**Practice Questions for Algorithm Design**

**Series -1: Binary Tree**

Given a complete binary tree of height 5, where the keys from 1 to 63 are organized level-wise, and each level from left to right. For example the root at level 0 containing the key 1 and the left child of root contains the key 2, and the right child contains the key 3, and so on. Based on the above tree, answer the following questions.

1. Write an algorithm to return the **left-most** and **right-most** keys of the tree. Analyze the complexity of your algorithm. Convert your algorithm into viable code abstraction.

**Soln:**

Leftmost(node):

If 2\*node>size:

Return node

Else

Leftmost(2\*node)

Rightmost(node):

If 2\*node+1>size:

Return node

Else

Rightmost(2\*i+1)

Time Complexity: O(h) or O(log n)

We have to traverse the entire height of tree. Height of tree is h or log n.

Code abstraction:

def leftmost(node):

#function to find left most child

if 2\*node>self.size:

return node

else

leftmost(2\*node)

def rightmost(node):

#function to find right most child

if 2\*node+1>self.size:

return node

else

rightmost(2\*node+1)

1. Write an algorithm to return the **parent of the node** containing the key 42. Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

**Soln:**

Parent(node):

If node==1

Return None

Else

Return node//2

Time Complexity: O(1)

We need not traverse or use recursion.

Code abstraction:

def parent(node):

if node==1:

print(“Root Node”)

return None

else:

return (node//2)

parent(42)

1. Write an algorithm to return the **children of the node** containing the key 30. Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

Children(node):

If 2\*node>size:

Leftchild=None

Else

Leftchild=2\*node

If 2\*node+1>size:

Rightchild=None

Else

Rightchild=2\*node+1

Return Leftchild,Rightchild

Time Complexity: O(1)

We need not traverse or use recursion.

Code abstraction:

def children(node):

if 2\*node>self.size:

leftchild=None

else:

leftchild=2\*node

if 2\*node+1>self.size:

rightchild=None

else:

rightchild=2\*node+1

return leftchild,rightchild

left,right=children(30)

1. Write an algorithm to return the **sibling of the node containing the key 20**. Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

Sibiling(node)

If node==1

Return None

Else

If node%2==0

If node+1< size:

Return node+1

Else:

Return None

Else

Return node-1

Time Complexity: O(1)

We need not traverse or use recursion.

Code abstraction:

def sibiling(node):

if node==1:

return None

else:

if node%2==0:

if node+1<self.size:

return node+1

else:

return None

else

return node-1

print(sibiling(20))

1. Write an algorithm to return **all the keys reside at level 4, where root resides at level 0.** Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

Soln:

KeysLevel(level)

If level==0:

Print(tree[1])

Else

If 2\*\*level<size:

Print:Invalid Level

Else:

For i=2\*\*level to 2\*\*(level+1)-1:

If i>size:

Break

Else:

Print:tree[i]

Time Complexity:

O(2\*\*h)

Code Abstraction:

def keys\_at\_level(level):

if 2\*\*level<self.size:

print(“Invalid Level”)

else:

for i in range(2\*\*level,min(2\*\*(level+1)-1,size):

print(tree[i])

keys\_at\_level(4)

1. Write an algorithm to return **all keys residing at the internal nodes of the tree.** Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

Soln:

InternalNodes():

For i:=1 to Size//2+1:

Print:i

Time Complexity: O(n/2) == O(n)

A complete binary tree will have an approx. of n/2 internal nodes

def internalnodes():

for i in range(1,self.size//2+1):

if i\*2<=self.size:

print(i)

internalnodes()

1. Write an algorithm to return **all keys reside at the leaf nodes** of the tree. Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

Soln:

ExternalNodes():

For i:=Size//2+1 to Size:

Print:i

Time Complexity: O(n/2) == O(n)

A complete binary tree will have an approx. of n/2 internal nodes

def externalnodes():

for i in range(self.size//2+1,self.size):

print(i)

externalnodes()

1. Write an algorithm to return the **ancestors of the node containing the key 63**. Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

Soln:

Ancestor(node):

Print(Node)

If node!=1:

Ancestor(node//2)

Time Complexity: O(log n)  
will have go through entire height

def ancestor(node):

print(node)

if node!=1:

ancestor(node//2)

ancestor(63)

1. Write an algorithm to return all the **descendants of the node containing the key 2**. Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

Descendants(node):

Print(Node)

If node\*2<Size:

Descendant(node\*2)

If node\*2+1<Size:

Descendant(node\*2+1)

Time Complexity: O(n)  
Descandant of root is entire tree

def descendant(node):

print(node)

if node\*2<self.size:

descendant(node\*2)

if node\*2+1<self.size:

descendant(node\*2+1)

descendant(2)

1. Write an algorithm to return the **keys reside in the path traversed from the root node to the leaf node containing the key 50**. Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

Soln:

Path(root,p,x):

If root>size:

Return False

p.add(root)

if tree[root]==x:

Return True

If Path(root\*2,p,x) or Path(root\*2+1,p,x):

Return True

p.remove(root)

Return False

Time Complexity:O(log n)

Code Abstraction:

def path(root,p,x):

if root>self.size:

return False

p.append(root)

if tree[root]==x:

return True

if path(root\*2,p,x) or path(root\*2+1,p,x):

return True

p.pop(-1)

return False

path(1,p,50)

1. Write an algorithm to return the keys in the **inorder traversal** of the tree. Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

Soln:

Inorder(node):

If node<=Size:

Inorder(node\*2)

Print:node

Inorder(node\*2+1)

Time Complexity: O(n)

Visit every node

Code abstraction:

def inorder(node):

if node<=self.size:

inorder(node\*2)

print(tree[node])

inorder(node\*2+1)

inorder(1)

1. Write an algorithm to return the keys in the **preorder traversal** of the tree. Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

Preorder(node):

If node<=Size:

Print:node

Preorder(node\*2)

Preorder(node\*2+1)

Time Complexity: O(n) .Visit every node

Code abstraction:

def preorder(node):

if node<=self.size:

print(tree[node])

preorder(node\*2)

preorder(node\*2+1)

preorder(1)

1. Write an algorithm to return the keys in the **postorder traversal** of the tree. Analyze the time complexity of your algorithm. Convert your algorithm into viable code abstraction.

Postorder(node):

If node<=Size:

Postorder(node\*2)

Postorder(node\*2+1)

print(tree[node])

Time Complexity: O(n). Visit every node

Code abstraction:

def postorder(node):

if node<=self.size:

postorder(node\*2)

postorder(node\*2+1)

print(tree[node])

postorder(1)

**BINARY TREE**

#define node class

Class Node:

def \_\_init\_\_(value):

self.data=value

self.right=None

self.left=None

Class BinaryTree:

def \_\_init\_\_():

self.tree=[0]

self.size=0

def leftmost(node):

#function to find left most child

if 2\*node>self.size:

return node

else

leftmost(2\*node)

def rightmost(node):

#function to find right most child

if 2\*node+1>self.size:

return node

else

rightmost(2\*node+1)

def parent(node):

#function to find parent node

if node==1:

print(“Root Node”)

return None

else:

return (node//2)

def children(node):

if 2\*node>self.size:

leftchild=None

else:

leftchild=2\*node

if 2\*node+1>self.size:

rightchild=None

else:

rightchild=2\*node+1

return leftchild,rightchild

def sibiling(node):

if node==1:

return None

else:

if node%2==0:

if node+1<self.size:

return node+1

else:

return None

else:

return node-1

def keys\_at\_level(level):

if 2\*\*level<self.size:

print(“Invalid Level”)

else:

for i in range(2\*\*level,min(2\*\*(level+1)-1,size):

print(tree[i])

def internalnodes():

for i in range(1,self.size//2+1):

if i\*2<=self.size:

print(i)

def externalnodes():

for i in range(self.size//2+1,self.size):

print(i)

def ancestor(node):

print(node)

if node!=1:

ancestor(node//2)

def descendant(node):

print(node)

if node\*2<self.size:

descendant(node\*2)

if node\*2+1<self.size:

descendant(node\*2+1)

def path(root,p,x):

if root>self.size:

return False

p.append(root)

if tree[root]==x:

return True

if path(root\*2,p,x) or path(root\*2+1,p,x):

return True

p.pop(-1)

return False

def inorder(node):

if node<=self.size:

inorder(node\*2)

print(tree[node])

inorder(node\*2+1)

def preorder(node):

if node<=self.size:

print(tree[node])

preorder(node\*2)

preorder(node\*2+1)

def postorder(node):

if node<=self.size:

postorder(node\*2)

postorder(node\*2+1)

print(tree[node])

#main

#Initialize a tree

binaryTree=BinaryTree()

#insert values

for i in range(1,64):

binaryTree.tree.append(i)

binaryTree.size+=1

print(“Left most:”,binaryTree.leftmost(1))

print(“Right most”,binaryTree.rightmost(1))

left,right= binaryTree .children(30)

print(binaryTree .sibiling(20))

binaryTree .keys\_at\_level(4)

binaryTree .internalnodes()

binaryTree .externalnodes()

binaryTree .ancestor(63)

binaryTree .descendant(2)

p=[]

binaryTree .path(1,p,50)

binaryTree .inorder(1)

binaryTree .preorder(1)

binaryTree .postorder(1)