

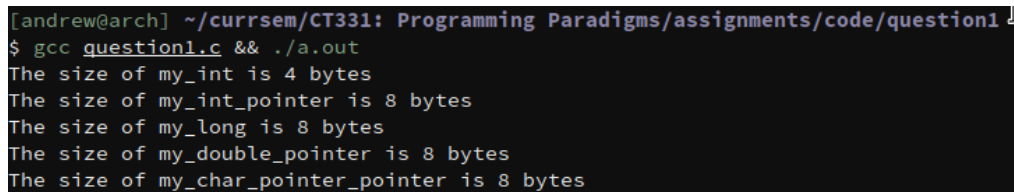
Assignment 1: Procedural Programming with C

1 Question 1

1.1 Part (A): Code

```
1 #include <stdio.h>
2
3 int main() {
4     int my_int;
5     int* my_int_pointer;
6     long my_long;
7     double * my_double_pointer;
8     char ** my_char_pointer_pointer;
9
10    printf("The size of my_int is %lu bytes\n", sizeof(my_int));
11    printf("The size of my_int_pointer is %lu bytes\n", sizeof(my_int_pointer));
12    printf("The size of my_long is %lu bytes\n", sizeof(my_long));
13    printf("The size of my_double_pointer is %lu bytes\n", sizeof(my_double_pointer));
14    printf("The size of my_char_pointer_pointer is %lu bytes\n", sizeof(my_char_pointer_pointer));
15 }
```

Listing 1: question1.c



```
[andrew@arch] ~/currsem/CT331: Programming Paradigms/assignments/code/question1.c
$ gcc question1.c && ./a.out
The size of my_int is 4 bytes
The size of my_int_pointer is 8 bytes
The size of my_long is 8 bytes
The size of my_double_pointer is 8 bytes
The size of my_char_pointer_pointer is 8 bytes
```

Figure 1: Console Output of question1.c

1.2 Part (B): Comments

The amount of memory allocated to variables of different types in C is determined at compile-time, and is dependent on the architecture of the machine for which it is being compiled and the compiler used.

- On my machine, using GCC, an `int` is allocated 4 bytes. This is the usual amount allocated on both 32-bit and 64-bit systems (my machine being of the latter kind), although older 32-bit systems used 2 bytes for an `int` (the same amount as for a `short`). 4 bytes is used even on 64-bit machines to maintain backwards compatibility with older 32-bit architectures.
- An `int*` (a pointer to a variable of type `int`) is allocated 8 bytes on my machine. This is because that my machine has a 64-bit architecture, and therefore an address in memory is represented using 64 bits (8 bytes). If this were compiled for a 32-bit machine, the size of a pointer would be 4 bytes since addresses are 32-bit.
- A `long` is allocated 8 bytes on my machine. This is because my machine is 64-bit and a `long` is typically 8 bytes in length on such machines. On 32-bit machines, a `long` is typically 4 bytes.
- The size of a pointer to a `double` is the same as the size of any other pointer on the same machine; on 64-bit machines, pointers are 8 bytes, and on 32-bit machines, they are 4 bytes. The type of data to which a pointer points has no effect on the size of the pointer, as the pointer is just a memory address.
- A pointer to a `char` pointer is the same size as any other pointer: 8 bytes on a 64-bit machine and 4 bytes on a 32-bit machine. Note: it might be more intuitive to refer to a “character pointer pointer” as a pointer to a string in certain situations, as strings are character arrays, and an array variable acts as a pointer to the first element in the array.

2 Question 2

```
1 // returns the number of elements in the list
2 int length(listElement* list);
3
4 // push a new element onto the head of a list and update the list reference using side effects
5 void push(listElement** list, char* data, size_t size);
6
7 // pop an element from the head of a list and update the list reference using side effects
8 listElement* pop(listElement** list);
9
10 // enqueue a new element onto the head of the list and update the list reference using side effects
11 void enqueue(listElement** list, char* data, size_t size);
12
13 // dequeue an element from the tail of the list
14 listElement* dequeue(listElement* list);
```

Listing 2: My Additions to linkedList.h

```
1 // returns the number of elements in the list
2 int length(listElement* list) {
3     int length = 0;
4     listElement* current = list;
5
6     // traversing the list and counting each element
7     while(current != NULL){
8         length++;
9         current = current->next;
10    }
11
12    return length;
13 }
14
15 // push a new element onto the head of a list and update the list reference using side effects
16 void push(listElement** list, char* data, size_t size) {
17     // create the new element
18     listElement* newElement = createEl(data, size);
19
20     // handle malloc errors
21     if (newElement == NULL) {
22         fprintf(stderr, "Memory allocation failed.\n");
23         exit(EXIT_FAILURE);
24     }
25
26     // make the the new element point to the current head of the list
27     newElement->next = *list;
28
29     // make the list reference to point to the new head element
30     *list = newElement;
31 }
32
33
34 // pop an element from the head of a list and update the list reference using side effects
35 // assuming that the desired return value here is the popped element, as is standard for POP operations
36 listElement* pop(listElement** list) {
37     // don't bother if list is non existent
38     if (*list == NULL) { return NULL; }
39 ;
40     // getting reference to the element to be popped
41     listElement* poppedElement = *list;
42 }
```

```

43 // make the the second element the new head of the list -- this could be NULL, so the list would be
↳ NULL also
44 *list = (*list)->next;
45
46 // detach the popped element from the list
47 poppedElement->next = NULL;
48
49 return poppedElement;
50 }
51
52
53 // enqueue a new element onto the head of the list and update the list reference using side effects
54 // essentially the same as push
55 void enqueue(listElement** list, char* data, size_t size) {
56 // create the new element
57 listElement* newElement = createEl(data, size);
58
59 // handle malloc errors
60 if (newElement == NULL) {
61     fprintf(stderr, "Memory allocation failed.\n");
62     exit(EXIT_FAILURE);
63 }
64
65 // make the the new element point to the current head of the list
66 newElement->next = *list;
67
68 // make the list reference to point to the new head element
69 *list = newElement;
70 }
71
72
73 // dequeue an element from the tail of the list by removing the element from the list via side effects,
↳ and returning the removed item
74 // assuming that we want to return the dequeued element rather than the list itself, as enqueue returns
↳ nothing and uses side effects, so dequeue should also use side effects
75 listElement* dequeue(listElement* list) {
76 // there are three cases that we must consider: a list with 0 elements, a list with 1 element, & a
↳ list with >=2 elements
77
78 // don't bother if list is non existent
79 if (list == NULL) { return NULL; }
80
81 // if there is only one element in the list, i.e. the head element is also the tail element, just
↳ returning this element
82 // this means that the listElement pointer that was passed to this function won't be updated
83 // ideally, we would set it to NULL but we can't do that since `list` is a pointer that has been
↳ passed by value, so we can't update the pointer itself. we would need a pointer to a pointer to
↳ have been passed
84 if (list->next == NULL) {
85     return list;
86 }
87
88 // traversing the list to find the second-to-last element
89 listElement* current = list;
90 while (current->next->next != NULL) {
91     current = current->next;
92 }
93
94 // get reference to the element to be dequeued
95 listElement* dequeuedElement = current->next;
96

```

```

97 // make the penultimate element the tail by removing reference to the old tail
98 current->next = NULL;
99
100 return list;
101 }

```

Listing 3: My Additions to linkedList.c

```

1 // test length function
2 printf("Testing length()\n");
3 int l_length = length(l);
4 printf("The length of l is %d\n\n", l_length);
5
6 // test push
7 printf("Testing push()\n");
8 push(&l, "yet another test string", sizeof("yet another test string"));
9 traverse(l);
10 printf("\n\n");
11
12 // test pop
13 printf("Testing pop()\n");
14 listElement* popped = pop(&l);
15 traverse(l);
16 printf("\n\n");
17
18 // Test delete after
19 printf("Testing deleteAfter()\n");
20 deleteAfter(l);
21 traverse(l);
22 printf("\n");
23
24 // test enqueue
25 printf("Testing enqueue()\n");
26 enqueue(&l, "enqueued test string", sizeof("enqueued test string"));
27 traverse(l);
28 printf("\n");
29
30 // test dequeue
31 printf("Testing dequeue()\n");
32 dequeue(l);
33 traverse(l);
34 printf("\n");
35
36 printf("\nTests complete.\n");

```

Listing 4: My Additions to tests.c

```

[andrew@arch] ~/cursem/CT331: Programming Paradigms/assignments/assignment1/code/question2
$ gcc *.c && ./a.out
Tests running...
Test String (1).

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

Testing length()
The length of l is 3

Testing push()
yet another test string
Test String (1).
another string (2)
a final string (3)

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

Testing deleteAfter()
Test String (1).
a final string (3)

Testing enqueue()
enqueued test string
Test String (1).
a final string (3)

Testing dequeue()
enqueued test string
Test String (1).

Tests complete.

```

Figure 2: Console Output for Question 2

3 Question 3

```

1  #ifndef CT331_ASSIGNMENT_LINKED_LIST
2  #define CT331_ASSIGNMENT_LINKED_LIST
3
4  typedef struct listElementStruct listElement;
5
6  //Creates a new linked list element with given content of size
7  //Returns a pointer to the element
8  listElement* createEl(void* data, size_t size, void (*printFunction)(void*));
9
10 //Prints out each element in the list
11 void traverse(listElement* start);
12
13 //Inserts a new element after the given el
14 //Returns the pointer to the new element
15 listElement* insertAfter(listElement* after, void* data, size_t size, void (*printFunction)(void*));
16
17 //Delete the element after the given el
18 void deleteAfter(listElement* after);
19
20 // returns the number of elements in the list
21 int length(listElement* list);
22
23 // push a new element onto the head of a list and update the list reference using side effects

```

```

24 void push(listElement** list, void* data, size_t size, void (*printFunction)(void*));
25
26 // pop an element from the head of a list and update the list reference using side effects
27 listElement* pop(listElement** list);
28
29 // enqueue a new element onto the head of the list and update the list reference using side effects
30 void enqueue(listElement** list, void* data, size_t size, void (*printFunction)(void*));
31
32 // dequeue an element from the tail of the list
33 listElement* dequeue(listElement* list);
34
35 #endif

```

Listing 5: genericLinkedList.h

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "genericLinkedList.h"
5
6 typedef struct listElementStruct{
7     void* data;
8     void (*printFunction)(void*);
9     size_t size;
10    struct listElementStruct* next;
11 } listElement;
12
13 //Creates a new linked list element with given content of size
14 //Returns a pointer to the element
15 listElement* createEl(void* data, size_t size, void (*printFunction)(void*)) {
16     listElement* e = malloc(sizeof(listElement));
17     if(e == NULL){
18         //malloc has had an error
19         return NULL; //return NULL to indicate an error.
20     }
21     void* dataPointer = malloc(sizeof(void)*size);
22     if(dataPointer == NULL){
23         //malloc has had an error
24         free(e); //release the previously allocated memory
25         return NULL; //return NULL to indicate an error.
26     }
27     strcpy(dataPointer, data);
28     e->data = dataPointer;
29
30     e->printFunction = printFunction;
31
32     e->size = size;
33     e->next = NULL;
34     return e;
35 }
36
37 //Prints out each element in the list
38 void traverse(listElement* start){
39     listElement* current = start;
40     while(current != NULL){
41         current->printFunction(current->data);
42         // printf("%s\n", current->data);
43         current = current->next;
44     }
45 }
46
47 //Inserts a new element after the given el

```

```

48 //Returns the pointer to the new element
49 listElement* insertAfter(listElement* el, void* data, size_t size, void (*printFunction)(void*)){
50     listElement* newEl = createEl(data, size, printFunction);
51     listElement* next = el->next;
52     newEl->next = next;
53     el->next = newEl;
54     return newEl;
55 }
56
57 //Delete the element after the given el
58 void deleteAfter(listElement* after){
59     listElement* delete = after->next;
60     listElement* newNext = delete->next;
61     after->next = newNext;
62     //need to free the memory because we used malloc
63     free(delete->data);
64     free(delete);
65 }
66
67 // returns the number of elements in the list
68 int length(listElement* list) {
69     int length = 0;
70     listElement* current = list;
71
72     // traversing the list and counting each element
73     while(current != NULL){
74         length++;
75         current = current->next;
76     }
77
78     return length;
79 }
80
81 // push a new element onto the head of a list and update the list reference using side effects
82 void push(listElement** list, void* data, size_t size, void (*printFunction)(void*)) {
83     // create the new element
84     listElement* newElement = createEl(data, size, printFunction);
85
86     // handle malloc errors
87     if (newElement == NULL) {
88         fprintf(stderr, "Memory allocation failed.\n");
89         exit(EXIT_FAILURE);
90     }
91
92     // make the the new element point to the current head of the list
93     newElement->next = *list;
94
95     // make the list reference to point to the new head element
96     *list = newElement;
97 }
98
99
100 // pop an element from the head of a list and update the list reference using side effects
101 // assuming that the desired return value here is the popped element, as is standard for POP operations
102 listElement* pop(listElement** list) {
103     // don't bother if list is non existent
104     if (*list == NULL) { return NULL; }
105     ;
106     // getting reference to the element to be popped
107     listElement* poppedElement = *list;
108

```

```

109 // make the the second element the new head of the list -- this could be NULL, so the list would be
    ↪ NULL also
110 *list = (*list)->next;
111
112 // detach the popped element from the list
113 poppedElement->next = NULL;
114
115 return poppedElement;
116 }
117
118
119 // enqueue a new element onto the head of the list and update the list reference using side effects
120 // essentially the same as push
121 void enqueue(listElement** list, void* data, size_t size, void (*printFunction)(void*)) {
122     // create the new element
123     listElement* newElement = createEl(data, size, printFunction);
124
125     // handle malloc errors
126     if (newElement == NULL) {
127         fprintf(stderr, "Memory allocation failed.\n");
128         exit(EXIT_FAILURE);
129     }
130
131     // make the the new element point to the current head of the list
132     newElement->next = *list;
133
134     // make the list reference to point to the new head element
135     *list = newElement;
136 }
137
138
139 // dequeue an element from the tail of the list by removing the element from the list via side effects,
    ↪ and returning the removed item
140 // assuming that we want to return the dequeued element rather than the list itself, as enqueue returns
    ↪ nothing and uses side effects, so dequeue should also use side effects
141 listElement* dequeue(listElement* list) {
142     // there are three cases that we must consider: a list with 0 elements, a list with 1 element, & a
    ↪ list with >=2 elements
143
144     // don't bother if list is non existent
145     if (list == NULL) { return NULL; }
146
147     // if there is only one element in the list, i.e. the head element is also the tail element, just
    ↪ returning this element
148     // this means that the listElement pointer that was passed to this function won't be updated
149     // ideally, we would set it to NULL but we can't do that since `list` is a pointer that has been
    ↪ passed by value, so we can't update the pointer itself. we would need a pointer to a pointer to
    ↪ have been passed
150     if (list->next == NULL) {
151         return list;
152     }
153
154     // traversing the list to find the second-to-last element
155     listElement* current = list;
156     while (current->next->next != NULL) {
157         current = current->next;
158     }
159
160     // get reference to the element to be dequeued
161     listElement* dequeuedElement = current->next;
162

```



```

163 // make the penultimate element the tail by removing reference to the old tail
164 current->next = NULL;
165
166 return list;
167 }

```

Listing 6: genericLinkedList.c

```

1 #include <stdio.h>
2 #include "tests.h"
3 #include "genericLinkedList.h"
4
5 // functions to print out different data types
6 // a more professional design might be to put these in the genericLinkedList header file but i only need
7 // these for testing purposes
8 void printChar(void* data) {
9     printf("%c\n", *(char*) data);
10 }
11 void printStr(void* data) {
12     printf("%s\n", (char*) data);
13 }
14 void printInt(void* data) {
15     printf("%d\n", *(int*) data);
16 }
17
18 void runTests(){
19     printf("Tests running...\n");
20
21     listElement* l = createEl("Test String (1).", sizeof("Test String (1)."), printStr);
22     //printf("%s\n%p\n", l->data, l->next);
23     //Test create and traverse
24     traverse(l);
25     printf("\n");
26
27     //Test insert after
28     printf("Testing insertAfter()\n");
29     listElement* l2 = insertAfter(l, "another string (2)", sizeof("another string (2)"), printStr);
30     insertAfter(l2, "a final string (3)", sizeof("a final string (3)"), printStr);
31     traverse(l);
32     printf("\n");
33
34     // test length function
35     printf("Testing length()\n");
36     int l_length = length(l);
37     printf("The length of l is %d\n", l_length);
38
39     // test push
40     printf("Testing push()\n");
41     push(&l, "yet another test string", sizeof("yet another test string"), printStr);
42     traverse(l);
43     printf("\n\n");
44
45     // test pop
46     printf("Testing pop()\n");
47     listElement* popped = pop(&l);
48     traverse(l);
49     printf("\n\n");
50
51     // Test delete after
52     printf("Testing deleteAfter()\n");
53

```

```

54 deleteAfter(l);
55 traverse(l);
56 printf("\n");
57
58 // test enqueue
59 printf("Testing enqueue()\n");
60 enqueue(&l, "enqueued test string", sizeof("enqueued test string"), printStr);
61 traverse(l);
62 printf("\n");
63
64 // test dequeue
65 printf("Testing dequeue()\n");
66 dequeue(l);
67 traverse(l);
68 printf("\n");
69
70 printf("Testing pushing different data types\n");
71 int myint = 42;
72 push(&l, &myint, sizeof(myint), printInt);
73 char mychar = 'c';
74 push(&l, &mychar, sizeof(mychar), printChar);
75 traverse(l);
76 printf("\n\n");
77
78 printf("\nTests complete.\n");
79 }

```

Listing 7: tests.c

```
[andrew@arch] ~/cursem/CT331: Programming Paradigms/assignments/assignment1/code/question3
$ gcc *.c && ./a.out
Tests running...
Test String (1).

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

Testing length()
The length of l is 3

Testing push()
yet another test string
Test String (1).
another string (2)
a final string (3)

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

Testing deleteAfter()
Test String (1).
a final string (3)

Testing enqueue()
enqueued test string
Test String (1).
a final string (3)

Testing dequeue()
enqueued test string
Test String (1).

Testing pushing different data types
c
42
enqueued test string
Test String (1).

Tests complete.
```

Figure 3: Console Output for Question 3

4 Question 4

4.1 Part 1

Any algorithm for traversing a singly linked list in reverse will always first require traversing the list forwards, and will therefore be *at least* somewhat less efficient than a forwards traversal. One of the simplest ways to traverse a linked list in reverse is to use a recursive function.

```
1 void reverse_traverse(listElement* current){
2     if (current == NULL) { return; }
3     reverse_traverse(current->next);
4     current->printFunction(current->data);
5 }
```

Listing 8: Recursive Function to Traverse a Singly Linked List in Reverse

This is quite inefficient as it requires that the function call for each node persists on the stack until the last node is reached, using a lot of stack memory. Another approach would be to iteratively reverse the linked list, by making some kind of data structure, linked list or otherwise, that contains the data of the original linked list but in reverse, and then iterating over that forwards. This would likely be more efficient in terms of memory & computation.

Because traversing a linked list in reverse always requires traversing it forwards first, any reverse algorithm will take at least twice as much memory & computation as traversing it forwards, which is $O(n)$. It will also require that some way of storing the data in reverse in memory, either explicitly with a data, like in the iterative approach, or in the same manner as the recursive approach, wherein the data is stored in reverse by the nested structure of the function calls: as each function call returns, the call structure

is iterated through in reverse. Therefore, we also have at least $O(n)$ memory usage, as we have to store some sort of reverse data structure.

4.2 Part 2

The simplest way in which the structure of a linked list could be changed to make backwards traversal less intensive is to change it from a singly linked list to a doubly linked list, i.e. instead of each node in the list containing a pointer to just the next node, make each node contain a pointer to both the next node & the previous node. The backwards traversal of a doubly linked list is no more intensive than the forwards traversal of a linked list. The drawback of using a doubly linked list is that it requires slightly more memory per node than a singly linked list, as you're storing an additional pointer for every node.