

Динамические структуры данных

Для решения сложных задач необходимо выбрать правильную структуру данных. Обычно структура данных и алгоритм тесно взаимосвязаны.

Все обрабатываемые данные, в конце концов компьютером разбиваются на отдельные биты. Но писать программы для работы с битами достаточно трудоемкое занятие. Поэтому, *типы* позволяют указать, как будут использоваться определенные наборы битов, *функции* — задавать операции, определяемые над данными, *структуры* используются для группировки разнородных частей информации, *указатели* для не прямой ссылки на информацию.

Возможно одной из фундаментальных структур данных является массив, представляющий собой фиксированный набор элементов одного типа, хранящихся в виде непрерывного ряда. Доступ к i -тому элементу производится по индексу, что совпадает с устройством почти всем системам памяти компьютера. При этом вставка и удаление элементов массива занимает линейное время, что не всегда удобно.

Очень часто выгоднее использовать *связные списки* — базовую структуру данных, в которой каждый элемент содержит информацию, необходимую для получения следующего элемента. Основное преимущество связных списков перед массивами — возможность эффективного изменения расположения элементов, при этом скорость доступа к элементам увеличивается, так как единственный способ состоит в отслеживании связей от начала списка.

Определение 1. Связный список — это набор элементов, причем каждый из них является частью **узла (node)**, который также содержит **ссылку (link)** на узел.

Узлы определяются ссылками на узлы, поэтому связные списки иногда называют *самоссылочными (selfreferent)* структурами. Обычно под связным списком понимается реализация последовательного расположения элементов. Начиная с некоторого узла (который считаем первым узлом последовательности) прослеживается ссылка на другой узел, который дает второй элемент последовательности и т. д. Для ссылки последнего узла принимается одно из следующих соглашений:

1. *Пустая ссылка (null)*, не указывающая на какой-нибудь узел.

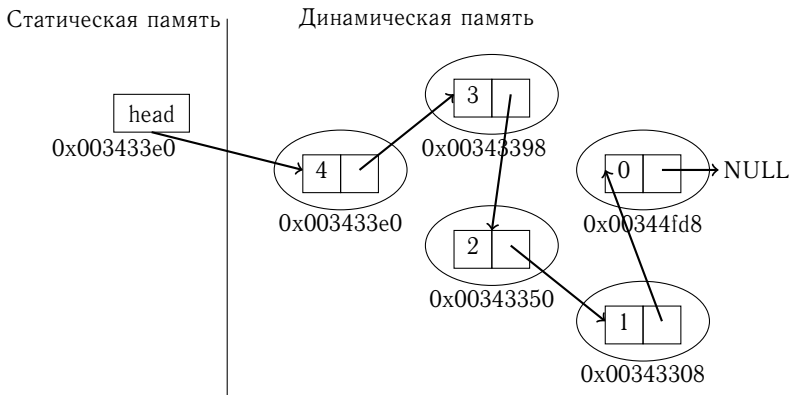


Рис. 1.1. Пример динамической структуры данных

2. Ссылка указывает на *фиктивный узел (dummy node)*, который не содержит элементов.
3. Ссылка указывает на первый узел, что делает список *циклическим*.

Динамические структуры данных создаются с помощью операции **new**, следовательно, располагаются в динамической памяти. Каждый элемент, помимо информационного поля, содержит адрес следующего элемента (ссылку). Элементы не имеют собственного наименования, имеется только указатель **head**, который расположен в статической памяти, и содержит адрес первого элемента структуры. Ко всем остальным элементам можно обратиться только пройдя последовательно по всем связям.

Указатели для ссылок и структуры для узлов описываются следующим образом:

```

struct node
{
    Item item;
    node *next;
};
  
```

Эта пара выражений — код C++ для определения 1. Узлы состоят из элементов типа **Item** и указателей на узлы. Указатели на узлы также часто называются ссылками.

Динамические структуры данных делятся на

- Списки:
 - Стеки;

- Очереди;
 - Односвязные списки;
 - Двусвязные списки;
 - Кольцевые списки;
- Деревья;
 - Графы.

1.1. Стеки

Стек — частный случай списка. Стек удовлетворяет принципу LIFO — «Last In First Out» (последний зашел — первый вышел). На основе стеков устроены большинство компьютерных операций, в частности, рекурсивные функции основаны на стеках.

По своему устройству стек напоминает детскую игрушку — пирамидку. На примере пирамидки понятно, что для того, чтобы добраться до одетого первым элементом, необходимо снять все верхние элементы. Другой пример стека — имеется сосуд, который последовательно заполняется шарами. Тогда, чтобы вынуть последний шар из сосуда, необходимо вынуть все остальные.

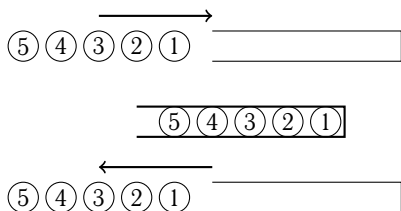


Рис. 1.2. Представление стека

Точно также для стека определены только две операции: добавить элемент в начало стека и извлечь элемент из начала стека. Других операций для стека НЕ ОПРЕДЕЛЕНО. Как видно из рисунка ?? элементы в стека расположены в порядке, обратном первоначальному.

Стек описывается как `struct` следующим образом:

Листинг 1.1.

```
1 struct stack{
2     int inf;
```

```
3  stack *next;
4  };
```

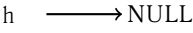

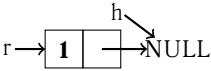
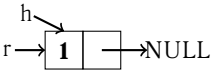
Рассмотрим функцию добавления элемента в стек. По сложившейся традиции эта функция называется `push()`.

Листинг 1.2.

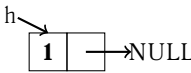
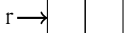
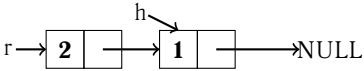
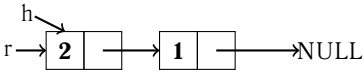
```
1 void push(stack *&h, int x){
2     stack *r = new stack; //создаем новый элемент
3     r->inf = x;           //поле inf = x
4     r->next = h;         //следующим элементов является h
5     h = r;               //теперь r является головой
6 }
```

В качестве параметров в функцию передаются указатель на стек `h` (в случае пустого стека он обязательно должен быть инициализирован, то есть, равен `NULL`), и значение элемента.

Добавление первого элемента в стек:

	
Сначала создается новый элемент типа <code>stack</code> (строка 2).	
Его поля заполняются, соответственно, информационное поле (<code>inf</code>) значением <code>x</code> (строка 3), а поле ссылок (<code>next</code>) — адресом <code>h</code> (первый элемент стека) (строка 4).	
После присоединения нового элемента к голове стека, уже новый элемент является головой и <code>h</code> указывает на этот элемент (строка 5).	

Добавление второго элемента в стек:

	
Сначала создается новый элемент типа <code>stack</code> (строка 2).	
Его поля заполняются, соответственно, информационное поле (<code>inf</code>) значением <code>x</code> (строка 3), а поле ссылок (<code>next</code>) — адресом <code>h</code> (первый элемент стека) (строка 4).	
После присоединения нового элемента к голове стека, уже новый элемент является головой и <code>h</code> указывает на этот элемент (строка 5).	

Рассмотрим теперь функцию удаления элемента из стека. По сложившейся традиции эта функция называется `pop`. Для простоты будем не только удалять элемент, но и возвращать его значение.

Листинг 1.3.

```

1 int pop (stack *&h){
2     int i = h->inf; //значение первого элемента
3     stack *r = h;  //указатель на голову стека
4     h = h->next;    //переносим указатель на следующий элемент
5     delete r;       //удаляем первый элемент
6     return i;       //возвращаем значение
7 }
```

Сохраняем значение первого элемента	
Создаем указатель r на голову (строка 3.)	
Переносим указатель h на следующий элемент (строка 4.)	
Удаляем элемент, на который показывает указатель r (строка 5.)	
Возвращаем значение удаленного элемента (строка 6.)	

Такие свойства стека не позволяют изменять элементы этого стека. Для того, чтобы выполнять какие-то действия, необходимо создавать еще один стек и перебрасывать элементы из одного стека в другой.

Рассмотрим следующую задачу.

Пример 1.1. УСЛОВИЕ: Создать стек, состоящий из целых чисел. Удалить все повторяющиеся элементы, оставив только их первые вхождения. Порядок следования элементов должен совпадать с порядком ввода элементов (Например, вводится следующий набор данных: 1 2 3 1 2 4 1. Результат: 1 2 3 4.)

РЕШЕНИЕ:

Поскольку нельзя просмотреть стек, то в один стек будем записывать первые вхождения элементов, а два других будем использовать для перезаписи элементов стека, исключая повторяющиеся.

Так как запись элементов производится в обратном порядке, создадим функцию, которая будет переписывать элементы в новый стек, чтобы получить нужный порядок следования элементов.

Листинг 1.4.

```
1 void reverse(stack *&h){    //"обращение"стека
2     stack *head1 = NULL;    //"инициализация буферного стека
3     while (h)                //пока стек не пуст
4         push(head1, pop(h)); //переписываем из одного стека в другой
5     h = head1;                //переобозначаем указатели
6 }
```

Создадим два стека `head` и `head1`. Для записи результата создадим стек `res`.

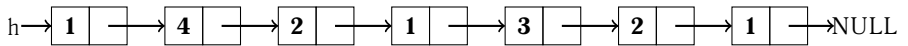
Алгоритм решения задачи следующий:

1. Извлекаем первый элемент из стека `head`, сохраняем его значение в `x` и записываем его в результирующий стек.
2. Если стек не пуст, извлекаем первый элемент, и, если он не равен `x`, записываем в новый стек `head1`.
3. Переобозначаем указатели.
4. Если стек `head` пуст, завершаем работу, иначе возвращаемся к шагу 1.

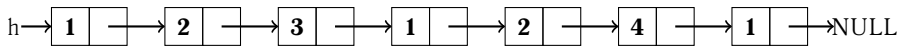
Листинг 1.5.

```
1 stack *result(stack *&h){
2     stack *res = NULL; //инициализация
3     stack *h1 = NULL;
4     while(h){
5         int x = pop(h); //удаляем первый элемент
6         push(res, x);   //записываем в результат
7         while(h){
8             int y = pop(h); //удаляем элемент из стека
9             if (x != y) push(h1, y); //записываем в новый стек
10        }
11        reverse(h1); //переворачиваем стек
12        h = h1;      //переобозначаем указатели
13        h1 = NULL;
14    }
15    reverse(res);    //переворачиваем результирующий стек
16    return res;
17 }
```

Создаем стек:

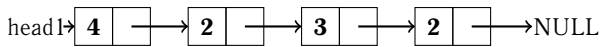


Переворачиваем его:

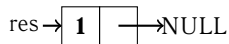


Пока стек **head** не пуст:

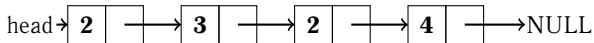
Извлекаем первый элемент, записываем его в стек **res**. В стек **head1** записываем элементы, не совпадающие с первым:



head → NULL



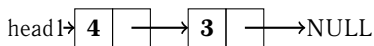
Переобозначаем указатели и переворачиваем стек:



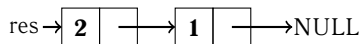
head1 → NULL



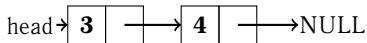
Повторяем алгоритм:



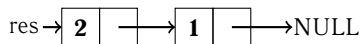
head → NULL



Переобозначаем указатели и переворачиваем стек:



head1 → NULL



Далее:

head →

4	→ NULL
---	--------

head → NULL

res →

3	→
---	---

 →

2	→
---	---

 →

1	→ NULL
---	--------

Наконец,

head → NULL

head → NULL

res →

4	→
---	---

 →

3	→
---	---

 →

2	→
---	---

 →

1	→ NULL
---	--------

Переворачиваем полученный стек и выводим результат на экран:

res →

1	→
---	---

 →

2	→
---	---

 →

3	→
---	---

 →

4	→ NULL
---	--------

Ниже приведен код программы, реализующей предложенный алгоритм:

Листинг 1.6.

```
1 #include<iostream>
2 using namespace std;
3
4 struct stack{
5     int inf;
6     stack *next;
7 };
8
9 void push(stack *&h, int x){//вставка
10     stack *r = new stack;
11     r->inf = x;
12     r->next = h;
13     h = r;
14 }
15
16 int pop (stack *&h){ //удаление
17     int i = h->inf;
18     stack *r = h;
19     h = h->next;
20     delete r;
21     return i;
22 }
```

```

23
24 void reverse(stack *&h){ //"обращение"стека
25     stack *head1 = NULL; //инициализация буферного стека
26     while (h)           //пока стек не пуст
27         push(head1, pop(h)); //переписываем из одного стека в другой
28     h = head1;           //переобозначаем указатели
29 }
30
31 stack *result(stack *&h){
32     stack *res = NULL; //инициализация
33     stack *h1 = NULL;
34     while(h){
35         int x = pop(h); //удаляем первый элемент
36         push(res, x);   //записываем в результат
37         while(h){
38             int y = pop(h); //удаляем элемент из стека
39             if (x != y) push(h1, y); //записываем в новый стек
40         }
41         reverse(h1); //переворачиваем стек
42         h = h1;      //переобозначаем указатели
43         h1 = NULL;
44     }
45     reverse(res);    //переворачиваем результирующий стек
46     return res;
47 }
48
49 int main(){
50     int n;
51     cout << " n = ";
52     cin >> n;
53     stack *head = NULL; //инициализация
54     int x;
55     for (int i = 0; i < n; i++){ //создаем стек
56         cin >> x;
57         push(head, x);
58     }
59     reverse(head); //переворачиваем стек
60     stack *res = result(head); //результат
61     while(res)
62         cout << pop(res) << " "; //выводим на экран

```

```

63     cout << endl;
64     return 0;
65 }

```

□

1.2. Очередь

Очередь также представляет собой частный случай списка. Элементы очереди устроены по принципу FIFO — First In First Out (Первый зашел — первый вышел). Очередь можно представить как сосуд без дна. С одной стороны заполняется, с другой извлекается:

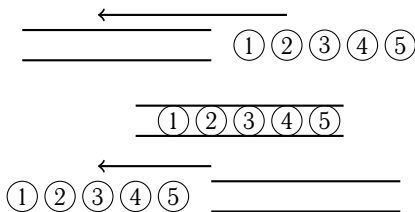


Рис. 1.3. Представление очереди

Как видно из рисунка ?? вставлять или удалять элементы в середину очереди нельзя. То есть, для очереди определены только две операции: добавить элемент в хвост очереди и извлечь элемент из начала очереди. Других операций для стека НЕ ОПРЕДЕЛЕНО.

Очередь описывается как `struct` следующим образом:

Листинг 1.7.

```

1 struct queue {
2     int inf;
3     queue *next;
4 };

```

Рассмотрим функцию добавления элемента в очередь. По сложившейся традиции эта функция называется `push()`.

Листинг 1.8.

```

1 void push (queue *&h, *&t, int x){ //вставка элемента в очередь
2     queue *г = new queue;        //создаем новый элемент
3     г->inf = x;

```

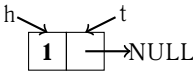

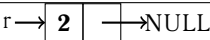
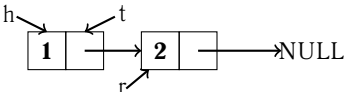
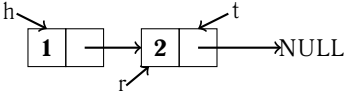
```
4   r->next = NULL;           //всегда последний
5   if (!h && !t){             //если очередь пуста
6       h = t = r;             //это и голова и хвост
7   }
8   else {
9       t->next = r;             //r - следующий для хвоста
10      t = r;                  //теперь r - хвост
11  }
12 }
```

В качестве параметров в функцию передаются указатели на голову `h` и хвост `t` очереди (в случае пустой очереди они обязательно должны быть инициализированы, то есть, равны `NULL`), и значение элемента.

Добавление первого элемента в очередь:

	<div>h → NULL ← t</div>		
Сначала создается новый элемент типа <code>queue</code> (строка 2).	<div>г → <table border="1"><tr><td></td><td></td></tr></table></div>		
Его поля заполняются, соответственно, информационное поле (<code>inf</code>) значением <code>x</code> (строка 3), а поле ссылок (<code>next</code>) — <code>NULL</code> (последний элемент очереди) (строка 4).	<div>г → <table border="1"><tr><td>1</td><td>→ NULL</td></tr></table></div>	1	→ NULL
1	→ NULL		
Так как очередь пуста, и <code>h</code> и <code>t</code> являются указателями на <code>r</code> (строки 5 и 6).	<div><div>h →</div><div>г → <table border="1"><tr><td>1</td><td>→ NULL</td></tr></table></div><div>← t</div></div>	1	→ NULL
1	→ NULL		

Добавление второго элемента в очередь:

	
Сначала создается новый элемент типа <code>queue</code> (строка 2).	
Его поля заполняются, соответственно, информационное поле (<code>inf</code>) значением <code>x</code> (строка 3), а поле ссылок (<code>next</code>) — <code>NULL</code> (последний элемент очереди) (строка 4).	
Так как элемент присоединяется к хвосту очереди, то этот элемент является следующим для хвоста (строка 9).	
Новый элемент теперь является хвостом (строка 10).	

Рассмотрим теперь функцию удаления элемента из головы очереди. По сложившейся традиции эта функция называется `pop`. Для простоты будем не только удалять элемент, но и возвращать его значение.

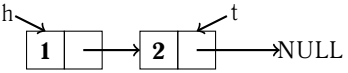
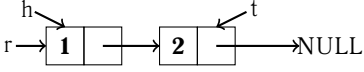
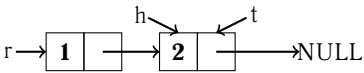
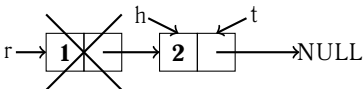
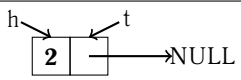
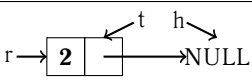
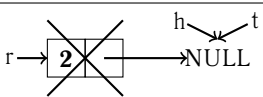

Листинг 1.9.

```

1  int pop (queue *&h, *&t){ //удаление элемента из очереди
2      queue *r = h;         //создаем указатель на голову
3      int i = h->inf;         //сохраняем значение головы
4      h = h->next;            //сдвигаем указатель на следующий элемент
5      if (!h)                 //если удаляем последний элемент из очереди
6          t = NULL;
7      delete r;               //удаляем первый элемент

```

```
8     return i;
9 }
10 }
```

Сохраняем значение перво- го элемента (строка 3).	
Создаем указатель r на го- лову очереди (строка 2).	
Переносим указатель h на следующий элемент (строка 4.)	
Удаляем элемент, на кото- рый показывает указатель r (строка 7).	
Возвращаем значение уда- ленного элемента (строка 8.)	
Удаляем единственный эле- мент очереди (Повторяем строки 2-4).	
Обнуляем указатель t (строка 6).	
Возвращаем значением уда- ленного элемента (строка 8).	

Такие свойства очереди не позволяют изменять элементы этой очереди. Для того, чтобы выполнять какие-то действия, необходимо создавать еще одну очередь и перебирать элементы из одной очереди в другую.

Рассмотрим в качестве примера ту же задачу, что и для стека.

Пример 1.2. УСЛОВИЕ: Создать очередь, состоящую из целых чисел. Удалить все повторяющиеся элементы, оставив только их первые вхождения. (Например, вводится следующий набор данных: 1 2 3 1 2 4 1. Результат: 1 2 3 4.)

РЕШЕНИЕ:

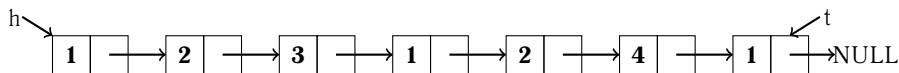
Поскольку нельзя просмотреть очередь, то в одну очередь будем записывать первые вхождения элементов, а две других будем использовать для перезаписи элементов очереди, исключая повторяющиеся.

Создадим две очереди `head`, `tail` и `head1`, `tail1`. Для записи результата создадим очередь `resh`, `rest`.

Алгоритм решения задачи следующий:

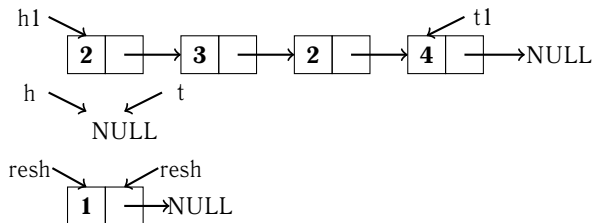
1. Извлекаем первый элемент из очереди `head`, `tail`, сохраняем его значение в `x` и записываем его в результирующую очередь.
2. Если очередь не пуста, извлекаем первый элемент, и, если он не равен `x`, записываем в новую очередь `head1`, `tail1`.
3. Переобозначаем указатели.
4. Если очередь `head`, `tail` пуста, завершаем работу, иначе возвращаемся к шагу 1.

Создаем очередь `h`, `t`:

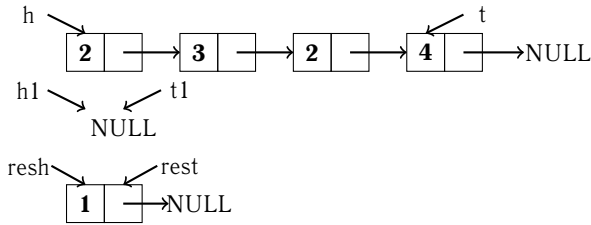


Удаляем первый элемент, записываем его в результирующую очередь `resh`, `rest`.

Остальные элементы, не равные первому, записываем в очередь `h1`, `t1`:

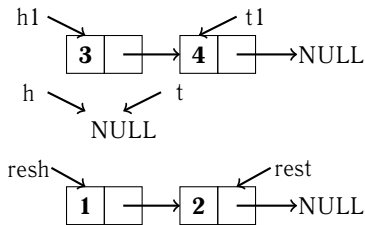


Переобозначаем указатели:

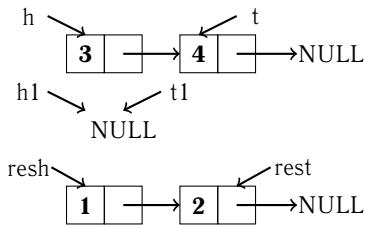


Удаляем первый элемент, записываем его в результирующую очередь **resh**, **rest**.

Остальные элементы, не равные первому, записываем в очередь **h1**, **t1**:

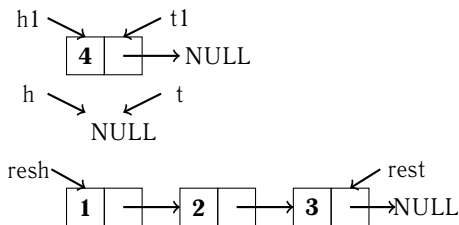


Переобозначаем указатели:

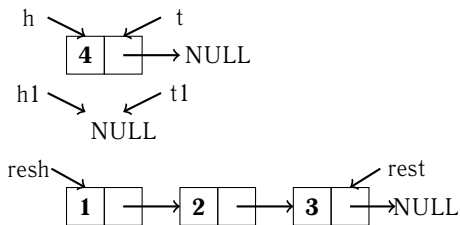


Удаляем первый элемент, записываем его в результирующую очередь **resh**, **rest**.

Остальные элементы, не равные первому, записываем в очередь **h1**, **t1**:

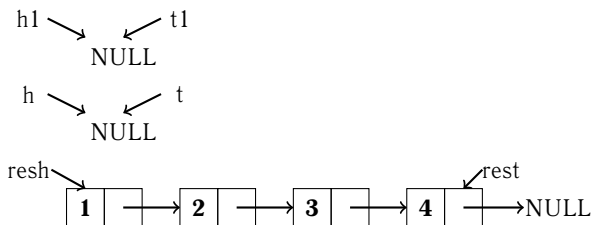


Переобозначаем указатели:



Удаляем первый элемент, записываем его в результирующую очередь `resh`, `rest`.

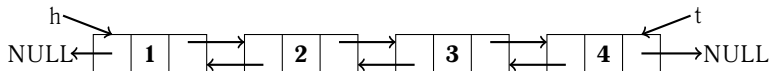
Остальные элементы, не равные первому, записываем в очередь `h1`, `t1`:



□

1.3. Двусвязный список

Наиболее общий случай связанных списков. Каждый элемент списка состоит из трех полей: информационного и двух ссылочных на следующий и предыдущий элементы:



С элементами списка можно выполнять любые действия: добавлять, удалять, просматривать и т. д. Вставка и удаление элемента выполняется за время $O(1)$, так как надо всего лишь поменять значения в ссылочных полях.

Список относится к элементам с последовательным доступом, т. е., чтобы просмотреть пятый элемент, необходимо последовательно просмотреть первые четыре элемента. Для перехода к следующему элементу указателю `p` присваивается значение `p->next`. Так как список двусвязный, можно просматривать элементы как в прямом (`p = p->next`), так и в обратном порядке (`p = p->prev`).

Рассмотрим основные функции для работы со списками: добавление элемента в конец списка, просмотр элементов списка, удаление элемента из списка, вставка элемента в список, поиск элемента, удаление всего списка.

Список описывается как `struct` следующим образом:

Листинг 1.10.

```

1 struct list {
2     int inf;
3     list *next;
4     list *prev;
5 };

```

Рассмотрим функцию добавления элемента в конец списка. По сложившейся традиции эта функция называется `push()`.

Листинг 1.11.

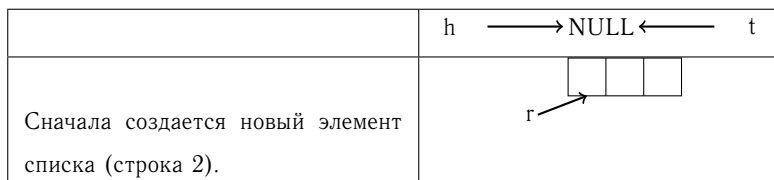
```

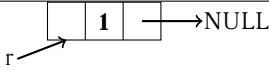
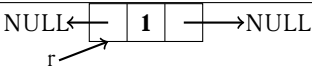
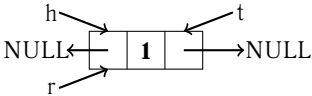
1 void push (list *&h, list *&t, int x){ //вставка элемента в конец списка
2     list *r = new list;               //создаем новый элемент
3     r->inf = x;
4     r->next = NULL;                   //всегда последний
5     if (!h && !t){                    //если список пуст
6         r->prev = NULL;               //первый элемент
7         h = r;                       //это голова
8     }
9     else{
10        t->next = r;                   //r - следующий для хвоста
11        r->prev = t;                  //хвост - предыдущий для r
12    }
13    t = r;                           //r теперь хвост
14 }

```

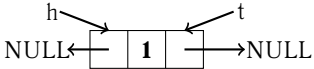

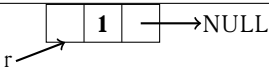
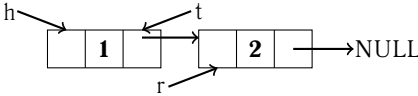
В качестве параметров в функцию передаются указатели на голову `h` и хвост `t` списка (в случае пустого списка они обязательно должны быть инициализированы, то есть, равны `NULL`), и значение элемента.

Добавление первого элемента в список:



Его поля заполняются, соответственно, информационное поле (<i>inf</i>) значением <i>x</i> (строка 3), а поле ссылок (<i>next</i>) — <i>NULL</i> (последний элемент списка) (строка 4).	
Поле <i>prev</i> заполняется <i>NULL</i> (будет первый элемент списка, строка 6).	
Этот элемент является и головой и хвостом списка (строки 7 и 13).	

Добавление второго элемента в список.

	
Сначала создается новый элемент списка (строка 2).	
Его поля заполняются, соответственно, информационное поле (<i>inf</i>) значением <i>x</i> (строка 3), а поле ссылок (<i>next</i>) — <i>NULL</i> (последний элемент списка) (строка 4).	
Текущий элемент становится следующим для хвоста (строка 10).	

<p>Предыдущим для текущего элемента является хвост (строка 11).</p>	
<p>Текущий элемент становится хвостом (строка 13).</p>	

Для добавления элементов в начало списка, нужно в функции `push()` поменять местами `prev` и `next`, а также `h` и `t`.

Для вывода данных на экран необходимо создать указатель на голову, и с его помощью пройти все элементы:

Листинг 1.12.

```

1 void print (list *h, list *t){ //печать элементов списка
2     list *p = h;              //указатель на голову
3     while (p){                 //пока не дошли до конца списка
4         cout << p->inf << " ";
5         p = p->next;            //переход к следующему элементу
6     }
7 }
```

Для поиска элемента в списке пользуемся тем же алгоритмом: создаем указатель на голову, и с его помощью проходим по всему списку. Если встретили искомый элемент, прекращаем работу. Результат работы функции: либо адрес искомого элемента, либо `NULL`, если искомого элемента в списке нет:

Листинг 1.13.

```

1 list *find (list *h, list *t, int x){ //печать элементов списка
2     list *p = h;                  //указатель на голову
3     while (p){                    //пока не дошли до конца списка
4         if (p->inf == x) break // если нашли, прекращаем цикл
5         p = p->next;              //переход к следующему элементу
6     }
7     return p;                    //возвращаем указатель
8 }
```

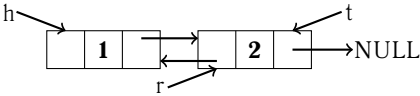
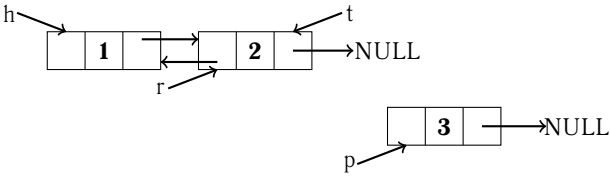
Рассмотрим подробно функцию вставки элемента в список после определенного

элемента. В качестве параметров передаются указатели на голову (h) и хвост (t) списка, указатель на элемент, после которого будем вставлять (r) и значение нового элемента (x).

Листинг 1.14.

```
1 void insert_after ( list *&h, list *&t, list *r, int y){ //вставляем после r
2     list *p = new list;                                //создаем новый элемент
3     p->inf = y;
4     if (r == t){                                        //если вставляем после хвоста
5         p->next = NULL;                                //вставляемый эл-т - последний
6         p->prev = r;                                    //вставляем после r
7         r->next = p;
8         t = p;                                          //теперь хвост - p
9     }
10    else{                                                //вставляем в середину списка
11        r->next->prev = p;                                //для следующего за r эл-та предыдущий - p
12        p->next = r->next;                                //следующий за p - следующий за r
13        p->prev = r;                                      //p вставляем после r
14        r->next = p;
15    }
16 }
```

Возможно два варианта вставки: в конец списка и в середину. Рассмотрим сначала вставку в конец списка.

Вставляем после элемента, на который указывает r	
Создаем новый элемент (строка 2) и заполняем информационное поле значением x (строка 3), поле next — NULL (строка 5).	

<p>Предыдущим для p становится r (строка 6).</p>	
<p>p становится следующим за r (строка 7).</p>	
<p>p становится хвостом списка (строка 8).</p>	

Рассмотрим вставку элемента в середину списка.

<p>Вставляем после элемента, на который указывает r</p>	
<p>Создаем новый элемент (строка 2) и заполняем информационное поле значением x (строка 3).</p>	

<p>Для следующего за r элемента предыдущим становится p (строка 11).</p>	
<p>Для p следующим становится r->next (строка 12).</p>	
<p>Для r следующим становится p (строка 14).</p>	
<p>Для p предыдущим становится r (строка 13).</p>	

Для добавления элемента перед элементом **r**, нужно в функции `insert_after()` поменять местами `prev` и `next`, а также **h** и **t**.

Рассмотрим теперь удаление элемента из списка.

Листинг 1.15.

```

1 void del_node (list *&h, list *&t, list *r){ //удаляем после r
2     if (r == h && r == t)                      //единственный элемент списка
3         h = t = NULL;
4     else if (r == h){                          //удаляем голову списка
5         h = h->next;                            //сдвигаем голову
6         h->prev = NULL;

```

```

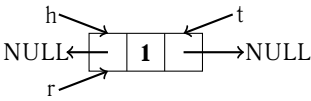
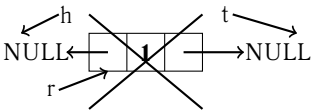
7   }
8   else if (r == t){                      //удаляем хвост списка
9       t = t->prev;                       //сдвигаем хвост
10      t->next = NULL;
11  }
12  else{
13      r->next->prev = r->prev;             //для следующего от r предыдущим
        становится r->prev
14      r->prev->next = r->next;             //для предыдущего от r следующим
        становится r->next
15  }
16  delete r;                             //удаляем r
17  }

```

Как видно из приведенного листинга, возможно четыре варианта удаления элемента из списка:

1. Удаление единственного элемента.
2. Удаление первого элемента списка.
3. Удаление концевго элемента списка.
4. Удаление элемента из середины списка.

Рассмотрим удаление единственного элемента списка:

Удаляем элемент, на который указывает <i>r</i>	
<i>h</i> и <i>t</i> обнуляем (строка 3) и удаляем <i>r</i> (строка 16).	

Рассмотрим теперь удаление первого элемента списка:

Удаляем элемент, на который указывает <code>r</code>	
Сдвигаем указатель на голову на следующий элемент (строка 5), обнуляем <code>h->prev</code> (строка 6) и удаляем <code>r</code> (строка 16).	

Рассмотрим удаление конечного элемента списка:

Удаляем элемент, на который указывает <code>r</code>	
Сдвигаем указатель на хвост на следующий элемент (строка 9), обнуляем <code>t->next</code> (строка 10) и удаляем <code>r</code> (строка 16).	

Рассмотрим теперь удаление из середины списка:

Удаляем элемент, на который указывает r	
Связываем r->prev и r->next. Предыдущим для r->next становится r->prev (строка 13).	
Связываем r->prev и r->next. Следующим для r->prev становится r->next (строка 14).	
Удаляем r (строка 16).	

Для очистки памяти необходимо удалить список:

Листинг 1.16.

```

1 void del_list( list *&h, list *&t){ //удаляем список
2     while (h){                      //пока список не пуст
3         list *p = h;                //указатель на голову
4         h = h->next;                 //переносим голову
5         h->prev = NULL;              //обнуляем
6         delete p;                   //удаляем p
7     }
8 }
```