Vector – Milestone 1

## 1. Open and Validate Input File

procedure loadCoursesFromFile(filename):  
  
 open file with name = filename  
 if file cannot be opened:  
 print "Error: Cannot open file"  
 exit program  
  
 initialize empty Vector<Course> courses  
  
 for each line in file:  
 tokens = split(line, ",")  
   
 // Validation: at least courseNumber and courseName  
 if length(tokens) < 2:  
 print "Error: invalid format on line"  
 continue  
  
 courseNumber = tokens[0]  
 courseName = tokens[1]  
 prereqs = tokens[2..end] // may be empty  
  
 create new Course object c  
 set c.number = courseNumber  
 set c.name = courseName  
 set c.prerequisites = prereqs  
  
 add c to courses  
 end for  
  
 // Validation: verify prerequisites exist in file  
 for each course in courses:  
 for each prereq in course.prerequisites:  
 if prereq not found in any course.number in courses:  
 print "Error: prerequisite " + prereq + " not found in file"  
 end for  
  
 close file  
 return courses

## 2. Search and Print Course Information

procedure searchCourse(courses, courseNumber):  
  
 for each course in courses:  
 if course.number == courseNumber:  
 print "Course Number: " + course.number  
 print "Course Name: " + course.name  
  
 if course.prerequisites is empty:  
 print "Prerequisites: None"  
 else:  
 print "Prerequisites: "  
 for each prereq in course.prerequisites:  
 print prereq  
 return  
 end for  
  
 print "Course not found."

HashTable - Milestone 2

## 1. Open and Validate Input File

procedure loadCoursesFromFile(filename):  
  
 open file with name = filename  
 if file cannot be opened:  
 print "Error: Cannot open file"  
 exit program  
  
 initialize HashTable<Course> courses with size N  
  
 for each line in file:  
 tokens = split(line, ",")  
   
 // Validation: at least courseNumber and courseName  
 if length(tokens) < 2:  
 print "Error: invalid format on line"  
 continue  
  
 courseNumber = tokens[0]  
 courseName = tokens[1]  
 prereqs = tokens[2..end]   
  
 create new Course object c  
 set c.number = courseNumber  
 set c.name = courseName  
 set c.prerequisites = prereqs  
  
 // Insert into hash table  
 insertCourse(courses, c)  
 end for  
  
 // Validation: verify prerequisites exist in table  
 for each bucket in courses:  
 for each course in bucket (linked list):  
 for each prereq in course.prerequisites:  
 if not findCourse(courses, prereq):  
 print "Error: prerequisite " + prereq + " not found in file"  
  
 close file  
 return courses

## 2. Course Object and Supporting Structures

// Course object  
structure Course:  
 string number  
 string name  
 Vector<string> prerequisites  
  
// Node for chaining  
structure Node:  
 Course data  
 Node next  
  
// Hash Table  
structure HashTable:  
 array of Node lists (size N)

## 3. Hash and Insert Functions

// Simple modulo-based hash  
function hash(courseNumber, N):  
 numericPart = extract digits from courseNumber  
 index = numericPart mod N  
 return index  
  
// Insert with chaining  
procedure insertCourse(courses, course):  
 index = hash(course.number, N)  
 bucket = courses[index]  
  
 newNode = Node(course)  
 if bucket.head is null:  
 bucket.head = newNode  
 else:  
 append newNode to end of bucket list

## 4. Search and Print Course Information

// Search by course number  
function findCourse(courses, courseNumber):  
 index = hash(courseNumber, N)  
 bucket = courses[index]  
  
 current = bucket.head  
 while current != null:  
 if current.data.number == courseNumber:  
 return current.data  
 current = current.next  
 return null  
  
// Print details  
procedure printCourseInfo(courses, courseNumber):  
 course = findCourse(courses, courseNumber)  
 if course == null:  
 print "Course not found."  
 return  
  
 print "Course Number: " + course.number  
 print "Course Name: " + course.name  
  
 if course.prerequisites is empty:  
 print "Prerequisites: None"  
 else:  
 print "Prerequisites: "  
 for each prereq in course.prerequisites:  
 print prereq

Binary Search Tree – Milestone 3

## 1. Data Structures

structure Course:  
 string number  
 string title  
 Vector<string> prerequisites  
  
structure BSTNode:  
 Course data  
 BSTNode left  
 BSTNode right  
  
structure BST:  
 BSTNode root  
  
 function insert(node, course) -> BSTNode:  
 if node == null:  
 return new BSTNode(data = course, left = null, right = null)  
 if course.number < node.data.number:  
 node.left = insert(node.left, course)  
 else if course.number > node.data.number:  
 node.right = insert(node.right, course)  
 else:  
 // duplicate key; update title & prerequisites  
 node.data.title = course.title  
 node.data.prerequisites = course.prerequisites  
 return node  
  
 function add(course):  
 root = insert(root, course)  
  
 function search(node, courseNumber) -> Course or null:  
 if node == null:  
 return null  
 if courseNumber == node.data.number:  
 return node.data  
 if courseNumber < node.data.number:  
 return search(node.left, courseNumber)  
 else:  
 return search(node.right, courseNumber)  
  
 function inorder(node, action):  
 if node == null: return  
 inorder(node.left, action)  
 action(node.data)  
 inorder(node.right, action)

## 2. Load Courses

function loadCoursesFromFile(filename   
 create empty BST tree  
 open filename for reading  
 if file not opened  
 print "Error opening file."  
 return tree  
  
 set lineNumber = 0  
  
 for each line in file  
 lineNumber = lineNumber + 1  
 line = trim(line)  
  
 if line == "" // skip blank lines  
 continue  
  
 parts = split(line, ",")  
  
 // must have at least course number and title  
 if count(parts) < 2  
 print "Format error on line " + toString(lineNumber)  
 continue  
  
 number = trim(parts[0])  
 title = trim(parts[1])  
  
 // collect any prerequisites after the first two fields  
 prereqs = empty list  
 for i from 2 to count(parts)-1  
 p = trim(parts[i])  
 if p != ""  
 add p to prereqs  
  
 // create Course and insert into BST (keyed by number)  
 c = new Course  
 c.number = number  
 c.title = title  
 c.prerequisites = prereqs  
 tree.add(c)  
 end for  
  
 close file  
  
 // prerequisite existence check  
 for each course in tree (in-order traversal)  
 for each p in course.prerequisites  
 if tree.search(tree.root, p) == null  
 print "Missing prerequisite '" + p + "' for course " + course.number  
  
 return tree  
end procedure

## 3. Print a Single Course

procedure printCourseInfo(tree, courseNumber):  
 course = tree.search(tree.root, courseNumber)  
 if course == null:  
 print "Course not found: " + courseNumber  
 return  
  
 print "Course Number: " + course.number  
 print "Title: " + course.title  
  
 if course.prerequisites is empty:  
 print "Prerequisites: None"  
 else:  
 print "Prerequisites: " + join(course.prerequisites, ", ")

Menu Psuedocode

Function printMenu():

print "-------------------------------"

print " ABCU Course Menu"

print "-------------------------------"

print "1. Load course data from file"

print "2. Print all CS courses (alphanumeric order)"

print "3. Print one course's title and prerequisites"

print "9. Exit"

print "-------------------------------"

procedure mainMenu() //main loop?

// state tracked across menu selections

dataLoaded = false;

data;

while true

printMenu()

prompt "Enter selection: "

read selection

switch selection

case 1:

prompt "Enter path to course data file: "

read filename

//load data from file based on implementation chosen

data = loadCoursesFromFile(filename)

if data != null and isValid(data)

dataLoaded = true

print "Courses Loaded.”

else

dataLoaded = false

print "Error: failed to load file. Please try again."

end if

break

case 2:

if not dataLoaded

print "Please load the course data first."

break

end if

print "Computer Science Courses (Alphanumeric Order):”

printAllCoursesInOrder(data)

break

case 3:

if not dataLoaded

print("Please load the course data first (Option 1).")

break

end if

prompt "Enter a course number (e.g., CS200): "

read courseToFind

courseToFind = toUpper(trim(courseToFind))

course = findCourse(store, query) // null if not found

if course == null

print "Course not found: " + courseToFind

else

print "Course Number: " + course.number

print "Title: " + course.title

if isEmpty(course.prerequisites)

print "Prerequisites: None"

else

print "Prerequisites: " + join(course.prerequisites, ", "))

end if

end if

break

case 9:

print "Exiting program. Goodbye!"

break while

default:

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

end switch

print “\n”

end while

end procedure

PrintInOrder Psuedocode

//BST implementation just uses the inorder traversal with a display course,  
//such as:  
function inorder(node, action):  
 if node == null: return  
 inorder(node.left, action)  
 displayCourse(node.data)  
 inorder(node.right, action)

//Vectors and Hashtables will need to be sorted, then displayed.

//vector:

Function printInOrder():  
 quicksort(data)

For(course of data):

displayCourse(course)

//hashTable:

Function printInOrder():

Gather keys in list

Quicksort(keys)

For (courseKey of keys):

displayCourse(search(key))

Runtime Analysis

**Vector — (A) Populate from file**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| open file | 1 | 1 | 1 |
| init empty vector | 1 | 1 | 1 |
| for each line in file | 1 | n | n |
| split(line, ",") | 1 | n | n |
| validate token count | 1 | n | n |
| assign fields | 1 | n | n |
| create Course object | 1 | n | n |
| vectorAdd(course) | 1 (amortized) | n | n |
| close file | 1 | 1 | 1 |
| Total Cost | | | 5n + 3 |
| Runtime | | | O(n) |

**Vector — (B) Validate Prerequisites by Vector Scan**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for each course | 1 | n | n |
| for each prereq of the course | 1 | n | n |
| scan vector to find matching course | O(n) | n | n·n |
| Total Cost | | | n2+ n + r |
| Runtime | | | O(n2) |

**Hash Table — (A) Populate from file(Separate Chaining)**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| open file / init table of size N | 1 | 1 | 1 |
| for each line: split/validate/assign/create | 4 | n | 4n |
| insertCourse | O(1) avg / O(n) worst | n | n (avg) / n² (worst) |
| close file | 1 | 1 | 1 |
| Total (avg) |  |  | ≈ 5n + 2 → O(n) |
| Total (worst collisions) |  |  | ≈ n² + 4n + 2 → O(n²) |

**Hash Table — (B) Validate Prerequisites via Hash Lookup**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| for each prereq entry | 1 | r | r |
| findCourse (hash lookup) | O(1) avg / O(n) worst | r | r (avg) / r·n (worst) |
| Total (avg) |  |  | O(r) |
| Total (worst collisions) |  |  | O(r·n) |

**Binary Search Tree — (A) Populate from file**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| open file / init empty BST | 1 | 1 | 1 |
| for each line: split/validate/assign/create | 4 | n | 4n |
| bstInsert(course) | O(h) | n | n·h |
| close file | 1 | 1 | 1 |
| Total (balanced) |  |  | ≈ 4n + n·log n + 2 → O(n log n) |
| Total (worst unbalanced) |  |  | ≈ 4n + n² + 2 → O(n²) |

**Binary Search Tree — (B) Validate Prerequisites via BST Search**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| traverse courses | 1 | n | n |
| for each prereq of the course | 1 | r | r |
| bstSearch(prereq) | O(h) | r | r·h |
| Total (balanced) |  |  | O(r log n) |
| Total (worst unbalanced) |  |  | O(r·n) |

Advantages and Disadvantages Analysis

Overall, the Binary Search Tree has the fastest insert and search operations, which is what it was designed for. The disadvantage is the complex implementation and recursion required to make implement a BST. Which is why high-level languages like Java have an implementation built in. Hash Tables are great for key value pairs and O(1) lookups. However, depending on the implementation as they grow the lookups can be even longer than a vector or linked list in the worse case scenarios. Speaking of Vectors, vectors can be great for just storing ordered data for display. However, sorting and lookups at O(N) or greater make these the slowest data type of the 3.  
  
In this project I will likely work with a Binary Search Tree. Since lookup and display (especially in alphanumerical order) are the most important features of the program, a BST makes the most sense. It is the most performant option for this scenario for insertion, searching, and displaying in order (although quicksort is quite fast). Also this gives good practice for recursion and trees that I don’t often get to work with professionally in web development.