

**Project Title**

**Weather Report Using Time Series Analysis**

**Team members:**

* RISHABH PARASHER [CB.EN.U4CSE21250]
* T SRI VALLI RAMYA [CB.EN.U4CSE21263]

**Introduction to Hybrid Data Structures:**

The data structures that are formed by combining the characteristics and functionalities of more than one data structure to meet the requirement of the current problem are known as hybrid data structures. They are designed to make use of the advantages of various data structures by minimizing their drawbacks. As the hybrid data structures are customized for specialized scenarios, they provide us with more adaptability by allowing them to handle problems efficiently in dynamic (real-time) environments and reduces complexity as each of these problems are distributed and handled more effectively.

**Project’s objective:**

To design and implement a hybrid data structure that includes a Binary Search Tree, a priority queue using a heap, to provide us with an analysis of the data obtained of different locations that are recorded. The weather report that is produced by time series analysis using the above-mentioned data structures is useful for:

* Short-term weather prediction such as humidity, precipitation, wind patterns, etc.,
* Long-term climate modeling by analyzing the historical climate data over a span of time
* Agricultural planning to decide what is the optimal timing for planting and harvesting etc.,

**Overview of the Hybrid Data Structure:**

The hybrid data structure that is chosen for the implementation is BST + priority queue. The composition of the hybrid data structure involves maintaining BST and priority queue where one is used to store and retrieve historical data while the other is used to manage and maintain real-time data. The specific advantages and motivation for choosing a hybrid data structure of BST + priority queue is mainly the trade-off between time and space complexity as the same data structures when used separately are much worse and fail to give a combined analysis of historical and real-time data.

**Implementation Details:**

Firstly, the function inside Binary Search Tree (BST) class is used to store the timestamp and weather data and we use this data structure to make it easy to map the attributes from a given API. The ‘insert’ method inserts a new node in the BST based on the ‘timestamp’ and the BST also contains a ‘search\_range\_method’ performs an In-Order Traversal to search through the BST, to find and return the weather data that is within the range of mentioned timestamp.

Secondly, there is ‘RealtimeDataQueue’ which is a priority queue that is used with the ‘heapq’ module. It is used to add weather data to the queue by increasing its timestamp. In the ‘remove’ function it returns the earliest weather timestamp.

Thirdly, in the ‘WeatherForecast’ class, manages the overall weather forecasting functioning. There are a bunch of functionalities of this class that are to be illustrated, they are:

* add\_location: To add new locations to the dictionary based on which it is associated to the BST
* store\_historical\_data: To insert historical weather data into the specific location mentioned
* fetch\_realtime\_weather: To use the ‘Weather API’ to fetch the real-time weather\_data for the given location
* receive\_realtime\_data: To use the ‘fetch\_realtime\_weather’ function to get real-time data and then enqueues it to the ‘RealtimeDataQueue’

Overall, the storing, retrieval, and analysis of both historical and real-time weather data in the WeatherForecast class are made efficient by the integration of the BST and priority queue (heapq).

**The trade-offs that were made during the implementation phase are:**

* In the sample use, the location and API key are hardcoded. Although the code is made simpler, this reduces the application's flexibility and may expose sensitive information
* The Priority queues guarantees that the data is kept sorted according to timestamps and offers effective enqueue and dequeue actions. Priority queues, however, take up more capacity than straightforward data structures like lists or arrays.
* In the worst-case scenario, it can result in an imbalanced tree and poor performance. To assure constant results, one alternative is to use a self-balancing tree such an AVL tree or a Red-Black tree.

**GitHub Repo link**:

https://github.com/idlidosasambar/DSAWeatherPredicterCST

**Practical Applications:**

There are numerous practical applications possible with the weather forecast using Time Series Analysis. Some of the major applications are as follows:

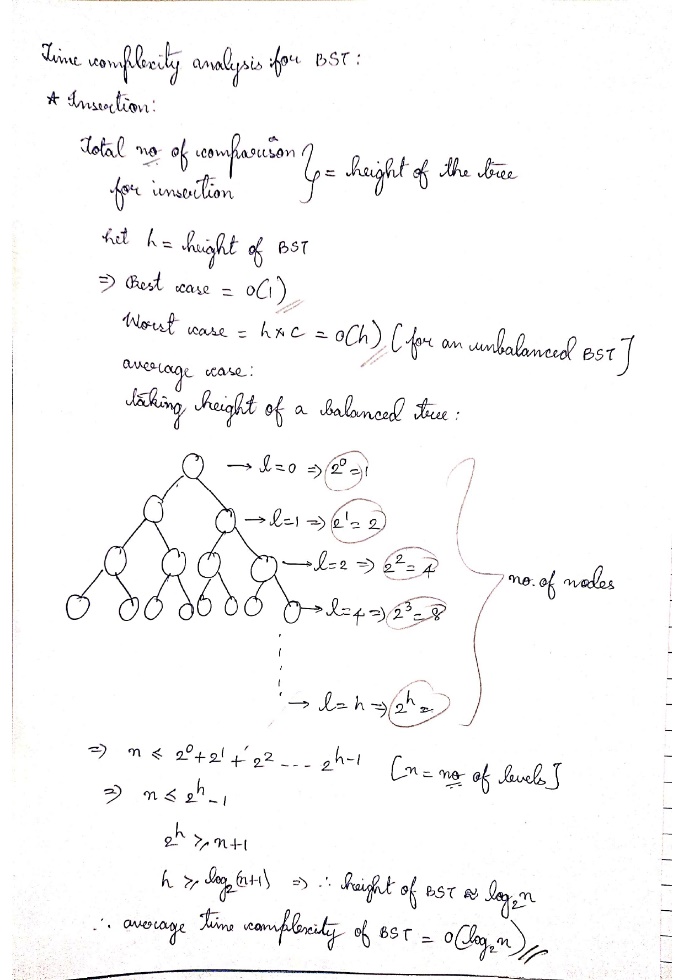
* Decision Support Systems: In the agriculture industry, the planting and harvesting of crops can be planned appropriately with the help of the up-to-date information provided by the priority-queue as the operations in agriculture are largely weather dependent
* Disaster Management and Emergency Response: At the time of natural disasters like Cyclone, Flooding caused due to heavy rainfall- effective decision-making is possible with accurate weather predictions
* Climate Research and Studies: Large-scale temporal assessments of historical weather data are frequently used in climate research. In order to extract pertinent subsets of data depending on certain time periods or interesting places, researchers can efficiently store and retrieve historical data using the hybrid data structure.

“Efficient storage” and “faster retrieval” are possible through BST, storing and letting us access the historical data and priority queue with ‘weather data’ being stored in chronological order based on the timestamps helps us to get faster access to the live weather updates. The logarithmic search complexity of the BST and the constant-time access of the priority queue make them “scalable” for handling large datasets and high-frequency updates. “Flexibility” and “Adaptability” is also achieved by accommodating various weather data and supporting a varied range of operations such as visualization through weather maps, searching, averaging, etc.,

**Performance Analysis:**

Time complexity of the operations in BST:

* Insertion:



* Search function (‘search\_range’): This is similar to the calculations of the insertion function in BST but not same. Therefore,

Best Case: O(1) [Fully balanced tree – occurs when the root node is the required node]

Worst Case: O(n) [Fully unbalanced tree – occurs when the leaf node is the required node]

Average time case: O(log(n) + k), where

n = number of nodes in the tree

k = number of nodes in the specified range

The k is added to log(n) because of filtering within the specified range based on the timestamps within the code.

The space complexity of both the insertion and search operations in BST is O(n)

Time complexity of the priority queue ‘RealTimeDataQueue’:

* Enqueue:

Best Case: O(1) [Occurs when the element that is being inserted has the highest priority and is made the new root of the heap]

Worst Case: O(log(n)) [Occurs when the element that is being inserted, has the least priority and is inserted at the leaf level. To retain the heap property, this calls for comparing the element with its parent and sometimes making swaps.] O(log(n)) because log(n) is the height of the binary heap

Average Case: O(log(n))

* Dequeue:

Best Case: O(1) [Occurs when the element that is being deleted has the highest priority and is made the new root of the heap]

Worst Case: O(log(n)) [Occurs when the element that is being removed, has the least priority and is removed at the leaf level. To retain the heap property, this calls for comparing the element with its parent and sometimes making swaps down the tree] O(log(n)) because log(n) is the height of the binary heap

Average Case: O(log(n))

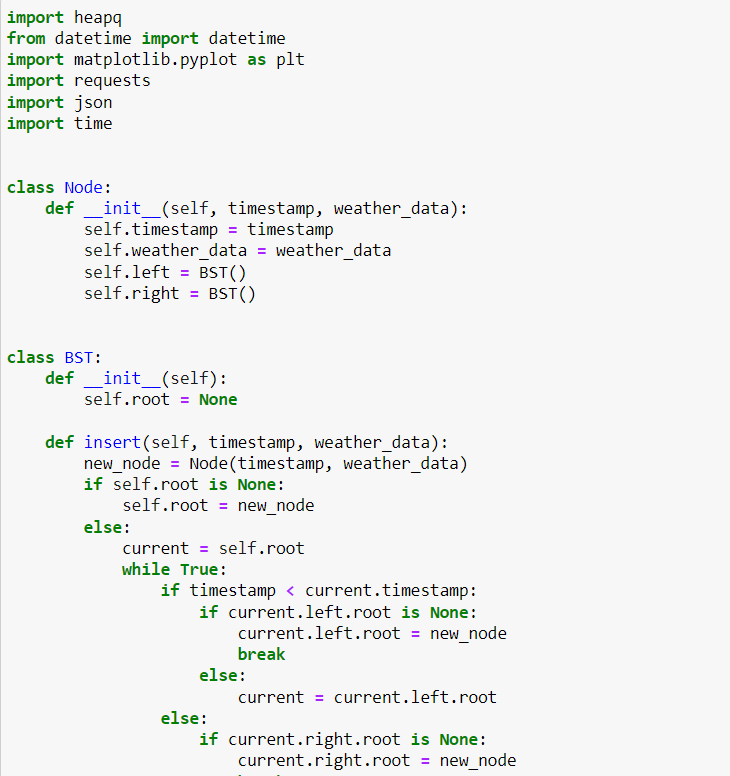
The space complexity for enqueue and dequeue operations in the priority queue is O(n), where n = number of nodes in the priority queue

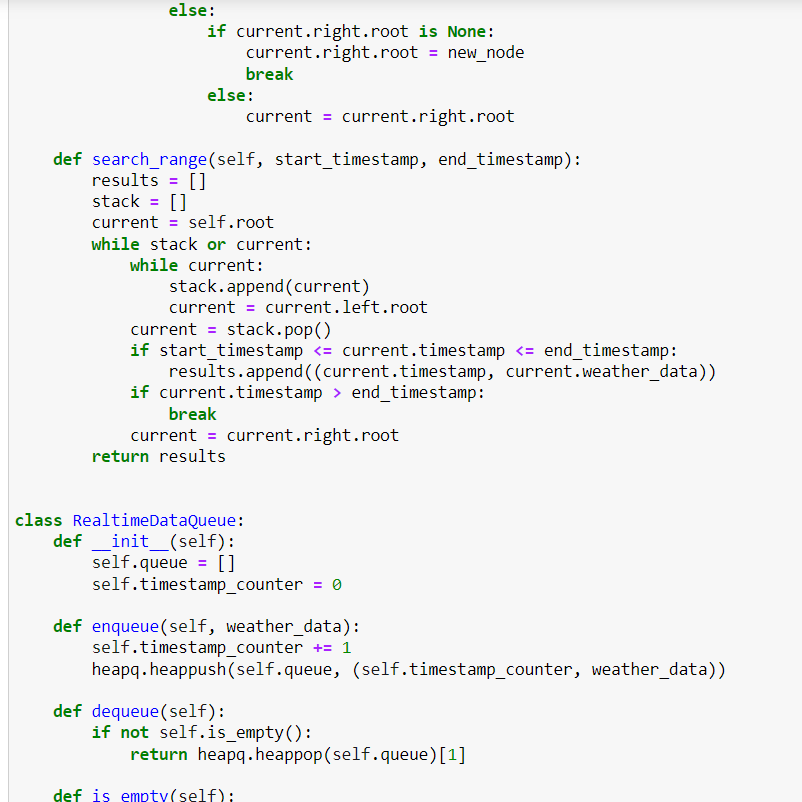
[As it is being implemented using an array and thus each element is stored when enqueued and remains the same for dequeue operation as no additional memory is required while removing an element from the priority queue only reorganization is required]

**Experimental Evaluation:**

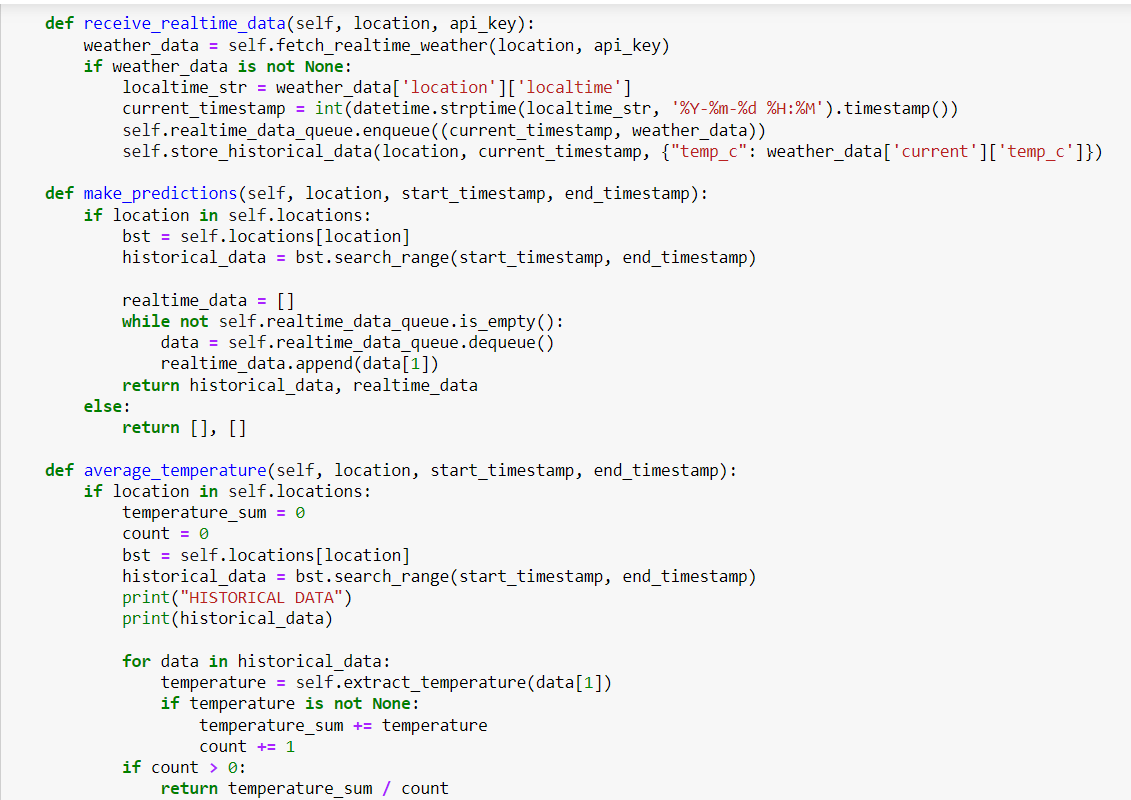
The data sets that are used are the real time data that is taken using an API i.e ‘https://www.weatherapi.com/'

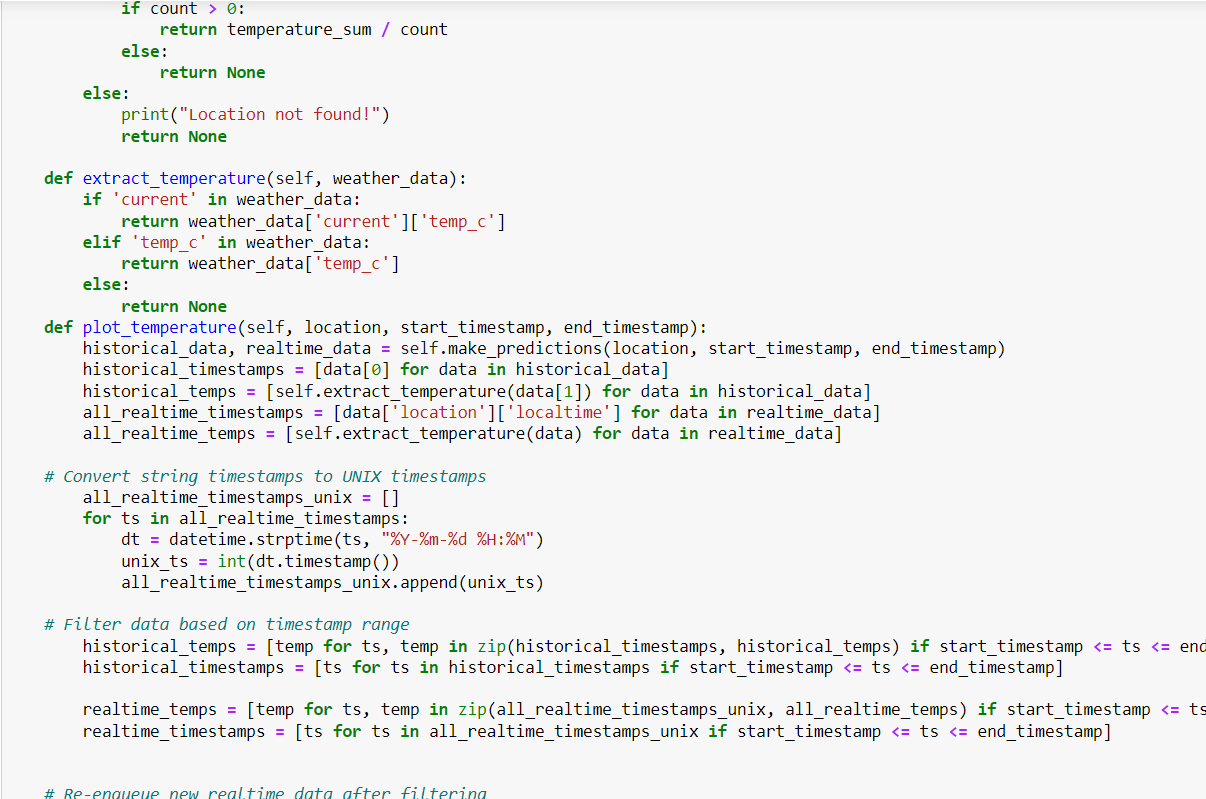
Implementation - code:



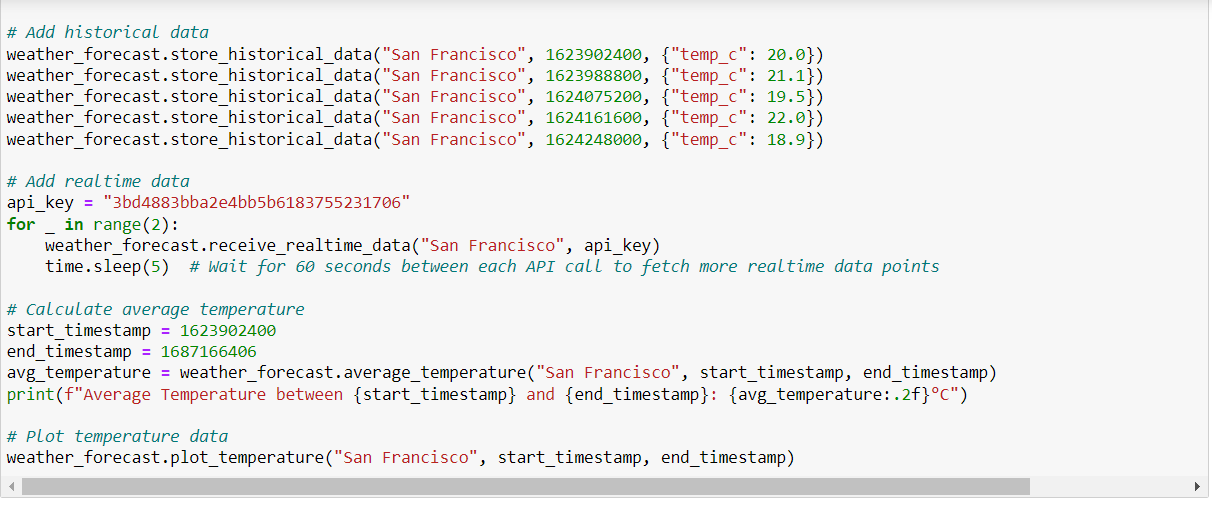




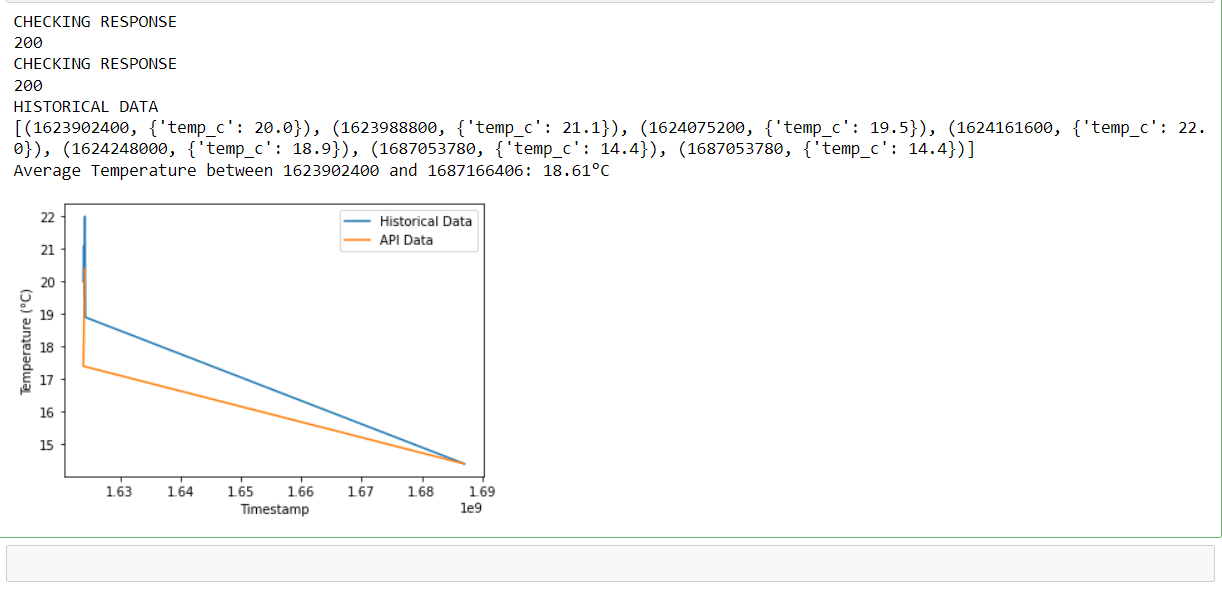








Output:

**Discussion:**

As mentioned earlier the implemented hybrid data structure is very practically applicable in real-world scenarios, especially the average temperature calculation facilitates analyzing temperature patterns, locating patterns, and offering insights on temporal temperature changes, also the code being able to generate temperature plots, aids in the data analysis, forecasting, and decision-making processes.

Following the merits, there also exist demerits like the binary search nature of the BST by restricting range queries to a single continuous interval, even though the priority queue effectively handles real-time data updates, but as the amount of incoming data grows, scaling might become problematic. The temporal complexity of enqueue and dequeue operations may deteriorate as the queue gets bigger and the BST not being self-balanced might increase the time-complexity of operations which can be corrected by using Red-Black Trees.

**Conclusion:**

The project is implemented to provide a weather report using time series analysis, to calculates average temperature within a given timestamp range, real-time weather data is handled effectively by the priority queue by giving priority to the most recent data. We were successfully able to store and retrieve real-time data from API and compare it with historical data. This enabled us to calculate the average temperature and the comparison between API data (real-time data) and historical data is showcased with the help of a graph. Within a certain temporal range, the BST component enables effective search and retrieval of previous weather data. For applications that need to retrieve weather data for analysis or forecasting, this makes it useful.

The limitations of the currently implemented hybrid data structure are:

* Limited Historical Data Storage: As the code can be less efficient in terms of search and insertion operations as the historical data increases
* Updation and deletion of data are the aspects that aren’t focused upon apart from storage and retrieval

Further, the improvements can be made-

* By addressing the drawbacks of reduced efficiency due to large datasets
* By enhancing the ability of the code by adding data updation and deletion operations
* By enhancing the implementation by making it capable to integrate multiple API’s and not just one specific API