Parul® University

Data Structure

Dr. Ghanshyam Rathod, Assistant Professor
IT and Computer Science

Parul®
University

DIGITAL LEARNING CONTENT

CHAPTER – 6

Searching, Sorting and Hashing

Linear Search (For Unsorted Array)

- Searching is the process of finding some particular element in the list.
- Two popular search methods are Linear Search and Binary Search.
- Linear search is also called as sequential search algorithm.
- It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found.
- If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.
- It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is O(n).

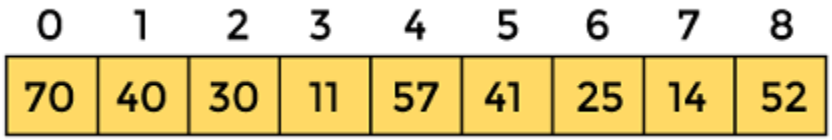
Linear Search (For Unsorted Array) steps

The steps used in the implementation of Linear Search are:

- First, we have to traverse the array elements using a for loop.
- In each iteration of for loop, compare the search element with the current array element, and -
- If the element matches, then return the index of the corresponding array element.
- If the element does not match, then move to the next element.
- If there is no match or the search element is not present in the given array, return -1.

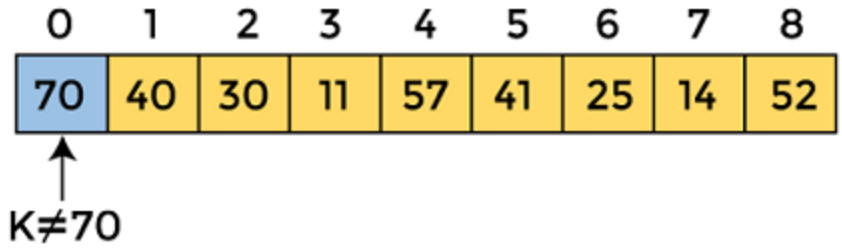
Working of Linear Search (For Unsorted Array)

Let's take an unsorted array



Let the element to be searched is K = 41

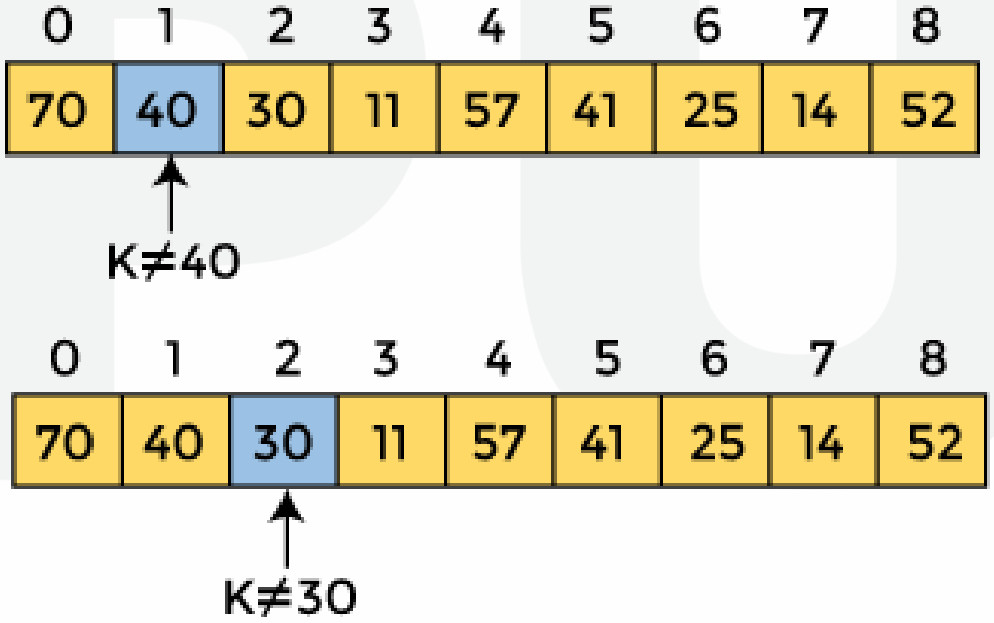
Now, start from the first element and compare K with each element of the array.



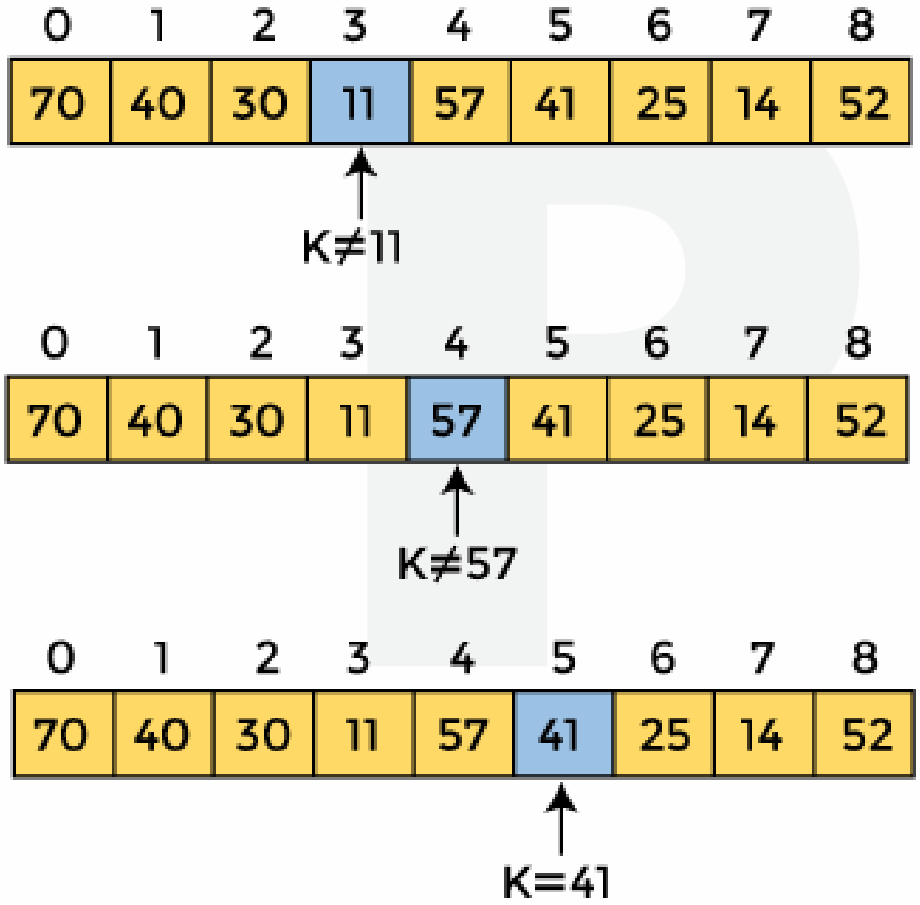
The value of K, i.e., 41, is not matched with the first element of the array. So, move to the next element.

Working of Linear Search (For Unsorted Array)

Follow the same process until the respective element is found.



Working of Linear Search (For Unsorted Array)



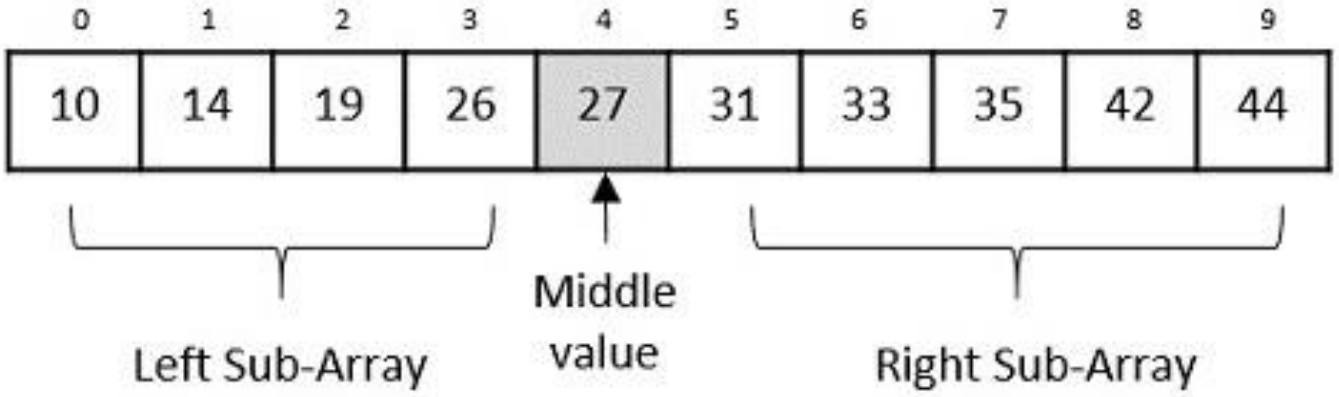
Now, the element to be searched is found. So, algorithm will return the index of the element matched.

Binary Search (For Sorted Array)

- Binary search is the search technique that works efficiently on sorted lists.
- Binary search is a fast search algorithm with run-time complexity of O(log n).
- This search algorithm works on the principle of divide and conquer, since it divides the array into half before searching.
- For this algorithm to work properly, the data collection should be in the sorted form.
- Binary search looks for a particular key value by comparing the middle most item of the collection.
- If a match occurs, then the index of item is returned.

Binary Search (For Sorted Array)

- But if the middle item has a value greater than the key value, the right sub-array of the middle item is searched.
- Otherwise, the left sub-array is searched. This process continues recursively until the size of a subarray reduces to zero.



Binary Search Algorithm (Only for Sorted Array)

- Step 1 – Select the middle item in the array and compare it with the key value to be searched. If it is matched, return the position of the median.
- Step 2 – If it does not match the key value, check if the key value is either greater than or less than the median value.
- Step 3 – If the key is greater, perform the search in the right sub-array; but if the key is lower than the median value, perform the search in the left sub-array.
- Step 4 – Repeat Steps 1, 2 and 3 iteratively, until the size of sub-array becomes 1.
- Step 5 – If the key value does not exist in the array, then the algorithm returns an unsuccessful search.

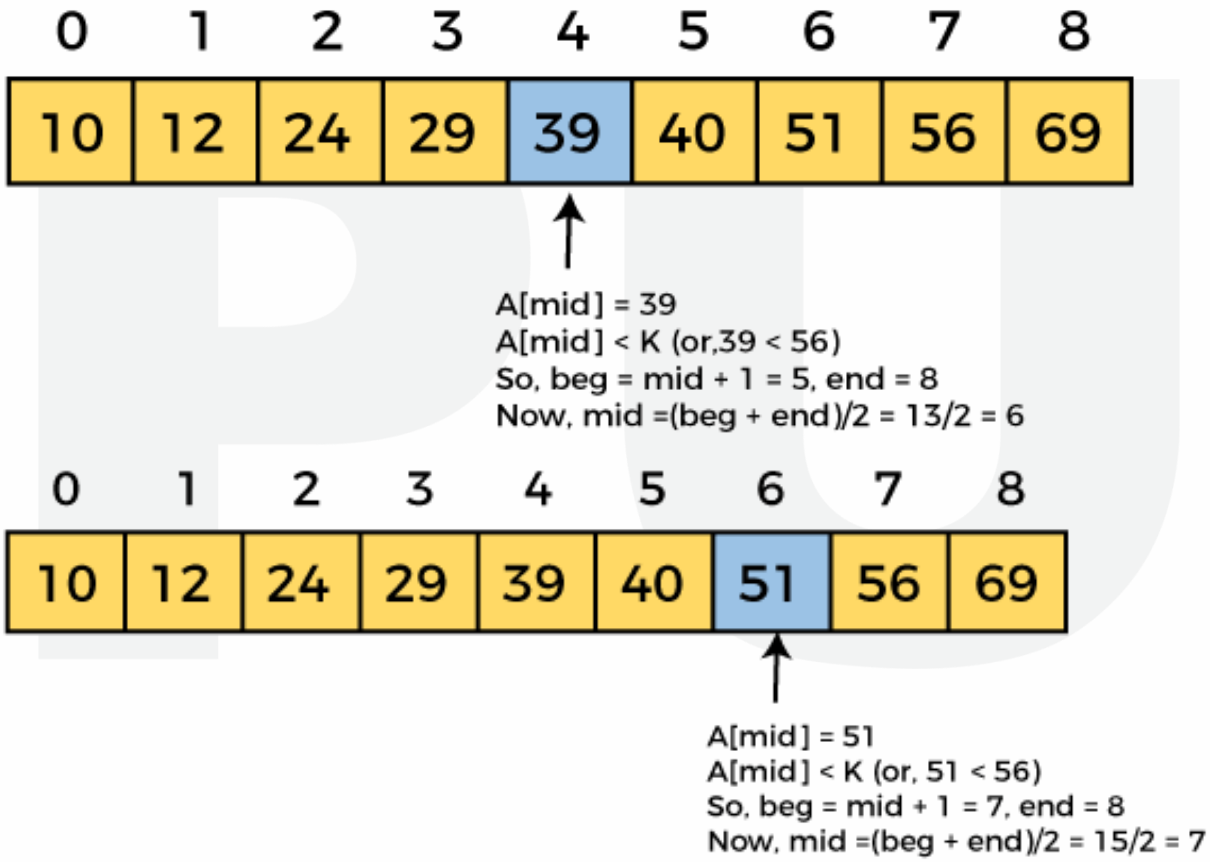
Binary Search Algorithm Example

- Let the element to search is, K = 56

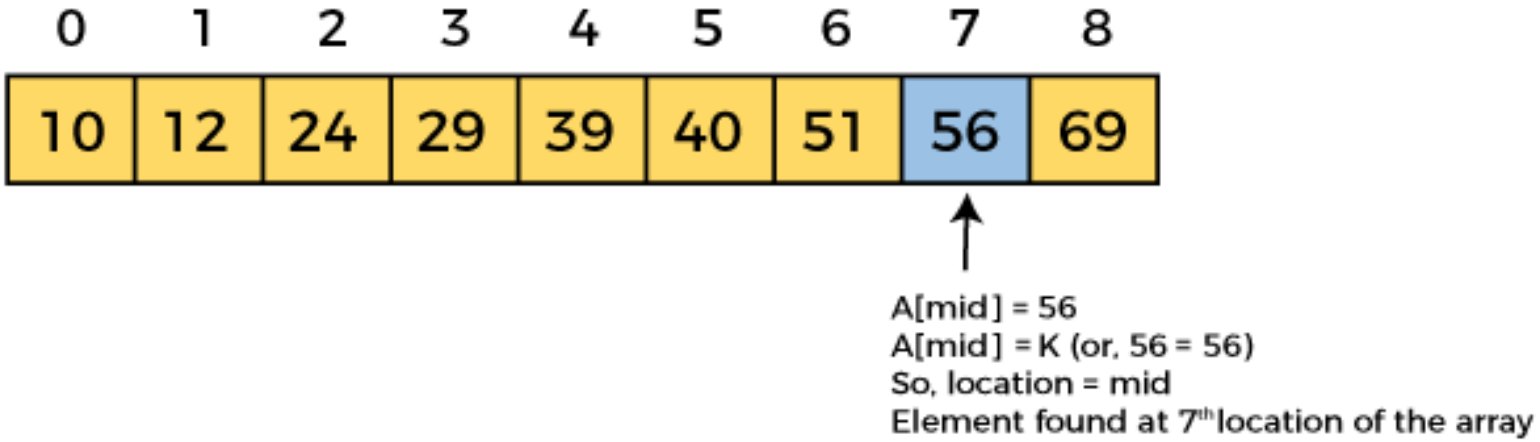
0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

- We have to use the below formula to calculate the mid of the array-
- mid = (beg + end)/2
- So, in the given array -
- beg = 0,
- end = 8
- mid = (0 + 8)/2 = 4. So, now 4 is the mid of the array.

Binary Search Algorithm Example



Binary Search Algorithm Example



- Now, the element to search is found.
- So, algorithm will return the index of the element matched.

What is Sorting?

- Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order.
- Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order.
- Sorting arranges data in a sequence which makes searching easier.

Complexity of Sorting algorithm

- The complexity of sorting algorithm calculates the running time of a function in which ‘n’ number of items are to be sorted.
- The choice for which sorting method is suitable for a problem depends on several dependency configurations for different problems.
- The most noteworthy of these considerations are:
  - The length of time spent by the programmer in programming a specific sorting program
  - Amount of machine time necessary for running the program – Time Complexity
  - The amount of memory necessary for running the program - Space Complexity

Types of Sorting

1. Bubble sort
2. Selection Sort
3. Insertion sort
4. Shell Sort
5. Quick Sort
6. Heap Sort
7. Merge Sort
8. Radix sort.

Bubble Sort Algorithm

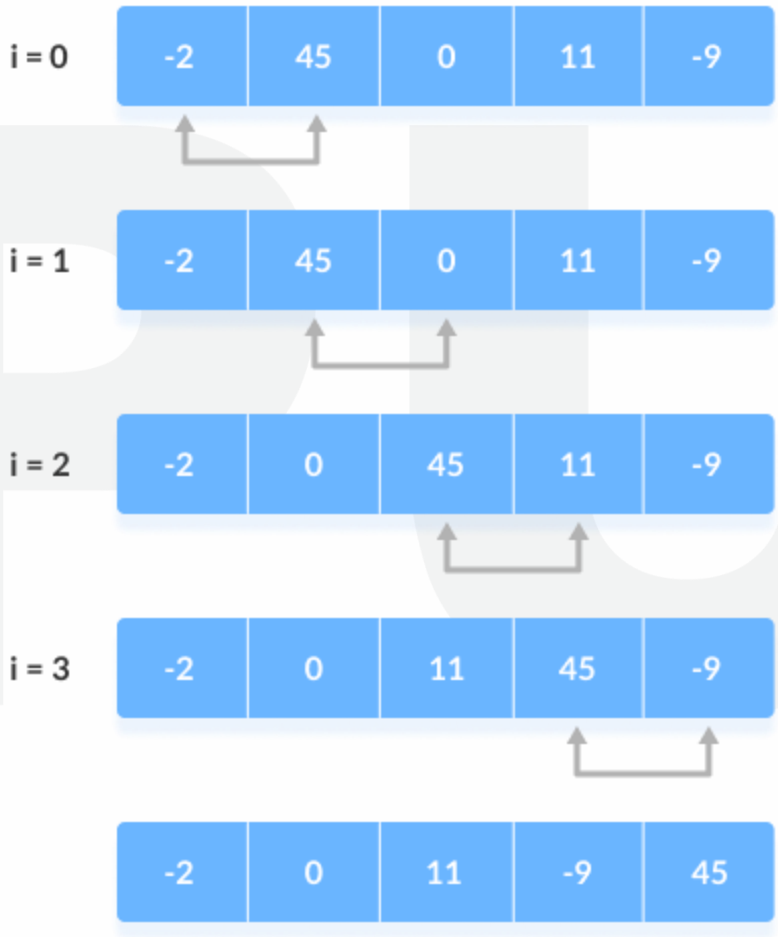
- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.
- We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.
- In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.
- In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.

Bubble Sort Algorithm (Working)

- Suppose we are trying to sort the elements in ascending order.

1. First Iteration (Compare and Swap)
   - Starting from the first index, compare the first and the second elements.
   - If the first element is greater than the second element, they are swapped.
   - Now, compare the second and the third elements. Swap them if they are not in order.
   - The above process goes on until the last element.

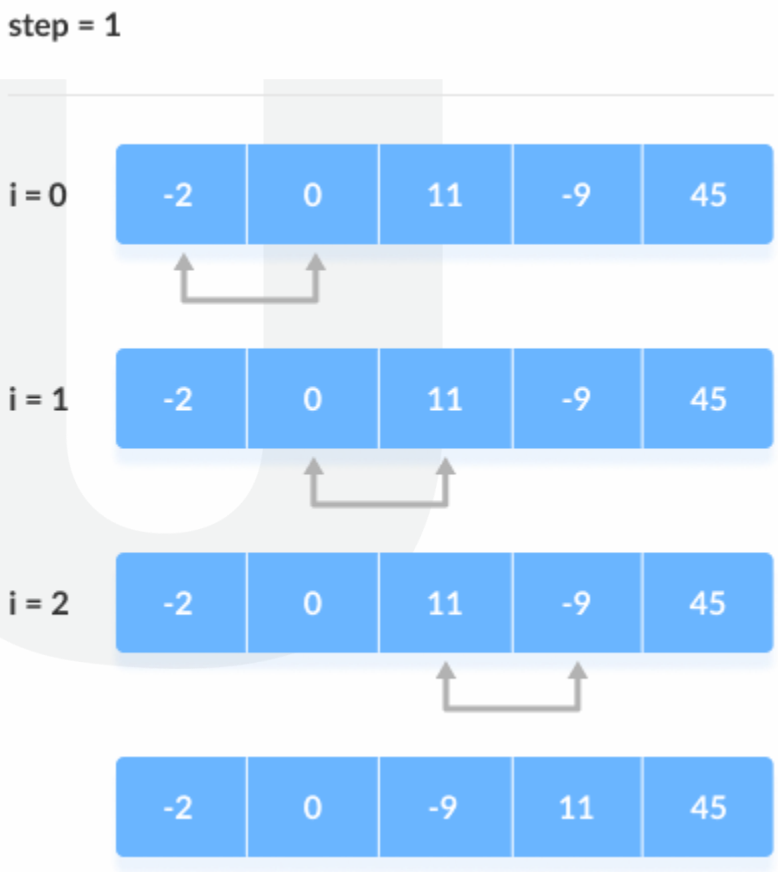
step = 0



Bubble Sort Algorithm (Working)

2. Remaining Iteration (Compare and Swap)

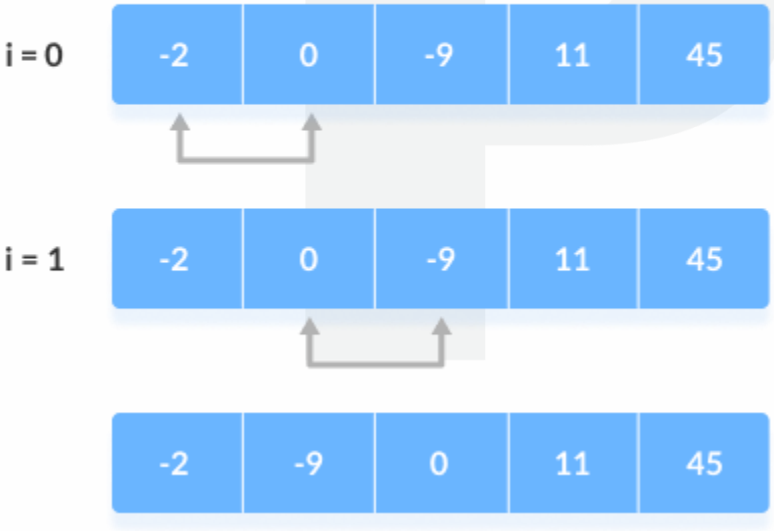
- The same process goes on for the remaining iterations.
- After each iteration, the largest element among the unsorted elements is placed at the end.



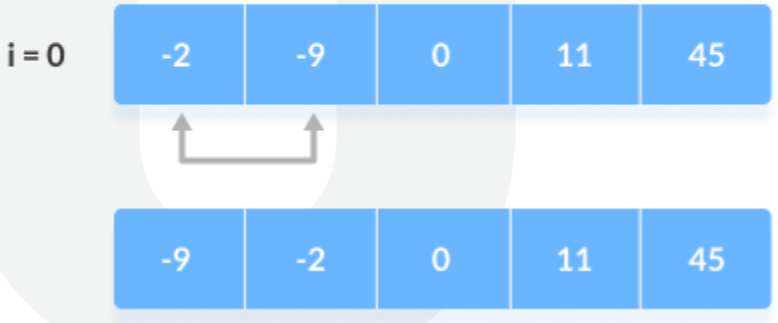
Bubble Sort Algorithm (Working)

In each iteration, the comparison takes place up to the last unsorted element.

step = 2



step = 3



Bubble Sort Algorithm (Code)

```
void main()
{
    int arr[] = { 6, 0, 3, 5 };
    int i, j, temp;
    int n = sizeof(arr);
    for (i = 0; i < n - 1; i++)
    {
        for (j = 0; j < n-i-1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
    for (i = 0; i < n; i++)
        printf("%d, ", arr[i]);
}
```

Selection Sort Algorithm

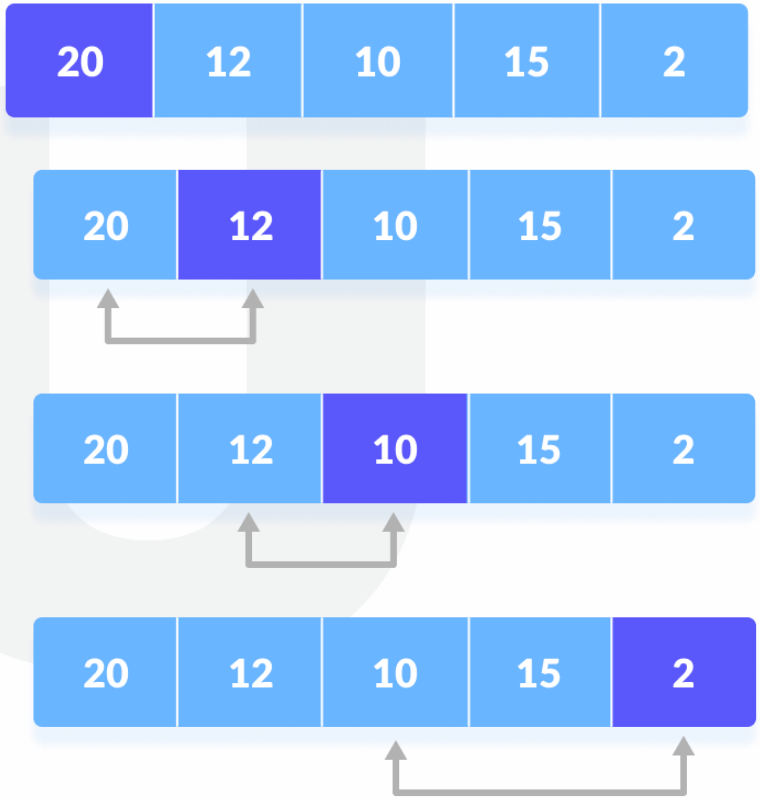
- Selection Sort is a comparison-based sorting algorithm.
- It sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element.
- This process continues until the entire array is sorted.
- First, we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
- Then we find the smallest among remaining elements (or second smallest) and move it to its correct position by swapping.
- We keep doing this until we get all elements moved to correct position.

Selection Sort Algorithm (Working)

1. Set the first element as minimum

2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

   Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

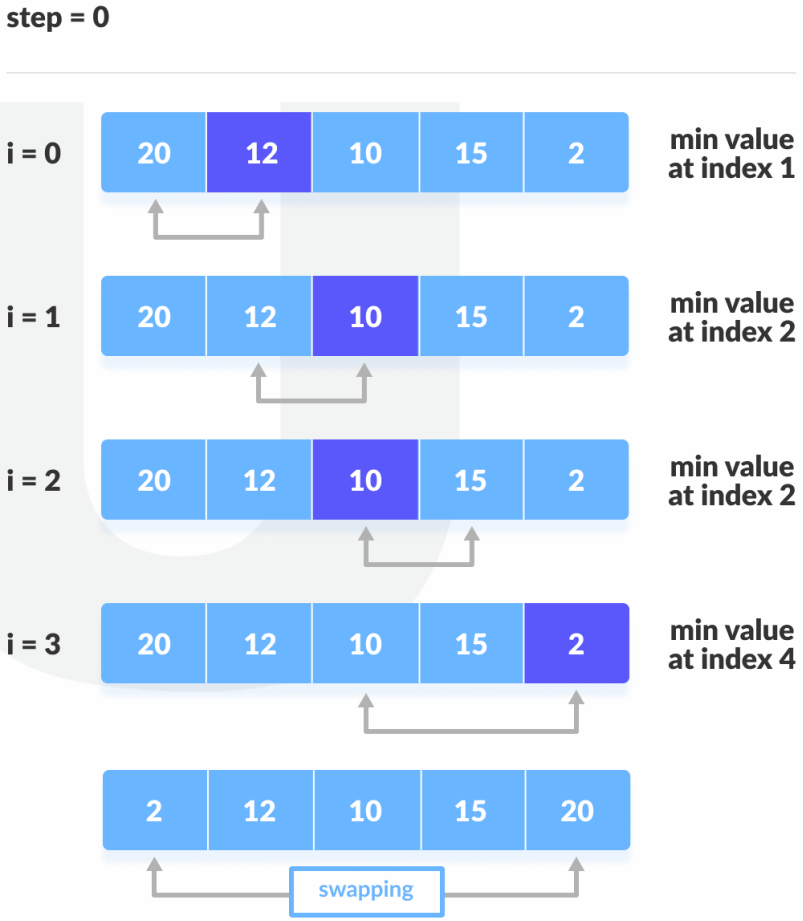


Selection Sort Algorithm (Working)

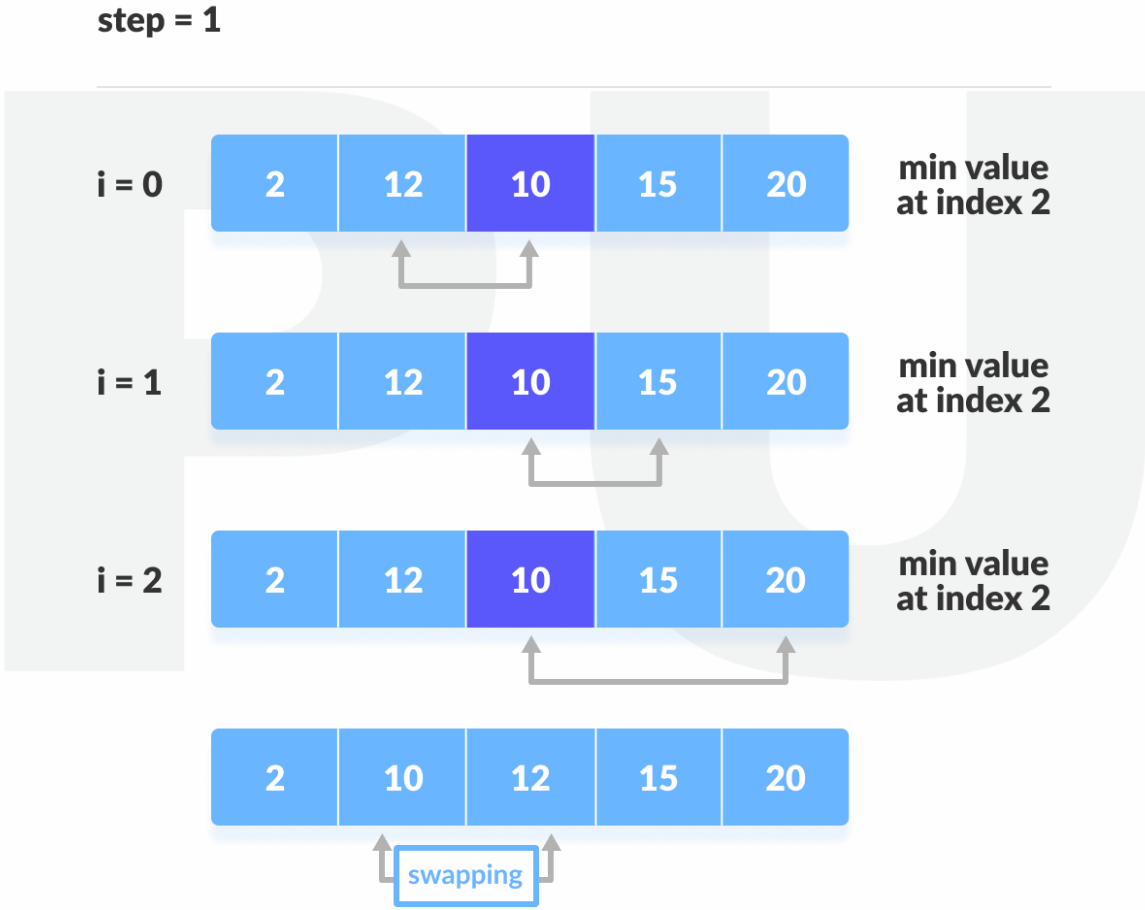
3. After each iteration, minimum is placed in the front of the unsorted list.

4. For each iteration, indexing starts from the first unsorted element.

   Step 1 to 3 are repeated until all the elements are placed at their correct positions.

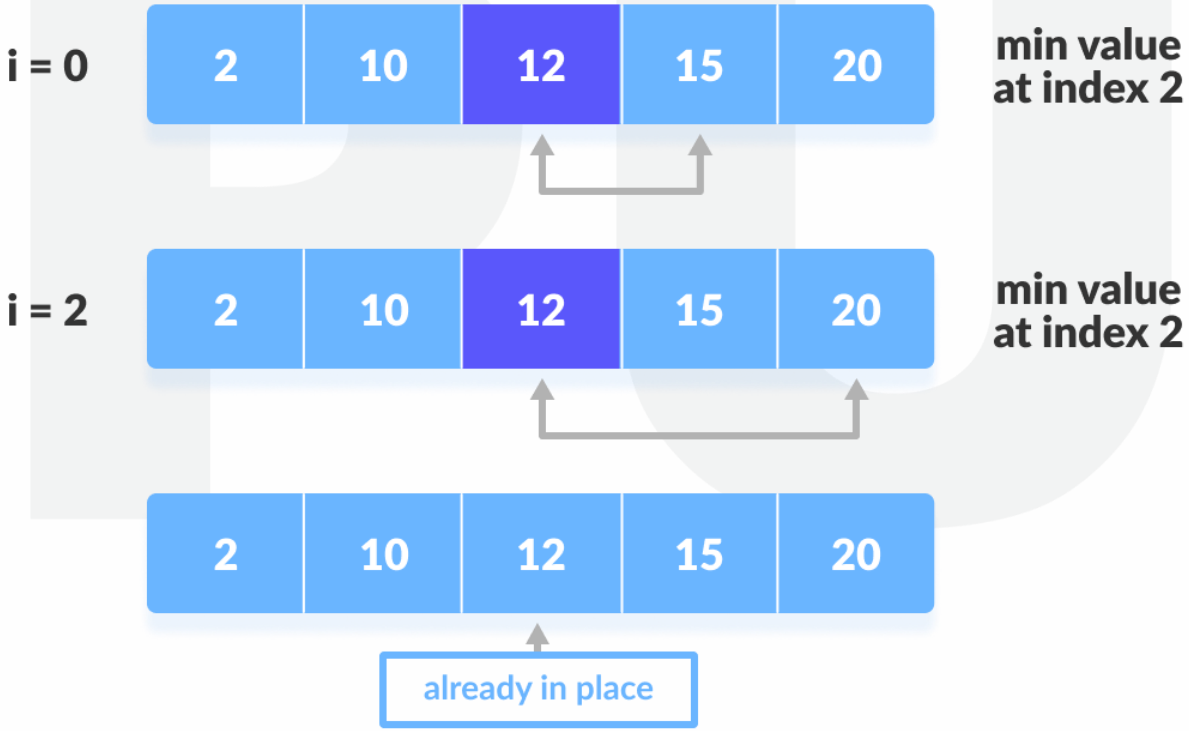


Selection Sort Algorithm (Working)

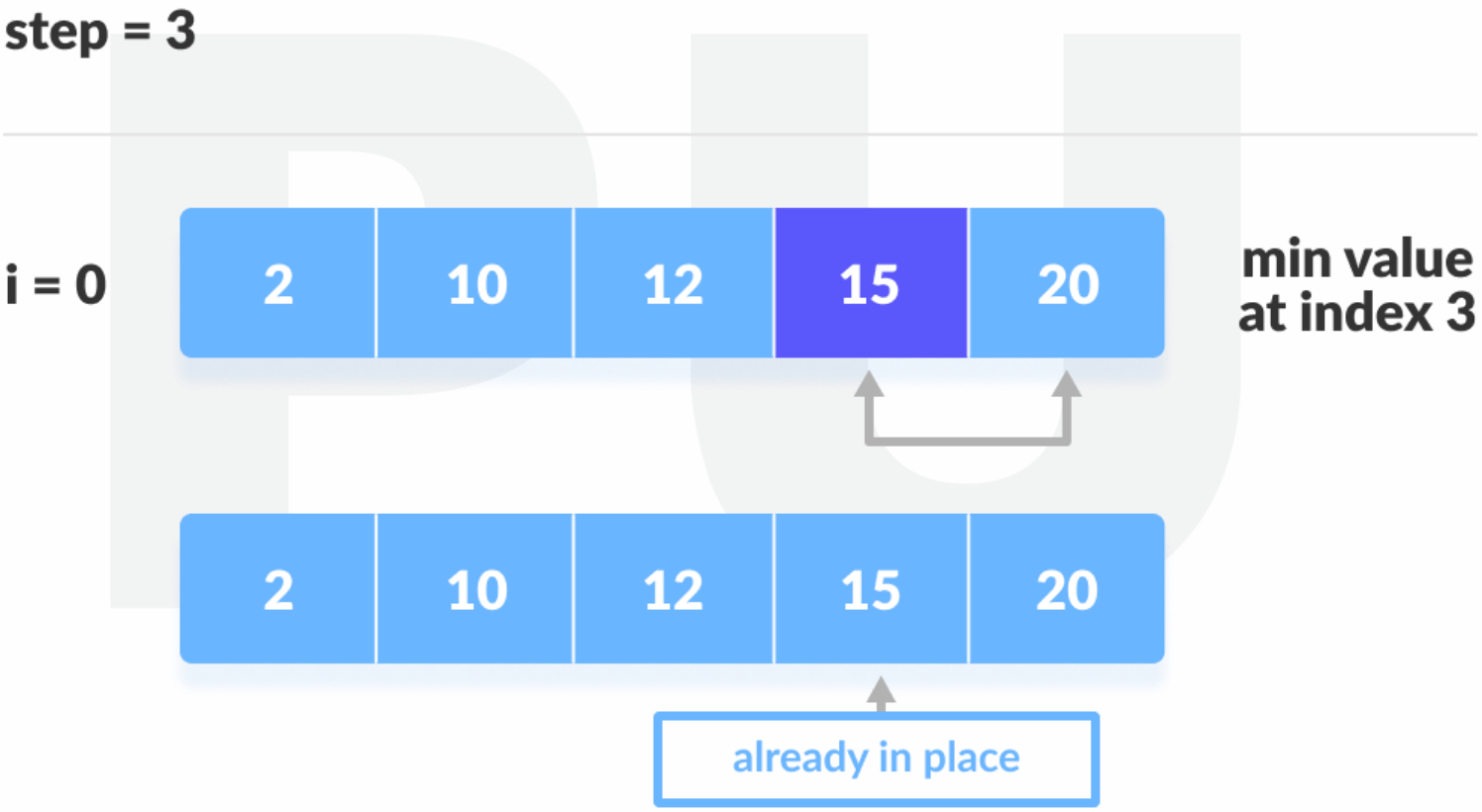


Selection Sort Algorithm (Working)

step = 2



Selection Sort Algorithm (Working)



Selection Sort Algorithm (Code)

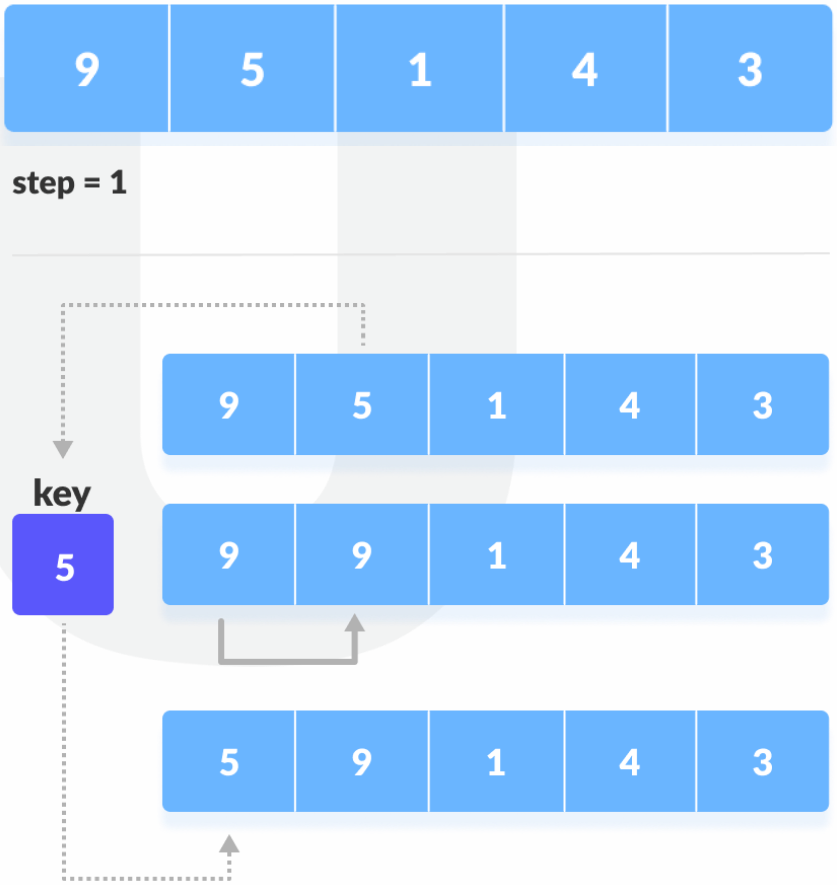
```
void main()
{
    int a[] = { 6, 0, 3, 5 };
    int n = sizeof(a) / sizeof(a[0]);
    int i, j, small, temp;

    for (i = 0; i < n-1; i++)
    {
        small = i;
        for (j = i+1; j < n; j++)
        {
            if (a[j] < a[small])
                small = j;
        }

        temp = a[small];
        a[small] = a[i];
        a[i] = temp;
    }
    for (i = 0; i < n; i++)
        printf("%d, ", a[i]);
}
```

Insertion Sort Algorithm

- Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.
- We start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the first two elements and put at its correct position
- Repeat until the entire array is sorted.

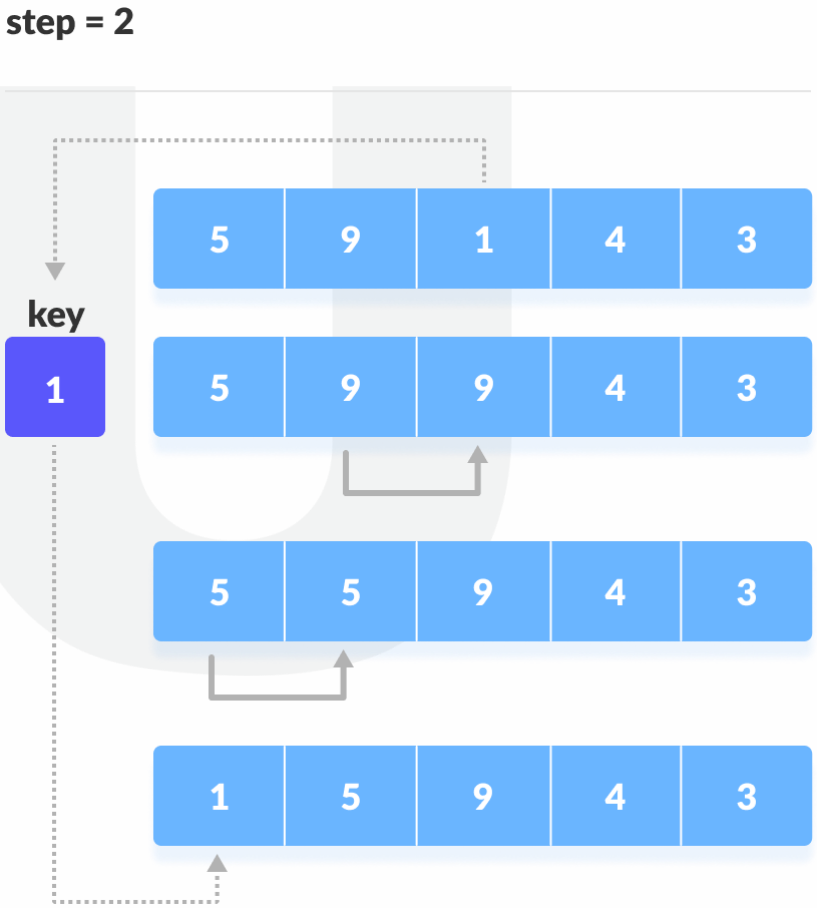
Insertion Sort Algorithm (Working)

- Suppose we need to sort the following array.
- The first element in the array is assumed to be sorted. Take the second element and store it separately in key.
- Compare key with the first element.
- If the first element is greater than key, then key is placed in front of the first element.



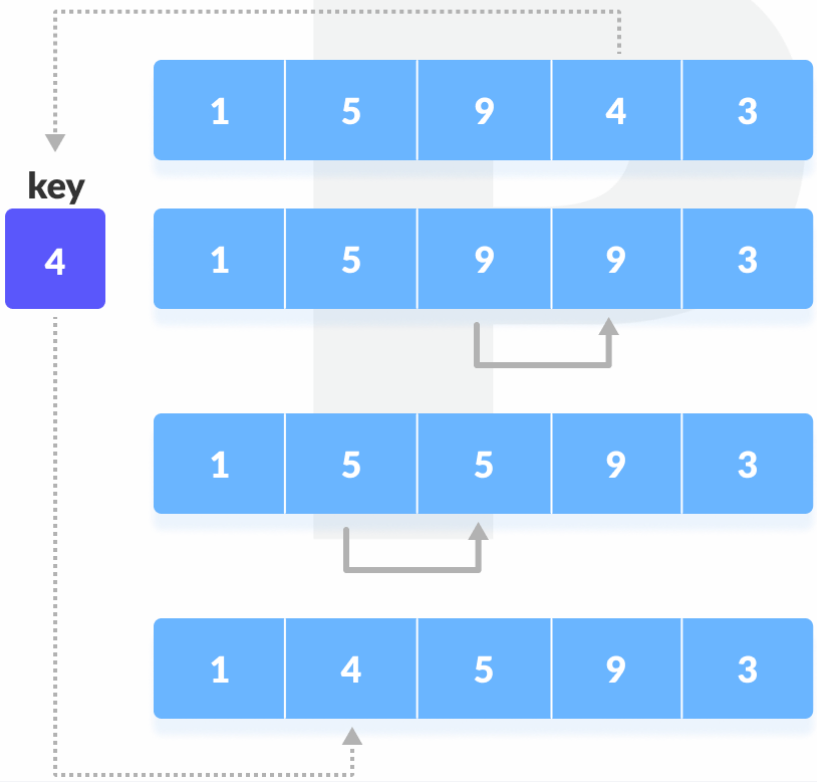
Insertion Sort Algorithm (Working)

- Now, the first two elements are sorted.
- Take the third element and compare it with the elements on the left of it.
- Placed it just behind the element smaller than it.
- If there is no element smaller than it, then place 1 at the beginning of the array.
- Similarly, place every unsorted element at its correct position.

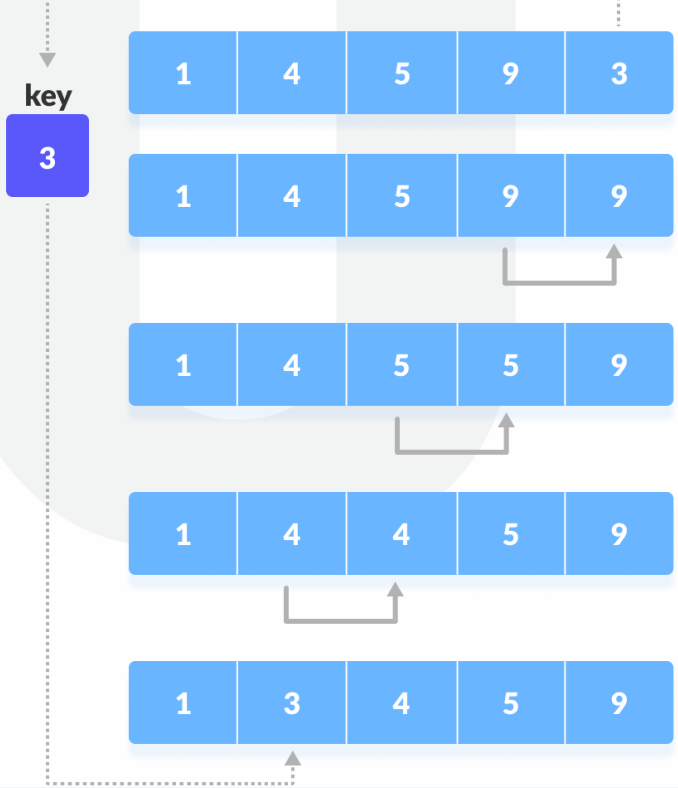


Insertion Sort Algorithm (Working)

step = 3



step = 4



Insertion Sort Algorithm (Code)

```
void main()
{
    int a[] = { 6, 0, 3, 5 };
    int n = sizeof(a) / sizeof(a[0]);
    int i, j, key;

    for (i = 1; i < n; i++)
    {
        key = a[i];
        for (j = i-1; j >=0 && key <
                                a[j]; --j)
        {
            a[j + 1] = a [j];
        }
        a[j + 1] = key;
    }
    for (i = 0; i < n; i++)
        printf("%d, ", a[i]);
}
```

Shell Sort Algorithm

- It is a sorting algorithm that is an extended version of insertion sort.
- Shell sort has improved the average time complexity of insertion sort. As similar to insertion sort, it is a comparison-based and in-place sorting algorithm. Shell sort is efficient for medium-sized data sets.
- In insertion sort, at a time, elements can be moved ahead by one position only. To move an element to a far-away position, many movements are required that increase the algorithm's execution time. But shell sort overcomes this drawback of insertion sort. It allows the movement and swapping of far-away elements as well.
- This algorithm first sorts the elements that are far away from each other, then it subsequently reduces the gap between them. This gap is called as interval.

Shell Sort Algorithm (Working)

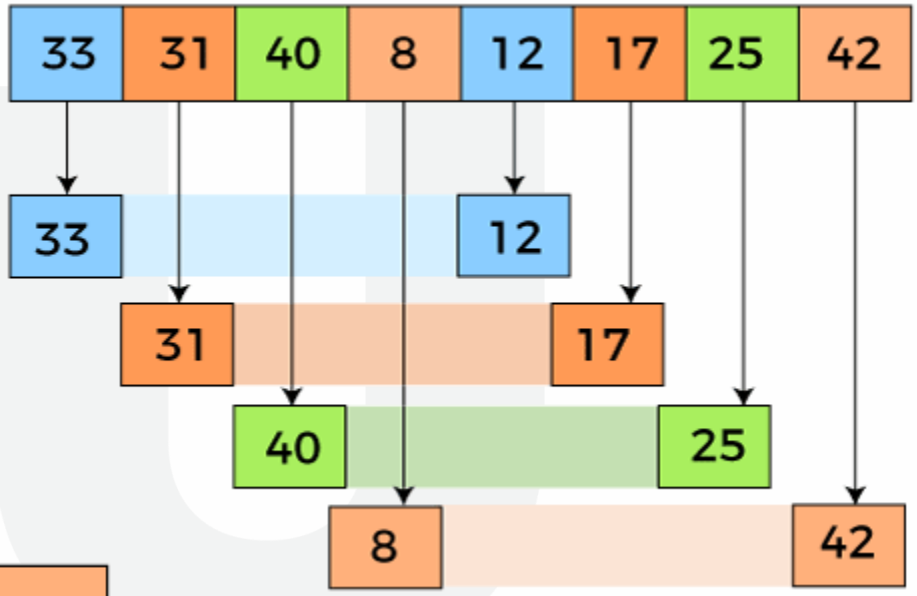
- Let the elements of array are:

33	31	40	8	12	17	25	42

- We will use the original sequence of shell sort, i.e., N/2, N/4,....,1 as the intervals.
- In the first loop, n is equal to 8 (size of the array), so the elements are lying at the interval of 4 (n/2 = 4). Elements will be compared and swapped if they are not in order.
- Here, in the first loop, the element at the 0^{th} position will be compared with the element at 4^{th} position. If the 0^{th} element is greater, it will be swapped with the element at 4^{th} position. Otherwise, it remains the same. This process will continue for the remaining elements.
- At the interval of 4, the sub-lists are {33, 12}, {31, 17}, {40, 25}, {8, 42}.

Shell Sort Algorithm (Working)

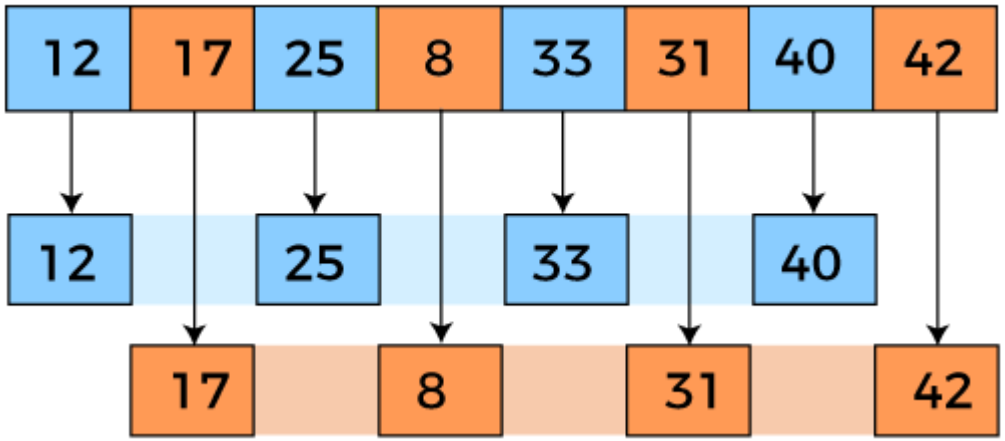
- Now, we have to compare the values in every sub-list.
- After comparing, we have to swap them if required in the original array.
- After comparing and swapping, the updated array will look as follows -



Shell Sort Algorithm (Working)

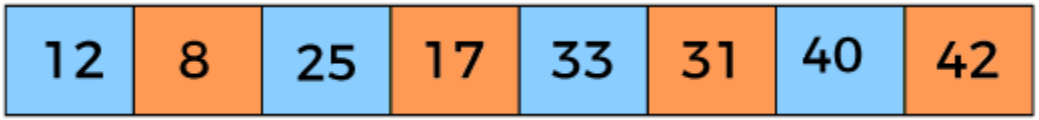
- In the second loop, elements are lying at the interval of 2 (n/4 = 2),

  where n = 8.
- Now, we are taking the interval of 2 to sort the rest of the array.
- With an interval of 2, two sub-lists will be generated - {12, 25, 33, 40}, and {17, 8, 31, 42}.



Shell Sort Algorithm (Working)

- Now, we again have to compare the values in every sub-list.
- After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows:



- In the third loop, elements are lying at the interval of 1 (n/8 = 1), where n = 8.
- At last, we use the interval of value 1 to sort the rest of the array elements. In this step, shell sort uses insertion sort to sort the array elements.

Shell Sort Algorithm (Working)

- Now, the array is sorted in ascending order.

12	8	25	17	33	31	40	42
12	8	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	25	17	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	33	31	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42
8	12	17	25	31	33	40	42

Shell Sort Algorithm (Code)

```
void main()
{
	int array[]={ 6, 0, 3, 5 }, n, i, j, gap, temp;
	for (gap = n / 2; gap > 0; gap /= 2)
	{
		for (i = gap; i < n; i++)
		{
			temp = array[i];
			for (j = i; j >= gap && array[j - gap] > temp; j -= gap)
				array[j] = array[j - gap];
			array[j] = temp;
		}
	}
	for (i = 0; i < n; i++)
	printf("%d ", array[i]);
}
```

QuickSort Algorithm

- QuickSort works on the principle of divide and conquer, breaking down the problem into smaller sub-problems.
- There are mainly three steps in the algorithm:
- **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
- **Partition the Array:** Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
- **Recursively Call:** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
- **Base Case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

QuickSort Algorithm (Working)

1. Select the Pivot Element

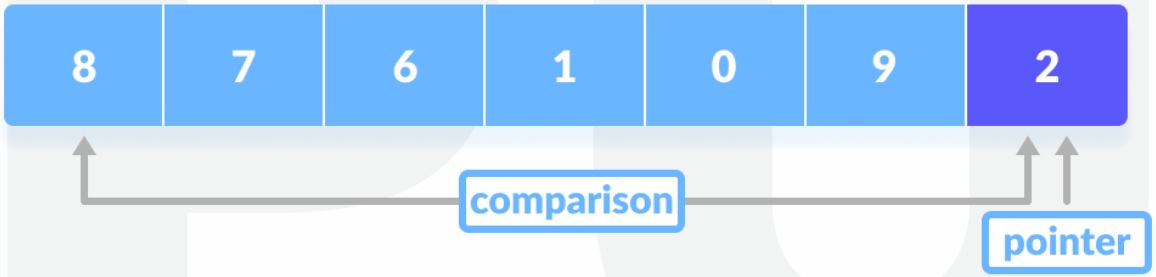
- There are different variations of quicksort where the pivot element is selected from different positions.
- Here, we will be selecting the rightmost element of the array as the pivot element.

8	7	6	1	0	9	2

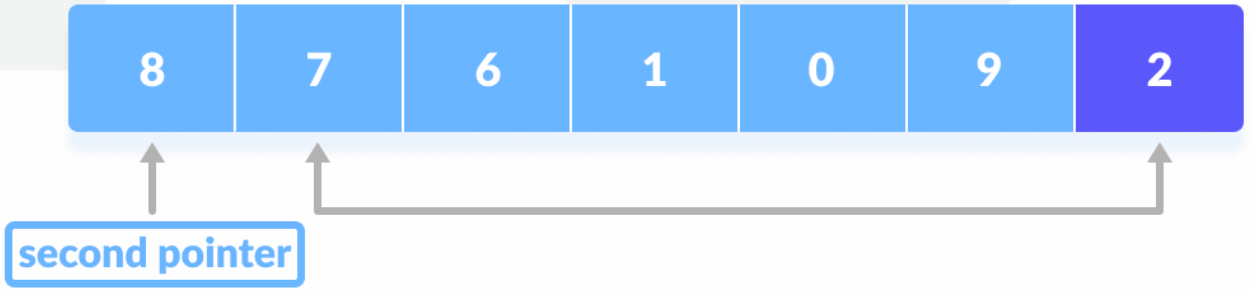
QuickSort Algorithm (Working)

2. Rearrange the Array

- A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.

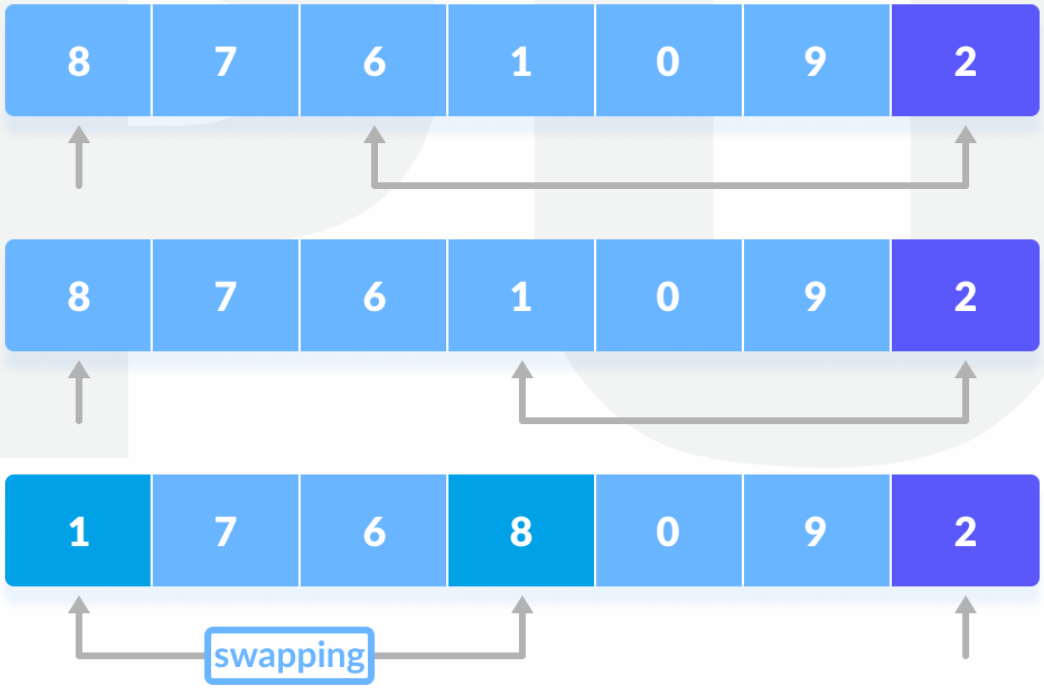


- If the element is greater than the pivot element, a second pointer is set for that element.



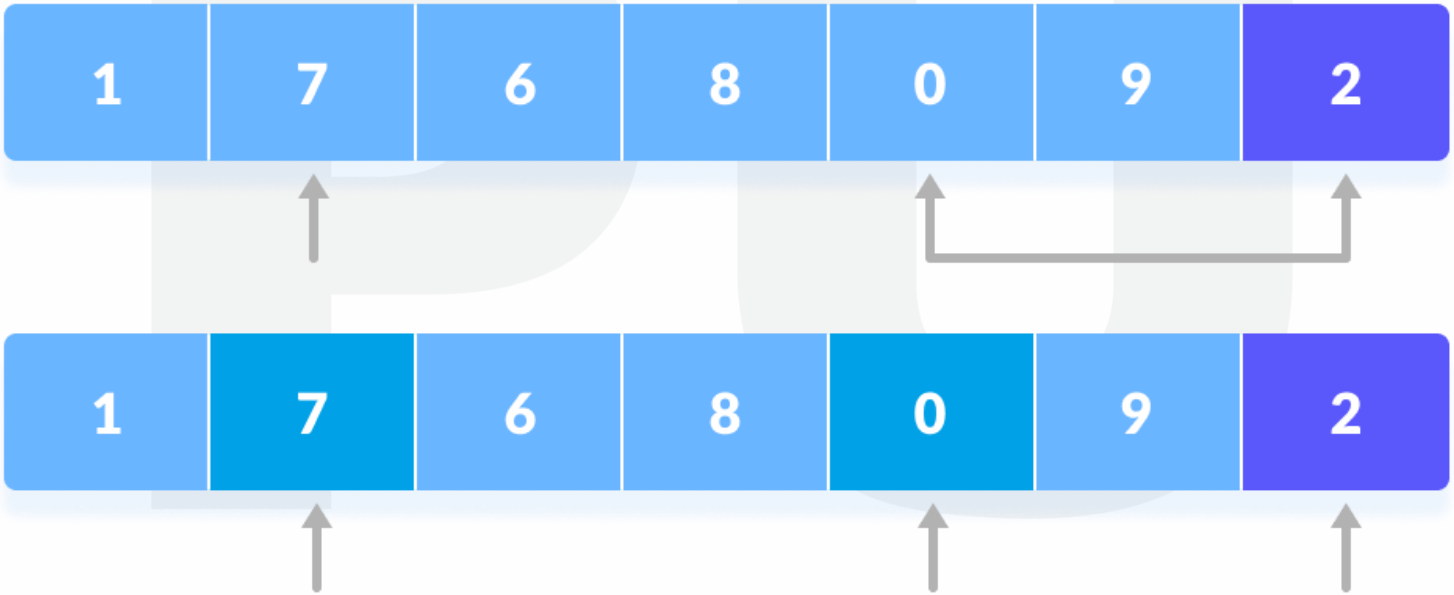
QuickSort Algorithm (Working)

- Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



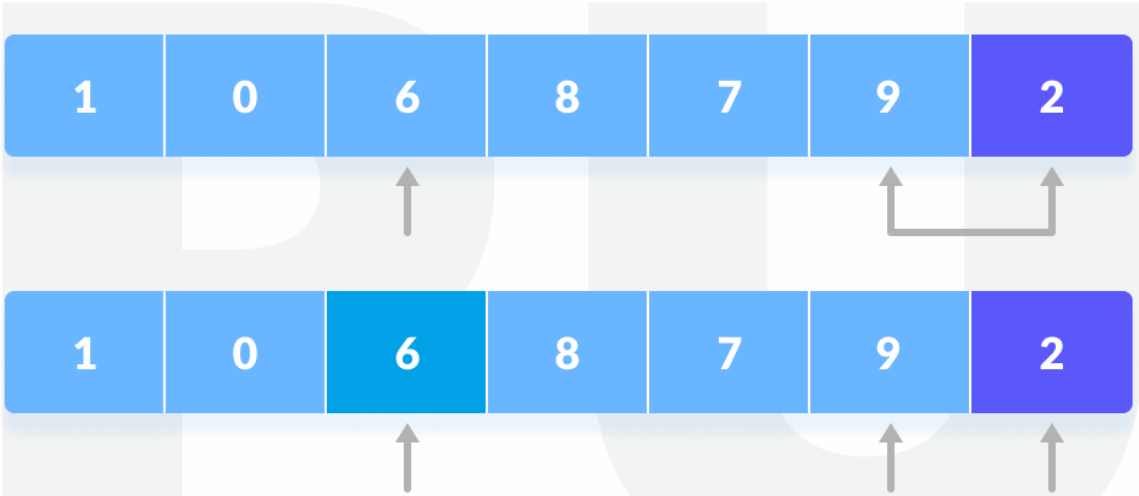
QuickSort Algorithm (Working)

- Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.



QuickSort Algorithm (Working)

- The process goes on until the second last element is reached.

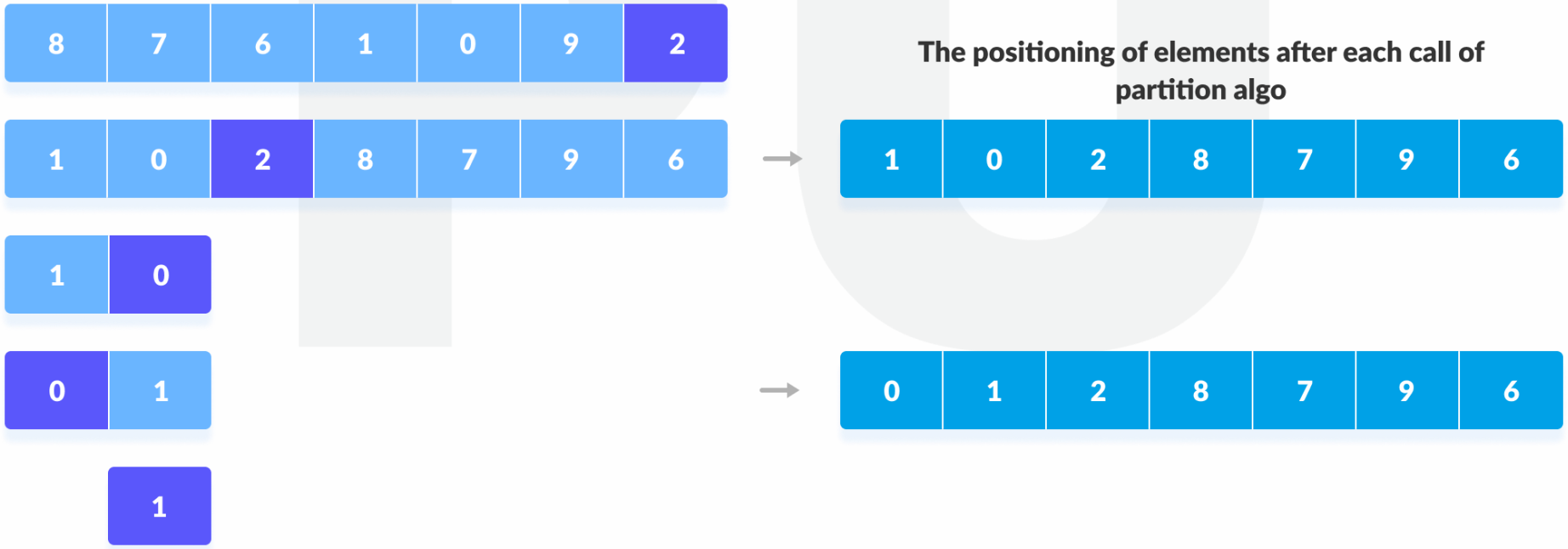


- Finally, the pivot element is swapped with the second pointer.

QuickSort Algorithm (Working)

3. Divide Subarrays: Pivot elements are again chosen for the left and the right sub-parts separately. And, step 2 is repeated.

quicksort(arr, low, pi-1)



QuickSort Algorithm (Working)

quicksort(arr, pi+1, high)

QuickSort Algorithm (Code)

```
int partition(int arr[], int low, int high)
{
        int pivot = arr[high], temp;
        int i = (low - 1);

        for (int j = low; j <= high - 1; j++)
        {
                if (arr[j] < pivot)
                {
                        i++;
                        temp = arr[i];
                        arr[i] = arr[j];
                        arr[j] = temp;
                }
        }
        // Swap arr[i + 1] and
        //arr[high] (or pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return (i + 1);
}
```

QuickSort Algorithm (Code)

```
void quickSort(int arr[], int low, int high)
{
	if (low < high)
	{
		int pi = partition(arr, low, high);

		// Recursively sort elements
		//before and after partition
		quickSort(arr, low, pi - 1);
		quickSort(arr, pi + 1, high);
	}
}
```

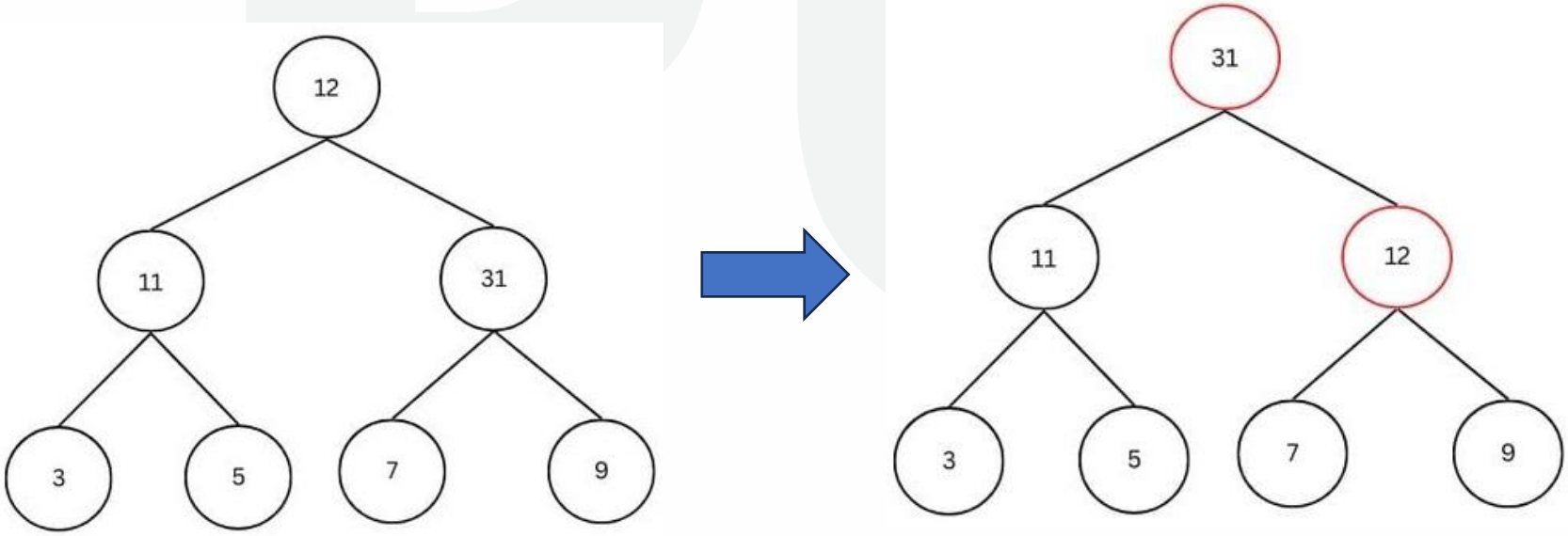
```
void main()
{
	int arr[] = {10, 7, 8, 9, 1, 5};
	int n = sizeof(arr) / sizeof(arr[0]);

	int low = 0, high = n - 1;

	quickSort(arr, low, high);

	printf("Sorted array: ");
	for (int i = 0; i < n; i++)
		printf("%d ", arr[i]);
}
```

Heap Sort Algorithm

- Heap sort is a comparison-based, in-place sorting algorithm that visualizes the elements of the array in the form of a heap data structure to sort it.
- It divides the input array into two parts; sorted region, and unsorted region. The sorted region is initially empty, while the unsorted region contains all the elements.
- The largest element from the unsorted region is picked iteratively and added to the sorted region. The algorithm arranges the unsorted region in a heap.
- The heap sort algorithm is the combination of two other sorting algorithms: insertion sort and merge sort.

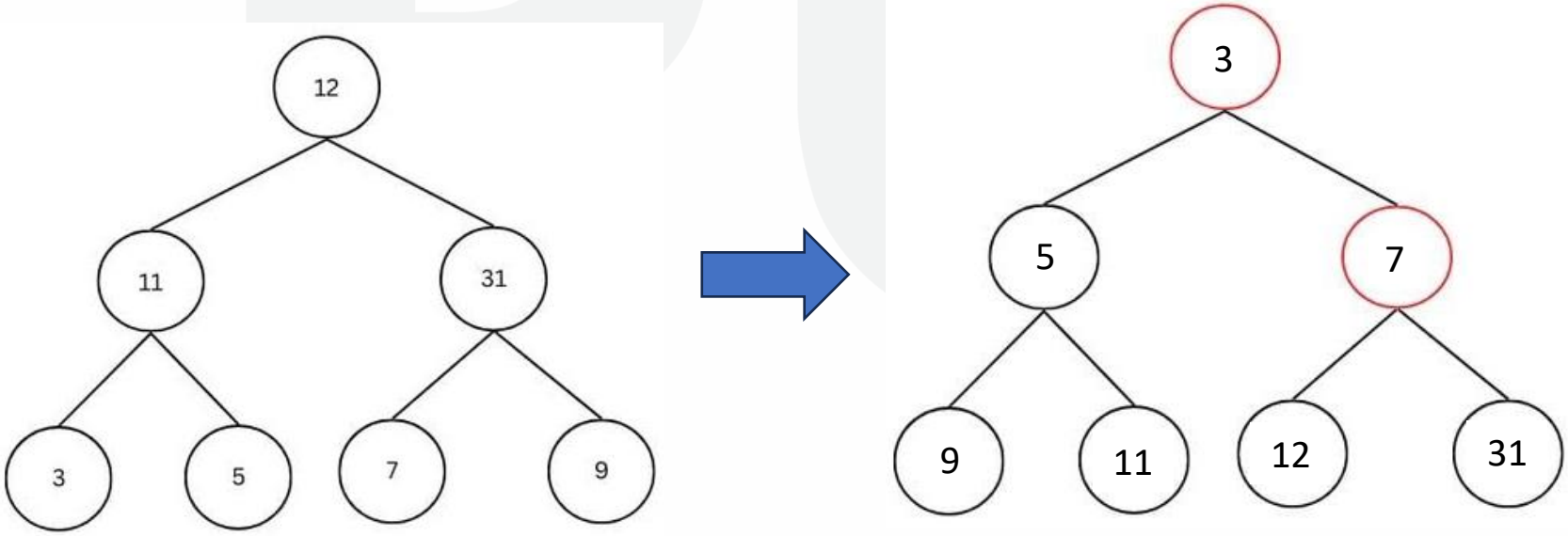
Heap Sort Algorithm

Max-Heap: All parent nodes must have values that are greater than or equal to the values of their children. This will ensure that the highest value is always at the root of the tree. For the tree above, this means swapping node 12 and node 31 positions in the tree to satisfy the requirements for a max-heap tree. Suppose the array elements are: [12, 11, 31, 3, 5, 7, 9]



Heap Sort Algorithm

Min-Heap: All parent nodes must have values that are less than or equal to the values of their children. This will ensure that the lowest value is always at the root of the tree. For the tree above, this means swapping node 3 and node 7 positions in the tree to satisfy the requirements for a min-heap tree. Suppose the array elements are: [12, 11, 31, 3, 5, 7, 9]

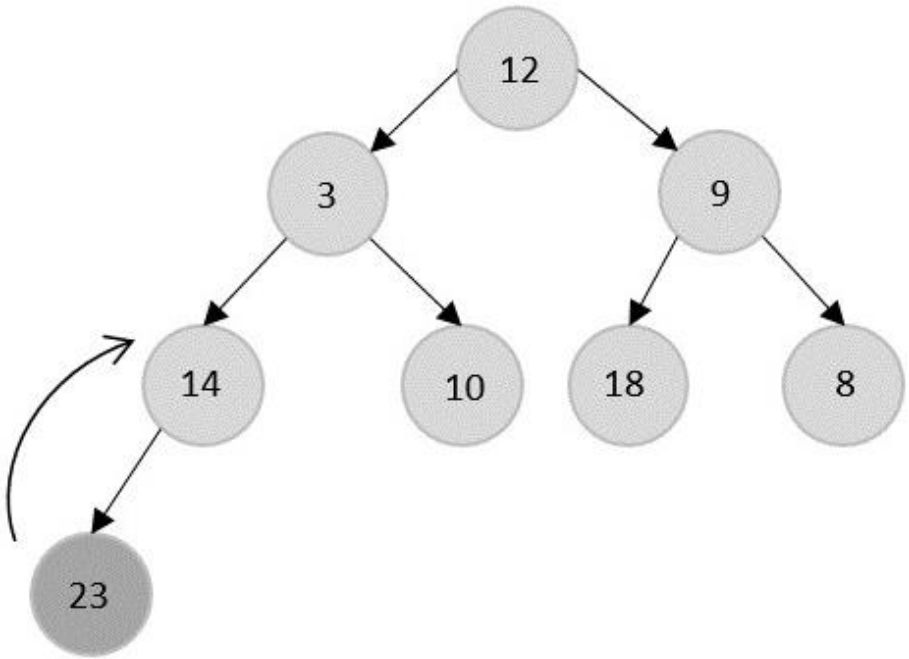


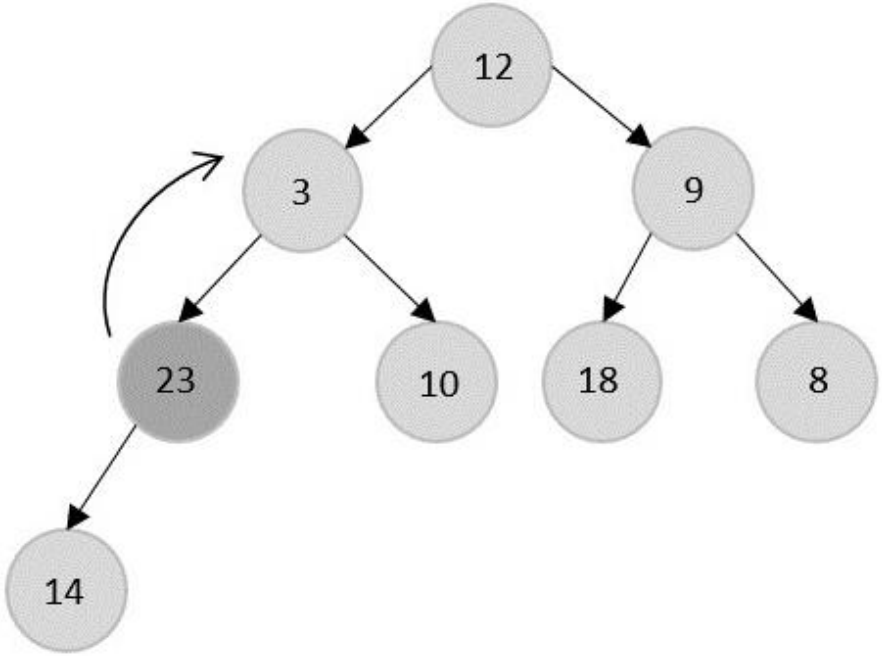
Heap Sort Algorithm (Working)

Let the elements of array are:

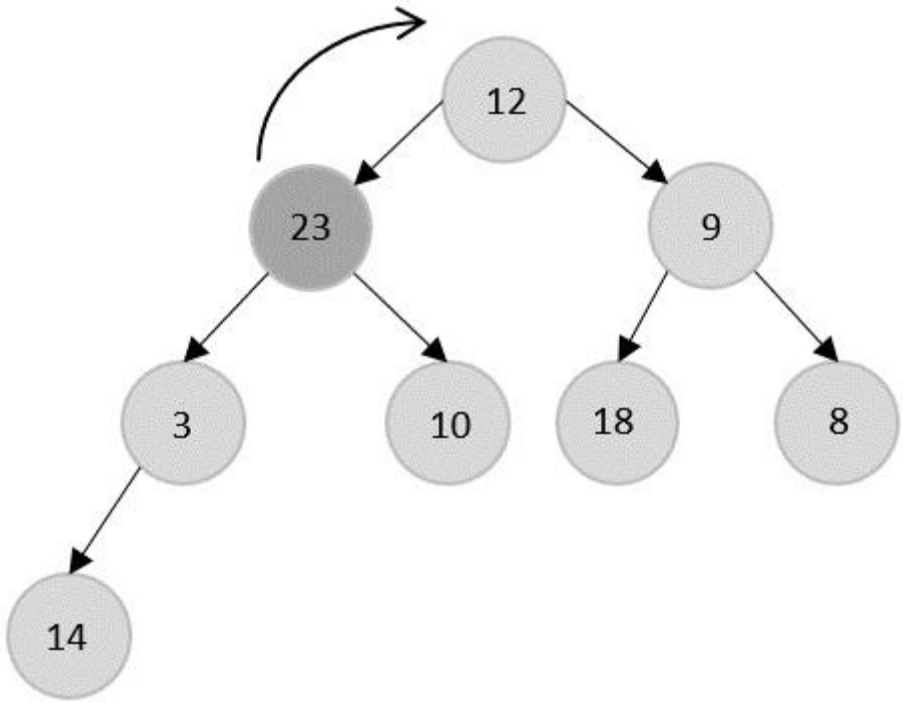
12	3	9	14	10	18	8	23

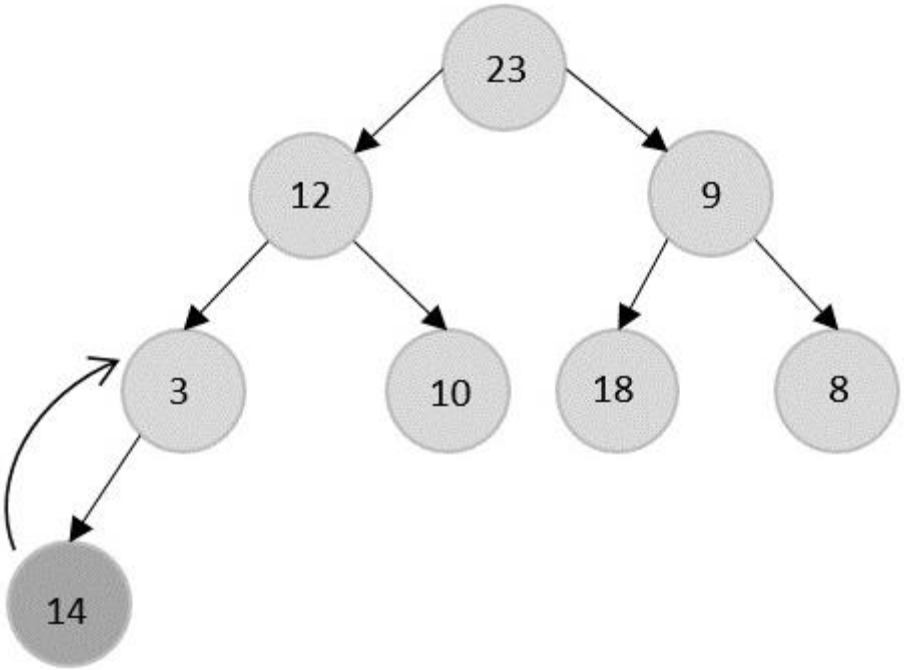
Building a heap using the Max-Heap algorithm from the input array



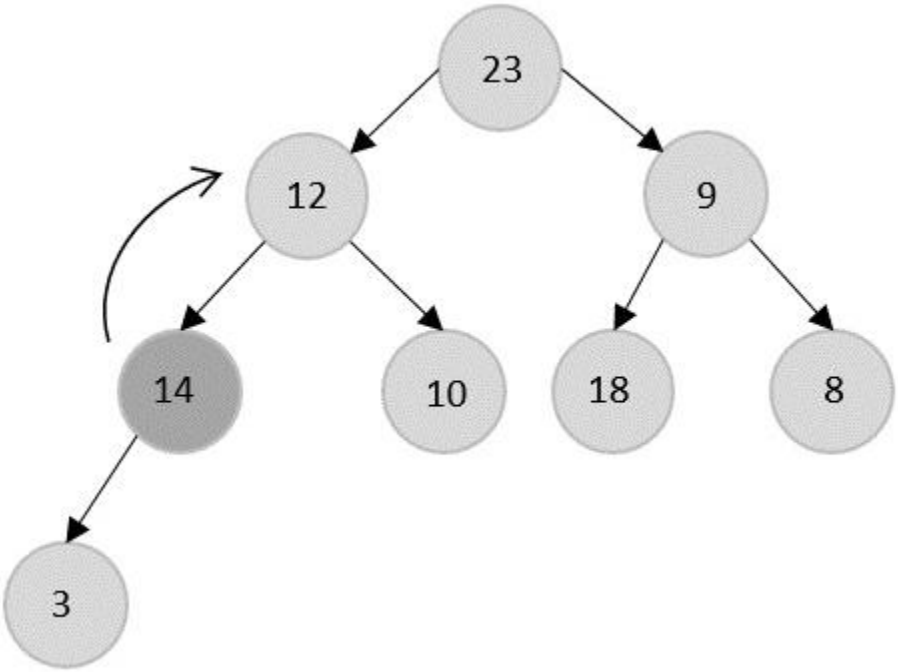


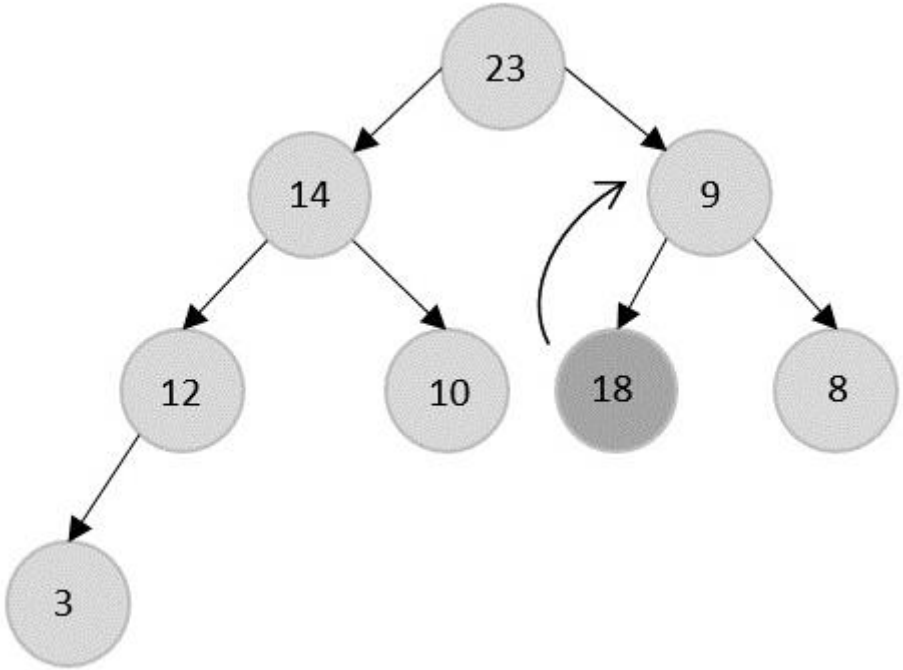
Heap Sort Algorithm (Working)





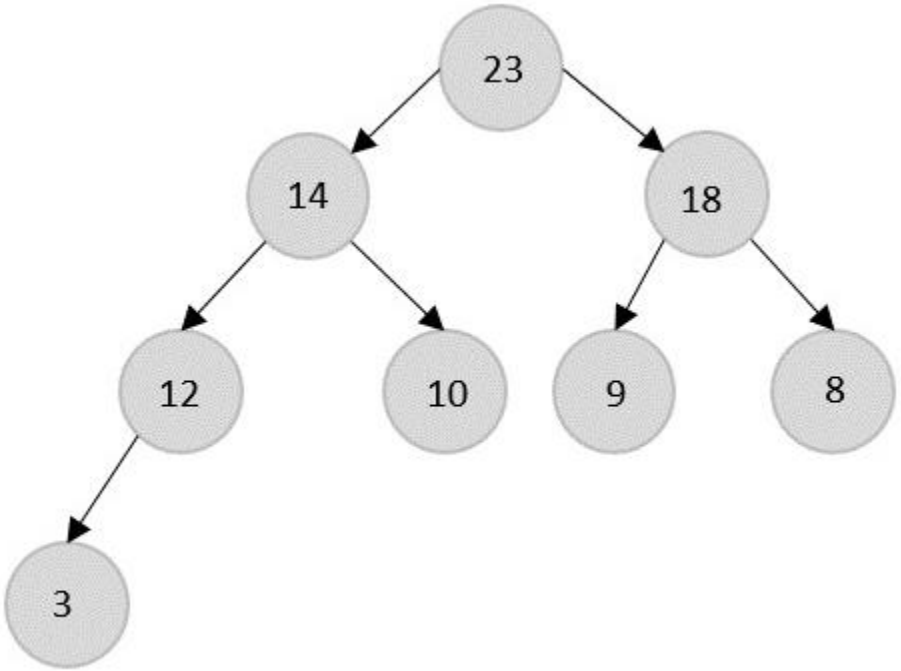
Heap Sort Algorithm (Working)





Heap Sort Algorithm (Working)

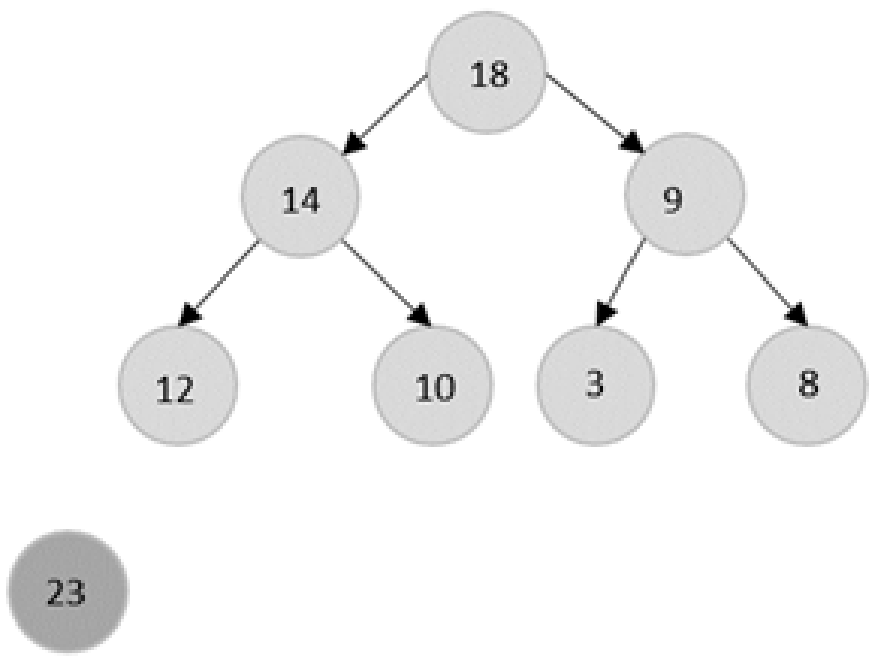
Final binary tree after Max-Heap Algorithm:



Heap Sort Algorithm (Working)

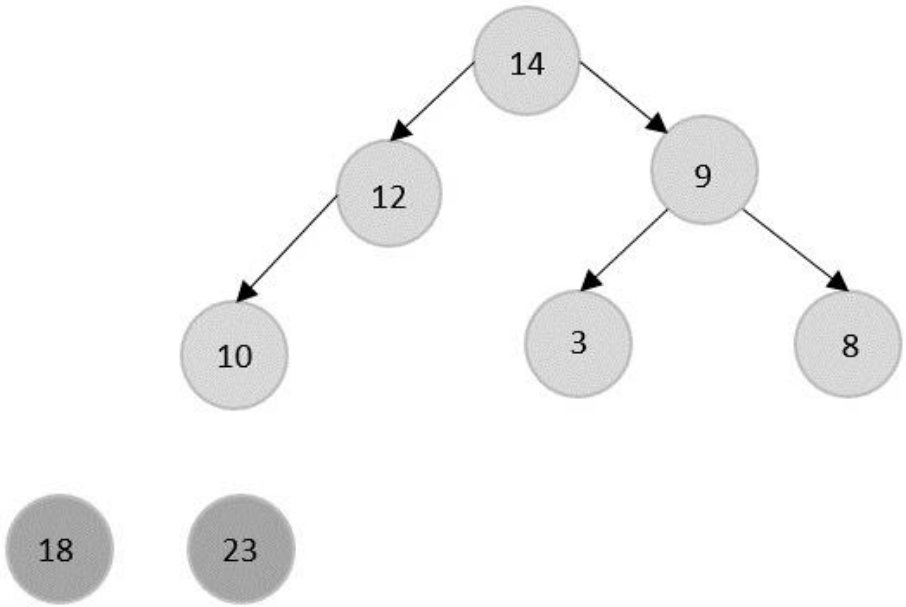
The Heapify Algorithm: Applying the heapify method, remove the root node from the heap and replace it with the next immediate maximum valued child of the root.

The root node is 23, so, 23 is popped and 18 is made the next root because it is the next maximum node in the heap.

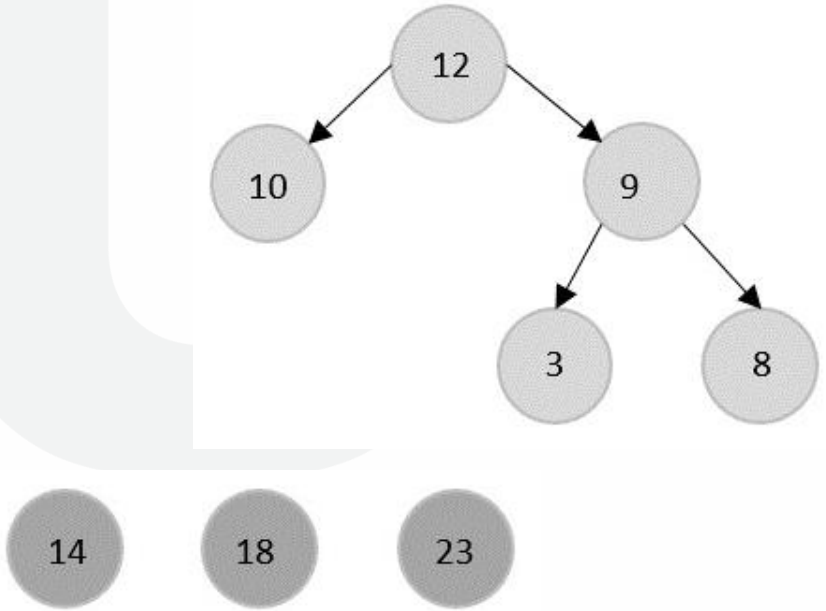


Heap Sort Algorithm (Working)

Now, 18 is popped after 23 which is replaced by 14.

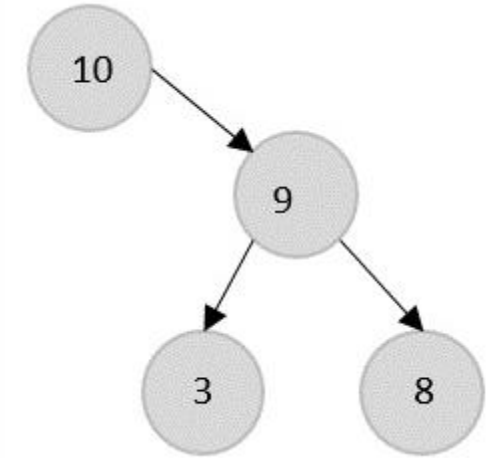


The current root 14 is popped from the heap and is replaced by 12.

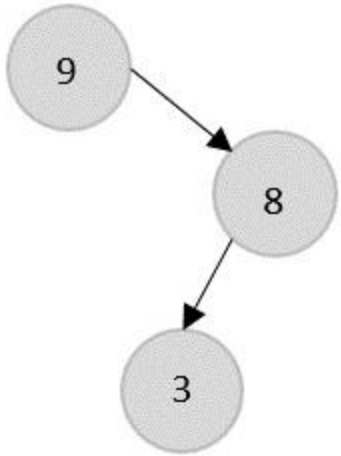


Heap Sort Algorithm (Working)

12 is popped and replaced with 10.

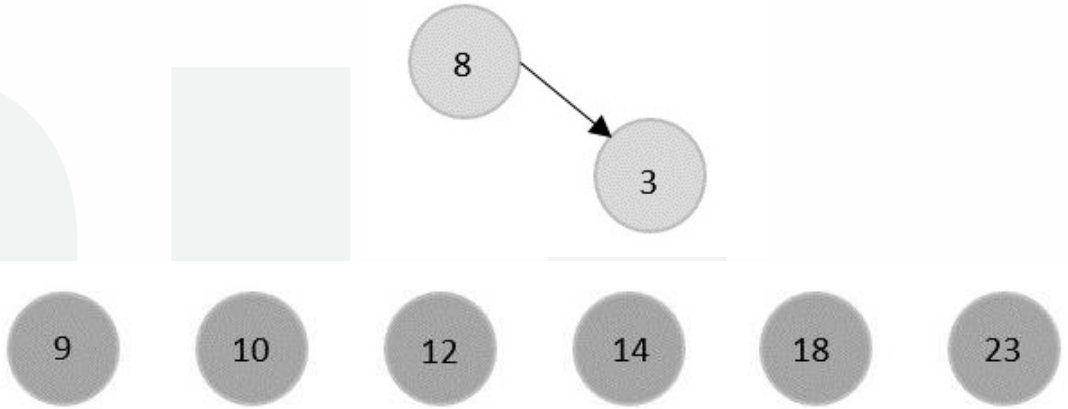


The current root 14 is popped from the heap and is replaced by 12.

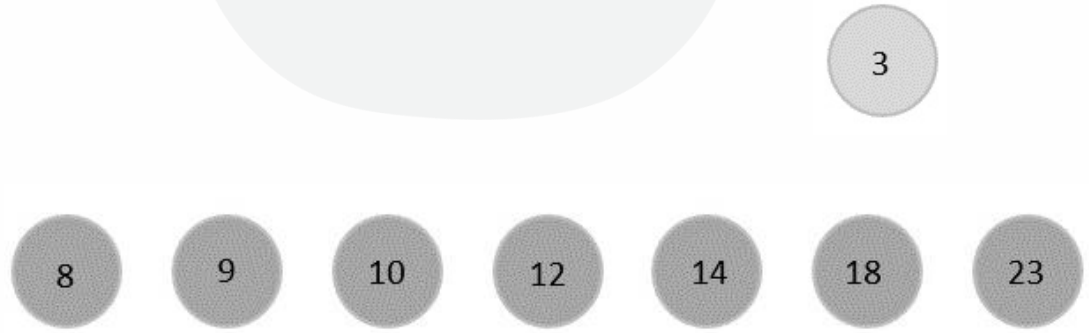


Heap Sort Algorithm (Working)

Here the current root element 9 is popped and the elements 8 and 3 are remained in the tree.

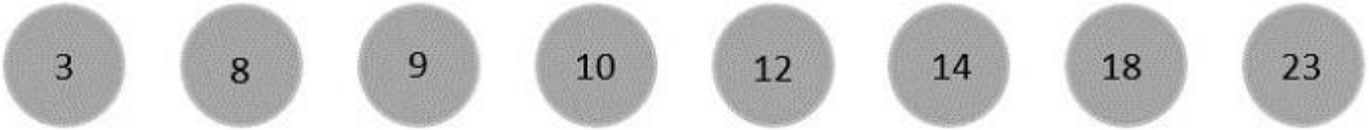


Then, 8 will be popped leaving 3 in the tree.



Heap Sort Algorithm (Working)

After completing the heap sort operation on the given heap, the sorted elements are displayed as shown below −



Every time an element is popped, it is added at the beginning of the output array since the heap data structure formed is a max-heap. But if the heapify method converts the binary tree to the min-heap, add the popped elements are on the end of the output array.

The Final sort list is:

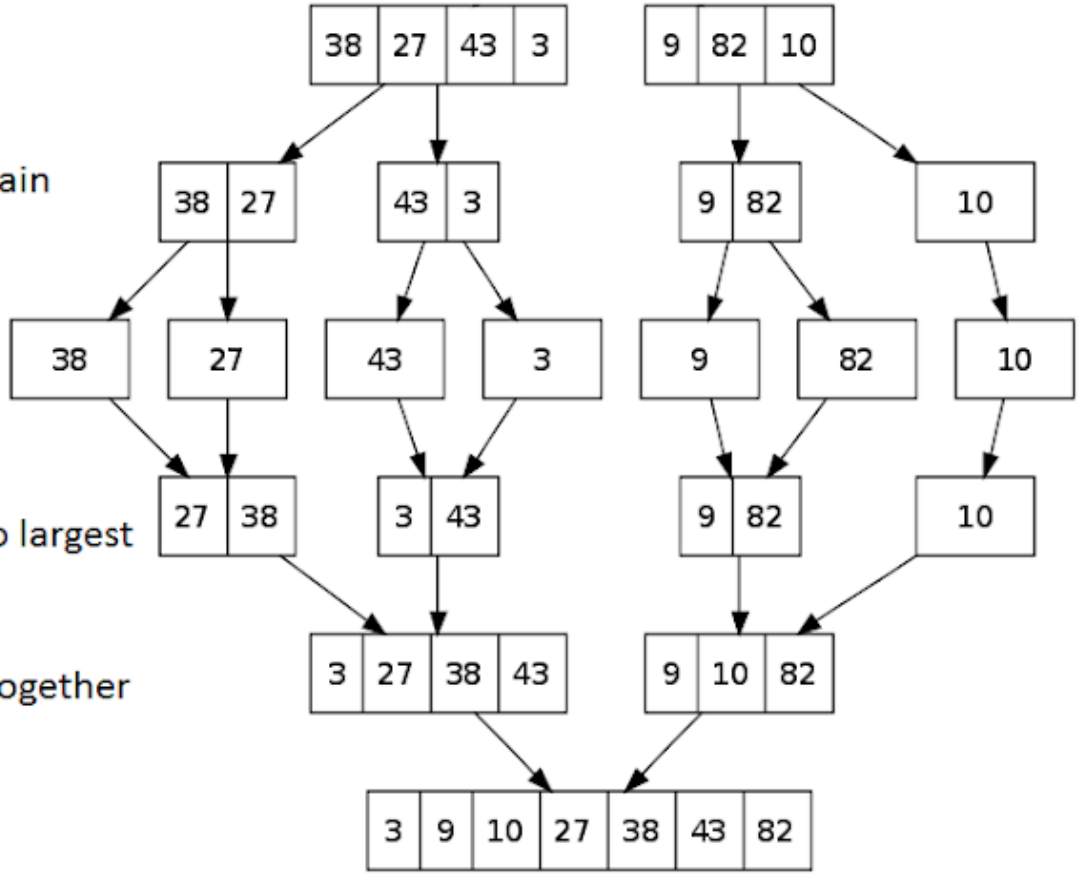
3	8	9	10	12	14	18	23

Merge Sort Algorithm

- Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the divide-and-conquer approach to sort a given array of elements.
- Here's a step-by-step explanation of how merge sort works:
- Divide: Divide the list or array recursively into two halves until it can no more be divided.
- Conquer: Each subarray is sorted individually using the merge sort algorithm.
- Merge: The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Merge Sort Algorithm

1. Divide the array into two parts
2. Divide the array into two parts again
3. Break each element into single parts
4. Sort the elements from smallest to largest
5. Merge the divided sorted arrays together
6. The array has been sorted



Merge Sort Algorithm (Working)

Let the elements of array are

12	31	25	8	32	17	40	42

According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.

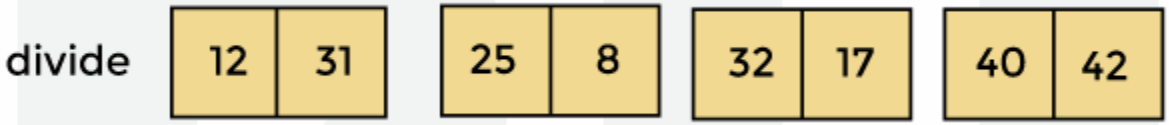
divide

12	31	25	8

32	17	40	42

Merge Sort Algorithm (Working)

Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.



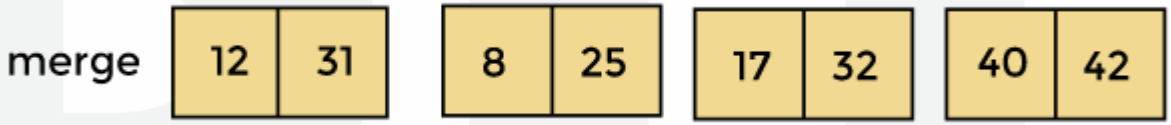
Now, again divide these arrays to get the atomic value that cannot be further divided.

Now, combine them in the same manner they were broken.

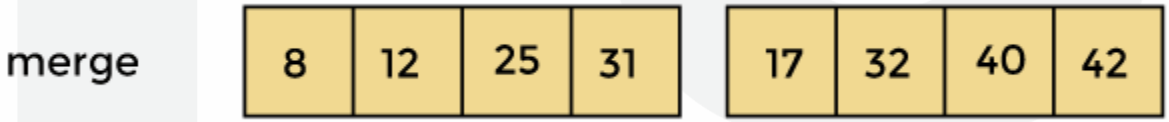
In combining, first compare the element of each array and then combine them into another array in sorted order.

Merge Sort Algorithm (Working)

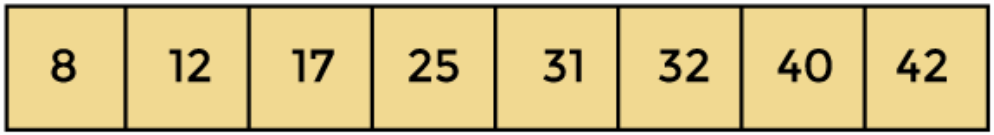
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like



Radix Sort Algorithm

- The Radix Sort algorithm sorts an array by individual digits, starting with the least significant digit (the rightmost one).
- The radix (or base) is the number of unique digits in a number system. In the decimal system we normally use, there are 10 different digits from 0 till 9.
- Radix Sort uses the radix so that decimal values are put into 10 different buckets (or containers) corresponding to the digit that is in focus, then put back into the array before moving on to the next digit.
- Radix Sort is a non relative algorithm that only works with non negative integers.

Radix Sort Algorithm

- The Radix Sort algorithm sorts an array by individual digits, starting with the least significant digit (the rightmost one).
- The radix (or base) is the number of unique digits in a number system. In the decimal system we normally use, there are 10 different digits from 0 till 9.
- Radix Sort uses the radix so that decimal values are put into 10 different buckets (or containers) corresponding to the digit that is in focus, then put back into the array before moving on to the next digit.
- Radix Sort is a non relative algorithm that only works with non negative integers.

Radix Sort Algorithm (Working)

Step 1: Now start with an unsorted array, and an empty array to fit values with corresponding radices 0 till 9.

```
myArray = [ 33, 45, 40, 25, 17, 24]
radixArray = [ [], [], [], [], [], [], [], [], [], [] ]
```

Step 2: Now start sorting by focusing on the least significant digit.

```
myArray = [ 33, 45, 40, 25, 17, 24]
radixArray = [ [], [], [], [], [], [], [], [], [], [] ]
```

Radix Sort Algorithm (Working)

Step 3: Now we move the elements into the correct positions in the radix array according to the digit in focus. Elements are taken from the start of myArray and pushed into the correct position in the radixArray.

```
myArray = [ ]
radixArray = [ [40], [], [], [33], [24], [45, 25], [], [17], [], [] ]
```

Step 4: Now move the elements back into the initial array, and the sorting is now done for the least significant digit. Elements are taken from the end radixArray, and put into the start of myArray.

```
myArray = [ 40, 33, 24, 45, 25, 17 ]
radixArray = [ [], [], [], [], [], [], [], [], [], [] ]
```

Radix Sort Algorithm (Working)

Step 5: Now move focus to the next digit. Notice that values 45 and 25 are still in the same order relative to each other as they were to start with, because we sort in a stable way.

```
myArray = [ 40, 33, 24, 45, 25, 17 ]
radixArray = [ [], [], [], [], [], [], [], [], [], [] ]
```

Step 6: Now move elements into the radix array according to the focused digit.

```
myArray = [ ]
radixArray = [ [], [17], [24, 25], [33], [40, 45], [], [], [], [], [] ]
```

Radix Sort Algorithm (Working)

Step 5: Now move focus to the next digit. Notice that values 45 and 25 are still in the same order relative to each other as they were to start with, because we sort in a stable way.

```
myArray = [ 40, 33, 24, 45, 25, 17 ]
radixArray = [ [], [], [], [], [], [], [], [], [], [] ]
```

Step 6: Now move elements into the radix array according to the focused digit.

```
myArray = [ ]
radixArray = [ [], [17], [24, 25], [33], [40, 45], [], [], [], [], [] ]
```

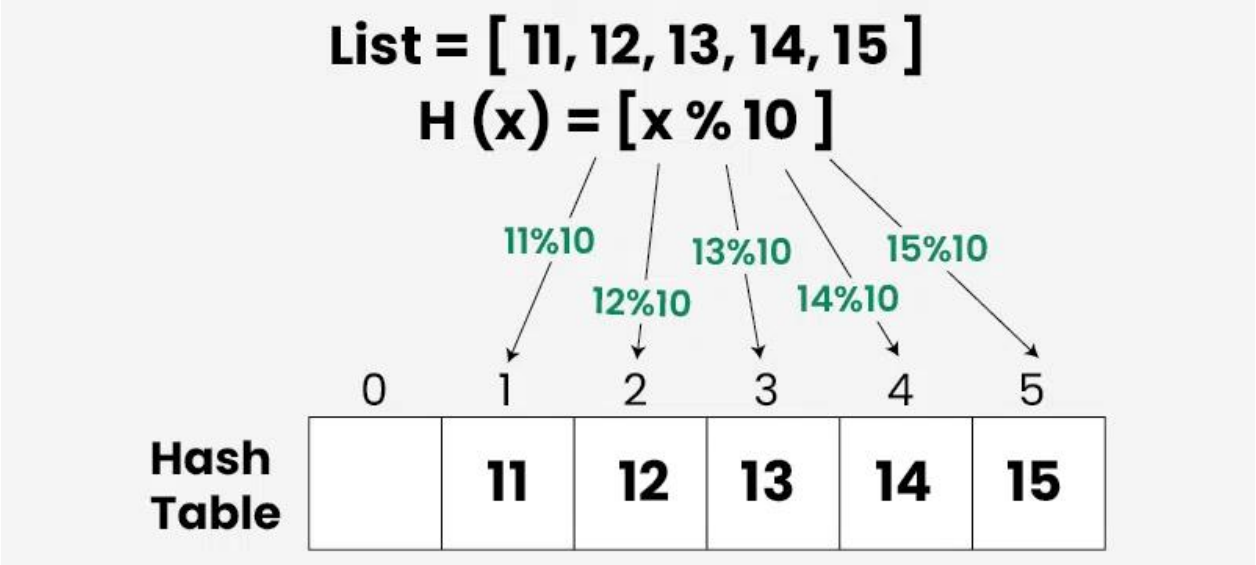
Step 7: Now move elements back into the start of myArray, from the back of radixArray.

```
myArray = [ 17, 24, 25, 33, 40, 45 ]
radixArray = [ [], [], [], [], [], [], [], [], [], [] ]
```

Hashing and Hashing Function

- Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access.
- Hashing is used to efficiently organize and retrieve data. It's a common method for implementing hash tables, which store key/value pairs in a list that's accessible by index.
- **Hashing is used in:** Databases, Compilers, Caching mechanisms, Cybersecurity, Cryptography
- A hash function takes an input, like a key or identifier, and computes a hash code that serves as a unique index or address in a hash table.

Hashing and Hashing Function



- Hash Table: It is a type of data structure that stores data in an array format. The table maps keys to values using a hash function.

Hashing and Hashing Function

- **Hash Table:** It is a type of data structure that stores data in an array format. The table maps keys to values using a hash function.
- **Hash Function:** It performs the mathematical operation of accepting the key value as input and producing the hash code or hash value as the output. Some of the characteristics of an ideal hash function are as follows:
  - It must produce the same hash value for the same hash key to be deterministic.
  - Every input has a unique hash code. This feature is known as the hash property.
  - It must be collision-friendly.
  - A little bit of change leads to a drastic change in the output.
  - The calculation must be quick

Collision Resolution Techniques

- The hash function is used to find the index of the array.
- The hash value is used to create an index for the key in the hash table.
- The hash function may return the same hash value for two or more keys. When two or more keys have the same hash value, a collision happens. To handle this collision, we use collision resolution techniques.
- There are two types of collision resolution techniques.
- **Separate chaining:** This method involves making a linked list out of the slot where the collision happened, then adding the new key to the list. Separate chaining is the term used to describe how this connected list of slots resembles a chain. It is more frequently utilized when we are unsure of the number of keys to add or remove.

Collision Resolution Techniques

- **Open addressing:** To prevent collisions in the hashing table, open addressing is employed as a collision-resolution technique. No key is kept anywhere else besides the hash table. As a result, the hash table's size is never equal to or less than the number of keys. Additionally known as closed hashing.
- The following techniques are used in open addressing:
  1. Linear probing
  2. Quadratic probing
  3. Double hashing

Collision Resolution Techniques

	Separate Chaining	Open Addressing
1.	Keys are stored inside the hash table as well as outside the hash table.	All the keys are stored only inside the hash table. No key is outside the hash table.
2.	The number of keys to be stored in the hash table can even exceed the size of the hash table.	The number of keys to be stored in the hash table can never exceed the size of the hash table.
3.	Deletion is easier.	Deletion is difficult.
4.	Extra space is required for the pointers to store the keys outside the hash table.	No extra space is required.
5.	Cache performance is poor. This is because of linked lists which store the keys outside the hash table.	Cache performance is better. This is because here no linked lists are used.

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in