**Project One**

Menu Loop Pseudo Code

Main() function {

**int menuSelection = -1;**

**While menuSelection is not equal to 9 {**

**Output Menu options;**

**Get user input, store in menuSelection;**

**If menuSelection is not 1-3 or 9 {**

**Output that selection is invalid and prompt for new selection;**

**Continue;**

**}**

**If menuSelection is 1 {**

**Call loadBids function; // This function calls parameters will vary**

**depending on which data structure is used, but functionally performs**

**the same outcome.**

**Output number of records parsed in the CSV file.**

**}**

**Else if menuSelection is 2 {**

**Call printSortedList function; // This function calls parameters will vary depending on which data structure is used but functionally performs the same outcome.**

**}**

**Else if menuSelection is 3 {**

**Call searchCourse function; // This function calls parameters will vary depending on which data structure is used but functionally performs the same outcome.**

**}**

**}**

**}**

Vector Structure

Data Parser Pseudocode

void parseContent(String csvFile) {

**Open the CSV file passed into the function.**

**Loop through each row in the CSV file {**

**If row has at least two parameters (courseNumber, name) {**

**If row has no prerequisites {**

**Push current row’s course to the end of the vector.**

**}**

**Else {**

**If prerequisite(s) is already stored in the vector {**

**Push current row’s course to the end of the vector.**

**}**

**Else {**

**Throw error that prerequisite is not found in vector.**

**}**

**}**

**Else {**

**Throw error that row is missing necessary parameters.**

**}**

**}**

Data Loading Pseudocode

struct Course {

string courseNumber (unique identifier)

string name

vector<courseNumber> prerequisites

}

vector<Course> loadCourses(String csvPath) {

**Define a vector data structure to hold a collection of courses;**

**Initialize the CSV parser using the given csvPath parameter;**

**If file is open {**

**Loop to read rows of the CSV file {**

**Push the course to the end of the vector**

**}**

**Close File;**

**If file size is 0 {**

**Throw a “No data in file error”;**

**}**

**}**

**Else {**

**Throw a “Failed to open file” error;**

**}**

**Return vector of Courses.**

}

Search and Print Pseudocode

void searchCourse(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**

}

Alphanumeric Sorted Print Pseudocode

void quickSort(Vector<Course> courses, int begin, int end) {

**Set mid equal to 0;**

**If begin is greater than or equal to end {**

**return;**

**}**

**Partition bids into low and high such that mid is location of last element in low;**

**quickSort(courses, begin, mid); // Recursively sorts low partition.**

**quickSort(courses, mid + 1, end); //Recursively sorts high partition.**

}

void printSortedCourses(Vector<Course> courses) {

**Call quickSort(courses);**

**For all courses {**

**Print out the course information;**

**For each prerequisite of the course {**

**Print the prerequisite course information;**

**}**

**}**

}

Runtime Analysis of loadCourses Function

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Define a vector data structure to hold a collection of courses** | 1 | 1 | 1 |
| **Initialize the CSV parser using the given csvPath parameter** | n | 1 | n |
| **If file is open** | 1 | 1 | 1 |
| **Loop to read rows of the CSV file** | 1 | n | n |
| **Push the course to the end of the vector** | 1 | n | n |
| **Close File** | 1 | 1 | 1 |
| **If file size is 0** | 1 | 1 | 1 |
| **Throw a “No data in file error”** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Throw a “Failed to open file” error** | 1 | 1 | 1 |
| **Return vector of Courses.** | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 8 |
| **Runtime** | | | O(n) |

Hash Table Structure

Data Parser Pseudocode

void parseContent(String csvFile) {

**Open the CSV file passed into the function.**

**Loop through each row in the CSV file {**

**If row has at least two parameters (courseNumber, name) {**

**If row has no prerequisites {**

**Push current row’s course to the end of the vector.**

**}**

**Else {**

**If prerequisite(s) is already stored in the vector {**

**Push current row’s course to the end of the vector.**

**}**

**Else {**

**Throw error that prerequisite is not found in vector.**

**}**

**}**

**Else {**

**Throw error that row is missing necessary parameters.**

**}**

**}**

Data Loading Pseudocode

struct Node {

Course course;

unsigned int key;

Node \*next;

}

struct Course {

string courseNumber (unique identifier)

string name

vector<courseNumber> prerequisites

}

vector<Node> loadCourses(String csvPath) {

**Define a vector data structure to hold a collection of nodes**

**Initialize the CSV parser using the given csvPath parameter**

**If file is open**

**Loop to read rows of the CSV file**

**Use the modulo hash to determine the key value to insert each bid into**

**(courseNumber % Table Size)**

**If Key value’s node is empty**

**Create a Linked List to store each Course**

**Insert Course as Linked List head and tail**

**Else**

**Append Course to end of Linked List**

**Close File**

**If File Size is 0**

**Throw a “No data in File” error**

**Else**

**Throw a “Failed to open file” error**

**Return the vector of nodes.**

}

Search and Print Pseudocode

void searchCourse(Vector<Course> nodes, String courseNumber) {

**Use the courseNumber to generate a hash key**

**For all nodes**

**If hash key matches node key**

**Iterate through node’s list of courses**

**If courseNumber matches courseNumber of current list item**

**Print out the course information**

**For each prerequisite of the course**

**Print the prerequisite course information**

}

Alphanumeric Sorted Print Pseudocode

void quickSort(Vector<Course> courses, int begin, int end) {

**Set mid equal to 0;**

**If begin is greater than or equal to end {**

**return;**

**}**

**Partition bids into low and high such that mid is location of last element in low;**

**quickSort(courses, begin, mid); // Recursively sorts low partition.**

**quickSort(courses, mid + 1, end); //Recursively sorts high partition.**

}

void printSortedCourses() {

**Create an empty vector to store the courses in;**

**For each bucket in the hash table {**

**For each course in the bucket {**

**Add the course to the courses vector;**

**}**

**}**

**Call quickSort() function, passing in the courses vector, to sort by courseId;**

**For each course in courses {**

**Output course information;**

**For each prerequisite {**

**Output prerequisite information;**

**}**

**}**

}

Runtime Analysis of loadCourses Function

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Define a vector data structure to hold a collection of nodes** | 1 | 1 | 1 |
| **Initialize the CSV parser using the given csvPath parameter** | n | 1 | n |
| **If file is open** | 1 | 1 | 1 |
| **Loop to read rows of the CSV file** | 1 | n | n |
| **Use the modulo hash to determine the key value to insert each bid into** | 1 | n | n |
| **If Key value’s node is empty** | 1 | n | n |
| **Create a Linked List to store each Course** | 1 | n | n |
| **Insert Course as Linked List head and tail** | 1 | n | n |
| **Else** | 1 | n | n |
| **Append Course to end of Linked List** | 1 | n | n |
| **Close File** | 1 | 1 | 1 |
| **If file size is 0** | 1 | 1 | 1 |
| **Throw a “No data in file error”** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Throw a “Failed to open file” error** | 1 | 1 | 1 |
| **Return vector of Courses.** | 1 | 1 | 1 |
| **Total Cost** | | | 8n + 8 |
| **Runtime** | | | O(n) |

Binary Search Tree Structure

Data Parser Pseudocode

void parseContent(String csvFile) {

**Open the CSV file passed into the function.**

**Loop through each row in the CSV file {**

**If row has at least two parameters (courseNumber, name) {**

**If row has no prerequisites {**

**Push current row’s course to the end of the vector.**

**}**

**Else {**

**If prerequisite(s) is already stored in the vector {**

**Push current row’s course to the end of the vector.**

**}**

**Else {**

**Throw error that prerequisite is not found in vector.**

**}**

**}**

**Else {**

**Throw error that row is missing necessary parameters.**

**}**

**}**

Data Loading Pseudocode

struct Node {

Course course;

Node \*left;

Node \*right;

}

struct Course {

string courseNumber (unique identifier)

string name

vector<courseNumber> prerequisites

void loadCourses(String csvPath, BinarySearchTree\* bst) {

**Define a BST data structure to hold a collection of nodes**

**Initialize the CSV parser using the given csvPath parameter**

**If file is open**

**Loop to read rows of the CSV file**

**If Root node is empty**

**Insert Course as Root node**

**Else**

**If courseNumber is smaller than current node’s (starting at root) courseNumber**

**If current node has no left child**

**Insert Course as new left child**

**Else**

**Recurse down the left node**

**Else If courseNumber is larger than current node’s courseNumber**

**If current node has no right child**

**Insert Course as new right child**

**Else**

**Recurse down the right node**

**Close File**

**If File Size is 0**

**Throw a “No data in File” error**

**Else**

**Throw a “Failed to open file” error**

}

Search and Print Pseudocode

Course searchCourse(String courseNumber) {

**Set current node equal to root of the BST**

**While current isn’t an empty node**

**If current’s courseNumber is equal to the input courseNumber**

**Return Course**

**Else If current’s courseNumber is larger than the input courseNumber**

**Set current node to current’s left child (to traverse left in tree)**

**Else (If current’s courseNumber is smaller than the input courseNumber)**

**Set current node to current’s right child (to traverse right in tree)**

**Return Empty Course**

}

Alphanumeric Sorted Print Pseudocode

void inOrder(Node \*node) {

**if node is not equal to nullptr {**

**Call inOrder(), passing in the current node’s left node; // Recurses to the left**

**Output Course information;**

**For each prerequisite of the course {**

**Output prerequisite information;**

**}**

**Call inOrder(), passing in the current node’s right node; //Recurses to the**

**right;**

**}**

}

void printSortedCourses() {

**Call inOrder(), passing the root of the BST as the argument of the function;**

}

Runtime Analysis of loadCourses Function

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Define a BST data structure to hold a collection of nodes** | 1 | 1 | 1 |
| **Initialize the CSV parser using the given csvPath parameter** | n | 1 | n |
| **If file is open** | 1 | 1 | 1 |
| **Loop to read rows of the CSV file** | 1 | n | n |
| **If Root node is empty** | 1 | n | n |
| **Insert Course as Root node** | 1 | n | n |
| **Else** | 1 | n | n |
| **If courseNumber is smaller than current node’s (starting at root) courseNumber** | 1 | n | n |
| **If current node has no left child** | 1 | n | n |
| **Insert Course as new left child** | 1 | n | n |
| **Else** | 1 | n | n |
| **Recurse down the left node** | n | n | n^2 |
| **Else If courseNumber is larger than current node’s courseNumber** | 1 | n | n |
| **If current node has no right child** | 1 | n | n |
| **Insert Course as new right child** | 1 | n | n |
| **Else** | 1 | n | n |
| **Recurse down the right node** | n | n | n^2 |
| **Close File** | 1 | 1 | 1 |
| **If file size is 0** | 1 | 1 | 1 |
| **Throw a “No data in file error”** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Throw a “Failed to open file” error** | 1 | 1 | 1 |
| **Return vector of Courses.** | 1 | 1 | 1 |
| **Total Cost** | | | 2n^2 + 13n + 8 |
| **Runtime** | | | O(n^2) |

**Advantages / Disadvantages**

Each of the three considered data structures has its strengths and weaknesses.

Starting with an unstructured vector, it is incredibly simple and efficient to insert new courses into the vector, but the process of sorting through them after insertion can often be complicated and time-consuming, especially as it grows in size. For this reason, a vector would be best suited for smaller sets of courses, or if the courses won’t need to be sorted often.

With a well structured Hash Table, the average runtime can fall between 1 and N, making it incredibly efficient, so long as it is well maintained to the size of the data. Assuming the courses need to be accessed via search frequently (as could be expected by a large faculty and student base), the hash table would be an appropriate choice for this need.

If it is well balanced, a BST can be incredibly efficient with an average runtime of log N, but if it’s not well maintained, that runtime can slow down. The insertion and removal of records from a BST can be somewhat more complicated than other data structures, but given that it is well balanced, searching for a specific record within a BST can be incredibly efficient and quick. Because the nature of Binary Search Trees mean they are already sorted in their structure, this eliminates the need to sort the data within them separately, but in the case of storing courses, if the data does not need to be in a sorted format at all times, this could make a BST inefficient when it comes to modifying the data within the tree itself.

**Recommendation**

Considering the various advantages and disadvantages between vectors, hash tables, and binary search trees as listed above, along with the context of how the system being designed will likely be used, the seemingly smartest option for the data structure would be a hash table. Considering course listings at a university only tend to change at most once per term, the insertion function of the data structure is fairly negligible, as it won’t often be used. However, multiple users within a university setting (students, faculty, support staff) would be expected to frequently search through course listings as they plan their educational tracks, meaning a data structure that can efficiently handle search functions should be prioritized, which points to a Hash Table being the most appropriate option in the context of a course listing.