数据结构编程题目整理

1. 判断是否为完全二叉树 (CBT)

逻辑是利用**层序遍历**。一旦在遍历过程中遇到了空节点，那么之后出现的所有节点都必须是空，否则就不是完全二叉树。

```cpp
#include <queue>
using namespace std;

struct TreeNode {
    int val;
    TreeNode *left, *right;
};

bool isCBT(TreeNode* root) {
    if (root == nullptr) return true;

    queue<TreeNode*> q;
    q.push(root);
    bool reachNull = false; // 标记是否已经遇到过空节点

    while (!q.empty()) {
        TreeNode* curr = q.front();
        q.pop();

        if (curr == nullptr) {
            reachNull = true;
        } else {
            // 如果之前已经遇到过空节点，但现在又出现了非空节点，说明不连续
            if (reachNull) return false;

            q.push(curr->left);
            q.push(curr->right);
        }
    }
    return true;
}
```

---

2. 判断满二叉树 (Full Binary Tree)

满二叉树的定义逻辑是：每个节点要么没有子节点，要么有两个子节点。

```cpp
bool isFull(TreeNode* root) {
    if (root == nullptr) return true;

    // 如果只有一个孩子，则不是满二叉树
    if ((root->left == nullptr && root->right != nullptr) ||
        (root->left != nullptr && root->right == nullptr)) {
        return false;
    }

    // 递归检查左右子树
    return isFull(root->left) && isFull(root->right);
}
```

---

3. 二叉树的最小深度 (Min Depth)

展示了两种主流解法：**DFS（递归）**和 **BFS（层序遍历）**。

A. 递归法 (DFS)

注意：最小深度是从根节点到最近**叶子节点**的路径。如果一个节点只有左子树或只有右子树，不能直接取 `min`，否则会误把空的一侧当成深度 0。

```cpp
int minDepth(TreeNode* root) {
    if (root == nullptr) return 0;

    // 1. 如果左子树为空，递归右子树
    if (root->left == nullptr) return 1 + minDepth(root->right);
    // 2. 如果右子树为空，递归左子树
    if (root->right == nullptr) return 1 + minDepth(root->left);

    // 3. 左右都不为空，取最小值
    return 1 + min(minDepth(root->left), minDepth(root->right));
}
```

B. 层序遍历法 (BFS) - 效率最高

BFS 寻找最小深度的优势在于：一旦遇到第一个叶子节点，当前深度即为最小深度，无需遍历整棵树。

```cpp
int minDepthBFS(TreeNode* root) {
    if (root == nullptr) return 0;

    queue<TreeNode*> q;
    q.push(root);
    int depth = 1;

    while (!q.empty()) {
        int size = q.size();
        for (int i = 0; i < size; i++) {
            TreeNode* curr = q.front();
            q.pop();

            // 发现第一个叶子节点，立即返回
            if (curr->left == nullptr && curr->right == nullptr) {
                return depth;
            }

```

```
            if (curr->left) q.push(curr->left);
            if (curr->right) q.push(curr->right);
        }
        depth++;
    }
    return depth;
}
```

1. 图的遍历 (Graph Traversal)

基于邻接表或邻接矩阵的通用遍历算法。

深度优先搜索 (DFS)

DFS 使用递归思想，尽可能深地探索分支。

```
void DFS(Graph* G, int v) {
    // 标记当前顶点为已访问
    G->setMark(v, VISITED);
    printf("%d ", v);

    // 遍历所有相邻节点
    for (int w = G->first(v); w < G->n(); w = G->next(v, w)) {
        if (G->getMark(w) == UNVISITED) {
            DFS(G, w);
        }
    }
}
```

广度优先搜索 (BFS)

BFS 使用队列实现，按层级由近及远访问节点。

```
void BFS(Graph* G, int start) {
    queue<int> Q;
    Q.push(start);
    G->setMark(start, VISITED);
    printf("%d ", start);

    while (!Q.empty()) {
        int v = Q.front();
        Q.pop();

        for (int w = G->first(v); w < G->n(); w = G->next(v, w)) {
            if (G->getMark(w) == UNVISITED) {
                G->setMark(w, VISITED);
                printf("%d ", w);
                Q.push(w);
            }
        }
    }
}
```

2. 链表插入排序 (Insertion Sort for List)

逻辑是将无序节点一个个插入到前面已经排好序的序列中。

```
ListNode* insertionSortList(ListNode* head) {
    if (head == nullptr || head->next == nullptr) return head;

    // lastSorted 为已排序部分的最后一个节点
    ListNode* lastSorted = head;
    // curr 为当前待排序的节点
    ListNode* curr = head->next;

    while (curr != nullptr) {
        if (lastSorted->val <= curr->val) {
            // 情况1：当前值比前面都大，直接后移
            lastSorted = lastSorted->next;
        } else {
            // 情况2：需要从头开始找插入位置
            ListNode* prev = head;
            // 如果 head 节点就比 curr 大，代码逻辑需要处理 dummyNode 更好
            // 笔记中的逻辑是找到插入位置的前驱节点 prev
            while (prev->next->val < curr->val) {
                prev = prev->next;
            }
            // 进行节点搬运（删除并重新插入）
            lastSorted->next = curr->next;
            curr->next = prev->next;
            prev->next = curr;
        }
        curr = lastSorted->next; // 移动到下一个待处理节点
    }
    return head;
}
```

---

3. 寻找二叉搜索树中第二大的值

利用了 **中序遍历（In-order Traversal）**。在 BST 中，中序遍历得到的是升序序列。

```
// 辅助函数：中序遍历并将结果压入栈
void inOrder(TreeNode* root, stack<int>& s) {
    if (root == nullptr) return;
    inOrder(root->left, s);
    s.push(root->val);
    inOrder(root->right, s);
}

// 寻找第二大的值
int findSecondLarge(TreeNode* root) {
    stack<int> s;
    inOrder(root, s);

    if (s.size() < 2) return -1; // 不存在第二大

    s.pop();            // 弹出最大的值
    return s.top();     // 现在的栈顶就是第二大的值
}
```

1. 经典排序算法 (Sorting Algorithms)

冒泡排序 (Bubble Sort)

```
for (int i = 0; i < n - 1; i++) {
    for (int j = n - 1; j > i; j--) {
        if (A[j] < A[j - 1]) swap(A, j, j - 1);
    }
}
```

选择排序 (Selection Sort)

```
for (int i = 0; i < n - 1; i++) {
    int lowindex = i;
    for (int j = n - 1; j > i; j--) {
        if (A[j] < A[lowindex]) lowindex = j;
    }
    swap(A, i, lowindex);
}
```

希尔排序 (Shell Sort) 核心逻辑

```
// 增量 d = n/2, 逐渐减小至 1
for (int i = d; i < n; i++) {
    for (int j = i; j >= d && A[j] < A[j - d]; j -= d) {
        swap(A, j, j - d);
    }
}
```

---

2. 二叉树 (BT) 与 普通树 (NBT) 属性计算

统计叶子节点 (Count Leaf)

```
// 二叉树 (BT)
int countLeaf(TreeNode* root) {
    if (root == nullptr) return 0;
    if (root->left == nullptr && root->right == nullptr) return 1;
    return countLeaf(root->left) + countLeaf(root->right);
}

// 普通树 (NBT)
int countLeaf(GNode* root) {
    if (root == nullptr) return 0;
    if (root->left == nullptr) return 1 + countLeaf(root->right);
    return countLeaf(root->left) + countLeaf(root->right);
}
```

计算树的高度 (Depth)

```
// 二叉树 (BT)
int depth(GNode* root) {
    if (root == nullptr) return 0;
    return max(depth(root->left), depth(root->right)) + 1;
}

// 普通树 (NBT)
int depth(GNode* root) {
    if (root == nullptr) return 0;
    int childHeight = depth(root->left) + 1;
    int siblingHeight = depth(root->right);
    return max(childHeight, siblingHeight);
}
```

统计非叶子节点 (Count Internal)

```
// 二叉树 (BT)
int countInternal(GNode* root) {
    if (root == nullptr || (root->left == nullptr && root->right == nullptr)) return 0;
    return 1 + countInternal(root->left) + countInternal(root->right);
}
```

---

3. 树的等价性与堆操作

检查两棵树是否相同 (isSame)

```
bool isSame(TreeNode* root1, TreeNode* root2) {
    if (root1 == nullptr && root2 == nullptr) return true;
    if (root1 == nullptr || root2 == nullptr) return false;
    if (root1->val != root2->val) return false;
    return isSame(root1->left, root2->left) && isSame(root1->right, root2->right);
}
```

最小堆插入 (Heap Insert Upheap)

```
void insert(int x) {
    H[++n] = x;
    int curr = n;
    while (curr > 1 && H[curr] < H[curr / 2]) {
        swap(H[curr], H[curr / 2]);
        curr = curr / 2;
    }
}
```

1. 二叉搜索树 (BST) 相关操作

判断是否为 BST

```cpp
bool help(TreeNode* root, TreeNode* min, TreeNode* max) {
    if (root == nullptr) return true;
    if (min != nullptr && root->val <= min->val) return false;
    if (max != nullptr && root->val >= max->val) return false;
    return help(root->left, min, root) && help(root->right, root, max);
}

bool isValidBST(TreeNode* root) {
    return help(root, nullptr, nullptr);
}
```

BST/NBT 查找特定值 (Find/Search)

```cpp
// 二叉树查找
TreeNode* find(TreeNode* root, int k) {
    if (root == nullptr || root->val == k) return root;
    TreeNode* found = find(root->left, k);
    if (found) return found;
    return find(root->right, k);
}

// 普通树 (NBT) 查找
TreeNode* findNBT(TreeNode* root, int k) {
    if (root == nullptr || root->val == k) return root;
    TreeNode* child = root->left;
    while (child != nullptr) {
        TreeNode* found = findNBT(child, k);
        if (found) return found;
        child = child->right;
    }
    return nullptr;
}
```

---

2. 树的高度与平衡性 (Balance)

检查是否为平衡二叉树 (AVL Check)

```cpp
int getHeight(TreeNode* root) {
    if (root == nullptr) return 0;
    int leftH = getHeight(root->left);
    int rightH = getHeight(root->right);
    if (leftH == -1 || rightH == -1 || abs(leftH - rightH) > 1) return -1;
    return max(leftH, rightH) + 1;
}

bool isBalanced(TreeNode* root) {
    return getHeight(root) != -1;
}
```

---

3. 图论算法 (Graph Algorithms)

拓扑排序 (Topological Sort)

利用入度数组和队列实现：

```cpp
void topSort(Graph* G) {
```

```
    queue<int> q;
    vector<int> inDegree(G->n(), 0);
    // 初始化入度并入队入度为0的顶点
    for (int i = 0; i < G->n(); i++) {
        if (inDegree[i] == 0) q.push(i);
    }
    int count = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        count++;
        for (int v : G->adj(u)) {
            if (--inDegree[v] == 0) q.push(v);
        }
    }
    if (count != G->n()) return; // 存在环
}
```

并查集 (Union-Find) 核心逻辑

包含查找（Find）和合并（Union）：

```
// 查找（含路径压缩）
int find(int x) {
    if (parent[x] == x) return x;
    return parent[x] = find(parent[x]);
}

// 合并
bool unite(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);
    if (rootX != rootY) {
        parent[rootX] = rootY;
        return true;
    }
    return false;
}
```

1. 拓扑排序完整代码 (Topological Sort)

```
void topsort(Graph* G, Queue<int>* Q) {
    int v, w;
    int* inDegree = new int[G->n()]; // 初始化入度数组

    // 1. 初始化：将所有顶点的入度设为 0
    for (int i = 0; i < G->n(); i++) inDegree[i] = 0;

    // 2. 统计所有顶点的入度
    for (int u = 0; u < G->n(); u++) {
        for (w = G->first(u); w < G->n(); w = G->next(u, w)) {
            inDegree[w]++; // 每发现一条指向 w 的边，入度 +1
        }
    }

    // 3. 将所有入度为 0 的顶点入队
    for (int v = 0; v < G->n(); v++) {
        if (inDegree[v] == 0) Q->enqueue(v);
    }
```

```
19
20      // 4. 处理队列
21      while (Q->length() != 0) {
22          v = Q->dequeue();
23          print(v); // 输出拓扑序列
24
25          // 5. 遍历邻接点，模拟移除 v 及其发出的边
26          for (w = G->first(v); w < G->n(); w = G->next(v, w)) {
27              inDegree[w]--; // 相邻点入度 -1
28              if (inDegree[w] == 0) Q->enqueue(w); // 若入度降为 0，入队
29          }
30      }
31  }
```

---

2. 图的边操作：寻找并修改边 (Set Edge)

```
1   void setEdge(int v, int w, int wgt) {
2       // 获取顶点 v 对应的邻接链表
3       Edge curr = vertex[v].getValue();
4
5       // 遍历链表寻找目标顶点 w
6       while (curr != nullptr && curr.vertexIndex != w) {
7           curr = vertex[v].next();
8       }
9
10      if (curr != nullptr && curr.vertexIndex == w) {
11          // 如果边已存在：更新权值或根据逻辑删除
12          vertex[v].remove(curr);
13      } else {
14          // 如果边不存在：执行插入操作
15          numEdge++;
16          vertex[v].inst(it);
17      }
18  }
```