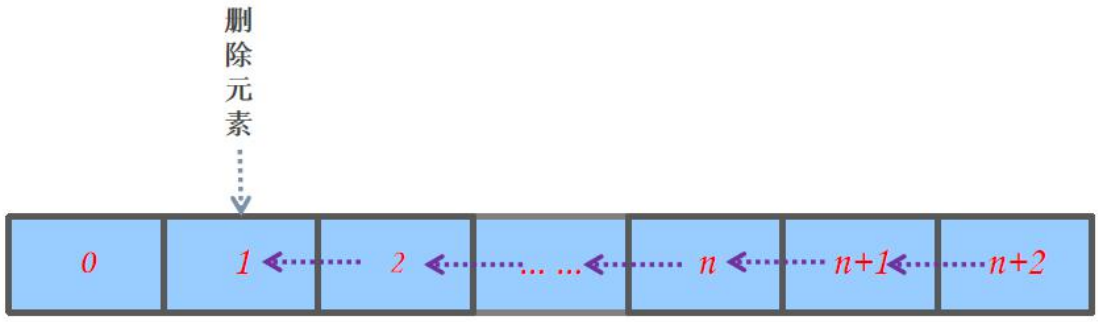
C/C++从入门到精通-高级程序员之路

【 数据结构 】

链表及其企业级应用

第 1 节 链表的故事导入

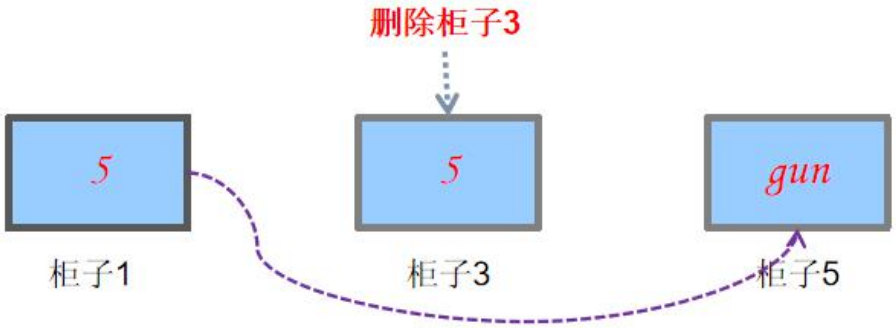
坐在屏幕前，看着满屏幕的星光，Jack 限入了沉思，做为一个程序员，直觉告诉他，他总感觉到使用顺序表存储运动中的星星状态虽然没有什么不妥，但是，每次移除一颗星星导致其后的所有元素都向前移动并不是一种高效的做法，如果不止几十颗星星，而是有成千上万颗星星，每**删除**一颗，无异于是对数据进行乾坤大挪移(同样，**插入**也是如此)... ...

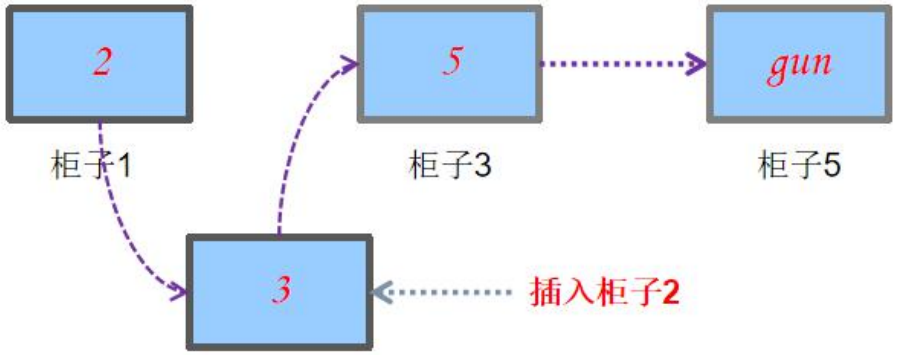


有没有一种方法，可以让删除或插入尽可能少的移动数据？
Jack 突然想到电影《龙之吻》中主人公刘剑（李连杰饰）取枪的场景，第一个柜子里虽然没有枪，但是有放枪柜子的地址（编号），那无论是哪个柜子放枪，我们只要知道了它的地址，就都可以取到枪。。。

即是说，无论我增加间接取枪的柜子数还是减少间接取枪的柜子数，我们只需要控制好箱子里存储的柜子地址，就可以在不移动任何箱子的情况下增删柜子数:


3
5
gun
柜子1
柜子3
柜子5


删除柜子3
5
5
gun
柜子1
柜子3
柜子5


2
5
gun
柜子1
柜子3
柜子5
3
插入柜子2

第 2 节 链表的原理精讲

链表是线性表的链式存储方式，逻辑上相邻的数据在计算机内的存储位置不必须相邻，那么怎么表示逻辑上的相邻关系呢？可以给每个元素附加一个指针域，指向下一个元素的存储位置。如图所示：

从图中可以看出,每个结点包含两个域：数据域和指针域，指针域存储下一个结点的地址，因此指针指向的类型也是结点类型

链表的核心要素：

- 每个节点由数据域和指针域组成
- 指针域指向下一个节点的内存地址

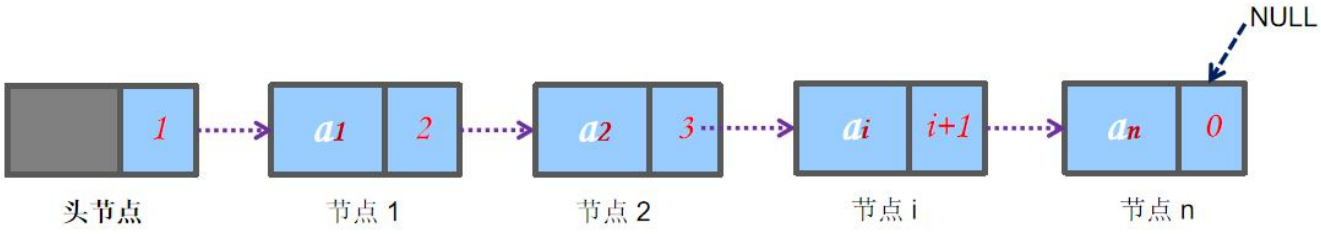
其结构体定义：

```
Typedef struct LinkNode{
      ElemType data;
     struct LinkNode *next;
}LinkList, LinkNode;
```

第 3 节 链表的算法实现

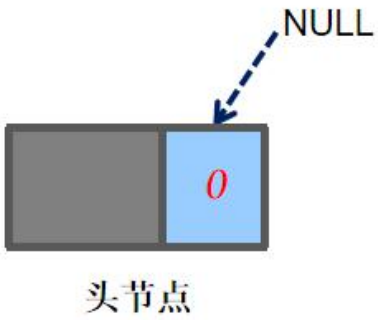
单链表的概念

链表的节点均单向指向下一个节点，形成一条单向访问的数据链



单链表的初始化

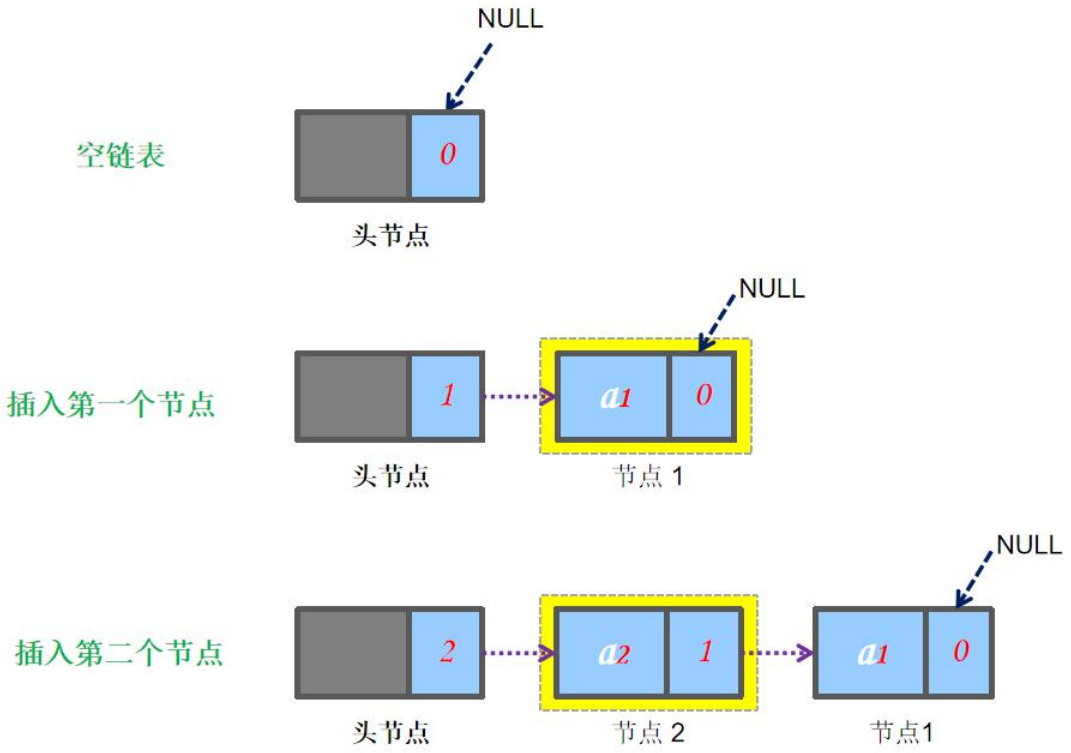
```
typedef struct _LinkNode {
    int data; //结点的数据域
    struct _LinkNode *next; //结点的指针域
}LinkNode, LinkList; //链表节点、链表

bool InitList(LinkList* &L)//构造一个空的单链表 L
{
    L=new LinkNode; //生成新结点作为头结点，用头指针 L 指向头结点
    if(!L)return false; //生成结点失败
    L->next=NULL; //头结点的指针域置空
    return true;
}
```



单链表增加元素

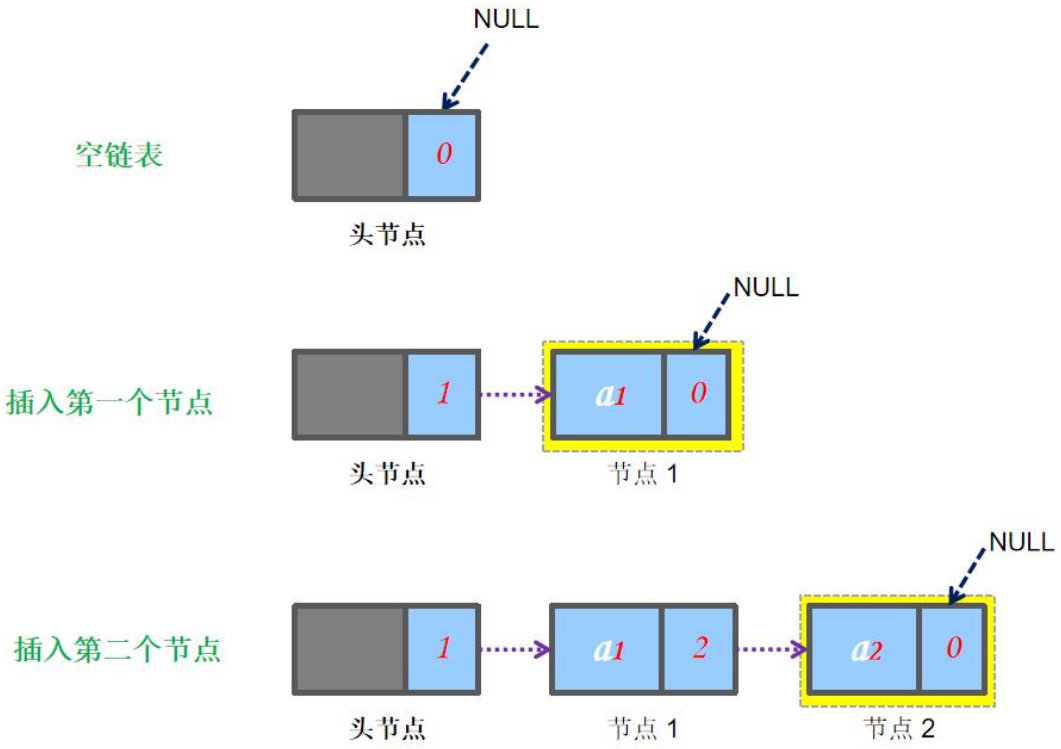
前插法

```
//前插法
bool  ListInsert_front(LinkList* &L, LinkNode * node){
    if(!L || !node ) return false;

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



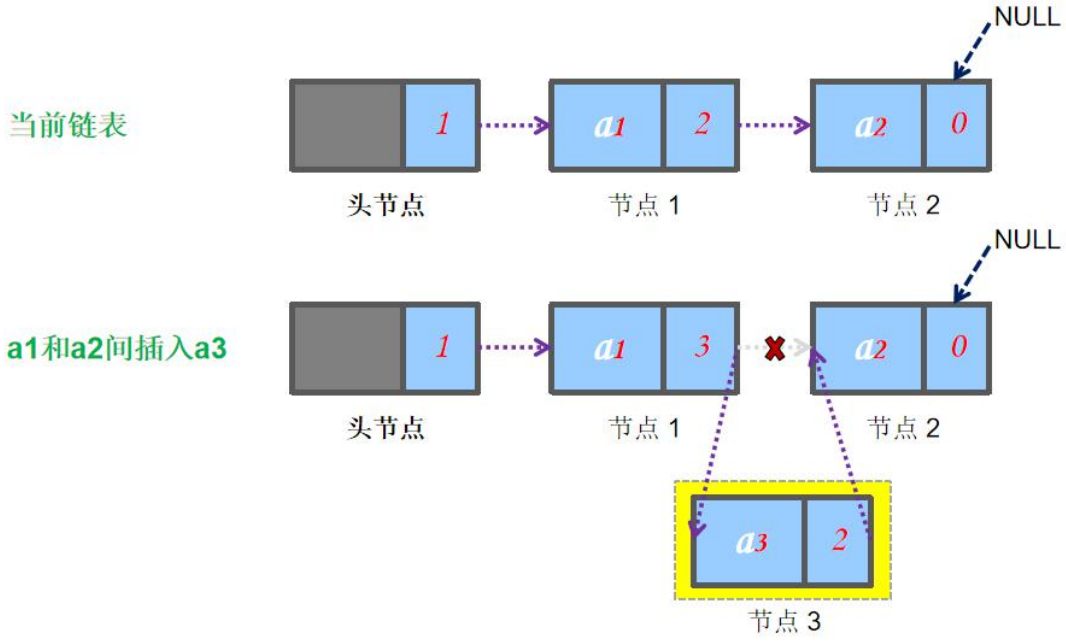
尾插法

```
//尾插法
bool  ListInsert_back(LinkList* &L, LinkNode *node){
    LinkNode *last = NULL;

    if(!L || !node ) return false;

    //找到最后一个节点
    last = L;
    while(last->next) last=last->next;

    //新的节点链接到最尾部
    node->next = NULL;
    last->next = node;
    return true;
}
```



任意位置插入

```
//任意位置插法
bool LinkInsert(LinkList* &L, int i, int &e)//单链表的插入
{
    //在带头结点的单链表 L 中第 i 个位置插入值为 e 的新结点
    int j;
    LinkList *p, *s;

    p=L;
    j=0;
    while (p&&j<i-1) //查找第 i-1 个结点，p 指向该结点
    {
        p=p->next;
        j++;
    }

    if (!p || j>i-1){//i＞n+1 或者 i＜1
        return false;
    }

    s=new LinkNode; //生成新结点
    s->data=e; //将新结点的数据域置为 e
    s->next=p->next; //将新结点的指针域指向结点 ai
    p->next=s; //将结点 p 的指针域指向结点 s
    return true;
}
```



单链表的遍历

```
void LinkPrint(LinkList* &L) //单链表的输出
{
    LinkNode* p;
    p=L->next;

    while (p)
    {
        cout <<p->data <<"\t";
        p=p->next;
    }

    cout<<endl;
}
```

单链表获取元素

```
bool Link_GetElem(LinkList* &L, int i, int &e)//单链表的取值
{
    //在带头结点的单链表 L 中查找第 i 个元素
    //用 e 记录 L 中第 i 个数据元素的值
    int j;
    LinkList* p;

    p=L->next;//p 指向第一个结点,
    j=1; //j 为计数器

    while (j<i && p) //顺链域向后扫描，直到 p 指向第 i 个元素或 p 为空
    {
        p=p->next; //p 指向下一个结点
        j++; //计数器 j 相应加 1
    }

    if (!p || j>i){
        return false; //i 值不合法 i>n 或 i<=0
    }

    e=p->data;          //取第 i 个结点的数据域
    return true;
}
```

单链表查找元素

```
bool Link_FindElem(LinkList *L, int e) //按值查找
{
    //在带头结点的单链表 L 中查找值为 e 的元素
    LinkList *p;

    p=L->next;

    while (p && p->data!=e){//顺链域向后扫描，直到 p 为空或 p 所指结点的
数据域等于 e
        p=p->next;          //p 指向下一个结点
    }

    if(!p)return false;     //查找失败 p 为 NULL

    return true;
}
```

单链表删除元素

```
bool LinkDelete(LinkList* &L, int i) //单链表的删除
{
    //在带头结点的单链表 L 中，删除第 i 个位置
    LinkList *p, *q;
    int j;
    p=L;
    j=0;

    while((p->next)&&(j<i-1)) //查找第 i-1 个结点，p 指向该结点
    {
        p=p->next;
        j++;
    }

    if (!(p->next)||(j>i-1))//当 i>n 或 i<1 时，删除位置不合理
        return false;

    q=p->next; //临时保存被删结点的地址以备释放空间
```

```
    p->next=q->next; //改变删除结点前驱结点的指针域
    delete q; //释放被删除结点的空间
    return true;
}
```

单链表销毁

```
void LinkDestroy(LinkList* &L) //单链表的销毁
{

    //定义临时节点 p 指向头节点
    LinkList *p = L;
    cout<<"销毁链表!"<<endl;

    while(p)
    {
        L=L->next; //L 指向下一个节点
        cout<<"删除元素: "<<p->data<<endl;
        delete p;  //删除当前节点
        p=L;       //p 移向下一个节点
    }
}
```

完整代码实现:

```
#include<iostream>
#include<string>
#include<stdlib.h>

using namespace std;

typedef struct _LinkNode {
    int data; //结点的数据域
    struct _LinkNode *next; //结点的指针域
}LinkNode, LinkList; //LinkList 为指向结构体 LNode 的指针类型


bool InitList(LinkList* &L){
    L = new LinkNode;

    if(!L) return false;//生成节点失败

    L->next = NULL;
    L->data = -1;
```

```
    return true;
}

//前插法
bool ListInsert_front(LinkList* &L, LinkNode *node){
    if(!L || !node) return false;

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

//尾插法
bool ListInsert_back(LinkList* &L, LinkNode *node){
    LinkNode *last = NULL;

    if(!L || !node) return false;

    last = L;

    while(last->next) last = last->next;

    node->next = NULL;
    last->next = node;
    return true;
}

//指定位置插入
bool LinkInsert(LinkList* &L, int i, int &e){

    if(!L) return false;

    int j =0;
    LinkList *p, *s;

    p = L;

    while(p && j<i-1){//查找位置为 i-1 的结点，p 指向该结点
        p = p->next;
        j++;
    }

    if(!p || j>i-1){
        return false;
    }

    s=new LinkNode;//生成新节点
```

```
    s->data = e;
    s->next = p->next;
    p->next = s;

    return true;
}



void LinkPrint(LinkList* &L){
    LinkNode *p = NULL;

    if(!L){
        cout<<"链表为空."<<endl;
        return ;
    }

    p = L->next;

    while(p){
        cout<<p->data<<"\t";
        p = p->next;
    }
    cout<<endl;
}

bool Link_GetElem(LinkList* &L, int i, int &e)//单链表的取值
{
    //在带头结点的单链表 L 中查找第 i 个元素
    //用 e 记录 L 中第 i 个数据元素的值

    int index;
    LinkList *p;

    if(!L || !L->next) return false;

    p = L->next;
    index = 1;

    while(p && index<i){//顺链表向后扫描，直到 p 指向第 i 个元素或 p 为空
        p = p->next;     //p 指向下一个结点
        index++;         //计数器 index 相应加 1
    }

    if(!p || index>i){
        return false;    //i 值不合法,i＞n 或 i<=0
    }
```

```
    e=p->data;
    return true;
}


bool Link_FindElem(LinkList *L, int e, int &index) //按值查找
{
    //在带头结点的单链表 L 中查找值为 e 的元素

    LinkList *p;
    p=L->next;
    index = 1;
    if(!L || !L->next){
        index = 0;
        return false;
    }

    while(p && p->data!=e){
        p=p->next;
        index ++;
    }

    if(!p){
        index = 0;
        return false;//查无此值
    }

    return true;
}

bool LinkDelete(LinkList* &L, int i) //单链表的删除
{
    LinkList *p, *q;

    int index = 0;
    p=L;

    if(!L || !L->next){
        return  false;
    }

    while((p->next) && (index<i-1)){
        p = p->next;
        index++;
    }
```

```
    if(!p->next || (index>i-1) ){ //当 i>n 或 i<1 时，删除位置不合理
        return false;
    }

    q = p->next;       //临时保存被删结点的地址以备释放空间
    p->next = q->next;//改变删除结点前驱结点的指针域
    delete q;          //释放被删除结点的空间

    return true;
}

void LinkDestroy(LinkList* &L) //单链表的销毁
{
    //定义临时节点 p 指向头节点
    LinkList *p = L;
    cout<<"销毁链表!"<<endl;

    while(p){
        L=L->next;//L 指向下一个节点
        cout<<"删除元素: "<<p->data<<endl;
        delete p; //删除当前节点
        p = L;    //p 移向下一个节点
    }
}


int main(void){
    LinkList *L = NULL;
    LinkNode *s = NULL;

    //1. 初始化一个空的链表
    InitList(L);

    //2. 使用前插法插入数据
    /*int n;

    cout<<"前插法创建单链表"<<endl;
    std::cout<<"请输入元素个数 n: ";
    cin>>n;
    cout<<"\n 请依次输入 n 个元素: " <<endl;

    while(n>0){
    s = new LinkNode; //生成新节点 s

    cin>>s->data;
    ListInsert_front(L, s);
    n--;
```

```
    }
    */
    //3. 使用尾插法插入数据
    /*int n;

    cout<<"尾插法创建单链表"<<endl;
    std::cout<<"请输入元素个数 n: ";
    cin>>n;
    cout<<"\n 请依次输入 n 个元素: " <<endl;

    while(n>0){
        s = new LinkNode; //生成新节点 s

        cin>>s->data;
        ListInsert_back(L, s);
        n--;
    }

    //4. 单链表的输出
    LinkPrint(L);
    */
    //5. 任意位置插入元素
    for(int j=0; j<3; j++){
        int i, x;
        cout << "请输入插入的位置和元素（用空格隔开）:";
        cin >> i;
        cin >> x;

        if(LinkInsert(L, i, x)){
            cout << "插入成功.\n\n";
        }else{
            cout << "插入失败!\n\n";
        }

        LinkPrint(L);
    }

    //6. 单链表根据位置获取元素
    int element = 0;
    if(Link_GetElem(L, 2, element)){
        cout<<"获取第二个元素成功, 值: "<<element<<endl;
    }else {
        cout<<"获取第二个元素失败!"<<endl;
    }

    //7. 单链表根据值查询元素所在的位置
    int index=0;
```

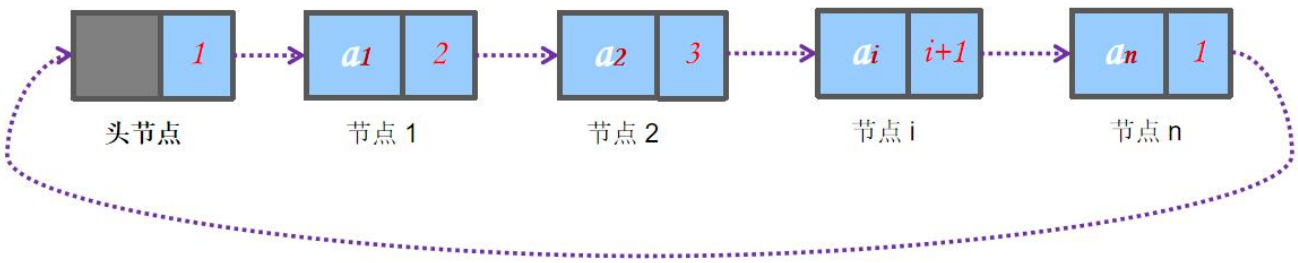
```
    if(Link_FindElem(L, 10, index)){
        cout<<"查找元素 10 存在,所在位置: "<<index<<endl;
    }else {
        cout<<"不存在元素 10."<<endl;
    }

    //8. 单链表删除元素
    if(LinkDelete(L, 2)){
        cout<<"删除第 2 个元素成功!"<<endl;
        LinkPrint(L);
    }else {
        cout<<"删除第 2 个元素失败!"<<endl;
    }

    //9. 销毁单链表
    LinkDestroy(L);
    system("pause");
    return 0;
}
```

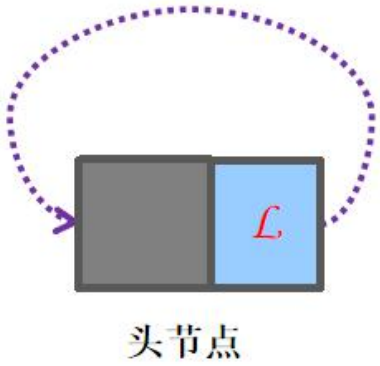
第 4 节　循环链表的算法实现

Joseph 问题

有 10 个小朋友按编号顺序 1，2，。。。。，10 顺时针方向围成一圈。从 1 号开始顺时针方向 1，2，。。。。，9 报数，凡报数 9 者出列（显然，第一个出圈为编号 9 者）。

最后一个出圈者的编号是多少？第 5 个出圈者的编号是多少？





空的循环链表

完整代码实现:

```
#include<iostream>
#include<string>
#include<stdlib.h>

using namespace std;

typedef struct _LinkNode {
    int data; //结点的数据域
    struct _LinkNode *next; //结点的指针域
}LinkNode, LinkList; //LinkList 为指向结构体 LNode 的指针类型

void LinkPrint(LinkList *L);

bool InitList(LinkList* &L)//构造一个空的循环链表 L
{
    L=new LinkNode; //生成新结点作为头结点，用头指针 L 指向头结点

    if(!L)return false; //生成结点失败

    L->next=L; //头结点的指针域指向自己
    L->data = -1;
    return true;
}

//尾插法
bool  ListInsert_back(LinkList* &L, LinkNode * node){
    LinkNode *last = NULL;

    if(!L || !node ) return false;
```

```
    //找到最后一个节点
    last = L;
    while(last->next!=L) last=last->next;

    //新的节点链接到最尾部
    node->next = L;
    last->next = node;
    return true;
}



bool Joseph(LinkList* &L, int interval)
{

    //在带头结点的循环链表 L 中，每个 interval 个间隔循环删除节点
    LinkList *p, *q;
    int j = 0, i = 0;
    int times = 0, num = 0;
    p=L;

    if(!L || p->next == L) {
        cout<<"链表为空!"<<endl;
        return false;
    }

    if(interval<1){
        cout<<"报数淘汰口令不能小于 1!"<<endl;
        return false;
    }

    do{
    i += interval;
    while((p->next)) //查找第 i 个结点，p 指向该结点的上一个节点
    {
        if(p->next!=L) j++;
        if(j>=i) break;
        p=p->next;
    }

    times++;
    /*if (!(p->next)||(j>i))//当 i>n 或 i<1 时，删除位置不合理
        return false;*/

    q=p->next; //临时保存被删结点的地址以备释放空间
    num = q->data;
```

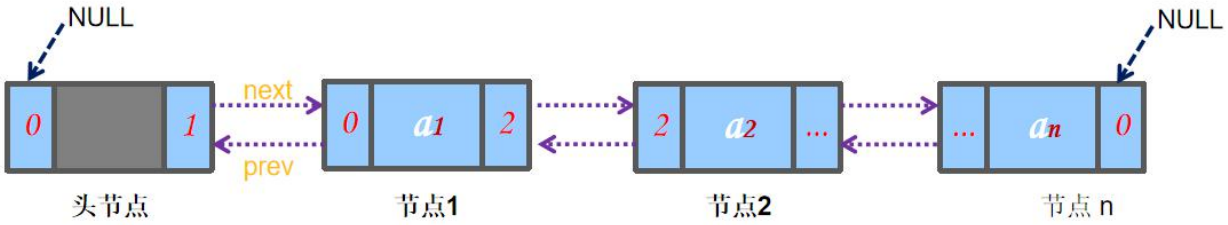
```
    if(times==5) cout<<"第 5 个出圈的编号是："<<num<<endl;
    printf("cur: %d   last: %d  next:%d\n",q->data, p->data,
q->next->data);
    p->next=q->next; //改变删除结点前驱结点的指针域

    delete q; //释放被删除结点的空间
    LinkPrint(L);

    }while(L->next!=L);//链表不为空，继续报数

    cout<<"最后一个出圈的编号是："<<num<<endl;
    return true;
}



void LinkPrint(LinkList *L) //循环链表的输出
{
    LinkList *p;

    if(!L || L==L->next){
        cout<<"链表为空!"<<endl;
        return ;
    }

    p=L->next;

    while (p!=L)
    {
        cout <<p->data <<"\t";
        p=p->next;
    }

    cout<<endl;
}

int main(){
    int i, x;
    LinkList *L;
    LinkNode *s;

    //1. 初始化一个空的循环链表
    if (InitList(L)){
        cout << "初始化一个空的循环链表!\n";
    }

    //2. 创建循环链表（尾插法）
```

```
    std::cout <<"尾插法创建循环链表, 插入 10 个元素..." <<endl;
    i = 0;

    while((++i)<=10)
    {
        s=new LinkNode;//生成新结点
        s->data=i; //输入元素值赋给新结点的数据域
        s->next=NULL;
        if(ListInsert_back(L, s)){
            cout<<"插入成功!"<<endl;
        }else {
            cout<<"插入失败!"<<endl;
        }
    }

    cout << "尾插法创建循环链表输出结果:\n";
    LinkPrint(L);

    //3. 解答约瑟夫问题
    Joseph(L, 9);
    system("pause");
    return 0;
}
```

第 5 节　双向链表的算法实现

单链表中每个结点除了存储自身数据之后，还存储了下一个结点的地址，因此可以轻松访问下一个结点，以及后面的后继结点，但是如果想访问前面的结点就不行了，再也回不去了。例如删除结点 p 时，要先找到它的前一个结点 q，然后才能删掉 p 结点，单向链表只能往后走，不能向前走。如果需要向前走，怎么办呢?

可以在单链表的基础上给每个元素附加两个指针域，一个存储前一个元素的地址，一个存储下一个元素的地址。这种链表称为双向链表.

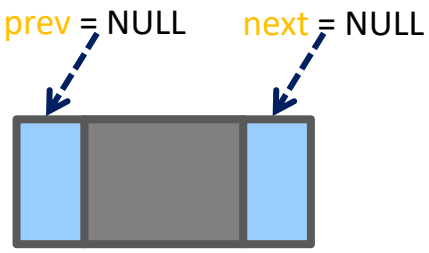


其结构体定义：

```
typedef struct _LinkNode {
    int data; //结点的数据域
    struct _LinkNode *next; //下一个节点的指针域
    struct _LinkNode *prev; //上一个结点的指针域
}LinkNode, LinkList; //LinkList 为指向结构体 LNode 的指针类型
```

双向链表的初始化

```
typedef struct _DoubleLinkNode {
    int data; //结点的数据域
    struct _DoubleLinkNode *next; //下一个节点的指针域
    struct _DoubleLinkNode *prev; //上一个结点的指针域
}DbLinkNode, DbLinkList; //LinkList 为指向结构体 LNode 的指针类型

bool DbInit_List(DbLinkList* &L)//构造一个空的双向链表 L
{
    L=new DbLinkNode; //生成新结点作为头结点，用头指针 L 指向头结点
    if(!L)return false; //生成结点失败
    L->next=NULL; //头结点的 next 指针域置空
    L->prev=NULL; //头结点的指针域置空
    L->data = -1;
    return true;
}
```



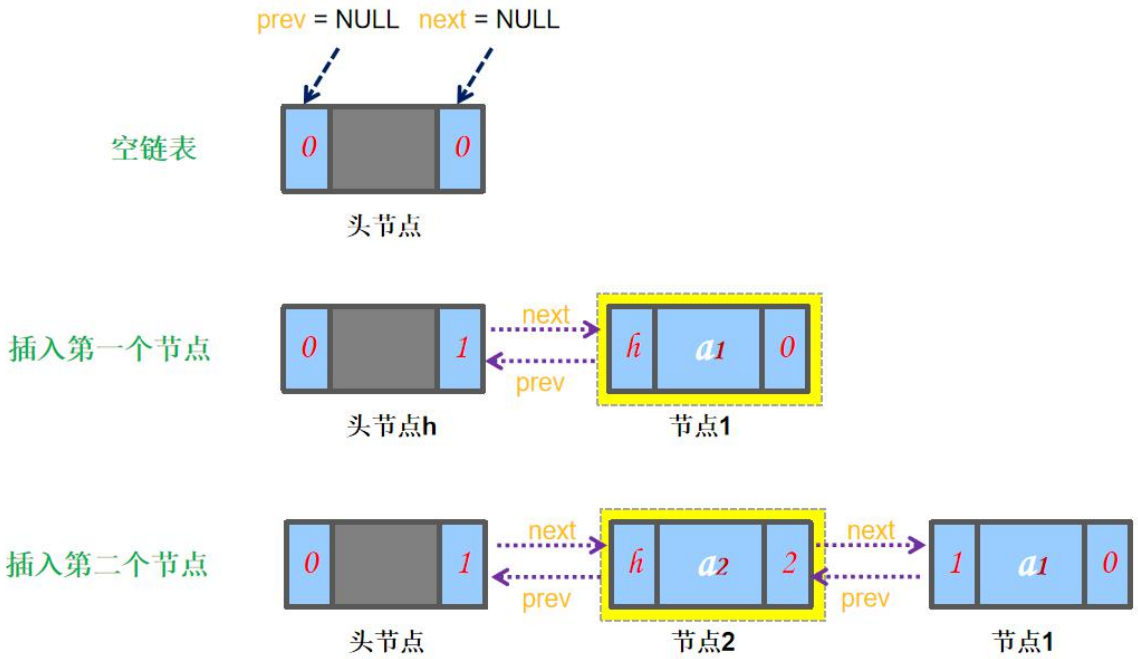
头节点

双向链表增加元素

前插法

```
//前插法
bool DbListInsert_front(DbLinkList* &L, DbLinkNode *node){
    if(!L || !node) return false;


    //1.只有头节点
    if(L->next==NULL){
```

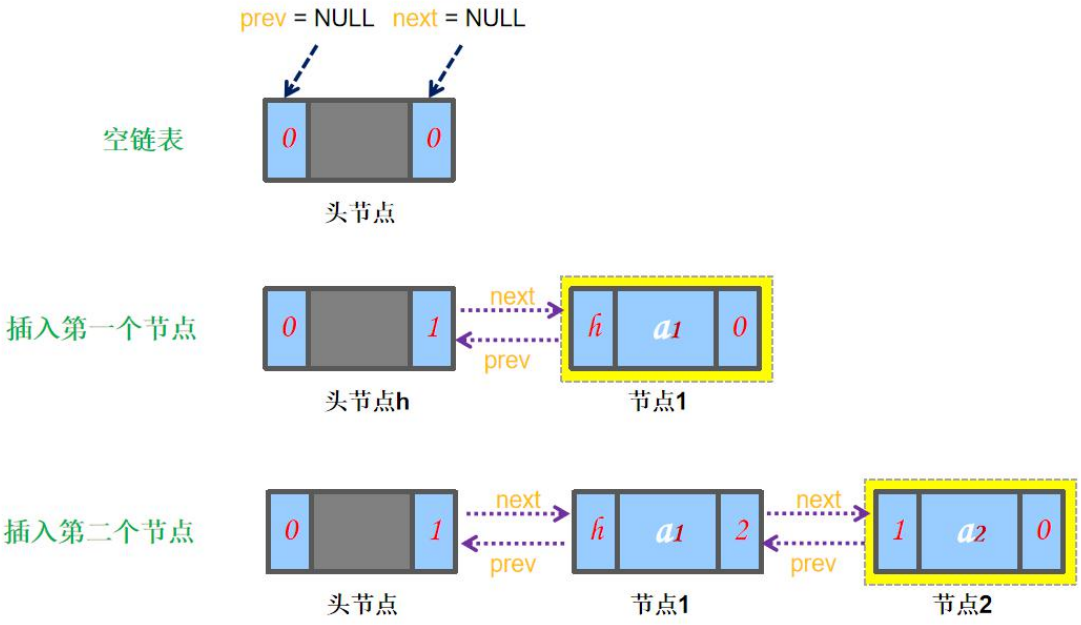
```
        node->next=NULL;
        node->prev=L;           //新节点 prev 指针指向头节点
        L->next=node;           //头节点 next 指针指向新节点
    }else {
        L->next->prev=node;     //第二个节点的 prev 指向新节点
        node->next = L->next; //新节点 next 指针指向第二个节点
        node->prev=L;           //新节点 prev 指针指向头节点
        L->next=node;           //头节点 next 指针指向新节点，完成插入
    }

    return true;
}
```



尾插法

```
//尾插法
bool DbListInsert_back(DbLinkList* &L, DbLinkNode *node){
    DbLinkNode *last = NULL;

    if(!L || !node) return false;

    last = L;

    while(last->next) last = last->next;

    node->next = NULL;
    last->next = node;
```

```
    node->prev = last;
    return true;
}
```



任意位置插入

```
//指定位置插入
bool DbLink_Insert(DbLinkList* &L, int i, int &e){

    if(!L||!L->next) return false;

    if(i<1) return false;

    int j =0;
    DbLinkList *p, *s;

    p = L;

    while(p && j<i){//查找位置为 i 的结点，p 指向该结点
        p = p->next;
        j++;
    }

    if(!p || j!=i){
        cout<<"不存在节点："<<i<<endl;
        return false;
    }

    cout<<"p: "<<p<<endl;

    s=new DbLinkNode;//生成新节点
    s->data = e;

    s->next = p;
    s->prev = p->prev;

    p->prev->next = s;
    p->prev = s;
    return true;
}
```

双向链表的遍历

```
//双向链表的遍历输出
void DbLink_Print(DbLinkList* &L ){
    DbLinkNode *p = NULL;

    if(!L){
        cout<<"链表为空."<<endl;
        return ;
    }

    p = L;

    while(p->next){
        cout<<p->next->data<<"\t";
        p = p->next;
    }

    //逆向打印
    cout<<endl<<"逆向打印"<<endl;
    while(p){
        cout<<p->data<<"\t";
        p = p->prev;
    }

    cout<<endl;
}
```

双向链表获取元素

```
bool DbLink_GetElem(DbLinkList* &L, int i, int &e)//双向链表的取值
{
    //在带头结点的双向链表 L 中查找第 i 个元素
    //用 e 记录 L 中第 i 个数据元素的值

    int index;
    DbLinkList *p;

    if(!L || !L->next) return false;

    p = L->next;
    index = 1;

    while(p && index<i){//顺链表向后扫描，直到 p 指向第 i 个元素或 p 为空
```

```
        p = p->next;     //p 指向下一个结点
        index++;         //计数器 index 相应加 1
    }

    if(!p || index>i){
        return false;    //i 值不合法,i>n 或 i<=0
    }

    e=p->data;
    return true;
}
```

双向链表删除元素

```
//任意位置删除
bool DbLink_Delete(DbLinkList* &L, int i) //双向链表的删除
{
    DbLinkList *p;

    int index = 0;

    if(!L || !L->next){
        cout<<"双向链表为空!"<<endl;
        return  false;
    }

    if(i<1) return false; //不能删除头节点

    p=L;
    while(p && index<i){
        p = p->next;
        index++;
    }

    if(!p){ //当节点不存在时，返回失败
        return false;
    }

    p->prev->next=p->next;  //改变删除结点前驱结点的 next 指针域
   if(p->next){
        p->next->prev = p->prev; //改变删除节点后继节点的 prev 指针域
    }
    delete p;          //释放被删除结点的空间

    return true;
```

```
}
```

双向链表销毁

```
void DbLink_Destroy(DbLinkList* &L) //双向链表的销毁
{
    //定义临时节点 p 指向头节点
    DbLinkList *p = L;
    cout<<"销毁链表!"<<endl;

    while(p){
        L=L->next;//L 指向下一个节点
        cout<<"删除元素: "<<p->data<<endl;
        delete p; //删除当前节点
        p = L;   //p 移向下一个节点
    }
}
```

完整代码实现:

```
#include<iostream>
#include<string>
#include<stdlib.h>

using namespace std;

typedef struct _DoubleLinkNode {
    int data; //结点的数据域
    struct _DoubleLinkNode *next; //下一个节点的指针域
    struct _DoubleLinkNode *prev; //上一个结点的指针域
}DbLinkNode, DbLinkList; //LinkList 为指向结构体 LNode 的指针类型

bool DbList_Init(DbLinkList* &L)//构造一个空的双向链表 L
{
    L=new DbLinkNode; //生成新结点作为头结点，用头指针 L 指向头结点
    if(!L)return false; //生成结点失败

    L->next=NULL; //头结点的 next 指针域置空
    L->prev=NULL; //头结点的 prev 指针域置空
    L->data = -1;
```

```
    return true;
}

//前插法
bool DbListInsert_front(DbLinkList* &L, DbLinkNode *node){
    if(!L || !node) return false;


    //1.只有头节点
    if(L->next==NULL){
        node->next=NULL;
        node->prev=L;          //新节点 prev 指针指向头节点
        L->next=node;          //头节点 next 指针指向新节点
    }else {
        L->next->prev=node;    //第二个节点的 prev 指向新节点
        node->next = L->next; //新节点 next 指针指向第二个节点
        node->prev=L;          //新节点 prev 指针指向头节点
        L->next=node;          //头节点 next 指针指向新节点，完成插入
    }

    return true;
}

//尾插法
bool DbListInsert_back(DbLinkList* &L, DbLinkNode *node){
    DbLinkNode *last = NULL;

    if(!L || !node) return false;

    last = L;

    while(last->next) last = last->next;

    node->next = NULL;
    last->next = node;
    node->prev = last;
    return true;
}

//指定位置插入
bool DbLink_Insert(DbLinkList* &L, int i, int &e){

    if(!L||!L->next) return false;

    if(i<1) return false;

    int j =0;
```

```
    DbLinkList *p, *s;

    p = L;

    while(p && j<i){//查找位置为 i 的结点，p 指向该结点
        p = p->next;
        j++;
    }

    if(!p || j!=i){
        cout<<"不存在节点："<<i<<endl;
        return false;
    }

    cout<<"p: "<<p<<endl;

    s=new DbLinkNode;//生成新节点
    s->data = e;

    s->next = p;
    s->prev = p->prev;

    p->prev->next = s;
    p->prev = s;
    return true;
}



void DbLink_Print(DbLinkList* &L ){
    DbLinkNode *p = NULL;

    if(!L){
        cout<<"链表为空."<<endl;
        return ;
    }

    p = L;

    while(p->next){
        cout<<p->next->data<<"\t";
        p = p->next;
    }

    //逆向打印
    cout<<endl<<"逆向打印"<<endl;
    while(p){
```

```
        cout<<p->data<<"\t";
        p = p->prev;
    }

    cout<<endl;
}

bool DbLink_GetElem(DbLinkList* &L, int i, int &e)//双向链表的取值
{
    //在带头结点的双向链表 L 中查找第 i 个元素
    //用 e 记录 L 中第 i 个数据元素的值

    int index;
    DbLinkList *p;

    if(!L || !L->next) return false;

    p = L->next;
    index = 1;

    while(p && index<i){//顺链表向后扫描，直到 p 指向第 i 个元素或 p 为空
        p = p->next;    //p 指向下一个结点
        index++;        //计数器 index 相应加 1
    }

    if(!p || index>i){
        return false;   //i 值不合法,i>n 或 i<=0
    }

    e=p->data;
    return true;
}



bool DbLink_Delete(DbLinkList* &L, int i) //双向链表的删除
{
    DbLinkList *p;

    int index = 0;

    if(!L || !L->next){
        cout<<"双向链表为空!"<<endl;
        return  false;
    }

    if(i<1) return false; //不能删除头节点
```

```
    p=L;
    while(p && index<i){
        p = p->next;
        index++;
    }

    if(!p){ //当节点不存在时，返回失败
        return false;
    }

    p->prev->next=p->next;  //改变删除结点前驱结点的 next 指针域
    p->next->prev = p->prev;       //改变删除节点后继节点的 prev 指针域

    delete p;          //释放被删除结点的空间

    return true;
}

void DbLink_Destroy(DbLinkList* &L) //双向链表的销毁
{
    //定义临时节点 p 指向头节点
    DbLinkList *p = L;
    cout<<"销毁链表!"<<endl;

    while(p){
        L=L->next;//L 指向下一个节点
        cout<<"删除元素: "<<p->data<<endl;
        delete p; //删除当前节点
        p = L;   //p 移向下一个节点
    }
}


int main(void){
    DbLinkList *L = NULL;
    DbLinkNode *s = NULL;

    //1. 初始化一个空的双向链表
    DbList_Init(L);

    //2. 使用前插法插入数据
    int n;

    cout<<"前插法创建双向链表"<<endl;
    std::cout<<"请输入元素个数 n: ";
    cin>>n;
```

```
    cout<<"\n 请依次输入 n 个元素： " <<endl;

    while(n>0){
    s = new DbLinkNode; //生成新节点 s

    cin>>s->data;
    DbListInsert_front(L, s);
    n--;
    }

    //3. 使用尾插法插入数据
    cout<<"尾插法创建双向链表"<<endl;
    std::cout<<"请输入元素个数 n： ";
    cin>>n;
    cout<<"\n 请依次输入 n 个元素： " <<endl;

    while(n>0){
        s = new DbLinkNode; //生成新节点 s

        cin>>s->data;
        DbListInsert_back(L, s);
        n--;
    }

    //4. 双向链表的输出
    DbLink_Print(L);

    //5. 任意位置插入元素
    for(int j=0; j<3; j++){
        int i, x;
        cout << "请输入插入的位置和元素（用空格隔开）:";
        cin >> i;
        cin >> x;

        if(DbLink_Insert(L, i, x)){
            cout << "插入成功.\n\n";
        }else{
            cout << "插入失败!\n\n";
        }

        DbLink_Print(L);
    }

    //6. 双向链表根据位置获取元素
    int element = 0;
    if(DbLink_GetElem(L, 2, element)){
        cout<<"获取第二个元素成功, 值： "<<element<<endl;
```

```
    }else {
        cout<<"获取第二个元素失败!"<<endl;
    }

    //7. 双向链表删除元素
    if(DbLink_Delete(L, 2)){
        cout<<"删除第 2 个元素成功!"<<endl;
        DbLink_Print(L);
    }else {
        cout<<"删除第 2 个元素失败!"<<endl;
    }

    if(DbLink_Delete(L, 1)){
        cout<<"删除第 1 个元素成功!"<<endl;
        DbLink_Print(L);
    }else {
        cout<<"删除第 1 个元素失败!"<<endl;
    }

    //8. 销毁双向链表
    DbLink_Destroy(L);
    system("pause");
    return 0;
}
```

第 6 节 链表的企业级应用案例

6.1 Linux 内核“共享”双向链表

在 linux 内核中，有大量的数据结构需要用到双向链表，例如进程、文件、模块、页面等。

若采用双向链表的传统实现方式，需要为这些数据结构维护各自的链表，并且为每个链表都要设计插入、删除等操作函数。因为用来维持链表的 next 和 prev 指针指向对应类型的对象，因此一种数据结构的链表操作函数不能用于操作其它数据结构的链表。

比如，我们需要分别定义星星和 web 服务器超时的链表结构 ：

一． web 服务器超时的链表结构

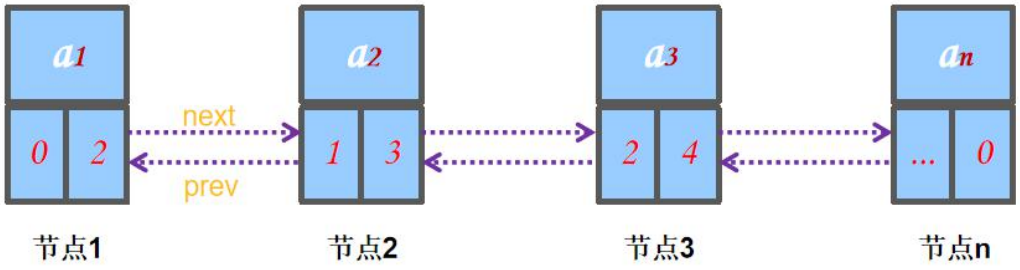
```
typedef struct {
    int fd ;
    time_t timeout; // 使用超时时刻的时间戳表示
}ConnTimeout;

struct Link_Node{
    ConnTimeout   conn;
    struct Link_Node *next;
}
```

二．璀璨星空的链表结构

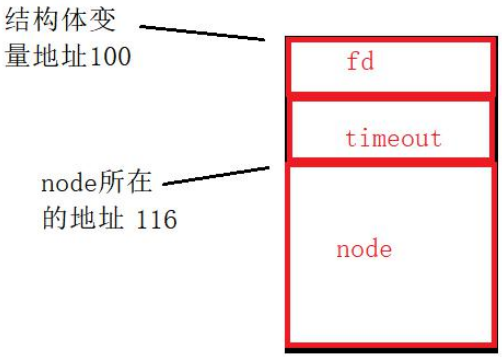
```
typedef struct {
    int x;              //星星的 x 坐标
    int y;              //星星的 y 坐标
    enum STATUS stat;   //状态
    unsigned radius;    //星星的半径
    int step;           //每次跳跃的间隔
    int color;          //星星的颜色
}STAR;


struct Link_Node{
    STAR   star;
    struct Link_Node *next;
```

```
}
```

有没有一种方式，可以让多个链表共享同一套链表的操作？请看下图：




```
typedef struct _DoubleLinkNode {
    struct _DoubleLinkNode *next; //下一个节点的指针域
    struct _DoubleLinkNode *prev; //上一个结点的指针域
}DbLinkNode;

typedef struct {
    int fd ;
    time_t timeout;  // 使用超时时刻的时间戳表示
    DbLinkNode node; // 双向链表节点“挂件”
}ConnTimeout;
```




```
typedef struct {
    int x;              //星星的 x 坐标
    int y;              //星星的 y 坐标
    enum STATUS stat;   //状态
    unsigned radius;    //星星的半径
    int step;           //每次跳跃的间隔
    int color;          //星星的颜色
     DbLinkNode node; // 双向链表节点“挂件”
}STAR;
```

实现要点:

使用 offsetof 可以根据链表节点在结构体中的地址逆推出结构体变量的位置.

如:

```
typedef struct {
    int fd ;
    time_t timeout;  // 使用超时时刻的时间戳表示
    DbLinkNode node; // 双向链表节点“挂件”
}ConnTimeout;
```

```
//通过节点访问到节点承载的数据
    ConnTimeout  *ct = new ConnTimeout;
    DbLinkNode   *p = &(ct->node);

    cout<<"请输入超时节点对应的 fd: ";
     cin>>ct->fd;
```

```
    cout<<"\n 通过链表中的节点访问节点上承载的数据："<<endl;
    int offset = offsetof(ConnTimeout, node);

    ConnTimeout *tmp = (ConnTimeout *)((size_t)p-offset);
    printf("offset: %d\n", offset);
    printf("通过链表节点 node 访问到的数据：%d\n", tmp->fd);
```

源码实现：

```
#include<iostream>
#include<string>
#include<stdlib.h>

using namespace std;

typedef struct _DoubleLinkNode {
    //int data; //结点的数据域
    struct _DoubleLinkNode *next; //下一个节点的指针域
    struct _DoubleLinkNode *prev; //上一个结点的指针域
}DbLinkNode, DbLinkList; //LinkList 为指向结构体 LNode 的指针类型

typedef struct {
    int fd ;
    time_t timeout;  // 使用超时时刻的时间戳表示
    DbLinkNode node; // 双向链表节点“挂件”
}ConnTimeout;

typedef struct {
    int x;              //星星的 x 坐标
    int y;              //星星的 y 坐标
    enum STATUS stat;   //状态
    unsigned radius;    //星星的半径
    int step;           //每次跳跃的间隔
    int color;          //星星的颜色
    DbLinkNode node; // 双向链表节点“挂件”
}STAR;

bool DbList_Init(DbLinkList &L)//构造一个空的双向链表 L
{
    L.next=NULL; //头结点的 next 指针域置空
    L.prev=NULL; //头结点的 prev 指针域置空
    return true;
}
```

```
//尾插法
bool DbListInsert_back(DbLinkList &L, DbLinkNode &node){
    DbLinkNode *last = NULL;

    last = &L;

    while(last->next) last = last->next;

    node.next = NULL;
    last->next = &node;
    node.prev = last;
    return true;
}


int main(void){
    ConnTimeout *cl = NULL, *s = NULL;
    STAR  *sl = NULL;
    int n = 0;

    //1.初始化一个空的双向链表
    cl = new ConnTimeout;
    cl->fd = -1;

    sl = new STAR;
    sl->x = -1;

    DbList_Init(cl->node);
    DbList_Init(sl->node);

    //2.使用尾插法插入数据
    cout<<"尾插法创建双向链表"<<endl;
    std::cout<<"请输入元素个数 n: ";
    cin>>n;
    cout<<"\n 请依次输入 n 个元素的文件句柄: " <<endl;

    while(n>0){
        s = new ConnTimeout; //生成新节点 s

        cin>>s->fd;
        printf("s  的地址:%p  node: %p\n",s, &(s->node));
        DbListInsert_back(cl->node, s->node);
        n--;
    }

    //3.根据链表节点访问数据
    DbLinkNode  *p = NULL;
```

```
    p = &(cl->node);
    cout<<"遍历连接超时链表中的节点："<<endl;

    while(p){
        int offset = offsetof(ConnTimeout, node);
        ConnTimeout *ct = (ConnTimeout *)((size_t)p-offset);
        cout<<ct->fd<<endl;
        p=p->next;
    }


    //4. 销毁双向链表
    p = &(cl->node);
    cout<<"销毁连接超时链表中的节点："<<endl;
    while(p){
        int offset = offsetof(ConnTimeout, node);

        ConnTimeout *ct = (ConnTimeout *)((size_t)p-offset);
        printf("offset: %u  ct: %p  p:%p\n", offset, ct, p);
        cout<<ct->fd<<endl;
        p = p->next;
        delete ct;
    }

    system("pause");
    return 0;
}
```

项目练习

1. 浪漫星空代码继续优化，将使用顺序表存储运动中的星星状态改为使用单链表存储。