Lecture-75

(Generic Trees) (N-Array Trees)

Q. What are Generic Trees?

⇒ every node has data & list of references to its children node.

Unlike the linked list, each node stores the address of multiple nodes. Every node stores address of its children and the very first node's address will be stored in a separate pointer called root.

- Creating a node class:-

```
class Node{
public: char data;
        Vector<Node*> children;
};
        Node(char data){
            this->data = data;
        }
};
int main(){
    Node* root = new Node('A');
    root->children.push_back(new Node('B'));
    root->children.push_back(new Node('C'));
    cout << root->data << endl;
    for(Node* child : children->data){
        cout << child->data << " ";
    } cout << endl;
    return 0;
}
```

A. Traversal :- ① DFS → Depth First Search
   - ↳ Pre-order
   - ↳ In-order
   - ↳ Post-order

   ② BFS
   - ↳ Breadth First Search

⇒ Preorder Traversal

```
Void preorderTraversal(Node* root) {

    if (root == NULL) {
        return;
    }
    cout << root -> data << " ";
    for (Node *child: root -> children) {
        preorderTraversal (child);
    }
    return;
}
int main () {
    Node* root = new Node ('A');
    root -> children.push_back (new Node ('B'));
    root -> children.push_back (new Node ('C'));
    root -> children.push_back (new Node ('D'));
    root -> children[0] -> children.push_back (new Node ('E'));
    root -> children[0] -> children.push_back (new Node ('F'));
    root -> children[2] -> children.push_back (new Node ('G'));
    return 0;
    preorderTraversal (root);

    return 0;
}
```

⇒ Inorder Traversal:

① Recursively visit all child nodes except last node
② Print root node
③ Recursively visit last child node.

```
Void InorderTraversal (Node* root) {

    if (root == NULL) {
        return;
    }
    int childnodes = (root->)children.size();
    for (int i=0; i < childnodes-1; i++) {
        InorderTraversal (root->children[i]);
    }
    Cout << root -> data << " ";
    if( childnodes > 0) {
        InorderTraversal (root->children[childNodes-1]);
    }
}
```

⇒ PostorderTraversal

① Recursively visit all child nodes

② Print root

```
Void postorderTraversal (Node* root) {
    if (root == NULL) {                    // base case
        return;
    }
    for (Node* child : root->children) {   // Recursive call
        postorderTraversal (child);
    }
    Cout << root -> data << " ";
}
```

CLASSTIME Pg. No.
Date / /

BFS → Level order traversal

queue → FIFO

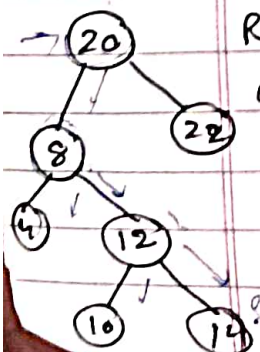
```
Void levelorderTraversal (Node* root) {

    if (root == NULL) {
        return;
    }
    queue <Node*> q;
    q.push(root);

    while (!q.empty()) {
        int currLevelNodes = q.size();
        while (currLevelNodes--) {
            Node* curr = q.front();
            q.pop();
            cout << curr->data << " ";

            for (Node* child: curr->children) {
                q.push(child);
            }
        } cout << endl;
    }
}
```

Ques: Given a generic tree and an integer (x). Find and return the node with the next larger element in the tree i.e. find a node just greater than x. Return NULL if no node is present with value greater than x.



Input: X = 10

output: 12

⇒ if currNode->data > x
&& currnode->data < ansNode->data
→ update ansNode

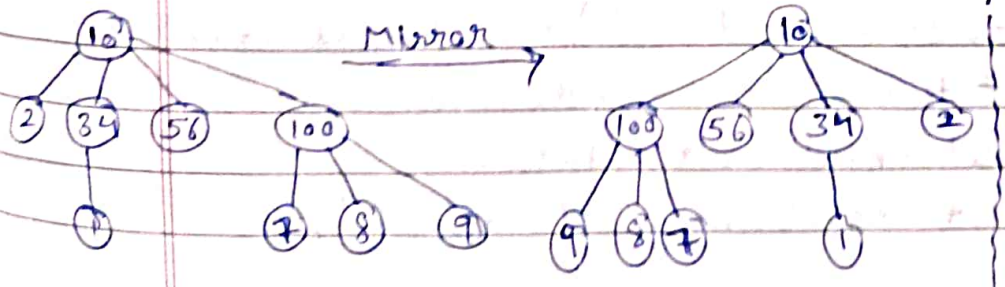
Sol:-

```
Void justGreaterNode(Node* root, int x, Node*& ans) {
    if (root == NULL) {
        return;
    }
    if (root->data > x && (ans == NULL || root->data < ans->data)
    {
        ans = root;
    }
    for (Node* child : root->children) {
        justGreaterNode(child, x, ans);
    }
}
int main() {
    }

    int x = 10;
    Node* ans = NULL;
    justGreaterNode(root, x, ans);
    cout << "ANS - ";
    if (ans == NULL) cout << "NULL" << endl;
    else cout << ans->data << endl;

    return 0;
}
```

Ques:- Given a tree where every node contains variable no. of children, convert the tree into its mirror



① data
Vector<Node*> children;
reverse.

CLASSTIME | Pg. No.
Date / /

Sol.→

```
Void mirrorTree(Node* root){
    if (root==NULL){
        return;
    }
    if (root->children.size()<2)
        return;
    }
    reverse(root->children.begin(), root->children.end());
    for (Node* child: root->children){
        mirrorTree(child);
    }
}
```

T.C ⇒ O(n)
S.C ⇒ O(h)

Ques:- Serialize and deserialize an N-ary tree.
Serialization is the process of converting an object into a format that can be stored or transmitted. Deserialization is the process of converting the serialized string back into an object. Serialization is to store ~~tree~~ tree in a file so that it can be later restored. The structure of tree must be maintained. Deserialization is reading tree back from file.

Sol:→

```
Void SerializeTree(Node* root, string& ans){
    if (root==NULL){
        return;
    }
    ans += to_string(root->data) + ":" + ~~to_string(root~~
        + to_string(root->children.size()) + ":"
    for (Node* child: root->children){
        ans += to_string(child->data) + ",";
    }
```

```
    ans.pop_back();
    ans += "\n";
    for (Node* child: root->children) {
        SerializeTree(child);
    }
}

Node* deserialiseTree(serialisedStr) {
    if (serialisedStr == "") {
        return NULL;
    }
    unordered_map<int, string> mp;

Node* deserialiseTreeHelper(int NodeValue, unordered_map<int,string> mp)

    Node* node = new Node(NodeValue);
    string nodestr = mp[nodeValue];  // "2:30,40"
    if (nodestr[0] == 'o') {
        //leaf node
        return node;
    }
    int breakPos = nodestr.find(':')
    int childNodeNumber = sto i(nodestr.substr(0, breakPos));
    string childNodestr = nodestr.substr(breakPos+1); //"30,40
    int start = 0;
    for (int i=0; i<childNodeNumber; i++) {
        int end = childNodestr.find(',', start);
        int childNodeValue = sto i(childNodestr.substr(start, end));
        node->children.push_back(deserialiseTreeHelper
                                 (childNodeValue, mp));
        start = end+1;
    }
    return node;
{

if(end == string::npos){
end = childNodestr.size()
}
```

CLASSTIME Pg. No.
Date / /

string

```
Node * deserialiseTree (string serialisedstr) {

    if (serialisedstr == "") {
        return NULL;
    }

    unordered_map<int, string> mp;
    int start = 0;
    for (int i = 0; i < serialisedstr.size(), i++) {
        if (serialisedstr[i] == '\n') {
            string nodestr = serialisedstr.substr(start, i - start);
            int breakPos1 = nodestr.find(':');
            int nodeValue = stoi(nodestr.substr(0, breakPos1));
            mp[nodeValue] = nodestr.substr(breakPos1 + 1);
            start = i + 1;
        }
    }

    int rootNodeValue = stoi(serialised.substr(0, serialisedstr.find(':')));
    return dereriliseTreeHelper(rootNodeValue, mp);
}
int main() {
    //Tree banao:
    levelorderTraversal(root);
    string serialisedTree = "";
    SericlizeTree(root, serialisedTree);
    cout << serialisedTree << endl;
    Node * deserialiseRoot = deserialiseTree(serialisedTree);
    levelOrderTraversal(deserialisedRoot);
    return 0;
}
```