**private void** createNode(Node root, BulkloadContext bulkloadContext, SelectVpStrategy selectVpStrategy) {  
 Deque<Node> nodeStack = **new** LinkedList<>();  
 IntStack offsetStack = **new** IntStack(64), lengthStack = **new** IntStack(64);  
 //迭代使用的偏移量栈和长度  
 nodeStack.push(root);  
 offsetStack.push(0);  
 lengthStack.push(positions.length);  
 //初始状态  
 **float**[] distanceBuffer = **new float**[bulkloadContext.total];  
 Node currentNode = **null**;  
 **int** currentOffset, currentLength;  
 **int** fanout = configuration.getFanout();  
 SelectVpResult selectVpResult = **new** SelectVpResult();  
 **while** (!nodeStack.isEmpty()) {  
 currentNode = nodeStack.pop();  
 currentOffset = offsetStack.pop();  
 currentLength = lengthStack.pop();  
 **if** (currentLength > configuration.getEntrySize()) {  
 currentNode.initAsNonLeaf(configuration);  
 selectVpStrategy.selectVp();//选取优先点  
 ………//计算各个数据点与优先点的距离并

--currentLength;//数据点总量减1  
 **if** (currentLength <= fanout) {

currentNode.childrenBounds.add(distanceBuffer[1]);  
 ……..//初始化为单个非叶节点并且入栈

} **else** {  
**int** childSize = (**int**) Math.ceil(currentLength \* 1.0 / fanout);

//计算每个子树应有的数据点量  
 **for** (**int** i = 0, start = 1, end; i < fanout; i++) {  
 end = Math.min(start + childSize - 1, currentLength);  
 **if** (end < start) {  
 **break**;  
 }  
 currentNode.childrenBounds.add(distanceBuffer[start]);  
 currentNode.childrenBounds.add(distanceBuffer[end]);

currentNode.distances[i]= distanceBuffer[end];

currentNode.childrenNodes[i]=**new** Node(nextNodeId(), **false**)  
 nodeStack.push(currentNode.childrenNodes[i]);

offsetStack.push(currentOffset + start);  
 lengthStack.push(end - start + 1);  
 start = end + 1;

//设置子树指针并分别为每路子树，设置最大距离值，距离上下界值  
 }  
 }  
 } **else** {  
 currentNode.initAsLeaf(currentLength);  
 **for** (**int** i = 0; i < currentLength; i++) {  
 currentNode.children.add(bulkloadContext.ids.get(currentOffset+ i]);  
 }//初始化叶节点，结束一个分支  
 }  
 }  
}