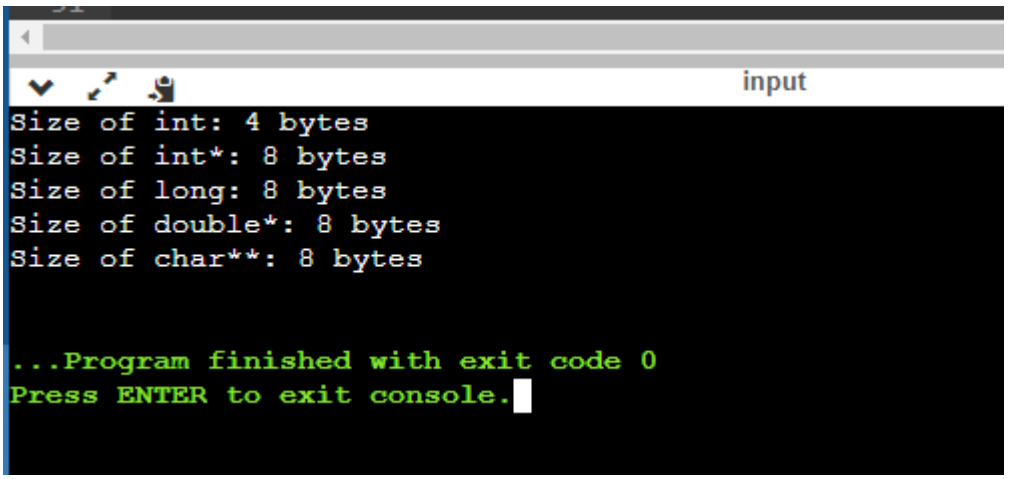
Assignment Name: CT331 Assignment 1
By: Paulius Zabinskas
Student ID: 20120267

Question 1

```
void question1(){
    // Creating variables of specified types
    // (architecture of my PC is 64-bit system)
    int intVar; // expected size 4 bytes (32-bit)

    int* intPtr;// expected size 8 bytes
    // stores the memory address of an integer variable

    long longVar;// expected size 8 bytes

    double* doublePtr;// expected size 8 bytes

    char** charPtrPtr;// expected size 8 bytes
    // Holds the address of an char*.


    // Printing out the sizes of the variables
    printf("Size of int: %zu bytes\n", sizeof(intVar));
    printf("Size of int*: %zu bytes\n", sizeof(intPtr));
    printf("Size of long: %zu bytes\n", sizeof(longVar));
    printf("Size of double*: %zu bytes\n", sizeof(doublePtr));
    printf("Size of char**: %zu bytes\n", sizeof(charPtrPtr));
}

int main()
{
    question1();
    return 0;
}
```



Question 2

File Name: main.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "linkedList.h"

int main() {
    // Create a new linked list
    listElement* list = createEl("Element 1", strlen("Element 1") + 1);

    // Print initial list
    printf("Initial list:\n");
    traverse(list);
    printf("Length: %d\n", length(list));

    // Test push operation
    push(&list, "Element 2", strlen("Element 2") + 1);
    printf("\nAfter push:\n");
    traverse(list);
    printf("Length: %d\n", length(list));

    // Test pop operation
    listElement* popped = pop(&list);
    printf("\nAfter pop:\n");
    traverse(list);
```

```
    printf("Popped element: %s\n", popped->data);
    printf("Length: %d\n", length(list));
    // Free the popped element data
    free(popped->data);
    // Free the popped element
    free(popped);

    // Test enqueue operation
    enqueue(&list, "Element 3", strlen("Element 3") + 1);
    printf("\nAfter enqueue:\n");
    traverse(list);
    printf("Length: %d\n", length(list));

    // Test dequeue operation
    listElement* dequeued = dequeue(list);
    printf("\nAfter dequeue:\n");
    traverse(list);
    if (dequeued != NULL) {
        printf("Dequeued element: %s\n", dequeued->data);
        // Free the dequeued element
        free(dequeued->data);
        free(dequeued);
    }
    printf("Length: %d\n", length(list));

    // Good practice
    // Free remaining elements in the list
    while (list != NULL) {
        listElement* next = list->next;
        free(list->data);
        free(list);
        list = next;
    }

    return 0;
}
```

File Name: linkedList.h

```
#ifndef CT331_ASSIGNMENT_LINKED_LIST
#define CT331_ASSIGNMENT_LINKED_LIST

typedef struct listElementStruct {
```

```
    char* data;
    size_t size;
    struct listElementStruct* next;
} listElement;

typedef struct listElementStruct listElement;

// int length(listElement* list)
// Returns the number of elements in a linked list

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);
#endif
```

File Name: linkedList.c

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

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

// Pop an element from the head of a list.
listElement* pop(listElement** list) {
    if (*list == NULL) return NULL;
    listElement* oldHead = *list;
    *list = (*list)->next;
    oldHead->next = NULL;  // Detach the old head from the list
```

```
    return oldHead;
}

// Enqueue a new element onto the head of the list.
void enqueue(listElement** list, char* data, size_t size) {
// Reusing push functionality as enqueue to head is same as push (I think)
    push(list, data, size);  }

// Dequeue an element from the tail of the list.
listElement* dequeue(listElement* list) {
    if (list == NULL || list->next == NULL) return NULL;  // Empty or single-element list
    listElement* prev = NULL;
    listElement* tail = list;
    while (tail->next != NULL) {
        prev = tail;
        tail = tail->next;
    }
    if (prev != NULL) {
        prev->next = NULL;  // Detach the last element
    }
    return tail;
}
```

Question 3

File Name: main.c

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

// Print functions
void printChar(void* data){
    printf("%c\n", *(char*)data);
}

void printInt(void* data){
    printf("%d\n", *(int*)data);
}
```

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

int main() {
    // Create a new linked list
    listElement* list = createEl("Element 1", strlen("Element 1") + 1, printStr);  // pass
printStr function pointer

    // Print initial list
    printf("Initial list:\n");
    traverse(list);
    printf("Length: %d\n", length(list));

    // Test push operation
    push(&list, "Element 2", strlen("Element 2") + 1, printStr);  // pass printStr function
pointer
    printf("\nAfter push:\n");
    traverse(list);
    printf("Length: %d\n", length(list));

    // Test pop operation
    listElement* popped = pop(&list);
    printf("\nAfter pop:\n");
    traverse(list);
    printf("Popped element: ");
    popped->printFunction(popped->data);  // use print function pointer
    printf("Length: %d\n", length(list));
    // Free the popped element
    free(popped->data);
    free(popped);

    // Test enqueue operation
    enqueue(&list, "Element 3", strlen("Element 3") + 1, printStr);  // pass printStr
function pointer
    printf("\nAfter enqueue:\n");
    traverse(list);
    printf("Length: %d\n", length(list));

    // Test dequeue operation
    listElement* dequeued = dequeue(list);
    printf("\nAfter dequeue:\n");
    traverse(list);
    if (dequeued != NULL) {
```

```
        printf("Dequeued element: ");
        dequeued->printFunction(dequeued->data);  // use print function pointer
        // Free the dequeued element
        free(dequeued->data);
        free(dequeued);
    }
    printf("Length: %d\n", length(list));

    // Free remaining elements in the list
    while (list != NULL) {
        listElement* next = list->next;
        free(list->data);
        free(list);
        list = next;
    }

    return 0;
}
```

File Name: genericLinkedList.h

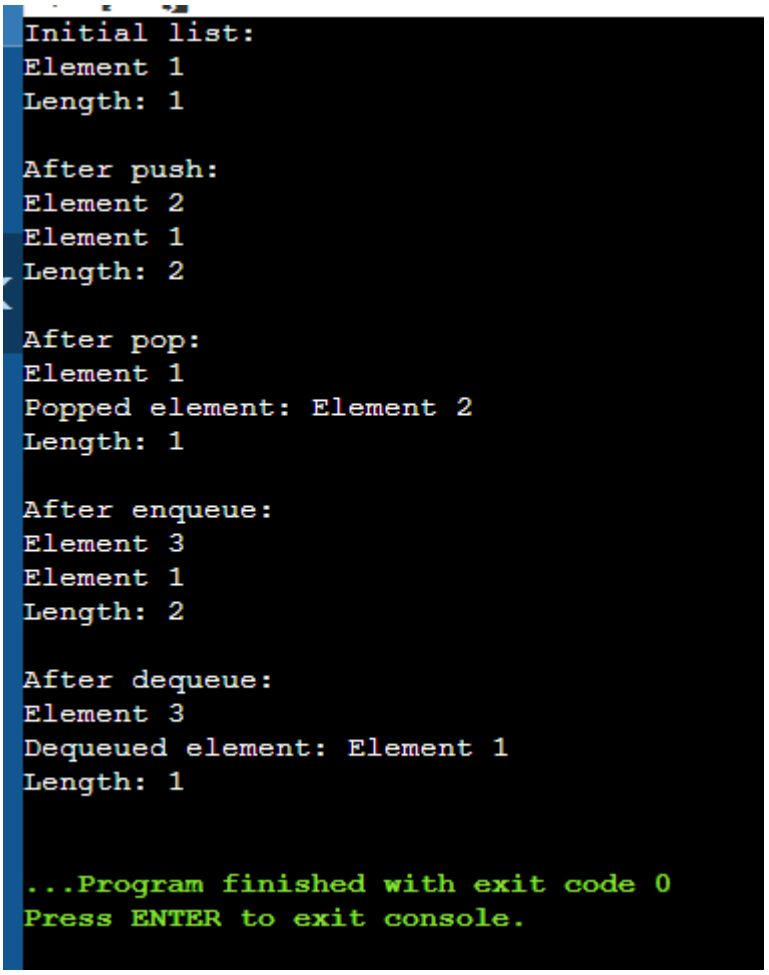
```
#ifndef CT331_ASSIGNMENT_LINKED_LIST
#define CT331_ASSIGNMENT_LINKED_LIST

typedef void (*printFunctionType)(void*);

typedef struct listElementStruct {
    void* data;
    size_t size;
    printFunctionType printFunction;
    struct listElementStruct* next;
} listElement;
typedef struct listElementStruct listElement;


int length(listElement* list);
void push(listElement** list, void* data, size_t size, printFunctionType print);
listElement* pop(listElement** list);
void enqueue(listElement** list, void* data, size_t size, printFunctionType print);
listElement* dequeue(listElement* list);
#endif
```

File Name: genericLinkedList.c

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



//Creates a new linked list element with given content of size
//Returns a pointer to the element
listElement* createEl(void* data, size_t size, printFunctionType print){
    listElement* e = malloc(sizeof(listElement));
    if(e == NULL){
        return NULL;
    }
    void* dataPointer = malloc(size);
    if(dataPointer == NULL){
        free(e);
        return NULL;
    }
    memcpy(dataPointer, data, size); // memcpy works on any data type
    e->data = dataPointer;
    e->size = size;
    e->printFunction = print;
    e->next = NULL;
    return e;
}
//Prints out each element in the list
void traverse(listElement* start){
    listElement* current = start;
    while(current != NULL){
        current->printFunction(current->data);
        current = current->next;
    }
}

//Inserts a new element after the given el
//Returns the pointer to the new element
listElement* insertAfter(listElement* el, void* data, size_t size, printFunctionType
print){
    listElement* newEl = createEl(data, size, print);
```

```
    listElement* next = el->next;
    newEl->next = next;
    el->next = newEl;
    return newEl;
}


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

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

// Pop an element from the head of a list.
listElement* pop(listElement** list) {
    if (*list == NULL) return NULL;
    listElement* oldHead = *list;
    *list = (*list)->next;
    oldHead->next = NULL;  // Detach the old head from the list
    return oldHead;
}

// Enqueue a new element onto the head of the list.
void enqueue(listElement** list, void* data, size_t size, printFunctionType print) {
    push(list, data, size, print);
}

// Dequeue an element from the tail of the list.
listElement* dequeue(listElement* list) {
    if (list == NULL || list->next == NULL) return NULL;  // Empty or single-element list
    listElement* prev = NULL;
    listElement* tail = list;
    while (tail->next != NULL) {
```

```
        prev = tail;
        tail = tail->next;
    }
    if (prev != NULL) {
        prev->next = NULL;  // Detach the last element
    }
    return tail;
}
```



Question 4

Question A: Comment on the memory and processing required to TRAVERSE a linked list in reverse (tail to head).

One possible strategy could be to store references to all nodes in an array or a stack data type. Current strategy would yield O(n) memory requirements, where n is the number of nodes in the list. The process would be quite simple. While traversing over the list, push each node reference onto the stack, or add it in an array. Once the end of the list is reached, pop nodes off the stack or iterate backward through the array.

The second strategy involves modifying the current singly linked list to support a doubly linked list structure. This modification would double the memory requirements used to store pointers, compared to singly linked lists, changing the memory requirements from O(n) to O(2n), although often this is still referred to as O(n). In this strategy, each node in the list would have two pointers: one pointing to the next element and one pointing to the previous element (with the head's previous pointer set to NULL). A method could then be created to traverse the list from the tail to the head, using the previous pointers to move from each node to its predecessor until the head is reached

How could the structure of a linked list be changed to make this less intensive?

Using a stack data structure can help with reverse traversal. As the list is being traversed from head to tail, push each node onto a stack. Once you reach the end of the list, pop each node off the stack. This will process the nodes in reverse order.