

链表

【存储方式的分类】：顺序存储结构和链式存储结构；

【顺序存储结构】：在（子）程序的说明部分就必须加以说明，以便分配固定大小的存储单元，直到（子）程序结束，才释放空间。因此，这种存储方式又称为静态存储。所定义的变量相应的称为静态变量。它的优缺点如下：

1、优点：可以通过一个简单的公式随机存取表中的任一元素，逻辑关系上相邻的两个元素在物理位置上也是相邻的，且很容易找到前趋与后继元素；

2、缺点：在线性表的长度不确定时，必须分配最大存储空间，使存储空间得不到充分利用，浪费了宝贵的存储资源；线性表的容量一经定义就难以扩充；在插入和删除线性表的元素时，需要移动大量的元素，浪费了时间；

【链式存储结构】：在程序的执行过程中，通过两个命令向计算机随时申请存储空间或随时释放存储空间，以达到动态管理、使用计算机的存储空间，保证存储资源的充分利用。这样的存储方式称为动态存储。所定义的变量称为动态变量。它的优点如下：

【优点】：可以用一组任意的存储单元（这些存储单元可以是连续的，也可以不连续的）存储线性表的数据元素，这样就可以充分利用存储器的零碎空间；

【概念】：为了表示任意存储单元之间的逻辑关系，对于每个数据元素来说，除了要存储它本身的信息（数据域、data）外，还要存储它的直接后继元素的存储位置（指针域、link 或 next）。我们把这两部分信息合在一起称为一个“结点 node”。

1、N 个结点链接在一起就构成了一个链表。N=0 时，称为空链表。

2、为了按照逻辑顺序对链表中的元素进行各种操作，我们需要定义一个变量用来存储整个链表的第一个结点的物理位置，这个变量称为“头指针、H 或 head”。也可以把头指针定义成一个结点，称为“头结点”，头结点的数据域可以不存储任何信息，也可以存储线性表的长度等附加信息，头结点的指针域（头指针）存储指向第一个结点的指针，若线性表为空表，则头结点的指针域为空（NIL）。由于最后一个元素没有后继，所以线性表中最后一个结点的指针域为空（NIL）。

3、由于此链表中的每个结点都只包含一个指针域，故称为“线性链表或单链表”。

（一）单链表的定义

1. 类型和变量的说明

```
struct Node
{
    int data;
    Node *next;
};
Node *p;
```

2. 申请存储单元 //动态申请、空间大小由指针变量的基类型决定

```
p=new Node;
```

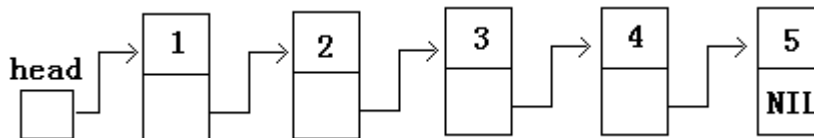
3. 指针变量的赋值

指针变量名=NULL; //初始化，暂时不指向任何存储单元

如何表示和操作指针变量？不同于简单变量（如 A=0;），c++规定用“指针变量名->”的形式引用指针变量（如 P->data=0;）。

(二) 单链表的结构、建立、输出

由于单链表的每个结点都有一个数据域和一个指针域，所以，每个结点都可以定义成一个记录。比如，有如下一个单链表，如何定义这种数据结构呢？



下面给出建立并输出单链表的程序，大家可以把它改成过程用在以后的程序当中。

```

#include<iostream>
using namespace std;
struct Node
{
    int data;
    Node *next;
};
Node *head,*p,*r;      //r 指向链表的当前最后一个结点，可以称为尾指针
int x;
int main()
{
    cin>>x;
    head=new Node;      //申请头结点
    r=head;
    while(x!=-1)         //读入的数非-1
    {
        p=new Node;     //则，申请一个新结点
        p->data=x;
        p->next=NULL;
        r->next=p;       //把新结点链接到前面的链表中，实际上 r 是 p 的直接前趋
        r=p;            //尾指针后移一个
        cin>>x;
    }
    p=head->next;        //头指针没有数据，只要从第一个结点开始就可以了}
    while(p->next!=NULL)
    {
        cout<<p->data<<" ";
        p=p->next;
    }
    cout<<p->data<<endl; //最后一个结点的数据单独输出，也可以改用 REPEAT 循环}
    system("pause");
}
  
```

(三) 单链表的操作

1. 查找“数据域满足一定条件的结点”

```

p=head->next;
while((p->data!=x)&&(p->next!=NULL) p=p->next; //找到第一个就结束
if(p->data==x)找到了处理;
else 输出不存在;
  
```

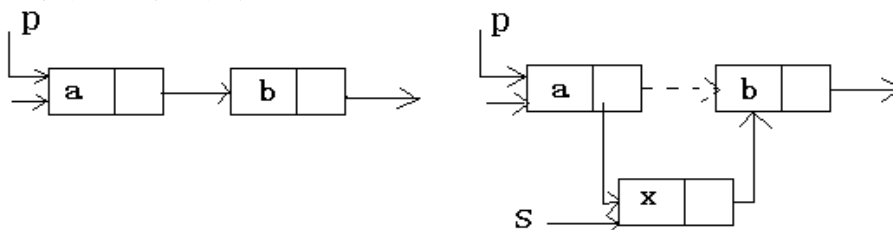
如果想找到所有满足条件的结点，则修改如下：

```
p=head->next;
while (p->next!=NULL)    //一个一个判断
{
    if (p->data==x) 找到一个处理一个;
    p=p->next;
}
```

2. 取出单链表的第 i 个结点的数据域

```
void get(Node *head,int i)
{   Node *p;int j;
    p=head->next;
    j=1;
    while((p!=NULL)&&(j<i))
    {
        p=p->next;
        j=j+1;
    }
    if((p!=NULL)&&(j==i))cout<<p->data;
    else    cout<<"i not exist!";
}
```

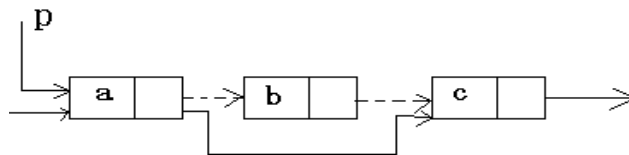
3. 插入一个结点在单链表中去



插入结点前和后的链表变化

```
void insert(Node *head,int i,int x)//插入 X 到第 i 个元素之前
{   Node *p,*s;int j;
    p=head;
    j=0;
    while((p!=NULL)&&(j<i-1))//寻找第 i-1 个结点,插在它的后面
    {
        p=p->next;
        j=j+1;
    }
    if(p==NULL)cout<<"no this position!";
    else{    //插入
        s=new Node;
        s->data=x;
        s->next=p->next;
        p->next=s;
    };
}
```

4. 删除单链表中的第 i 个结点（如下图的“b”结点）



删除一个结点前和后链表的变化

```

void delete(Node *head,int i)//删除第 i 个元素
{
    Node *p,*s;int j;
    p=head;
    j=0;
    while((p->next!=NULL)&&(j<i-1))
    {
        p=p->next;
        j=j+1;
    } //p 指向第 i-1 个结点
    if (p->next==NULL)cout<<"no this position!";
    else { //删除 p 的后继结点，假设为 s
        s=p->next;
        p->next=p->next->next; //或 p->next=s->next
        free(s);
    }
}

```

5. 求单链表的实际长度

```

int len(Node *head)
{
    int n=0;
    p=head;
    while(p!=NULL)
    {
        n=n+1;
        p=p->next;
    }
    return n;
}

```

(四) 双向链表

每个结点有两个指针域和若干数据域，其中一个指针域指向它的前趋结点，一个指向它的后继结点。它的优点是访问、插入、删除更方便，速度也快了。但“是以空间换时间”。

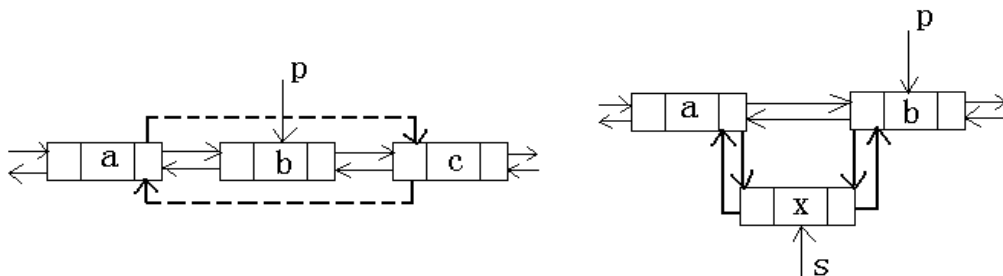
【数据结构的定义】:

```

struct node
{
    int data;
    node *pre,*next; //pre 指向前趋，next 指向后继
}
node *head,*p,*q,*r;

```

下面给出双向链表的插入和删除过程。



删除P结点前后的指针变化

在P结点之前插入S结点前后的指针变化

void insert(node *head,int i,int x)//在双向链表的第 i 个结点之前插入 X

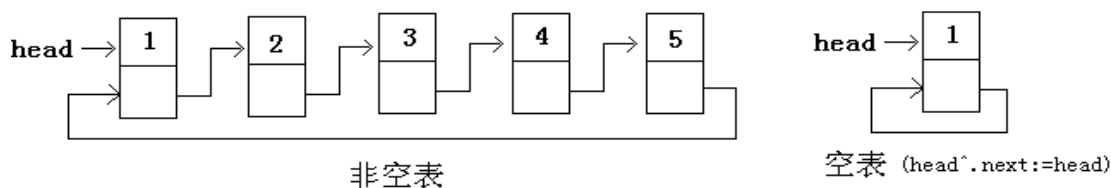
```
{  node *s,*p;
    int j;
    s=new node;
    S->data=x;
    P=head;
    j=0;
    while((p->next!=NULL)&&(j<i))
    {
        p=p->next;
        j=j+1;
    } //p 指向第 i 个结点
    if(p==NULL) cout<<"no this position!";
    else { //将结点 S 插入到结点 P 之前
        s->pre=p->pre; //将 S 的前趋指向 P 的前趋
        p->pre=s;      //将 S 作为 P 的新前趋
        s->next=p;     //将 S 的后继指向 P
        p->pre->next=s; //将 P 的本来前趋结点的后继指向 S
    }
}
```

void delete(node *head,int i)//删除双向链表的第 i 个结点

```
{  int j;
    node *p;
    P=head;
    j=0;
    while((p->next!=NULL)&&(j<i))
    {
        p=p->next;
        j=j+1;
    } //p 指向第 i 个结点
    if(p==NULL) cout<<"no this position!";
    else { //将结点 P 删除
        p->pre->next=p->next; //P 的前趋结点的后继赋值为 P 的后继
        p->next->pre=p->pre;  //P 的后继结点的前趋赋值为 P 的前趋
    }
}
```

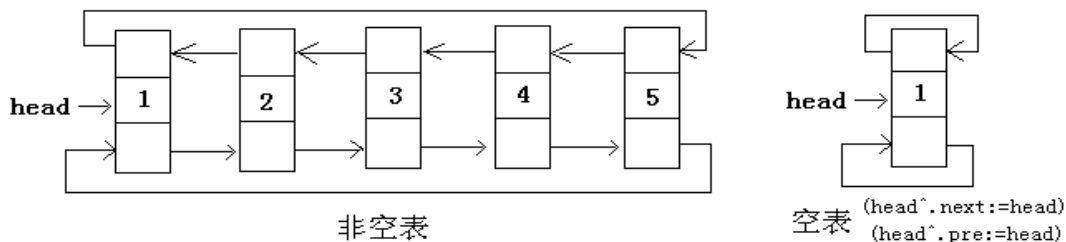
（五）循环链表

单向循环链表：最后一个结点的指针指向头结点。如下图：



单向循环链表

双向循环链表：最后一个结点的指针指向头结点，且头结点的前趋指向最后一个结点。如下图：



双向循环链表

（六）循环链表的应用举例

约瑟夫环问题

【问题描述】

有 M 个人，其编号分别为 $1-M$ 。这 M 个人按顺序排成一个圈（如图）。现在给定一个数 N ，从第一个人开始依次报数，数到 N 的人出列，然后又从下一个人开始又从 1 开始依次报数，数到 N 的人又出列。... 如此循环，直到最后一个人出列为止。

【输入格式】

输入只有一行，包括 2 个整数 M ， N 。之间用一个空格分开 ($0 < n \leq m \leq 100$)。

【输出格式】

输出只有一行，包括 M 个整数

【样例输入】

8 5

【样例输出】

5 2 8 7 1 4 6 3

【参考程序】

```
#include <iostream>
using namespace std;
struct node
{
    long d;
    node *next;
};
long n, m;
node *head, *p, *r;
int main()
```

```
{
    long i, j, k, l;
    cin>>n>>m;
    head=new node;
    head->d=1; head->next=NULL; r=head;
    for (i=2;i<=n;i++)
    {
        p=new node;
        p->d=i;
        p->next=NULL;
        r->next=p;
        r=p;
    }
    r->next=head; r=head;
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=m-2;j++) r=r->next;
        cout<<r->next->d<<" ";
        r->next=r->next->next;
        r=r->next;
    }
}
```