

Elementary Algorithms

CSE1303 Part A

Data Structures and Algorithms

Overview

- Selection Sort
- Insertion Sort
- Linear Search.
- Binary Search.
- Growth Rates.
- Implementation.

Selection Sort

First find the largest element in the array and exchange it with the element in the last position, then find the second largest element in array and exchange it with the element in the second last position, and continue in this way until the entire array is sorted.

Selection Sort

1	5	0	2	6	3	4
---	---	---	---	---	---	---

1	5	0	2	4	3	6
---	---	---	---	---	---	---

1	3	0	2	4	5	6
---	---	---	---	---	---	---

1	2	0	3	4	5	6
---	---	---	---	---	---	---

...

Insertion Sort

The items of the array are considered one at a time, and each new item is inserted into the appropriate position relative to the previously sorted items.

Insertion Sort

1	5	0	2	6	3	4
---	---	---	---	---	---	---

1	0	5	2	6	3	4
---	---	---	---	---	---	---

0	1	5	2	6	3	4
---	---	---	---	---	---	---

0	1	2	5	6	3	4
---	---	---	---	---	---	---

0	1	2	5	3	6	4
---	---	---	---	---	---	---

0	1	2	3	5	6	4
---	---	---	---	---	---	---

...

Linear Search

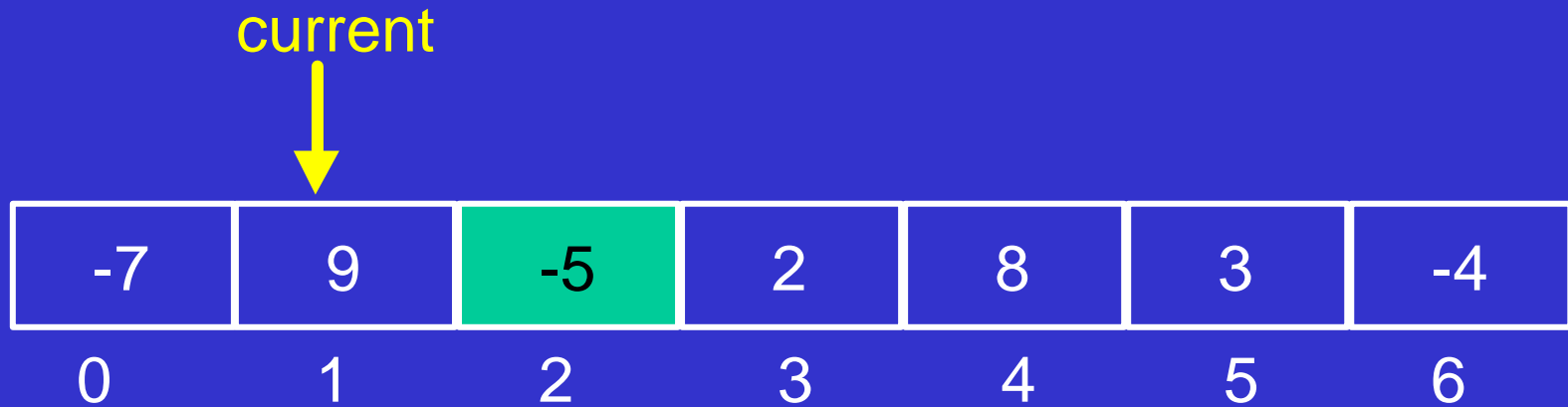
current



-7	9	-5	2	8	3	-4
0	1	2	3	4	5	6

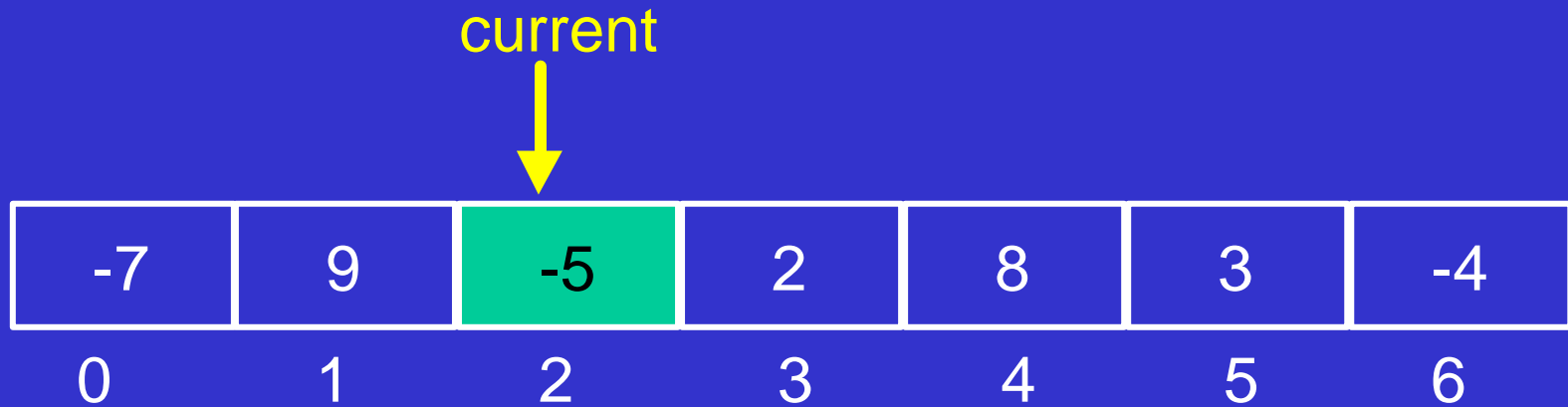
- Target is -5

Linear Search



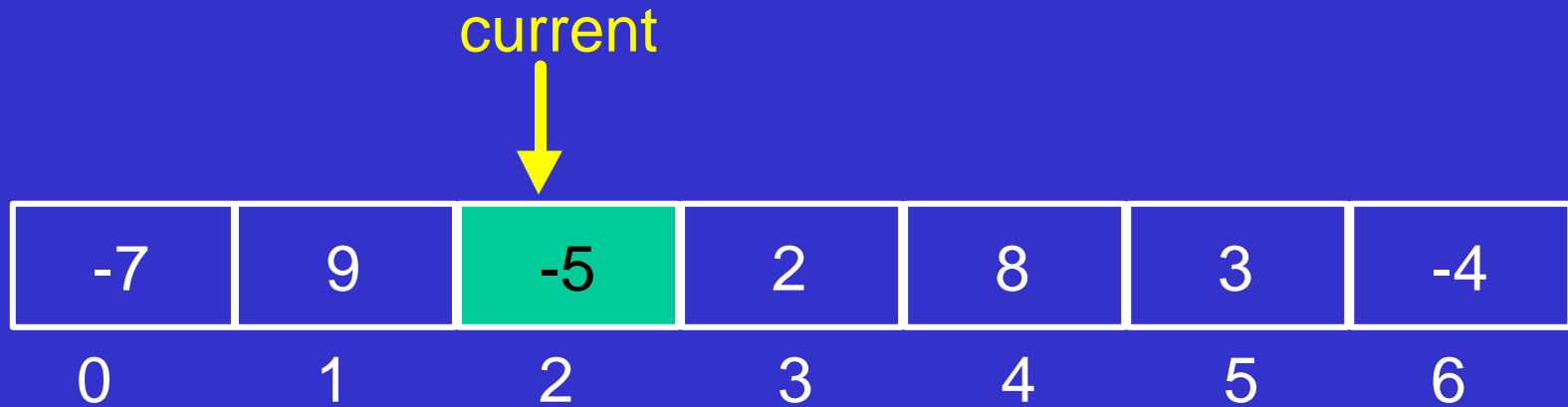
- Target is -5

Linear Search



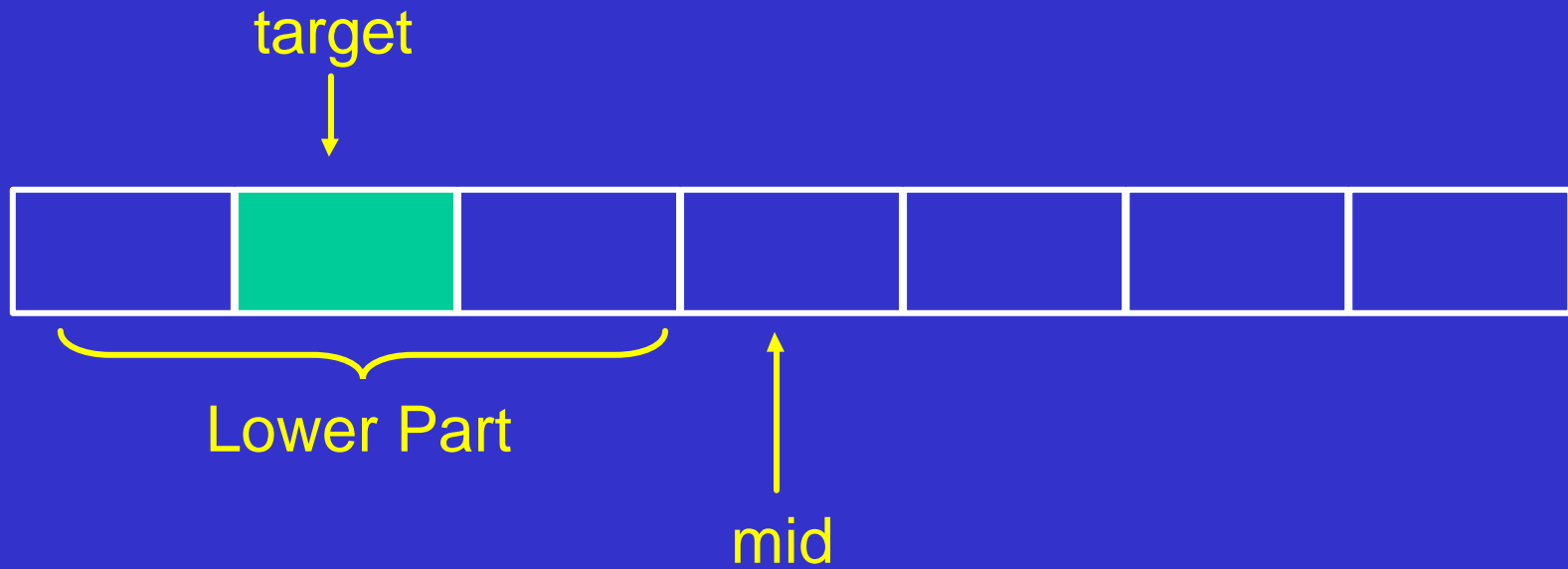
- Target is -5

Linear Search

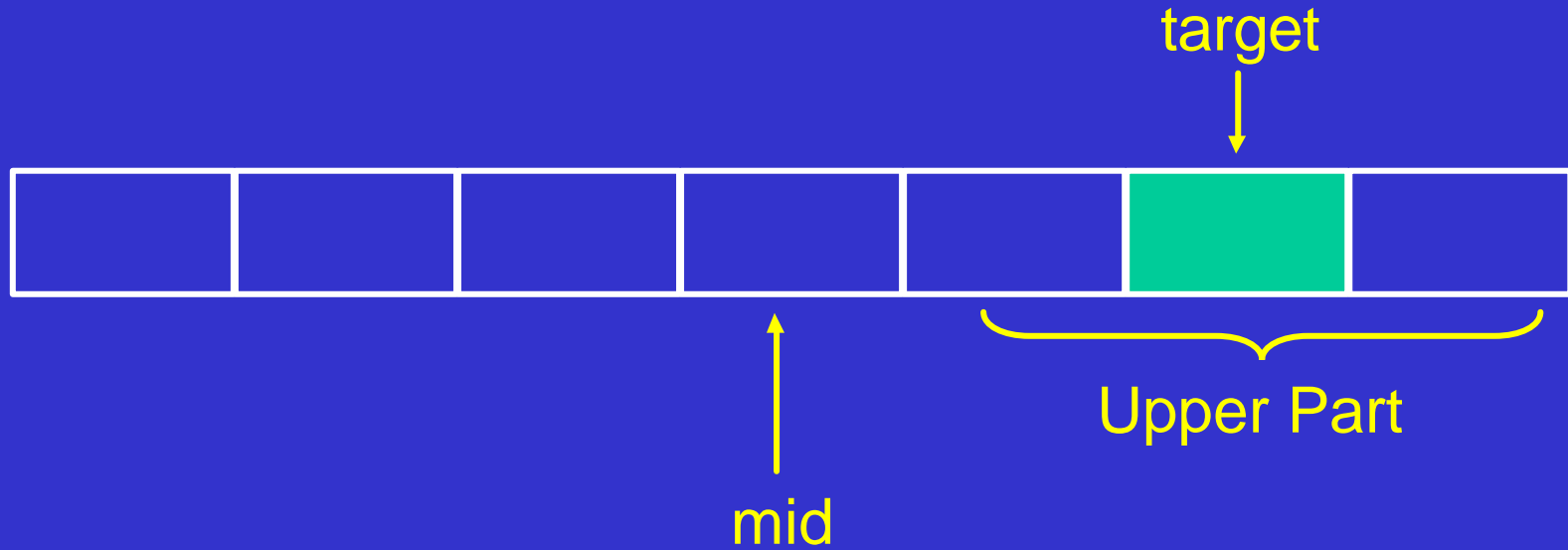


- Numbers can be in any order.
- Works for Linked Lists.

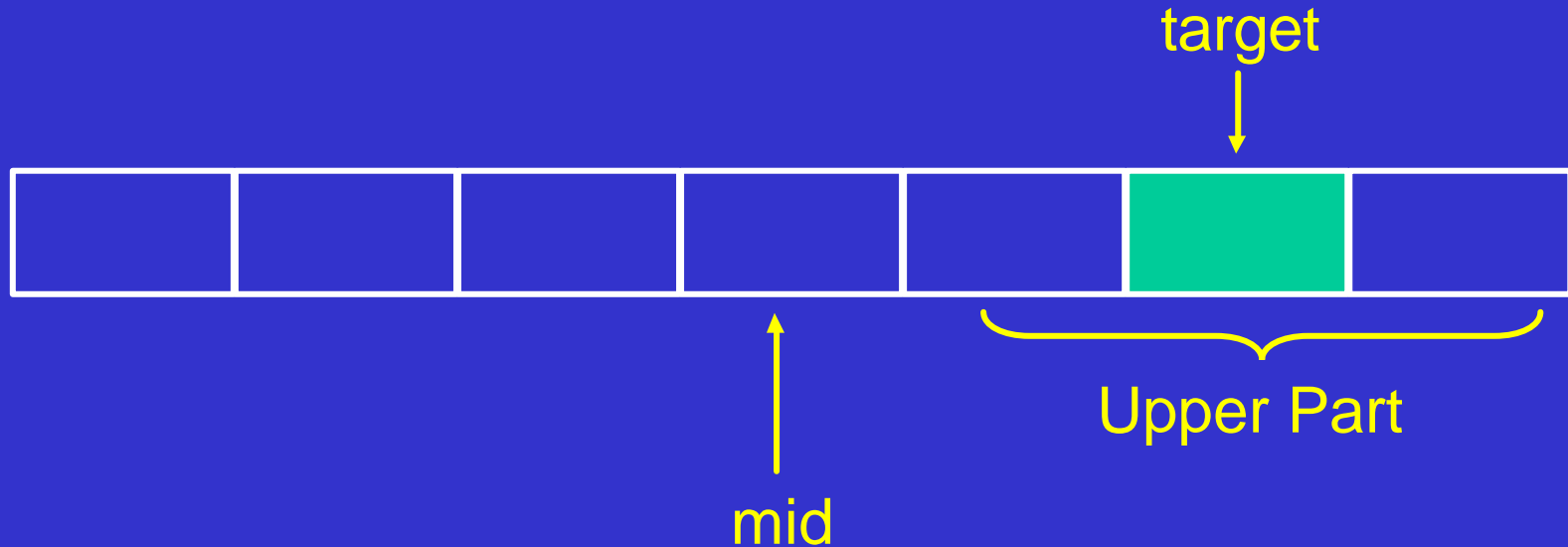
Binary Search



Binary Search

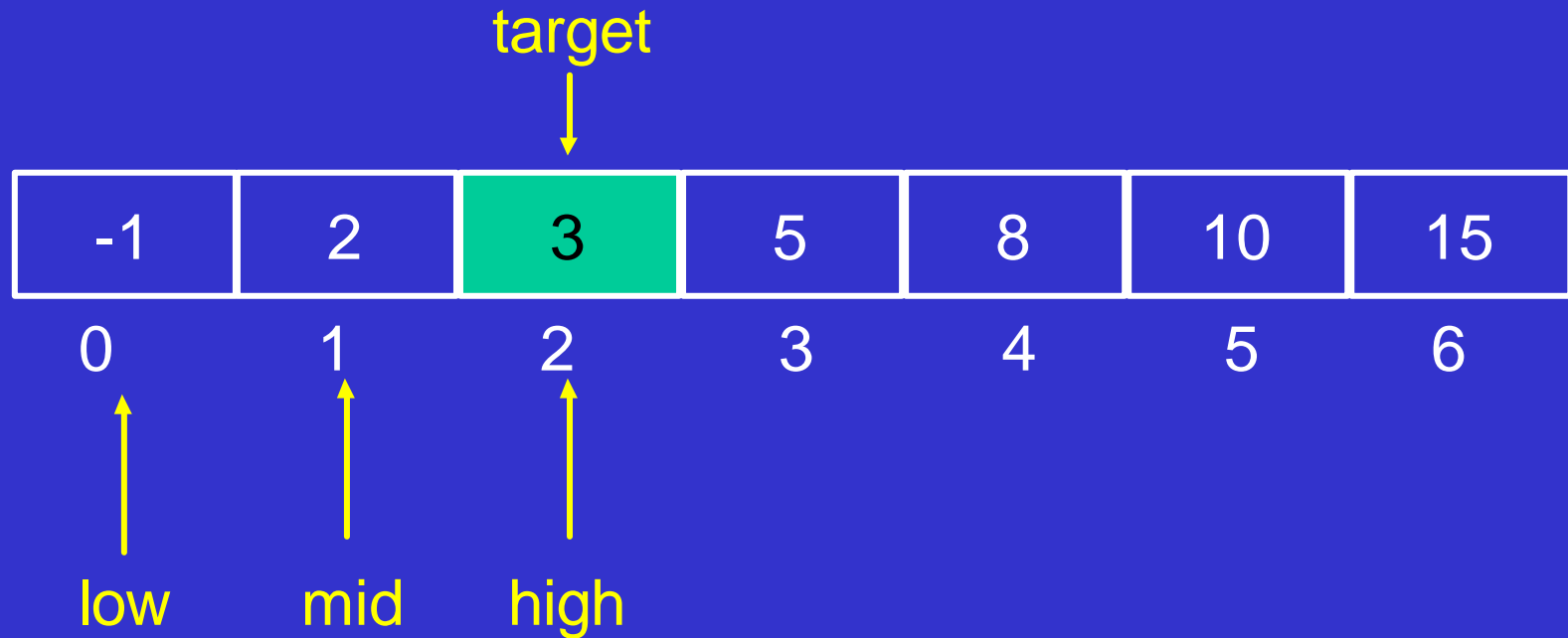


Binary Search



- Need
 - List to be sorted.
 - To be able to do random accesses.

Binary Search



Binary Search



Other Uses of Binary Search

- To find where a function is zero.
- Compute functions.
- Tree data structures.
- Data processing.
- Debugging code.



Recall:

Growth Rates

- Constant
- Logarithmic
- Linear
- $n \log(n)$
- Quadratic
- Cubic
- Exponential
- $O(1)$
- $O(\log(n))$
- $O(n)$
- $O(n \log(n))$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$

Selection Sort

First find the largest element in the array and exchange it with the element in the last position, then find the second largest element in array and exchange it with the element in the second last position, and continue in this way until the entire array is sorted.

Find where the Maximum is.

```
/*
 * Find the position of the maximum in
 * list[0],...,list[k]
 */

maxPosition = 0;

for (j = 1; j <= k; j++) {
    if (array[j] > array[maxPosition])
    {
        maxPosition = j;
    }
}
```

```
void selectionSort(float array[], int size)
{
    int j, k;
    int maxPosition;
    float tmp;

    for (k = size-1; k > 0; k--) {
        maxPosition = 0;

        for (j = 1; j <= k; j++) {
            if (array[j] > array[maxPosition])
            {
                maxPosition = j;
            }
        }
        tmp = array[k];
        array[k] = array[maxPosition];
        array[maxPosition] = tmp;
    }
}
```

```
void selectionSort(float array[], int size)
{
    int j, k;
    int maxPosition;
    float tmp;

    for (k = size-1; k > 0; k--) {
        maxPosition = 0;

        for (j = 1; j <= k; j++) {
            if (array[j] > array[maxPosition])
            {
                maxPosition = j;
            }
        }
        tmp = array[k];
        array[k] = array[maxPosition];
        array[maxPosition] = tmp;
    }
}
```

Insertion Sort

The items of the array are considered one at a time, and each new item is inserted into the appropriate position relative to the previously sorted items.

```
void insertionSort(float array[], int size)
{
    int j, k;
    float current;

    for (k = 1; k < size; k++)
    {
        current = array[k];
        j = k;

        while (j > 0 && current < array[j-1]) {
            array[j] = array[j-1];
            j--;
        }

        array[j] = current;
    }
}
```

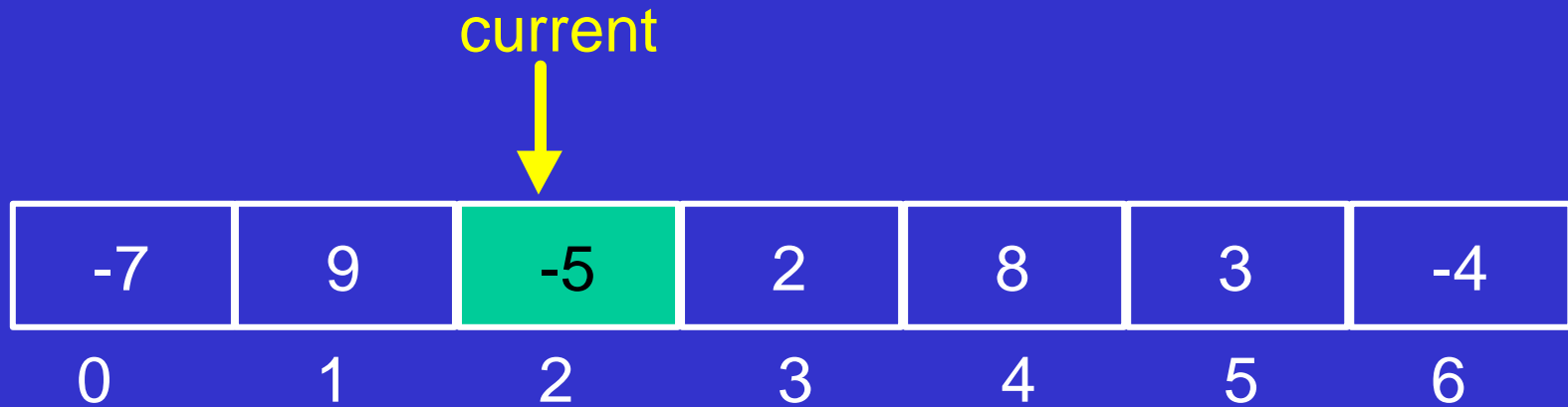
```
void insertionSort(float array[], int size)
{
    int j, k;
    float current;

    for (k = 1; k < size; k++)
    {
        current = array[k];
        j = k;

        while (j > 0 && current < array[j-1])
        {
            array[j] = array[j-1];
            j--;
        }

        array[j] = current;
    }
}
```

Linear Search



- Numbers can be in any order.
- Works for Linked Lists.

Linear Search

```
int linearSearch(float array[], int size, int target)
{
    int i;

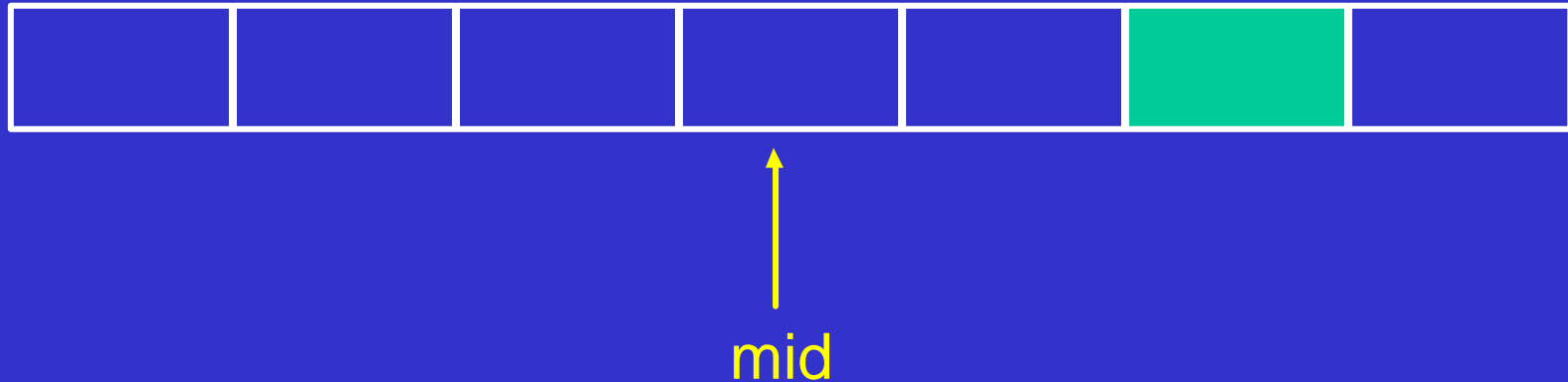
    for (i = 0; i < size; i++)
    {
        if (array[i] == target)
        {
            return i;
        }
    }
    return -1;
}
```

Linear Search

```
int linearSearch(float array[], int size, int target)
{
    int i;

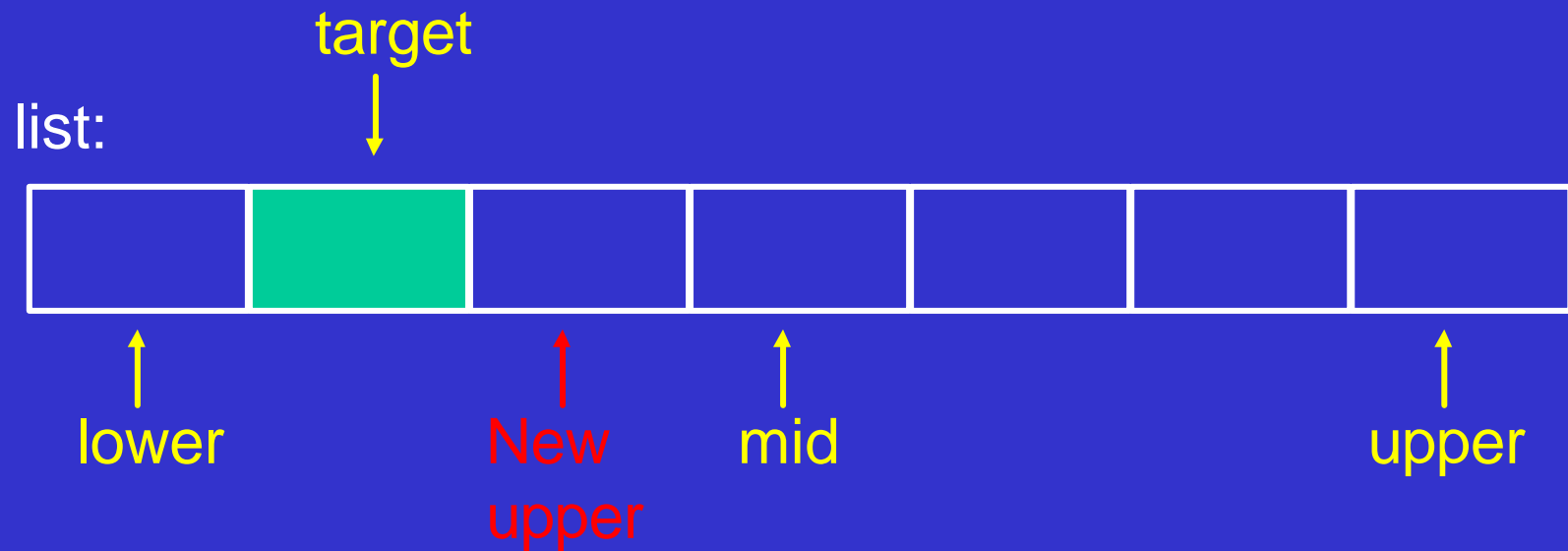
    for (i = 0; i < size; i++)
    {
        if (array[i] == target)
        {
            return i;
        }
    }
    return -1;
}
```

Binary Search

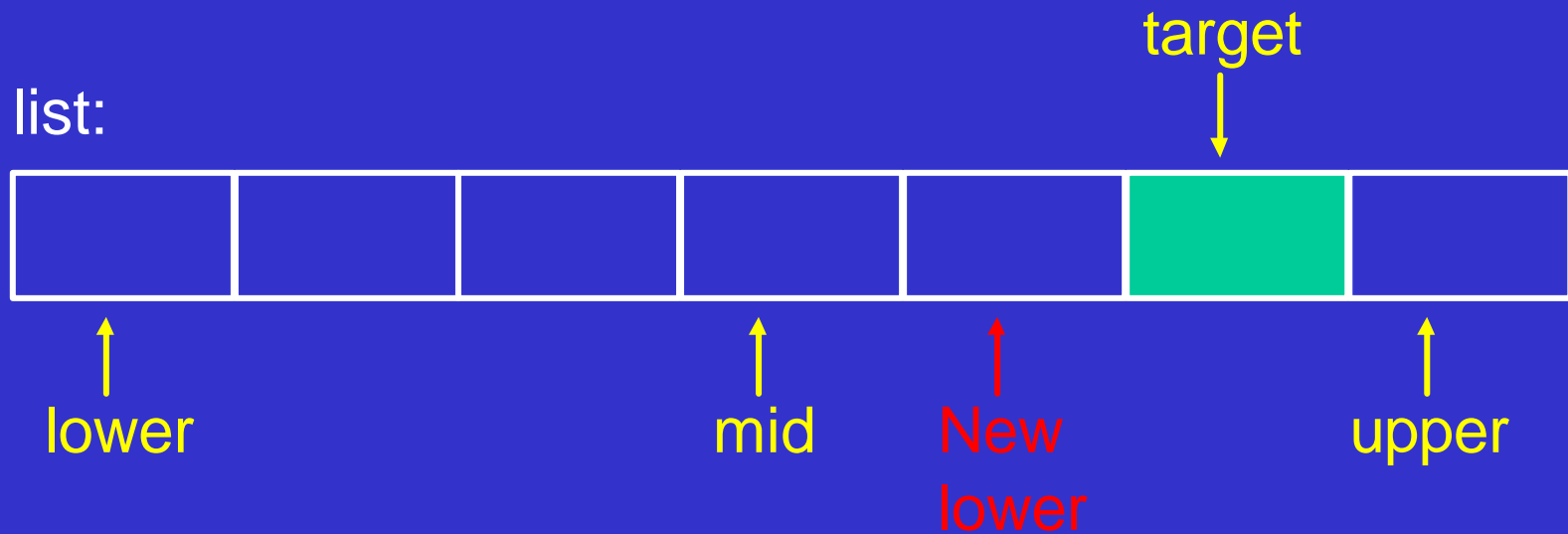


- Need
 - List to be sorted.
 - To be able to do random accesses.

Case 1: target < list[mid]



Case 2: $\text{list}[\text{mid}] < \text{target}$



```
int binarySearch(float array[], int size, int target)
{
    int lower = 0, upper = size - 1, mid;

    while (lower <= upper) {
        mid = (upper + lower)/2;
        if (array[mid] > target)
        {
            upper = mid - 1;
        }
        else if (array[mid] < target)
        {
            lower = mid + 1;
        }
        else
        {
            return mid;
        }
    }
    return -1;
}
```

```
int binarySearch(float array[], int size, int target)
{
    int lower = 0, upper = size - 1, mid;

    while (lower <= upper) {
        mid = (upper + lower)/2;
        if (array[mid] > target)
        {
            upper = mid - 1;
        }
        else if (array[mid] < target)
        {
            lower = mid + 1;
        }
        else
        {
            return mid;
        }
    }
    return -1;
}
```

The section where
the target could be found
halves in size each time

Comparison

	Array	Linked List
Selection Sort		
Insertion Sort		
Linear Search		
Binary Search		

Revision

- Selection Sort
- Insertion Sort
- Linear Search
- Binary Search

Preparation

- Read Section 3.2 and 3.4 in Kruse et al.