[1. Динамические структуры данных C++. Однонаправленный список, очередь, стек, дек, двунаправленный список, двунаправленный кольцевой список, дек на двунаправленном списке,](#_Toc515770989)

[1.1. Линейные однонаправленные списки](#_Toc515770990)

[1.1.1. Общие сведения](#_Toc515770991)

[1.1.2. Однонаправленные списки без заглавного звена](#_Toc515770992)

[1.1.3. Построение списка с заглавным звеном](#_Toc515770993)

[1.1.4. Удаление списка из памяти](#_Toc515770994)

[1.2. Операции над списками с заглавным звеном.](#_Toc515770995)

[1.2.1. Поиск звена](#_Toc515770996)

[1.2.2. Включение звена после звена (1-й случай)](#_Toc515770997)

[1.2.3. Включение звена с заданным информационным полем перед звеном (2-й случай)](#_Toc515770998)

[1.2.4. Удаление звена после звена (1-й случай)](#_Toc515770999)

[1.2.5. Удаление звена на которое указывает ссылка (2-й случай)](#_Toc515771000)

[1.3. Ортогональные списки](#_Toc515771001)

[1.3.1. Реализация операций над ортогональными списками](#_Toc515771002)

[1.4. Кольцевые списки](#_Toc515771003)

[1.4.1. Построение и вывод кольца](#_Toc515771004)

[1.4.2. Основные операции](#_Toc515771005)

[1.5. Списки магазинного типа](#_Toc515771006)

[1.6. Списки магазинного типа. Очереди](#_Toc515771007)

[1.6.1. Формирование очереди](#_Toc515771008)

[1.6.2. Добавление звена к очереди](#_Toc515771009)

[1.6.3. Удаление звена из очереди](#_Toc515771010)

[1.7. Стек](#_Toc515771011)

[1.7.1. Формирование стека](#_Toc515771012)

[1.7.2. Включение звена в стек](#_Toc515771013)

[1.7.3. Удаление звена из стека](#_Toc515771014)

[1.8. Дек](#_Toc515771015)

[1.9. Линейные двунаправленные списки](#_Toc515771016)

[1.9.1. Формирование линейного двунаправленного списка](#_Toc515771017)

[1.9.2. Проход по линейному двунаправленному списку, начиная с его начала](#_Toc515771018)

[1.9.3. Проход по линейному двунаправленному списку, начиная с его конца](#_Toc515771019)

[1.9.4. Поиск звена в двунаправленном списке, начиная с начала списка](#_Toc515771020)

[1.9.5. Поиск звена в двунаправленном списке, начиная с конца списка](#_Toc515771021)

[1.9.6. Вставка звена в двунаправленный список (1-й случай)](#_Toc515771022)

[1.9.7. Вставка звена в двунаправленный список (2-й случай)](#_Toc515771023)

[1.9.8. Удаление звена из двунаправленного списка. Указатель на удаляемое звено (1-й случай)](#_Toc515771024)

[1.9.9. Удаление звена из двунаправленного списка. После звена (2-й случай)](#_Toc515771025)

[1.10. Двунаправленные кольцевые списки](#_Toc515771026)

[1.11. Деки на базе двунаправленных списков](#_Toc515771027)

[1.11.1. Формирование дека и его просмотр](#_Toc515771028)

[1.11.2. Добавление звена в начало дека](#_Toc515771029)

[1.11.3. Добавление звена в конец дека](#_Toc515771030)

[1.11.4. Удаление звена из дека слева](#_Toc515771031)

[1.11.5. Удаление звена из дека справа](#_Toc515771032)

1. Динамические структуры данных C++. Однонаправленный список, очередь, стек, дек, двунаправленный список, двунаправленный кольцевой список, дек на двунаправленном списке,

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book\_sod/kgsu/oglav.html

## Линейные однонаправленные списки

### Общие сведения

На этом шаге мы приведем **общие сведения о списках**.

**Список** - это совокупность объектов, называемых элементами списка, в которой каждый объект содержит информацию о местоположении связанного с ним объекта [1, с. 214].

Если список располагается в оперативной памяти, то, как правило, информация для поиска следующего объекта - это адрес (указатель) в памяти. Если список хранится в файле на диске, то информация о следующем элементе может включать смещение элемента от начала файла, положение указателя записи/считывания файла, ключ записи и любую другую информацию, позволяющую однозначно отыскать следующий элемент списка.

Каждый элемент списка представим структурой языка C++ с двумя полями:

* **информационное поле**, которое в общем случае может содержать произвольное количество полей разных типов. Ясно, что если значением переменной p является ссылка на элемент списка, то присоединяя к обозначению (\*p) с помощью точки имя соответствующего поля, можно манипулировать со значением любого поля информационной части;
* **ссылка на следующий элемент списка**.

Каждую пару будем называть звеном, а ссылки, содержащиеся в каждом из звеньев, будем использовать для соединения звеньев в список. Такой способ представления упорядоченной последовательности звеньев называется **сцеплением**.

С учетом сказанного, мы можем описать звено списка так:

struct **node**

**{**

int **elem;** //Информационный элемент звена списка

**node \*sled;** // Указатель на следующее звено списка

**};**

Чтобы иметь возможность оперировать со списком как с единым объектом, введем в употребление статическую ссылочную переменную **phead**, которая указывает на первое звено списка и описывается следующим образом:

struct **node \*phead;**

### Однонаправленные списки без заглавного звена

На этом шаге мы рассмотрим **построение списка без заглавного звена**.

Приведем алгоритм построения однонаправленного списка без заглавного звена с помощью схем "до и после" [1]. В ниже приведенной схеме переменная **phead** - указатель на первый элемент списка, а **t** - указатель, содержащий адрес последнего (текущего) элемента списка.

1. Отведем место для указателей в статической памяти:

struct **node \*phead;**

struct **node \*t;**

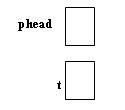


Рис.1. Размещение указателей

1. В куче резервируем место для динамического объекта:

**phead =** new **(node);**

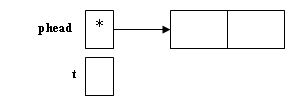


Рис.2. Размещение объекта в куче

1. Присвоим значение переменной **t**, и поместим в информационное поле значение элемента:

**t = phead;**

**(\*t).value = Элем;**

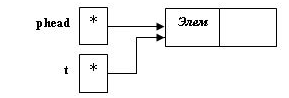


Рис.3. Определение t

1. Поместим в поле звена адрес нового динамического объекта, зарезервированного в куче:

**(\*t).next =** new **(node);**

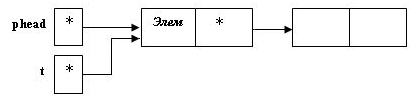


Рис.4. Добавление нового элемента

1. Переменная **t** должна содержать адрес последнего добавленного элемента:

**t = (\*t).next;**

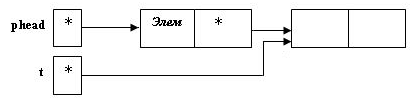


Рис.5. t содержит адрес последнего элемента

1. Если требуется завершить построение списка, то в поле указателя последнего элемента нужно поместить **NULL**:

**(\*t).next = NULL;**

**(\*t).value = Элем1;**

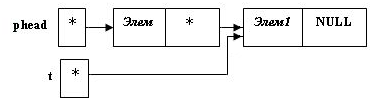


Рис.6. Результат выполнения операций

В результате построен линейный однонаправленный список без заглавного звена, содержащий два узла:



Рис.7. Конечный результат

Оформим алгоритм в виде программы на языке C++.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **value;**

**node \*next;**

};

void **main ()**

{

int **i;**

**node \*phead, \*t;**

**setlocale(LC\_ALL,**"Rus");

**phead =** new **(node);**

**t = phead;**

**(\*t).value = 1;**

**(\*t).next =** new **(node);**

**t = (\*t).next;**

**(\*t).value = 2;**

**(\*t).next =** new **(node);**

**t = (\*t).next;**

**(\*t).value = 6;**

**(\*t).next =** new **(node);**

**t = (\*t).next;**

**(\*t).value = 17;**

**(\*t).next =** new **(node);**

**(\*t).next = NULL;**

// Вывод содержимого информационных полей списка

for **(t = phead; t != NULL; t = (\*t).next)**

**cout << (\*t).value <<** " ";

**cout <<** "\n";

**system(**"PAUSE");

}



### Построение списка с заглавным звеном

На этом шаге мы рассмотрим **алгоритм построения списка с заглавным звеном**.

Чтобы сделать действия, выполняемые при включении элемента в список (исключении элемента из списка), единообразными, обычно применяется следующий прием. В начало каждого списка добавляется заглавное звено (**заголовок списка**). Оно никогда не исключается из списка и перед ним в список никогда не включаются новые элементы.

Информационная часть заглавного звена или не используется вовсе, или используется для специальных целей. Например, в случае списка целых чисел она может содержать число, равное количеству звеньев в списке. Добавление заглавного звена в список приводит к тому, что теперь у всех элементов, в том числе и у первого, имеется предшественник, и действия по включению новых элементов в список (или исключение элементов из списка) проводятся единым способом.

Приведем алгоритм построения однонаправленного списка с заглавным звеном с сохранением порядка поступления звеньев. Шаги построения алгоритма будем иллюстрировать с помощью схем "до и после" Д.Кнута.

1) Вначале отведем место для указателей в статической памяти:

struct **node \*phead;**

struct **node \*t;**

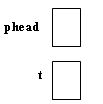


Рис.1. Резервирование места в памяти

2) В куче зарезервируем место для динамического объекта, на который указывает **phead**:

**phead =** new(node);

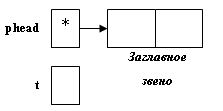


Рис.2. Выделили место для элемента списка

Построено **заглавное звено** будущего однонаправленного списка.

Выполним еще ряд подготовительных действий:

**t = phead;**

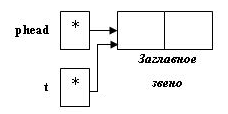


Рис.3. Присвоим значение t

**(\*t).sled = NULL;**

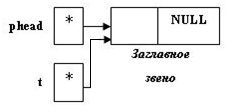


Рис.4. Присвоим указателю заглавного звена NULL

Теперь можно приступить к циклическому процессу построения списка. Идентификатор **Число** обозначает объект языка C++, вводимый с клавиатуры.

**cin >> Число;**

while **(Число != Числу, определяющему окончание ввода)**

**{**

**(\*t).sled =** new **(node);** //Резервируем место для нового объекта.

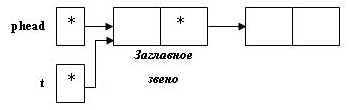


Рис.5. Создаем новый объект

**t = (\*t).sled;** //Указатель t содержит адрес

//расположения созданного объекта.

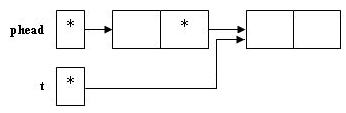


Рис.6. Определяем t

**(\*t).elem = Число;** //Заполняем поля объекта.

**(\*t).sled = NULL;**

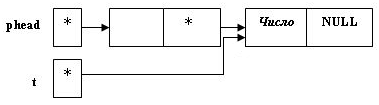


Рис.7. Результат заполнения полей

**cin >> Число;** //Запрос на ввод следующего значения.

**}**

Алгоритм построения оформим в виде функции на языке C++:

void **POSTROENIE (node \*\*phead)**

// Построение списка с заглавным звеном.

// \*phead - указатель на заглавное звено.

{

**node \*t;**

int **el;**

// Вначале создадим заглавное звено

**\*phead =** new **(node);**

**t = \*phead; (\*t).sled = NULL;**

**cout <<** "Вводите элементы звеньев списка: ";

**cin >> el;**

while **(el!=0)**

**{**

**(\*t).sled =** new **(node); t = (\*t).sled;**

**(\*t).elem = el; (\*t).sled = NULL;**

**cin >> el;**

**}**

}

Ясно, что пустой однонаправленный список с заглавным звеном можно схематически представить так:

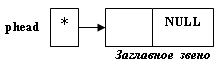


Рис.8. Пустой однонаправленный список с заглавным звеном

На следующем шаге мы рассмотрим удаление списка из памяти.

### Удаление списка из памяти

На этом шаге мы рассмотрим **алгоритм удаления списка из памяти**.

Остановимся теперь на удалении однонаправленного линейного списка из динамической памяти.

Алгоритм удаления можно сформулировать следующим образом:

* заводим два указателя **q** и **q1**, один из которых будет "опережать" другой (пусть **q1** опережает **q**);
* начальное значение **q** - это адрес расположения в памяти заглавного звена, а **q1** - адрес расположения первого элемента списка;
* организуем цикл, пока **q1 != NULL**, то есть пока **q1** "не доберется" до указателя последнего элемента списка;
* в теле цикла переместим указатели на следующую пару элементов списка (то есть для первого случая **q** будет содержать адрес первого звена списка, а **q1** - второго), и удалим элемент, который адресуется значением **q**;
* после выполнения цикла у нас останется только заглавное звено, адресуемое указателем **phead**. Его также нужно удалить.

Приведем текст функции, реализующей указанный алгоритм.

void **OCHISTKA (node \*\*phead)**

//Удаление однонаправленного списка из памяти.

// \*phead - указатель на заглавное звено списка.

{

struct **node \*q, \*q1;** // Рабочие указатели.

**q = \*phead;**

**q1 = (\*q).sled;** // Указатель q1 "опережает" указатель q.

while **(q1!=NULL)**

**{ q = q1; q1 = (\*q1).sled;** delete **q;}**

delete **\*phead;** //Удаление заглавного звена.

}

Отметим, что одной из распространенных ошибок при составлении программ является неаккуратное обращение с динамически распределяемой памятью, в частности, "забывание" освободить ее после использования, что в конце концов может привести к аварийному завершению программы из-за нехватки оперативной памяти. Поиск же мест "засорения" оперативной памяти в отлаживаемых программах, занимающих значительный объем, может занять много места, поскольку к моменту комплексной отладки, когда эти ошибки попадаются на глаза, естественным образом забываются многие детали создания программных компонентов. Поэтому целесообразно с самого начала отладки программ отслеживать процесс возврата динамической памяти в "кучу".

Соберем рассмотренные функции в объектно-ориентированную программу.

Пример. Построение однонаправленного списка с заглавным звеном и его просмотр.

#include<iostream>

usingnamespace **std;**

class **Spisok**

{

private:

struct **node**

**{**

int **elem;**

**node \*sled;**

**} \*phead;** //Указатель на начало списка.

public:

**Spisok() {phead =** new **(node); (\*phead).sled=NULL;}** //Конструктор.

**~Spisok() {** delete **phead; }** //Деструктор.

void **POSTROENIE ();**

void **VYVOD ();**

void **OCHISTKA ();**

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

**A.POSTROENIE ();**

**A.VYVOD ();**

**A.OCHISTKA ();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::POSTROENIE ()**

//Построение однонаправленного списка с заглавным звеном.

// phead - указатель на заглавное звено списка.

{

**node \*t;**

int **el;**

**t = phead;**

**cout <<** "Вводите элементы списка: ";

**cin >> el;**

while **(el!=0)**

**{**

**(\*t).sled =** new **(node);**

**t = (\*t).sled; (\*t).elem = el; (\*t).sled = NULL;**

**cin >> el;**

**}**

}

void **Spisok::VYVOD ()**

//Вывод содержимого однонаправленного линейного списка

//с заглавным звеном.

// phead - указатель на заглавное звено списка.

{

**node \*t;**

**t = (\*phead).sled;**

**cout <<** "Список: ";

while **(t != NULL)**

**{**

**cout << (\*t).elem <<** " ";

**t = (\*t).sled;**

**}**

**cout << endl;**

}

void **Spisok::OCHISTKA ()**

//Удаление однонаправленного списка из памяти.

// phead - указатель на заглавное звено списка.

{

**node \*q, \*q1;**// Рабочие указатели.

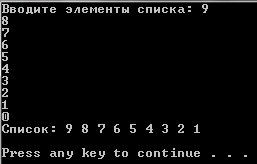
**q = phead;**

**q1 = (\*q).sled;** // Указатель q1 "опережает" указатель q.

while **(q1 != NULL)**

**{ q = q1; q1 = (\*q1).sled;** delete **q;}**

}



Со следующего шага мы начнем рассматривать основные операции над списками с заглавным звеном.

## Операции над списками с заглавным звеном.

### Поиск звена

На этом шаге мы приведем **алгоритм поиска звена**.

Приведем алгоритм последовательного поиска звена с заданным значением информационного поля в однонаправленном списке, записанный в виде функции языка C++:

void **POISK (node \*\*phead,** int **el, node \*\*Res)**

// Поиск звена с элементом el в списке, заданном указателем \*phead.

// В случае успешного поиска в \*Res находится адрес

// звена списка, содержащего элемент el, в случае неуспеха в \*Res помещается NULL.

{

**node \*t;**

**\*Res = NULL;**

**t = \*phead; t = (\*t).sled;**

while **(t != NULL && \*Res == NULL)**

if **((\*t).elem == el) \*Res = t;** else **t = (\*t).sled;**

}

Со следующего шага мы начнем рассматривать **алгоритмы включения звена в список**.

### Включение звена после звена (1-й случай)

На этом шаге мы рассмотрим **первый случай включения звена в список**.

На этом шаге мы рассмотрим **алгоритм включения звена в список после звена, на которое указывает заданная ссылка**.

Предположим, что имеется однонаправленный список с заглавным звеном, и необходимо вставить звено с заданным информационным полем после звена, на которое указывает ссылка **Res**. Вставка звена осуществляется по следующему алгоритму.

1. В куче резервируется место для динамического объекта:

**q =** new **(node);**

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris6_1.jpg

Рис.1. Зарезервировали место под элемент списка

В информационное поле этого объекта помещается значение элемента, который необходимо вставить:

**(\*q).elem = el;**

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris6_2.jpg

Рис.2. Заполнение информационного поля

1. В поле указателя помещается адрес элемента, следующего за звеном, на которое указывает **Res**:

**(\*q).sled = (\*Res).sled;**

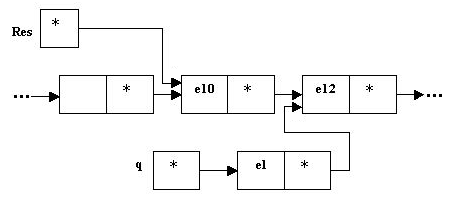


Рис.3. Заполнение поля указателя вставляемого элемента

И, наконец, после выполнения оператора

**(\*Res).sled = q;**

получаем результат, изображенный на следующей схеме:

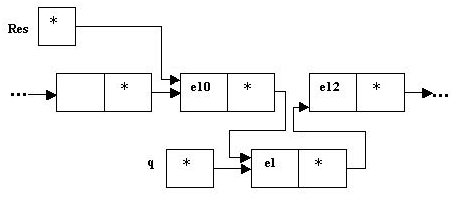


Рис.4. Изменение указателя у предыдущего элемента

Алгоритм оформим в виде функции.

void **VSTAV (node \*\*Res,** int **el)**

// Включение звена с информационным полем el

//после звена, на которое указывает ссылка \*Res.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = el; (\*q).sled = (\*\*Res).sled;**

**(\*\*Res).sled = q;**

**}**

На следующем шаге мы рассмотрим **второй случай алгоритма включения в список звена**.

### Включение звена с заданным информационным полем перед звеном (2-й случай)

На этом шаге мы рассмотрим второй случай включения звена и приведем пример программы, реализующей рассмотренные действия.

На этом шаге мы рассмотрим **алгоритм включения в однонаправленный список звена с заданным информационным полем перед звеном, на которое указывает заданная ссылка Res**.

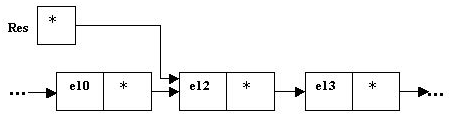


Рис.1. Начальное положение

1. Опишем алгоритм при помощи схем Д.Кнута "до и после":

**q =** new **(node);**

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris7_2.jpg

Рис.2. Выделение места в памяти для нового звена

**(\*q).elem = (\*Res).elem;**

**(\*q).sled = (\*Res).sled;**

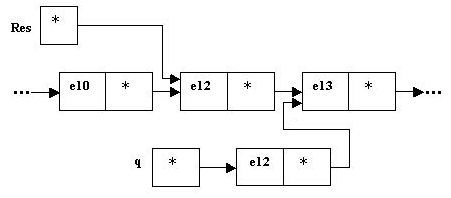


Рис.3. Заполнение полей нового элемента

**(\*Res).elem = el;**

**(\*Res).sled = q;**

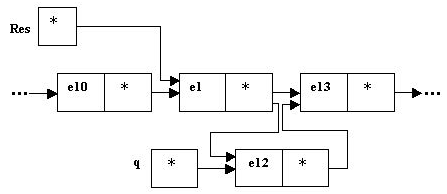


Рис.4. Включение элемента в список

Приведем функцию, реализующую описанное включение:

void **VSTAV1 (node \*\*Res,** int **el)**

//Включение звена с информационным полем el

//перед звеном, на которое указывает \*Res.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = (\*\*Res).elem; (\*q).sled = (\*\*Res).sled;**

**(\*\*Res).elem = el; (\*\*Res).sled = q;**

}

Пример. Включение звена с заданным элементом в однонаправленный список с заглавным звеном.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **elem;**

**node \*sled;**

};

class **Spisok**

{

private:

**node \*phead;** //Указатель на заглавное звено списка.

**node \*Res;** //Указатель на найденное звено списка.

public:

**Spisok() {phead=**new(node); (\*phead).sled=NULL; Res=NULL;}

**~Spisok() {** delete **phead; }**

void **POSTROENIE();**

void **VYVOD();**

**node \*POISK(**int);

void **VSTAV(**int);

void **VSTAV1(**int);

void **OCHISTKA();**

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

int **el,el1;**

**A.POSTROENIE ();**

**A.VYVOD ();**

**cout<<**"\nВведите элемент звена, после которого ";

**cout<<**"осуществляется вставка:\n";

**cin>>el;**

**cout<<**"Введите элемент вставляемого звена:\n";

**cin>>el1;**

if **(A.POISK(el)!=NULL)**

**{**

**A.VSTAV (el1); A.VYVOD ();**

**}**

else **cout<<**"Звена с заданным элементом в списке нет!";

**cout<<**"\nВведите элемент звена, перед которым ";

**cout<<**"осуществляется вставка:\n";

**cin>>el;**

**cout<<**" Введите элемент вставляемого звена:\n";

**cin>>el1;**

if **(A.POISK (el)!=NULL)**

**{**

**A.VSTAV1 (el1); A.VYVOD ();**

**}**

else **cout<<**" Звена с заданным элементом в списке нет!";

**A.OCHISTKA();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::POSTROENIE ()**

// Построение однонаправленного списка с заглавным звеном.

// phead - указатель на заглавное звено списка

{

**node \*t;**

int **el;**

**t = phead;**

**cout<<**"Вводите элементы звеньев списка: ";

**cin>>el;**

while **(el!=0)**

**{**

**(\*t).sled =** new **(node);**

**t = (\*t).sled; (\*t).elem = el; (\*t).sled = NULL;**

**cin>>el;**

**}**

}

void **Spisok::VYVOD ()**

// Вывод содержимого однонаправленного списка с

// заглавным звеном. phead - указатель на заглавное звено списка.

{

**node \*t;**

**t =(\*phead).sled;**

**cout<<**"Список: ";

while **(t!=NULL)**

**{**

**cout<<(\*t).elem<<**" "; t = (\*t).sled;

**}**

**cout<<endl;**

}

node \*Spisok::POISK (int **el)**

//Поиск звена с элементом el в списке, заданном указателем

//phead. В случае успешного поиска в Res находится адрес

//звена списка, содержащего элемент el, в случае неуспеха

//в Res помещается NULL.

{

**node \*t;**

**Res = NULL; t = (\*phead).sled;**

while **(t!=NULL && Res==NULL)**

if **((\*t).elem==el)**

**Res = t;**

else **t = (\*t).sled;**

return **Res;**

}

void **Spisok::VSTAV (**int **el)**

// Включение звена с информационным полем el

// после звена, на которое указывает ссылка Res.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

**(\*q).sled = (\*Res).sled; (\*Res).sled = q;**

}

void **Spisok::VSTAV1 (**int **el)**

// Включение звена с информационным полем el

// перед звеном, на которое указывает Res.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = (\*Res).elem; (\*q).sled = (\*Res).sled;**

**(\*Res).elem = el; (\*Res).sled = q;**

}

void **Spisok::OCHISTKA ()**

//Удаление однонаправленного списка из памяти.

// phead - указатель на заглавное звено списка.

{

**node \*q,\*q1;**// Рабочие указатели.

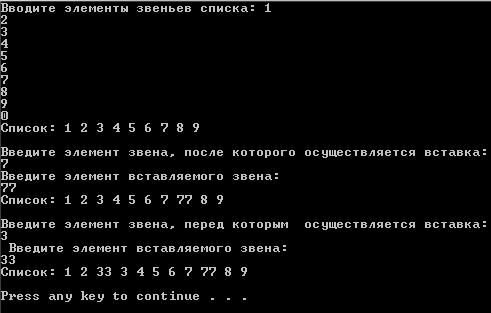
**q = phead;**

**q1 = (\*q).sled;** // Указатель q1 "опережает" указатель q.

while **(q1!=NULL)**

**{ q = q1; q1 = (\*q1).sled;** delete **q;}**

}



Со следующего шага мы начнем рассматривать **удаление звена из списка**.

### Удаление звена после звена (1-й случай)

На этом шаге мы рассмотрим **один из алгоритмов удаления звена**.

На этом шаге мы рассмотрим **алгоритм удаления звена, расположенного после звена, на которое указывает ссылка Res**.

Воспользуемся схемами "до и после":

**q = (\*Res).sled;**

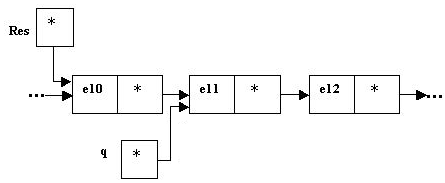


Рис.1. Определение местоположения удаляемого звена.

1. Проверяем, не является ли звено, после которого нужно удалять, последним. В этом случае удалять нечего.

if **(q!=NULL)** //Если звено, после которого нужно удалять,

// не является последним, то...

**{**

**(\*Res).sled = (\*(\*Res).sled).sled;**

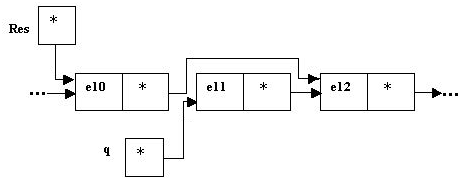


Рис.2. Исключение удаляемого элемента из списка

1. Последняя тонкость: присоединение "кусочка" **heap**-области к списку свободной памяти:

delete **q;**

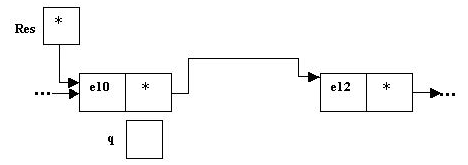


Рис.3. Возврат памяти в кучу

**}**

Приведем текст функции удаления:

void **YDALE (node \*\*Res)**

// Удаление звена, расположенного после

// звена, на которое указывает ссылка \*Res.

{

**node \*q;**

**q = (\*\*Res).sled;**

if **(q!=NULL)**

// Если звено, после которого нужно удалять,

// не является последним, то...

**{ (\*\*Res).sled = (\*(\*\*Res).sled).sled;** delete **q; }**

else

**cout<<**"Звено с заданным элементом - последнее!\n";

}

На следующем шаге мы продолжим рассматривать алгоритмы удаления звена.

### Удаление звена на которое указывает ссылка (2-й случай)

На этом шаге мы рассмотрим **другой случай удаления звена**.

На этом шаге мы рассмотрим **алгоритм удаления звена, на которое указывает ссылка Res**.

Воспользуемся схемами "до и после":

1. Определяем местоположение следующего за удаляемым элемента.

**q = (\*Res).sled;**

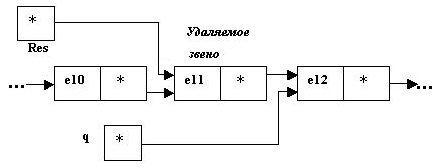


Рис.1. Сохранение адреса следующего элемента

1. Определяем, не является ли удаляемое звено последним. В зависимости от этого реализация алгоритма будет различной.

if **(q!=NULL)**

**{**//Если удаляемое звено не является последним, то ...

**(\*Res).elem = (\*q).elem;**

**(\*Res).sled = (\*q).sled;**

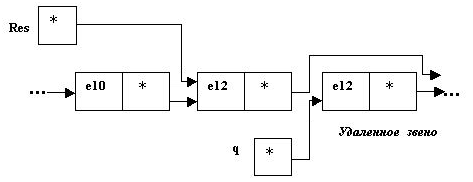


Рис.2. "Переписываем" следующий элемент в удаляемый

1. Последняя тонкость: присоединение "кусочка" **heap**-области к списку свободной памяти:

delete **q;**

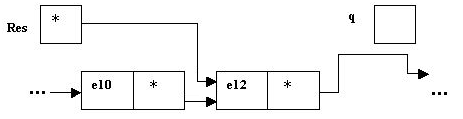


Рис.3. Возврат памяти в кучу

**}**

1. Теперь рассмотрим ситуацию, когда удаляемое звено является последним звеном списка:

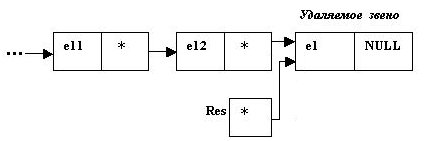


Рис.4. Удаляемый элемент – последний

Найдем указатель на предпоследнее звено линейного списка с помощью двух вспомогательных указателей **q1** и **q2**, перемещающихся по списку "параллельно друг другу", причем указатель **q2** "опережает" указатель **q1** на один шаг.

**q1 = phead; q2 = (\*q1).sled;** //Инициализация указателей.

while **(q2!=Res)**

**{ q1 = q2; q2 = (\*q2).sled;}**

После выполнения цикла **while**, мы получим следующую ситуацию:

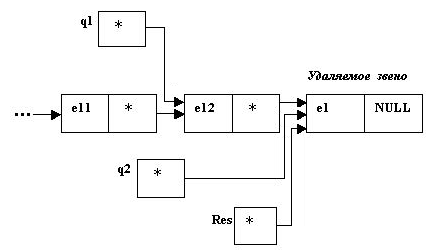


Рис.5. Положение указателей

1. Приступим к удалению:

**(\*q1).sled = NULL; q2 = NULL;** delete **(Res);**

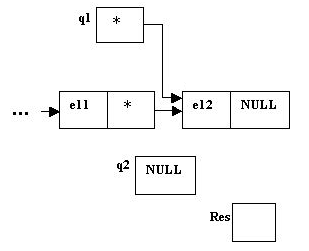


Рис.6. После удаления

Приведем текст функции удаления:

void **YDALE1 (node \*\*phead, node \*\*Res)**

//Удаление звена, на которое указывает ссылка \*Res

//из однонаправленного списка с заглавным звеном.

//\*phead - указатель на заглавное звено списка.

{

**node \*q,\*q1,\*q2;**

**q = (\*\*Res).sled;**

if **(q!=NULL)**

**{ (\*\*Res).elem = (\*q).elem; (\*\*Res).sled = (\*q).sled;**delete **q; }**

else

**{ q1 = \*phead; q2 = (\*q1).sled;** //Инициализация указателей.

while **(q2!=\*Res)**

**{ q1 = q2; q2 = (\*q2).sled; }**

**(\*q1).sled = NULL; q2 = NULL;** delete **\*Res;**

**}**

}

Приведем пример программы, осуществляющей **удаление звена из однонаправленного списка с заглавным звеном**.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **elem;**

**node \*sled;**

};

class **Spisok {**

private:

**node \*phead, \*Res;**

public:

**Spisok() {phead=**new(node);Res=NULL;}

**~Spisok() {**delete **phead;}**

void **POSTROENIE ();**

void **VYVOD ();**

**node \*POISK (**int);

void **YDALE ();**

void **YDALE1();**

void **OCHISTKA();**

**};**

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

int **el;**

**node \*Res\_Zn;**

**A.POSTROENIE ();**

**A.VYVOD ();**

**cout<<**"\nВведите элемент звена, после которого ";

**cout<<**"осуществляется удаление:\n";

**cin>>el;**

**Res\_Zn=A.POISK (el);**

if **(Res\_Zn!=NULL && (\*Res\_Zn).sled!=NULL)**

**{A.YDALE (); A.VYVOD ();}**

else **cout<<**"Звена с заданным элементом в списке нет!";

**cout<<**"\nВведите удаляемый элемент звена:\n";

**cin>>el;**

**Res\_Zn=A.POISK (el);**

if **(A.POISK (el)!=NULL)**

**{**

**A.YDALE1 (); A.VYVOD (); cout<<endl;**

**}**

else **cout<<**"Звена с заданным элементом в списке нет!";

**A. OCHISTKA();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::POSTROENIE ()**

//Построение однонаправленного списка с заглавным звеном.

//phead -указатель на заглавное звено

{

**node \*t;**

int **el;**

**t = phead; (\*t).sled = NULL;**

**cout<<**"Вводите элементы звеньев списка: ";

**cin>>el;**

while **(el!=0)**

**{**

**(\*t).sled =** new **(node);**

**t = (\*t).sled; (\*t).elem = el; (\*t).sled = NULL;**

**cin>>el;**

**}**

}

void **Spisok::VYVOD ()**

//Вывод содержимого однонаправленного списка.

//phead - указатель на заглавное звено.

{

**node \*t;**

**t = phead; t = (\*t).sled;**

**cout<<**"Список: ";

while **(t!=NULL)**

**{**

**cout<<(\*t).elem <<**" ";

**t = (\*t).sled;**

**}**

}

**node \*Spisok::POISK (**int **el)**

//Поиск звена с элементом el в списке, заданном указателем phead.

//В случае успешного поиска в Res находится адрес искомого звена

//списка. В противном случае Res содержит NULL.

{

**node \*t;**

**Res = NULL; t = phead; t = (\*t).sled;**

while **(t!=NULL && Res==NULL)**

if **((\*t).elem==el) Res = t;**

else **t = (\*t).sled;**

return **Res;**

}

void **Spisok::YDALE ()**

//Удаление звена, расположенного после звена,

//на которое указывает ссылка Res.

{

**node \*q;**

**q = (\*Res).sled;**

if **(q!=NULL)**

**{**

//Если звено, после которого нужно удалять,

//не является последним, то:

**(\*Res).sled = (\*(\*Res).sled).sled;** delete **q;**

**}**

else

**cout<<**"Звено с заданным элементом - последнее!\n";

}

void **Spisok::YDALE1 ()**

//Удаление звена, на которое указывает ссылка Res.

{

**node \*q,\*q1,\*q2;**

**q = (\*Res).sled;**

if **(q!=NULL)**

**{**

**(\*Res).elem = (\*q).elem; (\*Res).sled = (\*q).sled;**

delete **q;**

**}**

else

**{**

**q1 = phead; q2 = (\*q1).sled;**

while **(q2!=Res)**

**{**

**q1 = q2; q2 = (\*q2).sled;**

**}**

**(\*q1).sled = NULL; q2 = NULL;** delete **Res;**

**}**

}

void **Spisok::OCHISTKA ()**

//Удаление однонаправленного списка из памяти.

// phead - указатель на заглавное звено списка.

{

**node \*q,\*q1;**// Рабочие указатели.

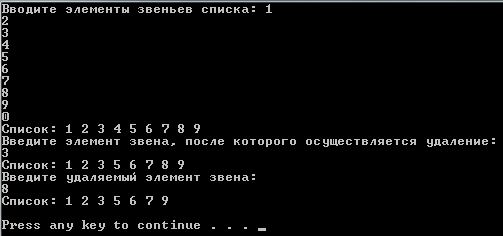
**q = phead;**

**q1 = (\*q).sled;** // Указатель q1 "опережает" указатель q.

while **(q1!=NULL)**

**{ q = q1; q1 = (\*q1).sled;** delete **q;}**

}



Со следующего шага мы начнем рассматривать **ортогональные списки**.

## Ортогональные списки

На этом шаге мы рассмотрим **структуру ортогональных списков**.

До сих пор мы рассматривали **линейные структуры динамических переменных**. Добавление к динамической переменной двух и более полей указателей создает возможность построения **нелинейных структур**. Дело в том, что при решении практических задач обычно не удается обойтись только линейными динамическими структурами данных (**список, очередь, стек, дек** и т.д.): приходится создавать структуры данных, максимально отражающие существо выполняемой исполнителем задачи.

Мы рассмотрим только простейшие нелинейные динамические структуры, которые называются **ортогональными списочными структурами (ортогональными списками, многосвязными списками)**.

Более точно, **ортогональными списками** называется списочная структура данных, в которой узлы могут принадлежать более чем одному списку и содержать более одного указателя.

На рисунке изображено графическое представление ортогональных списков:

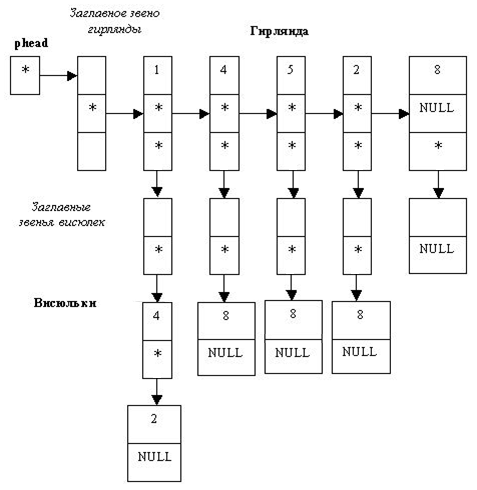


Рис.1. "Гирлянды" и "висюльки"

Горизонтальный линейный однонаправленный список с заглавным звеном мы будем называть **гирляндой**. Каждое звено этого списка содержит три поля, причем, если указатель **P** указывает на звено гирлянды, то:

* поле **(\*P).Key** является **информационным полем** узла гирлянды;
* поле **(\*P).Next** содержит **указатель на следующее звено** гирлянды;
* поле **(\*P).Trail** содержит **указатель на линейный однонаправленный список с заглавным звеном**, который называется **висюлькой** (английское слово **Trail** переводится как "тащиться, свисать, волочиться").

Звено каждой висюльки содержит два поля: **Id** и **Next**, причем, если указатель Q указывает на элемент висюльки, то:

поле **(\*Q).Id** является **информационным полем** звена висюльки;

поле **(\*Q).Next** **указывает на следующее звено** данной висюльки.

На следующем шаге мы приведем **реализацию простейших операций над ортогональными списками**.

### Реализация операций над ортогональными списками

На этом шаге мы приведем **пример программы, реализующей основные операции над ортогональными списками**.

Приведем реализацию на языке C++ простейших операций над ортогональными списками ("гирляндо-висюлечной" структурой).

Сначала опишем типы данных:

// Описание типа звена гирлянды.

struct **nodeGir**

{

int **elem;** // Информационное поле звена гирлянды.

nodeVis \*vniz; // Указатель на звено висюльки.

nodeGir \*sled; // Указатель на звено гирлянды.

};

// Описание типа звена висюльки.

struct **nodeVis**

{

int **elem;** // Информационное поле звена висюльки.

nodeVis \*vniz; // Указатель на звено висюльки.

};

Приведем программу, демонстрирующую работу с "гирляндо-висюлечной" структурой.

#include<iostream>

usingnamespace **std;**

struct **nodeVis**

{

int **elem;** //Информационное поле звена висюльки.

**nodeVis \*vniz;**//Указатель на звено висюльки.

};

struct **nodeGir**

{

int **elem;**//Информационное поле звена гирлянды.

**nodeVis \*vniz;**//Указатель на звено висюльки.

**nodeGir \*sled;**//Указатель на звено гирлянды.

};

class **GirVis {**

private:

**nodeGir \*phead;**//Голова гирлянды.

**nodeVis \*pheadVis;**//Голова висюльки.

void **VisVyvod ();**

public:

**GirVis() {phead =** new **(nodeGir); }**

**~GirVis() {**delete **phead;}**

**nodeVis \*VisPostr ();**

**nodeVis\* VisPoisk (**int);

void **SetpheadVis (nodeVis \*r) {pheadVis=r;}** //Определение головы висюльки.

void **VisVstav (nodeVis \*,**int);

void **Vis1Vstav (nodeVis \*,**int);

void **VisUdale (nodeVis \*);**

void **Vis1Udale (nodeVis \*);**

void **GirPostr ();**

void **GirVyvod ();**

**nodeGir \*GirPoisk (**int);

void **OCHISTKA();**

void **OCHISTKA1();**

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**GirVis A;**

int **el,elGir,elVis;**

**nodeGir \*Res;** //Рабочий указатель.

**nodeVis \*ResVis;** //Указатель на звено висюльки.

**A.GirPostr ();**

**A.GirVyvod ();**

**cout<<**"\nВведите элемент звена гирлянды, ";

**cout<<**"чьи висюльки будем изменять:\n";

**cin>>elGir;**

**cout<<**"\nВведите элемент звена висюльки, после которого ";

**cout<<**"осуществляется вставка:\n";

**cin>>elVis;**

**cout<<**"\nВведите вставляемый элемент:\n";

**cin>>el;**

//Поиск элемента elGir в гирлянде.

**Res=A.GirPoisk (elGir);**

if **(Res!=NULL)**

**{**

//Поиск элемента elVis в висюльке.

**A.SetpheadVis((\*Res).vniz);**

**ResVis=A.VisPoisk (elVis);**

if **(ResVis!=NULL)**

**A.VisVstav (ResVis,el);**

else **cout<<**"Элемента в висюльке нет!\n";

**}**

else **cout<<**"Элемента в гирлянде нет\n";

**A.GirVyvod ();**

**cout<<**"\nВведите элемент гирлянды, чью висюльку будем изменять:\n";

**cin>>elGir;**

**cout<<**"Введите элемент висюльки, перед которым ";

**cout<<**"осуществляется вставка:\n";

**cin>>elVis;**

**cout<<**"Введите вставляемый элемент:\n";

**cin>>el;**

//Поиск элемента elGir в гирлянде.

**Res=A.GirPoisk (elGir);**

if **(Res!=NULL)**

**{**

//Поиск элемента elVis в висюльке.

**A.SetpheadVis((\*Res).vniz);**

**ResVis=A.VisPoisk (elVis);**

if **(ResVis!=NULL)**

**A.Vis1Vstav (ResVis,el);**

else **cout<<**"Элемента в висюльке нет!\n";

**}**

else **cout<<**"Элемента в гирлянде нет!\n";

**A.GirVyvod ();**

**cout<<**"\nВведите элемент гирлянды, чью висюльку будем изменять:\n";

**cin>>elGir;**

**cout<<**"Введите элемент висюльки, после которого нужно удалить:\n";

**cin>>elVis;**

//Поиск элемента elGir в гирлянде.

**Res=A.GirPoisk (elGir);**

if **(Res!=NULL)**

**{**

//Поиск элемента elVis в висюльке.

**A.SetpheadVis((\*Res).vniz);**

**ResVis=A.VisPoisk (elVis);**

if **((ResVis!=NULL) && ((\*ResVis).vniz!=NULL))**

**A.VisUdale (ResVis);**

else **cout<<**"Элемента в висюльке нет!\n";

**}**

else **cout<<**"Элемента в гирлянде нет!\n";

**A.GirVyvod ();**

**cout<<**"\nВведите элемент гирлянды, чью висюльку будем изменять:\n";

**cin>>elGir;**

**cout<<**"Введите элемент висюльки, который удаляется:\n";

**cin>>elVis;**

//Поиск элемента elGir в гирлянде.

**Res=A.GirPoisk (elGir);**

if **(Res!=NULL)**

**{**

//Поиск элемента elVis в висюльке.

**A.SetpheadVis((\*Res).vniz);**

**ResVis=A.VisPoisk (elVis);**

if **((ResVis!=NULL) && ((\*ResVis).vniz!=NULL))**

**A.Vis1Udale (ResVis);**

else **cout<<**"Элемента в висюльке нет или он последний!\n";

**}**

else **cout<<**"Элемента в гирлянде нет!\n";

**A.GirVyvod ();**

**A.OCHISTKA();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **GirVis::OCHISTKA()**

{

**nodeGir \*q,\*q1;**//Рабочие указатели.

**q = phead;**

**q1 = (\*q).sled;** //Указатель q1 "опережает" указатель q.

while **(q1!=NULL)**

**{ q = q1; q1 = (\*q1).sled;**

**pheadVis=(\*q).vniz;**

**OCHISTKA1();** //Очистка висюльки.

delete **q;}**

}

void **GirVis::OCHISTKA1()**

{

**nodeVis \*q,\*q1;**

**q=pheadVis;**

**q1 = (\*q).vniz;**

while **(q1!=NULL)**

**{ q = q1; q1 = (\*q1).vniz;**

delete **q;}**

}

void **GirVis::GirPostr ()**

//Построение однонаправленного списка с заглавным звеном,

//заданного указателем phead (построение гирлянды).

{

**nodeGir \*t;**

int **el;**

**t = phead; (\*t).sled = NULL;**

**cout<<**"Вводите элемент гирлянды: \n";

**cin>>el;**

while **(el!=0)**

**{**

**(\*t).sled =** new **(nodeGir);**

**t = (\*t).sled; (\*t).elem = el; (\*t).sled = NULL;**

**(\*t).vniz=VisPostr();**

**cout<<**" Вводите элемент гирлянды: \n";

**cin>>el;**

**}**

}

nodeVis \*GirVis::VisPostr ()

//Построение однонаправленного списка с заглавным звеном

//(построение висюльки). pheadVis - указатель на висюльку.

{

**nodeVis \*t;**

int **el;**

//Создадим заглавное звено списка.

**pheadVis =** new **(nodeVis);**

**t = pheadVis; (\*t).vniz = NULL;**

**cout<<**"Вводите элементы звеньев висюльки: \n";

**cin>>el;**

while **(el!=0)**

**{**

**(\*t).vniz =** new **(nodeVis);**

**t = (\*t).vniz; (\*t).elem = el; (\*t).vniz = NULL;**

**cin>>el;**

**}**

return **pheadVis;**

}

void **GirVis::GirVyvod ()**

//Вывод содержимого однонаправленного списка, заданного

//указателем phead (вывод содержимого гирлянды).

{

**nodeGir \*t;**

**t = phead; t = (\*t).sled;**

**cout<<**"Гирлянда: ";

while **(t!=NULL)**

**{**

**cout<<(\*t).elem<<**" ";

**pheadVis=(\*t).vniz;**

**VisVyvod ();**

**t = (\*t).sled;**

**}**

}

nodeGir \*GirVis::GirPoisk (int **el)**

//Поиск элемента el в списке, заданном указателем phead.

//В случае успешного поиска возвращается адрес звена списка,

//содержащего элемент el. В противном случае - NULL.

{

**nodeGir \*t,\*r;**

**r = NULL; t = phead; t = (\*t).sled;**

while **(t!=NULL && r==NULL)**

if **((\*t).elem==el) r = t;**

else **t = (\*t).sled;**

return **r;**

}

void **GirVis::VisVyvod ()**

//Вывод содержимого однонаправленного списка с заглавным звеном,

//заданного указателем pheadVis (вывод содержимого висюльки).

{

**nodeVis \*t;**

**t = pheadVis; t = (\*t).vniz;**

**cout<<**"(";

while **(t!=NULL)**

**{**

**cout<<(\*t).elem<<**" "; t = (\*t).vniz;

**}**

**cout<<**")";

}

nodeVis \*GirVis::VisPoisk (int **el)**

//Поиск элемента el в списке, заданном указателем pheadVis.

//В случае успешного поиска возвращается адрес звена списка,

//содержащего элемент el. В противном случае - NULL.

{

**nodeVis \*t,\*r;**

**r = NULL; t = pheadVis; t = (\*t).vniz;**

while **(t!=NULL && r==NULL)**

if **((\*t).elem==el) r = t;**

else **t = (\*t).vniz;**

return **r;**

}

void **GirVis::VisVstav (nodeVis \*r,**int **el)**

//Включение звена с информационным полем el

//после звена, на которое указывает r

//(включение звена в висюльку).

{

**nodeVis \*q;**

**q =** new **(nodeVis);**

**(\*q).elem = el; (\*q).vniz = (\*r).vniz; (\*r).vniz = q;**

}

void **GirVis::Vis1Vstav (nodeVis \*r,**int **el)**

//Включение звена с информационным полем el

//перед звеном, на которое указывает r

//(включение звена в висюльку).

{

**nodeVis \*q;**

**q =** new **(nodeVis);**

**(\*q).elem = (\*r).elem; (\*q).vniz = (\*r).vniz;**

**(\*r).elem = el; (\*r).vniz = q;**

}

void **GirVis::VisUdale (nodeVis \*r)**

//Удаление звена, расположенного после звена,

//на которое указывает ссылка r

//(удаление звена висюльки).

{

**nodeVis \*q;**

**q = (\*r).vniz;**

if **((\*r).vniz!=NULL)**

**{**

**(\*r).vniz = (\*(\*r).vniz).vniz;** delete **q;**

**}**

else **cout<<**"Звено с заданным элементом - последнее!\n";

}

void **GirVis::Vis1Udale (nodeVis \*r)**

//Удаление звена, на которое указывает ссылка r

//(удаление звена висюльки).

{

**nodeVis \*g;**

if **((\*r).vniz!=NULL)**

**{**

**g = (\*r).vniz;**

**(\*r).elem = (\*(\*r).vniz).elem;**

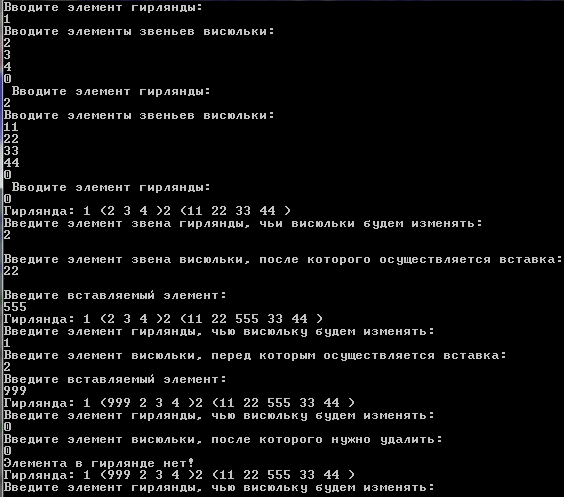
**(\*r).vniz = (\*(\*r).vniz).vniz;**

delete **g;**

**}**

else **cout<<**"Не умею удалять последнее звено!\n";

}



Со следующего шага мы начнем знакомиться с **кольцами**.

## Кольцевые списки

### Построение и вывод кольца

На этом шаге мы рассмотрим **построение и вывод кольца**.

Хотя структура в виде линейного списка является весьма полезной, у нее имеется ряд недостатков. Сейчас мы рассмотрим другие методы организации списков и использование их с целью устранения этих недостатков.

Один из **недостатков** линейных списков заключается в том, что, **зная указатель p на звено списка, мы не имеем доступа к предшествующим ему звеньям**. Если производится просмотр списка, то для повторного обращения к нему исходный указатель на начало списка должен быть сохранен.

Предположим теперь, что в структуре линейного списка было сделано изменение, при котором поле **sled** последнего элемента содержит указатель "**назад**" или на заглавное звено, или на элемент, следующий за заглавным звеном.

Под **кольцевым (циклическим) списком** понимается список, в котором указатель из некоторой ячейки направлен на такое место в списке, откуда данная ячейка может быть достигнута снова [1]. Очевидно, что теперь мы можем из любого звена списка, "перемещаясь" по указателям достичь любого другого звена.

**Кольцевым списком** (кольцом) на базе линейного однонаправленного списка называется линейный список, в котором указатель из некоторого звена направлен на такое звено в списке, из которого данное звено может быть достигнуто вновь.

Опишем **два** **способа представления однонаправленного кольцевого списка с заглавным звеном**:

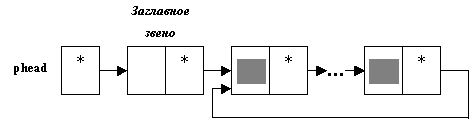


Рис.1. Кольцо с удаленным заглавным звеном

Такой кольцевой список будем называть **кольцевым списком с удаленным заглавным звеном**.

**Пустой кольцевой список с удаленным заглавным звеном** представим так:

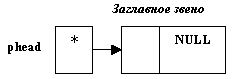


Рис.2. Пустое кольцо с удаленным заглавным звеном

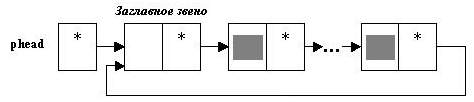


Рис.3. Кольцо с включенным заглавным звеном

А этот кольцевой список назовем **кольцевым списком с включенным заглавным звеном**.

**Пустой кольцевой список с включенным заглавным звеном** представим так:

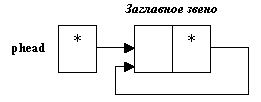


Рис.4. Пустое кольцо с включенным заглавным звеном

Рассмотрим способ **построения кольцевого списка**. Построение выполняется так же, как и в случае линейного однонаправленного списка с заглавным звеном, только после окончания ввода элементов кольцевого списка, в поле указателя последнего звена списка помещается адрес звена, следующего за заглавным.

Сказанное легко формализуется:

void **POSTROENIE (node \*\*phead)**

//Построение кольцевого списка с заглавным звеном.

//\*phead - указатель на заглавное звено.

{

int **el;**

struct **node \*t;**

// Вначале сформируем заглавное звено.

**\*phead =** new **(node);**

**t = \*phead; (\*t).sled = NULL;**

**cout<<**"Вводите элементы кольца: "; cin>>el;

while **(el!=0)**

**{ (\*t).sled =** new **(node); t = (\*t).sled; (\*t).elem = el;**

**cin>>el;**

**}**

**(\*t).sled = (\*(\*phead)).sled;**

}

**Вывод** на экран дисплея **содержимого информационных полей кольцевого списка** производится до тех пор, пока рабочий указатель, перемещающийся по кольцу, не совпадет с указателем на звено, расположенное после заглавного:

void **VYVOD (node \*\*phead)**

//Вывод содержимого кольцевого списка с удаленным

// заглавным звеном.

//\*phead - указатель на заглавное звено.

{

struct **node \*t;**

**t = (\*\*phead).sled; cout<<** "Кольцо: ";

if **(t!=NULL)**

**{ cout<<(\*t).elem; t = (\*t).sled;**

while **(t!=(\*\*phead).sled)**

**{**

**cout<<(\*t).elem;**

**t = (\*t).sled; }**

**}**

else **cout<<**"пусто!\n";

}

На следующем шаге мы перечислим **основные операции над кольцами**.

### Основные операции

На этом шаге мы приведем пример, иллюстрирующий использование **основных операций** над кольцевыми списками (**построение кольцевого списка, вывод списка, поиск элемента(по данным), вставка звена после заданного, вставка звена перед заданным, удаление заданного звена, удаление звена после заданного, очистка кольцевого списка**).

На этом шаге мы приведем лишь обобщающий пример. Приведем текст программы, осуществляющий **включение звена в кольцо с удаленным заглавным звеном, а также удаление звена с заданным значением информационного поля из кольца с удаленным заглавным звеном**.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **elem;**

**node \*sled;**

};

class **Spisok {**

private:

**node \*phead,\*Res;**

public:

**Spisok () {phead=**new(node);Res=NULL;}

**~Spisok() {**delete **phead;}**

void **POSTROENIE ();**

void **VYVOD ();**

**node \*POISK (**int);

void **InsAfter (**int);

void **InsBefore (**int);

void **Delete ();**

void **DelAfter ();**

void **OCHISTKA();**

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

int **el,el1;**

**node \*Res\_Zn;**

**A.POSTROENIE();**

**A.VYVOD();**

**cout<<**"\nВведите элемент звена, после которого ";

**cout<<**"осуществляется вставка: ";

**cin>>el;**

**cout<<**"\nВведите элемент вставляемого звена: ";

**cin>>el1;**

if **(A.POISK(el)!=NULL)**

**{ A.InsAfter (el1); A.VYVOD ();}**

else **cout<<**"Звена с заданным элементом в кольце нет!";

**cout<<**"\nВведите элемент звена, перед которым ";

**cout<<**"осуществляется вставка: ";

**cin>>el;**

**cout<<**"Введите элемент вставляемого звена: ";

**cin>>el1;**

if **(A.POISK(el)!=NULL)**

**{**

**A.InsBefore(el1); A.VYVOD ();**

**}**

else **cout<<**"Звена с заданным элементом в кольце нет!";

**cout<<**"\nВведите элемент удаляемого звена: ";

**cin>>el;**

if **(A.POISK(el)!=NULL)**

**{**

**A.Delete (); A.VYVOD ();**

**}**

else **cout<<**"Звена с заданным элементом в кольце нет!";

**cout<<**"\nВведите элемент звена, ";

**cout<<**"после которого нужно удалять: ";

**cin>>el;**

if **(A.POISK(el)!=NULL)**

**{**

**A.DelAfter (); A.VYVOD ();**

**}**

else **cout<<**" Звена с заданным элементом в кольце нет!";

**A.OCHISTKA();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::POSTROENIE ()**

//Построение кольцевого списка с удаленным заглавным звеном.

//phead - указатель на заглавное звено.

{

**node \*t;**

int **el;**

**t = phead; (\*t).sled = NULL;**

**cout<<**"Вводите элементы кольца: ";

**cin>>el;**

while **(el!=0)**

**{**

**(\*t).sled =** new **(node);**

**t = (\*t).sled; (\*t).elem = el;**

**cin>>el;}**

**(\*t).sled = (\*phead).sled;**

}

void **Spisok::VYVOD ()**

//Вывод содержимого кольцевого списка с удаленным заглавным звеном.

//phead - указатель на заглавное звено.

{

**node \*t;**

**t = (\*phead).sled;**

**cout<<**"Кольцо: ";

if **(t!=NULL)**

**{**

**cout<<(\*t).elem<<**" "; t = (\*t).sled;

while **(t!=(\*phead).sled)**

**{ cout<<(\*t).elem <<** " "; t = (\*t).sled; }

**}**

else **cout<<**"пусто!\n";

}

node \*Spisok:: POISK (int **el)**

//Поиск элемента el в кольцевом списке phead.

//Если элемент найден, то Res содержит указатель на звено,

//содержащее элемент el. В противном случае - NULL.

{

**node \*t;**

**Res = NULL; t =(\*phead).sled;**

while **((\*t).sled!=(\*phead).sled && Res==NULL)**

if **((\*t).elem==el) Res = t;**

else **t = (\*t).sled;**

if **(Res==NULL && (\*t).elem==el)**

**Res = t;**

return **Res;**

}

void **Spisok::InsAfter (**int **el)**

//Включение звена с информационным полем el в кольцо

//после звена, на которое указывает ссылка Res.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = el; (\*q).sled = (\*Res).sled;**

**(\*Res).sled = q;**

}

void **Spisok::InsBefore (**int **el)**

//Включение звена с информационным полем el в кольцо

//перед звеном, на которое указывает ссылка Res.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = (\*Res).elem; (\*q).sled = (\*Res).sled;**

**(\*Res).elem = el; (\*Res).sled = q;**

}

void **Spisok::Delete ()**

//Удаление звена, на которое указывает ссылка Res,

//из кольцевого списка с удаленным заглавным звеном,

//заданного указателем phead.

{

**node \*z,\*q;**

if **((\*Res).sled!=(\*phead).sled)**

**{**

**q = (\*Res).sled;**

**(\*Res).elem = (\*((\*Res).sled)).elem;**

**(\*Res).sled = (\*((\*Res).sled)).sled;**

delete **q;**

**}**

elseif **((\*Res).sled==Res)**

**{**

//В кольце единственное звено.

**q = (\*phead).sled; (\*phead).sled = NULL;**

delete **q; cout<<**"Кольцо пусто!";

**}**

else

**{**

//Удаляется "последнее" звено кольца.

**z = phead; q = (\*phead).sled;**

while **(q!=Res)**

**{ z = q; q = (\*q).sled; }**

**(\*z).sled = (\*((\*z).sled)).sled;**

delete **q;**

**}**

}

void **Spisok::DelAfter ()**

//Удаление звена, расположенного после звена,

//на которое указывает ссылка Res,

//из кольцевого списка с удаленным заглавным звеном,

//заданного указателем phead.

{

**node \*q;**

if **((\*Res).sled!=(\*phead).sled)**

**{**

//Ссылка Res не указывает на последнее звено.

**q = (\*Res).sled;**

**(\*Res).sled = (\*((\*Res).sled)).sled;**

delete **q;**

**}**

elseif **((\*Res).sled==Res)**

**{**

//Удаляемое звено - единственное в кольце.

**q = (\*phead).sled; (\*phead).sled = NULL;**

delete **q; cout<<**"Кольцо пусто!";

**}**

else

**{**

//Удаляемое звено - первое в кольце и не единственное.

**q = (\*phead).sled;**

**(\*Res).sled = (\*((\*Res).sled)).sled;**

**(\*phead).sled = (\*Res).sled;** delete **q;**

**}**

}

void **Spisok::OCHISTKA()**

{

**node \*q,\*q1;**// Рабочие указатели.

**q = phead;**

**q1 = (\*q).sled;** // Указатель q1 "опережает" указатель q.

do **{**

**q = q1;**

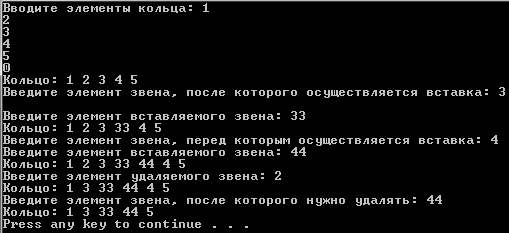
**q1 = (\*q1).sled;**

delete **q;**

**}**

while **(q1!=(\*phead).sled);**

}



Со следующего шага мы начнем рассматривать **списки магазинного типа.**

## Списки магазинного типа

На этом шаге мы дадим **общую характеристику списков магазинного типа**.

В практике информационного моделирования часто используются линейные списки, в которых включение, исключение или доступ к звеньям почти всегда производится в первом или последнем звеньях.

Назовем **списком магазинного типа** линейный список, все звенья которого вставляются и удаляются только с одного или обоих концов списка. Списки магазинного типа подразделяются на **очереди, стеки и деки**.

**Очередь** - список магазинного типа, в котором все включения производятся на одном конце списка, а все исключения делаются на другом его конце.

Очередь иногда называют **циклической памятью** или **списком типа FIFO** ("First In - First Out" - "первым включается - первым исключается").

Мы часто будем использовать термины **начало** и **конец очереди**: информация помещается в конец очереди и удаляется в момент, когда, наконец, достигает ее начала. Изобразим это схематически:

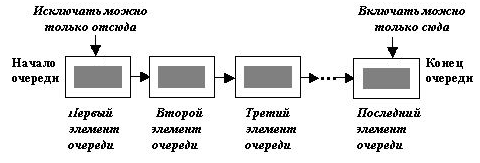


Рис.1. Очередь

**Стек** - список магазинного типа, в котором все включения и исключения звеньев делаются в одном конце списка.

Из стека мы всегда исключаем "младший" элемент из имеющихся в списке (тот элемент, который был включен позже других). Для очереди справедливо в точности противоположное правило: исключается всегда самый "старший" элемент; элементы "покидают" список в том порядке, в котором они в него вошли.

Существуют и другие названия стека: **магазин, список типа LIFO** ("Last In - First Out" - "последним включается - первым исключается"); "**пуш-даун" список ("push-down"), реверсивная память, гнездовая память**.

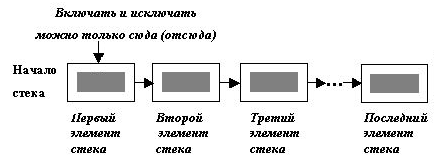


Рис.2. Стек

**Дек** ("Double-Ended Queue" - "двухсторонняя очередь") - список магазинного типа, в котором все включения и исключения звеньев делаются на обоих концах списка.

Очевидно, что дек обладает большей общностью, чем стек или очередь; он имеет некоторые общие свойства с колодой игральных карт.

**Замечание**. Понятие стек было введено А.М.Тьюрингом в 1947 г. и названо им реверсивной памятью (оно использовалось для связи подпрограмм). Термин "дек" был введен Швеппе (E.J.Schweppe).

Со следующего шага мы начнем знакомиться с очередью.

## Списки магазинного типа. Очереди

На этом шаге мы дадим **общую характеристику очереди**.

Отметим, что **очередь - динамическая структура данных, так как с течением времени длина очереди (количество входящих в нее звеньев) изменяется**. Например, все мы знакомы с **очередью** людей у кассы **в магазине** самообслуживания или очередью автомобилей у бензозаправочной станции. Вновь прибывшие становятся в один конец очереди и покидают ее после оплаты покупок или заправки с другого.

Другой, возможно более уместный, пример очереди может быть обнаружен в вычислительной **системе с разделением времени**, с которой одновременно работает несколько пользователей. Поскольку такая система обычно **имеет единственный центральный процессор и одну основную память**, то эти ресурсы должны разделяться среди пользователей путем выделения короткого интервала времени на выполнение программы одного пользователя, за которым следует выполнение программы другого пользователя и так далее до тех пор, пока не будет вновь выполняться программа первого пользователя. **Программы пользователей, ожидающие своего выполнения, образуют очередь ожидания**. **Управление такой очередью необязательно должно основываться на принципе "первым пришел" - "первым вышел", а можно использовать некоторую сложную приоритетную схему, учитывающую такие факторы, как используемый компилятор, требуемое время выполнения, желаемое число строк, выводимых на печать и так далее.**

На следующем шаге мы рассмотрим **формирование очереди**.

### Формирование очереди

На этом шаге мы рассмотрим **алгоритм формирования очереди**.

**Очередь на базе линейного однонаправленного списка** - линейный список, в котором включения элементов производятся на одном конце списка, а все исключения делаются на другом его конце. Обычно **удаление звеньев из очеред**и происходит из начала линейного списка, а **помещение звеньев** осуществляется в конец линейного списка.

Запишем алгоритм формирования очереди:

**r =** new **(node);**

**(\*r).elem = Элем;**

**(\*r).sled = NULL;**



Рис.1. Первый элемент в очереди

**no = r; ko = r;**

// \*no - указатель на начало очереди,

// \*ko - указатель на конец очереди.

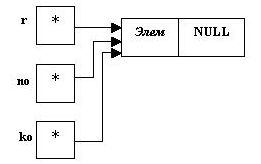


Рис.2. Настройка указателей начала и конца очереди

Итак, мы образовали очередь, состоящую из одного звена.

1. Продолжим заполнение очереди:

**r =** new **(node);**

**(\*r).elem = Элем1;**

**(\*r).sled = NULL;**

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris16_3.jpg

Рис.3. Создали новый элемент очереди

1. Настроим указатель на конец очереди:

**(\*ko).sled = r;**

**ko = r;**

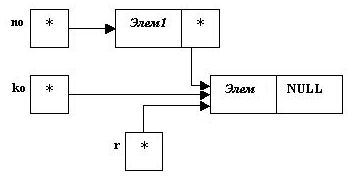


Рис.4. Настроили указатель на конец очереди

Таким образом, **очередь уже содержит два звена** и, нам думается, что процесс построения понятен.

Представим описанный алгоритм в виде функции на языке C++:

void **POSTROENIE (node \*\*no, node \*\*ko)**

// Построение очереди на базе однонаправленного

// линейного списка без заглавного звена:

// \*no - указатель на начало очереди,

// \*ko - указатель на конец очереди.

{

**node \*r;**

int **el;**

**cin>>el;**

if **(el!=0)**

**{**

**r =** new **(node);**

**(\*r).elem = el;**

**(\*r).sled = NULL;**

**\*no = r;**

**\*ko = r;**

**cin>> el;**

while **(el!=0)**

**{ r =** new **(node);**

**(\*r).elem = el; (\*r).sled = NULL;**

**(\*\*ko).sled = r; \*ko = r; cin>>el;}**

**}**

else

**{ r = NULL; \*no = r; \*ko = r;}**

}

Тут же приведем функцию для просмотра содержимого очереди:

void **VYVOD (node \*\*no, node \*\*ko)**

// Вывод содержимого очереди.

// \*no - указатель на начало очереди,

// \*ko - указатель на конец очереди.

{

**node \*r;**

**cout<<** "Очередь: "; r = \*no;

while **(r!=NULL)**

**{ cout<<(\*r).elem<<**" "; r = (\*r).sled; }

**cout<<endl;**

}

**Замечание**. Очереди целесообразно хранить в памяти ЭВМ в виде кольцевого списка с двумя указателями (один - на начало, другой - на конец очереди):

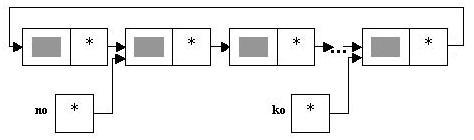


Рис.5. Представление очереди

При исключении из очереди первого звена в первый указатель записывается ссылка на следующее звено списка, которая содержалась в поле указателя исключаемого звена, а при включении в очередь нового звена во второй указатель записывается ссылка на новое звено.

Со следующего шага мы начнем рассматривать **операции с очередями**.

### Добавление звена к очереди

На этом шаге мы рассмотрим алгоритм добавления звена к очереди.

Приступим к описанию **алгоритма добавления звена к очереди**. Напомним, что **звено добавляется в конец очереди**.

1. Вначале построим добавляемое звено:

**r =** new **(node);**

**(\*r).elem = Элем;**

**(\*r).sled = NULL;**



Рис.1. Исходная очередь и заполнение добавляемого звена

1. Присоединяем звено к очереди:

**(\*ko).sled = r;**

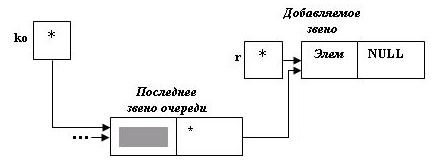


Рис.2. Результат присоединения звена

1. "Настраиваем" указатель ko на конец очереди:

**ko = r;**

Изобразим результат добавления звена:

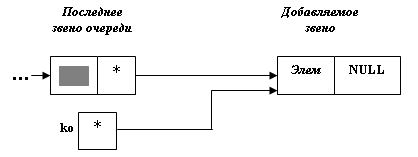


Рис.3. "Настройка" указателя

В результате добавляемое звено стало последним звеном очереди.

Оформим алгоритм в виде функции на языке C++:

void **DOBAVLENIE (node \*\*no, node \*\*ko,**int **el)**

// Добавление звена с информационным полем el

//\* к очереди, определенной указателями \*no и \*ko.

{

**node \*r;**

**r =** new **(node);**

**(\*r).elem = el; (\*r).sled = NULL;**

if **(\*no!=NULL)**

**{ (\*\*ko).sled = r; \*ko = r;}**

else

**{ \*no = r; \*ko = r;}**

}

На следующем шаге мы рассмотрим **алгоритм удаления звена из очереди**.

### Удаление звена из очереди

На этом шаге мы рассмотрим **алгоритм удаления звена из очереди**.

Пусть очередь не пуста (**no != NULL**). Изобразим ее схематически:

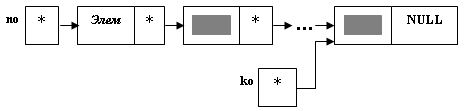


Рис.1. Очередь

Приступим к удалению звена. Напомним, что звено удаляется из очереди из ее начала.

1. Сохраним удаляемый элемент:

**klad = (\*no).elem;**

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris18_2.jpg

Рис.2. Сохранение удаляемого элемента

1. Сохраним указатель на удаляемый элемент и "перенастроим" указатель на начало очереди:

**q = no;**

**no = (\*no).sled;**

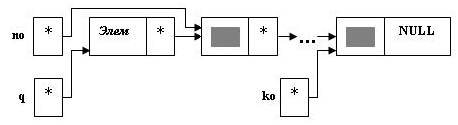


Рис.3. "Перенастройка" указателя на начало очереди

1. Теперь необходимо включить в список свободной памяти удаленное из очереди звено с помощью вызова функции:

delete **q;**

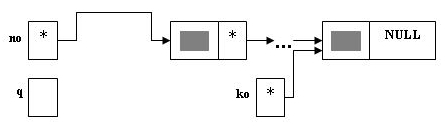


Рис.4. Возврат памяти в кучу

Запишем приведенную схему в виде функции на языке C++:

void **YDALENIE (node \*\*no, node \*\*ko,** int **klad)**

// Удаление звена из очереди, определенной указателями \*no

// и \*ko. Значение информационного поля удаленного звена

// сохраняется в параметре klad.

{

**node \*q;**

if **(\*no == NULL)**

**cout<<** "Удалить нельзя, так как очередь пуста!\n";

else

**{ \*klad = (\*\*no).elem; q = \*no; \*no = (\*\*no).sled;** delete **q;}**

}

**Пример**. Построение и просмотр очереди, добавление и удаление звеньев из очереди.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **elem;**

**node \*sled;**

};

class **Spisok {**

private:

**node \*no,\*ko;**

int **klad;**

public:

**Spisok () {no=ko=NULL;}**

void **POSTROENIE ();**

void **VYVOD ();**

void **DOBAVLENIE (**int);

int **Set\_Udal () {** return **klad; }**

void **YDALENIE ();**

void **OCHISTKA();**

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

int **el;**

**A.POSTROENIE ();**

**A.VYVOD ();**

**cout<<**"Введите добавляемый элемент: ";

**cin>>el;**

**A.DOBAVLENIE (el); A.VYVOD ();**

**cout<<**"Удалим элемент из очереди.\n";

**A.YDALENIE (); A.VYVOD ();**

**el=A.Set\_Udal();**

**cout<<**"Информационное поле удаленного звена: "<<el<<endl;

**A.OCHISTKA();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::POSTROENIE ()**

//Построение очереди на базе однонаправленного списка

//без заглавного звена.

//no - указатель на начало очереди.

//ko - указатель на конец очереди.

{

**node \*r;**

int **el;**

**cout<<**"Вводите элементы очереди:\n";

**cin>>el;**

if **(el!=0)**

**{**

**r =** new **(node);**

**(\*r).elem = el; (\*r).sled = NULL;**

**no = r; ko = r; cin>>el;**

while **(el!=0)**

**{**

**r =** new **(node);**

**(\*r).elem = el; (\*r).sled = NULL;**

**(\*ko).sled = r; ko = r; cin>>el;**

**}**

**}**

else

**{r = NULL; no = r; ko = r;}**

}

void **Spisok::VYVOD ()**

//Вывод содержимого очереди.

//no - указатель на начало очереди.

//ko - указатель на конец очереди.

{

**node \*r;**

**cout<<**"Очередь: "; r = no;

while **(r!=NULL)**

**{**

**cout<<(\*r).elem<<**" "; r = (\*r).sled;

**}**

**cout<<endl;**

}

void **Spisok::DOBAVLENIE (**int **el)**

//Добавление звена с информационным полем el к очереди,

//определенной указателями no и ko.

{

**node \*r;**

**r =** new **(node);**

**(\*r).elem = el; (\*r).sled = NULL;**

if **(no!=NULL)**

**{**

**(\*ko).sled = r; ko = r;**

**}**

else

**{no = r; ko = r;}**

}

void **Spisok::YDALENIE ()**

//Удаление звена из очереди, определенной указателями

//no и ko, с помещением его информационного поля в

//параметр klad.

{

**node \*q;**

if **(no==NULL)**

**cout<<**"Удалить нельзя, так как очередь пуста!\n";

else

**{**

**klad = (\*no).elem; q = no;**

**no = (\*no).sled;** delete **q;**

**}**

}

void **Spisok::OCHISTKA()**

//Возврат выделенной памяти в "кучу".

{

**node \*q;**

**q=no;**

if **(no!=NULL)**

**{**

while **(no!=ko)**

**{**

**no=(\*q).sled;**

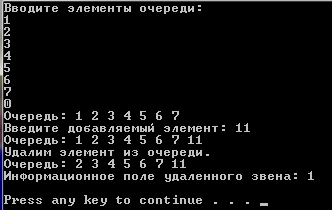
delete **q; q=no; }**

delete **no;**

**no=ko=NULL;**

**}**

}



Со следующего шага мы начнем рассматривать **стек**.

## Стек

### Формирование стека

На этом шаге мы рассмотрим **алгоритм формирования стека**.

Напомним, что **стек** - это специально организованная память, выборка и занесение данных в которую подчиняется дисциплине **LIFO** ("последним вошел - первым обслужен").

**Стек на базе линейного однонаправленного списка** - линейный однонаправленный список, в котором все включения и исключения звеньев делаются в одном (выбранном нами) конце списка.

Опишем алгоритм помещения в стек информации.

1. Вначале стек пуст:

**stk = NULL;**

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris19_1.jpg

Рис.1. Стек пуст

1. Содержимое стека будем вводить с клавиатуры, ввод заканчивается нулем:

**cin>>Элем;**

**t =** new **(node);**

**(\*t).elem = Элем; (\*t).sled = stk;**

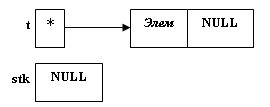


Рис.2. Новый элемент

1. "Настраиваем" указатель стека на созданный элемент:

**stk = t;**

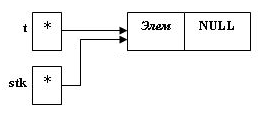


Рис.3. "Настройка" указателя стека

1. В результате в стек будет помещено первое звено:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris19_4.jpg

Рис.4. Первый элемент в стеке

1. Продолжим заполнение стека:

**cin>>Элем1;**

**t =** new **(node);**

**(\*t).elem = Элем1;**

**(\*t).sled = stk;**

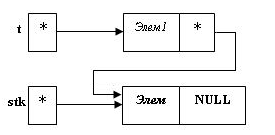


Рис.5. Размещение в стеке второго элемента

1. "Настраиваем" указатель стека на созданный элемент:

**stk = t;**

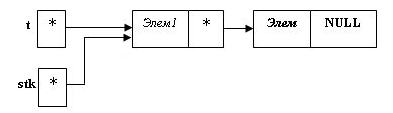


Рис.6. "Настройка" указателя стека

1. Теперь стек содержит уже два звена:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris19_7.jpg

Рис.7. В стеке два элемента

Продолжение процесса построения стека достаточно очевидно.

Оформим алгоритм в виде функции языка C++:

void **POSTROENIE (node \*\*stk)**

//Построение стека, заданного указателем \*stk с клавиатуры.

{

**node \*t;**

int **el;**

**\*stk = NULL;**

**cin>>el;**

while **(el!=0)**

**{**

**t =** new **(node);**

**(\*t).elem = el; (\*t).sled = \*stk; \*stk = t;**

**cin>>el;**

**}**

}

На следующего шаге мы рассмотрим **алгоритм включения звена в стек**.

### Включение звена в стек

На этом шаге мы рассмотрим алгоритм включения элемента в стек.

Опишем алгоритм включения звена с информационным полем Элем в стек.

1. Исходное состояние стека:

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris20_1.jpg

Рис.1. Исходное состояние стека

1. Создаем новый элемент:

**q =** new **(node);**

**(\*q).elem = Элем;**

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris20_2.jpg

Рис.2. Новый элемент

1. Включаем элемент в начало стека:

**(\*q).sled = stk;**

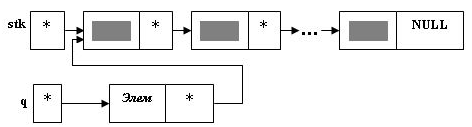


Рис.3. Включение элемента в стек

1. "Настроим" указатель вершины стека:

**stk = q;**

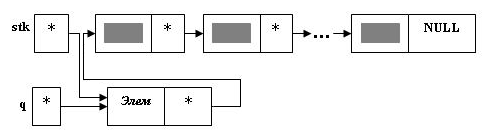


Рис.4. "Настройка" указателя вершины стека

Функция имеет вид:

void **W\_S (node \*\*stk,** int **el)**

//Включение звена с элементом el в стек,

// заданный указателем \*stk.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = el; (\*q).sled = \*stk; \*stk = q;**

}

Заметим, что функцию **[POSTROENIE()](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0019.html" \l "1)** можно переписать следующим образом, если воспользоваться функцией W\_S():

void **POSTROENIE (node \*\*stk)**

// Построение стека, заданного указателем \*stk с клавиатуры.

{

int **el;**

**\*stk = NULL; cin>>el;**

while **(el!=0)**

**{ W\_S (stk,el); cin>>el;}**

}

На следующем шаге мы рассмотрим **алгоритм удаления элемента из стека**.

### Удаление звена из стека

На этом шаге мы рассмотрим **алгоритм удаления звена из стека**.

Перед удалением звена из стека проверяем, пуст ли стек.

Пусть стек не пуст.

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris21_1.jpg

Рис.1. Исходный стек

Тогда спокойно приступаем к удалению.

1. Сохраняем удаляемый элемент:

**klad = (\*stk).elem;**

http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/ris21_2.jpg

Рис.2. Сохранение удаляемого элемента

1. "Перенастраиваем" указатель стека и сохраняем адрес удаляемого элемента:

**q = stk; stk = (\*stk).sled;**

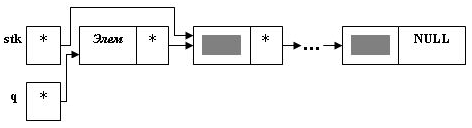


Рис.3. "Перенастройка" указателя стека

1. Возвращаем память в кучу:

delete **q;**

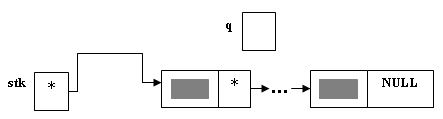


Рис.4. Возврат памяти в кучу

Запишем полученный алгоритм в виде функции:

void **YDALENIE (node \*\*stk,** int **klad)**

// Удаление звена из стека, заданного указателем \*stk, и

// помещение значения информационного поля удаленного звена

// в параметр klad.

{

**node \*q;**

if **(\*stk==NULL) cout<<**"Стек пуст!\n";

else

**{ \*klad = (\*\*stk).elem; q = \*stk;**

**\*stk = (\*\*stk).sled;** delete **q;}**

}

Приведем **пример** программы, реализующей действия со стеком. Формирование и вывод содержимого стека на экран дисплея. Удаление и вставка звена в стек.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **elem;**

**node \*sled;**

};

class **Spisok**

{

private:

**node \*stk;**

int **klad;**

public:

**Spisok () { stk=NULL; }**

int **Set\_Stack () {**return **klad;}**

void **POSTROENIE();**

void **VYVOD();**

void **W\_S(**int);

void **YDALENIE ();**

void **OCHISTKA();**

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

int **el;**

int **t;** //Содержимое информационного поля верхушки стека.

**A.POSTROENIE (); A.VYVOD ();**

**cout<<**"Введите вставляемый элемент: ";

**cin>>el;**

**A.W\_S (el); A.VYVOD ();**

**cout<<**"Удалим элемент из стека.\n";

**A.YDALENIE ();**

**t=A.Set\_Stack();**

**cout<<**"Из стека было извлечено число... "<<t<<endl;

**A.VYVOD ();**

**A.OCHISTKA();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::POSTROENIE ()**

//Построение стека, заданного указателем stk.

{

**node \*t;**

int **el;**

**cout<<**"Вводите элементы стека: ";

**cin>>el;**

while **(el!=0)**

**{**

**t =** new **(node);**

**(\*t).elem = el; (\*t).sled = stk;**

**stk = t; cin>>el;**

**}**

}

void **Spisok::VYVOD ()**

//Вывод содержимого стека, заданного указателем stk.

{

**node \*t;**

**cout<<**"Содержимое стека: "; t = stk;

while **(t!=NULL)**

**{cout<<(\*t).elem<<**" "; t = (\*t).sled;}

**cout<<endl;**

}

void **Spisok::W\_S (**int **el)**

//Помещение элемента el в стек stk.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = el; (\*q).sled = stk; stk = q;**

}

void **Spisok::YDALENIE ()**

//Удаление элемента из стека, заданного указателем stk.

//Значение информационного поля удаляемого элемента

//помещается в параметр klad.

{

**node \*q;**

if **(stk==NULL)**

**cout<<**"Стек пуст!\n";

else

**{**

**klad = (\*stk).elem; q = stk;**

**stk = (\*stk).sled;** delete **q;**

**}**

}

void **Spisok::OCHISTKA()**

//Возврат выделенной памяти в "кучу".

{

**node \*t,\*q;**

**t = stk;**

if **(t!=NULL)**

**{**

**q=(\*t).sled;**

while **(q!=NULL)**

**{**

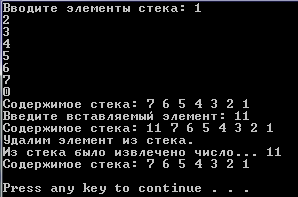
delete **t; t = q; q =(\*q).sled;**

**}**

delete **t;**

**}**

}



Со следующего шага мы познакомимся с **деком**.

## Дек

На этом шаге мы рассмотрим **создание и основные операции над деками**.

**Дек ("двухсторонняя очередь")** на базе однонаправленного линейного списка - список магазинного типа на базе однонаправленного линейного списка, в котором все **включения и исключения звеньев делаются на обеих концах очереди**.

Мы приведем лишь демонстрационный пример.

**Пример**. Формирование дека, просмотр его содержимого, добавление элемента к деку и удаление элемента из дека.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **elem;**

**node \*sled;**

};

class **Spisok**

{

private:

**node \*ld,\*rd;**

int **el\_left,el\_right;**

public:

void **POSTROENIE ();**

void **VYVOD ();**

void **VSTAV1 (**int);

void **VSTAV2 (**int);

int **SetElLeft() {**return **el\_left;}**

int **SetElRight() {**return **el\_right;}**

void **YDALE1 ();**

void **YDALE2 ();**

void **OCHISTKA();**

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

int **el;**

**A.POSTROENIE (); A.VYVOD ();**

**cout<<**"Добавим звено справа.\n";

**cout<<**"Введите элемент добавляемого звена: ";

**cin>>el;**

**A.VSTAV1 (el); A.VYVOD ();**

**cout<<**"Добавим звено слева.\n";

**cout<<**"Введите элемент добавляемого звена: ";

**cin>>el;**

**A.VSTAV2 (el); A.VYVOD ();**

**cout<<**"Удалим звено справа.\n";

**A.YDALE1 (); A.VYVOD (); cout<<A.SetElRight()<<endl;**

**cout<<**"Удалим зввено слева.\n";

**A.YDALE2 (); A.VYVOD (); cout<<A.SetElLeft()<<endl;**

**A.OCHISTKA();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::POSTROENIE ()**

//Построение дека :

// ld - указатель на левый конец дека,

// rd - Указатель на правый конец дека.

{

**node \*k;**

int **el;**

**cout<<**"Вводите содержимое звеньев дека: \n";

**cin>>el;**

if **(el!=0)**

**{**

**k =** new **(node);**

**(\*k).elem = el; (\*k).sled = NULL;**

**ld = k; rd = k; cin>>el;**

while **(el!=0)**

**{VSTAV1 (el); cin>>el;}**

**}**

else

**{rd = NULL; ld = NULL;}**

}

void **Spisok::VYVOD ()**

//Вывод содержимого дека:

// ld - указатель на левый конец дека.

{

**node \*k;**

**k = ld; cout<<**"Дек: ";

while **(k!=NULL)**

**{cout<<(\*k).elem<<**" "; k = (\*k).sled;}

**cout<<endl;**

}

void **Spisok::VSTAV1 (**int **el)**

// Помещение звена, содержащего элемент el, в дек справа.

// ld - указатель на левый конец дека,

// rd - указатель на правый конец дека.

{

**node \*k;**

**k =** new **(node);**

**(\*k).elem = el; (\*k).sled = NULL;**

if **(rd!=NULL)**

**{(\*rd).sled = k; rd = k;}**

else

**{rd = k; ld = k;}**

}

void **Spisok::VSTAV2 (**int **el)**

//Помещение звена, содержащего элемент el, в дек слева.

// ld - указатель на левый конец дека,

// rd - указатель на правый конец дека.

{

**node \*k;**

**k =** new **(node);**

**(\*k).elem = el; (\*k).sled = ld;**

if **(ld!=NULL) ld = k;**

else **{ld = k; rd = k;}**

}

void **Spisok::YDALE1 ()**

//Удаление звена из дека справа

//с сохранением удаляемого звена в переменной el\_right.

// ld - указатель на левый конец дека,

// rd - указатель на правый конец дека.

{

**node \*z;**

**node \*k;**

if **(rd==ld)**

**{**

**el\_right = (\*rd).elem;** delete **rd;**

**ld = rd = NULL; cout<<**"Дек пуст!\n";

**}**

else

**{**

**z = ld; k = (\*ld).sled;**

while **(k!=rd)**

**{z = k; k = (\*k).sled;}**

**el\_right = (\*rd).elem; (\*z).sled = NULL;** delete **rd;**

**rd = z;**

**}**

}

void **Spisok::YDALE2 ()**

// Удаление звена из дека слева

// с сохранением удаляемого звена в переменной el\_left.

// ld - указатель на левый конец дека,

// rd - указатель на правый конец дека.

{

**node \*q;**

if **(ld!=NULL)**

**{**

**el\_left = (\*ld).elem; q = ld;**

**ld = (\*ld).sled;** delete **q;**

**}**

else **cout<<**"Дек пуст!\n";

}

void **Spisok::OCHISTKA()**

{

**node \*k,\*q;**

**k = ld;**

if **(k!=NULL)**

**{**

**q = (\*k).sled;**

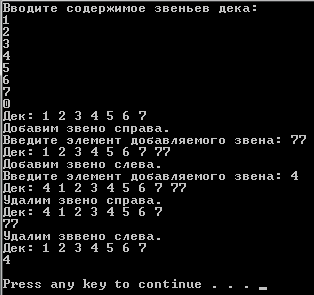
while **(q!=NULL)**

**{**delete **k; k = q; q=(\*k).sled;}**

delete **k;**

**}**

}



Со следующего шага мы начнем рассматривать **линейные двунаправленные списки**.

## Линейные двунаправленные списки

На этом шаге мы приведем **общую информацию о линейных двунаправленных списках**.

Мы будем рассматривать лишь **двунаправленные списки с заглавным звеном**, которые имеют следующую структуру:

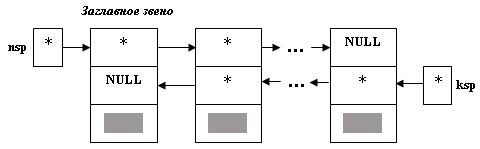


Рис.1. Общий вид двунаправленного списка с заглавным звеном

Здесь **nsp** - указатель на заглавное звено двунаправленного списка, **ksp** - указатель на последнее звено двунаправленного списка.

Тип каждого звена списка можно описать так:

struct **node**

{

int **elem;**//Информационное поле.

**node \*sled;** // Указатель на следующее звено.

**node \*pred;** // Указатель на предыдущее звено.

};



Рис.2. Структура звена списка

На следующем шаге мы разберем **формирование списка**.

### Формирование линейного двунаправленного списка

На этом шаге мы рассмотрим **алгоритм формирования списка**.

Приступим к построению алгоритма формирования двунаправленного списка с заглавным звеном. Опишем необходимые переменные:

**node \*nsp;** // Указатель на заглавное звено списка.

**node \*ksp;** // Указатель на последнее звено списка.

**node \*rsp;** // Рабочий указатель для перемещения по списку.

1. Построим заглавное звено:

**nsp =** new(node);

**rsp = nsp;**

**(\*nsp).pred = NULL;**

**(\*nsp).sled = NULL;**

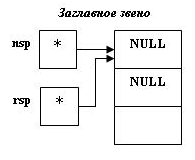


Рис.1. Заглавное звено

1. Создаем элемент списка:

**cin>>el;**

**(\*rsp).sled =** new(node);

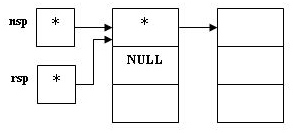


Рис.2. Элемент списка

1. Заполняем поля элемента:

**(\*((\*rsp).sled)).pred = rsp;**

**rsp = (\*rsp).sled;**

**(\*rsp).sled = NULL;**

**(\*rsp).elem = el;**

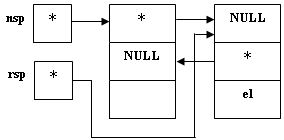


Рис.3. Заполнили поля элемента списка

1. "Настраиваем" указатель на последний элемент списка:

**ksp = rsp;**

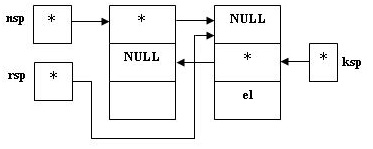


Рис.4. "Настроили" указатель

Построен двунаправленный список с заглавным звеном, содержащий одно звено. Мы предоставляем возможность читателю проделать самостоятельно несколько последующих шагов алгоритма.

Запишем функцию построения двунаправленного списка с заглавным звеном:

void **Postroenie (node \*\*nsp, node \*\*ksp)**

// Построение двунаправленного списка с заглавным звеном:

// \*nsp - указатель на начало списка,

// \*ksp - указатель на конец списка.

{

**node \*rsp;**

int **el;**

**\*nsp =** new(node);

**rsp = \*nsp;**

**(\*\*nsp).pred = (\*\*nsp).sled = NULL;**

**cout<<**"Вводите последовательность:\n"; cin>>el;

while **(el!=0)**

**{ (\*rsp).sled =** new(node); (\*((\*rsp).sled)).pred = rsp;

**rsp = (\*rsp).sled; (\*rsp).sled = NULL; (\*rsp).elem = el;**

**cin>>el; }**

**\*ksp = rsp;**

}

Со следующего шага мы начнем рассматривать **алгоритмы обхода двунаправленного списка**.

### Проход по линейному двунаправленному списку, начиная с его начала

На этом шаге мы рассмотрим **один из способов перемещения по списку**.

Алгоритм перемещения может быть следующим.

* Установим указатель **rsp** на звено, следующее за заглавным.
* Далее, в цикле указатель будем перемещать по списку "с помощью" оператора **rsp = (\*rsp).sled;**.
* "Движение" указателя продолжается, пока **rsp<>NULL**.

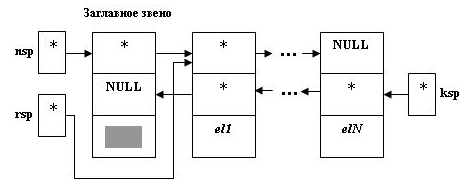


Рис.1. "Начальная позиция"

Алгоритм оформим в виде функции:

void **VyvodForward (node \*\*nsp, node \*\*ksp)**

// Вывод содержимого двунаправленного списка от его начала.

// \*nsp - указатель на начало списка,

// \*ksp - указатель на конец списка.

{

**node \*rsp;**

**rsp = (\*\*nsp).sled;**

**cout<<**"Двунаправленный список содержит: ";

while **(rsp!=NULL)**

**{ cout<<(\*rsp).elem; rsp = (\*rsp).sled;}**

**cout<<endl;**

}

На следующем шаге мы рассмотрим **другой способ перемещения по списку.**

### Проход по линейному двунаправленному списку, начиная с его конца

На этом шаге мы рассмотрим проход по списку, начиная с его конца.

Алгоритм перемещения может быть следующим.

* Установим указатель **rsp** на последнее звено двунаправленного списка.
* Далее, в цикле указатель перемещается по списку "с помощью" оператора **rsp = (\*rsp).pred;**.
* Ясно, что "движение" указателя продолжается, пока **(\*rsp).pred<>NULL**.

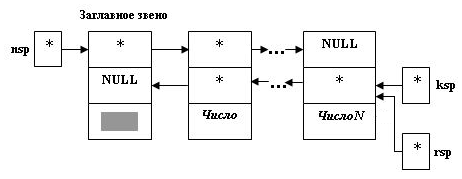


Рис.1. "Начальная позиция"

Алгоритм оформим в виде функции:

void **VyvodBack (node \*\*nsp, node \*\*ksp)**

// Вывод содержимого двунаправленного списка от его конца.

// \*nsp - указатель на начало списка,

// \*ksp - указатель на конец списка.

{

**node \*rsp;**

**rsp = \*ksp;**

**cout<<**"Двунаправленный список в обратном порядке: ";

while **((\*rsp).pred!=NULL)**

**{ cout<< (\*rsp).elem; rsp = (\*rsp).pred;}**

cout<<endl;

}

Со следующего шага мы начнем рассматривать **основные операции над списками**.

### Поиск звена в двунаправленном списке, начиная с начала списка

На этом шаге мы рассмотрим первый способ поиска звена.

Поиск звена в двунаправленном списке можно производить в двух направлениях: с начала и с конца списка. Рассмотрим алгоритм поиска с начала списка.

Обозначим через **Элем** информационное поле звена, которое необходимо найти в списке. Вначале опишем алгоритм:

**Res = NULL;** //Указатель на искомое звено содержит NULL.

**q = (\*nsp).sled;** //Рабочий указатель установим на звено,

//следующее за заглавным.

while **(q!=NULL && Res=NULL)**

**{**

if **((\*q).elem==Элем)**

// Если звено, содержащее элемент Элем, найдено, то

//запоминаем указатель на найденное звено.

**Res = q;**

else

// ... иначе: перемещаем рабочий указатель по списку.

**q = (\*q).sled;**

**}**

Оформим приведенный выше алгоритм в виде функции:

node \*PoiskForward (int **el,node\*\*nsp,node\*\*ksp)**

// Функция возвращает указатель на найденное звено,

// содержащее элемент el двунаправленного списка,

// заданного указателями \*nsp и \*ksp, или NULL, если

// звено в списке не найдено.

{

**node \*q;** // Текущий указатель.

**node \*Res;**

**\*Res = NULL; q = (\*\*nsp).sled;**

while **(q!=NULL && \*Res==NULL)**

**{** if **((\*q).elem==el) \*Res = q;** else **q = (\*q).sled;}**

return **Res;**

}

На следующем шаге мы рассмотрим **поиск звена, начиная с конца списка**.

### Поиск звена в двунаправленном списке, начиная с конца списка

На этом шаге мы рассмотрим **другой способ поиска звена в списке**.

Рассмотрим алгоритм поиска элемента, начиная с конца списка.

* Рабочий указатель устанавливается на последнее звено двунаправленного списка.
* Анализ содержимого информационных полей происходит таким же образом, как и в функции [**PoiskForward()**](http://khpi-iip.mipk.kharkiv.edu/library/datastr/book_sod/kgsu/din_0027.html), только рабочий указатель q теперь перемещается в **"обратном" направлении** с помощью оператора **q = (\*q).pred;**.

Оформим алгоритм в виде функции:

node \*PoiskBack (int **el,node \*\*nsp,node \*\*ksp)**

// Функция возвращает указатель на найденное звено,

// содержащее элемент el двунаправленного списка,

// заданного указателями \*nsp и \*ksp, или NULL, если

// звено в списке не найдено.

{

**node \*q;**

**node \*Res;**

**Res = NULL; q = \*ksp;**

while **(q!=NULL && Res==NULL)**

**{** if **((\*q).elem==el) Res = q;** else **q = (\*q).pred;}**

return **Res;**

}

Со следующего шага мы начнем знакомиться **со способами вставки звена в линейный двунаправленный список**.

### Вставка звена в двунаправленный список (1-й случай)

На этом шаге мы рассмотрим **алгоритм вставки нового звена после звена, на которое указывает ссылка.**

Вначале рассмотрим **алгоритм вставки звена после звена, на которое указывает ссылка Res**. Пусть ссылка **Res** указывает на звено, после которого будет производиться вставка нового звена. Изобразим это схематически:

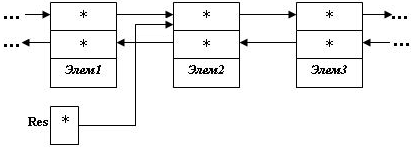


Рис.1. "Начальная позиция"

Проверим, является ли звено, на которое указывает ссылка **Res**, последним в списке. Это осуществляется путем анализа значения операции отношения **(\*Res).sled != NULL**.

Пусть звено, на которое указывает **Res, не является последним**.

1. В **heap**-области резервируем место для нового динамического объекта, а в информационное поле этого объекта помещаем значение информационного поля звена, которое желательно вставить в двунаправленный список:

**q =** new(node);

**(\*q).elem = Элем;**

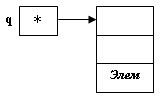


Рис.2. Вставляемый элемент

1. "Настраиваем" указатели вставляемого элемента:

**(\*q).sled = (\*Res).sled;**

**(\*q).pred = (\*\*Res.sled).pred;**

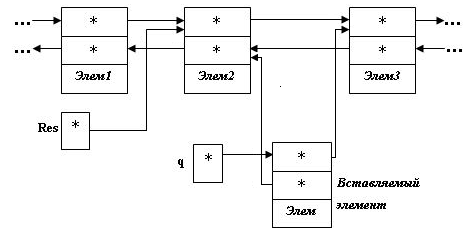


Рис.3. "Настройка" указателей вставляемого элемента

1. "Настраиваем" указатели списка на вставляемый элемент:

**(\*\*Res.sled).pred = q;**

**(\*Res).sled = q;**

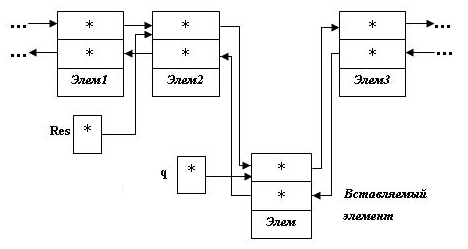


Рис.4. "Настройка" указателей элементов списка на вставляемый элемент

1. В итоге получим:

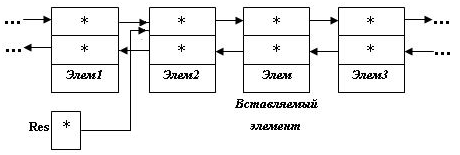


Рис.5. Результат вставки элемента

Если элемент, после которого размещается новый элемент, окажется **последним в двунаправленном списке**, тогда алгоритм изменится следующим образом:

1. Создадим новый элемент и заполним его поля:

**q =** new(node);

**(\*q).elem = Элем;**

**(\*q).sled = NULL;**

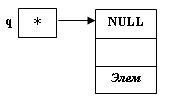


Рис.6. Вставляемый элемент

1. Осуществим "настройку" указателей:

**(\*q).pred = Res; \*ksp = q;**

**(\*Res).sled = q;**

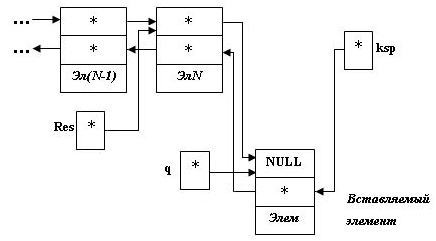


Рис.7. "Настройка" указателей

Оформим алгоритм вставки звена в список в виде функции:

void **InsAfter (**int **el, node \*\*nsp, node \*\*ksp, node \*Res)**

// Вставление звена с информационным полем el в

// двунаправленный список, заданный указателями

// \*nsp и \*ksp, после звена, на которое указывает Res.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

if **((\*Res).sled!=NULL)**

**{**

**(\*q).sled = (\*Res).sled;**

**(\*q).pred = (\*(\*Res).sled).pred; (\*(\*Res).sled).pred = q; (\*Res).sled = q;**

**}**

else

**{ (\*q).sled = NULL; (\*q).pred = Res; \*ksp = q; (\*Res).sled = q;}**

}

На следующем шаге мы продолжим рассматривать **алгоритмы вставки звена в список**.

### Вставка звена в двунаправленный список (2-й случай)

На этом шаге мы рассмотрим **другой способ вставки звена в двунаправленный список**.

Рассмотрим **алгоритм вставки звена перед звеном, на которое указывает ссылка Res**. Опишем его с помощью схем "до и после" Д.Кнута:

1. Создаем элемент:

**q =** new(node);

**(\*q).elem = Элем;**

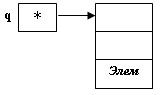


Рис.1. Создание элемента

1. "Настраиваем" его указатели:

**(\*q).sled = (\*(\*Res).pred).sled;**

**(\*q).pred = (\*Res).pred;**

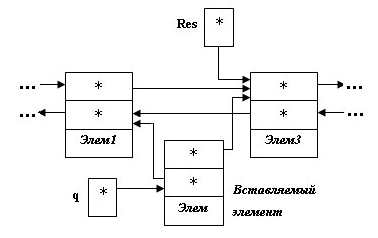


Рис.2. "Настройка" указателей созданного элемента

1. "Настраиваем" указатели элементов списка на вставляемый элемент:

**(\*(\*Res).pred).sled = q;**

**(\*Res).pred = q;**

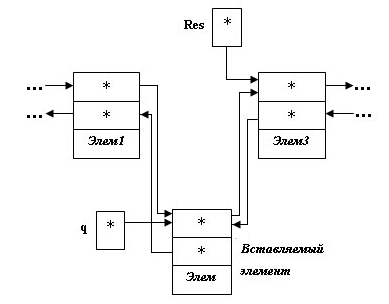


Рис.3. "Настройка" указателей элементов списка на созданный элемент

1. В итоге получим:

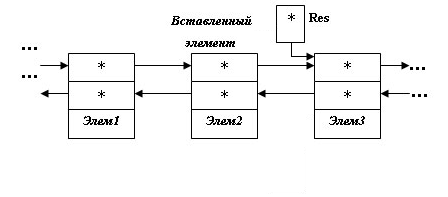


Рис.4. Итоговый результат

Оформим алгоритм помещения звена в список в виде функции:

void **InsBefore (**int **el, node \*\*nsp, node \*\*ksp, node \*Res)**

// Вставление звена с информационным полем el в двуна-

// правленный список, заданный указателями \*nsp и \*ksp,

// перед звеном, на которое указывает Res.

{

**node \*q;**

**q =** new **(node);**

**(\*q).elem = el;**

**(\*q).sled = (\*(\*Res).pred).sled;**

**(\*q).pred = (\*Res).pred;**

**(\*(\*Res).pred).sled = q; (\*Res).pred = q;**

}

Со следующего шага мы начнем знакомиться с **алгоритмами удаления звена из двунаправленного списка**.

### Удаление звена из двунаправленного списка. Указатель на удаляемое звено (1-й случай)

На этом шаге мы рассмотрим **первый случай удаления звена**.

Пусть нам **известен указатель Res на удаляемое звено**. Возможны два случая:

* удаляемый элемент является последним в двунаправленном списке,
* удаляемый элемент не является последним в двунаправленном списке.

Вначале рассмотрим ситуацию, когда он **не последний**, то есть **(\*Res).sled != NULL**. Изобразим схему "до", соответствующую этому случаю:

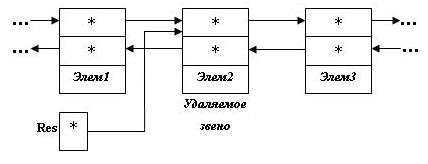


Рис.1. "Начальная" позиция

1. Исключим звено из списка, "перенастроив" указатели предыдущего и последующего звеньев:

**(\*(\*Res).sled).pred = (\*Res).pred;**

**(\*(\*Res).pred).sled = (\*Res).sled;**

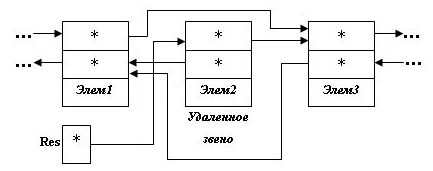


Рис.2. "Перенастройка" указателей

1. Осталось только удалить звено, на которое указывает Res, из heap-области:

delete **Res;**

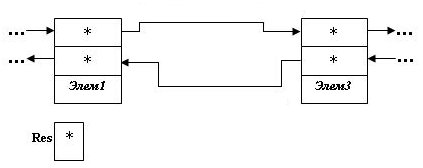


Рис.3. Освобождение памяти

Пусть теперь **удаляемый элемент оказался последним**.

Взгляните на схему "до":

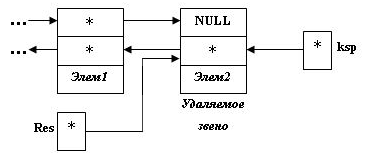


Рис.4. "Начальная" позиция

1. "Настроим" указатели предпоследнего элемента и конца списка:

**(\*(\*Res).pred).sled = NULL;**

**ksp = (\*ksp).pred;**

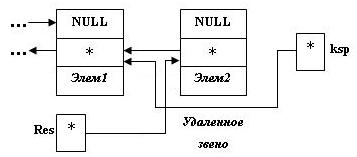


Рис.5. "Настройка" указателей

1. Теперь удалим из кучи звено, на которое указывает Res:

delete **Res;**

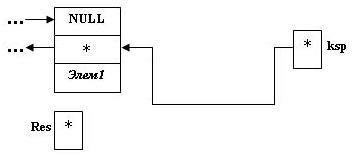


Рис.6. Возврат памяти в кучу

Запишем функцию для удаления звена, на которое указывает ссылка:

void **Delete (node \*\*nsp, node \*\*ksp, node \*Res)**

// Удаление звена из двунаправленного списка.

// \*nsp - указатель на начало списка,

// \*ksp - указатель на конец списка,

// Res - указатель на удаляемое звено.

{

if **((\*Res).sled!=NULL)**

**{ (\*(\*Res).sled).pred = (\*Res).pred;**

**(\*(\*Res).pred).sled = (\*Res).sled;** delete **Res;}**

else

**{ (\*(\*Res).pred).sled = NULL; \*ksp = (\*\*ksp).pred;** delete **Res;}**

}

На следующем шаге мы продолжим знакомиться с алгоритмами **удаления звена**.

### Удаление звена из двунаправленного списка. После звена (2-й случай)

На этом шаге мы рассмотрим другой **алгоритм удаления звена из двунаправленного списка**.

Построим теперь **алгоритм для удаления звена, стоящего после звена, на которое указывает ссылка Res**. Пусть удаляемое звено не является последним, то есть **(\*(\*Res).sled).sled!=NULL**. Изобразим этот факт на схеме "до":

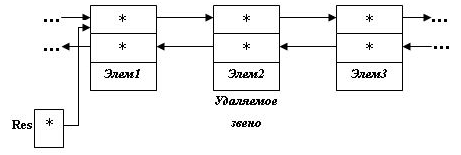


Рис.1. "Начальная" позиция

1. Исключим звено из списка, "перенастроив" указатели предыдущего и последующего звеньев, а также сохраним адрес удаляемого звена:

**q = (\*Res).sled;**

**(\*(\*(\*Res).sled).sled).pred = Res;**

**(\*Res).sled = (\*(\*Res).sled).sled;**

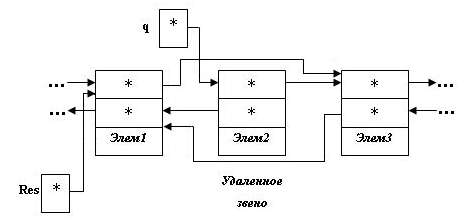


Рис.2. "Перенастройка" указателей

1. Осталось только удалить звено, на которое указывает **q**, из **heap**-области:

delete **q;**

Если же удаляемый элемент является **последним** в двунаправленном списке,

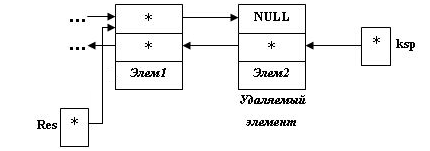


Рис.3. "Начальная" позиция

то алгоритм становится тривиальным:

1. "Настроим" указатели предпоследнего элемента и конца списка:

**q = (\*Res).sled; (\*Res).sled = NULL; ksp = (\*ksp).pred;**

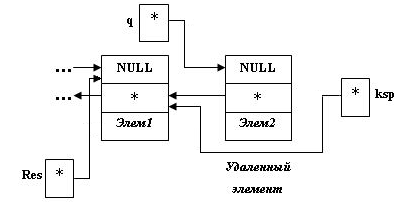


Рис.4. "Настройка" указателей

1. Теперь удалим из кучи звено, на которое указывает **q:**

delete **q;**

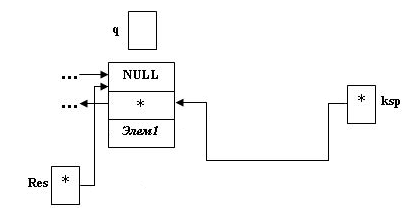


Рис.5. Возврат памяти в кучу

Функция для удаления звена, стоящего после звена, на которое указывает ссылка **Res**, имеет вид:

void **DelAfter (node \*\*nsp, node \*\*ksp, node \*Res)**

// Удаление звена из двунаправленного списка.

// \*nsp - указатель на начало списка,

// \*ksp - указатель на конец списка,

// Res - указатель на звено, предыдущее удаляемому.

{

**node \*q;**

if **((\*Res).sled==NULL)**

**cout<<**"Вы хотите удалить звено за последним звеном!\n";

else

if **((\*(\*Res).sled).sled!=NULL)**

**{**

**q = (\*Res).sled;**

**(\*(\*(\*Res).sled).sled).pred = Res;**

**(\*Res).sled = (\*(\*Res).sled).sled;**

delete **q;**

**}**

else

**{ q = (\*Res).sled; (\*Res).sled = NULL;**

**\*ksp = (\*\*ksp).pred;** delete **q; }**

}

Приведем пример программы, иллюстрирующей применение рассмотренных функций.

Пример: построение двунаправленного списка. Помещение звена в двунаправленный список. Удаление звена из двунаправленного списка. Очистка двунаправленного списка.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **elem;**

**node \*sled;**

**node \*pred;**

};

class **Spisok**

{

private:

**node \*nsp,\*ksp;**

public:

**Spisok() {nsp=ksp=NULL;}**

void **Postroenie ();**

void **VyvodForward ();**

void **VyvodBack ();**

void **Ochistka ();**

void **InsAfter (**int,node\*);

void **InsBefore (**int,node\*);

void **Delete (node\*);**

void **DelAfter (node\*);**

**node \*PoiskForward (**int);

**node \*PoiskBack (**int);

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

**node \*Res;**

int **el,el1;**

**A.Postroenie ();**

**A.VyvodForward (); A.VyvodBack ();**

**cout<<**"Введите элемент звена, после которого ";

**cout<<**"осуществляется вставка: ";

**cin>>el;**

**cout<<**"Введите элемент вставляемого звена: ";

**cin>>el1;**

**Res=A.PoiskForward (el);**

if **(Res!=NULL)**

**{**

**A.InsAfter (el1,Res);**

**A.VyvodForward (); A.VyvodBack ();**

**}**

else **cout<<**"Звена с заданным элементом в списке нет!\n";

**cout<<**"Введите элемент звена, перед которым ";

**cout<<**"осуществляется вставка: ";

**cin>>el;**

**cout<<**"Введите элемент вставляемого звена: ";

**cin>>el1;**

**Res = A.PoiskBack (el);**

if **(Res!=NULL)**

**{**

**A.InsBefore (el1,Res);**

**A.VyvodForward (); A.VyvodBack ();**

**}**

else **cout<<**"Звена с заданным элементом в списке нет!\n";

**cout<<**"Введите элемент звена, после которого ";

**cout<<**"осуществляется удаление: ";

**cin>>el;**

**Res = A.PoiskForward (el);**

if **(Res!=NULL)**

**{**

**A.DelAfter (Res);**

**A.VyvodForward (); A.VyvodBack (); }**

else **cout<<**"Звена с заданным элементом в списке нет!\n";

**cout<<**"Введите элемент звена, которое ";

**cout<<**"надо удалить: ";

**cin>>el;**

**Res = A.PoiskForward (el);**

if **(Res!=NULL)**

**{**

**A.Delete (Res);**

**A.VyvodForward (); A.VyvodBack ();**

**}**

else **cout<<**"Звена с заданным элементом в списке нет!\n";

**A.Ochistka ();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::Postroenie ()**

//Построение двунаправленного списка с заглавным звеном:

// nsp - указатель на начало списка,

// ksp - указатель на конец списка.

{

**node \*rsp;**

int **el;**

**nsp =** new(node);

**rsp = nsp;**

**(\*nsp).pred = NULL; (\*nsp).sled = NULL;**

**cout<<**"Вводите последовательность:\n";

**cin>>el;**

while **(el!=0)**

**{**

**(\*rsp).sled =** new(node);

**(\*((\*rsp).sled)).pred = rsp; rsp = (\*rsp).sled;**

**(\*rsp).sled = NULL; (\*rsp).elem = el;**

**cin>>el;**

**}**

**ksp = rsp;**

}

void **Spisok::VyvodForward ()**

//Вывод содержимого двунаправленного списка от его начала.

// nsp - указатель на начало списка, ksp - указатель на конец списка.

{

**node \*rsp;**

**rsp = (\*nsp).sled;**

**cout<<**"Двунаправленный список содержит: ";

while **(rsp!=NULL)**

**{**

**cout<<(\*rsp).elem<<**" "; rsp = (\*rsp).sled;

**}**

**cout<<endl;**

}

void **Spisok::VyvodBack ()**

//Вывод содержимого двунаправленного списка от его конца.

// nsp - указатель на начало списка, ksp - указатель на конец списка.

{

**node \*rsp;**

**rsp = ksp;**

**cout<<**"Двунаправленный список в обратном порядке: ";

while **((\*rsp).pred!=NULL)**

**{**

**cout<<(\*rsp).elem<<**" "; rsp = (\*rsp).pred;

**}**

**cout<<endl;**

}

node \*Spisok::PoiskForward (int **el)**

//Функция возвращает указатель на найденный элемент el

//двунаправленного списка, заданного указателями nsp

// и ksp, или NULL, если элемент в списке не найден.

{

**node \*q;**

**node \*Res;**

**Res = NULL; q = (\*nsp).sled;**

while **(q!=NULL && Res==NULL)**

if **((\*q).elem==el) Res = q;**

else **q = (\*q).sled;**

return **Res;**

}

node \*Spisok::PoiskBack (int **el)**

//Функция возвращает указатель на найденный элемент el

//двунаправленного списка, заданного указателями nsp

// и ksp, или NULL, если элемент в списке не найден.

{

**node \*q;**

**node \*Res;**

**Res = NULL; q = ksp;**

while **(q!=NULL && Res==NULL)**

if **((\*q).elem==el) Res = q;**

else **q = (\*q).pred;**

return **Res;**

}

void **Spisok::InsAfter (**int **el,node \*Res)**

//Вставка звена с информационным полем el в

//в двунаправленный список, заданный указателями

// nsp и ksp, после звена, на которое указывает Res.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

if **((\*Res).sled!=NULL)**

**{**

**(\*q).sled = (\*Res).sled;**

**(\*q).pred = (\*(\*Res).sled).pred;**

**(\*(\*Res).sled).pred = q; (\*Res).sled = q;**

**}**

else

**{**

**(\*q).sled = NULL;**

**(\*q).pred = Res; ksp = q; (\*Res).sled = q;**

**}**

}

void **Spisok::InsBefore (**int **el,node \*Res)**

//Вставка звена с информационным полем el в

//в двунаправленный список, заданный указателями

// nsp и ksp, перед звеном, на которое указывает Res.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

**(\*q).sled = (\*(\*Res).pred).sled;**

**(\*q).pred = (\*Res).pred;**

**(\*(\*Res).pred).sled = q; (\*Res).pred = q;**

}

void **Spisok::Delete (node \*Res)**

//Удаление звена из двунаправленного списка.

// nsp - указатель на начало списка,

// ksp - указатель на конец списка,

// Res - указатель на удаляемое звено.

{

if **((\*Res).sled!=NULL)**

**{**

**(\*(\*Res).sled).pred = (\*Res).pred;**

**(\*(\*Res).pred).sled = (\*Res).sled;**

delete **Res;**

**}**

else

**{**

**(\*(\*Res).pred).sled = NULL; ksp = (\*ksp).pred;**

delete **Res;**

**}**

}

void **Spisok::DelAfter (node \*Res)**

//Удаление звена из двунаправленного списка.

// nsp - указатель на начало списка,

// ksp - указатель на конец списка,

// Res - указатель на звено, предыдущее удаляемому.

{

**node \*q;**

if **((\*Res).sled==NULL) cout<<**"Указано последнее звено!\n";

else

if **((\*(\*Res).sled).sled!=NULL)**

**{**

**q = (\*Res).sled;**

**(\*(\*(\*Res).sled).sled).pred = Res;**

**(\*Res).sled = (\*(\*Res).sled).sled;**

delete **q;**

**}**

else

**{**

**q = (\*Res).sled; (\*Res).sled = NULL;**

**ksp = (\*ksp).pred;** delete **q; }**

}

void **Spisok::Ochistka ()**

//Удаление двунаправленного списка из памяти.

// nsp - указатель на заглавное звено списка,

// ksp - указатель на последнее звено списка.

{

**node \*q,\*q1;**

**q = nsp; q1 = (\*q).sled;**

while **(q1!=NULL)**

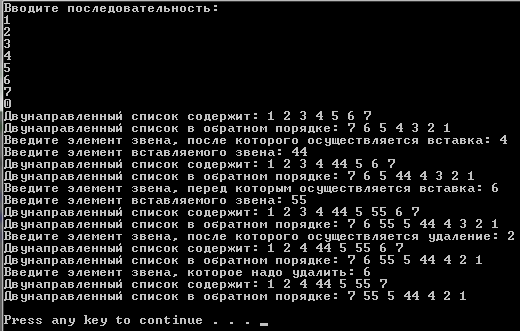
**{**

**q = q1; q1 = (\*q1).sled;** delete **q;**

**}**

delete **nsp; nsp = ksp = NULL;**

}



На следующем шаге мы рассмотрим **двунаправленные кольцевые списки**.

## Двунаправленные кольцевые списки

На этом шаге мы познакомимся с двунаправленными кольцевыми списками.

В программировании двунаправленные списки часто преобразовывают следующим образом: "обычный" линейный двунаправленный список замыкают в своеобразное "кольцо": при движении по списку **в прямом направлении** можно от последнего звена переходить к звену, следующему прямо за заглавным звеном, а при движении **в обратную сторону** - от заглавного звена переходить сразу к последнему звену. В связи с этим мы будем называть двунаправленные списки подобного типа **кольцевыми двунаправленными списками (двунаправленными кольцами** или просто **кольцами**).

Подобная организация списка упрощает процедуру поиска или перебора звеньев с любого места списка с автоматическим переходом от конца к началу или наоборот.

Мы приведем два варианта структуры кольцевых двунаправленных списков.

Рассмотрим первый вариант структуры кольцевого списка, которую мы будем называть **кольцевым** **двунаправленным списком с удаленным заглавным звеном**. Уже само название структуры говорит о том, что заглавное звено двунаправленного списка в кольцо не включается.

Взгляните на приведенную ниже схему:

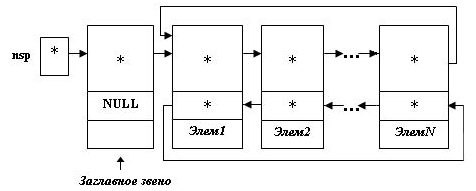


Рис.1. Кольцевой двунаправленный список с удаленным заглавным звеном

Пустое кольцо можно представить следующим образом:

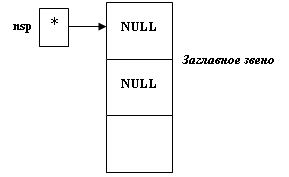


Рис.2. Пустой кольцевой двунаправленный список

Проще всего построить кольцевой двунаправленный список с удаленным заглавным звеном с использованием алгоритма построения двунаправленного списка с заглавным звеном и последующего "замыкания" кольца. Это реализовано в приведенной ниже функции:

void **BuiltRing (node \*\*nsp)**

// Построение двунаправленного кольцевого списка nsp

// с удаленным заглавным звеном.

// \*nsp - указатель на заглавное звено списка.

{

**node \*r;**

int **el;**

// Построим заглавное звено будущего кольцевого списка.

**\*nsp =** new(node);

**r = \*nsp; (\*\*nsp).pred = NULL; (\*\*nsp).sled = NULL;**

**cout<<**"Вводите элементы звеньев списка: \n";

**cin>>el;**

while **(el!=0)**

**{ (\*r).sled =** new(node); (\*((\*r).sled)).pred = r;

**r = (\*r).sled; (\*r).sled = NULL; (\*r).elem = el;**

**cin>>el; }**

// Образуем кольцевой список с удаленным заглавным звеном.

if **((\*\*nsp).sled!=NULL)**

**{(\*((\*\*nsp).sled)).pred = r; (\*r).sled = (\*\*nsp).sled;}**

else **cout<<**"Кольцевой список пуст!\n";

}

Обход кольцевого списка реализовать несложно. Приведем, например, функцию, позволяющую обойти кольцо "по часовой стрелке":

void **VyvodLeftRight (node \*\*nsp)**

// Вывод содержимого двунаправленного кольцевого списка

// с удаленным заглавным звеном "по часовой стрелке".

// \*nsp - указатель на заглавное звено списка.

{

**node \*r;**

**cout<<**"Кольцевой список: ";

if **((\*\*nsp).sled!=NULL)**

**{ cout<<(\*((\*\*nsp).sled)).elem)<<**" ";

**r = (\*((\*\*nsp).sled)).sled;**

while **(r!=(\*\*nsp).sled)**

**{ cout<<(\*r).elem)<<**" "; r = (\*r).sled; }

**cout<<endl;}**

else **cout<<**"пуст!";

}

Пример. Построение двунаправленного кольца с удаленным заглавным звеном, вывод содержимого кольцевого списка, вставка звена, удаление звена.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **elem;**

**node \*sled;**

**node \*pred;**

};

class **Spisok**

{

private:

**node \*nsp;**

public:

**Spisok() {nsp=NULL;}**

void **BuiltRing ();**

void **VyvodLeftRight ();**

void **VyvodRightLeft ();**

void **InsAfter (node\*,**int);

void **InsBefore (node\*,**int);

void **Delete (node\*);**

void **DelAfter (node\*);**

**node \*SearchRing (**int);

void **Ochistka();**

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

**node \*Res;**

int **el,el1;**

**A.BuiltRing ();**

**cout<<**"Содержимое кольца 'по часовой стрелке': \n";

**A.VyvodLeftRight ();**

**cout<<**"Содержимое кольца'против часовой стрелки': \n";

**A.VyvodRightLeft ();**

**cout<<**"Введите элемент звена, после которого ";

**cout<<**"осуществляется вставка: ";

**cin>>el;**

**cout<<**"Введите элемент вставляемого звена: ";

**cin>>el1;**

**Res = A.SearchRing (el);**

if **(Res!=NULL)**

**{A.InsAfter (Res,el1); A.VyvodLeftRight ();}**

else **cout<<**"Звена с таким элементом в списке нет!\n";

**cout<<**"Введите элемент звена, перед которым ";

**cout<<**"осуществляется вставка: ";

**cin>>el;**

**cout<<**"Введите элемент вставляемого звена: ";

**cin>>el1;**

**Res = A.SearchRing (el);**

if **(Res!=NULL)**

**{ A.InsBefore (Res,el1); A.VyvodLeftRight (); }**

else **cout<<**"Звена с таким элементом в списке нет!\n";

**cout<<**"Введите элемент звена, который ";

**cout<<**"надо удалить: ";

**cin>>el;**

**Res = A.SearchRing (el);**

if **(Res!=NULL)**

**{ A.Delete (Res); A.VyvodLeftRight (); }**

else **cout<<**"Звена с таким элементом в списке нет!\n";

**cout<<**"Введите элемент звена, после которого ";

**cout<<**"осуществляется удаление: ";

**cin>>el;**

**Res = A.SearchRing (el);**

if **(Res!=NULL)**

**{ A.DelAfter (Res); A.VyvodLeftRight (); }**

else **cout<<**"Звена с таким элементом в списке нет!\n";

**A.Ochistka();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::BuiltRing ()**

// Построение двунаправленного кольцевого списка nsp

// с удаленным заглавным звеном.

// nsp - указатель на заглавное звено списка.

{

**node \*r;**

int **el;**

//Построим заглавное звено кольцевого списка.

**nsp =** new(node);

**r = nsp; (\*nsp).pred = NULL; (\*nsp).sled = NULL;**

**cout<<**"Вводите элементы списка: \n";

**cin>>el;**

while **(el!=0)**

**{**

**(\*r).sled =** new **(node);**

**(\*((\*r).sled)).pred = r; r = (\*r).sled;**

**(\*r).sled = NULL; (\*r).elem = el;**

**cin>>el;**

**}**

//А теперь - образуем кольцевой список!

if **((\*nsp).sled!=NULL)**

**{ (\*((\*nsp).sled)).pred = r; (\*r).sled = (\*nsp).sled; }**

else

**cout<<**"Кольцевой список пуст!\n";

}

void **Spisok::VyvodLeftRight ()**

// Вывод содержимого двунаправленного кольцевого списка

// с удаленным заглавным звеном "по часовой стрелке".

// nsp - указатель на заглавное звено списка.

{

**node \*r;**

**cout<<**"Кольцевой список: ";

if **((\*nsp).sled!=NULL)**

**{**

**cout<<(\*((\*nsp).sled)).elem<<**" ";

**r = (\*((\*nsp).sled)).sled;**

while **(r!=(\*nsp).sled)**

**{ cout<<(\*r).elem<<**" "; r = (\*r).sled; }

**cout<<endl;**

**}**

else **cout<<**"пуст!";

}

void **Spisok::VyvodRightLeft ()**

// Вывод содержимого двунаправленного кольцевого списка

// с удаленным заглавным звеном "против часовой стрелки".

// nsp - указатель на заглавное звено списка.

{

**node \*r;**

**cout<<**"Кольцевой список: ";

if **((\*nsp).sled!=NULL)**

**{**

**cout<<(\*((\*((\*nsp).sled)).pred)).elem<<**" ";

**r = (\*((\*((\*nsp).sled)).pred)).pred;**

while **(r!=(\*((\*nsp).sled)).pred)**

**{ cout<<(\*r).elem<<**" "; r = (\*r).pred; }

**cout<<endl;**

**}**

else **cout<<**"пуст!";

}

node \*Spisok::SearchRing (int **el)**

// Поиск элемента el в кольцевом двунаправленном списке

// с удаленным заглавным звеном.

// nsp - указатель на заглавное звено списка.

{

**node \*q;**

**node \*p;**

**node \*Res;**

**Res = NULL; p = nsp;**

if **((\*((\*p).sled)).elem==el) Res = (\*p).sled;**

else

**{**

**q = (\*((\*p).sled)).sled;**

while **(q!=(\*p).sled && Res==NULL)**

if **((\*q).elem==el) Res = q;**

else **q = (\*q).sled;**

**}**

return **Res;**

}

void **Spisok::InsAfter (node \*Res,**int **el)**

// Вставление в кольцевой двунаправленный список звена

// с информационным полем el после звена, на которое

// указывает ссылка Res.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el; (\*q).sled = (\*Res).sled;**

**(\*q).pred = (\*(\*Res).sled).pred;**

**(\*(\*Res).sled).pred = q; (\*Res).sled = q;**

}

void **Spisok::InsBefore (node \*Res,**int **el)**

// Вставка в кольцевой двунаправленный список звена

// с информационным полем el перед звеном, на которое

// указывает ссылка Res.

// nsp - указатель на заглавное звено списка.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

**(\*q).sled = (\*(\*Res).pred).sled; (\*q).pred = (\*Res).pred;**

**(\*(\*Res).pred).sled = q; (\*Res).pred = q;**

if **(Res==(\*nsp).sled) (\*nsp).sled = q;**

}

void **Spisok::Delete (node \*Res)**

// Удаление из кольцевого двунаправленного списка

// звена, на которое указывает ссылка Res.

// nsp - указатель на заглавное звено списка.

{

if **((\*Res).sled==Res)**

**{ (\*nsp).sled = NULL;** delete **Res; }**

else

**{**

**(\*(\*Res).sled).pred = (\*Res).pred;**

**(\*(\*Res).pred).sled = (\*Res).sled;**

if **((\*nsp).sled==Res)**

// Удаляем "первое" звено кольца.

**(\*nsp).sled = (\*Res).sled;**

delete **Res;**

**}**

}

void **Spisok::DelAfter (node \*Res)**

// Удаление из кольцевого двунаправленного списка звена,

// расположенного после звена, на которое указывает

// ссылка Res.

// nsp - указатель на заглавное звено списка.

{

**node \*q;**

if **((\*Res).sled==Res)**

**{ (\*nsp).sled = NULL;** delete **Res;}**

else

**{**

**q = (\*Res).sled;**

**(\*(\*(\*Res).sled).sled).pred = (\*(\*Res).sled).pred;**

**(\*Res).sled = (\*(\*Res).sled).sled;**

if **((\*(\*nsp).sled).pred==Res)**

// Удаляем "последнее" звено кольца.

**(\*nsp).sled = (\*Res).sled;**

delete **q;**

**}**

}

void **Spisok::Ochistka()**

{

**node \*q,\*q1;**

**q = (\*((\*nsp).sled)).sled;**

**q1 = (\*q).sled;**

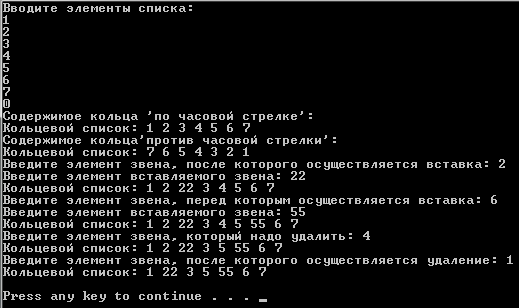
while **(q1!=(\*((\*nsp).sled)).sled)**

**{** delete **q; q=q1; q1=(\*q1).sled; }**

delete **q;**

delete **nsp;**

}



Следует заметить, что предложенный выше способ образования двунаправленного кольцевого списка не является единственно возможным. Имеется еще один вариант представления кольцевых двунаправленных списков, который схематически можно представить так:

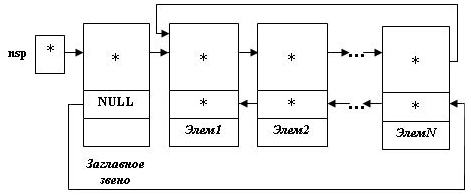


Рис.3. Кольцевой двунаправленный список с включенным заглавным звеном

Мы будем называть подобную структуру **кольцевым двунаправленным списком с включенным заглавным звеном**.

Опишем структуру **пустого кольцевого списка с включенным заглавным звеном**:

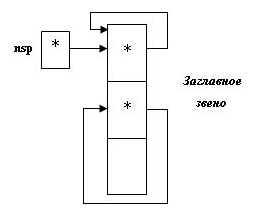


Рис.4. Пустой кольцевой двунаправленный список с включенным заглавным звеном

Каждый из способов образования кольцевого списка имеет как положительные, так и отрицательные стороны. Например, во втором варианте образования списка очень просто реализуется вставка нового звена как в начало списка (после заглавного звена), так и в его конец, ибо вставка нового звена в конец списка эквивалентна его вставке перед заглавным звеном. Однако здесь при циклической обработке элементов списка придется каждый раз проверять, не является ли очередное звено заглавным звеном списка.

Этого недостатка лишен первый способ организации списка, но в этом случае труднее реализуется добавление звена в конец списка.

На следующем шаге мы рассмотрим **деки на базе двунаправленных списков**.

## Деки на базе двунаправленных списков

### Формирование дека и его просмотр

На этом шаге мы рассмотрим **алгоритмы создания и просмотра дека**.

Мы будем моделировать дек с помощью **двунаправленного списка без заглавного звена**.

Вначале опишем алгоритм формирования двунаправленного списка без заглавного звена из двунаправленного списка с заглавным звеном, изображенного ниже:

1. "Начальная" позиция:

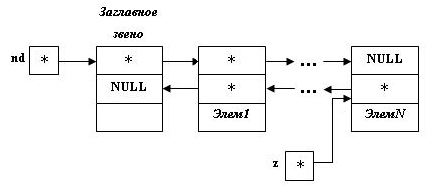


Рис.1. "Начальная" позиция

1. Исключим заглавное звено из списка:

**q = nd;**

**nd = (\*nd).sled;**

**(\*nd).pred = NULL;**

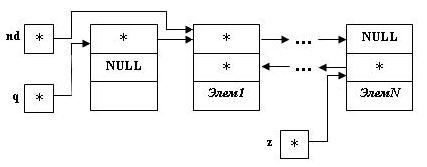


Рис.2. Исключение заглавного звена

1. Возврат памяти в кучу и "настройка" указателя на конец дека:

delete **q;**

**kd = z;**

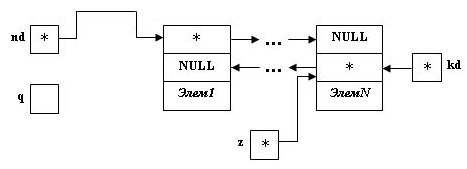


Рис.3. "Настройка" указателя и возврат памяти в кучу

Очевидно, что суть алгоритма состоит в "отбрасывании" заглавного звена двунаправленного списка.

Оформим данный алгоритм в виде функции:

void **BuiltDeck (node \*\*nd, node \*\*kd)**

// Построение дека на базе двунаправленного

// списка с заглавным звеном.

// \*nd - указатель на начало дека.

// \*kd - указатель на конец дека.

{

**node \*q, \*z;**

int **el;**

// Построение заглавного звена.

**\*nd =** new(node);

**z = \*nd;**

**(\*\*nd).pred = (\*\*nd).sled = NULL;**

**cout<<**"Введите последовательность: \n";

**cin>>el;**

while **(el!=0)**

**{ (\*z).sled =** new(node);

**(\*((\*z).sled)).pred = z; z = (\*z).sled;**

**(\*z).sled = NULL; (\*z).elem = el; cin>>el; }**

if **((\*\*nd).sled!=NULL)**

**{ q = \*nd; \*nd = (\*\*nd).sled; (\*\*nd).pred = NULL; \*kd = z;** delete **q; }**

else

**{** delete **\*nd; \*nd = \*kd = NULL;}**

}

Со следующего шага мы начнем рассматривать основные операции над деками.

### Добавление звена в начало дека

На этом шаге мы рассмотрим **алгоритм добавления звена в начало дека**.

Алгоритм добавления звена в начало дека заключается в создании из кучи нового элемента и настройки указателя начала дека на новый элемент:

**q =** new(node);

**(\*q).elem = Элем;**

**(\*q).sled = nd; (\*q).pred = NULL;**

**(\*nd).pred = q; nd = q;**

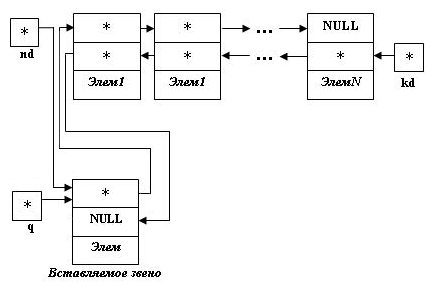


Рис.1. Вставка звена в начало дека

Приведем текст функции:

void **InsLeft (node \*\*nd, node \*\*kd,**int **el)**

// Вставка звена, содержащего элемент el, в дек слева.

// \*nd - указатель на начало дека.

// \*kd - указатель на конец дека.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

if **(\*nd==NULL)**

**{** // Если дек пуст, то...

**\*nd = q; (\*q).sled = (\*q).pred = NULL; \*kd = q;}**

else

**{ (\*q).sled = \*nd; (\*q).pred = NULL; (\*\*nd).pred = q; \*nd = q;}**

}

На следующем шаге мы рассмотрим **алгоритм добавления звена в дек справа**.

### Добавление звена в конец дека

На этом шаге мы рассмотрим алгоритм добавления звена в конец дека.

Алгоритм добавления звена в конец дека заключается в выделении памяти под новый элемент и включение его в конец дека:

**q =** new(node);

**(\*q).elem = Элем;**

**(\*q).sled = NULL;**

**(\*q).pred = \*kd;**

**(\*kd).sled = q;**

**\*kd = q;**

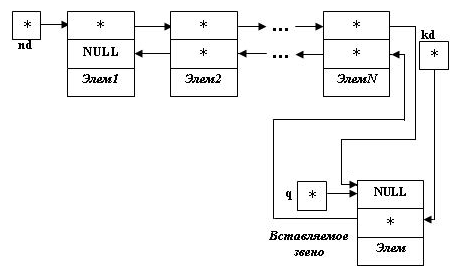


Рис.1. Вставка звена в конец дека

Приведем текст функции:

void **InsRight (node \*\*nd, node \*\*kd,**int **el)**

// Добавление звена, содержащего элемент el, в дек справа.

// \*nd - указатель на начало дека.

// \*kd - указатель на конец дека.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

if **(\*kd==NULL)**

**{**// Если дек пуст, то...

**\*nd = q; (\*q).sled = (\*q).pred = NULL; \*kd = q;}**

else

**{ (\*q).sled = NULL; (\*q).pred = \*kd; (\*\*kd).sled = q; \*kd = q;}**

}

Со следующего шага мы начнем рассматривать **алгоритмы удаления звена из дека**.

### Удаление звена из дека слева

На этом шаге мы рассмотрим **первый алгоритм удаления звена дека**.

Рассмотрим **алгоритм удаления звена слева**.

1. Пусть уже построен дек на базе двунаправленного списка без заглавного звена:

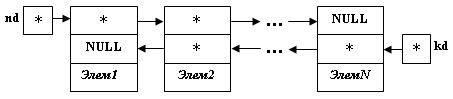


Рис.1. "Начальная" позиция

1. Сохраним адрес удаляемого элемента и "настроим" указатель начала дека:

**q = nd;**

**nd = (\*nd).sled;**

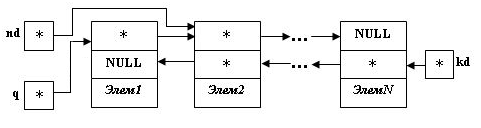


Рис.2. "Настройка" указателей

1. Возвратим память в кучу:

delete **q;**

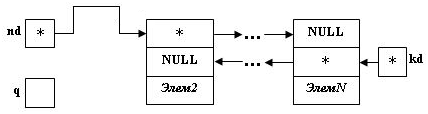


Рис.3. Возврат памяти в кучу

Запишем по описанному алгоритму функцию на языке C++:

void **DelLeft (node \*\*nd, node \*\*kd,**int **\*el)**

// Удаление звена из дека слева с помещением элемента

// удаляемого звена в переменную el.

// \*nd - указатель на начало дека.

// \*kd - указатель на конец дека.

{

**node \*q;**

if **((\*\*nd).sled!=NULL)**

**{ q = \*nd; \*el =(\*q).elem; \*nd = (\*\*nd).sled; (\*\*nd).pred = NULL;** delete **q;}**

else

**{** //В деке находится один элемент.

**q = \*nd; \*el =(\*q).elem; \*nd = \*kd = NULL;**

delete **q; cout<<**"Дек пуст!\n";}

}

На следующем шаге мы рассмотрим **алгоритм удаления дека справа**.

### Удаление звена из дека справа

На этом шаге мы рассмотрим **алгоритм удаления звена из дека справа**.

Рассмотрим **алгоритм удаления звена справа**. Пусть уже построен дек на базе двунаправленного списка без заглавного звена:

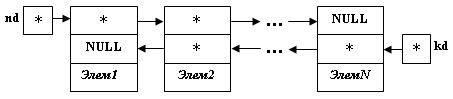


Рис.1. "Начальная" позиция

Приступим к пошаговому выполнению алгоритма:

1. Сохраним адрес удаляемого элемента и "настроим" указатель конца дека:

**q = kd;**

**kd = (\*kd).pred;**

**(\*kd).sled = NULL;**

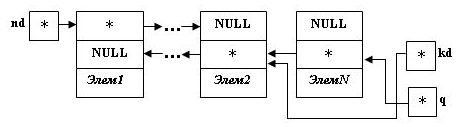


Рис.2. "Настройка" указателей

1. Возвратим память в кучу:

delete **q;**

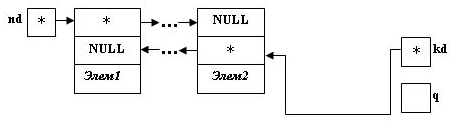


Рис.3. Возврат памяти в кучу

Запишем функцию на языке C++, реализующую рассмотренный алгоритм:

void **DelRight (node \*\*nd, node \*\*kd,**int **\*el)**

// Удаление звена из дека справа с помещением элемента

// удаляемого звена в переменную el.

// \*nd - указатель на начало дека.

// \*kd - указатель на конец дека.

{

**node \*q;**

if **((\*\*kd).pred!=NULL)**

**{ q = \*kd; \*el =(\*q).elem; \*kd = (\*\*kd).pred; (\*\*kd).sled = NULL;** delete **q;}**

else

**{** // В деке находится один элемент.

**q = \*kd; \*el =(\*q).elem; \*nd = \*kd = NULL;**

delete **q; cout<<**"Дек пуст!\n";}

}

Приведем обобщающий пример. Построение дека, вставка и удаление элементов из дека.

#include<iostream>

usingnamespace **std;**

struct **node**

{

int **elem;**

**node \*sled;**

**node \*pred;**

};

class **Spisok**

{

private:

**node \*nd;**//Указатель на начало дека.

**node \*kd;**//Указатель на конец дека.

int **klad;**//Информационное поле удаленного элемента.

public:

void **BuiltDeck ();**

void **VyvodDeck ();**

void **InsLeft (**int);

void **InsRight (**int);

void **DelLeft ();**

void **DelRight ();**

int **Get\_Klad () {**return **klad;}**

void **Ochistka();**

};

void **main ()**

{

**setlocale(LC\_ALL,**"Rus");

**Spisok A;**

int **el;**

**A.BuiltDeck ();**

**A.VyvodDeck ();**

**cout<<**"Введите элемент звена, вставляемого справа: ";

**cin>>el; A.InsRight (el); A.VyvodDeck ();**

**cout<<**"Введите элемент звена, вставляемого слева: ";

**cin>>el; A.InsLeft (el); A.VyvodDeck ();**

**cout<<**"Удалим звено справа: \n";

**A.DelRight (); A.VyvodDeck ();**

**cout<<**"Был удален элемент: "<<A.Get\_Klad()<<endl;

**cout<<**"Удалим звено слева: \n";

**A.DelLeft (); A.VyvodDeck ();**

**cout<<**"Был удален элемент: "<<A.Get\_Klad()<<endl;

**A.Ochistka();**

**cout <<** "\n";

**system(**"PAUSE");

}

void **Spisok::BuiltDeck ()**

// Построение дека на базе двунаправленного

// списка с заглавным звеном.

// nd - указатель на начало дека,

// \*kd - указатель на конец дека.

{

**node \*q;**

**node \*z;**

int **el;**

**nd =** new(node);

**z = nd;**

**(\*nd).pred = (\*nd).sled = NULL;**

**cout<<**"Введите последовательность: \n";

**cin>>el;**

while **(el!=0)**

**{ (\*z).sled =** new **(node);**

**(\*((\*z).sled)).pred = z;**

**z = (\*z).sled; (\*z).sled = NULL;**

**(\*z).elem = el; cin>>el;}**

if **((\*nd).sled!=NULL)**

**{ q = nd; nd = (\*nd).sled; (\*nd).pred = NULL;**

**kd = z;** delete **q; }**

else

**{** delete **nd; nd = kd = NULL; }**

}

void **Spisok::VyvodDeck ()**

// Вывод содержимого дека.

// nd - указатель на начало дека.

{

**node \*z;**

**z = nd; cout<<**"Содержимое дека: ";

if **(z!=NULL)**

while **(z!=NULL)**

**{ cout<<(\*z).elem<<**" "; z = (\*z).sled; }

else **cout<<**"он пуст!\n";

**cout<<endl;**

}

void **Spisok::InsLeft (**int **el)**

// Добавление звена, содержащего элемент el, в дек слева.

// nd - указатель на начало дека,

// kd - указатель на конец дека.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

if **(nd==NULL)**

**{ nd = q; (\*q).sled = (\*q).pred = NULL; kd = q; }**

else

**{ (\*q).sled = nd; (\*q).pred = NULL;**

**(\*nd).pred = q; nd = q; }**

}

void **Spisok::InsRight (**int **el)**

// Добавление звена, содержащего элемент el, в дек справа.

// nd - указатель на начало дека,

// kd - указатель на конец дека.

{

**node \*q;**

**q =** new(node);

**(\*q).elem = el;**

if **(kd==NULL)**

**{ nd = q; (\*q).sled = (\*q).pred = NULL; kd = q; }**

else

**{ (\*q).sled = NULL; (\*q).pred = kd;**

**(\*kd).sled = q; kd = q; }**

}

void **Spisok::DelLeft ()**

// Удаление звена из дека слева с помещением

// элемента удаляемого звена в переменную klad.

// nd - указатель на начало дека,

// kd - указатель на конец дека.

{

**node \*q;**

if **((\*nd).sled!=NULL)**

**{ q = nd; klad =(\*q).elem;**

**nd = (\*nd).sled; (\*nd).pred = NULL;** delete **q;}**

else

**{** // В деке находится один элемент.

**q = nd; klad =(\*q).elem;**

**nd = kd = NULL;** delete **q;cout<<**"Дек пуст!\n"; }

}

void **Spisok::DelRight ()**

// Удаление звена из дека справа с помещением

// элемента удаляемого звена в переменную klad.

// nd - указатель на начало дека,

// kd - указатель на конец дека.

{

**node \*q;**

if **((\*kd).pred!=NULL)**

**{ q = kd; klad =(\*q).elem;**

**kd = (\*kd).pred; (\*kd).sled = NULL;** delete **q; }**

else

**{**// В деке находится один элемент.

**q = kd; klad =(\*q).elem;**

**nd = kd = NULL;** delete **q; cout<<**"Дек пуст!\n"; }

}

void **Spisok::Ochistka()**

{

**node \*q,\*q1;**

**q = nd;**

**q1 = (\*q).sled;**

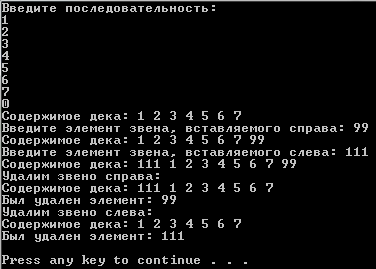
while **(q1!=NULL)**

**{** delete **q; q = q1; q1 = (\*q).sled;}**

delete **q;**

**nd = kd = NULL;**

}



Со следующего шага мы начнем знакомиться с **бинарными деревьями**.