# CS 300 Pseudocode Document

## Example Function Signatures

Below is an example of a function signature that you can use as a guide to help address the program requirements using each data structure for the milestones. The pseudocode for finding and printing course information is also given below and depicted in bold to help you get started. The provided pseudocode is for a vector data structure, so you may use this pseudocode in your first milestone as is. The hash table and tree structures are also shown below. But these structures are left for you to do in future milestones.

**// Milestone One: Vector**

Void loadFile()

Find file path and open file

If file found open file

Read file and parse text from file

Else print “file not found”

Void readFile()

If file opened

Read file format and data

If file format corrects and data not corrupted

Print data read

Else print “file format corrupted”

Void textParser()

If file opened and format correct

Loop through entire file until end of file is reached

If two strings are present

Add first string to courseNumber

Add second string to courseName

If end of file reached

Close file

Struct Course () //this is to store information read from file

String courseNumber

String courseName

void searchCourse(Vector<Course> courses, String courseNumber) {

for all courses

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

**// Milestone Two: Hash Table**

Void newHashtable()

Create new hash table to input file info in.

Set size of hashtable large enough for info coming in.

Void loadHashtablefile()

Find hashtable file path and open file

If hashtable file found open file

Read hashtable file and parse text from file

Else print “hashtable file not found”

Void readhashtableFile()

If hashtablefile opened

Read hashtablefile format and data

If hashtablefile format correct and data not corrupted

Print data read

Else print “hashtablefile format corrupted”

Void hashtableParser()

If hashtablefile opened and format correct &&

hashtablefile large enough to accept amount of data being read

Loop through entire hashtablefile until end of file is reached

If two strings are present

Insert first string to courseNumber

Insert second string to courseName

If end of file reached

Close hashtablefile

Else resize newhashtable to accommodate incoming data

Loop through entire hashtablefile until end of file is reached

If two strings are present

Insert first string to courseNumber

Insert second string to courseName

If end of hashtablefile reached

Close hashtablefile

void searchCourse(HashTable<Course> courses, String courseNumber) {

}

**// Milestone Three: Binary Search Tree**

Void Buildtree()

Create root

Create left pointer

Create right pointer

Void loadtreefile()

Find tree file path and open file

If tree file found open file

Read tree file and parse text from file

Else print “tree file not found”

Void readtreeFile()

If treefile opened

Read treefile format and data

If treefile format correct and data not corrupted

Print data read

Else print “treefile format corrupted”

Void treefileParser()

If treefile opened and format correct

Set root

If next data is smaller than current root

Make this data new root

Adjust old root to right child if greater than new root or left child if less than new root

If two strings are present

Insert first string to courseNumber

Insert second string to courseName

If end of file reached

Close treefile

void searchCourse(Tree<Course> courses, String courseNumber) {

//Pseudocode for Menu

Start program

While input not equal to 5:

Menu Display shows “Select menu option”

“1. Load Data Structure”

“2. Print Course List”

“3. Print Course”

“9. Exit”

If selection 1 made Load data structure and return to display menu

If file found load file

Else “file not found”

If selection 2 Print course List and return to menu

While not at the end of list print data

Else end of list reached no data left to print

If selection 3 print course and prerequisites then return to menu

If course found print course with prerequisites

Else “course not found”

If selection 9 Exit Program

//Pseudocode that print courses in alphanumerical order

Current node equals head node

While node not equal to NULL

If headnode is greater than next node, swap nodes

Else move to next node

Print list created from start to finish

## Example Runtime Analysis

When you are ready to analyze the runtime for the Project One data structures for which you created the pseudocode, use the example chart below to support your work. This example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document. The example only covers the search function for the vector structure. You do not have to complete your runtime analysis until Project One. However, working on your analysis now may help you understand the changes as you complete the milestones. Don’t forget to include your charts in Project One. You will submit Project One in Module Six.

**Vector Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Best Case** | **Average Case** | **Worst Case** |
| Access | 0(1) | 0(1) | 0(1) |
| Search | 0(1) | 0(n) | 0(n) |
| Insert | 0(1) | 0(n) | 0(n) |
| Delete | 0(1) | 0(n) | 0(n) |

**Hash Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table** | **Best Case** | **Average Case** | **Worst Case** |
| Access | 0(1) | 0(1) | 0(n) |
| Search | 0(1) | 0(1) | 0(n) |
| Insert | 0(1) | 0(1) | 0(n) |
| Delete | 0(1) | 0(1) | 0(n) |

**Binary Search Tree**

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Search Tree** | **Best Case** | **Average Case** | **Worst Case** |
| Access | 0(log n) | 0(log n) | 0(n) |
| Search | 0(log n) | 0(log n) | 0(n) |
| Insert | 0(log n) | 0(log n) | 0(n) |
| Delete | 0(log n) | 0(log n) | 0(n) |

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | 5(n) |

Ranking the three scenarios based on runtime from best to worst we have:

1. 0(1) – being constant time
2. 0(log n) – logarithmic time
3. 0(n) – linear time

In the tables above, I analyzed the best, average, and worst case scenarios for each algorithm. The findings indicate that a vector, when considering average case scenarios, would be the least suitable option for our project, with a time complexity of O(n). The binary search tree follows as the second best option, with a time complexity of O (log n). Ultimately, the most favorable choice for our project is the hash table, which offers the fastest runtime analysis at O (1).

Every algorithm comes with its own set of pros and cons. Vectors, for instance, offer advantages such as scalability, editability, and versatility. Scalability allows them to be resized without loss of quality. Editability enables easy modifications, such as adding or removing items. Versatility means they are suitable for a wide variety of data points. However, vectors can be complex to create and may consume more memory compared to other algorithms. On the other hand, binary search trees excel in sorting and searching operations and are advantageous when data structures change frequently. Their drawbacks include the complexity of creation and increased memory usage with each child node after the root or parent. An unbalanced tree can also degrade performance. Hash tables have advantages such as efficient insertion and deletion, quick lookups, and flexibility. However, they face challenges like collisions when similar items exist, a maximum capacity that can lead to filling up, and difficulties in implementation. Additionally, hash tables cannot maintain a sorted order, making item retrieval more complicated.