Курс «Основы программирования»

Федорук Елена Владимировна ст. преподаватель каф РК-6 МГТУ им.Н.Э.Баумана

Лекция №17

Виртуальная память пользователя

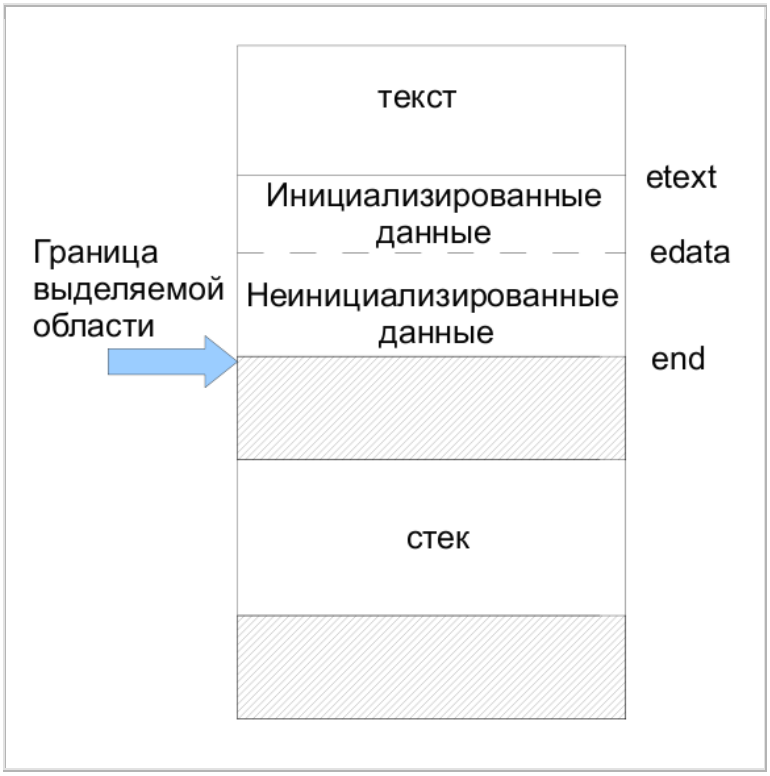

Рис. 1.

Диаграмма на рис. 1 показывает *виртуальную память пользователя*. Она организована из трех сегментов: текст, данные и стек.

Текстовый сегмент – это просто выполняемый код, то есть машинный код программы.

Сегмент данных используется для хранения внешних (**external**) и статических (**static**) переменных. Он содержит инициализированные и неинициализированные данные. Неинициализированные данные автоматически устанавливаются равными нулю при входе в программу. Адрес внешней памяти **etext** обозначает конец текстового сегмента. Конец инициализированных и неинициализированных данных обозначен адресом внешних переменных **edata** и **end**, соответственно. Переменные **etext, edata** и **end** – целые.

Первоначально, граница выделяемой памяти программы – это адрес **end**. Граница выделяемой памяти может изменяться при вызове системных вызовов или библиотечных функций управления памятью. Однако адреса трех переменных **etext**, **edata** и **end** неизмены.

Стек программы используется для хранения аргументов функции и ее локальных (**automatic**) переменных. Стек также используется для сохранения регистров при вызове функции. Расположение вершины стека труднодоступно.

Любая попытка исправить память между концом данных и вершиной стека может привести к порче памяти. Также недоступна память расположенная ниже конца стека.

Можно изменить размер сегмента данных с помощью системных вызовов **brk** и **sbrk**, изменяя границы выделяемой области.

В отличие от сегмента данных, стек программы автоматически увеличивается и уменьшается с каждым вызовом функции и выходом из нее, соответственно.

Функции управления динамической памятью

Функции управления динамической памятью дают возможность легко управлять памятью.

Они являются функциями библиотеки общего назначения.

Объявления этих функций находится в заголовочном файле **<stdlib.h>**, поэтому в программах, использующих эти функции, необходимо включать следующую директиву препроцессора:

```
#include <stdlib.h>
```

Выделение памяти

```
void * malloc(size_t size)
```

Функция выделяет память размером **size** и возвращает адрес начала области памяти. При присвоении указателю происходит явное преобразование типа. Выравнивание границы происходит автоматически, но выделенная память не инициируется нулями.

```
void realloc (void *block, size_t size)
```

Функция делает размер блока по указателю **block** равным **size** байтов и возвращает указатель на блок, возможно перемещенный. В случае перемещения блока данные из исходного блока копируются в новый блок, освобождая исходный блок.

```
void * calloc(size_t nelem, size_t elsize)
```

Функция выделяет блок, который вмещает **nelem** элементов размером **elsize** байтов каждый, возвращает адрес начала области памяти. Выделенная память инициируется нулями.

У каждого блока памяти есть байты, где хранятся длина блока и признак того, занят он или свободен. Эти функции управления памятью используют системные вызовы **brk** и **sbrk**. Когда функциям **malloc**, **calloc** и **realloc** нужно изменить границу выделяемой памяти, они обычно делают это с запасом. Если будущим запросам хватает выделенного пространства, они не используют системный вызов. Так библиотечные функции минимизируют число выполнения системных вызовов.

Если вызов функции закончился неуспешно, то она возвращает значение **NULL**.

Освобождение памяти

`void free(void *block)`

Функция освобождает блок памяти по указателю **`*block`** и делает его доступным для последующих выделений памяти. Указатель должен указывать на блок, полученный ранее вызовом **`malloc`**, **`calloc`** или **`realloc`**.

Функция **`free()`** никогда не возвращает память назад ядру. Вместо этого она устанавливает признак того, что блок свободен. Будущие запросы выделения памяти ищут среди всех свободных блоков блок подходящего размера, прежде чем использовать системный вызов увеличения памяти. Смежные свободные блоки сливаются во время поиска для их более эффективного использования.

Пример 1

```
int *a;


if ((a=(int *)malloc(sizeof(int)))==NULL)    /* выделение памяти под целое типа int*/
  {
   printf("Не хватает памяти для числа.");
   exit(1);
   }
*a=-244;
*a+=10;
free(a);                                      /* освобождение памяти*/
```

Пример 2

```
struct node {
          int       data;
          struct node *next;
          };
struct node *pnode;
if ((pnode = malloc(sizeof(struct node)))==NULL) /* выделение памяти под структуру node*/
  {
   printf("Не хватает памяти для структуры.");
   exit(1);
   }
pnod->data=100;
pnod->next=NULL;
```

Пример 3

```
int *array;
/* выделение памяти под массив из 3 элементов  типа int*/
                        /*элементы массива равны 0*/
if ((array = calloc(3, sizeof(int)))==NULL) /* выделение памяти под структуру node*/
  {
   printf("Не хватает памяти для массива.");
   exit(1);
   }
for (i=0; i<3; i++, array++)
   *array=i+1;
```

Пример 4

```
struct node {
          int         data;
          struct node *next;
          };
struct node *table;
if ((table = calloc(3, sizeof(struct node)))==NULL)
  {
   printf("Не хватает памяти для массива структур.");
   exit(1);
   }
```

Выделяется память под массив из трех элементов структур **node**. Ссылаться на элементы можно как в любом другом массиве.

```
table[2].data = 3;
```

Пример 5

Вставка узла в список структур

```
struct node {
          int         data;
          struct node *next;
          };
struct node *insert (struct node *p, int i)                /*вставка узла после узла *P*/
{struct node *q;
if ((q= malloc(sizeof(struct node)))==NULL)
  {
   printf("Не хватает памяти для структуры.");
   exit(1);
   }
```

```
q->next = p->next;
q->data = i;
p->next = q;
return (g);
}
```

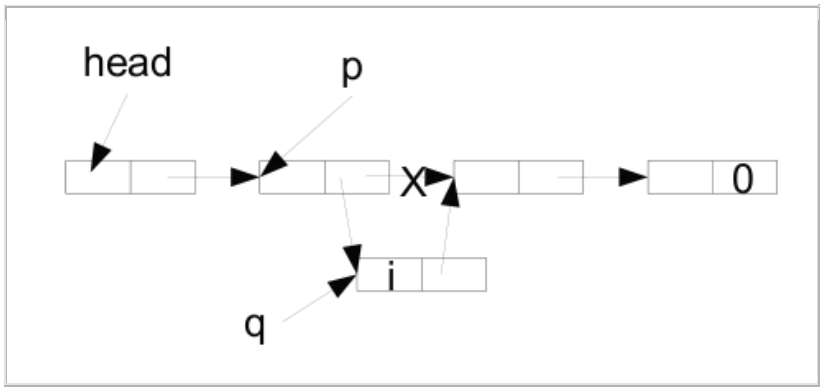

Рис. 1. Вставка узла

Пример 6

Уничтожение узла в списке структур

```
void delete (struct node *p)
{ struct node *q;
  q = p->next;
  p->data=q->data;
  p->next=q->next;
  free (q);
}
```

Эта функция уничтожает данные, содержащиеся в узле с указателем **p**. Данные в узле с указателем **q** копируются в узел с указателем **p**. Потом узел **q** уничтожается. Чтобы этот алгоритм уничтожения работал со всеми узлами, включая последний в списке, пустой список должен содержать голову **head** и узел с нулевым указателем в поле **next**. Другими словами, последний узел является ограничителем и его нельзя уничтожать. Поэтому в программу надо включить следующую проверку:

```
if(p->next==NULL) return;
```

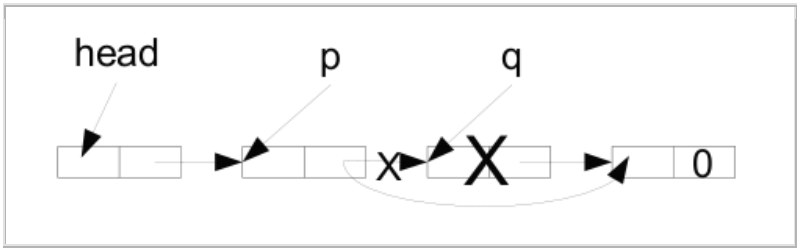

Рис. 2. Уничтожение узла

Динамические структуры данных

Динамические структуры данных могут быть организованы линейно, в виде дерева и в виде сети.

Линейная динамическая структура - изменяемая последовательность элементов. Примеры таких структур:

- *стек* (элемент можно добавлять только в конец; удаляется только последний элемент);
- *очередь* (элемент можно добавлять только в конец; удаляется только первый элемент);
- *дек* (добавление и удаление элементов и с начала, и с конца).

Дерево - структура, в которой каждый элемент (вершина) ссылается на один или более элементов следующего уровня.

В сетевой структуре никаких ограничений на связи не накладывается.

Линейные динамические структуры, такие как стеки, очереди и деки, при известном количестве элементов в них можно реализовать в виде динамических или статических одномерных массивов. В противном случае используют списки.

Список - это структура, в которой помимо данных хранятся также адреса элементов. Элемент списка состоит из двух частей: информационной и адресной, где хранятся указатели на следующие элементы. В зависимости от количества полей в адресной части и порядка связывания элементов различают:

- линейные односвязные списки - единственное адресное поле содержит адрес следующего элемента, для последнего элемента списка этот адрес равен нулю (**NULL**),
- кольцевые односвязные списки - единственное адресное поле содержит адрес следующего элемента, последний элемент ссылается на первый,
- линейные двусвязные списки - каждый элемент содержит адреса предыдущего и последующего элемента; первый элемент в качестве адреса предыдущего элемента, а последний элемент в качестве адреса последующего элемента содержат нули (**NULL**),
- кольцевые двусвязные списки - каждый элемент содержит адреса предыдущего и последующего элемента, причем первый элемент в качестве адреса предыдущего элемента содержит адрес последнего элемента, а последний элемент в качестве адреса предыдущего элемента содержит первого элемента,
- n-связные списки - каждый элемент включает несколько адресных полей, в которых записаны адреса других элементов или **NULL**.

Для описания элементов списка используют в программах на языке Си структуры.

Линейные односвязные списки

Линейные односвязные списки используют чаще других списковых структур, так как они сравнительно просты. Эти списки позволяют работать с произвольным количеством элементов, добавляя и удаляя их по мере надобности, а также осуществлять вставку и удаление элементов, не перемещая другие элементы последовательности. Однако, реализация односвязного списка требует дополнительной памяти для хранения адресной части элемента.

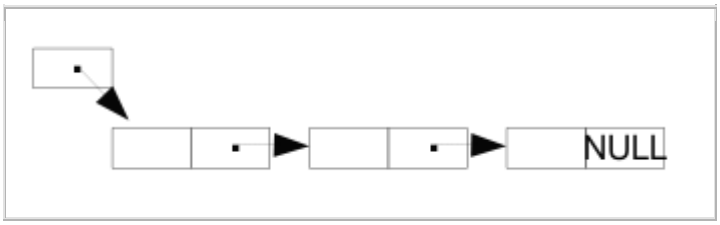

Рис. 1. Линейный односвязный список

Основные операции с линейными односвязными списками

1. Добавление нового элемента к списку.
2. Поиск элемента в списке.
3. Удаление элемента из списка.

Добавление элемента к списку

Добавление элемента к списку включает запрос памяти для размещения элемента и заполнение его информационной части. Построенный таким образом элемент добавляется к уже существующей части списка.

В общем случае при добавлении элемента к списку возможны следующие варианты:

- список пуст, добавляемый элемент станет единственным элементом списка;
- элемент необходимо вставить перед первым элементом списка;
- элемент необходимо вставить перед заданным элементом списка;
- элемент необходимо дописать в конец списка.

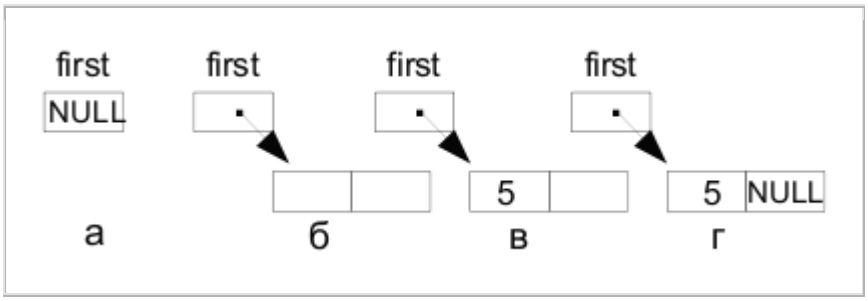

Рис. 1.

На рис. 1 показана последовательность операций при добавлении элемента к пустому списку:

а) исходное состояние;

б) запрос памяти под элемент;

в) заполнение элемента;

г) занесение NULL в адрес следующего элемента.

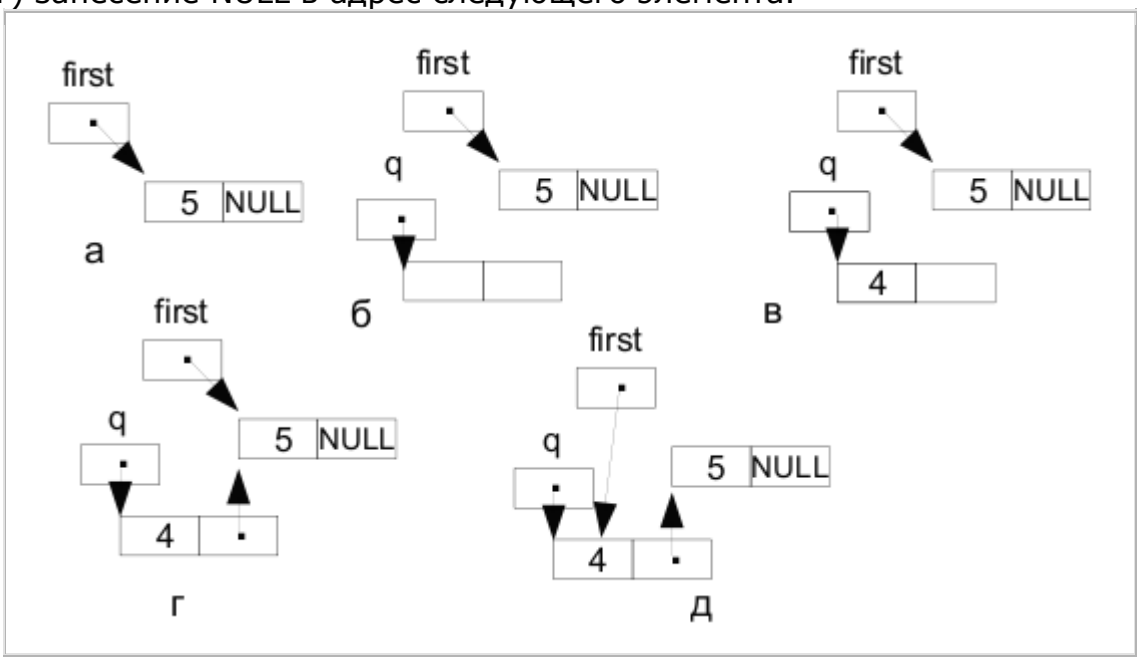

Рис. 2.

На рис. 2 показана последовательность операций при добавлении элемента перед первым:

а) исходное состояние;

б) запрос памяти под элемент;

в) заполнение элемента;

г-д) шаги включения элемента в список.

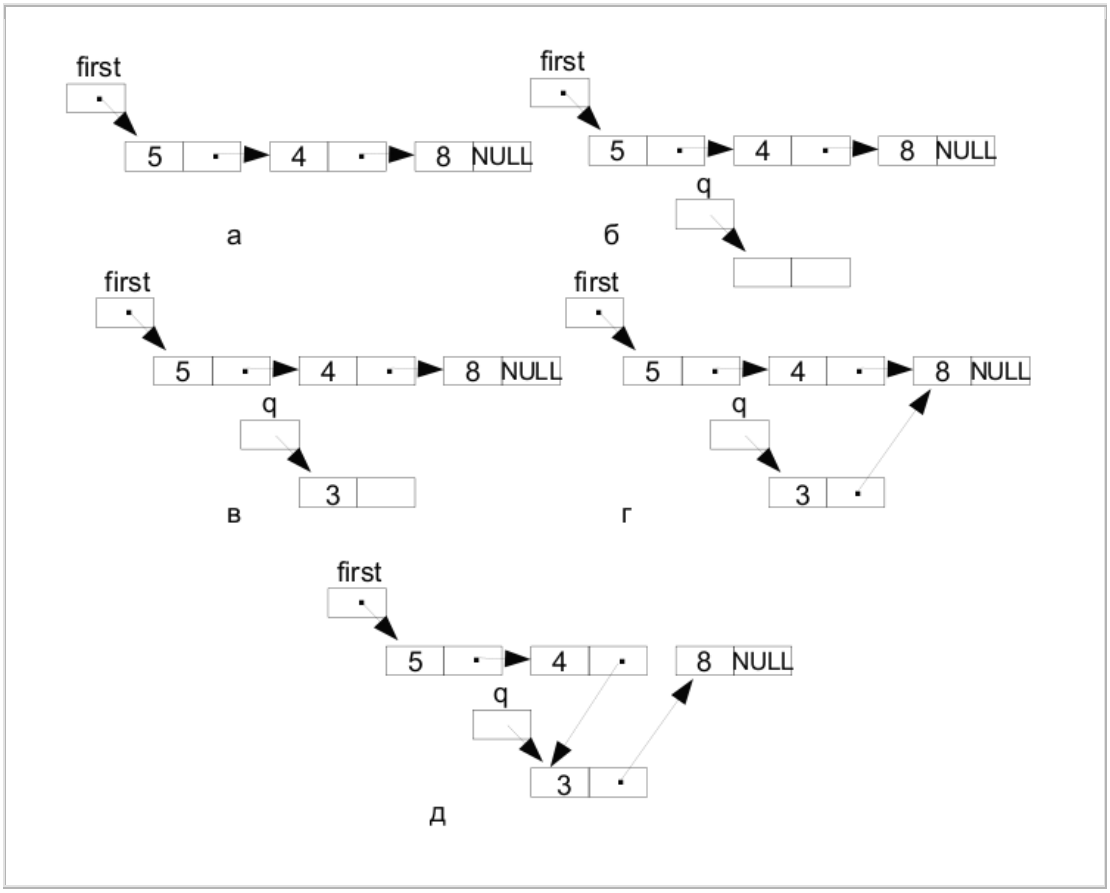

Рис. 3.

На рис. 3 показана последовательность операций при добавлении элемента перед заданным (не первым):

а) исходное состояние;

б) запрос памяти под элемент;

в) заполнение элемента;

г-д) шаги включения элемента в список.

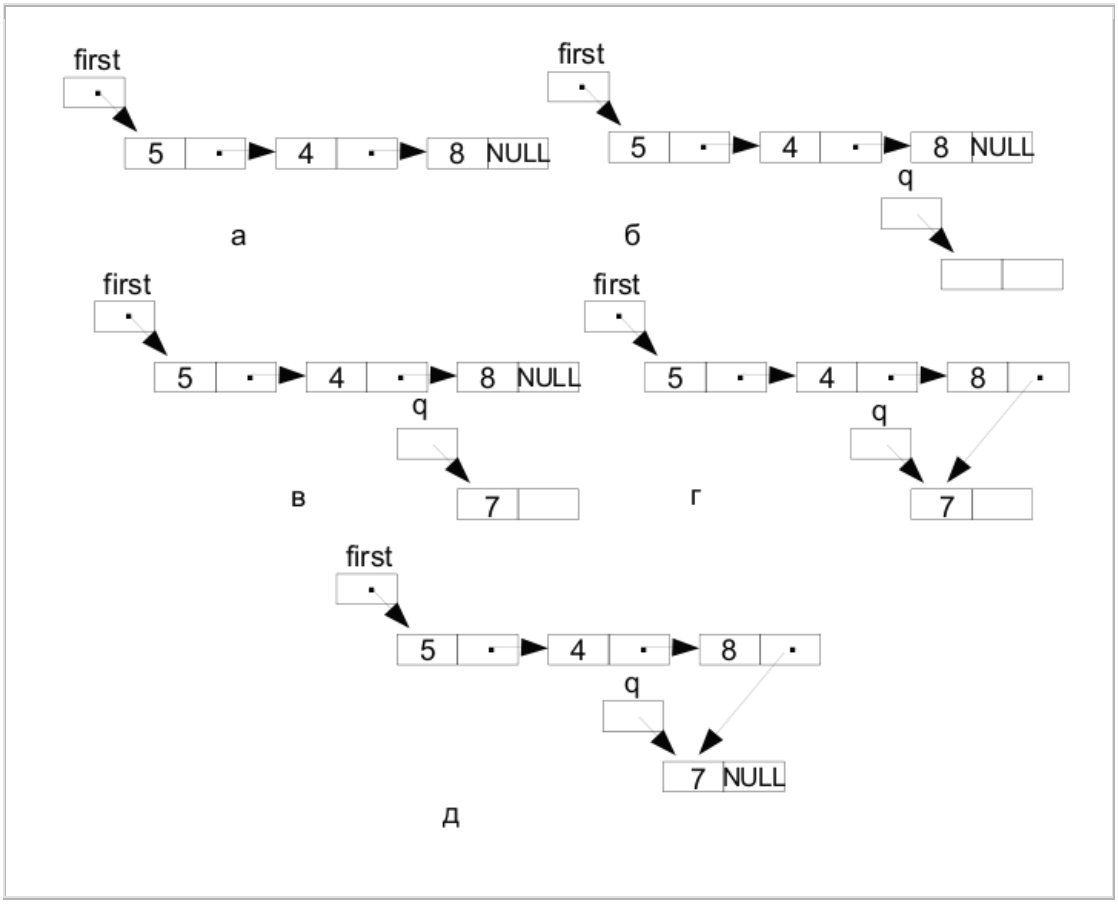

Рис. 4.

На рис. 4 показана последовательность операций при добавлении элемента в конец списка:

а) исходное состояние;

б) запрос памяти под элемент;

в) заполнение элемента;

г-д) шаги включения элемента в список.

Просмотр и обработка элементов списка

Просмотр и обработка элементов списка выполняется последовательно с использованием дополнительного указателя.

Пример 1

```
    f=first;                    /* first -начальный адрес списка */
while (f!=NULL)
  {
   f=f->next;                   /* обработка элемента списка*/
  }
```

Поиск элемента в списке также выполняется последовательно, но при этом, как это и положено в поисковом цикле, обычно организуют выход из цикла, если нужный элемент найден, и осуществляют проверку после цикла, был ли найден элемент.

Пример 2

```
f=first;                /* first -начальный адрес списка */
```

```
flag=0;
while(f!=NULL && flag==0)
  {
   if(...)            /*условие поиска элемента*/
    {                 /*действие над найденным элементом*/
     flag=1;
     }
   else
     f=f->next;
   }
if(flag==1)
  /*элемент найден*/
else
  /*элемент не найден*/
```

Удаление элемента списка

При выполнении операции удаления также возможны четыре случая:

1. удаление единственного элемента;
2. удаление первого (не единственного) элемента;
3. удаление элемента, следующего за данным;
4. удаление последнего элемента.

После удаление единственного элемента список становится пустым, следовательно, при выполнении этой операции необходимо не только освободить память, выделенную для размещения элемента, но и занести **`NULL`** в указатель списка **`first`**.

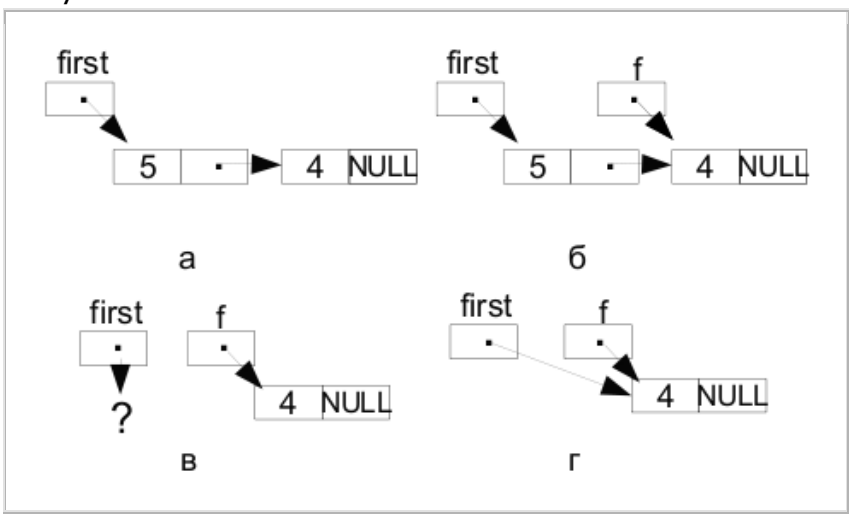

Рис. 1.

На рис. 1 показана последовательность операций при удалении первого (не единственного) элемента:

а) исходное состояние;

б) сохранение адреса следующего элемента в специальном указателе;

в) освобождение памяти;

г) запись в указатель списка адрес следующего элемента.

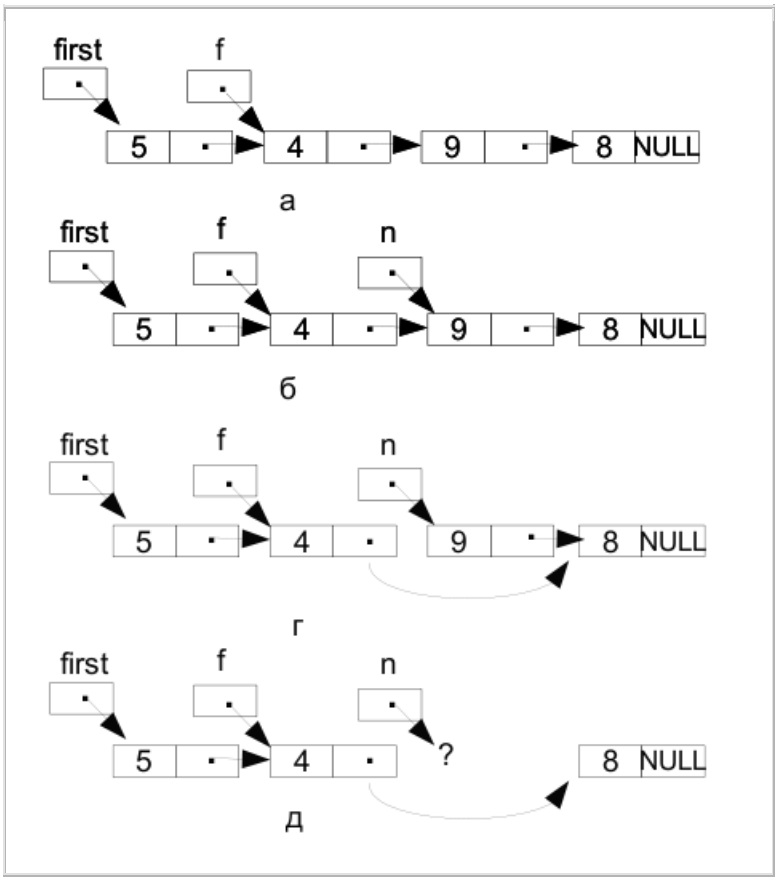

Рис. 2.

На рис. 2 показана последовательность операций при удалении элемента, следующего за данным:

а) исходное состояние;

б) сохранение адреса удаляемого элемента в специальном указателе;

в) исключение удаляемого элемента из списка;

г) освобождение памяти; .

Удаление последнего элемента отличается только тем, что в поле "адрес следующего " заданного элемента записывается **NULL**.