[The Rationale on the selected data structure 4](#_Toc162724388)

[Pro and Cons 4](#_Toc162724389)

[UML 5](#_Toc162724390)

[High Level UML 5](#_Toc162724391)

[Low Level UML 6](#_Toc162724392)

[Classes 8](#_Toc162724393)

[Date Class 8](#_Toc162724394)

[Time Class 9](#_Toc162724395)

[Measurement class 9](#_Toc162724396)

[WindSpeed class 10](#_Toc162724397)

[SolarRadiation class 10](#_Toc162724398)

[Temperature class 11](#_Toc162724399)

[SensorMeasurementType class 11](#_Toc162724400)

[SensorRecType class 12](#_Toc162724401)

[Vector class 13](#_Toc162724402)

[Tree class 14](#_Toc162724403)

[BST class 15](#_Toc162724404)

[AVL class 15](#_Toc162724405)

[Processor class 16](#_Toc162724406)

[AoAvlMap class 17](#_Toc162724407)

[InputValidator class 18](#_Toc162724408)

[Menu class 18](#_Toc162724409)

[Unit Testing 19](#_Toc162724410)

[Date class 19](#_Toc162724411)

[Time class 21](#_Toc162724412)

[Measurement class 22](#_Toc162724413)

[WindSpeed class 22](#_Toc162724414)

[SolarRadiation class 23](#_Toc162724415)

[Temperature class 24](#_Toc162724416)

[SensorMeasurementType class 24](#_Toc162724417)

[Vector class 25](#_Toc162724418)

[Tree class 26](#_Toc162724419)

[BST class 26](#_Toc162724420)

[AVL class 28](#_Toc162724421)

[Processor class 30](#_Toc162724422)

[AoAvlMap class 30](#_Toc162724423)

[InputValidator class 31](#_Toc162724424)

[Menu class 32](#_Toc162724425)

[Algorithm 33](#_Toc162724426)

[Date class 33](#_Toc162724427)

[Operator == ( otherDate ) 33](#_Toc162724428)

[Operator < (otherDate) 33](#_Toc162724429)

[Operator != ( otherDate ) 33](#_Toc162724430)

[Operator > (otherDate) 33](#_Toc162724431)

[Time class 34](#_Toc162724432)

[Operator == ( otherTime ) 34](#_Toc162724433)

[Operator < (otherTime) 34](#_Toc162724434)

[Operator != ( otherTime ) 34](#_Toc162724435)

[Operator > (otherTime) 34](#_Toc162724436)

[WindSpeed class 34](#_Toc162724437)

[SolarRadiation class 34](#_Toc162724438)

[ConvertUnit( measurement ) 34](#_Toc162724439)

[ConvertUnit( measurement ) 34](#_Toc162724440)

[Temperature class 35](#_Toc162724441)

[ConvertUnit( measurement ) 35](#_Toc162724442)

[SensorRecType class 35](#_Toc162724443)

[Operator == ( other ) 35](#_Toc162724444)

[Operator < (other) 35](#_Toc162724445)

[Operator > ( other ) 35](#_Toc162724446)

[Tree class 36](#_Toc162724447)

[Destroy(ptr) 36](#_Toc162724448)

[GetMaxTreeSubHeight(left, right) 36](#_Toc162724449)

[GetHeight(ptr) 36](#_Toc162724450)

[GetNodes(ptr) 36](#_Toc162724451)

[GetLeaves(ptr) 36](#_Toc162724452)

[CopyTree(copiedTreeRoot, otherTreeRoot) 36](#_Toc162724453)

[Tree(otherTree) 37](#_Toc162724454)

[Operator = (otherTree) 37](#_Toc162724455)

[PreOrder( traversal, ptr ) 37](#_Toc162724456)

[InOrder( traversal, ptr ) 37](#_Toc162724457)

[PostOrder( traversal, ptr ) 38](#_Toc162724458)

[Search(current, object) 38](#_Toc162724459)

[AVL class 38](#_Toc162724460)

[GetBalanceFactor(ptr) 38](#_Toc162724461)

[RightRotate(rst) 38](#_Toc162724462)

[LeftRotate(lst) 38](#_Toc162724463)

[INSERT(current, object) 39](#_Toc162724464)

[Delete(current, object) 40](#_Toc162724465)

[AoAvlMap class 42](#_Toc162724466)

[InsertSensorData( sensorData) 42](#_Toc162724467)

[InOrderTraversalSum(root, sensorMeasurement, sum) 42](#_Toc162724468)

[InOrderTraversalSumAndSquareSum(root, sensorMeasurementType, sum, sumSquare) 43](#_Toc162724469)

[CalculateSampleStandardDeviation(root, SensorMeasurementType, count) 43](#_Toc162724470)

[FindHighestSolarRadiation(day, month, year) 43](#_Toc162724471)

[Processor class 44](#_Toc162724472)

[LoadCSVFilePathToVector(Vector<string> &csvFilePath, ifstream &input, const string &filename) 44](#_Toc162724473)

[LoadCSVData(Vector<string> &csvFilePath, AoAvlMap<SensorRecType> &mapAoBst) 44](#_Toc162724474)

[References 46](#_Toc162724475)

# The Rationale on the selected data structure

The selected data structure comprises a map where each key represents the year of a date, and the corresponding value is an array of AVL trees holding the months of that year. Each AVL tree node holds the data for a specific date, including time and sensor measurements.

The chosen data structure of a map is used for organizing data by year, facilitating rapid retrieval based on the year with a constant time complexity of O(1). This approach ensures efficient access to data. Additionally, implementing AVL trees for each month in the array within a year guarantees balanced search, insertion, and deletion operations. AVL trees maintain logarithmic time complexity, ensuring efficient management of data within each month. Furthermore, the inherent sorted order of AVL trees enables organized storage of dates and time within each month. This ordered storage system simplifies data processing tasks, enhancing overall efficiency. Moreover, storing data within AVL tree nodes reduces memory overhead compared to alternative structures for each data point. This compact representation optimizes memory usage, particularly beneficial for managing large datasets efficiently.

The preference of using the AVL trees over binary search trees (BSTs) arises from their inherent property of self-balancing, ensuring consistent performance in search, insertion, and deletion operations. AVL trees maintain a balanced height, guaranteeing that the tree remains relatively balanced, which is the critical reason behind this choice. This self-balancing feature optimizes data management within each month of the dataset, preventing performance degradation and ensuring efficient memory utilization. Moreover, AVL trees maintain sorted order, facilitating ordered storage of dates and time within each month.

## 

## Pro and Cons

The chosen data structure combines the efficiency of a map and AVL trees to enable swift retrieval of data based on year, month. Utilizing a map allows for direct access to data based on the year, and direct access to the month using an array. While AVL trees ensure balanced searching, and insertion operations within ear year and each month, maintaining consistent performance. Additionally, the inherent ordering of AVL trees simplifies data processing, as dates are stored in sorted order within each year and each month. This organized storage enhances the efficiency of operations such as range queries. Moreover, storing data within AVL tree nodes reduces memory overhead, optimizing memory usage, and making the structure space-efficient, particularly advantageous for managing large datasets effectively. Overall, the combination of map and AVL trees offers a robust solution for efficient, balanced, and ordered storage of data.

The implementation of AVL trees within the data structure introduce additional complexity to the codebase, especially concerning insertion operations, which require careful handling to maintain the tree's balanced structure. Despite being self-balancing, AVL trees may become imbalanced due to improper implementation, potentially impacting performance. Moreover, the overhead of AVL tree nodes may outweigh the benefits of balanced searching, leading to reduced efficiency. Additionally, while the structure is efficient for moderate-sized datasets, its scalability may be limited for extremely large datasets due to memory constraints and potential performance bottlenecks.

# UML

## High Level UML

A diagram of a computer

Description automatically generatedA diagram of a diagram

Description automatically generated

## Low Level UML

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

# Classes

## Date Class

This class enables easy manipulation of dates, allowing access and modification of individual components (day, month, year). It supports comparison operations and overloads comparison operators for chronological checks and equality. The GetMonthInStr method provides month names for readability.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| Date | Class |  | Represents a Date Class. |
| m\_day | unsigned | - | An unsigned day of a date. |
| m\_month | unsigned | - | An unsigned month of a date. |
| m\_year | unsigned | - | An unsigned year of a date. |
| Date() | procedure | + | Sets the day, month, and year to 0. |
| Date(day, month, year) | procedure | + | Sets a specific day, month, and year. |
| GetDay() | query | + | Gets the day of a date. |
| GetMonth() | query | + | Gets the month of a date. |
| GetYear() | query | + | Gets the year of a date. |
| SetDay(unsigned day) | procedure | + | Sets the day of a date. |
| SetMonth(unsigned month) | procedure | + | Sets the month of a date. |
| SetYear(unsigned year) | procedure | + | Sets the year of a date. |
| operator == ( const Date &otherDate ) | query | + | operator checks whether two Date objects have the same day, month, and year. |
| operator != ( const Date &otherDate ) | query | + | operator checks whether two Date objects does not have the same day, month, and year. |
| operator < ( const Date &otherDate ) | query | + | operator checks whether this Date object is less than other Date object. |
| operator > ( const Date &otherDate ) | query | + | operator checks whether this Date object is greater than other Date object. |
| &operator << (ostream, date) | query | + | Format Date object, inserts into output stream. |
| &operator >> (istream, date) | query | + | Reads a Date Object from input stream. |

## Time Class

The Time class simplifies time management by encapsulating hours and minutes. It allows setting, retrieving, and comparing time components. Constructors initialize objects with default or specific values. Methods GetHour and GetMinute access time components, while SetHour and SetMinute modify them. Comparison operators enable chronological comparisons.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| Time | Class |  | Represents a Time Class. |
| m\_hour | unsigned | - | An unsigned hour of a time. |
| m\_minute | unsigned | - | An unsigned minute of a time. |
| Time() | procedure | + | Sets the hour, and minute to 0. |
| Time(hour, minute) | procedure | + | Sets a specific hour, and minute. |
| GetHour() | query | + | Gets the hour of a time. |
| GetMinute() | query | + | Gets the minute of a time. |
| SetHour(unsigned hour) | procedure | + | Sets the hour of a time. |
| SetMinute(unsigned minute) | procedure | + | Sets the minute of a time. |
| operator == ( const Time &otherTime ) | query | + | operator checks whether two Time objects have the same hour and minute. |
| operator != ( const Time &otherTime ) | query | + | operator checks whether two Time objects does not have the same hour, and minute. |
| operator < ( const Time &otherTime ) | query | + | operator checks whether this Time object is less than other Time object. |
| operator > ( const Time &otherTime ) | query | + | operator checks whether this Time object is greater than other Time object. |
| &operator << (ostream, time) | query | + | Format Time object, inserts into output stream. |
| &operator >> (istream, time) | query | + | Reads a Time Object from input stream. |

## Measurement class

The ‘Measurement’ class serves as the base for windspeed, temperature, and solar radiation measurements. It manages measurement values from sensor readings, offering methods for setting, retrieving, and converting units. Constructors and getter/setter functions provide flexibility for different measurements. Overloaded stream operators facilitate I/O operations, while the `ConvertUnit` function allows custom conversions tailored to specific measurement types. This versatile framework is essential for processing sensor data and scientific measurements, ensuring precise handling and conversion capabilities across various applications.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| Measurement | Class |  | Represents an Observations Class. |
| m\_measurement | float | - | A float measurement of an observation. |
| Measurement() | procedure | + | Sets the measurement to 0.0 |
| Measurement(measurement) | procedure | + | Sets a specific measurement. |
| GetMeasurement() | query | + | Gets the measurement of an observation. |
| SetMeasurement(measurement) | procedure | + | Sets the measurement of an observation. |
| virtual float ConvertUnit( float &measurement ) const = 0 | virtual | + | pure virtual function is intended to be overridden by derived classes |

## WindSpeed class

The ‘WindSpeed’ class defines objects representing wind speed measurements. As a subclass of the ‘Measurement’ class, it inherits functionality for managing measurement values, including setting, retrieving, and converting units. The class provides constructors for creating ‘WindSpeed’ objects with default or specified wind speed measurements. Additionally, it includes a method ‘ConvertUnit’ for converting wind speed readings to kilowatt-hours per square meter (kWh/m2), offering flexibility in unit representation. Overloading the stream insertion operator allows seamless output of ‘WindSpeed’ objects. ‘WindSpeed’ class serves as a specialized framework for handling wind speed data, facilitating precise measurement management and conversion capabilities.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| WindSpeed | Class |  | Represents a WindSpeed Class. |
| WindSpeed() | procedure | + | Sets the wind speed measurement to 0.0 |
| WindSpeed(measurement) | procedure | + | Sets a specific wind speed measurement. |
| float ConvertUnit( float &measurement ) const override | override method | + | Converts windspeed reading to kWh/m2. |
| &operator << (ostream, W) | query | + | Format WindSpeed object, inserts into output stream. |
| &operator >> (istream, W) | query | + | Reads a WindSpeed Object from input stream. |

## SolarRadiation class

The ‘SolarRadiation’ class encapsulates solar radiation measurements, inheriting functionality from the ‘Measurement’ class. It facilitates creation of objects with default or specified solar radiation values through its constructors. The ‘ConvertUnit’ method efficiently converts solar radiation readings aiding in unit conversion. Additionally, the class overloads the stream insertion operator for effortless output of ‘SolarRadiation’ objects. With these features, the ‘SolarRadiation’ class offers a robust framework for managing solar radiation data with accuracy and flexibility in handling and conversion.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| SolarRadiation | Class |  | Represents a SolarRadiation Class. |
| SolarRadiation () | procedure | + | Sets the solar radiation measurement to 0.0 |
| SolarRadiation(measurement) | procedure | + | Sets a specific solar radiation measurement. |
| float ConvertUnit( float &measurement ) const override | query | + | Used for converting unit of solar radiation reading. |
| &operator << (ostream, SR) | query | + | Format SolarRadiation object, inserts into output stream. |
| &operator >> (istream, SR) | query | + | Reads a SolarRadiation Object from input stream. |

## Temperature class

The ‘Temperature’ class serves to handle temperature measurements and extends functionalities inherited from the ‘Measurement’ class. It facilitates object creation with default or custom temperature values through its constructors. Utilizing the ‘ConvertUnit’ method, it adeptly converts temperature readings, enhancing versatility in unit representation. Moreover, by overloading the stream insertion operator, it enables effortless output of ‘Temperature’ objects. With these capabilities, the ‘Temperature’ class provides a robust framework for managing temperature data, ensuring accurate handling and conversion.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| Temperature | Class |  | Represents a Temperature Class. |
| Temperature () | procedure | + | Sets the temperature measurement to 0.0 |
| Temperature(measurement) | procedure | + | Sets a specific temperature measurement. |
| float ConvertUnit( float &measurement ) const override | query | + | Used for converting unit of temperature reading. |
| &operator << (ostream, TEMP) | query | + | Format Temperature object, inserts into output stream. |
| &operator >> (istream, TEMP) | query | + | Reads a Temperature Object from input stream. |

## SensorMeasurementType class

The ‘SensorMeasurementType’ class defines an enum for different sensor measurements for wind speed, ambient temperature, and solar radiation. Employing a factory pattern, it allows easy integration of new measurement types, this promotes flexibility for the sensor readings.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| SensorMeasurementType | Class |  | Represents a SensorMeasurementType enum Class. |
| WIND\_SPEED | enum constant | + | Identifier for wind speed measurement. |
| AMBIENT\_TEMPERATURE | enum constant | + | Identifier for ambient temperature measurement. |
| SOLAR\_RADIATION | enum constant | + | Identifier for solar radiation measurement. |

## SensorRecType class

The ‘SensorRecType’ class serves as a comprehensive representation of sensor records, encompassing vital components such as date, time, wind speed, solar radiation, and temperature. By providing methods for both retrieval and setting of these attributes, it facilitates seamless management of sensor data. Offering a versatile framework for handling sensor records, the class ensures precise manipulation and efficient data management tasks. With its ability to encapsulate and manage crucial information, the ‘SensorRecType’ class plays a pivotal role in effectively handling sensor data in various applications.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| SensorRecType | Class |  | Represents a SensorRecType Class. |
| m\_date | Date | - | A Date of a date Type. |
| m\_time | Time | - | A Time of a time Type. |
| m\_windSpeed | WindSpeed | - | A WindSpeed of a SensorRecType. |
| m\_solarRadiation | SolarRadiation | - | A SolarRadiation of a SensorRecType. |
| m\_temperature | Temperature | - | A Temperature of a SensorRecType. |
| SensorRecType() | procedure | + | Sets the date, time, windspeed, solar radiation and temperature to default value. |
| SensorRecType(date, time, windspeed, solarRadiation, temperature) | procedure | + | Sets a specific date, time, windspeed, solarRadiation, temperature. |
| GetSensorDate() | query | + | Gets the date of a SensorRecType. |
| GetSensorTime() | query | + | Gets the time of a SensorRecType. |
| GetSensorWindSpeed() | query | + | Gets the wind speed of a SensorRecType. |
| GetSensorSolarRadiation() | query | + | Gets the solar radiation of a SensorRecType. |
| GetSensorTemperature() | query | + | Gets the temperature of a SensorRecType. |
| SetSensorDate (Date date) | procedure | + | Sets the date of a SensorRecType. |
| SetSensorTime (Time time) | procedure | + | Sets the time of a SensorRecType. |
| SetSensorWindSpeed(  WindSpeed windspeed) | procedure | + | Sets the wind speed of a SensorRecType. |
| SetSensorSolarRadiation(  SolarRadiation solarRadiation) | procedure | + | Sets the solar radiation of a SensorRecType. |
| SetSensorTemperature(  Temperature temperature) | procedure | + | Sets the temperature of a SensorRecType. |
| &operator << (ostream, SRT) | query | + | Format SolarRecType object, inserts into output stream. |
| &operator >> (istream, SRT) | query | + | Reads a SolarRecType Object from input stream. |

## Vector class

The ‘Vector’ template class offers a dynamic array implementation, providing fundamental functionalities such as appending objects, accessing elements by index, and resizing the array. With an initial capacity set to two, it initializes a dynamic array on the heap memory and manages the number of used elements. The class includes methods for copying vectors, retrieving capacity and usage statistics, and inserting or removing elements at specified positions. By doubling the capacity when necessary and deallocating memory appropriately, the ‘Vector’ class ensures efficient memory management and seamless manipulation of dynamic arrays.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| Vector | Class<T> |  | Represents a Vector Class. |
| INITIAL\_CAPACITY | unsigned | - | Represents the initial capacity of the vector. |
| INITIAL\_VALUE | unsigned | - | Represents the default initial value. |
| m\_capacity | unsigned | - | Represents the current capacity of the vector. |
| m\_used | unsigned | - | Represents the number of elements used in the vector. |
| m\_array | T\* | - | Represents the pointer to the dynamic array storing vector elements. |
| Vector() | Constructor | + | Default constructor. Initializes the vector with an initial capacity of two and creates a new dynamic array on the heap memory. |
| ~Vector() | Destructor | + | Destructor. Frees the dynamically allocated memory for the vector, sets the capacity and used elements to zero. |
| Vector(const Vector &otherVector) | Copy constructor | + | Copy constructor. Creates a new vector as a copy of another vector. |
| operator=(const Vector &otherVector) | Assignment operator | + | Assignment operator. Assigns the contents of another vector to this vector. |
| GetCapacity() | unsigned | + | Get the current capacity of the vector. |
| GetUsed() | unsigned | + | Get the number of elements used in the vector. |
| IsEmpty() | bool | + | Check if the vector is empty. |
| Start() | T\* | + | Pointer that points to the start of the vector index [0]. |
| Finish() | T\* | + | Pointer that points to the end of the vector index. |
| operator[](const unsigned &index) | T& | + | Access an element by vector's index. |
| operator[](const unsigned &index) const | const T& | + | Const version of access an element by vector's index. |
| InsertLast(const T &object) | bool | + | Insert an object to the end of the vector. |
| InsertAt(const T &object, const unsigned &index) | bool | + | Insert an object to a desired index of the vector. |
| RemoveLast() | bool | + | Remove an object from the end index of the vector. |
| RemoveAt(const unsigned &index) | bool | + | Remove an object from a desired index of the vector. |
| Deallocate(T \*&ptr) | void | - | Deallocate memory for a given pointer. |
| CopyVector(const Vector &otherVector) | void | - | Copy the contents of another vector. |
| CopyArray(T \*copiedArray, T \*otherArray) | void | - | Copy the contents of one array to another. |
| Resize() | void | - | Resize the vector by doubling its capacity. |

## Tree class

The ‘Tree’ class serves as a base class for implementing tree data structures. It provides fundamental functionality for managing data, including methods for insertion, deletion, traversal, and assessment of tree properties such as height, number of nodes, and number of leaf nodes. Leveraging a template-based design, the ‘Tree’ class allows for the storage of diverse data types within its nodes. Additionally, the class offers virtual methods, enabling derived classes to implement specific tree variants of binary search trees or AVL trees.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| Tree | Class<T> |  | Represents a Tree Class. |
| m\_root | Node<T>\* | # | Pointer to the root node of the tree. |
| Tree() | Constructor | + | Default constructor used for creating an empty tree instance. |
| ~Tree() | Destructor | + | Destructor used for releasing allocated memory. |
| Tree(const Tree&) | Copy Constructor | + | Copy constructor used for creating a new tree from another tree. |
| operator= | Overload Equals Assignment Operator | + | Equals assignment operator used for assigning the contents from another tree. |
| IsEmpty() | bool | + | Method to check if the tree is empty. |
| GetRoot() | Node<T>\* | + | Method to get the root node of the tree. |
| GetTreeHeight() | int | + | Method to get the height of the tree. |
| GetTreeNodes() | int | + | Method to get the number of nodes in the tree. |
| GetTreeLeaves() | int | + | Method to get the number of leaf nodes in the tree. |
| InsertNode() | virtual bool | + | Method to insert a new node into the tree. |
| SearchNode() | bool | + | Method to search for a node with a specified object in the tree. |
| DeleteNode() | virtual bool | + | Method to delete a node with a specified object from the tree. |
| PreOrderTraversal() | procedure | + | Method to perform a pre-order traversal of the tree and print the node values. |
| InOrderTraversal() | procedure | + | Method to perform an in-order traversal of the tree and print the node values. |
| PostOrderTraversal() | procedure | + | Method to perform a post-order traversal of the tree and print the node values. |
| Destroy() | procedure | # | Recursively destroy the tree starting from the given node. |
| CopyTree() | procedure | # | Method to copy the contents of one subtree to another. |
| GetMaxTreeSubHeight() | int | # | Method to get the maximum of two integers. |
| GetHeight() | int | # | Method to get the height of a subtree at the given node. |
| GetNodes() | int | # | Method to get the total number of nodes in a subtree at the given node. |
| GetLeaves() | int | # | Method to get the number of leaf nodes in a subtree at the given node. |
| Insert() | virtual bool | # | Method to insert a new node with the specified object into a subtree. |
| Search() | bool | # | Method to search for a node with the specified object in a subtree. |
| Delete() | virtual bool | # | Method to delete a node with the specified object in a subtree. |
| PreOrder() | procedure | # | Method to perform a pre-order traversal of a subtree and print the node values. |
| InOrder() | procedure | # | Method to perform an in-order traversal of a subtree and print the node values. |
| PostOrder() | procedure | # | Method to perform a post-order traversal of a subtree and print the node values. |

## BST class

The ‘BST’ class represents a Binary Search Tree (BST) and is derived from the base class ‘Tree’. It provides functionality to insert and delete nodes in the tree while maintaining the binary search tree property. This class implements the methods for insertion and deletion. It offers operations for managing data in a sorted order.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| BST | Class<T> |  | A template class representing a Binary Search Tree (BST), which is derived from the Tree base class. It provides methods for inserting and deleting nodes in the tree. |
| InsertNode | Method | + | Inserts a node with the given object into the binary search tree (BST). Returns true if the insertion is successful, false if the object already exists in the tree. |
| DeleteNode | Method | + | Deletes a node with the given object from the binary search tree (BST). Returns true if the deletion is successful, false if the object is not found in the tree. |
| ~BST | Destructor | + | Destroys the binary search tree by recursively deleting all nodes. |
| Insert | Method | - | Inserts a new node containing the specified object into the binary search tree (BST). This private method is called recursively from InsertNode. Returns true if the insertion is successful, false otherwise. |
| Delete | Method | - | Deletes a node containing the specified object from the binary search tree (BST). This private method is called recursively from DeleteNode. Returns true if the deletion is successful, false otherwise. |

## AVL class

The ‘AVL’ class represents an AVL tree, which is a self-balancing binary search tree. This class inherits from the `Tree` base class and provides functionality for inserting and deleting nodes while maintaining the AVL tree property, which ensures that the tree remains balanced. It implements the AVL-specific methods for insertion and deletion, including rotations to balance the tree after modifications. The AVL tree offers efficient operations for managing data in a sorted order while keeping the tree height logarithmic, making it suitable for fast retrieval and insertion of sorted data.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| AVL | Class<T> |  | Represents the AVL tree class. |
| ~AVL | Destructor | + | Destructor for the AVL tree. Destroys the AVL tree by recursively deleting all nodes. |
| InsertNode | Method | + | Insert a node with the given object into the AVL tree while maintaining AVL tree property. |
| DeleteNode | Method | + | Delete a node with the given object from the AVL tree while maintaining AVL tree property. |
| GetBalanceFactor | Method | - | Get the balance factor of a given AVL tree node. |
| RightRotate | Method | - | Perform a right rotation on the given AVL tree node. |
| LeftRotate | Method | - | Perform a left rotation on the given AVL tree node. |
| Insert | Method | - | Insert a new node with the given object into the AVL tree while maintaining AVL tree property. |
| Delete | Method | - | Delete a node with the given object from the AVL tree while maintaining AVL tree property. |

## Processor class

The ‘Processor’ class encapsulates functionality for handling CSV data processing with a singleton pattern. It provides methods for loading CSV file paths from a text file into a vector, loading CSV data from the paths into a data structure, and recording sensor data to an output file while computing statistical measurements such as mean, standard deviation, and sum of sensor readings. This class serves as a centralized entity for managing data processing tasks, ensuring a approach to file I/O operations and computation in the context of the sensor data.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| Processor | Class |  | Represents a Processor Class. |
| s\_instance | Processor\* | - | Pointer to the singleton instance of the Processor class. |
| m\_dataRead | unsigned | - | Counter for the number of data read. |
| m\_dataInserted | unsigned | - | Counter for the number of data inserted. |
| Processor() | Constructor | - | Private constructor to prevent instantiation of Processor class objects. |
| LoadDisplayCount() | procedure | - | Displays the count for the Processor instance. |
| Vector<string> Split( const string &aString, char delimiter ) | method | - | splitting the input string into substrings based on the specified delimiter character. |
| GetInstance() | Processor& | + | Returns the singleton instance of Processor. |
| LoadCSVFilePathToVector(Vector<string> &csvFilePath, ifstream &input, const string &filename) | void | + | Loads the CSV file paths from a text file into a vector. |
| LoadCSVData( Vector<string> &csvFilePath, AoAvlMap<SensorRecType> &mapAoBst ) | void | + | Loads CSV data from a vector of file paths into a AoAvlMap object. |
| OutputStreamMeasurement(  SensorRecType &sensorRecType,  ofstream &outputFile,  unsigned &month,  float &speedMean,  float &speedSsd,  float &tempMean,  float &tempSsd,  float &srSum ); | bool | + | Records sensor data to an output file and computes statistical measurements. |
| void ReloadData( AoAvlMap<SensorRecType> &mapAoBst, const string& filename ) | procedure | + | Destroys existing map, then recreates the map following by inserting the data. |

## AoAvlMap class

The ‘AoAvlMap’ class implements a map structure that organizes sensor data by year and month using an array of AVL trees. It provides methods for inserting sensor data into the map, performing in-order traversals to calculate the sum and sample standard deviation of different types of measurements, and finding the highest solar radiation readings for a given date. The `InsertSensorData` method inserts sensor data into the appropriate AVL tree based on the year and month. The `InOrderTraversalSum` and `CalculateSampleStandardDeviation` methods perform in-order traversals of AVL trees to calculate the sum and sample standard deviation of specified measurement types. Finally, the `FindHighestSolarRadiation` method retrieves the highest solar radiation readings for a given date from the AVL tree corresponding to the specified year and month, returning a vector of pairs containing the highest radiation values and their corresponding times.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| AoAvlMap | Class |  | Represents a AoAvlMap Class. |
| NUM\_MTH | const unsigned | + | Number of months in a year. |
| m\_data | map<unsigned, array<AVL<SensorRecType>, NUM\_MTH>> | - | Map storing arrays of sensor data vectors organized by month and year. |
| InOrderTraversalSumAndSquareSum(Node<T>\* root, sensorMeasurementType, float& sum, float& sumSquare) const | procedure | - | perform an in-order traversal of an AVL tree containing sensor data, calculating the sum and sum of squares of a specified measurement type |
| &GetSensorData() | const map<unsigned, array<AVL<T>, NUM\_MTH>> | + | Get a constant reference to the sensor data map. |
| &GetSensorData() | map<unsigned, array<AVL<T>, NUM\_MTH>> | + | Get a reference to the sensor data map. |
| InsertSensorData(const T &sensorData) | bool | + | Insert sensor data into the map consisting of arrays of AVL. |
| InOrderTraversalSum(Node<T>\* root, SensorMeasurementType sensorMeasurementType, float& sum) const | procedure | + | perform an in-order traversal of an AVL tree containing sensor data while concurrently computing the sum and sum of squares of a specified measurement type. |
| CalculateSampleStandardDeviation(Node<T>\* root, SensorMeasurementType sensorMeasurementType, unsigned &count) const; | float | + | method computes the sample standard deviation (SSD) for a given sensor measurement type within the AVL tree represented by the root node. |
| ExtractMeasurements(Node<T>\* root, SensorMeasurementType sensorMeasurementType, Vector<float>& measurements) const | procedure | + | Used for extracting measurements from the tree and inserting it into provided vector |
| CalculateSPCC( const Vector<float>& data1, const Vector<float>& data2 ) const | float | + | Used to calculate the SPCC |
| bool IsEmpty() const | boolean | + | Used to check if the map / bst structure is empty. |
| void AoAvlMap<T>::Destroy() | procedure | + | Used for destroying existing map. |

## InputValidator class

The ‘InputValidator’ class provides functionality for validating user input related to year, month, and day. It ensures that the provided input is within valid ranges and formats. With methods like YearValidation, MonthValidation, and DayValidation, users can input values for these components (Year, Month, and Day) and the class ensures that the inputs are valid. The class also includes private methods to check the validity of individual inputs. This class is essential for ensuring data integrity and preventing errors when working with date-related user inputs.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| InputValidator | Class |  | Represents a InputValidator Class |
| YearValidation | Method | + | Validates user input for year. |
| MonthValidation | Method | + | Validates user input for month. |
| DayValidation | Method | + | Validates user input for day. |
| m\_userYear | Member | - | Holds the user input for year. |
| m\_userMonth | Member | - | Holds the user input for month. |
| m\_userDay | Member | - | Holds the user input for day. |
| IsYearInputValid | Private | - | Checks if the user input for year is valid. |
| IsMonthInputValid | Private | - | Checks if the user input for month is valid. |
| IsDayInputValid | Private | - | Checks if the user input for day is valid. |

## Menu class

The ‘Menu’ class provides a interface for analyzing the data, offering a range of options for processing and extracting valuable insights from the collected information. With functions to calculate mean wind speed and its standard deviation, ambient temperature, and solar radiation across different time periods. Additionally, the class facilitates data exportation to external files for further analysis.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Protection | Description |
| Menu | Class |  | Represents a Menu Class |
| OptionOne | Method | + | Method representing for option one in the menu. |
| OptionTwoThree | Method | + | Method representing for option two or three in the menu. |
| OptionFour | Method | + | Method representing for option four in the menu. |
| OptionFive | Method | + | Method representing for option five int the menu. |
| PrintSpccValues | Method | - | Method used in displaying for option Five. |

# Unit Testing

## Date class

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments | A screenshot of a computer program  Description automatically generated |
| 1 | Constructed - Default | None | day: 0, month:0, year: 0 | Y | Default constructor should initialize day, month, and year to the initial value. |
| 2 | Constructed - Valid input parameters | day: 11, month: 12, year: 2002 | day: 11, month: 12, year: 2002 | Y | The constructor should correctly initialize day, month, and year with the provided value. |
| 3 | Getter - GetDay | Set day to 15 | GetDay() returns 15 | Y | The GetDay() function should return the day set previously. |
| 4 | Getter - GetMonth | Set month to 7 | GetMonth() returns 7 | Y | The GetMonth() function should return the month set previously. |
| 5 | Getter - GetYear | Set year to 2022 | GetYear() return 2022 | Y | The GetYear() function should return the year set previously. |
| 6 | Setter - SetDay | Set day to 10 | GetDay() returns 10 | Y | The SetDay() function should correctly set the day. |
| 7 | Setter - SetMonth | Set month to 3 | GetMonth() returns 3 | Y | The SetMonth() function should correctly set the month. |
| 8 | Setter - SetYear | Set month to 2024 | GetYear() returns 2024 | Y | The SetYear() function should correctly set the year. |
| 9 | GetMonthInStr - Valid month | Input: 4 | Returns "April" | Y | The GetMonthInStr() function should return the name if the month corresponding to the input month number. |
| 10 | GetMonthInStr - Invalid month | Input: 15 | Return "Error" | Y | The GetMonthInStr() function should return "Error" for invalid month numbers. |
| 11 | Operator << Overloading | Date: 12/5/2023 | Output: "12 May 2023" | Y | The overloading << operator should correctly output the date in the format "day month year". |

## Time class

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments |  |
| 1 | Default Constructor | None | Hour, Minute initialized to 0 | Y | Constructor should initialize hour and minute to default values. |
| 2 | Parameterized Constructor | Hour = 12, Minute = 20 | Hour = 12, Minute = 20 | Y | Constructor should set hour and minute to specified values. |
| 3 | Setters | Hour = 10, Minute = 20 | Hour = 10, Minute = 20 | Y | Setters should update hour and minute to specified values. |
| 4 | Getters | Hour = 10, Minute = 20 | Hour = 10, Minute = 20 | Y | Getters should return the correct hour and minute values. |
| 5 | operator == Overloading - Equals Time | thisTime, otherTime | "Equals" | Y | Overloaded == operator should correctly compare two Time objects for equality. |
| 6 | operator != Overloading - Not Equals | thisTime, otherTime | "Not Equals" | Y | Overloaded != operator should correctly compare two Time objects for inequality. |
| 7 | operator < Overloading - Lesser < Time | thisTime, otherTime | "thisTime < otherTime" | Y | Overloaded < operator should correctly compare two Time objects to check if one is lesser. |
| 8 | operator > Overloading - Greater > Time | thisTime, otherTime | "thisTime > otherTime" | Y | Overloaded > operator should correctly compare two Time objects to check if one is greater. |
| 9 | Operator << Overloading | Time object with hour = 12, minute = 20 | "12:20" | Y | Overloaded << operator should correctly output Time object in hh:mm format. |
| 10 | Operator >> Overloading - Valid input | Input = '10:20' | Time object with Hour = 10, Minute = 20 | Y | Overloaded >> operator should correctly parse input string into Time object. |

## Measurement class

The ‘Measurement’ class cannot be directly tested as it is an abstract class. Abstract classes provide blueprints for other classes and cannot be instantiated on their own.

## WindSpeed class

The ‘WindSpeed’ class represents wind speed measurements and is a child class of the ‘Measurement’ class. It encapsulates a single float value representing wind speed in meters per second (m/s). The class allows instantiation with either a default value of 0 or a specified wind speed measurement. Additionally, it includes a method to convert wind speed from meters per second to kilometres per hour (km/h).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments |  |
| 1 | Default Constructor | None | WindSpeed initialized to 0.0 | Y | Constructor initializes windspeed to initial value. |
| 2 | Parameterized Constructor | windspeed = 12.12 | windspeed = 12.12 | Y | Constructor sets windspeed to provided value. |
| 3 | Setters | windspeed = 11.11 | windspeed = 11.11 | Y | Setter method sets windspeed to new value. |
| 4 | Getters | windspeed = 11.11 | windspeed = 11.11 | Y | Getter method returns correct windspeed value. |
| 5 | Converting WindSpeed to kWh/m2 | windspeed = 30.0 | windspeed = 108.0 | Y | Method converts windspeed correctly to kWh/m2. |

## SolarRadiation class

The SolarRadiation class represents a measurement of solar radiation. It is a child class of the Measurement class and provides functionality for storing and manipulating solar radiation data. This class encapsulates methods for setting and retrieving solar radiation measurements, as well as converting between different units of measurement.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments |  |
| 1 | Default Constructor | None | Solar radiation initialized to INITIAL\_VALUE | Y | Constructor initializes solar radiation measurement to initial value. |
| 2 | Constructor with Valid Input Parameters | Solar radiation = 12.12 | solar radiation = 12.12 | Y | Constructor sets solar radiation measurement to provided value. |
| 3 | Setters | Solar radiation = 11.11 | solar radiation = 11.11 | Y | Setter method sets solar radiation measurement to new value. |
| 4 | Getters | Solar radiation = 11.11 | solar radiation = 11.11 | Y | Getter method returns correct solar radiation value. |
| 5 | Converting SolarRadiation | solarRadiation = 30.0 | solarRadiation = 0.005 | Y | The actual output may vary slightly due to precision. |

## Temperature class

The Temperature class represents a measurement of temperature. It is a child class of the Measurement class and provides functionality for storing and manipulating the temperature data. This class encapsulates methods for setting and retrieving temperature measurements, as well as converting between different units of measurement.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments |  |
| 1 | Default Constructor | None | temperature initialized to INITIAL\_VALUE | Y | Constructor initializes temperature measurement to initial value. |
| 2 | Constructor with Valid Input Parameters | temperature = 12.12 | temperature = 12.12 | Y | Constructor sets temperature measurement to provided value. |
| 3 | Setters | temperature = 11.11 | temperature = 11.11 | Y | Setter method sets temperature measurement to new value. |
| 4 | Getters | temperature = 11.11 | temperature = 11.11 | Y | Getter method returns correct temperature value. |
| 5 | Converting Fahrenheit To DegreeC | Fahrenheit = 118.4 | DegreeC = 48.0 | Y | Ensure proper conversion calculation |

## SensorMeasurementType class

The ‘SensorMeasurementType’ class cannot be directly tested as it is an enum class. Enum class are used to define a set of named constants, and they cannot be instantiated or tested like regular classes.

## Vector class

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments | A screenshot of a computer  Description automatically generated |
| 1 | Test the initialization of the Vector class. | None | The initial capacity should be set to the default value (2), and the number of used slots should be set to 0. | Y | Default constructor should initialize all sensor data to the default values. |
| 2 | Test on data type initialization of Vector | None | Vector data type set as unsigned data type. | Y | None |
| 3 | Test on insertion of object to Vector's last index. | 1st Object: 10, 2nd Object: 20, 3nd Object: 30, 4th Object: 40 | Objects successfully inserted into Vector's last index. | Y | None |
| 4 | Test on insertion of object of object to Vector's index [1]. | index[0]: 10, index[1]: 20, index[2]: 30, index[3]: 40. Object to insert: 50 at index[1] | Object successfully inserted at specific index. | Y | None |
| 5 | Test on removing object of the last element in Vector. | 10, 20, 30, 40 | Last element removed successfully from the Vector | Y | None |
| 6 | Test on removing object from a specific index | 10, 20, 30, 40; Removing object at index 1 | Object successfully removed from specified index | Y | None |
| 7 | Test for loop output of Vector's element using \*Start And \*Finish | 10, 20, 30, 40 | Pointers correctly traverse Vector | Y | None |
| 8 | Test for index accessor | 10, 20, 30, 40 | Index operator correctly accesses Vector elements | Y | None |
| 9 | Test on Copy Constructor | 10, 20, 30, 40 | Copied Vector matches original Vector | Y | Elements address are to be different on both Vectors. |
| 10 | Test on Equals Assignment Operator | 10, 20, 30, 40 | Assigned Vector matches original Vector | Y | Elements address are to be different on both Vectors. |

## Tree class

The ‘Tree’ class cannot be directly tested as it is an abstract class. Abstract classes provide blueprints for other classes and cannot be instantiated on their own.

## BST class

The ‘BST’ class represents a Binary Search Tree (BST), derived from the `Tree` class. This class provides functionality to create, insert, delete, and search nodes within a binary search tree structure. It implements the basic operations of a BST, including insertion and deletion of nodes while maintaining the binary search tree properties. Additionally, it supports traversal operations such as pre-order, in-order, and post-order traversal. The class ensures that the elements in the BST are stored in a sorted order, making searching and retrieval operations efficient.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments |  |
| 1 | Initialization. | None | Initial tree height, nodes, and leaves are set to zero | Y | Test confirms basic initialization functionality. |
| 2 | Node Insertion and Search Operation | Insertion: 5, 6, 10; Search: 5, 12 | Nodes inserted correctly; search operation returns expected results | Y | Confirms correct insertion and search functionality. |
| 3 | Traversal Orders | Tree with nodes 5, 6, 10 | Pre-order, in-order, and post-order traversals match expected results | Y | Verifies traversal methods produce expected output. |
| 4 | Deep Copying Operations | Tree with nodes 5, 6, 10 | Copy constructor and equal assignment operator work as expected | Y | Copy constructor and assignment operator fail to create a deep copy. |
| 5 | Node Deletion | Tree with nodes 5, 6, 10, 15, 16, 17, 19  Deletion: 5, 15, 17 | Nodes deleted correctly; tree height, nodes, and leaves updated as expected | Y | Confirms proper deletion of nodes and updates tree metrics. |
|  | | | | | |  |

## AVL class

The AVL Tree class provides an implementation of a self-balancing binary search tree known as an AVL tree. This class ensures that the tree remains balanced after insertion and deletion operations, which helps maintain efficient search, insertion, and deletion time complexities.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments |  |
| 1 | Initialization | AVL tree initialized | Tree height: 0, Nodes: 0, Leaves: 0 | Y |  |
| 2 | Insertion of Nodes | Insert 5, 6, 10 | Tree height: 2, Nodes: 3, Leaves: 2 | Y |  |
| 3 | Search Operation | Search for existing and non-existing nodes | Search for 5: true, Search for 12: false | Y |  |
| 4 | Traversal Orders | N/A | Pre-order: 6 5 10, In-order: 5 6 10, Post-order: 5 10 6 | Y |  |
| 5 | Deep Copying Operations | Copy constructor, Assignment operator | Same tree structure and values as original | Y |  |
|  | | | | | |  |

## Processor class

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments |  |
| 1 | Load CSV File Paths to Vector | Vector containing 3 CSV file paths | Vector contains 3 CSV file paths | Y | - |
| 2 | Load CSV Data into Map | CSV file paths, AVL map object | Correct loading of data into the AVL map | Y | Ensure data is correctly loaded |
| 3 | Reloading of CSV Data into map | Using test file: data/data\_source\_test.txt | Correctly destroys the map and bst. And Inserts to a new map / bst structure | Y | Ensures data is loaded correctly. |

## AoAvlMap class

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments |  |
| 1 | Insert Data into Map, Array of AVL | Sensor record objects | Successful insertion of data | Y | Ensure data is correctly inserted into the map |
| 2 | Display Data After Insertion | Sensor data map | Display of inserted data | Y | Verify that inserted data is correctly displayed |
| 3 | Display Data With User Year and Month | Sensor data map, year, month | Display of data for specified year/month | Y | Check if data for specified year/month is displayed correctly |
| 4 | In-Order Traversal Sum | Sensor data map, year, month | Sum of specific sensor measurements | Y | Validate the sum of sensor measurements for specified year/month |
| 5 | In-Order Traversal SSD (Sample Standard Deviation) | Sensor data map, year, month | Sample standard deviation of sensor data | Y | Verify the calculation of sample standard deviation of sensor data |
| 6 | Extract Measurement from AVL to Vector | Sensor data map, day, month, year | Extracted to Vector | Y | Ensure Measurements extracted into the vector |
| 7 | Destroying previous Map / Bst structure | Sensor data map, day, month, year | BEFORE: With data  AFTER: Without data | Y | Ensure Deallocation done in destroying the structure. And ensure data is reloaded into a new map / bst structure. |

## InputValidator class

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments |  |
| 1 | DayValidation: Valid input | "31" | Success message for valid input | Y | Valid day input within the range. |
| 2 | Day Validation | "c", "0", "32" | Error message for invalid input, | Y | Invalid day input within the range. |
| 3 | MonthValidation: Valid input | "12" | Success message for valid input | Y | Valid month input within the range. |
| 2 | Month Validation | "c", "0", "13" | Error message for invalid input. | Y | Invalid month input within the range. |
| 5 | YearValidation: Valid input | "2011" | Success message for valid input | Y | Valid year input within the range. |
| 3 | Year Validation | "c", "2009", "2031" | Error message for invalid input. | Y | Invalid year input within the range. |

## Menu class

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No | Description / Condition Tested | Test Data / Values | Expected Output | Passed (Y / N) | Comments |  |
| 1 | Calculate and display average wind speed | Year: 2014, Month: 1. | Average Wind Speed: 32.4 | Y |  |
|  | Calculate and display standard deviation for a specified year and month. | Year: 2014 Month: 1 | Standard Deviation: 2.94 | Y |  |
| 2 | Calculate and display average ambient temperature and stdev for each month of a specified year. | Year: 2014 | Success displaying ambient temperature and stdev for each month of the year. | Y |  |
| 3 | Calculate and display sum of solar radiation on each month of the year. | Year: 2014 | Success Displaying sum of solar radiation on each month of the year. | Y |  |
| 4 | Calculate and display option Four for each month of the year. Additionally, exporting the result to a csv file | Year: 2014 | Success Displaying option four required reading on each month of the year. And exported the csv file | Y |  |
| 5 | Calculate and display sample Pearson correlation, coefficient values for wind speed, temperature, and solar radiation for a specified month. | Month: 1 | Pairs are calculated. Sample Pearson correlation coefficients are displayed. | Y |  |
|  | | | | |  | |

# 

# Algorithm

## Date class

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Operator == ( otherDate ) | |  |  | Operator < (otherDate) | | |
|  |  | **IF** year of current Date object is equal to year of OtherDate object |  |  |  | **IF** current year is less than otherDate year | |
|  |  | **IF** month of current Date object is equal to month of OtherDate object |  |  |  |  | **RETURN** true |
|  |  | **IF** day of current Date object is equal to day of OtherDate object |  |  |  | **ELSE IF** current year is greater than otherDate year | |
|  |  | **RETURN** true if all condition met |  |  |  |  | **RETURN** false |
|  | END operator == ( otherDate ) | |  |  |  | **IF** current month is less than otherDate month | |
|  |  | |  |  |  |  | **RETURN** true |
|  |  | |  |  |  | **ELSE IF** current month is greater than otherDate month | |
|  |  | |  |  |  |  | **RETURN** false |
|  |  | |  |  |  | **RETURN** current day is less than otherDate day | |
|  |  | |  |  | **END** operator < ( otherDate ) | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Operator != ( otherDate ) | |  |  | Operator > (otherDate) | |
|  |  | **NEGATES** the result of operator == (otherDate) |  |  |  | **NEGATES** the result of operator < (otherDate) |
|  |  | **RETURN** NOT (\*this == otherDate) |  |  |  | **RETURN** (otherDate < \*this) |
|  | END operator != ( otherDate ) | |  |  | END operator > ( otherDate ) | |

## Time class

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Operator == ( otherTime ) | |  |  | Operator < (otherTime) | | |
|  |  | **IF** hour of current Time object is equal to hour of OtherTime object |  |  |  | **IF** current hour is not equals to otherTime hour | |
|  |  | **IF** minute of current Time object is equal to minute of OtherTime object |  |  |  |  | **RETURN** true ( when this hour is less otherTime hour) |
|  |  | **RETURN** true if all condition met |  |  |  | **ELSE** | |
|  | **END** operator == ( otherDate ) | |  |  |  |  | **RETURN** true (when this min is less than otherTime min) |
|  |  | |  |  |  | **END IF ELSE** | |
|  |  | |  |  | **END** operator < ( otherTime ) | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Operator != ( otherTime ) | |  |  | Operator > (otherTime) | |
|  |  | **NEGATES** the result of operator == (otherTime) |  |  |  | **NEGATES** the result of operator < (otherTime) |
|  |  | **RETURN** NOT (\*this == otherTime) |  |  |  | **RETURN** (otherTime < \*this) |
|  | END operator != ( otherTime ) | |  |  | END operator > ( otherTime ) | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| WindSpeed class | | |  | SolarRadiation class | | |
|  | ConvertUnit( measurement ) | |  |  | ConvertUnit( measurement ) | |
|  |  | Multiply measurement by 3.6 |  |  |  | **CONVERT** from mW/m2 to W/m2, then to kW/m2 |
|  |  | **RETURN** converted Measurement |  |  |  | **RETURN** converted Measurement |
|  | **END** ConvertUnit | |  |  | **END** ConvertUnit | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Temperature class | | |  |  | | |
|  | ConvertUnit( measurement ) | |  |  |  | |
|  |  | **CONVERT** from Fahrenheit to Celsius |  |  |  |  |
|  |  | **RETURN** converted Measurement |  |  |  |  |
|  | **END** ConvertUnit | |  |  |  | |

## SensorRecType class

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Operator == ( other ) | |  |  | Operator < (other) | | |
|  |  | **IF** date of current object is equal to date of other object |  |  |  | **IF** current date is less than other time | |
|  |  | **IF** time of current object is equal to time of other object |  |  |  |  | **RETURN** true |
|  |  | **RETURN** true if all condition met |  |  |  | **ELSE IF** current date is greater than other date | |
|  | **END** operator == ( other ) | |  |  |  |  | **RETURN** false |
|  |  | |  |  |  | **ELSE** | |
|  |  | |  |  |  |  | **RETURN** current time is less than other time |
|  |  | |  |  | **END** operator < ( otherTime ) | | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Operator > ( other ) | |  |  |  | |
|  |  | **NEGATES** the result of operator < (other) |  |  |  |  |
|  |  | **RETURN** (other < \*this) |  |  |  |  |
|  | **END** operator > ( otherTime ) | |  |  |  | |

## Tree class

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Destroy(ptr) | | |  |  | GetMaxTreeSubHeight(left, right) | | | |
|  |  | **IF** ptr is not pointing at null pointer | |  |  |  | **IF l**eft is greater or equals to right | | |
|  |  |  | **CALL** Destroy on ptr to the left |  |  |  |  | **RETURN** left | |
|  |  |  | **CALL** Destroy on ptr to the right |  |  |  | **ELSE** | | |
|  |  |  | **DELETE** the ptr |  |  |  |  | **RETURN** right |  |
|  |  |  | **SET** null to ptr |  |  |  | **END IF ELSE** | | |
|  |  | **END IF** | |  |  | **END** GetMaxTreeSubHeight | | | |
|  | **END** Destroy | | |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | GetHeight(ptr) | | |  |  | GetNodes(ptr) | | | |
|  |  | **IF** ptr is pointing at null pointer | |  |  |  | **IF** ptr is pointing at null pointer | | |
|  |  |  | **RETURN** 0 |  |  |  |  | **RETURN** 0 | |
|  |  | **ELSE** | |  |  |  | **ELSE** | | |
|  |  |  | **RETURN** 1 + GetMaxTreeSubHeight(leftSub, rightSub) |  |  |  |  | **RETURN** 1+ GetNodes of left + GetNodes of right | |
|  |  | **END IF ELSE** | |  |  |  | **END IF ELSE** | | |
|  | **END** Destroy | | |  |  | **END** GetNodes | | | |
|  |  | | |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | GetLeaves(ptr) | | |  |  | CopyTree(copiedTreeRoot, otherTreeRoot) | | | |
|  |  | **IF** ptr is pointing at null pointer | |  |  |  | **IF** otherTreeRoot is pointing at null pointer | | |
|  |  |  | **RETURN** 0 |  |  |  |  | **SET** null pointer to copiedTreeRoot | |
|  |  | **ELSE** **IF** pointer of left and right pointing at null pointer | |  |  |  | **ELSE** | | |
|  |  |  | **RETURN** 1 |  |  |  |  | **CREATE** a new Node<T> for copiedTreeRoot | |
|  |  | **ELSE** | |  |  |  |  | **SET** otherTreeRoot object to copiedTreeRoot | |
|  |  |  | **RETURN** GetLeaves of left + GetLeaves of right |  |  |  |  | **CALL** CopyTree to copy otherTreeRoot left object | |
|  |  | **END IF ELSE** | |  |  |  |  | **CALL** CopyTree to copy otherTreeRoot right object | |
|  | **END** GetLeave | | |  |  |  | **END IF ELSE** | | |
|  |  | | |  |  | **END** CopyTree | | | |
|  |  | | |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Tree(otherTree) | | |  |  | Operator = (otherTree) | | | |
|  |  | **IF** root of the otherTree is pointing at null | |  |  |  | **IF** address of this Tree and the otherTree is different | | |
|  |  |  | **SET** null pointer to this root |  |  |  |  | **IF** this root is not empty | |
|  |  | **ELSE** | |  |  |  |  |  | **CALL** Destroy to delete this node |
|  |  |  | **CALL** CopyTree in copying otherTree content to this root. |  |  |  |  | **END IF** | |
|  |  | **END IF ELSE** | |  |  |  |  | **IF** otherTreeRoot is empty | |
|  | **END** Tree | | |  |  |  |  |  | **SET** nullptr to this root |
|  |  |  | |  |  |  |  | **ELSE** | |
|  |  | | |  |  |  |  |  | **CALL** CopyTree to copy the otherTree to this Tree |
|  |  | | |  |  |  |  | **END IF ELSE** | |
|  |  | | |  |  |  | **END IF** | | |
|  |  | | |  |  | **END** operator = (otherTree) | | | |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | PreOrder( traversal, ptr ) | | |  |  |  | InOrder( traversal, ptr ) | | |  |  |
|  |  | **IF** tree is not empty | |  |  |  |  | **IF** tree is not empty | |  |  |
|  |  |  | **CALL** traversal on pointing object |  |  |  |  |  | **CALL** InOrder traversal on the left pointer | |  |
|  |  |  | **CALL** PreOrder traversal on the left pointer |  |  |  |  |  | **CALL** traversal on pointing object | |  |
|  |  |  | **CALL** PreOrder traversal on the right pointer |  |  |  |  |  | **CALL** InOrder traversal on the right pointer | |  |
|  |  | **END IF** | |  |  |  |  | **END IF** | | |  |
|  | **END** PreOrder | | |  |  |  | **END** InOrder | | | |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | PostOrder( traversal, ptr ) | | |  |  |  | Search(current, object) | | |  |  |
|  |  | **IF** tree is not empty | |  |  |  |  | **IF** current is empty | |  |  |
|  |  |  | **CALL** PostOrder traversal on the left pointer | |  |  |  |  | **RETURN** false | |  |
|  |  |  | **CALL** PostOrder traversal on the right pointer | |  |  |  | **ELSE IF** object is equals to the current pointed object | | | |
|  |  |  | **CALL** traversal on pointing object | |  |  |  |  | **RETURN** true | |  |
|  |  | **END IF** | | |  |  |  | **ELSE IF** object is less than the current pointed object | | | |
|  | **END** PostOrder | | | |  |  |  |  | **RETURN CALL** Search to left on comparing the object | | |
|  |  |  |  | |  |  |  | **ELSE** | | | |
|  |  |  |  | |  |  |  |  | **RETURN CALL** Search to right on comparing the object | | |
|  |  |  | |  |  |  |  | **END IF** | | |  |
|  |  | | |  |  |  | **END** Search | | | |  |

## AVL class

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | GetBalanceFactor(ptr) | | | | |  |  |  |  |  |
|  |  | **IF** ptr is pointing at null | | | |  |  |  |  |  |
|  |  |  | **RETURN** -1 | | |  |  |  |  |  |
|  |  | **ELSE** | | | |  |  |  |  |  |
|  |  |  | **RETURN** (**CALL** GetHeight of leftSubTree - GetHeight of rightSubTree) | | | |  |  |  |  |
|  |  | **END IF** | |  |  |  |  |  |  |  |
|  | **END** GetBalanceFactor | | |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | RightRotate(rst) | |  |  |  | LeftRotate(lst) | |
|  |  | **SET** right Subtree(lst) of left node to leftSubTree(lst) |  |  |  |  | **SET** leftSubTree(lst) of right node to rightSubTree(rst) |
|  |  | **SET** left Subtree(lst) of right node to otherTree |  |  |  |  | **SET** rightSubTree(rst) of left node to otherTree |
|  |  | **SET** right Subtree(rst) to leftSubTree(lst) right node |  |  |  |  | **SET** leftSubTree(lst) to rightSubTree(lst) left node |
|  |  | **SET** other Tree back to rightSubTree(rst) left node |  |  |  |  | **SET** otherTree back to leftSubTree(rst) right node |
|  |  | **RETURN** left Subtree(lst) |  |  |  |  | **RETURN** rightSubTree(rst) |
|  | **END** RightRotate | |  |  |  | **END** LeftRotate | |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | INSERT(current, object) | | | | | |
|  |  | **IF** current is empty | |  |  |  |
|  |  |  | **CREATE** new node<T> to current |  |  |  |
|  |  |  | **SET** object to current node |  |  |  |
|  |  |  | **SET** null to left and right pointer |  |  |  |
|  |  |  | **RETURN** true |  |  |  |
|  |  | **ELSE IF** object is less than the current object | |  |  |  |
|  |  |  | **CALL INSERT** object to current left node |  |  |  |
|  |  |  | **IF** not inserted, RETURN false |  |  |  |
|  |  |  | **END IF** |  |  |  |
|  |  | **ELSE IF** object is greater than the current object | |  |  |  |
|  |  |  | **CALL INSERT** object to current right node |  |  |  |
|  |  |  | **IF** not inserted, RETURN false |  |  |  |
|  |  |  | **END IF** |  |  |  |
|  |  | **ELSE** | |  |  |  |
|  |  |  | **RETURN** false |  |  |  |
|  |  | **END IF ELSE** | |  |  |  |
|  |  | **CALL** GetBalanceFactor for the current node | |  |  |  |
|  |  | **IF** balanceFactor is greater than 1 AND object is lesser than the current left node’s object | | | | |
|  |  |  | **SET AND CALL** RightRotate of current node ( This is a Left-left rotate ) | | | |
|  |  | **ELSE IF** balanceFactor is less than -1 AND object is greater than the current right node’s object | | | | |
|  |  |  | **SET AND CALL** LeftRotate of current node ( This is a Right-right rotate ) | | | |
|  |  | **ELSE IF** balanceFactor is greater than 1 AND object is greater than the current left node’s object | | | | |
|  |  |  | **SET AND CALL** LeftRotate of current left node to the current left node | | | |
|  |  |  | **SET AND CALL** RightRotate of the current ( This is Left-right rotate ) | | | |
|  |  | **ELSE IF** balanceFactor is less than -1 AND object is lesser than the current right node’s object | | | | |
|  |  |  | **SET AND CALL** RightRotate of current right node to the current right node | | | |
|  |  |  | **SET AND CALL** LeftRotate of the current ( This is Right-left rotate ) | | | |
|  |  | **END IF** | | | | |
|  |  | **RETURN** true | | | | |
|  | **END** INSERT | | | | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Delete(current, object) | | | | | | | | | | | |
|  | **IF** current is empty | | | |  |  | | |  | |  | |
|  |  | **RETURN** false | | |  |  | | |  | |  | |
|  | **ELSE IF** object is less than the current object | | | |  |  | | |  | |  | |
|  |  | **RETURN CALL** Delete the current left node when object is equals | | |  |  | | |  | |  | |
|  | **ELSE IF** object is greater than the current object | | | |  |  | | |  | |  | |
|  |  | **RETURN** **CALL** Delete the current right node when object is equals | | |  |  | | |  | |  | |
|  | **ELSE** | | | |  |  | | |  | |  | |
|  |  | **RETURN** false | | |  |  | | |  | |  | |
|  |  | **IF** the current left node is empty | | |  |  | | |  | |  | |
|  |  |  | **SET** current right node to a temp variable | |  |  | | |  | |  | |
|  |  |  | **RELEASE** the current node | |  |  | | |  | |  | |
|  |  |  | **SET** the temp variable back to current | |  |  | | |  | |  | |
|  |  |  | **RETURN** true | |  |  | | |  | |  | |
|  |  | **ELSE** **IF** the current right node is empty | | |  |  | | |  | |  | |
|  |  |  | **SET** current left node to a temp variable | |  |  | | |  | |  | |
|  |  |  | **RELEASE** the current node | |  |  | | |  | |  | |
|  |  |  | **SET** the temp variable back to current | |  |  | | |  | |  | |
|  |  |  | **RETURN** true | |  |  | | |  | |  | |
|  |  | **ELSE** | | |  |  | | |  | |  | |
|  |  |  | **SET** current right node to a temp variable | |  |  | | |  | |  | |
|  |  |  | **WHILE** the temp left node is not empty | |  |  | | |  | |  | |
|  |  |  |  | **SET** the temp left node to temp |  | |  |  | |  | |
|  |  |  | **END WHILE** | |  |  | | |  | |  | |
|  |  |  | **SET** the temp object to the current object | |  |  | | |  | |  | |
|  |  |  | **CALL** Delete recursion with current right node and temp object | |  |  | | |  | |  | |
|  |  | **END IF ELSE** | | |  |  | | |  | |  | |
|  |  | **IF** true | | |  |  | | |  | |  | |
|  |  |  | **CALL** GetBalanceFactor of current | |  |  | | |  | |  | |
|  |  |  | **IF** balanceFactor is equals to 2 AND GetBalanceFactor of the current left node is less than and equals to 0 | | | | | | | | | |
|  |  |  |  | **SET** RightRotate of the current to current | | | | | | | | |
|  |  |  | **ELSE IF** balanceFactor is equals to 2 AND GetBalanceFactor of the current left node is equals to -1 | | | | | | | | | |
|  |  |  |  | **CALL AND SET** LeftRotate on the current left node to the current left node | | | | | | | | |
|  |  |  |  | **CALL AND SET** RightRotate on the current back to current | | | | | | | | |
|  |  |  | **ELSE IF** balanceFactor is equals to -2 AND GetBalanceFactor of the current right node is lesser or equals to 0 | | | | | | | | | |
|  |  |  |  | **SET** LeftRotate of the current to current | | | | | | | | |
|  |  |  | **ELSE IF** balanceFactor is equals to -2 AND GetBalanceFactor of the current right node is equals to 1 | | | | | | | | | |
|  |  |  |  | **CALL AND SET** RightRotate on the current right node to the current right node | | | | | | | | |
|  |  |  |  | **CALL AND SET** LeftRotate on the current back to current | | | | | | | | |
|  |  |  | **END IF ELSE IF** | | | | | | | | | |
|  |  | **END IF** | | | | | | | | | | |
|  |  | **RETURN** deleted | | | | | | | | | | |
|  | **END** Delete | | | | | | | | | | | |

## AoAvlMap class

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | InsertSensorData( sensorData) | | | |  |  |  |
|  |  | Check IF enter for the year exists in the map | | | |  |  |
|  |  |  | IF entry does not exist | | |  |  |
|  |  |  |  | CREATE it and initialize the array | |  |  |
|  |  |  | END IF | | |  |  |
|  |  | END IF | | | |  |  |
|  |  | ACCESS the array for the month and add the sensor data | | | |  |  |
|  |  | RETURN true | | | |  |  |
|  | END InsertSensorData | | | | |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | InOrderTraversalSum(root, sensorMeasurement, sum) | | | |  |  |
|  |  | Base case: if the root node is nullptr, return | | |  |  |
|  |  | Recursive case: perform an in-order traversal of the left subtree | | |  |  |
|  |  |  | SWITCH and calculate the sum based on sensorMeasurement type | |  |  |
|  |  |  |  | IF CASE of sensor measurement type is WIND\_SPEED, calculate the sum | |  |
|  |  |  |  | IF CASE of sensor measurement type is SOLAR\_RADIATION, calculate the sum | |  |
|  |  |  |  | IF CASE of sensor measurement type is WIND\_SPEED, calculate the sum | |  |
|  |  |  | END SWITCH | | |  |
|  |  | Recursive case: perform an in-order traversal of the right subtree | | | |  |
|  | END InOrderTraversalSum | | | | |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | InOrderTraversalSumAndSquareSum(root, sensorMeasurementType, sum, sumSquare) | | | | | |
|  |  | Base case: if the root node is nullptr, return | |  |  |  |
|  |  | Recursive case: perform an in-order traversal of the left subtree | |  |  |  |
|  |  | **SWITCH** and calculate the sum and sum of squares based on sensorMeasurement type | | |  |  |
|  |  |  | **IF** CASE of sensor measurement type is WIND\_SPEED, calculate the sum and sum of squares | | |  |
|  |  |  | **IF** CASE of sensor measurement type is SOLAR\_RADIATION, calculate the sum and sum of squares | | |  |
|  |  |  | **IF** CASE of sensor measurement type is AMBIENT\_TEMPERATURE, calculate the sum and sum of squares | | |  |
|  |  | **END** **SWITCH** | | | |  |
|  | **END** InOrderTraversalSumAndSquareSum | | | | |  |

|  |  |  |
| --- | --- | --- |
|  | CalculateSampleStandardDeviation(root, SensorMeasurementType, count) | |
|  |  | **CALL** InOrderTraversalSumAndSquaredSum |
|  |  | **CALCULATE** variance as (sumSquare - (pow(sum, 2) / count)) / (count - 1) |
|  |  | **RETURN** the square root of variance as the sample standard deviation |
|  | **END** CalculateSampleStandardDeviation | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | FindHighestSolarRadiation(day, month, year) | | | | | | |  | |  | |  | |  | | | |
|  |  | **FIND** the AVL tree corresponding to the given day, month, and year | | | | | |  | |  | |  | |  | | | |
|  |  |  | **IF** yearIter is not equal to m\_data.end(): | | | | |  | |  | |  | |  | | | |
|  |  |  |  | **TRAVERSET** the AVL tree to find the highest solar radiation reading | | |  | |  | |  | |  | |
|  |  |  |  | **WHILE** root is not nullptr | | |  | |  | |  | |  | |
|  |  |  |  |  | **IF** root's date matches given day, month, and year AND root's solar radiation is greater than current highest: | | | | | | | | | | | |
|  |  |  |  |  |  | **Update** highest radiation and corresponding time | | | | | | | | | |
|  |  |  |  |  | **ELSE** **IF** root's date matches given day, month, and year AND root's solar radiation equals current highest: | | | | | | | | | | |
|  |  |  |  |  |  | **ADD** time to list of highest times | | | | | | | | | |
|  |  |  |  |  | **Move** to the right child of the current node | | | | | | | | | | |
|  |  |  |  |  | **END IF ELSE IF** | | | | | | | | | | |
|  |  |  |  | **END WHILE** | | | | | | | | | | | |
|  |  |  | **END IF** | | | | | | | | | | | | |
|  | **END** FindHighestSolarRadiation | | | | | | | | | | | | | | |

## Processor class

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | LoadCSVFilePathToVector(Vector<string> &csvFilePath, ifstream &input, const string &filename) | | | |
|  |  | IF input is not open | | |
|  |  |  | Print error message indicating failure to open file | |
|  |  | ELSE | | |
|  |  |  | Print message indicating successful file opening | |
|  |  |  | WHILE input stream is not at end-of-file | |
|  |  |  |  | READ file path from the input stream |
|  |  |  |  | INSERT file path into the csvFilePath vector |
|  |  |  | END WHILE | |
|  |  | END IF ELSE | | |
|  |  | CLOSE the input stream | | |
|  | END LoadCSVFilePathToVector | | | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | LoadCSVData(Vector<string> &csvFilePath, AoAvlMap<SensorRecType> &mapAoBst) | | | | | | |
|  |  | **FOR** each file path in csvFilePath | | | | | |
|  |  |  | string modifiedFilePath = sysFilePath + file path; | | | | |
|  |  |  | ifstream inFile(modifiedFilePath); | | | | |
|  |  |  | **IF** file is open | | | | |
|  |  |  |  | **Print** message indicating successful file opening | | | |
|  |  |  |  | **Read** header line from inFile and store in headers vector | | | |
|  |  |  |  | **WHILE** getline returns true (reading lines from inFile): | | | |
|  |  |  |  |  | **IF** line is empty or contains "N/A" or contains only commas: | | |
|  |  |  |  |  |  | **CONTINUE** to next iteration | |
|  |  |  |  |  | **END IF** | | |
|  |  |  |  |  | SPLIT line into records using comma delimiter | | |
|  |  |  |  |  | **INITIALIZE** indices for WAST, T, S, and SR columns to -1 | | |
|  |  |  |  |  | **FOR** each header in headers: | | |
|  |  |  |  |  |  | **IF** header is "WAST", "T", "S", or "SR": | |
|  |  |  |  |  |  |  | **UPDATE** corresponding index with current header index | |
|  |  |  |  |  |  | **END IF** | | |
|  |  |  |  |  | **END FOR** | | | |
|  |  |  |  |  | **IF** all required indices are valid | | | |
|  |  |  |  |  |  | **EXTRACT** WAST, T, S, and SR values from records using indices | | |
|  |  |  |  |  |  | **SPLIT** WAST into date and time parts | | |
|  |  |  |  |  |  | **PARSE** date and time parts into Date and Time objects | | |
|  |  |  |  |  |  | **PARSE** S, T, and SR values into WindSpeed, Temperature, an | | |
|  |  |  |  |  |  | **CREATE** SensorRecType object with parsed data | | |
|  |  |  |  |  |  | **INSERT** SensorRecType object into mapAoBst | | |
|  |  |  |  |  |  | **INCREMENT** data inserted count | | |
|  |  |  |  |  | **END** **IF** | | | |
|  |  |  |  | **END** **WHILE** | | | | |
|  |  |  |  | **CLOSE** inFile | | | | |
|  |  |  | **ELSE** | | | | | |
|  |  |  |  | **PRINT** error message indicating failure to open file | | | | |
|  |  | **END** **FOR** | | | | | | |
|  | **END** LoadCSVData | | | | | | | |

# References

* Mike Shah (2022, January 30). Function pointers, typedef a function pointer, and std::function | Modern Cpp Series. YouTube. <https://www.youtube.com/watch?v=Go_mafBvnhk>.
* Balaji Gunasekaran (2019, December 17). Complete Guide On Using Doxygen To Document C Source Code. <https://embeddedinventor.com/guide-to-configure-doxygen-to-document-c-source-code-for-beginners/>.
* Mike Shah (2022, March 14). Default Constructor, default destructor, and more! | Modern Cpp Series. YouTube. <https://www.youtube.com/watch?v=EhinJ94emf8&list=PLvv0ScY6vfd8j-tlhYVPYgiIyXduu6m-L&index=38>.
* Mike Shah (2022, March 16). Copy Constructor and Copy Assignment (Deep and Shallow Copy) | Modern Cpp Series. YouTube. <https://www.youtube.com/watch?v=EBgBM7rPDic&list=PLvv0ScY6vfd8j-tlhYVPYgiIyXduu6m-L&index=39>.
* Mike Shah (2022, March 20). The Rule of 3 (and the lesser known Law of the Big Two) | Modern Cpp Series. YouTube. <https://www.youtube.com/watch?v=bvCEqS4S0sg&list=PLvv0ScY6vfd8j-tlhYVPYgiIyXduu6m-L&index=40>.
* Mike Shah (2022, March 24). Operator overloading | Modern Cpp Series. YouTube. <https://www.youtube.com/watch?v=8IFe9CzVWoQ&list=PLvv0ScY6vfd8j-tlhYVPYgiIyXduu6m-L&index=42>.
* Mike Shah (2022, April 20). Inheritance calling different constructors | Modern Cpp Series. YouTube. <https://www.youtube.com/watch?v=nqeQOlqJeCo&list=PLvv0ScY6vfd8j-tlhYVPYgiIyXduu6m-L&index=51>.
* Mike Shah (2021, November 1). Design Patterns - Singleton Pattern | Explanation and Implementation in C++. YouTube. <https://www.youtube.com/watch?v=eLAvry56vLU&t=341s>.
* Mike Shah (2021, November 22). Design Patterns - Factory Method Pattern Explanation and Implementation in C++. YouTube. <https://www.youtube.com/watch?v=vAmDQKeC99g&list=PLvv0ScY6vfd9wBflF0f6ynlDQuaeKYzyc&index=3>.
* Simple Snippets (2021, January 12). Working of AVL Tree Insertion Operation(Recursive Method) with Rotations & Full C++ Program Code. YouTube. https://www.youtube.com/watch?v=otiDcwZbCo4.