1，8大排序算法

1. 冒泡排序

基本思想：对于一个要排序的数组，首先遍历数组，从第一个数开始，每次将其与它右边的数作比较，如果该数大于它右边的数，则将其交换位置，然后再将第二个数与它右边的数比较，这样比较一趟，就会确定一个最大的数，经过n趟之后，所有的元素就都有序了。冒泡排序的时间复杂度O(n2),空间复杂度O(1),稳定，代码如下：

void bubbleSort1(vector<int> &A, int n)

{

// write code here

if(A.size() == 0 || A.size() != n)

return ;

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

{

for(int j = 0; j < n - i -1; j++)

{

if(A[j] > A[j + 1])

swap(A[j],A[j+1]);

}

}

}

(2)选择排序

基本思想：对于一个要排序的数组，定义一个下标变量，初始化为0，然后从第二个数开始遍历，如果其小于下标所对应的数，则将该数对应的下标赋值给下标变量，这样一趟遍历完成之后，就可以找到最小的数，然后与第一个数进行交换。然后再从第二个数在开始遍历，用前面的方法，当完成第n次遍历后，整个数组就变得有序了。选择排序的时间复杂度是O(n2),空间复杂度是O(1),不稳定。代码如下：

void selectionSort(vector<int> &A, int n)

{

// write code here

if(A.size() == 0 || A.size() != n)

return ;

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

{

int min = i;

for(int j = i + 1; j < n; j++)

{

if(A[min] > A[j])

min = j;

}

if(i != min)

swap(A[i],A[min]);

}

}

(3)插入排序：

基本思想：首先将第二个元素与第一个元素进行比较，如果小于第一个数，就插入到第一个数的前面，再从第3个数开始，与前面的数进行比较，一直向左找，如果遇到比当前数大的就向前插入，知道遇到比自己小的数，就退出当前循环。插入排序时间复杂度O(n2),空间复杂度O(1),稳定，代码如下：

void insertionSort(vector<int> &A, int n)

{

// write code here

if(A.size() == 0 || A.size() != n)

return ;

for(int i = 1; i < n; i++)

{

for(int j = i; j > 0; j--)

{

if(A[j] < A[j-1])

{

swap(A[j],A[j-1]);

}

else

break;

}

}

}

(4) 希尔排序

基本思想：希尔排序是对插入排序的改进，插入排序的步长为1，而希尔排序的初始步长是比1大，一般用你n/3+1；每次遍历一遍就会减少，最后都是一步长1结尾的。希尔排序的时间复杂度O(n2/3),空间复杂度O(1),不稳定。

void shellSort(vector<int> &A, int n)

{

// write code here

vector<int> increment;

for(int i = n/3 +1; i > 1; i = i/3 + 1)

increment.push\_back(i);

increment.push\_back(1);

for(int i = 0; i < increment.size(); i++)

{

for(int j = increment[i] ; j < n; j++) {

for(int k = j - increment[i]; k >= 0; k = k - increment[i]) {

if(A[k+increment[i]] < A[k])

swap(A[k+increment[i]],A[k]);

}

}

}

}

(5)快速排序

基本思想：快排是典型的分治法。首先找一个关键字(也叫枢轴)，然后将数组划分为两部分，左边的部分都比关键字小，右边的部分都比关键字大。然后递归的对左右两边的数进行前面的操作，直接数组中的所有元素有序。时间复杂度O(n2),空间复杂度O(logn)-O(n)，不稳定，代码如下:

//改进的快速排序算法,改进的方法为优化选择划分时的基准

int Partion(vector<int> &A, int low, int high)

{

int mid = low + (high - low)/2;

if(A[low] > A[high])

swap(A[low],A[high]);

if(A[mid] > A[high])

swap(A[mid],A[high]);

if(A[low] < A[mid])

swap(A[low],A[mid]);

int keyNum = A[low];

while(low < high)

{

while(low < high && keyNum <= A[high])

high--;

swap(A[low],A[high]);

while(low < high && keyNum >= A[low])

low++;

swap(A[low],A[high]);

}

return low;

}

void Qsort(vector<int> &A, int low, int high)

{

if(low >= high)

return;

int mid;

mid = Partion(A,low,high);

Qsort(A,low,mid-1);

Qsort(A,mid+1,high);

}

void quickSort(vector<int> &A, int n) {

// write code here

Qsort(A,0,n-1);

}

//非递归

void QuickSort1(vector<int> &a, int left,int right)

{

if (a.size() == 0 || left < 0 || right <= 0 || left>right)

return;

stack<int>temp;

int i, j;

temp.push(right);//先存左指针

temp.push(left);//再存右指针

while (!temp.empty())

{

i = temp.top();//先弹出左指针

temp.pop();

j = temp.top();//再弹出右指针

temp.pop();

if (i < j){

int k = partion(a, i, j);

if (k > i){

temp.push(k - 1);//保存中间变量

temp.push(i); //保存中间变量

}

if (j > k) {

temp.push(j);

temp.push(k + 1);

}

}

}

}

（6）堆排序

基本思想：先构建一个大根堆，然后每次将堆顶元素与数组最后一个元素进行交换，交换之后再进行对重建。由于建立大根堆的时间复杂度为O(n),重建堆的时间复杂度为O(logn),所以堆排序的总的时间复杂度为O(nlog(n)),空间复杂度O(1),不稳定。代码如下：

void Make\_heap(vector<int> &A,int i, int n) //n为数组的长度，i为完全二叉树的非终端节点

{

//int temp = A[i];

for(int j = 2\*i + 1; j <= n-1; j = j\*2 + 1)

{

if(j < n - 1 && A[j] < A[j + 1])

j++;

if(A[i] > A[j])

break;

swap(A[i],A[j]);

i = j;

}

}

void heapSort(vector<int> &A, int n) //n为数组的长度

{

for(int i = n/2 -1; i >= 0; i--) //将无序数组构成大顶堆

Make\_heap(A,i,n);

for(int i = n-1; i >= 0; i--) //依次数组最顶端的元素A[0]与A[i]进行交换，然后再重新构造大顶堆函数

{

swap(A[0],A[i]);

Make\_heap(A,0,i);

}

}

(7)归并排序

基本思想：先将数组从中间划分成两部分，然后将左右两部分递归再划分，当划分不了时，再将划分的两部分依次进行排序合并。时间复杂度O(nlogn),空间复杂度O(n),稳定。

//排序主函数

void mergeSort(vector<int> &A,int n)

{

// write code here

if(A.size() == 0 || A.size() != n)

return ;

int begin = 0;

int end = A.size() -1;

M\_sort(A,begin,end);

}

//子函数1

void M\_sort(vector<int> &A,int begin,int end)

{

if((end - begin) == 0)

return;

int mid = (begin + end)/2;

M\_sort(A, begin, mid);

M\_sort(A, mid+1, end);

merge(A,begin,mid,end);

}

//子函数2

void merge(vector<int> &A, int left, int mid, int right)

{

int i = left, j = mid+1, k = 0;

//int\* temp = new int[right - left + 1];

vector<int> temp(right - left + 1);

while (i <= mid && j <= right){

if (A[i] <= A[j]){

temp[k++] = A[i++];

}

else{

temp[k++] = A[j++];

}

}

while (i <= mid){

temp[k++] = A[i++];

}

while (j <= right){

temp[k++] = A[j++];

}

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

A[left+i] = temp[i];

}

//delete []temp;

}

数组：

(1)求连续子数组的最大和(剑指offer)

{6,-3,-2,7,-15,1,2,2},连续子向量的最大和为8(从第0个开始,到第3个为止)。

基本思路：定义两个变量，一个变量存储当前子数组的最大和，当小于0时就进行更新，另外一个变量存储出现的最大字数组的和，遍历一遍数组即可，时间复杂度O(n),空间复杂度O(1).代码如下：

int FindGreatestSumOfSubArray(vector<int> array) {

if(array.empty())

return 0;

int cSum = 0;

int result = array[0]; // result存储最大和，不能初始为0，存在负数

for(int i = 0;i<array.size();++i)

{

if(cSum < 0) // 当前和<0，抛弃不要

cSum = array[i];

else

cSum += array[i];

if(cSum > result) // 存储最大结果

result = cSum;

}

return result;

}

(2)求两个整数的最大公约数和最小公倍数

基本思路：辗转相除法，取两个数中较大的数做除数，较小的数做被除数，用较大的数除较小数，如果余数为0，则较小数为这两个数的最大公约数，如果余数不为0，用较小数除上一步计算出的余数，直到余数为0，则这两个数的最大公约数为上一步的余数。

求得最大公约数之后，用较大数和较小数的乘积再除以最大公约数就得到了最小公倍数。

Void func() {

int num1, num2,temp;

int r;

printf("请输入两个正整数：\n");

scanf("%d %d", &num1, &num2);

r = num1 % num2;

temp = num2;

while(r!=0) {

num1 = num2;

num2 = r;

r = num1 % num2;

}

printf("它们的最大公约数为：%d\n", num2);

printf("它们的最小公倍数为：%d\n", num1\*temp/num2);

}

(3)股票买进卖出的最大收益：

基本思路：动态规划方法，建立以为动态数组dp数组长度为数组的长度。定义变量最小价格minPrice，并对其进行不断更新。 动态规划方程：dp[i] = max(dp[i-1),prices[i]-minPrice]。

时间复杂度O(n),空间复杂度O(n)。代码如下：

//动态规划

int maxProfit(vector<int> &prices) {

if(prices.size() == 0)

return 0;

int minPrice = prices[0];

int len = prices.size();

vector<int> dp(len);

dp[0] = 0;

for(int i = 1; i < len; ++i){

dp[i] = max(dp[i-1], prices[i]-minPrice);

if(prices[i] < minPrice)

minPrice = prices[i];

}

return dp[len - 1];

}

变种1，不限制买入卖出次数：

int maxProfit(vector<int> &prices)

{

int minCurr = prices[0];

int maxProfit = 0;

for(int i = 1; i < prices.size(); ++i)

{

if(prices[i] - prices[i-1] > 0)

maxProfit += prices[i] - prices[i-1];

else

{

minCurr = prices[i];

}

}

return maxProfit;

}

变种2，有两次机会

基本思路：数组l[i]记录了price[0..i]的最大profit，

数组r[i]记录了price[i..n]的最大profit。

已知l[i]，求l[i+1]是简单的，同样已知r[i]，求r[i-1]也很容易。

最后，我们再用O(n)的时间找出最大的l[i]+r[i]，即为题目所求。

代码如下：

int maxProfit(vector<int> &prices) {

if(prices.size() < 2)

return 0;

int len = prices.size();

vector<int> preProfits(len);

vector<int> posProfits(len);

int minCurr = prices[0];

for(int i = 1; i < len; ++i){

preProfits[i] = max(preProfits[i-1],prices[i]-minCurr);

if(prices[i] < minCurr)

minCurr = prices[i];

}

int maxCurr = prices[len-1];

for(int i = len-2; i >= 0; --i){

posProfits[i] = max(posProfits[i+1],maxCurr-prices[i]);

if(prices[i] > maxCurr)

maxCurr = prices[i];

}

int maxProfit = 0;

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

maxProfit = max(preProfits[i] + posProfits[i],maxProfit);

}

return maxProfit;

}

(4)随机打乱顺序化初始数组的值。

vector<int> Trans(vector<int> &A, int n)

{

srand((unsigned)time(NULL));

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

{

int j = rand() % n;

swap(A[i], A[j]);

}

return A;

}

(5)数组中只出现一次的数字(剑指offer)

可以用位运算实现，如果将所有所有数字相异或，则最后的结果肯定是那两个只出现一次的数字异或的结果，所以根据异或的结果1所在的最低位，把数字分成两半，每一半里都还有只出现一次的数据和成对出现的数据 这样继续对每一半相异或则可以分别求出两个只出现一次的数字。代码如下：

bool IsBit(int num, int index)

{

num >>= index;

return num&1;

}

void FindNumsAppearOnce1(vector<int> data,int\* num1,int \*num2)

{

int partion = 0;

vector<int> vec1;

vector<int> vec2;

for(size\_t i = 0; i < data.size(); ++i)

{

partion ^= data[i];

}

int index = 0;

if(partion == 0)

return;

while((partion&1) == 0)

{

partion >>= 1;

++index;

}

for(size\_t i = 0; i < data.size(); ++i)

{

if(IsBit(data[i],index))

vec1.push\_back(data[i]);

else

vec2.push\_back(data[i]);

}

\*num1 = 0,\*num2 = 0;

for(size\_t i = 0; i < vec1.size(); ++i)

{

\*num1 ^= vec1[i];

}

for(size\_t i = 0; i < vec2.size(); ++i)

{

\*num2 ^= vec2[i];

}

}

(6)旋转数组的最小值

基本思路：二分法查找，注意有一种特殊情况：如数组为2，2，2，2，1，2时。

代码如下：

int minNumberInRotateArray(vector<int> rotateArray)

{

if(rotateArray.size() == 0)

return 0;

int begin,mid,end;

begin = 0;

bool flag = true; //标志位，

end = rotateArray.size()-1;

//二分法查找

while((end - begin) != 1)

{

mid = (begin+end)/2;

if(rotateArray[mid]==rotateArray[begin]&&rotateArray[mid]== rotateArray[end])

{

flag = false; //当出现if语句中存在的情况时，不能再继续用二分法，比 如数组｛2，2，2，2，1，2｝，只能用一般的顺序查找

break;

}

else if(rotateArray[mid] >= rotateArray[begin])

begin = mid;

else if(rotateArray[mid] <= rotateArray[end])

end = mid;

}

if(flag)

return rotateArray[end];

//顺序查找

else

{

int Min = rotateArray[0];

for(int i = 1; i < rotateArray.size(); i++)

{

if(Min > rotateArray[i])

Min = rotateArray[i];

}

return Min;

}

}

(7) 斐波那契数列

int Fibonacci(int n)

{

if(n == 0)

return 0;

if(n == 1)

return 1;

int preNum = 0,preNum2 = 1;

int currNum = 0;

for(int i = 2; i <= n; ++i)

{

currNum = preNum + preNum2;

preNum = preNum2;

preNum2 = currNum;

}

return currNum;

}

(8)求整数二进制中1的个数(剑指offer)

int NumberOf1(int n) {

int count = 0;

while(n != 0)

{

n = n&(n-1);

++count;

}

return count;

}

(9)数组中出现次数超过一半的数。(剑指offer)

基本思路：数组中有一个数字出现的次数超过了数组长度的一半，则说明这个数出现的次数比其它所有数字加起来出现的次数还要多。遍历数组，定义两个变量，一个变量为数组第一个数，另外一个变量为出现次数，初始化为1，从第二个数开始遍历数组，如果该数与变量相等，次数加1，不等变量次数减1，如果次数小于0了，则更新当前变量。当遍历到最后时，如果次数大于0，则说明存在出现次数大于数组长度一半的数，该数为第一个变量所存的数。否则，不存在这样的数。

bool checkMoreThanHalf(vector<int> &numbers,int result){

int count = 0;

for(size\_t i = 0; i < numbers.size(); ++i){

if(numbers[i] == result)

++count;

}

if(count > numbers.size()/2)

return true;

return false;

}

int MoreThanHalfNum\_Solution(vector<int> numbers) {

if(numbers.size() == 0)

return 0;

int num = numbers[0];

int times = 1;

for(size\_t i = 1; i < numbers.size(); ++i){

if(numbers[i] == num)

++times;

else {

--times;

if(times < 0){

num = numbers[i];

times = 1;

}

}

}

if(checkMoreThanHalf(numbers,num))

return num;

return 0;

}

(10)数组中最小的k个数

基本思路：用快排的划分的方法，时间复杂度为O(n).代码如下：

int parition(vector<int> &input, int low, int high){

int pviot = input[low];

while(low < high) {

while(low < high && input[high] >= pviot)

--high;

swap(input[low],input[high]);

while(low < high && input[low] <= pviot)

++low;

swap(input[low],input[high]); }

return low;}

vector<int> GetLeastNumbers\_Solution(vector<int> input, int k) {

if(input.size() == 0 || input.size() < k || k <= 0)

return vector<int>();

vector<int> res(k);

int begin = 0, end = input.size()-1;

int index,mid;

index = parition(input,begin,end);

while(index != k-1){

if(index < k-1){

begin = index+1;

index = parition(input,begin,end);

}

if(index > k-1){

end = index-1;

index = parition(input,begin,end);}

}

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

res[i] = input[i];

return res;}

解法2：O(nlogk)的算法，特别适合海量数据的处理。

vector<int> GetLeastNumbers\_Solution(vector<int> input, int k)

{

vector<int> res;

int len = input.size();

if(input.size() == 0 || k <= 0 || len < k)

return res;

multiset<int,greater<int>> leastNumbers;//从ä¨®大ä¨®到Ì?小?排?序¨°

multiset<int,greater<int>>::iterator iterGreater;

for(size\_t i = 0; i < input.size(); ++i){

if(leastNumbers.size() < k)

leastNumbers.insert(input[i]);

else{

iterGreater = leastNumbers.begin();

if(input[i] < \*(leastNumbers.begin())){

leastNumbers.erase(iterGreater);

leastNumbers.insert(input[i]);

}

}

}

for(iterGreater=leastNumbers.begin();iterGreater != leastNumbers.end();++iterGreater)

res.push\_back(\*iterGreater);

return res;

}

(11) 判断编译器是大端模式还是小端模式：

对于大端模式，低地址存放的是高字节的数据，高地址是存放的高字节数据，小端模式刚好相反。代码如下：

#include <iostream>

using namespace std;

int main(){

int nNum = 0x12345678;

char chData = \*(char\*)(&nNum);

if (chData == 0x12){

cout << "big" << endl;

}

else{

cout << "small" << endl;

}

system("PAUSE");

return 0;

}

(12) 求1到n中1出现的次数：

基本思路：常规方法就是求1到n中每个数字中1出现的次数：时间复杂度为O(NlgN)

找数字出现的规律，可以将时间复杂度做到O(lgN\*lgN);找规律基本思路：先求最高位上1出现的次数，如果最高位上的数位1，则最高位上出现1的次数=除去最高位后的数+1；如果最高位不是1，则最高位上出现1的次数=10的k次方，k=n-1;低位的数任意固定一位后其他位出现的次数可以在0到9之间变化，所以地位数中出现1的次数= （最高为数字）\*（除去最高位后剩下的位数）\*（固定一位为1后剩余的位数）；举例；如114 ，最高位出现1的次数为114%100 +1= 15，个位和十位出现1的次数=1\*2\*101 。举例：21345 ，先看1346~21345最高位出现1的次数 = 104，其余位出现1的次数=2\*4\*103。然后再递归346~1345.最后知道为数为1位，则返回1.代码如下：

//常¡ê规?方¤?法¤¡§

int getNumOf1(int num){

int count = 0;

while(num) {

if(num % 10 == 1)

++count;

num = num/10;

}

return count;

}

int NumberOf1Between1AndN\_Solution(int n){

if(n < 1)

return 0;

int res = 0;

for(int i = 1; i <= n; ++i){

res += getNumOf1(i);

}

return res;

}

//找¨°规?律¨¦方¤?法¤¡§

int getLenOfNum(int n){

int count = 0;

while(n){

n /= 10;

++count;

}

return count;

}

int powerBaseOf10(int len){

return (int)pow(10.0,(double)(len));

}

int NumberOf1Between1AndN\_Solution1(int n){

if(n < 1)

return 0;

int len = getLenOfNum(n);

if(len == 1)

return 1;

int tmp = powerBaseOf10(len-1);

int first = n / tmp;

int firsrOneNumber = first == 1 ? n % tmp +1 : tmp;

int otherOneNumber = first \* (len - 1) \* (tmp / 10);

return firsrOneNumber+otherOneNumber+NumberOf1Between1AndN\_Solution1(n % tmp);

}

(13) 把数组排成最小的数

基本思路：借助排序的思想，但与排序思想交换思想不同，自定义交换的规则；代码如下：

string PrintMinNumber(vector<int> numbers) {

string result;

if(numbers.size()<=0){

return result;

}

vector<string> strNum;

for(int i=0;i<numbers.size();i++ ){

stringstream ss;

ss << numbers[i];

string s = ss.str();

strNum.push\_back(s);

}

sort(strNum.begin(),strNum.end(),compare);

for(int i=0;i<strNum.size();i++){

result.append(strNum[i]);

}

return result；

}

bool compare( const string &st1,const string &st2){

string s1 = st1+st2;

string s2 = st2+st1;

return s1<s2;

}

(14) 丑数

题意：把只包含因子2，3，5的数称为丑数。例如6，8都是丑数，但14不是，因为它包含因子7。求按顺序从小到大的第N个丑数。基本思路：三指针法。代码如下：

int GetUglyNumber\_Solution(int index) {

if(index <= 0)

return 0;

vector<int> res(index);

res[0] = 1;

int t1 = 0,t2 = 0,t3 = 0;

int tmp1,tmp2,tmp3;

for(int i = 1; i < index; ++i){

tmp1 = res[t1]\*2;

tmp2 = res[t2]\*3;

tmp3 = res[t3]\*5;

if(tmp1 < tmp2 && tmp1 < tmp3){

++t1;

res[i] = tmp1;

}

else if(tmp2 < tmp1 && tmp2 < tmp3){

++t2;

res[i] = tmp2;

}

else if(tmp3 < tmp1 && tmp3 < tmp2){

++t3;

res[i] = tmp3;

}

else if(tmp1 == tmp2 && tmp1 < tmp3){

++t1;

++t2;

res[i] = tmp1;

}

else if(tmp1 == tmp3 && tmp1 < tmp2){

++t1;

++t3;

res[i] = tmp1;

}

else if(tmp2 == tmp3 && tmp2 < tmp1){

++t2;

++t3;

res[i] = tmp2;

}

else{

++t1;

++t2;

++t3;

res[i] = tmp1;

}

}

return res[index-1];

}

(15) 数组中的逆序对

基本思路：先把子数组分隔成子数组，先统计出子数组内部的逆序对的数目，然后再统计出两个相邻子数组之间的逆序对的数目。在统计逆序对的过程中，还需要对数组进行排序。

这个题目实际上就是归并排序的一个应用。时间复杂度O(nlgn),空间复杂度O(n).如果用常规方法，时间复杂度为O(n2),空间复杂度为O(1)。代码如下：

void merge(vector<int> &data, int begin, int mid, int end,int& count){

vector<int> tmp(end-begin+1);

int i,j;

i = mid,j = end;

while(i >= begin && j >= mid+1){

if(data[i] > data[j]){

count += j-mid;

count %= 1000000007;

--i;

}

else if(data[i] < data[j]){

--j;

}

}

int index = 0;

i = begin,j = mid+1;

while(i <= mid && j <= end){

if(data[i] <= data[j]){

tmp[index++] = data[i];

++i;

}

else{

tmp[index++] = data[j];

++j;

}

}

while(j <= end){

tmp[index++] = data[j++];

}

while(i <= mid){

tmp[index++] = data[i++];

}

for(i = 0; i < end-begin+1; ++i){

data[begin+i] = tmp[i];

}

}

void InversePairsResolve(vector<int> &data, int begin, int end,int& count){

if(end == begin)

return;

int res = 0;

int mid = (end - begin) / 2 + begin;

InversePairsResolve(data, begin, mid,count);

InversePairsResolve(data, mid+1, end,count);

merge(data, begin, mid, end,count);

}

int InversePairs(vector<int> data) {

if(data.size() <= 1)

return 0;

int count = 0;

InversePairsResolve(data,0,data.size()-1, count);

return count;

}

(16)数值的整数次方：(常规方法O(n),改进后的方法O(logn)),注意考虑代码的鲁棒性。

bool g\_inputValid = false;

//逐个连乘

double func\_power1(double base,int exponet){

double res = 1.0;

for(int i = 0; i < exponet; i++){

res \*= base;

}

return res;

}

//改进方法

double func\_power(double base,int exponet){

double res = base;

for(int i = exponet; i > 1; i >>= 1){

res \*= res;

}

if((exponet % 2) == 1)

res \*= base;

return res;

}

bool isDouble0(double base){

if(base < 0.0000001 && base > -0.0000001){

g\_inputValid = true;

return true;

}

return false;

}

//给定一个double类型的浮点数base和int类型的整数exponent。求base的exponent次方。

double Power(double base, int exponent) {

double res = 0.0;

if(exponent > 0)

res = func\_power(base,exponent);

if(!isDouble0(base) && exponent < 0){

res = 1.0 / func\_power(base,-exponent);

}

if(exponent == 0){

if(!isDouble0(base))

res = 1;

else

res = 0;

}

return res;

}

(17) 生成单例模式的类

#include <iostream>

class Singleton{

public:

static Singleton\* getInstance();

private:

Singleton();

Singleton(const Singleton&);

Singleton& operator=(const Singleton&);

static Singleton\* instance;

};

Singleton::Singleton(){

}

Singleton\* Singleton::instance = new Singleton();

Singleton\* Singleton::getInstance(){

return instance;

}

int main(){

Singleton\* singleton1 = Singleton::getInstance();

Singleton\* singleton2 = Singleton::getInstance();

if (singleton1 == singleton2)

fprintf(stderr,"singleton1 = singleton2\n");

system("pause");

return 0;

}

三，链表

1. 反转链表(剑指offer)

直接上代码

ListNode\* ReverseList(ListNode\* pHead) {

if(pHead == NULL || pHead->next == NULL)

return pHead;

ListNode\* pPre = NULL;

ListNode\* pCur = pHead;

ListNode\* pNex = pHead->next;

while(pNex != NULL) {

pCur->next = pPre;

pPre = pCur;

pCur = pNex;

pNex = pNex->next;

}

pCur->next = pPre;

return pCur; }

1. 求链表的倒数第k个结点(剑指offer)

定义快慢指针，开始时都指向头结点，然后快指针先向前走k-1步，如果快指针中途为空，则直接返回空。然后快慢指针同时向后移动，当快指针指向最后一个节点时，慢指针指向的就是倒数第k个结点：代码如下：

ListNode\* FindKthToTail(ListNode\* pListHead, unsigned int k) {

if(pListHead == NULL)

return NULL;

ListNode\* pQuick = pListHead;

ListNode\* pSlow = pListHead;

for(unsigned i = 0; i < k-1; ++i){}

pQuick = pQuick->next;

if(pQuick == NULL)

return NULL;

}

while(pQuick->next != NULL) {

pSlow = pSlow->next;

pQuick = pQuick->next;

}

return pSlow;

}

3.合并两个排序链表(剑指offer)

基本思想：递归和非递归两种方法；

方法一：

ListNode\* Merge(ListNode\* pHead1, ListNode\* pHead2) {

if(NULL == pHead1)

return pHead2;

if(NULL == pHead2)

return pHead1;

ListNode\* pNewHead = NULL;

if(pHead1->val < pHead2->val) {

pNewHead = pHead1;

pNewHead->next = Merge(pHead1->next,pHead2);

}

else{

pNewHead = pHead2;

pNewHead->next = Merge(pHead1,pHead2->next);

}

return pNewHead； }

方法二：

//改变了原来的链表的结构,非递归

ListNode\* Merge1(ListNode\* pHead1, ListNode\* pHead2)

{

if(pHead1 == NULL)

return pHead2;

if(pHead2 == NULL)

return pHead1;

ListNode\* newList = NULL;

ListNode\* pCurrent;

while(pHead1 != NULL && pHead2 != NULL) {

if(pHead1->val < pHead2->val) {

if(newList == NULL)

newList = pCurrent = pHead1;

else {

pCurrent->next = pHead1;

pCurrent = pCurrent->next;

}

pHead1 = pHead1->next;

}

else {

if(newList == NULL)

newList = pCurrent = pHead1;

else {

pCurrent->next = pHead2;

pCurrent = pCurrent->next;

}

pHead2 = pHead2->next;

}

}

if(pHead1 == NULL) {

pCurrent->next = pHead2;

}

else{

pCurrent->next = pHead1;

}

return newList;

}

4,求链表的环入口结点

基本思路：定义快慢指针，初始都指向头结点，快指针每次走两步，慢指针每次走一步，当快慢指针第一次相遇时，肯定在环内，然后将慢指针重新指向头结点，快慢指针每次只移动一步，当快慢指针再次相遇时，相遇的结点就是环入口结点。

ListNode\* EntryNodeOfLoop(ListNode\* pHead)

{

if(pHead == NULL || pHead->next == NULL)

return NULL;

ListNode\* pQuick = pHead;

ListNode\* pSlow = pHead;

pQuick = pQuick->next->next;

pSlow = pSlow->next;

while(pQuick != pSlow) {

pQuick = pQuick->next->next;

pSlow = pSlow->next； }

pSlow = pHead;

while(pSlow != pQuick) {

pQuick = pQuick->next;

pSlow = pSlow->next;

}

return pSlow;

}

1. 两个链表的第一个公共结点

基本思路：两个链表如果有公共结点，一定会呈Y字形，也就是说从两个链表的第一个公共结点起，后面的结点应该都对一样的。先分别遍历两个链表的长度，求得两个连表的长度只差differ，长度更长的链表先走differ个节点，然后两个链表一起走，如果遇到相同的结点，则说明该节点是两链表的公共结点。时间复杂度O(n);

ListNode\* FindFirstCommonNode( ListNode\* pHead1, ListNode\* pHead2) {

if(pHead1 == NULL || pHead2 == NULL)

return NULL;

unsigned len1 = 0;

unsigned len2 = 0;

ListNode\* p1 = pHead1;

ListNode\* p2 = pHead2;

while(p1){

p1 = p1->next;

++len1 }

while(p2){

p2 = p2->next;

++len2; }

int differ = 0;

p1 = pHead1,p2 = pHead2;

if(len1 < len2){

differ = len2-len1;

while(differ--)

p2 = p2->next;

}

else{

differ = len1-len2;

while(differ--)

p1 = p1->next;

}

while(p1 != p2){

p1 = p1->next;

p2 = p2->next;

}

if(p1)

return p1;

return NULL; }

6,删除链表中重复的结点：在一个排序的链表中，存在重复的结点，请删除该链表中重复的结点，重复的结点不保留，返回链表头指针。例如，链表1->2->3->3->4->4->5 处理后为 1->2->5。

基本思路：不涉及复杂的算法，单纯的考察编程实现能力

ListNode\* deleteDuplication1(ListNode\* pHead)

{

if(pHead == NULL || pHead->next == NULL)

return pHead;

else {

ListNode\* newHead = new ListNode(-1);

newHead->next = pHead;

ListNode\* pre = newHead;

ListNode\* p = pHead;

ListNode\* next = NULL;

while( p != NULL && p->next != NULL){

next = p->next;

if(p->val == next->val){

int val = p->val;

delete p;

while(next != NULL && next->val == val){

ListNode\* tmp = next->next;

delete next;

next = tmp;

}

pre->next = next;

p = next;

}

else{

pre = p;

p = p->next;

}

}

return newHead->next; }

7复杂链表的复制(剑指offer)

基本思路：第一步，先复制每个节点连在该节点后面，遍历链表，复制每个节点的random指针，第三步，分解链表；

RandomListNode\* Clone(RandomListNode\* pHead)

{

if (pHead == NULL){

return NULL;

}

RandomListNode\* pc = pHead;

RandomListNode\* pM = NULL;

while (pc != NULL){

pM = new RandomListNode(pc->label);

pM->next = pc->next;

pc->next = pM;

pc = pM->next;

}

pc = pHead;

while (pc != NULL){

pM = pc->next;

if (pc->random != NULL){

pM->random = pc->random->next;

}

pc = pc->next->next;

}

RandomListNode \*pHead1 = pHead;

RandomListNode \*pHead2 = pHead->next;

RandomListNode \*p1 = pHead1;

RandomListNode \*p2 = pHead2;

while ( p2->next != NULL){

p1->next = p1->next->next;

p2->next = p2->next->next;

p1 = p1->next;

p2 = p2->next;

}

p1->next = NULL;

p2->next = NULL;

return pHead2;

}

8 二叉搜索树转换成排序链表

基本思路，先中序遍历把链表的每个节点都push到一个vector里，然后再进行两次循环

代码：

//非递归

vector<TreeNode\*> nodes；

void tranverse(TreeNode\* pRoot) {

     if (nullptr == pRoot)

         return;

      tranverse(pRoot->left);

      nodes.push\_back(pRoot);

      tranverse(pRoot->right);

    }

TreeNode\* adjustTree() {

  for (int i = 0; i < nodes.size() - 1; ++i)

      nodes[i]->right = nodes[i+1];

      nodes[nodes.size()-1]->right = nullptr;

   for (int i = nodes.size() - 1; i > 0; --i)

      nodes[i]->left = nodes[i-1];

      nodes[0]->left = nullptr;

      return nodes[0];

    }

  TreeNode\* Convert(TreeNode\* pRoot)

   {

     if (nullptr == pRoot)

      return nullptr;

      tranverse(pRoot);

      return adjustTree();

    }

//递归

void convertHelper(TreeNode\* cur,TreeNode\* &pre)

{

if(cur == NULL)

return;

convertHelper(cur->left,pre);

cur->left = pre;

if(pre)

pre->right = cur;

pre = cur;

convertHelper(cur->right,pre);

}

TreeNode\* Convert(TreeNode\* pRootofTree)

{

if(pRootofTree == NULL)

return NULL;

TreeNode\* pre = NULL;

convertHelper(pRootofTree,pre);

TreeNode\* res = pRootofTree;

while(res->left)

res = res->left;

return res;

}

三，二叉树

(1)重建二叉树：给定两个数组，一个是二叉树前序遍历的数组，一个是二叉树中序遍历的数组，根据这两个数组重建出二叉树。直接上代码：

TreeNode\* reConstructBinaryTree1(vector<int> pre,vector<int> vin) {

if(pre.size() == 0 || vin.size() == 0 || pre.size() != vin.size())

return NULL;

int val = pre[0];

TreeNode\* root = new TreeNode(val);

vector<int> pre\_left,pre\_right,vin\_left,vin\_right;

bool flag = false;

for(size\_t i = 0; i < vin.size(); ++i){

if(!flag && val != vin[i])

vin\_left.push\_back(vin[i]);

else if(val == vin[i])

flag = true;

else

vin\_right.push\_back(vin[i]);

}

int count = vin\_left.size();

for(size\_t i = 1; i < pre.size(); ++i){

if(count > 0){

pre\_left.push\_back(pre[i]);

--count;

}

else

pre\_right.push\_back(pre[i]);

}

root->left = reConstructBinaryTree1(pre\_left,vin\_left);

root->right = reConstructBinaryTree1(pre\_right,vin\_right);

return root;

}

1. 树的子结构：判断一棵树是否是另一棵树的子结构：

代码如下：

bool isTheSame(TreeNode\* root, TreeNode\* pRoot2){

if(pRoot2 == NULL)

return true;

if(pRoot2 != NULL && root == NULL)

return false;

if(root->val != pRoot2->val)

return false；

return isTheSame(root->left, pRoot2->left) && isTheSame(root->right, pRoot2->right);

}

bool TraverseTree(TreeNode\* pRoot1, TreeNode\* pRoot2){

if(pRoot1 == NULL)

return false;

if(pRoot1->val == pRoot2->val){

if(isTheSame(pRoot1,pRoot2))

return true;

}

return TraverseTree(pRoot1->left,pRoot2)||

TraverseTree(pRoot1->right,pRoot2);

}

bool HasSubtree(TreeNode\* pRoot1, TreeNode\* pRoot2){

if(pRoot1 == NULL || pRoot2 == NULL)

return false;

return TraverseTree(pRoot1,pRoot2);

}

1. 二叉树的镜像(剑指offer)

基本思路：先序遍历二叉树，再遍历的过程中如果该节点不是叶节点，则交换两个子节点的位置，当遍历完成二叉树之后，则二叉树就变成原来的镜像。

void Mirror(TreeNode \*pRoot) {

if(pRoot == NULL)

return;

if(pRoot->left == NULL && pRoot->right == NULL)

return;

TreeNode\* tmp = pRoot->left;

pRoot->left = pRoot->right;

pRoot->right = tmp;

Mirror(pRoot->left);

Mirror(pRoot->right);

}

1. 按层打印二叉树(剑指offer)

基本思路：借助一个队列。广度遍历思想。代码如下：

vector<int> PrintFromTopToBottom(TreeNode\* root) {

vector<int> res;

if(root == NULL)

return res;

deque<TreeNode\*> deq;

TreeNode\* tmp = NULL;

deq.push\_back(root);

while(!deq.empty()){

int len = deq.size();

while(len--){

tmp = deq.front();

res.push\_back(tmp->val);

deq.pop\_front();

if(tmp->left)

deq.push\_back(tmp->left);

if(tmp->right)

deq.push\_back(tmp->right);

}

}

return res;

}

1. 二叉搜索树的后序遍历序列

输入一个整数数组，判断这个是不是某二叉搜索树的后序遍历序列。

基本思路：与重建二叉树的思路基本一致，重建二叉树是给出了两种方式遍历的数组，而本题给了后序遍历的数组，但是还有信息就是该二叉树是二叉搜索树，根据这一信息，每次也可以将数组划分为左子树部分和右子树部分。代码如下：

bool funcPostTreeTraverse(vector<int> sequence) {

if(sequence.size() == 0)

return true;

vector<int> left;

vector<int> right;

int len = sequence.size();

int root = sequence[len-1];

int i;

for(i = 0; i < len-1; ++i){

if(sequence[i] < root)

left.push\_back(sequence[i]);

else

break;

}

for(int j = i; j < len-1; ++j){

if(sequence[j] > root)

right.push\_back(sequence[j]);

else

return false;

}

return funcPostTreeTraverse(left) && funcPostTreeTraverse(right);

}

bool VerifySquenceOfBST(vector<int> sequence) {

if(sequence.size() == 0)

return false;

return funcPostTreeTraverse(sequence);

}

1. 二叉树中和为某一值的路径

基本思路：根据二叉树的先序遍历，遍历的过程中记录下从根节点累计到该节点的值，当遍历到叶子节点时，与给定的数进行比较，如果相等则保存。当遍历完所有节点之后，所有的二叉树中和为某一值的路径就全部被记录下来。代码如下：

void FuncFindPath(TreeNode\* root,vector<vector<int> > &res, vector<int> &tmp, int num, int expectNumber){

if(root == NULL)

{

return;

}

num += root->val;

tmp.push\_back(root->val);

if(root->left == NULL && root->right == NULL){

if(num == expectNumber){

res.push\_back(tmp);

num -= tmp[tmp.size()-1];

tmp.pop\_back();

return;

}

}

FuncFindPath(root->left,res,tmp,num,expectNumber);

FuncFindPath(root->right,res,tmp,num,expectNumber);

num -= tmp[tmp.size()-1];

tmp.pop\_back();

}

vector<vector<int> > FindPath(TreeNode\* root,int expectNumber) {

vector<vector<int> > res;

vector<int> tmp;

int num = 0;

if(root == NULL)

return res;

FuncFindPath(root,res,tmp,num,expectNumber);

return res;

}

1. 二叉树的深度：代码如下

int TreeDepth(TreeNode\* pRoot)

{

if(pRoot == NULL)

return depth；

return max(TreeDepth(pRoot->left)+1,TreeDepth(pRoot->right)+1);

}

//非递归方法

int TreeDepth(TreeNode\* pRoot){

queue<TreeNode\*> q;

if(!pRoot) return 0;

q.push(pRoot);

int level=0;

while(!q.empty()){

int len=q.size();

level++;

while(len--){

TreeNode\* tem=q.front();

q.pop();

if(tem->left) q.push(tem->left);

if(tem->right) q.push(tem->right);

}

}

return level;

}

1. 求二叉树的下一个节点：

题目：给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。

代码如下：

TreeLinkNode\* GetNext(TreeLinkNode\* pNode){

if(pNode==NULL)

return NULL;

if(pNode->right!=NULL){

pNode=pNode->right;

while(pNode->left!=NULL)

pNode=pNode->left;

return pNode;

}

while(pNode->next!=NULL){

TreeLinkNode \*proot=pNode->next;

if(proot->left==pNode)

return proot;

pNode=pNode->next;

}

return NULL;

}

(9)对称的二叉树：判断一个二叉树是否和她的镜像相同。

bool isSymmetrical(TreeNode\* pRoot){

if(pRoot == NULL)

return true;

deque<TreeNode \*> deq;

vector<char> vec;

deq.push\_back(pRoot);

bool isSymmetric = false;

while(!deq.empty())

{

int len = deq.size();

while(len--){

TreeNode \* tmp = deq.front();

deq.pop\_front();

if(tmp) {

vec.push\_back(tmp->val);

deq.push\_back(tmp->left);

deq.push\_back(tmp->right);

}

else

vec.push\_back('#');

}

//判断是否对称

int begin = 0;

int end = vec.size()-1;

if(begin == end)

isSymmetric = true;

else {

while(begin < end) {

if(vec[begin] == vec[end]) {

++begin;

--end;

}

else {

isSymmetric = false;

return isSymmetric;

}

}

isSymmetric = true;

}

vec.clear();

}

return isSymmetric;

}

1. 序列化二叉树：请实现两个函数，用来序列化和反序列化函数：

string toString(TreeNode\* root) {

// write code here

string res;

if(root == NULL){

res += "#!";

return res;

}

int val = root->val;

string temp;

stringstream s;

s << val;

s >> temp;

s.str("");

res += temp;

res += "!";

res += toString(root->left);

res += toString(root->right);

return res;

}

char\* Serialize(TreeNode \*root) {

string res = toString(root);

char\* chr = new char(res.size()+1);

strcpy(chr,res.c\_str());

return chr;

}

int p = 0;

void func(TreeNode \*\*root,char\* str){

if( str[p] == '\0')

return;

if(str[p] == '#'){

p = p+2;

return;

}

if(\*root == NULL){

\*root = new TreeNode(str[p]);

p = p+2;

}

func(&((\*root)->left),str);

func(&((\*root)->right),str);

}

TreeNode\* Deserialize(char \*str) {

if(str == NULL)

return NULL;

int len = strlen(str);

TreeNode\* root = NULL;

func(&root,str);

return root;

}

(10)搜索树的第k个结点

题意：给定一颗二叉搜索树，找出其中第k大的节点

代码如下：

int count1 = 0;

TreeNode\* KthNode(TreeNode\* pRoot, int k){

if(pRoot == NULL)

return NULL;

TreeNode\* node = KthNode(pRoot->left,k);

if(node != NULL)

return node;

if(++count1 == k)

return pRoot;

node = KthNode(pRoot->right,k);

if(node != NULL)

return node;

return NULL;

}

(11)树中两个节点的最低公共祖先。

基本思路：从根节点开始遍历二叉树，如果根结点大于两个节点，则遍历其左子节点，如果根节点小于两个节点，则遍历右子节点，否则，该节点就是我们要找的最低公共祖先。代码如下：

TreeNode\* LowestCommonAncester(TreeNode\* root, int val1,int val2){

if(NULL == root)

return NULL;

TreeNode\* res = NULL;

if(root->val > val1 && root->val > val2){

res = LowestCommonAncester(root->left,val1,val2);

}

else if(root->val < val1 && root->val < val2){

res = LowestCommonAncester(root->right,val1,val2);

}

else

return root;

}

(12)判断一颗二叉树是否是平衡二叉树：

代码如下：

int getHeight(TreeNode\* pRoot,int level, bool& flag){

if(pRoot == NULL)

return level;

int left = getHeight(pRoot->left,level+1,flag);

if(!flag)

return left;

int right = getHeight(pRoot->right,level+1,flag);

if(!flag)

return right;

if(abs(left-right) > 1)

flag = false;

return max(left,right);

}

bool IsBalanced\_Solution(TreeNode\* pRoot) {

if(NULL == pRoot){

return true;

}

bool flag = true;

getHeight(pRoot,0,flag);

return flag;

}

四：字符串：

1. Strcpy函数的编写：考虑内存覆盖和不考虑内存覆盖

char\* strcpy1(char\* strDest, const char\* strSrc)

{

assert(strDest != NULL && strSrc != NULL);

char\* strRes = strDest;

while((\*strDest++ = \*strSrc++) != '\0');

return strRes;

}

void my\_strCopy(char\* strDest, const char\* strSrc, int len)

{

if(strDest >= strSrc && strDest <= strSrc + len - 1)

{

strDest = strDest + len -1;

strSrc = strSrc + len -1;

while(--len)

{

\*strDest-- = \*strSrc--;

}

}

while((\*strDest++ = \*strSrc++) != '\0');

}

char\* strcpy(char\* strDest, const char\* strSrc)

{

assert(strDest != NULL && strSrc != NULL);

char\* strRes = strDest;

my\_strCopy(strDest,strSrc,strlen(strSrc)+1);

return strRes;

}

1. String类的编写

#include<iostream>

class String

{

public:

String(const char \*str = NULL);

String(const String &other);

~ String(void);

String & operator =(const String &other);

void print();

private:

char \*m\_data;

};

String::String(const char \*str )

{

if(str == NULL)

{

m\_data = new char[1];

\*m\_data = '\0';

}

else

{

m\_data = new char[strlen(str)+1];

if(m\_data != NULL)

strcpy(m\_data,str);

}

std::cout << "String()" << std::endl;

}

String::String(const String &other)

{

m\_data = new char[strlen(other.m\_data)+1];

if(m\_data != NULL)

strcpy(m\_data,other.m\_data);

std::cout << "拷?贝À¡äString()" << std::endl;

}

String & String::operator =(const String &other)

{

if(this == &other)

return \*this;

delete []m\_data;

m\_data = new char[strlen(other.m\_data)+1];

if(m\_data != NULL)

strcpy(m\_data,other.m\_data);

std::cout << "赋3值¦ÌString()" << std::endl;

return \*this;

}

void String::print()

{

std::cout << m\_data << std::endl;

}

String::~ String(void)

{

delete []m\_data;

m\_data = NULL;

}

五、矩阵的三种打印方式：