Quick Sort
+
Heap Sort

Partition the Array: Problem 01

You are given an Array, and you have to partition the array such that all the elements smaller than the first element should be on the left side of the element and all the elements greater than or equal to the first element should be on the right side of the element.

Note: You can not use any additional data structure like a hash table, arrays, etc.

Partition the Array: Test Cases

Input:

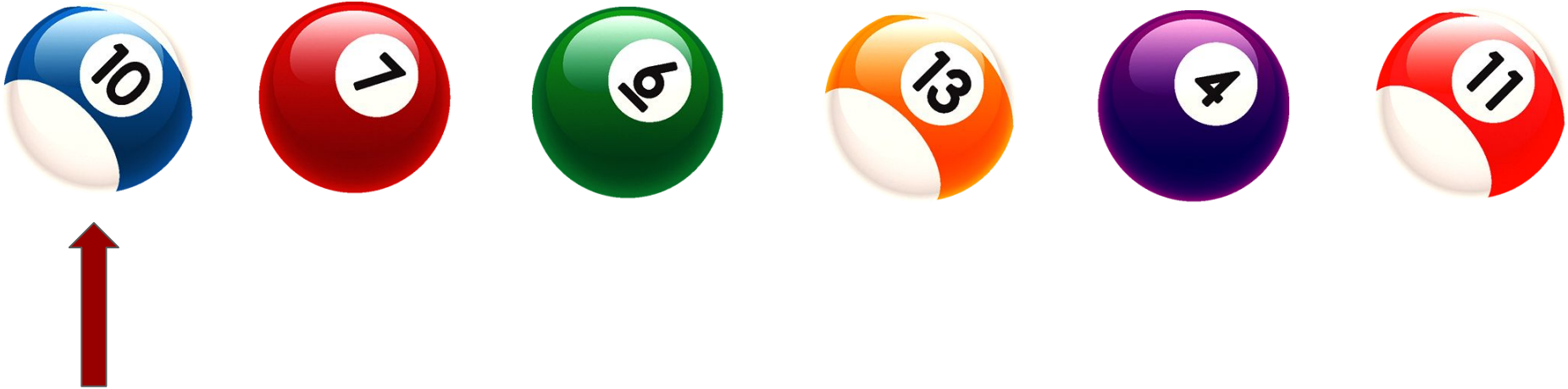
Output:

Partition the Array: Problem 01

How will you do that without using any extra memory?

Partition the Array: Solution

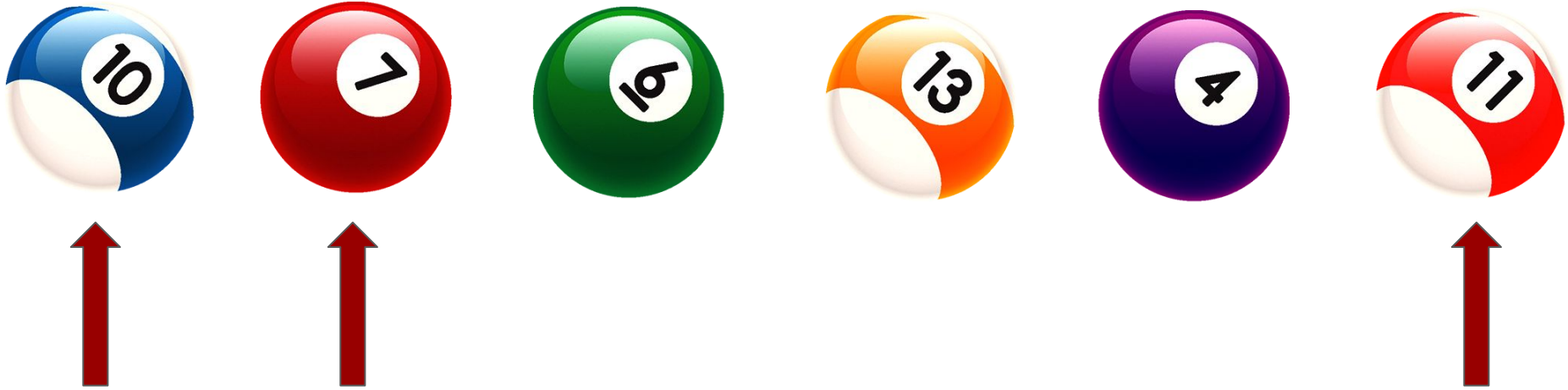
Let's call the element as pivot around which we want to partition



Pivot

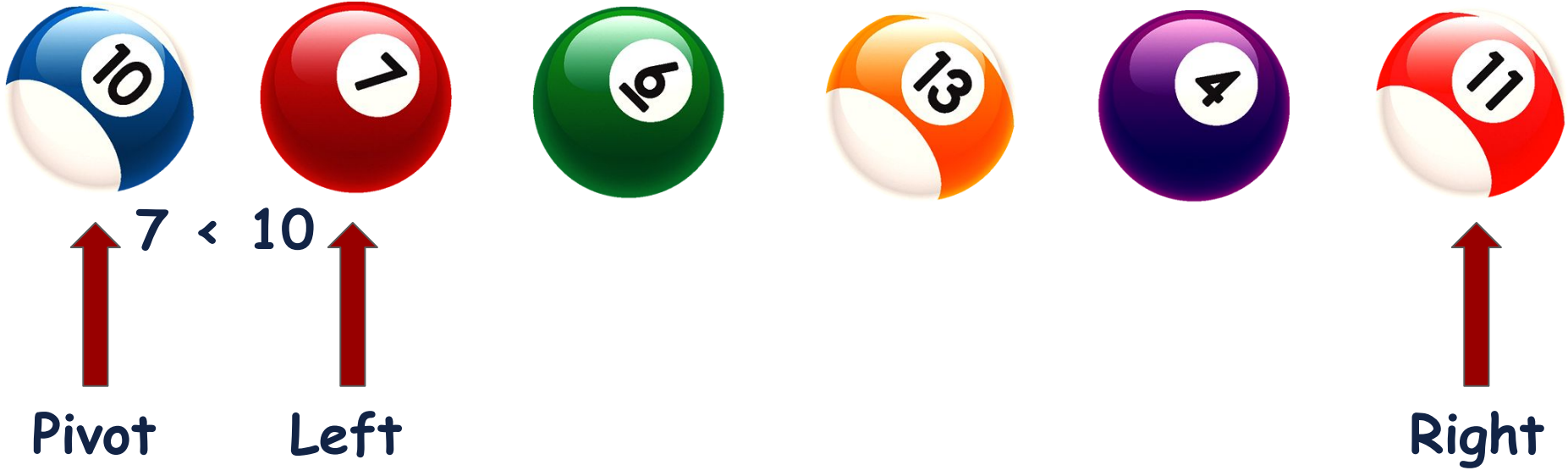
Partition the Array: Solution

Let's make 2 pointers, 1 pointing to the start of the array after pivot and one to the end.



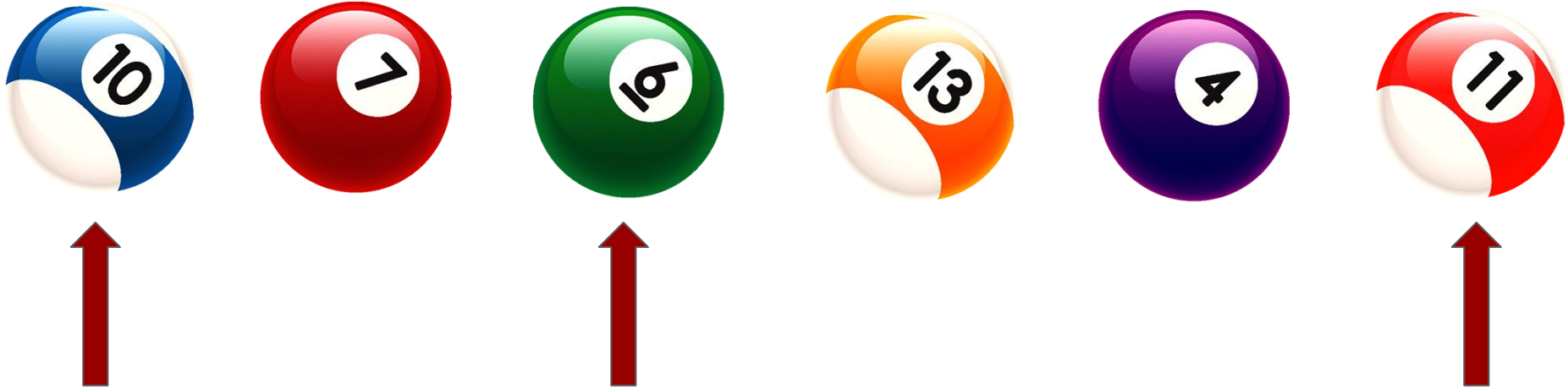
Partition the Array: Solution

Now, we will keep incrementing the left pointer if the element is less than the pivot element.



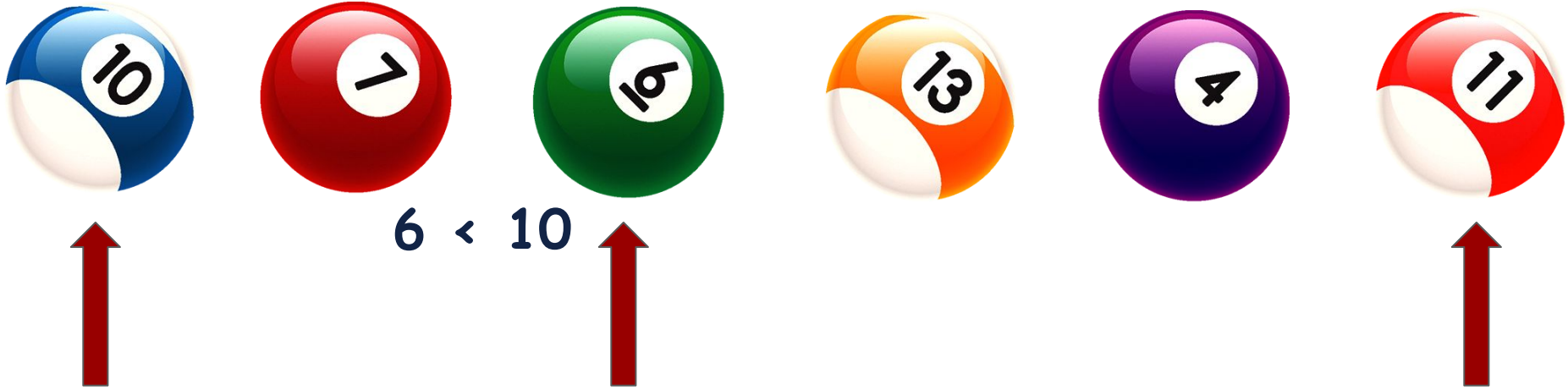
Partition the Array: Solution

Now, we will keep incrementing the left pointer if the element is less than the pivot element.



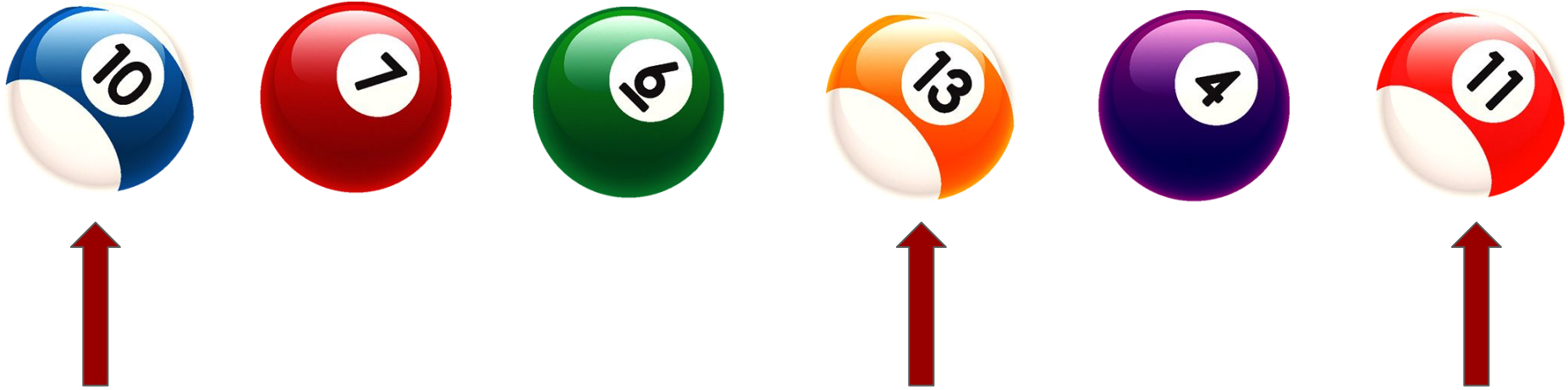
Partition the Array: Solution

Now, we will keep incrementing the left pointer if the element is less than the pivot element.



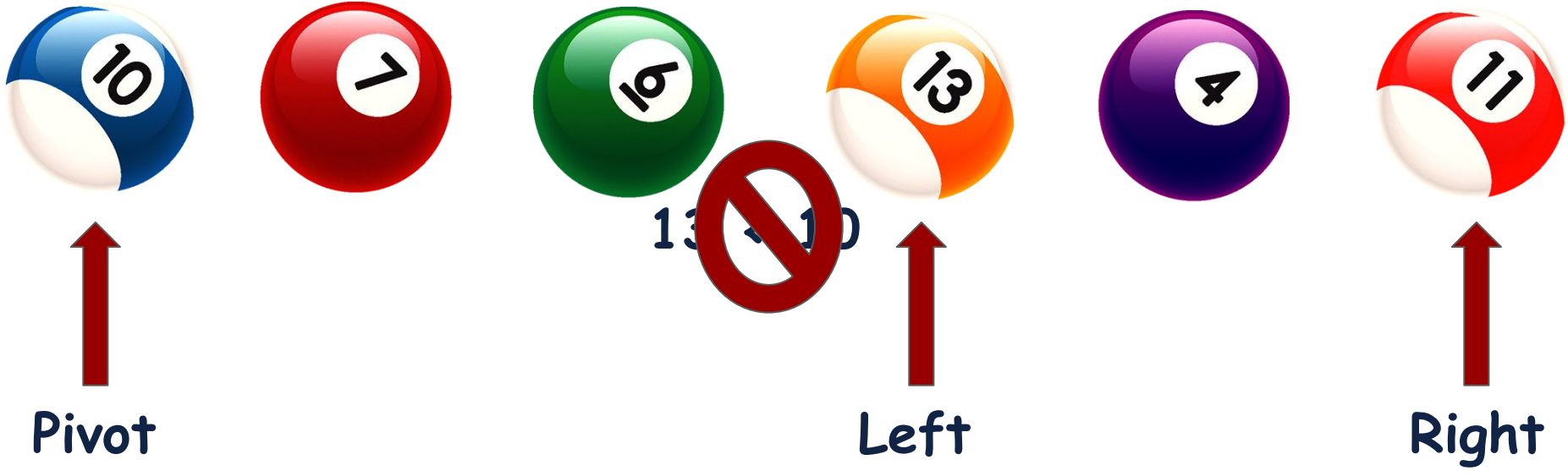
Partition the Array: Solution

Now, we will keep incrementing the left pointer if the element is less than the pivot element.



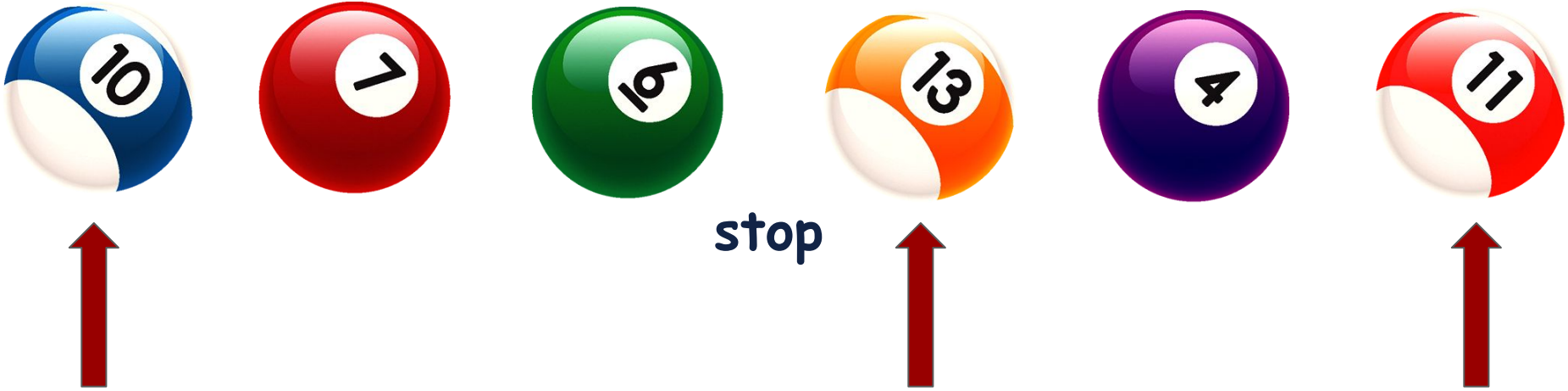
Partition the Array: Solution

Now, we will keep incrementing the left pointer if the element is less than the pivot element.



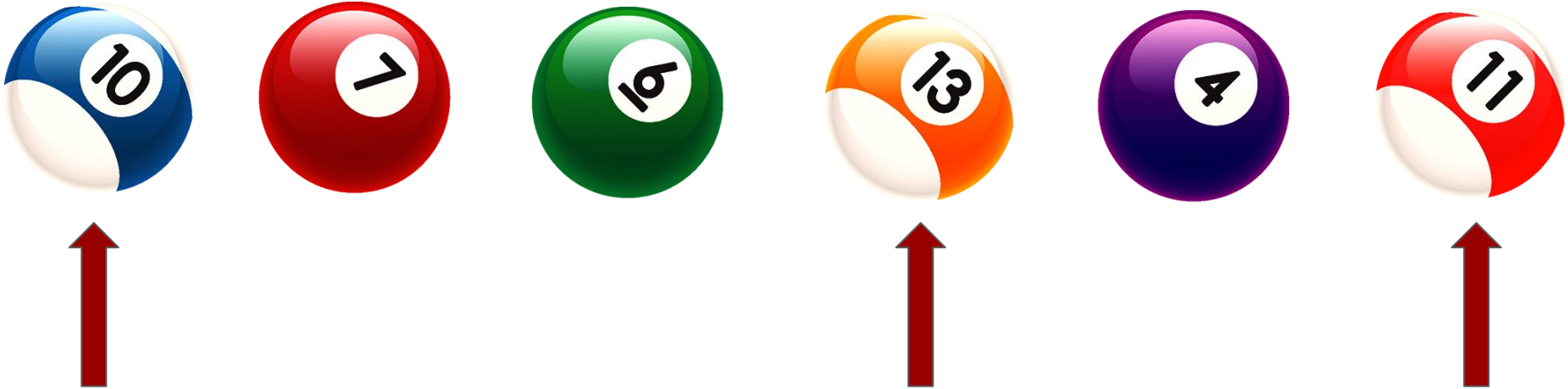
Partition the Array: Solution

Now, we will keep incrementing the left pointer if the element is less than the pivot element.



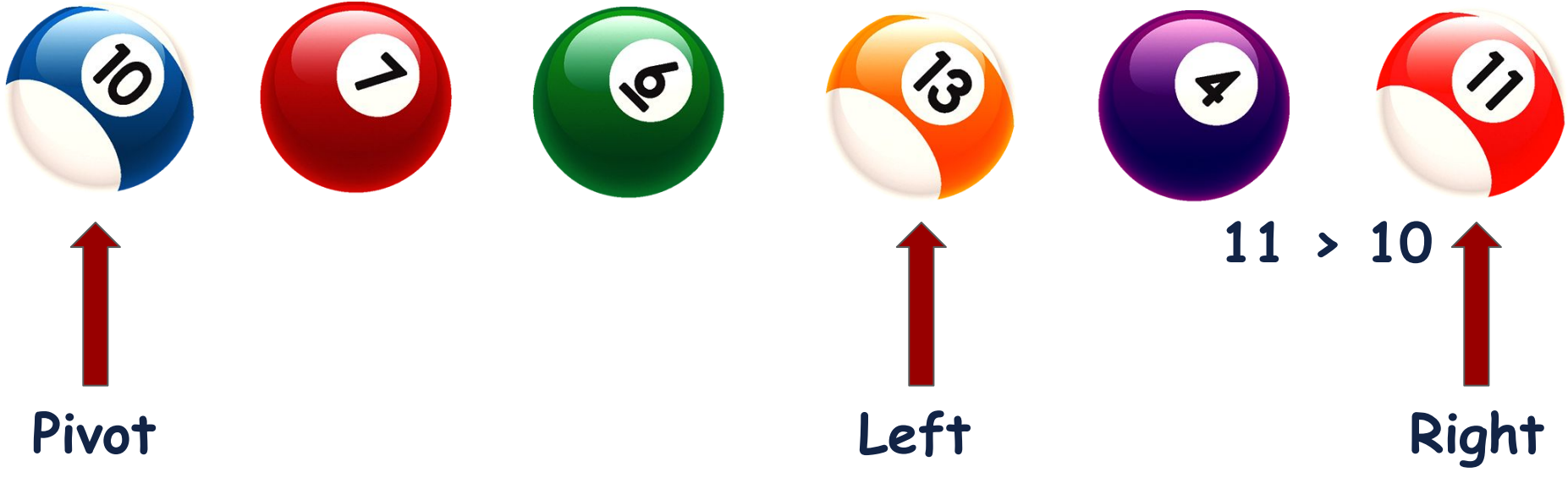
Partition the Array: Solution

We will also keep decrementing the right pointer if the element is greater than or equal to the pivot element.



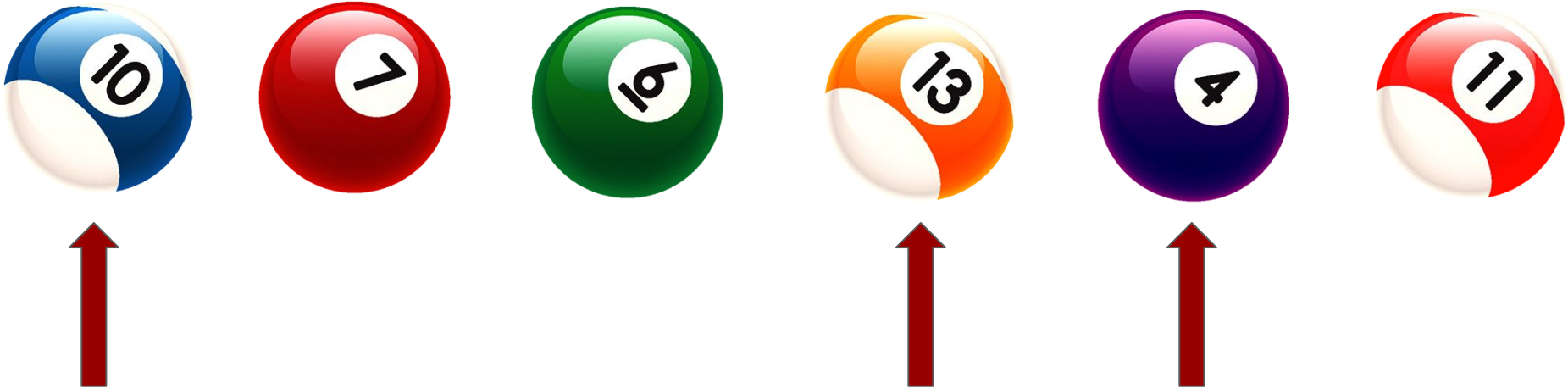
Partition the Array: Solution

We will also keep decrementing the right pointer if the element is greater than or equal to the pivot element.



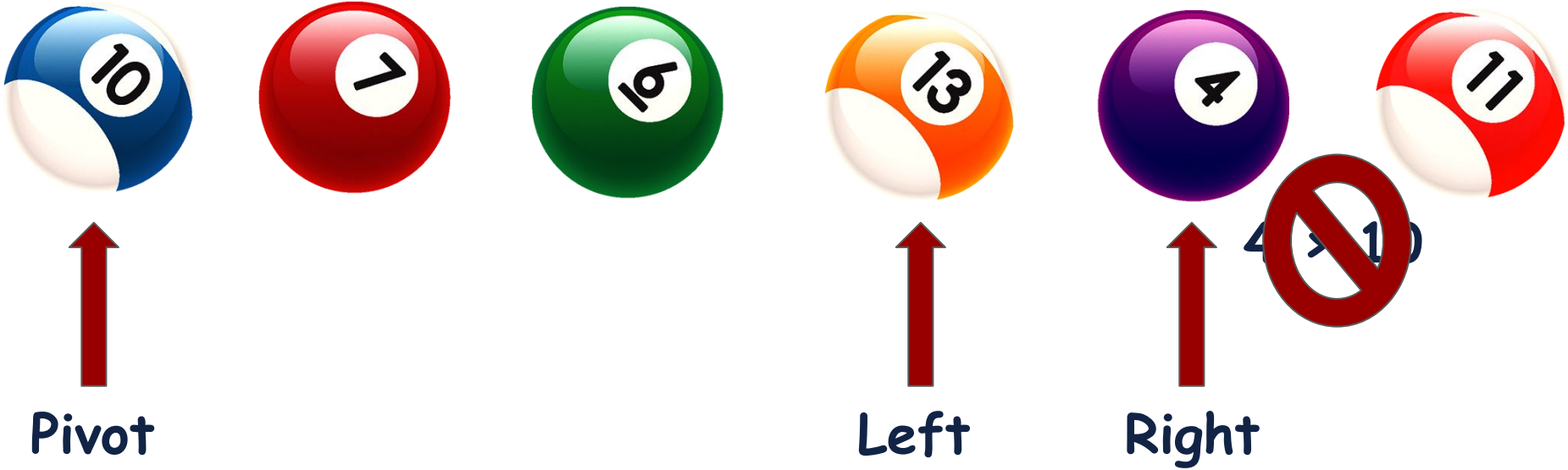
Partition the Array: Solution

We will also keep decrementing the right pointer if the element is greater than or equal to the pivot element.



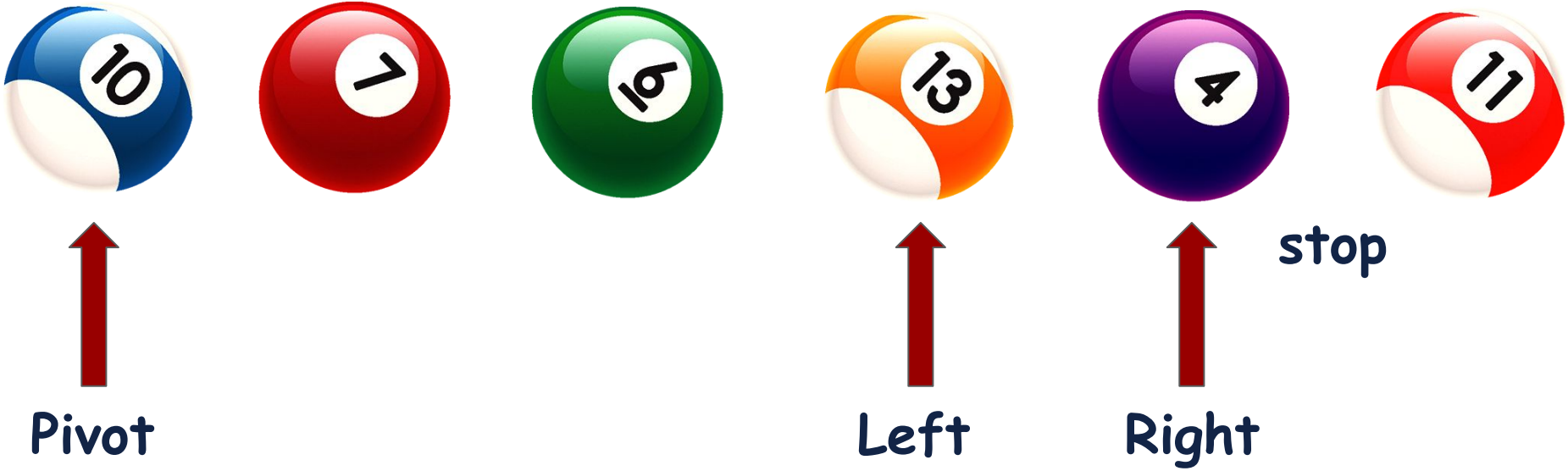
Partition the Array: Solution

We will also keep decrementing the right pointer if the element is greater than or equal to the pivot element.



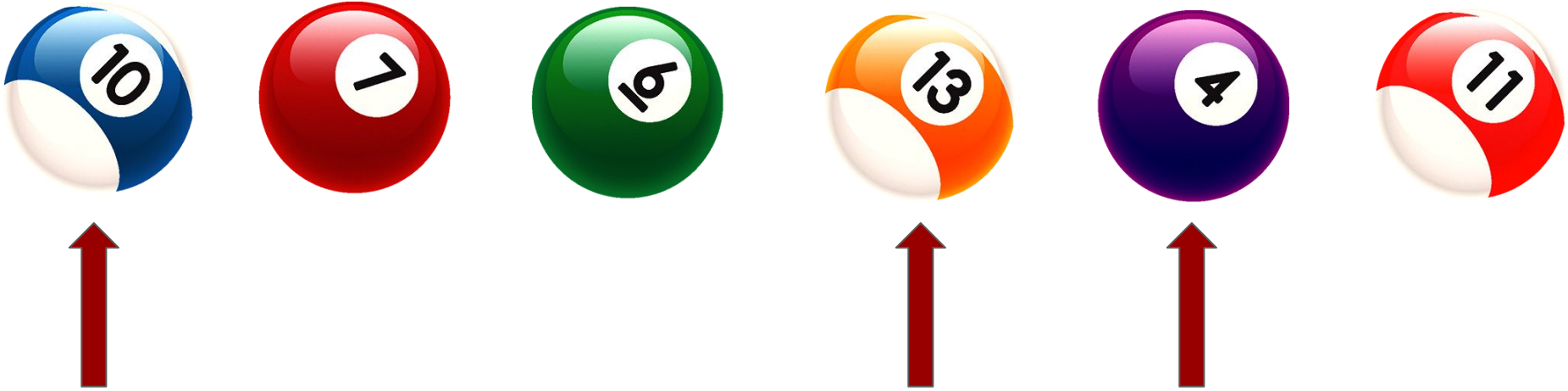
Partition the Array: Solution

We will also keep decrementing the right pointer if the element is greater than or equal to the pivot element.



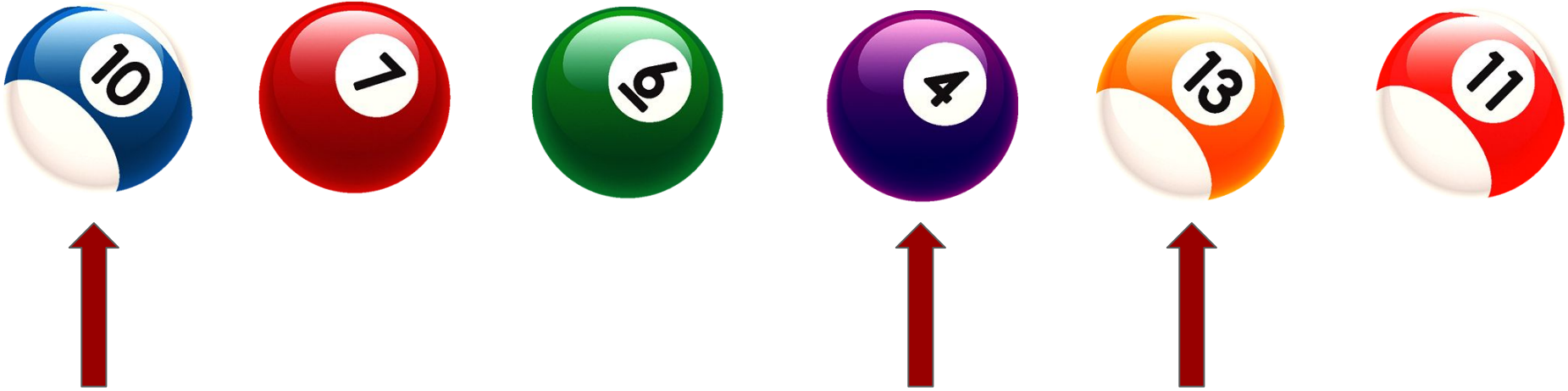
Partition the Array: Solution

Now, we will swap the elements of left and right pointer.



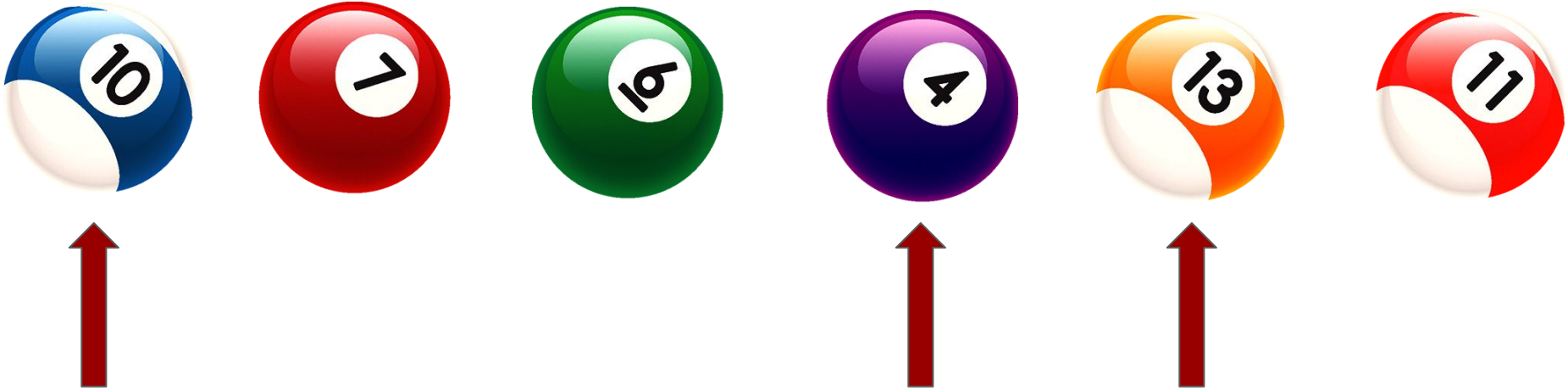
Partition the Array: Solution

Now, we will swap the elements of left and right pointer.



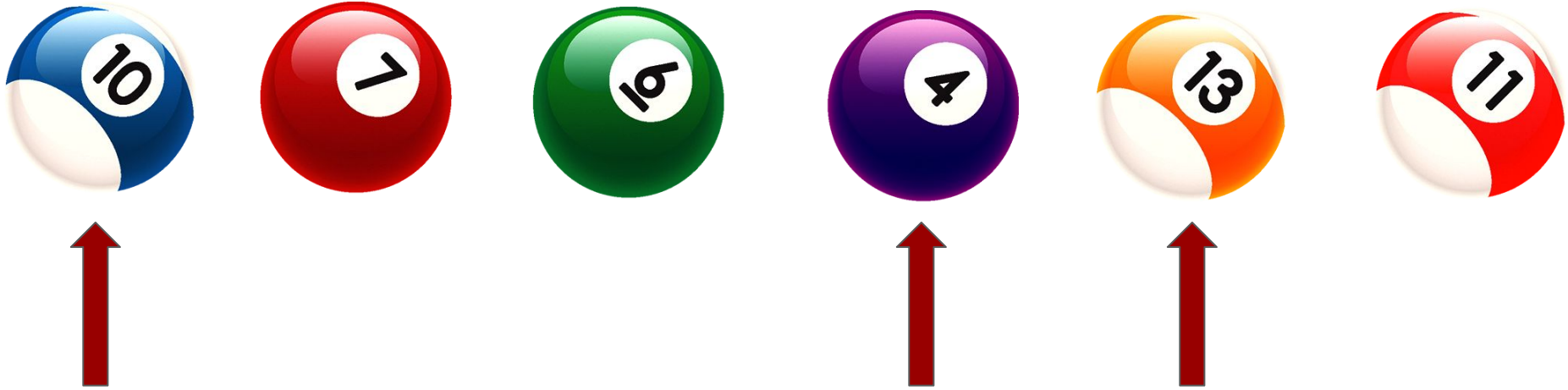
Partition the Array: Solution

We will keep updating left and right pointers until the left pointer is less than or equal to the right pointer.



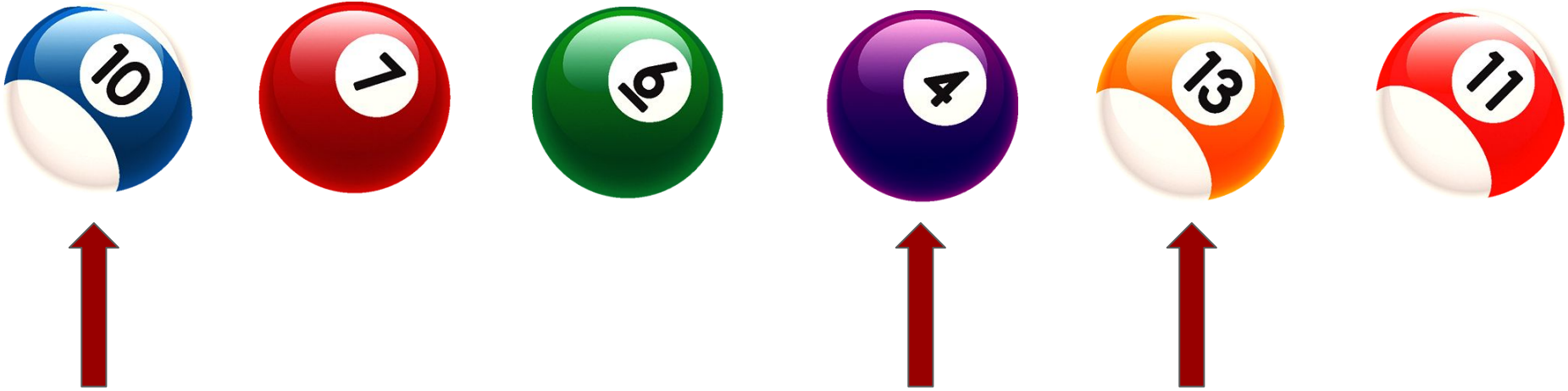
Partition the Array: Solution

We will keep updating left and right pointers until the left pointer is less than or equal to the right pointer.



Partition the Array: Solution

Now, we will stop and swap the right element with the pivot



Partition the Array: Solution

Now, we will stop and swap the right element with the pivot

Partition the Array: Solution

All the elements less than the pivot are on left side and all the elements greater than pivot are on right side.

Partition the Array: Solution

Let's code the solution.

Partition the Array: Solution

Let's code the solution.

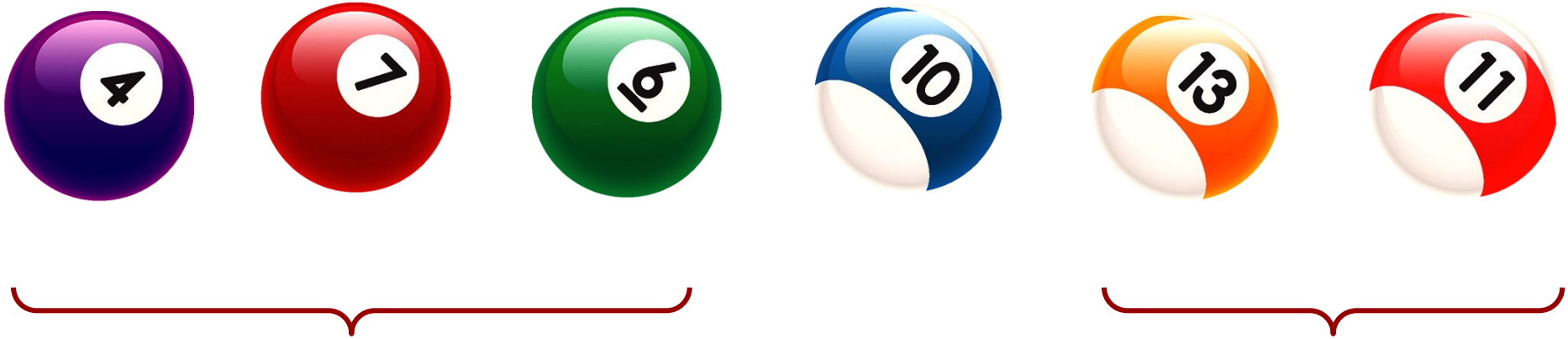
```
void partition(int arr[], int start, int end, int pivot)
{
    int left = start;
    int right = end;
    while (left <= right)
    {
        while (arr[left] < arr[pivot] && left <= end)
            left++;
        while (arr[right] >= arr[pivot] && right >= start)
            right--;
        if (left < right)
            swap(arr[left], arr[right]);
    }
    swap(arr[right], arr[pivot]);
}
```

Partition the Array: 1st Pass

After the first pass, values smaller than 10 are on the left side and greater than 10 are on the right side.

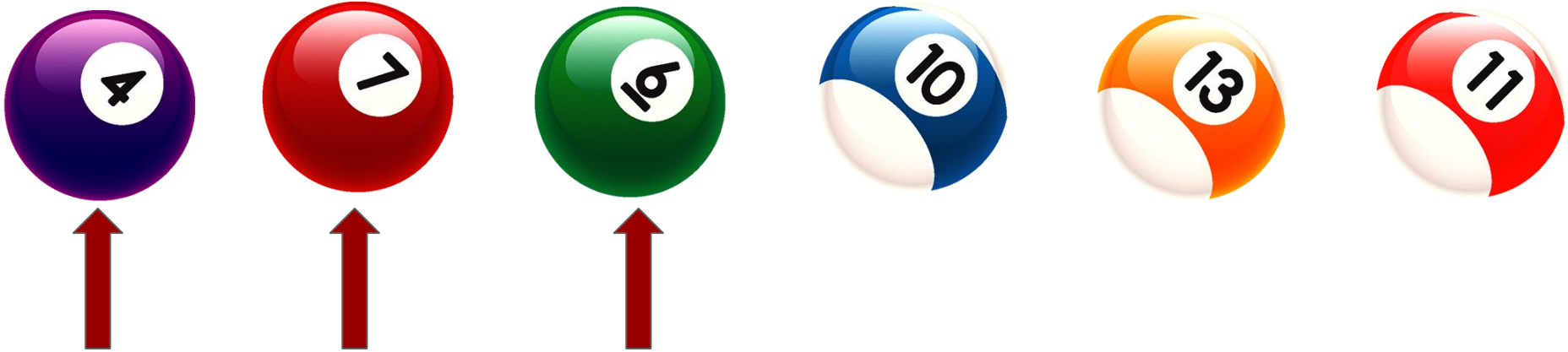
Partition the Array: 1st Pass

Now, if we again apply the same partition procedure on the left and right elements separately, and keep on applying, then the data will be sorted.



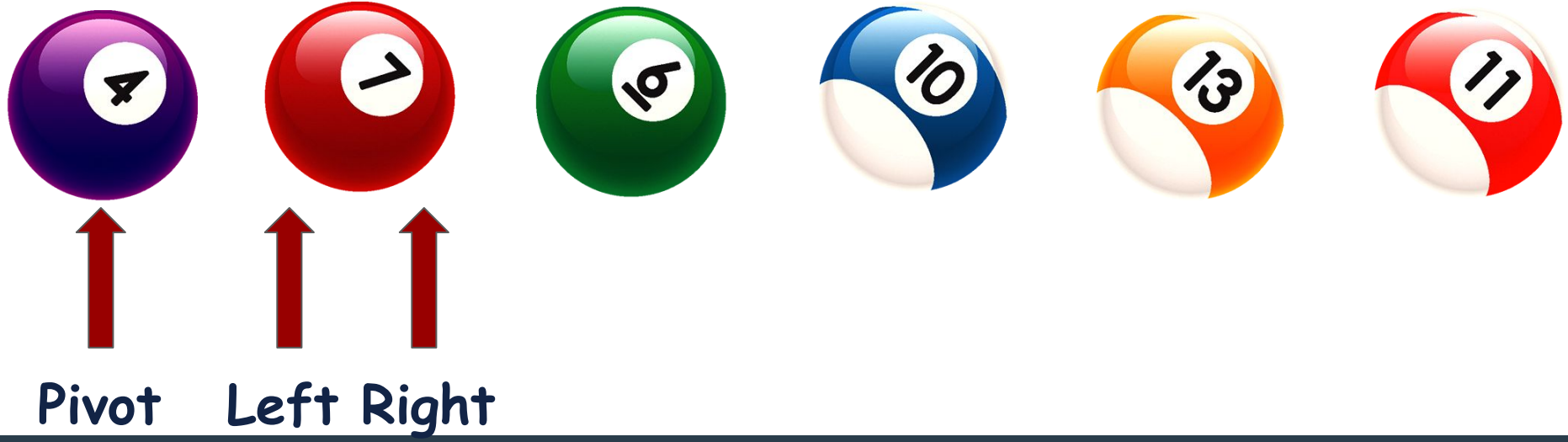
Partition the Array

Now, if we again apply the same partition procedure on the left and right elements separately, and keep on applying, then the data will be sorted.



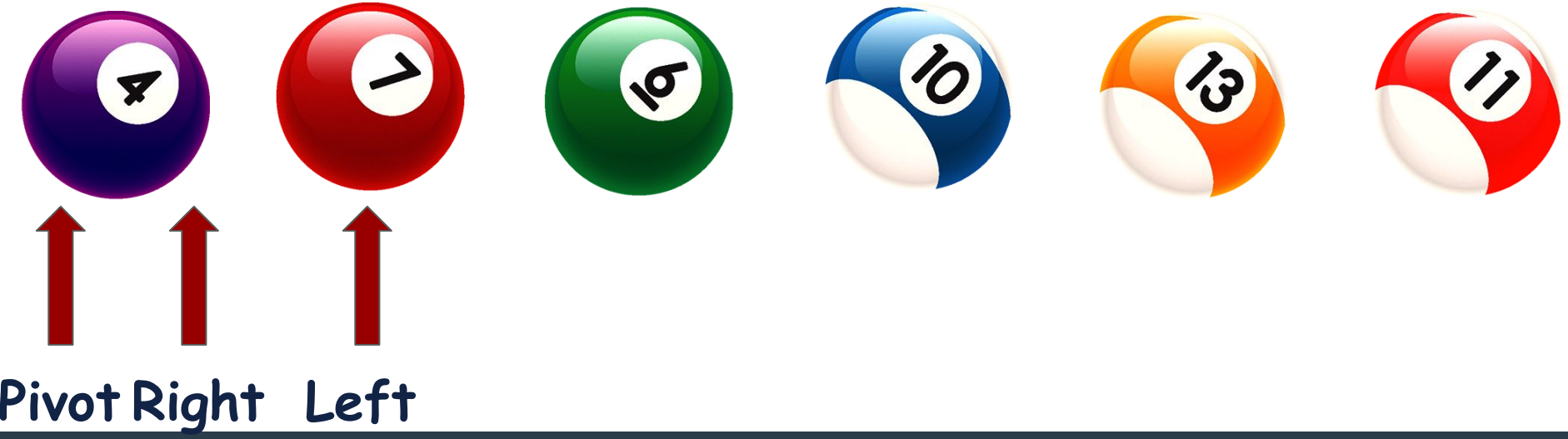
Partition the Array

Now, if we again apply the same partition procedure on the left and right elements separately, and keep on applying, then the data will be sorted.



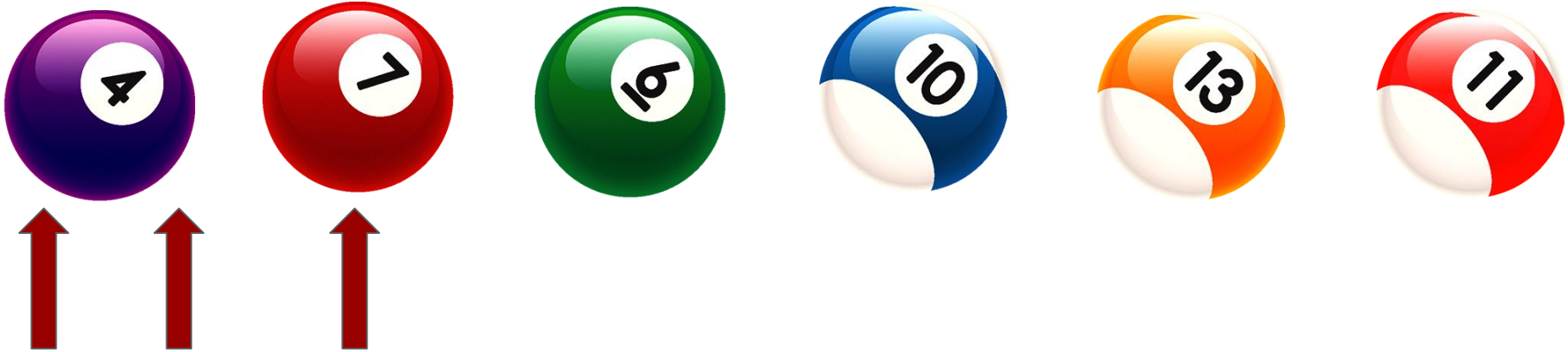
Partition the Array

Now, if we again apply the same partition procedure on the left and right elements separately, and keep on applying, then the data will be sorted.



Partition the Array

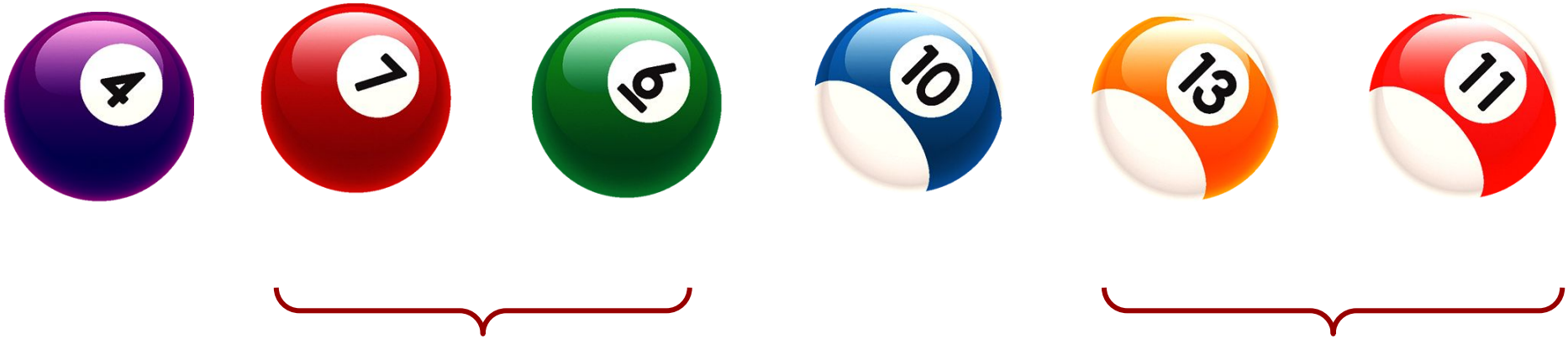
Now, if we again apply the same partition procedure on the left and right elements separately, and keep on applying, then the data will be sorted.



4 will be swapped with 4

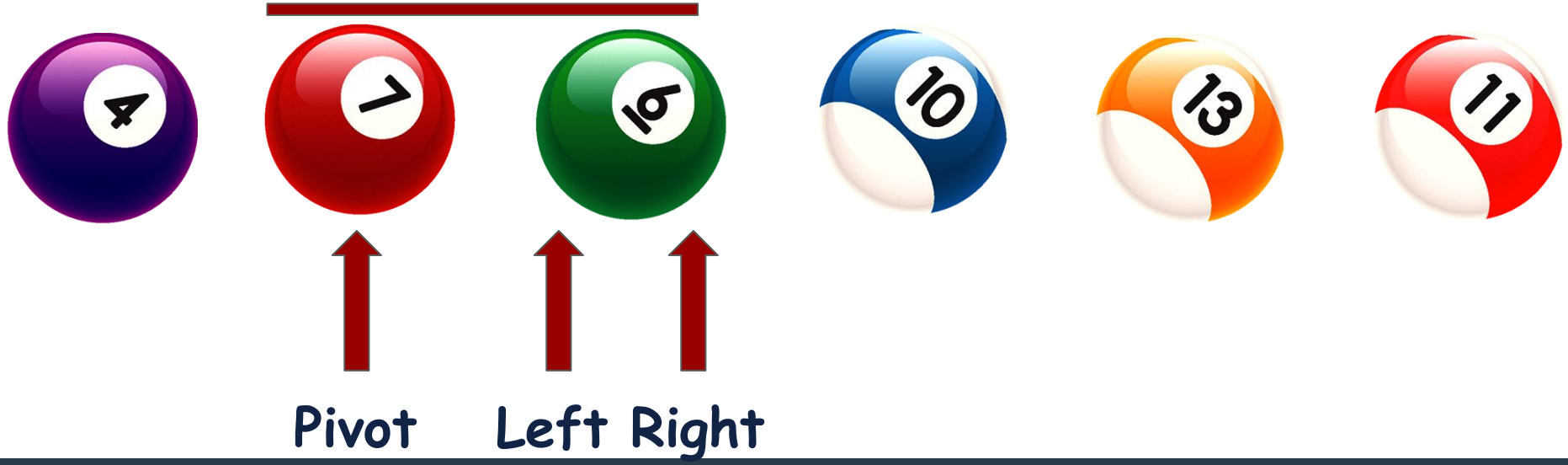
Partition the Array

Now, the following partitions remained to be further partitioned.



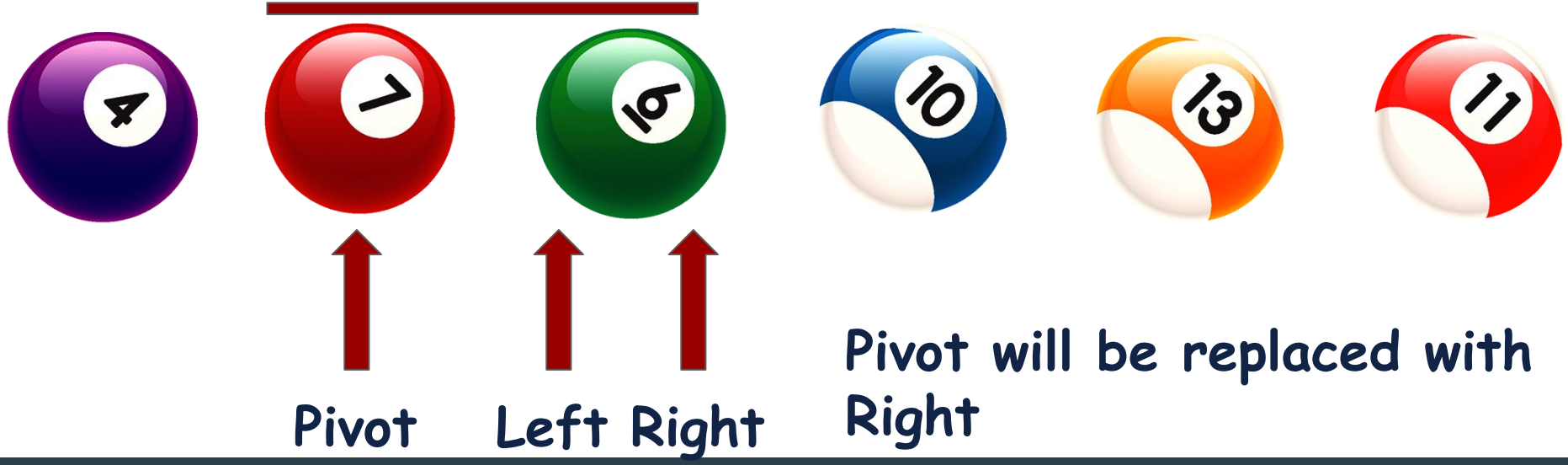
Partition the Array

If we again apply the same partition procedure on the left and right elements separately, and keep on applying, then the data will be sorted.



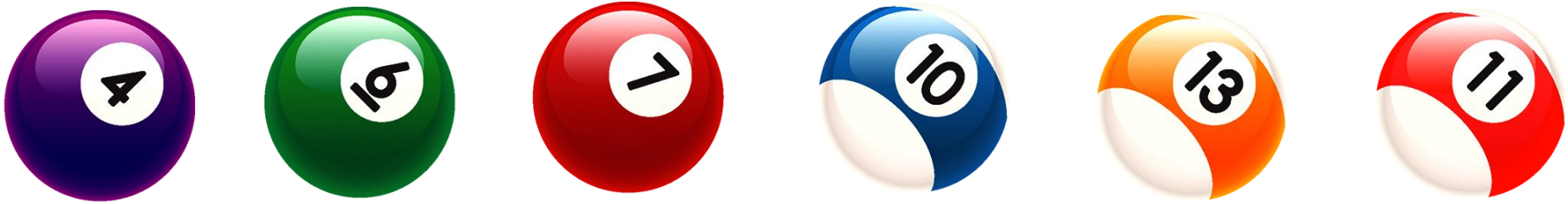
Partition the Array

If we again apply the same partition procedure on the left and right elements separately, and keep on applying, then the data will be sorted.



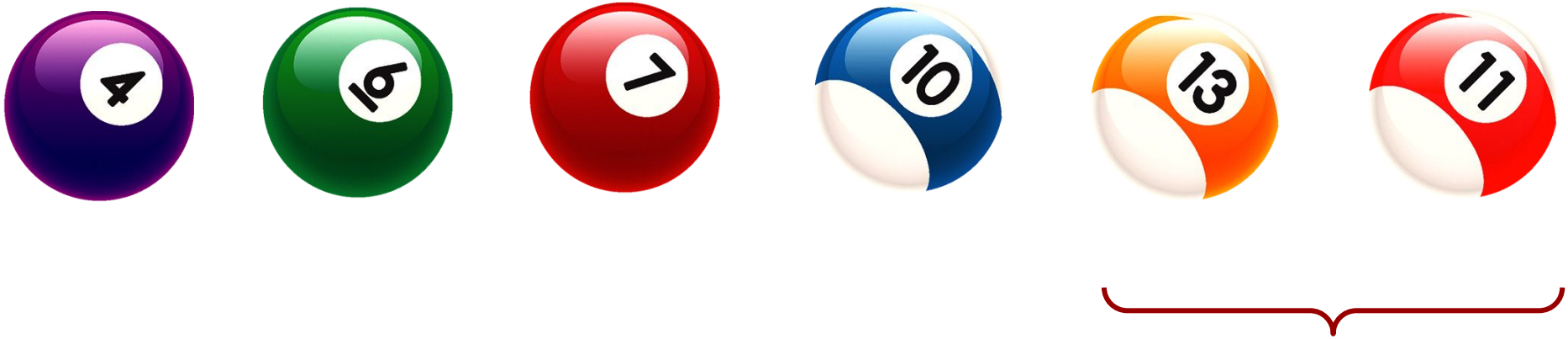
Partition the Array

If we again apply the same partition procedure on the left and right elements separately, and keep on applying, then the data will be sorted.



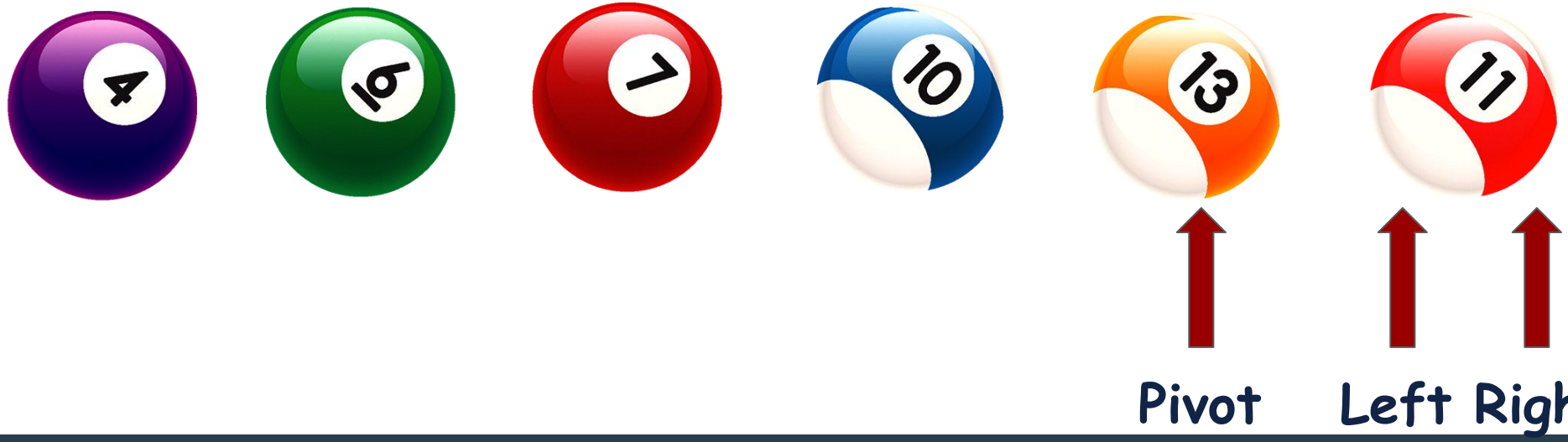
Partition the Array

Now, the following partitions remained to be further partitioned.



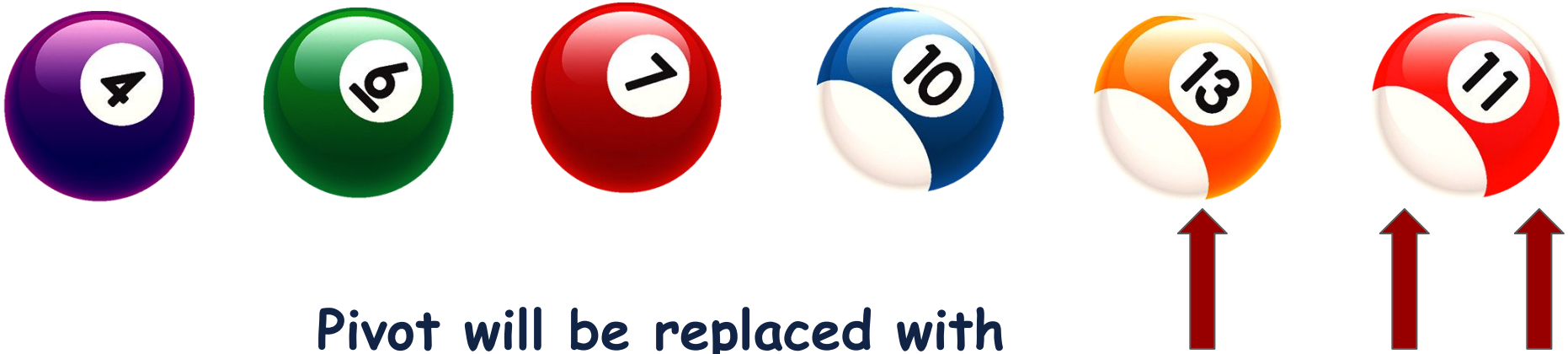
Partition the Array

If we again apply the same partition procedure on the left and right elements separately, and keep on applying, then the data will be sorted.



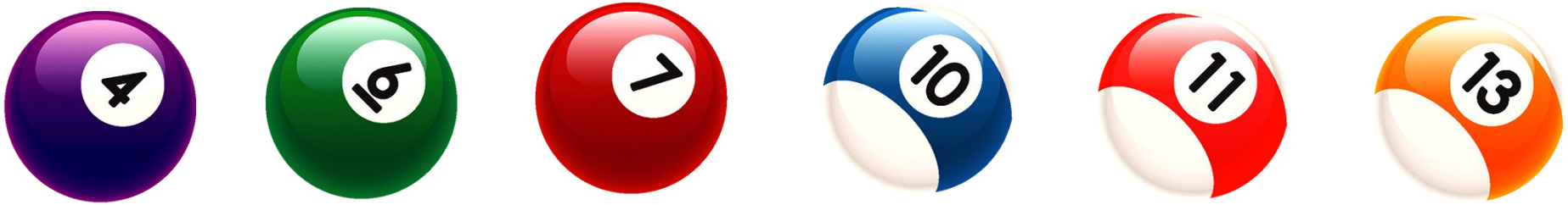
Partition the Array

If we again apply the same partition procedure on the left and right elements separately, and keep on applying, then the data will be sorted.



Partition the Array

If we again apply the same partition procedure on the left and right elements separately, and keep on applying, then the data will be sorted.



Quick Sort

This Algorithm of sorting is called as **Quick Sort**. QuickSort is a **Divide and Conquer** algorithm. It picks an element as a pivot and partitions the given array around the picked pivot.

Quick Sort: Implementation

Let's code the solution. Use the previous partition function, just return the index around which the elements are partitioned and then call the same function recursively.

Quick Sort: Implementation

```
main()
{
    int arr[6] = {10, 7, 6, 13, 4, 11};
    quickSort(arr, 0, 5);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
void quickSort(int arr[], int start, int end)
{
    if (start < end)
    {
        int pivot = start;
        int mid = partition(arr, start + 1, end, pivot);
        cout << endl;
        quickSort(arr, start, mid - 1);
        quickSort(arr, mid + 1, end);
    }
}
```

Quick Sort: Implementation

```
int partition(int arr[], int start, int end, int pivot)
{
    int left = start;
    int right = end;
    while (left <= right)
    {
        while (arr[left] < arr[pivot] && left <= end)
            left++;
        while (arr[right] >= arr[pivot] && right >= start)
            right--;
        if (left < right)
            swap(arr[left], arr[right]);
    }
    swap(arr[right], arr[pivot]);
    return right;
}
```

Quick Sort: Time Complexity

What is the time complexity of Quick Sort?

Quick Sort: Time Complexity

What is the time complexity of Quick Sort?
It Depends on the Pivot Element.

Quick Sort: Time Complexity

What is the time complexity of Quick Sort?
It Depends on the Pivot Element.
If the pivot element partitions the array into 2 equal halves every time then the time complexity is same as Merge Sort.

Quick Sort: Time Complexity

What is the time complexity of Quick Sort?
It Depends on the Pivot Element.
If the pivot element partitions the array into 2 equal halves every time then the time complexity is same as Merge Sort.

Average Time Complexity: $O(n * \log_2(n))$

Sorting Algorithms

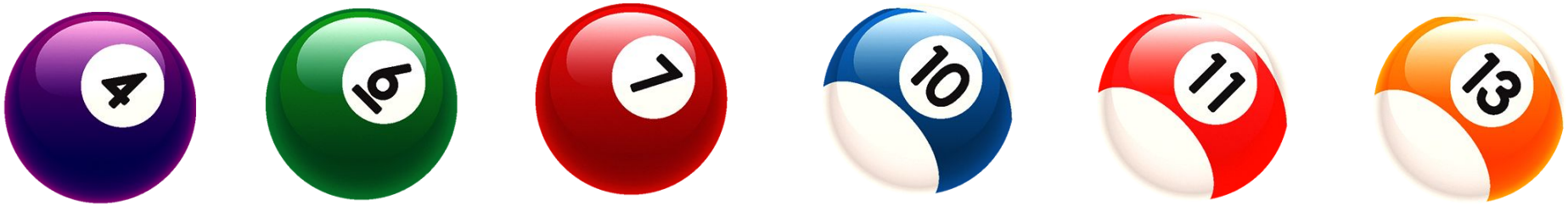
Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Merge Sort	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N)$
Quick Sort	$O(N*\log_2 N)$	$O(N*\log_2 N)$		

Quick Sort: Time Complexity

What is the worst time complexity of Quick Sort?

Quick Sort: Time Complexity

What is the worst time complexity of Quick Sort?
If the pivot element is such that all the elements are greater than or less than the pivot element then there will be only 1 half.
Consider the case of sorted array as input.



Quick Sort: Time Complexity

What is the worst time complexity of Quick Sort?
If the pivot element is such that all the elements are greater than or less than the pivot element then there will be only 1 half.
Consider the case of sorted array as input.

Worst Time Complexity: $O(n^2)$

Sorting Algorithms

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Merge Sort	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N)$
Quick Sort	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N^2)$	

Quick Sort: Time Complexity

Is there anything we can do to avoid the worst time complexity of the Quick Sort?

Quick Sort: Time Complexity

We can choose the pivot randomly to avoid the worst time complexity of the Quick Sort.

Quick Sort: Space Complexity

What is the Space complexity of the Quick Sort?

Quick Sort: Space Complexity

What is the Space complexity of the Quick Sort?
Stack is used for recursive calls.
How much is the depth till which we divide the array?

Sorting Algorithms

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Merge Sort	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N)$
Quick Sort	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N^2)$	$O(N)$

Sorting Algorithms

Sorting Algorithm	In-Place	Stable
Bubble Sort	Yes	Yes
Selection Sort	Yes	No
Insertion Sort	Yes	Yes
Merge Sort	No	Yes
Quick Sort	Yes	No

Heap Sort

Heap Sort

We can also use **Max Heap** to sort the elements.

Heap Sort

We can also use **Max Heap** to sort the elements.

But if we first add all the elements in the max heap (Priority Queue), and then pop the elements from the heap one by one and store it into another array it will take extra time and space.

Heap Sort

So instead of declaring a new Heap, can we convert the input array into max heap?

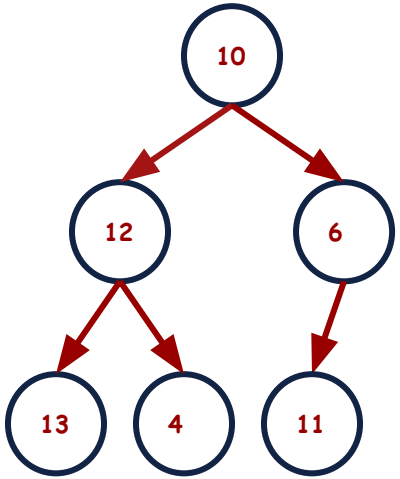
Heap Sort

So instead of declaring a new Heap, can we convert the input array into max heap?
How?

Heap Sort

Suppose the given input elements are:

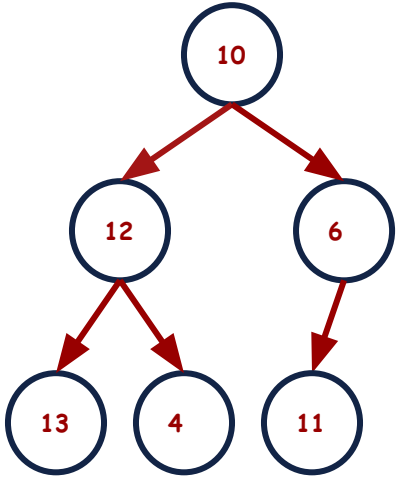
10	12	6	13	4	11
0	1	2	3	4	5



Heap Sort

We will start the heapify method from the non-leaf elements.

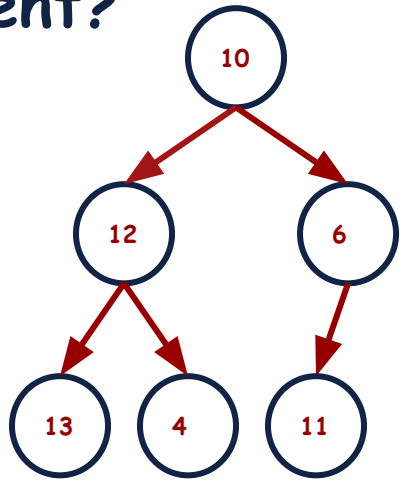
10	12	6	13	4	11
0	1	2	3	4	5



Heap Sort

We will start the heapify method from the non-leaf elements.
What is the index of last non-leaf element?

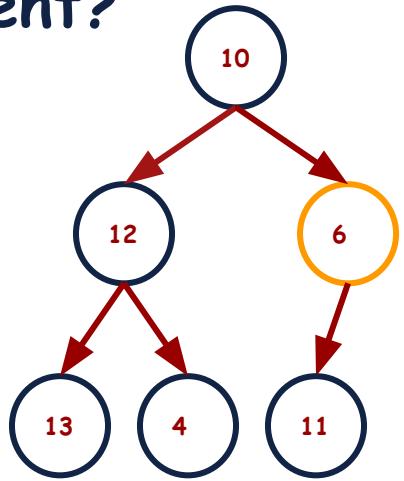
10	12	6	13	4	11
0	1	2	3	4	5



Heap Sort

We will start the heapify method from the non-leaf elements.
What is the index of last non-leaf element?

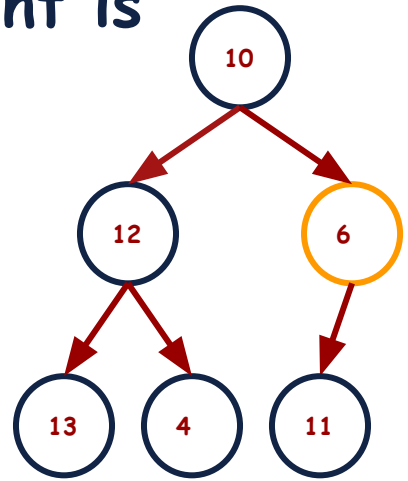
10	12	6	13	4	11
0	1	2	3	4	5



Heap Sort

So, we will start with the ((size / 2) - 1) element, and compare it with its left and right child and swap the largest child with the parent if the parent is less than the child.

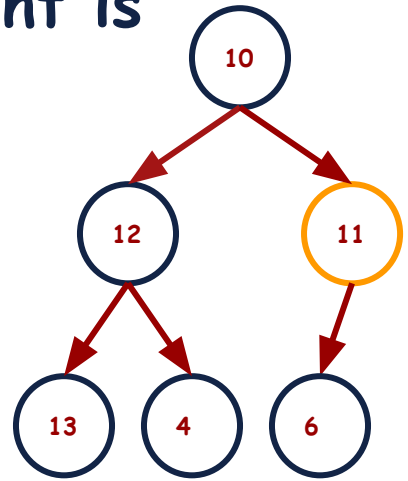
10	12	6	13	4	11
0	1	2	3	4	5



Heap Sort

So, we will start with the ((size / 2) - 1) element, and compare it with its left and right child and swap the largest child with the parent if the parent is less than the child.

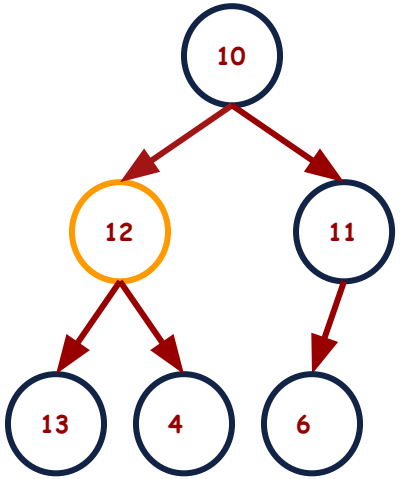
10	12	11	13	4	6
0	1	2	3	4	5



Heap Sort

Now, do the same process for all the previous elements one by one.

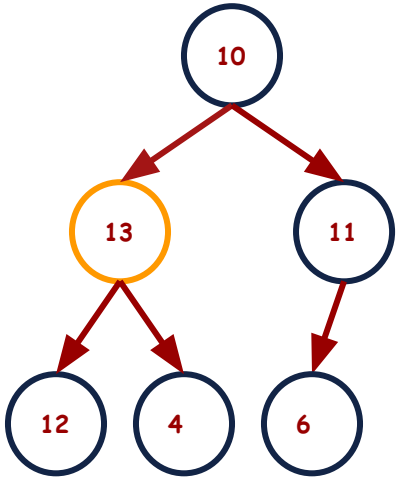
10	12	11	13	4	6
0	1	2	3	4	5



Heap Sort

Now, do the same process for all the previous elements one by one.

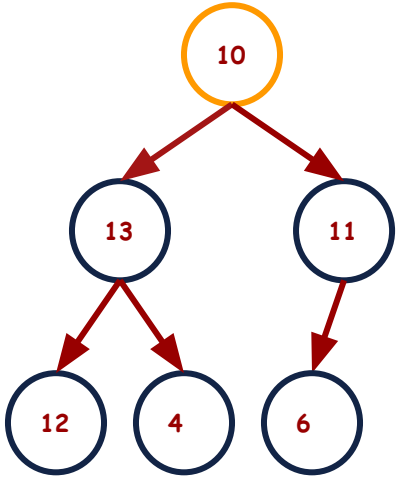
10	13	11	12	4	6
0	1	2	3	4	5



Heap Sort

Now, do the same process for all the previous elements one by one.

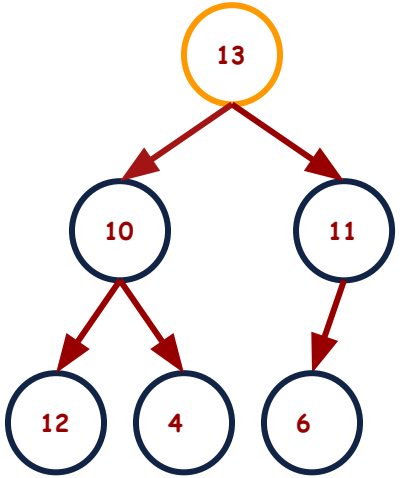
10	13	11	12	4	6
0	1	2	3	4	5



Heap Sort

Now, do the same process for all the previous elements one by one.

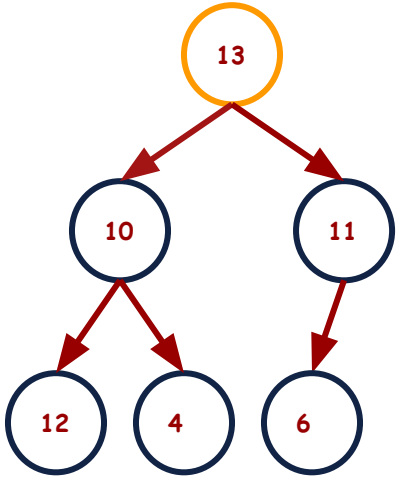
13	10	11	12	4	6
0	1	2	3	4	5



Heap Sort

Now, 10 is not in its right place. Therefore, we have to apply the heapify(sift down) method until we reach the last non-leaf element.

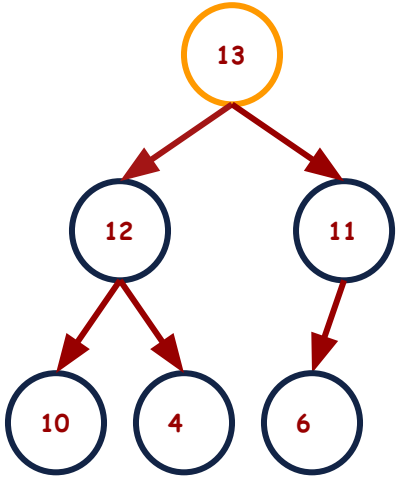
13	10	11	12	4	6
0	1	2	3	4	5



Heap Sort

Now, 10 is not in its right place. Therefore, we have to apply the heapify(sift down) method until we reach the last non-leaf element.

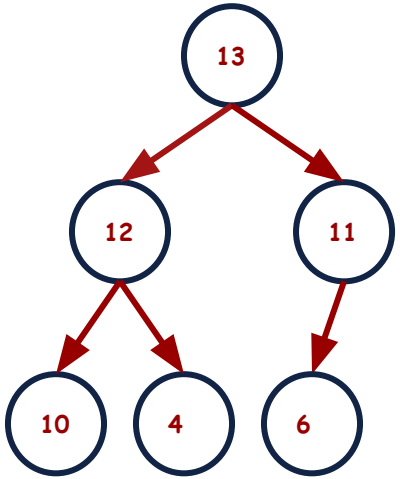
13	12	11	10	4	6
0	1	2	3	4	5



Heap Sort

Now, we have made the Max Heap in the same array in which data was given.

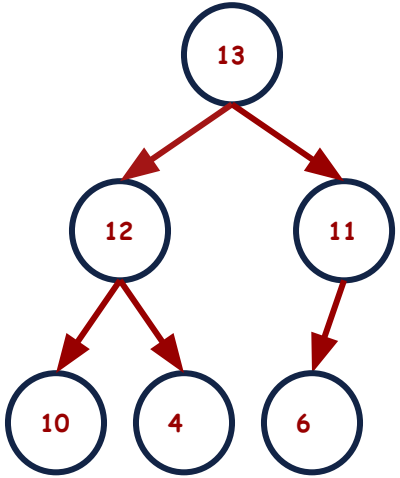
13	12	11	10	4	6
0	1	2	3	4	5



Heap Sort

Now, what should we do to sort the data in the same heap?

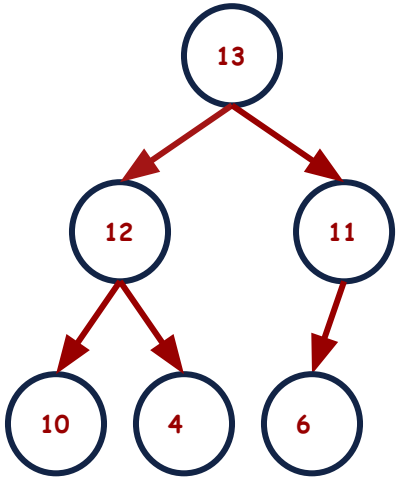
13	12	11	10	4	6
0	1	2	3	4	5



Heap Sort

We know that the max element is at the start of the array.

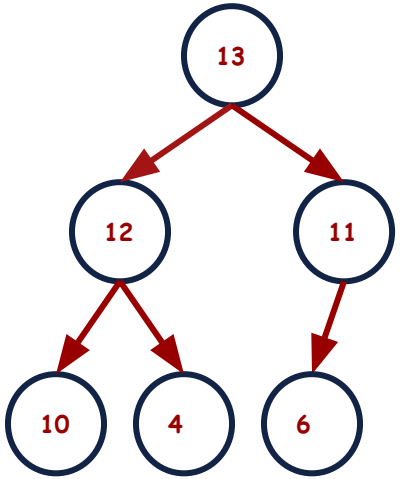
13	12	11	10	4	6
0	1	2	3	4	5



Heap Sort

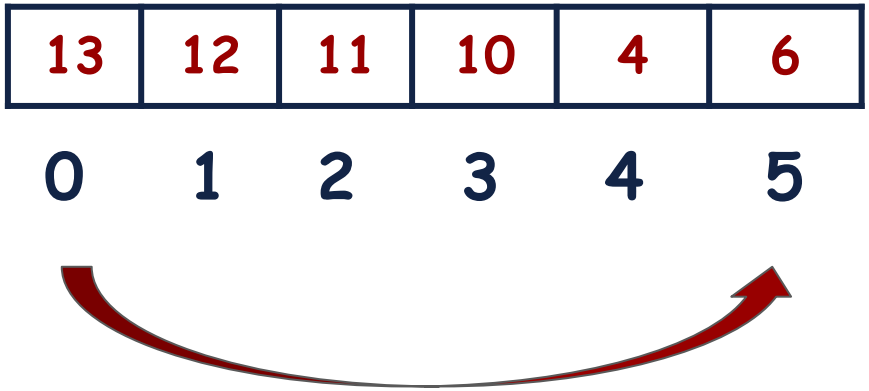
Let's swap the largest element with the last element and then apply the heapify (Sift Down) method to all the array except the last element.

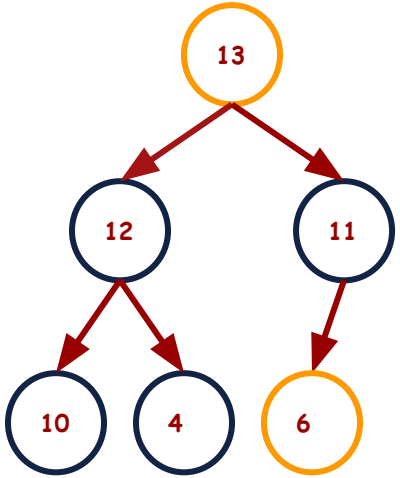
13	12	11	10	4	6
0	1	2	3	4	5



Heap Sort

Let's swap the largest element with the last element and then apply the heapify (Sift Down) method to all the array except the last element.

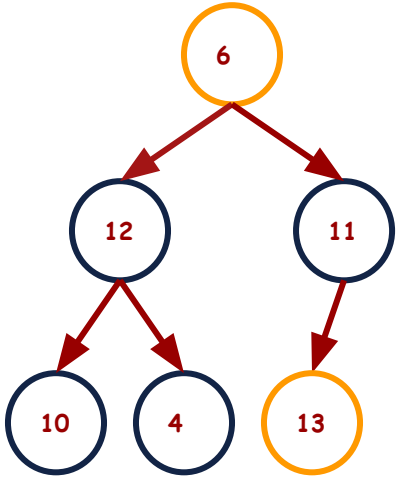




Heap Sort

Let's swap the largest element with the last element and then apply the heapify (Sift Down) method to all the array except the last element.

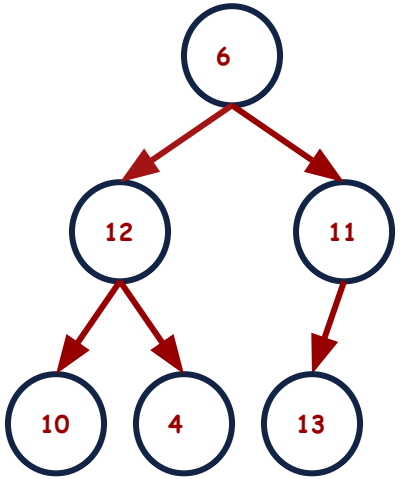
6	12	11	10	4	13
0	1	2	3	4	5



Heap Sort

Now, Apply the heapify (Sift Down) method to all the array except the last element.

6	12	11	10	4	13
0	1	2	3	4	5

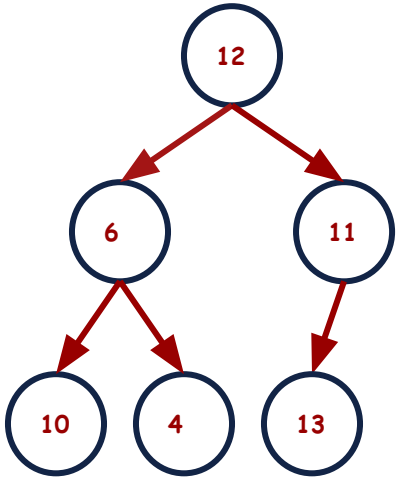


Apply Heapify Again

Heap Sort

Now, Apply the heapify (Sift Down) method to all the array except the last element.

12	6	11	10	4	13
0	1	2	3	4	5

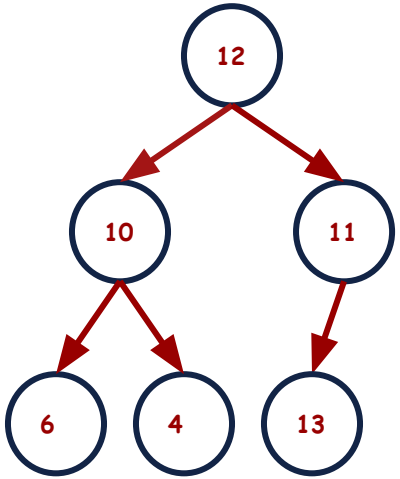


Apply Heapify Again

Heap Sort

Now, Apply the heapify (Sift Down) method to all the array except the last element.

12	10	11	6	4	13
0	1	2	3	4	5

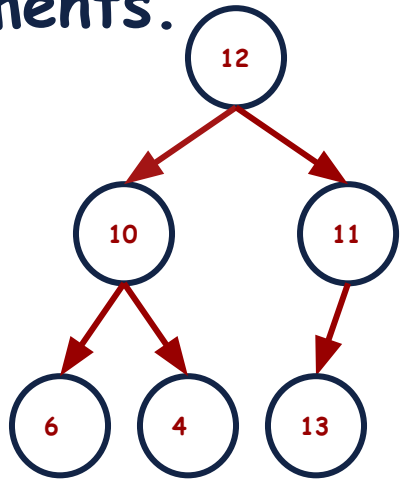


Apply Heapify Again

Heap Sort

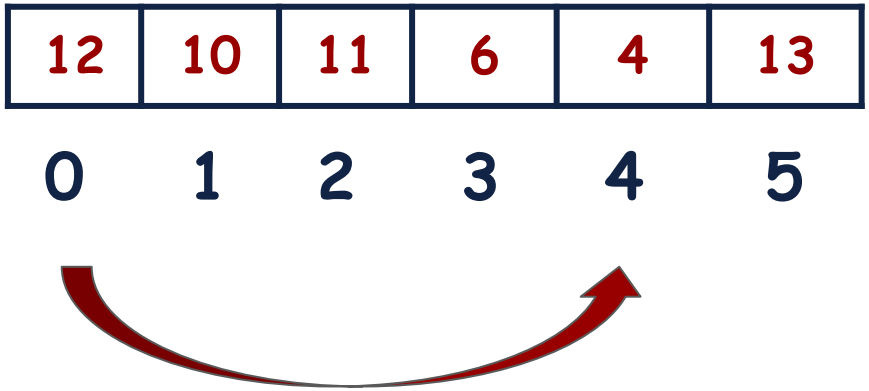
Let's swap the largest element with the second last element and then apply the heapify (Sift Down) method to all the array except the last two elements.

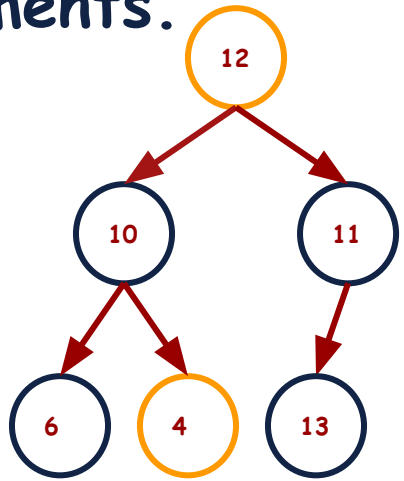
12	10	11	6	4	13
0	1	2	3	4	5



Heap Sort

Let's swap the largest element with the second last element and then apply the heapify (Sift Down) method to all the array except the last two elements.

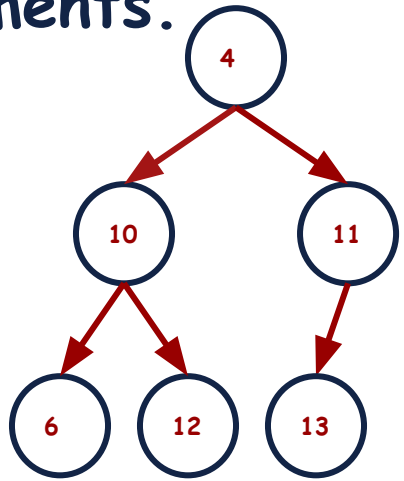




Heap Sort

Let's swap the largest element with the second last element and then apply the heapify (Sift Down) method to all the array except the last two elements.

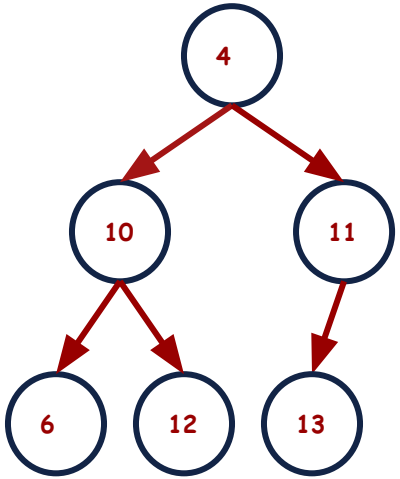
4	10	11	6	12	13
0	1	2	3	4	5



Heap Sort

Now, Apply the heapify (Sift Down) method to all the array except the last two elements.

4	10	11	6	12	13
0	1	2	3	4	5

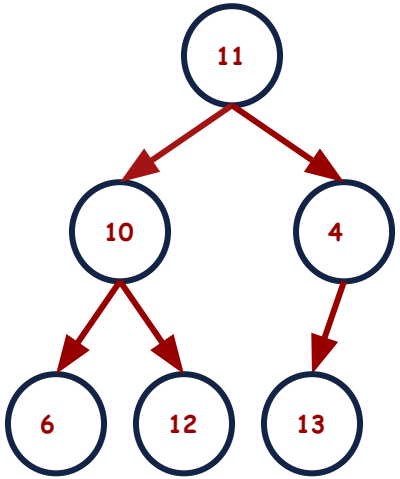


Apply Heapify Again

Heap Sort

Now, Apply the heapify (Sift Down) method to all the array except the last two elements.

11	10	4	6	12	13
0	1	2	3	4	5

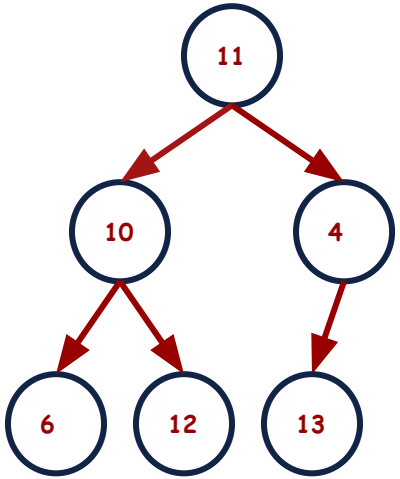


Apply Heapify Again

Heap Sort

Let's swap the largest element with the third last element and then apply the heapify (Sift Down) method to all the array except the last three elements.

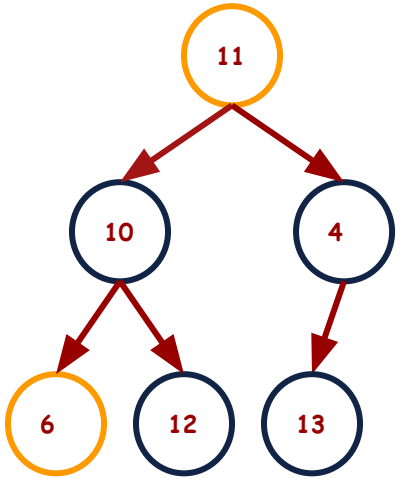
11	10	4	6	12	13
0	1	2	3	4	5



Heap Sort

Let's swap the largest element with the third last element and then apply the heapify (Sift Down) method to all the array except the last three elements.

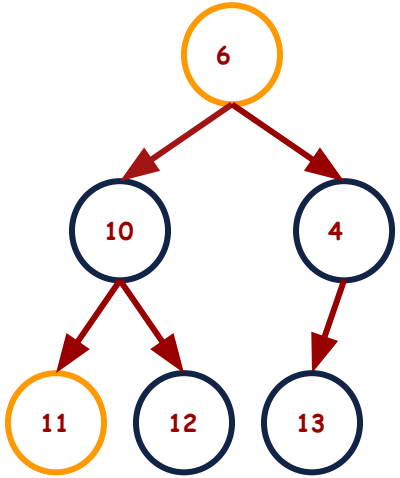
11	10	4	6	12	13
0	1	2	3	4	5



Heap Sort

Let's swap the largest element with the third last element and then apply the heapify (Sift Down) method to all the array except the last three elements.

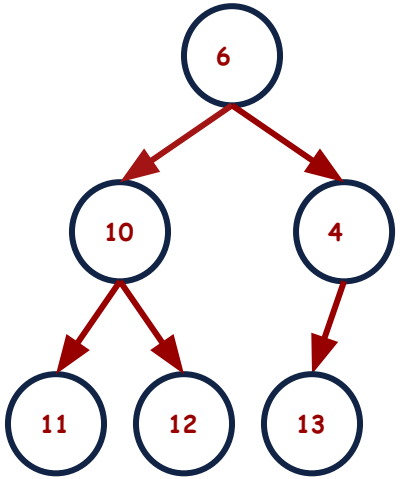
6	10	4	11	12	13
0	1	2	3	4	5



Heap Sort

Now, apply the heapify (Sift Down) method to all the array except the last three elements.

6	10	4	11	12	13
0	1	2	3	4	5

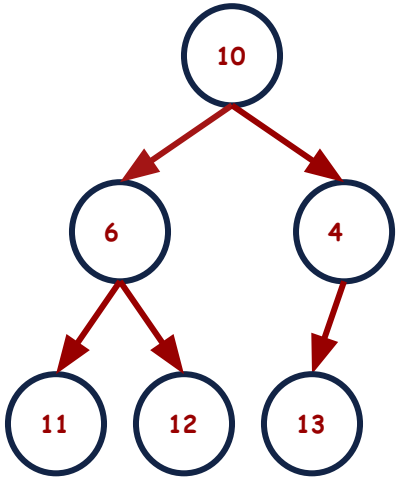


Apply Heapify Again

Heap Sort

Now, apply the heapify (Sift Down) method to all the array except the last three elements.

10	6	4	11	12	13
0	1	2	3	4	5

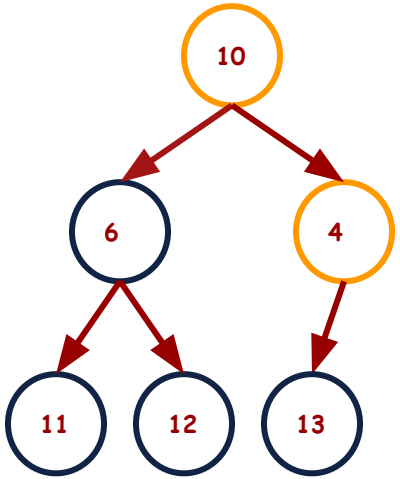


Apply Heapify Again

Heap Sort

Let's swap the largest element with the fourth last element and then apply the heapify (Sift Down) method to all the array except the last four elements.

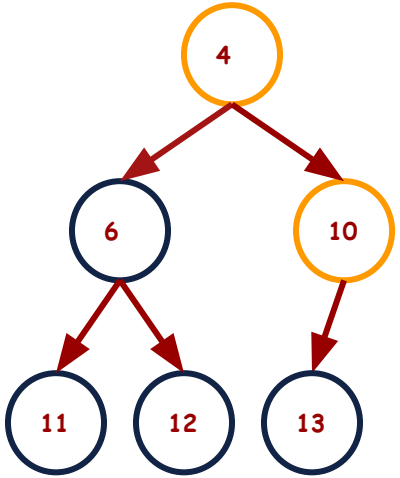
10	6	4	11	12	13
0	1	2	3	4	5



Heap Sort

Let's swap the largest element with the fourth last element and then apply the heapify (Sift Down) method to all the array except the last four elements.

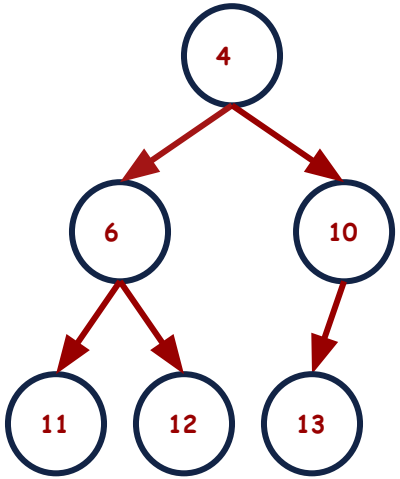
4	6	10	11	12	13
0	1	2	3	4	5



Heap Sort

Now, apply the heapify (Sift Down) method to all the array except the last four elements.

4	6	10	11	12	13
0	1	2	3	4	5

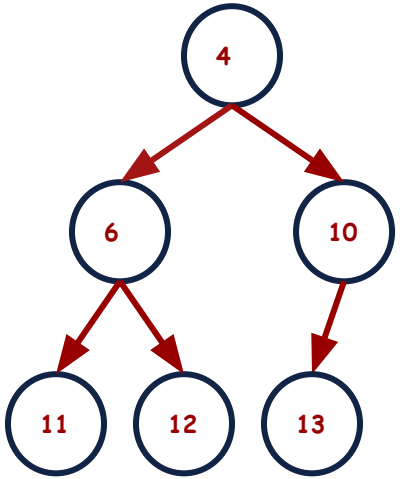


Heap Sort

Now, apply the heapify (Sift Down) method to all the array except the last four elements.

4	6	10	11	12	13
0	1	2	3	4	5

Apply Heapify Again

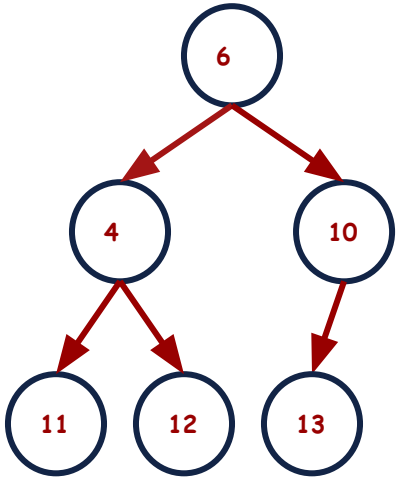


Heap Sort

Now, apply the heapify (Sift Down) method to all the array except the last four elements.

6	4	10	11	12	13
0	1	2	3	4	5

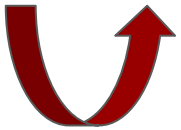
Apply Heapify Again

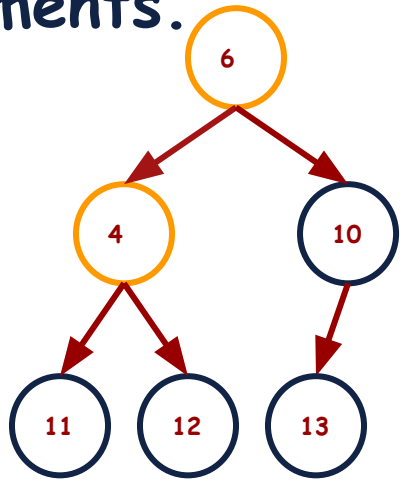


Heap Sort

Let's swap the largest element with the fifth last element and then apply the heapify (Sift Down) method to all the array except the last five elements.

6	4	10	11	12	13
0	1	2	3	4	5

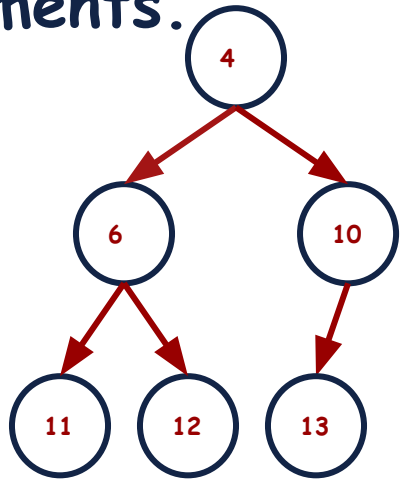




Heap Sort

Let's swap the largest element with the fifth last element and then apply the heapify (Sift Down) method to all the array except the last five elements.

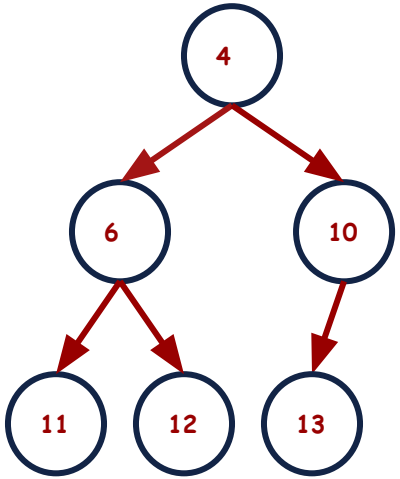
4	6	10	11	12	13
0	1	2	3	4	5



Heap Sort

Now, the data is sorted in the same array after applying heapsort.

4	6	10	11	12	13
0	1	2	3	4	5



Heap Sort: Implementation

Let's code the solution.

Heap Sort

```
int parentIndex(int i)
{
    return (i - 1) / 2;
}

int leftChildIndex(int i)
{
    return (2 * i) + 1;
}

int rightChildIndex(int i)
{
    return (2 * i) + 2;
}

void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
void heapify(int heapArr[], int size, int index){
    int maxIndex;
    while (true){
        int lIdx = leftChildIndex(index);
        int rIdx = rightChildIndex(index);
        if (rIdx >= size){
            if (lIdx >= size)
                return;
            else
                maxIndex = lIdx;
        }
        else{
            if (heapArr[lIdx] >= heapArr[rIdx])
                maxIndex = lIdx;
            else
                maxIndex = rIdx;
        }
        if (heapArr[index] < heapArr[maxIndex]){
            swap(heapArr[index], heapArr[maxIndex]);
            index = maxIndex;
        }
        else
            return;
    }
}
```

Heap Sort: Implementation

```
int main()
{
    int arr[6] = {10, 12, 6, 13, 4, 11};
    heapSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
int heapSort(int heapArr[], int size)
{
    for (int x = (size / 2) - 1; x >= 0; x--)
    {
        heapify(heapArr, size, x);
    }
    for (int x = size - 1; x > 0; x--)
    {
        swap(heapArr[0], heapArr[x]);
        heapify(heapArr, x, 0);
    }
}
```

Heap Sort: Time Complexity

What is the Time Complexity?

```
int main()
{
    int arr[6] = {10, 12, 6, 13, 4, 11};
    heapSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
int heapSort(int heapArr[], int size)
{
    for (int x = (size / 2) - 1; x >= 0; x--)
    {
        heapify(heapArr, size, x);
    }
    for (int x = size - 1; x > 0; x--)
    {
        swap(heapArr[0], heapArr[x]);
        heapify(heapArr, x, 0);
    }
}
```

Heap Sort: Time Complexity

What is the Time Complexity?

```
int main()
{
    int arr[6] = {10, 12, 6, 13, 4, 11};
    heapSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

Heapify method takes $\log_2(n)$ time.

```
int heapSort(int heapArr[], int size)
{
    for (int x = (size / 2) - 1; x >= 0; x--)
    {
        heapify(heapArr, size, x);
    }
    for (int x = size - 1; x > 0; x--)
    {
        swap(heapArr[0], heapArr[x]);
        heapify(heapArr, x, 0);
    }
}
```

Heap Sort: Time Complexity

What is the Time Complexity?

```
int main()
{
    int arr[6] = {10, 12, 6, 13, 4, 11};
    heapSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

Heapify method takes $\log_2(n)$ time. There are 2 for loops that call the heapify method.

```
int heapSort(int heapArr[], int size)
{
    for (int x = (size / 2) - 1; x >= 0; x--)
    {
        heapify(heapArr, size, x);
    }
    for (int x = size - 1; x > 0; x--)
    {
        swap(heapArr[0], heapArr[x]);
        heapify(heapArr, x, 0);
    }
}
```

Heap Sort: Time Complexity

What is the Time Complexity?

```
int main()
{
    int arr[6] = {10, 12, 6, 13, 4, 11};
    heapSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
int heapSort(int heapArr[], int size)
{
    for (int x = (size / 2) - 1; x >= 0; x--)
    {
        heapify(heapArr, size, x);
    }
    for (int x = size - 1; x > 0; x--)
    {
        swap(heapArr[0], heapArr[x]);
        heapify(heapArr, x, 0);
    }
}
```

Heapify method takes $\log_2(n)$ time.
There are 2 for loops that call the heapify method.
First calls n/2 times and second calls n times.

Heap Sort: Time Complexity

What is the Time Complexity?

```
int main()
{
    int arr[6] = {10, 12, 6, 13, 4, 11};
    heapSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

Time Complexity: $O(n*\log_2(n))$

```
int heapSort(int heapArr[], int size)
{
    for (int x = (size / 2) - 1; x >= 0; x--)
    {
        heapify(heapArr, size, x);
    }
    for (int x = size - 1; x > 0; x--)
    {
        swap(heapArr[0], heapArr[x]);
        heapify(heapArr, x, 0);
    }
}
```

Sorting Algorithms

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Merge Sort	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N)$
Quick Sort	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N^2)$	$O(N)$
Heap Sort	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(1)$

Sorting Algorithms

https://www.geeksforgeeks.org/quick-sort/

https://www.geeksforgeeks.org/heap-sort/

Sorting Algorithm	In-Place	Stable
Bubble Sort	Yes	Yes
Selection Sort	Yes	No
Insertion Sort	Yes	Yes
Merge Sort	No	Yes
Quick Sort	Yes	No
Heap Sort	Yes	No

Learning Objective

Students should be able to apply sorting using Quick Sort and Heap Sort.

Self Assessment

To visually see the Algorithms Running

https://visualgo.net/en/sorting

Self Assessment

Out of Merge Sort, Quick Sort and Heap Sort, which algorithm is best Sorting Algorithm?