**Dennis T Sherpa 10/10/2024**

**Pseudocode and Runtime Analysis**

For all three data structures the structure that will define a course object remains the same. Therefore, I will provide its pseudocde and runtime analysis before we start:

1. **Pseudocode**

struct Course {

CREATE a string variable named courseNumber

CREATE a string variable named courseName

CREATE a vector of strings named prereqCourseNumbers

}

1. **Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| CREATE a string variables named courseNumber | 1 | 1 | 1 |
| CREATE a string variable named courseName | 1 | 1 | 1 |
| CREATE a vector of strings named prereqCourseNumbers | 1 | 1 | 1 |
| **Total Cost** | | | 3 |
| **Runtime** | | | O(1) |

The following are the pseudocode and runtime analysis regarding reading a file (and everything it entails) and creating course objects, which will be joined together because they occur together in a function, for the vector, hash table, and binary search tree data structures.

1. **Vector** 
   1. Reading the file and creating course objects
      1. Pseudocode
         1. Checks if a prerequisite course exists as a course

bool checkPrereqCourseExists (const string fileName, string prereqCourseNum) {

WHILE getting a line from fileName does not prove False

PARSE each line by commas into tokens

IF the first token matches the prereqCourseNum

RETURN True

RETURN False

}

* + - 1. Loads a file into a vector data structure and checks if a file is formatted correctly

bool loadFile (const string& fileName, vector<Course>& courses) {

OPEN the CSV file named fileName

IF the file is not open

PRINT error message

Return False

CREATE a boolean variable named prereqCourseExists

CREATE a string variable named prereqCourseNum

WHILE getting a line from the file does not prove False

CREATE a Course object called course

PARSE the line into tokens by commas

IF the amount of tokens is less than 2

PRINT “File formatted incorrectly.”

RETURN False

STORE the first token into the course’s courseNumber

STORE the second token into the course’s courseName

IF the amount of tokens is greater than 2

FOR each token FROM the third to the last token

SET prereqCourseNum to the current token

SET prereqCourseExists to the RETURN value from CALL checkPrereqCourseExists (w/ arguments: fileName, prereqCourseNum)

IF prereqCourseExists equals True

PUSH BACK prereqCourseNum into the course’s prereqCourseNumbers vector

ELSE

PRINT “(value of prereqCourseNum) does not exist.”

RETURN False

PUSH BACK course into courses

CLOSE file named fileName

RETURN True

}

* + 1. Runtime Analysis
       1. Checks if a prerequisite course exists as a course

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| WHILE getting a line from the fileName does not prove False | 1 | n | n |
| PARSE each line by commas into tokens | 1 | n | n |
| IF the first token matches the prereqCourseNum | 1 | n | n |
| RETURN True | 1 | 1 | 1 |
| RETURN False | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 2 |
| **Runtime** | | | O(n) |

* + - 1. Loads a file into a vector data structure and checks if a file is formatted correctly

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| OPEN the CSV file named fileName | 1 | 1 | 1 |
| IF the file is not open | 1 | 1 | 1 |
| PRINT error message | 1 | 1 | 1 |
| RETURN False | 1 | 1 | 1 |
| CREATE a boolean variable named prereqCourseExists | 1 | 1 | 1 |
| CREATE a string variable named prereqCourseNum | 1 | 1 | 1 |
| WHILE getting a line from the file does not prove false | 1 | n | n |
| CREATE a Course object called course | 1 | n | n |
| PARSE the line into tokens by commas | 1 | n | n |
| IF the amount of tokens is less than 2 | 1 | n | n |
| PRINT "File formatted incorrectly." | 1 | 1 | 1 |
| RETURN False | 1 | 1 | 1 |
| STORE the first token into the course's courseNumber | 1 | n | n |
| STORE the second token into the course's courseName | 1 | n | n |
| IF the amount of tokens is greater than 2 | 1 | n | n |
| FOR each token FROM the third to last token | 1 |  |  |
| SET prereqCourseNum to the current token | 1 |  |  |
| SET prereqCourseExists to the RETURN value from CALL checkPrereqCourseExists (w/ arguments: fileName, prereqCourseNum) | n |  |  |
| IF prereqCourseExists equals True | 1 |  |  |
| PUSH BACK prereqCourseNum into the course's prereqCourseNumbers vector | 1 |  |  |
| ELSE | 1 |  |  |
| PRINT "(value of prereqCourseNum) does not exist." | 1 |  |  |
| RETURN False | 1 |  |  |
| PUSH BACK course into courses | 1 | n | n |
| CLOSE file named fileName | 1 | 1 | 1 |
| RETURN True | 1 | 1 | 1 |
| **Total Cost** | | | 12 + 8n + 5 + |
| **Runtime** | | | O() |

* 1. Advantages
     1. **Simple Insertion Order:** Elements are stored in the order they are inserted, making traversal straightforward.
     2. **Efficient Index Access:** Direct access to elements using indices, enabling O(1) access for a known index.
     3. **Memory Efficiency:** Elements are stored contiguously, minimizing overhead and maximizing cache efficiency.
     4. **Easy to Sort:** Built-in support for sorting in alphanumeric order using algorithms like **std::sort**.
     5. **Good for Sequential Access:** Ideal for sequential access or when iterating over all elements in a sorted manner.
     6. **Supports Random Access:** Random access is efficient due to contiguous memory allocation.
     7. **Low Overhead:** Simple implementation without the overhead of pointers, unlike trees.
  2. Disadvantages
     1. **Costly Insertions/Deletions:** Inserting or deleting elements (except at the end) can be expensive, O(n) due to shifting.
     2. **Memory Reallocation:** When resized, vectors may need to reallocate memory, which can be expensive.
     3. **Poor Performance for Large Datasets:** Performance may degrade if the dataset is large and frequent insertions are needed.
     4. **No Efficient Search:** Finding a course title or prerequisites in an unsorted vector is O(n).
     5. **No Efficient Duplicate Handling:** Cannot handle duplicate elements or complex keys efficiently.
     6. **High Cost for Sorted Order:** If the vector needs to be maintained in sorted order, every insertion will have O(n) complexity.
     7. **Limited by Size Constraints:** If not dynamically resized, it may quickly exhaust memory or require resizing operations.

1. **Hash Table**
   1. Reading the file and creating course objects
      1. Pseudocode
         1. Calculates a hash value of a key

unsigned int HashTable::hash(int key) {

RETURN (key % HashTable's tableSize)

}

* + - 1. Opens a file

bool HashTable::OpenFile(const string fileName) {

OPEN CSV file named fileName

IF fileName is not open THEN

PRINT error message

RETURN FALSE

RETURN TRUE

}

* + - 1. Checks if a prerequisite course exists as a course

bool HashTable::PrereqCourseExists (const string fileName, string prereqCourseNum) {

WHILE getting a line from fileName does not prove False

PARSE each line by commas into tokens

IF the first token matches the prereqCourseNum

RETURN True

RETURN False

}

* + - 1. Checks a file for format errors

bool HashTable::CheckFile(const string fileName) {

CREATE a boolean variable called prereqCourseExists

CREATE a string variable called prereqCourseNum

WHILE getting a line from the file does not prove False DO

PARSE the line into tokens by commas

IF the amount of tokens is less than 2

PRINT “File formatted incorrectly.”

RETURN FALSE

IF the amount of tokens is greater than 2

FOR each token FROM the third to the last token

SET prereqCourseNum to the current token

SET prereqCourseExists to the RETURN value from CALL PrereqCourseExists (fileName, prereqCourseNum)

IF prereqCourseExists does not equal TRUE THEN

PRINT “prereqCourseNum does not exist.”

RETURN FALSE

RETURN TRUE

}

* + - 1. Inserts a course object into the hash table

void HashTable::Insert(Course course) {

CONVERT course's courseNumber to an int, CALL hash(stoi(course.courseNumber)), and ASSIGN the return value to hashValue (unsigned int).

INITIALIZE a new node pointer, cur.

POINT cur to the node at the hashValue of nodes.

IF cur points to a nullptr THEN

CREATE a new node, newNode, that POINTS to the course and hashValue.

ASSIGN the newNode to the node at the hashValue of nodes.

ELSE IF, the node at the hashValue of nodes does not contain a course THEN

POINT cur's course to the course (passed as an argument),

POINT cur's key to the hashValue (variable containing the hash value)

POINT cur's next pointer to a null pointer.

ELSE, traverse through the linked list until an empty space is found.

POINT the temp node to the head node at the hashValue of nodes.

WHILE temp's next pointer does not point to a null pointer DO

POINT temp to the node its next pointer points to.

SINCE temp is now the tail node, POINT its next pointer to the new node.

POINT the new node's next pointer to a null pointer, making it the new tail node.

RETURN nothing to exit this method.

}

* + - 1. Loads a file into the hash table

void HashTable::LoadFile(const string fileName) {

INITIALIZE a string named prereqCourseNum

WHILE getting a line from the file does not prove False DO

CREATE a Course object called course

PARSE the line into tokens by commas

STORE the first token into the course’s courseNumber

STORE the second token into the course’s courseName

IF the amount of tokens is greater than 2

FOR each token FROM the third to the last token

SET prereqCourseNum to the current token

PUSH BACK prereqCourseNum into the course’s prereqCourseNumbers vector

CALL Insert(course)

RETURN nothing to exit method

}

* + 1. Runtime Analysis
       1. Calculates a hash value of a key

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| RETURN (key % HashTable's tableSize) | 1 | 1 | 1 |
| **Total Cost** | | | 1 |
| **Runtime** | | | O(1) |

* + - 1. Opens a file

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| OPEN CSV file named fileName | 1 | 1 | 1 |
| IF fileName is not open THEN | 1 | 1 | 1 |
| PRINT error message | 1 | 1 | 1 |
| RETURN FALSE | 1 | 1 | 1 |
| RETURN TRUE | 1 | 1 | 1 |
| **Total Cost** | | | 5 |
| **Runtime** | | | O(1) |

* + - 1. Checks if a prerequisite course exists as a course

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| WHILE getting a line from fileName does not prove False | 1 | n | n |
| PARSE each line by commas into tokens | 1 | n | n |
| IF the first token matches the prereqCourseNum | 1 | n | n |
| RETURN True | 1 | 1 | 1 |
| RETURN False | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 2 |
| **Runtime** | | | O(n) |

* + - 1. Checks a file for format errors

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| CREATE a boolean variable called prereqCourseExists | 1 | 1 | 1 |
| CREATE a string variable called prereqCourseNum | 1 | 1 | 1 |
| WHILE getting a line from the file does not prove False DO | 1 | n | n |
| PARSE the line into tokens by commas | 1 | n | n |
| IF the amount of tokens is less than 2 | 1 | n | n |
| PRINT “File formatted incorrectly.” | 1 | 1 | 1 |
| RETURN FALSE | 1 | 1 | 1 |
| IF the amount of tokens is greater than 2 | 1 | n | n |
| FOR each token FROM the third to the last token | 1 |  |  |
| SET prereqCourseNum to the current token | 1 |  |  |
| SET prereqCourseExists to the RETURN value from CALL prereqCourseExists (fileName, prereqCourseNum) | n |  |  |
| IF prereqCourseExists does not equal TRUE THEN | 1 |  |  |
| PRINT "prereqCourseNum does not exist." | 1 | 1 | 1 |
| RETURN FALSE | 1 | 1 | 1 |
| RETURN TRUE | 1 | 1 | 1 |
| **Total Cost** | | | 7 + 4n + 3 + |
| **Runtime** | | | O() |

* + - 1. Inserts a course object into the hash table

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| CONVERT course's courseNumber to an int, CALL hash(stoi(course.courseNumber)), and ASSIGN the return value to hashValue (unsigned int). | 1 | 1 | 1 |
| INITIALIZE a new node pointer, cur. | 1 | 1 | 1 |
| POINT cur to the node at the hashValue of nodes. | 1 | 1 | 1 |
| IF cur points to a nullptr THEN | 1 | 1 | 1 |
| CREATE a new node, newNode, that POINTS to the course and hashValue. | 1 | 1 | 1 |
| ASSIGN the newNode to the node at the hashValue of nodes. | 1 | 1 | 1 |
| ELSE IF, the node at the hashValue of nodes does not contain a course THEN | 1 | 1 | 1 |
| POINT cur's course to the course (passed as an argument), | 1 | 1 | 1 |
| POINT cur's key to the hashValue (variable containing the hash value) | 1 | 1 | 1 |
| POINT cur's next pointer to a null pointer. | 1 | 1 | 1 |
| ELSE, (comment: traverse through the linked list until an empty space is found.) | 1 | 1 | 1 |
| POINT the temp node to the head node at the hashValue of nodes. | 1 | 1 | 1 |
| WHILE temp's next pointer does not point to a null pointer DO | 1 | n | n |
| POINT temp to the node its next pointer points to. | 1 | n | n |
| SINCE temp is now the tail node, POINT its next pointer to the new node. | 1 | 1 | 1 |
| POINT the new node's next pointer to a null pointer, making it the new tail node. | 1 | 1 | 1 |
| RETURN nothing to exit this method. | 1 | 1 | 1 |
| **Total Cost** | | | 15 + 2n |
| **Runtime** | | | O(n) |

* + - 1. Loads a file into the hash table

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| INITIALIZE a string named prereqCourseNum | 1 | 1 | 1 |
| WHILE getting a line from the file does not prove False DO | 1 | n | n |
| CREATE a Course object called course | 1 | n | n |
| PARSE the line into tokens by commas | 1 | n | n |
| STORE the first token into the course's courseNumber | 1 | n | n |
| STORE the second token into the course's courseName | 1 | n | n |
| IF the amount of tokens is greater than 2 | 1 | n | n |
| FOR each token FROM the third to the last token | 1 |  |  |
| SET prereqCourseNum to the current token | 1 |  |  |
| PUSH BACK prereqCourseNum into the course's prereqCourseNumbers vector | 1 |  |  |
| CALL Insert(course) | n |  |  |
| RETURN nothing to exit method | 1 | 1 | 1 |
| **Total Cost** | | | 2 + 6n + 4 |
| **Runtime** | | | O() |

* 1. Advantages
     1. **Fast Lookups:** Access elements in O(1) average time complexity for searching and retrieving course data.
     2. **Efficient Insertions/Deletions:** Insertion and deletion operations are also O(1) on average.
     3. **Handles Large Datasets Well:** Can store and retrieve large datasets efficiently.
     4. **No Order Constraints:** Efficiently retrieves course titles and prerequisites without maintaining sorted order.
     5. **Handles Complex Keys:** Supports key-based access, making it easy to retrieve specific course data.
     6. **Handles Duplicates Using Separate Chaining:** Can handle multiple courses with similar identifiers using techniques like chaining.
     7. **Collision Resolution Mechanisms:** Provides various collision resolution strategies, such as open addressing or chaining.
  2. Disadvantages
     1. **Unordered:** Cannot directly print courses in alphanumeric order without additional sorting, which would be O(n log n).
     2. **Memory Overhead:** Requires additional memory for storing keys, values, and pointers for collision resolution.
     3. **Collisions Can Reduce Efficiency:** Poorly distributed keys can lead to collisions, reducing performance to O(n).
     4. **Not Cache Friendly:** Hash tables are often scattered in memory, making them less cache-friendly compared to vectors.
     5. **Rehashing Can Be Expensive:** When the table grows, rehashing can be computationally expensive.
     6. **Requires a Good Hash Function:** A poorly chosen hash function can result in high collision rates.
     7. **Complex Implementation:** More complex to implement and maintain compared to other data structures.

1. **Binary Search Tree**
   1. Reading the file and creating course objects
      1. Pseudocode
         1. (PU 1) Default Constructor

A default constructor for a Binary Search Tree:

SET the root node (V 1) to null

* + - 1. (PU 2) Loads a file into a Binary Search Tree

A method, returning nothing, that will load all courses in a file into a Binary Search Tree (w/ parameters: file name):

INITIALIZE a string named prereqCourseNum

WHILE getting a line from the file does not prove false

CREATE a new Course object called course

PARSE the line into tokens by commas

STORE the first token into this course’s courseNumber

STORE the second token into this course’s courseName

IF the amount of tokens is greater than 2

FOR each token FROM the third to the last token

SET prereqCourseNum to the current token

PUSH BACK prereqCourseNum into this course’s prereqCourseNumbers vector

CALL the method (PU 5) to insert this course into a Binary Search Tree (w/ parameter: this iteration's course object)

* + - 1. (PU 3) Checks if a prerequisite course exists as a course

A method, returning a boolean value, that will check if a prerequisite course exists as a course (w/ parameters: file name, prerequisite course number):

WHILE getting a line from the file (labeled the file name (argument)) does not prove FALSE

PARSE each line by commas into tokens

IF the first token matches the prerequisite course number (argument)

RETURN True

RETURN False

* + - 1. (PU 4) Checks if a file is formatted correctly

A method, returning a boolean value, that will check if a file is formatted correctly (w/ parameter: file name):

CREATE a boolean variable called prereqCourseExists

CREATE a string variable called prereqCourseNum

WHILE getting a line from the file does not prove false

PARSE the line into tokens by commas

IF the amount of tokens is less than 2

PRINT “File formatted incorrectly.”

RETURN False

IF the amount of tokens is greater than 2

FOR each token FROM the third to the last token

SET prereqCourseNum to the current token

SET prereqCourseExists to the RETURN value from CALLING the method (PU 3), which checks if a prerequisite course exists as a course (w/ arguments: file name, current prerequisite course number)

IF prereqCourseExists does not equal True

PRINT "(current prerequisite course number) does not exist.”

RETURN False

RETURN True

* + - 1. (PU 5) Inserts a node into a Binary Search Tree

A method, returning nothing, that will insert a node (containing a course) into a Binary Search Tree as a root, internal, or leaf node (w/ parameter: a course):

IF the root node (V 1) equals null

SET the root node (V 1) to a new node containing a course (argument)

ELSE

CALL the method (PV 1) that will add a node to a Binary Search Tree (w/ arguments: root node and a course (argument))

* + - 1. (PV 1) Adds a node to a Binary Search Tree

A private method of the Binary Search Tree class, returning nothing, to add a node to the Binary Search Tree as an internal or leaf node (w/ parameters: a node pointer (root node), a course object):

IF node's (argument) course number is greater than the course (argument) course number

IF node's (argument) left child equals null

ASSIGN node's (argument) left child with a new node containing course (argument)

ELSE

RECURSIVELY CALL this method (PV 1) by passing node's (argument) left child and course (argument)

ELSE

IF node's (argument) right child equals null

ASSIGN node's (argument) right child with a new node containing course (argument)

ELSE

RECURSIVELY CALL this method (PV 1) by passing node's (argument) right child and course (argument)

* + - 1. (PU 7) Opens a CSV file

A method, returning a boolean value, that will open a CSV file (w/ argument: file name):

OPEN CSV file named the file name (argument)

IF the file name (argument) is not open

PRINT error message

RETURN FALSE

RETURN TRUE

* + - 1. (PU 8) Closes a CSV file

A method, returning a boolean value, that will close a CSV file (w/ argument: file name):

CLOSE CSV file named the file name (argument)

IF the file name (argument) is closed

PRINT error message

RETURN TRUE

RETURN FALSE

* + 1. Runtime Analysis
       1. (PU 1) Default Constructor

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| SET the root node to null | 1 | 1 | 1 |
| **Total Cost** | | | 1 |
| **Runtime** | | | O(1) |

* + - 1. (PU 2) Loads a file into a Binary Search Tree

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| INITIALIZE a string named prereqCourseNum | 1 | 1 | 1 |
| WHILE getting a line from the file does not prove false | 1 | n | n |
| CREATE a new Course object called course | 1 | n | n |
| PARSE the line into tokens by commas | 1 | n | n |
| STORE the first token into this course’s courseNumber | 1 | n | n |
| STORE the second token into this course’s courseNumber | 1 | n | n |
| IF the amount of tokens is greater than 2 | 1 | n | n |
| FOR each token FROM the third to the last token | 1 |  |  |
| SET prereqCourseNum to the current token | 1 |  |  |
| PUSH BACK prereqCourseNum into this course’s prereqCourseNumbers vector | 1 |  |  |
| CALL the method (PU 5) to insert this course into a Binary Search Tree (w/ parameters: this iteration’s course object) | 1 | 1 | 1 |
| **Total Cost** | | |  |
| **Runtime** | | | O() |

* + - 1. (PU 3) Checks if a prerequisite course exists as a course

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| WHILE getting a line from the file (labeled the file name(argument)) does not prove FALSE | 1 | n | n |
| PARSE each line by comma into tokens | 1 | n | n |
| IF the first token matches the prerequisite course number (argument) | 1 | n | n |
| RETURN True | 1 | 1 | 1 |
| RETURN False | 1 | 1 | 1 |
| **Total Cost** | | | 3n + 2 |
| **Runtime** | | | O(n) |

* + - 1. (PU 4) Checks if a file is formatted correctly

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| CREATE a boolean variable called prereqCourseExists | 1 | 1 | 1 |
| CREATE a string variable called prereqCourseNum | 1 | 1 | 1 |
| WHILE getting a line from the file does not prove false | 1 | n | n |
| PARSE the line into tokens by commas | 1 | n | n |
| IF the amount of tokens is less than 2 | 1 | n | n |
| PRINT "File formatted incorrectly. | 1 | 1 | 1 |
| RETURN False | 1 | 1 | 1 |
| IF the amount of tokens is greater than 2 | 1 | n | n |
| FOR each token FROM the third to the last token | 1 |  |  |
| SET prereqCourseNum to the current token | 1 |  |  |
| SET prereqCourseExists to the RETURN value from CALLING the method (PU 3), which checks if a prerequisite course exists as a course (w/ arguments: file name, current prerequisite course number) | n |  |  |
| IF prereqCourseExists does not equal True | 1 |  |  |
| PRINT "(current prerequisite course number) does not exist." | 1 | 1 | 1 |
| RETURN False | 1 | 1 | 1 |
| RETURN True | 1 | 1 | 1 |
| **Total Cost** | | | 7 + 4n + 3 + |
| **Runtime** | | | O() |

* + - 1. (PU 5) Inserts a node into a Binary Search Tree

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| IF the root node (V 1) equals null | 1 | 1 | 1 |
| SET the root node (V 1) to a new node containing a course (argument) | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| CALL the method (PV 1) that will add a node to a Binary Search Tree (w/ arguments: root node and a course (argument)) | n | 1 | n |
| **Total Cost** | | | 3 + n |
| **Runtime** | | | O(n) |

* + - 1. (PV 1) Adds a node to a Binary Search Tree

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| IF node's (argument) course number is greater than the course (argument) course number | 1 | 1 | 1 |
| IF node's (argument) left child equals null | 1 | 1 | 1 |
| ASSIGN node's (argument) left child with a new node containing course (argument) | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| RECURSIVELY CALL this method (PV 1) by passing node's (argument) left child and course (argument)  \*\*Line cost equals n assuming the depth of the BST on the left side is n. | n | 1 | n |
| ELSE | 1 | 1 | 1 |
| IF node's (argument) right child equals null | 1 | 1 | 1 |
| ASSIGN node's (argument) right child with a new node containing course (argument) | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| RECURSIVELY CALL this method (PV 1) by passing node's (argument) right child and course (argument)  \*\*Line cost equals n assuming the depth of the BST on the right side is n. | n | 1 | n |
| **Total Cost** | | | 8 + 2n |
| **Runtime** | | | O(n) |

* + - 1. (PU 7) Opens a CSV file

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| OPEN CSV file named the file name (argument) | 1 | 1 | 1 |
| IF the file name (argument) is not open | 1 | 1 | 1 |
| PRINT error message | 1 | 1 | 1 |
| RETURN FALSE | 1 | 1 | 1 |
| RETURN TRUE | 1 | 1 | 1 |
| **Total Cost** | | | 5 |
| **Runtime** | | | O(5) |

* + - 1. (PU 8) Closes a CSV file

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **Number of Times Executed** | **Total Cost** |
| CLOSE CSV file named the file name (argument) | 1 | 1 | 1 |
| IF the file name (argument) is closed | 1 | 1 | 1 |
| PRINT error message | 1 | 1 | 1 |
| RETURN TRUE | 1 | 1 | 1 |
| RETURN FALSE | 1 | 1 | 1 |
| **Total Cost** | | | 5 |
| **Runtime** | | | O(1) |

* 1. Advantages
     1. **Sorted Order:** Automatically maintains elements in sorted order, perfect for the requirement to print courses in alphanumeric order.
     2. **Efficient Search Operations:** Searching for a course title or prerequisites is O(log n) in a balanced BST.
     3. **Supports Range Queries:** Easy to find a range of courses or perform operations like "print courses in a range."
     4. **Balanced Trees are Efficient:** AVL or Red-Black trees ensure O(log n) for insertions, deletions, and lookups.
     5. **Handles Duplicates:** Can be customized to handle duplicate course identifiers.
     6. **Easy Traversal:** Inorder traversal provides sorted order access for printing course titles.
     7. **Flexible Structure:** Supports dynamic insertions and deletions without needing to rehash or reallocate memory.
  2. Disadavantages
     1. **Imbalanced Trees Degrade Performance:** An unbalanced BST can degrade to O(n) operations, similar to a linked list.
     2. **Higher Memory Overhead:** Each node requires memory for pointers, unlike vectors.
     3. **Complex Implementation:** More complex to implement and maintain, especially for balancing (e.g., AVL, Red-Black trees).
     4. **Costly Rotations:** Insertion or deletion in self-balancing trees may trigger costly rotations.
     5. **Requires Consistent Balancing:** Must maintain balance to ensure optimal performance.
     6. **Not Cache Friendly:** Scattered nodes in memory reduce cache performance.
     7. **Difficult for Non-Unique Keys:** Handling duplicate keys or complex course identifiers can complicate the structure.

**Data Structure Recommendation**

My recommended data structure which I plan to use in my code: **Binary Search Tree**

My reasoning is as follows:

Given the advisor's requirements of printing a list of all computer science courses in alphanumeric order and providing quick access to course titles and prerequisites, a **Binary Search Tree (BST)** is the most appropriate data structure. The primary advantage of using a BST is its ability to maintain elements in sorted order by default, which is crucial for satisfying the alphanumeric sorting requirement. With a balanced implementation like an AVL or Red-Black tree the BST ensures O(log n) time complexity for insertion, deletion, and search operations (Wikimedia Foundation. (2024, January 27)). This allows the structure to not only store the course data but also provide efficient lookups for prerequisites. In contrast, using a data structure like a hash table would require additional sorting steps, which can introduce unnecessary computational overhead.

Furthermore, a balanced BST’s ability to perform in-order traversal naturally outputs elements in a sorted manner, making it ideal for scenarios where the primary task is to display all the course titles sequentially. Additionally, a BST efficiently maps each course to its corresponding data by utilizing the course identifier as the key and storing associated metadata (such as titles and prerequisites) in the node. This makes it straightforward to retrieve and print all relevant information for a specific course. The structured nature of a BST also supports range queries, making it flexible for extending the program to handle more complex queries in the future, such as retrieving a subset of courses.

While hash tables offer O(1) average time complexity for direct lookups, they lack inherent support for maintaining elements in sorted order. Thus, if the primary requirement is to frequently access course data in an ordered sequence, a hash table would necessitate either additional sorting or maintaining a parallel data structure. Similarly, while providing good support for sequential access, vectors suffer from O(n) time complexity when inserting or deleting elements in sorted order, making them unsuitable for dynamic datasets that may change frequently. Overall, a balanced BST strikes the best balance between efficient sorted storage and dynamic operations, effectively fulfilling all of the advisor's requirements.

**References**

Wikimedia Foundation. (2024, January 27). *Self-balancing binary search tree*. Wikipedia.

https://en.wikipedia.org/wiki/Self-balancing\_binary\_search\_tree