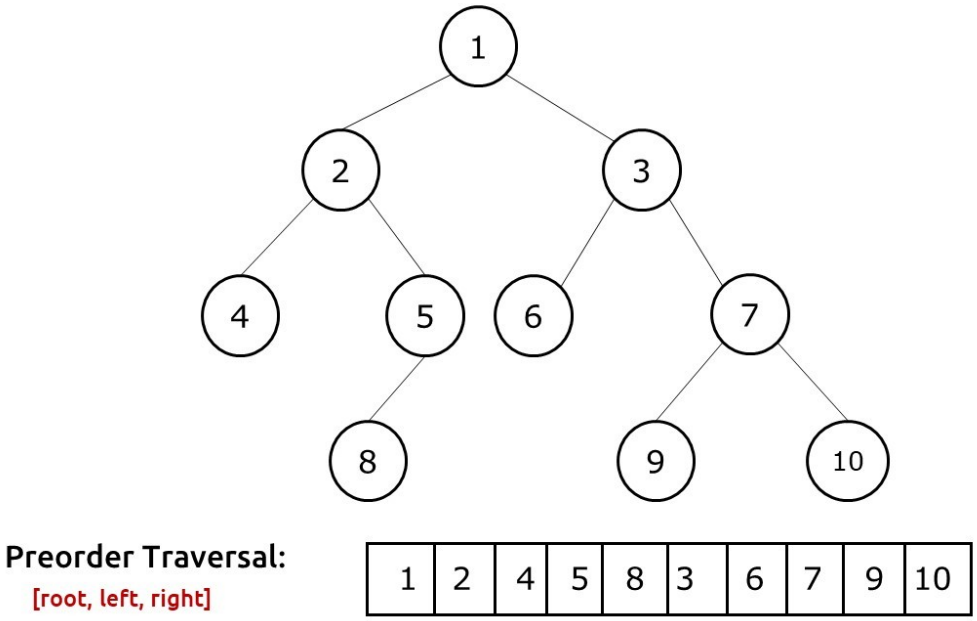
Preorder Traversal of Binary Tree
Problem Statement: Given a binary tree print the preorder traversal of binary tree.
Example:



Solution:
Disclaimer. *Don't jump directly to the solution, try it out yourself first.*

Solution 1: Iterative
Intuition: In preorder traversal, the tree is traversed in this way: **root,** left, right. When we visit a node, we print its value, and then we want to visit the left child followed by the right child. The fundamental problem we face in this scenario is that there is no way that we can move from a child to a parent. To solve this problem, we use an explicit stack data structure. While traversing we can insert node values to the stack in such a way that we always get the next node value at the top of the stack.
Approach:
The algorithm approach can be stated as:

- We first take an explicit stack data structure and push the root node to it.(if the root node is not NULL).
- Then we use a while loop to iterate over the stack till the stack remains non-empty.
- In every iteration we first pop the stack's top and print it.
- Then we first push the right child of this popped node and then push the left child, if they are not NULL. We do so because stack is a last-in-first-out(LIFO) data structure. We need to access the left child first, so we need to push it at the last.
- The execution continues and will stop when the stack becomes empty. In this process, we will get the preorder traversal of the tree.

Dry Run: In case you want to watch the dry run for this approach, please watch the video attached below.
Code:

- C++ Code
- Java Code

```cpp
#include <bits/stdc++.h>

using namespace std;

struct node {
  int data;
  struct node * left, * right;
};

vector < int > preOrderTrav(node * curr) {
  vector < int > preOrder;
  if (curr == NULL)
    return preOrder;

  stack < node * > s;
  s.push(curr);

  while (!s.empty()) {
    node * topNode = s.top();
    preOrder.push_back(topNode -> data);
    s.pop();
    if (topNode -> right != NULL)
```

```
            s.push(topNode -> right);
        if (topNode -> left != NULL)
            s.push(topNode -> left);
    }
    return preOrder;

}

struct node * newNode(int data) {
    struct node * node = (struct node * ) malloc(sizeof(struct node));
    node -> data = data;
    node -> left = NULL;
    node -> right = NULL;

    return (node);
}

int main() {

    struct node * root = newNode(1);
    root -> left = newNode(2);
    root -> right = newNode(3);
    root -> left -> left = newNode(4);
    root -> left -> right = newNode(5);
    root -> left -> right -> left = newNode(8);
    root -> right -> left = newNode(6);
    root -> right -> right = newNode(7);
    root -> right -> right -> left = newNode(9);
    root -> right -> right -> right = newNode(10);

    vector < int > preOrder;
    preOrder = preOrderTrav(root);

    cout << "The preOrder Traversal is : ";
    for (int i = 0; i < preOrder.size(); i++) {
        cout << preOrder[i] << " ";
    }
    return 0;
}
```

Output:

The preOrder Traversal is : 1 2 4 5 8 3 6 7 9 10

Time Complexity: O(N).

Reason: We are traversing N nodes and every node is visited exactly once.

Space Complexity: O(N)

Reason: In the worst case, (a tree with every node having a single right child and left-subtree, follow the video attached below to see the illustration), the space complexity can be considered as O(N).

Solution 2: Recursive

Intuition: In preorder traversal, the tree is traversed in this way: **root,** left, right. When we visit a node, we print its value, and then we want to visit the left child followed by the right child. The fundamental problem we face in this scenario is that there is no way that we can move from a child to a parent. To solve this problem, we use recursion and the recursive call stack to locate ourselves back to the parent node when execution at a child node is completed.

Approach: In preorder traversal, the tree is traversed in this way: **root, left, right**.

The algorithm approach can be stated as:

- We first visit the root node and before visiting its children we print its value.
- After this, we recursively visit its left child.
- Then we recursively visit the right child.
- If we encounter a node pointing to NULL, we simply return to its parent.

Code:

- C++ Code
- Java Code

```cpp
#include <bits/stdc++.h>

using namespace std;

struct node {
   int data;
   struct node * left, * right;
};

void preOrderTrav(node * curr, vector < int > & preOrder) {
   if (curr == NULL)
      return;

   preOrder.push_back(curr -> data);
   preOrderTrav(curr -> left, preOrder);
   preOrderTrav(curr -> right, preOrder);
}

struct node * newNode(int data) {
   struct node * node = (struct node * ) malloc(sizeof(struct node));
   node -> data = data;
   node -> left = NULL;
   node -> right = NULL;

   return (node);
}

int main() {

   struct node * root = newNode(1);
   root -> left = newNode(2);
   root -> right = newNode(3);
   root -> left -> left = newNode(4);
   root -> left -> right = newNode(5);
   root -> left -> right -> left = newNode(8);
   root -> right -> left = newNode(6);
   root -> right -> right = newNode(7);
   root -> right -> right -> left = newNode(9);
   root -> right -> right -> right = newNode(10);

   vector < int > preOrder;
   preOrderTrav(root, preOrder);

   cout << "The preOrder Traversal is : ";
   for (int i = 0; i < preOrder.size(); i++) {
      cout << preOrder[i] << " ";
   }
   return 0;
}
```

Output:

The preOrder Traversal is : 1 2 4 5 8 3 6 7 9 10

Time Complexity: O(N).

Reason: We are traversing N nodes and every node is visited exactly once.

Space Complexity: O(N)

Reason: Space is needed for the recursion stack. In the worst case (skewed tree), space complexity can be O(N).