Explanation of binary_tree Function (Using Queue for Binary Tree Construction)

```
node* binary_tree(int arr[],int n){
  node* root = new node(arr[0]);
  int i = 1;
  int j = 2;
  queue<node*>q;
  q.push(root);
  while(q.size()>0 && i<n){
    node* temp = q.front();
    q.pop();
    node* l;
    node* r;
    if(arr[i] != INT_MIN) l = new node(arr[i]);
    else  l = NULL;
    if(j<n && arr[j] != INT_MIN) r = new node(arr[j]);
    else  r = NULL;
    temp -> left = l;
    temp -> right = r;
    if(l)q.push(l);
    if(r)q.push(r);
    i += 2;
    j += 2;
  }
  return root;
}
```

The function binary_tree(int arr[], int n) constructs a binary tree from a given array using a **queue-based level-order approach**. This ensures that nodes are assigned to the tree in a **breadth-first manner**, maintaining a proper binary structure.

Step-by-Step Breakdown

1. **Creating the Root Node**:

2. node* root = new node(arr[0]); // Create root node
    - The first element of the array (arr[0]) becomes the root of the tree.
    - A node object is created dynamically using new node(arr[0]).
3. **Initializing Queue for Level-Order Processing**:
4. queue<node*> q;
5. q.push(root);
    - A queue (q) is used to maintain nodes whose children are yet to be assigned.
    - Initially, the root node is pushed into the queue.
6. **Iterating Over the Array to Assign Children:**
7. int i = 1;
8. while(q.size() > 0 && i < n){
    - The loop runs until **all elements are processed** or **the queue is empty**.
    - i is used to track the next index in the array.
9. **Processing Each Node in the Queue**:
10. node* temp = q.front();
11. q.pop();
    - Extract the **front node** from the queue.
    - This node is the **parent** for the next two children (if available).
12. **Assigning the Left Child:**
13. node* l = NULL;
14. if(arr[i] != INT_MIN) l = new node(arr[i]);
15. temp->left = l;
16. if(l) q.push(l);
17. i++;
    - A **left child** node is created only if arr[i] is not INT_MIN (which represents NULL).
    - The new node is assigned to temp->left.
    - If the left child is created, it is **added to the queue** for further processing.
18. **Assigning the Right Child (if exists):**
19. if(i < n){

20. node* r = NULL;
21. if(arr[i] != INT_MIN) r = new node(arr[i]);
22. temp->right = r;
23. if(r) q.push(r);
24. i++;
25. }
    - If another element exists (i < n), a **right child** is assigned.
    - The same **NULL check** is applied using INT_MIN.
    - If a right child is created, it is **added to the queue**.
26. **Returning the Constructed Tree Root:**
27. return root;
    - The root of the constructed tree is returned.

Why Use a Queue?

- The queue ensures that nodes are **processed in level order**.
- It maintains a **FIFO (First In, First Out)** approach, which is perfect for **breadth-first tree construction**.

Dry Run Example

Input Array:

arr[] = {1, 2, 3, 4, 5, INT_MIN, 6}

Step-by-Step Tree Construction

Step	Queue (Front to Back)	Processed Node	Left Child	Right Child
1	[1]	1	2	3
2	[2, 3]	2	4	5
3	[3, 4, 5]	3	NULL	6
4	[4, 5, 6]	4	NULL	NULL
5	[5, 6]	5	NULL	NULL
6	[6]	6	NULL	NULL

Step	Queue (Front to Back)	Processed Node	Left Child	Right Child
7	[] (empty)	-	-	-

Final Binary Tree Representation:

```
    1
   / \
  2   3
 / \   \
4   5   6
```

This method ensures that the tree is constructed correctly while handling missing nodes (INT_MIN) properly. 🚀