**Project One**

**Task One:**

* [Milestone One](Milestone%201.docx)
* [Milestone Two](Milestone%202.docx)
* [Milestone Three](Milestone%203.docx)

**Task Two:**

* Loop
  + Print “1. Load data 2. Print all (alphanumeric) 3. Print Course 9. Exit”
  + Read option
  + Switch option
    - Case 1
      * LoadData()
    - Case 2
      * PrintAllSorted()
    - Case 3
      * Prompt “Enter course number: “
      * Read key
      * PrintCourse(key)
    - Case 9
      * Break
    - Default
      * Print “Invalid selection”

**Task Three:**

**Vector**

**Load Data into Vector**

* Clear the vector of courses
* For each line in the input file
  + Split by commas into tokens
  + Create a course
  + Set course number = first token
  + Set title = second token
  + For any remaining tokens
    - Add each as a prerequisite to the course
  + Add the course to the vector

**Print All Courses (Alphanumeric)**

* Make a temporary copy of the vector
* Sort the copy by course number using alphanumeric order
  + Compare the letter prefix first (case-insensitive)
  + If the prefix matches
    - Compare the numeric part as a number
* For each course in the sorted copy
  + Print course number and title

**Print One Course**

* Prompt for a course number
* For each course in the vector
  + If numbers match
    - Print number and title
    - If it has prerequisites
      * Print each prerequisite
    - Else print “No prerequisites”
    - Stop searching
  + Else
    - Print “Course not found”

**Hash Table**

**Load Data into Hash Table**

* Clear the hash table
* For each line in the input file
  + Split by commas into tokens
  + Create a course and fill fields (number, title, prerequisites)
  + Insert into the hash table using course number as the key

**Print All Courses (Alphanumeric)**

* Create an empty list of keys
* For each entry in the hash table
  + Add the course number (key) to the list
* Sort the list of keys using alphanumeric order
  + Compare letter prefix (case-insensitive), then numeric part
* For each key in the sorted list
  + Look up course in hash table
  + Print course number and title

**Print One Course**

* Prompt for a course number
* For each course in hash table
  + If numbers match
    - Print number and title
    - If it has prerequisites
      * Print each prerequisite
    - Else print “No prerequisites”
    - Stop searching
  + Else
    - Print “Course not found”

**Binary Search Tree**

**Load Data into BST**

* Clear the tree
* For each line in the input file
  + Split by commas into tokens
  + Create a course and fill fields
  + Insert into the BST using the course number as the key

**Print All Courses (Alphanumeric)**

* Do an in-order traversal of the BST
  + Visit left child
  + Print this node’s course number and title
  + Visit right child

**Print One Course**

* Prompt for a course number
* Search BST for that key
  + If found
    - Print number and title
    - If it has prerequisites
      * Print each prerequisite
    - Else print “No prerequisites”
    - Stop searching
  + Else
    - Print “Course not found”

**Tasks** **Four and Five:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Data Structure** | **Read & split** | **Create object** | **Insert Cost** | **Total worst case Big-O** | **Advantages** | **Disadvantages** |
| **Vector** | O(1+p) | O(1+p) | O(1)\* | O(n^2) | * Simple * Contiguous memory * Fastest iteration * Minimal overhead | * Course lookup is linear * Insert/delete in middle is O(n) |
| **Hash Table** | O(1+p) | O(1+p) | O(1) avg O(n) worst | Worst O(n^2) | * Course lookup/insert average ideal for “print single course” * Load is linear on average | * Printing in order needs gather → sort keys O(n log n) * Worst-case degenerates if poor hash or tiny table; uses extra memory for buckets |
| **BST (unbalanced)** | O(1+p) | O(1+p) | O(log n) avg O(n) worst | Worst O(n^2) | * In-order traversal prints sorted output in O(n) without extra sort * Lookup/insert O(log n) on average | * Worst-case O(n) if tree becomes skewed (e.g. sorted input) * More complex than vector; pointer overhead |
| **BST (balanced)** | O(1+p) | O(1+p) | O(log n) worst-case (height kept O(log n)) | O(n log n) | * Guaranteed O(log n) search/insert/delete (no “sorted input” degeneration” * In-order traversal prints the full list already sorted * Good balance between frequent single lookups and frequent sorted listings * Predictable performance across datasets | * Load time higher than vector or hash (avg) * Larger constants (rotations, pointers), usually slower than a hash table for single lookups * Extra memory per node * More complex to implement and debug |

n = number of courses, p = average prerequisites per course

Amortized note: With geometric capacity growth, vector push\_back is O(1) amortized; occasional resizes copy existing elements but the total cost of n appends is O(n). Hash-table inserts are O(1) on average under a good hash and bounded load factor (rehashes are amortized).

**Task Six:**

Vector! This is surely a personal bias because if you asked me before this course, I still would’ve said vector barely knowing what the other options were. A vector will be the easiest to implement, easiest to debug when things go wrong, and have the least amount of overhead. It has the fastest iteration, and its only downside is where it lags behind in potential lookup speeds as it traverses the entire vector. However, in this case; I don’t imagine a university will have enough courses offered for the speed differences to be perceptible. The biggest universities offer hundreds of different courses. Even if the scope is widened to each time they offer a course (like adding the term to the ID) the number is still only in the thousands. For quite a while, CPUs have been rated in billions of ticks per second, so it’s working through a thousand items faster than we can comprehend. I think chasing the fastest lookup speeds in a data set this small shouldn’t be near as high of a priority as the code’s maintainability, especially when that maintainability also adds overhead.