1.数据结构概念

1.1 数据结构相关概念

1.1.1 为什么要学习数据结构?

为什么要学习数据结构？在回答这个问题之前，我们是不是应该弄清楚什么是数据结构，数据结构能够用来做什么？最直白的，它能够帮我们解决什么问题？

我们之前的学习的设计模式和现在数据结构，有个相似的特点，他们两个都是在思想层面的东西，和具体的语言无关，你可以用其他的语言去实现这些思想都没有问题。

设计模式在教我们如何编写代码，让我们的代码具有可扩展性，灵活性，可复用性，这个是编码层次上的，那么数据结构呢？我们举一个例子:

比如我们 C 语言中没有数组这么个数据结构，那么你如何实现 10 个数排序呢？是不是要定义 10 个变量，然后让 10 个变量互相比较，重复劳动，但是你用数组之后，是不是问题变得简单了，只需要通过数组下表就可以，提高了程序的编写效率。

再比如说，我们有了数组了，为什么还需要学习链表这种数据结构？数组是连续内存空间，一旦定义了不能概念，适应性差，但是链表你有多少数据，我就创建多少个结点，而且比如说数据，你删除中间位置一个元素，会引起后面数据的移动，但是链表不会啊，在有些场合下，你使用链表是不是会增加程序的效率。

从我们的讲的东西中，也可以得出数据结构的概念，数据结构就是帮我们解决如何组织和存储数据的方式。

数据结构主要**研究非数值计算问题的程序中的操作对象以及他们之间的关系，不是研究**

复杂的算法。

数据结构是计算机存储、组织数据的方式。

1.1.2 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。

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）。

```
//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 main()
{
    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;
}
```

1.2 算法

1.2.1 算法的概念

为啥子我们学习数据结构还要了解算法？

比如说:我有 10 个学生，我们将是 10 个学生保存在一个链表中，但是我不能把学生保存进去就完事了吧？我放进去是为了使用这些个数据完成一定的业务需求，比如按成绩大小排序并显示，比如计算这些学生的平均分等等，这些才是我们最终要解决的问题，既然要解决问题，那么就需要一些算法，比如排序算法，比如计算平均分的算法，对吧？所以数据结构和算法是互相配合完成工作。

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。

- **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，并且每一个步骤可以在接收的实际内完成
- **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，也就是说每一步都能够执行有限的次数完成。

问题:针对某一具体的问题，解决此问题的算法是唯一的吗？

比如说:求从 1 到 100 的和？

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
//算法 1
long sum1(int n){
    long ret = 0;
    int* array = (int*)malloc(n * sizeof(int));
    int i = 0;
    for (int i = 0; i < n; i ++){
        array[i] = i + 1;
    }
    for (int i = 0; i < n;i++){
        ret += array[i];
    }
    free(array);

    return ret;
}
//算法 2
long sum2(int n){
    long ret = 0;
```

```
    int i = 0;
    for (i = 1; i <= n;i++){
        ret += i;
    }
    return ret;
}
//算法 3
long sum3(int n){
    long ret = 0; // 1
    if (n > 0){
        ret = (1 + n)*n / 2;
    }
    return ret;
}
int main(){
    printf("%d", sum1(100));
    printf("%d", sum2(100));
    printf("%d", sum3(100));
    system("pause");
    return EXIT_SUCCESS;
}
```

同样一个问题，我有三种不同的算法，这三种算法都可以解决同样的问题，那么我们如何选择？需要有个方法来衡量算法的效率吧？

1.2.4 算法效率的度量

1.2.4.1 事后统计法

主要通过设计好的测试程序和数据，利用计算机的计时器对不同算法的编制的程序的运行时间进行比较，从而确定算法效率的高低。

- 统计方法

  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

- 总结
  - 事后统计法虽然直观，但是实施困难且缺陷多

1.2.4.2 事前分析估算

在计算机程序编制前，依据统计方法对算法进行估算。

- 统计方法：

  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

算法推倒的理论基础：

- 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
- 通过具体的步骤，就可以推导出算法的复杂度（如下表）

次数	算法 C（4n+8）	算法 C^1（n）	算法 D（$2n^2+1$）	算法 $D^1(n^2)$
n = 1	12	1	3	1
n = 2	16	2	9	4

n = 3	20	3	19	9
n = 10	48	10	201	100
n = 100	408	100	20001	10000
n = 1000	4008	1000	2000001	1000000

次数	算法 G（$2n^2$）	算法 H（3n+1）	算法 I（$2n^2$+3n+1）
n = 1	2	4	6
n = 2	8	7	15
n = 5	50	16	66
n = 10	200	31	231
n = 100	20000	301	20301
n = 1000	2000000	3001	2003001
n = 10000	2000000000	30001	200030001
n = 100000	20000000000	300001	20000300001
n = 1000000	2000000000000	3000001	2000003000001

怎么判断一个算法的效率?（规则如下）

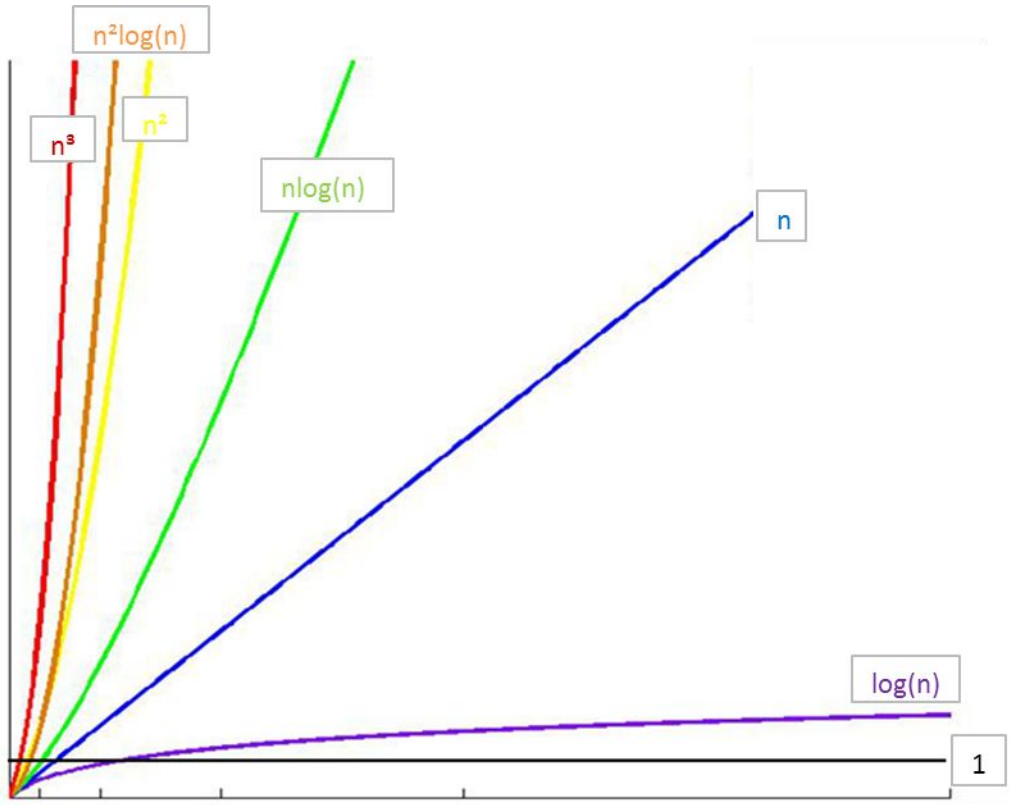
- 判断一个算法的效率时，往往**只需要关注操作数量的最高次项，其它次要项和常数项可以忽略。**
- 在没有特殊说明时，我们所分析的**算法的时间复杂度都是指最坏时间复杂度**。
- **只有常数项记做 1**
- **操作数量的估算可以作为时间复杂度的估算**

1.2.4.3 大 O 表示法

- 算法的时间复杂度
  - 常见的时间复杂度

执行次数函数	阶	非正式术语
12	O(1)	常数阶
2n+3	O(n)	线性阶
$3n^2+2n+1$	$O(n^2)$	平方阶
$5\log_2 n+20$	O(logn)	对数阶
$2n+3n\log_2 n+19$	O(nlogn)	nlogn 阶
$6n^3+2n^2+3n+4$	$O(n^3)$	立方阶
2^n	$O(2^n)$	指数阶

- 常见的时间复杂度之间的关系



常用的时间复杂度所耗费的时间从小到大依次是：

$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

- 时间复杂度练习(参考算法的效率规则判断)

  - $O(5) = O(1)$

- $O(2n + 1) = O(n)$
- $O(6n^2 + n + 1) = O(n^2)$
- $O(3n^3 + 1) = O(n^3)$

总结:

- **只关注最高次项**
- **时间复杂度是指最坏时间复杂度**
- **只有常数项记做 1**

- 算法的空间复杂度

  算法的空间复杂度并不是计算所有算法所占的空间，而是使用的辅助空间的大小。

1.2.4.4 大 O 表示法练习题

看数据结构练习题 PDF.

2.线性表

2.1 线性表基本概念

2.1.1 基本概念

线性表是零个或者多个数据元素的有限序列。

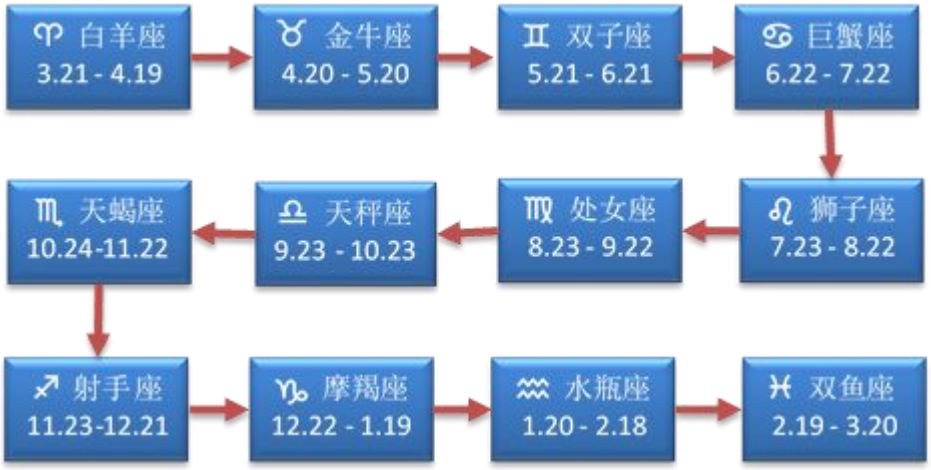
特性:

数据元素之间是**有顺序的.**

数据元素个数是**有限的.**

数据元素的**类型必须相同.**

例：先来看一个大家感兴趣的话题，一年里的星座列表，是不是线性表呢？如图所示：



2.1.1 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（a0,a1, a2, …, an）ai 是表项，n 是表长度。

2.1.2 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。

2.1.3 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

线性表的抽象数据类型定义：

```
ADT 线性表（List）
Data
    线性表的数据对象集合为{ a1, a2, ……, an }，每个元素的类型均为 DataType。其中，除第一个元素 a1 外，每个元素有且只有一个直接前驱元素，除了最后一个元素 an 外，每个元素有且只有一个直接后继元素。数据元素之间的关系是一一对应的。

Operation（操作）
// 初始化，建立一个空的线性表 L。
InitList(*L);
// 若线性表为空，返回 true，否则返回 false
ListEmpty(L);
// 将线性表清空
ClearList(*L);
// 将线性表 L 中的第 i 个位置的元素返回给 e
GetElem(L, i, *e);
// 在线性表 L 中的第 i 个位置插入新元素 e
ListInsert(*L, i, e);
// 删除线性表 L 中的第 i 个位置元素，并用 e 返回其值
ListDelete(*L, i, *e);
// 返回线性表 L 的元素个数
ListLength(L);
// 销毁线性表
DestroyList(*L);
```

2.2 线性表的顺序存储

2.2.1 线性表顺序存储基本概念

线性表的顺序存储结构，指的是**用一段地址连续的存储单元依次存储线性表的数据元素**。

线性表 （a1，a2，……，an）的顺序存储示意图如下：

2.2.2 线性表顺序存储的设计与实现

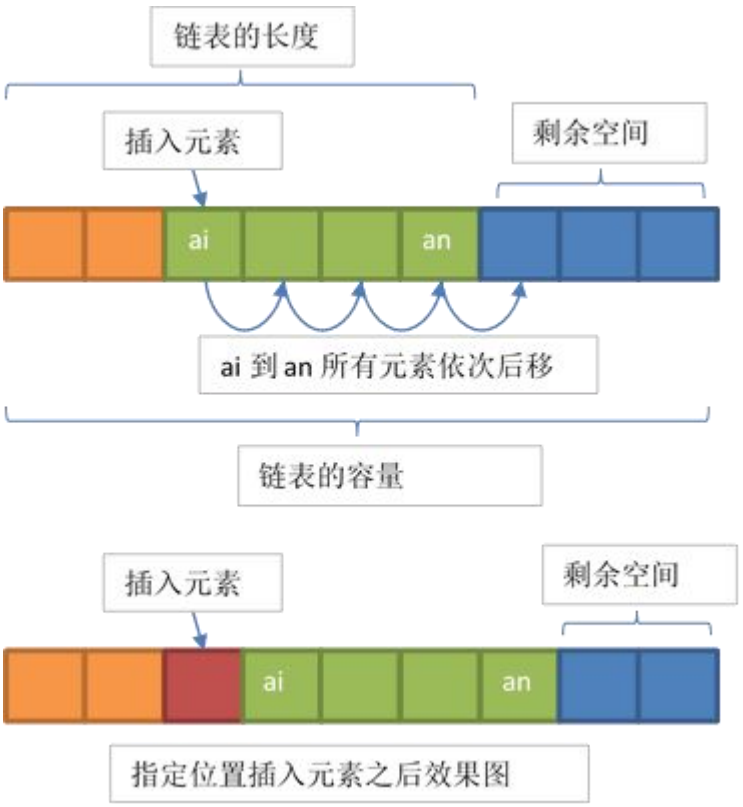
操作要点:

- 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

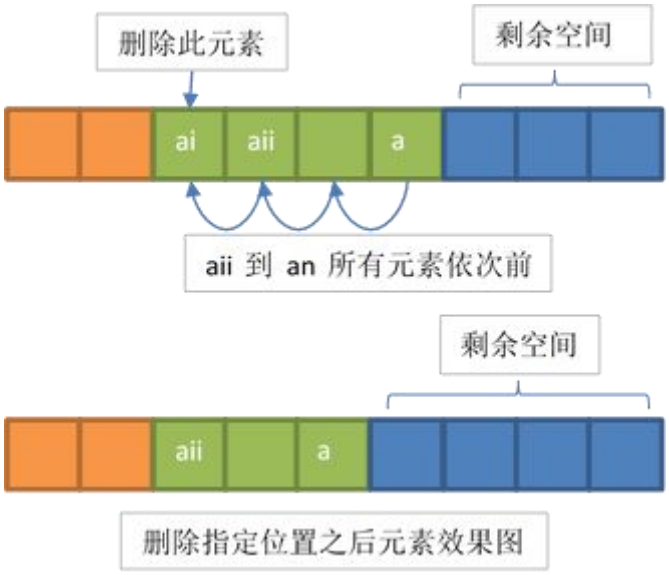
- 将删除位置后的元素分别向前移动一个位置
- 线性表长度减 1

链表顺序存储插入算法和删除算法:

- 元素的插入



- 元素的删除



注意: 链表的容量和链表的长度是两个不同的概念

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的时候，难以确定存储空间的容量。

2.2.4 线性表顺序存储案例

动态数组案例:

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define MYARRAY_TRUE 1
#define MYARRAY_FALSE 0

//定义结构体 保存数据首地址和数组长度
typedef struct MyArray{
    int* ArrayAddr; //数据首地址
    int capacity; //数据容量
    int length; //数据长度
}MyArray;


//初始化数组
MyArray* Init_MyArray(){

    MyArray* array = (MyArray*)malloc(sizeof(MyArray)); //给数组结构体分配内存
    array->ArrayAddr = (int*)malloc(sizeof(int)*20); //初始化分配内存 20 个 int 类型长度
    if (array->ArrayAddr == NULL){
        printf("申请内存失败!\n");
        return NULL;
    }
    memset(array->ArrayAddr, 0, sizeof(int)* 20); //初始化内存空间
    array->capacity = 20; //初始化数据容量
    array->length = 0; //初始化长度为 0

    return array; //成功返回数组指针
}

//判断数组是否为空 0:false 不为空  1:true 为空
int IsEmpty_MyArray(MyArray* array){

    if (array == NULL){
        return -1;
    }
```

```
    if (array->length == 0){
        return MYARRAY_TRUE;
    }
    return MYARRAY_FALSE;
}

//清空数组
int Clear_MyArray(MyArray* array){

    if (array == NULL){
        return -1;
    }
    if (array->ArrayAddr != NULL){
        memset(array->ArrayAddr, 0, array->capacity*sizeof(int)); //重新初始化内存空间
    }
    array->length = 0;

    return 0;
}

//获得指定位置元素
int GetElement_MyArray(MyArray* array,int pos){

    if (array == NULL){
        return -1;
    }
    if (pos > array->length-1){
        printf("数组访问越界!\n");
        return -2;
    }

    return array->ArrayAddr[pos];
}

//向指定位置插入元素
int InsertElement_MyArray(MyArray* array, int pos,int val){

    if (array == NULL){
        return -1;
    }
    //判断是否插入越界
    if (pos > array->length){
        printf("元素插入位置不合法!\n");
        return -2;
```

```
    }
    //判断当前是否还有容量，没有的话，重新申请内存空间
    if (array->length == array->capacity){
        //申请新的内存空间
        int* newSpaceAddr = (int*)malloc(array->capacity * 2 * sizeof(int));
        //将原内存空间数据拷贝到新的空间
        memcpy(newSpaceAddr,array->ArrayAddr,array->length*sizeof(int));
        //释放旧内存空间的内存
        if (NULL != array->ArrayAddr){
            free(array->ArrayAddr);
        }
        //让 ArrayAddr 重新指向新的内存空间
        array->ArrayAddr = newSpaceAddr;
        //更新容量
        array->capacity = array->capacity * 2;
    }
    //插入元素
    //第一步 移动元素
    for (int i = array->length-1; i >= pos; i--){
        array->ArrayAddr[i + 1] = array->ArrayAddr[i];
    }
    //第二部 在 Pos 位置插入元素
    array->ArrayAddr[pos] = val;
    array->length++;

    return 0;
}

//删除指定位置元素
int RemoevElement_MyArray(MyArray* array, int pos){

    if (array == NULL){
        return -1;
    }
    //移动元素
    for (int i = pos; i < array->length; i ++){
        array->ArrayAddr[i] = array->ArrayAddr[i + 1];
    }
    array->length--;

    return 0;
}

//获得链表长度
```

```
int GetLength_MyArray(MyArray* array){

    if (array == NULL){
        return -1;
    }
    return array->length;
}

//获得链表容量
int GetCapacity_MyArray(MyArray* array){

    if (array == NULL){
        return -1;
    }
    return array->capacity;
}

//打印数组内容
void PrintArray_MyArray(MyArray* array){

    for (int i = 0; i < array->length;i++){
        printf("%d ",array->ArrayAddr[i]);
    }
    printf("\n");
}

//销毁数组
int DestroyArray_MyArray(MyArray* array){

    if (array == NULL){
        return -1;
    }
    if (array->ArrayAddr != NULL){
        delete array->ArrayAddr;
        array->ArrayAddr = NULL;
    }
    array->capacity = 0;
    array->length = 0;

    //销毁数组
    delete array;

    return 0;
}
```

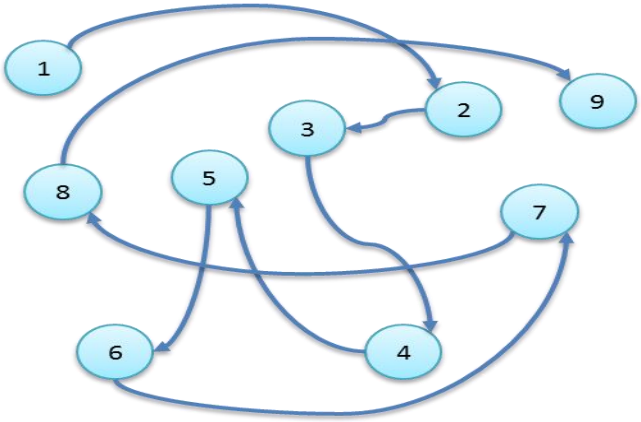
```
//测试 API
void test01(){

    MyArray* array = Init_MyArray(); //初始化数组
    //打印数组容量和长度
    printf("数组初始化长度:%d ,容
量:%d\n",GetLength_MyArray(array),GetCapacity_MyArray(array));
    //循环向数组中插入元素
    for (int i = 10; i <= 300;i++){
        InsertElement_MyArray(array,0,i);
    }
    printf("数组插入元素长度:%d ,容量:%d\n", GetLength_MyArray(array),
GetCapacity_MyArray(array));
    //打印数组
    PrintArray_MyArray(array);
    //删除位置 10 的元素
    RemoevElement_MyArray(array,10);
    //打印数组
    PrintArray_MyArray(array);
    printf("数组插入元素长度:%d ,容量:%d\n", GetLength_MyArray(array),
GetCapacity_MyArray(array));
    //销毁数组
    DestroyArray_MyArray(array);
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

2.3 线性表的链式存储

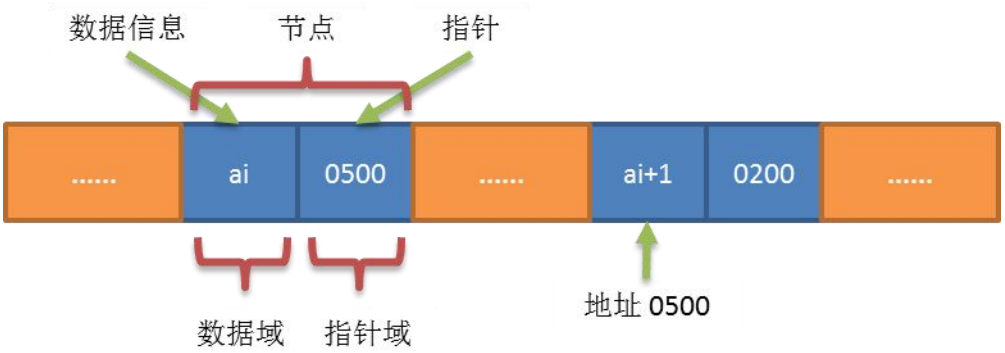
2.3.1 基本概念

- 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。



- 单链表
  - 线性表的链式存储结构中，每个节点中只包含一个指针域，这样的链表叫单链表。
  - 通过每个节点的指针域将线性表的数据元素按其逻辑次序链接在一起（如图）。

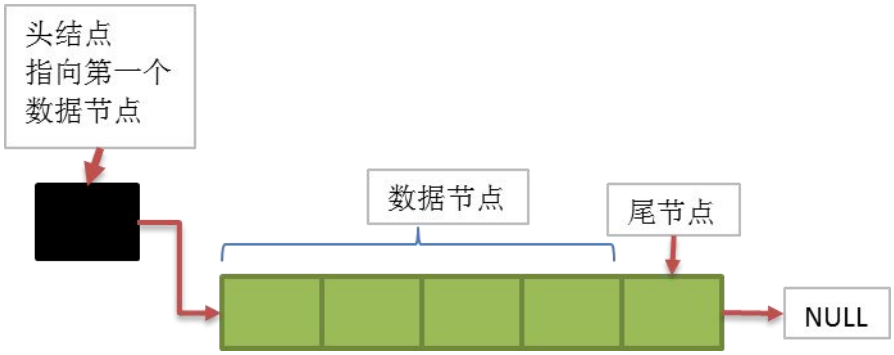


- 概念解释：
  - 表头结点

    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

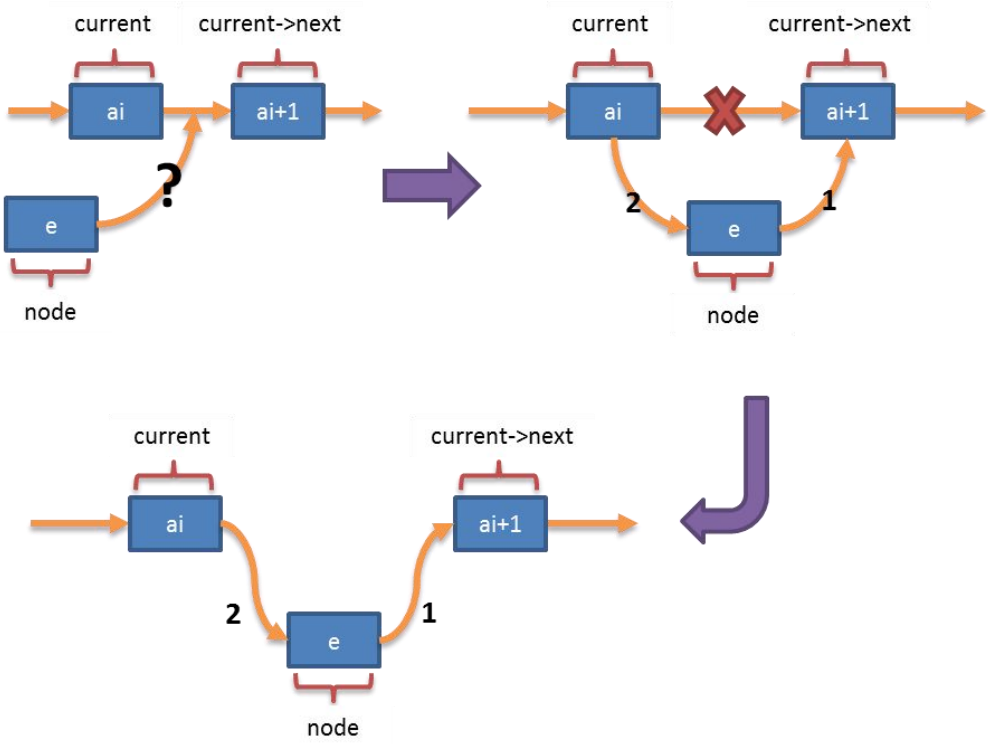
    链表中代表数据元素的结点，包含指向下一个数据元素的指针和数据元素的信息
  - 尾结点

    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。



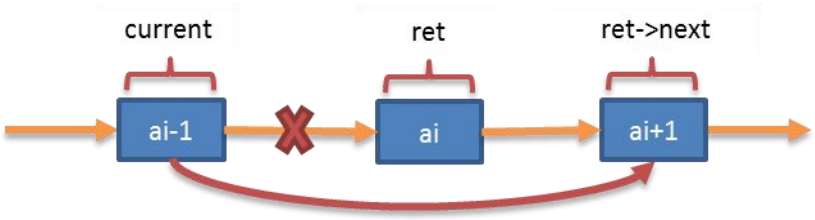
2.3.2 设计与实现

- 插入操作




```
node->next = current->next;

current->next = node;
```

- 删除操作




```
current->next = ret->next
```

2.3.3 优点和缺点

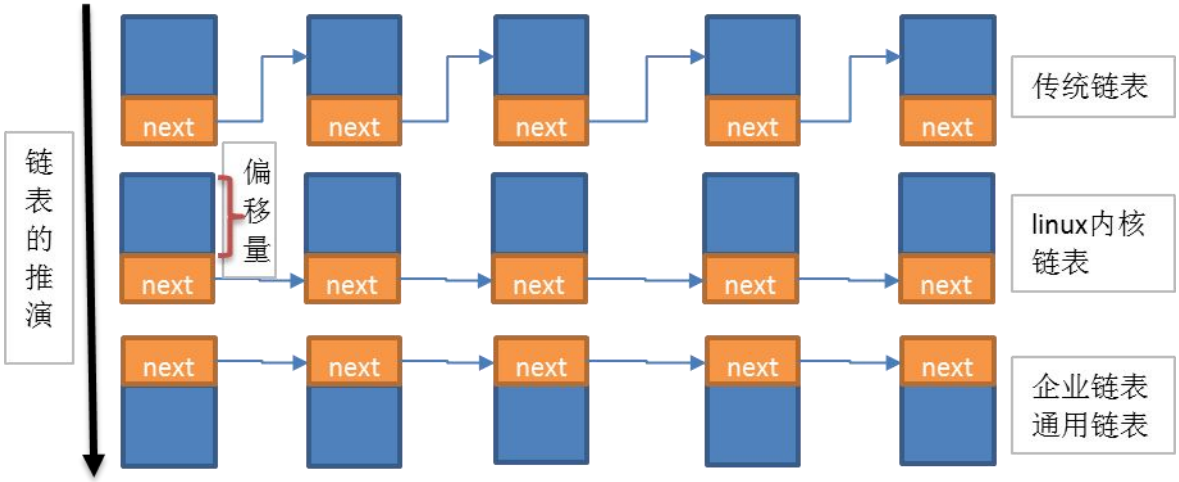
- **优点：**

■ 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

2.3.4 链表技术领域推演(能力提高)



2.3.5 企业链表/通用链表(单向链表)案例

- 头文件 LinkList.h

```
#ifndef LINKLIST_H
#define LINKLIST_H

#include<stdlib.h>
#include<stdio.h>
#define LINK_TRUE 1
#define LINK_FALSE 0

//链表结点
typedef struct _ListNode{
    struct _ListNode* next;
}ListNode;

//表头结点
typedef struct _LinkList{
    ListNode header; //头结点
    int length; //链表长度
}LinkList;
//打印回调函数
typedef void(*PrintLinkListData)(ListNode*);
//初始化链表
LinkList* InitLinkList();
//销毁链表
void DestroyLinkList(LinkList* list);
//链表中在指定位置插入结点
int InsertLinkList(LinkList* list, int pos, ListNode* node);
//删除指定位置结点
int DeleteLinkList(LinkList* list, int pos);
//链表是否为空
int IsEmptyLinkList(LinkList* list);
//返回链表长度
int GetLengthLinkList(LinkList* list);
//清空链表
void ClearLinkList(LinkList* list);
//打印链表结点
void PrintLinkList(LinkList* list, PrintLinkListData print);
//获得指定位置的结点
ListNode* GetNodeLinkList(LinkList* list, int pos);
#endif
```

- C 文件 LinkList.c

```c
#include"LinkList.h"


//初始化链表
LinkList* InitLinkList(){

    LinkList* list = (LinkList*)malloc(sizeof(LinkList));
    if (list == NULL){
        return NULL;
    }
    list->header.next = NULL;
    list->length = 0;

    return list;

}
//销毁链表
void DestroyLinkList(LinkList* list){
    if (NULL != list){
        list->header.next = NULL;
        list->length = 0;
        free(list);
    }
}
//链表中在指定位置插入结点
int InsertLinkList(LinkList* list, int pos, ListNode* node){

    ListNode* current = &(list->header);
    //如果 pos 位置超出链表长度，那么就插入到最后一个位置
    if (pos > list->length) pos = list->length;
    //找到插入的位置
    for (int i = 0; i < pos; i ++){
        current = current->next;
    }
    //新节点插入到 pos 位置
    node->next = current->next;
    current->next = node;

    list->length++;

    return 0;
```

```
}
//删除指定位置结点
int DeleteLinkList(LinkList* list, int pos){

    if (pos > list->length-1){
        printf("删除位置%d 不合法!\n",pos);
        return -1;
    }
    ListNode* current = &(list->header);
    //找到要删除的位置
    for (int i = 0; i < pos; i++){
        current = current->next;
    }
    //要删除的结点
    ListNode* delNode = current->next;
    //删除结点
    current->next = delNode->next;

    list->length--;

    return 0;
}

//链表是否为空
int IsEmptyLinkList(LinkList* list){
    if (list->length == 0){
        return LINK_TRUE;
    }
    return LINK_FALSE;
}
//返回链表长度
int GetLengthLinkList(LinkList* list){
    return list->length;
}
//清空链表
void ClearLinkList(LinkList* list){
    list->header.next = NULL;
    list->length = 0;
}
//打印链表结点
void PrintLinkList(LinkList* list, PrintLinkListData print){

    ListNode* current = (&(list->header))->next;
    while (current != NULL){
```

```
        print(current);
        current = current->next;
    }
    printf("----------------------------\n");

}

//获得指定位置的结点
ListNode* GetNodeLinkList(LinkList* list, int pos){

    if (pos > list->length-1){
        printf("位置%d 不合法!\n",pos);
        return NULL;
    }

    ListNode* current = &(list->header);
    for (int i = 0; i <= pos; i ++){
        current = current->next;
    }
    return current;
}
```

- 测试文件

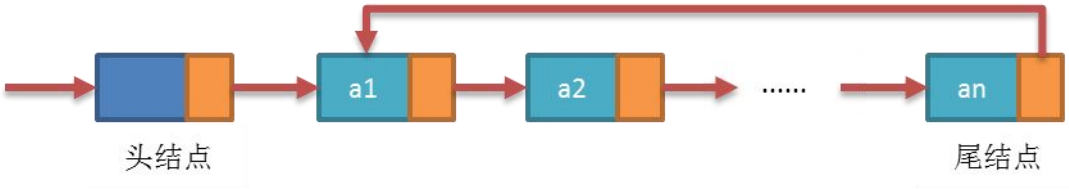
```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#include"LinkList.h"

typedef struct _Teacher{
    ListNode node;
    char name[128];
    int age;
}Teacher;

void PrintTeacher(ListNode* node){
    Teacher* teacher = (Teacher*)node;
    printf("Name:%s Age:%d\n", teacher->name, teacher->age);
}

//测试链表 API
void test01(){
```

```
    //初始化链表
    LinkList* list = InitLinkList();

    //创建数据
    Teacher t1, t2, t3;
    t1.age = 10;
    t2.age = 20;
    t3.age = 30;
    strcpy(t1.name, "aaa");
    strcpy(t2.name, "bbb");
    strcpy(t3.name, "ccc");

    //向链表中插入数据
    InsertLinkList(list, 0, (ListNode*)&t1);
    InsertLinkList(list, 0, (ListNode*)&t2);
    InsertLinkList(list, 0, (ListNode*)&t3);
    //打印链表内容
    PrintLinkList(list, PrintTeacher);
    //删除位置 1 的结点
    DeleteLinkList(list,2);
    //打印链表内容
    PrintLinkList(list, PrintTeacher);
    //获得位置 1 的结点
    Teacher* node = (Teacher*)GetNodeLinkList(list, 1);
    printf("位置 1 结点为:\n");
    printf("Name:%s Age:%d\n", node->name, node->age);

    //销毁链表
    DestroyLinkList(list);
}
int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

2.4 循环链表

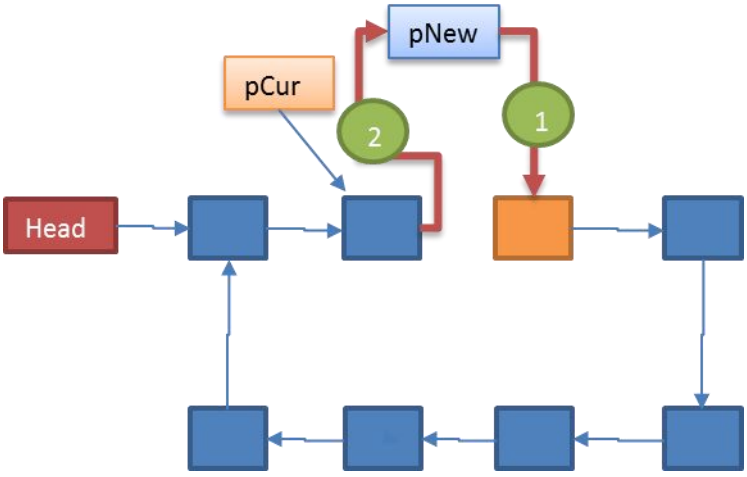
2.4.1 循环链表基本概念

链表中最后一个结点的指针域指向头结点，整个链表形成一个环。

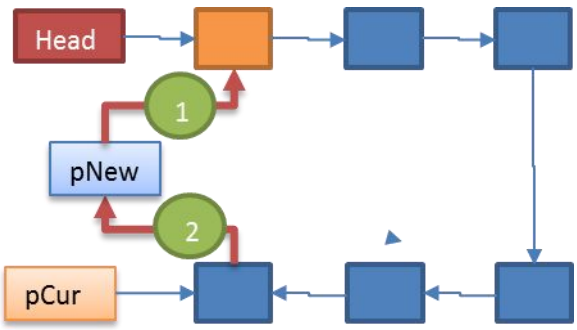


2.4.2 循环链表设计与实现

- **插入元素分析**
  - 普通插入元素（和单链表是一样的）



  - 尾插法（和单链表是一样的，单链表的写法支持尾插法



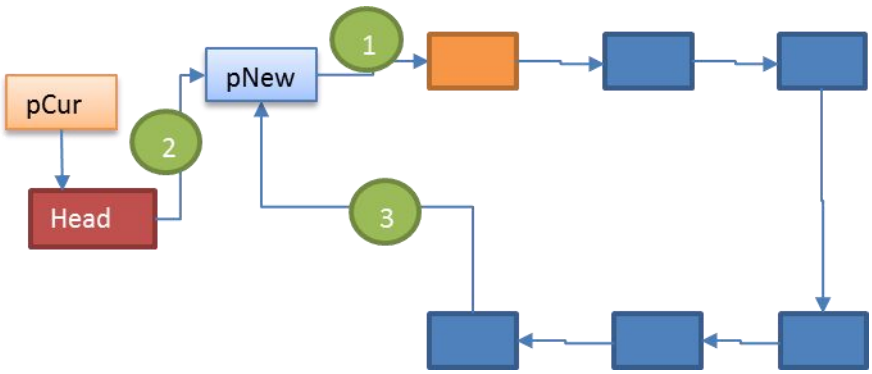
分析：最后一个结点的 next 指针指向新添加的结点，新结点的 next 指向第一个结点。

- 头插法

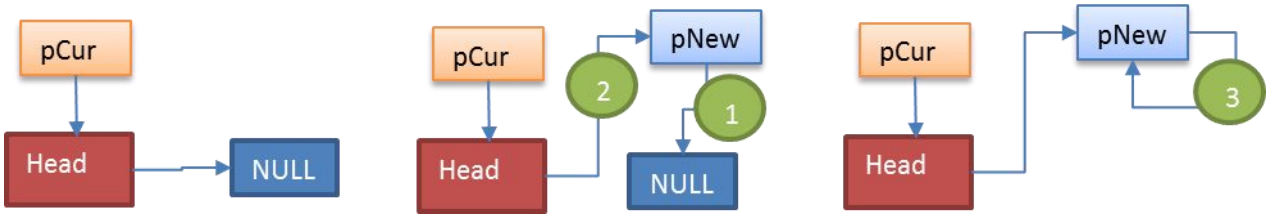
分析：

新节点指向当前的第一个结点

尾结点指向新节点



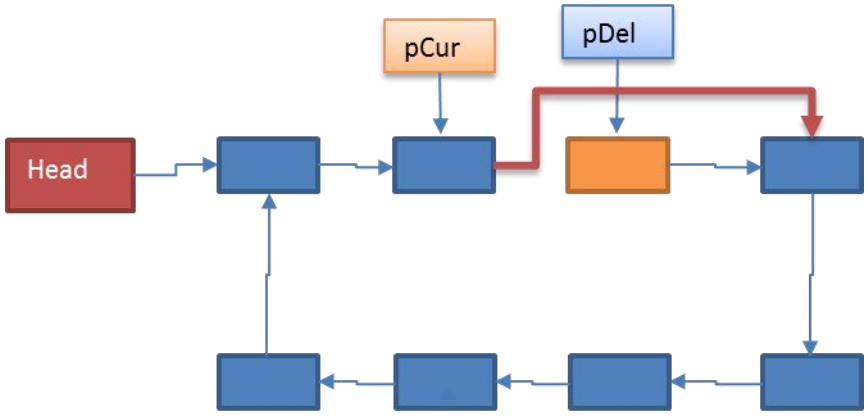
- 第一次插入结点



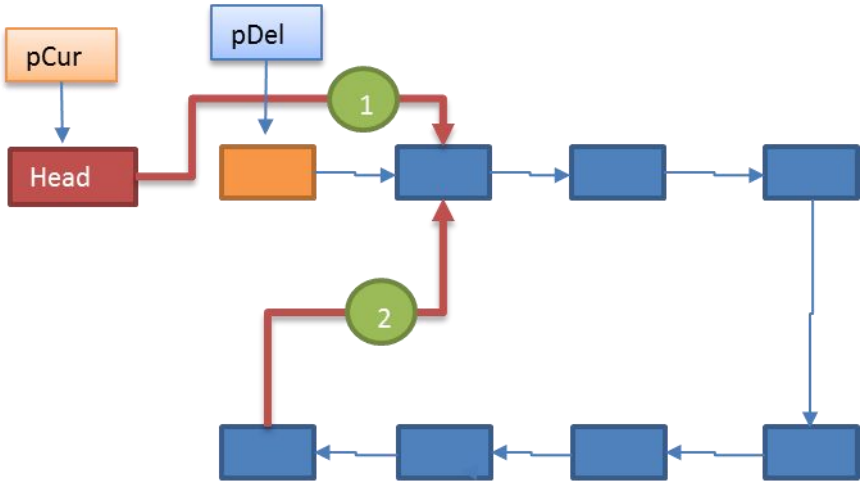
尾节点指针指向第一个数据节点（即自己指向自己）

➢ 删除结点分析

- 删除普通结点



- 删除头结点



更新 head 结点，并且尾结点重新连接新的头结点

2.4.3 循环链表案例

- 头文件 CircleLinkList.h

```
#ifndef CIRCLELINKLIST_H
#define CIRCLELINKLIST_H

#include<stdio.h>
#include<stdlib.h>

typedef void(*CIRCLEPRINT)(void*);
#define CIRCLE_TRUE 1
#define CIRCLE_FALSE 0

//循环链表结点
```

```
typedef struct _CircleLinkNode{

    void* data;
    struct _CircleLinkNode* next;

}CircleLinkNode;

//链表
typedef struct _DoubleLinkList{

    CircleLinkNode* head; //头结点
    CircleLinkNode* rear; //尾节点
    int length;

}CircleLinkList;



//循环链表初始化
CircleLinkNode* CircleLinkList_Init();
//循环链表插入数据
void CircleLinkList_Insert(CircleLinkList* dlist, int pos, void* data);
//循环链表删除数据
void CircleLinkList_Delete(CircleLinkList* dlist, int pos);
//销毁链表
void CircleLinkList_Destroy(CircleLinkList* dlist);
//打印链表
void CircleLinkList_Print(CircleLinkList* dlist, CIRCLEPRINT print);
//获得链表长度
int CircleLinkList_Length(CircleLinkList* dlist);
//判断链表是否为空
int CircleLinkList_Length(CircleLinkList* dlist);
//清空链表
void CircleLinkList_Clear(CircleLinkList* dlist);

#endif
```

- C 文件 CircleLinkList.c

```
#include"CircleLinkList.h"

//循环链表初始化
CircleLinkNode* CircleLinkList_Init(){

    //链表结构体分配内存
```

```
    CircleLinkList* dlist = (CircleLinkList*)malloc(sizeof(CircleLinkList));
    if (dlist == NULL){
        return NULL;
    }
    //创建头结点 创建头结点 是为了空链表和非空链表处理一致
    CircleLinkNode* head = (CircleLinkNode*)malloc(sizeof(CircleLinkNode));
    if (head == NULL){
        if (dlist != NULL){
            free(dlist);
        }
        return NULL;
    }
    head->data = NULL;
    head->next = NULL;
    //初始化链表
    dlist->head = head;
    dlist->rear = head;
    dlist->length = 0;

    return dlist;

}
//循环链表插入数据
void CircleLinkList_Insert(CircleLinkList* dlist, int pos, void* data){

    if (dlist == NULL || data == NULL || pos < 0){
        return;
    }

    if (pos > dlist->length){
        pos = dlist->length;
    }

    //辅助指针
    CircleLinkNode* pCurrent = dlist->head;
    for (int i = 0; i < pos;i++){
        pCurrent = pCurrent->next;
    }

    //创建新节点
    CircleLinkNode* newnode = (CircleLinkNode*)malloc(sizeof(CircleLinkNode));
    if (newnode == NULL){
        return;
    }
```

```
    newnode->data = data;
    newnode->next = NULL;

    //将新结点插入到链表中
    newnode->next = pCurrent->next;
    pCurrent->next = newnode;

    //判断是否在尾部插入结点
    if (pCurrent == dlist->rear){
        dlist->rear = newnode;
        dlist->rear->next = dlist->head;
    }

    dlist->length++;
}
//循环链表删除数据
void CircleLinkList_Delete(CircleLinkList* dlist, int pos){

    if (dlist == NULL || pos < 0){
        return;
    }

    if (pos >= dlist->length){
        return;
    }

    //赋值指针
    CircleLinkNode* pCurrent = dlist->head;
    for (int i = 0; i < pos;i++){
        pCurrent = pCurrent->next;
    }
    //缓存被删除结点
    CircleLinkNode* pDel = pCurrent->next;
    //重新建立前后结点关系
    pCurrent->next = pDel->next;
    //释放被删除结点内存
    free(pDel);

    //链表结点数量减 1
    dlist->length--;

}
//打印链表
void CircleLinkList_Print(CircleLinkList* dlist, CIRCLEPRINT print){
```

```
    CircleLinkNode* pCurrent = dlist->head->next;
    while (pCurrent != dlist->head){
        print(pCurrent->data);
        pCurrent = pCurrent->next;
    }
}
//销毁链表
void CircleLinkList_Destroy(CircleLinkList* dlist){

    CircleLinkNode* pCurrent = dlist->head;
    while (pCurrent!=dlist->head){
        //缓存被删除结点的下一个结点
        CircleLinkNode* pNext = pCurrent->next;
        free(pCurrent);
        pCurrent = pNext;
    }

    free(dlist);
}

//获得链表长度
int CircleLinkList_Length(CircleLinkList* dlist){
    if (dlist == NULL){
        return -1;
    }
    return dlist->length;
}
//判断链表是否为空
int CircleLinkList_Length(CircleLinkList* dlist){
    if (dlist->length == 0){
        return CIRCLE_TRUE;
    }
    return CIRCLE_FALSE;
}
//清空链表
void CircleLinkList_Clear(CircleLinkList* dlist){

    CircleLinkNode* pCurrent = dlist->head->next;
    while (pCurrent != dlist->head){
        CircleLinkNode* pNext = pCurrent->next;
        free(pCurrent);
        pCurrent = pNext;
    }
```

```
    dlist->head->next = NULL;
    dlist->rear = dlist->head;
    dlist->length = 0;
}
```

- 测试文件

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"CircleLinkList.h"

typedef struct _Teacher{
    int age;
    double salary;
}Teacher;

//打印回调函数
void ShowLinkList(void* data){
    Teacher* t = (Teacher*)data;
    printf("age:%d salay:%f \n", t->age, t->salary);
}

void test01(){

    //创建循环链表
    CircleLinkList* clist =  CircleLinkList_Init();
    //创建测试数据
    Teacher t1 = { 10, 200.0 };
    Teacher t2 = { 20, 300.0 };
    Teacher t3 = { 30, 400.0 };
    //向循环链表中插入数据
    CircleLinkList_Insert(clist, 0, &t1);
    CircleLinkList_Insert(clist, 0, &t2);
    CircleLinkList_Insert(clist, 0, &t3);
    CircleLinkList_Insert(clist, 0, &t2);
    CircleLinkList_Insert(clist, 0, &t3);
    //打印链表
    CircleLinkList_Print(clist, ShowLinkList);
    //删除链表结点
    CircleLinkList_Delete(clist,3);
    //打印链表
```

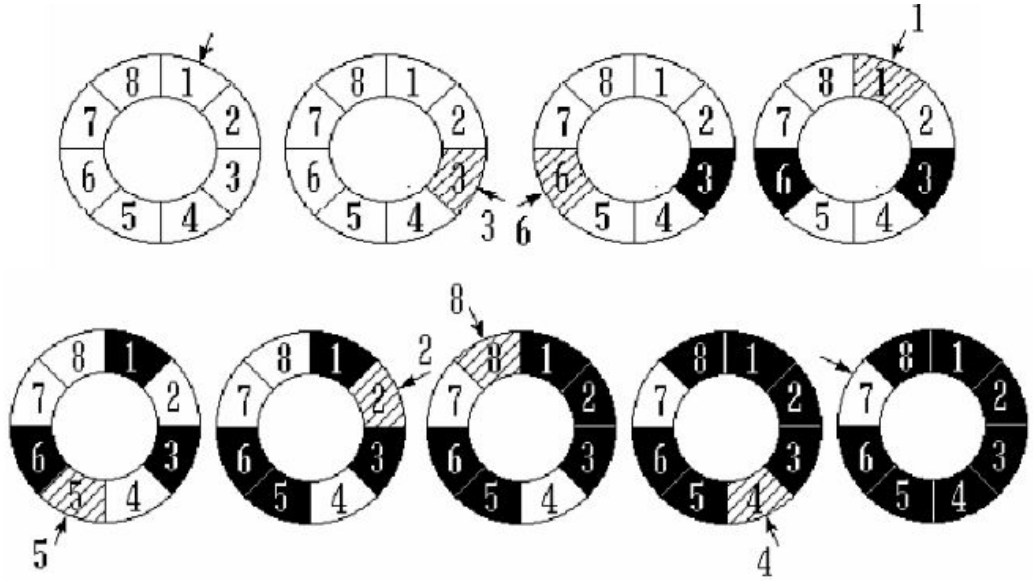
```
    printf("------------------\n");
    CircleLinkList_Print(clist, ShowLinkList);
    //销毁链表
    CircleLinkList_Destroy(clist);
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

2.4.4 循环链表应用(约瑟夫问题)

- 约瑟夫问题-循环链表典型应用

  例题：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。

  假设： m = 8，n=3



实现代码(通过我们自己编写的循环链表库):

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include"CircleLinkList.h"

typedef struct _Teacher{
    int age;
    double salary;
}Teacher;

//打印回调函数
void ShowLinkList(void* data){

    Teacher* t = (Teacher*)data;
    printf("age:%d salay:%f \n", t->age, t->salary);

}

void ShowData(void* data){
    printf("%d ",(int)data);
}

//约瑟夫问题
void test02(){

    //m 表示 8 个人 n 表示第几个出列
    int m = 8;
    int n = 3;
    //创建循环链表
    CircleLinkList* clist = CircleLinkList_Init();
    for (int i = 1; i <= m; i ++){
        CircleLinkList_Insert(clist,i,i);
    }
    //打印循环链表
    CircleLinkList_Print(clist, ShowData);
    printf("\n");
    CircleLinkNode* pCurrent = clist->head;
    int index = 0;
    while (pCurrent){
        if (index == n){
            index = 1;
```

```
            printf("%d ",(int)pCurrent->data);
            //缓存删除结点的下一个结点
            CircleLinkNode* pNext = pCurrent->next;
            if (pNext == clist->head){
                pNext = pNext->next;
            }
            CircleLinkList_DeleteByValue(clist, pCurrent->data);
            pCurrent = pNext;
            if (CircleLinkList_Length(clist) == 1){
                printf("%d ", (int)pCurrent->data);
                break;
            }
        }

        pCurrent = pCurrent->next;
        if (pCurrent == clist->head){
            pCurrent = pCurrent->next;
        }
        index++;
    }

    //销毁链表
    CircleLinkList_Destroy(clist);
}

int main(){

    //test01();
    test02();

    system("pause");
    return EXIT_SUCCESS;
}
```

2.4.5 优缺点分析

- 优点
  - 功能增强了（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（成也萧何败萧何）

2.5 双向链表

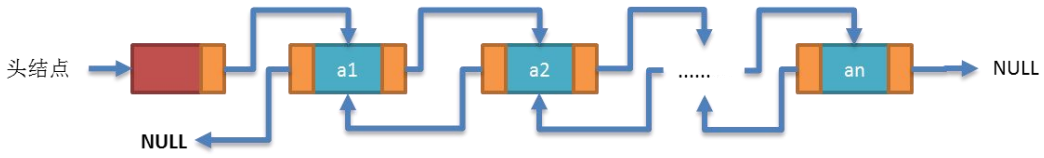
2.5.1 双向链表基本概念

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！（如图）

● 双向链表的定义

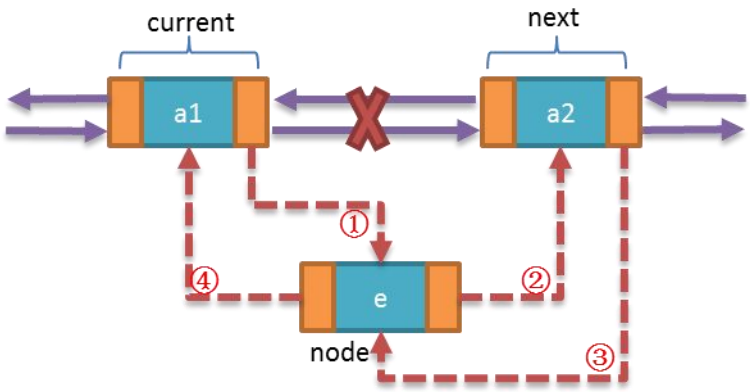
在单链表的结点中增加一个指向其前驱的 pre 指针。



2.5.2 双向链表设计与实现

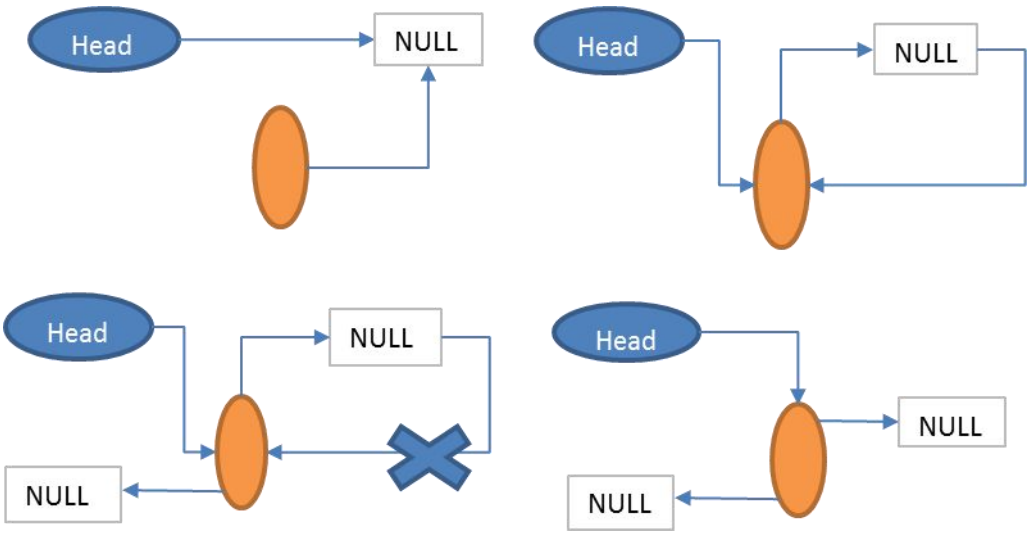
➢ 插入操作

- 在普通位置插入节点




```
current->next = node;
node->next = next;
next->pre = node;
node->pre = current;
```

■ **在空链表中插入节点**

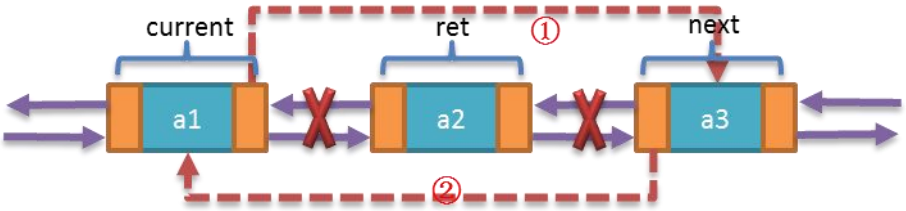


插入操作异常处理

◆ 在空链表中插入新的元素

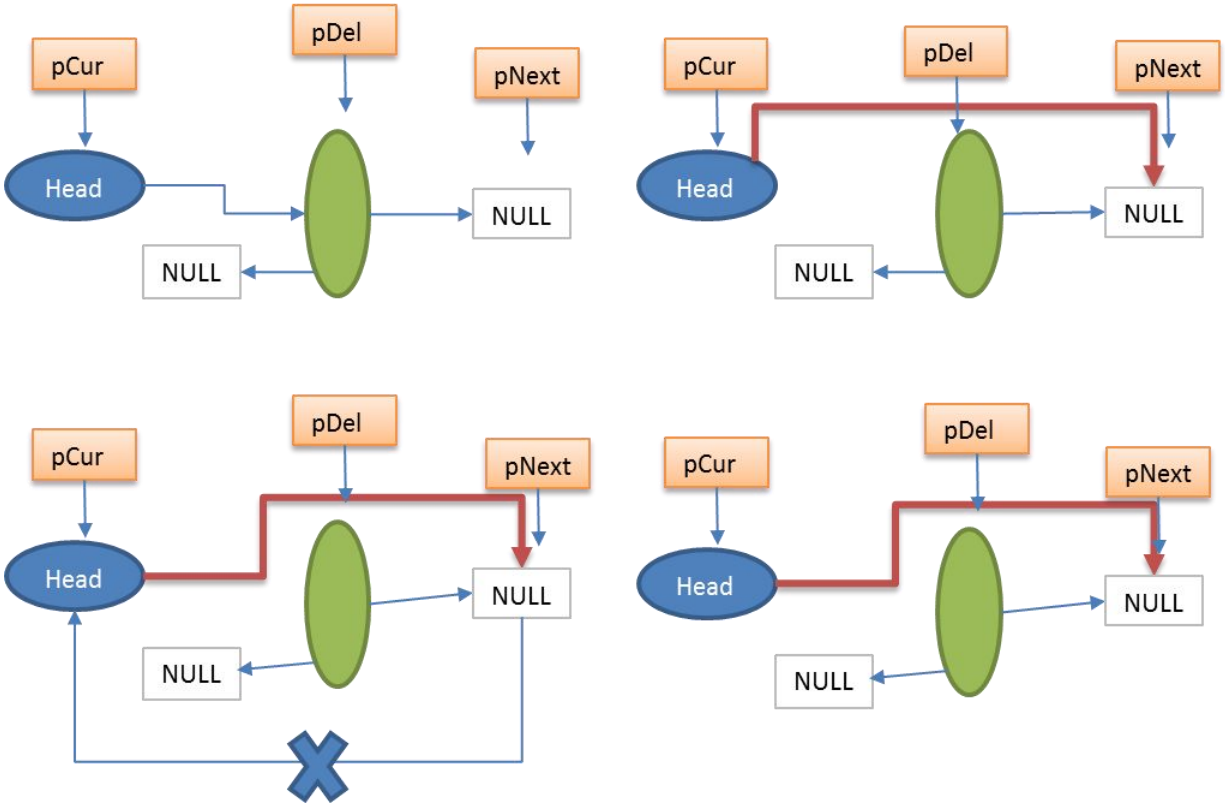
- 空节点（NULL）没有前驱指针；
- 新节点的前驱指针，需要指向 NULL；

- 删除操作



```
current->next = next;
next->pre = current;
```

- 删除操作异常处理：当双向链表仅有一个节点的时候



如果辅助指针变量 pNext == NULL，那么 pNext 是没有前驱操作的。空指针既没有前驱也没有后继。

2.5.3 双向链表案例

- DoubleLinkList.h

```
#ifndef DOUBLELINKLIST_H
#define DOUBLELINKLIST_H

#define DOUBLE_FALSE 0
#define DOUBLE_TRUE 1

#include<stdio.h>
#include<stdlib.h>

//链表结点
typedef struct _DOUBLELISTNODE{
```

```
    void* data;
    struct _DOUBLELISTNODE* next;
    struct _DOUBLELISTNODE* prev;
}DoubleListNode;

//头结点
typedef struct _DOUBLELINKLIST{
    DoubleListNode* head; //首结点
    DoubleListNode* rear; //尾结点
    int length; //链表长度
}DoubleLinkList;

//打印回调函数
typedef void(*PrintDoubleLinkListNode)(DoubleListNode*);

//双向链表初始化
DoubleLinkList* InitDoubleLinkList();
//销毁双向链表
void DestroyDoubleLinkList(DoubleLinkList* dlist);
//向链表中插入结点
int InsertDoubleLinkList(DoubleLinkList* dlist, int pos, void* data);
//打印链表
void PrintDoubleListNode(DoubleLinkList* dlist, PrintDoubleLinkListNode* print,int
IsReverse);
//获得链表长度
int GetLengthDoubleLinkList(DoubleLinkList* dlist);
//判断链表是否为空
int IsEmptyDoubleLinkList(DoubleLinkList* dlist);

#endif
```

- DoubleLinkList.c

```
#include"DoubleLinkList.h"


//双向链表初始化
DoubleLinkList* InitDoubleLinkList(){

    DoubleLinkList* dlist = (DoubleLinkList*)malloc(sizeof(DoubleLinkList));
    if (dlist == NULL){
        return NULL;
    }
```

```
    dlist->head = NULL;
    dlist->rear = NULL;
    dlist->length = 0;

    return dlist;
}
//销毁双向链表
void DestroyDoubleLinkList(DoubleLinkList* dlist){

}
//向链表中插入结点
int InsertDoubleLinkList(DoubleLinkList* dlist, int pos, void* data){

    //创建新的结点
    DoubleListNode* newnode = (DoubleListNode*)malloc(sizeof(DoubleListNode));
    newnode->data = data;
    newnode->next = NULL;
    newnode->prev = NULL;

    //第一次插入
    if (dlist->head == NULL && dlist->rear == NULL){
        dlist->head = newnode;
        dlist->rear = newnode;

        //初始化 head 结点前驱和后继结点
        dlist->head->next = NULL;
        dlist->head->prev = NULL;
        //初始化 rear 结点的前驱和后继结点
        dlist->rear->next = NULL;
        dlist->rear->prev = NULL;

        dlist->length++;
        return 0;
    }

    //头插法
    if (pos == 0){
        newnode->prev = NULL;
        newnode->next = dlist->head;

        dlist->head->prev = newnode;
        dlist->head = newnode;

        dlist->length++;
```

```
        return 0;
    }

    //尾差法
    if (pos >= dlist->length){

        dlist->rear->next = newnode;
        newnode->prev = dlist->rear;
        newnode->next = NULL;
        dlist->rear = newnode;
        dlist->length++;
        return 0;
    }

    //其他插入
    DoubleListNode* pPrev = dlist->head;
    DoubleListNode* pCurrent = pPrev->next;
    for (int i = 1; i < pos;i++){
        pPrev = pCurrent;
        pCurrent = pPrev->next;
    }

    newnode->next = pCurrent;
    pCurrent->prev = newnode;

    pPrev->next = newnode;
    newnode->prev = pPrev;

    dlist->length++;

    return 0;
}
//打印链表
void PrintDoubleListNode(DoubleLinkList* dlist, PrintDoubleLinkListNode print, int
IsReverse){

    DoubleListNode* pCurrent = NULL;
    //true 表示 逆序遍历
    if (IsReverse == DOUBLE_TRUE){
        pCurrent = dlist->rear;
        while (pCurrent != NULL){
            print(pCurrent);
            pCurrent = pCurrent->prev;
        }
```

```
    }
    else{
        pCurrent = dlist->head;
        while (pCurrent != NULL){
            print(pCurrent);
            pCurrent = pCurrent->next;
        }
    }
    printf("----------------------------\n");
}
//获得链表长度
int GetLengthDoubleLinkList(DoubleLinkList* dlist){
    return dlist->length;
}
//判断链表是否为空
int IsEmptyDoubleLinkList(DoubleLinkList* dlist){
    return DOUBLE_TRUE;
}
```

- 测试文件

```
#include"DoubleLinkList.h"


typedef struct _TEAHCER{
    char name[64];
    int age;
}Teacher;

void PrintNode(DoubleListNode* node){
    Teacher* teacher = (Teacher*)(node->data);
    printf("Name:%s Age:%d\n",teacher->name,teacher->age);
}

//测试双向链表函数
void test01(){

    //创建双向链表
    DoubleLinkList* dlist = InitDoubleLinkList();
    //创建数据
    Teacher t1, t2, t3;
    t1.age = 10;
    t2.age = 20;
    t3.age = 30;
```

```
    strcpy(t1.name, "aaa");
    strcpy(t2.name, "bbb");
    strcpy(t3.name, "ccc");
    //插入数据
    InsertDoubleLinkList(dlist, 0, &t1);
    InsertDoubleLinkList(dlist, 10, &t2);
    InsertDoubleLinkList(dlist, 0, &t3);
    //打印数据
    printf("----------正向遍历----------\n");
    PrintDoubleListNode(dlist, PrintNode, DOUBLE_FALSE);
    printf("----------逆序遍历----------\n");
    PrintDoubleListNode(dlist, PrintNode,DOUBLE_TRUE);

}

int main(){
    test01();
    system("pause");
    return EXIT_SUCCESS;
}
```

2.5.4 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.受限线性表

3.1 栈(Stack)

3.1.1 栈的基本概念

- 概念：

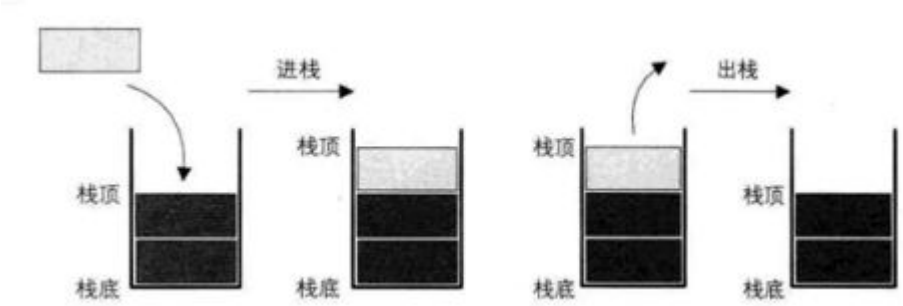
  首先它是一个线性表，也就是说，栈元素具有线性关系，即前驱后继关系。只不过**它是一种特殊的线性表**而已。定义中说是在线性表的表尾进行插入和删除操作，这里表尾是指栈顶，而不是栈底。

- 特性

  它的特殊之处在于限制了这个线性表的插入和删除的位置，它始终只在栈顶进行。这也就使得：栈底是固定的，最先进栈的只能在栈底。

- 操作

  - 栈的插入操作，叫做进栈，也成压栈。类似子弹入弹夹（如下图所示）
  - 栈的删除操作，叫做出栈，也有的叫做弹栈，退栈。如同弹夹中的子弹出夹（如下图所示）

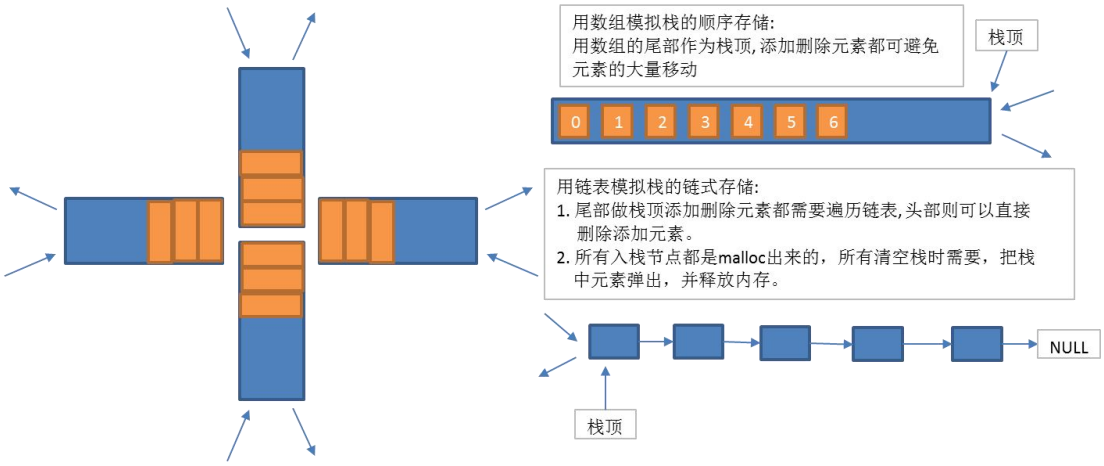


- 创建栈

- 销毁栈
- 清空栈
- 进栈
- 出栈
- 获取栈顶元素
- 获取栈的大小

栈的抽象数据类型

```
ADT 栈（stack）
Data
  通线性表。元素具有相同的类型，相邻的元素具有前驱和后继关系。
Operation
  // 初始化，建立一个空栈 S
  InitStack(*S);
  // 若栈存在，则销毁它
  DestroyStack(*S);
  // 将栈清空
  ClearStack(*S);
  // 若栈为空则返回 true，否则返回 false
  StackEmpty(S);
  // 若栈存在且非空，用 e 返回 S 的栈顶元素
  GetTop(S，*e);
  // 若栈 S 存在，插入新元素 e 到栈 S 中并成为其栈顶元素
  Push(*S，e);
  // 删除栈 S 中的栈顶元素，并用 e 返回其值
  Pop（*S，*e）；
  // 返回栈 S 的元素个数
  StackLength(S);
endADT
```



3.1.2 栈的顺序存储

- **基本概念**

  栈的顺序存储结构简称顺序栈，它是运算受限制的顺序表。顺序栈的存储结构是：**利用一组地址连续的的存储单元依次存放自栈底到栈顶的数据元素**，同时附设指针 top 只是栈顶元素在顺序表中的位置。

- **设计与实现**

  因为栈是一种特殊的线性表，所以栈的顺序存储可以通过顺序线性表来实现。

- SqStack.h

```
#ifndef SQSTACK_H
#define SQSTACK_H

#include<stdio.h>
#include<stdlib.h>

#define STACK_FALSE 0
#define STACK_TRUE 1

#define MAX 256
//栈结构体
typedef struct _SQSTACK{
    void* data[MAX];
    int length;
}SqStack;
```

```
//初始化顺序栈
SqStack* InitSqStack();
//销毁栈
void DestroySqStack(SqStack* stack);
//弹出栈顶元素
void PopSqStack(SqStack* stack);
//获得栈顶元素
void* TopSqStack(SqStack* stack);
//获得栈中元素个数
int GetLengthSqStack(SqStack* stack);
//栈中是否为空
int IsEmptySqStack(SqStack* stack);
//向栈中添加元素
int PushSqStack(SqStack* stack,void* data);

#endif
```

- SqStack.c

```
#include"SqStack.h"

//初始化顺序栈
SqStack* InitSqStack(){
    SqStack* stack = (SqStack*)malloc(sizeof(SqStack));
    if (stack == NULL){
        return NULL;
    }
    //初始化
    for (int i = 0; i < MAX; i++){
        stack->data[i] = 0;
    }
    stack->length = 0;

    return stack;
}
//销毁栈
void DestroySqStack(SqStack* stack){
    if (NULL != stack){
        free(stack);
        stack = NULL;
    }
}
```

```
//弹出栈顶元素
void PopSqStack(SqStack* stack){
    stack->data[stack->length - 1] = 0;
    stack->length--;
}
//获得栈顶元素
void* TopSqStack(SqStack* stack){
    return stack->data[stack->length - 1];
}
//获得栈中元素个数
int GetLengthSqStack(SqStack* stack){
    return stack->length;
}
//栈中是否为空
int IsEmptySqStack(SqStack* stack){
    if (stack->length == 0){
        return STACK_TRUE;
    }
    return STACK_FALSE;
}
//向栈中添加元素
int PushSqStack(SqStack* stack, void* data){

    if (stack->length > MAX){
        return -1;
    }
    stack->data[stack->length] = data;
    stack->length++;
}
```

- 测试文件

```
#include"SqStack.h"

typedef struct _TEAHCER{
    char name[64];
    int age;
}Teacher;

void test01(){

    //创建空栈
    SqStack* stack = InitSqStack();
    //创建数据
```

```
    Teacher t1, t2, t3;
    t1.age = 10;
    t2.age = 20;
    t3.age = 30;
    strcpy(t1.name, "aaa");
    strcpy(t2.name, "bbb");
    strcpy(t3.name, "ccc");
    //向栈中添加元素
    PushSqStack(stack, &t1);
    PushSqStack(stack, &t3);
    PushSqStack(stack, &t2);
    //打印栈中元素个数
    printf("栈元素个数:%d\n", GetLengthSqStack(stack));
    //遍历栈中元素
    while (GetLengthSqStack(stack) > 0){
        Teacher* teacher =  (Teacher*)TopSqStack(stack);
        printf("Name:%s Age:%d\n",teacher->name,teacher->age);
        PopSqStack(stack);
    }
    //打印栈中元素个数
    printf("栈元素个数:%d\n",GetLengthSqStack(stack));
    //销毁栈
    DestroySqStack(stack);

}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

3.1.3 栈的链式存储

- **基本概念**

  栈的链式存储结构简称链栈。

  思考如下问题：

栈只是栈顶来做插入和删除操作，栈顶放在链表的头部还是尾部呢？

由于单链表有头指针，而栈顶指针也是必须的，那干嘛不让他俩合二为一呢，所以比较好的办法就是把栈顶放在单链表的头部。另外都已经有了栈顶在头部了，单链表中比较常用的头结点也就失去了意义，通常对于链栈来说，是不需要头结点的。

- **设计与实现**

链栈是一种特殊的线性表，链栈可以通过链式线性表来实现。

- LinkStack.h

```
#ifndef LINKSTACK_H
#define LINKSTACK_H

#include<stdio.h>
#include<stdlib.h>

#define STACK_FALSE 0
#define STACK_TRUE 1

//链栈结点
typedef struct _LINKNODE{
    void* data;
    struct _LINKNODE* next;
}LinkNode;
//头结点
typedef struct _LINKSTACK{
    LinkNode* head;
    int length;
}LinkStack;

//初始化链栈
LinkStack* InitLinkStack();
//销毁链栈
void DestroyLinkStack(LinkStack* lstack);
//获得栈顶元素
void* TopLinkStack(LinkStack* lstack);
//弹出栈顶元素
void PopLinkStack(LinkStack* lstack);
//获得栈元素个数
```

```
int GetLengthLinkStack(LinkStack* lstack);
//向栈中加入元素
int PushLinkStack(LinkStack* lstack,void* data);
//判断栈是否为空
int IsEmptyLinkStack(LinkStack* lstack);

#endif
```

- LinkStack.c

```
#include"LinkStack.h"

//初始化链栈
LinkStack* InitLinkStack(){

    LinkStack* lstack = (LinkStack*)malloc(sizeof(LinkStack));
    if (lstack == NULL){
        return NULL;
    }
    //初始化
    lstack->head = NULL;
    lstack->length = 0;

    return lstack;
}
//销毁链栈
void DestroyLinkStack(LinkStack* lstack){

    while (!IsEmptyLinkStack(lstack)){
        PopLinkStack(lstack);
    }
    free(lstack);
}
//获得栈顶元素
void* TopLinkStack(LinkStack* lstack){
    return lstack->head->data;
}
//弹出栈顶元素
void PopLinkStack(LinkStack* lstack){
    if (lstack->length == 0){
        return;
    }
    //缓存要删除的结点
    LinkNode* pDel =  lstack->head;
    //将被删除结点的下一个结点作为 head 结点
```

```
    lstack->head = pDel->next;
    //释放被删除结点内存
    free(pDel);
    //结点数量减 1
    lstack->length--;
}
//获得栈元素个数
int GetLengthLinkStack(LinkStack* lstack){
    return lstack->length;
}
//向栈中加入元素
int PushLinkStack(LinkStack* lstack, void* data){

    //创建结点
    LinkNode* newnode = (LinkNode*)malloc(sizeof(LinkNode));
    newnode->data = data;
    newnode->next = NULL;

    //是否第一次插入元素
    if (lstack->head == NULL){
        lstack->head = newnode;
        lstack->length++;
        return 0;
    }

    //其他插入情况
    newnode->next = lstack->head;
    lstack->head = newnode;
    lstack->length++;

    return 0;
}
//判断栈是否为空
int IsEmptyLinkStack(LinkStack* lstack){
    if (lstack->length == 0){
        return STACK_TRUE;
    }
    return STACK_FALSE;
}
```

- 测试文件

```
#include"LinkStack.h"
```

```
typedef struct _TEAHCER{
    char name[64];
    int age;
}Teacher;

void test01(){

    //创建链栈
    LinkStack* lstack = InitLinkStack();
    //创建数据
    Teacher t1, t2, t3;
    t1.age = 10;
    t2.age = 20;
    t3.age = 30;
    strcpy(t1.name, "aaa");
    strcpy(t2.name, "bbb");
    strcpy(t3.name, "ccc");
    //向栈中添加元素
    PushLinkStack(lstack, &t1);
    PushLinkStack(lstack, &t3);
    PushLinkStack(lstack, &t2);
    //打印链栈数据
    while (!IsEmptyLinkStack(lstack)){
        Teacher* teacher = (Teacher*)TopLinkStack(lstack);
        printf("Name:%s Age:%d\n",teacher->name,teacher->age);
        PopLinkStack(lstack);
    }
    //销毁链表
    DestroyLinkStack(lstack);
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

3.1.4 栈的应用(案例)

➢ 案例 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 ,那么如何实现编译器中的符号成对检测？如下字符串:

#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
- 总结
  - **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**

案例代码:

```
#include"LinkStack.h"

//案例一 就近匹配
void test02(){

    char* str = "#include <stdio.h> int main() { int a[4][4]; int (*p)[4]; p = a[0]; return 0;}";
    //初始化栈
```

```
    LinkStack* lstack = InitLinkStack();
    //匹配括号
    char* pCurrent = str;
    while (*pCurrent != '\0'){
        if (*pCurrent == '('){
            PushLinkStack(lstack, pCurrent);
        }
        else if (*pCurrent == ')'){
            char* p = (char*)TopLinkStack(lstack);
            if (*p == '('){
                PopLinkStack(lstack);
            }
        }
        pCurrent++;
    }
    if (GetLengthLinkStack(lstack) > 0){
        printf("匹配失败!\n");
    }
    //销毁栈
    DestroyLinkStack(lstack);
}

int main(){

    test02();

    system("pause");
    return EXIT_SUCCESS;
}
```

➢ 案例 2：中缀表达式和后缀表达式

- 后缀表达式（由波兰科学家在 20 世纪 50 年代提出）
  - **将运算符放在数字后面 ===》 符合计算机运算**
  - 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：此符号进栈

        （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

- 中缀转后缀伪代码 priority

```
transform(exp)
{
创建栈 S;
i= 0;
while(exp[i] !=  '\0' )
    {
    if(exp[i] 为数字)
    {
        Output（exp[i]）;
    }
    else if(exp[i] 为符号)
    {
        while(exp[i]优先级 <= 栈顶符号优先级)
        {
            output（栈顶符号）;
            Pop（S）;
        }
        Push(S, exp[i]);
```

```
    }
    else if(exp[i] 为左括号)
    {
        Push(S, exp[i]);
    }
    else if(exp[i] 为右括号)
    {
        while(栈顶符号不为左括号)
        {
            output（栈顶符号）;
            Pop（S）;
        }
        从 S 中弹出左括号;
    }
    else
    {
        报错，停止循环;
    }
    i++;
  }
  while(size(S) > 0 && exp[i] ==  '\0' )
  {
    output(栈顶符号);
    Pop（S）;
  }
}
```

- 动手练习

将我们喜欢的读的中缀表达式转换成计算机喜欢的后缀表达式

中缀表达式: 8 + (3 – 1) * 5

后缀表达式: 8 3 1 – 5 * +

➢ 案例 3：计算机如何基于后缀表达式计算

- 思考

计算机是如何基于后缀表达式计算的？

例如：8 3 1 – 5 * +

- 计算规则

  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

- 代码实现（伪代码） express

```
compute(exp)
{
创建栈;
int i = 0;
while( exp[i] != ‘\0’ )
{
    if（exp[i]为数字）
    {
        Push（S, exp[i]）;
    }
    else if(exp[i]为符号)
    {
           1. 从栈顶弹出右操作数;
           2. 从栈中弹出左操作数;
           3. 根据符号进行运算;
           4. Push（stack， 结果）;
    }
    else
    {
        报错，停止循环;
    }
    i++;
}
if( Size(s) == 1 && exp[i] == ‘\0’ )
```

```
{
    栈中唯一的数字为运算结果；
}
返回结果；
}
```

3.2 队列(Queue)

3.2.1 队列基本概念

队列是一种特殊的受限制的线性表。

队列（queue）是只允许在一端进行插入操作，而在另一端进行删除操作的线性表。

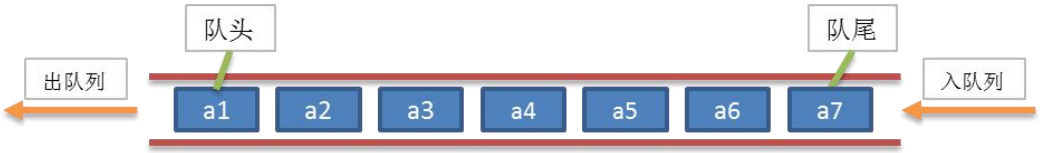
队列是一种先进先出的（First In First Out）的线性表，简称 FIFO。允许插入的一端为队尾，允许删除的一端为队头。队列不允许在中间部位进行操作！假设队列是 q=（a1，a2，……，an），那么 a1 就是队头元素，而 an 是队尾元素。这样我们就可以删除时，总是从 a1 开始，而插入时，总是在队列最后。这也比较符合我们通常生活中的习惯，排在第一个的优先出列，最后来的当然排在队伍最后。如下图：

3.2.2 队列常用操作

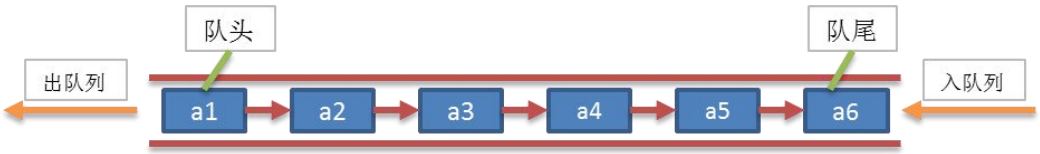
- 创建队列
- 销毁队列
- 清空队列
- 进队列
- 出队列

- 获取队头元素
- 获取队列的长度

```
ADT 队列（Queue）
Data
  通线性表。元素具有相同的类型，相邻元素具有前驱后继关系。
Operation
  // 初始化操作，建立一个空队列 Q
  InitQueue（*Q）;
  // 若队列 Q 存储，则销毁它。
  DestroyQueue(*Q);
  // 将队列 Q 清空
  ClearQueue(*Q);
  // 若队列为空则返回 true，否则返回 false
  QueueEmpty(Q);
  // 若队列 Q 存在且非空，用 e 返回队列 Q 的队头元素
  GetHead(Q, *e);
  // 若队列 Q 存在，插入新元素 e 到队列 Q 中并成为队尾元素。
  EnQueue(*Q, e);
  // 删除队列 Q 中的队头元素，并用 e 返回其值
  DeQueue(*Q, *e);
  // 返回队列 Q 的元素个数
  QueueLength(Q);
endADT
```



可以使用数组来模拟队列的顺序存储:
因为队列需要对两端进行操作，所以数组的哪端作为队头，哪端作为队尾都是可以的。



用单向链表来模拟队列的链式存储:
与数组一样，哪端作为队头，哪端作为队尾都是可以的。
注意事项:
1. 需要将队列节点 ==> 链表节点 ==> 入链表库
2. 需要为新节点分配存储空间。
3. 节点出队列，需要释放内存空间。

3.2.3 队列的顺序存储

- 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。

- SqQueue.h

```
#ifndef SQQUEUE_H
#define SQQUEUE_H

#include<stdlib.h>
#include<stdio.h>

#define MAX 256
#define QUEUE_FALSE 0
#define QUEUE_TRUE 1

//队列头结点
typedef struct _SQQUEUE{
    void* data[MAX];
    int length;
}SqQueue;

//初始化空队列
```

```
SqQueue* InitSqQueue();
//销毁队列
void DestroySqQueue(SqQueue* queue);
//向队列中插入元素
int PushSqQueue(SqQueue* queue, void* data);
//获得队列长度
int GetLengthSqQueue(SqQueue* queue);
//获得对头元素
void* FrontSqQueue(SqQueue* queue);
//从对头弹出元素
void PopSqQueue(SqQueue* queue);
//判断队列是否为空
int IsEmptySqQueue(SqQueue* queue);

#endif
```

- SqQueue.c

```
#include"SqQueue.h"


//初始化空队列
SqQueue* InitSqQueue(){

    SqQueue* queue = (SqQueue*)malloc(sizeof(SqQueue));
    if (queue == NULL){
        return NULL;
    }
    //初始化
    for (int i = 0; i < MAX;i++){
        queue->data[i] = 0;
    }
    queue->length = 0;

    return queue;
}
//销毁队列
void DestroySqQueue(SqQueue* queue){
    if (queue != NULL){
        free(queue);
    }
}
//向队列中插入元素
```

```
int PushSqQueue(SqQueue* queue, void* data){
    if (queue->length >= MAX){
        return -1;
    }
    queue->data[queue->length] = data;
    queue->length++;
}
//获得队列长度
int GetLengthSqQueue(SqQueue* queue){
    return queue->length;
}
//获得对头元素
void* FrontSqQueue(SqQueue* queue){
    return queue->data[0];
}
//从对头弹出元素
void PopSqQueue(SqQueue* queue){
    for (int i = 0; i < queue->length;i++){
        queue->data[i] = queue->data[i + 1];
    }
    queue->length--;
}
//判断队列是否为空
int IsEmptySqQueue(SqQueue* queue){
    if (queue->length == 0){
        return QUEUE_TRUE;
    }
    return QUEUE_FALSE;
}
```

- 测试文件

```
#include"SqQueue.h"

typedef struct _TEAHCER{
    char name[64];
    int age;
}Teacher;

void test01(){

    //创建队列
    SqQueue* queue = InitSqQueue();
```

```
    //创建数据
    Teacher t1, t2, t3;
    t1.age = 10;
    t2.age = 20;
    t3.age = 30;
    strcpy(t1.name, "aaa");
    strcpy(t2.name, "bbb");
    strcpy(t3.name, "ccc");
    //队列中插入数据
    PushSqQueue(queue, &t3);
    PushSqQueue(queue, &t1);
    PushSqQueue(queue, &t2);
    //打印队列数据
    while (!IsEmptySqQueue(queue)){
        Teacher* teacher = (Teacher*)FrontSqQueue(queue);
        printf("Name:%s Age:%d\n",teacher->name,teacher->age);
        PopSqQueue(queue);
    }
    //销毁队列
    DestroySqQueue(queue);
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

3.2.4 队列的链式存储

- **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。

- **LinkQueue.h**

```
#ifndef LINKQUEUE_H
#define LINKQUEUE_H
```

```c
#define QUEUE_FALSE 0
#define QUEUE_TRUE 1

#include<stdlib.h>
#include<stdio.h>

//链表队列结点
typedef struct _LINKNODE{
    void* data;
    struct _LINKNODE* next;
}LinkNode;

//链式队列头结点
typedef struct _LINKQUEUE{
    LinkNode* head;
    int length;
}LinkQueue;

//初始化链式队列
LinkQueue* InitLinkQueue();
//销毁链式队列
void DestroyLinkQueue(LinkQueue* lqueue);
//向队尾插入结点
int PushLinkQueue(LinkQueue* lqueue,void* data);
//返回对头数据
void* FrontLinkQueue(LinkQueue* lqueue);
//弹出对头结点
void PopLinkQueue(LinkQueue* lqueue);
//返回队列长度
int GetLengthLinkQueue(LinkQueue* lqueue);
//判断队列是否为空
int IsEmptyLinkQueue(LinkQueue* lqueue);

#endif
```

- **LinkQueue.c**

```c
#include"LinkQueue.h"

//初始化链式队列
LinkQueue* InitLinkQueue(){

    LinkQueue* lqueue = (LinkQueue*)malloc(sizeof(LinkQueue));
    if (lqueue == NULL){
```

```
        return NULL;
    }

    lqueue->head = NULL;
    lqueue->length = 0;

    return lqueue;
}
//销毁链式队列
void DestroyLinkQueue(LinkQueue* lqueue){

    while (!IsEmptyLinkQueue(lqueue)){
        PopLinkQueue(lqueue);
    }
    free(lqueue);
}
//向队尾插入结点
int PushLinkQueue(LinkQueue* lqueue, void* data){

    //创建新结点
    LinkNode* newnode = (LinkNode*)malloc(sizeof(LinkNode));
    newnode->data = data;
    newnode->next = NULL;

    //第一次插入
    if (lqueue->head == NULL){
        lqueue->head = newnode;
        lqueue->length++;
        return 0;
    }

    //其他插入
    newnode->next = lqueue->head;
    lqueue->head = newnode;
    lqueue->length++;

    return 0;
}
//返回对头数据
void* FrontLinkQueue(LinkQueue* lqueue){
    LinkNode* pCurrent = lqueue->head;
    while (pCurrent->next != NULL){
        pCurrent = pCurrent->next;
    }
```

```
    return pCurrent->data;
}
//弹出对头结点
void PopLinkQueue(LinkQueue* lqueue){

    //如果只有一个结点
    if (lqueue->length == 1){
        free(lqueue->head);
        lqueue->head = NULL;
        lqueue->length--;
        return;
    }
    //多于一个结点
    LinkNode* pPrev = lqueue->head;
    LinkNode* pCurrent = pPrev->next;
    while (pCurrent->next != NULL){
        pPrev = pCurrent;
        pCurrent = pPrev->next;
    }
    free(pCurrent);
    pPrev->next = NULL;
    lqueue->length--;
}
//返回队列长度
int GetLengthLinkQueue(LinkQueue* lqueue){
    return lqueue->length;
}
//判断队列是否为空
int IsEmptyLinkQueue(LinkQueue* lqueue){
    if (lqueue->length == 0){
        return QUEUE_TRUE;
    }
    return QUEUE_FALSE;
}
```

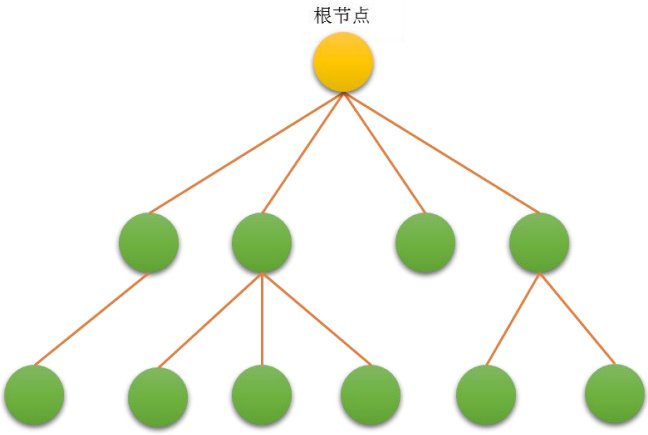
- 测试文件

```
#include"LinkQueue.h"


typedef struct _TEAHCER{
    char name[64];
    int age;
}Teacher;
```

```
void test01(){

    //创建链式队列
    LinkQueue* queue = InitLinkQueue();
    //创建数据
    Teacher t1, t2, t3;
    t1.age = 10;
    t2.age = 20;
    t3.age = 30;
    strcpy(t1.name, "aaa");
    strcpy(t2.name, "bbb");
    strcpy(t3.name, "ccc");
    //链式队列插入数据
    PushLinkQueue(queue, &t1);
    PushLinkQueue(queue, &t3);
    PushLinkQueue(queue, &t2);
    //打印链式队列
    printf("链式队列长度:%d\n",GetLengthLinkQueue(queue));
    while (!IsEmptyLinkQueue(queue)){
        Teacher* teacher = (Teacher*)FrontLinkQueue(queue);
        printf("Name:%s Age:%d\n",teacher->name,teacher->age);
        PopLinkQueue(queue);
    }
    printf("链式队列长度:%d\n", GetLengthLinkQueue(queue));
    //销毁链式队列
    DestroyLinkQueue(queue);

}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

4.树和二叉树

4.1 树的基本概念

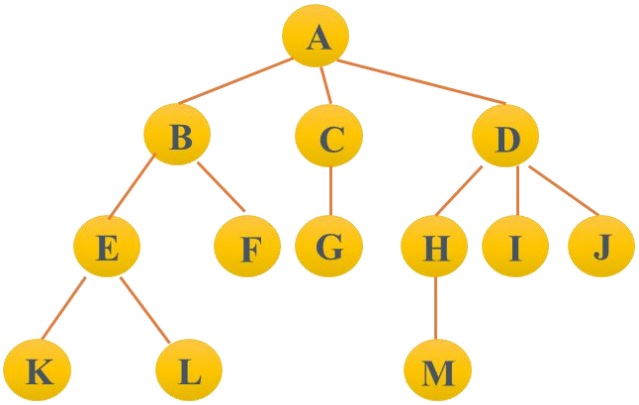


- 树的定义：

  由一个或多个(n≥0)结点组成的有限集合 T ,有且仅有一个结点称为根(root) ,当 n>1 时，其余的结点分为 **m(m≥0)个互不相交的有限集合** T1,T2，...，Tm。每个集合本身又是棵树，被称作这个根的子树 。

- 树的结构特点
  - 非线性结构，有一个直接前驱，但可能有多个直接后继（ 1:n ）
  - 树的定义具有递归性，树中还有树。
  - 树可以为空，即节点个数为 0。
- 若干术语
  - 根 → 即根结点(没有前驱)
  - 叶子 → 即终端结点(没有后继)
  - 森林 → 指 m 棵不相交的树的集合(例如删除 A 后的子树个数)

- 有序树 → 结点各子树从左至右有序，不能互换（左为第一）
- 无序树 → 结点各子树可互换位置。
- 双亲 → 即上层的那个结点(直接前驱) parent
- 孩子 → 即下层结点的子树 (直接后继) child
- 兄弟 → 同一双亲下的同层结点（孩子之间互称兄弟）sibling
- 堂兄弟 → 即双亲位于同一层的结点（但并非同一双亲）cousin
- 祖先 → 即从根到该结点所经分支的所有结点
- 子孙 → 即该结点下层子树中的任一结点



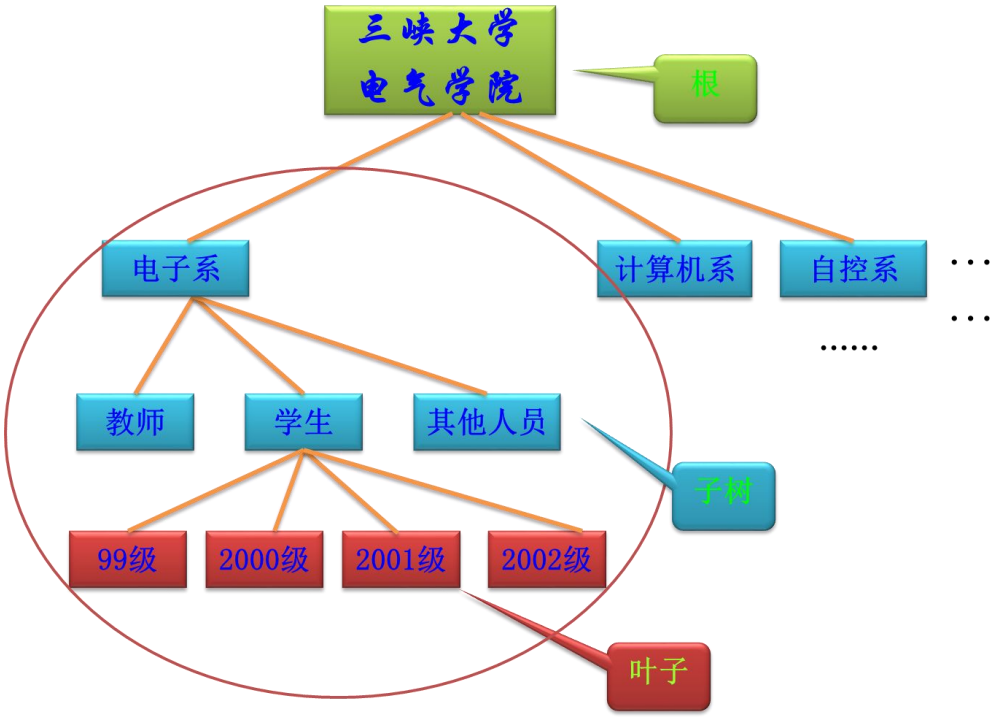
- 结点 → 即树的数据元素
- 结点的度 → 结点挂接的子树数（有几个直接后继就是几度）
- 结点的层次 → 从根到该结点的层数（根结点算第一层）
- 终端结点 → 即度为 0 的结点，即叶子
- 分支结点 → 除树根以外的结点（也称为内部结点）
- 树的度 → 所有结点度中的最大值（Max{各结点的度}）
- 树的深度(或高度) → 指所有结点中最大的层数（Max{各结点的层次}）

上图中的结点数 = 13，树的度 = 3，树的深度 = 4

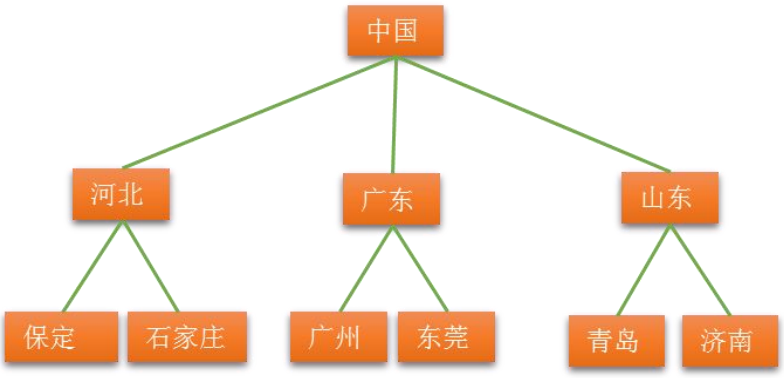
4.2 树的表示法

4.2.1 图形表示法

事物之间的逻辑关系可以通过数的形式很直观的表示出来，如下图：



4.2.2 广义表表示法

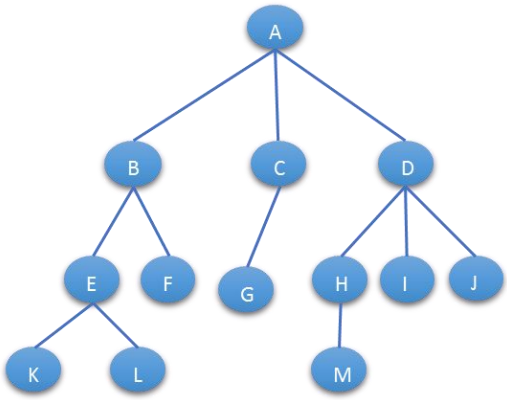


用广义表表示法表示上图：

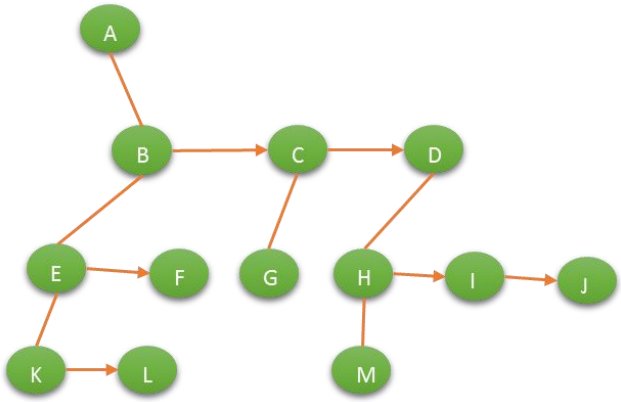
中国（河北（保定，石家庄），广东（广州，东莞），山东（青岛，济南））

根作为由子树森林组成的表的名字写在表的左边。

4.2.3 左孩子右兄弟表示法



左孩子右兄弟表示法可以将一颗多叉树转化为一颗二叉树：



节点的结构：

节点有两个指针域，其中一个指针指向子节点，另一个指针指向其兄弟节点。

4.3 树的结构

4.3.1 树的逻辑结构

树的逻辑结构特点：

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。

4.3.2 树的存储结构

树的存储仍然有两种方式：

- **顺序存储**

  可规定为：从上至下、从左至右将树的结点依次存入内存。

  重大缺陷：复原**困难**（不能唯一复原就没有实用价值）。

- **链式存储**

  可用多重链表：一个前趋指针，n 个后继指针。

  细节问题：树中结点的结构类型样式该如何设计？

  即应该设计成“等长”还是“不等长”？

  缺点：等长结构太浪费（每个结点的度不一定相同）；

  不等长结构太复杂（要定义好多种结构类型）。

  以上两种存储方式都存在重大缺陷，应该如何解决呢？

计算机实现各种不同进制的运算是通过先研究最简单、最有规律的二进制运算规律，然后设法把各种不同进制的运算转化二进制运算。树的存储也可以通过先研究最简单、最有规律的树，然后设法把一般的树转化为这种简单的树，这种树就是**二叉树**。

4.4 二叉树概念

4.4.1 二叉树基本概念

- **定义：**

  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 。

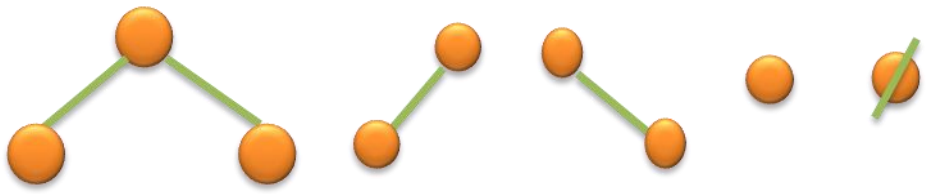
- **逻辑结构：**

  一对二（1：2）

- **基本特征:**

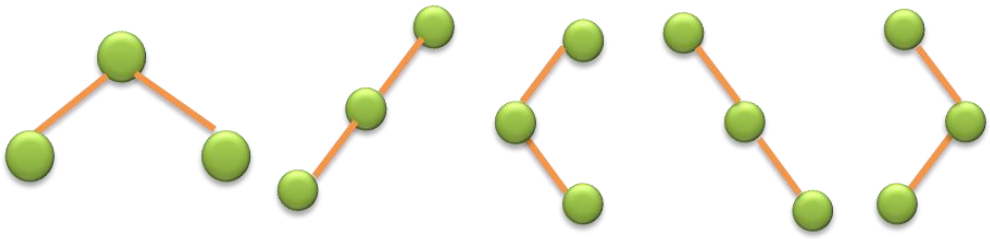
  - 每个结点最多只有两棵子树（**不存在度大于 2 的结点**）；
  - 左子树和右子树次序不能颠倒（**有序树**）。

- **基本形态:**



- **问题：**

  具有 3 个结点的二叉树可能有几种不同形态？普通树呢？



  二叉树可以画出五种形态，但是普通的数只能有两种形态。

- **二叉树性质**

  - 性质 1: 在二叉树的第 i 层上至多有 2^{i-1} 个结点（i>0）

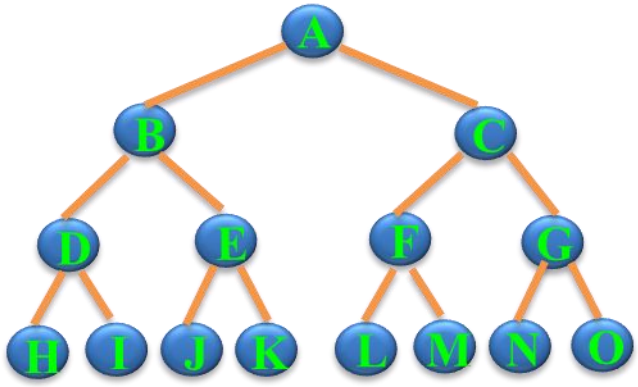
- 性质 2: 深度为 k 的二叉树至多有 2^k-1 个结点（k>0）
- 性质 3: 对于任何一棵二叉树，若度为 2 的结点数有 n2 个，则叶子数（n0）必定为 n2＋1 （即 n0=n2+1）

概念解释：

✧ **满二叉树**

一棵深度为 k 且有 2^k -1 个结点的二叉树。

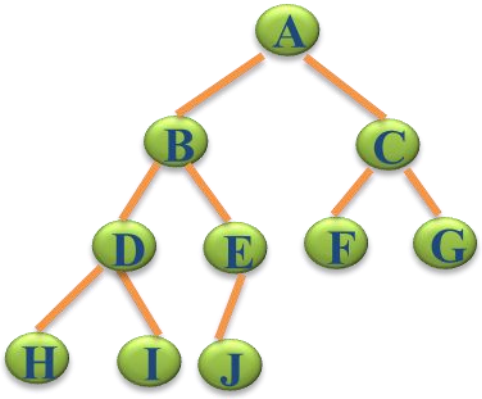
特点：每层都“充满”了结点



深度为4的满二叉树

✧ **完全二叉树**

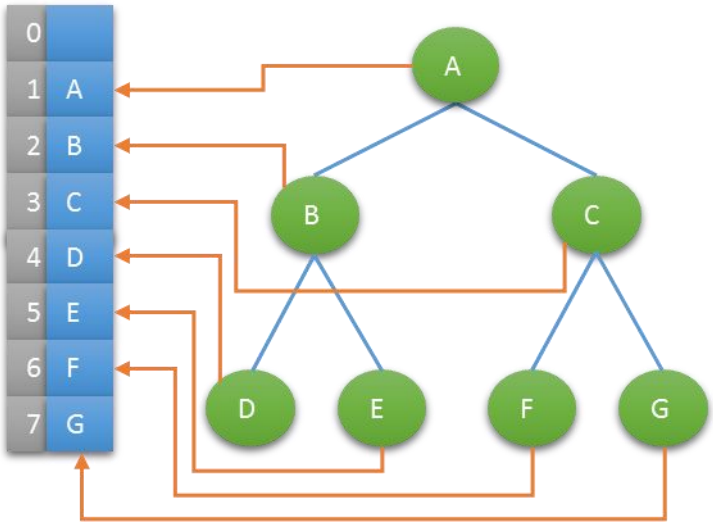
除最后一层外，每一层上的节点数均达到最大值；在最后一层上只缺少右边的若干结点。



深度为4的完全二叉树

理解：k-1 层与满二叉树完全相同，第 k 层结点尽力靠左

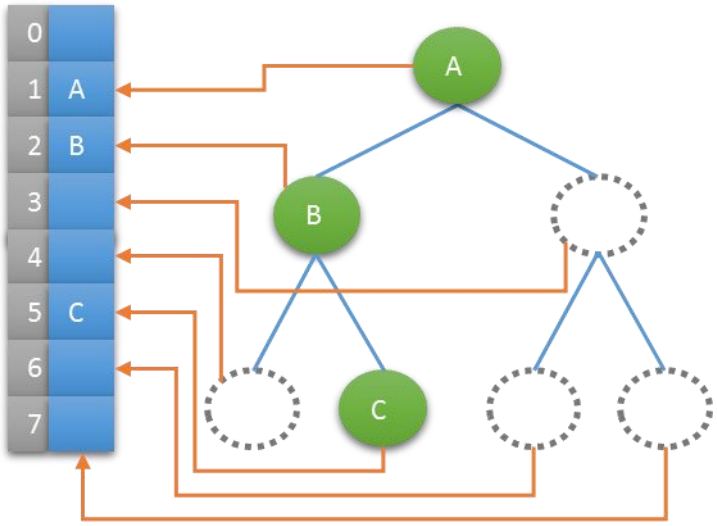
- 性质 4: 具有 n 个结点的完全二叉树的深度必为 $\lfloor \log_2 n \rfloor + 1$
- **性质 5: 对完全二叉树，若从上至下、从左至右编号，则编号为 i 的结点，其左孩子编号必为 2i，其右孩子编号必为 2i + 1；其双亲的编号必为 i/2（i = 1 时为根，除外）**



使用此性质可以使用完全二叉树实现树的顺序存储。

如果不是完全二叉树咋整???

------ 将其转换成完全二叉树即可



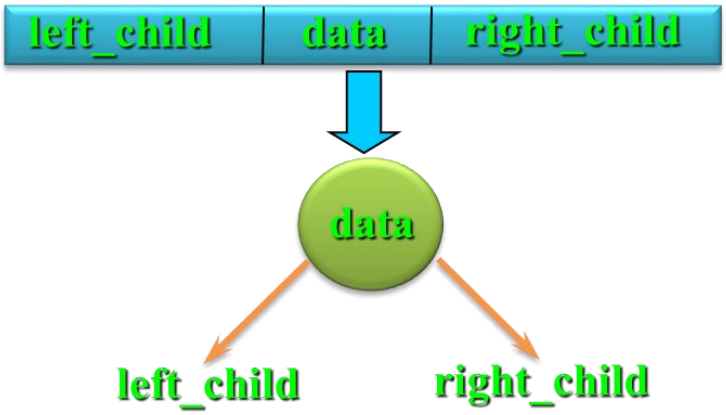
缺点：①浪费空间；②插入、删除不便

4.4.2 二叉树的表示

- **二叉链表示法**

  一般从根结点开始存储。相应地，访问树中结点时也只能从根开始。

  - **存储结构**



  - **结点数据类型定义：**

```
typedef struct BiTNode
{
    int     data;
    struct BiTNode *lchild, *rchild;
}BiTNode, *BiTree;
```

- **三叉链表表示法**
  - **存储结构**

每个节点有三个指针域，其中两个分别指向子节点（左孩子，右孩子），还有一共指针指向该节点的父节点。

  - **节点数据类型定义**

```
//三叉链表
typedef struct TriTNode
{
    int data;
    //左右孩子指针
    struct TriTNode *lchild, *rchild;
    struct TriTNode *parent;
}TriTNode, *TriTree;
```

4.4.3 二叉树的遍历

- 遍历定义

指按某条搜索路线**遍访每个结点且不重复**（又称周游）。

- 遍历用途

它是树结构插入、删除、修改、查找和排序运算的前提，是二叉树一切运算的基础和核心。

- 遍历方法

牢记一种约定，**对每个结点的查看都是“先左后右”** 。

限定先左后右，树的遍历有三种实现方案：

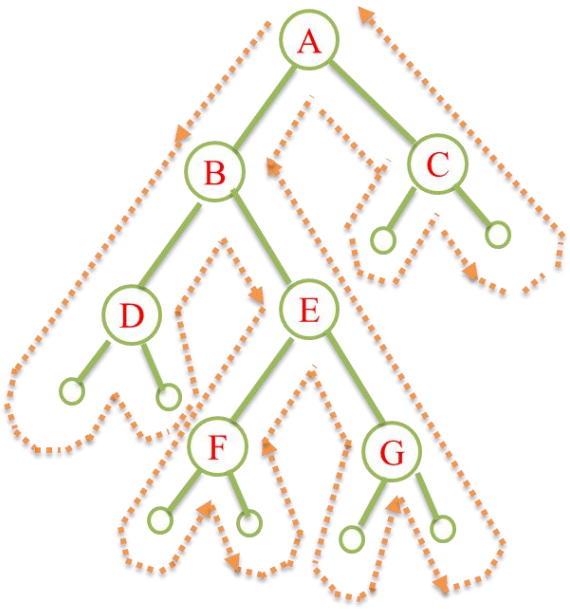
DLR　　LDR　　LRD

先 (**根**)序遍历　　**中** (**根**)序遍历　　**后**(**根**)序遍历

- **DLR — 先序遍历，即先根再左再右**
- **LDR — 中序遍历，即先左再根再右**
- **LRD — 后序遍历，即先左再右再根**

注：“先、中、后”的意思是指访问的结点 D 是先于子树出现还是后于子树出现。

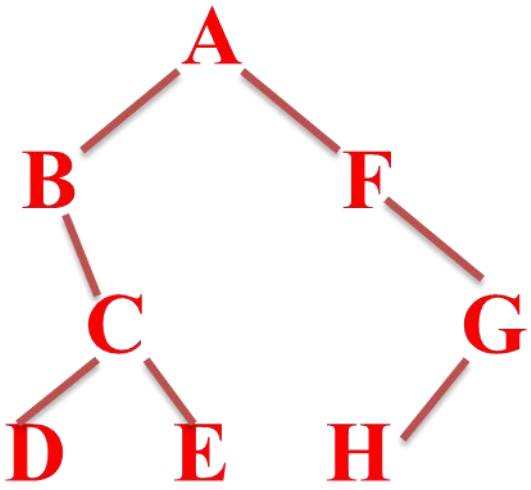
从递归的角度看，这三种算法是完全相同的，或者说这三种遍历算法的访问路径是相同的，只是访问结点的时机不同。



从虚线的出发点到终点的路径上，每个结点经过 3 次。

- **第 1 次经过时访问 = 先序遍历**
- **第 2 次经过时访问 = 中序遍历**
- **第 3 次经过时访问 = 后序遍历**

案例：



代码:

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

//二叉树结点 二叉链表
typedef struct _PTNode{
    char ch;
    struct _PTNode* lchild;
    struct _PTNode* rchild;
}PTNode;

//递归遍历
void RecursionBiTree(PTNode* root){
    if (root == NULL){
        return;
    }
    printf("%c", root->ch); //先序遍历
    RecursionBiTree(root->lchild);
    //printf("%c", root->ch); //中序遍历
    RecursionBiTree(root->rchild);
    //printf("%c", root->ch); //后序遍历
}

//测试
void test01(){
```

```
    //创建结点
    PTNode node1, node2, node3, node4, node5, node6, node7, node8;
    node1.ch = 'A'; node1.lchild = NULL; node1.rchild = NULL;
    node2.ch = 'B'; node2.lchild = NULL; node2.rchild = NULL;
    node3.ch = 'C';   node3.lchild = NULL; node3.rchild = NULL;
    node4.ch = 'D';   node4.lchild = NULL; node4.rchild = NULL;
    node5.ch = 'E';   node5.lchild = NULL; node5.rchild = NULL;
    node6.ch = 'F';   node6.lchild = NULL; node6.rchild = NULL;
    node7.ch = 'G';   node7.lchild = NULL; node7.rchild = NULL;
    node8.ch = 'H';   node8.lchild = NULL; node8.rchild = NULL;

    //建立结点关系
    node1.lchild = &node2;
    node1.rchild = &node6;

    node2.lchild = NULL;
    node2.rchild = &node3;

    node3.lchild = &node4;
    node3.rchild = &node5;

    node6.lchild = NULL;
    node6.rchild = &node7;

    node7.lchild = &node8;
    node7.rchild = NULL;

    RecursionBiTree(&node1);
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

非递归前序遍历：

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
```

```c
#include"LinkStack.h"

//二叉树结点 二叉链表
typedef struct _PTNode{
    char ch;
    struct _PTNode* lchild;
    struct _PTNode* rchild;
}PTNode;

//递归遍历
void RecursionBiTree(PTNode* root){
    if (root == NULL){
        return;
    }
    //printf("%c", root->ch); //先序遍历
    RecursionBiTree(root->lchild);
    printf("%c", root->ch); //中序遍历
    RecursionBiTree(root->rchild);
    //printf("%c", root->ch); //后序遍历
}

//非递归的方式遍历二叉树
void NonRecursion01(PTNode* root){

    //创建栈
    LinkStack* stack =  InitLinkStack();

    PTNode* pChild = root;
    PushLinkStack(stack,root);
    while (!IsEmptyLinkStack(stack) || pChild ){

        while (pChild != NULL){
            printf("%c\n",pChild->ch);
            if (pChild != root){
                PushLinkStack(stack, pChild);
            }
            pChild = pChild->lchild;
        }

        if (!IsEmptyLinkStack(stack)){
            pChild = (PTNode*)TopLinkStack(stack);
            PopLinkStack(stack);
            pChild = pChild->rchild;
        }
```

```
    }
}

//测试
void test01(){

    //创建结点
    PTNode node1, node2, node3, node4, node5, node6, node7, node8;
    node1.ch = 'A'; node1.lchild = NULL; node1.rchild = NULL;
    node2.ch = 'B'; node2.lchild = NULL; node2.rchild = NULL;
    node3.ch = 'C';   node3.lchild = NULL; node3.rchild = NULL;
    node4.ch = 'D';   node4.lchild = NULL; node4.rchild = NULL;
    node5.ch = 'E';   node5.lchild = NULL; node5.rchild = NULL;
    node6.ch = 'F';   node6.lchild = NULL; node6.rchild = NULL;
    node7.ch = 'G';   node7.lchild = NULL; node7.rchild = NULL;
    node8.ch = 'H';   node8.lchild = NULL; node8.rchild = NULL;

    //建立结点关系
    node1.lchild = &node2;
    node1.rchild = &node6;

    node2.lchild = NULL;
    node2.rchild = &node3;

    node3.lchild = &node4;
    node3.rchild = &node5;

    node6.lchild = NULL;
    node6.rchild = &node7;

    node7.lchild = &node8;
    node7.rchild = NULL;

    //非递归前序遍历
    printf("\n 非递归前序遍历:\n");
    NonRecursion01(&node1);
}

int main(){

    test01();
    system("pause");
    return EXIT_SUCCESS;
}
```

4.4.5 二叉树编程实践

案例 1：计算二叉树中叶子结点的数目

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

//二叉树结点 二叉链表
typedef struct _PTNode{
    char ch;
    struct _PTNode* lchild;
    struct _PTNode* rchild;
}PTNode;

//求叶子结点数量
int leafNum = 0;
void RecursionBiTree(PTNode* root){
    if (root == NULL){
        return;
    }

    if (!root->lchild && !root->rchild){
        leafNum++;
    }

    RecursionLeafNumTree(root->lchild);
    RecursionLeafNumTree(root->rchild);
}

//测试
void test01(){

    //创建结点
    PTNode node1, node2, node3, node4, node5, node6, node7, node8;
    node1.ch = 'A'; node1.lchild = NULL; node1.rchild = NULL;
    node2.ch = 'B'; node2.lchild = NULL; node2.rchild = NULL;
    node3.ch = 'C';   node3.lchild = NULL; node3.rchild = NULL;
    node4.ch = 'D';   node4.lchild = NULL; node4.rchild = NULL;
    node5.ch = 'E';   node5.lchild = NULL; node5.rchild = NULL;
    node6.ch = 'F';   node6.lchild = NULL; node6.rchild = NULL;
```

```
    node7.ch = 'G';   node7.lchild = NULL; node7.rchild = NULL;
    node8.ch = 'H';   node8.lchild = NULL; node8.rchild = NULL;

    //建立结点关系
    node1.lchild = &node2;
    node1.rchild = &node6;

    node2.lchild = NULL;
    node2.rchild = &node3;

    node3.lchild = &node4;
    node3.rchild = &node5;

    node6.lchild = NULL;
    node6.rchild = &node7;

    node7.lchild = &node8;
    node7.rchild = NULL;

    RecursionBiTree(&node1); //求叶子结点书目
    printf("叶子结点数量:%d\n",leafNum);
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

思想：

- 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。
- 若左子树还是树，重复步骤 1；若右子树还是树，重复步骤 1。
- **全局变量转成函数参数**
- 可以按照先序、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：

- 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。
- 若左子树还是树，重复步骤 1；若右子树还是树，重复步骤 1。

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

//二叉树结点 二叉链表
typedef struct _PTNode{
    char ch;
    struct _PTNode* lchild;
    struct _PTNode* rchild;
}PTNode;

//求二叉树深度
int RecursionTreeDepth(PTNode* root){

    int depth = 0;
    if (root == NULL){
        return depth;
    }

    int ldepth = RecursionTreeDepth(root->lchild);
    int rdepth = RecursionTreeDepth(root->rchild);
    depth = ldepth >= rdepth ? ldepth + 1 : rdepth + 1;

    return depth;
}

//测试
void test01(){

    //创建结点
    PTNode node1, node2, node3, node4, node5, node6, node7, node8;
    node1.ch = 'A';    node1.lchild = NULL; node1.rchild = NULL;
    node2.ch = 'B';    node2.lchild = NULL; node2.rchild = NULL;
```

```
    node3.ch = 'C';   node3.lchild = NULL; node3.rchild = NULL;
    node4.ch = 'D';   node4.lchild = NULL; node4.rchild = NULL;
    node5.ch = 'E';   node5.lchild = NULL; node5.rchild = NULL;
    node6.ch = 'F';   node6.lchild = NULL; node6.rchild = NULL;
    node7.ch = 'G';   node7.lchild = NULL; node7.rchild = NULL;
    node8.ch = 'H';   node8.lchild = NULL; node8.rchild = NULL;

    //建立结点关系
    node1.lchild = &node2;
    node1.rchild = &node6;

    node2.lchild = NULL;
    node2.rchild = &node3;

    node3.lchild = &node4;
    node3.rchild = &node5;

    node6.lchild = NULL;
    node6.rchild = &node7;

    node7.lchild = &node8;
    node7.rchild = NULL;

    int depth = RecursionTreeDepth(&node1);
    printf("树的深度为:%d\n",depth);
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

案例 3：Copy 二叉树

思想：

- malloc 新结点，
- 拷贝左子树，拷贝右子树，让新结点连接左子树，右子树。

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。

```c
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

//二叉树结点 二叉链表
typedef struct _PTNode{
    char ch;
    struct _PTNode* lchild;
    struct _PTNode* rchild;
}PTNode;

//递归遍历
void RecursionBiTree(PTNode* root){
    if (root == NULL){
        return;
    }
    printf("%c", root->ch); //先序遍历
    RecursionBiTree(root->lchild);
    //printf("%c", root->ch); //中序遍历
    RecursionBiTree(root->rchild);
    //printf("%c", root->ch); //后序遍历
}

//拷贝二叉树
PTNode* CopyTree(PTNode* root){
    if (root == NULL){
        return NULL;
    }

    //拷贝左子树
    if (root->lchild){
        root->lchild =  CopyTree(root->lchild);
    }
    else{
        root->lchild = NULL;
    }
    //拷贝右子树
    if (root->rchild){
        root->rchild = CopyTree(root->rchild);
    }
    else{
        root->rchild = NULL;
```

```
    }

    //创建新节点
    PTNode* newnode = (PTNode*)malloc(sizeof(PTNode));
    newnode->ch = root->ch;
    newnode->lchild = root->lchild;
    newnode->rchild = root->rchild;

    return newnode;
}

//测试
void test01(){

    //创建结点
    PTNode node1, node2, node3, node4, node5, node6, node7, node8;
    node1.ch = 'A'; node1.lchild = NULL; node1.rchild = NULL;
    node2.ch = 'B'; node2.lchild = NULL; node2.rchild = NULL;
    node3.ch = 'C';   node3.lchild = NULL; node3.rchild = NULL;
    node4.ch = 'D';   node4.lchild = NULL; node4.rchild = NULL;
    node5.ch = 'E';   node5.lchild = NULL; node5.rchild = NULL;
    node6.ch = 'F';   node6.lchild = NULL; node6.rchild = NULL;
    node7.ch = 'G';   node7.lchild = NULL; node7.rchild = NULL;
    node8.ch = 'H';   node8.lchild = NULL; node8.rchild = NULL;

    //建立结点关系
    node1.lchild = &node2;
    node1.rchild = &node6;

    node2.lchild = NULL;
    node2.rchild = &node3;

    node3.lchild = &node4;
    node3.rchild = &node5;

    node6.lchild = NULL;
    node6.rchild = &node7;

    node7.lchild = &node8;
    node7.rchild = NULL;

    printf("打印拷贝二叉树:\n");
    PTNode* root = CopyTree(&node1);
    RecursionBiTree(root); //打印结点
```

```
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

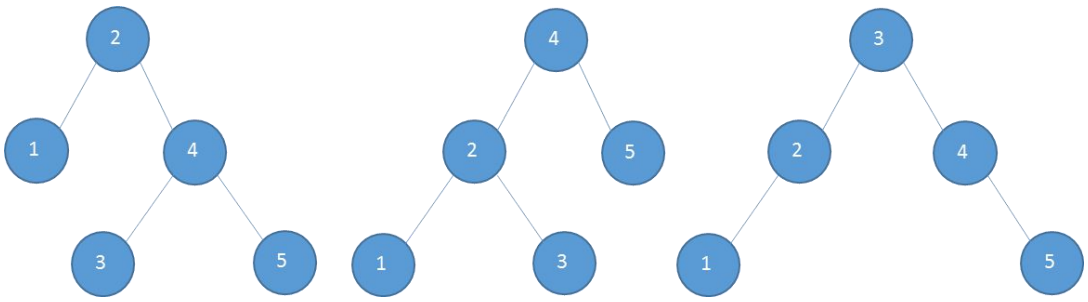
4.6 二叉树的创建

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。**树的形状能唯一确定吗？？？**



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

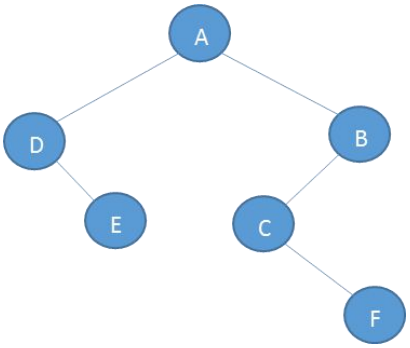
算法

- 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
- 在 A 的左子树中，找左子树的根结点（在先序中找），转步骤 1
- 在 A 的右子树中，找右子树的根结点（在先序中找），转步骤 1

练习 1

先序遍历结果：A　D　E　B　C　F

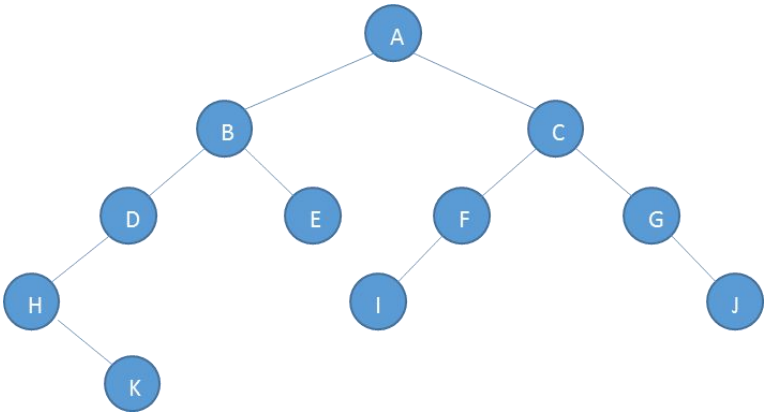
中序遍历结果：D　E　A　C　F　B



练习 2

先序遍历结果：A　B　D　H　K　E　C　F　I　G　J

中序遍历结果：H　K　D　B　E　A　I　F　C　G　J

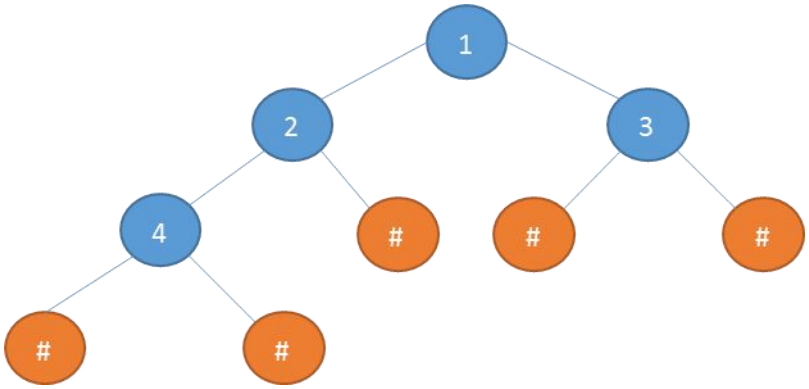


4.6.2#号法创建树

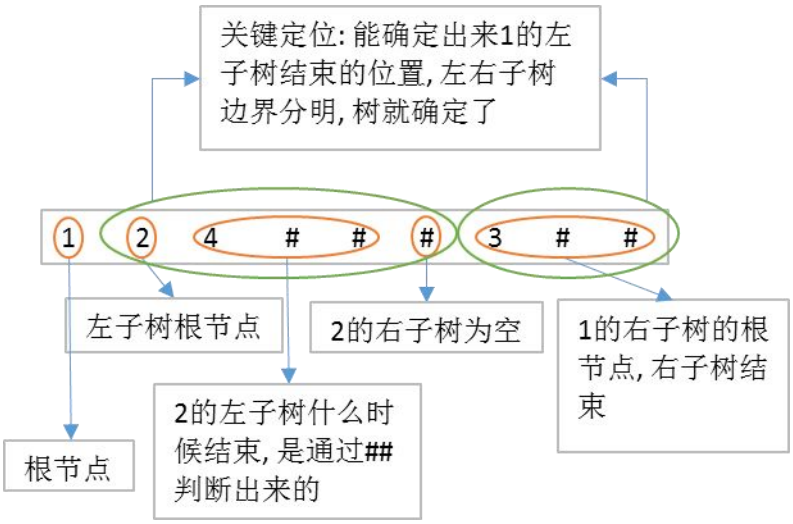
1、什么是#号法创建树

#创建树，让树的每一个节点都变成度数为 2 的树

先序遍历结果: 124###3##



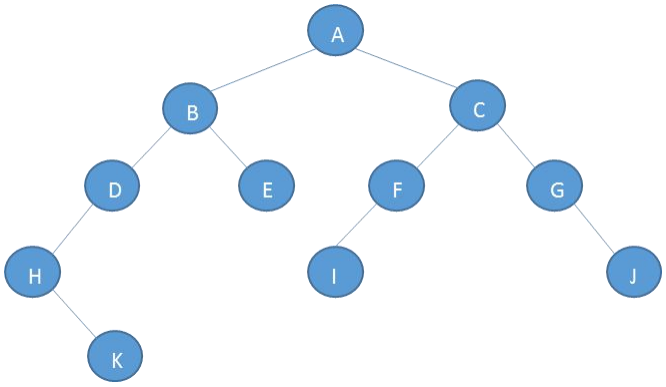
先序遍历：124###3##可以唯一确定一棵树吗，为什么？



确定出来的树模型: 上图即是。

2 #创建树练习 ABDH#K###E##CFI###G#J##

先序遍历：ABDH#K###E##CFI###G#J##，请画出树的形状：



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;
}
```

使用后序遍历的方式销毁一棵树， 先释放叶子节点，在释放根节点

//销毁树

```
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;
     }
 }
}
```

5.排序

5.1 排序基本概念

现实生活中排序很重要，例如：淘宝按条件搜索的结果展示等。

- 概念

  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}

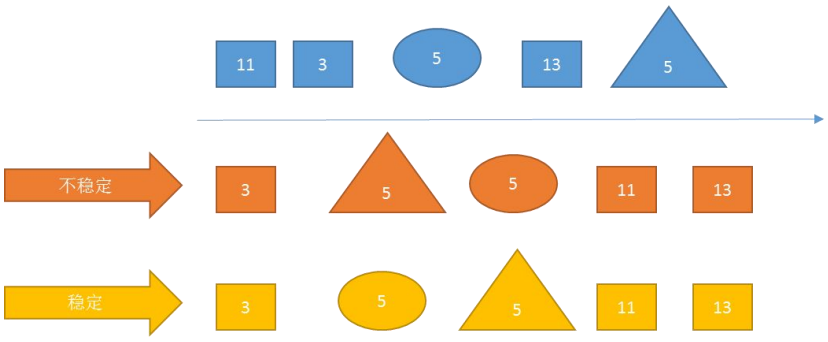
  这些关键字相互之间可以进行比较，即在它们之间存在着这样一个关系 ：

  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。



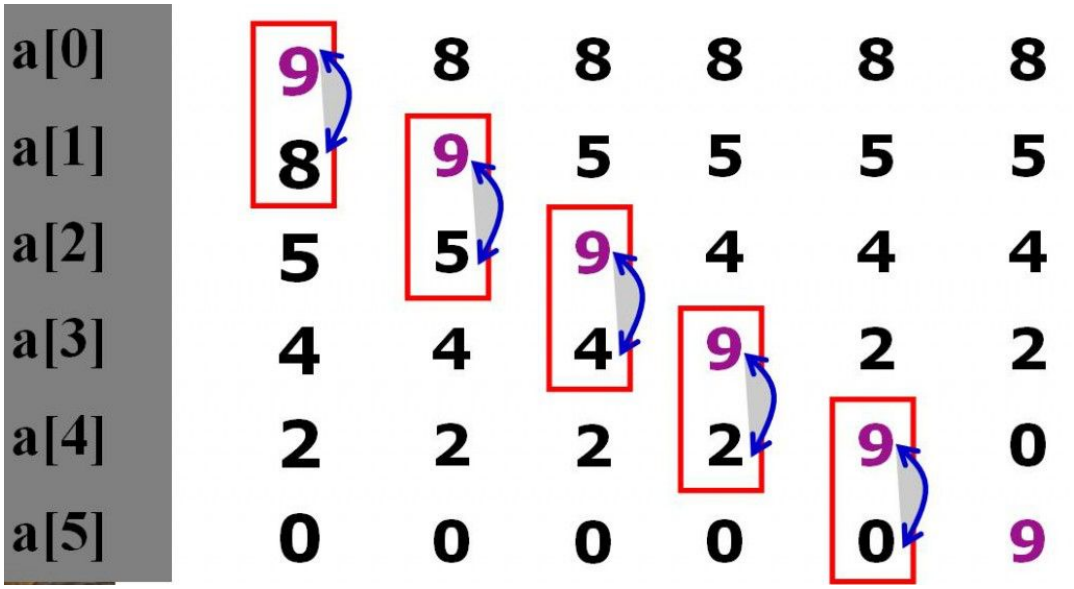
- 多关键字排序

  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：在排序过程中，待排序的所有记录全部都放置在内存中，排序分为：内排序和外排序。
  - 外排序：由于排序的记录个数太多，不能同时放置在内存，整个排序过程需要在内外存之间多次交换数据才能进行。
- 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 冒泡排序



5.2.1 冒泡排序实现案例

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

//打印数组
void PrintArray(int* arr,int length){
    for (int i = 0; i < length;i++){
        printf("%d ",arr[i]);
    }
    printf("\n");
}

//交换函数
void swap(int* arr,int pos1,int pos2){

    int temp = arr[pos1];
```

```
    arr[pos1] = arr[pos2];
    arr[pos2] = temp;

}

//冒泡排序
void BubbleSort(int* arr, int length){

    int i,j;
    for (i = 0; i < length; i ++){

        for (j = length - 1; j > i;j --){

            if (arr[j-1] < arr[j]){
                swap(arr,i,j);
            }
        }
    }
}

void test01(){

    int array[] = {23,12,89,6,45,77,67};
    int length = sizeof(array) / sizeof(int);
    //打印排序前的数组
    printf("打印冒泡排序前的数组:\n");
    PrintArray(array, length);
    //对数组进行冒泡排序
    BubbleSort(array,length);
    printf("打印冒泡排序后的数组:\n");
    PrintArray(array, length);
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

5.2.2 冒泡排序(改进版)实现案例

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define BUBBLE_TRUE 1
#define BUBBLE_FALSE 0

//打印数组
void PrintArray(int* arr,int length){
    for (int i = 0; i < length;i++){
        printf("%d ",arr[i]);
    }
    printf("\n");
}

//交换函数
void swap(int* arr,int pos1,int pos2){
    int temp = arr[pos1];
    arr[pos1] = arr[pos2];
    arr[pos2] = temp;
}

//冒泡排序
void BubbleSort(int* arr, int length){

    int i,j;
    int flag = BUBBLE_TRUE; //是否排序完成 BUBBLE_TRUE 表示没有排序完成，BUBBLE_FALSE 表示排序完成
    for (i = 0; i < length && flag == BUBBLE_TRUE; i++){
        flag = BUBBLE_FALSE;
        for (j = length - 1; j > i;j --){

            if (arr[j-1] < arr[j]){
                flag = BUBBLE_TRUE;
                swap(arr,i,j);
            }
        }
    }
}
```

```
void test01(){

    int array[] = {23,12,89,6,45,77,67};
    int length = sizeof(array) / sizeof(int);
    //打印排序前的数组
    printf("打印冒泡排序前的数组:\n");
    PrintArray(array, length);
    //对数组进行冒泡排序
    BubbleSort(array,length);
    printf("打印冒泡排序后的数组:\n");
    PrintArray(array, length);
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

5.2.3 冒泡排序总结

- 冒泡总结：
  - 冒泡排序是一种效率低下的排序方法，在数据规模很小时，可以采用。数据规模比较大时，最好用其它排序方法。
  - 上述例子总对冒泡做了优化，添加了 flag 作为标记，记录序列是否已经有序，减少循环次数。
- 稳定性
  - **冒泡排序是一种稳定的排序算法**
  - **冒泡排序的效率：O（n^2）**

5.3 选择排序

初始序列：	9	0	5	3	7	8	4	2	10	1
第一遍排序：	【0】	9	5	3	7	8	4	2	10	1
第二遍排序：	【0	1】	5	3	7	8	4	2	10	9
第三遍排序：	【0	1	2】	3	7	8	4	5	10	9
第四遍排序：	【0	1	2	3】	7	8	4	5	10	9
第五遍排序：	【0	1	2	3	4】	8	7	5	10	9
第六遍排序：	【0	1	2	3	4	5】	7	8	10	9
第七遍排序：	【0	1	2	3	4	5	7】	8	10	9
第八遍排序：	【0	1	2	3	4	5	7	8】	10	9
第九遍排序：	【0	1	2	3	4	5	7	8	9】	10
排序结果：	【0	1	2	3	4	5	7	8	9	10】

图 1：选择法排序具体示例

5.3.1 选择排序实现案例

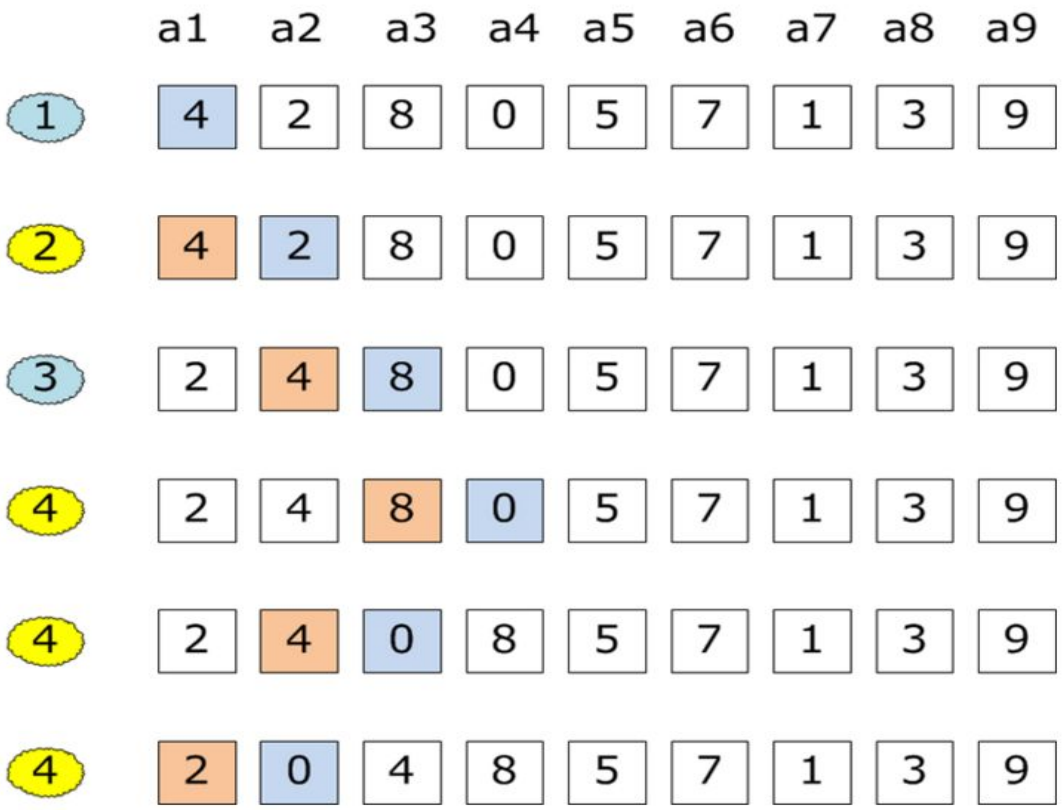
```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

//打印数组
void PrintArray(int* arr, int length){
    for (int i = 0; i < length; i++){
        printf("%d ", arr[i]);
    }
    printf("\n");
}

//交换函数
void swap(int* arr, int pos1, int pos2){

    int temp = arr[pos1];
    arr[pos1] = arr[pos2];
    arr[pos2] = temp;

}

//选择排序
```

```c
void SelectSort(int* arr,int length){

    int i, j;
    int min;

    for (i = 0; i < length; i++){

        min = i;
        for (j = i + 1; j < length;j++){

            if (arr[min] > arr[j]){
                min = j;
            }
        }

        if (i != min){
            swap(arr,min,i);
        }
    }
}

void test01(){

    int array[] = { 23, 12, 89, 6, 45, 77, 67 };
    int length = sizeof(array) / sizeof(int);
    //打印选择排序前的数组
    printf("打印选择排序前的数组:\n");
    PrintArray(array,length);
    //对数组进行选择排序
    SelectSort(array, length);
    //打印选择排序前的数组
    printf("打印选择排序后的数组:\n");
    PrintArray(array, length);
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

5.3.2 选择排序总结

- 稳定性
  - **选择排序是不稳定的排序方法**
  - **选择排序效率：O（n^2）**

5.4 插入排序



5.4.1 插入排序实现案例

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>


//打印数组
void PrintArray(int* arr, int length){
```

```
    for (int i = 0; i < length; i++){
        printf("%d ", arr[i]);
    }
    printf("\n");
}

//插入排序
void InsertSort(int* arr, int length){

    int i, j;
    for (i = 1; i < length; i++){

        if (arr[i] < arr[i-1]){

            int temp = arr[i];
            for (j = i - 1; j >= 0 && arr[j] > temp; j--){
                arr[j + 1] = arr[j];
            }
            arr[j+1] = temp;
        }
    }
}

void test01(){

    int array[] = { 23, 12, 89, 6, 45, 77, 67 };
    int length = sizeof(array) / sizeof(int);
    //打印插入排序前的数组
    printf("打印插入排序前的数组:\n");
    PrintArray(array, length);
    //对数组进行插入排序
    InsertSort(array, length);
    printf("打印插入排序后的数组:\n");
    PrintArray(array, length);
}

int main(){

    test01();



    system("pause");
    return EXIT_SUCCESS;
}
```

5.4.2 插入排序总结

- 稳定性
  - **插入排序是稳定的排序算法**
  - **插入排序效率：O（n^2）**

5.5 希尔排序

- 算法介绍

  希尔排序的实质就是分组插入排序，该方法又称缩小增量排序，因 DL．Shell 于 1959 年提出而得名。

- 基本思想

  先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。因为直接插入排序在元素基本有序的情况下（接近最好情况），效率是很高的，因此希尔排序在时间效率上比前三种方法有较大提高。

5.5.1 希尔排序实现案例

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

//打印数组
```

```c
void PrintArray(int* arr, int length){
    for (int i = 0; i < length; i++){
        printf("%d ", arr[i]);
    }
    printf("\n");
}

//希尔排序
void ShellSort(int* arr, int length){

    int i, j, k;
    int increament = length;  //初始增量

    while (increament > 1){

        increament = increament / 3 + 1;
        //分组
        for (i = 0; i < increament; i ++){
            //遍历每一个分组，并对每一组的元素进行插入排序
            for (j = i + increament; j < length; j+=increament){
                int temp = arr[j];
                if (arr[j-increament] > arr[j]){
                    for (k = j - increament; k >= 0 && temp < arr[k]; k -= increament){
                        arr[k + increament] = arr[k];
                    }
                    arr[k + increament] = temp;
                }
            }
        }
    }
}

void test01(){

    int array[] = { 23, 12, 89, 6, 45, 77, 67 };
    int length = sizeof(array) / sizeof(int);
    //打印希尔排序前的数组
    printf("打印希尔排序前的数组:\n");
    PrintArray(array, length);
    //对数组进行希尔排序
    ShellSort(array, length);
    printf("打印希尔排序后的数组:\n");
    PrintArray(array, length);
```

```
}

int main(){

    test01();
    system("pause");
    return EXIT_SUCCESS;
}
```

通过这段代码，大家应该能够看到在希尔排序中我们并不是随便的把序列分组，然后对每个子序列分别排序，而是将相隔“增量”的记录组成一个子序列，实现跳跃式移动，使得排序效率提高。

这个增量的选取就非常关键了。我们案例中是用 Increasement = increasement / 3 + 1,至于这个增量如何选取非常难，没有明确的确定，但是通过前人的研究，我们用

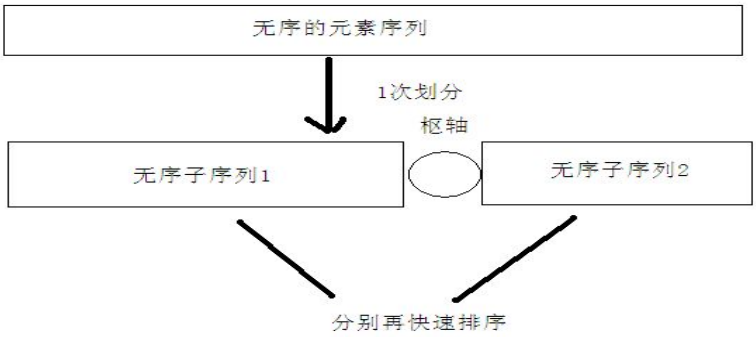
步长 = 步长 / 3 + 1。

5.6 快速排序

- 算法介绍

  快速排序是 C.R.A.Hoare 于 1962 年提出的**一种划分交换排序**。它**采用了一种分治的策略**，通常称其为分治法(Divide-and-ConquerMethod)。

- 分治法基本思想
  - 先从数列中取出一个数作为基准数（枢轴）。
  - **分区过程将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边。(升序)**
  - 再对左右区间重复第二步，直到各区间只有一个数。



5.6.1 快速排序实现案例

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

//打印数组
void PrintArray(int* arr, int length){
    for (int i = 0; i < length; i++){
        printf("%d ", arr[i]);
    }
    printf("\n");
}

//快速排序
void QuickSort(int* arr,int low,int height){

    int left = low;
    int right = height;
    //取出基准数
    int target = arr[left];

    if (left < right){

        while (left < right){

            //因为在左边挖坑，从右面开始  找比基准数小的
            while (left < right && arr[right] > target){
                right--;
            }
            if (left < right){
                arr[left] = arr[right];
```

```
                left++;
            }

            //从左向右找比基准数大的
            while (left < right && arr[left] < target){
                left++;
            }
            if (left < right){
                arr[right] = arr[left];
                right--;
            }

        }
        //这个时候 left 和 right 同时指向的位置，将基准数插入这个位置
        arr[left] = target;

        QuickSort(arr, low, left - 1);
        QuickSort(arr, left + 1, height);

    }
}

void test01(){

    int array[] = { 23, 12, 89, 6, 45, 77, 67 };
    int length = sizeof(array) / sizeof(int);
    //打印快速排序前的数组
    printf("打印快速排序前的数组:\n");
    PrintArray(array, length);
    //对数组进行快速排序
    QuickSort(array, 0,length-1);
    printf("打印快速排序后的数组:\n");
    PrintArray(array, length);

}

int main(){
    test01();
    system("pause");
    return EXIT_SUCCESS;
}
```

5.7 归并排序

- 算法介绍

  归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

- 基本思想

  基本思路就是将数组分成二组 A，B，如果这二组组内的数据都是有序的，那么就可以很方便的将这二组数据进行排序。如何让这二组组内数据有序了？

  可以将 A，B 组各自再分成二组。依次类推，当分出来的小组只有一个数据时，可以认为这个小组组内已经达到了有序，然后再合并相邻的二个小组就可以了。这样通过先递归的分解数列，再合并数列就完成了归并排序。

- **归并的定义**

  将两个或两个以上的有序序列合并成一个新的有序序列：

  有序序列V[1] ...V[m]和V[m+1] ...V[n]

  ↓

  V[1] ...V[n]

  这种归并方法称为2路归并。

  将3个有序序列归并为一个新的有序序列，称为3路归并。

  将多个有序序列归并为一个新的有序序列，称为多路归并。

- **如何合并连个有序序列？？？**

只要从比较二个数列的第一个数，谁小就先取谁，取了后就在对应数列中删除这个数。然后再进行比较，如果有数列为空，那直接将另一个数列的数据依次取出即可。

5.6.1 归并排序实现案例

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>



//打印数组
void PrintArray(int* arr, int length){
    for (int i = 0; i < length; i++){
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void Merge(int arr[],int first,int last,int mid,int temp[]){

    int i = first; //第一个有序序列开始的下标
    int j = mid + 1; //第二个有序序列开始的下标
    int length = 0;

    //开始合并两个有序序列
    while (i <= mid && j <= last){

        //找当前两个数中最小的数
        if (arr[i] < arr[j]){
            //将最小数放到 temp 中
            temp[length] = arr[i];
            i++;
        }
        else{
            temp[length] = arr[j];
            j++;
        }

        length++;
```

```
    }

    //两个序列中肯定有一个还剩下数据，但是这里我不知道那个还剩下数据,所以我写两个 while 循环
    while (i <= mid){
        temp[length] = arr[i];
        i++;
        length++;
    }

    while (j <= last){
        temp[length] = arr[j];
        j++;
        length++;
    }

    //覆盖原来位置的无序序列
    for (int i = 0; i < length; i++){
        //printf("%d ",temp[i]);
        arr[first + i] = temp[i];
    }
    //printf("\n");

}

//归并排序
void MergeSort(int arr[], int first, int last, int temp[]){

    if (first == last){
        return;
    }
    //从哪里拆分？我们这里从中间拆分
    int mid = (first + last) / 2;
    //对左半部分进行拆分
    MergeSort(arr, first, mid, temp);
    //对右半部分拆分
    MergeSort(arr, mid + 1, last, temp);
    //拆分完了，然后合并
    Merge(arr, first, last, mid, temp);
}

void test01(){

    int array[] = { 23, 12, 89, 6, 45, 77, 67};
```

```
    int length = sizeof(array) / sizeof(int);
    //打印归并排序前的数组
    printf("打印归并排序前的数组:\n");
    PrintArray(array, length);
    //对数组进行归并排序
    int* temp = (int*)malloc(sizeof(int)* length);
    MergeSort(array, 0, length - 1, temp);
    free(temp);
    printf("打印归并排序后的数组:\n");
    PrintArray(array, length);

}

int main(){
    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

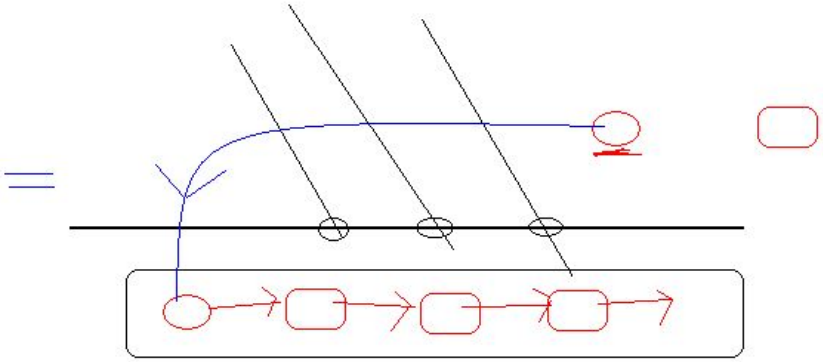
5.8 排序总结

排序算法	平均时间复杂度	最坏时间复杂度	平均空间复杂度	稳定性
选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(nlogn)$	$O(n^2)$	$O(log_2n)$	不稳定
快速排序	$O(nlogn)$	$O(n^2)$	$O(1)$	不稳定
归并排序	$O(nlogn)$	$O(nlogn)$	$O(n)$	稳定

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。



1 容器中缓存了 用户的结点
2 结点的类,要保证结点能够插入到容器中
一般结点类,需要提供
无参构造函数
拷贝构造函数
重载=操作符

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

6.2.1 链表类_链式存储设计与实现

- LinkList.hpp

```
#ifndef LINKLIST_HPP
#define LINKLIST_HPP
#include<string>

//c++模板完成单向链表

//结点结构
template<class T>
class ListNode{
```

```
public:
    T data;  //数据域
    ListNode* next; //指针域
};

//定义链表类
template<class T>
class LinkList{
public:

    //构造函数
    LinkList():length(0),head(NULL){}

    //链表操作相关 API
    //设置链表长度
    void setLength(int length){
        if (length < 0){
            return;
        }
        this->length = length;
    }
    //获得链表长度
    int getLength(){
        return this->length;
    }
    //设置链表头结点
    void setHead(ListNode<T>* head){
        this->head = head;
    }
    //获得链表的头结点
    ListNode<T>* getHead(){
        return this->head;
    }
    //插入元素
    int Insert(int pos, T value){

        //创建结点
        ListNode<T>* newnode = new ListNode<T>();
        newnode->next = NULL;
        newnode->data = value;

        //第一次插入
        if (this->head == NULL){
            this->head = newnode;
```

```
            this->length++;
            return 0;
        }

        //头插法
        ListNode<T>* pCurrent = head;
        if(pos == 0){
            newnode->next = pCurrent;
            this->head = newnode;
            this->length++;
            return 0;
        }

        //找到插入位置
        //ListNode<T>* pCurrent = head;
        for (int i = 1; i < pos;i++){
            if (pCurrent->next == NULL){
                break;
            }
            pCurrent = pCurrent->next;
        }
        //插入新的结点
        newnode->next = pCurrent->next;
        pCurrent->next = newnode;
        this->length++;
    }

    //删除某个位置的结点
    int Delete(int pos){

        if (this->length == 0){
            return -1;
        }
        if(pos > this->length || pos < 0){
            return -2;
        }

        //头删法
        if (pos == 0){
            ListNode<T>* pDel = this->head;
            this->head = pDel->next;
            this->length--;
            delete pDel;
            return 0;
```

```
        }

        //找删除的位置
        ListNode<T>* pCurrent = head;
        for (int i = 1; i < pos;i++){
            pCurrent = pCurrent->next;
        }

        ListNode<T>* pDel = pCurrent->next;
        //重新连接结点
        pCurrent->next = pDel->next;
        //删除结点
        delete pDel;
        this->length--;
    }

    //判断链表是否为空
    bool IsEmpty(){
        if (this->length == 0){
            return true;
        }
        return false;
    }
    ~LinkList(){
        while (this->length){
            Delete(0);
        }
    }
private:
    int length; //保存结点数量
    ListNode<T>* head;
};

#endif
```

- LinkListTest.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include"LinkList.hpp"
using namespace std;

//测试单向链表容器
void test01(){
```

```
    //创建容器
    LinkList<int> list;
    //向容器中插入数据
    for (int i = 0; i < 10; i++){
        list.Insert(list.getLength(), i);
    }
    //遍历打印链表
    ListNode<int>* pCurrent = list.getHead();
    while (pCurrent != NULL){
        cout << pCurrent->data << " ";
        pCurrent = pCurrent->next;
    }
    cout << endl;


    //删除结点
    list.Delete(0);
    list.Delete(1);

    //遍历打印链表
    pCurrent = list.getHead();
    while (pCurrent != NULL){
        cout << pCurrent->data << " ";
        pCurrent = pCurrent->next;
    }
    cout << endl;
}


//Teacher 类
class Teacher{
public:
    Teacher(){}
    Teacher(string name, int age){
        this->m_name = name;
        this->m_age = age;
    }
    //拷贝构造
    Teacher(const Teacher& t){
        this->m_name = t.m_name;
        this->m_age = t.m_age;
    }
    //重载=操作符
```

```
    Teacher operator=(Teacher& t){
        this->m_name = t.m_name;
        this->m_age = t.m_age;
        return *this;
    }
public:
    string m_name;
    int m_age;
};

//容器中存储对象 重点：容器元素都是值寓意，而非引用寓意
void test02(){

    //创建两个 Teacher 类的实例
    Teacher t1("aaa", 10), t2("bbb", 20);
    //创建容器
    LinkList<Teacher> list;
    //向 list 中插入元素
    list.Insert(0, t1);
    list.Insert(0, t2);
    //打印容器中的元素的值
    ListNode<Teacher>* pCurrent = list.getHead();
    while (pCurrent != NULL){
        cout << "Name:" << pCurrent->data.m_name << " Age:" << pCurrent->data.m_age << endl;
        pCurrent = pCurrent->next;
    }
    list.Delete(1);
    //打印容器中的元素的值
    pCurrent = list.getHead();
    while (pCurrent != NULL){
        cout << "Name:" << pCurrent->data.m_name << " Age:" << pCurrent->data.m_age << endl;
        pCurrent = pCurrent->next;
    }

}

//容器中存储指针
void test03(){

    //创建两个 Teacher 类的实例
    Teacher t1("aaa", 10), t2("bbb", 20);
    //创建容器
    LinkList<Teacher*> list;
    //插入元素
```

```
    list.Insert(0, &t1);
    list.Insert(0, &t2);

    //打印容器中的元素的值
    ListNode<Teacher*>* pCurrent = list.getHead();
    while (pCurrent != NULL){
        cout << "Name:" << pCurrent->data->m_name << " Age:" << pCurrent->data->m_age <<
endl;
        pCurrent = pCurrent->next;
    }
}

int main(){

    //test01();
    //test02();
    test03();

    system("pause");
    return EXIT_SUCCESS;
}
```

6.2.2 栈类_链式存储设计与实现

- LinkStack.hpp

```
#ifndef LINKQUEUE_HPP
#define LINKQUEUE_HPP

//结点结构
template<class T>
class LinkNode{
public:
    T data;
    LinkNode* next;
};

//队列类
template<class T>
class LinkQueue{
public:
    //初始化链式队列
```

```
LinkQueue(){
    pFront = NULL;
    pBack = NULL;
    mLength = 0;
}
//获得队列长度
int getLength(){
    return this->mLength;
}
//队列加入元素
void Push(T data){

    //创建新的结点
    LinkNode<T>* newnode = new LinkNode<T>();
    newnode->data = data;
    newnode->next = NULL;

    //判断是不是第一次插入
    if (pFront == NULL && pBack == NULL){
        pFront = newnode;
        pBack = newnode;
        this->mLength++;
        return;
    }

    //其他情况
    this->pBack->next = newnode;
    this->pBack = newnode;
    this->mLength++;

    return;
}

T& Front(){
    return this->pFront->data;
}

void Pop(){

    if (this->mLength == 0){
        return;
    }

    //当队列中只有一个元素的时候
```

```
        if (this->mLength == 1){
            delete this->pFront;
            this->pFront = NULL;
            this->pBack = NULL;
            this->mLength--;
            return;
        }

        //其他情况
        LinkNode<T>* pDel = this->pFront;
        this->pFront = pDel->next;
        delete pDel;
        this->mLength--;
        return;
    }
    ~LinkQueue(){
        while (this->mLength > 0){
            Pop();
        }
    }
private:
    LinkNode<T>* pFront; //队头
    LinkNode<T>* pBack; //队尾
    int mLength; //队列长度
};
#endif
```

- LinkStackTest.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<string>
#include"LinkStack.hpp"
using namespace std;

//栈容器存储基础数据类型
void test01(){

    //创建栈容器
    LinkStack<int> lstack;
    //向栈中加入元素
    for (int i = 10; i < 20;i++){
        lstack.Push(i);
    }
```

```
    //打印栈中元素
    while (lstack.getLength() > 0){
        int val = lstack.Top();
        cout << val << " ";
        lstack.Pop();
    }
    cout << endl;
}

//栈容器存储对象

//Teacher 类
class Teacher{
public:
    Teacher(){}
    Teacher(string name, int age){
        this->m_name = name;
        this->m_age = age;
    }
    //拷贝构造
    Teacher(const Teacher& t){
        this->m_name = t.m_name;
        this->m_age = t.m_age;
    }
    //重载=操作符
    Teacher operator=(Teacher& t){
        this->m_name = t.m_name;
        this->m_age = t.m_age;
        return *this;
    }
public:
    string m_name;
    int m_age;
};

void test02(){

    //创建栈
    LinkStack<Teacher> lstack;
    //插入数据
    Teacher t1("aaa", 10), t2("bbb", 20),t3("ccc",30);
    lstack.Push(t1);
    lstack.Push(t2);
    lstack.Push(t3);
```

```
    //遍历打印
    while (lstack.getLength() > 0){
        Teacher teahcer = lstack.Top();
        cout << "Name:" << teahcer.m_name << " Age:" << teahcer.m_age << endl;;
        lstack.Pop();
    }

}

//容器存储对象指针
void test03(){

    //创建栈
    LinkStack<Teacher*> lstack;
    //插入数据
    Teacher t1("aaa", 10), t2("bbb", 20), t3("ccc", 30);
    lstack.Push(&t1);
    lstack.Push(&t2);
    lstack.Push(&t3);
    //遍历打印
    while (lstack.getLength() > 0){
        Teacher* teahcer = lstack.Top();
        cout << "Name:" << teahcer->m_name << " Age:" << teahcer->m_age << endl;;
        lstack.Pop();
    }

}

int main(){

    //test01();
    //test02();
    test03();
    system("pause");
    return EXIT_SUCCESS;
}
```

6.2.3 队列类_链式存储设计与实现

- LinkQueue.hpp

```
#ifndef LINKQUEUE_HPP
```

```
#define LINKQUEUE_HPP

//结点结构
template<class T>
class LinkNode{
public:
    T data;
    LinkNode* next;
};

//队列类
template<class T>
class LinkQueue{
public:
    //初始化链式队列
    LinkQueue(){
        pFront = NULL;
        pBack = NULL;
        mLength = 0;
    }
    //获得队列长度
    int getLength(){
        return this->mLength;
    }
    //队列加入元素
    void Push(T data){

        //创建新的结点
        LinkNode<T>* newnode = new LinkNode<T>();
        newnode->data = data;
        newnode->next = NULL;

        //判断是不是第一次插入
        if (pFront == NULL && pBack == NULL){
            pFront = newnode;
            pBack = newnode;
            this->mLength++;
            return;
        }

        //其他情况
        this->pBack->next = newnode;
        this->pBack = newnode;
        this->mLength++;
```

```
        return;
    }

    T& Front(){
        return this->pFront->data;
    }

    void Pop(){

        if (this->mLength == 0){
            return;
        }

        //当队列中只有一个元素的时候
        if (this->mLength == 1){
            delete this->pFront;
            this->pFront = NULL;
            this->pBack = NULL;
            this->mLength--;
            return;
        }

        //其他情况
        LinkNode<T>* pDel = this->pFront;
        this->pFront = pDel->next;
        delete pDel;
        this->mLength--;
        return;
    }

private:
    LinkNode<T>* pFront; //队头
    LinkNode<T>* pBack; //队尾
    int mLength; //队列长度
};

#endif
```

- LinkQueueTest.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<string>
```

```
#include"LinkQueue.hpp"
using namespace std;

//存储基础数据类型
void test01(){

    //创建链式队列
    LinkQueue<int> queue;
    //添加数据
    for (int i = 0; i < 10;i++){
        queue.Push(i);
    }
    //修改队头元素的值
    queue.Front() = 100;
    //打印队列
    while (queue.getLength() > 0){
        int val = queue.Front();
        cout << val << " ";
        queue.Pop();
    }
    cout << endl;
}

int main(){

    test01();

    system("pause");
    return EXIT_SUCCESS;
}
```

6.2.4 链表类_顺序存储设计与实现

- SqList.hpp

```
#ifndef SQLIST_HPP
#define SQLIST_HPP

//链表类
template<class T>
class SqList{
public:
```

```
    //构造函数，由用户指定容量
    SqList(int capacity){
        this->mCapacity = capacity;
        this->mLength = 0;
        this->mArray = new T[capacity];
    }
    //获得容量
    int getCapacity(){
        return this->mCapacity;
    }
    //获得链表长度
    int getLength(){
        return this->mLength;
    }
    //获得数组
    T* getArray(){
        return this->mArray;
    }
    //获得指定位置元素
    T Get(int pos){
        if (pos > this->mLength - 1){
            return -1;
        }
        return this->mArray[pos];
    }
    //链表中指定位置插入结点
    void Insert(int pos, T data){
        if (this->mLength >= this->mCapacity){
            return;
        }
        if (pos > this->mLength){
            pos = this->mLength - 1;
        }

        this->mArray[pos] = data;
        this->mLength++;
    }
    //删除指定位置结点
    void Delete(int pos){

        if (pos > this->mLength-1){
            return;
        }
```

```
        for (int i = pos; i < mLength;i++){
            this->mArray[i] = this->mArray[i + 1];
        }
        this->mLength--;
    }

    ~SqList(){
        if (mArray != NULL){
            delete mArray;
        }
    }
private:
    int mCapacity;//容量
    int mLength;//长度
    T* mArray;
};

#endif
```

- SqListTest.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include"SqList.hpp"
using namespace std;

void test01(){

    //创建栈
    SqList<int> list(20);
    //插入数据
    for (int i = 0; i < 20;i++){
        list.Insert(i,i+1);
    }
    //打印数据
    for (int i = 0; i < 20;i++){
        cout << list.Get(i) << " ";
    }
    cout << endl;
}

int main(){
    test01();
    system("pause");
```

```
    return EXIT_SUCCESS;
}
```

6.2.5 栈类_顺序存储设计与实现

- SqStack.hpp

```
#ifndef SQSTACK_HPP
#define SQSTACK_HPP

//栈的顺序存储
template<class T>
class SqStack{
public:
    //构造函数
    SqStack(int capacity){
        this->mCapacity = capacity;
        this->mLength = 0;
        this->mArray = new T[capacity];
    }
    int getLength(){
        return this->mLength;
    }
    //压栈
    void Push(T data){

        if (this->mLength >= this->mCapacity){
            return;
        }

        this->mArray[this->mLength] = data;
        this->mLength++;

    }

    //出栈
    T Top(){
        return this->mArray[this->mLength - 1];
    }

    //弹出栈顶元素
    void Pop(){
```

```
        this->mLength--;
    }

    ~SqStack(){
        if (this->mArray != NULL){
            delete this->mArray;
        }
    }
private:
    T* mArray;
    int mCapacity; //容量
    int mLength; //长度
};

#endif
```

- SqStackTest.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include"SqStack.hpp"
using namespace std;

void test01(){

    //创建栈
    SqStack<int> stack(20);
    //压栈
    for (int i = 0; i < 20;i++){
        stack.Push(i);
    }
    //遍历
    while (stack.getLength() > 0){
        cout << stack.Top() << " ";
        stack.Pop();
    }
    cout << endl;

}

int main(){
    test01();
    system("pause");
    return EXIT_SUCCESS;
```

```
}
```

6.2.6 队列类_顺序存储设计与实现

- SqQueue.hpp

```
#ifndef SQQUEUE_HPP
#define SQQUEUE_HPP

//队列的顺序存储
template<class T>
class SqQueue{
public:
    SqQueue(int capacity){
        this->mCapacity = capacity;
        this->mLength = 0;
        this->mArray = new T[capacity];
    }
    //获得队列长度
    int getLength(){
        return this->mLength;
    }

    //获得容量
    int getCapacity(){
        return this->mCapacity;
    }

    //入队操作
    void Push(T data){
        if (this->mLength >= this->mCapacity){
            return;
        }
        this->mArray[mLength] = data;
        this->mLength++;
    }

    //出队操作
    T Front(){
        return this->mArray[0];
    }
```

```
    //队头弹出元素
    void Pop(){
        for (int i = 0; i < mLength; i++){
            mArray[i] = mArray[i + 1];
        }
        this->mLength--;
    }
private:
    int mLength;
    T* mArray;
    int mCapacity;
};

#endif
```

- SqQueueTest.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include"SqQueue.hpp"
using namespace std;

void test01(){

    //创建队列
    SqQueue<int> queue(20);
    //向队列中插入元素
    for (int i = 0; i < 20;i++){
        queue.Push(i);
    }
    //打印队列元素
    while (queue.getLength() > 0){
        cout << queue.Front() << " ";
        queue.Pop();
    }
    cout << endl;

}

int main(){
    test01();
    system("pause");
    return EXIT_SUCCESS;
```

```
}
```