CS 300 Pseudocode Document

Matt Zindler

Function Signatures

Below are the function signatures that you can fill in to address each of the three program requirements using each of the data structures. The pseudocode for printing course information, if a vector is the data structure, is also given to you below (depicted in bold).

```
// Vector pseudocode
void printSampleSchedule(Vector<Course> courses) {
Opening a file and reading the data:
Open the file.
Verify file exists.
Read the file.
While the file has unread lines:
      If readline has less than 2 parameters:
            Skip the line and output and error message.
      Else:
            If readline has a prerequisite
                  Check with previous prerequisites from the file
                  Validate the line
            Else:
                  Skip the line, and output an error message.
      Increment file line.
Stop reading the file.
}

int numPrerequisiteCourses(Vector<Course> courses, Course c) {
      totalPrerequisites = prerequisites of course c
      for each prerequisite p in totalPrerequisites
            add prerequisites of p to totalPrerequisites
      print number of totalPrerequisites
}

void printSampleSchedule(Vector<Course> courses) {
      While schedule has room
            generate random number for a course
            if course has completed prerequisite:
                  for all courses in the vector:
                        search Courses for course
                  add course to sample schedule
            else:
                  choose a new random course number
}
```

```
void printCourseInformation(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
}
void alphaNumericSort (Vector<Course> courses){
      Create new vector<alphaNumSort>
      For each course in courses:
            add to alphaNumSort alphabetically
      For each course in AlphaNumSort:
            if a letter has more than one course assigned to it:
                  if the courses at index have the same character:
                        increment letter
                  else:
                        sort courses alphanumerically
      print AlphaNumSort to screen
}

// Hashtable pseudocode

void readNewFile(Hashtable<Course> courses) {
      Opening a file and reading the data:
      Open the file.
      Verify file exists.
      Read the file.
      While the file has unread lines:
            If readline has less than 2 parameters:
                  Skip the line and output and error message.
            Else:
                  If readline has a prerequisite:
                        Check with previous prerequisites from the
file.
                        Validate the line.
                        insert line to the hashtable at the bucket =
                        courseNumber ID, or next closest bucket.
                  Else:
                        Skip the line, and output an error message.
            Increment file line.
            Stop reading the file.
}

int numPrerequisiteCourses(Hashtable<Course> courses) {
      For all courses:
```

```
            Search the hash table for each course:
            If the course has a prerequisite:
                  Increment total prerequisite courses.
            increment course
}

void printSampleSchedule(Hashtable<Course> courses) {
      While schedule has room
            generate random number for a course
            if course has completed prerequisite:
                  for all courses in the hashtable:
                        search bucket for course
                              if bucket does not match ID:
                                    check the next bucket
                  add course to sample schedule
            else:
                  choose a new random course number

}

void printCourseInformation(Hashtable<Course> courses, String
courseNumber) {
      Parse hashtable at courseNumber's ID bucket.
      If bucket contains courseNumber ID:
            Print info at courseNumber ID
      Else:
            While the bucket does not contain courseNumber ID:
                  Search next bucket in the list.
                  increment bucket number
            Print information and prerequisites at Bucket.

}


int hashTablestorage (Hashtable<courses> numCourses){
      courses = new HashTable(numCourses)
      for courses in pseudocode document:
            match course ID with bucket in the hashtable.
            if the bucket is empty:
                  put the course and additional info into the bucket.
            else:
                  while course ID is not assigned to a bucket:
                        move one bucket up and check contents.
                        if the bucket is empty:
                              fill the bucket with the ID.
                        else:
```

```
                              increment bucket number.
            increment file line.
}

void alphaNumericSort (Hashtable<Course> courses){
      Create new Hashtable<alphaNumSort>
      For each course in courses:
            Add to alphaNumSort alphabetically by bucket
            While bucket is full:
                  Search for a new bucket with the next letter
                  If no bucket exists:
                        Create a new bucket and insert course
                  Else:
                        Increment letter
      print AlphaNumSort to screen
}

// Tree pseudocode

int numPrerequisiteCourses(Tree<Course> courses) {
      For all courses in the tree:
            start at the root, and search through the tree
            if course has a prerequisite
                  increment num prerequisites
            move to the next branch
}

void printSampleSchedule(Tree<Course> courses) {
      While schedule has room
            generate random number for a course
            if course has completed prerequisite:
                  for all courses in the tree:
                        if course number is less than node:
                              search left branch
                        else:
                              search the right branch
                  add course to sample schedule
            else:
                  choose a new random course number

}

void printCourseInformation(Tree<Course> courses, String courseNumber)
{
      while the course does not equal the node
            if course number is less than node
```

```
                search left branch
            else
                search the right branch
        print course information
}

void alphaNumericSort (Tree<Course> courses){
        Create new Tree<alphaNumSort>
        For each course in courses:
            While course is less than curr node:
                check left node
            While course is greater than curr node:
                check right node
            if child node = null:
                add course as new child node
        print AlphaNumSort to screen
}
```

Menu Pseudocode:

```
void mainMenu (Vector<Course> courses, String courseNumber){
        Print display to user, "enter number for what function you
would             like to use"
        Collect user input
        If user input calls loadDataStructure:
            call loadDataStructure
        If user input calls printCorseList:
            call printCorseList
        If user input calls printCourse:
            call printCourse
        If user input calls quitProgram:
            call quitProgram
}

void loadDataStructure (Vector<Course> courses, String courseNumber){
        Open the file.
        Verify file exists.
        Read the file.
        While the file has unread lines:
            If readline has less than 2 parameters:
                Skip the line and output and error message.
            Else:
                If readline has a prerequisite:
                Check with previous prerequisites from the file.
                Validate the line.
```

```
                    insert line to the node sorted by course number
               Else:
                    Skip the line, and output an error message.
               Increment file line.
               Stop reading the file.
}

void printCourseList (Vector<Course> courses, String courseNumber){
     create new data structure organized alphabetically (vector)
     For each course in the data structure:
          go through each node and copy each course into the
          alphabetically sorted vector
     print alphabetically sorted vector
}

void printCourse (Vector<Course> courses, String courseNumber){
     For each course in the data structure:
          if the courseNumber matches:
               print course information (name/prerequisites)
          else:
               increment search location
}

void quitProgram (Vector<Course> courses, String courseNumber){
     terminate program
}
```

Example Runtime Analysis

When you are ready to begin analyzing the runtime for the data structures that you have created pseudocode for, use the 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.

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	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)

Runtime analysis:

Worst Case	**Reading a file**	**Creating course objects**	**Cost per line**
Vector	**n**	**n**	**1**
Hashtable	**n**	**n**	**1**
Tree	**n**	**log*n***	**1**

Worst Case	**Reading a file**	**Creating course objects**
Vector	**4 + n(1+n+n+1) = 4 + n(2+n)**	**n**

Worst Case	**Reading a file**	**Creating course objects**
Hashtable	**4 + n(1+n+n+1) = 4 + n(2+n)**	**n**

Worst Case	**Reading a file**	**Creating course objects**
Tree	**4 + n(1+n+n+1) = 4 + n(2+n)**	**log*n***

Trees:

Advantages:

Fast/efficient insertion and sorting of data into the tree, only needs to check one single lineage of a branch

Easy to manipulate as needed: swapping, increasing/decreasing size.

Disadvantages:

Can take up more space than other storage methods

Vectors:

Advantages:

Can store multiple data types.

Doesn’t require changing or extra analysis to sort data

Variable storage size

Disadvantages:

Can take extra time to sort based on multiple data types

Longer worst case scenario than trees

Hash tables:

Advantages:

Average faster search than vector and trees: search the first known bucket where it likely is, then each bucket after

Disadvantages:

Longer worst case scenario than trees

Collisions are likely

the less buckets there are, the more inefficient it becomes

For sorting courses, I think that hash tables will be the most efficient data structure to use. If we are sorting by courses, we can have an amount of buckets equal to how many courses there are, ensuring that there are no collisions. This negates the main downside of hash tables, when you would have collisions. Searching and sorting would be easy, as each course would have its spot that it would be sent to every time. This means that the run time for searching and sorting each course would be the best case run time of O(1).