算法基础（四): 链表【数组实现，静态链表】

单链表

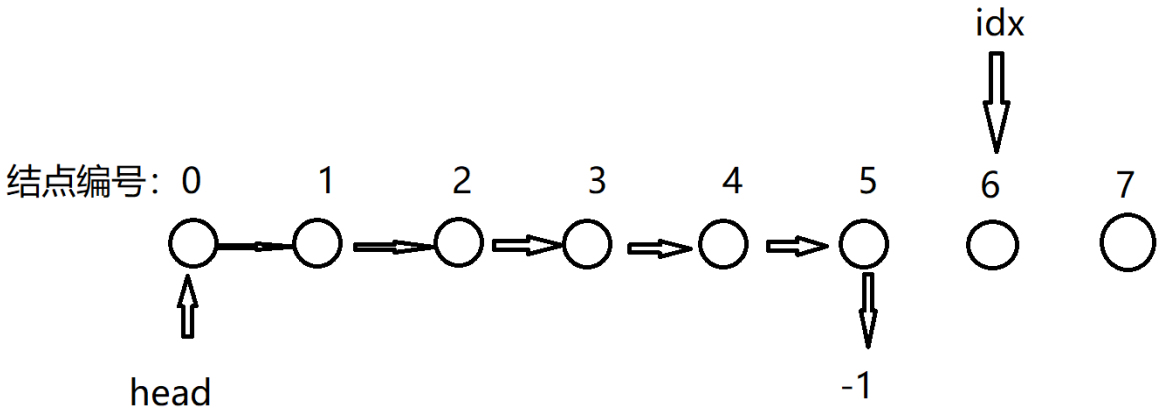
基本思想

先搞明白几个最基本的概念：

1. **头指针**，是指链表中第一个结点的存储位置，在静态链表中`head`表示的是头指针，其值是头结点的下标，也就是头结点的位置，表示指向头结点，没有头结点的情况下指向的是首结点
2. **头结点**，是链表中第一个结点，有时可以不存元素，只为了操作的统一与方便而设置的，这样在链表的头部插入一个元素的时候可以统一看作是在结点后插入，操作方便不用特判
3. **首结点**，当我们没有设置头结点的时候，链表的第一个结点是存储元素的，而我们插入元素的时候需要进行特判，因为在头部插入与普通的插入语句是不同的，在下面的操作中`ne[0]`表示的是首结点，不是头结点

基本结构：

分配了六个结点的情况：



每个结点包括两个内容，结点的值和下一个结点的地址，如果我们用数组来表示的话就是：

1. 结点的值数组`e[N]`
2. 节点的指针数组`ne[N]`

这两个数组用下标连接再一起，下标相同的`e[i]`和`ne[i]`是属于同一个结点，`head`指向首结点，尾结点指向`-1`

假如每个结点的值分别为：`1 3 5 7 9 11`，则对应的数组的值为：

`e[]`：`e[0] = 1, e[1] = 3, e[2] = 5, e[3] = 7, e[4] = 9  e[5] = 11`

`ne[]`：`ne[0] = 1, ne[1] = 2, ne[2] = 3, ne[3] = 4, ne[4] = 5, ne[5] = -1`

这个是最基本的数据结构，具体的插入什么操作的就不说了。

我们使用链表的时候**存储空间是分为已分配和未分配**两个部分的，在上面的数组实现中，存储空间就是`N`，表示我们可以分配`N`个结点，上述的例子表示已经分配了6个结点，剩下下标`6`之后的数组空间并没有使用，我们使用`idx`这个变量来指示当前用到了什么地方，在上面的图中，表示我们用到了下标为6的这个结点，当需要再次分配一个结点的时候，我们直接使用`idx`指向的这个结点，然后`idx++`

需要注意的是我们不需要考虑空闲结点的分配与回收，我们只需要在开始分配一个空间足够大的数组`N`，插入空闲结点的时候需要再分配一个空闲结点，那么我们只需要`idx`一直`++`即可，当删除一个结点的时候，被删除的那个结点`i`，它所对应的数组空间可以直接浪费

这里是用的是静态链表来实现单链表的，在严书里有一节很详细实现了这个单链表，严书里面有静态单链表的创建与回收，那个思想非常妙，他用一个链表将所有的空间链接起来了，下文中会有对比，但是在算法题目里面不需要考虑内存泄漏问题，算法只需要考虑，**如何快速地解决问题**，而不考虑工程上的健壮性

基本操作：

1. 初始化：

```
//初始化
void init(){
    head = -1;
    idx = 0;
}
```

2. 插入操作：

```
//将x插入到首结点，由于没有头节点，这时得特判操作
void add_to_head(int x){
    e[idx] = x;
    ne[idx] = head;
    head = idx;
    idx++;//可分配结点++
}
//普通插入操作，插入到下标是k的结点后面
void add(int k, int x){
    e[idx] = x;
    ne[idx] = ne[k];
    ne[k] = idx;
    idx++;
}
```

3. 删除操作

```
//删掉下标为k的结点的后面的点
void remove(int k){
    ne[k] = ne[ne[k]];
}
```

代码实现

实现一个单链表，链表初始为空，支持三种操作：

1. 向链表头插入一个数；
2. 删除第 k 个插入的数后面的数；
3. 在第 k 个插入的数后插入一个数。

现在要对该链表进行 M 次操作，进行完所有操作后，从头到尾输出整个链表。

注意:题目中第 k 个插入的数并不是指当前链表的第 k 个数。例如操作过程中一共插入了 n 个数，则按照插入的时间顺序，这 n 个数依次为：第 1 个插入的数，第 2 个插入的数，…第 n 个插入的数。

输入格式

第一行包含整数 M，表示操作次数。

接下来 M 行，每行包含一个操作命令，操作命令可能为以下几种：

1. `H x`，表示向链表头插入一个数 x。
2. `D k`，表示删除第 k 个插入的数后面的数（当 k 为 0 时，表示删除头结点）。
3. `I k x`，表示在第 k 个插入的数后面插入一个数 x（此操作中 k 均大于 0）。

输出格式

共一行，将整个链表从头到尾输出。

数据范围

$1 \leq M \leq 100000$
所有操作保证合法。

输入样例：

```
10
H 9
I 1 1
D 1
D 0
H 6
I 3 6
I 4 5
I 4 5
I 3 4
D 6
```

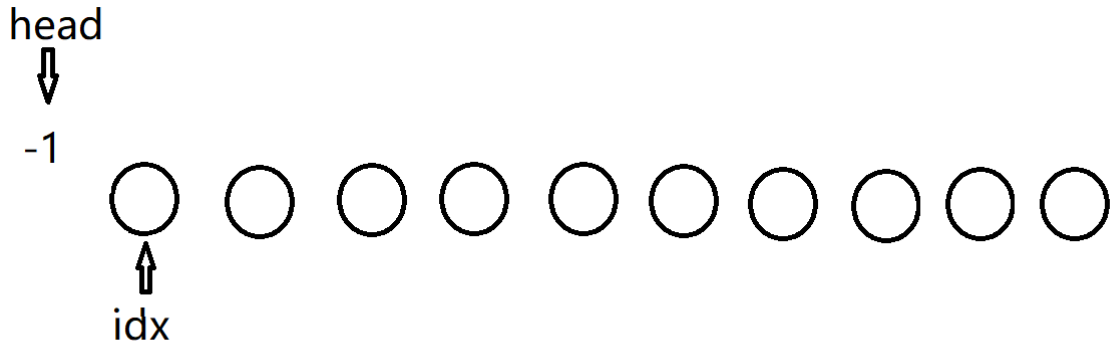
输出样例：

```
6 4 6 5
```

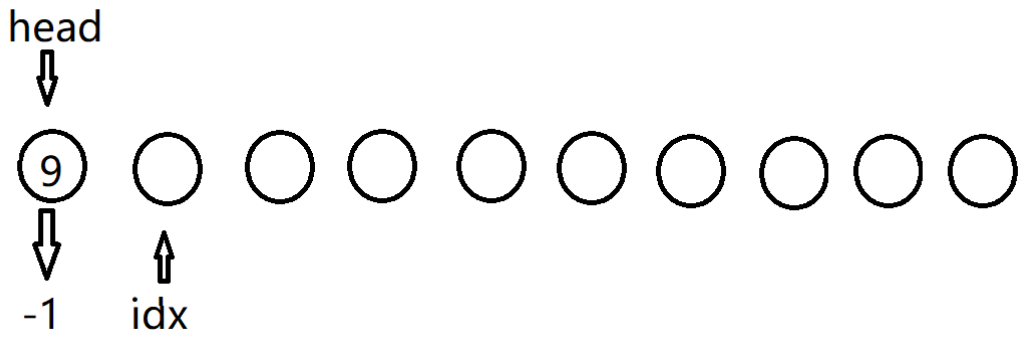
删除第 `k` 个插入的数后面的数，这句话的意思就是删除下标为 `k-1` 结点的后面的结点，因为插入操作始终是通过 `idx++` 一个个递增分配空闲结点，不管前面有没有删除操作，第 `k` 个插入的点的下标一定是 `k-1`（因为不考虑结点回收）

在第 `k` 个插入的数后插入一个数，同理，也就是再下标 `k-1` 的结点后面再插入一个数

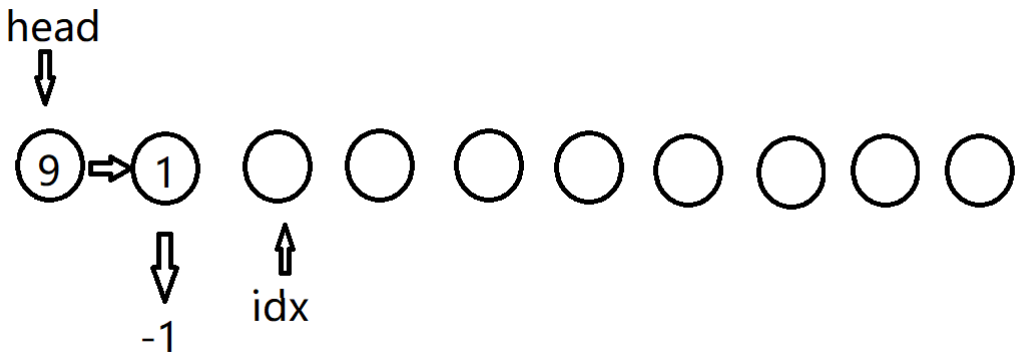
样例过程分析：



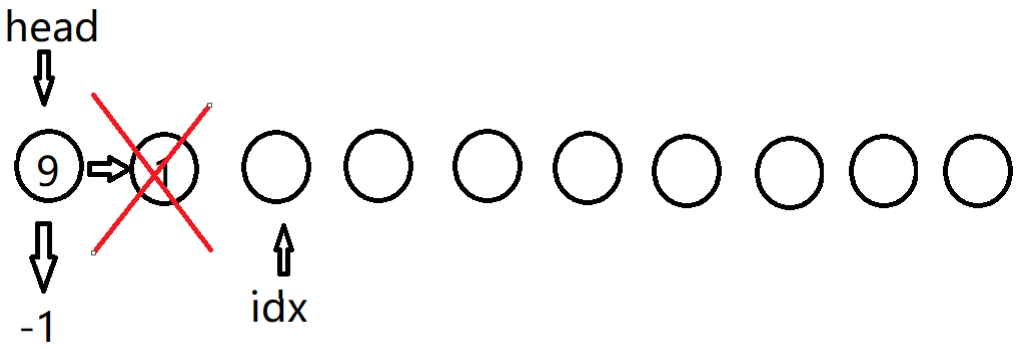
`H 9:`



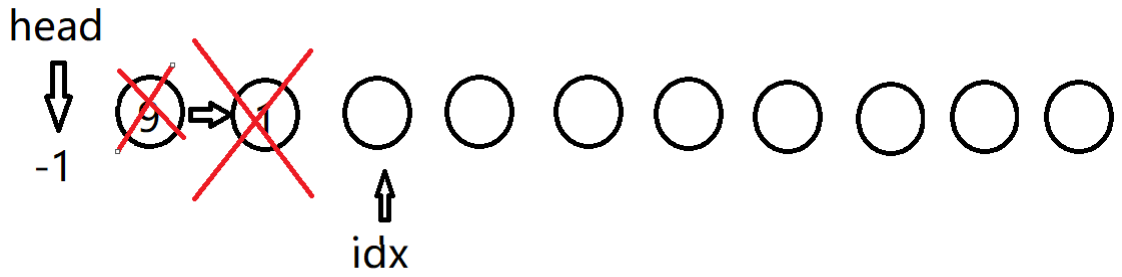
`I 1 1:`



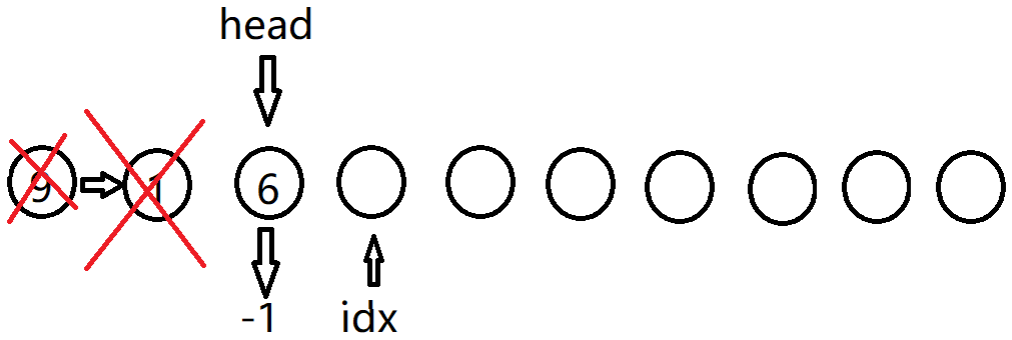
`D 1:`



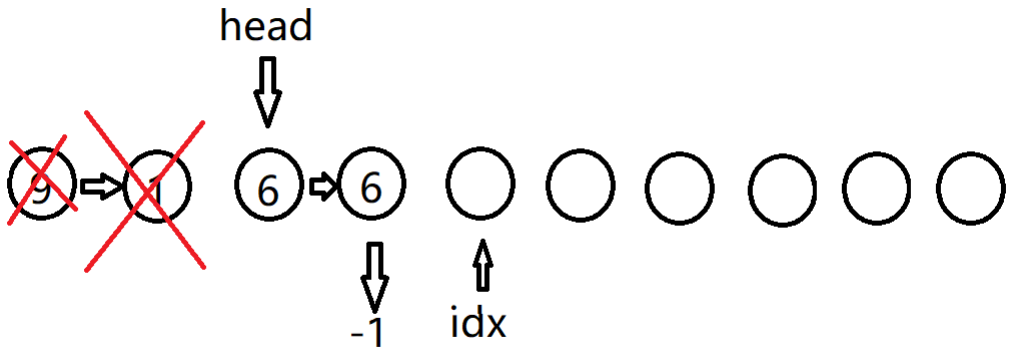
`D 0:`



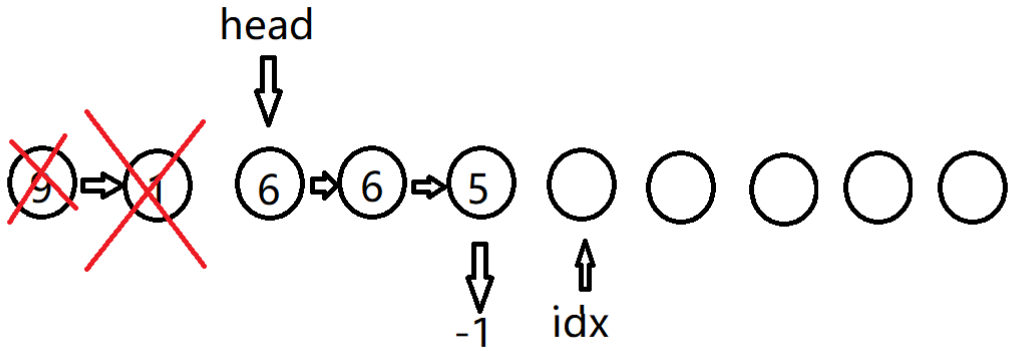
`H 6`:



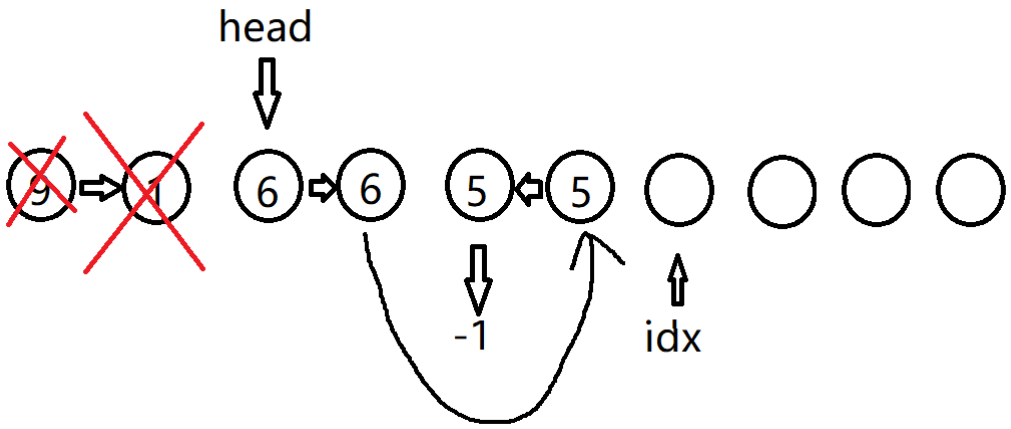
`I 3 6`



`I 4 5`

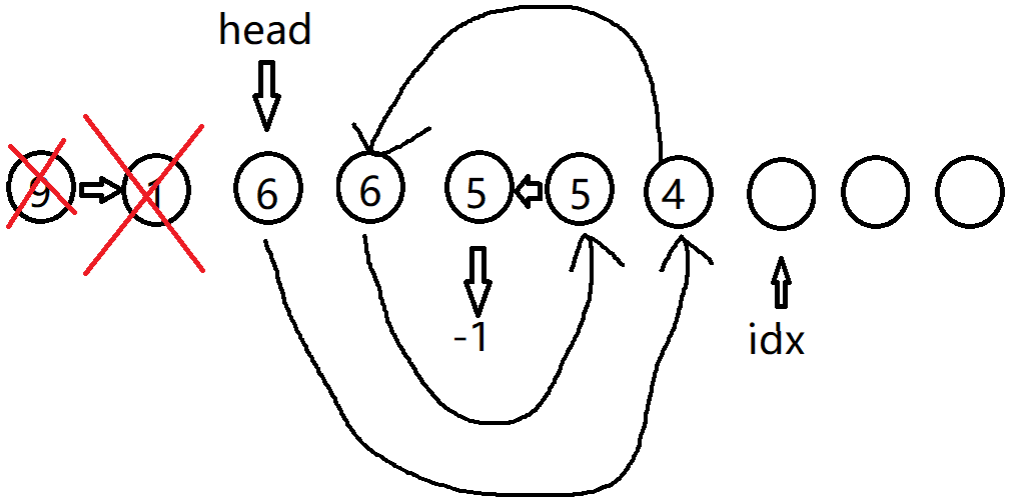


`I 4 5`

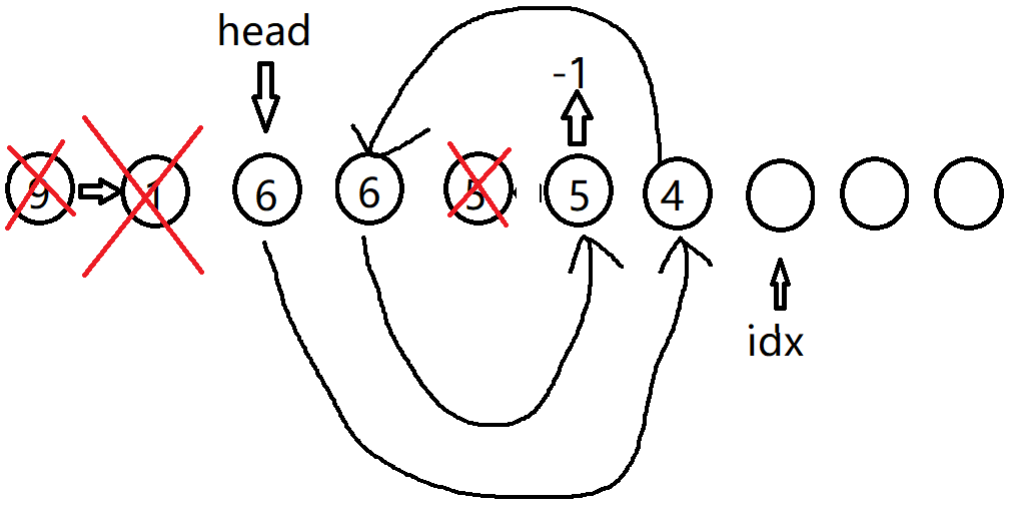


可以看到此时链表的顺序与数组的顺序已经开始不同了

`I 3 4`



`D 6`



此时链表的值按顺序输出应该为`6 4 6 5`

```
#include<iostream>
using namespace std;
const int N = 100010;

int head, e[N], ne[N], idx;


//初始化
void init(){
    head = -1;
    idx = 0;//一开始idx为0表示所有的结点全是空闲结点，都可以进行分配
}

//将x插入到首结点，由于没有头节点，这时得特判操作
void add_to_head(int x){
    e[idx] = x;
    ne[idx] = head;
    head = idx;
    //e[idx] = ele;
    idx++;//可分配结点++
}

void add(int k, int x){
    e[idx] = x;
    ne[idx] = ne[k];
    ne[k] = idx;
    idx++;
}

void remove(int k){
    ne[k] = ne[ne[k]];
}

int main(){
    int m;
    cin >> m;
    //cout<<"kkk";
    init();
    while(m--){
        int k, x;
        char op;
        cin >> op;
        if(op == 'H'){
            cin >> x;
            add_to_head(x);
        }else if(op == 'D'){
            cin >> k;
            remove(k-1);
```

```
            if(k == 0){
                head = ne[head];
            }
        }else if(op == 'I'){
            cin >> k >> x;
            add(k-1, x);
        }
    }

    for(int i = head; i != -1; i = ne[i]){
            cout << e[i] << " ";
    }
    return 0;
}

```

空闲链表的回收机制

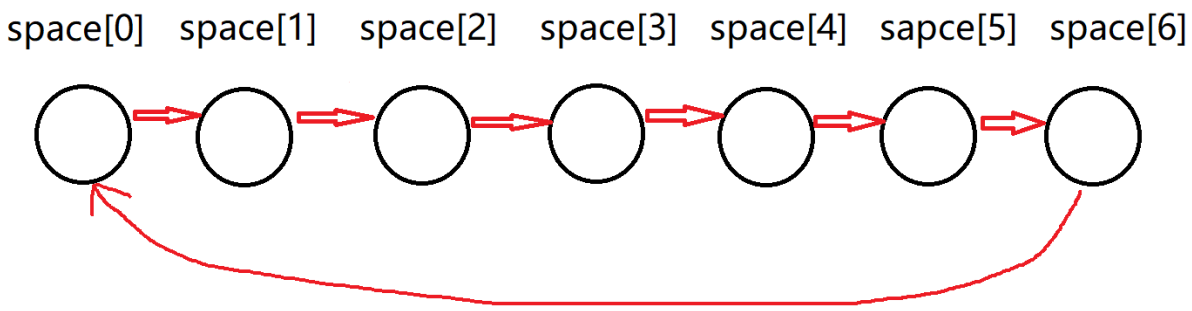
在这里介绍课本上的实现主要是觉得那个空闲结点的回收机制太巧妙了，值得学习。

结点结构，在课本中，一个结点用一个结构体来描述,里面包含一个数据，一个指向下一个结点的指针,地址仍然是结点的下标：

```
struct Node{
    int data;
    int next;
} space[N];//声明一组可分配的空间
```

空闲结点的初始化：

我们先将所有的结点串成一个循环的链表：



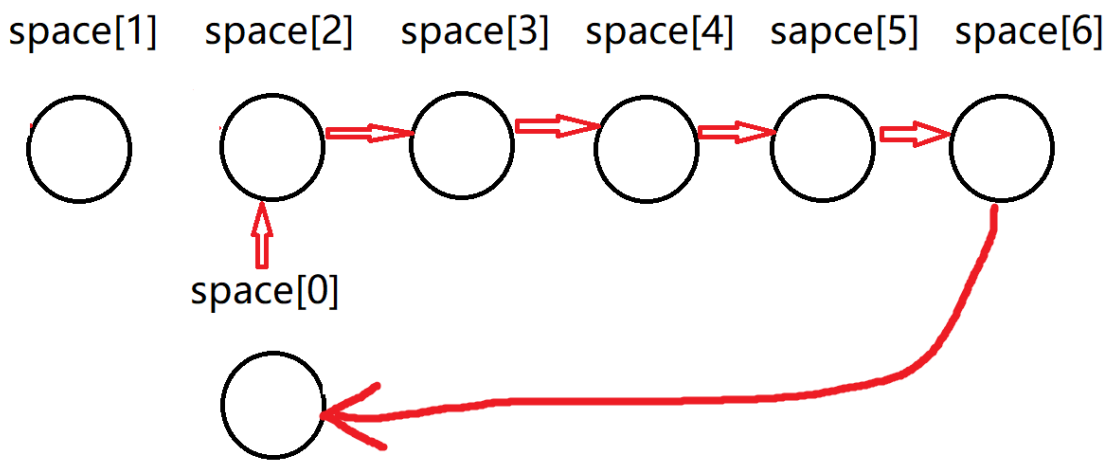
代码如下：

```
void init_SL(Node space[]){
    //将一维数组中的各分量链成一个链表，space[0].cur是头指针
    for(int i = 0; i < N - 1; i ++){
        space[i].next = i + 1;
    }
    space[N-1].cur = 0;
}
```

空闲结点的分配：

这里也就是类似于`malloc`函数的实现，这里`space[0].cur`当作上文中`idx`指针的功能，始终指向我们用到了哪个空闲结点，在上图中我们结点一个都没用，所以`space[0].cur`指向`space[1]`

当分配了一个结点之后，也与`idx`类似，但是我们不是`++`，而是指向当前分配结点的下一个，在上图中，当`space[1]`分配出去被使用之后，我们就指向`space[2]`，如下图：



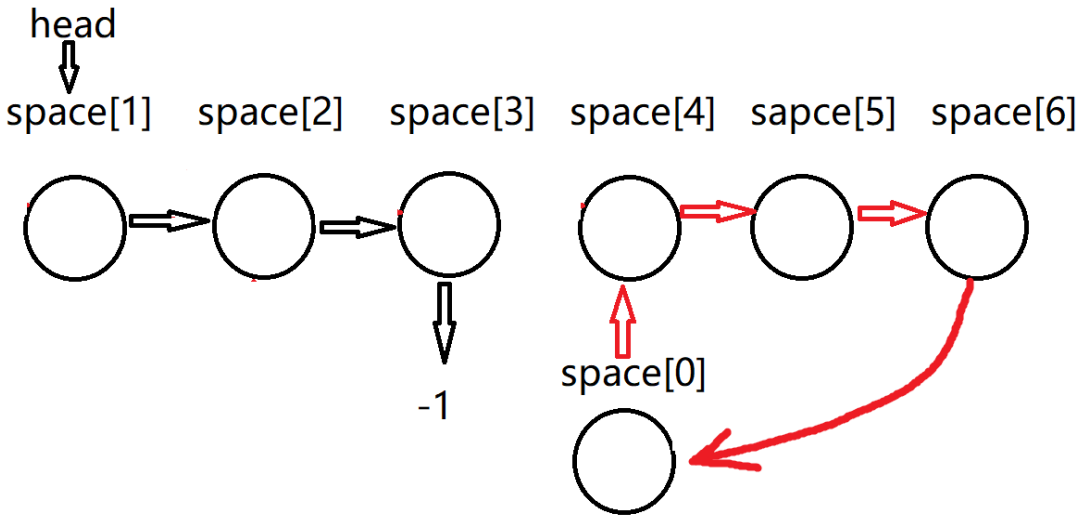
从上图我们也可以看到，当空闲结点全部分配完成后，`space[0].cur = 0`指向自己

代码如下：

```
int malloc_SL(Node space[]){
    i = space[0].cur;//分配当前空闲结点的第一个，也就是我们用到了的这一个空闲结点
    if(space[0].cur){//若为0，则表示空闲结点已经用完了
        space[0].cur = space[i].cur;//若space[i].cur != 0表示空闲结点链表不为空，space[0].cur后移一位
    }
    return i;
}
```

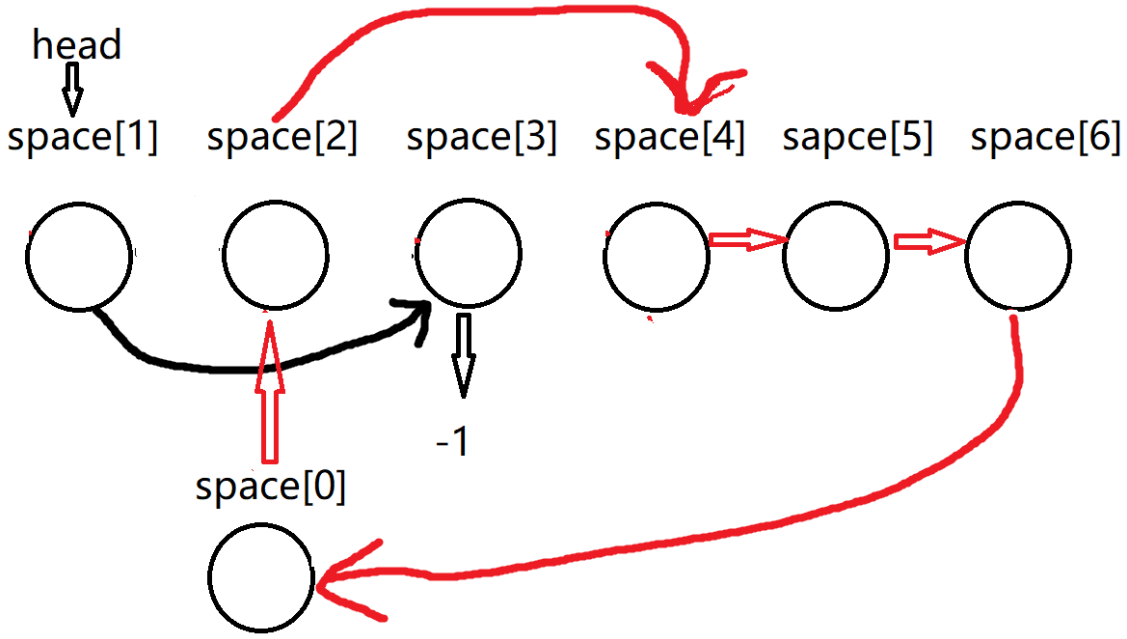
空闲结点的回收：

这里也就是类似于`free`函数，回收这个结点，我们做的操作时统一将这个结点插入到头指针之后变成首结点，比如我们已经分配了下标为`1 2 3`的结点：



在用户的使用过程中必然建立了自己的链表结构，如上图的黑色箭头所示，空闲的链表结构就如红色箭头表示，现在用户删除了下标为`2`的结点，这时，我们为了不使空间浪费，于是需要将`space[2]`回收到空闲的结点链表中，以便于下次使用再进行分配

用户的删除过程不再赘述，删除好了之后我们将`space[2]`插入到空闲链表的首结点，在上图中也就是将`space[2]`指向`space[4]`，然后再将`space[0]`指向`space[2]`，如下图：



这样就达成了结点的回收，等下次再需要分配空闲结点的时候直接分配`space[2]`即可

代码如下：

```
void free_SL(Node space[], int k){//将下标为k的链表回收到空闲链表中
    space[k].cur = space[0].cur;
    spcace[0].cur = k;
}
```

怎么样，这个思想非常巧妙吧，下面再用一个例子来彻底理解这个过程（也可以不看

一个简单的例子：求$(A-B)\cup(B-A)$

要求：依次输入集合`A`和集合`B`的元素，然后在一维数组中建立表示集合$(A-B)\cup(B-A)$的静态链表

基本思想：

1. 先输入`A`的所有元素，在`space[]`中建立一个`A`的链表
2. 输入`B`的元素，对于`B`的每一个元素，依次遍历`A`，若不在`A`中，则分配一个空闲结点，插入`B`的这个元素进入到`A`链表中，若在`A`中，则删除`A`中的这个元素

代码实现：

```
void defference(Node space[], int &S){
    init_SL(space);//空闲链表的初始化
    S = malloc_SL(space);//分配一个结点作为用户链表的头结点
    r = S;//这里指针r指向用户链表的最后一个结点作为尾指针，方便下面的判断
    int m,n;
    cin >> m >> n;//输入集合A和集合B的个数

    for(int j = 1; j <= m; j++){//建立A链表，这时A链表也就是用户链表
        i = malloc_SL(space);//分配一个空闲结点，并返回地址
        cin >> space[i].data;//输入元素值
        space[r].next = i, r = i;//插入到链表的尾部
    }
    int b;

    for(int j = 1; j <= n; j ++){
        //依次输入B的元素遍历A，若不在A中，则分配一个空闲结点，插入这个元素进入到A链表中，若在A中，则删除A中的这个元素
        cin >> b;
        p = S;//p指向头结点,p在这里始终指向k指针的前一个结点，方便后面的删除操作
        k = space[S].next;//k先指向A中的第一个结点(不是头节点)
        while(k != space[r].next && space[k].data != b){//若k没有遍历完，并且A中也没有b
            p = k, k = space[k].next//继续遍历
        }
        //退出循环就两种可能，一是将A链表遍历完了，没有找到此时k指向A链表尾结点的下一个即space[r].next
        //这时需要将b元素插入到A链表中
        if(k == space[r].next){
            //插入A链表尾部
            i = malloc_SL(space);
            space[i].data = b;
            space[i].next = space[r].next;
            space[r].next = i;
        }else{//第二种情况，这时在A中找到了b元素，此时k指向这个结点，p指向这个结点的前一个结点
            //这时删除这个结点
            space[p].next = space[k].next;
            free_SL(space, k);
```

```
            if(r == k) r = p;//如果删除的是尾结点，需要重新设置尾指针，特判
        }
    }

}
```

双链表

基本思想

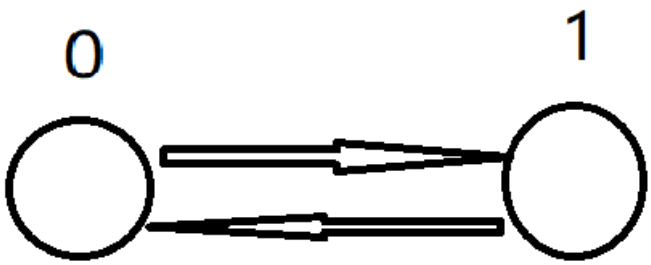
双链表的话就是每一个结点都具有两个指针，一个指向前一个指向后，这是很基本的数据结构，具体就不再赘述了，只说说几个关键的东西

存储结构：

我们用`l[]`数组表示每个结点的左指针，用`r[]`表示每个数组的右指针，用`e[]`来表示结点的值，同单链表一样，我们用下标来将这三个数组联系起来

初始化：

我们用`0`号结点当作头指针，用`1`号结点当作尾指针，初始化的时候我们是形成下图的结构，当作指针的两个结点没有左右指针：




```
//初始化
void init(){
    //0表示左端点，1表示右端点
    r[0] = 1;
    l[1] = 0;
    idx = 2;
}
```

插入操作，插入到`k`结点的右面：

```
//在下标是k的结点的右边插入一个点
void insert(int k, int x){
    e[idx] = x;//赋值
    l[idx] = k;//插入结点的左指针指向k
    r[idx] = r[k];//插入结点的右指针指向原来k后面的结点
    l[r[k]] = idx;//原来k结点后面的结点的左指针指向插入的结点
    r[k] = idx;//k指针的右结点指向插入的结点
    idx++;
}
//插入到首结点以及插入到尾结点的时候不用进行特判
```

删除操作：

```
void remove(int k){
   r[l[k]] = r[k];//让被删除结点的左结点的右指针指向被删除结点的右结点
   l[r[k]] = l[k];//让被删除结点的右结点的左指针指向被删除结点的左结点
}
```

代码实现

实现一个双链表，双链表初始为空，支持 5 种操作：

1. 在最左侧插入一个数；
2. 在最右侧插入一个数；
3. 将第 k 个插入的数删除；
4. 在第 k 个插入的数左侧插入一个数；
5. 在第 k 个插入的数右侧插入一个数

现在要对该链表进行 M 次操作，进行完所有操作后，从左到右输出整个链表。

注意:题目中第 k 个插入的数并不是指当前链表的第 k 个数。例如操作过程中一共插入了 n 个数，则按照插入的时间顺序，这 n 个数依次为：第 1 个插入的数，第 2 个插入的数，…第 n 个插入的数。

输入格式

第一行包含整数 M，表示操作次数。

接下来 M 行，每行包含一个操作命令，操作命令可能为以下几种：

1. `L x`，表示在链表的最左端插入数 x。
2. `R x`，表示在链表的最右端插入数 x。
3. `D k`，表示将第 k 个插入的数删除。
4. `IL k x`，表示在第 k 个插入的数左侧插入一个数。
5. `IR k x`，表示在第 k 个插入的数右侧插入一个数。

输出格式

共一行，将整个链表从左到右输出。

数据范围

$1 \leq M \leq 100000$

所有操作保证合法。

输入样例：

```
10
R 7
D 1
L 3
IL 2 10
D 3
IL 2 7
L 8
R 9
IL 4 7
IR 2 2
```

输出样例：

```
8 7 7 3 2 9
```

```cpp
#include<iostream>
using namespace std;
const int N = 100010;

int m;
int e[N], l[N], r[N], idx;

//初始化
void init(){
    //0表示左端点，1表示右端点
    r[0] = 1;
    l[1] = 0;
    idx = 2;
}

//插入操作
void add(int k, int x){
    e[idx] = x;
    l[idx] = k;
    r[idx] = r[k];
    l[r[k]] = idx;
    r[k] = idx;
    idx ++;
}

//删除下标为k的结点
void removex(int k){
    r[l[k]] = r[k];
    l[r[k]] = l[k];
}
```

```
31
32  int main(){
33      cin >> m;
34      char op[2];
35      init();
36      while(m --){
37          scanf("%s", op);
38          int k, x;
39          //cout << "sss";
40          if(op[0] == 'L'){
41              scanf("%d", &x);
42              add(0, x);
43          }else if(op[0] == 'R'){
44              scanf("%d", &x);
45              add(l[1], x);
46          }else if(op[0] == 'D'){
47              scanf("%d", &k);
48              removex(k + 1);
49          }else if(op[1] == 'L'){
50              scanf("%d%d", &k, &x);
51              add(l[k + 1], x);
52          }else{
53              scanf("%d%d", &k, &x);
54              add(k + 1, x);
55          }
56      }
57
58      for(int i = r[0]; i != 1; i = r[i]){
59          printf("%d ", e[i]);
60      }
61
62      return 0;
63  }
64
65
```