: Full binary tree algorithm

```
// Checking if a binary tree is a full binary tree
#include <iostream>
using namespace std;

struct Node {
  int key;
  struct Node *left, *right;
};
// New node creation
 struct Node *newNode(char k) {
 struct Node *node = new Node;
  node->key = k;
  node->right = node->left = NULL;
  return node;
}
bool isFullBinaryTree(struct Node *root) {

  // Checking for emptiness
  if (root == NULL)
    return true;
  // Checking for the presence of children
  if (root->left == NULL && root->right == NULL)
    return true;
  if ((root->left) && (root->right))
    return (isFullBinaryTree(root->left) &&
isFullBinaryTree(root->right));
  return false;
}
int main() {
  struct Node *root = NULL;
  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(6);
  root->left->right->right = newNode(7);
  if (isFullBinaryTree(root))
    cout << "The tree is a full binary tree\n";
  else
    cout << "The tree is not a full binary tree\n";
}
```

: Perfect binary tree algorithm

```
// Checking if a binary tree is a perfect binary tree
#include <iostream>
using namespace std;

struct Node {
  int key;
  struct Node *left, *right;
};

int depth(Node *node) {
  int d = 0;
  while (node != NULL) {
    d++;
    node = node->left;
  }
  return d;
}

bool isPerfectR(struct Node *root, int d, int level = 0) {
  if (root == NULL)
    return true;
  if (root->left == NULL && root->right == NULL)
    return (d == level + 1);
  if (root->left == NULL || root->right == NULL)
    return false;
  return isPerfectR(root->left, d, level + 1) &&
       isPerfectR(root->right, d, level + 1);
}
bool isPerfect(Node *root) {
  int d = depth(root);
  return isPerfectR(root, d);
}
struct Node *newNode(int k) {
  struct Node *node = new Node;
  node->key = k;
  node->right = node->left = NULL;
   return node;
}

int main() {
  struct Node *root = NULL;
  root = newNode(1);
  root->left = newNode(2);
```

```
  root->right = newNode(3);
  root->left->left = newNode(4);
  root->left->right = newNode(5);
  root->right->left = newNode(6);
  root->right->right = newNode(7);

  if (isPerfect(root))
    cout << "The tree is a perfect binary tree\n";
      else
    cout << "The tree is not a perfect binary tree\n";
}
```

: Complete binary tree algorithm

```
// Checking if a binary tree is a complete binary tree
#include <iostream>

using namespace std;

struct Node {
  int key;
  struct Node *left, *right;
};

// Create node
struct Node *newNode(char k) {

  struct Node *node = new Node;
  node->key = k;
  node->right = node->left = NULL;
  return node;
}

// Count the number of nodes
int countNumNodes(struct Node *root) {
  if (root == NULL)
    return (0);
  return (1 + countNumNodes(root->left) + countNumNodes(root-
>right));
}

// Check if the tree is a complete binary tree
bool checkComplete(struct Node *root, int index, int
numberNodes) {
```

```
  // Check if the tree is empty
  if (root == NULL)
    return true;
  if (index >= numberNodes)
    return false;
   return (checkComplete(root->left, 2 * index + 1,
numberNodes) &&
checkComplete(root->right, 2 * index + 2, numberNodes));
}

int main() {
  struct Node *root = NULL;
  root = newNode(1);
  root->left = newNode(2);
  root->right = newNode(3);
  root->left->left = newNode(4);
  root->left->right = newNode(5);
  root->right->left = newNode(6);

  int node_count = countNumNodes(root);
  int index = 0;

  if (checkComplete(root, index, node_count))
    cout << "The tree is a complete binary tree\n";
  else
    cout << "The tree is not a complete binary tree\n";
}
```

: Balanced binary tree algorithm

```
 /* C++ program to check if
a tree is height-balanced or not */
#include<iostream>
using namespace std;

class node {
public:
    int data;
    node* left;
     node* right;
};
/* Returns the height of a binary tree */
int height(node* node);
/* Returns true if binary tree
with root as root is height-balanced */
bool isBalanced(node* root)
{
    int lh; /* for height of left subtree */
    int rh; /* for height of right subtree */
    /* If tree is empty then return true */
    if (root == NULL)
         return 1;
    /* Get the height of left and right sub trees */
    lh = height(root->left);
    rh = height(root->right);
    if (abs(lh - rh) <= 1 && isBalanced(root->left) &&
isBalanced(root->right))
        return 1;
    /* If we reach here then
    tree is not height-balanced */
    return 0;
}
/* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */
/* returns maximum of two integers */
int max(int a, int b)
{
 return (a >= b) ? a : b;
}

/* The function Compute the "height"
of a tree. Height is the number of
nodes along the longest path from
the root node down to the farthest leaf node.*/
int height(node* node)
{
```

```
    /* base case tree is empty */
    if (node == NULL)
        return 0;
    /* If tree is not empty then
    height = 1 + max of left
        height and right heights */
         return 1 + max(height(node->left),
                    height(node->right));
}

/* Helper function that allocates
a new node with the given data
and NULL left and right pointers. */
node* newNode(int data)
{
    node* Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;

    return (Node);
}

// main program
int main()
{
    node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    if (isBalanced(root))
        cout << "Tree is balanced";
    else
        cout << "Tree is not balanced";
    return 0;
}
```