**Shane O Grady 16357921**

**CT331 Assignment 1**

Question 1(A):

#include <stdio.h>

#include <stdlib.h>

int main(int arg, char\* argc[]) {

int a = 0;

int\* b = 0;

long c = 0;

double \* d = 0;

char\*\* e = 0;

printf("Shane O Grady 16357921 \n");

printf("int - %d \n", sizeof(a));

printf("int\* - %d \n", sizeof(b));

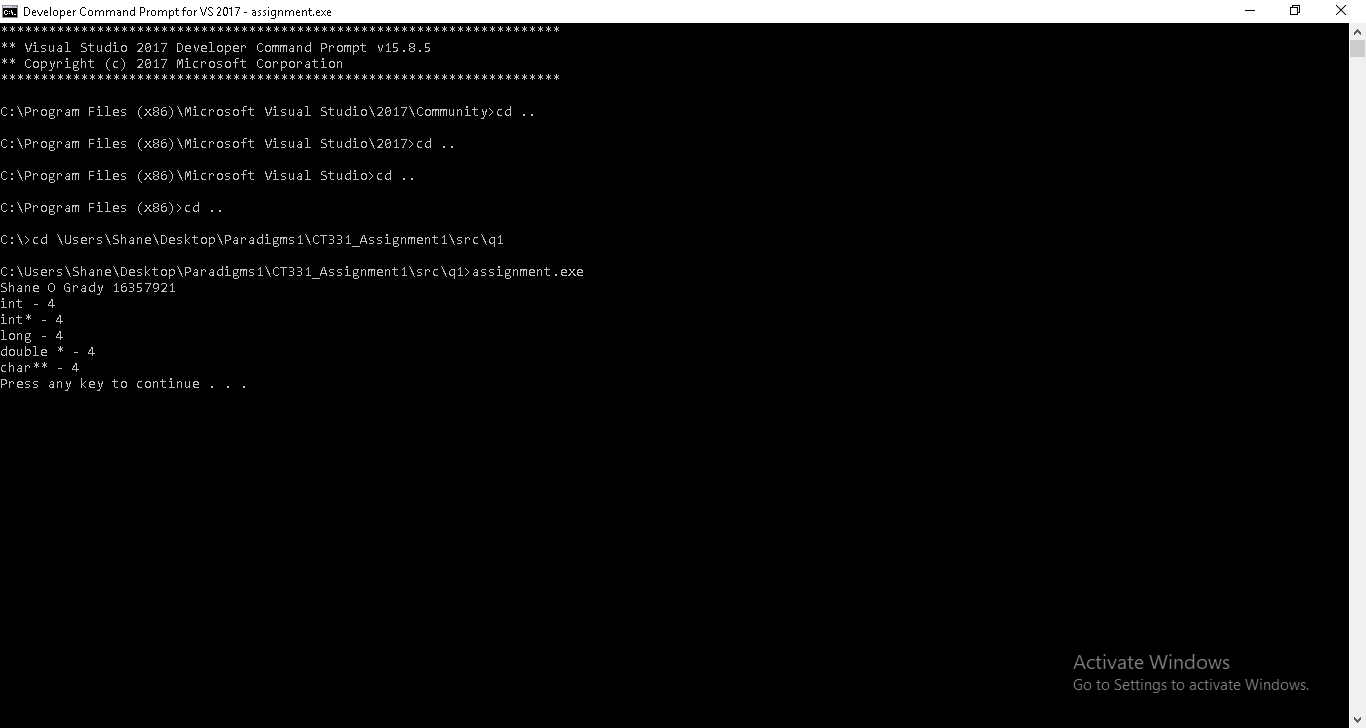
printf("long - %d \n", sizeof(c));

printf("double \* - %d \n", sizeof(d));

printf("char\*\* - %d \n", sizeof(e));

system("pause");

}



Question 1(B) :

The size of every primitive value turned out to be 4 which surprised me. However, I discovered that a pointer has 4 bytes which explains why the char and double have a result of 4. As well as this I discovered that both the int and long have a size of 4 bytes through research carried out online

Question 2:

**Linkedlist.h**

#ifndef CT331\_ASSIGNMENT\_LINKED\_LIST

#define CT331\_ASSIGNMENT\_LINKED\_LIST

typedef struct listElementStruct listElement;

//Creates a new linked list element with given content of size

//Returns a pointer to the element

listElement\* createEl(char\* data, size\_t size);

//Prints out each element in the list

void traverse(listElement\* start);

//Inserts a new element after the given el

//Returns the pointer to the new element

listElement\* insertAfter(listElement\* after, char\* data, size\_t size);

//Delete the element after the given el

void deleteAfter(listElement\* after);

//Returns the number of elements in a linked 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);

#endif

**Linkedlist.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "linkedList.h"

typedef struct listElementStruct{

char\* data;

size\_t size;

struct listElementStruct\* next;

} listElement;

//Creates a new linked list element with given content of size

//Returns a pointer to the element

listElement\* createEl(char\* data, size\_t size){

listElement\* e = malloc(sizeof(listElement));

if(e == NULL){

//malloc has had an error

return NULL; //return NULL to indicate an error.

}

char\* dataPointer = malloc(sizeof(char)\*size);

if(dataPointer == NULL){

//malloc has had an error

free(e); //release the previously allocated memory

return NULL; //return NULL to indicate an error.

}

strcpy(dataPointer, data);

e->data = dataPointer;

e->size = size;

e->next = NULL;

return e;

}

//Prints out each element in the list

void traverse(listElement\* start){

listElement\* current = start;

while(current != NULL){

printf("%s\n", current->data);

current = current->next;

}

}

//Inserts a new element after the given el

//Returns the pointer to the new element

listElement\* insertAfter(listElement\* el, char\* data, size\_t size){

listElement\* newEl = createEl(data, size);

listElement\* next = el->next;

newEl->next = next;

el->next = newEl;

return newEl;

}

//Delete the element after the given el

void deleteAfter(listElement\* after){

listElement\* delete = after->next;

listElement\* newNext = delete->next;

after->next = newNext;

//need to free the memory because we used malloc

free(delete->data);

free(delete);

}

// Returns the number of elements in a linked list.

int length(listElement\* list) {

int counter = 0;

listElement\* temp = list;

while (temp != NULL) {

temp = temp->next;

counter++;

}

return counter;

}

// Push a new element onto the head of a list.

void push(listElement\*\* list, char \*data, size\_t size) {

listElement\* node = createEl(data, size);

node->next = \*list;

\*list = node;

}

// Pop an element from the head of a list.

listElement\* pop(listElement\*\* list) {

if (\*list != NULL) {

listElement\* node = (\*list)->next;

\*list = (\*list)->next;

return node;

}

return \*list;

}

//Enqueue a new element onto the head of the list.

void enqueue(listElement\*\* list, char\* data, size\_t size) {

push(list, data, size);

}

//Dequeue an element from the tail of the list.

listElement\* dequeue(listElement\* list) {

listElement\* temp = list;

while ((temp->next)->next != NULL)

{

temp = temp->next;

}

listElement\* last = temp->next;

temp->next = NULL;

return last;

}

**Tests.c**

#include <stdio.h>

#include "tests.h"

#include "linkedList.h"

void runTests(){

printf("Tests running...\n");

listElement\* l = createEl("Test String (1).", 30);

//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);

insertAfter(l2, "a final string (3)", 30);

traverse(l);

printf("\n");

// Test delete after

deleteAfter(l);

traverse(l);

printf("\n");

//Test length

int num = length(l2);

printf("%d\n\n", num);

//Test enqueue

enqueue(&l, "Last string", 30);

traverse(l);

printf("\n");

//Test pop

pop(&l);

traverse(l);

printf("\n");

//Test push

push(&l, "Last string", 30);

traverse(l);

printf("\n");

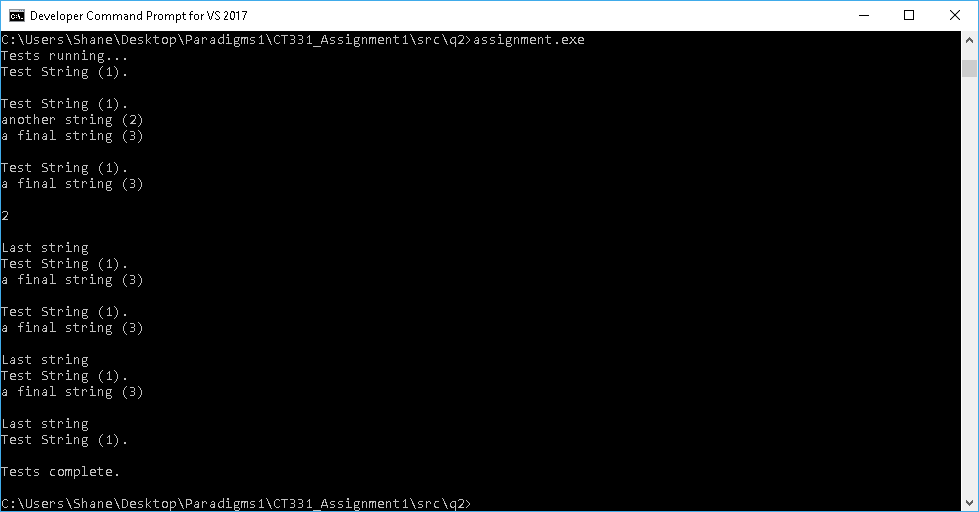
//Test dequeue

dequeue(l);

traverse(l);

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

}



Question 3:

**Genericlinkedlist.h**

#ifndef CT331\_ASSIGNMENT\_GENERIC\_LINKED\_LIST

#define CT331\_ASSIGNMENT\_GENERIC\_LINKED\_LIST

typedef void(\*printFn)(void\* data);

typedef struct genericlistElementStruct {

void\* data;

size\_t size;

printFn print;

struct genericlistElementStruct\* next;

} genericlistElement;

//Creates a new linked list element with given content of size

//Returns a pointer to the element

genericlistElement\* createEl(void\* data, size\_t size, printFn print);

//Prints out each element in the list

void traverse(genericlistElement\* start);

//Inserts a new element after the given el

//Returns the pointer to the new element

genericlistElement\* insertAfter(genericlistElement\* after, void\* data, size\_t size, printFn print);

//Delete the element after the given el

void deleteAfter(genericlistElement\* after);

//Returns the number of elements in a linked list

int length(genericlistElement\* list);

//Push a new element onto the head of a list.

void push(genericlistElement\*\* list, void\* data, size\_t size, printFn print);

//Pop an element from the head of a list.

genericlistElement\* pop(genericlistElement\*\* head);

//Enqueue a new element onto the head of the list.

void enqueue(genericlistElement\*\* list, void\* data, size\_t size, printFn print);

//Dequeue an element from the tail of the list.

genericlistElement\* dequeue(genericlistElement\* list);

//Prints out an integer element

void printInt(void\* data);

//Prints out a float element

void printFloat(void\* data);

//Prints out a char element

void printChar(void\* data);

//Prints out a string element

void printStr(void\* data);

#endif

**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

genericlistElement\* createEl(void\* data, size\_t size, printFn print) {

genericlistElement\* e = malloc(sizeof(genericlistElement));

if (e == NULL) {

//malloc has had an error

return NULL; //return NULL to indicate an error.

}

void\* dataPointer = malloc(size);

if (dataPointer == NULL) {

//malloc has had an error

free(e); //release the previously allocated memory

return NULL; //return NULL to indicate an error.

}

memmove(dataPointer, data, size);

e->data = dataPointer;

e->size = size;

e->print = print;

e->next = NULL;

return e;

}

//Prints out each element in the list

void traverse(genericlistElement\* head) {

genericlistElement\* current = head;

while (current != NULL) {

current->print(current->data);

current = current->next;

}

}

//Inserts a new element after the given el

//Returns the pointer to the new element

genericlistElement\* insertAfter(genericlistElement\* el, void\* data,

size\_t size, printFn print) {

genericlistElement\* newEl = createEl(data, size, print);

genericlistElement\* next = el->next;

newEl->next = next;

el->next = newEl;

return newEl;

}

//Delete the element after the given el

void deleteAfter(genericlistElement\* after) {

genericlistElement\* delete = after->next;

genericlistElement\* newNext = delete->next;

after->next = newNext;

//need to free the memory because we used malloc

free(delete->data);

free(delete);

}

// Returns the number of elements in a linked list.

int length(genericlistElement\* list) {

int counter = 0;

genericlistElement\* temp = list;

while (temp != NULL) {

temp = temp->next;

counter++;

}

return counter;

}

// Push a new element onto the head of a list.

void push(genericlistElement\*\* list, void\* data, size\_t size, printFn print) {

genericlistElement\* node = createEl(data, size, print);

node->next = \*list;

\*list = node;

}

// Pop an element from the head of a list.

genericlistElement\* pop(genericlistElement\*\* list) {

if (\*list != NULL) {

genericlistElement\* node = (\*list)->next;

\*list = (\*list)->next;

return node;

}

return \*list;

}

//Enqueue a new element onto the head of the list.

void enqueue(genericlistElement\*\* list, void\* data, size\_t size, printFn print) {

push(list, data, size, print);

}

//Dequeue an element from the tail of the list.

genericlistElement\* dequeue(genericlistElement\* list) {

genericlistElement\* temp = list;

while (temp->next->next != NULL) {

temp = temp->next;

}

genericlistElement\* last = temp->next;

temp->next = NULL;

return last;

}

void printChar(void\* data){

printf("%c\n", \*(char\*)data);

}

//Print an integer element

void printInt(void\* data){

printf("%d\n", \*(int\*)data);

}

//Print a float element

void printFloat(void\* data){

printf("%f\n", \*(float\*)data);

}

//Print a string element

void printStr(void\* data){

printf("%s\n", data);

}

**tests.c**

#include <stdio.h>

#include "tests.h"

#include "genericLinkedList.h"

void runTests(){

printf("Tests running...\n");

genericlistElement\* l = createEl("Initial test", 30, printStr);

//Test for create and traverse

traverse(l);

printf("\n");

//Test for insert after

int num = 123;

insertAfter(l, &num, sizeof(int), &printInt);

traverse(l);

printf("\n");

// Test for delete after

deleteAfter(l);

traverse(l);

printf("\n");

//Test for push

char a = 'a';

push(&l, &a, sizeof(char), &printChar);

traverse(l);

printf("\n");

//Test for length

printf("%d\n\n", length(l));

//Test for pop

pop(&l);

traverse(l);

printf("\n");

//Test for enqueue

float i = 22.8;

enqueue(&l, &i, sizeof(float), &printFloat);

traverse(l);

printf("\n");

//Test for dequeue

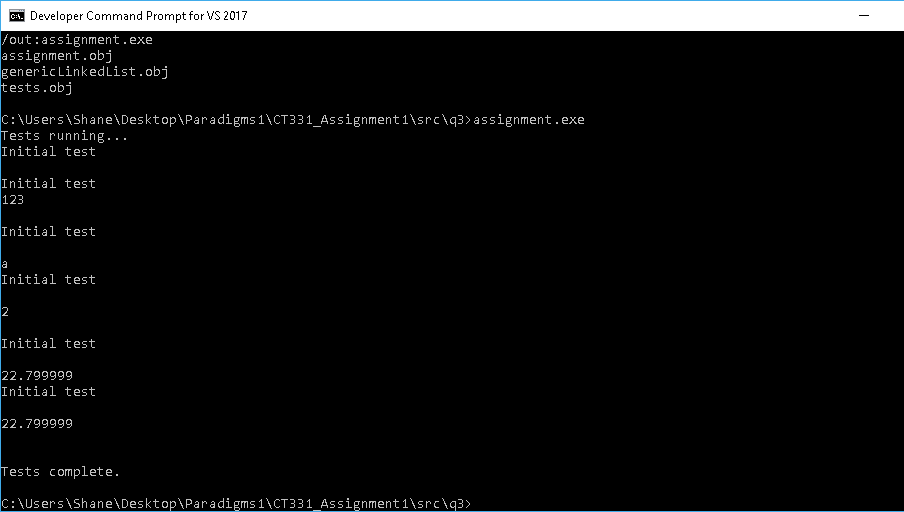
dequeue(l);

traverse(l);

printf("\n");

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

}



Question 4:

i)

Traversing a linked list in reverse will have high memory intensity as singly linked lists cannot be reversed directly. Once you reach the last node which is now the head you can traverse backwards. In order to traverse the linked list in reverse it is required to iterate one position back each time. This requires a lot of memory as a result.

ii)

To reduce memory intensity, the structure should be changed to a doubly linked list. Unlike the singly linked list, it can travel in two directions, next and previous. Depending on the size of the list, this can make a major impact when traversing in reverse as it would require much more memory to do so in a singly linked list.