Algorithm used for inserting an element:

Start from the root.

If the new value is smaller → go left.

If greater or equal → go right.

Repeat until an empty spot is found, then place the new node.

Time Complexity:

- Best/Average: O(log n)
- Worst (skewed tree): O(n)

Algorithm used for deleting an element:

- Start
- First, we check if the tree is empty or not.
- Next, traverse the tree until we reach the node that is to be deleted by comparing the node values to the value entered:
    - If the value entered is less than the node value, move to the left sub-tree
    - Else, move to the right sub-tree.
- Once the value entered matches the node value, check for the following conditions:
    - If the node is a leaf node, simply remove it.
    - If the node has one child (left or right), replace node with its child (left or right) respectively.
    - If the node has two children, find the minimum value node in the right sub-tree (in-order successor) and replace the node's value with the successor's value. Delete the successor.
- End

Time Complexity:

- Best/Average: O(log n)
- Worst (skewed tree): O(n)

Algorithm used for searching an element:

- Start
- First, we check if the tree is empty or not.
- Next, we start comparing the value entered with the root node value.
- If the value matches then the element is found.

- Else:
    - If the value entered is less than the node value, then move to the left sub-tree and compare the value.
    - Else, move to the right sub-tree and compare the value.
- Repeat these steps until the value entered matches the node value.
- End

Time Complexity:

- Best/Average: O(log n)
- Worst (skewed tree): O(n)

Algorithm used for traversing in level-order:

- Start
- First, we check if the tree is empty or not.
- Next, the root is added to the queue.
- While there exist an element in queue:
    - Remove the first element in queue and push the element into the list
    - Then, push the left node of the element into the list (if exists)
    - And, push the right node of the element into the list (if exists)
- Repeat these steps until the entire tree is traversed
- End

Time Complexity:

- Best/Average: O(n)
- Worst (skewed tree): O(n)

Algorithm used for traversing in in-order:

- Start
- First, we check if the tree is empty or not.
- Next, move to the left node of the root element (if exists)
    - If sub-tree exists, traverse through the left node first and then the right node
- Then, push the root node element into the list.
- At the end, move to the right node of the root element (if exists)
    - If sub-tree exists, traverse through the left node first and then the right node
- Repeat these steps until the entire tree is traversed
- End

Time Complexity:

- Best/Average: O(n)
- Worst (skewed tree): O(n)

Algorithm used for traversing in pre-order:

- Start
- First, we check if the tree is empty or not.
- Next, push the root node element into the list
- Then, move to the left node of the element (if exists)
- And, move to the right node of the element (if exists)
- Repeat these steps until the entire tree is traversed
- End

Time Complexity:

- Best/Average: O(n)
- Worst (skewed tree): O(n)

Algorithm used for traversing in post-order:

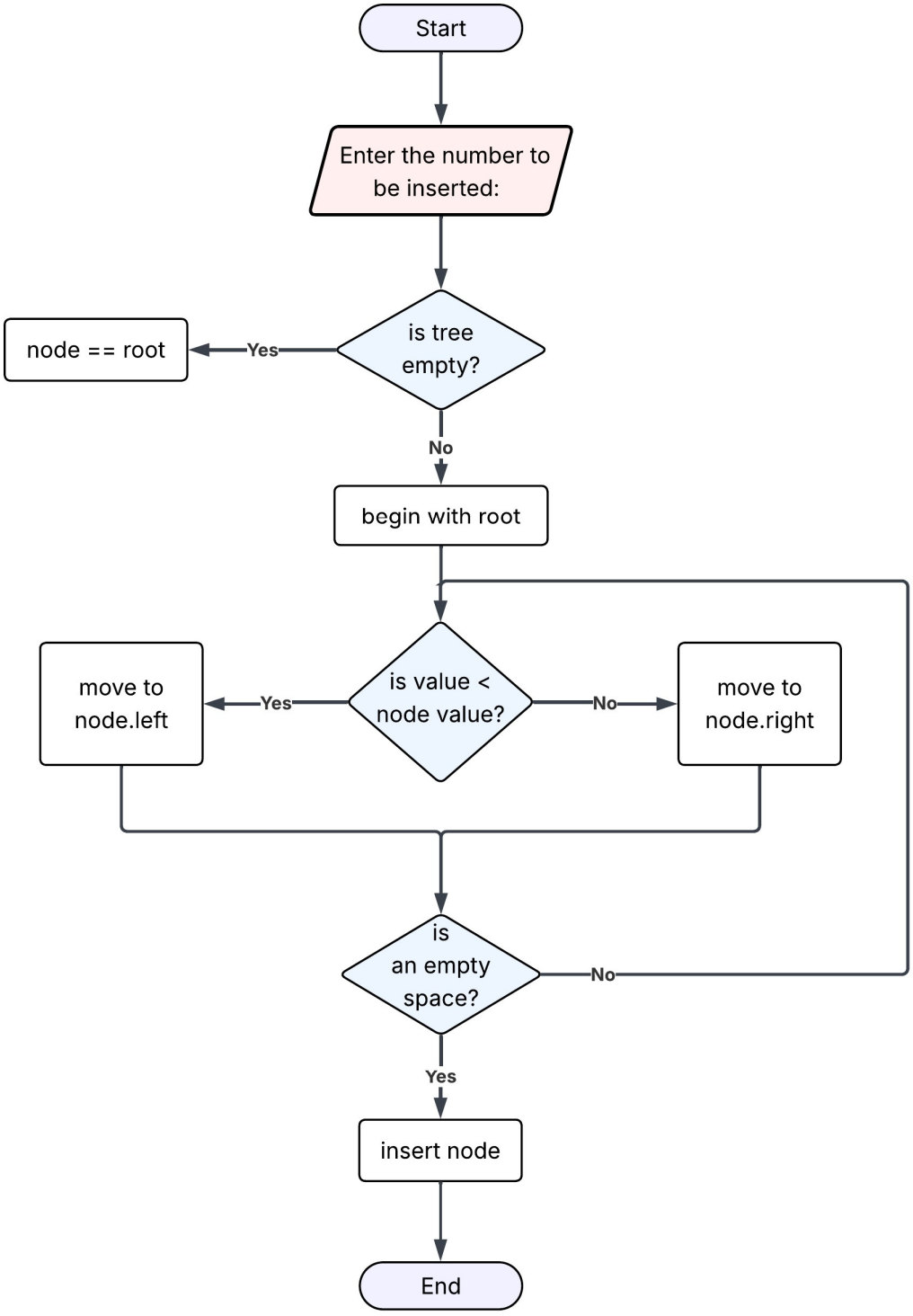
- Start
- First, we check if the tree is empty or not.
- Next, move to the left node of the root element (if exists)
    - If sub-tree exists, traverse through the left node first and then the right node
- Then, move to the right node of the root element (if exists)
    - If sub-tree exists, traverse through the left node first and then the right node
- At the end, push the root node element into the list.
- Repeat these steps until the entire tree is traversed
- End

Time Complexity:

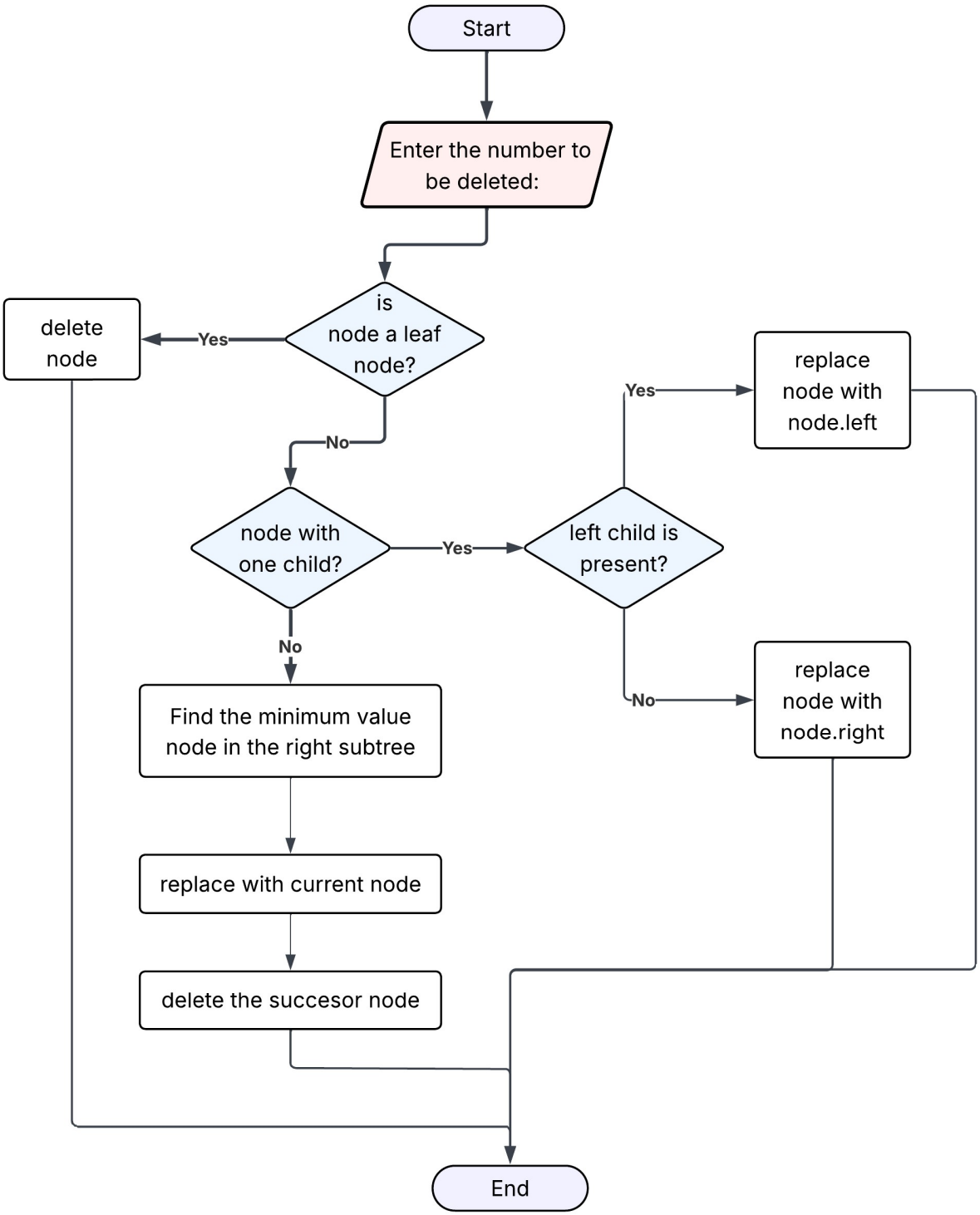
- Best/Average: O(n)
- Worst (skewed tree): O(n)

Flowcharts:

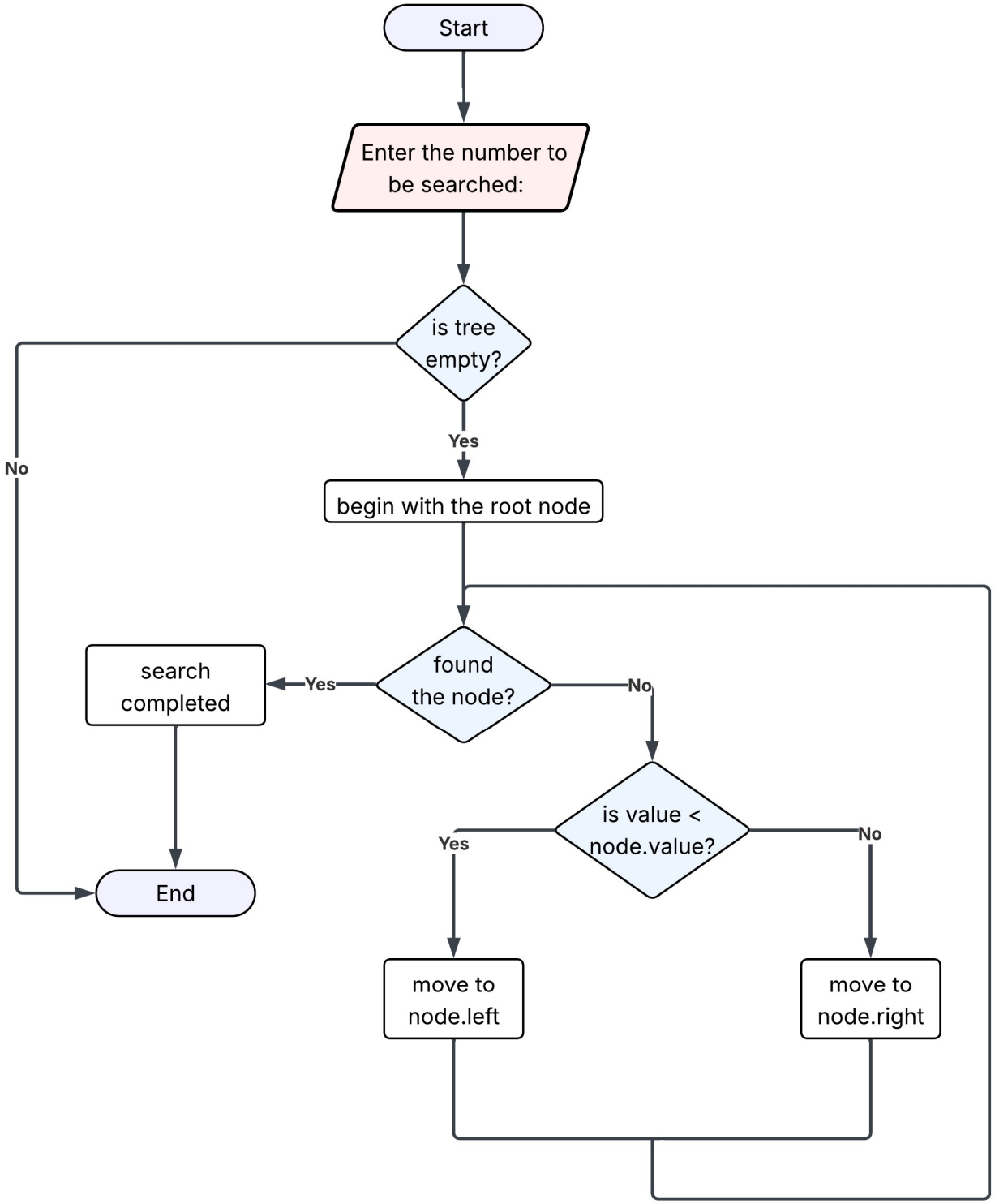
Insertion of an element



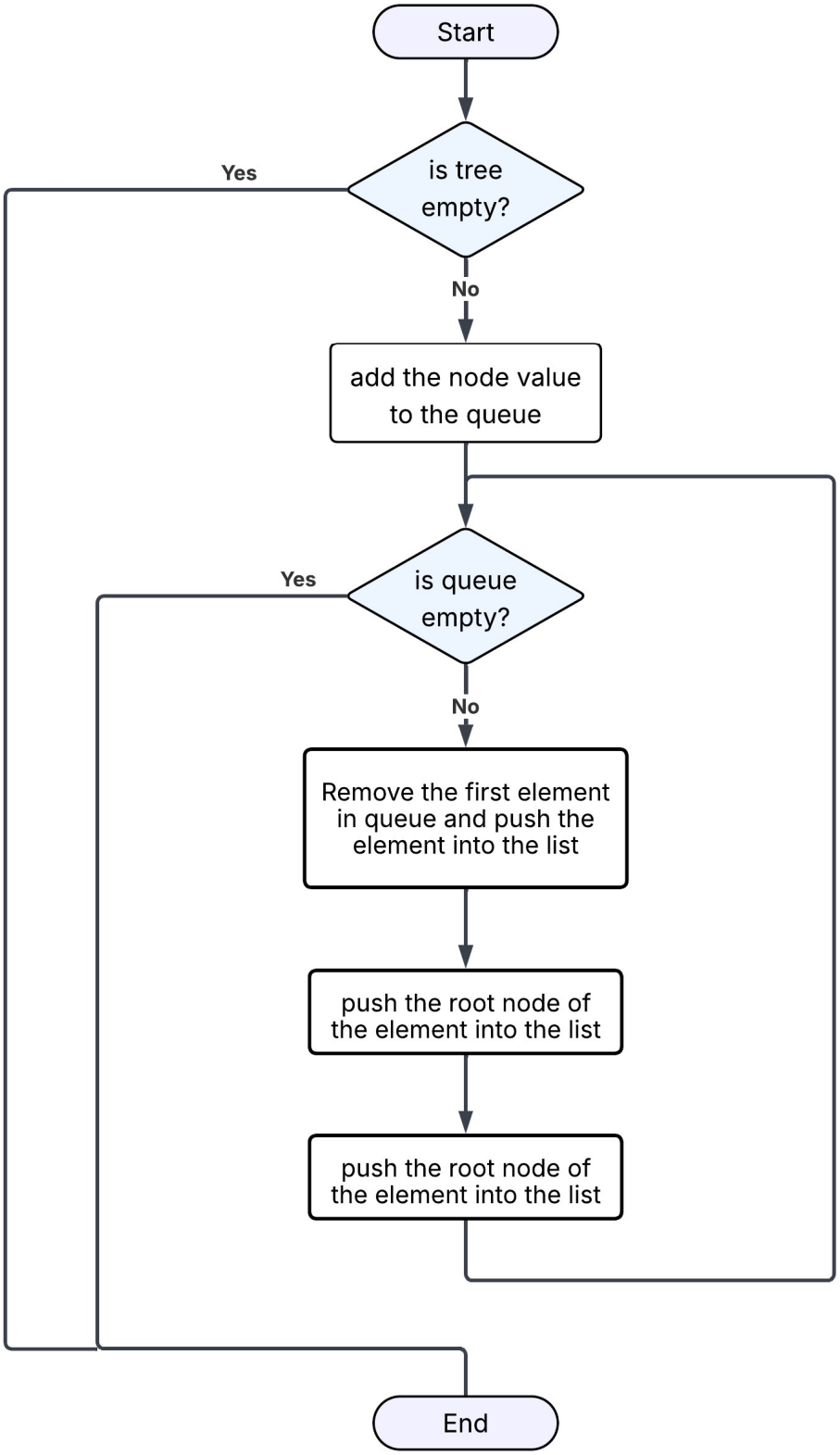
Deletion of an element



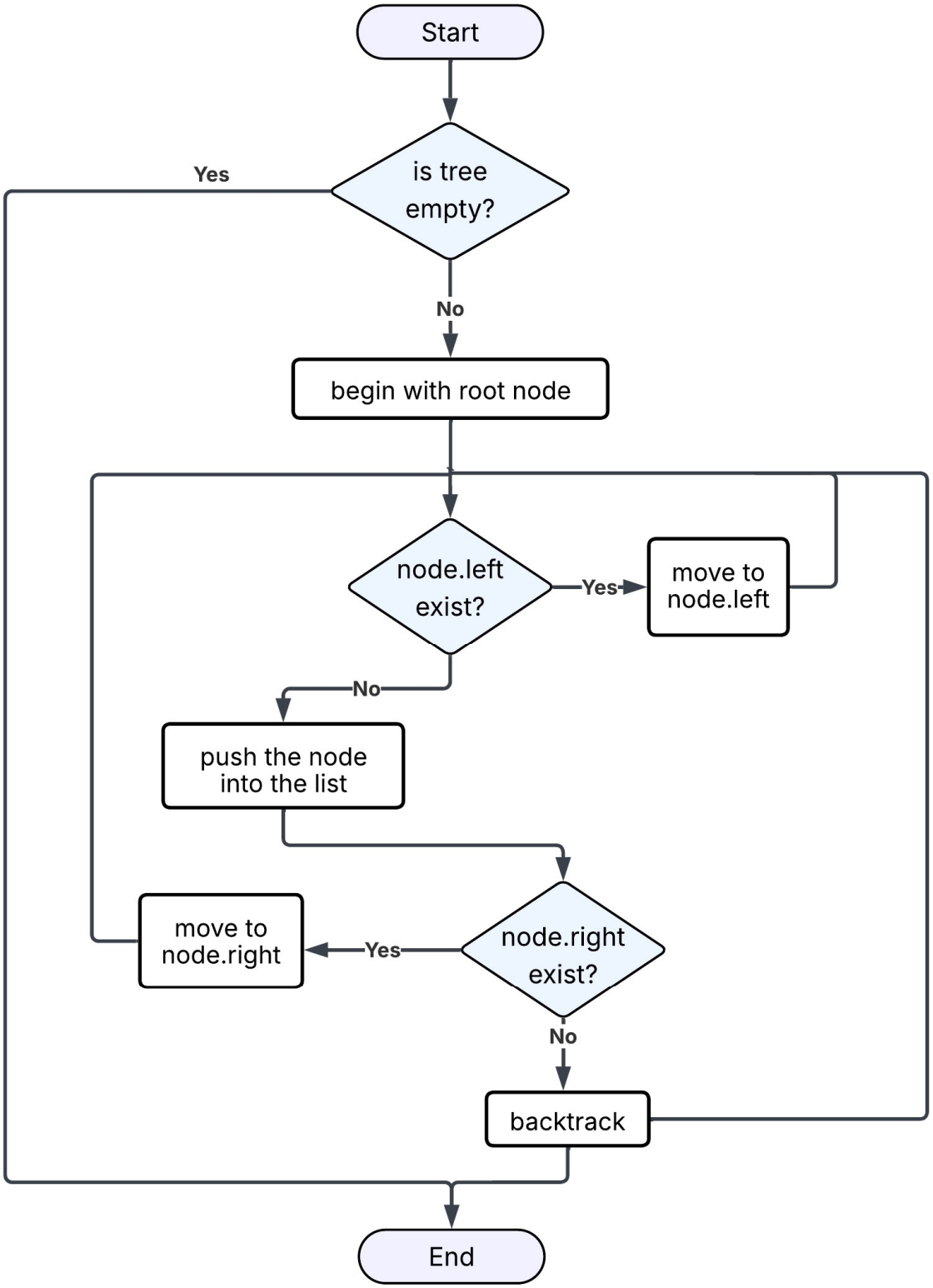
Searching an element



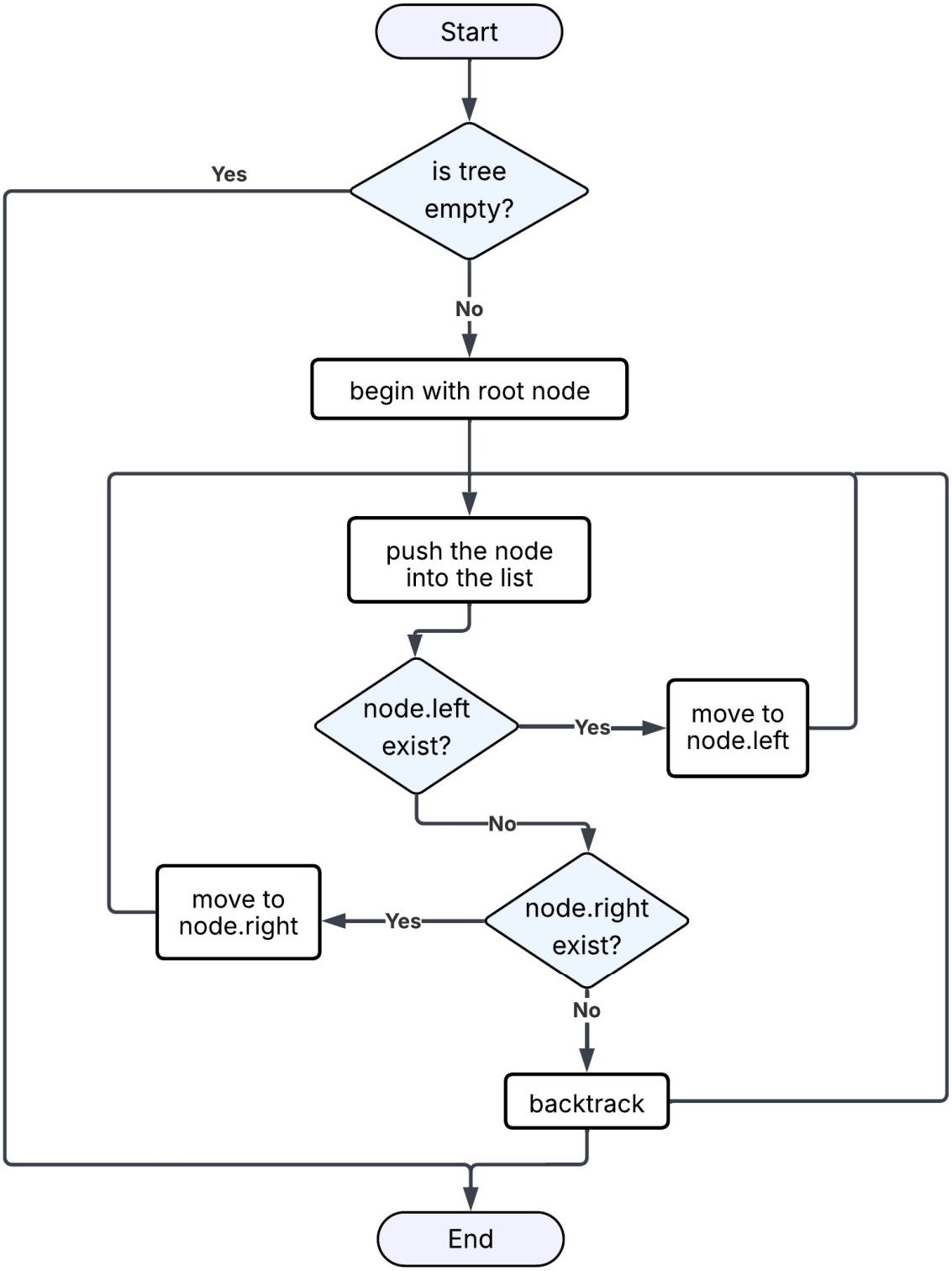
Traverse in Level-order



Traverse in In-order



Traverse in Pre-order



Traverse in Post-order