# CS 300 Pseudocode Document

// Vector pseudocode

// Requirement 1a

**void validateCourseCSV(String fileName)** {

// parse lines and find prereqs

create empty list prereqs

for each line in course file

// check that each line has at least 2 parameters

if line has less than 2 parameters

print error course data incorrect format

continue

courseNum = first parameters

courseTitle = second parameters

for each remaining parameters

add courseNum to prereqs

// search file for courses matching prerequisites

for each prereq in prereqs

for each line in course file

if courseNum matches

continue

else

return error prereq not in list

}

// Requirement 1b

**Vector<Course> loadCourses(String fileName)** {

open courses file “fileName”

for each course in course file

create course object

course number = first parameter

course title = second parameter

add course number and course title to course object

for each remaining parameter

add to course object as prerequisite

append course object to vector

return vector of course objects

}

// Requirement 1c

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

for all courses

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

// Requirement 2

void menu(String csvFile) {

initialize vector object to hold each course

while choice is not “exit program”

get choice from input

// 2a

if choice is 1

load csvFile contents to vector with loadCourses function

// 2b

elif choice is 2

print alphanumerically ordered list of all courses with printAllSorted function

// 2c

elif choice is 3

get course number from input

print course info and all prereq info with printCourseInformation function

// 2d

elif choice is 4

exit while loop and end program

}

// Requirement 3

void printAllSorted() {

// 3a: bubble sort

for all elements **i** in vector up until 2 before the last

for all elements **j** in vector until 2+i before the last

if j is greater than the next element

swap j and the next element

//3b: print sorted

for each element in sorted vector

print courseNum and courseName

}

// Hashtable pseudocode

// Requirement 1

**void validateCourseCSV(String fileName)** {

// parse lines and find prereqs

create empty list prereqs

for each line in course file

// check that each line has at least 2 parameters

if line has less than 2 parameters

print error course data incorrect format

continue

courseNum = first parameters

courseTitle = second parameters

for each remaining parameters

add courseNum to prereqs

// search file for courses matching prerequisites

for each prereq in prereqs

for each line in course file

if courseNum matches

continue

else

return error prereq not in list

}

// Requirement 2

**HashTable<Course> loadCourses(String fileName)** {

open courses file “fileName”

for each course in course file

// generate course info from line in file

create course object

course number = first parameter

course title = second parameter

add course number and course title to course object

for each remaining parameter

add to course object as prerequisite

// add course to hash table

generate key from course id hash

while hash table at key index is taken

go to next index

add course to hash table at key index

return hash table of course objects

}

// Requirement 3

**void printCourseInformation(HashTable<Course> courses, String courseNumber)** {

// hash table lookup is O(1), no search needed

get course at table index of hashed courseNumber

print out the course information

for each prerequisite of the course

get course at table index of hashed prereq course number

print the prerequisite course information

}

// Requirement 2

void menu(String csvFile) {

initialize vector object to hold each course

while choice is not “exit program”

get choice from input

// 2a

if choice is 1

load csvFile contents to vector with loadCourses function

// 2b

elif choice is 2

print alphanumerically ordered list of all courses with printAllSorted function

// 2c

elif choice is 3

get course number from input

print course info and all prereq info with printCourseInformation function

// 2d

elif choice is 4

exit while loop and end program

}

// Requirement 3

void printAllSorted() {

// 3a: insertion sort

initialize empty array

for each bucket in pre-filled hash table with chaining

for each course in bucket

add course to array

for each course in array

for each course before current course

if course is greater than current course

move greater course to the right of current course

//3b: print sorted

for each element in sorted array

print courseNum and courseName

}

// Tree pseudocode

// Requirement 1

**void validateCourseCSV(String fileName)** {

// parse lines and find prereqs

create empty list prereqs

for each line in course file

// check that each line has at least 2 parameters

if line has less than 2 parameters

print error course data incorrect format

continue

courseNum = first parameters

courseTitle = second parameters

for each remaining parameters

add courseNum to prereqs

// search file for courses matching prerequisites

for each prereq in prereqs

for each line in course file

if courseNum matches

continue

else

return error prereq not in list

}

// Requirement 2

**BinarySearchTree<Course> loadCourses(String fileName)** {

open courses file “fileName”

for each course in course file

// generate course info from line in file

create course object

course number = first parameter

course title = second parameter

add course number and course title to course object

for each remaining parameter

add to course object as prerequisite

// add course to binary tree

// SEPARATE RECURSIVE FUNCTION

start at root of tree

if root of tree is null

add course node to tree as root

else

if course id is smaller than current node

if node has no left child

add course as left child

else if node has left child

recurse down left subtree

else if course id is larger than current node

if node has no right child

add course as right child

else if node has right child

recurse down right subtree

return binary tree of course objects

}

// Requirement 3

**void printCourseInformation(BinarySearchTree<Course> courses, String courseNumber)** {

start at root of tree

while current node not null

if current node matches course number

print course information

for each prereq of course

print out course information

if course number is smaller than current node’s course number

traverse to left child

else if course number is larger than current node’s course number

traverse to right child

print course not found if course not found

}

// Requirement 2

void menu(String csvFile) {

initialize vector object to hold each course

while choice is not “exit program”

get choice from input

// 2a

if choice is 1

load csvFile contents to vector with loadCourses function

// 2b

elif choice is 2

print alphanumerically ordered list of all courses with printAllSorted function

// 2c

elif choice is 3

get course number from input

print course info and all prereq info with printCourseInformation function

// 2d

elif choice is 4

exit while loop and end program

}

// Requirement 3

void printAllSorted() {

//3a+b: print sorted

start at root node

// this is a recursive inOrderTraversal function that visits and prints each node information from lowest to highest alphanumerically

recursively traverse left children using root as its parent

print course info at node

recursively traverse right children using root as its parent

}

**Runtime Analysis**

| **Vector::validateCourses()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| create empty list prereqs | 1 | 1 | 1 |
| for each line in course file | 1 | n | n |
| if line has less than 2 parameters | 1 | 1 | 1 |
| print error course data incorrect format | 1 | 1 | 1 |
| **continue** | 1 | 0 | 0 |
| courseNum = first parameters | 1 | 1 | 1 |
| courseTitle = second parameters | 1 | 1 | 1 |
| for each remaining parameters | 1 | n | n |
| add courseNum to prereqs | 1 | 1 | 1 |
| for each prereq in prereqs | 1 | n | n |
| for each line in course file | 1 | n | n |
| if courseNum matches | 1 | 1 | 1 |
| continue | 1 | 0 | 1 |
| else | 1 | 0 | 1 |
| return error prereq not in list | 1 | 1 | 1 |
| **Total Cost** | | | 4n + 10 |
| **Runtime** | | | O(n) |

| **Vector::loadCourses()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| for each course in course file | 1 | n | n |
| create course object | 1 | 1 | 1 |
| course number = first param | 1 | 1 | 1 |
| course title = second param | 1 | 1 | 1 |
| add coursenum and coursetitle to course object | 1 | 1 | 1 |
| for each remaining parameter | 1 | ~2 | 1 |
| add to course object as prereq | 1 | 1 | 1 |
| append course object to vector | 1 | 1 | 1 |
| **Total Cost** | | | n + 7 |
| **Runtime** | | | O(n) |

| **Hash Table::validateCourses()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| create empty list prereqs | 1 | 1 | 1 |
| for each line in course file | 1 | n | n |
| if line has less than 2 parameters | 1 | 1 | 1 |
| print error course data incorrect format | 1 | 1 | 1 |
| continue | 1 | 0 | 1 |
| course number = first param | 1 | 1 | 1 |
| course title = second param | 1 | 1 | 1 |
| **for each remaining parameter** | 1 | ~2 | 3 |
| **add courseNum to prereqs** | 1 | 1 | 1 |
| **for each prereq in prereqs** | 1 | n | n |
| **for each line in course file** | 1 | n | n |
| **if coursenum matches** | 1 | 1 | 1 |
| **continue** | 1 | 1 | 1 |
| **else** | 1 | 0 | 1 |
| **return error prereq not in list** | 1 | 1 | 1 |
| **Total Cost** | | | 3n+11 |
| **Runtime** | | | O(n) |

| **Hash Table::loadCourses()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| open courses file “fileName” | 1 | 1 | 1 |
| for each course in course file | 1 | n | n |
| create course object | 1 | 1 | 1 |
| course number = first parameter | 1 | 1 | 1 |
| course title = second parameter | 1 | 1 | 1 |
| add course number and course title to course object | 1 | 1 | 1 |
| for each remaining parameter | 1 | 2 | 1 |
| add to course object as prerequisite | 1 | 1 | 1 |
| generate key from course id hash | 1 | 1 | 1 |
| while hash table at key index is taken | 1 | 6 | 6 |
| go to next index | 1 | 1 | 1 |
| add course to hash table at key index | 1 | 1 | 1 |
| return hash table of course objects | 1 | 1 | 1 |
| **Total Cost** | | | n+18 |
| **Runtime** | | | O(n) |

| **Binary Search Tree::validateCourses()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| create empty list prereqs | 1 | 1 | 1 |
| for each line in course file | 1 | n | n |
| if line has less than 2 parameters | 1 | 1 | 1 |
| print error course data incorrect format | 1 | 1 | 1 |
| continue | 1 | 0 | 0 |
| courseNum = first parameters | 1 | 1 | 1 |
| courseTitle = second parameters | 1 | 1 | 1 |
| for each remaining parameters | 1 | 3 | 3 |
| add courseNum to prereqs | 1 | 1 | 1 |
| for each prereq in prereqs | 1 | n | n |
| for each line in course file | 1 | n | n |
| if courseNum matches | 1 | 1 | 1 |
| continue | 1 | 0 | 0 |
| else | 1 | 0 | 0 |
| return error prereq not in list | 1 | 1 | 1 |
| **Total Cost** | | | 3n+11 |
| **Runtime** | | | O(n) |

| **Binary Search Tree::loadCourses()** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| for each course in course file | 1 | n | n |
| create course object | 1 | 1 | 1 |
| coursenum, coursetitle = first, second param | 2 | 2 | 2 |
| add course number and course title to course object | 1 | 1 | 1 |
| for each remaining parameter | 1 | 3 | 3 |
| add to course object as prerequisite | 1 | 1 | 1 |
| start at root of tree | 1 | 1 | 1 |
| if root of tree is null | 1 | 1 | 1 |
| add course node to tree as root | 1 | 1 | 1 |
| else | 1 | 1 | 1 |
| if course id is smaller than current node | 1 | 1 | 1 |
| if node has no left child | 1 | 1 | 1 |
| add course as left child | 1 | 1 | 1 |
| else if node has left child | 1 | 1 | 1 |
| recurse down left subtree | 1 | 1 | 1 |
| else if course id is larger than current node | 1 | 1 | 1 |
| if node has no right child | 1 | 1 | 1 |
| add course as right child | 1 | 1 | 1 |
| else if node has right child | 1 | 1 | 1 |
| recurse down right subtree | 1 | 1 | 1 |
| **Total Cost** | | | n + 20 |
| **Runtime** | | | O(n) |

**Vector Analysis**

Vectors have a few key benefits in terms of being able to quickly access elements and dynamically resize themselves, as well as having a simple interface with a high level of abstraction. They are well rounded and can be applied to many different use cases. The downside is that inserting and deleting elements might be slow, but since we are using a singly-linked list for implementation, this is not an issue. Searching however, does take linear time to perform as each element in the vector must be checked in order.

**Hash Table Analysis**

Hash tables provide incredibly fast searching by using a hashed key to find elements in constant time, the best and most efficient among the data types we see here. It is also very quick at inserting and deleting, especially when using chaining, which I have assumed and used here. The downsides rely mainly on the hash function and if it is efficient, but since we are dealing with courses at a school it should be fairly simple based on their unique course ID key. Resizing can also be taxing, but again, chaining would be a logical way to reduce the number of resize functions necessary, and this would only occur rarely.

**Binary Search Tree Analysis**

Binary search trees provide logarithmic time searching, making it very fast and efficient to find elements. Since the data structure itself is created and populated in a logical manner, it will be sorted by default and lookups are very quick. Recursion is used often, but overall makes things a lot more efficient. In order traversal is a great tool that we have with BSTs and allows for simple sorting, which is really just traversal by passing over each node in a certain way. Disadvantages to BSTs show themselves when you have to remove or insert a node, because you have to make sure everything is in the correct position once you are done. A BST is by nature, organized and must stay that way, so maintenance is necessary every time you change it, sometimes more than others.

**Recommendation**

I would recommend using a hash table for this use case, as there will not realistically be too large of a list of courses. Even if you wanted to list every possible course in the entire world at every single school, the number of courses for input would probably not exceed 10,000. The benefit of a hash table in this case would be that each course could inhabit its own bucket in the table, with chaining to add prerequisites to the bucket of the class they refer to. This would add an additional layer of functionality that could be implemented however might seem logical, but this still may not be the most efficient use of memory. The good part is that the list of courses would not take up much memory since it would usually not be that large.

If however, the number of elements were to grow substantially, I would still recommend the hash table since looking up an element can be done in constant time using the hashing function to retrieve values from each course's hashed key. This is the main benefit of hash tables, they are very quick for lookup. The algorithms that we have analyzed here for the hash table are all done in linear time, but lookups are done in constant time, so inherently any other functions you would like to create with this data structure would usually be less complex and more efficient.