

BAHRIA UNIVERSITY (KARACHI CAMPUS)

|  |  |
| --- | --- |
| **CSC 221 - Data Structures & Algorithms – Assignment 2** | |
| CLO-3 | Deadline: 18th December, 22 |
| Class: BSE-3A/B | Total Marks 5 |
| Name: Muhammad Shoaib Akhter Qadri | Enrollment No: 02-131212-009 |

Suppose you are asked to store a binary tree in a file considering each node in the binary tree contains a character string. To create the file in preorder file format, do a preorder traversal of the tree, when a node is visited put the character string in the file followed by a newline and when a null is visited put a dot, ".", followed by a newline. For example, the below tree is stored as the file "m n o . . . p q . . r . ." where spaces indicate newlines.

m

/ \

n p

/ / \

o q r

* Design an algorithm which gives the preorder file format of a binary tree given a pointer to the root of a binary tree. Consider the binary tree has nodes with fields "data", "left\_child", and "right\_child".
* Design an algorithm which takes a preorder file format of a binary tree and produces the binary tree. Use recursive approach.

**Answer:**

A tree which elements have at most 2 children is called a **binary tree**. Each element in a binary tree can have only 2 children, we can say the left and right child. A binary tree has a special condition that each node can have a maximum of two children. Binary Tree is a special data structure used for data storage purposes. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation.

* **Path**:

Path refers to the sequence of nodes along the edges of a tree.

* **Root**:

The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.

* **Parent**:

Any node except the root node has one edge upward to a node called parent.

* **Child**:

The node below a given node connected by its edge downward is called its child node.

* **Leaf**:

The node which does not have any child node is called the leaf node.

* **Subtree**:

Subtree represents the descendants of a node.

* **Visiting**:

Visiting refers to checking the value of a node when control is on the node.

* **Traversing**:

Traversing means passing through nodes in a specific order.

* **Levels**:

Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.

* **Keys**:

Key represents a value of a node based on which a search operation is to be carried out for a node

# Algorithm to produce preorder file format of binary tree:

* Start at the root of the binary tree and visit the current node.
* Write the character string in the file, followed by a newline.
* If the current node has a left child, repeat steps 1-2 on the left child.
* If the current node has a right child, repeat steps 1-2 on the right child.
* If the current node does not have a left or right child, write a dot, ".", followed by a newline in the file.
* Continue with steps 1-5 until all nodes in the binary tree have been visited.

# Algorithm to produce binary tree from preorder file format:

* Read the first line of the preorder file format and create a new node with the character string as the "data" field.
* If the next line in the file is not a dot, recursively call the algorithm on the next line to create the left child of the current node.
* If the next line in the file is not a dot, recursively call the algorithm on the next line to create the right child of the current node.
* Continue with steps 2-3 until all lines in the preorder file format have been processed.
* Return the root node of the binary tree.

**Implementation In Coding:**

class Node:

def \_\_init\_\_(self, val):  
self.val = val  
self.left = None  
self.right = None

def \_\_repr\_\_(self):  
string = "{}=({},{})".format(self.val, self.left, self.right)  
return string  
def invert\_tree(root):  
if not root:  
return

inverted\_left = invert\_tree(root.right)  
inverted\_right = invert\_tree(root.left)  
root.left = inverted\_left  
root.right = inverted\_right

return root  
a = Node('a')  
b = Node('b')  
c = Node('c')  
d = Node('d')  
e = Node('e')  
f = Node('f')

a.left = b  
a.right = c  
b.left = d  
b.right = e  
c.left = f

inverted\_a = invert\_tree(a)  
assert inverted\_a.left == c  
assert inverted\_a.right == b  
assert inverted\_a.left.right == f  
assert inverted\_a.right.right == d

def \_\_init\_\_(self, val):  
self.val = val  
self.left = None  
self.right = None

def \_\_repr\_\_(self):  
return "{}=[{},{}]".format(self.val, self.left, self.right)

def \_\_hash\_\_(self):  
return hash(self.val)  
def recover\_full(root):  
if not root.left and not root.right:  
return root  
elif root.left and root.right:  
root.left = recover\_full(root.left)  
root.right = recover\_full(root.right)  
return root  
elif root.left:  
return recover\_full(root.left)  
elif root.right:  
return recover\_full(root.right)  
# Tests  
a = Node("a")  
b = Node("b")  
c = Node("c")  
d = Node("d")  
e = Node("e")  
f = Node("f")

a.left = b  
a.right = c  
b.left = d  
d.right = f

print(a)  
print(recover\_full(a))

Inorder Tree Traversal  
Recursive function:  
void inorder (ptr\_node ptr){  
if (ptr){  
inorder (ptr->left\_child);  
cout << ptr->data;  
inorder (ptr->right\_child);}}

Preorder Tree Traversal  
Recursive function:  
void preorder (ptr\_node ptr){  
if (ptr){  
cout << ptr->data;  
preorder (ptr->left\_child);  
preorder (ptr->right\_child);}}

void insert\_node ( tree\_ptr \*node, int num ){  
tree\_ptr ptr, temp = modified\_search ( \*node, num ); // \*\*  
if ( temp || ! ( \*node ) ){  
ptr = new node;  
if ( ptr = = NULL){  
cout << “The memory is full \n”;  
exit ( 1 );}  
ptr->data = num;  
ptr->left\_child = ptr->right\_child = NULL;  
if ( \*node ){  
 if ( num < temp->data )  
 temp->left\_child = ptr;  
else  
 temp->right\_child = ptr;}  
else{  
 \*node = ptr;}}

**Output**

a b c d e f

a

                            /   \

                           b     c

                          /    /   \

                         d    e     f