# 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.

//Vector - Milestone 1

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 correct 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**

}

//Hash Table - Milestone 2

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) {

}

//Binary Search Tree – Milestone 3

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 a 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 alphanumeric 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 particular 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** | | | O(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

Looking at the tables above, I broke down the best case, average case and worst-case scenarios for each algorithm. The tables show that a vector using average case scenarios for each algorithm would be the worst to use for our project at 0(n). The next best scenario would be the binary search tree at 0(log n). Finally, the best scenario would be a hash table for our project as it has the fastest runtime analysis available at 0(1).

Each algorithm will have its advantages and disadvantages. Some of the advantages of a vector are scalability, editability, and versatility. Scalability means it can be made bigger or smaller without compromising quality. Editability because it’s easy to edit a vector whether inserting or deleting an item. Versatility because it can be used for many different types of data points. Some disadvantages of a vector are that it is complex to create and takes up more memory than other algorithms. Advantages of a binary search tree are sorting and searching algorithms. They are also used when data structures change frequently. Some disadvantages are that they are complex to create, the memory can take up more space for each child after the root or parent. Any unbalanced tree can lower performance. Advantages of a hash table are efficient for inserting and deleting, they have a fast look up process and are flexible to use. Some disadvantages are collisions, if like items exist it can cause collision issues. Hash tables also have a max capacity which means they can get full. They can also be hard to implement along with not being able to maintain a sorted order, which in turn makes retrieving items a bit hard.