Assignment 1
DATA STRUCTURES & ALGORITHMS

Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

Identify the Data Structures

1. Array
Definition: An array is a collection of elements of the same type, stored consecutively in memory. Each element can be accessed through an index.
Characteristics:
Fixed size (in many programming languages).
Access elements by index in O(1) time.
2. Stack
Definition: Stack is a data structure that follows the LIFO (Last In, First Out) principle, meaning the last element added will be taken out first.
Characteristics:
Operate only on the top element of the stack.
Push (add) and Pop (take out) operations take place in O(1) time.

3. Tree
Definition: A tree is a hierarchical data structure, each node has a value and can have many children. A Binary Tree is a common type of tree, in which each node has a maximum of two children.
Characteristics:
There are many ways to traverse a tree: Preorder, Inorder, Postorder.
The insertion and search operations have a complexity ranging from O(log n) to O(n) depending on whether the tree is balanced or not.
4. Hash Table
Definition: A hash table uses a hash function to map a key to a specific location in the table. This allows for fast searches, insertions, and deletions.
Characteristics:
The average search and insertion time is O(1) but can be as bad as O(n) in the case of key conflicts.

Define the Operations

2.1 Stack Operations

Push: Add an element to the top of the stack.

Pop: Remove and return the top element from the stack.

Peek/Top: Return the top element without removing it.

isEmpty: Check if the stack is empty.

size: Return the number of elements in the stack.

2.2 Array Operations

Access: Retrieve an element at a specified index.

Insert: Add an element at a specified index, shifting subsequent elements.

Delete: Remove an element at a specified index, shifting subsequent elements left.

Search: Find the index of a specified element.

Update: Change the value of an element at a specified index.

2.3 Tree Operations

Insert: Add a new node to the tree based on its value.

Delete: Remove a node from the tree.

Search: Find a node with a specified value in the tree.

Traversal: Visit all nodes in a specific order (pre-order, in-order, post-order).

Update: Change the value of a specified node.

2.4 Hash Table Operations

Insert: Add a key-value pair to the hash table.

Delete: Remove a key-value pair based on the key.

Search: Retrieve the value associated with a specified key.

Update: Change the value of an existing key.

isEmpty: Check if the hash table is empty.

Define Pre- and Post-conditions

2.1 Stack Pre- and Post-conditions

Push(value)

- Pre-condition: The stack is not full (for bounded stacks).
- Post-condition: The stack size increases by 1, and the top element becomes the pushed value.

Pop()

- Pre-condition: The stack is not empty.
- Post-condition: The stack size decreases by 1, and the returned value is the previous top element.

Peek()

- Pre-condition: The stack is not empty.
- Post-condition: The top element remains the same, and the returned value is the top element.

isEmpty()

- Pre-condition: None.
- Post-condition: Returns true if the stack size is 0, otherwise false.

size()

- Pre-condition: None.
- Post-condition: Returns the current number of elements in the stack.

2.2 Array Pre- and Post-conditions

Access(index)

- Pre-condition: The index is within bounds ($0 \le index < array\ length$).
- Post-condition: Returns the element at the specified index.

Insert(index, value)

- Pre-condition: The index is within bounds for insertion ($0 \le index \le array\ length$).
- Post-condition: The value is added at the specified index, and subsequent elements are shifted to the right.

Delete(index)

- Pre-condition: The index is within bounds ($0 \le index < array\ length$).
- Post-condition: The element at the specified index is removed, and subsequent elements are shifted to the left.

Search(value)

- Pre-condition: None.
- Post-condition: Returns the index of the value if found; otherwise, returns -1.

Update(index, value)

- Pre-condition: The index is within bounds ($0 \le index < array\ length$).
- Post-condition: The value at the specified index is updated to the new value.

2.3 Tree Pre- and Post-conditions

Insert(value)

- Pre-condition: None.
- Post-condition: A new node with the specified value is added to the tree.

Delete(value)

- Pre-condition: The value exists in the tree.
- Post-condition: The node with the specified value is removed from the tree.

Search(value)

- Pre-condition: None.
- Post-condition: Returns the node with the specified value if found; otherwise, returns null.

Traversal(order)

- Pre-condition: None.
- Post-condition: All nodes are visited in the specified order.

Update(oldValue, newValue)

- Pre-condition: The oldValue exists in the tree.
- Post-condition: The value of the specified node is updated to newValue.

2.4 Hash Table Pre- and Post-conditions

Insert(key, value)

- Pre-condition: The key does not already exist in the hash table (for unique keys).
- Post-condition: A new key-value pair is added to the hash table.

Delete(key)

- Pre-condition: The key exists in the hash table.
- Post-condition: The key-value pair associated with the specified key is removed.

Search(key)

- Pre-condition: None.
- Post-condition: Returns the value associated with the specified key if found; otherwise, returns null.

Update(key, value)

- Pre-condition: The key exists in the hash table.
- Post-condition: The value associated with the specified key is updated to the new value.

isEmpty()

- Pre-condition: None.
- Post-condition: Returns true if the hash table contains no key-value pairs; otherwise, returns false.

Discuss Time and Space Complexity

2.1 Stack Complexity

- **Push(value)**
  - **Time Complexity**: O(1) - Adding an element to the top is done in constant time.
  - **Space Complexity**: O(n) - The space complexity is dependent on the number of elements stored in the stack.
- **Pop()**
  - **Time Complexity**: O(1) - Removing the top element is done in constant time.
  - **Space Complexity**: O(n) - The space complexity remains dependent on the number of elements.
- **Peek()**
  - **Time Complexity**: O(1) - Accessing the top element is done in constant time.
  - **Space Complexity**: O(1) - No additional space is used.
- **isEmpty()**
  - **Time Complexity**: O(1) - Checking if the stack is empty is done in constant time.
  - **Space Complexity**: O(1) - No additional space is used.
- **size()**
  - **Time Complexity**: O(1) - Returning the size is done in constant time.
  - **Space Complexity**: O(1) - No additional space is used.

2.2 Array Complexity

- **Access(index)**
  - **Time Complexity**: O(1) - Accessing an element by index is done in constant time.
  - **Space Complexity**: O(1) - No additional space is used.
- **Insert(index, value)**
  - **Time Complexity**: O(n) - Inserting requires shifting elements, leading to linear time complexity.
  - **Space Complexity**: O(n) - The space complexity is dependent on the number of elements in the array.
- **Delete(index)**
  - **Time Complexity**: O(n) - Deleting requires shifting elements, leading to linear time complexity.
  - **Space Complexity**: O(n) - The space complexity is dependent on the number of elements in the array.
- **Search(value)**
  - **Time Complexity**: O(n) - In the worst case, all elements need to be checked.
  - **Space Complexity**: O(1) - No additional space is used.
- **Update(index, value)**
  - **Time Complexity**: O(1) - Updating an element by index is done in constant time.
  - **Space Complexity**: O(1) - No additional space is used.

2.3 Tree Complexity

- **Insert(value)**
  - **Time Complexity**: O(h) - Where h is the height of the tree; in a balanced tree, this is O(log n), while in an unbalanced tree, it can be O(n).
  - **Space Complexity**: O(n) - The space complexity depends on the number of nodes.
- **Delete(value)**
  - **Time Complexity**: O(h) - Similar to insert; it depends on the height of the tree.
  - **Space Complexity**: O(n) - The space complexity is dependent on the number of nodes.
- **Search(value)**
  - **Time Complexity**: O(h) - Where h is the height of the tree.
  - **Space Complexity**: O(1) - No additional space is used.
- **Traversal(order)**
  - **Time Complexity**: O(n) - All nodes are visited.
  - **Space Complexity**: O(h) - Space used in recursion stack (h is the height).
- **Update(oldValue, newValue)**
  - **Time Complexity**: O(h) - Depends on the height of the tree.
  - **Space Complexity**: O(1) - No additional space is used.

2.4 Hash Table Complexity

- **Insert(key, value)**
  - **Time Complexity**: O(1) on average (amortized), O(n) in the worst case (due to collisions).
  - **Space Complexity**: O(n) - The space complexity is dependent on the number of key-value pairs.
- **Delete(key)**
  - **Time Complexity**: O(1) on average, O(n) in the worst case.
  - **Space Complexity**: O(n) - The space complexity is dependent on the number of key-value pairs.
- **Search(key)**
  - **Time Complexity**: O(1) on average, O(n) in the worst case.
  - **Space Complexity**: O(1) - No additional space is used.
- **Update(key, value)**
  - **Time Complexity**: O(1) on average, O(n) in the worst case.
  - **Space Complexity**: O(1) - No additional space is used.
- **isEmpty()**
  - **Time Complexity**: O(1) - Checking if the hash table is empty is done in constant time.
  - **Space Complexity**: O(1) - No additional space is used.