线性表

▼ 抽象类型定义

```
ADT List {
    数据对象：D={ai | 1≤i≤n, n>0, ai ∈ Elemtype}
    数据关系：R={<ai, ai+1> | ai, ai+1 ∈ D, i=1,2,...n-1}
    基本运算：
    InitList(&L);
    DestroyList(&L);
    ListEmpty(L);
    ListLength(L);
    GetElem(L, i, &e);
    LocateElem(L, e);
    ListInsert(&L,i,e);
    ListDelete(&L,i,&e);
}
```

线性表L = （a1,a2,a3,.....ai,.....an），下列说法正确的是

A. 每个元素都有一个直接前件和直接后件

B. 线性表中至少要有一个元素

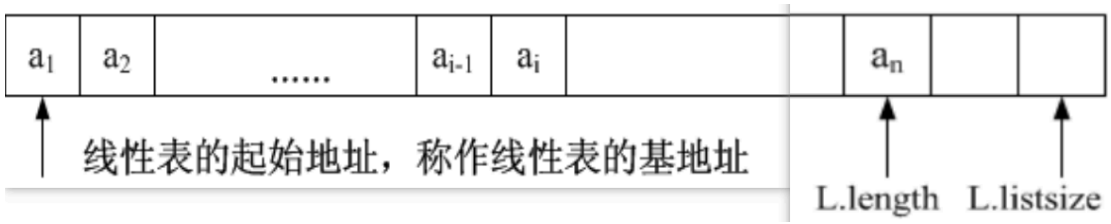
C. 表中诸元素的排列顺序必须是由小到大或由大到小

D. 除第一个元素和最后一个元素外，其余每个元素都有一个且只有一个直接前件和直接后件

解析：本题考点是线性表的性质。线性表的特点是：除第一个元素和最后一个元素外，其余每个元素都有一个且只有一个直接前件和直接后件。因此，本题参考答案是D。

▼ 顺序表

- 地址连续
- $LOC(a_i) = LOC(a_1) + (i-1) \times C$
- 线性表长度可变



- InitList、ListLength、GetElem $O(1)$
- LocateElem、ListInsert、ListDelete $O(n)$

静态顺序表

```
//静态顺序表
typedef struct{
    int data[MaxSize];
    int length;
}SqList;

void InitList(SqList &L){
    for(int i=0;i<MaxSize;i++){
        L.data[i]=0;
    }
    L.length=0;
}

bool ListInsert(SqList &L,int i,int e){
    if(i<1||i>L.length+1) return false;
    if(L.length>=MaxSize) return false;
    for(int j=L.length;j>=i;j--){
        L.data[j]=L.data[j-1];
    }
    L.data[i-1]=e;
```

```
    L.length++;
    return true;
}

bool ListDelete(SqList &L,int i,int &e){
    if(i<1||i>L.length+1) return false;
    e=L.data[i-1];
    for(int j=i;j<L.length;j++){
        L.data[j-1]=L.data[j];
    }
    L.length--;
    return true;
}

int GetElem(SqList L,int i){
    if(i<1||i>L.length+1){
        printf("i的值不合法\n");
        return -1;
    }
    return L.data[i-1];
}

int LocateElem(SqList L,int e){
    for(int i=0;i<L.length;i++){
        if(L.data[i]==e) return i+1;
    }
    return 0;
}
```

动态顺序表

```
//动态顺序表
typedef struct{
    int *data;
    int Maxsize;
    int length;
```

```
}SeqList;

void InitList(SeqList &L,int InitSize){
    L.data=(int *)malloc(InitSize*sizeof(int));
    L.length=0;
    L.Maxsize=InitSize;
}

void IncreaseSize(SeqList &L,int len){
    int *p=L.data;
    L.data=(int *)malloc((L.Maxsize+len)*sizeof(int));
    //将数据复制到新区域
    for(int i=0;i<L.length;i++){
        L.data[i]=p[i];
    }
    L.Maxsize=L.Maxsize+len;
    free(p);
}

void DestroyList(SeqList &L){
    if(L.data!=NULL){
        free(L.data);
        L.data=NULL;
        L.Maxsize=0;
        L.length=0;
    }
}
```

▼ 链表

按实现分：静态链表、动态链表

按链接形式分：单链表、双向链表、循环链表

动态链表

▼ 单链表

```
typedef struct LNode{
    int data;
    struct LNode *next;
}LNode,*LinkList;
```

▼ 带头结点（头结点不存放值）

- ListLength、ListInsert、ListDelete $O(n)$

```
bool InitList(LinkList &L){
    L=(LNode *)malloc(sizeof(LNode));
    if(L==NULL) return false;//内存不足，分配失败
    L→next=NULL;
    return true;
}

bool Empty(LinkList L){
    if(L→next==NULL) return true;
    else return false;
}

LNode *GetElem(LinkList L,int i){
    if(i<0) return NULL;
    LNode *p;
    p=L;
    int j=0;
    while(p!=NULL && j<i){
        p=p→next;
        j++;
    }
    return p;
}

int ListLength(LinkList L){
    int len=0;
```

```
    LNode *p=L->next;
    while(p!=NULL){
        p=p->next;
        len++;
    }
    return len;
}

//后插:在p结点之后插入元素e
bool InsertNextNode(LNode *p,int e){
    if(p==NULL) return false;  //i值太大了
    LNode *s=(LNode*)malloc(sizeof(LNode));
    //将s结点连接到p之后
    s->data=e;
    s->next=p->next;
    p->next=s;
    return true;
}

//前插:在p结点之前插入元素e
bool InsertPriorNode(LNode*p,int e){
    if(p==NULL) return false;
    LNode*s=(LNode*)malloc(sizeof(LNode));
    if(s==NULL) return false;
    //在p结点之后插入s结点
    s->next=p->next;
    p->next=s;
    //调换p与s的数据
    s->data=p->data;
    p->data=e;
    return true;
}

//按位序插入
bool ListInsert(LinkList &L,int i,int e){
    if(i<1) return false;
```

```
    LNode *p=GetElem_head(L,i-1);
    return InsertNextNode(p,e);
}

//按位序删除
bool ListDelete(LinkList &L,int i,int &e){
    if(i<1) return false;
    LNode *p=GetElem_head(L,i-1);
    if(p==NULL || p->next==NULL) return false;
    LNode *q=p->next;
    e=q->data;
    p->next=q->next;
    free(q);
    return true;
}

//删除指定结点
//思想:将p的后继结点q的值移至p 再删除q
//存在问题:若p为最后一个结点则会报错
bool DeleteNode(LNode *p){
    if(p==NULL) return false;
    LNode *q=p->next;
    p->data=p->next->data;
    p->next=q->next;
    return true;
}

void ClearList(LinkList &L){
    LNode*p=NULL;
    LNode*q=L->next;
    while(q!=NULL){
        p=q;
        q=q->next;
        free(p);
    }
    L->next=NULL;
```

```
}

void DestroyList(LinkList &L){
    ClearList(L);
    free(L);
    L=NULL;
}
```

建立单链表

```
//尾插法建立单链表
//r指向尾结点
LinkList List_TailInsert(LinkList &L){
    printf("尾插法:输入需要插入的元素，以-1结尾\n");
    int x;
    L=(LinkList)malloc(sizeof(LNode));
    LNode *s,*r=L;
    scanf("%d",&x);
    while(x!=-1){
        s=(LNode *)malloc(sizeof(LNode));
        s→data=x;
        r→next=s;
        r=s; //r移动到最后
        scanf("%d",&x);
    }
    r→next=NULL;
    return L;
}

//头插法
//可用于倒置
LinkList List_HeadInsert(LinkList &L){
    printf("头插法:输入需要插入的元素，以-1结尾\n");
    L=(LinkList)malloc(sizeof(LNode));
    int x;
    LNode *s;
```

```
    L->next=NULL;
    scanf("%d",&x);
    //对头指针进行尾插
    while(x!=-1){
        s=(LNode*)malloc(sizeof(LNode));
        s->data=x;
        s->next=L->next;
        L->next=s;
        scanf("%d",&x);
    }
    return L;
}
```

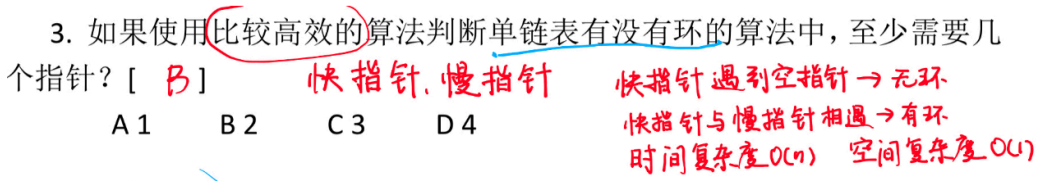
▼ 不带头结点

```
bool InitList(LinkList &L){
    L=NULL;
    return true;
}

bool Empty(LinkList L){
    return (L==NULL);
}

LNode *GetElem(LinkList L,int i){
    if(i<0) return NULL;
    LNode *p;
    p=L;
    int j=1;
    while(p!=NULL && j<i){
        p=p->next;
        j++;
    }
    return p;
}
```

```
LNode *LocateElem(LinkList L,int e){
    LNode *p=L;
    while(p!=NULL && p->data!=e){
        p=p->next;
    }
    return p;
}

int Length(LinkList L){
    int len=0;
    LNode *p=L;
    while(p!=NULL){
        p=p->next;
        len++;
    }
    return len;
}

bool ListInsert(LinkList &L,int i,int e){
    if(i<1) return false;
    if(i==1){
        LNode *s=(LNode*)malloc(sizeof(LNode));
        s->data=e;
        s->next=L;
        L=s;  //改头指针
        return true;
    }
    LNode *p=GetElem(L,i-1);
    return InsertNextNode(p,e);
}

bool ListDelete(LinkList &L,int i,int &e){
    if(i<1) return false;
    if(i==1){
        LNode *s=L;
```

```
        L=L->next;
        e=s->data;
        free(s);
        return true;
    }
    LNode *p;
    p=L;
    int j=1;
    //循环至第i-1个结点
    while(p!=NULL && j<i-1){
        p=p->next;
        j++;
    }
    if(p==NULL || p->next==NULL) return false;
    LNode *q=p->next;
    e=q->data;
    p->next=q->next;
    free(q);
    return true;
}

void ClearList(LinkList &L){
    LNode*p=NULL;
    LNode*q=L;
    while(q!=NULL){
        p=q;
        q=q->next;
        free(p);
    }
    L=NULL;
}
//不带头结点的链表清空与销毁链表的结果是一样的
void DestroyList(LinkList &L){
    ClearList(L);
}
```

- **判断单链表是否有环**



▼ **双向链表**

```
typedef struct DNode{
   int data;
   struct DNode *prior,*next;
}DNode,*DLinkList;
```

基本操作

```
bool InitDLinkList(DLinkList &L){
   L=(DNode*)malloc(sizeof(DNode));
   if(L==NULL) return false;
   L→prior=NULL;
   L→next=NULL;
   return true;
}

bool Empty(DLinkList L){
   return (L→next==NULL);
}

//按位序查找
DNode*  GetElem(DLinkList L,int i){
   if(i<0) return NULL;
   int j=0;
   DNode*p=L;
   while(p!=NULL && j<i){
      p=p→next;
      j++;
```

```
    }
    return p;
}

//按值查找
DNode* LocateElem(DLinkList L,int e){
    DNode *p=L->next;
    while(p!=NULL && p->data!=e){
        p=p->next;
    }
    return p;
}

//后插 p后插s
bool InsertNextDNode(DNode*p,DNode*s){
    if(p==NULL || s==NULL) return false;
    s->next=p->next;
    if(p->next!=NULL) p->next->prior=s;
    s->prior=p;
    p->next=s;
    return true;
}

//前插 p前插入s
bool InsertPriorNode(DNode *p,DNode *s){
    DNode *q=p->prior;
    return InsertNextDNode(q,s);
}

//按位序插入
bool ListInsert(DLinkList L,int i,int e){
    DNode* s=(DLinkList)malloc(sizeof(DNode));
    s->data=e;
    DNode*p=GetElem(L,i-1);
    return InsertNextDNode(p,s);
}
```

```
//删除p结点的后继结点q
bool DeleteNextDNode(DNode *p){
    if(p==NULL) return false;
    DNode *q=p→next;
    if(q==NULL) return false;
    p→next=q→next;
    if(q→next!=NULL)
        q→next→prior=p;
    free(q);
    return true;
}

//删除位序为i的结点并返回e
bool ListDelete(DLinkList &L,int i,int &e){
    DNode *p=GetElem(L,i-1);
    e=p→next→data;
    return DeleteNextDNode(p);
}

void DestroyList(DLinkList &L){
    while(L→next!=NULL){
        DeleteNextDNode(L);
    }
    free(L);
    L=NULL;
}
```

▼ 循环链表

```
//循环单链表
bool InitList(LinkList &L){
    L=(LNode*)malloc(sizeof(LNode));
    if(L==NULL) return false;
    //尾指针指向头
```

```
    L->next=L;
    return true;
}

//循环双链表
bool InitDList(DLinkList &L){
    L=(DNode *)malloc(sizeof(DNode));
    if(L==NULL) return false;
    L->prior=L;
    L->next=L;
    return true;
}

bool Locate(LinkList &L,int e,LinkList *&p){
    p=L->next;
    while(p!=L && p->data!=e) p=p->next;
    if(p==L) return false;
    else return true;
}
```

静态链表

```
typedef struct Node{
    int data;
    int next;
}SLinkList[MaxSize];
```

一元多项式的表示及相加

一元多项式

$$P_n(x) = p_1x^{e_1} + p_2x^{e_2} + p_3x^{e_3} + \cdots p_mx^{e_m}$$

满足

$$0 \le e_1 < e_2 < \cdots e_m = n$$

可以表示为：

$$((p_1, e_1), (p_2, e_2), \cdots (p_m, e_m))$$

其中每个元素有两个数据项：（系数项和指数项）

两个多项式的相加则可表示为指数项相同的数据元素对应的系数项相加。

对于只在表的首、尾进行插入操作的线性表，宜采用的存储结构为_

A. 顺序表

B. 用头指针表示的单循环链表

C. 用尾指针表示的单循环链表

D. 单链表

解析：本题考点是循环链表的优点。插入和删除方便的存储结构是链表，这是因为链表插入和删除时不需要移动元素就能实现。只在表的首、尾进行插入操作的线性表用尾指针表示的单循环链表最适宜，减少了移动指针的次数。因此，本题参考答案是C

1. LinkList mynote(LinkList L)
{//L 是不带头结点的单链表的头指针
if(L&&L->next){
q=L；L=L—>next；p=L；
S1： while(p—>next) p=p—>next；
S2： p—>next=q；q—>next=NULL；
}
return L；
}

头指针改为第二个元素

找到尾元素

尾元素接原来的头

请回答下列问题：
（1）说明语句 S1 的功能；
查询链表的尾结点
（2）说明语句组 S2 的功能；
将第一个结点链接到链表的尾部，作为新的尾结点
（3）设链表表示的线性表为（$a_1,a_2,\cdots,a_n$）,写出算法执行后的返回值所表示的线性表。
返回的线性表为（$a_2,a_3,\cdots,a_n,a_1$）

▼ 有序表

```
//结点结构
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}*SLink

typedef struct{
    SLink head,tail,curPtr;
    int length,curPos;
}OrderedLinkList;
```

合并La和Lb到Lc

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc) {
    // 已知单
    链线性表 La 和 Lb 的元素按值非递减排列。本算法
    // 归并 La 和 Lb 得到新的单链线性表 Lc，Lc 的元素也按
    // 值非递减排列。操作之后 La 和 Lb 消失
    LNode *pa = La→next; // La的当前节点指针
    LNode *pb = Lb→next; // Lb的当前节点指针
```

```
    Lc = rc = new LNode;  // 建空表，Lc 为头指针
    rc→next = nullptr;   // 初始化rc的next为nullptr

    while (pa && pb) {
        if (pa→data <= pb→data) {
            // 将 *pa 插入Lc表，指针 pa 后移
            rc→next = pa;
            rc = pa;
            pa = pa→next;
        } else {
            // 将 *pb 插入Lc表，指针 pb 后移
            rc→next = pb;
            rc = pb;
            pb = pb→next;
        }
    }

    // 若La还有剩余结点，将 La剩余结点插入Lc表
    if (pa) {
        rc→next = pa;
    }
    // 若Lb还有剩余结点，将 Lb剩余结点插入Lc表
    if (pb) {
        rc→next = pb;
    }

    // 释放La和Lb的头节点，因为它们的节点已经被Lc接管
    delete La;
    delete Lb;
}
```

删除有序表中重复的元素，只保留一个

```
void purge_OL(SqList &L) {
    int k = -1; // k 指示新表的表尾
    for (int i = 0; i < L.length - 1; ++i) { // 顺序考察表中每个元素
```

```
        if (k == -1 || L.elem[k] != L.elem[i]) {
            L.elem[++k] = L.elem[i]; // 在新表中不存在和L.elem[i]相同的元素
        }
    }
    L.length = k + 1; // 修改表长
}
```

查找链表中间结点

- p（快指针）：p=p→next→next
- q（慢指针）：q=q→next
- p=NULL 时 return q

基本操作

```
bool MakeNode(SLink &p,ElemType e){
    p=new LNode;
    if(!p) return False;
    p→data=e;
    p→next=NULL;
    return TRUE;
}

bool InitList(OrderedLinkList &L){
    L.head=NULL
    if(MakeNode(L.head,0){
        L.tail=L.curPtr=L.head;
        L.length=L.curPos=0;
        return TRUE;
    }
    else{
        L.head=NULL;
        return False;
    }
}
```

```
bool GetPos(OrderedLinkList L,int pos){
   if(pos<1||pos>L.len) return FALSE;
   if(L.curPos>pos){
      L.curPtr=L.head→next;
      L.curPos=1;
   }
   while(L.curPos<pos){
      L.curPtr=L.curPtr→next;++L.curPos;
   }
   return TRUE;
}

bool LocateElem(OrderedLinkList L,ElemType e,int(*compare)(ElemType,E
   L.current=L.head;
   L.curPos=0;
   while(L.current→next && compare(e,L.current→next→data)>0){
      L.current=L.current→next;
      L.curPos++l
   }
   if(L.current→next && compare(e,L.current→next→data)==0){
      L.current=L.current→next;
      L.curPos++;
      return TRUE;
   }
   else return FALSE;
}

//在当前指针所指结点之后插入s
void InsAfter(OrderedLinkList &L,SLink s){
   s→next=L.curPtr→next;
   L.curPtr→next=s;
   if(L.tail==L.curPtr) L.tail=s;  //在尾结点之后插入，修改尾指针
   L.curPtr=s;
   ++L.curPos;
   ++L.length;
}
```

```
bool DelAfter(OrderedLinkList &L,ElemType &e){
   if(L.curPtr==L.tail) return FALSE;
   p=L.curPtr→next;
   e=p→data;
   L.curPtr→next=p→next;
   if(L.tail==p) L.tail=L.curPtr;
   delete p;
   --L.length;
   return TRUE;
}
```

应用

```
//C=A∪B
void unionList(OrderLinkList A, OrderLinkList B, OrderLinkList &C) {
   if (InitList(C)) { // 初始化建空表
      int m = ListLength(A); // 分别求得表长
      int n = ListLength(B);
      int i = 1;
      int j = 1;
      while (i <= m || j <= n) {
         if (GetPos(A, i) && GetPos(B, j)) { // 两个表中都还有元素未曾考察到
            ElementType ea, eb;
            GetCurElem(A, ea);
            GetCurElem(B, eb);
            if (ea <= eb) { // 插入 A 中的元素
               if (!MakeNode(s, ea)) exit(1);
               ++i;
               if (ea == eb) ++j; // 舍弃 B 表中相同元素
            } else { // 插入 B 中的元素
               if (!MakeNode(s, eb)) exit(1);
               ++j;
            }
         } else if (GetPos(A, i)) { // A 表中尚有元素未曾插入
            ElementType ea;
```

```
            GetCurElem(A, ea);
            if (!MakeNode(s, ea)) exit(1);
            ++i;
        } else { // B 表中尚有元素未曾插入
            ElementType eb;
            GetCurElem(B, eb);
            if (!MakeNode(s, eb)) exit(1);
            ++j;
        }
        InsAfter(C, s); // 插入到 C 表
    }
  }
}

//C=A∩B
void Intersection(OrderLinkList A, OrderLinkList B, OrderLinkList &C) {
  if (InitList(C)) { // 初始化建空表
    int m = ListLength(A); // 分别求得表长
    int n = ListLength(B);
    int i = 1;
    int j = 1;
    while (i <= m && j <= n) { // 顺序考察表中元素
      if (GetPos(A, i) && GetPos(B, j)) { // 两个表中都还有元素未曾考察
        ElementType ea, eb;
        GetCurElem(A, ea);
        GetCurElem(B, eb);
        if (ea < eb) {
          ++i;
        } else if (ea > eb) {
          ++j;
        } else { // 插入相同的元素
          if (!MakeNode(s, ea)) exit(1);
          ++i;
          ++j;
          InsAfter(C, s); // 插入到 C 表
        }
```

```
        }
      }
    }
}
```