

# Project

---

## Finding the Kth Ancestor in a Binary Tree

DSA  
PROGRAMMING  
PROGRAMMING

---

Osaid Ahmed Saeed



# Introduction

- The concept of ancestors in a binary tree.
- How we can determine the Kth ancestor of a given node.
- Recursion as an approach to solving this problem.



```
"container">
  class="row">
    class="col-md-6 col-lg-8"> <!--
      <nav id="nav" role="navigation">
        <ul>
          <li><a href="index.html">Home</a></li>
          <li><a href="home-events.html">Home
          <li><a href="multi-col-menu.html">
          <li class="has-children"> <a href="
            <ul>
              <li><a href="tall-button-he
              <li><a href="image-logo.htm
              <li class="active"><a href=
            </ul>
          </li>
          <li class="has-children"> <a href="
            <ul>
              <li><a href="variable-width-
```



# Problem Statement

- Given a binary tree and a specific node, find its Kth ancestor.
- The ancestor at Kth level above the node is required.
- If the Kth ancestor does not exist, return -1.



```
17 string sInput;  
18 int iLength, iN;  
19 double dblTemp;  
20 bool again = true;  
21  
22 while (again) {  
23     iN = -1;  
24     again = false;  
25     getline(cin, sInput);  
26     system("cls");  
27     stringstream(sInput) >> dblTemp;  
28     iLength = sInput.length();  
29     if (iLength < 4) {  
30         again = true;  
31         continue;  
32     } else if (sInput[iLength - 3] != '.') {  
33         again = true;  
34         continue;  
35     } while (++iN < iLength) {  
36         if (isdigit(sInput[iN])) {  
37             continue;  
38         } else if (iN == (iLength - 3) ) {  
39             continue;  
40         }
```



# Binary Tree Structure

- A binary tree is a hierarchical structure where each node has at most two children.
- Parent-child relationships define the hierarchy.
- The importance of left and right child pointers in traversal.





# Understanding Tree Traversal

- The process of visiting nodes in a specific order.
- Depth-First Search (DFS) helps in finding ancestors.
- Recursion is used to traverse up the tree.

```
int r2, c2;
cout << "Enter rows and columns for second matrix: ";
cin >> r2 >> c2;

// Taking elements of first matrix.
cout << endl << "Enter elements of matrix 1:" << endl;
for (int i = 0; i < r1; ++i)
    for (int j = 0; j < c1; ++j)
    {
        cout << "Enter element a" << i + 1 << j + 1 << " : ";
        cin >> a[i][j];
    }

// Taking elements of second matrix.
cout << endl << "Enter elements of matrix 2:" << endl;
for (int i = 0; i < r2; ++i)
    for (int j = 0; j < c2; ++j)
    {
        cout << "Enter element b" << i + 1 << j + 1 << " : ";
        cin >> b[i][j];
    }
```



- Start from the root and search for the target node.
- Once the node is found, move upwards to count ancestors.
- Reduce K each time a parent is found.
- Stop when K becomes zero, returning the ancestor.

## Recursive Approach for Finding Kth Ancestor

```
self.dict_key_name = {}  
resp_iter = self.stub.GetResponseIter()  
  
statuses = {}  
for data in resp_iter:  
    status = Status(  
        status_id=data.id, name=data.name  
    )  
    statuses[status.name] = status  
  
return statuses
```

# Working of Kth Ancestor Search



- The function explores both left and right subtrees.
- If the target node is found in a subtree, backtrack upwards.
- Maintain a counter for K to determine when to stop.

```
self.dict_key_name(mapping) = mapping
resp_iter = self.stub.GetResponse(mapping)

statuses = {}
for data in resp_iter:
    status = Status(
        status_id=data.id, name=data.name
    )
    statuses[status.name] = status

return statuses
```



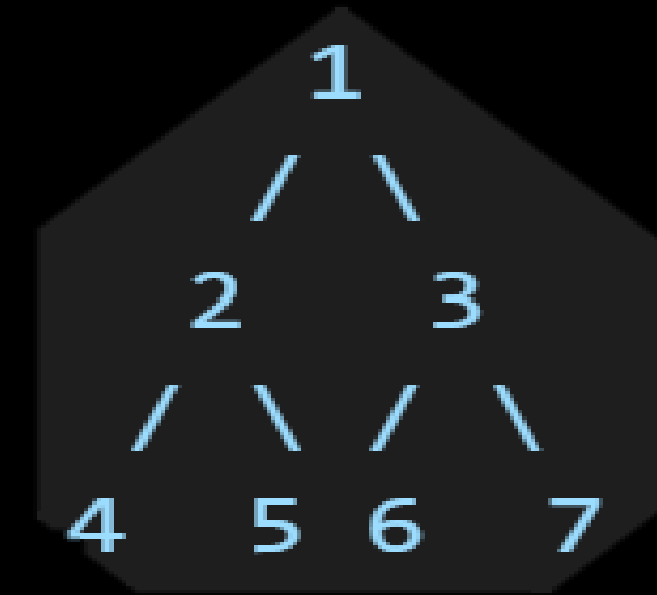
# Sample Binary Tree Structure

- A simple binary tree example:
- Root node with two children.
- Each child has its own left and right children.





- Input: target = 5, k = 2
- Output: The 2nd ancestor of node 5  
is 1



# Output

# Example

```
int n1, m1; // n1, m1 are row and columns for first matrix.
int n2, m2; // n2, m2 are row and columns for second matrix.

// Input of first matrix.
for (int i = 0; i < n1; ++i)
    for (int j = 0; j < m1; ++j)
        cin >> a[i][j];

// Input of second matrix.
for (int i = 0; i < n2; ++i)
    for (int j = 0; j < m2; ++j)
        cin >> b[i][j];

// Output of first matrix.
for (int i = 0; i < n1; ++i)
    for (int j = 0; j < m1; ++j)
        cout << a[i][j] << " ";
    cout << endl;
```



- Implemented a recursive approach to find the Kth ancestor.
- Efficiently tracks ancestors during the traversal.
- Can be extended to handle larger trees and different K values.



# Conclusion

```
int n1, n2; // n1: rows and columns for first matrix, n2: rows and columns for second matrix;
// Input for first matrix.
int i = 1, j = 1;
while (i <= n1)
{
    cout << "Enter elements of matrix 1:" << endl;
    while (j <= n2)
    {
        int a;
        cout << "Enter element a" << i + 1 << j + 1 << " : ";
        cin >> a;
        m1[i][j] = a;
        j++;
    }
    i++;
}

// Input for second matrix.
int r2 = 1, c2 = 1;
while (r2 <= n2)
{
    cout << "Enter elements of matrix 2:" << endl;
    while (c2 <= n1)
    {
        int b;
        cout << "Enter element b" << i + 1 << j + 1 << " : ";
        cin >> b;
        m2[r2][c2] = b;
        c2++;
    }
    r2++;
}
```



THANK YOU  
THANK YOU