Top View of Binary Tree — Stepwise Algorithm & Pseudocode

Goal: Compute the top view of a binary tree (nodes visible when the tree is viewed from above). Output nodes from left to right.

High-level idea:

1. Find horizontal distance (HD) bounds: minimum HD (Lmin) and maximum HD (Rmax).

2. Allocate an index-shifted array (result) of size = Rmax - Lmin + 1. Use a shift = abs(Lmin) so HD maps to indices.

3. Do a level-order (BFS) traversal storing the first node encountered for each HD. This guarantees the top (shallowest) node is recorded.

4. Return the result vector.

Why BFS (level-order)?

Top view requires the shallowest node at each horizontal distance. BFS visits nodes by increasing depth, so the first node seen for each HD is the correct top-view node.

Step-wise algorithm (detailed)

Step 1 — Compute width (Lmin, Rmax)

Traverse the tree recursively to compute the minimum and maximum horizontal distances. Start with pos = 0 at root.

```
void width(Node * root, int pos, int &Lmin, int &Rmax) {
    if (!root) return;

    // visit left subtree (HD - 1)
    width(root->left, pos - 1, Lmin, Rmax);

    // include current node's position in bounds
    Lmin = min(Lmin, pos);

    // visit right subtree (HD + 1)
    width(root->right, pos + 1, Lmin, Rmax);

    // update right bound after recursion (or before — either works)
    Rmax = max(Rmax, pos);
}
```

Step 2 — Allocate result and visited arrays

```
// after calling width(root, 0, Lmin, Rmax)
int size = (Rmax - Lmin) + 1;
vector<int> result(size);        // will hold top-view node data; default 0 if int
vector<int> visited(size, 0);    // flag array: 0 => not set yet, 1 => set
```

Step 3 — BFS to fill result

Perform BFS storing the first node encountered for each index (HD mapped to index). Use initial pos = abs(Lmin) so that HD=0 maps to index=shift.

```
void View(Node *root, int pos, vector<int> &v, vector<int> &result) {
    // pos is the index corresponding to root's HD (use shift = abs(Lmin))
    queue<pair<Node*, int>> q;
    q.push({root, pos});

    while (!q.empty()) {
        Node *temp = q.front().first;
        int idx = q.front().second;
        q.pop();

        // first node at this horizontal index — record it
        if (v[idx] == 0) {
            result[idx] = temp->data;
            v[idx] = 1;
        }

        // push children with their index relative to current node
        if (temp->left)  q.push({temp->left,  idx - 1});
        if (temp->right) q.push({temp->right, idx + 1});
    }
}
```

Step 4 — topView wrapper

```
vector<int> topView(Node *root) {
    if (!root) return {};

    int Lmin = 0, Rmax = 0;
    width(root, 0, Lmin, Rmax);               // compute bounds
    int size = (Rmax - Lmin) + 1;
    vector<int> result(size);
    vector<int> visited(size, 0);

    int shift = abs(Lmin);                    // map HD to index: index = hd + shift
    View(root, shift, visited, result);       // BFS to fill result

    return result;                            // left-to-right order already maintained by indices
}
```

Example mapping (small)

For input tree: 10, 20, 30, 40, 60, 90, 100 (complete 3-level):

Computed Lmin = -2 (node 40), Rmax = +2 (node 100) => size = 5, shift = 2.

Index mapping: HD -2 -> idx 0, HD -1 -> idx 1, HD 0 -> idx 2, HD +1 -> idx 3, HD +2 -> idx 4.

BFS fills result[0]=40, result[1]=20, result[2]=10, result[3]=30, result[4]=100.

Complexity

Time: O(n) — each node visited once (width + BFS).

Space: O(n) — queue + result arrays.

Notes / Tips

- Using a map with BFS (hd -> first node) avoids manual shifting and makes code terser.
- Ensure using idx (current node index) when pushing children, not the original pos parameter.
- Initialize Lmin and Rmax to 0 before calling width.