Breath First Search和Java Generics

### 理解Breath First Search

1. **引入**

在课上我们讲了三种binary tree的iteration，分别是pre-order, in-order and post-order，但是如果我们想要一层层地去打印tree的node，又称level-order呢？

比如给定一个二叉树，按照层数依次print out 3， 9， 20， 15， 7

3

/ \

9 20

/ \

15 7

1. **Breath First Search**

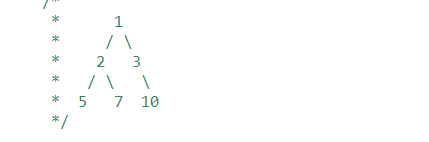
在Depth first search和binary search之后，我们现在将学习Breath first search：

如果说DFS在树中是从一个node直接找到它的叶子node，也就是最下面的左node或者右node，那么breath first search则是从开始的node一层层地往下找

具体的步骤是：从当前节点找到和它相关，连在一起的节点，加入未来的搜索路径当中（一般用queue来保存未来将要探索的节点），然后每次把queue里面的node给poll出来，再找到其连接的下面2个或者1个node放进queue里面，然后再poll出来第一个再找，直到从queue里面的node被poll出来之后发现不存在与之相连接的node为止

因为queue是满足依次往下的顺序，先加进去的会先被poll出来，后加进去的会被后poll出来，所以通过这样的方式，我们把某个node poll出来之后，我们会把它下一层的node全部加入queue，然后再poll下一个node（注意这里poll出来的一定是和刚刚那个在同一层的node），然后再把这个node的下一层node加入queue，所以queue里面能够按照顺序依次存放整一层的node

比如说我们的tree长这样：



我们会有以下的步骤执行：

1. **建立一个queue来保存还没搜索的node**
2. **把root，也就是1加入queue**
3. **poll出来一个node，因为queue里只有一个node，就是root，所以1这个node被poll出来了，queue目前是空的**
4. **把poll出来的node的左node和右node加入queue，所以现在queue里面又有2个node了，并且目前的顺序是2，3 （因为1的左边和右边的node都被放入queue中了，不存在其他1的child node了，所以我们loop的第一个轮回结束（也就是只要被poll的node都被放进queue，就表明loop进入下一轮））**
5. **再poll出来下一个node，也就是2，这个时候queue里面只剩下3了，那么对于poll出来的2，我们再找到其两个child node，也就是5和7放进queue里面，所以现在queue里面是3，5，7，因为2的子node都被放进去了，那么loop又进入下一轮**
6. **再把3poll出来，这个时候queue里面剩下5，7，然后我们把3的子node，也就是10，加入queue，这个时候queue里面就是5，7，10了**
7. **按照刚刚我说的步骤，我们可以看到，queue能够很好地保存node被放进去和poll出来的次序，所以可以不重复，不落下地找到一个BST最底层的node，而且是一层层地找下去的，这个是BFS的优点**

我们看到它的时间复杂度基本和之前的preorder，postorder，inorder的复杂度差不多，但是空间复杂度要大一些，因为需要用queue来保存每一次还没有被print out出来的那些node

1. **BFS的复杂度**

**时间复杂度 O(N) ： N 为二叉树的节点数量，即 BFS 需循环 N 次。**

**空间复杂度 O(N)： 最差情况下，即当树为平衡二叉树时，最多有 N/2 个树节点同时在 queue 中，使用 O(N)大小的额外空间。**

1. **BFS的优点**
2. 可以找到tree的maximum depth，也可以找到最底层的node
3. 可以记录每层的node，也就是说在使用BFS的时候，我们是可以知道tree的每一层是哪些node，而且还可以知道层数，所以假设让我们去搜索在tree里面特定node的位置，我们使用BFS可以很快地找出来，关于这一点，如何控制记录层数，是个小拓展，我们可以记录每次poll出来的node个数和每次被加入的queue的node的个数，当poll出来的等于被加入的个数后，那么说明上一层我们已经遍历完了

代码示例：

public List<List<Integer>> levelOrder(TreeNode root) {

List<List<Integer>> input = new ArrayList<>();

if(root == null){

return input;

}

// to store the nodes that have not been iterated

Queue<TreeNode> store = new LinkedList<>();

store.add(root);

//this is to record the number of polled out nodes

int currentlevel = 1;

// this is to record the number of added nodes

int nextlevel = 0;

List<Integer> temp = new ArrayList<>();

while(!store.isEmpty()){

TreeNode current = store.poll();

temp.add(current.val);

currentlevel--;

if(current.left != null){

store.add(current.left);

nextlevel++;

}

if(current.right != null){

store.add(current.right);

nextlevel++;

}

//determine whether to go to the next level

if(currentlevel == 0){

currentlevel = nextlevel;

nextlevel = 0;

input.add(temp);

temp = new ArrayList<>();

}

}

return input;

}

1. **算法example**

**根据我们课上内容改编，课上我们提到了关于最大深度，但是现在最小深度，最小深度就是从root开始直到某个node，满足这个node不存在left child和right child而且到root的距离最短**

思考：

我们需要满足的条件是：

1. 我们需要找到的那个node节点的定义是左孩子和右孩子都为 null
2. 当某个node 节点左右孩子都为空时，返回 1
3. 当某个node 节点左右孩子有一个为空时，返回不为空的孩子节点的深度
4. 当某个node 节点左右孩子都不为空时，返回左右孩子较小深度的节点值

**方法1： 我们可以使用上面说的BFS，一个个node往下找，直到找到某个node不存在能被add到queue中的那个node，直接停止loop并且return**

**方法2： 我们可以使用recursion，类似于找到maxLevel，把MaxLevel改编一下，我们可以得到如下代码：**

public int minDepth(TreeNode root) {

if(root == null) return 0;

//这道题递归条件里分为三种情况

//1.左孩子和有孩子都为空的情况，说明到达了叶子节点，直接返回1即可

if(root.left == null && root.right == null) return 1;

//2.如果左孩子和由孩子其中一个为空，那么需要返回比较大的那个孩子的深度

int m1 = minDepth(root.left);

int m2 = minDepth(root.right);

//这里其中一个节点为空，说明m1和m2有一个必然为0，所以可以返回m1 + m2 + 1;

if(root.left == null || root.right == null) return m1 + m2 + 1;

//3.最后一种情况，也就是左右孩子都不为空，返回最小深度+1即可

return Math.min(m1,m2) + 1;

}

### Java Generics：

1. **什么是Generics（范型）：**

在编程语言中，范型一般指的是某一个数据结构，class等可以支持任何类型的参数

1. **Java的Generics**

java的generics是假的generics，因为java的generics其实仅仅是把所有的type变成了Object（java最顶层的type可以说，但是不是基本类型），并没有真正做到支持所有类型

我们知道type转换是需要比如Integer.parseInt(里面填string）这样的，所以其实在java里只是不断地进行type的转换让它变成Object

同时，使用generics，我们其实并没有真正地使用比如基本类型，而是使用了Integer，Short，Character等封装的class（注意，Integer和int不是同一个类型哦，Integer是可以转换成Object的，因为Object是Integer的父类，但是int类型和Object直接是无法转换的，所以Generics是支持所有Object的子类的那些基本累=类型的变体加上String或者我们自己定义的一些class）

在ArrayList里面我们一般的写法是ArrayList<Integer> res = new ArrayList<>()就是这个道理，其实我们这里给arraylist加入的并不是int类型，而是Integer类型哦！！！

1. **为什么要使用范型**

使用泛型的意义在于：

1. 适用于多种数据类型执行相同的代码（代码复用）
2. 泛型中的类型在使用时指定，不需要强制类型转换（类型安全，编译器会检查类型）

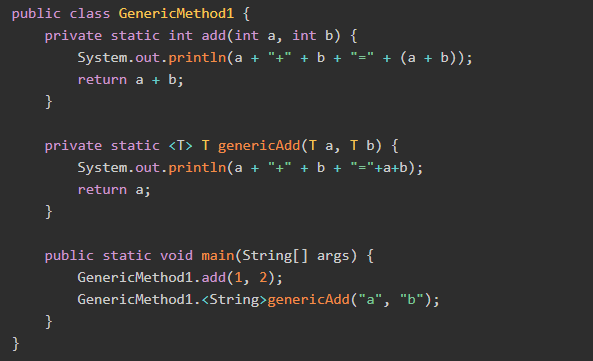
假设我们定义的arraylist不使用范型，那么他是可以加入不同的类型的元素的，但是如果我们再想要转换type变成String，那么就会报错，如下图：



1. **建立Generic的class**

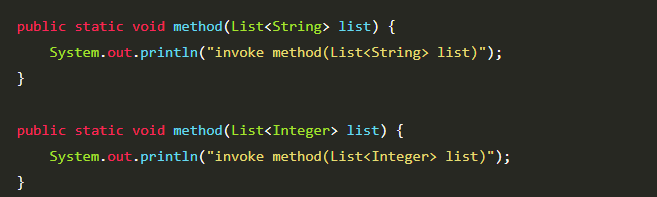
我们可以定义一个generics类，注意在class那边名称后需要加上<T>,这里可以是各种类型都行，但是如果你写的是<String>那就意味着这个class只能支持string类型的参数，相反，如果写的是<T>或者<E>其他等等，那都是可以支持任何类型的，有了范型，我们就可支持写一个add的method，可以同时add两个String或者add两个Integer，这个能很好的减少我们代码量，本来我们是需要add的overload的

对于method，我们的有两种，第一种是return一个值，那么我们只要把method的return type改成 T就行了，如果是void，我们需要在void前面加上<T>表明这个是一个能够支持generics的method



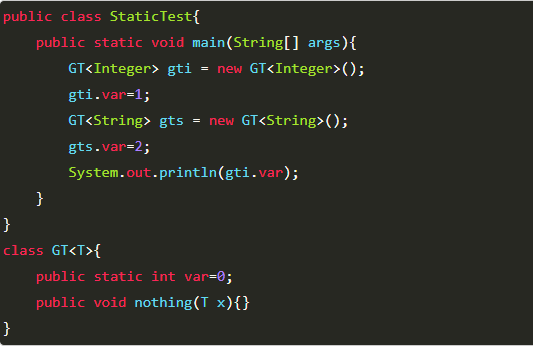
1. **Java的Generics存在的问题**

问题一：类型擦除后都变成一个Object



如果我们使用overload，大家可以看到，两个parameter分别是List<String>，一个是List<Integer>，两个是完全不同的，但是因为java的generics其实是把类型都转换成了Object，所以在java进行编译的时候，会发生两个都变成了单纯的List

问题二：static variable被共享了



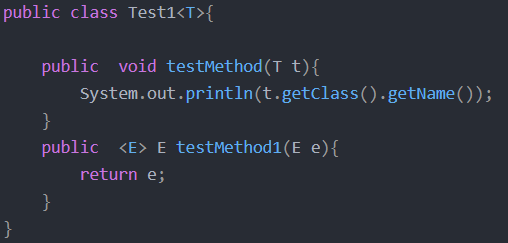
我们建立了一个generics的class，然后我们定义了两个这个class的对象，因为本身这个var是一个static variable，所以在经过java的编译后，因为类型的擦除都变成object，导致gts和gti都会关联到一个字节码上，也就是gti和gts其实是一个东西，所以这个时候print out出来gti，我们会发现它和gts的var的一样的结果，都是2

因为这个原因，所以要提示大家，如果想要使用generics的class，尽量不要设置static variable哦！！！，不然每个对象会共享这个static variable的

1. **Generic class和Generic method同时出现**

因为范型的类如果定义了比如说<T>，那么着意味着下面的method都是支持<T>的，

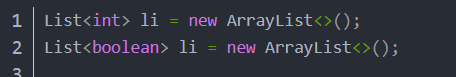
如果我们想要去return一个和我们当前的class不同的一种类型，我们需要在一个class里的method单独去定义它的generics



可以看到这样，就可以同时支持两种generics的

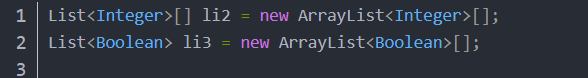
1. **java不能支持的几种generics的使用**

**类型1：基本类型和Object封装的基本类型乱用**



基本类型是不能放入generics的，我们需要Integer，Character，Boolean， Short！！

**类型2：建立了generic的array**



java不能建带有范型的array，因为类型会被擦除，所以在编译器编译时，就会发生array的type变成了List而非上面的List<Boolean>和List<Integer>，也就是说array其实无法识别自己到底在存什么type，所以不能使用