List [Interface] & Queue [Interface]: LinkedList

Definition

LinkedList class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data and address part. The elements are linked using pointer and addresses. Each element is known as a node. It has disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach a node we wish to access.

Normal LinkedList

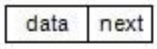


Figure 1: Element of a singly linked list

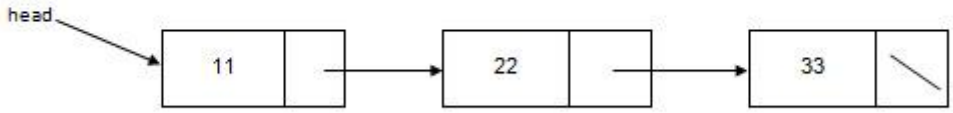
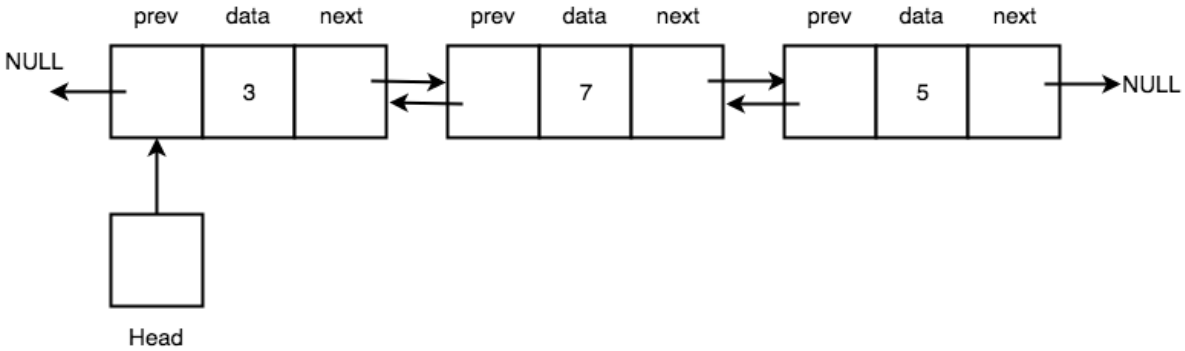


Figure 2: A singly linked list

Since a LinkedList acts as a dynamic array and we do not have to specify the size while creating it, the size of the list automatically increases when we dynamically add and remove items.

The LinkedList is implemented using the doubly linked list data structure. The main difference between a normal LinkedList and doubly LinkedList is that a doubly linked list contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

doubly LinkedList



Constructor summary

LinkedList linkL = new LinkedList();	Constructs an empty list
LinkedList (Collection < ? extends E > c)	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Methods summary

Modifier and Type	**Method and Description**
boolean	**add**(E e) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(Collection < ? extends E > c) Appends all of the elements in the specified collections to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean	**addAll** (int index, Collection < ? extends E > c) Insert all of the elements in the specified collection into this list, starting at the specified position.
void	**addFirst**(**E** e) Insert the specified element at the beginning of this list.
void	**clear**() Removes all of the elements from this list.
Object	**clone()** Returns a shallow copy of this LinkedList
boolean	**contains** (Object o) Returns true if this list contains the specified element
Iterator <E>	**descendingIterator()** Returns an iterator over the elements in this deque in reverse sequential order.
E	**element()** Retrieves, but does not remove, the head (first element) of this list.
E	**get** (int index) Returns the element at the specified position in this list.
E	**getFirst()** Returns the first element in this list.
E	**getLast()**

	Returns the last element in this list.
int	**indexOf** (Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
int	**lastIndexOf** (Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
ListIterator<E>	**listIterator (**int index**)** Returns a list – iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
Spliterator<E>	**spliterator** ( ) creates a **late-binding** and fail-fast **Spliterator** ober the elements in this list.
boolean	**offer(E** e) Adds the specified element as the tail (last element) of this list.
boolean	**offerFirst(E e)** Inserts the specified element at the front of this list.
boolean	**offerLast(E e)** Insert the specified element at the end of this list.
E	**peek()** Retrieves, but does not remove, the head (first element) of this list.
E	**poll()** Retrieves and removes the head (first element) of this list.
E	**pollFirst()** Retrieves and removes the head (first element) of this list or returns null if this list is empty.
E	**pop()** Pops an element from the stack represented by this list.
void	**push(E e)** Pushes an element onto the stack represented by this list.
E	**remove()** Retrieves and removes the head (firs element) of this list.
E	**size()** Returns the number of elements in this list.

E	**set (**int index, **E** element) Replaces the element at the specified position in this list with the specified element.
boolean	**removeFirstOccurrence**(**Object** o) Removes the first occurrence of the specified element in this list (when traversing the list from head to tail).

When is its use recommended?

- ArrayList provides constant time for search operation, so it's better to use ArrayList if searching is more frequent operation than add and remove operation.
- When we don't know the size of the array because is a dynamic array, so we don't have to specify the size while creating it.
- When we want to access randomly to the Array. ArrayList has 0(1)-time complexity to access elements via the get and set methods.
- When we work with data that duplicate values matters.