Ibrahim Mohamed El Sayed

2000803

KH5003CEM

Advanced Algorithms

By Dr Mennatullah Gamil

Task 1

To remove duplicates from an array of characters:

1. We can use a hash set.

2. Iterate through the array of characters.

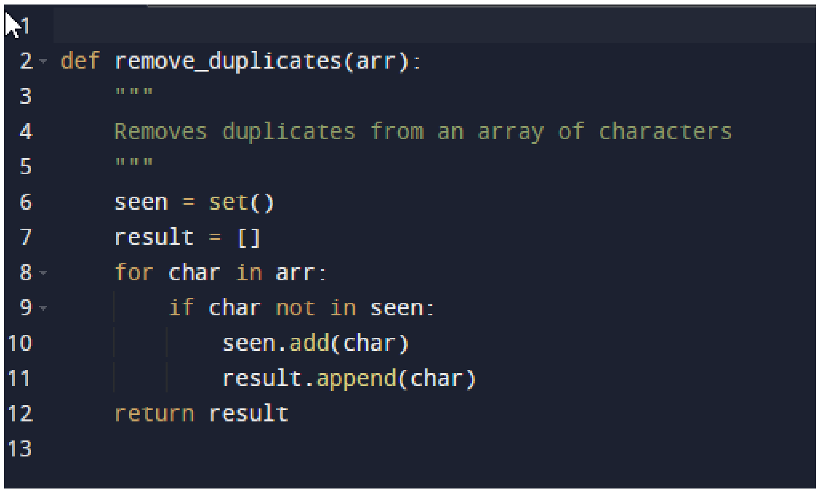
3. For each character, check if it is in the hash set.

4. If the character is not present in the hash set, add it and output it.

5. If the character is present in the hash set , skip it.

6. When all characters have been processed, the output array will contain only unique characters.

Python code that implements this algorithm:



```python

def remove_duplicates(arr):
    """
    Removes duplicates from an array of characters
    """
    seen = set()
    result = []
    for char in arr:
        if char not in seen:
            seen.add(char)
            result.append(char)
    return result

```

This algorithm has a time complexity of O(n), where n is the length of the input array,. The space complexity of this algorithm is also O(n) since we need to store all the unique characters in the result array. This function takes an array of characters as input and returns an array of unique characters.

Task 2

The algorithm for checking whether two arrays have the same set of numbers.

1. Sort both arrays .

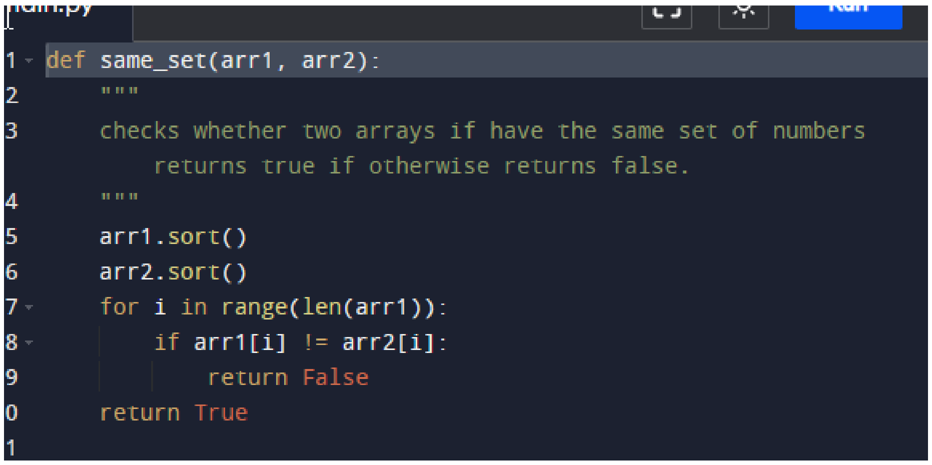
2. Iterate through both arrays .

3. If any elements differ, the arrays have different sets of numbers and the algorithm returns False.

4. If all elements are the same, the arrays have the same set of numbers and the algorithm returns True.

This algorithm has a time complexity of O(n log n), where n is the length of the input arrays, because creating a set of an array takes O(n) time.

Python code that implements this algorithm:



```python
def same_set(arr1, arr2):
    """
    checks whether two arrays if have the same set of numbers
        returns true if otherwise returns false.
    """
    arr1.sort()
    arr2.sort()
    for i in range(len(arr1)):
        if arr1[i] != arr2[i]:
            return False
    return True
```

Task 3

An ADT for a two-color, double-stack ADT:

1. The ADT consists of two stacks, one "red" and one "blue" with color coded versions of the regular stack ADT operation.

3. The ADT is implemented using a single array of size N, where N is the capacity of the ADT, which is assumed to always be larger than the sizes of the red and blue stacks combined.

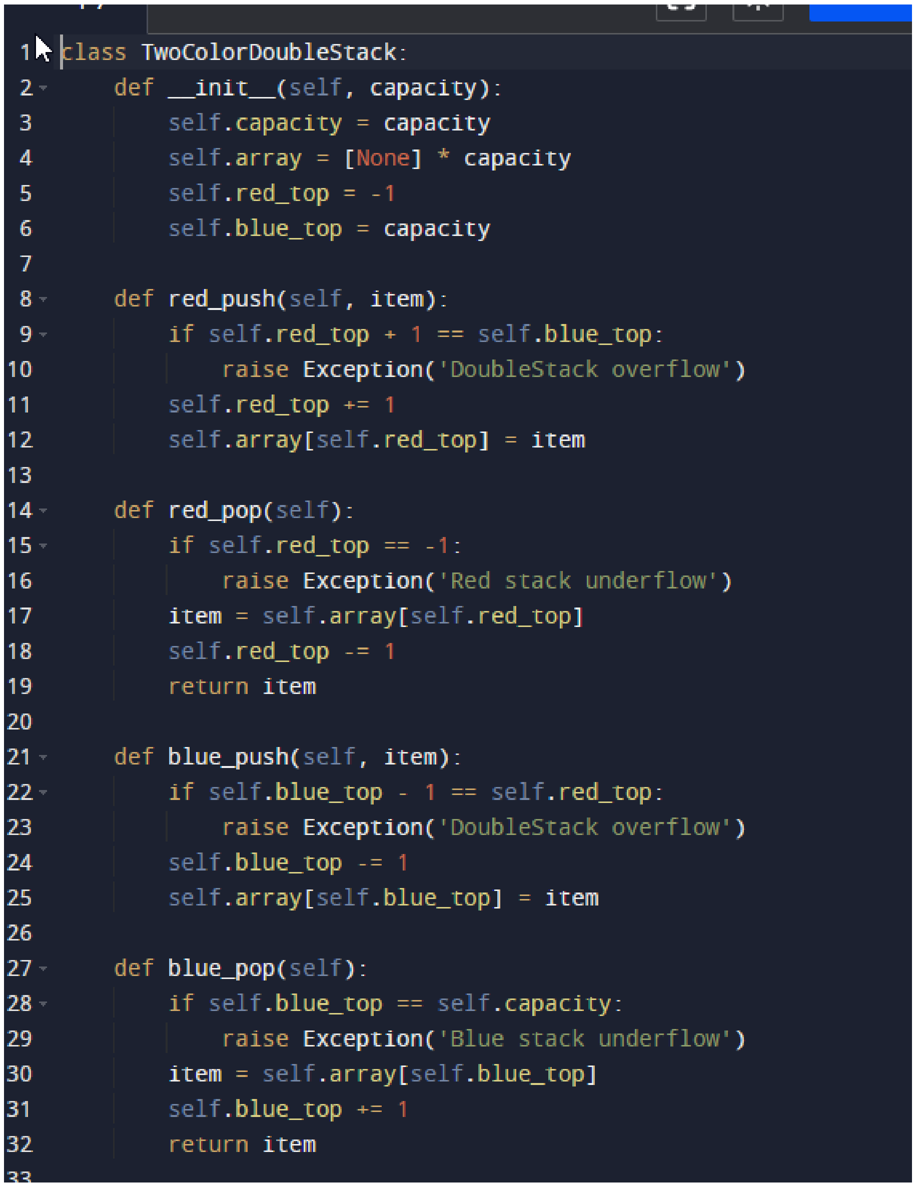
4. One efficient implementation for the two-color, double-stack ADT is to use a single array and two pointers to keep track of the top of each stack.

5. Each stack's elements can be color-coded to distinguish them.

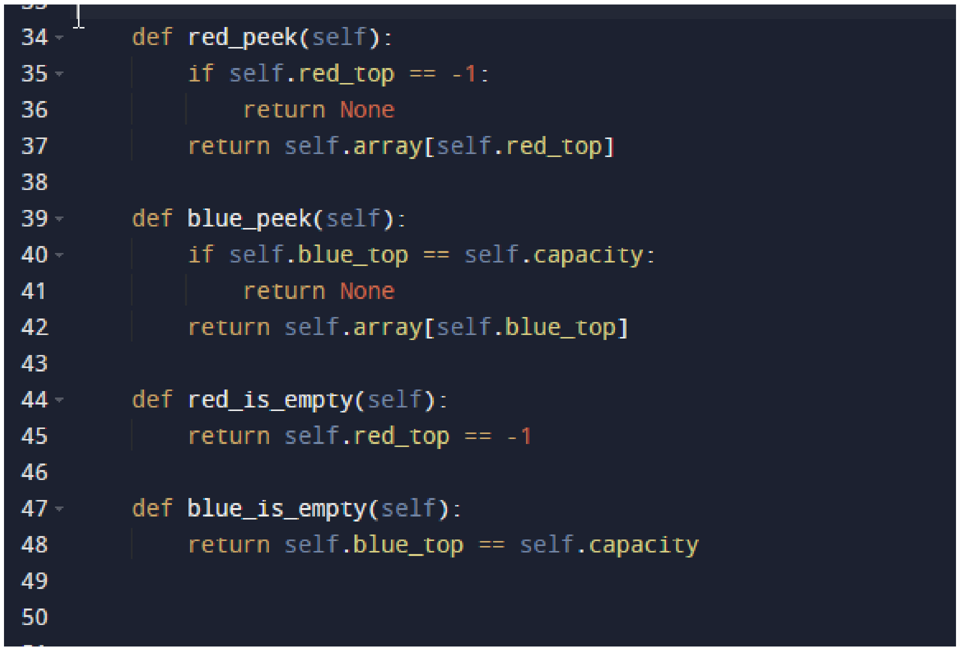
6. The push operation for each stack can be implemented by incrementing the corresponding pointer and adding the element at that index.

7.The pop operation can be implemented by decrementing the pointer and returning the element at that index. This implementation has a time complexity of O(1) for push and pop operations.

8. The space complexity of this implementation is O(N).



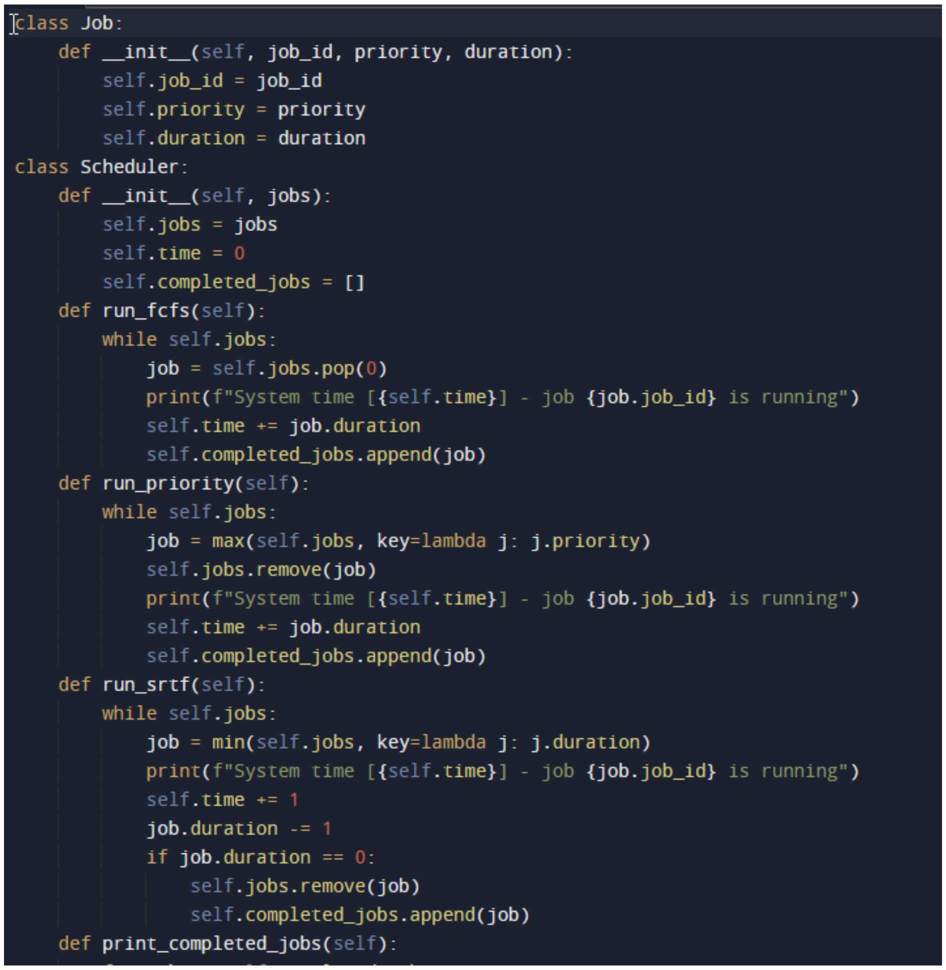
```python
class TwoColorDoubleStack:
    def __init__(self, capacity):
        self.capacity = capacity
        self.array = [None] * capacity
        self.red_top = -1
        self.blue_top = capacity

    def red_push(self, item):
        if self.red_top + 1 == self.blue_top:
            raise Exception('DoubleStack overflow')
        self.red_top += 1
        self.array[self.red_top] = item

    def red_pop(self):
        if self.red_top == -1:
            raise Exception('Red stack underflow')
        item = self.array[self.red_top]
        self.red_top -= 1
        return item

    def blue_push(self, item):
        if self.blue_top - 1 == self.red_top:
            raise Exception('DoubleStack overflow')
        self.blue_top -= 1
        self.array[self.blue_top] = item

    def blue_pop(self):
        if self.blue_top == self.capacity:
            raise Exception('Blue stack underflow')
        item = self.array[self.blue_top]
        self.blue_top += 1
        return item
```


```
    def red_peek(self):
        if self.red_top == -1:
            return None
        return self.array[self.red_top]

    def blue_peek(self):
        if self.blue_top == self.capacity:
            return None
        return self.array[self.blue_top]

    def red_is_empty(self):
        return self.red_top == -1

    def blue_is_empty(self):
        return self.blue_top == self.capacity
```

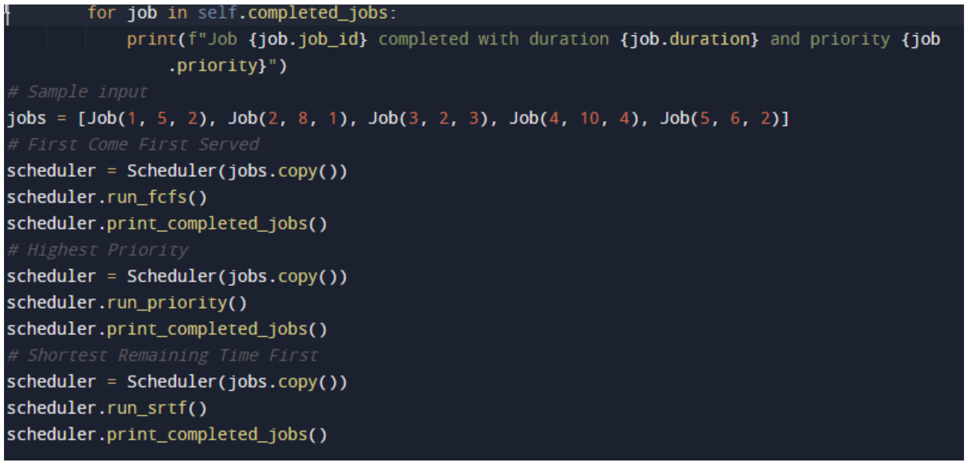
Task 4

1. In this implementation, the Job class represents a CPU job .
2. It has attributes for its job ID, priority, and duration.
3. The Scheduler class simulates the scheduling process and has methods for three different scheduling policies:
4. The algorithm to simulate the scheduling process
5. First Served (FCFS), Highest Priority, and Shortest Remaining Time First (SRTF).
6. The run_fcfs method implements the FCFS policy by executing jobs in the order they were added to the list of jobs.
7. The run_priority method implements the Highest Priority policy by executing jobs in order of decreasing priority.
8. The run_srtf method implements the SRTF policy by executing the job with the shortest remaining time first.
9. The print_completed_jobs method prints the completed jobs along with their durations and priorities.
10. Finally, the sample input is defined as a list of jobs, and the three scheduling policies are run on this list of jobs.

Python code that simulates the scheduling process of CPU



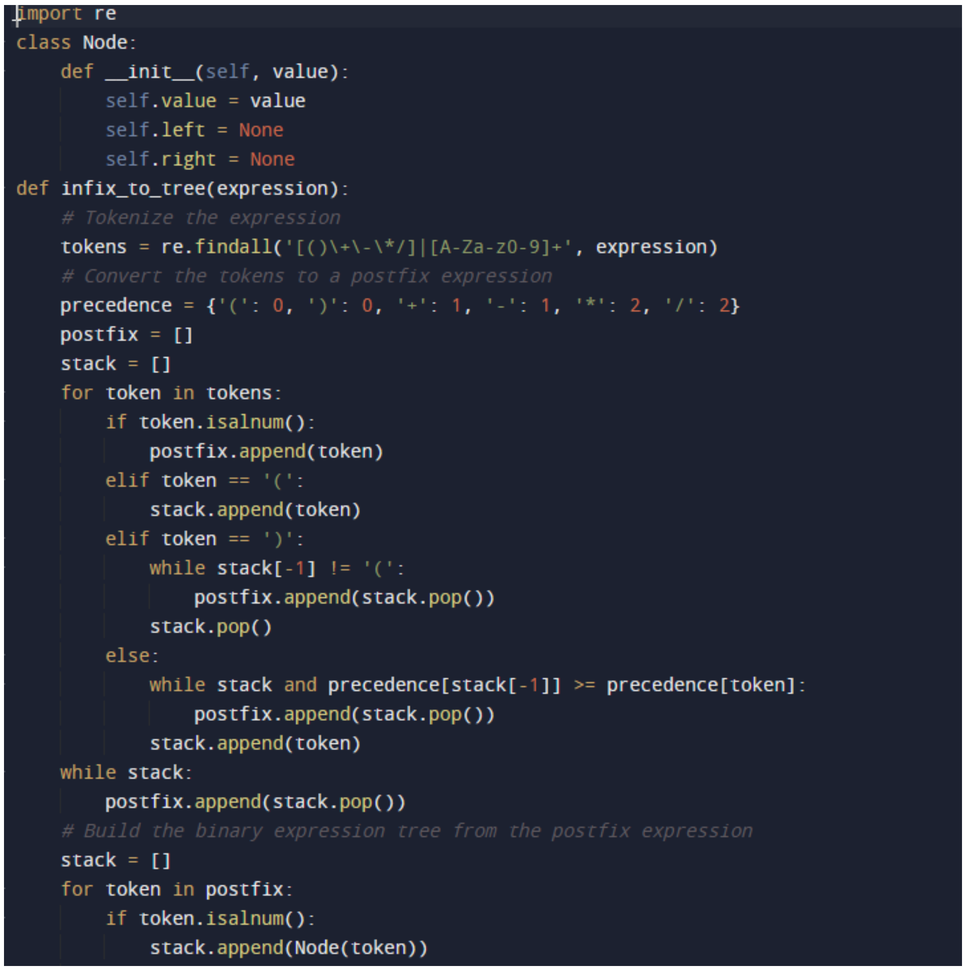
```python
class Job:
    def __init__(self, job_id, priority, duration):
        self.job_id = job_id
        self.priority = priority
        self.duration = duration
class Scheduler:
    def __init__(self, jobs):
        self.jobs = jobs
        self.time = 0
        self.completed_jobs = []
    def run_fcfs(self):
        while self.jobs:
            job = self.jobs.pop(0)
            print(f"System time [{self.time}] - job {job.job_id} is running")
            self.time += job.duration
            self.completed_jobs.append(job)
    def run_priority(self):
        while self.jobs:
            job = max(self.jobs, key=lambda j: j.priority)
            self.jobs.remove(job)
            print(f"System time [{self.time}] - job {job.job_id} is running")
            self.time += job.duration
            self.completed_jobs.append(job)
    def run_srtf(self):
        while self.jobs:
            job = min(self.jobs, key=lambda j: j.duration)
            print(f"System time [{self.time}] - job {job.job_id} is running")
            self.time += 1
            job.duration -= 1
            if job.duration == 0:
                self.jobs.remove(job)
                self.completed_jobs.append(job)
    def print_completed_jobs(self):
```



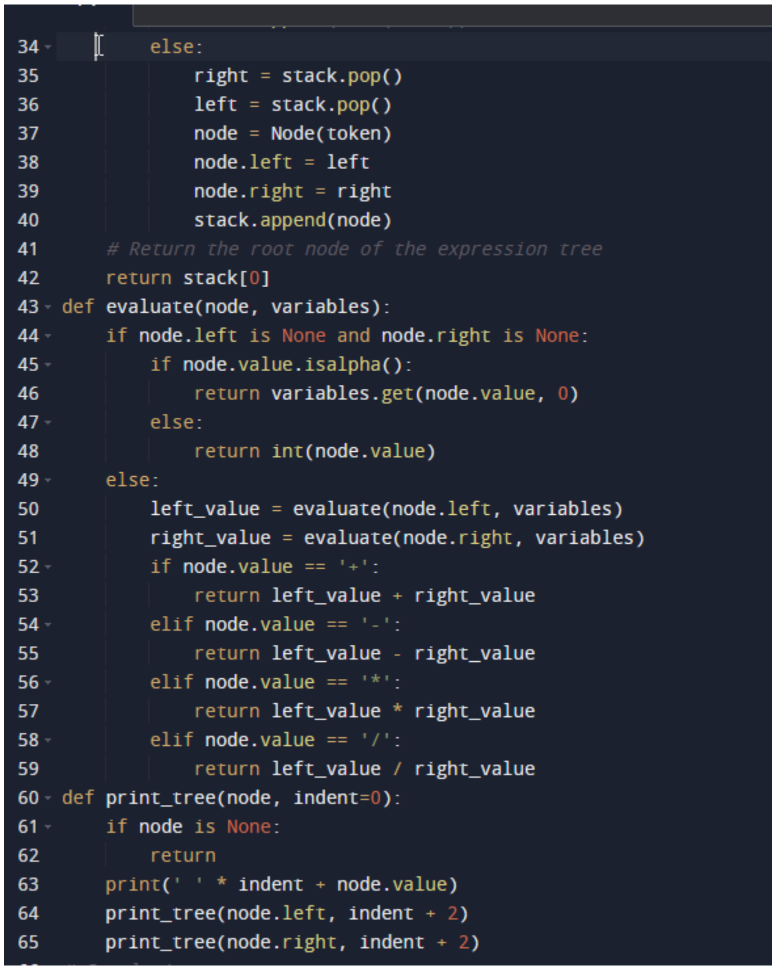
```
        for job in self.completed_jobs:
            print(f"Job {job.job_id} completed with duration {job.duration} and priority {job
                .priority}")
# Sample input
jobs = [Job(1, 5, 2), Job(2, 8, 1), Job(3, 2, 3), Job(4, 10, 4), Job(5, 6, 2)]
# First Come First Served
scheduler = Scheduler(jobs.copy())
scheduler.run_fcfs()
scheduler.print_completed_jobs()
# Highest Priority
scheduler = Scheduler(jobs.copy())
scheduler.run_priority()
scheduler.print_completed_jobs()
# Shortest Remaining Time First
scheduler = Scheduler(jobs.copy())
scheduler.run_srtf()
scheduler.print_completed_jobs()
```

Task 5

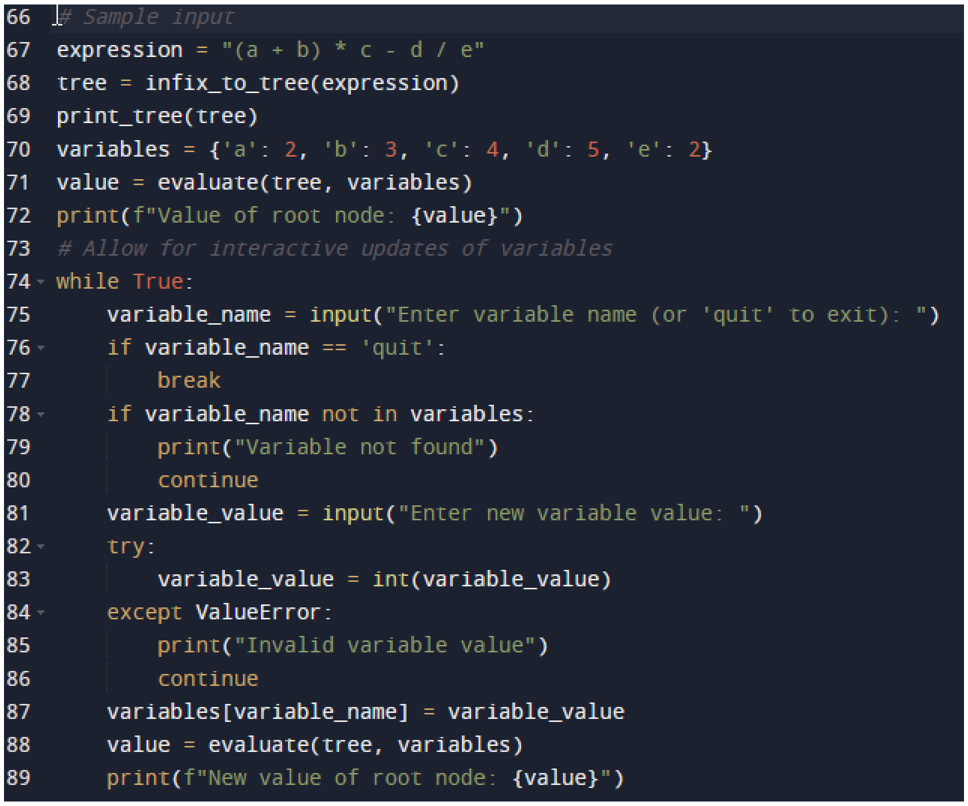
Python code that converts an infix expression to a binary expression tree and allows for interactive updates of variables.



```python
import re
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
def infix_to_tree(expression):
    # Tokenize the expression
    tokens = re.findall('[()\+\-\*/]|[A-Za-z0-9]+', expression)
    # Convert the tokens to a postfix expression
    precedence = {'(': 0, ')': 0, '+': 1, '-': 1, '*': 2, '/': 2}
    postfix = []
    stack = []
    for token in tokens:
        if token.isalnum():
            postfix.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            while stack[-1] != '(':
                postfix.append(stack.pop())
            stack.pop()
        else:
            while stack and precedence[stack[-1]] >= precedence[token]:
                postfix.append(stack.pop())
            stack.append(token)
    while stack:
        postfix.append(stack.pop())
    # Build the binary expression tree from the postfix expression
    stack = []
    for token in postfix:
        if token.isalnum():
            stack.append(Node(token))
```



```python
        else:
            right = stack.pop()
            left = stack.pop()
            node = Node(token)
            node.left = left
            node.right = right
            stack.append(node)
    # Return the root node of the expression tree
    return stack[0]
def evaluate(node, variables):
    if node.left is None and node.right is None:
        if node.value.isalpha():
            return variables.get(node.value, 0)
        else:
            return int(node.value)
    else:
        left_value = evaluate(node.left, variables)
        right_value = evaluate(node.right, variables)
        if node.value == '+':
            return left_value + right_value
        elif node.value == '-':
            return left_value - right_value
        elif node.value == '*':
            return left_value * right_value
        elif node.value == '/':
            return left_value / right_value
def print_tree(node, indent=0):
    if node is None:
        return
    print(' ' * indent + node.value)
    print_tree(node.left, indent + 2)
    print_tree(node.right, indent + 2)
```



```python
# Sample input
expression = "(a + b) * c - d / e"
tree = infix_to_tree(expression)
print_tree(tree)
variables = {'a': 2, 'b': 3, 'c': 4, 'd': 5, 'e': 2}
value = evaluate(tree, variables)
print(f"Value of root node: {value}")
# Allow for interactive updates of variables
while True:
    variable_name = input("Enter variable name (or 'quit' to exit): ")
    if variable_name == 'quit':
        break
    if variable_name not in variables:
        print("Variable not found")
        continue
    variable_value = input("Enter new variable value: ")
    try:
        variable_value = int(variable_value)
    except ValueError:
        print("Invalid variable value")
        continue
    variables[variable_name] = variable_value
    value = evaluate(tree, variables)
    print(f"New value of root node: {value}")
```

1. In this implementation, the Node class represents a node in the binary expression tree and has attributes for its value, left child, and right child.

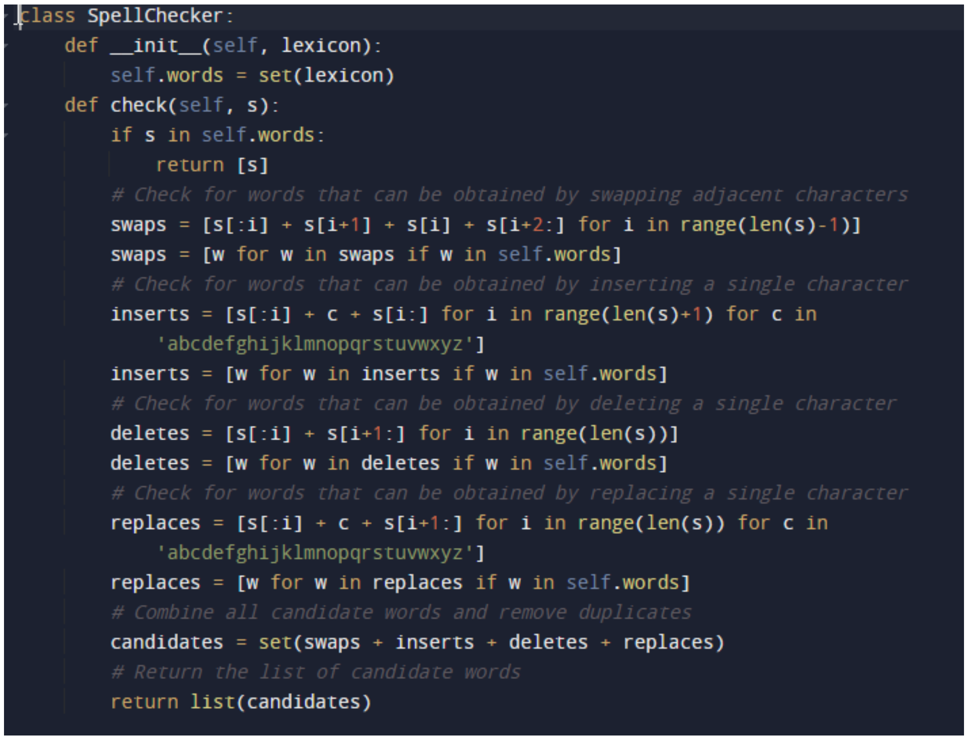
2. The infix_to_tree function converts the input infix expression to binary expression tree using the shunting-yard algorithm.

3. The evaluate function recursively evaluates the expression tree using a dictionary of variables, which are initially set to 0 and can be updated interactively by the user.

4. The print_tree function prints the binary expression tree using a pre-order traversal.
5. Finally, the sample input expression is converted to a binary expression tree and printed using the print_tree function.
6. The initial values of the variables are set in a dictionary, and the value of the root node is computed using the evaluate function and printed.
7. The program then enters a loop where the user can interactively update the values of the variables and see the corresponding change in the value of the root node.

Task 6

implementation of the spell-checker class in Python



```python
class SpellChecker:
    def __init__(self, lexicon):
        self.words = set(lexicon)
    def check(self, s):
        if s in self.words:
            return [s]
        # Check for words that can be obtained by swapping adjacent characters
        swaps = [s[:i] + s[i+1] + s[i] + s[i+2:] for i in range(len(s)-1)]
        swaps = [w for w in swaps if w in self.words]
        # Check for words that can be obtained by inserting a single character
        inserts = [s[:i] + c + s[i:] for i in range(len(s)+1) for c in
            'abcdefghijklmnopqrstuvwxyz']
        inserts = [w for w in inserts if w in self.words]
        # Check for words that can be obtained by deleting a single character
        deletes = [s[:i] + s[i+1:] for i in range(len(s))]
        deletes = [w for w in deletes if w in self.words]
        # Check for words that can be obtained by replacing a single character
        replaces = [s[:i] + c + s[i+1:] for i in range(len(s)) for c in
            'abcdefghijklmnopqrstuvwxyz']
        replaces = [w for w in replaces if w in self.words]
        # Combine all candidate words and remove duplicates
        candidates = set(swaps + inserts + deletes + replaces)
        # Return the list of candidate words
        return list(candidates)
```

1.The Spellchecker class takes a lexicon of words as input and stores it in a set.

2.W, in a set, and implements a method, check(s), which performs a spell check on the strings with respect to the set of words, W. If s is in W, then the call to check(s) returns a list containing only s, as it is assumed to be spelled correctly in this case. If s is not in W, then the call to checks) returns a list of every word in W that might be a correct spelling of s.

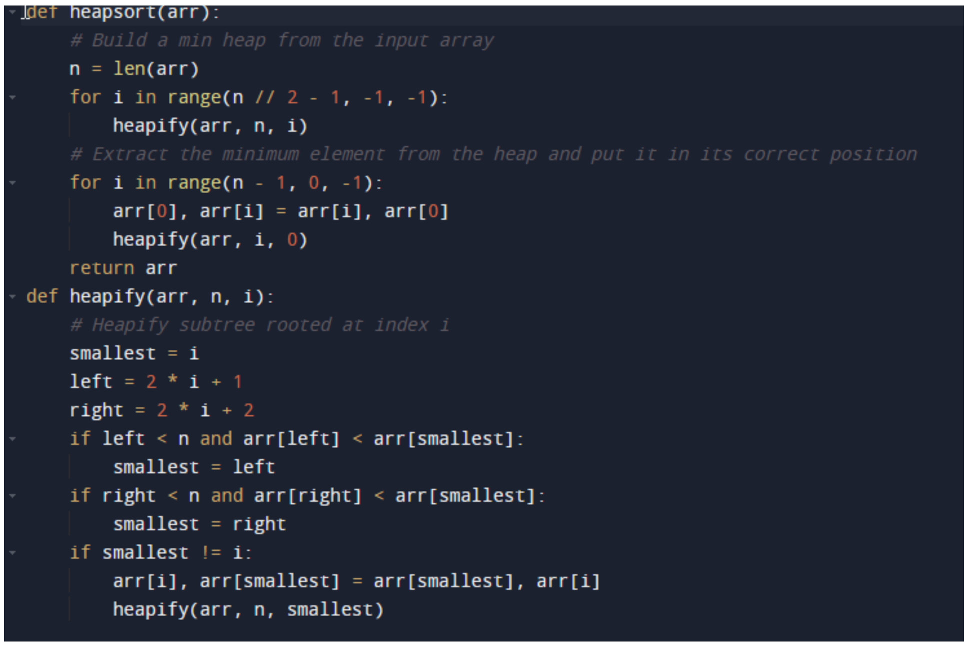
3. Your program should be able to handle all the common ways that s might be a misspelling of a word in W, including swapping adjacent characters in a word, inserting a single character in between two adjacent characters in a word, deleting a single character from a word, and replacing a character in a word with another character.

4. Levenshtein distance algorithm, which is a measure of the difference between two strings. The algorithm computes the minimum number of operations (insertion, deletion, or substitution of a character) needed to transform one string into another.

5. To adapt this algorithm for spell checking, we can compute the Levenshtein distance between the input string and each word in the lexicon, and return the words that have a distance below a certain threshold (e.g., 2 or 3).

Task 7

Python implementation of heapsort using a min heap:



```
def heapsort(arr):
    # Build a min heap from the input array
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    # Extract the minimum element from the heap and put it in its correct position
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)
    return arr
def heapify(arr, n, i):
    # Heapify subtree rooted at index i
    smallest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] < arr[smallest]:
        smallest = left
    if right < n and arr[right] < arr[smallest]:
        smallest = right
    if smallest != i:
        arr[i], arr[smallest] = arr[smallest], arr[i]
        heapify(arr, n, smallest)
```

1. The heapsort function takes a list arr as input and returns a sorted list using the heapsort algorithm.

2. The function first builds a min heap from the input array by calling the heapify function on each non-leaf node in reverse order.

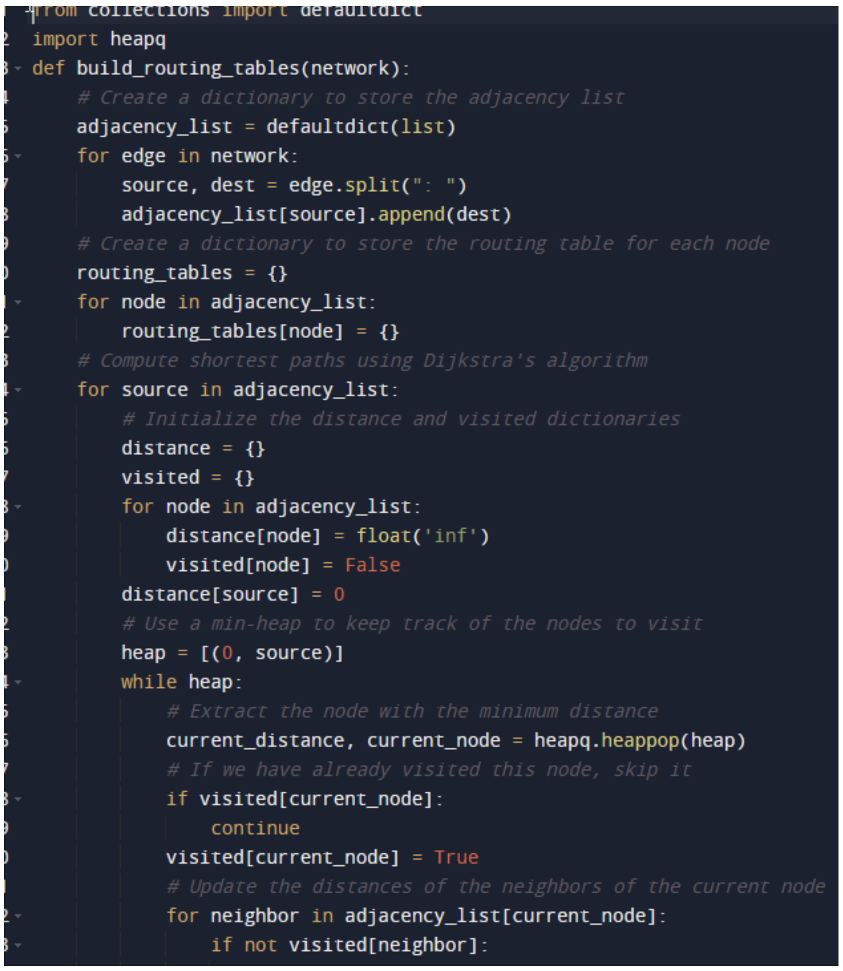
3. The heapify function takes an array, its length, and an index i as input, and recursively moves the element at index `i` down the heap until the subtree rooted at `i` satisfies the min heap property.

4. Once the min heap is built, the heapsort function repeatedly extracts the minimum element from the heap and puts it in its correct position by swapping it with the last element in the heap, reducing the heap size by 1, and calling heapify on the root of the heap.

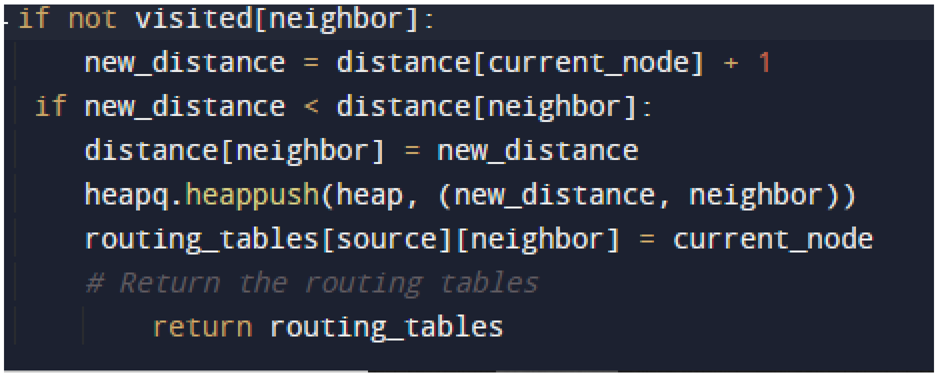
5. This process continues until the heap is empty, at which point the function returns .

Task 8

Python implementation of the program that builds routing tables for a computer network based on shortest-path routing:



```python
from collections import defaultdict
import heapq
def build_routing_tables(network):
    # Create a dictionary to store the adjacency list
    adjacency_list = defaultdict(list)
    for edge in network:
        source, dest = edge.split(": ")
        adjacency_list[source].append(dest)
    # Create a dictionary to store the routing table for each node
    routing_tables = {}
    for node in adjacency_list:
        routing_tables[node] = {}
    # Compute shortest paths using Dijkstra's algorithm
    for source in adjacency_list:
        # Initialize the distance and visited dictionaries
        distance = {}
        visited = {}
        for node in adjacency_list:
            distance[node] = float('inf')
            visited[node] = False
        distance[source] = 0
        # Use a min-heap to keep track of the nodes to visit
        heap = [(0, source)]
        while heap:
            # Extract the node with the minimum distance
            current_distance, current_node = heapq.heappop(heap)
            # If we have already visited this node, skip it
            if visited[current_node]:
                continue
            visited[current_node] = True
            # Update the distances of the neighbors of the current node
            for neighbor in adjacency_list[current_node]:
                if not visited[neighbor]:
```



```
if not visited[neighbor]:
    new_distance = distance[current_node] + 1
 if new_distance < distance[neighbor]:
    distance[neighbor] = new_distance
    heapq.heappush(heap, (new_distance, neighbor))
    routing_tables[source][neighbor] = current_node
    # Return the routing tables
        return routing_tables
```

1.The build_routing_tables function takes as input a list of node connectivity lists, where each list is of the form "Node A: B" and indicates that there is a direct connection from node A to node B .

2.The function uses Dijkstra's algorithm to compute the shortest paths from each node to every other node in the network, and builds the corresponding routing tables.

3.The function first constructs an adjacency list based on the connectivity information. It then initializes the routing table for each node as an empty dictionary.

4. For each node in the network, the function runs Dijkstra's algorithm to compute the shortest paths to all other nodes.

5. This is done by maintaining a min-heap of nodes to visit, and updating the distances of the neighbors of each node as we explore the network.

6.For each node, the function records the next node in the shortest path to every other node in the network in the routing table.

7. for each pair (source, dest), the function records the next node to send to (on the shortest path from source to dest) in the routing table for source.

8. Finally, the function returns the routing tables for all nodes in the network as a dictionary of dictionaries. Specifically, for each node, the function returns a dictionary that maps each destination node to the next node on the shortest path to that destination.