数据结构

课程要求

① 基本数据结构及实现：线性表、链表、栈、队列
② 高级数据结构：树、二叉树、搜索树、图，以及散列表等数据结构；
③ 检索、排序、分类、查找算法：冒泡排序、插入排序、交换排序、快排序、归并排序、堆排序，以及折半查找等算法；

第一章 绪论

1 数据结构基本概念

1.1 基本概念

数据：描述客观事物的数字字符以及一切能输入到计算机中，并且能被计算机程序处理的符号的集合

数据元素：表示一个事务的一组数据，是数据的基本单位，又称结点，在计算机程序中通常作为一个整体进行处理。一个数据元素由若干　　个数据项组成

数据对象：具有相同性质的数据元素组成的集合。

数据结构：数据元素之间的关系，数据的组织形式

1.2 研究内容

逻辑结构：数据元素之间的客观联系

存储结构：研究具有某种逻辑关系的数据在计算机存储器内的存储方式

算法：研究如何在数据的各种结构的基础上对数据实施一系列有效的基本操作

数据结构（逻辑结构）

计算机处理的数据元素的组织形式及其相互间的关系，记为：

```
Data_Structure=(D,R)
```

D是数据元素的有限集，R是所有数据元素之间关系的有限集合，有四种基本数据结构

1，集合结构：数据元素间除了同属于一个集合外，无其他关系

```
SET=(D,R)
D={1,2,3}
R={}
```

2，线性结构

```
LINEARITY=(D,R)
D={1,2,3,4}
R={<1,3>,<3.2>,<2,4>}
```

数据元素之间的关系1：1

1→3→2→4

除首尾，每个数据元素都有一个前趋和后继

3，树结构

```
TREE=(D,R)
D={1,2,3,4,5}
R={<1,2>,<1,3>,<3.4>,<3,5>}
```

数据之间的联系1：N

4，图结构

```
GRAPH=(D,R)
D={1,2,3,4}
R={<1,2>,<1,3>,<4,1>,<4,2>,<4,3>}
```

数据之间的联系 M：N

存储结构（物理结构）

数据在计算机中的表示称为数据的存储结构，有四种基本的存储方法

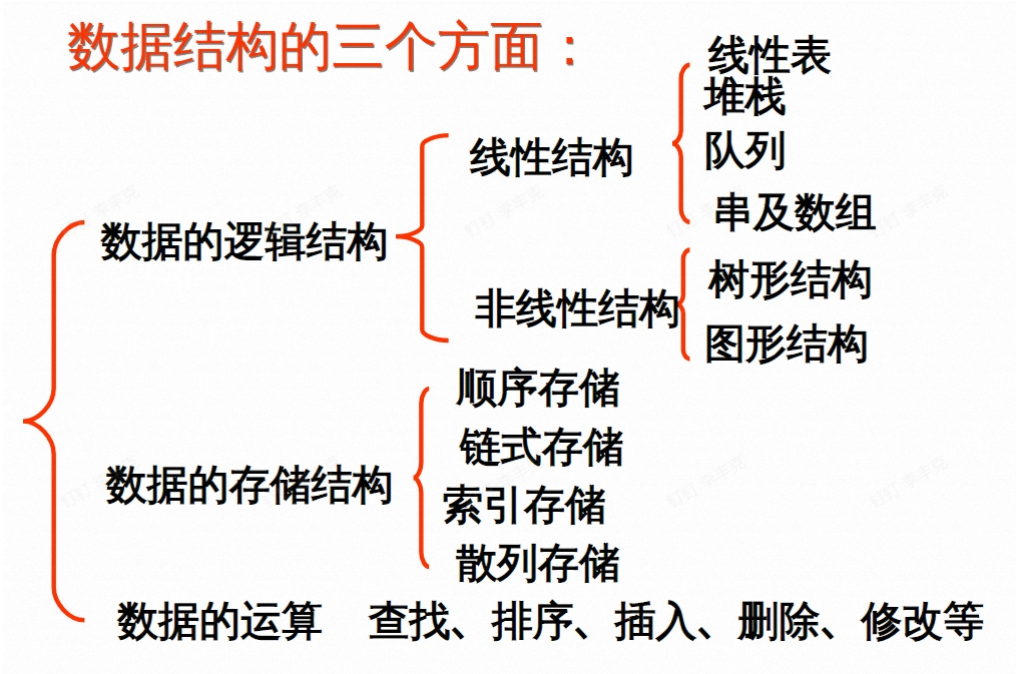
1，顺序存储结构：用数据元素在存储器中的相对位置来表示数据元素之间的逻辑关系

2，链式存储结构：在每一个数据元素中增加一个存放地址的指针，用此指针来表示数据元素之间的逻辑关系

3，索引存储方法：

4，散列存储方式

一个逻辑结构可以结合任一种或多种存储结构



1.3 算法和算法分析

算法是用来解决某个问题的指令的集合

算法性质

输入性 有0或者多个输入

输出性 有一个或者多个输出，或者处理结果

确定性 每步都是确切有意义的

有穷性 算法应该在执行有穷步后结束，整个指令序列在有限时间内完成

可行性 算法的每一步都应能够被有效的执行，并得到确定的结果

算法设计目的

正确性　可读性　健壮性　高时间效率　高空间效率

算法分析

时间复杂度T(n)　空间复杂度S(n)　其他

时间复杂度：若有一辅助函数O(f(n))，当n趋于无穷大，T(n)/f(n)为一不等于0的实数，T(n)=O(f(n))，称O(f(n))为时间复杂度，

用于描述程序中语句的执行次数

计算规则：

加法：

$$T_1(n)+T_2(n)=O(max(f_1(n),f_2(n)))$$

乘法：

$$T_1(n)\times T_2(n)=O(max(f_1(n)\times f_2(n)))$$

时间复杂度按数量级递增为：

常数阶	对数阶	线性阶	线性对数阶	平方阶	立方阶	...	K次方阶	指数阶
O（1）	O(log2n)	O(n)	O(nlog2n)	O(n^2)	O(n^#)		O(n^K)	O(2^m)

例题：

```
for(i=1,k=0;i<5;i++){
k+=i;
}
```

调用频度4，时间复杂度O(1)

```
for(i=1;i<=n;i++){
x++;
s+=x;
}
```

调用频度n+2n，时间复杂度O(n)

```
for(i=1;i<=n;i++){
for(j=1;j<=n;j++){
x++;
s+=x;
}
}
```

调用频度2n+2n^2，时间复杂度O(n^2)

```
x=n;
y=0;
while(x>(y+1)*(y+1)){
y++;
}
```

调用频度sqrt(n)-1，时间复杂度O(sqrt(n))

第二章 基本数据结构

2.1线性表

2.1.1 基本概念

逻辑结构是线性结构

n个数据元素的有限序列，记作(a0,a1,a2,...,an-1)

数据类型有任意性和一致性

除首尾，每个结点都有唯一一个前驱和后继

相邻数据元素之间存在序偶关系，含义为顺序和配对

```
ADT List=(
D={ai|i=1,2,...,n,n>=0}
R={<ai-1,ai>|i=2,...,n}
)
```

基本操作

1，初始化操作

```
InitList(&L)#创建新列表
```

2，结构销毁

```
DestoryList(&L)#销毁线性表
```

3，引用形操作，不改变原表

```
Empty(L)#判断表是否为空
Length(L)#求表长度
Prior(L,x,&pre)#求前驱
Next(L,x,&next)#求后继
Get(L,i)#获得某个数据元素
Locate(L,x)#确定某元素在表中的位置
```

4，加工形操作，改变原表

```
Clear(L)#线性表置空
PutElem(&L,i,x)#改变数据元素的值
```

线性表常用的两种存储结构：

顺式存储结构（顺序表）

链式存储结构（链表）

任一种存储结构，不仅要存储数据对象，也要存储数据对象之间的逻辑关系，并且在数据操作过程中不能破坏这种逻辑关系

2.1.2 顺序表

线性表的顺序存储结构：将线性表中的元素相继存放在一个连续的存储空间，数组

元素存储位置体现逻辑关系

一维数组实现顺序表，需要预先定义数组大小，可以任意存取（直接指定下标存储）

顺序表操作

顺序表定义

```
#define MaxSize
Type SeqList[MaxSize];
int last; \\最后元素的下标
```

元素查找，从前到后，顺序查找

```
Locate(Type x,Type &data){
int i=0;
while(i<=last&&data[i]!=x){
i++;
}
if(i>last) return -1;
else return i;
}
```

插入函数

```
Insert(int i,Type x,Type &data){
if(i<0||i>last+1) return 0;
else{
last++;
for(j=last;j>i;j--) data[j]=data[j-1];\\最后一位直接丢了？
data[i]=x;
return 1;
}
}
```

删除函数

```
Delect(int i,Type &data){
if(i<0||i>last||last<0) return 0;
else{
for(int j=i+1;j<=last;j++) data[j-1]=data[j];
last--;
return 1;
}
}
```

顺序表特点

随机存取，数据元素寻址时间确定

元素需设置最大个数，如果设置不当会造成数据溢出或浪费

插入或删除操作需要耗费较多时间移动数据元素，其时间复杂度为O(n)，n为线性表长度

2.1.3 链表

特点：用一组地址任意的存储单元存放线性表中的数据元素，用指针反映数据元素之间的线性关系，以“结点的序列"来表示线性表。

元素=指针（指示后继元素的存储位置）+结点（表示数据元素）

分为：单链表，双向链表，循环链表，循环双向链表

单链表

链表首结点作为链表的存储地址，称为头指针（head），用一个指针指向尾结点（last）

尾结点里面是空指针

基本操作

节点定义

```
Typedef struct LNode{
    Type data;
    struct LNode *next;
}LNode *head;
```

单链表查找(i是第i个)

```
int GetElem(int i,Type &e ){
    LNode* p=head;
    int j=0;
    while(p!=NULL&&j<i-1){
    p=p->next;
    j++;
    }
    if(p==NULL||p->next==NULL||i<0) return 0;
    e=p->next->data;
    return 1;
}
```

插入(在第i个结点前插入x)

```
int Insert(const int x,const int i){
    LNode*p=head,*q;
    int j=0;
    while(p!=NULL&&j<i-1){
    p=p->next;
    j++;
    }
    if(p==NULL&&head!=NULL||i<0) return 0;
    LNode *newcode;
    newcode=(LNode*)malloc(sizeof(LNode));
    newcode->data=x;
    if(head=NULL||i=0){
    newcode->next=head;
    head=newcode;
    }
    else{
    newcode->next=p->next;
    p->next=newcode;
    }
}
```

删除第i个结点

```
int Remove(int i){
    LNode *p=head;
    int k;
    while(p!=NULL&&k<i-1){
    p=p->next;
    k++;
    }
    if(p==NULL||p->==NULL||i<0) return 0;
    if(i==0){
    q=head;
    p=head->next;
    q->next=NULL;
    }
    else{
    q=p->next;
    p->next=q->next;
    q->next=NULL;
```

```
    }
    k=q->data;
    free(k);
    retutn k;
}
```

带表头结点的单链表

在头结点前加一个表头结点，表头结点的指针指向链表头指针

令指向表头结点的指针作为链表头指针head;

表头结点的特点与作用

表头结点与其它结点在结构上完全相同。

表头结点位于表的最前端，数据域通常不带数据，仅标志表头；表头结点指针域指向链表的首元素（首结点）。

设置表头结点的目的是简化链表操作，例如在进行结点插入和删除操作时无需额外判断是否在表头位置进行，可将不同情形下的处理统一起来。

表头结点的插入操作

```
newnode->next=p->link;
if(p->next==NULL) last=newnode;
p->next=newcode;
```

表头结点的删除操作

```
q=p->next;
p->next=q->next
delect q;
if(p->next==NULL) last=p;
```

一个链表类应存在的操作：

构造函数，析构函数，置空函数，求链表长度函数，查找函数，取值函数，结点插入函数，结点删除函数

链表特点

数据元素（结点）由数据域和指针域共同构成；

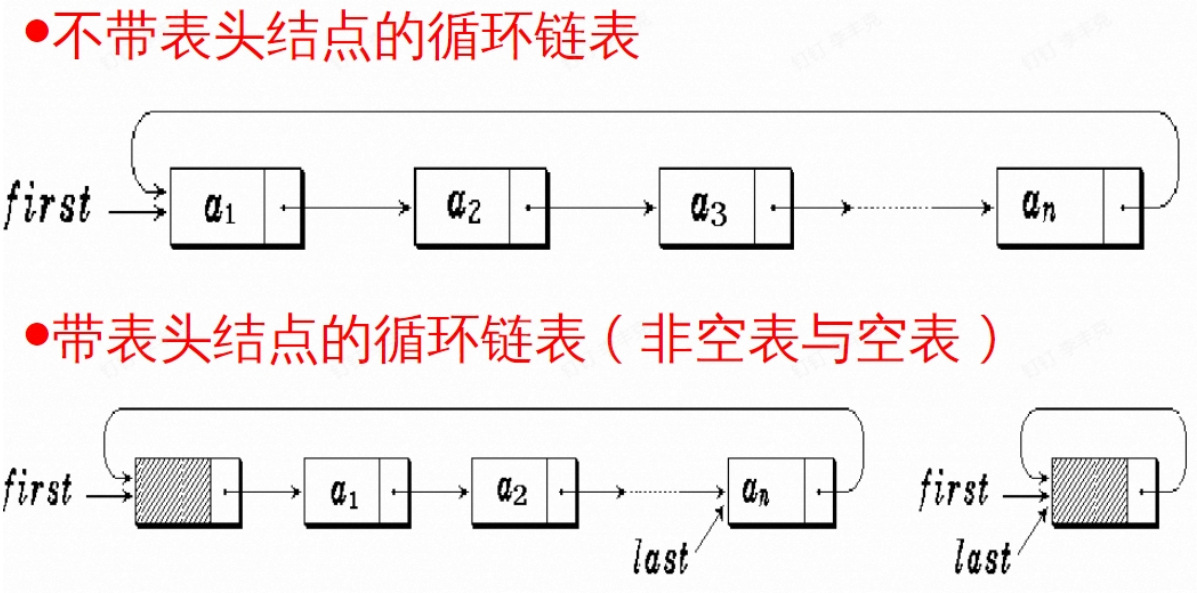
链表属于顺序存取结构，数据元素的搜索需按指针方向逐个结点顺序进行，寻址时间取决于被搜寻结点在表中的位置；

在主存允许的情况下，链表可以按需扩充，不存在空间浪费或空间溢出问题；

插入或删除操作只需修改相关结点的指针，操作简单。

循环链表

循环链表的尾结点的next指针指向头结点（或者表头结点）



循环链表只要知道表中某一结点的地址就可以找到所有其他结点

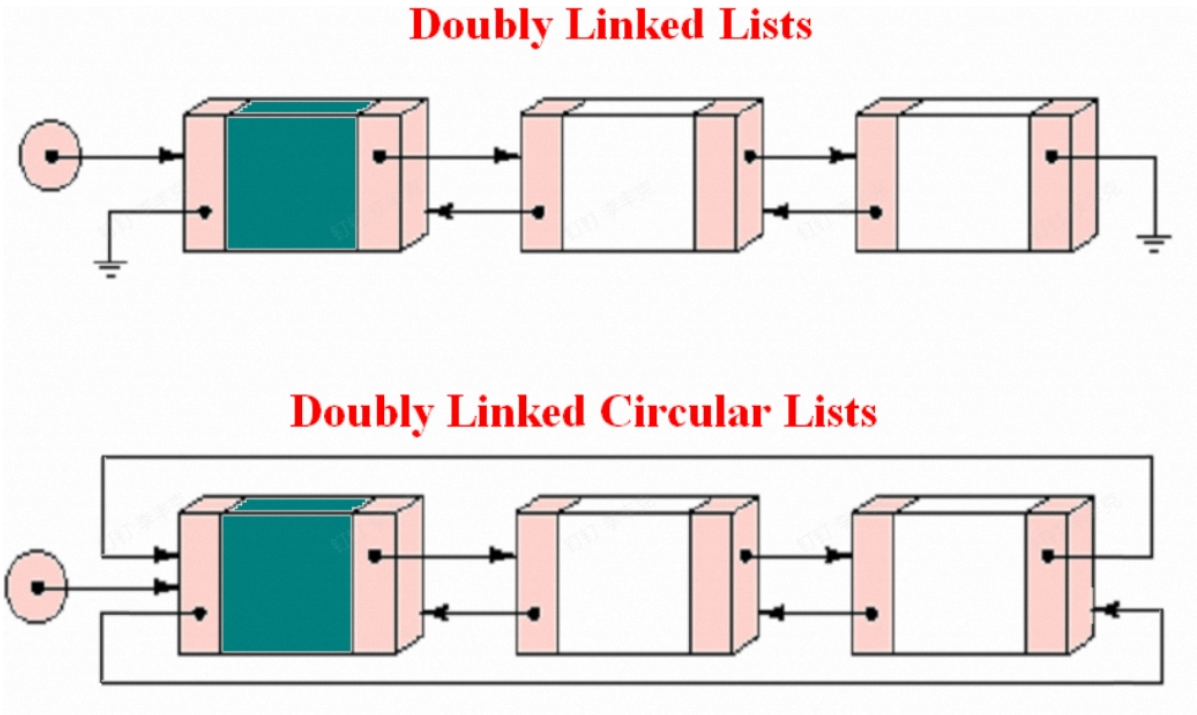
双向链表

在前驱和后继方向都有指针指向的线性链表

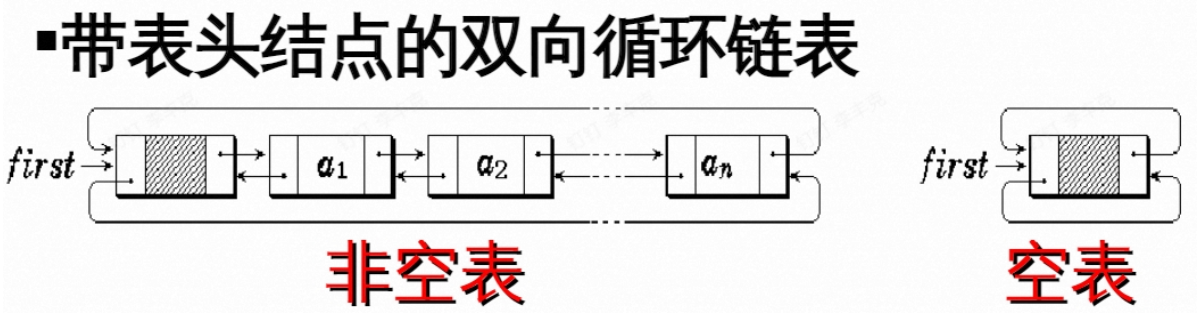
```
Typedef struct DNode{
    Type data;
    struct DNode *llink;
    struct DNode *rlink;
}DNode *head;
```

双向循环链表

头结点的llink指向尾结点，尾结点的rlink指向头结点



带表头结点的双向循环链表



双向循环链表搜索算法

```
DblNode*current;// 当前结点
int Find(const Type &target){
    DblNode*p=head->rlink;
    while(p!=first&&p->data!=target) p=p->rlink;
    if(p!=first){current=p;return 1;}
    return 0;
}
```

双向循环链表插入算法

```
void Insert(const Type&value){
    DblNode *p;
    if(first->rlink==first){//空表插入，只能插入后方
    p={value,first,first};
    current=p;
    current=first->rlink;
    first->llink=current;
    }
    else{//非空表插入，插入current的前方
    p={value,current->llink,current};
    current->llink=p;
    current=current0>llink;
    current->llink->rlink=current;
    }
}
```

双向循环链表删除操作

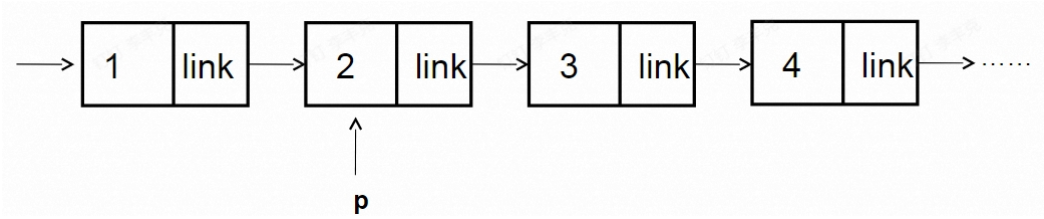
```
void Remove() {
    if(current!=NULL){
    DblNode*temp=current;
    current=current->rlink;
    current->llink=temp->llink;
    temp->llink->rlink=current;
    delect temp;
    }
    if(current==first)
    current=NULL;
}
```

循环链表和双向链表特点

循环链表可由任意一个结点出发访问其它结点；

双向链表可以快速访问任意结点的直接前驱结点；

EXAMPLE



1，如何删除2的结点

将2处结点赋值为3，3处结点赋值为2，然后删除值为2的结点，也满足要求；

2，反向打印单链表

递归

```
void PrintListReversingly(LNode*head){
    if(head!=NULL){
    if(head->next!=NULL){
    PrintListReversingly(head->next);
    }
    printf("%d",head->value);
    }
}
```

3，输入一个链表，输出倒数第k个元素，计数从1开始

用两个指针，一个从头开始遍历，行动到k-1步时，第二个指针开始从头行动，等第一个指针行动到末尾时，第二个指针正好行动到倒数第k个位置。

4，如何找到单向链表的中间结点

两个指针，一个步长1一个步长2，等2的到终点，1的到中间。

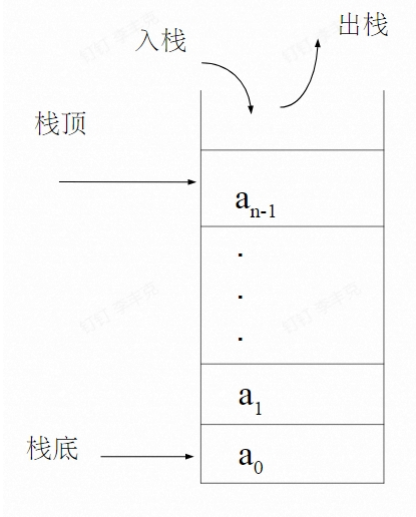
2.2 栈和队列

栈和队列是限定删除和插入操作只能在表的端点处实现的线性表

2.2.1 堆栈

栈是限定仅在表的一端（一般表尾）进行插入和删除的线性表

允许插入和删除的这一端叫栈顶，另一端叫栈底，栈的插入叫做入栈、进栈；栈的删除叫做出栈、退栈。



按a0,a1,a2的顺序入栈，a0是栈底，an是栈顶，遵循后进先出的原则。又称LIFO表。

入栈要求栈不能满，否则报错overflow（上溢）；出栈要求栈不能空，否则报错underflow（下溢）。

抽象模型

```
ADT Stack{
    data:a0,a1,a2;
    relation:<a0,a1>,<a1,a2>;
}
```

栈也有两种储存结构：顺序存储和链式存储

2.2.1.1 顺序栈

顺序栈存储结构是一组连续地址的存储单元依次自栈底到栈顶存储，同时有top指针（下标），指示栈顶元素的当前位置。

top=-1表示空栈，入栈top+1，出栈top-1

定义

```
#define MAXSIZE 1000
typedef struct stack{
    int elem[MAXSIZE];
    int top;
}SQSTACK;
```

基本操作

初始化

```
void InitStack(SQSTACK *ps){
    ps->top=-1;
}
```

判断栈是否空

```
int StackEmpty(SQSTACK s){
    if(s.top==-1) return 1;
    else return 0;
}
```

入栈

```
int Push(SQSTACK *ps,int e){
    if(ps->top==MAXSIZE) return 1;
    ps->top++;
    ps->elem[ps->top]=e;
    return 0;
}
```

出栈

```
int Pop(SQSTACK *ps,int *pe){
    if(ps->top==-1) return 1;
    *pe=ps->elem[ps->top];
    ps->top--;
    return 0;
}
```

读栈顶元素

```
int GetTop(SQSTACK s,int *pe){
    if(s.top==-1) return 1;
    *pe=s.elem[s.top];
    return 0;
}
```

2.2.1.2 链栈

链式存储结构，是一种特殊的单链表，仅限在表头处进行插入和删除的单链表

链栈中指针的方向是从栈顶指向栈底

top→a0（栈顶）→a1→a2→a3（栈底）

链栈中没必要附加表头结点

定义

```
typedef struct snode{
    int data;
    struct node* next;
}SNODE;
SNODE top;
```

基本操作

入栈：相当于在单链表第一个节点前进行插入操作

```
void push(SNODE **pps,int e){
    SNODE *p;
    p=(SNODE*)malloc(sizeof(SNODE));
    p->data=e;
    p->next=*pps;
    *pps=p;
}
```

出栈：相当于单链表中删除第一个结点

```
int pop(SNODE**pps,int *pe){
    SNODE*p;
    if(*pps==NULL) return 1;
    *pe=*pps->data;
    p=*pps;
    *pps=p->next;
    delect p;
    retutn 0;
}
```

给定序列ABC，用栈实现其所有的序列变换

ABC: pushA; pop; pushB; pop; pushC; pop;

ACB: pushA; pop; pushB; pushC; pop; pop;

BCA: pushA; pushB; pop; pushC; pop; pop;

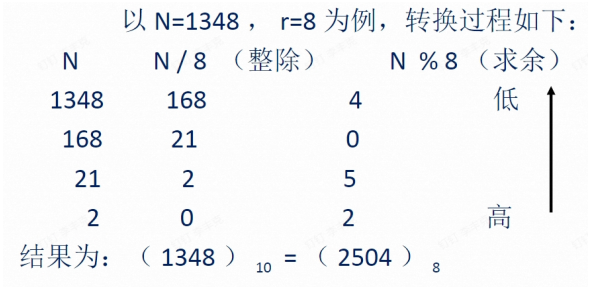
BAC: pushA; pushB; pop; pop; pushC; pop;

CBA: pushA; pushB; pushC; pop; pop; pop;

不能实现CAB

2.2.1.3 栈的应用

1，进制转换，十进制数N转换为r进制数，用辗转相除法



```
void conversion(int N,int r){
    SQSTACK *ps;int x;
    InitStack(ps);
    while(N){
    push(ps,N%r);
    N=N/r;
    }
    while(!StackEmpty(*ps)){
        Pop(ps,&x);
        printf("%d",&x);
    }
    return 0;
}
```

2，程序嵌套调用，如递归算法

3，表达式求值

一个表达式由操作数（运算对象），操作符（运算符），分界符（空格 # ； 等）

算术表达式有三种表达形式

中缀：<操作数><操作符><操作数> A+B

前缀：<操作符><操作数><操作数> +AB

后缀：<操作数><操作数><操作符> AB+

操作符有四种类型：

算数操作符：双目操作符（++，--），单目操作符（% + - * /）

关系操作符：< <= > >= == !=

逻辑操作符：&& ||

括号：（）

中缀表示的相邻两个操作符计算次序：

优先级高的先算

优先级相同的按结合顺序

有括号先算括号内

中缀算术表达式求值流程：

两个栈：操作数栈s1，操作符栈s2，先都初始化为空栈

对输入的字符如果是操作数，放入栈s1

对输入的字符如果是操作符，比较char与s2栈顶元素的优先级：

如果s2_top<char，则char进s2栈

如果s2_top>char，则依次弹出s1的两个栈顶元素a2，a1，从s2栈弹出栈顶操作符θ，然后运算(a1)θ(a2)，运算结结果进s1栈，　　继续判断此操作符与新栈顶元素操作符的优先级

如果s2_top=char，则判断s2_top的结合性

如果是左到右，则从s1栈依次弹出两个栈顶元素a2和a1，从s2栈弹出操作符θ，然后运算(a1)θ(a2)，运算结结果进s1　　栈，char进s2栈

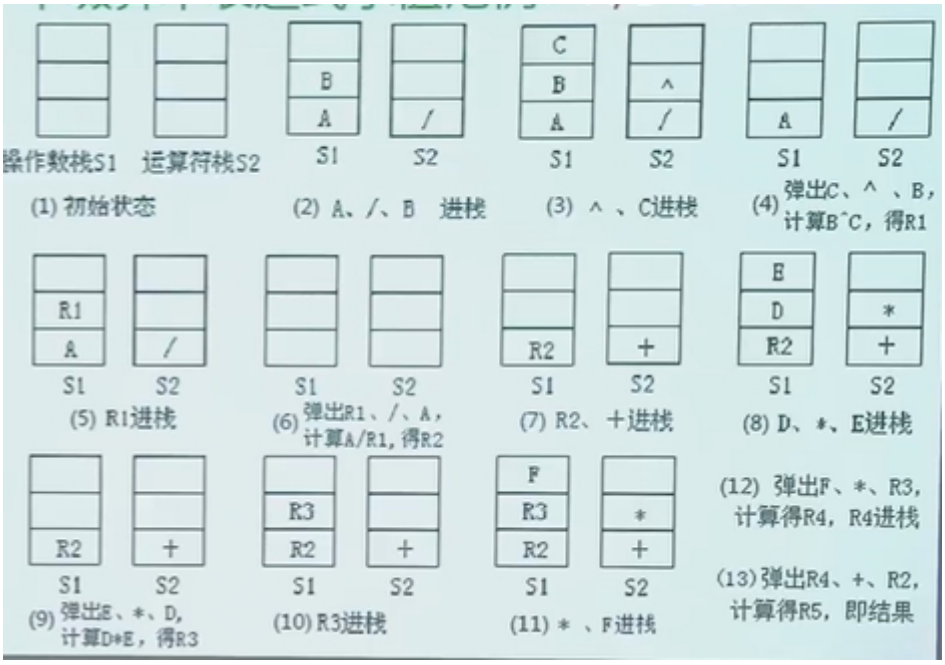
如果是右到左，直接进s2栈。

所有char都读完后，根据s2栈情况做判断。

如果s2栈为空，s1栈栈顶元素即为所求值，

如果s2栈不空，从s1栈依次弹出两个栈顶元素a2和a1，从s2栈弹出操作符θ，然后运算(a1)θ(a2)，运算结结果进s1栈，重复此操作直至s2栈变空。

求A



后缀表达式：与相应的中缀表达式操作数次序相同 没有括号

中缀转化为后缀规则：

如果是操作数，直接输出到队列

如果当前运算符高于栈顶运算符 入栈

如果当前运算符小于栈顶运算符，栈顶的退栈，输出到队列，再比较

如果当前运算符等于栈顶运算符：

栈顶为（，当前为），则栈顶退栈

栈顶为#，当前为#，栈顶退栈，运算结束.

	+	—	×	/	(	)	#
+	>	>	<	<	<	>	>
—	>	>	<	<	<	>	>
×	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

后缀表达式求值：

遇到数就入栈，遇到操作符就弹出依次栈顶元素b、a，运算a操作符b，结果再入栈，重复直至无操作符，栈顶元素即结果。

计算机计算中缀表达式就是先换为后缀表达式，再计算。

2.2.1.4 思考题

1，如何得到栈的最小元素

添加一个辅助栈，第一个数据入栈时，辅助栈也同时入栈，后续再有数据入栈时，和辅助栈栈顶元素比较，如果更小就入辅助栈，否则不入栈。每次出栈时，也同时辅助栈比较出栈。

步骤	数据栈	辅助栈	最小值
1	3	3	3
2	3,4	3	3
3	3,4,2	3,2	2
4	3,4,2,1	3,2,1	1
5	3,4,2	3,2	2
6	3,4	3	3
7	3,4,0	3,0	0

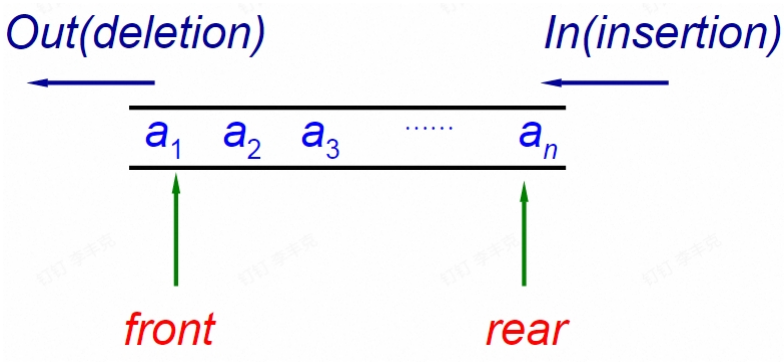
2，栈升序排列

原栈，升序栈，辅助栈

原栈弹出元素，入升序栈；后续原栈弹出元素，此元素与升序栈栈顶元素比较，符合升序就入栈，不符合就升序栈弹出栈顶，辅助栈入栈，再比较，符合升序就入栈，不符合继续弹，直至符合入栈后，辅助栈再依次出栈入栈。

2.2.2 队列

先进先出，一个底部是空的栈，当满的时候再输入，会把最前面的挤出来。



2.2.2.1 顺序队列

一维数组来表示，入队要考虑队满，出队要考虑队空。有队首指针front，队尾指针rear。

front始终指向队列第一个元素，rear始终指向队尾元素下一个单元。

入队时在rear处添加元素，rear+=1；

出队时取出front元素，front+=1；

front之前的区域被忽视了，会出现假溢出现象。

2.2.2.2 循环队列

为区别队满队空，规定队列中始终留一个空单元

队首指针进1：front=（front+1）%MAXSIZE；

队尾指针进1：rear=（rear+1）%MAXSIZE；

队列初始化：front=rear=0；

队空条件：front==rear

队满条件：（rear+1）%MAXSIZE==front

```
//定义队列
typedef struct QUEUE{
    int count;
    int front,rear;
    int QUEAE_array[MAXSIZE];
}Queue;
//初始化队列
int InitQueue(Queue *q){
    count=0;
    rear=front=0;
    q->QUEUE_array=(int*)malloc(sizeof(int)*MAXSIZE);
    return 0;
}
//判断空
int Empty(Queue *q){
    if(q->count==0) return 1;
    else return 0;
}
//判断满
int Full(Queue *q){
    if((rear+1)%MAXSIZE==front)) return 1;
    else return 0;
}
```

```
//入队
int Append(Queue *q,int e){
    if(Full(q)) return overflow;
    q->QUEUE_array[rear]=e;
    rear=(rear+1)%MAXSIZE;
    count++;
    return 1;
}
//出队
int Serve(Queue *q,int &item){
    if(Empty(q)) return underflow;
    item=q->QUEUE_array[front];
    front=(front+1)%MAXSIZE;
    return 1;
}
//取队首元素
```

2.2.2.3 链队列

队首在链头，队尾在链尾

链队列在进队时不用考虑队满，出队要考虑队空。

队空条件 front==NULL;

进队只需改变rear，出队改变front

```
//入队
if(front==NULL) front=rear=newnode;
else{
    rear->next=newnode;
    rear=rear->next;
    }
```

```
//出队
if(front==NULL) return underflow;
e=front->data;
front=front->next;
```

2.2.3 summary

相同点:

逻辑结构相同，都是线性的;都可以用顺序存储或链表存储;栈和队列是两种特殊的线性表，即受限的线性表(只是对插入、删除运算　　加以限制)

不同点:
① 运算规则不同，线性表为随机存取，而栈是只允许在一端进行插入和删除运算，因而是后进先出表 LIFO;队列是只允许在一端进行插入、另一端进行删除运算，因而是先进先出表 FIFO。
②用途不同，线性表比较通用;堆栈用于元素的保存次序与使用顺序相反的情况;队列用于元素的保存次序与使用顺序相同的情况。

两个栈实现一个队列

入队时，元素弹入s1。出队时，判断s2是否为空，不为空就弹出栈顶元素；为空就先将s1元素逐个出栈再入栈，相当于反方向入栈，再弹出栈顶元素。0

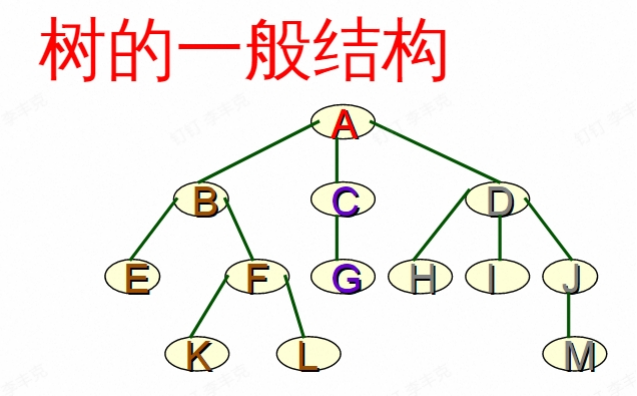
第三章 树

3.1 定义与基本术语

定义

D中的元素都有相同的数据类型

存在唯一的成为根root的元素，其余结点又可分为若干互不相交的有限子集，每个自己都符合树的定义，成为子树。



结点：数据元素+若干指向其子树的分支

结点的度：结点的子结点的个数

树的度：树中所有结点的度的最大值

叶子结点：度为0的结点

分支结点：度大于0的结点

双亲和孩子：结点的子树的根节点成为该节点的孩子，反之结点是孩子的双亲

层：根节点为层一，任意结点的层等于其双亲层数加一

深度：各叶子结点层数最大值

兄弟：有同一双亲结点

堂兄弟：双亲在同一层的结点

路径：根节点到某节点经过的分支和结点组成

祖先：从根结点到此结点所经分支上所有结点；结点的子树上所有结点都是其子孙

森林：m棵互不相交的树的集合；任一个非空树是一个二元组Tree=(root,F)，root是根节点，F是子树森林

有序树&无序树：子树之间从左到右存在次序关系不能互换，称为有序树；子树之间不存在次序关系，称为无序树。

基本操作：

查找类：

Root(T)　　// 求树的根结点

Value(T, cur_e)　// 求当前结点的元素值

Parent(T, cur_e)　// 求当前结点的双亲结点

LeftChild(T, cur_e) // 求当前结点的最左孩子

RightSibling(T, cur_e)　// 求当前结点的右兄弟

TreeEmpty(T)　// 判定树是否为空树

TreeDepth(T)　// 求树的深度

TraverseTree(T, Visit())　// 遍历

插入类：

InitTree(&T) //初始化，置空树

CreateTree(&T, definition) // 按定义构造树

Assign(T, cur_e, value)　// 给当前结点赋值

InsertChild(&T, &p, i, c) // 将以c为根的树插入为结点2p的第i棵

删除类：

ClearTree(&T)　//将树清空

DestroyTree(&T)　// 销毁树的结构

DeleteChild(&T, &p, i) // 删除结点p的第i棵子树

3.2 树的存储结构

3.2.1 双亲数组表示法（顺序存储结构）

每个结点含两个域：

数据域：放结点本身信息

双亲域：指示本结点双亲结点在数组中位置

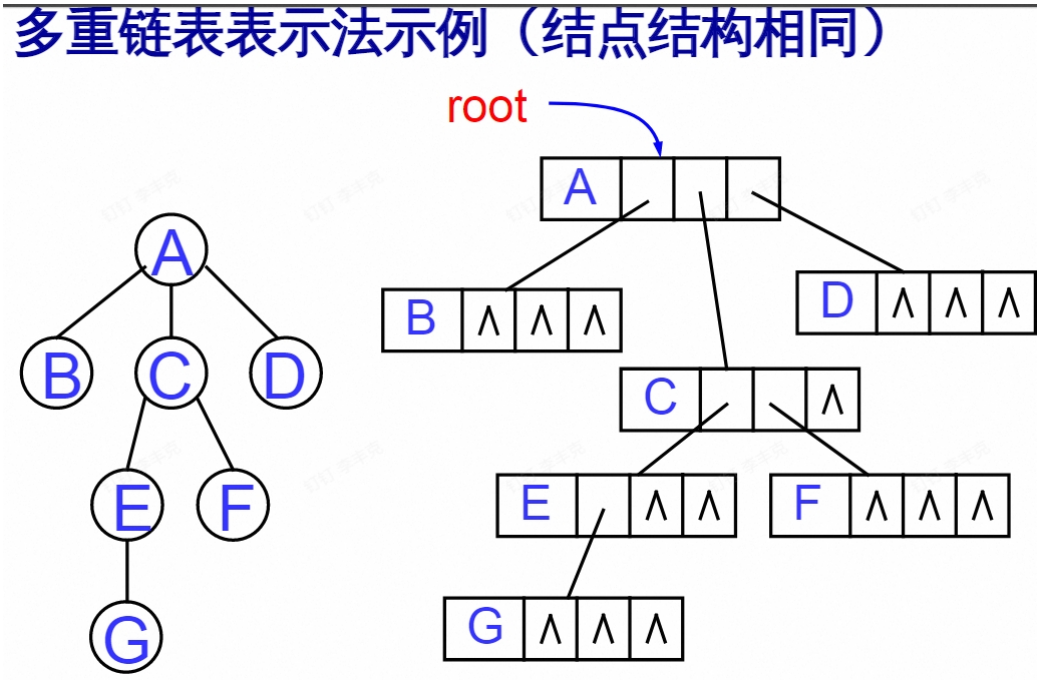
找双亲容易，找孩子难

3.2.2 孩子链表表示法

每个结点除了数据域还有指针域存储其孩子结点的指针。

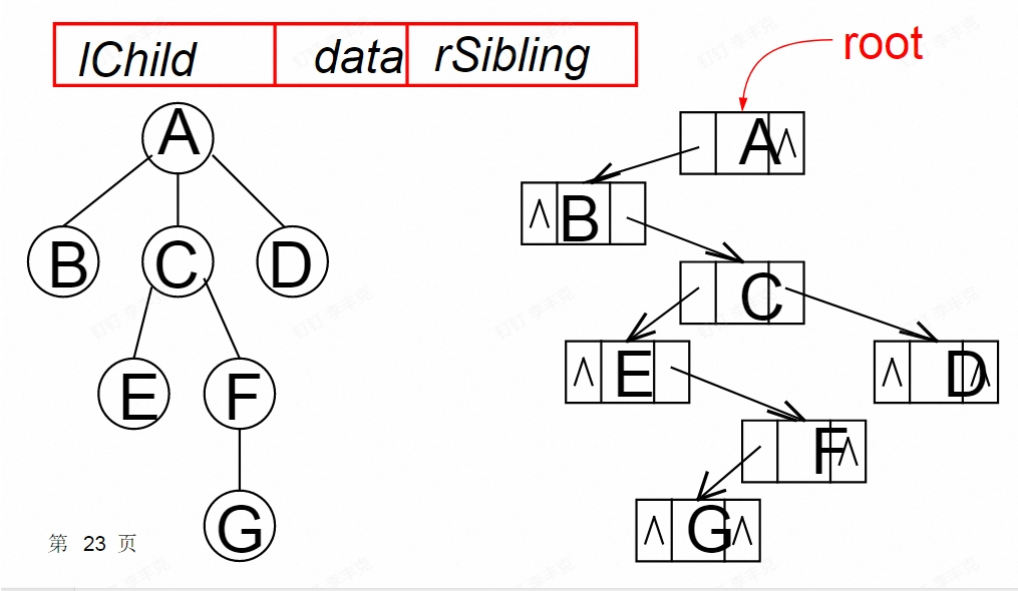
各结点结构相同：指针个数相等，都是树的度

各结点结构不同：指针个数不等，是该节点的度



3.2.3 左孩子右兄弟表示法

用二叉链表：每个结点的两个指针分别指向第一个孩子结点和下一个兄弟结点



3.3 二叉树

3.3.1 定义

每个结点最多有两个孩子结点的树

二叉树五种形态：空树，只含根节点，左子树为空树，右子树为空树，左右均不为空树。

二叉树基本操作：

Root(T)：求二叉树T的根结点；

Value(T, e)：求结点e的值；

Parent(T, e)：求结点e的双亲结点；

LeftChild(T, e)：求结点e的左孩子结点；

RightChild(T, e)：求结点e的右孩子结点；

LeftSibling(T, e)：求结点e的左兄弟结点；

RightSibling(T, e)：求结点e的右兄弟结点；

BiTreeEmpty(T)：判 断二叉树T是否为空树；

BiTreeDepth(T)：求二叉树T的深度；

InitBiTree(&T)：初始化二叉树T为空树；

Assign(T, &e, value)：为结点e赋值；

CreateBiTree(&T, definition)：根据定义创建 二叉树；

InsertChild(T, p, LR, c)：将c插入到二叉树T中成为结点p的左（或右）孩子结点；

ClearBiTree(&T)：清空二叉树T；

DestroyBiTree(&T)： 销毁二叉树T；

DeleteChild(T, p, LR)：删除二叉树T中结点p的左（或右）孩子结点

二叉树的遍历：

PreOrderTraverse(T, Visit())：前序遍历

InOrderTraverse(T, Visit())：中序遍历

PostOrderTraverse(T, Visit())：后序遍历

LevelOrderTraverse(T, Visit())：层序遍历

重要性质：

性质一：

在二叉树的第i层上最多有2^(i-1)个结点

性质二：

深度为k的二叉树上至多有2^k-1个结点。

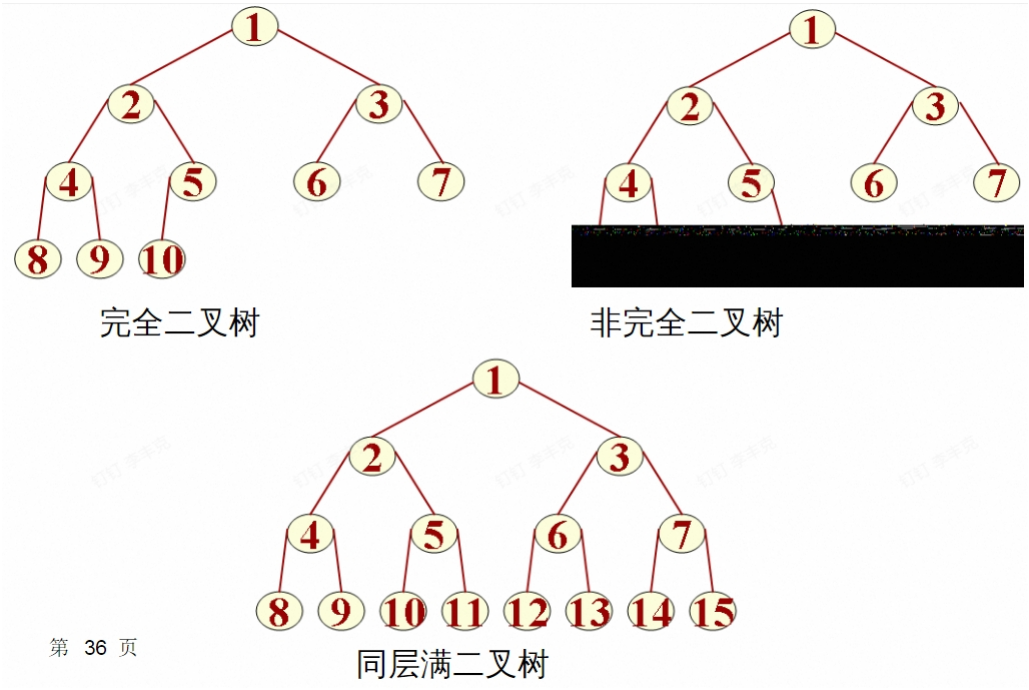
性质三：

对任何一个二叉树，如果他有n0个叶子结点，n2个度为2的结点，则必有n0=n2+1

两种二叉树：

满二叉树：每一层上的结点数都是最大结点数，，所有支结点都有左右子树，结点可以连续编号。

完全二叉树：深度为k，有n个结点的二叉树，当且仅当其每一个结点都与深度为k的满二叉树中编号从1到n的结点一一对应，该二叉树称为完全二叉树。或者深度为k的满二叉树从1到n的前n个编号组成了一个完全二叉树，要求n大到编号到最后一层。



区别：满二叉树是叶子一个也不少的树(所有 k层的结点全满)，而完全二叉树虽然前 k-1 层是满的，但最后一层(第k层)却允许在右边缺少连续若干个结点。满二叉树是完全二叉树的一个特例:满二叉树一定是完全二叉树，而完全二叉树不一定是满二叉树。

性质四:

具有n个结点的完全二叉树的深度为[log_2 n]+1(以2为底n的对数)

性质五:

对含n个结点的完全二叉树从上到下，从左到右进行1-n的编号，对任一个编号为i的结点:

如果i=1，则是根节点，无双亲；否则[i/2]就是其双亲结点。

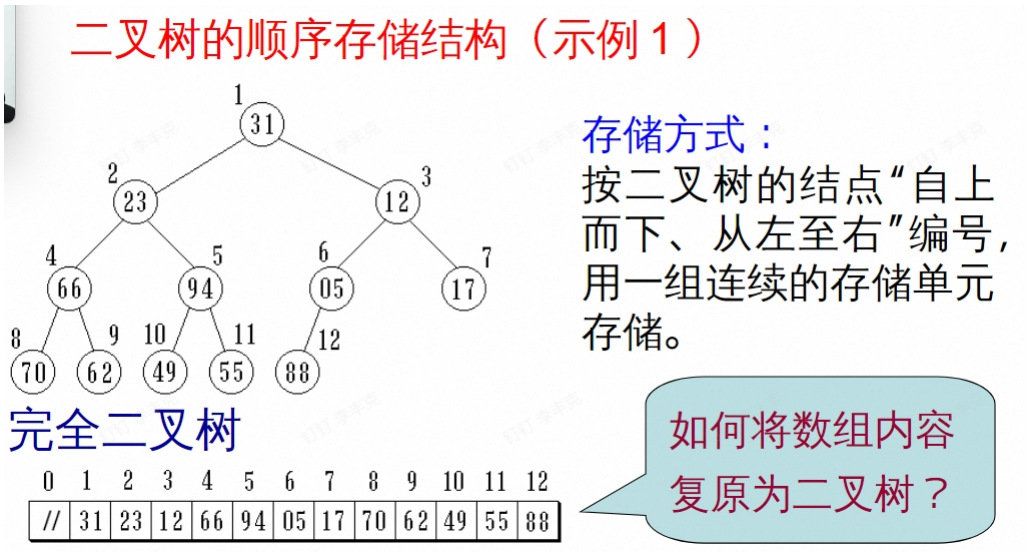
如果2i>n，则此结点无左孩子结点；如果2i+1>n,则此结点无右孩子结点。

3.3.2 二叉树的存储结构

顺序存储

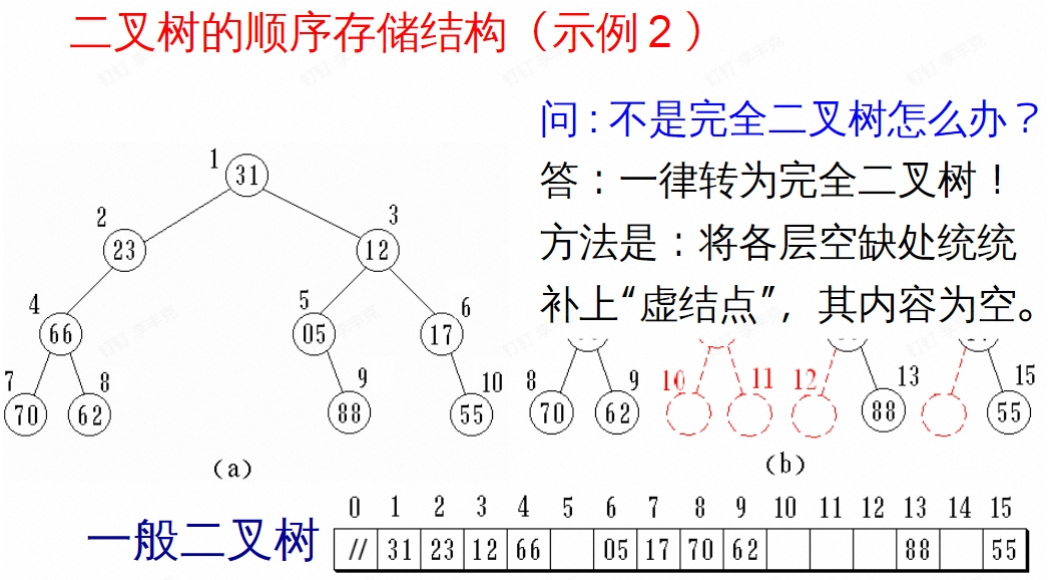
完全二叉树的顺序存储:

按照自上而下、从左到右的顺序给结点编号，用数组存储，结点从1开始编号，数组也从1下标开始存储。编号为i的结点，其双亲结点为[i/2]，左孩子结点2i，右孩子结点2i+1



非完全二叉树顺序存储：

一律转为完全二叉树，补充空结点，此时结点编号之间的关系仍存在。

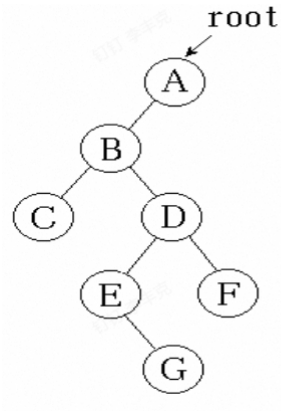


编号从零开始时，i结点的双亲结点[(i+1)/2]-1，左孩子结点2i+1，右孩子结点2(i+1)。(建议别用)

一般二叉树都要改为完全二叉树，会造成存储空间浪费。

结构数组：

每个元素都是一个结构体，包含data，parent，leftchild，rightchild，四个域分别存储数据，双亲结点、左孩子、右孩子结点所在的数组单元下标。如果不存在那么相对域的下标取-1；



	data	*parent*	*leftChild*	*rightChild*
0	A	−1	1	−1
1	B	0	2	3
2	C	1	−1	−1
3	D	1	4	5
4	E	3	−1	6
5	F	3	−1	−1
6	G	4	−1	−1

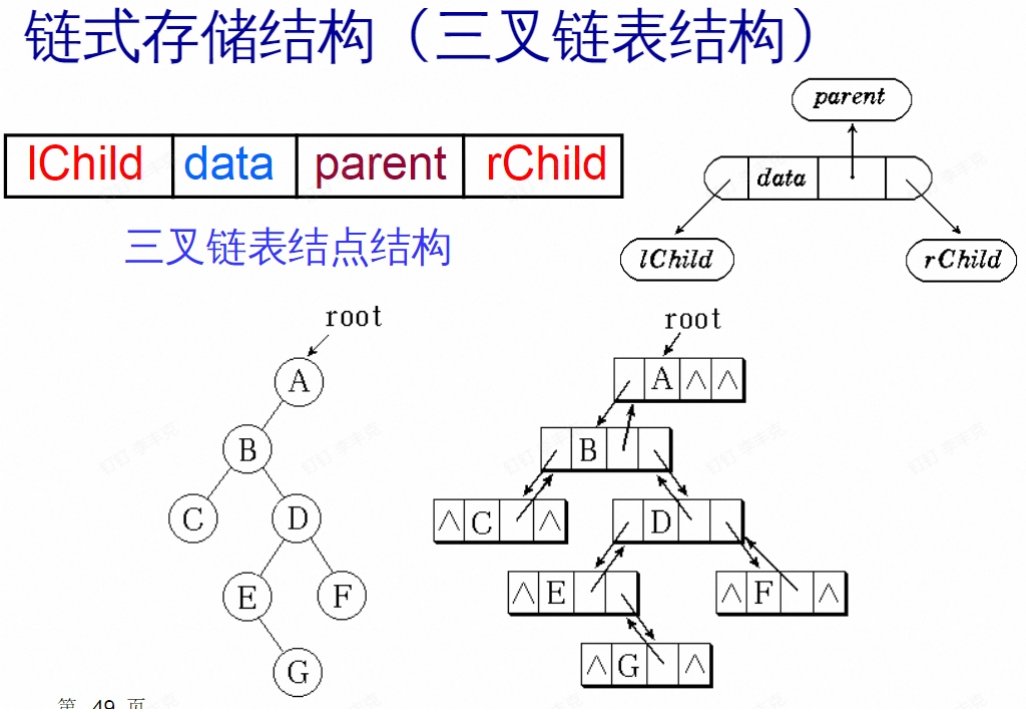
链式存储结构

二叉链表：

每个结点由lchild，data，rchild组成。lchild指向左孩子结点，rchild指向右孩子结点。

三叉链表：

每个结点由parent，lchild，data，rchild。多了个parent指向双亲结点。



3.3.3 二叉树的遍历

设访问根结点记作V，遍历根的左子树记作L，遍历根的右子树记作R。

可能的遍历次序有：

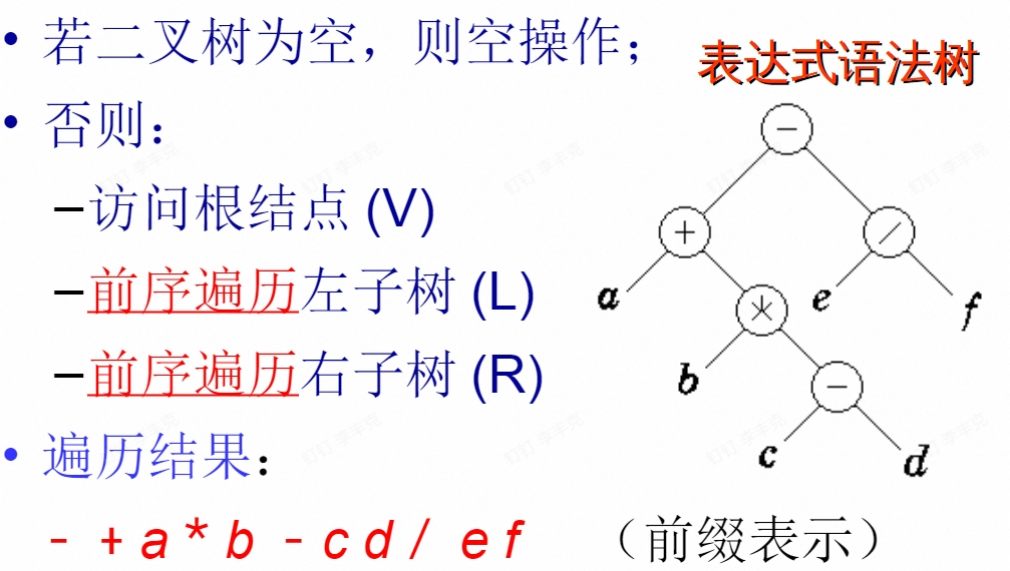
前序VLR，逆前序VRL

中序LVR，逆中序RVL

后序LRV，逆后序RLV

层序遍历

前序遍历

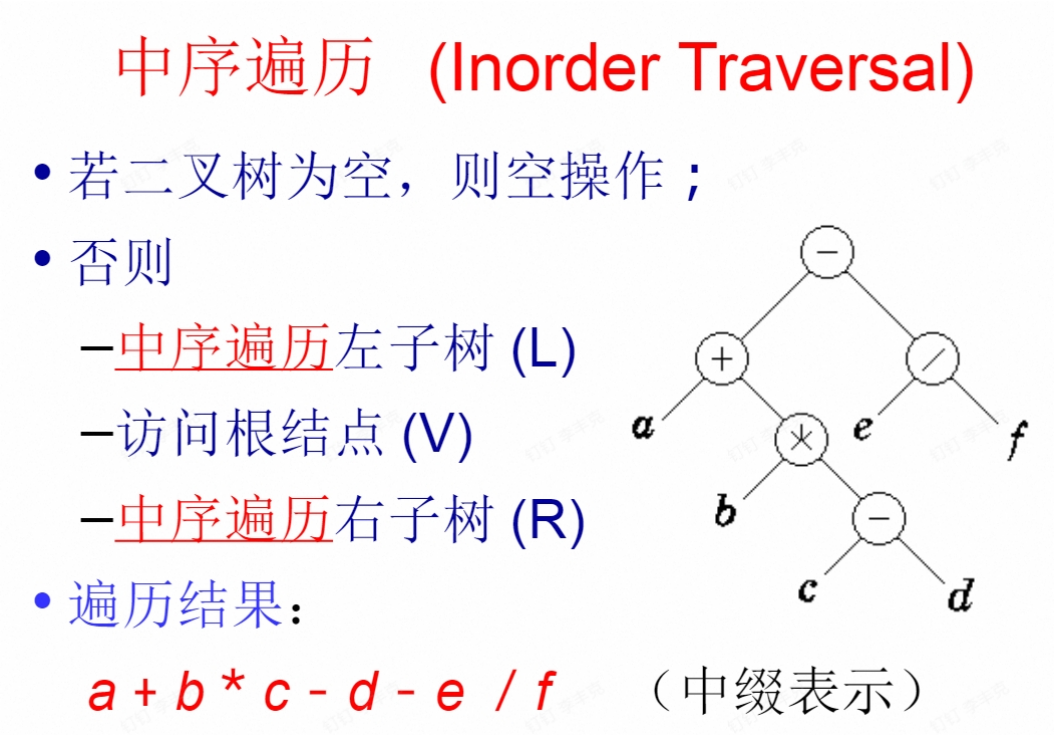


递归算法：

```
void  PreorderTraverse(BTNode  *T){
    if  (T!=NULL) {
        visit(T->data) ;/*  访问根结点*/
        PreorderTraverse(T->Lchild) ;
        PreorderTraverse(T->Rchild) ;
    }}
```

非递归算法略

中序遍历



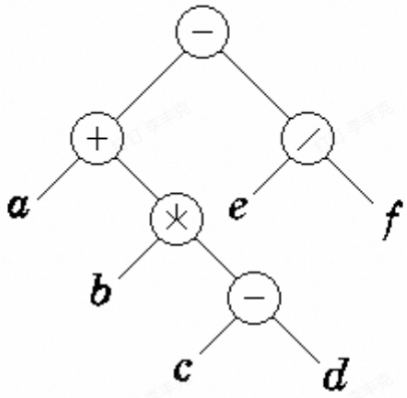
递归算法

```
void   InorderTraverse(BTNode  *T){
    if  (T!=NULL) {
         InorderTraverse(T->Lchild) ;
        visit(T->data) ;/*  访问根结点*/
         InorderTraverse(T->Rchild) ;
    }}
```

后序遍历

后序遍历 (Postorder Traversal)

- 若二叉树为空，则空操作；
- 否则
  - –后序遍历左子树 (L)
  - –后序遍历右子树 (R)
  - –访问根结点 (V)
- 遍历结果 ：



a b c d - * + e f / -　　（后缀表示）

递归算法

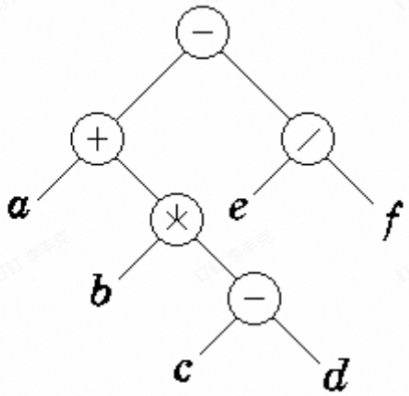
```
void   PostorderTraverse(BTNode  *T){
    if  (T!=NULL) {
         PostorderTraverse(T->Lchild) ;
         PostorderTraverse(T->Rchild) ;
         visit(T->data) ;/*  访问根结点*/
    }}
```

层序遍历

层次遍历 (Levelorder Traversal)

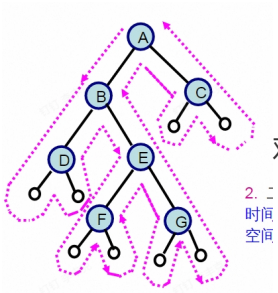
–从上到下

–从左到右



–遍历结果 ：　- ＋ / a* e f b -cd

```
#define MAX_NODE  50
void  LevelorderTraverse( BTNode  *T)
  {  BTNode  *Queue[MAX_NODE] ,*p=T ;
     int  front=0 , rear=0 ;
     if  (p!=NULL)
        {  Queue[++rear]=p;    /*  根结点入队  */
           while (front<rear)
              {  p=Queue[++front];  visit( p->data );
                 if (p->Lchild!=NULL)
                      Queue[++rear]=p;  /*  左结点入队  */
                 if (p->Rchild!=NULL)
                      Queue[++rear]=p;    /*  左结点入队  */
               }
        }
  }
```

简单判断： 

对二叉树做如图画线，在每个结点左方，下方，右方各标记一个小圆圈。如图画线后，前序遍历按照画到左侧标记的顺序来；中序遍历按照画到下侧标记的顺序来；后序遍历按照画到右侧标记的顺序来；层序遍历从上到下，从左到右，一层一层。

前序：ABDEFGC

中序：DBFEGAC

后序：DFGEBCA

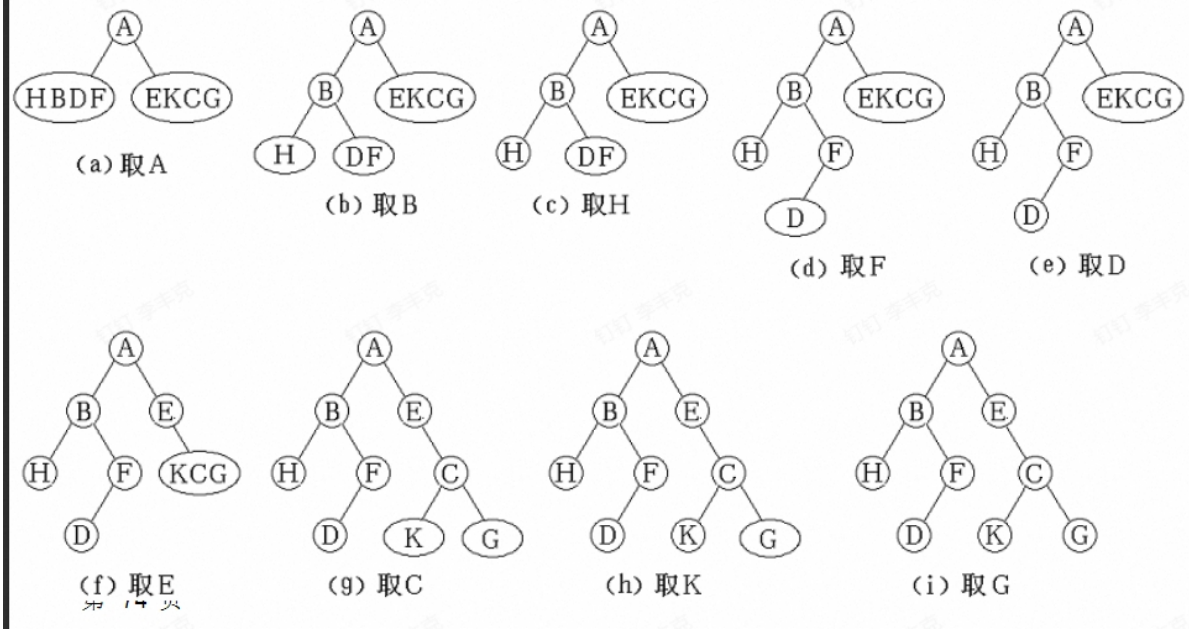
层序：ABCDEFG

前序遍历+中序遍历和后序遍历+中序遍历 可以唯一确定一棵二叉树

前序遍历+后序遍历　　不可唯一确定一棵二叉树

前序遍历序列{ABHFDECKG}

中序遍历序列{HBDFAEKCG}



由前序和中序，前序定抽取顺序，从中序开始分支。

遍历算法的应用有（未完待续）

1. 求二叉树中的节点个数
2. 求二叉树的深度
3. 前序遍历，中序遍历，后序遍历
4. 分层遍历二叉树（按层次从上往下，从左往右）
5. 将二叉 查找树变为有序的双向链表
6. 求二叉树第K层的节点个数
7. 求二叉树中叶子节点的个数
8. 判断两棵二叉树是否结构相同
9. 判断二叉树是不是平衡 二叉树
10. 求二叉树的镜像
11. 求二叉树中两个节点的最低公 共祖先节点
12. 求二叉树中节点的最大距离
13. 由前序遍历序列和中序遍历序列重建二叉树
14. 判断二叉树是不是完全二叉树

3.3.4树与森林转换

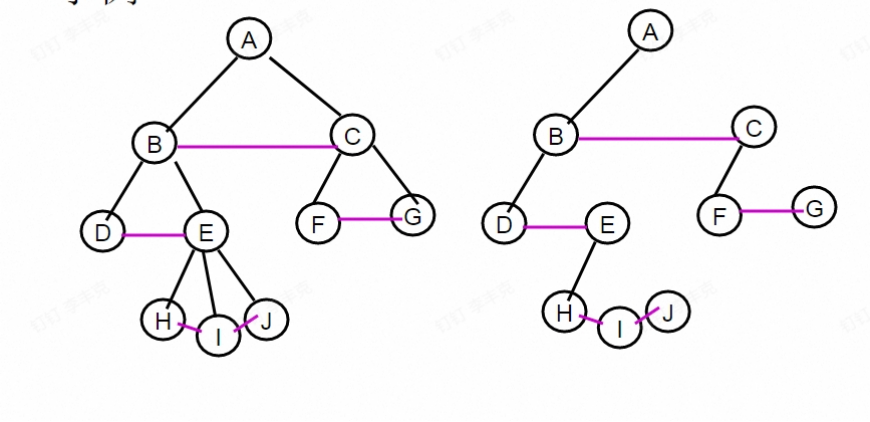
树转化为二叉树

同父亲兄弟结点连接

对每一个非叶子结点只保留第一个孩子的链

树向左旋转45°

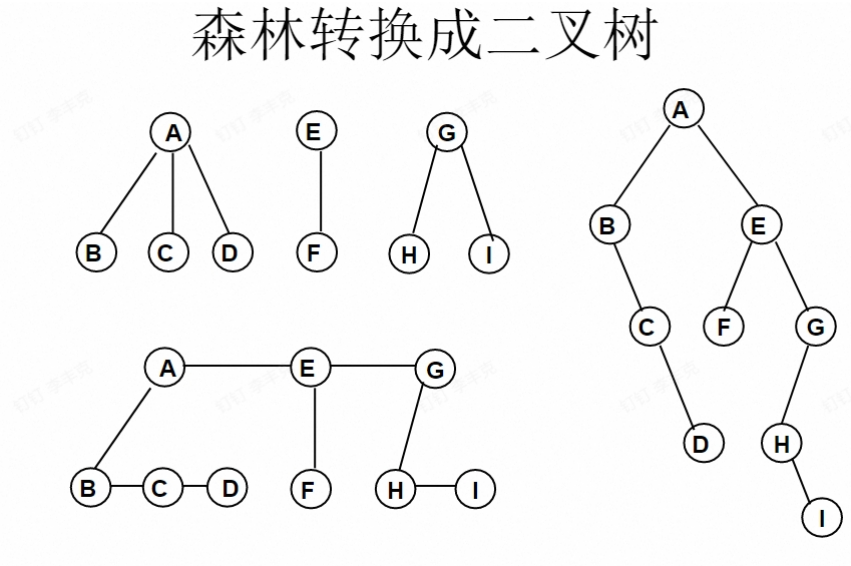
- 事例



森林转化为二叉树

如果F ={T, T...., T} 是森林，则可按如下规则转换成一棵二叉树 B ={root, LB, RB}。

(1)若F为空，则B为空树;
(2)若F非空，则B的根 root 为森林中第一棵树的根 Root(T,)，B 的左子树 LB 是从 T 中根结点的子树森林转换而成的二叉树;

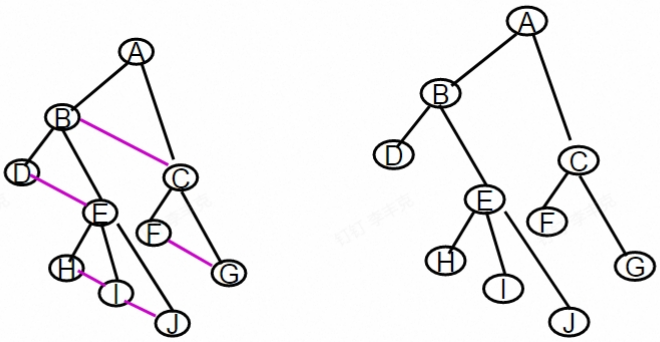


二叉树转换为树

- 转换规则：

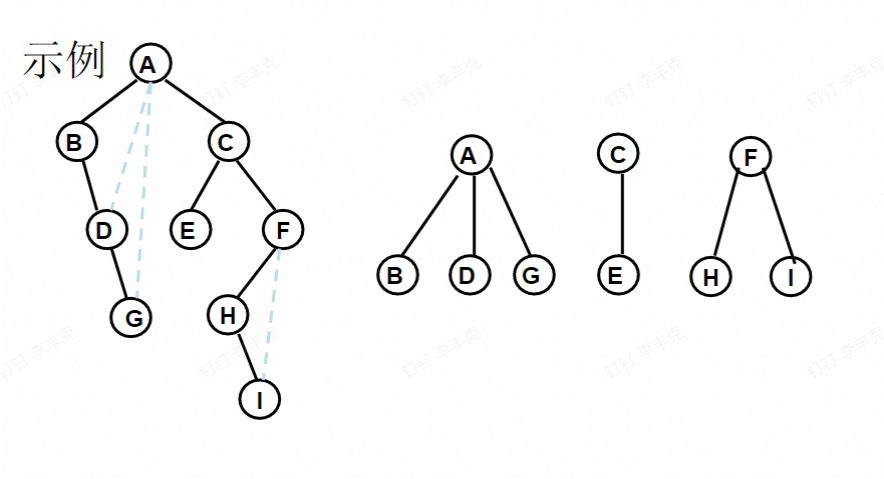
1. 对有左子树的所有结点与其左孩子的右链上的所有结点进行连接；

2. 删除以前所有的旧右链。



二叉树转化为森林

如果 B = {root, LB, RB}是一棵二叉树，则可按如下规则转换成森林 F ={T,T,..., T...}
(1)若B为空，则F为空树;
(2)若B非空，则为森林中第一棵树的根 Root(T)即为 B的根 root， T 中根结点的子树森林是有 B 的左子树 LB 转换而来的森林;F 中除T, 之外的其余树组成的森林，是由 B的右子树转换而成的。



3.4.5 树和森林遍历

树的先根次序遍历：和先序遍历一样

树的后跟次序遍历：和后序遍历一样

树的广度优先遍历：和层序遍历一样

均可用二叉树中画圈的方法写出次序

森林的先根次序遍历：按树的顺序，先序遍历

森林的中根次序遍历：按树的顺序，中序遍历

森林的广度优先遍历：按树的顺序，层序遍历

3.4 哈夫曼树与哈夫曼编码

结点的路径长度：从根到此结点的路径分支数目

树的路径长度：树中每个结点的路径长度的和

树的带权路径长度：每个叶子的权重乘此叶子结点的路径长度的和

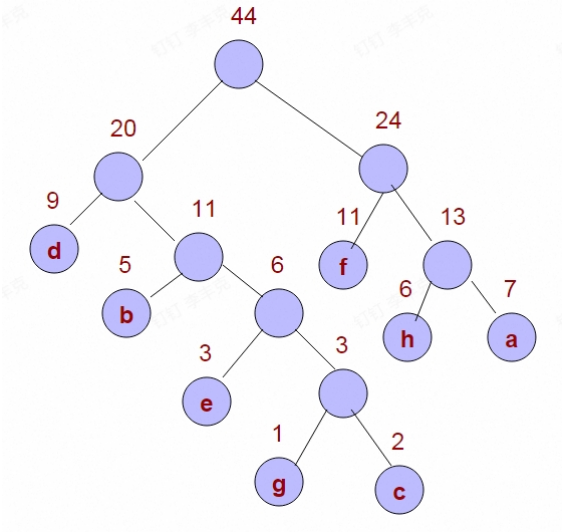
最优树就是寻找带权路径长度最短的树

3.4.1 构造最优树

（1）给定n个权值W={w1,w2...wn}，构造n棵二叉树的集合F={T1,T2...Tn}，其中每棵树都是只含有权值为wi的根节点，左右子树都是空树。（最开始时是n个根）

（2）找出根节点权重最小的两棵树，将这两棵树分别作为第二层层左右子树，其双亲结点也就是根节点，是这两棵子树的根节点的权重之和。从F中删去这两棵树，并且添加进新树。

（3）重新进行（2）纸质F中只剩下一棵树。　　（简直就是天才）



3.4.2 哈夫曼编码

在远程通讯中，要将字符串转化为二进制，为了转换效率与避免歧义，采用前缀编码；即任何一个字符的编码都不是另一个字符编码的前缀。可以用二叉树来设计，约定左分支是0，右分支是1，通过根节点到叶子节点的路径来规定编码，路径越少的字符为越常用的字符，这样可以提高效率。

构建哈夫曼编码，哈夫曼树每个结点的度都不为1，含有n个叶子结点的哈夫曼树共含有2n-1个结点。根据所给的字符及其对应的权重，通过构建哈夫曼树的原理，从叶子到根节点构建，规定左分支为0，右分支为1（不一定）。然后再从根节点到对应字符的路径读出编码。

构造哈夫曼树的算法（未完待续）

第四章 图（未施工）

第五章 搜索树

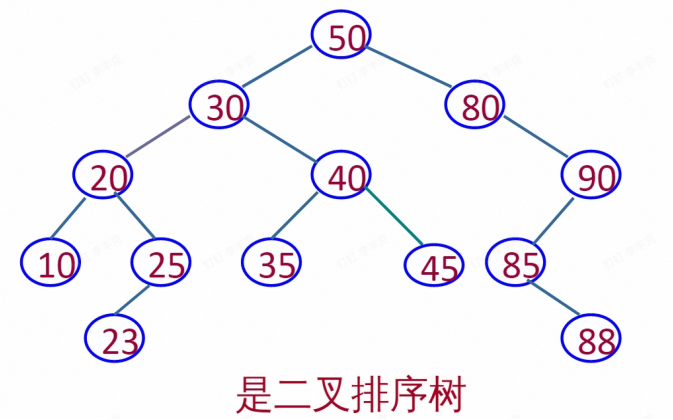
5.1 二叉搜索树

二叉搜索树（二叉排序树）满足如下条件：

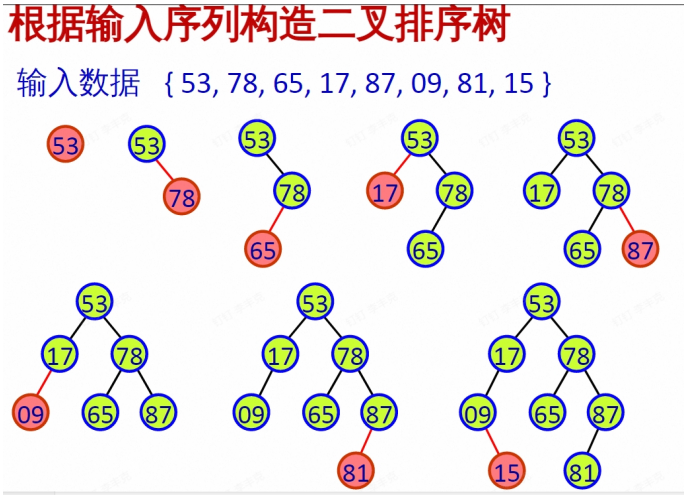
如果他的左子树不空，那么左子树上所有结点都小于根节点；如果他的右子树不空，那么他的右子树上所有结点都大于根节点；

他的左右子树也都是二叉排序树。

即对于任何一个结点，如果他或者他的某个祖宗结点是其双亲结点的左结点，那么这个结点的值小于这个双亲结点；反之如果是右结点，那么这个结点的值大于这个双亲结点。



构建



对新来的数，与每层根节点比较，看自己该去左子树上还是右子树上，比他大的去右边，小的去左边。

输入的序列不同，构建的二叉排序树也不同

遍历

如果对一个二叉搜索树中序遍历，那么输出的值将从小到大排序（我验证过了，是真的）

查找

如果定值等于根节点，查找成功；

如果定值小于根节点，进到左子树上找；

如果定值大于根节点，进到右子树上找；

如果最后指向了空树，查找失败。

插入

二叉搜索树的插入都插入在叶子结点上，其规则与构建二叉搜索树相同。

删除

应该保障删除后二叉排序树特性不变，且中序遍历结果仍然从小到大。

1）如果删除的是叶子结点，直接删

2）如果删除的结点只有左子树或者右子树，那么直接将左子树或者右子树的根节点替代被删除的结点即可

3）如果左右子树都不空：

①将删除结点的右子树，作为左子树最右结点的右子树，左子树根节点再代替删除节点。

（右派投奔左派，但是比左派中最右的还要右）

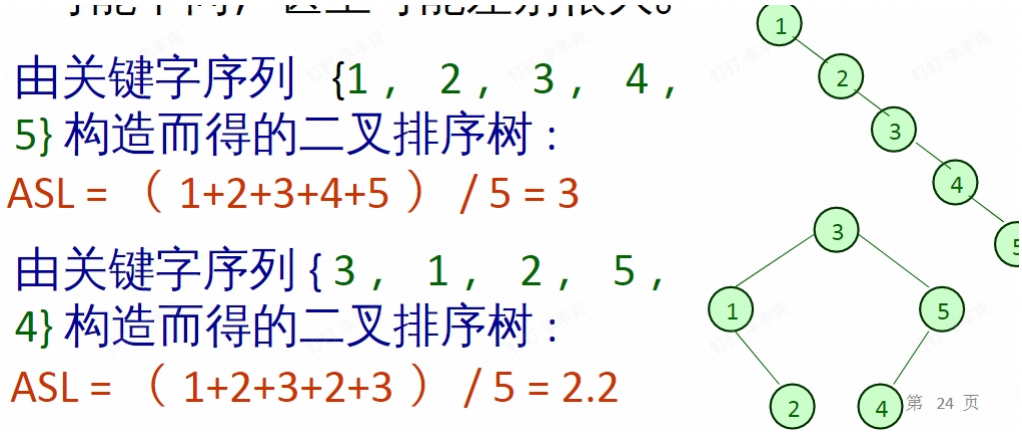
怎么判断是最右结点，我的建议是把图画清楚点，一眼就能看出来；理论上说，按从上到下遍历，第一个 是双亲结点的右子结点并且没有右子结点的结点就是最右结点。一般来说最右结点要么没有子结点，要么只有左子结点。（这个最右结点其实就是这棵树中的最大值）

②先将删除结点的左子树的最右结点代替删除结点； 如果这个最右结点有左子树，就将这个左子树替代这个结点；如果没有，就当作删除了个叶子结点。（同理也可以选择右子树的最左结点，即右子树的最小值，然后再将最左结点的右子树替代他，没有就算了）。

（用左中最大值替代，被替代的结点看作被删除，按前面的规律来）

查找性能

对于任一棵确定的二叉搜索树，按照他的平均查找长度ASL来定义其性能，每经过一个结点查找长度加一。

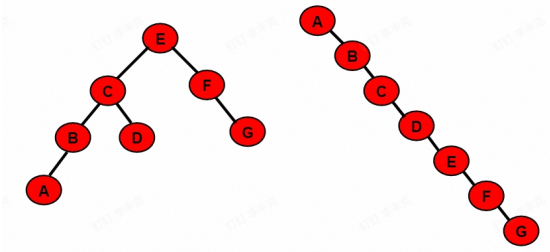


二叉排序树的查找性能与折半查找差不多

二叉排序树的插入与删除更方便，无需移动太多结点，适用于动态查找环境。

5.2 平衡二叉树（AVL树）

为了提高查找速度，避免最坏情况出现，我们要约束平衡性。



平衡因子：结点的平衡度是左子树高度减去右子树高度

平衡二叉树：左子树与右子树的高度差只能为+1，0，-1

其左子树与右子树也是平衡二叉树。

平衡二叉树的平均查找长度为O(log2n)

平衡

危机结点，平衡度不为0，再插入新结点可能会导致失衡的结点

关键：将导致出现危机结点的情况全部分析清楚，就可以使得平衡分类二叉树的性质保持不变！！

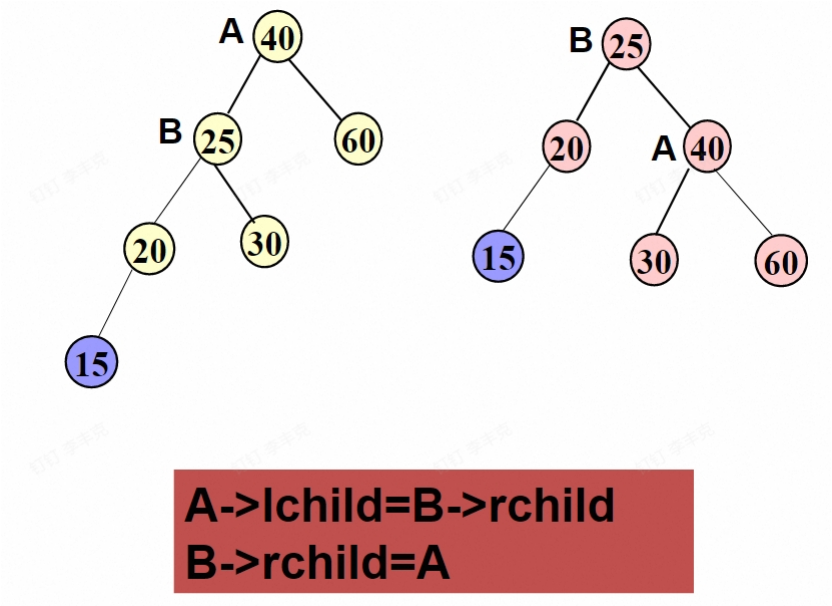
（太经典了，后面红黑树的平衡就是基于他的平衡操作）

LL型

最低失衡点为A，如果在A的左子树B的左子树上插入新结点S后失衡。

此时应该让B做根节点位置，将A作为B的右子，B的右子作为A的左子。

（二换一，旧右补新右）

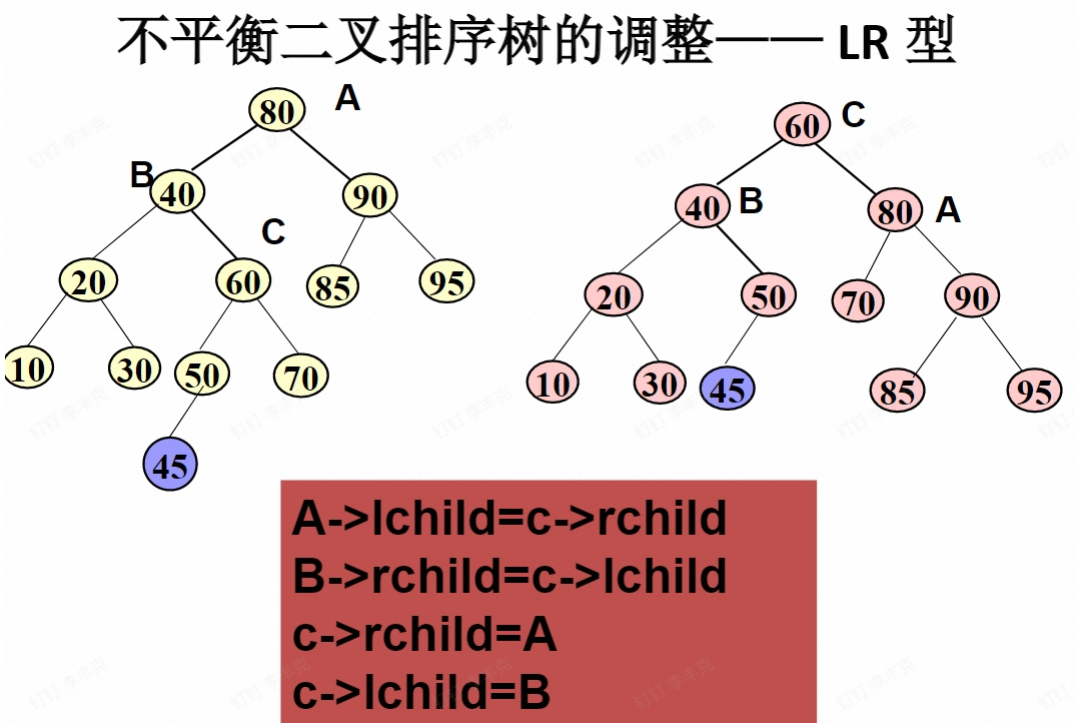


LR型

最低失衡点为A，如果在A的左子树B的右子树C上插入新结点S后失衡。

此时应该让C做根节点位置，B做C的左子（只保留B与左子链接），C原来的左子CL改为B的右子；A做C的右子（只保留右子链接），C原来的右子CR改为A的左子

（三层变一层，二变左，只剩左；一变右，只剩右；原三左补左，右补右）

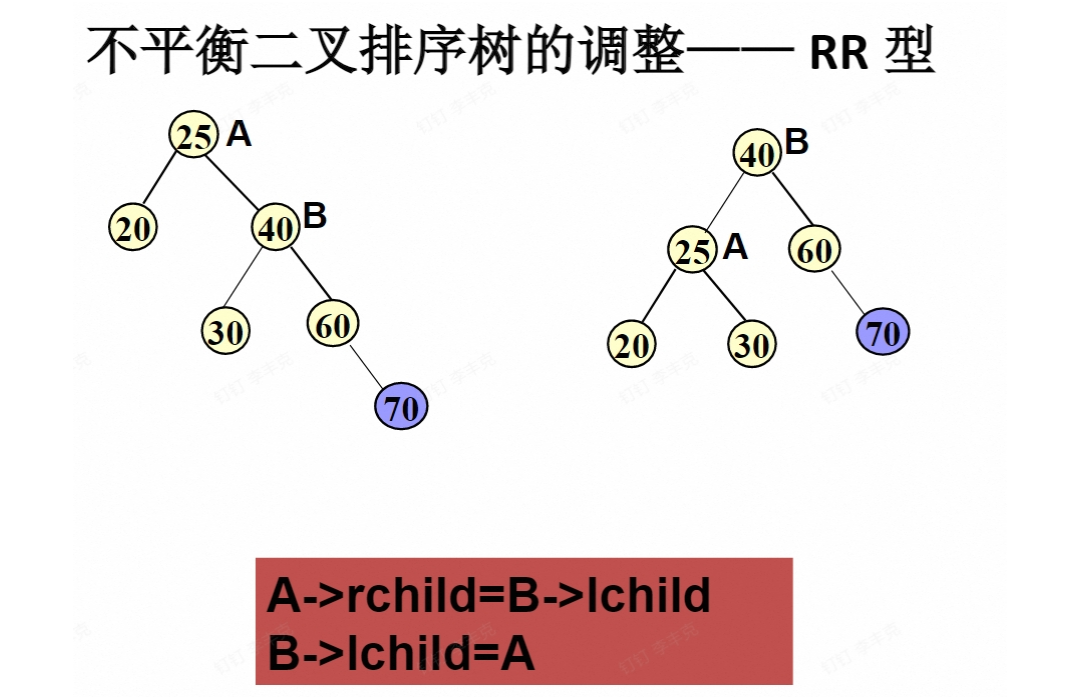


RR型

最低失衡点为A，如果在A的右子树B的右子树上插入新结点S后失衡。

此时应该让B做根节点位置，将A作为B的左子，B的左子作为A的右子。

（二换一，旧左补新左）与LL型对称

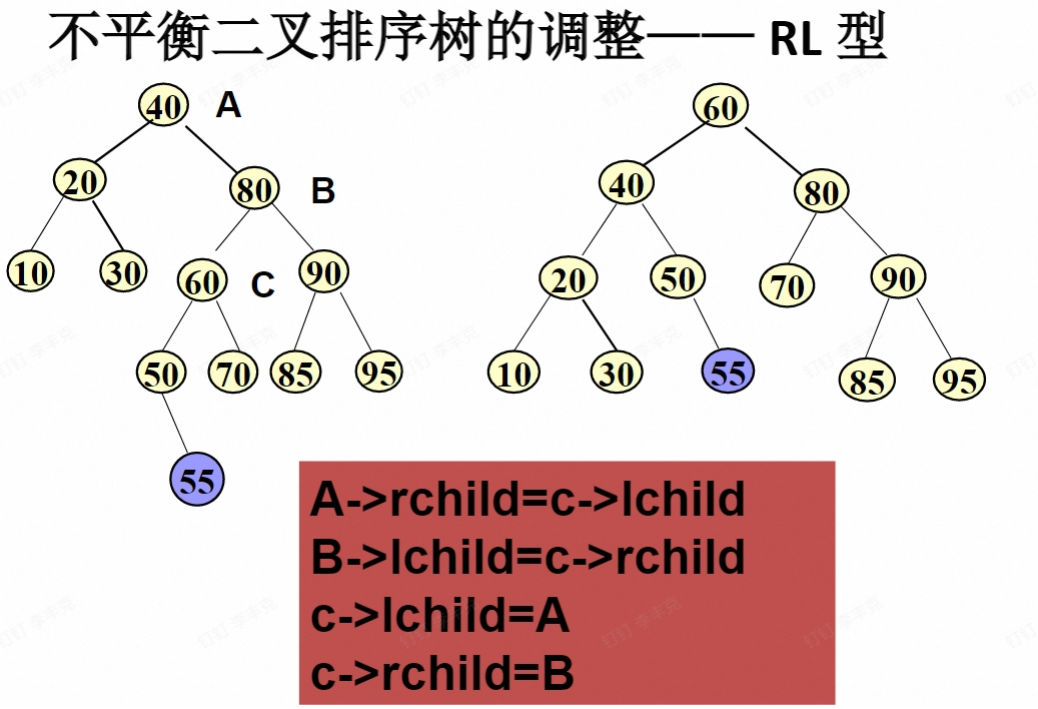


RL型

最低失衡点为A，如果在A的右子树B的左子树C上插入新结点S后失衡。

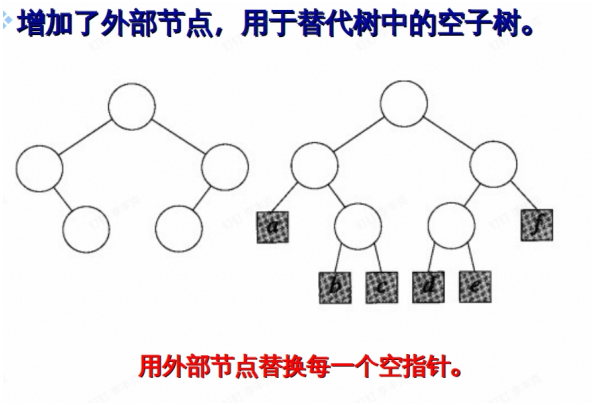
此时应该让C做根节点位置，B做C的右子（只保留B与右子链接），C原来的右子CR改为B的左子；A做C的左子（只保留左子链接），C原来的左子CL改为A的右子

（三层变一层，二变右，只剩右；一变左，只剩左；原三左补左，右补右）与LR型对称



扩充二叉树~~(没什么用)~~（非常有用）

增加了外部节点用于替代树中的空子树



5.3 红黑树

与AVL树对比：

AVL树要求完全平衡，过于严格，会影响性能

红黑树只要求局部平衡：懒汉平衡，不要求每个结点都必须平衡。

时间复杂度与AVL相同，统计性能比AVL树更好；任何不平衡都能在三次选择中解决；增删算法性能好、易于实现；减少了开销，　　性能几乎没有下降。

定义：

平衡的扩充二叉搜索树，且满足：

每个结点永远为黑色或红色

根节点永远为黑色

扩充的外部节点都是黑色的空结点，每个叶结点再补两个空黑色结点作为外部结点，或者有缺的结点也补一个。

红色结点的两个子结点都是黑色的，红色结点不能连续

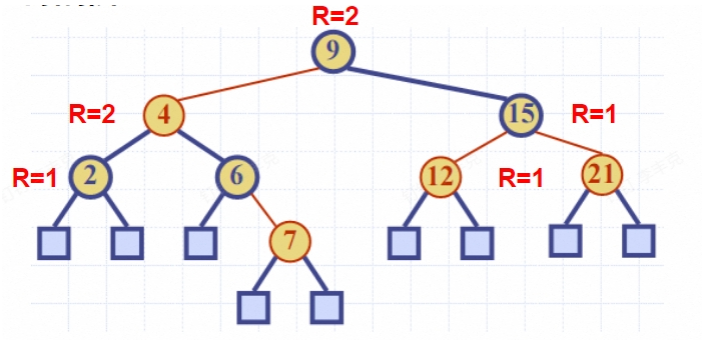
对于某个结点，从该结点到其所有子孙叶结点的路径中包含的黑色结点数目必须相同

每个结点的结构体定义包括颜色指针。

性质：

结点的阶：平衡性指标

为从这个结点到其子树中任意外部结点的任意一条路径上的黑色结点数量，不包括该结点本身，包括叶结点。



外部结点的阶为零

根节点的阶就是树的阶

设r是红黑树的阶，h是红黑树的高（不包括外部结点），n是内部结点的个数

$$r \le h \le 2r$$
$$n \ge 2^r - 1, \text{最少是满二叉树}$$
$$h \le 2\log_2 n + 1$$

红黑树搜索插入的时间复杂度都为O(logn)

插入（避免双红）：

先调用二叉排序树的插入算法，插入的地方为补充的外部结点，插入后再补上新的外部结点

把新结点着色为红色，如果父节点是黑色，就ok；

如果父亲结点是红色，进行双红调整：

（叔父结点：结点的父亲结点的兄弟结点）

当叔父结点为黑色：XYb

需要选择或同构，每个结点的阶保持原样。

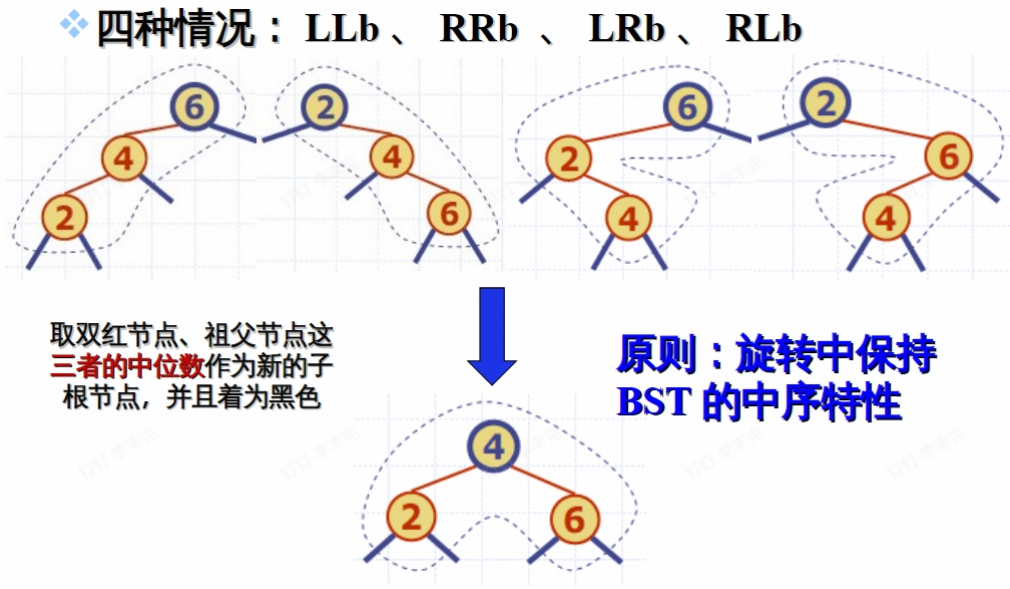
旋转

四种情况LLb，RRb，LRb，RLb

取双红结点以及新结点的祖父结点三者的中位数为新的根节点，并且着色为黑色。（另外两个均统一为红）

如果父结点为中位数，按照AVL树LL，RR情况旋转

如果新结点为中位数，按照AVL树LR，RL情况旋转



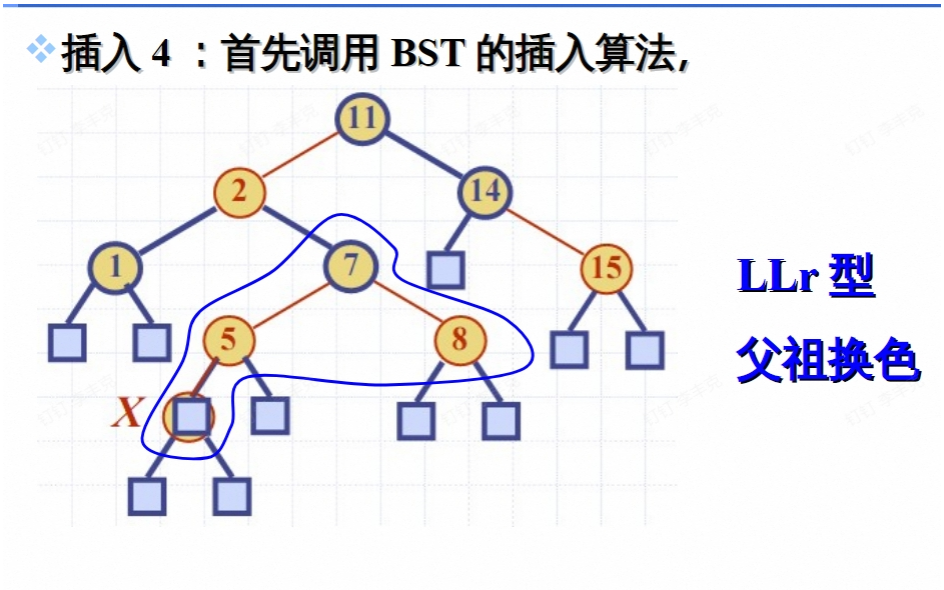
当叔父结点为红色：XYr

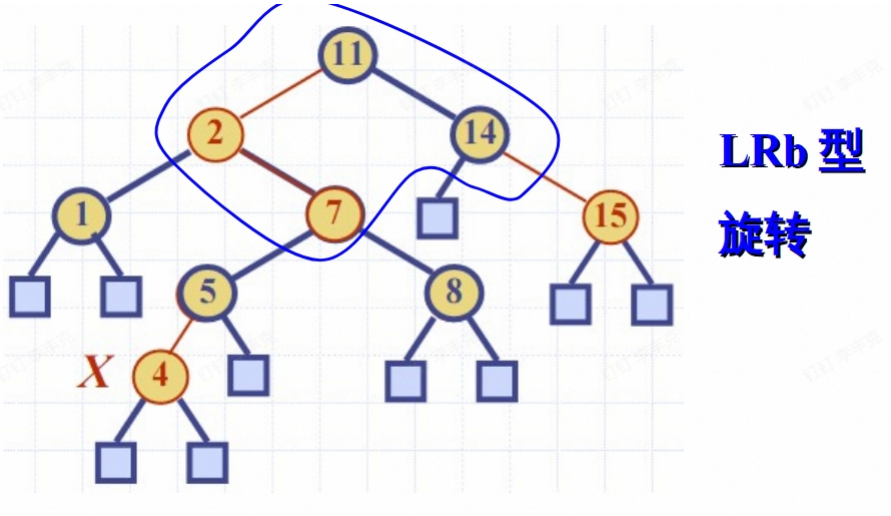
需要换色，换色后继续检查平衡

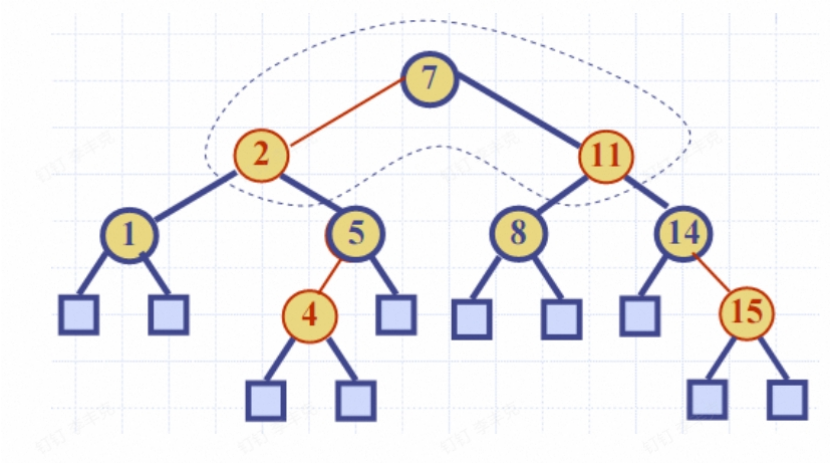
换色

四种情况：LLr，RRr，LRr，RLr

先父祖换色，将新结点的父亲结点与叔父结点改为黑色，祖父结点改为红色；（相当于父亲辈与祖父换颜色）再分析祖父结点是否存在双红现象，在判断是XYb还是XYr类型进行平衡。







先按二叉排序树插入，再看插入新结点的父结点是否为红色：

如果是黑色，没问题

如果是红色，看他的叔父结点是否为红色

如果是黑色，移动双红结点以及祖父结点，找中位数

中位数是二层，单旋转，中位数结点为黑色，其他两个为红色

中位数是三层，双旋转，中位数结点为黑色，其他两个为红色

如果是红色，叔父、父结点与祖父换色，再分析祖父结点是否双红，双红结点传递给祖父结点。

删除：

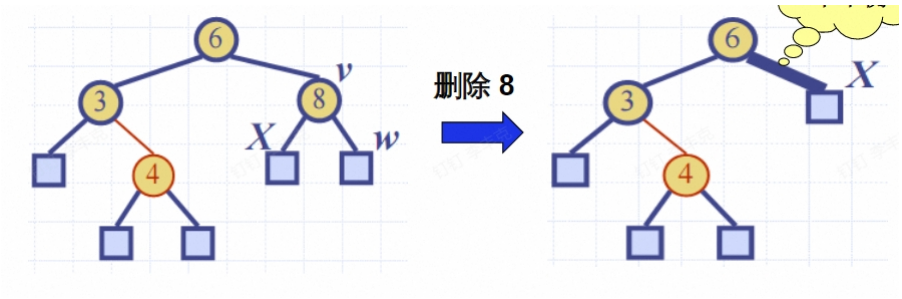
先进行平衡二叉树（BTS）删除的算法：

如果被删除的结点有一个或者两个外部结点，直接删除（就是平衡二叉树删除的前两种情况）

如果有两个非叶子结点，则在右子树中找到最小值结点，与这个结点交换，颜色不变（不变的是结点自身带的颜色，不是这个位置的颜色）。**删除的结点就传递到这个最小值结点，由此传递下来，任何一个删除操作，最终都会传递为删除一个有一个或两个外部结点的情况。**因此只需对这种情况进行分析。

双黑概念：

当删除的结点为黑色，代替他的结点也为黑色，那么这个子节点就会被标记为双黑。



我们要在删除完结点后解决双黑不平衡问题

以双黑结点是删除结点的左子结点为例（右子结点情况对称过来即可）

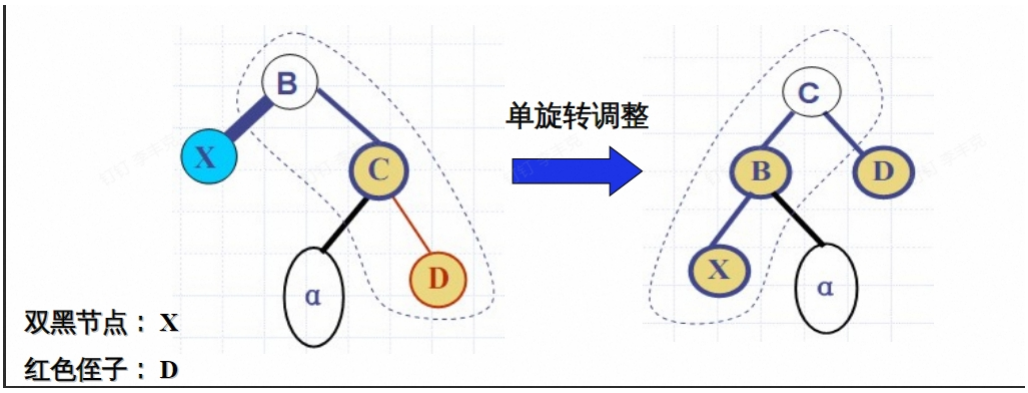
有三种情况：

双黑结点兄弟是黑的，兄弟结点的子节点右红色

第一种：双黑结点与他的红色侄子八字形外撇，离着远

单旋转调整：兄弟结点C提上去，继承原来父结点的颜色；父结点与红色侄子结点都改为黑色。

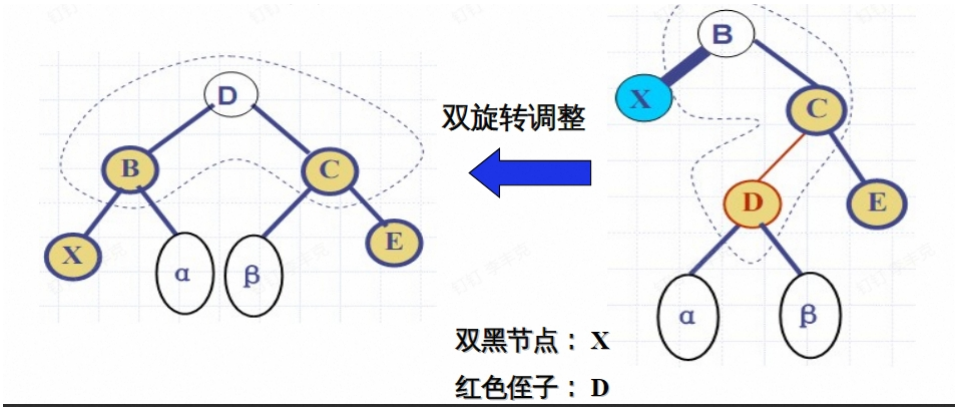
（其实就是AVL二叉树LL型与RR型的旋转操作，甚至连对应的位置都一样）（别忘了颜色变化）



第二种：双黑结点与他的红色侄子同边顺，就是离着近

双旋转调整：侄子结点提上去，继承原来父结点的颜色；父结点改为黑色。

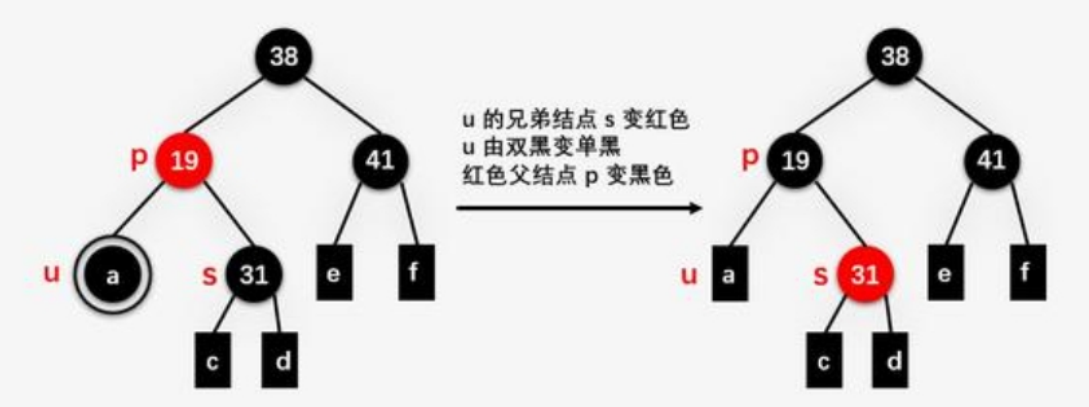
（还真是AVL二叉树LR型RL型的旋转操作，位置情况都一样）（改的色就是把侄子继承父结点颜色，父结点改为黑色）



双黑结点兄弟结点是黑的，且兄弟有两个黑色子节点

第一种：如果双黑结点的父结点是红色

只需将双黑结点的兄弟结点改为红色，父结点改为黑色即可



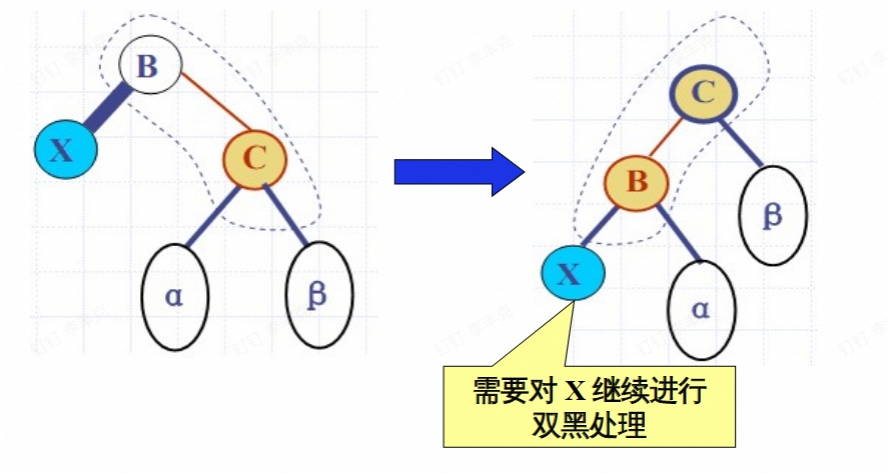
第二种：如果双黑结点的父结点为黑色

把兄弟结点改为红色，双黑结点传递到父结点，对父结点再进行去双黑处理。（开始递归了）

双黑结点的兄弟结点是红色

直接一个单旋转转变为情况1 或情况2

注意转完后把原父结点改色为红色，这个兄弟结点改色为黑色。但是双黑结点不变，还是他。



先看是不是双黑，再看双黑的兄弟是不是黑，

如果是红，单旋转再判断；如果是黑，看有没有红色儿子，

如果没有，看父结点是不是红的：

如果是，就父祖换色；如果不是就兄弟改红，双黑传递到父结点

如果有，看侄子结点是靠近双黑还是远离双黑；

（与AVL单旋转一样）如果远离就单旋转，二变一，兄弟继承父结点颜色，父结点和红侄子变黑

（与AVL双旋转一样）如果靠近就双旋转，三变一，侄子继承父结点颜色，父结点变黑

红黑树平均和最差检索的时间复杂度O（log2 n）

5.4 B-树

二叉搜索树只适合内存中组织较少的索引，对于存放在外存中较大的文件系统，引入高度数搜索树。

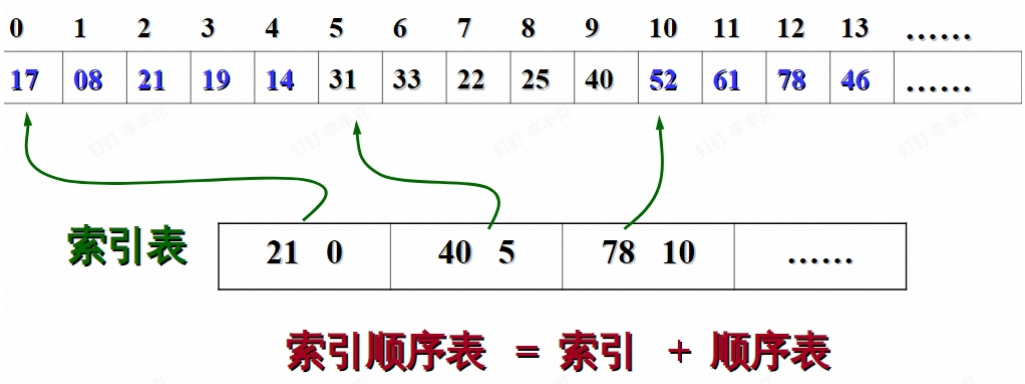
（在面对特别复杂的文件系统中，二叉搜索树往往不能胜任，因此要引入更高度数的搜索树。）

5.4.1 索引顺序访问方法

建立顺序表时建立一个索引项，包括两项：关键字项和指针项

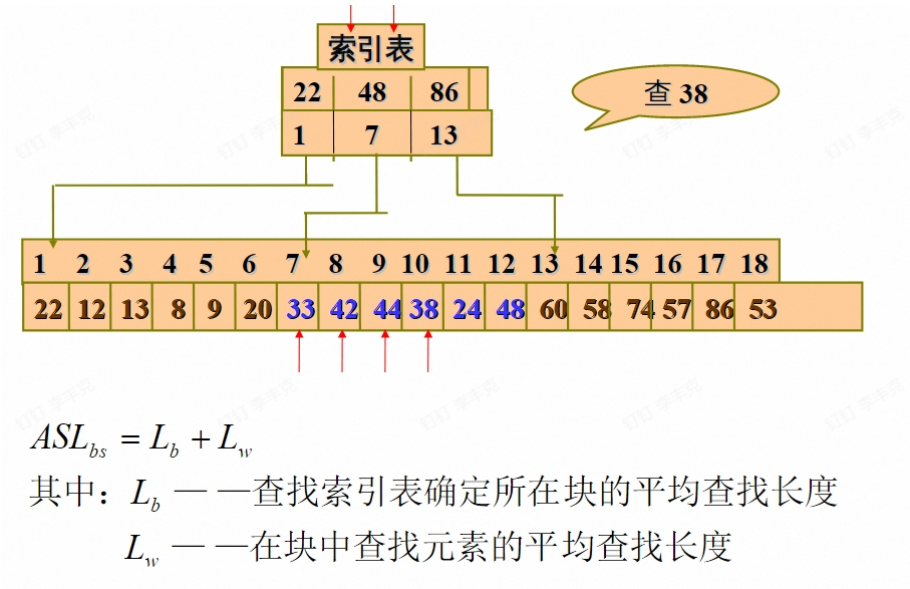
其中索引项按关键字有序，顺序表分块有序

将表分成若干块，这个块里的最大值作为索引表对应位置的关键字，索引表对应位置指针指向所在块的第一位，在索引表上，这些块上的最大值呈现有序排列。



查找过程

将表分成几块，块内无序，块间有序。先确定待查记录所在块，再在块内查找。（有点像概统的分布函数）



$$ASL_{bs} = L_b + L_w$$

其中：L_b ——查找索引表确定所在块的平均查找长度

L_w ——在块中查找元素的平均查找长度

应用

磁盘的顺序索引访问方法：

数据组织形式：磁盘空间被分成很多块，字典元素以升序存储在块中。

顺序访问：依次在每个块中按升序访问元素。

随机访问：建立索引表，索引中关键值数量与块数相同，包含每个块的最大关键值。对于任意要访问的值先找到包含这个块的索引，再在块内遍历即可

跨磁盘字典存储模式：不仅每个磁盘中有块索引，还有磁盘索引，保存了磁盘中的最大值。要访问值时按磁盘索引找到在哪个磁盘，再按磁盘中的块索引找到在哪个块，再遍历。

优点是索引快，缺点是插入和删除时在块之间移动元素代价高。

二叉搜索树只适合内存中组织较少的索引，对于存放在外存中较大的文件系统，引入高度数搜索树

5.4.2 m叉搜索树

1）每个结点最多可以含有m个子女和（m-1）个元素，外部结点不含元素和子女（一个结点可以含有多个元素）

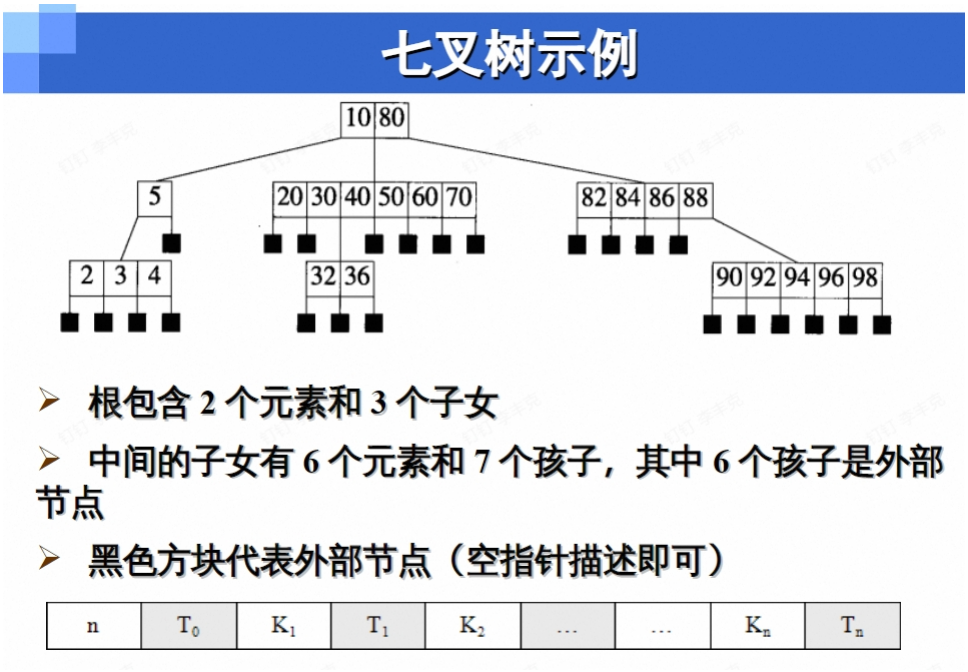
2）每个含有p个元素的结点，含有p+1个子女，（子女中有多少个元素不管）

3）设k1...kp是p个元素的关键值，有k1<k2<...<kp；c0...cp是起p+1个子女

c0为根的子树中的元素的关键值都小于k1

cp为根的子树中的元素的关键值都大于kp

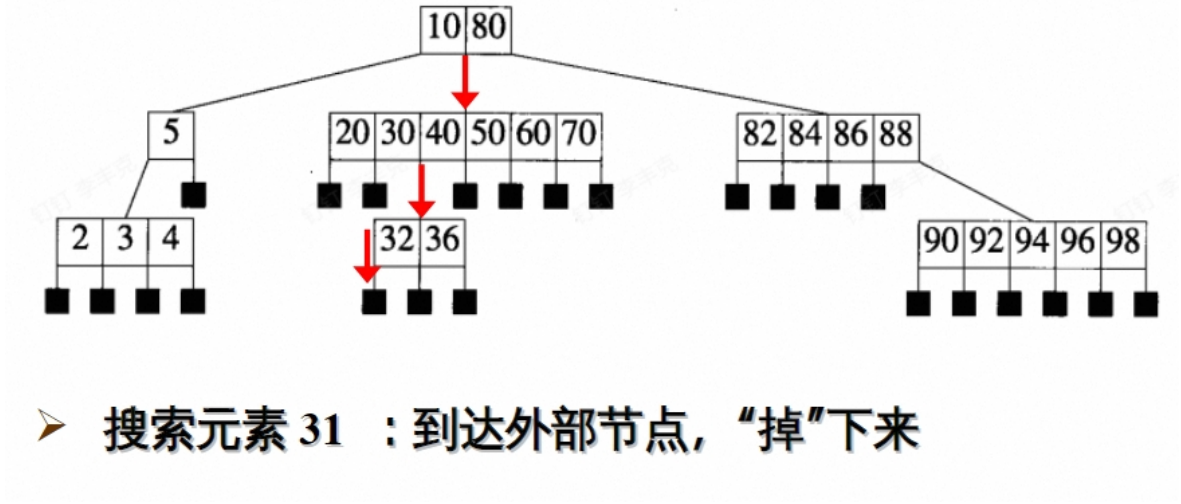
以ci为根的子树中的元素的关键值大于ki小于ki+1



(太棒了这个图画的，子女结点穿插在父结点元素之间，正好展现出对应子结点元素值处在两个父结点元素间的情况)

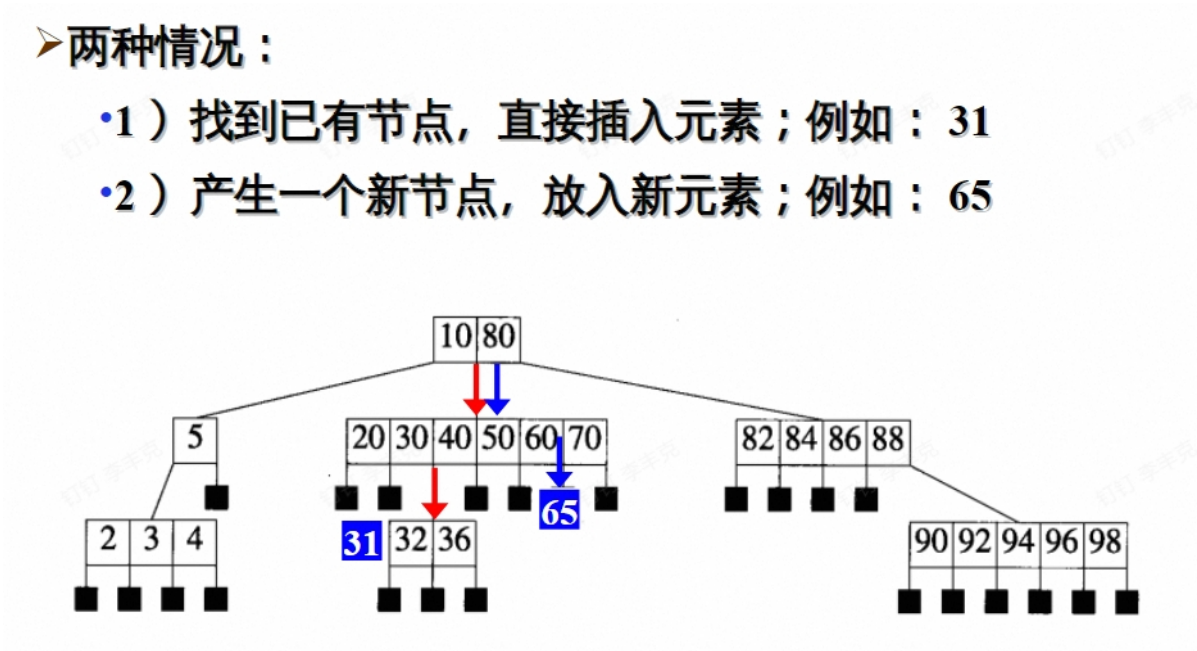
（这么一看二叉搜索树真是m叉搜索树最简单一种情况）

搜索



一层层夹出来

插入

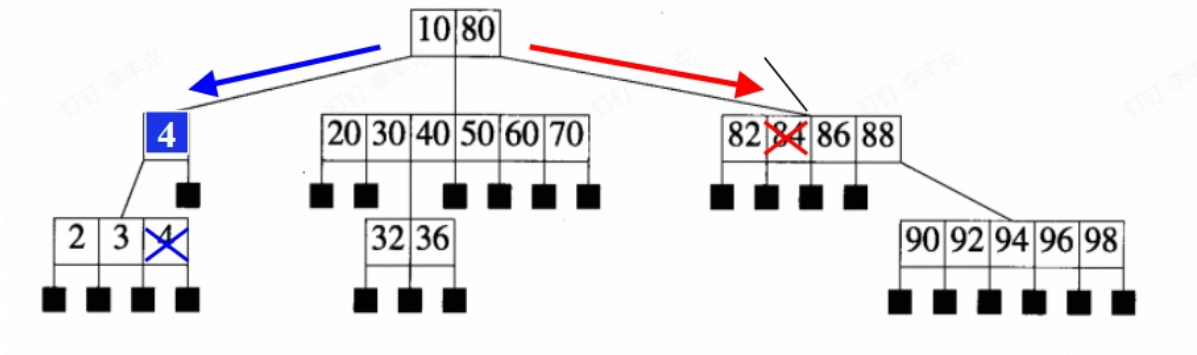


一层层向下顺延对比

删除

找到已有节点，直接删除对应元素

如果这个节点只有一个元素，那么就在其非空相邻子树中找到一个对应的最大或者最小值代替，如果没子树，删了就删了。



高度

一棵高度为h的m叉搜索树最少有j个元素，最多有m^h-1个元素；

一棵结点树为n的m叉搜索树高度在$log_m(n+1)$到n之间；

m叉平衡搜索树确保高度值接近$log_m(n+1)$

m叉平衡搜索树有上限，无下限。

5.4.3 m序 B-树

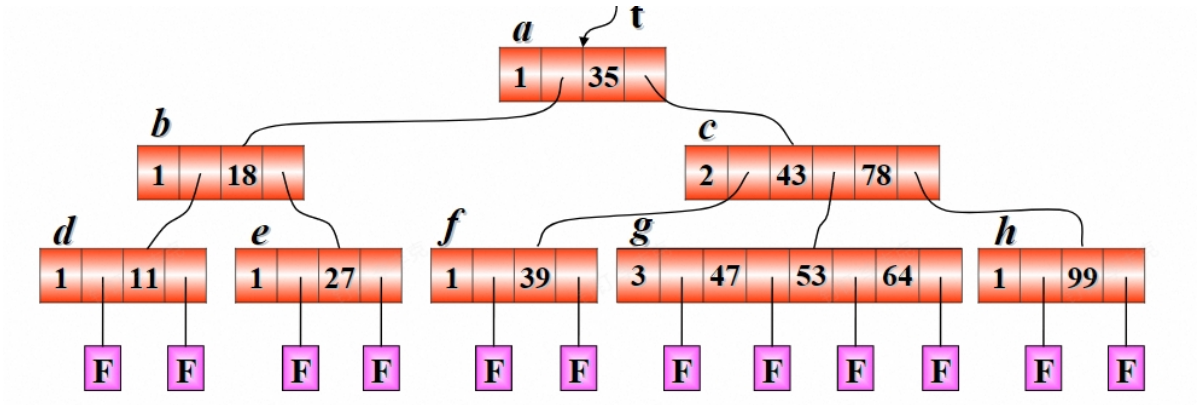
m序B-树是一棵m叉搜索树，满足

1）根节点至少有两个孩子

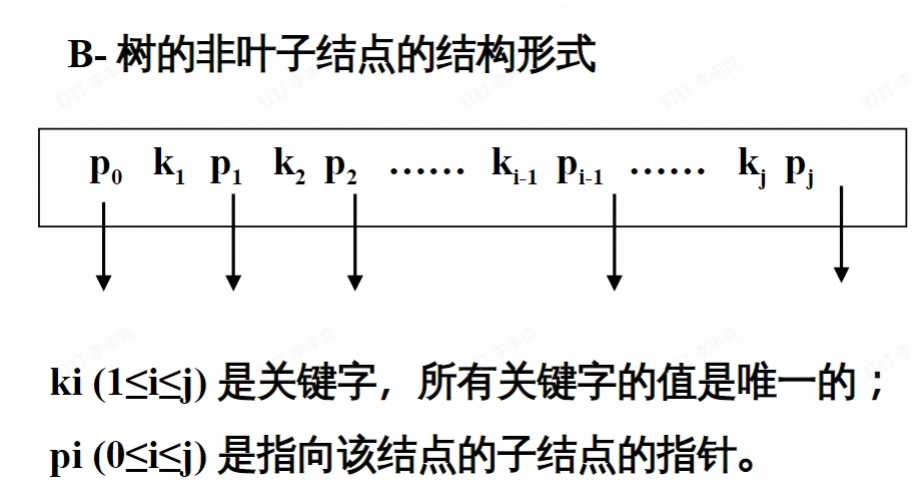
2）除了根节点外，所有内部节点至少有$\lceil \frac{m}{2} \rceil + 1$个孩子（注意是孩子节点的个数，不要和元素个数搞混）

这就说明了每个节点包含的元素树$\lceil m/2 \rceil \le k \le m-1$，例如7序的内部节点元素数为3~6。

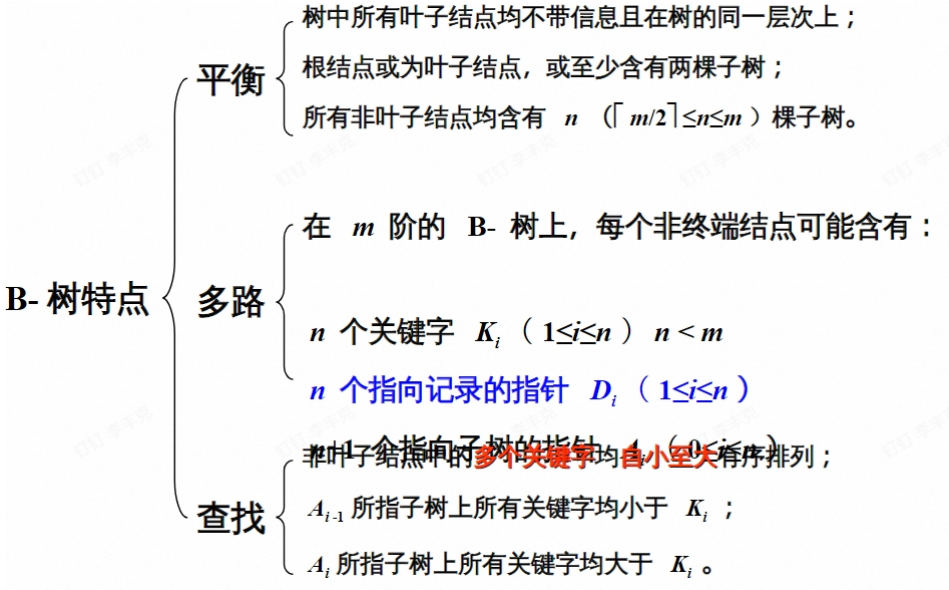
3）所有外部节点处在同一层上。



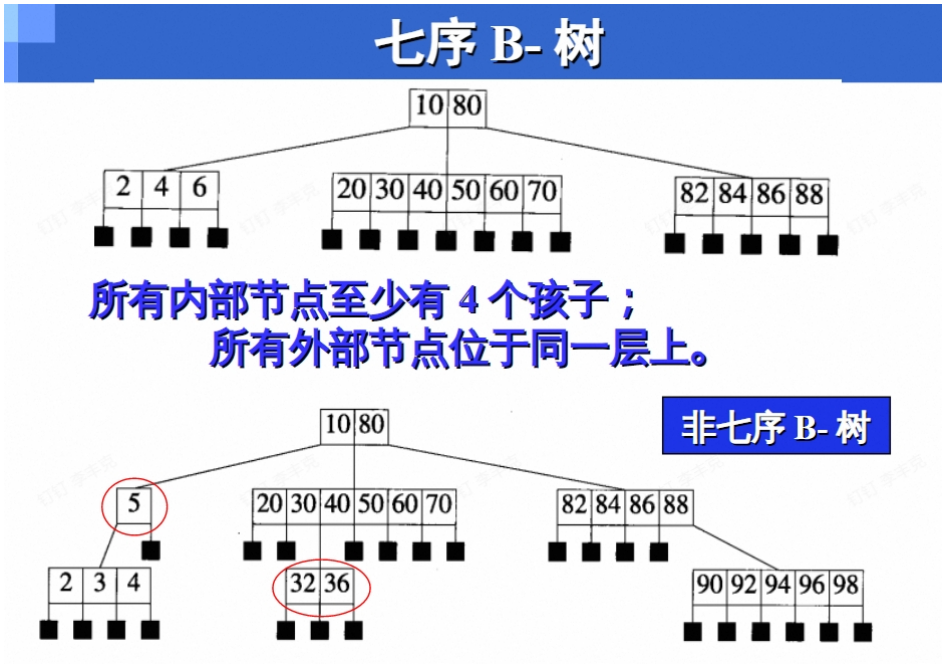
B-树非叶子结点的结构，指针与元素穿插排列



B-树的美德：（应该不考）



B-树的例子



特殊的B-树

二序B-树：一棵满二叉树

三序B-树：内部节点可以有2或3个孩子，称23树

四序B-树：内部节点可以有2或3或4个孩子，称234树

5.4.4 B-树的高度

设T是一棵高度为h的m序B-树，d=[m/2]+1（比如是5序，d=[5/2]+1=3）且n是T中的元素个数，则有

$2d^{h-1}-1 \leq n \leq m^h-1$

$log_m(n+1) \leq h \leq log_d(\frac{n+1}{2})+1$

一棵高度为3的200序B-树至少可存放$2 \times 100^2-1$个元素。2层5序B-树，至少有$2 \times 3^1-1=5$个元素

3层5序B-树，至少有$2 \times 3^2-1=17$个元素

B-树的搜索与m叉搜索树的应用，磁盘访问次数最多是(高度)h。

5.4.5 B-树的插入

先要检查是否有相同值存在，B-树不允许重复值存在。

5.4.5.1 如果插入的节点没有饱和，直接插入

按照m叉搜索树插入

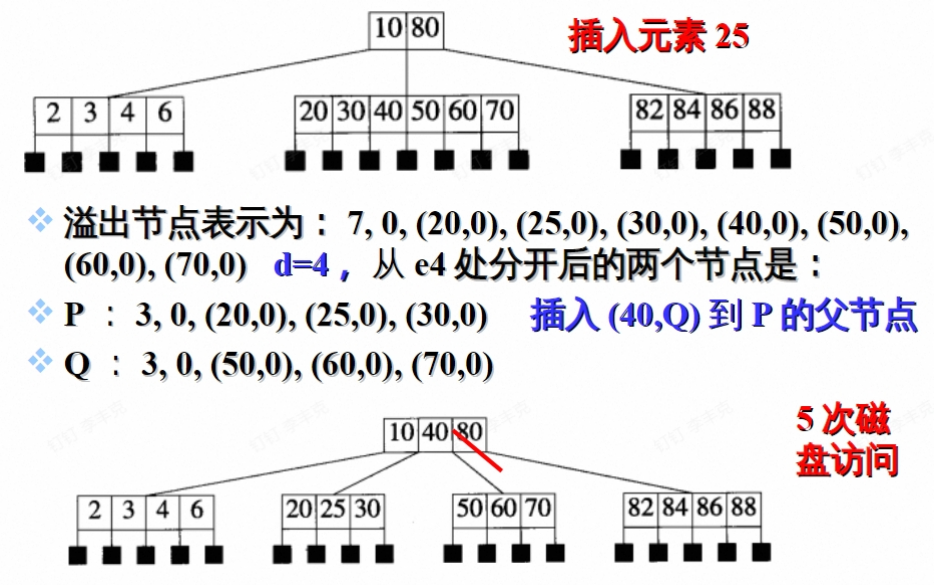
5.4.5.2 如果插入的节点已经饱和，即原来元素数已经达到了m-1，需要分裂节点

设P是饱和节点，把新元素插进去，得到一个有m个元素和m+1和孩子的溢出节点

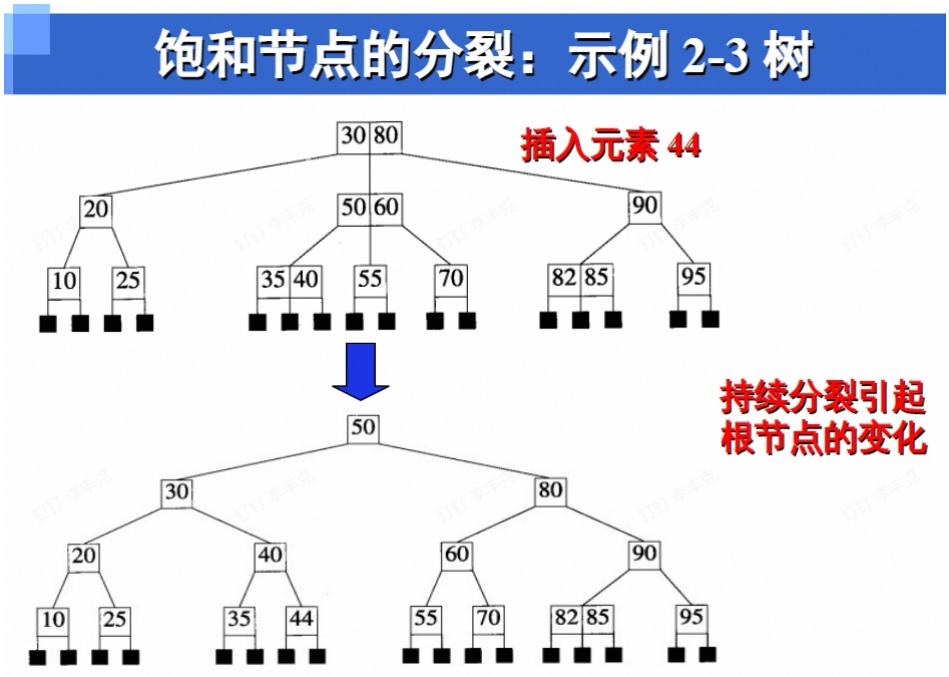
对于新得到的溢出节点，

如果m是奇数，那么这m个元素有一个中位数，左右可均分，把这个中位数插入到父结点中，中位数左边的作为左子节点，右边的作为右子节点（别忘了分成两个节点，别只移走数就不管了）。

如果m是偶数，那么这m个元素的中位数将是个指针，不用移出数就能均分，直接均分，不必再向父结点插入新数。



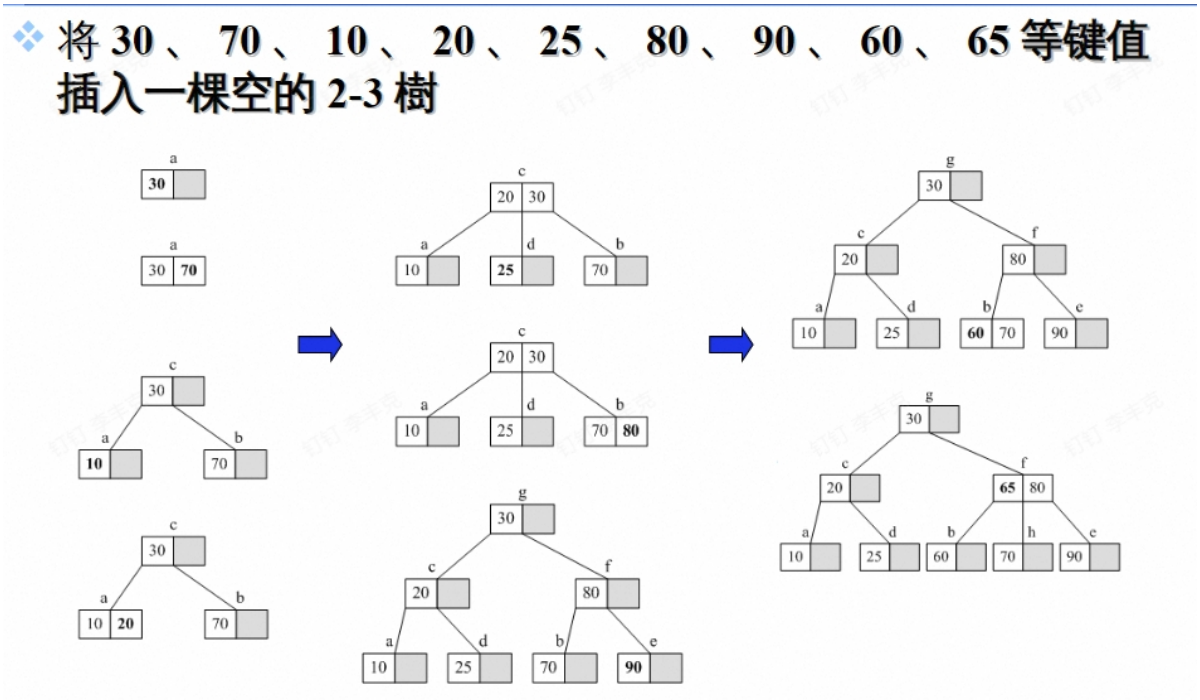
插入后再观察父结点是否变为溢出节点，如果是继续分裂，不是就算了。



当插入引起了s个节点的分类，磁盘访问次数变为h（读搜索路径）+2s（回写两个新节点）+1（回写新的根节点）=h+2s+1次

最多可达3h+1次（s=h）

5.4.5.3 B-树的生长过程



先按m叉搜索树那么插入，看看插入的结点是否饱和

没饱和，直接插入。

饱和了，看插入过后结点含有的元素数目m是奇数还是偶数

是奇数，取出溢出结点的中位数插入父结点相应位置，两侧均分为左子节点和右子节点。看父结点插入后是否溢出

是偶数，直接均分，裂开分成左子节点与右子节点。

5.4.6 B-树的删除

删除元素的节点孩子都为外部结点，也就是叶结点

删除元素在非叶子节点上，可以用该元素的左相邻子树最大元素或者右相邻子树的最小元素来替代，这样就把删除传递给其子树上，都转化为删除叶子节点的问题。

5.4.6.1 当这个叶节点的元素数目大于[m/2]，也就是允许的最小数目

直接删就行，删完了再写回去

5.4.6.2 当这个叶节点元素数目等于允许的最小数目，但是兄弟至少有一个不是

可以从左右相邻兄弟节点中取出一个元素来替代他。注意选取的节点包含的元素数应该大于[m/2]。

调整步骤：如果是取的左兄弟的元素

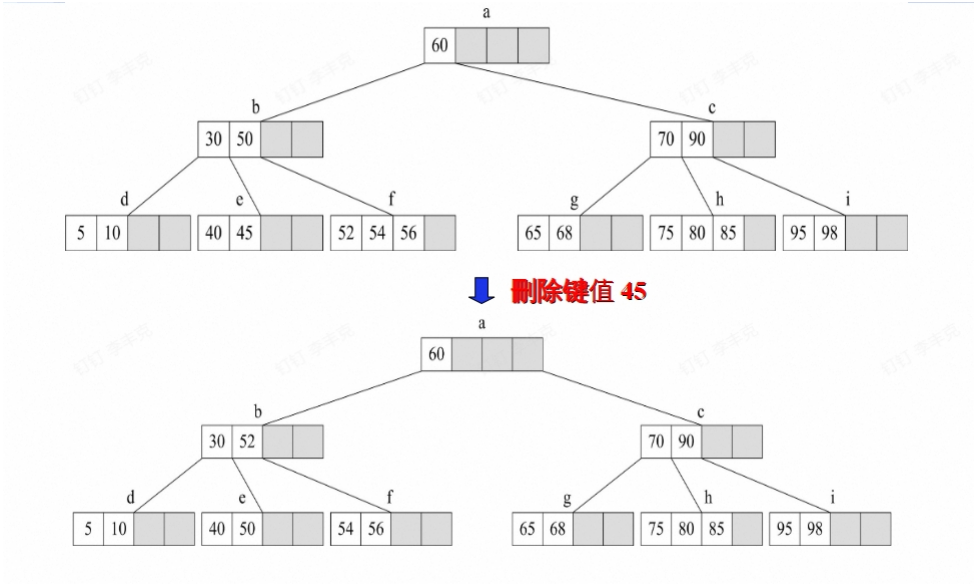
1）将双亲结点中刚好小于删除值的元素移下来；

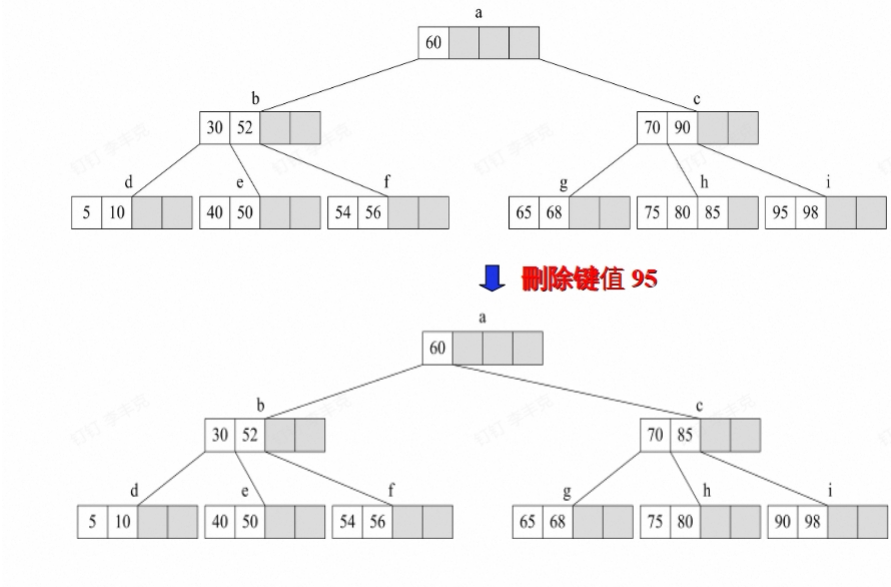
2）把左兄弟中最大值元素移上去补位；

3）左兄弟结点中最右侧的指针（孤单的指针）移动到被删除值的结点最左侧子树指针；

4）再调整其左兄弟节点指针，并且其元素数-1。

如果取得是右兄弟指针，左右大小对称操作就行。（就近补位：就是在左兄弟中找最右的，右兄弟中找最左的）





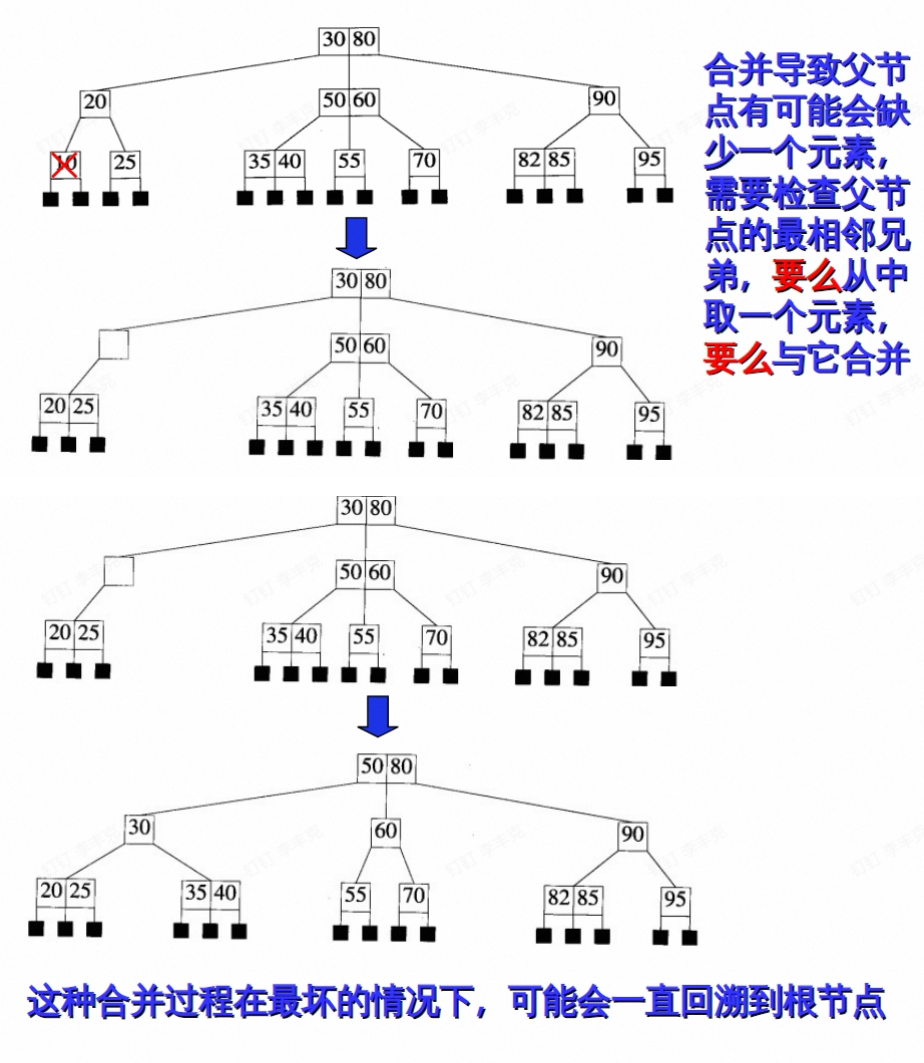
5.4.6.3 当这个节点元素数目达到最低，相邻兄弟也是最低

选定一个相邻兄弟，把删除元素结点与这个兄弟节点之夹的父结点上的元素拉下来，这几个元素合并成一个新结点。这时把删除节点问题传递给了父结点，如果父结点本来也是最小数目，那么再看父结点的相邻兄弟节点有没有救济的，进行移动或者再合并。

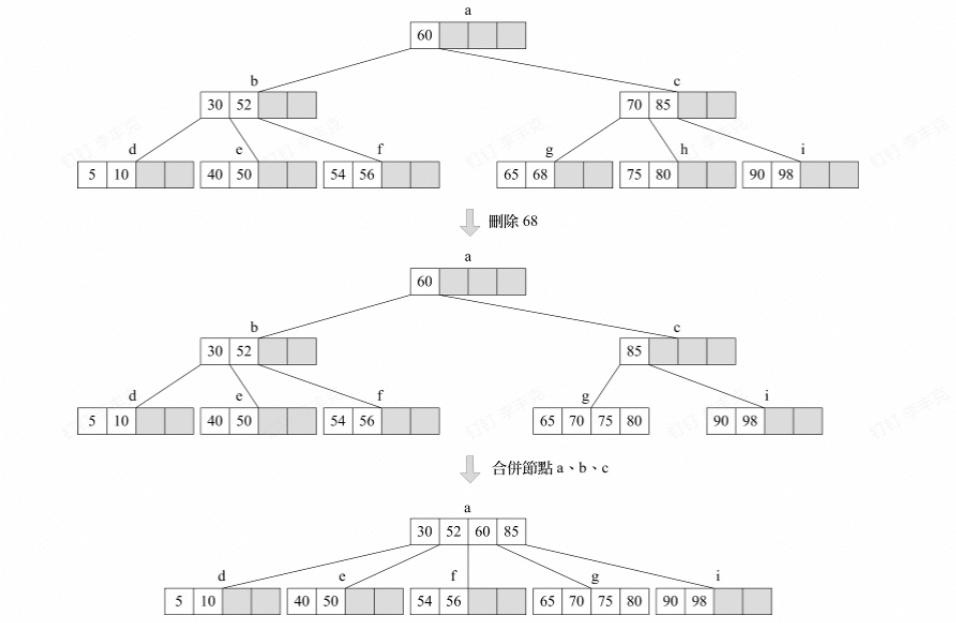
这种一层层的传递最终可能会传到根结点那里。注意到非叶结点的移动，指针的跟随与平移。

注意合并后的指针到了谁与谁之间，相邻兄弟救济时，兄弟的指针跟随过来。

指针的位置可以根据子树与值的关系来判断

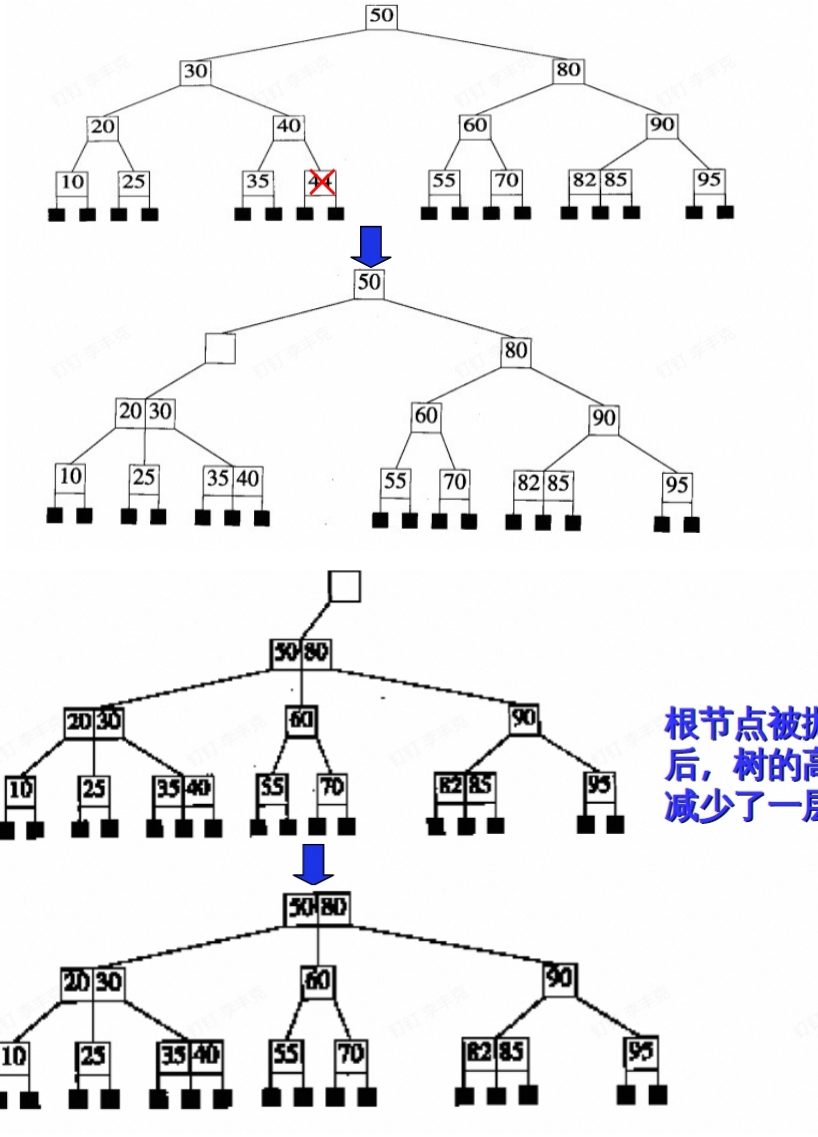


如果一直传递到根节点且把根节点拉成不含关键值的了，那么就删除原来根节点，合并后的根节点作为新的根节点。



移动时就跟拼积木一样，该是谁夹谁的指针，改了之后相对位置也不变

最坏情况：波及到根节点，高度之间少了一层



先看删除的是否为叶节点，

不是，用这个元素的左相邻子树上最大元素或者右相邻子树上最小元素替代，删除操作传递到子树上，直到传到叶结点。

是，看这个节点元素数目是否大于允许的最小元素数[m/2]

大于，直接删就行

恰好等于，看看这个节点左右相邻兄弟是否存在元素数目大于[m/2]的。

存在，看看这位好兄弟是左兄弟还是右兄弟：

左兄弟，父结点中恰好小于删除值的元素移下来，左兄弟的最大值移上去。

右兄弟，父结点中恰好大于删除值的元素移下来，右兄弟的最小值移上去。

不存在，随机挑一个相邻好兄弟，把这个节点与相邻兄弟节点之间夹的父结点上的元素拉下来合并成一个全新节点，删除问题就传递给了父结点。

一定要注意挪动元素时对应的指针跟随移动。

5.4.7 B+树和B*树

B+树

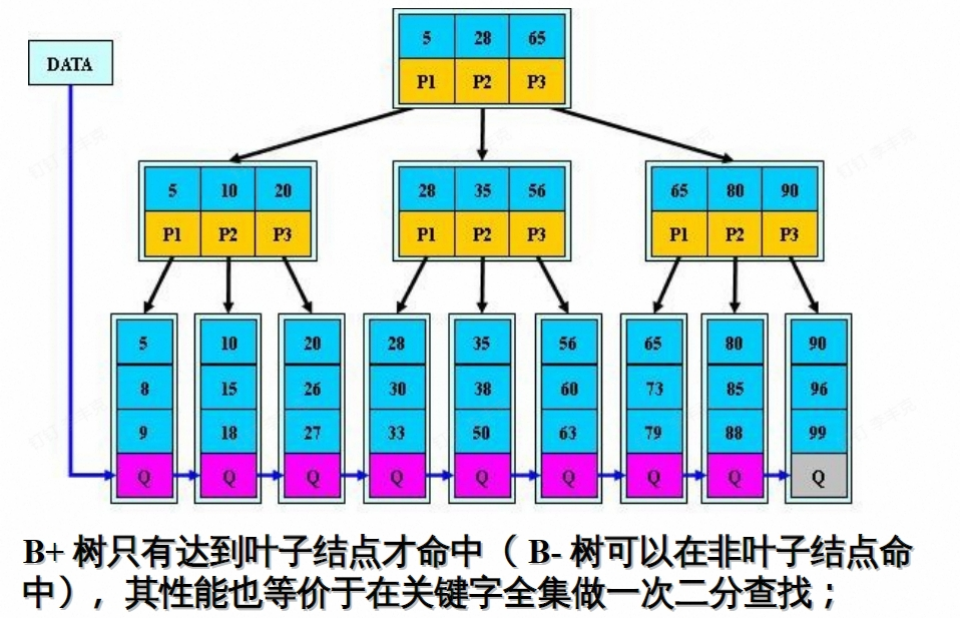
B+树是B-树的一种变形

B+树基本定义与B-树一样。区别有：

1）B+树非叶子节点的关键值数目与指针数目一样，关键值ki对应的子树的值满足$k_i \leq e < k_{i+1}$

2）给所有叶子节点加了一个链指针

3）所有关键字都在叶子节点出现



特性：

1）所有关键字都在叶子节点的链表中（稠密索引），且恰好是有序的

2）不可能出出现在非叶子节点

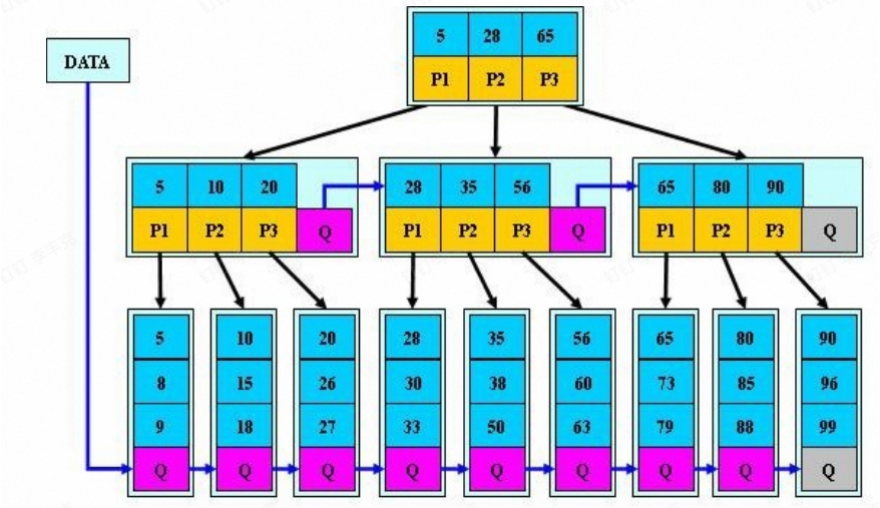
3）非叶子节点相当于叶子节点的索引（稀疏索引），叶子节点是数据层

4）更适合文件管理系统，比如mysql数据库

B*树

在B+树的基础上，非根非叶子结点再加一个指向兄弟的指针，增加空间利用率。

是 B+ 树的变体，在 B+ 树的非根和非叶子结点再增加指向兄弟的指针，增加空间使用率



对比：

B树：二叉搜索树，每个结点只存一个关键字，等于就是命中，小于走左，大于走右

B-树：多路搜索树，每个结点存[m/2]到m-1个关键字。非叶子结点指针指向范围，相当于判断区间层层收缩，最终锁定到最小区间，性能更好。非叶子节点可以命中

B+树：在B-树基础上，为叶子增加链表指针，只能在叶子节点命中，非叶子节点为叶子节点的索引

B*树：在B+树基础上，为非叶子节点增加指向兄弟的链表指针，提高节点利用率。

第六章 查找与排序

6.1 查找

关键字：每个对象有若干属性，其中有一个属性，能够唯一的标识这个对象，称为关键字。使用基于关键字的搜索，查找结果是唯一的。

6.1.1 顺序查找

依次遍历对比。

查找算法的平均查找长度ASL：给定值进行关键字比较的次数的期望

顺序查找时间复杂度O(n)

查找长度：

成功：最少比较次数1，最多比较次数n，平均比较次数（n+1）/2

失败：最少比较次数n，最多比较次数n，平均比较次数n

算法简单，但是效率太低了

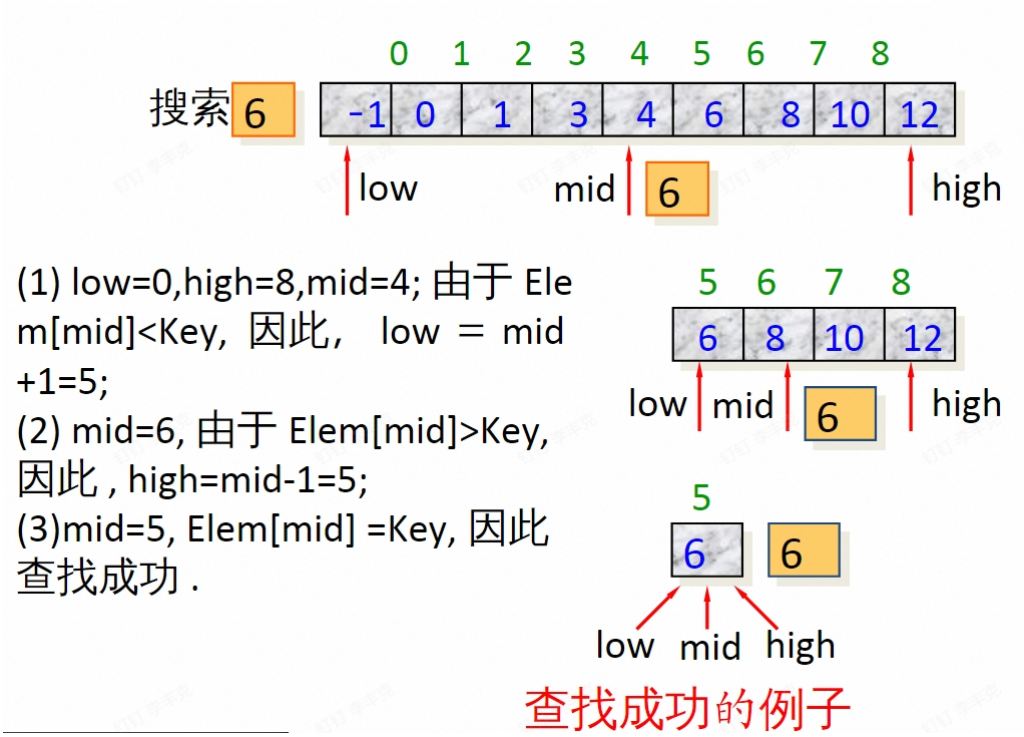
6.1.2 折半查找（二分查找）

如果静态查找表是有序表，可以进行折半查找

每次先求出查找区间正中对象的下标mid，用其关键字与给定值x比较，然后根据比较结果将查找区间缩小一半，直到找到查找对象。

假设是升序：

x=MID，成功；x<MID，区间缩小为前半部分；x>MID， 区间缩小为后半部分。



查找成功：$ASL_{suss}=\frac{n+1}{n}log_2(n+1)-1$

时间复杂度$O(log_2n)$

查找效率高，但是查找结构有限制，插入删除操作困难。

6.1.3 二叉搜索树

二叉搜索树，平衡二叉树，B-树 都看上一章的搜索树

二叉搜索树，平衡二叉树其时间复杂度都是$O(log_2n)$

B-树时间复杂度是$O(log_mn)$

6.2 哈希表

哈希查找根本思想：在记录的存储地址和它的关键字之间建立一个确定的对应关系，不用经过比较，一次存取就能读出。

哈希函数：在记录的关键词与关键字的位置之间建立一个函数关系，以f(key)作为关键字key的位置。称f为哈希函数

哈希地址：由哈希函数求出的记录存储位置称为哈希地址，表示成addr(ai)=f(ki)

ai是表中的一个元素(记录)；addr（ai）是ai的存储地址；ki是ai的关键字

哈希表：应用哈希函数，由记录的关键字确定记录在表中的地址，并将记录放入此地址，这样的表叫做哈希表（散列表）

哈希查找：又叫散列查找，利用哈希函数进行查找的过程

哈希函数是一个从关键字集合到一个地址集合的映射

由于哈希函数是一个压缩映像，一般情况下，很容易产生冲突现象。即f(key1)=f(key2)

很难找到一个不冲突的哈希函数，只能尽量减少冲突或者冲突时解决它。

6.2.1 构造哈希函数的几种方法

有 直接定址法，数字分析法，平方取中法，折叠法，除留余数法，随机数法

如果是非数字关键字，要先对其进行数字化处理

6.2.1.1 直接定址法

构造：取关键字或者关键字的某个线性函数作为哈希函数

$$H(key) = key$$
$$或者 H(key) = a \times key + b$$

特点：直接定址法所得的地址集合与关键字集合大小相等，不会发生冲突。

实际能用这种哈希函数的情况很少。

6.2.1.2 数字分析法

构造：对关键字进行分析，取关键字的若干位或其组合作哈希函数

特点：适于关键字位数比哈希地址位数大的，且可能出现的关键字事先知道的情况

6.2.1.3 平方取中法

构造：以关键字的平方值的中间几位作为存储地址。求关键字的平方值的目的是扩大差别，同时平方值的中间各位又能受到整个关键字中各位的影响

特点：关键字中的每一位都有某些数字重复出现频度很高的情况

6.2.1.4 折叠法

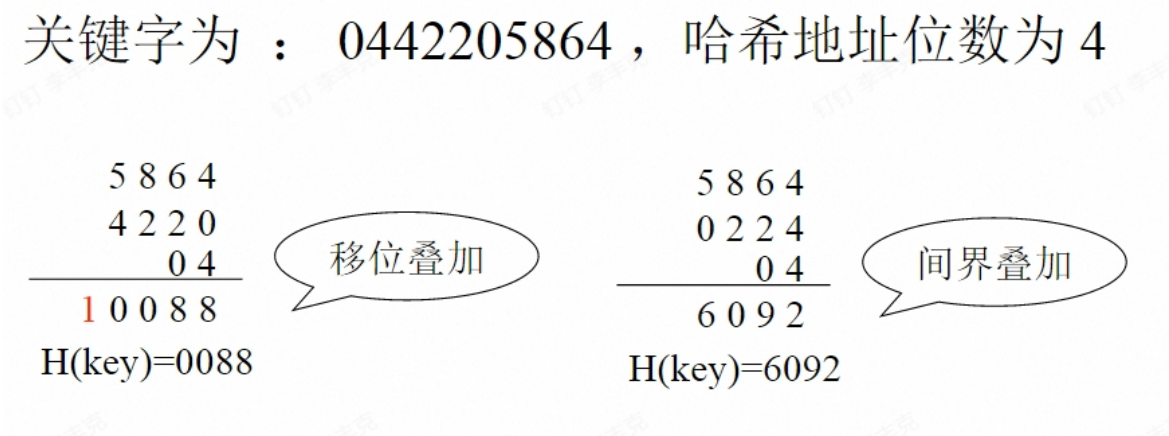
构造：将关键字分割成位数相同的几部分，然后取这几部分的叠加和（舍去进位）作为哈希地址。

种类：

移位叠加：将分割后的几部分低位对齐相加

间界叠加：从一端沿分割界来回折送，然后对齐相加。

特点：适于关键字位数很多，且每一位上数字分布大致均匀的情况



移位：按哈希位数分关键字，右对齐相加，再去后面所需位数

间界：一条龙从最右端开始，右到左，拐过来到右，再拐回去。

6.1.2.5 除留余数法

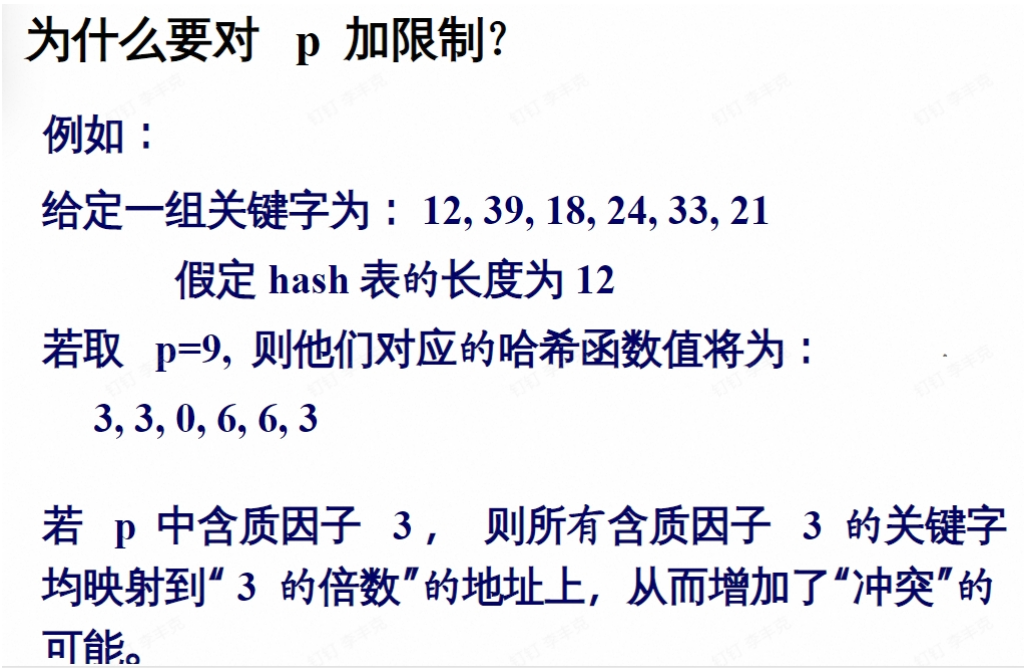
构造：取关键字被某个不大于哈希表表长m的数p除后得到的余数作哈希地址。

即H（key）=key%p

p<=m且p一般应为接近m的素数或是不含20以下的质因子

特点：简单、常用：可以与上述几种方法结合使用

p的选取很重要，p选的不好会产生同义词

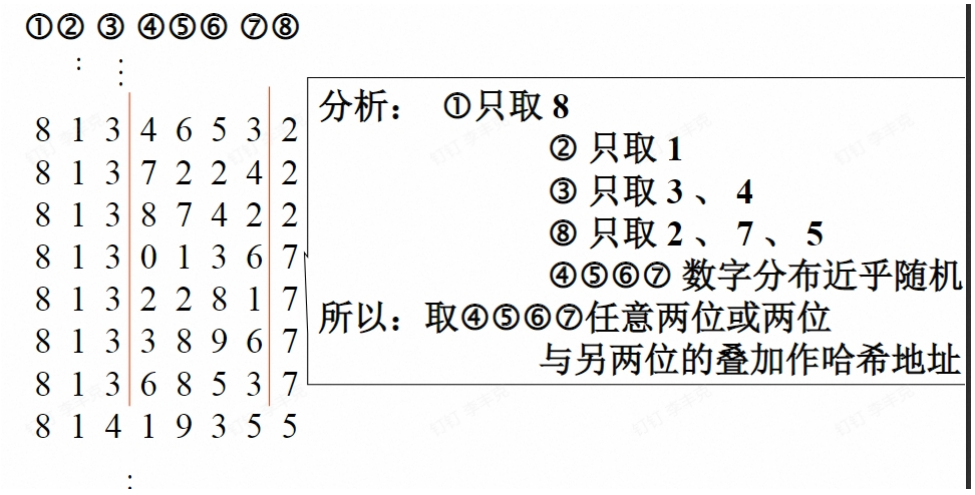


6.1.2.6 随机数法

构造：取关键字的随机函数值作哈希地址H（key）=random（key）

适用于关键字长度不等的情况

实际造表时，采用何种构造哈希函数的方法取决于建表的关键字集合的情况(包括关键字的范围和形态)，总的原则是使产生冲突的可能性降到尽可能地小。



6.2.2 处理冲突的方法

为产生冲突的地址寻找下一个哈希地址。

两种方法：开放定址法，链地址法。

6.2.2.1 开放定址法

思想：为产生冲突的关键字寻找一个新的地址Hi(key)，求得一个地址序列：

m是表长。

$$H_0, H_1, \dots, H_S \quad 1 \le s \le m-1$$
$$\text{其中：}\ H_0 = H(key)$$
$$H_1 = (H(key) + d_1)\%m$$
$$H_2 = (H(key) + d2)\%m$$
$$\dots$$
$$H_i = (H(key) + d_i)\%m$$
$$i = 1, 2, \dots, s$$

就是对冲突地址加一个增量再对表长取余数（其实就是用不同的方式挪到表的其他位置，再看是否冲突）

其中增量di的取值有三种取法：

线性探测再散列：$d_i = 1, 2, 3, \dots, m-1$，相当于从原冲突地址位数开始，向正方向一个个遍历看是否冲突，到底的时候再从开头遍历。

二次探测再散列：$d_i = 1^2, -1^2, 2^2, -2^2, \dots, k^2, -k^2$,相当于向前看一位，向后看一位，再向前看4位，向后看4位等等。

伪随机探测再散列：d_i =伪随机数列

增量di应该具有完备性，即产生的Hi均不相等，且所产生的s个Hi值能覆盖哈希表中所有地址。

要求平方探测时的表长m必须为形如4j+3的素数，如7，11，19，23等等。

随机探测时m和di没有公因子。

例　表长为 11 的哈希表中已填有关键字为 17， 60， 29 的记录，
H(key)=key % 11, 现有第 4 个记录，其关键字为 38，
按三种处理冲突的方法，将它填入表中

0	1	2	3	4	5	6	7	8	9	10
			38	38	60	17	29	38		

线性探测

(1) H(38)=38 % 11=5　冲突
H1=(5+1) % 11=6　冲突
H2=(5+2) % 11=7　冲突
H3=(5+3) % 11=8　不冲突

二次探测

(2) H(38)=38 % 11=5　冲突
H1=(5+1²) % 11=6　冲突
H2=(5-1²) % 11=4　不冲突

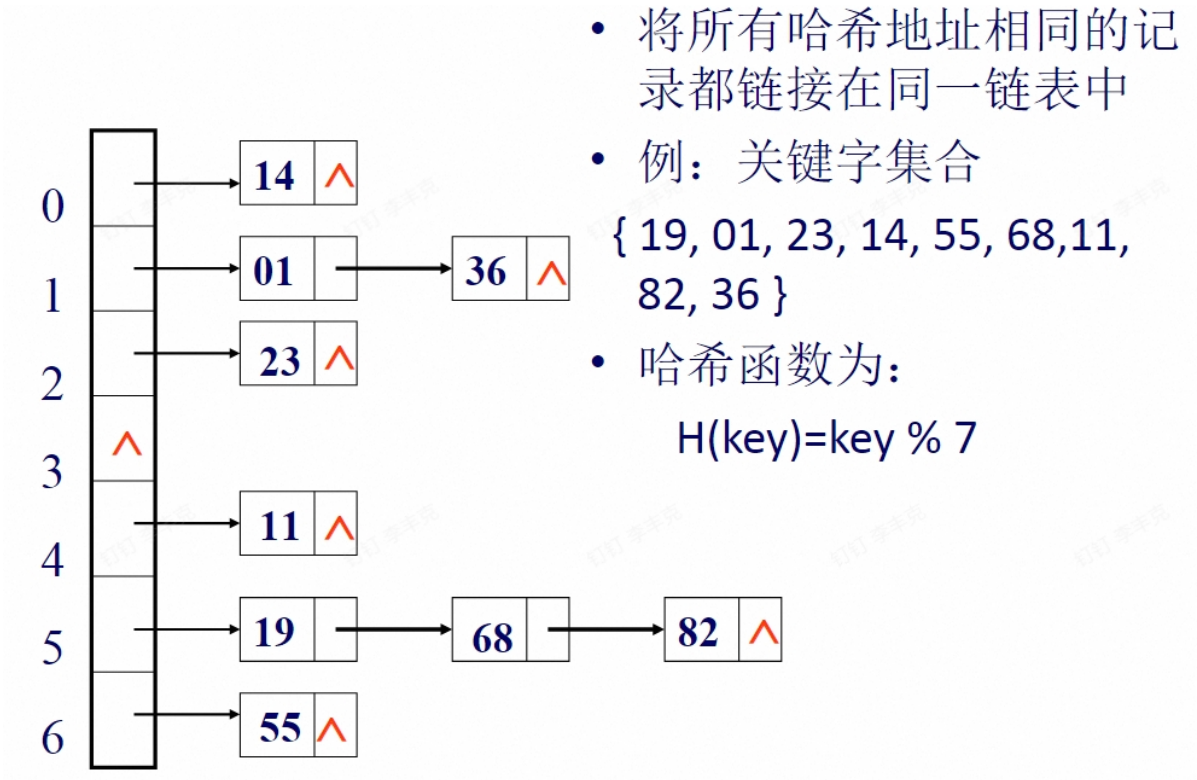
随机探测

(3) H(38)=38 % 11=5　冲突
设伪随机数序列为 9，则：
H1=(5+9) % 11=3　不冲突

84

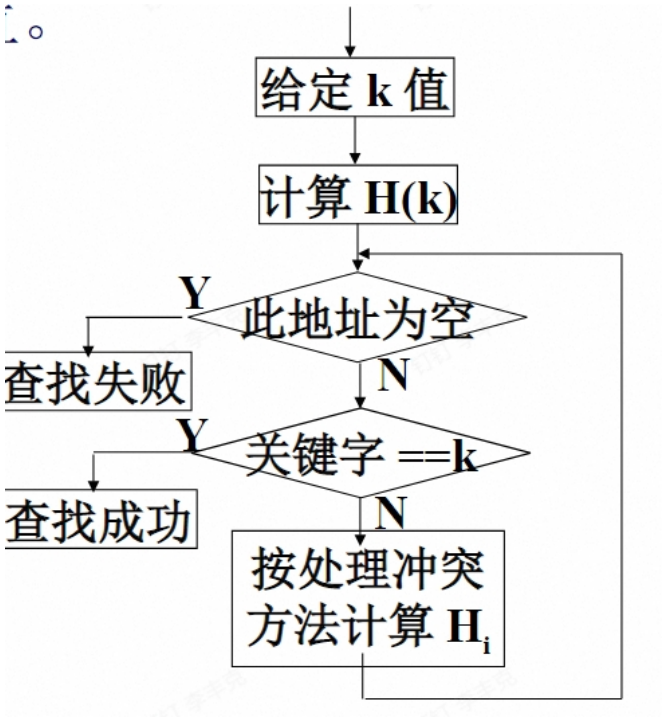
6.2.2.2 链地址法

将所有哈希地址相同的记录都链接在同一链表中



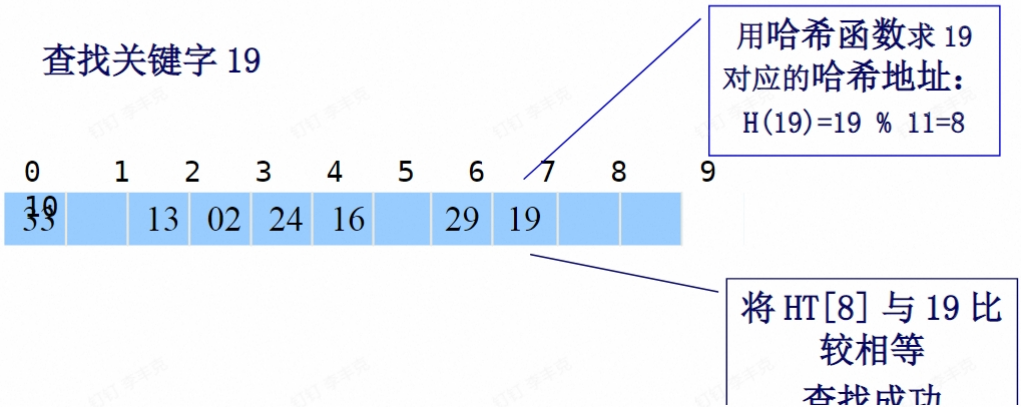
6.2.3 查找方法

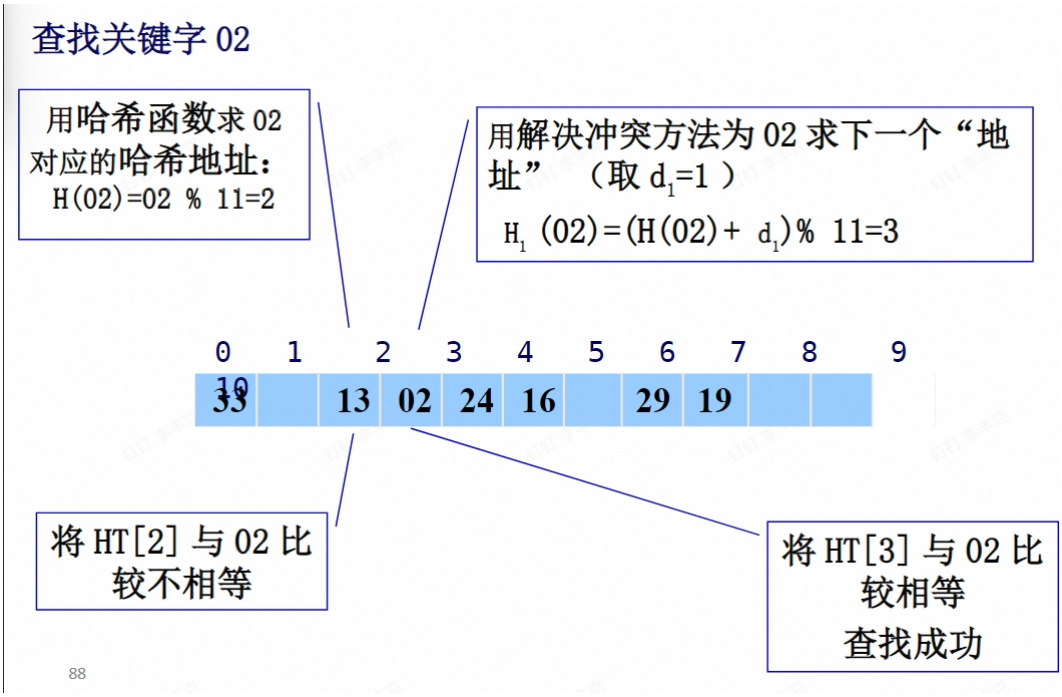
查找过程与造表过程一致，对于给定值，由哈希函数和解决冲突的方法定位记录的存储位置



一个实例：（注意图中的哈希表的地址没对应好，只看文字就行）

例：给出哈希表 HT ，哈希函数 H(key)=key % 11, 解决冲突方法：开放地址法中线性探测再散列 $Hi(key)=(H(KEY)+d_i)\%\ 11$ ($d_1=1, d_2=2, d_3=3,\dots$), 试查找关键字 19 、 02





哈希表的平均查找长度实际上并不等于零。

决定哈希表查找的ASL因素：

1）选用的哈希函数

2）选用的处理冲突的方法

3）哈希表饱和的程度，装载因子$\alpha = \frac{n}{m}$，其中n是记录数，m是表长。

哈希表的ASL是处理冲突方法和装载因子的函数。

求哈希表ASL的方法

一、查找成功时，平均查找长度

> 查找成功时的平均查找长度 = 表中每个元素查找成功时的比较次数之和 / 表中元素个数;

使用以下关键字建立 哈希表 { 19, 13, 20, 21, 23, 27, 26, 30, 50 }，哈希表长度为 10，hash function 为 H(K) = K % 10，并且使用**链地址法**（就是开链法）解决冲突，则等概率下查找成功的平均查找长度为（）？

首先就是按照题意进行 hash 表的构建，构建玩应该如下图：

存放元素	**50 30 20**			**23 13**			**26**	**27**		**19**
hash表索引	0	1	2	3	4	5	6	7	8	9

可以看到，查找一个元素，最少需要一次，最多需要三次，那么查找成功时的平均查找长度为：

> (1 * 5 + 2 * 2 + 3 * 1) / 9 = 4 / 3

其中，1 * 5 表示，比较一次就可以查找到的元素有 5 个（20, 13, 26, 27, 19），同理 2 * 2 表示比较两次的元素有 2 个（30, 23），3 * 1 表示比较三次的元素有 1 个（50），最后除以 9 也就是**哈希表中的元素个数**，就是查找成功时，平均查找长度 4 / 3。

用哈希函数查找每一个元素，一次查找到就计1，冲突时用对应的冲突解决方法，看要用几次，算上一开始的查找一共经历的次数为这个元素对应的查找次数。所有元素的查找次数之和/元素个数=平均查找长度。

图中是用链地址法，更加直观一点。

- 例：关键字集合

{ 7, 15, 20, 31, 48, 53,64, 76, 82, 99 }

线性探测处理冲突时， ASL = 2.4

平方探测处理冲突时， ASL = 2.0

链地址法处理冲突时， ASL = 1.7

6.3 排序

内部排序：不用访问外存就能排序

外部排序：排序的数量很大，不可能在内存中完成。

排序的稳定性能：对于两个关键字相等的记录，他们在序列中的相对位置，在排序之前与排序之后没有改变，称之为稳定，反之不稳定。

稳定的排序方法对任何例子 都稳定，不稳定的排序方法一定存在一个反例体现其不稳定。

6.3.1冒泡排序

对序列相邻记录进行对比，通过相邻的两两交换，把最大的记录放到最后一位上。第二轮遍历时只需遍历到倒数第二位。

同理第i次遍历只需遍历到n+i-1位置上。

冒泡排序方法实例分析

初始序列： 45* 23 32 39 50 88 05 66 45

	1	2	3	4	5	6	7	8	9	交换次数	swap
第 1 趟：	23	32	39	45*	50	05	66	45	[88]	6	1
第 2 趟：	23	32	39	45*	05	50	45	[66	88]	2	1
第 3 趟：	23	32	39	05	45*	45	[50	66	88]	2	1
第 4 趟：	23	32	05	39	45*	[45	50	66	88]	1	1
第 5 趟：	23	05	32	39	[45*	45	50	66	88]	1	1
第 6 趟：	05	23	32	[39	45*	45	50	66	88]	1	1
第 7 趟：	05	23	[32	39	45*	45	50	66	88]	0	0

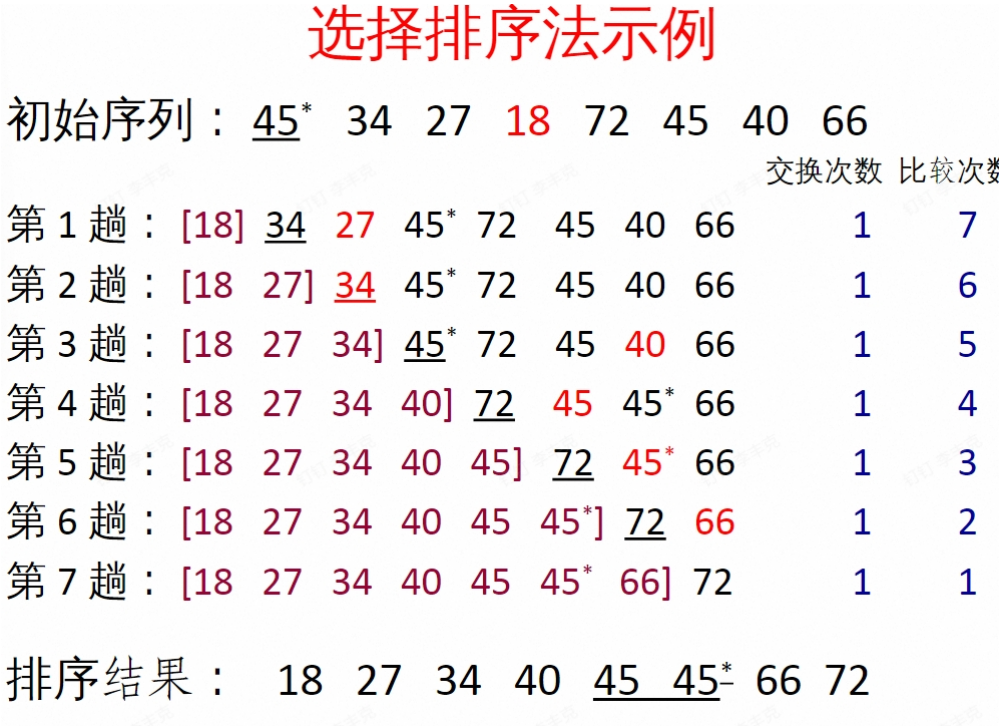
最后结果： 05 23 32 39 45* 45 50 66 88

- 冒泡排序的结束条件为：已经排序 n - 1 趟或者最近一趟没有进行“记录交换” (swap=0)。

时间复杂度$O(n^2)$，稳定但是效率低。

6.3.2 选择排序

先将tool变量赋一个初值，然后与遍历整个序列，遇到比他小的就把tool赋值为这个更小值，记录下标。遍历完后找到最小值及其下标，把这个最小值与第一位值交换位置。第二遍的时候从第二位开始，最后也与第二位交换。同理第i遍从第i位开始，最后与第i位交换。



时间复杂度$O(n^2)$，不稳定且效率低。

6.3.3 插入排序

先将第一个记录看作有序的，将下一个记录插入到前面有序序列的合适位置；第二次时前两位是有序的，把第三位记录插入到前两位的合适位置；同理到第i次时，此时前i位序列是有序的，把第i+1位插入到前i位的合适位置。以此类推直到全都有序。

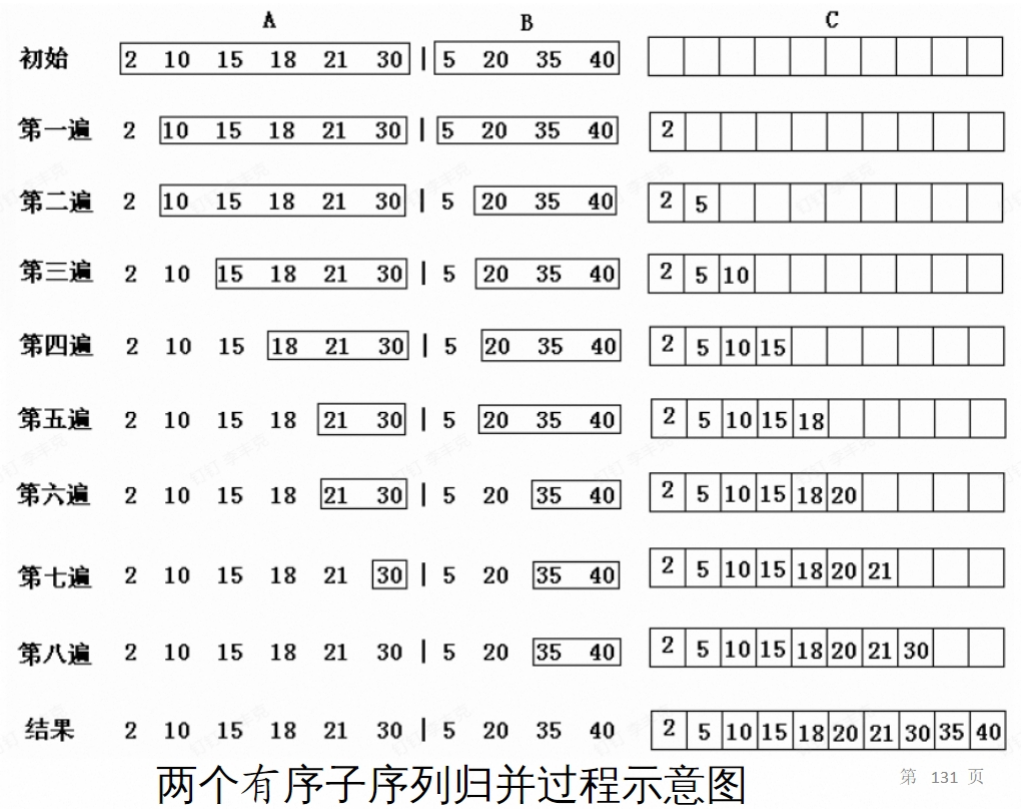
可分为直接插入类型和折半插入类型

时间复杂度$O(n^2)$，稳定但是效率低。

6.3.4 归并排序

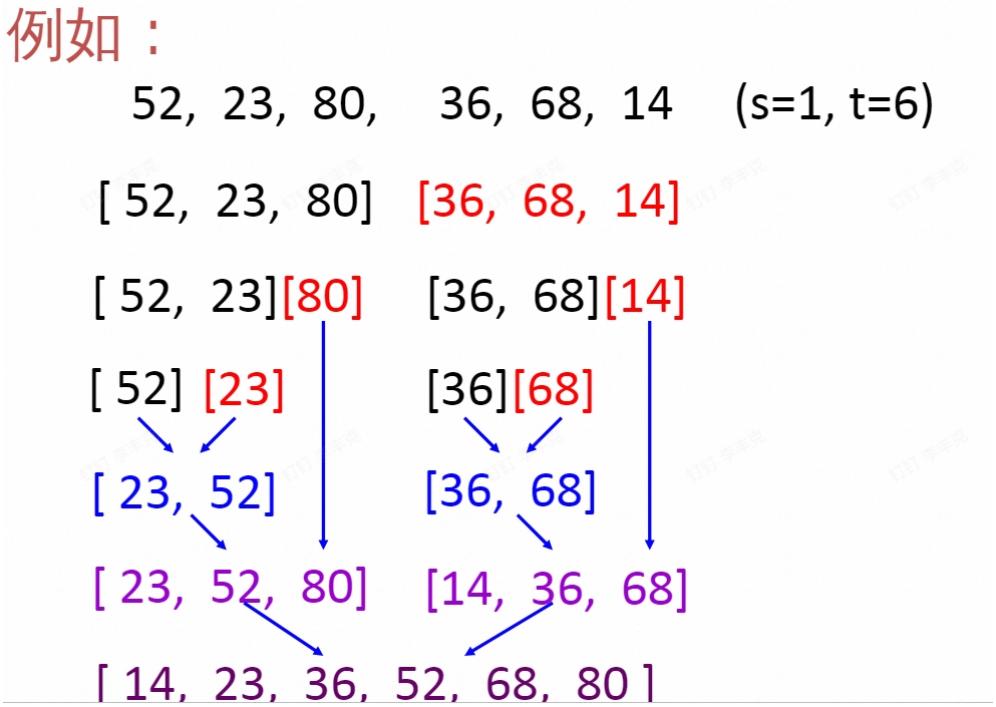
先将序列分为两个或者两个以上子序列，对这些子序列分别排序，再把这些有序子序列归并起来。可用递归的思想，把序列两两分得的子序列再两两分直到全分为只包含一个记录的序列，再两两归并回去。重点在于归并。

对归并一般采用2路归并排序



两个有序子序列归并过程示意图

归并排序就是两两分开再排好的过程



归并排序时间复杂度$O(nlog_n)$。

6.3.5 快速排序

先找到一个枢轴，小于它的关键字都移到它左边，大于它的移到它右边。这样枢轴的绝对位置其实就已经定下来了，枢轴把原序列分成了两个新序列，再分别对两个新序列进行快速排序，连续递归直到都有序。

一次快速排列的过程：

假设在第一位值有一个low指针，最后一位有个high指针。

确定一个枢轴，从high指针逆向移动遍历，比较第一个：

如果比枢轴大，就让这个值不动，high指针向前移动一位；

如果这个值比枢轴小，把枢轴与这个值交换位置，转而开始移动low指针，high指针还是指向这个位置。

从low指针开始正向移动遍历，比较其与枢轴大小：

如果它比枢轴小，这个值不动，low向后移动一位。

如果它比枢轴大，把枢轴与这个值交换位置，转而开始移动high指针，low指针还是指向这个位置。

以此类推形成递归，直到high与low都指向枢轴。枢轴右侧时high逆向移动，要求比枢轴大；枢轴左侧时low正向移动，要求比枢轴小。

同时发现异常时直接交换枢轴与异常值位置并且改变移动的指针。

例：34 66 27 18 72 40 45 把34作为枢轴

先比较high指针：发现45 40 72 都比34大，到18时比34小，交换34与18，此时high指向倒数第四位。

交换后：18 66 27 34 72 40 45

开始比较low指针：18比它小，66比他大异常，此时low指向第二位

交换后：18 34 27 66 72 40 45

再动high指针：66比他大，27比他小异常，交换

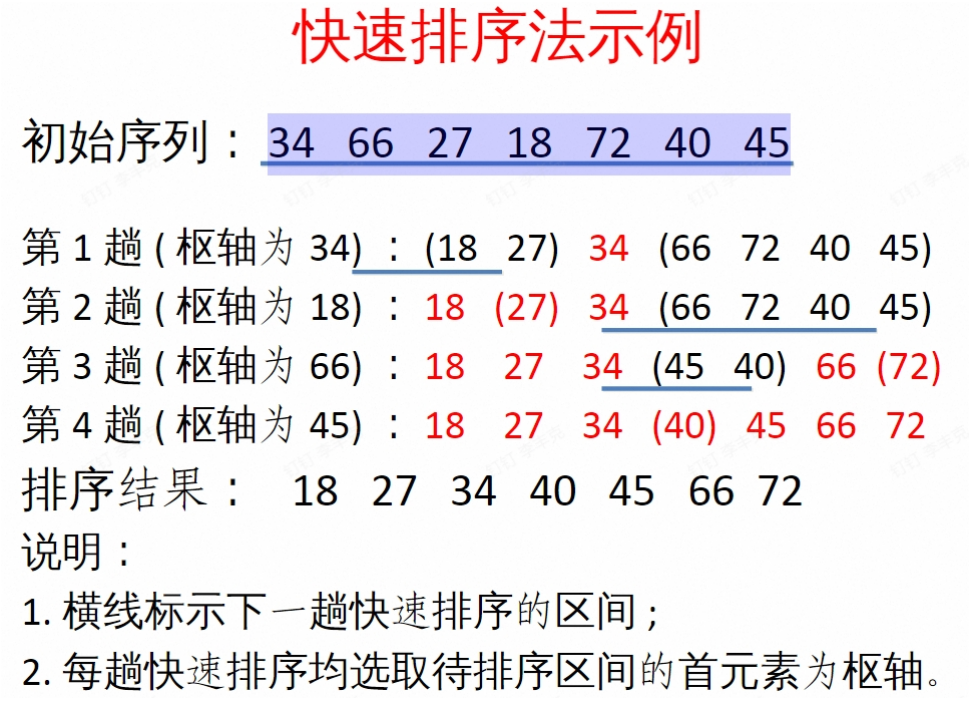
交换后：18 27 34 66 72 40 45

在动low：27比他小，再动low指向34了，high也指向34.遍历完成。

此时完成了一次快速排序。

最终结果：18 27 34 66 72 40 45

此时对34前面的（18 27）；后面的（66 72 40 45）分别进行快速排序，再得出枢纽或者到只含有一个元素的有序序列。再拼起来。



[快速排序详解]

快速排序时间复杂度$O(nlog_n)$。

6.3.6 堆排序

堆的定义：

n个元素的序列，满足

$$k_i \le k_{2i} \quad 2i \le n$$
$$k_i \le k_{2i+1} \quad 2i+1 \le n$$

或者

$$k_i \ge k_{2i} \quad 2i \le n$$
$$k_i \ge k_{2i+1} \quad 2i+1 \le n$$

可以看出来堆是一棵完全二叉树，从二叉树按照从上到下，从左到右的编号后输出的序列就是堆的序列。

堆的性质：

1）堆是一棵采用顺序存储结构的完全二叉树，k1是根节点

2）堆的根节点是最小或最大值，分别称为小根堆或者大根堆

3）从根节点到每一个叶结点的路径上经过的元素都是非递减（递增加相等）或者非递增（递减加相等）的

4）堆的任一子树也是堆

5）任一个节点的子节点均大于或者小于这个节点。

堆排序思想

1）对一组序列，先建立堆。

2）将堆顶记录和最后一个记录交换位置，前n-1个是无序的，而最后一个位置是有序的。（注意只交换数，位置不动）

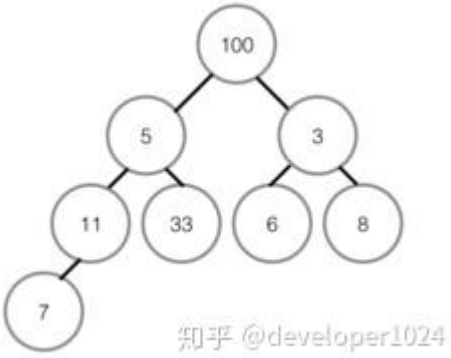
3）将前n-1个记录再调整成堆，然后堆顶元素再和倒数第二个位置交换。

4）一直重复直到记录全部排好序。

堆的建立

比如建立大根堆

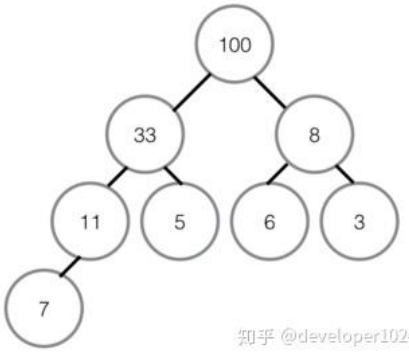
对序列 100 5 3 11 33 6 8 7



从最小子树开始构建堆，即左下角11 7，发现11>7满足，已经是堆了

再看右边子树3 6 8，3<6 3<8，大根堆，最大的肯定要做父结点，把三者最大的8作为父结点，3 8换位。

再看左边5 11 33 7这个堆对于5 11 33，33最大做父结点，5 33换位

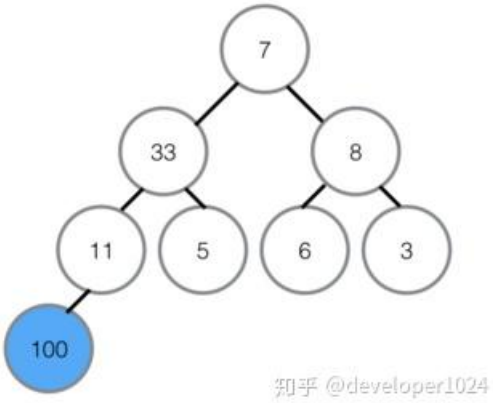


再看上面的堆100 33 8，已经满足了，那么堆就建立好了。

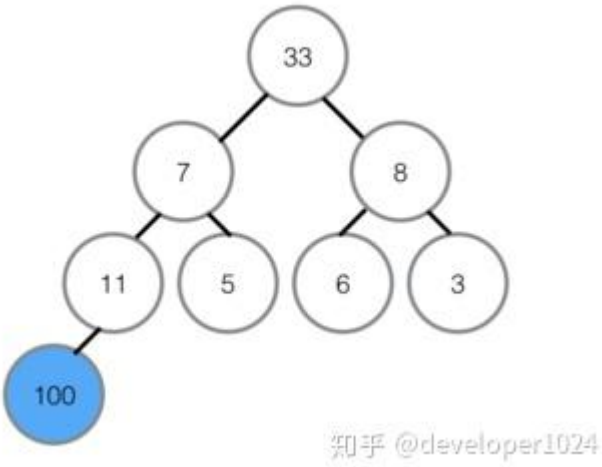
从最底层的堆一点点向上建立堆，对构成堆的三个节点肯定会存在一个最大值，无非就三种情况，按照大根堆还是小根堆把最大最小值换到子树根节点就行了。

堆的调整

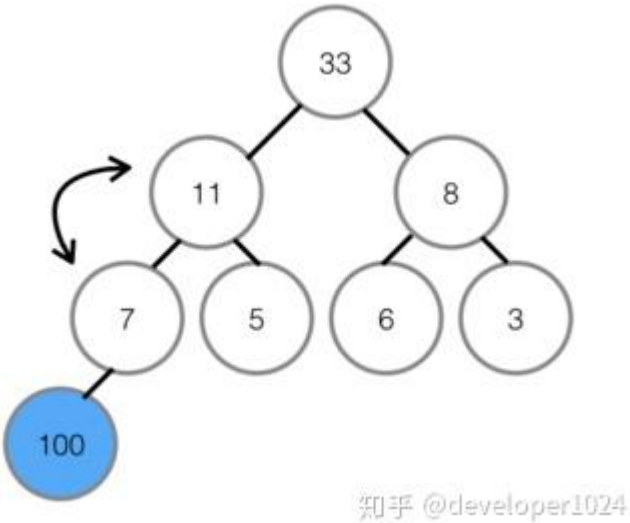
把堆顶和最后一位换位



自上而下调整，7 33 8对比，33 7换位



7 11 5对比，11 7换位



100的位置始终不变，其余节点已经构成堆了，3变成了末端位置，此时再将3和33换位；不断调整成堆，最终全部有序。

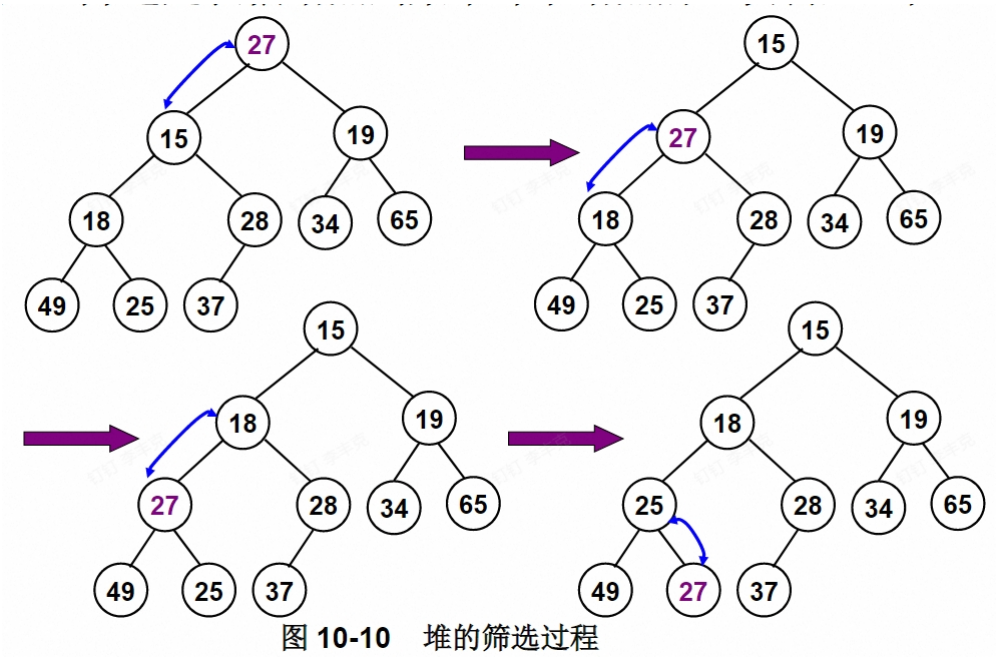


图 10-10　堆的筛选过程

堆的建立是从下到上修正；堆的调整是从上到下调整，记住末端位置一定不能变。

堆排序时间复杂度$O(nlog_n)$。

6.3.7 希尔排序（不考）

先取一个正整数d1<n作为第一个增量，将全部n个记录分成d1组，把所有相隔为d1的记录放在一组，即R[k],R[k+d1],R[k+2d1]...放在一组。

然后在各组内进行插入排序，这样分组加排序一次称为希尔排序。

取新的增量d2<d1，重复分组与排序；直到所取的增量di为1为止。

例如9,13,8,2,5,13*,7,1,15,11，增量序列是5,3,1

第一次排序分组[9,13*] [13,7] [8,1] [2,15] [5,11]

排序完后：9 7 1 2 5 13* 13 8 15 11

第二次排序分组[9,2,13,11] [7,5,8] [1,13*,15]

排序完后：2 5 1 9 7 13* 11 8 15 13

第三次d3=1，相当于直接插入排序。

排序完后：1 2 5 7 8 9 11 13* 13 15

并不是简单分割，而是有增量分割。

希尔排序的时间复杂度是其所取的增量序列的函数，并没有一个准确值。

由上面的例子可得：希尔排序不稳定。

增量序列取法：除1外都是素数，最后一个增量值必须为1。

6.4 小结

排序方法	时间复杂度	是否稳定
冒牌排序	$O(n^2)$	稳定
选择排序	$O(n^2)$	不稳定
插入排序	$O(n^2)$	稳定
归并排序	$O(nlog_n)$	稳定
快速排序	$O(nlog_n)$	不稳定
堆排序	$O(nlog_n)$	不稳定
希尔排序	不是准确值	不稳定

第七章 考试相关

划重点：

掌握逻辑结构。

四种存储方式：顺序存储，链式存储，哈希存储，索引存储（B-树那里）。

链表和顺序表：会顺序表求交并集

栈：ppt里面的一些面试题，几乎都包括在里面了，不会有新的。

队列：了解先进先出后进后出的原理，回文判断，两个栈怎么实现一个队列。

二叉树：前序，中序，后序，层序遍历。结果及实现。

哈夫曼编码：给字符及权重如何构建一个哈夫曼树和得到哈夫曼编码。

二叉搜索树，AVL树，红黑树，B-树之间关系。（在数据检索中，为了适合动态查找环境、提高查找性能，提出了二叉搜索树；但是由于二叉搜索树会有不平衡现象，极端情况下有单边现象，会降低检索效率，因此提出平衡二叉树，保证了二叉排序树的平衡；但是平衡二叉树要求全局平衡，要求较为严苛，维护代价很高，为了改善这一点提出了红黑树；红黑树只用满足局部平衡，即根节点到任何一个叶节点经过的黑色节点数数目相同，放宽了平衡的要求，不必严苛的全局平衡，出现不平衡经过简单的换色和旋转即可调整；前面的所有树都是二叉树，只适用于量很小时的检索，当对于量很大时的文件检索，效率又会很低，由此引进高度数的搜索树m叉树，降低树的高度，而为了m叉树的平衡问题又引进了m叉B-树。）

构建二叉搜索树，B-树。

（这些搜索树的删除与插入，不太一定）

哈希存储：哈希函数怎么构造（六种方法：直接选址法，平方取中法，折叠法，数字分析法，随机数法，除留余数法）遇到冲突怎么解决（两种：开放定址法（又分三种增量取值：线性探测再散列，二次探测再散列，伪随机数再散列），链地址法）

排序：快速排序，归并排序，堆排序。

时间复杂度，给出一个序列，问你走一遍排成什么样子。

栈后面题ppt原文，虽然上面也写了

1. How would you design a stack which, in addition to push and pop, also has a function min which returns the minimum element? Push, pop and min should all operate in O(1) time.

定义栈的数据结构，要求添加一个 **min** 函数，能够得到栈的最小元素。要求函数 **min** 、 **push** 以及 **pop** 的时间复杂度都是 **O(1)** 。

思路：需要一个辅助栈。每次 push 一个新元素的时候，同时将最小元素（或最小元素的位置。考虑到栈元素的类型可能是复杂的数据结构，用最小元素的位置将能减少空间消耗） push 到辅助栈中；每次 pop 一个元素出栈的时候，同时 pop 辅助栈。

步骤	数据栈	辅助栈	最小值
1	3	3	3
2	3,4	3	3
3	3,4,2	3,2	2
4	3,4,2,1	3,2,1	1
5	3,4,2	3,2	2
6	3,4	3	3
7	3,4,0	3,0	0

2. Write a program to sort a stack in ascending order You should not make any assumptions about how the stack is implemented The following are the only functions that should be used to write this program: push | pop | peek | isEmpty

写一个程序将堆栈升序排序。该堆栈就是一个普通的堆栈，不能有其他假设。函数实现是只能调用函数 **push(),pop(),peek()** 和 **isEmpty** 这几个函数。

思路：再建一个堆栈，从原堆栈中弹出一个元素到新的堆栈中。然后再比较原堆栈栈顶元素和新堆栈栈顶大小。如果符合排序规则，再次入新堆栈。如果不符合，在弹出新堆栈中的元素逐个比较直到满足排序关系。

```
while(!s.isEmpty()) {
    int tmp = s.pop();
    while(!r.isEmpty() && r.peek() > tmp) {
        s.push(r.pop());
    }
    r.push(tmp);
}
```

判断回文：

算法思想：

字符序列先分别入队和进栈，再逐个出队和退栈进行比较

模块划分：

1> Panduan(char str[],int n)
2> EnterStr(char str[],int *n)
3> main()

数据结构：

1> 顺序堆栈抽象数据类型
2> 顺序队列抽象数据类型

两个栈实现一个队列:

面试题： 用两个栈（ Stack ）实现一个队列（ Queue ）（百度、滴滴打车面试等）

思路：入队时，将元素压入 s1 。出队时，判断 s2 是否为空，如不为空，则直接弹出顶元素；如为空，则将 s1 的元素逐个“倒入” s2 ，把最后一个元素弹出并出队。

交并集运算:

顺序表的应用：集合的“并”运算

已知两集合 A 、 B ，求两集合的并集，结果存放于 A

```
void Union ( SeqList<int> & A,
                          SeqList<int> & B ) {
    int n = A.Length ( );
    int m = B.Length ( );
    for ( int i = 0; i < m; i++ ) {
        int x = B.Get(i);          // 在 B 中取一元素
        int k = A.Locate (x);      // 在 A 中搜索它
        if ( k == -1 )             // 若未找到 , 插入元
素
            { A.Insert (n, x);  n++; }
    }
}
```

顺序表的应用：集合的“交”运算

已知两集合 A 、 B ，求两集合的交集，结果存放于 A

```
void Intersection ( SeqList<int> & A,
                            SeqList<int> & B ) {
    int n = A.Length ( );
    int m = B.Length ( );  int i = 0;
    while ( i < n ) {
        int x = A.Get (i);             // 在 A 中取一元素
        int k = B.Locate (x);          // 在 B 中搜索它
        if ( k = = -1 ) { A.Delete (i);  n--; }
        else i++;                      // 未找到，在 A 中删除它
    }
}
```