Bradley Manzo
April 8th, 2022
Prof. Porquet
ECS 36C A03 SQ

ECS 36C Programming Assignment 2 Report

In order to implement the desired behavior of the air luggage management, I constructed a stack of queue objects holding integers. I chose this implementation after reenacting the provided example on paper. I instantiated a single temporary queue object to fill with bags and I iterated through the input file pushing each new bag name onto the queue until the queue bound was reached. Then the temporary queue was pushed onto the stack by reference and said queue was consequently popped clean in order to gather the next set of bags. After the file is done being read, I push the remaining queue that did not reach the queue bound onto the stack. I then printed the contents of the stack in a similar fashion, by instantiating a new temporary queue to hold the Top() of the stack. After iterating through each item of the queue, printing the Front() and popping the value off of the queue, I popped the stack and retrieved the new Top(). This behavior repeats until the stack is empty using a stack function I wrote which wraps .empty().

For the second program of this assignment, I elected to adapt the structure of the Professor's list_dynamic_array.h implementation, creating a smart pointer to a templated array as the basis of my deque. I then chose to initialize my front and back as integers to index said array. Upon instantiation, both are set to -1, but after receiving the first item, both front and back are set to 0. To push to the deque, the respective index is incremented and the desired value is inserted at the new position. To pop from the deque, all that is required is for the front or back to move out of bounds and ignore the item. When an item is outside of the front and back indexes, although it is within memory, it lies outside of the deque, and cannot be accessed. This was achieved by overloading the [] operator to refer to the position in the deque rather than the position in capacity. Because the position always increases from left to right, we can use the modulo operator to return the remainder of the new position if it exceeds the capacity Some similarities in traversal I recognized were that PushBack() and PopFront() move right, while PushFront() and PopBack() move left. To move right, the index is incremented, unless at the rightmost position where the next position is set to the leftmost, and vice versa for moving to the left. Upon each push and pop, the current size and both indices are dually updated and maintained as private variables.

Resizing occurs either automatically when the deque is full, or when the deque is a quarter of the allocated capacity. Just as in the dynamic array, the contents of the deque are copied using std::move into a new array of the desired size, and thanks to the smart pointer nature of the deque, the ownership of the new deque is swapped with the original. When resizing, the most critical condition is whether or not said deque is contiguous. If back > front, this means that we need only to move one section of code, however if front > back, this means that the deque is split in two as it wraps around the end of capacity. This requires the movement of both sections of code into the new deque, left section after the right. ShrinkToFit() takes advantage of Resize by resizing to the current size. Front() and Back() merely return what value is placed at each front and back, and clear() simply sets front and back to -1, updating the current size to zero. Size() returns the current size, while Empty() returns true when current size is zero. When designing my gtest code, I did my best account for the possible scenarios that I recognized requiring multiple pushes or pops, which resulted in both contiguous and noncontiguous deques, seeing how this affected the order of the deque and also the behavior of Resize(). After validating these behaviors and consequent functions, I am confident in the performance of my deque ADT.