Searching

Vivek K. S., Deepak G.

Information Systems Decision Sciences (ISDS)
MUMA College of Business
University of South Florida
Tampa, Florida

2017

Searching

Searching is the process of finding a particular item of interest among a collection of items.

- The result of the operation is True or False indicating the presence of the item.
- Lets consider the following example. Note that we are using a list and the membership check (using "in") here:

```
15 in [3,5,2,4,1]
False
3 in [3,5,2,4,1]
True
```

- Though it looks simple on the outside, there is an underlying mechanism that we are interested in.
- We wish to compare the different techniques in terms of their implementation and performance.

Sequential Search

- When data items are stored in a sequential manner, we say they have a linear or sequential relationship.
- They are arranged sequentially in relation to one another and we call this index positions.
- Since these indexes are ordered, it is possible for us to visit the item sequentially in search of our item.
- The simplest searching approach, "The Sequential Search" involves using the underlying sequential ordering to access each item.

Python Implementation

The Python implementation of this algorithm is as follows:

```
def sequential_search(input_list, item):
    pos = 0
    found = False
    while pos < len(input_list) and not found:
        if input_list[pos] == item:
            found = True
        else:
            pos = pos+1
    return found
```

Analysis of Sequential Search

- To analyze this search algorithm, we will consider the number of comparison operation done as our unit of computation.
- The sequential search is straightforward. There are four possible scenarios that we could think of.
- When the item is at the beginning of the list, the search is a O(1) operation.
- In the case that the item is in the end of the list or not present at all (we will need to traverse the entire list to be sure) it become a O(n) operation.
- In the average case, when the item is in the list somewhere else in the middle, it becomes a O(n/2) operation.
- In general, as the number of items increase, the coefficients become negligible, that the algorithm become O(n) which is not good.

Ways to Improve Performance

- To improve the performance of the sequential search, we could order the list.
- But ordering can only help so much.
- In the even that the item is at the end of the list, the search is still O(n).
- Ordering can only help in cases where the item is not present and the ordering gives us clues to drop the search immediately when an item larger than the item being searched for is encountered.

Ways to Improve Performance

- To improve the performance of the sequential search, we could order the list.
- But ordering can only help so much.
- In the even that the item is at the end of the list, the search is still O(n).
- Ordering can only help in cases where the item is not present and the ordering gives us clues to drop the search immediately when an item larger than the item being searched for is encountered.

The Binary Search

Binary search is a very clever variation of ordered lists.

- The binary search starts from the middle of the list. If it is the item we are looking for, we are done.
- If not, we split the list into two based on the comparison between the middle item and the item we are looking for.
- This is repeated over and over again until we zero-in on our item.
- This is a great example of the divide and conquer strategy.

Python Implementation

```
def binary_search(input_list, item):
    first = 0
    last = len(input_list) - 1
    found = False
    while first <= last and not found:
        midpoint = (first + last) // 2
        if input_list[midpoint] == item:
            found = True
        elif item < input_list[midpoint]:
                last = midpoint - 1
        else:
            first = midpoint + 1
    return found
```

Python Implementation Using Recursion

The same could be implemented using recursion as follows:

```
def rbs(input_list, item):
    if len(input_list) == 0:
        return False
    else:
        mp = len(input_list) // 2
        if input_list[mp] == item:
            return True
        elif item < input_list[mp]:
            return rbs(input_list[:mp], item)
        else:
            return rbs(input_list[mp + 1:], item)
```

Analysis of The Binary Search

- In Binary search, with each midpoint comparison, we eliminate half of the list and that saves a lot of computational requirements.
- In mathematical terms, If we start with n items, about n/2 items will be left after the first comparison. After the second comparison, there will be about n/4 left. Then n/8, n/16 , and so on.
- When we split the list to the absolute minimum, that is down to one item (which is the worst case here), we either end up finding the item or not.
- The number of comparisons required to get to this point is "i" where n/2**i = 1. Solving for i gives us i = log n.
- The maximum number of comparisons is logarithmic with respect to the number of items in the list. Therefore, the binary search is O(log n).

Hashing

Hashing is an improvement over the search philosophy based on knowing the relative positioning of the items through the index positions.

- In Hashing, we attempt to achieve search in O(1) time.
- In order to achieve this, it is imperative that we know the location of each unique item.
- A hash table is a collection of items which are stored in such a way as to make it easy to find them later.
- Each position in a hash table is called a slot.
- The mapping between the items and the slots is called the hash function.
- The hash function takes an item and returns an integer in the range of the slot names (which are numbers). There are many different types of hash functions.

Remainder Method

- The remainder method take an item and divides it by the table size and uses the remainder as the hash value.
- Once the hash values are calculated, the items are inserted into the hash table at the designated position.
- Now, when we want to search for an item, we compute the hash function, determine the slot and finally check the slot to see if the item is present there.
- This operation is O(1).
- Since a constant time is required to compute the hash value and then index the hash table for the item, it is a constant time search algorithm.
- The problem with this algorithm is that the same values and items that hash out to the same hash values will create a collision problem.

Perfect Hash Function

- Given a collection of items, a hash function that maps each item into a unique slot is referred to as a perfect hash function. But this is an ideal situation.
- Unfortunately, for an arbitrary set of items, there is no systematic way to construct a perfect hash function.
- We need a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the items in the hash table.
- Two methods that are commonly used for constructing hash functions.
  - Folding method.
  - Mid-square method.

Folding Method

- In this method, the item is split into equal-sized pieces(the last piece may not be of the same size).
- The pieces are then added together to give the resulting hash function.
- For example, it it is a phone number 210-564-1241. We would divide the number into pieces of 2 numbers each as follows: (21,05,64,12,41)
- We would later add them up as $21+05+64+12+41 = 143$.
- If the size of the hash table is less than that, we would need to perform another operation.
- Imagining our hash table has only 11 slots, we would divide the sum by 11 to find the remainder as 143
- Hence, the item gets stored in the slot named '0'.
- In some folding methods, we might also reverse the pieces before adding them all together.

Mid-square Method

- In this method, we square the item and extract a portion of it.
- For example, if our item was the number 47, we would square it which results in 2209.
- By extracting the two middle digits, 20, and performing the remainder step 20
- We can also create hash functions for strings by considering strings as a sequence of characters and considering the ascii equivalent of the characters.

Collision Resolution

Collision resolution is the systematic way of placing the second item that has the same hash value as another item in the hash table.

- There are two approaches to do it. One method is to push the second colliding item to the next open slot that is available in the table.
- This technique is called open addressing. It is done by what is commonly referred to as "linear probing".
- Once, a hash table is constructed using linear probing, the same methods should be used for searching as well.
- The disadvantage of linear probing is the tendency for clustering. If a lot of collisions occur, a large number of surrounding slots will be filled by linear probing resolution.

Collision Resolution Continued

- Another solution to this is to use "Quadratic Probing".
- In this technique, instead of going to the next open slot, we skip slots (by a definite uniform count) sequentially, thereby distributing the items that caused collisions more evenly.
- The usual norm is "plus 3" probing. It is important such that the skip must be such that all the slots will be visited.
- Otherwise, part of the table will be unused. To ensure that, we choose the size of the table to be a prime number.
- Another alternative is to use "Chaining" where a single slot could hold references to multiple items that all hashed out to that slot number.

Map Abstract Data Type

- One of the most useful Python collections is the dictionary.
- A dictionary is an associative data type composed of key-data pairs. We refer to this ideas as "Map".
- By definition, a Map is an unordered collection of associations between keys and data values.
- The keys are all unique and they share a one-to-one relationship with the data values.
- This helps in easy lookup of data if the key is given/known.
- The fast lookup is efficiently implemented using the hash table concepts we discussed earlier as it enables O(1) performance.

Operations of a Map

- Map() - Creates a new, empty map. It returns an empty map collection.
- put(key,val) - Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.
- get(key) - Given a key, return the value stored in the map or None otherwise.
- del - Delete the key-value pair from the map using a statement of the form del map[key].
- len() - Return the number of key-value pairs stored in the map.
- in - Return True for a statement of the form key in map, if the given key is in the map, False otherwise.

Implementation of Map Using Hash Table

- We can create a Map data type by using the list data type and a Hash Table class implementation.
- We will need two lists. One list to hold the slots and another to hold the data at corresponding slot positions.
- It is important that the size of the hash table has to be a prime number so that collision resolution algorithms can be efficiently implemented.
- We will use the simple remainder method for our hash function.
- The collision resolution will use linear probing with plus 1 rehash.

Summary

- We learned the importance of searching in Computer Science.
- Sequential search is the simplest way to search for an item in a collection by sequentially visiting every item in the collection.
- The Binary search offers a great performance over sequential search by splitting the collection into sub-collections for ease and efficiency in searching.
- The Binary search can also be done recursively thereby making the algorithm easier to implement.