```python
class Node:
    """Lightweight, non-public class for storing a doubly linked node."""

    __slots__ = _element, _prev, _next  # streamline memory

    def __init__(self, element, prev, next):  # initialize node’s fields
        self._element = element  # user’s element
        self._prev = prev  # previous node reference
        self._next = next  # next node reference
```

```python
from .doubly_linked_base import DoublyLinkedBase


class Empty(Exception):
    """Exception for requesting data from an empty collection"""
    pass


class LinkedDeque(DoublyLinkedBase):  # note the use of inheritance
    """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.

        Raise Empty exception if the deque is empty.
        """
        if self.is_empty():
            raise Empty("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.

        Raise Empty exception if the deque is empty.
        """
        if self.is_empty():
            raise Empty("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.

        Raise Empty exception if the deque is empty.
        """
        if self.is_empty():
            raise Empty("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.

        Raise Empty exception if the deque is empty.
        """
        if self.is_empty():
            raise Empty("Deque is empty")
        return self._delete_node(self._trailer._prev)  # use inherited method
```

```python
class Empty(Exception):
    """Exception for requesting data from an empty collection"""
    pass


class LinkedQueue:
    """FIFO queue implementation using a singly linked list for storage."""

    # ------------- nested _Node class -------------
    class _Node:
        """Lightweight, nonpublic class for storing a singly linked node."""
        __slots__ = '_element', '_next'  # streamline memory usage

        def __init__(self, element, next):
            self._element = element
            self._next = next

    # ------------------ queue methods ------------------
    def __init__(self):
        """Create an empty queue."""
        self._head = None
        self._tail = None
        self._size = 0  # number of queue elements

    def __len__(self):
        """Return the number of elements in the queue."""
        return self._size

    def is_empty(self):
        """Return True if the queue is empty."""
        return self._size == 0

    def first(self):
        """Return (but do not remove) the element at the front of the queue.

        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        return self._head._element  # front aligned with head of list

    def dequeue(self):
        """Remove and return the first element of the queue (i.e., FIFO).

        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        answer = self._head._element
        self._head = self._head._next
        self._size -= 1
        if self.is_empty():  # special case as queue is empty
            self._tail = None  # removed head had been the tail
        return answer

    def enqueue(self, e):
        """Add an element to the back of queue."""
        newest = self._Node(e, None)  # node will be new tail node
        if self.is_empty():
            self._head = newest  # special case: previously empty
        else:
            self._tail._next = newest
        self._tail = newest  # update reference to tail node
        self._size += 1
```

```python
class Empty(Exception):
    """Exception for requesting data from an empty collection"""
    pass


class LinkedStack:
    """LIFO Stack implementation using a singly linked list for storage."""

    # ------------- nested _Node class -------------
    class _Node:
        """Lightweight, nonpublic class for storing a singly linked node."""
        __slots__ = '_element', '_next'  # streamline memory usage

        def __init__(self, element, next):  # initialize node's fields
            self._element = element  # reference to user's element
            self._next = next  # reference to next node

    # ------------------ stack methods ------------------
    def __init__(self):
        """Create an empty stack."""
        self._head = None  # reference to the head node
        self._size = 0  # number of stack elements

    def __len__(self):
        """Return the number of elements in the stack."""
        return self._size

    def is_empty(self):
        """Return True if the stack is empty."""
        return self._size == 0

    def push(self, e):
        """Add element e to the top of the stack."""
        self._head = self._Node(e, self._head)  # create and link a new node
        self._size += 1

    def top(self):
        """Return (but do not remove) the element at the top of the stack.

        Raise Empty exception if the stack is empty.
        """
        if self.is_empty():
            raise Empty('Stack is empty')
        return self._head._element  # top of stack is at head of list

    def pop(self):
        """Remove and return the element from the top of the stack (i.e., LIFO).

        Raise Empty exception if the stack is empty.
        """
        if self.is_empty():
            raise Empty('Stack is empty')
        answer = self._head._element
        self._head = self._head._next  # bypass the former top node
        self._size -= 1
        return answer
```

```python
class Empty(Exception):
    """Exception for requesting data from an empty collection"""
    pass


class CircularQueue:
    """Queue implementation using circularly linked list for storage."""

    # ---------------------------------------------------------------------------
    # nested _Node class
    class _Node:
        """Lightweight, nonpublic class for storing a singly linked node."""
        __slots__ = '_element', '_next'  # streamline memory usage

        def __init__(self, element, next):
            self._element = element
            self._next = next

    # end of _Node class
    # ---------------------------------------------------------------------------

    def __init__(self):
        """Create an empty queue."""
        self._tail = None  # will represent tail of queue
        self._size = 0  # number of queue elements

    def __len__(self):
        """Return the number of elements in the queue."""
        return self._size

    def is_empty(self):
        """Return True if the queue is empty."""
        return self._size == 0

    def first(self):
        """Return (but do not remove) the element at the front of the queue.

        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        head = self._tail._next
        return head._element

    def dequeue(self):
        """Remove and return the first element of the queue (i.e., FIFO).

        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        oldhead = self._tail._next
        if self._size == 1:  # removing only element
            self._tail = None  # queue becomes empty
        else:
            self._tail._next = oldhead._next  # bypass the old head
        self._size -= 1
        return oldhead._element

    def enqueue(self, e):
        """Add an element to the back of queue."""
        newest = self._Node(e, None)  # node will be new tail node
        if self.is_empty():
            newest._next = newest  # initialize circularly
        else:
            newest._next = self._tail._next  # new node points to head
            self._tail._next = newest  # old tail points to new node
        self._tail = newest  # new node becomes the tail
        self._size += 1

    def rotate(self):
        """Rotate front element to the back of the queue."""
        if self._size > 0:
            self._tail = self._tail._next  # old head becomes new tail
```

```python
from .positional_list import PositionalList


class FavoritesList:
    """List of elements ordered from most frequently accessed to least."""

    # ------------------ nested _Item class ------------------
    class _Item:
        __slots__ = '_value', '_count'  # streamline memory usage

        def __init__(self, e):
            self._value = e  # the user's element
            self._count = 0  # access count initially zero

    # ------------------- nonpublic utilities -------------------
    def _find_position(self, e):
        """Search for element e and return its Position (or None if not found)."""
        walk = self._data.first()
        while walk is not None and walk.element()._value != e:
            walk = self._data.after(walk)
        return walk

    def _move_up(self, p):
        """Move item at Position p earlier in the list based on access count."""
        if p != self._data.first():  # consider moving...
            cnt = p.element()._count
            walk = self._data.before(p)
            if cnt > walk.element()._count:  # must shift forward
                while (walk != self._data.first() and
                               cnt > self._data.before(walk).element()._count):
                    walk = self._data.before(walk)
                self._data.add_before(walk, self._data.delete(p))  # delete/reinsert

    # ------------------- public methods -------------------
    def __init__(self):
        """Create an empty list of favorites."""
        self._data = PositionalList()  # will be list of _Item instances

    def __len__(self):
        """Return number of entries on favorites list."""
        return len(self._data)

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

    def access(self, e):
        """Access element e, thereby increasing its access count."""
        p = self._find_position(e)  # try to locate existing element
        if p is None:
            p = self._data.add_last(self._Item(e))  # if new, place at end
        p.element()._count += 1  # always increment count
        self._move_up(p)  # consider moving forward

    def remove(self, e):
        """Remove element e from the list of favorites."""
        p = self._find_position(e)  # try to locate existing element
        if p is not None:
            self._data.delete(p)  # delete, if found

    def top(self, k):
        """Generate sequence of top k elements in terms of access count."""
        if not 1 <= k <= len(self):
            raise ValueError('Illegal value for k')
        walk = self._data.first()
        for j in range(k):
            item = walk.element()  # element of list is _Item
            yield item._value  # report user's element
            walk = self._data.after(walk)

    def __repr__(self):
        """Create string representation of the favorites list."""
        return ', '.join('({0}:{1})'.format(i._value, i._count) for i in self._data)


if __name__ == '__main__':
```

```
    fav = FavoritesList()
    for c in 'hello. this is a test of mtf':  # well, not the mtf part...
        fav.access(c)
        k = min(5, len(fav))
        print('Top {0}) {1:25} {2}'.format(k, [x for x in fav.top(k)], fav))
```

```python
from .doubly_linked_base import DoublyLinkedBase


class PositionalList(DoublyLinkedBase):
    """A sequential container of elements allowing positional access."""

    # ------------- nested Position class -------------
    class Position:
        """An abstraction representing the location of a single element.

        Note that two position instaces may represent the same inherent
        location in the list.  Therefore, users should always rely on
        syntax 'p == q' rather than 'p is q' when testing equivalence of
        positions.
        """

        def __init__(self, container, node):
            """Constructor should not be invoked by user."""
            self._container = container
            self._node = node

        def element(self):
            """Return the element stored at this Position."""
            return self._node._element

        def __eq__(self, other):
            """Return True if other is a Position representing the same location."""
            return type(other) is type(self) and other._node is self._node

        def __ne__(self, other):
            """Return True if other does not represent the same location."""
            return not (self == other)  # opposite of __eq__

    # ------------------ utility methods ------------------
    def _validate(self, p):
        """Return position's node, or raise appropriate error if invalid."""
        if not isinstance(p, self.Position):
            raise TypeError('p must be proper Position type')
        if p._container is not self:
            raise ValueError('p does not belong to this container')
        if p._node._next is None:  # convention for deprecated nodes
            raise ValueError('p is no longer valid')
        return p._node

    def _make_position(self, node):
        """Return Position instance for given node (or None if sentinel)."""
        if node is self._header or node is self._trailer:
            return None  # boundary violation
        else:
            return self.Position(self, node)  # legitimate position

    # ------------------ accessors ------------------
    def first(self):
        """Return the first Position in the list (or None if list is empty)."""
        return self._make_position(self._header._next)

    def last(self):
        """Return the last Position in the list (or None if list is empty)."""
        return self._make_position(self._trailer._prev)

    def before(self, p):
        """Return the Position just before Position p (or None if p is first)."""
        node = self._validate(p)
        return self._make_position(node._prev)

    def after(self, p):
        """Return the Position just after Position p (or None if p is last)."""
        node = self._validate(p)
        return self._make_position(node._next)

    def __iter__(self):
        """Generate a forward iteration of the elements of the list."""
        cursor = self.first()
        while cursor is not None:
            yield cursor.element()
            cursor = self.after(cursor)
```

```
    # ------------------ mutators ------------------
    # override inherited version to return Position, rather than Node
    def _insert_between(self, e, predecessor, successor):
        """Add element between existing nodes and return new Position."""
        node = super()._insert_between(e, predecessor, successor)
        return self._make_position(node)

    def add_first(self, e):
        """Insert element e at the front of the list and return new Position."""
        return self._insert_between(e, self._header, self._header._next)

    def add_last(self, e):
        """Insert element e at the back of the list and return new Position."""
        return self._insert_between(e, self._trailer._prev, self._trailer)

    def add_before(self, p, e):
        """Insert element e into list before Position p and return new Position."""
        original = self._validate(p)
        return self._insert_between(e, original._prev, original)

    def add_after(self, p, e):
        """Insert element e into list after Position p and return new Position."""
        original = self._validate(p)
        return self._insert_between(e, original, original._next)

    def delete(self, p):
        """Remove and return the element at Position p."""
        original = self._validate(p)
        return self._delete_node(original)  # inherited method returns element

    def replace(self, p, e):
        """Replace the element at Position p with e.

        Return the element formerly at Position p.
        """
        original = self._validate(p)
        old_value = original._element  # temporarily store old element
        original._element = e  # replace with new element
        return old_value  # return the old element value
```

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

    # ------------- nested _Node class -------------
    # nested _Node class
    class _Node:
        """Lightweight, nonpublic class for storing a doubly linked node."""
        __slots__ = '_element', '_prev', '_next'  # streamline memory

        def __init__(self, element, prev, next):  # initialize node's fields
            self._element = element  # user's element
            self._prev = prev  # previous node reference
            self._next = next  # next node reference

    # ------------- list constructor -------------

    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

    # ------------- public accessors -------------

    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

    # ------------- nonpublic utilities -------------

    def _insert_between(self, e, predecessor, successor):
        """Add element e between two existing nodes and return new node."""
        newest = self._Node(e, predecessor, successor)  # linked to neighbors
        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  # return deleted element
```

```python
from .favorites_list import FavoritesList
from .positional_list import PositionalList


class FavoritesListMTF(FavoritesList):
    """List of elements ordered with move-to-front heuristic."""

    # we override _move_up to provide move-to-front semantics
    def _move_up(self, p):
        """Move accessed item at Position p to front of list."""
        if p != self._data.first():
            self._data.add_first(self._data.delete(p))  # delete/reinsert

    # we override top because list is no longer sorted
    def top(self, k):
        """Generate sequence of top k elements in terms of access count."""
        if not 1 <= k <= len(self):
            raise ValueError('Illegal value for k')

        # we begin by making a copy of the original list
        temp = PositionalList()
        for item in self._data:  # positional lists support iteration
            temp.add_last(item)

        # we repeatedly find, report, and remove element with largest count
        for j in range(k):
            # find and report next highest from temp
            highPos = temp.first()
            walk = temp.after(highPos)
            while walk is not None:
                if walk.element()._count > highPos.element()._count:
                    highPos = walk
                walk = temp.after(walk)
            # we have found the element with highest count
            yield highPos.element()._value  # report element to user
            temp.delete(highPos)  # remove from temp list


if __name__ == '__main__':
    fav = FavoritesListMTF()
    for c in 'hello. this is a test of mtf':
        fav.access(c)
        k = min(5, len(fav))
        print('Top {0}) {1:25} {2}'.format(k, [x for x in fav.top(k)], fav))
```

```python
"""Sort PositionalList of comparable elements into non-decreasing order."""


def insertion_sort(L):
    if len(L) > 1:  # otherwise, no need to sort it
        marker = L.first()
        while marker != L.last():
            pivot = L.after(marker)  # next item to place
            value = pivot.element()
            if value > marker.element():  # pivot is already sorted
                marker = pivot  # pivot becomes new marker
            else:  # must relocate pivot
                walk = marker  # find leftmost item greater than value
                while walk != L.first() and L.before(walk).element() > value:
                    walk = L.before(walk)
                L.delete(pivot)
                L.add_before(walk, value)  # reinsert value before walk
```