# CS 300 Pseudocode Document

## Function Signatures

// Vector pseudocode

// Function void loadData(file) to load data from file

// and store in vector data structure

{

OPEN file

IF file does not exist:

OUTPUT “File not found” error message

RETURN

END IF

INITIALIZE empty vector called courses

FOR each line in file:

IF line is empty:

CONTINUE

END IF

SPLIT line by delimiter “,” into courseData

IF length of courseData < 2:

OUTPUT “Invalid line format” error message

CONTINUE

END IF

SET courseNumber equal to courseData[0]

SET courseTitle equal to courseData[1]

INITIALIZE list prerequisites as empty list

IF length of courseData > 2:

FOR i from 2 to length of courseData:

SET courseData[i] equal to prerequisite

IF prerequisite does not exist in courses:

OUTPUT “Prerequisite not found” error message

END IF

ELSE:

ADD prerequisite to list prerequisites

END ELSE

END FOR

END IF

INITIALIZE new Course object with courseNumber, courseTitle, and prerequisites

ADD new course to courses vector

END FOR

CLOSE file

RETURN courses vector

}

// Function void printCourseInformation(courseNumber) to search

// for specific course and print information

{

SET foundCourse equal to null

FOR each course in vector data structure:

IF course.courseNumber equals courseNumber (key):

SET foundCourse equal to course

BREAK

END IF

END FOR

IF foundCourse is null:

OUTPUT “Course not found” error message

END IF

ELSE:

OUTPUT “Course Information:”, end with new line

OUTPUT “Number: “ + foundCourse.courseNumber, end with new line

OUTPUT “Title: “ + foundCourse.courseTitle, end with new line

OUTPUT “Prerequisites: “

FOR each prerequisite in foundCourses.prerequisites:

OUTPUT “- “ + prerequisite, end with new line

END FOR

END ELSE

}

// Function Vector mergeSort(Vector<Course> courses) that sorts

// all courses through a merge sort

{

IF length of courses <= 1:

RETURN courses

END IF

INITIALIZE int midpoint, SET equal to length of courses / 2

INITIALIZE left as vector of courses from index 0 to (mid – 1)

INITIALIZE right as vector of courses from index mid to end

CALL mergeSort on left vector, save return as left vector

CALL mergeSort on right vector, save return as right vector

RETURN the value returned by CALL of function merge(left, right)

}

// Function Vector merge(Vector<Course> left, Vector<Course> right)

// that is a private recursive function, compares course numbers

// to sort them through merge

{

INITALIZE Vector mergedCourses of type courses

WHILE both left and right are not empty:

IF first element of left <= first element of right:

APPEND first element of left to mergedCourses

REMOVE first element from left

END IF

ELSE:

APPEND first element of right to mergedCourses

REMOVE first element from right

END WHILE

APPEND all remaining elements of left to mergedCourses

APPEND all remaining elements of right to mergedCourses

RETURN mergedCourses

}

// Function void printAllCourses(Vector<Course> courses) that

// displays all courses and their information

{

INITIALIZE vector sortedCourses

SET sortedCourses equal to return from CALL mergeSort(courses)

FOR each course in sortedCourses:

CALL printCourseInformation(currentCourse.courseNumber)

END FOR

}

// Hashtable pseudocode

// Function void loadData(file) to load data from file

{

OPEN file

IF file does not exist:

OUTPUT “File not found” error message

RETURN

END IF

INITIALIZE empty hash table called courses

FOR each line in file:

IF line is empty:

CONTINUE

END IF

SPLIT line by delimiter “,” into courseData

IF length of courseData < 2:

OUTPUT “Invalid line format” error message

CONTINUE

END IF

SET courseNumber equal to courseData[0]

SET courseTitle equal to courseData[1]

INITIALIZE list prerequisites as empty list

IF length of courseData > 2:

FOR i from 2 to length of courseData:

SET prerequisite equal to courseData[i]

IF prerequisite does not exist in courses:

OUTPUT “Prerequisite not found” error message

END IF

ELSE:

ADD prerequisite to list prerequisites

END ELSE

END FOR

END IF

INITIALIZE new Course object with courseNumber, courseTitle, and prerequisites

CALCULATE hash table index using key = hash(course.courseNumber)

ADD new course to courses hash table with index key

END FOR

CLOSE file

RETURN courses

}

// Function void printCourseInformation (Hashtable<course> courses,

// String courseNumber) to print course information

{

SET course equal to hash(course.courseNumber)

IF course is null:

OUTPUT “Course not found” error message

END IF

ELSE:

OUTPUT “Course Information:”, end with new line

OUTPUT “Number: “ + foundCourse.courseNumber, end with new line

OUTPUT “Title: “ + foundCourse.courseTitle, end with new line

OUTPUT “Prerequisites: “

FOR each prerequisite in foundCourses.prerequisites:

OUTPUT “- “ + prerequisite, end with new line

END FOR

END ELSE

}

// Function Vector mergeSort(Vector<Course> courses) that sorts

// all courses through a merge sort

{

IF length of courses <= 1:

RETURN courses

END IF

INITIALIZE int midpoint, SET equal to length of courses / 2

INITIALIZE left as vector of courses from index 0 to (mid – 1)

INITIALIZE right as vector of courses from index mid to end

CALL mergeSort on left vector, save return as left vector

CALL mergeSort on right vector, save return as right vector

RETURN the value returned by CALL of function merge(left, right)

}

// Function Vector merge(Vector<Course> left, Vector<Course> right)

// that is a private recursive function, compares course numbers

// to sort them through merge

{

INITALIZE Vector mergedCourses of type courses

WHILE both left and right are not empty:

IF first element of left <= first element of right:

APPEND first element of left to mergedCourses

REMOVE first element from left

END IF

ELSE:

APPEND first element of right to mergedCourses

REMOVE first element from right

END WHILE

APPEND all remaining elements of left to mergedCourses

APPEND all remaining elements of right to mergedCourses

RETURN mergedCourses

}

// Function void printAllCourses(Hashtable<Course> courses) that

// displays all courses and their information

{

INITIALIZE vector coursesVector

FOR each course in hashTable:

Append course to coursesVector

END FOR

INITIALIZE vector sortedCourses

SET sortedCourses equal to return from CALL mergeSort(courses)

FOR each course in sortedCourses:

CALL printCourseInformation(currentCourse.courseNumber)

END FOR

}

// Tree pseudocode

// Function Node\* loadData(file) to load data from file

{

OPEN file

IF file does not exist:

OUTPUT “File not found” error message

RETURN

END IF

INITIALIZE empty root of bst

FOR each line in file:

IF line is empty:

CONTINUE

END IF

SPLIT line by delimiter “,” into courseData

IF length of courseData < 2:

OUTPUT “Invalid line format” error message

CONTINUE

END IF

SET courseNumber equal to courseData[0]

SET courseTitle equal to courseData[1]

INITIALIZE list prerequisites as empty list

IF length of courseData > 2:

FOR i from 2 to length of courseData:

SET prerequisite equal to courseData[i]

IF prerequisite does not exist in courses:

OUTPUT “Prerequisite not found” error message

END IF

ELSE:

ADD prerequisite to list prerequisites

END ELSE

END FOR

END IF

INITIALIZE new Course object with courseNumber, courseTitle, and prerequisites, SET as courseObj

CALL insertNode function, pass in root and courseObj

END FOR

CLOSE file

RETURN

}

// Function Node\* insertNode(Node\* node, Course courseObj) to store

// objects in bst

{

IF node is empty:

OUTPUT “No valid course data found in file” error message

RETURN node

END IF

ELSE IF courseObj.courseNumber < node.courseNumber:

CALL insertNode(node.left, courseObj),

SET return value equal to node.left

END ELSE IF

ELSE IF courseObj.courseNumber > node.courseNumber:

CALL printCourseInformation(node.right, courseObj),

SET return value equal to node.right

END ELSE IF

OUTPUT “Course objects created and stored successfully” confirmation message

RETURN node

}

// Function void printCourseInformation (Node\* node, String

// courseNumber) to print course information

{

IF node is null:

OUTPUT “Course not found” error message

RETURN

END IF

IF courseNumber < node.courseNumber:

CALL printCourseInformation(node.left, courseNumber)

END IF

ELSE IF courseNumber > node.courseNumber:

CALL printCourseInformation(node.right, courseNumber)

END ELSE IF

ELSE:

OUTPUT “Course Information:”, end with new line

OUTPUT “Number: “ + node.courseNumber, end with new line

OUTPUT “Title: “ + node.courseTitle, end with new line

OUTPUT “Prerequisites: “

FOR each prerequisite in node.prerequisites:

OUTPUT “- “ + prerequisite, end with new line

END FOR

END ELSE

}

// Function void inOrderTraversal(Node\* node) that

// displays all courses in alphanumeric order through

// in-order traversal of the binary search tree

{

IF node is null:

RETURN

END IF

CALL function inOrderTraversal(node.left) on left subtree

CALL function printCourseInformation(node, node.courseNumber)

CALL function inOrderTraversal(node.right) on right subtree

}

// Function void printAllCourses(root) that

// displays all courses and their information

{

CALL function inOrderTraversal(root)

}

// Menu pseudocode in main()

{

OUTPUT menu in following format:

1: Load Courses From File

2: Print All Courses

3: Print Specific Course

9: Exit

Please select an option:

INPUT int userChoice from user

WHILE userChoice is not 9:

SWITCH:

CASE 1:

CALL function loadData(file)

END CASE 1

CASE 2:

CALL function printAllCourses(courses)

END CASE 2

CASE 3:

OUTPUT prompt for user to enter desired course number

INPUT desired course number to search for from user

CALL function printCourseInformation(courseNumber)

END CASE 3

END SWITCH

END WHILE

RETURN

}

## Runtime Analysis

Vector Runtime Evaluation

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **If file does not exist** | 1 | 1 | 1 |
| **Print error message** | 1 | 1 | 1 |
| **Initialize vector courses** | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| If line is empty | 1 | n | n |
| Continue | 1 | n | n |
| Split line by delimiter “,” into courseData | 1 | n | n |
| If length of courseData < 2 | 1 | n | n |
| Print error message | 1 | n | n |
| Continue | 1 | n | n |
| Set courseNumber = courseData[0] | 1 | n | n |
| Set courseTitle = courseData[1] | 1 | n | n |
| Initialize list prerequisites | 1 | n | n |
| If length courseData > 2 | 1 | n | n |
| For i from 2 to length of courseData | 1 | x\*n | x\*n |
| Set prerequisite = courseData[i] | 1 | x\*n | x\*n |
| Append prerequisites to list prerequisites | 1 | x\*n | x\*n |
| Initialize new course object with courseNumber, courseTitle, prerequisites | 1 | n | n |
| Append course to courses | 1 | n | n |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | 11n + 3xn + 4 |
| **Runtime** | | | O(n) |

Hash Table Runtime Evaluation

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **If file does not exist** | 1 | 1 | 1 |
| **Print error message** | 1 | 1 | 1 |
| **Initialize hash table courses** | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| If line is empty | 1 | n | n |
| Continue | 1 | n | n |
| Split line by delimiter “,” into courseData | 1 | n | n |
| If length of courseData < 2 | 1 | n | n |
| Print error message | 1 | n | n |
| Continue | 1 | n | n |
| Set courseNumber = courseData[0] | 1 | n | n |
| Set courseTitle = courseData[1] | 1 | n | n |
| Initialize list prerequisites | 1 | n | n |
| If length courseData > 2 | 1 | n | n |
| For i from 2 to length of courseData | 1 | x\*n | x\*n |
| Set prerequisite = courseData[i] | 1 | x\*n | x\*n |
| Append prerequisites to list prerequisites | 1 | x\*n | x\*n |
| Initialize new course object with courseNumber, courseTitle, prerequisites | 1 | n | n |
| Calculate hash value using key=hash(course.courseNumber) | 4 | n | 4n |
| Insert course to courses at index key | 1 | n | n |
| **Total Cost** | | | 14n + 3xn + 4 |
| **Runtime** | | | O(n) |

Binary Search Tree Runtime Evaluation

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **If file does not exist** | 1 | 1 | 1 |
| **Print error message** | 1 | 1 | 1 |
| **Initialize vector courses** | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| If line is empty | 1 | n | n |
| Continue | 1 | n | n |
| Split line by delimiter “,” into courseData | 1 | n | n |
| If length of courseData < 2 | 1 | n | n |
| Print error message | 1 | n | n |
| Continue | 1 | n | n |
| Set courseNumber = courseData[0] | 1 | n | n |
| Set courseTitle = courseData[1] | 1 | n | n |
| Initialize list prerequisites | 1 | n | n |
| If length courseData > 2 | 1 | n | n |
| For i from 2 to length of courseData | 1 | x\*n | x\*n |
| Set prerequisite = courseData[i] | 1 | x\*n | x\*n |
| Append prerequisites to list prerequisites | 1 | x\*n | x\*n |
| Initialize new course object with courseNumber, courseTitle, prerequisites | 1 | n | n |
| Call insertNode function to insert into bst | n | n | n2 |
| Close file | 1 | 1 | 1 |
| **Total Cost** | | | n2 + 9n + 3xn + 4 |
| **Runtime** | | | O(n2) |

**Data Structure Evaluation**

Vectors, hash tables, and binary search trees are all used to store and organize data, but each offer their own unique strengths and weaknesses. Vectors are dynamic, resizable array-like containers that store elements in contiguous memory locations, which enables efficient random access through indexing. They have a flexible size and can dynamically grow or shrink as elements are added or removed. The time complexity for random access is constant, as well as insertion and deletion operations at the beginning or end of a vector. However, inserting or removing elements in the middle requires shifting elements, making these operations less efficient (O(n)). Vectors are best utilized in scenarios where efficient random access and dynamic resizing are required.

Hash tables use a hash function to map keys to array indices, allowing for efficient key-value retrieval. They provide constant-time complexity for insertion, deletion, and search operations, making them extremely efficient for accessing data using keys. Their performance, however, depends heavily upon the quality of the hash function and how well it distributes keys across the array. A collision, which occurs when different keys are mapped to the same array index, requires a special resolution technique to be handled. Hash tables are best suited for scenarios where fast data retrieval on keys is essential.

Binary search trees are hierarchical data structures where each node has at most two children, a left child and a right child. The elements in a BST are organized based on their relative values, with the left child being smaller (or equal to) and the right child being larger (or equal to) than the parent node. Searching, insertion, and deletion operations in a balanced BST have a time complexity of O(log n), where n is the number of nodes in the tree. In the worst case (unbalanced tree), these operations could become O(n). BSTs are efficient for maintaining a sorted collection of elements and can be used for tasks like searching, in-order traversal, and maintaining sorted order efficiently. However, in worst-case scenario, an unbalanced BST can degrade to a linked list with linear-time operations, making it less suitable for tasks with significant dynamic data changes or when a balanced tree is not guaranteed.

In summary, vectors provide dynamic resizing and efficient random access, hash tables offer constant-time key-based access and are ideal for associative arrays, and binary search trees maintain sorted order efficiently while providing logarithmic search and insertion time complexity (assuming they are balanced).

For the scenario of storing and organizing course information for ABC University, a hash table would be the most suitable data structure. This is due to its efficient key-based access, fast data retrieval, and flexible size. The hash table data structure provides constant-time complexity for insertion, deletion, and search operations, assuming a well-distributed hash function. Since the advisors at ABCU need to access course information based on the course number (key), a hash table allows them to efficiently retrieve course details without having to traverse the entire data structure.

Similarly, a primary requirement for the advisors is to quickly access course information using course numbers. Hash tables excel at key-based data retrieval, making them ideal for this scenario. The constant-time complexity for these operations ensures quick access to course information. Since hash tables can dynamically grow or shrink as elements are added or removed, their flexibility allows for efficient memory usage and ensures that the hash table can handle varying amounts of course data without wasting memory.

Although collisions can occur in hash tables, there are effective collision resolution techniques, such as chaining, that can handle these situations. With a proper collision resolution strategy, hash tables can maintain their constant-time complexity for most operations. While binary search trees could potentially provide sorted order for the course information, ensuring a balanced BST would require additional effort. Unbalanced BSTs can lead to performance degradation, reducing the advantages of their logarithmic time complexity for search and insertion operations.

In conclusion, a hash table is recommended for this scenario because it offers efficient key-based access, constant-time complexity for most operations, and a dynamic size that can handle varying amounts of course data. It provides a well-organized and efficient data structure for the advisors to access course information quickly and effectively at ABC University.