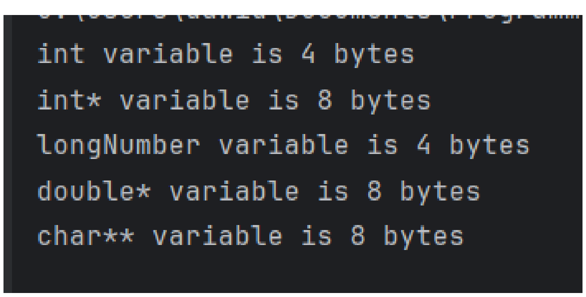
CT331 Assignment 1

Question 1

A)

```
// tests.h
#ifndef ASSIGNMENT1_TESTS_H
#define ASSIGNMENT1_TESTS_H

void runTests();

#endif //ASSIGNMENT1_TESTS_H

// tests.c
#include <stdio.h>
#include "tests.h"

void runTests()
{
    int integer;
    int * integerPointer;
    long longNumber;
    double* doublePointer;
    char** charDoublePointer;

    printf("int variable is %zu bytes\n", sizeof(integer));
    printf("int* variable is %zu bytes\n", sizeof(integerPointer));
    printf("longNumber variable is %zu bytes\n", sizeof(longNumber));
    printf("double* variable is %zu bytes\n", sizeof(doublePointer));
    printf("char** variable is %zu bytes\n", sizeof(charDoublePointer));
}
```

Output



```
int variable is 4 bytes
int* variable is 8 bytes
longNumber variable is 4 bytes
double* variable is 8 bytes
char** variable is 8 bytes
```

B)

Comments on the results

I am running a 64-bit windows version on an AMD processor. On 64-bit we would expect ints and longs to be 8 bytes. However, we get 4 bytes because I am running 64-bit window which reduces them to 4-bit for backwards compatability, for systems that are only 32 bit.

We see that pointers are 8 bytes as 8 bytes is the size of memory blocks on a 64 bit computer. Pointers only point at the memory location of whatever they are pointer to. A 32 bit computer would have a pointer size of 4 bytes.

Question 2

```
// linkedlist.h
// previous code

// Returns length of linked list
int length(listElement* list);

// Adds a new element at the top of stack
void push(listElement** list, char* data, size_t size);

// Removes and returns element at the top of stack
listElement* pop(listElement ** list);

// Add a new element in front of head.
void enqueue(listElement ** list, char* data, size_t size);

// Remove last element in linked list
listElement * dequeue(listElement * list);
```

```
// linkedlist.c
// previous code

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

```
    return count;
}

// Add element to the top
void push(listElement** list, char* data, size_t size)
{
    // create new element
    listElement * newEl = createEl(data, size);
    newEl->next = *list; // put the existing list as its next
    *list = newEl; // swap new element as lists head
}

// Remove element from top of stack
listElement* pop(listElement ** list)
{
    // check if list is empty
    if(*list == NULL){return NULL;}

    listElement * poppedElement = *list;
    *list = (*list)->next; // set the list head to its second element
    poppedElement->next = NULL;
    return poppedElement;
}

// Adds element to the front of linked list
void enqueue(listElement ** list, char* data, size_t size)
{
    // Create new element
    listElement * newEl = createEl(data, size);
    // set the list in front of it.
    newEl->next = *list;
    *list = newEl;
}

// Remove last element in linked list
listElement * dequeue(listElement * list)
{
    // Null check
    if(list == NULL)
    {
        return NULL;
    }else if(list->next == NULL){
        return list;
    }

    // Loop until we find second last element
    listElement * current = list;
    while(current->next->next != NULL){
        current = current->next;
    }

    // reference the last element
    listElement * dequeuedElement = current->next;
    current->next = NULL; // clear the second last element's next pointer

    // returned dequeue element
    return dequeuedElement;
}
```

Example code

```
int main() {
    // create 3 elements
    listElement * head = createEl("First" , 5);
    listElement  * second =
        insertAfter(head, "Second", 6);
    listElement  * third =
        insertAfter(second, "Third", 5);

    traverse(head);
    printf("\n");
    deleteAfter(second); // deletes element "Third"

    traverse(head);
    printf("\n");

    pop(&head); // removes first element i.e. "head"
    traverse(head);

    push(&head, "Fourth", 6); // Add to top
    printf("\n");
    traverse(head);

    enqueue(&head, "Fifth", 5); // Adds to front
    printf("\n");
    traverse(head);

    printf("\n");
    dequeue(head); // Removes from rear
    traverse(head);

    return 0;
}
```

```
First
Second
Third

First
Second

Second

Fourth
Second

Fifth
Fourth
Second

Fifth
Fourth

Process finished with exit code 0
```

Question 3

```
// genericLinkedList.h

#ifndef ASSIGNMENT1_GENERICLINKEDLIST_H
#define ASSIGNMENT1_GENERICLINKEDLIST_H

// pointer function
typedef void (*PrintFunction)(void*);
typedef struct genericListElementStruct genericElement;

// creates a new linked list element with given content, size, and print function
// returns a pointer to the element
```

```
genericElement * gen_createEl(void * data, size_t size, PrintFunction printFunction);

// Prints out each element in the linkedlist
void gen_traverse(genericElement * start);

//Inserts a new element after the given el
//Returns the pointer to the new element
genericElement * gen_insertAfter(genericElement * after, void * data, size_t size, PrintFunction printFunction);

//Delete the element after the given el
void gen_deleteAfter(genericElement * after);

// Returns length of linked list
int gen_length(genericElement * list);


// Add element to the top of the stack
void gen_push(genericElement ** list, void * data, size_t size, PrintFunction printFunction);

// Remove element from top of stack
genericElement * gen_pop(genericElement ** list);

// Add a new element in front of head.
void gen_enqueue(genericElement ** list, void * data, size_t size, PrintFunction printFunction);

// Remove last element in linked list
genericElement * gen_dequeue(genericElement * list);

// Print functions
void printInt(void * data);
void printStr(void * data);
void printChar(void * data);


#endif //ASSIGNMENT1_GENERICLINKEDLIST_H
```

```
// genericLinkedList.c

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

typedef struct genericListElementStruct{
    void * data;
    size_t size;
    struct genericListElementStruct * next;
    PrintFunction printFunction;

} genericElement;

// creates a new linked list element with given content, size, and print function
// returns a pointer to the element
genericElement * gen_createEl(void * data, size_t size, PrintFunction printFunction)
{
    // allocating memory for element
    genericElement * e = malloc(sizeof(genericElement));
    if(e == NULL){
```

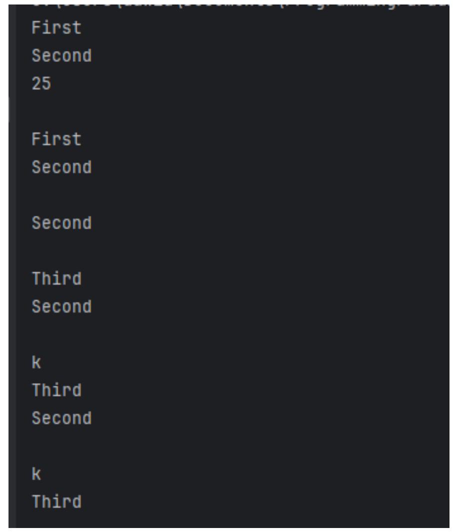
```
        // if malloc() failed
        return NULL;
    }

    // set elements size, allocate memory for data, and copy to it
    e->size = size;
    e->data = malloc(size);
    if(e->data == NULL){
        free(e);
        return NULL;
    }
    memcpy(e->data, data, size);
    e->next = NULL;
    e->printFunction=printFunction;
    return e;
}

// Prints out each element in the linkedlist
void gen_traverse(genericElement * start)
{
    genericElement * 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
genericElement * gen_insertAfter(genericElement * el, void * data, size_t size, PrintFunction printFunction)
{
    genericElement * newEl = gen_createEl(data, size, printFunction);
    genericElement * next = el->next;
    newEl->next = next;
    el->next = newEl;
    return newEl;
}

//Delete the element after the given el
void gen_deleteAfter(genericElement * after)
{
    genericElement  * delete = after->next;
    genericElement  * newNext = delete->next;

    after->next = newNext;

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

}

// Returns length of linked list
int gen_length(genericElement * list)
{
    int count = 0;
    genericElement * current = list;
    while(current != NULL)
    {
        count++;
        current = current->next;
    }

    return count;
```

```
}

// Add element to the top of the stack
void gen_push(genericElement ** list, void * data, size_t size, PrintFunction printFunction)
{
    genericElement * newEl = gen_createEl(data, size, printFunction);
    newEl->next = *list;
    *list = newEl;
}

// Remove element from top of stack
genericElement * gen_pop(genericElement ** list)
{
    if(*list == NULL){
        return NULL;
    }

    genericElement * poppedElement = *list;
    *list = (*list)->next;

    poppedElement->next = NULL;
    return poppedElement;
}

// Add a new element in front of head.
void gen_enqueue(genericElement ** list, void * data, size_t size, PrintFunction printFunction)
{
    genericElement * newEl = gen_createEl(data, size, printFunction);
    newEl->next = *list;
    *list = newEl;
}

// Remove last element in linked list
genericElement * gen_dequeue(genericElement * list) {
    // Null check
    if(list == NULL)
    {
        return NULL;
    }else if(list->next == NULL){
        return list;
    }

    // Loop until we find second last element
    genericElement * current = list;
    while(current->next->next != NULL){
        current = current->next;
    }

    // reference the last element
    genericElement * dequeuedElement = current->next;
    current->next = NULL; // clear the second last element's next pointer

    // returned dequeue element
    return dequeuedElement;
}

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

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

```
}

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

Example Code

```
int main() {
    char* firstString = "First";
    char* secondString = "Second";
    int number = 25;
    char* thirdString = "Third";
    char character = 'k';

    genericElement * head =
        gen_createEl(firstString ,
        sizeof(firstString), printStr);
    genericElement  * second =
        gen_insertAfter(head, secondString,
        sizeof(second) , printStr);
    genericElement  * third =
        gen_insertAfter(second, &number,
        sizeof(number), printInt);

    gen_traverse(head);
    printf("\n");
    gen_deleteAfter(second);

    gen_traverse(head);
    printf("\n");

    gen_pop(&head);
    gen_traverse(head);

    gen_push(&head, thirdString,
      sizeof(thirdString), printStr);
    printf("\n");
    gen_traverse(head);

    gen_enqueue(&head, &character,
      sizeof(character), printChar);
    printf("\n");
    gen_traverse(head);

    printf("\n");
    gen_dequeue(head);
    gen_traverse(head);

    return 0;
}
```



Question 4

- Transversing a linkedlist in reverse. You would first reverse the linkedlist and then running the transverse function on it.
    - Memory implication: O(1). The reversing of a linkedlist has a constant space complexity.
    - Processing implication: O(n). Reversing a linkedlist only loops through the list once.
    - Transversing the linkedlist: O(n): Loops through the list itself printing out each node.

Reversing a linkedlist doesn't require much memory or processing power. However, if we need to reverse our linkedlist frequently, a way of improving performance is by turning our linkedlist into a doubly linkedlist. This would be in cost of extra memory allocation as each node now needs two pointers (one for previous node, and one for next node). This would result in the processing implication of O(n), where n is a smaller factor than in a singly linkedlist.