(Exercises 1-3 are trivial).

Suppose the necessary assumptions for the following exercises:

Exercise 4

Solution

```cpp
#include <queue>
#include <stack>

std::queue<char> reverse_queue_using_stack(std::queue<char> input_queue) {
    std::stack<char> temporary_stack;

    while (!input_queue.empty()) {
        char current_element = input_queue.front();
        input_queue.pop();
        temporary_stack.push(current_element);
    }

    std::queue<char> reversed_queue;
    while (!temporary_stack.empty()) {
        char top_element = temporary_stack.top();
        temporary_stack.pop();
        reversed_queue.push(top_element);
    }

    return reversed_queue;
}
```

Exercise 5

Solution

We can make 2 different queuess growing in opposite directions (they'll colide at the middle, this may or may not be relevant to the problem):

```cpp
struct TwoStacksInOneArray {
  int* shared_array;
  int array_capacity;
  int top_index_stack_1;
  int top_index_stack_2;

  TwoStacksInOneArray(int total_capacity) {
      array_capacity = total_capacity;
      shared_array = new int[array_capacity];
      top_index_stack_1 = -1;
      top_index_stack_2 = array_capacity;
  }
```

```
13
14    void push_to_stack_1(int value_to_push) {
15        if (top_index_stack_1 < top_index_stack_2 - 1) {
16          shared_array[++top_index_stack_1] = value_to_push;
17        }
18    }
19
20    void push_to_stack_2(int value_to_push) {
21        if (top_index_stack_1 < top_index_stack_2 - 1) {
22            shared_array[--top_index_stack_2] = value_to_push;
23        }
24    }
25
26    int pop_from_stack_1() {
27        return (top_index_stack_1 >= 0) ? shared_array[top_index_stack_1--] : -1;
28    }
29
30    int pop_from_stack_2() {
31        return (top_index_stack_2 < array_capacity) ?
          shared_array[top_index_stack_2++] : -1;
32    }
33 };
```

Exercise 6

Solution

```
1 int circular_sequential_search(int* circular_list, int list_length, int
  target_value, int start_index) {                                          C++
2     for (int offset = 0; offset < list_length; offset++) {
3         int current_index = (start_index + offset) % list_length;
4         if (circular_list[current_index] == target_value) {
5             return current_index;
6         }
7     }
8     return -1;
9 }
```

Exercise 7

Solution

```
1  struct DequeWithFixedArray {                                             C++
2      int* deque_array;
3      int front_index;
4      int rear_index;
5      int current_size;
6      int array_capacity;
```

```
7
8       DequeWithFixedArray(int maximum_capacity) {
9           array_capacity = maximum_capacity;
10          deque_array = new int[array_capacity];
11          front_index = 0;
12          rear_index = 0;
13          current_size = 0;
14      }
15
16      void insert_at_front(int value_to_insert) {
17          if (current_size == array_capacity) return;
18          front_index = (front_index - 1 + array_capacity) % array_capacity;
19          deque_array[front_index] = value_to_insert;
20          current_size++;
21      }
22
23      void insert_at_rear(int value_to_insert) {
24          if (current_size == array_capacity) return;
25          deque_array[rear_index] = value_to_insert;
26          rear_index = (rear_index + 1) % array_capacity;
27          current_size++;
28      }
29
30      void remove_from_front() {
31          if (current_size == 0) return;
32          front_index = (front_index + 1) % array_capacity;
33          current_size--;
34      }
35
36      void remove_from_rear() {
37          if (current_size == 0) return;
38          rear_index = (rear_index - 1 + array_capacity) % array_capacity;
39          current_size--;
40      }
41  };
```

Exercise 8

Solution

```
1   void enqueue_value_into_queue(int value_to_enqueue) {                C++
2       input_stack_for_enqueue_operations.push(value_to_enqueue); // O(1)
3   }
4
5   void transfer_elements_from_input_to_output_stack() {
6       while (!input_stack_for_enqueue_operations.empty()) {
```

```
            int top_value_from_input_stack =
            input_stack_for_enqueue_operations.top();
            input_stack_for_enqueue_operations.pop();
            output_stack_for_dequeue_operations.push(top_value_from_input_stack);
        }
    }

    int dequeue_value_from_queue() {
        if (output_stack_for_dequeue_operations.empty()) {
            transfer_elements_from_input_to_output_stack();
        }

        if (output_stack_for_dequeue_operations.empty()) {
            return -1; // Queue is empty
        }

        int value_to_return = output_stack_for_dequeue_operations.top();
        output_stack_for_dequeue_operations.pop();
        return value_to_return;
    }
```

Exercise 9

Solution

a)

$O(n)$:

```
int get_minimum_value_linear(int* stack_array, int current_stack_size) {
    if (current_stack_size == 0) return -1;
    int minimum_value = stack_array[0];
    for (int i = 1; i < current_stack_size; i++) {
        if (stack_array[i] < minimum_value) {
            minimum_value = stack_array[i];
        }
    }
    return minimum_value;
}
```

b)

$O(1)$:

```
struct StackWithMinimum {
    int* element_array;
    int current_size;
    int maximum_capacity;
    std::stack<int> minimum_tracking_stack;
};
```

```
7
8   void push_with_min_tracking(StackWithMinimum* stack, int value_to_push) {
9       if (stack->current_size < stack->maximum_capacity) {
10          stack->element_array[stack->current_size++] = value_to_push;
11          if (stack->minimum_tracking_stack.empty() || value_to_push <= stack-
            >minimum_tracking_stack.top()) {
12              stack->minimum_tracking_stack.push(value_to_push);
13          }
14      }
15  }
16
17  int pop_with_min_tracking(StackWithMinimum* stack) {
18      if (stack->current_size == 0) return -1;
19      int popped_value = stack->element_array[--stack->current_size];
20      if (!stack->minimum_tracking_stack.empty() && popped_value == stack-
        >minimum_tracking_stack.top()) {
21          stack->minimum_tracking_stack.pop();
22      }
23      return popped_value;
24  }
25
26  int get_minimum_value_constant_time(StackWithMinimum* stack) {
27      return (!stack->minimum_tracking_stack.empty()) ? stack-
        >minimum_tracking_stack.top() : -1;
28  }
```

Exercise 10

Solution

a)

$O(n)$:

```
1   int get_minimum_value_in_queue_linear(std::queue<int> input_queue) {
2       if (input_queue.empty()) return -1;
3       int minimum_value = input_queue.front();
4       while (!input_queue.empty()) {
5           if (input_queue.front() < minimum_value) {
6               minimum_value = input_queue.front();
7           }
8           input_queue.pop();
9       }
10      return minimum_value;
11  }
```

C++

b)

Optimized if $n \in [1, 10]$:

```cpp
int get_minimum_value_with_fixed_range(std::queue<int> input_queue) {
    int frequency_counter[11] = {0};
    while (!input_queue.empty()) {
        int current_value = input_queue.front();
        input_queue.pop();
        frequency_counter[current_value]++;
    }

    for (int value = 1; value <= 10; value++) {
        if (frequency_counter[value] > 0) {
            return value;
        }
    }
    return -1;
}
```