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

**Module 4**

**Project One Milestone Two**

**Hash Table Data Structure**

**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

Saturday, September 27, 2025

**Hash Table Data Structure**

# **PSEUDOCODE**

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

ABCU Course Advising Tool

# **GOAL**

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

# **AUTHOR**

HAMNA KHALID

ID: 2902671

**Department Of Computer Science**

**ABC University**

# **Introduction**

For this milestone, I expanded my design for ABC University’s Computer Science department. The objective is to give advisors fast and reliable access to course information, including course numbers, titles, and prerequisites. All course data is provided in a CSV file. This pseudocode outlines how to import the file, validate its format, and store the courses in a hash table data structure.

A hash table is the chosen structure because it offers average-case **O(1)** insertion and search time, which is much more efficient than linear structures for large datasets. Each course is represented as an object with fields for course number, title, and prerequisites. A hashing function maps each course number to a bucket, and collisions are handled with chaining using linked lists.

Once stored in the hash table, advisors can quickly retrieve course details by searching for a specific course number or by printing all available courses with their prerequisites.

This milestone builds on the earlier vector-based design from Milestone One but shifts the focus toward efficiency and scalability. The pseudocode presented here provides a roadmap for implementing the hash table in C++ for Project One.

# **Pseudocode**

## **File Input and Validation**

The first step is to open the input file, read each line, and verify that it is formatted correctly. Each line must have at least a course number and title, while additional tokens represent prerequisites. To validate prerequisites, the pseudocode first collects all course numbers in one pass and then checks that every prerequisite exists in the dataset during a second pass.

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

* By separating the file reading into two passes, the pseudocode ensures that all prerequisites are validated against existing course numbers. Empty or malformed lines are flagged, and only valid courses are inserted into the hash table. This prevents dangling prerequisites and guarantees data integrity before advisors access the information.

## **Course Object**

The Course object serves as the building block of the data structure. Each course must capture three elements: its course number (unique identifier), its title, and a list of its prerequisites.

struct Course:

courseNumber : string

title : string

prerequisites : List<string>

* This simple structure ensures each course can be stored, validated, and printed consistently. By keeping prerequisites in a list, the program can easily traverse them later when advisors request course details.

## **Hash Table Structure**

The hash table is the central data structure for this milestone. It organizes courses into buckets (an array of linked lists) and uses a hash function to compute bucket indices. This allows efficient insertion, searching, printing, and removal of courses. Collisions are resolved through **chaining**, where multiple courses that hash to the same index are stored in a linked list within that bucket.

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 function ensures each course maps to a specific bucket. Insert handles collisions naturally with chaining. Search allows advisors to quickly look up a course by jumping directly to its bucket. PrintAll provides a complete overview of all courses in the department. This design balances speed and reliability, particularly with larger course catalogs.

## **Print All Courses**

Once the data is loaded, advisors may want to view all courses and their prerequisites. Instead of re-implementing traversal, this function simply calls the PrintAll() method defined in the HashTable class.

function printAllCourses(courses):

courses.PrintAll()

* This approach avoids duplicating logic. The heavy lifting of iterating through buckets and resolving collisions is already handled inside the HashTable. The user-facing function simply provides a clean entry point for displaying all courses.

## **Search for a Specific Course**

Advisors must also be able to find information on a single course by its number quickly. This function uses the hash table’s Search() method and formats the results.

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"

* This version delegates lookup to the HashTable’s Search() method. The user-facing function only handles output formatting. This separation of responsibilities shows an understanding of data structure internals vs. application layer logic.

## **Remove a Course**

To mirror assignment requirements, a remove function can also be added.

function removeCourse(courses, courseNumber):

courses.Remove(courseNumber)

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

* Advisors do not need to know how nodes are unlinked inside the hash table. They simply call Remove(), and the structure takes care of it internally.

# **Runtime Analysis**

To understand the efficiency of this design, it is important to analyze how long operations take as the dataset grows.

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Load Courses (n) | O(n) | Each line parsed once; prerequisite checks are O(1) due to set lookup. |
| Insert Course | O(1) average | The hash function computes bucket index, and chaining allows quick insertion. |
| Search Course | O(1) avg,  O(k) worst | Directs to the bucket; traverses only a small chain unless poor hashing. |
| Print All Courses | O(n) | Every course is printed once. |

**Table 1: Runtime Analysis Of Functions**

* The hash table offers near-constant time performance for search, insert, and remove operations, making it significantly faster than the vector structure in Milestone One for large datasets. Only in worst-case scenarios with poor hashing would searches degrade to linear time.

# **Conclusion**

This pseudocode provides a complete design for using a hash table to manage course information at ABCU. It includes file input and validation, object creation, efficient storage in the hash table, and the ability to print all courses or search for a specific one.

By building this design, I reinforced my knowledge of hash tables, collision handling, and runtime analysis. Compared to linear structures like vectors, hash tables provide superior scalability and efficiency, making them ideal for this application. This pseudocode will guide the upcoming C++ implementation for Project One.

# **References**

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

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

* + - * Southern New Hampshire University. (2025). *CS 300 Pseudocode Document*. Unpublished Supporting Material.
      * Weiss, M. A. (2014). *Data Structures and Algorithm Analysis in C++* (4th ed.). Pearson.

<https://www.pearson.com/en-us/subject-catalog/p/data-structures-and-algorithm-analysis-in-c/P200000003536/9780132847377>

* + - * ZyBooks. (2023). *Data Structures: Hash Tables*. Southern New Hampshire University.

<https://learn.zybooks.com/zybook/CS-300-10324.202581-1>