Lists and Sequences
Outline and Reading

- Singly linked list
- Position ADT and List ADT (§2.2.2)
- Doubly linked list (§2.2.2)
- Sequence ADT (§ 2.2.3)
- Implementations of the sequence ADT (§ 2.2.3)
- Iterators (2.2.3)
Singly Linked List

A singly linked list is a concrete data structure consisting of a sequence of nodes.

- Each node stores:
  - element
  - link to the next node
Stack with a Singly Linked List

- We can implement a stack with a singly linked list.
- The top element is stored at the first node of the list.
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time.
Queue with a Singly Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time
Position ADT

The **Position** ADT models the notion of place within a data structure where a single object is stored.

It gives a unified view of diverse ways of storing data, such as:
- a cell of an array
- a node of a linked list

Just one method:
- `object element()`: returns the element stored at the position.
The List ADT models a sequence of positions storing arbitrary objects. It establishes a before/after relation between positions.

Generic methods:
- size(), isEmpty()

Query methods:
- isFirst(p), isLast(p)

Accessor methods:
- first(), last()
- before(p), after(p)

Update methods:
- replaceElement(p, o),
  swapElements(p, q)
- insertBefore(p, o),
  insertAfter(p, o),
- insertFirst(o),
  insertLast(o)
- remove(p)
Doubly Linked List

- 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 trailer and header nodes
We visualize operation `insertAfter(p, X)`, which returns position q.

**Insertion**
Deletion

We visualize \texttt{remove(p)}, where \( p = \text{last()} \)
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
Sequence ADT

- The **Sequence ADT** is the union of the Vector and List ADTs
- Elements accessed by
  - Rank, or
  - Position
- Generic methods:
  - size(), isEmpty()
- Vector-based methods:
  - elemAtRank(r), replaceAtRank(r, o), insertAtRank(r, o), removeAtRank(r)
- List-based methods:
  - 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)
Applications of Sequences

- The Sequence ADT is a basic, general-purpose, data structure for storing an ordered collection of elements

Direct applications:
- Generic replacement for stack, queue, vector, or list
- small database (e.g., address book)

Indirect applications:
- Building block of more complex data structures
Array-based Implementation

- We use a circular array storing positions
- A position object stores:
  - Element
  - Rank
- Indices \( f \) and \( l \) keep track of first and last positions
# Sequence Implementations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>size, isEmpty</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>atRank, rankOf, elemAtRank</td>
<td>1</td>
<td>(n)</td>
</tr>
<tr>
<td>first, last, before, after</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>replaceElement, swapElements</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>replaceAtRank</td>
<td>1</td>
<td>(n)</td>
</tr>
<tr>
<td>insertAtRank, removeAtRank</td>
<td>(n)</td>
<td>(n)</td>
</tr>
<tr>
<td>insertFirst, insertLast</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>insertAfter, insertBefore</td>
<td>(n)</td>
<td>1</td>
</tr>
<tr>
<td>remove</td>
<td>(n)</td>
<td>1</td>
</tr>
</tbody>
</table>
Iterators

- An iterator abstracts the process of scanning through a collection of elements.

Methods of the ObjectIterator ADT:
- `object object()`
- `boolean hasNext()`
- `object nextObject()`
- `reset()`

- 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, Vector, List and Sequence 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.