# Vector Data Structure Pseudocode

Opening a File:

Open File (param 1, param 2 , param n)

{

Open file

File = open(filename)

IF file does not open

Return 1

ELSE

WHILE file is not empty

Line = readline (filename)

IF there are < 2 values per line

Return ERROR

ELSE

Read params

IF params does not exist in list

Return error

Else

continue

Close File

Creating course objects:

Initialize vector<Course\*>

Start a loop through file

While not at the End Of File

Readline

For param1 , param2

Pushback to vector

Check for param n

If exists

Use pushback to vector

Else

Continue

Searching and Printing Data Structure:

Get USER input

LOOP through the vector

IF USER INPUT = Course number

Print course number , course name

Check for prerequisite

If present

Print prerequisite

Else

Print not a valid course

# Binary Search Tree Pseudocode:

Opening a File:

Open File (param 1, param 2 , param n)

{

Open file

File = open(filename)

IF file does not open

Return 1

ELSE

WHILE file is not empty

Line = readline (filename)

IF there are < 2 values per line

Return ERROR

ELSE

Read params

IF params does not exist in list

Return error

Else

continue

Close File

Creating course objects:

courseNumber = node

Binary Search Tree (tree, node) {

If (tree->root is null)

Tree->root = node

Node->left = null

Node->right = null

Else

currentNode = tree->root

while(currentNode is not null)

if(node->key < currentNode->key)

if(currentNode->left is null)

currentNode->left = node

currentNode = null

else

currentNode = currentNode->left

else

if(currentNode->right is null)

currentNode->right = node

currentNode = null

else

currentNode = currentNode->right

node->left = null

node->right = null

End

Print Data Function:

BSTPrintInOrder(node) {

If(node is null)

Return

BSTPrintInOrder(node->left)

Print node

BSTPrintInOrder(node->right)

Print node

}

# Hash Table Pseudocode:

Opening a File:

Open File (param 1, param 2 , param n)

{

Open file

File = open(filename)

IF file does not open

Return 1

ELSE

WHILE file is not empty

Line = readline (filename)

IF there are < 2 values per line

Return ERROR

ELSE

Read params

IF params does not exist in list

Return error

Else

continue

Close File

Creating course objects:

Initialize vector<Course\*>

Create HashTable class

Insert method for hashtable

Key = current bid node

Searches list for node of that value

If the node doesn’t exist yet

Assign key

Else

Loops through till next open node

Searching and Printing Data Structure:

Get USER input

LOOP through the vector

IF USER INPUT = Course number

Print course number , course name

Check for prerequisite

If present

Print prerequisite

Else

Print not a valid course

# Menu Pseudocode:

SET user\_choice = 0

WHILE user\_choice DOES NOT = 4

OUTPUT

1.Load Course File

2.Print Course List

3.Print Course

4.Exit

IF CASE 1:

loadCourse(file, dataStructure)

IF CASE 2:

printSorted(course)

IF CASE 3:

printCourse(courseNumber)

IF CASE 4:

Exit the program

# Sorted List Pseudocode:

printSorted(course)

partition(vector<Course\*> &course, start, end) {

pivot = end

j = start

FOR(i = start and i < end; increment i)

IF(course[i] < course[pivot]){

SWAP(course[i],course[j])

Increment j

}

}

SWAP(course[j], course[pivot])

RETURN j

}

QuickSort(vector<Course\*> &course, start, end) {

IF(start<end){

P = partition(course,start,end)

quicksort(course, start, p-1)

quicksort(course,p+1, end)

}

}

PrintCourse(Vector<Course\*> course){

For(i= 0; I < course.size(); increment I;

OUTPUT course[i]

# Runtime Analysis:

Vector:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 2 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 2 | n | n |
| **Total Cost** | | | 6n + 1 |
| **Runtime** | | | 1(n) |

Hash Table:

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 2 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 2 | n | n |
| **print the prerequisite course information** | 4 | n | n |
| **Total Cost** | | | 9n + 1 |
| **Runtime** | | | O(n) |

Binary Search Tree:

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

# Advantages and Disadvantages:

Vector: Advantages of using a vector are fast searches as it can find a element in constant time. You are also able to increase the size of a vector quite easily and add space onto the end of the vector. One of the disadvantages of vectors is it is not as easy to add elements to any part other than the end of the vector, this causes you to have to shift each element which increases the cost of operation. Vectors are also known to be a bit excessive in their memory usage.

Hash Table: The advantages of using a hash table would be that it provides constant time for searching, inserting, and deleting operations on average. This provides fast lookup times for any element since it uses a hash function to map keys to array indices. One of the disadvantages of using a hash table would be for instance if all the data falls into the same bucket this is when it becomes less efficient, it also doesn’t easily support the ability to find things such as the largest number or smallest number quickly.

Binary Search Tree: The main advantages of using a binary search tree would be fast insertion and deletion when the tree is balanced, while also being efficient since it only stores the elements and doesn’t need things like pointers. If the tree is unbalanced however, this can lead to higher cost of operations, changing from a logarithmic search to a linear search which would be one of the main disadvantages of using a BST.

# Conclusion:

After reviewing each of the data structures, I believe that using a vector would be the best choice for this project. This is due to the fact that the runtime analysis shows that it will have a lower total cost than the other two data structures.