**Pseudo Code for Menu**

Int choice = 0;

While (choice is not 4){

cout << "Menu:" << endl;

cout << " 1. Load Courses" << endl;

cout << " 2. Display List of Courses" << endl;

cout << " 3. Find Course" << endl;

cout << " 4. Exit" << endl;

cout << "Enter choice: ";

cin >> choice;

switch (choice) {

case 1:

call load courses function

break;

case 2:

call print all courses function

break;

case 3:

prompt user for course num

call search function for given course num

if (course exists){

print course

}

else {

tell user course does not exist

}

break;

}

}

**Run time analysis for vector**

|  |  |  |  |
| --- | --- | --- | --- |
| code | Line cost | # Times executed | Total cost |
| Vector<string> courseInfo; | 1 | 1 | 1 |
| Vector<string> courseNum; | 1 | 1 | 1 |
| Vector<string> courseName; | 1 | 1 | 1 |
| Vector<string> coursePrerequisites; | 1 | 1 | 1 |
| i=0; | 1 | 1 | 1 |
| j=0; | 1 | 1 | 1 |
| Ifstream cInfo; | 1 | 1 | 1 |
| if (file not open){ | 1 | 1 | 1 |
| cout << “ could not open file”; | 1 | 1 | 1 |
| While (not end of file) { | 1 | n | n |
| Split every line into strings at every comma; | 1 | n | n |
| Add every string to courseInfo vector; | 1 | n | n |
| If (courseInfo.size() < 2){ | 1 | n | n |
| Cout<< “info missing from file” | 1 | n | n |
| CourseNum.push\_back(courseInfo.at(0)) | 1 | n | n |
| CourseName.push\_back(courseInfo.at(1)) | 1 | n | n |
| If (courseInfo.size() > 2){ | 1 | n | n |
| For (i = 2; i < courseInfo.size(); ++i) { | 1 | n(n) | N^2 |
| coursePrerequisites.push\_back(courseInfo.at(i)) | 1 | Constant  (c) | c |
|  |  | **Total cost** | 8n+cn^2 |
|  |  | **runtime** | O(n^2) |

**Run time analysis for hash table**

|  |  |  |  |
| --- | --- | --- | --- |
| code | Line cost | # Times executed | Total cost |
| Vector<string> courseInfo; | 1 | 1 | 1 |
| Vector<string> confirmCoursePrerequisites; | 1 | 1 | 1 |
| Vector<string> confirmcourseNum; | 1 | 1 | 1 |
| i=0; | 1 | 1 | 1 |
| j=0; | 1 | 1 | 1 |
| Ifstream cInfo; | 1 | 1 | 1 |
| cout << “ could not open file”; | 1 | 1 | 1 |
| While (not end of file) { | 1 | n | n |
| Split every line into strings at every comma; | 1 | n | n |
| Add every string to courseInfo vector; | 1 | n | n |
| If (courseInfo.size() < 2){ | 1 | n | n |
| Cout<< “info missing from file” | 1 | 1 | 1 |
| confirmCourseNum.push\_back(courseInfo.at(0)) | 1 | n | n |
| Course course; | 1 | n | n |
| course.num = courseInfo.at(0) | 1 | n | n |
| CourseName.push\_back(courseInfo.at(1)) | 1 | n | n |
| If (courseInfo.size() > 2){ | 1 | n | n |
| For (i = 2; i < courseInfo.size(); ++i) { | 1 | n(n) | N^2 |
| Course.prerequisites = courseInfo.at(i) + “ , ” | 1 | Constant c | c |
| ConfirmCoursePrerequisites.push\_back(courseInfo.at(i)) | 1 | Constant c | c |
| hashTable->Insert(course); | 1 | n | n |
|  |  | **Total cost** | 12n+2cn^2 |
|  |  | **runtime** | O(n^2) |

**Run time analysis for binary tree**

|  |  |  |  |
| --- | --- | --- | --- |
| code | Line cost | # Times executed | Total cost |
| Vector<string> courseInfo; | 1 | 1 | 1 |
| Vector<string> confirmCoursePrerequisites; | 1 | 1 | 1 |
| Vector<string> confirmcourseNum; | 1 | 1 | 1 |
| Int i = 0; | 1 | 1 | 1 |
| Int j = 0; | 1 | 1 | 1 |
| Int k = 0; | 1 | 1 | 1 |
| Ifstream cInfo; | 1 | 1 | 1 |
| if (file not open){ | 1 | 1 | 1 |
| cout << “ could not open file”; | 1 | 1 | 1 |
| BinarySearchTree\* bst; | 1 | 1 | 1 |
| bst = new BinarySearchTree(); | 1 | 1 | 1 |
| While (not end of file) { | 1 | n | n |
| Split every line into strings at every comma | 1 | n | n |
| Add every string to courseInfo vector; | 1 | n | n |
| confirmCourseNum.push\_back(courseInfo.at(0)) | 1 | n | n |
| Course course; | 1 | n | n |
| course.num = courseInfo.at(0) | 1 | n | n |
| course.name = courseInfo.at(1) | 1 | n | n |
| If (courseInfo.size() > 2){ | 1 | n | n |
| For (i = 2; i < courseInfo.size(); ++i) { | 1 | n(n) | N^2 |
| Course.prerequisites = courseInfo.at(i) + “ , ” | 1 | Constant c | c |
| ConfirmCoursePrerequisites.push\_back(courseInfo.at(i)) | 1 | Constant c | c |
| bst->Insert(course); | 1 | n | n |
|  |  | **Total cost** | 9n+2cn^2 |
|  |  | **runtime** | N^2 |

**Analysis of each data structure**

The requirements include reading a file, then we must create a menu to give the user the option to load the data from the file and place it into a data structure, print the data in order, print a specific piece of data or exit the program. If a vector is used, then a function to sort the data must be created and called before the printing method can be called so that the data can be printed in order. The same would be required for a hash table. a binary search tree would not require this since traversing the tree already traverses in order. To search for a specific piece of data in the data structure, a vector would have to compare every entry from the beginning until it is found (O(n)), a hash table only compares if the hash key matches(O(1)), and a tree would do it in O(log n). The hash table could do it the fastest. When inserting a new piece of data, a vector would be fastest since it is simply added to the end. A hash table would need to create a has key first and then handle a possible collision before it is added. A binary tree would need to traverse the tree to find the correct place for the data.

**Recommendation**

Bases on the advantaged and disadvantages I would recommend going with the hash table. Although we will need collision handling and separate sorting function, it will be faster to search for and insert data than both the vector and binary search tree. This will lead to better overall efficiency which is why it is the best choice.