



Data Structures

线性表 Linear Lists

2024年9月13日

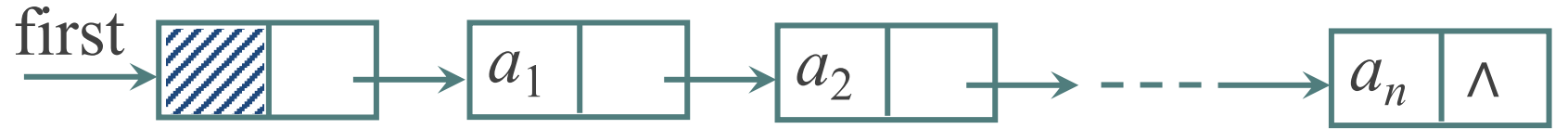
学而不厌 诲人不倦

- ➡ **2.1** 引言
- ➡ **2.2** 线性表的逻辑结构
- ➡ **2.3** 线性表的顺序存储结构及实现
- ➡ **2.4** 线性表的链接存储结构及实现
- ➡ **2.5** 顺序表和链表的比较
- ➡ **2.6** 约瑟夫环与一元多项式求和

2.4 线性表的链接存储结构及实现

【回顾】

单链表的定义



```
template <typename DataType>
struct Node
{
    DataType data;
    Node<DataType>*next;
};
```

```
template <typename DataType>
DataType LinkList<DataType>::Get(int i)
{
}
}
```

main()函数中:

```
int r[5]={1,2,3,4,5}, i, x;
LinkList<int> L(r,5);
```

```
template <typename DataType>
class LinkList
{
public:
    LinkList();
    LinkList(DataType a[],int n);
    ~LinkList();
    void PrintList();
    int Length();
    DataType Get(int i);
    int Locate(DataType x);
    void Insert(int i, DataType x);
    DataType Delete(int i);
private:
    Node<DataType> * first;
};
```

第二章 线性表

2.4 线性表的链接存储结构及实现

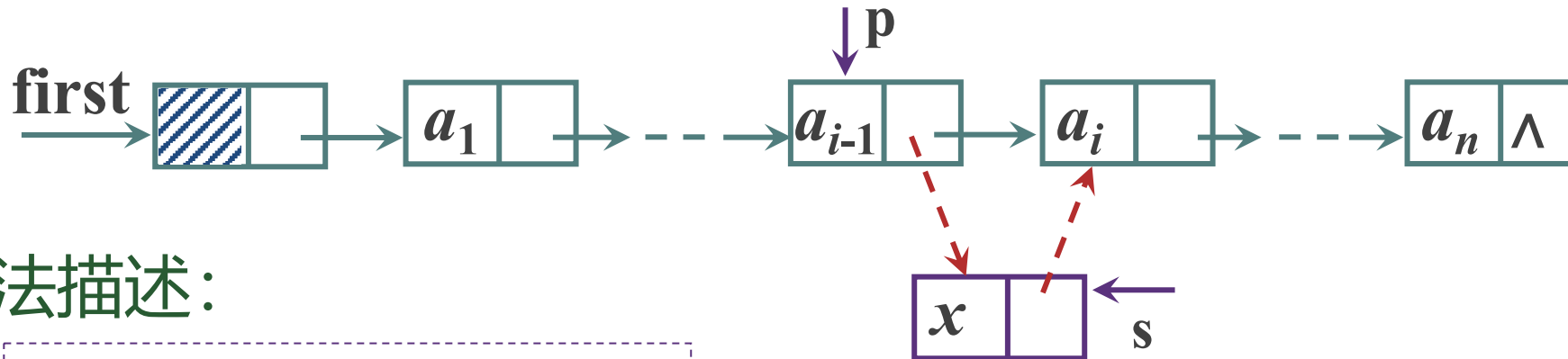
2-4-2b 单链表的实现 2

2.4 线性表的链接存储结构及实现

1. 单链表的实现——插入 (1/5)

 功能：按位置插入一个新节点。

```
template <typename DataType>
void LinkList<DataType>::Insert(int i, DataType x)
{
}
```



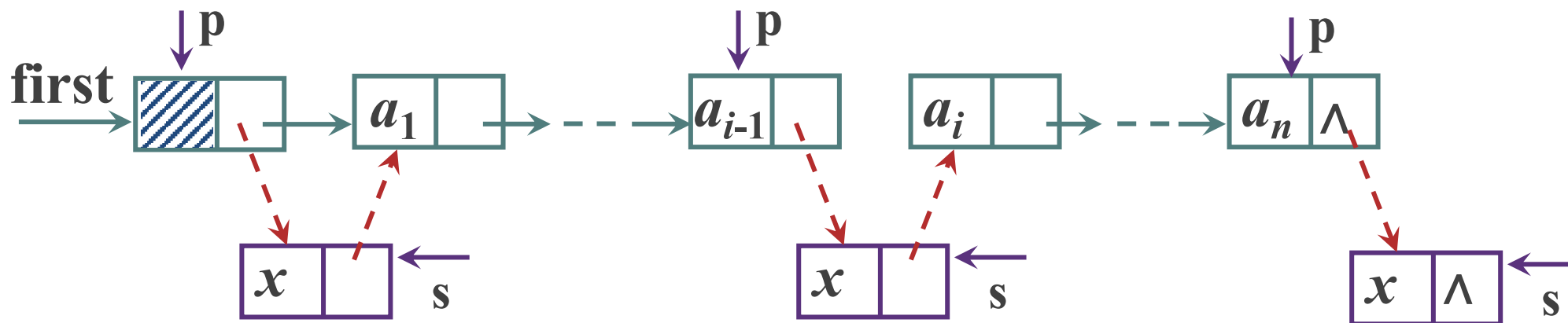
算法描述：

```
s = new Node;
s->data = x;
s->next = p->next;
p->next = s;
```

2.4 线性表的链接存储结构及实现

1. 单链表的实现——插入 (2/5)

 注意分析边界情况——表头、表尾？



$s \rightarrow \text{next} = p \rightarrow \text{next};$
 $p \rightarrow \text{next} = s;$

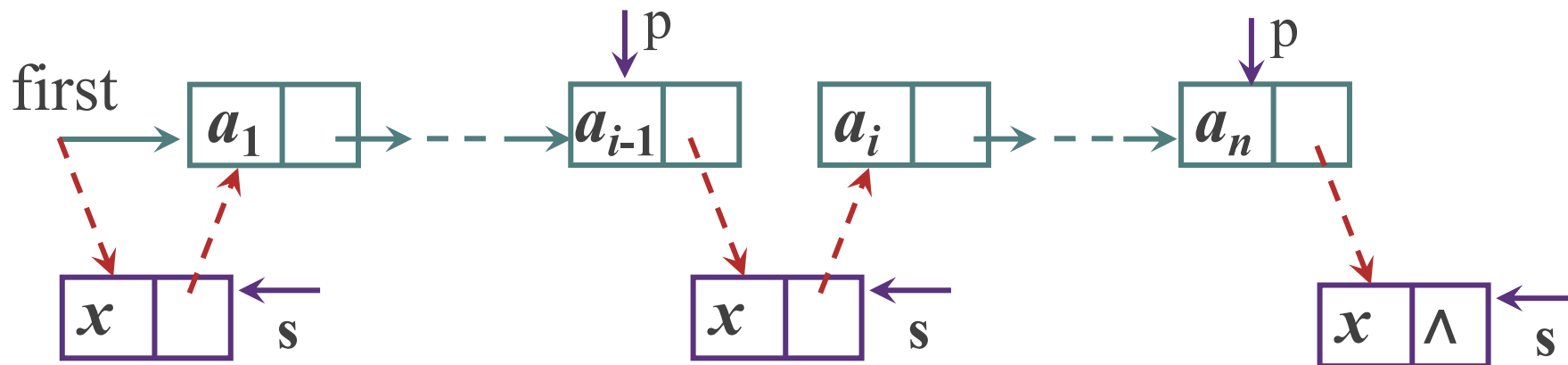


单链表没有特殊说明，都带头结点

2.4 线性表的链接存储结构及实现

1. 单链表的实现——插入 (3/5)

 如果不带头结点，会怎么样呢？



$s \rightarrow next = first;$
 $first = s;$

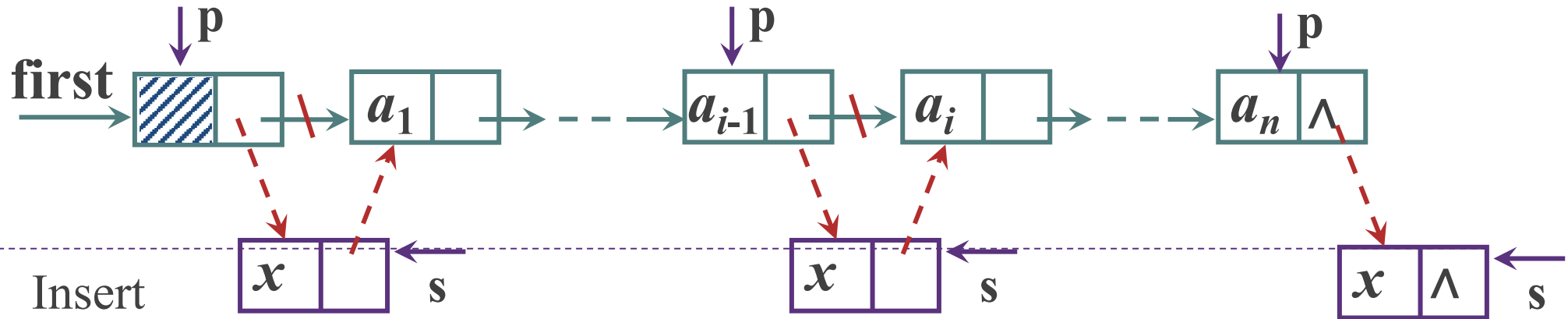
$s \rightarrow next = p \rightarrow next;$
 $p \rightarrow next = s;$

 操作不一致会导致算法增加处理步骤，而且容易出错

2.4 线性表的链接存储结构及实现

1. 单链表的实现——插入 (4/5)

```
template <typename DataType>
void LinkList<DataType>::Insert(int i, DataType x)
{
}
}
```



算法: Insert

输入: 单链表的头指针 $first$, **插入位置 i** , **待插值 x**



输出: 如果插入成功, 返回新的单链表, 否则返回插入失败信息

1. 工作指针 p 初始化为指向头结点;
2. 查找第 $i-1$ 个结点并使工作指针 p 指向该结点;
3. 若查找不成功, 说明插入位置不合理, 返回插入失败信息;
否则, 生成元素值为 x 的新结点 s , 将结点 s 插入到结点 p 之后;

2.4 线性表的链接存储结构及实现

1. 单链表的实现——插入 (5/5)

```
void LinkList<DataType> :: Insert(int i, DataType x)
{
    Node<DataType> *p = first, *s = nullptr;           //工作指针p初始化
    int count = 0;
    while (p != nullptr && count < i - 1)              //查找第 i 个结点的前驱结点
    {
        p = p->next;                                     //工作指针p后移
        count++;
    }
    if (p == nullptr) throw "插入位置错误";            //没有找到第i-1个结点
    else {
        s = new Node<DataType>; s->data = x;           //申请结点s, 数据域为x
        s->next = p->next; p->next = s;                //将结点s插入到结点p之后
    }
}
```

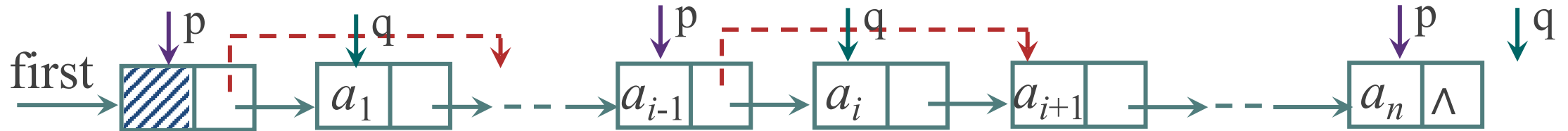
 时间复杂度? $\Rightarrow O(n)$  已知指针p $\Rightarrow O(1)$

2.4 线性表的链接存储结构及实现

2. 单链表的实现——删除 (1/3)

 功能：按位置删除特定节点。

```
template <typename DataType>
DataType LinkList<DataType>::Delete(int i)
{
}
```



 算法描述：

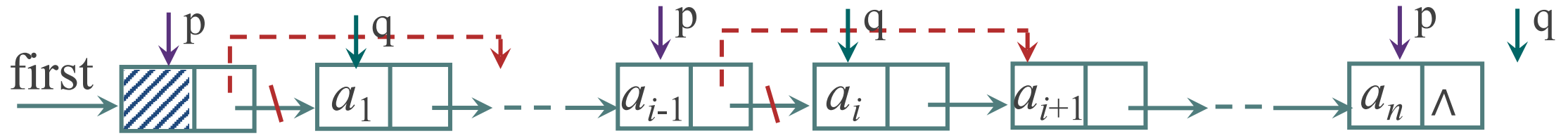
```
q = p->next;  
x = q->data;  
p->next = q->next;  
delete q;
```

 注意分析边界情况
如何删除表头和表尾？

2.4 线性表的链接存储结构及实现

```
template <typename DataType>
DataType LinkList<DataType>::Delete(int i)
{
}
```

2. 单链表的实现——删除 (2/3)



算法: Delete

输入: **单链表的头指针first** (类成员变量), 删除的位置 i

输出: 如果删除成功, 返回被删除的元素值, 否则返回删除失败信息

1. 工作指针 p 初始化; 累加器count初始化;
2. 查找第 $i-1$ 个结点并使工作指针 p 指向该结点;
3. 若 p 不存在或 p 的后继结点不存在, 则出现删除位置错误, 删除失败;
否则,
 - 3.1 存储被删结点和被删元素值;
 - 3.2 摘链, 将结点 p 的后继结点从链表上摘下;
 - 3.3 释放被删结点;

2.4 线性表的链接存储结构及实现

2. 单链表的实现——删除 (3/3)

```
DataType LinkList<DataType> :: Delete(int i)
{
    DataType x; int count = 0;
    Node<DataType> *p = first, *q = nullptr;           //工作指针p指向头结点
    while (p != nullptr && count < i - 1)               //查找第i-1个结点
    {
        p = p->next;
        count++;
    }
    if (p == nullptr || p->next == nullptr) throw "删除位置错误";
    else {
        q = p->next; x = q->data;                       //暂存被删结点
        p->next = q->next;                               //摘链
        delete q;
        return x;
    }
}
```

2.4 线性表的链接存储结构及实现

3. 单链表的实现——建立 (1/6)

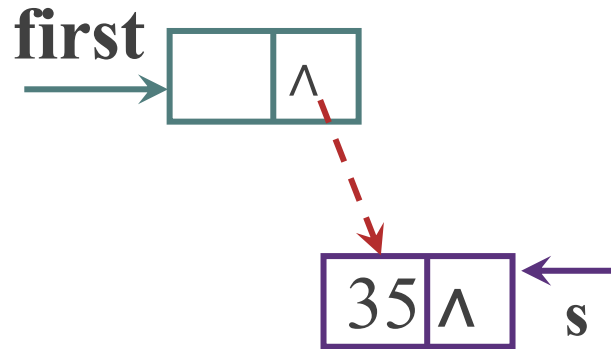
操作接口: `Node * LinkList(DataType a[], int n)`

数组 a

35	12	24	33	42
----	----	----	----	----

 **头插法: 插在头结点的后面**

初始化



插入第一个元素结点

```
first = new Node;  
first->next = nullptr;  
  
s = new Node;  
s->data = a[0];  
s->next = first->next;  
first->next = s;
```

2.4 线性表的链接存储结构及实现

3. 单链表的实现——建立 (2/6)

操作接口: `Node * LinkList(DataType a[], int n)`

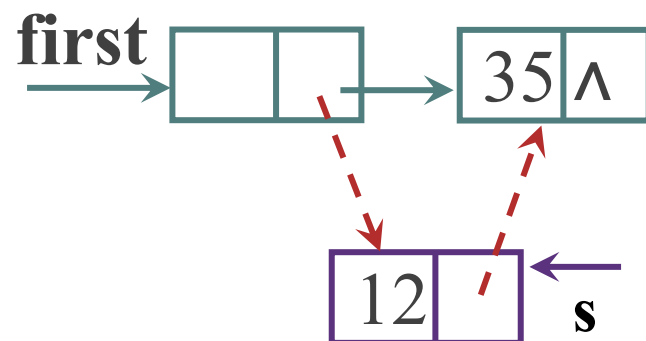
数组 a

35	12	24	33	42
----	----	----	----	----

 **头插法: 插在头结点的后面**

依次插入每一个结点

 单链表的元素顺序有什么特点?



```
s = new Node;  
s->data = a[i];  
s->next = first->next;  
first->next = s;
```

2.4 线性表的链接存储结构及实现

3. 单链表的实现——建立 (3/6)

```
template <typename DataType>
LinkedList<DataType> :: LinkedList(DataType a[ ], int n)
{
    first = new Node<DataType>; first->next = nullptr;    //初始化一个空链表
    for (int i = 0; i < n; i++)
    {
        Node<DataType> *s = nullptr;
        s = new Node<DataType>;
        s->data = a[i];
        s->next = first->next;
        first->next = s;    //将结点s插入到头结点之后
    }
}
```

2.4 线性表的链接存储结构及实现

3. 单链表的实现——建立 (4/6)

操作接口: `Node * LinkList(DataType a[], int n)`

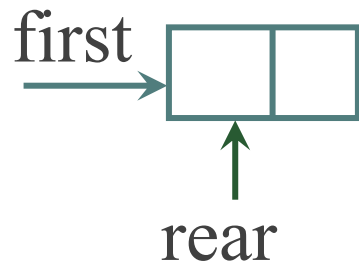
数组 a

35	12	24	33	42
----	----	----	----	----

 **尾插法: 插在尾结点的后面**

📍 为方便在尾结点后面插入结点, 设指针 `rear` 指向尾结点

初始化



```
first = new Node;  
rear = first;
```


2.4 线性表的链接存储结构及实现

3. 单链表的实现——建立 (5/6)

操作接口: `Node * LinkList(DataType a[], int n)`

数组 a

35	12	24	33	42
----	----	----	----	----

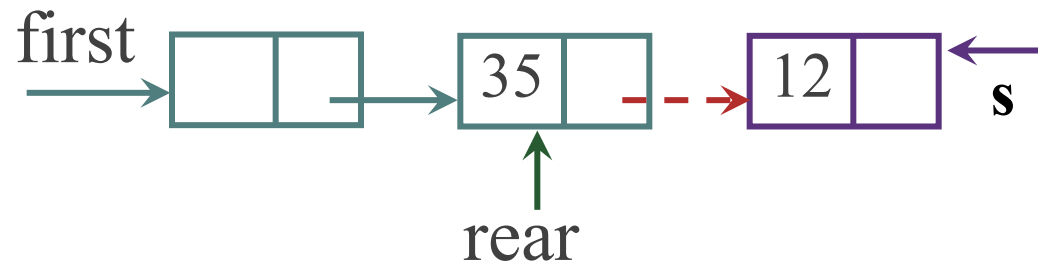


尾插法: 插在尾结点的后面

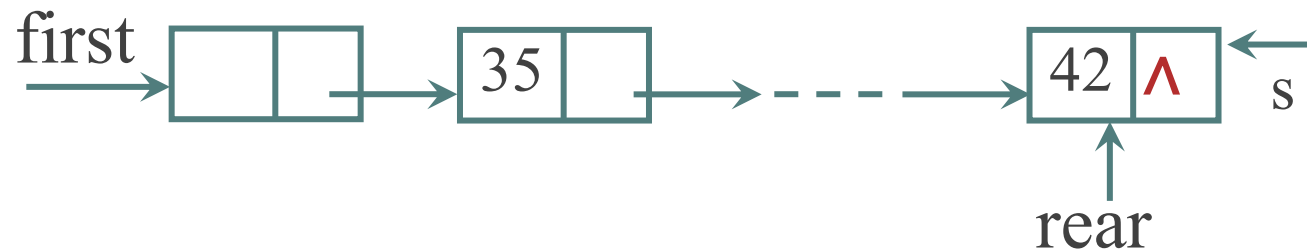
依次插入每一个结点



单链表的元素顺序有什么特点?



```
s = new Node;  
s->data = a[i];  
rear->next = s;  
rear = s;
```





2.4 线性表的链接存储结构及实现

3. 单链表的实现——建立 (6/6)

```
template <typename DataType>
LinkedList<DataType> :: LinkedList(DataType a[ ], int n)
{
    first = new Node<DataType>;           //生成头结点
    Node<DataType> *r = first, *s = nullptr; //尾指针初始化
    for (int i = 0; i < n; i++)
    {
        s = new Node<DataType>;    s->data = a[i];
        r->next = s;
        r = s;                     //将结点s插入到终端结点之后
    }
    r->next = nullptr;             //单链表建立完毕，将终端结点的指针域置空
}
```

小结

- 📖 掌握单链表的逻辑结构特点和定义方式
- 📖 单链表基本操作：建立、遍历、查找、插入、删除
- 📖 能够从时间和空间复杂度的角度分析基本操作的实现算法。



作业

1. 简答：引入头结点的原因？
2. 编程：逆置一个单链表为一个新表。

实验二、链式存储结构线性表的建立及操作

一、实验目的

1. 掌握线性表的链式存储结构的特点，理解链式存储结构表示线性表的方法。
2. 掌握链式存储结构线性表数据元素类型定义的格式与方法。
3. 掌握单链表建立、遍历、查找、新元素插入、及元素删除的原理与方法。
4. 用C++语言实现单链表，并上机调试。

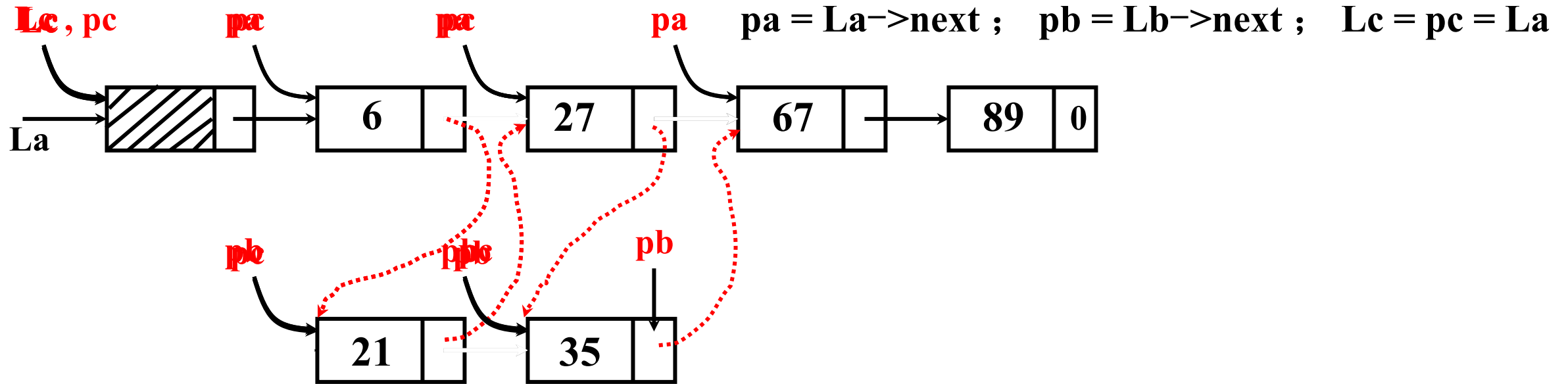
二、实验内容

1. 设计C++类及相关方法，用于维护单链表。
2. 写出建立单链表，并向单链表中输入数据的函数。
3. 实现单链表的建立、遍历、查找、新元素插入、及元素删除，写出输入及输出的内容。
4. 将两个有序单链表合并为一个有序单链表（**扩展内容**）
5. 双链表及循环链表的实现（**扩展内容**）

实验时间： 第4周周四晚 19:00-21:00
实验地点： 格物楼A216

扩展

算法： 将两个有序单链表合并为一个有序单链表



```
while ( pa && pb ) {
    if ( pa->data <= pb->data ) {
        pc->next = pa ; pc = pa ; pa = pa->next ;
    }
    else { pc->next = pb ; pc = pb ; pb = pb->next ; }
}
```

//处理剩余部分

$pc \rightarrow next = pa ? pa : pb ;$

$free(Lb) ;$



Thank You !

Q & A