



Data Structures

线性表 Linear Lists

2024年9月10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

第二章 线性表

2.4 线性表的链接存储结构及实现

2-4-1 单链表的存储结构

2.4 线性表的链接存储结构及实现

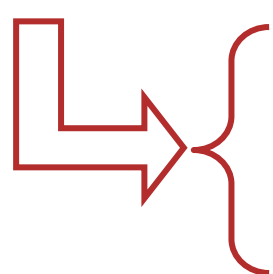
单链表的引入

顺序表 \Rightarrow 静态存储分配 \Rightarrow 事先确定容量

单链表 \Rightarrow 动态存储分配 \Rightarrow 运行时分配空间

✦ 单链表：线性表的链接存储结构

✦ 存储思想：用一组**任意**的存储单元存放线性表

 **连续**
不连续
零散分布

2.4 线性表的链接存储结构及实现

单链表的存储方法



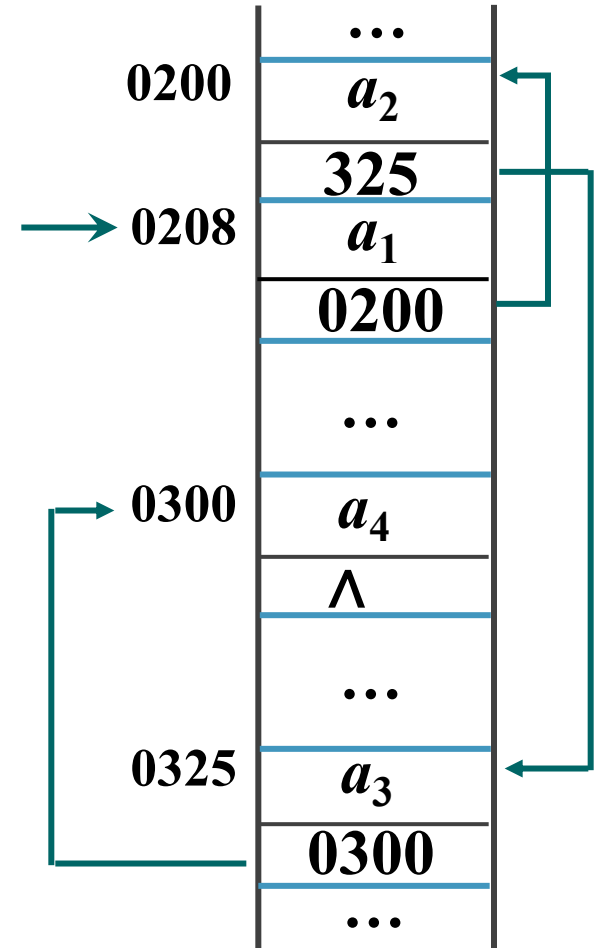
存储特点:

1. 逻辑次序和物理次序**不一定**相同
2. 元素之间的逻辑关系用**指针**表示

例: (a_1, a_2, a_3, a_4) 的存储示意图

- ✦ 单链表: 线性表的链接存储结构
- ✦ 存储思想: 用一组**任意**的存储单元存放线性表

连续
不连续
零散分布



2.4 线性表的链接存储结构及实现

单链表的存储方法

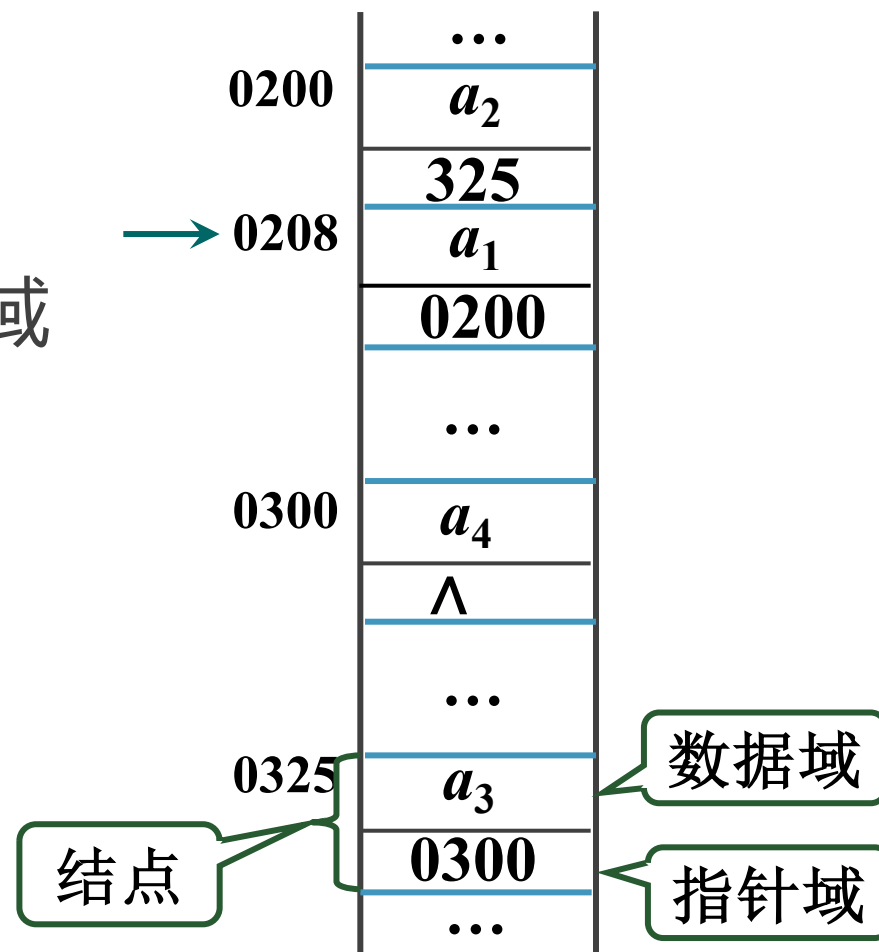
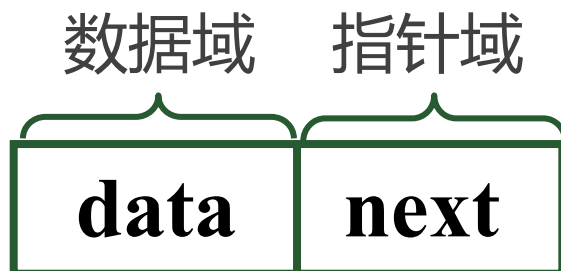
📖 观察1：单链表由若干结点构成

📖 观察2：单链表的结点只有一个指针域

data：存储数据元素

next：存储指向后继结点的地址

单链表的结点结构



2.4 线性表的链接存储结构及实现

单链表的存储方法

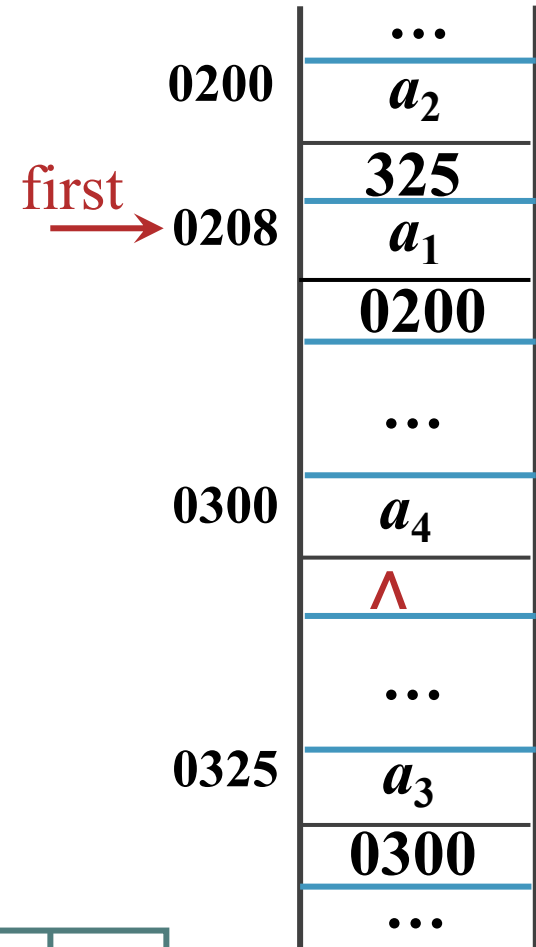
 头指针：指向**第一个结点**的存储地址

 尾标志：**终端结点**的指针域为空

 空表和非空表不统一，有什么缺点？

空表 $\text{first} = \text{NULL}$

非空表



2.4 线性表的链接存储结构及实现

单链表的存储方法

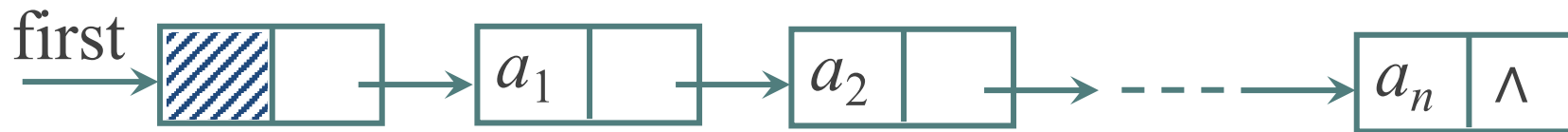
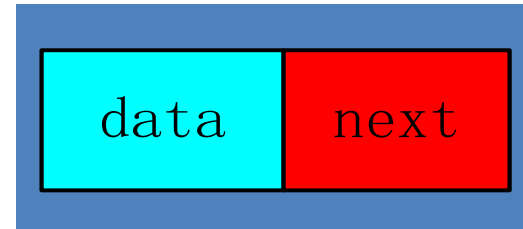
- 📎 头指针：指向**第一个结点**的存储地址
- 📎 尾标志：**终端结点**的指针域为空
- 📎 头结点：在第一个元素结点之前**附设**一个类型相同的结点
头结点简化了对边界的处理——插入、删除、构造等



2.4 线性表的链接存储结构及实现

单链表的结点结构定义

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



2.4 线性表的链接存储结构及实现

指针的相关问题

📎 设指针 p 指向某个Node类型的结点

该结点用 $*p$ 来表示, $*p$ 为结点变量。将“指针 p 所指结点”简称为“结点 p ”

🕒 如何引用结点 p 的数据域 (指针域) ?

$*p.data$



$p->data$

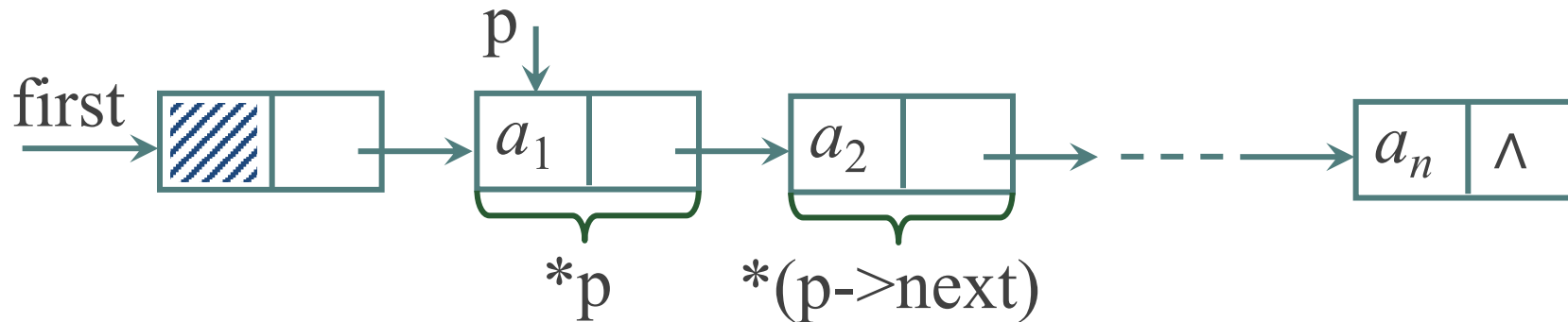
$*p.next$



$p->next$



如何表示结点 p 的下一个结点?



2.4 线性表的链接存储结构及实现

指针的相关问题

`new` 类型名T (初值列表)

功能:

申请用于存放T类型对象的内存空间，并依初值列表赋以初值

结果值:

成功: T类型的指针，指向新分配的内存

失败: 0 (NULL)

```
int *p1 = new int;  
或 int *p1 = new int(10);
```

`delete` 指针P

```
delete p1;
```

功能: 释放指针P所指向的内存。P必须是new操作的返回值

2.4 线性表的链接存储结构及实现

函数的相关问题

值传递

把实参的值传送给函数局部工作区相应的副本中，函数使用这个副本执行必要的功能。函数修改的是副本的值，实参的值不变

```
#include <iostream>
void swap(float m,float n)
{
    float temp;
    temp=m;
    m=n;
    n=temp;
}
```

```
void main()
{
    float a,b;
    cin>>a>>b;
    swap(a,b);
    cout<<a<<endl<<b<<endl;
}
```

2.4 线性表的链接存储结构及实现

函数的相关问题

地址传递

```
#include <iostream>
void swap(float *m, float *n)
{
    float temp;
    temp=*m;
    *m=*n;
    *n=temp;
}
```

```
void main()
{
    float a,b;
    cin>>a>>b;
    p1=&a;
    p2=&b;
    swap(p1, p2);
    cout<<a<<endl<<b<<endl;
}
```

2.4 线性表的链接存储结构及实现

函数的相关问题

引用：它用来给一个对象提供一个替代的名字

引用类型作为参数

```
#include <iostream>
void swap(float &m, float &n)
{
    float temp;
    temp=m;
    m=n;
    n=temp;
}
```

```
void main()
{
    float a,b;
    cin>>a>>b;
    swap(a,b);
    cout<<a<<endl<<b<<endl;
}
```

(1) 传递引用给函数与传递指针的效果是一样的，形参变化实参也发生变化。

(2) 引用类型作形参，在内存中并没有产生实参的副本，它直接对实参操作；

当参数传递的数据量较大时，用引用比用一般变量传递参数的时间和空间效率都好。

第二章 线性表

2.4 线性表的链接存储结构及实现

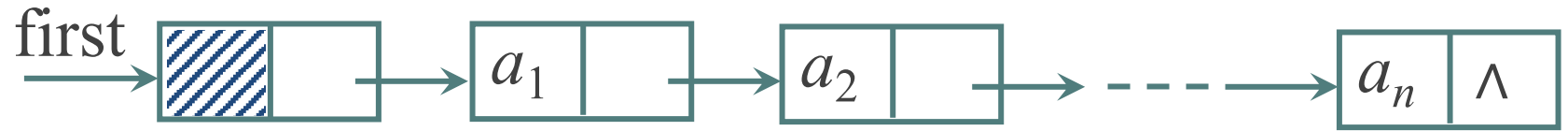
2-4-2a 单链表的实现 1

2.4 线性表的链接存储结构及实现

LinkList-base.cpp



单链表的定



float data; ?

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

```
int LinkList::Get(int i)
{
}
}
```

main()函数中:

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

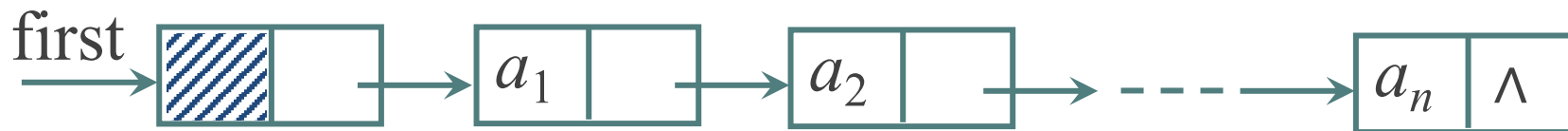
```
class LinkList
{
public:
    LinkList();
    LinkList(int a[],int n);
    ~LinkList();
    void PrintList();
    int Length();
    int Get(int i);
    int Locate(int x);
    void Insert(int i, int x);
    int Delete(int i);
private:
    Node * first;
};
```


2.4 线性表的链接存储结构及实现

LinkList-pro.cpp



单链表的定



```
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);
```

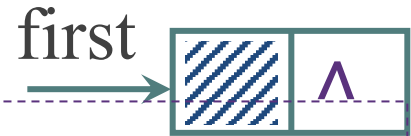
```
template <typename DataType>
class LinkList
{
public:
    LinkList();
    LinkList(DataType a[],int n);
    ~LinkList();
    void PrintList();
    int Length();
    DataType Get(int i);
    int Locate(DataType x);
    void Insert(int i, DataType x);
    DataType Delete(int i);
private:
    Node<DataType>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

2.4 线性表的链接存储结构及实现

1. 单链表的实现——初始化



初始化一个单链表要完成哪些工作呢？



```
template <typename DataType>
LinkedList<DataType> :: LinkedList( )
{
    first = new Node<DataType>;
    first->next = nullptr;
}
```

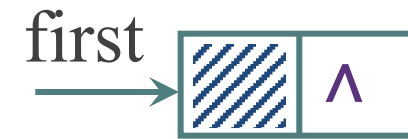
//生成头结点

//头结点的指针域置空

2.4 线性表的链接存储结构及实现

2. 单链表的实现——判空

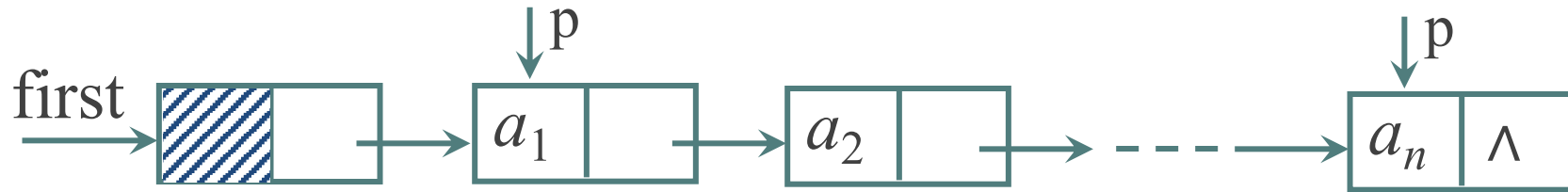
 空单链表满足什么条件呢?




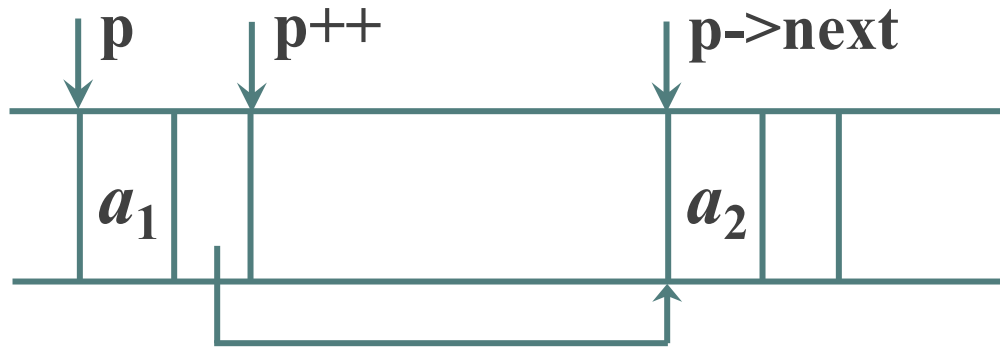
```
template <typename DataType>
int LinkList<DataType> :: Empty( )
{
    if (first->next == nullptr) return 1
    else return 0;
}
```

2.4 线性表的链接存储结构及实现

3. 单链表的实现——遍历 (1/4)



 如何实现工作指针后移？ $p++$ 能正确实现后移吗？

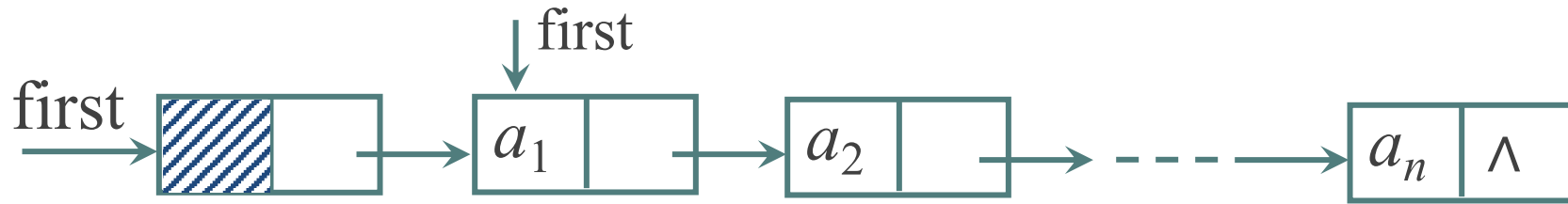



$p = p \rightarrow \text{next}$

核心操作：工作指针后移

2.4 线性表的链接存储结构及实现

3. 单链表的实现——遍历 (2/4)



 first 指向头结点

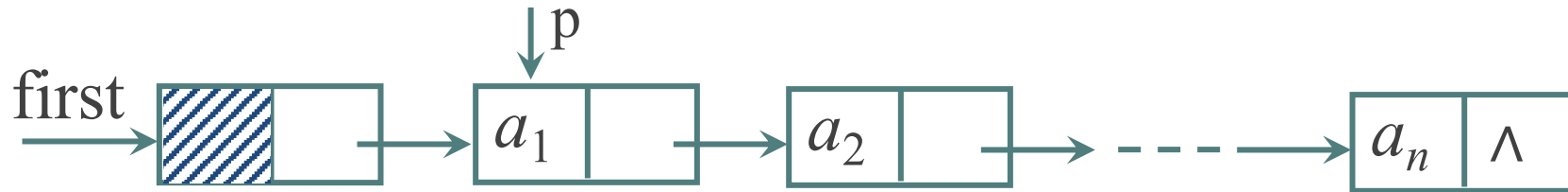
$\left\{ \begin{array}{l} \text{first 修改前: } (a_1, a_2, \dots, a_n) \\ \text{first 修改后: } (a_2, \dots, a_n) \end{array} \right.$

 为什么设置工作指针？通过头指针后移扫描单链表会有什么后果？

 单链表头指针的作用是标识单链表的开始，通常不修改头指针

2.4 线性表的链接存储结构及实现

3. 单链表的实现——遍历 (3/4)



 如何描述遍历的基本过程?  伪代码——梳理思路的好工具!

输入：无

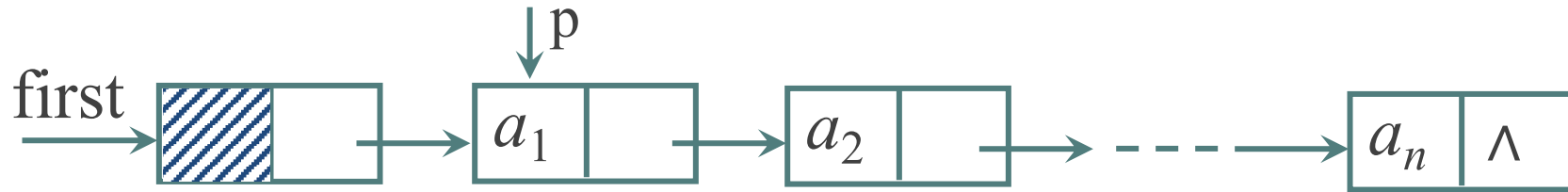
功能：遍历单链表 PrintList

输出：单链表的各个数据元素

1. 工作指针 p 初始化;
2. 重复执行下述操作, 直到指针 p 为空:
 - 2.1 输出结点 p 的数据域;
 - 2.2 工作指针 p 后移;

2.4 线性表的链接存储结构及实现

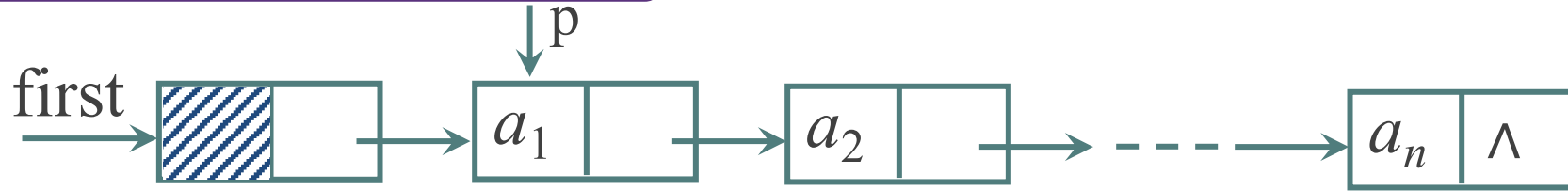
3. 单链表的实现——遍历 (4/4)



```
template <typename DataType>
void LinkList<DataType> :: PrintList( )
{
    Node<DataType> *p = first->next;           //工作指针p初始化
    while (p != nullptr)
    {
        cout << p->data << "\\t";
        p = p->next;                           //工作指针p后移，注意不能写作p++
    }
    cout << endl;
}
```

2.4 线性表的链接存储结构及实现

单链表算法的设计模式



单链表算法的设计模式：通过工作指针的反复后移扫描链表

```
p = first->next;  
while (p != nullptr)
```

```
{  
    访问结点 p 进行的操作  
    p = p->next;  
}
```

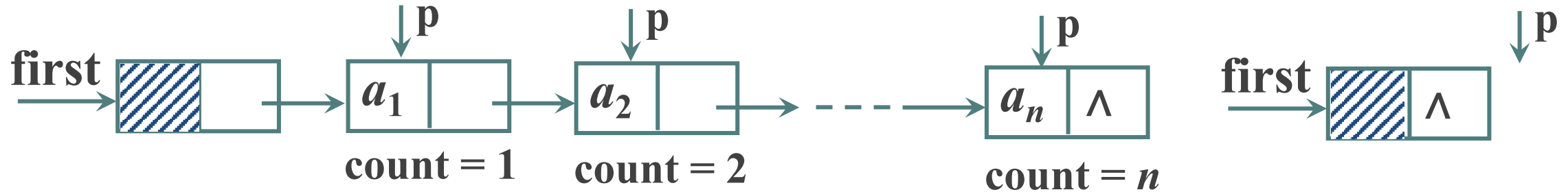
退出循环的操作

```
// 或 p = first;, 工作指针 p 初始化  
// 或 p->next != nullptr, 扫描单链表
```

```
//工作指针后移
```


2.4 线性表的链接存储结构及实现

4. 单链表的实现——长度

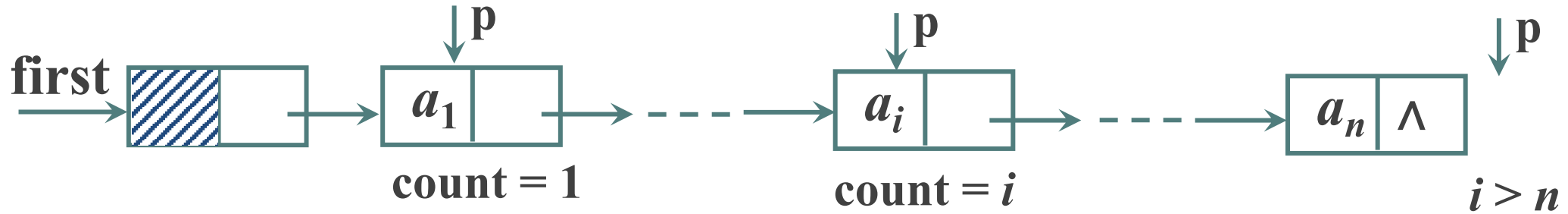


```
int LinkList<DataType> :: Length()  
{  
    Node *p = first->next;  
    int count = 0;  
    while (p != nullptr)  
    {  
        count++;  
        p = p->next;  
    }  
    return count;  
}
```

📍 注意count的初值和返回值之间的关系

2.4 线性表的链接存储结构及实现

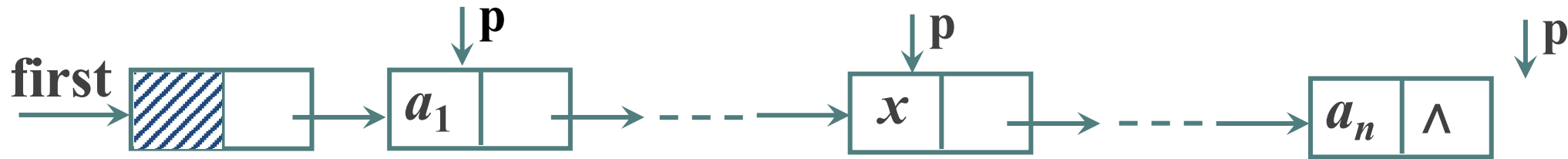
5. 单链表的实现——按位查找



```
DataType LinkList<DataType> :: Get(int i)
{
    Node<DataType> *p = first->next;    //工作指针p初始化
    int count = 1;                       //累加器count初始化
    while (p != nullptr && count < i)
    {
        p = p->next;                    //工作指针p后移
        count++;
    }
    if (p == nullptr) throw "查找位置错误";
    else return p->data;
}
```

2.4 线性表的链接存储结构及实现

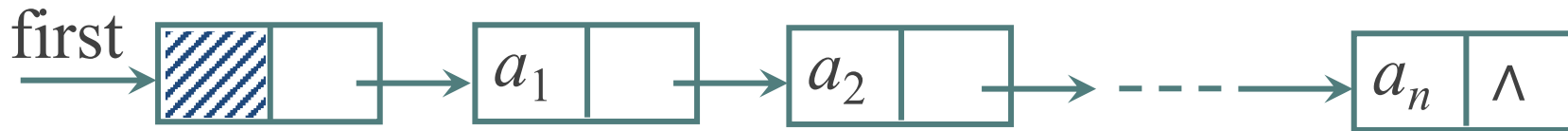
6. 单链表的实现——按值查找



```
int LinkList<DataType> :: Locate(DataType x)
{
    Node<DataType> *p = first->next;    //工作指针p初始化
    int count = 1;                       //累加器count初始化
    while (p != nullptr)
    {
        if (p->data == x) return count;  //查找成功，结束函数并返回序号
        p = p->next;
        count++;
    }
    return 0;                            //退出循环表明查找失败
}
```

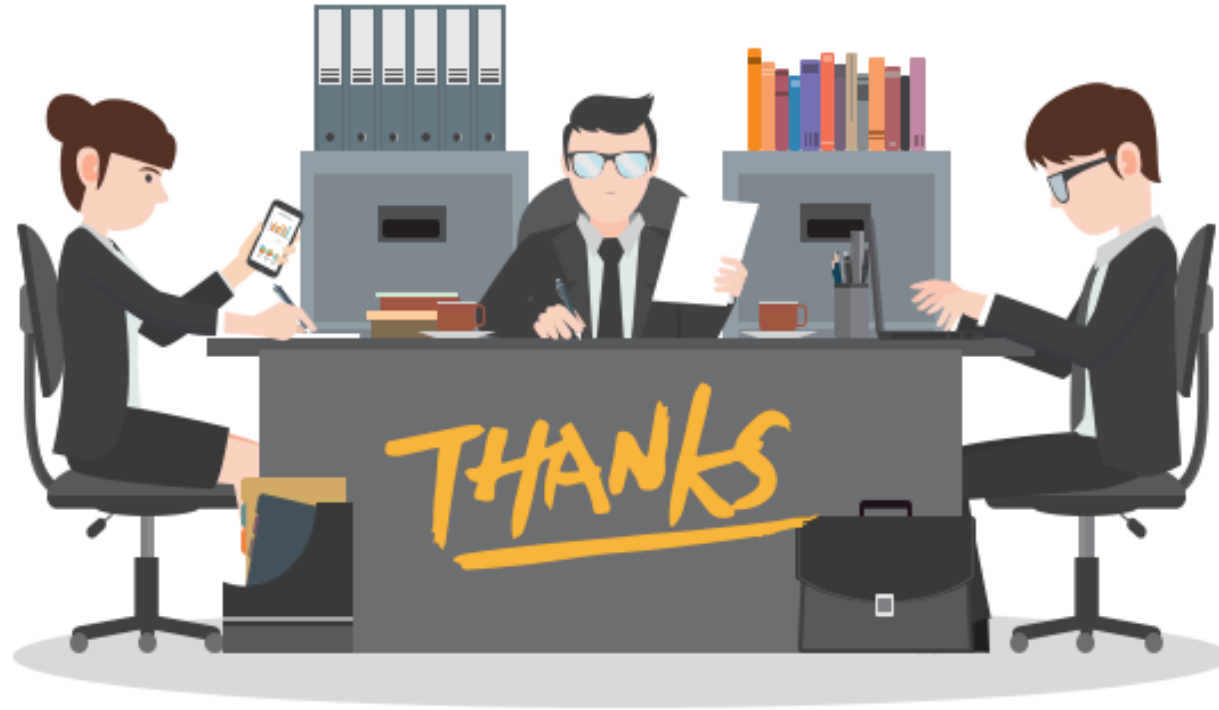
小结

- 📖 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。



Thank You !

Q & A