**ALGORITHMIC COMPLEXITY:**  For resolving algorithmic problems many solutions exist but we should select the most efficient. In this way, the idea is to search for an algorithm with an appropriate complexity, using certain criteria to compare solutions. These are time and space complexity.

* **Time Complexity:** Execution time of an algorithm
* **Space Complexity:** Memory used by an algorithm for its execution

In this project we are going to make both processes for two algorithms, these are:

| Method | Class |
| --- | --- |
| addFirst(T value) | DoubleLinkedList |
| deleteElement(K key) | Hashtable |

**Time Complexity for the method addFirst**

| Code: addFirst | steps |
| --- | --- |
| DoubleNode<T> newNode = new DoubleNode<>(value);  if (first == null) {  first = newNode;  last = newNode;  } else {  newNode.setNext(first);  first.setPrevious(newNode);  first = newNode;  }  numberOfElements++; | 1  1  1  1  1  1  1  1 |

T(n)=1+1+1+1+1+1= 6

T(n)=6

This result indicates that the complexity will always be constant regardless of the size of the list. In our case, for this method the complexity is O(6), in other words O(n).

**Space Complexity for the method addFirst**

| code | space |
| --- | --- |
| DoubleNode<T> newNode = new DoubleNode<>(value);  if (first == null) {  first = newNode;  last = newNode;  } else {  newNode.setNext(first);  first.setPrevious(newNode);  first = newNode;  }  numberOfElements++; | 1  1  1  1  1  1  1  1 |

The spatial complexity is given by the addition of all spaces in memory for each instruction. We will consider the simple operation as 1 access to the memory.

Ø(n)= 1+1+1+1+1+1+1+1

Ø(n)=8

This means that the spatial complexity is constant

**Time Complexity for the method for deleteElement**

| Code | Steps |
| --- | --- |
| int index = hashFunction(key);  if(isEmpty() == true) {  throw new HashIsEmptyException("The hash table is empty");  }  else{  if(hashList[index] == null){  throw new NonExistentKeyException("The object with the key provided doesn't exist");  }  else{  if(hashList[index].getNext() == null) {  hashList[index] = null;  size--;  }    else{  hashList[index].removeElement(key);  }  }  } | 1  1  1  1  1  1  1  1  n |

The complexity of the last operation (removeElement) is o(n) because it depends on the size of the list since it is recursive in the worst case, so our total complexity will be:

T(n)= 1+1+1+1+1+1+1+1+n

T(n)= 8+n

As we can see the n has dominance over this result, so we conclude that the complexity will be O(n), in other words constant.

**Spatial Complexity for the method deleteElement**

| code | space |
| --- | --- |
| int index = hashFunction(key);  if(isEmpty() == true) {  throw new HashIsEmptyException("The hash table is empty");  }  else{  if(hashList[index] == null){  throw new NonExistentKeyException("The object with the key provided doesn't exist");  }  else{  if(hashList[index].getNext() == null) {  hashList[index] = null;  size--;  }    else{  hashList[index].removeElement(key);  }  }  } | 1  1  1  n  1  1  n  1  n |

The spatial complexity is given by the addition of all spaces in memory for each instruction. We will consider the simple operation as 1 access to the memory and n as the access to a position in the hash table.

Ø(n)= 1+1+1+n+1+1+n+1+n

Ø(n)= 3n+6

Then we conclude that the spatial complexity is O(n) because the n predomines on the previous education.