Searching

AIE 311 : Data structure and Algorithm

Searching

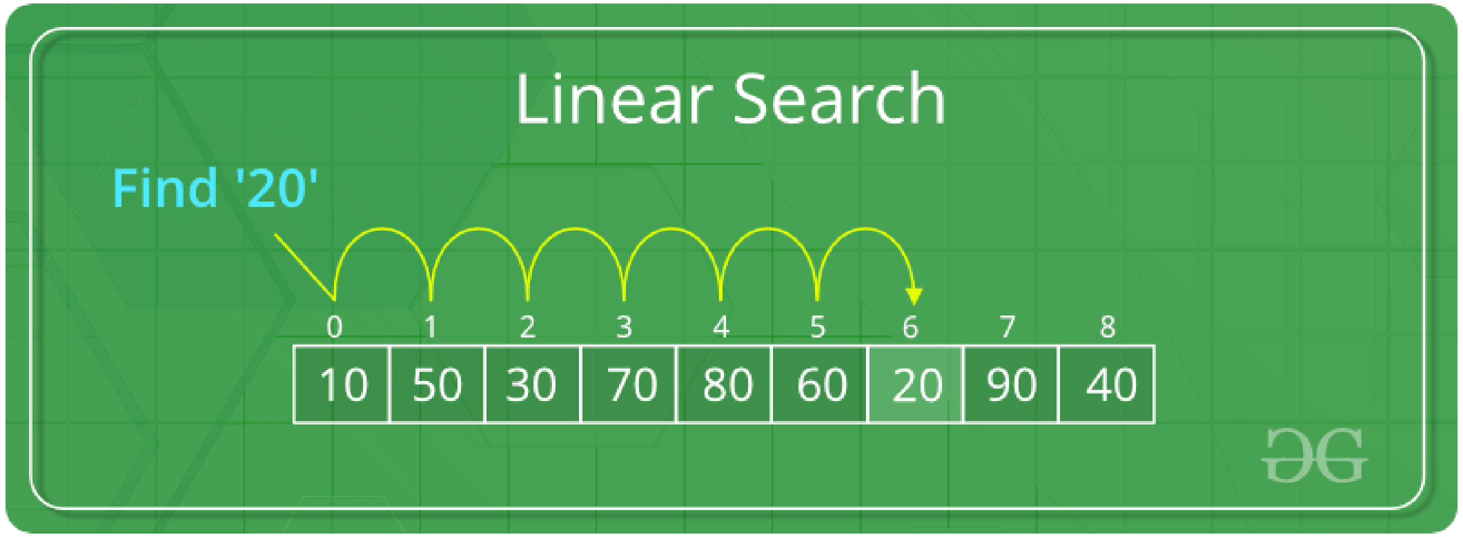
- Searching in data structure
  - The process of finding a particular element within a data structure. The goal of a searching algorithm is to efficiently locate the desired data, often based on a condition or specific key.

    Type of searching in data structure :

    1. Linear search
    2. Binary search
    3. Interpolation search
    4. Fibonacci search
    5. Exponential search
    6. Ternary search
    7. Jump search

Searching

- Linear search
  - The linear search algorithm is the easiest method to search but mediocre performance. Due to linear search will start from the starting point until reach the item that needed to be search by increase searching iterature by 1.



Ref : https://www.geeksforgeeks.org/difference-between-searching-and-sorting-algorithms/

Searching

- Linear search (continued)
  - Unsorted array

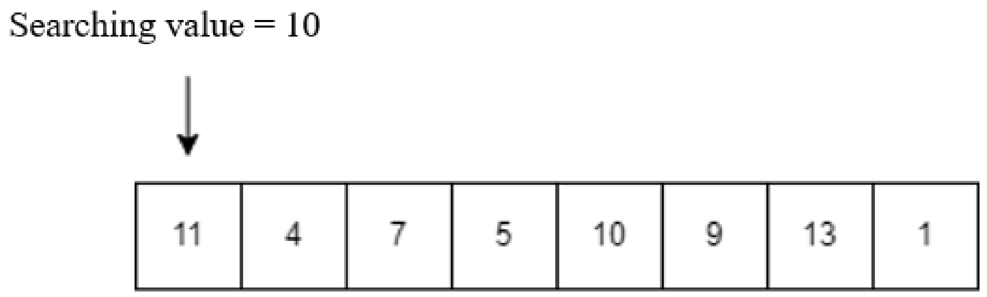


Figure 2: Linear search step 1 (Unsorted)

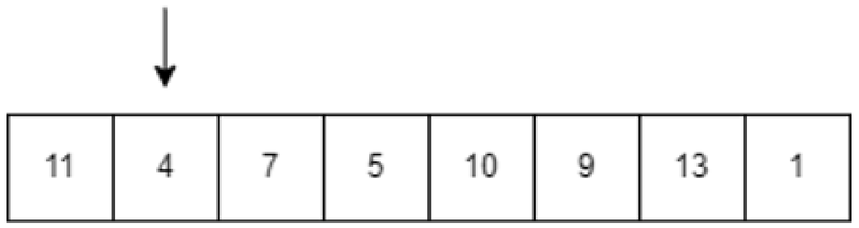


Figure 3: Linear search step 2 (Unsorted)

Searching

- Linear search (continued)
  - Unsorted array

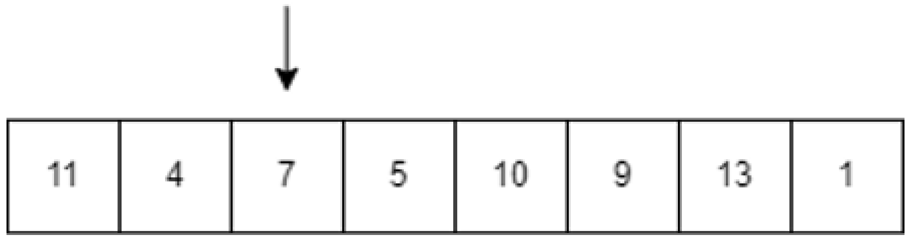


Figure 4: Linear search step 3 (Unsorted)

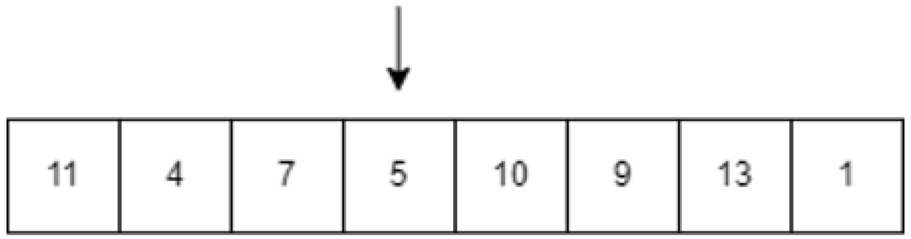


Figure 5: Linear search step 4 and hit (Unsorted)

Searching

- Linear search (continued)
    - Sorted array

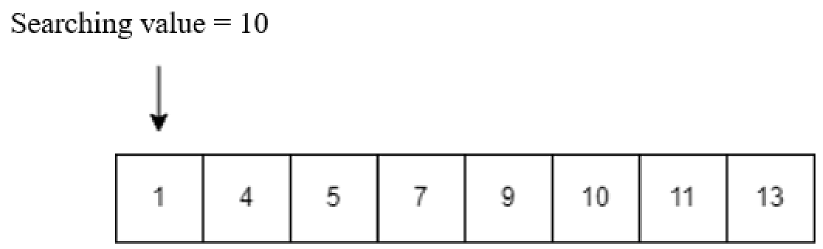


Figure 6: Linear search step 1 (Sorted)

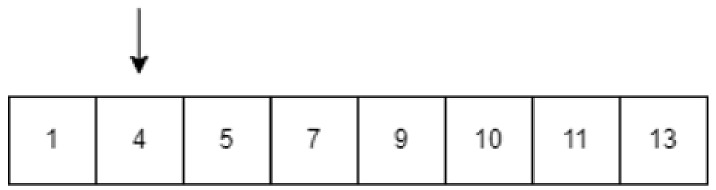


Figure 7: Linear search step 2 (Sorted)

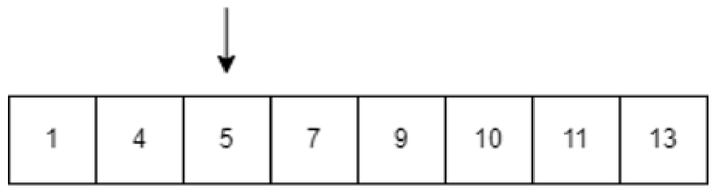


Figure 8: Linear search step 3 (Sorted)

Searching

- Linear search (continued)
    - Sorted array

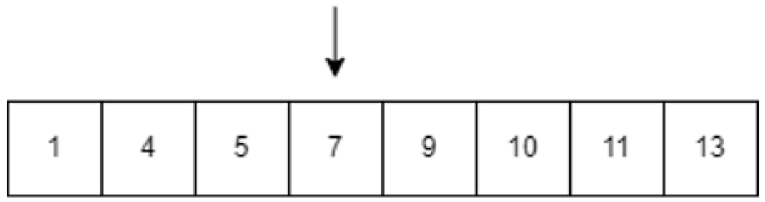


Figure 9: Linear search step 4 (Sorted)

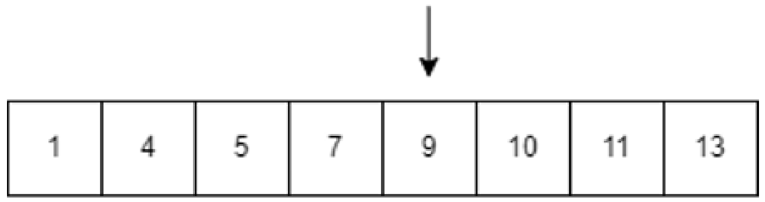


Figure 10: Linear search step 5 (Sorted)

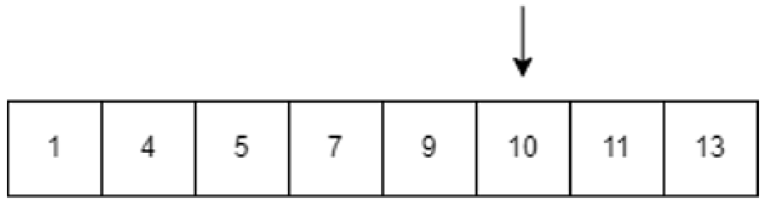
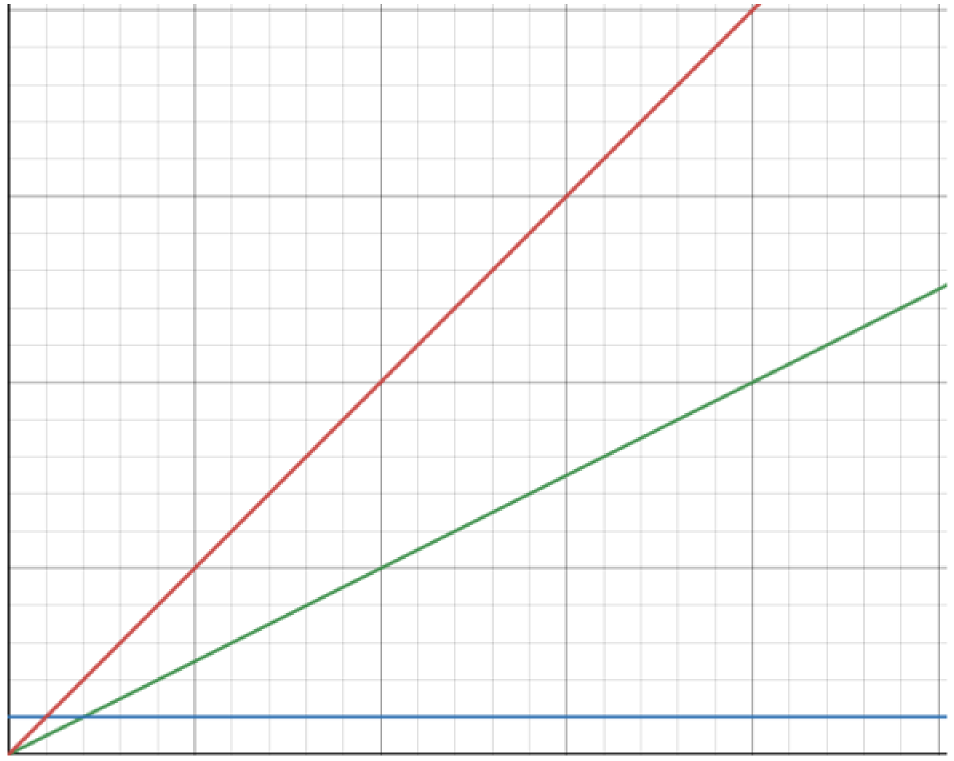


Figure 11: Linear search step 6 and hit (Sorted)

Searching

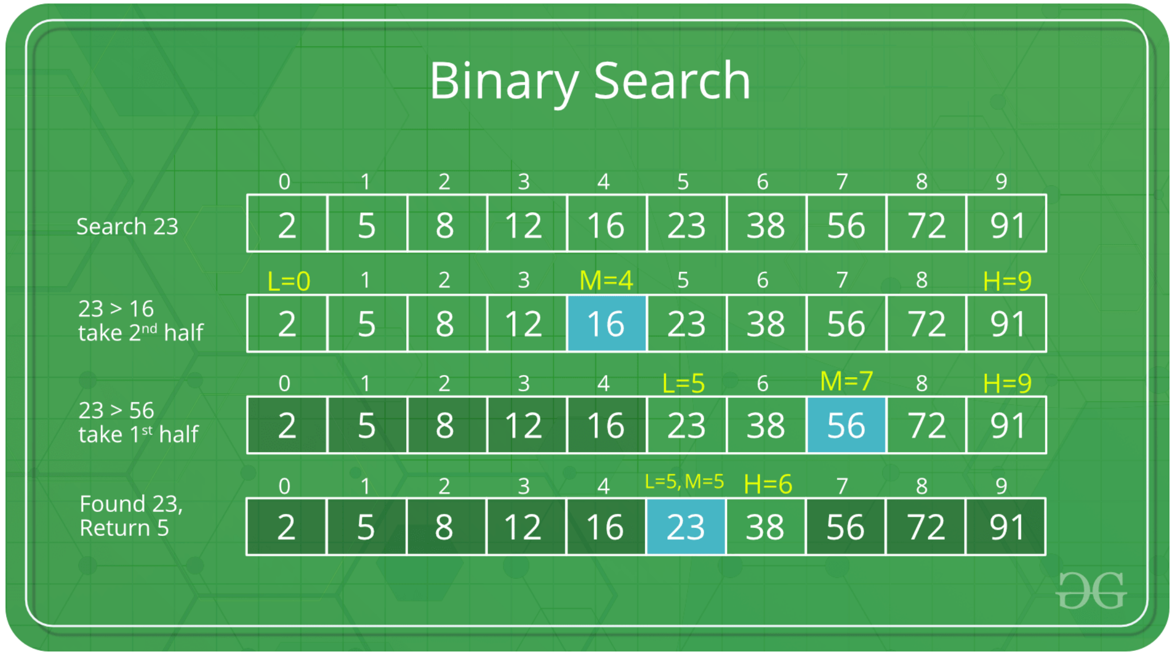
- Linear search (continued)
  - Performance
    1. Worst-case performance => O(n)
    2. Best-case performance => O(1)
    3. Average-case performance => O(n/2)



$O(n)$ as red, $O\left(\frac{n}{2}\right)$ as green, $O(1)$ as blue

Searching

- Binary search
  - Binary searh is the algorithm that separate searching array by half. Then comparing searching value and value in median address of the array. After comparing the values, if the searching value is more than value in the address. The next median address will be the right of its. If the searching value is less than the value in current median address. The next median address will be the left. This algorithm required sorted array.



Ref : https://www.geeksforgeeks.org/difference-between-searching-and-sorting-algorithms/

Searching

- Binary search (continued)

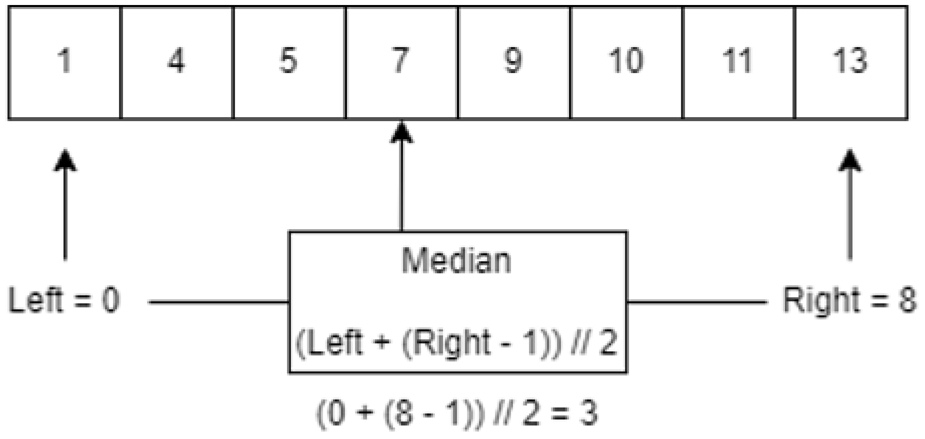


Figure 13: The starting point of binary search

Searching value = 10

Searching

- Binary search (continued)

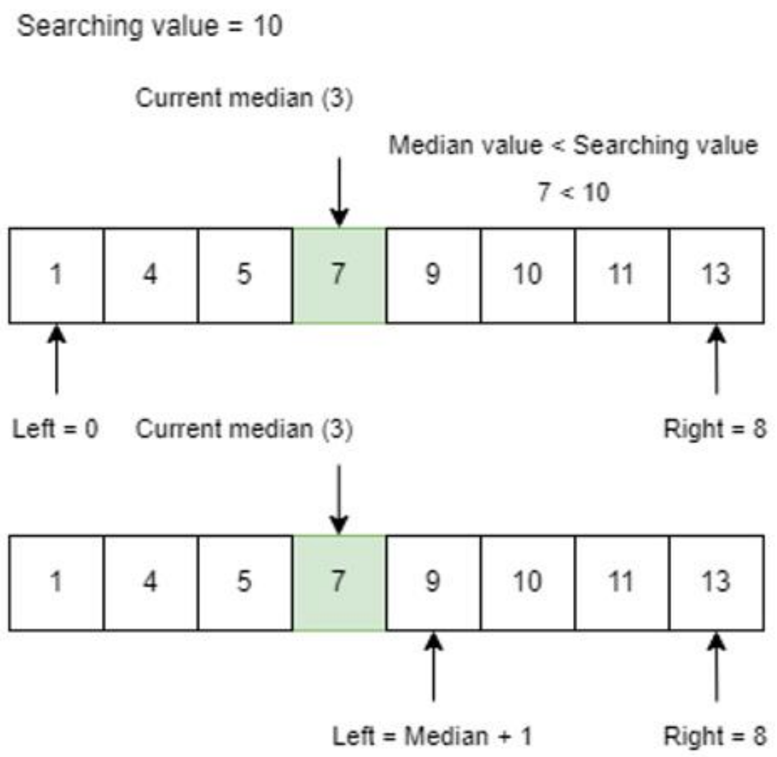


Figure 14: Binary search step 2 (Move left or right)

Searching

- Binary search (continued)

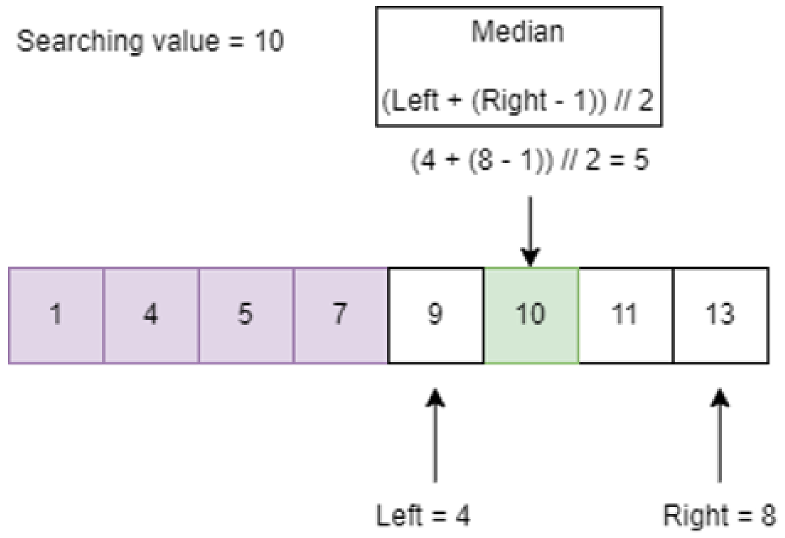


Figure 15: Binary search step 3 and hit (Find next median point)

Searching

- Binary search (continued)
  - Programming concept for binary search

    1. Find a staring median point by floor divide with left and right. Which left equal to 0 because the left side is never used and the right side is equal to array length – 1 due to the right side is never used before

Left = 0 and Right=array length - 1

Median = floor division of Left and Right

Median = $(\lfloor \text{Left/Right} \rfloor)$

Searching

- Binary search (continued)
  - Programming concept for binary search

    2. If median is not hit with searching value, the next step is comparing between current median value and searching value.

Current median value=Searching value

Condition 1: If hit

Current median value < Searching value

Left = Current median index + 1

Median = floor division of Left and Right

Condition 2: If median value less than searching value

Current median value > Searching value

Right = Current median index - 1

Median = floor division of Left and Right

Condition 3: If median value more than searching value

Searching

- Binary search (continued)
    - <u>Programming concept for binary search</u>

      3. All of above do until hit or under Left is less than or equal Right.

Do while: Left ≤ Right

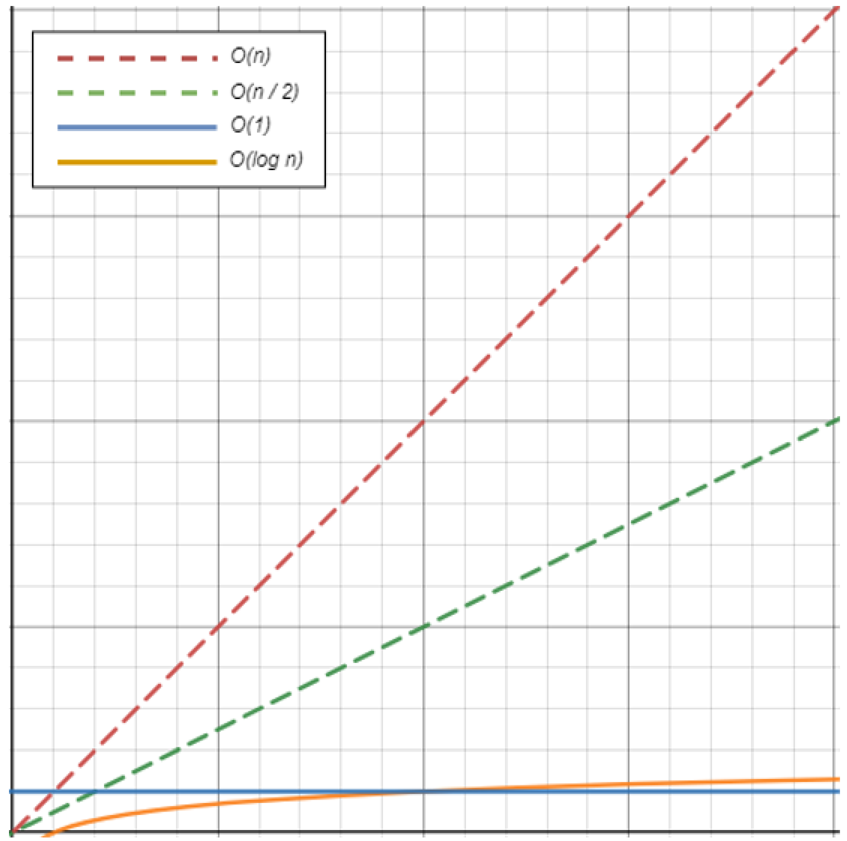
Condition 4: Do when under this condition

If median value = searching value: Return median address

Condition 5: If hit condition

Searching

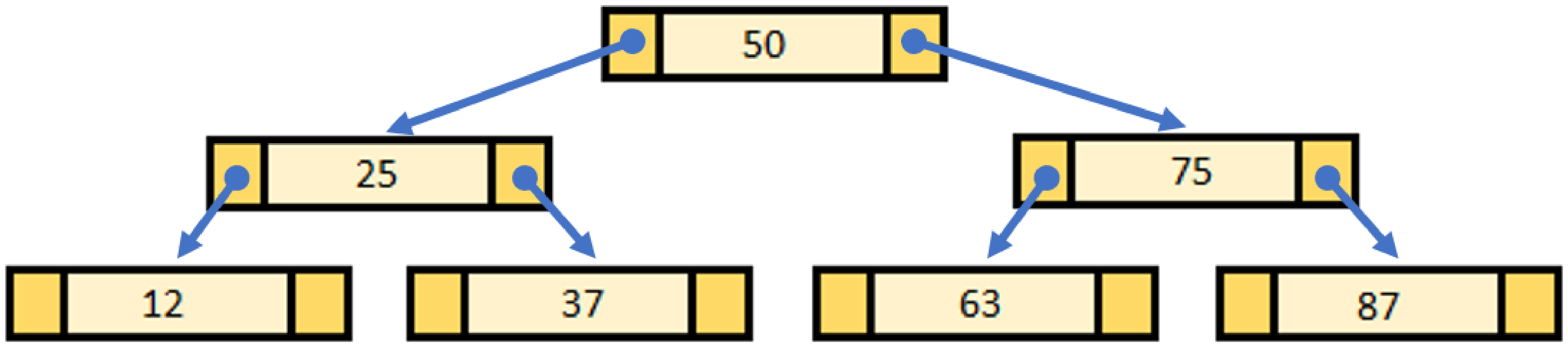
- Binary search (continued)
  - Performance
    1. Worst-case performance => O(logn)
    2. Best-case performance => O(1)
    3. Average-case performance => O(logn)



Comparison between performance of linear search and binary search

Searching

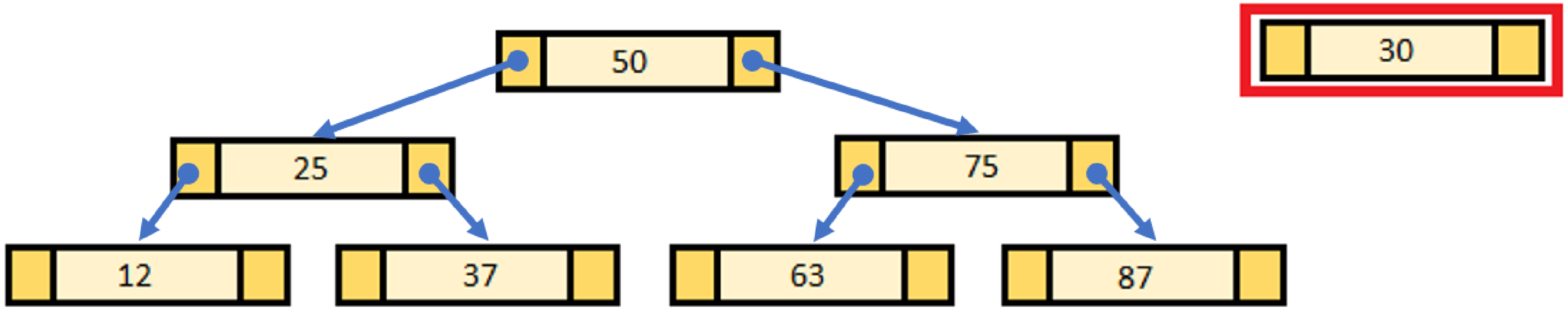
- Binary Search Tree (BST)
  - Add value in binary search tree



Binary tree with value

Searching

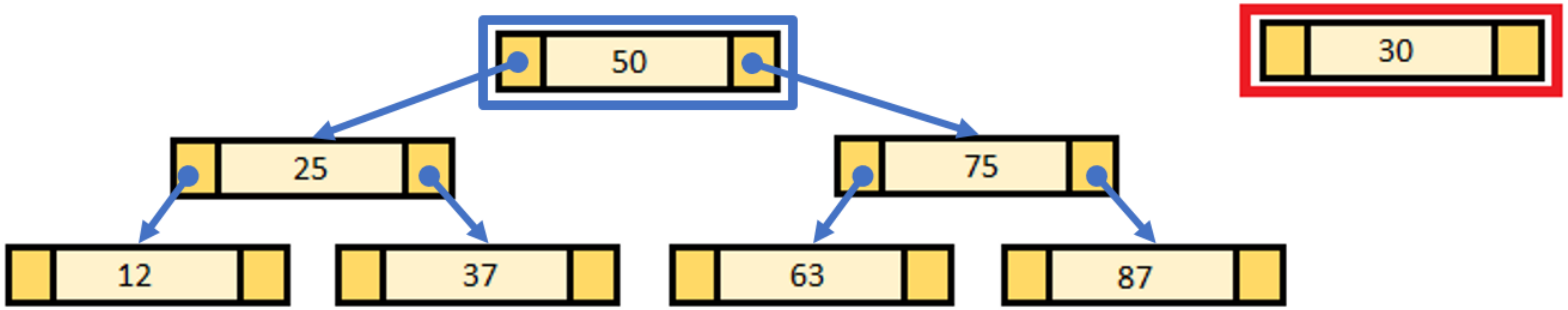
- Binary Search Tree (BST)
  - Adding new node (value = 30) : New node to add value is 30 (red rectangle)



Adding new value

Searching

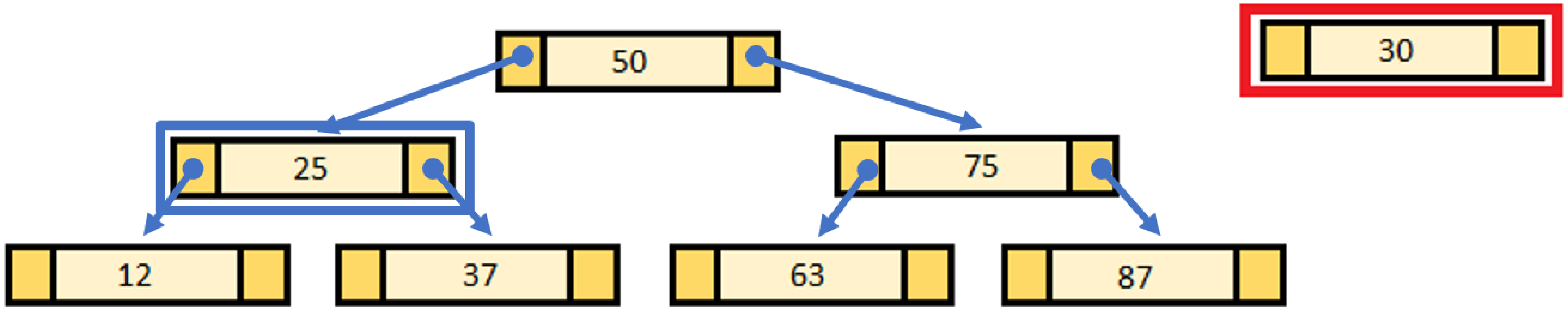
- Binary Search Tree (BST)
  - <u>Searching placing node</u> : New node value < root node ---> move left



Blue rectangle is current comparison node (value = 50)

Searching

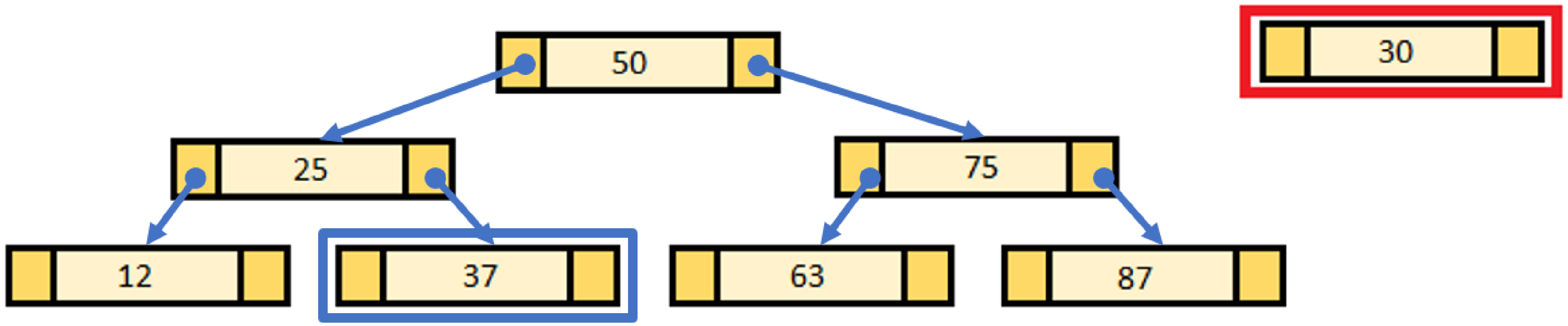
- Binary Search Tree (BST)
  - Searching placing node : New node value > current node ---> move right



Blue rectangle is current comparison node (value = 25)

Searching

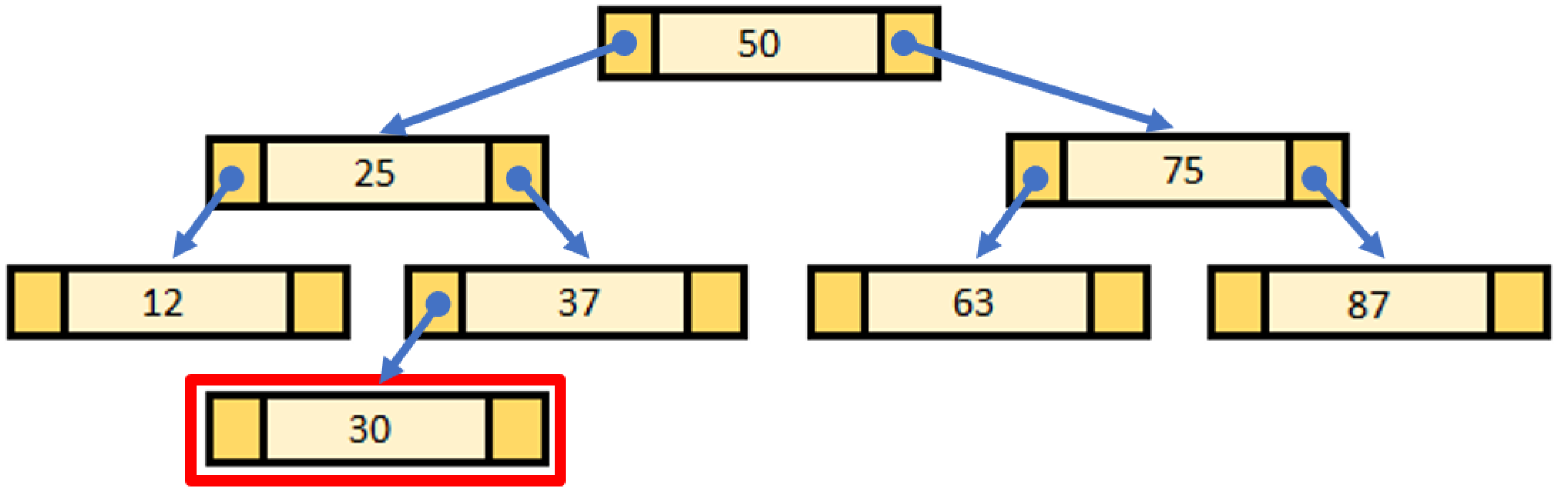
- Binary Search Tree (BST)
  - Searching placing node : New node value < current node ---> move left but there is no further node



Blue rectangle is current comparison node (value = 37)

Searching

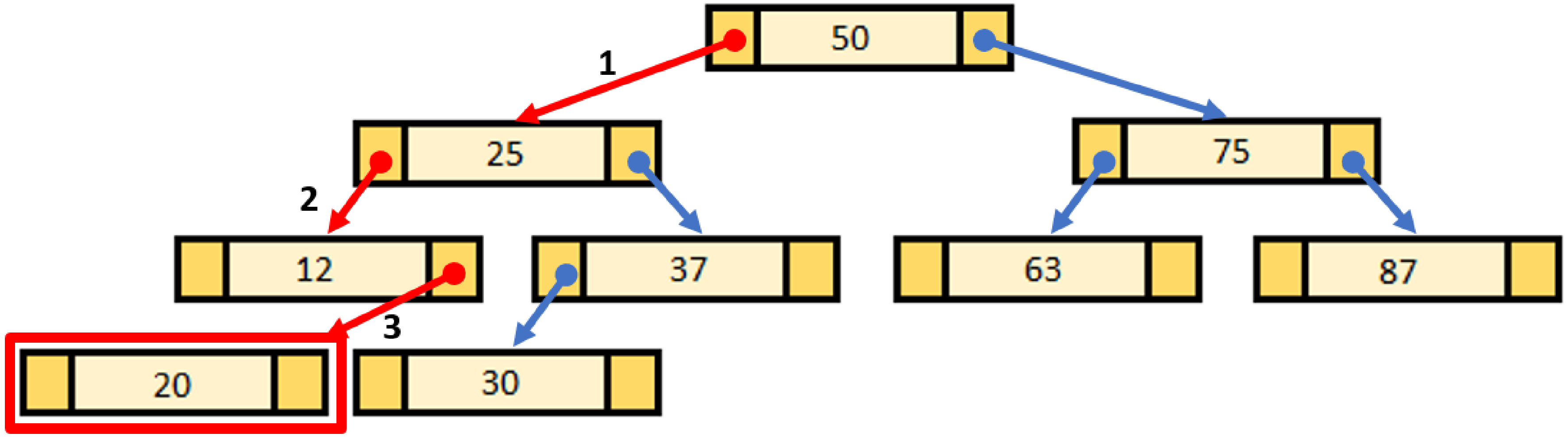
- Binary Search Tree (BST)
  - Connect the new node



Binary tree after added a new node (red rectangle, value = 30)

Searching

- Binary Search Tree (BST)
  - Another example of adding node in binary tree

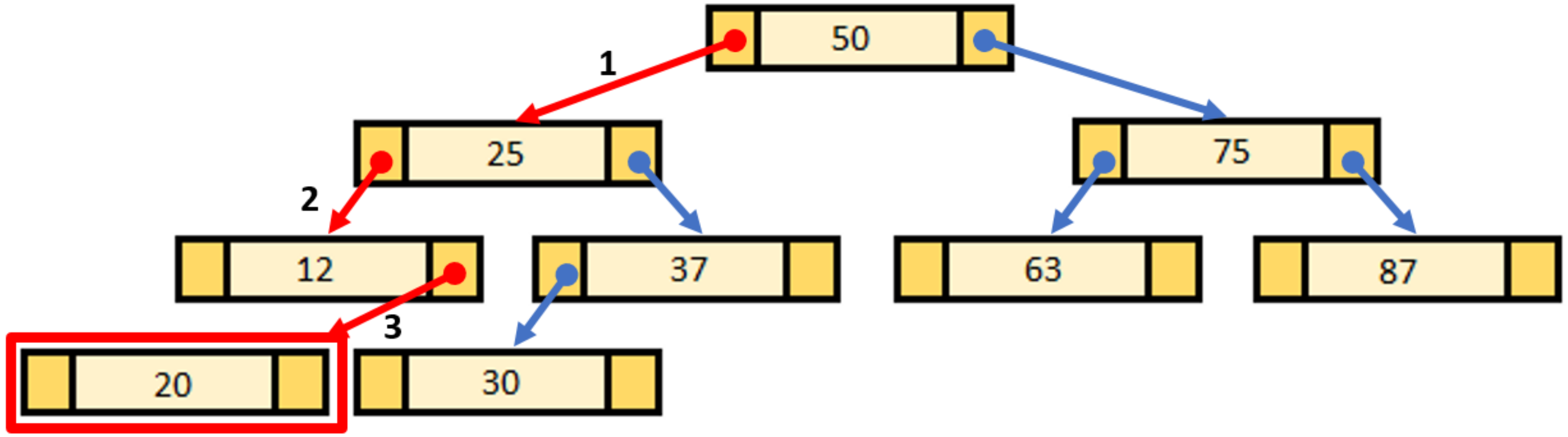


Binary tree after added a new node (red rectangle, value = 20)

Searching

- Depth first search (DFS) in binary search tree
  - Depth-first search is a tree traversal algorithm that progresses by exploring each branch deeply until it reaches a leaf node before backtracking to explore other branches. Tree traversal using Depth-First Search can be performed in three distinct ways:

1. **Preorder:** Node ⟶ Left ⟶ Right => 50, 25, 12, 20, 37, 30, 75, 63, 87
2. **Inorder:** Left ⟶ Node ⟶ Right => 12, 20, 25, 30, 37, 50, 63, 75, 87
3. **Postorder traversal:** Left ⟶ Right ⟶ Node => 20, 12, 30, 37, 25, 63, 87, 75, 50

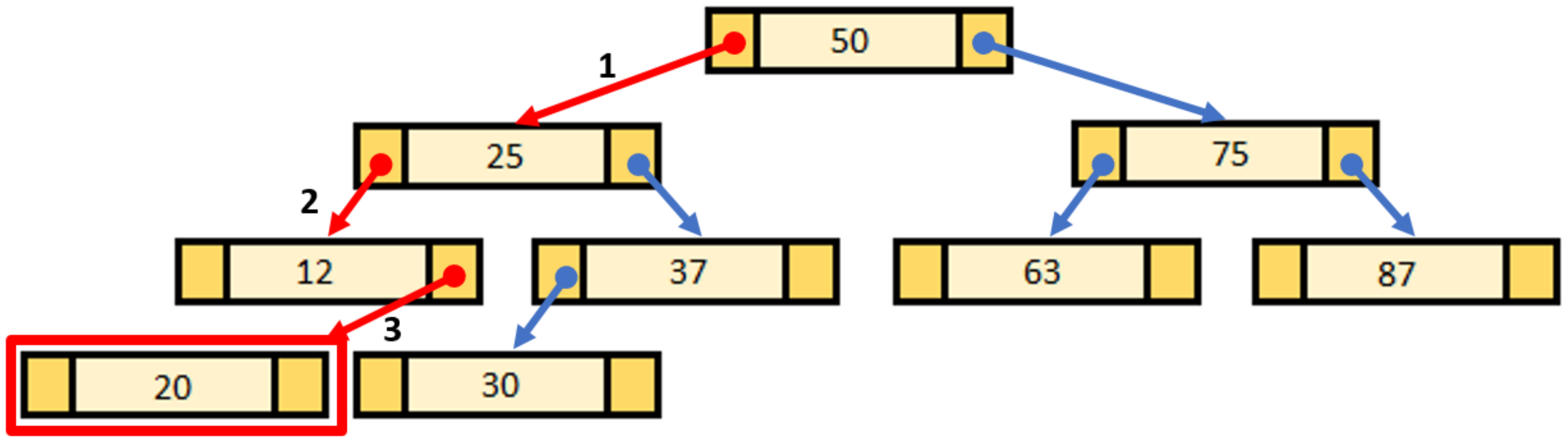


Binary tree after added a new node (red rectangle, value = 20)

Searching

- Breadth-First Search (BFS) in binary search tree
  - Breadth-First Search (BFS) traverses all nodes at the current depth level before progressing to the next level, making it synonymous with level-order traversal.

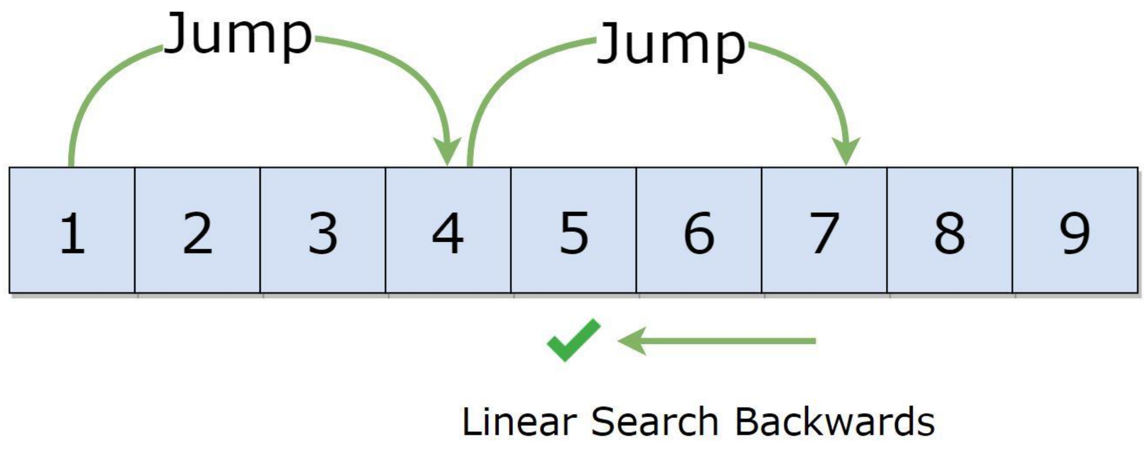
  Example of Breadth-First Search (BFS) => 50, 25, 75, 12, 37, 63, 87, 20, 30



Binary tree after added a new node (red rectangle, value = 20)

Searching

- Jump search
  - Jump Search is an algorithm used for searching in a sorted list or array, offering a compromise between linear search and binary search in terms of both time complexity and implementation complexity.



Ref :http://theoryofprogramming.azurewebsites.net/2016/11/10/jump-search-algorithm/

Searching

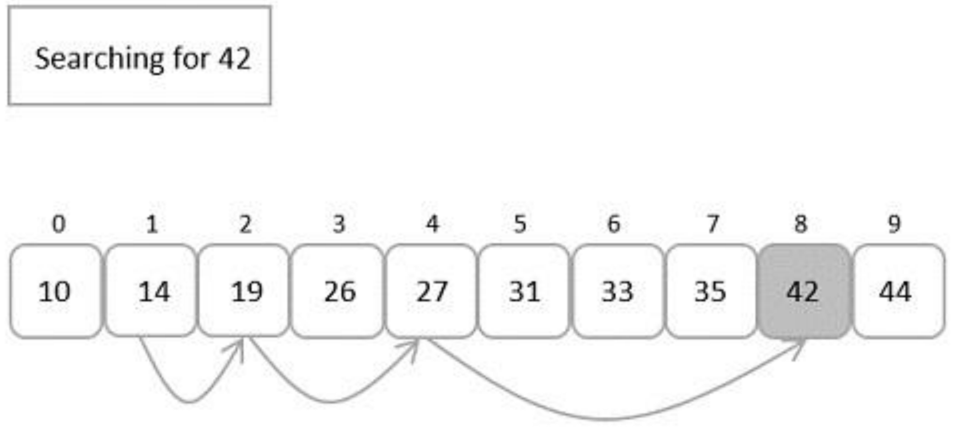
- Jump search

The process of jump search:

1. If the input array size is 'n', the block size is $\sqrt{n}$, and the index ***(i)*** is set to 0.
2. The search key (the element you're looking for) is compared to the element at the current index ***(i)*** of the array. If the key matches the element at that index, the algorithm returns the position (index) of the element. If the key does not match, the algorithm moves ahead by a block size (usually n elements) and continues comparing at the new index.
3. The previous step is repeated until the element at index ***(i)*** becomes greater than or equal to the key.
4. Since the array is sorted, the element is determined to be in the previous block, and a linear search is then performed within that block to locate the element.
5. If the element is located, its position is returned; otherwise, an indication of an unsuccessful search is provided.

Searching

- Exponential search
  - Exponential Search (also called Exponential Binary Search) is an efficient search algorithm designed for searching in a sorted array. It is particularly useful when the array is large and the index of the element is unknown. The fundamental idea of exponential search is to identify the range where the element might be located, and then apply binary search within that range. This approach makes exponential search more efficient than linear search and can provide better performance in specific scenarios.



Ref :https://www.tutorialspoint.com/data_structures_algorithms/exponential_search.htm

Searching

- Exponential search

The process of exponential search:

1. Compare the key with the first element of the array; if a match is found, return the index 0.
2. Initialize i=1 , then compare the element at index i with the key. If a match is found, return the index.
3. If the element does not match, the algorithm jumps through the array exponentially, following powers of 2. At each increment, it compares the element at the new position.
4. If a match is found, the index is returned. Otherwise, Step 2 continues iteratively until the element at the current position exceeds the key being searched for.
5. Since the next increment contains an element greater than the key and the input is sorted, the algorithm performs a binary search within the current block.
6. If a match is found, the index of the key is returned; otherwise, the search is determined to be unsuccessful.