## Load Courses Pseudocode

### loadCourses Pseudocode

\* Note: additional functions for Hash Table and Binary Search Trees found in Appendix A: Insert Functions used in loadCourses() on page 11.

START function loadCourses(string csvFilePath, ADT courseList)

CREATE parser object, for iterating through the file

FOR each line, csvLine, in csv file

CREATE new temporary course item, curCourse

SET variable, lineSize, to number of items found in line

SET curCourse ID as item 0 in csvLine

IF lineSize < 2

OUTPUT “course [curCourse ID] is missing a name; this course was not loaded”

CONTINE to next line

ELSE

SET curCourse name as item 1 in csvLine

END IF

IF size >2

FOR items at csvLine positions 2 to lineSize

CREATE Course, tempCourse with ID equal to the current prerequisite courseID

PUSHBACK to add tempCourse to curCourse prerequisite vector

END FOR

END IF

CALL ADT-specific function, insert curCourse to courseList (use pushback for vector)

END FOR

END function loadCourses(ADT)

## Print Course List Pseudocode

\* Note: because of the differing structures of vectors, hash tables, and binary search trees, each ADT has a different printCourses function. In main() the printCourses function called will be dependent on the ADT used.

### Vector Pseudocode

START printSampleSchedule (Vector courseVector)

CREATE Vector sortedCourseVector

QUICKSORT courseVector, output to sortedCourseVector

FOR each item in sortedCourseVector

OUTPUT course ID and name

IF course prerequisite vector size is not zero

FOR each item in prereqs vector

OUTPUT course ID

END FOR

END

END FOR

END printSampleSchedule (Vector courseVector)

### Hash Table Pseudocode

\* Note, for this function, items from the hash table are transferred to a binary search tree so they can be sorted. The insert function references the Insert() function for binary search trees found on page 12 in Appendix A. The inOrder function references the binary search tree function found on page 3, in the next section.

START function printSampleSchedule(HashTable courseList)

CREATE a binary search tree object, courseBst

CREATE a node, curNode for iterating across HashTable keys

FOR all nodes in courseList

IF node in hash table does not have default (empty) key

CALL binary search tree Insert function to insert current node’s course into courseBst

SET curNode to the next node in the table

WHILE curNode is not null

CALL binary search tree Insert function to insert current node’s course into courseBst

SET curNode to its next node

END WHILE

END IF

END FOR

CALL BST function inOrder, passing root of courseBst

END function printSampleSchedule (HashTable courseList)

### Binary Search Tree Pseudocode

START function inOrder(Node\* node)

IF (node is null)

RETURN

END IF

RECURSE inOrder of pointer to left of node

OUTPUT node course ID

IF course prerequisite vector size is not zero

FOR each item in prereqs vector

OUTPUT course ID

END FOR

END

RECURSE inOrder of pointer to right of node

END inOrder

START printSampleSchedule (BST courseList)

CALL function inOrder, passing root of courseList as argument

END printSampleSchedule (courseList)

## Main Menu Pseudocode

\*Note: Function pseudocode for menu options 1 and 2 are found within this document. Pseudocode for option 3 is found in module milestone documents, under function name “printCourse”

START main()

CREATE ADT object to hold course information, courseList

CREATE empty course object, course

CREATE integer, menuSelect to hold menu selections

WHILE menuSelect is not equal to 9

OUTPUT “Menu”

OUTPUT “ 1. Load Courses”

OUTPUT “ 2. Print sorted course list”

OUTPUT “ 3. Print course”

OUTPUT “ 9. Exit program”

SET user input to variable menuSelect

SWITCH using menuSelect as argument

CASE 1, if user selects “Load Courses”

OUTPUT: “Please enter csv file path with course information”

GET file path, filePath from user input

CALL function loadCourses, with argument courseList

CASE 2, if user selects “Print sorted course list”

IF courseList is empty

OUTPUT “Select ‘1. Load Courses’ prior to printing”

ELSE

CALL function printSampleSchedule with argument courseList

END IF

CASE 3, if user selects “Print Course”

IF courseList is empty

OUTPUT “Select ‘1. Load Courses’ prior to printing”

ELSE

OUTPUT “Enter course ID to print:”

SET string variable inputId as user input

CALL function printCourseInformation

END IF

END SWITCH

END WHILE

OUTPUT “Leaving program, goodbye”

END main()

## Runtime Analysis

### Worst case: runtime analysis of non-ADT-specific loadCourses()

For the worst-case scenario of load courses, all courses have a course name, and all courses have N-1 prerequisites. The base case substitutes X for the line cost of the ADT-specific insert function. All portions of the pseudocode evaluated for the worst case scenario are written in red below:

START function loadCourses(string csvFilePath, ADT courseList)

FOR each line, csvLine, in csv file

CREATE new temporary course item, curCourse

SET variable, lineSize, to number of items found in line

SET curCourse ID as item 0 in csvLine

IF lineSize < 2

OUTPUT “course [curCourse ID] is missing a name; it was not loaded from file”

CONTINE to next line

ELSE

SET curCourse name as item 1 in csvLine

END IF

IF size >2

FOR items at csvLine positions 2 to lineSize

CREATE Course, tempCourse with ID equal to the current prerequisite

PUSHBACK to add tempCourse to curCourse prerequisite vector

END FOR

END IF

CALL ADT-specific function, insert curCourse to courseList (use pushback for vector)

END FOR

END function loadCourses(ADT)

Table : Runtime analysis for base case

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Executions** | | **Total Cost** |
| FOR each line, csvLine, in csv file | 2 | N | | 2N |
| CREATE new temporary course item, curCourse | 1 | N | | N |
| SET variable, lineSize, to number of items found in line | 1 | N | | N |
| SET curCourse ID as item 0 in csvLine | 1 | N | | N |
| IF lineSize < 2 | 1 | N | | N |
| SET curCourse name as item 1 in csvLine | 1 | N | | N |
| IF size >2 | 1 | N | | N |
| FOR items at csvLine positions 2 to lineSize | 2 |  | | N2-1 |
| CREATE Course, tempCourse with ID equal to the current prerequisite courseID | 1 |  | |  |
| PUSHBACK to add tempCourse to curCourse prerequisite vector | 1 |  | |  |
| CALL ADT-specific function, insert curCourse to courseList (use pushback for vector) | X | N | | N\*X |
| Total Cost | | |  | |
| Runtime | | | O(N2) | |

­Now, including the analysis (X) for the insert functions:

**For a vector:**

The complexity of the push\_back function used is constant (AyushSaxena, 2023), so using pushback to add a course to the course list would be equal to N (+1 per iteration of loop sized N), so the worst case analysis would become

In other words, adding the push\_back function for the vector case does not change the Big O complexity of the loadCourses function.

**For a hash table:**

For the worst-case runtime analysis for the hash table, only the text in red in the pseudocode below was used because in the worst case scenario, every node has collision. This is perhaps not quite realistic because in the case of a completely empty hash table being filled with values, there should be at least one empty node to start. The insertion of a node at a key is estimated to have a linear (N) growth rate, while the addition at a collision has a quadratic growth rate (N2). Since a linear growth rate for several nodes will likely be negligible compared to the quadratic growth rate of other nodes, it is a reasonable assumption that the addition of new nodes can be ignored in the runtime analysis. Likewise, the addition of a course into an old pre-occupied node is a constant growth rate, so this also is not considered for this analysis.

START function Insert(HashTable hashTable, Course addCourse):

SET key equal to the hash value of addCourse’s course ID

CREATE node object, oldNode

SET oldNode equal to the node found at key

IF oldNode is null

CREATE node, newNode for the course at key

INSERT newNode into the node structure at key

ELSE IF the old node is not used anymore

SET oldNode’s key to the calculated key

SET oldNode’s course to addCourse

SET oldNode’s next pointer to the null pointer

ELSE

WHILE the node after oldNode is not null

SET old node to its subsequent node

END WHILE

SET the node after oldNode to a new node with addCourse and the calculated key

END IF

END function

Table : Runtime analysis for Hash Table Insert Function

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Executions** | **Total Cost** |
| SET key equal to the hash value of addCourse’s course ID | 1 | 1 | 1 |
| CREATE node object, oldNode | 1 | 1 | 1 |
| SET oldNode equal to the node found at key | 1 | 1 | 1 |
| IF oldNode is null | 1 | 1 | 1 |
| ELSE IF the old node is not used anymore | 1 | 1 | 1 |
| WHILE the node after oldNode is not null | 1 | N+1 | N+1 |
| SET old node to its subsequent node | 1 | N | N |
| SET the node after oldNode to a new node with addCourse and the calculated key | 1 | 1 | 1 |
| Total Cost | | | 2N+7 |

Added to the base case, this function becomes:

**For a binary search tree:**

For the worst case analysis for the hash table, only the text in red in the pseudocode below was used. This represents the runtime where the list being loaded into the binary search table is already organized in alphameric or reverse alphanumeric order.

START addNode(Node\* insertNode, Course addCourse)

IF insertNode’s course’s courseId is greater than addCourse’s courseId

IF the node to the left of insertNode is null

SET the node to the left of insertNode equal to a new node containing addCourse

ELSE

RECURSE addNode with insertNode equal to its left node, and addCourse

END IF

END IF

IF insertNode’s course’s courseId is less than or equal to than course’s courseId

IF the node to the right of insertNode is null

SET the node to the right of insertNode equal to a new node containing addCourse

ELSE

RECURSE addNode with insertNode equal to its right node, and the bid argument

END IF

END IF

END function addNode

START function Insert(BinarySearchTree courseTree, Course addCourse)

IF root is null

CREATE a new node with addCourse at root

ELSE

CALL function addNode, passing courseTree’s root and addCourse as arguments

END IF

END function Insert

Table : Runtime analysis for binary search tree Insert function

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Executions** | **Total Cost** |
| IF root is null | 1 | 1 | 1 |
| IF insertNode’s course’s courseId is greater than addCourse’s courseId | 1 | 1 | 1 |
| IF the node to the left of insertNode is null | 1 | 1 | 1 |
| RECURSE addNode with insertNode equal to its left node, and addCourse | 1 | N/2 | N/2 |
| IF insertNode’s course’s courseId is less than or equal to than course’s courseId | 1 | 1 | 1 |
| IF insertNode’s course’s courseId is less than or equal to than course’s courseId | 1 | 1 | 1 |
| IF the node to the right of insertNode is null | 1 | 1 | 1 |
| RECURSE addNode with insertNode equal to its right node, and the bid argument | 1 | N/2 | N/2 |
| Total Cost | | | N+6 |
| Runtime | | | O(N) |

Added to the base case, this function becomes:

## Discussion

All three data structures evaluated load courses with a growth rate of O(N2). The composite functions between each data structure indicate that vectors may have the fastest runtime and hash tables may have the longest runtime, but ultimately, for a list of any given size, the three algorithms are anticipated to have a similar runtime. Because the vector pushback function has a constant growth function, it is likely that the courses will be loaded into a vector slightly mor quickly than a binary search tree or hash table. However, I considered the growth rate of several other functions before determining which ADT is best overall. The following table shows how each functions worst case growth rate compares across the three focus ADTs. The methods for calculating each growth rate can be found in Appendix B: Worst Case Growth Rate Analysis for printCourseInformation and Appendix C: Worst Case Growth Rate Analysis for printSampleSchedule

Table : Big-O notation for worst case growth rate analyses performed for vectors, hash tables, and binary search trees

|  |  |  |  |
| --- | --- | --- | --- |
|  | **loadCourses** | **printSampleSchedule** | **printCourseInformation** |
| **Vector** | O(N2) | O(N2) | O(N2) |
| **HashTable** | O(N2) | O(N2) | O(N) |
| **Binary Search Tree** | O(N2) | O(N2) | O(N) |

Although all three data structures appear to have approximately equal worst-case runtimes, the are not altogether equal. The vector has a worst-case quadratic growth rate (Big-O value of O(N2)). Additionally, the hash table is unable to be sorted so it must be converted to a binary search tree to then print the courses in alphanumeric order for printSampleSchedule.

Vectors, in comparison with arrays, are useful in cases like this where the size of the vector is initially unknown. This makes it quite easy to add and remove items to the vector without wasting space. They are also quite straightforward to iterate over since each item is stored sequentially. A disadvantage of vectors (in contrast with binary search trees) is they are not inherently sorted. To access the vector values in alphanumeric order, as is the case in this project, a sorting algorithm must be performed. In the case of printCourseInformation, I chose to use Quicksort to sort the vector of courses to avoid linear searching, which can very inefficient in the worst-case scenario. Quicksort itself has a quadratic worst-case growth rate (Vahid F. , Lysecky, Wheatland, & Siu, 2019), which is ultimately why the vector has the largest Big-O value for printCourseInformation.

A major advantage hash tables generally have is the search speed based on the way items are mapped. When collisions do not occur, hash tables can have a constant growth rate for insertion, deletion, and searching for items (Vahid F. , Lysecky, Wheatland, & Siu, 2019). Even when collisions do occur, the table can use probing or chaining to complete these operations relatively quickly (although as a table fills up and collisions become more frequent, the complexity of the program will increase significantly). A disadvantage of hash tables, as previously mentioned, is the inability to sort them. Since they are mapped based on a hash value, their order will not be alphanumeric. Since 1 out of the 3 key functions relies on the items to be printed alphanumerically, this should discount hash tables as a contender for the project.

Although the worst-case-scenario does not reflect this, binary search trees have a complexity advantage over vectors as they approach being perfect. In a perfect binary search tree with 100 courses, the maximum number of searches required to find a course would be 7 (logN) (Vahid F. , Lysecky, Wheatland, & Siu, 2019). Binary search trees are inherently structured in a way that makes accessing items in alphanumeric order very straightforward and efficient. As previously mentioned, this is a huge advantage they have over hash tables and vectors (and why, ultimately, hash tables were converted into binary search trees to print the sample schedule). A downside of binary search trees is their efficiency dependency on the node selected for the root. As the root gets further away from the median value of the list, the tree will get more and more imperfect.

I am still getting used to understanding how to calculate runtime complexities, so I am not entirely confident in my conclusion based on Big-O value alone that binary search trees are the most optimal ADT to work with. However, looking at other advantages that binary search trees have over hash tables and vectors in this context reaffirm that they are the right choice for this project.

# Works Cited

AyushSaxena. (2023, February 16). *vector::push\_back() and vector::pop\_back() in C++ STL*. Retrieved August 4, 2023, from Geeks for Geeks: https://www.geeksforgeeks.org/vectorpush\_back-vectorpop\_back-c-stl/

Bajaj, T. (2023, May 06). *vector insert() Function in C++ STL*. Retrieved August 04, 2023, from Geeks for Geeks: https://www.geeksforgeeks.org/vector-insert-function-in-cpp-stl/

Vahid, F., Lysecky, S., Wheatland, N., & Siu, R. (2019). 3.21 Overview of fast sorting algorithms. In *CS 300: Data Structures and Algorithms.* Zyante Inc.

Vahid, F., Lysecky, S., Wheatland, N., & Siu, R. (2019). *CS 300: Data Structures and Algorithms.* Zyante Inc.

## Appendix A: Insert Functions used in loadCourses()

### Hash Table Insert Pseudocode

START function Insert(HashTable hashTable, Course addCourse):

SET key equal to the hash value of addCourse’s course ID

CREATE node object, oldNode

SET oldNode equal to the node found at key

IF oldNode is null

CREATE node, newNode for the course at key

INSERT newNode into the node structure at key

ELSE IF the old node is not used anymore

SET oldNode’s key to the calculated key

SET oldNode’s course to addCourse

SET oldNode’s next pointer to the null pointer

ELSE

WHILE the node after oldNode is not null

SET old node to its subsequent node

END WHILE

SET the node after oldNode to a new node with addCourse and the calculated key

END IF

END function

### Binary Search Tree Insert Pseudocode

START addNode(Node\* insertNode, Course addCourse)

IF insertNode’s course’s courseId is greater than addCourse’s courseId

IF the node to the left of insertNode is null

SET the node to the left of insertNode equal to a new node containing addCourse

ELSE

RECURSE addNode with insertNode equal to its left node, and the same addCourse

END IF

END IF

IF insertNode’s course’s courseId is less than or equal to than course’s courseId

IF the node to the right of insertNode is null

SET the node to the right of insertNode equal to a new node containing addCourse

ELSE

RECURSE addNode with insertNode equal to its right node, and the same bid argument

END IF

END IF

END function addNode

START function Insert(BinarySearchTree courseTree, Course addCourse)

IF root is null

CREATE a new node with addCourse at root

ELSE

CALL function addNode, passing courseTree’s root and addCourse as arguments

END IF

END function Insert

## Appendix B: Worst Case Growth Rate Analysis for printCourseInformation ()

### Vector

START function: void printCourseInformation (vector<Course> courseVector, string, courseId)

CREATE Vector, sortedCourseVector

QUICKSORT courseVector, return to sortedCourseVector

SET variable, high, to sortedCourseVector size – 1

SET variable, low, to 0

WHILE high is greater than low

SET mid to the average of high and low

IF sortedCourseVector’s courseId at index mid is less than courseId

SET low to mid + 1

ELSE IF sortedCourseVector’s courseId at index mid is greater than courseId

SET high to mid – 1

ELSE

OUTPUT sortedCourseVector’s information at index mid

IF curNode’s course’s prerequisite list is not null

FOR each item in the prerequisite list

OUTPUT current prerquisite’s course’s course ID

END FOR

END IF

RETURN

END IF

END WHILE

OUTPUT “no course found with course ID, courseId”

END function printCourseInformation

START Quicksort method

SET midpoint to zero

IF Bids vector size is less than or equal to one

END method

ELSE IF end index is less than begin index

END method

END IF

SET midpoint to partition of bids from beginning index to end of index

CALL quicksort (self) on bids from begin index to midpoint index

CALL quicksort (self) on bids from midpoint index + 1 to end index

END quick sort method

Table :Worst-case runtime analysis for vector printCourseInformation method

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Code** | **Line Cost** | | **# Executions** | **Total Cost** |
| CREATE Vector, sortedCourseVector | 1 | | 1 | 1 |
| QUICKSORT courseVector, return to sortedCourseVector | N2 (Vahid F. , Lysecky, Wheatland, & Siu, 2019) | | 1 | N2 |
| SET variable, high, to sortedCourseVector size – 1 | 1 | | 1 | 1 |
| SET variable, low, to 0 | 1 | | 1 | 1 |
| WHILE high is greater than low | 1 | | N+1 | N+1 |
| SET mid to the average of high and low | 1 | |  |  |
| IF sortedCourseVector’s courseId at index mid is less than courseId | 1 | |  |  |
| ELSE IF sortedCourseVector’s courseId at index mid is greater than courseId | 1 | |  |  |
| SET high to mid – 1 | 1 | |  |  |
| OUTPUT sortedCourseVector’s information at index mid | 1 | | 1 | 1 |
| IF curNode’s course’s prerequisite list is not null | 1 | | 1 | 1 |
| FOR each item in the prerequisite list | 2 | | N-1 | 2N-2 |
| OUTPUT current prerquisite’s course’s course ID | 1 | | N-1 | N-1 |
| Total Cost | |  | | |
| Runtime | | O(N2) | | |

### Hash Table

START function: void printCourseInformation (courseId)

SET key equal to the hash value of courseId

CREATE new node, printNode

SET printNode equal to the node found at key

IF printNode is not null, its key isn’t the default key and its course ID is equal to courseId

OUTPUT course info for printNode

RETURN

ELSE IF printNode is null or printNodes key is the default key

OUTPUT “No course found with this ID”

RETURN

END IF

WHILE printNode is not null

IF printNode’s key is not the default key and its course ID is equal to courseId

OUTPUT course ID and name for printNode

IF printNode’s course’s prerequisite list is not null

FOR each item in the prerequisite list

OUTPUT current prerquisite’s course’s course ID

END FOR

END IF

RETURN

ELSE

SET printNode equal to its successor

END IF

END WHILE

END function printCourseInformation

Table : Worst-case runtime analysis for Hash Table printCourseInformation() function

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Executions** | **Total Cost** |
| SET key equal to the hash value of courseId | 1 | 1 | 1 |
| CREATE new node, printNode | 1 | 1 | 1 |
| SET printNode equal to the node found at key | 1 | 1 | 1 |
| IF printNode is not null, its key isn’t the default key and its course ID is equal to courseId | 3 | 1 | 3 |
| ELSE IF printNode is null or printNodes key is the default key | 2 | 1 | 2 |
| WHILE printNode is not null | 1 | N+1 | N+1 |
| IF printNode’s key is not the default key and its course ID is equal to courseId | 2 | N | 2N |
| SET printNode equal to its successor | 1 | N-1 | N-1 |
| OUTPUT course ID and Name for printNode | 1 | 1 | 1 |
| IF printNode’s course’s prerequisite list is not null | 1 | 1 | 1 |
| FOR each item in the prerequisite list | 2 | N-1 | 2N-2 |
| OUTPUT current prerequisite’s course’s course ID | 1 | N-1 | N-1 |
| Total Cost | | | 7N+7 |
| Runtime | | | O(N) |

### Binary Search Tree

START function printCourseInformation (string courseId)

CREATE new node pointer, curNode

WHILE curNode is not null

IF curNode’s course’s course Id is equal to courseId

PRINT course ID and name

IF curNode’s course’s prerequisite list is not null

FOR curPrereq in prerequisite vector

OUTPUT curPrereq’s course’s course ID

END FOR

END IF

RETURN

ELSE IF curNode’s course’s course Id is greater than courseId

SET curNode equal to its left node

ELSE

SET curNode equal to its right node

END IF

END WHILE

PRINT No course found with ID: courseID

END function printCourseInformation

Table :Worst case runtime analysis for binary search tree printCourseInformation () function

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Code** | **Line Cost** | | **# Executions** | **Total Cost** |
| CREATE new node pointer, curNode | 1 | | 1 | 1 |
| WHILE curNode is not null | 1 | | N+1 | N+1 |
| IF curNode’s course’s course Id is equal to courseId | 1 | | N | N |
| ELSE IF curNode’s course’s course Id is greater than courseId | 1 | | N-1 | N-1 |
| SET curNode equal to its right node | 1 | | N-1 | N-1 |
| PRINT course ID and name | 1 | | 1 | 1 |
| IF curNode’s course’s prerequisite list is not null | 1 | | 1 | 1 |
| FOR curPrereq in prerequisite vector | 2 | | N-1 | 2N-2 |
| OUTPUT curPrereq’s course’s course ID | 1 | | N-1 | N-1 |
| Total Cost | | | 7N-1 | |
| Runtime | | O(N) | | |

## Appendix C: Worst Case Growth Rate Analysis for printSampleSchedule() function

### Vector Pseudocode

START printSampleSchedule(Vector courseVector)

CREATE Vector sortedCourseVector

QUICKSORT courseVector, output to sortedCourseVector

FOR each item in sortedCourseVector

OUTPUT course ID and name

IF course prerequisite vector size is not zero

FOR each item in prereqs vector

OUTPUT course ID

END FOR

END

END FOR

END printSampleSchedule(Vector courseVector)

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Executions** | **Total Cost** |
| CREATE Vector, sortedCourseVector | 1 | 1 | 1 |
| QUICKSORT courseVector, output to sortedCourseVector | N2 | 1 | N2 |
| FOR each item in sortedCourseVector | 2 | N | 2N |
| OUTPUT course ID and name | 1 | N | N |
| IF course prerequisite vector size is not zero | 1 | N | N |
| FOR each item in prereqs vector | 2 |  | N2-N |
| OUTPUT course ID | 1 |  |  |
| Total Cost | |  | |
| Runtime | | O(N2) | |

### Hash Table

Note: Items in red are considered to be part of the “worst case scenario” for the runtime analysis

START function printSampleSchedule (HashTable courseList)

CREATE a binary search tree object, courseBst

CREATE a node, curNode for iterating across HashTable keys

FOR all nodes in courseList

IF node in hash table does not have default (empty) key

CALL binary search tree Insert function to insert current node’s course into courseBst

SET curNode to the next node in the table

WHILE curNode is not null

CALL binary search tree Insert function to insert current node’s course into courseBst

SET curNode to its next node

END WHILE

END IF

END FOR

CALL BST function inOrder, passing root of courseBst

END function printSampleSchedule (HashTable courseList)

Table :Worst Case Growth Rate Analysis for Hash Table printSampleSchedule

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Executions** | **Total Cost** |
| CREATE a binary search tree object, courseBst | 1 | 1 | 1 |
| CREATE a node, curNode for iterating across HashTable keys | 1 | 1 | 1 |
| FOR all nodes in courseList | 2 | N | 2N |
| IF node in hash table does not have default (empty) key | 1 | N | N |
| CALL binary search tree Insert function to insert current node’s course into courseBst | N+6 | N |  |
| SET curNode to the next node in the table | 1 | N | N |
| WHILE curNode is not null | 2 |  |  |
| CALL binary search tree Insert function to insert current node’s course into courseBst | N+6 |  |  |
| SET curNode to its next node | 1 |  |  |
| CALL BST function inOrder, passing root of courseBst |  | 1 |  |
| Total Cost | |  | |
| Runtime | | O(N2) | |

### Binary Search Tree

START function inOrder(Node\* node)

IF (node is null)

RETURN

END IF

RECURSE inOrder of pointer to left of node

OUTPUT node course ID

IF course prerequisite vector size is not zero

FOR each item in prereqs vector

OUTPUT course ID

END FOR

END

RECURSE inOrder of pointer to right of node

END inOrder

START printSampleSchedule (BST courseList)

CALL function inOrder, passing root of courseList as argument

END printSampleSchedule (courseList)

Table : Worst Case Growth Rate Analysis for inOrder Function

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Executions** | **Total Cost** |
| IF (node is null) | 1 | 1 | 1 |
| RECURSE inOrder of pointer to left of node | 1 | N/2 | N/2 |
| OUTPUT node course ID | 1 | N | N |
| IF course prerequisite vector size is not zero | 1 | N | N |
| FOR each item in prereqs vector | 2 |  | N2-N |
| OUTPUT course ID | 1 |  |  |
| RECURSE inOrder of pointer to right of node | 1 | N/2 | N/2 |
| Total Cost | |  | |
| Runtime | | O(N2) | |

Table : Worst case growth rate for printSampleSchedule for Binary Search Tree

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Executions** | **Total Cost** |
| CALL function inOrder, passing root of courseList as argument |  | 1 |  |
| Total Cost | |  | |
| Runtime | | O(N2) | |