PRECISIONLIB240 DOCUMENTATION

`int findMax(int arr[], int size)`
- **Description**: Finds the maximum value in the given array.
- **Parameters**:
  - `arr`: An array of integers.
  - `size`: The number of elements in the array.
- **Return**: Returns the maximum value found in the array.

`int findMin(int arr[], int size)`
- **Description**: Finds the minimum value in the given array.
- **Parameters**:
  - `arr`: An array of integers.
  - `size`: The number of elements in the array.
- **Return**: Returns the minimum value found in the array.

`void reverseArray(int arr[], int size)`
- **Description**: Reverses the elements in the given array.
- **Parameters**:
  - `arr`: An array of integers.
  - `size`: The number of elements in the array.
- **Return**: No return value. The array is modified in place.

`void sortArray(int arr[], int size)`
- **Description**: Sorts the given array in ascending order.
- **Parameters**:
  - `arr`: An array of integers.
  - `size`: The number of elements in the array.
- **Return**: No return value. The array is modified in place.

`bool isArraySorted(int arr[], int size)`
- **Description**: Checks if the given array is sorted in ascending order.
- **Parameters**:
  - `arr`: An array of integers.
  - `size`: The number of elements in the array.
- **Return**: Returns `true` if the array is sorted, otherwise `false`.

`void mergeArrays(int arr1[], int size1, int arr2[], int size2, int result[])`
- **Description**: Merges two sorted arrays into a single sorted array.
- **Parameters**:
  - `arr1`: First sorted array of integers.

  - `size1`: Number of elements in the first array.
  - `arr2`: Second sorted array of integers.
  - `size2`: Number of elements in the second array.
  - `result`: Array to store the merged result.
- **Return**: No return value. The merged array is stored in `result`.

`void removeDuplicates(int arr[], int &size)`
- **Description**: Removes duplicate elements from the array.
- **Parameters**:
  - `arr`: An array of integers.
  - `size`: The number of elements in the array (updated after removing duplicates).
- **Return**: No return value. The array is modified in place.

`void rotateArrayLeft(int arr[], int size, int rotations)`
- **Description**: Rotates the elements of the array to the left by a given number of positions.
- **Parameters**:
  - `arr`: An array of integers.
  - `size`: The number of elements in the array.
  - `rotations`: The number of positions to rotate the array left.
- **Return**: No return value. The array is modified in place.

`void rotateArrayRight(int arr[], int size, int rotations)`
- **Description**: Rotates the elements of the array to the right by a given number of positions.
- **Parameters**:
  - `arr`: An array of integers.
  - `size`: The number of elements in the array.
  - `rotations`: The number of positions to rotate the array right.
- **Return**: No return value. The array is modified in place.

`bool isArrayEqual(int arr1[], int size1, int arr2[], int size2)`
- **Description**: Checks if two arrays are equal (have the same elements in the same order).
- **Parameters**:
  - `arr1`: First array of integers.
  - `size1`: Number of elements in the first array.
  - `arr2`: Second array of integers.
  - `size2`: Number of elements in the second array.
- **Return**: Returns `true` if the arrays are equal, otherwise `false`.

`int stringLength(const char* str)`
- **Description**: Calculates the length of the input string.
- **Parameters**:
  - `str`: Pointer to a null-terminated string.
- **Return**: Returns the length of the string.

`void stringCopy(char* destination, const char* source)`
- **Description**: Copies the contents of one string to another.
- **Parameters**:
  - `destination`: Pointer to the destination string.
  - `source`: Pointer to the source string.
- **Return**: No return value. The `destination` string is modified.

`int stringCompare(const char* str1, const char* str2)`
- **Description**: Compares two strings lexicographically.
- **Parameters**:
  - `str1`: Pointer to the first string.
  - `str2`: Pointer to the second string.
- **Return**: Returns `0` if the strings are equal, a negative value if `str1` is less than `str2`, or a positive value if `str1` is greater than `str2`.

`void stringConcatenate(char* destination, const char* source)`
- **Description**: Concatenates the `source` string at the end of the `destination` string.
- **Parameters**:
  - `destination`: Pointer to the destination string.
  - `source`: Pointer to the source string.
- **Return**: No return value. The `destination` string is modified.

`bool isPalindrome(const char* str)`
- **Description**: Checks if the given string is a palindrome.
- **Parameters**:
  - `str`: Pointer to the input string.
- **Return**: Returns `true` if the string is a palindrome, otherwise `false`.

`void reverseString(char* str)`
- **Description**: Reverses the input string.
- **Parameters**:
  - `str`: Pointer to the string to be reversed.
- **Return**: No return value. The `str` string is modified.

`int countOccurrences(const char* str, char ch)`
- **Description**: Counts the occurrences of a specific character in the given string.
- **Parameters**:
  - `str`: Pointer to the input string.
  - `ch`: Character to be counted in the string.
- **Return**: Returns the count of occurrences of the character in the string.

`char* convertToLowercase(char* str)`

- **Description**: Converts the characters in the string to lowercase.
- **Parameters**:
  - `str`: Pointer to the input string.
- **Return**: Returns a pointer to the modified string with lowercase characters.

`char* convertToUppercase(char* str)`
- **Description**: Converts the characters in the string to uppercase.
- **Parameters**:
  - `str`: Pointer to the input string.
- **Return**: Returns a pointer to the modified string with uppercase characters.

`char* removeSpaces(char* str)`
- **Description**: Removes all spaces from the string.
- **Parameters**:
  - `str`: Pointer to the input string.
- **Return**: Returns a pointer to the modified string without spaces.

`int factorial(int n)`
- **Description**: Calculates the factorial of a given integer.
- **Parameters**:
  - `n`: Integer value for which factorial is to be calculated.
- **Return**: Returns the factorial of `n`.

`bool isPrime(int num)`
- **Description**: Checks if a number is a prime number.
- **Parameters**:
  - `num`: Integer value to be checked for primality.
- **Return**: Returns `true` if `num` is prime, otherwise `false`.

`int gcd(int a, int b)`
- **Description**: Calculates the greatest common divisor (GCD) of two integers.
- **Parameters**:
  - `a`, `b`: Integers for which GCD is to be calculated.
- **Return**: Returns the GCD of `a` and `b`.

`int lcm(int a, int b)`
- **Description**: Calculates the least common multiple (LCM) of two integers.
- **Parameters**:
  - `a`, `b`: Integers for which LCM is to be calculated.
- **Return**: Returns the LCM of `a` and `b`.

`double calculatePower(double base, int exponent)`
- **Description**: Calculates the power of a number with a given exponent.

- **Parameters**:
  - `base`: Base value.
  - `exponent`: Exponent value.
- **Return**: Returns the result of `base` raised to the power of `exponent`.

`int calculateSquare(int num)`
- **Description**: Calculates the square of a given integer.
- **Parameters**:
  - `num`: Integer value to be squared.
- **Return**: Returns the square of `num`.

`double calculateSquareRoot(double num)`
- **Description**: Calculates the square root of a given number.
- **Parameters**:
  - `num`: Number for which square root is to be calculated.
- **Return**: Returns the square root of `num`.

`double calculateAverage(int arr[], int size)`
- **Description**: Calculates the average of elements in an integer array.
- **Parameters**:
  - `arr`: Integer array.
  - `size`: Number of elements in the array.
- **Return**: Returns the average value of elements in the array.

`int generateRandomNumber(int min, int max)`
- **Description**: Generates a random integer within a given range `[min, max]`.
- **Parameters**:
  - `min`: Minimum value of the range.
  - `max`: Maximum value of the range.
- **Return**: Returns a random integer within the specified range.

`int findClosestPowerOfTwo(int num)`
- **Description**: Finds the closest power of two for a given number.
- **Parameters**:
  - `num`: Integer value for which the closest power of two is to be found.
- **Return**: Returns the closest power of two to `num`.

`bool fileExists(const char* filename)`
- **Description**: Checks if a file exists in the specified location.
- **Parameters**:
  - `filename`: Path to the file.
- **Return**: Returns `true` if the file exists, otherwise `false`.

`bool createFile(const char* filename)`
- **Description**: Creates a new file at the specified location.
- **Parameters**:
  - `filename`: Path to the new file to be created.
- **Return**: Returns `true` if the file creation is successful, otherwise `false`.

`bool deleteFile(const char* filename)`
- **Description**: Deletes a file from the specified location.
- **Parameters**:
  - `filename`: Path to the file to be deleted.
- **Return**: Returns `true` if the file deletion is successful, otherwise `false`.

`int getFileSize(const char* filename)`
- **Description**: Retrieves the size of the specified file in bytes.
- **Parameters**:
  - `filename`: Path to the file.
- **Return**: Returns the size of the file in bytes.

`void readFromFile(const char* filename, char* buffer, int bufferSize)`
- **Description**: Reads data from a file into a buffer.
- **Parameters**:
  - `filename`: Path to the file to be read.
  - `buffer`: Pointer to the buffer where data will be stored.
  - `bufferSize`: Size of the buffer.
- **Note**: The function reads data from the file and stores it in the provided buffer.

`void writeToFile(const char* filename, const char* data)`
- **Description**: Writes data to a file.
- **Parameters**:
  - `filename`: Path to the file to which data will be written.
  - `data`: Data to be written into the file.

`void appendToFile(const char* filename, const char* data)`
- **Description**: Appends data to the end of a file.
- **Parameters**:
  - `filename`: Path to the file.
  - `data`: Data to be appended to the file.

`void copyFile(const char* sourceFilename, const char* destinationFilename)`
- **Description**: Copies a file from a source location to a destination location.
- **Parameters**:
  - `sourceFilename`: Path to the source file.
  - `destinationFilename`: Path to the destination where the file will be copied.

`void moveFile(const char* sourceFilename, const char* destinationFilename)`
- **Description**: Moves a file from a source location to a destination location.
- **Parameters**:
  - `sourceFilename`: Path to the source file.
  - `destinationFilename`: Path to the destination where the file will be moved.

`bool renameFile(const char* oldFilename, const char* newFilename)`
- **Description**: Renames a file.
- **Parameters**:
  - `oldFilename`: Current path of the file.
  - `newFilename`: New path and name for the file.
- **Return**: Returns `true` if the renaming is successful, otherwise `false`.

`Node* createNode(int data)`
- **Description**: Creates a new node with the given data value.
- **Parameters**:
  - `data`: Data value for the new node.
- **Return**: Returns a pointer to the newly created node.

`void insertNodeAtBeginning(Node*& head, int data)`
- **Description**: Inserts a new node with the given data at the beginning of the linked list.
- **Parameters**:
  - `head`: Reference to the head pointer of the linked list.
  - `data`: Data value for the new node to be inserted.

`void insertNodeAtEnd(Node*& head, int data)`
- **Description**: Inserts a new node with the given data at the end of the linked list.
- **Parameters**:
  - `head`: Reference to the head pointer of the linked list.
  - `data`: Data value for the new node to be inserted.

`void deleteNode(Node*& head, int data)`
- **Description**: Deletes the node containing the given data from the linked list.
- **Parameters**:
  - `head`: Reference to the head pointer of the linked list.
  - `data`: Data value of the node to be deleted.

`bool searchNode(const Node* head, int data)`
- **Description**: Searches for a node with the given data in the linked list.
- **Parameters**:
  - `head`: Pointer to the head of the linked list.
  - `data`: Data value to search for in the linked list.

- **Return**: Returns `true` if the data is found in the linked list, otherwise `false`.

`int getNodeCount(const Node* head)`
- **Description**: Counts the number of nodes in the linked list.
- **Parameters**:
  - `head`: Pointer to the head of the linked list.
- **Return**: Returns the count of nodes in the linked list.

`void reverseList(Node*& head)`
- **Description**: Reverses the linked list.
- **Parameters**:
  - `head`: Reference to the head pointer of the linked list.

`void deleteList(Node*& head)`
- **Description**: Deletes the entire linked list.
- **Parameters**:
  - `head`: Reference to the head pointer of the linked list.

`void printList(const Node* head)`
- **Description**: Prints the elements of the linked list.
- **Parameters**:
  - `head`: Pointer to the head of the linked list.

`Node* mergeLists(Node* list1, Node* list2)`
- **Description**: Merges two sorted linked lists into a single sorted linked list.
- **Parameters**:
  - `list1`: Pointer to the head of the first sorted linked list.
  - `list2`: Pointer to the head of the second sorted linked list.
- **Return**: Returns the head of the merged sorted linked list.

`void bubbleSort(int arr[], int size)`
- **Description**: Sorts an array of integers using the Bubble Sort algorithm.
- **Parameters**:
  - `arr`: Array of integers to be sorted.
  - `size`: Size of the array.

`void insertionSort(int arr[], int size)`
- **Description**: Sorts an array of integers using the Insertion Sort algorithm.
- **Parameters**:
  - `arr`: Array of integers to be sorted.
  - `size`: Size of the array.

`void selectionSort(int arr[], int size)`

- **Description**: Sorts an array of integers using the Selection Sort algorithm.
- **Parameters**:
  - `arr`: Array of integers to be sorted.
  - `size`: Size of the array.

`void mergeSort(int arr[], int left, int right)`
- **Description**: Sorts a specific range of an array of integers using the Merge Sort algorithm.
- **Parameters**:
  - `arr`: Array of integers to be sorted.
  - `left`: Index representing the start of the range.
  - `right`: Index representing the end of the range.

`void quickSort(int arr[], int low, int high)`
- **Description**: Sorts a specific range of an array of integers using the Quick Sort algorithm.
- **Parameters**:
  - `arr`: Array of integers to be sorted.
  - `low`: Index representing the start of the range.
  - `high`: Index representing the end of the range.

`void heapSort(int arr[], int size)`
- **Description**: Sorts an array of integers using the Heap Sort algorithm.
- **Parameters**:
  - `arr`: Array of integers to be sorted.
  - `size`: Size of the array.

`void cocktailSort(int arr[], int size)`
- **Description**: Sorts an array of integers using the Cocktail Shaker Sort (Bidirectional Bubble Sort) algorithm.
- **Parameters**:
  - `arr`: Array of integers to be sorted.
  - `size`: Size of the array.

`void combSort(int arr[], int size)`
- **Description**: Sorts an array of integers using the Comb Sort algorithm.
- **Parameters**:
  - `arr`: Array of integers to be sorted.
  - `size`: Size of the array.

`void gnomeSort(int arr[], int size)`
- **Description**: Sorts an array of integers using the Gnome Sort algorithm.
- **Parameters**:
  - `arr`: Array of integers to be sorted.
  - `size`: Size of the array.

`void cycleSort(int arr[], int size)`
- **Description**: Sorts an array of integers using the Cycle Sort algorithm.
- **Parameters**:
  - `arr`: Array of integers to be sorted.
  - `size`: Size of the array.

`int linearSearch(const int arr[], int size, int key)`
- **Description**: Searches for a key in an array using Linear Search.
- **Parameters**:
  - `arr`: Array of integers to be searched.
  - `size`: Size of the array.
  - `key`: Element to be searched.

`int binarySearch(const int arr[], int size, int key)`
- **Description**: Searches for a key in a sorted array using Binary Search.
- **Parameters**:
  - `arr`: Sorted array of integers to be searched.
  - `size`: Size of the array.
  - `key`: Element to be searched.

`int interpolationSearch(const int arr[], int size, int key)`
- **Description**: Searches for a key in a sorted array using Interpolation Search.
- **Parameters**:
  - `arr`: Sorted array of integers to be searched.
  - `size`: Size of the array.
  - `key`: Element to be searched.

`int jumpSearch(const int arr[], int size, int key)`
- **Description**: Searches for a key in a sorted array using Jump Search.
- **Parameters**:
  - `arr`: Sorted array of integers to be searched.
  - `size`: Size of the array.
  - `key`: Element to be searched.

`int exponentialSearch(const int arr[], int size, int key)`
- **Description**: Searches for a key in a sorted array using Exponential Search.
- **Parameters**:
  - `arr`: Sorted array of integers to be searched.
  - `size`: Size of the array.
  - `key`: Element to be searched.

`int fibonacciSearch(const int arr[], int size, int key)`

- **Description**: Searches for a key in a sorted array using Fibonacci Search.
- **Parameters**:
  - `arr`: Sorted array of integers to be searched.
  - `size`: Size of the array.
  - `key`: Element to be searched.

`int ternarySearch(const int arr[], int size, int key)`
- **Description**: Searches for a key in a sorted array using Ternary Search.
- **Parameters**:
  - `arr`: Sorted array of integers to be searched.
  - `size`: Size of the array.
  - `key`: Element to be searched.

`int linearProbing(int hashTable[], int size, int key)`
- **Description**: Searches for a key in a hash table using Linear Probing technique.
- **Parameters**:
  - `hashTable`: Hash table (array) of integers to be searched.
  - `size`: Size of the hash table.
  - `key`: Element to be searched.

`int quadraticProbing(int hashTable[], int size, int key)`
- **Description**: Searches for a key in a hash table using Quadratic Probing technique.
- **Parameters**:
  - `hashTable`: Hash table (array) of integers to be searched.
  - `size`: Size of the hash table.
  - `key`: Element to be searched.

`int doubleHashing(int hashTable[], int size, int key)`
- **Description**: Searches for a key in a hash table using Double Hashing technique.
- **Parameters**:
  - `hashTable`: Hash table (array) of integers to be searched.
  - `size`: Size of the hash table.
  - `key`: Element to be searched.

`void* customMalloc(size_t size)`
- **Description**: Allocates a block of memory of the specified size.
- **Parameters**:
  - `size`: Size of memory to allocate.

`void customFree(void* ptr)`
- **Description**: Deallocates the memory block previously allocated by `customMalloc`.
- **Parameters**:
  - `ptr`: Pointer to the memory block to free.

`void* customRealloc(void* ptr, size_t size)`
- **Description**: Changes the size of the memory block pointed to by `ptr`.
- **Parameters**:
  - `ptr`: Pointer to the memory block.
  - `size`: New size for the memory block.

`int getTotalSystemMemory()`
- **Description**: Retrieves the total amount of memory in the system.
- **Return**: Total system memory in bytes.

`int getAvailableMemory()`
- **Description**: Retrieves the amount of available memory in the system.
- **Return**: Available memory in bytes.

`void* allocateContiguousMemory(size_t size)`
- **Description**: Allocates a contiguous block of memory of the specified size.
- **Parameters**:
  - `size`: Size of memory to allocate.
- **Return**: Pointer to the allocated memory block.

`void* allocateVirtualMemory(size_t size)`
- **Description**: Allocates a virtual memory block of the specified size.
- **Parameters**:
  - `size`: Size of memory to allocate.
- **Return**: Pointer to the allocated virtual memory block.

`void deallocateMemory(void* ptr)`
- **Description**: Deallocates the memory block pointed to by `ptr`.
- **Parameters**:
  - `ptr`: Pointer to the memory block to deallocate.

`void* allocateAlignedMemory(size_t size, size_t alignment)`
- **Description**: Allocates aligned memory of the specified size and alignment.
- **Parameters**:
  - `size`: Size of memory to allocate.
  - `alignment`: Alignment of the memory.
- **Return**: Pointer to the allocated aligned memory block.

`void* allocateSharedMemory(size_t size, const char* name)`
- **Description**: Allocates shared memory of the specified size with a given name.
- **Parameters**:
  - `size`: Size of memory to allocate.

  - `name`: Name identifier for the shared memory.
- **Return**: Pointer to the allocated shared memory block.

`void handleInvalidInputError(const char* message)`
- **Description**: Handles errors related to invalid input with a provided error message.
- **Parameters**:
  - `message`: Error message describing the invalid input.

`void handleFileOpenError(const char* filename)`
- **Description**: Handles errors that occur during file opening.
- **Parameters**:
  - `filename`: Name of the file that encountered the error.

`void handleFileReadError(const char* filename)`
- **Description**: Handles errors related to reading from a file.
- **Parameters**:
  - `filename`: Name of the file that encountered the read error.

`void handleFileWriteError(const char* filename)`
- **Description**: Handles errors related to writing to a file.
- **Parameters**:
  - `filename`: Name of the file that encountered the write error.

`void handleMemoryAllocationError()`
- **Description**: Handles errors related to memory allocation failures.

`void handleNetworkError(int errorCode)`
- **Description**: Handles errors related to networking operations with a specific error code.
- **Parameters**:
  - `errorCode`: Error code indicating the type of network error.

`void handleDatabaseError(int errorCode)`
- **Description**: Handles errors related to database operations with a specific error code.
- **Parameters**:
  - `errorCode`: Error code indicating the type of database error.

`void handleThreadCreationError()`
- **Description**: Handles errors related to thread creation failures.

`void handleAssertionFailure(const char* expression)`
- **Description**: Handles errors due to failed assertions with the specific expression that failed.
- **Parameters**:
  - `expression`: The expression that failed the assertion.

`void handleUnexpectedError()`
- **Description**: Handles unexpected errors that do not fall under specific categories.

`bool isLeapYear(int year)`
- **Description**: Determines if a given year is a leap year.
- **Parameters**:
  - `year`: Year to check.

`bool isValidDate(int day, int month, int year)`
- **Description**: Checks if a given date is valid.
- **Parameters**:
  - `day`: Day of the month.
  - `month`: Month of the year.
  - `year`: Year.

`int getDaysInMonth(int month, int year)`
- **Description**: Retrieves the number of days in a given month of a specific year.
- **Parameters**:
  - `month`: Month of the year.
  - `year`: Year.

`int getDayOfWeek(int day, int month, int year)`
- **Description**: Determines the day of the week for a given date.
- **Parameters**:
  - `day`: Day of the month.
  - `month`: Month of the year.
  - `year`: Year.

`int calculateAge(int birthDay, int birthMonth, int birthYear, int currentDay, int currentMonth, int currentYear)`
- **Description**: Calculates the age based on the birthdate and current date.
- **Parameters**:
  - `birthDay`, `birthMonth`, `birthYear`: Date of birth (day, month, year).
  - `currentDay`, `currentMonth`, `currentYear`: Current date (day, month, year).

`bool isWeekend(int day, int month, int year)`
- **Description**: Checks if a given date falls on a weekend (Saturday or Sunday).
- **Parameters**:
  - `day`: Day of the month.
  - `month`: Month of the year.
  - `year`: Year.

`bool isSameDate(int day1, int month1, int year1, int day2, int month2, int year2)`
- **Description**: Checks if two dates are the same.
- **Parameters**:
  - `day1`, `month1`, `year1`: First date (day, month, year).
  - `day2`, `month2`, `year2`: Second date (day, month, year).

`bool isFutureDate(int day, int month, int year)`
- **Description**: Checks if a given date is in the future concerning the current date.
- **Parameters**:
  - `day`: Day of the month.
  - `month`: Month of the year.
  - `year`: Year.

`bool isPastDate(int day, int month, int year)`
- **Description**: Checks if a given date is in the past concerning the current date.
- **Parameters**:
  - `day`: Day of the month.
  - `month`: Month of the year.
  - `year`: Year.

`bool isToday(int day, int month, int year)`
- **Description**: Checks if a given date is today.
- **Parameters**:
  - `day`: Day of the month.
  - `month`: Month of the year.
  - `year`: Year.

`int generateRandomInt(int min, int max)`
- **Description**: Generates a random integer within a specified range.
- **Parameters**:
  - `min`: Minimum value for the range.
  - `max`: Maximum value for the range.

`double generateRandomDouble(double min, double max)`
- **Description**: Generates a random double within a specified range.
- **Parameters**:
  - `min`: Minimum value for the range.
  - `max`: Maximum value for the range.

`bool generateRandomBoolean()`
- **Description**: Generates a random boolean value (true or false).

`char generateRandomChar()`

- **Description**: Generates a random character.

`void generateRandomString(char* str, int length)`
- **Description**: Generates a random string of a specified length.
- **Parameters**:
  - `str`: Pointer to the string where the generated string will be stored.
  - `length`: Length of the string to be generated.

`int generateRandomInRange(int start, int end, int step)`
- **Description**: Generates a random integer within a specified range with a defined step.
- **Parameters**:
  - `start`: Starting value for the range.
  - `end`: Ending value for the range.
  - `step`: Difference between each number in the range.

`double generateGaussianRandom(double mean, double stddev)`
- **Description**: Generates a random number using a Gaussian distribution with a specified mean and standard deviation.
- **Parameters**:
  - `mean`: Mean value of the distribution.
  - `stddev`: Standard deviation of the distribution.

`int generateRandomPrime(int min, int max)`
- **Description**: Generates a random prime number within a specified range.
- **Parameters**:
  - `min`: Minimum value for the range.
  - `max`: Maximum value for the range.

`int generateFibonacciRandom(int min, int max)`
- **Description**: Generates a random number within a specified range that is a member of the Fibonacci sequence.
- **Parameters**:
  - `min`: Minimum value for the range.
  - `max`: Maximum value for the range.

`void seedRandomGenerator(unsigned int seed)`
- **Description**: Seeds the random number generator with a specific seed value.
- **Parameters**:
  - `seed`: Seed value for the random number generator.

Queue Operations:

`void initializeQueue(Queue& q)`
- **Description**: Initializes an empty queue.
- **Parameters**:
  - `q`: Reference to the queue to be initialized.

`void enqueue(Queue& q, int value)`
- **Description**: Adds an element to the back of the queue.
- **Parameters**:
  - `q`: Reference to the queue.
  - `value`: Value to be added to the queue.

`int dequeue(Queue& q)`
- **Description**: Removes and returns the element from the front of the queue.
- **Parameters**:
  - `q`: Reference to the queue.
- **Returns**: The value dequeued from the queue.

`bool isQueueEmpty(const Queue& q)`
- **Description**: Checks if the queue is empty.
- **Parameters**:
  - `q`: Reference to the queue.
- **Returns**: Boolean value (`true` if the queue is empty, `false` otherwise).

`int getQueueSize(const Queue& q)`
- **Description**: Retrieves the current size of the queue.
- **Parameters**:
  - `q`: Reference to the queue.
- **Returns**: The number of elements present in the queue.

`void clearQueue(Queue& q)`
- **Description**: Clears all elements from the queue.
- **Parameters**:
  - `q`: Reference to the queue.

Stack Operations:

`void initializeStack(Stack& s)`
- **Description**: Initializes an empty stack.
- **Parameters**:
  - `s`: Reference to the stack to be initialized.

`void push(Stack& s, int value)`
- **Description**: Pushes an element onto the top of the stack.
- **Parameters**:
  - `s`: Reference to the stack.
  - `value`: Value to be pushed onto the stack.

`int pop(Stack& s)`
- **Description**: Removes and returns the element from the top of the stack.
- **Parameters**:
  - `s`: Reference to the stack.
- **Returns**: The value popped from the stack.

`bool isStackEmpty(const Stack& s)`
- **Description**: Checks if the stack is empty.
- **Parameters**:
  - `s`: Reference to the stack.
- **Returns**: Boolean value (`true` if the stack is empty, `false` otherwise).

Stack Operations:

`void push(Stack& s, int value)`
- **Description**: Pushes an element onto the top of the stack.
- **Parameters**:
  - `s`: Reference to the stack.
  - `value`: Value to be pushed onto the stack.

`int pop(Stack& s)`
- **Description**: Removes and returns the element from the top of the stack.
- **Parameters**:
  - `s`: Reference to the stack.
- **Returns**: The value popped from the stack.

`int peek(const Stack& s)`
- **Description**: Returns the value at the top of the stack without removing it.
- **Parameters**:
  - `s`: Reference to the stack.
- **Returns**: The value at the top of the stack.

`bool isStackEmpty(const Stack& s)`
- **Description**: Checks if the stack is empty.
- **Parameters**:
  - `s`: Reference to the stack.
- **Returns**: Boolean value (`true` if the stack is empty, `false` otherwise).

`int getStackSize(const Stack& s)`
- **Description**: Retrieves the current size of the stack.
- **Parameters**:
  - `s`: Reference to the stack.
- **Returns**: The number of elements present in the stack.

`void clearStack(Stack& s)`
- **Description**: Clears all elements from the stack.
- **Parameters**:
  - `s`: Reference to the stack.

`bool isPalindrome(const char* str)`
- **Description**: Checks if the given string is a palindrome.
- **Parameters**:
  - `str`: Pointer to the string to be checked.
- **Returns**: Boolean value (`true` if the string is a palindrome, `false` otherwise).

`int evaluatePostfixExpression(const char* expression)`
- **Description**: Evaluates a postfix expression.
- **Parameters**:
  - `expression`: Pointer to the postfix expression.
- **Returns**: Result of the evaluated expression.

`void convertInfixToPostfix(const char* infix, char* postfix)`
- **Description**: Converts an infix expression to postfix.
- **Parameters**:
  - `infix`: Pointer to the infix expression.
  - `postfix`: Pointer to store the resulting postfix expression.

`bool isBalancedParentheses(const char* expression)`
- **Description**: Checks if parentheses in the expression are balanced.
- **Parameters**:
  - `expression`: Pointer to the expression.
- **Returns**: Boolean value (`true` if parentheses are balanced, `false` otherwise).

```
/**
 * Adds an element to the end of the queue.
 * @param q: Reference to the Queue.
 * @param value: Value to be added.
 */
void enqueue(Queue& q, int value);
```

```
/**
 * Removes an element from the front of the queue.
 * @param q: Reference to the Queue.
 * @return: Value removed from the queue.
 */
int dequeue(Queue& q);

/**
 * Retrieves the front element of the queue.
 * @param q: Reference to the Queue.
 * @return: Value at the front of the queue.
 */
int getFront(const Queue& q);

/**
 * Checks if the queue is empty.
 * @param q: Reference to the Queue.
 * @return: True if the queue is empty, otherwise false.
 */
bool isQueueEmpty(const Queue& q);

/**
 * Retrieves the size of the queue.
 * @param q: Reference to the Queue.
 * @return: Size of the queue.
 */
int getQueueSize(const Queue& q);

/**
 * Clears all elements from the queue.
 * @param q: Reference to the Queue.
 */
void clearQueue(Queue& q);

/**
 * Checks if the circular queue is empty.
 * @param cq: Reference to the CircularQueue.
 * @return: True if the circular queue is empty, otherwise false.
 */
bool isCircularQueueEmpty(const CircularQueue& cq);

/**
```

```
 * Retrieves the size of the circular queue.
 * @param cq: Reference to the CircularQueue.
 * @return: Size of the circular queue.
 */
int getCircularQueueSize(const CircularQueue& cq);

/**
 * Adds an element to the end of the circular queue.
 * @param cq: Reference to the CircularQueue.
 * @param value: Value to be added.
 */
void enqueueCircularQueue(CircularQueue& cq, int value);

/**
 * Removes an element from the front of the circular queue.
 * @param cq: Reference to the CircularQueue.
 * @return: Value removed from the circular queue.
 */
int dequeueCircularQueue(CircularQueue& cq);



/**
 * Creates a hash table with the specified size.
 * @param table: Reference to the HashTable.
 * @param size: Size of the hash table to be created.
 */
void createHashTable(HashTable& table, int size);

/**
 * Inserts a key-value pair into the hash table.
 * @param table: Reference to the HashTable.
 * @param key: Key to be inserted.
 * @param value: Value corresponding to the key.
 */
void insertIntoHashTable(HashTable& table, int key, int value);

/**
 * Searches for a key in the hash table.
 * @param table: Reference to the HashTable.
 * @param key: Key to search for.
 * @param value: Reference to the variable to store the found value (if key exists).
 * @return: True if the key is found, otherwise false.
 */
```

```
bool searchInHashTable(const HashTable& table, int key, int& value);

/**
 * Removes a key-value pair from the hash table.
 * @param table: Reference to the HashTable.
 * @param key: Key to be removed.
 */
void removeFromHashTable(HashTable& table, int key);

/**
 * Clears all elements from the hash table.
 * @param table: Reference to the HashTable.
 */
void clearHashTable(HashTable& table);

/**
 * Retrieves the number of elements in the hash table.
 * @param table: Reference to the HashTable.
 * @return: Number of elements in the hash table.
 */
int getHashTableSize(const HashTable& table);

/**
 * Retrieves the capacity (total size) of the hash table.
 * @param table: Reference to the HashTable.
 * @return: Capacity of the hash table.
 */
int getHashTableCapacity(const HashTable& table);

/**
 * Checks if the hash table is empty.
 * @param table: Reference to the HashTable.
 * @return: True if the hash table is empty, otherwise false.
 */
bool isHashTableEmpty(const HashTable& table);

/**
 * Resizes the hash table to a new specified size.
 * @param table: Reference to the HashTable.
 * @param newSize: New size for the hash table.
 */
void resizeHashTable(HashTable& table, int newSize);
```

```
/**
 * Calculates the load factor of the hash table.
 * @param table: Reference to the HashTable.
 * @return: Load factor of the hash table.
 */
float getLoadFactor(const HashTable& table);

/**
 * Creates a graph with the specified number of vertices.
 * @param graph: Reference to the Graph.
 * @param vertices: Number of vertices for the graph.
 */
void createGraph(Graph& graph, int vertices);

/**
 * Adds an edge with weight between source and destination vertices in the graph.
 * @param graph: Reference to the Graph.
 * @param src: Source vertex.
 * @param dest: Destination vertex.
 * @param weight: Weight of the edge.
 */
void addEdge(Graph& graph, int src, int dest, int weight);

/**
 * Checks if there is an edge between source and destination vertices in the graph.
 * @param graph: Reference to the Graph.
 * @param src: Source vertex.
 * @param dest: Destination vertex.
 * @return: True if there is an edge, otherwise false.
 */
bool hasEdge(const Graph& graph, int src, int dest);

/**
 * Removes an edge between source and destination vertices in the graph.
 * @param graph: Reference to the Graph.
 * @param src: Source vertex.
 * @param dest: Destination vertex.
 */
void removeEdge(Graph& graph, int src, int dest);

/**
 * Checks if the graph is connected.
 * @param graph: Reference to the Graph.
```

```
 * @return: True if the graph is connected, otherwise false.
 */
bool isGraphConnected(const Graph& graph);

/**
 * Checks if the graph contains a cycle.
 * @param graph: Reference to the Graph.
 * @return: True if the graph contains a cycle, otherwise false.
 */
bool isCyclic(const Graph& graph);

/**
 * Performs Depth-First Search (DFS) traversal starting from a given vertex.
 * @param graph: Reference to the Graph.
 * @param startVertex: Starting vertex for DFS.
 */
void depthFirstSearch(const Graph& graph, int startVertex);

/**
 * Performs Breadth-First Search (BFS) traversal starting from a given vertex.
 * @param graph: Reference to the Graph.
 * @param startVertex: Starting vertex for BFS.
 */
void breadthFirstSearch(const Graph& graph, int startVertex);

/**
 * Finds the shortest path between source and destination vertices in the graph.
 * @param graph: Reference to the Graph.
 * @param src: Source vertex.
 * @param dest: Destination vertex.
 * @param path: Pointer to an array to store the shortest path.
 * @return: Length of the shortest path.
 */
int getShortestPath(const Graph& graph, int src, int dest, int* path);

/**
 * Finds the Minimum Spanning Tree (MST) of the graph using Kruskal's or Prim's algorithm.
 * @param graph: Reference to the Graph.
 * @param mst: Reference to the Graph to store the Minimum Spanning Tree.
 * @return: Weight of the Minimum Spanning Tree.
 */
int getMinimumSpanningTree(const Graph& graph, Graph& mst);
```

```
/**
 * Creates a new tree node with the given value.
 * @param value: Value to be assigned to the node.
 * @return: Pointer to the created TreeNode.
 */
TreeNode* createTreeNode(int value);

/**
 * Inserts a node with the given value into the binary search tree.
 * @param root: Reference to the root of the tree.
 * @param value: Value to be inserted.
 */
void insertNode(TreeNode*& root, int value);

/**
 * Searches for a node with the given value in the binary search tree.
 * @param root: Pointer to the root of the tree.
 * @param value: Value to search for.
 * @return: True if the value is found, otherwise false.
 */
bool searchNode(TreeNode* root, int value);

/**
 * Deletes a node with the given value from the binary search tree.
 * @param root: Reference to the root of the tree.
 * @param value: Value to be deleted.
 */
void deleteNode(TreeNode*& root, int value);

/**
 * Gets the height of the binary tree.
 * @param root: Pointer to the root of the tree.
 * @return: Height of the tree.
 */
int getTreeHeight(TreeNode* root);

/**
 * Counts the number of nodes in the binary tree.
 * @param root: Pointer to the root of the tree.
 * @return: Number of nodes in the tree.
 */
int getTreeNodeCount(TreeNode* root);
```

```
/**
 * Performs in-order traversal of the binary tree.
 * @param root: Pointer to the root of the tree.
 */
void inOrderTraversal(TreeNode* root);

/**
 * Performs pre-order traversal of the binary tree.
 * @param root: Pointer to the root of the tree.
 */
void preOrderTraversal(TreeNode* root);

/**
 * Performs post-order traversal of the binary tree.
 * @param root: Pointer to the root of the tree.
 */
void postOrderTraversal(TreeNode* root);

/**
 * Performs level-order traversal of the binary tree.
 * @param root: Pointer to the root of the tree.
 */
void levelOrderTraversal(TreeNode* root);



/**
 * Swaps the values of two integers.
 * @param a: Reference to the first integer.
 * @param b: Reference to the second integer.
 */
void swap(int& a, int& b);

/**
 * Returns the maximum of two integers.
 * @param a: First integer.
 * @param b: Second integer.
 * @return: Maximum of the two integers.
 */
int getMax(int a, int b);

/**
 * Returns the minimum of two integers.
 * @param a: First integer.
```

```
 * @param b: Second integer.
 * @return: Minimum of the two integers.
 */
int getMin(int a, int b);

/**
 * Checks if a number is a power of two.
 * @param num: Number to be checked.
 * @return: True if the number is a power of two, otherwise false.
 */
bool isPowerOfTwo(int num);

/**
 * Returns the absolute value of a number.
 * @param num: Number to find the absolute value of.
 * @return: Absolute value of the number.
 */
int abs(int num);

/**
 * Checks if a number is even.
 * @param num: Number to be checked.
 * @return: True if the number is even, otherwise false.
 */
bool isEven(int num);

/**
 * Checks if a number is odd.
 * @param num: Number to be checked.
 * @return: True if the number is odd, otherwise false.
 */
bool isOdd(int num);

/**
 * Checks if a character is an alphabet.
 * @param ch: Character to be checked.
 * @return: True if the character is an alphabet, otherwise false.
 */
bool isAlpha(char ch);

/**
 * Checks if a character is a digit.
 * @param ch: Character to be checked.
```

```
 * @return: True if the character is a digit, otherwise false.
 */
bool isDigit(char ch);

/**
 * Converts a character to its uppercase equivalent.
 * @param ch: Character to be converted.
 * @return: Uppercase equivalent of the character.
 */
char toUpperCase(char ch);

/**
 * Performs bitwise AND operation between two integers.
 * @param a: First integer.
 * @param b: Second integer.
 * @return: Result of the bitwise AND operation.
 */
int bitwiseAnd(int a, int b);

/**
 * Performs bitwise OR operation between two integers.
 * @param a: First integer.
 * @param b: Second integer.
 * @return: Result of the bitwise OR operation.
 */
int bitwiseOr(int a, int b);

/**
 * Performs bitwise XOR operation between two integers.
 * @param a: First integer.
 * @param b: Second integer.
 * @return: Result of the bitwise XOR operation.
 */
int bitwiseXor(int a, int b);

/**
 * Performs bitwise NOT operation on an integer.
 * @param num: Integer to perform bitwise NOT.
 * @return: Result of the bitwise NOT operation.
 */
int bitwiseNot(int num);

/**
```

```
 * Performs left shift operation on an integer.
 * @param num: Integer to perform left shift.
 * @param shiftAmount: Number of positions to left shift.
 * @return: Result of the left shift operation.
 */
int leftShift(int num, int shiftAmount);

/**
 * Performs right shift operation on an integer.
 * @param num: Integer to perform right shift.
 * @param shiftAmount: Number of positions to right shift.
 * @return: Result of the right shift operation.
 */
int rightShift(int num, int shiftAmount);

/**
 * Checks if a bit is set at a specific position in an integer.
 * @param num: Integer to check for the bit.
 * @param position: Bit position to check.
 * @return: True if the bit is set, otherwise false.
 */
bool isBitSet(int num, int position);

/**
 * Sets a bit at a specific position in an integer.
 * @param num: Integer in which the bit is to be set.
 * @param position: Bit position to set.
 * @return: Integer with the bit set at the specified position.
 */
int setBit(int num, int position);

/**
 * Clears a bit at a specific position in an integer.
 * @param num: Integer in which the bit is to be cleared.
 * @param position: Bit position to clear.
 * @return: Integer with the bit cleared at the specified position.
 */
int clearBit(int num, int position);

/**
 * Toggles a bit at a specific position in an integer.
 * @param num: Integer in which the bit is to be toggled.
 * @param position: Bit position to toggle.
```

```
 * @return: Integer with the bit toggled at the specified position.
 */
int toggleBit(int num, int position);

/**
 * Sorts the array using Bitonic Sort algorithm.
 * @param arr: Array to be sorted.
 * @param size: Size of the array.
 * @param ascending: Boolean flag for sorting order (true for ascending, false for descending).
 */
void bitonicSort(int arr[], int size, bool ascending);

/**
 * Sorts the array using Odd-Even Merge Sort algorithm.
 * @param arr: Array to be sorted.
 * @param size: Size of the array.
 * @param ascending: Boolean flag for sorting order (true for ascending, false for descending).
 */
void oddEvenMergeSort(int arr[], int size, bool ascending);

/**
 * Sorts the array using Bogo Sort algorithm.
 * @param arr: Array to be sorted.
 * @param size: Size of the array.
 * @param ascending: Boolean flag for sorting order (true for ascending, false for descending).
 */
void bogoSort(int arr[], int size, bool ascending);

/**
 * Sorts the array using Pancake Sort algorithm.
 * @param arr: Array to be sorted.
 * @param size: Size of the array.
 * @param ascending: Boolean flag for sorting order (true for ascending, false for descending).
 */
void pancakeSort(int arr[], int size, bool ascending);

/**
 * Sorts the array using Stooge Sort algorithm.
 * @param arr: Array to be sorted.
 * @param size: Size of the array.
 * @param ascending: Boolean flag for sorting order (true for ascending, false for descending).
 */
void stoogeSort(int arr[], int size, bool ascending);
```

```
/**
 * Sorts the array using Sleep Sort algorithm.
 * @param arr: Array to be sorted (mainly containing non-negative integers).
 * @param size: Size of the array.
 * @param ascending: Boolean flag for sorting order (true for ascending, false for descending).
 */
void sleepSort(int arr[], int size, bool ascending);

/**
 * Sorts the array using Cycle Sort algorithm.
 * @param arr: Array to be sorted.
 * @param size: Size of the array.
 * @param ascending: Boolean flag for sorting order (true for ascending, false for descending).
 */
void cycleSort(int arr[], int size, bool ascending);

/**
 * Sorts the array using Cocktail Shaker Sort (Bidirectional Bubble Sort) algorithm.
 * @param arr: Array to be sorted.
 * @param size: Size of the array.
 * @param ascending: Boolean flag for sorting order (true for ascending, false for descending).
 */
void cocktailSort(int arr[], int size, bool ascending);

/**
 * Sorts the array using Gnome Sort (Stupid Sort) algorithm.
 * @param arr: Array to be sorted.
 * @param size: Size of the array.
 * @param ascending: Boolean flag for sorting order (true for ascending, false for descending).
 */
void gnomeSort(int arr[], int size, bool ascending);

/**
 * Sorts the array using Stooge Sort algorithm (recursive sorting algorithm).
 * @param arr: Array to be sorted.
 * @param size: Size of the array.
 * @param ascending: Boolean flag for sorting order (true for ascending, false for descending).
 */
void stoogeSort(int arr[], int size, bool ascending);


/**
```

```
 * Adds two matrices and stores the result in another matrix.
 * @param matrixA: Pointer to the first matrix.
 * @param matrixB: Pointer to the second matrix.
 * @param result: Pointer to the matrix to store the result.
 * @param rows: Number of rows in the matrices.
 * @param cols: Number of columns in the matrices.
 */
void addMatrices(int** matrixA, int** matrixB, int** result, int rows, int cols);

/**
 * Subtracts one matrix from another and stores the result in another matrix.
 * @param matrixA: Pointer to the first matrix.
 * @param matrixB: Pointer to the second matrix.
 * @param result: Pointer to the matrix to store the result.
 * @param rows: Number of rows in the matrices.
 * @param cols: Number of columns in the matrices.
 */
void subtractMatrices(int** matrixA, int** matrixB, int** result, int rows, int cols);

/**
 * Multiplies two matrices and stores the result in another matrix.
 * @param matrixA: Pointer to the first matrix.
 * @param matrixB: Pointer to the second matrix.
 * @param result: Pointer to the matrix to store the result.
 * @param rowsA: Number of rows in the first matrix.
 * @param colsA: Number of columns in the first matrix (should be equal to rowsB).
 * @param rowsB: Number of rows in the second matrix.
 * @param colsB: Number of columns in the second matrix.
 */
void multiplyMatrices(int** matrixA, int** matrixB, int** result, int rowsA, int colsA, int rowsB, int
colsB);

/**
 * Transposes a matrix and stores the result in another matrix.
 * @param matrix: Pointer to the matrix to be transposed.
 * @param result: Pointer to the matrix to store the transposed result.
 * @param rows: Number of rows in the matrix.
 * @param cols: Number of columns in the matrix.
 */
void transposeMatrix(int** matrix, int** result, int rows, int cols);

/**
 * Calculates the determinant of a square matrix.
```

```
 * @param matrix: Pointer to the square matrix.
 * @param size: Size of the square matrix (number of rows or columns).
 * @return: Determinant of the matrix.
 */
int determinant(int** matrix, int size);

/**
 * Inverts a square matrix and stores the result in another matrix.
 * @param matrix: Pointer to the square matrix to be inverted.
 * @param result: Pointer to the matrix to store the inverted result.
 * @param size: Size of the square matrix (number of rows or columns).
 */
void invertMatrix(int** matrix, int** result, int size);

/**
 * Creates an identity matrix and stores it in another matrix.
 * @param matrix: Pointer to the matrix to store the identity matrix.
 * @param size: Size of the square matrix (number of rows or columns).
 */
void identityMatrix(int** matrix, int size);

/**
 * Checks if a square matrix is symmetric.
 * @param matrix: Pointer to the square matrix.
 * @param size: Size of the square matrix (number of rows or columns).
 * @return: True if the matrix is symmetric, otherwise false.
 */
bool isSymmetric(int** matrix, int size);

/**
 * Checks if a matrix is sparse (contains mostly zeroes).
 * @param matrix: Pointer to the matrix.
 * @param rows: Number of rows in the matrix.
 * @param cols: Number of columns in the matrix.
 * @return: True if the matrix is sparse, otherwise false.
 */
bool isSparse(int** matrix, int rows, int cols);

/**
 * Rotates a square matrix clockwise by 90 degrees.
 * @param matrix: Pointer to the square matrix to be rotated.
 * @param size: Size of the square matrix (number of rows or columns).
 */
```

void rotateMatrixClockwise(int** matrix, int size);

Documenting the functions in `VectorUtils.h`:

1. `float dotProduct(const std::vector<float>& vectorA, const std::vector<float>& vectorB)`
- **Description:** Calculates the dot product of two input vectors.
- **Parameters:**
  - `vectorA`: First vector.
  - `vectorB`: Second vector.
- **Return Value:** Dot product of the input vectors.

2. `std::vector<float> crossProduct(const std::vector<float>& vectorA, const std::vector<float>& vectorB)`
- **Description:** Computes the cross product of two input vectors.
- **Parameters:**
  - `vectorA`: First vector.
  - `vectorB`: Second vector.
- **Return Value:** Cross product of the input vectors.

3. `std::vector<float> scalarMultiply(const std::vector<float>& vector, float scalar)`
- **Description:** Multiplies a vector by a scalar value.
- **Parameters:**
  - `vector`: Input vector.
  - `scalar`: Scalar value.
- **Return Value:** Vector resulting from scalar multiplication.

4. `std::vector<float> normalize(const std::vector<float>& vector)`
- **Description:** Normalizes a vector to unit length.
- **Parameter:**
  - `vector`: Input vector to be normalized.
- **Return Value:** Normalized vector.

5. `float magnitude(const std::vector<float>& vector)`
- **Description:** Calculates the magnitude of a vector.
- **Parameter:**
  - `vector`: Input vector.
- **Return Value:** Magnitude of the input vector.

6. `std::vector<float> addVectors(const std::vector<float>& vectorA, const std::vector<float>& vectorB)`
- **Description:** Adds two input vectors element-wise.

- **Parameters:**
  - `vectorA`: First input vector.
  - `vectorB`: Second input vector.
- **Return Value:** Resultant vector after element-wise addition.

7. `std::vector<float> subtractVectors(const std::vector<float>& vectorA, const std::vector<float>& vectorB)`
- **Description:** Subtracts one input vector from another element-wise.
- **Parameters:**
  - `vectorA`: Input vector from which `vectorB` will be subtracted.
  - `vectorB`: Input vector to subtract from `vectorA`.
- **Return Value:** Resultant vector after element-wise subtraction.

8. `float angleBetweenVectors(const std::vector<float>& vectorA, const std::vector<float>& vectorB)`
- **Description:** Calculates the angle between two input vectors.
- **Parameters:**
  - `vectorA`: First vector.
  - `vectorB`: Second vector.
- **Return Value:** Angle in radians between the input vectors.

9. `bool areCollinear(const std::vector<float>& vectorA, const std::vector<float>& vectorB)`
- **Description:** Checks if two input vectors are collinear.
- **Parameters:**
  - `vectorA`: First input vector.
  - `vectorB`: Second input vector.
- **Return Value:** `true` if vectors are collinear; otherwise, `false`.

10. `bool areOrthogonal(const std::vector<float>& vectorA, const std::vector<float>& vectorB)`
- **Description:** Checks if two input vectors are orthogonal (perpendicular).
- **Parameters:**
  - `vectorA`: First input vector.
  - `vectorB`: Second input vector.
- **Return Value:** `true` if vectors are orthogonal; otherwise, `false`.

Specified functions in `TreeAlgorithms.h`:

1. `int treeHeight(const TreeNode* root)`
- **Description:** Calculates the height of the tree rooted at the given node.
- **Parameters:**
  - `root`: Pointer to the root of the tree.
- **Return Value:** Height of the tree.

2. `int treeSize(const TreeNode* root)`
  - **Description:** Computes the total number of nodes in the tree.
  - **Parameters:**
    - `root`: Pointer to the root of the tree.
  - **Return Value:** Total number of nodes in the tree.

3. `int leafCount(const TreeNode* root)`
  - **Description:** Counts the number of leaf nodes in the tree.
  - **Parameters:**
    - `root`: Pointer to the root of the tree.
  - **Return Value:** Number of leaf nodes in the tree.

4. `int levelWidth(const TreeNode* root, int level)`
  - **Description:** Computes the width of the given level in the tree.
  - **Parameters:**
    - `root`: Pointer to the root of the tree.
    - `level`: Level for which width needs to be calculated.
  - **Return Value:** Width of the specified level in the tree.

5. `bool isFullBinaryTree(const TreeNode* root)`
  - **Description:** Checks if the given tree is a full binary tree.
  - **Parameters:**
    - `root`: Pointer to the root of the tree.
  - **Return Value:** `true` if the tree is a full binary tree; otherwise, `false`.

6. `bool isCompleteBinaryTree(const TreeNode* root)`
  - **Description:** Determines if the given tree is a complete binary tree.
  - **Parameters:**
    - `root`: Pointer to the root of the tree.
  - **Return Value:** `true` if the tree is a complete binary tree; otherwise, `false`.

7. `bool areStructurallyIdentical(const TreeNode* root1, const TreeNode* root2)`
  - **Description:** Checks whether two trees rooted at `root1` and `root2` are structurally identical.
  - **Parameters:**
    - `root1`: Pointer to the root of the first tree.
    - `root2`: Pointer to the root of the second tree.
  - **Return Value:** `true` if the trees are structurally identical; otherwise, `false`.

8. `bool areMirrorImages(const TreeNode* root1, const TreeNode* root2)`
  - **Description:** Determines if two trees rooted at `root1` and `root2` are mirror images of each other.
  - **Parameters:**
    - `root1`: Pointer to the root of the first tree.

  - `root2`: Pointer to the root of the second tree.
- **Return Value:** `true` if the trees are mirror images; otherwise, `false`.

9. `TreeNode* lowestCommonAncestor(const TreeNode* root, int value1, int value2)`
- **Description:** Finds the lowest common ancestor of nodes with values `value1` and `value2`.
- **Parameters:**
  - `root`: Pointer to the root of the tree.
  - `value1`: Value of the first node.
  - `value2`: Value of the second node.
- **Return Value:** Pointer to the lowest common ancestor node.

10. `bool isBST(const TreeNode* root)`
- **Description:** Checks if the given tree is a binary search tree (BST).
- **Parameters:**
  - `root`: Pointer to the root of the tree.
- **Return Value:** `true` if the tree is a BST; otherwise, `false`.