Jordan Clark
CourseList Pseudocode

class **CourseList**

structure to hold data for **Course**
courseNumber,
courseTitle,
vector<string> prerequisites

structure to hold data for **Node**
Course course
Node pointer left
Node pointer right

Node Constructor
left = null
right = null

Node Constructor (with course Data)
Node(Course course) :
Node()
course = aCourse

Structure for **hashTable_Item**
char* key
char* value
return hashTable_Item

Structure for **hashTable**
hashTable_Item** items
LinkedList** prerequisite
size *// return size of hashTable*
count // *to return number of elements*

Structure for **LinkedList**
hashTable_Item* item
struct LinkedList* next

method **parseFile(***splitter, delimiter = “,”***)**
// create empty list of strings for tokens
// declare start and end variables
// loop while (end is nonempty)
// take a portion from original string from start to end
// add it to tokens
// Update start
// Update end
// add last token to list

// return tokens

```
method loadCourseFile()
  // Declare data structure to store courses
  Vector<Course> courses / Hashtable<Course> courses / Tree<Course> courses

  try:
    // Open file and throw error if file fails to open
    ifstream fin("CourseInformation.txt", ios::in)
    if (file is not opening) {
      throw ifstream::failure("Failed to open file.");

    // declare variable to handle the strings
    string line
    while (getline(fin, line))
      Course course // In accordance to Vector/Hash/Tree
      vector<string> parsedLine = parseFile(line) // Create a string vector to hold parsed lines
      if (parsedLine.size() >= 2) // If there 2 or more parsed lines, assign them to index in string vector
        // Extract course information
        course.courseNumber = parsedLine[0]
        course.courseTitle = parsedLine[1]

        // Extract the prerequisites
        for (int i = 2; i < parsedLine.size(); ++i)
          / * attach every parsed line found after 2nd index to string vector titled prerequisites
            * which is found in the Course structure, referenced to by course object
            *\
          course.prerequisites.push_back(parsedLine[i])

      // Add the Course object to the vector
      courses.push_back(course)

      // Add  Course object to the hash table

      // Add Course object to the tree
      Insert(course)

      else
        cerr << "Invalid Format: At least 2 parameters required on each line"

    // Close file
    fin.close()

  catch (const ifstream::failure& e)
    cerr << "File Error: " << e.what() << endl

  // Return the vector of courses
  return courses

method quickSort(courses, low, hi)
  if ( low > hi) {return}
  lowEndIndex = partition(courses, low, hi)
  quickSort(courses, low, lowEndIndex)
  quickSort(courses, lowEndIndex + 1, hi)
```

method **partition(***courses, low, hi***)**

```
mid = low + (hi – low)/2
pivot = courses[mid]
done = false
while (not done)
   while (courses[low] < pivot) { low += 1}
   while (courses[low] < pivot) { low += 1}
   if (low > hi) { return done = true)}
   else
      temp = courses[hi]
      courses[hi] = courses[low]
      courses[low] = temp
      low += 1
      hi -= 1

return hi
```

method **Menu**()

```
output: "Welcome to the course planner.\n"
     "1. Load Data Structure.\n"
     "2. Print Course List.\n"
     "3. Print Course.\n"
     "4. Exit.\n"
     "What would you like to do?"

int userChoice:
switch (userChoice)
   case 1:
      load data from file into chosen data structure
      break

   case 2:
      retrieve data from data structure
      sort() // sort data in alphanumeric order
      printCourseInformation(Courses, courseNumber)
      break

   case 3:
      Output: "What course would you like to know about?"
      Input: UserInput
      SearchCourses() // returns course if found
      printCourseInformation(Courses, courseNumber)
      break

   case 4:
      Output: "Thank you for using the course planner."
      break

   default:
      output: userChoice + " is not a valid option."
      break
```

int numPrerequisiteCourses(Vector<Course> courses, Course c**)**

```
totalPrerequisites = prerequisites of course c
   for each prerequisite p in totalPrerequisites
      add prerequisites of p to totalPrerequisites
```

void printCourseInformation(Vector<Course> courses, String courseNumber**)**

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

/* Hashtable pseudocode *\

```
HashFunction(char* str)
 unsigned long i = 0

for (int j = 0; str[j]; j++)
   i += str[j]
return i % TABLE_SIZE
```

```
HashInsert(Course course)
   if (hashSearch(Course, course→key) == null)
      bucketList = Course[hash(course→key)]
      node = new Node
      node→next set to null
```

```
ListSearch(prerequisites, key)
   curNode = preprequisiteList→head
   while curNode is not empty
      if curNode→course == key
         return curNode
   otherwise return null
```

```
int numPrerequisiteCourses(Hashtable<Course> courses, char* key)
   totalPrerequisite = 0
   prerequisiteList = courses[hashFunction(key)]
   preprequisiteNode = ListSearch(prerequisites, key)

   if preprequisiteList is not empty
      for every preprequisiteNode found in preprequisiteList
         totalPrerequisite +=1

   return totalPrerequisite
```

```
void printSampleSchedule(Hashtable<Course> courses)
   for all key & value pair in <Courses>
      print key to course name
      if value has prerequisite
      for each prerequisite
         print prerequisite
```

```
void printCourseInformation(Hashtable<Course> courses, String courseNumber)
   for all courses
      if search courseNumber is the same as courseNumber
      print out info
   for each [prerequisite] in HashTable[course]
   print the prerequisite info
```

/* Tree pseudocode *\

```
tree ()
  set root to null

InOrder()
  call printCourseInfo function and pass root

Insert(Course course)
  if root is empty
    root is assigned new Node containing course info
  else
    this→addNode(root, course)

addNode(Node, course)
  if node→course.couseNumber is larger
    if left node is empty
      left Node becomes new Node containing course info
    else
      traverse down
  else
    if right node is empty
      right Node becomes new Node
    else  traverse down

int numPrerequisiteCourses(Tree<Course> courses)
  totalPrerequisite = 0
  current is the root
  while current is not empty

    if current pointer compares to the courseNumber
      while prerequisite not empty

        for node → course.prerequisite in prerequisites
          totalPrerequisite += 1

    if courseNumber is larger than the compared courseNumber
      current = current→left
    else
      current = current→ right

  return totalPrerequisite

void printSampleSchedule(Tree<Course> courses)
 if node is not empty
    printCourseInfo(node→left) traverse left subtree

    print node → course.courseNumber, node → course.title
    for node → course.prerequisite in prerequisites
      print prerequisite

    printCourseInfo(node→right) traverse right subtree

void printCourseInformation(Node* node)
  current is the root
  while current is not empty

    if current pointer compares to the courseNumber
      print node → course.courseNumber, node → course.title
      for node → course.prerequisite in prerequisites
        print prerequisite
    if courseNumber is larger than the compared courseNumber
      current = current→left
    else
      current = current→ right
```

Vector			
numPrerequisiteCourses	Line Cost	# Times Executed	Total Cost
totalPrerequisites = prerequisites of course c	1	1	1
for each prerequisite p in totalPrerequisites	1	N	N
add prerequisites of p to totalPrerequisites	1	1	1
		Total Cost	N + 2
		Runtime	O(N)
printCourseInformation	Line Cost	# Times Executed	Total Cost
LowEndIndex = partition(low, hi)	1	1	1
quickSort(listSize, low, lowEndIndex)	n	n / 2	log2N
quickSort(listSize, lowEndIndex +1, hi)	n	n / 2	log2N
for all courses	1	N	N
if course matches courseNumber	1	N	N
print out the course information	1	1	1
for each prerequisite of the course	1	N – 1	N(N-1)
print the prerequisite course information	1	1	1
		Total Cost	2N(N-1)
		Runtime	O(N^2)

Hash			
numPrerequisiteCourses	Line Cost	# Times Executed	Total Cost
totalPrerequisite = 0	1	1	1
prerequisiteList = courses[hashFunction(key)]	1	1	1
preprequisiteNode = ListSearch(prerequisites, key)	1	1	1
if preprequisiteList is not empty	1	N	N
for every preprequisiteNode found in preprequisiteList	1	N	N
totalPrerequisite +=1	1	N	N
return totalPrerequisite	1	1	1
		Total Cost	3N + 4
		Runtime	O(N)
printCourseInformation	Line Cost	# Times Executed	Total Cost
for all courses	1	N	N
if this course is the same as courseNumber	1	N	N
print out info	1	1	1
for each [prerequisite] in HashTable[course]	1	N	N
print the prerequisite info	1	1	1
		Total Cost	3N + 2
		Runtime	O(N)

Tree			
numPrerequisiteCourses	Line Cost	# Times Executed	Total Cost
TotalPrerequisite = 0 and current is the root	2	1	2
while current is not empty	1	N	N
if current pointer compares to the courseNumber	1	N	N
while prerequisite not empty	1	N	N
for node → course.prerequisite in prerequisites	1	N	N (N)
totalPrerequisite += 1	1	N	N
if courseNumber larger than the value of current node	1	N	N
Traverse left, else traverse right	1	1	1
return totalPrerequisites	1	1	1
		Total Cost	6N(N)+4
		Runtime	O(N^2)
printCourseInformation	Line Cost	# Times Executed	Total Cost
current is the root	1	1	1
while current is not empty	1	N	N
if current pointer compares to the courseNumber	1	N	N
print number and title	1	1	1
for node → course.prerequisite in prerequisites	1	N	N
print prerequisite	1	1	1
if courseNumber is larger than the compared courseNumber	1	N	N
traverse left. else traverse right	1	N	N
		Total Cost	5N + 3
		Runtime	O(N)

Advantages and disadvantages

The advantage of quickSort is that it could retrun the numPreprequisiteCourses in O(N) time but the drawback came in the second half of the printCourseInformation. We have to print out the course information for every course, but we also have to print every preprequisite for the courses that have them. This set our quickSort (log2N) back to O(N^2).

The advantage of the Tree was the opposite. Printing the course information could be done in O(N) runtime. The disadvantage is in the numPreprequisiteCourses because there is a for loop within a while loop. This means there is an outter loop running N times and an inner loop running (N-1) times. That brings the runtime to O(N^2).

The advantage of the hash algorithm is that both the numPreprequisiteCourses and printCourseInformation functions have a runtime of O(N). Tallying the preprequisite count requires a match to a key and search a node through a linked list. The printing function prints the course information from the hash table, then prints any itemsfrom the linkedList in a separate for loop. It's not nested like the others, allowing the algorithm to run at worst case O(N). The hash algorithm with chaining preprequisites in a linkedList is the best route for the project.