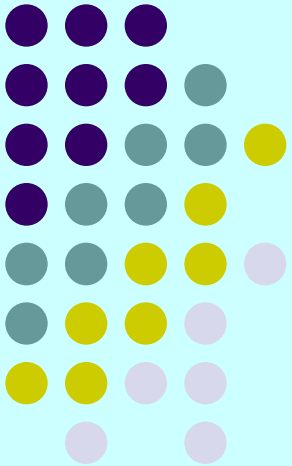
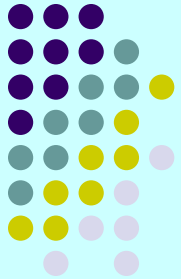
Тема 3
ЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ

Абстрактный тип данных «Список»

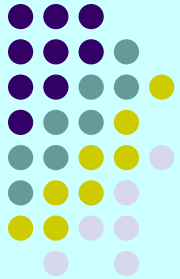




Списки являются чрезвычайно гибкой структурой, так как их легко сделать большими или меньшими, и их элементы доступны для вставки или удаления в любой позиции списка.

Списки также можно объединять или разбивать на меньшие списки.

Списки регулярно используются в приложениях, например в программах информационного поиска, трансляторах программных языков или при моделировании различных процессов. Ряд методов управления памятью широко используют технику обработки списков.



В математике список представляет собой последовательность элементов определенного типа, который в общем случае будем обозначать как ***elementtype*** (тип элемента).

Список можно представить в виде последовательности элементов, разделенных запятыми:

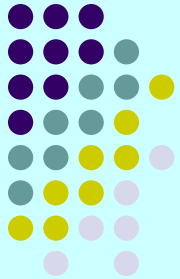
$a_1, a_2, \ldots, a_n$,

где $n \geq 0$ и все a_i имеют тип ***elementtype***.

Количество элементов ***n*** будем называть ***длиной списка***.

Если $n \geq 1$, то a_1 называется первым элементом, а a_n - последним элементом списка.

В случае $n = 0$ имеем пустой список, который не содержит элементов.



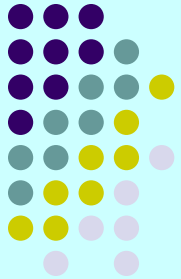
Важное свойство списка заключается в том, что его элементы можно линейно упорядочить в соответствии с их позицией в списке.

Мы говорим, что элемент a_i предшествует a_{i+1} для i=1, 2, ..., n-1 и a_i следует за a_{i-1} для i=2, 3, …, n. Мы также будем говорить, что элемент a_i имеет позицию i.

Кроме того, мы постулируем существование позиции, следующей за последним элементом списка.

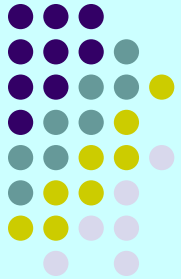
Функция ***END(L)*** будет возвращать позицию, следующую за позицией n в n-элементном списке L.

Позиция ***END(L)***, рассматриваемая как расстояние от начала списка, может изменяться при увеличении или уменьшении списка, в то время как другие позиции имеют фиксированное (неизменное) расстояние от начала списка.



Для формирования абстрактного типа данных на основе математического определения списка необходимо задать множество операторов, выполняемых над объектами типа ***LIST*** (***Список***).

Однако не существует одного множества операторов, выполняемых над списками, подходящего сразу всем возможным приложениям.



Чтобы показать некоторые общие операции, выполняемые над списками, предположим, что имеем приложение, содержащее список почтовой рассылки, который мы хотим очистить от повторяющихся адресов.

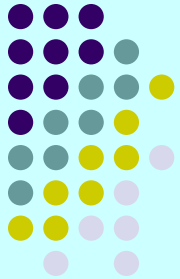
Концептуально эта задача решается очень просто: для каждого элемента списка удаляются все последующие элементы, совпадающие с данным.

Однако для записи такого алгоритма необходимо определить операторы, которые должны найти первый элемент списка, перейти к следующему элементу, осуществить поиск и удаление элементов.

Примем обозначения:

- ***L*** - список объектов типа ***elementtype***,
- ***x*** - объект этого типа,
- ***p*** - позиция элемента в списке. Отметим, что "позиция" имеет другой тип данных, чья реализация может быть различной для разных реализаций списков. Обычно мы понимаем позиции как множество целых положительных чисел, но на практике могут встретиться другие представления.

Операции, выполняемые над списком



1) ***INSERT(x, p, L).***

Эта операция вставки объекта ***x*** в позицию *p* в списке ***L***, с перемещением элементов от позиции ***p*** и далее в следующую, более высокую позицию.

Таким образом, если список ***L*** состоит из элементов

$$a_1, a_2, \ldots, a_n,$$

то после выполнения этой операции он будет иметь вид

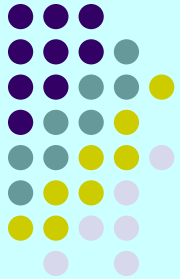
$$a_1, a_2, \ldots, a_{p-1}, x, a_p, \ldots, a_n.$$

Если ***p*** принимает значение ***END(L)***, то будем иметь

$$a_1, a_2, \ldots, a_n, x.$$

Если в списке ***L*** нет позиции ***p***, то результат выполнения операции не определен.

Операции, выполняемые над списком



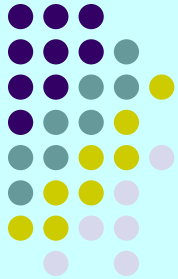
2) ***LOCATE(x, L).***

Эта функция возвращает позицию объекта x в списке ***L***.

Если в списке объект ***x*** встречается несколько раз, то возвращается позиция первого от начала списка объекта ***x***.

Если объекта ***x*** нет в списке ***L***, то возвращается ***END(L)***.

Операции, выполняемые над списком



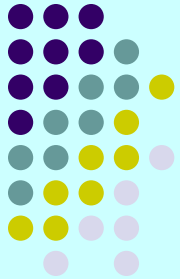
3) ***RETRIEVE(p, L).***

Эта функция возвращает элемент, который стоит в позиции ***p*** в списке ***L***.

Результат не определен, если ***p = END(L)*** или в списке ***L*** нет позиции ***p***.

Отметим, что элементы должны быть того типа, который в принципе может возвращать функция. Однако на практике мы всегда можем изменить эту функцию так, что она будет возвращать указатель на объект типа ***elementtype***.

Операции, выполняемые над списком



4) ***DELETE(p, L).***

Эта функция удаляет элемент в позиции ***p*** списка ***L***.

Так, если список ***L*** состоит из элементов

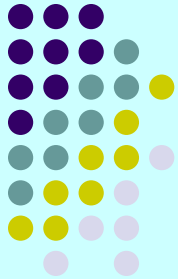
$$a_1, a_2, \ldots, a_n,$$

то после выполнения этого оператора он будет иметь вид

$$a_1, a_2, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n.$$

Результат не определен, если в списке ***L*** нет позиции ***p*** или ***p=END(L)***.

Операции, выполняемые над списком



5) ***NEXT(p, L)*** и ***PREVIOUS(p, L).***

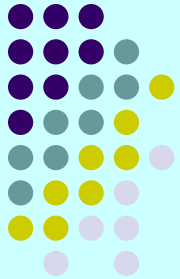
Эти функции возвращают соответственно следующую и предыдущую позиции от позиции ***p*** в списке ***L***.

Если ***p*** - последняя позиция в списке ***L***, то ***NEXT(p, L)*** = ***END(L)***. Функция ***NEXT*** не определена, когда ***p = END(L)***.

Функция ***PREVIOUS*** не определена, если ***p*** = 1.

Обе функции не определены, если в списке ***L*** нет позиции ***p***.

Операции, выполняемые над списком



6) ***MAKENULL(L).***

Эта функция делает список ***L*** пустым и возвращает позицию ***END(L)***.

7) ***FIRST(L).***

Эта функция возвращает первую позицию в списке ***L***.

Если список пустой, то возвращается позиция ***END(L)***.

8) ***PRINTLIST(L).***

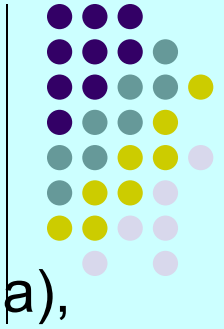
Печатает элементы списка ***L*** в порядке из расположения.

Пример 3.1.

Используя описанные выше операции, создадим процедуру ***PURGE*** (***Очистка***), которая в качестве аргумента использует список и удаляет из него повторяющиеся элементы.

Элементы списка имеют тип ***elementtype***, а список таких элементов имеет тип ***LIST*** (данного соглашения мы будем придерживаться на протяжении всей этой темы).

Пример 3.1.



Определим функцию ***same(x, y)***, где ***x*** и ***y*** имеют тип ***elementtype***, которая принимает значение ***true*** (истина), если ***x*** и ***y*** "одинаковые" (same), и значение ***false*** (ложь) в противном случае.

Поясним понятие "одинаковые". Если тип ***elementtype***, например, совпадает с типом действительных чисел, то мы можем положить, что функция ***same(x, y)*** будет иметь значение ***true*** тогда и только тогда, когда ***x*** = ***y***.

Но если тип ***elementtype*** является типом записи, содержащей поля почтового индекса (acctno), имени (name) и адреса абонента (address), этот тип можно объявить следующим образом:

```
type  elementtype = record
acctno: integer;
name: string[20]; address: string[50];
end
```

Теперь можно задать, чтобы функция ***same(x, y)*** принимала значение ***true*** всякий раз, когда ***x.acctno*** = ***y.acctno***.

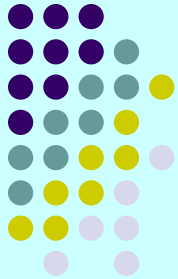
Листинг 3.1. Программа удаления совпадающих элементов

Переменные ***p*** и ***q*** используются для хранения двух позиций в списке. В процессе выполнения программы слева от позиции ***p*** все элементы уже не имеют дублирующих своих копий в списке.

В каждой итерации цикла (2)-(8) переменная ***q*** используется для просмотра элементов, находящихся в позициях, следующих за позицией ***p***, и удаления дубликатов элемента, располагающегося в позиции ***p***. Затем ***p*** перемещается в следующую позицию и процесс продолжается.

```
procedure PURGE ( var L: LIST);
      var  p, q: position;
begin
(1)  p:= FIRST(L);
(2)  while p <> END(L)   do   begin
(3)       q:= NEXT(p, L);
(4)       while q <> END(L) do
(5)        if  same(RETRIEVE(p, L), RETRIEVE(q, L))  then
(6)              DELETE(q, L)
(7)              else    q:= NEXT(q, L);
(8)       p:= NEXT(p, L)
           end
end;             { PURGE }
```

Реализация списков посредством массивов



При реализации списков с помощью массивов элементы списка располагаются в смежных ячейках массива.

Это представление позволяет легко просматривать содержимое списка и вставлять новые элементы в его конец.

Но вставка нового элемента в середину списка требует перемещения всех последующих элементов на одну позицию к концу массива, чтобы освободить место для нового элемента.

Удаление элемента также требует перемещения элементов, чтобы закрыть освободившуюся ячейку.

Реализация списков посредством массивов

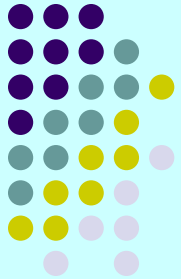
При использовании массива мы определяем тип ***LIST*** как запись, имеющую два поля. Первое поле ***elements*** (элементы) - это элементы массива, чей размер считается достаточным для хранения списков любой длины, встречающихся в данной реализации или программе. Второе поле целочисленного типа ***last*** (последний) указывает на позицию последнего элемента (

i-й элемент списка, если 1≤***i***≤***last***, находится в ***i***-й ячейке массива.

Позиции элементов в списке представимы в виде целых чисел, таким образом, ***i***-я позиция - это просто целое число ***i***.

Константа ***maxlength*** определяет максимальный размер массива. Функция ***END(L)*** возвращает значение ***last+1***.

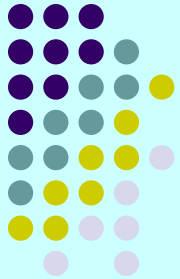
Реализация списков посредством массивов



Приведем необходимые объявления:

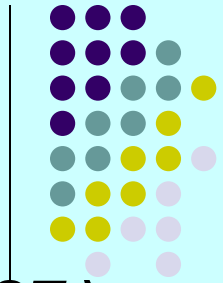
```
const   maxlength = 100  { или другое подходящее число };
type   LIST = record
            elements: array[1 .. maxlength]  of  elementtype;
            last: integer
       end;
       position = integer;

function  END ( var L: LIST ): position;
begin
    END:=  L.last + 1
end;
```

Реализация списков посредством массивов



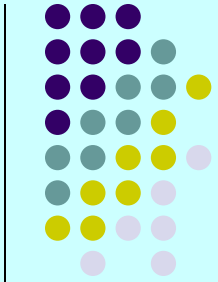
- В листинге 3.2 показано, как можно реализовать операторы ***INSERT***, ***DELETE*** и ***LOCATE*** при представлении списков с помощью массивов.
- Оператор ***INSERT*** перемещает элементы из позиций ***p***, ***p+1***, ..., ***last*** в позиции ***p+1***, ***p+2***, …, ***last+1*** и помещает новый элемент в позицию ***p***.
- Если в массиве уже нет места для нового элемента, то инициализируется подпрограмма ***error*** (ошибка), распечатывающая соответствующее сообщение, затем выполнение программы прекращается.
- Оператор ***DELETE*** удаляет элемент в позиции ***p***, перемещая элементы из позиций ***p+1***, ***p+2***, …, ***last*** в позиции ***p***, ***p+1***, ..., ***last-1***.
- Функция ***LOCATE*** последовательно просматривает элементы массива для поиска заданного элемента. Если этот элемент не найден, то возвращается ***last+1***.

Листинг 3.2. Реализация списков посредством массивов



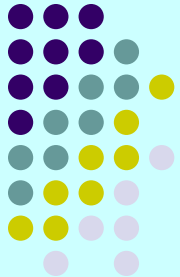
```
procedure INSERT (x: elementtype; p: position; var L: LIST );
      { INSERT вставляет элемент x в позицию p в списке L }
var   q: position;
begin
      if   L.last >= maxlength   then   error('Список полон')
      else   if  (p > L.last + 1) or (p < 1) then
                  error('Такой позиции не существует')
            else begin
                  for   q:= L.last  downto  p  do
                        { перемещение элементов из позиций p, p+1, ... на
                          одну позицию к концу списка }
            L.elements[q+1]:= L.elements[q];
                  L.last:= L.last + 1;
                  L.elements[p]:= x
            end
end;          { INSERT }
```

Листинг 3.2. Реализация списков посредством массивов



```
procedure   DELETE ( p: position; var L: LIST );
      { DELETE удаляет элемент в позиции p списка L }
var    q: position;
begin
    if   (p > L.last) or (p < 1)  then  error('Такой позиции не
                        существует')
    else   begin
     L.last:= L.last - 1;
     for  q:= p  to  L.last  do
                { перемещение элементов из позиций p+1, p+2, ...
                на одну позицию к началу списка }
          L.elements[q]:= L.elements[q+1]
    end
end;          { DELETE }
```

Листинг 3.2. Реализация списков посредством массивов



```
procedure   LOCATE ( x: elementtype; L: LIST ): position;
      { LOCATE возвращает позицию элемента x в списке L }
var    q: position;
begin
     for  q:= 1   to  L.last   do
          if   L.elements[q] = x   then  LOCATE:=q;
     LOCATE:= L.last +1   { элемент x не найден }
end;          { LOCATE }
```

Легко видеть, как можно записать другие операторы списка, используя данную реализацию списков.

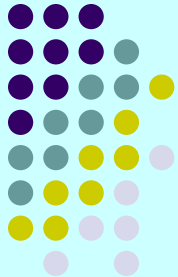
Функция ***FIRST*** всегда возвращает 1.

Функция ***NEXT*** возвращает значение, на единицу большее аргумента.

Функция ***PREVIOUS*** возвращает значение, на единицу меньшее аргумента.

Оператор ***MAKENULL(L)*** устанавливает ***L.last*** в 0.

Реализация списков посредством указателей



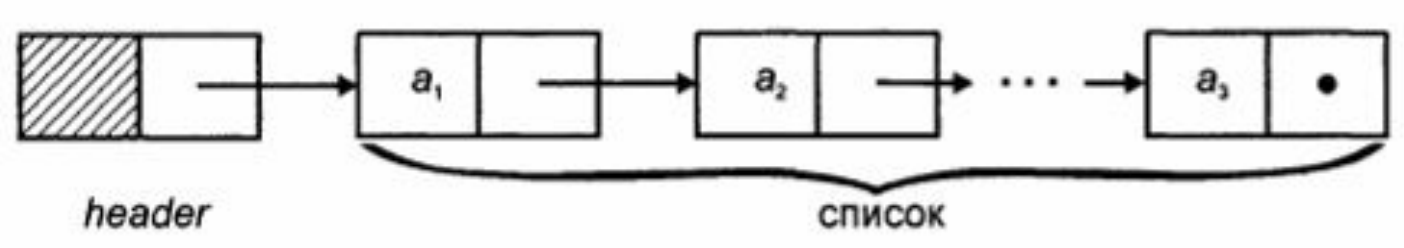
Для реализации однонаправленных списков используются указатели, связывающие последовательные элементы списка.

Эта реализация освобождает нас от использования непрерывной области памяти для хранения списка и, следовательно, от необходимости перемещения элементов списка при вставке или удалении элементов.

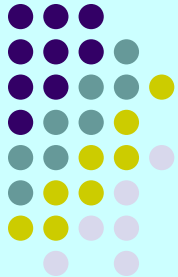
Однако ценой за это удобство становится дополнительная память для хранения указателей.

Реализация списков посредством указателей

- В этой реализации список состоит из ячеек, каждая из которых содержит элемент списка и указатель на следующую ячейку списка.
- Если список состоит из элементов a_1, a_2, ..., a_n, то для i=1, 2, ..., n-1 ячейка, содержащая элемент a_i имеет также указатель на ячейку, содержащую элемент a_{i+1}.
- Ячейка, содержащая элемент a_n, имеет указатель ***nil*** (нуль).
- Имеется также ячейка ***header*** (***заголовок***), которая указывает на ячейку, содержащую a_1. Ячейка ***header*** не содержит элементов списка. В случае пустого списка заголовок имеет указатель ***nil***, не указывающий ни на какую ячейку. На рис показан связанный список описанного вида.



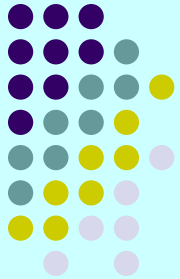
Реализация списков посредством указателей



- Для однонаправленных списков удобно использовать определение позиций элементов, отличное от того определения позиций, которое применялось реализации списков с помощью массивов. Здесь для ***i***=2, 3, ..., ***n*** позиция ***i*** определяется как указатель на ячейку, содержащую указатель на элемент a_i.
- Позиция 1 - это указатель в ячейке заголовка, а позиция ***END(L)*** - указатель в последней ячейке списка ***L***.
- Формально определить структуру связанного списка можно следующим образом:

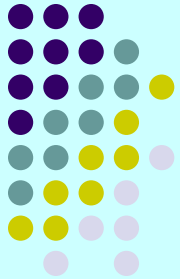
```
type   celltype = record
                    element: elementtype;
                    next: ^ celltype
                  end;
         LIST = ^ celltype;
    position = ^ celltype;
```

Листинг 3.3. Реализация списков посредством указателей



- Результат функции ***END(L)*** получаем путем перемещения указателя ***q*** от начала списка к его концу, пока не будет достигнут конец списка, который определяется тем, что ***q*** становится указателем на ячейку с указателем ***nil***.
- Но эта реализация функции ***END*** неэффективна, так как требует просмотра всего списка при каждом вычислении этой функции. Если необходимо частое использование данной функции, как в программе ***PURGE*** (листинг 3.1), то можно сделать на выбор следующее.
  1. Применить представление списков, которое не использует указатели на ячейки.
  2. Исключить использование функции ***END(L)*** там, где это возможно, заменив ее другими операторами. Например, условие ***p <> END(L)*** в строке (2) листинга 3.1 можно заменить условием ***p^.next <> nil***, но в этом случае программа становится зависимой от реализации списка.

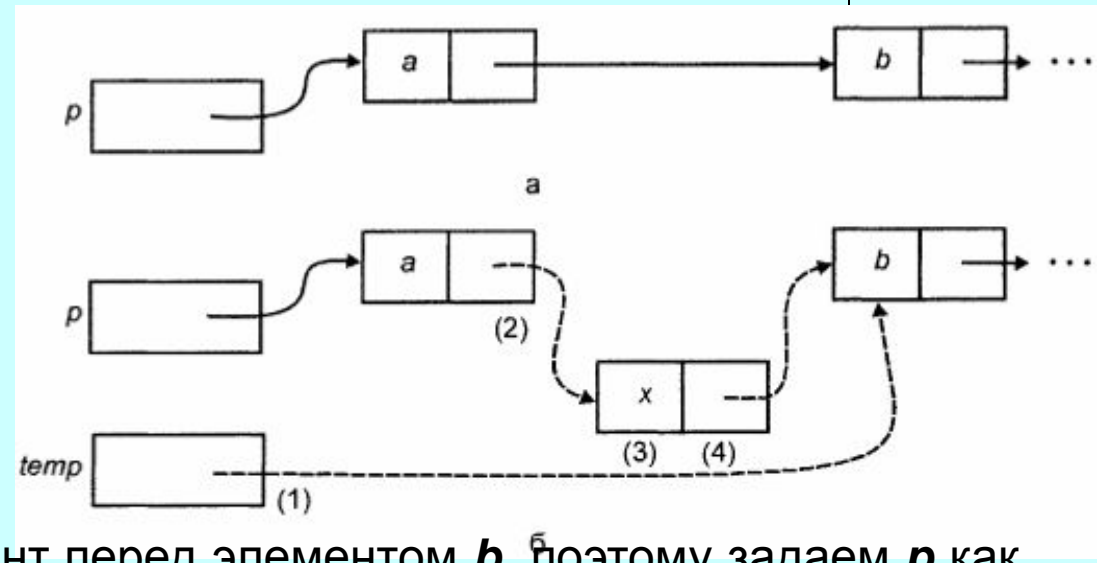
Листинг 3.3. Реализация списков посредством указателей



```
function   END ( L: LIST ): position;
      { END возвращает указатель на последнюю ячейку списка L }
     Var   q: position;
begin
(1)  q:= L;
(2)  while  q^.next <> nil   do
(3)       q:= q^.next;
(4)  END:=q
end;          { END }
```

Листинг 3.3. Реализация списков посредством указателей

```
procedure   INSERT ( x: elementtype; p: position );
var    temp: position;
begin
(1)    temp:= p^.next;
(2)    new(p^. next) ;
(3)    p^.next^.element:= x;
(4)    p^.next^.next:= temp
end;          { INSERT }
```

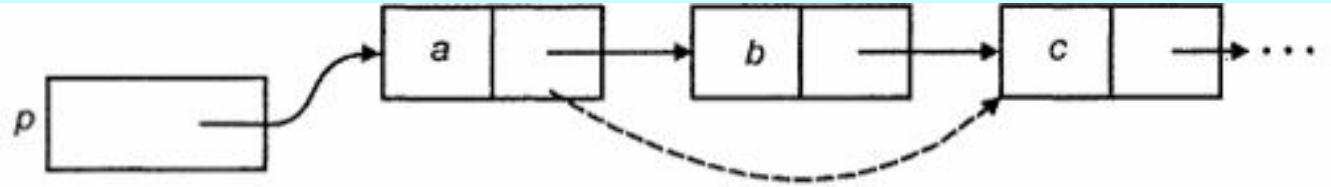


Мы хотим вставить новый элемент перед элементом ***b***, поэтому задаем ***p*** как указатель на ячейку, содержащую элемент ***b***. В строке (2) листинга создается новая ячейка, а в поле ***next*** ячейки, содержащей элемент ***a***, ставится указатель на новую ячейку. В строке (3) поле ***element*** вновь созданной ячейки принимает значение ***x***, а в строке (4) поле ***next*** этой ячейки принимает значение переменной ***temp***, которая хранит указатель на ячейку, содержащую элемент ***b***.

На рис. б представлен результат выполнения процедуры ***INSERT***, где пунктирными линиями показаны новые указатели и номерами (совпадающими с номерами строк в листинге 3.3) помечены этапы ее создания.

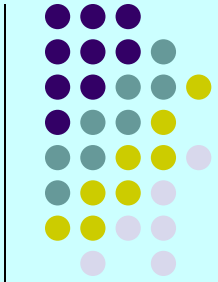
Листинг 3.3. Реализация списков посредством указателей

```
procedure   DELETE ( p: position );
begin
        p^.next:= p^.next^.next
end;        { DELETE }
```



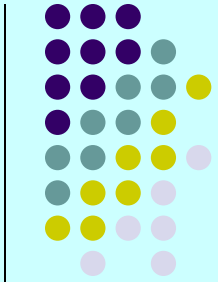
Показана схема манипулирования указателем в этой процедуре. Старые указатели показаны сплошными линиями, а новый - пунктирной.

Листинг 3.3. Реализация списков посредством указателей



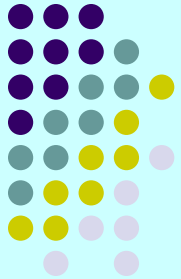
```
function   LOCATE ( x: elementtype; L: LIST ): position;
var   p: position;
begin
     p:= L;
     while   p^.next <> nil   do
          if   p^.next^.element = x   then  LOCATE:=p
          else   p:= p^.next;
     LOCATE:=p    { элемент не найден }
end;          { LOCATE }

function   MAKENULL ( var L: LIST ): position;
begin
     new(L);     L^.next:= nil;
     MAKENUL:=L
end;          { MAKENULL }
```

Реализация списков посредством указателей



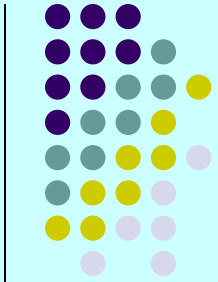
- Еще раз подчеркнем, что позиции в реализации однонаправленных списков ведут себя не так, как позиции в реализации списков посредством массивов. Предположим, что есть список из трех элементов ***a***, ***b*** и ***c*** и переменная ***p*** типа ***position*** (позиция), чье текущее значение равно позиции 3, т.е. это указатель, находящийся в ячейке, содержащей элемент ***b***, и указывающий на ячейку, содержащую элемент ***c***.
- Если теперь мы выполним команду вставки нового элемента ***x*** в позицию 2, так что список примет вид ***a***, ***x***, ***b***, ***c*** , элемент ***b*** переместится в позицию 3.

Реализация списков посредством указателей



- Если бы мы использовали реализацию списка с помощью массива, то элементы ***b*** и ***c*** должны были переместиться к концу массива, так что элемент ***b*** в самом деле должен оказаться на третьей позиции.
- Однако при использовании реализации списков с помощью указателей значение переменной ***p*** (т.е. указатель в ячейке, содержащей элемент ***b***) вследствие вставки нового элемента не изменится, продолжая указывать на ячейку, содержащую элемент ***c***. Значение этой переменной надо изменить, если мы хотим использовать ее как указатель именно на третью позицию, т.е. как указатель на ячейку, содержащую элемент ***b***.

Сравнение реализаций



1. Реализация списков с помощью массивов требует указания максимального размера списка до начала выполнения программ. Если мы не можем заранее ограничить сверху длину обрабатываемых списков, то, очевидно, более рациональным выбором будет реализация списков с помощью указателей.
2. Выполнение некоторых операторов в одной реализации требует больших вычислительных затрат, чем в другой. Например, процедуры ***INSERT*** и ***DELETE*** выполняются за постоянное число шагов в случае связанных списков любого размера, но требуют времени, пропорционального числу элементов, следующих за вставляемым (или удаляемым) элементом, при использовании массивов. И наоборот, время выполнения функций ***PREVIOUS*** и ***END*** постоянно при реализации списков посредством массивов, но это же время пропорционально длине списка в случае реализации, построенной с помощью указателей.

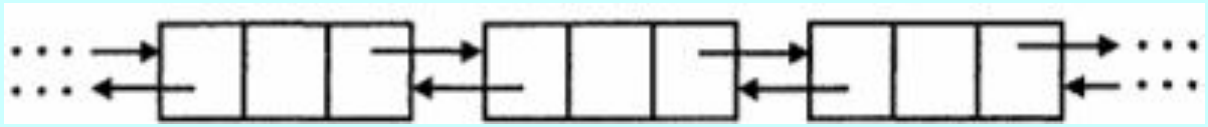
Сравнение реализаций

3. Если необходимо вставлять или удалять элементы, положение которых указано с помощью некой переменной типа ***position***, и значение этой переменной будет использовано позднее, то не целесообразно использовать реализацию с помощью указателей, поскольку эта переменная не "отслеживает" вставку и удаление элементов, как показано выше. Вообще использование указателей требует особого внимания и тщательности в работе.
4. Реализация списков с помощью массивов расточительна в отношении компьютерной памяти, поскольку резервируется объем памяти, достаточный для максимально возможного размера списка независимо от его реального размера в конкретный момент времени. Реализация с помощью указателей использует столько памяти, сколько необходимо для хранения текущего списка, но требует дополнительную память для указателя каждой ячейки. Таким образом, в разных ситуациях по критерию используемой памяти могут

Двунаправленные списки

Во многих приложениях возникает необходимость организовать эффективное перемещение по списку как в прямом, так и в обратном направлениях. Или по заданному элементу нужно быстро найти предшествующий ему и последующий элементы.

В этих ситуациях можно дать каждой ячейке указатели и на следующую, и на предыдущую ячейки списка, т.е. организовать дважды связный список.



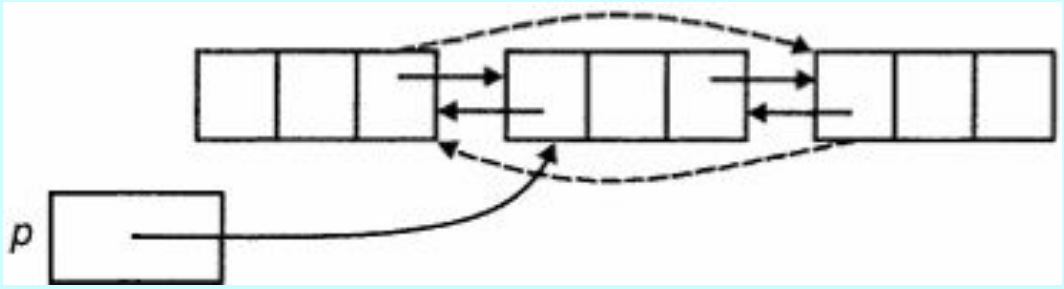
Двунаправленные списки

- Другое важное преимущество двунаправленных списков заключается в том, что мы можем использовать указатель ячейки, содержащей ***i***-й элемент, для определения ***i***-й позиции - вместо использования указателя предшествующей ячейки. Но ценой этой возможности являются дополнительные указатели в каждой ячейке и определенное удлинение некоторых процедур, реализующих основные операторы списка.
- Если мы используем указатели, то объявление ячеек, содержащих элементы списка и два указателя, можно выполнить следующим образом {***previous*** - поле, содержащее указатель на предшествующую ячейку):

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

Листинг 3.4. Удаление элемента из двунаправленного списка

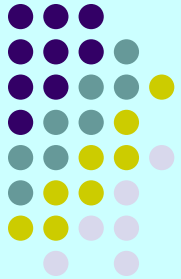
- Схема удаления элемента в позиции p списка в предположении, что удаляемая ячейка не является ни первой, ни последней в списке:



На этой схеме сплошными линиями показаны указатели до удаления, а пунктирными - после удаления элемента.

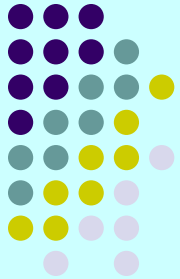
- На практике обычно делают так, что ячейка заголовка дважды связного списка "замыкает круг" ячеек, т.е. указатель поля ***previous*** ячейки заголовка указывает на последнюю ячейку, а указатель поля ***next -*** на первую. Поэтому при такой реализации дважды связного списка нет необходимости в выполнении проверки на "нулевой указатель".

Листинг 3.4. Удаление элемента из двунаправленного списка



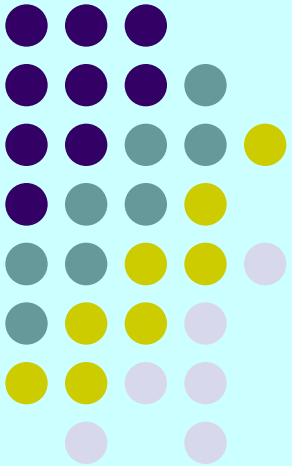
```
procedure  DELETE ( var p: position );
begin
     if   p^.previous <> nil   then
            { удаление ячейки, которая не является первой }
          p^.previous^.next:= p^.next;
     if   p^.next <> nil   then
            { удаление ячейки, которая не является последней }
          p^.next^. previous:= p^.previous
end;          { DELETE }
```

Листинг 3.4. Удаление элемента из двунаправленного списка



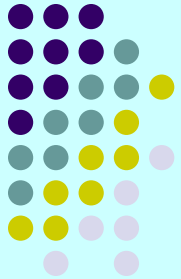
- В процедуре удаления сначала с помощью указателя поля ***previous*** определяется положение предыдущей ячейки.
- Затем в поле ***next*** этой (предыдущей) ячейки устанавливается указатель, указывающий на ячейку, следующую за позицией ***p***.
- Далее подобным образом определяется следующая за позицией ***p*** ячейка и в ее поле ***previous*** устанавливается указатель на ячейку, предшествующую позиции ***p***.
- Таким образом, ячейка в позиции ***p*** исключается из цепочек указателей и при необходимости может быть использована повторно.

Абстрактный тип данных «Стек»



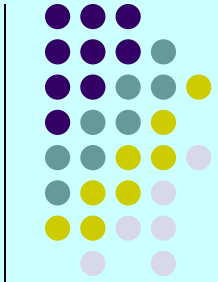
- ***Стек*** - это специальный тип списка, в котором все вставки и удаления выполняются только на одном конце, называемом ***вершиной*** (***top***).
- Стеки также иногда называют "магазинами", а в англоязычной литературе для обозначения стеков еще используется аббревиатура ***LIFO*** (last-in-first-out - последний вошел - первый вышел).
- Интуитивными моделями стека могут служить колода карт на столе при игре в покер, книги, сложенные в стопку, или стопка тарелок на полке буфета. Во всех этих моделях взять можно только верхний предмет, а добавить новый объект можно, только положив его на верхний.

Области применения стека



- передача параметров в функции;
- трансляция (синтаксический и семантический анализы, генерация кодов и т.д.);
- реализация рекурсии в программировании;
- реализация управления динамической памятью и т.п.

Операторы, выполняемые над стеком



1. ***MAKENULL(S).***

   Делает стек ***S*** пустым.

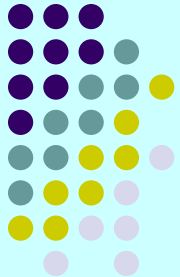
2. ***TOP(S).***

   Возвращает элемент из вершины стека ***S***. Обычно вершина стека идентифицируется позицией 1, тогда ***TOP(S)*** можно записать в терминах общих операторов списка как ***RETRIEVE(FIRST(S), S)***.

3. ***POP(S).***

   Удаляет элемент из вершины стека (выталкивает из стека). В терминах операторов списка этот оператор можно записать как ***DELETE(FIRST(S), S).***

Операторы, выполняемые над стеком



4. ***PUSH(x, S).***

   Вставляет элемент ***x*** в вершину стека ***S*** (заталкивает элемент в стек). Элемент, ранее находившийся в вершине стека, становится элементом, следующим за вершиной, и т.д. В терминах общих операторов списка данный оператор можно записать как

   INSERT(x, FIRST(S), S).

5. ***EMPTY(S).***

   Эта функция возвращает значение ***true*** (истина), если стек ***S*** пустой, и значение ***false*** (ложь) в противном случае.

Пример 3.2.

- Все текстовые редакторы имеют определенные символы, которые служат в качестве стирающих символов (erase character), т.е. таких, которые удаляют (стирают) символы, стоящие перед ними; эти символы "работают" как клавиша <Backspace> на клавиатуре компьютера. Например, если символ **#** определен стирающим символом, то строка ***abc#d##e*** в действительности является строкой ***ae***.
- Текстовые редакторы имеют также символ-убийцу (kill character), который удаляет все символы текущей строки, находящиеся перед ним. В этом примере в качестве символа-убийцы определим символ **@**.

Пример 3.2.

- Операции над текстовыми строками часто выполняются с использованием стеков.
- Текстовый редактор поочередно считывает символы, если считанный символ не является ни символом-убийцей, ни стирающим символом, то он помещается в стек.
- Если вновь считанный символ - стирающий символ, то удаляется символ в вершине стека.
- В случае, когда считанный символ является символом-убийцей, редактор очищает весь стек.
- В листинге 3.5 представлена программа, реализующая действия стирающего символа и символа-убийцы.

Листинг 3.5.

```
procedure EDIT;
var   s: STACK;    c: char;
begin
     MAKENULL(S);
     while   not eoln   do   begin
          read(c);
          if   c = '#'   then  POP(S)
          else   if   c = '@‘   then   MAKENULL(S);
               else    { c — обычный символ }
                    PUSH(c, S)
     end;
     печать содержимого стека S в обратном порядке
end;          { EDIT }
```

В этом листинге используется стандартная функция языка Pascal ***eoln***, возвращающая значение **true**, если текущий символ - символ конца строки.

Листинг 3.5.

- В этой программе тип ***STACK*** можно объявить как список символов.
- Процесс вывода содержимого стека в обратном порядке в последней строке программы требует небольшой хитрости.
  - Выталкивание элементов из стека по одному за один раз в принципе позволяет получить последовательность элементов стека в обратном порядке.
  - Некоторые реализации стеков, например с помощью массивов, как описано ниже, позволяют написать простые процедуры для печати содержимого стека, начиная с обратного конца стека.
  - В общем случае необходимо извлекать элементы стека по одному и вставлять их последовательно в другой стек, затем распечатать элементы из второго стека в прямом порядке.

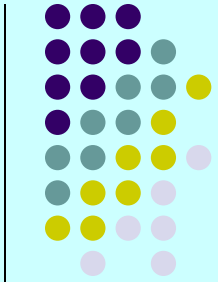
Реализация стеков

Стеки могут представляться в памяти либо в виде вектора, либо в виде цепного списка.

При ***векторном представлении*** под стек отводится сплошная область памяти, достаточно большая, чтобы в ней можно было поместить некоторое максимальное число элементов, которое определяется решаемой задачей.

Граничные адреса этой области являются параметрами физической структуры стека - вектора. В процессе заполнения стека место последнего элемента (его адрес) помешается в указатель вершины стека.

Реализация стеков



Если указатель выйдет за верхнюю границу стека, то стек считается переполненным и включение нового элемента становится невозможным.

Поэтому для стека надо отводить достаточно большую память, однако если стек в процессе решения задачи заполняется только частично, то память используется неэффективно.

Так как под стек отводится фиксированный объем памяти, а количество элементов переменно, то говорят, что стек в векторной памяти - это ***полустатическая структура данных***.

Обычно в стеке элементы имеют один и тот же тип, поэтому обработка такого стека достаточно проста.

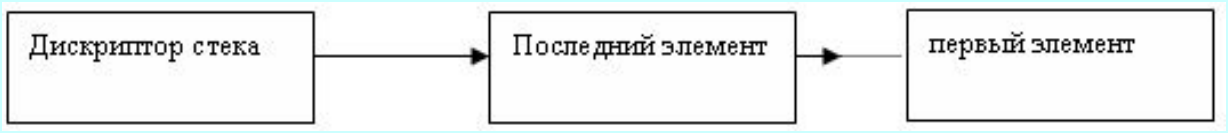
Реализация стеков

Многие современные ЭВМ содержат в своей конструкций аппаратные стеки или средства работы со стеками.

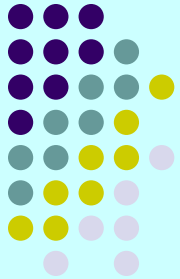
Однако даже в этом случае при разработке программ часто приходится использовать свои программные стеки.

Стек, представленный как вектор, имеет вид:

Структура стека в оперативной памяти:



Списковая структура стека

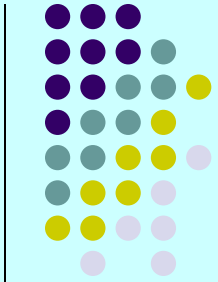


При списковом представлении стека память под дескриптор и под каждый элемент стека получают динамически.

Включение и выборка элемента осуществляются с начала списка, которое одновременно является вершиной стека.

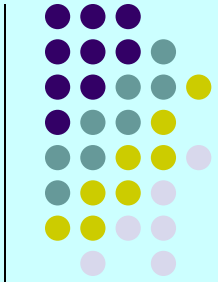
Переполнение стека в этом случае не происходит, однако алгоритмы обработки сложнее, а время обработки удлиняется, так как операции включения и выборки элементов сопряжены с обращением к операционной системе для получения или освобождения памяти.

Реализация стеков с помощью массивов



- Каждую реализацию списков можно рассматривать как реализацию стеков, т.к. стеки с их операторами являются частными случаями списков с операторами, выполняемыми над списками. Надо просто представить стек в виде однонаправленного списка, так как в этом случае операторы ***PUSH*** и ***POP*** будут работать только с ячейкой заголовка и первой ячейкой списка.
- Фактически заголовок может быть и указателем, а не полноценной ячейкой, поскольку стеки не используют такого понятия, как "позиция", и, следовательно, нет необходимости определять позицию 1 таким же образом, как и другие позиции.

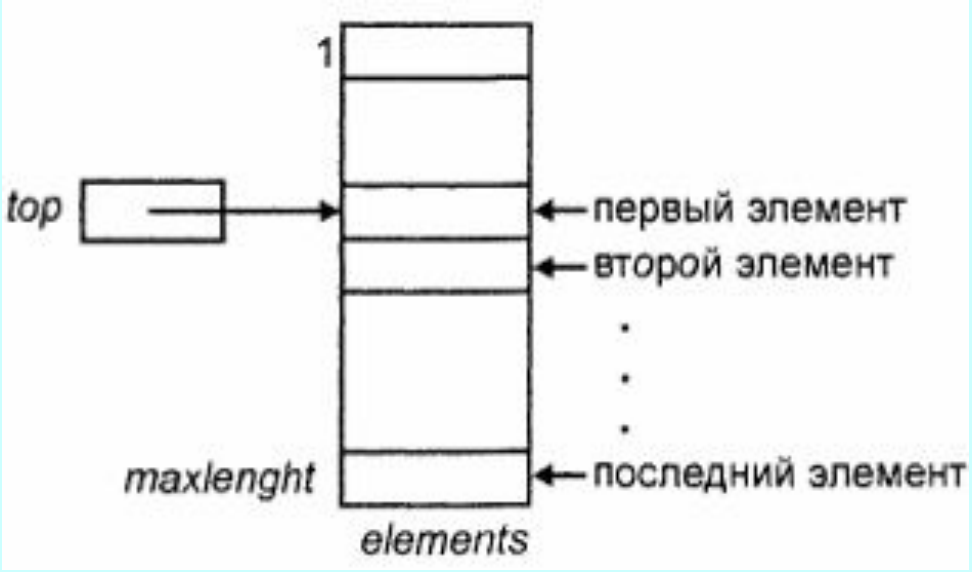
Реализация стеков с помощью массивов



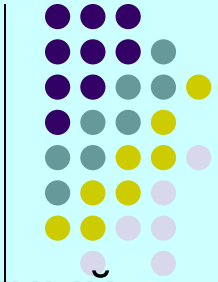
- Однако реализация списков на основе массивов не очень подходит для представления стеков, так как каждое выполнение операторов ***PUSH*** и ***POP*** в этом случае требует перемещения всех элементов стека и поэтому время их выполнения пропорционально числу элементов в стеке.
- Можно более рационально приспособить массивы для реализации стеков, если принять во внимание тот факт, что вставка и удаление элементов стека происходит только через вершину стека.
  - Можно зафиксировать "дно" стека в самом низу массива (в ячейке с наибольшим индексом) и позволить стеку расти вверх массива (к ячейке с наименьшим индексом).
  - Курсор с именем ***top*** (вершина) будет указывать положение текущей позиции первого элемента стека.

Схема реализации стека с помощью массива

- Можно зафиксировать "дно" стека в самом низу массива (в ячейке с наибольшим индексом) и позволить стеку расти вверх массива (к ячейке с наименьшим индексом).
- Курсор с именем ***top*** (вершина) будет указывать положение текущей позиции первого элемента стека.



Реализация стеков с помощью массивов



- Для такой реализации стеков можно определить абстрактный тип STACK следующим образом:

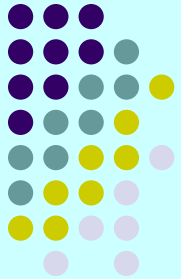
```
type  STACK = record
            top: integer;
            element: array[1..maxlength] of elementtype
end;
```

- В этой реализации стек состоит из последовательности элементов

  element[top], ***element[top + 1]***, ***...***, ***element[maxlength]***.

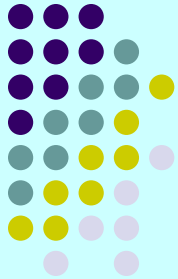
- Если ***top = maxlength+1***, то стек пустой.

Листинг 3.6. Реализация операторов, выполняемых над стеками



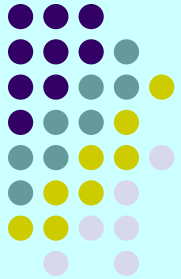
```
procedure  MAKENULL ( var S: STACK );
begin
     S.top:= maxlength + 1
end;          { MAKENULL }

function EMPTY ( S: STACK ): boolean;
begin
     if   S. top > maxlength  then
                EMPTY:=true
     else   EMPTY:=false
end;          { EMPTY }
```

Листинг 3.6. Реализация операторов, выполняемых над стеками



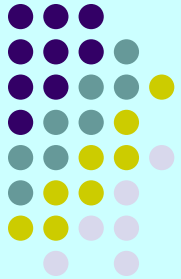
```
function  TOP ( var S: STACK ): elementtype;
begin
     if   EMPTY(S)   then   error('Стек пустой1)
     else  TOP:= S.elements[S.top]
end;          { TOP }

procedure   POP ( var S: STACK );
begin
     if   EMPTY(S)   then  error('Стек пустой')
     else  S.top:= S.top + 1
end;          { POP }
```

Листинг 3.6. Реализация операторов, выполняемых над стеками



```
procedure  PUSH ( x: elementtype; var S: STACK );
begin
     if   S.top = 1 then   error('Стек полон')
     else   begin
          S.top:= S.top - 1
          S.elements[S.top]:= x
     end
end;          { PUSH }
```

Применение стеков при разработке приложений



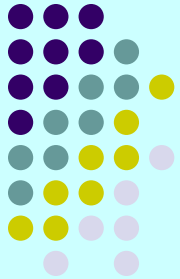
Одно из применений стеков можно продемонстрировать на примере вычисления значения арифметического выражения в калькуляторах.

Пусть арифметическое выражение составлено из комбинации

- чисел,
- знаков бинарных арифметических операций (операторов) +, -, *, /, ^,
- круглых скобок (,)
- и пробелов.

Алгоритм вычисления предусматривает представление выражения в определенном формате.

Применение стеков при разработке приложений



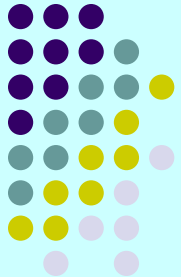
Различают представление выражения в инфиксной и постфиксной формах. В обеих формах выражения представляются в виде символьных строк.

В ***инфиксной форме*** записи каждый бинарный оператор помещается между двумя своими операндами. Для уточнения порядка вычислений могут использоваться круглые скобки. Инфиксный формат записи используется в большинстве языков программирования и калькуляторах и практически совпадает с естественной формой записи выражения.

Примеры записи выражений:

- 5.7+6.8=
- 15*4+(25/2-3)^2=
- 3*7.5+6e2/5=

Применение стеков при разработке приложений

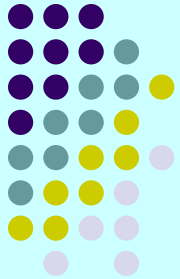


В ***постфиксной форме*** записи (обратной польской записи ОПЗ, или Reverse Polish Notation RPN) операнды предшествуют своему оператору.

Примеры записи выражений:

- 5.7 6.8 + =
- 15 4*25 2/3-^ + =
- 3 7.5 * 6e2 5/+=

Применение стеков при разработке приложений

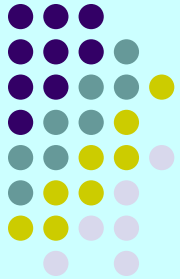


Постфиксный калькулятор.

Наиболее простым является алгоритм вычисления постфиксного выражения. Исходная строка содержит элементы только двух видов: числа и операторы. Пусть выражение заканчивается символом '='.

Алгоритм использует один стек, элементами которого являются числа вещественного типа.

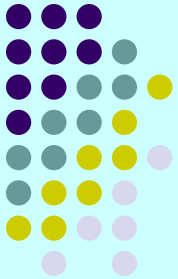
Применение стеков при разработке приложений



Алгоритм вычисления постфиксного калькулятора.

1. Из исходной строки выделяется очередной элемент.
2. Если элемент - число, то оно заносится в стек. Переход к п.1.
3. Если элемент - оператор, то из стека последовательно извлекаются два элемента, сначала правый операнд, затем левый операнд и над ними выполняется операция, определенная оператором. Результат операции (число) заносится в стек. Переход к п.1.
4. Пункты 1 - 3 выполняются до тех пор, пока в исходной строке не встретится признак конца выражения '='. В этом случае число, находящееся в стеке, является результатом вычисления.

Применение стеков при разработке приложений



Рассмотрим порядок вычисления выражения

3 7.5 * 6e2 5/ + =

Шаг 1. Из строки выделяется число 3 и помещается в стек. Стек: 3.

Шаг 2. Из строки выделяется число 7.5 и помещается в стек. Стек: 3 7.5.

Шаг 3. Из строки выделяется оператор '*'. Из стека извлекаются числа 7.5 и 3. Выполняется операция 3*7.5, результат 22.5 помещается в стек. Стек: 22.5.

Шаг 4. Из строки выделяется число 6e2 и помещается в стек. Стек: 22.5 600.

Шаг 5. Из строки выделяется число 5 и помещается в стек. Стек: 22.5 600 5.

Шаг 6. Из строки выделяется оператор '/'. Из стека извлекаются два числа 5 и 600. Выполняется операция 600/5, и результат 120 помещается в стек. Стек: 22.5 120.

Шаг 7. Из строки выделяется оператор '+'. Из стека извлекаются два числа 120 и 22.5. Выполняется операция 22.5+120. Результат 142.5 засылается в стек.

Шаг 8. Из строки выделяется символ '=' - признак конца выражения. Из стека извлекается результат вычисления - число 142.5.

Применение стеков при разработке приложений

Преобразование выражения из инфиксной формы в постфиксную.

Алгоритм преобразования основан на ***методе стека с приоритетами***. В нем всем операторам и скобкам-разделителям ставятся в соответствие целочисленные приоритеты. Чем старше операция, тем выше ее приоритет. Открывающая скобка имеет низший приоритет, равный 0, закрывающая скобка - равный 1.

В ходе обработки исходной строки операнды переносятся в выходную строку непосредственно, а операторы - через стек в соответствии со своими приоритетами.

Элемент стека состоит из двух полей:

- оператор или скобка - символьный тип;
- приоритет - целочисленный тип.

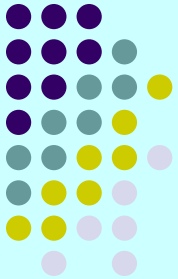
Приоритет пустого стека полагаем равным нулю.

Применение стеков при разработке приложений

Алгоритм метода.

1. Из исходной строки выделяется очередной элемент S_i.
2. Если S_i - операнд, то записать его в выходную строку и перейти к п.1; иначе перейти к п.3.
3. Если приоритет S_i равен нулю (т.е. элемент - открывающая скобка) или больше приоритета элемента S_j, находящегося в вершине стека, то добавить S_i в вершину стека и перейти к п.4; иначе перейти к п.5.
4. Если теперь элемент в вершине стека имеет приоритет, равный 1 (т.е. добавленный элемент S_i является закрывающей скобкой), то из стека удалить два верхних элемента (закрывающую и открывающую скобки); перейти к п.1.
5. Элемент (оператор) из вершины стека вытолкнуть в выходную строку и перейти к п.3.
6. Пункты 1 - 5 выполнять до тех пор, пока не встретится признак конца выражения - символ '='. Тогда все элементы из стека вытолкнуть в выходную строку, затем занести туда символ “=”.

Применение стеков при разработке приложений



Рассмотрим порядок преобразования выражения

15*4+(25/2-3)^2=.

Шаг 1. Выделен операнд 15, заносим его в выходную строку. Выходная строка: 15, стек пуст.

Шаг 2. Выделен оператор '*', его приоритет больше приоритета пустого стека, помещаем в стек. Стек: *, выходная строка: 15.

Шаг 3. Выделен операнд 4, его в выходную строку. Выходная строка: 15 4, стек: *.

Шаг 4. Выделен оператор '+', его приоритет меньше приоритета '*' в вершине стека, поэтому '*' выталкиваем в выходную строку, а '+' заносим в стек. Выходная строка: 15 4*, стек: +.

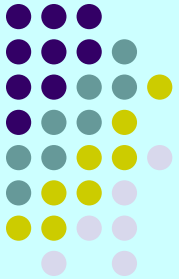
Шаг 5. Выделена открывающая скобка с нулевым приоритетом, помещаем его в стек. Выходная строка: 15 4*, стек: + (.

Шаг 6. Выделен операнд 25, помещаем в выходную строку. Выходная строка: 15 4 * 25, стек: + (.

Шаг 7. Выделен оператор “/”, его приоритет выше приоритета '('. помещаем в стек. Строка: 15 4 * 25, стек: + (/.

Шаг 8. Выделен операнд 2, его в строку: 15 4 * 25 2.

Применение стеков при разработке приложений



15*4+(25/2-3)^2=.

Шаг 9. Выделен оператор '-', его приоритет меньше приоритета “/” из вершины стека, поэтому ”/” - в строку, теперь приоритет '-' больше приоритета '(' и '-' заносим в стек. Выходная строка 15 4 * 25 2 /, стек: + (-.

Шаг 10. Выделен операнд 3, его в строку: 15 4 * 25 2 / 3.

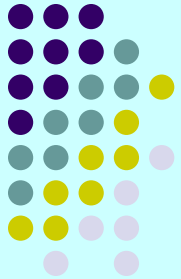
Шаг 11. Выделена закрывающая скобка ')', ее приоритет меньше приоритета '-' из стека, поэтому '-' - в строку. Теперь приоритет ')' больше приоритета '(‘, помещаем ')' в стек, но так как его приоритет равен 1, то из стека удаляем два элемента: ')' и '(' без занесения их в выходную строку). Выходная строка: 15 4 * 25 2 / 3 -, стек: +.

Шаг 12. Выделен оператор '^', его приоритет больше приоритета '+', '^' засылаем в стек: + ^, строка без изменения.

Шаг 13. Выделен операнд 2, его - в строку. 15 4 * 25 2 / 3 - 2.

Шаг 14. Выделен признак конца выражения '=', из стека выталкиваем в строку '^' и '+', затем в строку заносим '='. Выходная строка сформирована полностью: 15 4 * 25 2 / 3 – 2 ^ + =, стек пуст.

Применение стеков при разработке приложений



Инфиксный калькулятор.

Очевидно, что, используя рассмотренные выше алгоритмы преобразования инфиксного выражения в постфиксное и вычисления постфиксного выражения, легко создать инфиксный калькулятор.

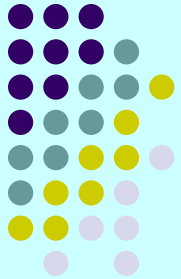
Его алгоритм будет основан на применении двух стеков:

- стека операторов в алгоритме преобразования
- и стека операндов в алгоритме вычисления.

Сам алгоритм будет состоять из двух самостоятельных частей, выполняемых последовательно.

Первая часть преобразует инфиксную строку в постфиксную, которая является входом для второй части, выполняющей вычисление постфиксного выражения.

Применение стеков при разработке приложений



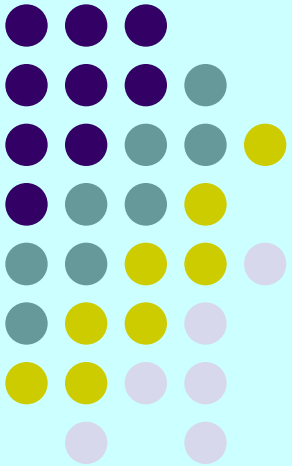
Инфиксный калькулятор.

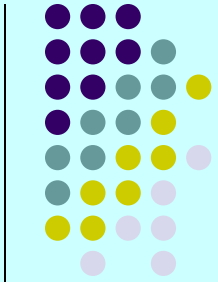
Более подходящим является алгоритм, в котором рассмотренные выше алгоритмы выполняются не последовательно, а совместно.

По мере того как часть инфиксной строки преобразуется в постфиксную, осуществляется вычисление преобразованной части выражения.

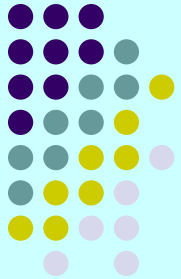
Такой подход облегчает проверку правильности исходного выражения и позволяет прекратить его обработку при выявлении ошибок.

Абстрактный тип данных «Очередь»



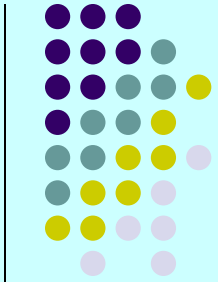


- Другой специальный тип списка - ***очередь*** (queue), где элементы вставляются с одного конца, называемого ***задним*** (rear), а удаляются с другого, ***переднего*** (front).
- Очереди также называют "***списками типа FIFO***" (аббревиатура FIFO расшифровывается как first-in-first-out: первым вошел - первым вышел). Такая очередь является простой очередью без приоритетов.
- Часто используются очереди с приоритетами, в них более приоритетные элементы включаются ближе к голове очереди, выборка осуществляется, как обычно, с головы очереди.
- Операторы, выполняемые над очередями, аналогичны операторам стеков. Существенное отличие между ними состоит в том, что вставка новых элементов осуществляется в конец списка, а не в начало, как в стеках.
- Кроме того, различна устоявшаяся терминология для стеков и очередей.



Очереди находят широкое применение в операционных системах (очереди задач, буфера ввода-вывода, буфер ввода с клавиатуры, очереди в сетях ЭВМ и т.п.), при моделировании реальных процессов и т.д.

Операторы, выполняемые над очередью



1. ***MAKENULL(Q).***

   Делает очередь ***Q*** пустой.

2. ***FRONT(Q).***

   Функция, возвращающая первый элемент очереди ***Q***.

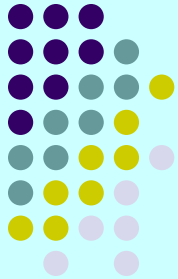
   Можно реализовать эту функцию с помощью операторов списка как ***RETRIEVE(FIRST(Q), Q)***.

3. ***ENQUEUED(x, Q).***

   Вставляет элемент ***x*** в конец очереди ***Q***.

   С помощью операторов списка этот оператор можно выполнить следующим образом: ***INSERT(x, END(Q), Q)***.

Операторы, выполняемые над очередью



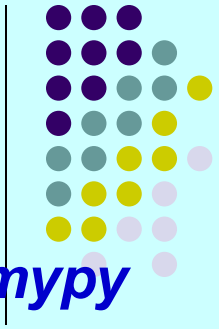
4. ***DEQUEUE(Q).***

   Удаляет первый элемент очереди ***Q***.

   Также реализуем с помощью операторов списка как ***DELETE(FIRST(Q), Q)***.

5. ***EMPTY(Q).***

   Возвращает значение ***true*** тогда и только тогда, когда ***Q*** является пустой очередью.

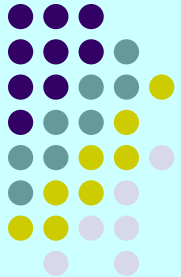


Очереди могут иметь ***векторную*** или ***списковую структуру хранения***, в свою очередь векторная структура может занимать статическую либо динамическую память.

Очередь ***векторной структуры*** из-за ограниченности элементов имеет свойство переполнения, когда хвост очереди достигнет конца вектора. В этом случае добавление элементов становится невозможным, даже если в начальной части вектора будут свободные элементы из-под выбранных элементов.

Для устранения такого недостатка образуют ***кольцевые очереди***. При достижении конца вектора новые элементы добавляются в свободные элементы с начала вектора. Здесь также возможно переполнение, когда хвост догонит голову. Если же голова догонит хвост, то очередь оказывается пустой.

Реализация очередей с помощью указателей



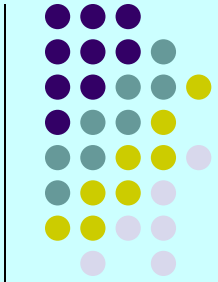
Как и для стеков, любая реализация списков допустима для представления очередей. Однако учитывая особенность очереди (вставка новых элементов только с одного, заднего, конца), можно реализовать оператор ***ENQUEUE*** более эффективно, чем при обычном представлении списков.

Вместо перемещения списка от начала к концу каждый раз при пополнении очереди мы можем хранить ***указатель на последний элемент очереди***.

Как и в случае со стеками, можно хранить ***указатель на начало списка*** - для очередей этот указатель будет полезен при выполнении команд ***FRONT*** и ***DEQUEUE***.

В языке Паскаль в качестве заголовка можно использовать динамическую переменную и поместить в нее указатель на начало очереди. Это позволяет удобно организовать очищение очереди.

Реализация очередей с помощью указателей



Объявление ячеек выполняется следующим образом:

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

Теперь можно определить список, содержащий указатели на начало и конец очереди.

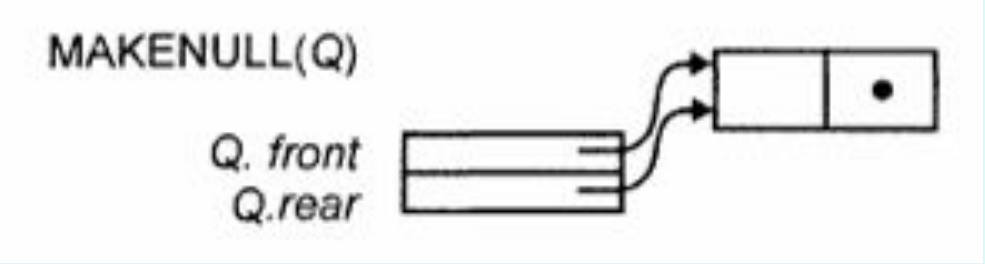
Первой ячейкой очереди является ячейка заголовка, в которой поле ***element*** игнорируется. Это позволяет упростить представление для любой очереди.

Мы определяем АТД ***QUEUE*** (Очередь)

```
type   QUEUE = record
           front, rear:  ^ celltype
       end;
```

Листинг 3.7. Реализация операторов, выполняемых над очередью

```
procedure   MAKENULL (var Q: QUEUE );
begin
      new(Q.front);       { создание ячейки заголовка }
      Q.front^.next:= nil;
      Q.rear:= Q.front
end;          { MAKENULL }
```

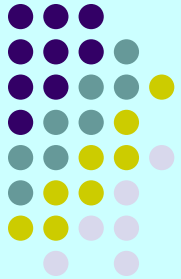


Первый оператор определяет динамическую переменную (ячейку) типа ***celltype*** и назначает ей адрес ***Q.front***.

Второй оператор задает значение поля ***next*** этой ячейки как ***nil***.

Третий оператор делает заголовок для первой и последней ячеек очереди.

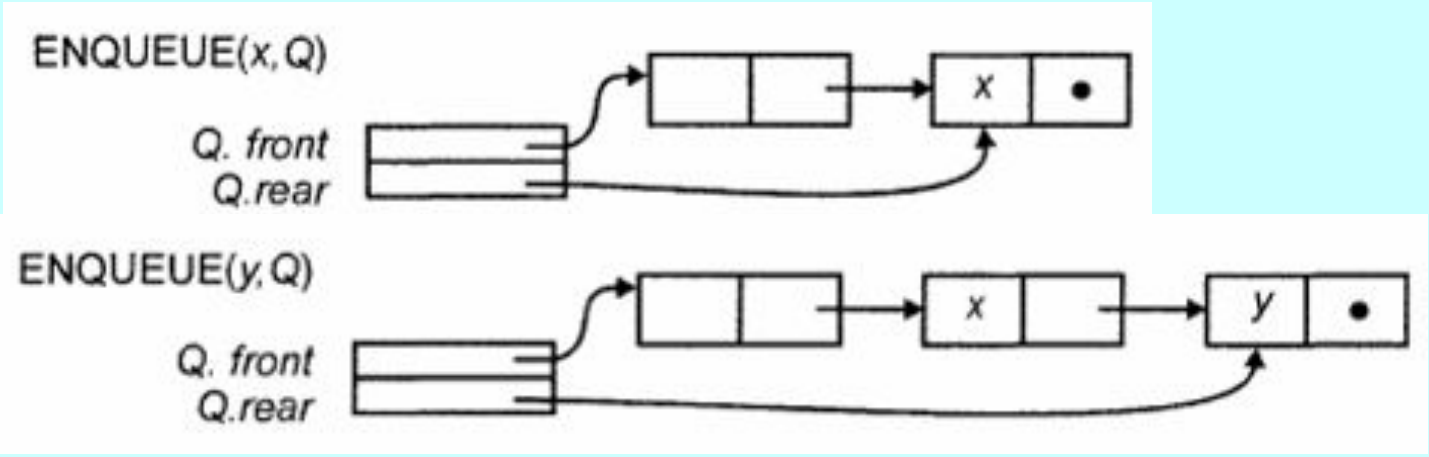
Листинг 3.7. Реализация операторов, выполняемых над очередью



```
function   EMPTY ( Q: QUEUE ): boolean;
begin
     if   Q.front = Q.rear  then EMPTY:= false
     else EMPTY:= true
end;          { EMPTY }

function   FRONT ( Q: QUEUE ): elementtype;
begin
     if   EMPTY(Q)  then  error('Очередь пуста')
     else
          FRONT:= Q.front^. Next^. element
end;   { FRONT }
```

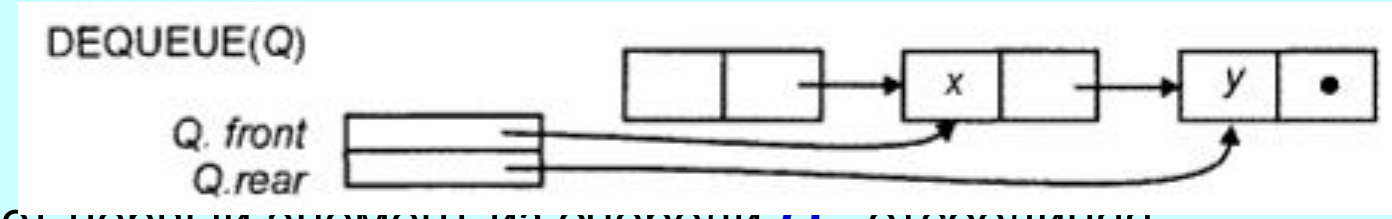
Листинг 3.7. Реализация операторов, выполняемых над очередью

```
procedure   ENQUEUE ( x: elementtype; var Q: QUEUE );
begin
      new(Q.rear^.next);
      Q.rear:= Q.rear^. next;
      Q.rear^.element:= x;
      Q.rear^.next:= nil
end;          { ENQUEUE }
```



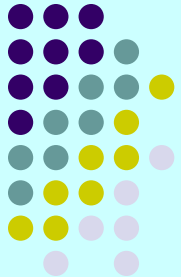
Листинг 3.7. Реализация операторов, выполняемых над очередью

```
procedure   DEQUEUE ( var Q: QUEUE );
begin
     if   EMPTY(Q)   then
          error('Очередь пуста')
     else
          Q.front:= Q.front^.next
end;          { DEQUEUE }
```



Процедура удаляет первый элемент из очереди Q, отсоединяя старый заголовок от очереди. Первым элементом списка становится новая динамическая переменная ячейки заголовка.

Реализация очередей с помощью циклических массивов



Реализацию списков посредством массивов, которая рассматривалась выше, можно применить для очередей, но в данном случае это не рационально.

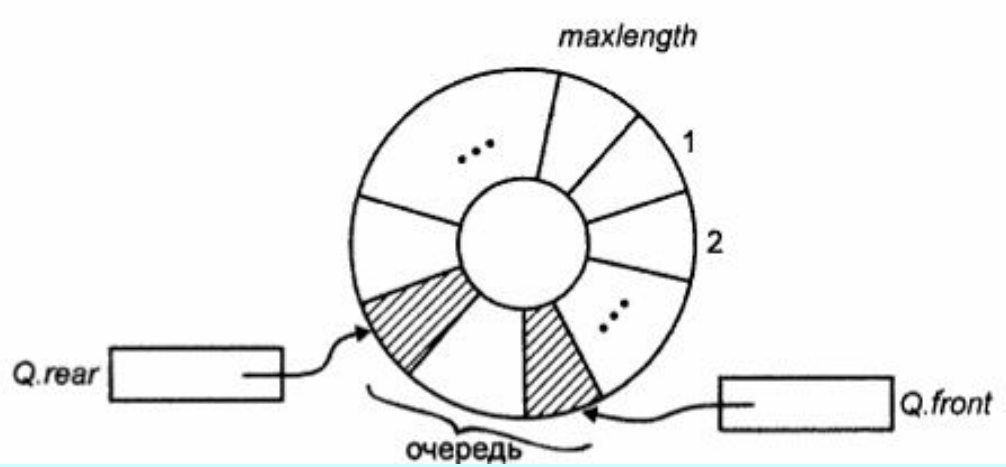
Действительно, с помощью указателя на последний элемент очереди можно выполнить оператор ***DEQUEUE*** за фиксированное число шагов (независимое от длины очереди).

Но оператор ***ENQUEUE***, который удаляет первый элемент, требует перемещения всех элементов очереди на одну позицию в массиве. Таким образом, ***ENQUEUE*** имеет время выполнения ***O(n)***, где ***n*** - длина очереди.

Чтобы избежать этих вычислительных затрат, воспользуемся другим подходом.

Реализация очередей с помощью циклических массивов

Представим массив в виде циклической структуры, где первая ячейка массива следует за последней:



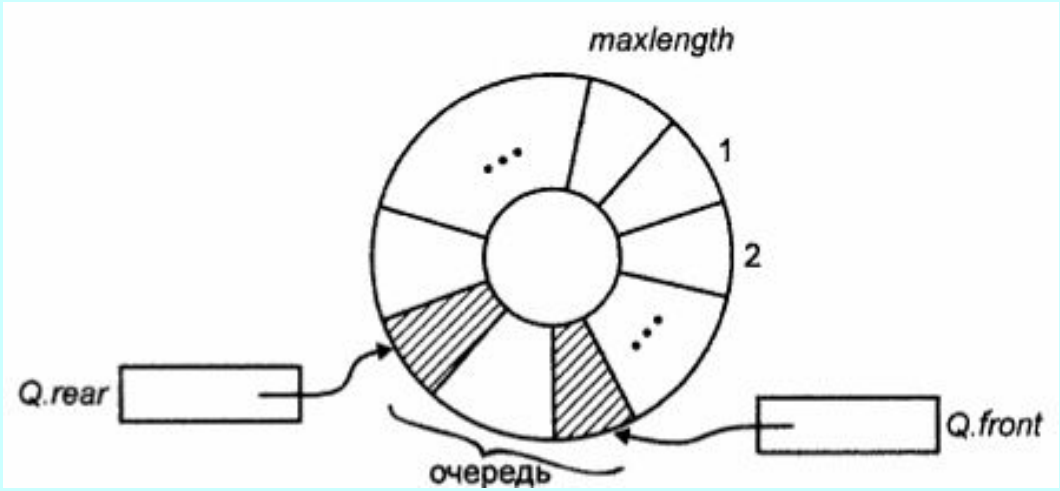
Элементы очереди располагаются в "круге" ячеек в последовательных позициях, конец очереди находится по часовой стрелке на определенном расстоянии от начала.

Реализация очередей с помощью циклических массивов

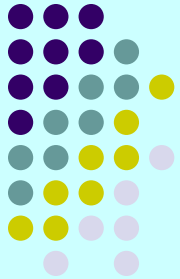
Для ***вставки нового элемента*** в очередь достаточно переместить указатель ***Q.rear*** (указатель на конец очереди) на одну позицию по часовой стрелке и записать элемент в эту позицию.

При ***удалении элемента*** из очереди надо просто переместить указатель ***Q.front*** (указатель на начало очереди) по часовой стрелке на одну позицию.

При таком представлении очереди операторы ***ENQUEUE*** и ***DEQUEUE*** выполняются за фиксированное время, независимое от длины очереди.



Реализация очередей с помощью циклических массивов

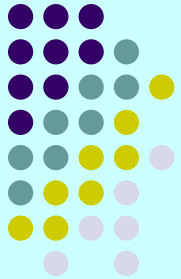


Есть одна сложность представления очередей с помощью циклических массивов и в любых вариациях этого представления (например, когда указатель ***Q.rear*** указывает по часовой стрелке на позицию, следующую за последним элементом, а не на сам последний элемент).

Проблема заключается в том, что только по формальному признаку взаимного расположения указателей ***Q.rear*** и ***Q.front*** нельзя сказать, когда очередь пуста, а когда заполнила весь массив.

Конечно, можно ввести специальную переменную, которая будет принимать значение ***true*** тогда и только тогда, когда очередь пуста, но если мы не собираемся вводить такую переменную, то необходимо предусмотреть иные средства, предотвращающие переполнение массива.

Листинг 3.8. Реализация очереди циклическим массивом

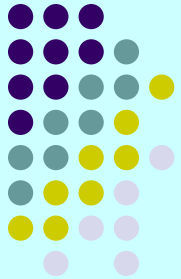


Формально очереди здесь определяются следующим образом:

```
type   QUEUE = record
            elements: array[1..maxlength] of elementtype;
            front, rear: integer
          end;


function   addone ( i: integer ): integer;
begin
     addone:= (i mod maxlength) + 1
end;         { addone }
```

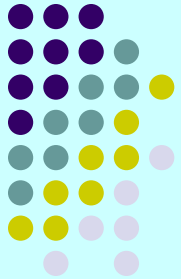
Функция ***addone(i)*** добавляет единицу к позиции ***i*** в "циклическом" смысле.

Листинг 3.8. Реализация очереди циклическим массивом



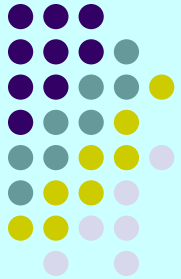
```
procedure   MAKENULL ( var Q: QUEUE );
begin
      Q.front:= 1;
      Q.rear:= maxlength
end;          { MAKENULL }

Function   EMPTY ( var Q: QUEUE ): boolean;
begin
      if   addone(Q.rear) = Q.front   then
            EMPTY:= true
      else   EMPTY:=false
end;          { EMPTY }
```

Листинг 3.8. Реализация очереди циклическим массивом



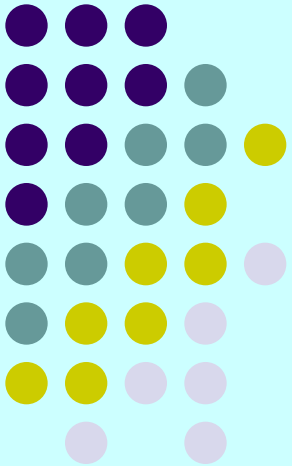
```
function   FRONT ( var Q: QUEUE ): elementtype;
begin
     if   EMPTY(Q)   then  error('Очередь пустая')
     else    FRONT:= Q.elements[Q.front]
end;          { FRONT }

procedure   ENQUEUE ( x: elementtype; var Q: QUEUE );
begin
     if  addone(addone(Q.rear)) = Q.front  then
                error('Очередь полная')
     else   begin
          Q.rear:= addone(Q.rear);
          Q.elements[Q.rear]:= x
end
end;          { ENQUEUE }
```

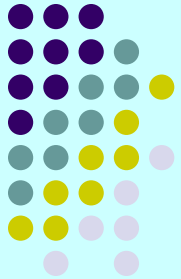
Листинг 3.8. Реализация очереди циклическим массивом



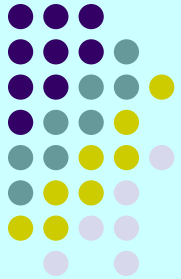
```
procedure DEQUEUE ( var Q: QUEUE );
begin
     if   EMPTY(Q)  then  error('Очередь пустая')
     else
          Q.front:= addone(Q. front)
end;          { DEQUEUE }
```

Абстрактный тип данных «Дек»





- ***Дек*** - это разновидность очереди, в которой включение и выборка элементов возможны с обоих концов.
- Например, может использоваться при управлении памятью, когда распределение памяти производится и сверху, и снизу.

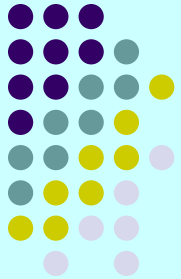


В свою очередь, существуют разновидности дека:

- дек с ограниченным входом
- и дек с ограниченный выходом.

Дек с ограниченным входом допускает включение элементов только на одном конце.

А ***дек с ограниченным выходом*** допускает выборку элементов только с одного конца.



Деки могут иметь как ***векторную***, так и ***списковую структуру хранения***.

Операции над деками такие же, как и над очередями.

При векторном способе хранения программная реализация операций достаточна сложна, она упрощается при представлении ***очереди в виде двунаправленного списка***.