# 数据结构概念

## 1.1数据结构相关概念

### 1.1.1疑惑

1、我学完了C语言，可是现在感觉还是写不出代码。

2、为什么会有各种各样的程序存在？

3、程序的本质是什么？

程序是为了具体问题而存在的

程序需要围绕问题的解决进行设计

同一个问题可以有多种解决方案

如何追求程序的“性价比”？

是否有可量化的方法判别程序的好坏？

### 1.1.2数据结构起源

计算机从解决数值计算问题到解决生活中的问题

现实生活中的问题涉及不同个体间的复杂联系

需要在计算机程序中描述生活中个体间的联系

数据结构主要研究非数值计算程序问题中的操作对象以及它们之间的关系

不是研究复杂的算法

### 1.1.3数据结构中的基本概念

数据 – 程序的操作对象，用于描述客观事物 (int a, int b,)

数据的特点：

可以输入到计算机

可以被计算机程序处理

数据是一个抽象的概念，将其进行分类后得到程序设计语言中的类型。如：int，float，char等等

数据元素：组成数据的基本单位

数据项：一个数据元素由若干数据项组成

数据对象 – 性质相同的数据元素的集合 （比如：数组，链表）

|  |
| --- |
| //友情提示，来自结构体课堂代码  //声明一个结构体类型  struct \_MyTeacher //一种数据类型  {  char name[32];  char tile[32];  int age;  char addr[128];  };  int main21()  {  struct \_MyTeacher t1; //数据元素  struct \_MyTeacher tArray[30]; //数据对象  memset(&t1, 0, sizeof(t1));  strcpy(t1.name, "name"); //数据项  strcpy(t1.addr, "addr"); //数据项  strcpy(t1.tile, "addr"); //数据项  t1.age = 1;  } |
|  |

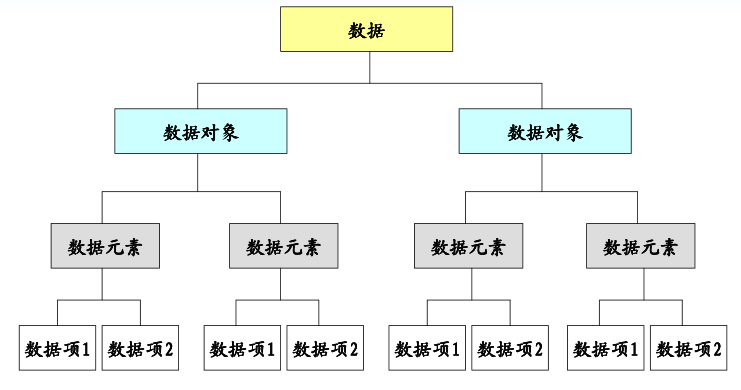
数据元素之间不是独立的，存在特定的关系，这些关系即结构

数据结构指数据对象中数据元素之间的关系

如：数组中各个元素之间存在固定的线性关系

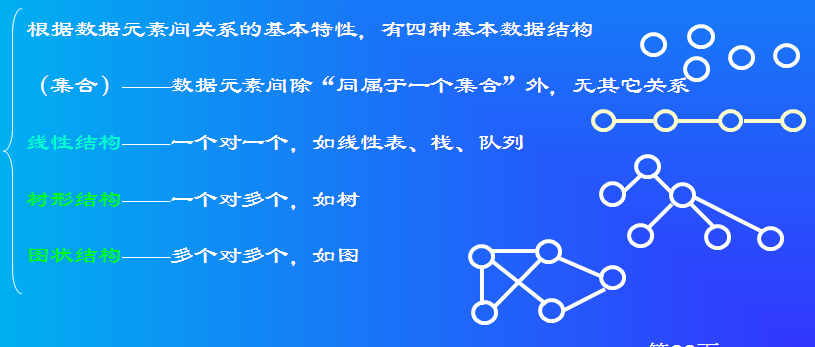
编写一个“好”的程序之前，必须分析待处理问题中各个对象的特性，以及对象之间的关系。

基本概念总结：

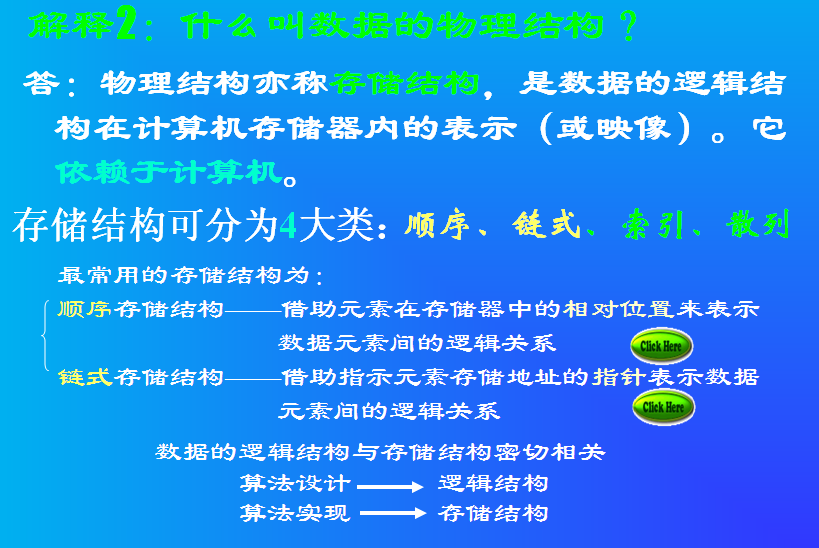


### 1.1.4数据的逻辑结构

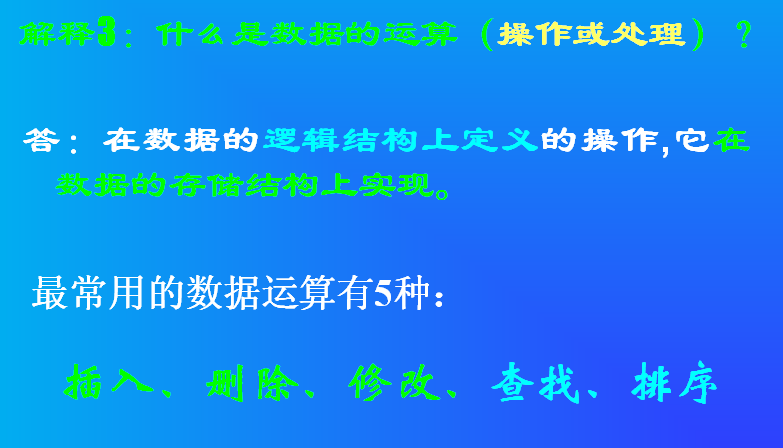
指数据元素之间的逻辑关系。即从逻辑关系上描述数据，它与数据的存储无关，是独立于计算机的。逻辑结构可细分为4类：



### 1.1.5数据的物理结构



### 1.1.6数据的运算



## 1.2、算法

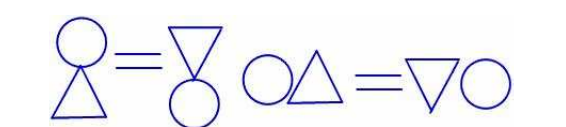
### 1.2.1算法概念

算法是特定问题求解步骤的描述

在计算机中表现为指令的有限序列

算法是独立存在的一种解决问题的方法和思想。

对于算法而言，语言并不重要，重要的是思想。



### 1.2.2算法和数据结构区别

数据结构只是静态的描述了数据元素之间的关系

高效的程序需要在数据结构的基础上设计和选择算法

===🡺程序=数据结构+算法

总结：

算法是为了解决实际问题而设计的

数据结构是算法需要处理的问题载体

数据结构与算法相辅相成

### 1.2.3算法特性

输入

算法具有0个或多个输入

输出

算法至少有1个或多个输出

有穷性

算法在有限的步骤之后会自动结束而不会无限循环

确定性

算法中的每一步都有确定的含义，不会出现二义性

可行性

算法的每一步都是可行的

### 1.2.4算法效率的度量

|  |
| --- |
| 1、事后统计法  比较不同算法对同一组输入数据的运行处理时间  缺陷  为了获得不同算法的运行时间必须编写相应程序  运行时间严重依赖硬件以及运行时的环境因素  算法的测试数据的选取相当困难  事后统计法虽然直观，但是实施困难且缺陷多 |
| 算法效率的度量  事前分析估算  依据统计的方法对算法效率进行估算  影响算法效率的主要因素  算法采用的策略和方法  问题的输入规模  编译器所产生的代码  计算机执行速度 |
| //算法最终编译成具体的计算机指令  //每一个指令，在具体的计算机上运行速度固定  //通过具体的n的步骤，就可以推导出算法的复杂度  long sum1(int n)  {  long ret = 0;  int\* array = (int\*)malloc(n \* sizeof(int));  int i = 0;    for(i=0; i<n; i++)  {  array[i] = i + 1;  }    for(i=0; i<n; i++)  {  ret += array[i];  }    free(array);    return ret;  }  long sum2(int n)  {  long ret = 0;  int i = 0;    for(i=1; i<=n; i++)  {  ret += i;  }    return ret;  }  long sum3(int n)  {  long ret = 0;    if( n > 0 )  {  ret = (1 + n) \* n / 2;  }    return ret;  }  int main()  {  printf("%d\n", sum1(100));  printf("%d\n", sum2(100));  printf("%d\n", sum3(100));    return 0;  } |
| int func(int a[], int len)  {  int i = 0;  int j = 0;  int s = 0;    for(i=0; i<len; i++) n  {  for(j=0; j<len; j++) n  {  s += i\*j; //n\*n  }  }  return s;  }  //n\*n |
| 注意1：判断一个算法的效率时，往往只需要关注操作数量的最高次项，其它次要项和常数项可以忽略。  注意2：在没有特殊说明时，我们所分析的算法的时间复杂度都是指最坏时间复杂度。 |

2、大O表示法

|  |
| --- |
| 算法效率严重依赖于操作(Operation)数量  在判断时首先关注操作数量的最高次项  操作数量的估算可以作为时间复杂度的估算 |
| O(5) = O(1)  O(2n + 1) = O(2n) = O(n)  O(n2+ n + 1) = O(n2)  O(3n3+1) = O(3n3) = O(n3) |
| 常见时间复杂度    关系 |
|  |

3、算法的空间复杂度

|  |
| --- |
| 算法的空间复杂度通过计算算法的存储空间实现  S(n) = O(f(n))  其中，n为问题规模，f(n))为在问题规模为n时所占用存储空间的函数  大O表示法同样适用于算法的空间复杂度  当算法执行时所需要的空间是常数时，空间复杂度为O(1) |
| 空间与时间的策略  多数情况下，算法执行时所用的时间更令人关注  如果有必要，可以通过增加空间复杂度来降低时间复杂度  同理，也可以通过增加时间复杂度来降低空间复杂度 |
| 练习1：分析sum1 sum2 sum3函数的空间复杂度  O(4n+12) O(8)=O(1) O(4)=O(1)  总结：实现算法时，需要分析具体问题，对执行时间和空间的要求。 |
|  |
| 练习2：时间换空间  /\*  问题：  在一个由自然数1-1000中某些数字所组成的数组中，每个数字可能出现零次或者多次。  设计一个算法，找出出现次数最多的数字。  \*/  方法1：  排序，然后找出出现次数最多的数字  方法2：  void search(int a[], int len)  {  int sp[1000] = {0};  int i = 0;  int max = 0;    for(i=0; i<len; i++)  {  int index = a[i] - 1;    sp[index]++;  }    for(i=0; i<1000; i++)  {  if( max < sp[i] )  {  max = sp[i];  }  }    for(i=0; i<1000; i++)  {  if( max == sp[i] )  {  printf("%d\n", i+1);  }  }  }  int main()  {  int array[] = {1, 1, 3, 4, 5, 6, 6, 6, 2, 3};    search(array, sizeof(array)/sizeof(\*array));    return 0;  } |
| 把每个数字出现的次数的中间结果，缓存下来；在缓存的结果中求最大值。 |
|  |

# 2、线性表

## 2.1线性表基本概念

### 2.1.1线性表定义

线性表(List)是零个或多个数据元素的集合

线性表中的数据元素之间是有顺序的

线性表中的数据元素个数是有限的

线性表中的数据元素的类型必须相同



### 2.1.2数学定义

线性表是具有相同类型的 n（ ≥ 0）个数据元素的有限序列

（a1, a2, …, an）

ai是表项，n 是表长度。

### 2.1.3性质

a0为线性表的第一个元素，只有一个后继

an为线性表的最后一个元素，只有一个前驱

除a0和an外的其它元素ai，既有前驱，又有后继  
线性表能够逐项访问和顺序存取

### 2.1.4练习

下面的关系中可以用线性表描述的是

A.班级中同学的友谊关系 N:N

B.公司中的上下级关系 1:N

C.冬天图书馆排队占座关系

D.花名册上名字之间的关系 1::1

### 2.2.5线性表的操作

创建线性表

销毁线性表

清空线性表

将元素插入线性表

将元素从线性表中删除

获取线性表中某个位置的元素

获取线性表的长度

线性表在程序中表现为一种特殊的数据类型

线性表的操作在程序中的表现为一组函数

|  |
| --- |
| C语言描述=====》线性表的设计与实现  ADT抽象层 《[数据结构(C语言版)].严蔚敏\_吴伟民.扫描版.pdf》 p44页  人生财富库积累 |
| #ifndef \_WBM\_LIST\_H\_  #define \_WBM\_LIST\_H\_  typedef void List;  typedef void ListNode;  //创建并且返回一个空的线性表  List\* List\_Create();  //销毁一个线性表list  void List\_Destroy(List\* list);  //将一个线性表list中的所有元素清空, 线性表回到创建时的初始状态  void List\_Clear(List\* list);  //返回一个线性表list中的所有元素个数  int List\_Length(List\* list);  //向一个线性表list的pos位置处插入新元素node  int List\_Insert(List\* list, ListNode\* node, int pos);  //获取一个线性表list的pos位置处的元素  ListNode\* List\_Get(List\* list, int pos);  //删除一个线性表list的pos位置处的元素 返回值为被删除的元素，NULL表示删除失败  ListNode\* List\_Delete(List\* list, int pos);  #endif |
| 注意:  int List\_Insert(List\* list, ListNode\* node, int pos); (重点:分离思想) |

## 2.2线性表的顺序存储结构

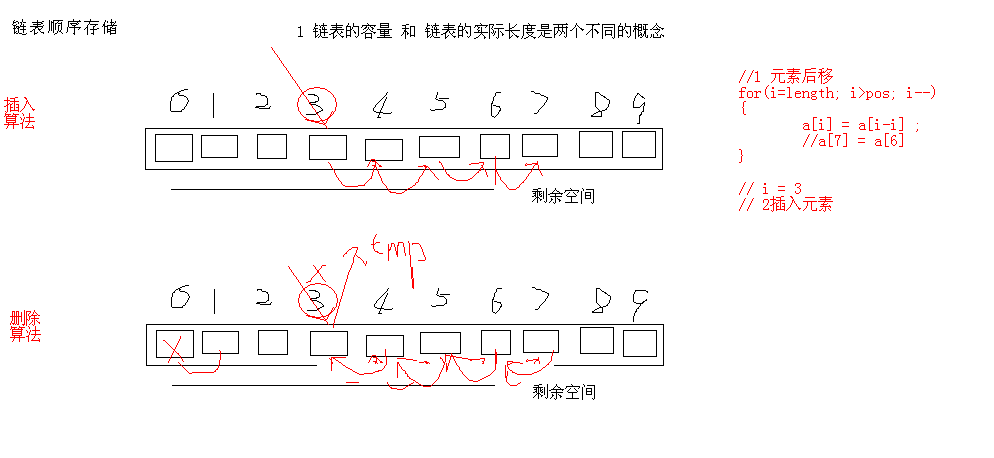
### 2.2.1基本概念

|  |
| --- |
|  |

### 2.2.2设计与实现

|  |
| --- |
| 插入元素算法  判断线性表是否合法  判断插入位置是否合法  把最后一个元素到插入位置的元素后移一个位置  将新元素插入  线性表长度加1 |
| 获取元素操作  判断线性表是否合法  判断位置是否合法  直接通过数组下标的方式获取元素 |
| 删除元素算法  判断线性表是否合法  判断删除位置是否合法  将元素取出  将删除位置后的元素分别向前移动一个位置  线性表长度减1 |
|  |

链表顺序存储插入算法和删除算法



### 2.2.3优点和缺点

优点：

无需为线性表中的逻辑关系增加额外的空间

可以快速的获取表中合法位置的元素

缺点：

插入和删除操作需要移动大量元素

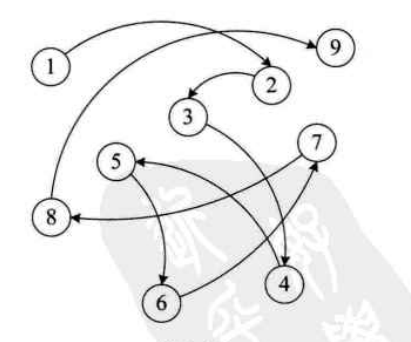
当线性表长度变化较大时难以确定存储空间的容量

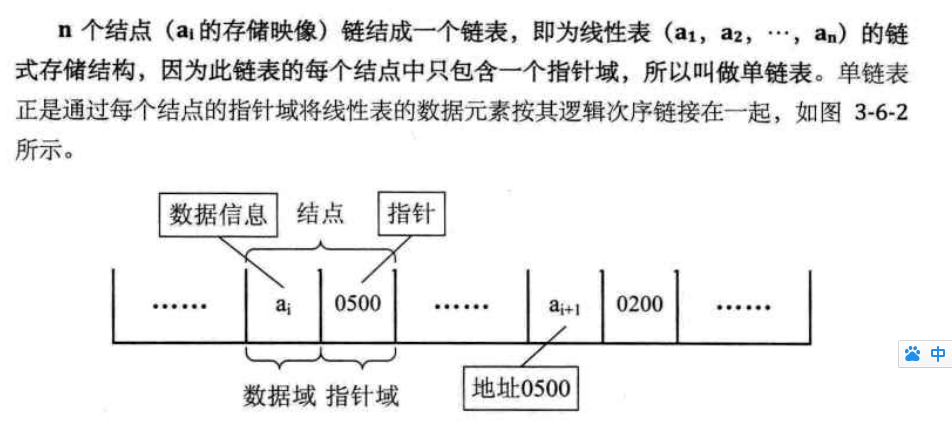
## 2.3线性表的链式存储

### 2.3.1基本概念

链式存储定义

为了表示每个数据元素与其直接后继元素之间的逻辑关系，每个元素除了存储本身的信息外，还需要存储指示其直接后继的信息。





表头结点

链表中的第一个结点，包含指向第一个数据元素的指针以及链表自身的一些信息

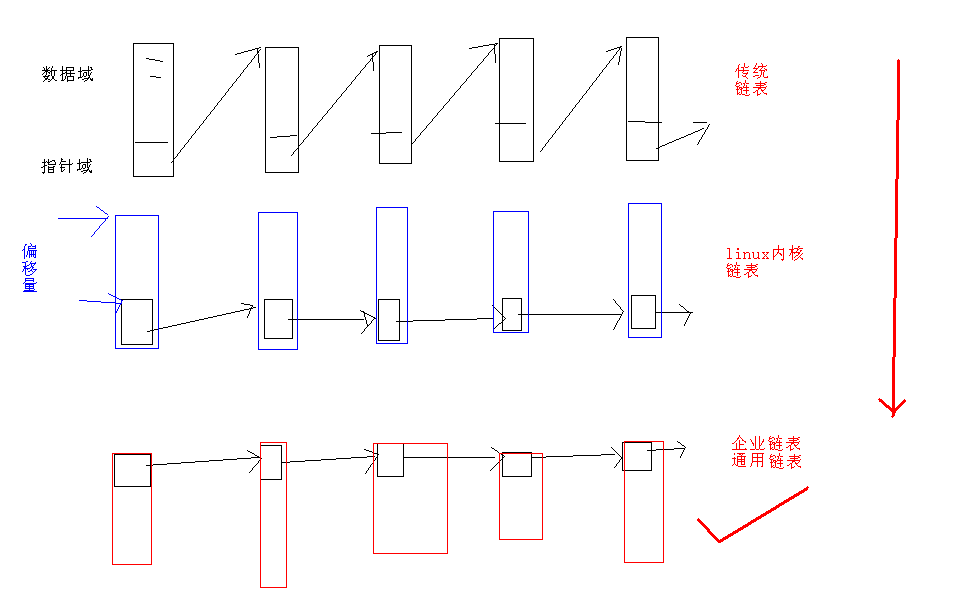
数据结点

链表中代表数据元素的结点，包含指向下一个数据元素的指针和数据元素的信息

尾结点

链表中的最后一个数据结点，其下一元素指针为空，表示无后继。

### 2.3.2链表技术领域推演



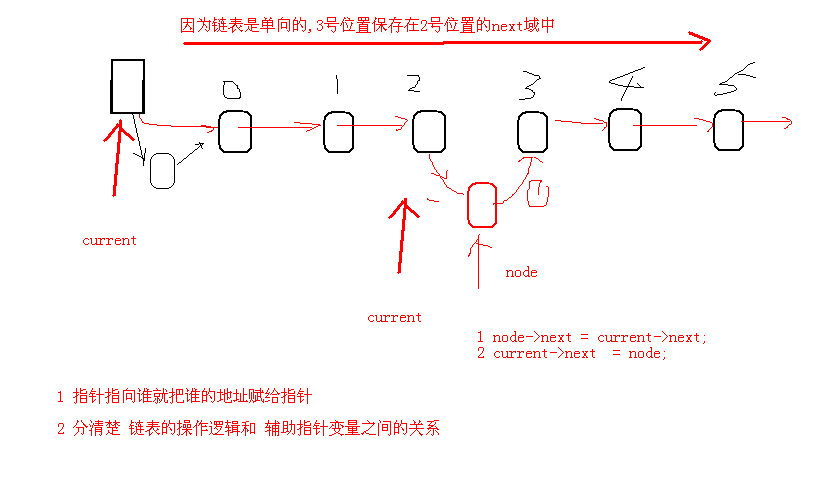
### 2.3.2设计与实现

#### 链表链式存储\_api实现分析

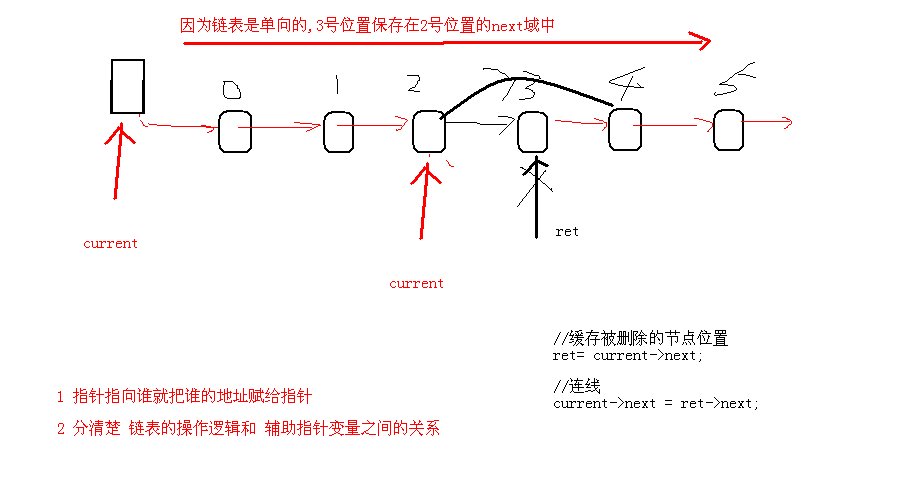
|  |
| --- |
| 在C语言中可以用结构体来定义链表中的指针域  链表中的表头结点也可以用结构体实现 |
|  |
|  |
| 带头结点、位置从0的单链表  返回链表中第3个位置处，元素的值  LinkListNode\* LinkList\_Get(LinkList\* list, int pos)  {  int i = 0;  TLinkList \*tList = (TLinkList \*)list;  LinkListNode \*current = NULL;  LinkListNode \*ret = NULL;  if (list==NULL ||pos<0 || pos>=tList->length)  {  return NULL;  }  current = (LinkListNode \*)tList;  for (i=0; i<pos; i++)  {  current = current->next;  }  ret = current->next;  return ret ;  }  返回第三个位置的  移动pos次以后，当前指针指向哪里？  答案：指向位置2，所以需要返回 ret = current->next;  备注：  循环遍历时， 遍历第1次，指向位置0  遍历第2次，指向位置1  遍历第3次，指向位置2  遍历第n次，指向位置n-1;  所以如果想返回位置n的元素的值，需要怎么做  ret = current->next;  此问题是：**指向头结点的指针移动n次 和 第n个元素之间的关系？** |
| 删除元素 |

**重要技术场景图**

#### 链表链式存储\_插入



#### 链表链式存储\_删除



### 2.3.3优点和缺点

优点：

无需一次性定制链表的容量

插入和删除操作无需移动数据元素

缺点：

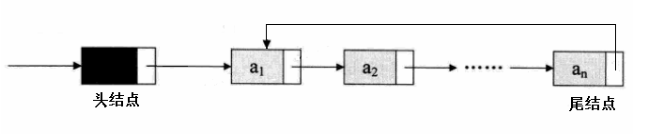
数据元素必须保存后继元素的位置信息

获取指定数据的元素操作需要顺序访问之前的元素

## 2.4循环链表

### 2.4.1基本概念

循环链表的定义：将单链表中最后一个数据元素的next指针指向第一个元素



|  |
| --- |
| 循环链表拥有单链表的所有操作 |
| 创建链表  销毁链表  获取链表长度  清空链表  获取第pos个元素操作  插入元素到位置pos  删除位置pos处的元素 |
| 新增功能：**游标**的定义 |
| 在循环链表中可以定义一个“当前”指针，这个指针通常称为游标，可以通过这个游标来遍历链表中的所有元素。 |
|  |
| 循环链表新操作 |
| 将游标重置指向链表中的第一个数据元素  CircleListNode\* CircleList\_Reset(CircleList\* list);  获取当前游标指向的数据元素  CircleListNode\* CircleList\_Current(CircleList\* list);  将游标移动指向到链表中的下一个数据元素  CircleListNode\* CircleList\_Next(CircleList\* list);  直接指定删除链表中的某个数据元素  CircleListNode\* CircleList\_DeleteNode(CircleList\* list, CircleListNode\* node);  // 根据元素的**值** 删除 元素 pk根据元素的**位置** 删除 元素 |

### 2.4.2循环链表的应用

#### 2.4.2.1 证明循环链表

打印两次。

#### 2.4.2.2 约瑟夫问题求解

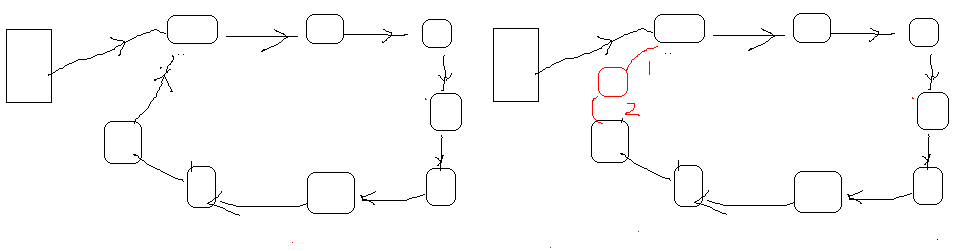
|  |
| --- |
| 约瑟夫问题-循环链表典型应用  n 个人围成一个圆圈，首先第 1 个人从 1 开始一个人一个人顺时针报数，报到第 m 个人，令其出列。然后再从下一 个人开始从 1 顺时针报数，报到第 m 个人，再令其出列，…，如此下去，求出列顺序。 |
|  |
| 大话数据结构 3.16结尾：写书的人，也很累；临界点 |
|  |

### 2.4.3设计与实现

#### 2.4.3.1循环链表插入元素的分析

1. 普通插入元素（和单链表是一样的）
2. 尾插法（和单链表是一样的，单链表的写法支持尾插法；

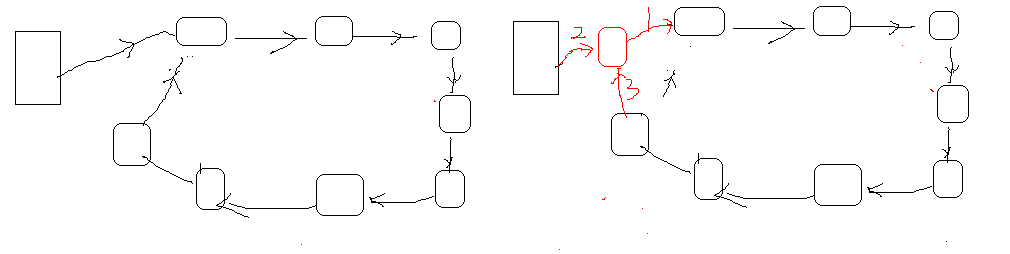
因：辅助指针向后跳length次，指向最后面那个元素）

 **CircleList\_Insert(list, (CircleListNode\*)&v1, CircleList\_Length(list));**

CircleList\_Insert(list, (CircleListNode\*)&v1, CircleList\_Length(list));

1. 头插法（要进行头插法，需要求出尾结点，和单链表不一样的地方，保证是循环链表）

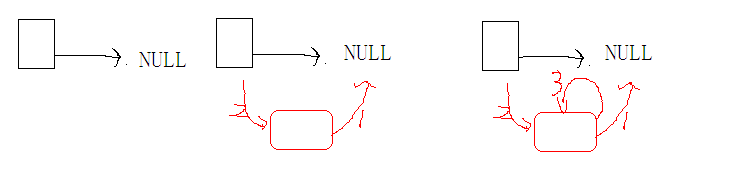
第一次插入元素时，让游标指向0号结点



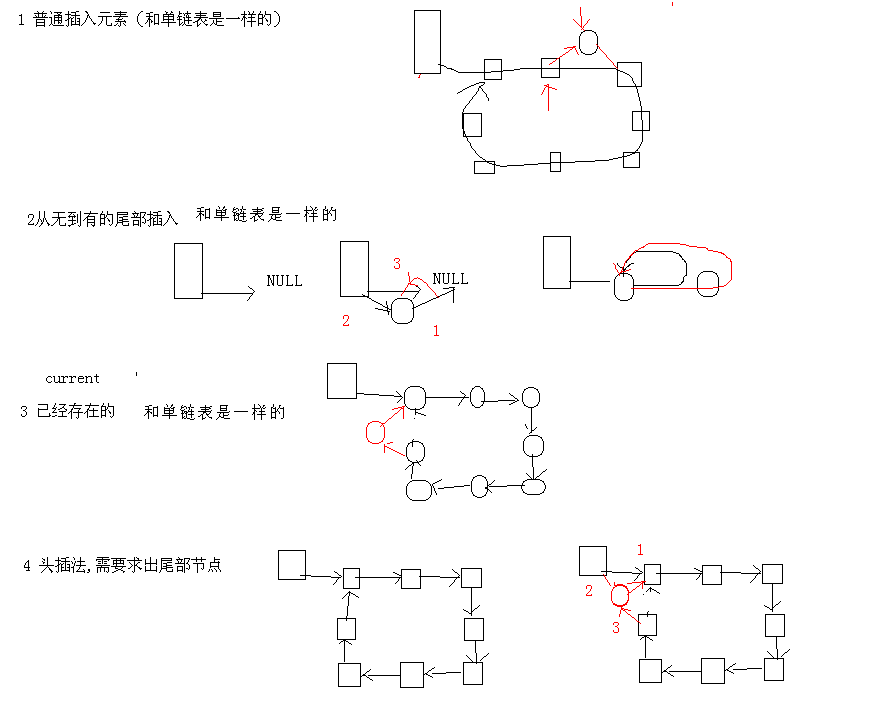
CircleList\_Insert(list, (CircleListNode\*)&v1, 0);

CircleList\_Insert(list, (CircleListNode\*)&v1, 0);

4)第一次插入元素

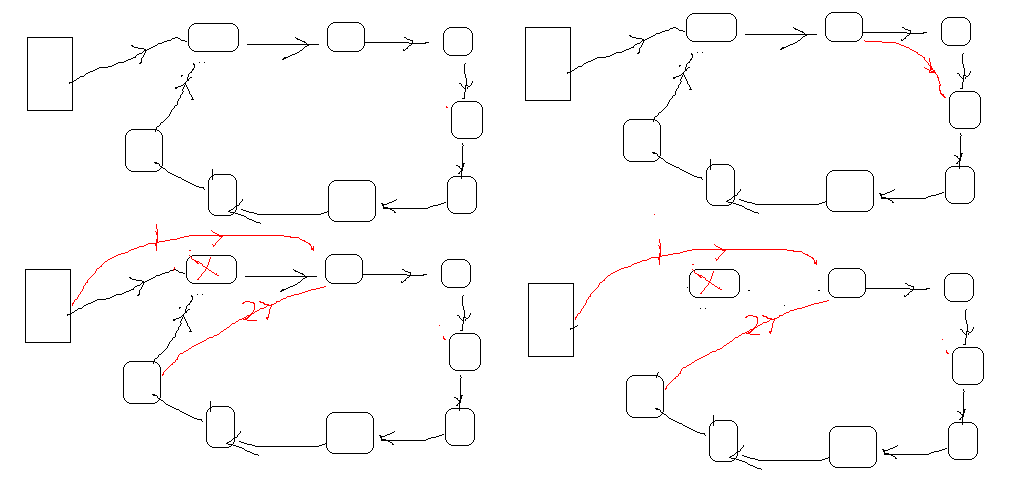


#### 2.4.3.2循环链表插入综合场景分析图



#### 2.4.3.3循环链表删除结点分析

1. 删除普通结点
2. 删除头结点（删除0号位置处元素），需要求出尾结点



### 2.4.4优点和缺点

|  |
| --- |
| 优点：功能强了。  循环链表只是在单链表的基础上做了一个加强 |
| 循环链表可以完全取代单链表的使用  循环链表的Next和Current操作可以高效的遍历链表中的所有元素 |
| 缺点：  代码复杂度提高了 |
| 大话数据结构 3.16结尾：写书的人，也很累；临界点 |

## 2.5双向链表

### 2.5.1基本概念

请思考： 为什么**需要**双向链表？

|  |
| --- |
| 单链表的结点都只有一个指向下一个结点的指针  单链表的数据元素无法直接访问其前驱元素  **逆序访问单链表**中的元素是极其**耗时**的操作！ |
| len = LinkList\_Length(list);  for (i=len-1; len>=0; i++) //O(n)  {  LinkListNode \*p = LinkList\_Get(list, i); //O(n)  //访问数据元素p中的元素  //  } |
| 双向链表的定义  在单链表的结点中增加一个指向其前驱的pre指针 |
|  |
| 双向链表拥有单链表的所有操作  创建链表  销毁链表  获取链表长度  清空链表  获取第pos个元素操作  插入元素到位置pos  删除位置pos处的元素 |

### 2.5.2设计与实现

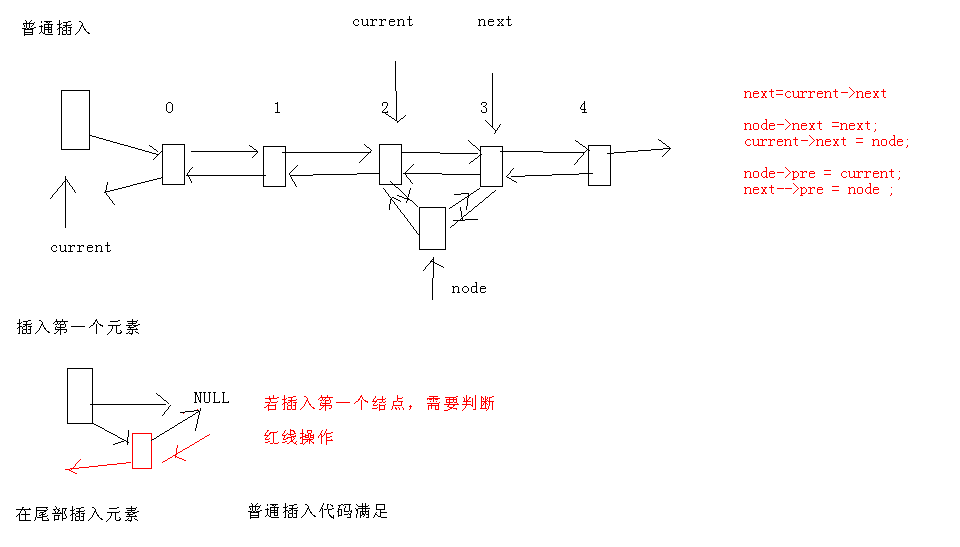
#### 循环链表一般操作

|  |
| --- |
| 插入操作 |
|  |
| 插入操作异常处理  插入第一个元素异常处理  在0号位置处插入元素； |
| 删除操作 |
|  |
| 删除操作异常处理 |
| 双向链表的新操作 |
| 获取当前游标指向的数据元素  将游标重置指向链表中的第一个数据元素  将游标移动指向到链表中的下一个数据元素  将游标移动指向到链表中的上一个数据元素  直接指定删除链表中的某个数据元素 |
| DLinkListNode\* DLinkList\_DeleteNode(DLinkList\* list, DLinkListNode\* node);  DLinkListNode\* DLinkList\_Reset(DLinkList\* list);  DLinkListNode\* DLinkList\_Current(DLinkList\* list);  DLinkListNode\* DLinkList\_Next(DLinkList\* list);  DLinkListNode\* DLinkList\_Pre(DLinkList\* list); |
| //大家一定要注意：教科书不会告诉你 项目上如何用；哪些点是项目的重点；  做一个企业级的财富库，完成你人生开发经验的积累，是我们的学习重点，要注意！ |

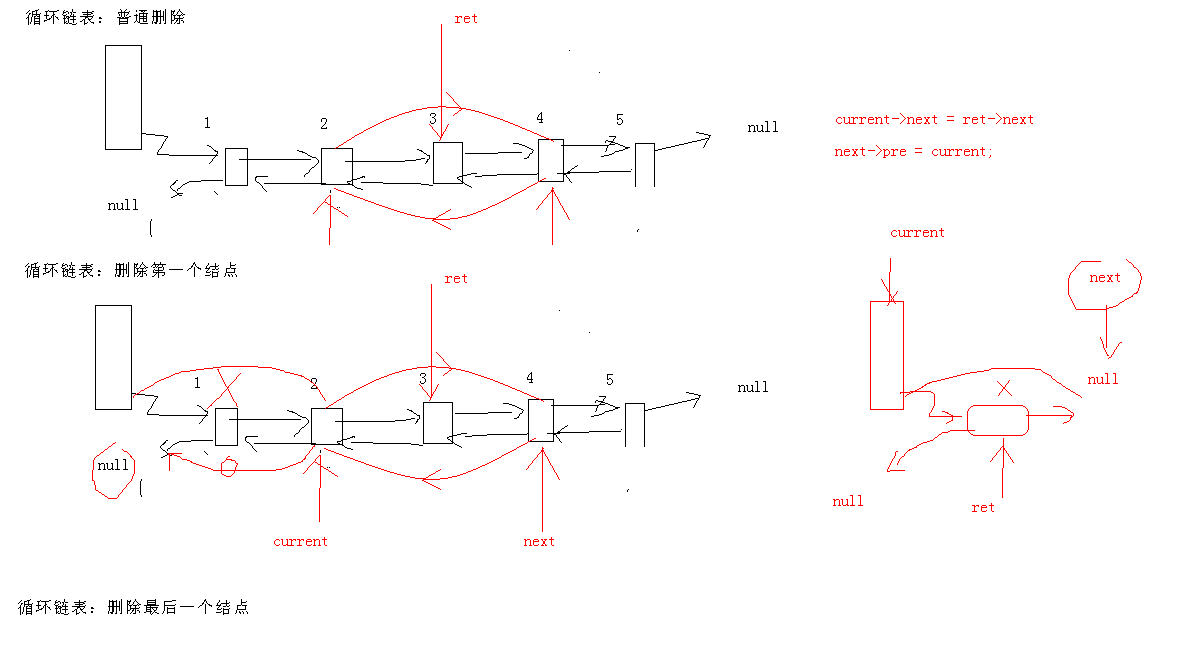
双向链表重要技术场景



#### 循环链表插入结点技术场景



#### 循环链表删除结点技术场景



### 2.5.3优点和缺点

|  |
| --- |
| 优点：双向链表在单链表的基础上增加了指向前驱的指针  功能上双向链表可以完全取代单链表的使用  双向链表的Next，Pre和Current操作可以高效的遍历链表中的所有元素 |
| 缺点：代码复杂 |

# 3、栈tack和队列queue

## 3.1栈stack

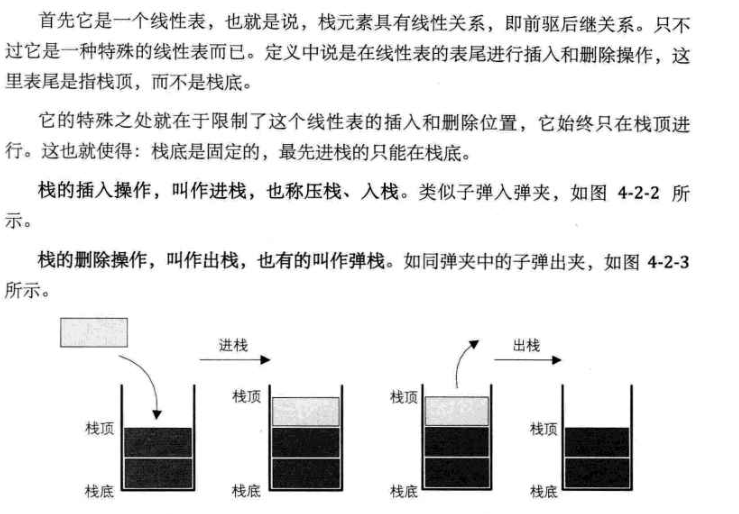
### 3.1.1Stack基本概念

栈是一种 特殊的线性表

栈仅能在线性表的一端进行操作

栈顶(Top)：允许操作的一端

栈底(Bottom)：不允许操作的一端



### 3.1.2Stack的常用操作

创建栈

销毁栈

清空栈

进栈

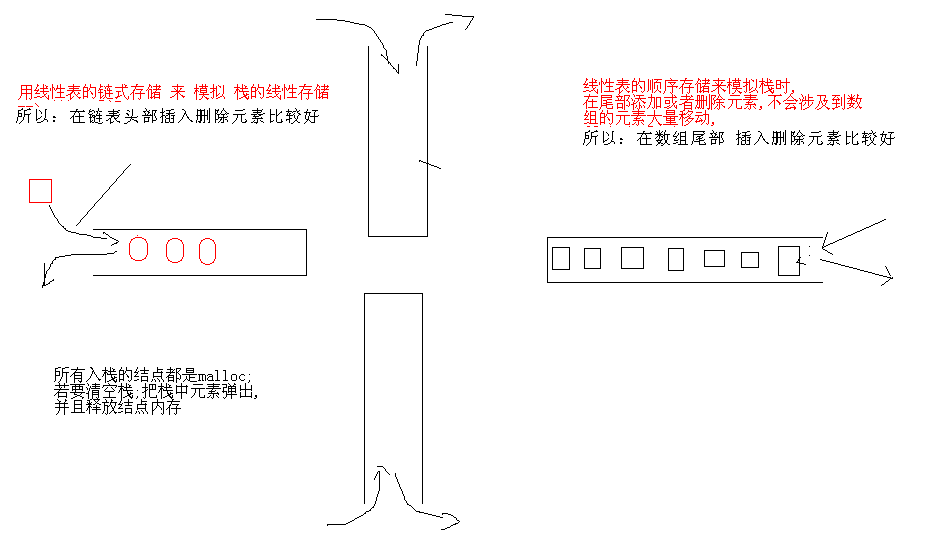
出栈

获取栈顶元素

获取栈的大小

|  |
| --- |
| C语言描述=====》栈的设计与实现 人生财富库积累 |
| #ifndef \_MY\_STACK\_H\_  #define \_MY\_STACK\_H\_  typedef void Stack;  Stack\* Stack\_Create();  void Stack\_Destroy(Stack\* stack);  void Stack\_Clear(Stack\* stack);  int Stack\_Push(Stack\* stack, void\* item);  void\* Stack\_Pop(Stack\* stack);  void\* Stack\_Top(Stack\* stack);  int Stack\_Size(Stack\* stack);  #endif //\_MY\_STACK\_H\_ |
|  |

### 3.1.3栈模型和链表模型关系分析



### 3.1.4栈的顺序存储设计与实现

#### 基本概念

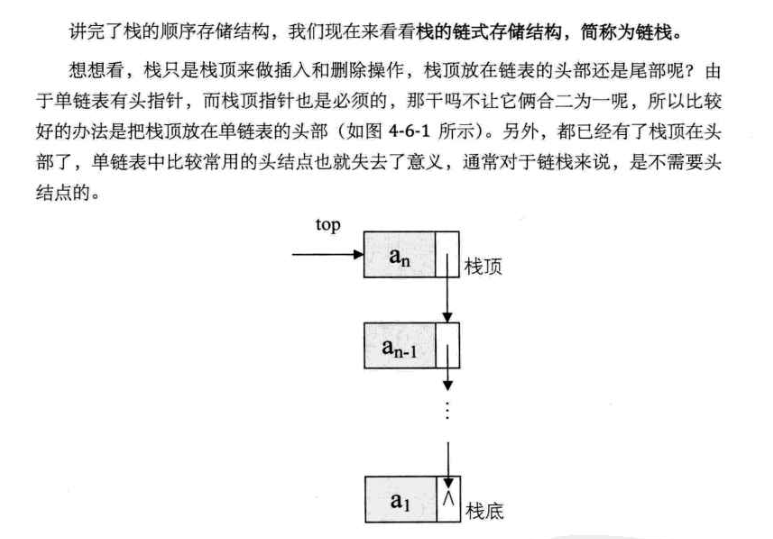
|  |
| --- |
|  |

#### 设计与实现

|  |
| --- |
| 头文件 |
| #ifndef \_\_MY\_SEQLIST\_H\_\_  #define \_\_MY\_SEQLIST\_H\_\_  typedef void SeqList;  typedef void SeqListNode;  SeqList\* SeqStack\_Create(int capacity);  void SeqStack \_Destroy(SeqStack \* list);  void SeqStack \_Clear(SeqStack \* list);  int SeqStack \_Length(SeqStack \* list);  int SeqStack \_Capacity(SeqStack \* list);  int SeqStack \_Insert(SeqStack \* list, SeqListNode\* node, int pos);  SeqListNode\* SeqList\_Get(SeqList\* list, int pos);  SeqListNode\* SeqList\_Delete(SeqList\* list, int pos);  #endif //\_\_MY\_SEQLIST\_H\_\_ |

### 3.1.5栈的链式存储设计与实现

#### 基本概念



#### 设计与实现

|  |
| --- |
| 头文件 |
| #ifndef \_MY\_LINKSTACK\_H\_  #define \_MY\_LINKSTACK\_H\_  typedef void LinkStack;  LinkStack\* LinkStack\_Create();  void LinkStack\_Destroy(LinkStack\* stack);  void LinkStack\_Clear(LinkStack\* stack);  int LinkStack\_Push(LinkStack\* stack, void\* item);  void\* LinkStack\_Pop(LinkStack\* stack);  void\* LinkStack\_Top(LinkStack\* stack);  int LinkStack\_Size(LinkStack\* stack);  #endif //\_MY\_LINKSTACK\_H\_ |

### 3.1.6栈的应用

#### 案例1：就近匹配

|  |
| --- |
| 应用1：就近匹配 |
| 几乎所有的编译器都具有检测括号是否匹配的能力  如何实现编译器中的符号成对检测？  #include <stdio.h> int main() { int a[4][4]; int (\*p)[4]; p = a[0]; return 0; |
| 算法思路  从第一个字符开始扫描  当遇见普通字符时忽略，  当遇见左符号时压入栈中  当遇见右符号时从栈中弹出栈顶符号，并进行匹配  匹配成功：继续读入下一个字符  匹配失败：立即停止，并报错  结束：  成功: 所有字符扫描完毕，且栈为空  失败：匹配失败或所有字符扫描完毕但栈非空 |
| 当需要检测成对出现但又互不相邻的事物时  可以使用栈“后进先出”的特性  栈非常适合于需要“就近匹配”的场合 |
|  |
| 计算机的本质工作就是做数学运算，那计算机可以读入字符串  “9 + (3 - 1) \* 5 + 8 / 2”并计算值吗？ |

#### 案例2：中缀表达式和后缀表达式

|  |
| --- |
| 应用2：中缀 后缀 |
| 计算机的本质工作就是做数学运算，那计算机可以读入字符串  “9 + (3 - 1) \* 5 + 8 / 2”并计算值吗？ |
| 后缀表达式 ==？符合计算机运算  波兰科学家在20世纪50年代提出了一种将运算符放在数字后面的后缀表达式对应的，  我们习惯的数学表达式叫做中缀表达式===》符合人类思考习惯 |
| 实例：  5 + 4=> 5 4 +  1 + 2 \* 3 => 1 2 3 \* +  8 + ( 3 – 1 ) \* 5 => 8 3 1 – 5 \* + |
| 中缀表达式符合人类的阅读和思维习惯  后缀表达式符合计算机的“运算习惯”  如何将中缀表达式转换成后缀表达式？ |
| 中缀转后缀算法： |
| 遍历中缀表达式中的数字和符号  对于数字：直接输出  对于符号：  左括号：进栈  运算符号：与栈顶符号进行优先级比较  若栈顶符号优先级低：此符合进栈 （默认栈顶若是左括号，左括号优先级最低）  若栈顶符号优先级不低：将栈顶符号弹出并输出，之后进栈  右括号：将栈顶符号弹出并输出，直到匹配左括号  遍历结束：将栈中的所有符号弹出并输出  中缀转后缀 |
|  |
| 计算机是如何基于后缀表达式计算的？  8 3 1 – 5 \* + |
| 遍历后缀表达式中的数字和符号  对于数字：进栈  对于符号：  从栈中弹出右操作数  从栈中弹出左操作数  根据符号进行运算  将运算结果压入栈中  遍历结束：栈中的唯一数字为计算结果 |
|  |
| 栈的神奇！  中缀表达式是人习惯的表达方式  后缀表达式是计算机喜欢的表达方式  通过栈可以方便的将中缀形式变换为后缀形式  中缀表达式的计算过程类似程序编译运行的过程 |
| 扩展：给你一个字符串，计算结果  “1 + 2 \* (66 / (2 \* 3) + 7 )”  1  字符串解析  词法语法分析  优先级分析  数据结构选型===》栈还是树？ |

## 3.2队列queue

### 3.2.1queue基本概念

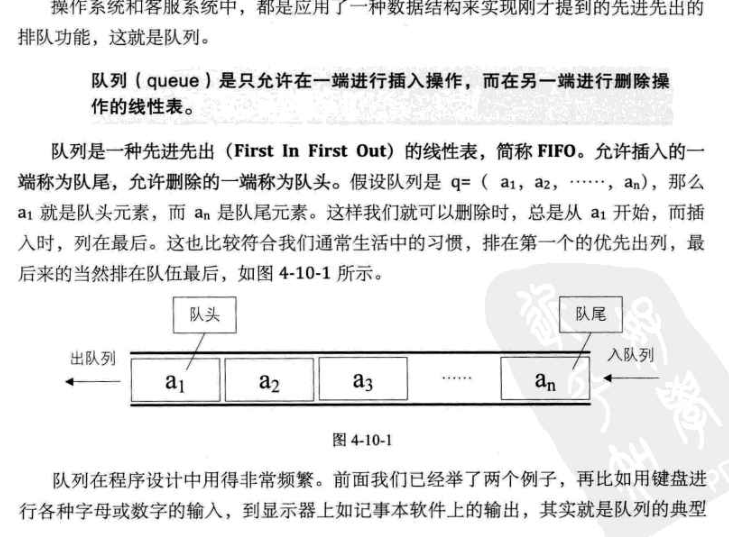
队列是一种特殊的线性表

队列仅在线性表的两端进行操作

队头(Front)：取出数据元素的一端

队尾(Rear)：插入数据元素的一端

队列不允许在中间部位进行操作！



### 3.2.2queue常用操作

销毁队列

清空队列

进队列

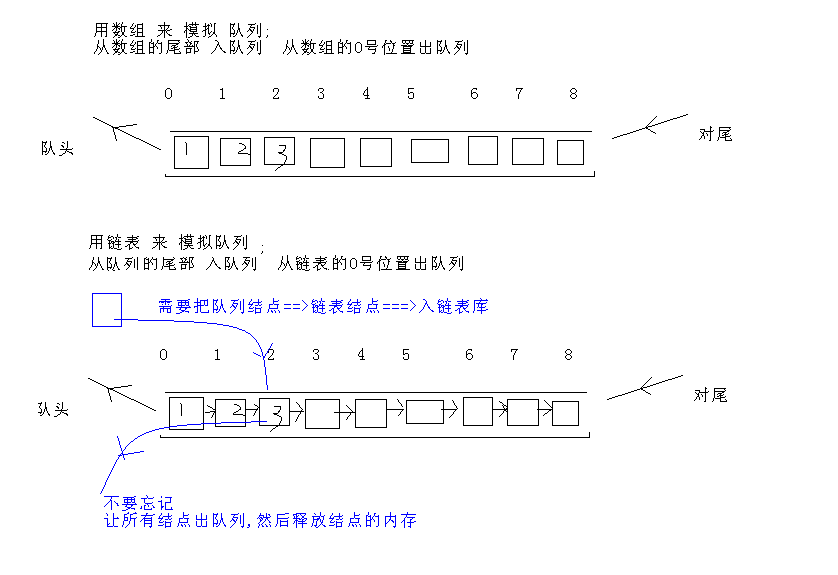
出队列

获取队头元素

获取队列的长度

|  |
| --- |
| C语言描述=====》队列的设计与实现 人生财富库积累 |
| #ifndef \_MY\_QUEUE\_H\_  #define \_MY\_QUEUE\_H\_  typedef void Queue;  Queue\* Queue\_Create();  void Queue\_Destroy(Queue\* queue);  void Queue\_Clear(Queue\* queue);  int Queue\_Append(Queue\* queue, void\* item);  void\* Queue\_Retrieve(Queue\* queue);  void\* Queue\_Header(Queue\* queue);  int Queue\_Length(Queue\* queue);  #endif //\_MY\_QUEUE\_H\_ |
|  |

### 3.2.3队列模型和链表模型关系分析



### 3.2.4队列的顺序存储设计与实现

#### 1基本概念

队列也是一种特殊的线性表；可以用线性表顺序存储来模拟队列。

#### 2设计与实现

|  |
| --- |
| #ifndef \_MY\_SEQQUEUE\_H\_  #define \_MY\_SEQQUEUE\_H\_  typedef void SeqQueue;  SeqQueue\* SeqQueue\_Create(int capacity);  void SeqQueue\_Destroy(SeqQueue\* queue);  void SeqQueue\_Clear(SeqQueue\* queue);  int SeqQueue\_Append(SeqQueue\* queue, void\* item);  void\* SeqQueue\_Retrieve(SeqQueue\* queue);  void\* SeqQueue\_Header(SeqQueue\* queue);  int SeqQueue\_Length(SeqQueue\* queue);  int SeqQueue\_Capacity(SeqQueue\* queue);  #endif //\_MY\_SEQQUEUE\_H\_ |
|  |

### 3.2.5队列的链式存储设计与实现

#### 1基本概念

队列也是一种特殊的线性表；可以用线性表链式存储来模拟队列的链式存储。

#### 2设计与实现

|  |
| --- |
| #ifndef \_MY\_LINKQUEUE\_H\_  #define \_MY\_LINKQUEUE\_H\_  typedef void LinkQueue;  LinkQueue\* LinkQueue\_Create();  void LinkQueue\_Destroy(LinkQueue\* queue);  void LinkQueue\_Clear(LinkQueue\* queue);  int LinkQueue\_Append(LinkQueue\* queue, void\* item);  void\* LinkQueue\_Retrieve(LinkQueue\* queue);  void\* LinkQueue\_Header(LinkQueue\* queue);  int LinkQueue\_Length(LinkQueue\* queue);  #endif //\_MY\_LINKQUEUE\_H\_ |

# 4、树专题

## 4.1树基本概念

|  |
| --- |
| 基础讲义： |
| 参考《数据结构\_树A.ppt》  参考《数据结构\_树B.ppt》 |
| 非线性结构，一个直接前驱，但可能有多个直接后继（1:n） |
| 树的定义  1）具有递归性，即树中还有树  2）m颗互不相交的集合 |
| 根 叶子 森林  有序树 无序树  双亲 孩子 兄弟 堂兄弟 祖先 子孙 |
| 结点 结点的度 结点的层次 终端结点 分支结点 |
| 树的度 **所有结点度中的最大值（Max{各结点的度}**  树的深度**指所有结点中最大的层数（Max{各结点的层次}**  (或高度) |

## 4.2树的表示法

|  |
| --- |
| 图形表示法 |
| 广义表表示法 |
| 左孩子－右兄弟表示法 |
| 双亲孩子表示法 |

## 4.3树的逻辑结构

|  |
| --- |
| 一对多（1:n），有多个直接后继（如家谱树、目录树等等），但只有一个根结点，且子树之间互不相交。 |
| 广义表表示法 |
| 左孩子－右兄弟表示法 |
|  |

## 4.4二叉树概念

### 4.4.1基本概念

|  |
| --- |
| 二叉树的结构最简单，规律性最强 |
| 可以证明，所有树都能转为唯一对应的二叉树，不失一般性 |
| 定义：是n（n≥0）个结点的有限集合，由一个根结点以及两棵互不相交的、分别称为左子树和右子树的二叉树组成 |
| 二叉树性质 |
| 性质1: 在二叉树的第i层上至多有2i-1个结点（i>0） |
| 性质2: 深度为k的二叉树至多有2k-1个结点（k>0） |
| 性质3: 对于任何一棵二叉树，若2度的结点数有n2个，则叶子数（n0）必定为n2＋1 （即n0=n2+1） |
| 满二叉树：一棵深度为*k* 且有2*k* -1个结点的二叉树。  （特点：每层都“充满”了结点） |
| 完全二叉树：深度为*k* 的*，*有*n*个结点的二叉树，当且仅当其每一个结点都与深度为*k* 的满二叉树中编号从1至*n*的结点一一对应。  理解：（k-1层与满二叉树完全相同，第k层结点尽力靠左）  数据结构数据：[数据结构(C语言版)].严蔚敏\_吴伟民.扫描版.pdf p124  数据结构数据：[数据结构(C语言版)].严蔚敏\_吴伟民.扫描版.pdf p127 |
|  |
| *性质4:* 具有n个结点的完全二叉树的深度必为⎣log2n⎦＋1  *性质5:* 对完全二叉树，若从上至下、从左至右编号，则编号为*i* 的结点，其左孩子编号必为*2i*，其右孩子编号必为*2i＋1*；其双亲的编号必为*i/2*（*i＝1* 时为根,除外） |
| ***二叉树的存储结构*** |
| 一、顺序存储结构  按二叉树的结点“自上而下、从左至右”编号，用一组连续的存储单元存储。  答：一律转为完全二叉树！  讨论：不是完全二叉树怎么办？  方法很简单，将各层空缺处统统补上“虚结点”，其内容为空 |
| **二、链式存储结构** |

### 4.4.2二叉树的表示

**二叉树表示法**

P127页

/\*

typedef struct BiTNode

{

int data;

struct BiTNode \*lchild, \*rchild;

}BiTNode, \*BiTree;

\*/

struct BiTNode

{

int data;

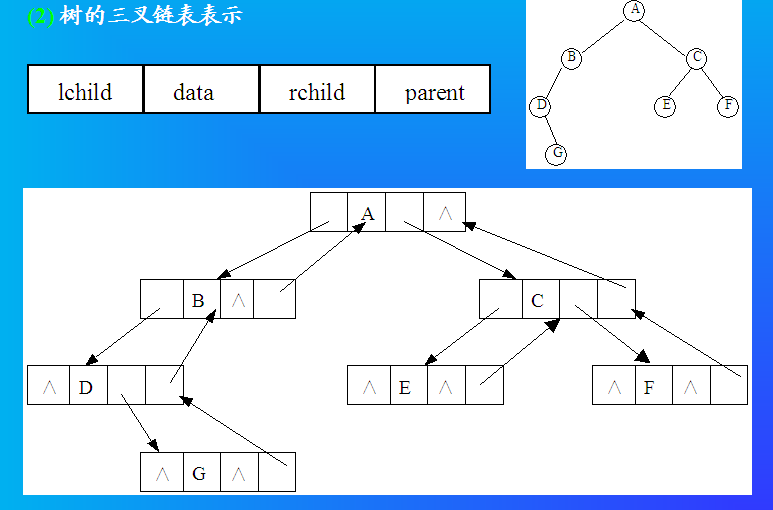
struct BiTNode \*lchild, \*rchild;

};

typedef struct BiTNode BiTNode;

typedef struct BiTNode \* BiTree;

**树的三叉链表表示**



//三叉链表

typedef struct TriTNode

{

int data;

//左右孩子指针

struct TriTNode \*lchild, \*rchild;

struct TriTNode \*parent;

}TriTNode, \*TriTree;

**双亲链表法**

//双亲链表

#define MAX\_TREE\_SIZE 100

typedef struct BPTNode

{

int data;

int parentPosition; //指向双亲的指针 //数组下标

char LRTag; //左右孩子标志域

}BPTNode;

typedef struct BPTree

{

BPTNode nodes[100]; //因为节点之间是分散的，需要把节点存储到数组中

int num\_node; //节点数目

int root; //根结点的位置 //注意此域存储的是父亲节点在数组的下标

}BPTree;

//用这个数据结构能表达出一颗树，为什么？

### 4.4.3二叉树的遍历

|  |
| --- |
|  |
| 树的遍历本质剖析 |
|  |

## 4.5二叉树编程实践

### 基本操作

|  |
| --- |
| typedef struct node{  int data;  struct node \*lchild,\*rchild；  } NODE;  NODE \*root; |
| DLR(NODE \*root )  {  if (root) //非空二叉树  {  printf(“%d”,root->data); //访问D  DLR(root->lchild); //递归遍历左子树  DLR(root->rchild); //递归遍历右子树  }  } |
| 中序遍历算法  LDR(NODE \*root)  { if(root !=NULL)  { LDR(root->lchild);  printf(“%d”,root->data);  LDR(root->rchild);  } } |
| 后序遍历算法  LRD (NODE \*root)  {if(root !=NULL)  {LRD(root->lchild);  LRD(root->rchild);  printf(“%d”,root->data);  } } |

### 案例1：计算二叉树中叶子结点的数目

int sum = 0; //全局变量

DLR\_CountLeafNum(NODE \*root)//采用中序遍历的递归算法

{

if ( root) //非空二叉树条件，还可写成if(root !=NULL )

{ if(!root->lchild&&!root->rchild) //是叶子结点则统计并打印

{ sum++; printf("%d\n",root->data); }

DLR\_CountLeafNum(root->lchild); //递归遍历左子树，直到叶子处；

DLR\_CountLeafNum(root->rchild);}//递归遍历右子树，直到叶子处；

} return(0);

}

思想： 1）求根结点左子树的叶子结点个数，累计到sum中，求根结点右子树的叶子结点个数累计到sum中。

2）若左子树还是树，重复步骤1；若右子树还是树，重复步骤1。

3）全局变量转成函数参数

4）按照先序、中序、后序方式计算叶子结点，

===》三种遍历的**本质**思想强化：**访问结点的路径**都是一样的，**计算结点的时机**不同。

### 案例2：求二叉树的深度

思想： 1）求根结点左子树高度，根结点右子树高度，比较的子树最大高度，再+1。

2）若左子树还是树，重复步骤1；若右子树还是树，重复步骤1。

### 案例3：完全Copy二叉树

思想： 1）malloc新结点，

2）拷贝左子树，拷贝右子树，让新结点连接左子树，右子树

3）若左子树还是树，重复步骤1、2；若右子树还是树，重复步骤1、2。

### 案例4：树的非递归遍历(中序遍历)

|  |
| --- |
| 中序 遍历的几种情况  分析1：什么时候访问根、什么时候访问左子树、什么访问右子树  当左子树为空或者左子树已经访问完毕以后，再访问根  访问完毕根以后，再访问右子树。  分析2：非递归遍历树，访问结点时，为什么是栈，而不是其他模型（比如说是队列）。  先走到的后访问、后走到的先访问，显然是栈结构  分析3：结点所有路径情况  步骤1：  如果结点有左子树，该结点入栈；  如果结点没有左子树，访问该结点；  步骤2：  如果结点有右子树，重复步骤1；  如果结点没有右子树（结点访问完毕），根据栈顶指示回退，访问栈顶元素，并访问右子树，重复步骤1  如果栈为空，表示遍历结束。  注意：入栈的结点表示，本身没有被访问过，同时右子树也没有被访问过。  分析4：有一个一直往左走入栈的操作，中序遍历的起点 |
|  |
|  |
|  |
| 作业：自己编写堆栈函数原型，实现中序遍历非递归算法 |
|  |

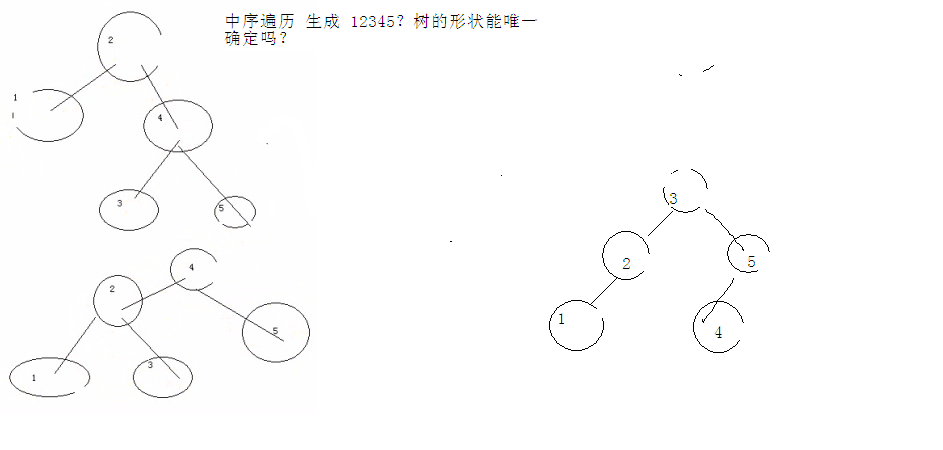
## 4.6二叉树的创建

### 4.6.1中序和先序创建树

1、根据中序遍历的结果能确定一棵树吗？

中序遍历：结果为：“12345”，这个“12345”能确定一棵树吗？

请思考，会有多少种形状。



2、如何才能确定一棵树？

结论： 通过中序遍历和先序遍历可以确定一个树

通过中序遍历和后续遍历可以确定一个树

通过先序遍历和后序遍历确定不了一个树。

单独先序遍历：能求解根，但不能求解左子树什么时候结束、右子树什么时候开始。

3、根据先序和中序结果画树

|  |
| --- |
| 算法  1、通过先序遍历找到根结点A，再通过A在中序遍历的位置找出左子树，右子树  2、在A的左子树中，找左子树的根结点（在先序中找），转步骤1  3、在A的右子树中，找右子树的根结点（在先序中找），转步骤1 |
|  |
| 讲解：  先序遍历结果：ADEBCF  中序遍历结果：DEACFB |
| 练习：  先序遍历结果：ABDHKECFIGJ  中序遍历结果：HKDBEAIFCGJ |

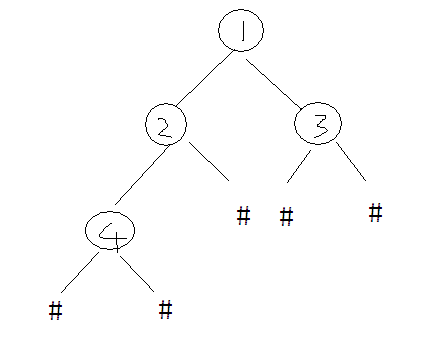
4、学习算法可借助工具、动画

光盘有2个；C++有1个

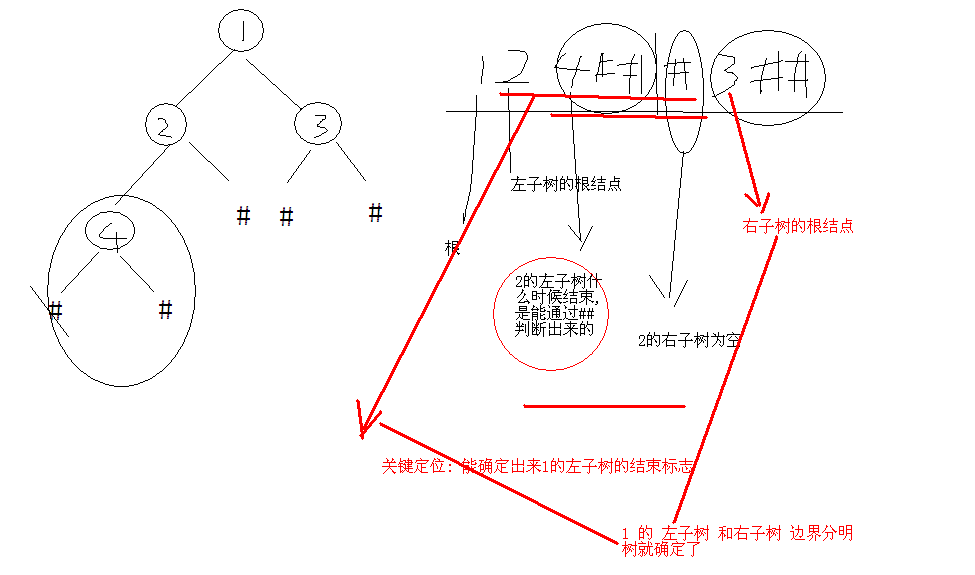
### 4.6.2#号法创建树

1、什么是#号法创建树

#创建树，让树的每一个节点都变成度数为2的树



先序遍历：124###3## 可以唯一确定一棵树吗，为什么？



2 #创建树练习

先序遍历：ABDH#K###E##CFI###G#J##,请画出树的形状

#号法画出树关键点：

要清楚的确定**左子树什么结束**，**右子树什么时候开始**。

3、#号法编程实践

利用前序遍历来建树（结点值陆续从键盘输入，用DLR为宜）

Bintree createBTpre( )

{

Bintree T;

char ch;

scanf(“%c”,&ch);

if(ch==’#’)

T=NULL;

else

{ T=( Bintree )malloc(sizeof(BinTNode));

T->data=ch;

T->lchild=createBTpre();

T->rchild=createBTpre();

}

return T;

}

//后序遍历销毁一个树

void BiTree\_Free(BiTNode\* T)

{

BiTNode \*tmp = NULL;

if (T!= NULL)

{

if (T->rchild != NULL) BiTree\_Free(T->rchild);

if (T->lchild != NULL) BiTree\_Free(T->lchild);

if (T != NULL)

{

free(T);

T = NULL;

}

}

}

## 4.7二叉线索树

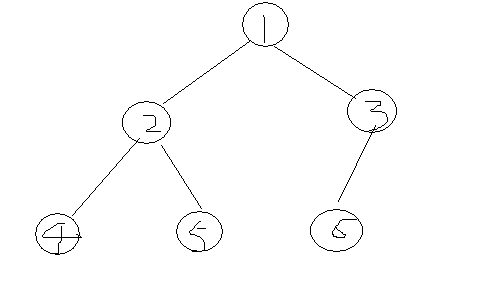
### 4.7.1线索化概念

1、前言

普通二叉树只能找到结点的左右孩子信息，而该结点的直接前驱和直接后继只能在遍历过程中获得。

若可将遍历后对应的有关前驱和后继预存起来，则从第一个结点开始就能很快“顺藤摸瓜”而遍历整个树了。

二叉线索树思想是干什么的？



中序遍历这棵树===》转换成链表访问

2线索化思想

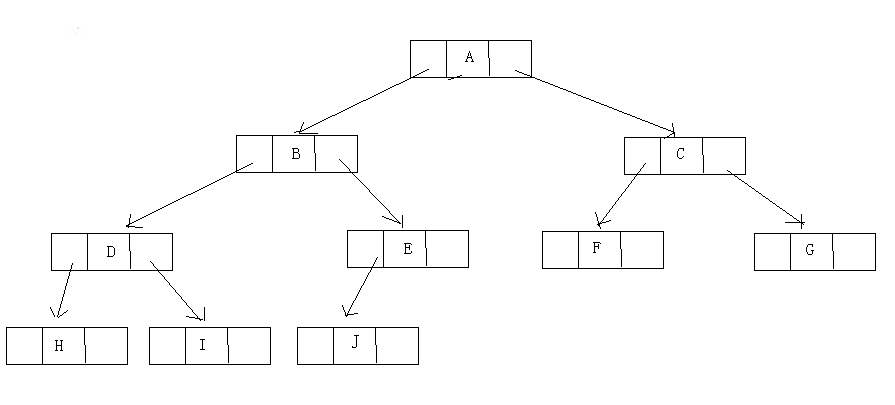
|  |
| --- |
|  |
|  |
|  |

结论：线索化过程就是在遍历过程（假设是中序遍历）中修改空指针的过程：

将空的lchild改为结点的直接前驱；

将空的rchild改为结点的直接后继。

3线索化思想训练

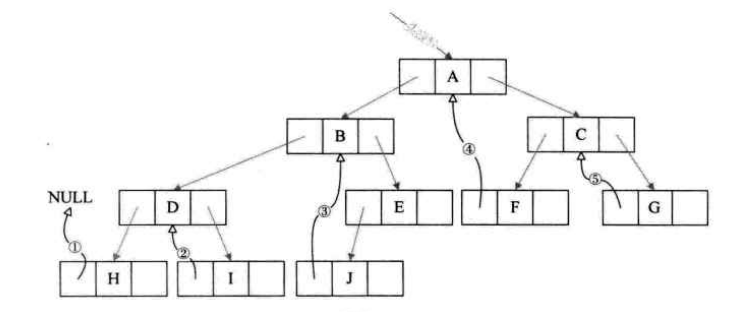


请将此树线索化。

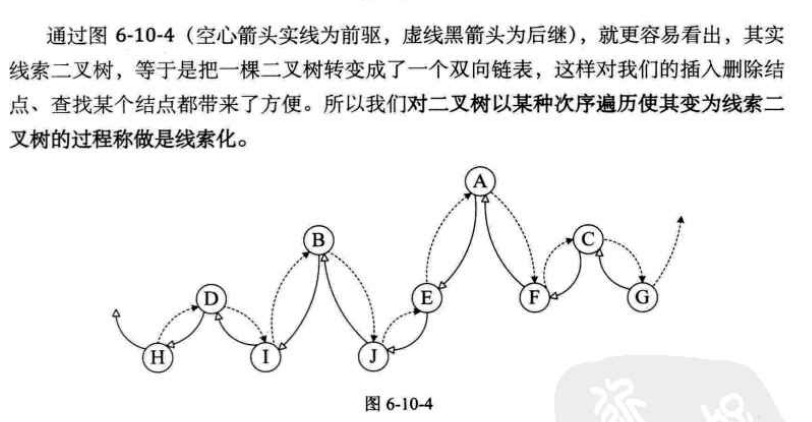
1）右空指针线索化：



2）左空指针线索化



3）总结



### 4.7.2线索化的实现

1）线索化树结点

typedef struct BiThrNode /\* 二叉线索存储结点结构 \*/

{

char data; /\* 结点数据 \*/

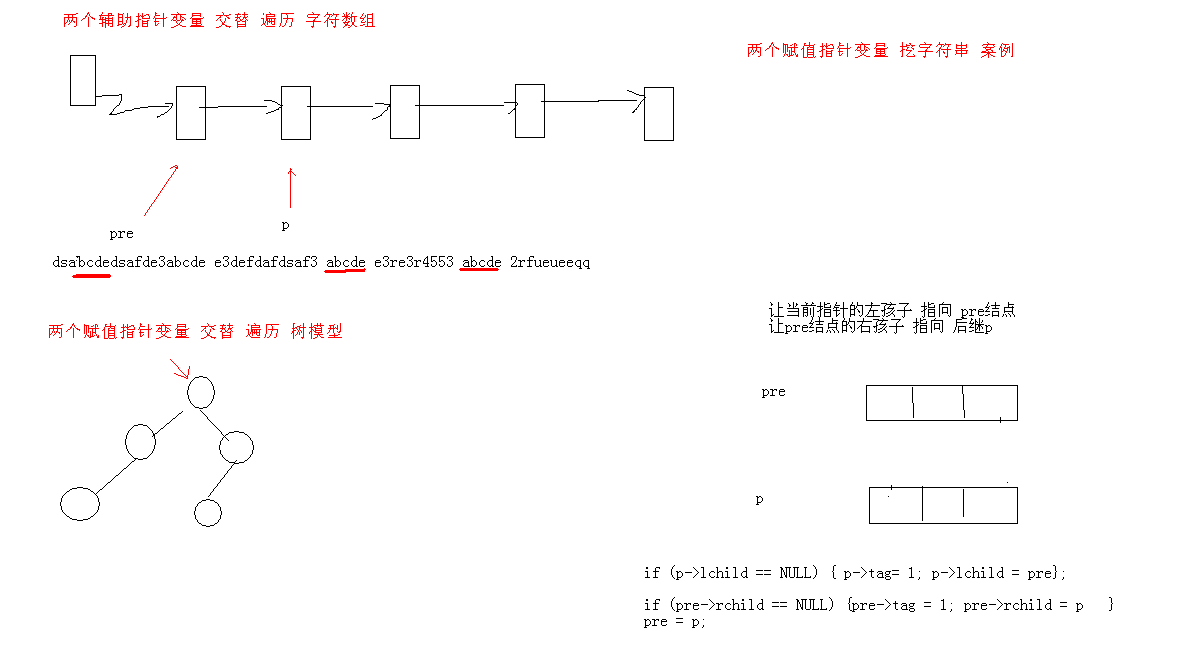
struct BiThrNode \*lchild, \*rchild; /\* 左右孩子指针 \*/

int LTag;

int RTag; /\* 左右标志 \*/

} BiThrNode, \*BiThrTree;

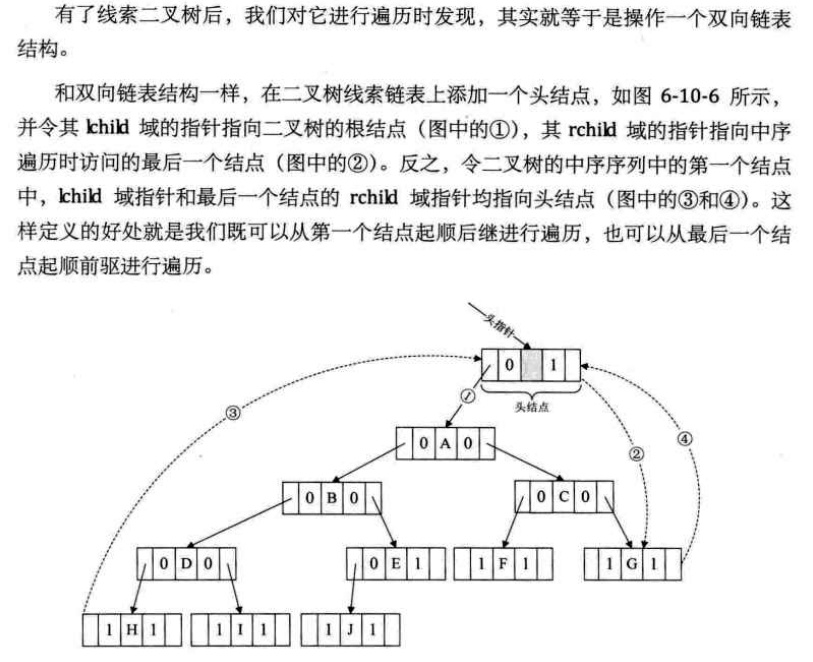
**2）线索化思想分析**

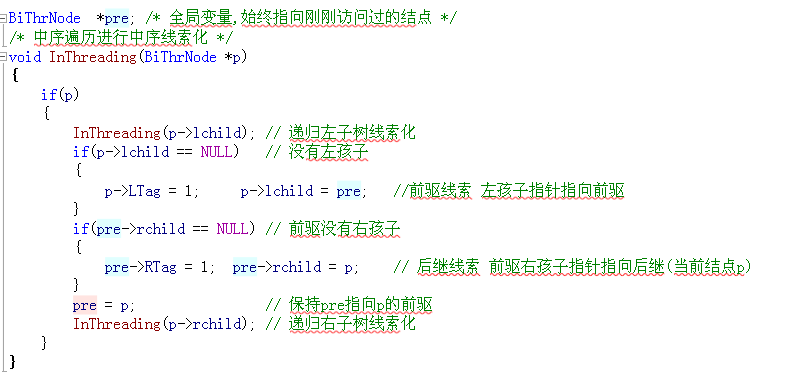


线索化的本质：让前后结点，建立关系；

1）两个辅助指针变量形成差值后：后继结点的左孩子指向前驱结点，前驱结点的右孩子指向后继结点。

2）赋值指针变量和业务操作的逻辑关系





1. 二叉树线索化树的遍历

/\* 中序遍历二叉线索树T(头结点)的非递归算法 \*/

int InOrderTraverse\_Thr(BiThrNode\* T)

{

BiThrNode\* p;

p = T->lchild; /\* p指向根结点 \*/

while (p != T)

{

/\* 空树或遍历结束时,p==T \*/

while (p->LTag == Link)

p = p->lchild;

printf("%c ", p->data);

while (p->RTag==Thread && p->rchild!=T)

{

p = p->rchild;

printf("%c ", p->data);

}

p=p->rchild;

}

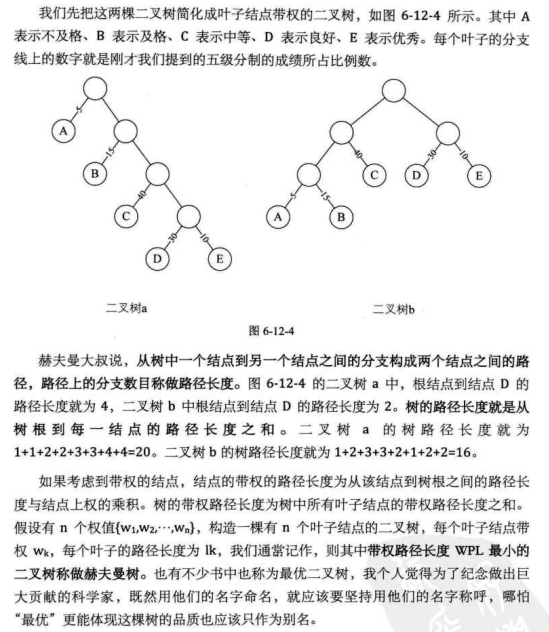
return 0;

}

## 4.8霍夫曼树

### 4.8.1概念

组建一个网络，耗费最小 WPL最小；这个方法是霍夫曼想出来的，称为霍夫曼树



### 4.8.2霍夫曼树的构造

|  |
| --- |
| 对于文本”BADCADFEED”的传输而言，因为重复出现的只有  ”ABCDEF”这6个字符，因此可以用下面的方式编码： |
|  |
| 接收方可以根据每3个bit进行一次字符解码的方式还原文本信息。  这样的编码方式需要30个bit位才能表示10个字符  那么当传输一篇500个字符的情报时，需要15000个bit位  在战争年代，这种编码方式对于情报的发送和接受是很低效且容易出错的。  如何提高收发效率？ |
| 要提高效率，必然要从编码方式的改进入手，要避免每个字符都占用相同的bit位 |
|  |
| 准则：任一字符的编码都不是另一个字符编码的前缀！  也就是说：每一个字符的编码路径，都不包含另外一个字符的路径。 |
| 霍夫曼树  1.给定n个数值{ v1, v2, …, vn}  2.根据这n个数值构造二叉树集合F  F = { T1, T2, …, Tn}  Ti的数据域为vi，左右子树为空  3.在F中选取两棵根结点的值最小的树作为左右子树构造一棵新的二叉树，这棵二叉树的根结点中的值为左右子树根结点中的值之和  4.在F中删除这两棵子树，并将构造的新二叉树加入F中  5.重复3和4，直到F中只剩下一个树为止。这棵树即霍夫曼树 |
| 假设经过统计ABCDEF在需要传输的报文中出现的概率如下 |
|  |
| 霍夫曼树是一种特殊的二叉树  霍夫曼树应用于信息编码和数据压缩领域  霍夫曼树是现代压缩算法的基础 |

# 5、排序

## 5.1排序基本概念

|  |
| --- |
| 现实生活中排序很重要 |
| 算法已写好代码复用 & 和我们需要学习前辈们的经验 不矛盾，也不代表我们不需要不学习。 |
| 排序是计算机内经常进行的一种操作，其目的是将一组“无序”的数据元素调整为“有序”的数据元素。 |
| 排序数学定义：假设含n个数据元素的序列为{ R1, R2, …, Rn}  其相应的关键字序列为{ K1, K2, …, Kn}这些关键字相互之间可以进行比较，即在它们之间存在着这样一个关系 ：Kp1≤Kp2≤…≤Kpn  按此固有关系将上式记录序列重新排列为{ Rp1, Rp2, …，Rpn}的操作称作排序 |
| 排序的稳定性 |
| 如果在序列中有两个数据元素r[i]和r[j]，它们的关键字k[i] == k [j]，且在排序之前，对象  r[i]排在r[j]前面。如果在排序之后，对象r[i]仍在r[j]前面，则称这个排序方法是稳定的；  否则称这个排序方法是不稳定的。 |
| 多关键字排序 |
| 排序时需要比较的关键字多余一个  排序结果首先按关键字1进行排序  当关键字1相同时按关键字2进行排序  当关键字n-1相同时按关键字n进行排序  对于多关键字排序，只需要在比较操作时同时考虑多个关键字即可！ |
| 排序中的关键操作 |
| 比较  任意两个数据元素通过比较操作确定先后次序  交换  数据元素之间需要交换才能得到预期结果 |
| 内排序和外排序 |
| 内排序  整个排序过程不需要访问外存便能完成  外排序  待排序的数据元素数量很大，整个序列的排序过程不可能在内存中完成 |
| 排序的审判 |
| 时间性能  关键性能差异体现在比较和交换的数量  辅助存储空间  为完成排序操作需要的额外的存储空间  必要时可以“空间换时间”  算法的实现复杂性  过于复杂的排序法会影响代码的可读性和可维护性，也可能影响排序的性能 |
| 总结  排序是数据元素从无序到有序的过程  排序具有稳定性，是选择排序算法的因素之一  比较和交换是排序的基本操作  多关键字排序与单关键字排序无本质区别  排序的时间性能是区分排序算法好坏的主要因素 |

## 5.2选择法

|  |
| --- |
| 基本思想 |
| 每一趟 (例如第 i 趟，i = 0, 1, …,n-2)在后面 n-i个待排的数据元素中选出关键字  最小的元素, 作为有序元素序列的第 i 个元素。 |
|  |
| 排序过程  首先通过n-1次关键字比较，从n个记录中找出关键字最小的记录，将它与第一个记录交换  再通过n-2次比较，从剩余的n-1个记录中找出关键字次小的记录，将它与第二个记录交换  重复上述操作，共进行n-1趟排序后，排序结束 |
|  |
|  |

## 5.3插入排序

|  |
| --- |
| 基本思想：  元素1个元素， |
| 排序过程：整个排序过程为n-1趟插入，即先将序列中第1个记录看成是一个有序子序列，然后从第2个记录开始，逐个进行插入，直至整个序列有序  实质：对线性表执行n-1次插入操作，只是先要找到插入位置 |
| V[0], V[1], …, V[i-1]已经排好序。这时已经排好序。这时,用V[i]的关键字与V[i-1], V[i-2], …的关键字进行比较, 找到插入位置即将V[i]]插入, 原来位置上的对象向后顺移。  插入排序关键点：1、拿出一个元素，留出位置、2 符合条件的元素后移 |
|  |
|  |

## 5.4冒泡排序

|  |
| --- |
|  |
|  |
|  |

## 5.5希尔排序

|  |
| --- |
| 排序过程：先取一个正整数d1<n，把所有相隔d1的记录放一组，组内进行直接插入排序；然后取d2<d1，重复上述分组和排序操作；直至di=1，即所有记录放进一个组中排序为止  O(n-1.3)  Q(nlogn)  希尔排序是不稳定的。 |
|  |
|  |

## 5.6快速排序

|  |
| --- |
| 思想：  快速排序是对冒泡排序的一种改进。它的基本思想是：  通过一躺排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，基准数据排在这两个子序列的中间；  然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。 |
|  |
|  |
|  |
| // O(n\*logn)  不稳定，分组，后面的有可能跑到前面去了。  21 100 3 50 1  3 1 21 100 50  1 3 21 50 100 |

## 5.7归并排序

|  |
| --- |
| 思想： |
| 注意：一个元素，可以看做有序的，是稳定的算法 |
|  |
|  |
|  |
| 对一个数组分成两路，mid中间 |
|  |
| 设两个有序的子文件(相当于输入堆)放在同一向量中相邻的位置上：R[low..m],R[m+1..high],先将它们合并到一个局部的暂存向量R1(相当于输出堆)中,待合并完成后将R1复制回R[low..high]中。 |
|  |
|  |
|  |

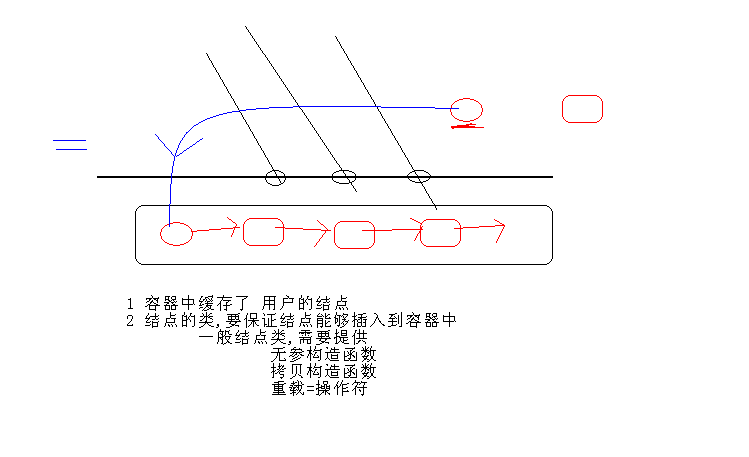
## 5.8排序总结

|  |
| --- |
|  |

# 6 C++模板类与数据结构基础

## 6.1前言

C++模板是容器的概念。



**理论提高：**所有容器提供的都是值（value）语意，而非引用（reference）语意。**容器执行插入元素的操作时，内部实施拷贝动作。**所以STL容器内存储的元素必须**能够被拷贝**（必须提供拷贝构造函数）。

加入到容器中的元素，应该可以被加入才行。

## 6.2模板类设计与实现

链表类\_链式存储设计与实现

栈类\_链式存储设计与实现

队列类\_链式存储设计与实现

链表类\_顺序存储设计与实现

栈类\_顺序存储设计与实现

队列类\_顺序存储设计与实现