**LINKED LIST**

**LINKED LIST PSEUDOCODE (OPEN, READ, PARSE):**

* CREATE struct
  + Number : string
  + Title: string
  + Prereqresite: string \*\*OPTIONAL\*\*
* CREATE vector of type Course
* OPEN file to be read
* WHILE lines remain in file
  + GET line
  + IF line has 2 or more parameters
    - SPLIT string by delimiter
    - SET currentLine.number to string [0]
    - SET currentLine.title to string [1]
    - IF string [2] does NOT equal NULL
      * SET currentLine.prereq to string [2]
  + PUSH\_BACK currentLine on course Vector

**LINKED LIST PSEUDOCODE (VALIDATE Prerequisite)**

* ITERATE through course vector
  + IF course had prerequisite
    - SET prereq as tempPrereq
    - ITERATE through course vector
      * IF tempPrereq does not equal a course title
        + DELETE course for tempPrereq

**LINKED LIST PSEUDOCODE (SEARCH AND PRINT)**

* GET searchString
* ITERATE through course vector
  + IF searchString EQUALS course number OR searchString EQUALS course title
    - PRINT course number , course title, and any prereq
  + ESCAPE function

**HASH TABLE**

**HASH TABLE PSEUDOCODE (OPEN, READ, PARSE)**

* CREATE struct for course \*Course
  + Number : string
  + Title: string
  + Prerequisite: string \*\*OPTIONAL\*\*
* CREATE struct to hold course \*Node
  + Course: course
  + Node: \*left
  + Node: \*right
  + CONSTRUCT node default
    - ASSIGN left to nullptr
    - ASSIGN right to nuillptr
  + CONSTRUCT node with a course parameter of course
    - CALL default constructor
      * and
    - SET course to equal parameter course
* CREATE root as Node\* type
* OPEN file to be read
* WHILE lines remain in file
  + GET line
  + IF line has 2 or more parameters
    - SPLIT string by delimiter
    - SET currentLine.number to string [0]
    - SET currentLine.title to string [1]
    - IF string [2] does NOT equal NULL
      * SET currentLine.prereq to string [2]
  + IF root is null add node as root
  + FOR subsequent entries is current node **Number** greater than or less than the **Number** for incoming nodes
    - IF node -> left is null
      * ADD incoming node as node-> left if current is greater than incoming
    - ELSE recursively call add node pointing to node-left and pass incoming bid
    - IF node ->right is null
      * ADD incoming node as node-> right if current is greater than incoming
    - ELSE recursively call add node pointing to node-right and pass incoming bid

**HASH TABLE PSEUDOCODE (VALIDATE Prerequisite)**

* FOR each prerequisite
* ITERATE through leaves by visiting all nodes with inorder traversal
* InOrder takes Node
  + IF node is not null
    - RECURSIVELY call inOrder with parameter at node->left
    - When node-> is null compare prereq to course number
      * IF match BREAK
    - RECURSIVELY call inOrder with parameter at node->right

**HASH TABLE PSEUDOCODE (PRINT)**

* ITERATE through nodes with in order sort
  + IF node is not null
    - RECURSIVELY call inOrder with parameter at node->left
    - Print course info
    - RECURSIVELY call inOrder with parameter at node->right

**BINARY SEARCH TREE**

**BINARY SEARCH TREE PSEUDOCODE (OPEN, READ, PARSE)**

* CREATE struct for course \*Course
  + Number : string
  + Title: string
  + Prerequisite: string \*\*OPTIONAL\*\*
* CREATE struct to hold course \*Node
  + Course: course
  + Node: \*left
  + Node: \*right
  + CONSTRUCT node default
    - ASSIGN left to nullptr
    - ASSIGN right to nuillptr
  + CONSTRUCT node with a course parameter of course
    - CALL default constructor
      * and
    - SET course to equal parameter course
* CREATE root as Node\* type
* OPEN file to be read
* WHILE lines remain in file
  + GET line
  + IF line has 2 or more parameters
    - SPLIT string by delimiter
    - SET currentLine.number to string [0]
    - SET currentLine.title to string [1]
    - IF string [2] does NOT equal NULL
      * SET currentLine.prereq to string [2]
  + IF root is null add node as root
  + FOR subsequent entries is current node **Number** greater than or less than the **Number** for incoming nodes
    - IF node -> left is null
      * ADD incoming node as node-> left if current is greater than incoming
    - ELSE recursively call add node pointing to node-left and pass incoming bid
    - IF node ->right is null
      * ADD incoming node as node-> right if current is greater than incoming
    - ELSE recursively call add node pointing to node-right and pass incoming bid

**BINARY SEARCH TREE PSEUDOCODE (VALIDATE Prerequisite)**

* FOR each prerequisite
* ITERATE through leaves by visiting all nodes with inorder traversal
* InOrder takes Node
  + IF node is not null
    - RECURSIVELY call inOrder with parameter at node->left
    - When node-> is null compare prereq to course number
      * IF match BREAK
    - RECURSIVELY call inOrder with parameter at node->right

**BINARY SEARCH TREE PSEUDOCODE (PRINT)**

* ITERATE through nodes with in order sort
  + IF node is not null
    - RECURSIVELY call inOrder with parameter at node->left
    - Print course info
    - RECURSIVELY call inOrder with parameter at node->right

**MENU**

**MENU PSEUDOCODE**

* CREATE selection variable INITIALIZE to 0
* WHILE section NOT equal to 9
  + DISPLAY menu selections
    - 1. Load Data
    - 2. Print Course List
    - 3. Print Course
    - 9. Exit
  + IF selection equals 1
    - INITIALIZE timer variable “ticks” and SET to clock()
    - CALL **OPEN READ PARSE** function defined above
    - CALL **VALIDATE Prerequisite** function defined above
    - SET ticks to equal clock() – ticks
    - DISPLAY ticks as time elapsed
  + IF selection equals 2
    - CALL **PRINT** function defined above
  + IF selection equals 3
    - CREATE function based on **VALIDATE Prerequisite** described above
    - FUNCTION takes string parameter courseName
      * FOR each course compare provided parameter with courseName
        + IF they match

DISPLAY course information

Return

* + - * DISPLAY No such course
  + IF selection equals 9
    - DISPLAY Thank you message

**Evaluation**

Linked list:

From a memory standpoint linked lists would use more memory than an equally sized array because each node not only contains its value but also point to the location of the next node. However, because of its dynamic nature if any changes to the list were to occur (insertions/deletions) the linked list would take less due to its ability to be dynamically sized. A benefit of a linked list is insertions and deletions can be completed very quickly (O(1)) because it is simply changing pointers. A drawback of the linked list is searching can be slow and costly as we have to navigate the list by going to each node

Hash table:

A hash tables complexity really results from the hashing function. At its base a hash table is an unordered key/value setup with the key running through a hash function to point to a particular stored value. With a simple hash function insertions deletion and even searches can all be done in O(1) as you are really only dealing with the key position which is a vector. The major benefit of using a hash table is the speed of insertions deletions and searches they are the fastest of the other collection types. A drawback of the linked list is something called a collision wherein an item is trying to be put in the same bucket as another item. This requires some additional setup but can be overcome relatively simply.

Binary Search Tree:

Binary search trees can be really great at memory or really terrible with it really depending on how they are set up and sometime more importantly, what data they are receiving. At its base a binary search tree is essentially a linked list with two pointers on every node. First a root is set then the subsequent node is put on its right or left pointer based on if its higher or lower. If the data you are working with is completely unordered and somewhat balanced (about half the entries above the median and half below) the binary search tree works great. The insertions and deletions are still the benefit hear while search remains slow. The problem presents when the data in question is already somewhat ordered, even if not exact. This essentially makes the Binary search tree are more expensive linked list and you can easily (especially with recursion methods) run out of memory. Insertions and deletions are also slower on the binary search tree than any of the other collections as it is entirely dependent on the height of the tree.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Insert | Delete | Search |
| Linked List | O(1) | O(1) | O(n) |
| Hash Table | O(1) | O(1) | O(1) |
| Binary Search Tree | O(height) | O(height) | O(height) |

Recommendation:

The linked list is probably the most straight forward and quickest of the three to set up correctly, the hash table really excels in insertions, deletions and searches (in O(1) constant time), the binary search tree I don’t feel is right for this project namely because of the memory issues outlined above. That leaves us with linked list or hash table the biggest difference between the two in terms of complexity is the search functionality since both hash tables and linked lists can handle insertions / deletions in O(1). Because search for a linked list is O(n) (worse case it travel to every other node first) and it is a constant O(1) with Hash tables, I would recommend the implementation of a hash table.