**Project One**

John Miller

CS – 300

August 11, 2024

Contents:

* PsuedoCode
  + Vector……….2
  + Hash Table. …3
  + BST………....6
  + MainMenu…..7
  + Alphanumeric 8
* Big O analysis………10
* Conclusion………….11
* Recommendation…...12

**Vector**

Read File:

TRY to open file IN file\_path

FOR each line IN file

SPLIT line by ‘,’ into tokensList

IF number in tokensList IS LESS than 2

PRINT Invalid file format

CLOSE file

RETURN

ENDIF

IF number in tokensList EQUALS 2

NEW course(courseName, courseID)

ADD course TO courseList

ELSE if number in tokensList is GREATER THAN 2

FOR token starting at index 2 IN tokenLIST

ADD token to prerequisites

END FORLOOP

NEW course(courseName, courseID, prerequisites)

ADD course TO courseList

ENDIF

END FORLOOP

CLOSE file

FOR each course IN courseList

IF course prerequisiteList IS NOT empty

FOR each prerequisite IN prerequisiteList

FOR each course IN courseLists

IF prerequisite EQUALS course

VALID prerequisite NEXT prerquisite

ENDIF

END FORLOOP

IF no match found

PRINT “Invalid “prerequisite” not found in courses”

ENDIF

END FORLOOP  
 ENDIF  
 END FORLOOP

Structure Of Courses

STRING courseName

STRING courseID

VECTOR prerequisiteList

CONSTRUCTOR course(courseName, courseID,)

THIS courseName EQUALS courseName

THIS courseID EQUALS courseID

CONSTRUCTOR course(courseName, courseID, prerequisiteList)

THIS courseName EQUALS courseName

THIS courseID EQUALS courseID

THIS prerequisiteList EQUALS prerequisiteList

Search and Print

FOR each course IN courseList

IF course EQUALS search parameter

IF prerequisiteList size is GREATER THAN 0

PRINT ‘courseName’: ‘courseID’ NEWLINE

PRINT “Prerequisities:” NEWLINE

FOR each prereq IN prerequisiteList

PRINT prereq NEWLINE

RETURN

ELSE IF prerequisiteList size EQUALS 0

PRINT ‘courseName’ : ‘courseList’ NEWLINE

RETURN

ENDIF

END FORLOOP

IF no match found

PRINT “No match found”

**Hash Table**

Structure Of Courses

STRING courseName

STRING courseID

VECTOR prerequisiteList

POINTERtoCourse next

INTEGER key EQUALS maxValue

CONSTRUCTOR course(courseName, courseID, key)

THIS courseName EQUALS courseName

THIS courseID EQUALS courseID

THIS key EQUALS key

Next EQUALS NULL POINTER

CONSTRUCTOR course(courseName, courseID, prerequisiteList, key)

THIS courseName EQUALS courseName

THIS courseID EQUALS courseID

THIS prerequisiteList EQUALS prerequisiteList

THIS key EQUALS key

Next EQUALS NULL POINTER

Read File:

TRY to open file IN file\_path

FOR each line IN file

SPLIT line by ‘,’ into tokensList

IF number in tokensList IS LESS than 2

PRINT Invalid file format

CLOSE file

RETURN

ENDIF

IF number in tokensList EQUALS 2

ADD course(courseName, courseID) TO courseList

ELSE if number in tokensList is GREATER THAN 2

FOR token starting at index 2 IN tokenLIST

ADD token to prerequisites

END FORLOOP

ADD course(courseID, courseName, prerequisites)TO courseList

ENDIF

END FORLOOP

CLOSE file

FOR each course IN courseList

POINT AT currentCourse

IF currentCourse prerequisiteList IS NOT empty

FOR each prerequisite IN prerequisiteList

FOR each course IN courseLists

IF prerequisite EQUALS course

VALID prerequisite NEXT prerequisite

ENDIF

END FORLOOP

IF no match found

PRINT “Invalid “prerequisite” not found in courses”

ENDIF

// Iterate over chain

WHILE currentCourse next IS NOT NULL

currentCourse EQUALS currentCourse next

IF currentCourse prerequisiteList IS NOT empty

FOR each prerequisite IN prerequisiteList

FOR each course IN courseLists

IF prerequisite EQUALS course

VALID prerequisite NEXT prerquisite

ENDIF

END FORLOOP

IF no match found

PRINT “Invalid “prerequisite” not found in courses

Add Course To courseList(courseID, courseName, prerequisites DEFAULT VALUE EQUALS null):

key EQUALS HASH courseID

current POINTS at courseList AT key

IF currentKey EQUALS maxValue //First item at this key’s index

courseList AT key -> course EQUALS course

courseList AT key -> courseID EQUALS courseID

courseList AT key -> key EQUALS key

courseList AT key -> prerequisites EQUALS prerequisites

RETURN

ENDIF

WHILE currentNext DOES NOT EQUAL NULL pointer //Find the end of chain

current EQUALS currentNext

END WHILE

//Point at new node

currentNext EQUALS NEW course(courseID, courseName, prerequisites, key)

Search and Print:

key EQUALS HASH courseID

currentCourse EQUALS POINTER to course AT courseList AT key

// If at the head of a chain

IF currentCourse courseID EQUALS search parameter

PRINT course information

FOR each prereq in prerequisites

PRINT prereq information

RETURN

ENDIF

WHILE currentCourse next is NOT null

currentCourse EQUALS currentCourse next

IF currentCourse courseID EQUALS search parameter

PRINT course information

FOR each prereq in perquisites

PRINT prereq information

RETURN

END IF

END WHILE

IF no match found

PRINT “No match found for ‘courseID’”

RETURN

**Binary Search Tree**

Structure Of Courses

STRING courseName

STRING courseID

VECTOR prerequisiteList

POINTER leftCourse

POINTER rightCourse

POINTER rootCouse EQUALS NULL

CONSTRUCTOR course(courseName, courseID, key)

THIS courseName EQUALS courseName

THIS courseID EQUALS courseID

THIS key EQUALS key

leftCourse EQUALS NULL

rightCourse EQUALS NULL

CONSTRUCTOR course(courseName, courseID, prerequisiteList, key)

THIS courseName EQUALS courseName

THIS courseID EQUALS courseID

THIS prerequisiteList EQUALS prerequisiteList

THIS key EQUALS key

Next EQUALS NULL POINTER

Read File:

TRY to open file IN file\_path

FOR each line IN file

SPLIT line by ‘,’ into tokensList

IF number in tokensList IS LESS than 2

PRINT Invalid file format

CLOSE file

RETURN

ENDIF

IF number in tokensList EQUALS 2

ADD course(courseName, courseID) TO courseList

ELSE if number in tokensList is GREATER THAN 2

FOR token starting at index 2 IN tokenLIST

ADD token to prerequisites

END FORLOOP

ADD course(courseID, courseName, prerequisites)TO courseList

ENDIF

END FORLOOP

CLOSE file

PrereqChecker(root)

PrereqChecker(course)

IF course IS NOT NULL

PrereqChecker(course->leftCourse)

FOR course->prereq IN prereqList

IF prereq NOT IN courses

PRINT “INVALID ‘prereq’ found”

RETURN

PrereqChecker(course->rightCourse)

addCourse

IF root EQUALS null

root EQUALS NEW course

ELSE insertCourse(root, newCourse)

insertCourse(course, newCourse)

IF newCourse.courseId is LESS THAN root->courseId

IF course->leftCourse EQUALS NULL

course->leftCourse EQUALS newCourse

ELSE

insertCourse(course->leftCourse, courseId)

ENDIF

ELSE (newCourse is greater than or equal)

IF course->rightCourse EQUALS NULL

course->rightCourse EQUALS newCourse

ELSE

insertCourse(course->rightCourse, newCourse)

ENDIF

ENDIF

PrintCourses(course) //Outputs courses in-order

IF course IS NOT null

PrintCourses(course->leftCourse)

PRINT course information

PrintCourses(course->rightCourses)

**MainMenu():**

PRINT menuOptions

GET userInput

IF userInput EQUALS “1”

readFile(filePath)

ELSE IF userInput EQUALS “2”

alphaNumeric(courseList)

ElSE IF userInputEQUALS “3”

GET courseID

searchAndPrint(courseID)

ELSE IF userInput EQUALS “9”

QUIT program

ELSE

PRINT “Invalid input please enter 1, 2, 3, or 9”

ENDIF

**Alphanumeric Order and Print**

**BTS:**

PrintCourses(course) //Outputs courses in-order, start from root

IF course IS NOT null

PrintCourses(course->leftCourse)

PRINT course information

PrintCourses(course->rightCourses)

**Vector:**

**Partition(courseList, lowIndex, highIndex):**

pivot EQUALS value at (highIndex + lowIndex) / 2

WHILE lowIndex is LESS than highIndex

WHILE value at lowIndex is LESS than pivot

INCREMENT lowIndex

WHILE value at highIndex is GREATER than pivot

DECREMENT highIndex

IF lowIndex is LESS than highIndex

SWAP value at lowIndex and value at highIndex

INCREMENT lowIndex

DECREMENT highIndex

RETURN highIndex

**Quick Sort(courseList, lowIndex, highIndex):**

IF lowIndex is GREATER than OR EQUAL to highIndex

Sorting complete

midPoint EQUALS CALL to Partition with values, lowIndex, highIndex

CALL to QuickSort with values, lowIndex, midPoint

CALL to QuickSort with values, midPoint+1, highIndex

**HashTable:**

**ExtractAndPrint(HashTable):**

VECTOR sortedList

FOR key IN HashTable

APPEND key TO sortedList

WHILE key->next IS NOT NULL

key EQUALS key->next

APPEND key TO sortedList

QuickSort(sortedList, 0, sortedList.size() -1)

DELETE sortedList

**Partition(sortedList, lowIndex, highIndex):**

pivot EQUALS value at (highIndex + lowIndex) / 2

WHILE lowIndex is LESS than highIndex

WHILE value at lowIndex is LESS than pivot

INCREMENT lowIndex

WHILE value at highIndex is GREATER than pivot

DECREMENT highIndex

IF lowIndex is LESS than highIndex

SWAP value at lowIndex and value at highIndex

INCREMENT lowIndex

DECREMENT highIndex

RETURN highIndex

**Quick Sort(courseList, lowIndex, highIndex):**

IF lowIndex is GREATER than OR EQUAL to highIndex

Sorting complete

FOR course IN courseList

PRINT course info

ENDIF

midPoint EQUALS CALL to Partition with values, lowIndex, highIndex

CALL to QuickSort with values, lowIndex, midPoint

CALL to QuickSort with values, midPoint+1, highIndex

**Big O complexity:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Vector** | **HashTable** | **BST** |
| ReadFile | O(n \* t) | O(n \* t) | O(n \* t) |
| AddCourse | O(n) | O(n \* m) | O(n \* log(n)) |
| PrereqChecker | O(n^2) | O(n) | O(n \* log(n)) |
| Alphanumeric | O(n) + O(n log(n)) | O(n) + O(n) + O(n log(n)) | O(n) |

**Analysis:**

ReadFile:

* The readFile() function loops n times
  + Each line is split into t tokens
    - If tokens are greater than two, then a prerequisite list is created, iterating t-2 times
  + The course is then added to the courseList via a function
* Thus, the function to represent the runtime complexity is:
  + O(n \* (t + (t – 2))) = O(n \* (2t – 2))
  + Simplified: O(n \* t) where n is several courses and t is the average number of tokens created per line.

AddCourse:

* Vector:
  + The course is added to the end of the vector, taking an average of O(1). This occurs n times, so O(n\*1) = O(n).
* HashTable:
  + Assuming a well-distributed hash table, each course has a key calculated O(1), and the average length of a chain is m. The big O notation is O(1\*n \* m)= O(n \*m).
* BST
  + The root pointer is first checked.
  + insertNode function is then recursively called til a null pointer is discovered, taking an average of O(log n). This occurs n times; therefore, the big O notation is O(n \* log(n)).

PrereqChecker:

* Vector:
  + The outer loop iterates over each course, running n times
  + The middle loop iterates over each prerequisite, iterating p times
  + The inner loop iterates over each course, running n times
  + O(n \*(p \* n) = O(n^2 \* p), where p is the average number of prerequisites per course.
  + Simplified to O(n^2)
* HashTable
  + The outer loop iterates over each course, running n times
  + Each prerequisite is then iterated over running p-times
  + The chain at the key is then checked running m times
  + O(n \* m \* p)
  + Simplified to O(n)
* BST
  + The outer loop iterates over each course, running n times
  + Each prerequisite is then iterated over running p-times
  + The average time for a search in a well-distributed BST is log n
  + O(n \* p \* log(n))
  + Simplified to O(n \* log(n))

AlphaNumericPrint:

* Vector:
  + The vector must first be sorted using a quicksort algorithm, which is linearithmic O(n log n)
  + Each course is then visited, printing the information O(n)
  + Thus, the total time is O(n + n log n)
* HashTable
  + The hash table must first be extracted and stored into a vector since, by nature, it may not be sorted. This has a complexity of O(n) as each node must be visited once.
  + The extracted table can then be quicksorted O(n log n)
  + Each course is then visited and printed O(n)
  + O(n + n log n + n)
* BST
  + Using recursive calls, each node is visited once in order, yielding a complexity of O(n).

**Conclusion:**

Vectors, hash tables, and binary search trees are valuable data structures with unique advantages and disadvantages. Vectors have the benefit of being the most straightforward syntax and, therefore, the easiest to implement. Each added object is appended to the end of the list, making list creation easy and efficient. Vectors also allow efficient algorithms such as quicksort, which uses recursive calls to sort data efficiently. However, vectors have disadvantages when it comes to search and comparisons. As each course must be visited for confidence, a course does not exist. This is evident in the PrereqChecker, as the list must be traversed in full twice.

Hash tables use a key to store links, also known as chains, in a vector. The key can be calculated for any course very quickly. The key is used as the index of the course in the vector with collisions adding a pointer to the next course. This pattern allows the easy search and retrieval of specific classes. This is shown by how dramatically faster the PrereqChecker is for hash tables than vectors. The disadvantages of hash tables include the inability to sort data quickly. As the key a course is assigned does not directly relate to its alphanumeric value, for example if the key was courseID % 10 and we had the entries 2, 12, 3, 4, 5. When visiting each list in indexed order, 12 would be visited before 3, 4, and 5. Thus, the hash table must be extracted to a vector, and then a quicksort could be applied to sort the courses. A hash table must be well-distributed. This means a key must be selected to spread the courses over the table, or each course may end in a single chain at one key.

Binary search Trees (BST) start at a root node and point to a left and right node. Courses get sorted to the right if they are greater than the root and to the left if they are lesser than. This pattern allows recursive calls to functions to insert, remove, or search for courses. The somewhat organized nature of a BST makes this functionality extremely efficient. However, BSTs may devolve to simply a linked list if courses are added in ascending/descending order. Thus, a root node near the median of the data set must be selected to ensure the integrity of the BST.

**Recommendation:**

Based on the Big O runtime and my analysis, I have selected a binary search tree as the data structure. All data structures have a similar runtime for creating the list. Hash tables are more efficient at checking the prerequisites but suffer when wishing to print the courses alphanumerically, taking considerable time and memory to accomplish these goals. Vectors are straightforward during creation and are efficient at printing the list alphanumerically. However, the list must be iterated several times when checking the prerequisite requirement, taking significantly longer than the hash table or the BST. The BST allows for efficient list creation. Iterating over the list to find a single value is recursive and cuts the search field in half for each iteration, making it highly efficient. A BST requires no sorting when wishing to print the list alphanumerically, making it highly efficient, and it only takes the time to visit each course to accomplish this goal. For these reasons, a BST is the ideal data structure for the course listings.