线性表的单链表存储及与顺序表的比较

学习目标

- 掌握线性表的单链表存储和相关操作
- 熟练分析相关存储算法的时间复杂度
- 熟知顺序表和单链表存储的各自优缺点

单链表的引入

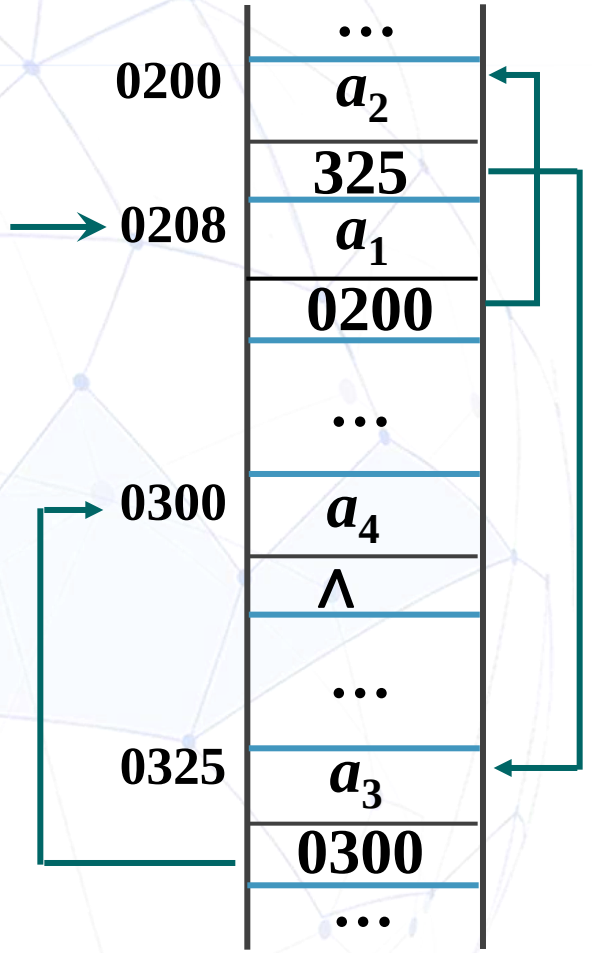
- 单链表：线性表的链接存储结构
- 存储思想：用一组**任意**的存储单元存放线性表

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

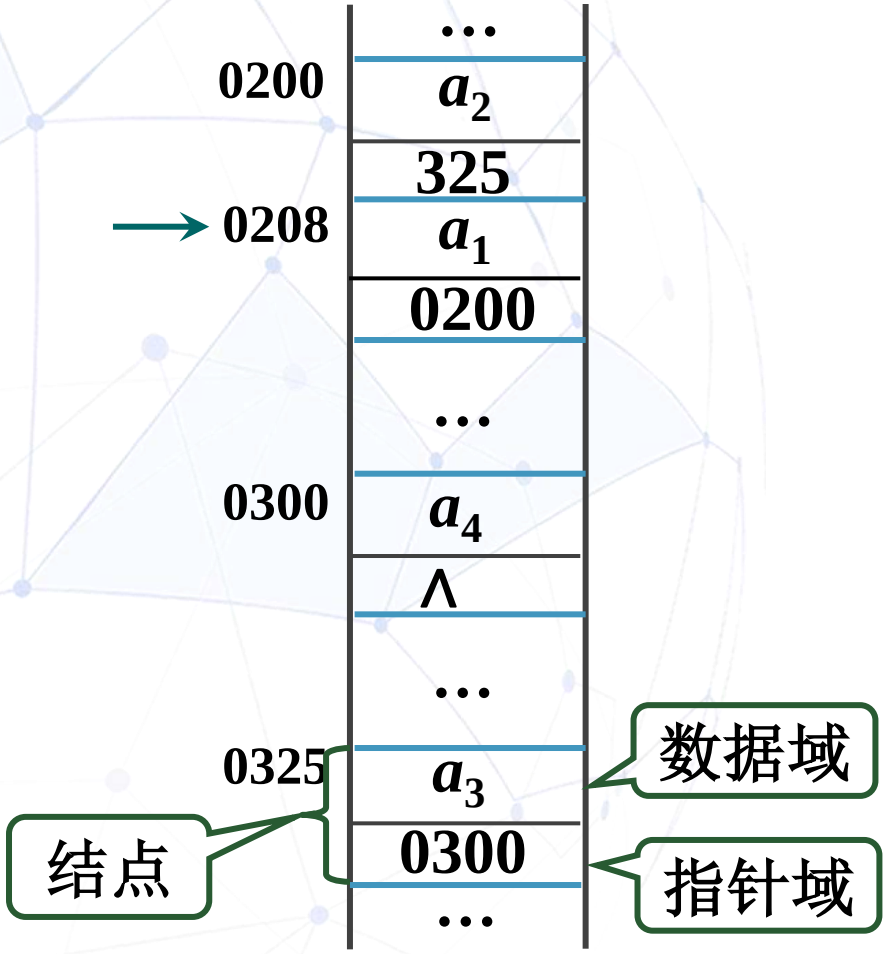
单链表的存储方法

观察1：单链表由若干结点构成

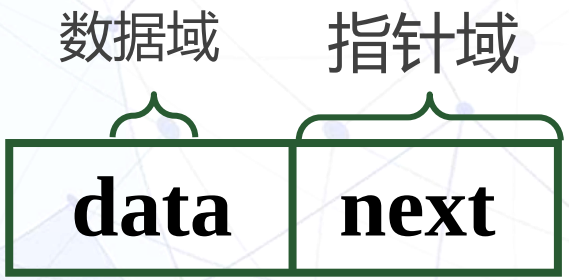
观察2：单链表的结点只有一个指针域

data：存储数据元素

next：存储指向后继结点的地址

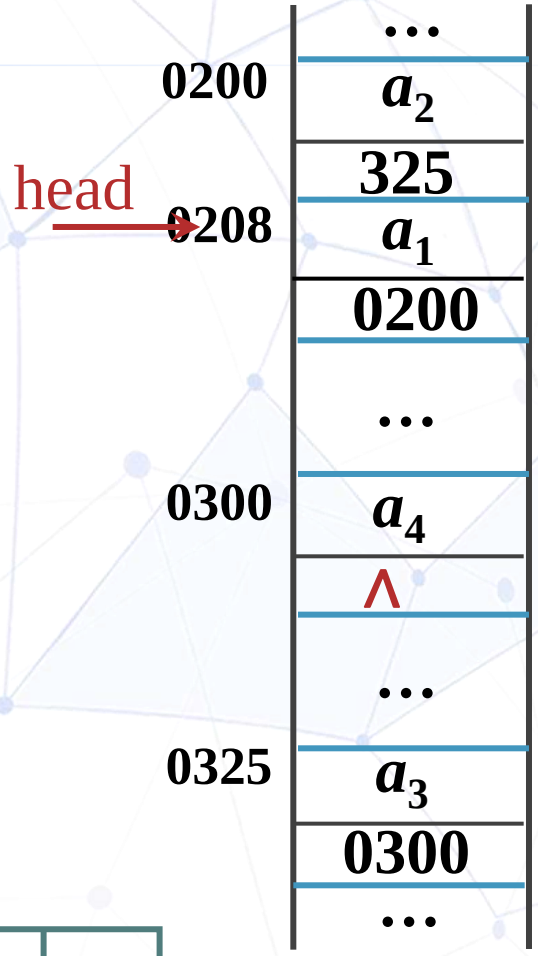


单链表的结点结构



单链表的存储方法

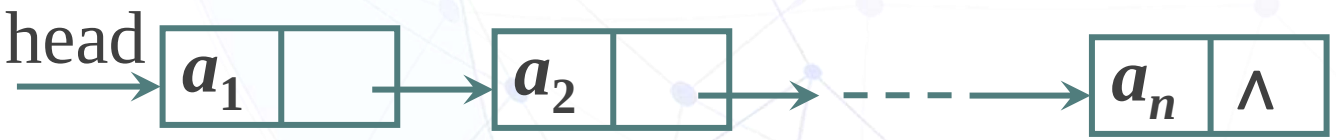
- 头指针：指向**第一个结点**的存储地址
- 尾标志：**终端结点**的指针域为空



空表和非空表不统一，有什么缺点？

空表 head = NULL

非空表



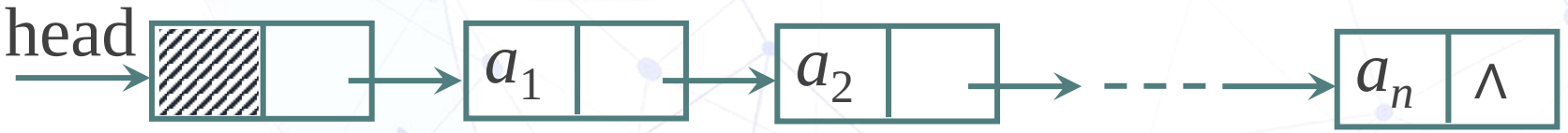
单链表的存储方法

- 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

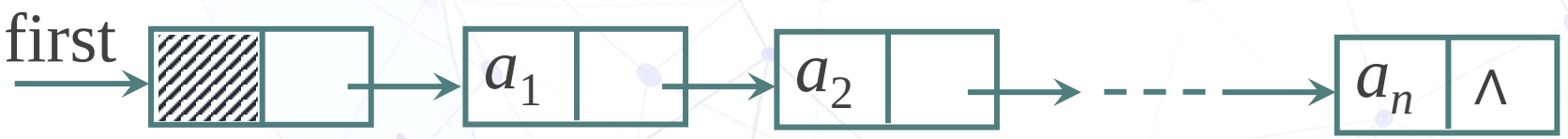
空表

非空表



单链表的结点结构定义

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



单链表类定义（C++）

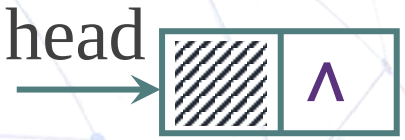
```
template <class DataType>
class LinkList
{
public:
        LinkList( );                          //无参构造函数，建立只有头结点的空链表
        ~LinkList( );                         //析构函数
        InitList();                           //初始化
       void TraverseList( );                  //遍历操作，按序号依次输出各元素
       DataType get(int i)                    //按位置查找
        int Search(DataType x);               //按值查找。在单链表中查找值为x的元素序号
        void Insert(int i, DataType x);       //插入操作，第i个位置插入元素值为x的结点
        DataType Delete(int i);               //删除操作，在单链表中删除第i个结点
        int Length();                         //获得实际长度
private:
        Node<DataType> *head;                 //单链表的头指针
};
```

单链表的基本操作

- 初始化
- 遍历
- 求表长
- 查找
- 插入
- 删除

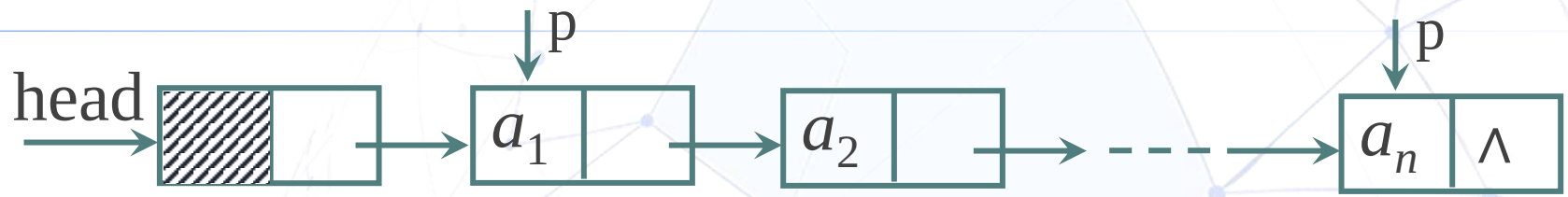
单链表的实现——初始化

初始化一个单链表要完成哪些工作呢?

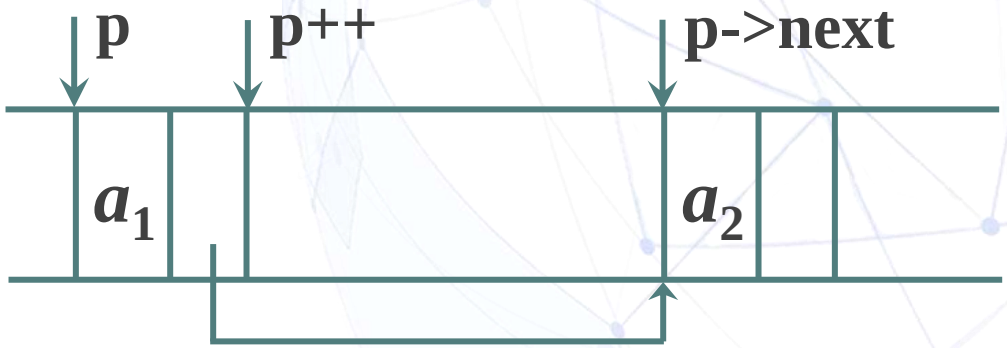



```
template <typename DataType>
LinkList<DataType> :: InitList( )
{
    head = new Node<DataType>;          //生成头结点
    head->next = nullptr;               //头结点的指针域置空
}
```

单链表的实现——遍历



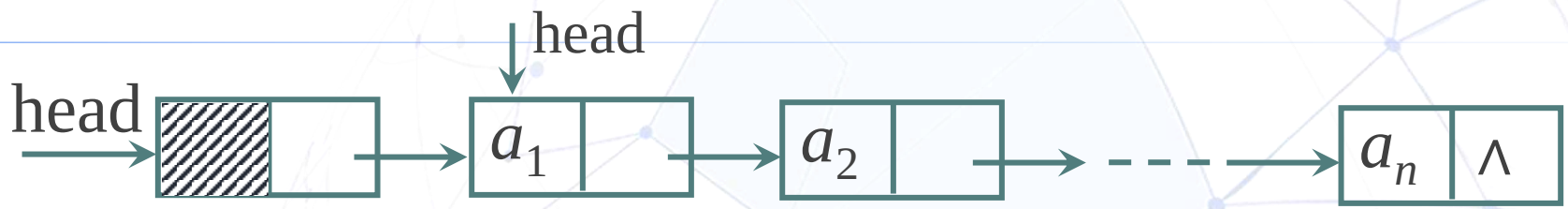
如何实现工作指针后移？p++ 能正确实现后移吗？



p = p->next

核心操作：工作指针后移

单链表的实现——遍历

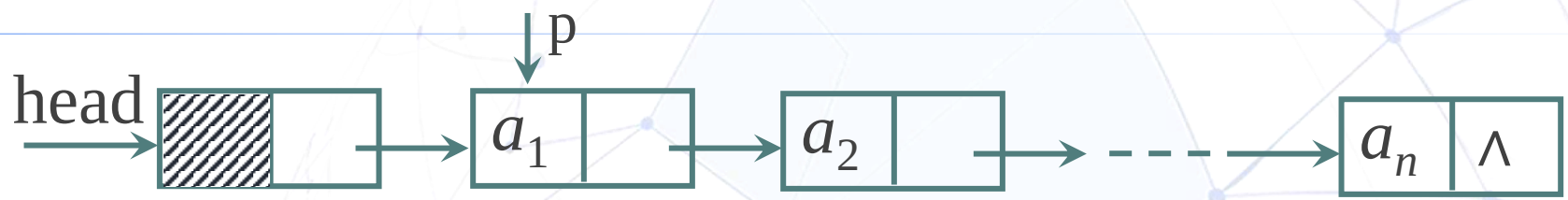


head 指向头结点

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

单链表的实现——遍历

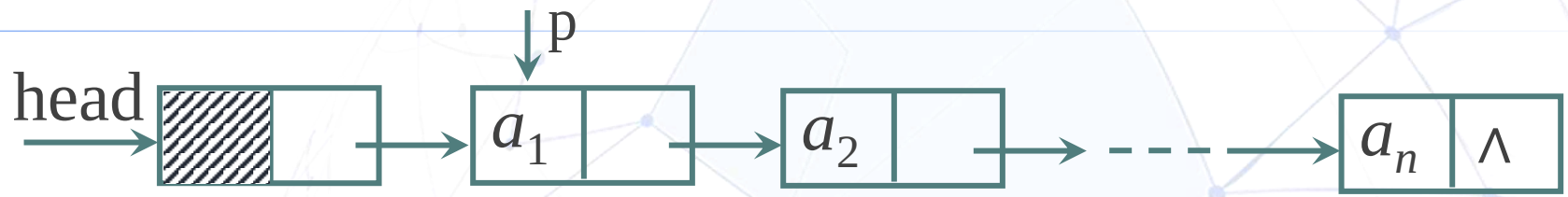


如何描述遍历的基本过程？ ⇨ 伪代码——梳理思路的好工具！

输入：无
功能：遍历单链表 TraverseList
输出：单链表的各个数据元素

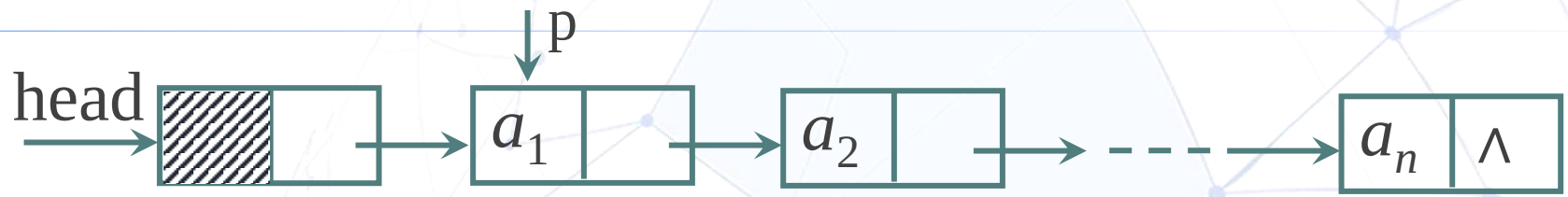
1. 工作指针 p 初始化;
2. 重复执行下述操作，直到指针 p 为空：
   2.1 输出结点 p 的数据域;
   2.2 工作指针 p 后移;

单链表的实现——遍历




```
template <typename DataType>
void LinkList<DataType> :: TraverseList( )
{
    Node<DataType> *p = head->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;
}
```

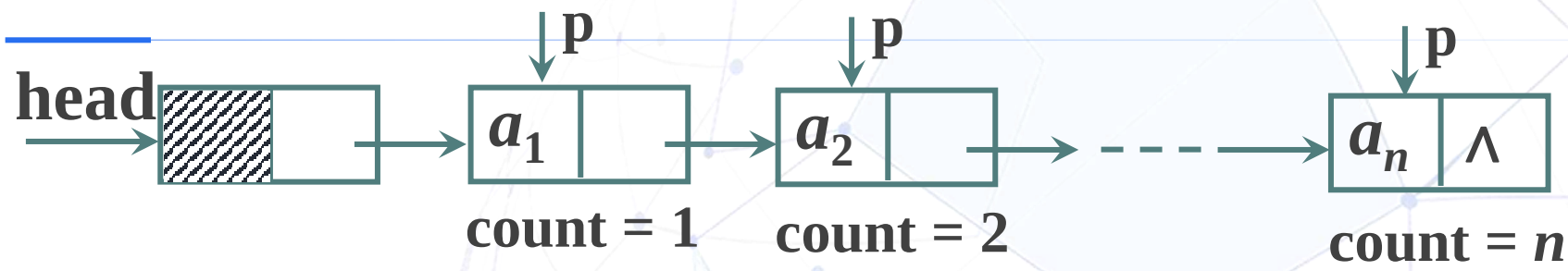
单链表算法的设计模式



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 = head->next;                    // 或p = head;，工作指针 p 初始化
while (p != nullptr)               // 或p->next != nullptr，扫描单链表
{
    访问结点 p 进行的操作
    p = p->next;                   //工作指针后移
}
退出循环的操作
```

单链表的实现——求长度



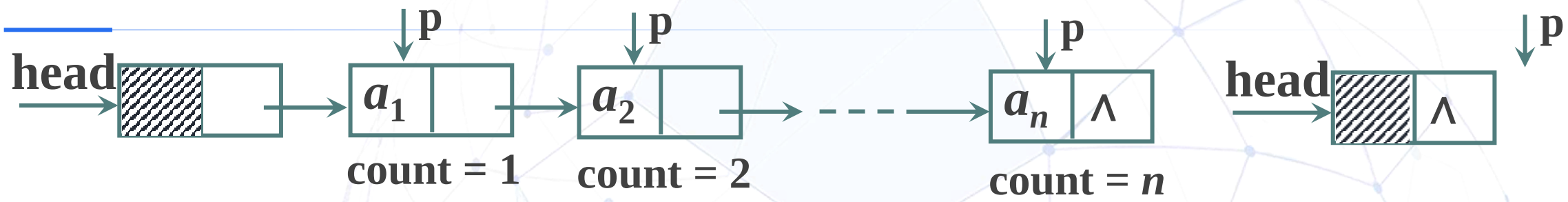
即求单链表元素个数

（1）设一个移动指针p和计数器counter，初始化

（2）p逐步往后移，同时计数器counter加1

（3）当后面不再有结点时，counter的值就是结点个数，即表长。

单链表的实现——长度




```
int LinkList<DataType> :: Length( )
{
    Node *p = head->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

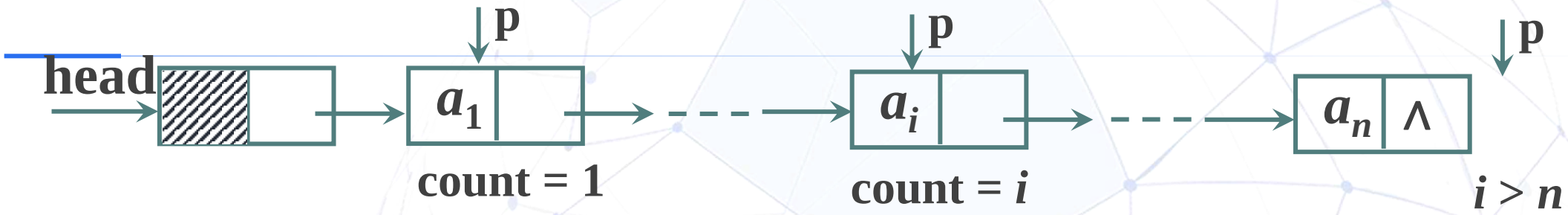
2.查找

分按序号查找Get (list, i)、按值查找Search(list, x)

（1）按序号查找步骤：

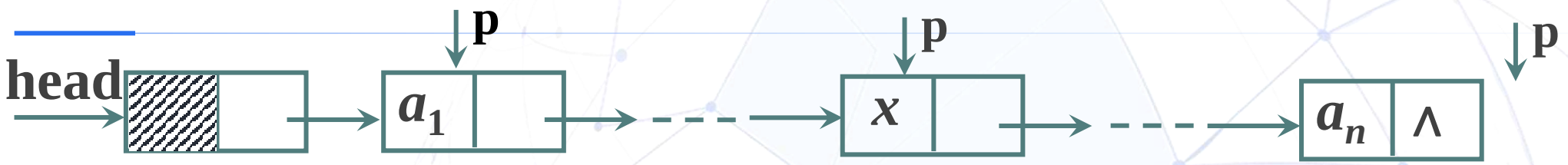
- 从链表的第一个元素结点起，判断当前结点是否是第i个；
- 若是，则返回该结点的值，否则继续后一个，直到表结束为止。
- 如果没有第i个结点则返回错误码(ErrorCode)。

单链表的实现——按位查找




```
DataType LinkList<DataType> :: Get(int i)
{
    Node<DataType> *p = head->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;
}
```

单链表的实现——按值查找




```
int LinkList<DataType> :: Search(DataType x)
{
    Node<DataType> *p = head->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
}
```

3. 插入

在list的第i个位置上插入元素x

步骤：

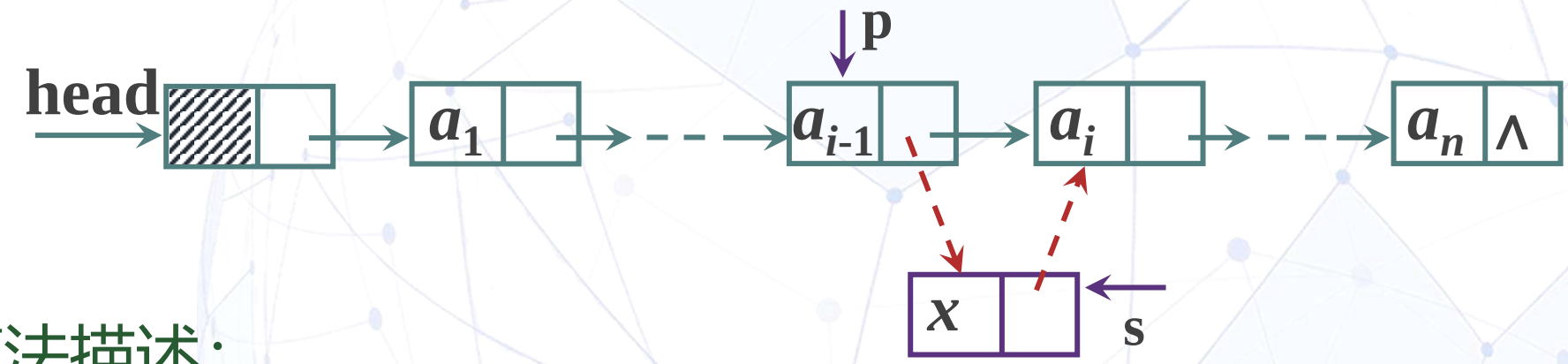
（1）找到第i-1个结点；

（2）若存在，则申请一个新结点的空间并填上相应值x，然后将新结点插到第i-1个结点之后；

（3）如果不存在则直接退出：

单链表的实现——插入

如何实现结点a_{i-1}、x 和 a_i之间逻辑关系的变化?

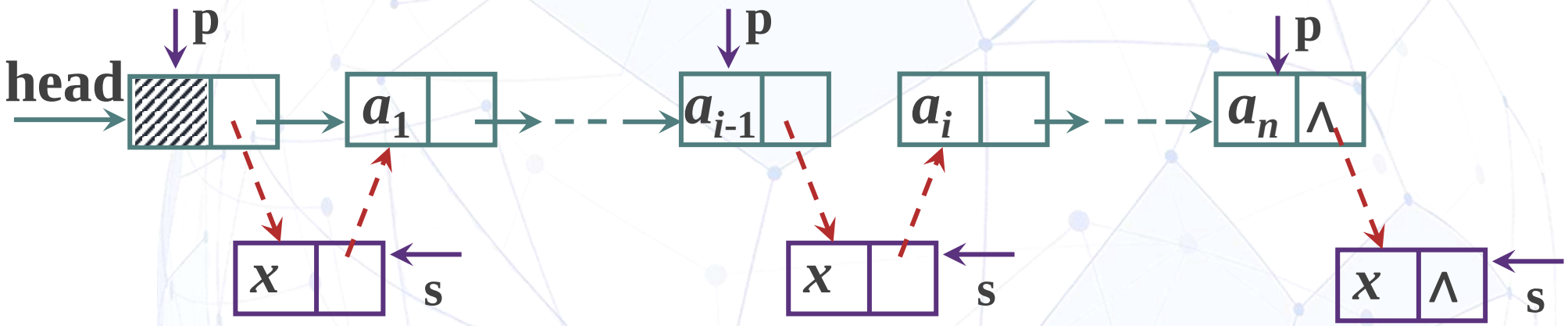


算法描述:

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

单链表的实现——插入

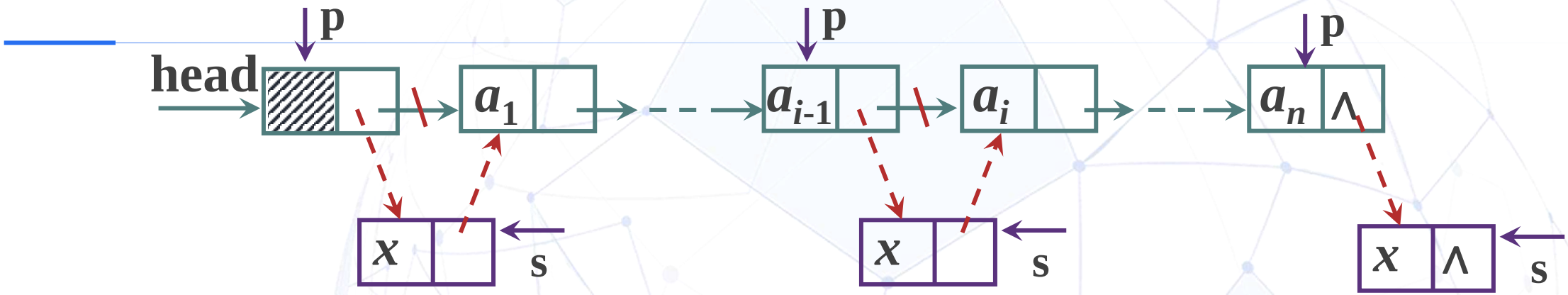
注意分析边界情况——表头、表尾?




```
s->next = p->next;
p->next = s;
```

单链表没有特殊说明，都带头结点

单链表的实现——插入



算法：Insert

输入：单链表的头指针head，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；

单链表的实现——插入

```
void LinkList<DataType> :: Insert(int i, DataType x)
{
    Node<DataType> *p = head, *s = nullptr;          //工作指针p初始化
    int count = 0;
    while (p != nullptr && count < i - 1)            //查找第i – 1个结点
    {
        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? ⇨ $O(n)$　已知指针p ⇨ $O(1)$

4. 删除

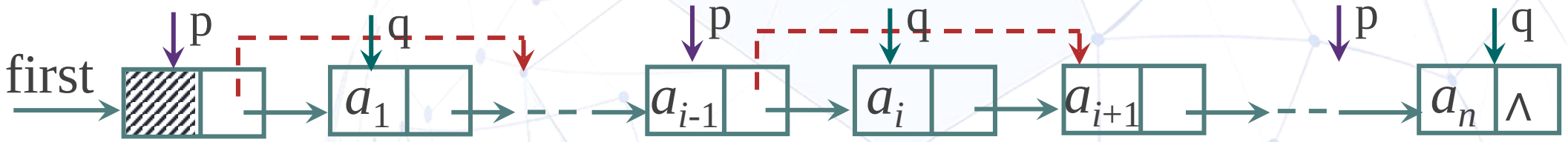
在单链表中删除指定位序i的元素

步骤：

- 找到被删除结点的前一个元素
- 再删除结点并释放空间。

单链表的实现——删除

如何实现结点a_{i-1}、a_i和a_{i+1}之间逻辑关系的变化?



注意分析边界情况——删除表头和表尾!

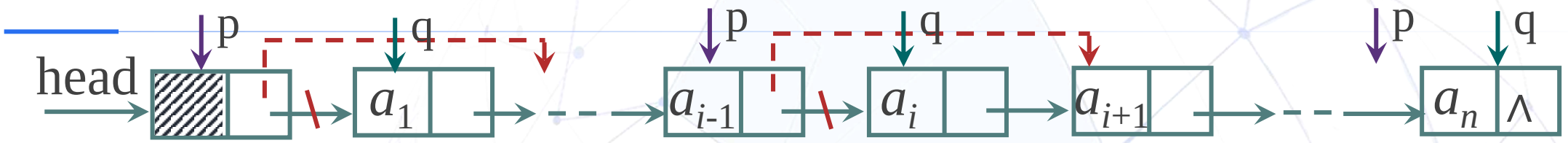
算法描述:

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

表尾的特殊情况:

虽然被删结点不存在,
但其前驱结点却存在!

单链表的实现——删除



算法：Delete

输入：单链表的头指针head，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；

单链表的实现——删除

```
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;
    }
}
```

单链表的实现——销毁

```
LinkList <DataType> ::~ LinkList (){
    //析构函数，删除链栈，要释放栈中每个结点
        Node<DataType> *p=first;
        while（first!=nullptr）
        {
            first = first ->next;
            delete p;
            p= first;
        }
}
```

单链表存储优缺点

优点：

在链表中进行插入和删除操作不需要移动元素

缺点：

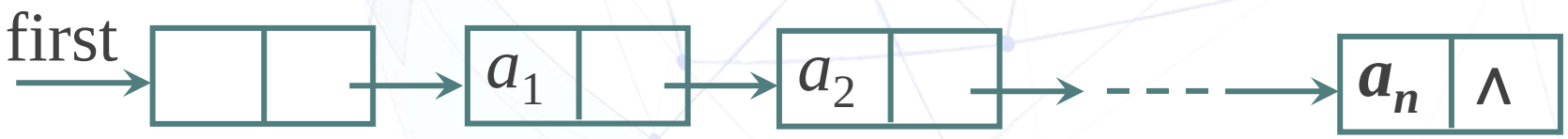
链表中按位置访问只能从表头开始依次向后扫描，直到找到那个特定位置

单链表VS顺序表

- 存储分配方式的比较　静态 OR 动态
- 时间性能比较　基本操作的时间复杂度
- 空间性能比较　所占存储空间的大小

存储分配方式

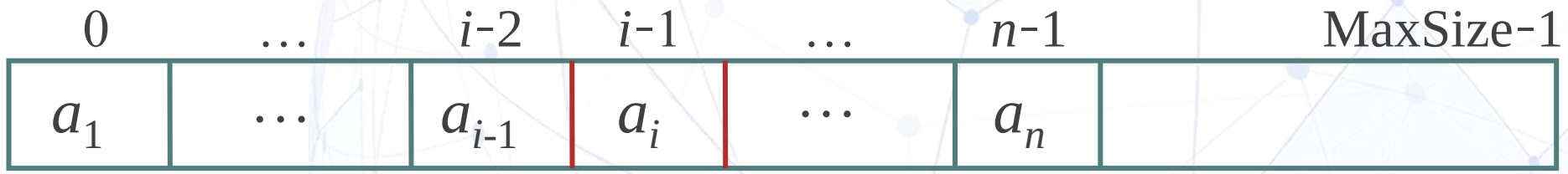
顺序表：采用**顺序**存储结构——**静态**存储分配，即用一段地址**连续**的存储单元**依次**存储线性表的数据元素，数据元素之间的逻辑关系通过**存储位置**（下标）来实现

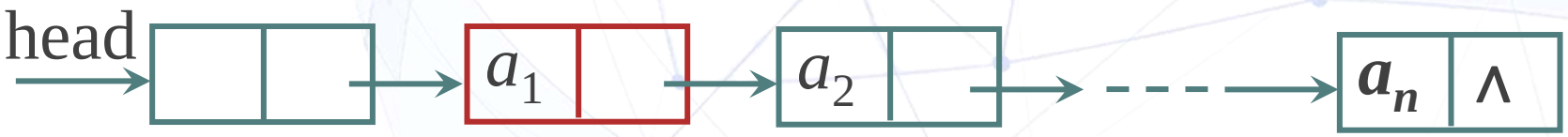


链表：采用**链接**存储结构——**动态**存储分配，即用一组**任意**的存储单元存放线性表的元素，用**指针**来反映数据元素之间的逻辑关系

空间性能比较

- 结点的存储密度比较
  - 顺序表：只存储数据元素
  - 链表：指针的结构性开销

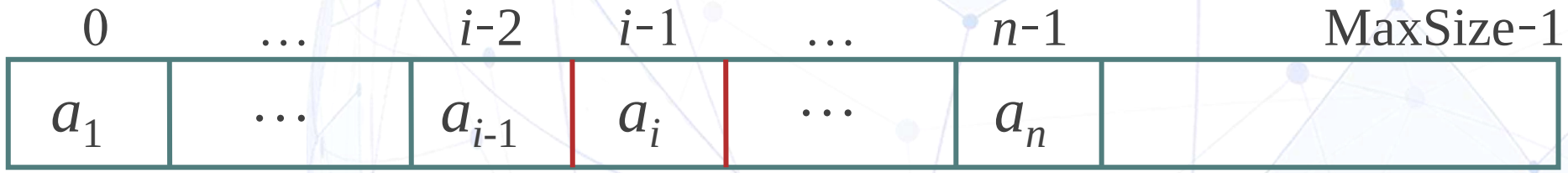


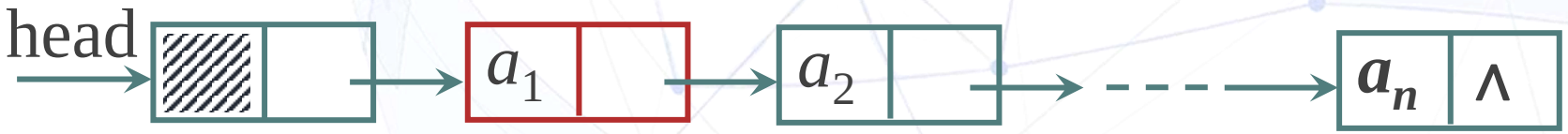


- 结构的存储密度比较
  - 顺序表：预分配存储空间
  - 链表：链表中的元素个数没有限制

时间性能比较

- 按位查找
  - 顺序表：$O(1)$，随机存取
  - 链表：$O(n)$，顺序存取





- 插入和删除
  - 顺序表：$O(n)$，平均移动表长一半的元素
  - 链表：不用移动元素，合适位置的指针—— $O(1)$

两者比较结论

- 从空间上讲，若线性表中元素**个数变化**较大或者未知，最好使用链表实现；如果用户事先知道线性表的大致长度，使用顺序表的空间效率会更高

- 从时间上讲，若线性表**频繁查找**却很少进行插入和删除操作，或其操作和元素在表中的位置密切相关时，宜采用顺序表作为存储结构；若线性表需**频繁插入和删除**时，则宜采用链表做存储结构

各有优缺点，应根据实际问题进行综合考虑，选定合适的实现方法

课堂小结

线性表的单链表存储特点

链式存储的常用操作

单链表和顺序表的比较