# CS 300 Project 1

Ty Simpson

**//Open, read and validate file**

openFile(string filePath)

file = filePath

Open file with fstream

For each row in file

if row length < 2

print “invalid file format”

else

for each row index[i] greater than 1

if row index[i] == row index[0]

return row

else

print “Error: prereq is not an existing course”

End

**//Course structure**

struct Courses

string courseId;

string courseName;

Vector<string> preReqs;

End

**// Vector pseudocode**

//Load course objects

loadCourses (string file)

initialize vector Vector<Courses> courses

for each row in file

Courses courses

if row length < 3

courses.courseId = file[0]

course.courseName = file[1]

push courses

else

courses.courseId = file[0]

courses.courseId = file[1]

for each row index > 1

courses.preReqs = file[row\_index]

push preReqs

push courses

End

//Search and Display

searchCourse(Vector<Courses>, string coursId)

user\_input = courseId

for each item in Vectors<Courses>

if user\_input == courseId

print course information

for each preReqs

print preReqs

End

**// Hashtable pseudocode**

//Hashtable Definitions 

Set DEFAULT\_SIZE

//Structure to hold courses

Struct Node

Courses course

int key

Node \* next

Node

Key = UINT\_MAX

next = null

//Values for Hash Table structure

vector<Node>nodes

int tableSize = DEFAULT\_SIZE

int hash (int key)

//Hash Function

Hash(int key)

Return key mod tableSize

End

//HashTable Insert(Courses course)

key = hash(courseId) //In order to Hash, courseId must be converted to int)   
retrieveNode = nodes at key

if retrieveNode is NULL

assign node to key

else

if retrieveNode->key == UINT\_MAX

assign node to key

assign node to course

assign next node to NULL

else

while retrieveNode->next is NOT NULL

retrieveNode = retrieveNode->next

add node to end

End

//Load Course

IoadCourse(string file, hashTable)

For each row in file

Courses course

if row length < 3

course.courseId = file[0]

course.courseName = file[1]

hashTable->Insert(course)

else

course.courseId = file[0]

course.courseId = file[1]

for each row index > 1

course.preReqs = file[row\_index]

push preReqs

hashTable->Insert(course)

End 

//Search and Display

Search(string courseId)

key = hash(courseId)       //In order to Hash, courseId must be converted to int)   
node = nodes at key

if node is NOT NULL AND node->key != UINT\_MAX AND node->course.courseId == courseId

print course information

for each preReq

print preReqs

if node is NULL OR node->key == UINT\_MAX

return “Course not found”

// Tree pseudocode

//Structure to hold courses 

Struct Node

Courses course

Node \*left

Node \*right

Node

left = null

right = null

//BST Constructor

root = null

//Insert for BST

Insert(Courses course)

If root is null

root = new Node(bid)

Else if node->course.courseId > course.courseId

Add node to left

Else

Add node to right

End

//Load Course 

IoadCourse(string file, BinarySearchTree)

For each row in file

Courses course

if row length < 3

course.courseId = file[0]

course.courseName = file[1]

BinarySearchTree->Insert(course)

else

course.courseId = file[0]

course.courseId = file[1]

for each row index > 1

course.preReqs = file[row\_index]

push preReqs

BinarySearchTree->Insert(course)

End

//Search and Display

Search(string courseId)

Node \*current = root

While current is NOT NULL

If current == courseId

Print course

For each prereq

Print prereq

If course < current

Current = current->left

If Course > current

Current = current->right

Print “Course not found”

**//Sort and Print**

//Vector

PrintSorted(Courses)

//Partition vector into low and high

int partition(Course, begin, end)

//Low is beginning of vector, high is end, pivot is middle element

int low = begin

int high = end

int pivot = middle

//Create while loop to iterate

bool done = false

while done is NOT true

while low < pivot

increment low

while pivot < high

decrement hight

if elements remaining <= 1

done = true

else

swap(low,high)

increment low

decrement high

END

//QuickSort Method

quickSort(Courses, begin, end)

int middle = 0

//If there are 1 or 0 courses, vector is already sorted

if numCourses <=1

return

middle = partition(courses, begin, end)

//Sort low partition recursively

quickSort(courses,begin,middle)

//Sort high partition recursively

quickSort(courses,middle+1,end)

displayCourse(Courses)

for each course in Courses

print course info

for each preReq

print preReq

END

//BST

InOrder(Node\* node)

If Node is NOT Null

Point Node->Left

Print Course Info

Point Node->Right

END

**//Menu**

Set choice to 0

While choice DOES NOT equal 9

Output “Choose an option: 1) Load Courses 2)Sort and Display Courses 3)Search and Display a Course 9)Exit.”

Choice = input

switch(choice)

case 1: loadCourse(file, structure)

case 2: sortAndDisplay(Courses)

case 3: searchAndDisplay(courseId)

case 9: Output “Thank you and goodbye.”

END

## Runtime Analysis

| **Vector loadCourses** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize Vector** | 1 | 1 | 1 |
| **For each row in file** | 1 | n | n |
| **Create Course** | 1 | n | n |
| **If row length < 3** | 1 | n | n |
| **Push Course** | 1 | n | n |
| **Else** | 1 | n | n |
| **For each rowIndex > 1** | 1 | n | n |
| **Create preReq** | 1 | n | n |
| **Push preReq** | 1 | n | n |
| **Push Course** | 1 | n | n |
| **Total Cost** | | | 9n + 1 |
| **Runtime** | | | O(n) |

| **HashTable loadCourses** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize Hashtable** | 1 | 1 | 1 |
| **Insert()** | 0 | 0 | 0 |
| **Create Key** | 1 | n | n |
| **IF no node at key** | 1 | n | n |
| **Assign node to key** | 1 | n | n |
| **Else** | 1 | n | n |
| **If key == UINT\_MAX** | 1 | n | n |
| **Assign node to key** | 1 | n | n |
| **Assign node to course** | 1 | n | n |
| **Assign next node to NULL** | 1 | n | n |
| **Else** | 1 | n | n |
| **While node->next NOT NULL** | 1 | n | n |
| **Assign node to next** | 1 | n | n |
| **Add node to end** | 1 | n | n |
| **For each row in file** | 1 | n | n |
| **Create Course** | 1 | n | n |
| **If row length < 3** | 1 | n | n |
| **Insert(Course) to Hashtable** | 1 | n | n |
| **Else** | 1 | n | n |
| **For each row index > 1** | 1 | n | n |
| **Create preReq** | 1 | n | n |
| **Push preReq** | 1 | n | n |
| **Insert(Course)to Hashtable** | 1 | n | n |
| **Total Cost** | | | 21n + 1 |
| **Runtime** | | | O(n) |

| **BST loadCourses** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Initialize BST** | 1 | 1 | 1 |
| **Insert()** | 0 | 0 | 0 |
| **If root is Null add root** | 1 | 1 | 1 |
| **Else if node courseId > course courseId** | 1 | n | n |
| **Add node to left** | 1 | n | n |
| **Else add node to right** | 1 | n | n |
| **For each row in file** | 1 | n | n |
| **Create Course** | 1 | n | n |
| **If row length < 3** | 1 | n | n |
| **Insert(Course)to Tree** | 1 | n | n |
| **Else** | 1 | n | n |
| **For each row index > 1** | 1 | n | n |
| **Create preReq** | 1 | n | n |
| **Push preReq** | 1 | n | n |
| **Insert(Course)to Tree** | 1 | n | n |
| **Total Cost** | | | 12n + 2 |
| **Runtime** | | | O(n) |

**Data Structure Pros and Cons**

**Vector**

Based on the runtime analysis above, vectors have a fast load time, with 9n+1 at the worst-case. Sorting for vectors can be relatively fast, with the quick sort algorithm. However, searching through a vector is considerably longer as it must go through each element until a match is found.   
  
**Hash Table**

Based on the runtime analysis above, hash tables have a relatively slow load time, with 21n+1 at the worst case. Hash Tables cannot be sorted alphanumerically without additional steps because they are a key-pair structure. The key-pair structure, however, makes searching through a hash table incredibly fast.   
  
**Binary Search Tree (BST)**

Based on the runtime analysis above, BSTs are relatively fast in loading – though not as fast as vectors with a 12n+2 worst-case. The time it takes to search through a BST depends on the height of the tree and searching is quick.

**Recommendation:** Binary Search Tree

Hash Tables cannot be easily sorted alphanumerically. Since sorting is a key feature request, I will not be using the Hash Table structure. While vectors can read and load data relatively quickly – BSTs are much more efficient and organized when it comes to sorting and searching for courses. I would rather have a slower load time in exchange for faster sorting and searching.