EECS2101 Assignment3 Solution

Huanrui Cao **219256809** **saikoro@my.yorku.ca**

Problem 1 . Card Shuffle:

```
Algorithm card_shuffle(list):
    Input: origin list
    Output: shuffled list

    n ← len(list) // 2  # Find the midpoint of the list

    # Reverse the second half of the list
    reverse_sublist(list, n, len(list) - 1)

    # Interleave the elements from the first and reversed second halves
    interleave(list, 0, n)
    return the shuffled list

# Helper method to reverse a sublist in-place
Algorithm reverse_sublist(list, start, end):
    Input: list, sublist start index, sublist end index
    Output: reversed sublist

    while start < end:
        list[start], list[end] ← list[end], list[start]
        start ← start + 1
        end ← end - 1

# Helper method to interleave two halves of the list in-place
Algorithm interleave(list, start, mid):
    Input: list, start index, mid index
    Output: interleave list
    while mid < len(list):
        list.insert(start + 1, list.pop(mid))
        start ← start + 2
        mid ← mid + 1a
```

Explanation:

1. **Finding the Midpoint:** Calculate the midpoint `n` of the list. This will be used to split the list into two halves.
2. **Reverse the Second Half:** Reverse the second half of the list in-place. This is done to make it easier to interleave the elements later.
3. **Interleave Elements:** Interleave the elements from the first and reversed second halves. This step involves swapping and inserting elements to achieve the card shuffle.

Correctness:

- Reversing the second half ensures that when interleaved, the elements will be in the correct order.
- Interleaving is done by inserting elements from the second half after every element in the first half, resulting in a shuffled order.

Time Analysis:

- Finding the midpoint takes $\mathcal{O}(1)$ time.
- Reversing a sublist takes $\mathcal{O}(n/2) = \mathcal{O}(n)$ time.
- Interleaving involves inserting $\frac{n}{2}$ elements in the worst case, and each insertion takes $\mathcal{O}(1)$ time. So, interleaving takes $\mathcal{O}(n)$ time.

The overall time complexity of the algorithm is $\mathcal{O}(n)$, and it works in-place using only $\mathcal{O}(1)$ additional memory cells.

Problem 2 . Binary Tree Node Balance Factors:

```
Algorithm eulerTourWithBalanceFactors(T, p):
   Input: Binary tree T with root at position p
   Output: List of pairs (node data, balance factor)

   Method updateHeights(T, p):
      // Update the heights of left and right subtrees for the given node
      if p is not null then
         if p has a left child lc then
            node.height_left ← max(lc.height_left, lc.height_right) + 1
         if p has a right child lc then
            node.height_right ← max(rc.height_left, rc.height_right) + 1

   Method eulerTour(T, p):
      // Perform Euler tour traversal to print balance factors
      if p is not null then
         // "Pre visit" action: Update heights
         updateHeights(T, p)
         // Recursively tour the left subtree
         if p has a left child lc then
            eulerTour(T, lc)
         // "In visit" action: Print balance factor
         balance_factor ← node.height_left - node.height_right
         print balance_factor
```

```
        // Recursively tour the right subtree
        if p has a right child lc then
            eulerTour(T, rc)
        // "Post visit" action: (optional)

    // Initialize an empty list to store (node data, balance factor) pairs
    balance_factors = []

    // Start Euler tour traversal from the root of the binary tree
    eulerTour(T, p)

    // Return the list of balance factors
    return balance_factors
```

Explanation:

1. **Balance Factor Calculation:** In the Euler tour traversal, for each internal node, we calculate and print the balance factor, which is the difference between the heights of the right and left subtrees.
2. **Height Calculation:** The height of a subtree rooted at a position `p` is calculated recursively. The height is the maximum of the heights of the left and right subtrees plus 1.

Time Complexity Analysis:

- The Euler tour traversal visits each internal node exactly once, and for each internal node, it calculates the balance factor. The time complexity is $\mathcal{O}(n)$, where n is the number of internal nodes in the binary tree.
- Calculating the height of each subtree takes $\mathcal{O}(1)$ time per node during the Euler tour traversal.

The overall time complexity of the algorithm is $\mathcal{O}(n)$, where n is the number of internal nodes in the binary tree.

Problem 3. Priority Search Tree

we can perform a modified post-order traversal of the tree while maintaining information about the highest y-coordinate point in $S(v)$ for each node v

```
Algorithm buildPrioritySearchTree(T):
    Input: Priority search tree T
    Output: None

    Method postOrder(node):
        // Post-order traversal to compute highest_y values
        if node is not null then
            postOrder(node.left)
            postOrder(node.right)
```

```
        if node.left is not null and node.right is not null then
                // For internal nodes, set highest_y to the max of highest_y
values in children
                node.highest_y = max(node.left.highest_y,
node.right.highest_y)
            else
                // For leaves, set highest_y to the y-coordinate of the point
                node.highest_y = node.point.y

    // Start the post-order traversal from the root of the tree
    postOrder(T.root)

// Example usage:
// Create a complete binary tree T
T = TreeNode(
    point=Point(4, 10),
    left=TreeNode(point=Point(2, 5), left=TreeNode(point=Point(1, 3)),
right=TreeNode(point=Point(3, 7))),
    right=TreeNode(point=Point(6, 12), left=TreeNode(point=Point(5, 8)),
right=TreeNode(point=Point(7, 15)))
)

// Build the priority search tree
buildPrioritySearchTree(T)
```

Explanation:

1. **Post-order Traversal:** The algorithm performs a post-order traversal of the binary tree T. This ensures that for each internal node, the highest y-coordinate values in its children are already computed.
2. **Leaf Nodes:** For leaf nodes, the `highest_y` is set to the y-coordinate of the point stored in the leaf.
3. **Internal Nodes:** For internal nodes, the `highest_y` is set to the maximum of the `highest_y` values in its left and right children.

Time Complexity Analysis:

- The algorithm traverses each node in the binary tree exactly once, performing constant-time operations at each step. Therefore, the time complexity of the algorithm is linear, $\mathcal{O}(n)$, where n is the number of nodes in the tree.
- The algorithm maintains the `highest_y` information efficiently during the traversal, ensuring correctness for subsequent priority search tree operations.

This linear-time algorithm ensures the construction of a priority search tree from a complete binary tree in an efficient manner.