6-2 Project 1

Alex Kastigar

8/10/25

CS-300-12213-M01 DSA: Analysis and Design

6-2 Project 1 All Pseudocode

ABC University Course Info System Pseudocode

**PART 1**

**A. Vector**

1. Load and Check Data From File

// try to open the input file

open inputFile

if file didn’t open

 print “could not open file”

 exit program

// set up a vector to hold Course objects

vector<Course> courses

// go through each line in the file

while there’s a line

 split the line by commas into parts

 if parts has less than 2 items

  print “line is missing stuff”

  move to next line

 // make a Course object from the line

 course.courseNumber = parts[0]

 course.courseName = parts[1]

 // check for prerequisites and add them

 for i from 2 to parts.length - 1

  course.prerequisites.push\_back(parts[i])

 add course to courses vector

close inputFile

// now double check prereqs

for each course in courses

 for each prereq in course.prerequisites

  see if a course in the vector has that courseNumber

  if not

   print “prerequisite not found: ” + prereq

2. Create Course Object and Store in Vector

// course struct to hold info

struct Course {

 string courseNumber

 string courseName

 vector<string> prerequisites

}

// Course objects are made and added during the file loop

3. Search and Print Course Info

// ask user what course they want

print “enter course number”

input searchCourse

// go through the vector

for each course in courses

 if course.courseNumber equals searchCourse

  print course.courseNumber and course.courseName

  if course.prerequisites is empty

   print “no prerequisites”

  else

   print “prerequisites:”

   for each prereq

    print prereq

**B. Hash Table**

1. Load and Check Data From File

// try to open the input file

open inputFile

if file didn’t open

 print “could not open file”

 exit program

// set up a hashTable to hold Course objects

hashTable<Course> courseTable

// also make a vector of courseNumbers for prereq validation

vector<string> courseKeys

// go through each line in the file

while there’s a line

 split the line by commas into parts

 if parts has less than 2 items

  print “line is missing data”

  move to next line

 // make a Course object from the line

 course.courseNumber = parts[0]

 course.courseName = parts[1]

 // check for prerequisites and add them

 for i from 2 to parts.length - 1

  course.prerequisites.push\_back(parts[i])

 // insert course into the hashTable using courseNumber as key

 courseTable.insert(course.courseNumber, course)

 // store courseNumber in courseKeys for prereq check

 courseKeys.push\_back(course.courseNumber)

close inputFile

// now double check prereqs

for each course in courseTable

 for each prereq in course.prerequisites

  if prereq not in courseKeys

   print “prerequisite not found: ” + prereq

2. Create Course Object and Insert into Hash Table

// course struct to hold info

struct Course {

 string courseNumber

 string courseName

 vector<string> prerequisites

}

// Course objects are created during the file loop

// each one is inserted into the hashTable using courseNumber as the key

3. Search and Print Course Info

// ask user what course they want

print “enter course number”

input searchCourse

// search the hashTable for that course

course = courseTable.search(searchCourse)

if course not found

 print “course not found”

else

 print course.courseNumber and course.courseName

 if course.prerequisites is empty

  print “no prerequisites”

 else

  print “prerequisites:”

  for each prereq

   print prereq

**C. Tree**

1. Load and Check Data From File

// try to open the input file  
open inputFile  
if file didn’t open  
 print “could not open file”  
 exit program

// set up a binarySearchTree to hold course objects  
binarySearchTree<Course> courseTree  
// also set up a vector of courseNumbers to check prereqs later  
vector<string> courseKeys

// read through each line of the file  
while there’s a line  
 split the line by commas into parts

 if parts has less than 2 items  
  print “line is missing data”  
  continue to next line

 // build the course object  
 course.courseNumber = parts[0]  
 course.courseName = parts[1]

 // loop through any prerequisites  
 for i from 2 to parts.length - 1  
  course.prerequisites.push\_back(parts[i])

 // insert the course into the tree  
 courseTree.insert(course)  
 // keep track of all course numbers for prereq validation  
 courseKeys.push\_back(course.courseNumber)

close inputFile

// check that all prerequisites actually exist  
for each course in courseTree (inorder traversal)  
 for each prereq in course.prerequisites  
  if prereq not found in courseKeys  
   print “missing prerequisite: ” + prereq

2. Create Course Object and Insert into Binary Search Tree

// define a course struct  
struct Course {  
 string courseNumber  
 string courseName  
 vector<string> prerequisites  
}

// each course is created while reading the file  
// and added to the binarySearchTree based on courseNumber

3. Search and Print Course Info

// ask user for a course number

print “enter course number”

input searchCourse

// search the tree for the course

course = courseTree.search(searchCourse)

if course not found

 print “course not found”

else

 print course.courseNumber + “: ” + course.courseName

 if course.prerequisites is empty

  print “no prerequisites”

 else

  print “prerequisites:”

  for each prereq in course.prerequisites

   print prereq

**PART 2**

**D. Menu**

// pick a data structure flavor: "vector", "hash", or "tree"

// choose one flavor and leave it:

// vector flavor: loadDataVector / printCourseListVector / printCourseInfoVector

// hash flavor: loadDataHashTable / printCourseListHash / printCourseInfoHash

// tree flavor: loadDataTree / printCourseListTree / printCourseInfoTree

// tiny wrappers so I only change 3 lines when flavor is picked

function loadData(filePath)

return loadDataTree(filePath) // <— insert chosen flavor here

function printCourseList()

return printCourseListTree() // <— insert chosen flavor here

function printCourseInfo(courseNumber)

return printCourseInfoTree(courseNumber) // <— insert chosen flavor here

bool isRunning = true

bool dataLoaded = false

string inputFilePath = "courses.txt"

while isRunning

print ""

print "1. Load Data"

print "2. Print Course List (alphanumeric)"

print "3. Print Course Info (title and prerequisites)"

print "9. Exit"

print "Choose an option:"

input choice

if choice == 1

loadData(inputFilePath)

dataLoaded = true

print "data loaded"

else if choice == 2

if dataLoaded == false

print "please load data first"

else

printCourseList()

else if choice == 3

if dataLoaded == false

print "please load data first"

else

print "enter course number:"

input queryCourse

printCourseInfo(queryCourse)

else if choice == 9

isRunning = false

print "goodbye"

else

print "invalid option"

**PART 3**

**E. Course List in Alphanumeric Order**

// vector option: sort by courseNumber, then print

function printCourseListVector()

sort courses by courseNumber ascending

for each course in courses

print course.courseNumber + “: ” + course.courseName

// hash table option: gather then sort, then print

function printCourseListHash()

vector<Course> allCourses

for each key in courseTable

allCourses.push\_back(courseTable.get(key))

sort allCourses by courseNumber ascending

for each course in allCourses

print course.courseNumber + “: ” + course.courseName

// tree option: inorder traversal yields sorted order

function printCourseListTree()

inorder(courseTree.root)

function inorder(node)

if node is null

return

inorder(node.left)

print node.course.courseNumber + “: ” + node.course.courseName

inorder(node.right)

**PART 4**

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Vector | Hash Table | Tree (BST) |
| Read File & Create Objects | O(n) | O(n^2) worst,  O(n) avg | O(n^2) worst,  O(n log n) avg |
| Validate Prerequisites (p total) | O(p·n) | O(p·n) worst,  O(p) avg | O(p·n) worst,  O(p log n) avg |
| Print Full Sorted List | O(n log n) | O(n log n) | O(n) |
| Lookup Single Course | O(n) | O(n) worst,  O(1) avg | O(n) worst,  O(log n) avg |
| Memory | O(n) | O(n + buckets) | O(n) |

This is my worst-case Big-O report for the parts that matter here: reading the file, building course objects, printing the sorted list, and finding one course. I’m dealing with n total courses and p total prerequisites in the file (so p could be zero up to around n\*k). I count a simple statement as cost 1, and a function call costs whatever that function uses. I did mix up Theta once in my notes, but I’m sticking to Big-O for worst-case here.

For loading, the vector version loops over the lines and push\_back is constant on average, so that stays around O(n). The hash table also goes through all lines; insert is average O(1) but can blow up to O(n) in the worst collision case, so worst-case over n inserts is O(n^2) (average close to O(n)). A basic BST insert is O(log n) average but can reach O(n) if the BST gets severely unbalanced; over all n inserts that’s O(n log n) average and O(n^2) worst. For prerequisite checking, the vector approach searches to find each prereq, so O(p·n). The hash table can check each prereq in O(1) average, so O(p) average, but nonetheless O(p·n) worst if the hash is bad. The BST search is O(log n) average and O(n) worst, so O(p log n) average and O(p·n) worst overall.

For alphabetizing the list, the vector needs a sort (O(n log n)) and then another pass to output, so O(n log n). The hash table needs to gather into a temporary list and sort, so still O(n log n). The tree gets this round because an inorder traversal is already sorted, so it outputs in O(n). For a single lookup, the vector is O(n) linear search, the hash table is O(1) average (O(n) worst), and the BST is O(log n) average (O(n) worst). Memory usage: vector is O(n) contiguous storage, the hash table is approximately O(n + buckets) with chaining nodes, and the BST is O(n) nodes plus pointers. Small caveat assuming separate chaining with a good load factor; when buckets are m, space is O(n + m). Also, I am mindful insertion order can make the BST perform like a linked list worst case; we have not employed a self-balancing tree in this case.

**PART 5**

Vector is simple, easy to iterate over, and std::sort is good for the list (O(n log n)), but searching is O(n) and my prereq check becomes O(p·n) since I scan. Hash table is efficient for lookups in constant time on average and prereq check is cheap, but I still have to gather + sort to print the list, and there is extra memory cost and the worst-case theoretical collisions. BST gives me sorted list for free with inorder (O(n)) and good average lookups (O(log n)), but it can be bad if it gets unbalanced.

**PART 6**

I recommend the tree. The two big advisor asks are “print everything sorted” and “show one course with prereqs.” The BST gives me O(n) printing without a separate sort and keeps lookups reasonable on average. Hash table would also be a strong pick for option 3, but since option 2 (the sorted list) is always required, the tree keeps the code and the outputs straightforward. This exercise permitted me to actually contrast O(1), O(log n), O(n), O(n log n), and O(n^2) hands-on, not on a chart.