Andrew Taylor

Atayl136

Homework 7: More on Trees and Sorting

1. **Implement maketree, setleft, and setright for right in-threaded binary trees using the sequential array representation.**

This pseudocode is for creating a right in-threaded binary tree using a sequential array representation. The maketree function initializes a tree node with a given value, while the setleft and setright functions set the left and right children for a given tree node.

```
Def maketree(value):
    tree = [value, None, None, False] # node structure: [value, left_child, right_child, right_thread]
    return tree


def setleft(node, value=None):
            successor = node
            node[1] = [value, None, successor, True]



def setright(node, value):
    if node[2] is None: # only set right child if it is empty
        if value is None:
            if count(node[2]) > 1:   #checks for successor
                node[2][3] = True
                node[2][2] = successor node  # set thread to successor node
                node[2][0] = None
            else:
                node[2] = [None, None, None, False]
        else:
            successor = node[2]
            if count(node[2]) > 1 and node[3] is True:     #checks for successor/thread
```

node[2] = [value, None, successor, True]

node[3] = false

2. **Implement inorder traversal for the right in-thread tree in the previous problem.**

This pseudocode provides an inorder traversal algorithm for the right in-threaded binary tree created in the previous task. The inorder_traversal function visits each node in the tree following an inorder sequence (left child, node, right child). The leftmost node function returns the leftmost node in a subtree, and the visit function prints the value of the visited node.

```
Def inorder_traversal(tree):
  current_node = left_node(tree)

  while current_node is not None:
    visit(current_node)

    if current_node[3]: # If the current node has a right thread
      current_node = current_node[2]
    else:
      current_node = left_node(current_node[2]) #if not it's the end of the tree

def left_node(node):
  while node[1] is not None:
    node = node[1]
  return node

def visit(node):
  print(node[0])
```

3. **Sort using a counting method.**

This pseudocode presents a sorting algorithm based on counting smaller elements in the input array. The modified_counting_sort function calculates the number of smaller elements for each value in the

input array A and stores the counts in the array Count. Then, it populates the Output array with the sorted elements from the input array based on the counts. This algorithm can handle arrays with duplicate values. The size of the "Output" array should be equal to the size of the input array "A," which is "n." This is because the goal of the sorting algorithm is to rearrange the elements of the input array in sorted order, and the Output array will store the sorted elements. Therefore, the size of the Output array should be the same as the size of the input array, which is "n."

```
def modified_counting_sort(A):
    n = len(A)
    Count = [0] * n
    Output = [None] * n

    # Count smaller elements
    for i in range(n):
        count_smaller = 0
        for j in range(n):
            if A[j] < A[i]:
                count_smaller += 1
        Count[i] = count_smaller

    # Assign values to Output array
    for i in range(n):
        index = Count[i]
        while Output[index] is not None:
            index += 1
        Output[index] = A[i]

    # Check if the output is sorted
    is_sorted = True
    for i in range(1, n):
        if not Output[i] < Output[i - 1]:
```

```
            is_sorted = False
            break

    if is_sorted:
        break
    else:
        A = Output
        Output = [None] * n  # Reset Output array
        modified_counting_sort(A)
return Output
```

4. Analyze the cost of the sort in the previous problem.

The time complexity of this sorting method is O(n^2) because there are two nested loops. The space complexity is O(n) due to the use of Count and Output arrays. The impact of input data (random, ordered, or reverse ordered) does not affect the time complexity of the algorithm since we always compare each element with all others.

5. Number of comparisons to find the largest and smallest elements.

This pseudocode describes a method for finding the largest and smallest elements in an array without sorting it. The find_min_max function iterates through the input array, comparing each element to the current minimum and maximum values, and updates these values accordingly. This method requires only a single pass through the array and (n-1) comparisons to find both the largest and smallest elements.

```
def find_min_max(A):
    n = len(A)
    min_element = A[0]
    max_element = A[0]

    for i in range(1, n):
        if A[i] < min_element:
```

```
            min_element = A[i]
        elif A[i] > max_element:
            max_element = A[i]
    return min_element, max_element
```

In this approach, we can find the largest and smallest elements in a single pass through the array. The number of comparisons required is 2(n-1) since we compare each element (except the first one) with both the current minimum and maximum elements. This method is more efficient than sorting the array and then selecting the largest and smallest elements, as it does not require the extra overhead of sorting. In the best, worst, and average cases, the number of comparisons will always be 2(n-1) since we perform a single pass through the array. For a set of n distinct elements, this algorithm compares each element (except the first one) with the current minimum and maximum values. Perhaps a better idea would be to compare pairs of numbers assigning a temporary variable to each depending on which is smaller. This would result in only 3(n/2) comparisons needed.