Algorithms

Lecture 8
Search Algorithms

A. S. M. Sanwar Hosen

Email: sanwar@wsu.ac.kr

Date: 11 May, 2023

Search Algorithms

- **Search Algorithm:** A search algorithm is a step-by-step procedure used to locate an element among the collection of elements.

Example: Find or locate $x = 17$ in an array $A[0\!:n]$

2	7	9	13	17	23

- Types of Search Algorithm

Different types of search algorithms are:

- ✓ Linear Search
- ✓ Binary Search
- ✓ Jump Search
- ✓ Interpolation Search
- ✓ Exponential Search
- ✓ Sublist Search

Linear Search (1)

- **Linear Search:** It finds the position of a target element sequentially among the elements in an array.
- How It Works?

Linear search works as follows:

- ✓ A sequential search is made over all elements one by one in an array.
- ✓ Every element is checked and if match is found then the particular element is returned.
- ✓ The process is continued until the target element is found in the array.

- **Time Complexity:** The time complexity of linear search is O(n).

Linear Search (2)

❑ Linear Search

Let’s look at an example, suppose an array is $A[0{:}\,n]$, search the element $x = 17$ in the array.

Steps:

	7	23	9	13	17	1
Index position →	0	1	2	3	4	5

Iteration 1:
Set $i = 0$, $A[i]$ is not equal to $x = 17$

$A[i] = 7$

7	23	9	13	17	1

$i = 0$

Iteration 2:
Set $i = 1$, $A[i]$ is not equal to $x = 17$

$A[i] = 23$

7	**23**	9	13	17	1

$i = 1$

Iteration 3:
Set i =2, $A[i]$ is not equal to $x = 17$

$A[i] = 9$

7	**23**	**9**	13	17	1

$i = 2$

Iteration 4:
Set i =3, $A[i]$ is not equal to $x = 17$

$A[i] = 13$

7	**23**	**9**	**13**	17	1

$i = 3$

Iteration 5:
Set $i = 4$, $A[i]$ is equal to $x = 17$,
Return $A[i]$ and stop searching

$A[i] = 17 = x$

7	**23**	**9**	**13**	**17**	1

$i = 4$

Binary Search (1)

- **Binary Search:** It finds the position of a target element among the elements in a sorted array.
- How It Works?

Binary search works as follows:

- ✓ Initially, it sorts an array if the array is not sorted already.
- ✓ Then it searches a sorted array by repeatedly dividing the search interval in half.
- ✓ Begin with an interval covering the whole array.
- ✓ If the value of search key is less than the element in the middle of the interval, narrow the interval to the lower half, otherwise narrow it to the upper half.
- ✓ Repeatedly check until the element is found, or the interval is empty.

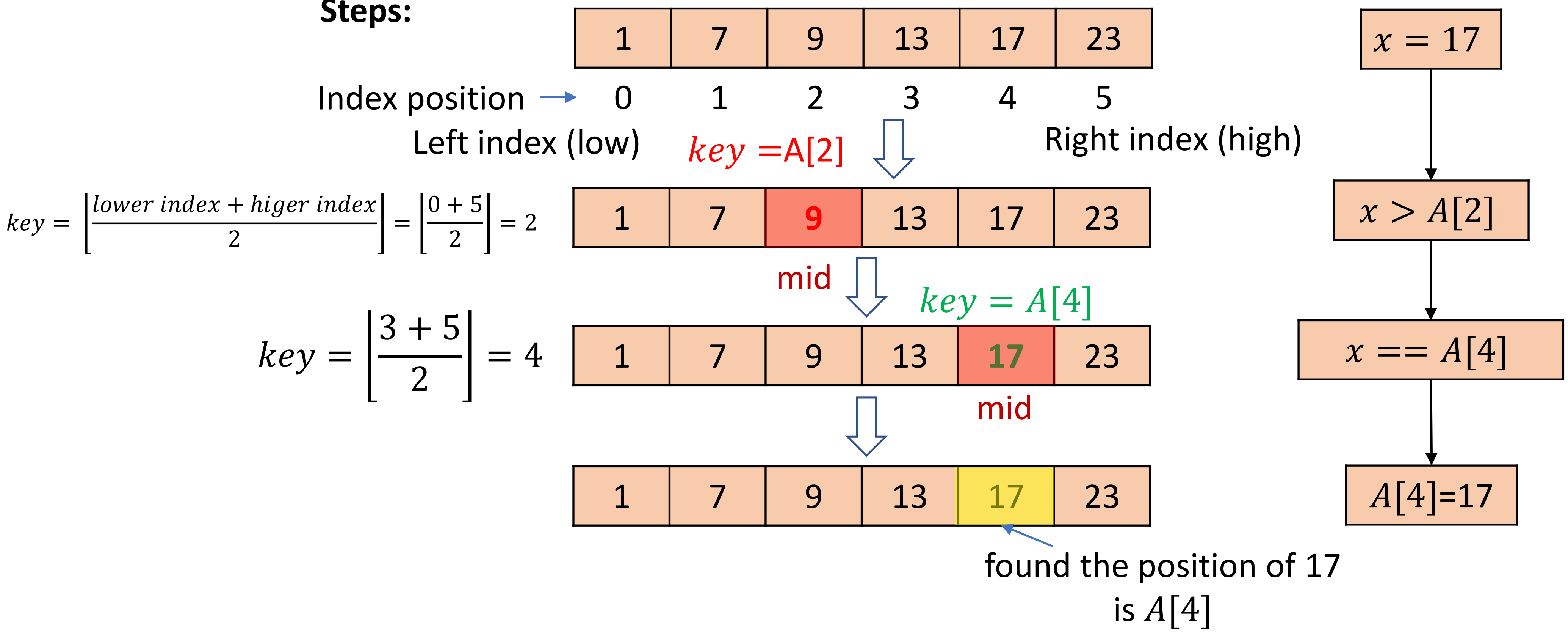
- **Time Complexity:** The time complexity of binary search is O($\log\ n$).

Binary Search (2)

- Binary Search

Let's look at an example, suppose a sorted array is $A[0{:}n]$, search the element $x = 17$ in the array.

Steps:



Jump Search (1)

- **Jump Search:** It finds the position of a target element among the elements in a sorted array. It checks fewer elements by jumping a defined steps ahead.
- How It Works?

Jump search works as follows:

- ✓ Initially, it sorts the array if the array is not sorted already.
- ✓ Then it calculates the block size to be jumped $m = \sqrt{n}$ (generally), where n is the array size.
- ✓ It searches the sorted array and jumps based on the calculated block size.
- ✓ Performs the linear search when current element is greater than the previous element.
- ✓ Returns the target index once a match of the target element is found.

- **Time Complexity:** The time complexity of jump search is between the linear search O(n) and binary search O(log n).

Jump Search (2)

- Jump Search

Let's look at an example, suppose a sorted array is $A[0{:}n]$, search the element $x = 13$ in the array. Given $n = 6$ and $m = 2$.

Steps:

	1	7	9	13	17	23
Index position →	0	1	2	3	4	5

Iteration 1: Jump 0
Set $i = 0, j = m$, finds $x > A[i]$ & $A[m]$

1	7	**9**	13	17	23
$i = 0$		$j = m = 2$			

Iteration 2: Jump 1
Set $i = m$, j $= 2m$, finds $A[m] < x < A[2m]$

1	7	**9**	13	**17**	23
		$i = m$		$j = 2m$	

Start: Linear search *from* $i = km$, k is the number of jumps

1	7	**9**	13	**17**	23

$i = km$

Iteration 1:
Set $i = km$, $A[km]$ is not equal to $x = 13$

$A[km] = 9$

1	7	**9**	13	**17**	23

Iteration 2:
Set $i = km + 1$, $A[km + 1]$ is equal to $x = 13$. Return $A[km + 1]$ and stop searching

$A[km + 1] = 13 = x$

1	7	**9**	**13**	**17**	23

Search Algorithms Implementation in Python (1)

- **Linear Search Algorithm in Python:**

```python
# Linear Search in Python:
# Define the function of linear serach:

def linearSearch(arr, n, x): # n is the size of the array and
                             # x is the element to search
    for i in range(0,n):
        if arr[i] == x:
            return i
    return False

# Define the array:
arr = [5, 7, 3, 13, 12]
x = 3 # Define the element to serach
n = len(arr) # Find the length of the array

# Call the serach function 'linearSearch' here:
result = linearSearch(arr, n, x)

# Print the outcomes of the linear search
print('Output:\n')
if(result == False):
    print("The element has not found")
else:
    print("The element has found at index:", result)
```

```
Output:

The element has found at index: 2
```

Search Algorithms Implementation in Python (2)

❑ Binary Search Algorithm in Python:

```python
# Binary Search in Python:
# Define the binary serach function:
def binarySearch(arr, x, lowIndex, highIndex):
    # x is the element to serach, lowIndex and highIndex are the
    # lowest and highest indices of the array arr
    # Repeat until the pointers low and high meet each other
    while lowIndex <= highIndex:
        midIndex = lowIndex + (highIndex - lowIndex)//2
        if arr[midIndex] == x:
            return midIndex
        elif arr[midIndex] < x:
            lowIndex = midIndex + 1
        else:
            highIndex = midIndex - 1
    return False

# Define the array
arr = [1, 7, 13, 17, 26, 31]
x = 13 # Define the element to serach

# Call the function here and assign the output of the function to 'result'
result = binarySearch(arr, x, 0, len(arr)-1)
# Print the output of the binary search
print("Output:\n")
if result != False:
    print("The element has found at index: " + str(result))
else:
    print("The element has not found in the array.")
```

```
Output:

The element has found at index: 2
```

Search Algorithms Implementation in Python (3)

❑ Jump Search Algorithm in Python:

```python
# Jump Search in Python
# Import the math library to do maths
import math # To use squared root and floor function

# Define the jump serach function
def jumpSearch(arr, n, x):
   # Define the steps (block size) to be jumped/skipped
    steps = math.floor(math.sqrt(n))
    # Search the block where the element is
    previous = 0
    while arr[int(min(steps, n)-1)] < x:
        previous = steps
        steps += math.floor(math.sqrt(n))
        if previous >= n:
            return False
    # Start a linear search for x, search from the previous index
    while arr[int(previous)] < x:
        previous += 1
        # If we reached at the end of array and the element has not found
        if previous == min(steps, n):
            return False
    # If the element has found
    if arr[int(previous)] == x:
        return previous
    return False
```

```python
# Define the array
arr = [1, 7, 13, 17, 21, 23, 37, 41, 45]
x = 13  # Define the element to search
n = len(arr) # Define the length of the array

# Call the function 'jumpSearch' and assign the output to 'result'
result = jumpSearch(arr, n, x)

# Print the outcome of the jump search
print('Output:\n')
if result != False: # != is the not equal to
    print("The element has found at index:",result)
else:
    print("The elment has not found in the array.")
```

```
Output:

The element has found at index: 2
```