Deque in C++ STL

Deque or *Doubly ended queues* are sequence containers with the feature of expansion and contraction on both the ends.

> Unlike, Queues or Stacks, Deques allows us to perform insertion and deletion at both ends.

Deques are generally implemented using doubly-linked lists. But in C++ STL, Deques has some additional features:

- It allows performing random access.
- It allows performing arbitrary insertions in constant time.

Syntax:

```
deque<Type of Elements> deque_name;
```

Creating a Deque and some important functions

We already discussed the syntax of creating a Deque in C++ STL above. We will discuss a few important functions to insert items at front and end and access items from both ends.

- **push_front()**: This function is used to push elements into a deque from the front.
- **push_back()**: This function is used to push elements into a deque from the back.
- **front()** and **back()**: front() function is used to reference the first element of the deque container. back() function is used to reference the last element of the deque container.

Below program illustrate the creation of Deque and implementation of the above functions:

```

#include <deque>
#include <iostream>

using namespace std;

int main()
{
    // Creates an empty Deque
    deque<int> dq;

    // Push element at back
    dq.push_back(10);

    // Push element at front
    dq.push_front(20);

    // The Deque is: 20 10

    // Access front and back elements
    // of the Deque
    cout << "dq.front() : " << dq.front();
    cout << "\ndq.back() : " << dq.back();

    return 0;
}

```

Run

Output:

```
dq.front() : 20
dq.back() : 10
```

Traversing a Deque

We can traverse a deque, just like we traverse a vector. That is, using for loops, for each loop, or even using iterators.

Below programs create a Deque and traverse the Deque using foreach loop.

```
#include <deque>
#include <iostream>

using namespace std;

int main()
{
    // Creates and initializes the
    // Deque at the same time
    deque<int> dq = { 10, 20, 30, 40, 50 };

    // Traversing the Deque using
    // foreach loop
    cout << "Deque : ";
    for (auto x : dq)
        cout << x << " ";

    return 0;
}

```

Run

Output:

```
Deque : 10 20 30 40 50
```

More functions on Deque

- **deque::begin() and deque::end in C++ STL**: begin() function is used to return an iterator pointing to the first element of the deque container. end() function is used to return an iterator pointing to the last element of the deque container.
- **deque insert() function in C++ STL**: Inserts an element. And returns an iterator that points to the first of the newly inserted elements. We can also insert multiple elements using insert() function in Deque. Learn about insert() in Deque in details here.
- **deque::pop_front() and deque::pop_back() in C++ STL**: pop_front() function is used to pop or remove elements from a deque from the front. pop_back() function is used to pop or remove elements from a deque from the back.
- **deque::empty() and deque::size() in C++ STL**: empty() function is used to check if the deque container is empty or not. size() function is used to return the size of the deque container or the number of elements in the deque container.
- **deque::clear() and deque::erase() in C++ STL**: The clear() function is used to remove all the elements of the deque container, thus making its size 0. The erase() function is used to remove elements from a container from the specified position or range.

Below is the implementation of the above functions:

```

#include <deque>
#include <iostream>

using namespace std;

int main()
{
    // Creates and initializes the
    // Deque with following elements
    deque<int> dq = { 10, 15, 30, 5, 12 };

    // Get the iterator to the first element
    // of the Deque
    auto it = dq.begin();

    // Increment iterator to second element
    it++;

    // Insert an element 20 at position
    // pointed by 2, that is, second position
    dq.insert(it, 20);

    // Deque now wil be: 10, 20, 15, 30, 5, 12

    // Pop front and back elements
    dq.pop_front();
    dq.pop_back();

    // Deque now is: 20, 15, 30, 5
```

Run

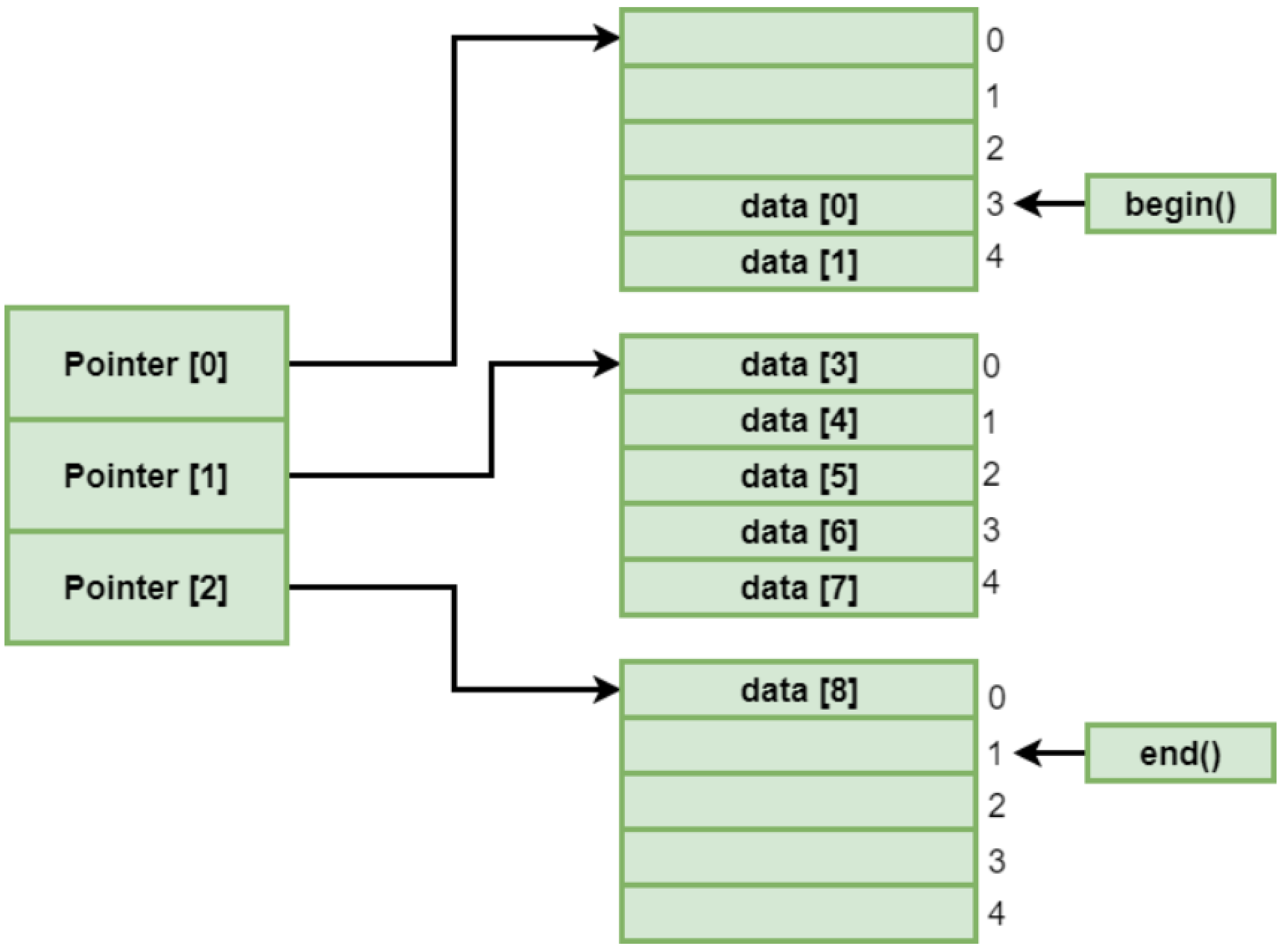
Output:

```
Deque Size: 4
Iterator it points to: 15
Curret Deque: 20 3 3 15 30 5
```

How does Deque works internally?

All the above-mentioned functions and operations are executed in **O(1)** time in a doubly-linked list but these lists cannot randomly access any elements. So goes with the deque in C++. This O(1) in deque can be achieved using a Circular Array. Using a circular array, operations like insertion and deletion from the front and back of an array can be achieved using in O(1) time along with random access of elements. But this comes with a problem. When the deque grows beyond the capacity, then the user will need to double the array size and copy all the data into the array. Also if the data is some user-defined object, then the cost of doubling and copying the data becomes really costlier.
Here is a basic solution. Deque uses some tricky implementations and when it says O(1) to push_back() and push_front(), then it is actually the constant time in terms of the number of copy constructors called. So if the data object is some class object which has multiple members, then minimizing the number of copy constructors call would save time. Also, the number of copy constructors calls are constant. Now let's look at how can this be achieved.

1. This can be achieved by the use array of pointers pointing to some fixed-sized blocks, containing the deque data. Here is an illustrative example.



2. These Deque data is divided into fixed-size blocks. Here we have considered the data being divided into fixed-block of size 5.

3. Filling of the blocks begins from the middle in both the arrays of deque ad the pointer and extends to forward and backward directions using push_front and push_back operations. The middle block is generally full and when it is filled then the data are moved to the upper or lower blocks.

4. In the upper block, elements are pushed in the reverse order as the first position to fill data would be 4 in this case, then 3, 2, 1, 0. But in the middle and lower blocks, data are filled in a forward order as in 0, 1, 2, 3, 4 and so on.

5. When the upper block is filled, then the pointer creates a new block and starts pointing to a new array block. This creates space for more data. In this case the pointer block can also be filled. This leads to a problem.

6. This is when doubling comes to rescue. In doubling, the pointer array doubles its size. This does not copy the whole of the data but just the pointers. This is the general argument proposed by many when the discussion of constant time takes place. The time remains constant in terms of the number of copy constructors called.

7. If the data set is very large, then the pointers block would hardly perform doubling as a single pointer can point to a huge chunk of data. Therefore there is a very rare chance of the pointer array getting filled and doubling taking place.

This sort of implementation is highly recommended as they have the following time complexities on these operations.

- **push_back():** O(1)
- **push_front():** O(1)
- **pop_front():** O(1)
- **pop_back():** O(1)
- **random_access():** O(1)

In this, the number of copy constructors called remains constant.

For the following operation, the contiguous memory setup is needed. Therefore access of any elements is done one by one leading to O(n) time complexity.

- **insert():** O(n)
- **erase():** O(n)

Note: The time complexity of push_back and push_front are arguable, as they are not constant in the worst case, bcoz of the array of pointers that might need resizing. The answer to that is they do a constant number of copy constructor calls and resizing happens very rarely.

Sample Problem : Sliding Window Maximum

Problem: Given an array and an integer **K**, find the maximum for each and every contiguous subarray of size k.

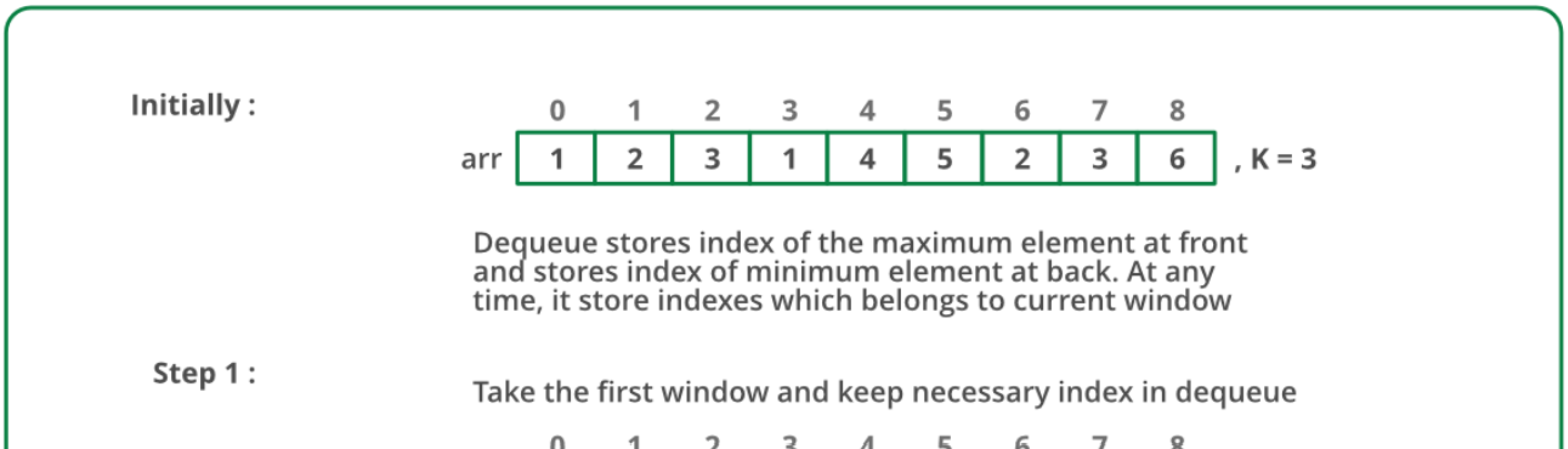
Examples :

```
Input: arr[] = {1, 2, 3, 1, 4, 5, 2, 3, 6}, K = 3
Output: 3 3 4 5 5 5 6

Input: arr[] = {8, 5, 10, 7, 9, 4, 15, 12, 90, 13}, K = 4
Output: 10 10 10 15 15 90 90
```

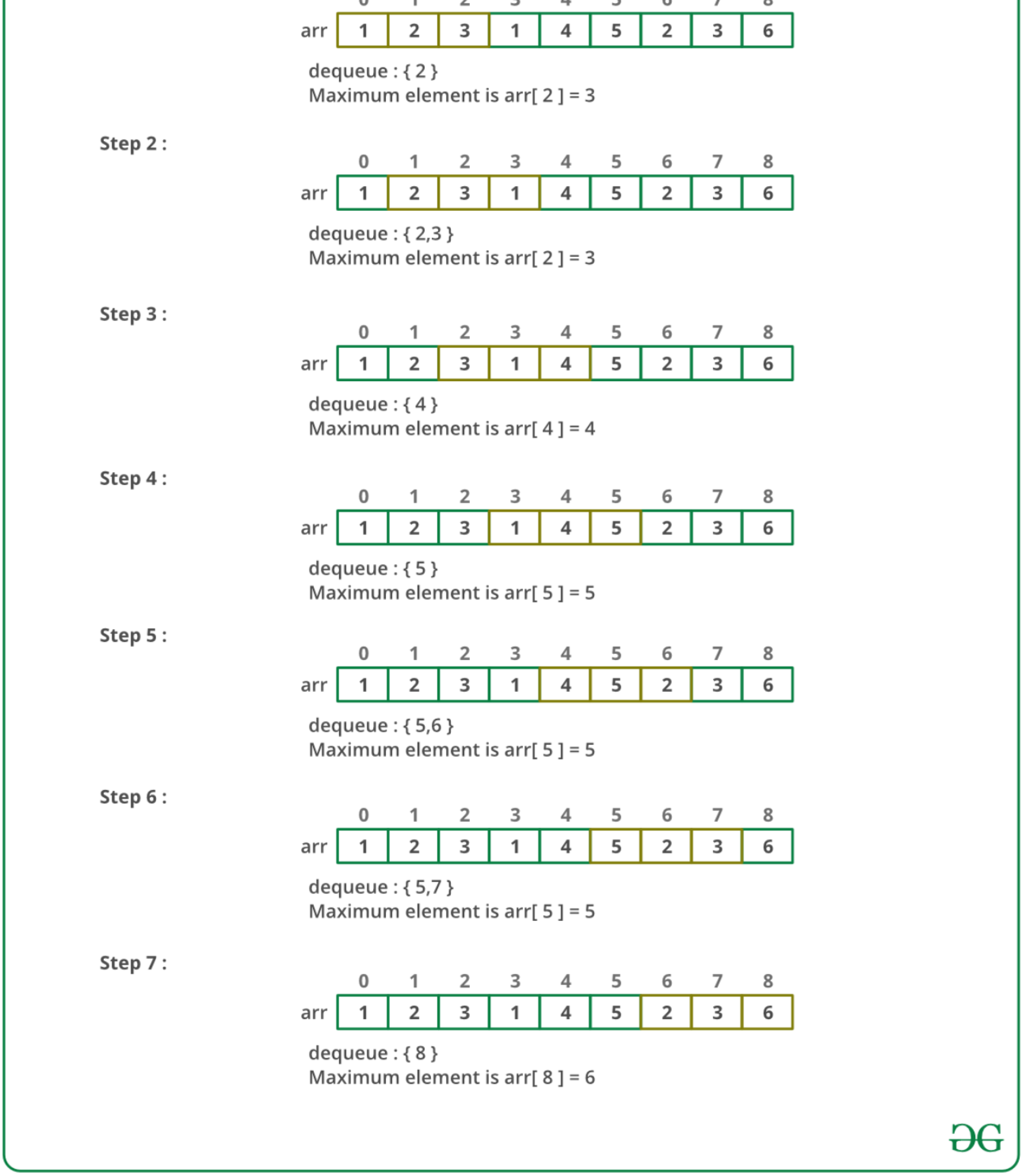
Solution (A O(n) method: using Deque): We create a Deque, *Qi* of capacity k, that stores only useful elements of current window of k elements. An element is useful if it is in the current window and is greater than all other elements on the left side of it in the current window. We process all array elements one by one and maintain *Qi* to contain useful elements of the current window and these useful elements are maintained in sorted order. The element at front of the *Qi* is the largest and element at rear of *Qi* is the smallest of current window.

Below image is a dry run of the above approach:



Dequeue stores index of the maximum element at front and stores index of minimum element at back. At any time, it store indexes which belongs to current window

Take the first window and keep necessary index in dequeue



Below is the implementation of the above approach:

```

#include <deque>
#include <iostream>

using namespace std;

// A Dequeue (Double ended queue) based method for
// printing maximum element of all subarrays of size k
void printKMax(int arr[], int n, int k)
{
    // Create a Double Ended Queue, Qi that will store indexes of array elements
    // The queue will store indexes of useful elements in every window and it will
    // maintain decreasing order of values from front to rear in Qi, i.e.,
```

```cpp
    // arr[Qi.front[]] to arr[Qi.rear()] are sorted in decreasing order
    std::deque<int> Qi(k);

    /* Process first k (or first window) elements of array */
    int i;
    for (i = 0; i < k; ++i) {
        // For every element, the previous smaller
        // elements are useless so remove them from Qi
        while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back(); // Remove from rear

        // Add new element at rear of queue
        Qi.push_back(i);
    }

    // Process rest of the elements, i.e.,
    // from arr[k] to arr[n-1]
```

Run

Output:

```
3 3 4 5 5 5 6
```

Time Complexity: O(n). It seems more than O(n) at first look. If we take a closer look, we can observe that every element of the array is added and removed at most once. So there are total 2*n operations.
Auxiliary Space: O(k)