**ABCU Advising Program Pseudocode**

*Reading the file:*

IFSTREAM readFile(“file name)

OPEN readFile

IF (file does not open)

PRINT error

ELSE

WHILE (not end of file)

IF (less than two parameters)

RETURN error

ELSE IF (more than 3 parameters)

IF (each parameter after second is not in string)

RETURN error

CLOSE readFile

*Creating course objects and storing them:*

CREATE Course class

DEFINE structure to hold course info

DECLARE courseNum

DECLARE courseName

DECLARE coursePre

CREATE courses vector

void createCourses(Vector<Course> courses, String file) {

FOR LOOP (Iterate through file info)

CREATE course object

SET courseNum equal to course number from file

SET courseName equal to course name from file

SET coursePre equal to course prerequisite from file

STORE course object in vector

}

*Printing course info:*

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

FOR LOOP (iterate through course vector)

IF (course equals courseNumber)

PRINT course info

FOR LOOP (iterate through course prerequisites

PRINT prerequisite course info

}

*Hash Table – load, store, and print course info:*

DECLARE HashTable class

DEFINE structure to hold course info

private:

INITIALIZE course

DEFINE courseNumber

DEFINE courseName

DEFINE preRequisites

DEFINE CONSTRUCTORS

INSTANTIATE node vector

DEFINE default tableSize

DEFINE hash key

public:

DECLARE methods

*Stores course info:*

void insertCourses(Vector<Course> courses) {

FOR-LOOP (iterate through course vector)

CREATE key for current course

DEFINE node pointer using key

IF (node equals null)

INITIALIZE new node pointer with key

ASSIGN node to key position

ELSE IF (node key equals default value)

ASSIGN key to key position

ASSIGN course to current course

ASSIGN node next to null

ELSE

WHILE-LOOP (node next is null)

SET node equal to next

SET node equal to new Node with course and key

}

*Searches a course:*

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

INITIALIZE course

CREATE key for course

CREATE node pointer at key position

IF (node does not equal null and key is not default value and courseNumber matches)

RETURN course node

ELSE IF (node equals null or key equals default value)

RETURN course

WHILE-LOOP (node does not equal null)

IF (node key does not equal null and node course matches)

RETURN course

SET node equal to next

RETURN course

}

*Prints course info:*

Void printCourses() {

FOR-LOOP (iterate through hash table)

CREATE node pointer at current iteration

IF (node key is not default value)

OUTPUT course info

SET node equal to next

WHILE (node next does not equal null)

OUTPUT course info

SET node equal to next

}

*Binary Search Tree – load, search, and print info*

DECLARE Binary SearchTree class

DEFINE structure to hold course info

private:

INITIALIZE course root

DEFINE addCourse

DEFINE removeCourse

DEFINE CONSTRUCTORS

DEFINE DECONSTRUCTOR

public:

DECLARE methods

*Stores course info:*

void insertCourse(Course course) {

IF (root equals null)

SET root equal to new Course (call constructor)

ELSE

CALL addCourse(root, course)

}

*Searches a course:*

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

SET current node pointer equal to root

WHILE-LOOP (current node does not equal null)

IF (current node course number equals course number input)

RETURN current node course

IF (course number is smaller than current node course number)

SET current node pointer to left node pointer

ELSE

SET current node pointer to current node right pointer

}

*Displays course in order traversal:*

void inOrder(Node pointer node) {

IF (node does not equal null)

RECURSIVE CALL inOrder(node left pointer)

DISPLAY Course info

RECURSIVE CALL inOrder(node right pointer)

}

*Menu Pseudocode:*

INITIALIZE choice integer variable

WHILE-LOOP (choice does not equal 9)

DISPLAY formatted menu

SWITCH (choice input)

CASE 1:

CALL loadBid method – loads file into the data structure

BREAK

CASE 2:

CALL orderList method – orders list of courses alphanumerically

BREAK

CASE 3:

CALL printCourses method – prints course title and prerequisites

BREAK

CASE 9:

SET choice equal to 9 – ends while loop and exits program

BREAK

*Sorts and prints list of courses in alphanumeric order:*

void printSort(Vector<Course> courses) {

CALL sort(courses begin, courses end, [] ( const course struct num, const course struct name (references) )

RETURN num greater than name

FOR – LOOP (iterate through course vector)

DISPLAY sorted course info

}

**Runtime Evaluation**

*Worst Case Running Times*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| Object Creation | O (N) – With N elements | O (1) - O(N) Collisions may impact runtime | O (N) – O (log N) Based on a tree’s levels |
| Search Function | O (N) – With N elements | O (N) or O (log N) - For binary search | O (log N) - O (N) Based on a tree’s height |
| Print Function | O (N) – With N elements | O (1) - O (N) Based on collision mitigation method | O (N) – For all traversal orders |

**Data Structures Analysis**

*Vector*

Advantages: Using a vector makes browsing for an element convenient by using an index associated with the position of an element, supporting random access. It may be simple to insert or remove elements if they are being modified at the end of the vector. Vectors also allow for a variety of types of elements to be stored whether it be objects or items of different data types. A vector is also dynamic, very much able to change in size based on the number of items allocated to the structure.

Disadvantages: Performing insertions or removals anywhere in between a vector is inefficient as every single element to be set before or after the current element must be shifted and their positions must be given a new index. Since a vector is also considered an object, it requires more memory allocation to store the vector within a system. When searching in larger collections of data stored in a vector, time complexity is O (n) with n being the number of elements in the vector required to iterate through to retrieve the intended element. This can lead to degraded performance in an instance where the last element in a vector meets the criteria, each element within the vector will be accessed until the element in question is found.

*Hash Table*

Advantages: Hashing enables fast data retrieval using key mapping to a unique value serving as the index for an element. It can be as fast as O (1) in constant time without collisions. It is also a secure method of storing information as it can be used to verify data integrity or hashing passwords for accounts. Hash functions may also be applied to cryptographic applications for authentication algorithms as well as data encryption or decryption algorithms with the use of hash values that are difficult to retrieve derived by the hash key.

Disadvantages: In certain instances, collision may occur whenever two inputs can produce the same hash value, causing inefficiency in data retrieval, insertion, modification, or removal. Hash tables also have limited capacity, which can lead to further situations with several inputs producing the same values. They also do not allow keys or values to contain null.

*Binary Search Tree*

Advantages: A BST (Binary Search Tree) maintains the order in which data is stored. Traversing through the tree is made much easier as elements are stored in sorted order. Making it much easier to find the next or previous element, allowing a quick time complexity of O (log n) when searching a BST. The size of a BST is also dynamic, automatically increasing or decreasing as required based on the amounts of data stored or removed without jeopardizing a system’s memory.

Disadvantages: For the cost of operations within a binary search tree to be efficient. A height-balanced BST much be implemented every time. It is the best way to maintain a logarithmic rather than linear array search. The time complexity being O (log n) is ideal for bigger data sets but may be slower than vectors, arrays, or hash tables where searches, insertions, and delete operations are quicker in those data structures. Searching for keys such as strings or objects that aren’t comparable may require another function or the use of another data structure depending on the application.

**Recommendation**

After analyzing the three data structures that may be used to store all the courses that will be included in the program. I recommend implementing a vector to store the list of courses listing the course number, name, and prerequisites using a structure. This is because a vector will allow for quick random access for a particular course in cases where an advisor must search for a course and its details when informing students during course registration. The data structure can be dynamically modified to insert or remove courses as required without reaching a limit on the number of elements being added. Since the application in this case does not require large amounts of data, its use is convenient without the concern of hindering the program’s performance by implementing a vector.