数据结构与算法 A 笔记（以模块划分）

第 1 章：课程概述

1.1 数据结构

一.数据结构

1.三要素：逻辑关系，存储表示，操作（运算）

二.逻辑结构

1.定义：数学抽象，组成与逻辑关系

2.表示：用一组数据（表示为结点集合 K，由有限个结点组成）和数据间二元关系（关系集合 R），表示（K，R）

三.节点类型

1.基本数据结构：`integer,real,char,pointer`

2.复合数据类型：由基本组合

3.结构关系分类：线性，树型，图

4.线性表⊆二叉树⊆树⊆图

五.线性结构

1.前驱关系，关系 r 是有向的，且满足全序性（全部节点两两皆可比较前后），单索性（每个节点都存在唯一前驱和后继节点）

六.树形结构

1.亦称层次结构，每个节点可有多于一个直接后继，但只有唯一的直接前驱

2.最高层节点为根节点，无父节点

七.图形结构

1.亦称网络结构

2.对于图结构的关系 r 没有加任何约束

3.树型结构和图型结构的基本区别是“每个结点是否仅仅属于一个直接前驱”。而线性结构和树型结构的基本区别是“每个结点是否仅仅有一个直接后继”

八.结点与结构

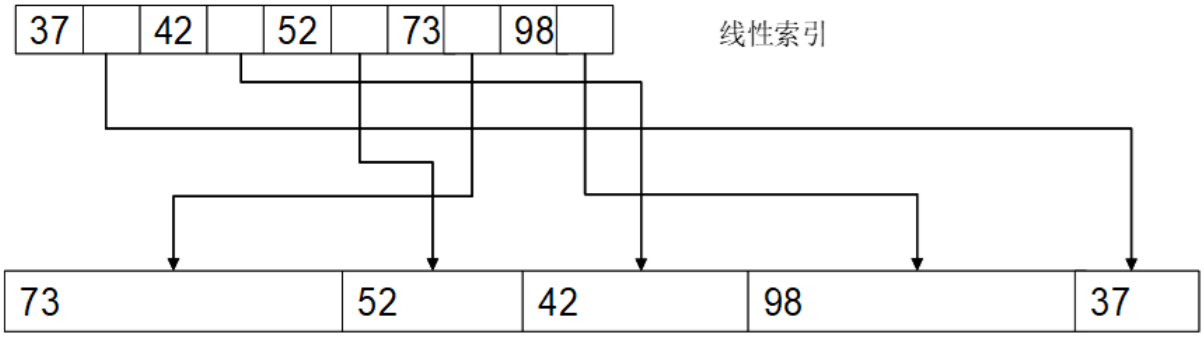
1.自顶向下
- 先明确结点和关系 r
- 分析数据类型
- 分析下一层次

九.存储结构

1. 计算机主存储器
- 空间相邻
- 随机访问

2. 存储结构建立一种逻辑结构到物理结构的映射

3. 四种基本存储映射方法
- 顺序
  - 结点按地址相邻关系顺序存储
  - 例如数组
  - 顺序存储是紧凑存储结构，存储密度 100%
- 链接
  - 利用指针指向表示两结点间逻辑关系
  - 任意的逻辑关系 r，都可以使用这种指针地址来表达。一般的做法是将数据结点分为两部分:数据字段，指针字段
  - 增删易，定位难
- 索引
  -



- 散列
  - 散列函数：将关键码 s 映射到非负整数 z

4. 抽象数据结构（ADT）
- 三要素：数据对象，关系，操作

1.2 算法

一·算法性质：通用性，有效性，确定性，有穷性

二·算法分类

1. 穷举法

2. 回溯，搜索

3. 贪心法

4. 递归分治

5. 动态规划

【例】（0-1 背包问题）有n个物品和一个容量为W的背包，每个物品有重量w_i和价值v_i两种属性，要求选若干物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

题目描述

复制 Markdown 折叠 进入 IDE 模式

辰辰是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”

如果你是辰辰，你能完成这个任务吗?

输入格式

第一行有 2 个整数 T（$1 \leq T \leq 1000$）和 M（$1 \leq M \leq 100$），用一个空格隔开，T 代表总共能够用来采药的时间，M 代表山洞里的草药的数目。

接下来的 M 行每行包括两个在 1 到 100 之间（包括 1 和 100）的整数，分别表示采摘某株草药的时间和这株草药的价值。

输出格式

输出在规定的时间内可以采到的草药的最大总价值。

输入输出样例

输入 #1 复制

```
70 3
71 100
69 1
1 2
```

输出 #1 复制

```
3
```

说明/提示

【数据范围】

- 对于 30% 的数据，$M \leq 10$；
- 对于全部的数据，$M \leq 100$。

解：

- 回溯

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void backtrack(int idx, int restTime, int curValue, int& maxValue,
               const vector<int>& time, const vector<int>& value) {
    if (idx == time.size()) {
        maxValue = max(maxValue, curValue);
        return;
```

```
    }
    // 不选当前草药
    backtrack(idx + 1, restTime, curValue, maxValue, time, value);
    // 选当前草药（需满足时间要求）
    if (restTime >= time[idx]) {
        backtrack(idx + 1, restTime - time[idx], curValue + value[idx], maxValue,
time, value);
    }
}

int main() {
    int T, M;
    cin >> T >> M;
    vector<int> time(M), value(M);
    for (int i = 0; i < M; ++i) {
        cin >> time[i] >> value[i];
    }

    int maxValue = 0;
    backtrack(0, T, 0, maxValue, time, value);
    cout << maxValue << endl;
    return 0;
}
```

- DP

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    int T, M;
    cin >> T >> M;
    vector<int> time(M), value(M);
    for (int i = 0; i < M; ++i) {
        cin >> time[i] >> value[i];
    }

    vector<int> dp(T + 1, 0);
    for (int i = 0; i < M; ++i) {
        for (int j = T; j >= time[i]; --j) {
            dp[j] = max(dp[j], dp[j - time[i]] + value[i]);
        }
    }
    cout << dp[T] << endl;
    return 0;
}
```

【例】（可分割背包问题）

【例】（归并排序）

三 • 算法分析

1. 时间和空间复杂度

【例】

```
for(i=sum=0;i<n;i++)
    sum+=a[i];
```

解：
- 数据规模：n
- 基本操作：赋值运算，共有$f(n)=2+2n$次赋值操作

三·算法渐进分析

1. 大 0 表示法
- 定义：若存在$c>0, n_0 \in N^*, s.t. \forall n \geq n_0, f(n) \leq cg(n)$
- 计算规则
  - $f_1(n)+f_2(n)=O(\max(f_1(n), f_2(n)))$
  - $f_1(n) \cdot f_2(n)=O(f_1(n), f_2(n))$

2. Ω表示法
- 定义：若存在正数$c, n_0, s.t. \forall n \geq n_0, f(n) \geq cg(n)$

3. Θ表示法
- 定义：若存在正数c_1, c_2，以及正整数$n_0, s.t. \forall n > n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$

【例】

```
for(i=0;i<n;i++){
    for(j=1,sum=a[0];j<=i;j++){
        sum+=a[j];
    }
}
```

解：$O=1(i=0)+3n+2(1+2+...+n-1)=O(n^2)$

【例】

```
for(i=4;i<n;i++)
    for(j=3;sum=a[j-4];j<=i;j++)
        sum+=a[j];
```

解：外层循环$n-4$次，对每个i，内层循环 4 次，共 11 次赋值操作，共进行$1+11 \times (n-4)=O(n)$

【例】顺序从一个规模为n的一维数组中找出一个给定的k值

解：
- 若等概率分布，平均代价$\frac{1+2+...+n}{n}=\frac{n+1}{2}$
- 若概率不等，第一个位置$\frac{1}{2}$，第一个位置$\frac{1}{4}$，其他位置都是$\frac{1}{4(n-2)}$，平均代价$\frac{1}{2}+\frac{2}{4}+\frac{3+4+...+n}{4(n-2)}=\frac{n+11}{8}$

【例】二分检索最大检索长度$[\log_2 n]+1$

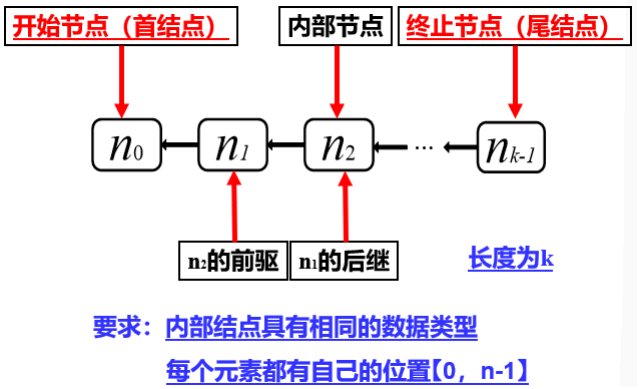
4. 时空折衷

第 2 章：线性表

2.1 线性表

一. 基本概念

1.



2. 线性表 ADT

```
template<class T>
class List{
    void clear();
    bool isEmpty();
    void append(const T value);
    void insert(int p,T value);
    void delete(int p);
    bool getPos(int& p,T value);//查找value，并返回其位置；
    bool getvalue(const int p,T& value);//把p位置的值返到value
    bool setvalue(const int p,T& value);//用value修改p处值；
};
```

3. 存储结构

- 定长，静态的存储结构
  - 向量型一维数组
  - 地址相邻表达线性关系，存储在连续的地址空间，随机访问但长度固定
- 变长，动态的存储结构
  - 链式存储
    - 指针指向表达线性关系，节点不必物理相邻
  - 动态数组
    - 提供空间表管理，为长度变化提供方法，长度增大，可申请大空间

2.2 顺序表

- 逻辑和存储结构

数据元素	k_0	K_1	…	k_i	…	k_{n-1}	…	…
逻辑地址	0	1	…	i	…	n-1	…	maxSize-1

(a) 线性表的逻辑结构

数据元素	k_0	k_1	…	K_i	…	k_{n-1}	…	
存储地址	b	b+L	…	b+i*L	…	b+(n-1)*L	…	b+(maxSize-1)*L

(b) 线性表的顺序存储结构

$$Loc(k_i) = b + L \times i,$$

其中：基地址：$b = Loc(k_0)$; 偏移量：$L = sizeof(ELEM)$

- 向量类定义

```
enum Boolean {False,True};
const int Max_length = 100;
template <class T>     //假定顺序表的元素类型T
class list  {                                    //顺序表，向量
private :
    T* nodelist;                 //私有变量，存储顺序表实例的向量
    int maxSize;               //私有变量，顺序表实例的最大长度
    int curLen;              //私有变量，顺序表实例的当前长度
    int position;            //私有变量，当前处理位置
public:
    list(const  int size);           //构造算子，实参是表实例的最大长度
    ~list();                                  //析构算子，用于将该表实例删去
    arrList(const int size) {// 创建一个新顺序表，参数为表实例的最大长度
        maxSize = size;
        aList = new T[maxSize];
        curLen = position = 0;
    }
    ~arrList() { // 析构函数，用于消除该表实例
        delete [] aList;
    }
    void clear() { // 将顺序表存储的内容清除，成为空表
        delete [] aList;
        curLen = position = 0;
        aList = new T[maxSize];
    }
    int length();                                        //返回此顺序表的当前实际长度
    bool append(const  T value);       //表尾增一新元素，表长加1
    bool insert(const  int p, const T value); //在p位置插入值value，表长加1
    bool delete(const int p);                //删去位置p的元素，表长减1;
    bool setValue(int p, const T value);          //用value修改位置p的元素值
    bool getvalue(const int p, T & value);  //把p位置值返回到变量value中
```

```
        // 查找值为value的元素，并返回第1次出现的位置
    bool getPos(int &p, const T value);
}
```

- 读取一个元素：$O(1)$
- 查找一个元素：$O(n)$
- 插入一个元素：$O(n)$
- 删除一个元素：$O(n)$

2.3 链表

- 单链表：
  - 结点指针

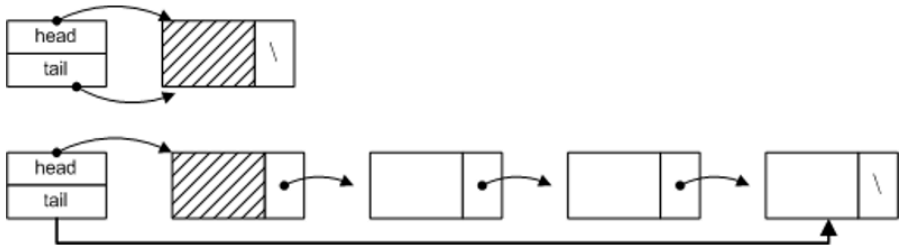
```
struct ListNode{
    ELEM data;
    ListNode* next;
};
typedef ListNode* ListPtr;
ListPtr head,tail;
```

  - 类型定义

```
template <class T>
class lnkList : public List<T> {
private:
   Link<T> *head, tail;      // 单链表的头、尾指针
   Link<T> *setPos(int p);                      // 返回线性表指向第p个元素的指针值
public:
   lnkList(ints);      // 构造函数
   ~lnkList();       // 析构函数
   bool isEmpty();       // 判断链表是否为空
   void clear();       // 将链表存储的内容清除，成为空表
   int length();                          // 返回此顺序表的当前实际长度
   bool append(T value);     // 在表尾添加一个元素value，表的长度增1
   bool insert(int p, T value);    // 在位置p插入一个元素value，表的长度增1
   bool delete(int p);       // 删除位置p上的元素，表的长度减 1
   bool getValue(int p, T value);    // 返回位置p的元素值
   bool getPos(int p, const T value);
   }  // 查找值为value的元素，并返回第1次出现的位置
```

  - 单链表头节点：

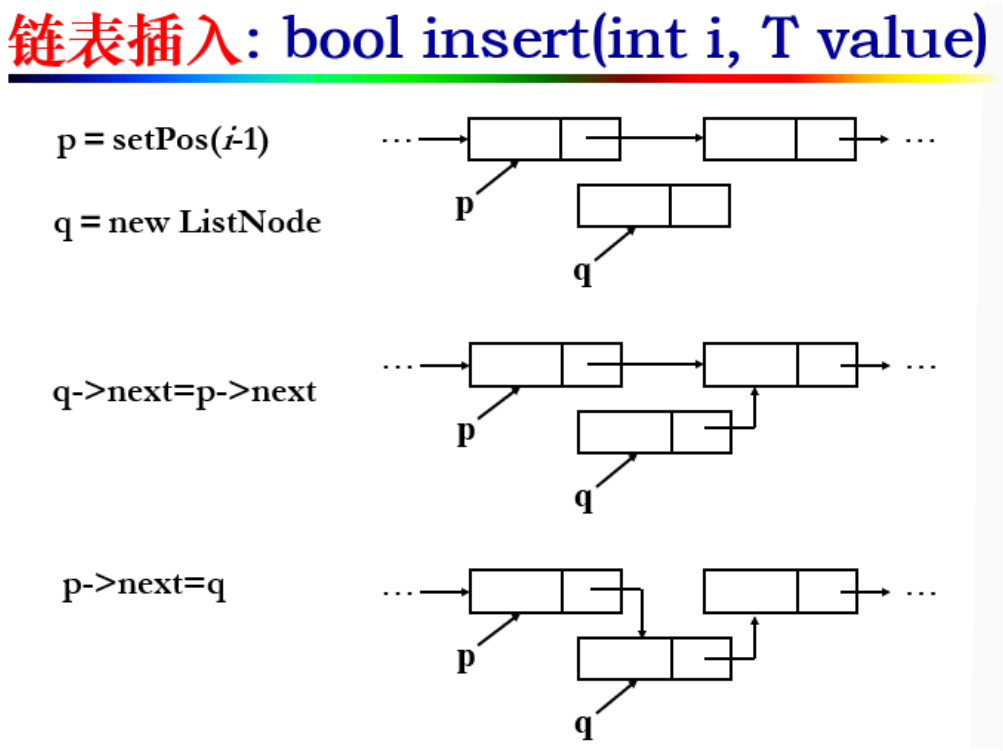
- **Header Node（或称“哨兵”）**
  - **不被作为表中的实际元素，值忽略**
  - **head指向该节点**
- **访问**
  - **必须从head开始查找链表中的元素**



- 链表检索：

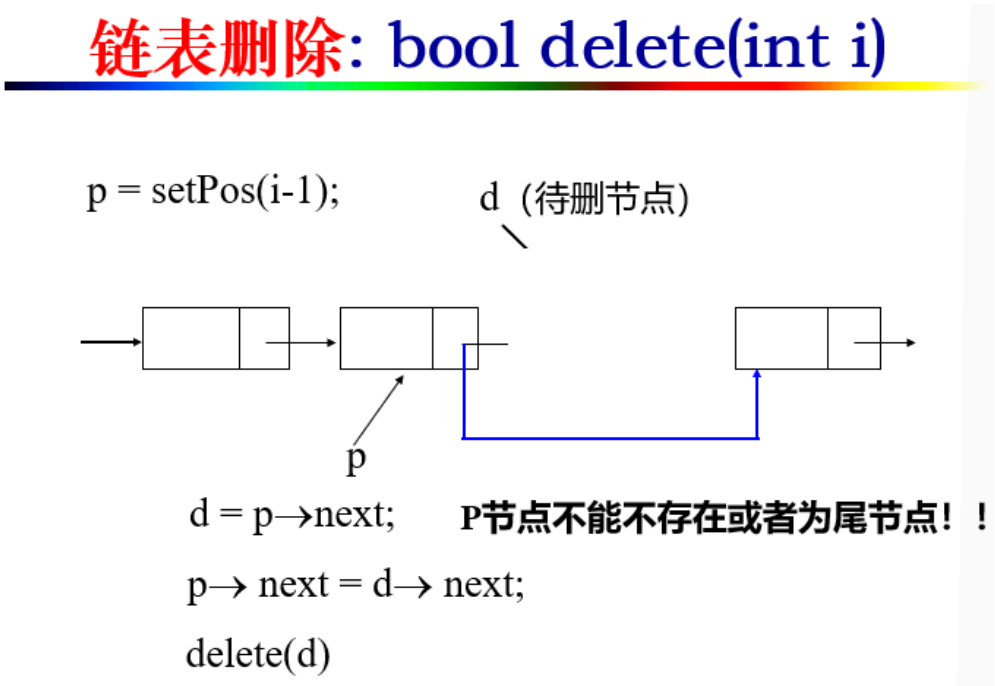
```
template<class T>
Link<T>* lnkList<T>::setPos(int i){
  int count=0;
  if(i<=-1) return head;
  Link<T>* p=head->next;
  while(p!=NULL&&count<i){
      p=p->next;
      count++;
  }
  return p;
}
```

- 链表插入



```
ListNode* Insert(int i,T value){
  ListNode* p,*q;
  q=new ListNode;
  p=setPos(i-1);
  if(p=NULL) return false;
  q->data=value;
  q->next=p->next;
  p->next=q;
  if(q->next=NULL) tail=q;
  return true;
}
```

- 链表删除

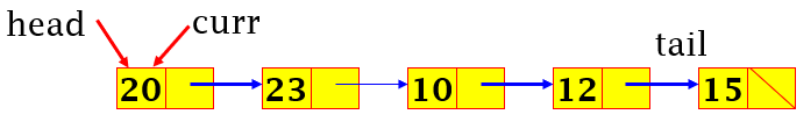



```
template<class T>
bool lnkList<T>::delete(const int i){
  Link<T> *p,*d;
  if((p=setPos(i-1))==NULL||p=tail) return false;
}
```

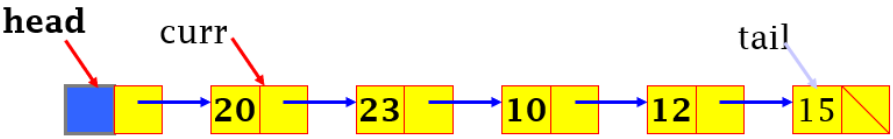
- 设置头结点的好处：
  - 由于开始结点的位置被存放在头结点的指针域中，所以在链表的第一个位置上的操作就和在表的其它位置上操作一致，无须进行特殊处理；
  - 由于开始结点的位置被存放在头结点的指针域中，所以在链表的第一个位置上的操作就和在表的其它位置上操作一致，无须进行特殊处理;无论链表是否为空，其头指针是指向头结点的非空指针（空表中头结点的指针域空），因此空表和非空表的处理也就统一了

```
if (i==0){
    s = new ListNode;                          //生成新结点
    s->data = value;
    s->next = head; // 插入到链表L中
    head = s;                    // 修改链头指针L
}
```

不带头结点

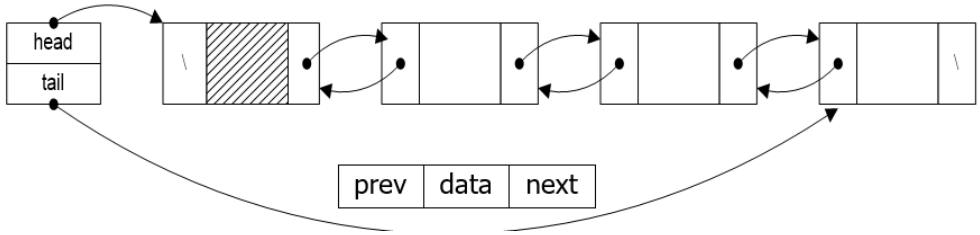


带头结点

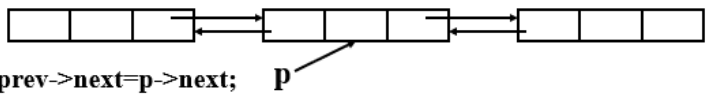


- 双链表
  - 类型说明

```
struct DoubleListNode{
  T data;
  DoubleListNode* prev;
  DoubleListNode* next;
};
struct DoubleList{
  DoubleListNode *head,*tail;
};
```

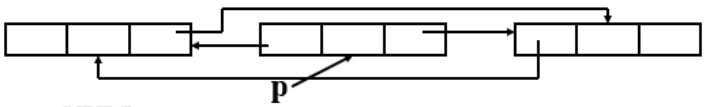


  - **删除p所指的结点setPos(i)**



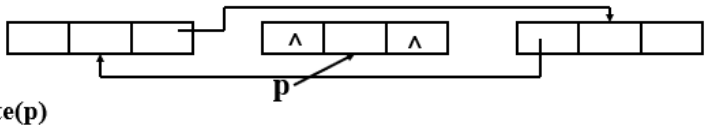
p->prev->next=p->next;

p->next->prev=p->prev;

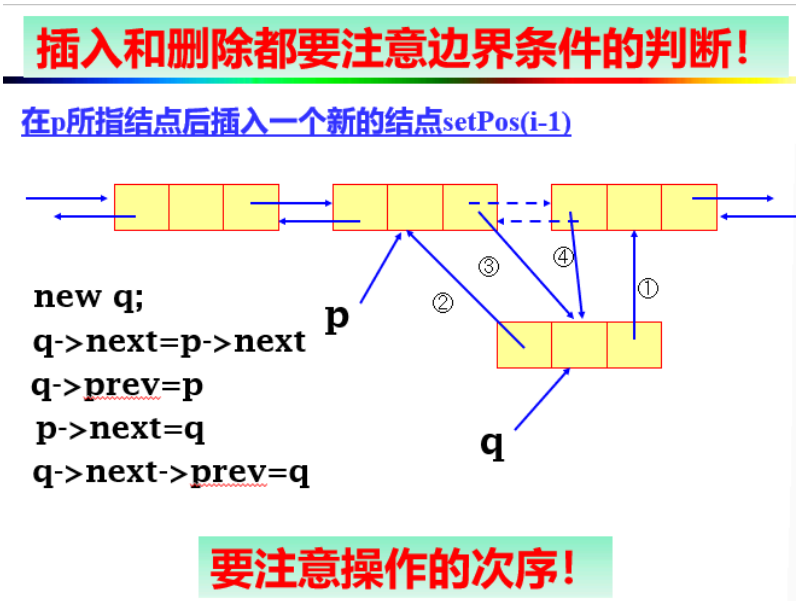


p->prev=NULL;

p->next=NULL;



delete(p)



- 循环链表：将单链表或者双链表的头尾结点链接起来，就是一个循环链表；不增加额外存储花销，却给不少操作带来了方便。从循环表中任一结点出发，都能访问到表中其他结点

2.4 线性表实现方法的比较

- 顺序表：
  - 无额外花销
  - 查找$O(1)$，插入删除$O(n)$
- 链表：
  - 能够适应经常插入删除内部元素的情况
  - 插入、删除运算时间代价$O(1)$，但找第 i 个元素时间代价$O(n)$

第 3 章：栈与队列

3.1 栈（运算只在表的一端进行）

一. 栈的定义

1. 后进先出（LIFO）

2. 栈的基本操作：入栈（push）、出栈（pop）、取栈顶元素（top）

3. 应用：表达式求值，消除递归，DFS

二. 栈的抽象数据类型

```
template<class T>
class Stack {
public:
    void clear(); // 清空栈
    bool push(const T item); // 入栈，成功返回真
```

```
    bool pop(T& item); // 出栈，成功返回真，并将栈顶元素存入item
    bool top(T& item); // 取栈顶元素，成功返回真，并将栈顶元素存入item
    bool isEmpty(); // 判断栈是否为空
    bool isFull(); // 判断栈是否已满
};
```

三. 火车进出栈问题

【例】给出入栈顺序，判定出栈顺序是否合法

解：若a_k是最后一个出栈元素，那么在它之前入栈元素一定比在他之后入栈元素先出栈

【例】给定一个长度为n的入栈序列，计算有多少种出栈序列

解：设出栈序列个数$f(n)$，根据上例，a_k把前后分为$k-1, n-k$两部分，又有$f(0)=f(1)=1, f(2)=f(1)f(0)+f(0)f(1), ...,$

$$f(n)=\sum_{k=1}^{n} f(n-k)f(k-1)=C_{2n}^{n}-C_{2n}^{n-1}=\frac{1}{n+1}C_{2n}^{n}$$

四. 栈的实现

1. 顺序栈：使用向量实现，顺序表的简化版

a. 顺序栈的类定义

```
template<class T>
class arrStack:public Stack<T> {
private:
    int mSize;// 栈的最大容量
    int top;//栈顶位置，应< mSize
    T* st;// 栈元素存储数组
public:
    arrStack(int size){
        mSize = size;
        top = -1; // 栈顶指针初始化为-1，表示栈为空
        st = new T[mSize]; // 动态分配数组
    }
    arrStack(){
        top=-1;
    }
    ~arrStack(){
        delete[] st; // 释放动态分配的数组
    }
    void clear() {
        top = -1; // 清空栈
    }
}
```

b. 按压入先后次序，最后压入的元素编号为 4，然后依次为 3, 2, 1

c. 顺序栈的溢出：

- 当栈满时，无法再入栈，可能导致内存溢出或程序崩溃（上溢）
- 对空栈进行出栈操作，可能导致访问非法内存（下溢）

d. 压入栈顶

```
bool arrStack<T>::push(const T item){
    if(top==mSize-1){
        cout<<"栈已满，无法入栈"<<endl;
        return false;
    }
    else{
        st[++top] = item; // 将元素压入栈顶
        return true;
    }
}
```

e.弹出栈顶

```
bool arrStack<T>::pop(T& item){
    if(top==-1){
        cout<<"栈为空，无法出栈"<<endl;
        return false;
    }
    else{
        item = st[top--]; // 将栈顶元素存入item，并将栈顶指针下移
        return true;
    }
}
```

2.链式栈：用单链表存储，指针方向从栈顶向下链接，原则上无栈满问题

a.链式栈的创建

```
template<class T>class lnkStack:public Stack<T> {
private:
    Link<T>* top; // 栈顶指针
    int size; // 栈的大小
public:
    lnkStack(int defSize) {
        top = NULL; // 初始化栈顶指针为空
        size = 0; // 栈的大小初始化为0
    }
    ~lnkStack() {
        clear(); // 析构时清空栈
    }
}
```

b.链式栈的入栈

```
bool lnkStack<T>::push(const T item){
    Link<T>* tmp=new Link<T>(item,top); // 创建新节点，指向当前栈顶
    top = tmp; // 更新栈顶指针
    size++; // 栈大小增加
    return true; // 入栈成功
}
Link(const T info,Link* nextValue){
    data=info;
    next=nextValue;
}
```

c.链式栈的出栈

```cpp
bool lnkStack<T>::pop(T& item){
    Link<T>* tmp;
    if(size==0){
        cout<<"栈为空，无法出栈"<<endl;
        return false; // 栈为空，出栈失败
    }
    else{
        item = top->data; // 将栈顶元素存入item
        tmp = top; // 保存当前栈顶节点
        top = top->next; // 更新栈顶指针
        delete tmp; // 释放原栈顶节点
        size--; // 栈大小减少
        return true; // 出栈成功
    }
}
```

3. 顺序栈和链式栈的比较 -时间效率

- 顺序栈：入栈和出栈操作时间复杂度均为 O(1)
- 链式栈：入栈和出栈操作时间复杂度均为 O(1)
- 空间效率
  - 顺序栈：空间利用率较低，可能浪费内存
  - 链式栈：空间利用率较高，动态分配内存，节省空间，但每个节点需要额外存储指针，增加了内存开销
  - 实际应用中，顺序栈更广泛

4. 栈的应用

- 表达式求值：使用栈来存储操作数和操作符，按照运算优先级进行计算

a. 中缀表达式

- 中缀表达式：操作符在操作数之间，如 `4*x*(2*x+a)-c`

b. 后缀表达式

- 后缀表达式：操作符在操作数之后，如 `4 x * 2 x * a + * c -`

c. 后缀表达式求值

- 循环：依次顺序读入表达式的符号序列（假设以＝作为输入序列的结束），并根据读入的元素符号逐一分析
  - 当遇到的是一个操作数，则压入栈顶
  - 当遇到的是一个运算符，就从栈中两次取出栈顶，按照运算符对这两个操作数进行计算。然后将计算结果压入栈顶
  - 如此继续，直到遇到符号＝，这时栈顶的值就是输入表达式的值

【例】

题目描述

复制 Markdown 展开 进入 IDE 模式

所谓后缀表达式是指这样的一个表达式：式中不再引用括号，运算符号放在两个运算对象之后，所有计算按运算符号出现的顺序，严格地由左而右新进行（不用考虑运算符的优先级）。

本题中运算符仅包含 +-*/。保证对于 / 运算除数不为 0。特别地，其中 / 运算的结果需要**向 0 取整**（即与 C++ `/` 运算的规则一致）。

如：3*(5-2)+7 对应的后缀表达式为：3.5.2.-*7.+@。在该式中，`@` 为表达式的结束符号。`.` 为操作数的结束符号。

输入格式

输入一行一个字符串 s，表示后缀表达式。

输出格式

输出一个整数，表示表达式的值。

输入输出样例

输入 #1	复制	输出 #1	复制
3. 5. 2. -*7. +@		16	

输入 #2	复制	输出 #2	复制
10. 28. 30. /*7. -@		-7	

说明/提示

数据保证，$1 \leq |s| \leq 50$，答案和计算过程中的每一个值的绝对值不超过 10^9。

```cpp
#include<iostream>
#include<string>
#include<stack>
#include<cctype>
using namespace std;
int main(){
    string s;
    cin>>s;
    stack<int> st;
    string tmp="";
    for(int i=0;i<s.size();i++){
        char c = s[i];
        if(c=='@')
            break;
        if(isdigit(c))
            tmp+=c;
        else if(c=='.'){
            if(!tmp.empty()){
                st.push(stoi(tmp));
                tmp = "";
```

```
            }
        }
        else if(c=='+' || c=='-' || c=='*' || c=='/'){
            int a=st.top();
            st.pop();
            int b=st.top();
            st.pop();
            int res;
            switch(c){
                case '+':
                    res = b + a;
                    break;
                case '-':
                    res = b - a;
                    break;
                case '*':
                    res = b * a;
                    break;
                case '/':
                    // if(a == 0) {
                    //     cout << "Error: Division by zero" << endl;
                    //     return 1; // Exit on division by zero
                    // }
                    res = b / a;
                    break;
            }
            st.push(res);
        }
    }
    cout<<st.top()<<endl;
    return 0;
}
```

【关键点】1. 边进栈边计算

2. >10 数的合并操作

d. 后缀计算器的类定义

```
class Calculator {
private:
    Stack<double> s;
    bool GetTwoOperands(double& opd1, double& opd2);
    void Compute(char op);
public:
    Calculator(){};
    void Run();
    void Clear();
}


template<class ELEM>
bool Calculator<ELEM>::GetTwoOperands(ELEM& opd1, ELEM& opd2) {
    if (s.isEmpty()) {
        cout << "栈为空，无法获取操作数" << endl;
```

```
        return false;
    }
    opd1 = s.top(); // 获取栈顶元素
    s.pop(); // 弹出栈顶元素
    if (s.isEmpty()) {
        cout << "栈中只有一个操作数" << endl;
        return false;
    }
    s.pop(opd2); // 再弹出第一个操作数
    return true; // 成功获取两个操作数
}


template<class ELEM>
void Calculator<ELEM>::Compute(char op) {
    bool result;
    ELEM opd1, opd2;
    result = GetTwoOperands(opd1, opd2);
    if(result==true)
          switch(op) {
              case '+':
                  s.push(opd2 + opd1);
                  break;
              case '-':
                  s.push(opd2 - opd1);
                  break;
              case '*':
                  s.push(opd2 * opd1);
                  break;
              case '/':
                  if (opd1 == 0) {
                      cout << "Error: Division by zero" << endl;
                      return; // Exit on division by zero
                  }
                  s.push(opd2 / opd1);
                  break;
              default:
                  cout << "Unknown operator: " << op << endl;
          }
      else s.ClearStack();
}


template <class ELEM> void Calculator<ELEM>::Run(void) {
char c;
ELEM newoperand;
while (cin >> c, c!= '=') {
    switch(c) {
        case '+': case '-': case '*': case '/':
             Compute(c);
             break;
        default:
             cin.putback(c); cin >> newoperand;
```

```
            S.Push(newoperand);
            break;
    }
}
if (!S.IsEmpty())
    cout << S.Pop() << endl; // 印出求值的最后结果
}
```

e. 中缀表达式转后缀表达式

- 当输入是操作数，直接输出到后缀表达式序列
- 当输入是左括号，压入栈顶
- 当输入的是运算符时： WHile

  1. 如果栈非空 and 栈顶不是左括号 and 输入运算符的优先级 “≤” 栈顶运算符的优先级，将当前栈顶元素弹栈，放到后缀表达式序列中（此步反复循环，直到上述 if 条件不成立）；将输入的运算符压入栈中。

  2. 否则把输入的运算符压栈（>当前栈顶运算符才压栈!）

- 当输入是右括号时，先判断栈顶是否为空

  1. 如果栈顶为空，报错

  2. 如果非空，则把栈中的元素依次弹出，遇到第一个左括号为止，将弹出的元素输出到后缀表达式的序列中（弹出的 开括号不放到序列中） 若没有遇到开括号，说明括号也不匹配，做异常处理，清栈退出

- 最后，当中缀表达式的符号序列全部读入时，若栈内仍有元素，把它们全部依次弹出，都放到后缀表达式序列尾部。 - 若弹出的元素遇到开括号时，则说明括号不匹配，做错误异常处理，清栈退出

【例】

题目描述

平常我们书写的表达式称为中缀表达式，因为它将运算符放在两个操作数中间，许多情况下为了确定运算顺序，括号是不可少的，而后缀表达式就不必用括号了。

后缀标记法：书写表达式时采用运算紧跟在两个操作数之后，从而实现了无括号处理和优先级处理，使计算机的处理规则简化为：从左到右顺序完成计算，并用结果取而代之。

例如：`8-(3+2*6)/5+4` 可以写为：`8 3 2 6 * + 5 / - 4 +`

其计算步骤为：

```
8 3 2 6 * + 5 / - 4 +
8 3 12 + 5 / - 4 +
8 15 5 / - 4 +
8 3 - 4 +
5 4 +
9
```

编写一个程序，完成这个转换，要求输出的每一个数据间都留一个空格。

输入格式

就一行，是一个中缀表达式。输入的符号中只有这些基本符号 `0123456789+-*/^()`，并且不会出现形如 `2*-3` 的格式。

表达式中的基本数字也都是一位的，不会出现形如 `12` 形式的数字。

所输入的字符串不要判错。

输出格式

若干个后缀表达式，第 $i+1$ 行比第 i 行少一个运算符和一个操作数，最后一行只有一个数字，表示运算结果。

输入输出样例

输入 #1 复制

```
8-(3+2*6)/5+4
```

输出 #1 复制

```
8 3 2 6 * + 5 / - 4 +
8 3 12 + 5 / - 4 +
8 15 5 / - 4 +
8 3 - 4 +
5 4 +
9
```

输入 #2 复制

```
2^2^3
```

输出 #2 复制

```
2 2 3 ^ ^
2 8 ^
256
```

说明/提示

运算的结果可能为负数，`/` 以整除运算。并且中间每一步都不会超过 2^{31}。字符串长度不超过 100。

注意乘方运算 `^` 是从右向左结合的，即 `2 ^ 2 ^ 3` 为 `2 ^ (2 ^ 3)`，后缀表达式为 `2 2 3 ^ ^`。

其他同优先级的运算是从左向右结合的，即 `4 / 2 / 2 * 2` 为 `((4 / 2) / 2) * 2`，后缀表达式为 `4 2 / 2 / 2 *`。

保证不会出现计算乘方时幂次为负数的情况，故保证一切中间结果为整数。

```cpp
#include<iostream>
#include<stack>
#include<string>
#include<vector>
#include<cctype>
using namespace std;

int grade(char op){
    if(op=='^')
        return 4;
    else if(op=='*'||op=='/')
        return 3;
    else if(op=='+'||op=='-')
        return 2;
    else if(op=='(')
        return 1;
    else
        return 0;
}

void cal(vector<string>& v){
    for(int i=0;i<v.size();i++){
        string c=v[i];
        if(c=="+"||c=="-"||c=="*"||c=="/"||c=="^"){
            int a=stoi(v[i-2]);
            int b=stoi(v[i-1]);
```

```
            int res;
            if(c=="+"){
                res=a+b;
            }
            else if(c=="-"){
                res=a-b;
            }
            else if(c=="*"){
                res=a*b;
            }
            else if(c=="/"){
                res=a/b;
            }
            else if(c=="^"){
                res=1;
                for(int j=0;j<b;j++){
                    res*=a;
                }
            }
            v[i-2]=to_string(res);
            v.erase(v.begin() + i - 1, v.begin() + i + 1);
            return;
        }
    }
}

int main() {
    stack<char> op;
    vector<string> all;
    string str;
    cin>>str;
    for(char c:str){
        if(isdigit(c)){
            all.push_back(string(1,c));
        }
        else if(c=='('){
            op.push(c);
        }
        else if (c == '+' || c == '-' || c == '*' || c == '/' || c == '^') {
            while (!op.empty() && op.top() != '(') {
                char top_op = op.top();
                if (grade(top_op) > grade(c) || (grade(top_op) == grade(c) && c !=
'^')) {  // 乘方右结合特判
                    all.push_back(string(1, op.top()));
                    op.pop();
                }
                else break;
            }
            op.push(c);
        }
        else if(c==')'){
            if(!op.empty()){
                while(op.top()!='('){
                    all.push_back(string(1,op.top()));
```

```
                    op.pop();
                }
                op.pop(); // 弹出 '('
            }
        }
    }
    while(!op.empty()){
        all.push_back(string(1,op.top()));
        op.pop();
    }
    while(1){
        for (int i = 0; i < all.size(); i++) {
            cout << all[i];
            if (i < all.size() - 1) cout << " ";
        }
        cout << endl;
        if(all.size()==1) break;
        cal(all);
    }
    return 0;
}
```

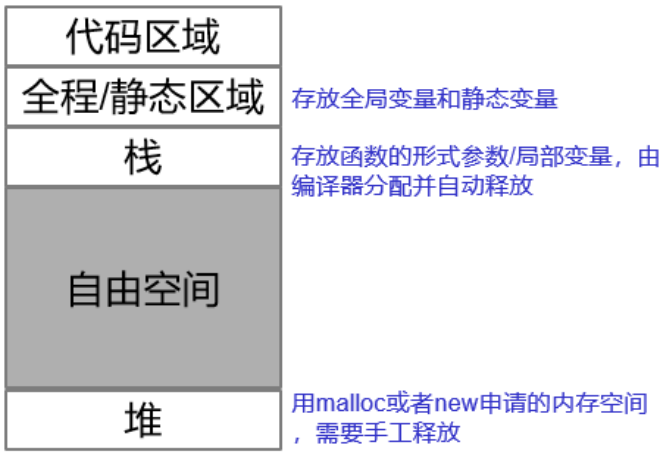
【关键点】

- 1. char 到 string 的转换 c->string(1,c)
- 2. 乘方右结合特判：if (grade(top_op) > grade(c) || (grade(top_op) == grade(c) && c != '^'))
- 3. 注意空栈问题易导致死循环

- 消除递归：将递归转化为迭代，使用栈来模拟函数调用
- 深度优先搜索（DFS）：使用栈来存储访问的节点，实现图的遍历
- 动态存储分配：

用作动态数据分配的存储区，分为堆（heap）和栈（stack）

- **堆区域则用于不符合LIFO（诸如指针的分配）的动态分配**
- **栈用于分配发生在后进先出LIFO风格中的数据（诸如函数的调用）**



3.2 队列（运算只在表的两端进行）

一. 队列的定义

1. 先进先出（FIFO）

2. 限制访问点的线性表

3. 队头：front 队尾：rear 入队：enQueue 出队：deQueue 取队首：getFront 判空：isEmpty

二. 队列的抽象数据结构

```
template<class T> class Queue{
public:
    void clear();
    bool enQueue(const T item);
    bool deQueue(T& item);
    bool getFront(T& item);
    bool isEmpty();
    bool isFull();
};
```

三. 队列的实现方式

1. 顺序队列（关键在防止假溢出）

a. 用向量存储队列元素，用两个变量分别指向 队列的前端(front)和尾端(rear)

b. 队列的溢出

- 上溢：当队列满时，再做进队操作
- 下溢：当队列空时，再做删除操作
- 假溢出：当 rear = mSize-1 时，再作插入运算就会产生溢出，如果这时队列的前端还有许多空位置，这种现象称为假溢出

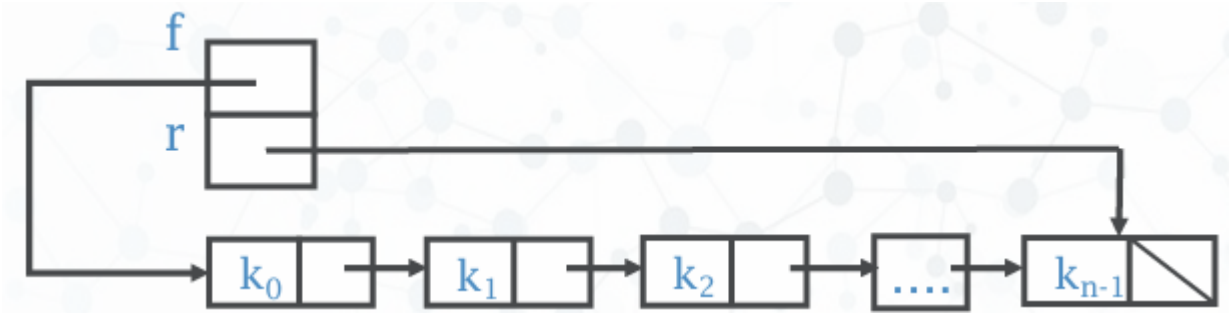
c. 循环队列的类定义：

```
class arrQueue:public Queue<T>{
private:
    int mSize;
    int front;
    int rear;
    T* qu;
public:
    arrQueue(int size){
        mSize=size+1;// 浪费一个存储空间，以区别队列空和队列满
        qu=new T[mSize];
        front=rear=0;
```

```
    }
    ~arrQueue(){
        delete [] qu;
    }
};

bool arrQueue<T>::enQueue(const T item){
    if(((rear+1)%mSize)==front){
        cout << "队列已满，溢出" << endl;
        return false;
    }
    qu[rear]=item;
    rear=(rear+1)%mSize;
    return true;
}

bool arrQueue<T>::deQueue(T& item){
    if(front==rear){
        cout << "队列为空" << endl;
        return false;
    }
    item=qu[front];
    front=(front+1)%mSize;
    return true;
}
```

2. 链式队列（用单链表方式存储，队列中每个元素对应链表中的一个结点）

a. 链式队列的表示（链接指针的方向是从队列的前端向尾端链接）



b. 链式队列的类定义

```
template<class T>
class lnkQueue:public Queue<T>{
private:
    int size;
    Link<T>* front;
    Lint<T>* rear;
public:
    lnkQueue(int size);
    ~lnkQueue();
}

bool enQueue(const T item){
    if(rear==NULL){
```

```
        front=rear=new Link<T>(item,NULL);
    }
    else{
        rear->next=new Link<T>(item,NULL);
        rear=rear->next;
    }
    size++;
    return true;
}

bool deQueue(T* item){
    Link<T>* tmp;
    if(size==0){
        cout << "队列为空" << endl;
        return false;
    }
    *item=front->data;
    tmp=front;
    front=front->next;
    delete tmp;
    if(front==NULL)
        rear=NULL;
    size--;
    return true;
}
```

c. 顺序队列与链式队列的比较

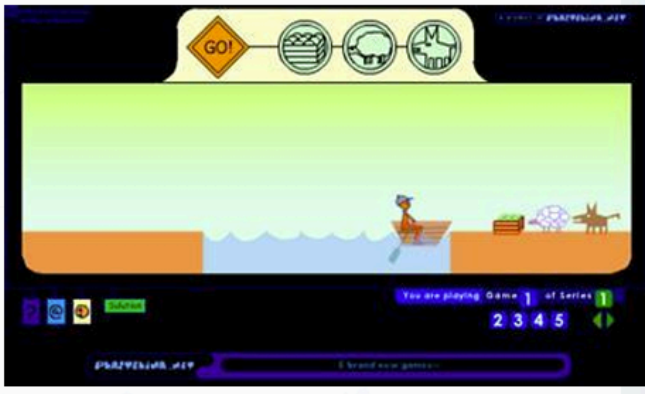
- 顺序队列 固定的存储空间
- 链式队列 可以满足大小无法估计的情况
- 都不允许访问队列内部元素

d. 队列的应用

- 调度或缓冲
- BFS

e. 农夫过河问题

- **问题抽象**：“人狼羊菜”乘船过河
  - 只有人能撑船，船只有两个位置（包括人）
  - 狼羊、羊菜不能在没有人时共处



- 假定采用 BFS 解决农夫过河问题
  - 采用队列做辅助结构，把下一步所有可能达到的状态都放在队列中，然后顺序取出对其分别处理，处理过程中再把下一步的状态放在队列中

- 数据抽象：起始岸位置：0，目标岸：1
- 数据表示：整数 status 表示上述四位二进制描述的状态
- 算法抽象：从状态 0000（整数 0）出发，寻找全部由安全状态构成的状态序列，以状态 1111（整数 15）为最终目标。状态序列中每个状态都可以从前一状态通过农夫（可以带一样东西）划船过河的动作到达。序列中不能出现重复状态
- 算法设计：

```
void solve(){
     int movers,i,location,newlocation;
     vector<int> route(END+1,-1); //记录已考虑的状态路径
     queue<int> moveTo;
     moveTo.push(0x00);// 相当于enQueue
     route[0]=0;
}

while(!moveTo.empty()&&route[15]==-1){
     status=moveTo.front();//得到现在的状态
     moveTo.pop();//相当于deQueue
     for(movers=1;movers<=8;movers<<=1){
          //农夫总是在移动，随农夫移动的也只能是在农夫同侧的东西
          if(farmer(status)==(bool)(status&movers)){
              newstatus =status ^(0x08|movers);
              if(safe(newstatus)&&(route[newstatus]==-1)){
                     route[newstatus]=status;
                     moveTo.push(newstatus);}
              }
          }
     }
}

// 反向打印出路径
if (route[15] != -1) {
    cout<<"The reverse path is :" << endl;
    for (intstatus = 15; status >= 0; status = route[status]) {
        cout<< "The status is : “ << status << endl;
        if(status == 0) break;
    }
}
else
    cout<< "No solution.“ << endl;
```

e. 栈和队列的相互模拟

3.3 栈的应用：递归到非递归

一. 简单的递归转换

1. 【例】阶乘

- 递归：

```
int f(int n){
    if(n<=0)
```

```
        return 1;
    return n*f(n-1);
}
```

- 非递归：

```
int f(int n){
    int m=1;
    for(int i=1;i<=n;i++)
        m*=i;
    return m;
}
```

- 尾递归：

```
int f(int n,int x){
    if(n<=0)
        return x;
    return f(n-1,x*n);
}
```

2. 一类特殊的递归函数—尾递归
- 指函数的最后一个动作是调用函数本身的递归函数，是递归的一种特殊情形
- 尾递归的本质是：将单次计算的结果缓存起来，传递给下次调用，相当于自动累积
- 计算仅占用常量栈空间
- 命令式语言：编译器可以对尾递归进行优化，没有必要存储函数调用栈信息，不会出现栈溢出 （例如 gcc -O2）
- 函数式语言：靠尾递归来实现循环

3. 函数运行时的动态内存分配
- stack：函数调用
- heap(堆)：指针所指向空间的分配、全局变量

二. 递归函数调用原理

1. 函数调用及返回的步骤
- 调用
  - 保存调用信息（参数，返回地址）
  - 分配数据区（局部变量）
  - 控制转移给被调函数的入口
- 返回
  - 保存返回信息
  - 释放数据区
  - 控制转移到上级函数（主调用函数）

2. 递归的实现
- 一个问题能否用递归实现，看其是否具有下面特点
  - 有递推公式（1 个或多个）
  - 有递归结束条件（1 个）
- 编写递归函数时，程序中必须有相应的语句
  - 一个（或者多个）递归调用语句
  - 测试结束语句
  - 先测试，后递归调用

- 递归程序的特点
  - 易读、易编，但占用额外内存空间

3. 函数运行时的存储分配
- 静态分配
  - 在非递归情况下，数据区的分配可以在程序运行前进行，一直到整个程序运行结束才释放，这种分配称为静态分配
  - 采用静态分配时，函数的调用和返回处理比较简单，不需要每次分配和释放被调函数的数据区
- 动态分配
  - 在递归（函数）调用的情况下，被调函数的局部变量不能静态地分配某些固定单元，而必须每调用一次就分配一份，以存放当前所使用的数据，当返回时随即释放。【大小不确定，值不确定】
  - 动态分配在内存中开辟一个称为运行栈的足够大的动态区

4. 动态存储分配

5. 运行栈中的活动记录
- 函数活动记录是动态存储分配中的基本单元
  - 当调用函数时，函数的活动记录包含为其局部数据分配的存储空间
- 运行栈随着程序执行时发生的调用链或生长或缩小
  - 每次调用执行进栈操作，把被调函数的活动信息压入栈顶
  - 函数返回执行出栈操作，恢复到上次调用所分配的数据区
- 一个函数在运行栈上可以有若干不同的活动记录，每个都代表了一个不同的调用
  - 递归深度决定运行栈中活动记录的数目
  - 同一局部变量在不同的递归层次被分配给不同的存储空间

三. 机械的递归转换

1. 递归转非递归的通用方法
- 1. 设置一工作栈保存当前工作记录

```
enumrdType{0, 1, 2}; //对应三种返回情况
public class knapNode{
    int s, n;          // 背包容量和物品数目
    rdType rd;     // 返回情况标号
```

```
    bool k;            // 结果单元
};
```

- 2. 设置 t+2 个语句标号
  - label 0: 第一个可执行语句
  - label t+1: 设置在函数体结束处
  - label i (1 <= i <= t): 第 i 个递归返回处
- 3. 增加非递归入口

```
Stack<knapNode> stack;
knapNodetmp;
tmp.s= s; tmp.n= n, tmp.rd = 0;
stack.push(tmp);    // 入栈
```

- 4. 替换第 i (i = 1, …, t)个递归规则
  - 若函数体第 i (i=1, …, t)个递归调用语句形如 recf(a1, a2, …, am)；则用以下语句替换：
  - 并增加标号为 i 的退栈语句

```
S.push(i, a1, ..., am);
goto label0;

label i: S.pop(&x);
//根据取值x进行相应的返回处理
```

- 5. 所有递归出口处增加语句：

```
goto label t+1;
```

- 6. 标号为 t+1 的语句的格式

```
S.pop(&tmp);
switch (tmp.rd) {
case 0: return;
case 1: gotolabel1; break;
// ……
cast t: gotolabelt; break;
default: break;
}
```

- 7. 改写循环和嵌套中的递归
- 8. 优化处理

2.[简化的 0-1 背包问题] 我们有 n 件物品，物品 i 的重量为 w[i] 。如果限定每种物品（0）要么完全放进背包（1）要么不放进背包；即物品是不可分割的。问： 能否从这 n 件物品中选择若干件放入背包，使其重量之和恰好为 s

```
bool knap(int s,int n){
    if(s==0)
        return true;
    if((s<0)||(s>0&&n<1))
        return false;
    if(knap(s-w[n-1],n-1)){
        cout<<w[n-1];
```

```
        return true;
    }
    else
        return knap(s,n-1);
}
```

第 4 章：字符串

4.1 字符串的基本概念

一.最基本定义

1.特殊的线性表，即元素为字符的线性表

2.$n(\geq 0)$ 个字符的有限序列， 一般记作 $S:c_0c_1c_2...c_{n-1}$，S是串名，$c_0c_1c_2...c_{n-1}$是串值，c_i是串中字符，n是串长

二.字符/符号

1.字符(char) ：组成字符串的基本单位

2.取值依赖于字符集 Σ（同线性表，结点的有限集合）

三.字符编码：单字节（8 bits）

- 用 ASCII 码对 128 个符号进行编码
- 其他编码方式： UNICODE…

四.处理子串的函数

1.子串（被包含）

- 空串是任意串的子串
- 真子串：非空且不为自身的子串

2.函数

操作类别	方法	描述
子串	substr ()	返回一个串的子串
拷贝/交换	swap ()	交换两个串的内容
	copy ()	将一个串拷贝到另一个串中
赋值	assign ()	把一个串、一个字符、一个子串赋值给另一个串中
	=	把一个串或一个字符赋值给另一个串中
插入/追加	insert()	在给定位置插入一个字符、多个字符或串
	append () / +=	将一个或多个字符、或串追加在另一个串后
拼接	+	通过将一个串放置在另一个串后面来构建新串
查询	find ()	找到并返回一个子序列的开始位置
替换/清除	replace ()	替换一个指定字符或一个串的字串
	clear ()	清除串中的所有字符
统计	size () / length()	返回串中字符的数目
	max_size ()	返回串允许的最大长度

五. 字符串中的字符

1. 重载下标运算符[ ]

```
char& string::operator[](int n);
```

2. 按字符定位下标

```
int string::find(char c,int start=0);
```

3. 反向寻找，定位尾部出现的字符

```
int string::rfind(char c,int pos=0);
```

4.2 字符串的存储结构

一. 字符串的顺序存储

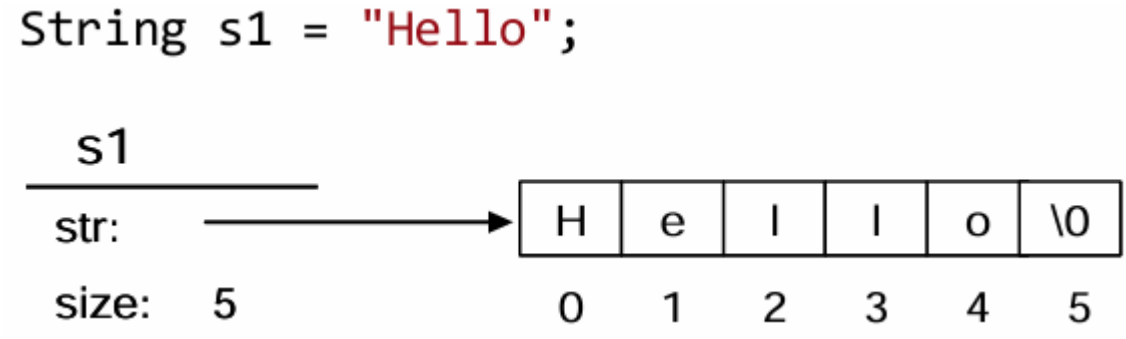
1. 处理方案

- 用 S[0] 作为记录串长的存储单元 (Pascal)
  - 缺点：限制了串的最大长度不能超过 256
- 为存储串的长度，另辟一个存储的地方
  - 缺点：串的最大长度一般是静态给定的，不是动态申请数组空间
- 用一个特殊的末尾标记 '\0' (C/C++)

2. 早期：顺序，链接，索引

二 • 字符串类的存储结构

```
private:
char* str;
int size;
```



三. 串的运算实现

```
int strcmp(char* d,char* s){
    int i;
    for(int i=0;d[i]==s[i];i++){
        if(d[i]=='\0'&&s[i]=='\0')
            return 0;
    }
    return (d[i]-s[i])/abs(d[i]-s[i]);
}
```

4.4 字符串的模式匹配

一. 模式匹配

1. 定义：在目标 T 中寻找一个给定的模式 P 的过程

2. 应用：文本编辑时的特定词，句的查找，DNA 信息的提取

3. 用给定的模式 P，在目标字符串 T 中搜索与模式 P 全同的一个子串，并求出 T 中第一个和 P 全同匹配的子串（简称为“配串”）,返回其首字符位置

二. 朴素算法

1. 穷举法

```
int Findpat(string T,string P,int startindex){
    for(int g=startindex;g<=T.length()-P.length();g++){
        for(int j=0;((j<P.length())&&(T[g+j]==P[j]));j++)
            if(j==P.length())
                return g;
    }
    return -1;
}
```

2. 效率分析:假定目标 T 的长度为 n，模式 P 长度为 m（m≤n），在最坏的情况下，每一次循环都不成功，则一共要进行比较（n-m+1）次,每一次“相同匹配”比较所耗费的时间，是 P 和 T 逐个字符比较的时间，最坏情况下，共 m 次- 因此，整个算法的最坏时间开销估计为 $O(m*n)$

三. KMP 算法

1. 简介：一种高效字符串匹配算法，通过预处理模式串生成 next 数组（部分匹配表），在匹配失败时跳过无效比较，将时间复杂度优化至 O(n+m)（n 为主串长度，m 为模式串长度）

2. next 数组

- 定义：next[i]表示模式串 P[0..i]中，最长相等真前缀和真后缀的长度（不包括子串本身）
- 作用：当匹配失败时，根据 next 值移动模式串指针，避免主串回溯
- 构建逻辑
  - P[0]：无前缀/后缀，next[0]=0
  - P[1..8]：若 P[i] == P[j]，则 j++；否则 j = next[j-1]回退

【例】

经典编程题：字符串匹配

题目描述

给定文本串text和模式串pattern，找出pattern在text中所有出现的起始位置（下标从0开始） 2 7 。

输入示例:

复制

```
5
ABABC
10
ABABABCABAB
```

输出示例:

复制

```
0 5
```

解：

【例】

题目描述

给出两个字符串 s_1 和 s_2，若 s_1 的区间 $[l,r]$ 子串与 s_2 完全相同，则称 s_2 在 s_1 中出现了，其出现位置为 l。
现在请你求出 s_2 在 s_1 中所有出现的位置。

定义一个字符串 s 的 border 为 s 的一个**非 s 本身**的子串 t，满足 t 既是 s 的前缀，又是 s 的后缀。
对于 s_2，你还需要求出对于其每个前缀 s' 的最长 border t' 的长度。

输入格式

第一行为一个字符串，即为 s_1。
第二行为一个字符串，即为 s_2。

输出格式

首先输出若干行，每行一个整数，**按从小到大的顺序**输出 s_2 在 s_1 中出现的位置。
最后一行输出 $|s_2|$ 个整数，第 i 个整数表示 s_2 的长度为 i 的前缀的最长 border 长度。

输入输出样例

输入 #1 复制

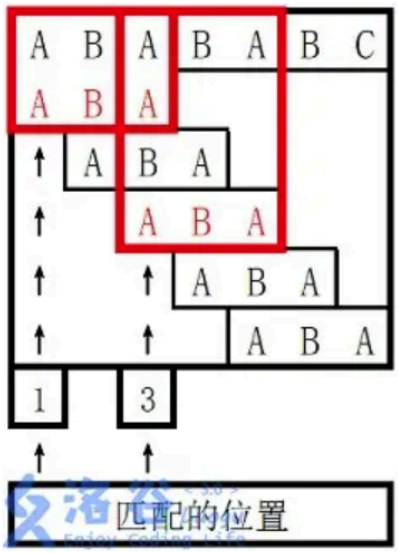
```
ABABABC
ABA
```

输出 #1 复制

```
1
3
0 0 1
```

说明/提示

样例 1 解释



对于 s_2 长度为 3 的前缀 `ABA`，字符串 `A` 既是其后缀也是其前缀，且是最长的，因此最长 border 长度为 1。

数据规模与约定

本题采用多测试点捆绑测试，共有 3 个子任务。

- Subtask 0（30 points）：$|s_1| \leq 15$，$|s_2| \leq 5$。
- Subtask 1（40 points）：$|s_1| \leq 10^4$，$|s_2| \leq 10^2$。
- Subtask 2（30 points）：无特殊约定。
- Subtask 3（0 points）：Hack。

对于全部的测试点，保证 $1 \leq |s_1|, |s_2| \leq 10^6$，$s_1, s_2$ 中均只含大写英文字母。

解：

第 5 章：二叉树

5.1 二叉树的概念

一·定义：二叉树(binary tree)由结点的有限集合构成，这个有限集合或者为空集(empty)，或者为由一个根结点(root)及两棵互不相交、分别称作这个根的左子树(left subtree)和右子树(right subtree)的二叉树组成的集合

二·五种基本形态

三·术语

1.结点
- 子结点、父结点、最左子结点
- 兄弟结点、左兄弟、右兄弟
- 分支结点、叶结点
  - 没有子树的结点称作 叶结点（或树叶、终端结点）
  - 非终端结点称为分支结点

2.边：两个结点的有序对

3.路径、路径长度
- 除结点 k0 外的任何结点 k∈K，都存在一个结点序列 k0，k1，…，ks，使得 k0 就是树根，且 ks=k，其中有序对 $< k_i - 1, k_i > \in$ r（1≤i≤s）。这样的结点序列称为从根到结点 k 的一条路径，其路径长度为 s（包含的边数）

4.祖先，后代
- 若有一条由 k 到达 ks 的路径，则称 k 是 ks 的祖先，ks 是 k 的子孙

5.层数
- 根为第 0 层，其他结点的层数等于其父结点的层数加 1

6.深度
- 层数最大的叶结点的层数

7.高度
- 层数最大的叶结点的层数加 1

8.满二叉树
- 一棵二叉树的任何结点，或者是树叶，或者恰有两棵非空子树

9.完全二叉树
- 最多只有最下面的两层结点度数可以小于 2，且最下一层的结点都集中最左边

10.扩充二叉树
- 所有空子树，都增加空树叶，
- 外部路径长度 E 和内部路径长度 I 满足：E = I + 2n（n 是内部结点个数）

四·主要性质

1.一棵二叉树，若其终端结点数为 n_0，度为 2 的结点数为n_2， 则 $n_0 = n_2 + 1$

证：设总边数 A，总结点数 B，节点分为n_0, n_1, n_2则$B = A - 1 = n_0 + n_1 + n_2 - 1$，而$B = n_1 + 2n_2$，联立即得

2.满二叉树定理：非空满二叉树树叶数目等于其分支结点数加 1

证：满二叉树要求所有分支结点（非叶结点）的度均为 2（即不存在度为 1 的结点），故 $n_2 = n_b$

3. 满二叉树定理推论：一个非空二叉树的空子树数目（空指针数）等于其结点数加 1

4.有 n 个结点（n>0）的完全二叉树的高度为 $\log_2(n+1)$

5.2 二叉树的抽象数据类型

一·抽象数据类型

1.逻辑结构 + 运算

- 针对整棵树
  - 初始化二叉树
  - 合并两棵二叉树
- 围绕结点
  - 访问某个结点的左子结点、右子结点、父结点
  - 访问结点存储的数据

```
template<class T>
class BinaryTreeNode{
friend class BinaryTree<T>;// 声明二叉树类为友元类
private:
    T info;// 二叉树结点数据域
public:
    BinaryTreeNode(); // 缺省构造函数
    BinaryTreeNode(const T& ele); // 给定数据的构造
    BinaryTreeNode(const T& ele,BinaryTreeNode<T>* l,BinaryTreeNode<T>* r);// 子树构造结点
    T value() const;// 返回当前结点数据
    BinaryTreeNode<T>* leftchild() const;// 返回左子树
    void setLeftchild(BinaryTreeNode<T>*); // 设置左子树
    void setRightchild(BinaryTreeNode<T>*); // 设置右子树
    void setValue(const T& val); // 设置数据域
    bool isLeaf() const; // 判断是否为叶结点
    BinaryTreeNode<T>& operator =(const BinaryTreeNode<T>& Node);// 重载赋值操作符
}

template <class T>
class BinaryTree {
private:
BinaryTreeNode<T>* root; // 二叉树根结点
public:
BinaryTree() {root = NULL;};
~BinaryTree(){DeleteBinaryTree(root);};
bool isEmpty() const;// 判定二叉树是否为空树
BinaryTreeNode<T>* Root() {return root;}; // 返回根结点
};

BinaryTreeNode<T>* Parent(BinaryTreeNode<T>* current);     // 返回父
BinaryTreeNode<T>* LeftSibling(BinaryTreeNode<T> *current);// 左兄
BinaryTreeNode<T>* RightSibling(BinaryTreeNode<T> *current); // 右兄
void CreateTree(const T& info,
BinaryTree<T>& leftTree, BinaryTree<T>&  rightTree);    // 构造新树
void PreOrder(BinaryTreeNode<T> *root);    // 前序遍历二叉树或其子树
void InOrder(BinaryTreeNode<T> *root);
// 中序遍历二叉树或其子树
void PostOrder(BinaryTreeNode<T> *root);   // 后序遍历二叉树或其子树
```

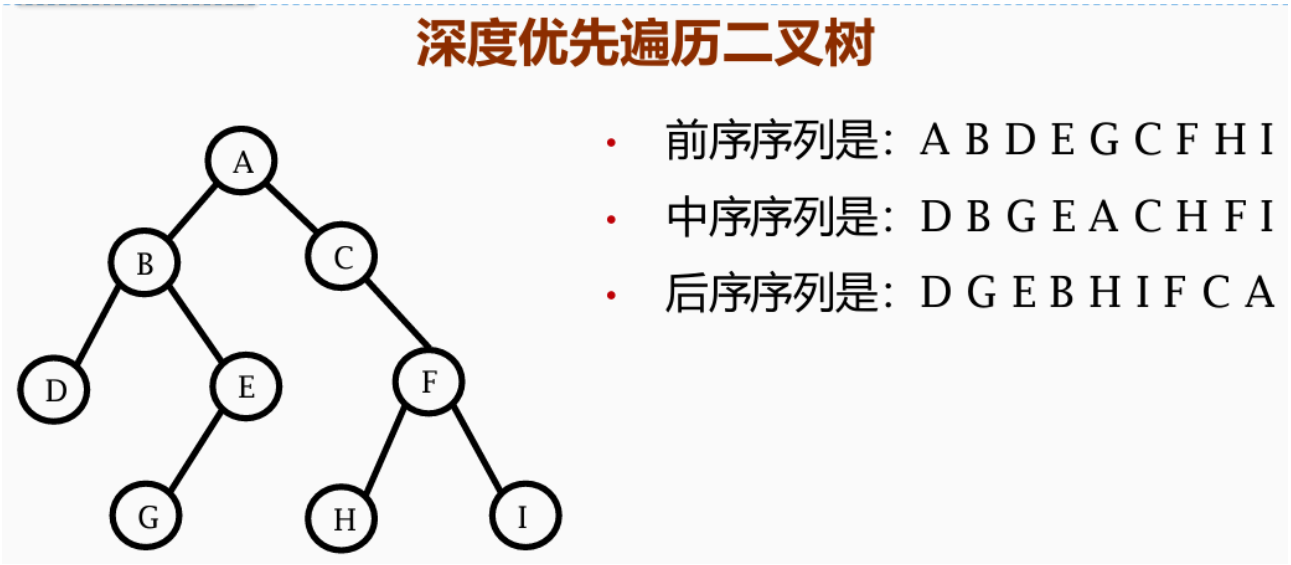
```
void LevelOrder(BinaryTreeNode<T> *root);  // 按层次遍历二叉树或其子树
void DeleteBinaryTree(BinaryTreeNode<T> *root); // 删除二叉树或其子树
```

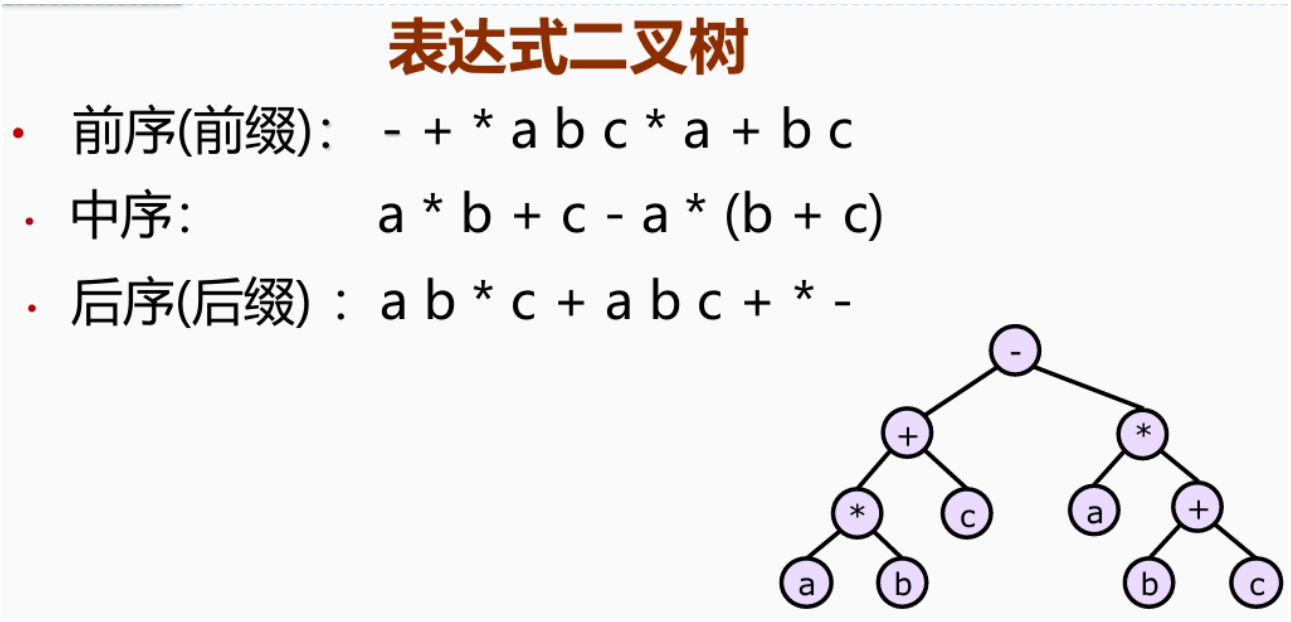
简化版

```
struct Node{
    int v;
    Node* l;
    Node* r;
    Node(int x):v{x},l{NULL},r{NULL}{}
};
```

二 • DFS 遍历二叉树

- 前序法：访问根结点；按前序遍历左子树；按前序遍历右子树
- 中序法：按中序遍历左子树；访问根结点；按中序遍历右子树
- 后序法：按后序遍历左子树；按后序遍历右子树；访问根结点





- 递归遍历

```
template<class T>
void BinaryTree<T>::DepthOrder(BinaryTree<T>* root){
    if(root!=NULL){
        Visit(root);//前序
        DepthOrder(root->leftchild()); // 递归访问左子树
        Visit(root);//中序
        DepthOrder(root->leftchild());// 递归访问右子树
        Visit(root);//后序
    }
}
```

【例】已知某二叉树的中序序列为 $\{A, B, C, D, E, F, G\}$，后序序列为 $\{B, D, C, A, F, G, E\}$；则其前序序列为何？

解：

1. **确定根节点**
   - 后序遍历的最后一个节点是整棵二叉树的根节点 [1][3][5]。
     - 后序序列：$\{B, D, C, A, F, G, \boxed{E}\}$ → 根节点为 E。
2. **划分左右子树（基于中序序列）**
   - 在中序序列 $\{A, B, C, D, E, F, G\}$ 中：
     - 根节点 E 左侧 $\{A, B, C, D\}$ 是左子树 [1][5]。
     - 根节点 E 右侧 $\{F, G\}$ 是右子树 [1][5]。
3. **提取子树的后序序列**
   - **左子树**：后序序列中根节点 E 前的部分（长度为左子树节点数 4）→ $\{B, D, C, A\}$
   - **右子树**：后序序列中左子树后、根节点前的部分 → $\{F, G\}$ [3][5]。
4. **递归处理子树**
   - **左子树（中序 $\{A, B, C, D\}$，后序 $\{B, D, C, A\}$）**：
     - 根节点：后序最后一个元素 A [3][5]。
     - 中序划分：A 左侧为空（无左子树），右侧 $\{B, C, D\}$ 为右子树 [5]。
     - 右子树的后序：$\{B, D, C\}$ → 根节点为 C。
       - C 的中序左侧 $\{B\}$ 是左子树，右侧 $\{D\}$ 是右子树 [5]。
   - **右子树（中序 $\{F, G\}$，后序 $\{F, G\}$）**：
     - 根节点：G（后序最后一个）[3][5]。
     - 中序划分：G 左侧 $\{F\}$ 是左子树，右侧为空 [1]。

- 非递归算法（用栈模拟）
  - 前序：遇到一个结点，就访问该结点，并把此结点的非空右结点推入栈中，然后下降去遍历它的左子树；遍历完左子树后，从栈顶托出一个结点，并按照它的右链接指示的地址再去遍历该结点的右子树结构

```
template<class T>
void BinaryTree<T>::PreOrderWithoutRecusion(BinaryTree<T>* root){
  using std::stack;
  stack<BinaryTree<T>*> aStack;
  BinaryTree<T>* pointer=root;
  aStack.push(NULL);
  while(pointer){
      Visit(pointer->value());
      if(pointer->rightchild()!=NULL)
          aStack.push(pointer->rightchild());
      if(pointer->leftchild()!=NULL)
          aStack.push(pointer->leftchild());
      else{
          pointer=aStack.top();
          aStack.pop();
      }
  }
}
```

- 中序：遇到一个结点，把它推入栈中，遍历其左子树；遍历完左子树，从栈顶托出该结点并访问之，按照其右链地址遍历该结点的右子树

```
template<class T>
void BinaryTree<T>::InOrderWithoutRecusion(BinaryTreeNode<T>* root){
  using std::stack;
  stack<BinaryTreeNode<T>*> aStack;
  BinaryTreeNode<T>* pointer=root;
  while(!aStack.empty()||pointer){
      if(pointer){
          aStack.push(pointer);
          pointer=pointer->leftchild();
      }
      else{
          pointer=aStack.top();
          aStack.pop();
          pointer=pointer->rightchild();
      }
  }
}
```

- 后序：给栈中元素加上一个特征位，Left 表示已进入该结点的左子树，将从左边回来；Right 表示已进入该结点的右子树，将从右边回来

```
enum Tags{Left,Right};
template<class T>
class StackElement{
public:
  BinaryTreeNode<T>* pointer;
  Tags tag;
};
template<class T>
void BinaryTree<T>::PostOrderWithoutRecusion(BinaryTreeNode<T>* root){
  using std::stack;
  StackElement<T> element;
  stack<StackElement<T>> aStack;
```

```
    BinaryTreeNode<T>* pointer;
    pointer=root;
    while(!aStack.empty()||pointer){
        while(pointer!=NULL){
            element.pointer = pointer;
            element.tag = Left;
            aStack.push(element);
            pointer = pointer->leftchild();
        }
        element = aStack.top();
        aStack.pop();
        pointer = element.pointer;
        if (element.tag == Left) {
            element.tag = Right;
            aStack.push(element);
            pointer = pointer->rightchild();
        }
        else {
            Visit(pointer->value());
            pointer = NULL;
        }
    }
  }
```

- 二叉树遍历算法的空间代价分析
  - 深搜：栈的深度与树的高度有关，最好$O(\log n)$；最坏$O(n)$

三 • BFS 遍历二叉树

1.

```
void BinaryTree<T>::LevelOrder(BinaryTreeNode<T>* root){
    using std::queue;
    queue<BinaryTreeNode<T>*> aQueue;
    BinaryTreeNode<T>* pointer=root;
    if(pointer)
        aQueue.push(pointer);
    while(!aQueue.empty()){
        pointer=aQueue.front();
        aQueue.pop();
        Visit(pointer->value());
        if(pointer->leftchild())
            aQueue.push(pointer->leftchild());
        if(pointer->rightchild())
            aQueue.push(pointer->rightchild());
    }
}
```

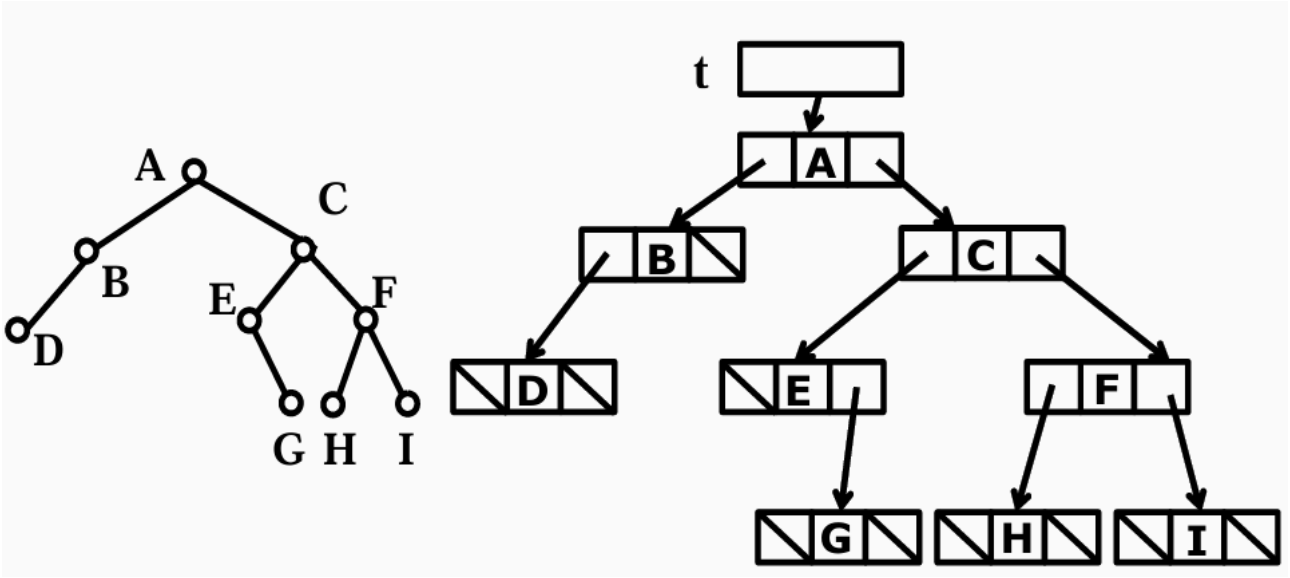
- 时间代价分析：$O(n)$
- 时间代价分析：最好$O(1)$；最坏$O(n)$

5.3 二叉树的存储结构

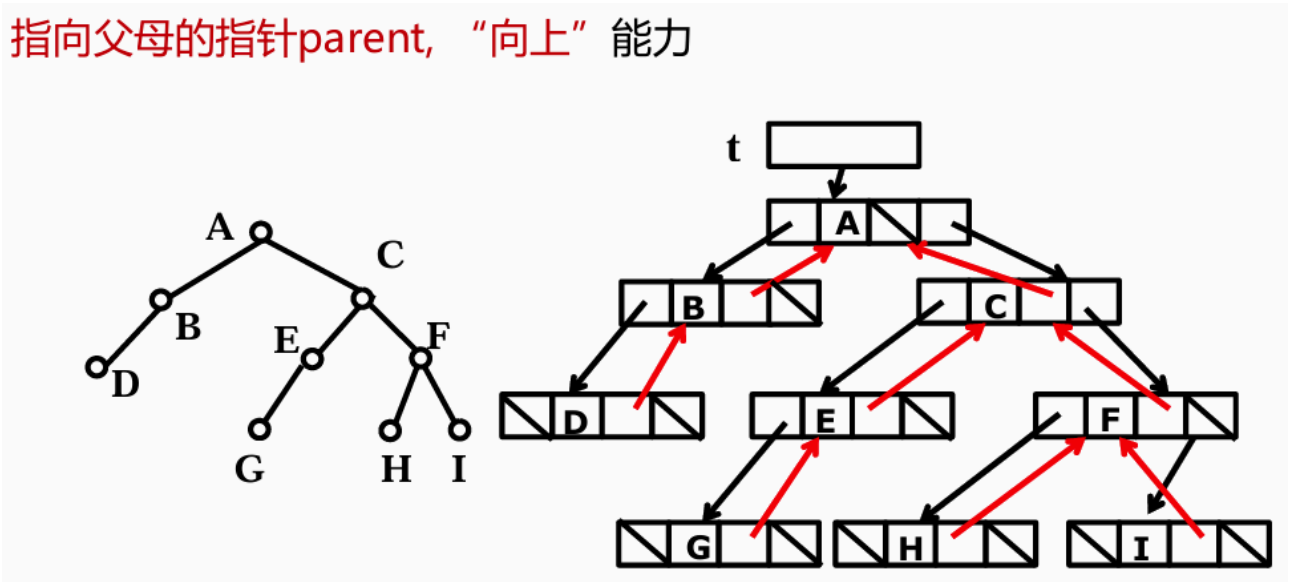
一·链式存储结构（二叉树的各结点随机地存储在内存空间中，结点之间的 逻辑关系用指针来链接）

1.二叉链表

- 指针 left 和 right，分别指向结点的左孩子和右孩子



2.三叉链表



3.BinaryTreeNode 类中增加两个私有数据成员

```
private:
    BinaryTreeNode<T> *left;
    BinaryTreeNode<T> *right;
```

4.递归框架寻找父结点——注意返回

```
template<class T>
BinaryTreeNode<T>* BinaryTree<T>::Parent(BinaryTreeNode<T>* rt,BinaryTreeNode<T>*
current){
```

```
    BinaryTreeNode<T>* tmp;
    if(rt==NULL)
        return (NULL);
    if(current==rt->leftchild()||current==rt->rightchild())
        return rt;
    if ((tmp =Parent(rt- >leftchild(), current) != NULL)
        return tmp;
    if ((tmp =Parent(rt- > rightchild(), current) != NULL)
        return tmp;
    return NULL;
}
```

5. 非递归框架找父结点

```
BinaryTreeNode<T>* BinaryTree<T>::Parent(BinaryTreeNode<T> *current)  {
    using std::stack;
    stack<BinaryTreeNode<T>*> aStack;
    BinaryTreeNode<T>* pointer = root;
    aStack.push(NULL);
    while (pointer) {
        if (current == pointer->leftchild() || current == pointer->rightchild())
            return pointer;
        if (pointer->rightchild() != NULL)
            aStack.push(pointer->rightchild());
        if (pointer->leftchild() != NULL)
            pointer = pointer->leftchild();
        else {
            pointer=aStack.top();
            aStack.pop();
        }
    }
}
```

6. 空间开销分析

- 存储密度$\alpha(\leq 1)$表示数据结构存储的效率，α =数据本身存储量/整个结构占用的存储总量
- 结构性开销$\gamma = 1 - \alpha$
- 每个结点存两个指针，一个指针域
  - 总空间：$(2p + d)n$
  - 结构性开销：$2pn$
- C++ 可以用两种方法来实现不同的分支与叶结点：
  - 用 union 联合类型定义
  - 使用 C++的子类来分别实现分支结点与叶结点，并采用虚函数 isLeaf 来区别分支结点与叶结点
- 早期节省内存资源:
  - 利用结点指针的一个空闲位（一个 bit）来标记结点所属的类型
  - 利用指向叶的指针或者叶中的指针域来存储该叶结点的值

7. 完全二叉树的下标公式（易推）

5.4 二叉搜索树（BST）

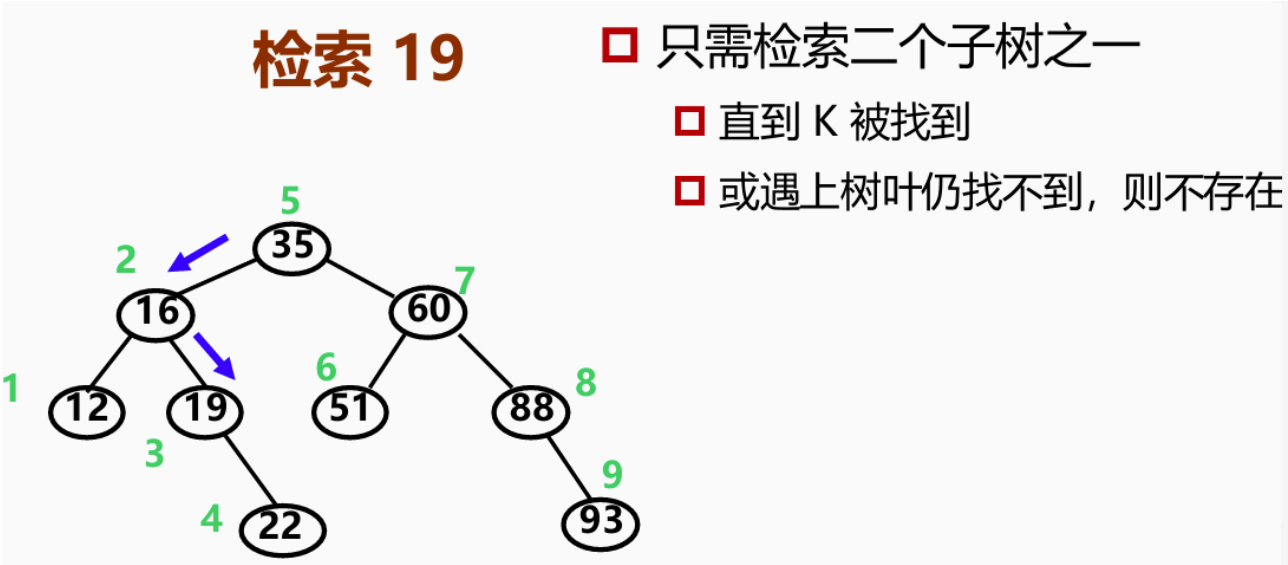
一·基本概念

1. 定义：或者是一棵空树；或者是具有下列性质的二叉树：对于任何一个结点，设其值为 K； 则该结点的 左子树(若不空)的任意一个结点的值都 小于 K；该结点的 右子树(若不空)的任意一个结点的值都 大于 K；而且它的左右子树也分别为 BST

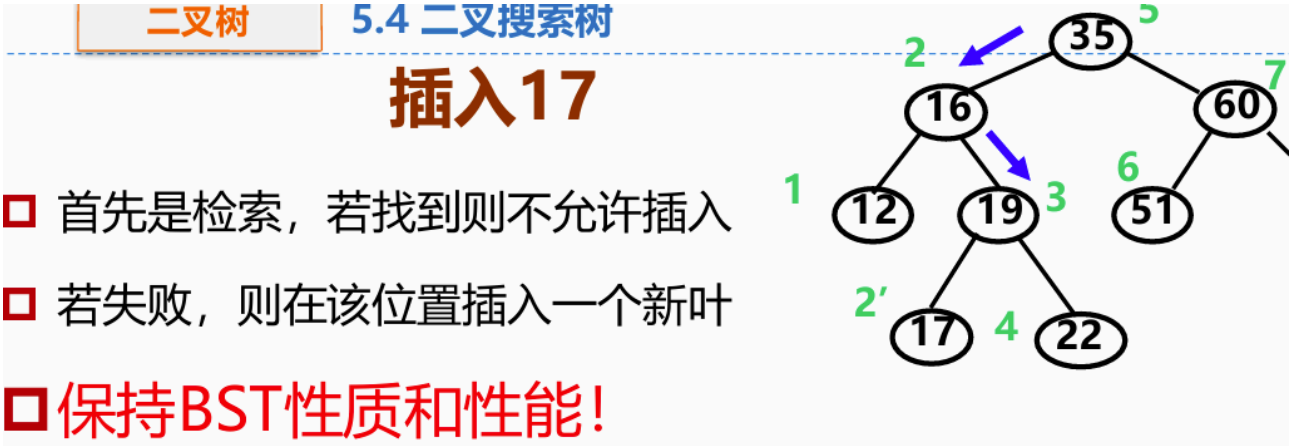
2. 性质： 中序遍历是正序的（由小到大的排列）

3. 功能：

- 检索



- 插入



- 删除
  - 度为 0：直接删除
  - 度为 1：用其子节点替代自身
  - 度为 2：

  a. 寻找替代节点：
  - 后继节点：目标节点右子树中的最小节点（即右子树的最左节点）。

- 前驱节点：目标节点左子树中的最大节点（即左子树的最右节点）。
- 任选一种方式，通常使用后继节点

b. 替换与删除：
- 将目标节点的值替换为后继/前驱节点的值。递归删除右子树（或左子树）中的后继/前驱节点（此时该节点必为叶子或单子节点，转为情况 1 或 2）。

4. 总结
- 组织内存索引
  - 适用于内存储器，常用红黑树、伸展树等，以维持平衡
  - 外存常用 B/B+树
- 保持性质 vs 保持性能

5.5 堆与优先队列

一 • 堆

1. 最小堆定义：对任意节点，其值小于或等于其子节点的值的完全二叉树

2. 性质
- 完全二叉树的层次序列，可以用数组表示
- 堆中储存的数是局部有序的，堆不唯一
- 从逻辑角度看，堆实际上是一种树形结构

3. 类定义

```
template<class T>
class MinHeap{
private:
    T* heapArray;
    int CurrentSize;
    int MaxSize;
    void BuildHeap();
public:
    MinHeap(const int n);
    virtual ~MinHeap(){delete []heapArray;};
    bool isLeaf(int pos)const;
    int leftchild(int pos)const;
    int rightchild(int pos)const;
    int parent(int pos)const;
    bool Remove(int pos,T& node);
    bool Insert(const T& newNode);
    T& RemoveMin();
    void SiftUp(int position);
    void SiftDown(int left);
}
```

4. 对最小堆用筛选法 SiftDown 调整

```
template<class T>
void MinHeap<T>::SiftDown(int position){
    int i=position;
    int j=2*i+1;
    int Ttemp=heapArray[i];
    while (j < CurrentSize) {
```

```
        if((j < CurrentSize-1)&&(heapArray[j] > heapArray[j+1]))
            j++;  // j指向数值较小的子结点
        if (temp > heapArray[j]) {
            heapArray[i] = heapArray[j];
            i = j;
            j = 2*j + 1;     // 向下继续
        }
        else break;
    }
    heapArray[i]=temp;
}
```

5. 对最小堆用筛选法 SiftUp 向上调整

```
template<class T>
void MinHeap<T>::SiftUp(int position){
    int temppos=position;
    T temp=heapArray[temppos];
    while((temppos>0) && (heapArray[parent(temppos)] > temp))   {
        heapArray[temppos]=heapArray[parent(temppos)];
        temppos=parent(temppos);
    }
    heapArray[temppos]=temp;
}
```

6. 建最小堆过程

- 方法

🔧 二、自底向上建堆法（高效批量构建）

适用场景：已知完整数组，时间复杂度 O(n)（优于逐次插入的 O(n log n)） [2] [5] [7] 。

步骤：

1. **定位起点**：
   - 从最后一个非叶子节点开始（索引 `start = 数组长度//2 - 1`）。
   - *例*：数组 `[50, 40, 90, 70, 60, 80]` 的最后一个非叶节点索引为 `6//2 - 1 = 2`（值 `90`） [2] [7] 。
2. **从后向前逐节点下沉（Sift Down）**：
   - 比较当前节点与其左右子节点，若父节点 > 子节点，则与**更小的子节点**交换 [4] [9] 。
   - 递归调整被交换的子树，直到满足堆性质。
   - *流程示例* [1] [4] ：
     - 调整节点 `90`（索引 2）：与子节点 `80` 交换 → `[50,40,80,70,60,90]`。
     - 调整节点 `40`（索引 1）：子节点 `70,60` 均 ≥40，无需交换。
     - 调整节点 `50`（索引 0）：与子节点 `40` 交换 → `[40,50,80,70,60,90]`，再调整子树（节点 `50` 与 `60` 交换）→ `[40,60,80,70,50,90]`。

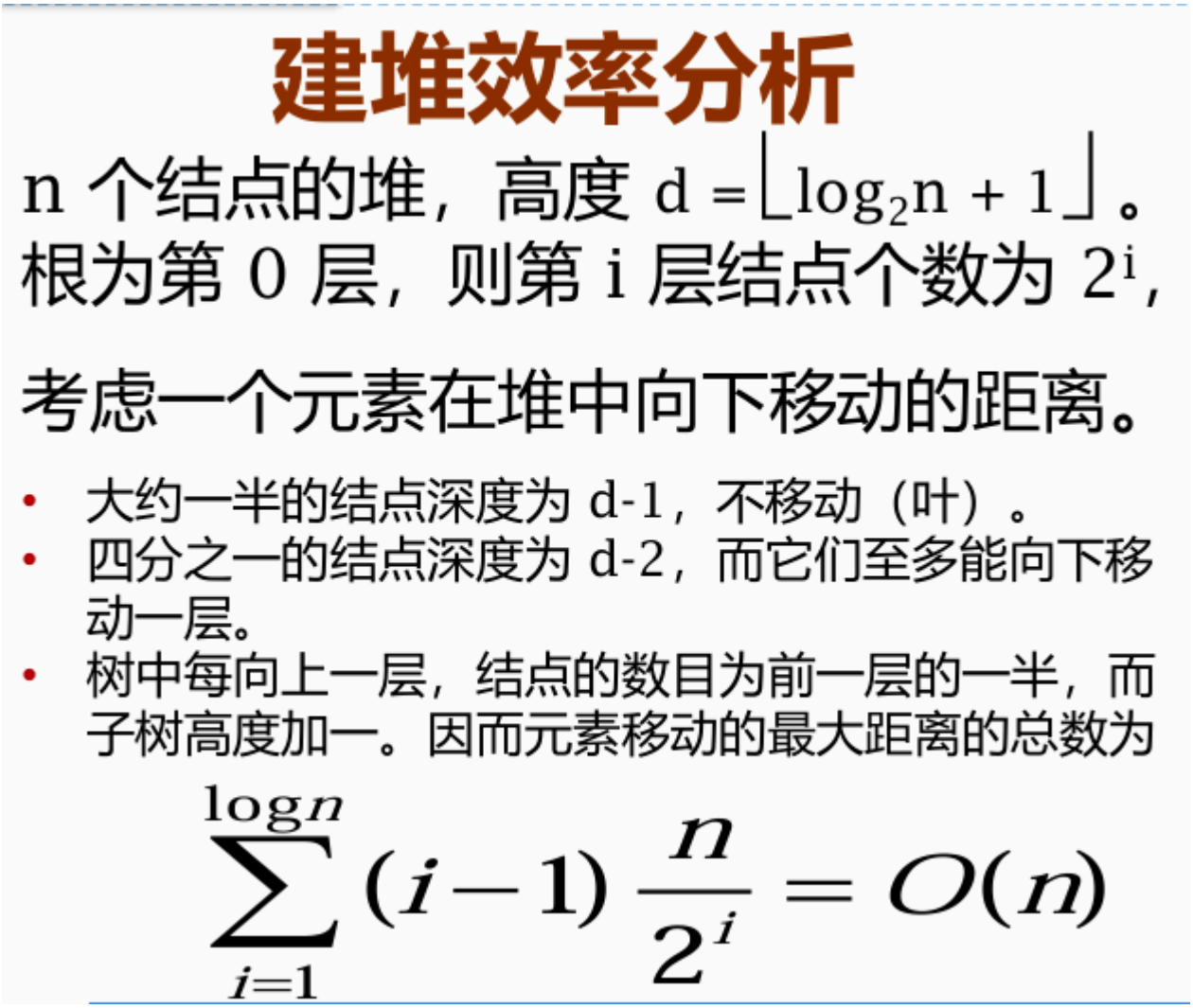
➕ 三、边插入边建堆法（动态构建）

适用场景：数据流式输入，单次插入时间复杂度 O(log n) [1] [6] [9] 。

步骤：

1. **插入新元素**：追加到数组末尾 [4] [6] 。
2. **上浮调整（Sift Up）**：
   - 比较新节点与其父节点，若子节点 < 父节点，则交换位置 [5] [8] 。
   - 重复上浮直到根节点或满足堆性质。
   - *示例*（依次插入 `[5, 3, 8, 1]`） [4] [6] ：
     - 插入 `5` → `[5]`（根节点）。
     - 插入 `3` → 与父节点 `5` 交换 → `[3,5]`。
     - 插入 `8` → 父节点 `3` 更小，不交换 → `[3,5,8]`。
     - 插入 `1` → 先与父节点 `5` 交换 → `[3,1,8,5]`，再与父节点 `3` 交换 → `[1,3,8,5]`。

- 操作
  - 删除特定元素
  - 插入特定元素
- 建堆效率分析
  - 建堆算法时间代价：$O(n)$



- 插入结点、删除普通元素和删除最小元素的平均时间代价和最差时间代价都是$O(\log(n))$

7. 堆的应用
- 优先队列：堆的应用之一，支持插入、删除最小元素、查找最小元素等操作
- 堆排序：利用堆的性质进行排序，时间复杂度为$O(n\log n)$
- 图算法：Dijkstra 算法、Prim 算法等都使用堆来优化性能

5.6 Huffman 树及其应用

一・等长编码

1. 定义：每个字符的编码长度相同的编码方式

2. 包括：ASCII 码、中文编码等

3. 表示 n 个不同字符需要 n 个二进制码字，长度为 $\log_2(n)$ 位

二・数据压缩与不等长编码

1. 特点：可以利用字符的出现频率来编码， 经常出现的字符的编码较短，不常出现的字符编码较长

2. 优点：数据压缩既能节省磁盘空间，又能提高运算速度

三・前缀编码

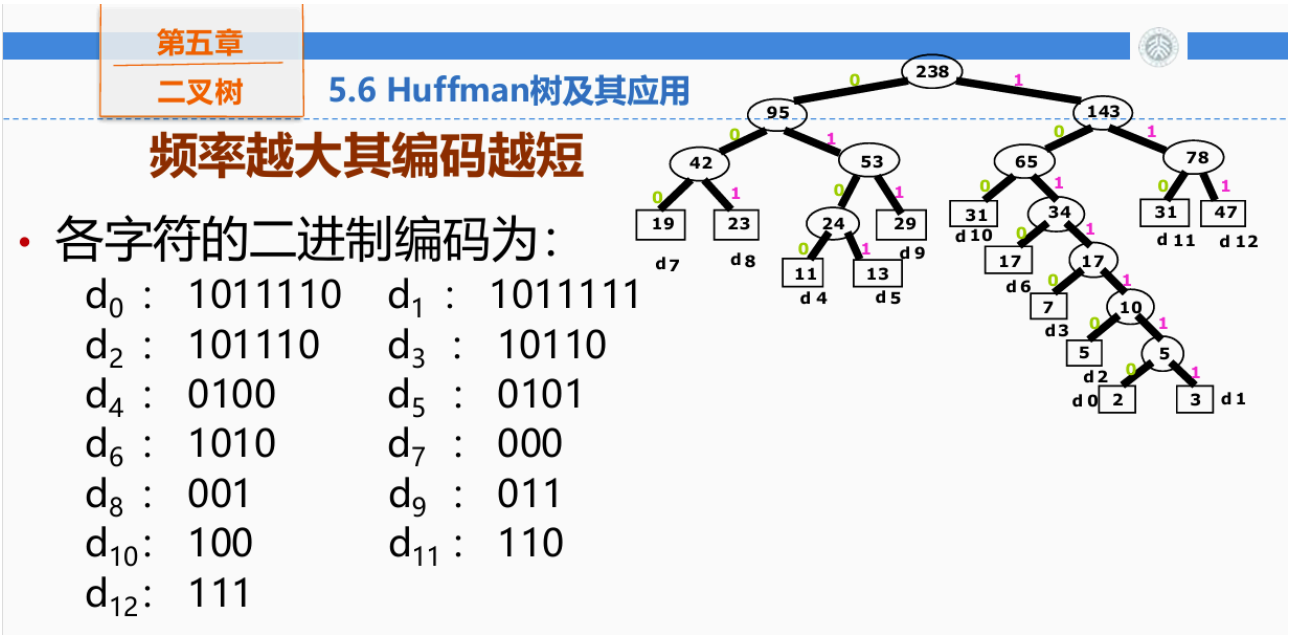
1. 定义：任何一个字符的编码都不是另外一个字符编码的前缀

2. 特点：前缀编码可以唯一地表示一个字符串，且不会产生歧义

四・Huffman 树与前缀编码

1. 建立 Huffman 树

- 首先，按照“权”(例如频率)将字符排为一列
- 然后，选择权值最小的两个结点作为左右子树，构造一个新结点，其权值为两子树权值之和
- 重复上述过程，直到所有结点合并为一棵树



2. 译码

- 从左至右逐位判别代码串，直至确定一个字符
- 译出了一个字符，再回到树根，从二进制位串中的下一位开始继续译码

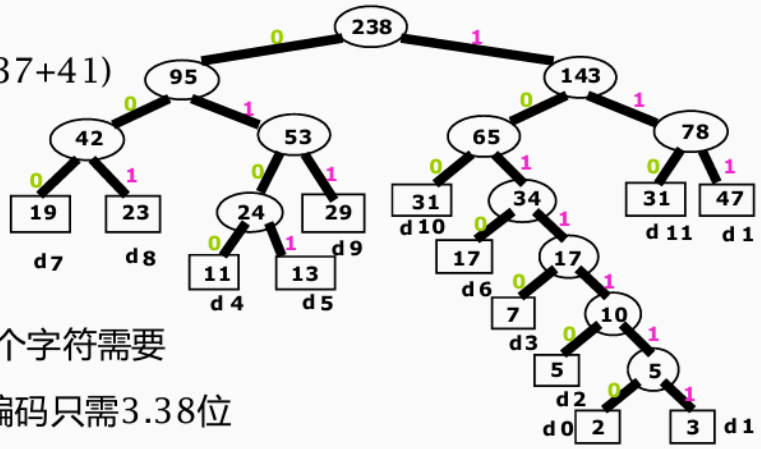
3. huffman 性质

- 含有两个以上结点的一棵 Huffman 树中，字符使用频率最小的两个字符是兄弟结点，而且其深度不比树中其他任何叶结点浅
- 对于给定的一组字符，函数 HuffmanTree 实现了“最小外部路径权重”

4. Huffman 树编码效率

Huffman树编码效率（续）

- 图中，平均代码长度为：
  (3*(19+23+24+29+31+34+37+41)
  + 4*(11+13+17)
  + 5*7
  + 6*5
  + 7*(2+3)) / 238
  = 804/238 ≈ 3.38



- 对于这13个字符，等长编码每个字符需要
  ⌈log 13⌉ = 4位，而Huffman编码只需3.38位
- Huffman编码预计只需要等长编码3.38/4≈ 84% 的空间

5. 应用
- 数据压缩：如 ZIP、RAR 等文件压缩格式
- 图像压缩：如 JPEG 图像格式
- 音频压缩：如 MP3、AAC 等音频格式

第 6 章：树

6.1 树的定义和基本术语

一・树和森林

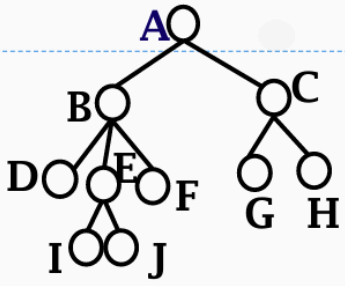
1. 定义：树是包含 n 个节点的有限集合 T，其中 n>=1，且满足以下性质：
- 存在一个特定的节点称为根节点(root)，其余节点分为若干互不相交的子集，每个子集也是一棵树，称为根节点的子树(subtree)
- 每个节点有且仅有一个父节点(parent)，根节点没有父节点

2. 有向有序树：子树的相对次序是重要的

3. 度为 2 的有序树并不是二叉树

4. 树的逻辑结构

树的逻辑结构

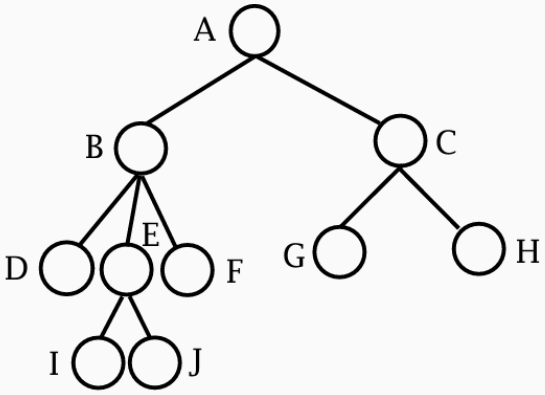


- 包含n个结点的有穷集合 K (n>0)，且在 K上定义了一个关系 r，关系 r 满足以下条件：
  - 有且仅有一个结点 $k_0 \in K$，它对于关系 r 来说没有前驱。结点 k_0 称作树的 根
  - 除结点 k_0 外，K中的每个结点对于关系 r 来说都有且仅有 一个前驱
- 例如，
  - 结点集合 K={ A，B，C，D，E，F，G，H，I，J }
  - K 上的关系 r = { <A，B>，<A，C>，<B，D>，<B，E>，<B，F>，<C，G>，<C，H>，<E，I>，<E，J> }

5. 树的相关术语
- 结点
  - 子结点、父结点、最左子结点
  - 兄弟结点、左兄弟、右兄弟
  - 分支结点、叶结点
    - 没有子树的结点称作叶结点（或树叶、终端结点）
    - 非终端结点称为分支结点
- 边：两个结点的有序对
- 路径、路径长度
  - 除结点 k0 外的任何结点 k∈K，都存在一个结点序列 k0，k1，…，ks，使得 k0 就是树根，且 ks=k，其中有序对 $< k_i - 1, k_i > \in$ r（1≤i≤s）。这样的结点序列称为从根到结点 k 的一条路径，其路径长度为 s（包含的边数）
- 祖先，后代
  - 若有一条由 k 到达 ks 的路径，则称 k 是 ks 的祖先，ks 是 k 的子孙
- 层数
  - 根为第 0 层，其他结点的层数等于其父结点的层数加 1
- 深度
  - 层数最大的叶结点的层数
- 高度
  - 层数最大的叶结点的层数加 1
- 度数
  - 结点的度数是该结点子树的个数
  - 树的度数是树中所有结点的度数的最大值

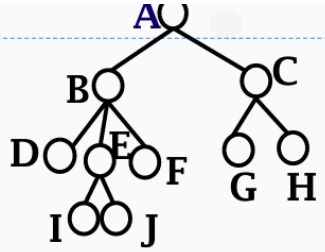
6. 树形结构的各种表示法
- 树形

树形表示法



- 形式语言

形式语言表示法



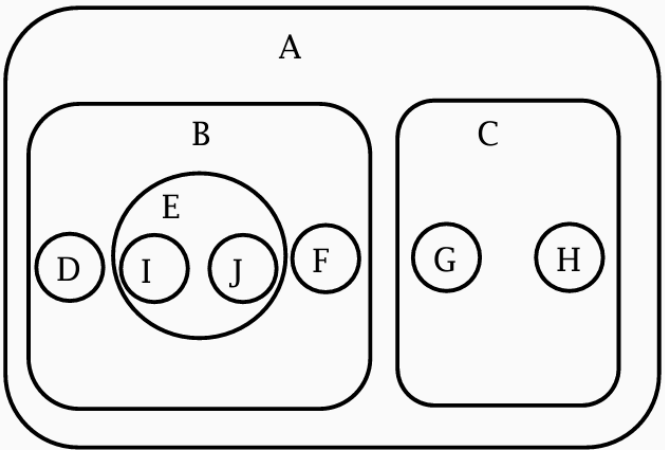
树的逻辑结构是：

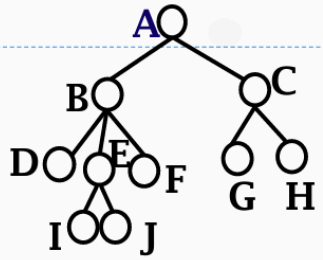
结点集合K={A，B，C，D，E，F，G，H，I，J}

K上的关系N={<A，B>，<A，C>，<B，D>，
<B，E>，<B，F>，<C，G>，
<C，H>，<E，I>，<E，J>}

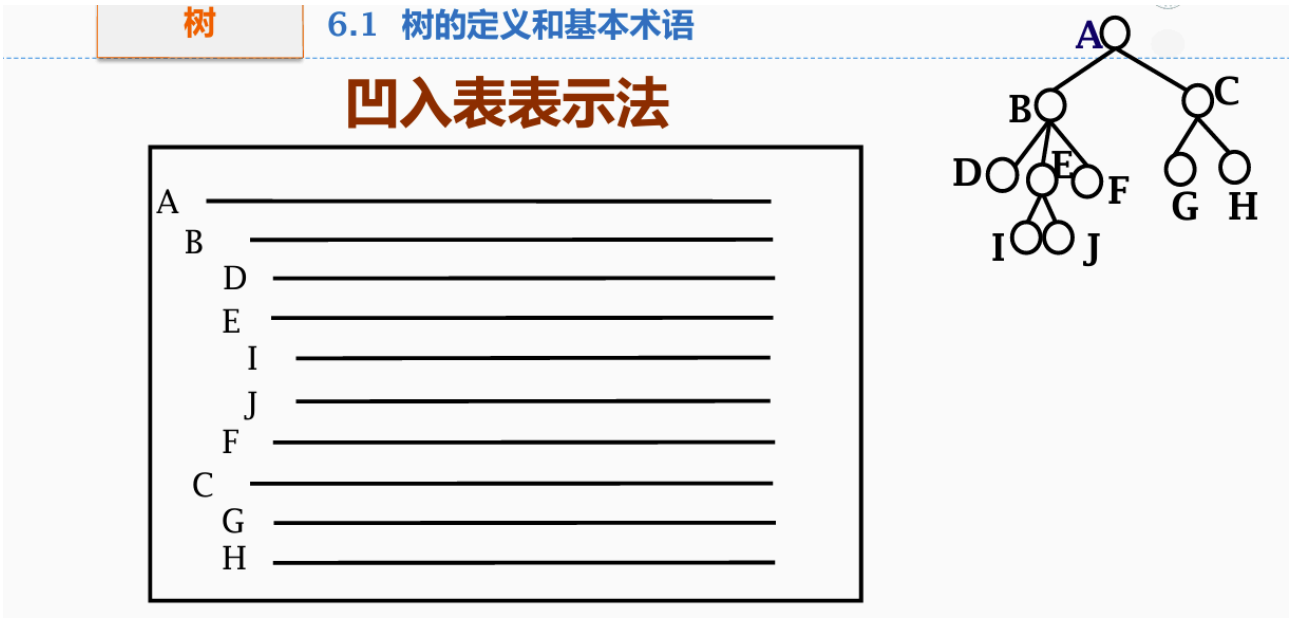
- 文氏图

文氏图表示法

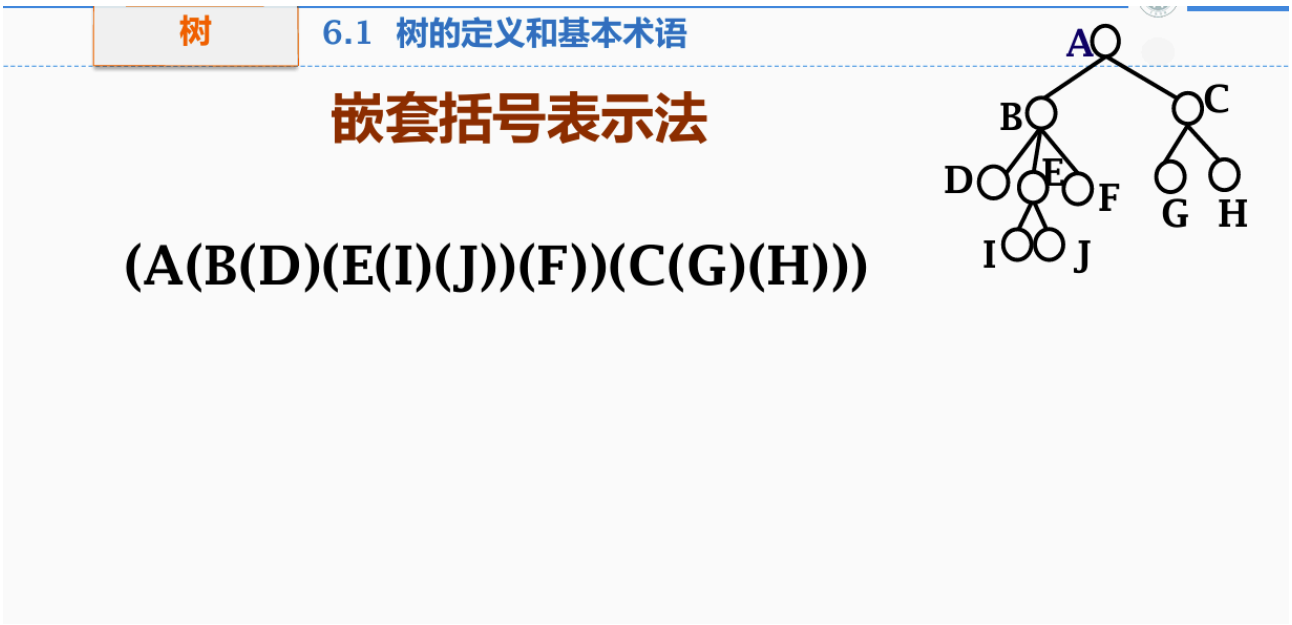




- 凹入表

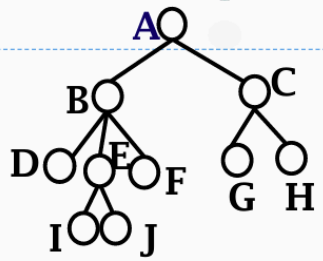


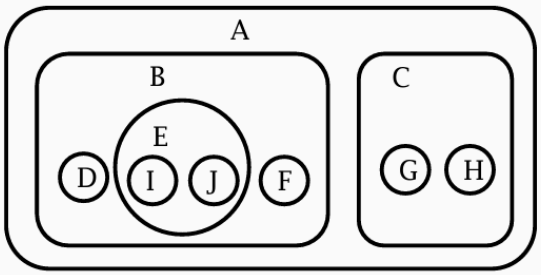
- 嵌套括号



- 文氏图到嵌套括号的转化

文氏图到嵌套括号表示的转化





(A(B(D)(E(I)(J))(F))(C(G)(H)))

二·森林和二叉树的等价转换

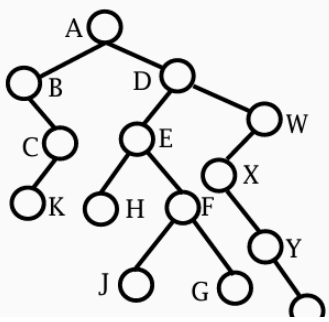
1.森林定义：由 m (m>=0) 棵互不相交的树组成的集合称为森林(forest)

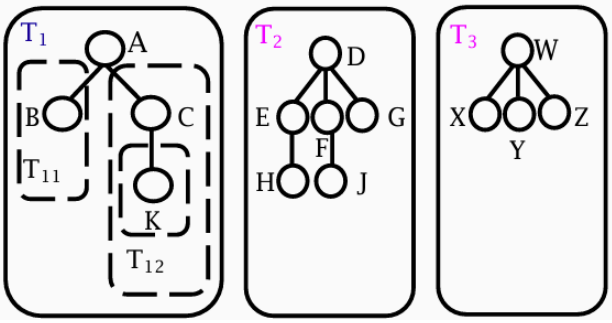
2.根节点加不加的问题罢了

3.森林转化为二叉树

森林转化成二叉树的形式定义

- 有序集合 F = {T_1, T_2, …, T_n} 是树 T_1, T_2, …, T_n 组成的森林，递归转换成二叉树B(F):
  - 若 F 为空，即 n = 0，则 B(F) 为空。
  - 若 F 非空，即 n > 0，则 B(F) 的根是森林中第一棵树 T_1 的根 W_1，
    B(F) 的左子树是树 T_1 中根结点 W_1的子树森林 F = {T_{11}, …, T_{1m}} 转换成的二叉树B(T_{11}, …, T_{1m});
    B(F)的右子树是从森林 F' = {T_2, …, T_n} 转换而成的二叉树





1. **独立转换每棵树**
   - 将森林中的每棵树分别转换为二叉树：
     - **连接兄弟节点**：为每棵树中同一父节点的所有兄弟节点添加水平连线（虚线）。
     - **删除非长子连线**：仅保留父节点与第一个孩子（长子）的连线，删除父节点与其他孩子的连线。
     - **旋转调整**：以根为轴心顺时针旋转45°，使长子成为左孩子，兄弟成为右孩子。
   - *示例*：

复制

```
树：    A        →  二叉树：  A
     / | \                 /
    B  C  D              B
                        / \
                       E   C
                            \
                             D
```

2. **连接多棵二叉树**
   - 将森林中所有转换后的二叉树按顺序链接：
     - **第一棵树不动**：其根作为最终二叉树的根。
     - **后续树的根作为右孩子**：第二棵树的根链接为第一棵树根的右孩子；第三棵树的根链接为第二棵树根的右孩子，依此类推。
   - *示例*：

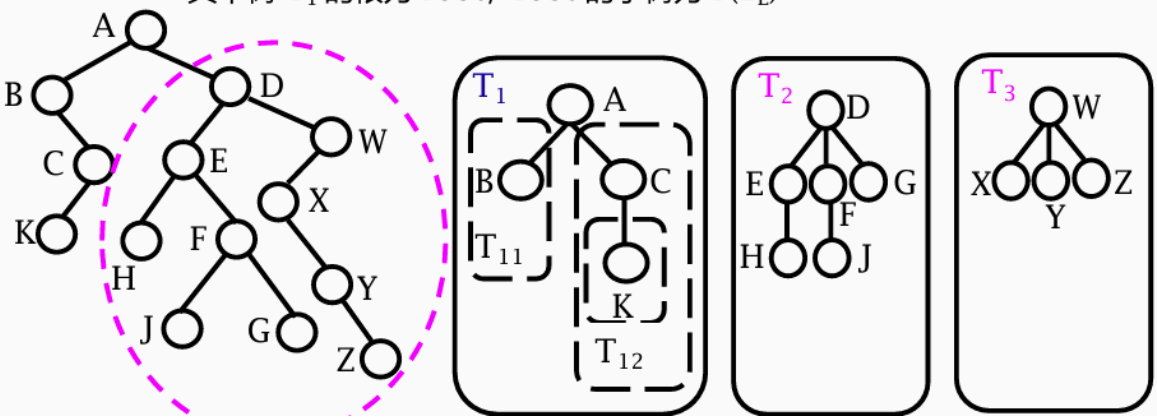
复制

```
森林：[树1：A]    →  二叉树：     A
     [树2：B]              /   \
     [树3：C]            D     B
                                \
                                 C
```

4. 二叉树转化为森林

二叉树转化成森林或树的形式定义

- 设B是一棵二叉树，root是 B 的根，B_L 是 root 的左子树，B_R 是 root 的右子树，则对应于二叉树B的森林或树 F(B) 的形式定义是：
  - 若 B 为空，则 F(B) 是空的森林
  - 若 B 不为空，则 F(B)是一棵树T_1加上森林 $F(B_R)$，其中树 T_1 的根为 root，root 的子树为 $F(B_L)$



🔍 核心判断依据

根据二叉树根节点的右子树存在性决定转换结果 [5]：

- **根无右子树** → 转换为一棵树
- **根有右子树** → 转换为森林（多棵树）

🔄 转换步骤

1. 断开所有右链

- 从根节点开始，递归删除所有**右指针连线**，将二叉树拆分为多棵独立的二叉树 [2] [5]：
  - 若当前根节点存在右孩子（如 A→C），则断开 A 与 C 的右链，得到独立子树 A 和 C。
  - 对分离后的每棵子树递归执行此操作（如 C 若有右子树 F，则继续断开 C→F）。

2. 每棵二叉树转为树

对分离后的每棵二叉树执行以下操作 [2] [5]：

- **加线**：
  - 若节点 X 有左孩子 L，则将 L 的所有**右路径节点**（$L \to R_1 \to R_2 \to \dots$）作为 X 的子节点。
  - 用新连线连接 X 与这些节点。
    示例：

```markdown
二叉树片段：  X          →  加线后：  X
            / \                  /|\
           L   R₁               L R₁ R₂
               \
                R₂
```

- **删线**：删除原二叉树中所有**右指针连线**（如 $L \to R_1$、$R_1 \to R_2$ 等）。
- **调整层次**：逆时针旋转，恢复树形结构。

三 • 树的抽象数据类型

```
template<class T>
class TreeNode {
public:
    TreeNode(const T& value);
    virtual ~TreeNode() {};
    bool isLeaf();
    T Value();
    TreeNode<T> *LeftMostChild();
    TreeNode<T> *RightSibling();
    void setValue(const T& value);
    void setChild(TreeNode<T> *pointer);
    void setSibling(TreeNode<T> *pointer);
    void InsertFirst(TreeNode<T> *node);
    void InsertNext(TreeNode<T> *node);
};
```

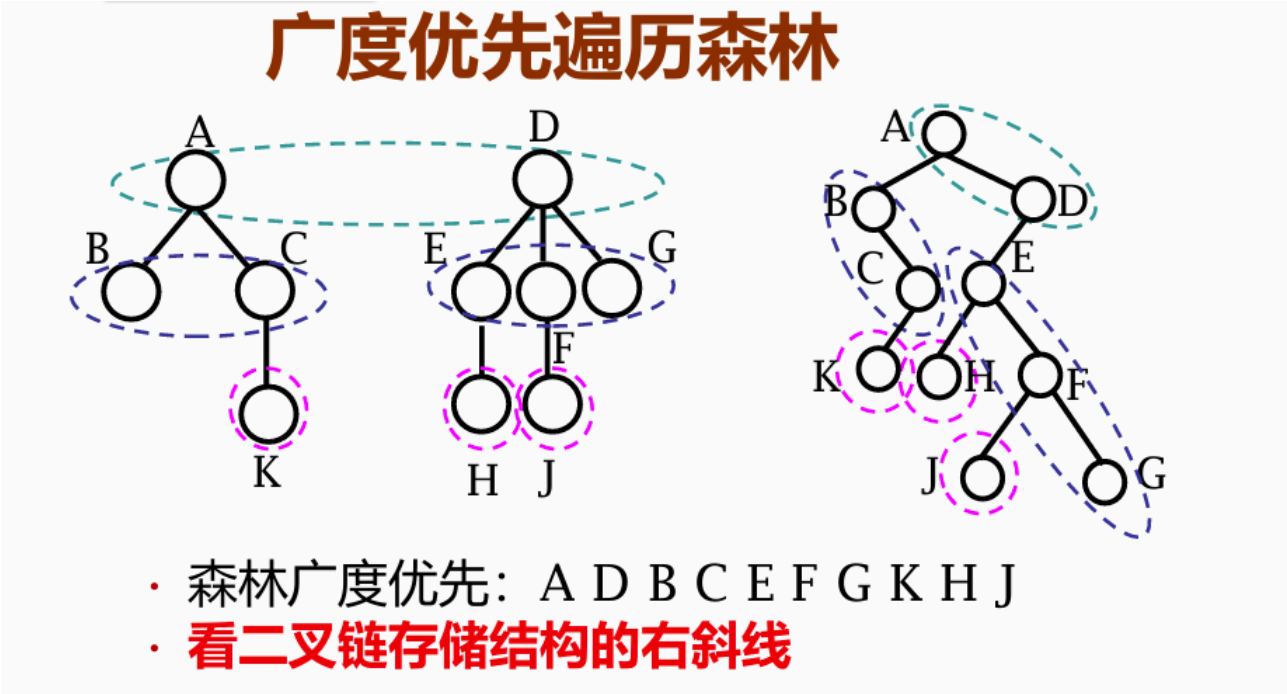
```
template<class T>
class Tree  {
public:
    Tree();
    virtual ~Tree();
    TreeNode<T>* getRoot();
    void CreateRoot(const T& rootValue);
    bool isEmpty();
    TreeNode<T>* Parent(TreeNode<T> *current);
    TreeNode<T>* PrevSibling(TreeNode<T> *current);
    void DeleteSubTree(TreeNode<T> *subroot);
    void RootFirstTraverse(TreeNode<T> *root);
    void RootLastTraverse(TreeNode<T> *root);
    void WidthTraverse(TreeNode<T> *root);
};
```

四·森林的遍历

1.先根深度优先遍历（前序遍历）

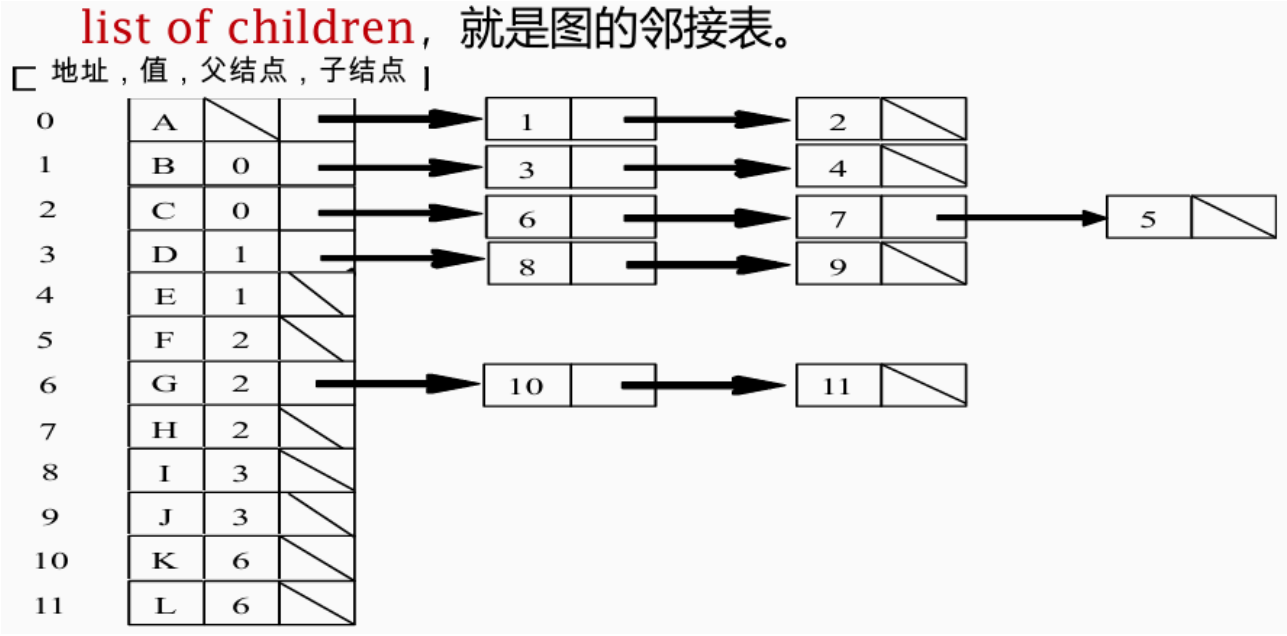
2.后根深度优先遍历（后序遍历）

3.广度优先遍历（层次遍历）

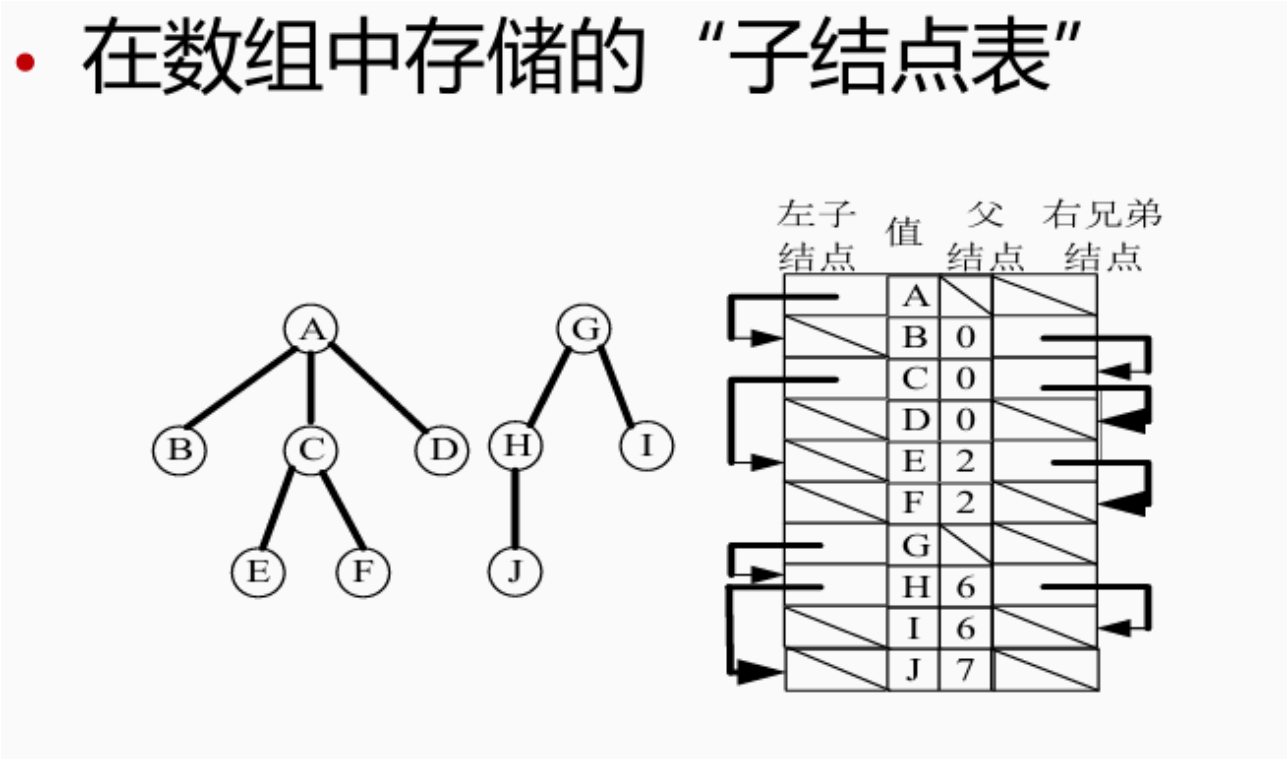


6.2 树的链式存储结构

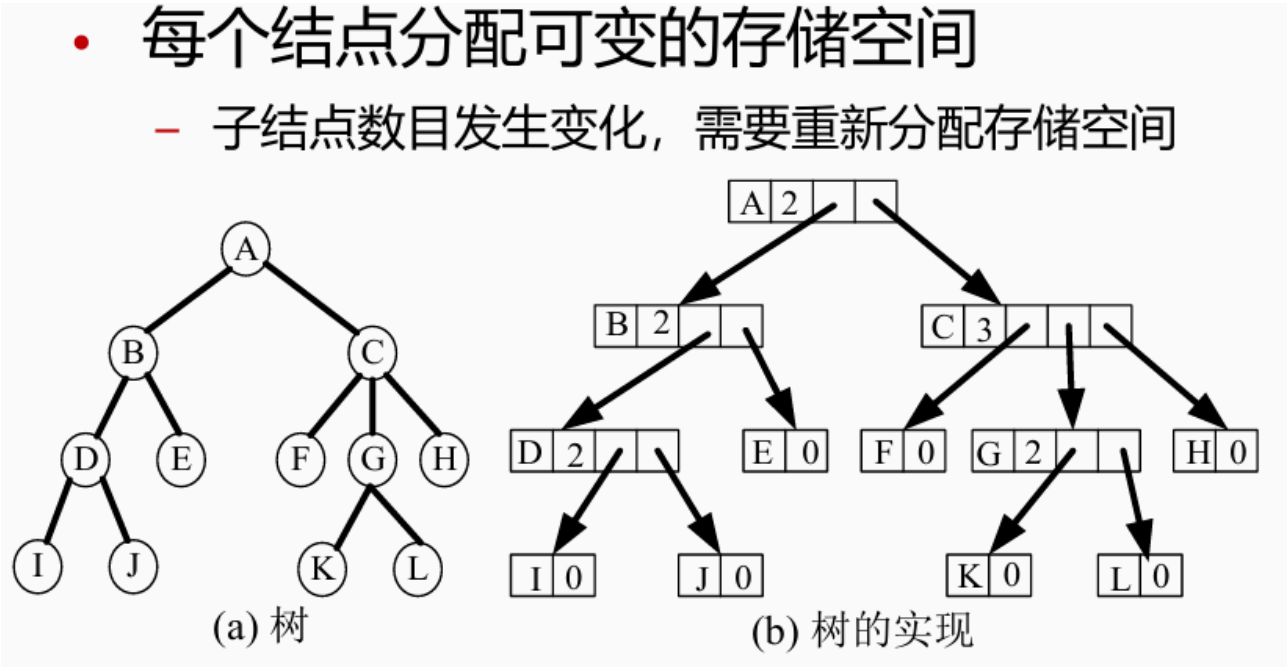
一·“子结点表”表示方法



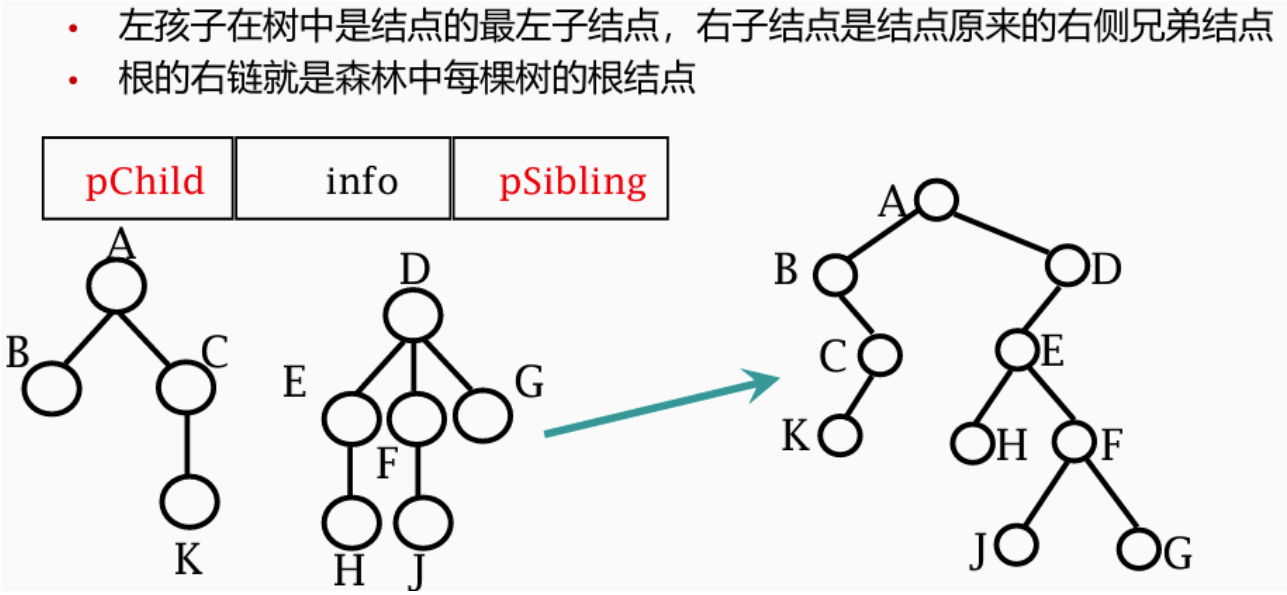
二・静态“左孩子/右兄弟”表示法



三・动态表示法

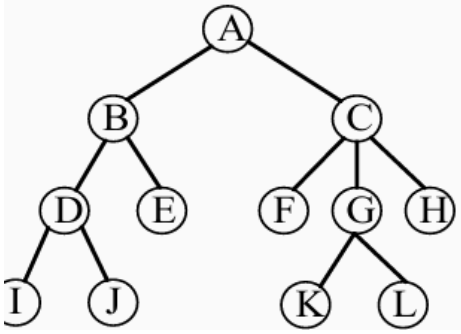


四 • 动态“左孩子/右兄弟”表示法



五 • 父指针表示法及其在并查集中的应用

- 只需要知道父结点的应用
- 只需要保存一个指向其父结点的指针域，称为 父指针(parent pointer)表示法
- 用数组存储树结点，同时在每个结点中附设一个指针指示其父结点的位置



结点索引	0	1	2	3	4	5	6	7	8	9	10	11
值	A	B	C	D	E	F	G	H	I	J	K	L
父结点索引	＼	0	0	1	1	2	2	2	3	3	6	6

1. 并查集
- 定义：一组不相交的集合的集合
- 主要操作：合并两个集合、查询元素所在的集合
- 应用：用于处理等价关系、连通性问题等

2. 等价关系与等价类
- 等价关系：是满足自反性、对称性和传递性的二元关系
- 等价类：相互等价的元素所组成的最大集合

3. 用树来表示等价类的并查
- 用一棵树代表一个集合
  - 集合用父结点代替
  - 若两个结点在同一棵树中，则它们处于同一个集合
- 树的实现
  - 存储在静态指针数组中
  - 结点中仅需保存父指针信息

4. 树的父指针表示与 Union/Find 算法实现

```
template<class T>
class ParTreeNode {
private:
    Tvalue;
    ParTreeNode<T>* parent;
    int nCount;
public:
    ParTreeNode();
    virtual ~ParTreeNode(){};
    TgetValue();
    void setValue(const T& val);
    ParTreeNode<T>*  getParent();
    void setParent(ParTreeNode<T>* par);
    int getCount();
    void setCount(const int count);
```

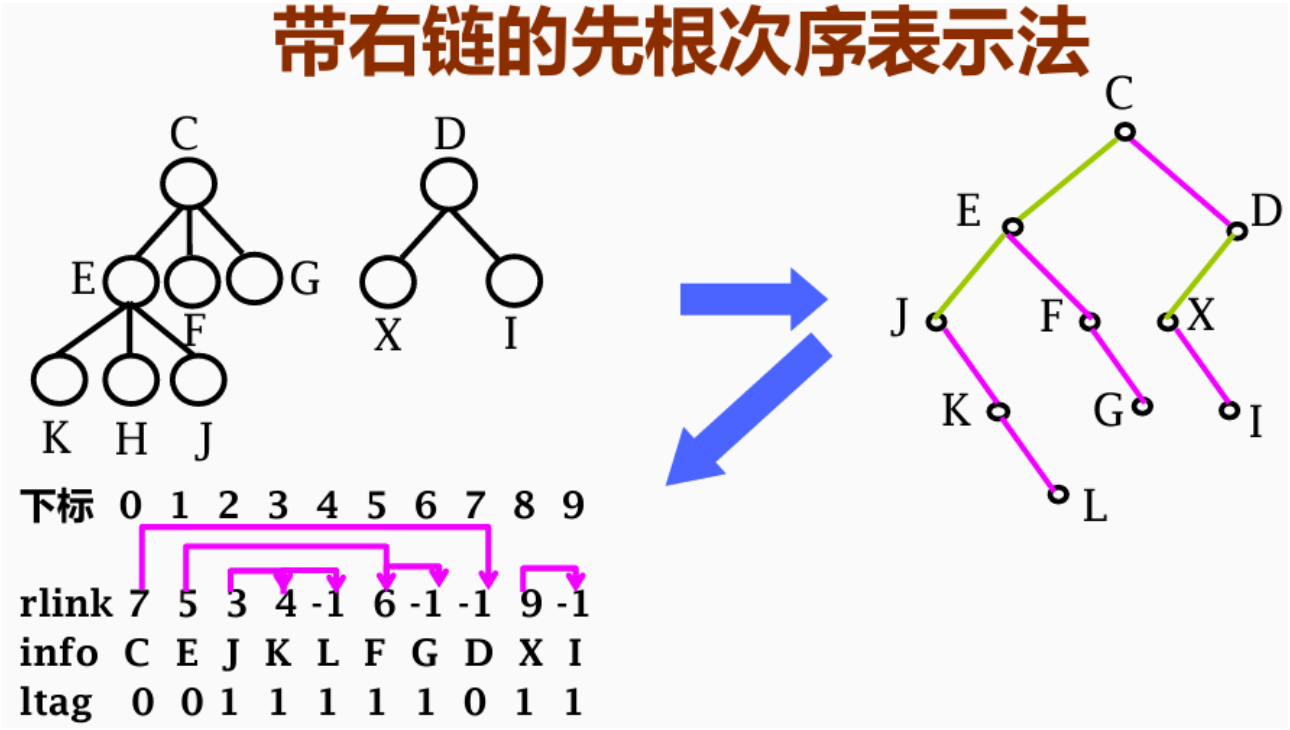
```
};

template<class T>
class ParTree {
public:
    ParTreeNode<T>* array;
    int Size;
    ParTreeNode<T>*
    Find(ParTreeNode<T>* node) const;  // 查找node结点的根结点
    ParTree(const int size);
    virtual ~ParTree();
    void Union(int i,int j);
    bool Different(int i,int j);
};

template <class T>
ParTreeNode<T>* ParTree<T>::Find(ParTreeNode<T>* node) const{
    ParTreeNode<T>* pointer=node;
    while ( pointer->getParent() != NULL )
    pointer=pointer->getParent();
    return pointer;
}

template<class T>
void ParTree<T>::Union(int i,int j) {
    ParTreeNode<T>* pointeri = Find(&array[i]);
    ParTreeNode<T>* pointerj = Find(&array[j]);
    if (pointeri != pointerj) {
        if(pointeri->getCount() >= pointerj->getCount()) {
            pointerj->setParent(pointeri);
            pointeri->setCount(pointeri->getCount() +
            pointerj->getCount());
        }
        else {
            pointeri->setParent(pointerj);
            pointerj->setCount(pointeri->getCount() +
            pointerj->getCount());
        }
    }
}
```

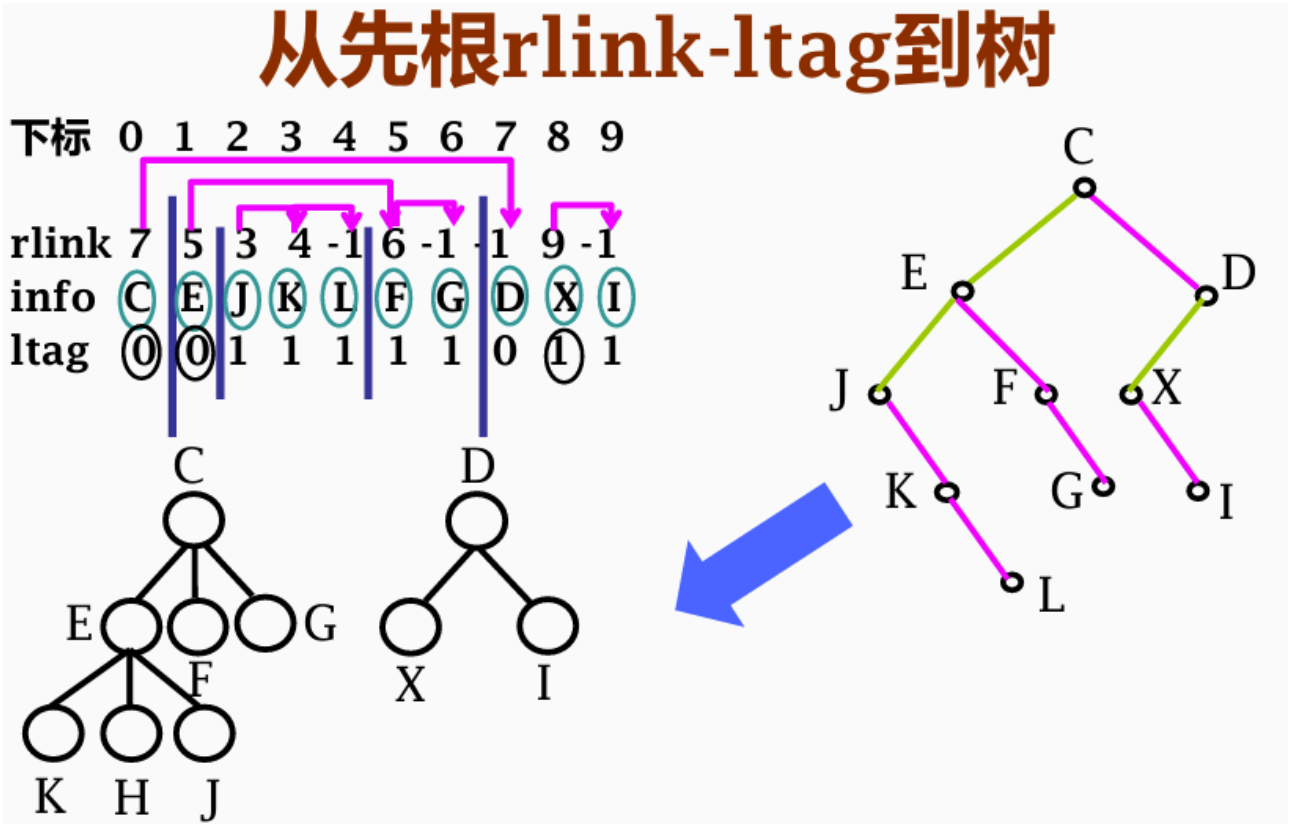
6.3 树的顺序存储结构

一·带右链的先根次序表示

1.结点按先根次序顺序连续存储

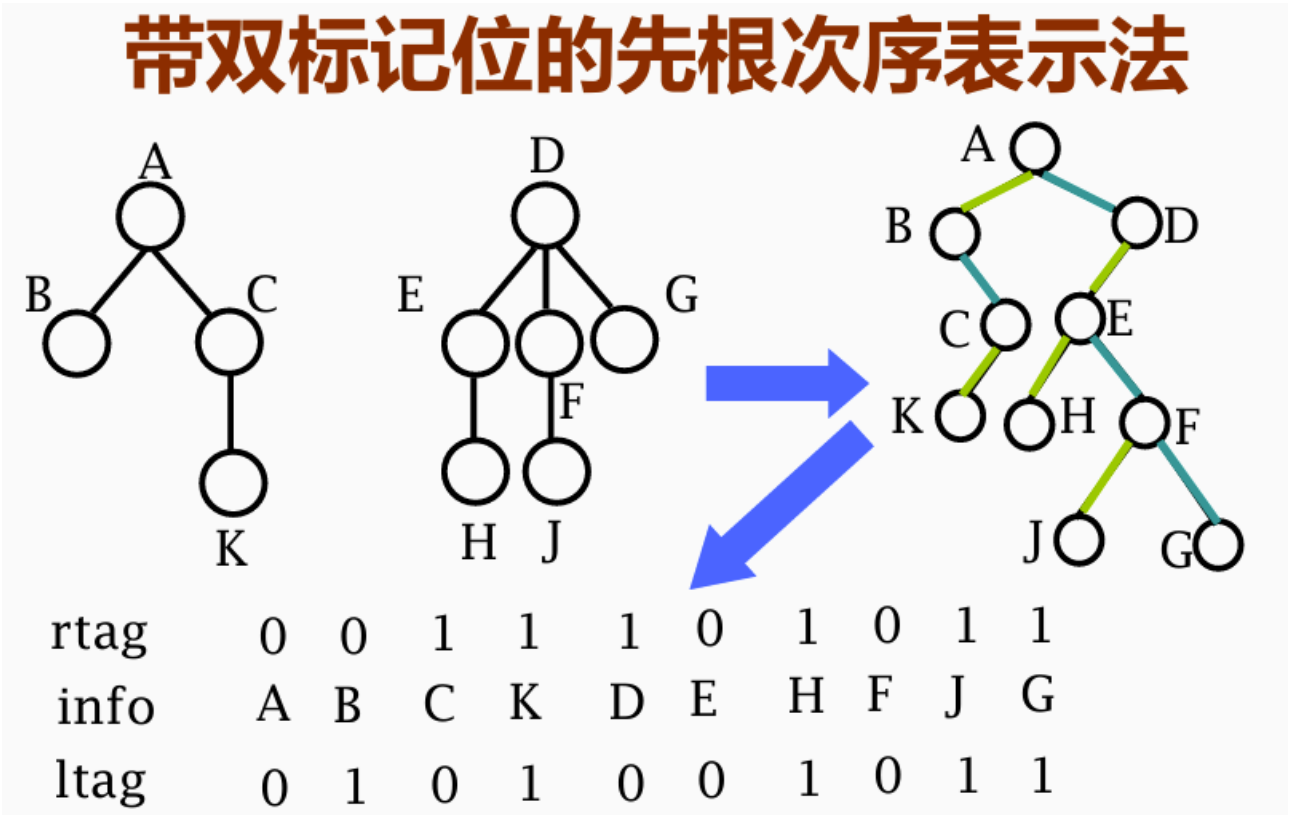
- info：结点的数据
- rlink：右指针
  - 指向结点的下一个兄弟、即对应的二叉树中结点的右子结点
- ltag：标记
  - 树结点没有子结点，即二叉树结点没有左子结点，ltag 为 1；否则为 0

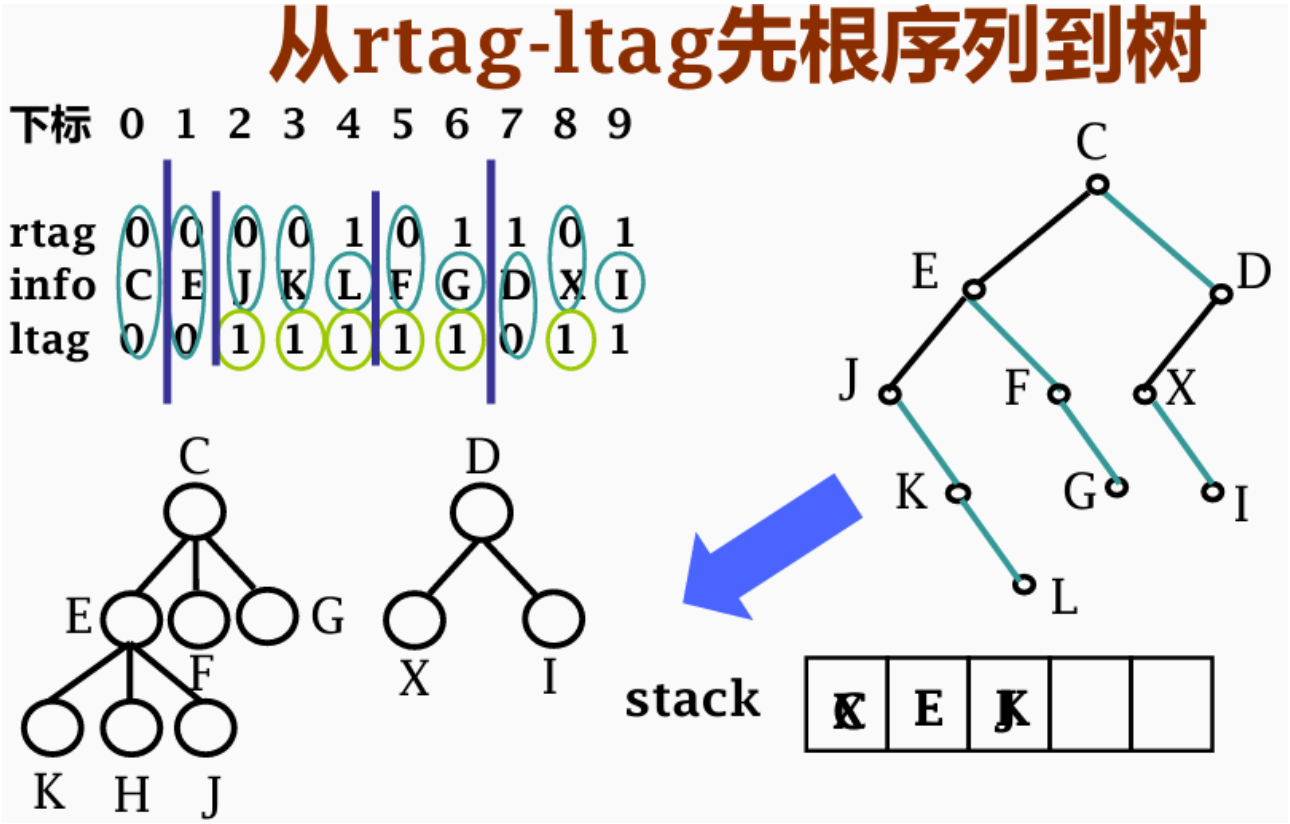




二・带双标记的先根次序表示

- 带右链的先根次序表示”中 rlink 也有冗余，可以把 rlink 指针替换为一个标志位 rtag，成为“带双标记的先根次序表示”。其中，每个结点包括结点本身数据，以及两个标志位 ltag 和 rtag，由结点的先根次序以及 ltag、rtag 两个标志位，就可以确定树“左孩子/右兄弟”链表中结点的 llink 和 rlink 值。其中 llink 的确定与带右链的先根次序表示法相同。



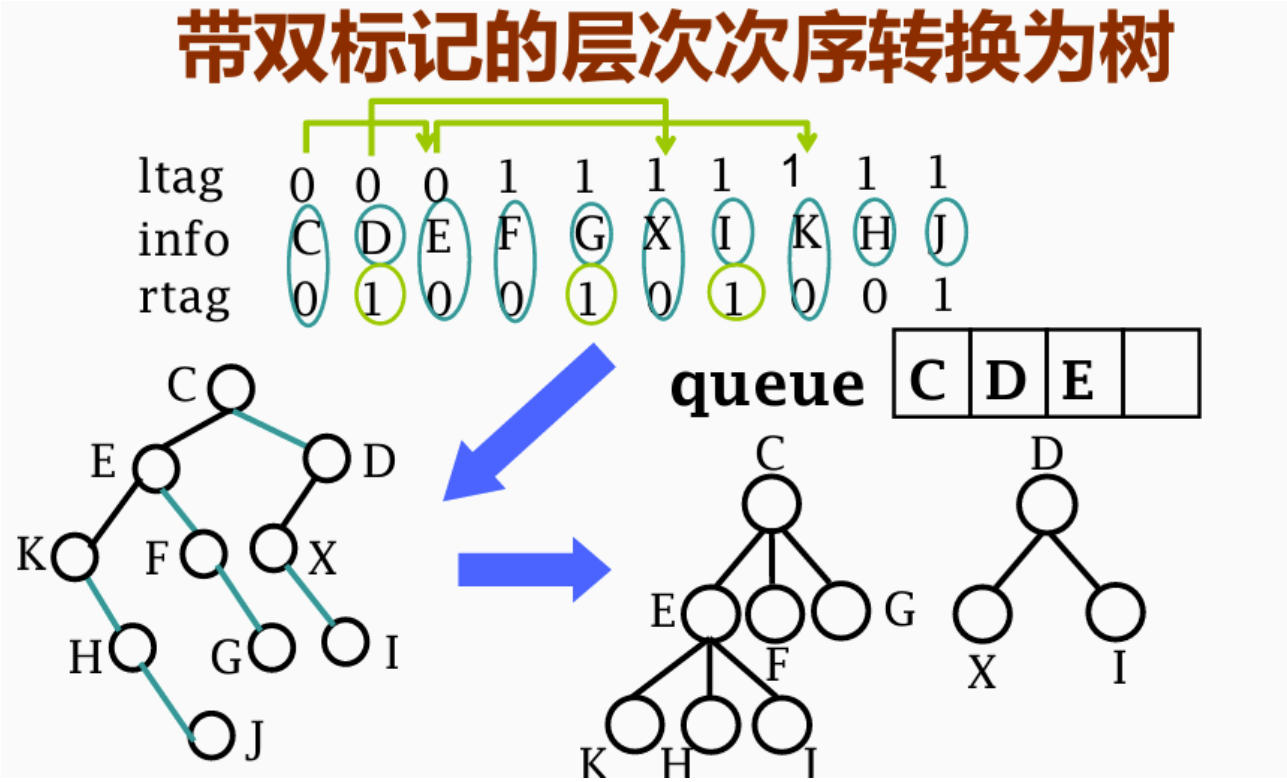


三・带双标记的层次次序表示

1. 结点按层次次序顺序存储在连续存储单元

- info 是结点的数据
- ltag 是一个一位的左标记，当结点没有子节点，即对应的二叉树中结点没有左子结点时，ltag 为 1，否则为 0

- rtag 是一个一位的右标记，当结点没有下一个兄弟，即对应的二叉树中结点没有右子结点时，rtag 为 1，否则为 0



四・带度数的后根次序表示

1. 结点按后根次序顺序存储在一片连续的存储单元中
- info 是结点的数据，degree 是结点的度数

6.4 K 叉树

类比二叉树即可