Data Structures and Algorithms

Lesson 9. Searching

Alexandra Ciobotaru

14 apr 2022

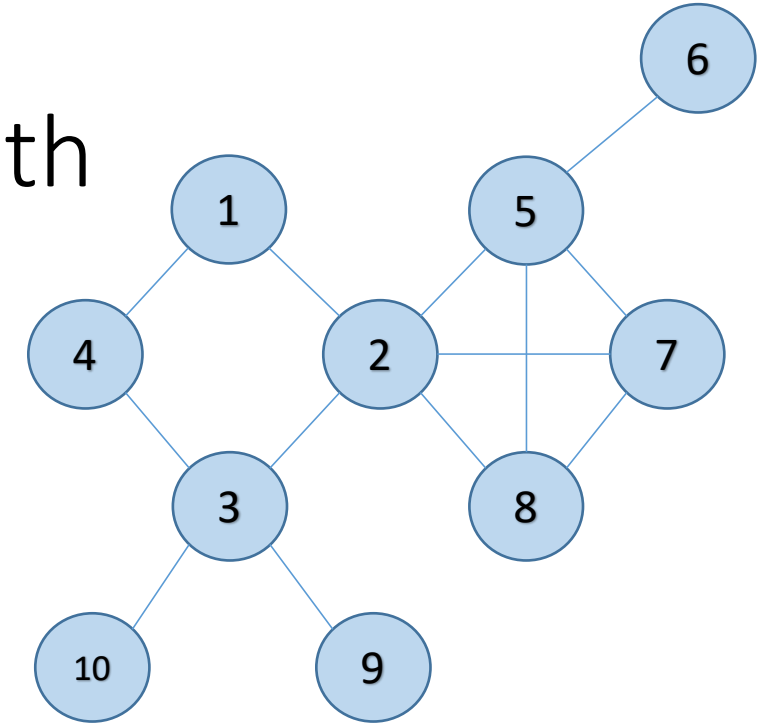
Syllabus

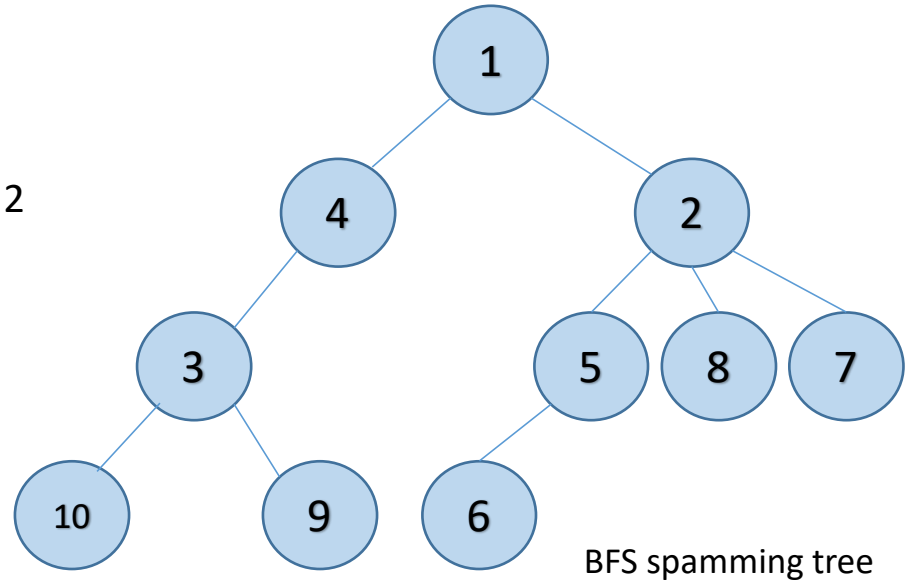
- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- **Lesson 3. Principles of Algorithm Design.**
- **Lesson 4. Lists and Pointer Structures.**
- **Lesson 5. Stacks and Queues.**
- **Lesson 6. Trees.**
- **Lesson 7. Hashing and Symbol Tables.**
- **Lesson 8. Graphs.**
- **Lesson 9. Searching.**
- Lesson 10. Sorting. Mid-term evaluation quiz (30%)
- Lesson 11. Selection Algorithms.
- Lesson 12. String Algorithms and Techniques.
- Lesson 13. Design Techniques and Strategies.
- Lesson 14. Implementations, Applications and Tools.
- Lab (20%) + Project (50%).

Last lesson we discussed BFS in-depth

- For traversing this complex graph we'll use a queue Q: | | | | | | | |.
- We start exploration from any vertex. I'll select vertex 1: Q: |1| | | | | | |. BFS: 1
- I take out from Q the vertex 1 and start exploring it – first I visit 4, so I add it to result and I add it to Q: |~~1~~|4| | | | | |. BFS: 1, 4
- Next I visit 2, so I add it to result and I add it to Q. Now 1 is completely explored. Q: |~~1~~|4|2| | | | |. BFS: 1, 4, 2
- Now I select the next node from exploration from Q, that is 4, and I start exploring 4. I visit 3, and I add it to the result and I add it to Q: |~~1~~|4|2|3| | | |. BFS: 1, 4, 2, 3
- Now I select in Q the next node for exploration, which is 2. It's adjacent vertices are 3, which is already explored, 5, 7, 8. I select 5, add it to result and add it to Q. Q: |~~1~~|4|~~2~~|3|5| | |. BFS: 1, 4, 2, 3, 5
- Next I go to 8: Q: |~~1~~|~~4~~| ~~2~~|3 | 5|8 | | |. BFS: 1, 4, 2, 3, 5, 8
- Next I go to 7: Q: |~~1~~|~~4~~| ~~2~~|3 | 5|8 |7 | | |. BFS: 1, 4, 2, 3, 5, 8, 7
- Now I select from Q the next vertex for exploration, 3, which has the adjacent nodes: 2 (was visited), 9, 10. I select 10: Q: |~~1~~|~~4~~| ~~2~~|~~3~~| 5|8|7|10| |. BFS: 1, 4, 2, 3, 5, 8, 7, 10
- Next I go to 9: Q: |~~1~~|~~4~~| ~~2~~|~~3~~| 5|8 |7 | | |. BFS: 1, 4, 2, 3, 5, 8, 7, 10, 9
- Now I select from Q the next vertex for exploration, 5, which has the neighbors: 2 , 8, 7, 6. I visit 6: Q: |~~1~~|4|~~2~~|~~3~~|~~5~~|8|7|6|. BFS: 1, 4, 2, 3, 5, 8, 7, 10, 9, 6
- All the other vertices were explored.

Source and DFS in-depth: https://youtu.be/pcKY4hjDrxk





BFS spamming tree

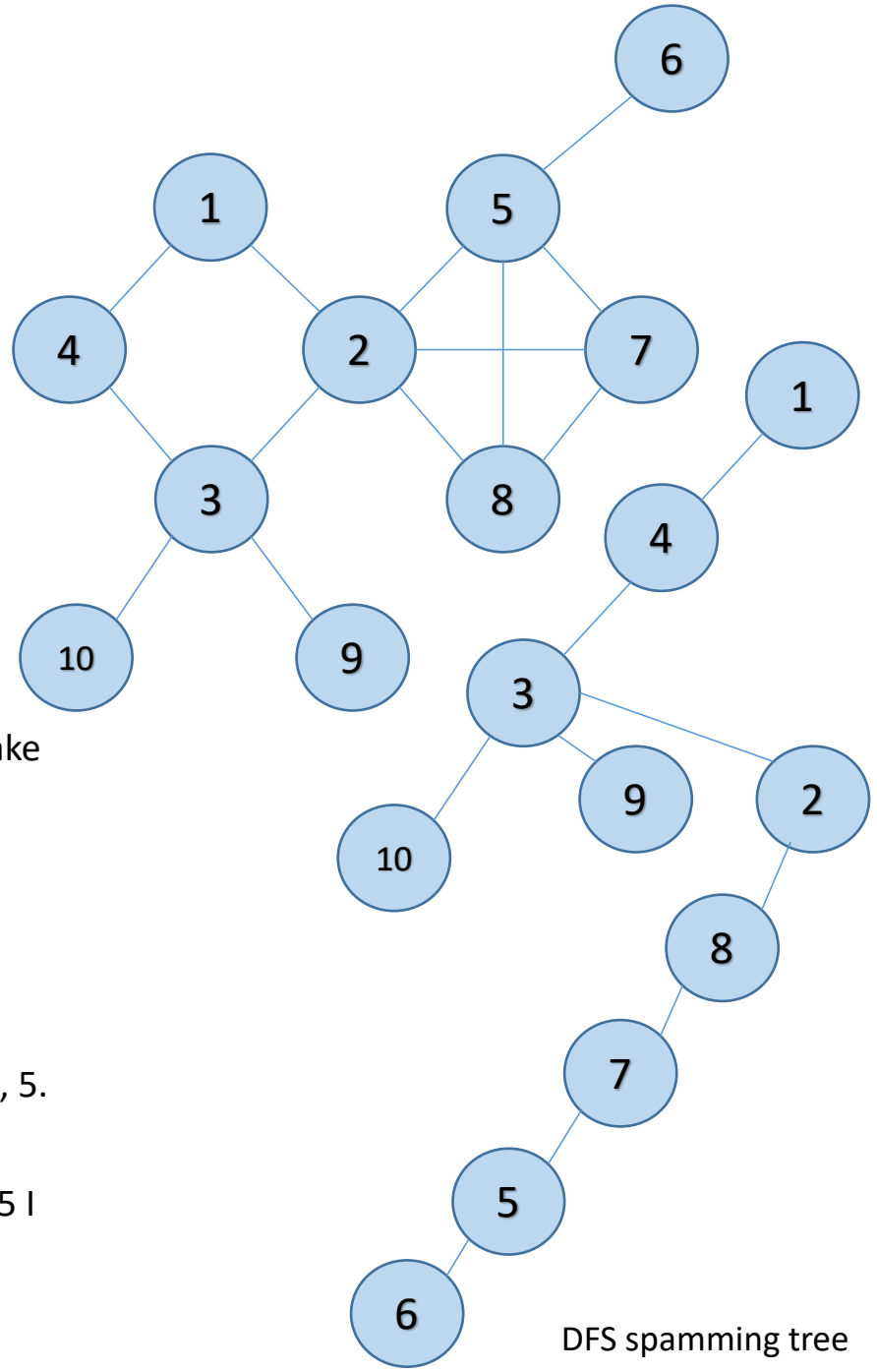
BFS rules

- You can start BFS from any vertex
- When you are exploring any vertex, then you can visit the adjacent vertices in any order
- But when you are selecting a vertex for exploration you must visit all its adjacent vertices, and then only you should go for next vertex for exploration
- You should select the next vertex for exploration from a queue

Depth First Search in-depth

Stk
~~5~~
7
8
2
~~3~~
3
~~3~~
4
1

- For DFS we will use a stack. Rule: once you have reached a new vertex, start exploring that new vertex
- I can start from any vertex – I will start at 1. DFS: 1
- Next I'll visit from its adjacent vertices, so I visit 4. DFS: 1, 4
- Even if there is one more adjacent vertex to 1, now I'll explore 4
- So I put 1 in stack so I can explore it later. DFS: 1, 4, 3
- Suspend 4 and start exploring 3. From 3 I can go to 10. DFS: 1, 4, 3, 10
- Suspend 3, start exploring 10. There is no adjacent vertex to 10 so I go back to 3. I take the 3 value out from the stack, and start exploring 9. Again I suspend 3 and start exploring 9. DFS: 1, 4, 3, 10, 9
- From 9 I cannot go anywhere so I go back to 3 (take it out from queue) and start exploring 3. 2 is adjacent to 3. DFS: 1, 4, 3, 10, 9, 2
- I suspend 2 and start exploring 2. From 2, 8 is adjacent. DFS: 1, 4, 3, 10, 9, 2, 8
- Now I start exploring 8. I suspend 8 and I can go to 7. DFS: 1, 4, 3, 10, 9, 2, 8, 7
- From 7 I can go to 5. I suspend 7 and push it into the stack. DFS: 1, 4, 3, 10, 9, 2, 8, 7, 5.
- From 5 I can go to 6. Visit 6 and suspend 5. DFS: 1, 4, 3, 10, 9, 2, 8, 7, 5, 6.
- There is no vertex adjacent to 6 so I go back to 5 (I take it out from the stack). From 5 I can visit 2, which is completed. 8 and 7 are also completed.
- Go back to 7. From 7 everything is visited and so on, until my stack is empty.



DFS spamming tree

Quick Note regarding searching inside lists

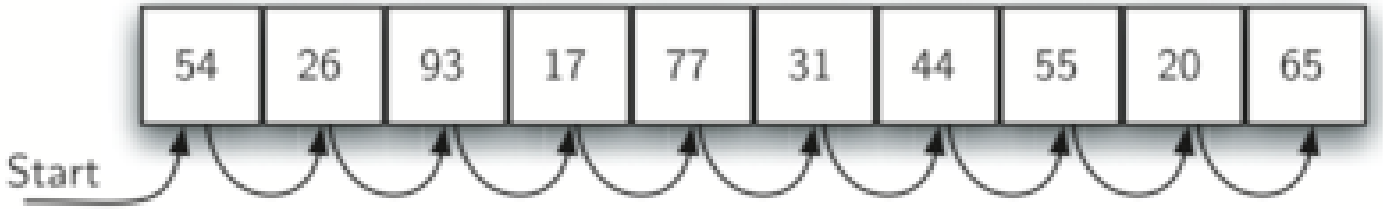
- In Python we can use the **in** operator to search for elements in a list

```
>>> 75 in [1, 2, 45]
False
>>> 45 in [1, 2, 45]
True
```

- But how the underline process actually work?
- What's the best way to search for things algorithmically?
- We'll see what's actually happening when we say `45 in [1, 2, 45]`

Sequential Search

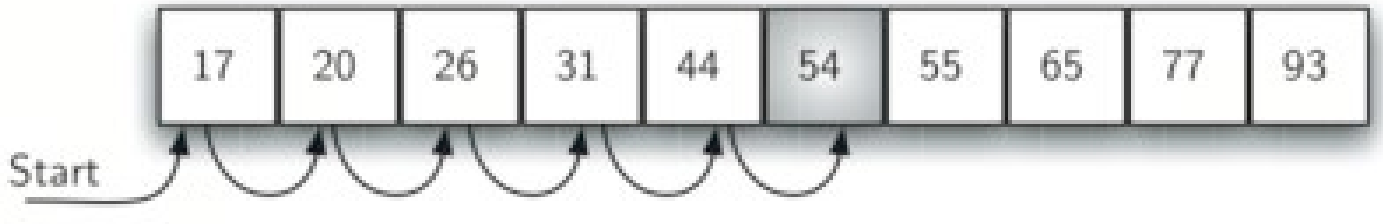
- Basic searching technique, sequentially go through the data structure, comparing elements as you go along.
- For example, on an unordered list searching for the element **50**:



- **50** was not present, but we still had to check every element in the array.
- But what if it was ordered?

Sequential Search

- If the list is ordered, we know we only have search until we reach an element which is a match or we reach an element which is greater than our search target.
- For example, searching for **50,** we can stop here at **54**.



Big O

Unordered list analysis

Case	Best Case	Worst Case	Average Case
Item is present	O(1)	O(N)	O(N)/2
Item is not present	O(N)	O(N)	O(N)

Ordered list analysis

Case	Best Case	Worst Case	Average Case
Item is present	O(1)	O(N)	O(N)/2
Item is not present	O(1)	O(N)	O(N)/2

Implementation

- We'll implement sequential search for an unordered list and then we'll adapt it to an ordered list

```
def seq_search(arr, ele): # the array we search through and the element we want to find
    pos = 0 # we start at position 0
    found = False # it will become true when we find the ele in arr
    while pos<len(arr) and not found:
        if arr[pos] == ele:
            found = True
        else:
            pos+=1
    return found

arr = [1,2,3,4,5]
print(seq_search(arr,2))
```

Implementation for ordered lists

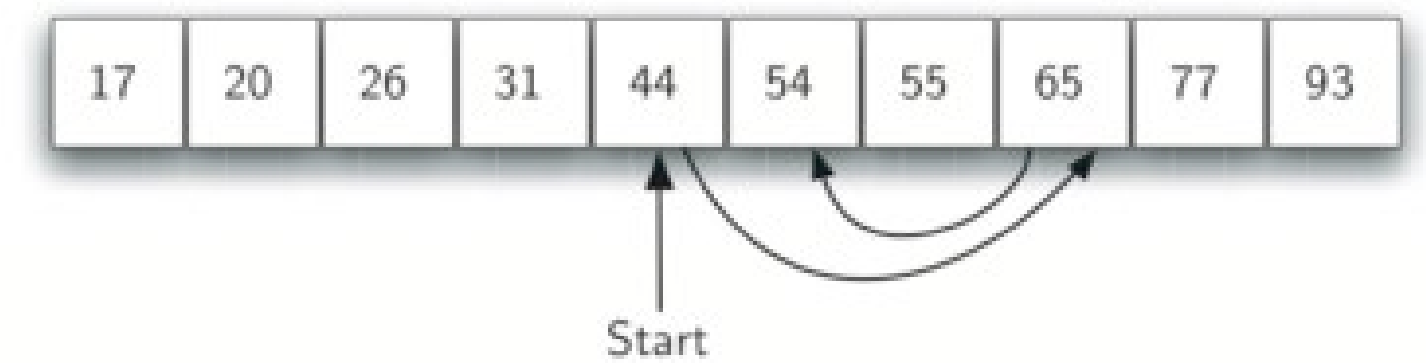
```
def ordered_seq_search(arr, ele):
    pos = 0 # I start at position 0
    found = False # it will become true if i find the elem in list
    stopped = False
    while pos<len(arr) and not found and not stopped: # stopped will come into play
when we reach an element that is greater than our target element
        if arr[pos] == ele:
            found = True
        else:
            if arr[pos] > ele:
                stopped = True
            else:
                pos+=1
    return found
arr = [1,2,3,4,5,6,7,8,9]
print(ordered_seq_search(arr, 12))
```

Binary Search

- We can take greater advantage of the ordered list!
- Instead of searching the list in sequence, a **binary search** will start by examining the middle item.
- A **binary search** will start by examining the middle item.
- If that item is the one we are searching for, we are done.
- If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration.
- The item, if it is in the list, must be in the upper half.

Binary Search

- We can then repeat the process with the upper half. Start at the middle item and compare it against what we are looking for.
- Again, we either find it or split the list in half, therefore eliminating another large part of our possible search space.
- For example, searching for **54** again:



- Binary search uses Divide and Conquer!
- **We divide the problem into smaller pieces, solve the smaller pieces in some way, and then reassemble the whole problem to get the result**.

Binary Search Big O

- Each comparison eliminates about half of the remaining items from consideration.
- What is the maximum number of comparisons this algorithm will require to check the entire list? It depends on the number of comparisons and the approximate number of items left.

Comparisons	Approximate Number of Items Left
1	n/2
2	n/4
3	n/8
…	…
i	$n/2^i$

Implementation – sorted arrays

```
def binary_search(arr,ele):
    # First and last index markers
    first = 0
    last = len(arr) - 1
    found = False # we'll use found as a marker
    while first <= last and not found: # while ele is not found we'll make a middle marker
        mid = (first+last)//2
        # Match found
        if arr[mid] == ele:
            found = True
        # Set new midpoints up or down depending on comparison
        else:
            if ele < arr[mid]:
                last = mid - 1 # we have chosen the first part of the list
            else:
                first = mid + 1 # we have chosen the last part of the list
    return found
arr = [1,2,3,4,5,6,7,8]
print(binary_search(arr,4))
```

Recursive Binary Search

```
def rec_bin_search(arr, ele):
    if len(arr) == 0:
        return False
    else:
        mid = len(arr)//2
        if arr[mid] == ele:
            return True
        else:
            if ele < arr[mid]:
                return rec_bin_search(arr[:mid], ele) # I call it on the first half of the array
            else:
                return rec_bin_search(arr[mid+1:], ele) # I call it on the second half of the array

print(rec_bin_search(arr, 3))
```

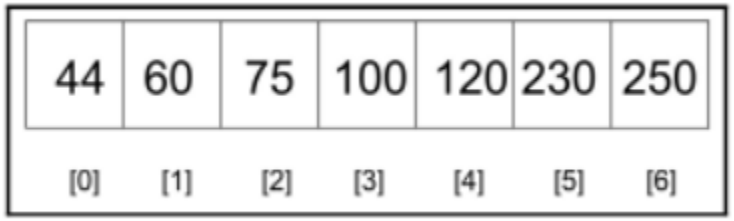
Interpolation Search

- It’s an improved version of binary search,
- Unlike binary search where we started searching form the middle, in interpolation search we determine the starting position in correspondence to the searched item.
- If the search item is near to the first element in the list, then the starting search position is likely to be near the start of the list.
- It works in a similar way to the binary search algorithm except for the method to determine the splitting criteria to divide the data in order to reduce the number of comparisons.
- Formula for dividing:
- `mid point = lower_bound_index + (( upper_bound index - lower_bound_index) // (input_list[upper_bound_index] - input_list[lower_bound_index])) * (search_value - input_list[lower_bound_index])`

In other words…

- $M = L + \frac{U-L}{in\_list[U]-in\_list[L]} * (searched - in\_list[L])$
- Where:
- M = middle point
- L = lower bound index
- U = upper bound index

Example



- L = 0
- U = 6
- in_l[U] = 250
- in_l[L] = 44
- *searched* = 230
- M = 0 + [(6-0)//(250-44) * (230-44) = 5
- So now the algorithm will start searching from index position 5

Exercise

- Modify `rec_bin_search` function so that it uses interpolation.

Wrapping everything up

- https://youtu.be/l1ed_bTv7Hw

Thank you!