**SPACE COMPLEXITY**

**.Heap class - toString**

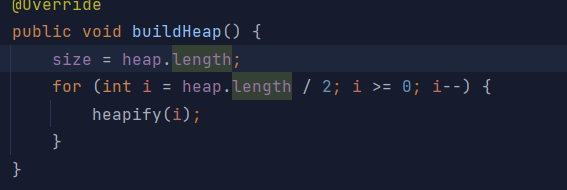
****

| **Variable Type** | **Variable Name** | **Atomic Value Size** | **Number of Atomic Values** |
| --- | --- | --- | --- |
| Output | StringBuilder msg | Variable (~4 bytes) | 1 |
| Auxiliary | t (inside the loop) | Variable (~4 bytes) | n |
| Auxiliary | t (local variable in the loop) | Variable (~4 bytes) | 1 |
| Auxiliary | size | Variable (~4 bytes) | 1 |

**Spatial = Output + Auxiliary = n + 3 = O(n)**

Therefore, in big O notation, the spatial complexity is O(n), where "n" represents the average number of elements in the heap, as the number of atomic values used by auxiliary variables depends linearly on this value.

**.Heap class - buildHeap**



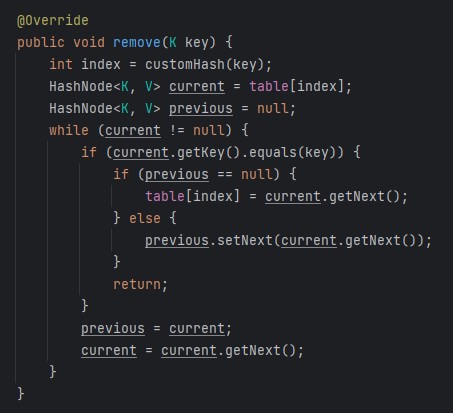
| **Type of variable** | **Variable name** | **Atomic Value Size** | **Number of atomic values** |
| --- | --- | --- | --- |
| Input | heap | 4 bytes | n |
| Auxiliary | i | 4 bytes | n |
| Output | size | 4 bytes | 1 |

The size of an atomic value in Java is 4 bytes. Therefore, the size of the variable heap is 4 \* n bytes, and the size of the variable i is 4 \* n bytes.The number of atomic values in the variable heap is n, and the number of atomic values in the variable i is n.

Therefore, the total spatial complexity of the buildHeap() method is O(n), where n is the size of the heap.

**TIME COMPLEXITY**

**HashTable class - remove**



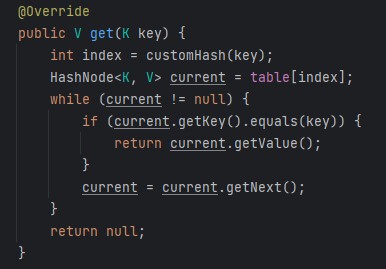
| Operation | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| remove(K key) | O(1) | O(1+α) | O(n) |

**Best Case** (O(1)): This remains the same as before - when the key to be deleted is at the first position of the linked list at the hashed index.

**Average Case** (O(1 + α)): On average, assuming simple uniform hashing (each key is equally likely to hash into any of the m slots independently of where other keys hash to), the time to search for an element in a hash table is O(1 + α), where α = n/m is the load factor.

**Worst Case** (O(n)): This also remains the same as before - when the key to be deleted is at the last position of the linked list at the hashed index or not present in the list.

**HashTable class - get**

****

| Operation | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| get(K key) | O(1) | O(1+α) | O(n) |

**Best Case (O(1)):** This scenario occurs when the key to be searched is at the first position of the linked list at the hashed index. In this case, the get operation only needs to return the value of the first node from the list. This operation is constant time because it only involves a single step, regardless of the size of the hash table or linked list.

**Average Case (O(1 + α))**: On average, assuming simple uniform hashing (each key is equally likely to hash into any of the m slots independently of where other keys hash to), the time to search for an element in a hash table is O(1 + α), where α = n/m is the load factor.

**Worst Case (O(n)):** This scenario occurs when the key to be searched is at the last position of the linked list at the hashed index or not present in the list. In this case, every node in the list needs to be traversed to find and return the value of the key. The time complexity is linear, O(n), where ‘n’ is the number of nodes in the list. This is because each node requires a constant amount of work (checking if current.getKey().equals(key) and updating pointers), and there are ‘n’ nodes to check