```
class DoublyLinkedBase:
    """A base class providing a doubly linked list representation."""

    class Node:
        """Lightweight, nonpublic class for storing a doubly linked
node."""
        __slots__ = 'element', 'prev', 'next'

        def __init__(self, element, prev, next):
            self.element = element
            self.prev = prev
            self.next = next

    def __init__(self):
        """Create an empty list."""
        self.header = self.Node(None, None, None)
        self.trailer = self.Node(None, None, None)
        self.header.next = self.trailer  # trailer is after header
        self.trailer.prev = self.header  # header is before trailer
        self.size = 0  # number of elements

    def __len__(self):
        """Return the number of elements in the list."""
        return self.size

    def is_empty(self):
        """Return True if list is empty."""
        return self.size == 0

    def insert_between(self, e, predecessor, successor):
        """Add element e between two existing nodes and return the new
node."""
        newest = self.Node(e, predecessor, successor)
        predecessor.next = newest
        successor.prev = newest
        self.size += 1
        return newest

    def delete_node(self, node):
        """Delete nonsentinel node from the list and return its
element."""
        predecessor = node.prev
        successor = node.next
        predecessor.next = successor
        successor.prev = predecessor
        self.size -= 1
        element = node.element  # record deleted element
        node.prev = node.next = node.element = None  # deprecate node
```

```
        return element


class LinkedDeque(DoublyLinkedBase):
    """Double-ended queue implementation based on a doubly linked
list."""

    def first(self):
        """Return (but do not remove) the element at the front of the
deque."""
        if self.is_empty():
            raise Exception("Deque is empty")
        return self.header.next.element  # real item just after header

    def last(self):
        """Return (but do not remove) the element at the back of the
deque."""
        if self.is_empty():
            raise Exception("Deque is empty")
        return self.trailer.prev.element  # real item just before
trailer

    def insert_first(self, e):
        """Add an element to the front of the deque."""
        self.insert_between(e, self.header, self.header.next)  # after
header

    def insert_last(self, e):
        """Add an element to the back of the deque."""
        self.insert_between(e, self.trailer.prev, self.trailer)  #
before trailer

    def delete_first(self):
        """Remove and return the element from the front of the
deque."""
        if self.is_empty():
            raise Exception("Deque is empty")
        return self.delete_node(self.header.next)  # use inherited
method

    def delete_last(self):
        """Remove and return the element from the back of the deque."""
        if self.is_empty():
            raise Exception("Deque is empty")
        return self.delete_node(self.trailer.prev)  # use inherited
method

    def print_deque(self):
        """Print all elements of the deque from front to back."""
```

```
        if self.is_empty():
            print("Deque is empty")
            return
        current = self.header.next
        elements = []
        while current != self.trailer:  # Traverse until the trailer
            elements.append(current.element)
            current = current.next
        print("Deque:", elements)
```

Input:

```
deque = LinkedDeque()  # LinkedDeque object is created
deque.insert_first(10)
deque.insert_first(5)
deque.insert_last(15)
deque.print_deque()
deque.delete_last()
deque.print_deque()
```

Output:

```
Deque: [5, 10, 15]
Deque: [5, 10]
```