Project One – ABCU Pseudocode & Runtime Analysis

Emeth Jamison

CS300 - Data Structures and Algorithms

Dr. Aline Yurik

December 8, 2024

Pseudocode

Vector

```
STRUCT Course {
   INSTANTIATE two strings: courseNumber and courseName
   INSTANTIATE a list vector of type String called prerequisites
}

VOID loadDataFromFile(Vector<Course> courses, String filename) {
   OPEN file(filename)

   WHILE there are lines to read
      LINE = readNextLine()
      Tokens = split(LINE, ",")
      courseNumber = Tokens[0]
      courseName = Tokens[1]
      prerequisites = empty list

      FOR EACH token from the third token to the last index in Tokens
         APPEND the current index of Tokens to prerequisites

      newCourse = new Course(courseNumber, courseName, prerequisites)
      APPEND newCourse to courses vector

   CLOSE file()
}

BOOL validateFile(Vector<Course> courses, String filename) {
```

```
    OPEN file(filename)

    WHILE there are lines to read
        LINE = readNextLine()
        Tokens = split(LINE, ",")

        IF length(Tokens) < 2 THEN
            PRINT("Error: Invalid format in line " + LINE)
            RETURN false

        courseNumber = Tokens[0]
        prerequisites = Tokens[2:]

        FOR EACH prerequisite IN prerequisites
            IF !courseExists(courses, prerequisite) THEN
                PRINT("Error: Prerequisite " + prerequisite + " does not exist for course " +
courseNumber)
                RETURN false

    CLOSE file()
    RETURN true
}

BOOL courseExists(Vector<Course> courses, String courseNumber) {
    FOR EACH course IN courses
        IF course.courseNumber == courseNumber
            RETURN true
    RETURN false
}
```

```
VOID searchCourse(Vector<Course> courses, String courseNumber) {
   FOR EACH course IN courses
      IF course.courseNumber == courseNumber
         PRINT("Course Number: " + course.courseNumber)
         PRINT("Course Name: " + course.courseName)

         IF course.prerequisites is empty
            PRINT("No prerequisites")
         ELSE
            PRINT("Prerequisites:")
            FOR EACH prerequisite IN course.prerequisites
               PRINT("- " + prerequisite)

         RETURN
   PRINT("Course not found")
}
```

Hash Table

```
STRUCT Course {
   INSTANTIATE String courseNumber
   INSTANTIATE String courseName
   INSTANTIATE List<String> prerequisites
}

VOID loadDataFromFile(HashTable<String, Course> courses, String filename) {
   OPEN file(filename)

   WHILE there are lines to read
```

```
        LINE = readNextLine()
        Tokens = split(LINE, ",")

        IF length(Tokens) < 2 THEN
            PRINT "Error: Invalid format in line"
            CONTINUE
        END IF

        courseNumber = Tokens[0]
        courseName = Tokens[1]
        prerequisites = EMPTY list

        FOR index = 2 TO length(Tokens) - 1 DO
            APPEND Tokens[index] TO prerequisites
        END FOR

        newCourse = new Course(courseNumber, courseName, prerequisites)
        INSERT newCourse INTO courses WITH courseNumber AS the key

    CLOSE file()
}

BOOL validateFile(HashTable<String, Course> courses, String filename) {
    OPEN file(filename)

    WHILE there are lines to read
        LINE = readNextLine()
        Tokens = split(LINE, ",")
```

```
        IF length(Tokens) < 2 THEN
            PRINT "Error: Invalid format in line"
            RETURN false
        END IF

        courseNumber = Tokens[0]
        prerequisites = Tokens[2 TO END]

        FOR EACH prerequisite IN prerequisites DO
            IF !courseExists(courses, prerequisite) THEN
                PRINT "Error: Prerequisite does not exist for course " + courseNumber
                RETURN false
            END IF
        END FOR

    CLOSE file()
    RETURN true
}

BOOL courseExists(HashTable<String, Course> courses, String courseNumber) {
    RETURN courseNumber EXISTS as a key IN courses
}

VOID searchCourse(HashTable<String, Course> courses, String courseNumber) {
    IF courseNumber EXISTS as a key IN courses THEN
        course = courses[courseNumber]
        PRINT "Course Number: " + course.courseNumber
        PRINT "Course Name: " + course.courseName
```

```
        IF course.prerequisites IS empty THEN
            PRINT "No prerequisites"
        ELSE
            PRINT "Prerequisites:"
            FOR EACH prerequisite IN course.prerequisites DO
                PRINT prerequisite
            END FOR
        END IF
    ELSE
        PRINT "Course not found"
    END IF
}
```

Binary Search Tree

```
STRUCT Course {
    INSTANTIATE String courseNumber
    INSTANTIATE String courseName
    INSTANTIATE List<String> prerequisites
}

VOID loadDataIntoTree(Tree<Course> courseTree, String filename) {
    OPEN file(filename)

    WHILE there are lines to read
        LINE = readNextLine()
        Tokens = split(LINE, ",")

        IF length(Tokens) < 2 THEN
```

```
        PRINT "Error: Invalid format in line"
        CONTINUE
    END IF

    courseNumber = Tokens[0]
    courseName = Tokens[1]
    prerequisites = EMPTY list

    FOR index = 2 TO length(Tokens) - 1 DO
        APPEND Tokens[index] TO prerequisites
    END FOR

    newCourse = new Course(courseNumber, courseName, prerequisites)
    INSERT newCourse INTO courseTree
  END WHILE

  CLOSE file()
}

BOOL validateFile(Tree<Course> courseTree, String filename) {
  OPEN file(filename)

  WHILE there are lines to read
    LINE = readNextLine()
    Tokens = split(LINE, ",")

    IF length(Tokens) < 2 THEN
      PRINT "Error: Invalid format in line"
      RETURN false
```

```
        END IF

        courseNumber = Tokens[0]
        prerequisites = Tokens[2 TO END]

        FOR EACH prerequisite IN prerequisites DO
            IF !courseExists(courseTree, prerequisite) THEN
                PRINT "Error: Prerequisite " + prerequisite + " does not exist for course " +
courseNumber
                RETURN false
            END IF
        END FOR

    CLOSE file()
    RETURN true
}

BOOL courseExists(Tree<Course> courseTree, String courseNumber) {
    RETURN courseNumber EXISTS IN courseTree
}

VOID searchAndPrintCourse(Tree<Course> courseTree, String courseNumber) {
    Course course = FIND course IN courseTree WHERE course.courseNumber = courseNumber

    IF course IS NOT NULL THEN
        PRINT "Course Number: " + course.courseNumber
        PRINT "Course Name: " + course.courseName

        IF course.prerequisites IS empty THEN
            PRINT "No prerequisites"
```

```
    ELSE
      PRINT "Prerequisites:"
      FOR EACH prerequisite IN course.prerequisites DO
        PRINT prerequisite
      END FOR
    END IF
  ELSE
    PRINT "Course not found"
  END IF
}

VOID printAllCourses(BST<Course> courseTree) {
  FUNCTION inOrderTraversal(Node current) {
    IF current IS NOT NULL THEN
      inOrderTraversal(current.leftChild)
      PRINT "Course Number: " + current.course.courseNumber
      PRINT "Course Name: " + current.course.courseName

      IF current.course.prerequisites IS empty THEN
        PRINT "No prerequisites"
      ELSE
        PRINT "Prerequisites:"
        FOR EACH prerequisite IN current.course.prerequisites DO
          PRINT prerequisite
        END FOR
      END IF
      inOrderTraversal(current.rightChild)
    END IF
  }
  inOrderTraversal(courseTree.root)
```

```
}
```

Universal Menu Pseudocode

```
VOID displayMenu() {
   PRINT "1. Load file data into data structure"
   PRINT "2. Print all courses in alphanumeric order"
   PRINT "3. Print a course's title and prerequisites"
   PRINT "9. Exit"
   PRINT "Enter your choice:"
}

VOID handleMenuOption(int option, DataStructure<Course> dataStructure, String filename) {
   SWITCH(option) {
      CASE 1:
         IF loadDataFromFile(dataStructure, filename) THEN
            PRINT "File data loaded successfully."
         ELSE
            PRINT "Error loading file data."
         END IF
         BREAK

      CASE 2:
         printAllCourses(dataStructure)
         BREAK

      CASE 3:
         PRINT "Enter course number:"
         courseNumber = getInput()
```

```
            searchAndPrintCourse(dataStructure, courseNumber)
            BREAK

        CASE 9:
            PRINT "Exiting program."
            RETURN

        DEFAULT:
            PRINT "Invalid option. Please try again."
    }
}
```

Runtime Analysis

Vector

Pseudocode Line	**Line Cost**	**# Times Executed**	**Total Cost**
Create courses vector	**1**	**1**	**1**
Read file lines (While loop)	**1**	**n**	**n**
Split line into tokens	**1**	**n**	**n**
Parse tokens	**1**	**n**	**n**
Create empty prerequisites list	**1**	**n**	**n**
For each token from the third to the last index, Parse prerequisites to new prerequisites list	**1**	**n * m**	**n * m**
Append newCourse to courses vector	**1**	**n**	**n**
Total Cost:			**O(n*m)**

The vector data structure has the advantages of simplicity and ease regarding implementation, the ability to dynamically resize, and efficient memory usage. These advantages make lists of unknown size easier to manage. The disadvantages of using a vector are search efficiency due to

linear scanning, insertion and deletion efficiency, and the need to reallocate and copy information when resizing a vector.

Hash Table

Pseudocode Line	**Line Cost**	**# Times Executed**	**Total Cost**
Create new courses hash table	**1**	**1**	**1**
Read file lines (While loop)	**1**	**n**	**n**
Split line into tokens	**1**	**n**	**n**
Parse Tokens			
Create empty prerequisites list	**1**	**n**	**n**
For each token from the third to the last index, Parse prerequisites to new prerequisites list	**1**	**m**	**n * m**
Insert newCourse into courses hash table with the courseNumber as the key	**1**	**n**	**n**
Validate Prerequisites	**1**	**n * m**	**n * m**
Total Cost:			**O(n*m)**

The hash table data structure has a few advantages including efficient searching due to searching by keys, dynamic resizing capabilities, and un-ordered data, making performance unaffected by data order and size. The disadvantages of hash tables is that they can introduce collisions in worst-case scenarios, they use more memory due to the need for a hashing function, and the complexity of a well-designed hashing function.

Binary Search Tree

Pseudocode Line	**Line Cost**	**# Times Executed**	**Total Cost**
Create new courses hash table	**1**	**1**	**1**
Read file lines (While loop)	**1**	**n**	**n**
Split line into tokens	**1**	**n**	**n**
Parse Tokens			
Create empty prerequisites list	**1**	**n**	**n**
For each token from the third to the last index, parse prerequisites to new prerequisites list	**1**	**m**	**n * m**
Insert newCourse into courses tree	***O*(logn)**	**n**	**n**
Validate Prerequisites	***O*(logn)**	**n * m**	**n * m**
Total Cost:			***O*(n*m*logn)**

The advantages of using binary search trees include search efficiency, sorted orders of elements, and the ability to traverse in order. The disadvantages of binary search trees include

computational overhead to balance the trees, inefficiency in operations if the trees are unbalanced (worst-case), and the complexity of implementation compared to the other data structures.

As a result of this analysis, my recommendation for the data structure to implement is the binary search tree. Although the performance time may end up being slightly higher than the other data structures, the ability of a binary search tree to order data correctly suits the needs of a class organization structure well. Data sets of classes will likely not be very large, which shifts the emphasis for the system from fastest runtime to better structure. Overall, the binary search tree maintains a sorted order of parent-child relationships that is characteristic of the needs of the ABCU Computer Science department.