**Evaluation of Data Structures**

**Vector Data Structure**

**Load Courses from File**

void loadCourses(String filename, Vector<Course> &courses) {

File file = open(filename) // Cost: 1, Executes: 1

if file is not open { // Cost: 1, Executes: 1

print "Error: Cannot open file" // Cost: 1, Executes: 1

return // Cost: 1, Executes: 1

}

while not end of file { // Cost: 1, Executes: n

String line = readLine(file) // Cost: 1, Executes: n

Vector<String> tokens = split(line, ',') // Cost: 1, Executes: n

if length(tokens) < 2 { // Cost: 1, Executes: n

print "Error: Line does not have enough parameters" // Cost: 1, Executes: n

continue // Cost: 1, Executes: n

}

Course course // Cost: 1, Executes: n

course.courseNumber = tokens[0] // Cost: 1, Executes: n

course.courseTitle = tokens[1] // Cost: 1, Executes: n

for i from 2 to length(tokens) - 1 { // Cost: 1, Executes: p (average number of prerequisites per course)

course.prerequisites.push\_back(tokens[i]) // Cost: 1, Executes: p

}

courses.push\_back(course) // Cost: 1, Executes: n

}

close(file) // Cost: 1, Executes: 1

validateCourses(courses) // Cost: O(n + p), Executes: 1

}

**Validate Courses**

void validateCourses(Vector<Course> &courses) {

Set<String> courseNumbers // Cost: 1, Executes: 1

for each course in courses { // Cost: 1, Executes: n

courseNumbers.insert(course.courseNumber) // Cost: 1, Executes: n

}

for each course in courses { // Cost: 1, Executes: n

for each prerequisite in course.prerequisites { // Cost: 1, Executes: p

if not courseNumbers.contains(prerequisite) { // Cost: 1, Executes: p

print "Error: Prerequisite " + prerequisite + " for course " + course.courseNumber + " does not exist" // Cost: 1, Executes: p

}

}

}

}

**Hash Table Data Structure**

**Load Courses from File**

void loadCourses(String filename, HashTable<Course> &courses) {

File file = open(filename) // Cost: 1, Executes: 1

if file is not open { // Cost: 1, Executes: 1

print "Error: Cannot open file" // Cost: 1, Executes: 1

return // Cost: 1, Executes: 1

}

while not end of file { // Cost: 1, Executes: n

String line = readLine(file) // Cost: 1, Executes: n

Vector<String> tokens = split(line, ',') // Cost: 1, Executes: n

if length(tokens) < 2 { // Cost: 1, Executes: n

print "Error: Line does not have enough parameters" // Cost: 1, Executes: n

continue // Cost: 1, Executes: n

}

Course course // Cost: 1, Executes: n

course.courseNumber = tokens[0] // Cost: 1, Executes: n

course.courseTitle = tokens[1] // Cost: 1, Executes: n

for i from 2 to length(tokens) - 1 { // Cost: 1, Executes: p

course.prerequisites.push\_back(tokens[i]) // Cost: 1, Executes: p

}

courses.insert(course.courseNumber, course) // Cost: O(1), Executes: n

}

close(file) // Cost: 1, Executes: 1

validateCourses(courses) // Cost: O(n + p), Executes: 1

}

**Validate Courses**

void validateCourses(HashTable<Course> &courses) {

Set<String> courseNumbers // Cost: 1, Executes: 1

for each course in courses { // Cost: 1, Executes: n

courseNumbers.insert(course.courseNumber) // Cost: 1, Executes: n

}

for each course in courses { // Cost: 1, Executes: n

for each prerequisite in course.prerequisites { // Cost: 1, Executes: p

if not courseNumbers.contains(prerequisite) { // Cost: 1, Executes: p

print "Error: Prerequisite " + prerequisite + " for course " + course.courseNumber + " does not exist" // Cost: 1, Executes: p

}

}

}

}

**Binary Search Tree Data Structure**

**Load Courses from File**

void loadCourses(String filename, Tree<Course> &courses) {

File file = open(filename) // Cost: 1, Executes: 1

if file is not open { // Cost: 1, Executes: 1

print "Error: Cannot open file" // Cost: 1, Executes: 1

return // Cost: 1, Executes: 1

}

while not end of file { // Cost: 1, Executes: n

String line = readLine(file) // Cost: 1, Executes: n

List<String> tokens = split(line, ',') // Cost: 1, Executes: n

if length(tokens) < 2 { // Cost: 1, Executes: n

print "Error: Line does not have enough parameters" // Cost: 1, Executes: n

continue // Cost: 1, Executes: n

}

Course course // Cost: 1, Executes: n

course.courseNumber = tokens[0] // Cost: 1, Executes: n

course.courseTitle = tokens[1] // Cost: 1, Executes: n

for i from 2 to length(tokens) - 1 { // Cost: 1, Executes: p

course.prerequisites.push\_back(tokens[i]) // Cost: 1, Executes: p

}

insertCourse(courses, course) // Cost: O(log n), Executes: n

}

close(file) // Cost: 1, Executes: 1

validateCourses(courses) // Cost: O(n + p), Executes: 1

}

**Validate Courses**

void validateCourses(Tree<Course> &courses) {

Set<String> courseNumbers // Cost: 1, Executes: 1

for each course in courses { // Cost: 1, Executes: n

courseNumbers.insert(course.courseNumber) // Cost: 1, Executes: n

}

for each course in courses { // Cost: 1, Executes: n

for each prerequisite in course.prerequisites { // Cost: 1, Executes: p

if not courseNumbers.contains(prerequisite) { // Cost: 1, Executes: p

print "Error: Prerequisite " + prerequisite + " for course " + course.courseNumber + " does not exist" // Cost: 1, Executes: p

}

}

}

}

**Vector Data Structure**

**Load Courses from File**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Line** | **Line Cost** | **# Times Executed** | **Total Cost** |
| File file = open(filename) | 1 | 1 | 1 |
| if file is not open { | 1 | 1 | 1 |
| return | 1 | 1 | 1 |
| while not end of file { | 1 | n | n |
| String line = readLine(file) | 1 | n | n |
| Vector<String> tokens = split(line, ',') | 1 | n | n |
| if length(tokens) < 2 { | 1 | n | n |
| continue | 1 | n | n |
| Course course | 1 | n | n |
| course.courseNumber = tokens[0] | 1 | n | n |
| course.courseTitle = tokens[1] | 1 | n | n |
| for i from 2 to length(tokens) - 1 { | 1 | p | p |
| course.prerequisites.push\_back(tokens[i]) | 1 | p | p |
| courses.push\_back(course) | 1 | n | n |
| close(file) | 1 | 1 | 1 |

**Total Cost**: 2n+2p+6

**Big O**: O(n+p)

**Hash Table Data Structure**

**Load Courses from File**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Line** | **Line Cost** | **# Times Executed** | **Total Cost** |
| File file = open(filename) | 1 | 1 | 1 |
| if file is not open { | 1 | 1 | 1 |
| return | 1 | 1 | 1 |
| while not end of file { | 1 | n | n |
| String line = readLine(file) | 1 | n | n |
| Vector<String> tokens = split(line, ',') | 1 | n | n |
| if length(tokens) < 2 { | 1 | n | n |
| continue | 1 | n | n |
| Course course | 1 | n | n |
| course.courseNumber = tokens[0] | 1 | n | n |
| course.courseTitle = tokens[1] | 1 | n | n |
| for i from 2 to length(tokens) - 1 { | 1 | p | p |
| course.prerequisites.push\_back(tokens[i]) | 1 | p | p |
| courses.insert(course.courseNumber, course) | O(1) | n | O(1) \* n |
| close(file) | 1 | 1 | 1 |

**Total Cost**: 3n+2p+6

**Big O**: O(n+p)

**Binary Search Tree Data Structure**

**Load Courses from File**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code Line** | **Line Cost** | **# Times Executed** | **Total Cost** |
| File file = open(filename) | 1 | 1 | 1 |
| if file is not open { | 1 | 1 | 1 |
| return | 1 | 1 | 1 |
| while not end of file { | 1 | n | n |
| String line = readLine(file) | 1 | n | n |
| List<String> tokens = split(line, ',') | 1 | n | n |
| if length(tokens) < 2 { | 1 | n | n |
| continue | 1 | n | n |
| Course course | 1 | n | n |
| course.courseNumber = tokens[0] | 1 | n | n |
| course.courseTitle = tokens[1] | 1 | n | n |
| for i from 2 to length(tokens) - 1 { | 1 | p | p |
| course.prerequisites.push\_back(tokens[i]) | 1 | p | p |
| insertCourse(courses, course) | O(log n) | n | O(n \* log n) |
| close(file) | 1 | 1 | 1 |

**Total Cost**: 2n+2p+O(n log n)+6

**Big O**: O(n log n+p)

**Advantages and Disadvantages**

**Vector**

* **Advantages**:
  + Simple to implement and use.
  + Provides fast access to elements via indexing.
  + Suitable for small datasets.
* **Disadvantages**:
  + Inefficient for dynamic data because it requires shifting elements.
  + Poor performance for search operations compared to other structures like hash tables and trees.

**Hash Table**

* **Advantages**:
  + Provides average-case constant-time complexity (O(1)) for insertion, deletion, and search operations.
  + Efficient for large datasets.
* **Disadvantages**:
  + Potential for hash collisions which can degrade performance to O(n).
  + Requires more memory due to the storage of hash table buckets.

**Binary Search Tree**

* **Advantages**:
  + Maintains elements in sorted order, facilitating efficient in-order traversal.
  + Provides logarithmic time complexity (O(log n)) for search, insertion, and deletion operations.
* **Disadvantages**:
  + Tree balance is important. Unbalanced trees can degrade performance to O(n).
  + More complex to implement compared to vectors and hash tables.

**Recommendation**

Based on the analysis, I recommend using the **Hash Table** data structure. The hash table offers the best average case performance for insertion and search operations. While vectors are simple and suitable for small datasets, they do not perform well with dynamic data. Binary search trees provide sorted order but require balancing to maintain efficient performance. The hash table strikes a good balance between performance and complexity, making it the most efficient choice for handling a potentially large dataset of courses.