**Vector**

**Load** parsing libraries and appropriate headers

**Define** the structure to hold course data

**struct course()**

courseId

courseName

prerequisiteNumber

prerequisiteCourse

Course() (constructor) courseId = courseName = “”; prerequisiteNumber = 0; prerequisiteCourse=””

**Main()**

**Create** a list *courseData* of the struct-type **Course**

**Get** user to upload CSV file

**If** no data given use default location

**Call parser()** to pass the provided CSV file

**Call validate()** to pass the provided *courseData*

**Get** user input to search and **store** in *searchData*

**Call printData()** to pass provided *searchData*

**END**

**parser()**

**Create** localized list name *temp*

**Open** file in the path in string by utilizing parsing libraries

**While(loop)** each row until the end of the correct file

**If** first and second string are matched

**Add** the first to courseId

**Add** the second to courseName

**While(loop)** continues until file has no value in column (or when there are no more prerequisites)

**Increase** count for prerequisiteNumber for every one found

**Create** localized string for prerequisiteCourse for every one found

**Add** prerequisiteNumber to structure at prerequisteNumber

**Add** prerequisiteCourse to structure at prerequisiteCourse

**Return** temp

**END**

**search()**

**Create** temporaryCourse to **course**

**While(loop)** through list for each course searched

**If** user search is the same as courseId

**Set** temporaryCourse to **course**

**Return** temporaryCourse

**END**

**printData()**

**Create** temporaryCourse to **course()**

**Set** temporaryCourse equal to **search()**

**Output** courseId

**Output** courseName

**While(loop)** 0 to prerequisiteNumber

**For** each course in prerequisiteName

**Call printData()** to pass prerequisiteNumber

**END**

**validate()**

**Create** temporaryCourse to **course()**

**Create** variable *courseNeeded* and **Set** to True

**For** each course

**If** *courseNeeded* is False

**break**

**While(loop)** 0 to prerequisiteNumber

**Set** temporaryCourse equal to **search()**

**If** temporaryCourse courseId empty **Set** *courseNeeded* to False

**Return** *courseNeeded*

**END**

**Hash Table**

**Load** parsing libraries and appropriate headers

**Define** the structure to hold course data

**struct course()**

courseId

courseName

prerequisiteNumber

prerequisiteCourse

Course() (constructor) courseId = courseName = “”; prerequisiteNumber = 0; prerequisiteCourse=””

**Class hashTable{}**

struct bucket

**Course**

**Key**

**Next (pointer)**

+ hash()

+ printAll()

+List (hashTable)

**Main()**

**Create** a list *courseData* of the struct-type **Course**

**Get** user to upload CSV file

**If** no data given use default location

**Call parser()** to pass the provided CSV file

**Call validate()** to pass the provided *courseData*

**Get** user input to search and **store** in *searchData*

**Call printData()** to pass provided *searchData*

**END**

**parser()**

**Open** file in the path in string by utilizing parsing libraries

**While(loop)** each row until the end of the correct file

**If** first and second string are matched

**Call** hash by passing first string

**Add** to structure at hash placement within temp

**Add** the first to courseId

**Add** the second to courseName

**While(loop)** continues until file has no value in column (or when there are no more prerequisites)

**Increase** count for prerequisiteNumber for every one found

**Create** localized string for prerequisiteCourse for every one found

**Add** prerequisiteNumber to structure at prerequisteNumber

**Add** prerequisiteCourse to structure at prerequisiteCourse

**Return** temp

**END**

**search()**

**Create** temporaryCourse to **bucket**

**Set** temporaryCourse to bucket at hash lo

**While(loop)** through list for each course searched

**If** user search is the same as courseId

**Set** temporaryCourse to **course**

**Return** temporaryCourse

**END**

**printData()**

**Create** temporaryCourse to **bucket**

**Set** temporaryCourse equal to hash of the string

**While(loop)** through all buckets at temporaryCourse

**Output** courseId in Course struct within temporaryCourse

**Output** courseName in Course struct within temporaryCourse

**While(loop)** 0 to prerequisiteNumber

**For** each course in prerequisiteName

**Call printData()** to pass prerequisiteNumber

**END**

**validate()**

**Create** temporaryCourse to **bucket**

**Create** variable *courseNeeded* and **Set** to True

**For** each course

**If** *courseNeeded* is False

**break**

**While(loop)** 0 to prerequisiteNumber

**Set** temporaryCourse equal to **search()**

**If** temporaryCourse courseId empty **Set** *courseNeeded* to False

**Return** *courseNeeded*

**END**

**int Hash(key)**

**Return** key

**Binary Tree**

**Load** parsing libraries and appropriate headers

**Define** the structure to hold course data

**struct course()**

courseId

courseName

prerequisiteNumber

prerequisiteCourse

Course() (constructor) courseId = courseName = “”; prerequisiteNumber = 0; prerequisiteCourse=””

**Class BinaryTree{}**

struct Node

**Course**

**right pointer**

**left pointer**

root

printTree()

BinaryTree()

**Main()**

**Create** new BinaryTree “courseTree” of the struct-type **Course**

**Get** user to upload CSV file

**If** no data given use default location

**Call parser()** to pass the provided CSV file

**Call validate()** to pass the provided *courseTree*

**Get** user input to search and **store** in *courseSearch*

**Call printData()** to pass provided *courseSearch*

**END**

**parser()**

**Open** file in the path in string by utilizing parsing libraries

**While(loop)** each row until the end of the correct file

**If** first and second string are matched

**Add** the first to courseId

**Add** the second to courseName

**While(loop)** continues until file has no value in column (or when there are no more prerequisites)

**Increase** count for prerequisiteNumber for every one found

**Create** localized string for prerequisiteCourse for every one found

**Add** prerequisiteNumber to structure at prerequisteNumber

**Add** prerequisiteCourse to structure at prerequisiteCourse

**Return** temp

**END**

**search()**

**Create** temporaryCourse to **node**

**While(loop)** through list for each course searched

**If** user search is the same as courseId

**Set** temporaryCourse to **course**

**Return** temporaryCourse

**END**

**printData()**

**Create** temporaryCourse to **node**

**Set** temporaryCourse equal to root

**While(loop)** through all buckets at temporaryCourse

**If** node at temporaryCourse contains a the bidId that equals the string

**Output** courseId in Course struct within temporaryCourse

**Output** courseName in Course struct within temporaryCourse

**While(loop)** 0 to prerequisiteNumber

**For** each course in prerequisiteName

**Call printData()** to pass prerequisiteNumber

**If** node at temporaryCourse contains courseId less than string

**Set** temporaryCourse equal to left node

**If** node at temporaryCourse contains courseId greater than string

**Set** temporaryCourse equal to right node

**END**

**validate()**

**Create** temporaryCourse to **node**

**Create** variable *courseNeeded* and **Set** to True

**For** each course

**If** *courseNeeded* is False

**Break**

**While(loop)** temporaryCourse is not null

**While(loop)** 0 to prerequisiteNumber

**Set** temporaryCourse equal to **search()**

**If** temporaryCourse courseId empty **Set** *courseNeeded* to False

**Return** *courseNeeded*

**END**

**Menu**

**Int Main()**

bool quit = false

print “1) Load files into data structure”

print “2) Print list of courses alphabetically”

print “3) Print course and prerequisites”

print “9) Exit the program”

case 1:

create data structure (courseName)

load the courses through the csv Path (courseName)

**Break**

case 2:

print schedule alphabetically (courseName)

**Break**

case 3:

print the course information (courseName)

**Break**

case 9:

quit = true

**Break**

print “Invalid Selection” for any other input

**Break**

**Run Time Table**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Vector** | **Hash Table** | **Binary Tree** |
| **Loading Data** | O(1) | O(1) – O(n)  \*dependent on collisions | O(log n) |
| **Searching** | O(n) | O(1) – O(n)  \*dependent on collisions | O(log n) – O(n)  \*dependent on tree balance |
| **Sorting and Printing** | O(n log n)  \*using quick sort | O(n)  \*with assumption table is in order | O(n)  \*in order |

**Analysis**

All three of these data structures have their advantages and disadvantages, which I will describe below:

**Vector**

A vector data structure has a good number of advantages. For instance, it highly precise, it has a large scalability, and it has a good amount of storage for discrete feature with definitive boundaries. But it has a number of disadvantages to go along with it, including a complex data manipulation and processing overhead for larger datasets. So in laymen’s terms, it has a very fast ability to search but when it comes to sorting the found files, it is the slowest of the three data structures.

**Hash Table**

A hash table data structure is a good option because it provides a fast lookup, it has efficient insertion and deletion, it uses space efficiently, it provides a nice amount of flexibility, and has built-in collision resolution mechanisms. However, there are some pretty big disadvantages to a hash table approach, for example it is inefficient when there are too many collisions, it does not allow null values, they have a limited capacity, and they can be complex to implement.

**Binary Search Tree**

A binary search tree data structure again has advantages and disadvantages like the other two. Some of the advantages is that it is extremely efficient when searching, it has an ordered structure, it can dynamically insert and delete and it has a balanced structure. The disadvantages with this data structure is that it is not self-balancing which can lead to poor performance, it is not suitable for larger datasets because it can become inefficient, and it has a limited functionality as it can only search, insert and delete operations.

**Recommendation**

If we were doing a larger number of courses, I would probably suggest using the Hash Table approach as it is the most efficient working with larger data sets. However, because we are working with a small set of data and we only need to search and order, I would say that the Binary Search Tree is the best option to write the program for ABCU’s Computer Science department.