指间宇宙
—编程高手—

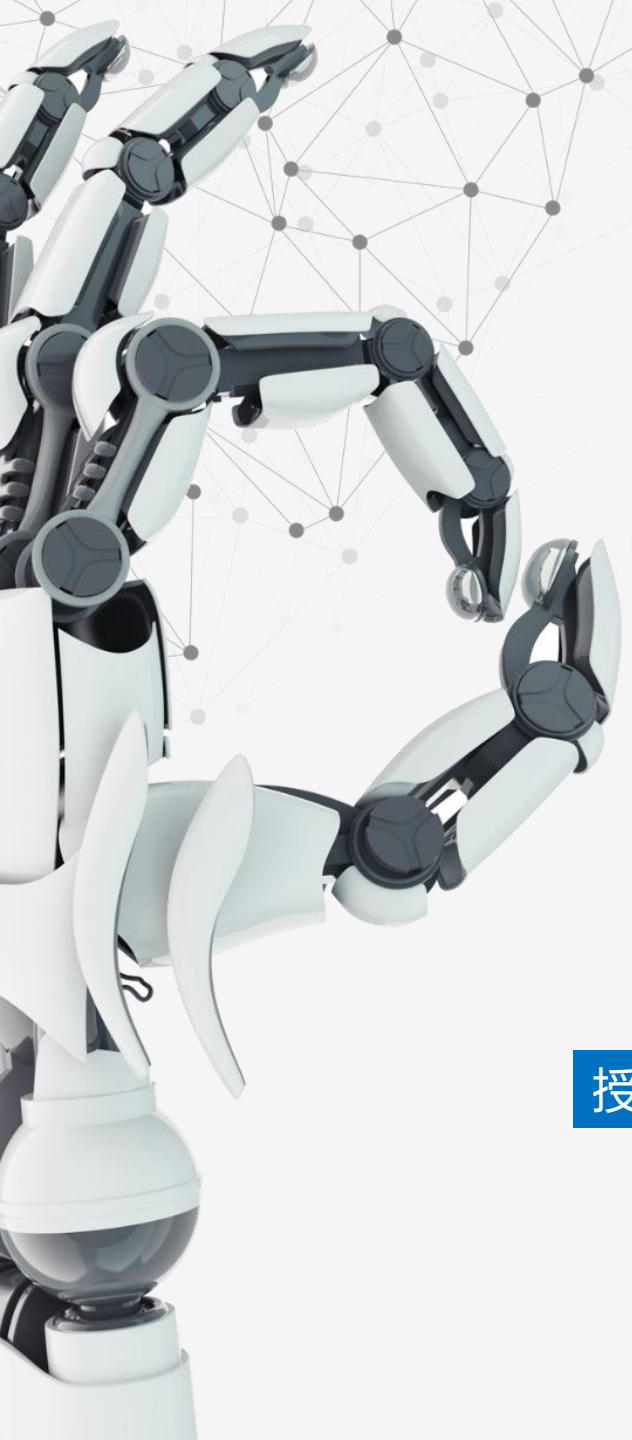
数据结构

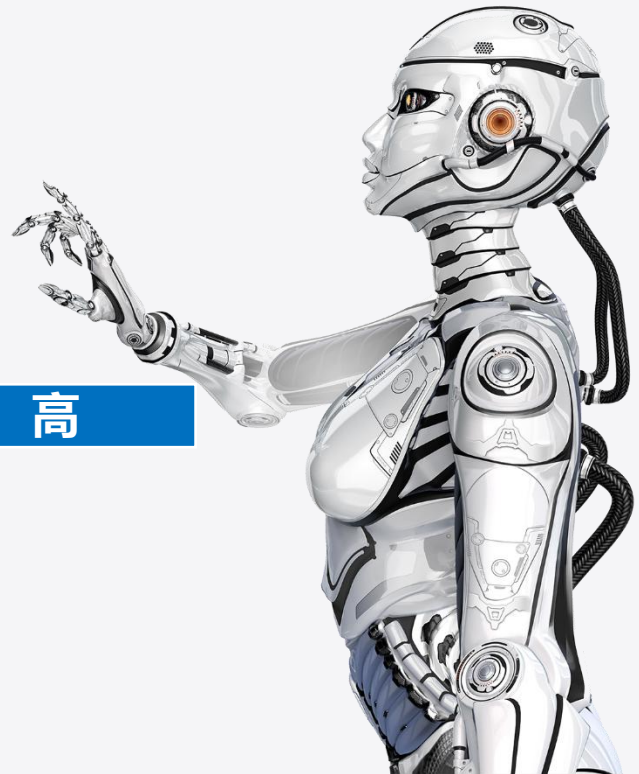
授课日期：

授课时间：

授课教师：

考试所涉及级别：中、高





指间宇宙
一编程高手一

目 录
CONTENT

STL库

- C++ 标准模板库（Standard Template Library，STL）是一套功能强大的 C++ 模板类和函数的集合，它提供了一系列通用的、可复用的算法和数据结构。
- STL内置了绝大多数我们所需要的数据结构及其基本操作，使用 STL 可以更加方便灵活地处理数据，而无需再纠结某些细节如何用代码实现。
- STL 的设计基于泛型编程，这意味着使用模板可以编写出独立于任何特定数据类型的代码。

- STL 分为多个组件，包括容器（Containers）、迭代器（Iterators）、算法（Algorithms）、函数对象（Function Objects）和适配器（Adapters）等。

- 使用 STL 的好处:
- 代码复用：STL 提供了大量的通用数据结构和算法，可以减少重复编写代码的工作。
- 性能优化：STL 中的算法和数据结构都经过了优化，以提供最佳的性能。
- 泛型编程：使用模板，STL 支持泛型编程，使得算法和数据结构可以适用于任何数据类型。
- 易于维护：STL 的设计使得代码更加模块化，易于阅读和维护。

C++ 标准模板库的核心包括以下重要组件组件：

组件	描述
容器（Containers）	容器是 STL 中最基本的组件之一，提供了各种数据结构，包括向量（vector）、链表（list）、队列（queue）、栈（stack）、集合（set）、映射（map）等。这些容器具有不同的特性和用途，可以根据实际需求选择合适的容器。
算法（Algorithms）	STL 提供了大量的算法，用于对容器中的元素进行各种操作，包括排序、搜索、复制、移动、变换等。这些算法在使用时不需要关心容器的具体类型，只需要指定要操作的范围即可。
迭代器（iterators）	迭代器用于遍历容器中的元素，允许以统一的方式访问容器中的元素，而不用关心容器的内部实现细节。STL 提供了多种类型的迭代器，包括随机访问迭代器、双向迭代器、前向迭代器和输入输出迭代器等。
函数对象（Function Objects）	函数对象是可以像函数一样调用的对象，可以用于算法中的各种操作。STL 提供了多种函数对象，包括一元函数对象、二元函数对象、谓词等，可以满足不同的需求。
适配器（Adapters）	适配器用于将一种容器或迭代器适配成另一种容器或迭代器，以满足特定的需求。STL 提供了多种适配器，包括栈适配器（stack adapter）、队列适配器（queue adapter）和优先队列适配器（priority queue adapter）等。

这些个组件都带有丰富的预定义函数，帮助我们通过简单的方式处理复杂的任务。

容器

- 容器是用来存储数据的序列，它们提供了不同的存储方式和访问模式。
- STL 中的容器可以分为三类：
- 1、序列容器：存储元素的序列，允许双向遍历。
- std::vector：动态数组，支持快速随机访问。
- std::deque：双端队列，支持快速插入和删除。
- std::list：链表，支持快速插入和删除，但不支持随机访问。

- 2、关联容器：存储键值对，每个元素都有一个键（key）和一个值（value），并且通过键来组织元素。
- std::set：集合，不允许重复元素。
- std::multiset：多重集合，允许多个元素具有相同的键。
- std::map：映射，每个键映射到一个值。
- std::multimap：多重映射，允许多个键映射到相同的值。

- 3、无序容器（C++11 引入）：哈希表，支持快速的查找、插入和删除。
- std::unordered_set：无序集合。
- std::unordered_multiset：无序多重集合。
- std::unordered_map：无序映射。
- std::unordered_multimap：无序多重映射。

补充说明

01

数据结构

做题的时候一定要注意哦

什么是数据结构呢？

指间宇宙
一编程高手一

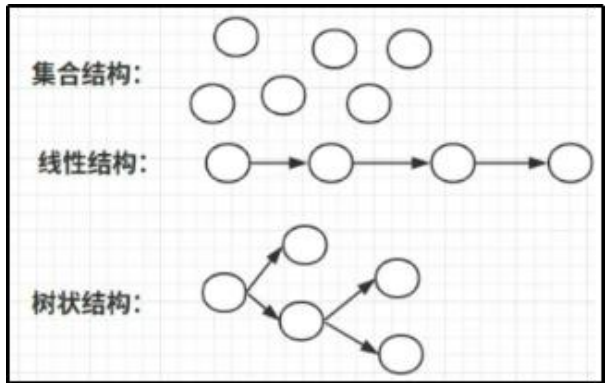
- 1、什么是程序？
- 程序=数据结构+算法

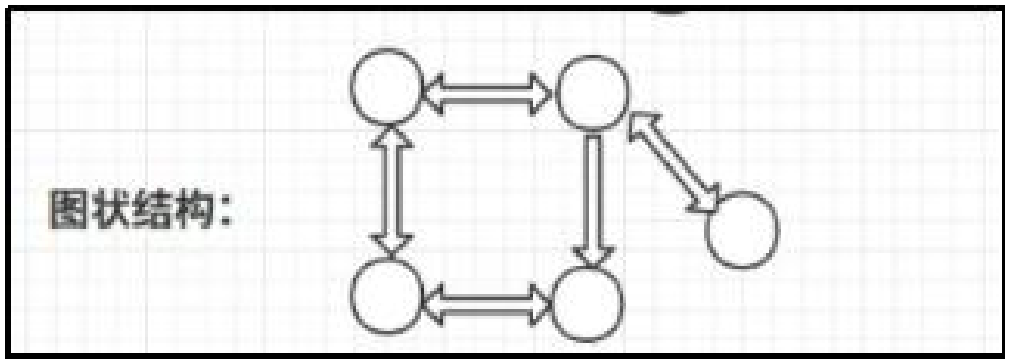
- 数据结构：如何把现实世界的问题信息化，将信息存储到计算机当中。同时要实现对数据结构的基本操作。也可以理解为是一种用于在C++中存储数据的方式，它可以是数组，指针，链表，堆栈，队列，树，图，哈希表等。

。

1.数据结构的概念

- 数据结构是计算机存储、组织数据的方式，是指相互之间存在一种或多种特定关系的数据元素的集合。描述数据结构通常包括以下几个方面。
- 1.数据的逻辑结构：描述数据之间的关系，常见的数据关系包括：
- a.集合：数据结构中的元素之间除了同属一个集合 外，别无其他关系。
- b.线性结构/线性表：数据结构中的元素存在一对一的相互关系。
- c.树结构：数据结构中的元素存在一对多的相互关系。
- d.图结构：数据结构中的元素存在多对多的相互关系。

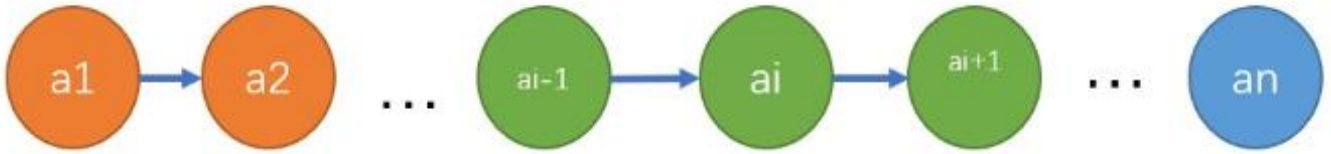




1. 数据的物理存储结构：描述数据在内存或磁盘中的实际存储方式

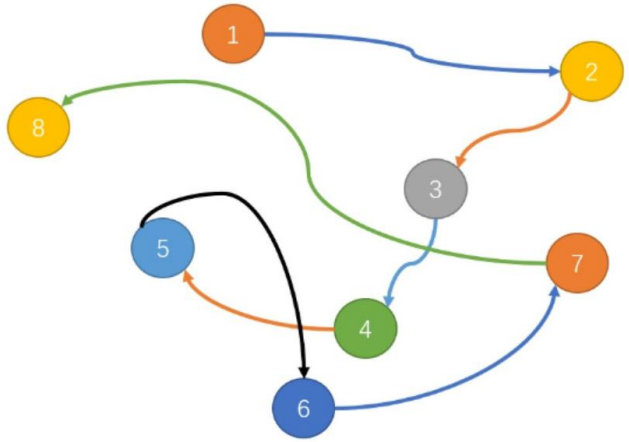
a. 顺序存储

使用连续的内存单元存储数据元素。顺序存储结构具有快速的随机访问特性，可以通过索引直接访问任意位置的元素。但是，插入和删除操作需要移动大量元素，因为需要保持数据的连续性，这可能导致效率较低。在具体实现上，通常以数组为基础实现。



b. 链式存储

使用不一定连续的内存单元存储数据元素，每个数据元素包含数据和标志下一个元素是谁的“指针”。链式存储结构支持动态的插入和删除操作，只需要改变“指针”的指向即可，而不需要移动其他元素。然而，链式存储结构对于随机访问来说效率较低，因为需要遍历链表才能找到目标元素。在具体实现上，通常以指针（一种暂未学习的语法）为基础实现。



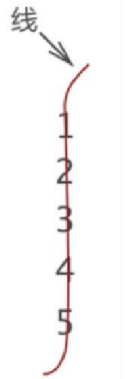
3. 数据的操作：定义了对数据的基本操作，包括定义数据结构，在数据结构中插入、删除、查找和排序等。

线性表

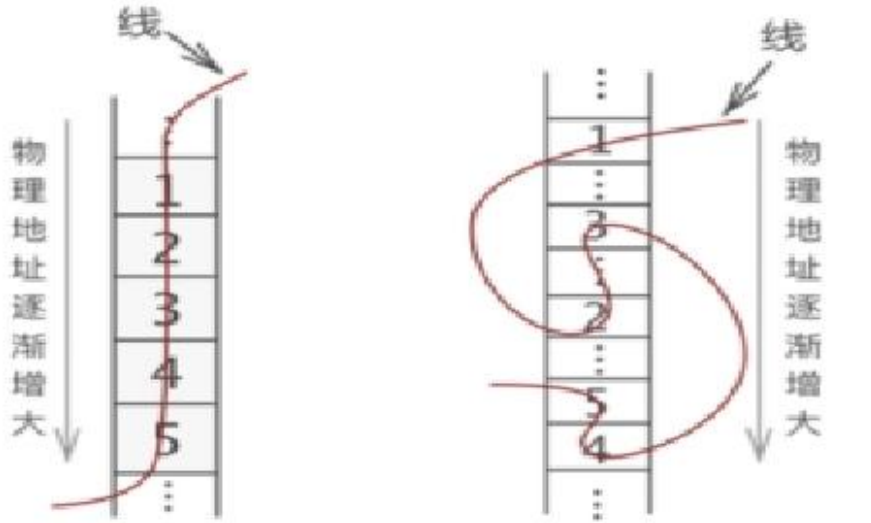
做题的时候一定要注意哦

1.线性表是什么

- 线性表又称线性存储结构，是最基础的数据结构，是由同类型的数据元素（节点）构成的有序序列。线性表的最大特点是逻辑上每个元素有且仅有一个直接前驱和一个直接后继，第一个元素没有直接前驱元素，最后一个元素没有直接后继元素。



根据物理存储结构不同，线性表可以分为顺序存储结构和链式存储结构。将数据依次存储在连续的整块物理空间中，这种存储结构称为顺序存储结构（又称数组/顺序表）。数据分散的存储在物理空间中，通过一根线（指针）保存着它们之间的逻辑关系，这种存储结构称为链式存储结构（又称单链表或线性链表）。



1. 带限制的线性表

线性表可以在任意位置进行增加和删除元素操作，如果将增减元素操作的位置局限于开头或结尾，就能得到两类带限制的线性数据结构——栈和队列。

1. 栈：只能在线性表的一端进行删除和插入

2. 队列：只能在线性表的一段进行插入，另一端进行删除

02
指针

https://blog.csdn.net/m0_64036070/article/details/124039548

众所周知

c++相对于其它语言的

优势就在与c++有「 指针 」

可能你之前有学习过指针

可是

你真的把指针彻底搞懂了吗🤔

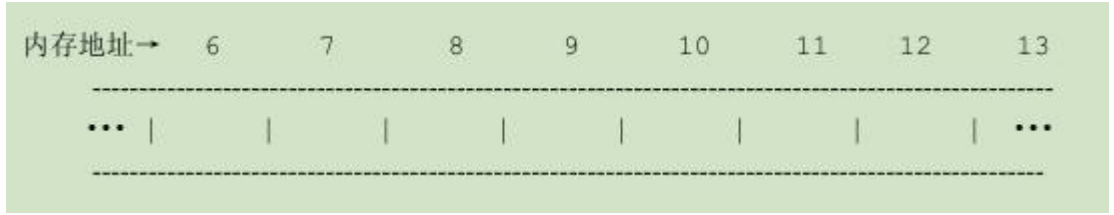
如果还没有，那再来复习一下❗

指针是什么？What is a pointer？

指针是变量在内存中的地址(A pointer is the memory address of a variable)

这个很重要

首先，来看一下内存空间(图片来源于网络)



简单点说

比如你有一个int类型的变量，里面的内容是10

把这个变量的地址保存在指针里，注意了，指针里面保存的不是这个变量本身，而是变量在内存中的地址

比如你要买一本书，你去了图书馆，查到了这本书在0x6ffe04位置上

你拿到了这个位置，并不代表你拿到了这本书，你要通过这个位置来找到这本书

这下懂了吧

一级指针的定义

```
int *p;//语法：类型 * 指针变量名;
```

指针的赋值

由于指针里存的是变量的地址，看这个程序

```
1
2  int *p;//定义一个指针
3  p=&a;//&是取址符
4
5  //或者
6  int *p=&a;
```

指针的输出
我们写这样一个程序

可以看到，我们如果要输出a

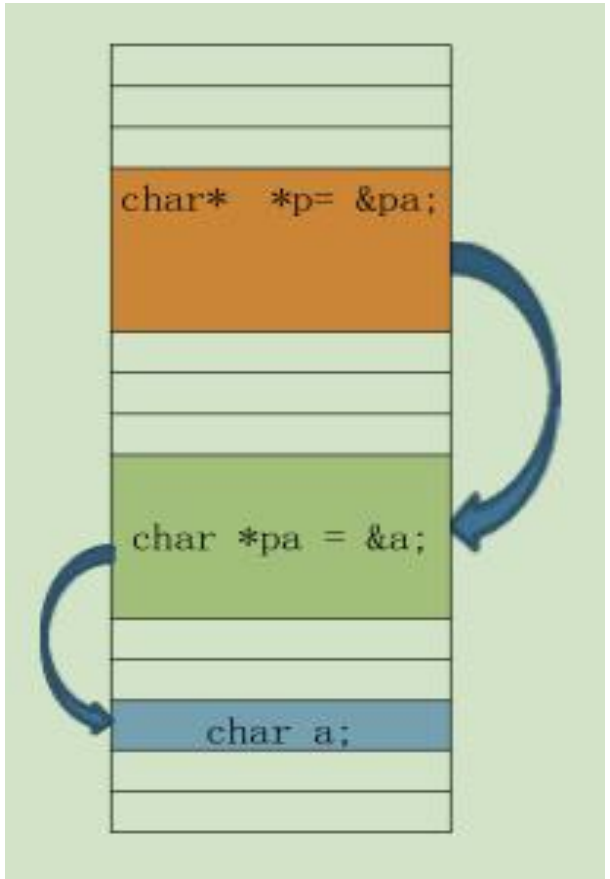
不管是通过指针输出还是直接输出变量a结果是一样的

```
#include<iostream>
using namespace std;
int main()
{
	int a=10;
	int *p=&a;
	cout<<"指针p:"<<*p<<endl<<"a的地址:"<<&a<<endl<<"a:"<<a;
	return 0;
}
```

运行结果：

```
指针p:10
a的地址:0x6ffe04
a:10
--------------------------------
Process exited after 0.04383 seconds with return value 0
请按任意键继续. . .
```

二级指针的定义
看这张图



简单说
简单说，二级指针就是指针的指针，二级指针里存放的是指针的地址

二级指针的赋值

二级指针的赋值和一级指针一样

```
1 int a=10;
2 int *pi=&a;
3 int **pi=&pi;
```

二级指针的输出

我们写这样一个程序

运行结果：

```
*p:10
&a:0x6ffdfc
a:10
**pp:10
&pp:0x6ffde8
&p:0x6ffdf0
--------------------------------
Process exited after 0.06748 seconds with return value 0
请按任意键继续. . .
```

```
#include<iostream>
using namespace std;
int main()
{
	int a=10;
	int *p=&a;
	int **pp=&p;
	cout<<"*p:"<<*p<<endl<<"&a:"<<&a<<endl<<"a:"<<a<<en
dl<<"**pp:"<<**pp<<endl<<"&pp:"<<&pp<<endl<<"&p:"<<&p;
	return 0;
}
```

由此可见，二级指针和一级指针的输出是一样的

03

栈

做题的时候一定要注意哦

1.什么是栈

- 栈(stack)是一种带限制的线性表，类似羽毛球桶，先放进去的羽毛球，后面才能拿出来

在实际应用中，通常只会对栈执行以下两种操作：

1、向栈中添加元素，此过程被称为"进栈"（入栈或压栈）；

2、从栈中提取出指定元素，此过程被称为"出栈"（或弹栈）；

在一些特殊的算法中会用到栈，stl提供了stack头文件，熟练其内部函数：

- 定义：stack<数据类型> 栈名
- 入栈：s.push()
- 出栈：s.pop()
- 大小：s.size()
- 栈顶：s.top()
- 栈空：s.empty() 空则返回true

```
#include<iostream>
#include<stack>
using namespace std;

int main()
{
  stack<int> s;

  s.push(1);
  s.push(2);
  s.push(3);
  s.push(4);
  s.push(5);

  cout << "栈大小：" << s.size() << endl;
  cout << "栈顶：" << s.top() << endl;

  while(!s.empty()){
    cout << s.top() << "出栈" << endl;
    s.pop();
  }

  return 0;
}
```

使用数组模拟实现一个栈

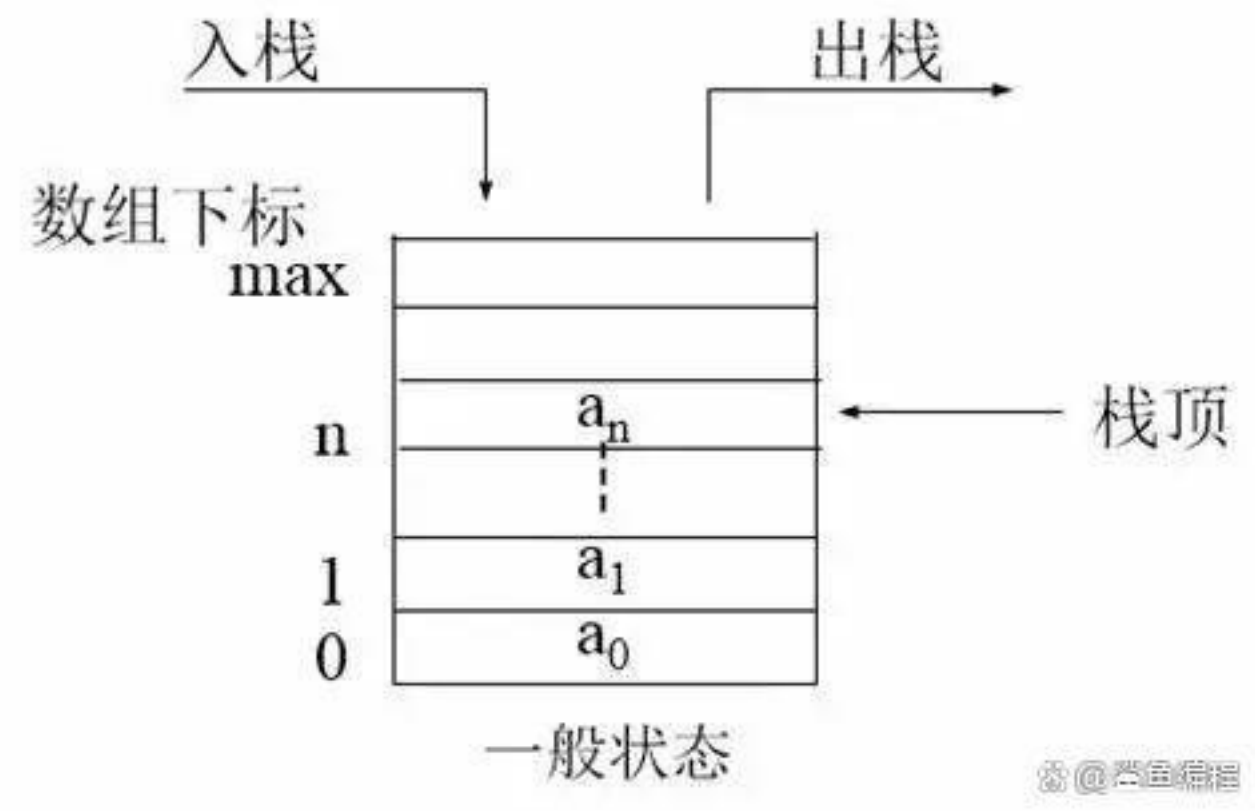
- 栈是一种操作受限的线性数据结构，可以理解为操作受限的数组，因此，我们可以自行模拟出栈。
- 例如OJ《模拟栈》

栈的数组模拟做法

```
//数组模拟做法
#include<iostream>
using namespace std;
const int N=1e5+5;

int m;
int st[N],tt;

int main(){
    cin>>m;
    while(m--){
        string op;
        int x;
        cin>>op;
        if(op=="push"){
            cin>>x;
            st[++tt]=x;
        }
        else if(op=="pop"){
            tt--;
        }
        else if(op=="empty"){
            cout<<(tt ? "NO" : "YES")<<endl;
        }
        else cout<<st[tt]<<endl;
    }
    return 0;
```

栈的stl做法

```
//stl做法
#include<iostream>
#include<stack>
using namespace std;
stack<int> st;

int main(){
	string op;
	int x;
	int m;
	cin>>m;
	while(m--){
		cin>>op;
		if(op=="push"){
			cin>>x;
			st.push(x);
		}
		else if(op=="pop"){
			st.pop();
		}
		else if(op=="empty"){
			if(st.empty()) cout<<"YES"<<endl;
			else cout<<"NO"<<endl;
		}
		else if(op=="query"){
			cout<<st.top()<<endl;
		}
	}
	return 0;
}
```



03
队列

做题的时候一定要注意哦

1.什么是队列

- 与栈类似，队列（queue）简称队，它也是一种操作受限的线性表，其限制为仅允许在表的一端进行插入操作，而在表的另一端进行删除操作
- 队列是一种先进先出表（FIFO），而前面介绍过的栈是一种先进后出表。

（一个队列）

1. queue的定义与初始化：queue<int> q，初始化方式与其他容器类似

2. queue的内部函数

a. 增加元素q.push(x)：入队

b. 删除元素q.pop()：出队

c. 控制函数

i. int x=q.front() ： 获取队首元素

ii. int x=q.back() ： 获取队尾元素

iii. q.empty()：当queue中没有元素时，该成员函数返回 true；反之，返回 false

iv. q.size()：返回queue中存储元素的个数

下面是使用队列的简单例子，输出队列中的元素

```
#include<bits/stdc++.h>
using namespace std;
int main(){
    queue<int> q;
    for(int i=1;i<=5;i++){
        int x;
        cin>>x;
        q.push(x);
    }
     while(q.empty()==false){
        cout<<q.front()<<" "<<q.back()<<endl;
        q.pop();
     }
     return 0;
}
```

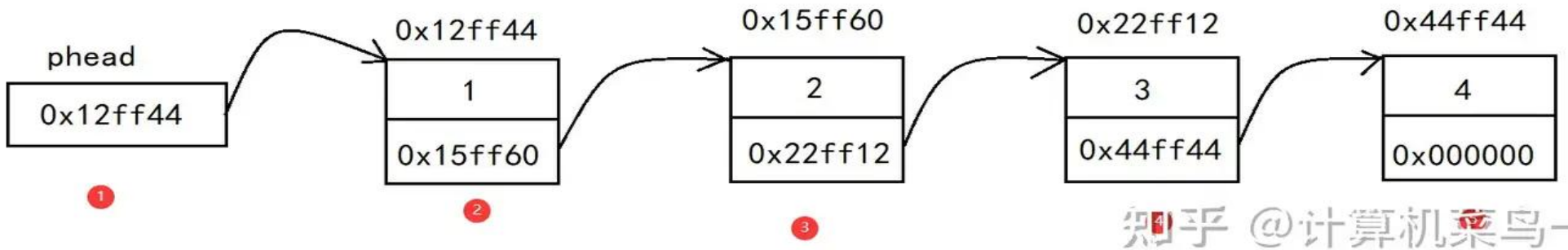
04

链表(这里介绍单链表)

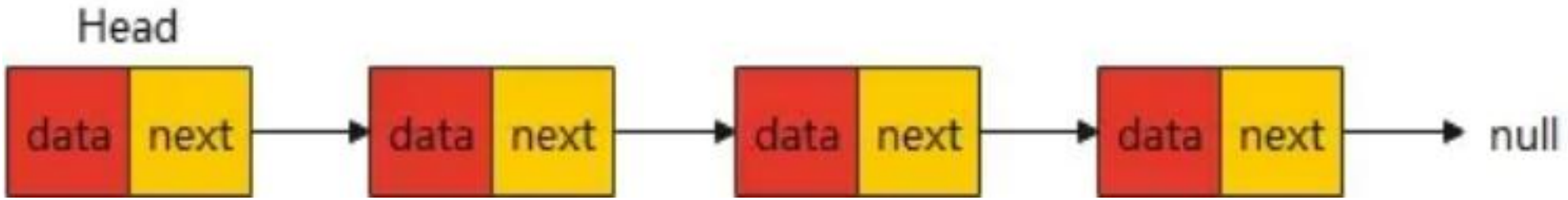
https://zhuanlan.zhihu.com/p/648341058

链表定义

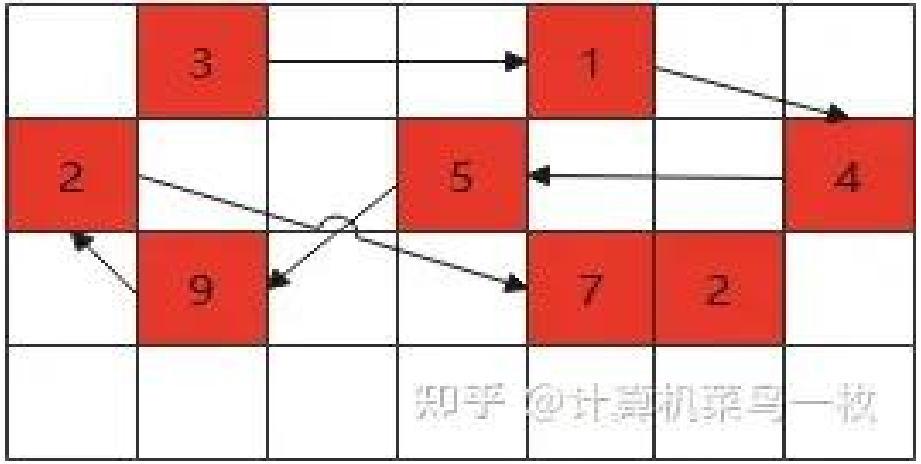
- 【1】概念：链表是一种物理存储结构上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。



【2】组成：由一系列节点（Node）通过指针连接而成，从一个头节点（Head）开始，头节点作为链表的入口点，它包含了对第一个节点的引用。最后一个节点的指针指向一个空值（NULL），表示链表的结束。



【3】存储：链表在内存中的存储方式则是随机存储（见缝插针），每一个节点分布在内存的不同位置，依靠指针关联起来。（只要有足够的内存空间，就能为链表分配内存）



【4】优缺点：相对于顺序储存（例如数组）：

优点：链表的插入操作更快（O(1)），无需预先分配内存空间

缺点：失去了随机读取的优点（需要从头节点开始依次遍历，直到找到目标节点。），内存消耗较大（每个节点都需要存储指向下一个节点的指针）

【5】对比：

链表是通过节点把离散的数据链接成一个表，通过对节点的插入和删除操作从而实现对数据的存取。而数组是通过开辟一段连续的内存来存储数据，这是数组和链表最大的区别。

数组的每个成员对应链表的节点，成员和节点的数据类型可以是标准的 C 类型或者是 用户自定义的结构体。

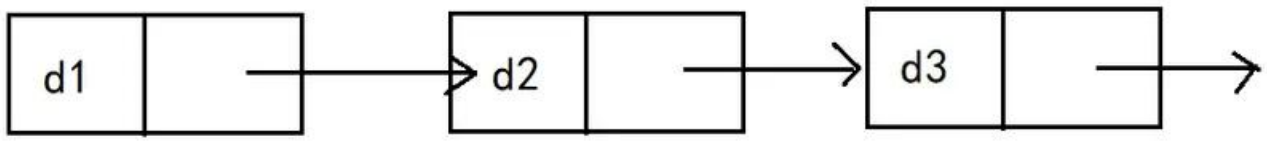
数组有起始地址和结束地址，而链表是一个圈，没有头和尾之分， 但是为了方便节点的插入和删除操作会人为的规定一个根节点。

不同点	顺序表	链表
存储空间上	物理上一定连续	逻辑上连续，但物理上不一定连续
随机访问	支持O(1)	不支持：O(N)
任意位置插入或者删除元素	可能需要搬移元素，效率低O(N)	只需修改指针指向
插入	动态顺序表，空间不够时需要扩容	没有容量的概念
应用场景	元素高效存储+频繁访问	任意位置插入和删除频繁
缓存利用率	高	低

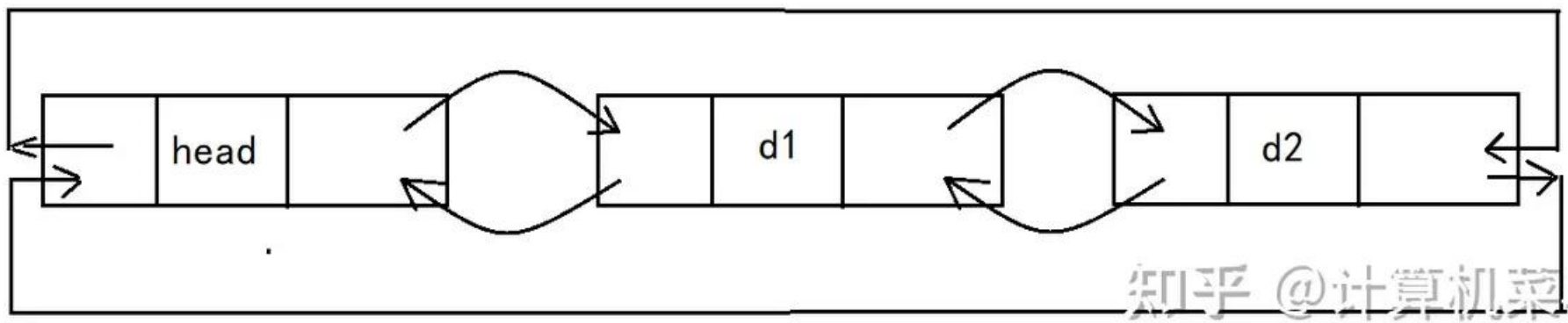
【5】分类：链表一般有单向链表，双向链表，循环链表，带头链表这四种形式。

常用的两种结构：

无头单向非循环链表



带头双向循环链表



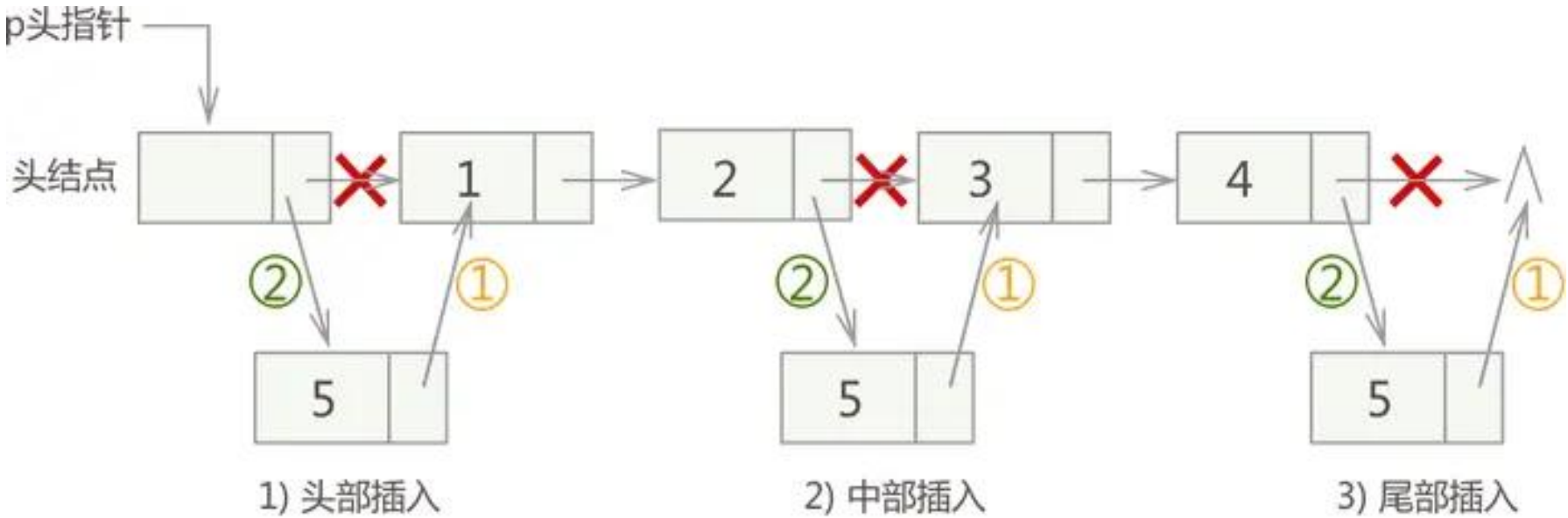
1. 无头单向非循环链表：结构简单，一般不会单独用来存数据。实际中更多是作为其他数据结构的子结构，如哈希桶、图的邻接表等等。另外这种结构在笔试面试中出现很多。

2. 带头双向循环链表：结构最复杂，一般用在单独存储数据。实际中使用的链表数据结构都是带头双向循环链表。另外这个结构虽然结构复杂，但是使用代码实现以后会发现结构会带来很多优势，实现反而简单。

链表的创建

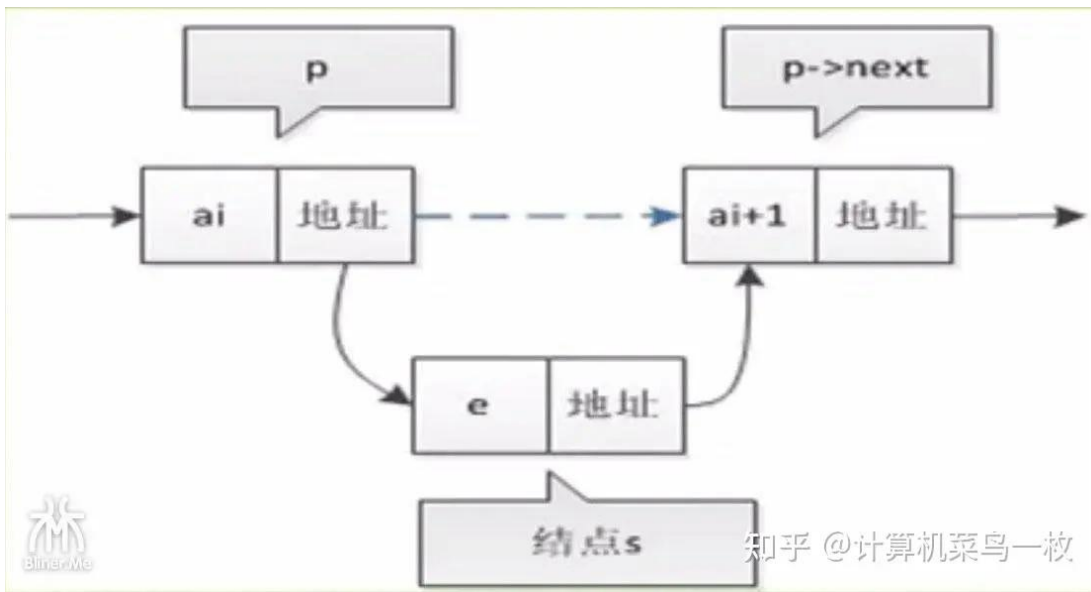

```
// 定义链表节点结构
struct Node {
    int data;         // 数据
    struct Node* next;  // 指向下一个节点的指针
};
// 创建链表函数
struct Node* createLinkedList(int arr[], int n)
 {
    struct Node* head = NULL;    // 头节点指针
    struct Node* tail = NULL;    // 尾节点指针

    // 遍历数组，为每个元素创建一个节点，并加入链表
    for (int i = 0; i < n; i++) {
        // 创建新节点并为其分配内存
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        if (newNode == NULL) {
            printf("内存分配失败.");
            return NULL;
        }
        // 设置新节点的数据
        newNode->data = arr[i];
        newNode->next = NULL;

        // 如果链表为空，将新节点设置为头节点和尾节点
        if (head == NULL) {
            head = tail = newNode;
        }
        // 如果链表非空，将新节点加入到尾部，并更新尾节点指针
        else {
            tail->next = newNode;
            tail = newNode;
        }
    }

    return head;
}
```

链表的插入



同顺序表一样，向链表中增添元素，根据添加位置不同，可分为以下 3 种情况：

插入到链表的头部，作为首元节点；
插入到链表中间的某个位置；
插入到链表的最末端，作为链表中最后一个结点；

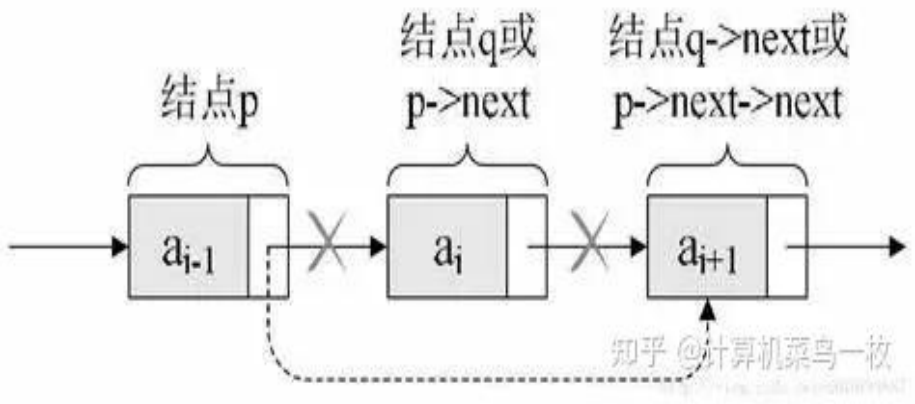


```
// 在链表中插入节点
void insert(struct Node** headRef, int position, int value) {
    // 创建新节点并为其分配内存
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    if (newNode == NULL) {
        printf("内存分配失败。\n");
        return;
    }
    newNode->data = value;

    // 如果要插入的位置是链表的头部或链表为空
    if (*headRef == NULL || position == 0) {
        newNode->next = *headRef;
        *headRef = newNode;
        return;
    }

    struct Node* current = *headRef;
    int count = 1;
    // 找到要插入位置的前一个节点
    while (current->next != NULL && count < position) {
        current = current->next;
        count++;
    }

    // 在指定位置插入节点
    newNode->next = current->next;
    current->next = newNode;
}
```

链表的查询


```
// 在链表中查询节点
struct Node* search(struct Node* head, int value) {
    struct Node* current = head;

    while (current != NULL) {
        if (current->data == value) {
            return current;  // 返回匹配的节点地址
        }
        current = current->next;
    }

    return NULL;  // 若没有找到匹配节点，则返回NULL
}
```

链表的删除




```
// 在链表中删除节点
void delete(struct Node** headRef, int value) {
    struct Node* current = *headRef;
    struct Node* prev = NULL;

    // 处理头节点为目标节点的情况
    if (current != NULL && current->data == value) {
        *headRef = current->next;
        free(current);
        return;
    }

    // 遍历链表找到要删除的节点
    while (current != NULL && current->data != value) {
        prev = current;
        current = current->next;
    }

    // 如果找到了目标节点，则删除它
    if (current != NULL) {
        prev->next = current->next;
        free(current);
    }
}
```

链表的释放


```
// 释放链表内存
void freeList(struct Node* head) {
    struct Node* current = head;
    struct Node* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
}
```

链表的遍历


```
void printLinkedList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}
```