Dominic Clapper

June 24, 2022

CS 300

Module 6 Data Structure Analysis Tables/Explanations

**BST Open/Read Runtime**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| BSTree\* bst = new BSTree() | 1 | 1 | 1 |
| string line | 1 | 1 | 1 |
| ifstream inFS  inFS.open(fileToOpen.csv) | 2 | 1 | 2 |
| If !inFS.is\_open():  Put “File could not be opened” to output  Return 0 | 3 | 1 | 3 |
| Else:  Vector<string> fileContents | 2 | 1 | 2 |
| While !inFS.eof():  If !inFS.fail():  getline(inFS, line)  fileContents.push\_back(line)  else:  Put “Error loading data encountered” to output  inFS.close()  break | 7 | n | n |
| inFS.close() | 1 | 1 | 1 |
| Parse(fileContents, allCourseInfo, bst) | 1 | 1 | 4n + 12 |
| **Total Cost** | | | 5n + 22 |
| **Runtime** | | | O(n) |

**BST Parse/Object creation Runtime**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Integer k = 0  string delim = “,” | 2 | 1 | 2 |
| For k = 0; k < fileContents.size(); ++k: | 1 | n | n |
| string tempHolder = fileContents.at(i)  size\_t pos = 0  string token  Course newCourse | 4 | n | n |
| while pos = tempHolder.find(delim) != std::string::npos:  token = tempHolder.substr(0, pos)  newCourse.setCourseNum()  tempHolder.erase(0, pos + delim.length()) | 4 | 1 | 4 |
| while pos = tempHolder.find(delim) != std::string::npos:  token = tempHolder.substr(0, pos) newCourse.setCourseTitle()  tempHolder.erase(0, pos + delim.length()) | 4 | 1 | 4 |
| while pos = tempHolder.find(delim) != std::string::npos:  newCourse.setCoursePrereqs() | 2 | 1 | 2 |
| allCourseInfo.push\_back(newCourse) | 1 | n | n |
| for integer i = 0; I < allCourseInfo.size(); i++ bst.Insert(allCourseInfo.at(i)) | 2 | n | n |
| **Total Cost** | | | 4n + 12 |
| **Runtime** | | | O(n) |

**Vector Open/Read Runtime**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| string line | 1 | 1 | 1 |
| ifstream inFS  inFS.open(fileToOpen.csv) | 2 | 1 | 2 |
| If !inFS.is\_open():  Put “File could not be opened” to output  Return 0 | 3 | 1 | 3 |
| Else:  Vector<string> fileContents | 2 | 1 | 2 |
| While !inFS.eof():  If !inFS.fail():  getline(inFS, line)  fileContents.push\_back(line)  else:  Put “Error loading data encountered” to output  inFS.close()  break | 7 | n | n |
| inFS.close() | 1 | 1 | 1 |
| Parse(fileContents, allCourseInfo) | 1 | 1 | 3n + 12 |
| **Total Cost** | | | 4n + 21 |
| **Runtime** | | | O(n) |

**Vector Parse/Object Creation Runtime**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Integer k = 0  string delim = “,” | 2 | 1 | 2 |
| For k = 0; k < fileContents.size(); ++k: | 1 | n | n |
| string tempHolder = fileContents.at(i)  size\_t pos = 0  string token  Course newCourse | 4 | n | n |
| while pos = tempHolder.find(delim) != std::string::npos:  token = tempHolder.substr(0, pos)  newCourse.setCourseNum()  tempHolder.erase(0, pos + delim.length()) | 4 | 1 | 4 |
| while pos = tempHolder.find(delim) != std::string::npos:  token = tempHolder.substr(0, pos) newCourse.setCourseTitle()  tempHolder.erase(0, pos + delim.length()) | 4 | 1 | 4 |
| while pos = tempHolder.find(delim) != std::string::npos:  newCourse.setCoursePrereqs() | 2 | 1 | 2 |
| allCourseInfo.push\_back(newCourse) | 1 | n | n |
| **Total Cost** | | | 3n + 12 |
| **Runtime** | | | O(n) |

**Hash Table Open/Read Runtime**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| string line | 1 | 1 | 1 |
| ifstream inFS  inFS.open(fileToOpen.csv) | 2 | 1 | 2 |
| If !inFS.is\_open():  Put “File could not be opened” to output  Return 0 | 3 | 1 | 3 |
| Else:  Vector<string> fileContents | 2 | 1 | 2 |
| While !inFS.eof():  If !inFS.fail():  getline(inFS, line)  fileContents.push\_back(line)  else:  Put “Error loading data encountered” to output  inFS.close()  break | 7 | n | n |
| inFS.close() | 1 | 1 | 1 |
| Parse(fileContents, allCourseInfo, table) | 1 | 1 | 4n + 12 |
| **Total Cost** | | | 5n + 21 |
| **Runtime** | | | O(n) |

**Hash Table Parse/Object Creation Runtime**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Integer k = 0  string delim = “,” | 2 | 1 | 2 |
| For k = 0; k < fileContents.size(); ++k: | 1 | n | n |
| string tempHolder = fileContents.at(i)  size\_t pos = 0  string token  Course newCourse | 4 | n | n |
| while pos = tempHolder.find(delim) != std::string::npos:  token = tempHolder.substr(0, pos)  newCourse.setCourseNum()  tempHolder.erase(0, pos + delim.length()) | 4 | 1 | 4 |
| while pos = tempHolder.find(delim) != std::string::npos:  token = tempHolder.substr(0, pos) newCourse.setCourseTitle()  tempHolder.erase(0, pos + delim.length()) | 4 | 1 | 4 |
| while pos = tempHolder.find(delim) != std::string::npos:  newCourse.setCoursePrereqs() | 2 | 1 | 2 |
| allCourseInfo.push\_back(newCourse) | 1 | n | n |
| for integer i = 0; I < allCourseInfo.size(); i++ table.HashTableInsert(allCourseInfo.at(i)) | 2 | n | n |
| **Total Cost** | | | 4n + 12 |
| **Runtime** | | | O(n) |

**Data Structure Evaluation**

Throughout this class, I have learned an insane amount about different types of data structures. Each data structure has its pros and cons, which make them all very unique. Depending on what you want your code to do, and the scope of which you want it to work for, certain data structures are better than others. The data structures utilized/mentioned above in the pseudocode are vectors, hash tables, and trees. All three data structures allow for data to be stored and accessed. There are also many other operations that can be performed on them.

In terms of pros and cons, vectors can be very useful given the correct situation. Vectors are capable of storing many objects, and they can be expanded automatically as well if need be. Vectors are very easy to understand when it comes to how they work and store info. They are also easy to understand in regard to the operations that can be performed on them. One major problem with vectors occurs when inserting objects/items in certain parts of the vector other than the end, such as the middle of the vector. Basically, the problem is that when you add an item to the middle of a vector, everything after the insertion position must be shifted over a spot in the vector, and in the memory. This can affect the runtime of insertion by making it longer. However, if you are adding to the end of a vector, the runtime will not be affected much.

As for hash tables, they also have many pros and cons that make them very situational like many other data structures. Compared to vectors, hash tables are a bit more difficult to understand, and, in my opinion, are the hardest to comprehend of the three mentioned in the pseudocode. Despite this, they are still very useful. For one, hash tables are resizable like vectors. Hash tables are pretty fast when it comes to searching through hash tables thanks to the use of hash function searching. They are also very fast when it comes to inserting and deleting items/objects due to the use of hash functions.

Trees are honestly quite different from both vectors and hash tables. Even though they are majorly different, trees are still very useful for storing, searching, and inserting data. Trees are a bit easier to understand when compared to hash tables, but they are definitely more complicated than vectors. Like I mentioned, trees are very good for storing, searching, and inserting data. Trees are faster in terms of runtime when compared to vectors, but they can be a bit slower than hash tables if the hash table is built correctly. Trees are pretty efficient in regard to traversing the tree, which can be done in multiple different ways such as in order or post order.

Out of the three data structures analyzed/mentioned in this project, I feel that a binary search tree would work the best for the program, as well as for me as the programmer. The main reason I feel this way is because trees are very efficient when searching, inserting, removing, and traversing. Even though hash tables can be faster in terms of runtime, I feel that using hash tables would result in some problems when it came to sorting the course list. (Hash tables would be the fastest of the three data structures, but I am not as comfortable with them) Vectors are out of the running too because the runtimes involved with searching and inserting in certain spots can get lengthy. Also, the other reason I feel trees would work the best is because I feel most comfortable working with trees. When I learned about them, they were easy for me to understand. Maybe not as easy as vectors, but much simpler than hash tables involving chaining.