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

**Module 5**

**Project One Milestone Three**

**Binary Search Tree 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

Wednesday, October 1, 2025

**Binary Search Tree Data Structure**

# **PSEUDOCODE**

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

ABCU Course Advising Tool

# **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, 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 the design for ABC University’s Computer Science department by implementing the tree version of the course advising tool. The purpose is to give advisors structured access to course information, including course numbers, titles, and prerequisites, while also supporting ordered traversal.

A binary search tree is the chosen structure because it organizes data hierarchically by course number, placing smaller numbers on the left and larger numbers on the right. This provides efficient search and ordered output compared to vectors and hash tables. Each course is represented as an object with fields for the course number, title, and prerequisites.

Once stored in the binary search tree, advisors can quickly retrieve course details, print all courses in sorted order, and ensure prerequisites are properly linked. This milestone builds on Milestone One (vector) and Milestone Two (hash table) but emphasizes recursive algorithms and hierarchical structure.

# **Pseudocode**

## **File Input and Validation**

The program must first open the course file, read each line, and split the data into tokens. Each line must contain at least two parameters: the course number and the course title. Additional tokens, if present, are prerequisites. This process also validates the format by ensuring prerequisites exist elsewhere in the file.

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 the file is processed safely, prerequisites are validated, and every course is inserted into the binary search tree in the correct place. By checking format errors, it prevents invalid data from entering the tree.

## **Course Object**

Each line from the file is stored in a course object. The course object holds the course number, the course title, and a list of prerequisites.

struct Course:

courseNumber : string

title : string

prerequisites : List<string>

* This structure ensures that each course is self-contained, making it easier to store, search, and display. The prerequisites list allows advisors to track and display dependencies between courses.

## **Binary Search Tree Structure**

The binary search tree stores all courses and allows efficient searching and ordered output. Insertion compares course numbers and places each course in its correct position in the tree. Searching follows the same logic, moving left or right until a match is found or the search ends.

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)

* This structure maintains the binary search tree property: smaller course numbers on the left, larger on the right. InOrderTraversal guarantees courses are printed in sorted order, which is useful for advisors to view the catalog alphabetically.

## **Print All Courses**

To display all courses, the program uses InOrderTraversal, which naturally prints them in ascending order by course number.

function printAllCourses(courses):

courses.InOrderTraversal(courses.root)

* This traversal visits each course exactly once, ensuring that all data is shown in the correct order with minimal overhead.

## **Search for a Specific Course**

Advisors often need to look up a single course by its number. This function searches the tree and prints the course’s details, including prerequisites.

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 search provides instant access to course details. Because the binary search tree reduces the number of comparisons, searching is much faster than scanning all courses in a list.

## **Remove a Course**

Removing a course requires adjusting the binary search tree while keeping its order property intact. There are three cases to handle:

1. The node is a **leaf** (no children) → it can be deleted directly.
2. The node has **one child** → the child replaces the node.
3. The node has **two children** → the node is replaced with its in-order successor (the smallest course in its right subtree).

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 pseudocode ensures that when a course is removed, the binary search tree remains valid and correctly ordered. Handling the three cases (leaf, one child, two children) guarantees that no data is lost and that future searches and traversals work as expected.

# **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 log n) average,  O(n²) worst | Each course is inserted into the binary search tree. Balanced tree = log n height, but skewed input may degrade. |
| Insert Course | O(log n) average,  O(n) worst | Navigates down the binary search tree by comparisons. |
| Search Course | O(log n) average,  O(n) worst | Follows binary search tree path left or right until match found. |
| Print All Courses | O(n) | InOrder traversal visits each course exactly once. |

**Table 1: Runtime Analysis Of Functions**

* The binary search tree is efficient for searching and printing, though worst-case performance can degrade if the tree becomes unbalanced.

# **Conclusion**

This pseudocode provides a complete design for using a binary search tree to manage course information at ABCU. It includes file input and validation, course object creation, insertion into the binary search tree, and the ability to print all courses or search for a specific one.

Compared to Milestone One (vector) and Milestone Two (hash table), the binary search tree offers ordered traversal and structured searching. Working on this milestone gave me hands-on practice with recursive logic, hierarchical data organization, and understanding how balance affects runtime. This design prepares me to implement the full C++ version of Project One.

# **References**

* + - * GeeksforGeeks. (n.d.). *Binary Search Tree Data Structure*.

<https://www.geeksforgeeks.org/binary-search-tree-data-structure>

* + - * 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: Binary Search Trees*. Southern New Hampshire University.

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

* + - * Southern New Hampshire University. (2025). *CS 300 Pseudocode Document*. Unpublished Supporting Material.
* SNHU CS-300. (2023). *Course Information Document*. Southern New Hampshire University.