CSEN2001 Data Structures Module 4

Dr. D.V.N. Siva Kumar

Assistant Professor

CSE Department

GITAM University, Hyderabad Campus

Email: vdonthu@gitam.edu

Agenda

- Searching
- Sorting
- Hashing

Searching

- Searching is one of the frequent operations that is performed on arrays to search for a given values.
- Searching can be done using either **Linear search** or **Binary Search**.

Searching for an element in an Array

Linear Search:

- *It is a sequential search that starts searching at the starting element and goes through each element of a list or array until the desired element is found and it returns the index position where the search element is found.*
- *If not found, the search continues till the last element of an array and returns -1 indicating element is not found.*
- ***Runtime** complexity of linear search is **O(n),** where n is the number of elements in the given array or list.*

Steps of Linear search

Step 1: Read the search element from the user

Step 2: Compare, the search element with the first element in the list.

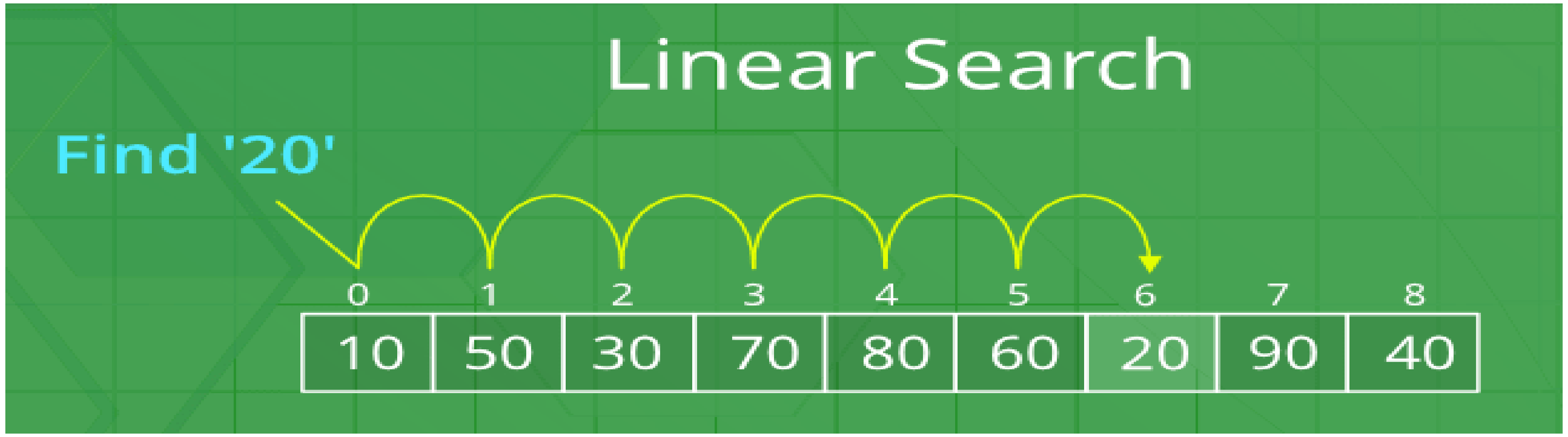
Step 3: If both are matching, then display "Given element found!!!" and terminate the function

Step 4: If both are not matching, then compare search element with the next element in the list.

Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in the list.

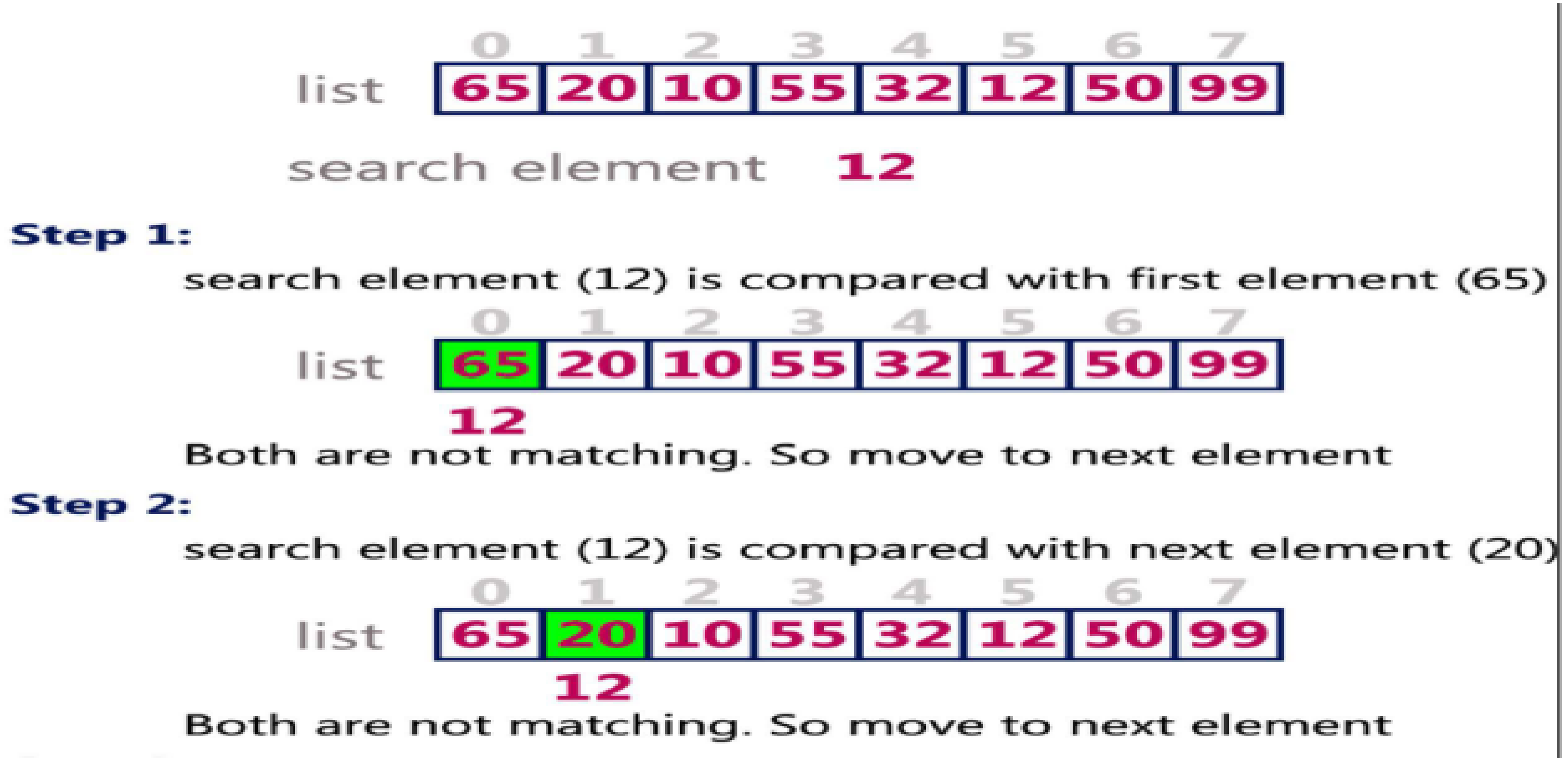
Step 6: If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Search for element 20 using linear search in the given array [10,50,,30, 70, 80, 60, 20, 90, 40]



It can be found in the above diagram that the **search element 20** is found at **index position 6.**
Ele

Another linear search example for searching 12 in a list


0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99

search element 12

Step 1:

search element (12) is compared with first element (65)

0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
12

Both are not matching. So move to next element

Step 2:

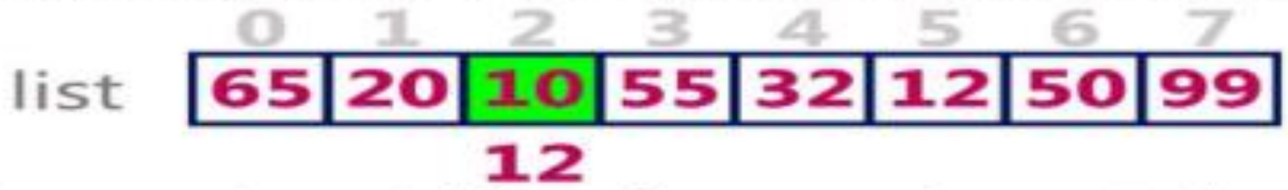
search element (12) is compared with next element (20)

0 1 2 3 4 5 6 7
list 65 20 10 55 32 12 50 99
12

Both are not matching. So move to next element

Step 3:

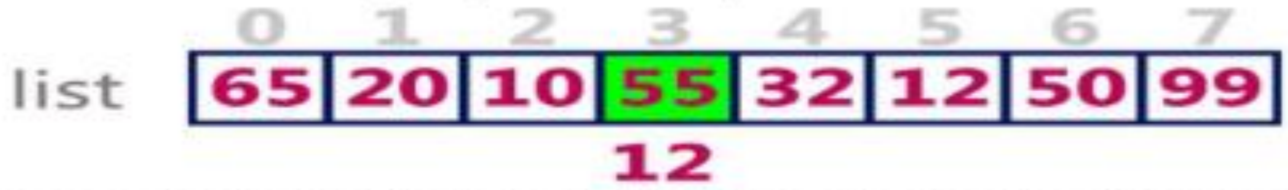
search element (12) is compared with next element (10)



Both are not matching. So move to next element

Step 4:

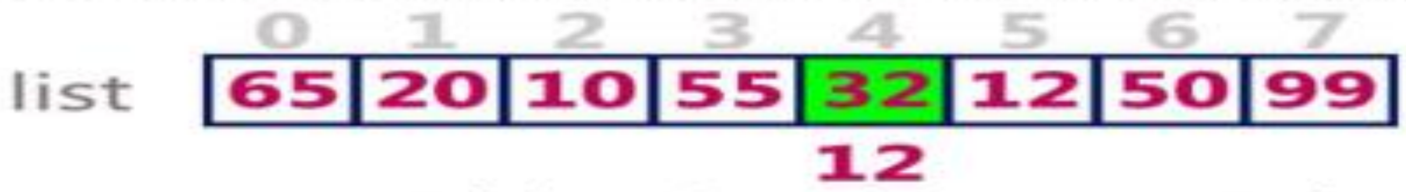
search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

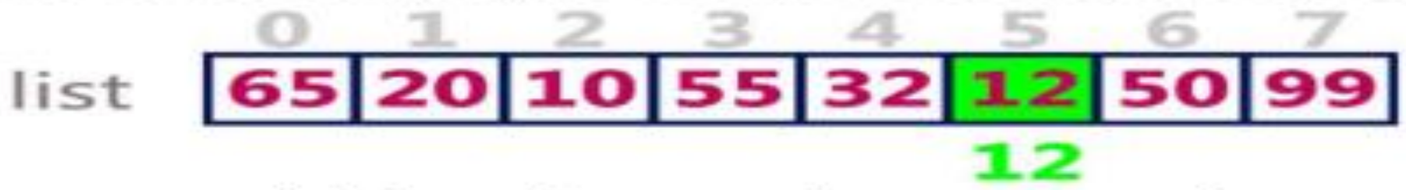
search element (12) is compared with next element (32)



Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)



Both are matching. So we stop comparing and display element found at index 5.

Linear Search program

Write a C program to search (linear) for a user element in the array of given elements and return its position where it is found.

If element is not found then return -1.

Test case1

Input:

Array elements: 1 15 8 7 16

Element to be searched: 8

Output:

It is found at index position 2

Test case2

Input:

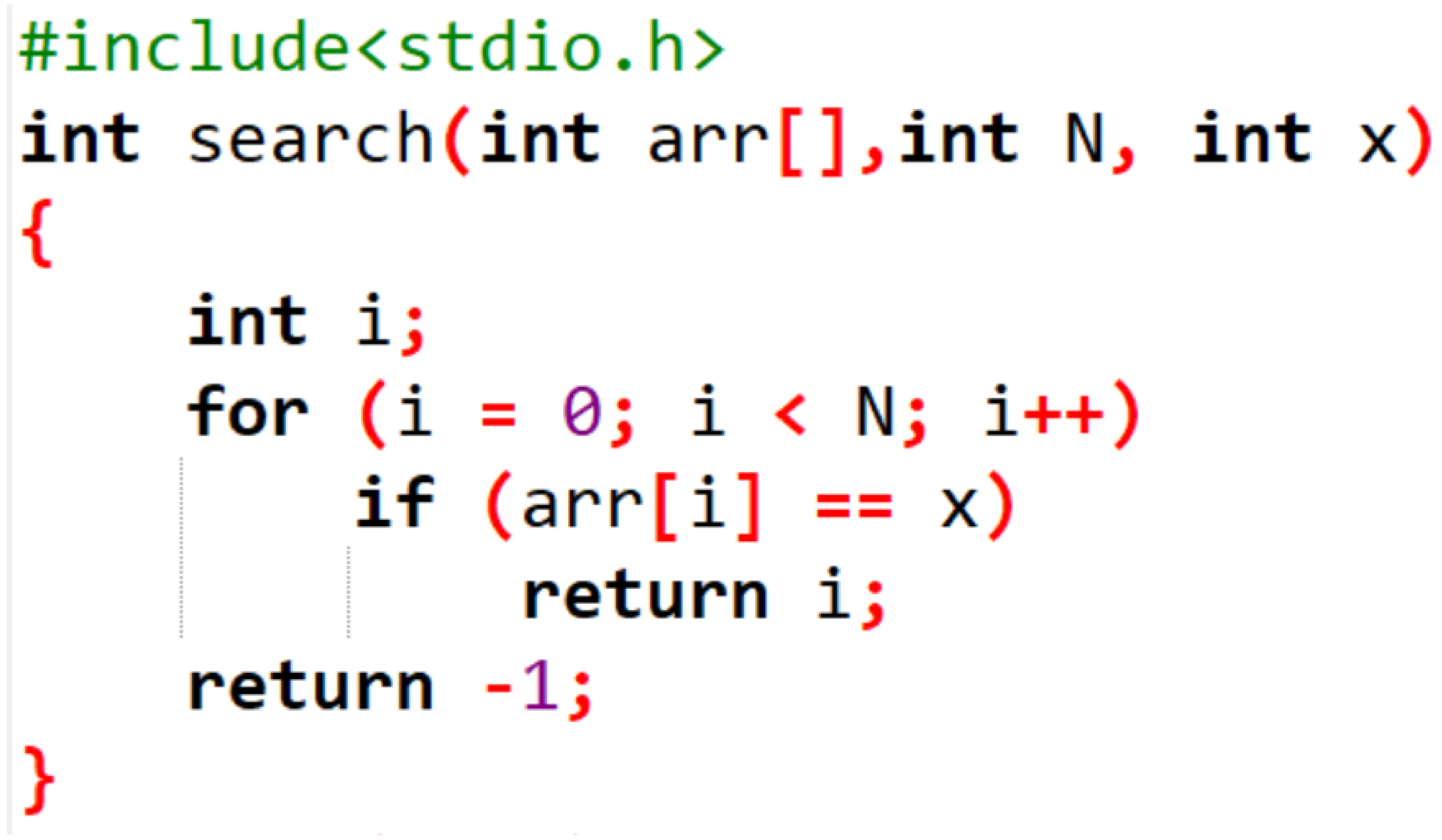
Array elements: 10 35 18 17 6

Element to be searched: 22

Output:

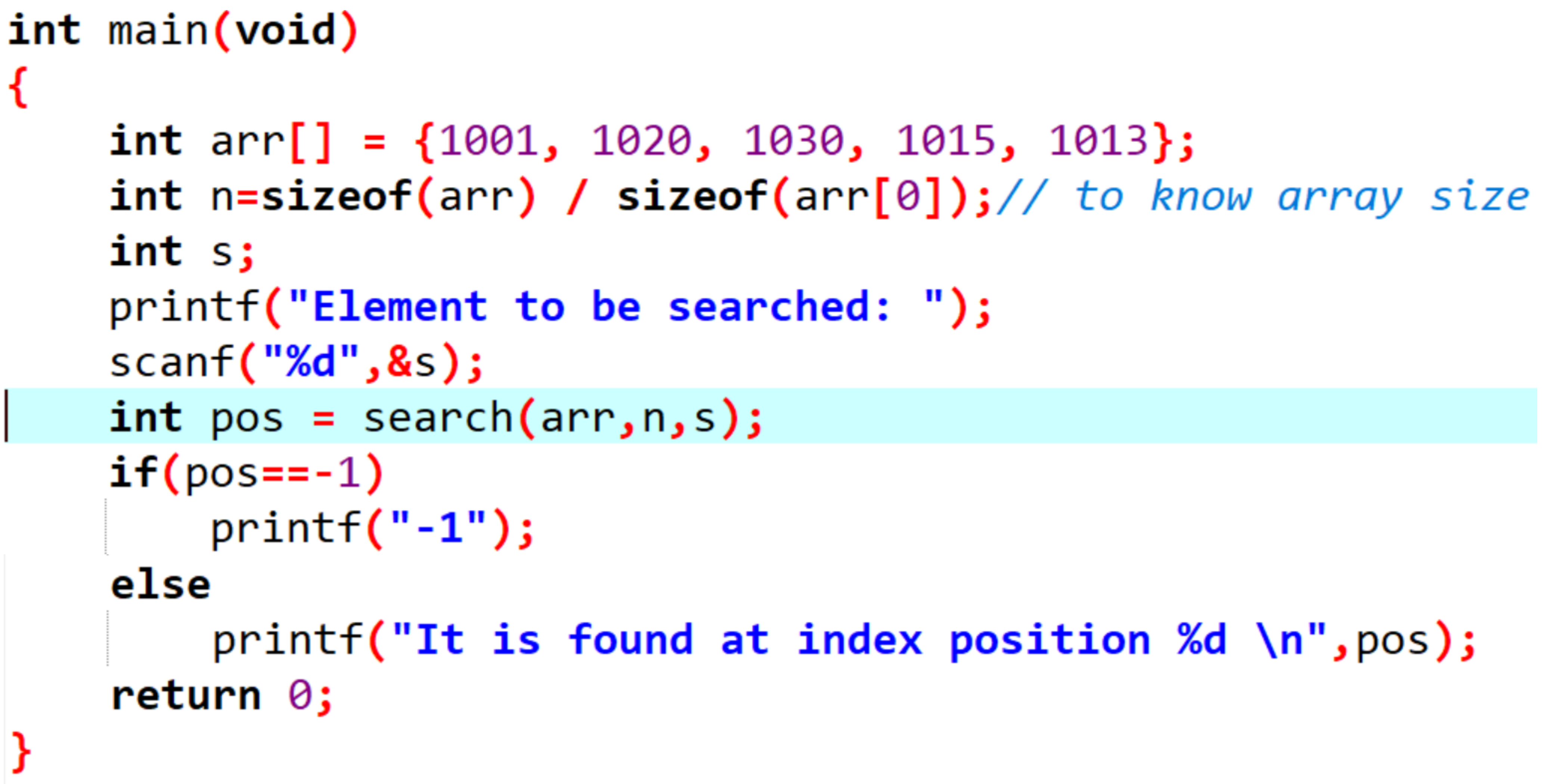
-1

Solution



```
#include<stdio.h>
int search(int arr[],int N, int x)
{
    int i;
    for (i = 0; i < N; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

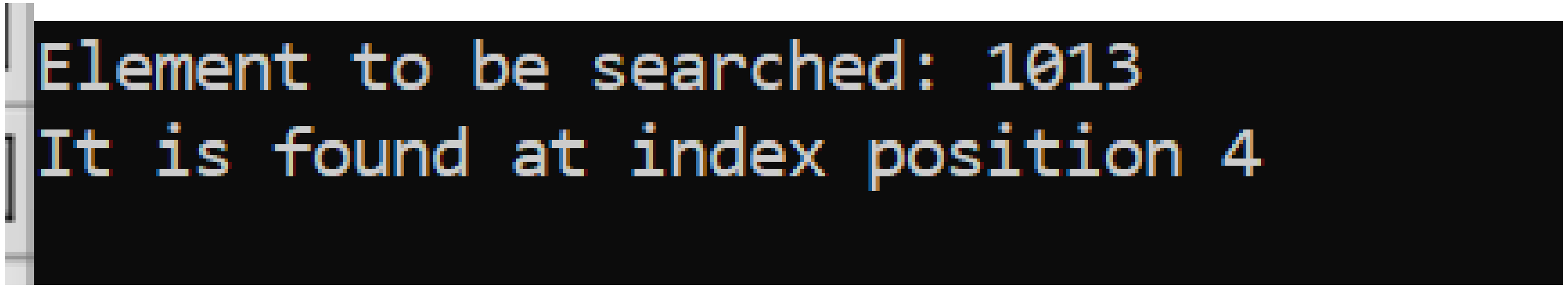
Solution (Cont...)



```c
int main(void)
{
    int arr[] = {1001, 1020, 1030, 1015, 1013};
    int n=sizeof(arr) / sizeof(arr[0]);// to know array size
    int s;
    printf("Element to be searched: ");
    scanf("%d",&s);
    int pos = search(arr,n,s);
    if(pos==-1)
        printf("-1");
    else
        printf("It is found at index position %d \n",pos);
    return 0;
}
```

Note: Array elements are direct given.

Output


```
Element to be searched: 1013
It is found at index position 4
```

Binary Search

- A binary search is an advanced type of search algorithm that allows us to check if the given element is present or not in a given array or list of elements more quickly in **O**(**log n)** time).
- Before performing Binary search, all the elements of the array or list need to be in sorted order first.
- Its core working principle involves dividing the sorted data into half until the required value is located and display index position of it to the user in the search result.

Steps for Performing Binary Search

1. First, find the middle element of the array

    The middle element of the array can be found using **(low+high)/2,** where low is the index position of the starting element and high is the position of the last element.

2**. Then, compare** the **mid element** with the **searching key** element.

3. There are possibilities that results this comparison:

    i) If the value of the searching key is **equal** to the middle item of the array

        then **return the index of the middle element**.

    ii) If the value of the searching key **is less than** the middle element of the array,

    then **search for a key element in the lower half of the array** (from low to mid-1).

    iii) If the value of the searching key is more than the middle element of the array,

    then **search for a key element in upper half** (from mid+1 to high)..

Note: Repeat the above steps until the search key element is found or otherwise return -1.

Binary Search Implementation

- Binary search can be implemented:
  - Iterative Method
  - Recursive Method

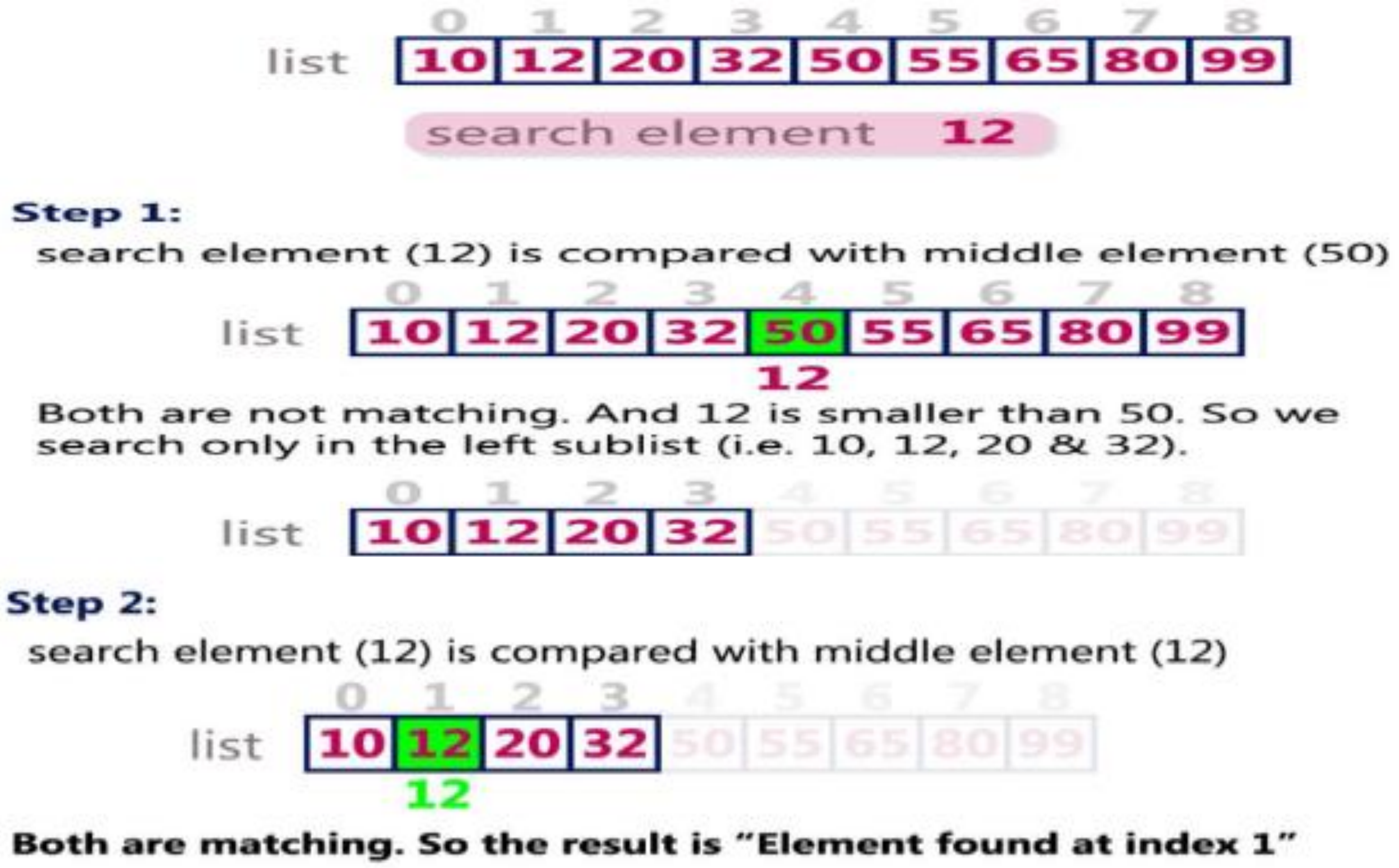
Binary Search Algorithm using Iterative Method

```
BinarySearch(arr, x, low, high)
    repeat until low = high
        mid=(low+high)/2
        if(x==arr[mid])
            return mid
        else if(x>arrr[mid])   // x is on the right side
            low=mid+1
        else                   // x is on the left side
            high=mid-1
```

Binary Search Implementation using Recursive Method

```
BinarySearch(arr, x, low, high)
    if (low>high)
        return False
    else
        mid=(low+high)/2
        if(x==arr[mid])
            return mid
        else if(x>arrr[mid])          // x is on the left side
            return BinarySearch(arr, x, mid+1, high)
        else                          // x is on the right side
            return  BinarySearch(arr, x, low, mid-1)
```

Binary Search Example: Search for 12 in a list of values [10, 12, 20, 32, 50, 55, 65, 80, 99]


0 1 2 3 4 5 6 7 8
list 10 12 20 32 50 55 65 80 99
search element 12

Step 1:

search element (12) is compared with middle element (50)

0 1 2 3 4 5 6 7 8
list 10 12 20 32 50 55 65 80 99
12

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

0 1 2 3 4 5 6 7 8
list 10 12 20 32 50 55 65 80 99

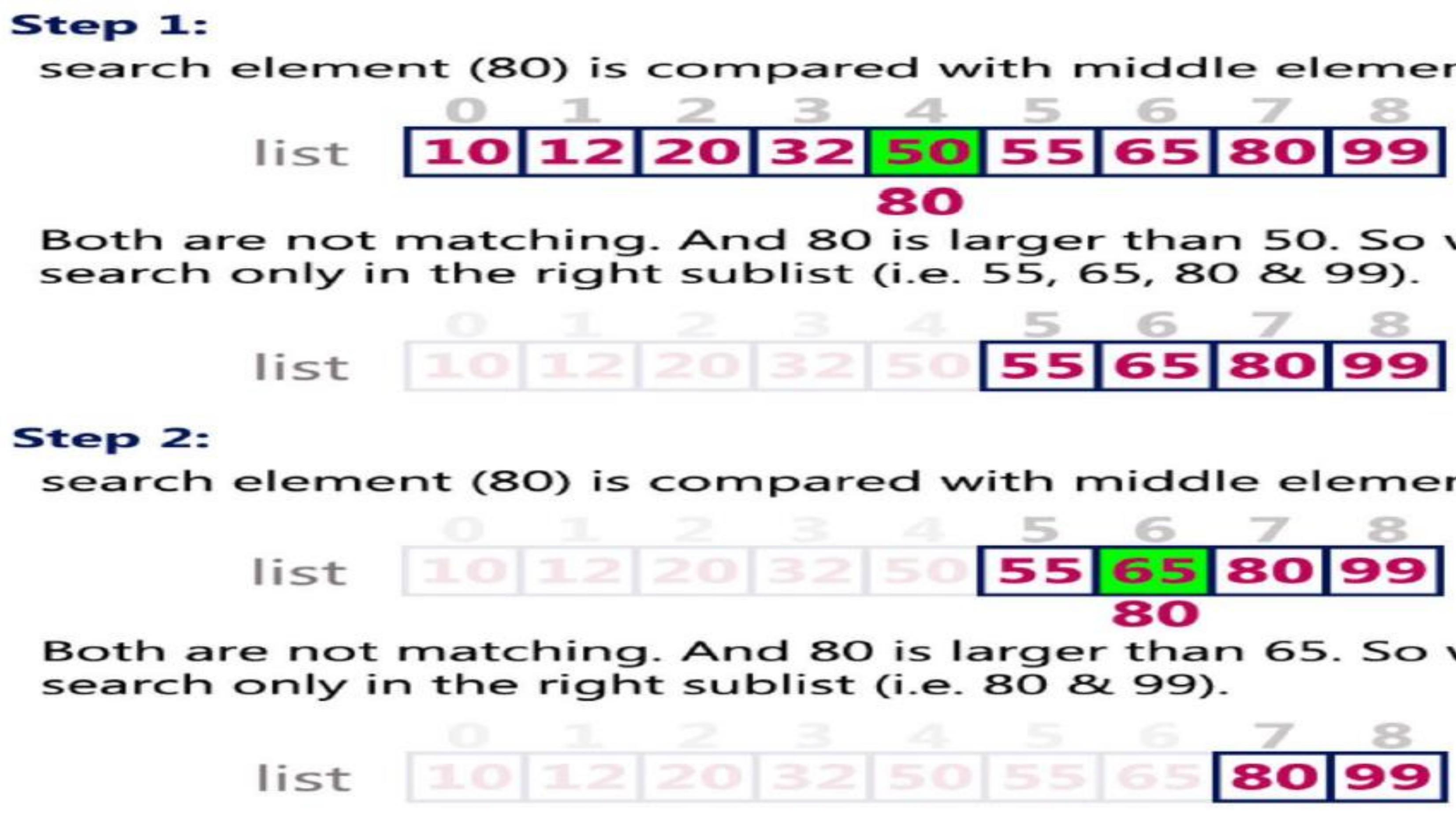
Step 2:

search element (12) is compared with middle element (12)

0 1 2 3 4 5 6 7 8
list 10 12 20 32 50 55 65 80 99
12

Both are matching. So the result is "Element found at index 1"

Another Binary Search Example: Search for 80 in a list of values [10, 12, 20, 32, 50, 55, 65, 80, 99]



Step 1:

search element (80) is compared with middle element (50)

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

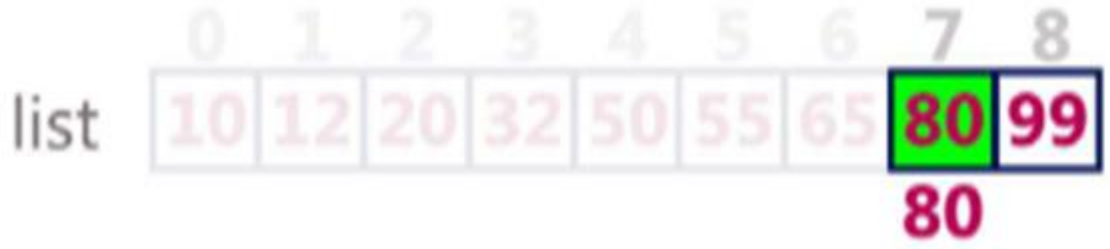
Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

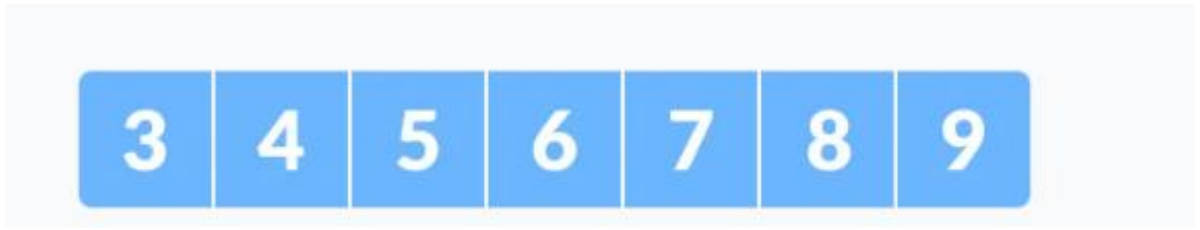
Step 3:

search element (80) is compared with middle element (80)

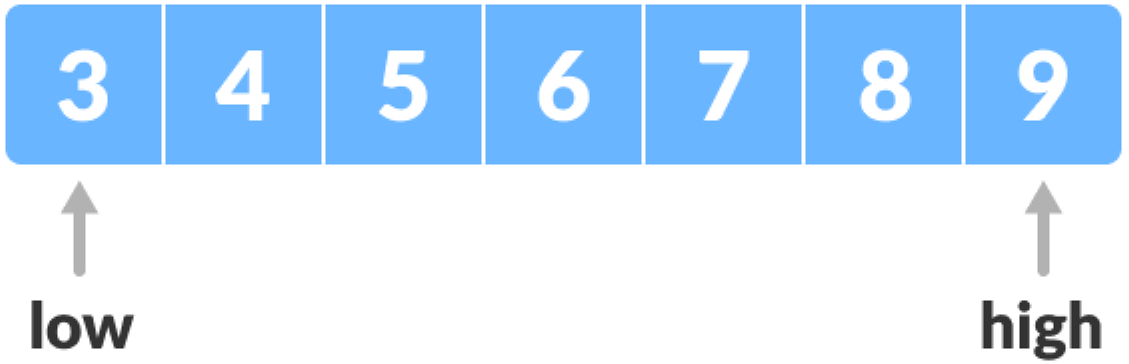


Both are matching. So the result is "Element found at index 7"

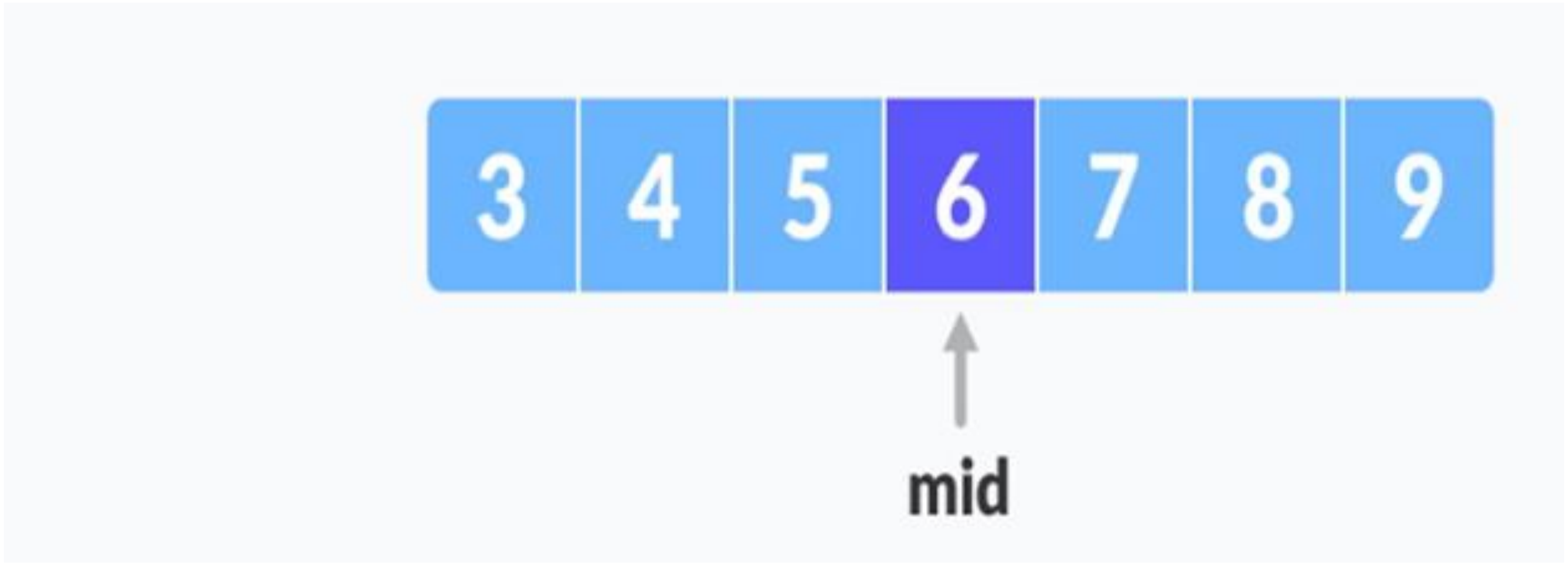
Another Binary Search Example

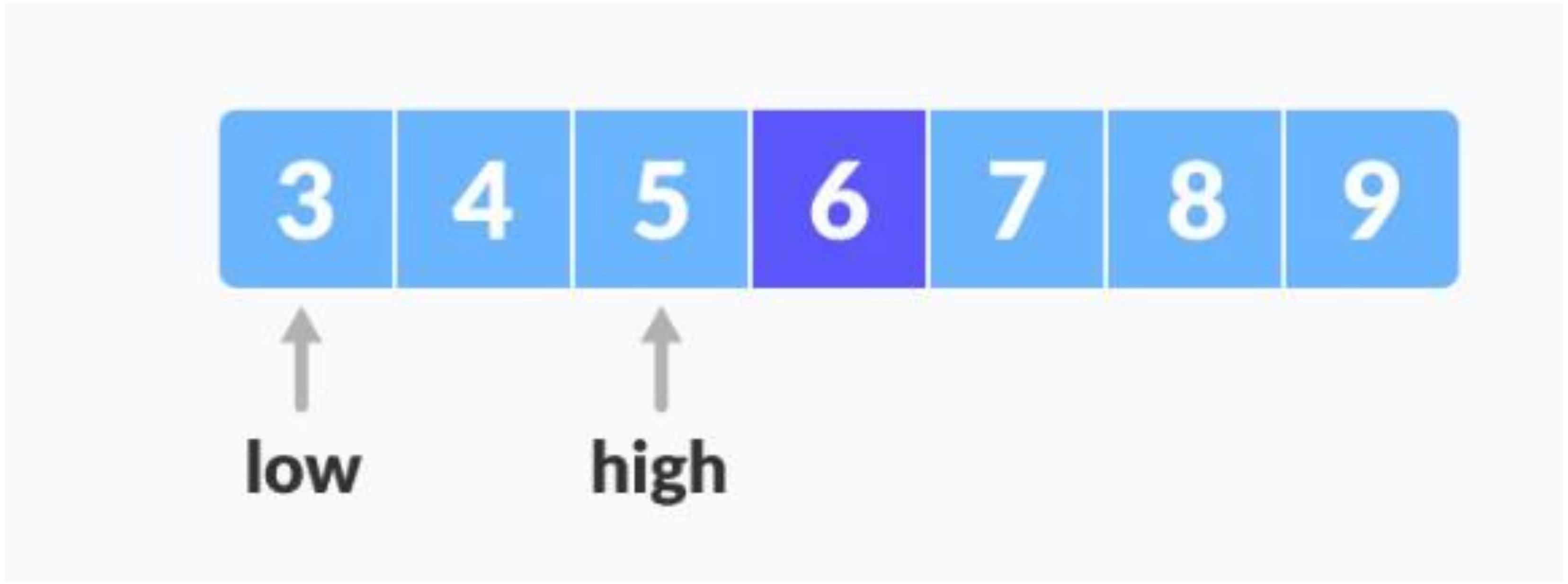


Let x = 4 **be the element to be searched.**

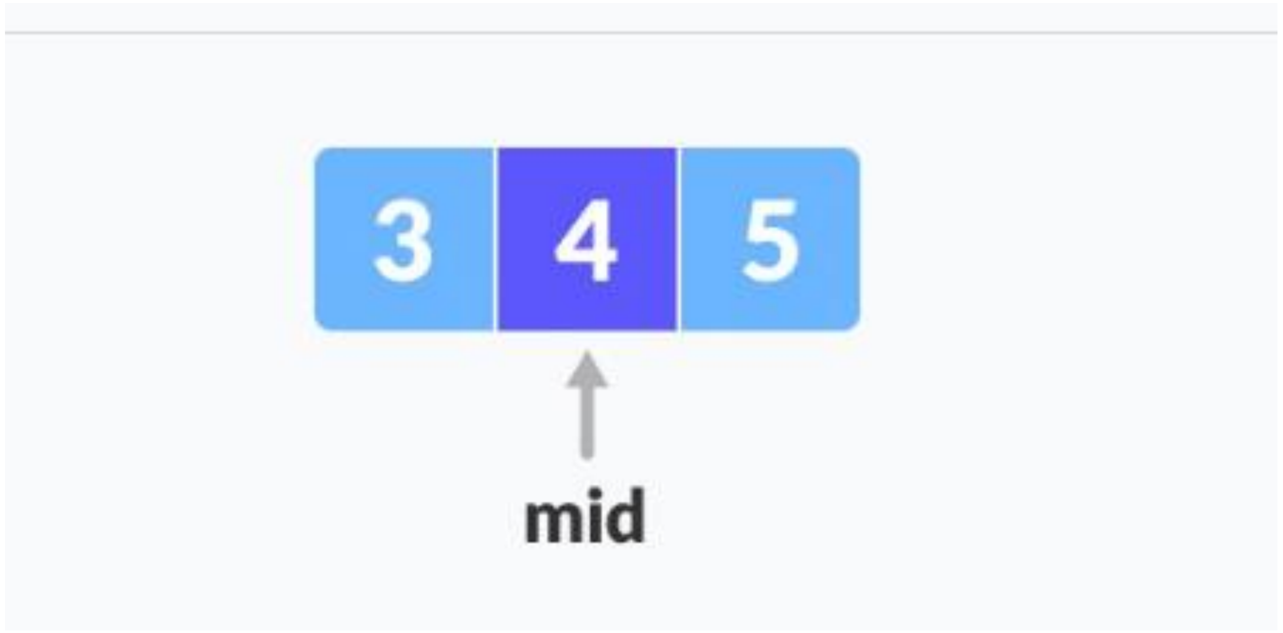


mid=(low+high)/2= (0+6)/2=3

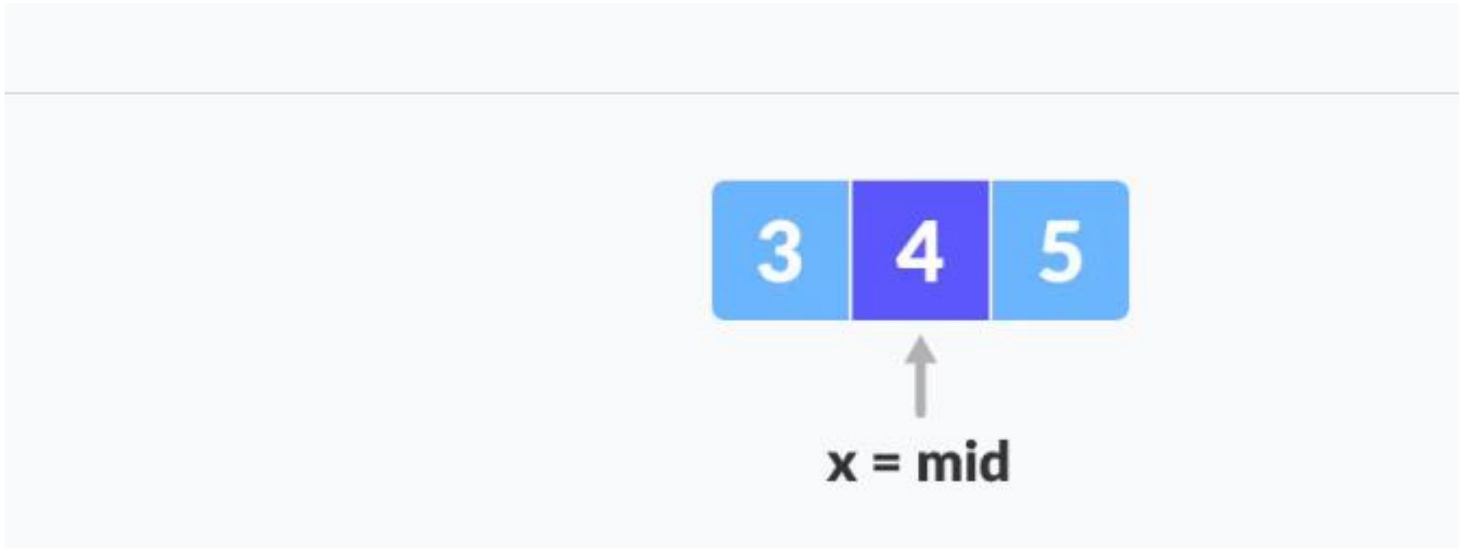

3
4
5
6
7
8
9
mid


3
4
5
6
7
8
9
low
high

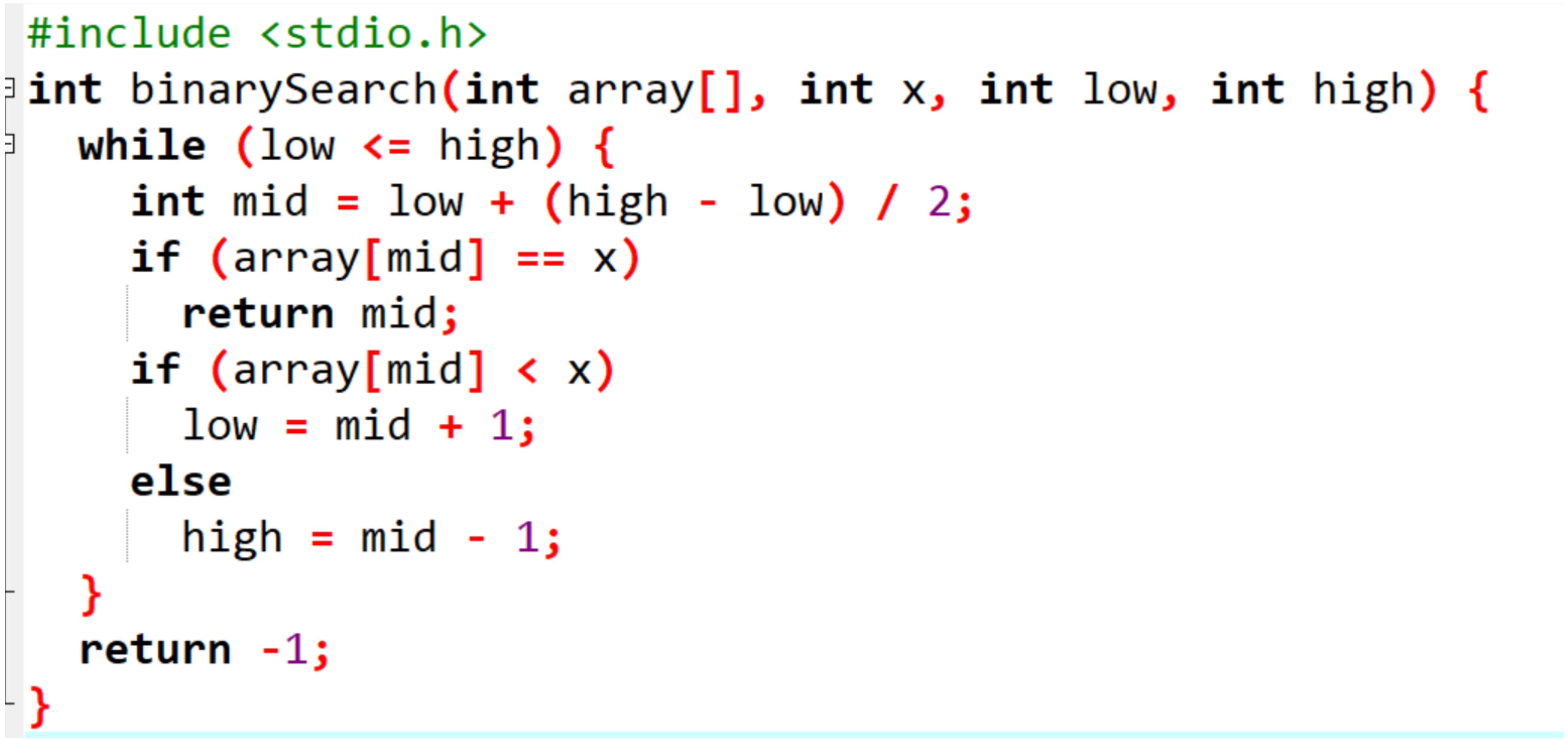
Repeat Steps (Find mid again)



Return position as element 4 is found



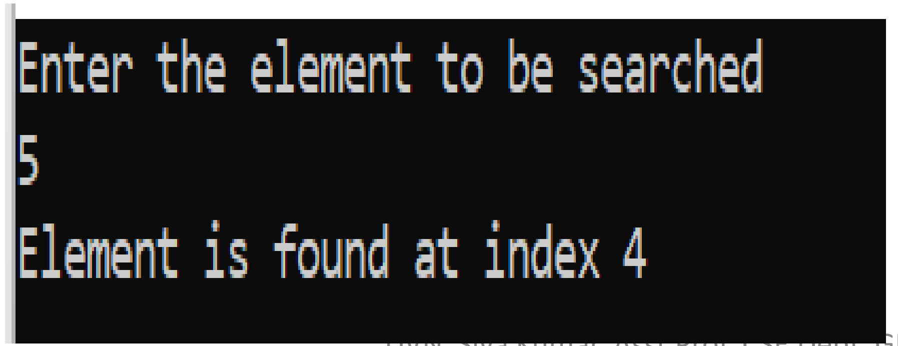
C Program for Binary Search using Iterative Method



```c
#include <stdio.h>
int binarySearch(int array[], int x, int low, int high) {
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (array[mid] == x)
      return mid;
    if (array[mid] < x)
      low = mid + 1;
    else
      high = mid - 1;
  }
  return -1;
}
```

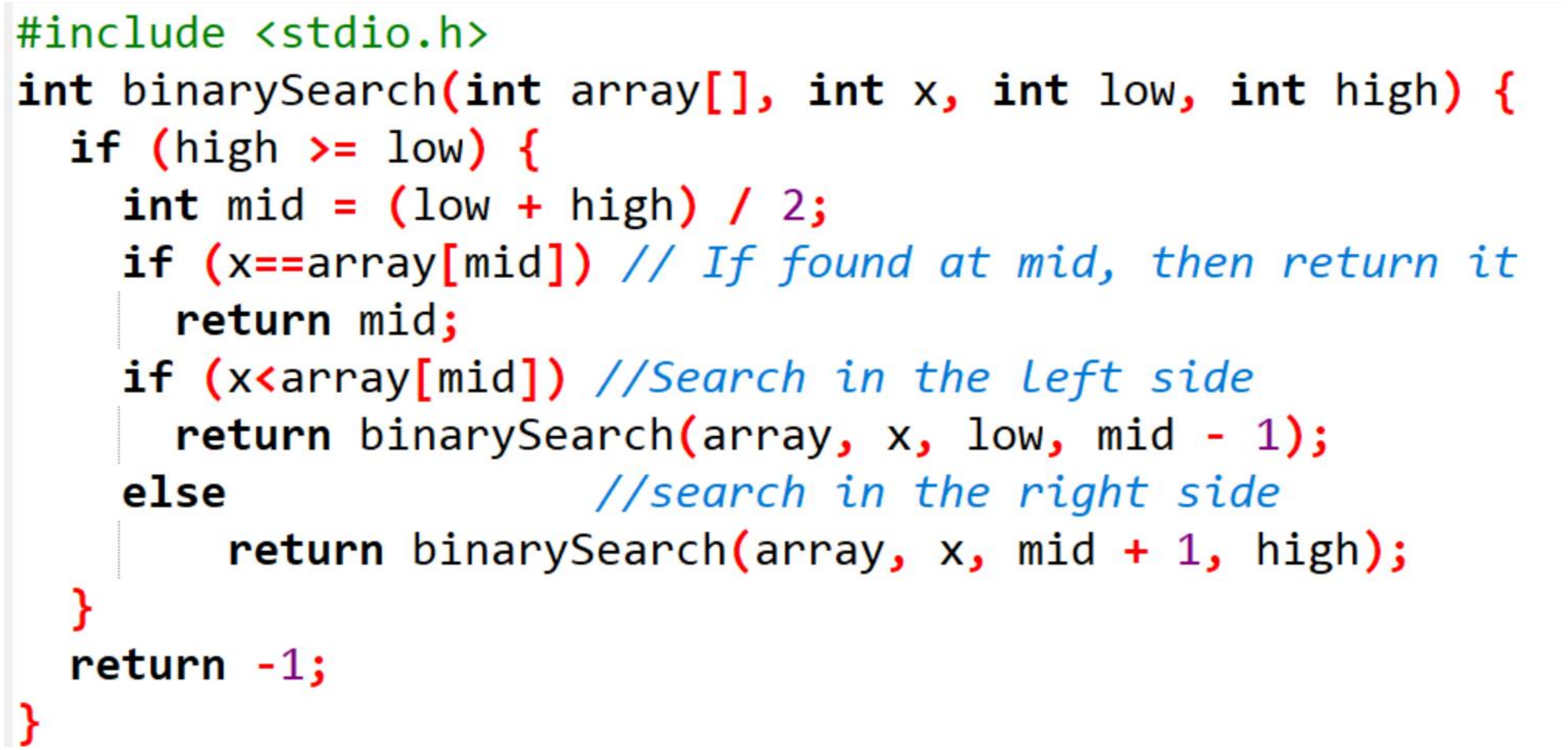
```c
int main() {
  int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x;
  printf("Enter the element to be searched\n");
  scanf("%d",&x);
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
  return 0;
}
```

Output



```
Enter the element to be searched
5
Element is found at index 4
```

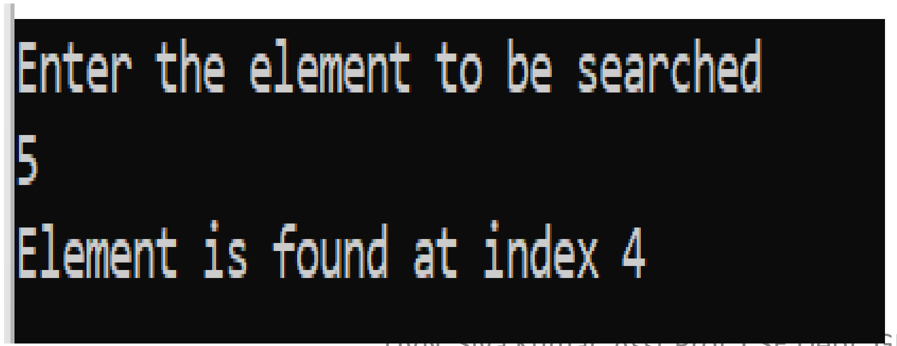
C Program for Binary Search using Recursive Method



```c
#include <stdio.h>
int binarySearch(int array[], int x, int low, int high) {
  if (high >= low) {
    int mid = (low + high) / 2;
    if (x==array[mid]) // If found at mid, then return it
      return mid;
    if (x<array[mid]) //Search in the left side
      return binarySearch(array, x, low, mid - 1);
    else              //search in the right side
        return binarySearch(array, x, mid + 1, high);
  }
  return -1;
}
```

```
int main() {
  int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x;
  printf("Enter the element to be searched\n");
  scanf("%d",&x);
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
  return 0;
}
```

Output



```
Enter the element to be searched
5
Element is found at index 4
```

Sorting

- Sorting is another frequently used operation on an arrays.
- It is the processing of arranging the data either in **Ascending order or Descending order** as per the requirement**.**
- Our syllabus includes the following sorting techniques:

  i) Insert sort

  ii) Merge Sort

  iii) Quick Sort

1. Insertion Sort

- Insertion sort is a sorting technique **that sorts a set of values by inserting values into an existing sorted list or array at appropriate positions.**
- **Insert sort** is also referred to as **in-place sorting** technique.
- The key idea of insertion sort is to build the final sorted array **one item at a time, with the new items being placed in their correct position relative to those already sorted.**
- Suppose an array **a** with **n** elements a[1], a[2], .., a[n] is in memory. The insertion sort algorithm scans array from a[1], a[2], …, a[k-1] and insert each value at appropriate position.
- The **Runtime** complexity of insertion sort in the worst case is **O($\boldsymbol{n^2}$)**

It means given any n input numbers, the insertion sort performs sorting in $\boldsymbol{n^2}$ steps.

Insertion Sort Process

Step 1 − If the element is the first one, it is already sorted.

Step 2 – Move to next element

Step 3 − Compare the current element with all elements in the sorted array

Step 4 – If the element in the sorted array is smaller than the current element, iterate to the next element. Otherwise, shift all the greater element in the array by one position towards the right

Step 5 − Insert the value at the correct position

Step 6 − Move to the Step 2 and Repeat until the complete list is sorted
Ref: https://yongdanielliang.github.io/animation/web/InsertionSortNew.html

Insertion Sort Algorithm

```
Insertion Sort(A[MAXSIZE], ITEM)
1.  Set k=1
2.  For k=1 to (n-1)
        Set temp=a[k]
        Set j=k-1
        While temp< a[j] and (j>=0) perform the following steps:
            Set a[j+1] = a[j]
                j=j-1
        [End of while loop]
        Assign the value of temp to a[j+1], i.e., a[j+1]=temp
    [End of for loop structure]
 3. Exit
```

Insertion sort example

Sort the below 7 elements using Insertion Sort

25, 15, 30, 9, 99, 20, 26.

Insertion Sort: Sort the below 7 elements 25, 15, 30, 9, 99, 20, 26.

Array of 7 elements is shown below

25	15	30	9	99	20	26
0	1	2	3	4	5	6

a[1]<a[0], the condition is true,
So interchange these two elements, then we get the below

Pass I:

15	25	30	9	99	20	26
0	1	2	3	4	5	6

a[2]<a[1], the condition is false,
So the position of elements remains same and the array is same

Pass II:

15	25	30	9	99	20	26
0	1	2	3	4	5	6

a[3] is less than a[2], a[1], and a[0],
so insert a[3] before a[0], and adjust these elements by moving one element right: we get the below array

Pass III:

9	15	25	30	99	20	26
0	1	2	3	4	5	6

a[4]<a[3], condition is false
no change is performed, so we get the same array.

Pass IV:

9	15	25	30	99	20	26
0	1	2	3	4	5	6

a[5] is less than a[4], a[3], and a[2]
therefore insert a[5] before a[2], and adjust these elements by moving one element right, we get the below arrray.

Pass V:

9	15	20	25	30	99	26
0	1	2	3	4	5	6

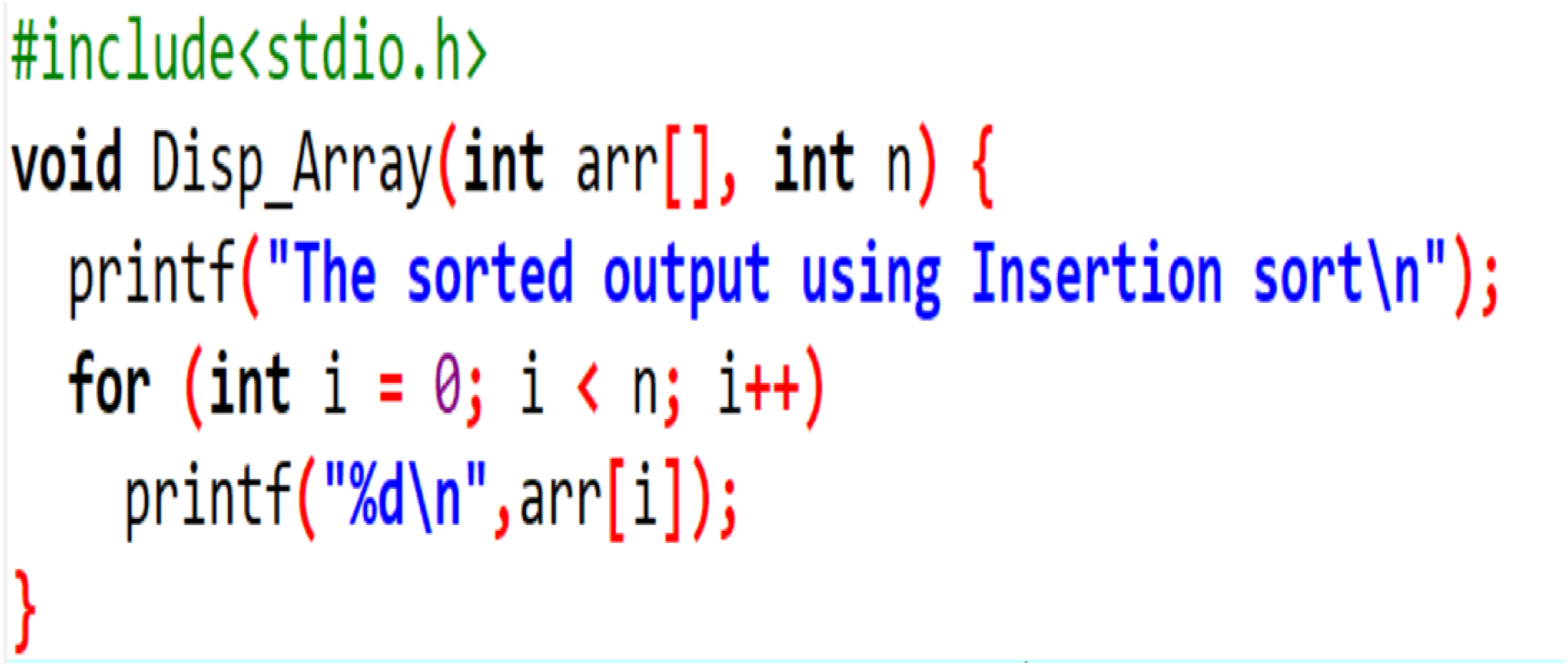
Now a[6] is less than a[5], and a[4], therefore insert a[6] before a[4], giving the below array as a result;

Pass Vi:

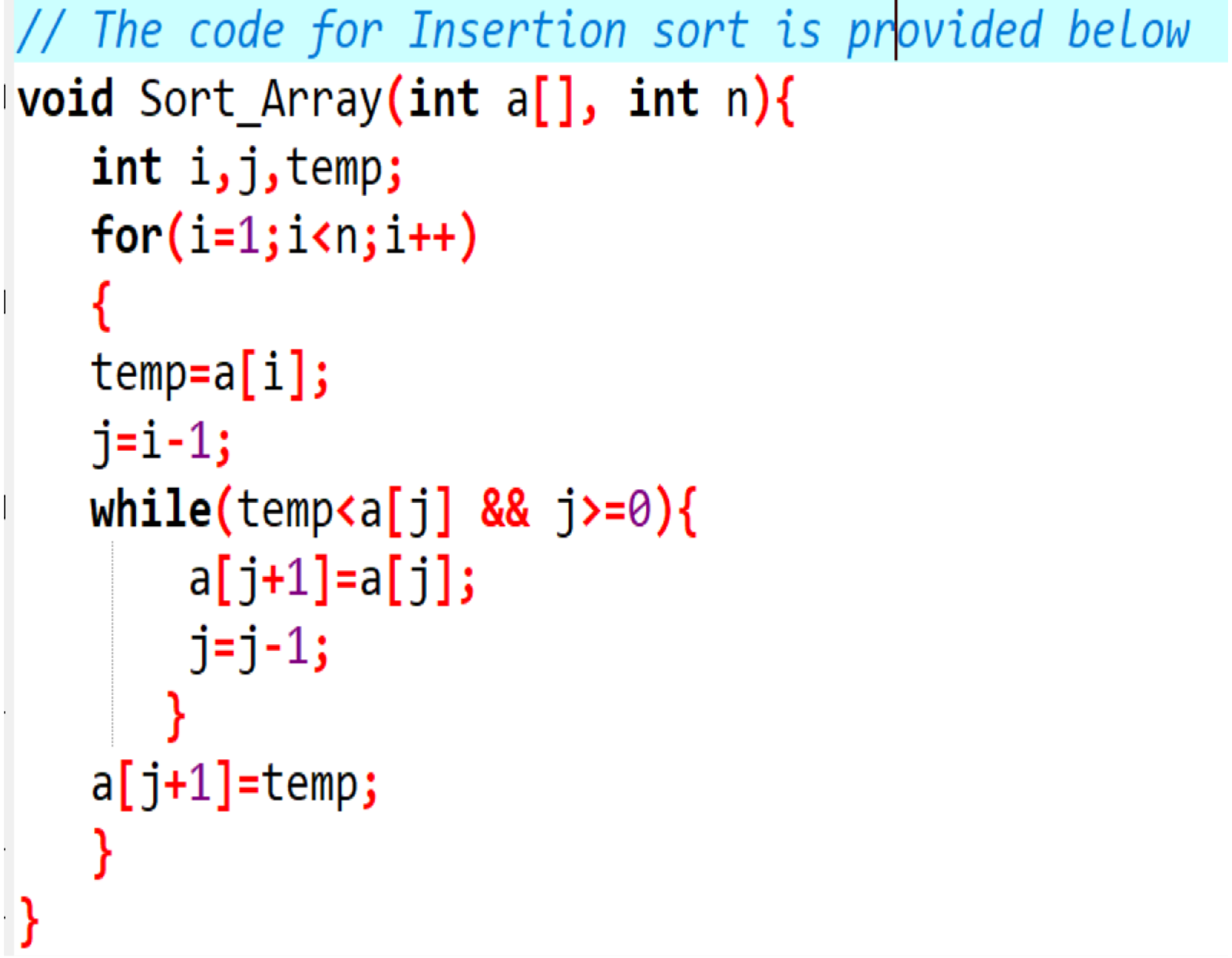
9	15	20	25	26	30	99
0	1	2	3	4	5	6

After the pass VI, we get array with sorted elements.

C Program for sorting array of n numbers using Insertion Sort



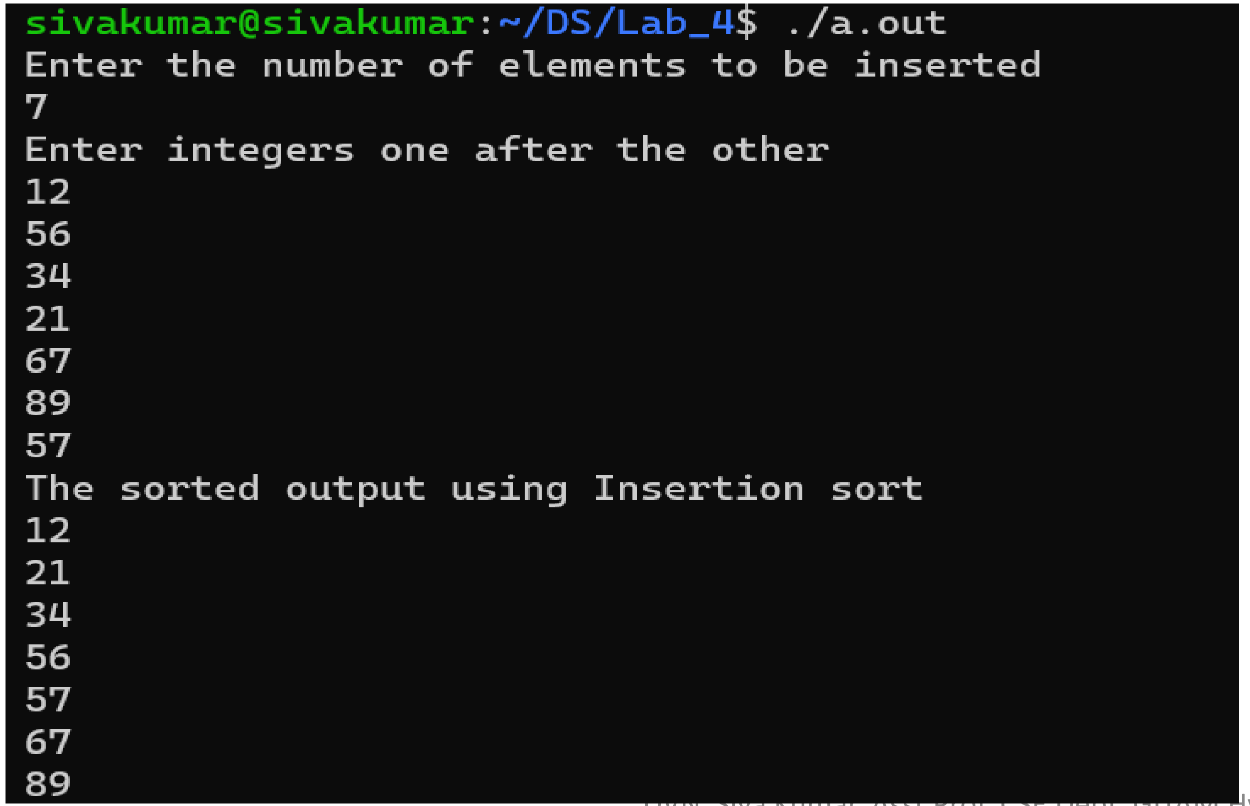
```
#include<stdio.h>
void Disp_Array(int arr[], int n) {
  printf("The sorted output using Insertion sort\n");
  for (int i = 0; i < n; i++)
    printf("%d\n",arr[i]);
}
```



```c
// The code for Insertion sort is provided below
void Sort_Array(int a[], int n){
    int i,j,temp;
    for(i=1;i<n;i++)
    {
    temp=a[i];
    j=i-1;
    while(temp<a[j] && j>=0){
        a[j+1]=a[j];
        j=j-1;
       }
    a[j+1]=temp;
    }
}
```

```c
int main() {
  int n, x;
  printf("Enter the number of elements to be inserted\n");
  scanf("%d",&n);
  int arr[n];
  printf("Enter integers one after the other\n");
  for (int i = 0; i < n; i++)
    scanf("%d", &arr[i]);
  Sort_Array(arr,n);
  Disp_Array(arr, n);
  return 0;
}
```

Output



```
sivakumar@sivakumar:~/DS/Lab_4$ ./a.out
Enter the number of elements to be inserted
7
Enter integers one after the other
12
56
34
21
67
89
57
The sorted output using Insertion sort
12
21
34
56
57
67
89
```

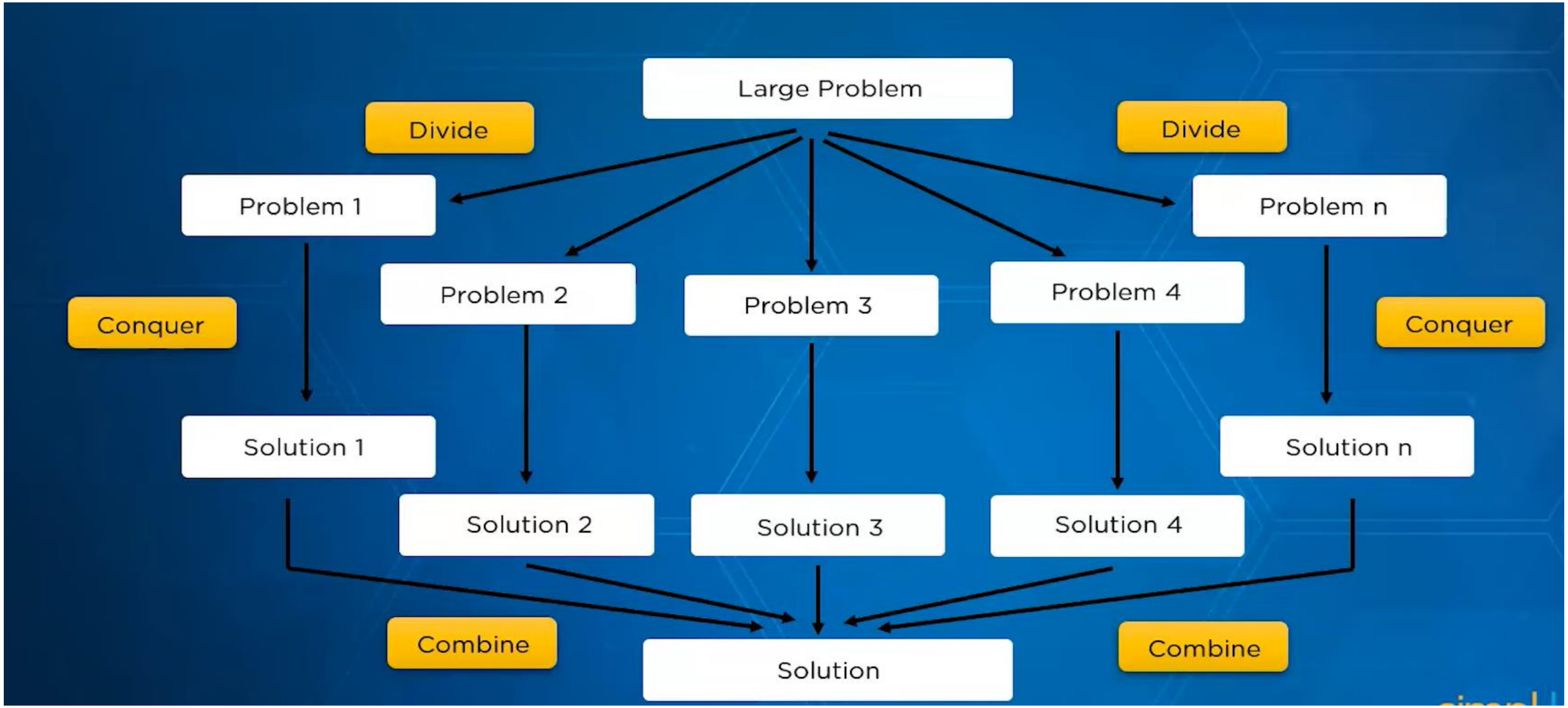
Divide and Conquer Approach

- Many algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems.
- These algorithms typically follow a **divide-and-conquer approach**: they break the problem into several subproblems that are similar to original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

Divide-and-and Conquer Subproblems:

1. **Divide**: Divide the problem into subproblems recursively until you cannot divide further.
2. **Conquer**: Conquer (solve) the subproblems by solving them recursively.
3. **Combine**: Combine the solutions to the subproblems to get the solution to the original problem

Nature of Divide-and-Conquer Problems



2. Merge Sort

Def: Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

How does it work?.

- Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided, i.e., the array has only one element left (an array with one element is always sorted).
- Then the sorted subarrays are merged into one sorted array.

Running time complexity of merge sort in the worst case is **O(n log n),** which means, given any n input numbers, the merge sort performs sorting in $\boldsymbol{n\,log\,n}$ steps.

Steps to perform Merge sort

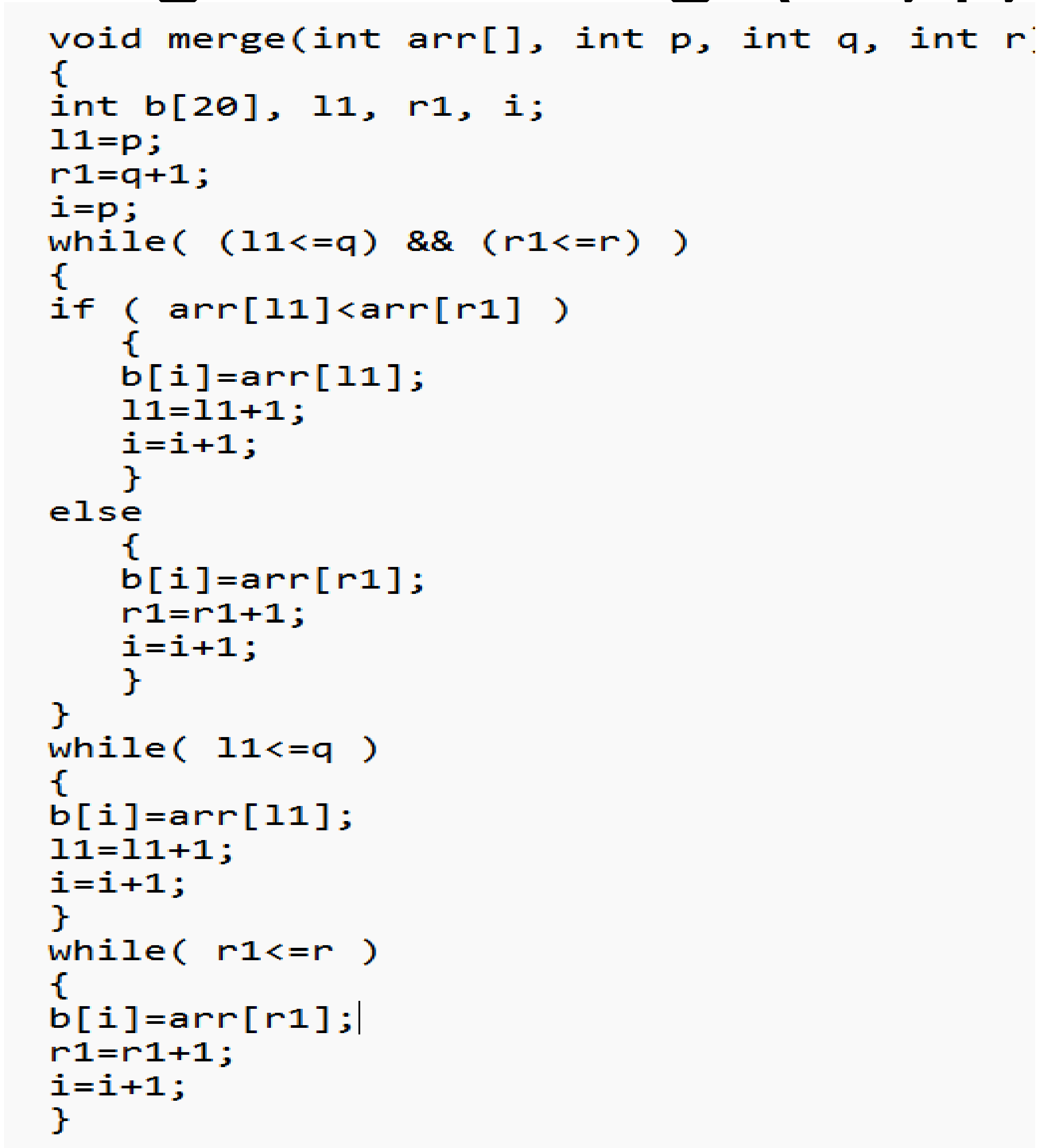
1. Divide the unsorted array into two sub-arrays, half the size of the original.
2. Continue to divide the sub-arrays as long as the current piece of the array has more than one element.
3. Merge two sub-arrays together by always putting the lowest value first.
4. Keep merging until there are no sub-arrays left.

Algorithm MergeSort (arr, l, h)

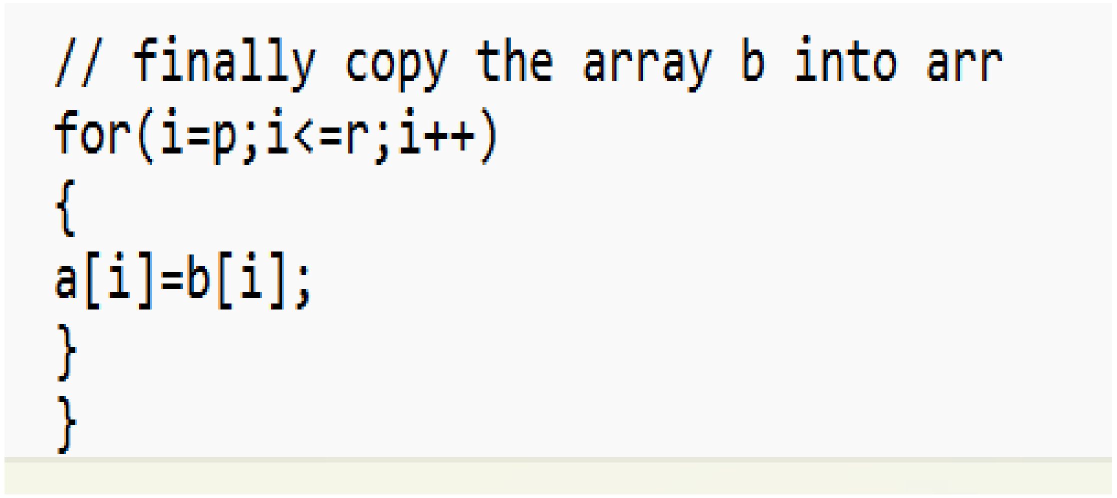
```
void Mergesort(int arr, int  l, int h)
//Here, arr is the name of the array, l is the lowest index value and h is the highest index value.
{
int mid;
if(l<h)
{
q=(l+h)/2;
// q is the middle element's position of the given array arr
Mergesort(arr, l,q);
Mergesort(arr, q+1,h);
merge(arr, l, q+1, h);
}
}
```

Note: merge algorithm is given in the next slide, which is also part of Mergesort algorithm

Algorithm merge(arr, p, q, r)

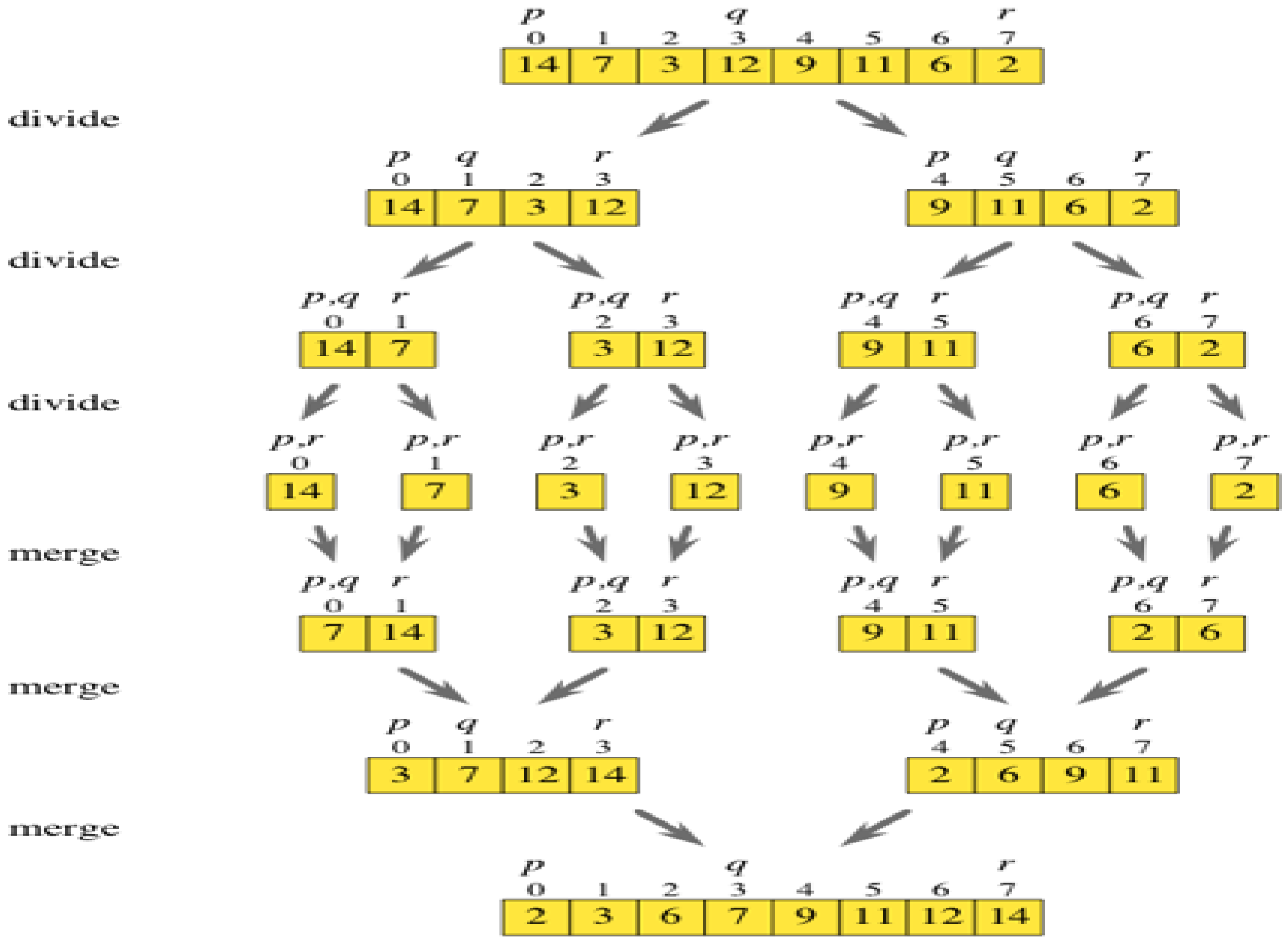


```
void merge(int arr[], int p, int q, int r
{
int b[20], l1, r1, i;
l1=p;
r1=q+1;
i=p;
while( (l1<=q) && (r1<=r) )
{
if ( arr[l1]<arr[r1] )
    {
    b[i]=arr[l1];
    l1=l1+1;
    i=i+1;
    }
else
    {
    b[i]=arr[r1];
    r1=r1+1;
    i=i+1;
    }
}
while( l1<=q )
{
b[i]=arr[l1];
l1=l1+1;
i=i+1;
}
while( r1<=r )
{
b[i]=arr[r1];
r1=r1+1;
i=i+1;
}
```

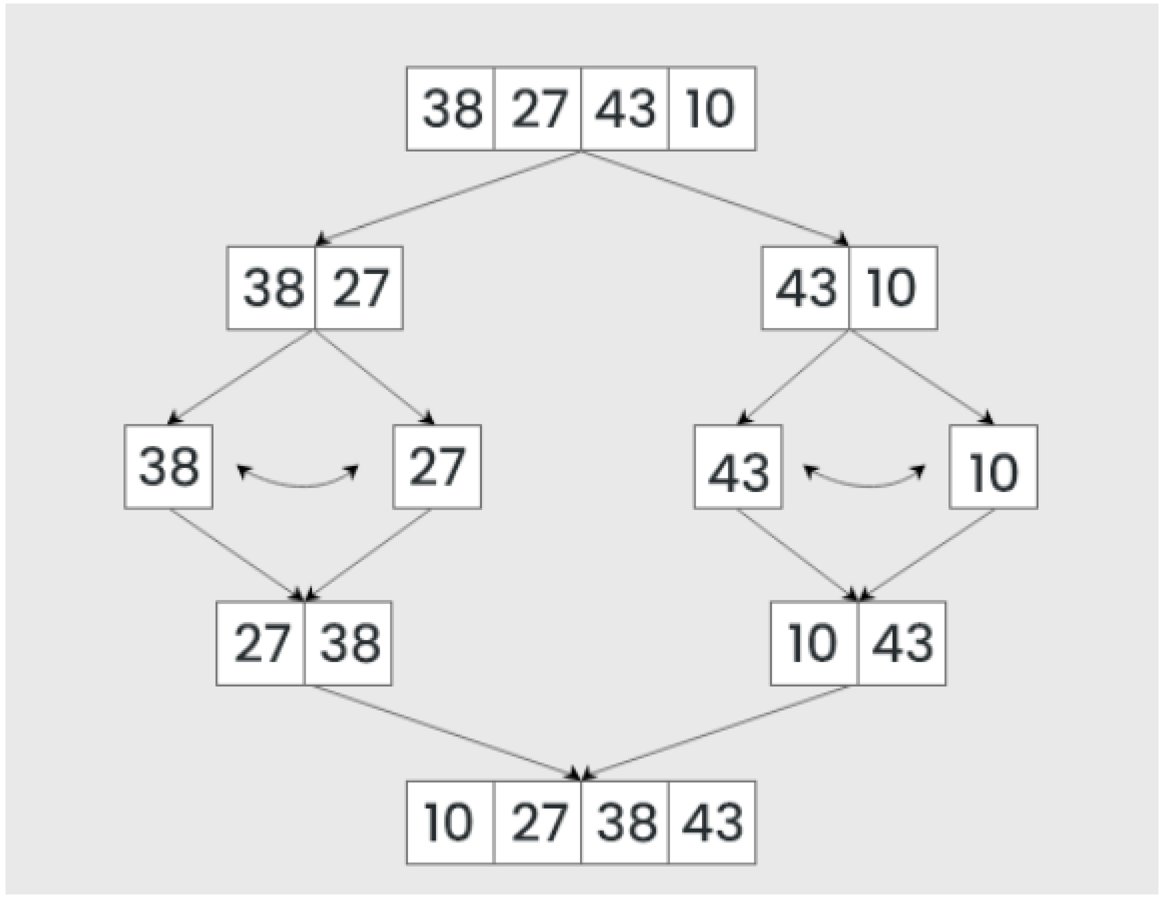


```
// finally copy the array b into arr
for(i=p;i<=r;i++)
{
a[i]=b[i];
}
}
```

Merge Sort Example



Another Merge Sort Example



Perform merge sort on the below array elements (Practice)

12	8	9	3	11	5	4

3. Quicksort

- Quick sort is another type of sorting technique that works on the principle of Divide and Conquere.
- It is a faster and one of the highly efficient sorting algorithms.
- This algorithm follows the divide and conquer approach to sort the given n numbers.

Quicksort steps:

1. First, pick an element as **pivot**, and then it partitions the given array around the picked pivot element
2. So, a large given array is divided into two arrays in which **one holds values that are smaller** than the **specified value (Pivot),** and **another array holds the values that are greater** than the **pivot**.
3. After step 2, left and right sub-arrays are also partitioned using the same approach (steps 1 and 2). It will continue until the single element remains in the sub-array.

Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort.

Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e., select the **random pivot** from the given array.
- Pivot can either be the **rightmost element** or the **leftmost element** of the given array.
- Select **median** as the pivot element.

Note: It is recommended to use first element of the given array as pivot element for convenient purpose.

Quicksort Algorithm

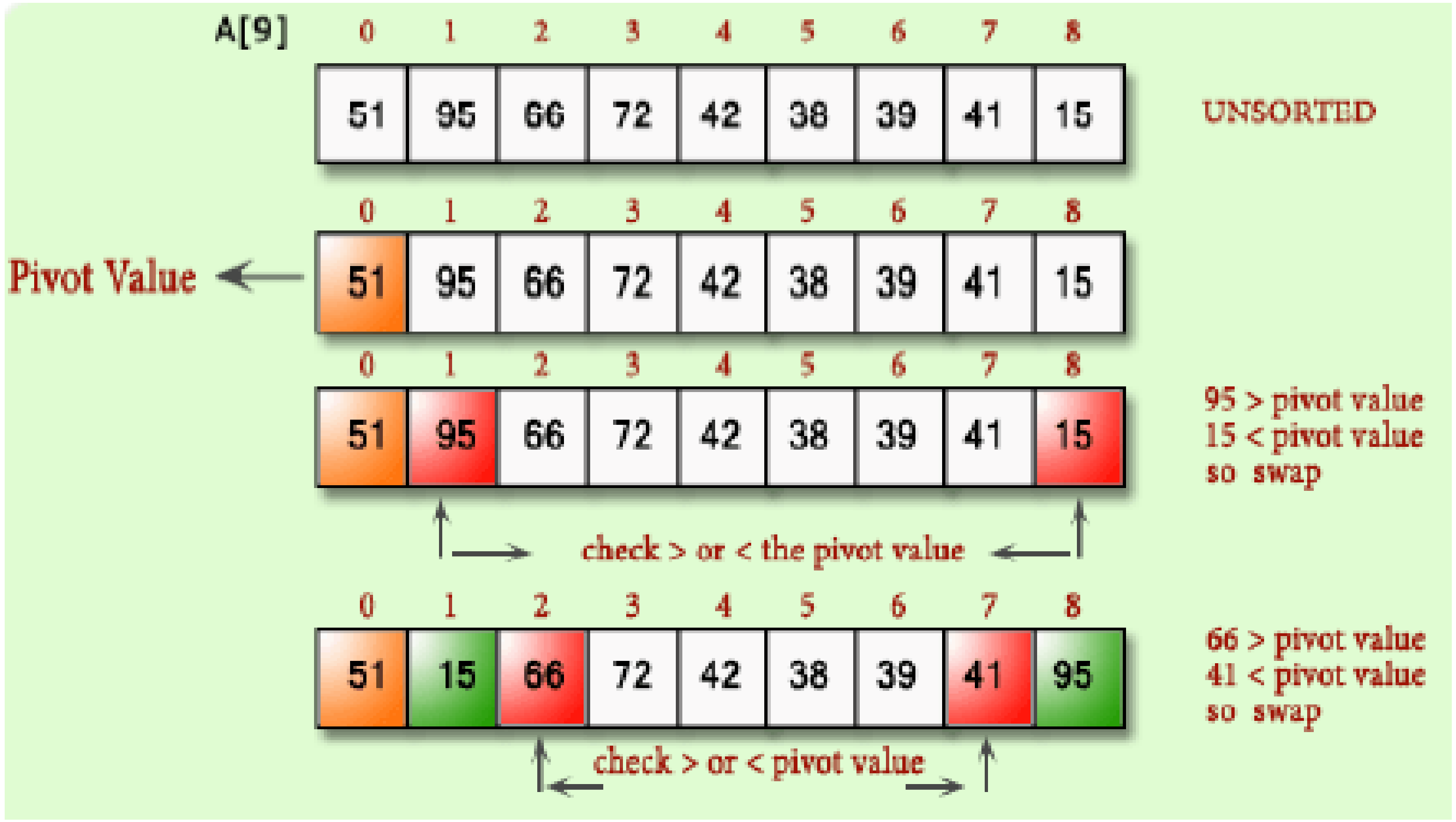
```
void Quicksort(int a[], int l, int h)
// a[] = array to be sorted, l = Starting index, h = Ending index
{
  if (l< h)
  {
    int p = partition(a, l, h); //p is the partitioning index
    Quicksort(a, l, p - 1);
    Quicksort(a, p + 1, h);
  }
}
```

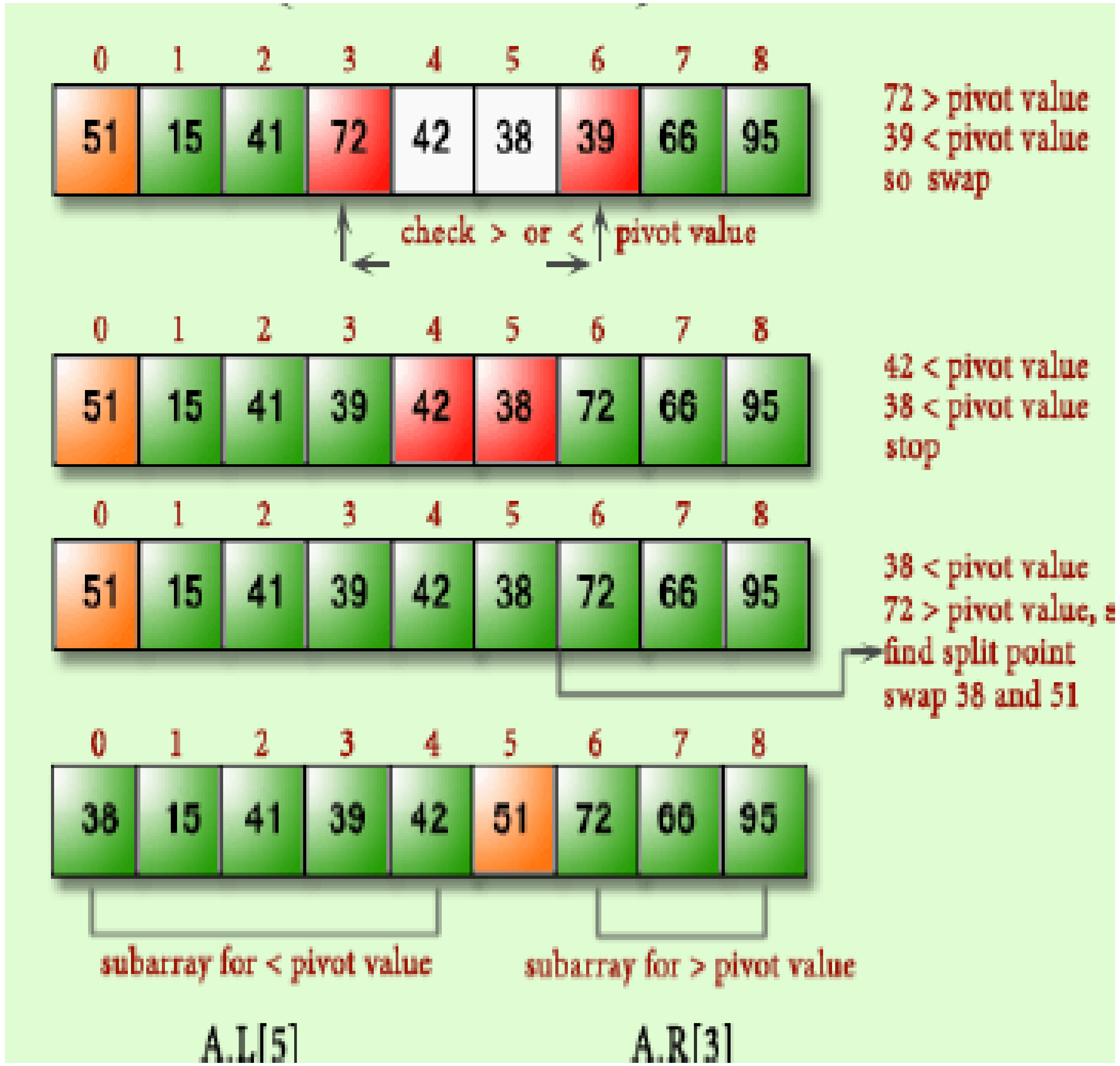
Note: Partition algorithm is given in the next slide, which is also part of Quicksort algorithm

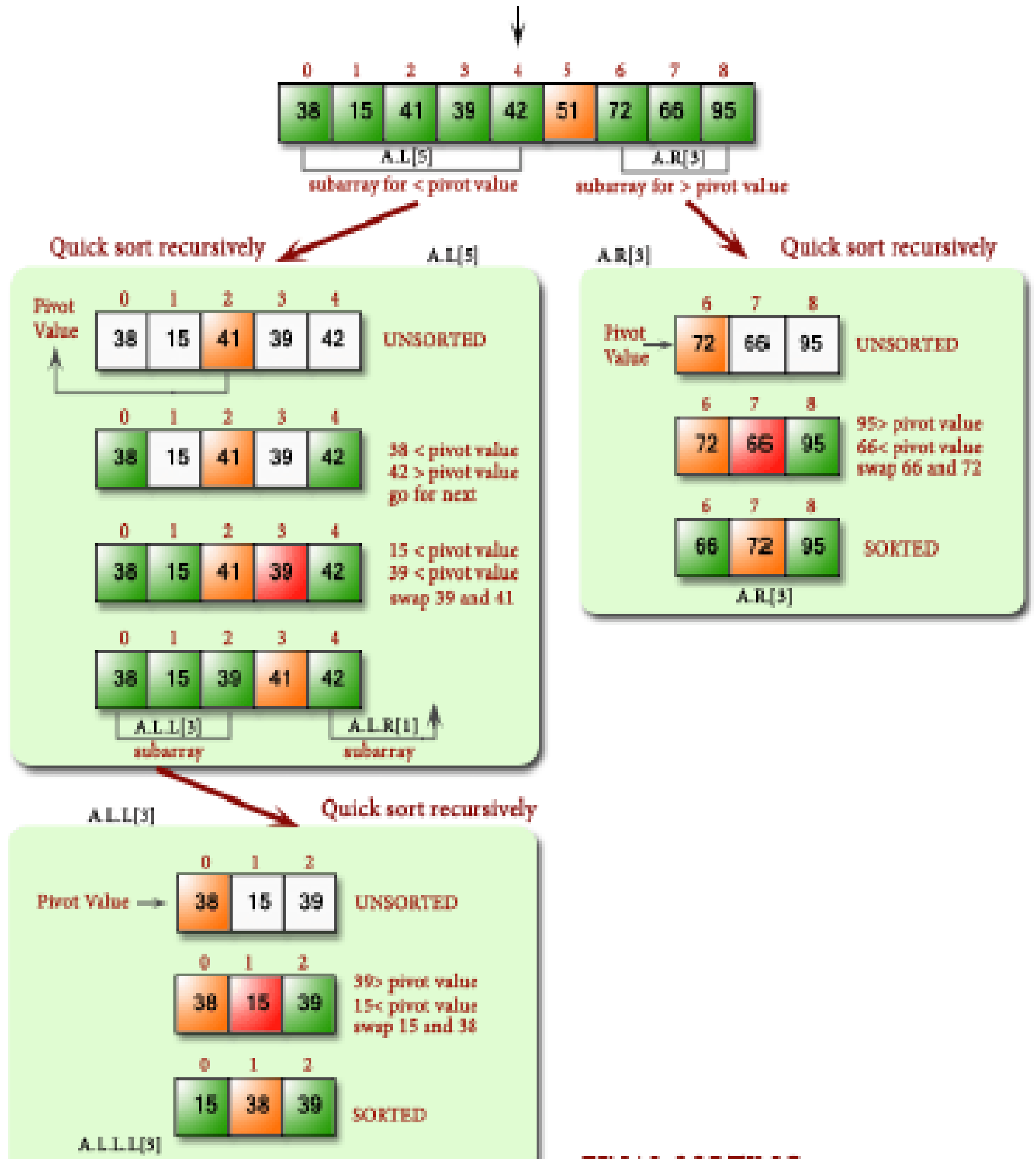
Partition Algorithm

```
Partition(A,l, h){
pivot= A[l]; // starting element as the pivot element
i=l;
j=h;
while(i<j)
{
while(A[i]<=pivot)
{
i=i+1;
}
while(A[j]>pivot)
{
j=j-1;
}
if (i<j))
swap A[i] with A[j]
}
Swap pivot with A[j]
return j;// returns the position of the sorted element (i.e, pivot value)
}
```

Quick Sort Example

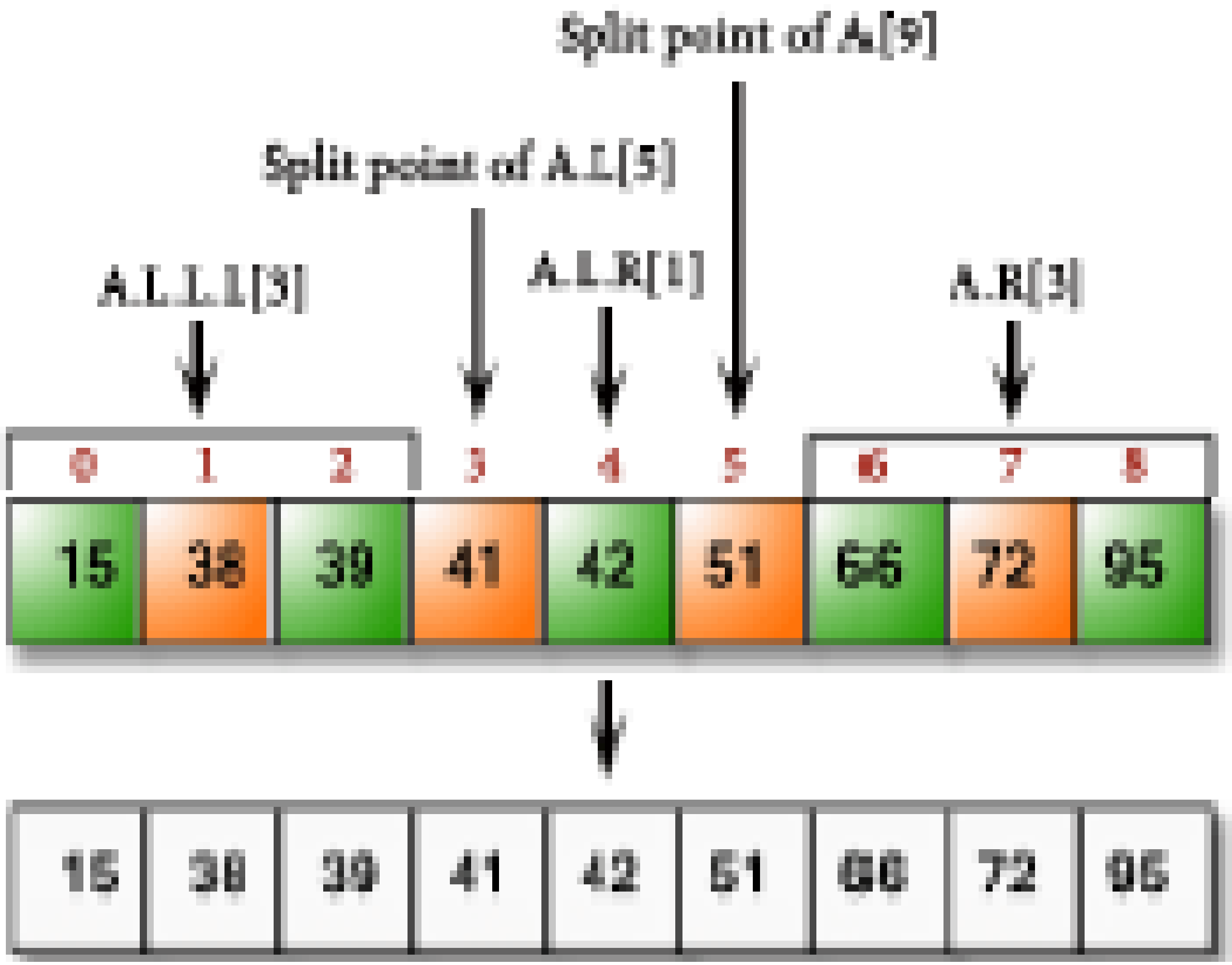



0 1 2 3 4 5 6 7 8
51 15 41 72 42 38 39 66 95
72 > pivot value
39 < pivot value
so swap
check > or < pivot value
0 1 2 3 4 5 6 7 8
51 15 41 39 42 38 72 66 95
42 < pivot value
38 < pivot value
stop
0 1 2 3 4 5 6 7 8
51 15 41 39 42 38 72 66 95
38 < pivot value
72 > pivot value, s
find split point
swap 38 and 51
0 1 2 3 4 5 6 7 8
38 15 41 39 42 51 72 66 95
subarray for < pivot value
subarray for > pivot value
A.L[5]
A.R[3]



38 15 41 39 42 51 72 66 95
A.L[5]
subarray for < pivot value
A.R[3]
subarray for > pivot value
Quick sort recursively
A.L[5]
A.R[3]
Quick sort recursively
Pivot Value
38 15 41 39 42
UNSORTED
38 < pivot value
42 > pivot value
go for next
15 < pivot value
39 < pivot value
swap 39 and 41
38 15 39 41 42
A.L.L[3]
subarray
A.L.R[1]
subarray
Pivot Value
72 66 95
UNSORTED
95> pivot value
66< pivot value
swap 66 and 72
66 72 95
SORTED
A.R[3]
Quick sort recursively
A.L.L[3]
Pivot Value
38 15 39
UNSORTED
39> pivot value
15< pivot value
swap 15 and 38
15 38 39
SORTED
A.L.L.L[3]

FINAL SORTING



Final sorted array

Go through another example discussed in Class

Analysis of Quicksort

- Runtime complexity of Quicksort in average case is **n log n**.

The average case means some input elements are partially sorted and some are not sorted.

- Run time complexity of Quicksort in the worst case (irrespective of the input) is **O(n^2)**.

Heap Sort

What is Heap Data Structure?

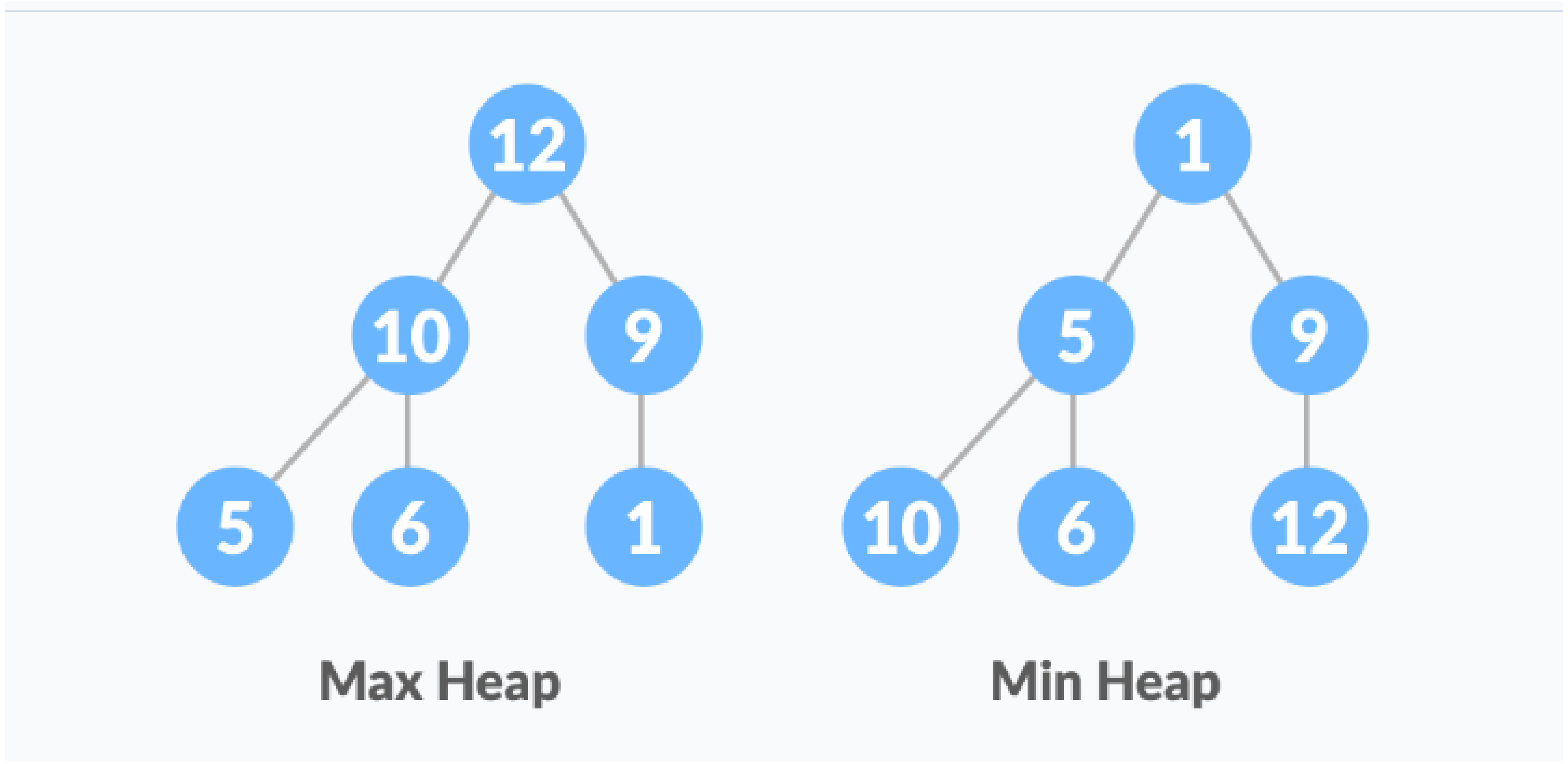
Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if

- It is a complete binary tree
- All nodes in the tree must be greater than or equal to their children. This type of heap is called a **max-heap**.

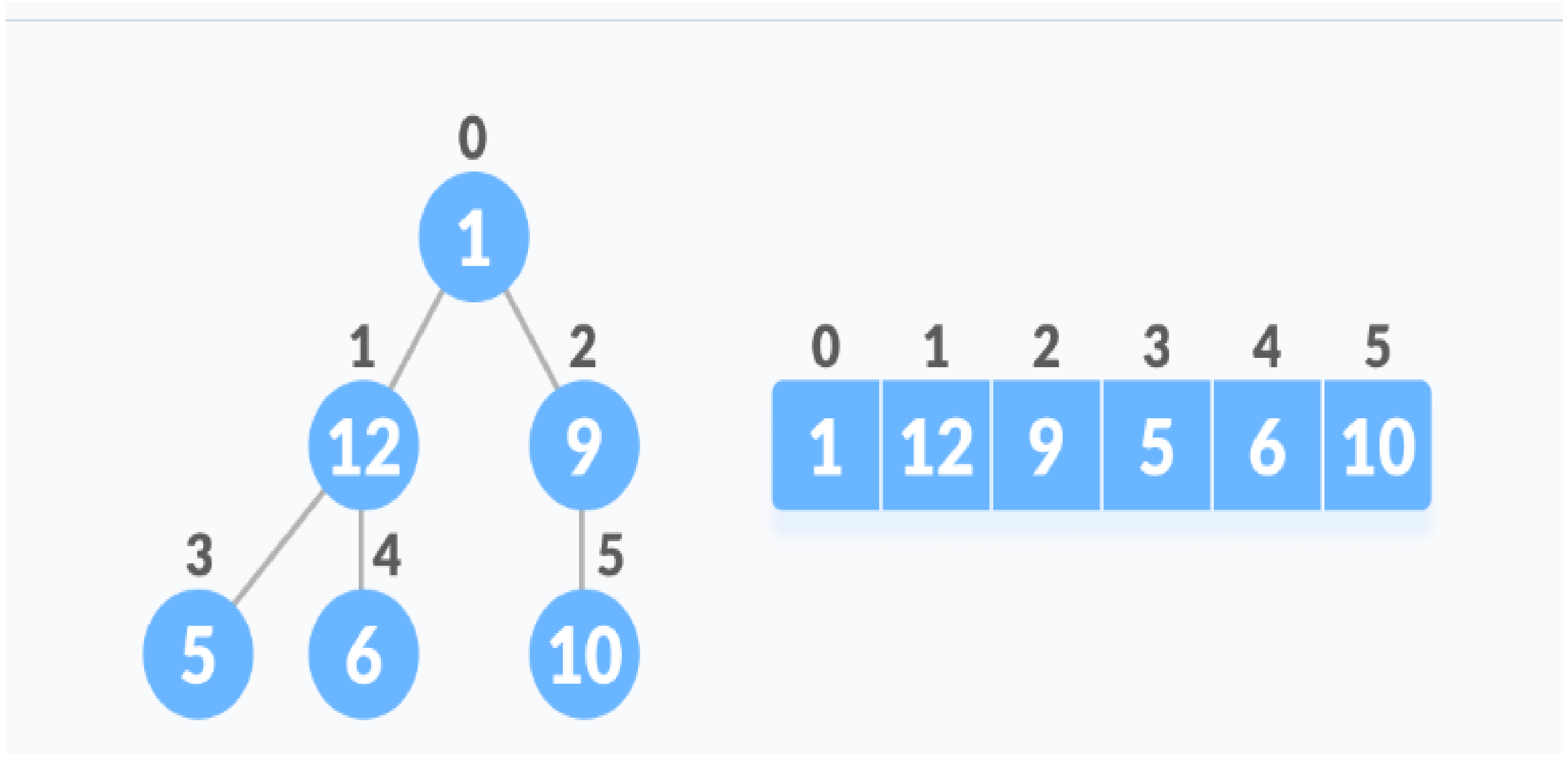
  Note: The largest item is stored at the root node.

  Note: Def **(min-heap)** All nodes must be less than or equal to children.

Examples of Max-heap and Min-heap



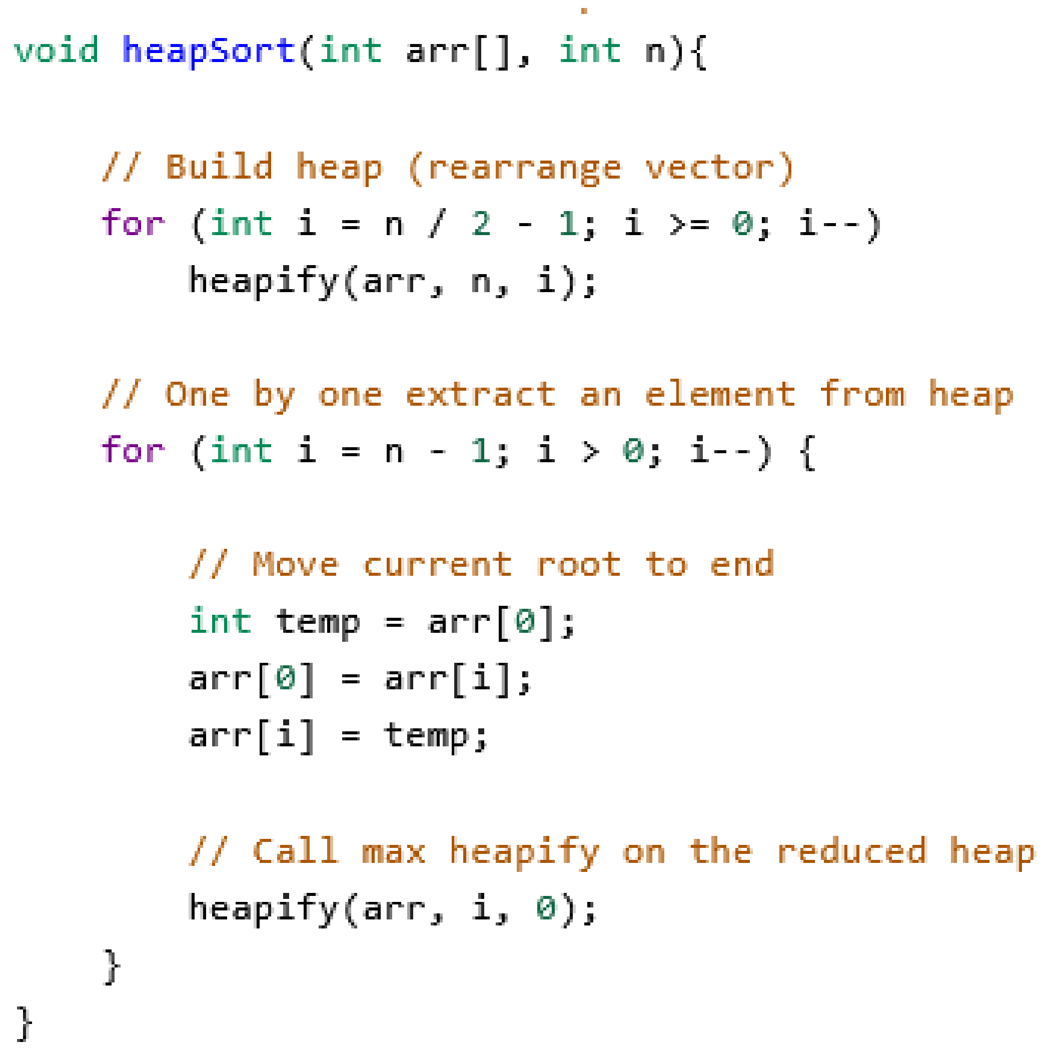
Relationship between array and heap indices



Working of Heap Sort (Max-heap)

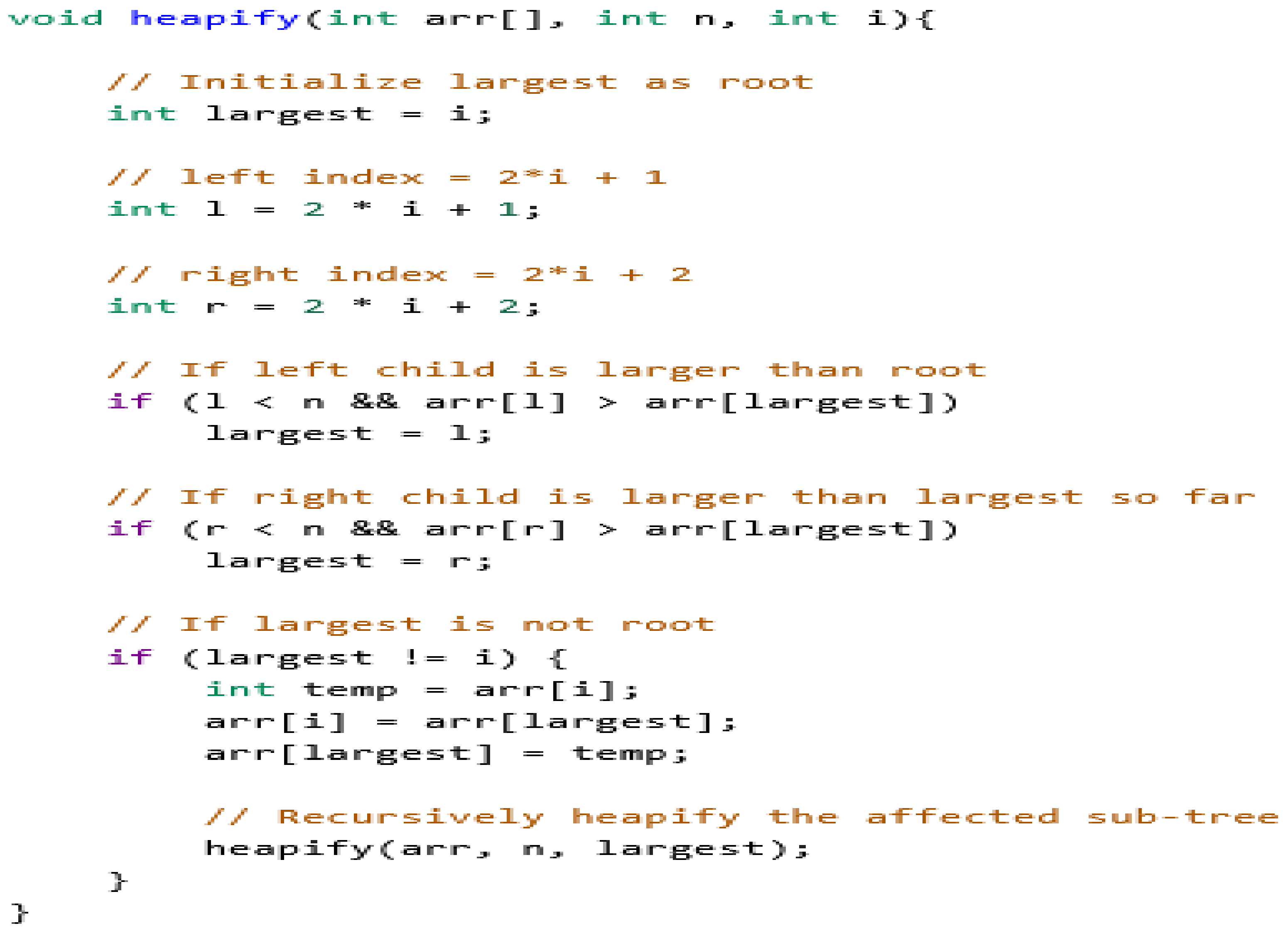
1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. **Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. **Remove:** Reduce the size of the heap by 1.
4. **Heapify:** Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.

Main function of Heap Sort



```
void heapSort(int arr[], int n){

    // Build heap (rearrange vector)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {

        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

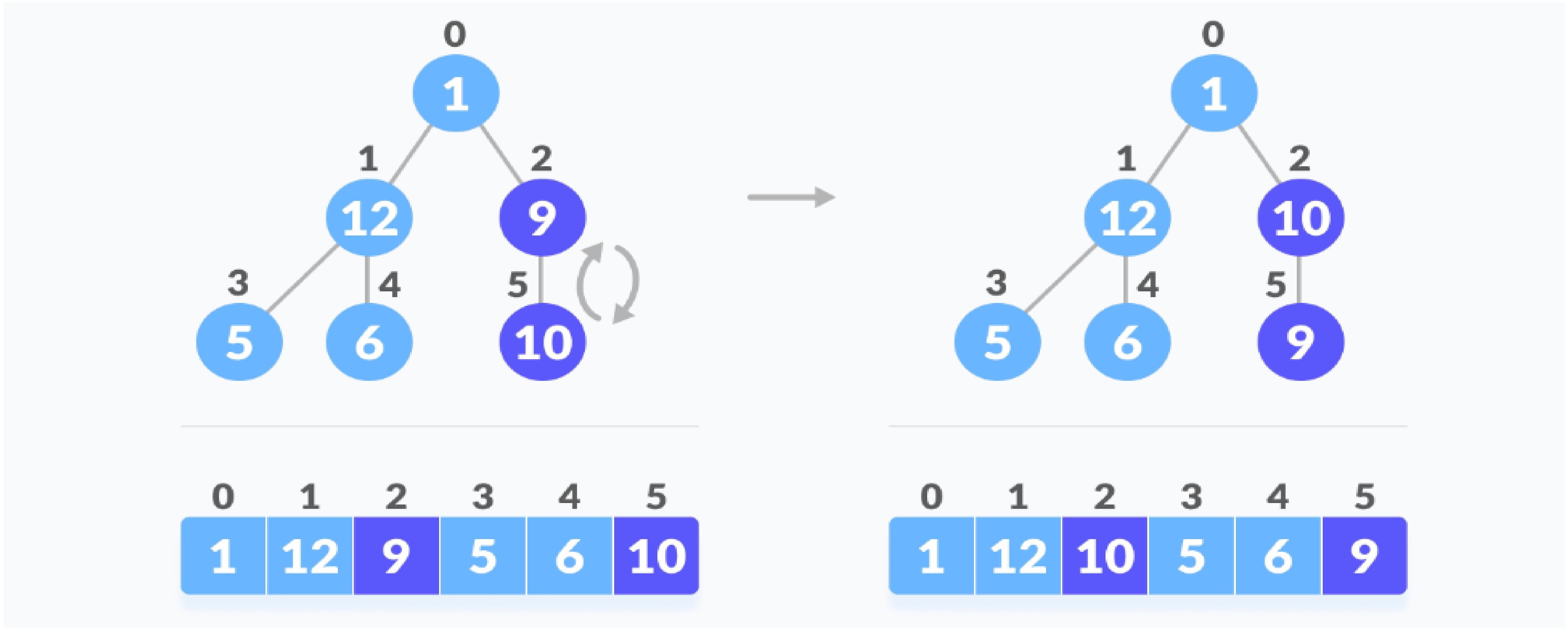
Pseudo Code Heapify Procedure



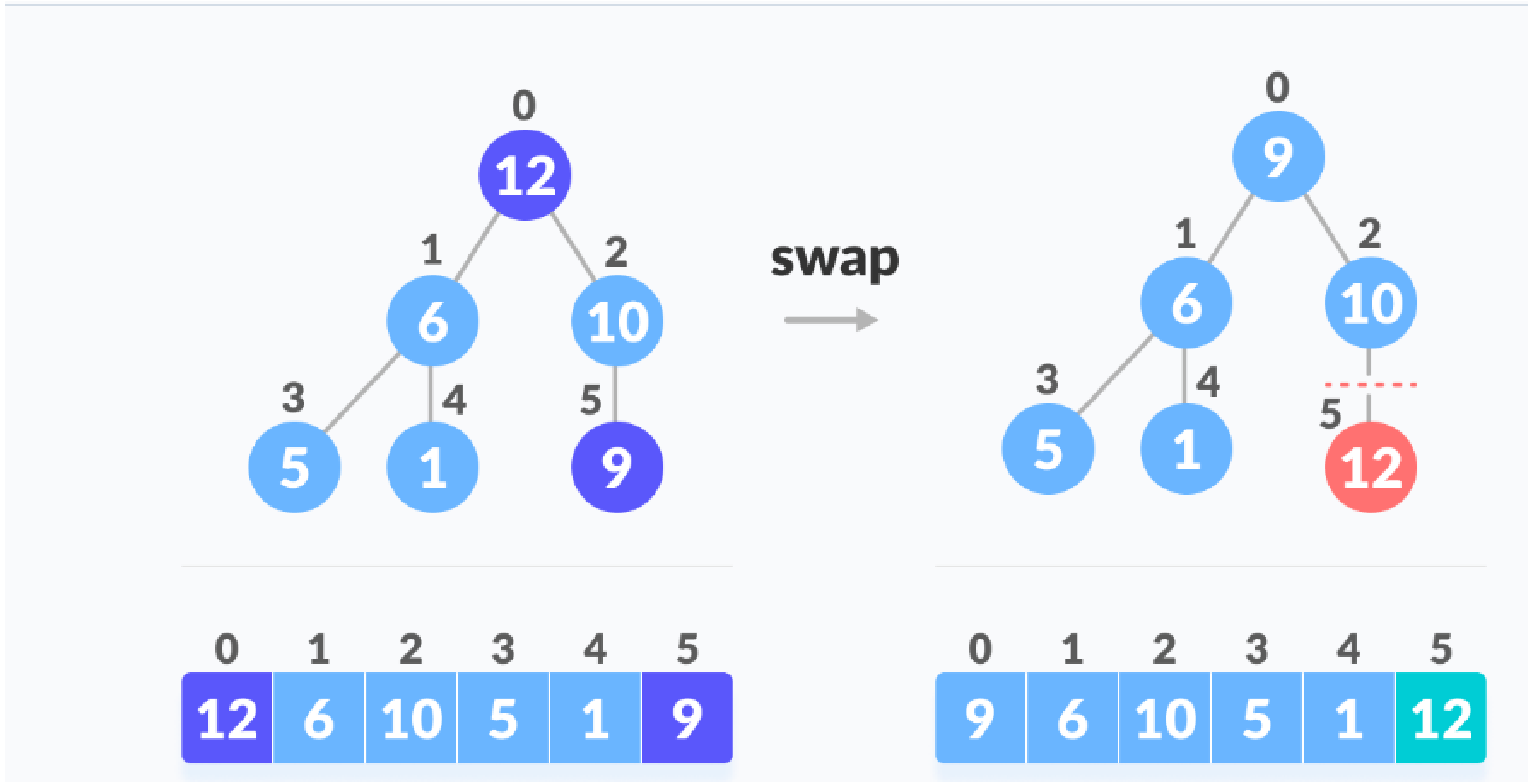
```
void heapify(int arr[], int n, int i){

    // Initialize largest as root
    int largest = i;

    // left index = 2*i + 1
    int l = 2 * i + 1;

    // right index = 2*i + 2
    int r = 2 * i + 2;

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

To heapify a subtree rooted with node i

Example of Heapifying

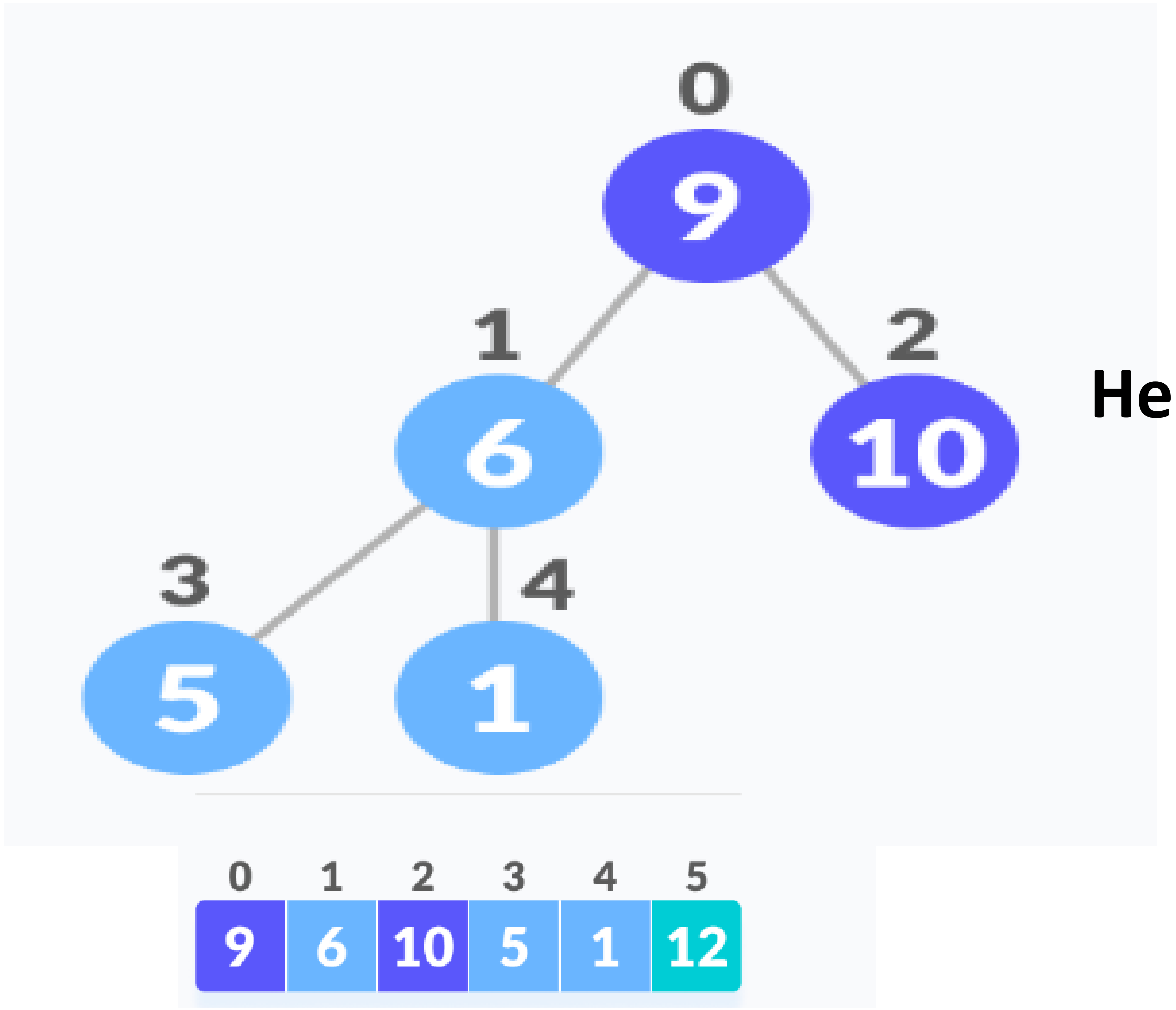


Sort the elements 12, 6, 10, 5, 1, 9 using heap sort

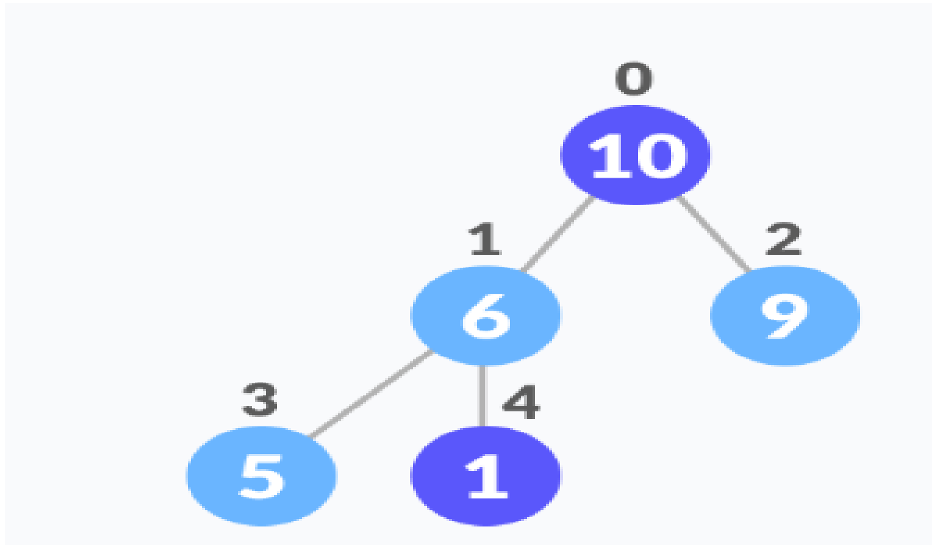

0
12
1
2
6
10
3
4
5
5
1
9
0 1 2 3 4 5
12 6 10 5 1 9
swap
0
9
1
2
6
10
3
4
5
5
1
12
0 1 2 3 4 5
9 6 10 5 1 12

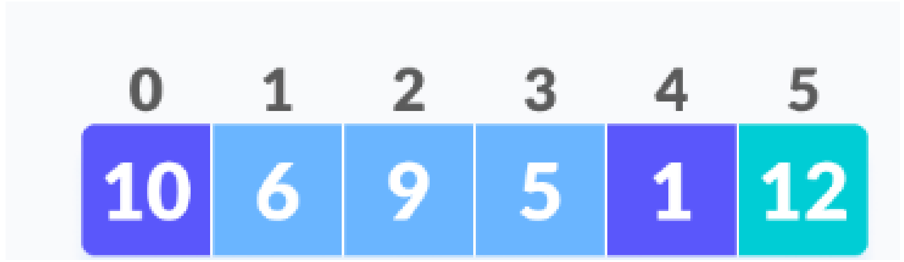
Remove 12

Resulting tree after removal of Node 12



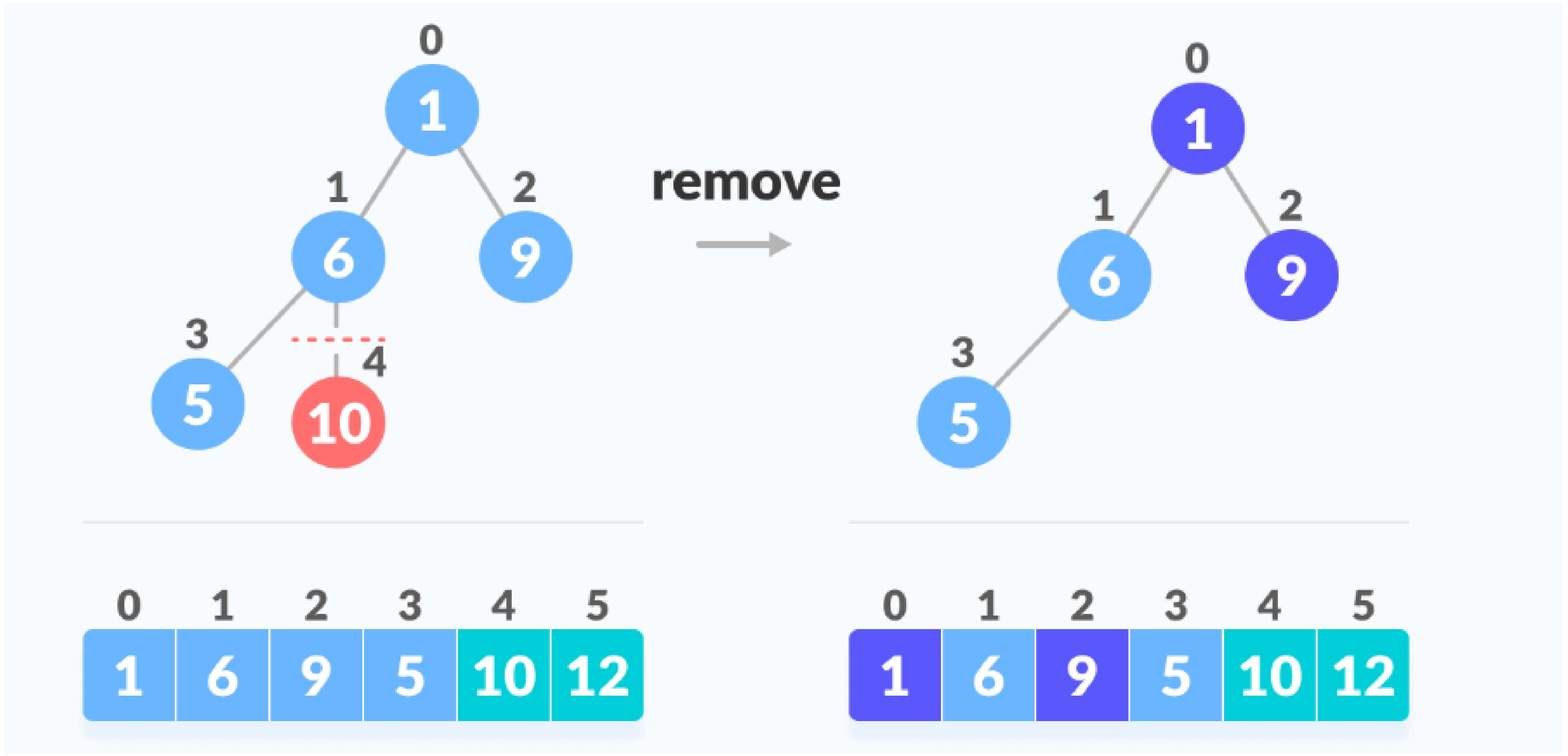
Heapify



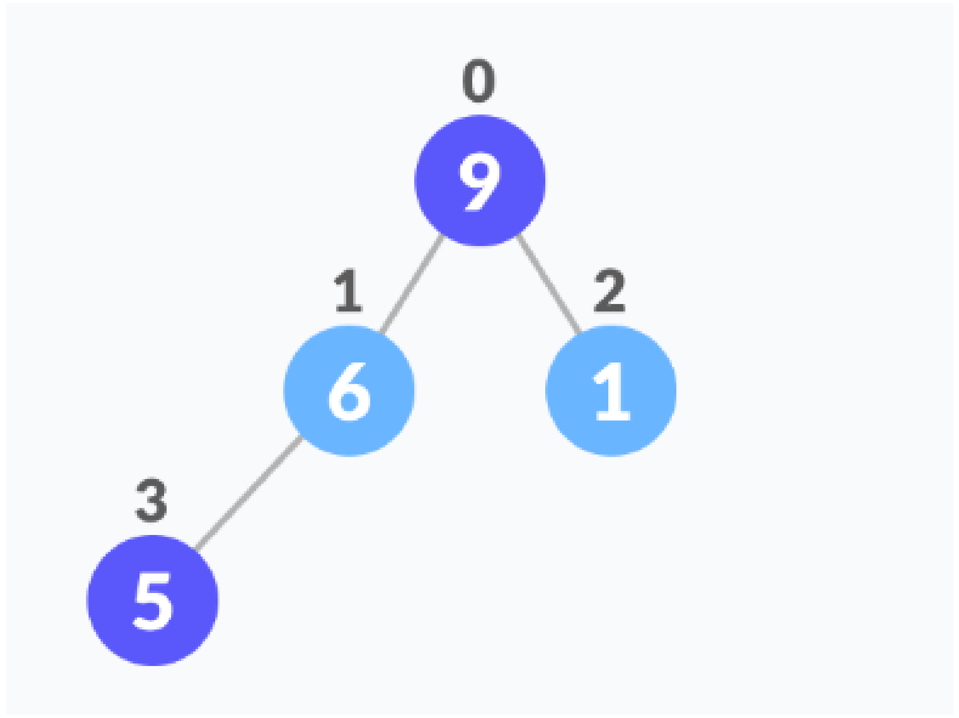


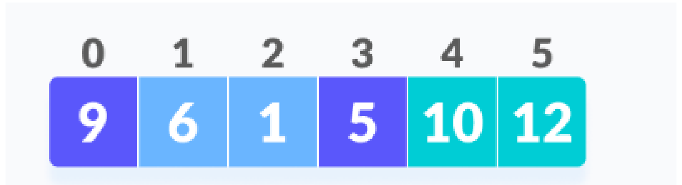
0	1	2	3	4	5
10	6	9	5	1	12

1. swap the root node 10 with the last node and then remove last node and then heapify

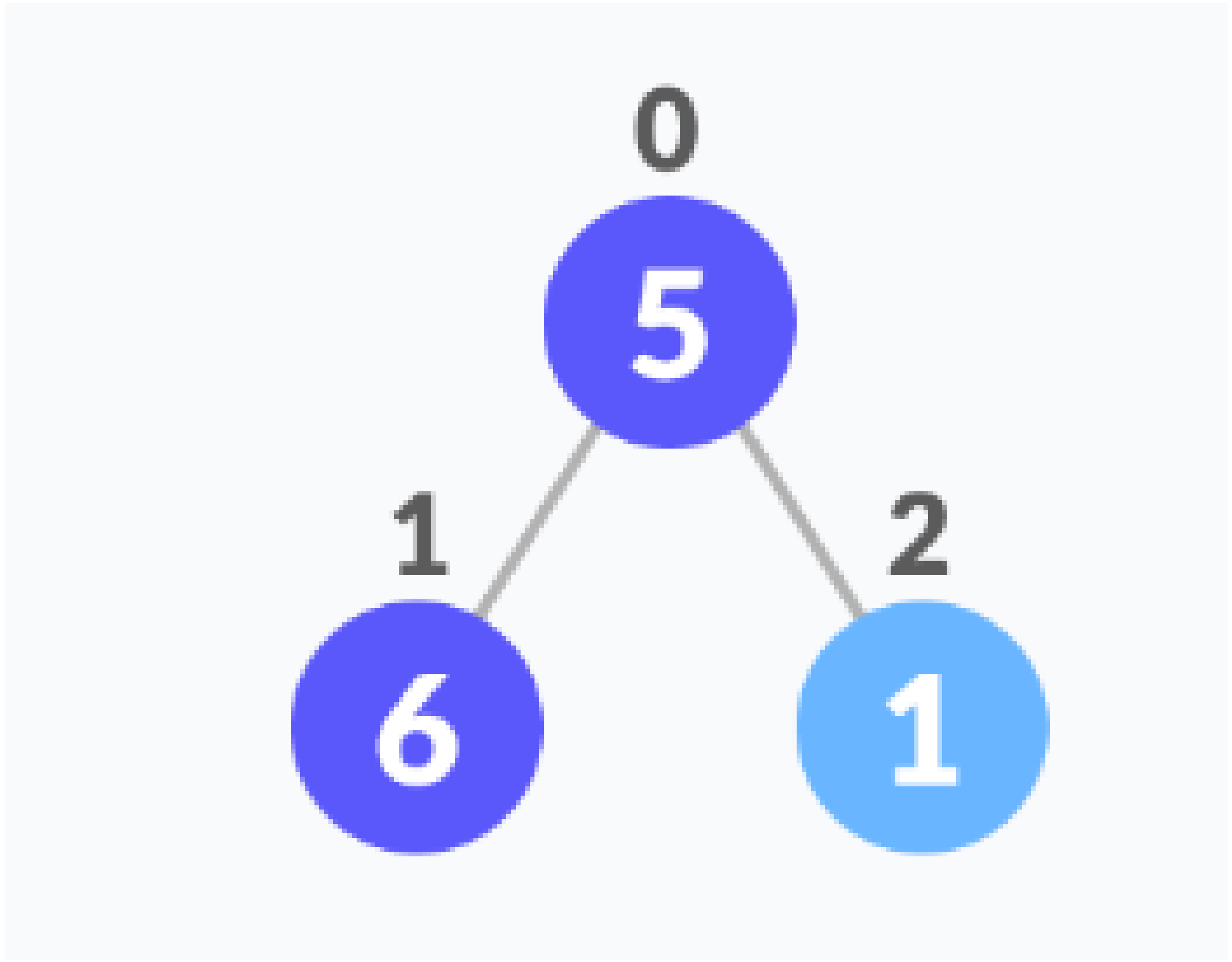


After heapifying

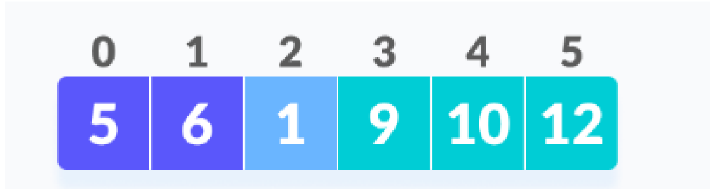




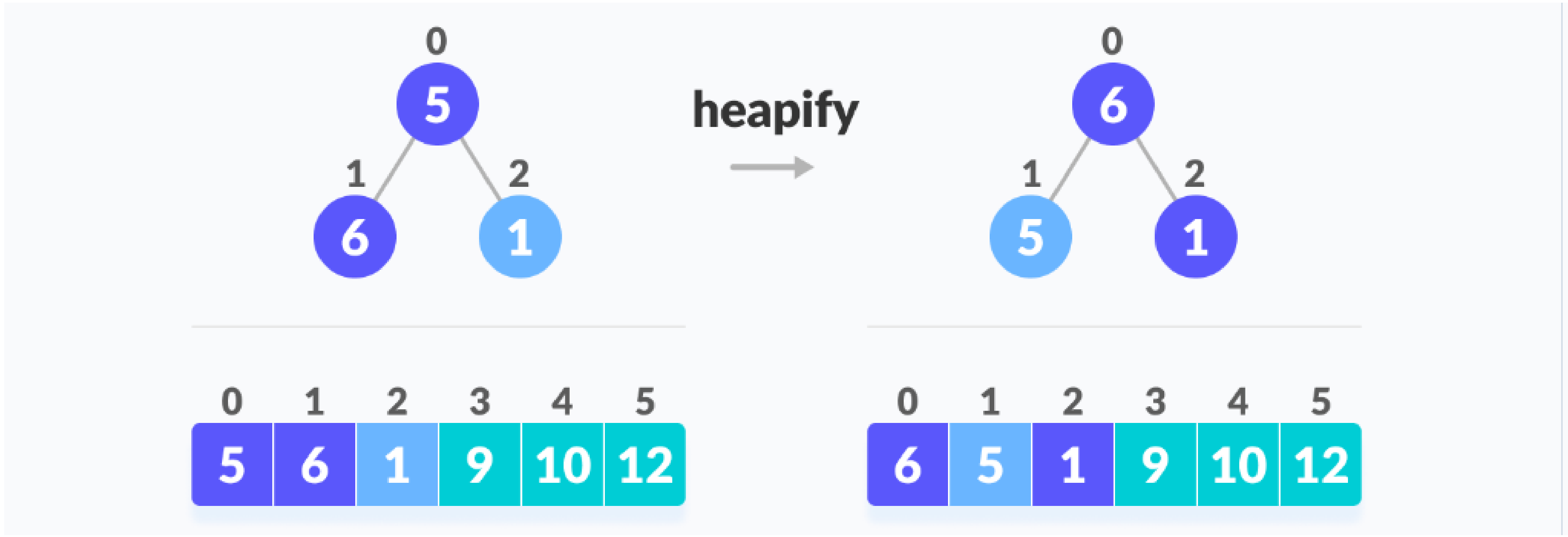
1. swap the root node with the last node
2. Remove last node
3. Heapify



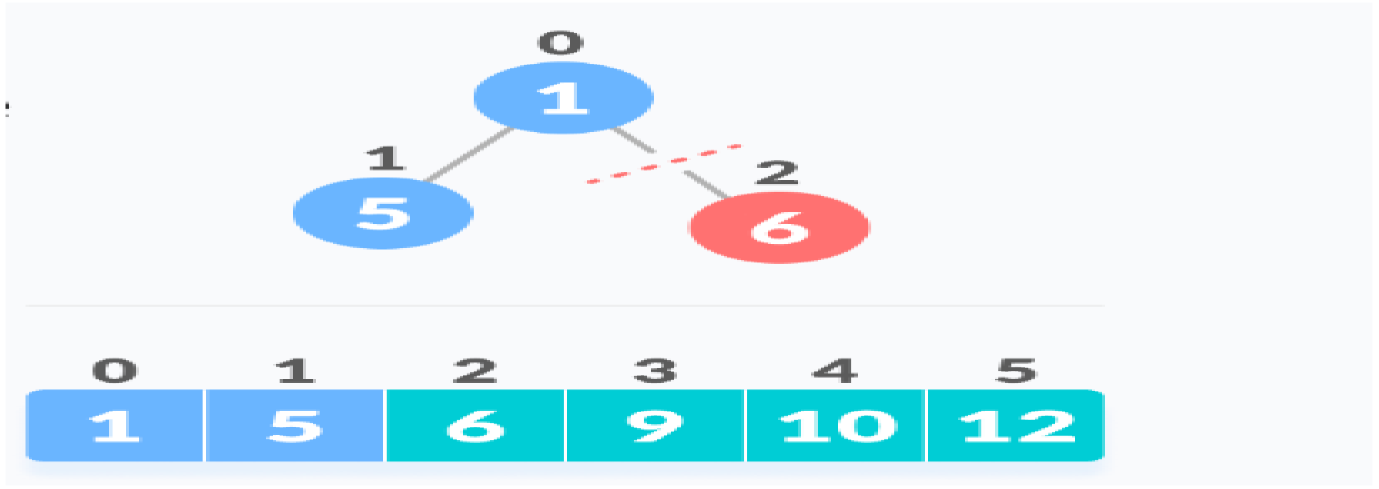
Heapify

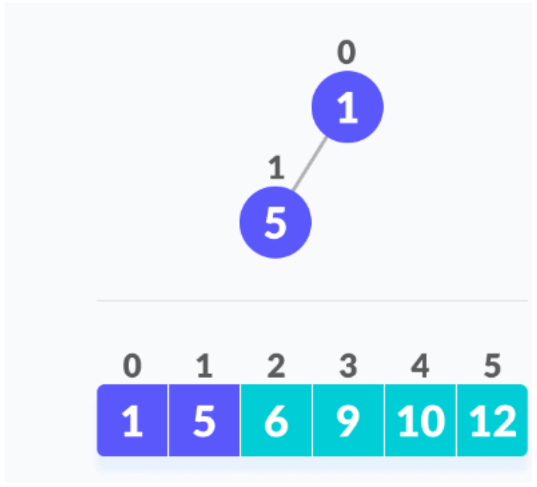


0	1	2	3	4	5
5	6	1	9	10	12


heapify
0
5
1
6
2
1
0 1 2 3 4 5
5 6 1 9 10 12
0
6
1
5
2
1
0 1 2 3 4 5
6 5 1 9 10 12

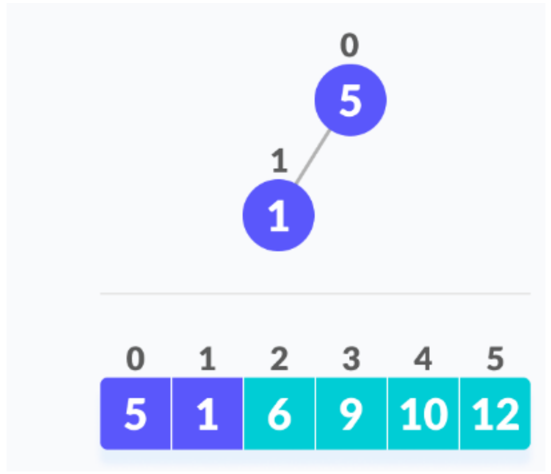
1. swap the root node with the last node
2. Remove last node
3. Heapify


0
1
1
5
2
6
0
1
2
3
4
5
1
5
6
9
10
12

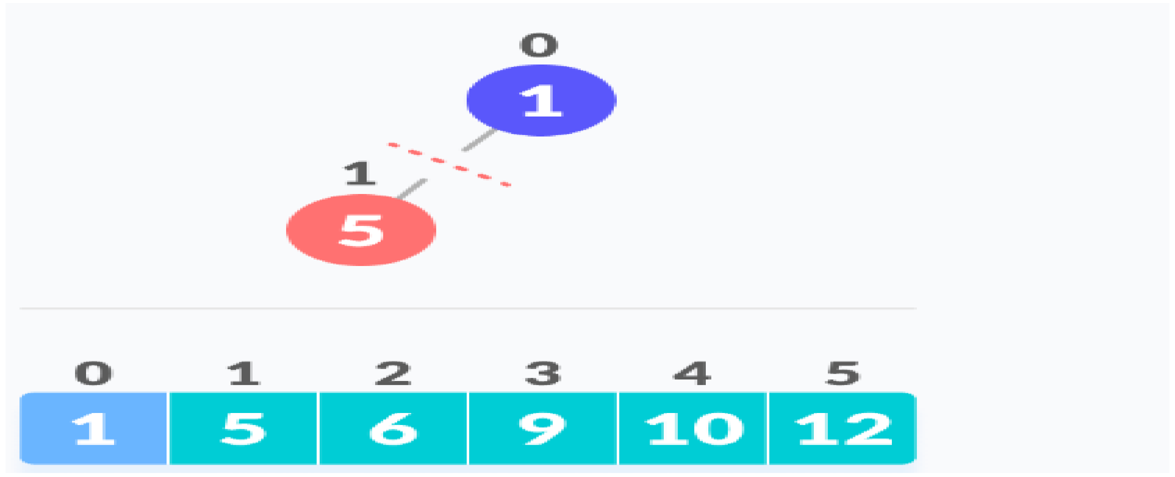


After heapifying

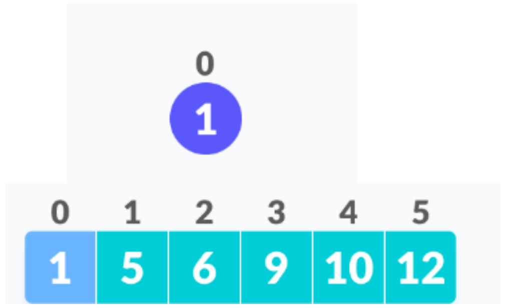
Heapify



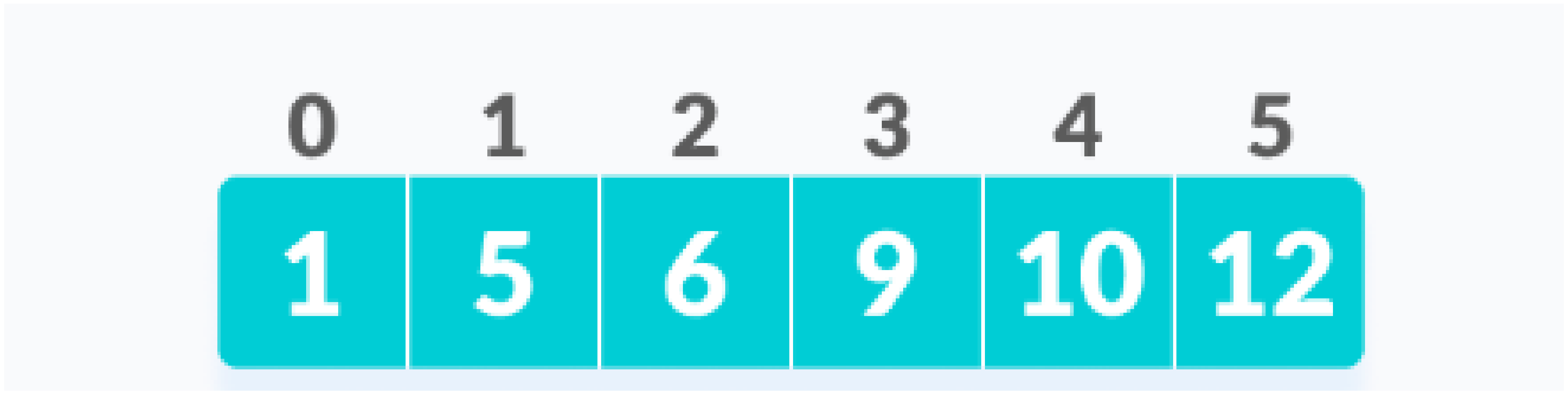
1. swap the root node with the last node
2. Remove last node
3. Heapify


0
1
1
5
0
1
2
3
4
5
1
5
6
9
10
12

Only one element, so remove it directly and place it in the array.



Final Sorted Array



Hashing

Hashing in Data Structures

Agenda

- General Idea
- Hash Function
- Open Hashing (Separate Chaining)
- Closed Hashing (Open Addressing) – Linear Probing

What is Hashing?

- Hashing is a key data structure that can assist in indexing and identifying a specific item when data mapping.
- Hashing maps data of arbitrary size to fixed-size values using a hash function for **fast retrieval**, storage or verification.
- Enables fast data access in constant average time O(1).
- Applications: Symbol tables, Database indexing, Caching, Password verification.

Why Hashing in Data Structures?

- To reduce the search time of data. It helps us find data much faster than the other data structures (we discussed).

Hashing Components

When using the hash technique, the following 3 main components are to be considered:

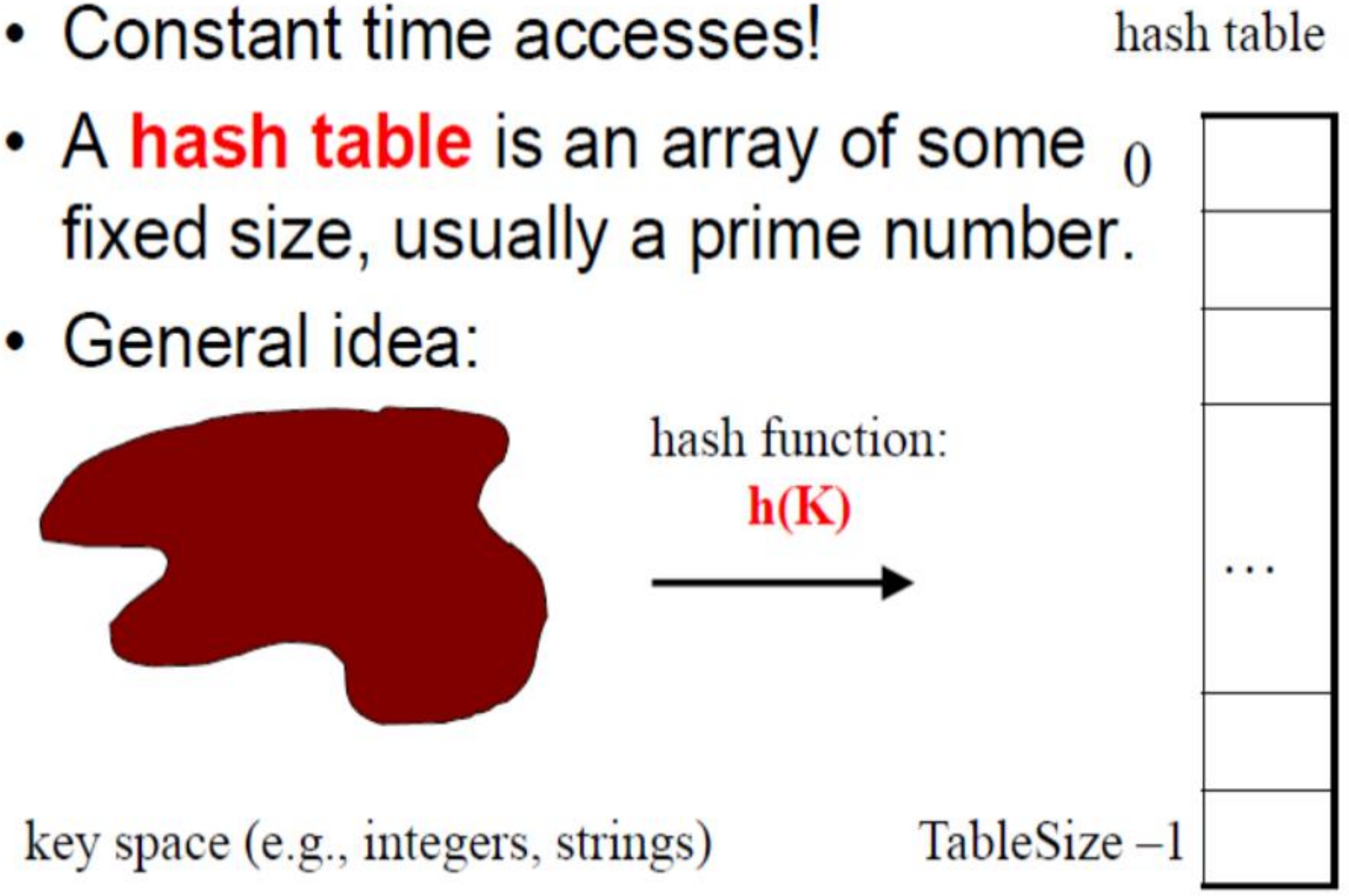
1. Hash Table,
2. Hash Function and
3. Handling Collisions

1. Hash Table

- Hash tables are the locations where a collection of data is stored so that it is easy to find the data when required.
- **Hash table** typically makes the searching process of an element significantly efficient.

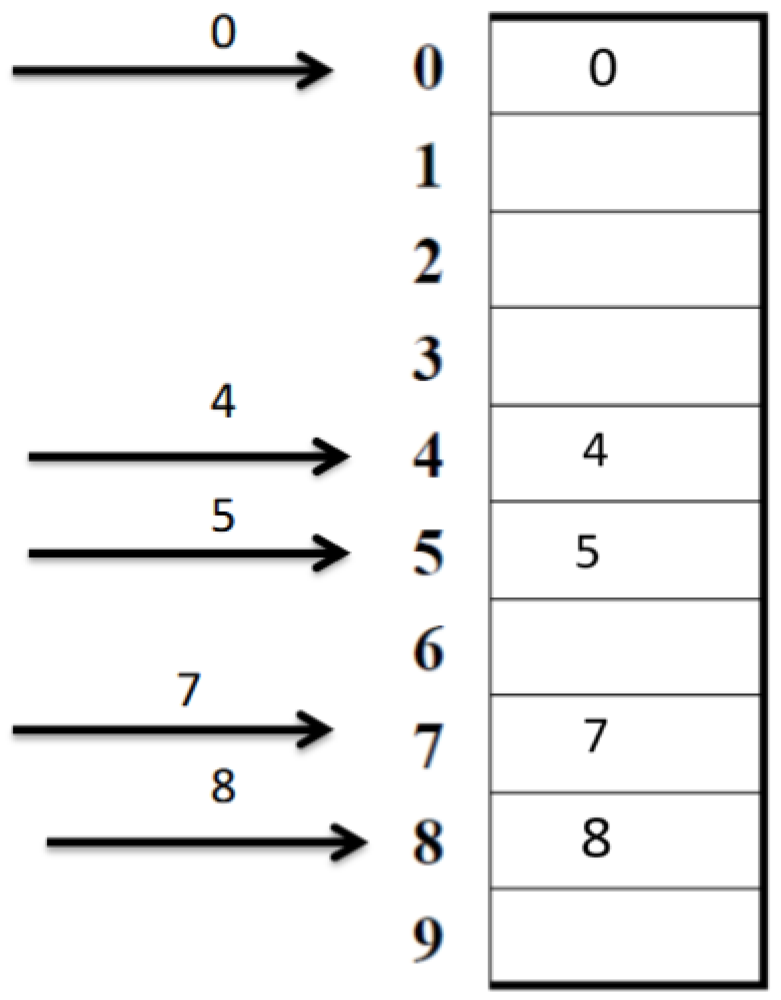
Hash Tables

- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:



Hash table

- key space = integers
- TableSize = 10
-
- insert: 7,0,5,4,8



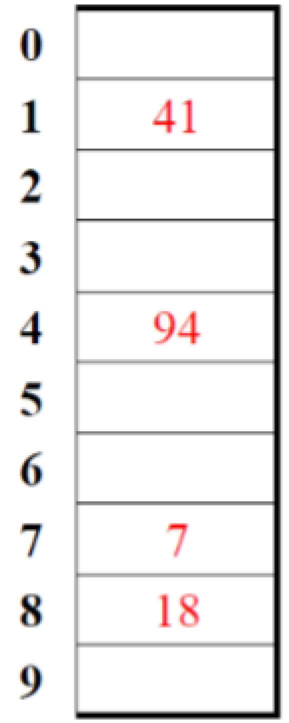
2. Hash Function

- A hash function maps keys to table indices.
- **Hash function acts as** mapping that maps an input key to an index in hash table.

- A simple hash function Example: **h(k) = k mod m**, where m is table size, k is the key value.
- **Desirable properties:** Simple, fast, uniform, minimal collisions.

Examples

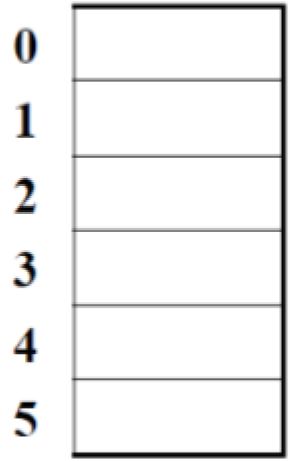
- **Ex1:**
- key space = integers
- TableSize = 10
- h(K) = K mod 10 ⇐ Hash Function
- **Insert**: 7, 18, 41, 94



- **Ex2:**

Task:

- key space = integers
- TableSize = 6
- h(K) = K mod 6 ⇐ Hash Function
- **Insert**: 7, 18, 41, 34



Some Good Properties of Hash Functions

1. Uniformity: The hash function should distribute the hash values uniformly across the output space to avoid clustering.

2. Deterministic: A hash function must consistently produce the same output for the same input.

3. Efficiency: The hash function should be able to process input quickly.

4. **Uniformity:** The hash function should distribute the hash values uniformly across the output space to avoid clustering.

3. Handling Collisions on Hash Table

Collision: It refers to mapping a given key value to an index where already some key value is stored.

- Collisions may occur when multiple keys map to same index.

An example of a hash function leading to a collision

- Hash Table size = 10, Keys = [23, 43, 13, 27]
- h(k) = k mod 10

Task: Map the above keys into Hash Table? Observe: Collision occurs.

How to handle Collisions when multiple keys map to same index?.

3: Collision Resolution Techniques

1. Open Hashing (**Separate Chaining)**
2. Closed Hashing (Open Addressing)
    i) Linear Probing (**Our syllabus is Up to Linear Probing only)**
    ii) Quadratic Probing
    iii) Double Hashing
    iv) Rehashing

3.1 Open Hashing (Separate Chaining)

- Each hash table index points to a linked list of records.
- **Separate chaining:** All keys that map to the same hash value are kept in a linked list (added one after the other)
- **Deletion:** Deleting a key requires searching the list and

  removing the element.
- **Advantages**: Easy to implement, table never fills up.
- **Disadvantages:** Extra memory, performance degrades with long chains.

Collision Resolution

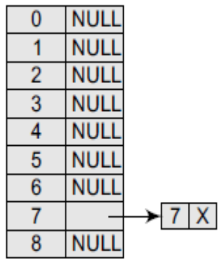
Insert the keys **7, 24, 18, 52, 36, 54, 11**, and **23** in a chained hash table of **9** memo

locations. Use h(k) = k mod m.

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL

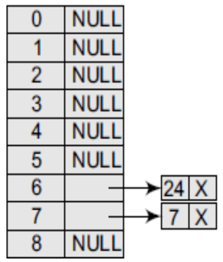
Step 1 Key = 7
h(k) = 7 mod 9
= 7

Create a linked list for location 7 and store the key value 7 in it as its only node.



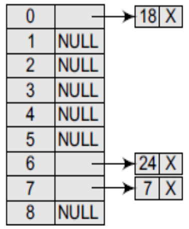
Step 2 Key = 24
h(k) = 24 mod 9
= 6

Create a linked list for location 6 and store the key value 24 in it as its only node.



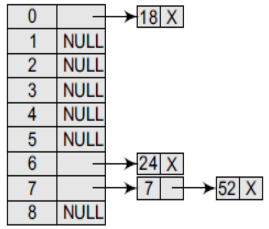
Step 3 Key = 18
h(k) = 18 mod 9 = 0

Create a linked list for location 0 and store the key value 18 in it as its only node.



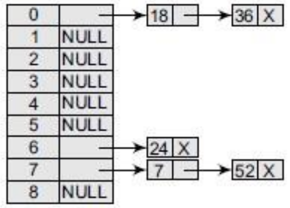
Step 4 Key = 52
h(k) = 52 mod 9 = 7

Insert 52 at the end of the linked list of location 7.



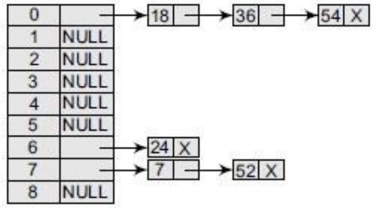
Step 5: Key = 36
h(k) = 36 mod 9 = 0

Insert 36 at the end of the linked list of location 0.



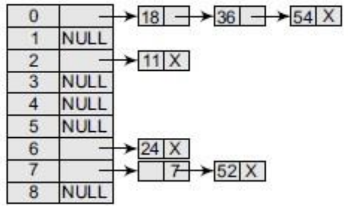
Step 6: Key = 54
h(k) = 54 mod 9 = 0

Insert 54 at the end of the linked list of location 0.



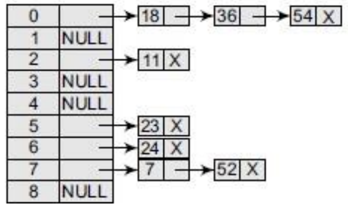
Step 7: Key = 11
h(k) = 11 mod 9 = 2

Create a linked list for location 2 and store the key value 11 in it as its only node.



Step 8: Key = 23
h(k) = 23 mod 9 = 5

Create a linked list for location 5 and store the key value 23 in it as its only node.



Another Example – Separate Chaining

- Keys: 10, 20, 30, 25
- Hash Function: h(k) = k mod 10
- Index 0: 10 → 20 → 30; Index 5: 25
- Keys with same hash value stored in linked list.

3.2 Closed Hashing (Open Addressing)

- All elements are stored within the hash table.
- When collision occurs, next available slot is searched.
- Probing sequence determines next slot for emptiness where the key value has to be inserted.
- **Types of Closed Hashing**: Linear Probing (our syllabus), Quadratic Probing, Double Hashing.

Handling Collisions using Opening address of Linear Probing technique

- If index i is occupied, check (i+1), (i+2), etc., until empty slot found.
- Hash function: **h_i(k) = (h(k) + i) mod m.**
    - **Here, k** is the key value, i is the probe number which can take values like 1, 2, 3, 4, …, so on until an empty slot is found in the hash table, m is the size of the hash table.
- **Advantages:** Simple implementation.
- **Disadvantages**: Causes primary clustering.

Example of Handling Collision using Open Addressing of Linear Probing

- Consider a hash table of size 10. Using **linear probing**, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.

Let $h'(k) = k \bmod m$, $m = 10$

Initially, the hash table can be given as:

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

Step 1 Key = 72

h(72, 0) = (72 mod 10 + 0) mod 10
= (2) mod 10
= 2

Since T[2] is vacant, insert key 72 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Step 2 Key = 27

h(27, 0) = (27 mod 10 + 0) mod 10
= (7) mod 10
= 7

Since T[7] is vacant, insert key 27 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Step 3 Key = 36

h(36, 0) = (36 mod 10 + 0) mod 10
= (6) mod 10
= 6

Since T[6] is vacant, insert key 36 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Step 4 Key = 24

h(24, 0) = (24 mod 10 + 0) mod 10
= (4) mod 10
= 4

Since T[4] is vacant, insert key 24 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Step 5 Key = 63

h(63, 0) = (63 mod 10 + 0) mod 10
= (3) mod 10
= 3

Since T[3] is vacant, insert key 63 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Step 6 Key = 81

h(81, 0) = (81 mod 10 + 0) mod 10
= (1) mod 10
= 1

Since T[1] is vacant, insert key 81 at this location.

0	1	2	3	4	5	6	7	8	9
0	81	72	63	24	–1	36	27	–1	–1

Step 7
```
          Key = 92
    h(92, 0) = (92 mod 10 + 0) mod 10
             = (2) mod 10
             = 2
```
Now T[2] is occupied, so we cannot store the key 92 in T[2]. Therefore, try again for the next location. Thus probe, i = 1, this time.
```
          Key = 92
    h(92, 1) = (92 mod 10 + 1) mod 10
             = (2 + 1) mod 10
             = 3
```
Now T[3] is occupied, so we cannot store the key 92 in T[3]. Therefore, try again for the next location. Thus probe, i = 2, this time.
```
          Key = 92
    h(92, 2) = (92 mod 10 + 2) mod 10
             = (2 + 2) mod 10
             = 4
```
Now T[4] is occupied, so we cannot store the key 92 in T[4]. Therefore, try again for the next location. Thus probe, i = 3, this time.
```
          Key = 92
    h(92, 3) = (92 mod 10 + 3) mod 10
             = (2 + 3) mod 10
             = 5
```
= 5

Since T[5] is vacant, insert key 92 at this location.

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	92	36	27	–1	–1

Step 8
```
          Key = 101
    h(101, 0) = (101 mod 10 + 0) mod 10
              = (1) mod 10
              = 1
```
Now T[1] is occupied, so we cannot store the key 101 in T[1]. Therefore, try again for the next location. Thus probe, i = 1, this time.
```
          Key = 101
    h(101, 1) = (101 mod 10 + 1) mod 10
              = (1 + 1) mod 10
              = 2
```
T[2] is also occupied, so we cannot store the key in this location. The procedure will be repeated until the hash function generates the address of location 8 which is vacant and can be used to store the value in it.

Another Example – Linear Probing

- Hash Table Size = 10, h(k) = k mod 10, Keys = [23, 43, 13, 27]
- 23→3, 43→3(Collision→4), 13→3(Collision→5), 27→7
- Items are stored by probing next available slots.

Comparison – Separate Chaining vs Linear Probing

- Separate Chaining: Uses linked lists, needs extra memory.
- Linear Probing: Uses only table slots, may cause clustering.
- Chaining easier to delete; probing depends on load factor.

Summary

- Hashing provides O(1) average access time.
- Good hash function minimizes collisions.
- Separate Chaining uses linked lists.
- Linear Probing searches for next open slot.
- Technique choice depends on memory and performance needs.

Extra Slide

Some common hashing algorithms that are are in use today are:

- **MD5** (Message Digest Algorithm 5)
- **SHA-1** (Secure Hash Algorithm 1)
- **SHA-256** (Secure Hash Algorithm 256)
- **SHA-512** (Secure Hash Algorithm 512)

References

- Alfred V. Aho et al., Data Structures and Algorithms
- E. Horowitz, S. Sahni, Fundamentals of Data Structures
- Cormen, Leiserson, Rivest, Stein, Introduction to Algorithms
- Geeksforgeeks, Tutorialpoint, etc.
- https://www.cse.iitd.ac.in/~mohanty/col106/Resources/Hashing.pptx&ved=2ahUKEwiA4Jio2cGQAxWCV3ADHeUSFPsQFnoECBYQAQ&usg=AOvVaw3m_6hloRz2Ole1_WQdlo30