

# Project One: Pseudocode and Runtime Analysis

Name: Matt

Date: 10/11/2025

## Contents

<b>1</b>	<b>Pseudocode</b>	<b>2</b>
1.1	Vector . . . . .	2
1.2	Hash Table . . . . .	8
1.3	Binary Search Tree . . . . .	18
<b>2</b>	<b>Runtime Analysis</b>	<b>27</b>
2.1	Vector . . . . .	27
2.2	Hash Table . . . . .	27
2.3	Binary Search Tree . . . . .	28
2.4	Comparison and Final Choice . . . . .	28

# 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

```

1  /*****
2  * VECTOR DATA STRUCTURE IMPLEMENTATION
3  *****/
4
5  /*****
6  * Structure for course data
7  *****/
8  struct Course {
9      string courseNumber // unique identifier
10     string name // course title/name
11     vector<string> prerequisites // list of prerequisite course numbers
12 }
13
14 /*****
15 * Helper functions
16 *****/
17 vector splitCSV(string line) {
18     split line on commas into parts
19     for each part
20         part = trim(part)
21     return vector of trimmed parts
22 }
23
24 string trim(string text) {
25     remove leading spaces
26     remove trailing spaces
27     return trimmed text
28 }
29
30 bool containsDigit(string course) {
31     for each character in course
32         if character is digit ('0' through '9')
33         return true

```

```
34     return false
35 }
36
37 // Bubble sort for course ordering
38 void sortCourses(vector<Course>& courses) {
39     for i from 0 to courses.size - 1
40         for j from 0 to courses.size - i - 2
41             if courses[j].courseNumber > courses[j+1].courseNumber
42                 swap courses[j] and courses[j+1]
43 }
44
45 /*****
46 * Load courses from file
47 *****/
48 bool loadCourses(string filePath, vector<Course>& courses) {
49     set<string> courseNumbers // track valid courses
50     integer lineNumber = 0
51
52     open filePath for reading
53     if open fails
54         print out error + filePath
55         return false
56     print out loading file path + filePath
57
58     // start clean after file opens
59     courses.clear()
60
61     // read and parse lines and build course objects
62     for each line in file
63         lineNumber = lineNumber + 1
64         if line is empty
65             continue to next line
66
67         vector<string> fields = splitCSV(line)
68
69         // validate at least 2 parameters (courseNumber and name)
70         if fields.size < 2
71             print "Error line " + lineNumber + ": missing courseNumber or name"
72             close file
73             courses.clear()
74             return false
75
76         // validate fields not empty after trim
77         if fields[0] == "" or fields[1] == ""
```

```
78         error due to empty courseNumber or name
79         close file
80         courses.clear()
81         return false
82
83     // building course object from fields
84     define new Course course object
85     course.courseNumber = fields[0]
86     course.name = fields[1]
87
88     // add prerequisites if they exist
89     for i from 2 to fields.size - 1
90         prereq = fields[i]
91         if prereq == ""
92             print "warning line " + lineNumber + ": empty prerequisite ignored"
93             continue
94         // check length and if prerequisite contains at least one digit
95         if length(prereq) < 4 OR not containsDigit(prereq)
96             print "error line " + lineNumber
97             + ": malformed prerequisite " + prereq
98             close file
99             courses.clear()
100             return false
101             course.prerequisites.push_back(prereq)
102
103     // duplicate check for courses
104     if course.courseNumber is in courseNumbers
105         print error duplicate course + lineNumber + course.courseNumber
106         close file
107         courses.clear()
108         return false
109
110     // record that it's been seen
111     add course.courseNumber into courseNumbers set
112
113     // store the course object
114     courses.push_back(course)
115
116 close file
117
118 // validation loop: prerequisites exist as courses
119 for each course in courses
120     for each prereq in course.prerequisites
121         if prereq is not in courseNumbers set
```

```

122         print "Error: unknown prerequisite '" + prereq
123         courses.clear()
124         return false
125
126     print "Successfully loaded " + courses.size + " courses"
127     return true
128 }
129
130 /*****
131 * Search for specific course
132 *****/
133 void searchCourse(vector<Course>& courses, string courseNumber) {
134     // Linear search through vector
135     for each course in courses
136         if course.courseNumber == courseNumber
137             print course.courseNumber + ", " + course.name
138             if course.prerequisites.size > 0
139                 print "Prerequisites:"
140                 for each prereq in course.prerequisites
141                     print " " + prereq
142             else
143                 print "No prerequisites"
144             return
145     // not found
146     print "Course " + courseNumber + " not found."
147 }
148
149 /*****
150 * Print all courses (sorted)
151 *****/
152 void printAll(vector<Course>& courses) {
153     print all courses header
154
155     // Create a copy for sorting
156     vector<Course> sortedCourses = courses
157
158     sortCourses(sortedCourses)
159
160     // Print sorted courses
161     for each course in sortedCourses
162         print course.courseNumber + ", " + course.name
163 }
164
165 /*****

```

```
166 * Main entry
167 *****/
168 main() {
169     string csvPath, courseNumber
170     clock_t ticks // timer variable
171
172     vector<Course> courses
173
174     integer choice = 0
175     // main loop until user exits
176     while choice is not 9
177         print display menu
178         case 1: Load Data Structure
179         case 2: Print Course List
180         case 3: Search and Print Course
181         case 9: Exit
182         Enter a choice
183
184         get user choice from input
185
186         switch (choice)
187             case 1:
188                 // load file data
189                 print "Enter CSV file path or press Enter for default: "
190                 get input for csvPath
191                 if csvPath is empty
192                     print "File path incorrect or empty, defaulting to coursefile.csv"
193                     csvPath = "coursefile.csv"
194
195                 // initialize timer variable before loading
196                 ticks = clock()
197
198                 // load courses
199                 loaded = loadCourses(csvPath, courses)
200                 if not loaded
201                     print "Failed to load: " + csvPath + ", trying default."
202                     loaded = loadCourses("coursefile.csv", courses)
203
204                 if not loaded
205                     print "Failed to load courses with default: coursefile.csv"
206                 else
207                     print "Data structure loaded."
208                     // calculate elapsed time and display results
209                     ticks = clock() - ticks
```

```
210         print "time: " + ticks + " clock ticks."
211         print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds."
212     break
213
214     case 2:
215         printAll(courses)
216         break
217
218     case 3:
219         print "Input course number to search: "
220         get courseNumber from user input
221         if courseNumber.empty()
222             print "Invalid input, try again"
223         else
224             ticks = clock()
225             searchCourse(courses, courseNumber)
226             ticks = clock() - ticks
227             print "time: " + ticks + " clock ticks."
228             print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds."
229         break
230
231     case 9:
232         break
233
234     default:
235         print invalid choice, try again
236
237 print Good Bye
238 return
239 }
```

## 1.2 Hash Table

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

```

1  /*****
2  * HASH TABLE DATA STRUCTURE IMPLEMENTATION
3  *****/
4
5  /*****
6  * Helper functions
7  * unnamed namespace, static, or forward declarations and move them down.
8  *****/
9  vector splitCSV(string line) {
10     split line on commas into parts
11     for each part
12         part = trim(part)
13     return vector of trimmed parts
14 }
15
16 string trim(string text) {
17     remove leading spaces
18     remove trailing spaces
19     return trimmed text
20 }
21
22 bool containsDigit(string course) {
23     for each character in course
24         if character is digit ('0' through '9')
25             return true
26     return false
27 }
28
29 bool isPrime(unsigned int n) {
30     if n <= 1 return false
31     if n <= 3 return true
32     if n % 2 == 0 or n % 3 == 0 return false
33     for i = 5 to i*i <= n step 6
34         if n % i == 0 or n % (i + 2) == 0 return false
35     return true
36 }
37
38 unsigned int nextPrime(unsigned int n) {
39     if n <= 2 return 2
40     if n % 2 == 0 n = n + 1
41     while not isPrime(n)

```



```

42     n = n + 2
43     return n
44 }
45
46 /*****
47 * Structure for course data
48 *****/
49 struct Course {
50     string courseNumber // unique identifier (key for hashing)
51     string name // course title/name
52     vector<string> prerequisites // list of prerequisite course numbers
53 }
54
55 /*****
56 * Structure for node chaining
57 *****/
58 struct Node {
59     Course course // the course data
60     unsigned int key // hash key value
61     Node* next // pointer to next node in chain
62
63     // default constructor
64     Node() {
65         key = UINT_MAX // empty bucket marker
66         next = nullptr
67     }
68 }
69
70 /*****
71 * Hash Table Class definition
72 *****/
73 class CourseHashTable {
74     vector<Node> buckets // vector of buckets (nodes)
75     unsigned int tableSize = 31 // initial prime number for small dataset
76     unsigned int numElements = 0 // track total elements
77     unsigned int maxChainLength = 5 // threshold for resizing
78
79     // private methods
80     unsigned int hash(string courseNumber)
81     void resize() // dynamic resizing when chains get too long
82
83     // public methods
84     void Insert(Course course)
85     void searchCourse(string courseNumber)

```

```

86     void printAll()
87     void clear()
88 }
89
90 /*****
91 * Constructor and Destructor
92 *****/
93 CourseHashTable::CourseHashTable() {
94     buckets.resize(tableSize)
95     // all Node objects created with default constructor
96 }
97
98 CourseHashTable::~~CourseHashTable() {
99     for i from 0 to tableSize - 1
100         current = buckets[i].next
101         while current != nullptr
102             temp = current
103             current = current->next
104             delete temp
105 }
106
107 /*****
108 * Hash function
109 *****/
110 unsigned int CourseHashTable::hash(string courseNumber) {
111     digitString = ""
112     for each character in courseNumber
113         if character is digit
114             digitString = digitString + character
115
116     if digitString is not empty
117         return atoi(digitString) % tableSize
118     else
119         // fallback: use first character code
120         return courseNumber[0] % tableSize
121 }
122
123 /*****
124 * Dynamic resize function
125 *****/
126 void CourseHashTable::resize() {
127     print "Resizing hash table from " + tableSize + " to "
128
129     // save old buckets

```

```
130     vector<Node> oldBuckets = buckets
131     unsigned int oldSize = tableSize
132
133     // double size and move to next prime
134     tableSize = nextPrime(tableSize * 2)
135     print tableSize + " buckets."
136
137     // create new bucket array
138     buckets.clear()
139     buckets.resize(tableSize)
140     numElements = 0
141
142     // rehash all elements from old buckets
143     for i from 0 to oldSize - 1
144         if oldBuckets[i].key != UINT_MAX
145             // rehash main node
146             Insert(oldBuckets[i].course)
147
148             // rehash chained nodes
149             current = oldBuckets[i].next
150             while current != nullptr
151                 Insert(current->course)
152                 temp = current
153                 current = current->next
154                 delete temp
155     }
156
157     /*****
158     * Insert course into hash table
159     *****/
160     void CourseHashTable::Insert(Course course) {
161         // convert courseNumber to key and hash it
162         key = hash(course.courseNumber)
163         // retrieve node/bucket using hash key
164         node = &buckets.at(key)
165
166         // if the head bucket/node is empty
167         if node->key == UINT_MAX
168             node->key = key
169             node->course = course
170             node->next = nullptr
171             numElements++
172         return
173     }
```

```

174 // update existing course
175 if node->course.courseNumber == course.courseNumber
176     node->course = course
177     return
178
179 // traverse chain to find course or end
180 chainLength = 1
181 while node->next != nullptr
182     node = node->next
183     chainLength = chainLength + 1
184     if node->course.courseNumber == course.courseNumber
185         node->course = course
186         return
187
188 // add new node at end of chain
189 node->next = new Node()
190 node->next->key = key
191 node->next->course = course
192 numElements++
193
194 // check if resize needed due to long chains
195 // chain length after append is chainLength + 1 from original head
196 if (chainLength + 1) > maxChainLength
197     print "Chain length " + (chainLength + 1) + " exceeds threshold"
198     resize()
199 }
200
201 /*****
202 * Search for specific course (class method)
203 *****/
204 void CourseHashTable::searchCourse(string courseNumber) {
205     key = hash(courseNumber)
206     node = &buckets.at(key)
207
208     // check if bucket is empty
209     if node->key == UINT_MAX
210         print "Course '" + courseNumber + "' not found."
211         return
212
213     // search through chain
214     while node != nullptr
215         if node->course.courseNumber == courseNumber
216             print node->course.courseNumber + ", " + node->course.name
217             if node->course.prerequisites.size > 0

```

```

218         print "Prerequisites:"
219         for each prereq in node->course.prerequisites
220             print " " + prereq
221         else
222             print "No prerequisites"
223         return
224     node = node->next
225
226     // not found
227     print "Course '" + courseNumber + "' not found."
228 }
229
230 /*****
231 * Print all courses (sorted by bubble for now -- class method)
232 *****/
233 void CourseHashTable::printAll() {
234     print all courses header
235
236     // collect all courses for sorting
237     vector<Course> allCourses
238
239     // iterate through all buckets
240     for i from 0 to tableSize - 1
241         if buckets[i].key != UINT_MAX
242             // add main node course
243             allCourses.push_back(buckets[i].course)
244
245             // add chained nodes
246             node = buckets[i].next
247             while node != nullptr
248                 allCourses.push_back(node->course)
249                 node = node->next
250
251     // sort courses by courseNumber
252     for i from 0 to allCourses.size - 1
253         for j from 0 to allCourses.size - i - 2
254             if allCourses[j].courseNumber > allCourses[j+1].courseNumber
255                 swap allCourses[j] and allCourses[j+1]
256
257     // print sorted courses
258     for each course in allCourses
259         print course.courseNumber + ", " + course.name
260 }
261 /****

```

```
262 * clear is a helper for loading data
263 *****/
264 void CourseHashTable::clear() {
265     for i from 0 to tableSize - 1
266         current = buckets[i].next
267         while current != nullptr
268             temp = current
269             current = current->next
270             delete temp
271         buckets[i].key = UINT_MAX
272         buckets[i].next = nullptr
273     numElements = 0
274 }
275
276
277 /*****/
278 * Load courses from file
279 *****/
280 bool loadCourses(string filePath, CourseHashTable* ht) {
281     set<string> courseNumbers // track valid courses
282     vector<Course> parsed // extra vector of what's inserted for validation
283     integer lineNumber = 0
284
285     open filePath for reading
286     if open fails
287         print "Error loading " + filePath
288         return false
289     print "Loading file path " + filePath
290
291     //starts clean
292     ht->clear()
293
294     // read and parse lines and build course objects
295     for each line in file
296         lineNumber = lineNumber + 1
297         if line is empty
298             continue to next line
299
300         vector<string> fields = splitCSV(line)
301
302         // validate at least 2 parameters (courseNumber and name)
303         if fields.size < 2
304             print "Error line " + lineNumber + ": missing courseNumber or name"
305             close file
```

```
306         ht->clear()
307         return false
308
309     // validate fields not empty after trim
310     if fields[0] == "" or fields[1] == ""
311         print "Error line " + lineNumber + ": empty courseNumber or name"
312         close file
313         ht->clear()
314         return false
315
316     // building course object from fields
317     define new Course course object
318     course.courseNumber = fields[0]
319     course.name = fields[1]
320
321     // add prerequisites if they exist
322     for i from 2 to fields.size - 1
323         prereq = fields[i]
324         if prereq == ""
325             print "Warning line " + lineNumber + ": empty prerequisite ignored"
326             continue
327         // check length and if prerequisite contains at least one digit
328         if length(prereq) < 4 OR not containsDigit(prereq)
329             print "Error line " + lineNumber
330             + ": malformed prerequisite " + prereq
331             close file
332             ht->clear()
333             return false
334         course.prerequisites.push_back(prereq)
335
336     // duplicate check for courses
337     if course.courseNumber is in courseNumbers
338         print "Error line " + lineNumber
339         + ": duplicate course " + course.courseNumber
340         close file
341         ht->clear()
342         return false
343
344     // record that it's been seen
345     add course.courseNumber into courseNumbers set
346
347     // insert into hash table
348     ht->Insert(course)
349     parsed.push_back(course)
```

```
350
351     close file
352
353     // validation loop for each prerequisite must exist in courseNumbers
354     for each course in parsed
355         for each prereq in course.prerequisites
356             if prereq is not in courseNumbers
357                 print "Error: unknown prerequisite " + prereq
358                 ht->clear()
359                 return false
360
361     print "Successfully loaded " + courseNumbers.size + " courses."
362     return true
363 }
364
365 /*****
366 * Main entry
367 *****/
368 main() {
369     string csvPath, courseNumber
370     clock_t ticks // timer variable
371
372     CourseHashTable* courseTable = new CourseHashTable()
373
374     integer choice = 0
375     // main loop until user exits
376     while choice is not 9
377         print display menu
378         case 1: Load Data Structure
379         case 2: Print Course List
380         case 3: Search and Print Course
381         case 9: Exit
382         Enter a choice
383
384         get user choice from input
385
386         switch (choice)
387             case 1:
388                 // load file data
389                 print "Enter CSV file path or press Enter for default: "
390                 get input for csvPath
391                 if csvPath is empty
392                     print "File path incorrect or empty, defaulting to coursefile.csv"
393                     csvPath = "coursefile.csv"
```



```
394
395     // initialize timer variable before loading
396     ticks = clock()
397
398     // load courses
399     loaded = loadCourses(csvPath, courseTable)
400     if not loaded
401         print "Failed to load: " + csvPath + ", trying default"
402         loaded = loadCourses("coursefile.csv", courseTable)
403
404         if not loaded
405             print "Failed to load courses."
406         else
407             print "Data structure loaded."
408             ticks = clock() - ticks
409             print "time: " + ticks + " clock ticks."
410             print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds
411     ."
412     break
413
414 case 2:
415     courseTable->printAll()
416     break
417
418 case 3:
419     print "Input course number to search: "
420     get courseNumber from user input
421     if courseNumber is empty
422         print "Invalid input, try again"
423     else
424         ticks = clock()
425         courseTable->searchCourse(courseNumber)
426         ticks = clock() - ticks
427         print "time: " + ticks + " clock ticks."
428         print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds."
429     break
430
431 case 9:
432     break
433
434 default:
435     print invalid choice, try again
436
437 print Good Bye
```

```

437
438 // clean up
439 delete courseTable
440 return
441 }

```

### 1.3 Binary Search Tree

```

1  /*****
2  * BINARY SEARCH TREE
3  * DATA STRUCTURE IMPLEMENTATION
4  *****/
5
6  /*****
7  * Helper functions:
8  * Split the CSV line at commas into parts.
9  *****/
10 vector splitCSV(string line) {
11     split line on commas into parts
12     for each part
13         part = trim(part) // remove leading/trailing spaces
14     return vector of trimmed parts
15 }
16
17 /*****
18 * Trim whitespace:
19 *****/
20 string trim(string text) {
21     remove leading spaces
22     remove trailing spaces
23     return trimmed text
24 }
25
26 /*****
27 * Check if string contains digit
28 *****/
29 bool containsDigit(string course) {
30     for each character in course
31         if character is digit ('0' through '9')
32             return true
33     return false
34 }
35
36 /*****

```

```

37 * Structure for course data
38 *****/
39 struct Course {
40     string courseNumber // unique identifier
41     string name // course title/name
42     vector<string> prerequisites // list of prerequisite course numbers
43 }
44
45 /*****
46 Structure Node for binary search tree
47 *****/
48 struct Node {
49     Course course // the course data stored in this Node
50     Node* left // pointer to the left child
51     Node* right // pointer to the right child
52
53     // default constructor with course parameter
54     // When I create a new Node, I pass a course object to it.
55     // Useful for inserting courses
56     Node(Course c) {
57         course = c // copies the passed Course into this node
58         left = nullptr
59         right = nullptr
60     }
61 }
62
63 /*****
64 * Binary Search Tree Class definition
65 *****/
66
67 class BinarySearchTree {
68     Node* root // creating a node pointer for the root of the tree
69
70     //public methods
71     void Insert(Course course)
72     void searchCourse(string courseNumber)
73     void printAll()
74     void clear()
75
76     //private helper methods
77     Node* findNode(string courseNumber) // helper to find a node for search
78     void deleteTree(Node* node) // needed for destructor
79     void inOrder(Node* node) // recursive traversal for printing
80 }

```

```
81
82 /*****
83 * Constructor
84 *****/
85 BinarySearchTree::BinarySearchTree() {
86     root = nullptr/empty
87 }
88
89 /*****
90 * Destructor
91 *****/
92 BinarySearchTree::~BinarySearchTree() {
93     deleteTree(root)
94 }
95
96 /*****
97 * Helper method for destructor
98 *****/
99 void BinarySearchTree::deleteTree(Node* node) {
100     if node == nullptr/empty
101         return
102     // post-order recursive deletion of entire tree to free memory
103     deleteTree(node->left)
104     deleteTree(node->right)
105     delete node
106 }
107
108 /*****
109 * Insert course into BST
110 *****/
111 void BinarySearchTree::Insert(Course course) {
112     // if tree is empty, create root
113     if root == nullptr/empty
114         root = new Node(course)
115         return
116
117     // start at root to find insertion point
118     current = root
119     parent = nullptr
120
121     // traverse tree to find where to insert
122     while current != nullptr (while it isn't empty)
123         parent = current
124
```

```
125     // if course already exists, update it
126     if current->course.courseNumber == course.courseNumber
127         current->course = course
128         return
129
130     // go left or right based on course number
131     // left if smaller, else right if bigger.
132     if course.courseNumber < current->course.courseNumber
133         current = current->left
134     else
135         current = current->right
136
137     // create new node and attach to parent
138     newNode = new Node(course)
139     if course.courseNumber < parent->course.courseNumber
140         parent->left = newNode
141     else
142         parent->right = newNode
143 }
144
145 /*****
146 * Search for course (find the data)
147 *****/
148 void BinarySearchTree::searchCourse(string courseNumber) {
149     current = root
150
151     // traverse tree until found
152     while current != nullptr // while not empty
153         if current->course.courseNumber == courseNumber
154             print current->course.courseNumber + ", " + current->course.name
155             if current->course.prerequisites.size > 0
156                 print "Prerequisites:"
157                 for each prereq in current->course.prerequisites
158                     print " " + prereq
159             else
160                 print "No prerequisites"
161             return
162         if courseNumber < current->course.courseNumber
163             current = current->left
164         else
165             current = current->right
166     // not found
167     print "Course '" + courseNumber + "' not found."
168 }
```

```
169
170 /*****
171 * Private helper to find a node for search
172 *****/
173 Node* BinarySearchTree::findNode(string courseNumber) {
174     current = root
175
176     while current != nullptr // while not empty
177         if current->course.courseNumber == courseNumber
178             return current
179         if courseNumber < current->course.courseNumber
180             current = current->left
181         else
182             current = current->right
183
184     return nullptr
185 }
186
187 /*****
188 * Print all courses (in-order traversal)
189 *****/
190 void BinarySearchTree::printAll() {
191     print all courses header
192     inOrder(root)
193 }
194
195 /*****
196 * Private method Print helper for PrintAll to avoid infinite recursion
197 *****/
198 void BinarySearchTree::inOrder(Node* node) {
199     if node == nullptr/empty
200         return
201     // traverse left subtree
202     inOrder(node->left)
203     // print current node course info
204     print node->course.courseNumber + ", " + node->course.name
205     // traverse right subtree
206     inOrder(node->right)
207 }
208
209 /*****
210 * clear is a helper for loadCourses
211 *****/
212 void BinarySearchTree::clear() {
```

```
213     deleteTree(root)
214     root = nullptr/empty
215 }
216 /*****
217 * load courses from file
218 * Re-wrote this to now return true/false instead of a pointer.
219 *****/
220 bool loadCourses(string filePath, BinarySearchTree* bst) {
221     set<string> courseNumbers // track valid courses
222     integer lineNumber = 0
223
224     open filePath for reading
225     if open fails
226         print out error + filePath
227         return false
228     print out loading file path + filePath
229
230     bst->clear() // clear for new load
231
232     // Read and parse lines
233     for each line in file
234         lineNumber = lineNumber + 1
235         if line is empty
236             continue to next line
237
238         vector<string> fields = splitCSV(line)
239
240         // validate at least 2 parameters (courseNumber and name)
241         if fields.size < 2
242             print format error due to courseNumber and or name size missing field
243             close file
244             bst->clear()
245             return false
246
247         // validate fields not empty after trim
248         if fields[0] == "" or fields[1] == ""
249             error due to empty courseNumber or name
250             close file
251             bst->clear()
252             return false
253
254         // building course object from fields
255         define new Course course object
256         course.courseNumber = fields[0]
```

```
257     course.name = fields[1]
258
259     // add prerequisites if they exist
260     for i from 2 to fields.size - 1
261         prereq = fields[i]
262         if prereq == ""
263             print "warn line " + lineNumber + ": empty prerequisite ignored"
264             continue
265         // check length and if prerequisite contains at least one digit
266         if length(prereq) < 4 OR not containsDigit(prereq)
267             print "error line " + lineNumber
268             + ": malformed prerequisite " + prereq
269             close file
270             bst->clear()
271             return false
272         course.prerequisites.push_back(prereq)
273
274     // duplicate check for courses
275     if course.courseNumber is in courseNumbers
276         print error duplicate course + lineNumber + course.courseNumber
277         close file
278         bst->clear()
279         return false
280
281     // record that it's been seen
282     add course.courseNumber into courseNumbers set
283     // insert into bst
284     bst->Insert(course)
285
286 close file
287
288 // validation loop: prerequisites exist as courses
289 // need to validate all courses in tree using courseNumbers set
290 for each courseNumber in courseNumbers set
291     node = bst->findNode(courseNumber)
292     if node != nullptr
293         for each prereq in node->course.prerequisites
294             if prereq is not in courseNumbers set
295                 print "Error: unknown prerequisite " + prereq
296                 bst->clear()
297                 return false
298
299 print Successfully loaded + courseNumbers.size + courses
300 return true
```



```
301 }
302
303 /*****
304 * main entry
305 *****/
306 main() {
307
308     string csvPath, courseNumber
309     // timer variable
310     clock_t ticks
311
312     BinarySearchTree* bst = new BinarySearchTree()
313
314     integer choice = 0
315     // main loop until user exits
316     while choice is not 9
317         print display menu
318         case 1: Load Data Structure
319         case 2: Print Course List
320         case 3: Search and Print Course
321         case 9: Exit
322         Enter a choice
323
324         get user choice from input
325
326         switch (choice)
327             case 1:
328                 // load file data
329                 print "Enter CSV file path or press Enter for default: "
330                 get input for csvPath
331                 if csvPath is empty
332                     print "File path incorrect or empty, defaulting to coursefile.
333 csv"
334
335                     csvPath = "coursefile.csv"
336
337                 // initialize timer variable before loading bids
338                 ticks = clock()
339                 loaded = loadCourses(csvPath, bst)
340                 if not loaded
341                     print "Failed to load: " + csvPath + ", trying default."
342                     loaded = loadCourses("coursefile.csv", bst)
343                     if not loaded
344                         print "Failed to load courses."
345                     else
```

```
344         print "Data structure loaded."
345         ticks = clock() - ticks
346         print "time: " + ticks + " clock ticks."
347         print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds."
348     break
349 case 2:
350     bst->printAll()
351     break
352 case 3:
353     print "Input course number to search: "
354     get courseNumber from user input
355     if courseNumber.empty()
356         print "Invalid input, try again"
357     else
358         ticks = clock()
359         bst->searchCourse(courseNumber)
360         ticks = clock() - ticks
361         print "time: " + ticks + " clock ticks."
362         print "time: " + ticks * 1.0 / CLOCKS_PER_SEC + " seconds."
363     break
364 case 9:
365     break
366 default:
367     print invalid choice, try again
368
369 print Good Bye
370
371 // clean up
372 delete bst
373 return
374 }
```

## 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.