C Assignment Report

Luca Muscat

November 26, 2020

Contents

Task 1

In this task, a number of array-manipulating functions will have to be implemented. The functions which were implemented are as follows:

1. generate(): A function that populates an array with a sequence of N integers, in ascending order, and starting off integer i having a minimum value of 1. i should be accepted as a function argument while N is defined as a constant. This function is destructive in the sense that it overwrites any previously generated values.

2. shuffle(): A function that shuffles the items of an array argument and which makes use of stdlib.h's rand() function.

3. sort(): A function (implemented from scratch) that returns a sorted array passed as an argument

4. shoot(): A function that zeros out one element from, a possibly unsorted, array at random. This function returns an error if at least one element had already been previously zeroed out.

5. target(): A function that returns the number (i.e. the actual value and not the array offset) that was zeroed out by a single call to shoot().

Task 1a

In this task, the functions listed in section 1 will be implemented. For the sake of readability, the functions were split into two files which are the "functions.h" (which holds the function prototypes and their description via comments) file and "functions.c" which contains function's implementations.

struct myint_t

A structure called *myint_t*, which contains an int array called *nums* of size N and another int variable called *shoot_value*. Due to the nature of the *shoot()* and *target()* functions, the best way to keep track of what value has been shot out is by keeping the shot value (value from the filled nums array picked by random by *shoot()*) tied together with the array it got shot out of. This way, for every int array created, a *shoot_value* will be associated with it.

```
1 /*
2   Creating a structure which holds an array of ints and shot value.
3 */
4
5 typedef struct myint_t{
6         int nums[N];
7         int shoot_value;
8 } myint_t;
```

Listing 1: myint_t implementation

A helper function was also created to help with the creation of *myint_t* variables. Really and truely all this function does is return a *myint_t* variable with its *shoot_value* set to -1 (-1 will signify a null).

```
1 myint_t create_t(){
2         myint_t temp;
3         temp.shoot_value = -1;
4         return temp;
5 }
```

Listing 2: Helper function.

generate()

The generate function generate a sequence of consequetive numbers starting from i (the minimum number provided by the user which has to be a value above 0). Kindly note that the constant N is defined in utils.h

```
/*
  Populates an array with a sequence of N integers, in
  ascending order , starting off integer i having a minimum value of
  1. N is defined as a constant.  This function is destructive,
  meaning that it takes a pointer to the array and overwrites values.
  @param *array: Pointer to array which is going to have values generated in it.
  @param i: First number of the generated numbers.
  @return: 1 for an overflow, 0 for no overflow.
*/
int generate(int *array, int i);
```

Listing 3: Generate Function Prototypes

In lines 2 and 3 of listing 4, a conditional statement was added to keep the user from providing a number smaller than 1. Line 11 implements a basic overflow handling mechanism which checks that i is not in range of overflowing. Inside of this conditional statement, a simple for loop is used to populate the array.

```
int generate(myint_t *array, int i){
  if(i < 1)
        i = 1;
        /*
           Check for an overflow. Since checking for an overflow once it
           happened doesn't work out so well, we check if i is smaller
           than INT_MAX - N. If it isn't, that means that the int will
           overflow and wrap around to a negative number. Rather than
           dealing with
        */
        if(i < INT_MAX - N){
                for(size_t j = i; j < i + N; ++j){
                        array->nums[j - i] = j;
                }
                return 0;
        }
        printf("Overflow detected");
```

```
18          return OVERFLOW;
19  }
```

Listing 4: Generate Function Implementation

shuffle()

The shuffle function implements the Fisher-Yates algorithm which is a basic shuffle algorithm[1]. In short, i starts at the end of the array and is also used as a bound. With every iteration, i is decremented, reducing the possible range of random numbers which can be generated. Once a random number is generated, the element at i will be swapped with the random number. The following is an implementation of the fisher-yates shuffle algorithm. Kindly note that the random function used can be found in the *utils.c* file.

[1]Fisher-Yates Shuffle Algorithm Wikipedia: `https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle`

```c
/*
  Shuffles an array
  @param *array: Pointer to the array which should be shuffled.
*/
int shuffle(myint_t *array);
```

Listing 5: Shuffle function prototype

```c
int random(int range){
        /*
          This way a completely random seed will be used for every random
          number
        */
        srand(time(NULL));
        /*
          rand() % i generates random numbers from 1 to i-1.
          Hence the +1.
        */
        return rand() % (range + 1);
}
```

Listing 6: Random function implementation.

```c
/*
  Using Fisher-Yates shuffle algorithm
  https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

  Code taken from
  https://stackoverflow.com/questions/42321370/fisher-yates-shuffling-algorithm-in-c
  - Tectrendz
*/
int shuffle(myint_t *array){
        int tmp, j;
        if(array->nums[0] == -1){
                puts(NOT_GENERATED_ERROR);
                return NOT_GENERATED;
        }
```

```
15          for(size_t i = N-1; i > 0; --i){
16                  j = random(i);
17                  tmp = array->nums[j];
18                  array -> nums[j] = array -> nums[i];
19                  array -> nums[i] = tmp;
20          }
21  }
```

Listing 7: Fisher-Yates shuffle algorithm.

sort()

The sort function was implemented by using the widely known selection sort algorithm. The following snippet is the implementation of the sort function.

```
1  myint_t sort(myint_t* array){
2          /*
3            Allocate the N * size of int in bytes.  Then, return the pointer
4            to the allocated contiguous memory.  and assign it to pointer p.
5          */
6
7          /*
8            Dont forget to free the returned value at the end to avoid
9            mem leaks.
10         */
11
12         myint_t p = *array;
13
14         size_t min_index;
15
16         for(size_t i = 0; i < N - 1; ++i){
17                 min_index = i;
18                 for(size_t j = i + 1; j < N; ++j){
19                         if(p.nums[j] < p.nums[min_index]){
20                                 min_index = j;
21                         }
22                 }
23                 int temp = p.nums[i];
24                 p.nums[i] = p.nums[min_index];
25                 p.nums[min_index] = temp;
26         }
```

```
27          return p;
28  }
```

Listing 8: Integer sorting.

shoot()

All the *shoot()* function does is generate a random number bounded by constant N. It then uses that random number as an index which will be passed onto the *shoot_value* member of *myint_t*. That index will then be zeroed out in the array.

```
/*
  Zeroes out an element from the array, returns an error if an element
  has already been zeroed out. Will return a 1 if successful and a 0
  if an error has been encountered
  @param *array: Takes a myint_t struct which holds the int array and the shotgunned v
  @return: 1 if already shot (error), 0 if not shot.
*/
int shoot(myint_t *array);
```

Listing 9: Shoot function prototype.

```
int shoot(myint_t *array){
        /*
           Check if the shoot_value is null, meaning that it hasn't
           been shot before.
        */
        if(array -> shoot_value == -1 && array -> nums[0] != -1){
                // Rng(R) = {x: 0 <= x <= N-1}
                int j = random(N-1);
                array -> shoot_value = array -> nums[j];
                array -> nums[j] = 0;
                return 0;
        }
        if(array->nums[0] == -1){
                puts(NOT_GENERATED_ERROR);
                return NOT_GENERATED;
        }

        puts("Shoot function has already been called");
        return SHOT;
}
```

Listing 10: Shoot function implementation.

target()

The *target* function extracts the *shoot_value* from a *myint_t* variable and displays it. The first conditional statement checks if the array has been generated by checking if the first value in the int array is a -1. The second condition checks if the shoot value has been called previously. If all the checks pass, the shoot value will be printed to *stdout*.

```
int target(myint_t* array){
        if(array->nums[0] == -1){
                puts(NOT_GENERATED_ERROR);
                return NOT_GENERATED;
        }

        if(!array -> shoot_value){
                puts("Please shoot first.");
                return NO_TARGET;
        }
        return array -> shoot_value;
}
```

Listing 11: Target function implementation.

Task 1b

The full code of task 1b can be seen in *functions_str.c* and *functions_str.h* respectively. The solution of task 1b is similar to the one of task 1a but slightly different to accommodate for strings.

A global variable (however, doesn't have external linkage) called *NUMS_STR_REPR* was used to represent the numbers from zero to ten. The good thing about this implementation is that the index of the element directly corresponds with its string value. A different struct will also be used to hold the strings and their shoot_value, this struct will be called *mystr_t*.

```
/*
  Since the string representations of integers are limited from 1 to
  10, this means that the memory allocated to mystr_t is dynamic
*/
typedef struct mystr_t{
        char nums[10][STR_N];
        char shoot_value[STR_N];
        size_t size;
} mystr_t;
```

Listing 12: mystr_t implementation

```
char NUMS_STR_REPR[11][STR_N] = {
        "zero",
        "one",
        "two",
        "three",
        "four",
        "five",
        "six",
        "seven",
        "eight",
        "nine",
        "ten"
};
```

Listing 13: nums_str_repr implementation

str_create_t()

Initializes a *mystr_t* variable.

```
/*
   This helper function was used to help give shoot_value a
   pseudo-null value when initialized.
   @return: myint_t with shoot_value declared to -1
*/
mystr_t str_create_t();
```

Listing 14: String Generate Prototype

```
mystr_t str_create_t() {
        mystr_t temp;
        strcpy(temp.shoot_value, EMPTY);
        strcpy(temp.nums[0], EMPTY);
        temp.size = 0;
        return temp;
}
```

Listing 15: String Generate Function

str_generate()

A loop starting from i (minimum number provided by the user) ending at 10 goes through the *NUMS_STR_REPR* array and strcpy's it into the *mystr_t* member *nums*.

```c
/*
  Populates an array with the string representation of the numbers from i
  to 10 (minimum of i being 1). The data won't be stored as a 2D array.
  Everything will be placed in a 1D array, having each element split by a string
  terminator byte. This allows for the data to be tokenized.
  @param *array: Pointer to array which is going to have values generated in it.
  @param i: First number of the generated numbers.
*/
void str_generate(mystr_t *array, int i);
```

Listing 16: str_generate prototype

```c
void str_generate(mystr_t* array, int i){
        strcpy(array -> shoot_value, EMPTY);
        if (i < 1)
                i = 1;
        for(size_t j = i; j < 11; ++j){
                strcpy(array -> nums[j-i], NUMS_STR_REPR[j]);
        }
        array -> size = 10 - i + 1;
}
```

Listing 17: str_generate implementation

str_shuffle()

The same principle used in the number version of the shuffle function. However the only difference is that we are using strcpy to swap values around.

```
/*
  Shuffles an array
  @param *array: Pointer to the array which should be shuffled.
  @return: Returns a NOT_GENERATED error code if not generated.
*/
int str_shuffle(mystr_t *array);
```

Listing 18: str_shuffle prototype

```
/*
  Same philosophy as shuffle() in functions.c but uses strcpy to swap
  strings around.
*/
int str_shuffle(mystr_t* array){
        char tmp[STR_N];
        int j;

        if(array->size == 0){
                return NOT_GENERATED;
        }

        for (size_t i = 0; i < array->size; ++i) {
                j = random(i);
                strcpy(tmp, array -> nums[j]);
                strcpy(array -> nums[j], array -> nums[i]);
                strcpy(array -> nums[i], tmp);
        }
}
```

Listing 19: str_shuffle implementation

str_sort()

In this case, we are sorting the values depending on their context (for instance, “ten” comes after “two” even though lexicographically “ten” comes before “two”). This is done by going through each values in the *nums* array and comparing it with the elements in *NUMS_STR_REPR*. Once

the value is found, the index will be placed in an int array since the index directly corresponds to its value. Since the assignment asked for a sorted array to be returned, a variable of type *mystr_t* containing a sorted *nums* array will be returned (non-destructive).

```c
/*
  @param *array: Pointer to mystr_t struct which needs to be sorted.
  @return: Returns a pointer to a sorted array. This is due to the
  fact that the question asked to return a sorted array and not to
  sort an array.
*/
mystr_t str_sort(mystr_t* array);
```

Listing 20: str_sort prototype

```c
/*
  Use the predefined 2d char array to compare values this is
  convenient as each index maps to their respective number.
*/
int* str_to_int(mystr_t *array){
        int *temp = malloc(sizeof(int) * array->size);
        for(size_t i = 0; i < array -> size; ++i){
                temp[i] = -1;
                for(size_t j = 0; j < 11; ++j){
                        if(strcmp(array->nums[i], NUMS_STR_REPR[j]) == 0){
                                temp[i] = j;
                                break;
                        }
                }
        }
        return temp;
}

// Using selection sort.
mystr_t str_sort(mystr_t *array){
        mystr_t temp = *array;
        size_t min_index;
        int *int_repr = str_to_int(array);
        for(size_t i = 0; i < array->size; ++i){
                min_index = i;
                for(size_t j = i +1; j < array->size; ++j){
```

```
27                          if(int_repr[j] < int_repr[min_index])
28                                  min_index = j;
29                  }
30                  int_swap(&int_repr[i], &int_repr[min_index]);
31                  /*
32                      temp.nums is a 2d array so the address operator won't need
33                      to be used
34                   */
35                  str_swap(temp.nums[i], temp.nums[min_index]);
36          }
37          return temp;
38  }
```

Listing 21: str_sort implementation

As can be seen in listing 21 we first go through each generated string and create another array with their integer representation. Then, the int array is sorted, changing the string array the exact same way the int array is being changed. The method used to convert a string array to its respective integer array is called *str_to_int* (this can be seen in listing 21)

str_shoot()

This works exactly as its integer counterpart. But, instead of directly assigning the shoot value into the shoot value member in mystr_t, we use strcpy. Kindly note that the constant EMPTY has been defined as "empty"

```
1  int str_shoot(mystr_t *array){
2          if(strcmp(array -> shoot_value, EMPTY) == 0){
3                  int rand = random(array -> size);
4                  strcpy(array -> shoot_value, array -> nums[rand]);
5                  strcpy(array -> nums[rand], "zero");
6
7                  return 0;
8          }
9
10         if(strcmp(array->nums[0], EMPTY) == 0){
11                 return NOT_GENERATED;
12         }
13         return SHOT;
14 }
```

Listing 22: str_shoot implementation

str_target()

This works exactly as its integer couterpart. All it does is return *shoot_value* from the provided *mystr_t*.

```
// Extract the shoot_value from the mystr_t var.
char* str_target(mystr_t *array){
        if(strcmp(array->shoot_value, EMPTY) == 0)
                return "error";
        return array->shoot_value;
}
```

Listing 23: str_target implementation

Task 1c

In this task, I glued together all of the functions and placed them into a menu-like interface so that the user may interact with it. It consists of validation.

The code to the menu can be seen in listing 24

```
void menu() {
        int status;
        int option;
        myint_t data_int = create_t();
        mystr_t data_str = str_create_t();
        int function_status = -1;
        do{
                menu_print();
                status = scanf("%d", &option);
                puts("=============");
                int input;
                switch(option){
                case 1:
                        // Generate
                        puts("Please insert the number you want to start from.");
                        int_input(&input);
                        function_status = generate(&data_int, input);
                        break;
                case 2:
                        // Shuffle
                        function_status = shuffle(&data_int);
```

```
22                      if(function_status != NOT_GENERATED)
23                              int_print_array(&data_int);
24                      break;
25              case 3:
26                      // Sort
27                      // Technically the sort function still returns a new array.
28                      data_int = sort(&data_int);
29                      int_print_array(&data_int);
30                      break;
31          case 4:
32                      function_status = shoot(&data_int);
33                      break;
34              case 5:
35                      function_status = target(&data_int);
36                      if(function_status != NO_TARGET || function_status != NOT_GENERA
37                              printf("%d", function_status);
38                      }
39                      break;
40              case 6:
41                      // str generate
42                      puts("Please insert starting number (between 1 and 10)");
43                      do{
44                              int_input(&input);
45                      }while(input < 1 || input > 10);
46                      str_generate(&data_str, input);
47                      str_print_array(&data_str);
48                      break;
49              case 7:
50                      // str shuffle
51                      function_status = str_shuffle(&data_str);
52                      str_print_array(&data_str);
53                      break;
54              case 8:
55                      // str_sort
56                      if(strcmp(data_str.nums[0], EMPTY) == 0){
57                              puts(NOT_GENERATED_ERROR);
58                      }else {
59                              data_str = str_sort(&data_str);
60                              str_print_array(&data_str);
61                      }
62                      break;
```

```
63                  case 9:
64                          function_status = str_shoot(&data_str);
65                          if(function_status != SHOT)
66                                  puts(data_str.shoot_value);
67                          break;
68                  case 10:
69                          puts(str_target(&data_str));
70                  default:
71                          error_handling(function_status);
72                          break;
73                  }
74          }while(status == 1); // Stop if a character is encountered.
75  }
```

Listing 24: Menu code

This snippet is composed of a switch statement encapsulated in a do while loop (which allows the user to reinput their value and continue using the menu, even after the first interaction).

All the *menu_print* function does is display what options the user can use in the interface.

```
1  void menu_print() {
2          puts("==================");
3          puts("Welcome to the Problem Solving Assignment\nPlease select an option.");
4          puts("1. Generate \t 6. String Generate");
5          puts("2. Shuffle \t 7. String Shuffle");
6          puts("3. Sort \t 8. String Sort");
7          puts("4. Shoot \t 9. String Shoot");
8          puts("5. Target \t 10. String Target");
9          puts("Please insert a character to terminate");
10         puts("==================");
11 }
```

Listing 25: menu_print function

One may see which functions were mapped to which cases in the *menu_print* method.

Task 2

The solution to this task can be found in the task2 directory where one may find both solutions to task 2a and task 2b. Since task 2a asked for a more limited approach, I took the liberty of using normal arrays. In task 2b I used queues constructed of linked lists to create a more dynamic solution. Task 2b also consists of more recursive functions.

Task 2a

Structs used

```
1  typedef struct MsgQs_t {
2    // This needs to be a pointer to nodes since we need to have the
3    // ability to relinquish resources from specific nodes.
4    node *nodes[NODE_LIMIT];
5    unsigned int index;
6  } MsgQs_t;
7
8  /*
9    Node represets a message queue. Each node needs to contain a null byte
10   terminated char array (message) and an identifier (ID)
11 */
12
13 typedef struct node {
14   char *messages[MESSAGE_LIMIT];
15   int ID;
16   unsigned int index;
17 } node;
```

Listing 26: Structs used in task 2a

As seen in listing 26, two structs were used. The *MsgQs_t* was the data type which the assignment asked for. This struct contains the size (index) and an array of node pointers. The *node* struct consists of a char* array, ID (identifier) and size.

initializeMsgQs

Since the assignment asked for everything to be stored on heap, the data related to MsgQs_t must have been completely malloced (and freed later).

```
/*
  Since the MsgQs_t has to be placed in heap, this function will
  malloc, give it default values and return a pointer to it.
*/
MsgQs_t *initializeMsgQs();
```

Listing 27: initializeMsgQs function prototype

```
MsgQs_t *initializeMsgQs() {
  // Initialize all of the nodes.
  MsgQs_t *q = (MsgQs_t *)malloc(sizeof(MsgQs_t));
  // Initialize the queue
  q->index = 0;
  return q;
}
```

Listing 28: initializeMsgQs function implementation

As can be seen in listing 28, all this function does is malloc the *MsgQs_t* struct enough memory and set its size (index) to 0. The pointer to the initialized struct is returned.

unloadMsgQs

This function frees all of the memory allocated to an entire MsgQs_t. This is done by looping through every node and freeing every single message in it. Once all the messages are freed, so are the nodes themselves since they have still been allocated memory to hold the pointers. The MsgQs_t will also be freed for the same reason.

```
/*
  Simply free the MsgQs_t which is passed through the parameter. Of
  course in task 2a this will be a simple free. However, since linked
  lists are going to be used in task 2b, this process will have to be
  done recursively.

  @param *q: Pointer to MsgQs_t variable which needs to be
  relinquished.
```

```
  *q will also be set to null once the freeing process is done.
*/
void unloadMsgQs(MsgQs_t *q);
```

Listing 29: unloadMsgQs header

```
void unloadMsgQs(MsgQs_t *q) {
  // free nodes and msgqs_t
  for (size_t i = 0; i < q->index; ++i) {
    for (size_t j = 0; j < q->nodes[i]->index; ++j) {
      free(q->nodes[i]->messages[j]);
    }
    free(q->nodes[i]);
  }
  free(q);
}
```

Listing 30: unloadMsgQs implementation

createQ

The *createQ* functions takes a MsgQs_t which is to be populated, and an identifier which will identify each and every message queue. The method first checks if the message queue has reached its limit, if it hasn't, it will malloc enough memory to create a node and initialize it with all the necessary data. It will then increment the size of the message queue.

```
/*
  Places a new node inside of a specified MsgQs_t variable. Its
  identifier will be given as an arguement.

  @param q: A variable of type MsgQs_t which is going to have a
  new node added to it.
  @param identifier: Identifier for node.
  @return: Returns a status code, 0 is successful, ENQUEUE_ERROR
  otherwise
*/
int createQ(MsgQs_t *q, int identifier);
```

Listing 31: createQ function prototype

```c
int createQ(MsgQs_t *q, int identifier) {
  if (q->index < QUEUE_LIMIT) {
    q->nodes[q->index] = malloc(sizeof(node));
    q->nodes[q->index]->index = 0;
    q->nodes[q->index]->ID = identifier;
    q->index++;
    return SUCCESS;
  }
  return ENQUEUE_ERROR;
}
```

Listing 32: createQ function implementation

listQs

This method goes through a message queue's nodes and displays all of their messages. If the provided message queue is empty, the user will be notified. Otherwise, all of the messages will be printed to stdout.

```c
/*
  Places a new node inside of a specified MsgQs_t variable. Its
  identifier will be given as an arguement.

  @param q: A variable of type MsgQs_t which is going to have a
  new node added to it.
  @param identifier: Identifier for node.
  @return: Returns a status code, 0 is successful, ENQUEUE_ERROR
  otherwise
*/
int createQ(MsgQs_t *q, int identifier);
```

Listing 33: listQs function prototype

```c
void listQs(MsgQs_t *q) {
  for (size_t i = 0; i < q->index; ++i) {
    printf("=========\nIdentifier: %d\n", q->nodes[i]->ID);
    if (q->nodes[i]->index == 0) {
      puts("This Message Queue is Empty :(");
      continue;
```

```
    }
    for (size_t j = 0; j < q->nodes[i]->index; ++j) {
      printf("%d: %s\n", j, q->nodes[i]->messages[j]);
    }
  }
}
```

Listing 34: listQs function implementation.

deleteQ

deleteQ goes through every node in the message queue until it finds a matching identifier, once it finds it, it will go through a simple freeing procedure to return the memory back to the heap. If the deleted node is intermediary, the rest of the nodes will be shifted back by one.

```
/*
  When a queue is deleted, it will simply be overwritten by the
  following queues. If it's the last node, set it as null. If each
  node is also malloced and the individual queues are freed, that
  means that the max number of queues will decrease each time.

  @return: Returns DELETE_ERROR if the queue identifier doesn't exist.
*/
int deleteQ(MsgQs_t *q, int indentifier);
```

Listing 35: deleteQ function prototype

```
int deleteQ(MsgQs_t *q, int identifier) {
  for (size_t i = 0; i < q->index; ++i) {
    if (q->nodes[i]->ID == identifier) {
      free(q->nodes[i]);
      // Final node in queue
      if (i == q->index) {
        q->index--;
      } else {
        // Not final node in queue
        // Shift following nodes back by 1.
        for (size_t j = i; j < q->index - 1; ++j) {
          q->nodes[i] = q->nodes[i + 1];
        }
```

```
14          // free the last node since this was already shifted
15          // one back
16          free(q->nodes[q->index]);
17          q->index--;
18        }
19        return 0;
20      }
21    }
22    return DEQUEUE_ERROR;
23  }
```

Listing 36: deleteQ implementation

sendMessage

sendMessage goes through every node in the message queue until it finds a matching identifier. Once it finds the node it was looking for, it will allocate enough memory to fit the string inside the node and will append it to the list. However, if a NULL is provided instead of an actual identifier, the sendMessageBatch method will be called instead (this can be done due to the fact that the argument which takes the ID is of type void*). Kindly note that the sendMessageBatch method will be covered in the next subsection.

```
1  int sendIndividualMessage(node *node, char *message) {
2    if (node->index < MESSAGE_LIMIT) {
3      /*
4            strcpy is used instead of assigning the message arguement
5            message[node->index] since side effects might occur on the
6            value referenced by the message pointer arguement.
7            This will cause unexpected bugs when a reference to a
8            mutable char array is passed into the message
9            arguement. Once the mutable char array is modified outside
10           of the function's scope, the value saved in the node will
11           also change.
12     */
13     node->messages[node->index] = malloc(sizeof(char) * CHAR_LIMIT);
14     // Strncpy will also prevent messages longer than CHAR_LIMIT
15     // from being sent.
16     strncpy(node->messages[node->index], message, CHAR_LIMIT);
17     node->index++;
18     return SUCCESS;
19   }
```

```
20   return SEND_ERROR;
21 }
```

Listing 37: sendIndividualMessage helper function

```
1  /*
2    Sends a message (enqueues a node) to all nodes or a specific node if
3    an identifier is specified.
4
5  @param queue: MsgQs_t in which this function will operate.
6
7  @param identifier: If this is NULL, then the message is sent to all of
8  the queues (sendMessageBatch is sent). Otherwise it is directed to the
9  specified message queue
10
11 @param message: Message which will be sent.
12
13 @return: Return SEND_ERROR if identifier doesn't exist
14 */
15 int sendMessage(MsgQs_t *q, void *identifier, char *message);
```

Listing 38: sendMessage function prototype

```
1  /*
2    If the identifer is null, use sendMessageBatch, otherwise just send
3    the message
4  */
5  int sendMessage(MsgQs_t *q, void *identifier, char *message) {
6    if (identifier == NULL)
7      sendMessageBatch(q, message);
8    for (size_t i = 0; i < q->index; ++i) {
9      if ((int)identifier == q->nodes[i]->ID) {
10       sendIndividualMessage(q->nodes[i], message);
11       return SUCCESS;
12     }
13   }
14   return IDENTIFIER_ERROR;
15 }
```

Listing 39: sendMessage method implementation

sendMessageBatch

The sendMessageBatch method is similar to the sendMessage but instead of finding a select node to place the message in, it will place the message in every single node.

```
void sendMessageBatch(MsgQs_t *q, char *message) {
  for (size_t i = 0; i < q->index; ++i) {
    sendIndividualMessage(q->nodes[i], message);
  }
}
```

Listing 40: sendMessageBatch function implementation

receiveMessage

The receiveMessage method displays a number of messages specified by the user starting from the rear of the queue (FIFO) and removes it. First, the node which is going to be interacted with is found and placed into a temporary variable called node (if identifier isn't found, the variable will be left as NULL). If the node is NULL, the method will return -1, signifying an error, otherwise it will display and remove messages until either the index (size) becomes 0 (exhausting the node) or the number of messages becomes 0.

```
int receiveMessages(MsgQs_t *q, int identifier, size_t num_of_messages) {
  node *node;
  for (size_t i = 0; i < q->index; ++i) {
    if (q->nodes[i]->ID == identifier) {
      node = q->nodes[i];
    }
  }
  if (node == NULL)
    return -1;
  // Print it then free it.
  while (node->index > 0 && num_of_messages > 0) {
    printf("%d: %s", node->index, node->messages[node->index - 1]);
    free(node->messages[node->index]);
    node->index--;
    num_of_messages--;
  }
  return SUCCESS;
}
```

Listing 41: receiveMessages function implementation

purgeQs

purgeQs makes use of two helper functions which are:

- purgeIndividualNode - Goes through the provided node and frees all of its messages.
- purgeAllNodes - Goes through the provided message queue and calls purgeIndividualNode on every node in the message queue.

```
1  void purgeIndividualNode(node *node) {
2    for (size_t i = 0; i < node->index; ++i) {
3      free(node->messages[i]);
4    }
5    node->index = 0;
6  }
7
8  void purgeAllNodes(MsgQs_t *q) {
9    for (size_t i = 0; i < q->index; ++i) {
10     purgeIndividualNode(q->nodes[i]);
11   }
12 }
13
14
15 int purgeQs(MsgQs_t *q, void *identifier) {
16   // purge all
17   if (identifier == NULL)
18     purgeAllNodes(q);
19   int int_id = (int)identifier;
20   for (size_t i = 0; i < q->index; ++i) {
21     if (int_id == q->nodes[i]->ID) {
22       purgeIndividualNode(q->nodes[i]);
23       return SUCCESS;
24     }
25   }
26   return IDENTIFIER_ERROR;
27 }
```

Listing 42: purgeQs function implementation

purgeQs uses the same mechanism as sendMessage for handling single node purging or purging of an entire queue. This is done by using the void* mechanism which was previously described in sendMessage.

persistQ

This function simply checks if the node with the specified identifier exists and creates named after the identifier. First, the identifier is converted into a string and concatenated with the ".dat" extension. The largest 4 byte integer ($2^{32} - 1 = 4,294,967,295$ if signed) takes 10 bytes to be represented, add another four bytes for the extension and another byte for the null byte terminator. Meaning that the filename will consume a maximum of 15 bytes.

If a node with the same ID has already been persisted, an error will be thrown. However, if it hasn't been persisted from before hand, it will just utilize the fputs method to write each message into the file, each message will be separated by a newline.

```
1  int persistQ(MsgQs_t *q, int identifier) {
2    node *node;
3    for (size_t i = 0; i < q->index; ++i) {
4      // Go through all the nodes and check their ID
5      if (q->nodes[i]->ID == identifier)
6        node = q->nodes[i];
7    }
8    if (node == NULL)
9      return -1;
10
11   FILE *file;
12   char filename[26];
13   sprintf(filename, "%d.dat", identifier);
14   if ((file = fopen(filename, "w")) == NULL) {
15     printf("ERROR OPENING %s", filename);
16     fclose(file);
17     return -1;
18   }
19   for (size_t i = 0; i < node->index; ++i) {
20         // Go through each node and place it in the file, seperated by a
21         // new line.
22     fputs(strcat(node->messages[i], "\n"), file);
23   }
24   // Close the stream
25   fclose(file);
26   return SUCCESS;
27 }
```

Listing 43: persistQ function implementation

restoreQ

Since each message has been separated by a new line, the getline method has been used to extract the message from the file and pass it into a sendMessage method to repopulate the node.

```
int restoreQ(MsgQs_t *q, int identifier) {
  // search if identifier is in q
  for (size_t i = 0; i < q->index; ++i) {
    if (q->nodes[i]->ID == identifier)
      // return an error if found.
      return -1;
  }

  FILE *file;
  char filename[26];
  sprintf(filename, "%d.dat", identifier);
  if ((file = fopen(filename, "r")) == NULL) {
    puts("This identifier has never been persisted");
    return -1;
  }
  // snippet taken from https://c-for-dummies.com/blog/?p=1112
  char buffer[CHAR_LIMIT];
  char *message = buffer;
  size_t buffer_size = 256;
  createQ(q, identifier);
  while (getline(&message, &buffer_size, file) != -1) {
    sendMessage(q, (void *)identifier, message);
  }
  free(message);
}
```

Listing 44: restoreQ function implementation

Task 2b

The solution of this task will be more dynamic than the previous. Queues made out of linked lists will be utilized. Most of the methods will also the written recursively as they are easier to visualize and write. This solution will also be provided as an ADT, meaning that the API (task2b_functions.h) contains adequate documentation.

Structs used

Due to the complexity of this solution (A queue of queues), a substantial amount of structs were created in order to ease the process of creating this solution. Each queue struct (MsgQs_t and nodeMsg_t) contains a size member, this is very useful for writing iterative solutions as opposed to recursive ones. One may also notice that each struct contains a common member called bytes. This member is supposed to store the number of bytes written into it. This will help with RLP encoding later down the line. Since pre and post message delivery is encoded in RLP (and will be stored in the the struct containing RLP encoding) we need to ignore the number of bytes consumed by the RLP encoding itself.

```c
struct Message {
  char *subject;
  char *content;
  size_t bytes;
};
typedef struct Message Message;

/*
  As you may have noticed, an extra member called bytes has been added.
  This is going to help with figuring out the payload.
*/

struct Item {
  char *sender;
  Message *message;
  time_t expiry;
  struct Item *next;
  size_t bytes;
};
typedef struct Item Item;

struct nodeMsg_t {
  Item *front;
  Item *rear;
  int ID;
  size_t size;
  struct nodeMsg_t *next;
  size_t bytes;
};
typedef struct nodeMsg_t nodeMsg_t;
```

```
31
32 typedef struct MsgQs_t {
33   size_t size;
34   nodeMsg_t *front;
35   nodeMsg_t *rear;
36   size_t bytes;
37 } MsgQs_t;
```

Listing 45: Structs used in task2b (found in task2b_structs.h)

initializeMsgQs

This method returns an empty MsgQs_t with some initialized values.

```
1 MsgQs_t *initializeMsgQs() {
2   MsgQs_t *q = (MsgQs_t *)malloc(sizeof(MsgQs_t));
3   q->size = 0;
4   q->front = NULL;
5   q->rear = NULL;
6   return q;
7 }
```

Listing 46: initializeMsgQs function implementation

Line 2 Allocate enough memory to the *MsgQs_t* struct.

Line 3 Initialize the size of the message queue as 0.

Line 4-6 Initialize the front and the rear of the message queue as NULL and return its pointer.

```
(gdb) p *test
$3 = {size = 0, front = 0x0, rear = 0x0, bytes = 3131961357}
```

Figure 1: The values of an initialized Message Queue (gdb output)

unloadMsgQs

This method makes use of the free_node method which checks if the next node in the linked list is null, if it isn't null, it will call itself again but passes the next node in the linked list as an argument. This continues until the next node is NULL. Once the next node is NULL, all the items will be freed using the free_item method which also uses tail recursion to free the individual items. Tail recursion was used as opposed to a normal iterative approach since an iterative approach would require a doubly linked list. However, using recursion, we can just go to the very last object of the linked list, free all of its contents (and the memory allocated for holding the pointers) and simply bubble up the recursion stack freeing everything else in the process.

```
1  // Free everything using tail-recursion.  This way, we won't need a
2  // pointer to the previous element Since all we need to do is go back
3  // up the stack.
4  void free_node(nodeMsg_t *node) {
5    if (node->next != NULL)
6      free_node(node->next);
7    free_item(node->front);
8    free(node);
9    return;
10 }
```

Listing 47: free_node implementation

Line 5 If the next node isn't null, pass the next node into the free_node method.

Line 7 Once the last node has been reached, the items inside of it will be freed recursively (in a similar manner to how the node is being freed).

Line 8 Free the memory assigned to the node (pointers are allocated memory too).

```
1 void free_item(Item *Item) {
2   if (Item->next != NULL)
3     free_item(Item->next);
4   free_individual_item(Item);
5   return;
6 }
```

Listing 48: free_item implementation

Line 2 Check if the next Item in the linked list isn't NULL

Line 3 If the next node isn't null, pass it into the free_item method recursively.

Line 4 Free all of the contents of the Item

Line 5 Bubble up the recursion stack, once the end if reached, the popped stack will return to line 3 and continue this process until the stack frame is empty.

```
1 void free_individual_item(Item *Item) {
2   free(Item->message->content);
3   free(Item->message->subject);
4   free(Item->message);
5   free(Item->sender);
6   free(Item);
7 }
```

Listing 49: free_individual_item implementation

```
1 // Free everything.
2 void unloadMsgQs(MsgQs_t *q) {
3   nodeMsg_t *node = q->front;
4   free_node(node);
5   q->size = 0;
6 }
```

Listing 50: unloadMsgQs method implementation

Line 3-4 Provide the free_node method with the front of the message queue.

Line 5 Set its size to 0 (since the message queue is being emptied.

```
(gdb) p *test
$19 = {size = 0, front = 0xce1a00, rear = 0xce1a00, bytes = 3131961357}
(gdb) p test->front->ID
$20 = -17891602
```

Figure 2: unloadMsgQs results. As one may observe, the size is set to 0 and its members are all uninitialized.

createQ

The createQ method goes through a simple enqueue process to append the new node into the message queue.

```
1  // Enqueue a MsgQs_t variable.
2  int createQ(MsgQs_t *q, int identifier) {
3    if (contains_id(q->front, identifier))
4      return -1;
5    nodeMsg_t *tmp;
6    tmp = malloc(sizeof(nodeMsg_t));
7    // Setting the next pointer to null is important for the recursive
8    // methods
9    tmp->next = NULL;
10   tmp->ID = identifier;
11   tmp->size = 0;
12   // A basic enqueue procedure.
13   if (is_empty(q)) {
14     q->front = tmp;
15     q->rear = tmp;
16   } else {
17     q->rear->next = tmp;
18     q->rear = tmp;
19   }
20   q->size++;
21   return 1;
22 }
```

Listing 51: createQ implementation

Line 3 Check if the ID provided in the arguments exists

Line 4 Return -1 if *contains_id* returns a 0.

Line 5-11 Allocate enough memory to a temporary variable, set the next element in the linked list to NULL (this is done to identify that this is the most recent node), initialize the size to zero.

Line 13 Checks if the message queue is empty

Line 14-15 If the message queue is empty, set the front and the rear of the message queue to the temporary variable (the front and the rear both need to be initialized).

Line 17-18 Otherwise, set the rear node's next pointer to tmp, then set the rear to tmp. This means that the newest element in the linked list is always on the rear and is always connected to the node preceeding it.

The result of the method *createQ(test, 110)*, test being an initialized MsgQs_t pointer; (ie. it has been assigned to the result of initializeMsgQs) can be seen in figure 3

```
(gdb) p *test
$23 = {size = 1, front = 0xc917b0, rear = 0xc917b0, bytes = 3131961357}
```

Figure 3: createQ output.

listQs

listQs is another recursive method which takes a nodeMsg_t as its argument. Firstly, the size of the node is checked, if it's empty, the user will be notified. Otherwise, all the details will be printed using the listItems method. If the next node isn't null, we recursively call listQs again providing the next nodeMsg_t as an argument. Once it reaches the the last nodeMsg_t, the recursion stack will be popped from all of its stack frames.

```
1 void printItem(Item *Item) {
2   printf("\nSender: %s\n", Item->sender);
3   printf("Subject: %s\n", Item->message->subject);
4   printf("%s\n", Item->message->content);
5   printf("Expiry: %llu\n", Item->expiry);
6 }
```

Listing 52: printItem helper function

```
1 void listItems(Item *Item) {
2   printItem(Item);
3   if (Item->next != NULL)
4     listItems(Item->next);
5   return;
6 }
```

Listing 53: listItems helper function

Line 1 Arguments takes the front of a nodeMsg_t queue.

Line 2 Display the item provided in the argument.

Line 3 Check if there is another item in the item linked list.

Line 4 Recursively call listItems with the next item in the item linked list.

Line 5 Bubble up the recursion stack.

```
1  // List everything.
2  void listQs(nodeMsg_t *front) {
3    if (front->size == 0) {
4      puts("This is empty :/");
5    } else {
6      printf("\nID: %d", front->ID);
7      listItems(front->front);
8    }
9    if (front->next != NULL) {
10     return listQs(front->next);
11   }
12   return;
13 }
```

Listing 54: listQs method implementation

Line 2 front argument takes the front of a MsgQs_t queue.

Line 3 Check if the node is empty.

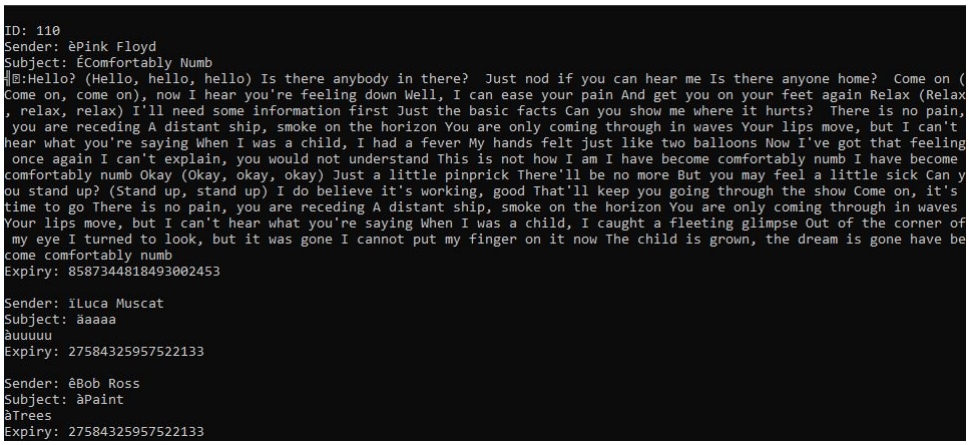
Line 6-7 print the ID of the node and use listItems (providing the front of the nodeMsg_t).

Line 9 Check if the next node isn't null.

Line 10 recursively call listQs with the next node in the linked list.

Line 12 Bubble up the recursion stack.

Figure 4: Output Listing of listQs(). In this figure one may observe that text which was previously inputted has been displayed


ID: 110
Sender: èPink Floyd
Subject: ÉComfortably Numb
╣:Hello? (Hello, hello, hello) Is there anybody in there? Just nod if you can hear me Is there anyone home? Come on (
Come on, come on), now I hear you're feeling down Well, I can ease your pain And get you on your feet again Relax (Relax
, relax, relax) I'll need some information first Just the basic facts Can you show me where it hurts? There is no pain,
you are receding A distant ship, smoke on the horizon You are only coming through in waves Your lips move, but I can't
hear what you're saying When I was a child, I had a fever My hands felt just like two balloons Now I've got that feeling
once again I can't explain, you would not understand This is not how I am I have become comfortably numb I have become
comfortably numb Okay (Okay, okay, okay) Just a little pinprick There'll be no more But you may feel a little sick Can y
ou stand up? (Stand up, stand up) I do believe it's working, good That'll keep you going through the show Come on, it's
time to go There is no pain, you are receding A distant ship, smoke on the horizon You are only coming through in waves
Your lips move, but I can't hear what you're saying When I was a child, I caught a fleeting glimpse Out of the corner of
my eye I turned to look, but it was gone I cannot put my finger on it now The child is grown, the dream is gone have be
come comfortably numb
Expiry: 8587344818493002453
Sender: ïLuca Muscat
Subject: äaaaa
àuuuuu
Expiry: 27584325957522133
Sender: êBob Ross
Subject: àPaint
àTrees
Expiry: 27584325957522133

deleteQ

In deleteQ, the front of the MsgQs_t provided is first checked to see if it matches with the specified identifier (this is done to avoid going into extra recursive methods). If the front of message queue provided matches the specified identifier, it will be freed and the front assigned to the next item linked to the freed item.

```
1 int deleteQ(MsgQs_t *q, int identifier) {
2   if (q->front->ID == identifier) {
3     nodeMsg_t *temp = q->front;
4     free_item(temp->front);
5     q->front = temp->next;
6     return 1;
7   }
8   return removeQ(q->front, identifier);
9 }
```

Listing 55: deleteQ method implementation

Line 2 Check if the front of the message queue matches the specified identifier.

Line 3 Create a temporary variable which holds the front (it's easier to write temp->front rather than q->front->front).

Line 4 Call the recursive method free_item providing the front of the node.

Line 5 Reassign the front of the message queue to the next node so that the front won't be empty.

Line 6 Return a success status.

Line 8 Call the recursive method removeQ, providing the front of the message queue and the specified identifier.

```
1  int removeQ(nodeMsg_t *front, int identifier) {
2    /*
3          Are we on the rear node? If so return an error since the ID
4          doesn't exist.
5    */
6    if (front->next == NULL)
7      return -1;
8
9    if (front->next->ID == identifier) {
10     nodeMsg_t *next = front->next;
11     next->size = 0;
12     // Free all the Qs items
13     free_item(next->front);
14     // Replace the next node with the next node's next node.
15     front->next = next->next;
16     // free the memory allocated to the item pointers.
17     free(next);
18     return 1;
19   }
20
21   /*
22     If the next node isn't null or an identifier, go to the next
23     node to repeat the process. This way, the result will bubble
24     back up the stack.
25   */
26
27   return removeQ(front->next, identifier);
28 }
```

Listing 56: removeQ helper function

Line 6-7 Check if the next node is NULL, if it is, a failure status will be returned (one might think that the last node isn't being checked, but since the first node provided to removeQ has already been checked outside of removeQ, we will always check if the next node's ID matches the specified identifier).

Line 9 Does the next node's ID match the specified identifier?

Line 11 Set its size to 0 since we are emptying it.

Line 12 Free all of the items it contains.

Line 15 Reassign the next pointer variable to the next nodes next pointer (this way, everything will remain linked).

Line 17 Free the node from memory allocated to its members .

Line 18 Return a success status.

Line 27 Recursively call removeQ providing the next node in the provided nodeMsg_t's linked list. If it hasn't been found, -1 will bubble up the recursion stack, otherwise, 1 will bubble up the recursion stack without having to go through all the other elements.

In this example, the method *deleteQ(test, 96)* will be called, test being the initialized message queue, and 96 being the ID of the node.

```
(gdb) p {test.front->ID, test.rear->ID}
$31 = {110, 96}
```

Figure 5: Proof that a node with ID 96 has been created.

```
(gdb) p {test.front->ID, test.rear->ID}
$32 = {110, -17891602}
```

Figure 6: One can see that the rear node which previously had the ID of 96 is now uninitialized.

RLP

Before I describe the rest of the methods, I think that it is important to describe how I have implemented RLP[2] encoding and decoding since from here on out, RLP will be used. In listing 57, one may find all of the code related to RLP procedures.

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
  The offsets and values needed were taken off the RLP ethereum wiki
  provided in the footnote of the assignment.
*/

size_t byte_len(size_t number) {
  /*
    The logic behind this is simple. Since binary numbers are base 2,
    we are applying log_2 on the number in order to get the number of
    bits needed to represent it. Flooring it will give us the the
    least possible value (look at it as a forced round down no matter
    what the fractional part is) and then we add by 1 so that any
    decimal value which was truncated can be made up for (not only
    that but you can't really represent a fraction of a bit). We are
    then dividing bit_len by 8 to get the number of bits needed and we
    are finding its ceiling (opposite of the floor, forced round up)
    This ensures that we always have enough bytes to represent all our
    bits.
*/
  double bit_len = floor(log2(number)) + 1.0;
  return ceil(bit_len / 8.0);
}

/*
  This method will be used to encode objects with a payload larger
  than 55 bytes. The flexibility of this method comes from encoding
  both strings and arrays.
*/
```

[2]Ethereum RLP wiki: `https://github.com/ethereum/wiki/wiki/RLP`

```c
char *encode_long_object(size_t length, char offset) {
  size_t byte_length = byte_len(length); // +2 to hold the offset and the null byte.
  size_t buffer_size = sizeof(char) * (byte_length + 2);
  char *buffer = malloc(buffer_size);
  // dont forget to free
  buffer[0] = offset + byte_length;
  buffer[buffer_size - 1] = '\0';
  size_t j = 0;
  for (size_t i = buffer_size - 2; i > 0; --i, ++j)
    buffer[i] = length >> (8 * j);
  return buffer;
}
// return the number of bytes returned in order to avoid strlen.
char *str_encode_length(size_t length) {
  size_t byte_length = byte_len(length);
  char *buffer;
  char offset;
  if (length < 56) {
    offset = 0x80;
    buffer = malloc(sizeof(unsigned char) * 2);
    buffer[0] = length + offset;
    buffer[1] = '\0';
    return buffer;
  } else {
    offset = 0xb7;
    return encode_long_object(length, offset);
  }
  return NULL;
}

// Returns the size of the rlp encoded string.
size_t rlp_strlen(char *string) {
  size_t length = strlen(string);
  if (length < 56)
    return length + 2;
  // We are adding one for the offset
  return length + byte_len(length) + 2;
}

void rlp_encode_str(char *dest, char *src) {
  size_t size = strlen(src);
```

```
75   /*
76         If the size if one, the message is self encoding.  If the size
77         is less than 56 and not 1, the encoding The encoding will be the
78         offset 0x80 + the size. Otherwise, use the encode_long_object method.
79   */
80   if (size == 1) {
81     dest[0] = src[0];
82     dest[1] = '\0';
83   } else if (size < 56) {
84     dest[0] = 0x80 + size;
85     dest[1] = '\0';
86   } else {
87     strcpy(dest, encode_long_object(size, 0xb7));
88   }
89   strcat(dest, src);
90 }
91
92 char *rlp_encode_list(size_t length) {
93   char *buffer;
94   if (length < 56) {
95     buffer = malloc(sizeof(char) * 2);
96     buffer[0] = 0xc0 + length;
97     buffer[1] = '\0';
98     return buffer;
99   } else {
100    return encode_long_object(length, 0xf7);
101  }
102  return NULL;
103 }
```

Listing 57: RLP Implementation

Line 11 Measure the number of bytes a number takes.

Line 25 Pass the number into a log2 function (binary is base 2, this will help us find the number of bits), floor it to round down and add 1 to make up for any truncated bits.

Line 26 Divide the value obtained in line 25 by 8 to convert bits to bytes and find its ceiling since you cannot have a fraction of a bit.

Line 34 This method will be used to encode both lists and strings with a payload greater than 55 bytes. They were grouped together due to their similarities, the only difference being their offset.

Line 36 Find the number of bytes the length occupies.

Line 37-38 Allocate enough memory to represent the encoding (number of bytes taken by length, a byte taken by the offset and another byte taken by the null byte).

Line 40-41 Assign the first byte as the sum of the offset and the number of bytes the length consumes. Assign the last byte as a null byte.

Line 43 The only remaining elements in the buffer array are the ones in the middle. Since integers are read using big endian, the MSB (most significant bit) will be on the left most side. buffer_size - 1 represents the null byte, therefore, the next empty bit will be found in buffer_size - 2 since all the bytes in the middle are empty. The for loop is also conditioned to stop when the number is smaller than or equal to 0, meaning that only the numbers between 0 and buffer_size - 1 will be created.

Line 44 I took a bit fiddling approach in order to break down an int into its respective bytes. Since i is starting from the last empty slot in the buffer, it will contain the last 8 bits (the LSBs), this will be done by truncating the int and taking the last 8 bits. With every iteration, i moves to the next empty slot and j increases by 1, meaning the int will by shifted by 8*j (to fit in the next 8 bits). A char will always take the last 8 bits of an integer, so to create multiple chars out of an integer, we need to shift by 8 to deconstruct the int. Kindly look at figure 7 for a visual representation.

Line 51-56 If the string is less than 56 characters long, return a 2 byte char array with the first byte as the offset 0x80 + string length and the second byte as a null byte.

Line 57-59 Otherwise use long object encoding.

Line 80-82 char array of size 1 is self encoding.

Line 83-85 char array of size greater than 1 and less than 56 has an encoding of 0x80 plus its size.

Line 87 char array of size greater than 55 will just call encode$_{l}ong_{o}bjectwithanoffsetof0xb7$.

Line 89 Concatenate the message with the encoding.

Line 92-103 Identical to lines 73-90 but with different offsets.

Figure 7: Visual representation of byte stuffing algorithm.

sendMessage

```
1  int sendMessage(MsgQs_t *q, void *ID, char *sender, char *subject,
2                  char *content) {
3    // If the provided ID is null, treat it as unspecified and use sendMessageBatch.
4    // otherwise, sendMessage to a single nodeMsg_t (if it exists ofcourse).
5    if (ID == NULL) {
6      sendMessageBatch(q, sender, subject, content);
7      return 1;
8    }
9    return populate_individual_node(q->front, (int)ID, sender, subject, content);
10 }
```

Listing 58: sendMessage method implementation

Line 1 sendMessage requires a message queue, ID to send message to (which can be null to signify that sendMessageBatch will be used instead), sender, subject and content

Line 5-7 If the ID provided is NULL, pass the arguments into the sendMessageBatch method. This will return a success status.

Line 9 Call the recursive method populate_individual_node.

```
1  int populate_individual_node(nodeMsg_t *front, int ID, char *sender,
2                               char *subject, char *content) {
3    // Return an error if the first node is null.
4    if (front == NULL)
5      return -1;
6
7    // If the ID's match, populate the node.
8    if (front->ID == ID) {
9      enqueue_nodeMsg_t(front, sender, subject, content);
10     return 1;
11   }
12
13   /*
14     Check if we aren't at the last node.  If we are, that means
15     that the previous condition which checks for the Identifier
16     failed
17   */
18   if (front->next != NULL)
19     return populate_individual_node(front->next, ID, sender, subject, content);
20
21   // The code can only reach this case if it is the last node in the linkedlist.
22
23   return -1;
24 }
```

Listing 59: populate_individual_node helper function

Line 4-5 Check if the provided node is NULL (meaning that the createQ method hasn't been called yet), if it is NULL, return a failure status.

Line 8 Check if the node's ID matches the specified identifier.

Line 9 Enqueue the node by calling enqueue_nodeMsg_t.

Line 18 Check if the next node isn't NULL (check if there is another node in the linked list).

Line 19 Recursively call populate_individual_node again providing the next node in the linked list (and bubble the result up the recursion stack).

```
1  int enqueue_nodeMsg_t(nodeMsg_t *q, char *sender, char *subject,
2                        char *content) {
3    Item *tmp;
4    tmp = create_item(sender, subject, content);
5    // is the node not empty?
6    if (q->size != 0) {
7      q->rear->next = tmp;
8      q->rear = tmp;
9    } else {
10     q->front = q->rear = tmp;
11   }
12   q->size++;
13 }
```

Listing 60: enqueue_nodeMsg_t helper function

Line 3-4 Create an Item (content of nodeMsg_t).

Line 6 Is the queue not empty?

Line 7 Set the current rear's next value as the newly created Item.

Line 8 Set the rear as the newly created Item.

Line 10 If the queue is empty, set the front and the rear as the newly created Item.

```
1  Item *create_item(char *sender, char *subject, char *content) {
2    Item *temp = malloc(sizeof(Item));
3    temp->sender = malloc(sizeof(char) * rlp_strlen(sender));
4    temp->message = malloc(sizeof(Message));
5    temp->message->subject = malloc(sizeof(char) * rlp_strlen(subject));
6    temp->message->content = malloc(sizeof(char) * rlp_strlen(content));
7    // Plus 4 since time_t is an unsigned int which is 4 bytes.
8    temp->bytes = strlen(sender) + strlen(subject) + strlen(content) + 4;
9    temp->message->bytes = strlen(subject) + strlen(content);
10
11   time(&temp->expiry);
12   rlp_encode_str(temp->sender, sender);
13   rlp_encode_str(temp->message->subject, subject);
14   rlp_encode_str(temp->message->content, content);
15   temp->next = NULL;
16
17   return temp;
18 }
```

Listing 61: create_item helper function

Line 2 Allocate enough memory to hold the members of the Item struct.

Line 3 Allocate enough memory to hold the message and it's RLP encoding.

Line 4 Allocate enough memory to hold the members of the Message struct.

Line 5-6 Allocate enough memory to contain the rlp encoded subject and content.

Line 8-9 Since we are placing rlp encoded strings into the item from the get go, we need to find the number of bytes which the variable holds without the rlp encoding. By adding the number of characters written and assigning them to a variable, we can keep track of this. This will also help with encoding down the line

Line 11 Place the current time in the Item.

Line 12-14 Encode the sender, message and content in RLP by calling the rlp_encode_str method which creates the RLP encoding and concatenates it with the actual char* provided.

Line 15 Set the next Item in the linked list as null, this will be necessary for the recursive methods.

Line 17 Return a pointer to the created item.

sendMessageBatch

```
1 // returns an error if queue id doesn't exist
2 void sendMessageBatch(MsgQs_t *q, char *sender, char *subject, char *content) {
3   batch_populate_node(q->front, sender, subject, content);
4 }
```

Listing 62: sendMessageBatch method implementation

```
1 void batch_populate_node(nodeMsg_t *front, char *sender, char *subject,
2                          char *content) {
3   if (front == NULL)
4     return;
5   enqueue_nodeMsg_t(front, sender, subject, content);
6   if (front->next != NULL)
7     batch_populate_node(front->next, sender, subject, content);
8   return;
9 }
```

Listing 63: batchPopulateNode helper function

Line 3-4 If the node is null, bubble up the recursion stack.

Line 5 Enqueue the node.

Line 6 If the next node isn't null, recursively call batch_populate_node providing the node in the next pointer.

Line 7 Go up the stack.

receiveMessages

```
1 int receiveMessages(MsgQs_t *q, int identifier, size_t num_of_messages) {
2   nodeMsg_t *front = q->front;
3   /*
4         Find the node which the ID belongs to.  Make sure that the
5         num_of_messages doesn't skip the max number of messages.  Once
```

```
6         found, display its contents, free it and do a normal dequeue
7         procedure.
8
9   */
10
11  for (size_t i = 0; i < q->size; ++i) {
12    if (front->ID == identifier) {
13      num_of_messages =
14          num_of_messages > front->size ? front->size : num_of_messages;
15      Item *front_item;
16      for (size_t j = 0; j < num_of_messages; ++j) {
17        // Link the front to the next item.
18        front_item = front->front;
19        printItem(front_item);
20        free_individual_item(front_item);
21        front->front = front_item->next;
22        front->size--;
23      }
24      return 1;
25    } else {
26      front = front->next;
27    }
28  }
29  return -1;
30 }
```

Listing 64: receiveMessages method implementation

Line 11-12 Go through every single node in the message queue until a node with a matching ID is found.

Line 13-14 Wrap num_of_messages if it exceeds the max number of messages available.

Line 16-21 Display the message, relinquish it of its resources, change the front of the queue to the next message to keep a functional linked list and reduce the size..

Line 25-26 Otherwise, set the temporary variable front as the next node in the message queue.

purgeQs

If the provided ID is null, purge all of the nodes, otherwise, check if the front of the message queue matches the identifier specified, if it doesn't, call empty_q, which will find the node and relinquish it of its resources.

```
1  int purgeQs(MsgQs_t *q, void *identifier) {
2    if (identifier == NULL) {
3      empty_all_qs(q->front);
4      return 1;
5    }
6
7    if (q->front->ID == (int)identifier) {
8      nodeMsg_t *temp = q->front;
9      free_item(temp->front);
10     q->front = temp->next;
11     return 1;
12   }
13   return empty_q(q->front, (int)identifier);
14 }
```

Listing 65: purgeQs method implementation

Line 7-11 If the node at the front of the message queue matches the specified ID, it will be freed and the front of the message queue will be replaced with the next node in the queue.

```
1  // Takes the front of a MsgQs_t.
2  void empty_all_qs(nodeMsg_t *front) {
3
4    free_item(front->front);
5    front->size = 0;
6    if (front->next != NULL) {
7      return empty_all_qs(front->next);
8    }
9    return;
10 }
```

Listing 66: empty_all_qs helper function

Line 4-5 Free all of the items in the node and set its size to 0.

Line 6-7 If the next node isn't NULL, recursively call empty_all_qs with the next node in the linked list.

```
1  // Empties a queue with a specific ID.
2  int empty_q(nodeMsg_t *front, int identifier) {
3
4    if (front->next == NULL)
5      return -1;
6
7    if (front->next->ID == identifier) {
8      nodeMsg_t *next = front->next;
9      next->size = 0;
10     // Free all the Qs items
11     free_item(next->front);
12     return 1;
13   }
14
15   /*
16     If the next node isn't null or an identifier, go to the next
17     node to repeat the process. This way, the result will bubble
18     back up the stack.
19   */
20   return empty_q(front->next, identifier);
21 }
```

Listing 67: empty_q helper function

Line 4-5 If the next node is NULL, return a failure status.

Line 7-11 If the next node's matches, set its size to zero and relinquish its resources.

Line 20 If the next node isn't NULL or doesn't match, recursively call empty_q with the next node in the linked list. This will also bubble the returned status up the recursion stack.

persistQ

```
int persistQ(MsgQs_t *MsgQ, int identifier) {
  /*
    The strings are already RLP encoded. The only thing that needs to
    be taken care of is the encoding of the expiry.

    Create a function which takes care of the list representation
    of nodeMsg_t (just get the payload size ie. go through the
    Item queue and get their sizes); Every time a function
    finishes, its result is directly written into the file. Once
    the list representation has been written, the sender and
    expiry are written, followed by the list representation of the
    message and its dumped text.

    This whole algorithm will be encapsulated into a function
  */

  // Go through all the nodes
  /*
    This is necessary as opposed to taking the size of the stored
    chars since the stored chars are already encoded.
  */

  nodeMsg_t *q;
  nodeMsg_t *tmp = MsgQ->front;
  // Find the node with the matching ID
  if (contains_id(MsgQ->front, identifier)) {
    for (size_t i = 0; i < MsgQ->size; ++i) {
      if (tmp->ID == identifier) {
        q = tmp;
      } else {
        tmp = tmp->next;
      }
    }
  } else {
    return -1;
  }
```

```
37
38    q->bytes = update_node_bytes(q->front);
39    // https://stackoverflow.com/questions/14564813/how-to-convert-integer-to-character-
40    // 10 max decimal digits with 32 bits (4 byte int)
41    // 4 extra to hold ".dat"
42    // 1 extra for the null byte.
43    FILE *file;
44    char file_name[15];
45    sprintf(file_name, "%d.dat", q->ID);
46    if ((file = fopen(file_name, "w")) == NULL) {
47      printf("ERROR OPENING %s", file_name);
48      fclose(file);
49      return -1;
50    }
51    char *list_encoding = rlp_encode_list(q->bytes);
52    fwrite(rlp_encode_list(q->bytes), sizeof(char), strlen(list_encoding), file);
53    Item *front = q->front;
54    // This loop writes out everything into a file.
55    for (size_t i = 0; i < q->size; ++i) {
56      char *item_encoding = rlp_encode_list(front->bytes);
57      fwrite(item_encoding, sizeof(char), strlen(item_encoding), file);
58      fwrite(front->sender, sizeof(char), strlen(front->sender), file);
59      char *expiry_encoding = str_encode_length(byte_len(front->expiry));
60      fwrite(expiry_encoding, sizeof(char), strlen(expiry_encoding), file);
61      fwrite(&front->expiry, sizeof(long int), 1, file);
62      char *message_encoding = rlp_encode_list(front->message->bytes);
63      fwrite(message_encoding, sizeof(char), strlen(message_encoding), file);
64      char *subject = front->message->subject;
65      char *content = front->message->content;
66      fwrite(subject, sizeof(char), strlen(subject), file);
67      fwrite(content, sizeof(char), strlen(content), file);
68      front = front->next;
69    }
70    fclose(file);
71    return 1;
72  }
```

Listing 68: persistQs method implementation

Line 23-36 Find the node which matches the ID.

Line 38 Simply sum the bytes member of every Item and assign it to the node's bytes member.

Line 43-50 Concatenate the ID and the ".dat" extension and check if a file with that name already exists, if so, return a failure status.

Line 51-52 Create the list encoding for a node and write it into the file.

Line 55-69 Encode the item struct payload and write it into the file, write the sender (recall that everything is RLP encoded on creation), time of expiry and its encoding (has to be treated as a long object since an integer is 4 bytes), write the list encoding for the message structure and write the subject and content into the file. Lastly, move onto the next Item in the linked list.

Line 70 Close the stream.

restoreQ

The strategy behind decoding the RLP encoded text consists of skipping every list encoding found and associating data with the order it was written, once one analyzes the written data, it is easy to see that there is a pattern which can be followed. If one skips the first list encoding, decodes the string encoding for the sender, decodes the encoding for the expiry, skips the next list encoding, decodes the subject and decodes the content and goes to the second step until all the data has been read, all the data will be decoded. The decoded data is passed into the sendMessage method.

```
1  // Precondition: MsgQs_t is created from before hand.
2  int restoreQ(MsgQs_t *q, int identifier) {
3    FILE *file;
4    char file_name[15];
5    sprintf(file_name, "%d.dat", identifier);
6    if ((file = fopen(file_name, "r")) == NULL) {
7      printf("ERROR OPENING %s", file_name);
8      fclose(file);
9      return -1;
10   }
11   skip_list(file);
12   char *sender;
13   long int expiry;
14   char *subject;
15   char *content;
16
17   // Decoding this takes a specific pattern.
18   // Analyze a .dat file which is produced by the code
19   // to see it.
20   while (!feof(file)) {
21     skip_list(file);
22     if (!feof(file)) {
23       sender = decode_string(file);
24       expiry = decode_expiry(file);
25       skip_list(file);
26       subject = decode_string(file);
27       content = decode_string(file);
28       sendMessage(q, (void *)identifier, sender, subject, content);
29     }
30   }
31 }
```

Listing 69: restoreQ method implementation

Line 12 Skip the node list encoding by moving the file pointer over it.

Line 23 Check if a file reading error has been thrown after the previous skip_list, if an error is thrown, this indicates that the EOF has been reached.

```
1  /*
2    Skip the list encoding of the file by moving the file pointer
3    accordingly.
4  */
5
6  void skip_list(FILE *file) {
7    unsigned char ch = fgetc(file);
8    if (ch >= 0xc0 && ch <= 0xf7) {
9      /*
10       Self encoding list
11       Since the encoding is only 1 byte long the next fgetc will get a valid
12       char
13     */
14     return;
15   } else if (ch >= 0xf8 && ch <= 0xff) {
16     size_t byte_length = ch - 0xf7;
17     while (byte_length > 0) {
18       // Skip the byte characters representing the list length
19       fgetc(file);
20       byte_length--;
21     }
22   }
23 }
24
25 long int decode_expiry(FILE *file) {
26   unsigned char ch = fgetc(file);
27   long int temp;
28   fread(&temp, sizeof(long int), 1, file);
```

```
29   return temp;
30 }
31
32
33 /*
34   Decode the string according to the character obtained.
35   The algorithm followed can be found in the Ethereum RLP wiki.
36 */
37 char *decode_string(FILE *file) {
38   unsigned char ch = fgetc(file);
39   char *buffer;
40   if (ch < 0x80) {
41     buffer = malloc(sizeof(char) * 2);
42     buffer[0] = ch;
43     buffer[1] = '\0';
44     return buffer;
45   } else if (ch >= 0x80 && ch < 0xb8) {
46     // 0x7f instead of 0x80 due to the fact that if we get 0x80 and minus it by
47     // 0x80 we will get a length of 0
48     size_t length = ch - 0x80;
49     buffer = malloc(sizeof(char) * length + 1);
50     fread(buffer, sizeof(char), length, file);
51     buffer[length] = '\0';
52     return buffer;
53   } else if (ch >= 0xb8 && ch < 0xbf) {
54     size_t byte_length = ch - 0xb7;
55     size_t size = 0;
56     for (size_t i = 0; i < byte_length; ++i) {
57       size |= fgetc(file) << 8 * (byte_length - 1 - i);
58     }
59     buffer = malloc(sizeof(char) * size);
60     fread(buffer, sizeof(char), size, file);
61     buffer[size] = '\0';
62     return buffer;
63   }
64 }
```

Listing 70: Helper functions used to decode from the previously RLP encoded file.

Line 7 Get the list encoding.

Line 8-14 If between 0xc0 and 0xf7, the list is self encoding.

Line 15-21 If the payload is greater than 55, find the byte length (which will determine to the number of characters to skip), a call fgetc (which also moves the file pointer by a character) until we've reached the last byte. Don't forget that fgetc will always return the value of the next char in the pointer.

Line 25-30 Get the first character and read the next 4 bytes as a long int.

Line 37-38 Get the offset.

Line 40-45 If the offset is less than 0x80, it is self encoding, return a char* containing the character and a null byte.

Line 45-52 If the offset is between 0x80 and 0xb8 (length less than 56 bytes), then call fread on the length of the string (fread behaves like fgetc ie. the file pointer will still be moved).

Line 53-57 If the length is greater than 55 bytes, get the byte length, shift each byte accordingly and OR it with the size, that way, the bytes which have been deconstructed are stuffed back into a 4 byte variable.

Line 59-61 Read the remaining text and return it.

Limitations and known bugs

Due to how the standard libraries are written, more specifically, strlen, returns a value of type size_t (which is an unsigned int). Using strlen, we are limiting the number of bytes which a length can be represented by to 4 bytes instead of the 7 bytes.
Another limitation and known bug due to the use of strcpy and strcat is that if a string or list with a payload greater than 55 which is a multiple of 256 will lead to faulty decoding since all multiples of 256 have their last 8 bits as 0x00 (which is also a null byte). Meaning that strcpy and strcat will overwrite this null byte changing the length. A solution to this would be to painstakingly use memcpy instead of the standard str methods.

Task 2c

One may find a full test driver for task 2c linked to the ADT shared library in a file called task2_main, under the task2 directory.