一：栈（存储结构、栈的操作、应用）

二：队列（存储结构、队列的操作、循环队列、应用）

三：表（存储结构、表的操作、应用）

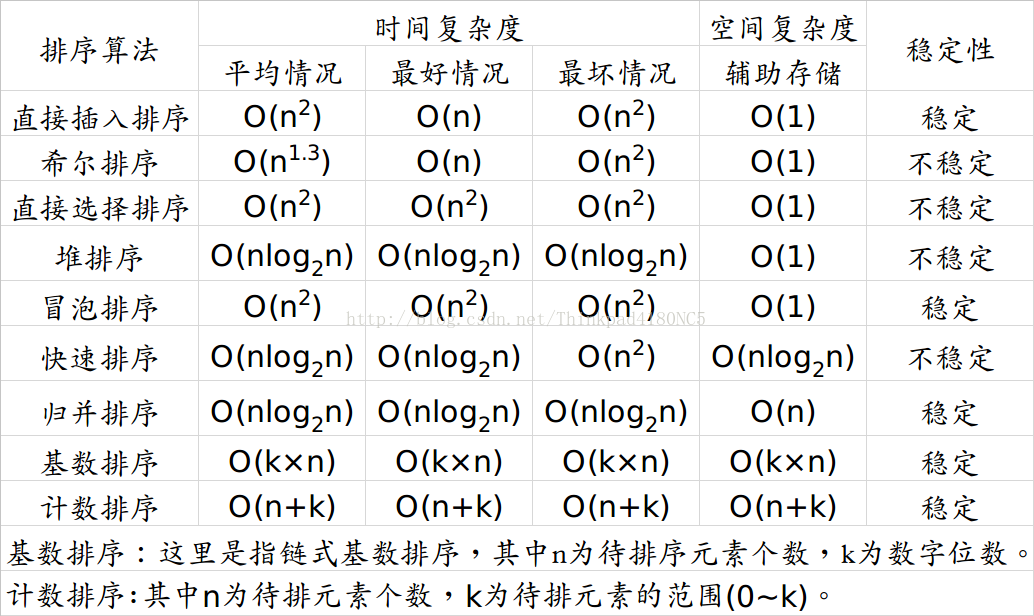
四：树（二叉树、遍历、线索二叉树、二叉搜索树、AVL树、B树）

五：图（图的存储结构、遍历、拓扑排序、最短路径、最小生成树）

六：递归（算法设计）

七：查找（顺序查找、二分查找）

八：排序（插入排序、交换排序、选择排序、归并排序，基数排序）



题型：选择题（2\*20=40分），简答题（10\*3=30分），设计题（10\*3=30分）

考核方式：考试70%，平时成绩30%

练习题：

一．简答题

1．对以下几种内部排序：①直接插入②shell排序③直接选择④堆排序⑤快速排序⑥归并排序中，哪一种排序最快？哪一种排序辅助空间要求最少？哪几种排序是不稳定的？

快速排序。

堆排序

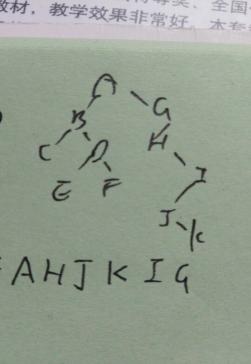
不稳定：2、3、4、5

稳定：1、6

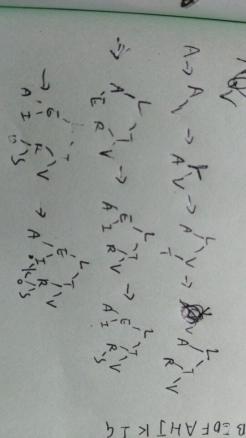
2．一棵二叉树的先序、中序序列如下，请构造出该二叉树。

先序序列：A B C D E F G H I J K

中序序列：C B E D F A H J K I G



3．设有一个关键字的输入序列{A，V，L，T，R，E，I，S，O，K}，

（1）从空树开始构造平衡二叉树，画出每加入一个新结点时二叉树的形态

1. 计算该平衡二叉树在等概率下查找成功的平均查找长度。

（1+2\*2+3\*4+4\*3）/10=29/10

4．假设有一副52张洗过的纸牌，共有4种花色，每种花色有13张牌。现在希望对这幅牌进行排序，使得4种花色是有序的，并且每种花色中的13张牌也是有序的。下列方法中哪一种最快？

（1）从头到尾查看这副牌，从中移出梅花，然后对它们单独排序。继续对方块、红心和黑桃试用同样的过程。（基数）

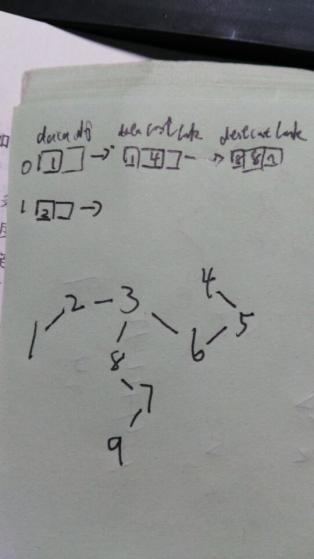
（2）按牌的大小将这些牌分成13堆，将这13堆再重新回到一起，然后按花色分成4堆，再将这4堆又堆回一起。（归并）

（3）通过将每张牌放在相对于前面已有序的牌的合适位置上，只对这些牌从头到尾做一趟排序。（直接插入）

5．对于如下图，给出它的实现（1）邻接表格（2）基于顺序表的邻接表；

分析从顶点1开始的深入优先遍历、广度优先遍历序列；

求解最小生成树；

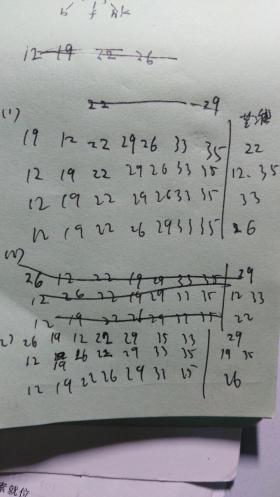


深入优先：1.2.3.4.5.6.7.9.8

广度优先：1.2.9.3.8.7.4.6.5

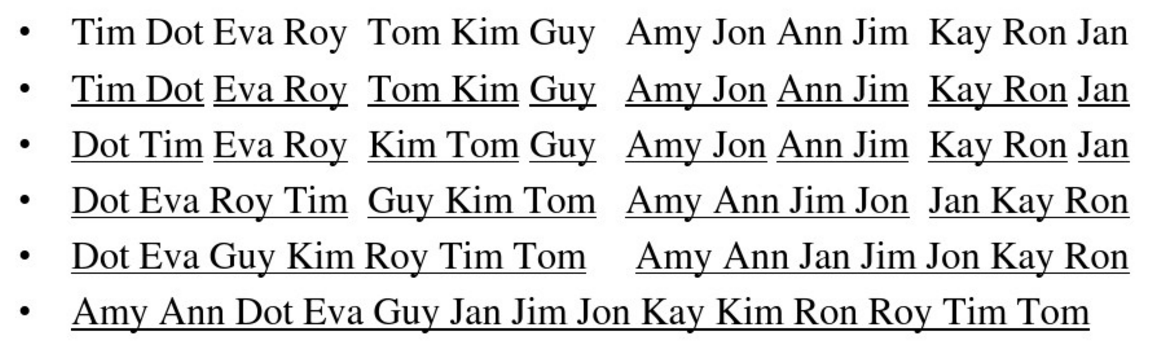
6．将下列元素以给定的次序插入到初始为的3阶B- 树中。

a g f b k d h m j e s I r x c l n t u p P482

7．将快速排序应用于表：26、33、35、29、19、12、22。每个子表的支点选择为：(1) 子表的最后一个数；(2) 子表的中央（或左中央）的数，在每种情况下，分析每趟排序过程。。

8．将归并排序应用于下表，给出排序过程。

　　　　　Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan



9．对下列每种应用，我们所学的哪种方法是较好的选择？为什么？如果表以顺序或链式存储表示，那么对你的选择有什么影响？说明如何影响。

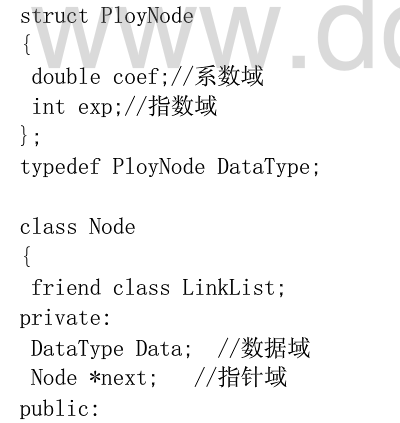
(a) 编写一个通用目的的排序程序，它将被许多人用于各种应用中。（快速排序）

(b) 一次性地对1000个数据排序，在完成这次排序后，不再保存这个程序。（归并排序）

(c) 一次性地对50个数据排序，在完成这次排序后，不再保存这个程序。（直接选择）

(d) 在一个较长的程序中对5个元素排序，而且此排序将被这个较长的程序用几百次。（堆排序，辅助空间少）

(e) 一个5000个单词的表已按字母顺序排列，现要对它进行检查以确信如此，并对所发现的任何失序的单词进行排序。（直接插入，已有序）

10．多元多项式的表示：讨论选择哪一种数据结构，表示m元多项式，例如三元多项式P(x,y,z)=x10y3z2+2x6y3z2+3x5y2z2+x4y4z+6x3y4z+2yz+15。

二．设计题

1．设计算法，实现表达式求值，例如对如下表达式求值：1+2\*(3+4)，只考虑+、-、\*、/、（）运算。

bool IsOperator(char ch)

{

char ops[] = "+-\*/";

for (int i = 0; i < sizeof(ops) / sizeof(char); i++)

{

if (ch == ops[i])

return true;

}

return false;

}

//////////////////////////////////////////////////////////////////////////// 比较两个操作符的优先级int Precedence(char op1, char op2)

{

if (op1 == '(')

{

return -1;

}

if (op1 == '+' || op1 == '-')

{

if (op2 == '\*' || op2 == '/')

{

return -1;

}

else

{

return 0;

}

}

if (op1 == '\*' || op1 == '/')

{

if (op2 == '+' || op2 == '-')

{

return 1;

}

else

{

return 0;

}

}

}

//////////////////////////////////////////////////////////////////////////// 中缀表达式转换成后缀表达式void inFix2PostFix(char\* inFix, char\* postFix)

{

int j = 0, len;

char c;

stack<char> st;

len = strlen(inFix);

for (int i = 0; i < len; i++)

{

c = inFix[i];

if (c == '(')

st.push(c);

else if (c == ')')

{

while (st.top() != '(')

{

postFix[j++] = st.top();

st.pop();

}

st.pop();

}

else

{

if (!IsOperator(c))

st.push(c);

else

{

while (st.empty() == false

&& Precedence(st.top(), c) >= 0)

{

postFix[j++] = st.top();

st.pop();

}

st.push(c);

}

}

}

while (st.empty() == false)

{

postFix[j++] = st.top();

st.pop();

}

postFix[j] = 0;

}

//////////////////////////////////////////////////////////////////////////// 后缀表达式求值程序double postFixEval(char\* postFix)

{

stack<char> st;

int len = strlen(postFix);

char c;

for (int i = 0; i < len; i++)

{

c = postFix[i];

if (IsOperator(c) == false)

{

st.push(c - '0');

}

else

{

char op1, op2;

int val;

op1 = st.top();

st.pop();

op2 = st.top();

st.pop();

switch (c)

{

case '+':

val = op1 + op2;

break;

case '-':

val = op2 - op1;

break;

case '\*':

val = op1 \* op2;

break;

case '/':

val = op2 / op1;

break;

}

st.push(val);

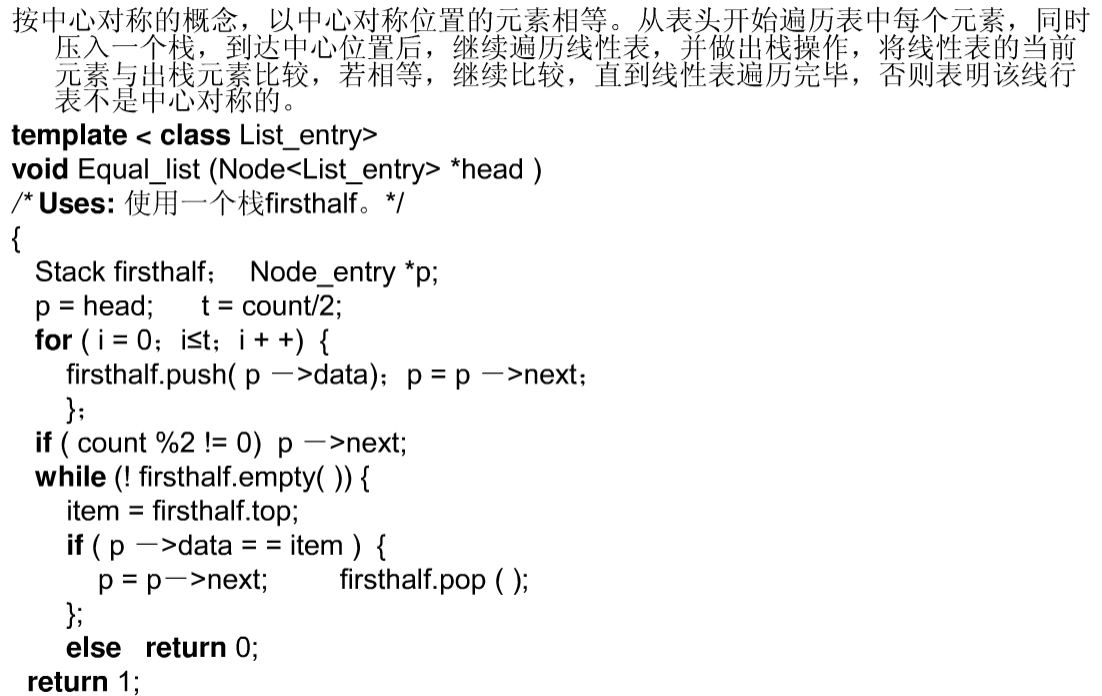
}

}

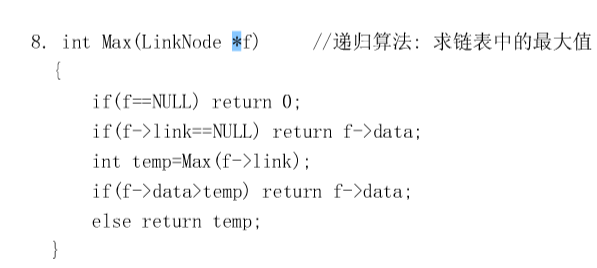
return st.top();

}

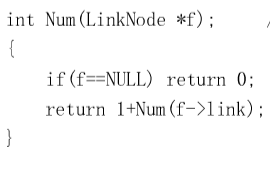
1. 编写一个判断单链表是否中心对称的算法。



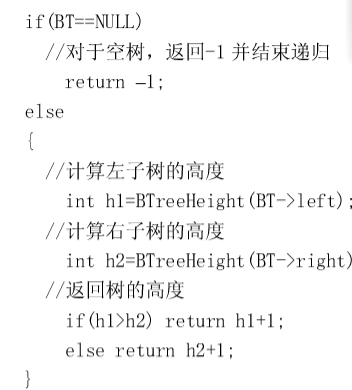
1. 已知p为单链表的表头指针，试写出求链表中最大值的递归算法。



1. 已知p为单链表的表头指针，试写出求链表长度的递归算法。



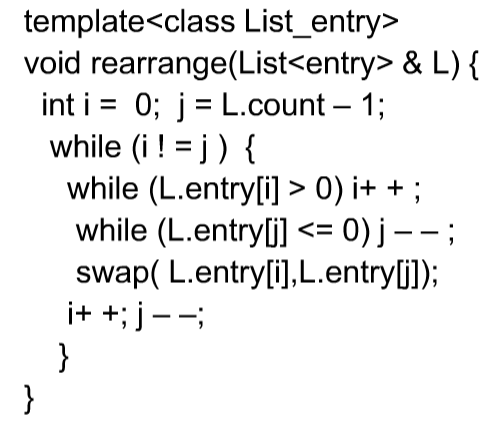
1. 试写一个递归算法，计算链式结构二叉树的深度，此处空树的深度为0，仅有一个结点的树深度为1。

 对于空树返回0

1. 试编写一个算法，将用二叉链表表示的完全二叉树转换成二叉树的顺序存储结构。

//root为当前树的根结点  
//array为存储树的数组  
//pos表示当前root所在array中的位置  
//起始调用时使用alertTheTree(root, array, 0);即可，默认array数组各元素值为非法值  
//标识当前位置无结点。MAX为数组array的最大长度  
void alertTheTree(TreeNode \*root, int \*array, int pos)  
{  
if (root == NULL || pos >= MAX)  
return;  
array[pos] = root->data;  
alertTheTree(root->left, array, 2 \* (pos + 1) -1);  
alertTheTree(root->right, array, 2 \* (pos + 1));  
} //这里用数组存放

1. 编写算法，使得在O(n)的时间内重排数组，将所有取正值的排序码排在所有取非正值的排序码之前。



8．**采用栈的方法，为下面的说明编写一个函数**copy\_stack :

// Error\_code copy\_stack(Stack &dest**, Stack &source);**

*precondition: 无.*

*postcondition: Stack dest 已经成为 Stack source 的一个完全副本; 而source 未*被改变。如果检测到错误，则返回一个合适的代码，否则返回 success 。 //

再设一个新栈做temp或让栈顶指针从0开始到栈顶

1. 有时一个程序需要两个包含同样数据类型的栈。如果两个栈存放在不同的数组里，那么当一个栈溢出时，另一个栈可能还有相当多的未用空间。避免这个问题的一个简洁的方法是把所有的空间都放在一个数组里，让一个栈从数组的一端增长，另一个栈从数组的另一端以相反方向朝第一个栈增长。以这种方式，如果一个栈大另一个栈小，则可以同时满足它们，除非所有的空间都被真正地使用了，否则，将不会产生溢出。定义一个新类Double\_stack，包含(作为私有数据成员)数组和两个符号top\_a 和top\_b，并为方法Double\_stack( )*,* push\_a( )*,* push\_b( )*,* pop\_a( )*, and* pop\_b( )编写代码，处理一个Double\_stack*.*内的两个栈。

bool Double\_stack::push\_a(T x){

If(top\_a+1==top\_b)return false;

Top\_a++;

Stack.Vector[Stack.[top\_a]]=x;

Return true;} b同理

bool Double\_stack::pop\_a(){

If(top\_a==栈底||top\_b==栈底)return false;

X=Stack.Vector[Stack.top\_a];

Top\_a--;return true;} b同理

10．双端队列

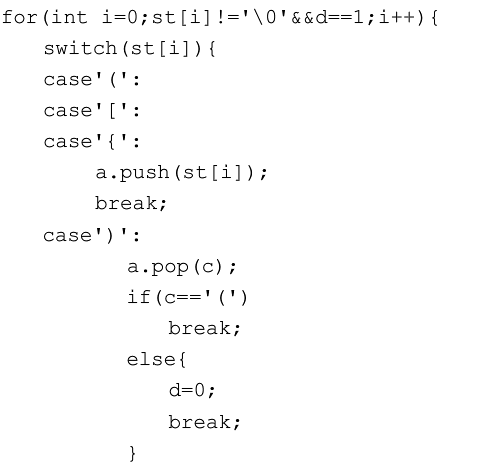
词***deque*** (读作“deck” 或 “DQ”) 是**double-ended queue**的缩写，表示一个表，其元素可以从队列的头部或尾部加入或删除，但是不改变表中的任何其他位置，因此，双端队列同时是栈和队列的广义化。 双端队列上的基本操作有：append\_front*,* append\_rear*,* serve\_front*,* serve\_rear*,* retrieve\_front*,* 和 retrieve\_rear。

（１）写出用线性数组实现双端队列的类定义和所需要的方法实现。P127

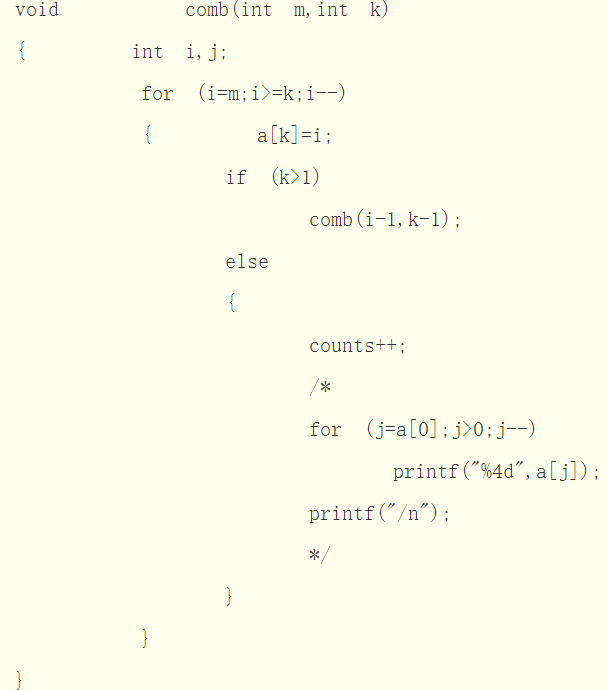
（２）假设编号为1, 2, 3, 4, 5, 6 的数据项以该次序出现在输入流中。 即第一是1，然后是2，依次类推。通过使用(1) a queue 和 (2) a deque, 能否得到下列重新排列的输出? 假设元素也以从左到右的顺序离开双端队列。

　　　　　　(a) 1 2 3 4 5 6 （1） (b) 2 4 3 6 5 1 (c) 1 5 2 4 3 6

1. 假设表达式中允许包含三种括号：圆括号，方括号和大括号。设计一个算法采用顺序栈判断表达式中的括号是否正确匹配。



1. 编写一个递归算法，找出从自然数1,2,3,4,…,n中任取r个数的所有组合。例如，n=5，r=3时的组合是543，542，541，532，531，521，432，431，421，321.



13. 已知递归函数,(1)写出求解F(n)的递归算法;(2)写出求解F(n)的非递归算法;



//递归：  
int f1(int n)  
{  
if(n==0)  
return 1;  
return n\*f1(n/2);  
}  
//非递归：  
int f2(int n)  
{  
int sum=1;  
while(n!=0)  
{  
sum\*=n;  
n/=2;  
}  
return sum;  
}