23. СПИСКИ

Определим тип elem:

```
typedef  int  ETYPE;
struct      elem      {
ETYPE data;
elem *next; };
```

Назовем data информационным элементом. У нас он типа int, но может быть любого сложного необходимого нам типа ETYPE.

Указатель next указывает на объект типа elem. Объекты типа elem можно упорядочить с помощью указателя next (рис. 2).

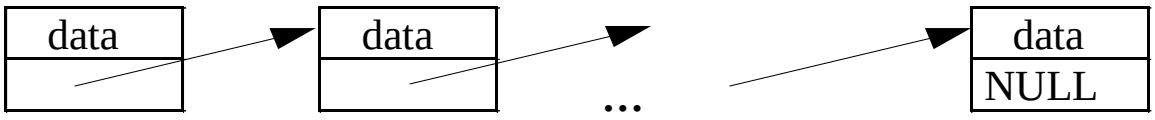


Рис. 2. Структура односвязного списка

Такая структура данных называется однонаправленным, или односвязным, списком, иногда – цепочкой.

Объекты типа elem из этого списка называют элементами списка или звеньями. Каждый элемент цепочки, кроме последнего, содержит указатель на следующий за ним элемент. Признаком того, что элемент является последним в списке, служит то, что член типа elem* этого звена равен NULL.

Вместе с каждым списком рассматривается переменная, значением которой является указатель на первый элемент списка. Если список не имеет ни одного элемента, то есть пуст, значением этой переменной должен быть NULL.

Рассмотрим методы работы с такими списками.

Пусть переменные p, q имеют тип elem*: elem

*p, *q;

Создадим список из двух звеньев, содержащих числа 17 и -9 в информационных элементах.

Значением переменной p всегда будет указатель на первый элемент уже построенной части списка. Переменная q будет использоваться для выделения с помощью new места в динамической памяти под размещение новых элементов списка. Выполнение оператора p = NULL; приводит к созданию пустого списка. После выполнения операторов q

= new elem; q -> data = -9; q -> next = p; p = q;

имеем список, состоящий из одного элемента, содержащего число -9 в информационной части. Переменные p, q указывают на этот элемент (рис. 3, *а*):

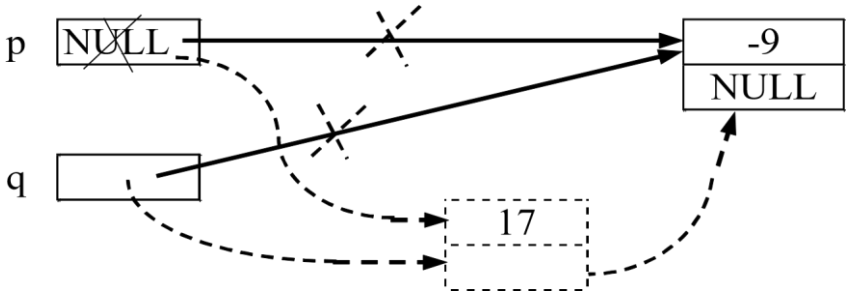


Рис. 3. Создание списка: а – из одного элемента (сплошные линии); б – из двух элементов (пунктир) Далее, выполнение операторов (рис. 3, *б*) q = new elem; q -> data = 17; q -> next = p; p = q;

приводит к тому, что в начало цепочки добавляется новый элемент, содержащий число 17. В результате получится список, изображенный на рис. 4. Значением переменных p и q снова является указатель на первый элемент списка.

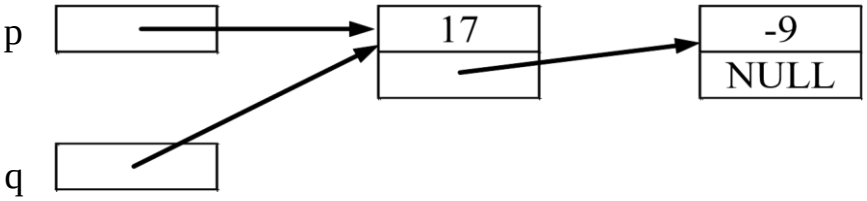


Рис. 4. Список из двух элементов

Фактически мы рассмотрели операцию включения нового элемента в начало, или голову списка, а формирование списка состоит в том, что начинают с пустого списка и последовательно добавляют в начало элементы. Пример 1:

Построим список, элементы которого содержат целые числа:

1, 2, 3, …, n. p

= NULL;

```
while ( n > 0 ){  q
= new elem; q-
>data = n;   q-
>next = p;
p        = q;
n --; }
```

Отметим, что при включении звена в голову списка порядок элементов в списке обратен порядку их включения.

Основная операция при работе со списком – это проход по списку.

Предположим, что с каждым информационным элементом звена нужно выполнить некоторую операцию, которая реализуется функцией void P(ETYPE). Пусть также p указывает на начало списка. Тогда проход по списку осуществляется так:

```
q       = p;
while(q){
P( q -> data ); q
= q -> next;  }
```

Пример 2:

Во входном файле num.txt находится последовательность, содержащая нечетное количество целых чисел.

Напишем программу, в результате выполнения которой выводится число, занимающее в этой последовательности центральную позицию.

```
#include < fstream.h >       // Для работы с файлом ввода.
#include < stdlib.h >
struct elem{ int data;
elem *next;
}; void main(
){
ifstream infile( "num.txt" );
```

/* Создается входной поток с именем infile для чтения данных, разыскивается файл с именем "num.txt"; если такой файл не существует, то конструктор завершает работу аварийно и возвращает для infile нулевое значение. */

```
if( !infile ){
cout << "Ошибка при открытии файла num.txt!\n";  exit(1);
}
elem *p = NULL, *q;
int j = 0;
while ( infile.peek() != EOF ){
```

/* Функция-член peek() класса ifstream возвращает очередной символ из входного потока infile, фактически не извлекая его оттуда. Если встретится конец файла, то будет возвращено значение EOF, то есть -1. */ q = new elem; infile >> q -> data; q -> next = p;   p = q; j++;

```
}
for ( int i = 1; i <= j/2; i++ ) q
= q -> next;
cout << q-> data << "\n";
}
```

23.1. Операции над односвязными списками

Основных операций над списками – три:

1. Проход по списку, или переход от элемента к следующему.

Как мы уже рассмотрели, это осуществляется с помощью присвоения типа q = q -> next;

2. Включение в список.

Пусть q, r – переменные типа elem*.

Предположим, что необходимо включить новый элемент в список после некоторого элемента, на который указывает q. Создадим этот новый элемент с помощью указателя r и занесем в его информационную часть число 19.

Такое включение осуществляется следующими операторами:

r = new elem; r -> data = 19; r -> next = q -> next; q -> next = r;

Проиллюстрируем это на примере (рис. 5).

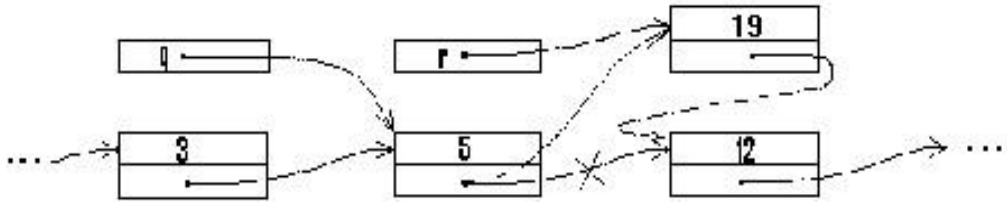


Рис. 5. Включение элемента в список 3.

Исключение из списка.

Пусть теперь значением переменной q является указатель на некоторый, не последний, элемент списка и требуется исключить из списка элемент, следующий за ним. Это можно сделать так:

```
r = q -> next;
q -> next = q-> next ->next; r
-> next = NULL;
```

Второе из приведенных присваиваний – это собственно исключение из списка, а первое выполняется для того, чтобы сохранить указатель на исключенный элемент, т. е. чтобы после исключения из списка он

оставался доступным и с ним можно было бы выполнять некоторые действия. Например, вставить исключенный элемент в другой список или освободить занимаемую им память с помощью операции delete r.

Третье присваивание выполняется для того, чтобы сделать исключение окончательным, т. е. чтобы из исключенного элемента нельзя было бы попасть в список, из которого он исключен.

Реализация списка

Реализуем понятие списка через механизм классов.

```
// Файл "list.cpp"
#include < iostream
> #include < stdlib.h
> typedef int
ETYPE;
struct elem{
ETYPE data; elem*
next;
elem ( ETYPE e, elem* n ){ data = e; next = n;}
}; class list { elem* h;     // Адрес начала списка. public:
list ( ){ h = NULL; }
~list( ){release ( ); }
void create( ETYPE ); // Добавляет элемент в начало списка. void
release( );    // Удаляет список. void insert(elem* q, ETYPE c);  //
Вставляет в список  c  после  q. void del0 ( ){ // Удаляет первый
элемент.
elem* t = h; h = h -> next; delete t;
}
void del( elem* q );         // Удаляет элемент после q. void
print( );        // Распечатывает список. friend class iter; elem*
first( ){ return h; }
};
class iter{  elem* current; public: ite ( list & l )  { current =
l.h; } elem* operator ++ ( );      // Продвижение по списку.
};
void list::create ( ETYPE c ){ h = new elem ( c, h ); }
void list::insert ( elem* q, ETYPE c ){  q->next =
new elem ( c, q -> next );} void list::del ( elem* q ){
if( q->next == NULL ){
cout << "Конец списка! " <<
"Удаление следующего элемента невозможно!\n"; exit (1); }
```

```
elem* r = q -> next; q -> next = q -> next -> next;
r->next = NULL; delete
r;}
elem iter::operator ++ ( ){
/* Возвращает указатель на текущий элемент списка. Осуществляет продвижение по списку. Запоминает новый текущий элемент списка.
*/
if( current ){ elem* tmp = current;
current = current -> next; return
tmp; }
return NULL;
}  void  list::release(
){
iter t( *this ); elem*
p;
while (( p = ++t ) != NULL ){h = h -> next; delete p;}} void
list::print ( ){
iter t ( *this );   elem* p;
while(( p = ++t )!= NULL )
cout << p -> data << "  ";
cout << '\n';}
// Конец файла list.cpp
```

Здесь реализован односвязный список. Это одна из простых моделей структур управления данными. Класс list, реализующий эту модель, – представитель так называемых содержательных, или **контейнерных**, типов. Класс iter создан специально для перебора элементов произвольного списка типа list. Объекты, предназначенные для перебора элементов внутри некоторого набора данных, обычно называют **итераторами**.

Приведем пример использования односвязного списка.

В файле int.txt расположена непустая последовательность целых чисел A_1, A_2, ..., A_n. Определить количество этих чисел n и вывести их в порядке возрастания.

Для решения этой задачи будем строить список, элементы которого упорядочены по возрастанию содержащихся в них целых чисел. Построение выполняется за n шагов. Первый шаг – это создание списка, состоящего из одного элемента, который содержит A_1. Очевидно, этот список упорядочен. На i-м шаге ($i = 2, 3, \ldots, n$) переходим от упорядоченного списка, элементы которого содержат числа A_1, ..., A_{i-1}, к

упорядоченному списку, элементы которого содержат A_1, …, A_{i-1}, A_i. Для выполнения такого перехода достаточно включить в список новый элемент, содержащий A_i. Его надо вставить непосредственно за последним по порядку элементом, содержащим число, меньшее, чем A_i.

Если же все элементы исходного списка содержат числа, не меньшие, чем A_i, то новый элемент нужно вставить в начало списка.

```
#include        "list.cpp"
#include < fstream.h >
void main( ){ ifstream
file( "int.txt" ); list  lst;
int i, n; file >> i;  n = 1;
lst.create( i );
while( file.peek( ) != EOF ){
file >> i;   n++; iter tmp( lst );            // Создаем объект-итератор
для перебора
// элементов списка lst.
elem *p, *q;
while(( p = ++tmp)!= NULL )if( p -> data < i ) q = p; else
break;
if( p == lst.first( )) lst.create( i ); else
lst.insert( q, i );}
cout << "В файле чисел:  " << n << "\n";
cout << "Упорядоченный список:\n";
lst.print ( );
}
```

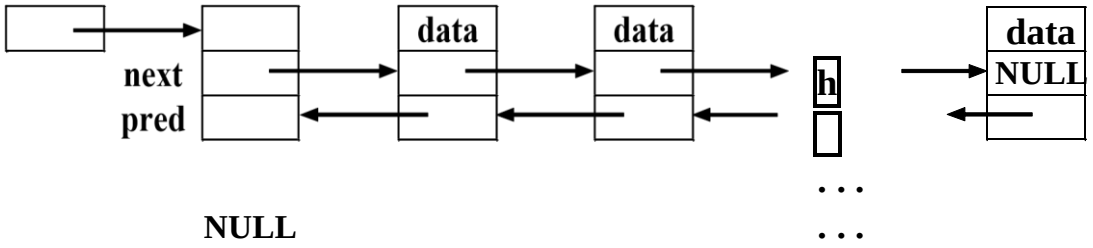
В последнем операторе if – else делается проверка p == lst.first(). Это необходимо из-за того, что механизм вставки звена в начало списка и в список после указателя p различен. Различие возникает из-за того, что у первого элемента нет предыдущего. Иногда для единообразия в начало списка помещают так называемый заглавный элемент, который никогда не удаляют и перед которым никогда не вставляют элемент. Его информационная часть обычно не используется.

23.2. Двунаправленные и кольцевые списки

Чтобы в списках был удобный способ доступа к предыдущим элементам, добавим в каждый элемент списка еще один указатель, значением которого будет адрес предыдущего звена списка:

```
struct elem{ ETYPE
data; elem * next; elem *
pred;
elem ( ETYPE c, elem * n, elem * p ){ data = c; next = n; pred = p; } };
```

С помощью элементов такого типа (рис. 6) можно сформировать так называемый двунаправленный список (с заглавным элементом).



Заглавный элемент

Рис. 6. Двунаправленный список

Здесь, в поле pred заглавного звена, содержится пустой указатель NULL, означающий, что у заглавного элемента нет предыдущего. Часто двунаправленные списки обобщают следующим образом (рис. 7, 8): в качестве значения next последнего звена принимают указатель на заглавное (или первое) звено, а в качестве значения поля pred заглавного (соответственно первого) звена – указатель на последнее звено.

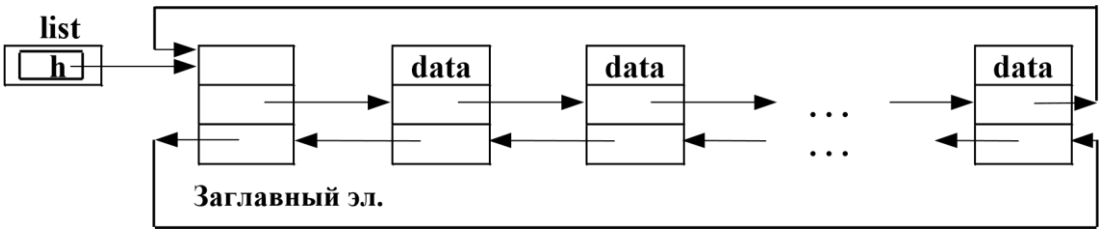


Рис. 7. Первый вариант двунаправленного кольцевого списка

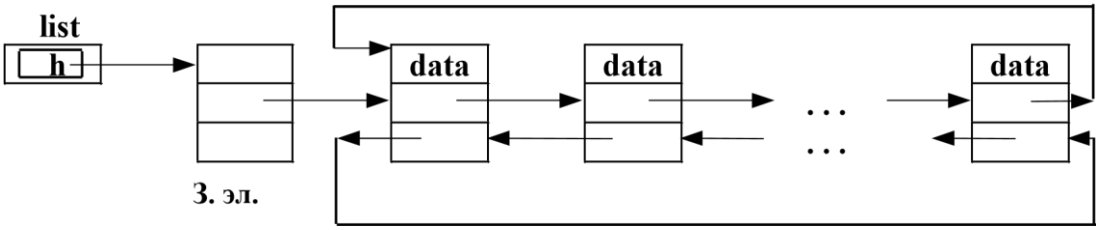


Рис. 8. Второй вариант двунаправленного кольцевого списка

На рис. 7 и 8 списки замыкаются в кольцо и поэтому такие списки называют двунаправленными кольцевыми.

В первом варианте (рис. 7) просто реализуется вставка нового звена как в начало списка (после заглавного звена) так и в его конец, так как

вставка звена в конец списка эквивалентна его вставке перед заглавным элементом. Но здесь при циклической обработке элементов придётся каждый раз проверять, не является ли очередное звено заглавным. Этого недостатка лишён второй вариант списка (рис. 8), но в этом случае труднее реализуется добавление в конец списка.

Рассмотрим основные операции с кольцевыми двунаправленными списками в первом варианте (рис. 7).

23.3. Операции над кольцевыми списками

Вставка элемента

Пусть h, p, q – переменные типа elem*, а k – переменная типа int. Занесем значение k в информационную часть элемента и вставим этот элемент после звена, на которое указывает указатель p. Эту вставку можно осуществить так:

q = new elem(k, p -> next, p); p->nex t-> pred=q; p -> next=q; // В таком порядке! Для вставки нового элемента в начало списка достаточно, чтобы указатель p принял значение адреса заглавного элемента списка: p = h;

Удаление элемента

Возможность двигаться по указателям в любом направлении позволяет задавать исключаемое звено указателем p непосредственно на само это звено:

```
p -> next -> pred = p -> pred; p
-> pred -> next = p -> next;
delete p;
```

Поиск элемента

Пусть h – указатель на заглавный элемент списка, r – указатель, который будет указывать на найденное звено, содержащее k. Пусть также p, q – переменные типа elem*, b – типа int. Поиск элемента, содержащего число k, осуществляется так:

```
b = 1;
h -> data = k + 1;        // В информационную часть заглавного звена
                          // заведомо занесём число, отличное от k.
p = h -> next;  r         // Сначала  p  указывает на первое звено.
= NULL;
q = p;                    // q указывает на первое звено.
do {
  if(p -> data == k){
```

```
  b = 0;
  r = p;
  }
  p = p -> next;
}
while((p != q) && b);
```

Заметим, что если в списке вообще нет звена, содержащего k, то после поиска значение b останется равным единице, указатель r будет равен NULL, а p примет значение q, т. е. снова будет указывать на первое звено (после заглавного).