Линейный список в C/C++.

Введение.

При работе с большим объемом данных одного и того же типа используют массивы. Массив – это набор из фиксированного количества однотипных данных, которые хранятся в памяти последовательно и доступны по индексу. Размер массива должен быть известен заранее; в языке С, он должен быть известен уже во время компиляции. Но самое главное свойство массивов состоит в том, что если индекс известен, то обращение к любому элементу массива производится за постоянное время.

Однако, чаще всего заранее не известно, какого размера память потребуется для решения задачи, использовать простой массив нельзя. В этом случае используют различные динамические структуры, иным словами, выделение и освобождение памяти в ходе работы программы, и представление ее в виде сложных структур данных: списки, деревья. Еще одно преимущество перед массивами использования динамических структур: простота операций добавления и удаления данных, достаточно определить необходимые связи между элементами, и нет необходимости реорганизовывать весь массив.

Внутренняя (оперативная) память компьютера представляет собой упорядоченную последовательность байтов или машинных слов (ячеек памяти). Номер байта или слова памяти, через который оно доступно как из команд компьютера, так и во всех других случаях, называется **адресом**

I. Динамические переменные.

Ссылки

Для объявления переменной *P* ссылкой на переменную базового типа *X* в языке C++ необходимо описание.

```
ИмяБазовогоТипа X;

ИмяБазовогоТипа &P = X;
```

Ссылка это "псевдоним" переменной. Ссылка инициализируется при объявлении, и изменению не подлежит.

При этом компилятор выделяет место в памяти, имя которого *X* (размер зависит от базового типа), и место в памяти, имя которого *P*, в которую записывается адрес ячейки памяти, выделенной для переменной *X*.

```
int x = 5; //объявляем переменную
int &p = x; //объявляем ссылку, теперь p это псевдоним x
cout << x << ' ' << p << endl; //выведет 5 5
x = 6;
cout << x << ' ' << p << endl; //выведет 6 6
p = 7;
cout << x << ' ' << p << endl; //выведет 7 7
```

Ссылку можно создать как аргумент в функции, чтобы из функции напрямую работать с переменной, которую в неё передали.

```
void func(int &p)
{
    p++; //увеличиваем переданную переменную
}

int main()
{
    int x = 5; //объявляем переменную
    func(x); //передаём переменную.
    cout << x << endl;//покажет 6
    return 0;
}
```

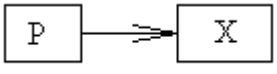
Указатели

Для объявления переменной *P* указателем на переменную базового типа *X* в языке C/C++ необходимо описание.

```
ИмяБазовогоТипа X;

ИмяБазовогоТипа *P;
```

Для того чтобы *P* стало указывать на *X*, надо в *P* записать адрес, по которому расположен *X* в памяти.




```
P = &X;
```

Для выделения места в памяти во время работы программы в C++ используется оператор *new*,

```
ИмяБазовогоТипа *T = new ИмяБазовогоТипа;
```

Если операция выделения памяти не может быть выполнена, то оператор *new* генерирует исключение типа *xalloc.* Если программа не перехватит это исключение, тогда она будет снята с выполнения.

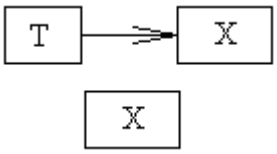
Для выделения места в памяти во время работы программы в С функция
void malloc(size_t size)*

ИмяБазовогоТипа T =(ИмяБазовогоТипа*)malloc(sizeof(ИмяБазовогоТипа));*

Эта функция выделяет в памяти блок, размер которого зависит от базового типа, и адрес первого байта этого блока записывает в *T*. Если памяти недостаточно, чтобы удовлетворить запрос, функция *malloc()* возвращает нулевой указатель - *NULL.*

Есть ряд преимуществ использования *new* перед использованием *malloc().* Во-первых, оператор *new* автоматически вычисляет размер необходимой памяти. Нет необходимости в использовании оператора *sizeof().* Более важно то, что он предотвращает случайное выделение неправильного количества памяти. Во-вторых, оператор *new* автоматически возвращает указатель требуемого типа, так что нет необходимости в использовании оператора преобразования типа. В-третьих, имеется возможность инициализации объекта при использовании оператора new.

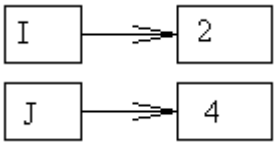
Присваивание **T = X*; скопирует значение, хранящееся в *X,* в ячейку памяти, на которую ссылается *T*, и при изменении *X* это значение меняться не будет. (**T* обозначает операцию разыменования - получить значение, хранящееся в ячейке памяти по адресу)



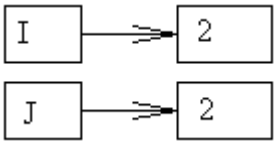
Рассмотрим пример: пусть у нас есть две переменные – указатели на целое число:

*int *I, *J;*

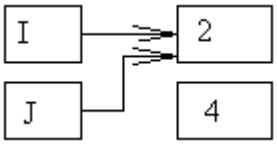
I указатель на ячейку памяти, в которой записано число *2,*
J указатель на ячейку памяти, в которой записано число *4.*
схематически это можно изобразить так:



После присваивания **J = *I* в ячейку памяти, на которую ссылалось *J*, запишется информация из ячейки памяти, на которую ссылалась переменная *I*.



А после присваивания *J = I* в *J* запишется адрес ячейки памяти, который хранится в *I*, таким образом *I* и *J* будут указывать на одну и ту же область памяти.



При выводе значения, записанного по адресу *J (cout<<*J;*) в обоих случаях будет напечатано число *2.*

После присваивания **I = 5*, в первом случае значение **J* останется равным *2,* а во втором случае станет равным *5.*

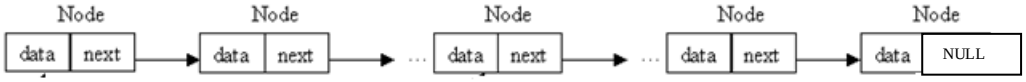
Если на ячейку, в которой хранится *4,* больше нет ссылок, то эта ячейка потеряна – так называемый «мусор» в памяти.

При активной работе с памятью «мусор» накапливается и от него надо избавляться. Для удаления «мусора» в С++ используется оператор *delete P;* или функция *free(P)* в С, где *P* – указатель. Эта функция удаляет из памяти ячейку, на которую указывает *P*, значения *P* и **P* становятся не определенными.

II. Простые списки.

Формальное определение: *списком называется сложная структура, которая помимо самих данных, хранит информацию о доступе к одному или нескольким элементам списка.*

Соответственно, простой список или *цепь является набором структур, в каждой из которых, помимо данных хранится ссылка на следующий элемент простого списка.*



где элемент списка `Node` – структура, состоящая из поля `data` – данные, которые хранятся в списке, и поля `next` указателя на следующий элемент списка.

Для создания списка используют тип данных структура**,** тип поля `data` может быть любым допустимым в языке, мы это поле будем использовать для хранения целых чисел, поэтому определим его тип как `int`.

Описание такой структуры имеет некоторые отличия в языке С и С++

В языке С:

```
struct node {
    int data;
    struct node *next;
  }
```

`struct node *X;` - указатель на структуру

Где `node` – тег(маркер, имя) структуры, а `struct node` – имя типа данных.

Можно объявить тип данных `list`:

```
typedef struct node {
    int data;
    struct node *next;
  } list;
list *tmp=(list*)malloc(sizeof(list));
free(tmp);
```

или ввести отдельный тип данных `link` – указатель на структуру:

```
typedef struct node *link;
typedef struct node {
    int data;
    link next;
  } list;
link tmp = (link)malloc(sizeof(list));
```

В языке С++:

```
struct list {
    int data;
    list *next;
  };
```

где `list` – имя типа данных

```
list *tmp = new list;
delete tmp;
```

или ввести отдельный тип данных `link` – указатель на структуру:

```
typedef struct list *link;
```

```
struct list{
    int data;
    link next;
  };
link tmp = new list;
```

В дальнейшем будем считать, что у нас определен тип данных *list*

```
list *head, *p,*z;
```

Указатель *head* ссылается на первый элемент списка, зная его можно получить доступ к *n*-ому элементу списка *p*. И *z* - указатель на последний элемент списка, поле *next* этого элемента равно *NULL* – это и служит признаком конца списка.

При таком описании указателей *head, p, z* важно понимать, в каждом из них хранится адрес какого-то элемента списка, что бы получить сам элемент надо применить операцию разыменования. Другими словами:

head – адрес первого элемента списка,
**head* – сам первый элемент – структура с двумя полями
*(*head).data* – данные, которые записаны в первом элементе
*(*head).next* – адрес второго элемента списка
*(*p).next* – адрес элемента списка следующего за *p*
*(*z).next == NULL* – пустая ссылка, элементов в списке больше нет.

Скобки в *(*head).data* необходимы, поскольку приоритет оператора **.** выше, чем приоритет ***. Выражение **head.data* будет проинтерпретировано как **(head.data)* что неверно, поскольку *head.data* не является указателем.

Указатели на структуры используются весьма часто, поэтому для доступа к ее элементам была придумана еще одна, более короткая форма записи. Если *head* — указатель на структуру, то *head->data* ее отдельный элемент. (Оператор *->* состоит из знака «*–*», за которым сразу следует знак «*>*».)

Приведенные выше записи полностью эквивалентны следующим:
head->data – данные, которые записаны в первом элементе
head->next – адрес второго элемента списка
p->next – адрес элемента списка следующего за *p*
z->next == NULL – пустая ссылка, элементов в списке больше нет.

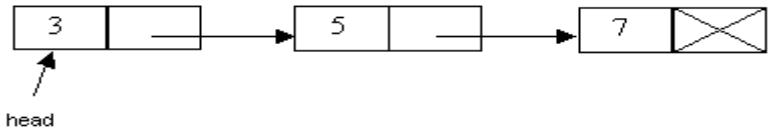
Операторы **.** и *->* выполняются слева направо, и могут использоваться в любой комбинации:

```
(*head).next->data
(*head->next).data
head->next->data
(*(*head).next).data
```

это эквивалентные обращения к полю *data* элемента, следующего за элементом, на который указывает *head*

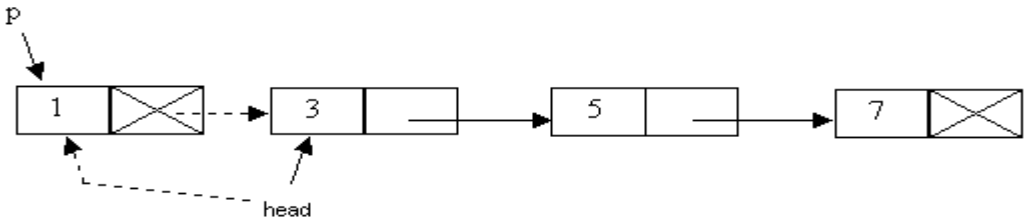
Рассмотрим основные приемы работы со списками.

Пусть мы уже создали некий список, и *head* - указатель на первый элемент этого списка.



При добавлении нового элемента в начало списка, необходимо выделить место в памяти под этот элемент, записать данные в поле *data*, и установить необходимые ссылки.

Оформим это в виде функции *push(list **ptr; int add);* где первый параметр *ptr* – указатель на первый элемент списка, а второй параметр *add* – данные которые добавляем, т.е. вызов *push(head, 1);* добавит *1* в начало списка *head*.




```
void push (list **ptr, int add){
    list *tmp =(list*)malloc(sizeof(list));
    if (tmp) exit();            //если место не выделяется, то выход
    (*tmp).data = add;
    tmp->next=*ptr;
    *ptr = tmp;
}
```

Стоит обратить внимание на способ передать указатель на начало списка в функцию. Так как в C/C++ все параметры передаются по значению, то фактически функция работает с копией параметра.

Если функцию определить так: *void push(list *ptr, int add)*, то при добавлении элемента в начало списка адрес нового элемента будет записан в копию *head*, и при выходе из функции будет недоступен

При таком определении функции *void push(list **ptr, int add)* в качестве параметра передается указатель на указатель *head*, строится копия указателя на указатель. В функции *head* становится указателем на новый элемент, и при выходе из функции адрес нового элемента не будет потерян. Общими словами можно дать такую рекомндацию: если есть вероятность, что в функции поменяется первый элемент списка (сам элемент, не поле *data*) передавать в функцию список в качестве параметра стоит через указатель на указатель на первый элемент списка.

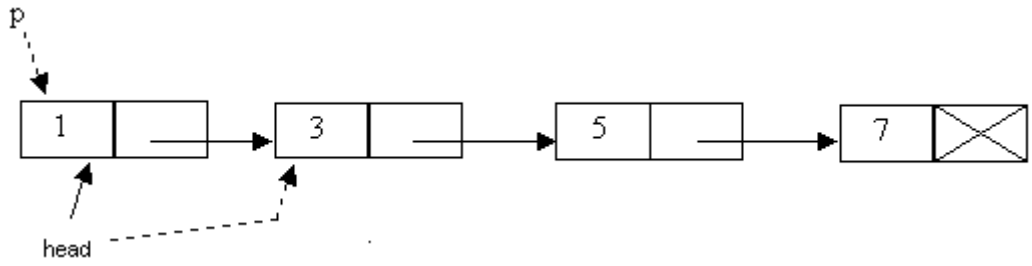
В C++ чаще используются ссылки для передачи параметра в функцию

```
void push(list* &ptr, int x){
    list *tmp = new list;
    tmp->data =x;
    tmp->next=ptr;
    ptr = tmp;
} ;
```

очевидно, что если списка нет, т.е. `head = NULL` эта функция тоже будет работать корректно, в поле `next` запишется `NULL` и при дальнейшем добавлении элементов в начало списка, этот элемент станет последним, и как положено не будет ни куда ссылаться.

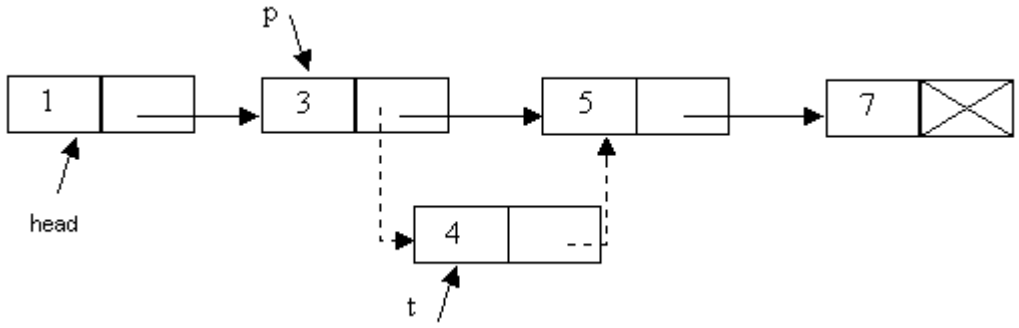
В дальнейшем будем придерживаться только синтаксиса C++

Функция `int pop(list* &ptr)` будет удаляет первый элемент списка и в качестве результата возвращает значение хранящееся в его поле `data`.




```
int pop(list* &ptr){
    list *tmp = ptr;
    int x = ptr->data;
    ptr = ptr->next;
    delete tmp;              // удаляем «мусор» из памяти
    return x;
}
```

Рассмотрим далее удаление и добавление элемента в середину списка, «до» и «после» заданного элемента, для простоты будем считать, что такой заданный элемент в списке есть, а при удалении «до» или «после» заданный элемент не является первым или последним элементом списка, т.е. существует элемент, который будем удалять «до» или «после».



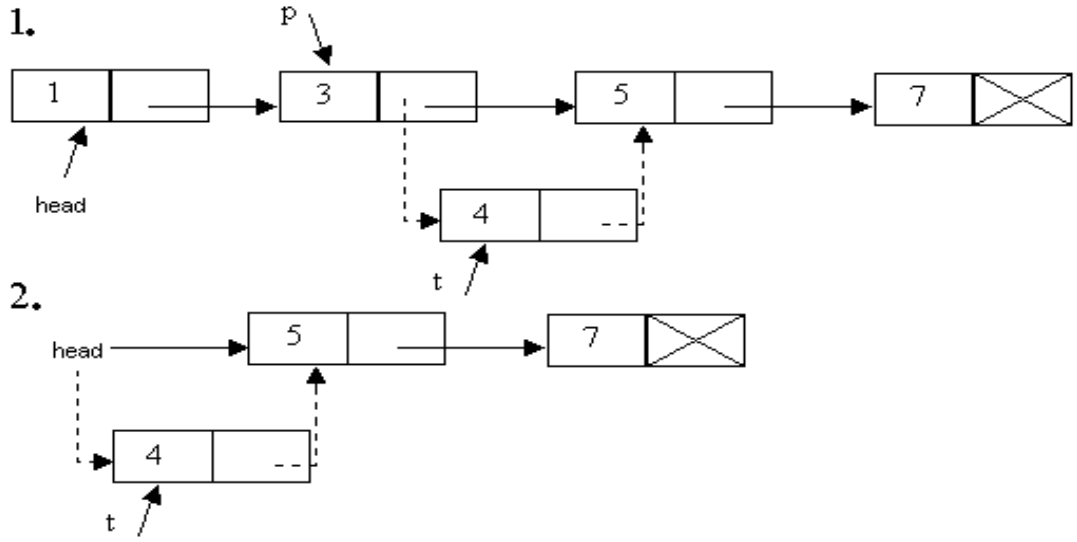
При вставке элемента с полем `data 4` после элемента с полем `data 3`, необходимо выделить место в памяти, в поле `data` записать `4`, найти элемент, в поле `data` которого записано `3`, и, установить новые ссылки.

```
void addafter(list *ptr, int after, int x){
    list *tmp = ptr;
    while(tmp && tmp->data != after) tmp = tmp->next;
    if (tmp){
        list *p = new list;
        p->data = x;
        p->next = tmp->next;
        tmp->next = p;
    }
}
```

Удаление элемента после данного, опишем как функцию, которая возвращает в качестве результата значение поля *data* удаляемого элемента.

```
int delafter(list *ptr, int a)
{
    int x;
    while(ptr && ptr-> data!= a) ptr = ptr->next;
    if (ptr)
    {
        list *tmp = ptr->next;
        x = tmp-> data;
        ptr->next = tmp->next;
        delete tmp;
    }
    return x;
}
```

Аналогично добавляется и удаляется элемент «до» заданного элемента, единственное изменение нужно внести в условие цикла движения по списку.

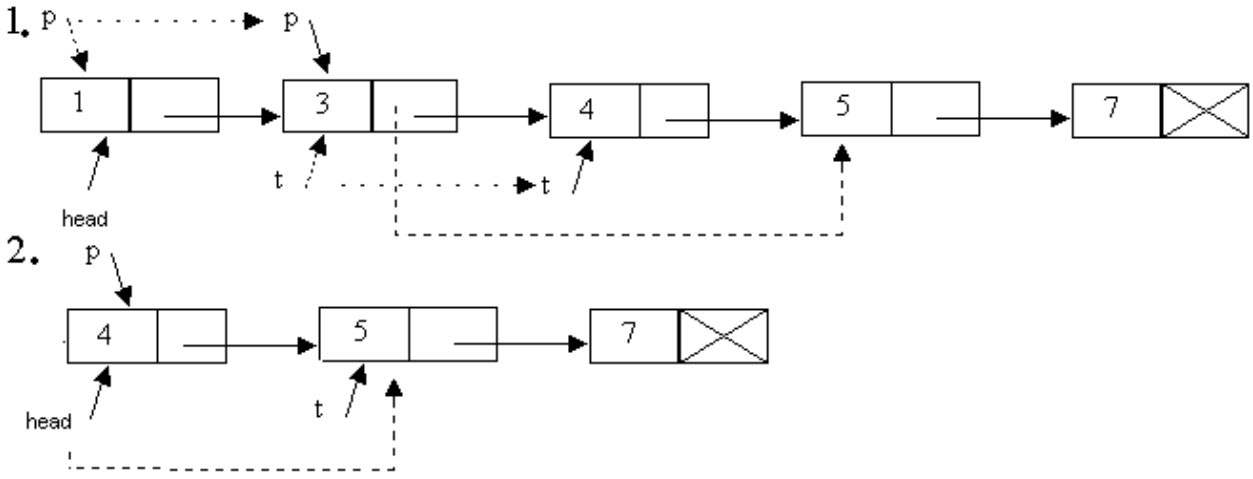


В первом случае, очевидно, что, если мы хотим добавить элемент перед элементом с полем *data 5*, необходим указатель на элемент с полем *data 3*. Но надо рассматреть и другой случай, когда элемент, перед которым мы хотим вставить новый элемент, сам является первым.

```
void addbefore(list* &ptr, int before, int x)
{
    list *tmp = ptr, *p;
    p = new list;
    p-> data = x;
    if(tmp-> data == before)
    {
        p->next = tmp;
        ptr = p;
    }
    else
    {
        while(tmp->next && tmp->next->data != before)
            tmp = tmp->next;
        p->next = tmp->next;
```

```
            tmp->next = p;
        }
    }
```

При удалении элемента «до» пришлось бы заглядывать на два элемента вперед для поиска заданного элемента *p->next->next->data*. Удобнее использовать пару указателей: один указатель *p* на первый элемент, а второй указатель *t* на следующий за ним. Остается рассмотреть два случая, если *t* указатель на заданный элемент, то элемент, на который ссылается *p*, удаляем, или двигаем по списку сразу оба указателя, пока следующим за элементом, на который ссылается *t*, не будет заданным, т.о. заглядываем только на один элемент вперед.




```
int delbefore(list* &ptr, int before)
{
    int x;
    list *p = ptr;
    list *t = p->next;
    if(t-> data == before)      //заданный элемент второй в списке
    {
        x = p-> data;
        delete p;
        ptr = t;
    }
    else
    {
        while(t->next && t->next->data != before)
        {
            p=t;                //двигаем сразу два указателя по списку
            t=t->next;
        }
        x = t->data;
        p->next = t->next;
        delete t;
    }
    return x;
}
```

При добавлении и удалении элемента в конец списка можно двигаться до конца списка, т.е. пока поле *next* не станет *NULL*, но мы можем представить себе список состоящим из «головы» и «хвоста» и использовать рекурсию.

```
void pushtail(list* &ptr,int x)
{
    if (ptr == NULL)
    {
        ptr = new list;
        ptr->data = x;
        ptr->next = NULL;
    }
    else pushtail(ptr->next, x);
}
 int poptail(list* &ptr)
{
    int x;
    if (!ptr->next)
    {
        x = ptr->data;
        delete ptr;
        ptr = NULL;
        return x;
    }
    else poptail(ptr->next);
}
```

При переборе элементов списка можно использовать как рекурсию, так и движение по списку в цикле. В качестве примера функция вывода элементов списка на консоль.

рекурсивный вариант:

```
void print_list_rec(list *ptr)
{
    if (ptr)
    {
        cout<<ptr->data<<" - > ";
        print_list_rec(ptr->next);
    }
}
```

не рекурсивный вариант:

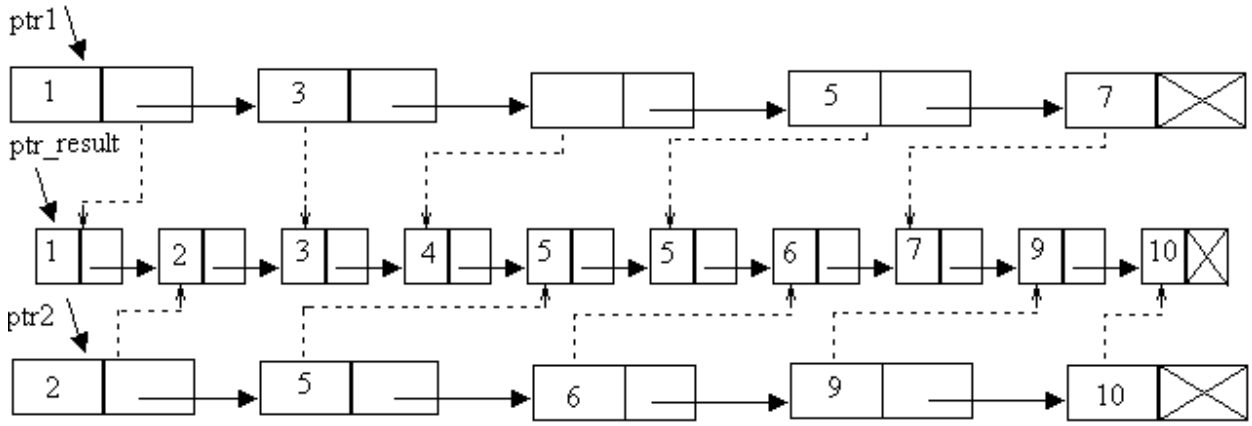
```
void print_list(list *ptr)
{
    while(ptr)
    {
        cout << ptr->data << " - > ";
        ptr = ptr->next;
    }
}
```

В заключении рассмотрим пару задач:

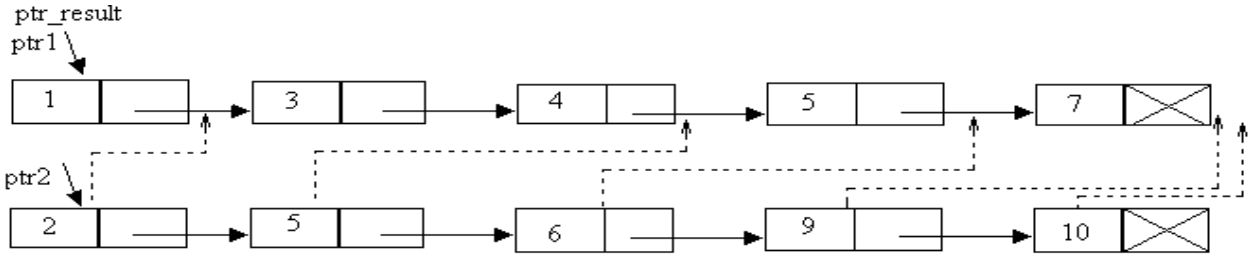
Пусть даны два упорядоченных по возрастанию списка. Объединить их в один упорядоченный по возрастанию список.

Эту задачу можно решать разными способами:

- строить третий список, добавляя в него элементы в порядке возрастания из двух данных списков.

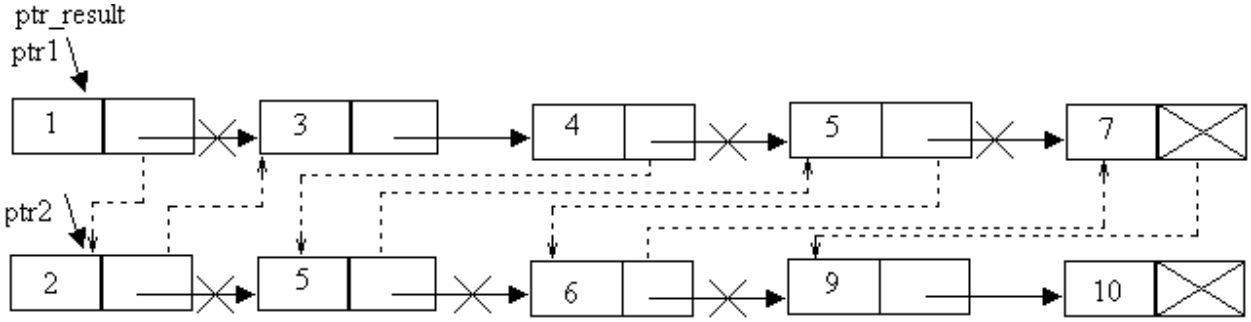


- в один из списков вставлять элементы второго списка, сохраняя упорядоченность



- переопределять ссылки между элементами первого и второго списка, что бы в результате получился упорядоченный список.

Первые два способа легко реализуются с использованием функций, описанных



нами ранее. Очевидно, что третий способ наиболее интересный с точки зрения работы с указателями, его мы и рассмотрим.

Сначала определим «голова» какого списка меньше, этот элемент и будет «головой» списка, который мы будем строить. В нашем случае, указатель *ptr_result* будет указателем на элемент с полем *data 1*, указатель *ptr1* сдвигаем на элемент с полем *data 3*.

Для того, что бы не искать каждый раз последний добавленный элемент, удобно ввести еще один указатель *p*, он будет указывать на последний присоединенный к списку *ptr_result* элемент. *{1}*.

Далее рассматриваем два случая. Пока оба списка не пусты, в зависимости от значений, которые записаны в поле *data* текущих элементов (на них установлены указатели *ptr1* и *ptr2)* добавляем один из элементов к *p*, и сдвигаем соответствующий указатель *{2}.*

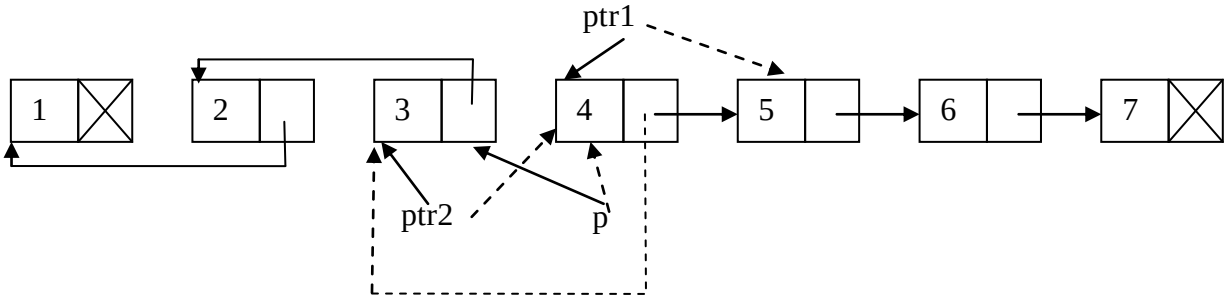
И другой случай, когда один из списков закончился, т.е. соответствующий указатель станет *NULL*, к *p* добавляем хвост второго списка *{3}.*

```
list* merge(list *ptr1, list *ptr2)
{
    list *p;
    if (ptr1)
    {
        if(ptr2)
        {
            if(ptr1->data <= ptr2->data)
            {
                p = ptr1;
                ptr1 = ptr1->next;
            }
            else
            {
                p = ptr2;
                ptr2 = ptr2->next;
            }
        }
        else p = ptr1;
    }
    else p = ptr2;
    list *ptr_rezult = p;                    //1
    while(ptr1 && ptr2)
    {
        if(ptr1->data <= ptr2->data)         //2
        {
            p->next = ptr1;
            p = ptr1;
            ptr1 = ptr1->next;
        }
        else
        {
            p->next = ptr2;
            p = ptr2;
            ptr2 = ptr2->next;
        }
    }
    if(ptr1)                                 //3
        p->next = ptr1;
    else
        p->next = ptr2;
    return ptr_rezult;
}
```

Вторая задача: дан список, перевернуть его.

Допустим, что у нас уже написана функция, которая умеет переворачивать список. В какой-то момент времени *ptr2* указывает на уже перевернутую часть списка, *ptr1* на оставшуюся часть исходного списка.

Вспомогательный указатель *p* устанавливаем на элемент, на который указывает *ptr1*, *ptr1* сдвигаем на следующий элемент, организовываем указатель с элемента, на который указывает *p* на элемент, на который указывает *ptr2*, *ptr2* сдвигаем на следующий элемент. Таким образом в перевернутой части списка получили на один элемент больше.




```
void reverse(list *&ptr1)
{
    list *ptr2 = ptr1;
    ptr1 = ptr1->next;
    ptr2->next = NULL;              //в перевернутом списке один элемент
    while(ptr1)
    {
        list *p = ptr1;
        ptr1 = ptr1->next;
        p->next = ptr2;
        ptr2 = p;
    }
    ptr1 = ptr2;
}
```

III. Задачи для самостоятельного решения

1. В список L вставить элемент E1 за каждым вхождением элемента E.
2. В список L , элементы которого упорядочены по возрастанию, вставить новый элемент E, так чтобы сохранить упорядоченность.
3. Из списка L, удалить один элемент за каждым вхождением элемента E, если такой есть и он отличен от E.
4. Удалить из списка L все отрицательные элементы.
5. Проверить есть ли в списке L хотя бы два одинаковых элемента.
6. В списке L из каждой группы подряд идущих одинаковых элементов оставить только один.
7. Сформировать список L из элементов, которые входят одновременно в списки L1 и L2.
8. Сформировать список L из элементов, которые входят список L1, но не входят в список L2.
9. Сформировать список L из элементов, которые входят в один из списков L1 или L2, но не входят во второй.
10. Вычислить сумму элементов списка, которые стоят на четных местах.
11. По списку L построить два списка: L1 - из положительных элементов и L2 – из отрицательных.
12. Найти максимальный элемент списка и вывести его порядковый номер, считая от начала списка.
13. Добавить в конец списка L1 максимальный четный элемент списка L2.
14. Подсчитать количество вхождений элемента E в список L.
15. Найти сумму таких элементов списка L, значения которых меньше значений элементов непосредственно следующих за ними.
16. В списке L упорядочить по возрастанию только положительные элементы.
17. В списке L упорядочить по возрастанию только элементы с четными порядковыми номерами (считая от начала списка)
18. В списке L упорядочить по возрастанию только элементы с порядковыми номерами, являющимися простыми числами (считая от начала списка)

IV. Литература

1. Керниган , Б.У. Язык программирования С/Брайан У. Керниган, Деннис М. Ритчи – 2 издание - “Вильямс”, Москва-Петербург-Киев, 2017. -288 с.
2. Кнут Д.Э. Искусство программирования, тт.1-3. “Диалектика-Вильямс ”, Москва-Петербург-Киев, 2005 г.
3. Вирт Н. Алгоритмы и структуры данных. (Новая версия для Оберона), ДМК Пресс,2014 г