Project One

|  |  |  |
| --- | --- | --- |
| **VECTORS** | **HASH TABLES** | **BINARY SEARCH TREE** |
| **Pros:**   * Scaleable * Small file size * Multipurpose * Reusable * Interactive * Animated * Edited with code | **Pros:**   * Lower level structure * Fixed for memory allocation * Memory efficient * Static or Dynamic | **Pros:**   * Efficient for insertion and deletion * Utlilize element insertion |
| **Cons:**   * Easily overused can cause memory issues * Slower processing time * Not structured * Harder to track | **Cons:**   * Fixed and not reallocatable * Less code efficient * Not effective if entries are small | **Cons:**   * Complicated than linear search * **Only works on sorted lists** * Only works on element types * Loss of efficiency * Faster search methods available such as hash look ups. |

In my evaluation I recommend using hash tables for ABCU course list. My reasoning is that the courses themselves follow a linear fashion of pre-requisites > next course. This method of using a hash table (or a linked list) will greatly reduce processing time on query and able the client to figure out which class is next. This will also greatly reduce the amount of memory needed to process the information when coding due to its simple linear structure.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | 3 | 4 |
| **if the course is the same as courseNumber** | 1 | 1 | 2 |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 8 | 8 | 16 |
| **print the prerequisite course information** |  | 16 | 16 |
| **Total Cost** | | | 39n + 1 |
| **Runtime** | | | 40(n) |

**Import fstream**

**MENU**

Start

Display menu to screen

Data Input

While course has preReq

Increment courseNum +1

IF course has preReq

Display preReq

Break

Display course list

User input

If user = eligible for course

Print course and information

Else if user is not eligible for course

Print “Please complete perquisites”

Else

Print “You are eligible for this course”

User to Press any key to continue

If user action = x

Exit program

Return

**Vector.ps**

Assign VARIABLES <COURSES>

Create DEFAULT CONSTRUCTOR FOR NODE

Initialize FILE

Initialize FILE AND KEY

Create VECTOR TO HOLD NODES

Create TABLE TO ALLOW SIZE CHANGING

Initialize STRUCTURE TO HOLD FILES

Call RESIZE

Apply LOGIC TO FREE STORAGE ONCE CLASS IS DESTROYED:

Call ERASE WITH THE BEGIN METHOD

Apply LOGIC TO CALCULATE A HASH VALUE

Return KEY OF TABLE

Apply LOGIC TO INSERT A COURSE

Calculate KEY FOR COURSEAS STRING FROM OBJECT

Retrieve NODE USING A KEY

If NO ENTRY IS FOUND

Create NODE

Endif

Else

If NODE FOUND

Assign KEY, FILE, AND NEXT POINTER TO NULL

Endif

Else

While NODE IS NOT NULL

Go THROUGH LIST UNTIL NULL

Endwhile

Assign NODE WITH FILE AND KEY

Endelse

Endelse

Apply LOGIC TO PRINT ALL FILES

For INTEGER SMALLER THAN SIZE OF STRUCTURE, INCREASE SIZE

Call displayfile

Endfor

Calculate KEY FOR fileid AS STRING FROM OBJECT

Erase MATCH FROM TABLE

Calculate KEY FOR FILE AS STRING FROM OBJECT

Call NODE USING RETRIEVAL KEY

If NO ENTRY FOUND

Return NULL

Endif

If NODE MATCHES KEY

Hash table psuedocode:

Node POINTER STRUCTURE {

Create VARIABLE courseid OF TYPE course

Create nextnodepointer

}

Create node pointer VARIABLE HEAD THAT WILL POINT TO THE HEAD OF THE LIST

Create node pointer VARIABLE TAIL THAT WILL POINT TO THE TAIL OF THE LIST

Create listsize VARIABLE OF TYPE int

Set HEAD to null

Set TAIL to null

Linkedlist::append(COURSE appcourse) {

Create new POINTER TEMP OF TYPE node

Set COURSEIN TEMP NODE to appcourse

Set nextnodepointer IN TEMP NODE to null

If HEAD equals null then {

Set HEAD to TEMP

Set TAIL to TEMP

}

Else{

Set nextnodepointer IN TAIL to TEMP

Set TAIL to TEMP

}

Increment listsize BY ONE

}

Linkedlist::prepend(courseprecourse) {

Create new pointer TEMP OF TYPE node

Set COURSEIN TEMP NODE to precourse

Set nextnodepointer IN TEMP NODE to null

Set nextnodepointer IN TEMP to HEAD

Set HEAD TO TEMP;

Increment listsize BY ONE

}

Linkedlist::printlist() {

Create new pointer TEMP OF TYPE node

Set TEMP to HEAD OF THE LIST

While (TEMP is not null) do {

Print courseid, coursetitle, courseamount, coursefund FROM TEMP

Set TEMP to TEMP’S nextnodepointer

Linkedlist::remove(STRING courseid){

Create new pointer CURRENT OF TYPE node

Create new pointer PREVIOUS OF TYPE node

Set CURRENT to HEAD

While (CURRENT is not null) do {

If (courseid IN CURRENT equals courseid) then {

Set PREVIOUS’ nextnodepointer to CURRENT’S nextnodepointer

Break

}

Set PREVIOUS to CURRENT

Set CURRENT to CURRENT’S nextnodepointer

}

Deincrument listsize

}

Linkedlist::search(STRING courseid) {

Create new pointer TEMP OF TYPE node

Set TEMP to HEAD;

Create VARIABLE COURSEOF TYPE course

While ( TEMP is not null) do {

If(courseid IN TEMP equals courseid) then {

Set courseto COURSEIN TEMP

Break

}

Else {

Set TEMP to nextnodepointer IN TEMP

}

Return COURSE;

}

Insert(root,course)

If root = null

Set node -> newcourse = new node(course)

Newcourse->left = null;

Newcourse -> right = null;

Root = newcourse

Else

Insert node(root,course)

Remove node (courseid)

If root == null

Return false

Else

Node \* start =root

While (start != null)

If (start -> data->courseid == courseid)

Return true

If (start -> data-> courseid)

Start = start ->right

Else

Start = start -> left

Return false

Visit node

Visit left branch

Visit right branch

Return

Goto left branch

Visit node

Goto right branch

Return

Goto left bran insert(root,course)

If root = null

Set node -> newcourse = new node(course)

Newcourse->left = null;

Newcourse -> right = null;

Root = newcourse

Else

Insert node(root,course)

Remove node (courseid)

If root == null

Return false

Else

Node \* start =root

While (start != null)

If (start -> data->courseid == courseid)

Return true

If (start -> data-> courseid)

Start = start ->right

Else

Start = start -> left

Return false

Visit node

Visit left branch

Visit right branch

Return

Goto left branch

Visit node

Goto right branch

Return

Goto left branch

Goto right branch

Visit node

Return

Goto right branch

Visit node

Return