# CS 300 Pseudocode Document

## //vector

## struct Course {

## private:

## string courseNumber

## string name

## vector<string> preRequisite

## public:

## Course() {

## set courseNumber to empty string

## set name to empty string

## }

## void setCourseNumber(string courseNumber) {

## set courseNumber to courseNumber

## }

## void setName(string name) {

## set name to name

## }

## void setPrerequisite(string preReq) {

## append preReq to preRequisite vector

## }

## }

## vector<Course> parseFile(csvPath) {

## vector<Course> courses

## vector<string> courseNames

## ifstream open csvPath

## for each line add the first substring to courseNames

## while it is not the end of the file

## read line

## if line has less than two values

## print error and exit

## else

## int index

## Course tempCourse = new Course()

## for each substring in line

## switch on index

## case 0

## setCourseNumber for tempCourse to substring

## case 1

## setName for tempCourse to substring

## default

## auto check = find substring in courseNames

## if check is equal to substring

## setPrerequisite for tempCourse to substring

## else

## print error and exit

## increment index

## add tempCourse to courses

## ifstream close file

## }

## void searchCourse(vector<Course> courses, String courseNumber) {

## for all courses

## if the course courseNumber is the same as courseNumber

## print out the course information

## for each preRequisite of the course

## print the preRequisite course information

## }

## int partition(vector<Course>& courses, int begin, int end) {

## set int lowIndex to begin

## set int highIndex to end

## set int middlePoint to lowIndex + (highIndex - lowIndex) / 2

## set string pivot to course number of Course at middlePoint

## set bool done to false

## while done is false

## while the course number for course at courses lowIndex compared to the pivot is less than 0

## lowIndex increment

## while the course number for course at courses highIndex compared to the pivot is greater than 0

## highIndex decrement

## if lowIndex is greater than highIndex

## set done to true

## else

## swap courses lowIndex and highIndex

## lowIndex increment

## highIndex decrement

## 

## return highIndex

## }

## void quickSort(vector<Course>& courses, int begin, int end) {

## 

## if begin is greater than or equal to end

## return

## 

## set int lowEndIndex to partition(courses, begin, end)

## 

## quickSort(courses, begin, lowEndIndex)

## quickSort(courses, lowEndIndex + 1, end)

## }

## void Menu() {

## 

## set bool exit to false

## 

## while exit is false

## print menu option to load data, print alphanumerically ordered list, print course title and prereqs, or exit the program

## set int userInput to users input

## switch on userInput

## case 1

## set vector<Course> courses to parseFile(csv file path)

## case 2

## quickSort(courses&)

## for each Course in courses

## print Course information

## case 3

## print prompt for a course number

## set String courseNumber to user input

## searchCourse(courses, courseNumber)

## case 9

## set exit to true

## default

## print that input was not recognized

## }

## //Hash Table

## struct Course {

## private:

## string courseNumber

## string name

## vector<string> preRequisite

## public:

## Course() {

## set courseNumber to empty string

## set name to empty string

## }

## void setCourseNumber(string courseNumber) {

## set courseNumber to courseNumber

## }

## void setName(string name) {

## set name to name

## }

## void setPrerequisite(string preReq) {

## append preReq to preRequisite vector

## }

## }

## class HashTable {

## private:

## struct Node {

## Course course

## unsigned int key

## Node\* next

## Node() {

## set key to UINT\_MAX

## set next to nullptr

## }

## Node(Course course) : Node() {

## set this course to course

## }

## Node(Course course, unsigned int aKey) : Node(course) {

## set key to aKey

## }

## };

## vector<Node> nodes

## unsigned int tableSize = DEFAULT\_SIZE

## unsigned int hash(int key)

## public:

## void parseFile(csvPath)

## void Insert(Course, unsigned int)

## void print(courseNumber)

## void printInOrder(HashTable)

## }

## HashTable::parseFile(csvPath) {

## vector<string> courseNames

## ifstream open csvPath

## for each line add the first substring to courseNames

## while it is not the end of the file

## read line

## if line has less than two values

## print error and exit

## else

## int index

## unsigned int key

## Course tempCourse = new Course()

## for each substring in line

## switch on index

## case 0

## setCourseNumber for tempCourse to substring

## key = hash(substring)

## case 1

## setName for tempCourse to substring

## default

## auto check = find substring in courseNames

## if check is equal to substring

## setPrerequisite for tempCourse to substring

## else

## print error and exit

## increment index

## Insert(tempCourse, key)

## ifstream close file

## }

## void HashTable::Insert(Course course, unsigned int tempKey) {

## if node key at tempKey is set to default value

## tempNode = Node(course, tempKey)

## node at tempKey = tempNode

## else

## currentPos = pointer to tempKey

## tempNode = Node(course, tempKey)

## while the next pointer is not null

## currentPos = next pointer

## next pointer = tempNode

## }

## unsigned int HashTable::hash(string key) {

## convert key to int

## return key % tableSize

## }

## void HashTable::printCourse(int courseNumber) {

## set tempKey to hash(courseNumber)

## set currNode to pointer of nodes at tempKey

## while currNode != NULL

## if currNode course number is equal to course number

## print course number, course name

## for each node in preRequisite

## print node

## else

## currNode = next node

## }

## void HashTable::printInOrder(HashTable course) {

## vector<String> courseNumbers

## for each node in HashTable nodes

## courseNumbers append node Course courseNumber

## quickSort(courseNumbers&)

## for each item in courseNumbers

## printCourse(item)

## }

## int HashTable::partition(vector<Course>& courses, int begin, int end) {

## set int lowIndex to begin

## set int highIndex to end

## set int middlePoint to lowIndex + (highIndex - lowIndex) / 2

## set string pivot to course number of Course at middlePoint

## set bool done to false

## while done is false

## while the course number for course at courses lowIndex compared to the pivot is less than 0

## lowIndex increment

## while the course number for course at courses highIndex compared to the pivot is greater than 0

## highIndex decrement

## if lowIndex is greater than highIndex

## set done to true

## else

## swap courses lowIndex and highIndex

## lowIndex increment

## highIndex decrement

## return highIndex

## }

## void HashTable::quickSort(vector<Course>& courses, int begin, int end) {

## if begin is greater than or equal to end

## return

## set int lowEndIndex to partition(courses, begin, end)

## quickSort(courses, begin, lowEndIndex)

## quickSort(courses, lowEndIndex + 1, end)

## }

## void Menu() {

## set bool exit to false

## while exit is false

## print menu option to load data, print alphanumerically ordered list, print course title and prereqs, or exit the program

## set int userInput to users input

## HashTable courses

## switch on userInput

## case 1

## set courses to parseFile(csv file path)

## case 2

## printInOrder(courses)

## case 3

## print prompt for a course number

## set String courseNumber to user input

## printCourse(courseNumber)

## case 9

## set exit to true

## default

## print that input was not recognized

## }

## //BST

## struct Course {

## private:

## string courseNumber

## string name

## vector<string> preRequisite

## public:

## Course() {

## set courseNumber to empty string

## set name to empty string

## }

## void setCourseNumber(string courseNumber) {

## set courseNumber to courseNumber

## }

## void setName(string name) {

## set name to name

## }

## void setPrerequisite(string preReq) {

## append preReq to preRequisite vector

## }

## }

## class BST {

## private:

## struct Node {

## Course course

## Node\* left

## Node\* right

## Node() {

## set left to nullptr

## set right to nullptr

## }

## Node(Course course) : Node() {

## set this course to course

## }

## }

## private:

## Node\* root;

## Node\* addNode(Node\* node, Course course);

## void inOrder(Node\* node);

## void SearchInOrder(String courseNumber, Node\*node)

## public:

## BinarySearchTree();

## void ParseFile(csvPath)

## void PrintCourses();

## void Insert(Course course);

## void Search(String courseNumber)

## }

## Node\* BST::addNode(Node\* node, Course course) {

## if node is null

## return new node(course)

## else if courseNumber is less than node course number

## set node left child to addNode(node left child, course)

## else

## set node right child to addNode(node right child, course)

## return node

## }

## void BST::inOrder(Node\* node) {

## if node is equal to nullptr then return

## inOrder(node left child)

## print course details

## inOrder(node right child)

## }

## void BST::SearchInOrder(String courseNumber, Node\* node) {

## if node is equal to nullptr then return

## if node Corse courseNumber is equal to courseNumber

## print course number and name

## for each preReq in Course preRequests

## print preReq

## return

## SearchInOrder(node left child)

## SearchInOrder(node right child)

## }

## 

## BST::parseFile(csvPath) {

## vector<string> courseNames

## ifstream open csvPath

## for each line add the first substring to courseNames

## while it is not the end of the file

## read line

## if line has less than two values

## print error and exit

## else

## int index

## Course tempCourse = new Course()

## for each substring in line

## switch on index

## case 0

## setCourseNumber for tempCourse to substring

## case 1

## setName for tempCourse to substring

## default

## auto check = find substring in courseNames

## if check is equal to substring

## setPrerequisite for tempCourse to substring

## else

## print error and exit

## increment index

## Insert(tempCourse)

## ifstream close file

## }

## void BST::Insert(Course course) {

## set root to addNode(root, course)

## }

## void BST::Search(String courseNumber)

## searchInOrder(coursNumber, root)

## void BST::PrintCourses() {

## inOrder(root)

## }

## void Menu() {

## set bool exit to false

## while exit is false

## print menu option to load data, print alphanumerically ordered list, print course title and prereqs, or exit the program

## set int userInput to users input

## BST courses

## switch on userInput

## case 1

## set courses to parseFile(csv file path)

## case 2

## PrintCourses()

## case 3

## print prompt for a course number

## set String courseNumber to user input

## printCourse(courseNumber)

## case 9

## set exit to true

## default

## print that input was not recognized

## }

## Runtime Analysis Vector

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Vector<Course> courses | 1 | 1 | 1 |
| vector<string> courseNames | 1 | 1 | 1 |
| ifstream open csvPath | 1 | 1 | 1 |
| for each line add the first substring to courseNames | 1 | N | N |
| while it is not the end of the file | 1 | N | N |
| read line | 1 | N | N |
| if line has less than two values | 1 | 1 | 1 |
| print error and exit | 1 | 1 | 1 |
| else | 1 | 2 | N |
| int index | 1 | N | N |
| Course tempCourse = new Course() | 2 | N | 2N |
| for each substring in line | 1 | N | N |
| switch on index | 1 | N | N |
| case 0 | 1 | N | N |
| setCourseNumber for tempCourse to substring | 1 | N | N |
| case 1 | 1 | N | N |
| setName for tempCourse to substring | 1 | N | N |
| default | 1 | N | N |
| auto check = find substring in courseNames | 2 | N | N |
| if check is equal to stubstring | 1 | N | N |
| setPrerequisite for tempCourse to substring | 1 | N | N |
| else | 1 | 1 | 1 |
| print error and exit | 1 | 1 | 1 |
| increment index | 1 | N | N |
| add tempCourse to coures | 1 | N | N |
| ifstream close file | 1 | 1 | 1 |
| **Total Cost** | | | 19n + 8 |
| **Runtime** | | | O(n) |

## Runtime Analysis Hash Table

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| vector<string> courseNames | 1 | 1 | 1 |
| ifstream open csvPath | 1 | 1 | 1 |
| for each line add the first substring to courseNames | 1 | N | N |
| while it is not the end of the file | 1 | N | N |
| read line | 1 | N | N |
| if line has less than two values | 1 | 1 | 1 |
| print error and exit | 1 | 1 | 1 |
| else | 1 | 2 | N |
| int index | 1 | N | N |
| Unsigned int key | 1 | N | N |
| Course tempCourse = new Course() | 2 | N | 2N |
| for each substring in line | 1 | N | N |
| switch on index | 1 | N | N |
| case 0 | 1 | N | N |
| setCourseNumber for tempCourse to substring | 1 | N | N |
| key = hash(substring) | 1 | N | N |
|  |  |  |  |
| convert key to int | 1 | 1 | 1 |
| return key % tableSize | 1 | 1 | 1 |
| case 1 | 1 | N | N |
| setName for tempCourse to substring | 1 | N | N |
| default | 1 | N | N |
| auto check = find substring in courseNames | 2 | N | N |
| if check is equal to stubstring | 1 | N | N |
| setPrerequisite for tempCourse to substring | 1 | N | N |
| else | 1 | 1 | 1 |
| print error and exit | 1 | 1 | 1 |
| increment index | 1 | N | N |
| Insert(tempCourse, key) | 1 | N | N |
| ifstream close file | 1 | 1 | 1 |
| **Total Cost** | | | 21n + 9 |
| **Runtime** | | | O(n) |

## Runtime Analysis BST

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| vector<string> courseNames | 1 | 1 | 1 |
| ifstream open csvPath | 1 | 1 | 1 |
| for each line add the first substring to courseNames | 1 | N | N |
| while it is not the end of the file | 1 | N | N |
| read line | 1 | N | N |
| if line has less than two values | 1 | 1 | 1 |
| print error and exit | 1 | 1 | 1 |
| else | 1 | 2 | N |
| int index | 1 | N | N |
| Course tempCourse = new Course() | 2 | N | 2N |
| for each substring in line | 1 | N | N |
| switch on index | 1 | N | N |
| case 0 | 1 | N | N |
| setCourseNumber for tempCourse to substring | 1 | N | N |
| case 1 | 1 | N | N |
| setName for tempCourse to substring | 1 | N | N |
| default | 1 | N | N |
| auto check = find substring in courseNames | 2 | N | N |
| if check is equal to stubstring | 1 | N | N |
| setPrerequisite for tempCourse to substring | 1 | N | N |
| else | 1 | 1 | 1 |
| print error and exit | 1 | 1 | 1 |
| increment index | 1 | N | N |
| Insert(tempCourse) | 1 | N | N |
| ifstream close file | 1 | 1 | 1 |
| **Total Cost** | | | 19n + 7 |
| **Runtime** | | | O(n) |

**Evaluation:**

The main advantage of the vector is that it is simple. The vector structure is built into the C++ std library and does not require building an object type to be used to store the data. Unfortunately, the data is stored unordered and therefore requires more work to sort and search for items stored in the vector.

The Hash Table is more complex than the vector and the implementation utilized in this project encapsulates a vector in the Hash Table. Its storage is also unordered which means that sorting the data within is easy. Its biggest advantage is that it is extremely efficient in retrieving data if you are aware of the key.

The BSTs main advantage is that the data is sorted as it is inserted into the tree. This makes outputting data in order easy. Additionally, since the data is sorted on the front end it reduces the number or compared items when searching for a specific node. The biggest downside is the complexity of removing a node from the structure.

Based off the runtime evaluation of parsing the document and creating the data structure there is not a clear advantage. All the data structures had a Big O notation of O(n). Therefore, the one I would recommend is BST. This is simply due to the fact that a major requirement of this system is to print an alphanumerically ordered list and the BST is the only data structure that sorts the data when inputted.