# ArrayList Implementation

## Add(Song)

We define N to be the number of elements in the array.

**Time Complexity – Claim O(N)**

Appending to the end of an Array Based list operates in constant time because, assuming proper implementation and managing of a count variable, we know exactly what index to insert the new song into. If there is enough space in the array, we can add a new element in one step. The problem is that in the case of the array being full we will have to allocate a new array, twice the size of the original array. This requires us to iterate through each element in the previous array and copy that over to a new array. This results in a time complexity of O(N), in the worst case.

## Remove()

We define N to be the number of elements in the array.

**Time Complexity** **– Claim O(N)**

While the act of removing from the head of the ArrayList itself is only one action, we are required to shift all the following elements up one index. This results in us iterating through N-1 elements (excluding the element to be deleted) and copying each element to the index prior. We generalize N-1 actions to the linear function N. There, we achieve time complexity of O(N) in all cases for removal.

for i in range (1, N):

data[i-1] = data[i]

data[i-1] = null

## Remove(Song)

We define N to be the number of elements in the array

**Time Complexity – Claim O(N)**

It is still required that we transverse the entire array, but instead of shifting each element we have to do a conditional check to find the song we have to delete. Once we find the song, we must shift every element after that down by one index. This results in us writing a for loop that touches each element in the array list once, resulting in a linear approximation of O(N)

## Clear()

We define N to be the number of elements in the array.

**Time Complexity – Claim O(1)**

Clearing the array is completely independent of the size of the array because we can just replace the data reference with a reference to a new, empty array of size 2 and then reset the count variable to 0. Therefore, an array can be cleared with a constant number of operations, resulting in O(1) time complexity.

## isEmpty()

We define N to be the number of elements in the array.

**Time Complexity – Claim O(n)**

We must loop through the entire array in order to confirm that every index is empty and that the array is empty. Because we are touching every element in the array, we have O(N) time complexity.

## count()

We define N to be the number of elements in the array.

**Time Complexity – Claim O(1)**

Because we are maintaining a count variable that stores the number of elements in the array, it is very simple to get the count just by returning that variable. Therefore we can access the count with a single operation, resulting in O(1) constant time.

## get(i)

We define N to be the number of elements in the array.

**Time Complexity – Claim O(1)**

Thanks to the way arrays work, we have immediate access to any element in the array if given the index, therefore retrieval from a specific index runs in O(1) constant time.

## contains(song)

We define N to be the number of elements in the array.

**Time Complexity – Claim O(N)**

In order to find the song, we have to loop through the entire array, checking if each element is equal to the song that we are trying to find. This result in a time complexity of O(N) because we are touching each element in the array.

# LinkedList Implementation

## Add(Song)

We define N to be the number of elements in the LinkedList.

**Time Complexity – Claim O(N)**

In a singly linked list, we do not have direct access to the tail end of a pointer and therefore we would have to transverse the entire LinkedList (N items) in order to find the end. Once finding the tail, it will take one extra operation to change the pointer to point at the new node that we are appending. Therefore, it takes N operations in order to append to a linked list. Or O(N).

## Remove()

We define N to be the number of elements in the array.

**Time Complexity – Claim O(1)**

Removing from the head of the linked list is much simpler because all we have to do is change the head pointer to point at the second node in the list. As a result, all references to the previous head will be erased and Java’s garbage collector will pick it up and delete the old head. This can be done in one operation, and therefore remove runs in O(1) constant time.

## Remove(Song)

We define N to be the number of elements in the array.

**Time Complexity – Claim O(N)**

In order to delete a specific song from the LinkedList, we need to transverse the array, checking if each element is the song we are looking for. While we won’t necessarily touch every element in the array, it is possible that it will take N operations in the worst case and we generalize this to O(N). Once we find the element, it will take an additional operation in order to adjust to pointers accordingly so that we erase references to the element in question and it can be garbage collected. Therefore, we have O(N) time complexity.

## Clear()

We define N to be the number of elements in the array.

**Time Complexity – Claim O(1)**

By replacing the head reference with a null reference, we erase all references to the head of the linked list. Without a reference, Java’s garbage collector will automatically erase the entire LinkedList for us. This can be conducted with one operation, and therefore Clear() runs in constant time.

## isEmpty(Song)

We define N to be the number of elements in the array.

**Time Complexity – Claim O(1)**

Checking if a linked list is empty is trivial due to the fact that we have access to the head pointer. If the pointer is a null reference, we know automatically that the rest of the linked list is empty (null) and we can determine if it is empty or not with only a single constant operation. Therefore, O(1).

## Count()

We define N to be the number of elements in the array.

**Time Complexity – Claim O(1)**

Without the maintaining and implementation of a count variable, it would require trans versing the entire linked list in order to determine the number of elements in the LinkedList (this would run in O(N). However, since we are maintaining this counter we can return that variable in a single operation. Therefore, O(1).

## Get(i)

We define N to be the number of elements in the array.

**Time Complexity – Claim O(N)**

We are required to transverse the linked list until we find the element we are searching for. This is because in a linked list we only have access to one element (the head) at a time. While we won’t exactly travel N elements in the average case, the worst-case O(N) is that we travel all the way to the end of the LinkedList. Therefore, time complexity is O(N)

## Contains(song)

We define N to be the number of elements in the array.

**Time Complexity – Claim O(N)**

Again, similar to get(i) it requires that we transverse the entire linked list in order to find the element that we are searching for. Since the worst case is that the element is at the end of the list and we have to travel N elements in order to find it, we get that the time complexity is O(N).