*import* math

class BinaryTree:

    class node:

        def \_\_init\_\_(*self*):

            self.element = 0

            self.parent = None

            self.leftchild = None

            self.rightchild = None

        def \_\_init\_\_(*self*):

            self.sz = 0

            self.root = self.node()

            self.ht = 0

        def \_\_str\_\_(*self*, *depth*=0):

            ret = ""

*# Print right branch*

*if* self.right != None:

                ret += self.right.\_\_str\_\_(depth + 1)

*# Print own value*

                ret += "\n" + ("    "\*depth) + str(self.value)

*# Print left branch*

*if* self.left != None:

                ret += self.left.\_\_str\_\_(depth + 1)

*return* ret

        def findElement(*self*, *e*):

*return*

        def addLeft(*self*, *e*):

*if* self.e:

*if* e < self.e:

*if* self.left is None:

                    self.left = Node(e)

*else*:

                    self.left.insert(e)

*else*:

                self.e=e

*return*

        def addRight(*self*, *e*):

*if* self.e:

*if* e > self.e:

*if* self.right is None:

                    self.right = Node(e)

*else*:

                    self.right.insert(e)

*else*:

                self.e=e

*return*

        def getChildren(*self*, *curnode*):

            children = []

*return* children

        def isExternal(*self*, *curnode*):

*return*

        def inorderTraverse(*self*,*v*):

*if* v == None:

*return* None

            self.inorderTraverse(v.leftchild)

            print(v.element)

            self.inorderTraverse(v.rightchild)

*return*

        def preorderTraverse(*self*,*v*):

*if* v == None:

*return* None

            print(v.element)

            self.preorderTraverse(v.leftchild)

            self.preorderTraverse(v.rightchild)

*return*

        def postorderTraverse(*self*,*v*):

*if* v == None:

*return* None

            self.postorderTraverse(v.leftchild)

            self.postorderTraverse(v.rightchild)

            print(v.element)

*return*

        def findDepth(*self*,*v*):

             depth = 0

            temp = v

*while* temp.parent != None:

            temp = temp.parent

            depth += 1

*return* depth

        def findHeight(*self*,*v*):

*if* v == None:

*return* -1

*return* 1 + max(self.findHeight(v.leftchild), self.findHeight(v.rightchild))

*return*

        def deleteNode(*self*,*n*):

*return*

        def sibling(*self*, *v*):

*return*

        def isRoot(*self*, *v*)

*return*

        def buildTree(*self*, *eltlist*):

            nodelist = []

            nodelist.append(None)

*for* i *in* range(len(eltlist)):

*if* (i!=0):

*if* (eltlist[i]!= None):

                        tempnode = self.node()

                        tempnode.element = eltlist[i]

*if* i!=1:

                            tempnode.parent = nodelist[math.floor(i/2)]

*if* (i%2==0):

                                nodelist[math.floor(i/2)].leftchild = tempnode

*else*:

                                nodelist[math.floor(i/2)].rightchild = tempnode

                        nodelist.append(tempnode)

*else*:

                        nodelist.append(None)

            self.root = nodelist[1]

*#print "final nodelist", len(nodelist)*

*return* nodelist

        def isEmpty(*self*):

*return*

        def size(*self*):

*return* self.sz

def main():

    tree = BinaryTree()

    print (tree.size())

    A=[]

    A = [None, 1, 2, 3, 4, 17, 5, 12, 7, 8, None, 9, None, None, None, None, None, None, None, None, None, None, 10]

    nlist = tree.buildTree(A)

    print("Inorder Traversal:")

    tree.inorderTraverse(tree.root)

    print("Preorder Traversal:")

    tree.preorderTraverse(tree.root)

    print("Postorder Traversal:")

    tree.postorderTraverse(tree.root)

    print("Height of the Tree:")

    print(tree.findHeight(tree.root))

    print("Depth of the Tree:")

    print(tree.findDepth(tree.root))

*if* \_\_name\_\_ == '\_\_main\_\_':

    main()