Amin Parvizi
Behrad Mousavi
Nicholas lozano
Huyang Yu
Runtime Error
05/17/2021

Project 2 BigO Analysis

1. Data Structures

A) Doubly Linked List – This doubly linked list structure is the basis for both the List and Stack classes which are both simply wrappers for the underlying doubly linked list

Add: O(1) Adding to the list is done by inserting an element before the node specified by a pointer which is passed in, which lets addition to the list run in constant time. However, finding that pointer might take additional run time outside of the add method. Two methods named AddFront and AddBack allow for the use of pre-defined pointers if you want to add to the front or back of the list. This means they will always run in constant time.

Remove: O(1) This method follows the same structure as add. Passing in a pointer lets it run in constant time, but it will likely require a search of the list to get the pointer. DeleteFront and DeleteBack use predefined pointers and will always run in constant time.

B) List : This is created by wrapping the doubly linked list described above.

Insert: O(1) Because this is a wrapper it is identical to the DLL above. If you wish to insert to the middle of the list you will have to search for an iterator to do so, which will eat additional time. The insertion itself though runs in constant time. InsertFront and InsertBack will always run in constant time.

Remove: O(1)/O(n) If you already have an iterator to the element you want to delete it runs in constant time. If you only have the element, then a search is performed to get an iterator which pushes the time up to O(n).

Search: O(n) The search on the list is a simple linear search. There is no way to optimize the search because the list is unordered. The unordered nature of the list means every element must be checked giving it a run time of O(n).

C) Stack – Another wrapper for the doubly linked list.

Push: O(1) Because insertions can only be run at the front of the stack, and because we always have a reference to the front of the stack, all insertions run in constant time.

Pop: O(1) All removals must happen at the front of the stack. We always have a reference to the front of the stack therefore all removals run in constant time.

Search: O(n) Searching for an element in the stack must be done linearly, ensuring every element is checked, and thus runs in O(n) time.

D) Map – This is a wrapper of a double-hashing hash table.

Put: Best Case O(1), Worst Case O(n) A double hashing formula is used to hash the key into an index for the bucket array. If the first attempt finds an open space then it will run in O(1). The more times the key must be hashed to find an open space the slower it runs, up to a run time of O(n) where every index in the bucket array is checked.

Erase: Best Case O(1), Worst Case O(n) Deletion itself will take constant time. First however a reference to the index in the bucket array must be found. Similar to insertion, if the first hashing gets us the right index then this will run in constant time. The more hashes it takes the closer to linear time we will get.

Find: Best Case O(1), Worst Case O(n) This is the equivalent of a search function. It will run in near constant time almost always. The more collisions the given key has the longer it will take to hash and the closer to linear time this method will run in.

F) Tree – This is a generic tree. It is unordered, does not enforce balance, and each node has no limit on its number of children. It was an ad hoc structure to support the graph and doesn't have a removal function because the graph did not demand one.

Add: O(1) Similar to the doubly linked list above, this function demands that a pointer to the parent node of your insertion already exist which lets the insertion method itself run in constant time. Finding that pointer may take additional run time however, especially if a search of the tree is needed to get it.

Search: O(n) The search is done by recursively searching down the list which means you need to check every node until you find your search key. This gives a linear time search.

G) Graph

AddVertex: O(n^2) Creating the new vertex object and inserting it into the graph's list of vertices take constant time. What pushes the run time of this method up is that the adjacency matrix the graph is built on must be reconstructed. This uses nested loops that each run n times which is why this runs in quadratic time.

AddEdge: O(1) Insertion into the graph's edge list runs in constant time because the edge is inserted to the back. Insertion into the adjacency matrix runs in constant time as well. If you must search the graph for a reference to the edge's incident vertices before calling this function then that will push total run time up.

EraseVertex: O(n^2) The erasure itself runs in O(n) time, since it must ensure that the vertex to be deleted is in the graph through a search and traverse the entire length of the adjacency matrix deleting edges to vertex. Resizing the adjacency matrix after the deletion takes quadratic time though which is what pushes the time here up.

FindShortestPath: O(n * m^2) Many of the loops in this method are dependent on the number of vertices you are trying to find a path to. While there are a lot of loops, not very many of them are nested which keeps run time relatively in line. Since m <= n by definition, this function will have a worst case run time of roughly O(n^3).

2) Trip Planner

Adding a Predefined List of Stadiums: O(n * m)
The user can choose to add a predefined list of stadiums to their trip (ie all stadiums with a team in the NFC). This adds all stadiums in one of the program's stadium lists which are not already in the trip to the list of stadiums to visit during the trip. Every stadium in the list of stadiums to add must be iterated, and for each stadium to add we must check each stadium already in the trip to determine if the stadium we are trying to add already exists in the trip. Accessing the stadium is done with a hash table lookup which has an expected run time of O(1) but which can take up to O(n). Insertion into the trip list is O(1) since it is inserted in the back of the list.
Buying a Souvenir from a Stadium: Best Case O(n), Worst Case O(n + m) A list of souvenirs must be iterated which gives the O(n) base time. Purchases are stored in a list but insertion to the list takes O(1) time because new orders are inserted to the back. These lists are themselves stored in a map. Accessing this map has best case O(1) and worst case O(m) time. The limiting factor on run time for this operation is the FindShortestPath function in the graph, which runs in O(n*m^2) time.
Adding a new stadium requires inserting that stadium into the proper location of all relevant stadium lists. This takes O(n) since a search must be performed on the lists to find the right place to insert the new stadium.

Changing a Stadium: O(n), Changing a stadium might require changing which lists the stadium is in and resorting its location in the lists, which will take O(n) time since searches must be performed on these lists.

Removing a Stadium: O(n^2) Removal requires searching through each data structure (lists, the map, and the graph) and removing the stadium object. The worst of these is the graph with a removal time of O(n^2).

Add a Team: O(m) The team object is inserted in alphabetical order into the stadiums list of teams which requires a search of the list.

Change a Team: O(n + m) Changing a team might require a resort of both the teams list (if the name changes the alphabetical order) and the stadium lists (if a change in conference changes

which lists the stadiums should be in). Each of these would require a search of their respective lists.