Technical Report: Final Project
EECE 2560: Fundamentals of Engineering Algorithms

Christopher Lam, Dante LoPriore, Jasime Sajna
Department of Electrical and Computer Engineering
Northeastern University
`lopriore.d, lam.chris, sajna.j @example.com`

November 18, 2024

Contents

1 Project Scope

1.1 Finalized Project Scope

The aim of this project is to design and develop a user-friendly flight reservation system to simulate for customers how they would book, cancel, and view available seats across the offered flights of an airline. This system will explore how data can be dynamically stored and managed using databases, and how this dynamically stored data can be sorted using the linked list data structure. The expected outcomes include a functioning well documented web-based system, a final project report, and a powerpoint presentation.

1.2 Project Performance Requirements To Create Flight Reservation System

The project's objectives for the anticipated system are:

- To allow customers to book, cancel, and search for an available seat using an interactive local web-based flight reservation system, and also easily view available flights using their starting and destination locations.
- Provide the current booking trends and flight availability statistics to help the airline make profitable decisions based on the given data results.
- Preform features with fast response times and allow for many users to interact with the system without any changes in performance.

1.3 Project Code Implementation Objectives

The project's coding secure architecture objectives are:

- **Frond End:** The Front End component would be the team building a responsive web application using a GUI to visualize the flight reservation functions that are set for the users and admin.
- **Back End:** The Back End component would be the team building a the server side of the application that enables the transactions and processes made in a database using Flask and SQL.
- **Emphasis on Linked Lists, BFS, Sorting Algorithms:** The team's primary data structures to include are linked lists, sorting algorithms and breath first search algorithms. The team plan to use the sorting algorithms to create the navigation, searching. The team will use linked lists to represent the nodes as the seats reference the seats with pointer values. BFS algorthim is used to find optimal flight paths.

2 Project Plan

2.1 Brief Timeline

The project is divided into phases, each with specific deliverables:

- **Week 1 (October 7 - October 13)**: Define scope, establish team roles, outline skills/tools.
- **Week 2 (October 14 - October 20)**: Begin development on the bare logic of the system, set up project repository
- **Week 3 (October 21 - October 27)**: Coding, work on backend and frontend separately
- **Week 4 (October 28 - November 3)**: Finish backend, integrate the frontend. Start PowerPoint presentation.
- **Week 5 (November 4 - November 10)**: Finalize system, organize test scripts, continue documenting in report.
- **Week 6 (November 11 - November 17)**: Revise and finalize technical report and presentation.
- **Week 7 (November 18 - November 28)**: Final presentation, submit report.

2.2 Milestones

Key milestones include:

- Project Scope and Plan (October 7).
- GitHub Repository Setup and Initial Development (October 9).
- Showcase Backend Progress and Completion (October 28).
- Final System Testing and Report Draft (November 10).
- Final Presentation and Report Submission (November 28).

2.3 Schedule Overview

The team divided the tasks based on the team members strengths and expertise, which then enabled the team to organize a plan to successfully complete the project within a timely manner. Figure 1 is a gnatt chart that shows how the team agreed to take on certain sections of the project throughout the semester.

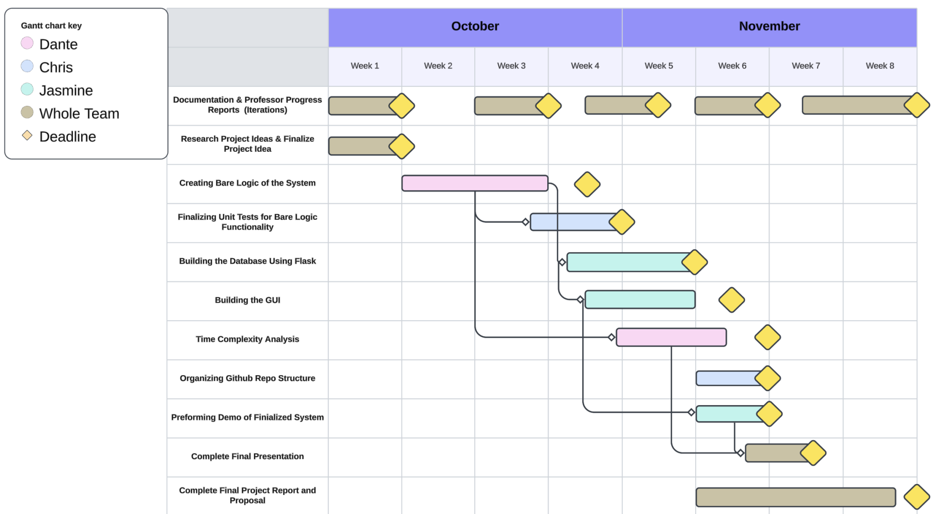

Figure 1: Gnatt Chart Schedule of Events

The team consistently meet after each Algorithms class to check-in with our progress on the desired tasks and determine if the team need to adjust the schedule of events to meet certain deadlines. Before any major project iterations and progress reports were due, the team typically meet a day before the due date on zoom or microsoft team to confirm the team's submission on describing the team's progress.

3 Team Roles

3.1 Primary Roles

- **Team Member 1**: **Chris Lam:** Testing Frontend and Backend Functionality, Time Complexity Analysis, Backend logic, UI
- **Team Member 2**: **Dante LoPriore:** Building and Testing Frontend, Creating Bare Logic, Testing Backend logic, UI
- **Team Member 3**: **Jasmine Sajna:** Testing and Creating Backend on Flask, backend logic, UI

3.2 Team Responsibilities

Each team member will focus on specific areas, but roles may adapt as the project progresses:

- **Frontend Development (Dante, Chris):** In this role, the group would build the user interface for the flight reservation system using a GUI. The GUI will allow the user to see the data visualization of the current navigation options to reserve and cancel an available flight.
- **Backend Development (Jasmine, Dante):** In this role, the group will be responsible for creating a reliable backend system that creates an universal database for users with different roles to schedule a flight, reserve an available seat, and display passengers info. The backend framework uses Flask SQL for the database design and the flight reservation system can be accessed on a local web server.
- **Technical Documentation (Whole Team):** As a collective group, it is each team member's responsibility to document their progress with their role and provide feedback to improve the project's projected goals.
- **System Testing (Jasmine, Chris):** Within this role, the team members would check in with the progress of the code functionality and validating that the system works for the front end to the back end.
- **Time Complexity Analysis (Dante, Chris):** Within this role, the team members would check if the algorithms and code functionality within the bare logic is an optimal solution. Also, this role helps improving code runtime and effeciency within a system.

4 Methodology

4.1 Data Entity Structure and Relationships for the Bare Logic

This section illustrates how the team structured the data entities within the bare logic of the flight reservation system. In Figure 1 displays the UML diagram to reflect the data relationships in the SQL database to model and represent the flight reservation system.

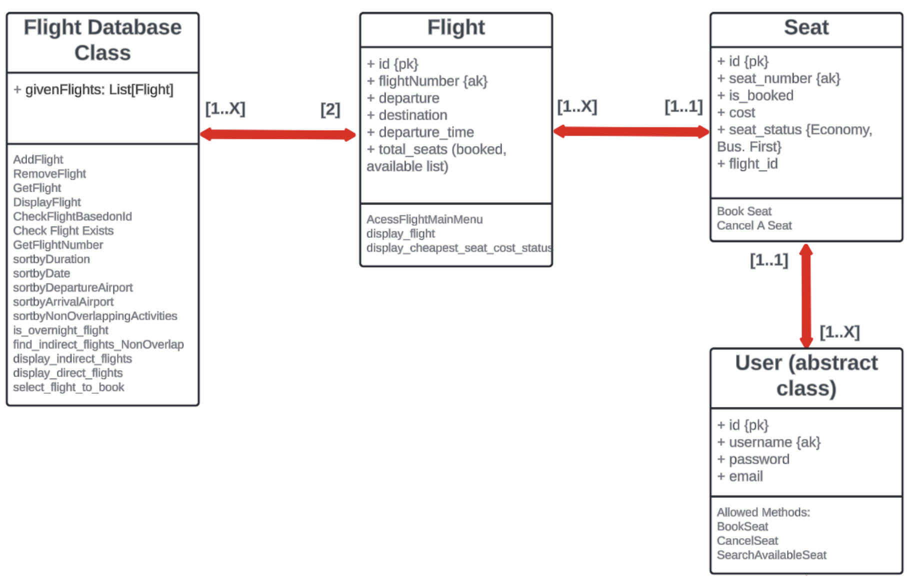

Figure 2: UML Diagram to Describe Data Entities in the Bare Logic

The UML diagram includes a flight database class that manages the multiple flights stored in the system, which practically stores a list of flight objects. This is where the interactive menu would be stored to navigate through each Flight's details, adding flights, deleting flights, filtering flights, and selecting the flights from the list. The framework builds off of how Flights and Seats interact with each other to organize the respective operations for each entity. The team found that each Flight will have a one-to-many relationship with the seats as a flight can have infinite seats, but only one seat can belong to a user on a single flight. Seats include the seat's status and cost based on whether the seat is first class, business class, or economy class. The relationship between the Flightdatabase and the Flight can be one too many as the flight database needs to have at least one Flight to operate the system, and the database can have as many flights as it wants. The relationship of the Flight to the flight database is referenced to the amount of single linked lists for each Flight, which are 2, one for booked flights and another for available flights. Thus, the LinkedLists and Flights relationship is deemed a composition relationship since the linked lists depend on whether the Flight exists in the Flightbase. Overall, the team is pleased with the results for the bare logic to serve as a foundation to help structure the backend structure.

4.2 Linked Lists Data Structure to Represent a Seating List For a Flight

Within this Flight reservation system, the seats are represented as nodes in a linked list and the linked list represents either the avaliable seating or booked seating for a given flight. The data visualization diagram to reflect this system using the linked list can be seen in Figure XX.

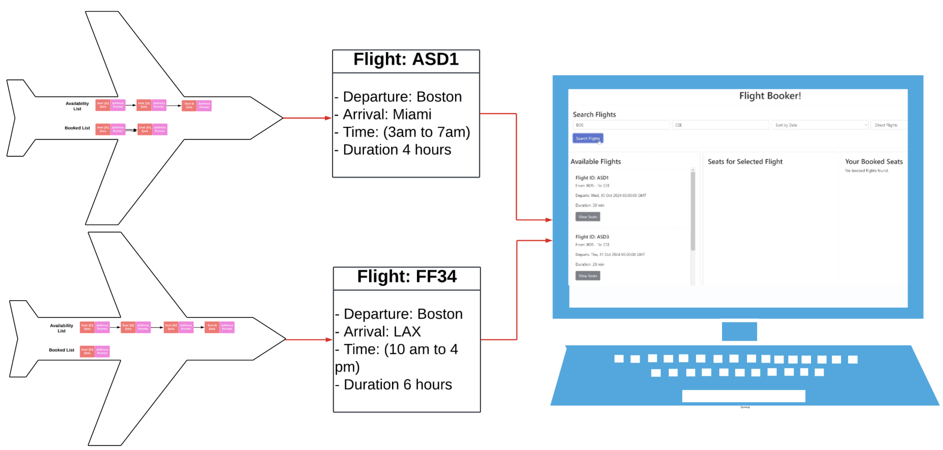

Figure 3: Flight Reservation System Structure Data Visualization

4.3 Indirect Flight Algorithm Methodology

The team considered using a BFS or breath first search sorting algorithm to help find the shortest indirect flight path from the user's inputted departure airport to the desired arrival location. The team used the same tree structure to search for the flights closest to the starting flight and then the path expands based on the given direction and threshold until it reaches the flight to the correct final destination. The team's approach to create the indirect flight path can be best described in a worked example seen using the known flight map seen in Figure XX. For the demonstration I wanted to go from NYC to LAX

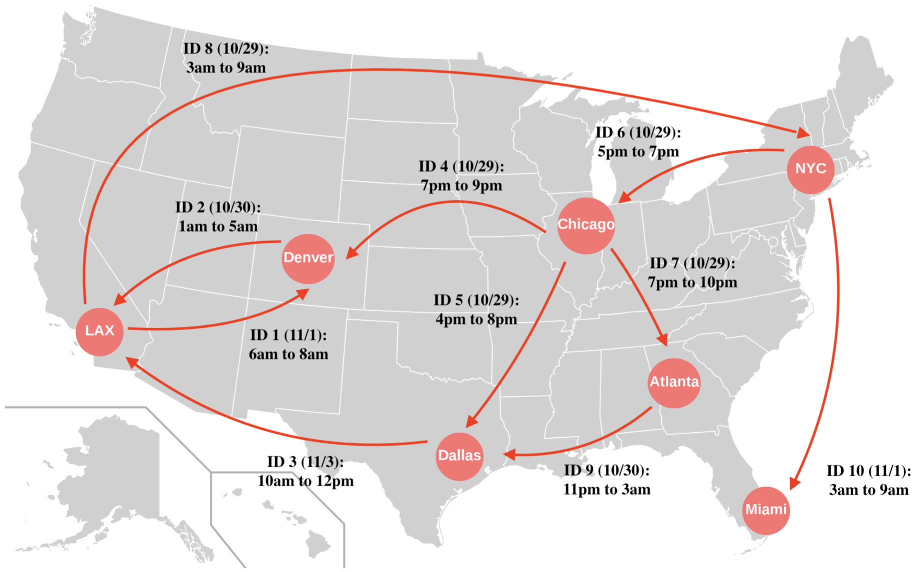

Figure 4: Example of Flight Map with System

4.3.1 Step 1: Initialize Known Variables

Before starting the algorithm, the team initialized a queue for the breath first search that contains the information for the current location, current path of flights represented as a list, the end of the last flight, and the date of the last flight.

4.3.2 Step 2: Create Dictionary Of All Departing Flights Each Airport

The Indirect Flight algorithm starts by creating a dictionary of all the departing flights for each airport. The sample dictionary shows how when we start doing the algorithm

- Key: NYC - Items: Chicago, Miami
- Key: Chicago - Items: Denver, Atlanta, Dallas
- Key: Atlanta - Items: Dallas
- Key: Dallas - Items: LAX
- Key: Denver - Items: LAX
- Key: LAX - Items: Denver, NYC

4.3.3 Step 3: Check if Destination is Reached

The breath first search is supposed to check for all known paths. My approach to this is to dequeue the flight from the BFS queue to extract and get the current flight to preform the BFS algorithm on. Start by doing a check where the current location should be the destination the user wants to reach. If this condition is true, extract all the flight numbers from the flight path or journey it took to reach this destination. Please, note that this will only work in causes where the flight path contains more than two flights.

4.3.4 Step 4: Start by Searching Known Paths

We start searching the known paths by extracting the key from the departure map dictionary to see what flights depart from the current location. In this process, cycles can not occur as current location is put into a set that tells the system to skip over the flight if that location was already used.

If this location is the first flight in the path, then create a path with its first element starting with the current flight and enqueue the next arrival location in the queue

If the fligth path contains any flights check to see if you can create a new path with the current flight added. The constraints to this are the following:

- Non-overlapping Time Intervals - The next flights start fly time is directly after the last flight's end time to ensure no conflicting events at the same tome
- Indirect Flight Path duration has limit of 2 days from original source
- Overnight Flights Are Considered
- Can't Visit Same Airport More Than Once

When all the constraints are meet, then you can enqueue the next location to be added to the bfs queue and also include the flight in the known flight path of indirect flights

4.3.5 Step 5: Do Steps 3 and 4 until the BFS searching has been done

The while loop will do steps 3 and 4 and iterate over all possible permutations of the flight path could be reached for an indirect flight path from the source to destination.

5 Algorithm Design for System's Basic Logic

The current pseudcode list in this section shows all the methods that make up the flight reservation system. The bare logic of the simple system are the sorting and navigation of traverse through seats, and allowing the user to dynamically enter a seat that is available or booked. The bare logic files can be seen in this github link to the branch of the working python file `https://github.com/sajna-j/flight-reservation/blob/bare-logic-in-python`.

5.1 General Booking System Helper Functions

5.1.1 insertHead Method

Algorithm 1 insertHead:

Input: An integer representing seat , *data*
Output (Changed State): A linked list with a new node inserted at the head of the list
Begin Algorithm: insertHead
$newNode \leftarrow SeatNode(data)$
if $head == NULL$ **then**
  $head \leftarrow newNode$
  $return;$
end
$newNode.next \leftarrow this.head$
$this.head \leftarrow newNode$
End Algorithm: insertHead

Frequency Count Analysis:
SeatNode creation: 1 operation
conditional check: 1 operation
assignment of newNode at head: 1 operation
Shifting of node at head back: 1 operation
Frequency Count: F(n) = 4
Time Complexity: O(1)
Purpose: To insert a node at the head of the list

5.1.2 insertAtEnd Method

Algorithm 2 insertAtEnd

Input: An integer representing seat , *data*
Output (Changed State): A linked list with a new node inserted at the tail of the list
Begin Algorithm: insertAtEnd
$newNode \leftarrow SeatNode(data)$
if $head == NULL$ **then**
  $head \leftarrow newNode$
  $return;$
else
  $temp \leftarrow head$
  while $temp.next! = NULL$ **do**
    $temp \leftarrow temp.next$
  end
  $temp.next \leftarrow newNode$
end
End Algorithm: insertAtEnd

Frequency Count Analysis:
newNode creation: 1 operation
conditional check: 1 operation
assignment of newNode: 1 operation

shifting head to temp: 1 operation
while loop: n iterations
shifting temp: n iterations

Frequency Count: F(n) = 2n + 4
Time Complexity: O(n)
Purpose: To insert a node at the tail of the list

5.1.3 deleteNodeAtPosition Method

Algorithm 3 deleteNodeAtPosition

Input: An integer value to represent the positionIndex
Output (Changed State): Get a linked list with a node deleted at a given index position.
Begin Algorithm: deleteNodeAtPosition
$count \leftarrow 0$
if $head == NULL$ **then**
  print $The * list * is * empty$
  $return$
end
if $positionIndex == 0$ **then**
  $temp = head$
  $head \leftarrow temp.next$
  $return$
end
$currNode = head$
while $(currNode.next$ *not equal* $nullptr)$ *and* $(positionIndex - 1$ *not equal* $count)$ **do**
  $count+ = 1$
  $currNode \leftarrow currNode.next$
end
$tempNode \leftarrow currNode.next$
print $DeletingSeat : tempNode.data, Address : currNode.next$
$currNode.next \leftarrow tempNode.next$
delete $tempNode$
End Algorithm: deleteNodeAtPosition

Frequency Count Analysis:
user input of position: 1 operation
count initialization: 1 operation
check head isn't NULL: 1 comparison
check if user wants to remove the first head: 1 comparison
assign head to currNode: 1 operation
while loop iterations: n operations
while loop comparisons: 2*n comparisons
inside the while loop operations: 2*n operations
assign currNode.next to tempNode: 1 operation
print seat being deleted and the address: 1 operation
assign tempNode.next to currNode.next: 1 operation
delete tempNode: 1 operation

Frequency Count: F(n) = $5n + 9$
Time Complexity: O(n)

Purpose: To remove the seat node in a single linked list at a given position index. Iterate over each node of the list and keep a count for each pass of the node in the linked list to keep track of the index to remove the node. When the list reaches the index that the user wants to delete then delete the node and redirect the node's pointers accordingly.

5.1.4 addNodeAtPosition Method

Algorithm 4 addNodeAtPosition

Input: An integer value or index to add the node *positionIndex*, the integer values representing the data in the node *data*

Output (Changed State): The linked list with the added seat based on the user's index

Begin Algorithm: addNodeAtPosition

Initialize SeatNode* newNode = new SeatNode(data)

Set count $\leftarrow$ 0

if *(head is equal to NULLpointer)* **then**

  print $->$ throw error how list is empty

  terminate program

end

if *(positionIndex is equal to 0)* **then**

  newNode$->$next = head

  head = newNode

  terminate program

end

Intialize SeatNode* currNode = head

while *((currNode$->$next not equal to nullptr) and (positionIndex - 1 not equal to count))* **do**

  increment count by 1

  currNode = currNode$->$next

end

print$->$ "Add Seat:" + newNode$->$data + ", Address: " + currNode-¿next

newNode$->$next = currNode$->$next

currNode$->$next = newNode

End Algorithm: addNodeAtPosition

Frequency Count Analysis:

initialize the new node that is going to be added on to the linked list: 1 operation

set a count to keep track the nodes in the linked list: 1 operation

check if the linked list is empty using if comparison: 1 operation

if condition is true throw an error how the operation can be preformed: 1 operation

if condition is true end the program: 1 operation

check if the new node should be placed at the start of the linked list using if comparison: 1 operation

if condition is true reassign the pointers to make the next item in the list for the new node to be the start of the list: 1 operation

if condition is true make the start of the list be the new node: 1 operation

if condition is true end the program: 1 operation

iterate each element in the list via while loop until the current position is found to place the new node: 2n operations

increment the count of the current place of the current node in list by 1: 1 operation

reassign the pointers of the current node to be the next item in the list: 1 operations

print the what the new node being placed into the list: 1 operations reasssign the pointers of the next element of the new node to be the next element of the current node : 1 comparison

reasssign the pointers of the next element of the current node to be the new node : 1 comparison

Frequency Count: F(n) = 2n + 15

Time Complexity: O(n)

Purpose: To add the seat node in a single linked list at a given position index. Iterate over each node of the list and keep a count for each pass of the node in the linked list to keep track of the index to add the node. When the list reaches the index that the user wants to add then add the node and redirect the node's pointers accordingly.

5.2 Book A Seat Function

5.2.1 bookSeat Method

Algorithm 5 bookSeat

Input: A single LinkedList, *availableSeatList*
Output: To add a seat node to the book seat list and remove the booked seat from the available seating list.
Begin Algorithm: bookSeat

$input(intuserSeatInput)$
$indexCount = 0$

while $(currentAvailable.next\ \ not\ \ equal\ \ nullptr) AND (currAvailableSeatNode.data\ \ not\ \ equal\ \ userSeatInput)$ **do**
  $indexCount \leftarrow indexCount + 1$
  $currAvailableSeatNode \leftarrow currAvailableSeatNode.next$
end
if $userSeatInput == currAvailableSeatNode.data$ **then**
  $insertAtEnd(userSeatInput)$
  $print\ seats$
  $availableSeatList.deleteNodeAtPosition(indexCount)$
  $return$
else
  $print\ UnabletoBook!$
end
End Algorithm: bookSeat

Frequency Count Analysis:
user input of seat: 1 operation
indexCount initialization: 1 operation
while loop iterations: n operations
while loop conditional checks: 2*n comparisons
indexCount increment: n operations
shift next SeatNode: n operations
if condition check: 1 comparison
insertion at end (O(n)): n operations
print seats (O(n)): n operations
deleteNodeAtPosition: n operations
Frequency Count: F(n) = 8n + 3
Time Complexity: O(n)
Purpose: To allow the user to book a seat based on the given a hand selected availability list. Make sure that the seat node exist in the list in the available list to be able to book the seat node to be stored in the booked list.

5.3 Cancel A Seat Function

5.3.1 cancelASeat Method

Algorithm 6 cancelASeat

Input: LinkedList, *availableSeatList*

Output: To remove the selected booked seat from the booked seat list and add the canceled seat back to the available seating list

Begin Algorithm: cancelASeat

$input(intuserSeatInput)$
$indexCount \leftarrow 0$

if $head == NULL$ **then**
  print$-$ $>$Can't be booked
  $return$
end
$currBookedSeatNode \leftarrow head$

while $((currentAvailable.next\ \ not\ \ equal\ \ nullptr)\ \ AND\ \ (currAvailableSeatNode.data\ \ not\ \ equal\ \ userSeatInput))$ **do**
  $indexCount \leftarrow indexCount + 1$
  $currBookedSeatNode \leftarrow currBookedSeatNode.next$
end
if $currBookedSeatNode.data == userSeatInput$ **then**
  $currAvailSeatNode \leftarrow availableSeatList.head$
  $indexCountAvail = 0$
  $deleteNodeAtPosition(indexCount)$

  while *(currentAvailable.next not equal nullptr) AND (userSeatInput > currAvailSeatNode.data)* **do**
    $currAvailSeatNode \leftarrow currAvailSeatNode.next$
    $indexCountAvail \leftarrow indexCountAvail + 1$
  end
  $availableSeatList.addNodeAtPosition(indexCountAvail, userSeatInput)$
else
  print $-$ $>$ uncancellable seat!
end
End Algorithm: cancelASeat

Frequency Count Analysis (Worst Case):
user input of seat: 1 operation
indexCount initialization: 1 operation
check head isn't NULL: 1 comparison
assign head to current booked node: 1 operation
while loop iterations: n operations
while loop comparisons: 2*n comparisons
inside while loop operations: 2*n operations
if block userinput comparison: 1 comparison
assign availableSeats head to current Node: 1 operation
indexCountAvail assignment: 1 operation
deleteNodeAtPosition(indexCount) (O(n)): n operations

while loop iterations: n operations
while loop comparisons: 2n comparisons
while loop operations: 2 operations
addNodeAtPosition (O(n)): n operations

Frequency Count: F(n) = 10n + 9

Time Complexity: O(n)

Purpose: To allow the user to cancel a seat from the booked list. Make sure that the seat node exist in the booked list to be able to cancel the seat node and then add the canceled seat back into the available seat in a sorted order.

5.4 Showing List of Available and Booked Seats Function

5.4.1 toPrint Method

Algorithm 7 toPrint

Input: The given single linked list
Output: Print the each seat from the full linked list
Begin Algorithm: toPrint

Initialize SeatNode *temp = head
if $head == NULL$ **then**
    print: There are no items
    return
end
while $temp\ not\ equal\ NULL$ **do**
    print the seat number and address
    $temp = temp.next$
end
Endline
End Algorithm: toPrint

Frequency Count Analysis (Worst Case):
temp initialization: 1 operation
if block head comparison: 1 comparison
while loop iterations: n operations
while loop comparisons: n comparisons
inside while loop operations: 2*n operations
End line to distinguish other print statements: 1 operation
Frequency Count: F(n) = $4n + 3$
Time Complexity: O(n)
Purpose: To print all the element and seat nodes in a given single linked list

5.5 Showing and Interacting With the Main Menu

5.5.1 mainMenuCatalog Method

Frequency Count Analysis (Worst Case):

userInput initialization: 1 operation
avalSeatingList initialization: 1 operation
bookedSeatingList initialization: 1 operation
maxSeatInRow initialization: 1 operation
for loop iterations: n operations
for loop comparisons: n comparisons
inside for loop operations: n operations
while loop iterations: n operations
while loop comparisons: n comparisons
inside while loop operations: 20 operations
inside while loop comparisons: 5 comparisons

Frequency Count: F(n) = $5n + 29$

Time Complexity: O(n)

Purpose: This is the UI feature in the terminal where the user can interact with different options seen in the the seat reservation system.

Algorithm 8 mainMenuCatalog

Input: userInput
Output: Linked list for booked seats or available seats, Booking a seat, canceling a booked seat, or exit
Begin Algorithm: mainMenuCatalog

Initialize string userInput

Initialize SingleLinkedSeatingList avalSeatingList

Initialize SingleLinkedSeatingList bookedSeatingList

int maxSeatInRow = 12 (host can choose number based on their need)

```
for i = 0 to maxSeatinRow do
    avalSeatingList.insertAtEnd(i)
end
while true do
    print: "Event Ticket Booking System: MAIN MENU"
    print: "Please Select an Option"
    print: "1. Book a Seat"
    print: "2. Cancel a Seat"
    print: "3. Show Available Seats (All Booked Seats List)"
    print: "4. Exit"
    print: "5. Show All Open Seats"
    print: "Insert Selection Number: "
    userInput ← user's menu choice
    if userInput == 1 then
        print: "You selected to book a seat"
        bookedSeatingList.bookSeat(avalSeatingList)
    end
    else if userInput == 2 then
        print: "You selected to cancel a seat"
        bookedSeatingList.cancelASeat(avalSeatingList)
    end
    else if userInput == 3 then
        print: "You selected display available seats. (booked Seats)"
        bookedSeatingList.toPrint()
    end
    else if userInput == 4 then
        print: "You selected to terminate the program"
        break from the loop
    end
    else if userInput == 5 then
        print: "You selected display all open seats"
        avalSeatingList.toPrint()
    end
    print: "You selected an invalid option. Please reselect your input"
end
```

End Algorithm: mainMenuCatalog

5.6 FlightDatabase Booking System Sorting Direct and Indirect Flights Functions

5.6.1 Find Indirect Flights NonOverlap Method (BFS Algorithm)

Frequency Count Analysis:

initialize known indirect flight variable: 1
initialize known departure map dictionary variable: 1
iterate through each flight in flight list : n
check if the departure location is in the dictionary if condition : 1
if condition is true assign the empty list to new departure map dictionary : 1
if condition is false assign the flight to new departure map dictionary based on departure key: 1
initialize a queue to store the known flight path: 1
iterate over a queue via a while loop : n
set variables for the queue : 1
check the if condition for the current location is the destination : 1
get a list of the known flight number iterating through for loop of known flights : n
check if the flight is located in the indirect flight list: 1 comparison
add the flight number to the knownIndirectFlights: 1 comparison
if condition to check if the current flight in departure mapping: 1 comparison
iterate through each flight in the departure map based on the current location: n^2 comparison
initialize a visited key: 1
check if a visited key in the visited set and add the key: 2
check if a current flight path exists: 1
assign and create flight path : 1
append the current flightpath to the queue : 1
initialize variables to check known constraints to define an in: 5
assign and create flight path and append the current flightpath to the queue : 2
return indirect flight list : 1

Frequency Count: F(n) = n^2+7n+22
Time Complexity: O(n^2)

Purpose: This function allocates the correct list for indirect flights based on a given source and destination airport that the user inputted. The BFS algorithm gets the all the valid indirect flight paths without excessive layovers and no overlapping flights. The BFS algorithm starts by creating a dictionary to represent a departure mapping system that stores the flights based on their departure locations to allow the search algo to pull from if needed. The queue is initialize to investigate all known possible flight paths in the BFS tree to reach from the departure location to the arrival location. The a valid indirect flight can be consider on to the ongoing list if the flight meets the requirements of being within a max layover of two days, prevents revisted already past flights using a set, check if the time of the next flight is valid and going to the correct path to the destination.

5.6.2 Display Indirect Flights

Frequency Count Analysis:

get all the known indirect flights: 1
set a count variable: 1
check if the indirect flight list has any flights using if condition : 1
iterate through each indirect flight path in indirect flight list : n
check if the departure location is in the dictionary if condition : 1
check if the the flight path in the indirect fight path has more than one flight to be considered an indirect flight and increment option : 2
iterate through each indirect flight with the valid indirect flight path via for each loop: n^2
use if condition to check if the indirect flights is not a direct flight: 1
print the current flight: 1

Algorithm 9 Indirect Flights BFS NonOverlap Method :

Input: The string value of the desired airport departure location, *source*, The string value of the desired airport arrival location, *dest*
Output: A list of correct indirect flights based on the set arrival and departure airports.
Begin Algorithm: Indirect Flights BFS NonOverlap Method

```
Initalize knownIndirectFlights = []
Initalize departureMap = empty dictionary
for (flight in self.givenFlights) do
    if (flight.departureLocation not in departureMap) then
        Assign departureMap[flight.departureLocation] = empty list
    end
    departureMap[flight.departureLocation].append(flight)
end
queue = deque([(source, emptylist, NULL pointer, NULL pointer)]) Initalize visitedSet = set()
while (queue is not NULL) do
    Set currentLocation, flightPath, lastEndTime, lastDate = queue.popleft()
    if (currentLocation == dest) then
        flightNumbers = [flight.flightNumber for flight in flightPath] if (flightNumbers not in knownIndirectFlights)
        then
            knownIndirectFlights.append(flightNumbers)
        end
        continue
    end
    if (currentLocation in departureMap) then
        for (nextFlight in departure_map[currentLocation]) do                    end return know
            Set visitKey = (nextFlight.flightNumber, nextFlight.timeInterval, nextFlight.date)
            if (visitKey in visitedSet) then
                continue
            end
            visitedSet.add(visitKey)
            if (flightPath does not exist) then
                Set newFlightPath = [nextFlight] queue.append((nextFlight.arrivalLocation, newFlightPath,
                nextFlight.timeInterval[1], nextFlight.date))
            else
                Set lastFlight = flightPath[-1] Set lastEndTime = lastFlight.timeInterval[1] Set nextStartTime =
                nextFlight.timeInterval[0]
                Assign isOvernight = self.isOvernight_flight(lastEndTime, nextStartTime) Assign differenceDatesTimes =
                (nextFlight.date − lastFlight.date).days
                if ((lastEndTime <= nextStartTime or isOvernight) and (lastFlight.date <= nextFlight.date) and (dif-
                ferenceDatesTimes <=2)) then
                    newFlightpath = flightPath + [nextFlight] queue.append((nextFlight.arrivalLocation, newFlight-
                    Path, nextFlight.timeInterval[1], nextFlight.date))
                end
            end
        end
    end
    End Algorithm: Indirect Flights BFS NonOverlap Method
```

Algorithm 10 Display Indirect Flights

Input: The string value of the desired airport departure location, *sourceLocation*, The string value of the desired airport arrival location, *destLocation*
Output: A printed list of correct indirect flights based on the set arrival and departure airports.
Begin Algorithm: Display Indirect Flights

Acquire allKnownIndirectFlights = findIndirectFlightsNonOverlap(sourceLocation, destLocation) Set count = 1
if *(allKnownIndirectFlights is not NULL)* **then**
  for *(curOverallFlightPath in allKnownIndirectFlights)* **do**
    if *(len(curOverallFlightPath) > 1)* **then**
      print $->$ Indirect Flight Option count Increment count by 1
    end
    curIndirectFlight in curOverallFlightPath **if** *(len(curOverallFlightPath) > 1)* **then**
      givenFlights[self.getFlight(curIndirectFlight)].displayFlight()
    end
  end
end
print("No indirect flights available.")
End Algorithm: Display Indirect Flights

Frequency Count: F(n) = n^2+n+8
Time Complexity: O(n^2)
Purpose: To output all the the valid indirect flight lists

5.7 Data Collection and Preprocessing

5.7.1 Actual Flights and Seats

To ensure users have access to a list of flights with seats to begin with, all flights and seats are initialized when the application is started by instantiating a single custom FlightDatabase object populated with Flights in the flight_objects.py script. By default when these flights are made, the seats of each are initialized as well.

In the constructor of each flight, a number of seats is provided and the first 20 percent of seats are made as First Class, second 20 percent are Business, and remaining 60 are Economy. The cost of each is made pseudorandomly using python's random module. Each seat is made by inserting a SeatNode into the available singly linked seating list (from the head) of the FlightDatabase.

To ensure users have a wide variety of flight cases available, the flight_objects.py script creates flights that cover the following cases:

- Overnight flights
- Indirect Flights (Limit of 2 days)
- Flights with the same SRC/DEST but different depart times

5.7.2 Enabling UI Data Collection

To create a working website for users, the backend needed to work with the bare logic algorithms and produce JSON lists and objects for the HTML and CSS frontend to properly display the results.

To ensure JSON conversions of our flightbooking system, methods were added to each class to create a JSON-able representation of the instance. For the SinglyLinkedSeatingList, a function $as\_list()$ is made to iterate through each node from the head, and create a Python List object representation. For Seats, a function $as\_dict()$ is made in which a key-value object is made that contains the seats number, status, and cost. Each flight's $as\_dict()$ function presents a key-value object containing the number, departure location, arrival location, date, and duration. Indirect Flight objects have an $as\_list()$ function in which all inner Flights are shown using their dictionary representations (from $as\_dict()$) in the order of the path searched to create them.

We verified these results were working and accessible through HTTP requests to the Flask Backend with Postman.

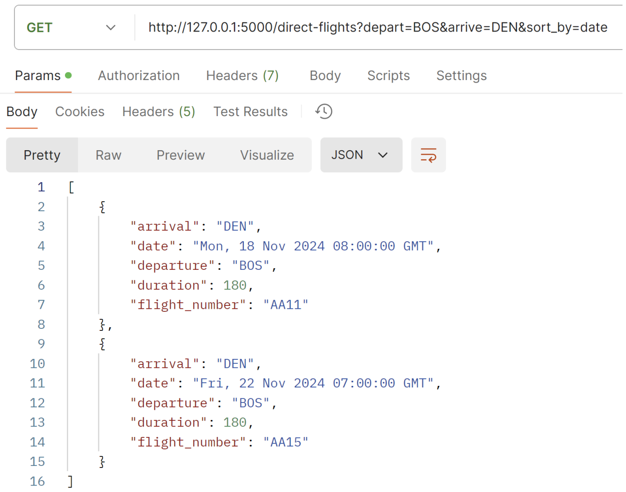

Figure 5: Sample Backend Return of List of JSON Objects from a GET flights request.

6 Results

Present the results, including key performance metrics and visualizations (e.g., tables, charts) that illustrate the effectiveness of the algorithms.

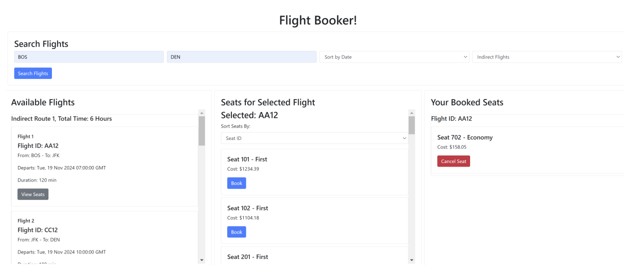

Figure 6: View Indirect Flights and Seats from BOS to DEN

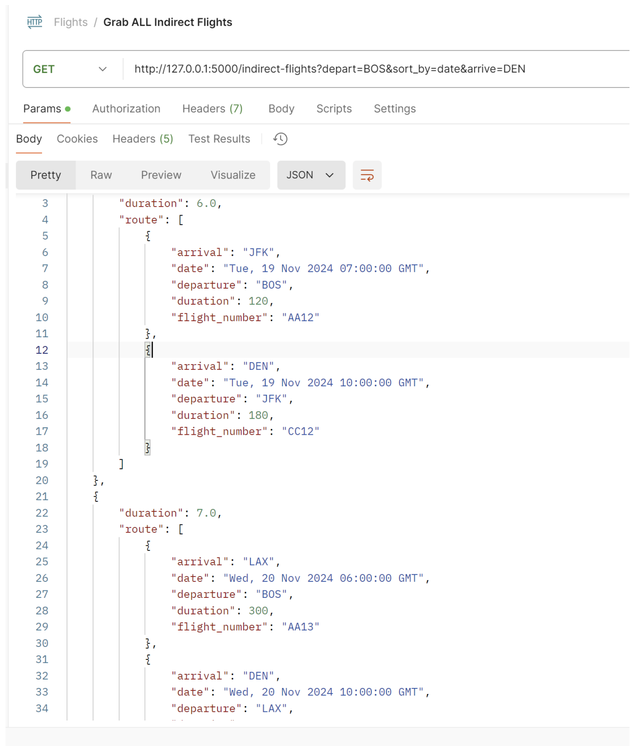

Figure 7: Sort Indirect Flights by Date

The working results of filtering and sorting indirect flights can be seen in the image above. Whereby the indirect flights view accurately shows all the indirect routes within 2 days from BOS to DEN using the BFS search algorithm detailed above, bubble-sorted by departure time. We can similarly sort using the duration correctly as seen below, where now the second route shown is 6 hours.

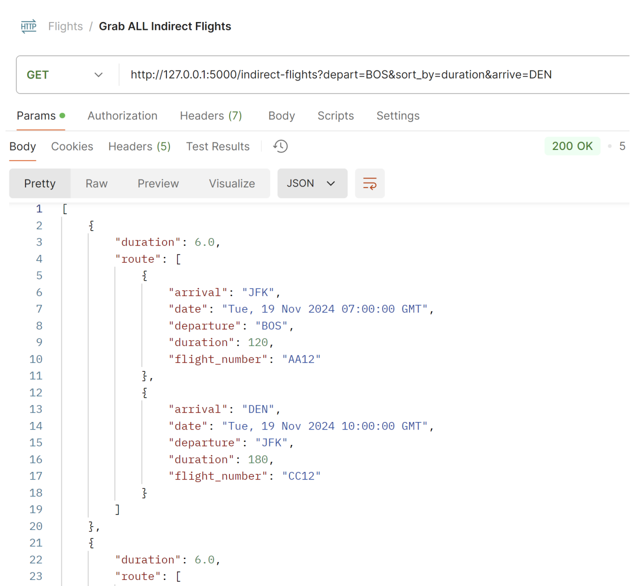

Figure 8: Sorted by Duration Indirect Flights

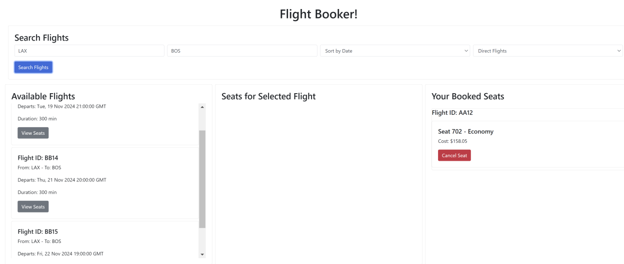

Figure 9: View Direct Flights from LAX to BOS

7 Discussion

Interpret the results and discuss their implications. Compare the findings with the initial objectives, and highlight any discrepancies or unexpected outcomes.

8 Conclusion

Summarize the key findings, discuss limitations, and suggest areas for future improvement.

9 References

[1] https://www.scaler.com/topics/breadth-first-search-python/

A Appendix A: Code

Include relevant code sections here.

B Appendix B: Additional Figures

Provide any additional figures or tables that support the analysis.