Everett De Bree

Ramsay Kraya

CS-300

August 15, 2025

Project One

**File Input**

Start by opening the course data file

If the file fails to open

    Display an error message and end the process

Read each line from the file one at a time

    Separate the line into parts using commas

    If the number of parts is not correct

        Output a formatting error and move to the next line

    Otherwise

        Temporarily hold the separated values for later steps

**Storing Data in a Vector**

Define a Course object that includes:

    course ID

    course name

    list of prerequisites (may be empty)

As each properly formatted line is processed

    Create a new instance of the Course object

    Assign the ID, name, and prerequisites using the parsed values

    Insert this Course object into a vector named courseCatalog

After loading all lines

    For every Course object in courseCatalog

        For each prerequisite listed

            Check that the prerequisite course ID exists somewhere in courseCatalog

            If not found, print an error noting the missing prerequisite

**Searching and Printing Course Info**

Ask the user to input the course ID they wish to look up

For every Course object in the courseCatalog vector

    Check if the course ID matches the user input

    If a match is found

        Print the course name

        If there are prerequisites

            Show the list of prerequisites

        Otherwise

            Indicate that there are no prerequisites

        Exit the loop

If the loop completes with no match

    Display a message saying the course could not be located

**Printing All Courses in Alphanumeric Order (Vector)**

Make a copy of courseCatalog

Sort the copy by course ID in ascending alphanumeric order

For each course in the sorted copy

    Print the course ID and course title

File Input and Validation (Hash Table)

Start by opening the course data file

If the file fails to open

    Display an error message and end the process

Create an empty hash table with a fixed number of buckets

Create an empty set (or temporary list) to track all course IDs encountered

Read each line from the file one at a time

    Trim whitespace; if the line is empty, continue to the next line

    Separate the line into parts using commas

    If fewer than two parts are present (course ID and title)

        Output a formatting error and move to the next line

    Otherwise

        Create a Course object with: ID, title, and remaining parts as prerequisites

        Insert this Course into the hash table using Insert() (defined below)

        Add the course ID to the set of known IDs

After loading all lines

    For each course currently stored in the hash table

        For each prerequisite in that course’s list

            Check that the prerequisite exists in the set of known IDs

            If not found, print an error noting the missing prerequisite

**Insert() (Chaining)**

Create a new node using the given course

Use the course number (ID) to calculate the hash index

If no node exists at that index:

    Insert the new node there

Else:

    Add the new node to the front of the linked list at that index (for chaining)

Increase size counter

**Remove() (Chaining)**

Calculate the index from the course number

Start at the head of the chain at that index

If head node matches courseNum:

    Move head to next node

    Delete original head

    Decrease size

    Return

While next node exists:

    If next node matches courseNum:

        Store next in temp

        Set current’s next to temp’s next

        Delete temp

        Decrease size

        Return

    Move to next node

**Search() (Chaining)**

Calculate hash index using course number

Start at the head of that index’s chain

While current node exists:

    If courseNum matches:

        Return the course

    Move to next node

If not found, return an empty course object

**PrintCourse() (Hash Table)**

Search for the course using courseNum

If found:

    Print course number and title

    If prerequisites exist:

        Print each one

    Else:

        Print “No prerequisites”

If not found:

    Print “Course not found”

**PrintAllCourses() in Alphanumeric Order (Hash Table)**

Create an empty temporary list

For each bucket in the hash table

    For each node in that bucket’s chain

        Append the node’s course to the temporary list

Sort the temporary list by course ID in ascending alphanumeric order

For each course in the sorted list

    Print the course ID and course title

**File Input and Validation (BST)**

Start by opening the course data file

If the file fails to open

    Display an error message and end the process

Create an empty binary search tree named courseTree

Create an empty list named allCourseIDs

Read each line from the file one at a time

    Remove any leading or trailing spaces

    If the line is empty, continue to the next line

    Separate the line into parts using commas

    If fewer than two parts are present (course ID and title)

        Output a formatting error and move to the next line

    Otherwise

        Create a Course object with the ID, title, and any remaining parts as prerequisites

        Insert the Course object into courseTree in alphanumeric order by course ID

        Add the course ID to the allCourseIDs list

After loading all lines

    For each course in the tree

        For each prerequisite in that course’s list

            If the prerequisite is not in allCourseIDs

            Display an error message noting the missing prerequisite

**Storing Data in the BST**

Define a Course object that includes:

    course ID

    course name

    list of prerequisites (may be empty)

Define a Node object that includes:

    Course object

    Left child pointer

    Right child pointer

When inserting a Course into the BST

    If the tree is empty, set the new Course as the root

    If the course ID is less than the current node’s ID, move left

    If the course ID is greater, move right

    Repeat until the correct position is found, then insert the new node

**Searching and Printing a Single Course (BST)**

Ask the user for a course ID

Start at the root of the tree

While the current node is not null

    If the current node’s ID matches the user input

        Print the course ID and course title

        If prerequisites exist

            Print each prerequisite

        Otherwise

            Indicate that no prerequisites are required

        End the search

    If the user input is less than the current node’s ID

        Move to the left child

    If the user input is greater

        Move to the right child

If no match is found

    Display a message saying the course could not be located

**Printing All Courses in Alphanumeric Order (BST)**

Start at the root of the tree

Recursively visit the left child

Print the current node’s course ID and title

Recursively visit the right child

Repeat until all nodes have been visited

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Insert | O(1) amortized (end) / O(n) with resizing | O(1) average, O(n) worst (collision chain) | O(log n) average, O(n) worst (unbalanced) |
| Search | O(n) (linear scan) | O(1) average, O(n) worst | O(log n) average, O(n) worst |
| Print All | O(n log n) (sort needed) | |  | | --- | |  |  |  | | --- | | O(n log n) if sorted output required | | O(n) (in-order traversal is naturally sorted) |
| Memory Use | Compact, minimal overhead | Extra overhead for buckets + chaining nodes | More overhead (node pointers for left/right) |

Analysis of Advantages and Disadvantages

The two main responsibilities of advisors include printing courses in alphabetical order and showing one course with its requirements so a BST represents the optimal balanced solution for your code. The BST data structure enables O(n) full sorted list printing without additional sorting operations while maintaining an average O(log n) time complexity for single-course lookup. The text suggests random insertion methods or rebalancing when permitted to reduce worst-case height. The text indicates that random insertion methods help maintain minimum height. A hash table would be more efficient than a BST for #2 when you need many random lookups but occasional full list prints because it would still require O(n log n) time to generate the sorted list.