Project One: Pseudocode and Runtime Analysis

Name: Matt

Date: 10/11/2025

Contents

1 Pseudocode

I went through each of the previous assignments to clean up the pseudocode and improve it where I could. This included reorganizing it and adding a bit more structure and clarity, along with rebuilding certain functions such as loadCourses.

To meet the requirements, I added a basic bubble sorting algorithm for the alphanumeric ordered printAll, which is $O(n^2)$ for both worst and average cases. In a full program implementation, merge sort would be the better choice since it offers a consistent $O(n \log n)$.

1.1 Vector

```
/*********************************
* VECTOR DATA STRUCTURE IMPLEMENTATION
**********************************/

/*********************************
* Structure for course data
**********************************/
struct Course {
    string courseNumber // unique identifier
    string name // course title/name
    vector<string> prerequisites // list of prerequisite course numbers
}

/*********************************
* Helper functions
**********************************/
vector splitCSV(string line) {
    split line on commas into parts
    for each part
        part = trim(part)
    return vector of trimmed parts
}

string trim(string text) {
    remove leading spaces
    remove trailing spaces
    return trimmed text
}

bool containsDigit(string course) {
    for each character in course
        if character is digit ('0' through '9')
            return true
    return false
}

// Bubble sort for course ordering
void sortCourses(vector<Course>& courses) {
    for i from 0 to courses.size - 1
        for j from 0 to courses.size - i - 2
            if courses[j].courseNumber > courses[j+1].courseNumber
                swap courses[j] and courses[j+1]
}

/*********************************
* Load courses from file
*********************************/
bool loadCourses(string filePath, vector<Course>& courses) {
    set<string> courseNumbers // track valid courses
    integer lineNumber = 0

    open filePath for reading
    if open fails
        print out error + filePath
        return false
    print out loading file path + filePath

    // start clean after file opens
    courses.clear()

    // read and parse lines and build course objects
    for each line in file
        lineNumber = lineNumber + 1
        if line is empty
            continue to next line

        vector<string> fields = splitCSV(line)

        // validate at least 2 parameters (courseNumber and name)
        if fields.size < 2
            print "Error line " + lineNumber + ": missing courseNumber or name"
            close file
            courses.clear()
            return false

        // validate fields not empty after trim
        if fields[0] == "" or fields[1] == ""
```

```
            error due to empty courseNumber or name
            close file
            courses.clear()
            return false

        // building course object from fields
        define new Course course object
        course.courseNumber = fields[0]
        course.name = fields[1]

        // add prerequisites if they exist
        for i from 2 to fields.size - 1
            prereq = fields[i]
            if prereq == ""
                print "warning line " + lineNumber + ": empty prerequisite ignored"
                continue
            // check length and if prerequisite contains at least one digit
            if length(prereq) < 4 OR not containsDigit(prereq)
                print "error line " + lineNumber
                + ": malformed prerequisite " + prereq
                close file
                courses.clear()
                return false
            course.prerequisites.push_back(prereq)

        // duplicate check for courses
        if course.courseNumber is in courseNumbers
            print error duplicate course + lineNumber + course.courseNumber
            close file
            courses.clear()
            return false

        // record that it's been seen
        add course.courseNumber into courseNumbers set

        // store the course object
        courses.push_back(course)

    close file

    // validation loop: prerequisites exist as courses
    for each course in courses
        for each prereq in course.prerequisites
            if prereq is not in courseNumbers set
```

```
                print "Error: unknown prerequisite '" + prereq
                courses.clear()
                return false

    print "Successfully loaded " + courses.size + " courses"
    return true
}

/*********************************
* Search for specific course
**********************************/
void searchCourse(vector<Course>& courses, string courseNumber) {
    // Linear search through vector
    for each course in courses
        if course.courseNumber == courseNumber
            print course.courseNumber + ", " + course.name
            if course.prerequisites.size > 0
                print "Prerequisites:"
                for each prereq in course.prerequisites
                    print " " + prereq
            else
                print "No prerequisites"
            return
    // not found
    print "Course " + courseNumber + " not found."
}

/*********************************
* Print all courses (sorted)
**********************************/
void printAll(vector<Course>& courses) {
    print all courses header

    // Create a copy for sorting
    vector<Course> sortedCourses = courses

    sortCourses(sortedCourses)

    // Print sorted courses
    for each course in sortedCourses
        print course.courseNumber + ", " + course.name
}

/*******************************
```

```
* Main entry
********************************/
main() {
    string csvPath, courseNumber
    clock_t ticks // timer variable

    vector<Course> courses

    integer choice = 0
    // main loop until user exits
    while choice is not 9
        print display menu
        case 1: Load Data Structure
        case 2: Print Course List
        case 3: Search and Print Course
        case 9: Exit
        Enter a choice

        get user choice from input

        switch (choice)
            case 1:
                // load file data
                print "Enter CSV file path or press Enter for default: "
                get input for csvPath
                if csvPath is empty
                    print "File path incorrect or empty, defaulting to coursefile.csv"
                    csvPath = "coursefile.csv"

                // initialize timer variable before loading
                ticks = clock()

                // load courses
                loaded = loadCourses(csvPath, courses)
                if not loaded
                    print "Failed to load: " + csvPath + ", trying default."
                    loaded = loadCourses("coursefile.csv", courses)

                if not loaded
                    print "Failed to load courses with default: coursefile.csv"
                else
                    print "Data structure loaded."
                    // calculate elapsed time and display results
                    ticks = clock() - ticks
                    print "time: " + ticks + " clock ticks."
                    print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds."
                break

            case 2:
                printAll(courses)
                break

            case 3:
                print "Input course number to search: "
                get courseNumber from user input
                if courseNumber.empty()
                    print "Invalid input, try again"
                else
                    ticks = clock()
                    searchCourse(courses, courseNumber)
                    ticks = clock() - ticks
                    print "time: " + ticks + " clock ticks."
                    print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds."
                break

            case 9:
                break

            default:
                print invalid choice, try again

    print Good Bye
    return
}
```

1.2 Hash Table

For the hash algorithm, the structure was updated to include collision handling and dynamic resizing.

```
/*********************************
* HASH TABLE DATA STRUCTURE IMPLEMENTATION
*********************************/

/*********************************
* Helper functions
* unnamed namespace, static, or forward declarations and move them down.
*********************************/
vector splitCSV(string line) {
    split line on commas into parts
    for each part
        part = trim(part)
    return vector of trimmed parts
}

string trim(string text) {
    remove leading spaces
    remove trailing spaces
    return trimmed text
}

bool containsDigit(string course) {
    for each character in course
        if character is digit ('0' through '9')
            return true
    return false
}

bool isPrime(unsigned int n) {
    if n <= 1 return false
    if n <= 3 return true
    if n % 2 == 0 or n % 3 == 0 return false
    for i = 5 to i*i <= n step 6
        if n % i == 0 or n % (i + 2) == 0 return false
    return true
}

unsigned int nextPrime(unsigned int n) {
    if n <= 2 return 2
    if n % 2 == 0 n = n + 1
    while not isPrime(n)
```

```
        n = n + 2
    return n
}

/********************************
* Structure for course data
*********************************/
struct Course {
    string courseNumber // unique identifier (key for hashing)
    string name // course title/name
    vector<string> prerequisites // list of prerequisite course numbers
}

/********************************
* Structure for node chaining
*********************************/
struct Node {
    Course course // the course data
    unsigned int key // hash key value
    Node* next // pointer to next node in chain

    // default constructor
    Node() {
        key = UINT_MAX // empty bucket marker
        next = nullptr
    }
}

/********************************
* Hash Table Class definition
*********************************/
class CourseHashTable {
    vector<Node> buckets // vector of buckets (nodes)
    unsigned int tableSize = 31 // initial prime number for small dataset
    unsigned int numElements = 0 // track total elements
    unsigned int maxChainLength = 5 // threshold for resizing

    // private methods
    unsigned int hash(string courseNumber)
    void resize() // dynamic resizing when chains get too long

    // public methods
    void Insert(Course course)
    void searchCourse(string courseNumber)
    void printAll()
    void clear()
}

/********************************
* Constructor and Destructor
*********************************/
CourseHashTable::CourseHashTable() {
    buckets.resize(tableSize)
    // all Node objects created with default constructor
}

CourseHashTable::~CourseHashTable() {
    for i from 0 to tableSize - 1
        current = buckets[i].next
        while current != nullptr
            temp = current
            current = current->next
            delete temp
}

/********************************
* Hash function
*********************************/
unsigned int CourseHashTable::hash(string courseNumber) {
    digitString = ""
    for each character in courseNumber
        if character is digit
            digitString = digitString + character

    if digitString is not empty
        return atoi(digitString) % tableSize
    else
        // fallback: use first character code
        return courseNumber[0] % tableSize
}

/********************************
* Dynamic resize function
*********************************/
void CourseHashTable::resize() {
    print ”Resizing hash table from ” + tableSize + ” to ”

    // save old buckets
    vector<Node> oldBuckets = buckets
    unsigned int oldSize = tableSize

    // double size and move to next prime
    tableSize = nextPrime(tableSize * 2)
    print tableSize + " buckets."

    // create new bucket array
    buckets.clear()
    buckets.resize(tableSize)
    numElements = 0

    // rehash all elements from old buckets
    for i from 0 to oldSize - 1
        if oldBuckets[i].key != UINT_MAX
            // rehash main node
            Insert(oldBuckets[i].course)

            // rehash chained nodes
            current = oldBuckets[i].next
            while current != nullptr
                Insert(current->course)
                temp = current
                current = current->next
                delete temp
}

/********************************
* Insert course into hash table
********************************/
void CourseHashTable::Insert(Course course) {
        // convert courseNumber to key and hash it
    key = hash(course.courseNumber)
    // retrieve node/bucket using hash key
    node = &buckets.at(key)

    // if the head bucket/node is empty
    if node->key == UINT_MAX
        node->key = key
        node->course = course
        node->next = nullptr
        numElements++
        return

    // update existing course
    if node->course.courseNumber == course.courseNumber
        node->course = course
        return

    // traverse chain to find course or end
    chainLength = 1
    while node->next != nullptr
        node = node->next
        chainLength = chainLength + 1
        if node->course.courseNumber == course.courseNumber
            node->course = course
            return

    // add new node at end of chain
    node->next = new Node()
    node->next->key = key
    node->next->course = course
    numElements++

    // check if resize needed due to long chains
    // chain length after append is chainLength + 1 from original head
    if (chainLength + 1) > maxChainLength
        print "Chain length " + (chainLength + 1) + " exceeds threshold"
        resize()
}

/*********************************
* Search for specific course (class method)
*********************************/
void CourseHashTable::searchCourse(string courseNumber) {
    key = hash(courseNumber)
    node = &buckets.at(key)

    // check if bucket is empty
    if node->key == UINT_MAX
        print "Course '" + courseNumber + "' not found."
        return

    // search through chain
    while node != nullptr
        if node->course.courseNumber == courseNumber
            print node->course.courseNumber + ", " + node->course.name
            if node->course.prerequisites.size > 0
```

```
                print "Prerequisites:"
                for each prereq in node->course.prerequisites
                    print " " + prereq
            else
                print "No prerequisites"
            return
        node = node->next

    // not found
    print "Course '" + courseNumber + "' not found."
}

/*********************************
* Print all courses (sorted by bubble for now -- class method)
*********************************/
void CourseHashTable::printAll() {
    print all courses header

    // collect all courses for sorting
    vector<Course> allCourses

    // iterate through all buckets
    for i from 0 to tableSize - 1
        if buckets[i].key != UINT_MAX
            // add main node course
            allCourses.push_back(buckets[i].course)

            // add chained nodes
            node = buckets[i].next
            while node != nullptr
                allCourses.push_back(node->course)
                node = node->next

    // sort courses by courseNumber
    for i from 0 to allCourses.size - 1
        for j from 0 to allCourses.size - i - 2
            if allCourses[j].courseNumber > allCourses[j+1].courseNumber
                swap allCourses[j] and allCourses[j+1]

    // print sorted courses
    for each course in allCourses
        print course.courseNumber + ", " + course.name
}
/*********************************
```

```
* clear is a helper for loading data
*********************************/
void CourseHashTable::clear() {
    for i from 0 to tableSize - 1
        current = buckets[i].next
        while current != nullptr
            temp = current
            current = current->next
            delete temp
        buckets[i].key = UINT_MAX
        buckets[i].next = nullptr
    numElements = 0
}


/*********************************
* Load courses from file
*********************************/
bool loadCourses(string filePath, CourseHashTable* ht) {
    set<string> courseNumbers // track valid courses
    vector<Course> parsed // extra vector of what's inserted for validation
    integer lineNumber = 0

    open filePath for reading
    if open fails
        print ”Error loading ” + filePath
        return false
    print ”Loading file path ” + filePath

    //starts clean
    ht->clear()

    // read and parse lines and build course objects
    for each line in file
        lineNumber = lineNumber + 1
        if line is empty
            continue to next line

        vector<string> fields = splitCSV(line)

        // validate at least 2 parameters (courseNumber and name)
        if fields.size < 2
            print ”Error line ” + lineNumber + ”: missing courseNumber or name”
            close file
            ht->clear()
            return false

        // validate fields not empty after trim
        if fields[0] == "" or fields[1] == ""
            print "Error line " + lineNumber + ": empty courseNumber or name"
            close file
            ht->clear()
            return false

        // building course object from fields
        define new Course course object
        course.courseNumber = fields[0]
        course.name = fields[1]

        // add prerequisites if they exist
        for i from 2 to fields.size - 1
            prereq = fields[i]
            if prereq == ""
                print "Warning line " + lineNumber + ": empty prerequisite ignored"
                continue
            // check length and if prerequisite contains at least one digit
            if length(prereq) < 4 OR not containsDigit(prereq)
                print "Error line " + lineNumber
                + ": malformed prerequisite " + prereq
                close file
                ht->clear()
                return false
            course.prerequisites.push_back(prereq)

        // duplicate check for courses
        if course.courseNumber is in courseNumbers
            print "Error line " + lineNumber
            + ": duplicate course " + course.courseNumber
            close file
            ht->clear()
            return false

        // record that it's been seen
        add course.courseNumber into courseNumbers set

        // insert into hash table
        ht->Insert(course)
        parsed.push_back(course)
```

```

    close file

    // validation loop for each prerequisite must exist in courseNumbers
    for each course in parsed
        for each prereq in course.prerequisites
            if prereq is not in courseNumbers
                    print ”Error: unknown prerequisite ” + prereq
                    ht->clear()
                    return false

    print ”Successfully loaded ” + courseNumbers.size + ” courses.”
    return true
}

/*******************************
* Main entry
********************************/
main() {
    string csvPath, courseNumber
    clock_t ticks // timer variable

    CourseHashTable* courseTable = new CourseHashTable()

    integer choice = 0
    // main loop until user exits
    while choice is not 9
        print display menu
        case 1: Load Data Structure
        case 2: Print Course List
        case 3: Search and Print Course
        case 9: Exit
        Enter a choice

        get user choice from input

        switch (choice)
            case 1:
                // load file data
                print ”Enter CSV file path or press Enter for default: ”
                get input for csvPath
                if csvPath is empty
                    print ”File path incorrect or empty, defaulting to coursefile.csv”
                    csvPath = ”coursefile.csv”

                // initialize timer variable before loading
                ticks = clock()

                // load courses
                loaded = loadCourses(csvPath, courseTable)
                if not loaded
                                    print "Failed to load: " + csvPath + ", trying default"
                                    loaded = loadCourses("coursefile.csv", courseTable)

                                if not loaded
                                    print "Failed to load courses."
                                else
                                    print "Data structure loaded."
                                    ticks = clock() - ticks
                                        print "time: " + ticks + " clock ticks."
                                    print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds."
                break

            case 2:
                courseTable->printAll()
                break

            case 3:
                print "Input course number to search: "
                get courseNumber from user input
                if courseNumber is empty
                    print "Invalid input, try again"
                else
                    ticks = clock()
                    courseTable->searchCourse(courseNumber)
                    ticks = clock() - ticks
                    print "time: " + ticks + " clock ticks."
                    print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds."
                break

            case 9:
                break

            default:
                print invalid choice, try again

    print Good Bye
```

```

    // clean up
    delete courseTable
    return
}
```

1.3 Binary Search Tree

```
/*********************************
* BINARY SEARCH TREE
* DATA STRUCTURE IMPLEMENTATION
**********************************/

/*********************************
* Helper functions:
* Split the CSV line at commas into parts.
**********************************/
vector splitCSV(string line) {
        split line on commas into parts
        for each part
                part = trim(part) // remove leading/trailing spaces
        return vector of trimmed parts
}

/*********************************
* Trim whitespace:
**********************************/
string trim(string text) {
        remove leading spaces
        remove trailing spaces
        return trimmed text
}

/*********************************
* Check if string contains digit
**********************************/
bool containsDigit(string course) {
    for each character in course
        if character is digit ('0' through '9')
            return true
    return false
}

/*********************************
```

```
* Structure for course data
*********************************/
struct Course {
        string courseNumber // unique identifier
        string name // course title/name
        vector<string> prerequisites // list of prerequisite course numbers
}

/**********************************
Structure Node for binary search tree
***********************************/
struct Node {
    Course course       // the course data stored in this Node
    Node* left          // pointer to the left child
    Node* right         // pointer to the right child

    // default constructor with course parameter
    // When I create a new Node, I pass a course object to it.
    // Useful for inserting courses
    Node(Course c) {
        course = c // copies the passed Course into this node
        left = nullptr
        right = nullptr
    }
}

/**********************************
* Binary Search Tree Class definition
***********************************/

class BinarySearchTree {
    Node* root // creating a node pointer for the root of the tree

    //public methods
    void Insert(Course course)
    void searchCourse(string courseNumber)
    void printAll()
    void clear()

    //private helper methods
    Node* findNode(string courseNumber) // helper to find a node for search
    void deleteTree(Node* node) // needed for destructor
    void inOrder(Node* node) // recursive traversal for printing
}
```

```
/*********************************
* Constructor
**********************************/
BinarySearchTree::BinarySearchTree() {
    root = nullptr/empty
}

/*********************************
* Destructor
**********************************/
BinarySearchTree::~BinarySearchTree() {
    deleteTree(root)
}

/*********************************
* Helper method for destructor
**********************************/
void BinarySearchTree::deleteTree(Node* node) {
   if node == nullptr/empty
      return
   // post-order recursive deletion of entire tree to free memory
   deleteTree(node->left)
   deleteTree(node->right)
   delete node
}

/*********************************
* Insert course into BST
**********************************/
void BinarySearchTree::Insert(Course course) {
    // if tree is empty, create root
    if root == nullptr/empty
       root = new Node(course)
       return

    // start at root to find insertion point
    current = root
    parent = nullptr

    // traverse tree to find where to insert
    while current != nullptr (while it isn't empty)
       parent = current

       // if course already exists, update it
       if current->course.courseNumber == course.courseNumber
          current->course = course
          return

       // go left or right based on course number
       // left if smaller, else right if bigger.
       if course.courseNumber < current->course.courseNumber
          current = current->left
       else
          current = current->right

   // create new node and attach to parent
   newNode = new Node(course)
   if course.courseNumber < parent->course.courseNumber
      parent->left = newNode
   else
      parent->right = newNode
}

/***********************************
* Search for course (find the data)
***********************************/
void BinarySearchTree::searchCourse(string courseNumber) {
   current = root

   // traverse tree until found
   while current != nullptr // while not empty
      if current->course.courseNumber == courseNumber
         print current->course.courseNumber + ”, ” + current->course.name
         if current->course.prerequisites.size > 0
            print ”Prerequisites:”
            for each prereq in current->course.prerequisites
               print ” ” + prereq
         else
            print ”No prerequisites”
         return
      if courseNumber < current->course.courseNumber
         current = current->left
      else
         current = current->right
   // not found
   print ”Course '” + courseNumber + ”' not found.”
}
```

```

/*****************************************
* Private helper to find a node for search
*****************************************/
Node* BinarySearchTree::findNode(string courseNumber) {
   current = root

   while current != nullptr // while not empty
      if current->course.courseNumber == courseNumber
         return current
      if courseNumber < current->course.courseNumber
         current = current->left
      else
         current = current->right

   return nullptr
}

/**********************************
* Print all courses (in-order traversal)
**********************************/
void BinarySearchTree::printAll() {
    print all courses header
    inOrder(root)
}

/**********************************
* Private method Print helper for PrintAll to avoid infinite recursion
**********************************/
void BinarySearchTree::inOrder(Node* node) {
   if node == nullptr/empty
      return
   // traverse left subtree
   inOrder(node->left)
   // print current node course info
   print node->course.courseNumber + ", " + node->course.name
   // traverse right subtree
   inOrder(node->right)
}

/**********************************
* clear is a helper for loadCourses
**********************************/
void BinarySearchTree::clear() {
    deleteTree(root)
    root = nullptr/empty
}
/**********************************
* load courses from file
* Re-wrote this to now return true/false instead of a pointer.
**********************************/
bool loadCourses(string filePath, BinarySearchTree* bst) {
    set<string> courseNumbers  // track valid courses
    integer lineNumber = 0

    open filePath for reading
    if open fails
        print out error + filePath
        return false
    print out loading file path + filePath

    bst->clear() // clear for new load

    // Read and parse lines
    for each line in file
        lineNumber = lineNumber + 1
        if line is empty
            continue to next line

        vector<string> fields = splitCSV(line)

        // validate at least 2 parameters (courseNumber and name)
        if fields.size < 2
            print format error due to courseNumber and or name size missing field
            close file
            bst->clear()
            return false

        // validate fields not empty after trim
        if fields[0] == "" or fields[1] == ""
            error due to empty courseNumber or name
            close file
            bst->clear()
            return false

        // building course object from fields
        define new Course course object
        course.courseNumber = fields[0]
        course.name = fields[1]

        // add prerequisites if they exist
        for i from 2 to fields.size - 1
            prereq = fields[i]
            if prereq == ""
                print "warn line " + lineNumber + ": empty prerequisite ignored"
                continue
            // check length and if prerequisite contains at least one digit
            if length(prereq) < 4 OR not containsDigit(prereq)
                print "error line " + lineNumber
                + ": malformed prerequisite " + prereq
                close file
                bst->clear()
                return false
            course.prerequisites.push_back(prereq)

        // duplicate check for courses
        if course.courseNumber is in courseNumbers
            print error duplicate course + lineNumber + course.courseNumber
            close file
            bst->clear()
            return false

        // record that it's been seen
        add course.courseNumber into courseNumbers set
        // insert into bst
        bst->Insert(course)

    close file

    // validation loop: prerequisites exist as courses
    // need to validate all courses in tree using courseNumbers set
    for each courseNumber in courseNumbers set
        node = bst->findNode(courseNumber)
        if node != nullptr
            for each prereq in node->course.prerequisites
                if prereq is not in courseNumbers set
                    print "Error: unknown prerequisite " + prereq
                    bst->clear()
                    return false

    print Successfully loaded + courseNumbers.size + courses
    return true
```

```
}

/******************************
* main entry
*******************************/
main() {

    string csvPath, courseNumber
    // timer variable
    clock_t ticks

        BinarySearchTree* bst = new BinarySearchTree()

    integer choice = 0
    // main loop until user exits
    while choice is not 9
        print display menu
        case 1: Load Data Structure
        case 2: Print Course List
        case 3: Search and Print Course
        case 9: Exit
        Enter a choice

        get user choice from input

        switch (choice)
            case 1:
                    // load file data
                    print "Enter CSV file path or press Enter for default: "
                    get input for csvPath
                    if csvPath is empty
                                print "File path incorrect or empty, defaulting to coursefile.csv"
                                csvPath = "coursefile.csv"

                    // initialize timer variable before loading bids
                            ticks = clock()
                    loaded = loadCourses(csvPath, bst)
                    if not loaded
                            print "Failed to load: " + csvPath + ", trying default."
                            loaded = loadCourses("coursefile.csv", bst)
                        if not loaded
                                print "Failed to load courses."
                            else
```

```
                                print "Data structure loaded."
                                ticks = clock() - ticks
                    print "time: " + ticks + " clock ticks."
                    print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds."
                break
            case 2:
                bst->printAll()
                break
            case 3:
                print "Input course number to search: "
                get courseNumber from user input
                if courseNumber.empty()
                    print "Invalid input, try again"
                else
                    ticks = clock()
                    bst->searchCourse(courseNumber)
                    ticks = clock() - ticks
                    print "time: " + ticks + " clock ticks."
                    print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds."
                break
            case 9:
                break
            default:
                print invalid choice, try again

    print Good Bye

    // clean up
    delete bst
    return
}
```

2 Runtime Analysis

General terminology:

- n = number of courses
- m = average line length
- p = average prerequisites per course

2.1 Vector

Operations in build order:

- Read and parse file: $O(n \cdot m)$
- Duplicate check via set: $O(n \log n)$
- Insert into vector via `push_back`: $O(n)$
- Prerequisite validation via set: $O(n \cdot p \cdot \log n)$
- **Total build:** $O(n \cdot m + n \log n + n \cdot p \cdot \log n)$; if m, p are small constants, this simplifies to $O(n \log n)$

After build:

- Option 3 (search one course and print its prerequisites): $O(n + p)$, reduced to $O(n)$
- Option 2 (print all sorted): $O(n \log n)$ per call.

Additionally, I could add an index for the vector such as an unordered map at load, which can make the specific course look up become $O(1 + p)$ average.

After considering some of the strengths, a vector would be the simplest implementation and easy to debug. It has predictable $O(n)$ space with no overhead along with stable performance since there are no worst-case surprises like an unbalanced binary tree. It can handle size changes easily and works well with smaller data sets. The problems start when the course list grows too big. $O(n)$ search would become too slow for frequent course lookups. As mentioned above, an unordered map can make this $O(1)$.

2.2 Hash Table

Operations in build order:

- Read and parse: $O(n \cdot m)$
- Duplicate check via set: $O(n \log n)$

- Insert into hash table (average, resizes amortized): $O(n)$
- Prerequisite validation via set: $O(n \cdot p \cdot \log n)$
- **Total build:** $O(n \cdot m + n \log n + n \cdot p \cdot \log n)$; if m, p are small constants, this simplifies to $O(n \log n)$

After build:

- Search one course: $O(1)$ average, $O(n)$ worst.
- Print all sorted: gather $O(n)$ then sort $O(n \log n)$.

The $O(1)$ average search, insertion, and deletion is great for the intended utility. It handles frequent look ups well. It has dynamic resizing to maintain performance and can scale well with larger course databases. It suits the project use case. A weakness is the additional memory overhead from the load factor, along with a worst-case of $O(n)$ if the hash function fails.

2.3 Binary Search Tree

Operations in build order:

- Read and parse: $O(n \cdot m)$
- Duplicate handling: via set $O(n \log n)$
- Insert (unbalanced): average $O(n \log n)$, worst $O(n^2)$
- Prerequisite validation via set: $O(n \cdot p \cdot \log n)$
- **Total build (avg):** $O(n \cdot m + n \log n + n \cdot p \cdot \log n)$; if m, p are small constants, this simplifies to $O(n \log n)$ **worst:** $O(n^2 + n \cdot p \cdot n)$ or as above, simplified to $O(n^2)$.

After build:

- Print all (automatic in-order traversal): $O(n)$
- Search one course: $O(\log n)$ average, $O(n)$ worst

2.4 Comparison and Final Choice

The runtime and build analysis is roughly the same across these data structures due to the way I implemented it, besides BST having a possibly worse worst case of $O(n^2)$. It doesn't have a major effect on the data structure chosen since the intended functionality isn't reloading and changing the loaded data often.

Print all (sorted alphanumeric):

- Vector: $O(n \log n)$ per call
- Hash: gather $O(n)$ then sort $O(n \log n)$
- BST: $O(n)$ via in-order traversal. The clear winner here.

Search one course:

- Vector: $O(n)$ via linear search. $O(1)$ average when using an unordered map index, though it can degrade to $O(n)$.
- Hash: $O(1)$ average, winner here.
- BST: in the middle with $O(\log n)$ average, $O(n)$ worst.

The real world application of loading a university course catalogue will have hundreds of courses, this represents a small dataset. Search and print to check classes and prerequisites seems like it'd be the most utilized functionality, meaning its speed matters most. Print sorted list in comparison wouldn't be used as much, maybe to review a curriculum. And the dataset would also be something that isn't updated frequently since new courses are not added often. With all of these considerations, I think the **hash table** is the ideal option, tied with a modified vector.