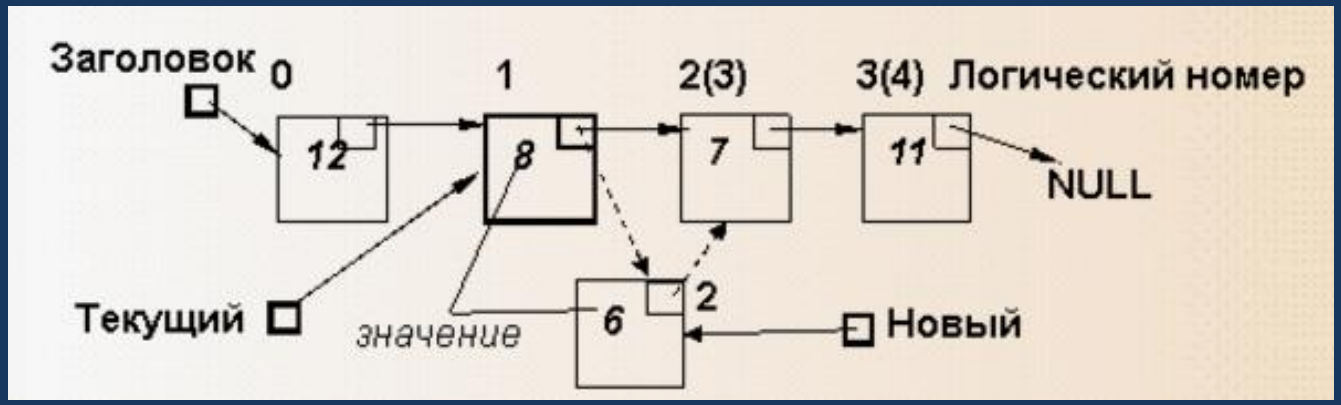
Списки

Список – СД, каждый элемент содержит указатели на соседей (next – следующий, prev - предыдущий)



Заголовок – указатель на первый элемент, указатели на первый (head) и последний (tail)

Изменение порядка – переустановка указателей на соседей (элементы остаются «на месте»)

Локальность изменений: при вставке/удалении элемента изменения касаются только текущего и его соседей.

Логический номер – порядковый номер в последовательности обхода списка

Последовательная СД – доступ к k-му элементу требует k шагов

Виды списков

Динамические и статические:

- **Статические** – элементы – статические переменные (например, элементы массива), связи – инициализируются адресами
- **Динамические** – элементы списка создаются как динамическизависимости

По виду связей:

- **односвязные** - каждый элемент списка имеет указатель на следующий
  - Самый простой, «одностороннее движение»
  - Моделирование стека и очереди (указатели head/tail)
  - Буферный пул
- **двусвязные** - каждый элемент списка имеет указатель на следующий и на предыдущий элементы
- **циклические** - первый и последний элементы списка ссылаются друг на друга и цепочка представляет собой кольцо
  - Заголовок – одни указатель (последний = предыдущий для первого)
  - Моделирование циклических цепочек (круговая диспетчеризация заданий в ОС), обычных линейных СД

Трудоемкость

Последовательность на списке – трудоемкость операций:

- Append - O(1), O(N)
- Insert - **O(1)** – для текущей позиции
- Remove - **O(1)**
- Get - **O(N)** – **последовательный доступ**
- Search - O(N) – не зависит от порядка

Выводы:

- Прямая противоположность массиву
- Списки используются там, где операций смены порядка больше, чем извлечения (get)

Технология

Элемент списка - struct:

- Указатели на соседей
- Данные

```
struct   elem                  // определение структурированного типа
{        int    value;         // значение элемента (хранимые данные)
         elem *next;           // единственный указатель или
         elem *next,*prev;     // два указателя или
         elem *links[10];      // ограниченное количество указателей (не больше 10) или
         elem **plinks;        // произвольное количество указателей (внешний МУ)
         };
```

Описание структуры элемента, а не списка. Список создается динамически или описан статически

```
struct list { int val; list *next; } a={0,NULL}, b={1,&a}, c={2,&b}, *ph = &c;
```

```
struct list2 { int val; list *next,*prev;};
extern list2 a,b,c;                          // предварительное объявление элементов списка
list2 a={0,&b,NULL}, b={1,&c,&a}, c={2,NULL&b}, *ph = &c; // выделены «ссылки вперед»
```

Технология. Аналогия с массивом

Указатель – движок со смысловой интерпретацией: первый, текущий, предыдущий, новый

	Список	Массив
Определение	struct list {int val; list *next, *prev; };	int A[100]; int n;
Пустой список	list *ph=NULL;	n=0;
Первый	list *p; p=ph;	Int i=0;
Следующий	p->next	i+1
Предыдущий	p->prev	i-1
К следующему	p=p->next	i++
К предыдущему	p=p->prev	i--
Просмотр	for (p=ph; p!=NULL; p=p->next) …p->val…	for (i=0; i<n; i++) …A[i]…
Проверка на последний	p->next ==NULL	i==n-1
К последнему	for (p=ph; p->next!=NULL; p=p->next);	i=n-1
Новый	list *q = new list; q->val = v;	int v;

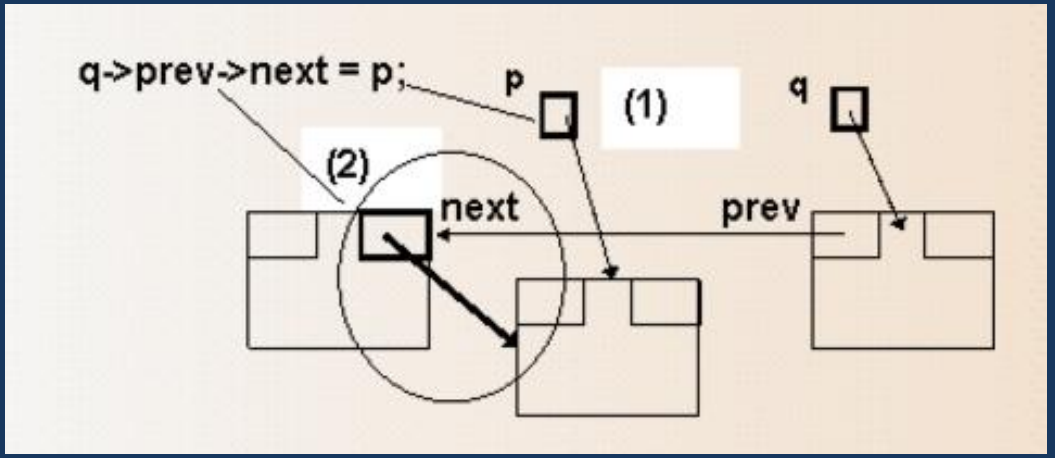
Включить последним	for (p=ph; p->next!=NULL;
	p=p->next);
	q->next=NULL; p->next=q;
		A[n++]=v;

Включить первым	q->next=ph; ph=q;	for (i=n; i>0; i--) A[i]=A[i-1]; A[0]=v;

Технология. Изменение порядка

«Перебрасывание» указателей, чтобы связность изменилась требуемым образом = операция присваивания указателей. Способы интерпретации:

- Графическая
- Адресная
- Смысловая (сематическая)



Графическая интерпретация: q->prev->next = p;

- Указатели на доступные элементы (q – текущий, p – новый)
- Правая часть – указатель на элемент, на который будут ссылаться
- Левая часть – указатель (поле списка), в который будет записана ссылка на элемент левой части

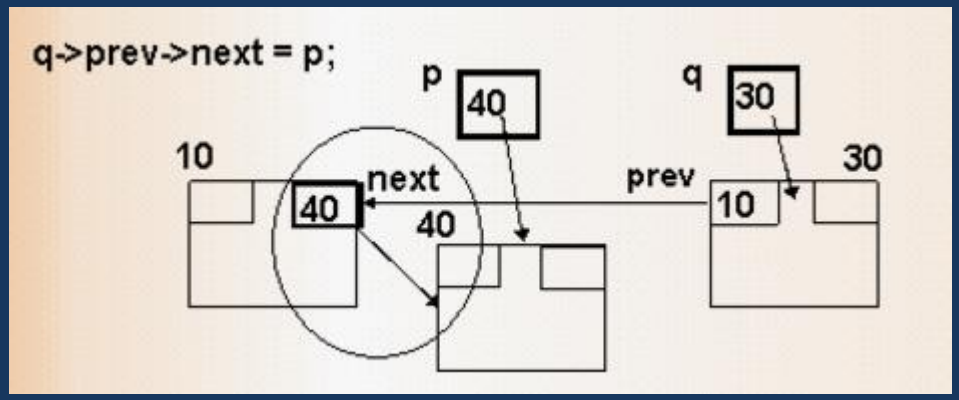
Смысловая интерпретация: q->prev->next = p;

- q – указатель на текущий, относительно которого изменения
- q->prev – указатель на предыдущий
- q->prev->next – левая часть – ссылка на следующий в предыдущем
- q->prev->next = p – следующий для предыдущего = новый

Технология. Изменение порядка

Адресная интерпретация: q->prev->next = p;

Элементам даются условные числовые адреса, присваивание указателей = присваивание адресов



Технология. Передача списка в функцию

- list *F(list *in) – ФП – заголовок, результат – новый заголовок
- void F(list **in) – ФП – указатель на заголовок (указатель на указатель)
- void F(list *&in) – ФП – ссылка на указатель (ссылка на заголовок)

```
//-------------------------------------------------------63-02.cpp
//---- включение в начало списка с изменением заголовка
                        // Вариант 1. Измененный указатель возвращается
list *Ins1(list *ph, int v)
{ list *q=new list;
q->val=v; q->next=ph; ph=q;
return ph; }
//-------------------------------------------------------------------
                       // Вариант 2. Используется указатель на заголовок
void Ins2(list **pp, int v)
{ list *q=new list;
q->val=v; q->next=*pp; *pp=q; }
//-------------------------------------------------------------------
                         // Вариант 3. Используется ссылка на указатель
void Ins3(list *&pp, int v)
{ list *q=new list;
q->val=v; q->next=pp; pp=q; }
//----------- Пример вызова----------------------------------------
void main(){
list *ph=NULL;                                        // пустой список
ph=Ins1(ph,5);                                  // сохранить новый заголовок
Ins2(&ph,66);                                  // передается адрес заголовка
Ins3(ph,7);                                 // передается ссылка на заголовок
show(ph);}
```

Односвязный список

Проблема: доступ к предыдущему элементу

```
//-----------------------------------------------------63-04.cpp
//--- Включение в односвязный с сохранением порядка
// pr указатель на предыдущий элемент списка
 void InsSort(list *&ph, int v)
 { list *q ,*pr,*p;          // перед переходом к следующему указатель на текущий
 q=new list; q->val=v;       // запоминается как указатель на предыдущий
 for ( p=ph,pr=NULL; p!=NULL && v>p->val; pr=p,p=p->next);
 if (pr==NULL)               // включение перед первым
      { q->next=ph; ph=q; }
 else                        // иначе после предыдущего
      { q->next=p;           // следующий для нового = текущий
      pr->next=q; }}         // следующий для предыдущего = новый
```

```
//-----------------------------------------------------63-05.cpp
//--- Сортировка односвязного списка вставками
 list *sort(list *ph)              // функция возвращает заголовок нового списка
 { list *q, *out, *p , *pr;
 out = NULL;                       // выходной список пуст
 while (ph !=NULL)                 // пока не пуст входной список
      { q = ph; ph = ph->next;     // исключить очередной
                                   // поиск места включения
      for ( p=out,pr=NULL; p!=NULL && q->val>p->val; pr=p,p=p->next);
      if (pr==NULL)                // включение перед первым
           { q->next=out; out=q; }
      else                         // иначе после предыдущего
           { q->next=p; pr->next=q; }
      }
 return out; }
```

Двусвязный список

```
//-------------------------------------------------------------63-07
//--------Включение в двусвязный список с сохранением порядка
void InsSort(list *&ph, int v)      // ссылка на заголовок
{ list *q , *p=new list;            // новый элемент списка
p->val = v;
p->prev = p->next = NULL;
     if (ph == NULL) {              // включение в пустой список
     ph=p; return ;
     }                              // поиск места включения - q
for (q=ph; q !=NULL && v > q->val; q=q->next);
if (q == NULL)                      // включение в конец списка
     {                              // восстановить указатель на последний
     for (q=ph; q->next!=NULL; q=q->next);
     p->prev = q;
     q->next = p;
     return;
     }                              // включить перед текущим
p->next=q;                          // следующий за новым = текущий
p->prev=q->prev;                    // предыдущий нового = предыдущий текущего
if (q->prev == NULL)                // включение в начало списка
   ph = p;
else                                // включение в середину
   q->prev->next = p;               // следующий за предыдущим = новый
q->prev=p;                          // предыдущий текущего = новый
}
```

Проблема концов списка

При манипуляциях над элементом и списком – крайние ситуации:

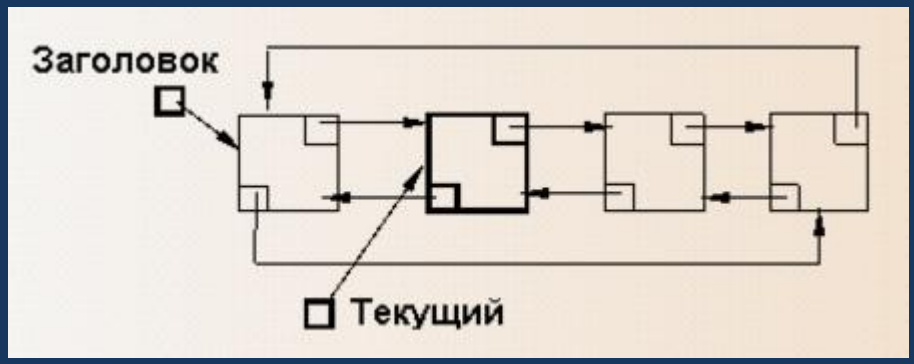
- Список пустой
- Элемент единственный
- Элемент первый
- Элемент последний
- Элемент в середине

Существующий код:

- Учитывает все ситуации по умолчанию
- Пропускает одну из них = ошибка **крайней ситуации (граничный дефект)**

Циклический список

```
struct list{
    int val;
    list *next,*prev;
};
void show(list *p) {
list *q=p; do{
    printf("%d ",p->val);
    p=p->next;
    } while(p!=q);
}
```




```
//-----------------------------------------------------63-08.cpp
//--- Очередь на циклическом списке
void In(list *&ph, int v){
 list *q = new list;                  // Новый элемент как единственный
 q->val = v; q->next = q->prev = q;
 if (ph == NULL) { ph=q; return; }    // Список пуст - единственный
 q->next = ph;                        // следующий за новым = первый
 q->prev = ph->prev;                  // предыдущий для нового = последний
 ph->prev->next = q;                  // следующий для последнего = новый
 ph->prev = q; }                      // предыдущий для первого = новый

int Out(list *&ph){
  if (ph==NULL) return -1;
  int v=ph->val;
  list *q=ph;                         // Запомнить текущий
  ph=ph->next;                        // Перейти на следующий
  if (ph->next==ph) ph=NULL;          // Единственный стал пустой
  q->next->prev=q->prev;              // Элемент сам себя "выкусывает"
  q->prev->next=q->next;
  delete q;
  return v;
}
```

Циклический список

```
//-----------------------------------------------------63-09.cpp
//--- Включение в циклический список с сохранением порядка
 list *InsSort(list *ph, int v)      // Функция возвращает новый заголовок
 { list *q = new list;               // Новый элемент как единственный
 q->val = v; q->next = q->prev = q;
 if (ph == NULL) return q;           // Список пуст вернуть новый
 list *p = ph;
      do { if (v < p->val) break;    // Место вставки перед первым,
      p=p->next;                     // большим заданного, иначе -
      } while (p!=ph);               // перед первым в списке (после последнего)
 q->next = p;                        // следующий за новым = текущий
 q->prev = p->prev;                  // предыдущий для нового = предыдущий текущего
 p->prev->next = q;                  // следующий для предыдущего = новый
 p->prev = q;                        // предыдущий для текущего = новый
 if ( ph->val > v) ph=q;             // включение перед первым -
 return ph; }                        // коррекция заголовка
```

Сортировка списка рекурсивным разделением

указатели на первый и последний - **list *p[2]**

```
struct list{
    int val;
    list *next;
    };
void show(list *pp[]) {
    for (list *p=pp[0]; p!=NULL; p=p->next) printf("%d ",p->val);
    puts("");
    }

void add(list *pp[], list *q){
    q->next = NULL;
    if (pp[0]==NULL)
        pp[0]=pp[1]=q;
    else{                       // Вставка последним
        pp[1]->next = q;        // Новый вслед за последним
        pp[1]=q;                // Послений = новый
        }
    }
```

```
list **create(int n){
    list *q;
    list **pp = new list*[2];
    pp[0]=NULL;
    pp[1]=NULL;
    while(n--!=0){
        q=new list;
        q->next=NULL;
        q->val=rand()%100;
        add(pp,q);
        }
    return pp;
    }
```

Сортировка списка рекурсивным разделением

```
//-----------------------------------------------------72-03.cpp
//  сортировка односвязного списка рекурсивным разделением
void sort(list **pp){
    list *m,*p,**p1,**p2,*q;
    if (pp[0]==NULL || pp[0]->next==NULL)
        return;                          // не более одного - конец разделения
    p = pp[0];
    m=p; p=p->next;                      // m-медиана - первый элемент
    m->next = NULL;
    p1 = new list*[2];
    p1[0] = p1[1] = NULL;
    p2 = new list*[2];
    p2[0] = p2[1] = NULL;                // p1,p2 - разделенные списки
    while(p!=NULL){
        q=p; p=p->next;                  // извлечь очередной из входного
        if (q->val < m->val)
            add(p1,q);
        else
            add(p2,q);
            }
    sort(p1);                            // рекурсивная сортировка частей
    sort(p2);
    m->next=p2[0];                       // "склеить" медиану и списки
    if (p1[0]!=NULL)
        p1[1]->next = m;
    pp[0] = (p1[0]==NULL ? m : p1[0]);   // первый - пустой = первая - медиана
    pp[1] = (p2[0]==NULL ? m : p2[1]);   // второй - пустой = последняя - медиана
    }
```

Трассировка. Лучшие отладчик - printf

```
33 43 62 29 0 8 52 56 56 19 11 51
m=33
s1= 29 0 8 19 11
s2= 43 62 52 56 56 51
m=29
s1= 0 8 19 11
s2=
m=0
s1=
s2= 8 19 11
m=8
s1=
s2= 19 11
m=19
s1= 11
s2=
s1= 11
s2=
s1=
s2= 11 19
s1=
s2= 8 11 19
s1= 0 8 11 19
s2=
m=43
s1=
```

```
m=43
s1=
s2= 62 52 56 56 51
m=62
s1= 52 56 56 51
s2=
m=52
s1= 51
s2= 56 56
m=56
s1=
s2= 56
s1=
s2= 56
s1= 51
s2= 56 56
s1= 51 52 56 56
s2=
s1=
s2= 51 52 56 56 62
s1= 0 8 11 19 29
s2= 43 51 52 56 56 62
0 8 11 19 29 33 43 51 52 56 56 62
```

```
    printf("m=%d\n",m->val);
    printf("s1= "); show(p1);
    printf("s2= "); show(p2);
    sort(p1);
    sort(p2);
    printf("s1= "); show(p1);
    printf("s2= "); show(p2);
```

Отладка. Граничный дефект

Частичная ООП-нотация. Функции init, before,remove встроены в list

```
struct list{                    // Класс элемента списка
    int v;
    list *next,*prev;           //
    void init (int v0){         // ООП - нотация, list *this - текущий
        next=prev=this;
        v=v0;
        }
    void before(list *ee){      // Вставка текущего перед ee
        next=ee;                // Вставляет себя ПЕРЕД указанным
        prev=ee->prev;
        ee->prev->next=this;
        ee->prev=this;
        }
    int remove(){               // Удаление текущего из цепочки
        if (next==this)         // Если единственный, т.е. ссылается на себя
            return 1;
        prev->next=next;
        next->prev=prev;
        return 0;
        }
    };
```

Отладка. Граничный дефект

```
void toString(list *q){
    if(q == NULL)
        return;
    printf("%d ",q->v);
    for(list *p=q->next;p!=q;p=p->next)
        printf("%d ",p->v);
    for(list *p=q->prev;p!=q;p=p->prev)
        printf("%d ",p->v);
    printf("%d ",q->v);
    puts("");
    }
list *toEnd(list *head, int vv){
    list *q = new list();
    q->init(vv);
    if (head==NULL)
        head=q;
    else
        q->before(head);
    return head;
    }
list *createList(int a[],int n){
    list *head=NULL;
    for(int i=0;i<n;i++)
        head = toEnd(head,a[i]);
    return head;
    }
```

```
int main(){
    int a[]={5,3,6,2};
    list *p = createList(a,4);
    toString(p);
    p = sort(p);
    toString(p);
    return 0;
    }
```

Отладка. Граничный дефект

Структура сортировки выбором: внешний цикл – пока не останется 1 элемент, выбрать минимальный, выкусить и добавить в хвост последнего
Из списка выкусываются элементы целиком (нет new.delete)

```
list *sort(list *head){                          // Сортировка выбором
    if (head==NULL || head->next==head)          // Граничное условие = пустой или единичный
        return head;
    list *out=NULL;                              // Выходной присок пуст
    list *pmin=head,*p;                          // Ссылки на текущий и минимальынй
    while(head->next!=head){                     // Пока не остался один во входном
        //for(pmin=head,p=head->next;p->next!=head;p=p->next)
        // ПОТЕНЦИАЛЬНЫЙ ГРАНИЧНЫЙ ДЕФЕКТ 4 - сортировка без последнего
        for(pmin=head,p=head->next;p!=head;p=p->next)
            if (p->v < pmin->v) pmin=p;          // Поиск минимального во входном
        pmin->remove();                          // Выкусить минимальный из входного
        if (pmin==head)                          // Убрать заголовок с минимального
            head=head->next;                     // ПОТЕНЦИАЛЬНЫЙ ГРАНИЧНЫЙ ДЕФЕКТ 2
        if (out==NULL){                          // Вставка первого в пустой присок
            pmin->next=pmin->prev=pmin;          // ПОТЕНЦИАЛЬНЫЙ ГРАНИЧНЫЙ ДЕФЕКТ 3
            out=pmin;
            }
        else  pmin->before(out);                 // Вставка в конец выходного
        }
    head->before(out);                           // ОБНАРУЖЕННЫЙ ДЕФЕКТ 1 - pmin ВМЕСТО head
                                                 // Список с закольцованным последним элементом
    return out;                                  // Вернуть выходной
    }
```

```
5 3 6 2 2 6 3 5
2 3 5 6 6 5 3 2
```

Отладка. Граничный дефект

Потенциальные граничные дефекты и «очепятки»:

- Дефект 1: Пустой или с одним элементом – выход **(падает по NULL)**
- Дефект 3: На первом шаге выходной = пустой **(падает по NULL)**
- Дефект 4: Сортировка без последнего (последний пропускается при поиске минимума) **(неотсортированный выходной, если минимальный встречается в конце исходного)**

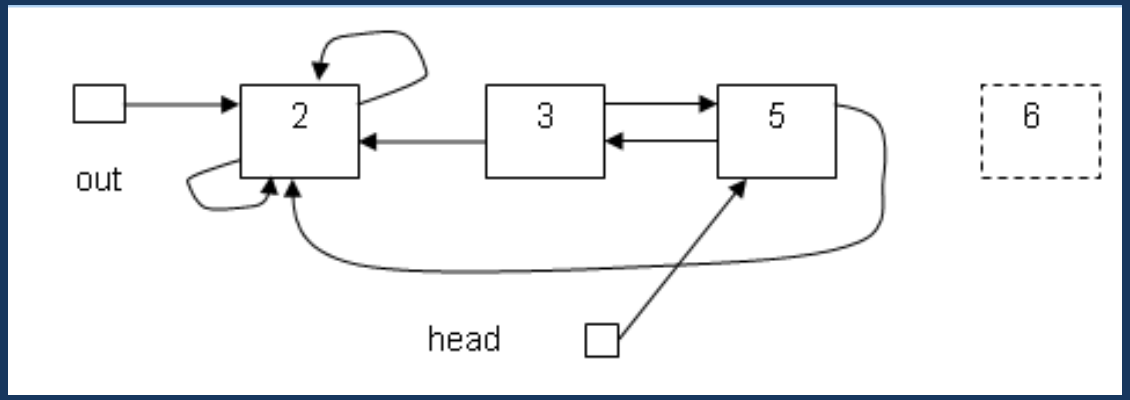
5	3	6	2	2	6	3	5
3	5	6	2	2	6	5	3

- Дефект 2: **М**инимальный = первый, при выкусывании сдвинуть заголовок.
- Обнаруженный дефект 5: «Очепятка» pmin->before(out) вместо head->before(out) из-за копипаста

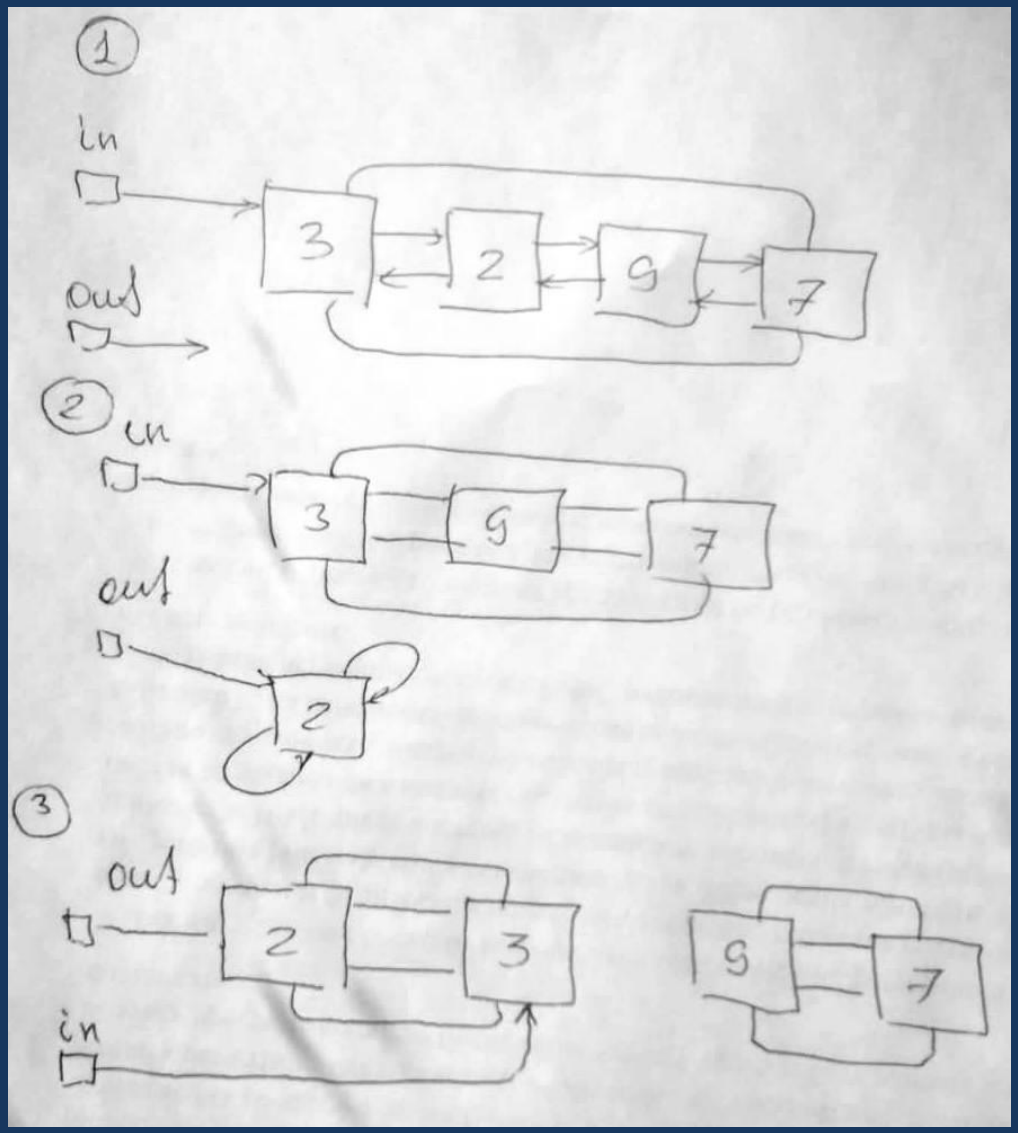
Отладка. Граничный дефект 2

Анализ дефекта:

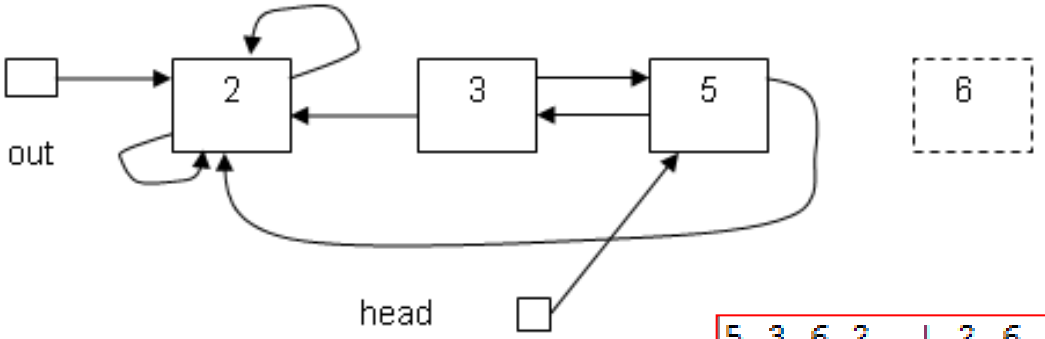
- дефект не будет проявлять себя, пока минимальный элемент не окажется в оставшемся списке первым
- при удалении первого элемента он помещается в конец выходного, но при этом **на нем остается ссылка – заголовок входного списка.** Таким образом, ссылки на входной и выходной списки ссылаются **на один и тот же список, только выходной – на первый, а входной – на последний.** Для случая циклического списка оба списка будут восприниматься **полностью идентично**
- в полученном списке будет выбран минимальный элемент (но уже не на первом месте), «выкушен» и включен в выходной список перед первым (минимальным), т.е. **перед самим собой**



Отладка. Граничный дефект 2



Отладка. Граничный дефект

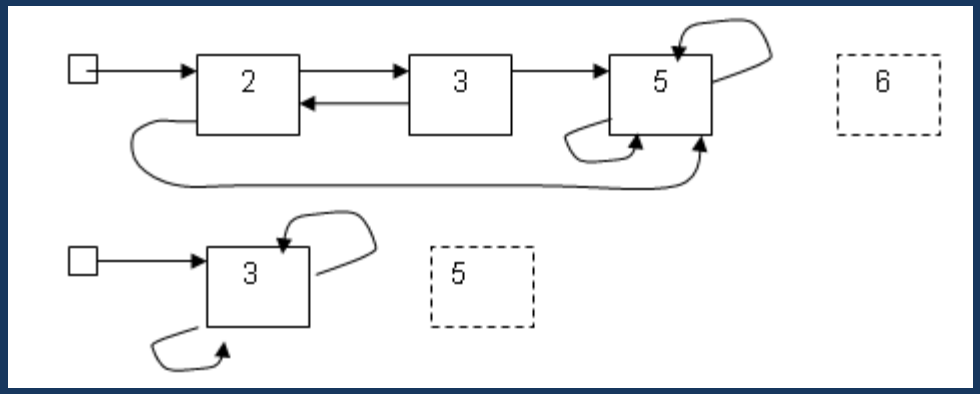



```
5 3 6 2  | 2 6 3 5
5 3 6  | 6 3 5
2  | 2
5 6  | 6 5
2 3  | 3 2
5 2 3  | 3 2 5
2 3 5  | 5 3 2
5 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2  | 3 5
2  | 2
```

```
void toString(list *q){
    if(q == NULL)
        return;
    printf("%d ",q->v);
    int i=0;
    for(list *p=q->next;p!=q && i<20 ;p=p->next,i++)
        printf("%d ",p->v);
    i=0;
    for(list *p=q->prev;p!=q && i<20;p=p->prev,i++)
        printf("%d ",p->v);
    printf("%d ",q->v);
    puts("");
    }
```

Отладка. Граничный дефект

- Обнаруженный дефект 5: «Очепятка» pmin->before(out) вместо head->before(out) из-за копипаста




```
5 3 6 2  | 2 6 3 5
5 3 6  | 6 3 5
2  | 2
5 6  | 6 5
2 3  | 3 2
6  | 6
2 3 5  | 5 3 2
2 3 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  | 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 2
```

При выходе из цикла **pmin** ссылается на минимальный элемент, найденный на последнем шаге, а он, в свою очередь, помещается в конец списка **out,** таким образом, **pmin.before(out)** вставляет **себя перед собой**