Лекции 12 - 13

Динамические структуры данных: списки

Динамические структуры данных

Динамическая структура данных – структура данных создаваемая в процессе выполнения программы и размещаемая в динамически выделенной памяти.

Классификация

По способу хранения:

- Последовательное хранение;
- Связанное хранение.

По методу доступа:

- Произвольный доступ;
- Упорядоченный доступ.

По логической структуре:

- Линейные;
- Разветвляющиеся;
- Произвольные.

Динамическая структура данных список

Список – линейная динамическая структура данных, как правило, одного типа, произвольного доступа к элементам, каждый элемент которой имеет два соседних элемента, называемых предыдущим и последующим элементом в списке (сам элемент в этом случае называется текущим).

Основные операции для работы со списками:

- Перемещение по списку;
- Добавление элементов в список;
- Удаление элементов из списка;
- Удаление всего списка;
- Доступ к элементам списка;
- Дополнительные операции (сортировка, поиск и т.д. и т.п.).

Виды списков

Виды списков определяются исходя из метода хранения:

- Последовательные списки;
- Связанные списки;
- Гибридные (смешанные) списки.

Последовательные списки

Основные переменные используемые для работы с последовательным списком:

- Указатель на начало списка;
- Текущий размер списка.

Принцип организации списка с последовательным хранением:

1-ый элемент	2-ой элемент	3-ий элемент	…	N-ый элемент

Переменные и типы для работы с последовательным списком

Типы данных:

```
typedef struct{
  TYPE *list;
  int count;
} LIST;
```

Добавление элемента в конец списка

```
int Add(LIST *ls, TYPE val)
{
  TYPE *tmp = (TYPE*)realloc(ls->list,(ls->count+1)*sizeof(TYPE));
  if(!tmp) return 0;
  if(tmp!=ls->list) ls->list = tmp;
  ls->list[ls->count] = val;
  ls->count++;
  return 1;
}
```

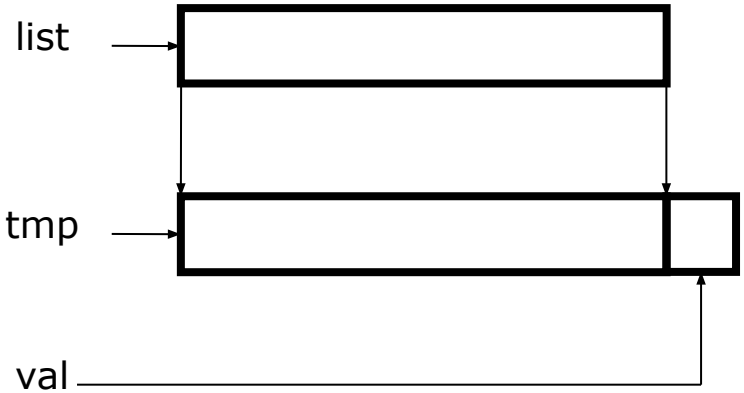
Вставка элемента на определенную позицию в списке

```
int Ins(LIST *ls,TYPE val, int ind)
{
  if(ind<0) return 0;
  if(ls->count <= ind) return Add(ls,val);
  TYPE *tmp = (TYPE*)realloc(ls->list,(ls->count+1)*sizeof(TYPE));
  if(!tmp) return 0;
  if(tmp!=ls->list) ls->list = tmp;
  for(int i=ls->count;i>ind;i--) ls->list[i] = ls->list[i-1];
  ls->list[ind] = val;
  ls->count++;
  return 1;
}
```

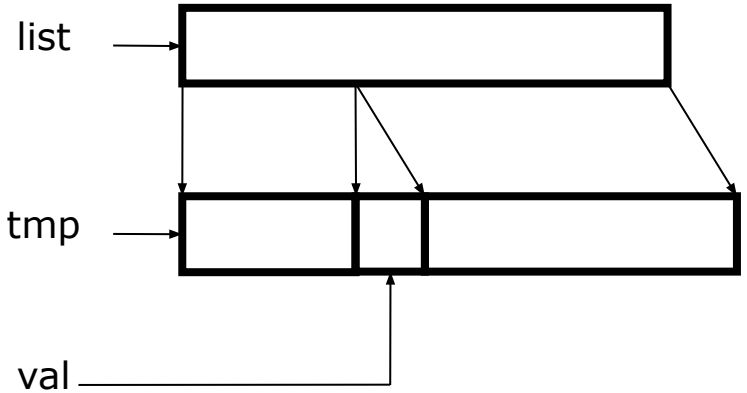
Удаление элемента из списка

```
int Del(LIST *ls, int ind)
{
  if((ind<0)||(ind>=ls->count)) return 0;
  for(int i=ind;i<ls->count-1;i++) ls->list[i] = ls->list[i+1];
  ls->list = (TYPE*)realloc(ls->list,(ls->count-1)*sizeof(TYPE));
  ls->count--;
  return 1;
}
```

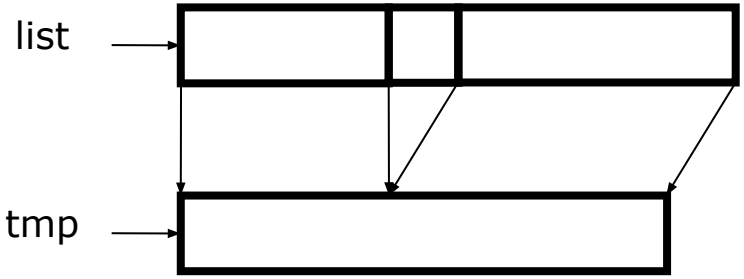
Графическое изображение операция добавления, вставки и удаления элемента



(а)



(б)



(в)

Изменение и получение элемента из списка

```
int Set(LIST *ls, TYPE val, int ind)
{
  if((ind<0)||(ind>=ls->count)) return 0;
  ls->list[ind] = val;
  return 1;
}

int Get(LIST *ls, TYPE *val, int ind)
{
  if((ind<0)||(ind>=ls->count)) return 0;
  *val = ls->list[ind];
  return 1;
}
```

Удаление всего списка, определение его размера, инициализация

```
void Destroy(LIST *ls)
{
  if(ls->list) free(ls->list);
  ls->list = NULL; ls->count = 0;
}

int Count(LIST *ls) { return ls->count; }

void Init(LIST *ls)
{
  ls->list = NULL; ls->count = 0;
}
```

Пример использования

Пользователь с клавиатуры вводит целые числа. Признак завершения – ввод пустой строки. Вывести на экран сначала четные значения, упорядоченные по возрастанию, а затем нечетные значения, упорядоченные по убыванию).

Пример использования

```
typedef int TYPE;
int cmpInc(const void *p1,const void *p2)
{
  return *((int*)p1) - *((int*)p2);
}
int cmpDec(const void *p1,const void *p2)
{
  return *((int*)p2) - *((int*)p1);
}
int main(int argc, char *argv[])
{
  LIST list1 = {NULL,0}, list2 = {NULL,0};
  while(1){
    char str[20];
    gets(str);
    if(str[0]==0) break;
    int val = atoi(str);
    Add((val%2==0)?&list2:&list1,val);
  }
  qsort(list2.list,list2.count,sizeof(TYPE),cmpInc);
  qsort(list1.list,list1.count,sizeof(TYPE),cmpDec);
```

Пример использования

```
  printf("\nЧетные значения: ");
  for(int i=0,n=Count(&list2);i<n;i++){
    int val;
    Get(&list2,&val,i);
    printf("%d ",val);
  }
  printf("\nНечетные значения: ");
  for(int i=0,n=Count(&list1);i<n;i++){
    int val;
    Get(&list1,&val,i);
    printf("%d ",val);
  }
  puts("");
  Destroy(&list1); Destroy(&list2);
  return 0;
}
```

Модификация

```
int InsInc(LIST *lst, TYPE val)
{
  for(int i=0,n=Count(lst);i<n;i++){
    int tmp;
    Get(lst,&tmp,i);
    if(tmp>=val) return Ins(lst,val,i);
  }
  return Add(lst,val);
}
int InsDec(LIST *lst, TYPE val)
{
  for(int i=0,n=Count(lst);i<n;i++){
    int tmp;
    Get(lst,&tmp,i);
    if(tmp<val) return Ins(lst,val,i);
  }
  return Add(lst,val);
}
```

Модификация (цикл ввода)

```
while(1){
  char str[20];
  gets(str);
  if(str[0]==0) break;
  int val = atoi(str);
  if(val%2==0) InsInc(&list2,val);
    else InsDec(&list1,val);
}
```

Преимущества и недостатки списков с последовательным хранением

Преимущества:

- Быстрый доступ к элементам списка посредством индексации.

Недостатки:

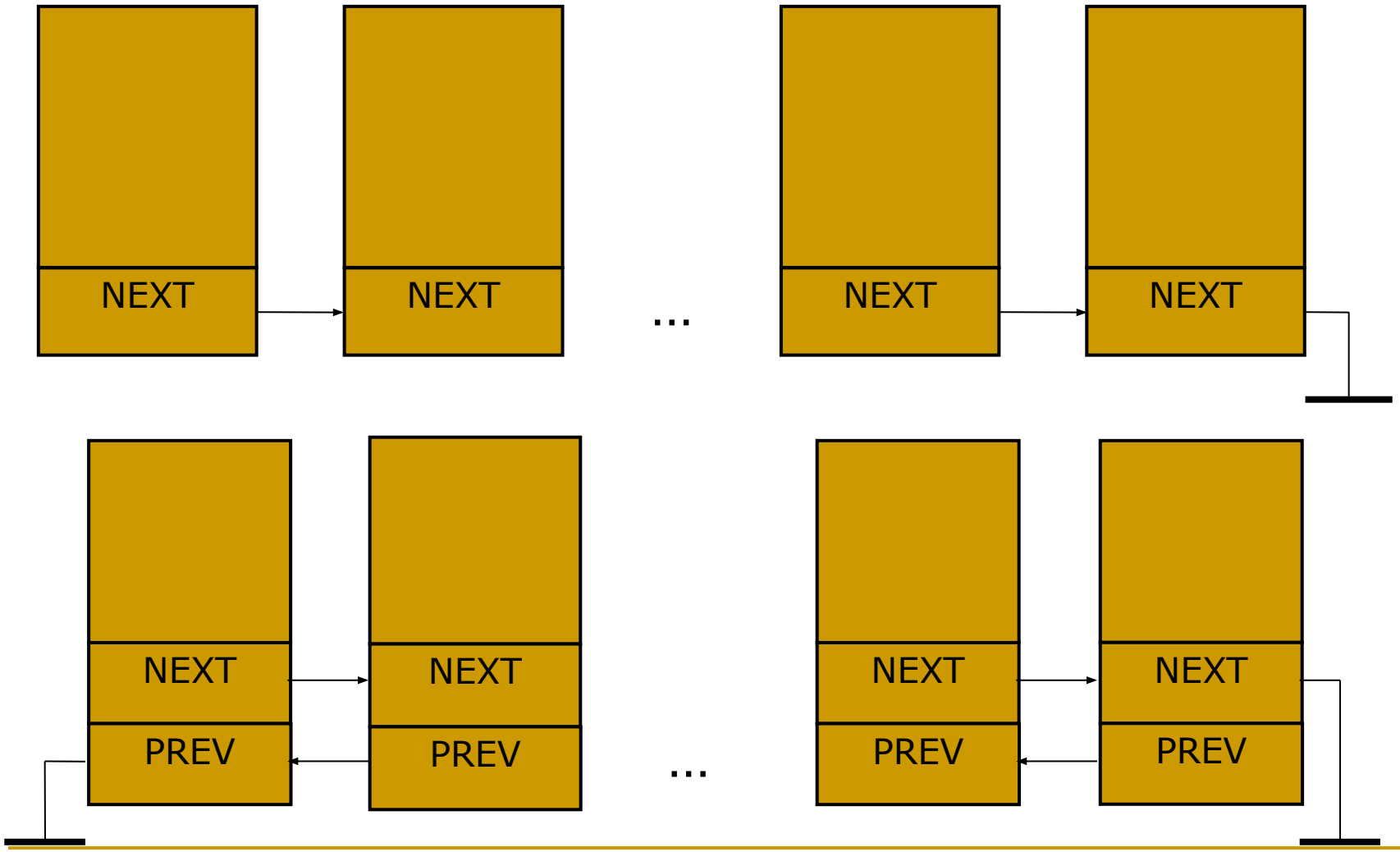
- Усложненность операций добавления и удаления элементов.

Списки со связанным хранением

Виды списков со связанным хранением:

- Однонаправленные списки;
- Двунаправленные списки.

Графическое изображение



Типы данных и переменные для работы со связанными списками

Необходимо иметь элемент списка, который бы агрегировал три поля:

- Данные заносимые в список;
- Указатель на следующий элемент в списке;
- Указатель на предыдущий элемент списка.

Пример для двунаправленного списка

Переменные и типы:

```
typedef struct _Element{
    TYPE val;
    struct _Element *next, *prev;
} Element;

typedef struct{
  Element *head, *curr;
} LIST;
```

Перемещение по списку

```
int MoveHead(LIST *ls)
{
  ls->curr = ls->head;
  if(ls->head == NULL) return 0;
  return 1;
}
int MoveNext(LIST *ls)
{
  if((ls->curr == NULL)||(ls->curr->next==NULL)) return 0;
  ls->curr = ls->curr->next;
  return 1;
}
int MovePrev(LIST *ls)
{
  if((ls->curr == NULL)||(ls->curr->prev==NULL)) return 0;
  ls->curr = ls->curr->prev;
  return 1;
}
```

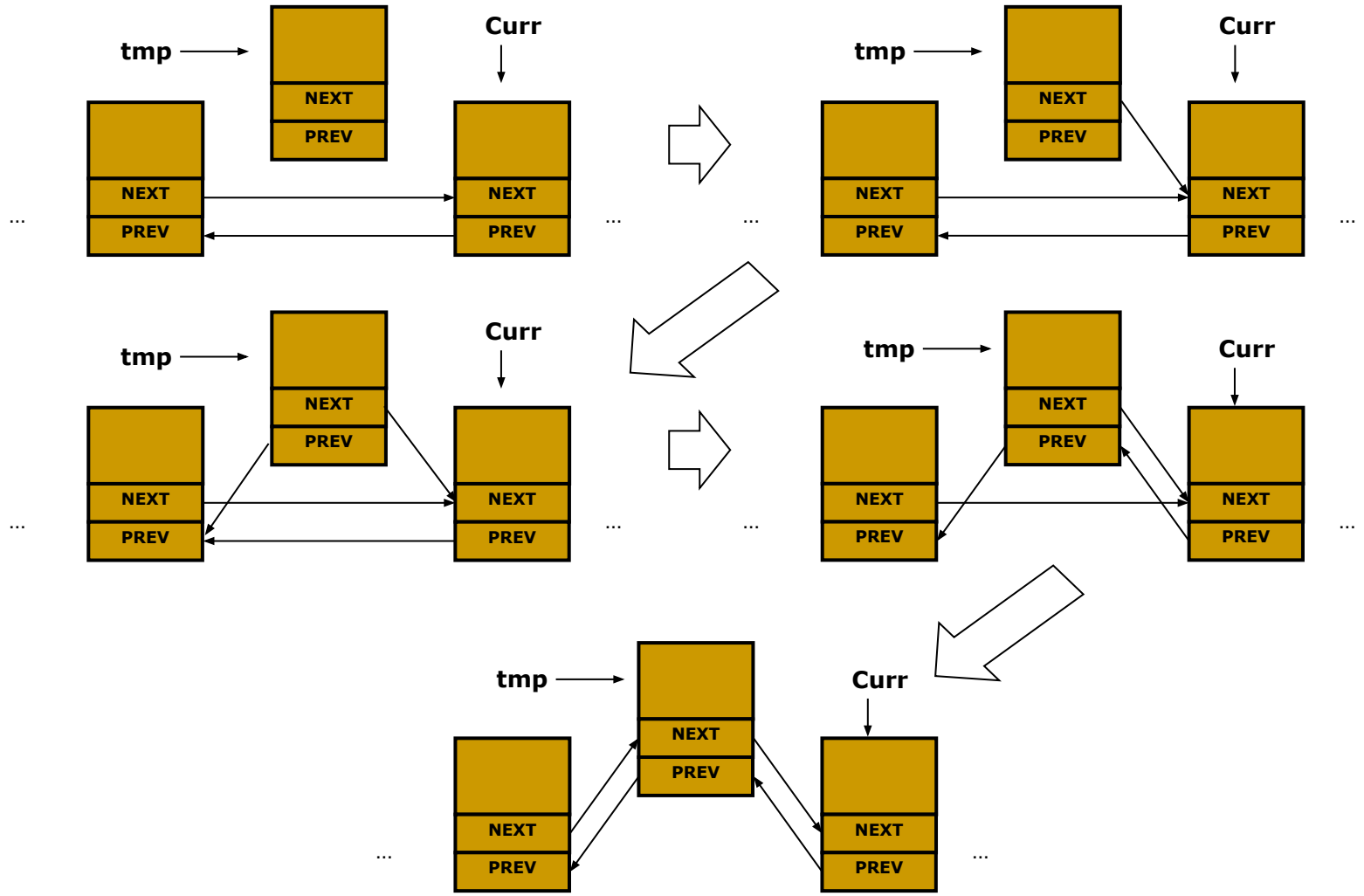
Добавление элемента в конец списка

```
int Add(LIST *ls, TYPE val)
{
  Element *tmp = (Element *)malloc(sizeof(Element));
  if(!tmp) return 0;
  if(!ls->head){
    ls->head=tmp; tmp->prev=NULL; tmp->next=NULL;
  }else{
    if(!ls->curr) ls->curr=ls->head;
    while(ls->curr->next) ls->curr=ls->curr->next;
    ls->curr->next=tmp;
    tmp->prev=ls->curr;
    tmp->next=NULL;
  }
  tmp->val=val; ls->curr=tmp;
  return 1;
}
```

Добавление элемента перед текущим элементом в списке

```
int Ins(LIST *ls, TYPE val)
{
  if(!ls->curr) return Add(ls,val);
  Element * tmp=(Element *)malloc(sizeof(Element));
  if(!tmp) return 0;
  tmp->next=ls->curr;
  tmp->prev=ls->curr->prev;
  ls->curr->prev=tmp;
  if(tmp->prev) tmp->prev->next=tmp;
    else ls->head=tmp;
  tmp->val=val; ls->curr = tmp;
  return 1;
}
```

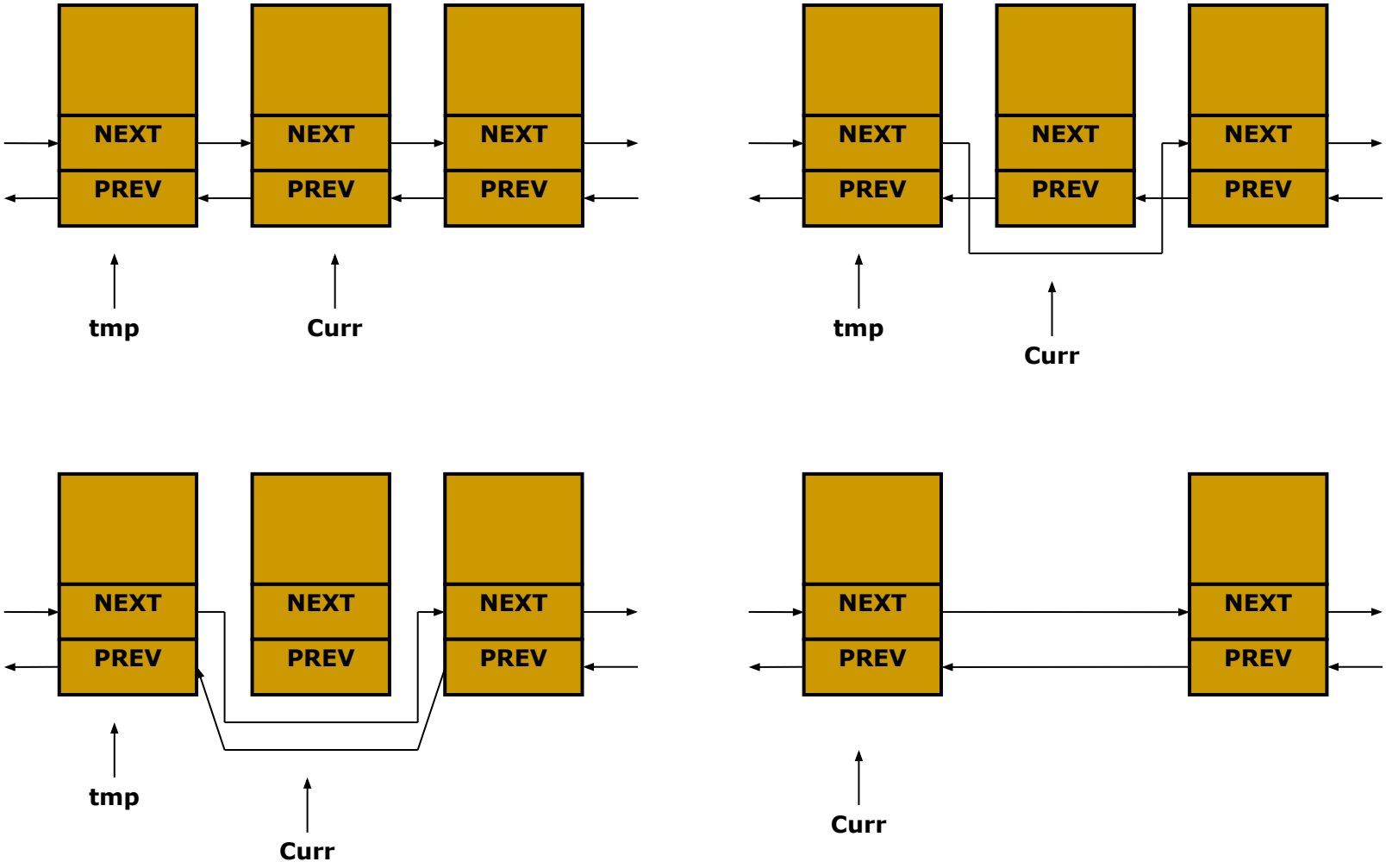
Вставка элемента



Удаление текущего элемента

```
int Del(LIST *ls)
{
  if(ls->curr==NULL) return 0;
  Element *tmp=ls->curr->prev;
  if(!tmp){
    ls->head=ls->head->next; ls->head->prev=NULL;
  }else{
    tmp->next=ls->curr->next;
    if(ls->curr->next) ls->curr->next->prev=tmp;
  }
  free(ls->curr); ls->curr=tmp;
  return 1;
}
```

Удаление элемента



Запись и получение значения элемента

```
int Set(LIST *ls, TYPE val)
{
  if(ls->curr == NULL) return 0;
  ls->curr->val = val;
  return 1;
}

int Get(LIST *ls,TYPE *val)
{
  if(ls->curr == NULL) return 0;
  *val = ls->curr->val;
  return 1;
}
```

Удаление и инициализация списка

```
void Init(LIST *ls)
{
  ls->head = ls->curr = NULL;
}
void Destroy(LIST *ls)
{
  while(ls->head != NULL){
    ls->curr = ls->head;
    ls->head = ls->head->next;
    free(ls->curr);
  }
  ls->curr = NULL;
}
```

Пример использования

```
void SortInc(LIST *ls)
{
  if((ls->head == NULL)||
     (ls->head->next == NULL)) return;
  int flag = 1;
  while(flag){
    flag = 0;
    ls->curr = ls->head;
    while(ls->curr->next != NULL){
      if(ls->curr->val > ls->curr->next->val){
        TYPE tmp = ls->curr->val;
        ls->curr->val = ls->curr->next->val;
        ls->curr->next->val = tmp;
        flag = 1;
      }
      ls->curr = ls->curr->next;
    }
  }
  ls->curr = ls->head;
}
```

```
void SortDec(LIST *ls)
{
  if((ls->head == NULL)||
     (ls->head->next == NULL)) return;
  int flag = 1;
  while(flag){
    flag = 0;
    ls->curr = ls->head;
    while(ls->curr->next != NULL){
      if(ls->curr->val < ls->curr->next->val){
        TYPE tmp = ls->curr->val;
        ls->curr->val = ls->curr->next->val;
        ls->curr->next->val = tmp;
        flag = 1;
      }
      ls->curr = ls->curr->next;
    }
  }
  ls->curr = ls->head;
}
```

Пример использования

```
int main(int argc, char *argv[])
{
  LIST list1 = {NULL,NULL}, list2 = {NULL,NULL};
  while(1){
    char str[20];
    gets(str);
    if(str[0]==0) break;
    int val = atoi(str);
    Add((val%2==0)?&list2:&list1,val);
  }
  SortInc(&list2); SortDec(&list1);
```

Пример использования

```
  printf("\nЧетные значения: ");
  if(MoveHead(&list2))
    do{
      int val; Get(&list2,&val);
      printf("%d ",val);
    }while(MoveNext(&list2));

  printf("\nНечетные значения: ");
  if(MoveHead(&list1))
    do{
      int val; Get(&list1,&val);
      printf("%d ",val);
    }while(MoveNext(&list1));
  puts("");
  Destroy(&list1); Destroy(&list2);
  return 0;
}
```

Модификация

```
int InsInc(LIST *ls, TYPE val)
{
  if(MoveHead(ls))
    do{
      int tmp;
      Get(ls,&tmp);
      if(tmp>=val)
         return Ins(ls,val);
    }while(MoveNext(ls));
  return Add(ls,val);
}
```

```
int InsDec(LIST *ls, TYPE val)
{
  if(MoveHead(ls))
    do{
      int tmp;
      Get(ls,&tmp);
      if(tmp<=val)
        return Ins(ls,val);
    }while(MoveNext(ls));
  return Add(ls,val);
}
```

Модификация (цикл ввода)

```
while(1){
  char str[20];
  gets(str);
  if(str[0]==0) break;
  int val = atoi(str);
  if(val%2==0) InsInc(&list2,val);
    else InsInc(&list1,val);
}
```

Преимущества и недостатки списков со связанным хранением

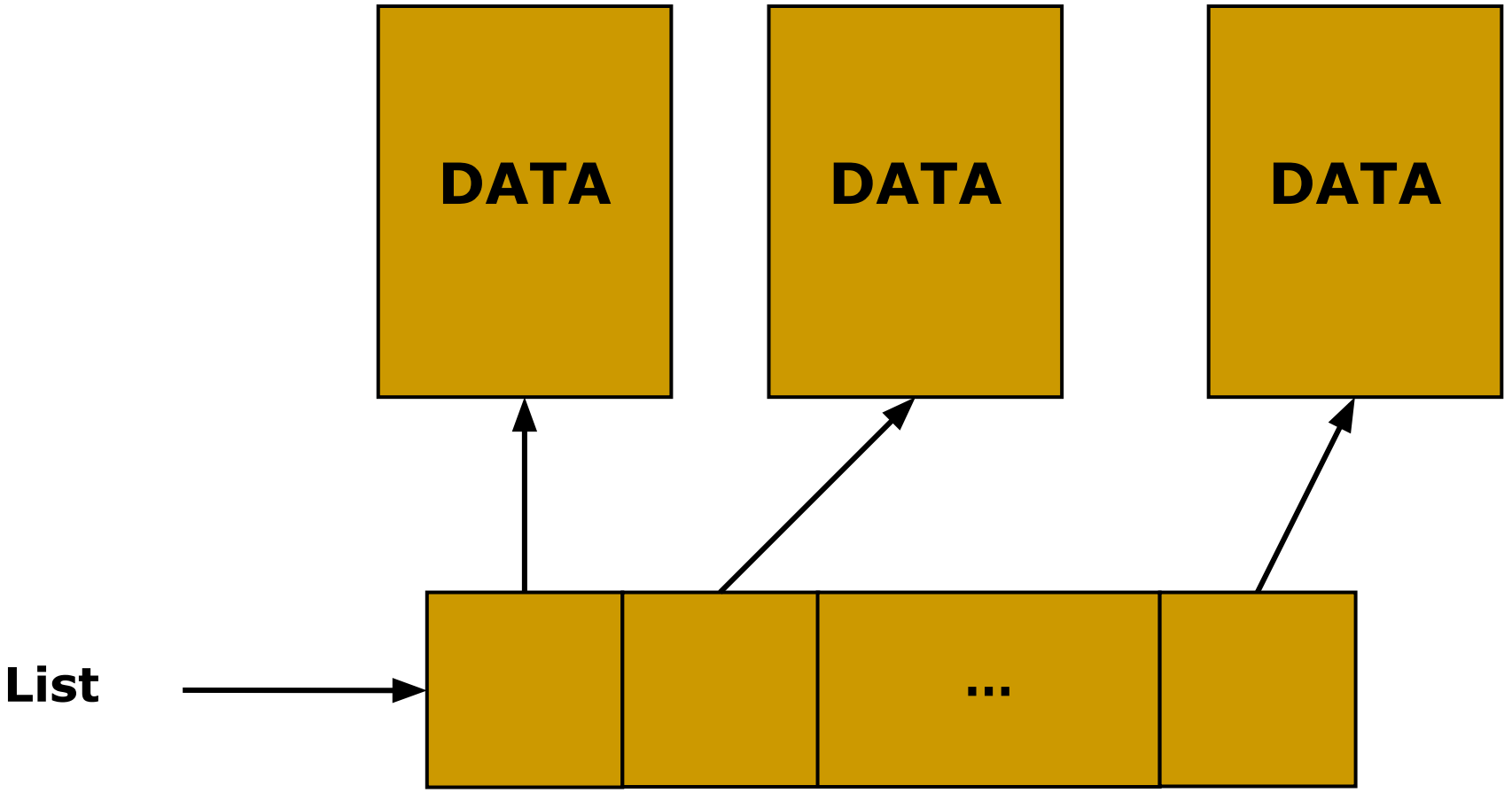
Преимущество:

- Эффективные функции работы с памятью при добавлении или удалении элементов

Недостаток:

- Нет возможности произвольного обращения к элементам списка.

Смешанный список



Переменные и типы данных

```
typedef struct{
  TYPE **list;  //Указатель на начало списка
  int count;     //Количество элементов в списке
} LIST;
```

Инициализация и уничтожение списка

```
void Init(LIST *ls)
{
  ls->list = NULL;
  ls->count = 0;
}

void Destroy(LIST *ls)
{
  for(int i=0;i<ls->count;i++) free(ls->list[i]);
  free(ls->list);
  ls->list = NULL;
  ls->count = 0;
}
```

Добавление элемента в конец списка

```
int Add(LIST *ls, TYPE val)
{
  TYPE *new = (TYPE*)malloc(sizeof(TYPE));
  if(!new) return 0;
  TYPE **tmp = (TYPE**)realloc(ls->list,(ls->count+1)*sizeof(TYPE*));
  if(!tmp) {free(new); return 0;}
  *new = val;
  if(ls->list != tmp) ls->list = tmp;
  ls->list[ls->count++] = new;
  return 1;
}
```

Вставка элемента в середину списка

```
int Ins(LIST *ls, TYPE val, int ind)
{
  if(ind<0) return 0;
  if(ind >= ls->count) return Add(ls,val);
  TYPE *new = (TYPE*)malloc(sizeof(TYPE));
  if(!new) return 0;
  TYPE **tmp = (TYPE**)realloc(ls->list,(ls->count+1)*sizeof(TYPE*));
  if(!tmp) {free(new); return 0;}
  *new = val;
  if(ls->list != tmp) ls->list = tmp;
  for(int i=ls->count;i>ind;i--) ls->list[i] = ls->list[i-1];
  ls->list[ind] = new;
  ls->count++;
  return 1;
}
```

Удаление элемента из списка

```
int Del(LIST *ls, int ind)
{
  if((ind<0)||(ind>=ls->count)) return 0;
  free(ls->list[ind]);
  for(int i=ind;i<ls->count-1;i++) ls->list[i] = ls->list[i+1];
  ls->list = (TYPE**)realloc(ls->list,(ls->count-1)*sizeof(TYPE*));
  ls->count--;
  return 1;
}
```

Запись и получение значения из списка

```
int Set(LIST *ls, TYPE val, int ind)
{
  if((ind<0)||(ind>=ls->count)) return 0;
  *ls->list[ind] = val;
  return 1;
}

int Get(LIST *ls, TYPE *val, int ind)
{
  if((ind<0)||(ind>=ls->count)) return 0;
  *val = *ls->list[ind];
  return 1;
}
```

Пример использования

```
int CmpInc(const void *p1, const void *p2)
{
  return **((int**)p1) - **((int**)p2);
}

int CmpDec(const void *p1, const void *p2)
{
  return **((int**)p2) - **((int**)p1);
}
```

Пример использования

```
int main(int argc, char *argv[])
{
  LIST list1 = {NULL,0}, list2 = {NULL,0};
  while(1){
    char str[20];
    gets(str);
    if(str[0]==0) break;
    int val = atoi(str);
    Add((val%2==0)?&list2:&list1,val);
  }
  qsort(list1.list,list1.count,sizeof(TYPE*),CmpDec);
  qsort(list2.list,list2.count,sizeof(TYPE*),CmpInc);
```

Пример использования

```
  printf("\nЧетные значения: ");
  for(int i=0,n=Count(&list2);i<n;i++){
    int val;
    Get(&list2,&val,i);
    printf("%d ",val);
  }
  printf("\nНечетные значения: ");
  for(int i=0,n=Count(&list1);i<n;i++){
    int val;
    Get(&list1,&val,i);
    printf("%d ",val);
  }
  puts("");
  Destroy(&list2); Destroy(&list1);
  return 0;
}
```

Модификация

```
int InsInc(LIST *lst, TYPE val)
{
  for(int i=0,n=Count(lst);i<n;i++){
    int tmp;
    Get(lst,&tmp,i);
    if(tmp>=val) return Ins(lst,val,i);
  }
  return Add(lst,val);
}
int InsDec(LIST *lst, TYPE val)
{
  for(int i=0,n=Count(lst);i<n;i++){
    int tmp;
    Get(lst,&tmp,i);
    if(tmp<val) return Ins(lst,val,i);
  }
  return Add(lst,val);
}
```

Модификация (цикл ввода)

```
while(1){
  char str[20];
  gets(str);
  if(str[0]==0) break;
  int val = atoi(str);
  if(val%2==0) InsInc(&list2,val);
    else InsDec(&list1,val);
}
```

Смешанный список

Преимущество:

- Быстрый доступ к элементам списка посредством индексации

Недостаток:

- Усложненность операций выделения и освобождения памяти при добавлении и удалении элементов

Кольца