COMP3506 – Homework 1

Memory Complexity – ArrayGrid.java:

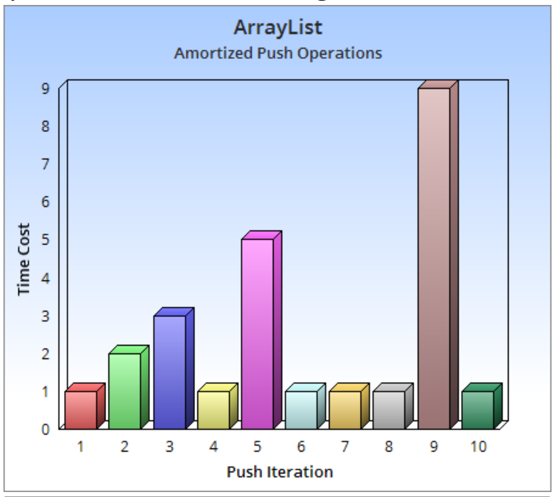
- public ArrayGrid(int width, int height) → O(n)
- public void add(int x, int y, T element) → O(1)
- public T get(int x, int y) → O(1)
- public boolean remove(int x, int y) → O(1)
- public void clear() → O(n)
- public void resize(int newWidth, int newHeight) → O(n)

This memory complexity is relative to the size of the grid, that is why big O notation is used.

In the case that the size of the grid is very large and most of the elements are null, the memory complexity of the functions is not going to change. This is because O(n) memory complexity is relative to the grid's size. Since the grid is initialized in full and its size is not dynamically allocated it will not have to resize when an element is inserted, only when the *resize* function is called.

Alternative Implementation:

The alternative implementation used is, **Constant Amortised Analysis**. The significant advantage of this method is that the array is dynamically sized rather than being initialized with a maximum height and width. This reduces the memory complexity of the constructor to a constant, O(1) which is more efficient that the current implementation, O(n). However, when elements are added to the array there is a chance that the index of the element being added is going to be greater than the fixed size array that has already been initialized. This causes a large operation for the program because it must double the size of the fixed size array. The operation to resize the array has a memory complexity of O(n) because the initial size of the array scales the time is takes to double it. There is going to be an infinite number of times that the array will double, n + 1, when the array is of size n (Amortized analysis, 2019).



https://upload.wikimedia.org/wikipedia/commons/e/e5/AmortizedPush.png

When an element is added to the array where the index is within the array's current size, the memory complexity of the operation is constant, O(1). However, when an element is added to the array where the index is outside of the array's current size the memory complexity of the operation is linear, O(n). Taking the average of adding elements and doubling the array evaluates the memory efficiency to be constant, O(1). Combining this with the constant memory complexity of the constructor means that the program, overall, has a more efficient memory complexity than the current implementation, this is an advantage.

References

Amortized analysis. (2019, 1 1). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Amortized_analysis