# COP 5536 Fall 2025
## Programming Project
## Due – November 12th

# Gator Air Traffic Slot Scheduler

# Problem Description

Imagine you are building a system to manage flights at an airport with several runways. Every flight sends a request to the system when it wants a slot to take off or land. The request tells us which airline it belongs to, when it was submitted, how urgent it is (its priority), and how long it needs the runway. Once a flight starts using a runway, it cannot be stopped or moved until it finishes. At any moment, some flights may still be waiting for assignment, some may already have a runway reserved for a future time, some may currently be using a runway, and some may have already finished. The goal of the system is to keep track of all these flights, decide the order in which they should use the runways, and update the plan whenever new requests arrive or time moves forward.

**Terms (defined before use)**

- **System time (** *currentTime* **): t**he time value the scheduler is currently considering**.** It only changes when an operation explicitly provides a time parameter.
  For example:
  - when a flight is submitted with a given ***submitTime*** (an integer), or
  - when the user calls Tick(t), where t is an integer input representing the new system time in minutes. (The exact behavior of Tick(t) will be explained later in the Operations section.)
- **Flight request:** a record containing
  - **flightID** (unique integer identifier),
  - **airlineID** (integer),
  - **submitTime** (integer, when the request enters the system),
  - **priority** (integer; larger = more urgent),
  - **duration** (integer minutes needed on a runway).
- **Start time** ( *startTime* ): the integer time (minutes) when a flight begins using its assigned runway.
- **End time / ETA**: the integer time when the flight finishes using its runway, calculated as ETA = *startTime* + duration.
- **Non-preemptive**: once a flight starts on a runway, it cannot be paused, moved, or shortened.
- **Unsatisfied flights**: flights that have not yet started. This set includes both "Pending" and "Scheduled (not started)" states.

Each Flight Lifecycle follows this pattern:

- **Pending** – the flight is in the request pool and not yet bound to a start time or runway.
- **Scheduled (not started)** - the flight has an assigned runway, startTime, and ETA (Estimated Time of Arrival), but *currentTime < startTime*.
- **InProgress** – the flight has started: *startTime <= currentTime < ETA*.
- **Landed** – the flight completed at its ETA and is removed from the system.

# How Time advances (two-phase Update)

Whenever *currentTime* changes (because an operation provides a time), the scheduler performs two phases in order before any requested operation:

**Phase 1 - Settle completions:**

- Find all flights with ETA <= *currentTime*.
- Mark them Completed, remove them from all data structures, and print them in ascending ETA order, breaking ties by smaller *flightID*.

Promotion Step (between phases):

Before Phase 2 begins, any flight with `startTime ≤ currentTime` is marked **InProgress** and excluded from rescheduling (**non-preemptive rule**).

**Phase 2 – Reschedule Unsatisfied Flights**

- Consider all unsatisfied flights (those that are still Pending or Scheduled-but-not-started).
- **Recompute the schedule from the new** *currentTime*.
  We do this because the situation may have changed since the last time we scheduled. For example, some flights may have already completed, a new flight may have arrived, or runways may now be free at different times. To keep the plan fair and consistent, the system clears previous assignments for unsatisfied flights and rebuilds their schedule from scratch.
- **Seeding runway availability**: We rebuild using a fresh copy of the runway pool seeded at *currentTime,* with each in-progress runways's *nextFreeTime* set to flight's ETA (so occupied runways remain blocked until they finish).
- **Greedy scheduling policy** (deterministic and fair):
  When assigning flights during this rebuild, always follow this global policy:
  1. **Choose the next flight** = the one with the highest priority.
     - Tie-breakers (in order): earlier *submitTime*, then smaller *flightID*.
     - This is implemented using a max pairing heap with key (priority, -submitTime, -flightID)
  2. **Choose the runway** = the one with the earliest availability ( nextFreeTime ).
     Tie-breaker: smaller runwayID.

3. **Assign times**:
   *startTime = max (currentTime, runway.nextFreeTime)*
   *ETA = startTime + duration*
4. **Update runway availability**: set *runway.nextFreeTime = ETA*.
5. Repeat until every unsatisfied flight has a new *startTime* and *ETA*.

- **Updated ETAs**:
  If this process causes any not-yet-started flight's ETA to change compared to its previously assigned value, the system prints exactly one line:
  ***Updated ETAs: [flightID1: ETA1, flightID2: ETA2, …]***
  Include only the flights whose ETAs changed, sorted by increasing flightID.

  ***Why can ETAs change?*** Because when flights complete or new requests arrive, the order of assignment may shift, and some flights may get a different runway or start later than before. This leads to different ETAs even for flights that were already scheduled.

- After Phases 1 and 2 are complete, the requested operation itself takes effect (for example, adding a runway, submitting a new flight, reprioritizing, cancelling, or applying a ground-hold).

  ***Why is scheduling updated before performing the requested operation?*** We perform the two phases first so that the system is always in a clean and consistent state before applying a new change.
  If the requested operation changes the set of unsatisfied flights, Phase 2 is run again immediately after, and Updated ETAs are printed if applicable.
  (The details of operations such as adding a runway, submitting a flight, or ground-holding will be fully explained later in the Operations section.)

# Data Structures

Below, each structure is introduced after we define the concept it supports. For each one, we state the key (the ordering rule), the payload (what extra data the node stores beside the key), and the operations you will actually need.

***Note: Max Pairing heap and binary min-heap must be implemented from scratch. Hash tables may use standard library implementations.***

**1. Pending Flights Queue – Max Pairing Heap (Two-Pass Scheme)**
**Role & Purpose:** This is where every new flight request enters first. It ensures the system can always pick the highest-priority flight next, breaking ties by *submit time* and *flightID*.

- **Key (max pairing heap over a triple):** (priority, -submitTime, -flightID)
  Higher priority first. For ties, **earlier** submitTime wins because -submitTime is larger. Next tie: **smaller** flightID wins because -flightID is larger.
- **Payload***:* a pointer/handle to the corresponding flight record.

- **Required Operations**: push, pop (extract best), increase-key (when priority increases), erase by handle (if a flight is cancelled or ground-held (*Explained in Operations section*)).
  (*store a handle/pointer to each pairing-heap node in the per-flight record so you can update/delete in O(log n) time without searching*)
- **Implementation Note:** Must use the two-pass max pairing heap scheme;

2. **Runway Pool – Binary Min Heap**

**Role & Purpose:** This tracks all runways by their next available time. It ensures that when a flight is assigned, it always goes to the earliest free runway (ties by runwayID).

- **Key (min-heap over a pair):** (nextFreeTime, runwayID)
- **Payload:** {runwayID, nextFreeTime}.
- **Workflow:** always pick the earliest free runway for the next assignment. After you schedule on it, update its nextFreeTime = ETA and push it back.

3. **Active Flights – Hash Table (by flightID)**

**Role & Purpose:** Once a flight is scheduled, it is stored here. This lets the system quickly find it by ID for operations like cancellation, reprioritization, or printing all active flights.
(*Cancel means removing a flight from the system if it hasn't started yet, Reprioritize means updating a flight's priority if the airline changes its urgency. Both are explained fully in the Operations section*.)

- **Key**: *flightID*
- **Payload**: {airlineID, priority, duration, runwayID, startTime, ETA}.
- **Why needed:** some operations are naturally ID-based - you want to look up a specific flight.
- **Ops**: insert/delete/search; For printing: Collect from the hash table and sort by flightID before printing to maintain the order.

4. **Timetable (Completions Queue) – Binary Min Heap**

**Role & Purpose:** This keeps all scheduled flights sorted by their completion time. It ensures the system can efficiently find which flights should finish when time advances or when a Tick(t) command moves the system clock forward. (*Tick(t) means advancing the current system time to t, which triggers settling completions and rescheduling. The full procedure is explained in the Operations section.*)

- **Key:** (ETA, flightID) (ordered first by completion time, then by ID).
- **Payload:** {runwayID} (other fields are retrieved via Active Flights)
- **Why needed:** some operations are naturally time-based - you want to know which flights complete next.
- **Ops**: insert, pop all flights with ETA <= currentTime

5. **Airline Index - Hashtable**

**Role & Purpose:** This groups flights by airline while they are still unsatisfied (pending or scheduled-not-started). It makes airline-wide operations efficient without scanning all flights. (*For example, a*

*GroundHold on an airline means all of its unsatisfied flights are temporarily blocked from being scheduled. The details of GroundHold will be described in the Operations section.*)

- **Key → value:** airlineID → set/list of flightIDs
- **Ops:** add/remove as state changes; iterate the set when applying a hold.
- **Note:** The exact behavior of GroundHold will be described in detail in the Operations section.

6. **Handles - HashTable (for cross-updates)**

**Role & Purpose:** This central map ties everything together. It stores references to where each flight lives in the other data structures, so updates and deletions happen quickly and consistently.

- **Key → value:**
  flightID → { state, pairingNode?} for O(1) lookups and coordinated updates.
  (If you implement extra handles, keep them here; at minimum you need the pairing-heap node for Pending.)
- **Benefit:** O(1) lookup, O(log n) structural updates, no linear scans.

***System rules to remember (summary):***

1. **State transitions**
   • A flight enters In Progress when currentTime = startTime.
   • A flight becomes Completed when currentTime ≥ ETA.
2. **Unsatisfied flights**
   • Always: Unsatisfied = Pending U Scheduled (not started) .
   • Only unsatisfied flights are rescheduled in Phase 2.
3. **Completions (Phase 1 output)**
   • Print flights that complete (ETA ≤ currentTime ) in ascending ETA.
   • Break ties by smaller flightID.
4. **Updated ETAs (Phase 2 output)**
   • Print only if a flight's ETA changes after rescheduling.
   • Include only those flights, sorted by flightID.
5. **Order of scheduling and operations**
   • Always run Phase 1 (settle completions) and Phase 2 (reschedule) before applying the requested operation.
   • If the operation changes the unsatisfied flights, run Phase 2 again and print Updated ETAs if needed.
6. **Determinism**
   • Ties between flights: earlier submitTime, then smaller flightID.
   • Ties between runways: smaller runwayID .
   • Ensures all correct implementations produce the same output.

## Operations:

(Rule that always applies*: If an operation has a time parameter, you must settle to that time first (landings + reschedule + maybe Updated ETAs ) before executing the operation's own action. If that action modifies the set of pending/scheduled flights, reschedule again and print Updated ETAs if some ETAs changed.)*

1. **Initialize(runwayCount)**

**Purpose:** Start the system with a given number of runways. No flights exist yet.
**Behavior:**

- If *runwayCount* ≤ 0 → print **"Invalid input."**
- Else:
    - Create runways with IDs *1 ... runwayCount,*
      Set each runway's *nextFreeTime = 0,* Set *currentTime = 0* .
    - Print confirmation that the system has been initialized with "*runwayCount"* runways.

**Outputs:**

- **Valid integer →**
  < runwayCount > Runways are now available
- **Invalid →**
  Invalid input. Please provide a valid number of runways.

2. **SubmitFlight(flightID, airlineID, submitTime, priority, duration)**

**Purpose:** Add a new flight request to the system at a given submitTime.
**Behavior:**

- Advance time to submitTime (settle completions + reschedule).
- If flightID already exists → print "Duplicate flight" and stop.
- Else:
    - Insert flight into the Pending Flights Queue.
    - Reschedule all unsatisfied flights using the greedy policy.
    - Print the new flight's assigned ETA.
- If this rescheduling causes other unsatisfied flights to get new ETAs → print a single Updated ETAs line listing only those flights (sorted by flightID).

**Outputs:**

- **On success (after re-schedule)**:
  Flight <flightID> scheduled - ETA: <ETA>
- **If flightID is a duplicate**:
  Duplicate FlightID

- **If other ETAs changed**:
  Updated ETAs: [<flightID1>: <ETA1>, ...]

**Example:**

**Flow of SubmitFlight(203, airlineID=1, submitTime=0, priority=10, duration=5)**

Let's Suppose we have Initial State:

- System initialized with 2 runways.
- currentTime = 0.
- Already scheduled:
  - Flight 201 → Runway 1, [0-4], ETA=4.
  - Flight 202 → Runway 2, [0-3], ETA=3.

**Step 1 - Advance time to submitTime (0)**

- submitTime = 0, which is equal to current time.
- **Phase 1**(settle completions): No flights have ETA ≤ 0, so no landings.
- **Promotion step:** Flights with startTime ≤ currentTime (201 and 202) are marked **InProgress** and excluded from rescheduling (non-preemptive).
- **Phase 2** (pre-operation reschedule): there are no unsatisfied flights yet → no changes.

**Step 2 - Check for duplicates**

- Flight 203 does not exist in the system, Continue.

**Step 3 - Insert into Pending Flights Queue (submitFlight Operation performed)**

- Flight 203 is added to the Pending Max Pairing Heap with key (priority, -submitTime, -flightID) = (10, 0, -203).

**Step 4 - Reschedule unsatisfied flights (post-operation Phase 2)**

- Unsatisfied = flights that have not yet started (currentTime < startTime). Since flights 201 and 202 are **InProgress**, only flight **203** is considered.
- The earliest free runway after `currentTime=0` is **Runway 2**, which becomes free at `t=3` (after flight 202 lands).
- Assign flight 203: `startTime=3`, `ETA=3+5=8`, Update Runway 2 → `nextFreeTime=8`

**Step 5 - Compare ETAs with old values**

- Flight 203: new flight, so we just print its ETA. No existing unsatisfied flights' ETAs changed.

**Final Output:** Flight 203 scheduled - ETA : 8

3. **CancelFlight(flightID, currentTime)**

**Purpose:** Remove a flight that has not started yet. If it started or completed, you can't cancel it.
**Behavior:**

- Advance time to currentTime → settle completions → reschedule unsatisfied.
- Lookup flightID. If not found → print "Flight does not exist."
- If flight is In Progress or Completed → print "Cannot cancel: Flight <id> has already departed."
- Else (Pending / Scheduled-not-started):
    - Remove it from: Pending Max Pairing heap / Active table / Timetable / Airline index / Handles.
    - Reschedule remaining unsatisfied flights from *currentTime*.
    - Print: "Flight <id> has been canceled."
    - If other ETAs changed → print single Updated ETAs: [...].

**Outputs:**

- **Removed (pending or not-started)**:
  Flight <flightID> has been canceled
- **Already in progress or landed**:
  Cannot cancel. Flight < flightID > has already departed
- **Not found:**
  Flight <flightID > does not exist
- **If other ETAs changed after reschedule**:
  Updated ETAs: [<flightID1>: <ETA1>, ...]

4. **Reprioritize (flightID, currentTime, newPriority)**

**Purpose:** Change a flight's priority before it starts; then rebuild schedule so it may move earlier/later.
**Behavior:**

- Advance time to *currentTime → settle completions → reschedule unsatisfied*.
- Lookup *flightID*. If not found → print “Flight <id> does not exist.”
- If flight is **In Progress** or **Completed** → print “Cannot reprioritize: Flight <id> has already departed.”
- Else (Pending / Scheduled-not-started):
    - Update priority (pairing heap: increase-key if priority increases; erase+insert if decreases).
    - Reschedule all unsatisfied flights from currentTime using greedy policy.
    - Print: "Priority of Flight <id > has been updated to <newPriority>."
    - If other flights' ETAs changed → print single Updated ETAs: [...].

**Outputs:**

- **Updated (pending or scheduled-not-started):**
  Priority of Flight < flightID > has been updated to < newPriority >
- **Already in progress or landed:**
  Cannot reprioritize. Flight <flightID> has already departed
- **Not found:**
  Flight < flightID > not found
- **If other ETAs changed after reschedule**:
  Updated ETAs: [<flightID1>: <ETA1>, ...]

**Example:**

- System: 2 runways, currentTime=0. Already scheduled:
  - 401 (priority 7, duration 4) → R1 [0-4], ETA=4
  - 404 (priority 5, duration 2) → R2 [0-2], ETA=2
  - 402 (priority 6, duration 3) → R2 [2-5], ETA=5
  - 403 (priority 5, duration 5) → R1 [4-9], ETA=9

  **Command:** Reprioritize(403, currentTime=0, newPriority=10)
  **Flow:**

  - Phase 1 @ t=0 → nothing lands.
  - Promotion step: 401 and 404 have startTime=0 → InProgress (non-preemptive).
  - Phase 2 (pre-operation): unsatisfied {402,403}
    - Rebuild schedule using current priorities → 402 [2-5], 403 [4-9] (no ETA change)
  - Apply operation: Increase 403's priority to 10.
  - Phase 2 (post-operation): Re-run scheduling with new priorities
    - 403: start=2, ETA=7 and 402: start=4, ETA=7

  **Output:**
  Priority of Flight 403 has been updated to 10
  Updated ETAs: [402: 7, 403: 7]

5. **AddRunways(count, currentTime)**

**Purpose:** Increase capacity by adding count new runways (available starting at currentTime ), then reschedule.
**Behavior:**

- Advance time to *currentTime → settle completions → reschedule unsatisfied*.
- If count <= 0 → print "Invalid input."
- Else: Create ***count*** new runways with consecutive IDs; set each *nextFreeTime = currentTime*; push into runway heap.
- Reschedule all unsatisfied flights from *currentTime* using the greedy policy.

- Print: confirmation that additional runways are available.
- If any ETAs changed → print single Updated ETAs: [...].

**Outputs:**

- **Valid integer →**
  Additional <count> Runways are now available
- **Invalid →**
  Invalid input. Please provide a valid number of runways.
- **If other ETAs changed after reschedule**:
  Updated ETAs: [<flightID1>: <ETA1>, ...]

**Example**:

- System initialized with 2 runways, currentTime=0. Already in system:
  - 501 (priority 8, duration 4) → R1 [0-4], ETA=4
  - 502 (priority 7, duration 6) → R2 [0-6], ETA=6
  - 503 (priority 6, duration 3) → scheduled later as R1 [4-7], ETA=7

  **Command:** AddRunways (count=1, currentTime=0)

  **Flow:**

  Reschedule with Runway 3 now also free at 0:

  - 501 stays R1 [0-4], 502 stays R2 [0-6]
  - 503 moves earlier → R3 [0-3], ETA=3

  **Output:**

  Additional 1 Runways are now available
  Updated ETAs: [503: 3]

6. **GroundHold (airlineLow, airlineHigh, currentTime)**

**Purpose:** Temporarily block unsatisfied flights whose airlineID is in [airlineLow, airlineHigh] from being scheduled.

**Behavior:**

- Advance time to *currentTime* → settle completions → reschedule unsatisfied.
- If *airlineHigh* < *airlineLow* → print "Invalid input. Please provide a valid airline range."
- Else: Remove all unsatisfied flights with *airlineID* $\in$ *[low, high]* from all structures (pending/active/timetable/airline-index/handles).
  - (Flights In Progress keep running; GroundHold does not affect them.)
- Reschedule the remaining unsatisfied flights from currentTime.
- Print: "GroundHold applied to airlines [low, high]."

- If any other ETAs changed → print single Updated ETAs: [...].

**Outputs:**

- **Valid range (high ≥ low) →**
  Flights of the airlines in the range [<airlineLow>, < airlineHigh>] have been grounded
- **Invalid range →**
  Invalid input. Please provide a valid airline range.
- **If other ETAs changed after reschedule**:
  Updated ETAs: [<flightID1>: <ETA1>, ...]

**Example:**

- System initialized with 2 runways. At currentTime=1, scheduled as:
  - 601 (airlineID=10, priority 8, duration 4) → R1 [0-4] (In Progress at t=1)
  - 602 (airlineID=12, priority 7, duration 3) → R2 [2-5] (Scheduled-not-started at t=1)
  - 603 (airlineID=9, priority 6, duration 3) → R1 [4-7] (Scheduled-not-started)

  **Command**: GroundHold(airlineLow=10, airlineHigh=12, currentTime=1)
  **Flow:**

  - Advance to 1 → nothing completes; 601 is In Progress (not affected).
  - Remove unsatisfied flights with *airline ∈ [10, 12]*: removes 602 (air=12).
  - Reschedule remaining unsatisfied (603) from t=1:
    - With 602 removed, R2 is free at 0, so start = max(1,0)=1 → 603 → R2 [1-4], ETA=4 (was 7).

  **Output:**

  Flights of the airlines in the range [10, 12] have been grounded
  Updated ETAs: [603: 4]

7. **PrintActive()**

**Purpose:** Show all flights that are still in the system (pending or scheduled/not-started or in-progress).
**Behavior:**

- Do not change *currentTime*.
- Gather flights from Active Flights (hash table), sort by flightID.
- Print one line per flight: *flightID, airlineID, runwayID, startTime, ETA*.
  - For Pending flights with no assigned times, use –1 for times.
    (e.g., startTime=-1, ETA=-1).

**Outputs:**

- **If flights exist (ordered by flightID):**
  [flight<flightID>, airline<airlineID>, runway<runwayID>, start<startTime>, ETA< ETA>]

- **If none:**
  No active flights

**Example:**

- **System:** 2 runways, *currentTime* = 2.
  Active contains:
  - 701 (In Progress) → R1 [1-5]
  - 702 (Scheduled) → R2 [3-4]
  - 703 (Pending) → (no times yet)

  **Command:** PrintActive ()
  **Output: (sorted by flightID):**
  [flight701, airline<id>, runway1, start1, ETA5]
  [flight702, airline<id>, runway2, start3, ETA4]
  [flight703, airline<id>, runway-1, start-1, ETA-1]

8. **PrintSchedule(t1, t2)**

**Purpose:** Show Only Unsatisfied flights (i.e., flights that are Scheduled-but-not-started) whose ETA lies within the interval [t1, t2].

**Behavior:**

- Do not change currentTime.
- Gather flights from Active Flights Table
- Filter only those flights that:
  - *Have state == SCHEDULED*
  - have *startTime > currentTime* (they have not started yet).
  - have *ETA ∈ [t1,t2]*
- Ignores **Pending** flights (no assigned ETA yet) and **InProgress** flights (already started).
- Sort results by (ETA, flightiD) before printing.

**Outputs:**

- **If flights exist (ordered by ETA, then flightID)**:
  [< flightID >]
- **If none:**
  There are no flights in that time period

**Example:**

- **Timetable currently has:**
  - 801 → ETA=4 (start=0) (In Progress at t=3),
  - 802 → ETA=6 (start = 5) (Scheduled, not started)
  - 803 → ETA=9 (start = 8) (Scheduled, not started)

**Command:** PrintSchedule (5, 9)

**Output:**
[802]
[803]

9. **Tick(t)**

**Purpose:** Advance the system clock to t, land whatever finishes by t, then reschedule what's left.
**Behavior:**

- Advance *currentTime* → t .
- Phase 1: complete & print all flights with ETA <= t in (ETA, flightID) order; remove from all structures.
- Phase 2: reschedule unsatisfied from time t.
- If any ETAs changed → print single Updated ETAs: [...].

**Outputs:**

- **For each landed flight (ordered by ETA, tie by flightID)**:
  Flight < flightID > has landed at time < ETA >
- **If none landed and no ETAs changed**: print nothing.
- **If ETAs changed after reschedule**:
  Updated ETAs: [<flightID1>: <ETA1>, ...]

10. **Quit()**

**Purpose:** End processing immediately.
**Behavior:** Print the termination line (e.g., "Program terminated.").
Output:

```
Program Terminated!!
```

# Programming Environment:

You may use either Java, C++, or Python for this project. Your program will be tested using the Java or g++ compiler or python interpreter on the thunder.cise.ufl.edu server. So, you should verify that it compiles and runs as expected on this server, which may be accessed via the Internet. Your submission must include a Makefile that creates an executable file named gatorAirTrafficScheduler.

Your program should execute using the following:

**For C/C++:**

$ ./gatorAirTrafficScheduler file_name

**For Java:**

$ java gatorAirTrafficScheduler file_name

**For Python:**

$ python3 gatorAirTrafficScheduler file_name

*Where file_name is the name of the file that has the input test data.*

## Input and Output Requirements

- Read Input from a text file where input_filename is specified as a command-line argument.
- All Output should be written to a text file having filename as concatenation of "input*_filename" + "_"* + "output_file.txt"(e.g., input_filename = "test1.txt", output_filename = "test1_output_file.txt")
- The program should terminate when the operation encountered in the input file is Quit()

## Example 1:

**Input**

```
Initialize(2)
SubmitFlight(201, 1, 0, 5, 4)
SubmitFlight(202, 2, 0, 6, 4)
SubmitFlight(203, 1, 0, 4, 5)
PrintSchedule(1, 10)
SubmitFlight(205, 4, 2, 7, 2)
PrintSchedule(4, 6)
Reprioritize(203, 3, 9)
GroundHold(5, 4, 3)
GroundHold(4, 4, 3)
AddRunways(1, 3)
SubmitFlight(206, 6, 3, 6, 4)
CancelFlight(206, 3)
Tick(4)
SubmitFlight(204, 3, 4, 8, 2)
AddRunways(0, 5)
PrintActive()
PrintSchedule(5, 10)
Quit()
```

**Output**

```
2 Runways are now available
Flight 201 scheduled - ETA: 4
Flight 202 scheduled - ETA: 4
Flight 203 scheduled - ETA: 9
[203]
Flight 205 scheduled - ETA: 6
[205]
```

```
Priority of Flight 203 has been updated to 9
Invalid input. Please provide a valid airline range.
Flights of the airlines in the range [4, 4] have been grounded
Additional 1 Runways are now available
Updated ETAs: [203: 8]
Flight 206 scheduled - ETA: 8
Flight 206 has been canceled
Flight 201 has landed at time 4
Flight 202 has landed at time 4
Flight 204 scheduled - ETA: 6
Invalid input. Please provide a valid number of runways.
[flight203, airline1, runway3, start3, ETA8]
[flight204, airline3, runway1, start4, ETA6]
There are no flights in that time period
Program Terminated!!
```

## Example 2:

**Input**

```
Initialize(2)
SubmitFlight(301, 5, 0, 7, 4)
SubmitFlight(302, 6, 0, 6, 5)
SubmitFlight(303, 7, 0, 5, 3)
PrintSchedule(5, 8)
GroundHold(6, 6, 0)
SubmitFlight(301, 5, 0, 7, 4)
SubmitFlight(304, 8, 1, 9, 4)
Reprioritize(304, 1, 10)
PrintSchedule(4, 8)
AddRunways(1, 1)
PrintSchedule(4, 8)
SubmitFlight(305, 9, 2, 6, 3)
CancelFlight(305, 2)
AddRunways(0, 2)
Tick(3)
PrintActive()
PrintSchedule(3, 7)
Tick(4)
Quit()
```

**Output**

```
2 Runways are now available
Flight 301 scheduled - ETA: 4
Flight 302 scheduled - ETA: 5
Flight 303 scheduled - ETA: 7
```

```
[303]
Flights of the airlines in the range [6, 6] have been grounded
Duplicate FlightID
Flight 304 scheduled - ETA: 8
Updated ETAs: [303: 8]
Priority of Flight 304 has been updated to 10
[303]
[304]
Additional 1 Runways are now available
Updated ETAs: [303: 7, 304: 5]
[303]
Flight 305 scheduled - ETA: 7
Updated ETAs: [303: 8]
Flight 305 has been canceled
Updated ETAs: [303: 7]
Invalid input. Please provide a valid number of runways.
[flight301, airline5, runway1, start0, ETA4]
[flight302, airline6, runway2, start0, ETA5]
[flight303, airline7, runway1, start4, ETA7]
[flight304, airline8, runway3, start1, ETA5]
[303]
Flight 301 has landed at time 4
Program Terminated!!
```

# Submission Requirements

- Include a makefile for easy compilation.
- Provide well-commented source code.
- Submit a PDF report that includes project details, function prototypes, and explanations.
- Follow the input/output and submission requirements as described above.

**Do not use nested directories.** All your files must be in the first directory that appears after unzipping. You must submit the following:

1. Makefile: You must design your makefile such that "make" command compiles the source code and produces an executable file. (For java class files that can be run with java command)
2. Source Program: Provide comments.
3. Report:
   - The report should be in PDF format.
   - The report should contain your basic info: Name, UFID and UF Email
   - Present function prototypes showing the structure of your programs. Include the structure of program.

To submit, please compress all your files together using a zip utility and submit to the Canvas system. You should look for the "Assignment Project" for the submission. Your submission should be named "**LastName_FirstName.zip**"

***Important:*** *Please make sure the name you provided is the same as the same that appears on the Canvas system.* ***Please do not submit directly to a TA****. All email submissions will be ignored without further notification. Please note that the due date is a hard deadline. No late submission will be allowed. Any submission after the deadline will not be accepted.*

# Grading Policy

Grading will be based on the correctness and efficiency of algorithms. Below are some details of the grading policy.

- Correct implementation and execution: 25 pts
- Comments and readability (Source code and naming conventions): 15 pts
- Report: 20 pts
- Testcases: 40 pts

***Important:*** *Your program will be graded based on the produced output. You must make sure to produce the correct output to get points. There will be a threshold for the running time of your program. If your program runs slow, we will assume that you have not implemented the required data structures properly. You will get negative points if you do not follow the input/output or submission requirements above.*

***Following is the clear guidance on how your marks will be deducted:***

- Source files are not in a single directory after unzipping: -5 points
- Incorrect output file name: -5 points
- Error in make file : -5 points
- Make file does not produce an executable file that can be run with one of the commands mentioned in the Programming Environment Section: -5 points
- Hard coded input file name instead of taking as an argument from the command prompt: - 5 points
- Not following the Output formatting specified in examples: -5 points
- Any other input/output or submission requirement mentioned in the document: -3 points

Also, we may ask you to fix the above problems and demonstrate your projects.

# Miscellaneous:

Implement Pairing Heap and Binary min-heap from scratch, without using built-in libraries. Your implementation should be your own. You must work by yourself for this assignment (discussion is allowed). Your submission will be checked for plagiarism!!