**COP 4530: Final Project**

**Team:**

Benjamin Barrera-Altuna

Hieu Nguyen

Richard Zhou

Project presentation video link: [COP4530 Final Project.mp4](https://drive.google.com/file/d/1ujwa8U_D2UzQGXWYth5sQrtxfMGvdL0U/view?usp=share_link)

<https://drive.google.com/file/d/1ujwa8U_D2UzQGXWYth5sQrtxfMGvdL0U/view?usp=share_link>

**Abstract/Overview:**

Airport immigration processing has always been one of the most time-consuming processes for travelers due to the huge number of people entering the United States every day. This program aims to improve efficiency by reducing the time taken to sort people and decide the priority of certain groups based on their information, including: country status (US citizen, doesn’t need a visa, needs a visa), traveler class (first class, business, economy), presence of disabilities, and veteran status. This program first begins by taking a text file and reading the data (formatted with comma separators) to create a vector of traveler instances. From this vector of traveler instances, the data is first sorted via either merge or quick sort based solely on their country status. The reasoning for this is that we wish to create separate subgroups of travelers based on their citizenship and visa status. From this sorted vector, we further split the vector into three sub-vectors based on each respective country status group previously mentioned. From each created sub-vector, further sorting is performed based on the other attributes the travelers have (including all but country status, as it was already used). To sort the travelers with the same country status, a priority queue and heap sort are used for sorting. We sort the travelers based on their attributes, which are as follows (in order): disabilities, first-class, veteran, business, and economy. A traveler node class is also used to construct a binary tree at first, which is used later in the heap sort process. Once the sub-group vector of the travelers had been sorted, an implementation of a binary search tree was created using all three sub-groups. From this binary search tree, priority values were given to each passenger (based on their sorted vector and their respective sub-group). These priority values were used as keys to create the binary search tree. This data structure was selected to allow efficient data access and deletion in case a passenger needs to be removed due to travel challenges or detention and to print out all travelers (based on their priority) with in-order traversal. Ultimately, this program allows for an applied showcase implementation of both efficient sorting algorithms (merge sort, quick sort, and heap sort) and data structures for data retrieval and deletion management (binary search tree). Future improvements to the program could possibly include a comparison of different sorting algorithms to make conclusions on which would be most suitable (which is dependent on the traveler data) and creating AVL trees rather than binary search trees to ensure a balanced tree (allowing for stable data retrieval or deletion).

**Details of the Implementation**

***Traveler, Traveler Node, and Read Travelers Functions & Classes:***

The main data unit used in this project is the ***“Traveler”*** (Traveler.hpp) class. Instances of this traveler class are used to represent the travelers at the immigration processing lines. Each traveler carries a name, country status (US citizen, doesn’t need visa, needs visa: represented as enum), traveler class (first class, business, economy: represented as enum), presence of disabilities (bool), and whether they are a veteran (bool). Originally, data of travelers are stored within a text file. To read this data, the ***readTravelersFile()*** (readTravelers.hpp) function is utilized to read all the data as formatted (comma separated for attributes, line separated for travelers) and then return a vector of traveler instances (unsorted). Moreover, a ***“travelerNode”*** (immigrationProcessing.hpp) class is created (for heaps and binary trees). The travelerNode class is used to create instances of travelerNodes that contain a ***“Traveler”*** as their main data storage along with a left and right pointer variables. The class also comes with getter (for traveler attributes), setter (for priority value only), and compare methods along with a priority value for each traveler that will be initialized later on.

***Heap:***

We first have a class called ***“TravelerNode”*** (TravelerNode.hpp). This class creates a binary tree of the travelers, which we can then use for heapsort; there is also a ***“compare”*** (TravelerNode.hpp) class which we can use to compare travelers, and sort the priority queue. The ***“compare”*** (TravelerNode.cpp) class implementation compares the following attributes in order: disabilities, first-class, veteran, business-class, and economy-class. We also have a ***“PriorityQueue”*** (PriorityQueue.hpp) class, which we use to insert the traveler nodes from the heap.

***Immigration Processing.(cpp/hpp):***

***\*\*\* the immigration processing file(s) are subdivided into 3 sections as follows : \*\*\****

***Section I:***

The immigration processing process begins with the sorting of the “Travelers” based on their country status represented as enum: “US citizen” (0), “does not need visa” (1), and “needs visa” (2). The read data, now in the form of an unsorted vector of traveler instances thanks to the ***readTravelersFile()*** function, will be sorted using two different methods: MergeSort and QuickSort.

**Method 1: MergeSort**

MergeSort requires two functions: ***merge()*** and ***mergeSort()***

* The ***merge()*** function takes 5 parameters: the merged vector, the left vector, the left vector’s size, the right vector, and the right vector’s size.

This function utilizes a while loop to iterate through both left and right vectors at the same time, comparing the elements and appending the smaller element to the merged vector. The vector (left/right) of which an element got added to the merged vector will have the iterator incremented so that its next element will be compared to the current element (not added to the merged vector) of the other vector.

The while loop ends when either of the iterators of both left and right vectors exceeds their respective vector’s sizes. Then, two while loops are added to add any remaining elements to the merged vector, resulting in a singular sorted vector.

* The ***mergeSort()*** function takes 3 parameters: the vector needs sorting, the beginning index, and the ending index of the vector.

***mergeSort()*** works by dividing the targeted data in half and calling mergeSort on each half. In order to do so, the midpoint of the vector is calculated with the indexes. Then, two traveler vectors are initialized with the first one (leftArr) containing the data in the first half of the targeted vector, and the second one (rightArr) containing the rest of the data. The data is transferred to the two sub-vectors from the targeted vector through two for loops.

After the data has been divided into two halves, mergeSort is called on both of the sub-vectors. As ***mergeSort()*** is a recursive algorithm, each half will also be divided until the sub-vectors have at most one element.

At the end, the sorted subvector will be merged using the ***merge()*** function mentioned above.

The mergeSort function is encapsulated in the ***passengerStatusOrganizerMS()*** method of the ***immigrationProcessing*** class, returning the sorted vector of traveler instances.

**Method 2: QuickSort**

QuickSort requires three functions: ***compareByCountryStatus()***, ***partition()***, and ***quicksort()***

* The ***compareByCountryStatus()*** function takes two traveler instances as parameters and returns a boolean output of (first traveler instance’s ***countryStatus*** < second traveler instance’s ***countryStatus***)
* The partition() function takes 3 parameters: the targeted array/vector, **low** index, and **high** index.

The pivot point is first chosen as **arr[high]**. Two iterators i and j were chosen as **low-1** and **low** respectively. A for loop is added as j increments until it reaches **high**. As the loop operates, if **arr[j]** is less than the pivot point (through ***compareByCountryStatus()*** function), i will be incremented as **arr[i]** and **arr[j]** will be swapped. When the loop is done, **arr[i+1]** will be swapped with **arr[high]** and **i+1** will be returned. **i+1** will serve as the point to divide the vector needs sorting into LHS and RHS, allowing for recursive calling of the sorting method.

* The ***quicksort()*** function takes 3 parameters: the array/vector needs sorting, the **low** index, and the **high** index. An integer **pi** is assigned the value received after the partition() function is called on the targeted vector. ***quicksort()*** is then called on the same array twice, with the first instance having **low** and **pi-1** as its parameters, and the second instance having **pi+1** and **high** as its parameter, allowing the vector to be sorted recursively.

The quicksort function is encapsulated in the ***passengerStatusOrganizerQS()*** method of the ***immigrationProcessing*** class, returning the sorted vector of traveler instances.

***Section II:***

This section is the second sorting process. Within each sub-array of traveler country status, the travelers will be sorted again in their own sub-array based on the following attributes (in order): disabilities, first-class, veteran, business-class, and economy-class. To start, we have to create a binary tree of travelers using the ***“TravelerNode”*** (TravelerNode.hpp) class. In the ***groupPrioritySorting()*** function (immigrationProcessing.cpp), we first created a priority queue using our custom PriorityQueue class (PriorityQueue.hpp). Then we traverse through the vector of travelers, make a node for every single traveler and add them to the priority queue. The priority queue also takes in a compare function which will compare the travelers and decide who gets priority over the traveler’s attributes. Lastly, after the priority queue is constructed, we pop the minimum node and add it into another array which will be the sorted array for the travelers.

***Section III:***

This section goes over the creation of the binary search tree. To create the binary search tree, the ***bstCreation()*** function is used. As inputs, it takes in a vector of travelers vectors (each outer vector represents a sub group of travelers categorized by country status, the inner vectors represent the sub groups sorted by other traveler attributes). Starting from the middle vector (outer vector) and taking the index of the middle value (vector[middle][middle]), we use that as the root node with the priority value set to the respective index it falls under. Afterwards, starting from the beginning vector (outer), we iterate through all travelers and insert them into a binary search tree with the ***bstInsert()*** function (priority value starts at 0, and will skip the insertion of the middle person since they were the first to be added). Using this method avoids the need to create an AVL tree as the tree will be balanced. We also additionally implement a node deletion function, ***deleteBSTNode()***, to delete nodes in the binary search tree given a priority value as input (along with the root node of the binary search tree).

***main.cpp***

The main.cpp file contains the sample usage of all the function implementations described in sections 1 to 3. To begin, a vector filled with traveler instances (unsorted) is created by reading the travelers text file. Then, the section I functions are used to sort the travelers into subgroups based on their country status (either merge or quick sort). Section II is then followed to sort the subgroups using heaps. Finally Section III creates a binary search tree using all the subgroups as data inputs (highest priority going to US citizens and lowest to visa required foreigners). The outputs of each respective function are then demonstrated through print statements.

**Challenges**

***Section I:***

For this section, there were not a lot of difficult challenges during the process of implementing the two sorting functions on the data. However, multiple attempts have been made regarding some additional functions to highlight the difference in terms of time and space complexities of the two sorting algorithms. Our initial idea involved comparing the time taken to sort the data with ***mergeSort()*** and ***quicksort()***, and we used the chrono library to calculate the time period. However, the result came back in seconds, and as the two sorting algorithms both executed in such a short time (close to 0), we believe it was not a good indication for comparison. There are different ways we could go around this, for example, by creating a function to convert from seconds to different units such as nanoseconds, yet that involves a lot of additional functions that we don’t believe are necessary to our program. Additionally, ***quicksort()***’s time complexity heavily depends on the pivot point chosen. As a result, we have decided not to include this aspect of the two sorting algorithms to our program.

***Section II:***

This section didn’t have a lot of challenges during the process of implementation. I did have to figure out how I can incorporate the travelers with the priority queue. It also took us a while to decide on how we were going to sort the travelers. Neither of us had any clue how the airport immigration process worked so we had to do extensive research. After deciding on how the travelers will be sorted, it was easy for me to create the compare class for the priority queue. All I had to do after was just to remove the minimum node from the priority queue and insert it to the final sorted array.

***Section III:***

The main challenge with implementing this section was how the binary search tree would be created. Since each node was being placed in descending order of priority (starting from most important to least important), we had to ensure that the starting root node would allow for a balanced binary search tree otherwise the tree would have been imbalanced and not had efficient data access. Instead of implementing an AVL, we simplified the process by ensuring that the root node of the tree would be the traveler with the priority value in the middle of both the lowest and highest priority value. This essentially ensures that the binary search tree is balanced and has efficient data accessing/deletion.

**Contributions**

***Hieu Nguyen:***

I am responsible for section I of the ImmigrationProcessing process which involves the implementation of mergeSort and quickSort to sort our dataset of traveler instances based on CountryStatus, enabling section II and III to take place.

***Richard Zhou:***

For this project, my job was to figure out how to sort the travelers efficiently. Heap sort immediately came to mind when I was deciding on a sorting algorithm to implement. I created a priority queue class for the use of heapsort, then proceeded to complete the class method for immigrationProcessing, completing the sorting for travelers based on their attributes.

***Benjamin Barrera-Altuna:***

My contributions to the project are as follows: I created the traveler class along with the read traveler file function which was utilized as the primary data unit. I also created the traveler node used for the heap and binary tree construction. Beyond that, my central contribution to this project was the creation of the binary search tree (including its helper functions). Finally, I assembled all the code together for a final and cohesive project.

**How to Run Instructions**

1. Connect to the student cluster
2. Transfer folder with all files in zip and cd into the folder
3. Inside the folder, run the following command: **g++ -std=c++11 -Wall \*.cpp**
4. Once the command has finished compiling, run the following to check the outputs of the program: **./a.out -s**

**Side Note:** *if you are having trouble connecting to the student cluster, ensure you are connected to the USF VPN to access the cluster. Additionally, we recommend WinSCP to transfer the folder.*