CSCE 2211 Term Project Spring 2025

Hamdy El-Madbouly, Malk Metwally, Sedra Elkhayat, Radwa AbuElfotouh

Department of Computer Science and Engineering, American University of Cairo

CSCE22101: Applied Data Structures

Dr. Ashraf AbdelRaouf, Dr. Dina Mahmoud

December 7, 2025

1 Abstract

This report documents the development of a conflict-detection system for university library services, where students can reserve study rooms, borrow books, and laptops. It begins with an introduction to the system and a problem definition outlining the challenges of managing time-based reservations efficiently. The methodology section then explains the data structures explored and selected, such as the Interval Tree, HashMap, and other supporting structures. Detailed algorithm specification, followed by the data format used for user inputs and stored records. The report also includes experimental test results, an analysis of the system's strengths and limitations, and suggestions for future improvements. Finally, team contributions, references, and a link to the GitHub repository where the project is uploaded.

2 Problem Definition

University libraries offer essential services such as study room reservations, book and laptop borrowing, all of which operate within specific time constraints. That is, to function effectively, these services must ensure that no overlapping reservations occur, whether between different users or within a single user's schedule. For example: two students reserving the same study room at the same time, attempting to borrow an already-loaned book, or assigning a laptop that is no longer available. Additionally, users themselves must not be allowed to book the same resources at the same time (e.g., rooms, laptops, or books).

This conflict could be solved through many data structures, from simple linear data structures to complex non-linear ones. What makes the difference is their efficiency and optimality. While simple linear data structures, such as arrays, can be used, they quickly become inefficient at larger scales. This inefficiency directly impacts system performance and user experience. To solve this issue, our project proposes the implementation of a conflict detection program that aims to design a non-linear data structure, which is the interval tree implemented using the Red-Black tree. It quickly detects time conflicts between existing and incoming reservations, efficiently updates stored intervals when new bookings are added or modified, and scales effectively as the number of users, rooms, and reservations increases. By integrating this structure, the system can deliver fast, reliable conflict detection.

3 Data Structures

This section will cover the purpose and a description of the key features of the main data structures used in this application to solve the problem as well as other data structures that were initially built but did not end up being used in the final application and why.

3.1 Hashmap

3.11 Overview

The HashMap class is a core component of the Library Time Conflict Detection Tool, providing efficient storage and retrieval of library resources and their associated booking schedules. This generic hash table implementation uses separate chaining for collision resolution and is designed to manage various library resources such as devices, books, study rooms, and user information.

3.12 Purpose

In the context of the library booking system, the HashMap serves two primary functions:

1. Resource Management: Maps resource identifiers (e.g., "StudyRoom-101", "Laptop-042", "Book-ISBN") and their corresponding interval trees that track booking time slots
2. User Management: Maps user identifiers to user profiles and their booking histories

3.13 Key Features

1. Constant-Time Operations: O(1) average lookup, insertion, and deletion. Efficient for large-scale systems with thousands of resources
2. Automatic Resizing: Automatically doubles capacity when the load factor exceeds 0.75
3. Collision Handling: Separate chaining prevents data loss in hash collisions
4. Type Safety: Template-based implementation enforces type safety at compile time

3.2 Interval Tree

3.21 Overview

It is a tree that stores time intervals by storing both the start and end of the time interval within each node. Each node also stores a max-end value - this is the largest endtime in its entire subtree. This makes adding new intervals and searching for conflicts within the interval tree to be done efficiently as the max-end values allow the tree to quickly skip entire subtrees that cannot contain overlapping intervals, reducing the search space and improving insertion and conflict detection efficiency. However, in our implementation,we added one more feature - and this is the self-balancing feature using Red-Black tree properties. This way our interval tree is a self-balancing tee. This is advantageous because all major operations - such as inserting a reservation, removing a reservation, or checking for conflicts - run in O(log n) time even as the

number of bookings grows. This makes it well-suited for a system that must handle many resources.

3.22 Purpose

In our library booking system, each resource (study room, laptop, or book) has its own Interval Tree that manages all its reservations. Each user also has their own Interval Tree to manage their own reservations. Thus, the Interval Tree is responsible for:

1. Preventing scheduling conflicts
2. Inserting new intervals after checking that they do not conflict with bookings already made

3.33 Key Features

1. Conflict Detection: Quickly checking whether a new reservation overlaps with any existing reservation for the same resource or the same user. This ensures that no two users can book the same room or borrow the same book/laptop at the same time and that a single user cannot book two of the same resource at the same time.
2. Updating Intervals: When users book new resources, the tree inserts this new interval into both the resources' interval tree and the user's interval tree quickly and efficiently.
3. Self-Balancing Structure: Automatic balancing through Red-Black Tree operations keeps operations efficient even as the booking dataset grows.
4. Free-Time Listing: In-order traversal that identifies gaps between existing bookings, converts them to human-readable timestamps, and prints them as numbered available time slots.

3.3 Priority Queue

3.31 Overview

A Priority Queue is a specialized data structure that stores elements in a way that allows the highest-priority element to be accessed first, regardless of the order they were added. Unlike a normal queue (FIFO), where the first element inserted is served first, a priority queue arranges elements by a defined priority rule - meaning insertion order does not determine service order. In this implementation, the priority queue is built using a binary heap, providing: O(log n) insertion and deletion O(1) access to the highest-priority element.

3.32 Purpose

In the context of the library booking system, the priority queue acts as follows: Each book in the library has its own priority queue that manages all active borrow requests for that specific title. The queue ensures: Requests are handled in the order they were submitted (standard queue logic). Conflicts between overlapping requests are resolved fairly. Students who borrowed the same book before cannot automatically monopolize it if another student requests it next. This is done by the Exception - Borrowing Conflict Rule: If a student who borrowed the same book previously submits a new request and another student requests it for an overlapping interval, the other student's request receives higher priority, even if it was submitted later.

3.33 Key Features

1. Priority-Based Ordering: Makes sure borrow requests are automatically sorted so that the most important ones are handled first.
2. Grows When Needed: The queue gets bigger automatically if there are too many requests, so it can handle lots of users.
3. Quick Access: Can find the highest-priority request efficiently
4. Efficient Adding and Removing: Adding a new request or removing one keeps everything in the right order without taking too long

3.4 Queue

3.4.1 Overview

In our library scheduling system, we implemented a Queue data structure to manage multiple user commands efficiently, such as borrowing books, reserving laptops, and booking study rooms, all within a single program run.

A Queue follows the First-In-First-Out (FIFO) principle, meaning that commands are processed in the exact order they are received. This guarantees fairness, order preservation, and smooth handling of sequential tasks, especially when users perform multiple actions at once.

3.4.2 Purpose

In our library scheduling system, the Queue was initially considered to manage multiple user commands efficiently. Users can perform various actions such as borrowing books, reserving laptops, or booking study rooms. By using a Queue, we aimed to:

1. Process commands in the order they are received.
2. Ensure fairness and consistency in handling multiple requests.
3. Prevent overlapping operations or conflicts caused by simultaneous command processing.

3.4.3 Key Features

1. Time complexity of the operations (enqueue, dequeue, peek): O(1)
2. FIFO Order: Guarantees that the first command entered is the first one executed.
3. Sequential Processing: Commands are handled one at a time, avoiding potential conflicts in resource allocation.
4. Circular Structure: Prevents memory waste by reusing spaces freed after dequeuing.
5. Dynamic Resizing: Automatically increases the queue size if it becomes full, ensuring smooth handling of a large number of commands.

4 Specifications of Algorithms

In this section, we delve into the specifications for our algorithmic approaches to handling time conflicts. First, we consider *Finding our Reference Point* and storing our time intervals (for scheduling) in the interval tree. Then, we look into the ways we converted the inputted (from the user) information into seconds passed from that particular reference point. In later subsections, we talk about booking resources in the form of rooms, laptops, and books.

4.1. Finding out the reference point

We considered a time reference point of 00:00 January 1, 2025. This was inspired by [2], where in computer systems, a reference point (epoch) is used to be able to calculate times in seconds. However, here we considered that particular reference to decrease the memory size required to store information, as referencing 1970 would be too long and simply unnecessary.

4.2 Converting inputted interval to seconds passed reference point

After a user enters their data in a HH:MM DD/MM/YYYY format, the date is converted to seconds with 00:00 January 1, 2025, as a reference date. It is then stored in the interval tree based on the number of seconds from the epoch, stored in the database, and then simply converted back to a human-readable format.

4.3 Booking Resources

In the next few sections (books), we note that resources were managed using nearly similar methodologies. For instance, all data/resources were managed using their own respective manager (inspired by Django's backend implementation design), where borrowing, adding (only accessible to admins), and storing the data are handled mainly using HashMaps. Interval trees were referenced for each resource to keep track of the schedules associated with each resource.

4.3.1 Booking a room

Users can book a room with a maximum limit of 3 hours per booking. We note that the only attribute characterizing a room is its ID, which represents its respective placement (position)–floor and number–inside the physical library.

4.3.2 Borrowing a laptop

Users can borrow a laptop for up to 30 days per borrow. The system will promote the first available laptop to the user. We note that the only attribute characterizing a laptop is its ID, whilst all laptops in a library should have the same hardware/software characteristics.

In later implementations, we consider using a priority queue such that the least used laptop to decrease in possibility of abused and create cycles of laptop usage.

4.3.3 Borrowing a book

Books are managed using a standalone BooksManager unit that handles borrowing, adding, and storing book resources inside the library. Books are stored in the database (modeled as a HashMap) by title as the primary key. That key serves as a secondary key (identification) to associate each book with its interval tree (schedule), enabling effective handling of possible time conflicts. Authors are stored as keys in a third HashMap, each associated with their respective book by a linked list. In other words, there are three HashMaps inside the BookManager:

1. A HashMap by book title to store books
2. A HashMap from book title to store books' schedule.
3. A HashMap of book author names to store books by that author in a linked list.

Using this model, we implemented a search by book title and author, allowing the user to enter either. If the input were a book title, the system would simply use it to retrieve the associated book, achieving a best time complexity of O(1) and worst time complexity of strictly less than O(n). In case the input was a book author name, the system will retrieve the linked list of all associated books and ask the user which one they would choose from. Then, using the latter HashMap, we retrieve the remaining Book data, maintaining the same complexity. In either case, the system will check the suitability of borrowing, and decide whether to store the booking in the second HashMap (which stores the books' schedules) and inside the appropriate user's schedule.

However, we note that each user has constraints of at most thirty (30) days of booking, and at most three (3) simultaneous bookings in the same time interval.

5 Data Specifications

This section will cover the input data used in the project. As we have both a Command Line Interface and a Graphical User Interface, this section will look at the input data for each section separately.

5.1 Command Line Interface

Part of the program	What the user will input	Format of Input	Any notes on input format
Login Page	Their username	String - no specific format	N/A
	Their password	String - no specific format	N/A
Booking a Study Room	Room ID of the room they would like to reserve	Example: R001	N/A
	Starting date of desired booking period	mm/dd/yyyy	Our system allows users to book a study room for the current date or the next day. Hence, the date entered will be the current day or the next day
	End date of desired booking period		
	Starting time of desired booking period	hh:mm	The 24-hour clock is used and so hours are from a range of 0-24
	End time of desired booking period		
Borrowing Laptop	Starting date of desired borrowing period	mm/dd/yyyy	Users are not allowed however to borrow a laptop for more than one month. So start and end date should be one part at a maximum
	End date of desired borrowing period		
	Starting time of desired borrowing period	hh:mm	The 24-hour clock is used and so hours are from a range of 0-24
	End time of desired borrowing period		

Borrowing a Book	The user chooses whether they would like to search by title or search by author	Integer - 1 or 2	N/A
	If user chooses search by title - they enter the title of the book they are searching for	String - no specific format	N/A
	If the user chooses to search by author - they enter the author they are searching for. After a list of the all the books written by desired author is shown, the user will be prompted to enter the title of the book from this author they are searching for	String - no specific format String - no specific format	If user inputs title where the author of entered title does not match the author whose books they were initially searching for - then warning appears and user is asked to confirm if they still want to borrow the book with the title they just entered
	Starting date of desired borrowing period	mm/dd/yyyy	Users are not allowed however to borrow a book for more than one month. So start and end date should be one part at a maximum
	End date of desired borrowing period		
	Starting time of desired borrowing period	hh:mm	The 24-hour clock is used and so hours are from a range of 0-24
	End time of desired borrowing period		

5.2 Graphical User Interface

Part of the program	What the user will input	Format of Input	Any notes on input format
Login Page	Their username	No specific format	N/A

	Their password	No specific format	N/A
Booking a Study Room	Room ID of the room they would like to reserve from a drop down menu	Select from drop down menu	N/A
	Starting date of desired booking period	Select appropriate tab corresponding to their desired date	Our system allows a user to book a study room for either the current day or the next day. So there are 2 tabs - once for the current day one for next - the user will choose between them
	End date of desired booking period		
	Starting time of desired booking period	They move the slider to indicate the start and end time and date of their desired booking period	N/A
	End time of desired booking period		N/A
Borrowing Laptop	Starting date of desired borrowing period	Enter day, month, and year into their slots on the screen	Users are not allowed however to borrow a laptop for more than one month. So start and end date should be one part at a maximum
	End date of desired borrowing period		
	Starting time of desired borrowing period	Enter hour and minute into their respective slots on the screen	The 24-hour clock is used and so hours are from a range of 0-24
	End time of desired borrowing period		
Borrowing a Book	If the user chooses to search by title - they enter the author they are searching for As they do the matching results show up and they click the select	String - no format Click “Select”	N/A

	button for their chosen book		
	If the user chooses to search by author - they enter the author they are searching for. After a list of the all the books written by desired author is shown, the user selects their desired book	String - no format Click “Select”	N/A
	Starting date of desired borrowing period	Enter day, month, and year into their respective slots on the screen	Users are not allowed however to borrow a book for more than one month. So start and end date should be one part at a maximum
	End date of desired borrowing period		
	Starting time of desired borrowing period	Enter hour and minute into their respective slots on the screen	The 24-hour clock is used and so hours are from a range of 0-24
	End time of desired borrowing period		

6 Experimental Results

Test Name	Purpose of Test Case	Input	Expected Output	Actual Output	Pass/Fail
Add and Get Book	Test adding a new book succeeds	addBookDirect("B001", "Title1", "Author1")	True	True	PASS
	Test retrieving existing book	getBook("B001")	non-null pointer	non-null pointer	PASS
	Test adding a duplicate book fails	ddBookDirect("B001", "Title1", "Author1")	False	False	PASS
Remove Book	Test removing an existing book succeeds	removeBookDirect("B002") after adding it	True	True	PASS
	Test removed book no longer exists	getBook("B002")	Nullptr	Nullptr	PASS
	Test removing non-existing book fails	removeBookDirect("B999")	False	False	PASS
Borrow Book — Non-Interactive	Test borrowing available book succeeds	borrowBookDirect(user, "B003", 1, 2)	True	True	PASS
	Test borrowing same book with overlapping time fails	borrowBookDirect(user, "B003", 1, 2)	False	False	PASS
	Test borrowing non-existing book fails	borrowBookDirect(user, "B999", 1, 2)	False	False	PASS
	Test borrowing the second book with a valid interval	borrowBookDirect(user, "B004", 1, 2)	True	True	PASS

	succeeds				
	Test borrowing the third book succeeds	borrowBookDirect(user, "B005", 1, 2)	True	True	PASS
	Test borrowing the fourth book fails (user limit reached)	borrowBookDirect(user, "B006", 1, 2)	False	False	PASS
Add and book a room	Test adding a new room succeeds	addRoomDirect("R 001")	True	True	PASS
	Test adding duplicate room fails	addRoomDirect("R 001")	False	False	PASS
	Test booking overlapping interval fails	bookRoomDirect(mockUser, "R001", start, end)	False	False	PASS
	Test booking a non-existing room fails	bookRoomDirect(mockUser, "R999", start, end)	False	False	PASS
	Test booking a non-overlapping future interval succeeds	bookRoomDirect(mockUser, "R001", start2, end2)	True	True	PASS
Remove Room	Test adding a new room succeeds	addRoomDirect("R 002")	True	True	PASS
	Test removing existing room succeeds	removeRoomDirec t("R002")	True	True	PASS
	Test removing the same room again fails	removeRoomDirec t("R002")	False	False	PASS
	Test adding room succeeds	addRoomDirect("R 003")	True	True	PASS

Booking Conflict for Mock User	Test initial booking succeeds	bookRoomDirect(mockUser, "R003", start, end)	True	True	PASS
	Test overlapping booking for the same user fails	bookRoomDirect(mockUser, "R003", startOverlap, endOverlap)	False	False	PASS
	Test non-overlapping booking succeeds	bookRoomDirect(mockUser, "R003", start2, end2)	True	True	PASS
Add and Remove Laptop	Test adding a new laptop succeeds	addLaptopDirect(" L001")	True	True	Pass
	Test adding a duplicate laptop fails	addLaptopDirect(" L001")	False	False	PASS
	Test removing existing laptop succeeds	removeLaptopDire ct("L001")	True	True	PASS
	Test removing a non-existent laptop fails	removeLaptopDire ct("L001")	False	False	PASS
borrowLaptop Direct	Test adding laptop succeeds	addLaptopDirect(" L002")	True	True	Pass
	Test successful first borrow	borrowLaptopDire ct(mockUser, "L002", 1000, 2000)	True	True	Pass
	Test borrowing the same interval fails	borrowLaptopDire ct(mockUser, "L002", 1000, 2000)	False	False	PASS
	Test borrowing non-overlapping interval succeeds	borrowLaptopDire ct(mockUser, "L002", 2001, 3000)	True	True	Pass

	Test borrowing a non-existent laptop fails	borrowLaptopDirect(mockUser, "L999", 1000, 2000)	False	False	PASS
Overlapping laptops for the Same User	Test adding laptops succeeds	addLaptopDirect("L005")	True	True	Pass
	Test first laptop borrow succeeds	borrowLaptopDirect(mockUser2, "L005", 1000, 2000)	True	True	Pass
	Test overlapping borrow on another laptop fails (user conflict)	borrowLaptopDirect(mockUser2, "L006", 1500, 2500)	False	False	PASS
	Test non-overlapping borrow on another laptop succeeds	borrowLaptopDirect(mockUser2, "L006", 2001, 3000)	True	True	Pass
Login and Mock users	Test adding mock users to UsersManager	addUserMock(alice), addUserMock(bob), addUserMock(charlie)	Users stored successfully	Users stored successfully	PASS
	Test successful login for existing user	login("alice", "pass123")	Returns a pointer to user "alice"	pointer returned, username = "alice"	PASS
	Test successful login for the second user	login("bob", "mypassword")	Returns a pointer to user "bob	pointer returned, username = "bob"	PASS
	Test login with the wrong password	login("charlie", "wrong")	nullptr	nullptr	PASS

	Test login with a non-existing username	login("david", "any")	nullptr	nullptr	PASS
Admin and Normal User Detection	Test admin user login	login("adminuser", "adminpass")	non-null pointer, isAdmin = true	non-null pointer, isAdmin = true	PASS
	Test normal user login	login("normaluser", "userpass")	non-null pointer, isAdmin = false	non-null pointer, isAdmin = false	PASS

7 Limitations and Future Improvements

7.1 Limitations After Implementing the Priority Queue

Our initial plan was to integrate a priority queue that ensures users receive either the newest laptop or the one that has been used the least. However, fully adopting this approach would require significant restructuring of the LaptopsManager. Currently, laptops are represented only by IDs, but a proper priority-queue system would require each laptop to be represented as a complete class with attributes such as usage counters (e.g., number of times borrowed and duration of use). These attributes would then determine each laptop's priority.

Because this feature is still under development, our current system has a limitation: it assigns any available laptop randomly rather than prioritizing newer or less-used devices. This affects our ability to guarantee optimal service quality to users.

7.2 Future Improvements for the Priority Queue Implementation

We plan to introduce a priority queue that tracks how many times each laptop has been borrowed, implemented using a min-heap. This will allow us to always provide users with the best, newest, or least-used laptops first. This enhancement will ensure a more efficient and user-centered allocation process.

7.3 Limitations faced after implementing the Queue

During the initial design, we considered implementing a Queue data structure to handle multiple user commands in a single session, following the First-In-First-Out (FIFO) principle. The idea was to allow users to enter several commands at once and have them processed sequentially. However, after implementing the system and running the program, it became clear that this approach was impractical and user-unfriendly. Some key challenges included:

1. Complex error handling: If a user entered multiple commands at once and several conflicts or errors occurred, such as room overlaps, unavailable books, all laptops borrowed, etc. So, managing and presenting all errors simultaneously would have been confusing.
2. User experience issues: Users would have had to navigate through multiple error messages and make corrections, or just log out and start all over again, which is less intuitive than handling each command individually.
3. Increased complexity: Implementing and maintaining the Queue-based system would have required additional logic and memory management, adding unnecessary complexity without significant user benefit.

7.4 Future Improvements for the queue implementation

Although the Queue was excluded in the final implementation, it could still have potential uses in enhancing user experience. For example:

1. Notifications and history: A Queue could be used to maintain a detailed log of all user actions, allowing the system to show a history of the user's commands, including confirmations and error messages.
2. Optimized dynamic resizing of the Queue: In the initial design, the Queue size was set to double whenever it became full. While this ensures that the Queue can handle additional commands, it can lead to memory wastage, especially if the Queue grows large. A better approach would be to increase the size by a smaller factor, such as 1.1×, to balance between accommodating more commands and efficient memory usage.

7. 5 Future Improvements to HashMap

We note that despite the low time complexity associated with HashMap operations, we note that it still requires further improvements. For instance, in case our data set expanded (reaching thousands, if not millions, of students/faculty/staff/alumni), the HashMap will be inefficient. Therefore, we consider scaling database management by introducing SQL/NoSQL databases, such as PostgreSQL and MySQL, to handle our dataset effectively.

7.6 Future Improvements to Interval Tree

While the Interval Tree works well for managing bookings in our current system, it could be improved for larger-scale use. Currently, all booking data is stored in memory, so when the program closes, all bookings are lost. In the future, we could store intervals in a database (e.g., SQLite or PostgreSQL) and load them when the program starts. This would allow the system to handle larger numbers of bookings and multiple resources more efficiently.

Additionally, our program currently only supports one user at a time. If we were to expand it to multiple simultaneous users, conflicts could occur when updating the tree or checking for overlapping bookings. This could be solved by letting the database handle concurrent booking requests safely.

7.7 Allowing Cancellations of Bookings

A planned improvement to the system is the introduction of a cancellation feature that allows users to cancel their existing laptop bookings. Currently, once a reservation is made, it cannot be modified or withdrawn through the interface, which can lead to unused or blocked time slots. Implementing a cancellation option - subject to reasonable rules or time limits - will free up laptops for other users, improve overall resource utilization, and enhance the system's flexibility and user experience.

7.8 Validating All User Input

We also plan to have the program validate all special cases of wrong user inputs, especially in the CLI version

For example,

Our program in the future should be able to send a warning message whenever a user:

- Enter a date or time that is before the current date and time
- Handling the possibility of users booking rooms in two consecutive time frames (which manipulates the 3-hour rule).
- We also plan to refine the borrowing time frame by limiting the dates that appear in the GUI so users cannot mistakenly select invalid options. Although users already receive a warning when attempting to reserve more than one month in advance, the interface will be improved to display only valid and available dates. This enhancement will prevent

accidental input errors, improve usability, and ensure a smoother and more accurate reservation process.

8 Contribution Breakdown

This section briefly mentions each member's contribution in the group project

Sedra Elkhayat:

- Queue Implementation
- User class initial Implementation
- UserManager initial class implementation
- Test cases

Radwa AbuElfotouh:

- Priority queue implementation
- Test cases
- Manual Testing

Hamdy El-Madbouly:

- HashMap and LinkedList implementation.
- Code/System refactoring after each development cycle.
- Handling syntax/logical errors in the program through both C++ and Python.
- CLI enhancements.
- GUI creation (using Vibe Coding on Gemini Pro 3 (High) AI model).

Malk Metwally:

- Interval Tree class implementation - first using arbitrary numbers for intervals
- Initial basic structure and flow of program and CLI interface - user, books, and rooms.txt files, as well as user, book, room, and laptop class
- Changing arbitrary dates and times to seconds passed per reference point where reference point was 00:00 01 January 2025
- Updating Interval tree class - to use times and dates instead of arbitrary numbers for the intervals making use of the reference point

Bibliography

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, and E. Al, *Introduction to Algorithms*. Cambridge (Massachusetts); London: MIT Press; Boston, etc., 2007.

[2] “Epoch (computing),” *Wikipedia*, Apr. 30, 2023. https://en.wikipedia.org/wiki/Epoch_(computing)
[3] "Interval Tree," GeeksforGeeks, July. 23, 2025. https://www.geeksforgeeks.org/dsa/interval-tree/
[4] “11 4 Interval Search Trees 1347”, YouTube, Nov. 23, 2012. https://www.youtube.com/watch?v=q0QOYtSsTg4
[5] “Red Black Tree Operations ,” YouTube, Feb. 12, 2025. https://www.youtube.com/watch?v=yiM8VlIJwms
[6] “Red-Black Tree Insertion Algorithm,” YouTube, Feb. 23, 2025. https://www.youtube.com/watch?v=yNP5OZGnQsw

Appendix

[1] Radwa-Ayman239, Hamdy410, SedraElKhayat, and MalkMetwally, “GitHub - radwa-ayman239/ads-project,” *GitHub*, 2025. https://github.com/Radwa-Ayman239/ADS-Project (accessed Dec. 07, 2025).