Runtime Analysis and Memory for Data Structures

**Runtime Analysis**

| ***Vector*** | | | |
| --- | --- | --- | --- |
| ***Opening, reading, parsing, checking errors, creating course objects*** | | | |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Initialize fileStream to get contents** | 1 | 1 | 1 |
| **Initialize line to hold line from file** | 1 | 1 | 1 |
| **Intialize lineStream to hold line’s contents** | 1 | 1 | 1 |
| **Initialize token to hold word in line** | 1 | 1 | 1 |
| **Open filename with fileStream** | 1 | 1 | 1 |
| **Initialize tokenCount to count tokens per line** | 1 | 1 | 1 |
| Get line(s) from fileStream | 1 | n | n |
| Fill lineStream with current | 1 | n | n |
| SET tokenCount to 1 | 1 | n | n |
| Create Course aCourse for each line | 1 | n | n |
| GET token from lineStream up to ‘,’ until none left | 1 | 2n | 2n |
| IF count == 1 | 1 | n | n |
| SET aCourse’s courseNumber to token | 1 | n | n |
| increment tokenCount | 1 | n | n |
| ELSE IF count == 2 | 1 | n | n |
| SET aCourse’s courseName to token | 1 | n | n |
| increment tokenCount | 1 | n | n |
| ELSE |  |  |  |
| IF token already in courses as course | 1 | n | n |
| add token to aCourse’s PreReqs | 1 | n | n |
| ELSE output file format error | 1 | 1 | 1 |
| increment tokenCount | 1 | n | n |
| IF count < 2 | 1 | 1 | 1 |
| OUTPUT “Error – each course must have course # and course name.” | 1 | 1 | 1 |
| Push aCourse to courses | 1 | n | n |
| Clear lineStream for next line | 1 | n | n |
| **Total Cost** | | | 17n + 9 |
| **Runtime** | | | O(n) |

| ***HashTable*** | | | |
| --- | --- | --- | --- |
| ***Opening the file, reading data, parsing each line, and checking for formatting errors.*** | | | |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Initialize fileStream to get contents** | 1 | 1 | 1 |
| **Initialize line to hold line from file** | 1 | 1 | 1 |
| **Intialize lineStream to hold line’s contents** | 1 | 1 | 1 |
| **Initialize token to hold word in line** | 1 | 1 | 1 |
| **Open filename with fileStream** | 1 | 1 | 1 |
| **Initialize tokenCount to count tokens per line** | 1 | 1 | 1 |
| Get line(s) from fileStream | 1 | n | n |
| Fill lineStream with current | 1 | n | n |
| SET tokenCount to 1 | 1 | n | n |
| Create Course aCourse for each line | 1 | n | n |
| GET token from lineStream up to ‘,’ until none left | 1 | 2n | 2n |
| IF count == 1 | 1 | n | n |
| SET aCourse’s courseNumber to token | 1 | n | n |
| increment tokenCount | 1 | n | n |
| ELSE IF count == 2 | 1 | n | n |
| SET aCourse’s courseName to token | 1 | n | n |
| increment tokenCount | 1 | n | n |
| ELSE |  |  |  |
| IF token already in Htable as course | 1 | n | n |
| add token to aCourse’s PreReqs | 1 | n | n |
| ELSE output file format error | 1 | 1 | 1 |
| increment tokenCount | 1 | n | n |
| IF count < 2 | 1 | 1 | 1 |
| OUTPUT “Error – each course must have course # and course name.” | 1 | 1 | 1 |
| Add aCourse to Htable | n | n | n^2 |
| Clear lineStream for next line | 1 | n | n |
| ***Creating course objects*** | | | |
| Create key for aCourse with hash from aCourse’s courseNumber | 1 | 1 | 1 |
| Create Node\* node to retrieve node via key | 1 | 1 | 1 |
| IF node == nullptr | 1 | 1 | 1 |
| Create new node newCourse using aCourse and key | 1 | 1 | 1 |
| Insert contents of newCourse into table at position[key] | 1 | 1 | 1 |
| ELSE IF node’s key == UINT\_MAX | 1 | 1 | 1 |
| Update node’s key to key | 1 | 1 | 1 |
| Update node’s course to aCourse | 1 | 1 | 1 |
| Update node’s next to nullptr | 1 | 1 | 1 |
| ELSE |  |  |  |
| WHILE node’s next != nullptr | 1 | n | n |
| SET node to node’s next | 1 | 1 | 1 |
| Create new node newCourse using aCourse and key | 1 | 1 | 1 |
| SET node’s next to newCourse | 1 | 1 | 1 |
| **Total Cost** | | | n^2 + 17n + 21 |
| **Runtime** | | | O(n^2) |

| ***Binary Search Tree*** | | | |
| --- | --- | --- | --- |
| ***Opening the file, reading data, parsing each line, and checking for formatting errors.*** | | | |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Initialize fileStream to get contents** | 1 | 1 | 1 |
| **Initialize line to hold line from file** | 1 | 1 | 1 |
| **Intialize lineStream to hold line’s contents** | 1 | 1 | 1 |
| **Initialize token to hold word in line** | 1 | 1 | 1 |
| **Open filename with fileStream** | 1 | 1 | 1 |
| **Open filename with fileStream** | 1 | 1 | 1 |
| **Initialize tokenCount to count tokens per line** | 1 | 1 | 1 |
| Get line(s) from fileStream | 1 | n | n |
| Fill lineStream with current | 1 | n | n |
| SET tokenCount to 1 |  |  |  |
| Create Course aCourse for each line | 1 | n | n |
| GET token from lineStream up to ‘,’ until none left | 1 | 2n | 2n |
| IF count == 1 | 1 | n | n |
| SET aCourse’s courseNumber to token | 1 | n | n |
| increment tokenCount | 1 | n | n |
| ELSE IF count == 2 | 1 | n | n |
| SET aCourse’s courseName to token | 1 | n | n |
| increment tokenCount | 1 | n | n |
| ELSE |  |  |  |
| IF token already in bst as course | 1 | n | n |
| add token to aCourse’s PreReqs | 1 | n | n |
| ELSE output file format error | 1 | 1 | 1 |
| IF tokenCount < 2 | 1 | 1 | 1 |
| OUTPUT “Error – each course must have course # and course name.” | 1 | 1 | 1 |
| Add aCourse to bst | n | n | n^2 |
| clear lineStream for next line | 1 | n | n |
| ***Creating course objects*** | | | |
| IF aCourse’s courseNumber < currentNode’s courseNumber | 1 | 1 | 1 |
| IF node’s left child is null | 1 | 1 | 1 |
| add new Node with course at node’s left child | 1 | 1 | 1 |
| ELSE traverse node’s left subtree recursively | 1 | n | n |
| ELSE |  |  |  |
| IF node’s right child is empty | 1 | 1 | 1 |
| add new Node with course at node’s right child | 1 | 1 | 1 |
| ELSE traverse node’s right subtree recursively | 1 | n | n |
| IF root is empty | 1 | 1 | 1 |
| SET root to new node using aCourse | 1 | 1 | 1 |
| ELSE addNode (root, aCourse) | n+3 | 1 | n+3 |
| **Total Cost** | | | n^2 + 17n + 20 |
| **Runtime** | | | O(n^2) |

**Advantages and Disadvantages of Data Structures**

*Vector*

Pros:

* The vector is simple to implement
* It *can* have a search time of O(log(n)) if the vector is sorted
* Cache friendly since it is a contiguous chunk of memory

Cons:

* *Must* be sorted to achieve quality search time mentioned in pros
* Frequently has to shift many elements (items in the vector *after* an insert/delete)
* Has to either take up wasted memory or risk issues with overhead due to reallocation of elements needed as vector grows

*HashTable*

Pros:

* Fast access time
* Can achieve constant time with careful hashing or O(n) at worst

Cons:

* Can take up too much space if size is unknown/can’t plan in advance (loses some efficiency then)
* Have to have method to resolve collisions

*Tree*

Pros:

* Node structure means elements can be retrieved in order easily
* Update data (insert/delete) relatively fast for same reason – not shifting means O(log(n))
* Searches in O(log(n)) as well even if n is huge

Cons:

* Must be balanced tree or performance suffers
* More complicated than other structures and is overkill for few elements
* Shape of tree depends on first item inserted and can cause issues without extra care