1. 常用的数据结构

1.Stack(栈)

C++ Stack（堆栈） 是一个容器类的改编，为程序员提供了堆栈的全部功能，——也就是说实现了一个**先进后出（FILO）**的数据结构。

**头文件为**: #include <stack>

**定义**：stack<char> sta;

**栈stack的成员函数介绍：**

**empty() 堆栈为空则返回真**

**pop() 移除栈顶元素 （删除）**

**push() 在栈顶增加元素 （增加）**

**size() 返回栈中元素数目**

**top() 返回栈顶元素，不删除（获取）**

栈（stack）是限制插入和删除只能在一个位置上进行的线性表，该位置在表的末端，叫做栈顶。添加元素只能在尾节点后添加，删除元素只能删除尾节点，查看节点也只能查看尾节点。添加、删除、查看依次为**入栈（push）、出栈（pop）、栈顶节点（top）**。形象的说，栈是一个先进后出（LIFO）表，先进去的节点要等到后边进去的节点出来才能出来。

**常见的题型：**如一个字符串需要处理特殊字符，如[]、()等。

例题：小Q想要给他的朋友发送一个神秘字符串，但是他发现字符串的过于长了，于是小Q发明了一种压缩算法对字符串中重复的部分进行了压缩，对于字符串中连续的m个相同字符串S将会压缩为[m|S](m为一个整数且1<=m<=100)，例如字符串ABCABCABC将会被压缩为[3|ABC]，现在小Q的同学收到了小Q发送过来的字符串，你能帮助他进行解压缩么？（牛客网）

2.C++ Vector容器

其实质是动态分配的数组结构。向量 vector 是一种对象实体, 能够容纳许多其他类型相同的元素, 因此又被称为容器。 与string相同, vector 同属于STL(Standard Template Library, 标准模板库)中的一种自定义的数据类型, 可以广义上认为是数组的增强版。vector 容器与数组相比其优点在于它能够根据需要随时自动调整自身的大小以便容下所要放入的元素。此外, vector 也提供了许多的方法来对自身进行操作。

**头文件**：include<vector>

**定义声明：**

vector<int> a ; //声明一个int型向量a

vector<int> a(10) ; //声明一个初始大小为10的向量

vector<int> a(10, 1) ; //声明一个初始大小为10且初始值都为1的向量 vector<int> b(a) ; //声明并用向量a初始化向量b

vector<int> b(a.begin(), a.begin()+3) ;//将a向量中从第0个到第2个(共3个)作为向量b的初始值

除此之外, 还可以直接使用数组来初始化向量:

int n[] = {1, 2, 3, 4, 5} ;

vector<int> a(n, n+5) ; /将数组n的前5个元素作为向量a的初值

vector<int> a(&n[1], &n[4]) ; //将n[1] - n[4]范围内的元素作为向量a的初值

在**元素的输出**上, 还可以使用遍历器(又称迭代器)进行输出控制。在 vector<int> b(a.begin(), a.begin()+3) ; 这种声明形式中, (a.begin()、a.begin()+3) 表示向量起始元素位置到起始元素+3之间的元素位置。(a.begin(), a.end())则表示起始元素和最后一个元素之外的元素位置。

向量元素的位置便成为遍历器, 同时, 向量元素的位置也是一种数据类型, 在向量中遍历器的类型为: vector<int>::iterator。 遍历器不但表示元素位置, 还可以再容器中前后移动

vector<int>::iterator t ;

for(t=a.begin(); t!=a.end(); t++)

cout<<\*t<<" " ;

\*t 为指针的间接访问形式, 意思是访问t所指向的元素值。

**成员函数：**

**a.size()**：获取向量中的元素个数

**a.empty()**：判断向量是否为空

**a.clear()**：清空向量中的元素

**a.push\_back():**从后面依次插入数据

a = b ：将b向量复制到a向量中

reverse(a.begin(),a.end()) 需要头文件#include<algorithm>：翻转函数

保持 ==、!=、>、>=、<、<= 的惯有含义；如: a == b ; //a向量与b向量比较, 相等则返回1

**Insert（插入函数）**：

①、 a.insert(a.begin(), 1000); //将1000插入到向量a的起始位置前

②、 a.insert(a.begin(), 3, 1000) ; //将1000分别插入到向量元素位置的0-2处(共3个元素)

③、 vector<int> a(5, 1) ;

vector<int> b(10) ;

b.insert(b.begin(), a.begin(), a.end());//将a.begin(), a.end()之间的全部元素插入到b.begin()前

**删除 - erase**

①b.erase(b.begin()) ; //将起始位置的元素删除

②b.erase(b.begin(), b.begin()+3) ;//将(b.begin(), b.begin()+3)之间的元素删除

**交换 - swap**

b.swap(a) ; //a向量与b向量进行交换

与数组相同, 向量也可以增加维数, 例如声明一个m\*n大小的**二维向量**方式可以像如下形式:

vector< vector<int> > b(10, vector<int>(5)); //创建一个10\*5的int型二维向量

在这里, 实际上创建的是一个向量中元素为向量的向量。同样可以根据一维向量的相关特性对二维向量进行操作。

3.单调栈

**定义：**从栈底元素到栈顶元素呈单调递增或单调递减，栈内序列满足单调性的栈。

1)当新元素在单调性上优于栈顶时（单增栈新元素比栈顶大，单减栈新元素比栈顶小），压栈，栈深+1；

(2)当新元素在单调性与栈顶相同（新元素于栈顶相同）或劣于栈顶时（单增栈新元素比栈顶小，单减栈新元素比栈顶大），弹栈，栈深-1。

**例题：**现在有一组数10，3，7，4，12。从左到右依次入栈，则如果栈为空或入栈元素值小于栈顶元素值，则入栈；否则，如果入栈则会破坏栈的单调性，则需要把比入栈元素小的元素全部出栈。单调递减的栈反之。

10入栈时，栈为空，直接入栈，栈内元素为10。

3入栈时，栈顶元素10比3大，则入栈，栈内元素为10，3。

7入栈时，栈顶元素3比7小，则栈顶元素出栈，此时栈顶元素为10，比7大，则7入栈，栈内元素为10，7。

4入栈时，栈顶元素7比4大，则入栈，栈内元素为10，7，4。

12入栈时，栈顶元素4比12小，4出栈，此时栈顶元素为7，仍比12小，栈顶元素7继续出栈，此时栈顶元素为10，仍比12小，10出栈，此时栈为空，12入栈，栈内元素为12。

应用：视野总和（小Q 看楼）、柱状图中的最大矩形等

4.链表List

链表是空节点，或者有一个值和一个指向下一个链表的指针，因此很多链表问题可以用递归来处理。链表的存储位置是任意的，用指针表示，一个结点包括数据域和指针域，也叫做数据的存储映像。链表开始叫做头指针，指向第一个结点，最后一个结点的指针域为空。

C++的链表表示：

* 构建链表结点结构体ListNode

 typedef struct ListNode {

 int val; //数据值

 ListNode \*next; //指向下一个链表结点的指针

 ListNode(int x) : val(x), next(NULL) {} //结构体的构造函数，数据赋初 始值，指针域为空指针

 }；ListNode; //起一个别名

* 创建结点

ListNode \*head=NULL;初始化链表头

head = new ListNode; //分配新结点，可以同时初始化

head->value = 12.5; //存储值

head->next = nullptr; //表示链表的结尾

//使用构造函数创建链表

ListNode \*secondPtr = new ListNode(13.5);

ListNode \*head = new ListNode(12.5, secondPtr);

* 遍历链表

ListNode \*ptr = numberList; //头结点

while (ptr != nullptr)

{

cout << ptr->value << " "; //处理结点（显示结点内容）

ptr = ptr->next; //移动到下一个结点

}

* 链表注意事项

1. 添加哑结点作为辅助，该结点位于列表头部。哑结点（新建一个在头结点的前面）用来简化某些极端情况，例如列表中只含有一个结点，或需要删除列表的头部。
2. 在遍历链表时要增加一个遍历兵，改变是整体链表也会改变。
3. 在进行链表操作，熟练使用双指针和递归操作。
4. 在求逆序时可以考虑把链表放入栈中来考虑
5. 解决链表问题最好的办法是在脑中或者纸上把链表画出来。如力扣328

* 例子：力扣160等链表题

涉及到链表的操作，一定要在纸上把过程先画出来，再写程序。

**5.树和二叉树**

树是n个结点的有限集，有一个根节点，称为父子关系。

二叉树建立：

struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode(int x) : val(x), left(NULL), right(NULL) {}

\* };

一棵树要么是空树，要么有两个指针，每个指针指向一棵树。树是一种递归结构，很多树的问题可以使用递归来处理。

**树的搜索算法（递归和迭代）:**

* 递归：

function recursion(大规模)

{

if (end\_condition)

{

end;

}

else

{ //在将问题转换为子问题描述的每一步，都解决该步中剩余部分的问题。

solve; //back;

recursion(小规模); //go;

}

}

* 树的非递归遍历
* 树的BFS遍历，也就是层次遍历

（以上具体内容见：）

1. **队列**

* 队列（Queue）与栈一样，是一种线性存储结构，它具有如下特点：

（1）队列中的数据元素遵循“先进先出”（First In First Out）的原则，简称FIFO

（2）在队尾添加元素，在队头删除元素。

* 队头与队尾：允许元素插入的一端称为队尾，允许元素删除的一端称为队头；

入队：队列的插入操作；

出队：队列的删除操作。

定义与使用：

使用标准库的队列时, 应包含相关头文件，在栈中应包含头文件： #include< queue> 定义：queue< int > q;

q.empty() 如果队列为空返回true，否则返回false

q.size() 返回队列中元素的个数

q.pop() 删除队列首元素但不返回其值

q.front() 返回队首元素的值，但不删除该元素

q.push() 在队尾压入新元素

q.back() 返回队列尾元素的值，但不删除该元素

那么我们如何判断队列是空队列还是已满呢？

a、栈空： 队首标志=队尾标志时，表示栈空。

b、栈满 : 队尾+1 = 队首时，表示栈满。

* 队列的分类：

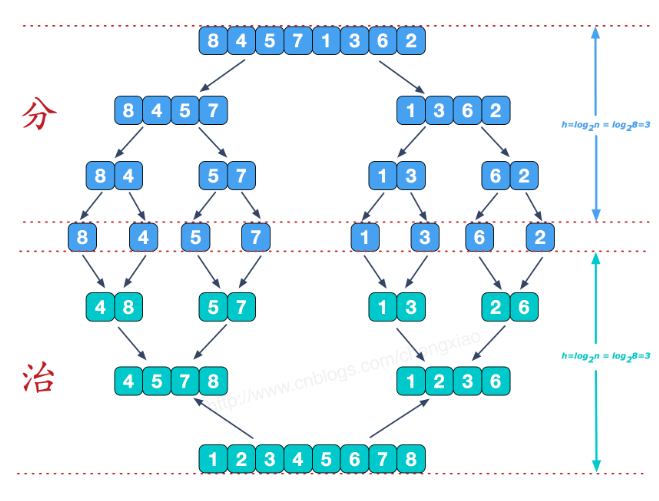
（1）基于数组的循环队列（循环队列）

（2）基于链表的队列（链队列）

**二 常用的算法**

1. 归并排序

归并排序（MERGE-SORT）是利用归并的思想实现的排序方法，该算法采用经典的**分治**（divide-and-conquer）策略（分治法将问题分(divide)成一些小的问题然后递归求解，而治(conquer)的阶段则将分的阶段得到的各答案"修补"在一起，即分而治之)。



例题：**在一个数组中求出逆序对**（归并排序使用分治策略，序列一分为二(O(1))后，将子序列递归排序(2 \* T(n / 2))，最后合并有序子序列(O(n)),T(n) = 2 \* T(n / 2) + O(n) = O(n \* logn)。）

1. 二分法查找

注意边界问题，以及二分法的变种问题。

正常实现

Input : [1,2,3,4,5]

key : 3

return the index : 2

public int binarySearch(int[] nums, int key) {

int l = 0, h = nums.length - 1;

while (l <= h) {

int m = l + (h - l) / 2;

if (nums[m] == key) {

return m;

} else if (nums[m] > key) {

h = m - 1;

} else {

l = m + 1;

}

}

return -1;

}

时间复杂度：

二分查找也称为折半查找，每次都能将查找区间减半，这种折半特性的算法时间复杂度为 O(logN)。

m 计算：

有两种计算中值 m 的方式：

**m = (l + h) / 2**

**m = l + (h - l) / 2**

l + h 可能出现加法溢出，也就是说加法的结果大于整型能够表示的范围。但是 l 和 h 都为正数，因此 h - l 不会出现加法溢出问题。所以，最好使用第二种计算法方法。

未成功查找的返回值

循环退出时如果仍然没有查找到 key，那么表示查找失败。可以有两种返回值：

-1：以一个错误码表示没有查找到 key

l：将 key 插入到 nums 中的正确位置

**变种**

二分查找可以有很多变种，实现变种要注意边界值的判断。例如在一个有重复元素的数组中查找 key 的最左位置的实现如下：

public int binarySearch(int[] nums, int key) {

int l = 0, h = nums.length - 1;

while (l < h) {

int m = l + (h - l) / 2;

if (nums[m] >= key) {

h = m;

} else {

l = m + 1;

}

}

return l;

}

该实现和正常实现有以下不同：

h 的赋值表达式为 h = m，循环条件为 l < h，最后返回 l 而不是 -1。在 nums[m] >= key 的情况下，可以推导出最左 key 位于 [l, m] 区间中，这是一个闭区间。h 的赋值表达式为 h = m，因为 m 位置也可能是解。

在 h 的赋值表达式为 h = m 的情况下，如果循环条件为 l <= h，那么会出现循环无法退出的情况，因此循环条件只能是 l < h。以下演示了循环条件为 l <= h 时循环无法退出的情况：

nums = {0, 1, 2}, key = 1

l m h

0 1 2 nums[m] >= key

0 0 1 nums[m] < key

1 1 1 nums[m] >= key

1 1 1 nums[m] >= key

...

当循环体退出时，不表示没有查找到 key，因此最后返回的结果不应该为 -1。为了验证有没有查找到，需要在调用端判断一下返回位置上的值和 key 是否相等。

1. 递归

何为递归？程序反复调用自身即是递归。既然递归是一个反复调用自身的过程，这就说明它每一级的功能都是一样的，因此我们只需要关注一级递归的解决过程即可。别去纠结这一层函数做了什么，它调用自身后的下一层函数又做了什么…然后就会觉得实现一个递归解法十分复杂，根本就无从下手。

解递归题的三部曲：

* 找整个递归的终止条件：递归应该在什么时候结束？
* 找返回值：应该给上一级返回什么信息？
* 本级递归应该做什么：在这一级递归中，应该完成什么任务？

一定要理解这3步，这就是以后递归秒杀算法题的依据和思路。

例1：求二叉树的最大深度

先看一道简单的Leetcode题目： Leetcode 104. 二叉树的最大深度

题目很简单，求二叉树的最大深度，那么直接套递归解题三部曲模版：

找终止条件。 什么情况下递归结束？当然是树为空的时候，此时树的深度为0，递归就结束了。

找返回值。 应该返回什么？题目求的是树的最大深度，我们需要从每一级得到的信息自然是当前这一级对应的树的最大深度，因此我们的返回值应该是当前树的最大深度，这一步可以结合第三步来看。

本级递归应该做什么。 首先，还是强调要走出之前的思维误区，递归后我们眼里的树一定是这个样子的，看下图。此时就三个节点：root、root.left、root.right，其中根据第二步，root.left和root.right分别记录的是root的左右子树的最大深度。那么本级递归应该做什么就很明确了，自然就是在root的左右子树中选择较大的一个，再加上1就是以root为根的子树的最大深度了，然后再返回这个深度即可。

class Solution {

public int maxDepth(TreeNode root) {

//终止条件：当树为空时结束递归，并返回当前深度0

if(root == null){

return 0;

}

//root的左、右子树的最大深度

int leftDepth = maxDepth(root.left);

int rightDepth = maxDepth(root.right);

//返回的是左右子树的最大深度+1

return Math.max(leftDepth, rightDepth) + 1;

}

}

当足够熟练后，也可以和Leetcode评论区一样，很骚的几行代码搞定问题，让之后的新手看的一脸懵逼(这道题也是我第一次一行代码搞定一道Leetcode题)：

class Solution {

public int maxDepth(TreeNode root) {

return root == null ? 0 : Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;

}

}

**更多例题见**<https://lyl0724.github.io/2020/01/25/1/>

三 C++ STL

1. Stringsteam ss

<sstream> 定义了三个类：istringstream、ostringstream 和 stringstream，分别用来进行流的输入、输出和输入输出操作。本文以 stringstream 为主，介绍流的输入和输出操作。

<sstream> 主要用来进行数据类型转换，由于 <sstream> 使用 string 对象来代替字符数组（snprintf方式），就避免缓冲区溢出的危险；而且，因为传入参数和目标对象的类型会被自动推导出来，所以不存在错误的格式化符的问题。简单说，相比c库的数据类型转换而言，<sstream> 更加安全、自动和直接。

**注意**：流操作，和cin cout类似，只是一个是屏幕流一个是字符流。如果遇到空格就结束了，遇到字符就读一个字符，他是从cin继承来的，符合cin的特性。