实验1 线性表的实现（基于链表）

原创

大白不白 于 2018-07-26 18:16:31 发布
分类专栏： 数据结构

回头整理一下以前课程实验的代码，顺便学习一波。

问题描述实验要求

基于教材内容，实现线性表。

基本要求

需要基于顺序表（数组）或链表来实现线性表ADT

需要实现线性表的各个基本操作

编写一个demo程序，测试线性表的各个基本操作的功能是否正常

一、实验分析：

ADT就是抽象数据结构，线性表有两种，顺序表也就是我们常用的数组，另一种是链表，顺序表和链表最大的区别就在于地址，顺序表存储数据的地址是连续的且空间利用率高。而链表存储数据的地址是不连续的，空间利用率比顺序表低很多，因为链表的空间需要存储指针和数据。顺序表如何实现我就不说了，重点说下链表的实现方式。实现链表一般基于类或者结构体均可。刚才我说了链表有两种东西是一定要定义的：数据域和指针域，因此可以这样定义：int data， Link * next ;那么单个链表节点类需要的属性就定义好了，注意是单个链表节点，一定要分清和链表的区别，多个链表节点连起来才是链表。构造方法就是初始化一下这些属性。然后就是链表的构造再单独写一个类，这个类的作用就是链接节点以及一些链表的基本操作（基本操作有删除一个节点、添加一个节点、查找某一个节点，就是删查找）。

二、具体实现:

链表单个节点类：

```
class LinkNode{   //单链表
    public:
        int elem;//数据域，用于保存节点的数据
        LinkNode *next; //next指针，指向节点的元素域
        LinkNode(int Elem = 0,LinkNode *link = NULL) { this->elem = Elem; this->next = link; }//构造一个链表
};
```

（整条链表）链表类：

```
class LLink{
    private:
        LinkNode* head;//表头节点
        LinkNode* tail;//表尾节点
        int length;//链表长度
        public:
        LLink();
        ~LLink();
        bool Add(LinkNode* );//向链表添加节点
        bool insert(int,LinkNode'*);//在任意位置插入一个新节点
        void print();//打印链表
        int getLenght(){ return this->length;}//返回链表长度
        int getElementByPos(int);//返回任意位置节点
        void pop();//删除链尾节点
        int getLast() const;//返回链尾节点
        bool deleteElementByPos(int);//删除任意位置节点
        int getPos(int,LLink* link);//返回元素的位置
};
```

以上这个类分别定义了一些基本操作。

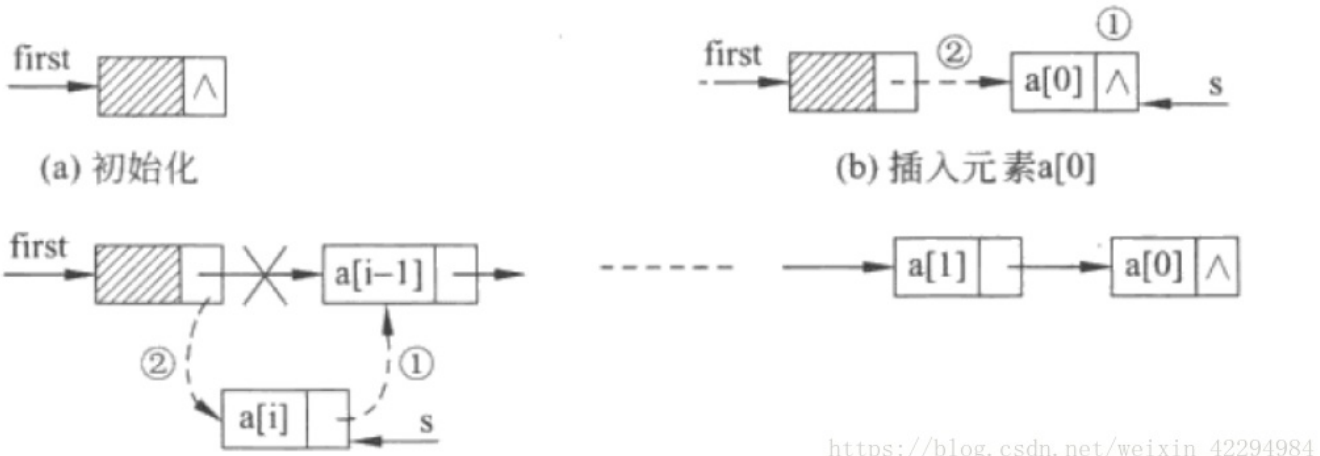
下面讲如何具体实现:

1.如何将节点连成一条链呢？

有两种方法：（1）头插法（2）尾插法。

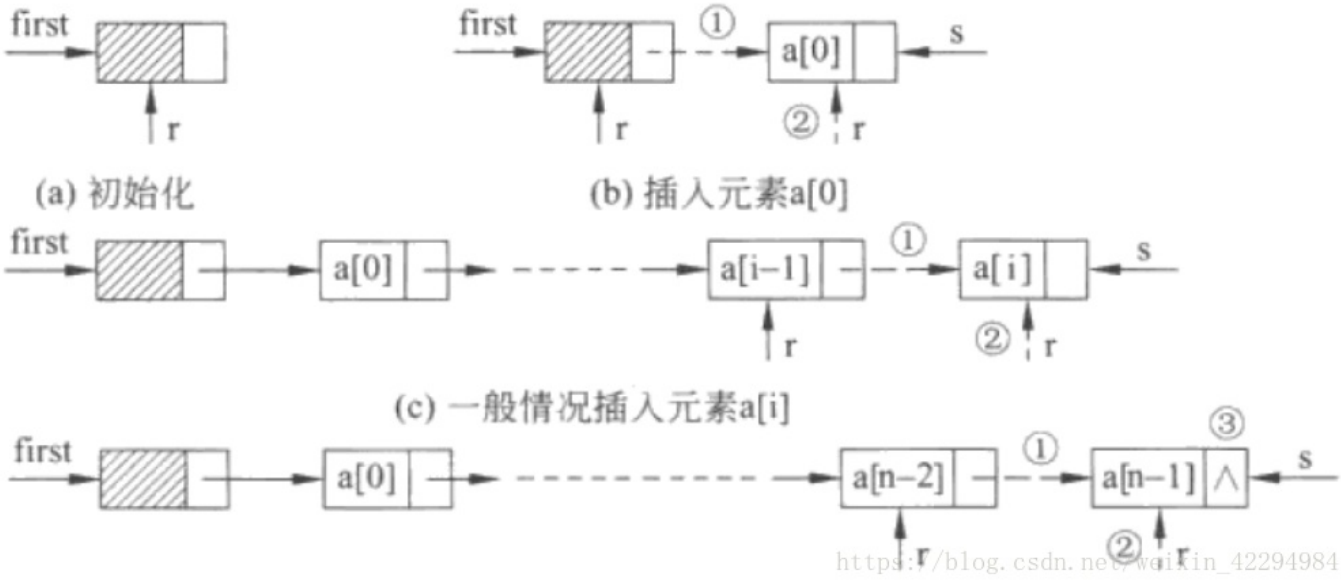
头插法就是先定义一个表头，就是那个head指针，表头独有的，然后每往链表加入一个新节点，都将新节点接在头节点的后面，这样子的插入方法叫头插法。

如图：



第二种就是尾插法啦，尾插顾名思义每次都将新节点作为尾节点插入，什么意思呢？就是接在上一次插入节点的后面。

如图：



使用这种方法务必注意：当你不再插入新节点时，务必把最后一个节点的next指针域设置为空，以及next = null；否则你的链表就永远没有结束，获取长度的时候会出问题的。

具体代码实现：(尾插法）

```
bool LLink::Add(Link* link)
{
    this->tail->next = link;//表尾元素的next指向link的元素域
 this->tail = link;//更新尾部为新插入的节点
 this->length++;
 return true;
}
```

使用bool只是为了判断有没有成功插入而已，具体何种返回值自己决定。插入一个节点后记得更新链表的长度。

就是两步工作：将前面插入的节点的next指针指向新插入节点的数据域，然后将尾部更新为新节点，也就是说链表的尾部现在是新插入的节点。（头插法我就简单说一下：新节点的next域与上一次插入的节点的数据域相连，然后新节点的数据域和头节点的next相连即可，每次都要更新head节点的next指针域。

2.基本操作的实现:

(1)查找某一个元素：

查找链表的某一个节点，可以按位置或者或节点数据域的存储的值来查找。我这里是按位置来查找，也就是直接返回某个位置的数据值。

实现思路，直接遍历整个链表，判断当前位置是否为需要返回数据的位置即可。

```
int LLink::getElementByPos(int pos)
{
 if(pos<0 || pos>this->getLenght()-1)//位置是否合法
 {
  cout<<"Out of range"<<endl;
  return -1;
 }
 Link* p = this->head->next;//开始找的位置
 int k=0;
 while(k<pos)
 {
  p = p->next;//节点递增
  k++;
  }
  int m = p->elem;
  return m;
}
```

2.删除某个节点

这里稍微有点复杂，因为涉及到前面节点的链接问题，我举个例子吧，假如现在有1，2，3总共3个节点，现在要删除2这个节点，我妈要考虑是如何将1，3节点正确连接，才不会使链表断开。一个节点有两个属性，next指针域和数据域，要连接两个节点，无非就是修改next指针的指向，因此通过修改1->next = 3就可以了。也就是这样的语句：1->next->next表示的3这个节点。

下面我贴一下按位置删除节点的代码：

思路是先找到待删除节点位置的前一个节点，然后将该节点与被删除节点的下一个节点连接即可。然后释放被删除节点的空间即可。

```
bool LLink::deleteElementByPos(int pos)
{
 if(pos<0 || pos>this->getLenght()-1)//位置是否合法
 {
  cout<<"Out of range"<<endl;
  return false;
 }
 Link* tmp = this->head;
 int k=-1;//设置为-1的目的就是让它找到被删除节点的前一个节点
 while(k<pos-1)
 {
  tmp = tmp->next;//节点的递增
  k++;
 }
 Link* del = tmp->next;
 tmp->next = tmp->next->next;//与它的下下个节点连接，跳过被删除的节点
 delete del;//释放被删除节点的空间
 this->length--;//注意更新链表的长度
 return true;
}
```

3.节点的插入:

还是像之前一样的，通过修改节点的next指向。但元素的插入有个规则，即先和后面的节点连接再和前面的节点连接。具体看代码注释。

```
bool LLink::insert(int pos, LinkNode* link)
{
 if(pos>this->getLenght())//判断插入是否合法
 {
  cout<<"Out of range !"<<endl;
  return false;
 }
 int i=0;
 Link* fence = new Link();//初始化一个空的节点指针
 fence = this->head;//开始找的位置，也就是从表头开始找 ，为什么不能直接插呢？因为确定位置要依靠表头或者表尾呀！
 while(i<pos)
 {
  fence = fence->next;//节点不断递增，直到i=pos
  i++;
  }
  link->next = fence->next;//这里是意思就是将待插入节点的next与fence节点的下一个节点连接
  fence->next = link;//将待插入位置前的节点的next指向待插入节点的数据域。
  this->length++;
  return true;
 }
```

fence->next表示的是fence的下一个节点，看循环里面就可以看得出来。这样子理解是因为fence的next域里面保存着下一个节点的相关信息，因此可以认为fence->next就是指它的下一个节点，将这些信息赋给新节点的next，就相当于新节点的next域保存了它下一个节点的信息，那这样子是不是连接在一起了呢？我反正是这样理解next的。理解好next对理解链表有着极大的帮助。

其它剩余的就靠你们自己去实现一下啦。原理就讲这么多啦。

附录完整代码：

main.cpp

```
#include<iostream>
#include"LLink.h"
using namespace std;
int main()
{
    Link* link1= new Link(1);
    Link* link2= new Link(2);
    Link* link3= new Link(3);
    Link* link4= new Link(4);
    LLink* link= new LLink();
    link->Add(link1);
    link->Add(link2);
    link->Add(link3);
    link->Add(link4);
    cout<<"初始链表："<<endl;
    link->print();
    cout<<"\n";
    cout<<"1.插入数据"<<endl;
    cout<<"2.查找数据"<<endl;
 cout<<"3.删除数据"<<endl;
 cout<<"4.返回链表尾部元素"<<endl;
 cout<<"5.返回某元素位置" <<endl;
 cout<<"6.删除链表尾部元素"<<endl;
 cout<<"7.输入0退出本程序"<<endl;
```

```
 cout<<"请输入您要执行的功能："<<endl;
 int n;
 while(cin>>n&&n!=0)
 {
 if(n==1)
 {
  int pos,data;
  cout<<"请输入您要插入的位置以及待插入的数据："<<endl;
  cin>>pos>>data;
  cout<<"原链表："<<endl;
  link->print();
  Link* LINK = new Link(data);
  link->insert(pos,LINK);
  cout<<"插入数据后："<<endl;
  link->print();
 }


    if(n==2)
    {
     int pos;
     cout<<"请输入待查找元素的位置："<<endl;
  cin>>pos;
  cout<<pos<<" 位置的元素为："<<link->getElementByPos(pos)<<endl;
 }

    if(n==3)
    {
     int pos;
     cout<<"请输入待删除元素的位置："<<endl;
     cin>>pos;
     cout<<"原链表："<<endl;
  link->print();
     link->deleteElementByPos(pos);
     cout<<"删除后的链表为："<<endl;
     link->print();
 }
 if(n==4)
 {
  cout<<"链表尾部的元素为: "<<link->getLast()<<endl;

 }
 if(n==5)
 {
  cout<<"请输入需要返回其位置的元素："<<endl;
  int pos;
  cin>>pos;
  cout<<pos<<" 元素的位置为："<<link->getPos(pos,link)<<endl;
 }
 if(n==6)
    {
     cout<<"原链表："<<endl;
  link->print();
  link->pop();
  cout<<"删除链尾元素后："<<endl;
     link->print();
 }
 }
 return 0;
}
```

```
}
```

Link.h:

```
#ifndef LLINK_H_INCLUDE
#define LLINK_H_INCLUDE
#include<iostream>
using namespace std;
class Link{   //单链表
 public:
  int elem;//数据域，用于保存节点的数据
  Link *next; //next指针，指向节点的元素域
  Link(int Elem = 0,Link *link = NULL) { this->elem = Elem; this->next = link; }//构造一个链表
};

class LLink{
 private:
  Link* head;//表头节点
  Link* tail;//表尾节点
  int length;//链表长度
  public:
  LLink();
  ~LLink();
  bool Add(Link* );//向链表添加节点
  bool insert(int,Link*);//在任意位置插入一个新节点
  void print();//打印链表
  int getLenght(){ return this->length;}//返回链表长度
  int getElementByPos(int);//返回任意位置节点
  void pop();//删除链尾节点
  int getLast() const;//返回链尾节点
  bool deleteElementByPos(int);//删除任意位置节点
  int getPos(int,LLink* link);//返回元素的位置
};
#endif
```

Link.cpp

```
#include"LLink.h"
#include<iostream>
using namespace std;

LLink::LLink()
{
 Link *init = new Link(4);//配合我主函数处的4个节点，故参数为4
 this->head = init;
 this->tail = init;
 this->length=0;//表头表位相等且长度为0，那肯定是一个空链表，也就是这里构造了一个空链表；
}

LLink::~LLink()
{
 while(head)
 {
  Link* tmp = new Link(0,head);//删除节点的老规矩，定义临时指针，并将head的next指针指向tmp
  tmp = head;
  head=head->next;//节点依次递增
  delete tmp;//删除节点的内存
```

```
 }
 delete head;//删除表头
 delete tail;//删除表尾
 }

bool LLink::Add(Link* link)
{
    this->tail->next = link;//表尾元素的next指向link的元素域
 this->tail = link;//赋值
 this->length++;
 return true;
}

bool LLink::insert(int pos, Link* link)
{
 if(pos>this->getLenght())//判断插入是否合法
 {
  cout<<"Out of range !"<<endl;
  return false;
 }
 int i=0;
 Link* fence = new Link();//初始化一个空的节点指针
 fence = this->head;//开始找的位置，也就是从表头开始找 ，为什么不能直接插呢？因为确定位置要依靠表头或者表尾呀！
 while(i<pos)
 {
  fence = fence->next;//节点不断递增，知道i=pos
  i++;
  }
  link->next = fence->next;
  fence->next = link;
  this->length++;
  return true;
 }

void LLink::print()
{
 Link* p = this->head->next;//打印开始的位置
 int k=0;
 while(k< this->getLenght())
 {
  cout<<p->elem<<"\t";
  p = p->next;//下一个节点
  k++;
  }
  cout<<endl;
}

int LLink::getElementByPos(int pos)
{
 if(pos<0 || pos>this->getLenght()-1)//位置是否合法
 {
  cout<<"Out of range"<<endl;
  return -1;
 }
 Link* p = this->head->next;//开始找的位置
 int k=0;
 while(k<pos)
 {
  p = p->next;//节点递增
  k++;
```

```
  k++;
  }
  int m = p->elem;
  return m;
}

void LLink::pop()
{
 Link* tmp = this->head;//开始找的位置
 int i=0;
 while(i< this->getLenght()-1)
 {
  tmp = tmp->next;//节点递增
  i++;
  }
 Link* t = this->tail;//t是临时指针，用于删除tail的内存
 this->tail = tmp;
 this->length--;
 delete t,tmp;

}

int LLink::getLast() const
{
 return this->tail->elem;
}

bool LLink::deleteElementByPos(int pos)
{
 if(pos<0 || pos>this->getLenght()-1)//位置是否合法
 {
  cout<<"Out of range"<<endl;
  return false;
 }
 Link* tmp = this->head;
 int k=-1;
 while(k<pos-1)
 {
  tmp = tmp->next;
  k++;
 }
 Link* del = tmp->next;
 tmp->next = tmp->next->next;
 delete del;
 this->length--;
 return true;
}

int LLink::getPos(int pos,LLink* link)
{

 for(int i=0;i<link->getLenght();i++)
 {
  if(pos==link->getElementByPos(i))
  {
   return i;
  }
 }
}
```

说明:完整代码跟博客说的代码可能有些地方不一样，我是从完整代码稍微改了一下写成博客的，因为我觉得那样子会比较好理解一些。