Quintin B. Rozelle

4/1/2023

CS-300

# **Runtime and memory analysis**

## **Vector**

### **Create data structure**

Note: Only the portion of the code that builds the data structure is analyzed.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| WHILE inFS is good: | 1 | n | n |
| (code that parses input but doesn’t directly build data structure) | N/A | N/A | N/A |
| PUSHBACK NEW Course(courseID, courseName, coursePreReqs) to courses | 1 | n | n |
| **Total Cost** | 2n | | |
| **Runtime** | O(n) | | |

Memory Analysis: Since this algorithm essentially just adds items to the end of a vector, it’s overall memory requirement is just n.

### **PrintAll function**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| FOR (start: i = 0; while: i < size of courses; i++): | 1 | n | n |
| SET j to i | 1 | n | n |
| WHILE (j > 0 AND courses.at(j).courseNumber < courses.at(j – 1).courseNumber): | 1 | n^2 | n^2 |
| SET temp to courses.at(j) | 1 | n^2 | n^2 |
| SET courses.at(j) to courses.at(j – 1) | 1 | n^2 | n^2 |
| SET courses.at(j – 1) to temp | 1 | n^2 | n^2 |
| DECREMENT j | 1 | n^2 | n^2 |
| FOR (start: i = 0; while: i < size of courses; i++): | 1 | n | n |
| OUTPUT “Course Number: “ courses.at(i).courseNumber | 1 | n | n |
| OUTPUT “Course Name: “ courses.at(i).courseName | 1 | n | n |
| FOR (start: j = 0; while: j < size of courses’ preReqs; j++): | 1 | n^2 | n^2 |
| OUTPUT “Pre reqs: “ courses.at(i).preReqs.at(j) | 1 | n^2 | n^2 |
| **Total Cost** | 7n^2 + 5n | | |
| **Runtime** | O(n^2) | | |

Memory Analysis: One vector of variable length (the vector passed into the printAll function) and three variables of static length (i.e., i, j, and temp) are used to print this data structure. The memory requirements for this algorithm will be n+3, or n.

Note: the “static” variable named temp used to store courses while sorting the course vector isn’t technically static since it varies based on the data within the course object (e.g., courseID, courseName, etc), but when compared to the memory requirements of the vector, it is significantly smaller and can be thought of as static.

## **Hash Table**

### **Create data structure**

Note: Only the portion of the code that builds the data structure is analyzed.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| WHILE inFS is good: | 1 | n | n |
| (code that parses input but doesn’t directly build data structure) | N/A | N/A | N/A |
| courses.**Insert**(NEW Course(courseID, courseName, coursePreReqs)) | 6 | n | 6n |
| Function Total | See below | | |
|  |  |  |  |
| **HashTable::Insert Function** |  |  |  |
| CREATE new int named key and SET to **hash**(course) | 1 | 1 | 1 |
| CREATE new LinkedList pointer named bucket and SET to address of nodes.at(key) | 1 | 1 | 1 |
| bucket->**Append**(course, key) | 4 | 1 | 4 |
| Function Subtotal | 6 | | |
|  |  | | |
| **HashTable::Hash Function** |  |  |  |
| RETURN substring of course’s courseNumber from [4, end) mod 10 | 1 | 1 | 1 |
| Function Subtotal | 1 | | |
|  |  |  |  |
| **HashTable::LinkedList::Append Function** |  |  |  |
| CREATE new node pointer named newNode and SET to NEW Node(course, key) | 1 | 1 | 1 |
| IF (head equals null pointer), THEN: | 1 | 1 | 1 |
| SET head to newNode | 1 | 1 | 1 |
| SET tail to newNode | 1 | 1 | 1 |
| ELSE: | 1 | 1 | 1 |
| SET tail->next to newNode | 1 | 1 | 1 |
| SET tail to newNode | 1 | 1 | 1 |
| Function Subtotal | 4  (both branches of IF statement have equal cost and only one will ever be entered) | | |
|  |  |  |  |
| **Total Cost** | 7n | | |
| **Runtime** | O(n) | | |

Memory Analysis: The hash table is built with 10 buckets and at worse, the largest bucket would have n courses assigned to it while the rest just contain null pointers. Thus, the memory requirements for this algorithm would be n+9, or n.

### **PrintAll function**

Note: This algorithm utilizes the same insertion sort algorithm to sort that was analyzed with the vector printAll function. To save space, it is listed as “insertion sort” and the runtime cost of the sorting algorithm is used instead.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| FOR (start: i = 0; while: i < nodes.size(); i++): | 1 | n | n |
| nodes.at(i).addToVector(tempVec) | 1 | n | n |
| **Insertion sort algorithm** | 5n^2 + 2n | 1 | 5n^2 + 2n |
| FOR (start: i = 0; while: i < size of courses; i++): | 1 | n | n |
| OUTPUT “Course Number: “ courses.at(i).courseNumber | 1 | n | n |
| OUTPUT “Course Name: “ courses.at(i).courseName | 1 | n | n |
| FOR (start: j = 0; while: j < size of courses’ preReqs; j++): | 1 | n^2 | n^2 |
| OUTPUT “Pre reqs: “ courses.at(i).preReqs.at(j) | 1 | n^2 | n^2 |
| **Total Cost** | 7n^2 + 7n | | |
| **Runtime** | O(n^2) | | |

Memory Analysis: Two vectors of variable length (the vector passed into the printAll function and tempVec) and three variables of static length (i.e., i, j, and temp) are used to print this data structure. The memory requirements for this algorithm will be 2n+3, or n.

The same note regarding “static” length variables given above applies here as well.

## **Binary Search Tree**

### **Create data structure**

Note: Only the portion of the code that builds the data structure is analyzed.

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| WHILE inFS is good: | 1 | n | n |
| (code that parses input but doesn’t directly build data structure) | N/A | N/A | N/A |
| courses.**Insert**(NEW Course(courseID, courseName, coursePreReqs)) | 4 log n + 4 | n | 4n log n + 4n |
| Function Total | See below | | |
|  |  |  |  |
| **BinarySearchTree::Insert Function** |  |  |  |
| IF (root is null pointer), THEN: | 1 | 1 | 1 |
| SET root to new Node(course) | 1 | 1 | 1 |
| ELSE: | 1 | 1 | 1 |
| SET current to pointer for root | 1 | 1 | 1 |
| WHILE (current is not a null pointer): | 1 | log n | log n |
| IF (course’s courseNumber is less than current’s courseNumber), THEN: | 1 | log n | log n |
| IF (current’s left equals null pointer), THEN: | 1 | log n | log n |
| SET current's left pointer to new Node(course) | 1 | log n | log n |
| SET current to null pointer | 1 | log n | log n |
| ELSE: | 1 | log n | log n |
| SET current to current’s left pointer | 1 | log n | log n |
| ELSE: | 1 | log n | log n |
| IF (current’s right equals null pointer), THEN: | 1 | log n | log n |
| SET current's right pointer to new Node(course) | 1 | log n | log n |
| SET current to null pointer | 1 | log n | log n |
| ELSE: | 1 | log n | log n |
| SET current to current’s right pointer | 1 | log n | log n |
| Function Subtotal | 4 log n + 4  (Most expensive IF statement branches chosen for worst case scenario) | | |
|  |  |  |  |
| **Total Cost** | 4n log n + 4n | | |
| **Runtime** | O(n log n) | | |

Memory Analysis: This algorithm does nothing more than create nodes and add them to the binary search tree with not significant intermediate steps. Therefore, its memory requirement is n.

### **PrintAll function**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| PrintAll(node’s left) | 1 | ½ n | ½ n |
| OUTPUT node’s information | 1 | 1 | 1 |
| PrintAll(node’s right) | 1 | ½ n | ½ n |
| **Total Cost** | n + 1 | | |
| **Runtime** | O(n) | | |

Memory Analysis: This algorithm uses minimum extra variables. It does use one int while cycling through the preReqs for a course so its memory requirement is 1.