Part B:

Jacob Holloway
827294826

```
*/
void PromotedCarModelStack::push(string model, int price) {


car = PromotedModel(model, price);
carModels.push_back(car);
HighLowCarPrices.push_back(car);

cout << "High Car Price: " << HighLowCarPrices.back().getPromotedPrice() << endl;



    if(price > HighLowCarPrices.front().getPromotedPrice()){

       HighLowCarPrices.push_back(highCar);


       highCar = HighLowCarPrices.back();

       cout << highCar.getModel() <<  " " << highCar.getPromotedPrice() << endl;

    }
    else{

       lowCar = carModels.front();
    }

    if(highCar.getPromotedPrice() < price){

       highCar = car;
    }


if(carModels.empty() || HighLowCarPrices.empty()){

   throw logic_error("Promoted car model stack is empty");
  }

}
```

The best case for the push is when the both stacks are empty because it does not need to look through anything at all and case easily traverse the code and then push the models of cars if needed
1 + 1 + 1 = O(1)

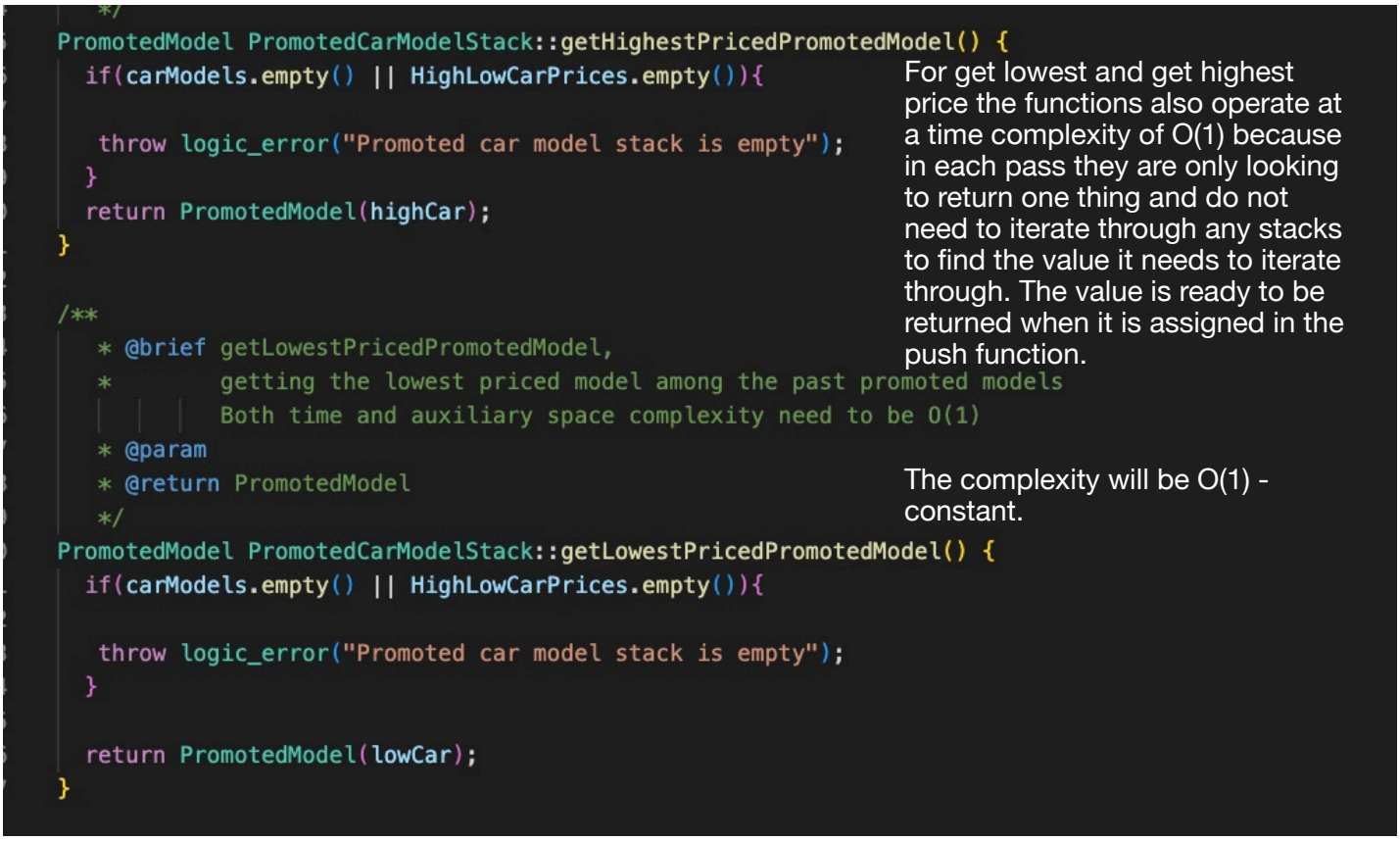
For the worst case it would end up at O(1) because the time complexity is always constant so regardless of how many implementations are happening with the code the push function and all the other functions will run at O(1) complexity.

T(n) = 1 + 1 + 1 + 1 + 1

O(1) = 1

```
PromotedModel PromotedCarModelStack::pop() {
  if(carModels.empty() || HighLowCarPrices.empty()){

   throw logic_error("Promoted car model stack is empty");
  }
  carModels.pop_back();
  HighLowCarPrices.pop_back();
  //lowCar = HighLowCarPrices.back();

  car = HighLowCarPrices.back();


  return PromotedModel(car);
}

/**
   * @brief peek operation, peeking the latest promoted model at the top of the stack (without popping)
          Both time and auxiliary space complexity need to be O(1)
   * @param
   * @return PromotedModel
   */
PromotedModel PromotedCarModelStack::peek() {
  if(carModels.empty() || HighLowCarPrices.empty()){

   throw logic_error("Promoted car model stack is empty");
  }
  return PromotedModel(carModels.back());
}
```

When the model is popped it will pull from the back of the stack and continue moving in the code. In the worst case the stack is empty which results in the if statement being thrown but will still operate in a constant time complexity of O(1) regardless of how many pop operations are done for each pass through.



```
 */
PromotedModel PromotedCarModelStack::getHighestPricedPromotedModel() {
  if(carModels.empty() || HighLowCarPrices.empty()){

   throw logic_error("Promoted car model stack is empty");
  }
  return PromotedModel(highCar);
}

/**
   * @brief getLowestPricedPromotedModel,
   *        getting the lowest priced model among the past promoted models
   *        Both time and auxiliary space complexity need to be O(1)
   * @param
   * @return PromotedModel
   */
PromotedModel PromotedCarModelStack::getLowestPricedPromotedModel() {
  if(carModels.empty() || HighLowCarPrices.empty()){

   throw logic_error("Promoted car model stack is empty");
  }

  return PromotedModel(lowCar);
}
```

For get lowest and get highest price the functions also operate at a time complexity of O(1) because in each pass they are only looking to return one thing and do not need to iterate through any stacks to find the value it needs to iterate through. The value is ready to be returned when it is assigned in the push function.

The complexity will be O(1) - constant.

Time Complexity explanation:

For my code, the program is running at a consistent 0(1) because no matter the size of the stack at each pass through it will grab new variables in the high and low cars as well as push back the lists. Given that peek and pop are looking at the back of the lists it will stay at O(1) because it does not have have to search through the list to get the result it is looking for. It stays constant which is why it carries the 0(1) time space.

1 + 1 + 1 + 1 + …. = O(1)

During each pass the stack continues to grow but because we pull so the complexity does not change or become O(N).

Time complexity would be T(1) = 1.

Space Complexity:

For this program, the code operates at the O(1) complexity. Space complexity looks into memory usage throughout the run time of the code. During each pass the complexity stays consistent.

So with all the space complexity the program still functions at

O(1) = 1 + 1 + 1 + 1;

Auxiliary Space:

Auxiliary space is referring to all memory that is not related to the program running.

My auxiliary space I believe is 5 with two vectors instantiated in promoted model.h and three objects of promoted model instantiated in promoted model.h

S(N) = 5

```
class PromotedCarModelStack {

private:
 vector<PromotedModel> carModels;
 vector<PromotedModel> HighLowCarPrices;

 PromotedModel car;
 PromotedModel lowCar;
 PromotedModel highCar;
```