Fast and Famished: Efficient Food Ordering Queue

Sofia Lombardo, Bella Matuch, Santiago Orozco, Hugh Smith
GitHub [github.com/bmatuch/Data-Structures-Final-Project]

Introduction

In the past ten years, apps like GrubHub and DoorDash have jumped to relevancy as people order food online now more than ever before. In niche areas, such as the University of Notre Dame's campus, students use GrubHub to order food for pickup. Due to the simplicity and efficiency of ordering via this app, restaurants are often overwhelmed with mobile orders and struggle to keep up in the kitchen. Students, in turn, suffer as they endure excessive wait times. With Notre Dame students constantly shuffling between classes, study spaces, and extracurriculars, there is hardly enough time to budget for meals, making long wait times impractical. To combat this issue, the goal of our project is to develop a system for efficiently queuing restaurant orders based on the estimated time of arrival (ETA) of the customer.

Algorithms

Since the primary goal of our project was to develop a system for efficiently queuing restaurant orders based on customers' estimated time of arrival (ETA), our program operates off the functionality of a priority queue. A priority queue is a special type of queue in which each element is associated with a "priority value." This allows us to sort and access elements based on a chosen priority. In simple cases, elements with higher values are considered the highest priority. In other cases, we can assume the element with the lowest value as the highest priority element. Or, we can set priorities according to specific needs. In our case, this priority value is customers' ETA. That way, we can access the next order that should be processed based on popped elements from our priority queue.

Eventually, we compare the effectiveness of priority queues versus queue in reducing wait times for customers. In this section, we will describe the underlying algorithms of both of these data structures, and more:

Queues

Queue is a data structure that functions similarly to the stack. Comparable to the stack, a queue is a linear data structure that stores items in a First In First Out (FIFO) manner. However, Unlike the stack, a queue is open at both of its ends.

A real-world example of a queue can be a single-lane, one-way road, where the car that enters first, exit first.

"Data Structure and Algorithms - Queue." *Tutorialspoint*, https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm. Accessed 26 April 2022.

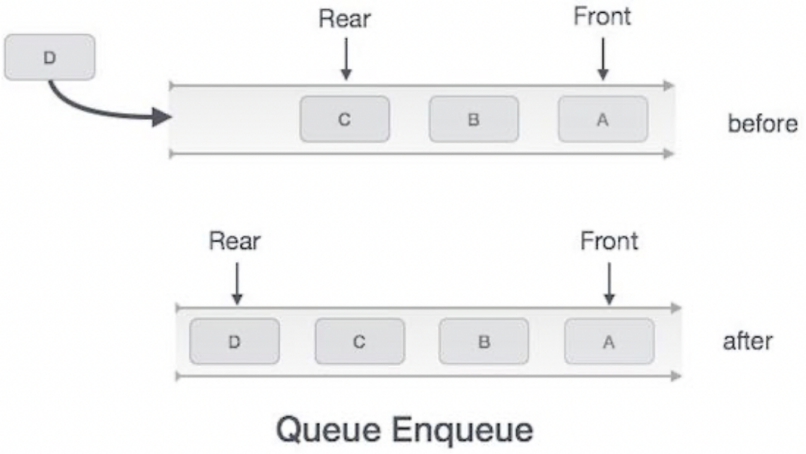
More real-world examples include any line of customers for a resource, where the customer that came first is served first.

Basic operations associated with queues are:

Enqueue: adds an item to the queue – Time complexity: O(1)

Queues maintain two data pointers, front and rear. The following steps should be taken to enqueue (insert) data into a queue

1. Check if the queue is full
2. If the queue is full, produce overflow error and exit
3. If the queue is not full, increment the rear pointer to point at the next empty space
4. Add data element to the queue location, where the rear is pointing
5. Return success

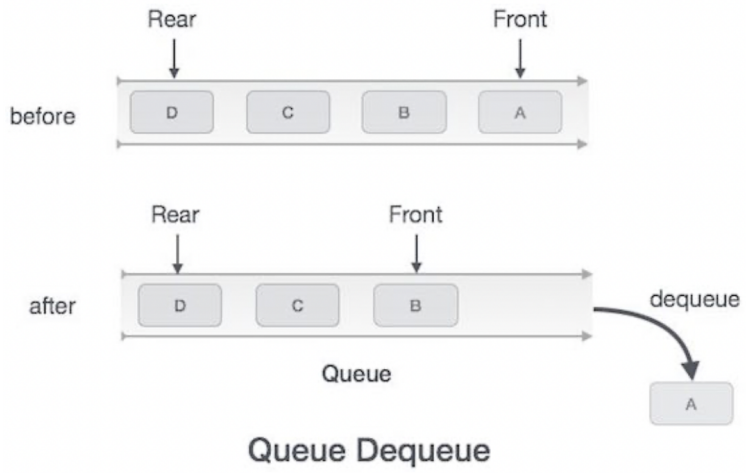


"Data Structure and Algorithms - Queue." *Tutorialspoint*, https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm. Accessed 26 April 2022.

Dequeue: removes an item from the queue – Time complexity: O(1)

Assessing data from the queue is a process of two tasks – access the data where The front pointer is pointing and removing the data after access. The following steps are taken to perform a dequeue operation.

1. Check if the queue is empty
2. If the queue is empty, produce underflow error and exit
3. If the queue is not empty, access the data from where front is pointing
4. Increment the front pointer to point to the next available data element
5. Return success



"Data Structure and Algorithms - Queue." *Tutorialspoint*, https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm. Accessed 26 April 2022.

Queues can be implemented using arrays, linked-lists, pointers, and structures.Particularly in our project, we have implemented the queue using Python lists. A list in Python is a type of dynamic array, but unlike array structures in other programming languages, lists can contain any data type. Since everything in Python is an object, we can simply store pointers to those objects in our dynamic array.

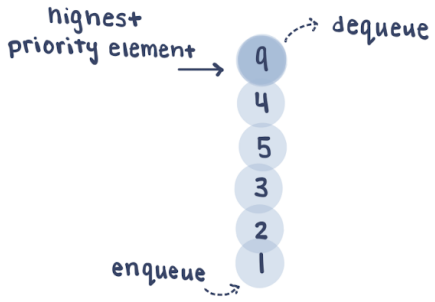
Dynamic arrays build upon static arrays but provide us the flexibility to increase and decrease their size and capacity. In general, this is achieved by allocating and copying over our elements to a new array when our current array becomes full. Typically, we allocate extra memory than we actually need at the moment to reduce the cost of resizing many times.

Priority Queues

A priority queue is a special type of queue in which each element is associated with a "priority value." The priority value of each element determines the order in which elements are removed from the priority queue. Therefore, all elements are either arranged in an ascending or descending order.
In other words, a Priority Queue is a Queue with the following adaptations:

- Every element has a priority associated with it

- An element with high priority is dequeued before an element with a low priority
- If two elements have the same priority, they are served according to their order in the queue



Priority Queues can be implemented using the following data structures: Arrays, Linked Lists, Heap Data Structures, and Binary Search Trees. The time complexity of the each method is shown in a diagram below:

Arrays	enqueue()	dequeue()	peek()
Time Complexity	O(1)	O(n)	O(n)

Linked List	push()	pop()	peek()
Time Complexity	O(n)	O(1)	O(1)

Binary Heap	insert()	remove()	peek()
Time Complexity	O(log n)	O(log n)	O(1)

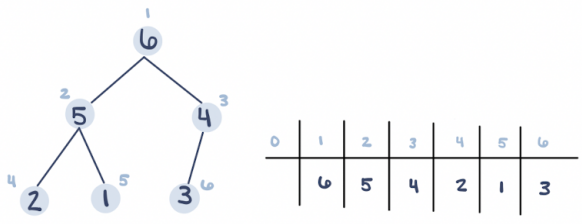
Binary Search Tree	peek()	insert()	delete()
Time Complexity	O(1)	O(log n)	O(log n)

"Priority Queue | Set 1 (Introduction)." *GeeksforGeeks*, 25 January 2022, https://www.geeksforgeeks.org/priority-queue-set-1-introduction/. Accessed 26 April 2022.

In our project we have implemented the Priority Queue structure using a built-in Python class called queue.PriorityQueue which is based on a binary heap, specifically a minimum heap. A

minimum heap is a complete binary tree where no value below a node is smaller than the node, hence we are guaranteed to have the smallest element at the root but the rest of the tree will not necessarily be strictly sorted as in a binary search tree.

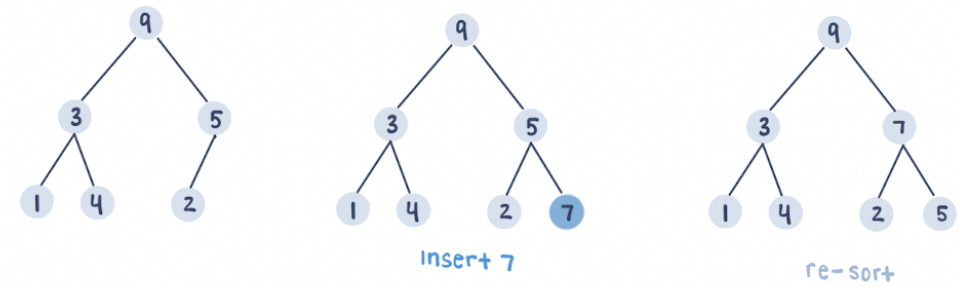
Since the minimum heap is a complete binary tree, this means we can logically represent it in memory by simply using an array thus reducing the usage of pointers and memory. To go from a tree representation to the representation as an array we simply perform a level order traversal.



Additionally, we define insertion and popping methods as follows to guarantee that we preserve the basic properties of the minimum heap.

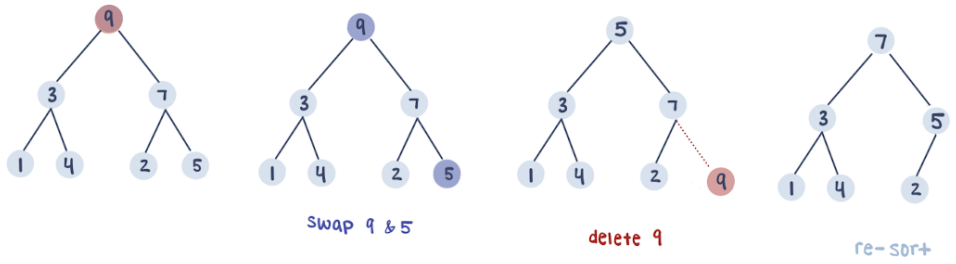
Insertion:

1. At first the element to be inserted is added in the next available position, i.e. we don't insert elements in a new level until the current level is full.
2. Re-sort by comparing the new element with its parent. If its parent is greater than it, swap them and repeat step 2. Otherwise, the new element is in the correct position.



Deletion/Pop:

1. Swap element at the root with the last element in the heap.
2. Delete the last element.
3. Re-sort by comparing the element at the root with its children. If its child is smaller than it, swap them.
4. Continue comparing it with its children and swapping until both of their children are bigger than it or it has no children.



During peaks of high demand, time efficiency is crucial for restaurants; therefore, hash tables come handy to store and retrieve information taking advantage of its O(1) *average* time complexity.

Hash Tables

A *hash table* is a data structure that implements an associative array, a structure that can map keys to values by storing *key-value pairs* which can be readily accessed thanks to the hash function.

The *hash function* can be any function that allows us to map our key with an integer we will use as index in our array, also known as *hash code*. Since our array must have a finite capacity, the hash code must be within the range of spaces available which we call *buckets*; this is typically accomplished by using modular arithmetic. For instance, if our keys x were integers and we had 23 buckets a simple hash function could be f(x) = x mod 23.

Unfortunately, a problem arises when doing this. Consider x = 48 and x = 25; both of them result in f(x) = 2. This is known as a collision. There are several ways of handling collisions such as separate chaining (each bucket contains a linked-list instead of a single pair) and open addressing.

Open addressing only holds one pair in each bucket and instead solves the collision by *probing*. That is, when visiting the location given by the hash code, if a collision arises (i.e. the bucket is occupied by a different pair) we try different buckets following a probing sequence until an available bucket is found. One of the simplest instances of open addressing collision resolution is *linear probing*. In linear probing we simply try the immediate next bucket.

Finally, we have one more issue to consider. You may note that as more and more insertions are performed the likelihood of collision increases. By continuing like these we may end up simply performing a linear search most of the time or we may not be able to even find an empty bucket in the case of open addressing. This would throw away our average time complexity of O(1), and we'd thus lose the benefits of using a hash table. To solve this problem, we implement dynamic resizing.

Dynamic resizing consists in creating a new hash table with more buckets once a given threshold is reached so as to minimize collisions. Since resizing is an expensive operation since it increases the amount of memory being used and every element in the table is re-hashed we would also like

to minimize the frequency of resizing. One way to achieve this is to double the capacity of the table each time and round-up to the nearest prime number; this way the mod function is less likely to collide.

In our project, we implemented hash tables using Python's built-in implementation of hash tables, dictionaries.

Python Dictionary

As said before, dictionaries are Python's implementation of hash tables. The hash function used to evaluate depends on the object being used as a key and each class can define its own __hash__() method. Moreover, collisions are handled via open addressing.

The specific open addressing collision resolution Python perform consists of probing through indices given by the recurrence relation j = ((5*j) + 1 + perturb) mod 2**i, where the variable j represents the index, 2**i the number of buckets, and perturb the hash code.[1]

Regarding dynamic resizing, Python checks for the dictionary size and capacity everytime we add a key, if the dictionary is two-thirds full, it gets resized in the following manner: if a dictionary has 50000 keys or fewer, the new capacity is current_size * 4; otherwise, new capacity is current_size * 2.[2]

The Data

In order to simulate an online ordering system, we created a function to replicate information that a restaurant would receive from customers. The function creates an arbitrary number of orders, containing the customers' names, orders, location (latitude and longitude), and mode of transportation.

Within the function, theoretical customers are assigned a number which represents what number order they are, allowing the difference between the use of a queue and priority queue to be visualized; if they were in a normal queue, the orders would appear in numerical order, but the priority queue rearranges their positioning based on the weights so the numbers are not chronological.

The order, coordinates, and mode of transportation were placed in lists and then accessed using a random number generator to select an index within the list. Coordinates were chosen from various common locations around campus including La Fun, student dorms, and within Duncan Student Center itself. Their coordinates were determined by placing points on Apple Maps at the locations desired and copying them up to five decimal places.

---

1 https://hg.python.org/cpython/file/52f68c95e025/Objects/dictobject.c#l33
2 https://fengsp.github.io/blog/2017/3/python-dictionary/

The function returns a list of Python dictionaries, with the amount of dictionaries in the list determined by the arbitrary number selected prior to its running. A dictionary makes sense for our project because we are able to store each customer's name as the key, and their relevant information as a list of values. Python implements dictionaries using hash tables, which allows for easily accessible keys and values with a constant look-up time complexity.

An example of a list of dictionaries returned by the function is shown below:


```
[
 {
 "Name": "Customer 1",
 "Order": "8 Nuggets",
 "Latitude": 41.705669,
 "Longitude": -86.247528,
 "Mode": "Walk"
 },
 {
 "Name": "Customer 2",
 "Order": "12 Nuggets Meal",
 "Latitude": 41.705402,
 "Longitude": -85.235831,
  "Mode": "Run"
 },
 {
 "Name": "Customer 3",
 "Order": "Sandwich Meal",
 "Latitude": 41.753510,
 "Longitude": -86.213130,
 "Mode": "Scooter"
 },
{
 "Name": "Customer 4",
   "Order": "Waffle Fries (M)",
 "Latitude": 41.697904,
   "Longitude": -86.231130,
   "Mode": "Bike"
 }
]
```


Code

The first step of orderprep.py is to create between five and fifteen randomly generated data points (Chick-Fil-A's "student customers"). This process occurs in the function, ***load_restaurant_data*** which takes in one parameter: the number of customers data should be generated for. Fields including their order, location, and mode of transportation are randomly chosen from a list of pre-selected choices. These

new customers are added to the list, new_customers, which is returned from the function, ***load_restaurant_data***, so that it is accessible in the main function. Here's what the code looks like:

```
# RANDOMLY GENERATE CUSTOMER DATA
def load_restaurant_data(num_customers, TOTAL_ORDERS):

        new_customers = []

        for i in range(num_customers):

                customer = {"Name": "", "Order": "", "Latitude": 0.0, "Longitude": 0.0, "Mode": ""}
                customer["Name"] = f'Customer {TOTAL_ORDERS}'

                customer["Order"] = ORDERS[random.randint(0, len(ORDERS)-1)]

                customer_loc = LOCATIONS[random.randint(0, len(LOCATIONS)-1)]

                customer["Latitude"] = customer_loc[0]
                customer["Longitude"] = customer_loc[1]

                customer["Mode"] = MODES[random.randint(0, len(MODES)-1)]

                new_customers.append(customer)

                TOTAL_ORDERS += 1


        return new_customers
```

Afterwards, each order is printed to the terminal. The first step of the function, ***print_orders***, is to calculate the ETA of each customer. This consists of two steps: calculating the distance between the coordinates of the restaurant and customer and then converting this distance to the estimated time until arrival at the restaurant based on the mode of transportation. Finding the distance between the restaurant and customer required utilization of the Haversine Formula, which takes two points on a sphere (in this case, Earth), and calculates the distance between them. The latitude and longitude are sent to the ***calculate_distance*** function. Both the coordinates of the customer and the restaurant are then converted to radians using the radians function from the math package. The differences in latitude and longitude are calculated, and then the equation is used. The distance is then calculated by multiplying the product of the previous step by the radius of the earth in miles, to give the answer in miles. Here's what the code *looks* like:

```
# FUNCTION TO CALCULATE THE DISTANCE FROM RESTAURANT TO USER
def calculate_distance(user_lat, user_long):

        # RESTAURANT + USER COORDINATES CONVERTED TO RADIANS
        r_lat_rad = radians(CHICK_LAT)
        r_long_rad = radians(CHICK_LONG)
        u_lat_rad = radians(user_lat)
        u_long_rad = radians(user_long)

        # CALCULATES THE CHANGE IN LAT + LONG BTWN CUSTOMER + RESTAURANT
        d_lat  = u_lat_rad  - r_lat_rad
        d_long = u_long_rad - r_long_rad

        # HAVERSINE'S FORMULA TO CALCULATE DISTANCE
        a = sin(d_lat / 2)**2 + cos(r_lat_rad) * cos(u_lat_rad) * sin(d_long / 2)**2
        c = 2 * asin(sqrt(a))

        r = 3956 # RADIUS OF THE EARTH IN MILES

        distance = r*c

        # RETURNS DISTANCE ROUNDED TO 5 DECIMAL PLACES
        return round(distance, 5)
```

Next, the time it will take the customer to arrive at Chick-Fil-A is calculated based upon their mode of transportation. To convert the distance to time, the mode of transportation is required. The function ***calculate_time*** is called, using the distance previously found and the mode of transportation of the customer. A simple velocity variable "v" was defined, which is set to a certain value of minutes per mile depending on the mode of transportation. For walking, for example, the average walking speed was found to be around 3 mph, so a "v" value of 20 was used; it would take 20 minutes to walk a mile at that speed. Similar values were found for biking, running, and riding electric scooters; their speeds were found and converted to minutes per mile. Subsequently, all that had to be done after the declaration of this variable was to multiply the distance by this v value, solving for the amount of minutes to travel that distance or the ETA. The amount of time it will take the customer to arrive at Chick-Fil-A is then returned to the ***print_orders*** function, rounded to the nearest minute. Here's what the code *looks* like:

```
# CALCULATE ESTIMATED USER TIME TO RESTAURANT IN MINUTES
def calculate_time(distance, mode):

        v = 0

        if mode == "Walk":
                v = 20
        elif mode == "Run":
                v = 10.625
        elif mode == "Bike":
                v = 5
        elif mode == "Scooter":
                v = 4

        time = v*distance
        return round (time)
```

Afterwards, each order is added to the priority queue, by calling the ***add_order_to_PQ***. This function takes three parameters: the priority queue itself, the order currently being added, and the time it will take the customer of this particular order to reach the restaurant. This function is as simple as it sounds. It utilizes .put() and adds the order to the priority queue, saving its attributes along with it (customer number, order, and mode of transportation) for future accessibility.

Next, the priority queue is printed in the ***printPQ*** function. This function contains a loop, which uses the .get() command to deque the highest priority elements from the queue. A new priority queue is also created to save each value because .get() pops them off the priority queue. Next, each customer's ETA is decremented by the current processing time. For our simulation the processing time for each order is three minutes. Additionally, every three minutes, eight orders are completed. Here's what the code *looks* like:

```
# PRINT THE PRIORITY QUEUE
def printPQ(thePQ, processing_time):
        print('\033[1m' + "\nORDER QUEUE:" + '\033[0m')
        newPQ = PriorityQueue()

        while not thePQ.empty():
                next_item = thePQ.get()
                print(f'{next_item[1][0]}\t\tETA: {next_item[0]} minutes\t\tOrder: {next_item[1][1]}\t\tMode of Transportation: {next_item[1][2]}')
                next_item[0] -= processing_time;
                newPQ.put(next_item)

        print("")

        return newPQ
```

Then, the code loops back to the top in an infinite loop so as many simulations can occur as one wants. Since eight orders can be processed every three minutes, the first eight orders on the priority queue are printed as completed. This list is reset each iteration for clarity purposes, so one only sees the newest orders that have been completed.

Results

As planned, our team successfully developed a system simulating real-world online-ordering that queues order preparations based on the estimated time of arrival of each customer. A screenshot of our program, orderprep.py. is shown below:

orderprep.py

```
There are currently 5 employees working.
For this simulation, 8 orders are completed every 3 minutes.


NEW ORDERS:
Customer 1              ETA: 1 minutes          Order: Market Salad             Mode of Transportation: Bike
Customer 2              ETA: 1 minutes          Order: Spicy Sandwich           Mode of Transportation: Scooter
Customer 3              ETA: 5 minutes          Order: Grilled Chicken          Mode of Transportation: Walk
Customer 4              ETA: 3 minutes          Order: Grilled Chicken          Mode of Transportation: Run
Customer 5              ETA: 1 minutes          Order: Vanilla Shake            Mode of Transportation: Bike
Customer 6              ETA: 5 minutes          Order: Frosted Lemonade         Mode of Transportation: Run
Customer 7              ETA: 3 minutes          Order: 12 Nuggets               Mode of Transportation: Run
Customer 8              ETA: 0 minutes          Order: Chicken Sandwich         Mode of Transportation: Scooter

PREPARATION QUEUE:
Customer 8              ETA: 0 minutes          Order: Chicken Sandwich         Mode of Transportation: Scooter
Customer 1              ETA: 1 minutes          Order: Market Salad             Mode of Transportation: Bike
Customer 2              ETA: 1 minutes          Order: Spicy Sandwich           Mode of Transportation: Scooter
Customer 5              ETA: 1 minutes          Order: Vanilla Shake            Mode of Transportation: Bike
Customer 4              ETA: 3 minutes          Order: Grilled Chicken          Mode of Transportation: Run
Customer 7              ETA: 3 minutes          Order: 12 Nuggets               Mode of Transportation: Run
Customer 3              ETA: 5 minutes          Order: Grilled Chicken          Mode of Transportation: Walk
Customer 6              ETA: 5 minutes          Order: Frosted Lemonade         Mode of Transportation: Run
```

three simulated minutes later …

```
There are currently 5 employees working.
For this simulation, 8 orders are completed every 3 minutes.


NEWLY COMPLETED ORDERS:
Customer 8              ETA: -3 minutes         Order: Chicken Sandwich         Mode of Transportation: Scooter
Customer 1              ETA: -2 minutes         Order: Market Salad             Mode of Transportation: Bike
Customer 2              ETA: -2 minutes         Order: Spicy Sandwich           Mode of Transportation: Scooter
Customer 5              ETA: -2 minutes         Order: Vanilla Shake            Mode of Transportation: Bike
Customer 4              ETA: 0 minutes          Order: Grilled Chicken          Mode of Transportation: Run
Customer 7              ETA: 0 minutes          Order: 12 Nuggets               Mode of Transportation: Run
Customer 3              ETA: 2 minutes          Order: Grilled Chicken          Mode of Transportation: Walk
Customer 6              ETA: 2 minutes          Order: Frosted Lemonade         Mode of Transportation: Run

NEW ORDERS:
Customer 9              ETA: 0 minutes          Order: Chicken Sandwich         Mode of Transportation: Bike
Customer 10             ETA: 0 minutes          Order: Market Salad             Mode of Transportation: Bike
Customer 11             ETA: 1 minutes          Order: Market Salad             Mode of Transportation: Scooter
Customer 12             ETA: 3 minutes          Order: Spicy Deluxe             Mode of Transportation: Run
Customer 13             ETA: 2 minutes          Order: Chicken Sandwich         Mode of Transportation: Bike
Customer 14             ETA: 8 minutes          Order: Frosted Lemonade         Mode of Transportation: Walk
Customer 15             ETA: 4 minutes          Order: Spicy Deluxe             Mode of Transportation: Run
Customer 16             ETA: 1 minutes          Order: Chicken Sandwich         Mode of Transportation: Scooter

PREPARATION QUEUE:
Customer 10             ETA: 0 minutes          Order: Market Salad             Mode of Transportation: Bike
Customer 9              ETA: 0 minutes          Order: Chicken Sandwich         Mode of Transportation: Bike
Customer 11             ETA: 1 minutes          Order: Market Salad             Mode of Transportation: Scooter
Customer 16             ETA: 1 minutes          Order: Chicken Sandwich         Mode of Transportation: Scooter
Customer 13             ETA: 2 minutes          Order: Chicken Sandwich         Mode of Transportation: Bike
Customer 12             ETA: 3 minutes          Order: Spicy Deluxe             Mode of Transportation: Run
Customer 15             ETA: 4 minutes          Order: Spicy Deluxe             Mode of Transportation: Run
Customer 14             ETA: 8 minutes          Order: Frosted Lemonade         Mode of Transportation: Walk
```

As shown above, customer orders enter the program in chronological order. However, they are sorted into the Preparation Queue based on their ETA. Every simulated three minutes, eight customer orders are moved to the Newly Completed Orders list. (Note that the simulated period, as well as the number of workers and number of orders completed per period are arbitrary values and can be changed within the program upon request). Taking note of the Newly Completed Orders list, a negative ETA indicates that a customer waited for the number of minutes associated with the absolute value of the ETA. For example, Customer 8 waited three minutes for their order upon arrival. On the other hand, any positive ETA in the Newly Completed Orders column indicates that that customer did not have any wait time.

To test the effectiveness of our program in reducing overall wait times for customers, we compared our results against a theoretical benchmark. In order to do so, we created a second version of our program, called reg_orderprep.py, that prepares orders on a first-order, first served basis by changing the priority queue to a queue. A screenshot of the new output is shown below:

reg_orderprep.py

```
There are currently 5 employees working.
For this simulation, 8 orders are completed every 3 minutes.

NEW ORDERS:
Customer 1          ETA: 0 minutes      Order: Spicy Deluxe          Mode of Transportation: Walk
Customer 2          ETA: 0 minutes      Order: Grilled Chicken       Mode of Transportation: Walk
Customer 3          ETA: 1 minutes      Order: Chicken Sandwich      Mode of Transportation: Scooter
Customer 4          ETA: 1 minutes      Order: Frosted Lemonade      Mode of Transportation: Bike
Customer 5          ETA: 0 minutes      Order: 8 Nuggets             Mode of Transportation: Scooter
Customer 6          ETA: 6 minutes      Order: 8 Nuggets             Mode of Transportation: Walk
Customer 7          ETA: 1 minutes      Order: Vanilla Shake         Mode of Transportation: Bike
Customer 8          ETA: 6 minutes      Order: Spicy Sandwich        Mode of Transportation: Walk
Customer 9          ETA: 9 minutes      Order: Frosted Lemonade      Mode of Transportation: Walk
Customer 10         ETA: 1 minutes      Order: Spicy Deluxe          Mode of Transportation: Bike
Customer 11         ETA: 0 minutes      Order: 12 Nuggets            Mode of Transportation: Run
Customer 12         ETA: 0 minutes      Order: Market Salad          Mode of Transportation: Run

ORDER QUEUE:
Customer 1          ETA: 0 minutes      Order: Spicy Deluxe          Mode of Transportation: Walk
Customer 2          ETA: 0 minutes      Order: Grilled Chicken       Mode of Transportation: Walk
Customer 3          ETA: 1 minutes      Order: Chicken Sandwich      Mode of Transportation: Scooter
Customer 4          ETA: 1 minutes      Order: Frosted Lemonade      Mode of Transportation: Bike
Customer 5          ETA: 0 minutes      Order: 8 Nuggets             Mode of Transportation: Scooter
Customer 6          ETA: 6 minutes      Order: 8 Nuggets             Mode of Transportation: Walk
Customer 7          ETA: 1 minutes      Order: Vanilla Shake         Mode of Transportation: Bike
Customer 8          ETA: 6 minutes      Order: Spicy Sandwich        Mode of Transportation: Walk
Customer 9          ETA: 9 minutes      Order: Frosted Lemonade      Mode of Transportation: Walk
Customer 10         ETA: 1 minutes      Order: Spicy Deluxe          Mode of Transportation: Bike
Customer 11         ETA: 0 minutes      Order: 12 Nuggets            Mode of Transportation: Run
Customer 12         ETA: 0 minutes      Order: Market Salad          Mode of Transportation: Run
```

three simulated minutes later …

```
There are currently 5 employees working.
For this simulation, 8 orders are completed every 3 minutes.

NEWLY COMPLETED ORDERS:
Customer 1              ETA: -3 minutes         Order: Spicy Deluxe             Mode of Transportation: Walk
Customer 2              ETA: -3 minutes         Order: Grilled Chicken          Mode of Transportation: Walk
Customer 3              ETA: -2 minutes         Order: Chicken Sandwich         Mode of Transportation: Scooter
Customer 4              ETA: -2 minutes         Order: Frosted Lemonade         Mode of Transportation: Bike
Customer 5              ETA: -3 minutes         Order: 8 Nuggets                Mode of Transportation: Scooter
Customer 6              ETA: 3 minutes          Order: 8 Nuggets                Mode of Transportation: Walk
Customer 7              ETA: -2 minutes         Order: Vanilla Shake            Mode of Transportation: Bike
Customer 8              ETA: 3 minutes          Order: Spicy Sandwich           Mode of Transportation: Walk

NEW ORDERS:
Customer 13             ETA: 0 minutes          Order: Vanilla Shake            Mode of Transportation: Bike
Customer 14             ETA: 2 minutes          Order: 12 Nuggets               Mode of Transportation: Scooter
Customer 15             ETA: 1 minutes          Order: Vanilla Shake            Mode of Transportation: Scooter
Customer 16             ETA: 5 minutes          Order: Spicy Deluxe             Mode of Transportation: Run
Customer 17             ETA: 4 minutes          Order: Vanilla Shake            Mode of Transportation: Run
Customer 18             ETA: 4 minutes          Order: 12 Nuggets               Mode of Transportation: Run
Customer 19             ETA: 1 minutes          Order: Frosted Lemonade         Mode of Transportation: Bike
Customer 20             ETA: 5 minutes          Order: Grilled Chicken          Mode of Transportation: Run
Customer 21             ETA: 4 minutes          Order: Spicy Deluxe             Mode of Transportation: Run

ORDER QUEUE:
Customer 9              ETA: 6 minutes          Order: Frosted Lemonade         Mode of Transportation: Walk
Customer 10             ETA: -2 minutes         Order: Spicy Deluxe             Mode of Transportation: Bike
Customer 11             ETA: -3 minutes         Order: 12 Nuggets               Mode of Transportation: Run
Customer 12             ETA: -3 minutes         Order: Market Salad             Mode of Transportation: Run
Customer 13             ETA: 0 minutes          Order: Vanilla Shake            Mode of Transportation: Bike
Customer 14             ETA: 2 minutes          Order: 12 Nuggets               Mode of Transportation: Scooter
Customer 15             ETA: 1 minutes          Order: Vanilla Shake            Mode of Transportation: Scooter
Customer 16             ETA: 5 minutes          Order: Spicy Deluxe             Mode of Transportation: Run
Customer 17             ETA: 4 minutes          Order: Vanilla Shake            Mode of Transportation: Run
Customer 18             ETA: 4 minutes          Order: 12 Nuggets               Mode of Transportation: Run
Customer 19             ETA: 1 minutes          Order: Frosted Lemonade         Mode of Transportation: Bike
Customer 20             ETA: 5 minutes          Order: Grilled Chicken          Mode of Transportation: Run
Customer 21             ETA: 4 minutes          Order: Spicy Deluxe             Mode of Transportation: Run
```

As shown above, this simulation is very similar to our program. However, the critical difference is that customers are sorted in the Preparations Queue identically to their *chronological order* placement. That is, orders are processed on a first-come, first-served basis.

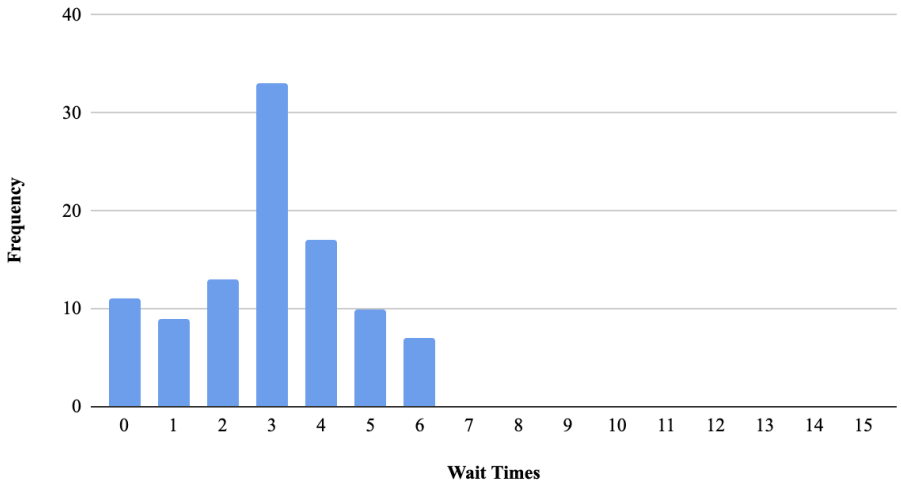
To compare the performance of orderprep.py versus reg_orderprep.py, we ran each program until 100 customers' orders were complete. Afterwards, we recorded the ETA of each customer at the time their order was complete. More specifically, we recorded the absolute value of negative ETAs and recorded 0s for any positive ETA. Initial results to this comparison are listed in the chart below:

	orderprep.py	**reg_orderprep.py**
Average Wait-Time	2.97	4.88
Standard Deviation	1.6419	3.6852
Range (Maximum)	6	15

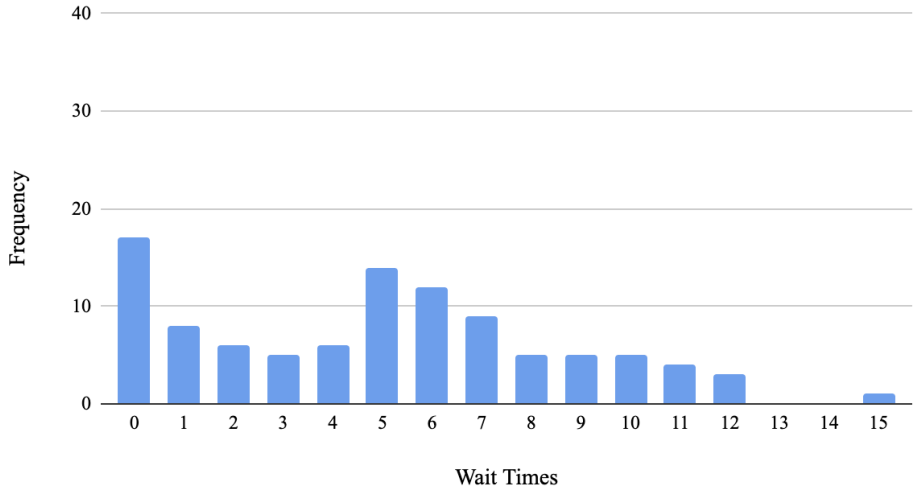
As shown in the table above, customers in the orderprep.py simulation had an average wait time of 2.97 minutes. This is approximately a 2-minute reduction in wait time from reg_orderprep.py. Additionally, the standard deviation from the orderprep.py simulation is lower, indicating that the variability in wait-times

was higher in the reg_orderprep.py. A similar trend is shown by the range (which in our case is also the maximum, since any simulation's minimum is always a wait time of 0 minutes). The longest a customer had to wait with our priority queue was 6 minutes, whereas the longest wait time for a customer in reg_orderprep.py was 15 minutes. A visual representation of this data is shown below:

orderprep.py



reg_orderprep.py



The most prominent result we can decipher from these charts is the difference in concentrated wait times. The wait times from the orderprep.py simulation are concentrated at the left side of the graph, indicating lower wait times. The wait times from reg_orderprep.py, on the other hand, are more spread out along the scale.

Although orderprep.py has lower wait times on average, one statistic that we found interesting and unexpected is that reg_orderprep.py had more customers with a wait time of 0. After further analysis of the simulation, we found that the 0 wait times end after the 64th customer. Overall, though, we still predict that a priority queue would better suit restaurants and customers based on the average wait time, standard deviation, and range.

Conclusion

At the conclusion of this project, our team has successfully developed an improved process for sorting and preparing orders which demonstrates significant potential for improving the wait times of customers at restaurants. It requires data which is either readily available or easily receivable. The order is already received by the restaurant in current ordering processes, the location of the customer can easily be received via location services (with user consent), and the mode of transportation would simply be requested from the customer. Should the customer not enable location services or provide a mode of transportation, the function can simply default to a certain location and mode of transportation, such as the center of campus and walking.

The program could easily be adapted to take into account further conditions which could affect wait times, such as the size of the order. A calculated preparation time requirement could be incorporated into determining the weight for placement into the priority queue, allowing for a more accurate and efficient preparation process. This would likely require detailed information and statistics from the restaurant being simulated.

While the comparison of the use of a priority queue to a normal queue shows that the function does indeed reduce the average waiting time of the customer, it is uncertain how it would work in the real world. As mentioned above, other circumstances mean that this data is not a perfect representation of what truly happens. Other points of consideration for implementation in real life would be a cap on the amount an order may be pushed down in the queue, the size of the order as mentioned above, and many other factors which would have to be researched before full incorporation. Nevertheless, the results of this program show promise and should be considered by online food ordering companies. Our algorithm is both easier on the customer and the employees. Happy employees lead to satisfied companies.