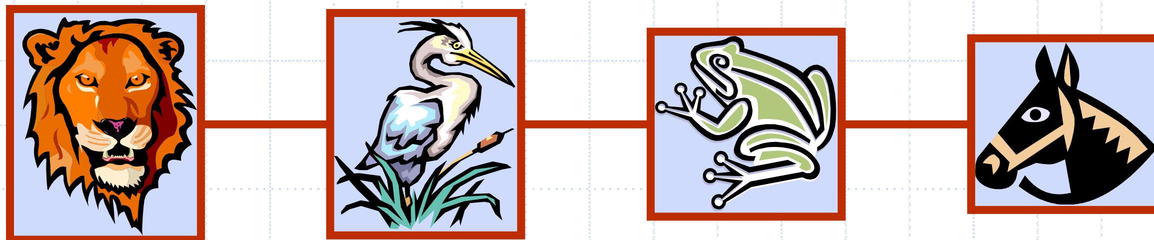
Lists and Sequences

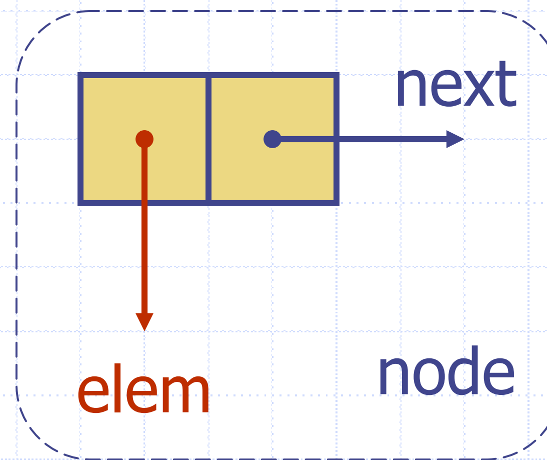


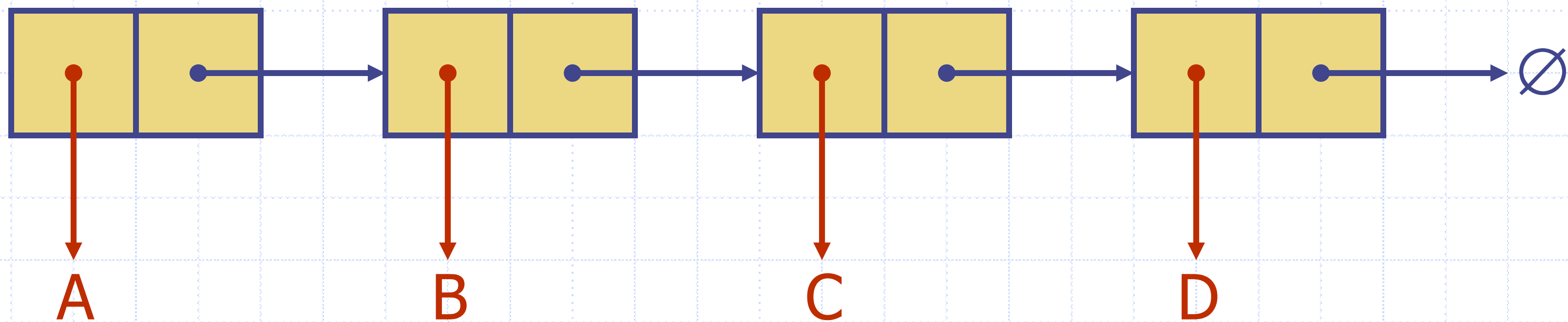
Outline and Reading

- Singly linked list (§3.2)
- Doubly linked list (§ 3.3)
- Position ADT and List ADT (§6.2)
- Sequence ADT (§ 6.4)
- Iterators (6.3)

Singly Linked List

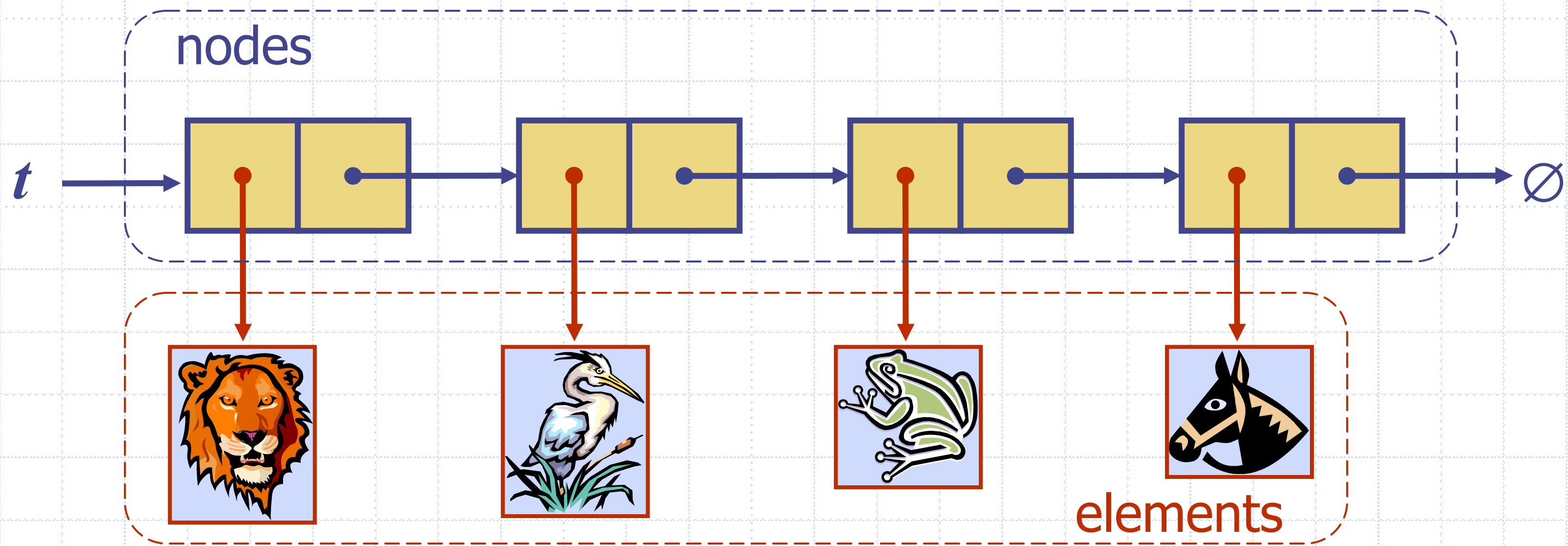
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





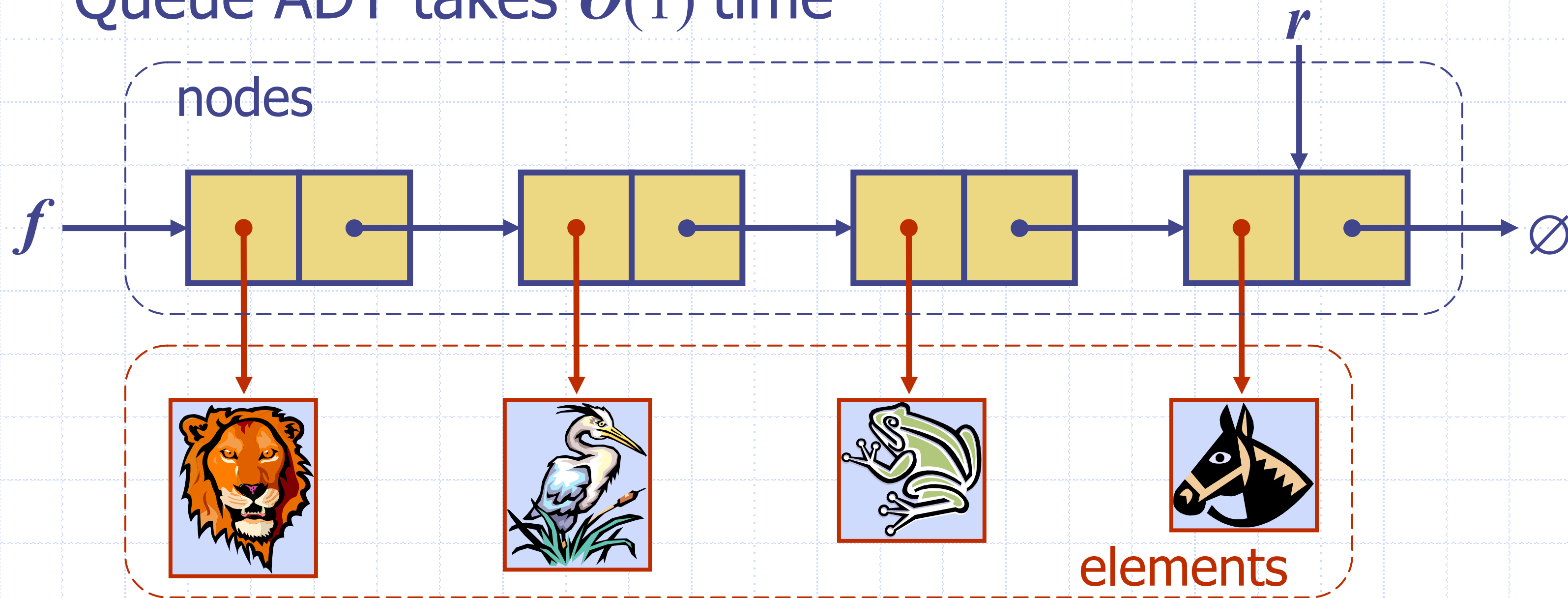
Stack with a Singly Linked List

- 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

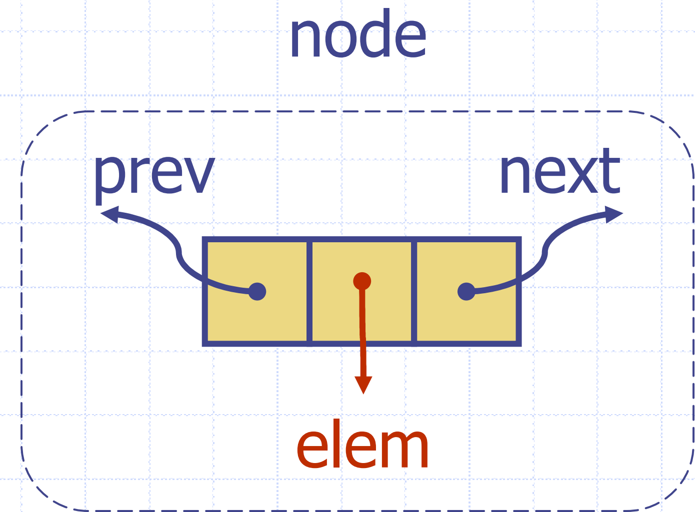


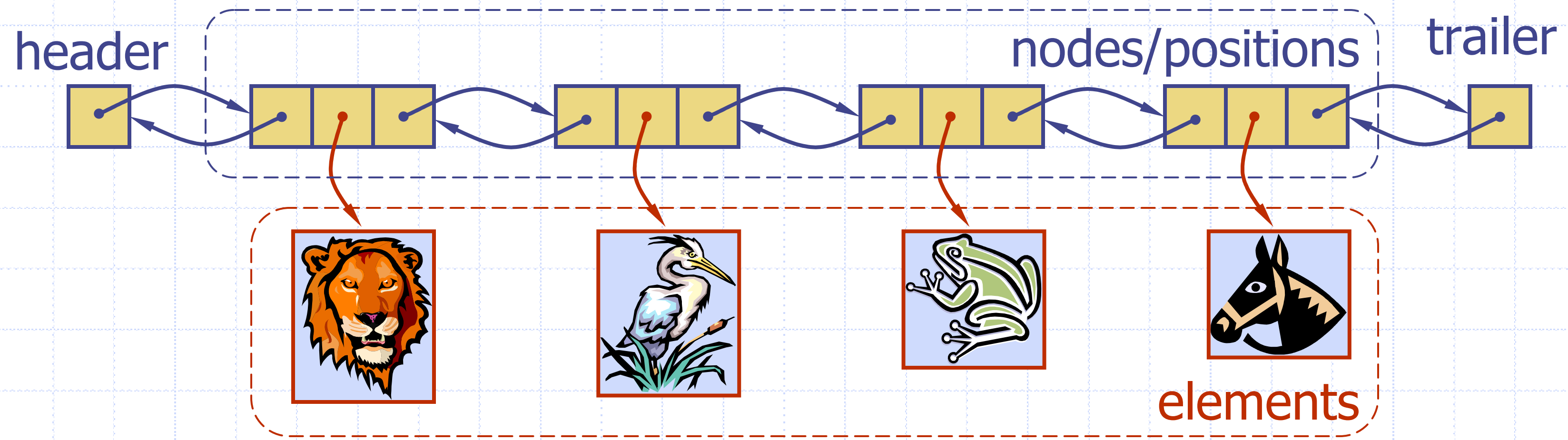
List ADT

- The List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), isEmpty()
- Accessor methods:
  - first(), last()
  - prev(p), next(p), get(p)
- Update methods:
  - set(p, e), set(i, e)
  - add(i, e),
  - addBefore(p, e), addAfter(p, e)
  - addFirst(e), addLast(e)
  - remove(i)

Doubly Linked List

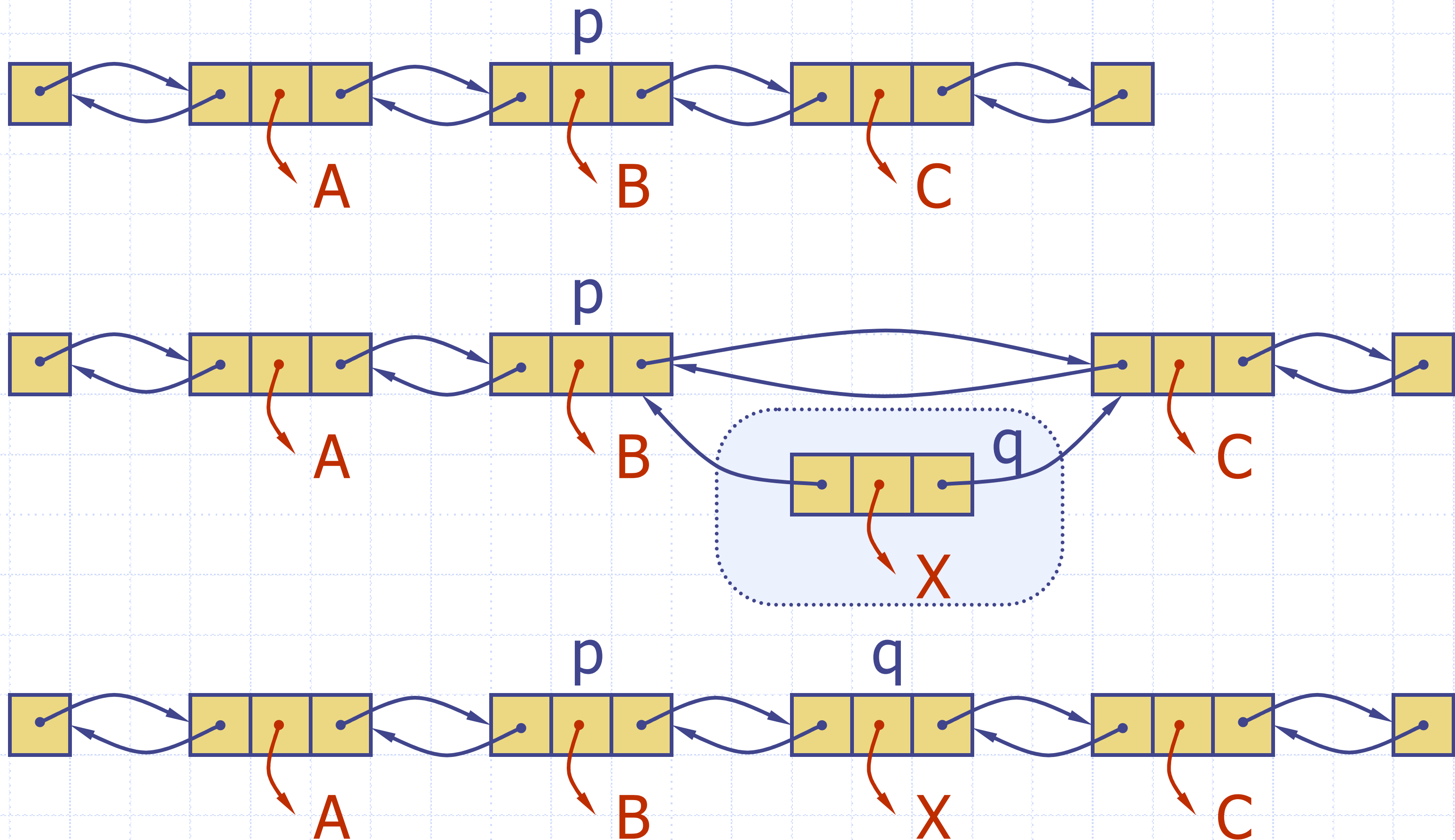
- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





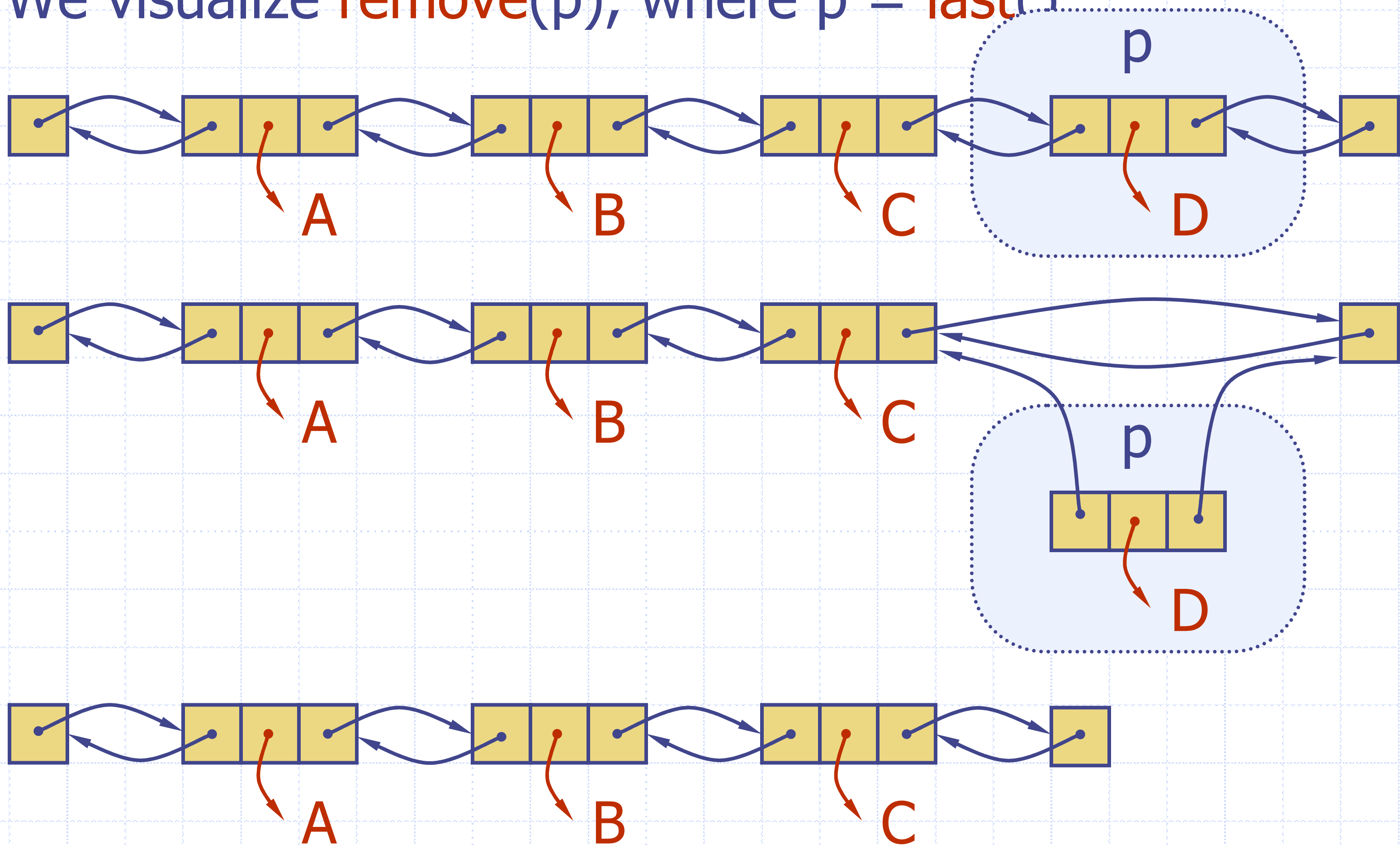
Insertion

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



Deletion

- We visualize remove(p), where p = 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

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

Sequence ADT

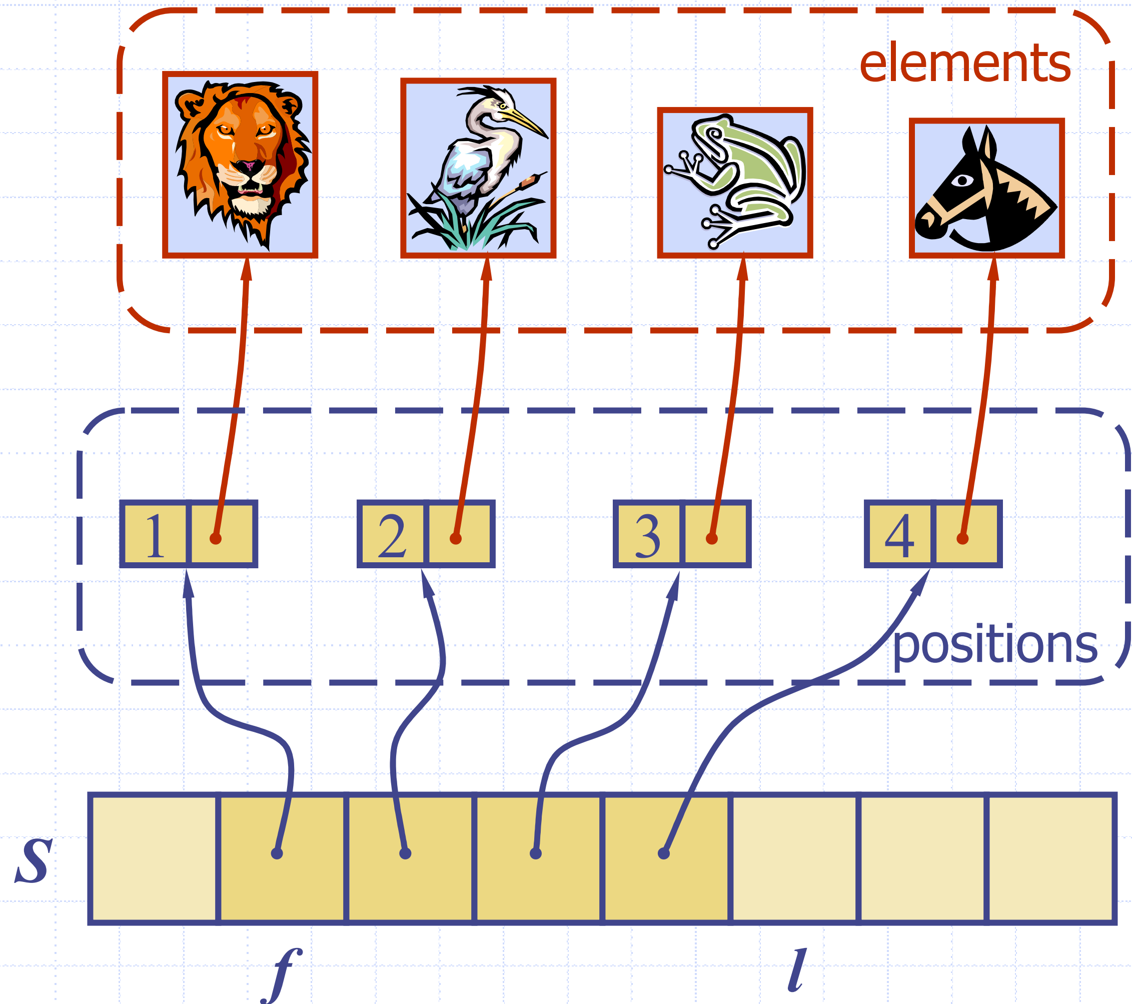
- The sequence ADT is the union of the array list and list ADTs
- Elements accessed by
  - Index, or
  - Position
- Generic methods:
  - size()
  - isEmpty()
- Deque-based methods:
  - addFirst(e)
  - addLast(e)
  - removeFirst()
  - removeLast()
  - getFirst()
  - getLast()
- List-based methods:
  - first()
  - last(),
  - prev(p)
  - next(p)
  - set(p, e)
  - addBefore(p, e)
  - addAfter(p, e)
  - remove(p), remove(i)
- Bridge methods:
  - atIndex(i)
  - indexOf(p)
- Array List-based methods
  - get(i)
  - set(i,e)

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
  - Index in array
- Indices *f* and *l* keep track of first and last positions



Sequence Implementations

Operation	Array	List
size, isEmpty	1	1
atIndex, IndexOf, get	1	n
first, last, prev, next	1	1
set(p,e)	1	1
set(i,e)	1	n
add, remove(i)	n	n
insertFirst, insertLast	1	1
insertAfter, insertBefore	n	1
remove(p)	n	1

Iterators

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