目录

[1. C/C++ 2](#_Toc506650480)

[1.1 const关键字 2](#_Toc506650481)

[1.2函数传值与传引用 7](#_Toc506650482)

[1.3 字符串转整型问题 8](#_Toc506650483)

[1.4 操作符重载 8](#_Toc506650484)

[2. Python 9](#_Toc506650485)

[3. Algorithm 9](#_Toc506650486)

[3.1 时间复杂度计算 9](#_Toc506650487)

[3.2 二叉树 9](#_Toc506650488)

[3.3 最大堆 10](#_Toc506650489)

[3.4 红黑树 10](#_Toc506650490)

[3.5 算法复杂度 10](#_Toc506650491)

[3.6 寻找链表倒数第k个节点 10](#_Toc506650492)

[4. OS 11](#_Toc506650493)

# C/C++

## const关键字

const关键字常用于数组边界和switch条件分支标号：

const type variable; //常变量等价于 type const variable

const type &reference; //常引用等价于 type const& variable

ClassName const Object; //常对象等价于const ClassName Object

ClassName::func(signature) const; //常成员函数

Type const ArrayName[size]; //常数组等价于 const type ArrayName[size]

const type\* pointer; //常指针定义方法1

type const\* pointer; //常指针定义方法2

C标准中，const变量定义为全局作用域，而C++中则要视其定义位置而定；

使用指针时涉及到两个关键要素：指针本身(地址)和指针所指向的对象(数据类型)；

将一个指针声明为const类型，是限制其指向的对象为指定类型；

将指针本身声明为type \*const pointer，则是将指针(地址)本身声明为常量；

可以将一个非const对象的地址赋给const对象的指针，反之不可；

void Func(const A \*arg); //传指针传递参数

void Func(const A &arg); //传引用传递参数

将函数传入参数声明为const，提高函数运行效率且禁止修改其内容；

在C++中使用const关键字修饰成员函数，则const对象只能访问const成员函数，而非const对象可以访问所有成员函数；const对象的成员不可修改，const成员函数不可修改对象的数据，无论对象是否具有const限定；

常量指针与指针常量：

const int \*p1 = new int(10); //指向常量的指针，不可通过p1修改其指向的内容

int \*const p2 = new int(10); //指向int类型的指针常量，不可指向其他内存

程序载入内存时，会分配常量区存储常量，可以通过直接修改变量数值或通过另外一个非常量指针进行修改：

int a = 10;

const int \*p = &a;

// error

// \*p = 100;

//直接修改变量

a = 100;

//

int \*pi = (int\*) p; //第三方指针

\*pi = 100;

函数参数中的指针常量表示不允许此指针指向其他内容：

void func(int \*const pt){

int \*p = new int(10);

pt = p; //error cannot point to a new pointer

}

函数参数中的常量指针表示此参数不允许被修改：

void func(const int \*pt){

\*pt = 100; //error cannot be modified

}

若将参数中的指针赋给一个新的指针，则会修改其指向的内容；

常量与引用的关系：引用就是变量的别名，常量引用即不允许此引用成为其他变量的别名；

int a = 10;

const int& ra = a; //常量引用，不可通过此引用改变其对应的内容

// int& const ra = a; //error

若不希望函数调用者修改参数本身的值，最可靠的方法是传递引用；

void func(const int& arg){

// arg = 100; //error cannot be modified

}

系统在加载程序时，会将内存分为4个区域：堆、栈、数据段和代码段，使用常量的方式保护数据是通过编译器语法规则限定来实现的，但是仍然可以通过以下方式完成修改：

const int a = 10;

int \*pa = (int\*) &a; //可以通过定义其他指针的方法修改常量数值

\*pa = 100;

常量函数在C++中可防止类的 数据成员被非法访问，将类的成员函数分为两类：常量成员函数和非常量成员函数；

class Test{

public:

void func() const;

private:

int intValue;

};

void Test::func() const{

intValue = 100; // raise error 常量函数尝试改变数据成员intValue的数值

//编译时引发异常

}

class Fred{

public:

void inspect() const;

void mutate();

};

void UserCode(Fred& changeable, const Fred& unChangeable){

changeable.mutate(); //非常量对象调用非常量函数

changeable.inspect(); //非常量对象调用常量函数

unChangeable.mutate(); //常量对象调用非常量函数，错误

unChangeable.inspect(); //常量对象只能调用常量函数

}

常量函数包含一个this的常量指针：

void inspect(const Fred\* this) const;

void mutate(Fred\* this);

对于常量函数，不能通过this指针修改对象对应的内存，但可以通过重新定义指针来修改内存中的内容；

void func(const int \*pt){

int \*ptr = (int\*) pt;

\*ptr = 100;

}

通过常量对象调用非常量函数将产生语法错误；对于常量函数，不能通过this指针修改对象对应的内存块；但是可以通过this指针重新定义一个指向同一内存单元的指针；

void Fred::inspect() const{

Fred \*pFred = (Fred\*) this;

pFred->intValue = 50;

}

对于常量对象，可以构造新指针，指向常量对象所在的内存单元；

C++允许在类的数据成员定义前加上mutable关键字以实现成员在常量函数中可修改：

class Fred{

public:

void inspect() const;

private:

mutable int intValue;

};

常量函数的重载问题：

class Fred{

public:

void func() const;

void func();

}

void Fred::func() const{}

viod Fred::func(){}

void UserCode(Fred& fred, const Fred& cFred){

fred.func(); //call func()

cFred.func(); //call func() const

}

int main(int argc, char\*\* argv){

Fred fred;

UserCode(fred, fred);

return 0;

}

当存在 同名同参数相同返回值的函数重载时，具体调用哪个函数取决于调用对象是常量对象还是非常量对象；

常量返回值：不希望函数调用者修改函数的返回值，将函数返回一个常量。

* Const常量与宏定义的区别：const常量有数据类型(编译器做类型检查)，宏定义没有数据类型(编译器仅作简单的字符替换)；
* Const修饰类的数据成员，const数据成员只在对象的生命周期内是常量，而对整个类而言是可变的，因此，const数据成员的初始化只能在类的构造函数的初始化列表中进行，不能在类中指定；

class A{

const int size = 100; //错误定义

int array[size]; //错误定义，未知的size

}

要在类中定义常量，应当使用enum类型：

class A{

enum{size1=100, size2=200};

int array1[size1];

int array2[size2];

}

枚举常量不占用对象的内存空间，在编译时全部求值，枚举常量隐含使用整型类型，最大数值有限，不能表示浮点型数据；

* const初始化：

T b;

const T a = b; //非指针const常量初始化

T\* p = new T();

const T\* pc = p; //指针常量初始化

const T\* pb = new T(); //同上

T r;

const T& rf = r; //rf只能访问const成员函数

const T\* c = new T();

T\* e = c; //声明指针的目的是要修改其指向的内容，但此处指向常量

T\* const c = new T();

T\* e = c; //声明指针指向的内容可变

* 参数const通常用于参数为指针或引用的情况下使用，若参数为值传递，则函数会自动产生临时变量复制，保护被传递对象的属性；
* 对于非内部数据类型的参数传递，将参数值传递改为const引用传递可以提高效率；而对于内部数据类型的参数传递，避免改为const引用传递，以免降低程序的可理解性；
* 修饰返回值的const关键字，const T func(); const T\* func();对函数返回值进行保护；
* 函数的返回值声明为const，通常用于操作符重载，若使用const修饰函数的返回值类型，则返回的const Object只能访问类中的公有数据成员和const成员函数，且无法对其进行赋值操作；指针传递函数返回值加const修饰符，则函数返回内容(指针)不能被修改，只能赋给同样加const修饰符的同类型指针；

const char\* GetString(void);

char \*str = GetString(); //compile error

const char\* str = GetString(); //compile success

* 函数返回值引用传递通常用于类的赋值函数中，实现链式表达：

class A{

A &operator = (const A& other); //赋值操作符

}

A a, b, c; //class A Object a, b, c

…

a = b = c; //all right

(a = b) = c; //legal but unusual

a.operator = (b)的返回值是const类型的引用，不可再次赋值给c；

返回值的内容不允许被修改；

* 类成员函数const关键字的使用：

class Stack{

public:

void Push(int elem);

int Pop(void);

int GetCount(void) const; //const成员函数

private:

int m\_num;

int m\_data[100];

};

//公有成员函数定义在类外

int Stack::GetCount(void) const{

++m\_num; //编译错误，企图修改数据成员

Pop(); //编译错误，企图调用非const成员函数

return m\_num;

}

* 在C中，const是一个不能被改变的普通变量，需要占用存储空间，编译器不知道编译时的数值，且数组下标必须为常量；而在C++中，将const看做编译时的常量，不为其分配存储空间，只是在编译时将其数值存储在名字列表中；
* 在C语言中，const int size; 语句正确，在C++中不正确；C编译器默认使用外部连接，将其看做声明，可在其他地方分配内存空间；C++编译器默认使用内部链接，必须在声明时初始化，将const对象默认看做文件的局部变量；
* 在C++中，是否为const 分配内存空间取决于是否添加extern关键字或者取const变量地址；

## 1.2函数传值与传引用

函数传值：压栈的是参数的副本，函数对传递参数的操作作用在参数副本之上，不会修改原始值本身；

函数传指针(地址)：压栈的是地址的副本，但指针指向的是同一内存单元，修改操作会作用在原内存之上；

函数传引用：压栈的是引用的副本，但引用指向同一个变量地址，即同变量的两个别名，对引用的操作即对其指向的变量的操作；

值传递与引用传递的区别：

值传递(passed-by-value)：被调函数的形式参数作为被调函数的局部变量处理，为其开辟内存空间，存放传递进来的主调函数的实参数值，成为实参的一个副本，对被调函数局部变量的操作即对实参副本的操作，因此不会改变主调函数中实参的原始数值；

引用传递(passed-by-reference)：被调函数的形式参数也被当做被调函数的局部变量处理，并在堆栈中开辟内存空间，但其存储的是主调函数实参的地址，因此，被调函数实际上是对主调函数实参的间接引用，操作作用在实参原始数值之上，会改变实参的数值；

## 1.3 字符串转整型问题

将一个字符串转换成整数，如”0295”转换成0295 (Microsoft)，容易写出以下代码：

int StrToInt(char\* string){

int number = 0;

while(\*string != 0){

number = number \* 10 + \*string – ‘0’;

++string;

}

return number;

}

需要考虑以下问题：

* 程序鲁棒性：函数传入空指针判断；
* 输入字符串正负号判断；
* 最大正整数，最小负整数及溢出问题；
* 非法字符输入；

## 1.4 操作符重载

# Python

## 可变类型与非可变类型

类型属于对象而非变量即对象才有类型，而变量没有类型；对象分为可变对象(mutable)和非可变对象(immutable)两种，常见的可变对象有list/dict/set，常见的不可变对象有strings/tuples/numbers；

当不可变对象的引用传递给函数时，函数自动生成一份引用的拷贝，函数内部的操作均在拷贝上进行，不会改变外部对象；

当可变对象的引用传递给函数时，函数内的引用指向可变对象，如同指针一样，对其操作即对定位的指针地址一样，在内存中完成对对象的修改；

a = 1

def fun(a):

a = 2

fun(a)

print a # 1

a = []

def fun(a):

a.append(1)

fun(a)

print a # [1]

## 静态方法，实例方法与类方法

Python中有三种方法：静态方法@staticmethod 类方法@classmethod 和实例方法：

def foo(x):

print "executing foo(%s)"%(x)

class A(object):

def foo(self,x):

print "executing foo(%s,%s)"%(self,x)

@classmethod

def class\_foo(cls,x):

print "executing class\_foo(%s,%s)"%(cls,x)

@staticmethod

def static\_foo(x):

print "executing static\_foo(%s)"%x

a=A()

函数参数self：对实例(object)的绑定，在类中每次定义方法都要绑定实例self，foo(self, x)调用可修改实例自身；

函数参数cls：对类(class)的绑定，类方法A.class\_foo(x)传递的是类class而非实例；

静态方法与普通方法一样，不需要绑定类或实例，仅需要通过实例调用a.static\_foo(x)或通过类调用A.static\_foo(x)；

## 类变量与实例变量

类变量可在类间共享，不会单独分配给每个实例；

实例变量是指实例化后，实例单独拥有的变量；

class Test(object):

num\_of\_instance = 0

def \_\_init\_\_(self, name):

self.name = name

Test.num\_of\_instance += 1

if \_\_name\_\_ == '\_\_main\_\_':

print Test.num\_of\_instance # 0

t1 = Test('jack')

print Test.num\_of\_instance # 1

t2 = Test('lucy')

print t1.name , t1.num\_of\_instance # jack 2

print t2.name , t2.num\_of\_instance # lucy 2

class Person:

name="aaa"

p1=Person()

p2=Person()

p1.name="bbb"

print p1.name # bbb

print p2.name # aaa

print Person.name # aaa

参数传递问题，p1.name最初指向的是类变量name = “aaa”，string为不可变对象，实例单独产生拷贝，不会改变类变量本身，而list []为可变对象，每个实例均会产生对类变量的引用，实例方法对类变量的操作通过间接引用作用在类变量上，在实例的作用域内将类变量的引用改变为实例变量；

class Person:

name=[]

p1=Person()

p2=Person()

p1.name.append(1)

print p1.name # [1]

print p2.name # [1]

print Person.name # [1]

# -\*- coding: utf-8 -\*-

# -\*- versison: python 3.4.5 -\*-

# class\_method.py

class Test(object):

num\_of\_instance = 0

name\_of\_instance = "Object1"

queue\_of\_instance = []

def \_\_init\_\_(self, name):

self.name = name

Test.num\_of\_instance += 1

Test.name\_of\_instance = name

Test.queue\_of\_instance.append(name)

if \_\_name\_\_ == '\_\_main\_\_':

print("Class attribute : ")

print("Test.num\_of\_instance = ", Test.num\_of\_instance) # 0

print("Test.name\_of\_instance = ", Test.name\_of\_instance) # "Object"

print("Test.queue\_of\_instance = ", Test.queue\_of\_instance) # []

# Create object named 'jack'

t1 = Test('jack')

print("After instantiation : ")

print("Test.num\_of\_instance = ", Test.num\_of\_instance) # 1

print("Test.name\_of\_instance = ", Test.name\_of\_instance) # "jack"

print("Test.queue\_of\_instance = ", Test.queue\_of\_instance) # ['jack']

# Object t1 shares class variable

print("t1.num\_of\_instance = ", t1.num\_of\_instance) # 1

print("t1.name\_of\_instance = ", t1.name\_of\_instance) # "jack"

print("t1.queue\_of\_instance = ", t1.queue\_of\_instance) # ['jack']

# Create object named 'tom'

t2 = Test('tom')

print("Test.num\_of\_instance = ", Test.num\_of\_instance) # 2

print("Test.name\_of\_instance = ", Test.name\_of\_instance) # "tom"

print("Test.queue\_of\_instance = ", Test.queue\_of\_instance) # ['jack', 'tom']

# Object t2 shares class variable

print("t2.num\_of\_instance = ", t2.num\_of\_instance) # 2

print("t2.name\_of\_instance = ", t2.name\_of\_instance) # "tom"

print("t2.queue\_of\_instance = ", t2.queue\_of\_instance) # ['jack', 'tom']

# number, string as immutable variable

# list as mutable variable

t1.name\_of\_instance = “jack”

print("t1.num\_of\_instance = ", t1.num\_of\_instance) # 2

print("t1.name\_of\_instance = ", t1.name\_of\_instance) # "jack"

print("t1.queue\_of\_instance = ", t1.queue\_of\_instance) # ['jack', 'tom']

## Python自省特性

面向对象语言可在运行时获得对象的类型，常用自省函数：

type()/dir()/getattr()/hasattr()/isinstance()

## 列表推导式与字典推导式

列表推导式与字典推导式具有高效简短的特性：

list = [element for element in iterable]

dict = {key: value for (key, value) in iterable}

使用内建函数enumerate()赋予元素下标：

{i: el for i, el in enumerate([“one”, “two”, “three”])}

# 不使用内建函数enumerate()

lst = [“one”, “two”, “three”]

i = 0

for e in lst:

lst[i] = ‘%d: %s’ %(i, lst[i])

i += 1

## 深拷贝与浅拷贝

## 类方法\_\_new\_\_( )与\_\_init\_\_( )

## 常见的几种设计模式

## 编码与解码

## 迭代器与生成器

迭代器可以减少内存开销：迭代器属于临时区，安排一些元素在其中，使用时才创建临时区，一旦遍历结束即清空临时区，再次遍历时临时区失效。

# -\*- coding: utf-8 -\*-

# -\*- version: python 3.5.4 -\*-

# \_iter.py

import sys

i = iter(range(10000))

print("id(i.\_\_next\_\_()) = ", id(i.\_\_next\_\_()))

print("sys.getsizeof(i) = ", sys.getsizeof(i))

print("sys.getsizeof(i.\_\_next\_\_()) = ", sys.getsizeof(i.\_\_next\_\_()))

# 若一次性将list对象全部加载进来需要90112bytes

# 使用迭代器仅需要28bytes

e = range(10000)

print("sys.getsizeof(e) = ", sys.getsizeof(e))

print("sys.getsizeof(list(e))", sys.getsizeof(list(e)))

可以使用next()函数不断返回下一个值的对象称为迭代器iterator，生成器都是迭代器对象；list dict string虽然具有可迭代属性，但不是Iterator，可以使用iter()转换成iterator，验证是否是迭代器的方法即查看是否具有.\_\_next\_\_()方法；

生成器是一种特殊的迭代器，常在性能限制的条件下应用，readline()/readlines()就是一种常用的生成器，可在循环读取时不断处理，节省内存空间；

生成生成器的方法：

* 方法一：g = (x\*x for x in range(10)) # 类似迭代器
* 在函数中加入yield关键字用来返回值，yield遇到next()方法即返回值，再次执行时从上次yield返回处继续执行；

查看一个函数是否是生成器 ：

使用自省函数dir(g)，返回：

[‘\_\_class\_\_’,

….

‘\_\_next\_\_’,

…

‘\_\_repr\_\_’,

‘\_\_setattr\_\_’,

]

其中，魔术方法\_\_next\_\_()是生成器特有的属性，可通过调用g.\_\_next\_\_()或next(g)获得下一个生成的对象；

## 装饰器

在函数之后进行装饰；

使用装饰器实现单例模式；

## 邮箱地址正则表达式

## 垃圾回收

引用计数/分代回收/孤立引用环

# Algorithm

## 时间复杂度计算

待解决问题的规模为n，基本操作被重复执行的次数为n的函数，时间复杂度记作：T(n) = O[f(n)]，表示随着问题规模n的增长，算法执行的时间增长率增长率与f(n)的增长率相同；

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

for(j=1; j<=i; j++){

++x; //执行频度为1+2+3+…+n=(n+1)n/2

a[i,j] = x;

}

}

常见的复杂度：

* 常数阶O(1)：没有循环(for/while)，仅对变量做常数范围以内的操作；
* 线性阶O(n)：一层循环

int i;

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

//operations

}

* 对数阶O(log(n))：

int count = 1;

while(count < n){

count = count \* 2;

}

未限定问题规模，2^x = n；得出x = log(n)，算法复杂度为O(log(n))；

* 平方阶O(n^2)：两重循环

int i,j;

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

for(j=0; j<n; j++){

//operations

}

}

## 二叉树

### 3.2.1 二叉树的数据结构



二叉树的三种遍历：

* 先序遍历：ParentNode—LeftNode—RightNode；
* 中序遍历：LeftNode—ParentNode—RightNode；
* 后序遍历：LeftNode—RightNode—ParentNode；
* 层序遍历：逐层遍历；

二叉树常用作二叉查找树、二叉堆和二叉排序树；

//二叉树节点数据结构

typedef struct BiTreeNode{

// Data Field

char data;

// Child pointer

struct BiTreeNode \*lchild, \*rchild;

}BiTreeNode, \*BiTree;

### 3.2.2 二叉树的生成

// 先序生成创建二叉树

int CreateBiTree(BiTree &T){

char data;

// 按先序顺序输入二叉树节点中的数值

InputNode(&data);

if(CHARP\_VALUE == data){

T = NULL;

}

else{

T = (BiTree)malloc(sizeof(BiTreeNode));

T->data = data;

CreateBiTree(T->lchild);

CreateBiTree(T->rchild);

}

return 0;

}

### 3.2.3 二叉树的遍历算法

三种遍历均有递归实现与循环实现两种方法(广度优先/深度优先)：

广度优先：逐层遍历，从左至右依次访问，可利用FIFO队列实现广度优先搜索；

深度优先：先访问根节点，然后遍历左子树后遍历右子树，可利用栈FILO的特点，将右子树压栈后将左子树压栈；

可以使用堆(单数组加上堆的最末节点的下标)来表示完全二叉树；

二叉搜索树上的基本操作花费的时间与树的高度成正比，完全二叉树操作的时间复杂度为O(log(n))，二叉搜索树的性质：对二叉搜索树中的节点x，若y为其左子树中的节点，y.key<=x.key；若y为其右子树中的节点，y.key>=x.key；

* 二叉树遍历
* 二叉树查找
* 二叉树插入与删除

中序遍历的递归写法：

void InorderTreeWalk(pTree\* tree){

if(NULL != tree){

InorderTreeWalk(tree->pLeft);

StoreTree(tree->key);

InorderTreeWalk(tree->pRight);

}

}

先序遍历的递归写法：

void PreorderTreeWalk(pTree\* tree){

if(NULL != tree){

StoreTree(tree->key);

PreorderTreeWalk(tree->pLeft);

PreorderTreeWalk(tree->pRight);

}

}

后序遍历的递归写法：

void PostorderTreeWalk(pTree\* tree){

if(NULL != tree){

PostorderTreeWalk(tree->pLeft);

PostorderTreeWalk(tree->pRight);

StoreTree(tree->key);

}

}

先序遍历的非递归方法：先使根节点tree入栈s.push(tree)，只要栈不为空!s.empty()，即可使栈中元素出栈s.pop()，每次弹出一个节点，都要将其右孩子节点tree->pLeft入栈s.push(tree->pLeft)，再将其左孩子节点tree->pRight入栈s.push(tree->pRight)；

访问tree->key后，将tree入栈，遍历tree->pLeft，遍历左子树结束后，栈顶元素为tree，tree出栈，遍历右子树；

**void PreorderNonrecursive(BiTree T){**

**if(! T){**

**return ;**

**}**

**stack<BiTree> s; // STL**

**s.push(T); // push root node**

**while(! s.empty()){**

**BiTree temp = s.top(); // top element in stack**

**StoreNode(temp->data);**

**s.pop(); // pop root**

**if(NULL != temp->rchild){ // right child push**

**s.push(temp->rchild);**

**}**

**if(NULL != temp->lchild){ // left child push**

**s.push(temp->lchild);**

**}**

**}**

**}**

中序遍历的非递归方法：

将tree入栈，遍历左子树tree->pLeft，栈顶元素为tree，tree出栈，访问tree->key，遍历右子树tree->pRight；

引入指向当前节点的指针curr，以curr指向非NULL或栈非空为外循环条件，先将左子树全部入栈；若栈非空，则将curr指向栈顶元素，并将其出栈，访问该节点，将curr指针指向当前节点的右孩子节点；

**void InorderNonrecursive(BiTree T){**

**if(! T){**

**return ;**

**}**

**BiTree curr = T; //point to current node**

**stack<BiTree> s;**

**while(NULL != curr || ! s.empty()){**

**//左子树全部入栈**

**while(NULL != curr){**

**s.push(curr);**

**curr = curr->lchild;**

**}//while**

**if(! s.empty()){**

**curr = s.top(); // point to top element in stack**

**s.pop(); // pop element out**

**StoreNode(curr->data); // store element**

**curr = curr->rchild; // point to right child**

**}**

**}//while**

**}**

后序遍历的非递归方法：

将tree入栈，遍历左子树tree->pLeft，栈顶元素为tree，将tree出栈，遍历右子树tree->pRight；

当前节点指针curr，节点已访问标记previsited，以curr指针非NULL或栈非空为外循环条件，将左子树全部入栈；curr指向栈顶元素，若curr节点的右孩子节点为NULL或curr节点的右孩子节点已被访问：

T：访问当前节点，并标记当前节点的previsited标记，curr节点出栈，并置curr为NULL；

F：否则访问curr的右孩子节点；

**void PostorderNonrecursive(BiTree T){**

**stack<BiTree> s;**

**BiTree curr = T; // point to current node**

**BiTree previsited = NULL;**

**while(NULL != curr || ! s.empty()){**

**while(NULL != curr){**

**s.push(curr); // 左子树全部入栈**

**curr = curr->lchild;**

**} //while**

**curr = s.top(); //左子树叶子节点**

**/\* 右子树为空或已被访问过，则访问当前节点并标记当前节点已被访问**

**\* 当前节点出栈，curr指针指空；**

**\*/**

**if(NULL == curr->rchild || previstited = curr->rchild){**

**StoreNode(curr->data); //store current node**

**previsted = curr; //flag pointer**

**s.pop(); //current node pop out**

**curr = NULL; //current pointer to NULL**

**} //if**

**else{**

**curr = curr->rchild; //访问右子树中的左子树**

**} //else**

**} //while**

**}**

后续遍历的非递归遍历，双栈法：

**void PostOrderDoubleStack(BiTree T){**

**stack<BiTree> s1, s2;**

**BiTree curr; //指向当前节点**

**s1.push(T);**

**while(! s1.empty()){**

**curr = s1.top();**

**s1.pop();**

**s2.push(curr);**

**if(curr->lchild != NULL){**

**s1.push(curr->lchild);**

**}**

**if(curr->rchild != NULL){**

**s1.push(curr->rchild);**

**}**

**}//while**

**while(! s2.empty()){**

**StoreNode(s2.top()->data);**

**s2.pop();**

**}**

**}**

二叉树的先序遍历，中序遍历和后序遍历均是DFS深度优先，易于使用递归方法实现；层序遍历是BFS广度优先，易于使用队列实现非递归方法：

int visit(BiTree T){

if(T){

StoreNode(T->data);

return 1;

}

else

return 0;

}

使用队列(FIFO)实现二叉树的层序遍历具有结构优势：

**void LevelOrderWalk(BiTree T){**

**queue<BiTree> Q;**

**BiTree p; //pointer to BiTree**

**p = T;**

**if(1 == visit(p)){**

**Q.push(p);**

**}**

**while(! Q.empty()){**

**p = Q.front();**

**Q.pop();**

**if(1 == visit(p->lchild)){**

**Q.push(p->lchild);**

**}**

**if(1 == visit(p->rchild)){**

**Q.push(p->rchild);**

**}**

**}**

**}**

二叉树求深度操作：

int BiTreeDepth(BiTNode \*T){

if(! T){

return 0;

}

int d1, d2;

d1 = BiTreeDepth(T->lchild);

d2 = BiTreeDepth(T->rchild);

return (d1>d2? d1:d2) + 1;

}

二叉树求节点数操作：

int BiTreeNodeCount(BiTreeNode \*T){

if(NULL == T){

return 0;

}

return (1 + BiTreeNodeCount(T->lchild) + BiTreeNodeCount(T->rchild));

}

## 最大堆

最大堆是一种完全二叉树，其父节点的关键字不小于其左右子节点的关键字；

## 红黑树

## B树

## 线性结构-栈与队列

栈与队列都是动态集合，栈是LIFO结构，只有栈顶指针指示当前所在位置，入栈操作PUSH和出栈操作POP只能在栈顶进行，栈有上溢出和下溢出错误；队列是FIFO结构，分别有队头指针和队尾指针，入队操作ENQUEUE在队尾进行，出队操作DEQUEUE在队头；

## 线性结构-链表

## 寻找链表倒数第k个节点

寻找链表中倒数第k个节点，定义两个指针pAhead和pBehind，使pAhead先走k-1步，然后两个指针一起移动，当pAhead移动到链表尾部时，pBehind所在位置即倒数第k个节点；容易写出以下代码：

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

Unsigned int i;

if(NULL == pListHead){

return nullptr;

}

ListNode\* pAhead = pListHead;

ListNode\* pBehind = nullptr;

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

pAhead = pAhead->next;

}

pBehind = pListHead;

while(nullptr != pAhead->pNext){

pAhead = pAhead->pNext;

pBehind = pBehind->pNext;

}

return pBehind;

}

需要考虑的问题：

* 输入参数为空指针，即pListHead为nullptr；
* 输入参数k非法，如k为0；
* 链表节点总数小于k；

## 快速排序

复杂度

最坏情况如何优化

## 堆排序

复杂度

最坏情况如何优化

## 无序数字列表寻找所有间隔为d的组合

## 列表[a1, a2, a3, …, an]求其所有组合

## 一行python代码实现1+2+3+…+10\*\*8

## 长度未知的单向链表求其是否有环

## 单向链表应用快速排序

## 长度为n的无序数字元素列表求其中位数

## 遍历一个内部未知的文件夹

# OS

## 多线程与多进程的区别

CPU密集型适合使用多线程还是多进程；

## 协程

## 进程间通信方式

## 虚拟存储系统中缺页计算

## 并发进程不会引起死锁的资源数量计算

## 常用的Linux/git命令和作用

## 查看当前进程的命令

# 网络

## TCP/IP协议

## OSI五层协议

## Socket长连接

## Select与epoll

## TCP与UDP协议的区别

## TIME\_WAIT过多的原因

## http一次连接的全过程描述

从用户发起request到用户接收到response

## http连接方式——get与post的区别

## restful

## http请求的状态码 200/403/404/504

# 数据库

## MySQL锁的种类

## 死锁的产生

## MySQL的char/varchar/text的区别

## Join的种类与区别

## A LEFT JOIN B的查询结果中，B缺少的部分如何显示

## 索引类型的种类

## BTree索引与hash索引的区别

## 如何对查询命令进行优化

## NoSQL与关系型数据库的区别

## Redis常用的存储类型