Southern New Hampshire University

CS 300 Project One Document

// Vector Pseudocode - Milestone 1

On Start:

- Open the file named courses.txt.
- Read all the courses and store them in a list.
- Ask the user to type a course number.
- Look through the list of courses to find the one that matches.
- Show the course name and its prerequisites (or say "None" if there are no prerequisites).

A course has:

- A course number (like CS101)
- A course name (like "Introduction to Programming")
- A list of prerequisite course numbers (may be empty)

Make an empty list to hold all courses.

Make another list to remember all valid course numbers (to check prerequisites later).

Go through the file one line at a time.

- Skip lines that are blank.
- Split the line into parts separated by commas.
- The first part is the course number, the second part is the course name, and anything after that is a prerequisite.
- Create a course record and add it to the list.
- Remember its course number for later checking.

After reading all courses, go back and check:

- For each course, make sure all of its listed prerequisites are also in the file.
- If not, show an error message.

Return the completed list of courses.

- Start with "not found" as the default.
- Go through each course in the list:
    - If the course number matches the one the user typed:
        - Show the course number and name.
            - If it has prerequisites, list them. Otherwise, say "None."
            - Mark it as "found."
            - Stop searching.
- If still "not found" after checking all courses, say that the course doesn't exist.

(alphanumeric course number sort)

Make a copy of the course list.

Sort the copy so course numbers go from smallest to largest (A to Z, numbers in order).

For each course in the sorted list:

- Show the course number and course title on the screen.

// Hash Table Pseudocode - Milestone 2

Main Program:

- Set the file name to "courses.txt"
- Load all course information from the file into a hash table
- Ask the user to enter a course number to search
- Read the user's input
- Look up that course and show its details

Course Structure:

- Each course has:
  - A course number (like CSCI101)
  - A name or title (like "Intro to Programming")
  - A list of other courses it requires before (prerequisites)

How to Load Course Data from File:

- Create an empty hash table to store the courses
- Make a list to keep track of all course numbers (to check for errors later)
- Also keep a list of all the course objects we create

Step 1: Read the file line by line

- If a line is blank, skip it
- Split each line by commas to separate the course info
- If the line doesn’t have at least a course number and a name, show an error and skip it
- The first item is the course number
- The second item is the course name
- Any items after that are prerequisites
- Create a course object with this information
- Add it to the hash table using the course number as the key

- Save the course number to the list of valid numbers
- Save the full course to the list for later checking

Step 2: Make sure all prerequisites exist

- For every course we added:
  - For each of its prerequisites:
    - If the prerequisite isn’t in our list of valid course numbers, show an error
- Return the full hash table of courses

How to Search and Show a Course:

- If the course number the user typed is not in the hash table:
  - Show a message saying it wasn't found
- Otherwise:
  - Find that course in the hash table
  - Show its number and name
  - If it has no prerequisites, say so
  - If it has prerequisites:
    - For each one:
      - If that course exists in the hash table, show its number and name
      - If it doesn’t, mention that it was not found

(alphanumeric course number sort)

Get all the courses from the hash table and put them into a list.

Sort the list so course numbers go from smallest to largest.

For each course in the sorted list:

- Show the course number and course title on the screen.

Southern New Hampshire University

// Binary Search Tree Pseudocode – Milestone 3

```
Each course includes:
    - A course number (like CSCI200)
    - A course name (like "Data Structures")
    - A list of prerequisites (other course numbers)
Each item in the tree holds:
    - One course
    - A link to the course with a smaller number (left)
    - A link to the course with a larger number (right)
We can:
    - Add a new course
    - Search for a course
    - Go through all courses in order
Set the file name to "courses.txt"
Load all the course information into the tree
Ask the user to type in a course number
Show that course's title and its list of prerequisites
Create an empty binary search tree to hold the courses
Create an empty list to keep the full course info
Create a list of all course numbers to help with checking
prerequisites
Open the course file
For each line in the file:
```

- If the line is blank, skip it
- Split the line by commas
- If there are fewer than 2 items, show an error and skip the line
- The first item is the course number
- The second item is the course name
- The remaining items are prerequisite course numbers
- Create a course object with the above info
- Add the course to the list of all courses
- Add the course number to the list of valid numbers
- Insert the course into the tree (based on course number)

After loading all the courses:

- For each course in the list:
  - Check each prerequisite
  - If any prerequisite isn’t in the valid course numbers list, show an error saying it’s missing

Return the completed tree of courses

If the tree is empty, place the course at the root

Otherwise:

- If the new course number is smaller, go left
- If it's larger, go right
- Repeat until a spot is found

Look up the course in the tree using its course number

If the course is not found:

- Show a message that it’s not available

If the course is found:

- Show the course number and name
- If it has no prerequisites:
    - Say "None"
- If it has prerequisites:
    - For each one:
        - Try to look it up in the tree
        - If found, show its course number and name
        - If not found, mention that it's missing

(alphanumeric course number sort)

Start from the smallest course in the tree (leftmost branch.

Visit each course in order from smallest to largest using an in-order walk:

- For each course you visit, show the course number and course title on the screen.

// Course Program Menu Pseudocode

Start the program.

Set a note that says “data is not loaded yet.”

Show the menu options:

1 - Load course data from the file.

2 - Show a list of all Computer Science courses in alphabetical order.

3 - Show the title and prerequisites for one specific course.

9 - Exit the program.

Ask the user to choose an option.

If the user chooses 1:

- Load the course data from the file.
- Update the note to say “data is loaded.”
- Tell the user the data was loaded successfully.

If the user chooses 2:

- If the data is not loaded yet, tell the user to load it first.
- Otherwise, show the Computer Science courses in A–Z order.

If the user chooses 3:

- If the data is not loaded yet, tell the user to load it first.
- Otherwise, ask the user for the course number and show that course’s details.

If the user chooses 9:

- Tell the user the program is closing.
- End the program.

If the user types anything else:

- Tell them it’s not a valid choice.

Repeat steps 3–9 until the user chooses 9.

Southern New Hampshire University

Evaluation and Runtime Analysis

Vector Data Structure

Code	Cost	# Times Executes	Total Cost
copy all courses into memory	1	n	n
sort courses alphabetically	log n	n	n log n
for each course	1	n	n
print course information	1	n	n
		Total Cost	n log n + 2n
		Runtime	O(n log n)

Advantages associated with the vector data structure are that it is simple to use because it is easy to add and store courses in order, good for small datasets since it works well when the number of courses is low, and fast to access by position when the index is known.

Disadvantages associated with the vector data structure are that searches are slow when the list is not sorted, sorting takes time (for example alphabetical ordered sorting takes O(n log n) time), and Insertions in the middle are slow because adding a course in a specific spot requires shifting many elements sometimes.

Hash Table Data Structure

Code	Cost	# Times Executes	Total Cost
copy all course keys from table	1	n	n
sort course keys alphabetically	log n	n	n log n
for each course key	1	n	n
print course information	1	n	n
		Total Cost	n log n + 2n
		Runtime	O(n log n)

Advantages associated with the hash table data structure are that course lookups are quick if the key is known, searching through everything is not needed since it jumps directly to where the course is stored, and it can handle large datasets and the many courses efficiently.

Disadvantages associated with the hash table data structure are that it is not naturally ordered and so manually keys must be collected and sorted to print in alphabetical order, collisions can slow performance when two courses hash to the same location, and there is more complexity in implementation due to the requirement of a hashing function and collision handling.

Binary Search Tree (BST) Data Structure

Code	Cost	# Times Executes	Total Cost
traverse tree in-order	1	n	n
print course information	1	n	n

Total Cost	2n
Runtime	O(n)

Advantages associated with the binary search tree data structure are that courses are kept sorted automatically using in-order traversals that prints them in alphabetical order, efficient searching (seen with the average time of O(log n) to find a course), and a good balance of search and ordered output (Works well for finding and listing courses).

Disadvantages associated with the binary search tree data structure are that unbalanced trees can be slow with the worst case being O(n) if the tree becomes like a linked list, requires pointer management and careful insertion logic and thus increasing the difficulty of programming, and with the average search slower than O(1), slow for single lookups compared to the hash table.

Recommendation

Based on the Big O analysis and the advantages and disadvantages of each data structure, the binary search tree (BST) is the most suitable choice for the advising program because it strikes a balance between search speed and ordered output. The hash table has the fastest average case lookup (O(1)) but it does not maintain any natural order. Because of this, additional work is needed to organize and produce an alphabetical list of the courses. Contrary to this, a vector is simple to implement and works well for small datasets but searching is slow (O(n)) if the vector is not sorted already and inserting data especially in the middle is inefficient. The binary search on the other hand strikes a balance between search speed and ordered output. Its average search time is O(log n), and it automatically maintains the courses in alphanumeric order, making it ideal for the advisor's requirement to quickly display courses in

alphabetical sequence without a separate sorting step. This efficiency in both retrieval and display outweigh the slightly more complex implementation. Due to this, the BST offers the best long-term scalability and meets both performance and functionality needs for ABCU's advising system.