COMP3506 – Assignment 1

Kenton Lam

Due 09/08/2019 6:00 pm

Question 2

Memory Complexity

We assume that each grid element takes constant space. Then, in this implementation, the memory used is of the order $O(wh)$ where w and h are the width and height of the grid, respectively. Note that in particular, if T is String, this is not the case as each string can take an arbitrary amount of space. A

This is calculated by considering the private fields of the implemented ArrayGrid class. The *width* and *height* members take constant space, k. The *arrayData* member takes space directly proportional to the area of the grid, $w \times h$. Together, it takes $O(wh + k)$ space. However, because the order of wh greater than the constant width and height integers, we discard the constant k to get $O(wh)$.

Memory Efficiency

The grid allocates space for every element even if the element has not been added yet. This is not a problem if you expect that most cells would be filled. However, if the grid is very large but mostly empty (i.e. it is sparse), most of this space would remain empty (holding null values).

An Alternative

One possible alternative implementation is a hash map (also known as an associative array or dictionary). This maps keys as ordered pairs (x, y) to their values by performing a hash of the key and storing the value in a 'bucket' determined by this hash [1]. This would use less memory by only allocating space as elements are added. A hashmap only needs to store the added elements and assumes everything else is null. As a result, the memory complexity is $O(n)$ where n is the number of elements added.

Comparison

Memory complexity has been discussed above. For time complexity of each method,

- **add(int x, int y, T element)** – both implementations have worst case $O(1)$ time inserts, because both the hash function and array indexing are constant time operations. Note that the hash map is only $O(1)$ if the hash function is properly implemented and keys are distributed evenly across its buckets; we assume this is the case [2].
- **get(int x, int y)** – both implementations are $O(1)$ time, similar to the add() function.

- **remove(int x, int y)** – again, both constant time because this is essentially the same as inserting a null value at the position.
- **clear()** – array grid would take $O(wh)$ time because it needs to iterate over every grid position to determine if a position has a stored value to clear. The hash map would take $O(n)$ time because it knows which positions have hashes stored and only needs to clear those.
- **resize(int newWidth, int newHeight)** – for array grid, this is $O(wh)$ (where w and h are the old width/height) because it must check every cell for elements and copy to the new array.

  However, a hash map is capable of growing to an arbitrary size [3]. Bounds on width and height would be done by restricting x and y in add/get/remove operations, not the underlying data structure. Because of this, a resize operation would also take $O(1)$ time on a hash map as only these limits would need to be changed.

In conclusion, a hash map would be more suitable for sparse values as it is more space efficient. However, although asymptotic analysis suggests the hash map is always better, the hash map's space efficiency required executing a hash function on every add/get/remove operation. If CPU time is more important than memory and the grid doesn't need to be resized or cleared often, an array-based implementation offers one of the fastest indexing operations to any position, resulting in extremely efficient add/get/remove's.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. Mit Press, 2009.

[2] E. Demaine, P. Indyk, and M. Kellis, "Resizing hash tables," Feb 2011. [Online]. Available: https://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf

[3] Oracle, "HashMap (Java Platform SE 8)," Mar 2019. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html