**Menu Pseudocode:**

void printMenu() {

print “Menu:”

print “ 1. Load Courses”

print “ 2. Print Course List”

print ” 3. Print Course”

print “ 4. Exit”

print “Enter Choice:”

int choice = user input

while (choice != 4) {

switch(choice) {

case 1:

var courses = loadCourses()

break

case 2:

printAllCourses()

break

case 3:

print “Enter Course Number for requested course:”

int courseNum = user input

printCourse(courses, courseNum)

break

}

}

}

**Vector printAllCourses():**

void printAllCourses(Vector<Course> courses) {

foreach course c in courses

print course info

}

**Hash Table printAllCourses():**

void printAllCourses(Hashtable<Course> courses) {

for (int i = 0; if i < courses.size; increment i)

if courses[i] is valid

print course info

}

**Binary Search Tree printAllCourses():**

void printAllCourses(Tree<Course> courses) {

Node currentNode = courses root

printAllCourses(currentNode)

}

void printAllCourses(Node currentNode) {

if currentNode != null

printAllCourses(currentNode left child)

print currentNode’s course info

printAllCourses(currentNode right child)

}

**Run-Time Evaluations:**

Vector printAllCourses():

| Code | Line Cost | # Times Executed | Total Cost |
| --- | --- | --- | --- |
| foreach course c in courses | 1 | n | n |
| print course info | 1 | n | n |
| Total Cost | | | 2n |
| Runtime | | | O(n) |

Hash Table printAllCourses():

| Code | Line Cost | # Times Executed | Total Cost |
| --- | --- | --- | --- |
| for (int i = 0; if i < courses.size; increment i) | 1 | n | n |
| if courses[i] is valid | 1 | n | n |
| print course info | 1 | n | n |
| Total Cost | | | 3n |
| Runtime | | | O(n) |

Binary Search Tree printAllCourses():

| Code | Line Cost | # Times Executed | Total Cost |
| --- | --- | --- | --- |
| Node currentNode = courses root | 1 | n | n |
| printAllCourses(currentNode) | 1 | n | n |
| if currentNode != null | 1 | n | n |
| printAllCourses(currentNode left child) | n | n | n |
| print currentNode’s course info | 1 | n | n |
| printAllCourses(currentNode right child) | n | n | n |
| Total Cost | | | 6n |
| Runtime | | | O(n) |

**Evaluation:**

Each data structure has its own inherent strengths and weaknesses. For example, the process to add a Course to a vector is significantly easier than adding one to a Binary Search Tree as the Course’s position is more arbitrary in the vector than the BST. The action of printing all the items in any of the three data structures is relatively the same in terms of runtime complexity but can require a bit more code or be more logically complex with some(BST, for example) than others(Vector or Hash table). A sorted vector would be the most inefficient to search through of the three but would have the lowest space requirement as it doesn’t require extra data structures(namely, nodes) to store the data in.

For my code, I will be using a Hash Table data structure to store the courses for its efficiency in both being able to add/find items and the relative ease with which you can display all of the items. The average runtime complexity for finding a specific item in a Hash Table is the best as it’s based on a hashing function and can greatly reduce the number of items that must be checked. If the ability to display all the courses in order were a top priority, I would instead choose to use a Binary Search Tree structure to store the Courses in as it would be the most efficient for that purpose.