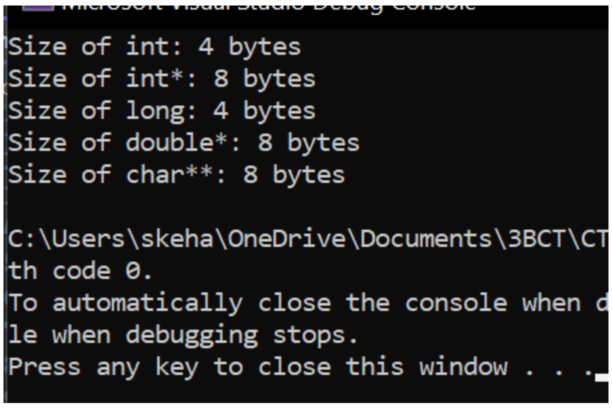
Programming Paradigms – Assignment 1

Question 1:

Output:



```
Size of int: 4 bytes
Size of int*: 8 bytes
Size of long: 4 bytes
Size of double*: 8 bytes
Size of char**: 8 bytes

C:\Users\skeha\OneDrive\Documents\3BCT\CT
th code 0.
To automatically close the console when d
le when debugging stops.
Press any key to close this window . . .
```

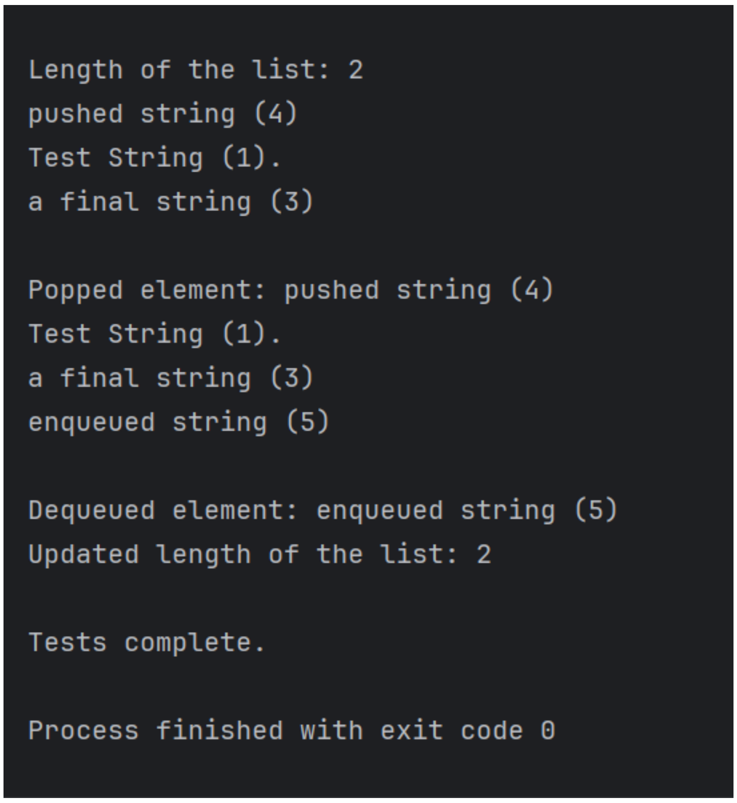
Code:

```c
#include <stdio.h>
#include "linkedList-1.c"
#include "tests.c"

int main(int argc, char* argv[]) {
    // Declare and initialize an integer variable 'x'
    int x = 23;
    // Print the size of 'int' in bytes
    printf("Size of int: %zu bytes\n", sizeof(x));

    // Declare a pointer to 'int' named 'y', initialize it to NULL
    int* y = NULL;
    // Print the size of 'int*' (pointer to int) in bytes
    printf("Size of int*: %zu bytes\n", sizeof(y));

    // Declare and initialize a 'long' variable 'z' with a large value
    long z = 1234567891L;
    // Print the size of 'long' in bytes
    printf("Size of long: %zu bytes\n", sizeof(z));

    // Declare a pointer to 'double' named 'i', initialize it to NULL
    double* i = NULL;
    // Print the size of 'double*' (pointer to double) in bytes
    printf("Size of double*: %zu bytes\n", sizeof(i));

    // Declare a pointer to a pointer to 'char' named 'j', initialize it to
NULL
    char** j = NULL;
    // Print the size of 'char**' (pointer to pointer to char) in bytes
    printf("Size of char**: %zu bytes\n", sizeof(j));

    runTests();

    return 0;
}
```

Results:

1. Size of int: 4 bytes - This result is consistent with many 32-bit and 64-bit systems where an int typically occupies 4 bytes of memory.
2. Size of int*: 8 bytes - On most 64-bit systems, a pointer to an int (int*) occupies 8 bytes due to the increased memory address space, while on 32-bit systems, it would typically be 4 bytes.
3. Size of long: 8 bytes - On most 64-bit systems, a long occupies 8 bytes of memory, allowing it to store larger values. On 32-bit systems, a long typically occupies 4 bytes.
4. Size of double*: 8 bytes - Pointers to doubles (double*) typically occupy 8 bytes on most systems, regardless of whether they are 32-bit or 64-bit.
5. Size of char**: 8 bytes - Pointers to pointers, such as char**, usually occupy 8 bytes on many systems due to the larger address space provided by 64-bit architectures.

Question 2:



```
Length of the list: 2
pushed string (4)
Test String (1).
a final string (3)

Popped element: pushed string (4)
Test String (1).
a final string (3)
enqueued string (5)

Dequeued element: enqueued string (5)
Updated length of the list: 2

Tests complete.

Process finished with exit code 0
```

Code:

linkedList.c

```c
// MY CODE STARTS HERE

// Returns the number of elements in a linked list.
int length(listElement* list) {
    int count = 0;
    listElement* current = list;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}

// Push a new element onto the head of a list.
void push(listElement** list, char* data, size_t size) {
    listElement* newEl = createEl(data, size);
    newEl->next = *list;
    *list = newEl;
}

// Pop an element from the head of a list.
listElement* pop(listElement** list) {
    if (*list == NULL) {
        return NULL;
    }

    listElement* popped = *list;
    *list = popped->next;
    popped->next = NULL; // Ensure popped element points to NULL
    return popped;
}

// Enqueue a new element onto the head of the list.
void enqueue(listElement** list, char* data, size_t size) {
    listElement* newEl = createEl(data, size);
    if (*list == NULL) {
        *list = newEl;
    }
    else {
        listElement* current = *list;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newEl;
    }
}

// Dequeue an element from the tail of the list.
listElement* dequeue(listElement* list) {
    if (list == NULL) {
        return NULL;
    }

    if (list->next == NULL) {
        listElement* dequeued = list;
        list = NULL; // The list is now empty
        dequeued->next = NULL;
        return dequeued;
    }

    listElement* current = list;
    while (current->next->next != NULL) {
        current = current->next;
    }

    listElement* dequeued = current->next;
    current->next = NULL;
    dequeued->next = NULL;
    return dequeued;
}
```

LinkedList.h

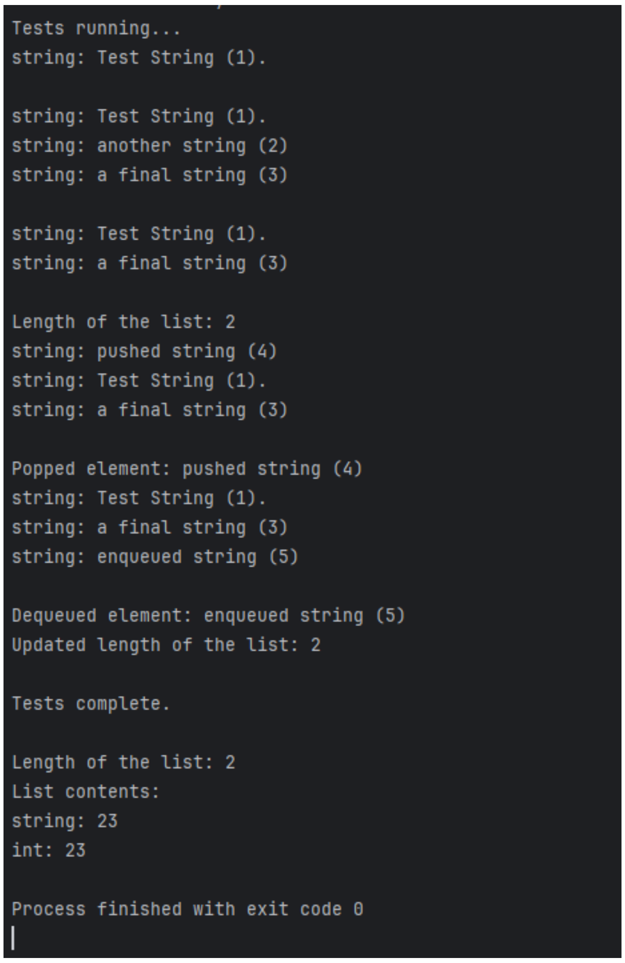
```
// MY CODE
int length(listElement* list);
void push(listElement** list, char* data, size_t size);
listElement* pop(listElement** list);
void enqueue(listElement** list, char* data, size_t size);
listElement* dequeue(listElement* list);
```

Test.c

```
// MY CODE
// Test length
int listLength = length(l);
printf("Length of the list: %d\n", listLength);

// Test push
push(&l, "pushed string (4)", 30);
traverse(l);
printf("\n");

// Test pop
listElement* poppedElement = pop(&l);
if (poppedElement) {
    printf("Popped element: %s\n", poppedElement->data);
    free(poppedElement->data);
    free(poppedElement);
}

// Test enqueue
enqueue(&l, "enqueued string (5)", 30);
traverse(l);
printf("\n");

// Test dequeue
listElement* dequeuedElement = dequeue(l);
if (dequeuedElement) {
    printf("Dequeued element: %s\n", dequeuedElement->data);
    free(dequeuedElement->data);
    free(dequeuedElement);
}

// Update length after manipulations
```

```
listLength = length(l);
printf("Updated length of the list: %d\n", listLength);
```

Question 3:



```
Tests running...
string: Test String (1).

string: Test String (1).
string: another string (2)
string: a final string (3)

string: Test String (1).
string: a final string (3)

Length of the list: 2
string: pushed string (4)
string: Test String (1).
string: a final string (3)

Popped element: pushed string (4)
string: Test String (1).
string: a final string (3)
string: enqueued string (5)

Dequeued element: enqueued string (5)
Updated length of the list: 2

Tests complete.

Length of the list: 2
List contents:
string: 23
int: 23

Process finished with exit code 0
```

Code:

genericLinkedList.c

```
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
#include "genericLinkedList.h"

listElement* createEl(void* data, size_t size, PrintFunction printFunction)
{
    listElement* e = malloc(sizeof(listElement));
    if (e == NULL) {
        return NULL;
    }

    void* dataPointer = malloc(size);
    if (dataPointer == NULL) {
        free(e);
        return NULL;
```

```c
    }

    memcpy(dataPointer, data, size);

    e->data = dataPointer;
    e->size = size;
    e->next = NULL;
    e->printFunction = printFunction;

    return e;
}

void traverse(listElement* start) {
    listElement* current = start;
    while (current != NULL) {
        current->printFunction(current->data);
        current = current->next;
    }
}

listElement* insertAfter(listElement* el, void* data, size_t size,
PrintFunction printFunction) {
    listElement* newEl = createEl(data, size, printFunction);
    listElement* next = el->next;
    newEl->next = next;
    el->next = newEl;
    return newEl;
}

void deleteAfter(listElement* after) {
    if (after != NULL && after->next != NULL) {
        listElement* delete = after->next;
        listElement* newNext = delete->next;
        after->next = newNext;

        free(delete->data);
        free(delete);
    }
}

int length(listElement* list) {
    int count = 0;
    listElement* current = list;
    while (current != NULL) {
        count++;
        current = current->next;
    }
    return count;
}

void push(listElement** list, void* data, size_t size, PrintFunction
printFunction) {
    listElement* newEl = createEl(data, size, printFunction);
    newEl->next = *list;
    *list = newEl;
}

listElement* pop(listElement** list) {
    if (*list == NULL) {
        return NULL;
    }
    listElement* popped = *list;
    *list = popped->next;
    popped->next = NULL;
    return popped;
}

void enqueue(listElement** list, void* data, size_t size, PrintFunction
printFunction) {
    listElement* newEl = createEl(data, size, printFunction);
    if (*list == NULL) {
        *list = newEl;
    } else {
        listElement* current = *list;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newEl;
    }
}

listElement* dequeue(listElement* list) {
    if (list == NULL) {
        return NULL;
    }

    if (list->next == NULL) {
        listElement* dequeued = list;
        list = NULL;
        dequeued->next = NULL;
        return dequeued;
    }

    listElement* current = list;
    while (current->next->next != NULL) {
        current = current->next;
    }

    listElement* dequeued = current->next;
    current->next = NULL;
    dequeued->next = NULL;
    return dequeued;
}
```

genericLinkedList.h

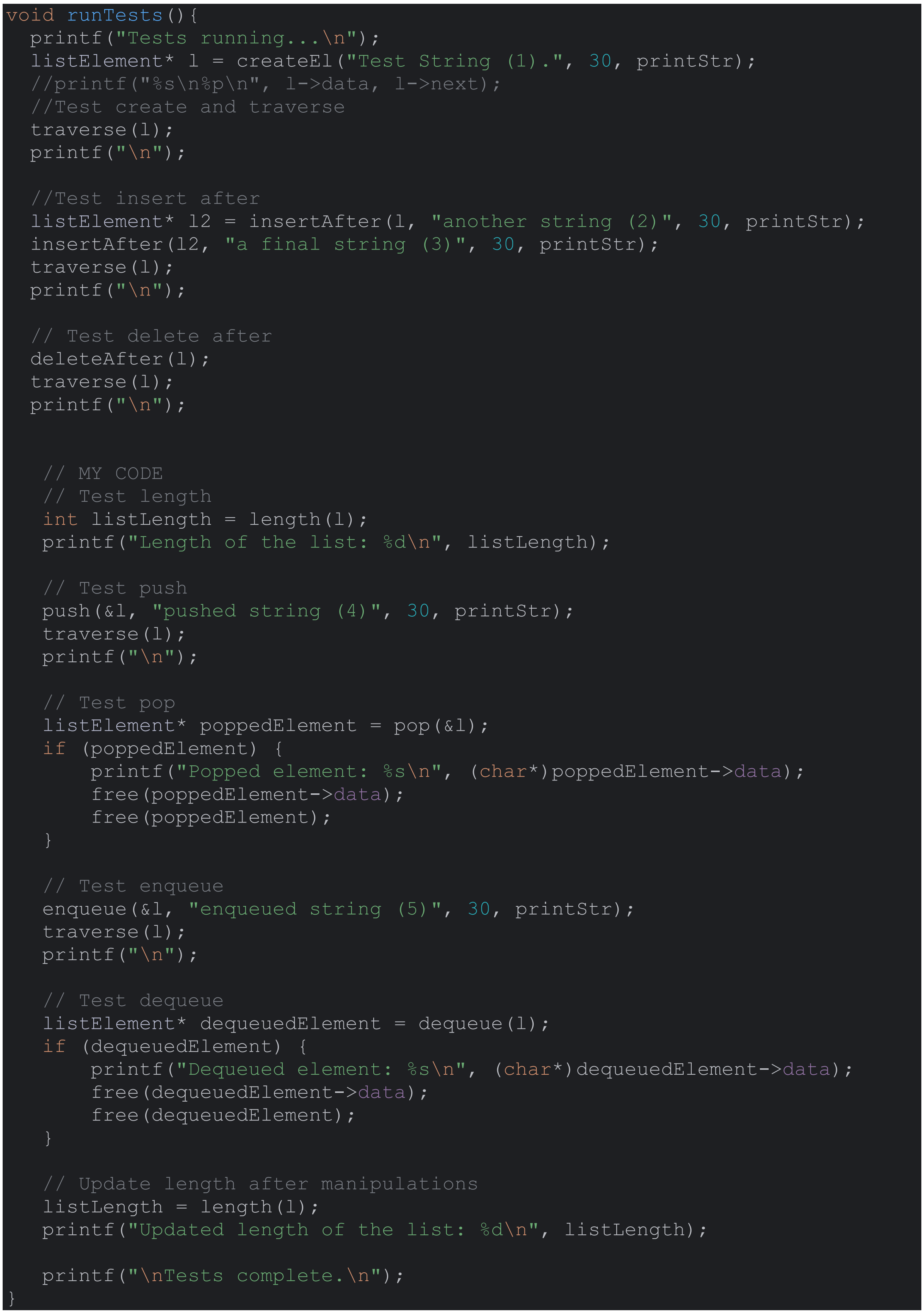
```c
#ifndef GENERIC_LINKED_LIST_H
#define GENERIC_LINKED_LIST_H

#include <stddef.h>

// Define a function pointer type for printing data
typedef void (*PrintFunction)(void* data);


typedef struct listElementStruct {
    void* data;
    size_t size;
    struct listElementStruct* next;
    PrintFunction printFunction; // Function pointer for printing data
} listElement;

listElement* createEl(void* data, size_t size, PrintFunction
printFunction);
void traverse(listElement* start);
listElement* insertAfter(listElement* el, void* data, size_t size,
PrintFunction printFunction);
void deleteAfter(listElement* after);
int length(listElement* list);
void push(listElement** list, void* data, size_t size, PrintFunction
printFunction);
listElement* pop(listElement** list);
void enqueue(listElement** list, void* data, size_t size, PrintFunction
printFunction);
listElement* dequeue(listElement* list);

#endif
```

Main.c

```

// Custom data elements
int intValue = 23;
char strValue[20];
snprintf(strValue, sizeof(strValue), "%d", x);

// Create an empty linked list
listElement* myList = NULL;

// Wrap the integer as a string for storage
char intValueStr[32];  // Adjust the buffer size as needed
snprintf(intValueStr, sizeof(intValueStr), "%d", intValue);

// Add elements to the list using the push function
push(&myList, &intValue, sizeof(int), printInt);
push(&myList, strValue, strlen(strValue) + 1, printStr);

// Print the length of the list
printf("\nLength of the list: %d\n", length(myList));

// Traverse the list to print its elements
printf("List contents:\n");
traverse(myList);
```

Tests.c

```
#include <stdio.h>
#include <stdlib.h>
#include "genericLinkedList.h"

void printStr(void* data) {
    printf("string: %s\n", (char*)data);
}

// Custom print functions for various data types
void printInt(void* data) {
    printf("int: %d\n", *(int*)data);
}

```



```c
void runTests(){
  printf("Tests running...\n");
  listElement* l = createEl("Test String (1).", 30, printStr);
  //printf("%s\n%p\n", l->data, l->next);
  //Test create and traverse
  traverse(l);
  printf("\n");

  //Test insert after
  listElement* l2 = insertAfter(l, "another string (2)", 30, printStr);
  insertAfter(l2, "a final string (3)", 30, printStr);
  traverse(l);
  printf("\n");

  // Test delete after
  deleteAfter(l);
  traverse(l);
  printf("\n");


   // MY CODE
   // Test length
   int listLength = length(l);
   printf("Length of the list: %d\n", listLength);

   // Test push
   push(&l, "pushed string (4)", 30, printStr);
   traverse(l);
   printf("\n");

   // Test pop
   listElement* poppedElement = pop(&l);
   if (poppedElement) {
       printf("Popped element: %s\n", (char*)poppedElement->data);
       free(poppedElement->data);
       free(poppedElement);
   }

   // Test enqueue
   enqueue(&l, "enqueued string (5)", 30, printStr);
   traverse(l);
   printf("\n");

   // Test dequeue
   listElement* dequeuedElement = dequeue(l);
   if (dequeuedElement) {
       printf("Dequeued element: %s\n", (char*)dequeuedElement->data);
       free(dequeuedElement->data);
       free(dequeuedElement);
   }

   // Update length after manipulations
   listLength = length(l);
   printf("Updated length of the list: %d\n", listLength);

   printf("\nTests complete.\n");
}
```

Question 4

a. **Comment on the memory and processing required to TRAVERSE a linked list in reverse (tail to head). [2.5 marks]**

- Traversing a linked list in reverse will generally require more memory and processing compared to traversing in the forward direction as linked lists are designed to traverse in the forward direction.

1. Memory Requirements: to traverse a linked list in reverse you need to maintain a stack or some other data structure to store each node as you visit. This extra data structure consumes additional memory, the memory requirements are directly proportional to the size of the linked list.
2. Processing Requirements: Traversing a linked list in reverse involves traversing from the head to the tail first to identify the tail node. Then you traverse the list again in reverse order. This requires two passes through the entire list, which doubles the processing time compared to a forward traversal.

b. **How could the structure of a linked list be changed to make this less intensive? [2.5 marks]**

- To make the traversing list in reverse less intensive you can change the structure of the linked list.

1. Doubly Linked List: In a doubly linked list, each node has a reference to both the next node and the previous node. You can traverse the list efficiently in both directions without the need for additional data structure to store nodes during reverse traversal. The only downside is that it requires more memory because it has an extra reference.
2. Caching: You can implement a caching mechanism by maintaining a cache of recently traversed nodes in reverse order. When you need to traverse the list in reverse, you can take advantage of the cache, which can significantly reduce processing time if you are accessing the same data multiple times.
3. Recursion: you can implement a recursive function to traverse the list in reverse. The function recursively moves to the end of the list and prints the data as it backwards. It is less memory efficient and may not suit long lists due to the risk of stack overflow.