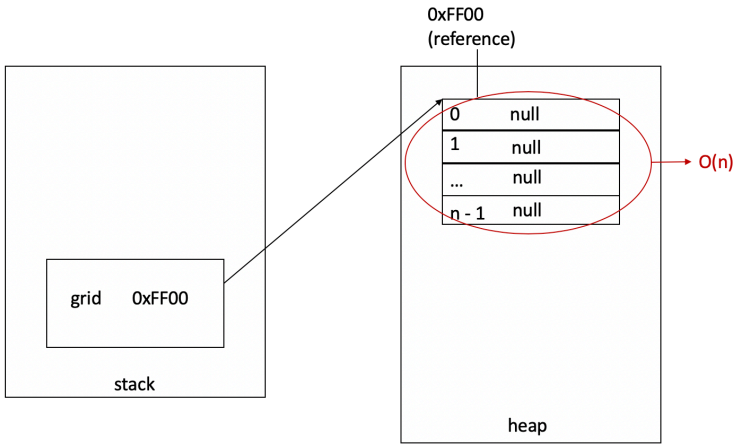
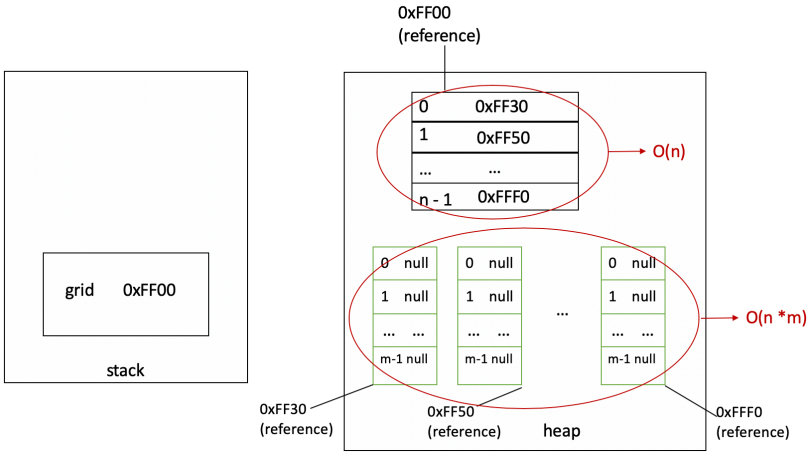
1. I consider that the implementation's memory complexity is O(n * (m + 1)) when I assume the grid's width is "n" and the grid's length is "m". I use the two-dimensional array to sort these elements. The specific step to count this bound is shown below.

   A) First of all, the Java Virtual Machine needs to distribute n space because the number of rows of this array are n. Therefore, the current memory complexity is O(n). The memory distribution like the diagram below (assume all references are correct):



   B) Secondly, we know that each row also can sort m elements, so the memory complexity of each row is O(m). we know we have n rows, so the current memory complexity is O(n * m). The memory distribution like the diagram below (assume all references are correct):



   C) Finally, From the two diagrams above , we know the whole data structure's memory complexity is O(n) + O(n * m), which is equal to O(n * (m + 1)).

2. I consider that if my grid is very large but has very few elements, it may lead to large waste of memory resources. Therefore, under this condition, my implementation is not efficient. For example, I use nested loop to implement the method about clearing the grid. The code is shown below:

```
public void clear() {
    for (int i = 0; i < grid.length; i++){
        for (int j = 0; j < grid[i].length; j++){
            grid[i][j] = null;
        }
    }
}
```

If the grid is very large but has very few elements, it means that the grid has been occupied by many null values. However, from the diagram above, the method of clearing grid will still store the null value in the grid again and again. Therefore, it may lead to waste of memory and time.

3. For the implementation of resizing the grid, there exists disturbing problem, which is that I use nested loop to complete the copy of array. We know that nested loop will consume much time. Therefore, I choose the “Arrays.copyOf” method which is under the “java.util.Arrays” class. This method can reduce the time of copying the elements from original to target array. The source code of “Arrays.copyOf” method is shown below:

[1]

```
@HotSpotIntrinsicCandidate
public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType) {
    @SuppressWarnings("unchecked")
    T[] copy = ((Object)newType == (Object)Object[].class)
        ? (T[]) new Object[newLength]
        : (T[]) Array.newInstance(newType.getComponentType(), newLength);
    System.arraycopy(original, 0, copy, 0,
                     Math.min(original.length, newLength));
    return copy;
}
```

The “Arrays.copyOf” mainly tells us that we need to create a new empty array called “copy”, which has the same type as the original array. After that, it will call continue the other method which is “System.arraycopy” method. The “System.arraycopy” method is defined as native method. The native type method means that it is coded with different programming language. Finally, the “Arrays.copyOf” method will return the “copy” array. However, from this method, we know that the value of return is one-dimensional array, so we cannot copy the two-dimensional array directly. We should put this method inside the loop. The advantage of this method is that we no longer need to use nested loop to copy the array, although it still needs one loop to implement, which means that we can save much time when we resize our grid.

For the implementation of clearing the grid, I select the “Arrays.fill” as alternative implementation. “Arrays.fill” is also under the “java.util.Arrays” class. Here is the source code of the method:

[2]

```
public static void fill(Object[] a, Object val) {
    for (int i = 0, len = a.length; i < len; i++)
        a[i] = val;
}
```

This method is quite easy to understand, which is similar with the implementation that I write because this method is only suitable to one-dimensional array. Therefore, both the alternative implementation and my own implementation are involved to nested loop. However, after the running time test by myself, I found that when the size of grid is very large (width:>=10000, height:>=10000), the alternative implementation always faster than the common nested loop implementation. Therefore, I consider that although both implementations of clearing grid use nested loop, “Arrays.fill” method is faster.

4. For the implementations of resizing grid, I think the runtime efficiency of alternative implementation is obviously higher than my own implementation, mainly because I use nested loop to copy the array, but the alternative implementation which uses “Arrays.copyOf” method only has one loop(O(n)). In the field of memory efficiency, I consider they are similar, because both implementations need to create an empty array for sorting all elements from original array.

   For the implementations of clearing grid, although they have similar principle, the runtime efficiency of alternative implementation is slightly better than my own implementation, but for the memory efficiency, I think they are in the same level.

5. References:
   [1] The source code is retrieved from JAVA API CLASS FILE.
   [2] The source code is retrieved from JAVA API CLASS FILE.