Southern New Hampshire University

CS 300 Pseudocode Document

Function Signatures

```
//Data structures

Struct Course {
      String courseNumber
      String coursename
      Vector<string> prerequisites
}

Class BinarySearchTree {
Node root
Course course

}

Class HashTable {
Struct Node
Course course
int key
Node* next
Unsigned int tableSize
      Vector of nodes
      Print funciton
Hash function
}


//Hash Table
void loadCourses(HashTable<Course>& courses, string filename){
Open filename
If file not opened
print "error: file not found"
exit
For each line in the file
      Vector<string> information //reset each time
      Split line into strings at each ','
      Store each string of the line in information
            If information vector size less than two
            Error, file format incorrect
            else
                  Create new Course called course
                  CourseNumber = information vector at zero
                  CourseName = information vector at one
                  For the rest of the strings in information
                        Add to prerequisite vector

                  For all prerequisites in prerequisite vector
                  If prerequisite does not exist in courses vector
```

```
                Print "error: prerequisite DNE"
                Exit

                Insert course into HashTable where key is equal to the
                courseNumber

Close filename

      Return courses
}


Void PrintAll() {
Sort by courseNumber
For all courses beginning to end
if key is not default value
                print courseNumber
                print courseName
                For all prerequisites of a course
                      Print prerequisites
            Set node to next if there is a chain
                      Print info throughout chain

}
void searchCourse(HashTable<Course> courses, String courseNumber) {
Create key using course Id

      If entry found at key

            return node at key's course

      If node at key is empty or default value

            return empty course

      While node is not null

            if node key not default value and course id matches

                  return node's course

            node set to node next

      If nothing found return empty course


}

Void insertCourse(Course course) {
      Create key using courseNumber
      Create curNode which equals the node at key
      If curNode is empty
      Create new node with course set to course
      Set curNode to new node
```

```
      Else if curNode not empty
      If curNode key is defauly val
      overwrite defaul val with new vals
      Else
                  iterate over chain until next open node
                  When cur node is at the end, set next to new
                  node

}

Unsigned int Hash(int key) {
Key % tableSize;
}


//Binary Tree
Load functions:
void loadCourses(BinarySearchTree<Course>& courses, string filename){
Open filename
If file not opened
print "error: file not found"
exit
For each line in the file
      Vector<string> information //reset each time
      Split line into strings at each ','
      Store each string of the line in information
            If information vector size less than two
            Error, file format incorrect
            else
                  Create new Course called course
                  CourseNumber = information vector at zero
                  CourseName = information vector at one
                  For the rest of the strings in information
                        Add to prerequisite vector

                  For all prerequisites in prerequisite vector
                  If prerequisite does not exist in courses vector
                  Print "error: prerequisite DNE"
                  Exit

                  Insert course into courses tree
Close filename

      Return courses
}


Other functions:
Void PrintAll(Node* node) {
If node not nullptr
recursive call with nodes left pointer
```

```
output node courseNumber, name, and prerequisites
recursive call with nodes right pointer

}
String searchCourse(BinarySearchTree<Course> courses, String
courseNumber) {
Set current node to root
While current node does not e qualify null pointer
if the current courseNumber matches
return current node course
else  if courseNumber is smaller then current nodes courseNumber
traverse left
else
traverse right
return the courseNumber

}

Void insertCourse(Course course) {
  If root is equal to null pointer
      Then root will equal new node set to bid
  Else
Add the Node
}

Void addNode(Node node, Course course) {

If Node isnt null and the node courseNumber is greater than comparison

If the left Node is null

Then new node will go on left

return

     Else

          Use recursive addNode to node's left


  Else if Node is not null and nodes courseNumber is less then
comparison

If the right is null

Then new node will go on right

          Return
     Else

          Use recursive addNode to node's right

}
```

```
//Vector

 loadCourses(Vector<Course>& courses, string filename){
Open filename
If file not opened
print "error: file not found"
exit
For each line in the file
      Vector<string> information //reset each time
      Split line into strings at each ','
      Store each string of the line in information
            If information vector size less than two
            Error, file format incorrect
            else
                  Create new Course called course
                  CourseNumber = information vector at zero
                  CourseName = information vector at one
                  For the rest of the strings in information
                        Add to prerequisite vector

                  For all prerequisites in prerequisite vector
                  If prerequisite does not exist in courses vector
                  Print "error: prerequisite DNE"
                  exit

                  Add course to the end of the courses vector
Close filename

}
void searchCourse(Vector<Course> courses, String courseNumber) {
      for all courses
            if the course is the same as courseNumber
                  print out the course information
                  for each prerequisite of the course
                        print the prerequisite course information
}

Void displayCourses(Vector<Course> courses, String courseNumber) {
Sort by courseNumber
For all courses
print courseNumber
print courseName
            For all prerequisites of a course
                  Print prerequisites

}
```

//Main Menu

```
While input not nine
      Print MainMenu
      Print 1: Load Data Structure
      Print 2: Print Course List (alphabetical)
      Print 3: Print Course (byID)
      Print 9: Exit

      Switch statement
            case 1
            LoadCourses(courses, filename)

      Case 2
            If vector
            SortCourses by CourseID
            displayCourse(CoursesVector)
            If Hashtable
            CourseHashtable.PrintAll()
                  else if BinarySearchTree
                        courseBST.PrintAll(coursesBST.root)
      Case 3
      SearchCourse()
      Case 9
                  print Leaving
      exit
```

Runtime Analysis

Below is a runtime analysis for loading courses into each data structure.

Vector: Constant is denoted by k (prereq, splits in data line)

Code	Line Cost	# Times Executes	Total Cost
Open filename	1	1	1
if file not opened error	1	1	1
for each line in the file	1	n	n
Vector information	1	n	n
Split line into string at ,	k	n	kn
Store each string in the line in info	k	n	kn

`If info vector <2 return error`	1	n	n
`Create new course set info at 0. to number, at 1 to name`	1	n	n
`For rest of strings in vector information`	1	n	n
`Store each in preqs`	k	n	kn
`For all preqs in preqs vector`	1	n	n
`If preq does not exist in courses print error`	1	n	n
`Push back course into courses vector`	1	n	n
`Close filename`	1	1	1
		Total Cost	3kn + 8n + 3
		Runtime	O(n)

Hash Table: Constant is denoted by k (prereq, splits in data line)

Code	Line Cost	# Times Executes	Total Cost
`Open filename`	1	1	1
`if file not opened error`	1	1	1
`for each line in the file`	1	n	n
`Vector information`	1	n	n
`Split line into string at ,`	k	n	kn
`Store each string in the line in info`	k	n	kn

`If info vector <2 return error`	1	n	n
`Create new course set info at 0. to number, at 1 to name`	1	n	n
`For rest of strings in vector information`	1	n	n
`Store each in preqs`	k	n	kn
`For all preqs in preqs vector`	1	n	n
`If preq does not exist in courses print error`	1	n	n
`Insert into hash table`	1	n	n
`Close filename`	1	1	1
		Total Cost	3kn + 8n + 3
		Runtime	O(n)

Binary Search Tree: Constant is denoted by k (prereq, splits in data line)

Code	Line Cost	# Times Executes	Total Cost
`Open filename`	1	1	1
`if file not opened error`	1	1	1
`for each line in the file`	1	n	n

`Vector information`	1	n	n
`Split line into string at ,`	k	n	kn
`Store each string in the line in info`	k	n	kn
`If info vector <2 return error`	1	n	n
`Create new course set info at 0. to number, at 1 to name`	1	n	n
`For rest of strings in vector information`	1	n	n
`Store each in preqs`	k	n	kn
`For all preqs in preqs vector`	1	n	n
`If preq does not exist in courses print error`	1	n	n
`Insert into binary tree`	Log n	n	nlogn
`Close filename`	1	1	1
		Total Cost	7n + 3kn + nlogn + 3
		Runtime	O(nlogn)

Chosen Data Structure:

When choosing data structures, you must have a balance in efficiency and organization. Binary Search Tree proves to be the best option, maintaining the best balance of efficiency. It allows for quick searches by course ID while also performing in-order traversal to display courses alphabetically. While Hash tables offer efficient searching, they struggle when it comes to displaying

in alphabetical order due to not maintaining natural order. Vectors require manual searching and sorting, so despite the easiness of loading courses into the vector, it becomes difficult when working with the vector itself. I believe the binary search tree outweighs the other two data structures, even if it may not appear to be so grand in the load run time analysis, it makes up for it with its ordered structure and ease of search/display.