**CS 300 Project 1 - Runtime Analysis and Data Structure Comparison**

**Student Name**: Rick Goshen  
**Course**: CS 300  
**Date**: June 13, 2025

**Runtime Analysis Chart**

| **Operation** | **Vector** | **Hash Table** | **Binary Search Tree** |
| --- | --- | --- | --- |
| **Load Data Structure** | O(n²) | O(n²) | O(n²) average, O(n²) worst |
| **Print Course List** | O(n log n) | O(n log n) | O(n) |
| **Print Course** | O(n) | O(1) average, O(n) worst | O(log n) average, O(n) worst |
| **Space Complexity** | O(n) | O(n + capacity) | O(n) |

*Note: Loading is O(n²) for all implementations due to prerequisite validation requirements.*

**Advantages and Disadvantages Analysis**

**Vector Implementation**

**Advantages:**

* Simple implementation and easy to understand
* Memory efficient with no pointer overhead
* Good cache performance due to contiguous memory layout
* Predictable performance characteristics
* Easy to debug and maintain

**Disadvantages:**

* O(n) search time becomes inefficient with large datasets
* Requires O(n log n) sorting for alphabetical course display
* Performance degrades linearly as dataset grows
* Poor scalability for frequent search operations

**Hash Table Implementation**

**Advantages:**

* Excellent O(1) average search performance
* Scales well regardless of dataset size
* Efficient for frequent individual course lookups
* Consistent performance with proper load factor management
* Optimal for lookup-intensive applications

**Disadvantages:**

* More complex implementation requiring collision handling
* Memory overhead for hash structure and empty buckets
* Can degrade to O(n) worst case with poor hash distribution
* Requires collecting and sorting data for alphabetical display
* Performance sensitive to hash function quality

**Binary Search Tree Implementation**

**Advantages:**

* Natural alphabetical ordering eliminates sorting overhead
* Good O(log n) average search performance
* Efficient O(n) in-order traversal for displaying course list
* Memory efficient with minimal pointer overhead
* Self-organizing structure

**Disadvantages:**

* Performance heavily depends on tree balance
* Can degrade to O(n) search time with unbalanced tree
* More complex recursive implementation
* Sensitive to data insertion order
* May require balancing mechanisms for guaranteed performance

**Recommendation**

**Hash Table** is the recommended implementation for the ABCU Course Management System.

**Justification:** The primary use case for academic advisors involves frequent individual course lookups during student advising sessions. Hash tables provide O(1) average search performance, which significantly outperforms vector's O(n) linear search and is more reliable than BST's performance that depends on tree balance. While hash tables require additional implementation complexity and memory overhead, the performance benefits justify these trade-offs for real-world academic advising workflows where search operations dominate the usage pattern.

The O(n log n) sorting overhead for course list display is acceptable since this operation occurs less frequently than individual searches, and the scalable performance ensures the system will handle growth from small colleges to large universities effectively.

**CS 300 Project 1 - Hash Table Implementation**

**Course Management System for ABCU Computer Science Department**

**Purpose**: This complete pseudocode demonstrates a hash table-based solution for loading course data from a file, validating the data format, providing efficient search functionality, menu system, and sorted course display.

**Author**: Rick Goshen  
**Date**: June 13, 2025  
**Course**: CS 300  
**Assignment**: Project 1 - Hash Table Implementation

**Data Structure Definitions**

**Course Structure**

// Structure to hold course information

// Contains course number, name, and list of prerequisites

struct Course {

String courseNumber // Unique identifier (e.g., "CSCI100")

String name // Full course name

Vector<String> prerequisites // List of prerequisite course numbers

}

**Hash Table Structures**

// Hash table node structure for chaining collision resolution

struct HashNode {

Course course // Course object containing course data

HashNode\* next // Pointer to next node in chain (for collisions)

}

// Hash table structure with dynamic resizing capability

struct HashTable {

Vector<HashNode\*> buckets // Array of pointers to hash nodes

int size // Current number of courses stored

int capacity // Current number of buckets

double maxLoadFactor // Maximum load factor before resize (0.7)

}

**File Input and Parsing Functions**

**Function: Open and Read File**

// Purpose: Opens a file and reads all valid lines into a vector

// Input: filename - path to the course data file, lines - reference to vector that will store file lines

// Output: true if file was successfully read, false otherwise

function readFileLines(String filename, Vector<String>& lines) {

open file with filename

if file cannot be opened

print "Error: Cannot open file '" + filename + "'"

return false

String line

while not end of file

read line from file

if line is not empty and line is not just whitespace

add line to lines

close file

if lines.size() == 0

print "Error: File is empty or contains no valid data"

return false

return true

}

**Function: Parse Single Line**

// Purpose: Splits a line by commas and cleans up tokens

// Input: line - raw line from file, tokens - reference to vector that will store parsed tokens

// Output: true if line was successfully parsed, false if malformed

function parseLine(String line, Vector<String>& tokens) {

if line is empty or line is just whitespace

return false

split line by comma into tokens

// Clean up tokens: trim whitespace and remove empty tokens

for i = tokens.size() - 1 down to 0

remove leading and trailing whitespace from tokens[i]

if tokens[i] is empty

remove tokens[i] from vector

return tokens.size() > 0

}

**File Validation Functions**

**Function: Validate Line Format**

// Purpose: Ensures each line has minimum required fields

// Input: tokens - parsed line tokens, originalLine - for error reporting

// Output: true if line format is valid, false otherwise

function validateLineFormat(Vector<String> tokens, String originalLine) {

if tokens.size() < 2

print "Error: Line '" + originalLine + "' does not have minimum required parameters"

return false

if tokens[0] is empty

print "Error: Course number cannot be empty"

return false

if tokens[1] is empty

print "Error: Course name cannot be empty"

return false

return true

}

**Function: Check All Prerequisites Exist**

// Purpose: Validates that all prerequisites exist as courses in the file

// Input: allLines - all lines from the file

// Output: true if all prerequisites are valid, false otherwise

function validatePrerequisites(Vector<String> allLines) {

for each line in allLines

Vector<String> tokens

if not parseLine(line, tokens)

continue // Skip malformed lines

// Check prerequisites (tokens 2 and beyond)

if tokens.size() > 2 // has prerequisites

for i = 2 to tokens.size() - 1

prerequisite = tokens[i]

if prerequisite is not empty and not courseExists(prerequisite, allLines)

print "Error: Prerequisite '" + prerequisite + "' in course '" + tokens[0] + "' does not exist as a course"

return false

return true

}

**Function: Check if Course Exists**

// Purpose: Searches for a course number in the file data

// Input: courseNumber - course to search for, allLines - all file lines

// Output: true if course exists, false otherwise

function courseExists(String courseNumber, Vector<String> allLines) {

if courseNumber is empty

return false

for each line in allLines

Vector<String> tokens

if parseLine(line, tokens) and tokens.size() >= 1

if tokens[0] equals courseNumber

return true

return false

}

**Function: Validate Entire File**

// Purpose: Orchestrates all file validation steps

// Input: lines - all lines read from file

// Output: true if entire file is valid, false if any validation fails

function validateFile(Vector<String> lines) {

if lines.size() == 0

print "Error: No valid lines found in file"

return false

// Step 1: Check that all lines can be parsed and have valid format

for each line in lines

Vector<String> tokens

if not parseLine(line, tokens)

print "Error: Unable to parse line '" + line + "'"

return false

if not validateLineFormat(tokens, line)

return false

// Step 2: Check that all prerequisites exist as courses

if not validatePrerequisites(lines)

return false

return true

}

**Course Object Creation Functions**

**Function: Create Single Course Object**

// Purpose: Creates a Course object from a validated line of data

// Input: line - validated line from file, course - reference to Course object to populate

// Output: true if course was created successfully, false otherwise

function createCourseObject(String line, Course& course) {

Vector<String> tokens

if not parseLine(line, tokens) or tokens.size() < 2

return false // Invalid line format

// Set required fields

course.courseNumber = tokens[0]

course.name = tokens[1]

course.prerequisites.clear()

// Add prerequisites (tokens 2 and beyond)

for i = 2 to tokens.size() - 1

if tokens[i] is not empty

course.prerequisites.add(tokens[i])

return true

}

**Hash Table Core Functions**

**Function: Initialize Hash Table**

// Purpose: Creates and initializes a new hash table with specified capacity

// Input: initialCapacity - starting number of buckets

// Output: Initialized HashTable structure

function initializeHashTable(int initialCapacity = 16) -> HashTable {

HashTable table

table.capacity = initialCapacity

table.size = 0

table.maxLoadFactor = 0.7

// Initialize all buckets to null

table.buckets.resize(initialCapacity)

for i = 0 to initialCapacity - 1

table.buckets[i] = null

return table

}

**Function: Hash Function**

// Purpose: Converts course number string to hash index using polynomial rolling hash

// Input: courseNumber - string to hash, capacity - table size for modulo

// Output: Hash index (0 to capacity-1)

function hashFunction(String courseNumber, int capacity) -> int {

if courseNumber is empty

return 0

int hash = 0

int prime = 31 // Common prime for polynomial hash

for i = 0 to courseNumber.length() - 1

char c = courseNumber[i]

hash = hash \* prime + ascii\_value(c)

// Ensure positive result and fit within table capacity

return abs(hash) % capacity

}

**Function: Calculate Load Factor**

// Purpose: Determines current load factor for resize decisions

// Input: table - hash table to analyze

// Output: Current load factor (size/capacity ratio)

function getLoadFactor(HashTable table) -> double {

if table.capacity == 0

return 0.0

return (double)table.size / (double)table.capacity

}

**Dynamic Resizing Functions**

**Function: Resize Hash Table**

// Purpose: Doubles table capacity and rehashes all existing courses

// Input: table - reference to hash table to resize

// Output: Updates table with new capacity and redistributed courses

function resizeHashTable(HashTable& table) {

// Store old buckets for rehashing

Vector<HashNode\*> oldBuckets = table.buckets

int oldCapacity = table.capacity

// Double the capacity

table.capacity = table.capacity \* 2

table.size = 0 // Will be recounted during rehashing

// Create new empty bucket array

table.buckets.clear()

table.buckets.resize(table.capacity)

for i = 0 to table.capacity - 1

table.buckets[i] = null

// Rehash all existing courses into new table

for i = 0 to oldCapacity - 1

HashNode\* current = oldBuckets[i]

while current is not null

HashNode\* next = current.next // Save next before reinsertion

// Reinsert course into resized table

insertCourseIntoTable(table, current.course)

// Clean up old node

delete current

current = next

print "Hash table resized from " + oldCapacity + " to " + table.capacity + " buckets"

}

**Function: Check and Resize if Needed**

// Purpose: Monitors load factor and triggers resize when necessary

// Input: table - reference to hash table to check

// Output: Resizes table if load factor > 0.7

function checkAndResize(HashTable& table) {

if getLoadFactor(table) > table.maxLoadFactor

resizeHashTable(table)

}

**Course Insertion Functions**

**Function: Insert Course into Hash Table**

// Purpose: Adds a course to the hash table using chaining for collision resolution

// Input: table - reference to hash table, course - Course object to insert

// Output: Updates table with new course, handles collisions via chaining

function insertCourseIntoTable(HashTable& table, Course course) {

// Check if resize needed before insertion

checkAndResize(table)

// Calculate hash index for course

int index = hashFunction(course.courseNumber, table.capacity)

// Create new node for the course

HashNode\* newNode = new HashNode()

newNode.course = course

newNode.next = null

// Handle collision using chaining

if table.buckets[index] is null

// No collision - first course in this bucket

table.buckets[index] = newNode

else

// Collision detected - check for duplicate course numbers

HashNode\* current = table.buckets[index]

while current is not null

if current.course.courseNumber equals course.courseNumber

// Update existing course instead of creating duplicate

current.course = course

delete newNode // Clean up unused node

return

current = current.next

// No duplicate found - add to front of chain

newNode.next = table.buckets[index]

table.buckets[index] = newNode

table.size = table.size + 1

}

**Search Functions**

**Function: Search Course in Hash Table**

// Purpose: Efficiently finds a course using hash table lookup

// Input: table - hash table to search, courseNumber - course to find,

// foundCourse - reference to Course object to populate if found

// Output: true if course found, false otherwise

// Complexity: Average O(1), worst case O(n) if many collisions

function searchCourse(HashTable table, String courseNumber, Course& foundCourse) {

if courseNumber is empty

return false

// Calculate hash index

int index = hashFunction(courseNumber, table.capacity)

// Search through chain at this index

HashNode\* current = table.buckets[index]

while current is not null

if current.course.courseNumber equals courseNumber

foundCourse = current.course

return true

current = current.next

return false // Course not found

}

**Display Functions**

**Function: Print Single Course Info**

// Purpose: Displays basic course information in required format

// Input: course - Course object to display

// Output: Prints "courseNumber, courseName" to console

function printCourseInfo(Course course) {

print course.courseNumber + ", " + course.name

}

**Function: Print Prerequisites**

// Purpose: Displays all prerequisites for a course using hash table lookups

// Input: course - Course object whose prerequisites to display,

// table - hash table for prerequisite lookups

// Output: Prints prerequisite course information or "No prerequisites required"

function printPrerequisites(Course course, HashTable table) {

if course.prerequisites.size() == 0

print "No prerequisites required"

return

print "Prerequisites:"

for each prerequisite in course.prerequisites

Course prereqCourse

if searchCourse(table, prerequisite, prereqCourse)

print " " + prereqCourse.courseNumber + ", " + prereqCourse.name

else

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

}

**Sorting and Collection Functions**

**Function: Collect All Courses from Hash Table**

// Purpose: Traverses hash table and collects all courses into a vector for sorting

// Input: table - hash table containing courses

// Output: Vector containing all courses from the hash table

function collectAllCourses(HashTable table) -> Vector<Course> {

Vector<Course> allCourses

for i = 0 to table.capacity - 1

HashNode\* current = table.buckets[i]

while current is not null

allCourses.add(current.course)

current = current.next

return allCourses

}

**Function: Sort Courses Alphanumerically**

// Purpose: Sorts a vector of courses alphanumerically by course number

// Input: courses - vector of Course objects to sort

// Output: courses vector is sorted in place by courseNumber

function sortCoursesAlphanumerically(Vector<Course>& courses) {

if courses.size() <= 1

return

mergeSort(courses, 0, courses.size() - 1)

}

**Function: Merge Sort Implementation**

// Purpose: Recursive merge sort algorithm for course sorting

// Input: courses - vector to sort, left - start index, right - end index

// Output: courses[left...right] is sorted by courseNumber

function mergeSort(Vector<Course>& courses, int left, int right) {

if left < right

int middle = (left + right) / 2

mergeSort(courses, left, middle)

mergeSort(courses, middle + 1, right)

merge(courses, left, middle, right)

}

**Function: Merge Function**

// Purpose: Merges two sorted subarrays into one sorted array

// Input: courses - vector containing subarrays, left - start of first subarray,

// middle - end of first subarray, right - end of second subarray

// Output: courses[left...right] contains merged sorted elements

function merge(Vector<Course>& courses, int left, int middle, int right) {

Vector<Course> leftArray

Vector<Course> rightArray

// Copy data to temporary arrays

for i = left to middle

leftArray.add(courses[i])

for j = middle + 1 to right

rightArray.add(courses[j])

// Merge the temporary arrays back

int i = 0, j = 0, k = left

while i < leftArray.size() and j < rightArray.size()

if leftArray[i].courseNumber <= rightArray[j].courseNumber

courses[k] = leftArray[i]

i = i + 1

else

courses[k] = rightArray[j]

j = j + 1

k = k + 1

// Copy remaining elements

while i < leftArray.size()

courses[k] = leftArray[i]

i = i + 1

k = k + 1

while j < rightArray.size()

courses[k] = rightArray[j]

j = j + 1

k = k + 1

}

**Function: Print All Courses Sorted**

// Purpose: Displays all courses from hash table in alphanumeric order with pagination

// Input: table - hash table containing courses

// Output: Displays all courses sorted by course number

function printAllCoursesSorted(HashTable table) {

if table.size == 0

print "No courses loaded. Please load data first."

return

// Collect all courses from hash table

Vector<Course> allCourses = collectAllCourses(table)

// Sort the collected courses

sortCoursesAlphanumerically(allCourses)

print "Course List (Alphanumeric Order):"

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

print "Load Factor: " + getLoadFactor(table)

print "Total Courses: " + table.size

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

int pageSize = 20

int currentPage = 1

int totalPages = (allCourses.size() + pageSize - 1) / pageSize

for i = 0 to allCourses.size() - 1

// Print page header

if i % pageSize == 0

if i > 0

print "\nPress Enter to continue to page " + (currentPage + 1) + " of " + totalPages + "..."

input dummy

print "\n--- Page " + currentPage + " of " + totalPages + " ---"

currentPage = currentPage + 1

printCourseInfo(allCourses[i])

print "\nTotal courses displayed: " + allCourses.size()

}

**Main Loading Function**

**Function: Load Courses From File into Hash Table**

// Purpose: Coordinates file loading, validation, and hash table population

// Input: filename - path to course data file, table - reference to hash table

// Output: true if courses loaded successfully, false if any step fails

function loadCoursesFromFileHashTable(String filename, HashTable& table) {

Vector<String> lines

// Step 1: Read file contents

if not readFileLines(filename, lines)

return false

// Step 2: Validate file format and prerequisites

if not validateFile(lines)

return false

// Step 3: Create course objects and insert into hash table

for each line in lines

Course newCourse

if createCourseObject(line, newCourse)

insertCourseIntoTable(table, newCourse)

else

print "Warning: Skipping invalid line during course creation"

return true

}

**Menu System**

**Function: Display Menu**

// Purpose: Displays the main menu options to the user

// Output: Prints menu options to console

function displayMenu() {

print "\n=========================="

print "ABCU Course Management System"

print "(Hash Table Implementation)"

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

print "1. Load Data Structure"

print "2. Print Course List"

print "3. Print Course"

print "9. Exit"

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

print "What would you like to do? "

}

**Function: Get Menu Choice**

// Purpose: Gets and validates user menu choice with error handling

// Input: choice - reference to store user's choice

// Output: true if valid choice entered, false if invalid

function getMenuChoice(String& choice) {

input choice

choice = trim(toUpperCase(choice))

if choice equals "1" or choice equals "2" or choice equals "3" or choice equals "9"

return true

else

print "Invalid choice. Please enter 1, 2, 3, or 9."

return false

}

**Function: Menu Option 1 - Load Data Structure**

// Purpose: Handles loading course data into hash table with comprehensive error handling

// Input: table - reference to hash table to populate

// Output: Hash table is populated with validated course data

function menuOption1(HashTable& table) {

String filename

print "Enter filename: "

input filename

if filename is empty

print "Error: Filename cannot be empty"

return

// Clear existing hash table

cleanupHashTable(table)

table = initializeHashTable(16)

// Attempt to load courses

if loadCoursesFromFileHashTable(filename, table)

if table.size == 0

print "Warning: No courses were loaded from the file"

return

print "Courses loaded successfully into hash table!"

print "Number of courses loaded: " + table.size

print "Hash table capacity: " + table.capacity

print "Current load factor: " + getLoadFactor(table)

// Display hash table statistics

displayHashTableStats(table)

else

print "Failed to load courses from file. Please check the file format and try again."

}

**Function: Menu Option 2 - Print Course List**

// Purpose: Prints all courses in alphanumeric order with error handling

// Input: table - hash table containing courses

// Output: All courses displayed in alphanumeric order by course number

function menuOption2(HashTable table) {

if table.size == 0

print "No courses loaded. Please load data first using option 1."

return

printAllCoursesSorted(table)

}

**Function: Menu Option 3 - Print Course**

// Purpose: Searches for and displays specific course information with prerequisites

// Input: table - hash table containing courses

// Output: Displays course information and prerequisites, or error message

function menuOption3(HashTable table) {

if table.size == 0

print "No courses loaded. Please load data first using option 1."

return

String courseNumber

print "What course do you want to know about? "

input courseNumber

if courseNumber is empty

print "Error: Course number cannot be empty"

return

courseNumber = trim(toUpperCase(courseNumber))

Course foundCourse

if searchCourse(table, courseNumber, foundCourse)

print "\nCourse Information:"

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

printCourseInfo(foundCourse)

print ""

printPrerequisites(foundCourse, table)

else

print "Course '" + courseNumber + "' not found."

print "Please check the course number and try again."

}

**Hash Table Statistics and Management**

**Function: Display Hash Table Statistics**

// Purpose: Provides detailed statistics about hash table performance

// Input: table - hash table to analyze

// Output: Collision statistics, distribution analysis, performance metrics

function displayHashTableStats(HashTable table) {

print "\n=== Hash Table Performance Statistics ==="

print "Total courses: " + table.size

print "Table capacity: " + table.capacity

print "Load factor: " + getLoadFactor(table)

// Analyze collision distribution

int emptyBuckets = 0

int maxChainLength = 0

int totalCollisions = 0

for i = 0 to table.capacity - 1

int chainLength = 0

HashNode\* current = table.buckets[i]

if current is null

emptyBuckets = emptyBuckets + 1

else

while current is not null

chainLength = chainLength + 1

current = current.next

if chainLength > 1

totalCollisions = totalCollisions + (chainLength - 1)

if chainLength > maxChainLength

maxChainLength = chainLength

print "Empty buckets: " + emptyBuckets + " (" + (emptyBuckets \* 100 / table.capacity) + "%)"

print "Maximum chain length: " + maxChainLength

print "Total collisions: " + totalCollisions

if table.size > 0

print "Average search operations: " + ((double)(table.size + totalCollisions) / table.size)

}

**Function: Clean Up Hash Table**

// Purpose: Deallocates all memory used by hash table

// Input: table - reference to hash table to clean up

// Output: Frees all nodes and resets table to empty state

function cleanupHashTable(HashTable& table) {

for i = 0 to table.capacity - 1

HashNode\* current = table.buckets[i]

while current is not null

HashNode\* next = current.next

delete current

current = next

table.buckets[i] = null

table.size = 0

print "Hash table memory cleaned up successfully"

}

**Main Program**

**Function: Main Program Entry Point**

// Purpose: Main program entry point with hash table implementation

// Process: 1) Initialize hash table, 2) Handle user interaction, 3) Clean up memory

function main() {

HashTable courseTable = initializeHashTable(16)

String choice

boolean running = true

print "Welcome to the ABCU Course Management System (Hash Table Implementation)"

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

while running

displayMenu()

if getMenuChoice(choice)

switch choice

case "1":

menuOption1(courseTable)

break

case "2":

menuOption2(courseTable)

break

case "3":

menuOption3(courseTable)

break

case "9":

print "\nCleaning up memory..."

cleanupHashTable(courseTable)

print "Thank you for using the ABCU Course Management System!"

print "Goodbye!"

running = false

break

default:

print "Unexpected error in menu selection."

break

else

continue

return 0

}

**Runtime Analysis for Hash Table Implementation**

**Menu Option 1: Load Data Structure**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| Read each line from file | 1 | n | n |
| Parse each line | 1 | n | n |
| Validate line format | 1 | n | n |
| Check prerequisites exist | n | n | n² |
| Create course object | 1 | n | n |
| Hash function calculation | 1 | n | n |
| Insert into hash table (avg) | 1 | n | n |
| Resize operations (amortized) | 1 | log n | log n |
| **Total Cost** |  |  | **n² + 5n + log n + 1** |
| **Runtime** |  |  | **O(n²)** |

**Menu Option 2: Print Course List**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Traverse all hash buckets | 1 | capacity | capacity |
| Collect all courses into vector | 1 | n | n |
| Merge sort collected courses | n log n | 1 | n log n |
| Print each course | 1 | n | n |
| Pagination logic | 1 | n | n |
| **Total Cost** |  |  | **n log n + capacity + 3n** |
| **Runtime** |  |  | **O(n log n)** |

**Menu Option 3: Print Course**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Hash function calculation | 1 | 1 | 1 |
| Search through chain (avg) | 1 | 1 | 1 |
| Search through chain (worst) | 1 | n | n |
| Compare course numbers | 1 | 1 (avg), n (worst) | 1 to n |
| Print course info | 1 | 1 | 1 |
| Search for each prerequisite | 1 | p | p |
| Print prerequisites | 1 | p | p |
| **Total Cost** |  |  | **2p + 4 (avg), 2p + n + 3 (worst)** |
| **Runtime** |  |  | **O(1) average, O(n) worst case** |

**Hash Table Specific Operations**

| **Operation** | **Average Case** | **Worst Case** | **Space** |
| --- | --- | --- | --- |
| **Hash Function** | O(k) where k = key length | O(k) | O(1) |
| **Insert** | O(1) | O(n) | O(1) |
| **Search** | O(1) | O(n) | O(1) |
| **Resize** | O(n) amortized | O(n) | O(n) |
| **Delete** | O(1) | O(n) | O(1) |

**Overall Analysis Summary**

**Best Case Performance:**

* Load Data: O(n²) - due to prerequisite validation, not hash operations
* Print Course List: O(n log n) - dominated by sorting
* Search Course: O(1) - ideal hash table performance

**Average Case Performance:**

* Load Data: O(n²) - prerequisite validation dominates
* Print Course List: O(n log n) - sorting collected courses
* Search Course: O(1) - hash table strength

**Worst Case Performance:**

* Load Data: O(n²) - when all courses hash to same bucket during validation
* Print Course List: O(n log n) - merge sort is consistent
* Search Course: O(n) - when all courses hash to same bucket

**Space Complexity:**

* O(n + capacity) - courses plus hash table overhead
* Additional O(n) during sorting for temporary arrays
* Load factor maintained at 0.7 keeps capacity reasonable

**Advantages of Hash Table Implementation:**

* Excellent average-case search performance: O(1)
* Efficient insertion: O(1) average case
* Handles large datasets well due to constant-time operations
* Dynamic resizing maintains performance as data grows
* Good performance for frequent course lookups

**Disadvantages of Hash Table Implementation:**

* Worst-case search performance: O(n) due to collisions
* Requires collecting and sorting data for alphabetical display
* More complex implementation with collision handling and resizing
* Memory overhead for pointers, empty buckets, and hash structure
* Performance depends on hash function quality and data distribution