Part A:

```
SQ * sq_create()
/**
* Function: sq_create()
* Description: creates and initializes an empty service queue.
*     It is returned as an SQ pointer.
*/
```

Runtime analysis: O(1)
lst_create is a constant time function since it just initializes a list. Sq_create consists of 2 lst_create calls resulting in a 2O(1). Dropping the constant results in O(1) runtime.

```
void sq_free(SQ *q)
/**
* RUNTIME REQUIREMENT:  O(N_t)
*
*/
```

Runtime analysis: O(N_t)
O(N_t) will be achieved in freeing the actual amount of nodes in the buzzer_bucket list. Then another O(N_t) will be required to required to free the same amount from the_queue list. This will result in O(N_t)+O(N_t) or 2O(N_t) but dropping the constant O(N_t).

```
void sq_display(SQ *q)
/**
* Function: sq_display()
* Description:  see sq.h
*
* REQUIRED RUNTIME:  O(N)  (N is the current queue length).
*/
```

Runtime analysis: O(N)(N is the current queue length)
This function is reliant on the lst_print function in llist.c file. If we look at this function it is a while loop that starts at the head of the list and iterates through the end making it a O(N).

```
int  sq_length(SQ *q)
/**
* Function: sq_length()
* Description:  see sq.h
*
* REQUIRED RUNTIME:  O(1)
*/
```

Runtime analysis O(N):

This function is reliant on the lst_length function in llist.c. If we look at the llist.c length function, we can see that it is similar to the print function in that it starts at the head and transverses the entire length of the list until it reaches the end incrementing a counter variable as it goes. Thus making it O(N). (where N is the size of the queue)

```
int  sq_give_buzzer(SQ *q)

/**
* Function: sq_give_buzzer()
*
* REQUIRED RUNTIME:  O(1)
*/
```

Runtime analysis O(N):
The only limiting factor in this function is the implementation of sq_length. And as discussed previously, sq_length has a O(N) thus sq_give_buzzer() would also have a O(N)

```
int sq_seat(SQ *q)
/**
* function: sq_seat()
*
* REQUIRED RUNTIME:  O(1)
*/
```

Runtime analysis O(1):
There is no limiting action within the function. All the functions it implements within itself have a constant runtime, so a combination of these would result in a constant runtime as well.

```
int sq_kick_out(SQ *q, int buzzer)
/**
* function: sq_kick_out()
*
* REQUIRED RUNTIME:  O(1)
*/
```

Runtime analysis O(N):
The limiting action in this function is the call to lst_remove_first. The O() of this function would be O(N) under the case that the last element was the first instance of the search variable. This is because it has to transverse the entire length of the list in a while loop. This would therefore make sq_kick_out O(N).

```
int sq_take_bribe(SQ *q, int buzzer)

/**
* function:  sq_take_bribe()
*
* REQUIRED RUNTIME:  O(1)
*/
```

Runtime analysis O(N):
The limiting action in this function is the call to lst_remove_first. The O() of this function would be O(N) under the case that the last element was the first instance of the search variable. This is because it has to transverse the entire length of the list in a while loop. This would therefore make sq_kick_out O(N).

Part B:

I did not have time to fully implement what I planned to for the the project. But the idea was to have a two doubly linked lists, one for queue and one for buzzers, that also had a struct for the size within their definitions. This would allow so all of the functions with lst_length (as discussed above) to be in constant runtime. This would be because as you push and pop the functions you would also increment or decrement the size integer of the doubly linked list. The part that I didn’t fully get to implement is the idea of an array of node pointers. This array would have node pointers that accessed every element of the queue and buzzers. The purpose of this would be to directly access nodes that you wish to push or pop. This would make the runtime of sq_kick_out and sq_take_bribe able to achieve a constant runtime as there would be no need to go through the list, you could just pop and push directly. All runtimes should be achieved except for the last two.