Project Report: Gator Air Traffic Slot Scheduler

Course: COP 5536 Fall 2025
Project: Programming Project - Air Traffic Scheduler
Name: Paras Mittal
UFID: 60617951
Email: paras.mittal@ufl.edu

---

1. Project Overview

This project implements a comprehensive airport runway scheduling system that manages flight requests across multiple runways. The system handles flight submissions, cancellations, reprioritization, runway additions, ground holds, and various query operations while maintaining optimal scheduling through a greedy algorithm.

---

2. System Architecture

2.1 Core Components

2.1.1 Flight Structure

```
struct Flight {
    int flightID;        // Unique identifier
    int airlineID;       // Airline identifier
    int submitTime;      // Request submission time
    int priority;        // Urgency level (higher = more urgent)
    int duration;        // Runway usage time
    int runwayID;        // Assigned runway (-1 if pending)
    int startTime;       // Scheduled start time
    int ETA;            // Estimated completion time
    FlightState state;   // Current state in lifecycle
};
```

2.1.2 Flight Lifecycle States

- **PENDING**: Submitted but not yet scheduled
- **SCHEDULED**: Assigned runway and time slot
- **IN_PROGRESS**: Currently using runway (non-preemptive)
- **COMPLETED**: Finished and removed from system

2.2 Data Structure Design

2.2.1 Pending Flights Queue - Max Pairing Heap

Purpose: Efficiently maintain and retrieve highest-priority pending flight

Design Rationale:

- Two-pass pairing heap provides O(log n) amortized operations
- Max heap ordering on (priority, -submitTime, -flightID)
- Supports increase-key for priority updates
- Maintains handles for O(log n) arbitrary deletions

Key Operations:

```
 PairingNode* push(Flight* flight);           // O(1) amortized
Flight* top();                               // O(1)
void pop();                                  // O(log n) amortized
void increaseKey(PairingNode* node, int p);  // O(log n) amortized
void remove(PairingNode* node);              // O(log n) amortized
```

Implementation Highlights:

- Two-pass merging in pop operation for optimal performance
- Recursive tree structure with leftChild and nextSibling pointers
- Handle storage enables direct node access

2.2.2 Runway Pool - Binary Min Heap

Purpose: Track runway availability and assign earliest free runway

Design Rationale:

- Min heap ordered by (nextFreeTime, runwayID)
- Always provides earliest available runway in O(log k) time
- Dynamic updates as runways are assigned and freed

Structure:

```
struct Runway {
    int nextFreeTime;  // When runway becomes available
    int runwayID;      // Runway identifier
};
```

2.2.3 Timetable - Binary Min Heap

Purpose: Efficiently settle flight completions as time advances

Design Rationale:

- Min heap ordered by (ETA, flightID)
- Enables efficient extraction of all completed flights
- Sorted output naturally from heap structure

Structure:

```
struct TimetableEntry {
    int ETA;       // Completion time
    int flightID;  // Flight identifier
    int runwayID;  // Assigned runway
};
```

2.2.4 Hash Tables

Active Flights Map:

- Maps flightID → Flight data
- O(1) lookup for any operation requiring flight details
- Central repository for all active flights

Airline Index:

- Maps airlineID → set of flightIDs
- Enables efficient ground hold operations
- O(1) access to airline's flights

Handles Map:

- Maps flightID → PairingNode pointer
- Enables O(log n) updates/deletions in pairing heap
- Critical for reprioritize and cancel operations

---

3. Core Algorithms

3.1 Two-Phase Update Protocol

Every time operation occurs, the system performs a two-phase update:

Phase 1: Settle Completions

```
1. Extract all flights with ETA ≤ currentTime
2. Sort by (ETA, flightID)
3. Print landing messages
4. Remove from all data structures
```

Time Complexity: O(c log c) where c = number of completions

Promotion Step

```
Mark all flights with startTime ≤ currentTime as IN_PROGRESS
```

This enforces the non-preemptive constraint.

Phase 2: Reschedule Unsatisfied Flights

```
1. Identify unsatisfied flights (PENDING or SCHEDULED-not-started)
2. Save old ETAs for change detection
3. Reset unsatisfied flights to PENDING
4. Rebuild runway pool with current availability
5. Rebuild pending heap with all unsatisfied flights
6. Apply greedy scheduling algorithm
7. Compare new ETAs with old values
8. Print "Updated ETAs" if changes detected
```

Time Complexity: O(n log n + n log k) where n = flights, k = runways

3.2 Greedy Scheduling Algorithm

The scheduler assigns flights using a deterministic greedy policy:

```
while pending heap is not empty:
    flight = pop highest priority flight
    runway = pop earliest available runway

    startTime = max(currentTime, runway.nextFreeTime)
    ETA = startTime + flight.duration

    assign flight to runway
    runway.nextFreeTime = ETA

    push runway back to pool
    add flight to timetable
```

Priority Ordering (with tie-breaking):

1. Higher priority value
2. Earlier submit time (if priorities equal)
3. Smaller flight ID (if priorities and submit times equal)

Runway Selection (with tie-breaking):

1. Earlier nextFreeTime
2. Smaller runway ID (if times equal)

Correctness: This greedy approach is optimal for non-preemptive scheduling because:

- We always serve the most urgent flight next
- We assign it to the earliest available resource
- Once started, flights run to completion (non-preemptive)
- Deterministic tie-breaking ensures reproducibility

---

4. Operation Implementations

4.1 Initialize(runwayCount)

Description: Initialize system with specified runways

Algorithm:

```
1. Validate runwayCount > 0
2. Create runways with IDs 1 to runwayCount
3. Set each runway.nextFreeTime = 0
4. Push all runways to runway pool
5. Set currentTime = 0
```

Time Complexity: O(k log k) where k = runwayCount

4.2 SubmitFlight(flightID, airlineID, submitTime, priority, duration)

Description: Add new flight request

Algorithm:

```
1. Advance time to submitTime (two-phase update)
2. Check if flightID already exists
3. Create new Flight object
4. Add to activeFlights map
5. Push to pending heap (save handle)
6. Add to airline index
7. Reschedule all unsatisfied flights
8. Print scheduled ETA
```

Time Complexity: O(log n + n log k) for time advancement and rescheduling

4.3 CancelFlight(flightID, currentTime)

Description: Remove flight that hasn't started

Algorithm:

```
1. Advance time to currentTime
2. Lookup flight
3. Check state (cannot cancel if IN_PROGRESS or COMPLETED)
4. Remove from all structures:
   - Pending heap (using handle)
   - Active flights map
   - Timetable
   - Airline index
5. Reschedule remaining unsatisfied flights
```

Time Complexity: O(log n + n log k)

4.4 Reprioritize(flightID, currentTime, newPriority)

Description: Update flight priority before it starts

Algorithm:

```
1. Advance time to currentTime
2. Lookup flight
3. Check state (cannot reprioritize if started)
4. Update priority in flight object
5. Update pairing heap:
   - If priority increased: use increaseKey
   - If priority decreased: remove and reinsert
6. Reschedule all unsatisfied flights
```

Time Complexity: O(log n + n log k)

4.5 AddRunways(count, currentTime)

Description: Add additional runways to system

Algorithm:

```
1. Advance time to currentTime
2. Validate count > 0
3. Create count new runways with consecutive IDs
4. Set each nextFreeTime = currentTime
5. Push to runway pool
6. Reschedule all unsatisfied flights
```

Time Complexity: O(count log k + n log k)

Effect: More runways → potentially earlier start times for waiting flights

4.6 GroundHold(airlineLow, airlineHigh, currentTime)

Description: Block unsatisfied flights in airline range

Algorithm:

```
1. Advance time to currentTime
2. Validate airlineHigh ≥ airlineLow
3. For each airline in [airlineLow, airlineHigh]:
   - Get flights from airline index
   - For each flight:
     - Check if unsatisfied
     - Add to removal list
4. Remove all identified flights
5. Reschedule remaining unsatisfied flights
```

Time Complexity: O(m log n + n log k) where m = affected flights

Important: IN_PROGRESS flights are not affected (non-preemptive)

4.7 PrintActive()

Description: Display all flights still in system

Algorithm:

```
1. Collect all flights from activeFlights map
2. Sort by flightID
3. Print each flight with format:
   [flightID, airlineID, runwayID, startTime, ETA]
4. For PENDING flights: use -1 for runway/times
```

Time Complexity: O(n log n)

4.8 PrintSchedule(t1, t2)

Description: Show scheduled flights with ETA in range

Algorithm:

```
1. Filter flights where:
   - state == SCHEDULED
   - startTime > currentTime
   - t1 ≤ ETA ≤ t2
2. Sort by (ETA, flightID)
3. Print flight IDs
```

Time Complexity: O(n log n)

Note: Excludes PENDING (no ETA) and IN_PROGRESS (already started)

4.9 Tick(t)

Description: Advance system clock to time t

Algorithm:

```
1. Set currentTime = t
2. Perform two-phase update:
   - Phase 1: Settle all completions
   - Promotion: Mark flights as IN_PROGRESS
   - Phase 2: Reschedule unsatisfied flights
```

Time Complexity: O(c log c + n log k) where c = completions

---

5. Complexity Analysis

Space Complexity

Data Structure	Space	Justification
Pairing Heap	O(n)	One node per pending flight
Runway Pool	O(k)	One entry per runway
Timetable	O(n)	One entry per scheduled flight
Active Flights	O(n)	One entry per active flight
Airline Index	O(n)	All flight IDs distributed
Handles	O(n)	One handle per flight
Total	**O(n + k)**	Linear in flights and runways

Time Complexity Summary

Operation	Average Case	Worst Case	Notes
Initialize	O(k log k)	O(k log k)	Initial heap construction
SubmitFlight	O(log n + n log k)	O(n log k)	Includes rescheduling
CancelFlight	O(log n + n log k)	O(n log k)	Includes rescheduling
Reprioritize	O(log n + n log k)	O(n log k)	Includes rescheduling
AddRunways	O(k' log k + n log k)	O(n log k)	k' = new runways
GroundHold	O(m log n + n log k)	O(n log k)	m = affected flights
PrintActive	O(n log n)	O(n log n)	Sorting output
PrintSchedule	O(n log n)	O(n log n)	Filtering and sorting
Tick	O(c log c + n log k)	O(n log k)	c = completions

Key Observation: Most operations are dominated by the rescheduling phase, which is O(n log k).

---

6. Design Decisions and Tradeoffs

6.1 Pairing Heap vs Fibonacci Heap

Decision: Pairing Heap
Rationale:

- Simpler implementation
- Better practical performance
- Sufficient theoretical bounds
- Easier to debug and maintain

6.2 Full Rescheduling vs Incremental Updates

Decision: Full rescheduling in Phase 2
Rationale:

- Simpler correctness proof
- Handles all scenarios uniformly
- Performance acceptable for problem size
- Easier to detect ETA changes

Tradeoff: O(n log k) per operation vs. potential O(log n) for incremental

6.3 Hash Table Implementation

Decision: std::unordered_map
Rationale:

- Allowed by assignment (not required from scratch)
- Standard, well-tested implementation
- O(1) average case performance
- Focus effort on required structures

---

7. Challenges and Solutions

Challenge 1: ETA Change Detection

Problem: Determining which flights have ETA changes after rescheduling

Solution:

```
// Save old ETAs before rescheduling
unordered_map<int, int> oldETAs;
for (int fid : unsatisfied) {
    if (activeFlights[fid].ETA != -1)
        oldETAs[fid] = activeFlights[fid].ETA;
}

// After rescheduling, compare
for (int fid : unsatisfied) {
    if (activeFlights[fid].ETA != oldETAs[fid])
        changedETAs.push_back({fid, activeFlights[fid].ETA});
}
```

Challenge 2: Pairing Heap Node Removal

Problem: Arbitrary node deletion in pairing heap is complex

Solution:

- Maintain handles map for O(1) node lookup
- Implement recursive removal with subtree merging
- Use two-pass merge for efficiency

Challenge 3: Non-Preemptive Enforcement

Problem: Ensuring IN_PROGRESS flights aren't affected by operations

Solution:

- Promotion step marks flights as IN_PROGRESS
- Reschedule only collects PENDING and SCHEDULED flights
- State checks in Cancel and Reprioritize operations

Appendix: Function Prototypes

Scheduler Class

```
class Scheduler {
public:
    // Constructor
    Scheduler();

    // Main operations
    void initialize(int runwayCount, vector<string>& output);
    void submitFlight(int flightID, int airlineID, int submitTime,
                     int priority, int duration, vector<string>& output);
    void cancelFlight(int flightID, int time, vector<string>& output);
    void reprioritize(int flightID, int time, int newPriority,
                     vector<string>& output);
    void addRunways(int count, int time, vector<string>& output);
    void groundHold(int airlineLow, int airlineHigh, int time,
                   vector<string>& output);
    void printActive(vector<string>& output);
    void printSchedule(int t1, int t2, vector<string>& output);
    void tick(int time, vector<string>& output);

private:
    // Helper methods
    void settleCompletions(int time, vector<string>& output);
    void promoteToInProgress(int time);
    void rescheduleUnsatisfied(vector<string>& output);
    void advanceTime(int time, vector<string>& output);
    vector<int> getUnsatisfiedFlights();
    void removeFlightFromStructures(int flightID);
};
```

Pairing Heap Class

```
class PairingHeap {
public:
    PairingHeap();
    ~PairingHeap();

    PairingNode* push(Flight* flight);
    Flight* top();
    void pop();
    bool empty() const;
    void increaseKey(PairingNode* node, int newPriority);
    void remove(PairingNode* node);
    void clear();

private:
    PairingNode* merge(PairingNode* h1, PairingNode* h2);
    PairingNode* mergePairs(PairingNode* node);
    bool removeHelper(PairingNode*& node, PairingNode* target,
                     PairingNode* parent);
    void clearHelper(PairingNode* node);
};
```

Min Heap Template

```
template<typename T>
class MinHeap {
public:
    MinHeap();

    void push(const T& value);
    T top() const;
    void pop();
    bool empty() const;
    size_t size() const;
    void clear();

private:
    int parent(int i);
    int leftChild(int i);
    int rightChild(int i);
    void heapifyUp(int i);
    void heapifyDown(int i);
};
```