Computer Science Basics: Data Structures

Behnam Nikkhah

Contents

Data Structure to Search for Food Types

Data Structure used

Trie

I used a Trie data structure to search for the food types because it is similar to a search engine. On *Google*, for example, when we type the *prefix* of what we're searching for, a number of suggestions appear below the text field to help with additional options or quickly retrieve the desired string that we want. This functionality is made possible with a Trie data structure, which is very fast in retrieving data with respect to the prefix of a string.

Run-time of a Trie Data Structure - Inserting

Case	**Run-time**
Worst	O(*n*)
Average	O(*n*)
Best	O(*1*)

Run-time of Insertion - Explanation

Inserting:

- **Worst Case:** Assuming an empty Trie and we want to insert a key *n*, which has a length greater than 1, we must iterate through each character of the key, which is *n* number of iterations. **O(*n*)**
- **Best Case:** Assuming an empty Trie and we want to insert a key with a length equal to 1, we only have to iterate one item, which is only 1 iteration. **O(*1*)**

Run-time of a Trie Data Structure - Searching

Case	**Run-time**
Worst	O(*n*)
Average	O(*n*)
Best	O(*1*)

Run-time of Searching - Explanation

Searching:

- **Worst Case:** Assuming a prefix of size *n*, which has a length greater than 1 and the prefix is found in the Trie, the iteration will take *n* amount of iterations depending on the size of the prefix. **O(*n*)**
- **Best Case:**
  - Assuming a prefix of length equal to 1 and the prefix is found in the Trie, this equals to only 1 iteration. **O(*1*)**
  - Assuming the first character of the prefix is not found in the Trie, there will be no iterations and equals to a prompt termination of the algorithm. **O(*1*)**

Answer to Food Type searching

From this analysis, we can conclude that the runtime for *searching* for a food type is **O(n)**.

There cannot be a more efficient run-time because we have to go through each character in order to insert or search within a data structure. A Trie data structure was used due to its efficiency in storing each character of a *word* when attempting to search for a prefix.

Data Structure to Retrieve Restaurant Data

Data Structure used

Linked List stored in a Hash Map

I used a Linked List stored in a Hash Map because a hash map can store any data type to a *key* for quick retrieval. In addition, a Linked List is used to store each restaurant data since we are simply *displaying* the information on the front-end and since the restaurant data, for this particular project is not variable (i.e. static), a linked list is suffice to displaying the data.

Run-time of a Hash Map - Inserting

Case	Run-time
Worst	O(*n*)
Average	O(*n*)
Best	O(*1*)

Run-time of Hash Table Insertion - Explanation

Inserting:

- **Worst Case:** Assuming a collision exists in the hash table, we then proceed with a hash collision resolution algorithm such as *Open Addressing* or *Separate Chaining*. Both resolutions will iterate through the hash table until an open slot is found, which iterates *n* number of times. **O(*n*)**
- **Best Case:** Assuming no collisions exists in the hash table, an open slot is immediately available for the *key* and is promptly placed in the hash table. **O(*1*)**

Run-time of a Linked List - Inserting

Case	**Run-time**
Worst	O(*n*)
Average	O(*n*)
Best	O(*1*)

Run-time of Linked List *Insertion* - Explanation

Inserting:

- **Worst Case:** Assuming we want to insert a value at the *end* of a Linked List (i.e. **append**), we have to traverse *n* amount of nodes until we reach a null pointer. **O(*n*)**
- **Best Case:** Assuming we want to insert a value to the *head* of a Linked List (i.e. **prepend**), the new node will point to the current head node and the new head node is now the new value we wanted to insert. **O(*1*)**

Answer to Restaurant Data retrieval

From this analysis, we can conclude that the runtime for *retrieving* restaurant data is **O(1)**. Since hash collisions are rare and the restaurant data is not substantial (i.e. more than a million entries), then it is a simple insertion for the hash table without resorting to a hash collision resolution.

This is the most efficient run-time and mainly the reason why storing data into a hash table is always a good idea due to its efficient retrieval.

Other Innovative ways to utilize Data Structures

Although not covered in this course, a **Red-Black Tree** data structure is a popular choice of many implementations because of its *self-balancing* property. A great advantage to using a Red-Black tree over a Hash Table is that it can traverse the tree efficiently in *sort order*. Therefore, a Red-Black Tree data structure is an optimal choice when implementing a dictionary of words, which can contain a large amount of entries and a dictionary is usually in sort order.