Positions' structure of the corporation is similar to a tree structure. Here only one Immediate supervisor available for every employee. The root node is considered as the president of the corporation. Each level represents the hierarchy level of the employee and upper levels indicate the higher ranks. The invitation of the president may not be necessary.

Solution:

For the solution we use dynamic approach.

```
maximumConvivality(Node):
If Node.Children = null
Return Node.rating
Else
    maxRatingChildren = 0
    For C in Node.Children
        maxRatingChildren =maxRatingChildren +maximumConvivality(C)

    maxRatingGrandChildren = Node.rating

    For G in Node.GrandChildren
        maxRatingGrandChildren=maxRatingGrandChildren+maximumConvivality(G)
    If maxRatingGrandChildren >= maxRatingChildren
        Return maxRatingGrandChildren
    Else
        Return maxRatingChildren
```

In dynamic approach we always try to reach the problem using bottom up method. Recursively we calculate take into consideration of conviviality up to certain node. When choosing nodes those which will be suitable for the invitation, can be chosen from maxRatingGrandChildren >= maxRatingChildren inequality. This will decide the suitability of the invitation. If this is true we consider as it is suitable for invite.When creating the invitee list top down approach will be use. If node is added to the invitee list then the children will not be added to invitee list and transverse to the next node.

Time Complexity

K- constant C- child G- grandchild N- no. of nodes X- no. of children

```
double getRatingSumOfChildren(Node parent) {
    Node currentNode = parent.leftChild;
    double ratingSum = 0;
    while (currentNode != null) {
       ratingSum += currentNode.rating;
       currentNode = currentNode.rightSibling;
    }
    return ratingSum;

  }
```

Time complexity 3k + 2k * C =O(c)

```
double getRatingSumOfGrandChildren(Node parent) {
    double ratingSum = 0;
    Node child = parent.leftChild;
    while (child != null) {
       Node grandChild = child.leftChild;
       while (grandChild != null) {
          ratingSum += grandChild.rating;
          grandChild = grandChild.rightSibling;
```

```
        }
        child = child.rightSibling;
      }
      return ratingSum;

  }
```

Time complexity = 3k + Cx (2k + (G x 2k)) = O(C * G)

```
List<String> solve(Tree tree) throws InterruptedException {
      //preprosessing
      Stack<Node> stack = new Stack<Node>();// #to acesses the tree from bottom to top
      Queue<Node> queue = new Queue<>();
      queue.enqueue(tree.root);
      stack.push(tree.root);
      Node currentNode = tree.root;
      while (!queue.isEmpty()) {
        currentNode = queue.dequeue();
        if (currentNode != null) {
          currentNode = currentNode.leftChild;
          if (currentNode != null) {
            queue.enqueue(currentNode);
            stack.push(currentNode);
            while (currentNode.rightSibling != null) {
              currentNode = currentNode.rightSibling;
              queue.enqueue(currentNode);
```

```
                stack.push(currentNode);
            }
          }
        }
      }
      while (!stack.isEmpty()) {//calculate maximum possible rating for each sub tree intiating from Node
        currentNode = stack.pop();
        double childrenRatingSum = getRatingSumOfChildren(currentNode);
        double grandChildrenRatingSum = getRatingSumOfGrandChildren(currentNode);
        if ((currentNode.rating + grandChildrenRatingSum) >= childrenRatingSum) {
          currentNode.invite = true;
          currentNode.rating += grandChildrenRatingSum;
        } else {
          currentNode.invite = false;
          currentNode.rating = childrenRatingSum;
        }
      }
```

Time complexity= 5k +N * (6k + Hx(3k)) + Nx(7k + O(X) + O(H * G))

= O(N * X * N * X * G)

= O(N * X * G)

```
      //#Create invite list
      List<String> inviteList = new ArrayList<String>();
      Queue<Node> queue1 = new Queue<>();
      queue1.enqueue(tree.root);
      currentNode = tree.root;
```

```
if (currentNode.invite) {
   inviteList.add(currentNode.name);
   Node childNode = currentNode.leftChild;
   while (childNode != null) {
      childNode.invite = false;
      childNode = childNode.rightSibling;
   }
}
while (!queue1.isEmpty()) {
   currentNode = queue1.dequeue();
   if (currentNode != null) {
      currentNode = currentNode.leftChild;
      if (currentNode != null) {
         queue1.enqueue(currentNode);
         if (currentNode.invite) {
            inviteList.add(currentNode.name);
            Node childNode = currentNode.leftChild;
            while (childNode != null) {
               childNode.invite = false;
               childNode = childNode.rightSibling;
            }
         }
         while (currentNode.rightSibling != null) {
            currentNode = currentNode.rightSibling;
            queue1.enqueue(currentNode);
            if (currentNode.invite) {
               inviteList.add(currentNode.name);
               Node childNode = currentNode.leftChild;
               while (childNode != null) {
```

```
                    childNode.invite = false;
                    childNode = childNode.rightSibling;
                }
            }
        }
    }
}
    return inviteList;

}
}
```

Time complexity= 8k + X* 2k + N * (8k + X * 2k +X * (5k + X * 2k))

=O(X + N*(X + (X*X)))

= O(N*X*X)

Total time complexity= O(N * X * X) + O(N * X * G)