# CS 300 Pseudocode Document

Demetri Pighetti

10/17/2024

CS 300

//Vector - Milestone 1

pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors.  
Create a file object ( ifstream)

Open the file with that file object

Set up an array called courseCheck to store course data

While the file isn't at the end:

* Read a line from the file
* Split the line
* Add the first part of the line into the courseCheck array

While there are still lines left in the file:

* Grab the next line from the file
* Split the line and store the data in an array courseInfo
* If the line doesn't have at least 2 pieces of info:
  + Print "Incorrect file format" and stop processing that line

For each part of the line (after the first two pieces):

* If the part isn’t found in the courseCheck array:
  + Print "Non-existent Prerequisite"

### Design pseudocode to show how to create course objects and store them in the appropriate data structure. Create Course Class

* Define courseNumber as a string
* Define courseName as a string

### Set up Vectors

* Make a prerequisites vector (list of strings)
* Make a courses vector (list of Course objects)

### Read File

* Open the file for reading

### While there’s still stuff to read in the file:

1. Get the next line
2. Split the line by commas
3. Create a new Course object
4. Assign the first part of the line to courseNumber
5. Assign the second part of the line to courseName
6. For each remaining part of the line:
   * Add that part to the Course's prerequisites vector
7. Add the new Course object to the courses vector

Pseudocode that will search the data structure for a specific course and print out course information and prerequisites.

### void printCourseInformation(Vector<Course> courses, String courseNumber)

* For each course in the list:
* If the course’s courseNumber matches the courseNumber passed in:
  + Print the course number and name
  + For each prerequisite the course has:
    - Print the prerequisite’s courseNumber

}

//Hash Table - Milestone 2

1. Read Command Line Arguments:
   * Extract arguments, save one as the CSV file path.
   * If no arguments are provided, load a default CSV file.
2. While User's Choice is Not '9':
   * Show the menu options to the user.
   * Take input from the user and save it in the choice variable.
   * Check if the input is valid.
     + If the input isn't between 1-4 or 9, raise an error.
3. If User Chooses '1':
   * Start the timer.
   * Load the data from the CSV file into the hash table bidTable.
   * Display how many entries were loaded from the CSV.
   * Stop the timer and display how long it took to load the data.
4. If User Chooses '2':
   * Call the PrintAll() function on bidTable to display all entries.
5. If User Chooses '3':
   * Start the timer.
   * Use the Search() function, passing a key to locate a specific bid.
   * Stop the timer and display how long the search took.
6. If User Chooses '4':
   * Call Remove() to delete a bid using the given key.
7. If User Chooses '9':
   * Exit the program and display a goodbye message.

### 

### HashTable Methods

1. HashTable::hash(int):
   * Return the result of key % tableSize.
2. HashTable::Insert(Bid):
   * Use the hash() function with the bid's bidId to compute tempKey.
   * Check if the computed hash index is available.
   * If the index is unused, create a new node for the bid and insert it at that location.
   * If the index is already occupied, navigate through the linked list at that index and append the new node at the end.
3. HashTable::PrintAll():
   * Initialize a pointer to traverse the list.
   * For each node, if the key is valid (i.e., not UINT\_MAX), display the bidId, title, amount, and fund.
4. HashTable::Search(String):
   * Create a pointer cursor and point it to the hash location.
   * Traverse the list until you find a node where the bidId matches the search string.
   * If found, return the node; if not, continue to the next node.
5. HashTable::Remove(String):
   * Set cursor to point to the hash location.
   * If a collision chain exists, check each node.
   * If a match is found, remove the node by adjusting pointers and deleting the node.

//Binary Search Tree – Milestone 3

Struct Course {

* courseID
* courseName
* preCount
* preList

Course() (constructor):

* courseID = ""
* courseName = ""
* preCount = 0
* preList = ""  
  }

Class BinaryTree {

* Struct Node
  + Course
  + right pointer
  + left pointer
* Private member: root
* Public methods:
  + printCourse()
  + BinaryTree()

Main()

* Create a new BinaryTree called courseTree of the Course type
* Get the CSV file path from the user
  + If no file path is provided, use the default location
* Call txtParser() with the CSV file path
* Call validateList() with courseTree
* Get user input for the value to search, and store it in userSearch
* Call printCourse() passing userSearch

txtParser(String)

* Open the file using the path in String, utilizing parser libraries
* Loop through the file row by row until the end of the file (EOF)
  + If both the first and second strings are present:
    - Add the first string to the struct as courseID
    - Add the second string to the struct as courseName
    - Continue looping until no value is found in a column (indicating no more prerequisites)
      * Increment a variable called preCount for every prerequisite found
      * Concatenate each prerequisite name to a local string called preNames
    - Add preCount to the struct as preCount
    - Add preNames to the struct as preList
* Return tempList

searchList(String)

* Create tempCourse of type Node
* Set tempCourse to the bucket at the hash location of String
* Loop through the list for each course:
  + If String matches courseID:
    - Set tempCourse to that Course
* Return tempCourse

printCourse(String)

* Create tempCourse of type bucket
* Set tempCourse to the root
* Loop while tempCourse is not null:
  + If the node at tempCourse has a bidId equal to String:
    - Output courseID and courseName of the course found in tempCourse
  + Loop through preList:
    - For each course in preList, call printCourse() passing preList
  + If courseID is less than String, move left
  + If courseID is greater than String, move right

validateList()

* Create tempCourse of type Node
* Create a variable valid and set it to true
* For each course:
  + If valid is false, break
  + While tempCourse's next is not null:
    - Loop through preCount:
      * Set tempCourse to the result of searchList(preList token)
      * If courseID is empty, set valid to false
* Return valid

End

**Runtime Analysis**  
  
We’re looking at three options: a vector, a hash table, and a binary search tree (BST). Vectors are fast for adding data but slow when searching or sorting. Hash tables are good for quick lookups and inserts but don't keep things sorted. BSTs are a nice balance because they keep the data sorted while still offering decent performance for searching and inserting.

When it comes to basic functions like showing the menu, searching for a course, or printing the catalog, all three data structures perform similarly. But when it comes to more complex tasks like loading data, searching, or sorting, the differences are more apparent. Vectors seem to struggle with searching and sorting, hash tables are fast unless you need sorting, and BSTs give solid performance overall.

**Recommendation:** Based on the analysis, I recommend using a binary search tree. It handles the key operations efficiently, especially since we need sorted data. If sorting wasn’t an issue, a hash table could be better. Vectors are the weakest option for this project.

| Application functions independent of data structure implementation | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| Average case | Display Menu() | Parse() | Display course() | Display catalog() | Exit() | Total |
| Time Complexity | 4 | O(n) | O(n) | O(n) | 1 | O(n) |
| Space Complexity | 4 | O(n) | 2 | 3 | 1 | O(n) |

| Application functions dependent on data structure implementation | | | | |
| --- | --- | --- | --- | --- |
| Average case | Load() | Search() | Sort() | Total |
| Time Complexity |  |  |  |  |
| Vector | O(1) | O(n) | O(n\*logn) | O(n) |
| Hash Table | O(1) | O(1) | O(n^2) | O(n^2) |
| Binary Search Tree | O(logn) | O(logn) | 1 | O(logn) |
| Space Complexity |  |  |  |  |
| Vector | O(1) | O(1) | O(logn) | O(logn) |
| Hash Table | O(n) | O(n) | O(n+logn) | O(n) |
| Binary Search Tree | O(n) | O(n) | 1 | O(n) |