Kennedy Uzoho

SNHU

Module 6-2

ABCU CS curriculum Pseudocode/Data Structure Evaluation

Project One

Pseudocode for a 'Menu' key

- Load the file data into the data structure. Before you can print the course information or the sorted list of courses, you must load the data into the data structure.

- Print Course List: This will print an alphanumerically ordered list of all the courses in the Computer Science department.
- Print Course: This will print the course title and the prerequisites for any individual course.
- Exit: This will exit you out of the program.

Create a schedule object to hold courses

```
Initialize string coursekey
Initialize Course acourse
Initialize int choice to 0
Initialize int choice2 to 0
     while (choice! = 9) {
          output "Menu:"
          output " 1. Load Data Structure\n"
          output " 2. Print Course List\n"
          output " 3. Print Course\n"
          output " 9. Exit\n"
          output "Enter choice: "
          wait for input and store in choice

          switch (choice) {

     case 1:
          LoadCourses(fileName, schedule)
          break
     case 2:
          while (choice2 == 0) {
               output "1). Display Schedule\n"
               output "2). Display Course\n"
               output "Enter choice: "
               wait for input and store in choice2
               switch (choice2) {
               case 1:
                    print schedule
```

```
                    break
               case 2:
                    output "Enter course number: "
                    wait for input and store in courseKey
                    set acourse to schedule.Search(courseKey)
                    if (acourse is empty) output "Course is not in
schedule.\n"
                    else print acourse
                    break
               }
          }
          Set choice2 to 0
          break
     case 3:
          output "Enter course number: "
          wait for input and store in courseKey

          if (coursekey is not found in schedule) {
               output "Course does not exist.\n"
               break
          }
          else remove courseKey from schedule
          output courseKey " removed.\n"
          break
     }
}
output "goodbye.\n"
```

Vectors

Designing pseudocode to define how the program opens a file reads data from the file, parses each line, and checks for file format errors.

```
FUNCTION readFile(File f, lines[])
     courseNumbers[], courseTitles[], prerequisites[], line
     i = 0, j = 0
     Flag = TRUE
     WHILE (NOT END OF FILE f)
          courseInfo[] = SPLIT (READLINE(f, line), DELIMETER = , )
          APPEND line TO lines
          IF (LENGTH of courseInfo < 2)
               Flag = FALSE
               BREAK
       END IF
       courseNumbers[i] = courseInfo[0]
          courseTitles[i] = courseInfo[1]
          INCREMENT i
```

```
            IF (LENGTH of courseInfo > 2)
             FOR k = 2 to LENGTH of courseInfo
                 prerequisites[j] = courseInfo[k]
                 INCREMENT j
             END FOR
         END IF
      END WHILE

      IF Flag == TRUE
           FOR each P in prerequisites
                IF P NOT IN courseNumbers
                  Flag = FALSE
                  BREAK
              END IF
         END FOR
      END IF

    RETURN Flag
END FUNCTION
```

Designing pseudocode to show how to create course objects and store them in the appropriate data structure.

```
CLASS Course
      Number: String
      Title: String
      Prerequisites []: String []

      CONSTRUCTOR Course(line)
           Number = SPLIT (line, DELIMETER =,) [0]
           Title = SPLIT (line, DELIMETER =,) [1]

           IF LENGTH of SPLIT (line, DELIMETER =,) > 2
            Prerequisites = SPLIT (line)[ 2 to LENGTH of SPLIT (line,
DELIMETER = ,)]
         END IF
      END CONSTRUCTOR
END CLASS

FUNCTION createObject(Courses <Course>, File f)
    Lines[] = " "
      IF readFile(f, Lines) == TRUE
           FOR each Line in Lines
            APPEND NEW Course(Line) TO Courses
           END FOR
      END IF

      ELSE PRINT("File cannot be read")
      END ELSE
```

```
END FUNCTION
```

Designing pseudocode that will search the data structure for a specific course and print out the course information and prerequisites.

```
FUNCTION MAIN ()
     Filename = INPUT ()
     File F = NEW File (Filename)
     Courses <Course>: vector

     CALL: createObject(Courses, F)
    CourseNumber = INPUT ()

    IF Courses is EMPTY
        PRINT ("No objects read from the file")
    END IF

    ELSE
     printCourseInformation (Courses, CourseNumber)
    END ELSE
END FUNCTION
```

Vector pseudocode

```
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) {
     for all key, value pair in courses
      print key course name
          if value has prerequisits
              for each prerequisits
                  print prerequisits

}

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
}
```

Hashtable pseudocode

```
int numPrerequisiteCourses(Hashtable courses, Course c) {

   totalPrerequisites = Hashtable[c]
   for each prerequisite p in totalPrerequisites
       add prerequisites in Hashtable[p] to totalPrerequisites
   print number of totalPrerequisites
}
void printSampleSchedule(Hashtable courses) {

   for all key, value pair in courses
       print key course name
           if value has prerequisits
               for each prerequisits
                   print prerequisits
}
void printCourseInformation(Hashtable courses, String courseNumber) {

   for all courses
       if the course is the same as courseNumber
           print out the course information
           for each prerequisite of the Hashtable[course]
               print the prerequisite course information
}
```

Tree pseudocode

```
int numPrerequisiteCourses(Tree courses, Node c) {
   totalPrerequisites = left and right child of Node c
   for each prerequisite p in totalPrerequisites
       add left and right Nodes of node p to totalPrerequisites
   print number of totalPrerequisites
}
void printSampleSchedule(Tree courses) {

   for all Nodes as courses
       print course name
           if course has left node
                   print left node as prerequisite
           if course has right node
                   print right node as prerequisite
}
void printCourseInformation(Tree courses, String courseNumber) {
   for all Nodes
       if the course is the same as courseNumber
           print out the node's information
           if course has left node
```

```
                print left node as prerequisite couse information
        if course has right node
                print right node as prerequisite couse information
       end Function
    else
        if course has left node
                goto left node
        if course has right node
                goto right node
}
```

Runtime Analysis

Vector

Code	Line Cost	# Times Executes	Total Cost
totalPrerequisites = prerequisites of course c	1	n	n
for each prerequisite P in totalPrerequisites	1	n	n
add prerequisites of p to totalPrerequisites	1	1	1
print number of totalPrerequisites	1	n	n
For all key, value pair in course	1	n	n
print key course name	1	1	1
if value has prerequisits	1	n	n
for each prerequisits	1	n	n
print prerequisits	1	1	1
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	**10n+3**
		Runtime	**0(n)**

Hash Table

Code	Line Cost	# Times Executes	Total Cost
totalPrerequisites = Hashtable[c]	1	n	n
for each prerequisite p in totalPrerequisites	1	n	n

add prerequisites in Hashtable[p] to totalPrerequisites	1	1	1
print number of totalPrerequisites	1	n	n
For all key, value pair in course	1	n	n
print key course name	1	1	1
if value has prerequisits	1	n	n
for each prerequisits	1	n	n
print prerequisits	1	1	1
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 Hashtable[course]	1	n	n
Print the prerequisite course information	1	n	n
		Total Cost	**9n+3**
		Runtime	**0(n)**

Tree

Code	**Line Cost**	**# Times Executes**	**Total Cost**
totalPrerequisites = left and right child of Node c	1	n	n
for each prerequisite p in totalPrerequisites	1	n	n
add left and right Nodes of node p to totalPrerequisites	1	1	1
print number of totalPrerequisites	1	1	1
for all Nodes as courses	1	n	n
print course name	1	1	1
if course has left node	1	n	n
print left node as prerequisite	1	1	1
if course has right node	1	n	n
print right node as prerequisite	1	1	1
for all Nodes	1	n	n
if the course is the same as courseNumber	1	n	n
print out the node's information	1	1	1
if course has left node	1	n	n
print left node as prerequisite couse information	1	1	1
if course has right node	1	n	n

print right node as prerequisite couse information	1	1	1
end Function	1	1	1
else	1	n	n
if course has left node	1	n	n
goto left node	1	1	1
if course has right node	1	n	n
goto right node	1	1	1
		Total Cost	**12n+9**
		Runtime	**0(n)**

Difference between Big O (1) vs Big O (n)
Big O (n) involves iteration that expands the size of the data structure as it iterates

Big O (1) is in constant time, moves linearly and the size of the structure does not enlarge as the functions increase.

Evaluation

Vector

Pros:

1. Easy to implement and understand
2. Searchable in O (log n) time if sorted with binary search
3. Insertion at the back is in constant time

Cons:

1. Must be sorted to take full advantage of search capabilities
2. Removing items from the front takes linear time because of shifting
3. Depending on the compiler used reallocation of the vector may take up more space than needed

Hash Table

Pros:

1. Direct access to items table
2. Able to insert and delete in constant time no matter the size of the table
3. When implemented correctly, hash tables can be the best data structure in terms of speed

Cons:

1. Consume more space than what is needed
2. No order to retrieve elements
3. Randomly stores elements in memory which can cause cache misses, resulting

in long delays.

Tree

Pros:

1. Able to retrieve items in order
2. Able to Insert and delete in O (log n) time
3. Speed is sufficient

Cons:

1. For best performance Tree must maintain balance
2. May quickly cause stack overflow when using recursion and iteration
3. The shape of the tree depends on the first item inserted

Recommendation

After working on all three data structures, I am recommending the binary search Tree for storing course objects. The binary search Tree does a good job of displaying courses in alphabetical order. There is no sorting needed to be done. In comparison to the other two data structures Hash Table and Vector sorting are needed before arrangement. Searching the binary Tree on average takes about O (log n) time. Using a Hash Table, one must have a good knowledge of sorting and knowledge of the data being sorted to use a Hash Table.