End;

- Остальные функции реализуются через функции Eq и More.
- Function Less(A,B:TLong):Boolean;{A<B}
- Begin
-     Less:=Not(More(A,B) Or Eq(A,B));
- End;
- Function More_Eq (A,B: TLong): Boolean;
- Begin
-     More_Eq:=More(A,B) Or Eq(A,B);
- End;
- И наконец, последняя функция A=<B.
- Function Less_Eq(A,B:TLong):Boolean;
- Begin
-    Less_Eq:=Not(More(A,B));
- End;

- Умножение многоразрядного числа на короткое. Под коротким понимается целое число, не превосходящее основание системы счисления. Процедура очень похожа на процедуру сложения двух чисел.
- Procedure Mul(Const A: TLong;Const К: LongInt;Var
- C: TLong);
- Var i: Integer;{*Результат - значение переменной C.*}
- Begin
-   FillChar(C, SizeOf(C), 0);
-   If K=0 Then Inc(C[0]){*Умножение на ноль.*}
-   Else Begin
-     For i:=1 To A[0] Do Begin
-       C[i+1]:=(LongInt (A[i])*K+C[i]) Div Osn;
-       C[i]:=(LongInt(A[i])*K+C[i]) Mod Osn;
-     End;
-    If C[A[0]+1]>0 Then C[0]:=A[0]+1
-    Else C[0]:=A[0];{*Определяем длину результата.*}
-   End; {else}
- End;

- Procedure MulLong(Const A,B: TLong; Var C: TLong);
- {*Умножение "длинного" на "длинное".*}
- Var i, j : Word;
-     dv: LongInt;
- Begin
-   FillChar(Cr SizeOf(C), 0) ;
-   For i:=1 To A[0] Do
-     For j:=1 To B[0] Do Begin
-       dv:=LongInt (A[i])*B[j]+C[i+j-1];
-       Inc (C[i+j], dv Div Osn);
-       C[i+j-1]:=dv Mod Osn;
-     End;
-   C[0] :=A[0]+B[0] ;
-   While (C[0]>1) And (C[C[0]]=0) Do Dec(C[0]);
- End;

- Вычитание двух многоразрядных чисел, с учетом сдвига. Если суть сдвига пока не понятна, то оставьте ее в покое, на самом деле вычитание с учетом сдвига потребуется при реализации операции деления. Выясните логику работы процедуры при нулевом сдвиге. Введем ограничение: число, из которого вычитают, больше числа, которое вычитается. Работать с длинными отрицательными числами мы не умеем. Процедура была бы похожа на процедуры сложения и умножения, если бы не одно «но» — не перенос единицы в старший разряд, а «заимствование единицы из старшего разряда». Например, в обычной системе счисления мы вычитаем 9 из 11 — идет заимствование 1 из разряда десятков, а если из 10000 той же 9 — процесс заимствования несколько сложнее.

```
• Procedure Sub (Var A: TLong;Const B: TLong; sp: Integer);
• Var i,j : Integer;{ *Из А вычитаем В с учетом сдвига sp, результат вычитания в А.*}
• Begin
• For i:=1 To B[0] Do Begin
•     Dec (A[i+sp], B[i]) ;{*Реализация сложного заимствования*} {*}
•     j:=i; {*}
•     While (A[j+sp]<0) And (j<=A[0]) Do Begin {*}
•              Inc(A[j+sp], Osn); Dec(A[j+sp+1] ); Inc(j) ; {*}
•      End; {*}
•  End;
•  i:=A[0];
•  While (i>1) And (A[i]=0) Do Dec (i) ;
•  A[0] :=i; (*Корректировка длины результата операции*)
• End;
```

Очередь с приоритетом

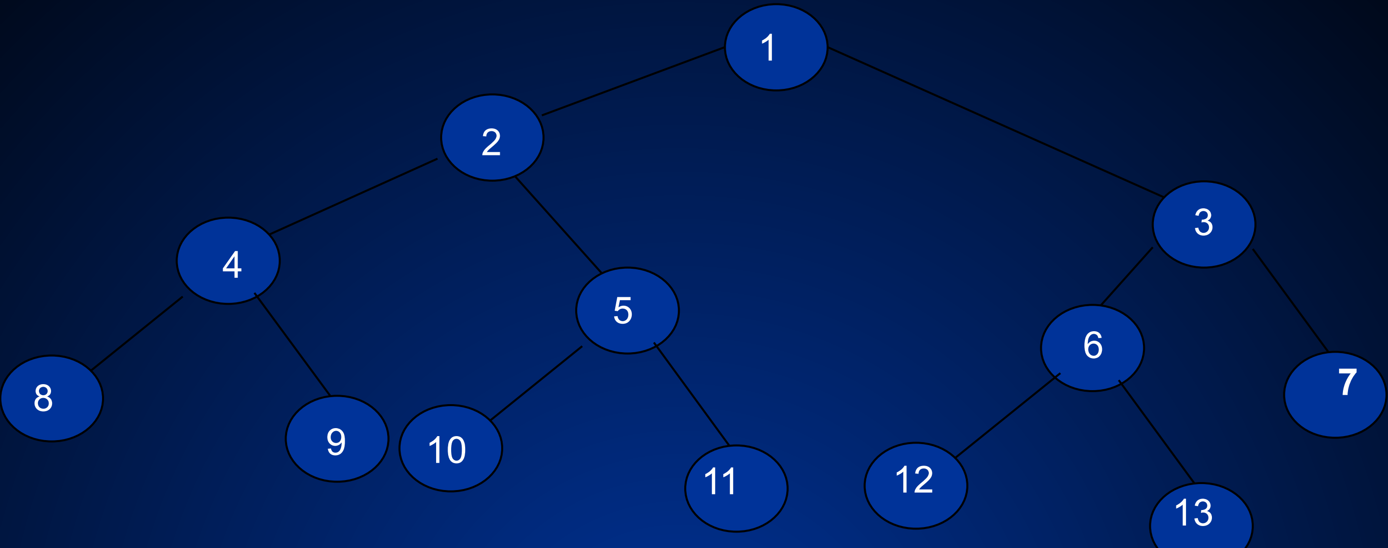
Иногда необходимо работать с динамически изменяющимся множеством объектов, среди которых часто нужно находить объект с минимальным ключом. В этом случае может пригодиться структура данных, называемая “приоритетной очередью”. Более точно, приоритетная очередь – это структура, хранящая набор объектов и поддерживающая следующие операции:

- **INSERT** (x) – добавляет в очередь новый объект x;
- **MINIMUM** – возвращает объект с минимальным значением ключа;
- **EXTRACT-MIN** – удаляет из очереди объект с минимальным значением ключа.

Бинарная куча

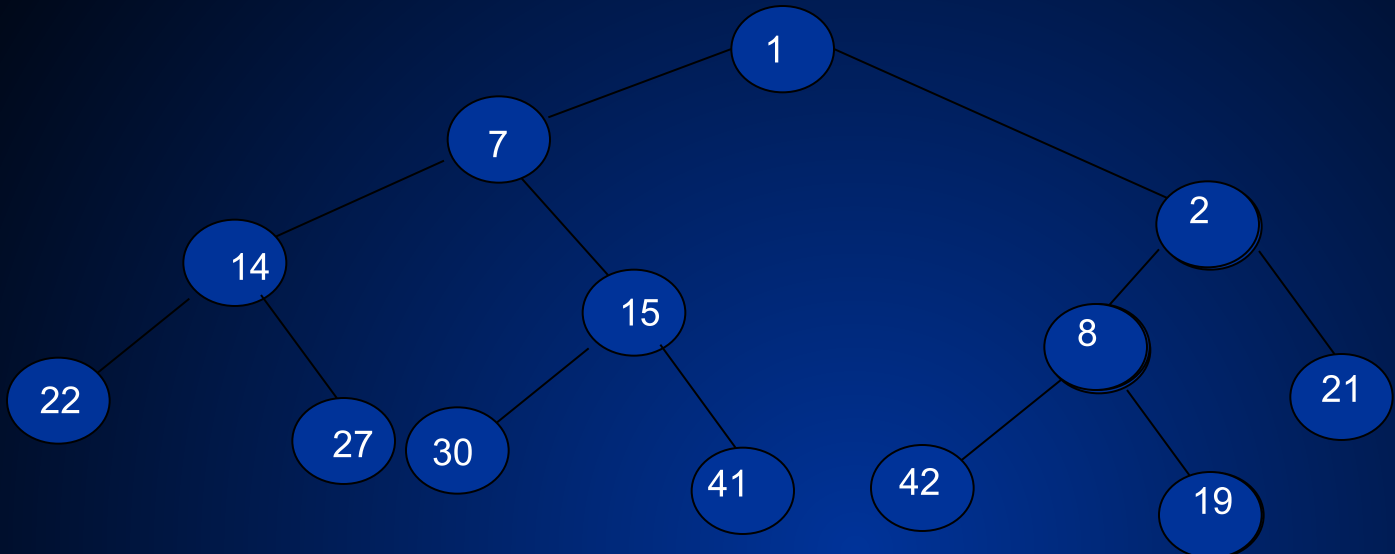
Будем считать, что объекты хранятся в вершинах полного двоичного дерева (самый нижний уровень дерева заполнен, возможно, не полностью).

Пронумеруем вершины этого дерева слева направо сверху вниз. Пусть N – количество вершин в дереве. Нетрудно видеть, что справедливы следующие свойства



- **Свойство 1.** Высота полного двоичного дерева из N вершин (то есть максимальное количество ребер на пути от корня к листьям) - O(log N).
- **Свойство 2.** Рассмотрим вершину полного двоичного дерева из N вершин, имеющую номер i. Если i = 1, то у вершины i нет отца. Если i > 1, то ее отец имеет номер i div 2. Если 2i < N, то у вершины i есть два сына с номерами 2i и 2i+1. Если 2i = N, то единственный сын вершины i имеет номер 2i. Если 2i > N, то у вершины i нет сыновей. Будем говорить что объекты, хранящиеся в дереве, образуют бинарную кучу, если ключ объекта, находящегося в любой вершине, всегда не превосходит ключей объектов в сыновьях этой вершины. Будем хранить бинарную кучу в массиве H. Элемент этого массива H[i] будет содержать объект, находящийся в вершине дерева с номером i.
- **Свойство 3.** В бинарной куче объект H[1] (или объект, хранящийся в корне дерева) имеет минимальное значение ключа из всех объектов.
- Реализация операции MINIMUM, работающая за O(1).

Добавление нового элемента в кучу



Это «всплытие» продолжается до тех пор, пока ключ объекта не станет больше (или равен) ключа его отца или пока объект не «всплывет» до самого корня дерева. Время работы операции INSERT прямо пропорционально высоте дерева - O(log N).

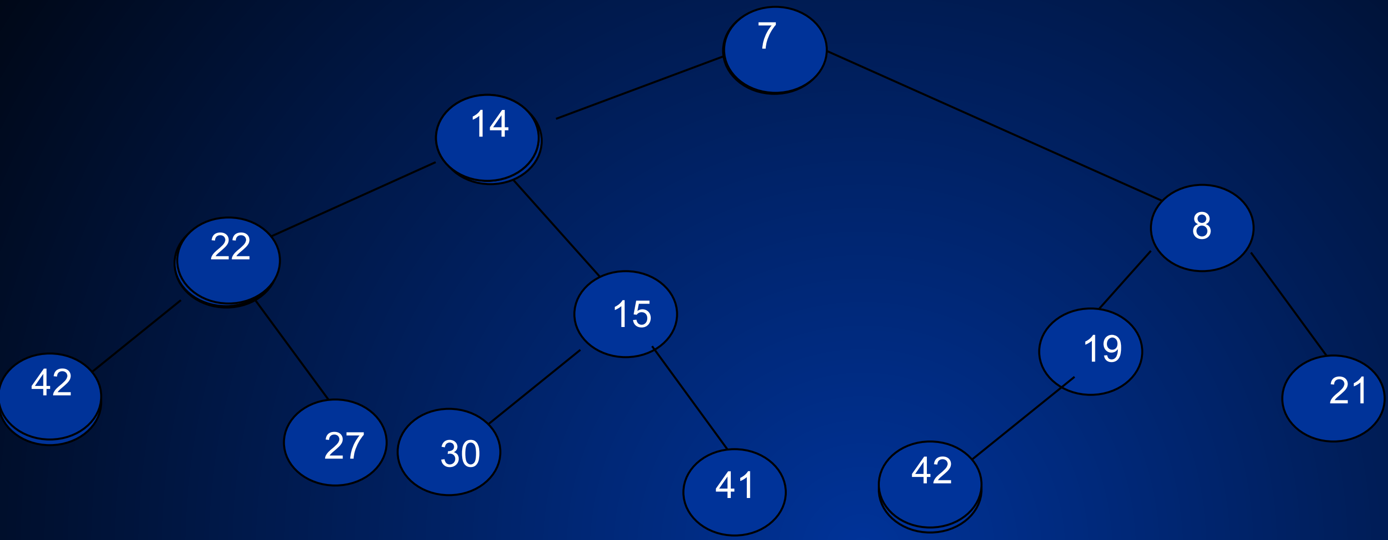
Реализация операции MINIMUM, работающая за O(1).

```
function MINIMUM:тип;
begin
    MINIMUM:=H[1];
End;
```

Рассмотрим операцию INSERT. Сначала мы помещаем добавляемый объект x на самый нижний уровень дерева на первое свободное место. Если окажется, что ключ этого объекта больше (или равен) ключа его отца, то свойство кучи нигде не нарушено, и мы корректно добавили вершину в кучу. В противном случае, поменяем местами объект с его отцом. В результате вершина с добавляемым объектом «всплывает» на одну позицию вверх. Это «всплытие» продолжается до тех пор, пока ключ объекта не станет больше (или равен) ключа его отца или пока объект не «всплывет» до самого корня дерева. Время работы операции INSERT прямо пропорционально высоте дерева - O(log N).

```
Procedure INSERT (x:тип);
begin
   N:= N+1;   H[N]:= x;   i:= N;
   While (i > 1) and (H[i].Key < H[i div 2].Key) Do Begin
      S:= H[i];      H[i]:= H[i div 2];      H[i div 2]:= S;      i:= i div 2;
   End;
End;
```

Удаление минимального элемента из кучи



Если теперь окажется, что ключ объекта в корне меньше (или равен) ключей объектов в его сыновьях (что очень маловероятно), то свойство кучи нигде не нарушено и удаление было проведено корректно. В противном случае, выберем сына корня с минимальным значением ключа и поменяем объект в корне с объектом в этом сыне. В результате объект, находившийся в корне, «спускается» на одну позицию вниз.

Теперь рассмотрим операцию EXTRACT-MIN. Для ее реализации мы сначала перемещаем объект из листа с номером N в корень (при этом объект в корне затирается, так как его и нужно удалить). Ставший свободным при этом лист удаляется. Если теперь окажется, что ключ объекта в корне меньше (или равен) ключей объектов в его сыновьях (что очень маловероятно), то свойство кучи нигде не нарушено и удаление было проведено корректно. В противном случае, выберем сына корня с минимальным значением ключа и поменяем объект в корне с объектом в этом сыне. В результате объект, находившийся в корне, «спускается» на одну позицию вниз. Этот «спуск» продолжается до тех пор, пока объект не окажется в листе или его ключ не станет меньше (или равен) ключей объектов в его сыновьях. Операция выполняется за O(log N), так как время ее работы пропорционально высоте дерева.

```
Procedure EXTRACT-MIN;
begin
  H[1]:= H[N];    N:= N-1;    i:= 1;
  While 2*i <= N Do Begin
      If (2*i = N) or (H[2*i].Key < H[2*i+1].Key)
      Then  Min:= 2*i      Else Min:= 2*i+1;
      If  H[i].Key <= H [Min].Key      Then Break;
      S:= H[i];      H [i]:= H [Min];     H [Min]:= S;    i:= Min;
  End;
End;
```

Алгоритм сжатия информации методом Хаффмана

- Сжатие информации происходит за счет устранения ее избыточности:
- Предложение ‘мама мыла раму.‘ состоит из 15 символов, каждый символ занимает в памяти ПК 1 байт, следовательно, все предложение занимает 15 байт

- мама мыла раму.
- 0001000110000101110001100110101001110111

17 133 198 106

м–4
а–4
_–2
ы–1
л–1
р–1
у–1
.–1

ма - 8
_ы - 3
лр - 2
у. - 2
лру. - 4
_ылру. - 7

0 1 0 1 0 1 0 1 0 1 0 0 1

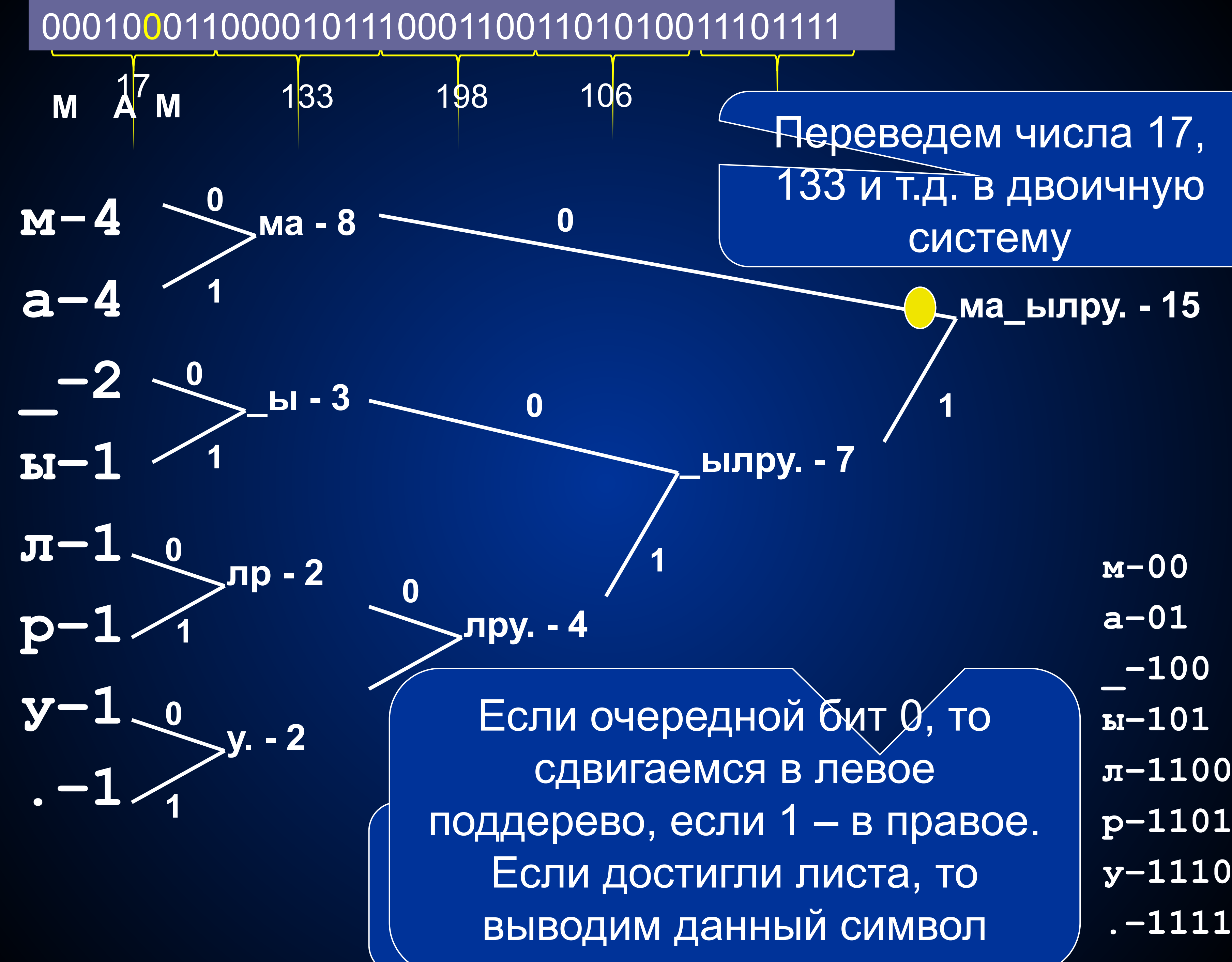
Заменим каждый символ его кодом,

В результате вместо 15 байт мы получили 5 полных байт и один неполный (1 бит).

Будем объединять два символа с наименьшими частотами

Обозначим левую связь - 0, а правую - 1

м–00
а–01
_–100
ы–101
л–1100
р–1101
у–1110
.–1111



000100011000010111000110011010100111011111
17
133
198
106
м
А
м
Переведем числа 17, 133 и т.д. в двоичную систему
м–4
а–4
_–2
ы–1
л–1
р–1
у–1
.–1
0
1
ма - 8
_ы - 3
лр - 2
у. - 2
лру. - 4
_ылру. - 7
ма_ылру. - 15
м–00
а–01
_–100
ы–101
л–1100
р–1101
у–1110
.–1111
Если очередной бит 0, то сдвигаемся в левое поддерево, если 1 – в правое. Если достигли листа, то выводим данный символ

Программа-архиватор

Алгоритм работы:

1. Подсчет частот вхождения каждого символа.
2. Построение дерева Хаффмана
3. Построение таблицы кодов символов
4. Замена символов текста двоичными кодами
5. Разбиение двоичной последовательности на группы по 8 бит и преобразование каждой группы в десятичное число типа byte

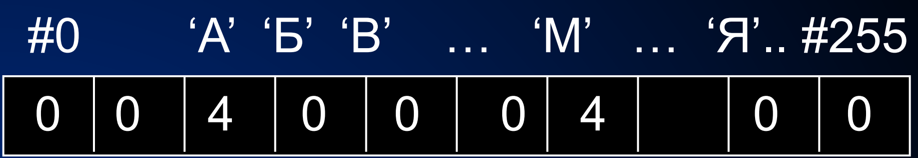
МАМА МЫЛА РАМУ.

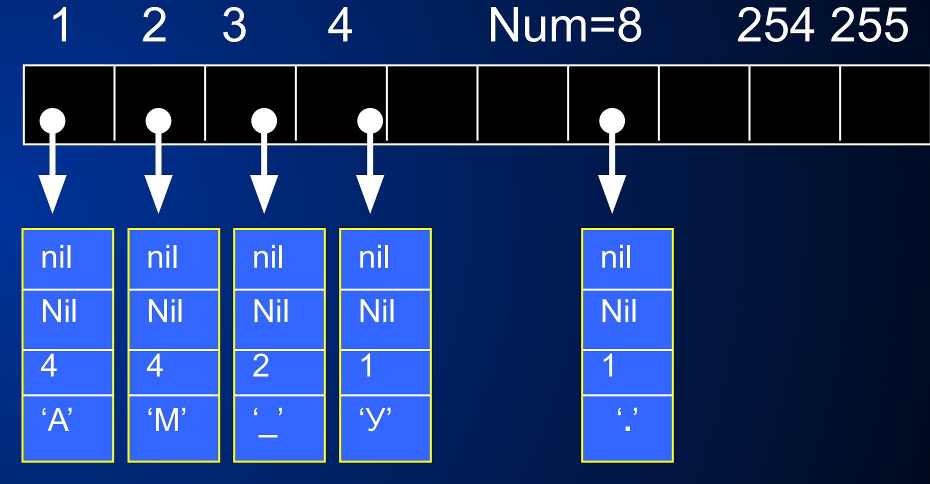
#0		‘А’	‘Б’	‘В’	…	‘М’	…	‘Я’..	#255
0	0	2	0	0	0	2		0	0

```
fillchar(m,sizeof(m),0);
while not eof(f) do
  begin
    read(f,a);
    inc(m[a])
  end;
```

Преобразуем массив m в массив динамических звеньев Ch

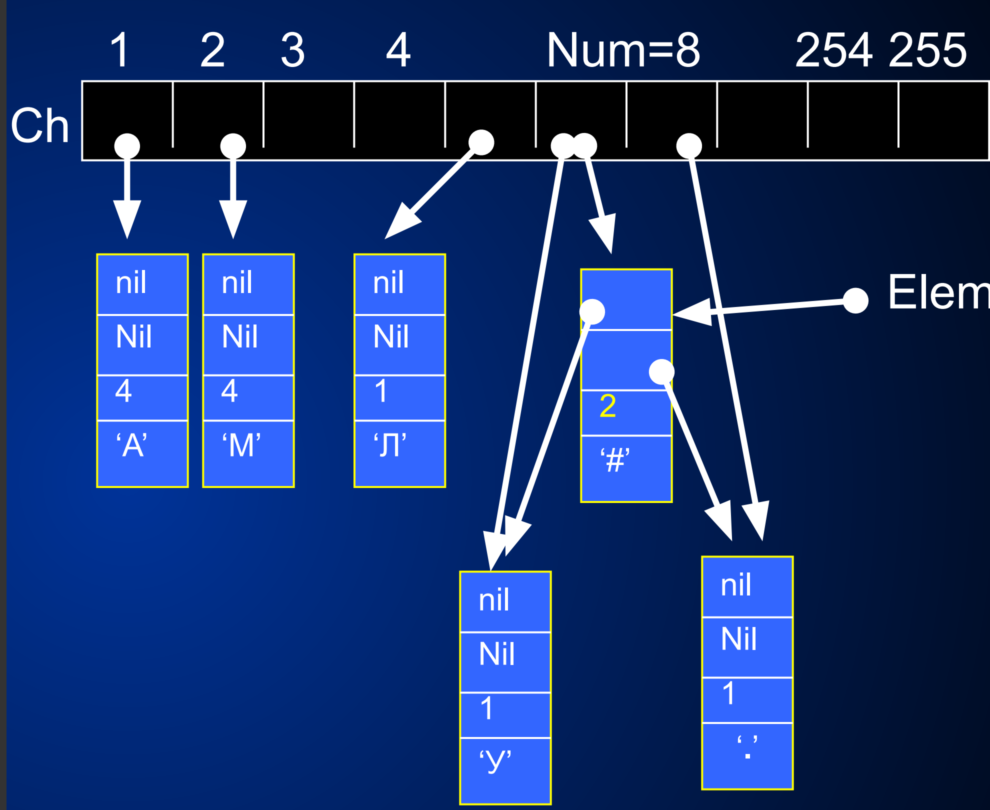
```
Numb:=0;
 for a:=#0 to #255 do
  if m[a]<>0
  then begin
       inc(Numb);
       New(Ch[numb]);
       Ch[numb]^.Lit:=a;
       Ch[numb]^.Number:=m[a];
       Ch[numb]^.Left:=nil;
       Ch[numb]^.Right:=nil;
     end;
```





Сортировка массива Ch и построение дерева

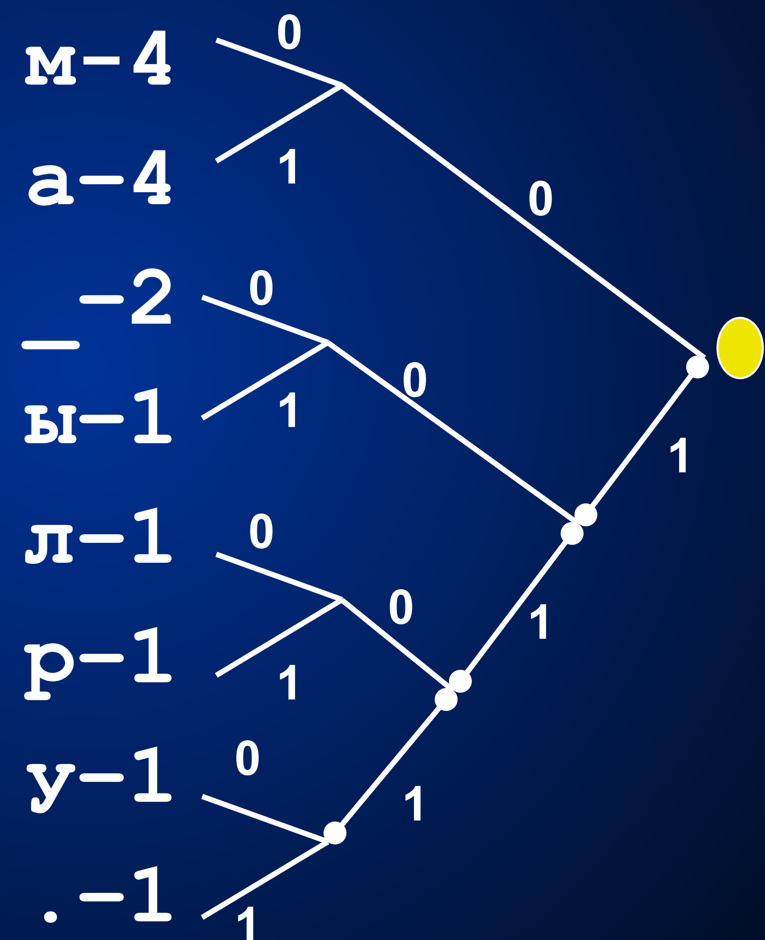
```
While Numb>1 do
    begin
      Sort(Ch);{Сортируем массив частот вхождения символов по убыванию}
      New(Elem); {Создаем новое звено}
      Elem^.Left:=Ch[Numb-1]; {"Прицепляем" два звена с наименьшим весом}
      Elem^.Right:=Ch[Numb];
      Dec(Numb);
      Ch[Numb]:=Elem; {Новое звено размещаем в массиве}
    end;
```



Построение таблицы кодировки символов

```
procedure CalcCode(Root:ref;s:tStr);
begin
  if (Root^.left=nil)and(Root^.Right=nil)
{Если это "Лист"}
  then {Запомнить его код}
        Code[Root^.Lit]:=s
   else begin
        if (Root^.left<>nil)  {Если есть левое
поддерево, то сместиться туда}
        then CalcCode(Root^.left,s+'0');
        if (Root^.Right<>nil) {Если есть
правое поддерево, то сместиться туда}
        then CalcCode(Root^.Right,s+'1')
      end
end;
```

#0	'_'	'А'	'Б'	'В'	…	'М'	…	'Ы'..#255	
	'100'	'01'				'00'		'101'	



Кодирование входного файла

```
S:='';
 while not eof(f) do
   begin
      read(f,a);
      s:=s + Code [ a] ;
      if length(s)>=8
      Then begin
               b:=BinToDec(copy(s,1,8));
               Delete(s,1,8);
               write(fOut,b)
            end
 end;
Доклеить s '0' до 8 символов
write(fOut,BinToDec(copy(s,1,8)));
write(fOut, длину неполного байта);
```

#0	'_'	'А'	'Б'	'В'	…	'М'	…	'Ы'..#255
	'100'	'01'				'00'		'101'	

МАМА МЫЛА РАМУ.

s 0

17 133 198

Массив частот m
17
133
198
…
Длина неполного байта

м-00
а-01
_-100
ы-101
л-1100
р-1101
у-1110
.-1111

Распаковка сжатого файла

```
Считать массив частот m;
Построить дерево Хаффмана;
Len:=FileSize(f)-1-SizeOf(m);
S:=‘’; p:=Root;
For z:=1 to Len-1 do
   begin
       read(f,a);
       s:=s + DecToBin(a) ;
       if length(s)>=220
       Then begin
                   for i:=1 to length(s) do
                     if p^.Left=p^.Right
                     then write(fOut, p^.Key)
                     else if s[i]=‘0’
                             then p:=p^.left
                             else p:=p^.rigth
                   s:=‘’;
              end
 end;
```

Вырезать информационные биты из неполного байта и распаковать оставшиеся символы

s 00010001100000010111000110

‘М’ ‘А’