# CS 300 Pseudocode Document

**Pseudocode – Vector Data Structure:**

// Vector Pseudocode

// Create a class for course

class Course {

Variables:

string courseNumber

string courseTitle

Vector<string> prerequisites

}

// Create a vector to hold all the course objects

Vector<Course> courseList

// Open the course information file

void openFile() {

Try to open “CourseInformation.txt” as file

if file cannot be opened

print “File could not be opened”

return false

return true

}

// Parse each line from the course information file

void parseFile() {

for each line in file

// Assuming a space-delimited file

courseData = line.split(“ “)

// Validate the data

if courseData.length < 2

print “Invalid course data”

continue

courseNumber = courseData[0]

courseTitle = courseData[1]

prerequisites = courseData[2..end] // Slice the data for prerequisites

// Create a new course object with the parsed data

course = new Course(courseNumber, courseTitle, prerequisites)

// Add the course object to vector

courseList.append(course)

}

// Find a specific course in the course list vector

Course findCourse(string courseNumber) {

for each course in courseList

if course.courseNumber == courseNumber

return course

print “Course not found”

return null

}

// Print course information

void printCourseInformation(Course course) {

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

// Load data structure

void loadDataStructure() {

if openFile() == true:

parseFile()

}

// Print Course List

void printCourseList() {

Sort the courseList by courseNumber

for each course in courseList

print course.courseNumber and course.courseTitle

}

// Main method

main() {

while true {

print "MENU"

print "1. Load Data Structure"

print "2. Print Course List"

print "3. Print Course"

print "4. Exit"

userOption = get user input

if userOption == 1:

loadDataStructure()

else if userOption == 2:

printCourseList()

else if userOption == 3:

courseNumber = get user input for course number

course = findCourse(courseNumber)

if course != null:

printCourseInformation(course)

else if userOption == 4:

Exit the program

}

}

**Pseudocode – Hash Table Data Structure:**

// Hash Table Pseudocode

// Create a class for course

class Course:

Variables:

string courseNumber

string courseTitle

HashTable<string, string> prerequisites

// Create a HashTable to hold all the course objects

HashTable<string, Course> courseTable

// Open the course information file

bool openFile() {

Try to open “CourseInformation.txt” as file

if file cannot be opened

print “File could not be opened”

return false

return true

}

// Parse each line from the course information file

void parseFile() {

for each line in file

// Assuming a space-delimited file

courseData = line.split(“ “)

// Validate the data

if courseData.length < 2

print “Invalid course data”

continue

courseNumber = courseData[0]

courseTitle = courseData[1]

prerequisites = courseData[2..end] // Slice the data for prerequisites

// Create a new course object with the parsed data

course = new Course(courseNumber, courseTitle, prerequisites)

// Add the course object to HashTable

courseTable.insert(courseNumber, course)

}

// Find a specific course in the course table

Course findCourse(string courseNumber) {

if courseTable.contains(courseNumber)

return courseTable.get(courseNumber)

print “Course not found”

return null

}

// Print course information

void printCourseInformation(string courseNumber) {

if courseTable.contains(courseNumber)

course = courseTable.get(courseNumber)

print out the course information

for each prerequisite of the course

if courseTable.contains(prerequisite)

print courseTable.get(prerequisite).courseInformation

}

// Load data structure

void loadDataStructure() {

if openFile() == true:

parseFile()

}

// Print Course List

void printCourseList() {

courseKeys = Sort the courseTable.keys() alphanumerically

for each key in courseKeys

print courseTable.get(key).courseNumber and courseTable.get(key).courseTitle

}

// Main method

main() {

While true {

print "MENU"

print "1. Load Data Structure"

print "2. Print Course List"

print "3. Print Course"

print "4. Exit"

userOption = get user input

if userOption == 1:

loadDataStructure()

else if userOption == 2:

printCourseList()

else if userOption == 3:

courseNumber = get user input for course number

course = findCourse(courseNumber)

if course != null:

printCourseInformation(courseNumber)

else if userOption == 4:

Exit the program

}

}

**Pseudocode – Tree Data Structure:**

// Tree implementation

// Course Node Pseudocode

class CourseNode {

variables:

string courseNumber

string courseTitle

BST<CourseNode> prerequisites // BST of prerequisites

}

// Binary Search Tree for Course

class BST {

variables:

CourseNode root

// Insert a node into the BST

void insert(CourseNode node) {

if root == null

root = node

else

insertNode(root, node)

}

// Helper function to recursively find the correct spot for insertion

void insertNode(CourseNode current, CourseNode node) {

if node.courseNumber < current.courseNumber

if current.left == null

current.left = node

else

insertNode(current.left, node)

else

if current.right == null

current.right = node

else

insertNode(current.right, node)

}

// Find a node in the BST

CourseNode find(string courseNumber) {

return findNode(root, courseNumber)

}

// Helper function to recursively find the node

CourseNode findNode(CourseNode current, string courseNumber) {

if current == null

return null

if current.courseNumber == courseNumber

return current

else if courseNumber < current.courseNumber

return findNode(current.left, courseNumber)

else

return findNode(current.right, courseNumber)

}

}

// Main BST to hold all the courses

BST<CourseNode> courseBST = new BST<CourseNode>()

// Open the course information file

void openFile() {

Open “CourseInformation.txt” as file

if file cannot be opened

print “File could not be opened”

}

// Parse each line from the course information

void parseFile() {

for each line in file

// Assuming a space-delimited file

courseData = line.split(“ “)

// Validate the data

if courseData.length < 2

print “Invalid course data”

continue

courseNumber = courseData[0]

courseTitle = courseData[1]

prerequisitesData = courseData[2..end] // Slice the data for prerequisites

// Create a new course node with the parsed information

courseNode = new CourseNode(courseNumber, courseTitle)

// Add the course node to the BST

courseBST.insert(courseNode)

// Process prerequisites

for each prerequisite in prerequisitesData

prerequisiteNode = courseBST.find(prerequisite)

if prerequisiteNode == null

print "Prerequisite course does not exist"

else

courseNode.prerequisites.insert(prerequisiteNode)

}

// Print course information

void printCourseInformation(string courseNumber) {

courseNode = courseBST.find(courseNumber)

if courseNode != null

print out the course information

printCoursePrerequisites(courseNode)

else

print "Course not found"

}

void printCoursePrerequisites(CourseNode node) {

// This function will need to traverse the BST of prerequisites

// and print the course information for each.

// The exact implementation depends on how you want the output formatted.

}

// Main method

main() {

openFile()

parseFile()

printCourseInformation("CS101")

}

**Runtime Analysis**

**Vector**:

Runtime:

* Insertion at the end of a vector takes constant time O(1), but inserted in the middle takes linear time O(n) as all following elements must be shifted.
* Searching for a course in a vector takes linear time O(n) as, in the worst case, we may have to traverse the entire vector.
* Sorting the vector will take O(n log n) time, which will be necessary to print the courses in sorted order.

Memory:

* Vectors use contiguous memory, which can lead to a waste of space if the vector is not full. However, it can dynamically grow when required.

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| For each line in file (parse) | 1 | N | N |
| split line into course data | 1 | N | N |
| create new Course object | 1 | 1 | 1 |
| add Course object to vector | 1 | N | N |
| for each course in vector (search) | 1 | N | N |
| If courseNumber matches | 1 | 1 | 1 |
| Total Cost | | | 4N + 2 |
| Runtime | | | O(N) |

**Hash Table**:

Runtime:

* Insertion, deletion, and search operations in a hash table typically take constant time O(1), assuming a good hash function that distributes keys uniformly.
* However, in the worst case (when all keys hash to the same index), these operations can take linear time O(n).

Memory:

* Hash tables require more memory than arrays because they need to maintain the hash table and linked lists for handling collisions.

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| for each line in file (parse) | 1 | N | N |
| split line into course data | 1 | N | N |
| create new Course object | 1 | 1 | 1 |
| insert Course object into hash table | 1 | N | N |
| retrieve Course object (search) | 1 | 1 | 1 |
| Total Cost | | | 3N + 2 |
| Runtime | | | O(N) |

**Tree Structures**:

Runtime:

* In a balanced tree, insertion, deletion, and search operations take logarithmic time, represented as O(log n), in both the best and average cases.
* However, when the tree becomes skewed or unbalanced, these operations could take linear time, represented as O(n), in the worst-case scenario.
* An in-order traversal of a balanced binary tree would yield elements in sorted order, and this operation takes O(n) time.

Memory:

* Tree structures generally require more memory than arrays and linked lists due to the need to store additional pointers for child nodes. These could be to the left and right child nodes in the case of binary trees or multiple child nodes in the case of n-ary trees.
* Remember, the efficiency of operations and memory usage in tree structures can depend highly on their type and state (balanced vs. unbalanced) and the application's specific requirements.

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| for each line in file (parse) | 1 | N | N |
| split line into course data | 1 | N | N |
| create new CourseNode object | 1 | 1 | 1 |
| insert CourseNode object into BST | logN | N | NlogN |
| find CourseNode (search) | logN | 1 | NlogN |
| Total Cost | | | 2N + NlogN + logN + 1 |
| Runtime | | | O(NlogN) |

**Evaluation**:

Based on these analyses, each data structure has pros and cons.

* Vectors are simple and require less memory but may be inefficient for search and insert operations.
* Hash Tables are fast for search, insert, and delete operations but require more memory and have slower worst-case performance.
* Tree structures offer a compelling balance between efficient search and insertion operations and the advantage of seamlessly printing elements in sorted order. However, they tend to require more memory than other data structures. Their efficiency is also contingent on maintaining a balanced state, achievable through certain types of trees like self-balancing binary search trees. Therefore, while tree structures offer specific, powerful capabilities, their memory footprint and need for balance should also be considered when considering them for specific applications.

A tree-based data structure is the most suitable option for the specific needs of the advising program we are discussing. This decision is driven by the need for efficient search and inserts operations and the frequent requirement to print courses in a sorted order.

Our analysis and demonstrations specifically employed a Binary Search Tree (BST) to represent tree structures. The use of BST in our discussion serves an illustrative purpose rather than declaring it as the definitive choice.

While BSTs can become inefficient when they become unbalanced, this can be effectively circumvented by utilizing self-balancing binary search trees, such as AVL or Red-Black trees. These types of trees automatically maintain balance, thereby ensuring consistently efficient operations. Therefore, a tree structure, in one of its many forms, can provide an effective solution for the requirements of our advising program.