CT331 Assignment 1

Programming Paradigms

Maxwell Maia 21236277

Question 1

(A)

```
int
Size: 4 bytes

int*
Size: 8 bytes

long
Size: 4 bytes

double*
Size: 8 bytes

char**
Size: 8 bytes
```

```c
#include <stdio.h>

int main() {
    //int
    int num = 68;
    printf("\nint\nSize: %llu bytes\n", sizeof(num));

    //int*
    int* numPointer = &num;
    printf("\nint*\nSize: %llu bytes\n", sizeof(numPointer));

    //long
    long bigLongNumber = 123456789;
    printf("\nlong\nSize: %llu bytes\n", sizeof(bigLongNumber));

    //double*
    double bigDoubleNumber = 5.67e+2;
    double* doublePointer = &bigDoubleNumber;
    printf("\ndouble*\nSize: %llu bytes\n", sizeof(doublePointer));

    //char**
    char myChar = 'a';
    char* myCharPointer = &myChar;
    char** myCharPointerToPointer = &myCharPointer;
    printf("\nchar**\nSize: %llu bytes\n", sizeof(myCharPointerToPointer));

    return 0;
```

(B)

The sizeof() function returns the size required to store a type on my computer in bytes. Using this on the variable gave the sizes for each. Longs and ints were 4 bytes. All the pointers, namely: int*, double* and char** were 8 bytes. This makes sense because they are all pointers so they should take up the same amount of space in memory.

The char** is an array of arrays. char** is a pointer to char*. char* is a pointer to a char. char** is effectively a 2d array

Question 2

linkedList.h

```
//Returns the number of elements in the list
int length(listElement* list);

//Push a new element onto the head of a list
void push(listElement** list, char* data, size_t size);

//Pop an element from the head of a list
listElement* pop(listElement** list);

//Enqueue a new element onto the head of the list
void enqueue(listElement** list, char* data, size_t size);

//Dequeue an element from the tail of the list
listElement* dequeue(listElement** list);
```

linkedList.c

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

```c
//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 there is no elements in the list
    if(*list == NULL)
    {
        return NULL;
    }

    listElement* head = *list;
    *list = head->next;
    head->next = NULL;
    return head;
}

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

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

    listElement* current = *list;
    listElement* previous = NULL;

    //Find the last element
    while (current->next != NULL)
    {
        previous = current;
        current = current->next;
    }

    //If the list has only 1 element, then update list to be NULL
    if (previous == NULL)
    {
        //Remove the only element in the list
        *list = NULL;
    }
    else
    {
        //Remove the last element from the list
        previous->next = NULL;
```

```
    }

    //Return the element dequeued
    return current;
}
```

Outputs

length:

```
Test String (1).
another string (2)
a final string (3)
length = 3
```

push:

```
Test push
This is the list before the push
Test String (1).
a final string (3)

Pushing a string to the head...
If you see this then push worked!
Test String (1).
a final string (3)
```

pop:

```
Test pop
This is the list before the pop

Pop from the head...
Test String (1).
a final string (3)
```

enqueue:

```
Test enqueue
This is the list before the enqueue
Test String (1).
a final string (3)

Enqueue a string to the head...
If you see this then enqueue worked!
Test String (1).
a final string (3)
```

dequeue:

```
Test dequeue
This is the list before the dequeue
If you see this then enqueue worked!
Test String (1).
a final string (3)

Dequeue from the head...
If you see this then enqueue worked!
Test String (1).
```

Question 3

Updated struct

```
typedef struct listElementStruct
{
    void* data;
    struct listElementStruct* next;
    void (*printFunction)(void*);
} listElement;
```

Custom print functions

```
// Custom print function for integers
void printInt(void* data)
{
    printf("%d\n", *(int*)data);
}
```

```c
// Custom print function for strings
void printStr(void* data)
{
    printf("%s\n", (char*)data);
}
```

Updated traverse function

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

genericLinkedList.h

```c
#ifndef GENERICLINKEDLIST_H
#define GENERICLINKEDLIST_H

//Updated struct that stores a function pointer
typedef struct listElementStruct
{
    void* data;
    struct listElementStruct* next;
    void (*printFunction)(void*);
} listElement;

//Create a new linked list element with the given data and print function.
listElement* createEl(void* data, void (*printFunction)(void*));

//Traverse the linked list and print each element using its print function.
void traverse(listElement* start);

//Insert a new element with the given data after the specified element.
void insertAfter(listElement* el, void* data, void
(*printFunction)(void*));

//Delete the element after the specified element.
void deleteAfter(listElement* after);

//Calculate the number of elements in the linked list.
int length(listElement* list);

//Push a new element onto the head of the list.
void push(listElement** list, void* data, void (*printFunction)(void*));

//Pop the element from the head of the list.
listElement* pop(listElement** list);

//Enqueue a new element onto the head of the list.
void enqueue(listElement** list, void* data, void (*printFunction)(void*));
```

```
//Dequeue an element from the tail of the list.
listElement* dequeue(listElement** list);


#endif
```

genericLinkedList.c

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

listElement* createEl(void* data, void (*printFunction)(void*))
{
    listElement* e = malloc(sizeof(listElement));
    if (e == NULL)
    {
        return NULL;
    }
    e->data = data;
    e->printFunction = printFunction;
    e->next = NULL;
    return e;
}

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

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

void deleteAfter(listElement* after)
{
    listElement* delete = after->next;
    listElement* newNext = delete->next;
    after->next = newNext;
    free(delete);
}

int length(listElement* list)
{
    int length = 0;
    listElement* current = list;
    while (current != NULL)
```

```
    {
        length++;
        current = current->next;
    }
    return length;
}

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

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

    listElement* head = *list;
    *list = head->next;
    head->next = NULL;
    return head;
}

void enqueue(listElement** list, void* data, void (*printFunction)(void*))
{
    listElement* newEl = createEl(data, printFunction);
    newEl->next = *list;
    *list = newEl;
}

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

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

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

    listElement* dequeued = current->next;
    current->next = NULL;

    return dequeued;
}
```

Outputs

```
Testing Question 3.

Create an empty list...

Push an integer onto the head of the list...

Enqueue a string onto the head of the list...


Let's look at the list...
Hello, world!
42


Pop an element from the head of the list and print it...
Hello, world!

Dequeue an element from the tail of the list and print it...
42


Let's look at the list again



Question 3 tests complete.
```

Question 4

Part 1

Memory and processing to traverse a linked list in reverse.

Let's call the number of elements in the linked list the variable "n".

To traverse in reverse:

Create a stack of size n.

Iterate through the linked list,
    push the object or reference to the object onto the stack. (n times)

After that…

Until the stack is empty
    Pop from the stack (n times)

Since the stack is a Last-In-First-Out data structure, popping elements will give you them in the reverse order. You can pop the elements to traverse them. This would not modify the elements of the linked list.

A linked list and stack with a capacity of at least n are required.

The memory requirements will be O(n) due to the stack's space usage.

The processing requirements will be O(n) because the elements need to be traversed twice, once when pushing them and once when popping them.

Part 2

If we instead use a doubly linked list we could traverse the linked list in reverse by starting at the tail and just following the pointer to the previous element. Traversing in reverse like this would cost O(n) processing time. The memory required would be O(n). The cost of this method is that the structure of each element would take more memory and storage because an additional pointer (the previous node pointer) needs to be added to the element to change the linked list to a doubly-linked list.