Assignment 4: Searching lists

CS 301

March 24, 2020

We've seen that searching for an item in a list generally takes linear time, but searching for an item in a dictionary generally takes constant time. In this assignment, we look at ways to search a list quickly, and try to explain how dictionaries are able to find items so quickly. There are related sections in the textbook. My recommendation is to try to write the code yourself, and then compare your answers to the textbook. However, if you are feeling stuck, you can always look at the relevant parts of the textbook. The functions that I am asking you to write are similar but not identical to things discussed in the textbook, so either way your code will look somewhat different.

1. We know that searching lists usually takes linear time, but what if the list is sorted? In class, we discussed how we could find an item in a sorted list in $\log(n)$ time using a binary search. Implement this idea in a python function `search_sorted_list(sorted_list,item)` that determines if `item` is in `sorted_list` in $\log(n)$ time using a binary search algorithm. Your function should be recursive, and should keep passing the whole list to itself (with other appropriate information), since if you keep taking slices, you add the time of copying the list slices and end up with an $O(n)$ algorithm.

2. Even if our list is sorted, this still doesn't give us the constant time searching behavior that we see in dictionaries. To achieve this, we use a *Hash Table*. Here's how it works:

   - First, we create an empty list that is much larger than the number of things we expect to put in it. We refer to each spot in the list as a slot.
   - We create a *hash function* that tells us which slot any input of a given type should go into. For example, if our inputs are numbers and our list is of length l, we could create a function that says that the number n should go into slot s, where s is the remainder when n is divided by l.
   - Each new item goes into the slot given by the hash function. But what if the slot is already full? This is called a *collision*. There are various ways we could deal with this. One of the easiest is just to put the item in the next empty slot. This is known as "rehashing by linear probing."
   - We look up an item using the same methods we used to decide where to put it in the first place. First we look for it where the hash function says it should go, then if there is something else there, we need to keep looking for it until we either find it or we find an open slot that means it isn't there.

Create a `HashList` class in python that implements these ideas. You can assume that inputs will be integers. It should contain the following methods:

HashList(length) creates a new empty HashList of the given length.

hashfunction(item) tells you which slot the item is assigned to.

put(item) adds the given item to the list. If the list is full, it throws an error.

contains(item) returns True if the given item is in the list, and False otherwise. Make sure your method still works in the extreme case in which the list is entirely full and the given item isn't in the list.

items() returns a list of all items in the HashList.

You may find it helpful to look at sections 6.5–6.5.2 in the book for a description of the related data type of a HashTable. You should try to write your own code for the HashList before looking at section 6.5.3, which contains code for a HashTable.

3. In a comment, explain the running times of the HashList methods in the best-case scenario in which there are no collisions, and in the worst-case scenario in which there are many collisions.

4. Also explain how we would have to modify things to convert our HashList into a dictionary.