**Southern New Hampshire University**

**Joshua Wozny**

[**joshua.wozny@snhu.edu**](mailto:joshua.wozny@snhu.edu)

**February 7, 2023**

**CS-300: Project One**

*ABCU Computer Science Dept.: Course List with Prerequisites Project*

The Computer Science Department has requested an analysis of several data structure options to store course information - including pre-requisite courses - to provide optimal performance for an application we are developing with the following functions:

1. Read and parse data from a .csv file, with each line of file representing a single course
2. Create course objects to represent each course - course number is an alphanumeric string
3. Store each object in a data structure
4. Search, retrieve, and display a single course object using the course number, including pre-requisite courses
5. Display all courses in sorted order by course number, including pre-requisite courses

We have been asked to evaluate the time and space complexities of the application, based on the use of three data structures: vector, hash table, and binary search tree. We have been asked to make a recommendation based on this analysis. Each data structure is unique with advantages and disadvantages.

Vectors store data by index in an unsorted state utilizing pointers in memory. This yields great management of memory and fast insertions. However it also yields slower sorting, dependent on the sorting algorithm used, and slower searching unless searching on the index. It is a good choice when an application needs to have many data elements added, without a need for sortation and only a rare need to search for particular data elements.

Hash tables store data in buckets using a hashed key. It is an efficient way to store and retrieve data, but the data is unsorted and cannot be sorted in place making any functions requiring sortation computationally expensive. Hash tables provide for fast insertions and searching and are therefore a solid choice in many circumstances.

Binary Search Trees are also fairly versatile with solid performance for insertions and searches, however hash tables outperform them in many circumstances. Binary search trees shine brightest where an application needs to have sorted data, as no additional sorting of the data is required as the binary search tree maintains its data elements in sorted order.

Below are the methods we will be evaluating for each of these data structures.

void DisplayMenu()

// Method remains the same for all data structures

// Time complexity: constant

// Space complexity: constant

* Print Menu Items
  + 1: Load Data
  + 2: Display Course
  + 3: Display Course Catalog
  + 4: Exit
* Read input into selection
* Switch (selection)
  + 1: LoadData(ParseCSVFile(string fileLocation)
  + 2:

{

Print message to request CourseNumber

ReadInput into courseNumber

DisplayCourse(string courseNumber)

}

* + 3: DisplaySortedCourseList()
  + 4: Exit()

void LoadData(arr parsedFile)

// Method remains the same for all data structures

// Time complexity: constant

// Space complexity: constant

* For each element in parsedFile:
  + Initialize DataStructure courses
  + course = createCourse()
    - //Method remains the same for all data structures
    - // Time complexity: O(n)
    - // Space Complexity: O(n)
    - Initialize a Course object with the following elements
      * string courseNumber = first element of row
      * string courseName = second element of row
      * Initialize a vector<string> prerequisites =
        + for each element of row after 2, if it exists, add to prerequisites
  + courses.Insert(course)

void DisplayCourse(string courseNumber)

// Method remains the same for all data structures

// Time complexity: constant

// Space complexity: constant

* course = Search(courseNumber)
* DisplayCourse(course)

Void DisplaySortedCourseList()

// Method remains the same for all data structures

// Time complexity: constant

// Space complexity: O(n)

* courses = SortCourses()
* For each course in courses
  + DisplayCourse(course)

int Exit()

// Method remains the same for all data structures

// Time complexity: 1

// Space complexity: 1

* Return 0 to main function

parsedFile ParseCSVFile(string fileLocation)

// Method remains the same for all data structures

// Time complexity: O(n)

// Space Complexity: O(n)

* Open a file stream
* Initialize a vector to hold each line from the input file
* While there are still lines in the file
  + add each line to the vector of input lines
* Parse the first element in the vector as the headers
  + separate the line into separate elements with each element separated by a comma into a vector to store header elements
    - add each element into the header vector
    - add the header element to the vector of rows as the first element
* Iterating over the vector of input lines, starting at the second element
  + for each line, separate the line into separate elements with each element separated by a comma into a vector of row elements
  + add the vector of row elements to the vector of rows
* Iterating over the vector of rows
  + count the number of elements in the row vector.
  + If number of elements < 2, return an error,
  + otherwise, iterating over the vector of rows again
    - add value in first element into a vector of course numbers
  + Iterating over the vector of rows again
    - for each element starting at element three
    - if value is in the vector of course numbers is in vector of course numbers
    - List is valid, return the vector of rows
  + List is invalid, return an error

DisplayCourse(course)

//Method remains the same for all data structures

// Time complexity: O(n)

// Space Complexity: constant

* Print to screen course.courseNumber course.courseName course.prerequisites

void Insert(Course course)

// Vector Implementation

// Time Complexity: Avg: O(1), Worst O(n):

// Space Complexity: O(1)

// Hash Table Implementation

// Time Complexity: Avg: O(1), Worst O(n):

// Space Complexity: O(m+n), where m = size of hash table

* Hash function returns an int between 0 and m (not inclusive of m)
* Create a key = hash(course.courseNumber
* Create new node
* If table.at(key) is empty, intialize a LinkedList with course
* Otherwise LinkedList.Append(course)
  + Append to tail

// Binary Search Tree Implementation

// Time Complexity: Avg: O(logn), Worst O(n):

// Space Complexity: O(n)

* if root equal to null ptr, root = newNode(course)
* otherwise, addNode(root, course)
  + addNode(Node\* node, Course coursebid)
    - if node is larger then add to left
      * if no left node
        + left = newNode(bid)
    - else recurse down the left node
      * if node is larger then add to left
        + if no left node

left = newNode(bid)

* else if if no right node
  + right = newNode(bid)
  + else recurse down the left node
    - if node is larger then add to left
      * if no left node
        + left = newNode(bid)
      * else if if no right node
      * right = newNode(bid)

Course Search(courseNumber)

// Vector Implementation

// Time Complexity: Avg: O(n)

// Space Complexity: O(1)

// Hash Table Implementation

// Time Complexity: Avg: O(1), Worst O(n):

// Space Complexity: O(n)

* Linked List table.at(hash(courseNumber))
  + if element is empty, return empty course;
  + otherwise, LinkedList.Search(courseNumber)
    - Iterates through each node of LinkedList
      * If node.course.courseNumber == courseNumber
        + Return node.course
      * Otherwise return empty course.

// Binary Search Tree Implementation

// Time Complexity: Avg: O(logn), Worst O(n):

// Space Complexity: O(n)

* set current node equal to root
* keep looping downwards until bottom reached or matching bidId found
  + if match found, return current bid
    - if bid is smaller than current node then traverse left
    - else traverse left

Iterable SortCourses()

// **Vector Implementation using QuickSort**

// vector can be sorted in place and returned

// Time Complexity: Avg: O(n\*logn), Worst O(n^2):

// Space Complexity: O(logn)

* QuickSort (vector list, int start, int end)
  + set mid as int (index of the middle pivot point), initialize as 0
  + Base case for recursion: if (start>=end) then list is already sorted
* Partition list into low and high such that mid is location of last element in low
  + mid = partition(list, start, end)
    - set low and high equal to start and end
    - middle element as pivot point mid as int = (start + end)/2
    - set pivot as value of list at mid
    - keep incrementing low index while list at low < list at pivot
    - keep decrementing high index while list at pivot < list at high
    - If there are zero or one elements remaining, all bids are partitioned.
    - value of high is returned to QuickSort
* Recursively sort low partition: QuickSort(list, start, mid) (Repeats until Base Case)
* Recursively sort high partition: QuickSort(list, mid+1, end) (Repeats until Base Case)

// **Hash Table Implementation**

// has table cannot be sorted in place, must iterate over each item in the hash table

// with array or vector returned

// Time Complexity: O(n^2)

// Space Complexity: O(n+logn)

* courses = vector<Courses> intialized with n elements, n=HashTable::Size()
* HashTable::InOrder()
  + For each course in hashTable add to courses.
* return vector = SortCourses() - see vector implementation

// **Binary Search Tree Implementation**

// binary search tree is already sorted, return itself

// Time Complexity: constant

// Space Complexity: constant

* Return self

Analysis Summary

| **Application functions independent of data structure implementation** | | | | | | |
| --- | --- | --- | --- | --- | --- | --- |
| **Average case** | **Display Menu()** | **Parse()** | **Display course()** | **Display catalog()** | **Exit()** | **Total** |
| **Time Complexity** | 4 | O(n) | O(n) | O(n) | 1 | O(n) |
| **Space Complexity** | 4 | O(n) | 2 | 3 | 1 | O(n) |

| **Application functions dependent on data structure implementation** | | | | |
| --- | --- | --- | --- | --- |
| **Average case** | **Load()** | **Search()** | **Sort()** | **Total** |
| **Time Complexity** |  |  |  |  |
| Vector | O(1) | O(n) | O(n\*logn) | O(n) |
| Hash Table | O(1) | O(1) | O(n^2) | O(n^2) |
| Binary Search Tree | O(logn) | O(logn) | 1 | O(logn) |
| **Space Complexity** |  |  |  |  |
| Vector | O(1) | O(1) | O(logn) | O(logn) |
| Hash Table | O(n) | O(n) | O(n+logn) | O(n) |
| Binary Search Tree | O(n) | O(n) | 1 | O(n) |

**Recommendation**

Based on the performance analysis, I would recommend using the binary search tree. It performs significantly better than the other options due to the overhead of sorting required for hash tables, and the relatively poor performance of searching for an item in a vector.

Other considerations:

If printing the course catalog in an unsorted manner was adequate, a hash table would be a superior option. In most situations a vector would be a poor choice unless memory is of greater concern than the speed of operations.