11. Linked Lists :: Implementation :: DList Class :: clear()

The *clear*() method:

`+clear(): void`

removes all of the elements from the *DList*. After this operation the *DList* will be empty. Clearing the list is simple enough: simply remove the head element (at index 0) until the list becomes empty.

11. Linked Lists :: Implementation :: DList Class :: Time Complexity :: add()

How efficient are the various operations on our *DList* class? The key operation is to count how many nodes we must access to reach the location in the list where the operation, e.g., *add*(), will be performed.

1. Adding an element at the head (index 0). This is the same as *prepend*().

   $getNodeAt(0)$ would return a reference to the node at index 0 in constant $O(1)$ time. We create the new node and link it into the list in $O(1)$ time.

2. Adding an element before the tail (index *getSize*() - 1).

   $getNodeAt(0)$ would return a reference to the node at index *getSize*() - 1 in constant $O(1)$ time. We create the new node and link it into the list in $O(1)$ time.

3. Adding an element after the tail (index *getSize*() - 1). This is the same as *append*().

   Since we are appending, we have immediate access to the tail node—we do not need to call $getNodeAt()$ during an append—so we create the new node and link it into the list in $O(1)$ time.

4. Adding an element with $0 < index <$ *getSize*() - 1.

   The worst case behavior of $getNodeAt()$ is to access no more than *mSize* nodes in order to reach the node at *index*. Once the node at *index* is found, we create the new node and link it into the list in constant time. Therefore, the time complexity is $O(n)$.

11. Linked Lists :: Implementation :: DList Class :: Time Complexity :: get()

5. Getting the element at the head (index 0).

   $get()$ simply calls $getNodeAt(0)$ to get a reference to the node at index 0. Since $getNodeAt()$ checks for the special case of index 0 and simply returns the head reference in constant time, getting the element at the head is $O(1)$.

6. Getting the element at the tail (index $getSize()$ - 1).

   $get()$ simply calls $getNodeAt(getSize()$ - 1) to get a reference to the node at index $getSize()$ - 1. Since $getNodeAt()$ checks for the special case of index $getSize()$ - 1 and simply returns the tail reference in constant time, getting the element at the tail is $O(1)$.

7. Getting an element with $0 < index < getSize()$ - 1.

   $get()$ simply calls $getNodeAt(index)$ to get a reference to the node at $index$ and $getNodeAt()$ will locate this node in no more than $mSize$ loop iterations. Therefore, getting an element in the middle of the list is $O(n)$.

11. Linked Lists :: Implementation :: DList Class :: Time Complexity :: remove()

8. Removing the element at the head (index 0).

   $getNodeAt(0)$ would return a reference to the node at index 0 in constant time and we unlink the node and in constant time, therefore removing the element at the head is $O(1)$.

9. Removing the element at the tail (index *getSize*() - 1).

   $getNodeAt(0)$ would return a reference to the node at index *getSize*() - 1 in constant time and we unlink the node and in constant time, therefore removing the element at the tail is $O(1)$.

10. Removing an element with $0 < index <$ *getSize*() - 1.

    The worst case behavior of $getNodeAt()$ is to access no more than *mSize* nodes in order to reach the node at *index*. Once the node at *index* is found, we unlink it from the list in constant time. Therefore, the time complexity is $O(n)$.

11. Linked Lists :: Implementation :: DList Class :: Time Complexity :: Summary

To summarize:

1. *add*(), *get*(), and *remove*() at the head and tail of a doubly linked list is $O(1)$.
2. *add*(), *get*(), and *remove*() on an interior element—that is, randomly accessing elements—is $O(n)$.
3. Iterating over the elements of a linked list in sequence is an efficient operation as we can access the next element in the sequence in $O(1)$ time.

Therefore, a linked list is a good data structure for storing a linear sequence of elements where:

1. The number of elements in the list will grow and shrink, i.e., it is a dynamic data structure.
2. Elements are primarily accessed at the beginning and ends of the list.
3. Elements are accessed in a linear sequence.