**CS-300 DSA: Analysis and Design**

**Module 7**

**Project One**

**Pseudocode And Runtime Analysis**

**Submitted By:**

HAMNA KHALID

ID: 2902671

[hamna.khalid@snhu.edu](mailto:hamna.khalid@snhu.edu)

**Submitted To:**

PROFESSOR SATHISH GOPALAKRISHNAN

s.gopalakrishnan1@snhu.edu

Southern New Hampshire University

Sunday, October 19, 2025

**Project One**

**ABCU Course Advising Tool**

# **GOAL**

This project consolidates the pseudocode and runtime analyses from previous milestones to create the final design for the **ABCU Course Advising Tool**.

The program is intended to help advisors access, validate, and display course information efficiently.

The project demonstrates the implementation of three core data structures:

* **Vector** (Milestone One)
* **Hash Table** (Milestone Two)
* **Binary Search Tree** (Milestone Three)

Each structure is analyzed in terms of **efficiency, runtime complexity, and scalability**. The final evaluation provides a **recommendation** for the most suitable data structure for the full C++ implementation in Project Two.

# **AUTHOR**

HAMNA KHALID

ID: 2902671

**Department Of Computer Science**

**ABC University**

# **Vector Data Structure**

## **PSEUDOCODE**

Load, Store, and Print Course Data (Vector)

## **GOAL**

Design pseudocode that loads course data from a file, validates format & prerequisites, stores each course as an object in a **vector**, and supports printing a specific course’s info and prerequisites.

## **OVERVIEW**

This pseudocode was first developed in *Milestone One* and defines the foundation of the advising tool. It demonstrates file input validation, object creation, and sequential access using vectors.

## **Introduction**

For **Project One**, I expanded on the original design developed for ABC University’s Computer Science department. The goal is to help academic advisors efficiently access and manage course information, including course numbers, titles, and prerequisites. All data is stored in a CSV file, and this pseudocode demonstrates how to import the file, validate its format, and store the information in a **vector** of course objects.

A **vector** is used because it is dynamic (automatically resizes as courses are added), provides quick access by index, and is simple to traverse sequentially. Each course is represented as an object containing a course number, title, and prerequisites. Once stored in the vector, courses can be easily searched and displayed, enabling advisors to view details and verify prerequisites.

This design forms the **foundation of the ABCU Course Advising Tool**, serving as the first data structure implemented and analyzed for this project. The pseudocode created here will directly guide the full **C++ implementation** in Project Two.

## **Pseudocode**

### **File Input And Validation**

This section describes how the program opens and reads the CSV file containing course data. Since real-world data may include formatting issues, validation ensures that every record is properly structured before being processed.

function loadCoursesFromFile(fileName):

try:

open file with name fileName

catch error:

print "Error: unable to open file."

exit program

create empty Vector<Course> courses

create empty Set courseNumbers // used to check prerequisites later

// -------- First Pass: Collect all course numbers --------

for each line in file:

split line into tokens by comma

if tokens.size < 2 then:

print "Error: Line does not contain both course number and title"

continue

// Note: some courses may not have prerequisites; in that case, only

// course number and title are required.

courseNumber ← tokens[0]

add courseNumber to courseNumbers

reset file pointer to beginning

// -------- Second Pass: Create course objects --------

for each line in file:

split line into tokens by comma

courseNumber ← tokens[0]

courseTitle ← tokens[1]

prerequisites ← remaining tokens after index 1

// Validate prerequisites

for each prereq in prerequisites:

if prereq not in courseNumbers:

print "Warning: prerequisite " + prereq + " does not exist in file"

skip this course

newCourse ← Course(courseNumber, courseTitle, prerequisites)

append newCourse to courses

close file

return courses

* The file is processed in two passes to ensure reliability and accuracy. In the first pass, all course numbers are collected to verify prerequisites later. This guarantees that no prerequisite references a non-existent course. During the second pass, valid course objects are created and appended to the vector. If a prerequisite is missing, the system prints a warning rather than crashing.

### **Course Object**

Before storing any information, the program defines a **Course object**. Each object represents one course record and contains the course number, title, and prerequisites. This modular design keeps data organized and makes later retrieval and display much easier.

struct Course:

courseNumber : string

title : string

prerequisites : Vector<string>

* This simple structure encapsulates each course’s data clearly. The Vector<string> for prerequisites allows flexibility, as some courses may have none while others have several. It also enables straightforward traversal when printing or validating prerequisites.

### **Print All Courses**

Once all courses are successfully loaded, this function allows the advisor to view every course in the system. It prints each course’s details, including its prerequisites, in the order they were read from the file.

function printAllCourses(courses):

if courses is empty:

print "No courses found in the file."

return

for each course in courses:

print "Course Number: " + course.courseNumber

print "Title: " + course.title

if course.prerequisites is not empty:

print "Prerequisites:"

for each prereq in course.prerequisites:

print " " + prereq

else:

print "Prerequisites: None"

* This function provides a clear, formatted list of all available courses. By checking whether the prerequisites list is empty, it ensures a user-friendly display, showing “None” when no prerequisites exist. This helps advisors quickly interpret the course structure.

### **Search For A Specific Course**

Advisors often need to find a single course quickly. This function performs a search through the vector to locate and display a course by its number, including its title and prerequisites.

function searchCourse(courses, searchNumber):

for each course in courses:

if course.courseNumber = searchNumber:

print "Course Number: " + course.courseNumber

print "Title: " + course.title

if course.prerequisites is not empty:

print "Prerequisites:"

for each prereq in course.prerequisites:

print " " + prereq

else:

print "Prerequisites: None"

return

print "Error: Course not found in the data."

* The search function scans through the vector sequentially (linear search). For a relatively small dataset like ABCU’s course list, this is efficient and simple. The function either prints the full course details or a message indicating the course was not found, ensuring clear communication with the user.

## **Flowchart**

The following flowchart illustrates the logical flow of the vector-based system. It visually represents how the program loads and processes course data for ABC University’s advising tool.

When the program begins, it opens the course data file and verifies that it can be accessed. Each line is read, validated, and converted into a Course object, which is then appended to the vector. Once all courses are loaded, advisors can select from the available options: **print all courses** or **search for a specific course**. The diagram ensures that every step of the pseudocode, validation, storage, and display, is clearly mapped for implementation.

A diagram of a course

AI-generated content may be incorrect.

**Figure 1: Flowchart of the course data program**

## **Program Flow Summary**

The vector-based flow ensures clarity and simplicity. The process can be summarized as:

1. **Open the file** and validate access.
2. **Read each line** and ensure the correct number of tokens.
3. **Create course objects** and check prerequisites.
4. **Append valid courses** to the vector.
5. **Provide menu options** to display or search courses.

This structured flow makes the program modular and easy to expand, setting the foundation for later improvements with hash tables and trees.

## **Runtime Analysis**

The runtime analysis below explains the efficiency of each major operation in the vector-based design. Each operation is measured in terms of **Big O** time complexity, assuming n represents the total number of courses and m represents the average number of prerequisites.

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Load Courses | O(n) | Each line of the file is read once and processed. |
| Validate Prerequisites | O(n·m) | Each course’s prerequisites are checked against all valid course numbers. |
| Append to Vector | O(1) Amortized | Vectors grow dynamically, keeping insertion efficient. |
| Print All Courses | O(n) | Each course is printed once sequentially. |
| Search for a Course | O(n) | The vector is scanned linearly until a match is found. |

* Even though both **searching** and **printing** require linear time, this poses no problem for ABC University because the number of Computer Science courses is relatively small (dozens, not thousands). The vector structure, therefore, balances **simplicity**, **speed**, and **memory efficiency**, making it ideal for small datasets and straightforward lookups.

## **Conclusion**

This pseudocode provides a complete vector-based design for managing course data at ABC University. The program efficiently opens the file, validates each line for accuracy, creates structured course objects, and stores them in a vector for easy access. Once the data is loaded, advisors can print all courses or search for specific ones to view their titles and prerequisites.

Developing this pseudocode strengthened my understanding of file handling, object-oriented organization, and runtime efficiency. It also demonstrated the importance of clean data validation and modular function design. While vectors are best suited for smaller datasets due to their linear search behavior, they offer simplicity and clarity—making them an excellent foundation for the ABCU Course Advising Tool.

This section forms the groundwork for later improvements using more advanced data structures like hash tables and binary search trees in subsequent phases of Project One.

# **Hash Table Data Structure**

## **PSEUDOCODE**

Load, Store, and Print Course Data (Hash Table)

## **GOAL**

Design pseudocode that loads course data from a file, validates format and prerequisites, stores each course as an object in a hash table, and supports printing or searching course information efficiently.

## **OVERVIEW**

This pseudocode was originally developed in **Milestone Two** and represents the next stage of the ABCU Course Advising Tool. It improves efficiency by replacing the linear vector design with a hash table, offering constant-time average performance for lookups and insertions.

## **Introduction**

For Project One, this section expands on the ABCU Course Advising Tool by implementing a **hash table data structure**. The objective is to provide academic advisors with rapid access to course information; specifically, course numbers, titles, and prerequisites. All course data is stored in a CSV file. This pseudocode outlines how to load the file, validate its structure, and insert each course object into the hash table efficiently.

A hash table was selected because it provides **average O(1)** time complexity for insertion and search operations, making it far more scalable than a vector when dealing with larger datasets. Each course is represented as an object, and a **hashing function maps** the course number to a bucket index. Collisions are handled with **chaining** through linked lists.  
Once the courses are stored, advisors can retrieve or display them with minimal delay. This section builds upon the earlier vector implementation but shifts focus toward **efficiency, modularity, and scalability** for larger data sets.

## **Pseudocode**

### **File Input And Validation**

This section explains how the system opens the CSV file, validates that each record is formatted correctly, and ensures that every prerequisite corresponds to an existing course. Validation occurs in two passes: first, collecting course numbers, then inserting valid entries into the hash table.

function loadCoursesFromFile(fileName):

try:

open file with name fileName

catch error:

print "Error: Unable to open file."

exit program

create empty HashTable<Course> courses

create empty Set courseNumbers // used to validate prerequisites

// -------- First Pass: Collect all course numbers --------

for each line in file:

if line is empty then:

print "Error: Empty line in file"

continue

split line into tokens by comma

if tokens.size < 2 then:

print "Error: Line missing course number or title"

continue

courseNumber ← tokens[0]

add courseNumber to courseNumbers

reset file pointer to beginning

// -------- Second Pass: Insert courses into hash table --------

for each line in file:

split line into tokens by comma

courseNumber ← tokens[0]

courseTitle ← tokens[1]

prerequisites ← remaining tokens after index 1

// Validate prerequisites

for each prereq in prerequisites:

if prereq not in courseNumbers:

print "Warning: prerequisite " + prereq + " not found in file"

skip this course

newCourse ← Course()

newCourse.courseNumber ← courseNumber

newCourse.title ← courseTitle

newCourse.prerequisites ← prerequisites

courses.Insert(newCourse)

close file

return courses

* The two-pass structure ensures complete validation and prevents dangling prerequisites. By verifying all course numbers first, data integrity is maintained before insertion into the hash table.

### **Course Object**

Before insertion, each course is encapsulated in a **Course object** that stores its identifying data and prerequisites.

struct Course:

courseNumber : string

title : string

prerequisites : List<string>

* This modular object-oriented design keeps the program flexible and readable. Storing prerequisites in a list makes it simple to iterate through them when printing or validating relationships between courses.

### **Hash Table Structure**

The hash table organizes courses into buckets and provides quick access using a hashing function. It supports insertion, searching, and removal with minimal time complexity.

class HashTable:

buckets : array of LinkedList<Course>

tableSize : integer

function hash(key):

return key mod tableSize

function Insert(course):

key ← hash(convert course.courseNumber to integer)

append course to buckets[key]

function Search(courseNumber):

key ← hash(convert courseNumber to integer)

for each course in buckets[key]:

if course.courseNumber = courseNumber:

return course

return empty Course

function PrintAll():

for each bucket in buckets:

for each course in bucket:

print course.courseNumber, course.title, course.prerequisites

function Remove(courseNumber):

key ← hash(convert courseNumber to integer)

find course in buckets[key] and unlink it if found

* The hash table provides fast access to stored courses while maintaining consistency through collision handling with chaining. Each function supports a specific responsibility, ensuring modular, reusable code.

### **Print All Courses**

This function displays all courses and their prerequisites by leveraging the PrintAll() function from the HashTable class.

function printAllCourses(courses):

courses.PrintAll()

* This modular approach avoids duplicating traversal logic. All output formatting remains clear and consistent for end users.

### **Search For A Specific Course**

This function retrieves and displays information about a single course based on its number.

function searchCourse(courses, searchNumber):

course ← courses.Search(searchNumber)

if course is empty:

print "Error: Course not found."

return

print "Course Number: " + course.courseNumber

print "Title: " + course.title

if course.prerequisites is not empty:

print "Prerequisites:"

for each prereq in course.prerequisites:

print " " + prereq

else:

print "Prerequisites: None"

* By separating the logic of searching from the display output, this pseudocode maintains **modular design principles** and clarity. The search operation runs in constant average time, highlighting the hash table’s efficiency advantage over vectors.

### **Remove A Course**

This optional function supports course deletion without exposing the internal linked list structure.

function removeCourse(courses, courseNumber):

courses.Remove(courseNumber)

print "Course " + courseNumber + " removed (if it existed)."

* Removal is handled within the hash table itself, ensuring proper abstraction and minimal external complexity.

## **Program Flow Summary**

Since the hash table design does not require a separate visual flowchart, this summary outlines the main operational steps for clarity:

1. Open and read the input CSV file.
2. Validate that each record has the required fields.
3. Store all course numbers for prerequisite validation.
4. Create course objects and insert them into the hash table.
5. Provide user options to print all courses or search for specific ones.
6. Support removal of courses if needed.

This logical flow mirrors the program’s underlying pseudocode and ensures reliable, modular functionality.

## **Runtime Analysis**

The following table presents the runtime efficiency of key operations within the hash table implementation. Each operation is expressed in **Big O notation**, where *n* represents the total number of courses, and *k* represents the number of elements in a bucket during collision handling.

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Load Courses (n) | O(n) | Each line is read and processed once; prerequisite validation uses constant-time lookups in a set. |
| Insert Course | O(1) average | The hash function computes a bucket index, and chaining enables efficient insertion even when collisions occur. |
| Search Course | O(1) average, O(k) worst | Lookup is near constant since access is based on the hash index. Only heavy collision scenarios cause degradation. |
| Print All Courses | O(n) | Every course is printed exactly once when traversing all buckets. |
| Remove Course | O(1) average | The course is found and unlinked within its bucket in constant time on average. |

* The hash table achieves **near-constant time** performance for most operations, making it ideal for larger datasets compared to the linear structure of vectors. Only in cases of excessive collisions or poor hash distribution would operations approach linear complexity. This efficiency demonstrates why the hash table is better suited for scalability in the ABCU Course Advising Tool.

## **Conclusion**

This pseudocode delivers a complete and efficient hash table design for managing ABCU’s course data. It validates and stores records, provides instant access to course details, and supports searching and removal operations.

Through this design, I learned how **hashing and collision resolution** improve program scalability compared to vectors. This section strengthens the advising tool’s architecture and prepares it for further optimization through **binary search trees** in later stages of Project One.

# **Binary Search Trees**

## **PSEUDOCODE**

Load, Store, and Print Course Data (Binary Search Tree)

## **GOAL**

Design pseudocode that loads course data from a file, validates format and prerequisites, stores each course as an object in a binary search tree (BST), and supports efficient retrieval and ordered display of all courses.

## **OVERVIEW**

This pseudocode was originally developed in **Milestone Three** and represents the final data structure implementation for the ABCU Course Advising Tool before the coding phase. It focuses on **ordered data access**, enabling advisors to view courses in sorted (alphanumeric) order while maintaining efficient search and insertion times.

## **Introduction**

For **Project One**, this section expands the ABC University Course Advising Tool by implementing a **Binary Search Tree (BST)** data structure. The goal is to give advisors structured and efficient access to course information, specifically course numbers, titles, and prerequisites, while maintaining a natural, alphanumeric order.

The binary search tree organizes courses **hierarchically** by their course numbers: smaller numbers are placed on the left branch, and larger numbers on the right. This arrangement enables **efficient searching, insertion, and ordered traversal**, allowing advisors to easily view all courses in ascending order or retrieve details for a specific course.

Each course is represented as an object containing its number, title, and prerequisites. Once stored in the BST, courses can be displayed in sorted order through **in-order traversal**, ensuring that all data remains logically structured and easy to navigate.

This design builds upon earlier Project One sections: **Vector (Milestone One)** and **Hash Table (Milestone Two)**, but introduces **recursive algorithms and hierarchical organization** to achieve both efficient data retrieval and automatic sorting. It serves as the final and most advanced design stage before full C++ implementation in Project Two.

## **Pseudocode**

### **File Input And Validation**

This section outlines how the program opens and processes the CSV file containing ABC University’s course data. Each record is validated to ensure it includes both a course number and title. Additional fields, if present, are treated as prerequisites. The file is read in two passes: the first validates all course numbers, and the second constructs and inserts valid courses into the binary search tree.

function loadCoursesFromFile(fileName):

try:

open file with name fileName

catch error:

print "Error: Unable to open file."

exit program

create empty BinarySearchTree<Course> courses

create empty Set courseNumbers // used to validate prerequisites

// First Pass: collect all course numbers

for each line in file:

split line into tokens by comma

if tokens.size < 2 then:

print "Error: Missing course number or title"

continue

courseNumber ← tokens[0]

add courseNumber to courseNumbers

reset file pointer to beginning

// Second Pass: insert courses into tree

for each line in file:

split line into tokens by comma

courseNumber ← tokens[0]

courseTitle ← tokens[1]

prerequisites ← remaining tokens after index 1

for each prereq in prerequisites:

if prereq not in courseNumbers:

print "Warning: prerequisite " + prereq + " not found"

skip this course

newCourse ← Course(courseNumber, courseTitle, prerequisites)

courses.Insert(newCourse)

close file

return courses

* This pseudocode ensures that course data is read safely, validated for missing or invalid prerequisites, and inserted into the binary search tree in the correct order. By using a two-pass process, the structure maintains both data integrity and logical consistency.

### **Course Object**

Before storing course data, each record is represented as a structured object. This object holds the course number, title, and list of prerequisites, allowing for easy insertion, retrieval, and printing later in the program.

struct Course:

courseNumber : string

title : string

prerequisites : List<string>

* This structure keeps course data modular and organized. The list of prerequisites makes it simple to display course dependencies and manage prerequisite validation during insertion and output.

### **Binary Search Tree Structure**

The Binary Search Tree organizes courses by their alphanumeric course numbers, ensuring that all left descendants contain smaller numbers and all right descendants contain larger numbers. This design supports efficient insertion, searching, and in-order traversal for sorted output.

class BinarySearchTree:

root : Node

function Insert(course):

if root = null:

root ← new Node(course)

else:

addNode(root, course)

function addNode(node, course):

if course.courseNumber < node.course.courseNumber:

if node.left = null:

node.left ← new Node(course)

else:

addNode(node.left, course)

else:

if node.right = null:

node.right ← new Node(course)

else:

addNode(node.right, course)

function Search(courseNumber):

current ← root

while current ≠ null:

if current.course.courseNumber = courseNumber:

return current.course

else if courseNumber < current.course.courseNumber:

current ← current.left

else:

current ← current.right

return empty Course

function InOrderTraversal(node):

if node ≠ null:

InOrderTraversal(node.left)

print node.course.courseNumber, node.course.title, node.course.prerequisites

InOrderTraversal(node.right)

* The BST structure maintains logical order automatically. The in-order traversal ensures courses are displayed in ascending order by course number, which is essential for advisors when reviewing the entire catalog. The recursive design also keeps the implementation simple and modular.

### **Print All Courses**

This function leverages in-order traversal to display all courses in ascending order by their course number. It ensures that every course is printed once in a neatly sorted list.

function printAllCourses(courses):

courses.InOrderTraversal(courses.root)

* By relying on recursive traversal, this function eliminates the need for manual sorting. It visits each course exactly once, maintaining both efficiency and readability

### **Search For A Specific Course**

Advisors may need to look up a single course by its number. This function searches the tree efficiently and prints the selected course’s details and prerequisites if found.

function searchCourse(courses, searchNumber):

course ← courses.Search(searchNumber)

if course is empty:

print "Error: Course not found."

return

print "Course Number: " + course.courseNumber

print "Title: " + course.title

if course.prerequisites is not empty:

print "Prerequisites:"

for each prereq in course.prerequisites:

print " " + prereq

else:

print "Prerequisites: None"

* The BST’s logarithmic search capability allows this function to locate a course quickly, even in large datasets. It ensures instant access to key information, improving advisor efficiency compared to linear data structures.

### **Remove A Course**

When a course needs to be removed, the BST handles this operation while maintaining its hierarchical order. There are three possible scenarios:

1. The node has **no children** → remove directly.
2. The node has **one child** → promote the child.
3. The node has **two children** → replace the node with its in-order successor.

function Remove(node, courseNumber):

if node = null:

return null

if courseNumber < node.course.courseNumber:

node.left ← Remove(node.left, courseNumber)

else if courseNumber > node.course.courseNumber:

node.right ← Remove(node.right, courseNumber)

else:

// Match found

if node.left = null and node.right = null:

delete node

node ← null

else if node.left ≠ null and node.right = null:

temp ← node.left

delete node

node ← temp

else if node.left = null and node.right ≠ null:

temp ← node.right

delete node

node ← temp

else:

temp ← node.right

while temp.left ≠ null:

temp ← temp.left

node.course ← temp.course

node.right ← Remove(node.right, temp.course.courseNumber)

return node

* This design ensures the binary search tree remains valid after deletion. By carefully managing leaf, single-child, and dual-child cases, the structure’s integrity and order are preserved, guaranteeing future insertions and searches remain efficient.

## **Program Flow Summary**

The binary search tree (BST) design introduces hierarchical organization while maintaining clear logic and modular flow. The process can be summarized as:

1. Open and validate the course data file.
2. Read and verify each line for correct formatting.
3. Create course objects and validate all prerequisites.
4. Insert each course into the BST in sorted order by course number.
5. Provide user options to print all courses (in-order), search for a specific course, or remove one if needed.

This structured flow ensures efficient data access, sorted output, and scalability—offering a balanced approach between speed and organization compared to vectors and hash tables.

## **Runtime Analysis**

To understand the efficiency of this design, it is important to analyze how operations scale as the number of courses increases.

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Load Courses (n) | O(n log n) average,  O(n²) worst | Each course is inserted into the binary search tree. A balanced tree maintains log n height, but a skewed tree (sorted input) increases comparisons. |
| Insert Course | O(log n) average,  O(n) worst | Courses are inserted by comparing course numbers and following left or right pointers. |
| Search Course | O(log n) average,  O(n) worst | The tree is traversed from the root until the target course is found or determined missing. |
| Print All Courses | O(n) | In-order traversal visits each node once, producing a sorted list of all courses. |

* The binary search tree offers efficient search and ordered printing under average conditions. While performance may degrade to O(n) in the worst case when the tree becomes unbalanced, it provides a strong balance between structure, readability, and speed compared to vectors and hash tables.

## **Conclusion**

This pseudocode presents a complete design for managing course information at ABC University using a binary search tree. It includes file input and validation, structured course object creation, and organized storage through efficient insertion and traversal methods. Advisors can print all courses in sorted order, search for specific courses, or remove them while maintaining the tree’s integrity.

Compared to the vector and hash table implementations, the binary search tree adds hierarchical structure and naturally supports ordered output. Working on this design strengthened my understanding of recursion, node relationships, and how tree balance influences performance. This final structure completes the foundation for Project One and prepares for full C++ implementation in Project Two.

# **Menu Pseudocode**

This pseudocode defines the main program menu that allows advisors to interact with the course data.  
It ensures that the data is loaded before performing any actions and provides options for displaying or searching courses.

function displayMenu():

courses ← empty

userChoice ← 0

while userChoice ≠ 9:

print "====== ABCU Course Advising Tool ======"

print "1. Load Data Structure"

print "2. Print Course List (Alphanumeric Order)"

print "3. Print Course Information"

print "9. Exit"

print "======================================="

print "Enter your choice: "

userChoice ← read input

if userChoice = 1:

print "Enter file name: "

fileName ← read input

courses ← loadCoursesFromFile(fileName)

print "Courses successfully loaded!"

else if userChoice = 2:

if courses is empty:

print "Error: No data loaded."

else:

printAllCourses(courses)

else if userChoice = 3:

if courses is empty:

print "Error: No data loaded."

else:

print "Enter course number to search: "

searchNumber ← read input

searchCourse(courses, searchNumber)

else if userChoice = 9:

print "Exiting program. Goodbye!"

else:

print "Invalid choice. Please enter a valid option."

* This menu gives advisors an intuitive way to load course data, view all courses, or look up individual ones. It ensures that no operations are attempted before data is loaded, maintaining program reliability and user control.

# **Runtime Comparison And Evaluation**

The three data structures, Vector, Hash Table, and Binary Search Tree, each provide unique advantages and trade-offs in terms of speed, organization, and scalability. The table below summarizes their average time complexities and main characteristics when used for the ABCU Course Advising Tool.

| **Data Structure** | **Average Time Complexity** | **Strengths** | **Weaknesses** |
| --- | --- | --- | --- |
| **Vector** | **Search:** O(n)  **Insert:** O(1) amortized  **Print:** O(n) | Simple to implement, predictable structure, ideal for small datasets | Inefficient for searching large datasets due to linear traversal |
| **Hash Table** | **Search:** O(1) average, O(n) worst  **Insert:** O(1) average  **Print:** O(n) | Speedy lookups and insertions, scalable for large data files | No inherent order; collisions are possible without proper hashing |
| **Binary Search Tree** | **Search:** O(log n) average, O(n) worst  **Insert:** O(log n) average  **Print (In-Order):** O(n) | Maintains sorted order, supports efficient recursive searches | Performance degrades if unbalanced (worst-case O(n)) |

In evaluating performance, the **Vector** performs adequately for small datasets, but its linear search time becomes inefficient as the course catalog grows. The **Binary Search Tree** improves efficiency through hierarchical organization, offering faster searches and automatic sorting, but requires careful balancing to avoid degradation. The **Hash Table** outperforms both for large datasets, offering nearly constant-time lookups and insertions due to its efficient hashing mechanism.

From both runtime and scalability perspectives, the **Hash Table** is the most practical choice for the ABCU Course Advising Tool. Its speed and direct-access lookup make it ideal for quick retrieval of course information, while the Binary Search Tree can be used when ordered data presentation is required.

# **Final Recommendation**

After completing the pseudocode design and runtime analysis for all three data structures, Vector, Hash Table, and Binary Search Tree, I recommend using the **Hash Table** for the final C++ implementation in Project Two.

The Hash Table offers **near-constant time performance** for insertion and searching, making it the most efficient structure for handling a growing number of course records. Its ability to provide instant lookups perfectly supports ABC University’s advising needs, where speed and accuracy are crucial.

While the **Vector** structure provides simplicity and ease of implementation, it becomes inefficient for larger datasets due to its linear search behavior. The **Binary Search Tree** supports sorted traversal and hierarchical organization, but can degrade to linear performance if not balanced. In contrast, the Hash Table maintains consistent efficiency even with large files, making it the most scalable and practical choice.

Overall, the Hash Table strikes the best balance between **speed, reliability, and memory efficiency**, making it the ideal data structure to power the ABCU Course Advising Tool in future development.

# **Final Reflection**

Completing Project One helped me strengthen my understanding of algorithmic design, data structure selection, and runtime analysis. Each milestone; vector, hash table, and binary search tree, taught me how to approach problems from different computational perspectives. I learned the importance of balancing simplicity, speed, and scalability when designing solutions for real-world applications.  
This project also improved my ability to think logically before coding, ensuring that every function and flow is planned out clearly. It prepared me to transition smoothly into Project Two, where I will translate these pseudocode designs into a working C++ implementation for the ABCU Course Advising Tool.

# **Acronyms And Full Forms**

* **ABCU** – ABC University
* **BST** – Binary Search Tree
* **CSV** – Comma-Separated Values
* **DSA** – Data Structures and Algorithms
* **IDE** – Integrated Development Environment
* **I/O** – Input / Output
* **O(1)** – Big O Notation for Constant Time
* **O(n)** – Big O Notation for Linear Time
* **OOP** – Object-Oriented Programming
* **SNHU** – Southern New Hampshire University
* These acronyms are used frequently throughout the pseudocode, runtime analyses, and design documentation. Defining them ensures clarity, consistency, and professionalism in technical communication.

# **References**

* GeeksforGeeks. (n.d.). *Hashing in Data Structures*.

<https://www.geeksforgeeks.org/hashing-data-structure>

* Khalid, H. (2025). *CS-300 Module Four Project One Milestone Two: Hash Table Data Structure*. Southern New Hampshire University.
* Khalid, H. (2025). *CS-300 Module Five Project One Milestone Three: Binary Search Tree Data Structure*. Southern New Hampshire University.
* Khalid, H. (2025). *CS-300 Module Three Project One Milestone One: Vector Data Structure*. Southern New Hampshire University.
* Southern New Hampshire University. (2025). *CS-300 Pseudocode Document*. Unpublished Supporting Material.
* TutorialsPoint. (n.d.). *Binary Search Tree Basics and Operations*. Retrieved from <https://www.tutorialspoint.com>
* Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.
* W3Schools. (n.d.). *C++ File Handling Tutorial*. Retrieved from <https://www.w3schools.com/cpp>
* ZyBooks. (2023). *Data Structures: Vectors, Hash Tables, and Trees*. Southern New Hampshire University.

<https://learn.zybooks.com>