# 上 海 交 通 大 学 试 卷（A 卷）

（ 2020 至 2021 学年 第一学期 ）

班级号_____ 学号_____ 姓名 _____

课程名称 _____数据结构_____ 成绩 _____

## 一、 选择题（每题 1 分，共 15 分）

1. 顺序存储的线性表，其长度为 n。假设在任何位置上插入或删除操作都是等概率的。插入一个元素时平均要移动表中元素个数为： A

   A. n/2 B. (n+1)/2 C. (n-1)/2 D. n

2. 带头结点 head 的单链表为空表的判定条件是： B

   A. head==null B. head->next==null C. head->next==head D. head!=null

3. 若某线性表最常用的操作是读取第 i 个元素和第 i 个元素的前趋元素，则采用下面哪种存储方式最节省运算时间？ B

   A. 单链表 B. 顺序表 C.双链表 D.单循环链表

4. 在操作系统内部，函数调用是用下面哪种数据结构来实现的？ C

   A. 线性表 B. 队列 C. 栈 D. 树

5. 从空栈开始依次将字符A、B、C、D、E入栈，在所有可能的出栈序列中，最后一个出栈元素是C的序列的个数是： C

   A. 5 B. 1 C. 4 D. 3

6. 深度为 k 的满二叉树若按自上而下，从左到右的顺序给结点进行编号（从 1 开始），则编号最小的叶子结点编号是： A

   A. $2^{k-1}$ B. $2^{k-1}-1$ C. $2^{k-1}+1$ D. $2^{k}-1$

7. 下面哪种数据结构最适合用于创建一个优先级队列？ D

   A. 栈 B.双向链表 C.单向链表 D. 堆

8. 对于下列关键字序列，不可能构成某二叉排序树中一条查找路径的序列是： A

   A. 98,22,91,24,94,71 B. 92,18,90,34,86,35

   C. 23,89,77,29,36,38 D. 10,25,71,68,33,34

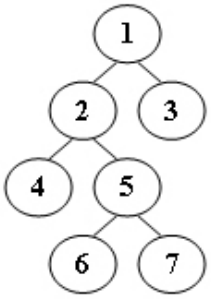
我承诺，我将严格遵守考试纪律。

承诺人：_____

| 题号 | 一 | 二 | 三 | 四 |
|---|---|---|---|---|
| 得分 | | | | |
| 批阅人(流水阅卷教师签名处) | | | | |

9. 给定二叉树如下图所示。设 N 代表二叉树的根，L 代表根结点的左子树，R 代表根结点的右子树。若遍历后的结点序列为 3、1、7、5、6、2、4，则其遍历方式是： D

A．NRL　　B.LRN　　C.RLN　　✔D. RNL




10. 现有一棵无重复关键字的 AVL 树，对其进行中序遍历可得到一个降序序列。下列关于该 AVL 树的叙述中，正确的是：D

A．根结点的度一定为 2　　B．树中最小元素一定是叶结点

C．最后插入的元素一定是叶结点　　D．树中最大元素一定是无左子树

rebalance

11. 用哈希（散列）方法处理冲突（碰撞）时可能出现堆积（聚集）现象，下列选项中， 会受堆积现象直接影响的是： D

A．存储效率　　B．散列函数　　C．装填(装载)因子　　D．平均查找长度

Clustering doesn't directly affect storage efficiency. It can lead to inefficient use of the hash table but doesn't directly affect how data is stored.

12. 稳定的排序方法是：B

A．直接插入排序和快速排序　　B．二分插入排序和冒泡排序

C．直接选择排序和四路归并排序　　D．堆排序和希尔排序

13. 置换-选择排序的作用是： C

A.置换-选择排序是完成将一个磁盘文件排序成有序文件的有效的外排序算法

B.置换-选择排序生成的初始归并段长度是内存工作区的 2 倍

C.置换-选择排序用于生成外排序的初始归并段

D.置换-选择排序是对外排序中输入/归并/输出的并行处理

14. 对于一棵 M 阶 B+树，下列哪个选项是正确的？ B

A.根节点一定有 2~M 个子节点　　B. 一个叶节点和一个非叶节点之间可以有相同的关键值

C.任意两个叶节点的深度不一定相同　　D.所有的非叶节点都有[M/2]~M 个子节点

root is 2 ~M

15. 无向图 G 有 22 条边，度为 5 的顶点有 3 个，度为 3 的顶点有 5 个，其余都是度为 2 的顶点，则图 G 最多有多少个顶点？ C

A.11　　B.12　　C.15　　D.16

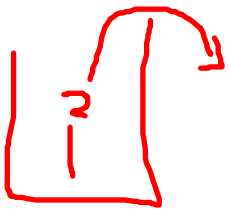
## 二、简答题（每题 5 分，共 40 分）

1. head 为某单链表头指针，请说明以下代码的功能，并做简单描述。

```
void SomeFunction1(node* head)
{
    if(head->next==NULL) return;
    node *p = head ->next->next;
    head ->next->next = NULL;
    while (p)
    {
        node *q= p->next;
        p->next = head->next;
        head ->next = p;
        p = q;
    }
}
```

答案：逆置单链表，从第二个数据结点开始，依次插入到 head 结点后面，完成逆序。

2. 从空栈开始依次将1,2,3,4,5入栈，判断2,4,5,3,1是否是一个合法的出栈序列？如果是，给出对应的 push/pop 操作顺序；如果不是，给出理由。



答案：

2,4,5,3,1是一个合法的出栈序列，

其对应的 push/pop 操作顺序如下：

push push pop push push pop push pop pop pop

3. 对于 n 个待编码的字符，可以采用规模为 2n 的数组来存储其哈夫曼树。请根据哈夫曼算法完善表 1 中哈夫曼树在内存中的表现（注意：无孩或无父用 0 表示），并在表 2 中列出每个字符的哈夫曼编码。

表 1：哈夫曼树

| 字符 | | | | | | | | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 权值 | | | | | | | | 0.18 | 0.15 | 0.14 | 0.05 | 0.12 | 0.19 | 0.17 |
| 父结点 | | | | | | | | | | | | | | |
| 左孩子 | | | | | | | | | | | | | | |
| 右孩子 | | | | | | | | | | | | | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

表 2：哈夫曼编码

| 字符 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 哈夫曼编码 | | | | | | | |

答案：

表1：哈夫曼树

| 字符 | | | | | | | | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 权值 | | 1 | 0.63 | 0.37 | 0.34 | 0.29 | 0.17 | 0.18 | 0.15 | 0.14 | 0.05 | 0.12 | 0.19 | 0.17 |
| 父结点 | | 0 | 1 | 1 | 2 | 2 | 4 | 3 | 5 | 5 | 6 | 6 | 3 | 4 |
| 左孩子 | | 2 | 4 | 7 | 6 | 8 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 右孩子 | | 3 | 5 | 12 | 13 | 9 | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

表2：哈夫曼编码（可以有其他情况，但各字符编码长度和本答案相同）

| 字符 | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 哈夫曼编码 | 10 | 010 | 011 | 0000 | 0001 | 11 | 001 |

4. 请给出一组元素（18，23，47，13，5，27，54）通过逐一插入法建立最大堆的执行过程

答案：（也可用图示方法）

18

18，23

23，18

23，18，47

47，18，23

47，18，23，13

47，18，23，13，5

47，18，23，13，5，27

47，18，27，13，5，23

47，18，27，13，5，23，54

47，18，54，13，5，23，27

54，18，47，13，5，23，27

5. 将关键字序列{8,10,13,3,6,7}依序插入一棵初始为空的 AVL 树，请画出各结点插入的结果。

答案：

| | | |
|---|---|---|
| 8<br>10 | 10<br>8 13 | 10<br>8 13<br>3 |
| 10<br>6 13<br>3 8 | 8<br>6 10<br>3 7 13 | |

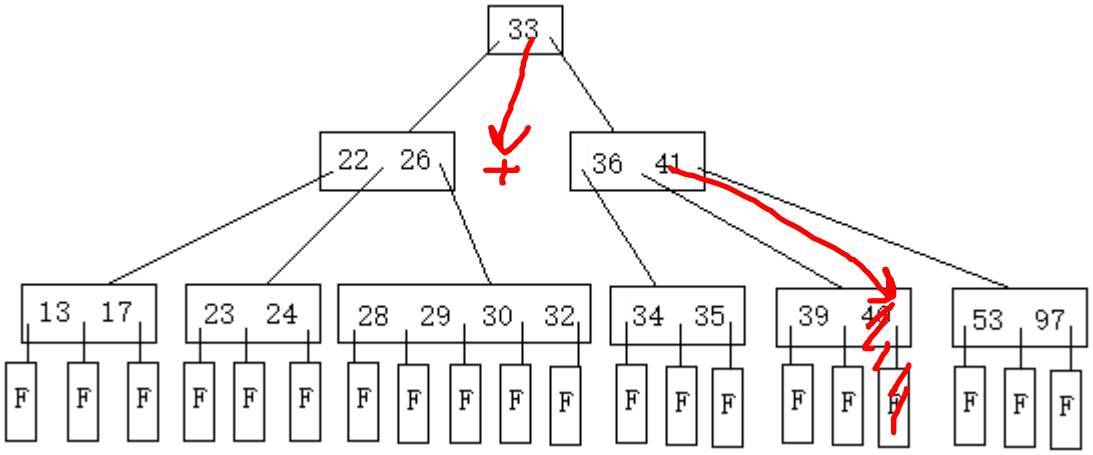
6. 外排序和内排序有什么不同？稳定排序和不稳定排序有什么不同？

答案：

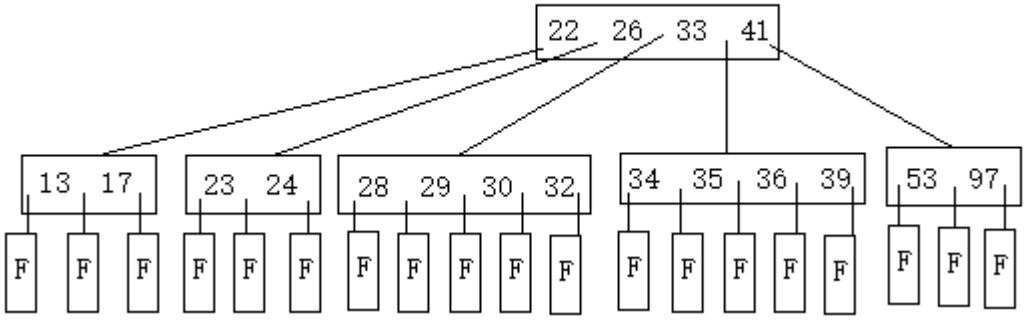
内排序中，所有的数据都在内存中，而外排序中，数据存储在外部存储器上。

具有相同关键字的元素在排序前后相对位置一定保持不变的是稳定排序，其相对位置不一定能保持不变的是不稳定排序。 relative position of elements with the same key must remain unchanged before and after sorting

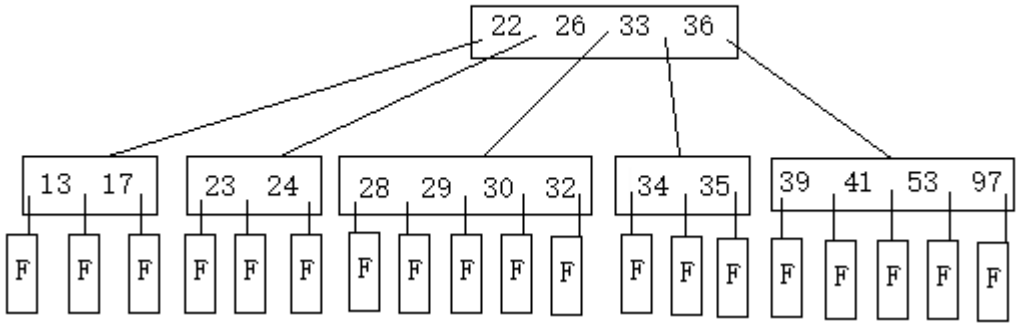
7. 画出对图中所示的 5 阶 B 树删除 40 后的 B 树



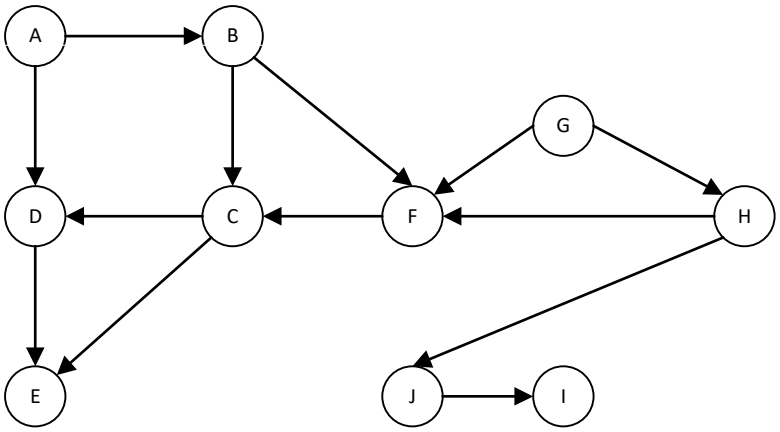

答案：




或者



8. 对于有向图，给出其一条拓扑排序序列。




答案： ABGHJIFCDE，或ABGHFCDEJI，或GHJIABFCDE，或GHABFCDEJI，或GHABFJICDE

三、程序填空（每空 2 分，共 20 分）

1. 完成如下以邻接表为存储结构的递归 DFS 算法

```
#define MAXSIZE 128 //最大节点数目
struct ArcNode //邻接表中存储边的节点类
{
    int adjvex;                 //所指向结点的位置
    struct ArcNode *nextArc; //下一条边的指针
};

struct VNode //顶点
{
    char data;              //顶点信息
    ArcNode *firstArc; //该顶点指向的第一条边的指针
};

struct Graph //图的定义
{
    VNode adjlist[MAXSIZE];//邻接表
    int n, e;//图的顶点数，边数
};

//以邻接表为储存结构的递归算法如下：
int visit[MAXSIZE];//全局变量  标记数组

void DFS(Graph *G,int v) //从结点 V 开始的遍历
{
    ArcNode *p;
    visit[v] = 1;       //置已访问标记
    cout<<v<<endl;              //访问节点
```

```
    p =_____;     //指向顶点 v 的第一条边
    while(p!=NULL)
    {
        if(visit[p->adjvex]==0)     //若未访问
        {
            _____
        };
        _____                    //继续访问下一条边
    }
}
```

**答案：**

1. **G->adjlist[v].firstArc**
2. **DFS(G,p->adjvex);**
3. **p = p->nextarc;**

2. 按如下代码进行冒泡排序，第 k 趟冒泡后，第 k 大的元素会排在倒数第 k 的位置上：

```
public static void bubblesort(int[] a) {
   for (int i = 1; i < a.length; i++) {
      boolean is_sorted = true;

      for (int j = 0; j < a.length - i; j++) {
         if (a[j] > a[j+1]) {
            int temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
            is_sorted = false;
         }
      }

      if(is_sorted) return;
   }
}
```

我们对上面的代码进一步优化：记录下最后一次交换的位置（lastSwap），在该序号后的位置上，数组已经排序好，故可避免对数组靠近尾部已经排序好的元素做不必要的比较。请填充以下代码完成上面的功能：

```
public static void bubblesort(int[] a) {
   int lastSwap = _____;
   for (int i = 1; i< a.length; i++) {
      boolean is_sorted = true;
      int currentSwap = -1;

      for (int j = 0; j <_____; j++) {
         if (a[j] > a[j+1]) {
            int temp = a[j];
```

```
            a[j] = a[j+1];
            a[j+1] = temp;
            is_sorted = false;
            _____;
        }
    }

    if (is_sorted) return;
    _____;
  }
}
```

答案：

```
  int lastSwap = a.length - 1;
  lastSwap
  currentSwap = j;
  lastSwap = currentSwap;
```

3. 下面程序实现二叉查找树的查找，空格处应该填入的语句是 （）

```
struct Node {
  int value;
  Node* left;
  Node* right;
};
Node* find(int value, Node* root) {
  if (root == NULL || value == root->value)  _____

  if (value < root->value)_____
  else  _____
}
```

答案：

```
return root;
return find(value, root->left)
return find(value, root->right)
```

四、编程题（共 25 分）

1.假设线性表采用顺序存储结构，试实现函数（int DelRepeat()），用以删除所有重复元素，并返回删除元素的个数。要求算法的时间复杂度为 O(n)。

线性表的定义如下：

```
template <class elemType>
class seqList:public list<elemType>
{
    private:
        elemType* data;
        int currentLength;
        elemType *noData; //一个结构中不存在的元素值
        ……
    public:
        int DelRepeat(); //删除所有重复元素
        ……
}（15 分）
```

答案：

1. 采用哈希方法，时间复杂度为 O(n) ---15 分

```
template <class elemType>
int DelRepeat()
{   int i, j, pos;
    bool *status;
    elemType *tmp;
    int oldLen;
    const elemType noData=-1;

    status = new bool[currentLength];
    tmp = new elemType[2*currentLength];
    for (i=0; i<2*currentLength; i++)
        tmp[i] = noData;

    for (i=0; i<currentLength; i++)
    {
        pos=data[i]%(2*currentLength);
        while (tmp[pos]!=noData && tmp[pos]!=data[i]) pos++;

        if (tmp[pos]==noData)
        {
            status[i] = true;
            tmp[pos] = data[i];
        }
        else status[i] = false;
    }

    j=0;
```

```
    for (i=0; i<currentLength; i++)
        if (status[i]) tmp[j++]=data[i];

    for (i=0; i<j; i++)
        data[i] = tmp[i];

    oldLen = currentLength;
    currentLength = j;
    return oldLen-currentLength;
}
```

如果大于 O(n)，满分 10 分

2. 时间复杂度最好为 O(n)，最差为 $O(n^2)$。---10 分

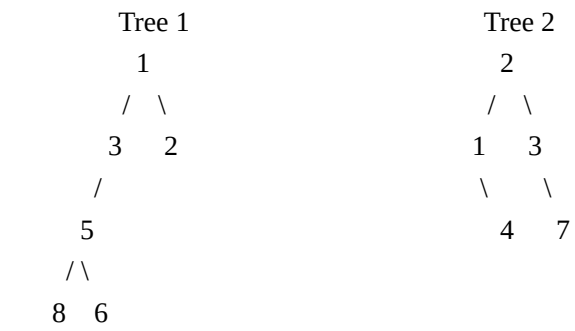
```
template <class elemType>
int DelRepeat()
{   int i, j, k;
    k=0;

    int oldLen = currentLength;
    for (i=0; i<currentLength; i++) //  从第 0 个元素开始检查
    {
        for (j=0; j<k; j++)          //检查 i 元素是否与前面 k 个元素重复
          if (data[i]==data[j]) break;
        if (j==k)
            data[k++]=data[i];   //将第 i 元素作为第 k 个不重复元素加入
    }
    currentLength = k;
    return oldLen-currentLength;
}
```

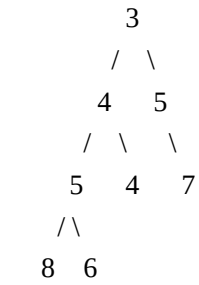
2. 给定两个二叉树，想象当你将它们中的一个覆盖到另一个上时，两个二叉树的一些节点便会重叠。请完善下列算法，实现将上述两个二叉树合并为一个新的二叉树。合并的规则是如果两个节点重叠，那么将他们的值相加作为节点合并后的新值，否则不为 NULL 的节点将直接作为新二叉树的节点。注意：合并后的二叉树中结点允许直接使用这两个二叉树上的结点。

示例 1:

输入:



输出:
合并后的树:



**注意:** 合并必须从两个树的根节点开始。

```
/*** Definition for a binary tree node.
struct TreeNode {
    int val;
     TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 };
class Solution {
public:
    TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2); //请实现合并函数
};
```

(10 分)

答案：一个空时的处理 3 分，两个不空时的处理 4 分，正确返回 3 分

方法一：采用先序遍历即深度优先搜索

可以使用深度优先搜索合并两个二叉树。从根节点开始同时遍历两个二叉树，并将对应的节点进行合并。

两个二叉树的对应节点可能存在以下三种情况，对于每种情况使用不同的合并方式。

（1）如果两个二叉树的对应节点都为空，则合并后的二叉树的对应节点也为空；

（2）如果两个二叉树的对应节点只有一个为空，则合并后的二叉树的对应节点为其中的非空节点；

（3）如果两个二叉树的对应节点都不为空，则合并后的二叉树的对应节点的值为两个二叉树的对应节点的值之和，此时需要显性合并两个节点。

对一个节点进行合并之后，还要对该节点的左右子树分别进行合并。这是一个递归的过程。

```
class Solution {
public:
TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2) {
        if (t1 == nullptr) {
            return t2;
        }
        if (t2 == nullptr) {
            return t1;
        }
        auto merged = new TreeNode(t1->val + t2->val);
        merged->left = mergeTrees(t1->left, t2->left);
        merged->right = mergeTrees(t1->right, t2->right);
        return merged;
```

```
    }
};
```

方法二：采用层次遍历即广度优先搜索

可以使用广度优先搜索合并两个二叉树。首先判断两个二叉树是否为空，如果两个二叉树都为空，则合并后的二叉树也为空，如果只有一个二叉树为空，则合并后的二叉树为另一个非空的二叉树。

如果两个二叉树都不为空，则首先计算合并后的根节点的值，然后从合并后的二叉树与两个原始二叉树的根节点开始广度优先搜索，从根节点开始同时遍历每个二叉树，并将对应的节点进行合并。

使用三个队列分别存储合并后的二叉树的节点以及两个原始二叉树的节点。初始时将每个二叉树的根节点分别加入相应的队列。每次从每个队列中取出一个节点，判断两个原始二叉树的节点的左右子节点是否为空。如果两个原始二叉树的当前节点中至少有一个节点的左子节点不为空，则合并后的二叉树的对应节点的左子节点也不为空。对于右子节点同理。

如果合并后的二叉树的左子节点不为空，则需要根据两个原始二叉树的左子节点计算合并后的二叉树的左子节点以及整个左子树。考虑以下两种情况：

（1）如果两个原始二叉树的左子节点都不为空，则合并后的二叉树的左子节点的值为两个原始二叉树的左子节点的值之和，在创建合并后的二叉树的左子节点之后，将每个二叉树中的左子节点都加入相应的队列；

（2）如果两个原始二叉树的左子节点有一个为空，即有一个原始二叉树的左子树为空，则合并后的二叉树的左子树即为另一个原始二叉树的左子树，此时也不需要对非空左子树继续遍历，因此不需要将左子节点加入队列。

对于右子节点和右子树，处理方法与左子节点和左子树相同

```
class Solution {
public:
    TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2) {
        if (t1 == null) {
            return t2;
        }
        if (t2 == null) {
            return t1;
        }
        auto merged = new TreeNode(t1->val + t2->val);
        auto q = queue<TreeNode*>();
        auto queue1 = queue<TreeNode*>();
        auto queue2 = queue<TreeNode*>();
        q.push(merged);
        queue1.push(t1);
        queue2.push(t2);
        while (!queue1.empty() && !queue2.empty()) {
            auto node = q.front(), node1 = queue1.front(), node2 = queue2.front();
            q.pop();
            queue1.pop();
            queue2.pop();
            auto left1 = node1->left, left2 = node2->left, right1 = node1->right, right2 = node2->right;
            if (left1 != null || left2 != null) {
                if (left1 != null && left2 != null) {
```

```
                    auto left = new TreeNode(left1->val + left2->val);
                    node->left = left;
                    q.push(left);
                    queue1.push(left1);
                    queue2.push(left2);
                } else if (left1 != nullptr) {
                    node->left = left1;
                } else if (left2 != nullptr) {
                    node->left = left2;
                }
            }
            if (right1 != nullptr || right2 != nullptr) {
                if (right1 != nullptr && right2 != nullptr) {
                    auto right = new TreeNode(right1->val + right2->val);
                    node->right = right;
                    q.push(right);
                    queue1.push(right1);
                    queue2.push(right2);
                } else if (right1 != nullptr) {
                    node->right = right1;
                } else {
                    node->right = right2;
                }
            }
        }
        return merged;
    }
};
```