**SSN College of Engineering**

Department of Information Technology

UIT2201 — Programming and Data Structures

2022 – 2023

**Exercise — 12**

U. Pranaav | IT-B | 3122225002093

1)

I. AIM:

To construct a binary search tree using a linked binary tree implementation.

II. CODE:

from abc import abstractmethod

from abc import ABC

*class* AbstractTree(ABC):

    @abstractmethod

*def* getRoot(*self*):

        """Returns the root position of the tree."""

        raise Exception("Not implemented")

    @abstractmethod

*def* getParent(*self*, *pos*):

        """Returns the parent position of the given position 'pos'."""

        raise Exception("Not implemented")

    @abstractmethod

*def* getNum\_children(*self*, *pos*):

        """Returns the number of children of the given position 'pos'."""

        raise Exception("Not implemented")

    @abstractmethod

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

        """Returns a list of children positions of the given position 'pos'."""

        raise Exception("Not implemented")

    @abstractmethod

*def* \_\_len\_\_(*self*):

        """Returns the total number of positions in the tree."""

        raise Exception("Not implemented")

*def* isRoot(*self*, *pos*):

        """Returns True if the given position 'pos' is the root of the tree, False otherwise."""

        return *self*.getRoot() == *pos*

*def* isLeaf(*self*, *pos*):

        """Returns True if the given position 'pos' is a leaf node (has no children), False otherwise."""

        return *self*.getNum\_children(*pos*) == 0

*def* isEmpty(*self*):

        """Returns True if the tree is empty (has no positions), False otherwise."""

        return len(*self*) == 0

*def* depthN(*self*, *pos*):

        """

        Returns the depth of the position 'pos' in the tree.

        Depth is the number of edges in the path from the root to 'pos'.

        """

        if *self*.isRoot(*pos*):

            return 0

        return 1 + *self*.depthN(*self*.getParent(*pos*))

*def* heightN(*self*, *pos*):

        """

        Returns the height of the position 'pos' in the tree.

        Height is the number of edges in the longest path from 'pos' to a leaf.

        """

        if *self*.isLeaf(*pos*):

            return 0

        return 1 + max([*self*.heightN(child) for child in *self*.getChildren(*pos*)])

*def* height(*self*):

        """Returns the height of the tree (i.e., the height of the root position)."""

        return *self*.heightN(*self*.getRoot())

*class* AbstractBinaryTree(AbstractTree):

    @abstractmethod

*def* getLeft(*self*, *pos*):

        raise Exception("Not implemented")

    @abstractmethod

*def* getRight(*self*, *pos*):

        raise Exception("Not implemented")

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

        if *pos* is None:

            return None

        if *self*.getLeft(*pos*) is not None:

            yield *self*.getLeft(*pos*)

        if *self*.getRight(*pos*) is not None:

            yield *self*.getRight(*pos*)

*def* sibling(*self*, *pos*):

        parent = *self*.getParent(*pos*)

        if parent is None:

            return None

        if *pos* == *self*.right(parent):

            return *self*.left(parent)

        else:

            return *self*.right(parent)

*class* BTNode:

    \_\_slots\_\_ = ["item", "left", "right", "parent"]

*def* \_\_init\_\_(*self*, *item*, *left*=None, *right*=None, *parent*=None):

*self*.item = *item*

*self*.left = *left*

*self*.right = *right*

*self*.parent = *parent*

*def* \_\_getitem\_\_(*self*):

        return *self*.item

*def* \_\_setitem\_\_(*self*, *item*):

*self*.item = *item*

*class* LinkedBinaryTree(AbstractBinaryTree):

*class* BTNode:

        \_\_slots\_\_ = ["item", "left", "right", "parent"]

*def* \_\_init\_\_(*self*, *item*, *left*=None, *right*=None, *parent*=None):

*self*.item = *item*

*self*.left = *left*

*self*.right = *right*

*self*.parent = *parent*

*def* \_\_getitem\_\_(*self*):

            return *self*.item

*def* \_\_setitem\_\_(*self*, *item*):

*self*.item = *item*

    \_\_slots\_\_ = ["root", "size"]

*def* \_\_init\_\_(*self*, *item*=None, *t\_left*=None, *t\_right*=None):

*self*.root = None

*self*.size = 0

*self*.string = ""

        if *item* is not None:

*self*.root = *self*.addRoot(*item*)

        if *t\_left* is not None:

            if *t\_left*.root is not None:

*t\_left*.root.parent = *self*.root

*self*.root.left = *t\_left*.root

*self*.size += *t\_left*.size

*t\_left*.root = None

        if *t\_right* is not None:

            if *t\_right*.root is not None:

*t\_right*.root.parent = *self*.root

*self*.root.right = *t\_right*.root

*self*.size += *t\_right*.size

*t\_right*.root = None

*def* addRoot(*self*, *item*):

        """

        Adds a root node with the given item to the tree.

        Args:

            item: The item to be stored in the root node.

        Returns:

            The root position of the added node.

        Raises:

            ValueError: If the root already exists.

        """

        if *self*.root is not None:

            raise ValueError("Root already exists")

        else:

*self*.root = *self*.BTNode(*item*)

*self*.size += 1

            return *self*.root

*def* \_\_len\_\_(*self*):

        """

        Returns the number of nodes in the tree.

        Returns:

            The size of the tree.

        """

        return *self*.size

*def* getParent(*self*, *pos*):

        """

        Returns the parent position of the given position 'pos'.

        Args:

            pos: The position to get the parent of.

        Returns:

            The parent position of 'pos'.

        """

        return *pos*.parent

*def* getLeft(*self*, *pos*):

        """

        Returns the left child position of the given position 'pos'.

        Args:

            pos: The position to get the left child of.

        Returns:

            The left child position of 'pos'.

        """

        return *pos*.left

*def* getRight(*self*, *pos*):

        """

        Returns the right child position of the given position 'pos'.

        Args:

            pos: The position to get the right child of.

        Returns:

            The right child position of 'pos'.

        """

        return *pos*.right

*def* getRoot(*self*):

        """

        Returns the root position of the tree.

        Returns:

            The root position.

        """

        return *self*.root

*def* getSize(*self*):

        """

        Returns the number of nodes in the tree.

        Returns:

            The size of the tree.

        """

        return *self*.size

*def* getNum\_children(*self*, *pos*):

        """

        Returns the number of children of the given position 'pos'.

        Args:

            pos: The position to get the number of children of.

        Returns:

            The number of children of 'pos'.

        """

        if *pos* is None:

            return 0

        else:

            return 1 + *self*.getNum\_children(*pos*.left) + *self*.getNum\_children(*pos*.right)

*def* addLeft(*self*, *item*, *pos*=None):

        """

        Adds a left child node with the given item to the specified position 'pos' or the root if 'pos' is None.

        Args:

            item: The item to be stored in the left child node.

            pos: The position to add the left child to. If None, the left child is added to the root.

        Returns:

            The position of the added left child node.

        Raises:

            ValueError: If the left child already exists.

        """

        if *pos* is None:

*pos* = *self*.root

        if *self*.getLeft(*pos*) is not None:

            raise ValueError("Left child already exists")

        else:

*pos*.left = *self*.BTNode(*item*, *parent*=*pos*)

*self*.size += 1

            return *pos*.left

*def* addRight(*self*, *item*, *pos*=None):

        """

        Adds a right child node with the given item to the specified position 'pos' or the root if 'pos' is None.

        Args:

            item: The item to be stored in the right child node.

            pos: The position to add the right child to. If None, the right child is added to the root.

        Returns:

            The position of the added right child node.

        Raises:

            ValueError: If the right child already exists.

        """

        if *pos* is None:

*pos* = *self*.root

        if *self*.getRight(*pos*) is not None:

            raise ValueError("Right child already exists")

        else:

*pos*.right = *self*.BTNode(*item*, *parent*=*pos*)

*self*.size += 1

            return *pos*.right

*def* preorder(*self*, *pos*):

        """

        Performs a preorder traversal starting from the given position 'pos'.

        Args:

            pos: The starting position for the preorder traversal.

        """

*self*.string += str(*pos*.item) + ","

        if *pos*.left is not None:

*self*.preorder(*pos*.left)

        if *pos*.right is not None:

*self*.preorder(*pos*.right)

*def* postorder(*self*, *pos*):

        """

        Performs a postorder traversal starting from the given position 'pos'.

        Args:

            pos: The starting position for the postorder traversal.

        """

        if *pos*.left is not None:

*self*.postorder(*pos*.left)

        if *pos*.right is not None:

*self*.postorder(*pos*.right)

*self*.string += str(*pos*.item) + ","

*def* inorder(*self*, *pos*):

        """

        Performs an inorder traversal starting from the given position 'pos'.

        Args:

            pos: The starting position for the inorder traversal.

        """

        if *pos*.left is not None:

*self*.inorder(*pos*.left)

*self*.string += str(*pos*.item) + ","

        if *pos*.right is not None:

*self*.inorder(*pos*.right)

*def* \_\_str\_\_(*self*):

        """

        Returns a string representation of the tree by performing preorder, inorder, and postorder traversals.

        Returns:

            A string representation of the tree.

        """

*self*.string = "The preorder is: "

*self*.preorder(*self*.root)

*self*.string = *self*.string[:-1]

*self*.string += "\nThe inorder is: "

*self*.inorder(*self*.root)

*self*.string = *self*.string[:-1]

*self*.string += "\nThe postorder is: "

*self*.postorder(*self*.root)

*self*.string = *self*.string[:-1]

        return *self*.string

*def* makeMirror(*self*, *pos*=None):

        """

        Constructs a new binary tree representing the mirror image of the original tree.

        Args:

            pos: The position to start the mirror operation. If None, starts from the root.

        Returns:

            The root position of the new mirror tree.

        """

        if *pos* is None:

*pos* = *self*.root

        if *pos* is None:

            return None

        mirnode = *self*.BTNode(*pos*.item)

        mirnode.right = *self*.makeMirror(*pos*.left)

        mirnode.left = *self*.makeMirror(*pos*.right)

        return mirnode

*def* mirror(*self*):

        mirrtree = LinkedBinaryTree()

        mirrtree.root = mirrtree.makeMirror(*self*.getRoot())

        return mirrtree

*class* BinarySearchTree(LinkedBinaryTree):

*def* \_\_init\_\_(*self*, *item*=None, *t\_left*=None, *t\_right*=None):

        super().\_\_init\_\_(*item*, *t\_left*, *t\_right*)

*def* isLeaf(*self*, *pos*):

        return *self*.getLeft(*pos*) is None and *self*.getRight(*pos*) is None

*def* append(*self*, *ele*, *pos*=None):

        if *pos* is None:

            if *self*.isEmpty():

*self*.addRoot(*ele*)

                return None

            else:

*pos* = *self*.getRoot()

        if *ele* < *pos*.item:

            if *self*.getLeft(*pos*) is None:

                return *self*.addLeft(*ele*, *pos*)

            else:

                return *self*.append(*ele*, *self*.getLeft(*pos*))

        elif *ele* > *pos*.item:

            if *self*.getRight(*pos*) is None:

                return *self*.addRight(*ele*, *pos*)

            else:

                return *self*.append(*ele*, *self*.getRight(*pos*))

        else:

            return None

*def* findmin(*self*, *pos*=0):

        if *pos* == 0:

*pos* = *self*.getRoot()

        if *pos*.left is not None:

            return *self*.findmin(*pos*.left)

        else:

            return *pos*.item

*def* findmax(*self*, *pos*=0):

        if *pos* == 0:

*pos* = *self*.getRoot()

        if *pos*.right is not None:

            return *self*.findmax(*pos*.right)

        else:

            return *pos*.item

*def* all\_children(*self*, *pos*):

        l = [childr.item for childr in *self*.getChildren(*pos*)]

        return l

*def* delete(*self*, *ele*, *pos*=0):

        if *pos* == 0:

*pos* = *self*.getRoot()

        if *pos* is None:

            return None

        if *ele* < *pos*.item:

*self*.delete(*ele*, *self*.getLeft(*pos*))

        elif *ele* > *pos*.item:

*self*.delete(*ele*, *self*.getRight(*pos*))

        else:

            if *self*.isLeaf(*pos*):

                if *pos* == *self*.getRoot():

*self*.root = None

                else:

                    parent = *self*.getParent(*pos*)

                    if *self*.getLeft(parent) == *pos*:

                        parent.left = None

                    else:

                        parent.right = None

            elif *self*.getLeft(*pos*) is None or *self*.getRight(*pos*) is None:

                if *self*.getLeft(*pos*) is None:

                    child = *self*.getRight(*pos*)

                else:

                    child = *self*.getLeft(*pos*)

                if *pos* == *self*.getRoot():

*self*.root = child

                else:

                    parent = *self*.getParent(*pos*)

                    if *self*.getLeft(parent) == *pos*:

                        parent.left = child

                    else:

                        parent.right = child

            else:

                min\_node = *self*.getminNode(*self*.getRight(*pos*))

*pos*.item = min\_node.item

*self*.delete(min\_node.item, *self*.getRight(*pos*))

*def* getminNode(*self*, *pos*):

        """Returns the node with the maximum value in the subtree rooted at 'pos'."""

        while *self*.getLeft(*pos*) is not None:

*pos* = *self*.getLeft(*pos*)

        return *pos*

# driver code

if \_\_name\_\_ == "\_\_main\_\_":

    # this part of the code will only be run when the function is called directly

    # it will not be executed when it is imported as a module

    binary\_search\_tree = BinarySearchTree()

    binary\_search\_tree.append(30)

    binary\_search\_tree.append(90)

    binary\_search\_tree.append(70)

    binary\_search\_tree.append(40)

    binary\_search\_tree.append(70)

    binary\_search\_tree.append(88)

    binary\_search\_tree.append(44)

    binary\_search\_tree.append(8)

    binary\_search\_tree.append(12)

    binary\_search\_tree.append(44)

    print(binary\_search\_tree)

    binary\_search\_tree.delete(88)

    print(binary\_search\_tree)

    binary\_search\_tree.delete(90)

    print(binary\_search\_tree)

    binary\_search\_tree.delete(30)

    print(binary\_search\_tree)

    print(binary\_search\_tree.findmax())

    print(binary\_search\_tree.findmin())

III. OUTPUT:

The preorder is: 30,8,12,90,70,40,44,88

The inorder is: 8,12,30,40,44,70,88,90

The postorder is: 12,8,44,40,88,70,90,30

The preorder is: 30,8,12,90,70,40,44

The inorder is: 8,12,30,40,44,70,90

The postorder is: 12,8,44,40,70,90,30

The preorder is: 30,8,12,70,40,44

The inorder is: 8,12,30,40,44,70

The postorder is: 12,8,44,40,70,30

The preorder is: 40,8,12,70,44

The inorder is: 8,12,40,44,70

The postorder is: 12,8,44,70,40

70

8