算法分析之demo

cout 格式化输出

```
#include<iomanip>——定义IO流输出输入格式控制相关函数

cout<<setw(n)    设置字段宽度为n位
cout<<setfill(c)     设置填充字符c
cout<<setfill(c)<<setw(n) 多于的位用c填充

cout<<fixed      设置浮点数以固定的小数位数显示
cout<<scientific     设置浮点数以科学记数法(即指数形式)显示
cout<<setprecision(n)   设置浮点数的精度为n位。在以一般十进制小数形式输出时，n代表有效数字
cout<<fixed<<setprecision(n)   fixed(固定小数位数)形式输出时，n为小数位数
cout<<scientific<<setprecision(n)    scientific(指数)形式输出时，n为小数位数

cout<<left   输出数据左对齐
cout<<right 输出数据右对齐
cout<<left<<setw(n)
cout<<right<<setw(n)
cout<<skipws     忽略前导的空格
cout<<uppercase 数据以十六进制形式输出时字母以大写表示
cout<<lowercase 数据以十六进制形式输出时字母以小写表示
cout<<showpos    输出正数时给出“+”号
```

链表

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

struct ListNode{
    int val;
    struct ListNode *next;
};

template <class T>
struct ListNode *CreatList(T &a){ //建立动态链表
    int ArrayLength =  sizeof(a)/sizeof(a[0]);
    if(ArrayLength==0) return NULL;

    struct ListNode *head;
```

```
    struct ListNode *p;
    struct ListNode *newnode;
    for(int i=0; i<ArrayLength; i++){
        newnode = (struct ListNode *)malloc( sizeof(struct ListNode) );
//开辟空间
        newnode->val = a[i];
        if(i==0) head = p = newnode; //记下首地址
        else p->next = newnode;
        p = newnode;
    }
    p->next = NULL;
    return head;
}

void output(struct ListNode *head){
    struct ListNode *p;
    p = head;
    while(p != NULL){
        cout << p->val << " ";
        p = p->next;
    }
}

int main(){
    struct ListNode *LN;
    int a[] = {0,1,2,3};
    LN = CreatList(a);
    output(LN);
    return 0;
}
```

归并排序

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

void reversePairs(vector<int>& nums) {
    //递归结束
    if(nums.size() < 2) return;

    //nums对半分成左右两个left、right，返回的是排好序的
    int mid = nums.size() >> 1;
    vector<int> left(nums.begin(), nums.begin() + mid);
    vector<int> right(nums.begin() + mid, nums.end());
    reversePairs(left);
    reversePairs(right);

    //左右两部left、right 分归并 nums
```

```
    nums.clear();
    int i=0,j=0;
    while(i<left.size() && j<right.size())
        left[i]<right[j] ? nums.push_back(left[i++]) : nums.push_back(rig
ht[j++]);
    while(i<left.size())
        nums.push_back(left[i++]);
    while(j<right.size())
        nums.push_back(right[j++]);
}

int main() {
    vector<int> nums = {1,65,8,7,500};
    reversePairs(nums);
    for(auto e:nums) cout<<e<<' ';
    return 0;
}
```

几个常用的std库函数

说明：std 是个名称空间标示符，要调用封装好的库函数需要用std::表明，例如 std::cout<<a; std::max(a,b); 但是为了书写方便在程序开头写上 using namespace std; 使用库函数时就不用标明了

```
头文件 algorithm
max（数值，数值）
sort（容器头地址，容器尾地址，定义比较方式）
find（容器头地址，容器尾地址，查找的东西） //能找到返回东西所在迭代器，找不到返回容器尾地址
reverse(容器头地址，容器尾地址);
auto it=容器.begin()   //迭代器的数据类型可以用auto
string 转 char[] 方法 ： char *c = str.c_str();
int 转 string 的方法 : string s = to_string(val)
string 转 int 的方法 ：int val = stoi(str);
                或者：int val = atoi(str.c_str());
```

string也是向vector、map等一样的容器

```
    元素访问
        s.at(pos)   访问指定字符,和下标访问一样 s[pos]
        s.front()   访问首字符
        s.back()    访问最后的字符
    容量
        s.empty()   检查字符串是否为空
```

```
        s.size() 或 s.length()  返回字符数
    插入
        s.insert(pos, c); 在pos位置前插入字符c
        s.insert(pos, n , c); 在pos位置前插入n个字符c
        s.insert(pos, str); 在pos位置前插入字符串str
        s.push_back(c)  等价与 s.insert(s.size(),c) 等价于 s += c
        s.append(str)   等价与 s.insert(s.size(),str) 等价于 s += str
    删除
        s.erase(迭代器); 移除所在字符
        s.erase( iterator first, iterator last ); 移除范围 [first, last) 中
的字符
        s.erase(s.begin()); 移除首字符
        s.clear() 清除所有内容 等价于  s.erase(s.begin(), s.end());
        s.pop_back()  移除末尾字符 等价于 s.erase(s.end()-1);
    查找
        auto it = s.find("is"); //搜索整串
        auto it = s.find("is", pos); //从字符串的pos位置开始搜索
    其他操作
        s.substr(pos, count) //返回子串 [pos, pos+count)
        s.substr(pos) //返回子串 [pos, s.size())
        s.swap(s2) //字符串s和s2交换内容
```

二叉树

```
/**
    二叉树 (BT)
     - 指针开辟空间用 malloc(空间大小)
     - 使用递归方法实现二叉树建立
     - 不用递归的话需要用列表记录
     - 三种方式遍历二叉树
*/
#include<vector>
#include<iostream>
using namespace std;

struct BinaryTreeNode{
    int val;
    struct BinaryTreeNode *left;
    struct BinaryTreeNode *right;
};

/*** 用递归建立二叉树 ***/
struct BinaryTreeNode *CreatBinaryTree(){
    int a;
    cin >> a;    //愚蠢的做法，多少叶子就需要输入2倍个 0 才能结束创建进程
    if(a == 0) return NULL;

    struct BinaryTreeNode *newnode;
```

```
    newnode = (struct BinaryTreeNode*)malloc( sizeof(struct BinaryTreeNod
e) );
    newnode->val = a;
    newnode->left  = CreatBinaryTree();  //递归创建左子树
    newnode->right = CreatBinaryTree();  //递归创建右子树
    return newnode;
}

//先序遍历
void PreOrderTraverse(struct BinaryTreeNode *root){
    if(root){
        cout << root->val << ' ';
        PreOrderTraverse(root->left);
        PreOrderTraverse(root->right);
    }
}

//中序遍历
void InOrderTraverse(struct BinaryTreeNode *root){
    if(root){
        InOrderTraverse(root->left);
        cout << root->val << ' ';
        InOrderTraverse(root->right);
    }
}

//后序遍历
void LastOrderTraverse(struct BinaryTreeNode *root){
    if(root){
        LastOrderTraverse(root->left);
        LastOrderTraverse(root->right);
        cout << root->val << ' ';
    }
}

//二叉树节点总数目
int NodeNum(struct BinaryTreeNode *root){
    if(root)
        return 1+NodeNum(root->left)+NodeNum(root->right);
    else
        return 0;
}

//二叉树叶子节点数
int LeafNum(struct BinaryTreeNode *root){
    if(!root)
        return 0;
    else if(  (root->left == NULL) && (root->right == NULL) )
        return 1;
    else
        return LeafNum(root->left) + LeafNum(root->right);
}
```

```
//二叉树最大深度
int maxDepth(struct BinaryTreeNode *root){
    if(root == NULL) return 0;
    return max(maxDepth(root->left), maxDepth(root->right)) + 1;
}

//二叉树最小深度(从根节点到最近叶子节点,注意是叶子)
int minDepth(struct BinaryTreeNode *root){
    if(root == NULL) return 0;
    if(root->left  == NULL) return minDepth(root->right) + 1;
    if(root->right == NULL) return minDepth(root->left)  + 1;
    return min(minDepth(root->left), minDepth(root->right)) + 1;
}


//二叉树所用路径
void dfspath(BinaryTreeNode* node,string path,vector<string> &result){
    path += to_string(node->val);

    //遍历到子节点了,
    if(node->left==NULL && node->right==NULL) result.push_back(path);

    if(node->left!=NULL)  dfspath(node->left, path + "->",result);
    if(node->right!=NULL) dfspath(node->right,path + "->",result);
}
vector<string> binaryTreePaths(BinaryTreeNode* root) {
    vector<string> result;
    if(root!=NULL) dfspath(root,"",result);
    return result;
}



int main()
{
    struct BinaryTreeNode *root;
    root = CreatBinaryTree();

    cout<<"二叉树总节点数为："<<NodeNum(root)<<endl;

    cout<<"二叉树叶子节点数为："<<LeafNum(root)<<endl;

    cout<<"二叉树最大深度(从根节点到最远叶子节点)："<<maxDepth(root)<<endl;

    cout<<"二叉树最小深度(从根节点到最近叶子节点)："<<minDepth(root)<<endl;

    cout<<"前序遍历结果:";
    PreOrderTraverse(root);
    cout<<endl;

    cout<<"中序遍历结果:";
```

```
    InOrderTraverse(root);
    cout<<endl;

    cout<<"后序遍历结果:";
    LastOrderTraverse(root);
    cout<<endl;

    cout<<"二叉树所有路径：\n";
    vector<string> allPaths = binaryTreePaths(root);
    for(auto  e:allPaths)   cout <<"\t" << e <<endl;


    return 0;
}

/*
测试用例:
3 9 0 0 20 15 0 0 7 0 0

1 2 0 5 0 0 3 0 0
*/
```

set

```
/******************
set中使用结构体
    find()的使用
*****************/
#include <iostream>
#include <set>
using namespace std;

struct Student {
    string name;
    int age;
    string sex;
};

/*“仿函数"。为Student set指定排序准则*/
class studentSortCriterion {
    public:
        bool operator() (const Student &a, const Student &b) const {
            /*先比较名字；若名字相同，则比较年龄。小的返回true*/
            if(a.name < b.name) return true;
            else if(a.name == b.name) {
                if(a.age < b.age)   return true;
                else    return false;
            } else
                return false;
```

```
        }
};

int main()
{
    set<Student, studentSortCriterion> stuSet;

    Student stu1, stu2;
    stu1.name = "张三";
    stu1.age = 13;
    stu1.sex = "male";
    stuSet.insert(stu1);

    stu2.name = "李四";
    stu2.age = 23;
    stu2.sex = "female";
    stuSet.insert(stu2);

    /*构造一个测试的Student，可以看到，即使stuTemp与stu1实际上并不是同一个对象，
     *但当在set中查找时，仍会查找成功。这是因为已定义的studentSortCriterion的缘故。
     */
    Student stuTemp;
    stuTemp.name = "张三";
    stuTemp.age = 13;

    set<Student, studentSortCriterion>::iterator iter;
    iter = stuSet.find(stuTemp);
    if(iter != stuSet.end()) {
        cout << (*iter).name << endl;
    } else {
        cout << "Cannot fine the student!" << endl;
    }

    return 0;
}
```

map

```
/*  这是一个关于 map 的demo
    插入  insert(key,val)--只有在key不存在时插入，返回指向该元素的迭代器和一个布尔值来说明是否成功的被插入了
    查找  map[key] -- 把键当成下标使用来访问值
          count(key)--返回返回键值等于key的元素的个数
          find(值)--如果找到键值为key的元素则返回其地址(iterator型)，如果没有找到则返回尾的地址
          begin()--返回头的地址(iterator型)
          end()--返回尾的地址(iterator型)
    大小  empty()--判断是否为空
```

```
            size()--返回集合中元素的数目
    删除   clear()--清除所有元素
            erase()--删除集合中的元素
                erase(iterator)  ,删除定位器iterator指向的值
                erase(first,second),删除定位器first和second之间的值
                erase(key),删除键值key 的元素
 * 时间：2018/9/24
 */
#include <map>
#include <iostream>
using namespace std;
int main() {
    map<int,int> numspair={{1,2},{2,2},{99,1}};
    cout<<numspair[99]<<endl; //键值即相当与下标来用

    //插入，或make_pair实现
    numspair.insert({100, 5}); //通过花括号构造
    numspair.insert(make_pair(101,6)); //make_pair实现
    cout<<numspair[100]<<endl;
    cout<<numspair[101]<<endl;
    numspair.insert({100, 6});//key值已存在，第二次插入不成功
    cout<<numspair[100]<<endl;

    //插入，用下标实现
    numspair[3]=3;
    cout<<numspair[3]<<endl;


    map<string,int> student={{"张三",20},{"李四",18},{"王五",18}};
    cout<<student["张三"]<<endl;
}
```

list 就像链表

```
/* 这是一个关于 list 的demo
List和Vector都是STL的序列式容器，唯一不同的地方就在于：Vector是一段连续的内存空间，List则是一段不连续的内存空间.
相比于Vector来说，List在每次插入和删除的时候，只需要配置或释放一个元素空间，对于任何位置的插入和删除操作，List永远能做到常数时间。
但是，List由于不连续的内存空间，导致不支持随机寻址.
元素访问
front   访问第一个元素
back 访问最后一个元素
begin   返回指向容器第一个元素的迭代器
end   返回指向容器尾端的迭代器

容量
empty 检查容器是否为空
size   返回容纳的元素数
```

```
max_size  返回可容纳的最大元素数

修改器
clear 清除内容
insert(iterator pos, const T& value)  在 pos 前插入 value
push_back  将元素添加到容器末尾
pop_back  移除末元素
push_front  插入元素到容器起始
pop_front  移除首元素
resize  改变容器中可存储元素的个数
swap 交换内容

操作
merge  合并二个已排序列表
reverse  将该链表的所有元素的顺序反转
unique  删除连续的重复元素
sort  对元素进行排序
*/
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;

int main() {
    list<int> l = { 7, 5, 16, 8 };

    // 添加整数到 list 开头
    l.push_front(25);
    // 添加整数到 list 结尾
    l.push_back(13);

    // 以搜索插入 16 前的值
    auto it = find(l.begin(), l.end(), 16); //find 在 algorithm 库函数中
    if (it != l.end()) {
        l.insert(it, 42);
    }

    // 迭代并打印 list 的值
    for (int n : l) {
        cout << n << '\n';
    }
}
```