1. Связные списки

Список – это последовательность однотипных элементов, связанных между собою указателями.

Элемент списка – узел (Node), см. рис 13. Простейший  узел включает в себя два поля:

§  *информационное поле* - данные (переменная, структура данных, объект класса и т. д.)

§  *адресное поле*- указатель на следующий/предыдущий/связанный узел в списке.



Рис.13  Модель узла односвязного списка.

Элементы связанного списка можно добавлять и удалять произвольным образом. Специфика  динамических связных списков:

·     для отдельных элементов, узлов структуры память выделяется в тот момент, когда в них появляется необходимость (а не сразу и одним блоком как для массивов).

·     число элементов динамической структуры заранее не объявляется и может изменяться от нуля до некоторого значения, определяемого спецификой соответствующей задачи или доступным объемом памяти.

·     память, занимаемая структурой, не представляет собой непрерывную область, т.е. элементы могут быть разбросаны в памяти хаотическим образом.

·     логическая последовательность элементов задается в явном виде с помощью одного или нескольких указателей, хранящихся в самих элементах.

·     каждый элемент, кроме своего значения, хранит указатель на следующий элемент или на несколько соседних с ним элемента.

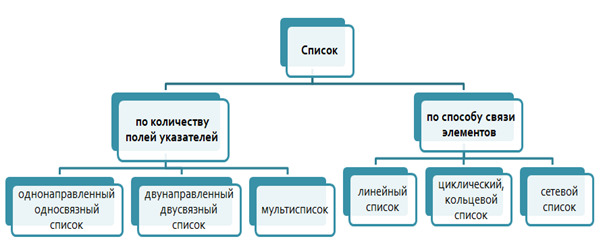


Рис. 14 Классификация списков

***По количеству полей указателей*** (рис.14) различают однонаправленный (односвязный), двунаправленный (двусвязный) список и мультисписок. Связный список, элемент которого имеет только один указатель на следующий узел, называется ***односвязным***. Связный список, элемент которого имеет два поля указателя – на следующий элемент и на предыдущий, называется ***двусвязным***. Связный список, элемент которого имеет ***n***  полей указателя, называется ***мультисписком***или ***n- связным списком***.

***По способу связи элементов*** (рис.15) различают линейные и циклические(кольцевые) списки. Связный список, в котором, последний элемент указывает на NULL, называется ***линейным***. Связный список, в котором последний элемент связан с первым, называется ***циклическим***или ***кольцевым***.

Операции  связного списка:

·     обход элементов списка в порядке их расположения;

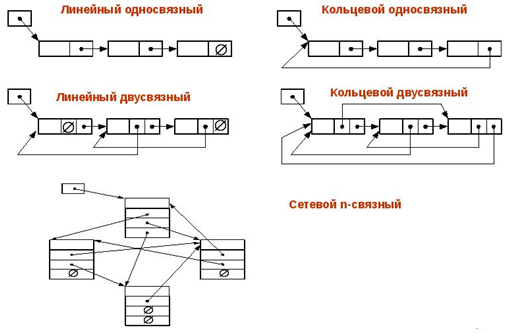
·     добавление/включение/вставка элемента(узла) в список;

·     удаление/исключение  элемента(узла) из списка;

·     поиск элементов списка по разным критериям;

·     очистка списка;

·     проверка на пустоту списка.



2. Алгоритмы обработки односвязных списков

Характеристика односвязного списка:

§  каждый элемент/узел (Node)  списка имеет указатель (next) на следующий элемент;

§  указатель на первый элемент,  является началом (head) списка или головным элементом списка, он не содержит данных;

§  указатель на текущий элемент списка temp, для обработки списка;

§  признак конца списка - последний элемент имеет в адресном поле next значение NULL; алгоритмы обработки списка будет контролировать именно это значение.

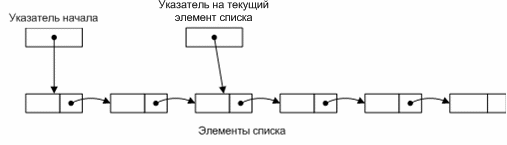




Рис. 16 Модели односвязного линейного списка

Специфика алгоритмов обработки списка:

§  можно передвигаться *только* *в сторону конца*списка

§  узнать адрес предыдущего элемента, опираясь только на содержимое текущего узла, *невозможно*.

Программирование переменных односвязного списка:

1.     Описание структуры данных узла как типа данных:

struct  Node {

     int data;   //информационное поле

     Node \*next; //адресное поле

};

2.     Объявление переменной начала (головы) списка:

Node \*head = NULL;

*Алгоритм добавления/включения узла  в начало (head)  списка*

3.     Создать новый узел:

     Node \*temp = new Node;

4.     Присвоить значения полям узла, где val – новое значение информационного поля. Новый узел должен указывать на начало (head) списка:

     temp->next = head;  //указываем на начало списка

     temp->data = val;

5.     Новый узел должен стать первым,  для этого head должен указывать на новый узел:

          head = temp;

Модель алгоритма представлена на рис.17.

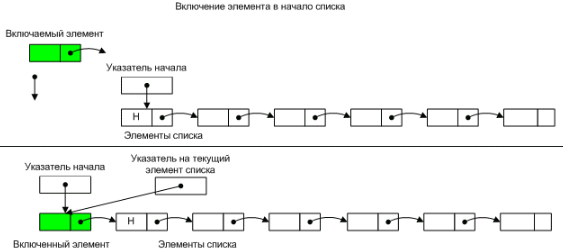


Рис.17 Модель  операции включения элемента/узла в начало списка.

Разработаем функцию программы на основе данного алгоритма, чтобы использовать многократно для добавления/включения узлов в начало списка.

**Программа 1.47** Функция алгоритма добавления/включения узла в начало списка.  Входными параметрами функции являются голова (head) списка и значение поля данных (val) включаемого элемента. Возвращает функция новое значение адреса начала списка(head).

//-------------------------------------------------------------

Node\* addHead (Node \*head, int val){

 Node \*temp = new Node;  //новый узел – включаемый элемент

  temp->next = head;

  temp->data = val;

 head = temp;

 return head;

}

//---------вызов функции для построения списка----------------------

int main()

{

     head = addHead(head, -3);

     head = addHead(head, 7);

     head = addHead(head, 0);

     outList(head);      // вывод в процессе обхода списка

}

//-------------------------------------------------------------

#### **2.1. Алгоритм полного обхода узлов списка в порядке их расположения**

Операция обхода является базовой для многих алгоритмов работы с узлами списка, таких как вывод, поиск, вычисления и преобразования списка. Суть ее в том, что надо организовать движение по узлам списка, начиная с головы.  Для этого создают указатель на текущий элемент, устанавливают его на начало списка и, последовательно изменяя его значение, продвигают по списку до конца, см. рис. 18.



Рис.18 Модель  обхода элементов/узлов списка.

1.       Создать указатель на текущий элемент и установить его на начало списка:

Node \*temp = head;

2.       Если указатель на текущий элемент temp равен NULL (дошли до конца списка), то остановиться:

while (temp != NULL)…

3.       Выполнить действия над текущим узлом temp (изменить, вывести данные, операция сравнения и др.)

4.       Перейти к следующему узлу, присвоив ему указатель следующего элемента, значение которого содержит поле   next текущего узла:

temp = temp->next;

Псевдокод общего алгоритма обхода следующий:

**Node \*temp = head;**

**while (temp != NULL) {**

**…                    // действия с текущим узлом**

**temp = temp->next;**

**}**

**Программа 1.48** Функции алгоритма обхода для вывода данных из узлов списка и  подсчета количества узлов.

//-----------вывод--------------------------------------------------

void outList (Node \*head){

 Node \*temp = head;

 while (temp != NULL){

         cout << setw(5) << temp->data << ' ';

         temp = temp->next;

 }

 cout << endl;

}

//-------количество узлов-------------------------------------------

 int countNode (Node \*head){

  Node \*temp = head;

int count = 0;

 while (temp != NULL){

         count++;

         temp = temp->next;

 }

 return count;

}

//-------------------------------------------------------------

#### **2.2. Алгоритмы поиска элементов списка по разным критериям**

Односвязный список является структурой последовательного доступа к элементам и поэтому  алгоритмы поиска линейны и основаны на *алгоритме обхода элементов* списка.

Критерии поиска могут быть различными в зависимости от данных списка и поставленной задачи, но и универсальными алгоритмами, которые используются в операциях над списком, будут:

a)     поиск по позиции(номер) узла в списке;

b)    поиск по значению данных в узле списка.

Рассмотрим функции данных алгоритмов. Функции как результат возвращают указатель на искомый  узел или NULL (в случае неудачного поиска).

**Программа 1.49** Функции алгоритма поиска по значению и позиции узла списка.

//----------------по значению---------------------------------------

Node\* findData (Node \*head, int val){

     Node \*temp = head;

 while (temp != NULL){

          if (temp->data == val) return temp;

         temp = temp->next;

 }

 return NULL;

}

//----------------по позиции---------------------------------------

Node\* findPos (Node \*head, int pos){

     Node \*temp = head;

     while (temp != NULL && pos>1){

            temp = temp->next;

       pos--;

 }

 return temp;

}

//-------------------------------------------------------------

#### **2.3. Алгоритмы вставки/включения узла в список**

Алгоритмы вставки узла в список:

a)     вставка узла в указанную позицию;

b)    вставка узла после указанного;

c)     вставка узла пред указанным;

d)    вставка/добавление узла в конец списка.

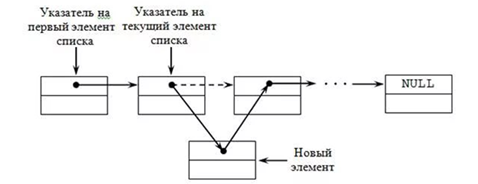


Рис.19 Операция вставки элемента/узла в список.

Программируя алгоритмы вставки, необходимо решать проблему адреса предыдущего узла, так как  идти назад нельзя. Эту проблему можно решать по-разному, но одним из решений будет применение функций поиска узлов по разным критериям. Например, необходимо вставить новый узел на позицию 7. Значит, надо знать адрес узла на позиции 6. Эту задачу решит функция findPos. Модель решения представлена на рис 20.

В алгоритмах вставки важно корректно присвоить адреса указателям соседних узлов, чтобы вставляемый узел не разбил цепочку узлов и встал на заданную позицию.

Вставляемый узел может быть задан двумя способами:

a)   только значением информационного поля данных и тогда в алгоритме надо создать новый узел;

b)  адресом узла, который надо вставить/включить в список.

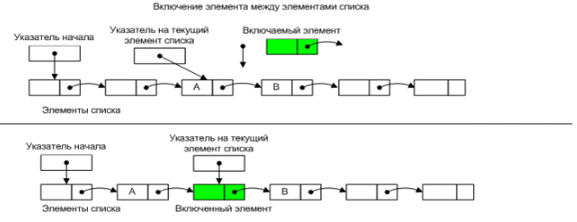


Рис.20 Модель  операции вставки элемента/узла между элементами списка.

**Программа 1.50** Функция алгоритма вставки на указанную  позицию узла списка, заданного значением поля данных. Как результат функция возвращает логическое значение – признак  выполнения/невыполнения операции, а в списке появляется новый узел, что проверяется выводом списка. Модель операции см. Рис. 20.

//-------------------------------------------------------------

bool insertPos(Node\* head, int pos, int val){

     if (pos == 1)  return false; // функция addHead

     Node\* p = findPos(head, pos-1); //поиск предыдущего узла

     if (p == NULL)  return false;

     Node \* temp = new Node;

      temp->data=val;

      temp->next=p->next;

      p->next = temp;

     return true;

}

//-------проверка работы функций------------------------------------

int main(){

head = addHead(head, -3);   head = addHead(head, 7);

head = addHead(head, 9);    head = addHead(head, 0);

outList(head);

      cout << findData(head, -3)->data<<endl;

cout << findPos(head, 2)->data<<endl;

cout << insertPos(head, 3, 44);

outList(head);

std::cout << insertPos(head, 1, 99);

     }

//-------------------------------------------------------------

**Программа 1.51** Функция алгоритма вставки после указанного узла списка. Здесь узел списка задан сразу своим адресом, это входной параметр функции – указатель  \*p. Другой входной параметр функции задает адрес вставляемого узла – указатель \*temp. Модель операции см. рис 19, 20.

//------------------------------------------------------------------

void insertAfter (Node \*p, Node \*temp){

     if (p == NULL) return;

     temp->next = p->next;

     p->next = temp;

}

//----------проверка------------------------------------------------

int main(){

     head=addHead(head, 5);    head = addHead(head, -3);

     outList(head);

     Node\* pF = findPos(head, 2);  //поиск узла списка

     Node\* temp = new Node; temp->data = 55;

     if (pF)           //проверка существования узла

          insertAfter(pF, temp);

     outList(head);

}

//------------------------------------------------------------------

В *алгоритме  добавления узла в конец* списка можно применить обход и достигнув последнего узла, где поле next равно NULL,   присвоить этому полю адрес нового узла. Новый узел можно создать или задать указателем на него. Модель алгоритма операции см. на рис 21.

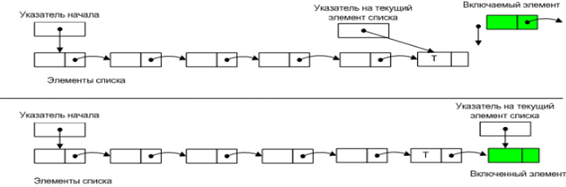


Рис.21 Модель  операции вставки элемента/узла в конец списка.

#### **2.4. Алгоритмы удаления/исключения узла из списка**

Алгоритмы удаления узла из списка:

a)     удаление узла из начала списка;

b)    удаление узла в указанной позиции;

c)     удаление узла после указанного;

d)    удаление  узла пред указанным;

e)     удаление узла из конца списка;

f)      очистка списка – удаление всех узлов списка.

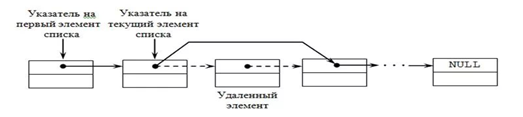


Рис.22 Модель  операции удаления

Удаление узла осуществляется командой освобождения памяти:

*delete temp;*

Программируя алгоритмы удаления, необходимо также решать проблему адреса предыдущего узла, так как  идти назад нельзя. Эту проблему можно решать по-разному, но одним из решений будет применение функций поиска узлов по разным критериям. Например, необходимо удалить новый узел c позицию 10. Значит, надо знать адрес узла на позиции 9. Эту задачу решит функция findPos. Модель решения представлена на рис 24.

В алгоритмах удаления важно корректно присвоить адреса указателям соседних узлов, чтобы удаляемый узел не разорвал цепочку узлов.

Удаляемый узел может быть задан способами:

a)   значением информационного поля данных;

b)   позицией узла в списке;

c)    указателем узла, который надо удалить из списка.

**Программа 1.51** Функция алгоритма  удаления узла из начала списка, см. рис 23.

//-------------------------------------------------------------

Node\*  removeHead (Node  \*head){

   if (head == NULL) return 0;

   Node \*temp = head;   //указатель удаляемого узла

//начало списка - это  указатель на следующий узел из удаляемого

   head = temp->next;

   delete temp;        //удаляем узел

 return head;           //возвращаем новое начало списка

}

//-------------------------------------------------------------

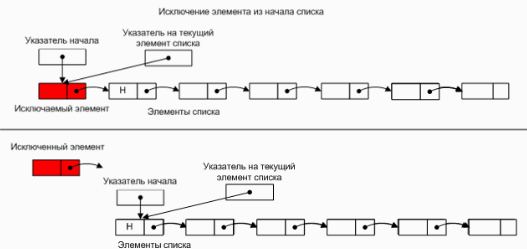


Рис.23 Модель  операции удаления  узла  из начала  списка.

**Программа 1.52** Функция алгоритма  удаления узла из списка, см. рис 22, 24.

Функция универсальна, она удаляет узел по указанному адресу, заданному указателем \*p. Если удаляется голова списка, то значение head тоже меняется, так как этот параметр передается в функцию по ссылке.

//-------------------------------------------------------------

void remove(Node\*& head, Node\* p){

          if (head == p)

                head=removeHead(head);

          Node\* temp= head;

          while (temp->next && temp->next != p)

                temp = temp->next;

          if (temp->next){

                temp->next = temp->next->next;

                delete p;

          }

}

//-------------------------------------------------------------

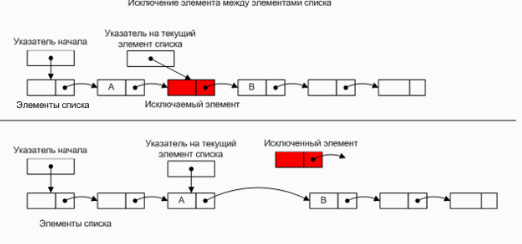


Рис.24 Модель  операции удаления  узла  из  списка.

В *алгоритме  удаления  узла из конца* списка можно применить обход, но идти до предпоследнего узла, где поле temp->next->next равно NULL  и удалить этот узел. А в поле next предпоследнего узла установить значение NULL. Модель алгоритма операции см. на рис 25.

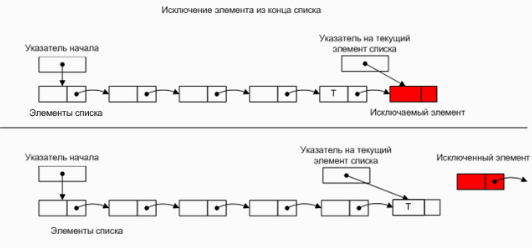


Рис.25 Модель  операции удаления  узла  из  конца списка.

В *алгоритме очистке* списка надо применить  полный обход, где основным действием над узлом будет его удаление и сохранение в текущем указателе следующей ссылки из удаленного узла. Это действие можно заменить вызовом функции removeHead.

Данная операция реализуется путем проверки значения головы списка (head) на наличие в нем нулевого адреса. Или подсчитать количество узлов.

**Программа 1.53** Функция проверки списка на пустоту.

//-------------------------------------------------------------

bool  isEmpty (Node \*head){

      return head== NULL;

}

//-------------------------------------------------------------