北京航空航天大學
BEIHANG UNIVERSITY


数据结构与程序设计（信息类）

Data Structure & Programming

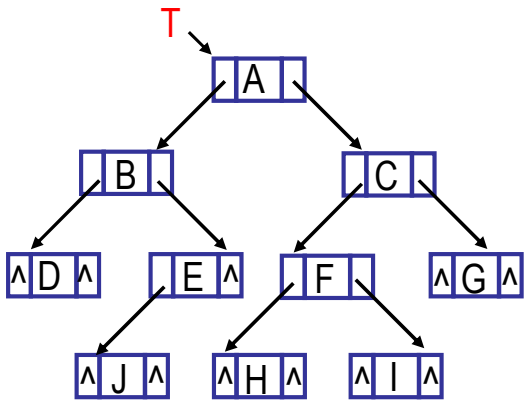
北京航空航天大学 数据结构课程组
软件学院 林广艳

2023年春

提纲：树和二叉树

- 树的基本概念
- 树的存储结构
- 二叉树的基本概念
- 二叉树的存储结构
- 二叉树的遍历
- **线索二叉树**
- 二叉查找树
- 堆和表达式树
- 哈夫曼树

6. 线索二叉树*



中序序列:

D, B, J, E, A, H, F, I, C, G

如何在存储结构中方便地知道结点的直接前驱或后继

对于具有n个结点的二叉树,二叉链表中空的指针域数目为 n+1

$$2n-(n-1)=n+1$$

指针域总数　已用指针域数

6. 线索二叉树*

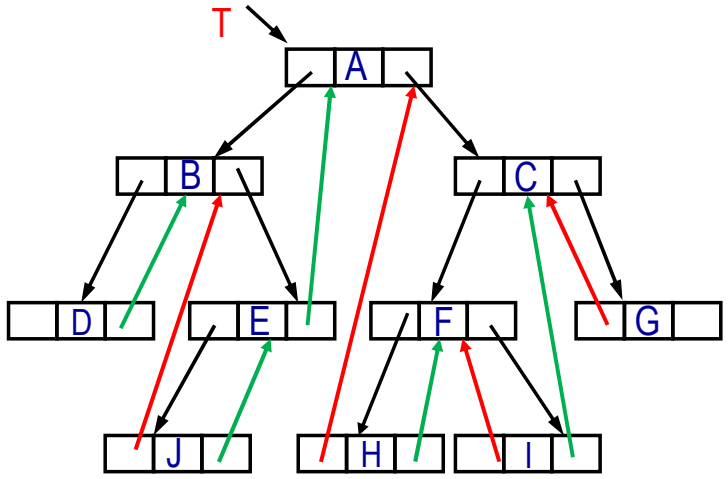
◆ 什么是线索二叉树

- ✓ 利用二叉链表中空的指针域指出结点在某种遍历序列中的直接前驱或直接后继，指向前驱和后继的指针称为**线索**，加了线索的二叉树称为线索二叉树

◆ 构造线索二叉树

- ✓ 利用链结点的**空的左指针域**存放该结点的直接前驱的地址，**空的右指针域**存放该结点直接后继的地址；而非空的指针域仍然存放结点的左孩子或右孩子的地址

中序线索二叉树



中序序列:

D, B, J, E, A, H, F, I, C, G

如何区分指针和线索?

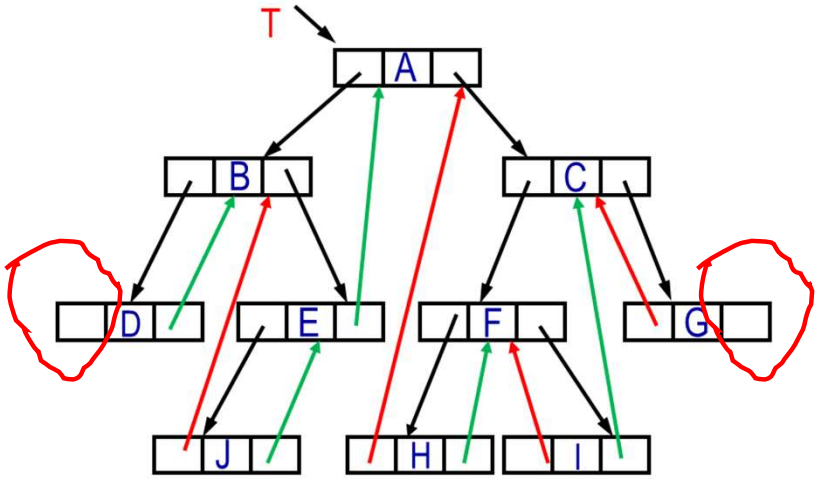
lbit	lchild	data	rchild	rbit

约定

p->lbit = { 0 表示p->lchild为指向直接前驱的线索
1 表示p->lchild为指向左孩子的指针

p->rbit = { 0 表示p->rchild为指向直接后继的线索
1 表示p->rchild为指向右孩子的指针

线索二叉树链结点类型定义



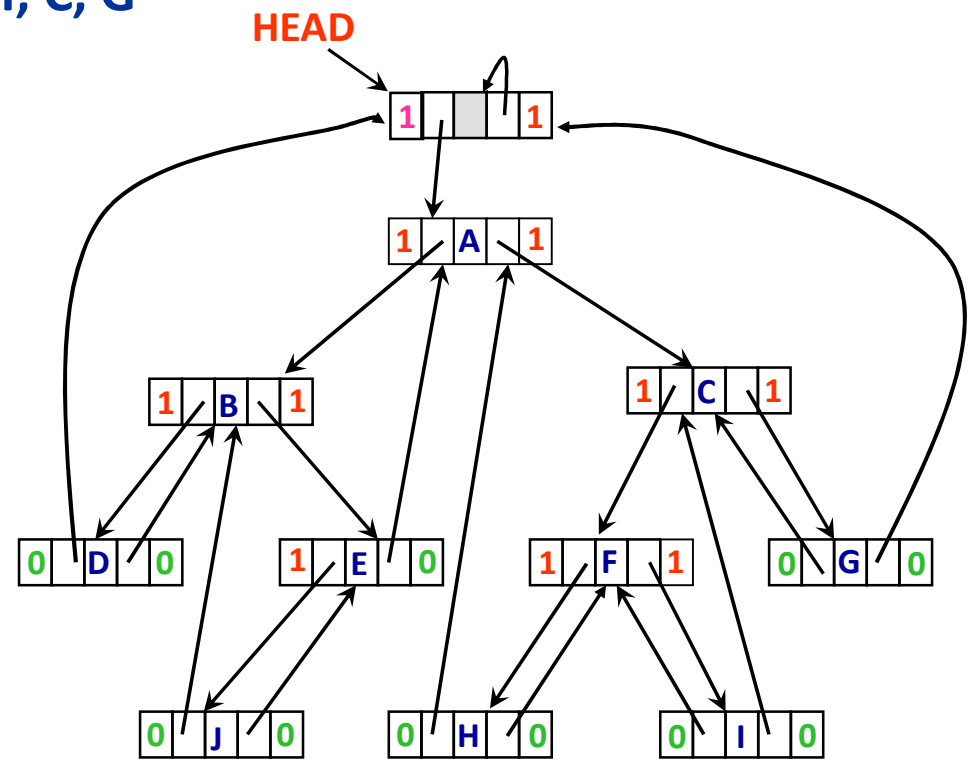

```
struct node
{
    DataType data;
    struct node *lchild, *rchild;
    char lbit, rbit;
};
typedef struct node TBTNode;
typedef struct node *TBTNodeptr;
```

中序序列:

D, B, J, E, A, H, F, I, C, G

增加头结点后完整的中序线索二叉树

中序序列：D, B, J, E, A, H, F, I, C, G



二叉树的线索化

- 通过遍历可以实现二叉树的线索化
  - ✓ 线索过程请参阅教材和PPT相关内容
- 线索化二叉树等于将一棵二叉树转变成了一个双向链表，这为二叉树结点的插入、删除和查找带来了方便
  - ✓ 在实际问题中，如果所用的二叉树需要经常遍历或查找结点时需要访问结点的前驱和后继，则采用线索二叉树结构是一个很好的选择
  - ✓ 将二叉树线索化可以实现不用栈的树深度优先遍历算法

提纲：树和二叉树

- 树的基本概念
- 树的存储结构
- 二叉树的基本概念
- 二叉树的存储结构
- 二叉树的遍历
- 线索二叉树
- **二叉查找树**
- 堆和表达式树
- 哈夫曼树

问题4.2：词频统计再讨论

- 在“线性表”一章中给出了两种解决方案

  - ✓ **基于顺序表（数组）**
    - **优点：** 单词**查找效率高**（可用折半查找）
    - **缺点：** 单词表长度需要事先定义，容易造成空间浪费或不足

      插入操作效率低（需要移动大量数据）
  - ✓ **基于链表**
    - **优点：** 空间动态管理(随问题规模动态改变)；

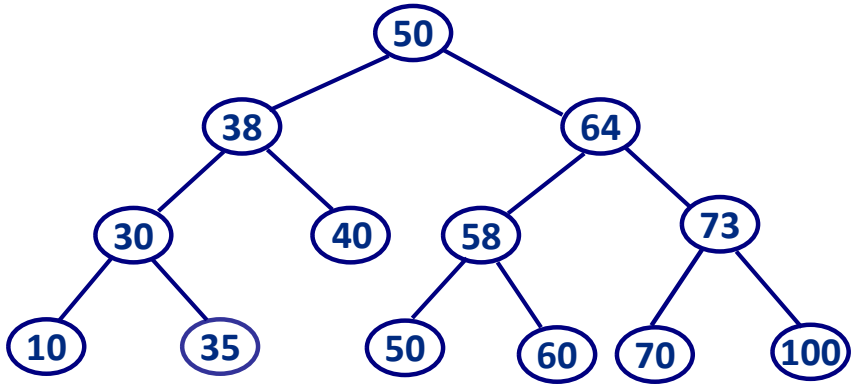
      单词插入不需要移动数据，效率高
    - **缺点:** 单词**查找效率低**（每次要从头开始查找）

有没有一种方法能兼顾两者的优点？

7. 二叉查找树（二叉搜索树、二叉排序树）

- 二叉查找树(Binary Search Tree, BST)：二叉查找树或者为空二叉树，或者为具有以下性质的二叉树:
  - ✓若根结点的左子树不空, 则左子树上所有结点的值都小于根结点的值
  - ✓若根结点的右子树不空, 则右子树上所有结点的值都大于或等于根结点的值
  - ✓每一棵子树分别也是二叉查找树（或称为二叉排序树）

一颗二叉查找树



中序序列

10, 30, 35, 38, 40, 50, 50, 58, 60, 64, 70, 73, 100

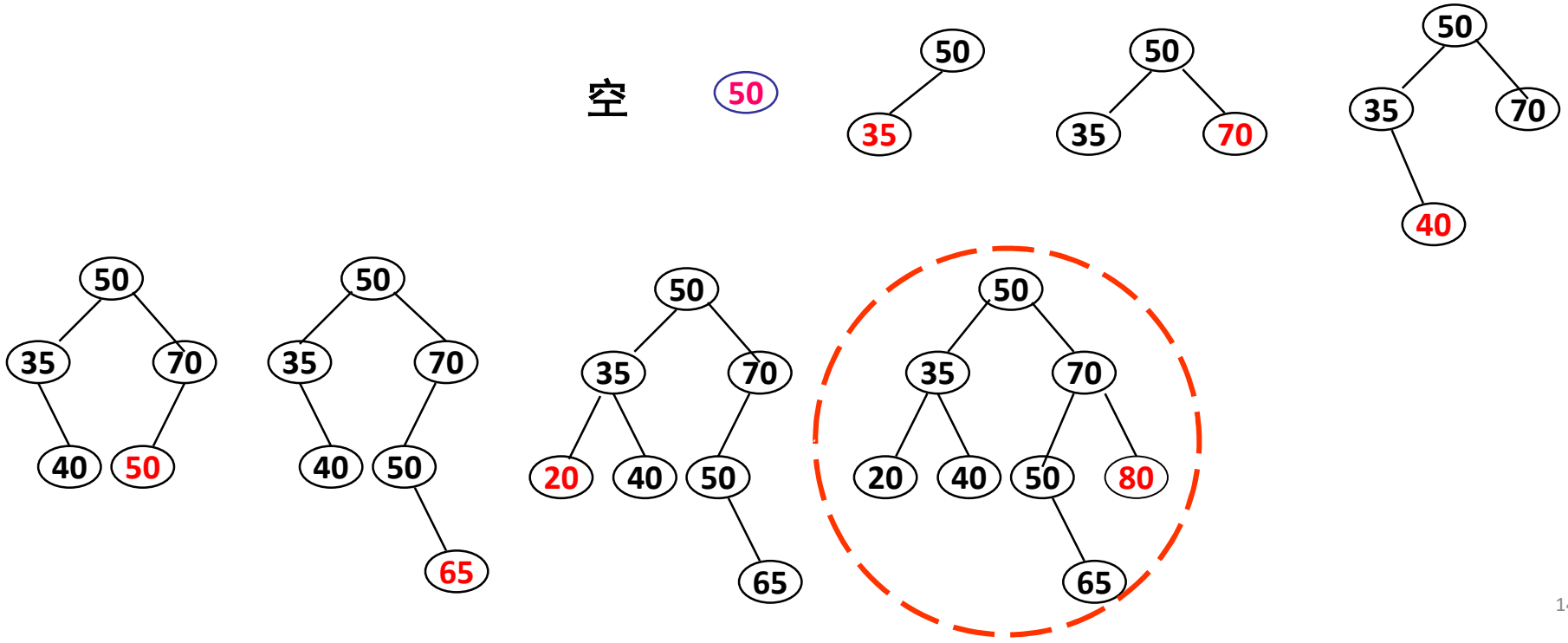
二叉查找树的建立（逐点插入法）

- 设K=(k_1, k_2, k_3, … , k_n)为具有n个数据元素的序列，从序列的第一个元素开始，依次取序列中的元素，每取一个元素k_i，按照下述原则将k_i 插入到二叉树中

1. 若二叉树为空，则k_i 作为该二叉树的根结点；
2. 若二叉树非空，则将k_i 与该二叉树的根结点的值进行比较：
   - 若k_i 小于根结点的值，则将k_i插入到根结点的左子树中；
   - 否则，将k_i 插入到根结点的右子树中。
3. 将k_i 插入到左子树或者右子树中仍然遵循上述原则**(递归)**。

二叉查找树的建立（逐点插入法）

示例： K = (50, 35, 70, 40, 50, 65, 20, 80)

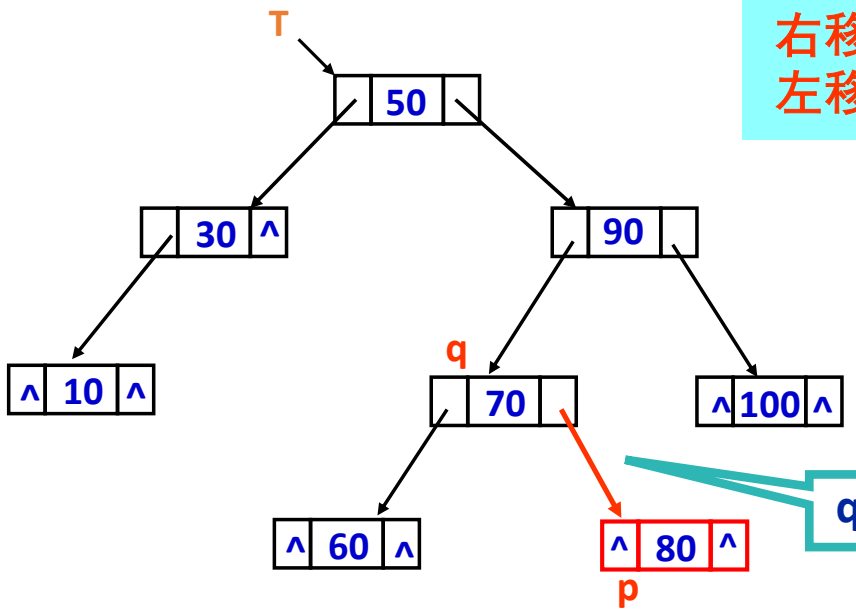


二叉查找树的建立（逐点插入法）

- 设K=(k_1, k_2, k_3, … , k_n)为具有n个数据元素的序列，从序列的第一个元素开始，依次取序列中的元素，每取一个元素k_i，按照下述原则将k_i 插入到二叉树中

1. 若二叉树为空，则k_i 作为该二叉树的根结点；
2. 若二叉树非空，则将k_i 与该二叉树的根结点的值进行比较：
   - 若k_i 小于根结点的值，则将k_i插入到根结点的左子树中；
   - 否则，将k_i 插入到根结点的右子树中。
3. 将k_i 插入到左子树或者右子树中仍然遵循上述原则**(递归)**。

二叉查找树中如何插入一个元素



右移 q=q->rchild;
左移 q=q->lchild;

插入

item=80

q->rchild=p;

二叉树采用二叉链式存储结构

算法实现

递归算法

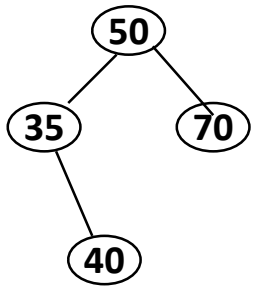
将一个数据元素item插入到根指针为root的二叉排序树中

```
#include <stdio.h>
typedef int Datatype;
struct node{
    Datatype data;
    struct node *lchild, *rchild;
};
typedef struct node BTNode, *BTNodeptr;
BTNodeptr insertBST(BTNodeptr p, Datatype item);
int main(){
    int i, item;
    BTNodeptr root = NULL;
    for (i = 0; i < 10; i++)
    { //构造一个有10个元素的BST树
        scanf("%d", &item);
        root = insertBST(root, item);
    }
    return 0;
}
```

```
BTNodeptr insertBST(BTNodeptr p, Datatype item)
{
    if (p == NULL)
    {
        p = (BTNodeptr)malloc(sizeof(BTNode));
        p->data = item;
        p->lchild = p->rchild = NULL;
    }
    else if (item < p->data)
        p->lchild = insertBST(p->lchild, item);
    else if (item > p->data)
        p->rchild = insertBST(p->rchild, item);
    else
        do-something; //树中存在该元素
    return p;
}
```

算法实现：非递归算法

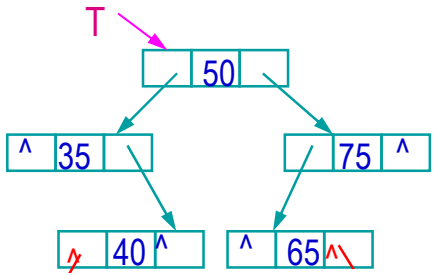
```
BTNodeptr Root = NULL; // Root是一个全局变量
void insertBST(Typedata item){
    BTNodeptr p, q;
    p = (BTNodeptr)malloc(sizeof(BTNode));
    p->data = item;
    p->lchild = NULL;
    p->rchild = NULL;
    if (Root == NULL)
        Root = p;
    else
    {
        q = Root;
        while (1){
            /* 比较值的大小 */
            /* 小于向左，大于向右 */
            if (item < q->data)  {
                if (q->lchild == NULL) {
                    q->lchild = p;
                    break;
                }
                else q = q->lchild;
            }
            else if (item > q->data) {
                if (q->rchild == NULL) {
                    q->rchild = p;
                    break;
                }
                else
                    q = q->rchild;
            }
            else
            { /* do-something */
            }
        }
    }
}
```



7.3 二叉查找树的删除

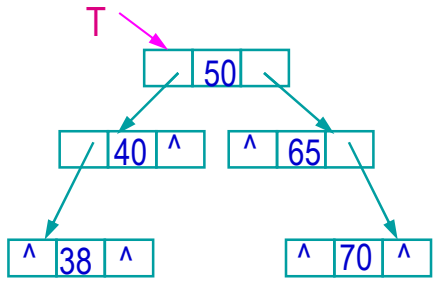
◆ 原则

1. 被删除结点为叶结点，则直接删除



2.被删除结点无左子树，则用右子树的根结点取代被删除结点，如：删除35

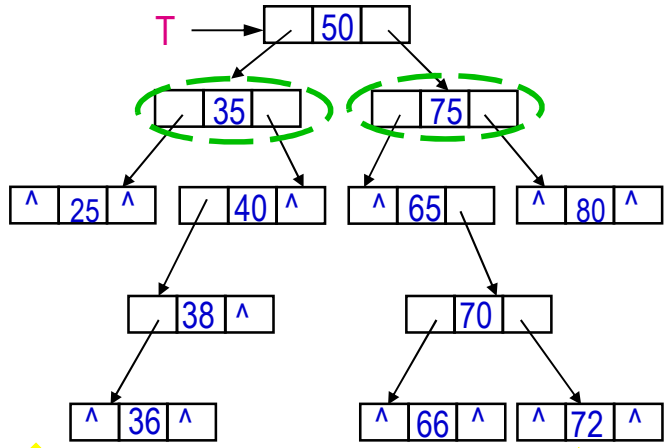
3.被删除结点无右子树，则用左子树的根结点取代被删除结点，如：删除75

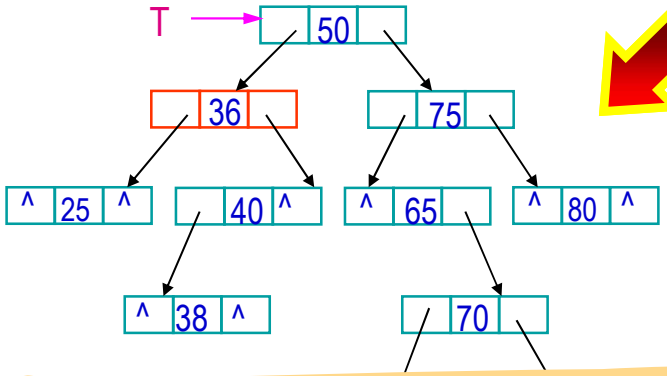


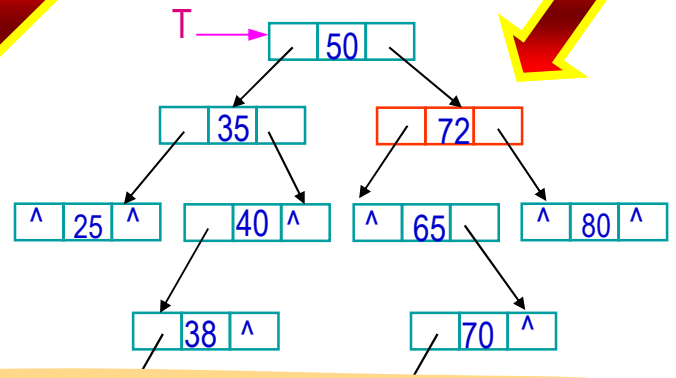
7.3 二叉查找树的删除

◆ 原则

4.被删除结点的左、右子树都存在，则用被删除结点的右子树中值最小的结点（或被删除结点的左子树中值最大的结点）取代被删除结点







延伸阅读*：请同学自学有关二叉查找树结点删除算法的C实现

7.4 二叉查找树的查找

◆ 查找过程

✓若二叉查找树为空，则查找失败，查找结束

✓若二叉查找树非空，则将被查找元素与二叉查找树的根结点的值进行比较(递归)

- 若等于根结点的值，则查找成功，结束
- 若小于根结点的值，则到根结点的左子树中重复上述查找过程
- 若大于根结点的值，则到根结点的右子树中重复上述查找过程

✓直到查找成功或者失败

算法实现

非递归算法

```
BTNodeptr searchBST(BTNodeptr t, Datatype key){
    BTNodeptr p = t;
    while (p != NULL)
    {
        if (key == p->data)
            return p; //查找成功
        if (key > p->data)
                  //将p移到右子树的根结点
            p = p->rchild;
        else      //将p移到左子树的根结点
            p = p->lchild;
    }
    return NULL; //查找失败
}
```

递归算法

```
BTNodeptr searchBST(BTNodeptr t, Datatype key)
{
    if (t != NULL)
    {
        if (key == t->data)
            return t;       //查找成功
        if (key > t->data) //查找T的右子树
            return searchBST(t->rchild, key);
        else
            return searchBST(t->lchild, key);
    } //查找T的左子树
    else
        return NULL; //查找失败
}
```

平均查找长度

- 平均查找长度ASL： 确定一个元素在树中位置所需要进行的元素间的比较次数的期望值(平均值)

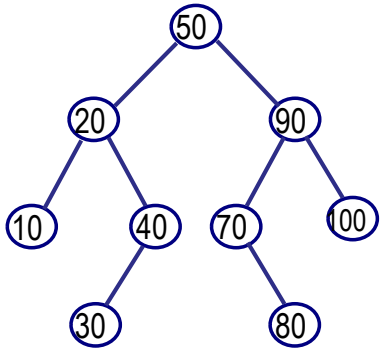
$$ASL = \sum_{i=1}^{n} p_i \, c_i$$

n 二叉树中结点的总数

p_i 表示查找第i个元素的概率；

c_i 表示查找第i个元素需要进行的元素之间的比较次数。

例



ASL = (1×1+2×2+3×4+4×2)/9

= (1+4+12+8)/9

= 25/9

查找效率与树的深度相关

如果被插入的元素序列是随机序列，或者序列的长度较小，采用“逐点插入法”建立二叉查找树可以接受。如果建立的二叉查找树中出现结点子树的深度之差较大时（即产生不平衡），就有必要采用其他方法建立二叉查找树，即建立所谓“**平衡二叉树**”。

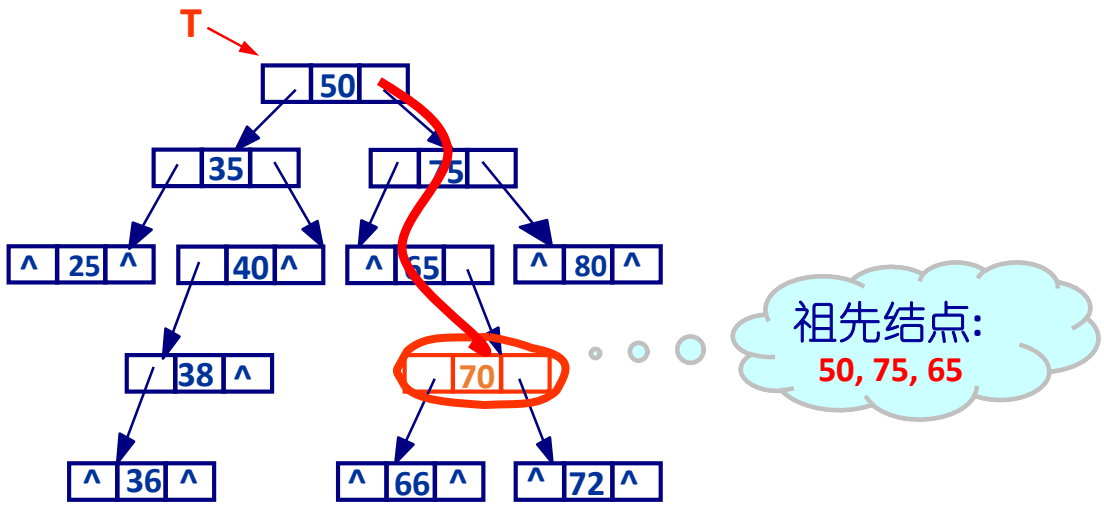
查找时间

$O(\log_2 n)$ 比较理想的情况

$O(n)$ 当出现“退化二叉树”时

二叉查找树的应用：祖先结点

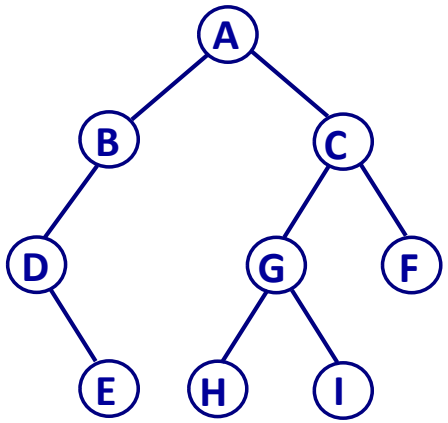
- 已知二叉查找树采用二叉链表存储结构，根结点地址为T，请写一非递归算法，打印数据信息为item的结点的所有祖先结点（设该结点存在祖先结点）
  - ✓祖先结点：从根结点到该结点的所有分支上经过的结点



寻找祖先结点：非递归算法

```
void searchBST(BTNodeptr t, DataType item){
    BTNodeptr p = t;
    while (p != NULL)    {
        if (item == p->data)
            break; /* 查找结束  */
        if (item > p->data) {
            printf("%d ", p->data);
            p = p->rchild; /* 将p 移到右子树的根结点 */
        }
        else {
            printf("%d ", p->data);
            p = p->lchild; /* 将p 移到左子树的根结点 */
        }
    }
}
```

如何获得普通二叉树的祖先结点



前序序列: A B D E C G H I F

前序遍历算法

```
void preorder(BTNodeptr t, DataType item)
{
    if (t != NULL)
    {
        push(t);
        if (item == t->data)
            //弹出栈中所有元素;
        preorder(t->lchild);
        preorder(t->rchild);
        pop();
    }
}
```

关于二叉查找树

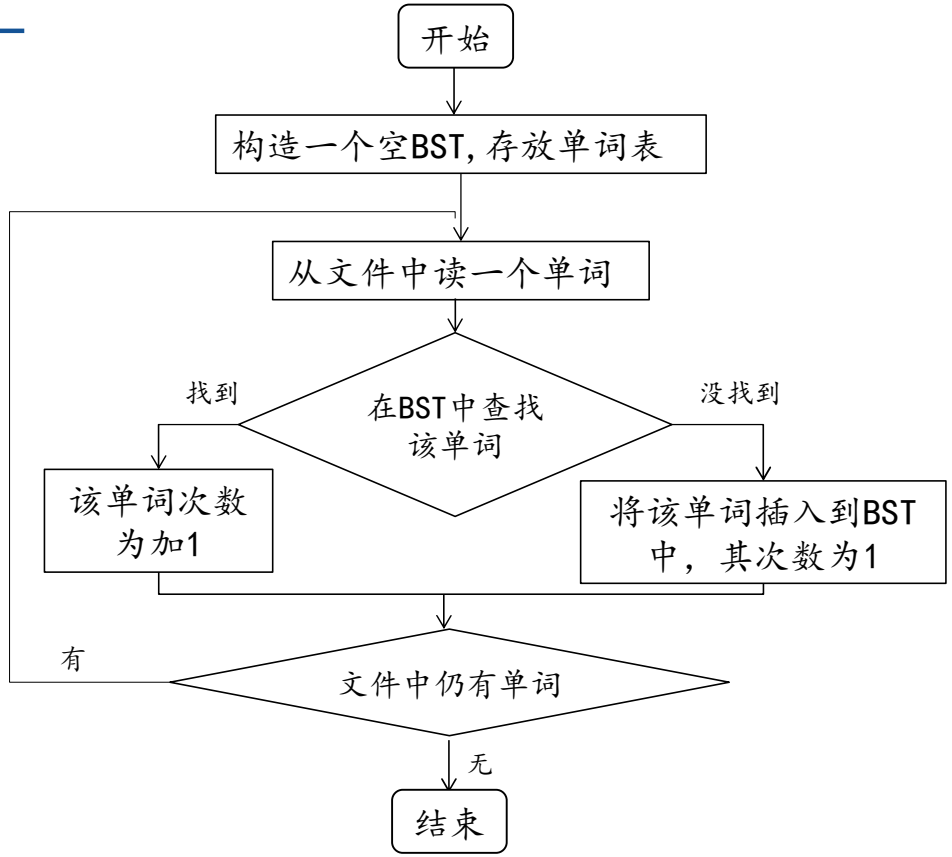
对于输入序列无序、且需要频繁进行查找、插入和删除操作的动态表（如词频统计中的单词表），如何选取数据结构和算法即能兼顾插入和删除效率，又能较高效率地实现查找？

二叉查找树是很好的选择

有序顺序表(数组)	有序链表	二叉查找树
✓ 有序顺序存储的数据能够采用如折半查找等算法，**查找效率高**	✓ 有序链表存储，数据**插入和删除操作效率高**	✓ 数据**插入和删除操作效率高**
✓有序顺序存储数据由于据需要移动数据，**插入和删除操作效率低**	✓链式存储能够动态分配空间，**空间利用率高**	✓能够动态分配空间，**空间利用率高**
✓顺序存储需要事先分配空间，**空间利用率低，同时存在溢出风险**	✓链表存储的数据查找必须从链头开始，数据**查找效率低**	✓数据**查找效率较高**（理想情况下查找效率为$O(\log_2 n)$）

问题4.2：词频统计 – 二叉查找树

- 问题：编写程序统计一个文件中每个单词的出现次数（词频统计），并按字典序输出每个单词及出现次数
- 算法分析：本问题算法很简单，基本上只有查找和插入操作



词频统计-二叉查找树的代码实现

```c
#include <stdio.h>
#include <string.h>
#define MAXWORD 32
struct tnode
{
    char word[MAXWORD];
    int count;
    struct tnode *left, *right;
}; // BST,单词树结构
//从文件中，读入一个单词
int getWord(FILE *bfp, char *w);
//查找单词，如果不存在，则插入，存在则增加计数
struct tnode *treeWords(struct tnode *p,
                        char *w);
//打印BST中的单词
void treePrint(struct tnode *p);
```

```c
int main()
{
    char filename[32], word[MAXWORD];
    FILE *bfp;
    //BST树根结点指针
    struct tnode *root = NULL;

    scanf("%s", filename);
    if ((bfp = fopen(filename, "r")) == NULL){
        //打开一个文件
        printf("%s  can't open!\n", filename);
        return -1;
    }
    while (getWord(bfp, word) != EOF)
        //从文件中读入一个单词
        root = treeWords(root, word);
    treePrint(root); //遍历输出单词树
    return 0;
}
```

```
//在P结点为根的树中插入w
struct tnode *treeWords(struct tnode *p, char *w){
    int cond;
    if (p == NULL){//p为null，新建结点作为当前子树的根
        p = (struct tnode *)malloc(sizeof(struct tnode));
        strcpy(p->word, w); p->count = 1;
        p->left = p->right = NULL;   }
    else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;//存在相同的单词，计数加1
    else if (cond < 0) //单词比根结点小，插入左子树
        p->left = treeWords(p->left, w);
    else //单词比根结点大，插入右子树
        p->right = treeWords(p->right, w);
    return p;
}

void treePrint(struct tnode *p){//中序遍历打印树中的结点
    if (p != NULL)    {
        treePrint(p->left);
        printf("%s  %4d\n", p->word, p->count);
        treePrint(p->right);
    }
}
```

词频统计-二叉查找树的代码实现 (续)

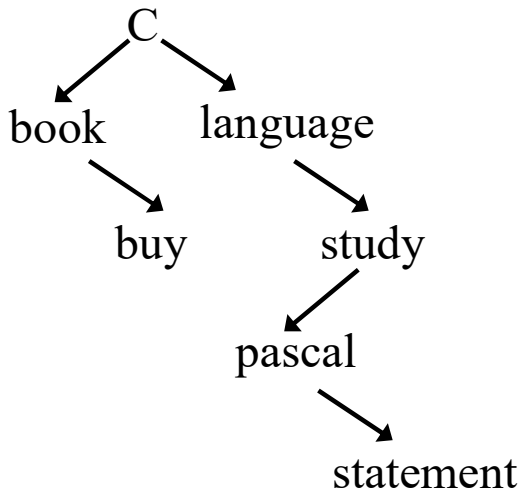
在本程序中：
在理想情况下（构成的是平衡二叉树），查找算法复杂度为$O(\log_2 n)$
插入算法复杂度为 **O(1)**

问题4.2：测试

若输入：

C language book study pascal statement buy

则生成如下树：



关于二叉查找树*

- 关于二叉查找树
  - ✓从二叉查找树的定义与构造方式不难看到，其查找某个元素的效率要比采用顺序比较（如链表）的效率要高得多
  - ✓二叉查找树特别适合于数据量大、且无序的数据，如单词词频统计（单词索引）等
- 二叉查找树是重要的一种数据结构，在实际应用中经常使用，请同学们自学下面内容：
  - ✓二叉查找树的结点的插入、删除操作
  - ✓平衡二叉树（AVL）

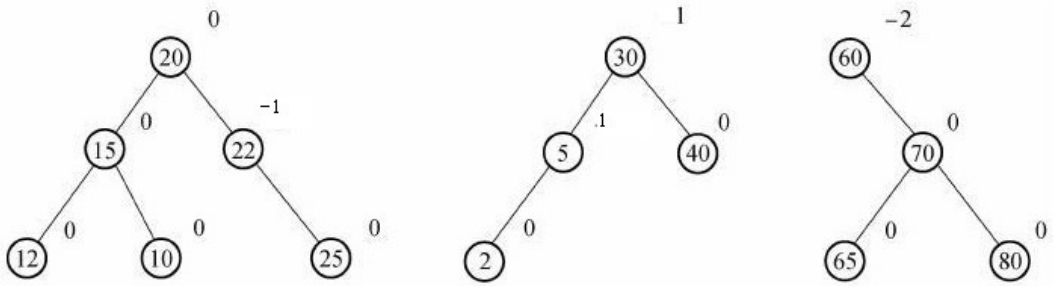
平衡二叉树（AVL）

二叉查找树的缺陷— 树的形态无法预料、随意性大。
得到的可能是一个不平衡的树，即树的深度差很大。
丧失了利用二叉树组织数据带来的好处。

◆ 平衡二叉树定义

平衡二叉树又称AVL树。它或者是一棵空树，或者是具有下列性质的二叉树：

- ✓它的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值不超过1。
- ✓若将二叉树的平衡因子定义为该结点左子树深度减去右子树深度，则平衡二叉树上所有结点的平衡因子只可能是-1、0和1。



平衡二叉树（AVL）

- 平衡二叉树存储结构描述

```
typedef struct _BSNode
{
    ElemType data;
    int bf;  //增加的平衡因子
    struct _BSNode *lchild, *rchild;
} BSTNode, *BSTNodeptr;
```

平衡二叉树（AVL）

◆ 平衡二叉树插入算法要点

假定向平衡二叉树中插入一个新结点后破坏了平衡二叉树的平衡性。

首先要找出插入新结点后失去平衡的**最小**子树根结点的指针，然后再调整这个子树中有关结点之间的链接关系，使之成为新的平衡子树。

当失去平衡的最小子树被调整为平衡子树后，原有其他所有不平衡子树无需调整，整个二叉排序树就又成为一棵平衡二叉树。

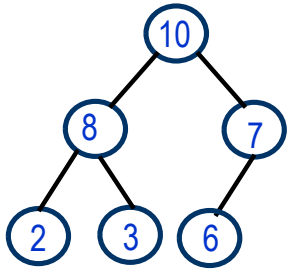
延伸阅读*：请同学自学有关平衡二叉树构造算法的C实现

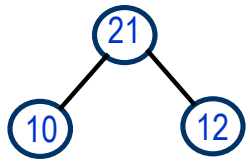
提纲：树和二叉树

- 树的基本概念
- 树的存储结构
- 二叉树的基本概念
- 二叉树的存储结构
- 二叉树的遍历
- 线索二叉树
- 二叉查找树
- **堆和表达式树**
- 哈夫曼树

8. 堆（heap)*

- 堆是一种特殊类型的二叉树，具有以下两个性质：
  （1）每个结点的值大于（或小于）等于其每个子结点的值
  （2）该树完全平衡，其最后一层的叶子都处于最左侧的位置
- 满足上面两个性质定义的是**大顶堆**（max heap）（或**小顶堆**(min heap)
- 这意味着大顶堆的根结点包含了最大的元素，小顶堆的根结点包含了最小的元素





大顶堆

由于堆是一个完全平衡树，其可以通过数组来实现：
heap[i] ≧heap[2*i]
heap[i] ≧heap[2*i+1]

堆结构

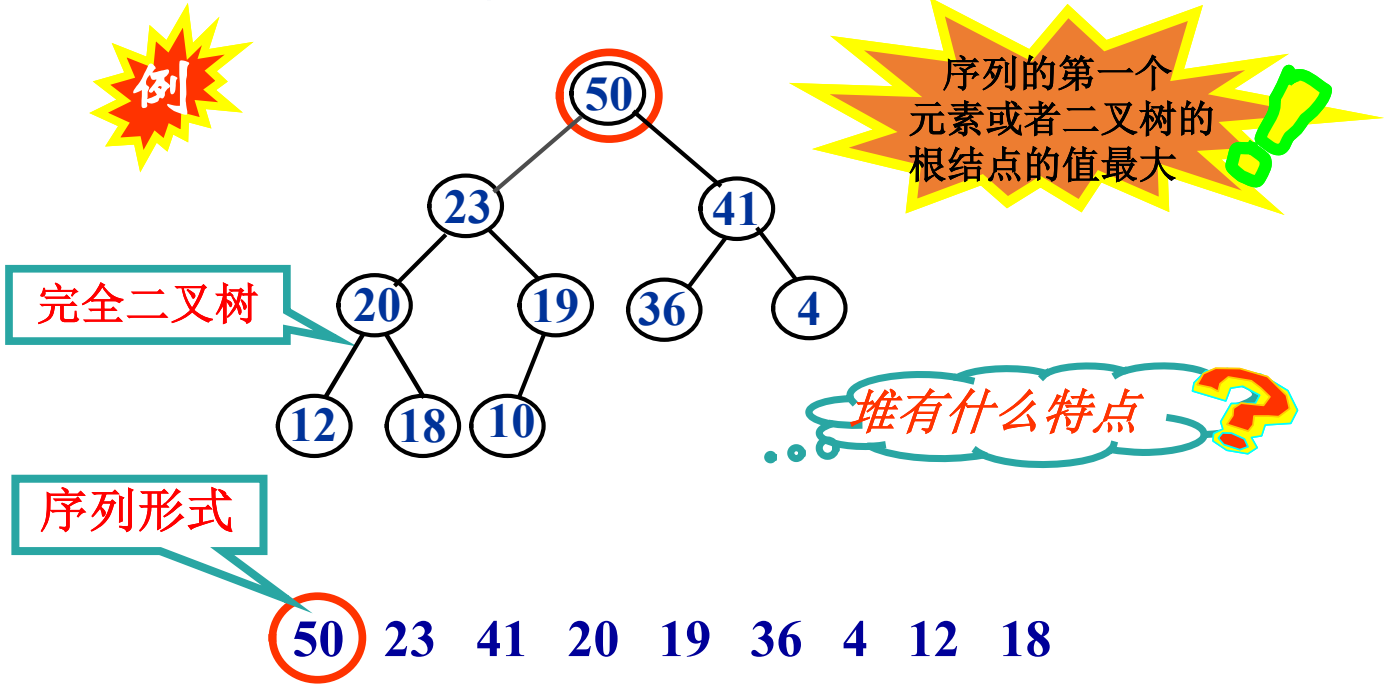
- 堆结构的最大好处是元素查找、插入和删除效率高（O(log2(n))）
- 堆的主要应用

  （1）可用来实现优先队列（Priority Queue）

  （2）用来实现一种高效排序算法-堆排序（Heap Sort），在排序一讲中详细介绍

堆结构的特点

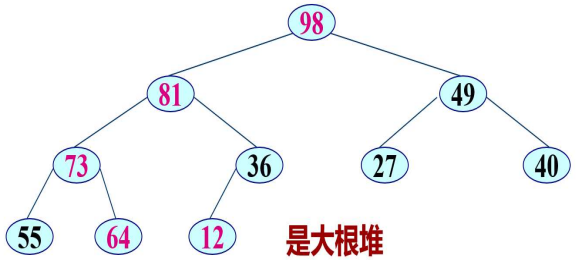
- 堆是一棵完全二叉树，二叉树中任何一个分支结点的值都大于或者等于它的孩子结点的值，并且每一棵子树也满足堆的特性



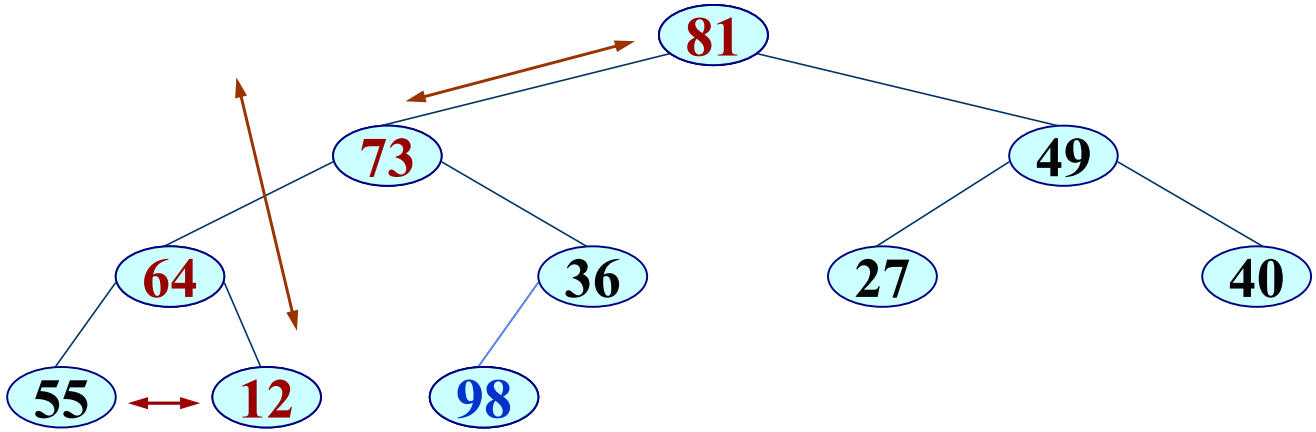
8.2 堆的基本操作*

- 堆的插入

```
//堆的插入算法要点
heapInsert(e)
    将e放在堆的末尾;
    while e 不是根 && e > parent(e)
        e 与其父结点交换
```



堆的删除举例



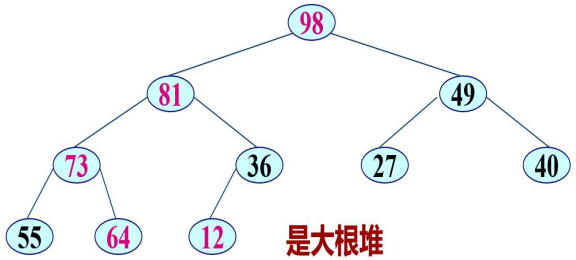
是大根堆

但在 98 和 12 进行互换之后，它就不是堆了，

8.2 堆的基本操作*

◆ 堆的删除

删除指获取堆顶元素，并从堆中删除。




```
//堆的删除算法要点
heapDelete()  //取堆顶（树根）元素
    从根结点提取元素;
    将最后一个叶结点中的元素放到要删除的元素位置；
    删除最后一个叶结点;
      //根的两个子树都是堆
      p = 根结点；
      while p 不是叶结点 && p < 它的任何子结点
            p与其较大的子结点交换
```

8.3 堆的构造*

- 堆的构造有两种方法
  - ✓ 自顶向下(John Williams提出)

    从空堆开始，按顺序向堆中添加（用headinsert函数）元素

  - ✓ 自底向上(Robert Floyd提出)

    首先从底层开始构造较小的堆，然后再重复构造较大的堆。（算法将在堆排序一节中介绍）

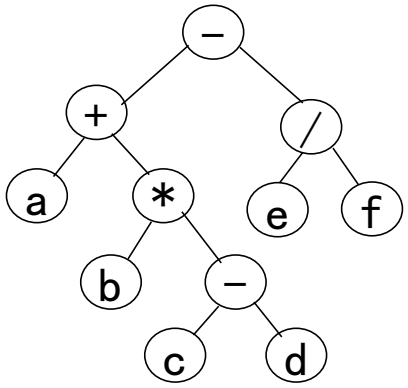
8a 表达式树（expression tree）*

- 表达式树是一种特殊类型的树，其叶结点是操作数（operand），而其它结点为运算符（operator）
  - （1）由于运算符一般都是双目的，通常情况下该树是一棵二叉树
  - （2）对于单目运算符（如++），其只有一个子结点

对于表达式：a + b * (c – d) – e / f ，其对应的表达树如下：

中序遍历：a + b * c – d – e / f　　(中缀表达式)

后序遍历：a b c d – * + e f / - (后缀表达式)



表达式树的最大好处是表达式没有括号，计算时也不用考虑运算符优先级。

主要应用：（1）编译器用来处理程序中的表达式

表达式树的构造*

- 表达式树是这样一种树，非叶结点为运算符，叶结点为操作数，对其进行遍历可计算表达式的值
- 由后缀表达式生成表达式树的方法如下：

1. 从左至右从后缀表达式中读入一个符号：
   - 如果是操作数，则建立一个单结点树并将指向该结点的指针推入栈中；（栈中元素为树结点的指针）
   - 如果是运算符，就从栈中弹出指向两棵树T1和T2的指针（T1先弹出）并形成一棵新树，树根为该运算符，它的左、右子树分别指向T2和T1，然后将新树的指针压入栈中
2. 重复步骤1，直到后缀表达式处理完

表达式树的应用*

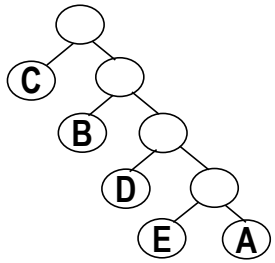
- 表达式树的优势
  - ✓ 表达式树的最大好处是表达式没有括号，计算时也不用考虑运算符优先级
- 表达式树的主要应用
  - ✓ 编译器用来处理程序中的表达式，如方便表达式的修改、分析与多次计算

提纲：树和二叉树

- 树的基本概念
- 树的存储结构
- 二叉树的基本概念
- 二叉树的存储结构
- 二叉树的遍历
- 线索二叉树
- 二叉查找树
- 堆和表达式树
- **哈夫曼树**

考察一个场景

等级	E	D	C	B	A
分数段	0~59	60~69	70~79	80~89	90~100
比例	0.05	0.15	0.40	0.30	0.10



假设输入10000个同学的成绩

```
if(a<60)
    printf("E\n");
else if(a<70)
    printf("D\n");
else if(a<80)
    printf("C\n");
else if(a<90)
    printf("B\n");
else
    printf("A\n");
```

比较31500次

```
if(a<80)
    if(a<70)
        if(a<60)
            printf("E\n");
        else   printf("D\n");
    else   printf("C\n");
else   if(a<90)
            printf("B\n");
    else
     printf("A\n");
```

比较22000次

```
if(70<=a && a<80)
    printf("C\n");
else if(80<=a && a<90)
    printf("B\n");
else if(60<=a && a<70)
    printf("D\n");
else if(a<60)
    printf("E\n");
else
    printf("A\n");
```

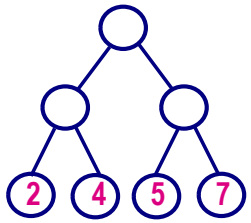
比较20500次

9. 哈夫曼树及其应用

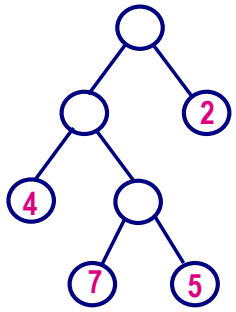
◆ 树的带权路径长度

若给具有m个叶结点的二叉树的每个叶结点赋予一个权值，则该二叉树的带权路径长度定义为$WPL=\sum_{i=1}^{m} w_i l_i$，其中，$w_i$为第i个叶结点被赋予的权值，$l_i$为第i个叶结点的路径长度。

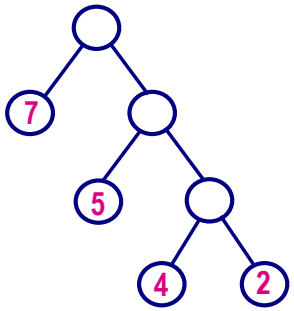
例 对于一组权值{ 7，5，2，4 }，可以有如下多种不同的带权二叉树



WPL=36



WPL=46



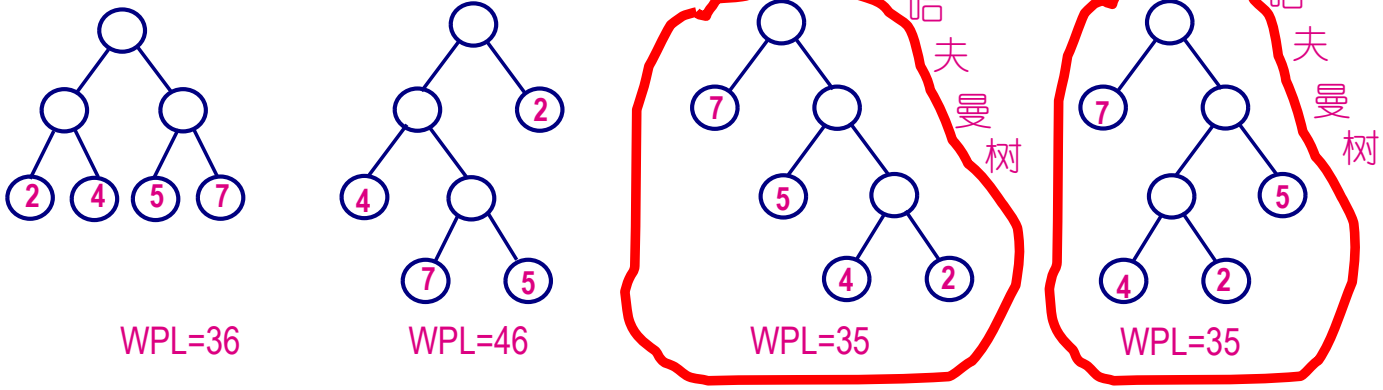
WPL=35

哈夫曼树的定义

- 给定一组权值，构造出的具有最小带权路径长度的二叉树称为哈夫曼树

例 **一组权值：**

{ 7, 5, 2, 4 }



- **哈夫曼树的特点**

（1）权值越大的叶结点离根结点越近，权值越小的叶结点离根结点越远

（这样的二叉树WPL最小）

（2）无度为1的结点

（3）不唯一

9.2 哈夫曼树的构造

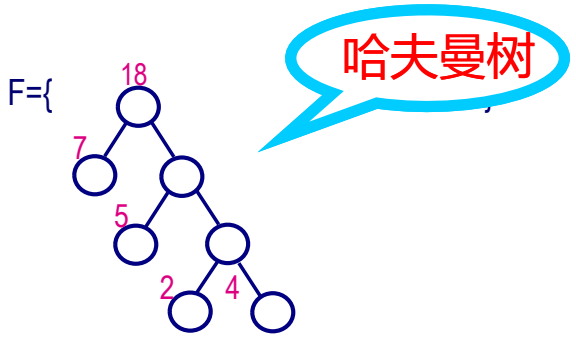
- 核心思想

1. 对于给定的权值W={w_1,w_2, …, w_m}，构造出树林F={T_1, T_2, …, T_m}，其中，T_i(1≤i≤m)为左、右子树为空，且根结点(叶结点)的权值为w_i的二叉树。

2. 将F中根结点权值最小的两棵二叉树合并成为一棵新的二叉树，即把这两棵二叉树分别作为新的二叉树的左、右子树，并令新的二叉树的根结点权值为这两棵二叉树的根结点的权值之和，将新的二叉树加入F的同时从F中 删除这两棵二叉树。

3. 重复步骤2，直到F中只有一棵二叉树。

例 W={7, 5, 2, 4}



基于哈夫曼编码的数据压缩

在信息和网络时代，数据的压缩解压是一个非常重要的技术，现有的数据压缩和解压技术很多是基于Huffman的研究之上发展而来。

9.3 一种熵编码方式

例

若假设 A=00 B=01 C=10 D=11

(编码等长)

要传送的文字 'ABACCDA'

由二进制代码组成的电文 00010010101100

若假设 A=0 B=00 C=1 D=01

(编码不等长)

要传送的文字 'ABACCDA'

由二进制代码组成的电文 0000 11010

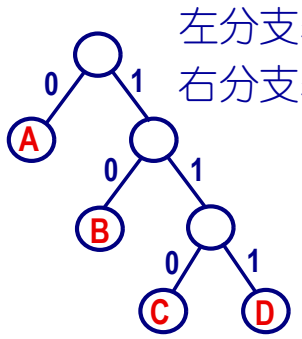
AAAA ABA BB

- 优点：电文中出现频率大的字符编码短，电文总长减少。
- 缺点：编码表示不惟一，致使电文难以翻译。
- 要求：任意一个字符的编码不能是另一个字符的编码的前缀。 前缀编码

利用二叉树设计二进制前缀编码

- 从根结点到叶结点的路径上分支字符组成的字符串作为该叶结点字符的编码。

A: 0　　B: 10　　C: 110　　D: 111



如何得到使电文总长最短的编码？

哈夫曼编码

设：电文中包含n种字符；
w_i为每种字符在电文中出现的次数；
l_i为相应的编码长度。

则：电文总长度为$\sum_{i=1}^{n} w_i l_i$

二叉树的带权路径长度

在二叉树中：
w_i为叶结点的权值；
l_i为根结点到叶结点的路径长度

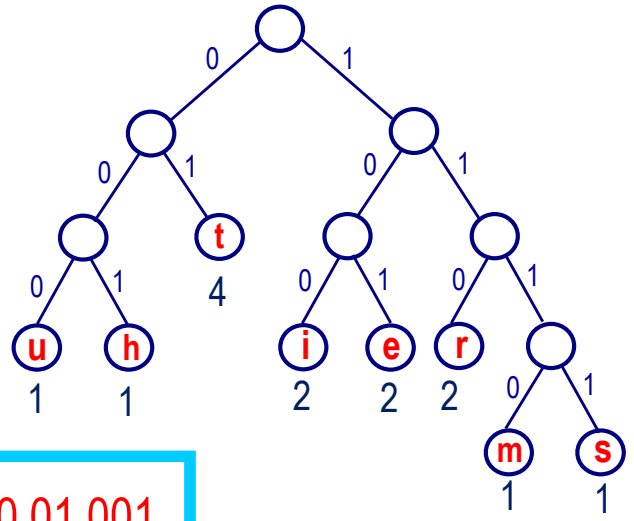
示例：哈夫曼编码

该电文以本Huffman编码传输时传输长度为：40 位bits（5字节），
而该电文以ASCII编码传输时传输长度为：14 字节(byte)

例 电文：time tries truth　　字符集：**{ t, i, m, e, r, s, u, h }**

字符在电文中出现的次数：　　**{ 4, 2, 1, 2, 2, 1, 1, 1 }**

大家来构造一下这棵哈夫曼树？



哈夫曼编码

t:	01
i:	100
m:	1110
e:	101
r:	110
s:	1111
u:	000
h:	001

01 100 1110 101 01 110 100 101 1111 01 110 000 01 001

问题4-5/6：文件压缩和解压-Huffman编码

- 【问题描述】编写程序实现利用Huffman编码对一个文件进行压缩和解压的工具hzip.exe
- Huffman压缩文件原理如下：
  1.对正文文件中字符按出现次数（即频率）进行统计
  2.依据字符频率生成相应的Huffman树（未出现的字符不生成）
  3.依据Huffman树生成相应字符的Huffman编码
  4.依据字符Huffman编码压缩文件（即按照Huffman编码依次输出源文件字符）
- 说明
  1. 只对文件中出现的字符生成Huffman码
  2. 采用ASCII码值为0的字符作为压缩文件的结束符（即可将其出现次数设为1来参与编码）
  3. 在生成Huffman树时，初始在对字符频率权重进行（由小至大）排序时，频率相同的字符ASCII编码值小的在前；新生成的权重结点插入到有序权重序列中时，出现相同权重时，插入到其后（采用稳定排序）
  4. 遍历Huffman树生成字符Huffman码时，左边为0右边为1
  5. 源文件是文本文件，字符采用ASCII编码，每个字符占8位；而采用Huffman编码后，最后输出时需要使用C语言中的位运算将字符Huffman码依次输出到每个字节中

应用：文件压缩和解压（续）

- 【输入形式】采用命令行参数

  hzip [-u] <filename.xxx>；　（注：xxx是文件扩展名）

  示例：

  >hzip myfile.txt

  - 采用Huffman编码方式压缩文件myfile.txt，并将压缩后结果存到文件myfile.hzip中。

  >hzip -u myfile.hzip

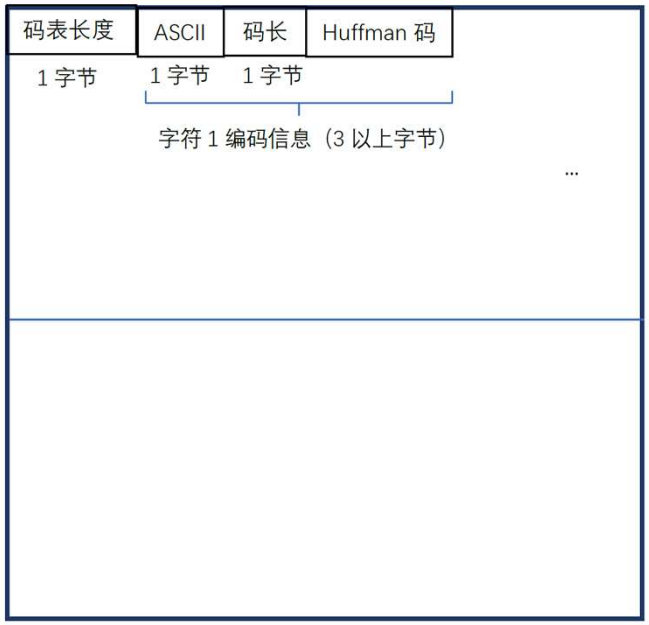
  - myfile.hzip必须是用hzip.exe压缩工具生成的压缩文件，解压后文件名为myfile.txt。当命令行有-u参数时，对当前目录下压缩文件<filename.hzip>进行解压，被解压的文件必须以.hzip为扩展名的文件，解压后的文本文件名同压缩文件，但扩展名为.txt

  ✓具有一定的参数错误处理能力，如参数个数不对、参数格式不正确等：

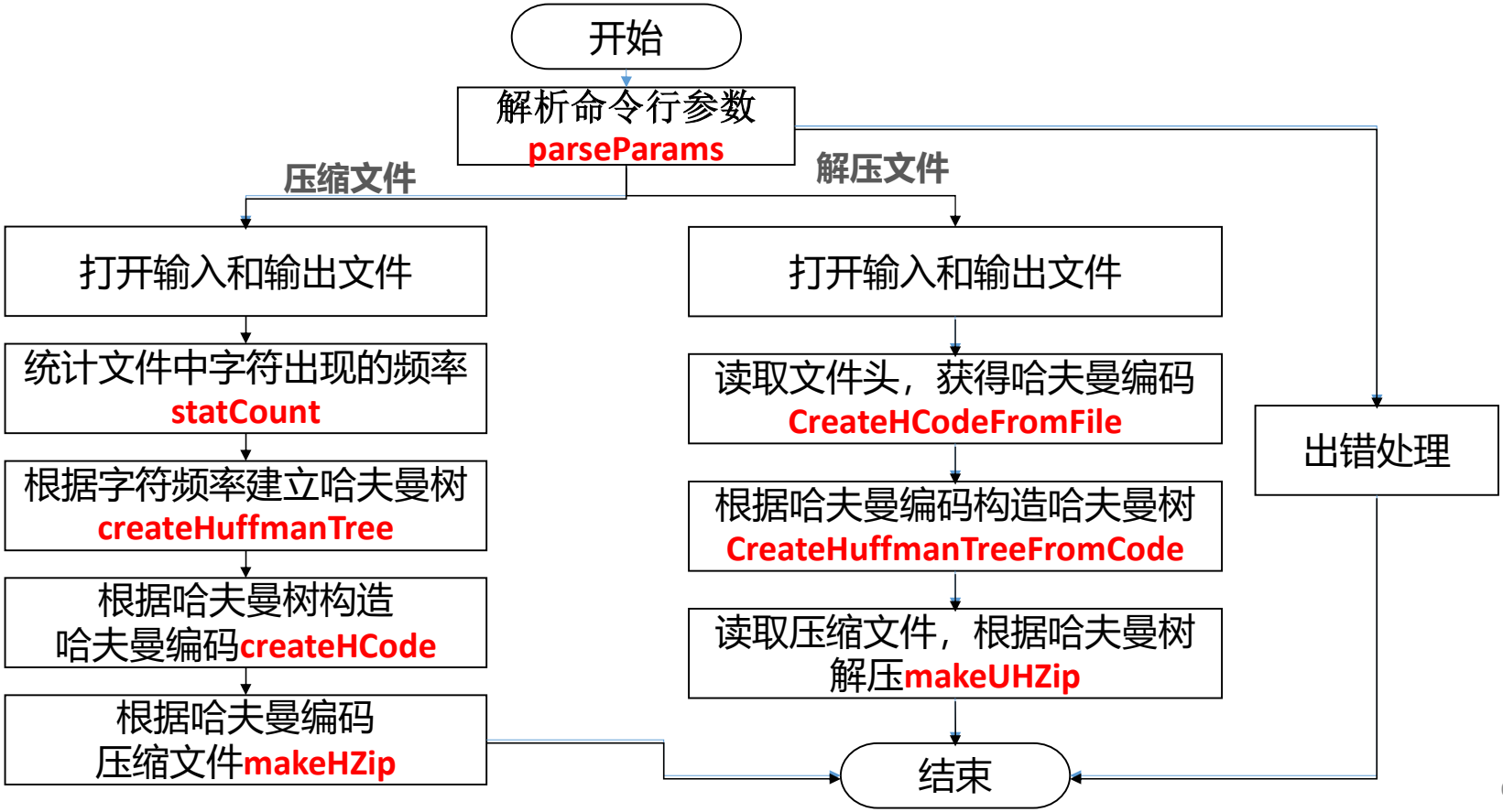
>hzip	Usage: hzip.exe [-u] <filename>
>hzip srcfile.txt objfile.hzip	Usage: hzip.exe [-u] <filename>
>hzip -h srcfile.txt	Usage: hzip.exe [-u] <filename>
>hzip -u myfile.c	File extension error!
>hzip myfile.c	File extension error!

应用：文件压缩和解压（续）

- 【输出形式】<filename.hzip>文件格式如下
  - ✓压缩文件由2部分组成，第一部分为ASCII码与Huffman码对照码表，第2部分为以Huffman编码形式的压缩后的文件。如下图所示
  - ✓文件头结构
    - 从文件头开始第1个字节为码表长度，即压缩文件时ASCII码与Huffman码对照表中实际字符个数（即字符频率为非0的字符个数，包括文件结束符）
    - 从文件头开始第2个字节为相应码表内容，每个码表项依次为每个字符的ASCII码与相应的Huffman码信息，码表项按ASCII码由小至大排列。每个码表项由3部分组成，第1部分为字符ASCII码（占一个字节）、第2部分为其对应Huffman码（二进制）长度（占一个字节）、第3部分为其对应Huffman码（占不定长位），每个码表项要占完整字节（为3个以上字节），多余位要补0



问题4.3：问题分析

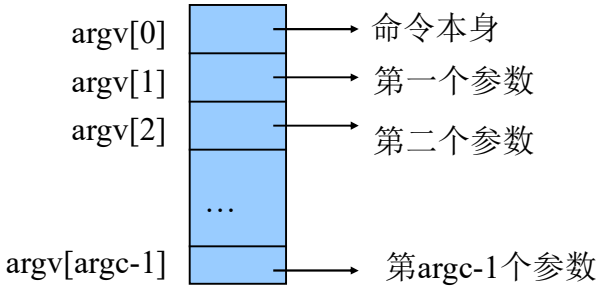

开始
解析命令行参数
parseParams
压缩文件
解压文件
打开输入和输出文件
统计文件中字符出现的频率
statCount
根据字符频率建立哈夫曼树
createHuffmanTree
根据哈夫曼树构造
哈夫曼编码createHCode
根据哈夫曼编码
压缩文件makeHZip
打开输入和输出文件
读取文件头，获得哈夫曼编码
CreateHCodeFromFile
根据哈夫曼编码构造哈夫曼树
CreateHuffmanTreeFromCode
读取压缩文件，根据哈夫曼树
解压makeUHZip
出错处理
结束

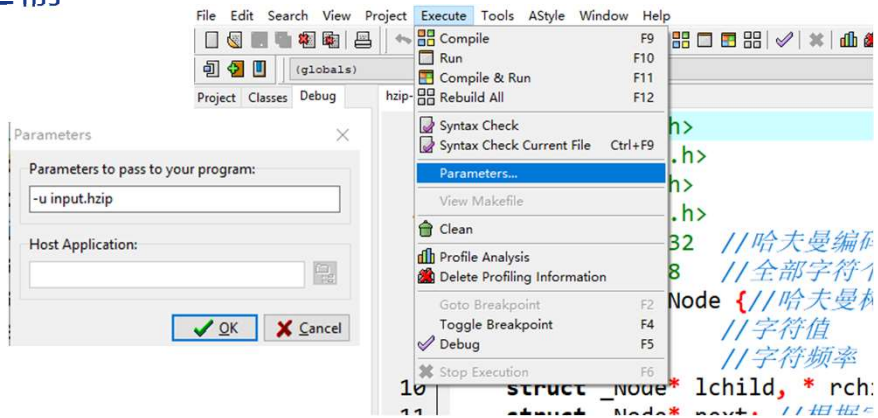
预备知识：命令行参数

- C语言中，主函数main还可以带有参数

  int main(int argc, char *argv[ ])

- 其中
  - ✓argc包含命令本身在内的参数个数
  - ✓argv指针数组，数组元素为指向各参数（包含命令本身）的指针
- 假设运行程序program时在终端键盘上输入下列命令：
  - ✓>program f1 6
  - ✓在程序program中，argc的值等于3，argv[0], argv[1]和argv[2]的内容分别是"program", "f1"和"6"





本题目的命令行参数解析

```
int parseParams(int count, char* params[], char* srcFile, char *destFile) {
    int ans;//命令行参数解析结果
    char* ext;
    if (count == 2) {//压缩文件
        strcpy(srcFile, params[1]);
        strcpy(destFile, srcFile);
        ext = getFileExt(destFile);
        if (!isequal(ext, "txt")) ans = -1;//文件扩展名错误
        else {
            ans = 1;//参数正确，进行压缩文件的操作
            strcpy(ext, "hzip");//修改目标文件的扩展名
        }
    }
```

```
//获得文件扩展名所在的位置
char* getFileExt(char* filename) {
    int len = strlen(filename);
    char* p = filename+len - 1;
    while (*p != '.' && p != filename) p--;
    return p + 1;
}
```

本题目的命令行参数解析（续）

```
else if (count == 3) {
        strcpy(srcFile, params[2]);
        strcpy(destFile, srcFile);
        if (!isequal(params[1], "-u")) ans = 0;//参数格式错误
        else{
            ext = getFileExt(destFile);
            if (!isequal(ext, "hzip")) ans = -1;//文件扩展名错误
            else {
                ans = 2;//参数正确，进行解压文件的操作
                strcpy(ext, "txt");//修改目标文件的扩展名
            }
        }
    }
    else ans = 0;//参数个数错误
    return ans;
}
```

```
//判断字符串是否相等，不区分大小写
int isequal(const char* s1,const char* s2) {
    int ans = 1, i;
    int len1 = strlen(s1);
    int len2 = strlen(s2);
    if (len1 != len2) ans = 0;
    else{
        for (i = 0; i < len1; i++) {
            if (tolower(s1[i]) != tolower(s2[i])) {
                ans = 0;
                break;
            }
        }
    }
    return ans;
}
```

预备知识：位运算

- 在实际应用中有时需要操作数据对象中的某些位（bit）
- 位运算符

~	按位取反，如~5
&	按位“与”
|	按位“或”
^	按位“异或”
>>	按位右移，低位移出，对int为算术右移（高位补符号位），对unsigned为逻辑右移（高位补0）
<<	按位左移，高位移出，低位补0

|	0	1
0	0	1
1	1	1

&	0	1
0	0	0
1	0	1

^	0	1
0	0	1
1	1	0

- 按位或（|）通常用于给字中某些位赋值
- 按位与（&）通常用于取出字中某些位的值
- 按位异或（^）通常用于图形/图像运算中

补充知识：二进制文件操作

- 文本文件按照ASCII编码方式解析和显示文件，并通过**回车**区分文件的每一行
- 计算机中文件本质上都是二进制文件，大部分文件（如word、ppt、图像、声音）都是不可见的ASCII内容，按照变量二进制编码存储和访问
- 思考
  - ✓用fprintf函数写入一个int到文件中？实际写入的是什么？
  - ✓是否可以能够把int类型的二进制编码直接写入文件？
  - ✓读取过程呢？

示例：文件的文本格式和二进制格式解析

```
int len = 0, buf[BUFSIZ];
FILE *fp;
```

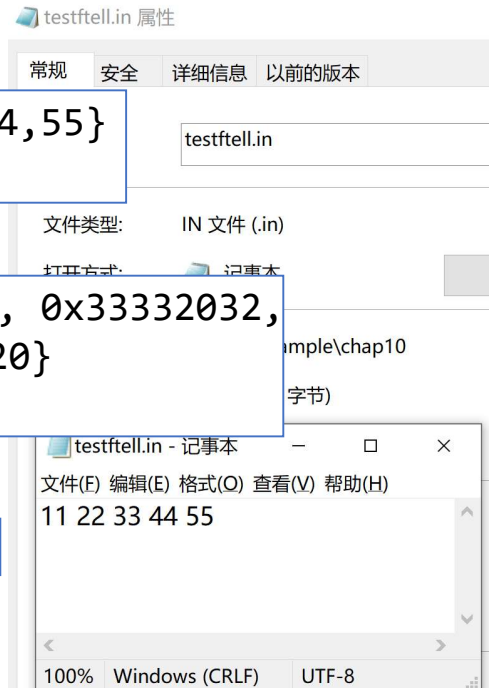
```
fp = fopen("testftell.in", "r"); //文本打开
//文本文件方式读写
while(!fscanf(fp, "%d", &buf[len])) len++;
```

```
buf {11,22,33,44,55}
len 5
```

```
fp = fopen("testftell.in", "rb");//二进制打开
//二进制方式读写
len = fread(buf, sizeof(int), BUFSIZ, fp);
```

```
buf {0x32203131, 0x33332032,
     0x20343420}
len 3
```

'1'	'1'	' '	'2'	'2'	' '	'3'	'3'	' '	'4'	'4'	' '	'5'	'5'



二进制文件读写函数

单字节读写操作: fputc与fgetc

int fputc(int c, FILE *fp)
int fgetc(FILE *fp)

- 作用：将一个字节代码**c** 写入到**fp**所指向的文件（从文件中读取一个字节码）
- 返回值：输出成功则返回所写入/读取的字节码**c**，失败则返回**EOF**

多字节读写操作: fwrite与fread

size_t fwrite(const void *buf, size_t size, size_t count, FILE *fp);
size_t fread (void *buf, size_t size, size_t count, FILE *fp);

- 作用：将buf缓冲区中的以size大小的count个数据项写入（或读取）到文件（buf）
- buf: 数据存储区（从file读入，或写进file）
- size: 以字节为单位，每个数据项的长度
- count: 读写数据项的个数
- fp: 要读写的文件指针
- 返回值：输出成功则返回所写入/读取的字节码**c**，失败则返回**EOF**

读写定位函数：多次读写、指定位置读写文件

```
int fseek(FILE *stream, long offset, int whence);
```

- 作用：将stream所指向文件的位置指针移到以**whence**所指的位置为起始点、以**offset**为位移量的位置，同时清除文件结束标志
- 返回值：定位成功则返回非0，否则返回 0

  如 fseek(stream, 0, SEEK_SET);
- 位移量**offset**表示以起始点为基准，向前或向后移动的字节数（为正表示向文件尾部的方向的移动，为负则表示向文件头部的方向移动）
- 起始点**whence**可以是：**SEEK_SET, SEEK_CUR, SEEK_END**三个符号常量，其值分别为 **0, 1, 2**，分别表示文件开始、文件当前位置、文件末尾

哈夫曼树主要数据结构

```
#define MAXSIZE 32  //哈夫曼编码最大长度
#define COUNT 128   //全部字符个数
typedef struct _Node {//哈夫曼树结点
    char ch;        //字符值
    int count;      //字符频率
    struct _Node* lchild, * rchild;//左孩子、右孩子指针
    struct _Node* next; //根据字符出现的频率排列的有序链表下一个结点地址
}Node, *PNode;

PNode root;  //哈夫曼树的根结点
PNode head; //为了便于建立哈夫曼树，增加一个带头结点的有序链表，head指向头结点
int hCodeLength = 0;//码表的长度，初始为0
//字符对应的哈夫曼表，hcode[0]为文件结束符的编码，
//其他按照ascii下标存储，如hcode['a']为字符a的哈夫曼编码
//以"110"这样的字符串格式存储哈夫曼编码
char hCode[COUNT][MAXSIZE];
char huffman[MAXSIZE];//临时生成单个字符的哈夫曼编码变量
```

几个工具函数

```c
//创建一个哈夫曼树结点
PNode createNode(char ch, int count) {
    PNode p;
    p = (PNode)malloc(sizeof(Node));
    p->ch = ch;
    p->count = count;
    p->lchild = p->rchild = p->next = NULL;
    return p;
}
```

```c
//统计词频
void statCount(FILE* src, int ccount[]){
    char ch;
    int i;
    ccount[0] = 1;//设置文件结束符标记为
    //初始化词频为0
    for (i = 1; i < COUNT; i++) ccount[i] = 0;
    while ((ch = fgetc(src)) != EOF) {//读单字符
        ccount[ch]++;   //对应词频+1
    }
    return;
}
```

```c
//将结点插入到有序链表中
void insertSortLink(PNode p) {
    PNode q, r;
    r = head;   //r指向头结点
    q = r->next;//q指向头结点后面的正式第一个数据结点
    while (q != NULL && q->count <= p->count) {//q指针指向要插入的位置
        r = q;
        q = q->next;
    }
    r->next = p;
    p->next = q;
}
```

根据词频创建哈夫曼树

```
void createHuffmanTree(int ccount[]){
    PNode p=NULL, q, r;     int i;
    //构建初始每个单词的哈夫曼结点，同时按照词频从小到大存储到链表中
    for (i = 0; i < COUNT; i++) {
        if (ccount[i] != 0) {
            p = createNode(i, ccount[i]);
            insertSortLink(p);
            hCodeLength++;
        }
    }
    while (head->next != NULL && head->next->next!=NULL) {
        q = head->next; //取出第一个结点
        r = head->next->next;//取出第二个结点
        p = createNode('\0', q->count + r->count);
        p->lchild = q;
        p->rchild = r;
        head->next = r->next;//从链表中去掉第一、第二个结点
        insertSortLink(p);  //将p重新插入链表
    }
    root = p;
}
```

根据哈夫曼树得到哈夫曼编码

```
void createHCode(PNode p, char code, int level) {
    if (level != 0) {//非根结点，得到一位编码
        huffman[level - 1] = code;
    }
    if (p->lchild == NULL && p->rchild == NULL) {//叶子结点，生成一个编码
        huffman[level] = '\0';
        strcpy(hCode[p->ch], huffman);
    }
    else {//中间结点，递归编码
        createHCode(p->lchild, '0', level + 1);//左子树
        createHCode(p->rchild, '1', level + 1);//右子树
    }
}

//主函数中的调用代码
createHCode(root, '\0', 0);
```

根据哈夫曼编码生成压缩文件

```
void makeHZip(FILE* src, FILE* zip) {
    char* pc;//用于访问哈夫曼编码中字符串的0、1位
    unsigned char hc = 0;//用于生成8位二进制位，不需要符号位
    int ch = 0, codeLen = 0;
    saveFileHead(zip); //先写入压缩文件头信息
    fseek(src, 0, SEEK_SET);//将文件读写指针移到开头
    do { ch = fgetc(src);//读取文件中的字符
        if (ch == EOF) ch = 0;//如果文件结束，则将ascii设置0,表示压缩文件结束
        pc = hCode[ch];//拿到单个字符的哈夫曼编码
        for (; *pc != '\0'; pc++) {//将"110"这样字符串转换为对应的二进制0、1位，存储到hc中
            hc = (hc << 1) | (*pc - '0');//根据当前字符'0'、'1'，修改hc的最后一位为二进制0、1
            codeLen++;//记录当前已经存储了多少位有效的二进制位
            if (codeLen == 8) {//如果存满8位，即生成一个完整字节，则写入文件
                fputc(hc, zip); //生成的8位二进制（即一个字节）写入压缩文件
                codeLen = 0;
            }   }
        if (ch == 0 && codeLen != 0) {//最后一个字符不满8位，不够一个字节
            while (codeLen++ < 8) hc = (hc << 1);//后面移入若干个0，补满8位
            fputc(hc, zip);
        }
    } while (ch);
}
```

工具函数：将哈夫曼编码保存到压缩文件头中

```
void saveFileHead(FILE* zip) {//保存文件头信息
    int i = 0, len;   unsigned char hc=0;   char* pc;   int codeLen = 0;
    fseek(zip, 0, SEEK_SET);//将文件读写指针移到开头
    fputc(hCodeLength, zip);//写入码表长度
    for (i = 0; i < COUNT; i++) {
        pc = hCode[i];      len = strlen(pc);
        if (len > 0) {//如果存在该字符的哈夫曼编码
            fputc(i, zip);//写入字符信息
            fputc(len, zip); //写入哈夫曼编码长度信息
            codeLen = 0;      hc = 0;
            for (; *pc != '\0'; pc++) {//将"110"这样字符串转换为对应的二进制0、1位，存储到hc中
                hc = (hc << 1) | (*pc - '0');//根据当前字符'0'、'1'，修改hc最后一位为二进制0、1
                codeLen++;//记录当前已经存储了多少位有效的二进制位
                if (codeLen == 8) {//如果存满8位，即生成一个完整字节，则写入文件
                    fputc(hc, zip); //生成的8位二进制（即一个字节）写入压缩文件
                    codeLen = 0;
                }
            }
            if (codeLen != 0) {//不满8位，不够一个字节
                while (codeLen++ < 8) hc = (hc << 1);//后面移入若干个0，补满8位
                fputc(hc, zip);
            } } }   }
```

解压：先从文件头部读出哈夫曼编码

```
void createHCodeFromFile(FILE* zip) {
    int ch, hc, len, mask;  int i, k, byte;
    fseek(zip, 0, SEEK_SET);  hCodeLength = fgetc(zip);//码表长度
    for (i = 0; i < COUNT; i++)//将编码表初始化为空
        hCode[i][0] = '\0';
    for (i = 0; i < hCodeLength; i++) {//依次读出编码，从压缩文件头中读出编码表
        ch = fgetc(zip);    //编码的字符
        len = fgetc(zip);   //编码的长度
        k = 0;              //记录存入码表的字符串位置
        while (len > 0) {
            byte = 0;       //一次写入8位，写满后充值
            hc = fgetc(zip);//读出编码字节
            mask = 0x80;    //掩码，用于获取某位是0还是1
            while (len > 0 && byte < 8) {
                hCode[ch][k] = (hc & mask) ? '1' : '0';//根据某位0、1，生成字符串0、1
                len--;      //已经处理的编码长度-1
                byte++;     //已经写入的长度+1
                k++;        //下标位置+1
                mask >>= 1; //取下一位编码
            }   }
        hCode[ch][k] = '\0';//补\0
    } }
```

根据哈夫曼编码构造哈夫曼树（用于解压）

```
void createHuffmanTreeFromCode() {//根据哈夫曼表，创建哈夫曼树
    int i, j, len;    PNode p;
    root = createNode('\0', 0); //创建根结点
    for (i = 0; i < COUNT; i++) {
        p = root;
        len = strlen(hCode[i]);//依据编码的长度构造树
        for (j = 0; j < len; j++) {
            if (hCode[i][j] == '0') {//进入左子树
                if (p->lchild == NULL) //如果左子树空，则创建左子树
                    p->lchild = createNode('\0', 0);
                p = p->lchild;
            }
            else {//进入右子树
                if (p->rchild == NULL) //如果右子树空，则创建右子树
                    p->rchild = createNode('\0', 0);
                p = p->rchild;
            }
        }
        p->ch = i;//最后p指向叶子结点，设置对应的字符编码
    }
}
```

根据哈夫曼树解压文件

```
void makeUHZip(FILE* zip, FILE* obj) {
    PNode p;    int ch, mask;    int i;    char org;
    p = root;
    fseek(obj, 0, SEEK_SET);
    while ((ch = fgetc(zip)) != EOF) {//读出一个字节的编码
        mask = 0x80; //掩码，用于判断某个二进制位是否0或1
        for (i = 0; i < 8; i++) {
            if ((ch & mask) == 0) //当前位为0
                p = p->lchild;
            else //当前位为1
                p = p->rchild;
            if (p->lchild == NULL && p->rchild == NULL) {//叶子结点，表明找到了字符
                org = p->ch;
                fputc(org, obj);
                p = root;
            }
            mask >>= 1;
        }
    }
}
```

主函数：解析命令行参数

```
int main(int argc, char* argv[]) {
    FILE* src;//源文件
    FILE* obj;//目标文件
    char srcFileName[50];    char destFileName[50];
    int op;    int ccount[COUNT];
    root = NULL;
    op = parseParams(argc, argv, srcFileName, destFileName);
    if (op == 0) {
        printf("Usage: hzip.exe [-u] <filename>\n");
    }
    else if (op == -1) {
        printf("File extension error!\n");
    }
```

主函数：处理压缩操作

```
    else if (op == 1) {//压缩文件
        head = createNode('\0', 0);//创建一个空的头结点
        src = fopen(srcFileName, "r");//打开文件
        if (src == NULL) {//打开失败，报错
            perror("Can't open the file!");
            exit(-1);
        }
        obj = fopen(destFileName, "wb");//打开文件
        if (obj == NULL) {//打开失败，报错
            perror("Can't open the file!");
            exit(-1);
        }
        statCount(src, ccount);
        createHuffmanTree(ccount);
        createHCode(root, '\0', 0);
        makeHZip(src, obj);
        fclose(src);
        fclose(obj);
    }
```

主函数：处理解压操作

```
    else if (op == 2) {//解压文件
        src = fopen(srcFileName, "rb");//打开文件
        if (src == NULL) {//打开失败，报错
            perror("Can't open the file!");
            exit(-1);
        }
        obj = fopen(destFileName, "w");//打开文件
        if (obj == NULL) {//打开失败，报错
            perror("Can't open the file!");
            exit(-1);
        }
        createHCodeFromFile(src);    //读出文件头，得到哈夫曼编码
        createHuffmanTreeFromCode();//根据哈夫曼编码构造哈夫曼树
        makeUHZip(src, obj);     //生成解压文件
        fclose(src);
        fclose(obj);
    }
}
```

单选题 1分

设置

用n个权值构造出来的哈夫曼树，共有多少个结点?

A n+1

B 2n-1

C 2n

D 2n+1

提交

81

树和二叉树总结

- **树基本概念及存储**
- **二叉树**
  - ✓ 二叉树的基本概念和性质
  - ✓ 二叉树的存储
- **二叉树的遍历**
  - ✓ 前序、中序和后续遍历算法
  - ✓ 递归算法的非递归设计
  - ✓ 层次遍历算法
  - ✓ 遍历算法的应用
  - ✓ （多叉）树的遍历
- 线索二叉树*
- **二叉查找树**
  - ✓ 二叉查找树的建立、删除
  - ✓ 二叉查找树的应用
- 平衡二叉树*
- 堆*
- 表达式树*
- **哈夫曼树**
  - ✓ 哈夫曼树的构造
  - ✓ 哈夫曼编码