Kristle Morales Figueroa

CS-300 Project One

1. **Pseudo code for opening, reading, parsing through a file:**

Make the call to open the file

**If** the file did not open

**Print** “File Not Open”

**End** the program

**Else**

**While** not at the end of the file

Read each line and copy into a string

Separate the string into a list of items separated by spaces

I**f** less than 2 spaces are present in the string

**Skip** to next line

**Else if** more than 2 spaces are present

**Add** string to object *Courses*

**Close** file

1. **Pseudocode for a menu**

**Loop** while choice is not ***9***

**Print** *Menu options*

**Ask** for user input

**Validate** user input

**If**  user input is not 1-3 or 9

**Print “*Error”***

**If** user input equals ‘1’

**Call** file parser routine and load file data onto the data structure

**Print** the number of records in the file

**If** user input equals ‘2’

**Print** an order list of all courses in the Computer Science department

**If** user input equals ‘3’

**Ask** user input for *courseNumber*

**Call** *search* routine

**Print** the course title and the prerequisites for any individual course.

**If** user input equals ‘9’

**End program**

1. **Pseudo code for creating course objects and store them in a data structure**

***Object***

**Define struct** Courses with items data type string courseNumber, name, prerequisite1, prerequisite 2

***Data Structure: Vector***

Define a **vector***<Courses> courses*

**Call** the parsing routing

**Loop** through the end of the file

**Add** items to the end of the structure

**Return** *courses*

***Data structure: Hash Table***

**Define** a **Hash Table** class

**Define** a **struct** *Node* with items **Courses** *courses,* **unsigned** *key*, **Node** *\*next*

Define a constructor **Node()** with items *key, next*

**Define** a vector <**Node**> *nodes*

**Initilize** the tableSize

**Initilize** hash(*key*)

**Define** public methods for the class

HashTable()

HashTable(unsigned size)

**void** *Insert*

**void** *Search*

**void** *PrintAll*

**void** *Remove*

***Data structure: Binary Search Tree***

**Create** a Binary Search Tree structure

**Create** a new node

**Define** the **Courses** element

**Define** a left and right node

**Initialize** the left and right node to nullptr using the constructor

**Declare** private functions

**Create** a new node called *root*

**Create** a function for adding an object

**Create** a function for removing an object

**Create** a function for displaying all objects

**Declare** public functions

**Call** methods for searching

**Call** methods for inserting

**Call** methods for removal

**Define** *root* as nullptr in the default constructor

BinarySearchTree::Insert(Courses course)

**If** the root is nullptr

**Assign** root to a new course node

**Else** add a node with the root and course

1. **Pseudo code for search and print:** (*This pseudo code was edited from the provided Supporting Materials Document.*)

***Data Structure: Vector***

**Define** searchCourse(**Vector**<Course> *courses*, string *courseNumber*)

**Loop** though *courses*

**If** the *course* and the *courseNumber* match

**Print** *course* information

**Else for** each prerequisite of *course*

**Print** the prerequisite course information

**Return** *courses*

***Data structure: Hash Table***

**Define** searchCourse(**HashTable**<Course> *courses*, string *courseNumber*)

**Call** the hash table

**Loop** through the table looking for the key

**If** the node is not empty

**Print** the information on in the node

**Else**

**Return** *null*

Define **PrintAll**

***Data structure: Binary Search Tree***

**Create** a Binary Search Tree structure

**Create** a new node

**Define** the **Courses** element

**Define** a left and right node

**Initialize** the left and right node to nullptr using the constructor

**Declare** private functions

**Create** a new node called *root*

**Create** a function for adding an object

**Create** a function for removing an object

**Create** a function for displaying all objects

**Declare** public functions

**Call** methods for searching

**Call** methods for inserting

**Call** methods for removal

**Define** *root* as nullptr in the default constructor

BinarySearchTree::Insert(Courses course)

**If** the root is nullptr

**Assign** root to a new course node

**Else** add a node with the root and course

BinarySearchTree::Remove(Courses course)

**If** the root is nullptr

**Return** node

**If** the root is not empty go down the left side of the tree to find a match

**If** a match is found remove it

**If** the root is not empty go down the right side of the tree to find a match

**If** a match is found remove it

**Else** the tree is a leaf node

**If** the left node is empty and the right node is empty

**Delete** the node

**If** the left node is not empty and the right node is empty

**Create** a temp node

**Assign** temp to the left node

**Delete** the node

**If** the left node is empty and the right node is not empty

**Create** a temp node

**Assign** temp to the right node

**Delete** the node

**Else** more than one child is present

**Create** a temp node assigned to the right node

**Loop** through the left side of the tree to the not empty node

**Assign** the node to temp

**Remove** the node

**Return**

1. **Evaluation of data structures**

Based on the table provided in the Pseudocode document and the rubric instructions. We can determine the Big O value of our different data structures. First the vector, for the loading routine we would have value O(1), the search value would be O(n) and the sort and Print functions would result in O(N log N) for a quick sort. This means that the vector is the simplest structure to use to load data but sorting the data would be extremely slow. Next, we consider the Hash Table structure considering the possibility of collisions the loading data and searching routines can have a value O(1) – O(N) and the sort and print routines will have O(N) assuming the table was created in a sorted order. Finally, we consider the binary tree structure which will have a loading data Big O value of O(log N), a search value of O(log N)-O(N) and the sort and print will have O(N).

Based on the Big O evaluation the data structures to use would be either a Hash Table or a Binary Tree. Binary trees can be traversed in order and don’t need to be sorted this would make it faster to create the data structure but slower to search. The Hash Table would be faster to search, could be loaded in a pre-sorted way but would open the door to collisions.

My recommendation would be to use the Hash Table, assuming the data structure would have to be searched more often than any other function, the hash table has the advantage over the binary tree.