Analysis of Algorithms: Sequential and Binary Searching Algorithms

Garrett Dancik, PhD

Fall 2024

Course Notes: https://gdancik.github.io

What do we mean by Searching?

- We want to search a list for a particular value, and output whether the value is in the list or is not.
  - Determine if a student is on my roster for CSC 180
  - Determine whether a song is on a play list
  - Modifications to this algorithm can be used to "look up" information
    - Look up a student's GPA (we need to find the student in the list)
    - Reverse phone number look up (look up number and get the person)
- For now we assume that the list is *unordered* (e.g., not sorted from smallest to largest; not alphabetical)

0	**21**	**19**	**0**	**18**	**19**

- In the list above,
  - If we look for 19, we should output "FOUND"
  - If we look for 100, we should output "NOT FOUND"

Sequential search algorithm

```
• Set the query to the desired search term (number, string, word, etc)
• Set index to 0
• Set Found to False
• While index <  n AND Found is False :
    • If  values[index] is equal to query:
        • Print message indicating the query has been found
        • Set FOUND to True
    • Else :
        • Increase index by 1
• If Found is False :
    • Print message indicating that the query was not found
```

We search through the list sequentially (from left to right), one element at a time, stopping when the element is found or when we reach the end of the list.

Sequential search algorithm (example)

- Search for "Jones" in a list of names
    - *query* = "Jones"

Found = False

index:	**0**					
	Gates	Jobs	Musk	Jones	Smith	Lovelace

Sequential search algorithm (example)

- Search for "Jones" in a list of names
  - *query* = "Jones"

Found = False

index:	**0**					
	Gates	Jobs	Musk	Jones	Smith	Lovelace

The list element ("Gates") is not equal to the query ("Jones"),
so increase *index* by 1

Sequential search algorithm (example)

- Search for "Jones" in a list of names
  - *query* = "Jones"

Found = False

index:	**0**	**1**				
	Gates	Jobs	Musk	Jones	Smith	Lovelace

The list element ("Jobs") is not equal to the query ("Jones"),
so increase *index* by 1

Sequential search algorithm (example)

- Search for "Jones" in a list of names
    - *query* = "Jones"

Found = False

index:	**0**	**1**	**2**			
	Gates	Jobs	Musk	Jones	Smith	Lovelace

The list element ("Musk") is not equal to the query ("Jones"),
so increase *index* by 1

Sequential search algorithm (example)

- Search for "Jones" in a list of names
  - *query* = "Jones"

Found = True

index:	0	1	2	3		
	Gates	Jobs	Musk	Jones	Smith	Lovelace

The list element ("Jones") **is** equal to the query ("Jones"), so:

- Print that the query was found
- Set *Found* to True
- We are now done with the loop

Sequential search algorithm:

- While *index* < *n* AND FOUND is False :
  - If *values*[*index*] is equal to query:
    - Print message indicating the *query* has been found
    - Set FOUND to True
  - Else :
    - Increase *index* by 1

Focus on the loop because this is where the majority of the work occurs (the other operations are all constant)

- Running time
  - Best case: the first element matches the query
    - The loop is repeated 1 time, so the order of magnitude for the running time is constant.
  - Worst case: the query is *not* in the list
    - The loop is repeated *n* times (we must check every element to ensure that it is not in the list). The order of magnitude for the running time is *n*.
  - Average case: How many times will the loop repeat, on average, assuming the *query* is equally likely to be anywhere in the list, or not in the list at all. The order of magnitude for the running time is *n*.

Binary search algorithm

- If the data is *sorted*, and we want to find an element in the list, we can do the following:
- Check the element in middle of the list
  - If the query comes before this element alphabetically or numerically, then we know the element we are looking for, if in the list, is in the first half. Select the middle element from the first half of the list.
  - Otherwise, we know the element must be in the second half of the list, if it exists. Select the middle element from the second half of the list.
- We repeat this process until the element is found or no elements remain
- Everytime we look at an element, we either:
  - Find the element we are looking for
  - Eliminate half of the remaining list

Binary search algorithm

- The data is *sorted*
- We want to find "Jobs"

index:	**0**	**1**	**2**	**3**	**4**	**5**	**6**
	Bezos	Gates	Jobs	Jones	Lovelace	Musk	Smith

Binary search algorithm

- The data is *sorted*
- We want to find "Jobs"

Check the middle element between index values
0 and 6, which has index (0+6)/2 = 3

index:	**0**	**1**	**2**	**3**	**4**	**5**	**6**
	Bezos	Gates	Jobs	Jones	Lovelace	Musk	Smith

The list element ("Jones") is *greater than* the query ("Jobs"),
so look at the middle element between index values
0 and 2, which is (0+2)/2 = 1

Binary search algorithm

- The data is *sorted*
- We want to find "Jobs"

Check the middle element between index values
0 and 6, which has index (0+6)/2 = 3

index:	**0**	**1**	**2**	**3**	**4**	**5**	**6**
	Bezos	Gates	Jobs	~~Jones~~	~~Lovelace~~	~~Musk~~	~~Smith~~

The list element ("Jones") is *greater than* the query ("Jobs"),
so look at the middle element between index values
0 and 2, which is (0+2)/2 = 1

Note that we can now rule out all elements from index 3 on

Binary search algorithm

- The data is *sorted*
- We want to find "Jobs"

Check the middle element between index values
0 and 2, which has index (0+2)/2 = 1

index:	0	1	2	3	4	5	6
	Bezos	Gates	Jobs	~~Jones~~	~~Lovelace~~	~~Musk~~	~~Smith~~

The list element ("Gates") is *less than* the query ("Jobs"),
so look at the middle element between index values
2 and 2, which is (2+2)/2 = 2

Note that we can also rule out all elements up to and including index 1

Binary search algorithm

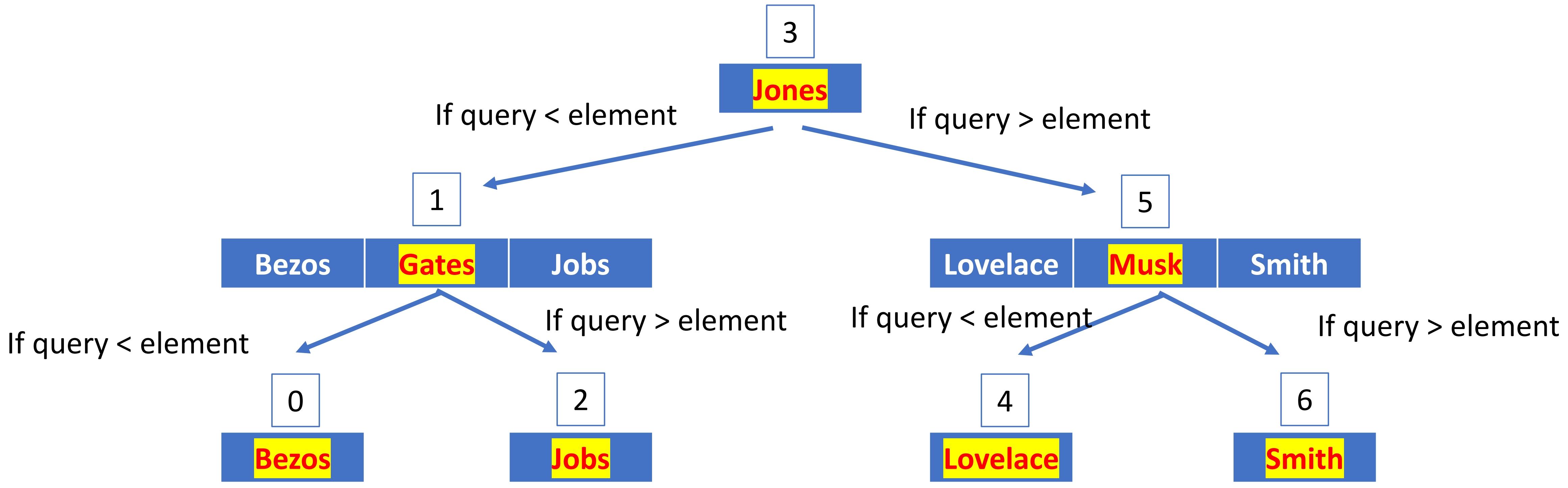
- The data is *sorted*
- We want to find "Jobs"

Check the middle element between index values
2 and 2, which has index (2+2)/2 = 2

index:	**0**	**1**	**2**	**3**	**4**	**5**	**6**
	~~Bezos~~	~~Gates~~	Jobs	~~Jones~~	~~Lovelace~~	~~Musk~~	~~Smith~~

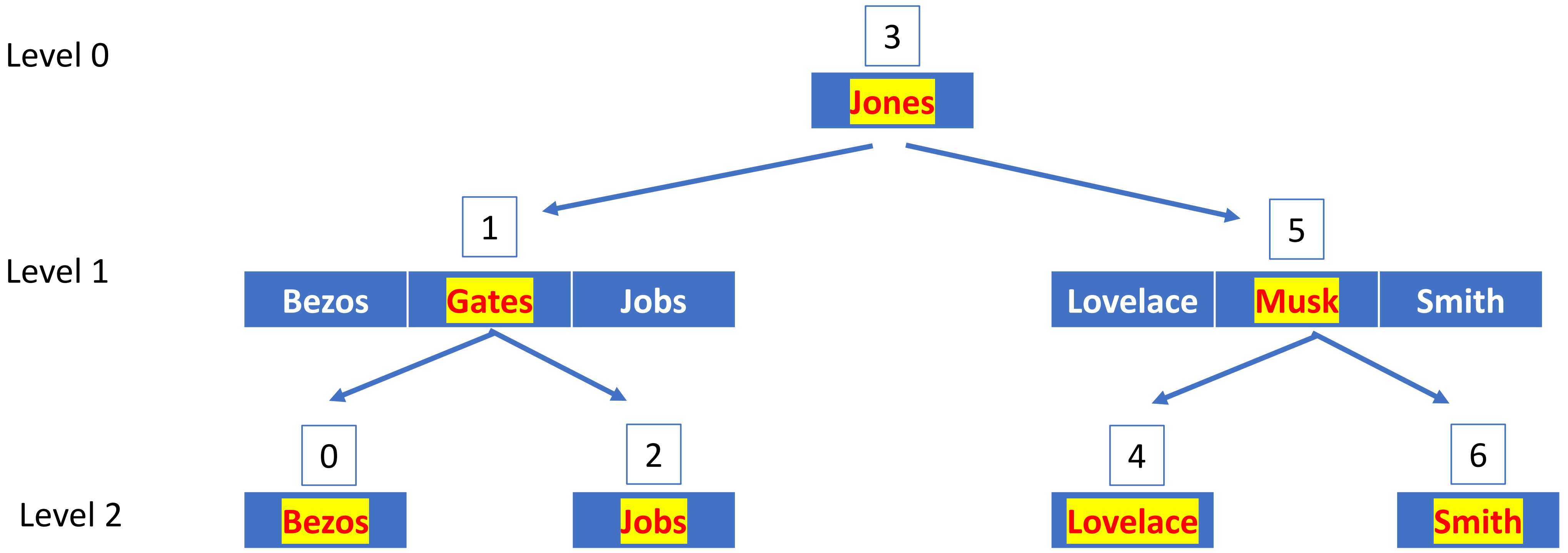
The list element ("Jobs") **is** the query element ("Jobs"),
we have found the element!

Binary search algorithm – tree visualization



index:	**0**	**1**	**2**	**3**	**4**	**5**	**6**
	Bezos	Gates	Jobs	Jones	Lovelace	Musk	Smith

Binary search algorithm – tree visualization



- The crux of the algorithm is checking whether the current element is what we are looking for.
  - We only need to do one check (comparison) for each level of the tree
  - What is the most checks that we will need to do?

Binary search algorithm

- The number of comparisons is roughly the number of times we can divide the list into 2
- Suppose we had 8 elements
  - 8 / 2 → 4
  - 4 / 2 → 2
  - 2 / 2 → 1
- 8 can be divided by 2 three times.
- In other words, $2^3 = 8$

- $\log_2 8 = 3$
- The number of times *n* can be divided by 2 is: $\log_2 n$

Binary search algorithm

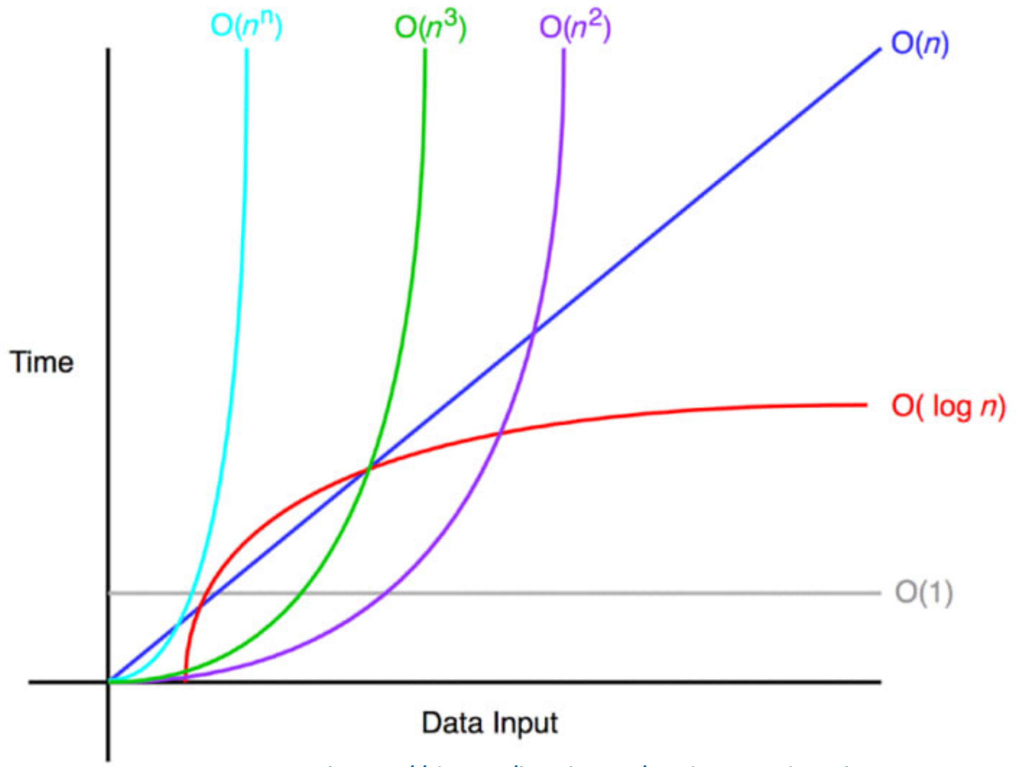
- The number of levels in the tree is $\lfloor log_2(n) \rfloor + 1$, where $\lfloor \quad \rfloor$ is the *floor* function which means to round down
- In the case of 7, $\lfloor log_2(7) \rfloor + 1 = \lfloor 2.8 \rfloor + 1 = \lfloor 2 \rfloor + 1 = 3$

- Best case (first value matches): $\theta(1)$
- Worst case (searches entire tree): $\theta(log_2 n)$
- Average case: $\theta(log_2 n)$

Order of magnitude: $log_2(n)$ vs n

- *Order n:* doubling the size of the input will result in an algorithm that takes twice as long
- Order $log_2(n)$: if you double the size of the input, you increase the number of operations by 1 (which hardly increases the running time)

n	log (n)
1	0
2	1
4	2
8	3
16	4
32	5
64	8



https://dev.to/b0nbon1/understanding-big-o-notation-with-javascript-25mc

Binary search algorithm

We have a list *L* of *n* elements and are looking for a target element *T*.

- Set *left* to 0 and *right* to *n – 1*
- *Set Found* to False
- While (not *Found* and *left* <= *right*) :
  - Set *m* to the middle value, at index (*left* + *right*) / 2, rounding down if the result is a decimal
  - If *L*[*m*] < T :
    - Set *L* to *m* + 1
  - Elif *L*[*m*] > T :
    - Set *R* to *m* – 1
  - Else : (we have found the target)
    - Set *Found* to True

Because each iteration of the *while* loop effectively reduces the searchable elements by ½, the loop is repeated at most $log_2(n) + 1$ times.

Hypothetical running times

- Suppose our computer can do 10,000 comparisons per second

	Linear Search	**Binary Search**
n	# comparisons (seconds)	# comparisons (seconds)
1,000	1,000 (0.1 seconds)	10 (0.001 seconds)
10,000	10,000 (1 second)	13 (0.0013 seconds)
1,000,000	1,000,000 (100 seconds = 1 min 40 sec)	20 (0.002 seconds)
1,000,000,000	1,000,000,000 (100000 seconds ~ 27 hours)	30 (0.003 seconds)

- Binary search requires that the data is sorted, but we can sort data in $n\ log_2(n)$ time. If we will need to search data multiple times, it is better to sort the data and use binary search.