

Lecture 2: Sequences & List

Pure Knowledge Has
Infinite Organizing Power

Wholeness Statement

The Sequence ADT is the most general data structure and can be used in place of a Stack, Queue, or sometimes a List. Knowledge of the pros and cons of these data structures allows us to properly organize the most efficient and useful algorithmic solution to a specific problem.

Science of Consciousness: Pure knowledge has infinite organizing power, and administers the whole universe with minimum effort.

Sequence ADT

Outline

- ADT
- Sequence ADT
- Implementation of the Sequence ADT for Array and LinkedList
- Iterators

ADT(Abstract Data Type)

- Problem solving with a computer means processing data. To process data, we need to define the data type and the operation to be performed on the data.
- The definition of the data type and the definition of the operation to be applied to the data is part of the idea behind an abstract data type (ADT)—to hide how the operation is performed on the data.
- In other words, the user of an ADT needs only to know that a set of operations are available for the data type but does not need to know how they are applied.

The Sequence ADT

- A Sequence ADT stores a sequence of elements
- Element access is based on the concept of Rank(index)
 - Rank is the number of elements that precede an element in the sequence
- An element can be accessed, inserted, or removed by specifying its rank
- An exception is thrown if an incorrect rank is specified (e.g., a negative rank)

Random Access operations of Sequences are based on Rank

elemAtRank(r):

- returns the element at rank r without removing it

replaceAtRank(r, e):

- replace the element at rank r with e and return the old element

insertAtRank(r, e):

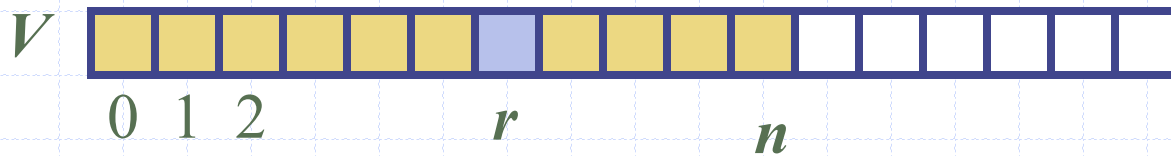
- insert a new element e to have rank r

removeAtRank(r):

- removes and returns the element at rank r

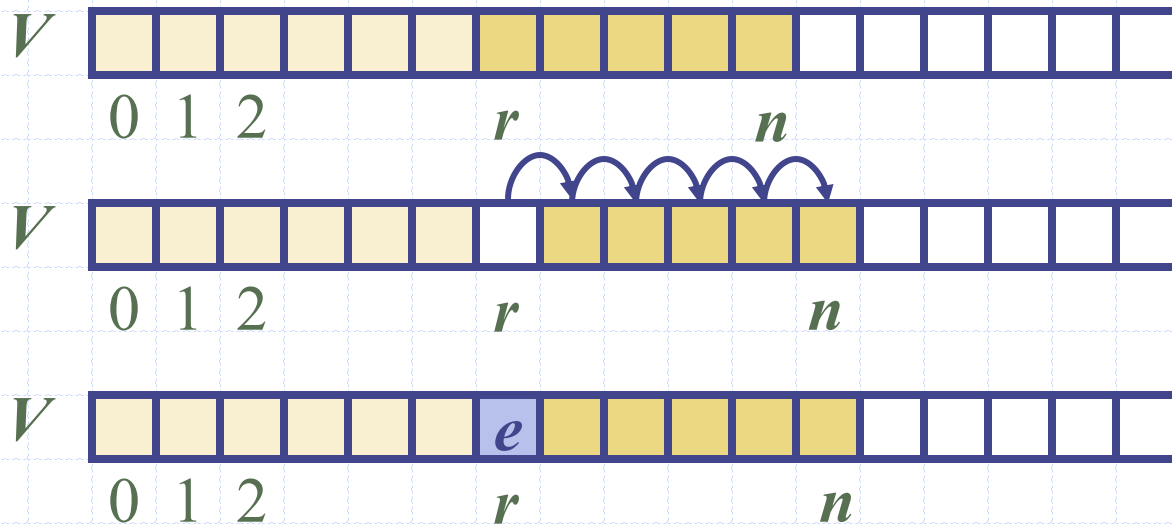
The Sequence is Array-based

- Use an array V of size N
- A variable n keeps track of the size of the array (number of elements stored)
- Operation *elemAtRank*(r) is implemented in $O(1)$ time by returning $V[r]$



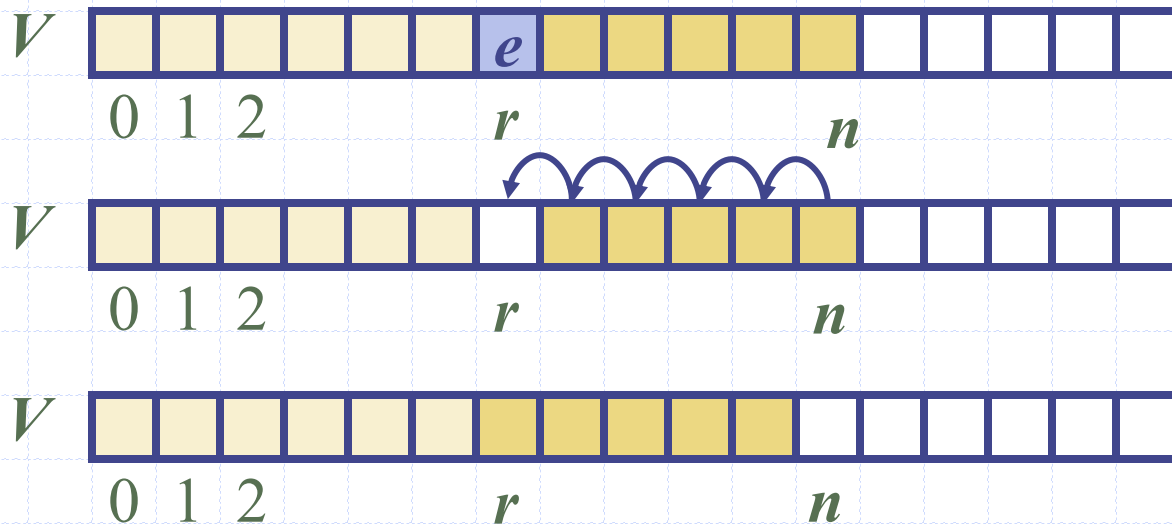
Insertion

- In operation *insertAtRank*(r, e), we need to make room for the new element by shifting forward the $n - r$ elements $V[r], \dots, V[n - 1]$
- In the worst case ($r = 0$), this takes $O(n)$ time



Deletion

- In operation *removeAtRank*(r), we need to fill the hole left by the removed element by shifting backward the $n - r - 1$ elements $V[r + 1], \dots, V[n - 1]$
- In the worst case ($r = 0$), this takes $O(n)$ time



Performance

- In the array-based implementation of a Sequence
 - The space used by the data structure is $O(N)$
 - *size*, *isEmpty*, *elemAtRank* and *replaceAtRank* run in $O(1)$ time
 - *insertAtRank* and *removeAtRank* run in $O(n)$ time

Sequence ADT

(also allows access via Position)

- The **Sequence** ADT is the union of rank-based and position-based methods
- Elements accessed by
 - Rank, or
 - Position
- Generic methods:
 - **size()**, **isEmpty()**
- Rank-based methods:
 - **elemAtRank(r)**,
replaceAtRank(r, o),
insertAtRank(r, o),
removeAtRank(r)
- Position-based (List) methods: p-position
 - **first()**, **last()**,
 - **before(p)**, **after(p)**,
replaceElement(p, o),
swapElements(p, q),
insertBefore(p, o),
insertAfter(p, o),
insertFirst(o),
insertLast(o),
remove(p)
- Bridge methods:
 - **atRank(r)**, **rankOf(p)**

Main Point

1. The Sequence ADT captures the abstract notion of a mathematical sequence; it specifies the operations that any container of elements with random access and sequential access should support.

Science of Consciousness: Likewise, pure awareness is an abstraction of individual awareness; each individual provides a specific, concrete realization of unbounded, unmoving pure awareness.

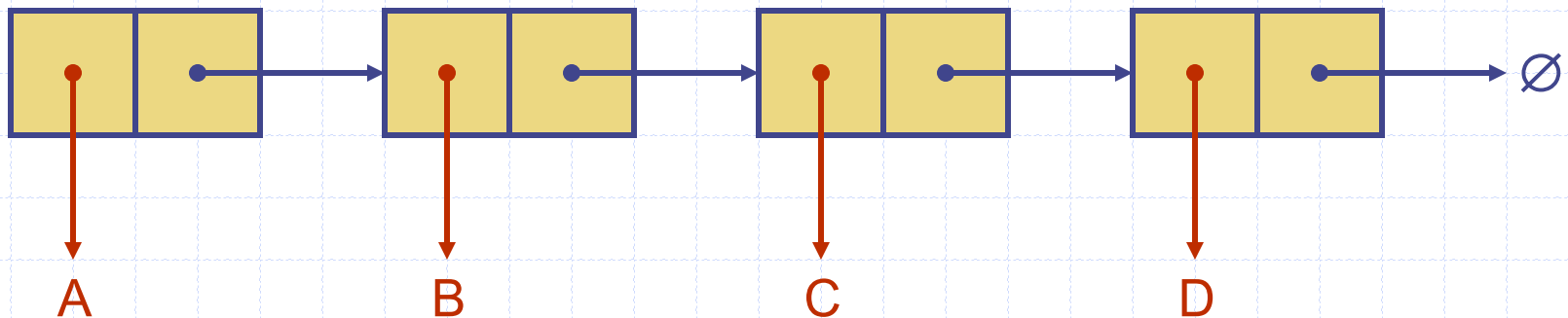
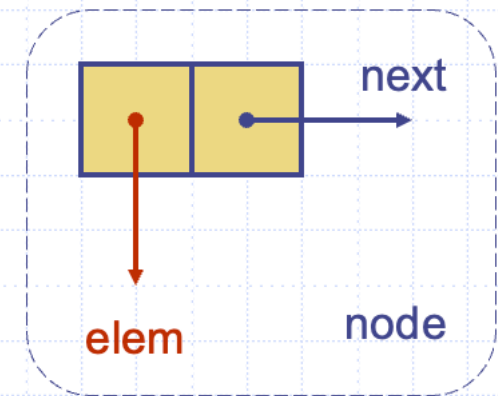
Linked List

- Motivation:
 - need to handle varying amounts of data
 - eliminate the need to resize the array
 - grows and shrinks exactly when necessary
 - efficient handling of insertion or removal from the middle of the data structure
 - random access is often unnecessary
- Built-in list data structures
 - Lisp, Scheme, ML, Haskell

Singly Linked List(SLL)

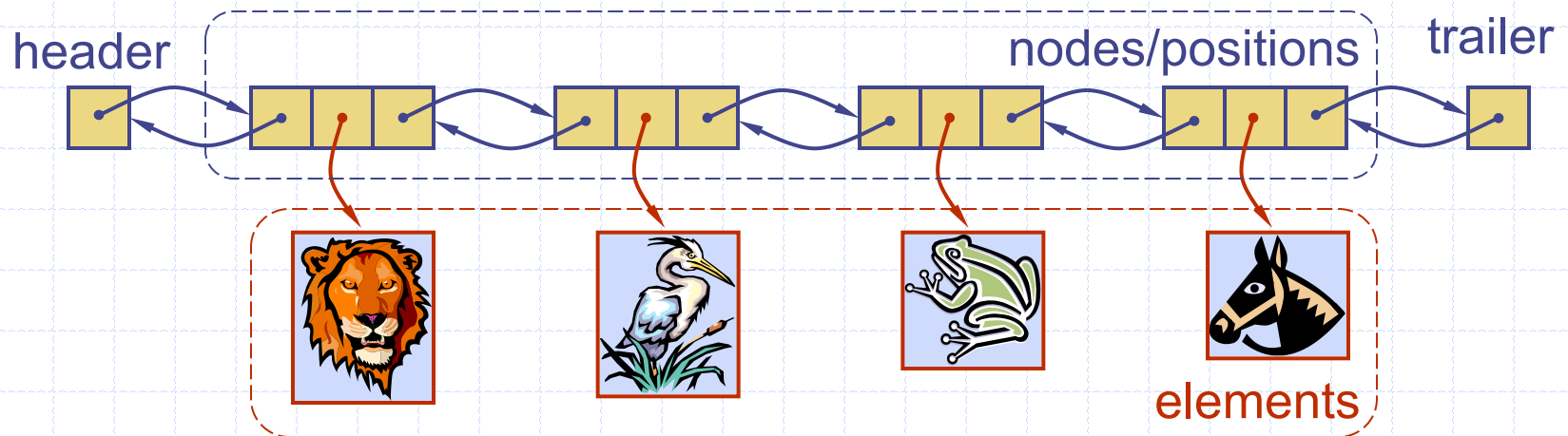
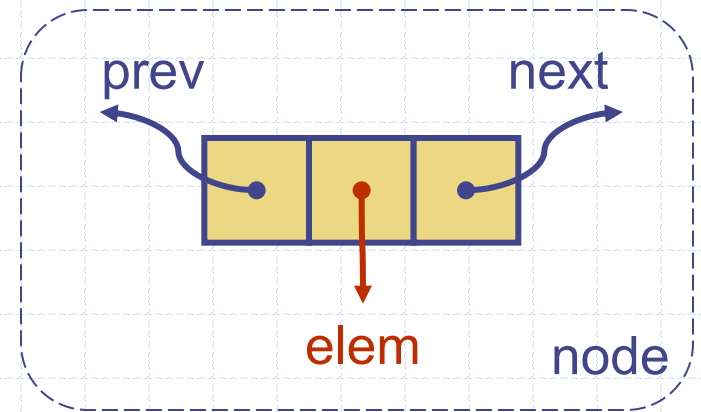
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
 - element
 - link to the next node

```
class Node {  
    constructor(d) {  
        this.elem = d;  
        this.next = null;  
    }  
}
```



Doubly Linked List(DLL)

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
 - element
 - link to the previous node
 - link to the next node
- Special header and trailer nodes

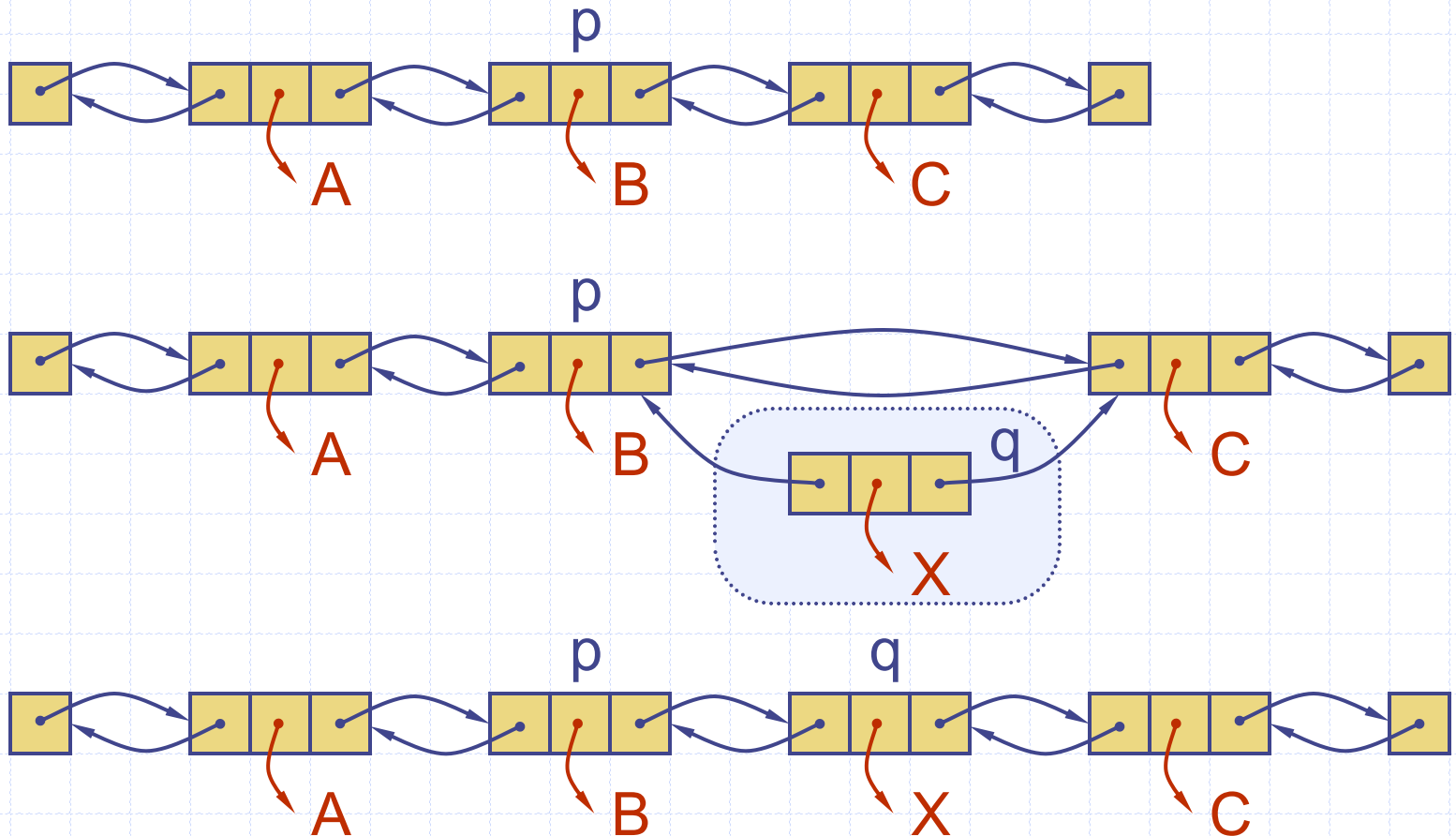


JavaScript Position ADT

```
class NPos {  
    #elem;  
    #prev;  
    #next;  
    constructor (elem, prev, next) {  
        this.#elem = elem;  
        this.#prev = prev;  
        this.#next = next;  
    }  
    element() {  
        return this.#elem;  
    }  
}
```

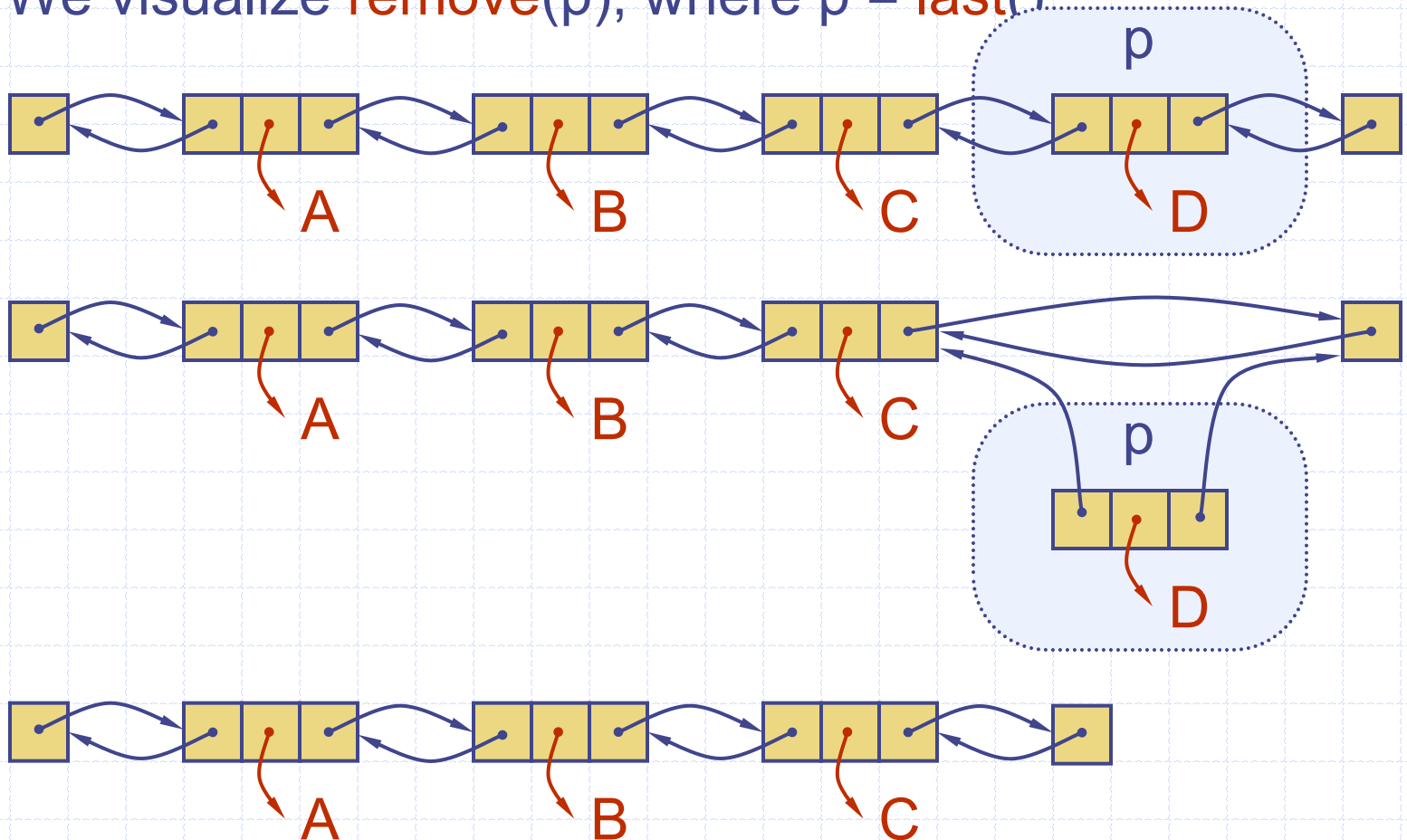
Insertion

- We visualize operation `insertAfter(p, X)`, which returns position `q`



Deletion

- We visualize `remove(p)`, where $p = \text{last}()$



List ADT for Linked List


- Generic methods:
 - `size()`, `isEmpty()`
- Query methods:
 - `isFirst(p)`, `isLast(p)`
- Accessor methods:
 - `first()`, `last()`
 - `before(p)`, `after(p)`
- Update methods:
 - `replaceElement(p, e)`,
 - `swapElements(p, q)`
 - `insertBefore(p, e)`,
`insertAfter(p, e)`,
 - `insertFirst(e)`,
`insertLast(e)`
 - `remove(p)`



Performance

- In the implementation of the List ADT by means of a doubly linked list
 - The space used by a list with n elements is $O(n)$
 - The space used by each position of the list is $O(1)$
 - All the operations of the List ADT run in $O(1)$ time
 - Operation `element()` of the Position ADT runs in $O(1)$ time

Exercise on List ADT for DLL

- Generic methods:
 - integer **size()**
 - boolean **isEmpty()**
 - objectIterator **elements()**
- Accessor methods:
 - position **first()**
 - position **last()**
 - position **before(p)**
 - position **after(p)**
- Query methods: 
 - boolean **isFirst(p)**
 - boolean **isLast(p)**
- Update methods:
 - **swapElements(p, q)**
 - object **replaceElement(p, o)**
 - **insertFirst(o)**
 - **insertLast(o)**
 - **insertBefore(p, o)**
 - **insertAfter(p, o)**
 - **remove(p)**

Exercise:

- Write a method to calculate the sum of the integers in a list of integers
 - Only use the methods in the list to the left side.

Algorithm **sum(L)**

Exercise on List ADT for DLL

- Generic methods:
 - integer **size()**
 - boolean **isEmpty()**
 - objectIterator **elements()**
- Accessor methods:
 - position **first()**
 - position **last()**
 - position **before(p)**
 - position **after(p)**
- Query methods:
 - boolean **isFirst(p)**
 - boolean **isLast(p)**
- Update methods:
 - **swapElements(p, q)**
 - object **replaceElement(p, o)**
 - **insertFirst(o)**
 - **insertLast(o)**
 - **insertBefore(p, o)**
 - **insertAfter(p, o)**
 - **remove(p)**

Exercise:

- Given a List L, write a method to find the **Position** that occurs in the middle of L
 - Specifically, find middle as follows:
 - when the number of elements is odd return the position p in L such that the same number of nodes occur before and after p
 - when number of elements is even find p such that there is one more element that occurs before p than after
 - Do this without using a counter of any kind
- Algorithm findMiddle(L)

Exercise on List ADT

- Generic methods:
 - integer `size()`
 - boolean `isEmpty()`
 - objectIterator `elements()`
- Accessor methods:
 - position `first()`
 - position `last()`
 - position `before(p)`
 - position `after(p)`
- Query methods:
 - boolean `isFirst(p)`
 - boolean `isLast(p)`
- Update methods:
 - `swapElements(p, q)`
 - object `replaceElement(p, o)`
 - `insertFirst(o)`
 - `insertLast(o)`
 - `insertBefore(p, o)`
 - `insertAfter(p, o)`
 - `remove(p)`

Exercise:

- Given a List L, write a method to remove the element that occurs at the middle of L
 - Specifically, remove as follows:
 - when the number of elements is odd, remove the element e such that the same number of elements occur before and after e in L
 - when the number of elements is even, remove element e such that there is one more element that occurs after e than before
 - Return the element e that was removed; implement this without using a counter of any kind
 - Analyze the complexity of your solution
- Algorithm `removeMiddle(L)`

Iterators

- An iterator abstracts the process of scanning through a collection of elements
- Methods of the *ObjectIterator* ADT:
 - boolean **hasNext()**
 - object **nextObject()**
- Extends the concept of Position by adding a traversal capability
- Implementation with an array or singly linked list
- An iterator is typically associated with another data structure
- We can augment the Stack, Queue, and List ADTs with method:
 - ObjectIterator **elements()**
- Two notions of iterator:
 - snapshot: freezes the contents of the data structure at a given time
 - dynamic: follows changes to the data structure

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. The List and Sequence ADTs may be used as an all-purpose class for storing collections of objects with both *random* and *sequential* access to its elements.
2. The underlying implementation of an ADT determines its efficiency depending on how that data structure is going to be used in practice.

3. **Transcendental Consciousness** is the unbounded, silent field of pure order and efficiency.
4. **Impulses within Transcendental Consciousness:** Within this field, the laws of nature continuously organize and govern all activities and processes in creation.
5. **Wholeness moving within itself:** In Unity Consciousness, when the home of all knowledge has become fully integrated in all phases of life, life is spontaneously lived in accord with natural law for maximum achievement with minimum effort.