Algorithms II

PhD. Mohammed-Amine KOULALI
amine.koulali@um6p.ma

UM6P-CS
Mohammed VI Polytechnic University

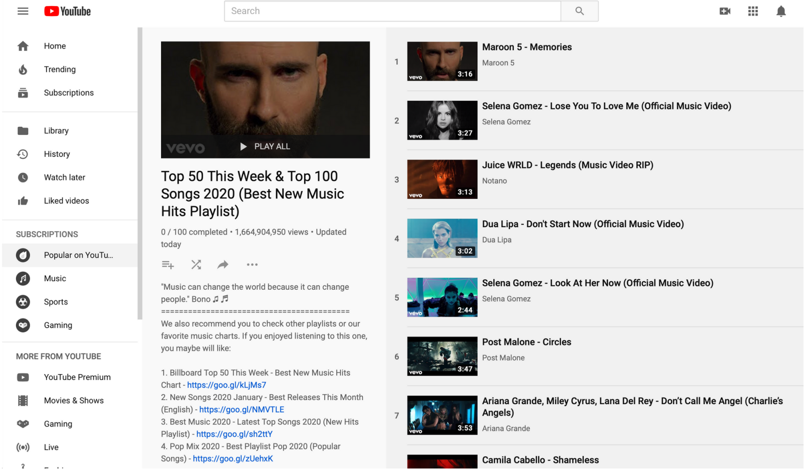
16 april 2022

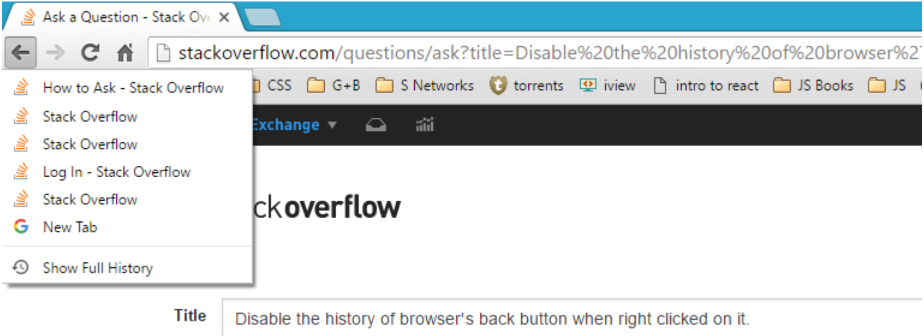
Table of Contents

Linked Lists

Applications

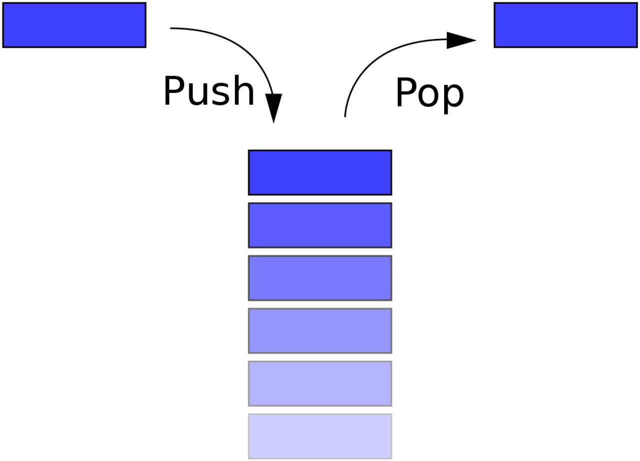
Your Playlist

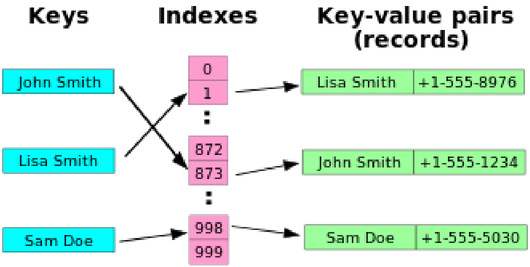




Ask a Question - Stack Ov
stackoverflow.com/questions/ask?title=Disable%20the%20history%20of%20browser%2
How to Ask - Stack Overflow
Stack Overflow
Stack Overflow
Log In - Stack Overflow
Stack Overflow
New Tab
Show Full History
CSS
G+B
S Networks
torrents
iview
intro to react
JS Books
JS
Exchange
ckoverflow
Title
Disable the history of browser's back button when right clicked on it.

Applications
Stacks





Keys
Indexes
Key-value pairs (records)
John Smith
Lisa Smith
Sam Doe
0
1
872
873
998
999
Lisa Smith
+1-555-8976
John Smith
+1-555-1234
Sam Doe
+1-555-5030

Python's list class

Lines 1–15 / 14

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.index('banana', 4)
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Python's list class

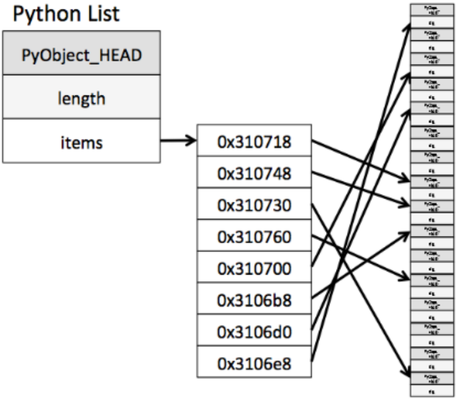
Memory Allocation

- A contiguous array of references to other objects is used.
- Python keeps a pointer to this array and the array's length is stored in a list head structure.
- When items are appended or inserted the array of references is resized.

UNIVERSITY MOHAMMED VI POLYTECHNIC

Python's list class

Memory Allocation



Python's list class

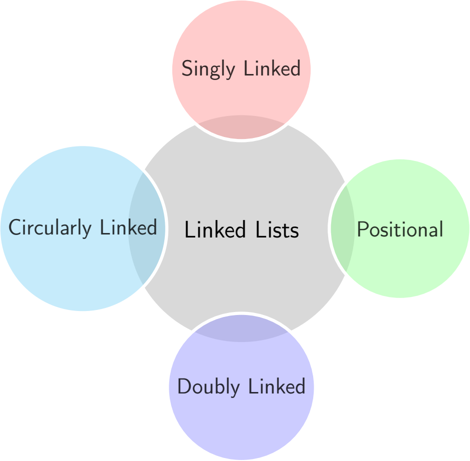
- Pros:
  - Highly optimized.
  - Off-the-shelf builtin methods: *pop*, *sort*, *append*,. . .
- Cons:
  - Length typically larger than the number of elements immediately required.
  - Operations time complexity may be unacceptable in real-time systems.
  - Insertions and deletions at interior positions of an array are expensive.

Definition

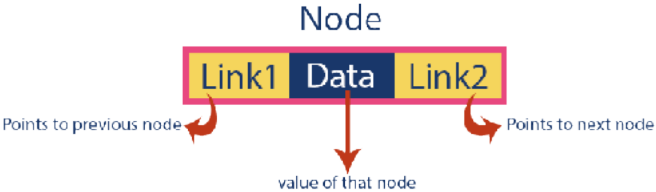
A linked list is a linear data structure stored randomly in memory and is made up of nodes that contain a value and pointers.

Linked Lists

Flavors



Building blocks
Nodes



When to use Linked Lists

- When your task will frequently insert items in its list.
- Searching is the area where linked lists aren't so great.
- Deletion method is not a highlight for Linked Lists.

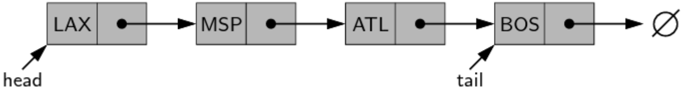
Singly Linked Lists
Definition

Definition

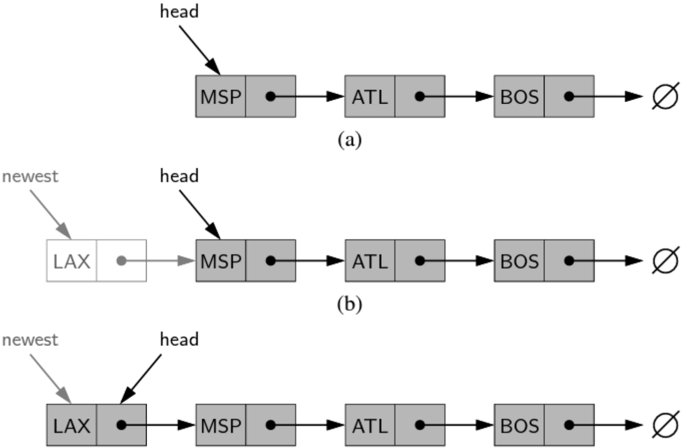
The singly linked list is a collection of nodes that collectively form a linear sequence. Each node stores a reference to an object that is an element of the sequence and a reference to the next node of the list.

- The first and last nodes of a linked list are known as the head and tail of the list, respectively.

Singly Linked List



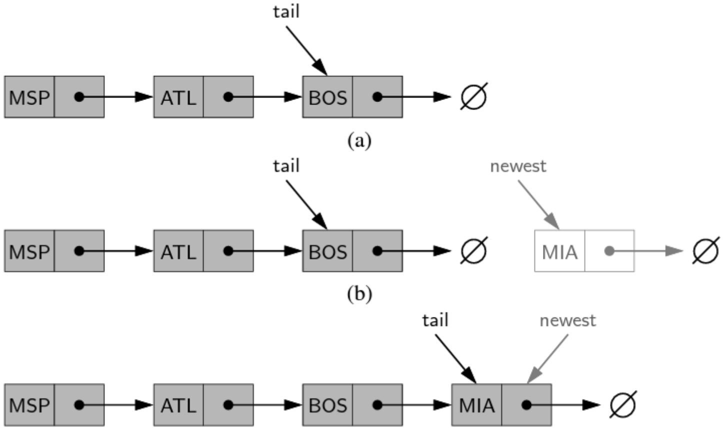
Insertion at the head



Insertion
At the head

```
Algorithm addFirst(L,e):
    newest = Node(e)
    newest.next = L.head
    L.head = newest
    L.size = L.size+1
```

Insertion at the tail



Insertion

At the tail

```
Algorithm addLast(L,e):
    newest = Node(e)
    newest.next = None
    L.tail.next = newest
    L.tail = newest
    L.size = L.size+1
```

UNIVERSITY MOHAMMED VI POLYTECHNIC

Removal

At the tail

```
Algorithm removeFirst(L):
    if L.head is None then
        Indicate an error: the list is_empty.
    L.head = L.head.next
    L.size = L.size-1
```

- We cannot easily delete the last node of a singly linked list

Stacks as Linked Lists

Lines 1–18 / 37

```python
class LinkedStack:
    """LIFO Stack implementatino using a singly linked list for storage"""

    class _Node:
        __slots__ = '_element', '_next'

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


    def __init__(self):
        self._head = None
        self._size = 0

    def __len__(self):
        return self._size
```

Stacks as Linked Lists

Lines 14–31 / 37

```python
        self._size = 0

    def __len__(self):
        return self._size

    def is_empty(self):
        return self._size == 0

    def push(self, e):
        self._head = self._Node(e, self._head)
        self._size += 1

    def top(self):
        if self.is_empty():
            raise Empty('Stack is empty')
        return self._head._element

    def pop(self):
```

Stacks as Linked Lists

Lines 27–44 / 37

```python
        if self.is_empty():
            raise Empty('Stack is empty')
        return self._head._element

    def pop(self):
        if self.is_empty():
            raise Empty('Stack is empty')
        answer = self._head._element
        self._head = self._head._next
        self._size -= 1
        return answer
```

Stacks as Linked Lists

Time complexity

```
S = LinkedStack
```

Operation	Complexity
S.push(e)	O(1)
S.pop()	O(1)
S.top()	O(1)
len(S)	O(1)
S.is_empty()	O(1)

Queues as Linked Lists

Lines 1–15 / 38

```python
class LinkedQueue:
  """FIFO queue implementation using a singly linked list for storage."""
  class _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
    def __init__(self):
      self._head = None
      self._tail = None
      self._size = 0 # number of queue elements
    def __len__(self):
      return self._size
    def is_empty(self):
      return self._size == 0
```

Queues as Linked Lists

Lines 14–28 / 38

```python
    def is_empty(self):
      return self._size == 0
    def first(self):
      if self.is_empty():
        raise Empty('Queue is empty')
      return self._head._element # front aligned with head of list

    def dequeue(self):
      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
```

Queues as Linked Lists

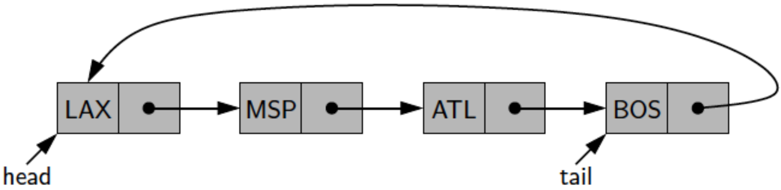
Lines 27–41 / 38

```python
        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):
        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
```

- A circularly linked list provides a more general model than a singly linked list.
- List for data sets that are cyclic:
  - Which do not have any particular notion of a beginning and end.
- We must maintain a reference to a particular node (current) in order to make use of the list.

Circularly Linked Lists



Queues as Circularly Linked Lists

Lines 1–13 / 42

```python
class CircularQueue:
  class _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
    def __init__(self):
      self._tail = None # will represent tail of queue
      self._size = 0 # number of queue elements
    def __len__(self):
      return self._size
    def is_empty(self):
      return self._size == 0
```

Queues as Circularly Linked Lists

Lines 14–26 / 42

```python
    def first(self):
        if self.is_empty():
            raise Empty('Queue is empty')
        head = self._tail._next
        return head._element
    def dequeue(self):
        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:
```

UNIVERSITY MOHAMMED VI POLYTECHNIC

```
            self._tail._next = oldhead._next # bypass the old head
        self._size-= 1
        return oldhead._element
    def enqueue(self, e):
        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
```

Queues as Circularly Linked Lists

Lines 40–52 / 42

```python
    def rotate(self):
        if self._size > 0:
            self._tail = self._tail._next # old head
```

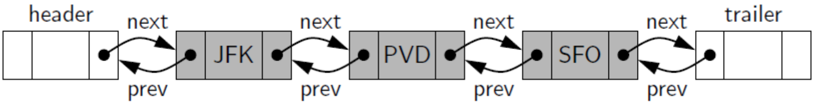
Doubly Linked Lists

- We can efficiently insert a node either at the head or end of a singly linked list,
- We can delete a node at the head of a list,
- We cannot efficiently delete an arbitrary node from an interior position of the list if only given a reference to that node.
- Solution: use nodes that point to their predecessor and successor.

- To avoid some special cases when operating near the boundaries of a doubly-linked list, it helps to add special nodes at both ends of the list.
- Header node at the beginning of the list, and trailer node at the end of the list.
- These nodes are known as sentinels (or guards), and they do not store elements of the primary sequence.

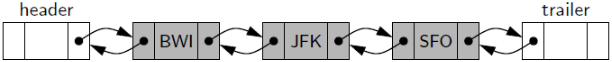
Doubly Linked Lists

Sentinels

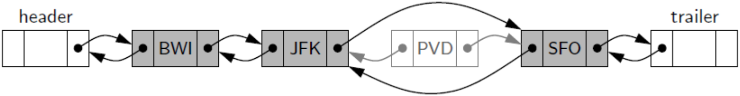


- The header and trailer nodes never change-only. The nodes between them change.
- We can treat all insertions in a unified manner because a new node will always be placed between a pair of existing nodes.
- Every element to be deleted is guaranteed to be stored in a node with neighbors on each side.

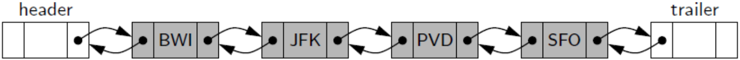
Doubly Linked Lists
Adding a node



(a)

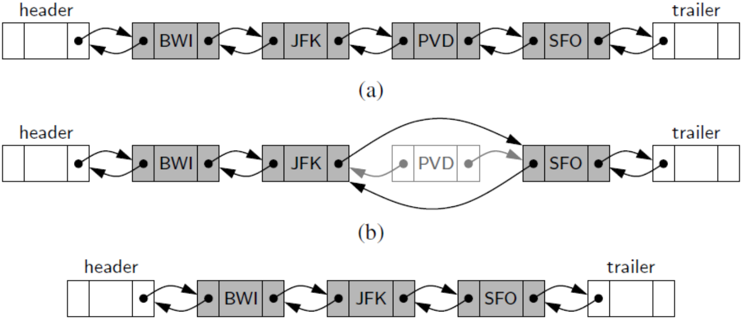


(b)



Doubly Linked Lists

Removal of a node



Doubly Linked Lists

Lines 1–13 / 37

```python
class _DoublyLinkedBase:
    class _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

    def __init__(self):
        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
```

Doubly Linked Lists

Lines 14–26 / 37

```python
        self._size = 0 # number of elements

    def __len__ (self):
        return self.size

    def is_empty(self):
        return self.size == 0

    def _insert_between(self, e, predecessor, successor):
        newest = self._Node(e, predecessor, successor)
        predecessor._next = newest
        successor._prev = newest
        self._size += 1
```

Doubly Linked Lists

Lines 27–39 / 37

```python
        return newest

    def _delete_node(self, node):
        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
```

UNIVERSITY MOHAMMED VI POLYTECHNIC

Double-Ended Queue

Lines 1–13 / 26

```
class LinkedDeque(_DoublyLinkedBase): # note the use of inheritance
    def first(self):
        if self.is_empty( ):
            raise Empty("Deque is empty")
        return self._header._next._element # real item just after header

    def last(self):
        if self.is_empty( ):
            raise Empty("Deque is empty")
        return self._trailer._prev._element # real item just before trailer

    def insert_first(self, e):
        self._insert_between(e, self._header, self._header._next) # after header
```

Double-Ended Queue

Lines 14–26 / 26

```python
    def insert_last(self, e):
        self._insert_between(e, self._trailer._prev, self._trailer) # before trailer

    def delete_first(self):
        if self.is_empty( ):
            raise Empty("Deque is empty")
        return self._delete_node(self._header._next) # use inherited method

    def delete_last(self):
        if self.is_empty( ):
            raise Empty("Deque is empty")
        return self._delete_node(self._trailer._prev) # use inherited method
```

Positional List

- We would like to design an abstract data type that provides a user a way to refer to elements anywhere in a sequence, and to perform arbitrary insertions and deletions.
- Indices are not a good abstraction for describing a local position in some applications (changes over time due to insertions or deletions).
- We prefer an abstraction, in which there is some other means for describing a position.
- Instead of relying directly on nodes, we introduce an independent position abstraction to denote the location of an element within a list.
- Our objective: Each method of the positional list ADT runs in worst-case $O(1)$ time when implemented with a doubly linked list.

Positional List

Lines 1–13 / 77

```python
class PositionalList(_DoublyLinkedBase):
    class _Position:
        def __init__ (self, container, node):
            self._container = container
            self._node = node

        def element(self):
            return self._node._element

        def __eq__(self, other):
            return type(other) is type(self) and other.__node is self.__node

        def __ne__ (self, other):
```

Positional List

Lines 14–26 / 77

```python
        return not (self == other) # opposite of eq

    def _validate(self, p):
        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):
        if node is self._header or node is self._trailer:
```

Positional List

Lines 27–39 / 77

```python
            return None # boundary violation
        else:
            return self._Position(self, node) # legitimate position

    def first(self):
        return self._make_position(self._header._next)

    def last(self):
        return self._make_position(self._trailer._prev)

    def before(self, p):
        node = self._validate(p)
        return self._make_position(node._prev)
```

Positional List

Lines 40–52 / 77

```python
    def after(self, p):
        node = self._validate(p)
        return self._make_position(node._next)

    def __iter__(self):
        cursor = self.first()
        while cursor is not None:
            yield cursor.element()
            cursor = self.after(cursor)

    def _insert_between(self, e, predecessor, successor):
        node = super()._insert_between(e, predecessor, successor)
```

```
            return self._make_position(node)

        def add_first(self, e):
            return self._insert_between(e, self._header, self._header._next)

        def add_last(self, e):
            return self._insert_between(e, self._trailer._prev, self._trailer)

        def add_before(self, p, e):
            original = self. validate(p)
            return self._insert_between(e, original._prev, original)

        def add_after(self, p, e):
```

Positional List

Lines 66–78 / 77

```python
        original = self._validate(p)
        return self._insert_between(e, original, original._next)

    def delete(self, p):
        original = self._validate(p)
        return self._delete_node(original) # inherited method returns element

    def replace(self, p, e):
        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
```

```
def insertion sort(L):
    """ort PositionalList of comparable elements into nondecreasing order."""
    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
```

Case study

- We consider maintaining a collection of elements while keeping track of the number of times each element is accessed.
- We would like to know which elements are among the most popular.
- Examples:
  - Web browser that keeps track of a user's most accessed URLs,
  - Music collection that maintains a list of the most frequently played songs for a user.

UNIVERSITY MOHAMMED VI POLYTECHNIC

Case study

Supported methods

- access(e): Access the element, incrementing its access count, and adding it to the favorites list if it is not already present.
- remove(e): Remove element e from the favorites list, if present.
- top(k): Return an iteration of the k most accessed elements.

Case study

Lines 1–15 / 62

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

    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
```

```
        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
    def __init__ (self):
    """Create an empty list of favorites."""
        self._data = PositionalList( ) # will be list of Item instances
```

```
        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
```

Case study

Lines 40–54 / 62

```python
        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')
```

Case study

Lines 53–67 / 62

```python
        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)
```

Arrays/Link-based Sequences

Pros

- Arrays provide $O(1)$-time access to an element based on an integer index.
- Array-based representations typically use proportionally less memory than linked structures
- Link-based structures support $O(1)$-time insertions and deletions at arbitrary positions.

Thanks for your Attention!