Course_Work

May 31, 2024

1 Report

Saul Vassallo 88812L

1.1 Statement of Completion

Item	Completed (Yes/No/Partial)
Created first array of integers	Yes
Knuth shuffle	Yes
Inserted in AVL tree	Yes
AVL tree insertion statistics	Yes
Inserted in Red-Black tree	Yes
Red-Black tree insertion statistics	Yes
Inserted in Skip List	Yes
Skip List insertion statistics	Yes
Discussion comparing data structures	Yes

1.2 Code and Report Structure

The code is structured around a class for each tree. Each class is contained in its own file. The *main.py* file runs all the code and executes the project as required.

1.3 Knuth Shuffle

There is not much to say about my implementation of knuth shuffle since there is not much in terms of ways to change the code. However, I will note that my implementation was guided by [1].

[ ]:
```
def knuth_shuffle(array):
    """
    Shuffles the elements of the given array using the Knuth Shuffle algorithm.

    Parameters:
    array (list): The array to be shuffled.

    Returns:
    None. The array is shuffled in-place.
```

```
    """
    for index in range(len(array) - 1, 0, -1):
        swap_index = randint(0, index)
        array[index], array[swap_index] = array[swap_index], array[index]
```

1.4 Binary Tree Base Class

My project is all built off of the BinaryTree abstract base class. This class defines all of the methods used throughout the project. These include: - __get__height(self, node) - traverse(self, string) - __in__order__traversal(self, node) - __pre__order__traversal(self, node) - __post__order__traversal(self, node) - is__binary__tree(self) - __is__binary__tree(self, node) - search(self, key) - __search(self, node, key) - get__leaves(self) - __get__leaves(self, node, count)

The explanations of these methods is documented in the code below.

The class also defines the following 2 abstract methods: - insert(self, key) - insertion__steps__and__rotation(self, key)

These abstract methods handle the tree specific insertion and the gathering of statistics related to that version of insertion.

[ ]:
```
from abc import ABC, abstractmethod

class BinaryTree(ABC):
    def __init__(self):
        self.root = None

    def _get_height(self, node):
        if not node:
            return 0
        return node.height

    def traverse(self, string):
        """
        Traverses the  tree in the specified order.

        Parameters:
        - string (str): The traversal order. Valid values are
        "in_order", "post_order", and "pre_order".

        Returns:
        - None

        Raises:
        - None
        """
        if string.lower() == "in_order":
            self._in_order_traversal(self.root)
        elif string.lower() == "post_order":
```

```python
            self._post_order_traversal(self.root)
        elif string.lower() == "pre_order":
            self._pre_order_traversal(self.root)

    def _in_order_traversal(self, node):
        if not node:
            return
        self._in_order_traversal(node.left)
        print(node.key)
        self._in_order_traversal(node.right)

    def _pre_order_traversal(self, node):
        if not node:
            return
        print(node.key)
        self._pre_order_traversal(node.left)
        self._pre_order_traversal(node.right)

    def _post_order_traversal(self, node):
        if not node:
            return
        self._post_order_traversal(node.left)
        self._post_order_traversal(node.right)
        print(node.key)

    def is_binary_tree(self):
        return self._is_binary_tree(self.root)

    def _is_binary_tree(self, node):
        if node is None:
            return True

        if node.left and node.left.key >= node.key:
            return False

        if node.right and node.right.key <= node.key:
            return False

        left_is_binary = self._is_binary_tree(node.left)
        right_is_binary = self._is_binary_tree(node.right)

        return left_is_binary and right_is_binary

    def search(self, key):
        """
        Search for a node with the given key in the tree.
```

```
        Parameters:
        - key: The key to search for.

        Returns:
        - True if found and False if otherwise
        """
        return self._search(self.root, key)

    def _search(self, node, key):
        if not node:
            return False
        if key == node.key:
            return True
        if key < node.left:
            return self._search(node.left, key)

        return self._search(node.right, key)

    @abstractmethod
    def insert(self, key):
        """
        Inserts a new key into the Tree.

        Args:
            key: The key to be inserted into the Tree.
        """

    def get_leaves(self):
        """
        Returns the number of leaves in the tree.

        Returns:
            int: The number of leaves in the tree.
        """
        leaves = 0
        return self._get_leaves(self.root, leaves)

    def _get_leaves(self, node, count):
        if not node:
            return count
        if not node.left and not node.right:
            return count + 1

        new_count = self._get_leaves(node.left, count)
        return self._get_leaves(node.right, new_count)

    @abstractmethod
```

```python
    def insertion_steps_and_rotation(self, key):
        """
        Perform an insertion of a key into the tree and return the number
        of steps taken and whether a rotation was performed or not.

        Parameters:
        - key: The key to be inserted into the tree.

        Returns:
        A tuple containing the number of steps taken during the
        insertion process and 1 if a rotation occured or 0 otherwise.
        """
```

1.5 AVL

The code for this section can be found below. On top of implementing insertion and extracting the statistics, I also implemented a method that checks whether the tree generated is an AVL tree. This test was heavily inspired by [2].

1.5.1 Insertion

AVL insertion is done by recursively calling the insertion method until the correct insertion place is found. Then while recursively unwinding, the balance factor of the current node is checked by making use of the *_get_height()* method. If the balance factor is not within the correct range, rotations are performed as needed.

1.5.2 Statistics

The statistics were gathered by simply keeping track and returning how many steps it took and whether a rotation was performed or not.

The statistics are as follows:
AVL Tree Insertion Steps Statistics:
Minimum: 10
Maximum: 15
Mean: 13.258741258741258
Standard Deviation: 0.9402053041681981
Median: 13

AVL Tree Rotations Statistics:
Minimum: 0
Maximum: 1
Mean: 0.34965034965034963
Standard Deviation: 0.4770978700669053
Median: 0

AVL Tree Height: 14
AVL Tree Leaves: 2251

[ ]:
```python
"""
"""
from BinaryTree import BinaryTree

class AVLNode:
    """

    """
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 0

    def __str__(self):
        return f"{self.key}"

    def set_height(self, new_height):
        """

        """
        self.height = new_height


class AVLTree(BinaryTree):
    """

    """
    def insert(self, key):
        """
        Inserts a new key into the Tree.

        Args:
            key: The key to be inserted into the Tree.
        """
        self.root = self._insert(self.root, key)

    def _insert(self, node, key):
        # Rec cases
        if node is None:
            return AVLNode(key)
        if key < node.key:
            node.left = self._insert(node.left, key)
        else:
            node.right = self._insert(node.right, key)

        # Adjust heights of nodes after insertion and check balancing condition
```

```
        height_left = self._get_height(node.left)
        height_right = self._get_height(node.right)
        node.set_height(1 + max(height_left, height_right))
        bal_factor = height_left - height_right

        # Perform rotations if required
        if bal_factor > 1:
            # LL or LR
            if key < node.left.key:
                return self._rotate_right(node)

            node.left = self._rotate_left(node.left)
            return self._rotate_right(node)
        if bal_factor < -1:
            # RR or RL
            if key >= node.right.key:
                return self._rotate_left(node)

            node.right = self._rotate_right(node.right)
            return self._rotate_left(node)

        return node

    def _rotate_left(self, node):
        right_tree = node.right
        node.right = right_tree.left
        right_tree.left = node

        # Reset heights
        node.set_height(1 + max(self._get_height(node.left), self._get_height(node.right)))
        right_tree.set_height(1 + max(self._get_height(right_tree.left),
                                      self._get_height(right_tree.right)))

        return right_tree

    def _rotate_right(self, node):
        left_tree = node.left
        node.left = left_tree.right
        left_tree.right = node

        # Reset heights
        node.height = 1 + max(self._get_height(node.left), self._get_height(node.right))
        left_tree.height = 1 + max(self._get_height(left_tree.left),
                                   self._get_height(left_tree.right))
```

```python
        return left_tree

    def insertion_steps_and_rotation(self, key):
        """
        Perform an insertion of a key into the tree and return the number
        of steps taken and whether a rotation was performed or not.

        Parameters:
        - key: The key to be inserted into the tree.

        Returns:
        A tuple containing the number of steps taken during the
        insertion process and 1 if a rotation occured or 0 otherwise.
        """
        self.root, steps, rotation = self._insert_steps_and_rotation(self.root,␣
↪key)
        return (steps, rotation)

    def _insert_steps_and_rotation(self, node, key):
        # Rec cases
        if node is None:
            return (AVLNode(key), 0, 0)
        if key < node.key:
            (node.left, steps, rotation) = self._insert_steps_and_rotation(node.
↪left, key)
        else:
            (node.right, steps, rotation) = self.
↪_insert_steps_and_rotation(node.right, key)

        # Adjust heights of nodes after insertion and check balancing condition
        height_left = self._get_height(node.left)
        height_right = self._get_height(node.right)
        node.set_height(1 + max(height_left, height_right))
        bal_factor = height_left - height_right

        # Perform rotations if required
        if bal_factor > 1:
            # LL or LR
            if key < node.left.key:
                return (self._rotate_right(node), steps + 1, rotation + 1)

            node.left = self._rotate_left(node.left)
            return (self._rotate_right(node), steps + 1, rotation + 1)
        if bal_factor < -1:
            # RR or RL
            if key >= node.right.key:
                return (self._rotate_left(node), steps + 1, rotation + 1)
```

```
            node.right = self._rotate_right(node.right)
            return (self._rotate_left(node), steps + 1, rotation + 1)

        return (node, steps + 1, rotation)

    def is_avl_tree(self):
        return self._is_avl_tree(self.root)

    def _is_avl_tree(self, node):
        # subtree is empty
        if not node:
            return True

        # check node has correct height
        height_left = self._get_height(node.left)
        height_right = self._get_height(node.right)
        if node.height != 0:
            if node.height != 1 + max(height_left, height_right):
                return False

        # check balance factor of the node
        bal_factor = height_left - height_right
        if not (bal_factor >= -1 and bal_factor <= 1):
            return False

        # check circular references
        if node.left is node or node.right is node:
            return False

        left_tree = self._is_avl_tree(node.left)
        right_tree = self._is_avl_tree(node.right)

        return all([left_tree, right_tree])
```

1.6 Red Black Tree

The code for this section can be found below. On top of implementing insertion and extracting the statistics, I also implemented a method that checks whether the tree generated is an valid red black tree. This test was heavily inspired by [3].

1.6.1 Insertion

The method I have chosen for insertion of new nodes is the top-down insertion strategy discussed in the lecture notes. In a nutshell, this strategy never allows a red uncle to exist, adjusting any found when traversing the tree to the desired location.

1.6.2 Statistics

The statistics were gathered in a similar way to the AVL tree. They can be viewed below:

RB Tree Insertion Steps Statistics:
Minimum: 12
Maximum: 18
Mean: 14.591408591408591
Standard Deviation: 1.0667174480086627
Median: 15

RB Tree Rotations Statistics:
Minimum: 0
Maximum: 2
Mean: 0.42757242757242276
Standard Deviation: 0.5167194606350732
Median: 0

RB Tree Height: 16
RB Tree Leaves: 2557

[ ]:
```python
from BinaryTree import BinaryTree

class RedBlackNode:
    def __init__(self, key, is_red=True, parent=None):
        self.key = key
        self.red = is_red
        self.left = None
        self.right = None
        self.parent = parent

    def is_red(self):
        return self.red

class RedBlackTree(BinaryTree):
    def insert(self, key):
        self._insert(self.root, key)

    def _insert(self, node, key):
        parent = None
        current_node = node
        while True:
            # insert here
            if current_node is None:
                current_node = RedBlackNode(key, parent=parent)
                # new node is root
                if not current_node.parent:
                    current_node.red = False
                    self.root = current_node
```

```
            else:
                # set parents pointer to new node
                if current_node.key < parent.key:
                    parent.left = current_node
                else:
                    parent.right = current_node
                # check for conflicts
                if parent.red:
                    self._resolve_problems(current_node)

            break

        parent = current_node.parent

        # remove red uncles
        if not current_node.red:
            # black node with red children
            if current_node.left and current_node.left.red:
                if current_node.right and current_node.right.red:
                    current_node.left.red = False
                    current_node.right.red = False
                    if parent:
                        current_node.red = True

                    # check for red red violations and then rotate
                    if parent and parent.red:
                        self._resolve_problems(current_node)

        parent = current_node
        if key < current_node.key:
            current_node = current_node.left
        else:
            current_node = current_node.right

def _left_rotate(self, node):
    right_child = node.right
    node.right = right_child.left

    if right_child.left:
        right_child.left.parent = node

    right_child.parent = node.parent

    if not node.parent:
        self.root = right_child
    elif node == node.parent.left:
        node.parent.left = right_child
```

```python
        else:
            node.parent.right = right_child

        right_child.left = node
        node.parent = right_child

    def _right_rotate(self, node):
        left_child = node.left
        node.left = left_child.right

        if left_child.right:
            left_child.right.parent = node

        left_child.parent = node.parent

        if not node.parent:
            self.root = left_child
        elif node == node.parent.left:
            node.parent.left = left_child
        else:
            node.parent.right = left_child

        left_child.right = node
        node.parent = left_child

    def _resolve_problems(self, node):
        parent = node.parent
        grandparent = parent.parent

        # check for inside
        if parent is grandparent.left and node is parent.right:
            self._left_rotate(parent)
            parent = node
        elif parent is grandparent.right and node is parent.left:
            self._right_rotate(parent)
            parent = node

        # check for outside
        if parent is grandparent.left:
            self._right_rotate(grandparent)
            parent.red = not parent.red
            grandparent.red = not grandparent.red
        elif parent is grandparent.right:
            self._left_rotate(grandparent)
            parent.red = not parent.red
            grandparent.red = not grandparent.red
```

```python
        if not parent.parent:
            self.root = parent
            parent.red = False

    def insertion_steps_and_rotation(self, key):
        return self._insertion_steps_and_rotation(self.root, key, 0, 0)

    def _insertion_steps_and_rotation(self, node, key, steps, rotations):
        parent = None
        current_node = node
        while True:
            steps += 1
            # insert here
            if current_node is None:
                current_node = RedBlackNode(key, parent=parent)
                # new node is root
                if not current_node.parent:
                    current_node.red = False
                    self.root = current_node
                else:
                    # set parents pointer to new node
                    if current_node.key < parent.key:
                        parent.left = current_node
                    else:
                        parent.right = current_node
                    # check for conflicts
                    if parent.red:
                        self._resolve_problems(current_node)
                        rotations += 1

                break

            parent = current_node.parent

            # remove red uncles
            if not current_node.red:
                # black node with red children
                if current_node.left and current_node.left.red:
                    if current_node.right and current_node.right.red:
                        current_node.left.red = False
                        current_node.right.red = False
                        if parent:
                            current_node.red = True

                        # check for red red violations and then rotate
                        if parent and parent.red:
                            self._resolve_problems(current_node)
```

```
                            rotations += 1

            parent = current_node
            if key < current_node.key:
                current_node = current_node.left
            else:
                current_node = current_node.right

        return (steps, rotations)

    def is_rb_tree(self):
        return self._is_rb_tree(self.root)[0]

    def _is_rb_tree(self, node):
        if not node:  # if node is a leaf, check #3
            return True, 1

        if not node.parent and node.red:  # If node is the root, check #2
            return False, 0

        if node.red:  # if node is red, check #4
            n_blacks = 0
            if (node.left and node.left.red) or (node.right and node.right.red):
                return False, -1
        else:  # else, the number of black nodes to the leaves includes the␣
↪same node
            n_blacks = 1

        # Check the subtrees for #5
        right, n_blacks_right = self._is_rb_tree(node.right)
        left, n_blacks_left = self._is_rb_tree(node.left)

        return all([right, left, n_blacks_right == n_blacks_left]),␣
↪n_blacks_right + n_blacks

    def get_height(self):
        return self._get_height(self.root)

    def _get_height(self, node):
        if not node:
            return 0

        left_height = self._get_height(node.left)
        right_height = self._get_height(node.right)
        return 1 + max(left_height, right_height)
```

1.7 Skip Lists

the implementation for this section can be viewed below. It was heavily inspired by [4-7] along with the lecture notes.

1.7.1 Insertion

Insertion is done by following the following steps:

1. Create a New Node: It initializes a new node (new_node) with a value and a random height.
2. Update Max Height and Head: It updates the maximum height of the skip list and ensures that the head node's next and previous lists are long enough to accommodate the new node's height.
3. Find Insertion Point: Starting from the highest level, it traverses the skip list to find the correct position for the new node. It moves forward at each level until it finds the right spot where the current node's next value is greater than or equal to the new node's value.
4. Update Pointers: For each level up to the new node's height, it adjusts the next and previous pointers to insert the new node. If the new node isn't at the end of the list at a given level, it also updates the previous pointer of the next node to point back to the new node.

1.7.2 Statistics

The insertion statistics are as follows: *Skip List Insertion Steps Statistics:*
Minimum: 3
Maximum: 25
Mean: 12.952047952047952
Standard Deviation: 3.475586037159532
Median: 13

Skip List Promotions Statistics:
Minimum: 0
Maximum: 11
Mean: 1.040959040959041
Standard Deviation: 1.4259455386937747
Median: 1

Skip List Levels: 14

[ ]:
```
from random import randint

class SkipNode:
    def __init__(self, height = 1, value = None):
        self.value = value
        self.next = [None] * height
        self.previous = [None] * height

    def __lt__(self, other):
        if isinstance(other, SkipNode):
            return self.value < other.value
        elif isinstance(other, int):
```

```python
            return self.value < other
        return NotImplemented

    def __le__(self, other):
        if isinstance(other, SkipNode):
            return self.value <= other.value
        elif isinstance(other, int):
            return self.value <= other
        return NotImplemented

    def __gt__(self, other):
        if isinstance(other, SkipNode):
            return self.value > other.value
        elif isinstance(other, int):
            return self.value > other
        return NotImplemented

    def __ge__(self, other):
        if isinstance(other, SkipNode):
            return self.value >= other.value
        elif isinstance(other, int):
            return self.value >= other
        return NotImplemented

    def __eq__(self, other):
        if isinstance(other, SkipNode):
            return self.value == other.value
        elif isinstance(other, int):
            return self.value == other
        return NotImplemented

    def __ne__(self, other):
        if isinstance(other, SkipNode):
            return self.value != other.value
        elif isinstance(other, int):
            return self.value != other
        return NotImplemented

class Head(SkipNode):
    def __lt__(self, other):
        return True

    def __le__(self, other):
        return True

    def __gt__(self, other):
        return False
```

```
    def __ge__(self, other):
        return False

    def __eq__(self, other):
        return False

    def __ne__(self, other):
        return True

class SkipList:
    def __init__(self):
        self.head = Head()
        self.len = 0
        self.max_height = 0

    def __len__(self):
        return self.len

    def _get_new_height(self):
        height = 1
        while randint(0, 1) == 0:
            height += 1
        return height

    def insert(self, value):
        new_node = SkipNode(self._get_new_height(), value)
        head = self.head

        # update max height and head next values
        self.max_height = max(self.max_height, len(new_node.next))
        while len(head.next) < len(new_node.next):
            head.next.append(None)
            head.previous.append(None)

        # find the correct place at each level
        current_node = self.head
        for level in reversed(range(self.max_height)):
            while current_node.next[level] and current_node.next[level] < value:
                current_node = current_node.next[level]

            # update next and previous pointers
            if level < len(new_node.next):
                new_node.previous[level] = current_node
                new_node.next[level] = current_node.next[level]
                current_node.next[level] = new_node
                # Node isn't at the end of a list
```

```python
                if new_node.next[level]:
                    next_node = new_node.next[level]
                    next_node.previous[level] = new_node

    def insert_steps_and_promotions(self, value):
        steps = 0
        promotions = self._get_new_height()
        new_node = SkipNode(promotions, value)
        head = self.head

        # update max height and head next values
        self.max_height = max(self.max_height, len(new_node.next))
        while len(head.next) < len(new_node.next):
            head.next.append(None)
            head.previous.append(None)

        # find the correct place at each level
        current_node = self.head
        for level in reversed(range(self.max_height)):
            while current_node.next[level] and current_node.next[level] < value:
                current_node = current_node.next[level]
                steps += 1

            # update next and previous pointers
            if level < len(new_node.next):
                new_node.previous[level] = current_node
                new_node.next[level] = current_node.next[level]
                current_node.next[level] = new_node
                # Node isn't at the end of a list
                if new_node.next[level]:
                    next_node = new_node.next[level]
                    next_node.previous[level] = new_node

        # promotions - 1 because by default it will be in the bottom list
        return (steps, promotions - 1)
```

1.8 Statistical Analyisis

My answer to the question "which data structure would you implement in real life" is not a simple x or y. The choice depends on specific use-case requirements such as insertion speed, search efficiency, and ease of implementation.

Each data structure has its strengths:

- AVL Tree: Best for strict balancing and predictable performance.
- Red-Black Tree: Good for guaranteed balanced operations with slightly higher complexity.
- Skip List: Suitable for simpler implementation and concurrent access scenarios with probabilistic balance.

Therefore, from my statistics I would recommend the following:

- AVL Tree: Suitable for scenarios requiring consistently balanced trees with efficient search and insert operations, particularly where minimal rotations are desirable.
- Red-Black Tree: Ideal for applications needing guaranteed balancing with slightly higher complexity but ensuring balanced structure.
- Skip List: Beneficial for applications needing fast average-case performance with simpler implementation and probabilistic balancing, especially in concurrent settings.

1.9 References

[1] "Knuth shuffle," Rosetta Code, Mar. 23, 2023. https://rosettacode.org/wiki/Knuth_shuffle

[2] P. Grafov, "pgrafov/python-avl-tree," GitHub, Feb. 26, 2024. https://github.com/pgrafov/python-avl-tree/tree/master (accessed May 30, 2024).

[3] 262588213843476, "Check if a tree is a balanced red-black tree. O(n) complexity.," Gist. https://gist.github.com/aldur/8c061c88b0f58e871776 (accessed May 30, 2024).

[4] "Skip List | Set 2 (Insertion)," GeeksforGeeks, Jul. 05, 2017. https://www.geeksforgeeks.org/skip-list-set-2-insertion/

[5] "sachinnair90's gists," Gist. https://gist.github.com/sachinnair90/ (accessed May 30, 2024).

[6]262588213843476, "Simple Skip list implementation in Python," Gist. https://gist.github.com/sachinnair90/3bee2ef7dd3ff0dc5aec44ec40e2d127 (accessed May 30, 2024).

[7]MIT OpenCourseWare, "7. Randomization: Skip Lists," YouTube. Mar. 04, 2016. Available: https://www.youtube.com/watch?v=2g9OSRKJuzM