(AS3\_ Double Linked Lists)

UML diagram with pseudo-code:

|  |
| --- |
| BasicDoubleLinkedList<T> |
| - Node head  - Node tail  - int size |
| + BasicDoubleLinkedList()  + getSize(): int  + addToEnd(data: T): void  + addToFront(data: T): void  + getFirst(): T  + getLast(): T  + remove(targetData: T, comparator: Comparator<T>): Node  + retrieveFirstElement(): T  + retrieveLastElement(): T  + toArrayList(): ArrayList<T>  + iterator(): ListIterator<T>''') |

|  |
| --- |
| ListNode<B> |
| - T data  - Node prev  - Node next |
| + Node(dataNode: T)''') |

|  |
| --- |
| DoubleLinkedListIterator<C> |
| - Node currentNode  - int currentIndex |
| + DoubleLinkedListIterator()  + hasNext(): boolean  + next(): T  + hasPrevious(): boolean  + previous(): T  + nextIndex(): int  + previousIndex(): int  + remove(): void  + set(e: T): void  + add(e: T): void''') |

|  |
| --- |
| SortedDoubleLinkedList<T> |
| - comparator: Comparator<T> |
| + add(data: T)  + addToEnd(data: T) throws UnsupportedOperationException  + addToFront(data: T) throws UnsupportedOperationException  + iterator() -> ListIterator<T>  + remove(data: T, comparator: Comparator<T>) -> Node ''') |

Psedocode:

* BasicDoubleLinkedList:
* Attributes:

Node head: The first node in the list.

Node tail: The last node in the list.

int size: The number of elements in the list.

* Constructor:

Initialize head, tail, and size to null and 0, respectively.

* Methods:

getSize(): Return the current size of the list.

addToEnd(data):

Create a new node with the given data.

If the list is empty (head is null):

Set both head and tail to the new node.

* + Otherwise:

Link the new node to the end of the list by updating the next of the tail

Update the tail reference to the new node.

* + Increment the size of the list.

addToFront(data):

Create a new node with the given data.

If the list is empty (head is null):

Set both head and tail to the new node.

* + Otherwise:

Link the new node to the front of the list by updating the prev of the current head and the next of the new node.

Update the head reference to the new node.

Increment the size of the list.

getFirst():

Return the data of the head node, or null if the list is empty.

getLast():

Return the data of the tail node, or null if the list is empty.

remove(targetData, comparator):

Initialize a current node to the head.

While the current node is not null:

If the comparator finds the current node's data equal to the target data:

Update the pointers of the surrounding nodes to remove the current node.

If the current node is the head, update head to the next node.

If the current node is the tail, update tail to the previous node.

Decrement the size of the list.

Return the removed node.

If the target data is not found, return null.

retrieveFirstElement():

If the head is null, return null.

Store the data of the head node.

Update the head to the next node.

If the new head is not null, update its previous reference to null.

If the head becomes null, set the tail to null.

Decrement the size of the list.

Return the stored data.

retrieveLastElement():

If the tail is null, return null.

Store the data of the tail node.

Update the tail to the previous node.

If the new tail is not null, update its next reference to null.

If the tail becomes null, set the head to null.

Decrement the size of the list.

Return the stored data.

toArrayList():

Create an empty ArrayList.

Initialize a current node to the head.

While the current node is not null:

Add the current node's data to the ArrayList.

Move to the next node.

Return the populated ArrayList.

iterator():

Return a new instance of the inner DoubleLinkedListIterator.

* SortedDoubleLinkedList:

Initialization:

The class is initialized with a comparator that decides how elements are compared (e.g., sorting in ascending or descending order).

Adding Elements:

When you add an element to the list, the algorithm:

Checks if the list is empty. If yes, the new element becomes both the head and tail.

Otherwise, it traverses the list and finds the correct position for the new element (based on the comparator).

The element is inserted either at the:

Beginning: If the new element is smaller than the first element.

End: If the new element is larger than the last element.

Middle: If the new element belongs somewhere between existing elements.

Finally, the size of the list is updated.

Prevent Invalid Operations:

The methods addToEnd() and addToFront() are disabled for this class, as direct addition to the start or end would break the list's sorting order. If someone tries to call these methods, the class throws an UnsupportedOperationException.

Iterating Through the List:

The list provides an iterator that allows you to go through the elements from the first to the last. This iterator is inherited from its parent class BasicDoubleLinkedList.

Removing Elements:

The remove() method is inherited from the parent class, allowing you to remove an element from the list by comparing it with the provided comparator.