**TAD’S of the used data structures.**

|  |
| --- |
| **Type Abstract Data (TAD): Hashtable<T>** |
| **Description:**  Hashtable is a synchronized data structure in Java that stores key-value pairs and provides efficient storage and retrieval based on their associated keys. In this case is generic. |
| **Invariants:**  1. Each key in the Hashtable is associated with a unique value.  2. Keys and values must be of non-null reference types.  3. The Hashtable maintains a balanced load factor for efficient performance. |
| **Primitive Operations:**  1. Insertion (put):  - Description: Associates a key with a value in the Hashtable.  - Inputs: T and a key.  - Precondition: Key and value must be different from null.  - Outputs: void.  - Postcondition: The key-value pair is added to the Hashtable, and the previous value associated with the key (if any) is returned. If the key already exists, its associated value is updated.  2. Retrieval (get):  - Description: Retrieves the value associated with a given key from the Hashtable.  - Inputs: key.  - Precondition: The key must be different from null.  - Outputs: T or null.  - Postcondition: Returns the value associated with the key if it exists in the Hashtable; otherwise, returns “null”.  3. Deletion (remove):  - Description: Removes the key-value pair associated with a given key from the Hashtable.  - Inputs: key.  - Precondition: The key must be different from null.  - Outputs: void or null.  - Postcondition: Removes the key-value pair from the Hashtable and returns the previously associated value (if the key existed). If the key is not found, returns “null”.  4. Size (size):  - Description: Returns the number of key-value pairs in the Hashtable.  - Inputs: None.  - Precondition: None.  - Outputs: int.  - Postcondition: Returns the number of key-value pairs currently stored in the Hashtable.  5. Containment (containsKey):  - Description: Checks if the Hashtable contains a specific key.  - Inputs: Key.  - Precondition: The key must be different from null.  - Outputs: Boolean.  - Postcondition: Returns “true” if the key exists in the Hashtable; otherwise, returns “false”. |

|  |
| --- |
| **Type Abstract Data (TAD): Queue<T>** |
| **Description:**  A Queue is an abstract data structure that represents a linear collection of elements where elements are added from the end (tail or rear) and removed from the front (head). It follows the First-In-First-Out (FIFO) principle, meaning that the first element added is the one that is removed first. In this case is generic. |
| **Invariants:**  1. The Queue is a linear data structure.  2. Elements are added at the rear (end or tail) of the Queue.  3. Elements are removed from the front (head) of the Queue.  4. The Queue may have a maximum capacity (bounded Queue) or be unbounded (no fixed capacity).  5. The size of the Queue is the number of elements it contains. |
| **Primitive Operations:**  1. Enqueue (add):  - Description: Adds an element to the rear of the Queue.  - Inputs: T.  - Precondition: The element to be added must not be null.  - Outputs: None.  - Postcondition: The element is added to the rear of the Queue.  2. Dequeue (remove):  - Description: Removes and returns the element from the front of the Queue.  - Inputs: None.  - Precondition: The Queue must not be empty.  - Outputs: T.  - Postcondition: The element is removed from the front of the Queue and returned.  3. Peek (front):  - Description: Returns the element at the front of the Queue without removing it.  - Inputs: None.  - Precondition: The Queue must not be empty.  - Outputs: T.  - Postcondition: The Queue remains unchanged, and the element at the front is returned.  4. IsEmpty (isEmpty):  - Description: Checks if the Queue is empty.  - Inputs: None.  - Precondition: None.  - Outputs: Boolean.  - Postcondition: Returns true if the Queue contains no elements; otherwise, returns false.  5. Size (size):  - Description: Returns the number of elements in the Queue.  - Inputs: None.  - Precondition: None.  - Outputs: int.  - Postcondition: Returns the number of elements in the Queue.  6. Clear (clear):  - Description: Removes all elements from the Queue.  - Inputs: None.  - Precondition: None.  - Outputs: None.  - Postcondition: The Queue becomes empty, with a size of 0. |

|  |
| --- |
| **Type Abstract Data (TAD): Stack<T>** |
| **Description:**  A Stack is an abstract data structure that represents a collection of elements with two primary operations: push and pop. It follows the Last-In-First-Out (LIFO) principle, meaning that the last added element is the first one to be removed. In this case is generic. |
| **Invariants:**  1. The Stack is a linear data structure.  2. Elements are added and removed from one end, traditionally known as the "top" of the Stack.  3. The Stack may be empty, containing no elements.  4. The size of the Stack is the number of elements it contains. |
| **Primitive Operations:**  1. Push (push):  - Description: Adds an element to the top of the Stack.  - Inputs: T.  - Precondition: The element to be added must not be null.  -Outputs: None.  - Postcondition: The element is added to the top of the Stack.  2. Pop (pop):  - Description: Removes and returns the element from the top of the Stack.  - Inputs: None.  - Precondition: The Stack must not be empty.  - Outputs: T.  - Postcondition: The element is removed from the top of the Stack and returned.  3. Peek (top):  - Description: Returns the element at the top of the Stack without removing it.  - Inputs: None.  - Precondition: The Stack must not be empty.  - Outputs: T.  - Postcondition: The Stack remains unchanged, and the element at the top is returned.  4. IsEmpty (isEmpty):  - Description: Checks if the Stack is empty.  - Inputs: None.  - Precondition: None.  - Outputs: Boolean.  - Postcondition: Returns true if the Stack contains no elements; otherwise, returns false.  5. Size (size):  - Description: Returns the number of elements in the Stack.  - Inputs: None.  - Precondition: None.  - Outputs: Int.  - Postcondition: Returns the count of elements currently in the Stack. |

|  |
| --- |
| **Type Abstract Data (TAD): Double Linked List<T>** |
| **Description:**  A Double Linked List is a linear data structure consisting of a sequence of elements called nodes, where each node contains a data element and references to the next and previous nodes in the list. In this case is generic. |
| **Invariants:**  1. The Double Linked List is composed of nodes.  2. Each node contains a data element.  3. Each node has a reference to the next node and a reference to the previous node (except for the first and last nodes, they only have one reference).  4. The list has a reference to the first (head) and last (tail) nodes. |
| **Primitive Operations:**  1. Insertion at the Beginning (prepend):  - Description: Adds an element to the beginning of the Doubly Linked List.  - Inputs: T.  - Precondition: The element to be added must not be null.  - Outputs: None.  - Postcondition: A new node containing the element is inserted at the front of the list, and the previous head becomes the second node.  2. Insertion at the End (append):  - Description: Adds an element to the end of the Doubly Linked List.  - Inputs: T.  - Precondition: The element to be added must not be null.  - Outputs: None.  - Postcondition: A new node containing the element is inserted at the end of the list, and the previous tail becomes the second-to-last node.  3. Insertion After a Node (insertAfter):  - Description: Adds an element after a specified node in the Doubly Linked List.  - Inputs: T, T.  - Precondition: The element to be added must not be null, and the specified node must be in the list.  - Outputs: None.  - Postcondition: A new node containing the element is inserted after the specified node, and the references are updated accordingly.  4. Deletion (delete):  - Description: Removes a specified node from the Doubly Linked List.  - Inputs: T.  - Precondition: The node to be deleted must be in the list.  - Outputs: None.  - Postcondition: The specified node is removed from the list, and the references of adjacent nodes are updated to maintain the list's integrity.  5. Search (search):  - Description: Searches for a specified element in the Doubly Linked List.  - Inputs: T.  - Precondition: None.  - Outputs: T.  - Postcondition: Returns the node containing the element if found; otherwise, returns null.  6. IsEmpty (isEmpty):  - Description: Checks if the Doubly Linked List is empty.  - Inputs: None.  - Precondition: None.  - Outputs: Boolean.  - Postcondition: Returns true if the list contains no nodes; otherwise, returns false.  7. Size (size):  - Description: Returns the number of nodes in the Doubly Linked List.  - Inputs: None.  - Precondition: None.  - Outputs: Int.  - Postcondition: Returns the count of nodes currently in the list.  8. Traversal:  - Description: Iterates through the Doubly Linked List, visiting each node.  - Inputs: None.  - Precondition: None.  - Outputs: None.  - Postcondition: The entire list is traversed in either forward or backward direction, depending on the traversal direction.  Note: This primitive operation could appear to be useless, but depending on the implementation, it may have different uses. In the previous description it just passes through all the list, but this could be used to print all the list, debug and etc. |