Задание. Реализовать линейный список и операции над ним, используя последовательное распределение.

Линейные списки

Наиболее общим представителем линейной структуры является АТД (абстрактный тип данных) линейный список (linealist).

Списки являются чрезвычайно гибкой структурой, т.к. их легко сделать большими или меньшими, 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.

В математике список представляет собой последовательность элементов $a_1, a_2, \dots, a_N$ $(N \geq 0)$, все элементы которого имеют один базовый тип (elemtype). Количество элементов N называется длиной списка. Если $N \geq 1$, то a_1 называется первым элементом списка, a_N – последним элементом, а у каждого внутреннего элемента a_i $(1 < i < N)$ есть только один предыдущий a_{i-1} и только один следующий a_{i+1}.

Если $N = 0$, то список называется пустым. Говорят также, что элемент a_i имеет позицию i.

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

Для формирования АТД нужно задать множество операций, выполняемых над линейным списком, которые могут быть следующими:

1. создание пустого списка,
2. вставка нового элемента в позицию p или после позиции p,

3. удаление элемента в позиции p или после позиции p,
4. получение доступа к элементу в позиции p для проверки и/или изменения,
5. переход к следующему или предыдущему элементу.

Отметим, что трудно сформулировать все необходимые операции над списком, которые потребуются при разработке алгоритма для произвольного приложения.

Названные операции относятся к наиболее часто используемым, дополнительно можно задать, например, операцию поиска элемента с заданным значением, операцию упорядочения элементов списка и т.д.

Более того, достаточно сложно реализовать выполнение всех операций с одинаковой эффективностью. Например, сравнительно непросто организовать быстрый доступ к элементу списка, если одновременно нужно выполнять и быструю вставку элемента в произвольной позиции.

Поэтому различают некоторые типы линейных списков в зависимости от выполняемых операций.

Реализация списков

Существуют различные способы реализации списка.

Мы рассмотрим реализацию списков с помощью массивов и с помощью указателей.

Реализация списков посредством массивов

Достаточно простой и естественной представляется реализация линейного списка посредством массивов.

Простейшим представлением множества элементов

$$X = \{x_1, x_2, \dots, x_n\}$$

является точный список элементов, расположенных в последовательных ячейках памяти, т.е. множество представляется с помощью массива и требует для своего хранения непрерывную область памяти. Такое представление будем называть последовательным распределением.

В этом случае используется последовательное распределение памяти, все элементы списка хранятся в смежных ячейках, к которым осуществляется эффективный прямой доступ (direct access).

Однако следует четко представлять себе как преимущества, так и недостатки такой реализации.

В этой реализации элементы массива располагаются по порядку в компонентах массива, начиная с первой; i-тый элемент списка хранится в i-той компоненте.

Размер массива определяет максимально возможную длину списка.

В программе размер массива зададим константой $\boldsymbol{maxlen}$.

Для работы со списком требуется еще знать его текущую длину (или номер компоненты массива, в которой находится текущий последний элемент).

Опишем список **list** (список) как запись, имеющую два поля: первое с именем **elem** – собственно массив, второе с именем **last** – позиция последнего элемента списка.

Дадим также описание переменной L типа list:

```
const
    maxlen=…; {подходящее число для конкретной задачи}
type
    elemtype = ….;  {подходящий для задачи тип элементов}
    list = record
```

```
        elem: array[1..maxlen] of elemtype;
        last: integer
    end;
var
    L:list;
```

Выражение L соответствует понятию «список». Конструкция L.elem[i] обозначает в программе i-ый элемент списка.

1	2	3	…	…	N				
1,2,3 элементы списка					N элемент списка	свободно			maxlen

Пусть в качестве elemtype задан тип integer. Реализация заданного списка [3,5,1,10,12] будет выглядеть так (считаем maxlen=10):

L	3	4
	5	
	1	
	10	
	12	

Начнем с функции End(L). Эта функция будет возвращать позицию, следующую за позицией N в N-элементном списке L.

Отметим, что позиция End(L), рассматриваемая как расстояние от начала списка, может меняться при увеличении (уменьшении) списка, в то время как другие позиции имеют фиксированное (неизменное) расстояние

от начала списка.

```
function EndL(var L:list):integer;
begin
   EndL:=L.N+1;
end;
```

Вернемся к нашему списку.

Создать такой список можно следующими действиями: сделать список L пустым, затем поочередно вставлять в него элементы в нужном порядке.

Сделать список пустым весьма просто – достаточно обнулить поле last, поскольку список пуст, если его длина равна нулю. Оформим это действие в виде процедуры ListInit(L):

```
procedure ListInit (var L:list)
begin   {процедура создания пустого списка}
   L.last:=0;
end;
```

Вставка нового элемента x в позицию i списка L.

Эта процедура осуществляется в два этапа:

1. все значения из компонент массива elem с индексами $i, i+1, \dots, last$ перемещаются соответственно в компоненты с индексами $i+1, i+2, \dots, last+1$,
2. в i-тую компоненту помещается элемент x, и значение поля last увеличивается на единицу.

Бывшие 3,4 и 5 элементы теперь зани-	L	3	5		L	3	6
		5				5	
						x=7	
		1				1	

мают 4,5,6 позиции. Третья компонента освобождается для записи нового значения		10 12		10 12
После 1-го этапа			После 2-го этапа	

Оформим операцию вставки в виде процедуры Insert(L,x,i).

```
procedure Insert (var L:list; x:elemtype;i:integer)
var
   q:integer;
begin
    for q:=L.last downto i do
{перемещение компонент i, i+1, ..,last на одну к концу массива}
            L.elem[q+1]:= L.elem[q];
    L.elem[i]:=x;
    L.last:=L.last+1;
end;
```

Итак, чтобы вставить значение x=7 перед третьим элементом списка L, в программе следует использовать оператор Insert(L,7,3).

Процедуру Insert можно использовать и для вставки элемента в пустой список, задав единицу в качестве третьего параметра.

Например, ListInit(L); Insert(L,12,1);

Теперь запишем последовательность операторов, после выполнения которой мы получим заданный список:

Insert(L,12,1); Insert(L,10,1); Insert(L,1,1); Insert(L,5,1); Insert(L,3,1);

Insert(L,7,3).

Для удаления i-го элемента опишем процедуру Delete(L,i).

```
procedure Delete (var L:list; i:integer)
var
   p:integer;
begin
    for p:=i to L.last-1 do
{перемещение значений из компонент i+1, i+2, ..,last на компо-
ненту к началу массива}
            L.elem[p]:= L.elem[p+1];
    L.last:=L.last-1;
end;
```

При использовании процедур Insert и Delete следует проявлять осторожность: нельзя вставлять элемент, если длина списка максимальна, т.е. равна maxlen, нельзя удалять элемент из пустого списка, необходимо правильно задавать номер i элемента в списке: $1 \leq i \leq last$.

Проверку условий для корректной работы со списками реализуем в виде логических функций:

```
function is_full (var L:list):boolean;
{возвращает истину, если длина списка максимальна, иначе - ложь}
begin
   is_full:=L.last=maxlen;
end;
```

```
function is_empty (var L:list):boolean;
{возвращает истину, если список пуст, иначе - ложь}
begin
   is_empty:=L.last=0;
```

```
end;
```

```
function is_valid (var L:list):boolean;
{возвращает истину, если в списке есть i-й элемент, иначе - ложь}
begin
   is_valid:=(1<=i) and (i<=L.last);
end;
```

Опишем также логическую функцию is_present(L,x), проверяющую, есть ли элемент x в списке L.

```
function is_present (var L:list;x:elemtype):boolean;
{возвращает истину, если элемент со значением x присутствует в списке, иначе - ложь}
Var
     found: boolean;
     p: integer;
begin
   found:=false;
   p:=1;
   while not found and (p<=L.last) do
      begin
          found:=x=L.elem[p];
          p:=p+1;
      end;
  is_present:=found;
end;
```

Рассмотрим работу этой функции для разных случаев. Если список L пуст (L.last=0), цикл в теле функции не выполнится ни разу, т.к. условие p<=last ложно, и функция возвращает «ложь». Если элемент со значением

x присутствует в списке, то переменная found после нескольких повторений тела цикла получит значение «истина», и цикл завершится, т.к. условие not found станет ложным. Функция возвращает значение «истина». Наконец, если элемент отсутствует в непустом списке, то значение переменной found в цикле не изменится. Цикл завершится, поскольку на каждой итерации значение p увеличивается, и условие p<=L.last станет ложным. значением функции будет «ложь».

Можно записать и другие операторы списка, используя данную реализацию списков.

Возвращение k-го по счету элемента списка L

```
function Get (var L:list; k:integer): integer;
begin
   Get:=L.elem[k];
end;
```

Изменение k-го по счету элемента списка L

```
procedure Put (var L:list; k:integer; x:integer)
begin
   L.elem[k]:=x;
end;
```

Возвращает позицию элемента x в списке L

```
procedure Locate (L:list; x:integer; var i:integer):integer;
var j,q,k:integer;
begin
    k:=0;
   for q:=1 to L.last do
      if L.elem[q]=x then begin  j:=q; inc(k); {return (q)};
   if k>0 then i:=j else i:=0;
```

```
 //return(l.last+1) {элемент x не найден}
end;
```

Также можно записать следующие операторы:

- функция **first(L**) – всегда возвращет 1;
- функция **next(L,p)** – возвращает значение, на единицу большее аргумента;
- функция **previous(L,p)** – возвращает значение на единицу меньше аргумента;
- функция **prinlist(L)** – печатает элементы списка L в порядке их расположения.
- **sortlist(L)**
- **max_in_list(L)**
- **min_in_list(L).**

Основным достоинством последовательного распределения является возможность прямого доступа к любому элементу множества, поскольку существует простое соотношение между порядковым номером элемента в множестве и адресом ячейки где он хранится.

Кроме того, данное представление легко реализуемо и требует небольших расходов памяти (для статических множеств).

Существенным недостатком последовательного распределения является неэффективность реализации динамических множеств. Этот недостаток является следствием того, что массив по своей сути является статической структурой.

Во-первых, размер массива приходится задавать (резервировать объем памяти) исходя из максимально возможного размера множества независимо от реального размера в конкретный момент времени, что может привести к неэффективному использованию памяти. Кроме того, не

всегда можно заранее определить верхнюю границу мощности множества.

Во-вторых, время выполнения операций включения и исключения зависят от размера множества. Операции вставки и удаления выполняются крайне неэффективно, т.к. требуют большого числа сдвигов элементов, за исключением случая, когда обрабатывается последний элемент.