**Main Function: Menu Loop**

* **Initialize CSV File Path**
  + Read from command-line arguments or use default if none provided
* **Menu Loop**: Continue until menuChoice equals '9'
  + **Display menu options** and prompt user for:
    - menuChoice: Action to perform (1: Load Data, 2: Validate, 3: Search, 4: Print Courses, 9: Exit)
    - dataChoice: Data structure to use (Vector, Hash Table, BinarySearchTree)
  + **Validate Input**: Ensure menuChoice is in {1-4, 9}; otherwise, display an error

**Execute Actions Based on menuChoice**:

* + **1: Load Data**
    - Load CSV file into the chosen data structure:
      * BinarySearchTree: loadBids() into bst
      * Vector: loadBids() into courseList
      * Hash Table: loadBids() into courseTable
    - Output number of records loaded
  + **2: Validate Data Structure**
    - Call respective validation functions based on dataChoice
  + **3: Search and Display Course Information**
    - Prompt for userSearch
    - Call appropriate search function for the selected data structure
  + **4: Print Courses in Alphanumeric Order**
    - Display sorted course list using the relevant data structure method
  + **9: Exit Program**
    - Display a farewell message and terminate the loop

**Course Struct**

* **Attributes**: courseID, courseName, preCount (number of prerequisites), preList (list of prerequisites)
* **Constructor**: Initializes all attributes to default values

**Data Structure Classes**

1. **BinarySearchTree Class**
   * **Node Struct**: Contains Course data, left, and right pointers
   * **printTree() Function**: Performs an in-order traversal to display courses and their prerequisites
2. **Vector Functions**
   * **sortList()**: Uses a QuickSort-like algorithm for efficient sorting
   * **printList()**: Iterates through the sorted vector, displaying each course and its prerequisites
3. **Hash Table Functions**
   * **printTable()**: Iterates through each hash table bucket to display course data, handling collisions gracefully

**Run Time Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| **Loading Data** | O(1) | O(1) – O(N)\* | O(log N) |
| **Search** | O(N) | O(1) – O(N)\* | O(log N) – O(N) |
| **Sort/Print** | O(N log N) | O(N)\* | O(N) |

\*Performance depends on hash collisions (for Hash Table) or tree balance (for Binary Tree).

**Analysis and Recommendation**

* **Vector**: Efficient for data loading but requires sorting for ordered data, making it slower for search operations.
* **Hash Table**: Provides the fastest search time in ideal scenarios but can degrade with poor hash functions causing collisions.
* **Binary Tree**: Offers balanced performance with logarithmic search times; however, efficiency drops if the tree becomes unbalanced.

**Conclusion**

* **Recommended Data Structure**: **Hash Table**
  + Chosen for its superior average-case search efficiency, assuming a well-optimized hash function to reduce collision frequency.
  + Its performance is particularly advantageous when frequent lookups are necessary, as specified in the program requirements.