**iTest Bangladesh**



**Topic: Linked List DUT Test Plan**

**By- A.S.M. Sarwar Jahan**

Contents

[1. Introduction 2](#_Toc183911633)

[2. DUT Overview 2](#_Toc183911634)

[3. Test Objectives 2](#_Toc183911635)

[4. Scope of Testing 3](#_Toc183911636)

[5. Methodology 3](#_Toc183911637)

[6. Test Cases: 4](#_Toc183911638)

[1. Test Case: NEW Opcode (Creating a New Linked List) 4](#_Toc183911639)

[2. Test Case: TRAVERSE Opcode (Traversing the Linked List) 4](#_Toc183911640)

[3. Test Case: FIND Opcode (Finding Elements) 5](#_Toc183911641)

[4. Test Case: INSERT Opcode (Inserting After an Element) 5](#_Toc183911642)

[5. Test Case: DELETE Opcode (Deleting an Element) 6](#_Toc183911643)

[6. Test Case: LEN Opcode (Length of the List) 8](#_Toc183911644)

[7. Test Case: REVERSE Opcode (Reversing the List) 8](#_Toc183911645)

[8. Test Case: Handling NULL or Undefined Data 9](#_Toc183911646)

[9. Test Case: Constrained Random Test 9](#_Toc183911647)

[Test Random Data Insertions and Deletions: 9](#_Toc183911648)

[Test Random List Length Calculation: 9](#_Toc183911649)

[Test Random Reversals: 10](#_Toc183911650)

# 1. Introduction

This test plan outlines the verification strategy for the Linked List Design Under Test (DUT). The primary objective of this test plan is to ensure the correct functionality and behavior of the Linked List operations implemented in the DUT. This plan will cover a variety of test scenarios, including both direct and constrained random testing, to validate the DUT against its specification.

# 2. DUT Overview

The Linked List DUT is a digital system that provides functionality for managing a singly linked list. The DUT has a single method, "run", which takes two inputs:

* opcode (a 3-bit enum) to specify the operation to be performed.
* data (a queue of bits, 8-bits wide) to provide input values for the operation.

The supported opcode operations are:

* NEW: Creates a new linked list using the given data queue.
* TRAVERSE: Traverses through the linked list and returns the data in a queue format.
* FIND: Searches for all occurrences of a specified value and returns the indexes where it is found.
* INSERT: Inserts a new value after a specified value in the list.
* DELETE: Deletes a specified value from the list.
* LEN: Returns the total number of elements in the list.
* REVERSE: Reverses the linked list and returns the updated list.

Each of these operations must be verified to ensure that the linked list behaves as expected, including edge cases and corner conditions.

# 3. Test Objectives

The goal of this test plan is to:

* Verify correctness: Ensure that each of the listed operations (NEW, TRAVERSE, FIND, INSERT, DELETE, LEN, REVERSE) behaves as specified.
* Ensure robustness: Validate the linked list against invalid inputs, boundary conditions, and edge cases (e.g., inserting into an empty list, finding non-existing elements, etc.).
* Check consistency: Validate that multiple operations can be performed consecutively without causing unintended side effects or failures.
* Measure performance: While not the primary focus of this test plan, ensure that operations like traversal, insertion, and deletion operate within reasonable time limits.

# 4. Scope of Testing

The scope of testing will include:

* Functional Tests: Testing all valid operations (NEW, TRAVERSE, FIND, INSERT, DELETE, LEN, REVERSE) to ensure they perform correctly.
* Boundary and Edge Cases: Testing the limits of the linked list (e.g., empty list, single-element list, large list).
* Error Handling: Verifying that the system handles invalid operations gracefully, such as attempting to find or delete non-existent values.
* Data Integrity: Ensuring that the list maintains data integrity after each operation (e.g., correct insertion and deletion behavior).

# 5. Methodology

The test will consist of:

* Direct Tests: These tests will involve providing specific inputs for each operation and verifying the expected output. This will include testing edge cases and common use cases.
* Constrained Random Tests: These tests will randomly generate a series of operations on the linked list, ensuring that the DUT can handle a variety of conditions and sequences of operations.

In addition, the test bench will include monitors, stimulus generators, and verification modules to manage the inputs and outputs, ensuring that the DUT is thoroughly tested in various conditions.

# 6. Test Cases:

## 1. Test Case: NEW Opcode (Creating a New Linked List)

**Goal**: Test the creation of a new linked list from a data queue.

**Edge Cases**:

* **Empty Data Queue**: The NEW opcode with an empty queue should initialize an empty list.
  + **Test Input**: {NEW, {}}
  + **Expected Output**: [] (Empty linked list).
* **Single Element Queue**: Ensure that the list is correctly initialized with only one element.
  + **Test Input**: {NEW, {5}}
  + **Expected Output**: [5] (List with one element).
* **Multiple Elements Queue**: Test creation with multiple elements.
  + **Test Input**: {NEW, {1, 2, 3, 4}}
  + **Expected Output**: [1 -> 2 -> 3 -> 4]

**Boundary Cases for Data Type**

* **Minimum Data Value**: Ensure that the linked list can handle the smallest possible value for data (which is 0 in this case, since it’s a queue of bit[0:7]).
  + **Test Input**: {NEW, {0}}
  + **Expected Output**: [0]
* **Maximum Data Value**: Ensure that the linked list can handle the largest possible value for data (which is 255 for bit[0:7]).
  + **Test Input**: {NEW, {255}}
  + **Expected Output**: [255]

## 2. Test Case: TRAVERSE Opcode (Traversing the Linked List)

**Goal**: Validate the traversal operation.

**Edge Cases**:

* **Empty List**: Traversing an empty list should return an empty queue.
  + **Test Input**: {TRAVERSE, {}} (after NEW with empty list).
  + **Expected Output**: [] (Empty list).
* **Non-empty List**: Traverse a list with several elements and check if it matches the order.
  + **Test Input**: {TRAVERSE, {}} (after NEW with {1, 2, 3}).
  + **Expected Output**: [1 -> 2 -> 3]
* **List with Duplicates**: Ensure traversal works correctly with duplicate elements.
  + **Test Input**: {TRAVERSE, {}} (after NEW with {1, 2, 2, 3}).
  + **Expected Output**: [1 -> 2 -> 2 -> 3]

## 3. Test Case: FIND Opcode (Finding Elements)

**Goal**: Validate the ability to find indices of elements in the linked list.

**Edge Cases**:

* **Empty List**: Finding any element in an empty list should return {-1}.
  + **Test Input**: {FIND, {5}} (after NEW with empty list).
  + **Expected Output**: {-1}
* **Element Found Once**: Ensure correct index is returned for elements appearing once.
  + **Test Input**: {FIND, {2}} (after NEW with {1, 2, 3}).
  + **Expected Output**: {1} (Element 2 found at index 1).
* **Element Found Multiple Times**: Verify correct handling of duplicates.
  + **Test Input**: {FIND, {2}} (after NEW with {1, 2, 2, 3}).
  + **Expected Output**: {1, 2} (Element 2 found at indices 1 and 2).
* **Element Not Found**: Ensure the method correctly returns {-1} if the element is not in the list.
  + **Test Input**: {FIND, {5}} (after NEW with {1, 2, 3}).
  + **Expected Output**: {-1}

## 4. Test Case: INSERT Opcode (Inserting After an Element)

**Goal**: Test inserting an element after a specified element.

**Edge Cases**:

* **Inserting in an Empty List**: Inserting into an empty list should result in the new element being the head of the list.
  + **Test Input**: {INSERT, {5, 10}} (after NEW with {}).
  + **Expected Output**: [10]
* **Element to Insert After Does Not Exist**: Ensure the insertion does not happen if the target element is not in the list.
  + **Test Input**: {INSERT, {5, 10}} (after NEW with {1, 2, 3}).
  + **Expected Output**: [1 -> 2 -> 3] (No change).
* **Inserting After the Last Element**: Inserting after the last element should place the new element at the end.
  + **Test Input**: {INSERT, {3, 10}} (after NEW with {1, 2, 3}).
  + **Expected Output**: [1 -> 2 -> 3 -> 10]
* **Inserting in a List with Multiple Occurrences**: Ensure the insert occurs after the first occurrence of the target.
  + **Test Input**: {INSERT, {2, 4}} (after NEW with {1, 2, 2, 3}).
  + **Expected Output**: [1 -> 2 -> 4 -> 2 -> 3] (Insert after the first 2).

**Multiple Inserts at the Same Index**

* **Inserting Multiple Times at the Same Position**: Test repeated insertions at the same index, ensuring the list correctly maintains the structure.
  + **Test Input**:
    1. {NEW, {1, 2, 3}}
    2. {INSERT, {2, 4}}
    3. {INSERT, {2, 5}}
  + **Expected Output**: [1 -> 2 -> 4 -> 5 -> 3]

## 5. Test Case: DELETE Opcode (Deleting an Element)

**Goal**: Verify that the delete operation works correctly.

**Edge Cases**:

* **Deleting from an Empty List**: Ensure that deleting from an empty list results in no changes.
  + **Test Input**: {DELETE, {5}} (after NEW with {}).
  + **Expected Output**: [] (No change).
* **Element Not Found**: Deleting an element that is not in the list should not alter the list.
  + **Test Input**: {DELETE, {5}} (after NEW with {1, 2, 3}).
  + **Expected Output**: [1 -> 2 -> 3]
* **Deleting the Head**: Ensure the head of the list is properly deleted.
  + **Test Input**: {DELETE, {1}} (after NEW with {1, 2, 3}).
  + **Expected Output**: [2 -> 3]
* **Deleting a Tail Element**: Deleting the last element should result in the list shrinking properly.
  + **Test Input**: {DELETE, {3}} (after NEW with {1, 2, 3}).
  + **Expected Output**: [1 -> 2]
* **Deleting Duplicates**: Deleting a duplicated element should only remove one occurrence.
  + **Test Input**: {DELETE, {2}} (after NEW with {1, 2, 2, 3}).
  + **Expected Output**: [1 -> 2 -> 3]
* **Insert and Delete in Quick Succession**: Insert and then immediately delete the same element to ensure the list handles operations one after another.
  + **Test Input**: {NEW, {1, 2, 3}} followed by {INSERT, {2, 5}} and {DELETE, {5}}
  + **Expected Output**: [1 -> 2 -> 3] (After insert and delete).
* **Delete the Same Element Multiple Times**: Check if multiple deletion calls for the same element are handled gracefully without crashing or creating unexpected behavior.
  + **Test Input**: {NEW, {1, 2, 3, 2, 4}}, followed by:
    1. {DELETE, {2}}
    2. {DELETE, {2}}
  + **Expected Output**:
    1. After first delete: [1 -> 3 -> 2 -> 4]
    2. After second delete: [1 -> 3 -> 4]

## 6. Test Case: LEN Opcode (Length of the List)

**Goal**: Ensure the length of the list is returned correctly.

**Edge Cases**:

* **Empty List**: The length of an empty list should be 0.
  + **Test Input**: {LEN, {}} (after NEW with empty list).
  + **Expected Output**: [0]
* **List with One Element**: Verify that the length of a single-element list is 1.
  + **Test Input**: {LEN, {}} (after NEW with {5}).
  + **Expected Output**: [1]
* **List with Multiple Elements**: Ensure correct length for multi-element lists.
  + **Test Input**: {LEN, {}} (after NEW with {1, 2, 3}).
  + **Expected Output**: [3]
* **After Deletion**: Test the length after a deletion.
  + **Test Input**: {LEN, {}} (after NEW with {1, 2, 3} and DELETE with {2}).
  + **Expected Output**: [2]

## 7. Test Case: REVERSE Opcode (Reversing the List)

**Goal**: Verify that the reverse operation works correctly.

**Edge Cases**:

* **Empty List**: Reversing an empty list should still result in an empty list.
  + **Test Input**: {REVERSE, {}} (after NEW with empty list).
  + **Expected Output**: []
* **Single Element List**: Reversing a list with only one element should return the same list.
  + **Test Input**: {REVERSE, {}} (after NEW with {5}).
  + **Expected Output**: [5]
* **Multiple Element List**: Ensure the list is properly reversed.
  + **Test Input**: {REVERSE, {}} (after NEW with {1, 2, 3}).
  + **Expected Output**: [3 -> 2 -> 1]
* **Reversing an Already Reversed List**: Ensure that reversing a list twice restores the original order.
  + **Test Input**: {REVERSE, {}} (after NEW with {1, 2, 3} and REVERSE).
  + **Expected Output**: [1 -> 2 -> 3] (Reverse twice).

## 8. Test Case: Handling NULL or Undefined Data

**Goal**: Validate the DUT's ability to handle unexpected or undefined inputs gracefully.

**Edge Cases**:

* **NULL Data Input**: Check if the code handles NULL or undefined inputs.
  + **Test Input**: {NEW, NULL}
  + **Expected Output**: Error or graceful handling (e.g., empty list or invalid operation message).
* **Invalid Opcode**: Test how the DUT handles unsupported or unknown opcodes.
  + **Test Input**: {INVALID\_OPCODE, {}}

**Expected Output**: Error or invalid operation message

## 9. Test Case: Constrained Random Test

### Test Random Data Insertions and Deletions:

* + **Test Case: Random insertion and deletion operations in a linked list.**
  + **Input:**
    - Randomly generate a sequence of INSERT and DELETE operations.
    - Each INSERT operation should randomly pick a value to insert after a randomly selected node.
    - Each DELETE operation should randomly pick a value to delete.
  + **Constraints:**
    - Ensure that each deletion only occurs if the value is present in the list.
    - Ensure that insertions respect the order of the list and do not break the structure.
  + **Expected Output:** The list should always maintain correct links between nodes.

### Test Random List Length Calculation:

* + **Test Case: Perform random insertions and deletions, and then check the length using LEN.**
  + **Input:**
    - Randomly insert and delete values in the list.
    - After a sequence of operations, call LEN to check the current list length.
  + **Constraints:**
    - Ensure that each operation respects the list's integrity (i.e., no breaking links).
  + **Expected Output:** The length returned by LEN should match the actual number of elements in the list.

### Test Random Reversals:

* + **Test Case: Randomly reverse the list after performing a sequence of operations.**
  + **Input:**
    - Perform a series of random INSERT and DELETE operations.
    - After each set of operations, perform REVERSE and verify that the list is reversed correctly.
  + **Constraints:**
    - The reversal should preserve the correct order of nodes in the reversed list.

**Expected Output:** The list should be reversed after each REVERSE operation