Python DSA my

Author : Arnab Dana

▼ Resource

```
https://github.com/campusx-official/dsa-using-python/blob/main/hashing_linear_probing.ipynb
```

▼ Content

i want to study string DSA
what are the topic

▼ 1. Roadmap

Phase 1: Python Fundamentals

1. **Basic Syntax and Operations**
    - Variables, data types, and operators (arithmetic, logical, bitwise).
    - Input/output functions.
    - Basic math and string operations.
2. **Control Structures**
    - Conditionals (`if`, `else`, `elif`).
    - Loops (`for`, `while`) and iteration patterns.
3. **Functions and Modules**
    - Function definitions, arguments, return statements.
    - Lambda functions, map, filter, and list comprehension.
    - Importing modules and using libraries.

4. **Error Handling and Debugging**
    - Understanding and handling exceptions with `try`, `except`, `finally`.
    - Debugging using tools like print statements and IDE debuggers.

Phase 2: Intermediate Python Skills

1. **Data Structures in Python**
    - Lists, tuples, sets, and dictionaries: creating, accessing, and modifying.
    - Stacks and queues using lists.
    - Basic operations: slicing, merging, searching, sorting.
2. **Object-Oriented Programming (OOP)**
    - Classes and objects, inheritance, polymorphism, encapsulation.
    - Special methods (like `__init__`, `__str__`).
    - Using OOP to organize code and handle complexity.
3. **File Handling**
    - Reading from and writing to files.
    - Handling different file formats (e.g., CSV, JSON).

Phase 3: Data Structures and Algorithms

1. **Understanding Basic Data Structures**
    - Arrays and linked lists.
    - Stacks, queues, hash tables (dictionaries).
    - Trees (binary trees, binary search trees) and graphs (representation in Python).
2. **Core Algorithms**
    - Sorting: Bubble, Insertion, Selection, Merge, and Quick Sort.
    - Searching: Linear search, Binary search.
    - Recursion fundamentals (Factorial, Fibonacci series, Tower of Hanoi).
3. **Complex Data Structures**

- Heaps, tries, AVL trees, and graphs (depth-first, breadth-first traversal).
- Advanced linked lists (doubly linked lists, circular linked lists).
- Implementing and using Python's `deque` and `heapq`.

4. **Dynamic Programming and Greedy Algorithms**
   - Dynamic programming (Memoization and tabulation, classic problems like Fibonacci, knapsack).
   - Greedy algorithms (coin change, activity selection, Huffman coding).
5. **Time Complexity and Space Complexity**
   - Big O notation, understanding the efficiency of algorithms.
   - Analyzing time complexity of various data structures and algorithms.

Phase 4: Practical Application and Problem-Solving

1. **Leetcode, HackerRank, CodeSignal Practice**
   - Practice data structures and algorithm problems on these platforms.
   - Focus on real-world problem-solving.
2. **Projects for Mastery**
   - **Data Structure Project**: Implement a custom data structure, like a mini-library for stacks, queues, or linked lists.
   - **Algorithm Project**: Develop a sorting algorithm visualizer or pathfinder (like BFS or DFS in a maze).
   - **Game or Simulation Project**: Build something simple like Tic-Tac-Toe, Sudoku, or even basic AI to implement algorithms in practice.
3. **Further Python Libraries and Modules**
   - Explore additional Python libraries such as NumPy, Pandas, and Matplotlib for data manipulation and visualization.

Phase 5: Advanced Algorithms (Optional)

1. **Graph Algorithms and Advanced DS**
   - Advanced graph algorithms (Dijkstra's, Floyd-Warshall, A*).
   - Segment trees, Fenwick trees, and other specialized data structures.

2. **Competitive Programming**
   - If interested in competitive programming, try tackling advanced problems on platforms like Codeforces or CodeChef.

▼ 2. Data Structures

1. **Arrays**
   - Single-dimensional Arrays
   - Multi-dimensional Arrays
   - Dynamic Arrays
   - Sparse Arrays
2. **Linked Lists**
   - Singly Linked List
   - Doubly Linked List
   - Circular Linked List
3. **Stacks**
   - Stack Implementation (Array & Linked List)
   - Stack Operations (Push, Pop, Peek)
   - Applications of Stacks (e.g., Expression Evaluation, Backtracking)
4. **Queues**
   - Queue Implementation (Array & Linked List)
   - Queue Operations (Enqueue, Dequeue, Front, Rear)
   - Circular Queue
   - Priority Queue
   - Double-ended Queue (Deque)
5. **Strings**
   - String Representation
   - String Matching Algorithms (e.g., KMP, Rabin-Karp)
   - String Manipulation Operations

6. **Hash Tables**
    - Hash Function
    - Collision Resolution (Chaining, Open Addressing)
    - Applications of Hash Tables
7. **Heaps**
    - Binary Heap
    - Min-Heap and Max-Heap
    - Heap Operations (Insert, Extract-Min/Max)
    - Heap Sort
8. **Binary Trees**
    - Binary Tree Structure
    - Tree Traversal (Inorder, Preorder, Postorder)
    - Tree Applications
9. **Binary Search Trees (BST)**
    - BST Properties
    - BST Operations (Insertion, Deletion, Search)
    - Balanced vs Unbalanced BST
10. **AVL Trees**
    - AVL Tree Properties
    - Rotations (Left, Right, Left-Right, Right-Left)
    - AVL Tree Operations
11. **Red-Black Trees**
    - Red-Black Tree Properties
    - Rotations in Red-Black Trees
    - Red-Black Tree Operations
12. **B-Trees**
    - B-Tree Properties

- B-Tree Operations (Insertion, Deletion, Search)
- B+ Trees and B* Trees

13. **Tries**
    - Trie Structure
    - Operations (Insertion, Search, Deletion)
    - Applications (e.g., Dictionary, Autocomplete)
14. **Graphs**
    - Graph Representation (Adjacency Matrix, Adjacency List)
    - Types of Graphs (Directed, Undirected, Weighted)
    - Graph Terminology (Vertex, Edge, Degree)
15. **Graph Traversal Algorithms**
    - Depth First Search (DFS)
    - Breadth First Search (BFS)
    - Applications of Graph Traversal
16. **Disjoint Set (Union-Find)**
    - Union and Find Operations
    - Path Compression
    - Union by Rank/Size
17. **Skip Lists**
    - Skip List Structure
    - Skip List Operations (Search, Insert, Delete)
    - Advantages over Linked Lists
18. **Bloom Filters**
    - Bloom Filter Structure
    - False Positives
    - Applications of Bloom Filters
19. **Fenwick Tree (Binary Indexed Tree)**

- Fenwick Tree Structure
- Fenwick Tree Operations (Update, Query)
- Applications in Range Queries

This provides a structured breakdown of each data structure along with its key subtopics! Would you like to explore any specific one in more detail?

▼ 3. Time and Space Complexity

▼ 4. Array

Index of Array Topics in Python

1. **Array Basics**
2. **Creating Arrays**
3. **Array Indexing**
4. **Array Operations**
5. **Array Methods**
6. **Multidimensional Arrays**
7. **Array Slicing**
8. **Array Iteration**
9. **Sorting Arrays**
10. **Array Concatenation**
11. **Array Reshaping**
12. **Array Filtering**
13. **Array Broadcasting**
14. **Memory Efficiency**
15. **Array Aggregation**
16. **Advanced Manipulations**

1. **Array Basics**

Arrays can be created using the `array` module or NumPy for better efficiency. Here's an example using the `array` module:

2. **Creating Arrays**

NumPy is widely used for array manipulation. Here's an example of creating an array using NumPy:

```
import numpy as np

# Creating a NumPy array
my_array = np.array([1, 2, 3, 4])

# Output the array
print(my_array)  # Output: [1 2 3 4]
```

3. **Array Indexing**

You can access array elements by indexing, similar to lists.

```
my_array = [10, 20, 30]
print(my_array[1])  # Output: 20
```

4. **Array Operations**

You can perform element-wise operations on NumPy arrays:

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Adding two arrays
result = arr1 + arr2
print(result)  # Output: [5 7 9]
```

5. **Array Methods**

Arrays come with several methods like `.append()`, `.remove()`, and `.insert()`.

```
import array as arr

my_array = arr.array('i', [1, 2, 3])
my_array.append(4)
```

```
# Output the array after appending
print(my_array)  # Output: array('i', [1, 2, 3, 4])
```

6. **Multidimensional Arrays**

NumPy arrays can be multidimensional (like matrices). Here's an example of creating and accessing a 2D array:

```
import numpy as np

# Creating a 2D array (matrix)
matrix = np.array([[1, 2], [3, 4]])

# Accessing an element
print(matrix[1][0])  # Output: 3
```

7. **Array Slicing**

You can slice arrays to access subarrays:

```
my_array = [10, 20, 30, 40, 50]
sliced = my_array[1:4]
print(sliced)  # Output: [20, 30, 40]
```

8. **Array Iteration**

You can iterate over the elements of an array:

```
for element in [10, 20, 30]:
    print(element)
# Output:
# 10
# 20
# 30
```

9. **Sorting Arrays**

Arrays can be sorted using `np.sort()` in NumPy:

```
import numpy as np

arr = np.array([3, 1, 2])
sorted_arr = np.sort(arr)
print(sorted_arr)  # Output: [1 2 3]
```

10. **Array Concatenation**

You can combine two or more arrays using `np.concatenate()`:

```
import numpy as np

arr1 = np.array([1, 2])
arr2 = np.array([3, 4])

# Concatenating the arrays
combined = np.concatenate((arr1, arr2))
print(combined)  # Output: [1 2 3 4]
```

11. **Array Reshaping**

You can change the shape of an array with `reshape()`:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

# Reshaping the array to 2×3
reshaped = arr.reshape(2, 3)
print(reshaped)
# Output:
# [[1 2 3]
#  [4 5 6]]
```

12. **Array Filtering**

You can filter elements in an array based on conditions:

```
import numpy as np

arr = np.array([10, 20, 30])
filtered = arr[arr > 15]
print(filtered)  # Output: [20 30]
```

13. **Array Broadcasting**

NumPy allows broadcasting, which is the ability to apply an operation on arrays of different shapes:

```
import numpy as np

arr = np.array([1, 2, 3])
result = arr + 10
print(result)  # Output: [11 12 13]
```

14. **Memory Efficiency**

NumPy arrays are more memory-efficient compared to regular Python lists:

```
import sys
import numpy as np

# List
my_list = list(range(1000))
print(sys.getsizeof(my_list))  # Output: size in bytes

# NumPy array
my_array = np.array(range(1000))
print(my_array.nbytes)  # Output: size in bytes
```

15. **Array Aggregation**

NumPy provides functions to compute aggregate values like sum, mean, etc.:

```
import numpy as np

arr = np.array([1, 2, 3])

# Sum of the array
print(arr.sum())  # Output: 6

# Mean of the array
print(arr.mean())  # Output: 2.0
```

16. **Advanced Manipulations**

Advanced manipulations like transposing arrays are simple in NumPy:

```
import numpy as np

matrix = np.array([[1, 2], [3, 4]])

# Transposing the matrix
transposed = np.transpose(matrix)
print(transposed)
# Output:
# [[1 3]
#  [2 4]]
```

These examples cover fundamental array topics in Python using lists and NumPy. Let me know if you'd like to dive deeper into any of these!

▼ 5. Strings

1. **Basic String Operations**

Example: String Reversal

```
def reverse_string(s):
    return s[::-1]
```

```
print(reverse_string("hello"))  # Output: "olleh"
```

2. **Pattern Matching Algorithms**

Example: Naive String Matching

```
def naive_search(text, pattern):
    n, m = len(text), len(pattern)
    for i in range(n - m + 1):
        if text[i:i + m] == pattern:
            print(f"Pattern found at index {i}")

naive_search("ababcabcab", "abc")  # Output: Pattern found at index 2
```

3. **String Searching Algorithms**

Example: Rabin-Karp Algorithm

```
def rabin_karp(text, pattern):
    d = 256  # Number of characters in the alphabet
    q = 101  # A prime number
    n, m = len(text), len(pattern)
    p_hash = 0  # Hash value for pattern
    t_hash = 0  # Hash value for text
    h = 1

    # Calculate the value of h (d^(m-1) % q)
    for i in range(m - 1):
        h = (h * d) % q

    # Calculate hash value for pattern and first window of text
    for i in range(m):
        p_hash = (d * p_hash + ord(pattern[i])) % q
        t_hash = (d * t_hash + ord(text[i])) % q

    # Slide the pattern over the text one by one
    for i in range(n - m + 1):
        if p_hash == t_hash:
```

```
            if text[i:i + m] == pattern:
                print(f"Pattern found at index {i}")
        if i < n - m:
            t_hash = (d * (t_hash - ord(text[i]) * h) + ord(text[i + m])) % q
            if t_hash < 0:
                t_hash += q

rabin_karp("ababcabcab", "abc")  # Output: Pattern found at index 2
```

4. **String Compression Algorithms**

Example: Run-Length Encoding

```
def run_length_encoding(s):
    encoding = ""
    count = 1
    for i in range(1, len(s)):
        if s[i] == s[i - 1]:
            count += 1
        else:
            encoding += s[i - 1] + str(count)
            count = 1
    encoding += s[-1] + str(count)
    return encoding

print(run_length_encoding("aaabbbcc"))  # Output: "a3b3c2"
```

5. **String Hashing**

Example: Polynomial Rolling Hash

```
def poly_hash(s, p=31, m=10**9 + 9):
    hash_value = 0
    p_pow = 1
    for c in s:
        hash_value = (hash_value + (ord(c) - ord('a') + 1) * p_pow) % m
        p_pow = (p_pow * p) % m
    return hash_value
```

```
print(poly_hash("abc"))  # Output: 294
```

6. **Dynamic Programming with Strings**

Example: Longest Common Subsequence (LCS)

```
def lcs(X, Y):
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0 or j == 0:
                dp[i][j] = 0
            elif X[i - 1] == Y[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]

print(lcs("ABCBDAB", "BDCAB"))  # Output: 4
```

7. **Advanced Topics**

Example: Suffix Array Construction

```
def suffix_array_construction(s):
    suffixes = [(s[i:], i) for i in range(len(s))]
    suffixes.sort()
    return [suffix[1] for suffix in suffixes]

print(suffix_array_construction("banana"))  # Output: [5, 3, 1, 0, 4, 2]
```

8. **Trie (Prefix Tree) Example**

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

trie = Trie()
trie.insert("hello")
print(trie.search("hello"))  # Output: True
print(trie.search("hell"))   # Output: False
```

▼ 6. Linked Lists

Index for Linked Lists

1. **Introduction to Linked Lists**
    - Node Definition

- LinkedList Class

2. **Basic Operations on Linked Lists**
   - Insertion at the beginning
   - Insertion at the end
   - Traversal
   - Deletion from the beginning
3. **Advanced Operations**
   - Reversing a Linked List
   - Detecting a Cycle (Floyd's Cycle-Finding Algorithm)
4. **Doubly Linked List**
   - Doubly Linked List Node Definition
   - Insertion at the beginning in Doubly Linked List
   - Traversal (Forward and Backward)
5. **Circular Linked List**
   - Circular Singly Linked List Node Definition
   - Insertion at the end in Circular Linked List
   - Traversal
6. **Linked List Problems**
   - Finding the Nth Node from the End
   - Palindrome Check
7. **Reversing a Sublist of Linked List**

1. Introduction to Linked Lists (Singly Linked List)

Node Definition

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

```
class LinkedList:
    def __init__(self):
        self.head = None
```

2. **Basic Operations on Linked Lists**

Insertion at the beginning

```
def insert_at_beginning(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node

# Example Usage
ll = LinkedList()
ll.insert_at_beginning(10)
ll.insert_at_beginning(20)
```

Insertion at the end

```
def insert_at_end(self, data):
    new_node = Node(data)
    if not self.head:
        self.head = new_node
        return
    last = self.head
    while last.next:
        last = last.next
    last.next = new_node

# Example Usage
ll.insert_at_end(30)
```

Traversal

```
def traverse(self):
    temp = self.head
    while temp:
```

```
        print(temp.data, end=" → ")
        temp = temp.next
    print("None")

ll.traverse()  # Output: 20 → 10 → 30 → None
```

Deletion from the beginning

```
def delete_at_beginning(self):
    if not self.head:
        return
    self.head = self.head.next

# Example Usage
ll.delete_at_beginning()
ll.traverse()  # Output: 10 → 30 → None
```

3. **Advanced Operations**

Reversing a Linked List

```
def reverse(self):
    prev = None
    current = self.head
    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node
    self.head = prev

# Example Usage
ll.reverse()
ll.traverse()  # Output: 30 → 10 → None
```

Detecting a Cycle (Floyd's Cycle-Finding Algorithm)

```
def has_cycle(self):
    slow = self.head
    fast = self.head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False

# Example Usage
ll.has_cycle()  # Output: False
```

4. **Doubly Linked List**

Doubly Linked List Node Definition

```
class DoublyNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
```

Insertion at the beginning in Doubly Linked List

```
def insert_at_beginning(self, data):
    new_node = DoublyNode(data)
    if not self.head:
        self.head = new_node
        return
    new_node.next = self.head
    self.head.prev = new_node
    self.head = new_node
```

```
# Example Usage
dll = DoublyLinkedList()
dll.insert_at_beginning(10)
dll.insert_at_beginning(20)
```

Traversal (Forward and Backward)

```
def traverse_forward(self):
    temp = self.head
    while temp:
        print(temp.data, end=" ↔ ")
        temp = temp.next
    print("None")

def traverse_backward(self):
    temp = self.head
    while temp and temp.next:
        temp = temp.next
    while temp:
        print(temp.data, end=" ↔ ")
        temp = temp.prev
    print("None")

dll.traverse_forward()  # Output: 20 ↔ 10 ↔ None
dll.traverse_backward()  # Output: 10 ↔ 20 ↔ None
```

5. **Circular Linked List**

Circular Singly Linked List Node Definition

```
class CircularNode:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
```

Insertion at the end in Circular Linked List

```
def insert_at_end(self, data):
    new_node = CircularNode(data)
    if not self.head:
        self.head = new_node
        new_node.next = self.head
        return
    last = self.head
    while last.next != self.head:
        last = last.next
    last.next = new_node
    new_node.next = self.head

# Example Usage
cll = CircularLinkedList()
cll.insert_at_end(10)
cll.insert_at_end(20)
```

Traversal

```
def traverse(self):
    if not self.head:
        print("List is empty")
        return
    temp = self.head
    while True:
        print(temp.data, end=" → ")
        temp = temp.next
        if temp == self.head:
            break
    print("Circular")

cll.traverse()  # Output: 10 → 20 → Circular
```

6. **Linked List Problems**

Finding the Nth Node from the End

```
def find_nth_from_end(self, n):
    first = self.head
    second = self.head
    count = 0
    while count < n:
        if not second:
            return None
        second = second.next
        count += 1
    while second:
        first = first.next
        second = second.next
    return first

# Example Usage
ll = LinkedList()
ll.insert_at_end(10)
ll.insert_at_end(20)
ll.insert_at_end(30)
ll.insert_at_end(40)
print(ll.find_nth_from_end(2).data)  # Output: 30
```

Palindrome Check

```
def is_palindrome(self):
    # Convert the linked list to a list to check palindrome
    elements = []
    current = self.head
    while current:
        elements.append(current.data)
        current = current.next
    return elements == elements[::-1]

# Example Usage
ll.is_palindrome()  # Output: False (for a non-palindrome list)
```

7. Reversing a Sublist of Linked List

```
def reverse_sublist(self, m, n):
    if not self.head or m == n:
        return
    dummy = Node(0)
    dummy.next = self.head
    prev = dummy
    for _ in range(m - 1):
        prev = prev.next
    start = prev.next
    then = start.next

    for _ in range(n - m):
        start.next = then.next
        then.next = prev.next
        prev.next = then
        then = start.next

    self.head = dummy.next

# Example Usage
ll.reverse_sublist(2, 4)
ll.traverse()  # Output: 10 → 40 → 30 → 20 → None
```

▼ 7. Stacks and Queues

Stacks

1. **Introduction to Stacks**

A stack is a linear data structure that follows the Last In First Out (LIFO) principle.

- **Stack Operations:**
    - **Push**: Add an element to the stack.
    - **Pop**: Remove the top element.

- **Peek**: View the top element without removing it.
- **IsEmpty**: Check if the stack is empty.
- **Size**: Get the size of the stack.

2. **Stack Implementation (Array-based)**

```
class Stack:
    def __init__(self):
        self.stack = []

    # Push operation
    def push(self, item):
        self.stack.append(item)

    # Pop operation
    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        return "Stack is empty"

    # Peek operation
    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        return "Stack is empty"

    # Check if the stack is empty
    def is_empty(self):
        return len(self.stack) == 0

    # Get the size of the stack
    def size(self):
        return len(self.stack)

# Example Usage
stack = Stack()
stack.push(10)
stack.push(20)
```

```
print(stack.peek())  # Output: 20
print(stack.pop())   # Output: 20
print(stack.size())  # Output: 1
```

3. **Applications of Stacks**

Expression Evaluation (Infix to Postfix)

Converting an infix expression to a postfix expression using a stack:

```
def precedence(op):
    if op == '+' or op == '-':
        return 1
    if op == '*' or op == '/':
        return 2
    return 0

def infix_to_postfix(expression):
    stack = []
    result = []
    for char in expression:
        if char.isalnum():  # Operand
            result.append(char)
        elif char == '(':  # Left parenthesis
            stack.append(char)
        elif char == ')':  # Right parenthesis
            while stack and stack[-1] != '(':
                result.append(stack.pop())
            stack.pop()  # Pop '('
        else:  # Operator
            while stack and precedence(char) <= precedence(stack[-1]):
                result.append(stack.pop())
            stack.append(char)

    # Pop remaining operators
    while stack:
        result.append(stack.pop())

    return ''.join(result)
```

```
# Example Usage
expression = "a+b*(c^d-e)^(f+g*h)-i"
print(infix_to_postfix(expression))  # Output: abcd^e-fgh*+^*+i-
```

Postfix Expression Evaluation

```
def evaluate_postfix(expression):
    stack = []
    for char in expression:
        if char.isdigit():
            stack.append(int(char))
        else:
            b = stack.pop()
            a = stack.pop()
            if char == '+':
                stack.append(a + b)
            elif char == '-':
                stack.append(a - b)
            elif char == '*':
                stack.append(a * b)
            elif char == '/':
                stack.append(a / b)
    return stack[0]

# Example Usage
postfix = "23*54*+9-"  # (2 * 3) + (5 * 4) - 9
print(evaluate_postfix(postfix))  # Output: 17
```

4. **Balanced Parentheses Problem**

Check if parentheses are balanced using a stack:

```
def is_balanced(expression):
    stack = []
    for char in expression:
        if char in "({[":
            stack.append(char)
        elif char in ")}]":
            if not stack:
```

```
                return False
            top = stack.pop()
            if char == ")" and top != "(":
                return False
            elif char == "}" and top != "{":
                return False
            elif char == "]" and top != "[":
                return False
    return not stack

# Example Usage
expression = "({[a+b]*[c/d]})"
print(is_balanced(expression))  # Output: True
```

5. **Stock Span Problem**

Given an array of stock prices, find the span of stock's price for all days.

```
def stock_span(prices):
    stack = []
    result = []
    for i, price in enumerate(prices):
        while stack and prices[stack[-1]] <= price:
            stack.pop()
        span = i + 1 if not stack else i - stack[-1]
        result.append(span)
        stack.append(i)
    return result

# Example Usage
prices = [100, 80, 60, 70, 60, 75, 85]
print(stock_span(prices))  # Output: [1, 1, 1, 2, 1, 4, 6]
```

6. **Time Complexity Analysis**

- **Push** and **Pop** operations: O(1)
- **Peek** operation: O(1)
- **Infix to Postfix conversion**: O(n), where n is the length of the expression

- **Postfix expression evaluation**: O(n), where n is the number of elements in the expression
- **Balanced parentheses**: O(n), where n is the length of the expression
- **Stock span**: O(n), where n is the number of days/prices

Queues

1. **Introduction to Queues**

A queue is a linear data structure that follows the First In First Out (FIFO) principle. The first element added to the queue will be the first one to be removed.

- **Queue Operations**:
  - **Enqueue**: Add an element to the queue.
  - **Dequeue**: Remove an element from the queue.
  - **Front**: View the front element of the queue without removing it.
  - **IsEmpty**: Check if the queue is empty.
  - **Size**: Get the size of the queue.

2. **Queue Implementation (Array-based)**

```
class Queue:
    def __init__(self):
        self.queue = []

    # Enqueue operation
    def enqueue(self, item):
        self.queue.append(item)

    # Dequeue operation
    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        return "Queue is empty"

    # Front operation
    def front(self):
```

```
        if not self.is_empty():
            return self.queue[0]
        return "Queue is empty"

    # Check if the queue is empty
    def is_empty(self):
        return len(self.queue) == 0

    # Get the size of the queue
    def size(self):
        return len(self.queue)

# Example Usage
queue = Queue()
queue.enqueue(10)
queue.enqueue(20)
print(queue.front())  # Output: 10
print(queue.dequeue())  # Output: 10
print(queue.size())  # Output: 1
```

3. **Circular Queue**

A circular queue is a queue in which the last element points to the first element, making it circular.

```
class CircularQueue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = self.rear = -1

    # Enqueue operation
    def enqueue(self, item):
        if (self.rear + 1) % self.capacity == self.front:
            return "Queue is full"
        elif self.front == -1:
            self.front = self.rear = 0
        else:
            self.rear = (self.rear + 1) % self.capacity
```

```
        self.queue[self.rear] = item

    # Dequeue operation
    def dequeue(self):
        if self.front == -1:
            return "Queue is empty"
        data = self.queue[self.front]
        if self.front == self.rear:
            self.front = self.rear = -1
        else:
            self.front = (self.front + 1) % self.capacity
        return data

    # Front operation
    def front(self):
        if self.front == -1:
            return "Queue is empty"
        return self.queue[self.front]

    # Check if the queue is empty
    def is_empty(self):
        return self.front == -1

    # Check if the queue is full
    def is_full(self):
        return (self.rear + 1) % self.capacity == self.front

    # Size of the queue
    def size(self):
        if self.front == -1:
            return 0
        return (self.rear - self.front + 1) % self.capacity

# Example Usage
cq = CircularQueue(3)
cq.enqueue(10)
cq.enqueue(20)
cq.enqueue(30)
```

```
print(cq.dequeue())  # Output: 10
cq.enqueue(40)
print(cq.front())  # Output: 20
```

4. **Queue Using Two Stacks**

You can implement a queue using two stacks to simulate enqueue and dequeue operations.

```
class QueueUsingStacks:
    def __init__(self):
        self.stack1 = []
        self.stack2 = []

    # Enqueue operation
    def enqueue(self, item):
        self.stack1.append(item)

    # Dequeue operation
    def dequeue(self):
        if not self.stack2:
            if not self.stack1:
                return "Queue is empty"
            while self.stack1:
                self.stack2.append(self.stack1.pop())
        return self.stack2.pop()

# Example Usage
queue = QueueUsingStacks()
queue.enqueue(10)
queue.enqueue(20)
print(queue.dequeue())  # Output: 10
print(queue.dequeue())  # Output: 20
```

5. **Priority Queue**

A priority queue is a type of queue where each element is assigned a priority. The element with the highest priority is dequeued first.

In Python, you can use `heapq` to implement a priority queue, where the element with the smallest value has the highest priority.

```
import heapq

class PriorityQueue:
    def __init__(self):
        self.pq = []
        self.counter = 0

    # Enqueue operation
    def enqueue(self, item, priority):
        heapq.heappush(self.pq, (priority, self.counter, item))
        self.counter += 1

    # Dequeue operation
    def dequeue(self):
        if not self.pq:
            return "Queue is empty"
        return heapq.heappop(self.pq)[-1]

    # Front operation
    def front(self):
        if not self.pq:
            return "Queue is empty"
        return self.pq[0][-1]

    # Check if the queue is empty
    def is_empty(self):
        return len(self.pq) == 0

    # Size of the queue
    def size(self):
        return len(self.pq)

# Example Usage
pq = PriorityQueue()
pq.enqueue("task1", 3)
```

```
pq.enqueue("task2", 1)
pq.enqueue("task3", 2)
print(pq.dequeue())  # Output: task2 (highest priority)
```

6. **Double-Ended Queue (Deque)**

A deque is a linear data structure that allows elements to be added or removed from both ends (front and rear).

```
from collections import deque

# Create a deque
d = deque()

# Enqueue at the front
d.appendleft(10)

# Enqueue at the rear
d.append(20)

# Dequeue from the front
print(d.popleft())  # Output: 10

# Dequeue from the rear
print(d.pop())  # Output: 20
```

7. **Queue Problems**

Reverse First K Elements of Queue

```
from collections import deque

def reverse_k_elements(queue, k):
    stack = []
    # Dequeue first k elements and push them onto a stack
    for _ in range(k):
        stack.append(queue.popleft())

    # Enqueue elements from the stack back to the queue
```

```
    while stack:
        queue.append(stack.pop())

    # Append the remaining elements from the front of the queue
    for _ in range(len(queue) - k):
        queue.append(queue.popleft())

    return queue

# Example Usage
q = deque([1, 2, 3, 4, 5])
result = reverse_k_elements(q, 3)
print(result)  # Output: [3, 2, 1, 4, 5]
```

8. **Time Complexity Analysis**

- **Enqueue** operation: O(1)
- **Dequeue** operation: O(1) for normal queues; O(n) for queue using two stacks
- **Front** operation: O(1)
- **Circular Queue**: O(1) for all operations
- **Priority Queue**: O(log n) for enqueue and dequeue operations (using heapq)

▼ 8. Hashing
▼ 9. **Heaps**
▼ 10. Trees
▼ 12. Graphs
▼ 13. Greedy Algorithms
▼ 14. Dynamic Programming
▼ 15. Backtracking
▼ 16.
▼ 17.
▼ 18.
▼ 19.

▼ 20.