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
```

2. 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:

```
/**
 *       1
 *      / \
 *     2   3
 *    / \   \
 *   5   7  10
 */
```

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

3. **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。

4. **BFS 的优点**
   1. 可以找到 tree 的 maximum depth，也可以找到最底层的 node
   2. 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;
    }
```

5. **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

2. **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！！！

3. **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:

```
List arrayList = new ArrayList();
arrayList.add("aaaa");
arrayList.add(100);

for(int i = 0; i< arrayList.size();i++){
    String item = (String)arrayList.get(i);
    Log.d("泛型测试","item = " + item);
}
```

4. **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

```
public class GenericMethod1 {
    private static int add(int a, int b) {
        System.out.println(a + "+" + b + "=" + (a + b));
        return a + b;
    }

    private static <T> T genericAdd(T a, T b) {
        System.out.println(a + "+" + b + "="+a+b);
        return a;
    }

    public static void main(String[] args) {
        GenericMethod1.add(1, 2);
        GenericMethod1.<String>genericAdd("a", "b");
    }
}
```

5. Java 的 Generics 存在的问题

问题一：类型擦除后都变成一个 Object

```
public static void method(List<String> list) {
    System.out.println("invoke method(List<String> list)");
}

public static void method(List<Integer> list) {
    System.out.println("invoke method(List<Integer> list)");
}
```

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

```java
public class StaticTest{
    public static void main(String[] args){
        GT<Integer> gti = new GT<Integer>();
        gti.var=1;
        GT<String> gts = new GT<String>();
        gts.var=2;
        System.out.println(gti.var);
    }
}
class GT<T>{
    public static int var=0;
    public void nothing(T x){}
}
```

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

6. 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

```java
public class Test1<T>{

    public  void testMethod(T t){
        System.out.println(t.getClass().getName());
    }
    public  <E> E testMethod1(E e){
        return e;
    }
}
```

可以看到这样，就可以同时支持两种 generics 的

7. 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

```
List<int> li = new ArrayList<>();
List<boolean> li = new ArrayList<>();
```

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

```
List<Integer>[] li2 = new ArrayList<Integer>[];
List<Boolean> li3 = new ArrayList<Boolean>[];
```

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