Data Structures & Algorithms Project One

Marissa Sihapanya

# Vector Data Structure Pseudocode

struct Course

define number as string

define name as string

define prereqs as vector of strings

vector<Course> parseCourses(String path, char sep)

courses = vector<Course>

open file with file path

if file is not open

throw open file error

while not end of file

line = file.readLine()

if line is not empty

tokens = []

currToken = ""

for each char in line

if char == sep

if currToken != empty

tokens.push\_back(currToken)

currToken = ""

else currToken += char

if currToken != empty

tokens.push\_back(currToken)

if tokens.size < 2

throw "File format error, course details not all present"

prereqs = []

if tokens.size > 2

for i from 2 to tokens.size - 1

prereqs.push\_back(tokens[i])

course = new Course(tokens[0], tokens[1], prereqs)

courses.push\_back(course)

return courses

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

}

void sortCourses(vector<Course> courses)

for i from 0 to end of courses

min = i

for j from i + 1 to end of courses

if courses[j].number < courses[min].number

min = j

swap courses[i] and courses[min]

void printCourses

for each course in vector courses

for each prereq in course.prereqs

print course.number << course.name << course.prereqs

int main():

choice = 0

while choice != 9:

print menu options

input choice

switch choice:

case 1:

parseCourses(filePath, ',')

break

case 2:

sortCourses(courses)

printCourses(courses)

break

case 3:

searchCourse(courses, courseNumber)

break

case 9:

exit program

break

# Hash Table Data Structure Pseudocode

Define the Course structure

define number as string

define name as string

define prereqs as vector of strings

class HashTable {

Define the Node structure

instantiate course

declare key

declare node next ptr

define Node default constructor

initialize with a course

initialize with a course and a key

Declare a node containing a course and a key

Declare a vector of nodes

Assign tableSize to DEFAULT\_SIZE

Declare hash function taking in a key

}

HashTable::HashTable() {

resize the nodes vector based on the table size

}

HashTable::HashTable(unsigned int size) {

invoke local tableSize member to size

resize the size of nodes vector to the provided size

}

HashTable::~HashTable() {

Destruct and clear all nodes

}

unsigned int HashTable::hash(string courseNumber) {

declare a string number to represent numeric part of courseNumber

for each character in courseNumber

if character is a digit

add the character to number

declare an integer key and use stoi to convert string number to int

return key mod tableSize

}

void HashTable::Insert(Course course) {

create the key for the given course with the hash function using course.number

if bucket at nodes[key] is empty

assign new node to the key position

insert the new node at the beginning plus key hash offset

else

while walk the linked list chain until an empty space is found

iterate to the next node

when an empty space is found

assign the node's next pointer to a new node (insert the new course)

}

void HashTable::ParseCourses(csvPath, sep) {

Instantiate a file object as file from the Parser method and specify the csv path of the file

open file with file path

if file is not open

throw open file error

while not end of file

line = file.readLine()

if line is not empty

tokens = []

currToken = ""

for each char in line

if char == sep

if currToken != empty

tokens.push\_back(currToken)

currToken = ""

else currToken += char

if currToken != empty

tokens.push\_back(currToken)

if tokens.size < 2

throw "File format error, course details not all present"

prereqs = []

if tokens.size > 2

for i from 2 to tokens.size - 1

prereqs.push\_back(tokens[i])

course = new Course(tokens[0], tokens[1], prereqs)

Assign new key to hash function on course.number

Insert(course)

}

Define a method to split each line in the file and store it in a tokens vector object

Define a vector tokens of strings

Define a single token variable as a string

Open a new string stream to read the line from the file

While getting the string stream and splitting each line into its tokens

Push back the parsed token into the tokens vector

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

calculate the key based on hashing the courseNumber

declare the node at the pointer of the key location in courses

traverse linked list at node

while node is not null

if node.course.number equals courseNumber

print the course details at the node

for each prequisite of the course at the node

print the prerequisite course information

if no entry found

print not found message

}

void sortCourses

vector<Course> courses

for each node in nodes

if node is not null

courses.push\_back(node->course)

node = node->next

std::sort(courses.begin(), courses.end(), course a, course b )

return a.number < b.number

clear hash table

for each course in courses

Insert(course)

void printCourses

for each node in nodes

while node is not null

print node->course.number

print node->course.name

for each prereq in node->course.prereqs

print prereq

node = node->next

int main():

choice = 0

while choice != 9:

print menu options

input choice

switch choice:

case 1:

ParseCourses(filePath, ',')

break

case 2:

sortCourses(hashTable)

printCourses(hashTable)

break

case 3:

searchCourse(courses, courseNumber)

break

case 9:

exit program

break

# Binary Search Tree Data Structure Pseudocode

struct *Course*

    number: as string

    name: as string

    prereqs: as vector of strings

*Constructor*(course: *Course*) {

        this.course = course

    }

struct *Node*

    initialize course

    initialize left node

    initialize right node

    default constructor for node initialize left and right to null

*Node*(*Course* aCourse) {

*Node*() {

            set course to a course

        }

    }

class BinarySearchTree {

    define private variables

        root node

        addNode method (node, courseNumber)

        inOrder method (node)

    define public variables

        BinarySearchTree() constructor

        BST destructor

*void* deleteNodes(*node*)

        void InOrder

*void* Insert(*course*)

}

*BinarySearchTree*::*BinarySearchTree*() {

    set root to nullptr

}

*void* *BinarySearchTree*::deleteNodes(*Node*\* node) {

    if node is not null

        deleteNodes(node.left)

        deleteNodes(node.right)

        delete node

}

*BinarySearchTree*::~*BinarySearchTree*() {

    deleteNodes(root)

}

*void* *BinarySearchTree*::*Insert*(*Course* course) {

    if root is null

        root = new *Node*(course)

    else addNode(root, course)

}

*void* *BinarySearchTree*::addNode(*Node*\* node, *Course* course) {

    if course.number < node.course.number

        traverse left

            if node.left == null

                node.left = new *Node*(course)

            else

                recurse addNode(node->left, course) and traverse left

    if course.number > node.course.number

        traverse right

            if right child is null

                node.right = new *Node*(course)

            else

                recurse addNode(node->right, course) and traverse right

}

*void* *BinarySearchTree*::inOrder(*Node*\* node) {

    if node is not null

        inOrder(node.left)

        print(node))

        inOrder(node.right)

}

*Course* *BinarySearchTree*::*Search*(string courseNumber) {

    currNode = root

    while currNode is not null

        if courseNumber == currNode.course.number

            return currNode.course

        else if courseNumber < currNode.course.number

            currNode = currNode.left

        else if courseNumber > currNode.course.number

            currNode = currNode.right

    instantiate course

    return course

}

vector<*Course*> parseCourses(*String* path, *char* sep)

    courses = vector<*Course*>

    open file with file path

    if file is not open

        throw open file error

    while not end of file

        line = file.readLine()

        if line is not empty

            tokens = []

            currToken = ""

            for each *char* in line

                if *char* == sep

                    if currToken != empty

                    tokens.push\_back(currToken)

                    currToken = ""

                else currToken += *char*

            if currToken != empty

                tokens.push\_back(currToken)

            if tokens.size < 2

                throw "File format error, course details not all present"

            prereqs = []

            if tokens.size > 2

                for i from 2 to tokens.size - 1

                prereqs.push\_back(tokens[i])

            course = new *Course*(tokens[0], tokens[1], prereqs)

            courses.push\_back(course)

        return courses

printCourse():

    course = searchCourse

    print course number

    print course name

    for each prereq in course.prereqs

        print each prereq

main() {

    courses = parseCourses("file.csv", ',')

    bst = new *BinarySearchTree*()

    for course in courses

        bst.insert(course)

    print all courses

*InOrder*(root)

}

*int* main():

    choice = 0

    while choice != 9:

        print menu options

        input choice

        switch choice:

            case 1:

                courses = parseCourses("file.csv", ',')

                bst = new *BinarySearchTree*()

                for each course in courses

                    bst->*Insert*(course)

                break

            case 2:

                InOrder(*root*)

                break

            case 3:

                printCourse()

                break

            case 9:

                exit program

                break

# Run time and memory of data structures evaluation

## Vector

#### Parse courses method

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Courses = vector<Course>** | 1 | 1 | 1 |
| **Open file with file path** | 1 | 1 | 1 |
| **If file is not open** | 1 | 1 | 1 |
| **Throw open file error** | 1 | 1 | 1 |
| **While not end of file** | 1 | 1 | 1 |
| **Line = file.readLine()** | 1 | n | n |
| **If line is not empty** | 1 | n | n |
| **Tokens = []** | 1 | n | n |
| **currToken = “”** | 1 | n | n |
| **For each char in line** | 1 | m | m |
| **If char == sep** | 1 | m | m |
| **If currToken != empty** | 1 | m | m |
| **Tokens.push\_back(currToken)** | 1 | m | m |
| **currToken = “”** | 1 | n | n |
| **Else currToken += char** | 1 | n | n |
| **If tokens.size < 2** | 1 | n | n |
| **Throw “File format error, course details not all present”** | 1 | 1 | 1 |
| **Prereqs = []** | 1 | n | n |
| **If tokens.size > 2** | 1 | p | p |
| **For I from 2 to tokens.size - 1** | 1 | 1 | 1 |
| **Prereqs.push\_back(tokens[i])** | 1 | np | np |
| **Course = new Course(tokens[0], tokens[i], prereqs)** | 1 | n | n |
| **Courses.push\_back(course)** | 1 | n | n |
| **Return courses** | 1 | 1 | 1 |
| **Total Cost** | | | 10n + 4m + np + p + 1 |
| **Runtime** | | | O(n) |

Where m = length of line read and p = # of prerequisites

## Hash Table

#### Hash method

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **String number = courseNumber** | 1 | 1 | 1 |
| **For each char in courseNumber** | 1 | 1 | 1 |
| **If char = digit** | 1 | n | n |
| **Add char to number** | 1 | n | n |
| **Int key = stoi(number)** | 1 | 1 | 1 |
| **Return key % tableSize** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 1 |
| **Runtime** | | | O(n) |

#### Insert method

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Call hash function** | 1 | 1 | 1 |
| **If bucket at nodes[key] is empty** | 1 | 1 | 1 |
| **New node = nodes[key]** | 1 | 1 | 1 |
| **Insert new node at beginning plus key hash offset** | 1 | 1 | 1 |
| **While walk the linked list chain until an empty space is found** | 1 | n | n |
| **Node = node->next** | 1 | n | n |
| **When empty space is found** | 1 | 1 | 1 |
| **Node->next = new node(course)** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 1 |
| **Runtime** | | | O(n) |

#### ParseCourses method

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **open file with file path** | 1 | 1 | 1 |
| **If file is not open** | 1 | 1 | 1 |
| **Throw open file error** | 1 | 1 | 1 |
| **While not end of file** | 1 | 1 | 1 |
| **Line = file.readLine()** | 1 | n | n |
| **If line is not empty** | 1 | n | n |
| **Tokens = []** | 1 | n | n |
| **currToken = “”** | 1 | n | n |
| **For each char in line** | 1 | m | m |
| **If char == sep** | 1 | m | m |
| **If currToken != empty** | 1 | m | m |
| **Tokens.push\_back(currToken)** | 1 | m | m |
| **currToken = “”** | 1 | 1 | 1 |
| **Else currToken += char** | 1 | n | n |
| **If currToken != empty** | 1 | n | n |
| **Tokens.push\_back(currToken)** | 1 | n | n |
| **If tokens.size < 2** | 1 | n | n |
| **Throw “File format error, course details not all present”** | 1 | 1 | 1 |
| **Prereqs = []** | 1 | n | n |
| **If tokens.size > 2** | 1 | p | p |
| **For I from 2 to tokens.size - 1** | 1 | 1 | 1 |
| **Prereqs.push\_back(tokens[i])** | 1 | np | np |
| **Course = new Course(tokens[0], tokens[1], prereqs)** | 1 | n | n |
| **Assign new key to hash function on course.number** | 1 | n | n |
| **Insert(course)** | 1 | n | n |
| **Total Cost** | | | 12n + 4m + np + p + 1 |
| **Runtime** | | | O(n) |

## BST

#### Insert method

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If root is null** | 1 | 1 | 1 |
| **Root = new Node(course)** | 1 | 1 | 1 |
| **Else addNode(root, course)** | 1 | n | n |
| **Total Cost** | | | n + 1 |
| **Runtime** | | | O(n) |

#### addNode method

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If course.number < node.course.number** | 1 | 1 | 1 |
| **If node.left == null** | 1 | 1 | 1 |
| **Node.left = new Node(course)** | 1 | 1 | 1 |
| **Else recurse addNode(node->left, course)** | 1 | n | n |
| **If course.number > node.course.number** | 1 | 1 | 1 |
| **If right child is null** | 1 | 1 | 1 |
| **Node.right = new Node(course)** | 1 | 1 | 1 |
| **Else recurse addNode (node->right, course)** | 1 | n | n |
| **Total Cost** | | | 2n + 1 |
| **Runtime** | | | O(n) |

#### parseCourses method

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Courses = vector<Course>** | 1 | n | n |
| **Open file with file path** | 1 | n | n |
| **If file is not open** | 1 | 1 | 1 |
| **Throw open file error** | 1 | 1 | 1 |
| **While not end of file** | 1 | 1 | 1 |
| **Line = file.readLine()** | 1 | n | n |
| **If line is not empty** | 1 | n | n |
| **Tokens = []** | 1 | n | n |
| **currToken = “”** | 1 | n | n |
| **For each char in line** | 1 | m | m |
| **If char == sep** | 1 | m | m |
| **If currToken != empty** | 1 | m | m |
| **Tokens.push\_back(currToken)** | 1 | m | m |
| **If tokens.size < 2** | 1 | n | n |
| **Throw “File format error, course details not all present”** | 1 | 1 | 1 |
| **Prereqs = []** | 1 | n | n |
| **If tokens.size > 2** | 1 | p | p |
| **For I from 2 to tokens.size - 1** | 1 | 1 | 1 |
| **Prereqs.push\_back(tokens[i])** | 1 | np | np |
| **Course = new Course(tokens[0], tokens[i], prereqs)** | 1 | n | n |
| **Courses.push\_back(course)** | 1 | n | n |
| **Return courses** | 1 | 1 | 1 |
| **Total Cost** | | | 10n + 4m + np + p + 1 |
| **Runtime** | | | O(n) |

# Analysis of data structures

## Vector

#### Advantages

Vectors are best suited when the size of the dataset is known. They are simple, easy to use, and provide easy access to elements by index.

#### Disadvantages

Vectors lose strength when the size of the dataset is dynamic and must be changed throughout the life of an application. Resizing operations become costly when the vector’s capacity is exceeded. Insertions also require the shifting of elements, which can quickly become costly or introduce undefined behavior. It is not quite powerful for dynamic applications which handle insertions and deletions, along with sorting.

## Hash Table

#### Advantages

Hash tables are more efficient than vectors for large datasets with dynamic operations. With its usage of key-value pairs and key generation for each object within a node, it provides flexibility.

#### Disadvantages

Robust hash functions are required to minimize collisions. Hash tables also introduce memory overhead due to potentially empty slots in the hash table. Sorting logic is not innate to the data structure and must be implemented separately.

## BST

#### Advantages

Elements of a BST are innately sorted and ordered, allowing ordered traversal to be maximally efficient. For balanced trees, update and search operations have more optimal complexities. They are best suited for applications requiring ordered traversals, especially when performed often.

#### Disadvantages

If a tree becomes unbalanced, performance degrades, and complexity rises. Additional logic for tree balancing is required for optimizing performance with dynamic datasets. Hash tables beat BST in space efficiency due to node overhead.

# Recommendation

I plan to use the binary search tree in my final project’s code to meet the academic advisors’ requirements. It is the best data structure for ease of use for the end user. In my analysis, I found that each data structure has a worst-case runtime complexity of O(n). Given the same worst runtime complexity, the BST is still the best fit. The sorting and printing methods can be combined into one method: the InOrder method. A binary search tree’s structure intuitively allows in order traversal, meeting the requirements of alphabetical order lowest-to-highest sorting and printing of the courses as they are held in the tree’s nodes. This is because the insertion of a node handles ordering, allowing the tree to stay ordered upon insertions and deletions. In contrast, a hash table does not preserve the order of insertion, causing data to be unexpectedly ordered within the table. A vector is not appropriate for the dynamic dataset of the advisors’ course information, as advisors will need to update courses in the program, and the size is not predetermined. Given these considerations, I will be implementing a binary search tree for the advisors at ABCU.