Project 1

Kenneth Pinkerton

CS-300-11419-M01

**Pseudocode**

1. struct Course{}
   * courseId
   * CourseName
   * preCourseCount
   * preCourseName
   * Course() (constructor) {courseId = “”; courseName = “”; preCourseId = 0; preCourseName = “”}
2. Class HashTable {}
   * -Struct bucket
     1. Course
     2. Key
     3. Next pointer
   * + hash()
   * + printAll()
   * + List<> hashTable
3. Class BinarySearchTree {}
   * -Struct Node
     1. Course
     2. Left pointer
     3. Right pointer
   * - root
   * + printCourse()
   * + BinarySearchTree()
4. txtParser(String)
   * Create local list named tempList
   * Open file at location provided in String
   * Loop through all items in file until none are left
     1. If item has 1st and 2nd strings
        1. Add 1st string to courseId
        2. Add 2nd string to courseName
        3. Loop until no more items in row
           1. Increment preCourseCount for each prerequisite found
           2. Concatenate a string for all prerequisites and assign them to preCourseName
        4. Add preCourseCount
        5. Add preCourseName
   * Return tempList
5. sortList()
   * Get vector to be sorted, lowest and highest index of the vector
   * Set targetIndex to value returned from partition()
   * Recursively call quicksort passing the vector, lowest index and the targetIndex
   * Recursively call quicksort passing the vector, targetIndex, and the highest index
6. Partition()
   * Determine the vector element at the midpoint between the lowest and highest index values
   * Set pivot to the midpoint of the vector
   * Loop until the lowest index value is greater than or equal to the highest index value
     1. Loop through the vector from the lowest index value until a larger element than the pivot is found
     2. Replace the lowest index with this element
     3. Loop through the vector from the lowest index until a smaller element than the pivot is found
     4. Preplace the highest index with this element
     5. Swap the elements at the new highest and lowest indeces
     6. Overwrite the lowest index by incrementing it by 1
     7. Overwrite the highest index by incrementing it by 1
   * Return the highest index
7. searchList(String)
   * Create tempCourse of type Course
   * Loop through list for each Course
     1. If String matches courseId
        1. Set tempCourse to Course
   * Return tempCourse
8. printCourse(String)
   * Create tempCourse of type Course
   * Set tempCourse equal to searchList(String)
   * Output courseId to screen
   * Output courseName to screen
   * Loop through prerequisites from 0 to preCourseCount
     1. For each Course in preCourseName
        1. printCourse(preCourseName)
9. printList()
   * Loop through courseList
   * Output courseId to screen
   * Output courseName to screen
   * Loop through prerequisites from 0 to preCourseCount
     1. For each Course in preCourseName
        1. printCourse(preCourseName)
10. printTree()
    * Create new Node pointer “root”
    * Set root to NULL
    * Check if Node is NULL, return if yes
    * Call recursively Node’s left pointer to find the left most Node
      1. Output courseId to screen
      2. Output courseName to screen
      3. Loop through prerequisites from 0 to preCourseCount
         1. For each Course in preCourseName
            1. printCourse(preCourseName)
    * Call recursively Node’s right pointer to find the right most Node
      1. Output courseId to screen
      2. Output courseName to screen
      3. Loop through prerequisites from 0 to preCourseCount
         1. For each Course in preCourseName
            1. printCourse(preCourseName)
11. printTable()
    * Create new Node pointer and Set to the head
    * Loop through Table starting at Head
      1. Output courseId to screen
      2. Output courseName to screen
      3. Loop through prerequisites from 0 to preCourseCount
         1. For each Course in preCourseName
            1. printCourse(preCourseName)
12. validateList()
    * Create tempCourse of type bucket
    * Create boolean variable valid and Set to True
    * For each Course
      1. If valid is False break
      2. While tempCourse next is not null
         1. Loop through prerequisites from 0 to preCourseCount
            1. Set tempCourse equal to searchList(preCourseName)
            2. If tempCourseiD is empty Set valid to False
         2. Return valid
13. Main function() //Program driver
    * Create new List named courseList of the given struct
    * Loop while choice is not equal to ‘9’ //exit command
      1. Output menu options
      2. Get user input, Store in menuChoice
      3. Validate user Input
         1. If choice is not 1-4 or 9 throw error and prompt another entry
      4. If choice is ‘1’ //Load data from file and call file parser
         1. Get CSV file path from user
            1. If no path given, use default location
         2. Call txtParser() passing CSV file path
         3. Call validateList() passing courseList
      5. If choice is ‘2’ //Print ordered list alphanumerically
         1. Get user value to search for and Store in userSearch
         2. Call printCourse() passing userSearch
      6. If choice is ‘3’ //Print the course title and prerequisites for a single course
         1. Call printList()/printTree()/printTable()
      7. If choice is ‘9’ //Exit program

**Run Time Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| **Loading** | O(1) | O(1) – O(N) | O(log N) |
| **Search** | O(N) | O(1) – O(N) | O(log N) – O(N) |
| **Sort/Print** | O(N log N) | O(N) | O(N) |

**Analysis breakdown**

Using a vector based approach can be fastest at first, but slowest when it comes to sorting this information as there frequent shifts needed in order to accomplish this.

Hash Tables can offer a decent middle ground as there loading an searching are both relatively quick, this can vary though depending on if there are potential collisions that may occur. In this example there shouldn’t be collisions or duplicate classes as long as the initial data is cleaned properly and it is read into the program in a uniform manner that does not need additional steps like removing or adding blank space that could cause errors or possible collisions or other erratic behavior.

Binary trees have some of the longer load times when the data is first being entered into the system, but because of the number of courses that may be present in this dataset the times should be similar to Hash tables and be complete in under a few seconds. The binary tree would have one of if not the fastest sort/print times as this is done during the loading of the data.

As long as data would not need to be continually reloaded and would written/stored in a permanent file, Hash Tables an Binary Trees both have favorable execution times for searching and printing. Hash tables would edge out ahead though in terms of performance as the chance of collisions I feel in this project are low and it’s times would likely be close to constant time for searching and loading, since sorting and printing are even for both Hash Tables should be used to maintain system resources and memory.