Project 4: Air Anarchy

Time due: 11 PM, ~~Tuesday, March 18~~ Wednesday, March 19

Introduction

Imagine you're an adventurous traveler trying to plan the perfect journey from your local airport to an exotic destination, but instead of a trusty travel agent, you've got an algorithm-powered, flight-finding wizard in your corner. Welcome to the world of automated travel planning, where algorithms do the heavy lifting and you just sit back, sip your coffee, and marvel at how quickly your itinerary materializes. In this project, you'll build your own virtual travel guru that not only understands flight schedules but also pieces together an efficient route, avoiding long layovers, missed connections, and the dreaded middle seat (okay, maybe not that last one). Get ready to code your way to the skies!

We're going to give you a bunch of authentic flight data, harvested from all domestic airlines in January, which you're going to use to produce efficient itineraries. For example, here's an itinerary that takes you from Hawthorne Airport (in Hawthorne, CA) to Platinum Airport (in Platinum, AK):

```
Source: HHR, Destination: PTU, Total Duration: 19.6111 hours
Arriving at source airport at: 2025-01-07 21:13 UTC (1736284400)
Wait time at initial airport: 0.277778 hours
Flights:
  HHR -> CLD, Airline: Desert Jet, Departure: 2025-01-07 21:30 UTC (1736285400), Arrival:
2025-01-07 22:18 UTC (1736288280), Duration: 0.8 hours
  Layover: 1.7 hours
  CLD -> COS, Airline: Netjets Aviation, Departure: 2025-01-08 00:00 UTC (1736294400),
Arrival: 2025-01-08 01:07 UTC (1736298420), Duration: 1.11667 hours
  Layover: 1.46667 hours
  COS -> DEN, Airline: Air Canada, Departure: 2025-01-08 02:35 UTC (1736303700), Arrival:
2025-01-08 03:39 UTC (1736307540), Duration: 1.06667 hours
  Layover: 1.01667 hours
  DEN -> SEA, Airline: Southwest Airlines, Departure: 2025-01-08 04:40 UTC (1736311200),
Arrival: 2025-01-08 06:40 UTC (1736318400), Duration: 2 hours
  Layover: 1.31667 hours
  SEA -> ANC, Airline: Alaska Airlines, Departure: 2025-01-08 07:59 UTC (1736323140), Arrival:
2025-01-08 10:46 UTC (1736333160), Duration: 2.78333 hours
  Layover: 2.73333 hours
  ANC -> BET, Airline: Everts Air Cargo, Departure: 2025-01-08 13:30 UTC (1736343000),
Arrival: 2025-01-08 14:18 UTC (1736345880), Duration: 0.8 hours
  Layover: 1.95 hours
  BET -> PTU, Airline: Xinjiang Skylink General Aviation, Departure: 2025-01-08 16:15 UTC
(1736352900), Arrival: 2025-01-08 16:50 UTC (1736355000), Duration: 0.583333 hours
Arriving at destination airport at: 2025-01-08 16:50 UTC (1736355000)
```

To successfully complete this project, **you will write three classes**:

1. **FlightManager** (derived from our provided `FlightManagerBase` class) – responsible for loading and providing flight information from real data.

2. **TravelPlanner** (derived from our provided `TravelPlannerBase` class) – responsible for building an actual flight itinerary (a sequence of connecting flights) given information such as the starting and ending airports as well as other criteria.
3. **BSTSet** – a class template implementing an ordered set via a binary search tree, complete with iterator support.

Your goal is to produce fully functional implementations of these three classes meeting the specifications listed below. If you do so, your program will work with our provided code and you can use them together to generate optimal flight routes from virtually any US airport to any other!

Background Information

Below is a high-level overview of the components in a flight planning system. You will implement some of these components.

1. **Airline Database and Data**
   We will provide you with a fully-implemented class called `AirportDB` that lets you calculate approximate distances between different airports. Your code can use it to determine the distance between two airports when necessary.
2. **Flight Manager**
   This component is responsible for keeping track of available flights out of each airport. In practice, the flight manager loads flight schedule data (e.g., airline names, flight numbers, departure times, durations) from a text file. It then makes these flights easily searchable by criteria such as origin airport and time window. You will implement a `FlightManager` class that is derived from our provided `FlightManagerBase` class.
3. **Travel Planner**
   The travel planner takes a source airport, a destination airport, and a desired start time, then creates an itinerary—a list of flights leading from the origin to the destination, subject to constraints such as minimum allowable connection time and maximum layover time. You will implement a `TravelPlanner` class that is derived from our provided `TravelPlannerBase`.
4. **Flight Schedule Data**
   Real-world flight data includes airlines, flight numbers, airport codes (e.g., LAX, SJC), departure and arrival times, flight durations, etc. We will give you this data as a file (all_flights.txt). Your flight manager will parse this file and store the information about the flights for efficient retrieval. The travel planner will then use the flight manager's search functions to construct itineraries.

Data File Format

We are providing you with a file that has all the flight schedule information needed by your `FlightManager` class (all_flights.txt). Each line in this text file describes one flight, with the following fields in this order, separated by commas:

1. **Airline Name** – The name of the airline offering the flight (e.g., Southwest).
2. **Flight#** – The airline's flight number (e.g., 3125).
3. **Source Airport** – The three-letter airport code where the flight departs.
4. **Destination Airport** – The three-letter airport code where the flight arrives.
5. **Departure Time (UTC)** – A UNIX timestamp representing when the flight departs, as the number of seconds that have elapsed since Jan 1, 1970, 00:00:00 UTC.
6. **Arrival Time (UTC)** – A UNIX timestamp representing when the flight arrives, in seconds.
7. **Duration (sec)** – Flight duration in seconds. This will always be arrival time minus departure time.

A sample line might look like this:

```
Korean Air,3656,ABE,ATL,1736386200,1736394960,8760
```

This corresponds to:

- Airline: `Korean Air`
- Flight Number: `3656`
- Departing Airport: `ABE`
- Arriving Airport: `ATL`
- Departure Time: `1736386200`
- Arrival Time: `1736394960`
- Duration: `8760` seconds

If you'd like to determine a human readable equivalent time for a UNIX timestamp, you can go to https://www.unixtimestamp.com/ . Alternatively, for the timestamp 173638200, say, you could run in a Mac Terminal window the command `date -r 173638200` or on a Linux command line the command `date -u -d @173638200` (notice the `@`).

What We Provide

The following structs, classes, and main routine are **fully provided** to you in our provided.h and provided.cpp files:

1. **`FlightSegment`**
    - A FlightSegment is a struct that represents one leg of a journey, describing a specific flight from one airport to another. It contains the following data:
        - Airline Name – A string indicating the airline offering the flight (e.g., "Korean Air").
        - Flight Number – An integer representing the flight number assigned by the airline.
        - Source Airport – A string containing the three-letter airport code where the flight departs.
        - Destination Airport – A string containing the three-letter airport code where the flight arrives.
        - Departure Time – An integer (UNIX timestamp in seconds) indicating when the flight is scheduled to depart.
        - Duration (Seconds) – An integer number of seconds indication the duration of the flight.
    - You will not need to, and MUST NOT, modify its implementation.

2. **`Itinerary`**
    - An Itinerary is a struct that brings together a sequence of one or more FlightSegment objects to form a complete route from a source airport to a destination airport. Along with storing the list of individual flights, it keeps note of:
        - Source Airport – The starting airport for the overall journey.
        - Destination Airport – The final airport where the overall journey ends.
        - Total Duration – An integer representing the total time (in seconds) from the initial flight departure to the final flight arrival.
    - You will not need to, and MUST NOT, modify its implementation.

3. **`AirportDB`**
    - Our provided airport database allows distance queries between two airports (e.g., how far LAX is from SJC in miles).
    - Public methods of interest:
        - `bool get_distance(std::string source_airport, std::string destination_airport, double& distance) const;`
            - This method determines the distance in miles between the source and destination airports.
            - Hint: This may be useful if you are implementing the A* algorithm to compute a lower bound on the flight time between two airports!
    - You will not need to, and MUST NOT, modify its implementation.

4. **`FlightManagerBase`**
    - This is an abstract base class that your `FlightManager` must be derived from.

- You must implement **all** pure virtual methods from this class in your derived `FlightManager` class.
- You will not need to, and MUST NOT, modify `FlightManagerBase`'s implementation.

5. **`TravelPlannerBase`**
   - This is an abstract base class that your `TravelPlanner` must derive from.
   - You must implement **all** pure virtual methods in your derived `TravelPlanner` class.
   - The TravelPlannerBase class lets you specify travel constraints that must be considered in your solution.
     - `void set_max_duration(int max_duration);`
       `int get_max_duration() const;`
       - Sets/gets the maximum total travel time allowed, in seconds.
     - `void set_max_layover(int max_layover);`
       `int get_max_layover() const;`
       - Sets/gets the maximum layover time allowed, in seconds.
     - `void set_min_connection_time(int min_connection_time);`
       `int get_min_connection_time() const;`
       - Sets/gets the minimum allowable connection time, in seconds.
     - `void add_preferred_airline(std::string airline);`
       - Adds a preferred airline(s) to a collection of airlines that are the only ones that an itinerary being generated may use.
   - You will not need to, and MUST NOT, modify `TravelPlannerBase`'s implementation.

6. **`main()`**
   - We provide a **main** function and several helper functions to help you test your implementation.
   - You prepare a file with a trip you want to plan (source and destination airports, time and airline constraints, etc.)
   - When you run the program built with our main function, you give the program the name of that trip file, either on the command line or interactively.
   - If a valid itinerary is found, our **main** function displays a summary of your journey, including each flight's departure and arrival times, total travel duration, and layover lengths.
   - If no valid itinerary fits your preferences and constraints, the program simply notifies you that there are no matching travel options. This makes it easy to quickly see whether a suitable travel plan is available and to review all necessary flight details in one place.
   - You will not be submitting the main.cpp file so you may change it however you like to make your development and testing easier.

The base classes we provide handle some internal management for you; your task is to flesh out the required logic in the derived classes (implementing all virtual functions).

Classes You Must Implement

You must provide implementations for the following three classes.

BSTSet<T>

You must create a class template named `BSTSet` that maintains an **ordered set** of **unique** elements of type `T`. Your class **must** use a binary search tree for its implementation, with the ordering determined by the < operator for the type T (just as it is for std::set). Your class must provide:

- **Public Methods**
  - **Constructor & Destructor**
    - `BSTSet()` – Initialize an empty set.
    - `~BSTSet()` – Release any internal resources.
  - **`void insert(const T& value)`**
    - Inserts `value` into the set if it is not already present.
    - If the value is already present, you MUST replace the existing value with the new value.
    - For a set of N items, this method must run in O(log N) time in the average case.
  - **`SetIterator find(const T& value) const`**
    - Returns an iterator pointing to an element that is equal to `value`, or an invalid iterator (defined below) if none is found.
    - For a set of N items, this method must run in O(log N) time in the average case.
  - **`SetIterator find_first_not_smaller(const T& value) const`**
    - Returns an iterator pointing to the smallest element in the set that is **not smaller** than `value` (i.e., that is at least `value`). If all elements in the set are smaller than `value` or if there are no elements in the set, return an invalid iterator.
    - For a set of N items, this method must run in O(log N) time in the average case.
- **Nested Iterator Class** (`BSTSet<T>::SetIterator`)
  - You must have a public nested class called `SetIterator` in your `BSTSet` class.
  - The `SetIterator` class must implement (in addition to a destructor or any public constructor(s) you choose to implement) this public method:

        - **`const T* get_and_advance()`** – This method returns a pointer to the current element that the iterator points at and advances the iterator to point to the next larger element in the set, or `nullptr` if there is no next larger element.
        - For a set of N items, this method must run in O(log N) time in the average case.
    - We define an *invalid iterator* to be one for which calling **`get_and_advance()`** on it returns `nullptr`.
    - If `iter` is an iterator pointing to an element of a BSTSet of N items of some type X, this code must run in O(N) time:

```
const X* p;
while ((p = it.get_and_advance()) != nullptr)
    std::cout << '#';
```

Other requirements:

- Your `BSTSet<T>` class must not use any STL containers.
- Your class must NOT have any public data members and must NOT have any public methods or public nested structs/classes other than those mentioned. Private members and structs/classes are allowed, of course.

If we're inserting N items into a BSTSet, there are N! different orders in which they might be inserted. The big-O requirements are for the average case assuming all orderings of insertions are equally probable. A small percentage of orderings cause a worse time complexity than is specified in a requirement, but you need not worry about those orderings. The implication of this is that you are not required to try to keep the binary search tree balanced.

Here is an example of how the class can be used:

```
BSTSet<int> mySet;
mySet.insert(5);
mySet.insert(2);
mySet.insert(8);
mySet.insert(7);
mySet.insert(5); // Duplicate insert (replaces old 5 with new 5)

  // Now let's find the first element not smaller than 4
BSTSet<int>::SetIterator it = mySet.find_first_not_smaller(4);
const int* p;
while ((p = it.get_and_advance()) != nullptr)
    std::cout << *p << ' ';
std::cout << std::endl;
// Output is 5 7 8
// It would also have been 5 7 8 if instead of 4, the argument
// passed to find_first_not_smaller had been 5 or 3
```

```
it = mySet.find(4);
if (it.get_and_advance() == nullptr)
    std::cout << "No 4 in the set" << std::endl;
```

`FlightManager` (Derived from `FlightManagerBase`)

Your `FlightManager` class must implement the following methods:

- **Construction & Destruction**
  - **`FlightManager()`**
  - **`~FlightManager()`**
  - You do not have to declare and implement these if the compiler-generated constructor and destructor do the right things.
- **File Loading**
  - **`bool load_flight_data(std::string filename)`**
  - Reads and parses the provided flight data file.
  - Each valid input line is converted into a `FlightSegment` and stored internally.
  - Returns `true` if the file is loaded successfully, otherwise `false`.
  - There is no big-O requirement for loading.
- **Flight Lookup**
  - **`std::vector<FlightSegment> find_flights(std::string source_airport, int start_time, int end_time) const`**
  - Given a departure airport (e.g., SJC), return a vector of all flights whose **departure time** is between `start_time` (inclusive) and `end_time` (exclusive).
  - The returned flights must be ordered by ascending departure time.
  - Return an empty vector if there are no flights from the specified airport during the specified time window, or if the airport is unknown.
  - If there are N airports, each with F flights leaving that airport that might match the search criteria, then the big-O of find_flights must be O(F). Notice that this is independent of N.

Other requirements:

- Your **`FlightManager`** class must use the **`BSTSet`** class in some meaningful way for full credit (e.g., to hold all outbound flights from each airport).
- You may also use the following STL classes to implement your `FlightManager` class: map, unordered_map, vector, list.
- You must NOT use any other STL containers in your FlightManager class.
- If you need to define custom comparison operators, you may do so in your fm.h or fm.cpp file(s).

`TravelPlanner` (Derived from `TravelPlannerBase`)

Your `TravelPlanner` class must implement the following methods:

- **Construction**
  - **`TravelPlanner(const FlightManagerBase& flight_manager, const AirportDB& airport_db)`**
  - Constructs a travel planner object that will consult the flight_manager and airport_db when planning a trip.
- **`Destruction`**
  - **`~TravelPlanner()`**
  - You do not have to declare and implement a destructor if the compiler-generated destructor does the right thing.
- **Specifying Preferred Airlines**
  - **`void add_preferred_airline(std::string airline)`**
  - Designates the specified airline (e.g., United Airlines) as one of the “preferred” airlines.
- **Core Trip Planning**
  - **`bool plan_travel(std::string source_airport, std::string destination_airport, int start_time, Itinerary& itinerary) const`**
  - Generate a sequence of flights taking you from `source_airport` to `destination_airport`.
  - Your goal is to generate an itinerary with the shortest travel time possible from `start_time`, the time that the traveler begins at the source airport, until arrival at the destination airport. You may find the A* algorithm helpful for this!
  - When generating an itinerary, you must account for:
    - **Initial Flight Time -** The initial flight must depart no earlier than `start_time` and no later than `start_time` + `get_max_layover()`.
    - **Minimum Connection Time** – The layover between any two consecutive flights must be at least `get_min_connection_time()`, so if a flight arrives at an airport at time t, then you must consider only departing flights that leave on or after t + `get_min_connection_time()`. This value is in seconds.
    - **Maximum Layover Time** – The layover between any two consecutive flights cannot exceed `get_max_layover()`, so if a flight arrives at time t, then you must only consider departing flights that leave before t + `get_max_layover()`. The concept of maximum layover time includes the concept of time spent waiting at the source airport for the first flight, so no flights after `start_time` + `get_max_layover()` may be considered for the first flight out of the source airport. This value is in seconds.
    - **Maximum Duration** – The total travel time from beginning at the source airport at `start_time` to the time of arrival at the destination airport,

including layover times, must not exceed `get_max_duration()`. This value is in seconds.
        - **Preferred Airlines** – If **at least one** airline has been designated as preferred, then only flights from airline(s) so designated may be included in your itinerary. An itinerary might include flights from more than one preferred airline (e.g., Southwest, then United, then Southwest). If **no** airlines have been designated as preferred, then flights from **all** airlines are eligible for your itinerary.
    - If a valid route can be created that routes from the source airport to the destination airport, store that valid route in `itinerary`, populate all of that `itinerary`'s other fields, and return `true`. If no valid route is found, return `false`.
    - When producing the itinerary:
        - The flights in the resulting itinerary must be in chronological order (each flight's departure must be after the previous flight's arrival).
        - The itinerary's source airport must match the first flight's origin; the itinerary's destination airport must match the last flight's destination.
        - When setting the total duration of travel in the itinerary, you must include not just time in the air, but also all waiting time at all airports, including the first — this should be the total time from `start_time` until the time of arrival at the destination airport.
    - For this project, you are not required to minimize the number of flights or connections, only the overall travel duration.
    - While it's difficult to place a big-O requirement on this function since there are many possible implementations, a correct solution given the provided data files should run in well under 10 seconds for almost all queries.

Other requirements:

- Your `TravelPlanner` class may use any STL data structures you like.

How to Implement a Route-finding Algorithm

Right now you're probably thinking: "It must be really complicated to compute an optimal route from one airport to another…" But in reality, you can implement an airport-routing algorithm in just a few hundred lines of code (or less!) by using a breadth-first search (BFS) queue-based approach or a well-known algorithm like A*. If you're comfortable with A*, that's fantastic; it can give you a true optimal solution. But if that seems a bit daunting right now, you can still adapt the queue-based "maze searching" approach we learned in class - just treat each airport as a square in a maze and each flight connection as a connection to an adjacent square.

Of course, there are some differences between a simple maze-search problem and airport routing:

1. **Integer vs. Airport Nodes**

In a typical maze-search, you enqueue integer-valued (x,y) coordinates. For an airport-routing task, you'll be enqueuing "airport codes" like "LAX"

2. **Marking Visited Airports**
   In maze-search, you used a 2D array to mark squares you've visited. In an airport-routing problem, there's no 2D grid for airports, so you need another mechanism to keep track of which airports you've visited.
3. **Finding Adjacent Airports**
   In a maze, you figure out adjacent squares via simple arithmetic (e.g., (x−1,y), (x+1,y), (x,y−1), (x,y+1)). In an airport-routing system, you have to rely on our provided flight database to find out which airports have direct flights from your current airport.
4. **Returning a Full Path**
   In a maze-search, you might just return true or false to indicate if the maze is solvable. But in an airport-routing scenario, you need to return the full list of airport connections (the entire route). That means storing and reconstructing the path you used to get from the origin airport to the destination airport, not just confirming that it exists.

Fortunately, with just a few modifications to the queue-based maze-search approach, you can adapt it to airport travel! You still need to figure out how to reconstruct the path of flights after your routing completes. In the maze-search approach, you simply returned true or false. But for airport routing, you have to actually list each airport in your route. One good way to do this is to keep a copy of your route segments (i.e., airlines and flight numbers, departure/arrival times, etc.) thus far within the data structure that you add to your queue.

Like the maze-search algorithm, your BFS must ensure you visit airports efficiently. For example, if you've already flown through SJC (San Jose) on your route to SEA (Seattle), then you don't want to fly back through SJC to finish your route to DEN (Denver).

What does this look like conceptually?

Well, BFS starts at your origin airport, enqueues all airports that can be reached directly from that origin (via flights that satisfy the user's layover and preferred airline requirements), then enqueues airports that can be reached from those, and so on. Eventually, it either finds the destination, or exhausts all possibilities and concludes no route is possible. If it does find the destination, you can output the segments for each leg of the trip.

Will BFS always find the best route?

Not necessarily. BFS finds the route with the fewest "hops" (connections). But that might not be the route with the shortest travel time or distance! Imagine airports A, B, and C on a line:

- A is at position 0
- B is at position 100 (100 miles from A)
- C is at position 2000 (2000 miles from A, 1900 miles from B)

Suppose there are two ways to get from A to B:

1. A route of **4 quick hops** with minimal layovers through other airports that eventually total 150 miles between A and B.
2. A single long-haul flight from A to C (2000 miles), then a second flight back from C to B (1900 miles).

A basic BFS in terms of “number of flights” would pick the second route (because it has only 2 flights vs. 4). But that’s obviously not the shortest actual distance or time!

If you want the truly optimal path (e.g., in travel time), you’ll need a more advanced algorithm. A* is one option: it uses a distance-based heuristic to guide the search so that it prefers airports closer to the destination. For instance, if you’re currently at airport **X** and you see possible flights to **Y**, **Z**, and **Q**, you can estimate each airport’s “distance” to your final destination and insert them in a priority queue ranked by that distance. This doesn’t just rely on “fewest hops” but tries to find the route with the least travel cost (time, distance, or whatever metric you choose). Ask ChatGPT for more information on how A* works!

Our Provided Driver Program

We provide a driver program (in main.cpp) which you can use to test your implementation. Once you compile your program, e.g., to proj4.exe, here's how you run it on the command line:

C:/PROJ4> proj4.exe params.txt

Where params.txt (you can name this file anything you like) contains the parameters of the travel that you must satisfy. Here's an example version of the parameter file:

```
all_flights.txt
HHR
PTU
1736284400
24
1
4
```

Where:

1. The first line specifies the data file containing the flight data.
2. The second line specifies the starting airport (e.g., HHR). It must be uppercase.
3. The third line specifies the destination airport (e.g., PTU). It must be uppercase.
4. The fourth line is a UNIX timestamp that specifies the start time of travel, i.e., the time at which the traveler will arrive at the source airport and be ready to board a flight. Note that this time might be minutes or hours before their first flight is ready to take off.
5. The fifth line is the total number of hours that the travel may take. If travel exceeds this time, then no itinerary may be generated.
6. The sixth line specifies the minimum number of hours required to make a connection between arriving on one flight and departing on the next flight (i.e., to walk between gates at the airport)

7. The seventh line specifies the maximum number of hours that the traveller is willing to wait to make a connection to their next flight, and also the maximum time the traveller is willing to wait for an outbound flight at the starting airport after their initial arrival.
8. The remaining lines are the case-sensitive names of the preferred airlines, one per line. In this example, there are no remaining lines, since no airlines are preferred.

Feel free to create your own text files to test various travel scenarios as you build your solution.

The output of our test program will look something like this for a correct implementation (though your route may be different than ours):

```
Source: HHR, Destination: PTU, Total Duration: 19.6111 hours
Arriving at source airport at: 2025-01-07 21:13 UTC (1736284400)
Wait time at initial airport: 0.277778 hours
Flights:
  HHR -> CLD, Airline: Desert Jet, Departure: 2025-01-07 21:30 UTC (1736285400), Arrival:
2025-01-07 22:18 UTC (1736288280), Duration: 0.8 hours
  Layover: 1.7 hours
  CLD -> COS, Airline: Netjets Aviation, Departure: 2025-01-08 00:00 UTC (1736294400),
Arrival: 2025-01-08 01:07 UTC (1736298420), Duration: 1.11667 hours
  Layover: 1.46667 hours
  COS -> DEN, Airline: Air Canada, Departure: 2025-01-08 02:35 UTC (1736303700), Arrival:
2025-01-08 03:39 UTC (1736307540), Duration: 1.06667 hours
  Layover: 1.01667 hours
  DEN -> SEA, Airline: Southwest Airlines, Departure: 2025-01-08 04:40 UTC (1736311200),
Arrival: 2025-01-08 06:40 UTC (1736318400), Duration: 2 hours
  Layover: 1.31667 hours
  SEA -> ANC, Airline: Alaska Airlines, Departure: 2025-01-08 07:59 UTC (1736323140), Arrival:
2025-01-08 10:46 UTC (1736333160), Duration: 2.78333 hours
  Layover: 2.73333 hours
  ANC -> BET, Airline: Everts Air Cargo, Departure: 2025-01-08 13:30 UTC (1736343000),
Arrival: 2025-01-08 14:18 UTC (1736345880), Duration: 0.8 hours
  Layover: 1.95 hours
  BET -> PTU, Airline: Xinjiang Skylink General Aviation, Departure: 2025-01-08 16:15 UTC
(1736352900), Arrival: 2025-01-08 16:50 UTC (1736355000), Duration: 0.583333 hours
Arriving at destination airport at: 2025-01-08 16:50 UTC (1736355000)
```

Project Requirements and Other Thoughts

Make sure to read this entire section before beginning your project!

1. In Visual C++, change your project from UNICODE to Multi Byte Character set, by going to Project / Properties / Configuration Properties / Advanced (or General) / Character Set. (This might not be necessary in Visual Studio 2022.)
2. The entire project can be completed in under 400 lines of C++ code beyond what we've already written for you, so if your program is getting larger than this, talk to a TA – you're probably doing something wrong.
3. Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. How will you use these data structures?

Plan before you program!

4. Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
5. You must not modify any of the code in the files we provide you that you will not turn in; since you're not turning them in, we will not see those changes. We will incorporate the required files that you turn in into a project with special test versions of the other files.
6. Your classes (*FlightManager*, *TravelPlanner*) must **never *directly*** refer to your other derived classes. They MUST refer to our provided base classes instead:

INCORRECT:

```
class TravelPlanner
{
  ...
private:
  const FlightManager& m_fm; // BAD!
  ...
};
```

CORRECT:

```
class TravelPlanner
{
 ...
private:
  const FlightManagerBase& m_fm; // GOOD!
 ...
};
```

7. Make sure to implement and test each class independently of the others that depend on it. Once you get the simplest class coded, get it to compile and test it with a number of different unit tests. Only once you have your first class working should you advance to the next class.
8. You may use only those STL containers that are explicitly permitted by this spec.
9. You may use STL algorithms, such as *min()*. If you do, make sure to include <algorithm>.
10. Let *Whatever* represent *FlightManager* and *TravelPlanner*. Subject to the constraints we imposed (e.g., no changes to the public interface of the *Whatever* class, no mention of *Whatever* in any file other than *Whatever*.cpp, no use of certain STL containers in your implementation), you're otherwise pretty much free to do whatever you want in *Whatever*.cpp as long as it's related to the support of only the *Whatever* implementation; for example, you may add non-member support functions (e.g., a custom comparison function for *FlightSegment*s).

If you don't think you'll be able to finish this project, then take some shortcuts. For example, if you can't get your *BSTSet* class working, create a simple version of your *BSTSet* class based on the STL set, and go back to fully implementing your *BSTSet* class later.

You can still get a good amount of partial credit if you implement most of the project. Why? Because if you fail to complete a class (e.g., *FlightManager*), we will provide a correct version of

that class and test it with the rest of your program (by changing our *TravelPlanner* class to use our correct, finished version of the class instead of your version). If you implemented the rest of the program properly, it should work perfectly with our version of the *FlightManager* class and we can give you credit for those parts of the project you completed (This is why we're using *derived* classes and base classes).

But whatever you do, make sure that ALL CODE THAT YOU TURN IN BUILDS without errors under both g32 and either Visual Studio or clang++!

What to Turn In

You must turn in **five to ten** files. These five are required and these are optional:

bstset.h Contains your BSTSet class template implementation
fm.h/fm.cpp Contains your FlightManager class implementation
tp.h/tp.cpp Contains your TravelPlanner class implementation
report.txt or report.docx Contains your report
support.h/support.cpp You may define support constants/classes/functions in these support files

Use support.h and support.cpp if there are constants, class declarations, functions, and the like that you want to use in *more than one* of the other files. (If you wanted to use something in only one file, then just put it in that file.) Use support.cpp only if you declare things in support.h that you want to implement in support.cpp.

You are to define your class declarations and all member function implementations directly within the specified .h and .cpp files. You may add any #includes or constants you like to these files. You may also add support functions for these classes if you like (e.g., *operator*<). We will not be grading the commenting of your code.

You must submit a brief (you're welcome!) report that presents the big-O for the average case of *FlightManager::load_flight_data*() and *FlightManager::find_flights*(). Be sure to make clear the meaning of the variables in your big-O expressions, e.g., "If the *FlightManager* holds N airports, and each airport is associated with F flights on average, *find_flights()* is $O(F^2 \log N)$."

Grading

- 90% of your grade will be assigned based on the correctness of your solution
- 5% of your grade will be based on the optimality of your itineraries (shorter travel time is better)
- 5% of your grade will be based on your report

Optimality Grading (5%)

Your travel planning algorithm does not need to find an optimal solution to get most of the credit for Project 4; it need only return a valid solution. That said, 5% of the points for Project 4

will be awarded based on the optimality of the routes your algorithm finds.

We'll run several tests for optimality. You'll earn credit for an optimality test if your route for that test case if it is a valid route that takes less time than 110% of our optimal least-total-time solution for that test case and takes less than 10 times as long to compute as ours or 3 seconds, whichever is longer.

Good luck! We hope you enjoy your flight!