**Vector**

// #1 Design pseudocode to define how the program opens the file, reads the data

// from the file, parses each line, and checks for file format errors

void ReadFile(string fileName) {

open fileName

if fileName did not open properly

return

for each line in file

split line on "," and store values in array

if array length < 2

print file format error

break out of loop

for each array element after first two elements

// prerequisite cannot be higher level than main course

// and prerequisite must exist as a main course in file

if element > courseNumber OR element does not exist as main course in file

print file format error

}

// #2 Design pseudocode to show how to create course objects

// and store them inthe appropriate data structure

struct course {

string courseNum, courseName

vector<string> coursePrereqs

course(string num, string name, vector<string> prereqs) :

courseNum(num), courseName(name), coursePrereqs(prereqs) {}

};

void StoreCourseInVector(vector<course> courses, array lineFromFile) {

string courseNum = lineFromFile[0] // course number is first element

string courseName = lineFromFile[1] // course name is second element

vector<string> prereqs

// use vector for prerequisites since we don't know how many there are

for each element of lineFromFile after first two elements

prereqs.push\_back(element)

courses.push\_back(course(courseNum, courseName, prereqs))

}

// #3 Design pseudocode that will search the data structure for a specific

// course and print out course information and prerequisites

void PrintCourseInfo(vector<course> courses, string courseNum) {

for each course in courses

if course.number equals courseNum

print course information

for each prereq in course

print prereq information

}

**Hash Table**

// #1 Design pseudocode to define how the program opens the file, reads the data

// from the file, parses each line, and checks for file format errors

// (this step is the same regardless of data structure being used)

void ReadFile(string fileName) {

open fileName

if fileName did not open properly

return

for each line in file

split line on "," and store values in array

if array length < 2

print file format error

break out of loop

for each array element after first two elements

// prerequisite cannot be higher level than main course

// and prerequisite must exist as a main course in file

if element > courseNumber OR element does not exist as main course in file

print file format error

}

// #2 Design pseudocode to show how to create course objects

// and store them in the appropriate data structure

// (the only thing that changes here is the name of the function

// and the data type of “course” being passed to the function.)

struct course {

string courseNum, courseName

vector<string> coursePrereqs

course(string num, string name, vector<string> prereqs) :

courseNum(num), courseName(name), coursePrereqs(prereqs) {}

};

void StoreCourseInHashMap(HashMap<course> courses, array lineFromFile) {

string courseNum = lineFromFile[0] // course number is first element

string courseName = lineFromFile[1] // course name is second element

vector<string> prereqs

// use vector for prerequisites since we don't know how many there are

for each element of lineFromFile after first two elements

prereqs.push\_back(element)

courses.InsertItem(course(courseNum, courseName, prereqs))

}

// #3 Design pseudocode that will search the data structure for a specific

// course and print out course information and prerequisites

// (the only thing that has changed here is the data type of “courses”)

void PrintCourseInfo(HashMap<course> courses, string courseNum) {

for each course in courses

if course.number equals courseNum

print course information

for each prereq in course

print prereq information

}

**Binary Search Tree**

// #1 Design pseudocode to define how the program opens the file, reads the data

// from the file, parses each line, and checks for file format errors

// (this step is the same regardless of data structure being used)

void ReadFile(string fileName) {

open fileName

if fileName did not open properly

return

for each line in file

split line on "," and store values in array

if array length < 2

print file format error

break out of loop

for each array element after first two elements

// prerequisite cannot be higher level than main course

// and prerequisite must exist as a main course in file

if element > courseNumber OR element does not exist as main course in file

print file format error

}

// #2 Design pseudocode to show how to create course objects

// and store them in the appropriate data structure

// (the only thing that changes here is the name of the function

// and the data type of “course” being passed to the function.)

struct course {

string courseNum, courseName

vector<string> coursePrereqs

course(string num, string name, vector<string> prereqs) :

courseNum(num), courseName(name), coursePrereqs(prereqs) {}

};

void StoreCourseInBSTree(BSTree<course> courses, array lineFromFile) {

string courseNum = lineFromFile[0] // course number is first element

string courseName = lineFromFile[1] // course name is second element

vector<string> prereqs

// use vector for prerequisites since we don't know how many there are

for each element of lineFromFile after first two elements

prereqs.push\_back(element)

courses.InsertItem(course(courseNum, courseName, prereqs))

}

// #3 Design pseudocode that will search the data structure for a specific

// course and print out course information and prerequisites

// (the only thing that has changed here is the data type of “courses”)

// (this is simplified, obviously. using “for each” here assumes inorder

// traversal of the BST)

void PrintCourseInfo(BSTree<course> courses, string courseNum) {

for each course in courses

if course.number equals courseNum

print course information

for each prereq in course

print prereq information

}

**Menu Pseudocode**

void printMenu() {

while input is not "exit"

if input is "load"

load file into data structure

else if input is "print list"

print alphanumerically ordered list

else if input is "print course"

get input for which course to print

if course input is valid

print course number, title, and prerequisites

else

print "invalid input"

else if input is "exit"

break out of loop // implied by while condition

else

print "invalid option"

}

**Print alphanumerically sorted list Pseudocode**

**Vector**

void printCourseList(vector<course> courseList) {

sort courseList using mergesort

for each course in courseList

print course

}

**Hash Table**

void printCourseList(HashTable<course> courseList) {

array sortedList = courseList[0]

for each course in courseList[1->n]

for each sortedCourse in sortedList

if course < sortedCourse

insert course into sortedList before sortedCourse

for each course in sortedList

print course

}

**Binary Search Tree**

void printCourseList(BSTree<course> courseList) {

for course in inorder traversal of courseList

print course

}

**Vector – Reading file and adding course object**

// I am re-writing the pseudocode for reading the file and

// inserting the object into the data structure for brevity,

// since I now better understand the purpose of the assignment.

// NOTE: since inserting an object into the middle of a vector

// is O(n), and appending is O(1), I am going to use the appending

// version since it is faster and we don't care what order the objects

// are inserted in

void readFileAndInsertObject(vector<course> courses, File file) {

for each line in file

create course object from line

add object to the END of courses

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for each line in file** | 1 | n | n |
| **Create course object from line** | 1 | n | n |
| **Add object to the END of courses (cost is O(1))** | 1 | n | n |
| **Total Cost** | | | 3n |
| **Runtime** | | | O(n) |

**Vector – Print course list alphanumerically**

void printCourseList(vector<course> courseList) {

sort courseList using mergesort

for each course in courseList

print course

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Sort courseList using mergesort** | n log n | 1 | n log n |
| **For each course in courseList** | 1 | n | n |
| **Print course** | 1 | n | n |
| **Total Cost** | | | 2n + n log n |
| **Runtime** | | | O(n log n) |

**Vector – print course information (search)**

void PrintCourseInfo(vector<course> courses, string courseNum) {

for each course in courses

if course.number equals courseNum

print course information

for each prereq in course

print prereq information

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for each course in courses** | 1 | n | n |
| **if course.number equals courseNum** | 1 | n | n |
| **print course information** | 1 | n | n |
| **for each prereq in course** | 1 | n | n |
| **print prereq information** | 1 | n | n |
| **Total Cost** | | | 5n |
| **Runtime** | | | O(n) |

**Hash Table – Reading file and adding course object**

void readFileAndInsertObject(HashTable<course> courses, File file) {

for each line in file

create course object from line

insert object to courses hash table

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **For each line in file** | 1 | n | n |
| **Create course object from line** | 1 | n | n |
| **Insert object to courses hash table** | 1 | n | n |
| **Total Cost** | | | 3n |
| **Runtime** | | | O(n) |

**Hash Table – Print course list alphanumerically**

void printCourseList(HashTable<course> courseList) {

array sortedList = courseList[0]

for each course in courseList[1->n]

for each sortedCourse in sortedList

if course < sortedCourse

insert course into sortedList before sortedCourse

for each course in sortedList

print course

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Array sourtedList = courseList[0]** | 1 | 1 | 1 |
| **For each course in courseList[1->n]** | 1 | n - 1 | n - 1 |
| **For each sortedCourse in sortedList** | 1 | n(n – 1) | n(n – 1) |
| **if course < sortedCourse** | 1 | n(n – 1) | n(n – 1) |
| **insert course into sortedList before sortedCourse** | n | 1 | n |
| **for each course in sortedList** | 1 | n | n |
| **print course** | 1 | n | n |
| **Total Cost** | | | 2n^2 + 4n -2 |
| **Runtime** | | | O(n^2) |

**Hash Table – print course information (search)**

void PrintCourseInfo(HashTable<course> courses, string courseNum) {

get key by hashing courseNum

print courses[key]

for each prereq of course

print prereq

}

NOTE: for the prerequisites here I am counting 1 because, although there can be multiple prerequisites, there are only a few and it doesn’t impact the runtime significantly. I don’t know how else to handle this part. ¯\\_(ツ)\_/¯

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **get key by hashing courseNum** | 1 | 1 | 1 |
| **print courses[key]** | 1 | 1 | 1 |
| **for each prereq of course** | 1 | 1 | 1 |
| **print prereq** | 1 | 1 | 1 |
| **Total Cost** | | | 4 |
| **Runtime** | | | O(1) |

**Binary Search Tree – Reading file and adding course objects**

void readFileAndInsertObject(BSTree<course> courses, File file) {

for each line in file

create course object from line

insert object into BST

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for each line in file** | 1 | n | n |
| **Create course object from line** | 1 | n | n |
| **insert object into BST** | log n | n | n log n |
| **Total Cost** | | | 2n + n log n |
| **Runtime** | | | O(n log n) |

**Binary Search Tree – Print course list alphanumerically**

void printCourseList(BSTree<course> courseList) {

for course in inorder traversal of courseList

print course

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for course in inorder traversal of courseList** | 1 | n | n |
| **print course** | 1 | n | n |
| **Total Cost** | | | 2n |
| **Runtime** | | | O(n) |

**Binary Search Tree – print course information (search)**

void PrintCourseInfo(BStree<course> courses, string courseNum) {

search courses tree using courseNum

print course information

for each prereq in course

print prereq information

}

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **search courses tree using courseNum** | log n | 1 | log n |
| **print course information** | 1 | 1 | 1 |
| **for each prereq in course** | 1 | 1 | 1 |
| **print prereq information** | 1 | 1 | 1 |
| **Total Cost** | | | log n + 3 |
| **Runtime** | | | O(log n) |

**Advantages and disadvantages**

The advantages of using a vector are that it is easy to use, and the computational cost of accessing an element is constant. Another advantage is that the major sorting algorithms can be applied to a vector, whereas those algorithms will not work with a hash table or a BST. Vectors are generally the “goto” data structure used in the majority of applications. Vectors do have a few disadvantages though, and they are not trivial. Since vectors allocate the memory they use dynamically, a disadvantage can be wasted memory (the vector was dynamically resized to hold only one more variable resulting in unused elements). Also, this dynamic allocation could result in slow program execution if the amount of elements being stored in the vector is constantly changing by a large amount.

The advantages to a hash table are speed, and ease of use. Hash tables are perfect for a set of data that doesn’t change much because accessing, searching, inserting, and deleting are all constant time operations (when no collisions are involved). Where hash tables become disadvantageous is when the data being stored results in a large number of collisions. If data is constantly being added to a hash table, it will result in more and more collisions and therefore slower performance. In my opinion, the best use for a hash table is for a data set that doesn’t change much, if at all. It is like a look-up table.

The advantages to a binary search tree are that all of the operations can be done in logN time, as opposed to an array which is N time. A tree is constructed using pointers, so memory is conserved by only using exactly what is needed by the data. On the other hand, memory leaks can be an issue if not implemented correctly. Another advantage is that sorting a BST is basically built into the structure. Using in order traversal of a BST we can get a sorted list of the data fairly easily and in O(n) time. The main disadvantage to a BST is similar to that of hash tables: the data has to be very specifically planned out in order to fully utilize the benefits of the structure. For example, it is possible to create a BST that only contains a right subtree! This would make for a very inefficient data structure and therefore, application.

**Recommendation**

Looking at the run-time analysis of each data structure, we have:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vector | Hash Table | Binary Search Tree |
| Reading file and inserting objects | O(n) | O(n) | O(n log n) |
| Printing course list sorted alphanumerically | O(n log n) | O(n^2) | O(n) |
| Searching for course and printing information | O(n) | O(1) | O(log n) |

So, in my opinion, hash table is immediately taken out of the running. O(n^2) for printing a sorted course list (which will likely happen quite frequently) is not that great, and even though a hash table has an amazing search runtime (constant), I think the n^2 runtime of the sorting ruins everything for this data structure. That leaves us with Vector and BST, both with comparable times but not exactly the same. Binary Search Tree has a better print and sort runtime than Vector does, but not by a huge amount (a factor of logN). Since courses will likely be added and removed somewhat regularly, I would recommend using a vector. I would like to use a BST here, but I don’t think it is a perfect fit. With a BST we have to worry about balancing the tree every so often, otherwise the efficiency gained by using it is lost. Since the run-time analysis of each structure is similar, I would go with a Vector here. One could argue that printing a sorted list will happen more often than inserting objects and therefore a BST might be faster, but I personally don’t think the increase is significant enough to warrant using it. BST has the best runtime performance, but it comes with the cost of complexity and overhead in having to keep the tree in a balanced state. Vectors are very simple, straightforward, and easy to use, and a course list such as this is a great fit for this data structure, in my opinion