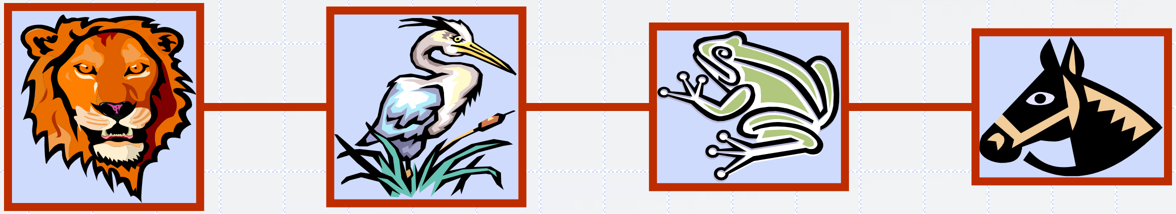
Array Lists, Node Lists & Sequences

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

Coverage



- Array Lists
- Node Lists
- Sequences

The Array List ADT

- The Array List ADT extends the notion of array by storing a sequence of arbitrary objects
- An element can be accessed, inserted or removed by specifying its index (number of elements preceding it)
- An exception is thrown if an incorrect index is given (e.g., a negative index)

The Array List ADT

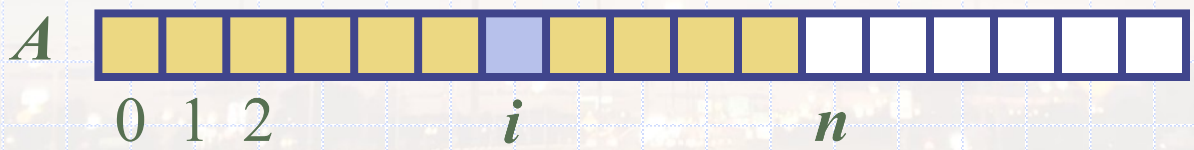
- Main methods:
  - get(integer i): returns the element at index i without removing it
  - set(integer i, object o): replace the element at index i with o and return the old element
  - add(integer i, object o): insert a new element o to have index i
  - remove(integer i): removes and returns the element at index i
- Additional methods:
  - size()
  - isEmpty()
- → See ArrayList1.java, ArrayList2.java & ArrayList3.java

Applications of Array Lists

- Direct applications
  - Sorted collection of objects (elementary database)
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

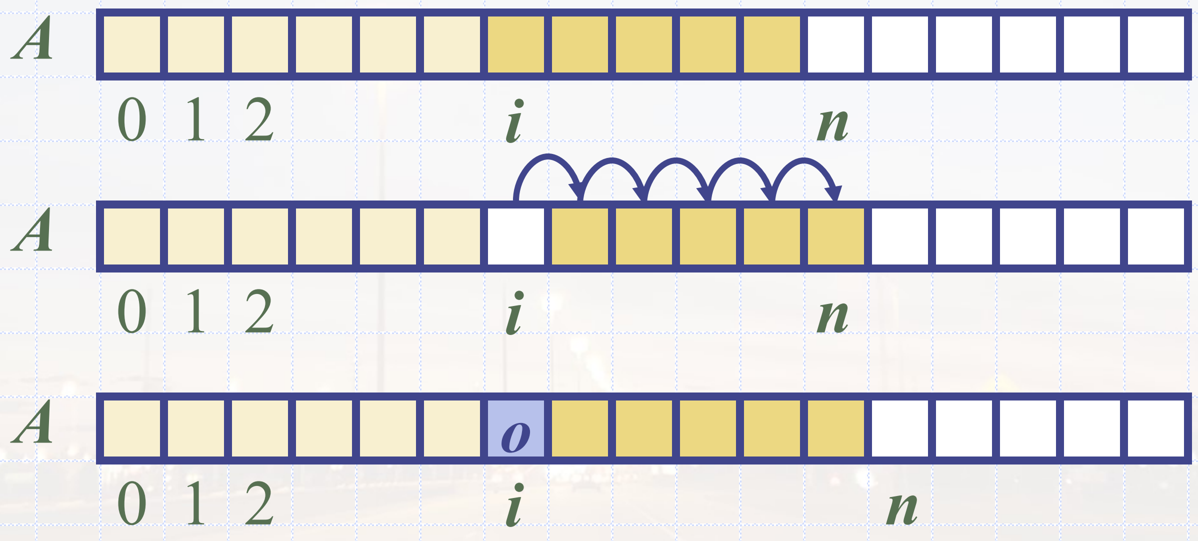
Array-based Implementation

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- Operation *get*(i) is implemented in $O(1)$ time by returning $A[i]$
- Operation *set*(i,o) is implemented in $O(1)$ time by performing $t = A[i]$, $A[i] = o$, and returning t.



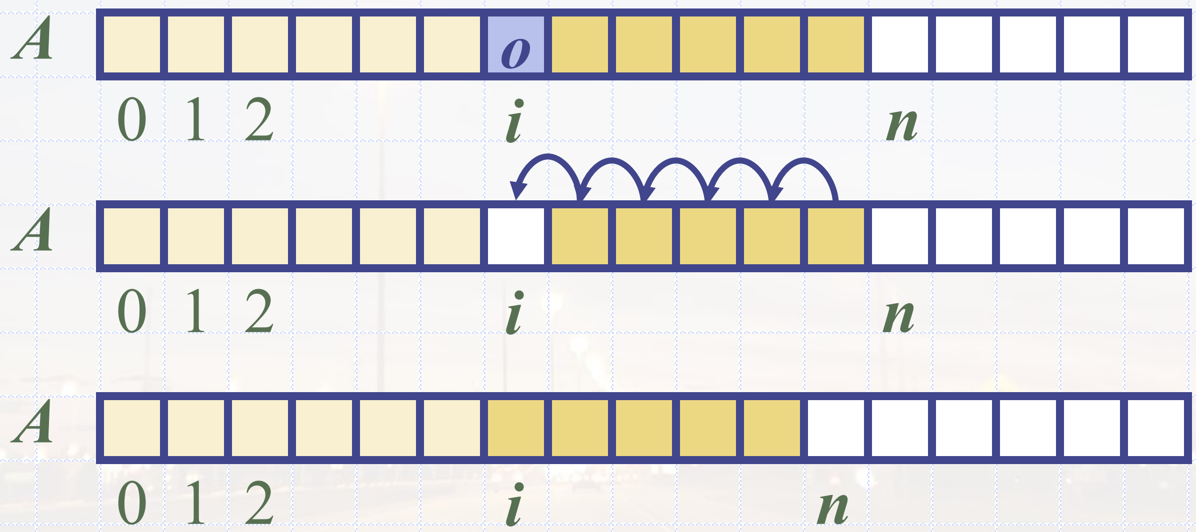
Insertion

- In operation ***add***(i, o), we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Element Removal

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



Performance

- In the array based implementation of an array list:
  - The space used by the data structure is $O(n)$
  - ***size***, ***isEmpty***, ***get*** and ***set*** run in $O(1)$ time
  - ***add*** and ***remove*** run in $O(n)$ time in worst case
- If we use the array in a circular fashion, operations ***add***$(0, x)$ and ***remove***$(0, x)$ run in $O(1)$ time
- In an ***add*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Growable Array-based Array List

- In an add(o) operation (without an index), we always add at the end
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant c
  - Doubling strategy: double the size

Algorithm *add*(*o*)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ **new array of size …**
    for $i \leftarrow 0$ **to** $n-1$ **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $n \leftarrow n + 1$
  $S[n-1] \leftarrow o$

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n add(o) operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an add operation the average time taken by an add over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$
$$n + c(1 + 2 + 3 + \ldots + k) =$$
$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an add operation is $O(n)$

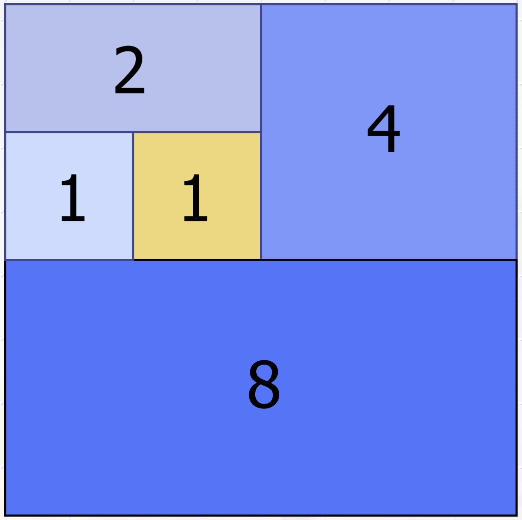
Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n add operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$
$$n + 2^{k+1} - 1 =$$
$$3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$

geometric series



Array List ADT

- That array list ADT can then provide an adapter class for the D.Q ADT, as shown below:

D.Q. Methods	Realization with the Array List Methods
size(), isEmpty()	size(), isEmpty()
getFirst()	get(0)
getLast()	get(size() - 1)
addFirst(e)	add(0, e)
addLast(e)	add(size(), e)
removeFirst()	remove(0)
removeLast()	remove(size() – 1)

Node List ADT

- Using an index is sometimes very suitable, such as in the case arrays, for referencing an element in a collection.
- However, in the case of linked lists, it could be more efficient to use a node (actually pointer to a node) instead of an index to reference and update an element.
- However, returning a pointer to a node to reference it may very well lead to privacy leak and compromises the list.

Node List ADT - Positions

- The node list ADT abstracts the concrete linked list data structure to allow elements of the list to be accessed by their relative position in the list.
- That is, a list is viewed as a collection of nodes that are referenced/located by their positions. The positions are arranged in a linear order.
- That is, instead of returning pointers to the nodes, which may compromise privacy, the node list ADT utilizes another ADT, referred to as Position ADT, to locate the elements of the list.

Node List ADT - Positions

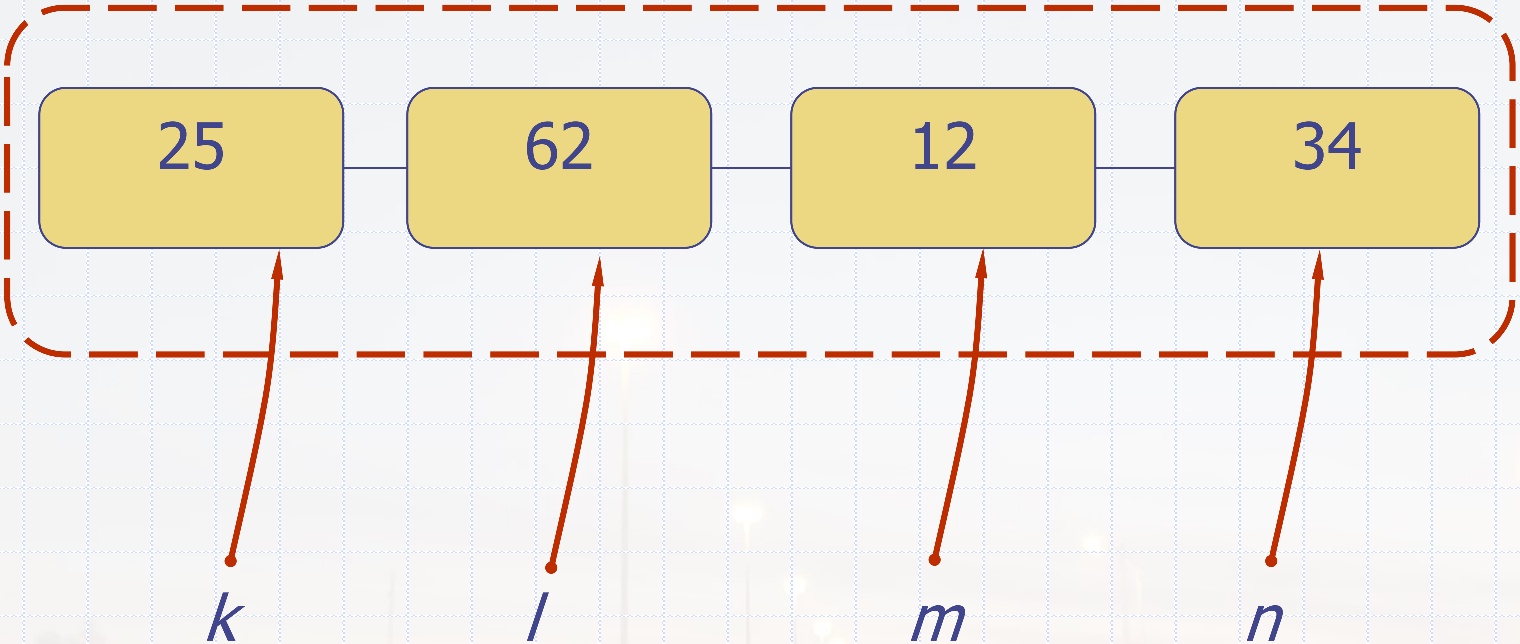
- The Position ADT support the following method:
  - element(), which returns the element stored at this position

- A position is always defined ***relatively***; that is in relation to its neighbors.

- That is, in a list, each position *b* will always be after a position, *a*, and before a position *c*, except when *b* is the first or last position in the list.

- A position in the list does not change even if the element it points to changes or swapped. It is only destroyed if the element it points to is explicitly removed.

Node List ADT - Positions



Node List ADT

- Using Positions, the node list ADT can define the following methods:
  - first(): returns the position of the first element; error if list is empty.
  - last(): returns the position of the last element; error if list is empty.
  - prev(*p*): returns the position preceding position *p* in the list; error if *p* is first position.
  - next(*p*): returns the position following position *p* in the list; error if *p* is last position.

- The above methods allows us to refer to relative positions in the list, starting at the beginning or end and to move inclemently up or down the list.

Node List ADT

- Additionally, the following methods are defined:
  - set(*p*, *e*): Replace the element at position *p* with *e* , and return that old element at position *p*.
  - addFirst(*e*): Insert a new element *e* as the first element and returns the position object.
  - addLast(*e*): Insert a new element *e* as the last element and returns the position object.
  - addBefore(*p*, *e*): Insert a new element *e* before position *p* and returns the position object.
  - addAfter(*p*, *e*): Insert a new element *e* after position *p* and returns the position object.
  - remove(*p*): Remove and return the element at position *p*. This also invalidates that position *p* in the list.
- size() and isEmpty() methods can also be defined.

Node List ADT

- Is there a redundancy between the methods of the ADT?
  - For instance,

    addFirst(*e*) can be replaced by addBefore(first(), *e*)

    addLast(*e*) can be replaced by addAfter(last(), *e*)

➔ No; these substitutions will only work if the list is not empty.

Node List ADT

- Example of Node List:

Operation	Output For clarity, object stored at the returned position is also shown after "/"	List
addFirst(8)	--	[8]
first()	p1 / 8	[8]
addAfter(p1, 5)	--	[8, 5]
next(p1)	p2 / 5	[8, 5]
addBefore(p2, 3)	--	[8, 3, 5]
prev(p2)	p3 / 3	[8, 3, 5]
addFirst(9)	--	[9, 8, 3, 5]
last()	p2 / 5	[9, 8, 3, 5]
remove(first())	9	[8, 3, 5]
set(p3, 7)	3	[8, 7, 5]
addAfter(first(), 2)	--	[8, 2, 7, 5]

Node List ADT

- That node list ADT can then provide an adapter class for the D.Q ADT, as shown below:

D.Q. Methods	Realization with the Node List Methods
size(), isEmpty()	size(), isEmpty()
getFirst()	first().element()
getLast()	last().element()
addFirst(e)	addFirst(e)
addLast(e)	addLast(e)
removeFirst()	remove(first())
removeLast()	remove(last())

Sequence ADT

- Sequence is an ADT that supports all the methods of the D.Q., the Array List, and the Node List ADTs
- Elements accessed by
  - Index, or
  - Position
- Generic methods:
  - size(), isEmpty()
- ArrayList-based methods:
  - get(i), set(i, e), add(i, e), remove(i)

Sequence ADT

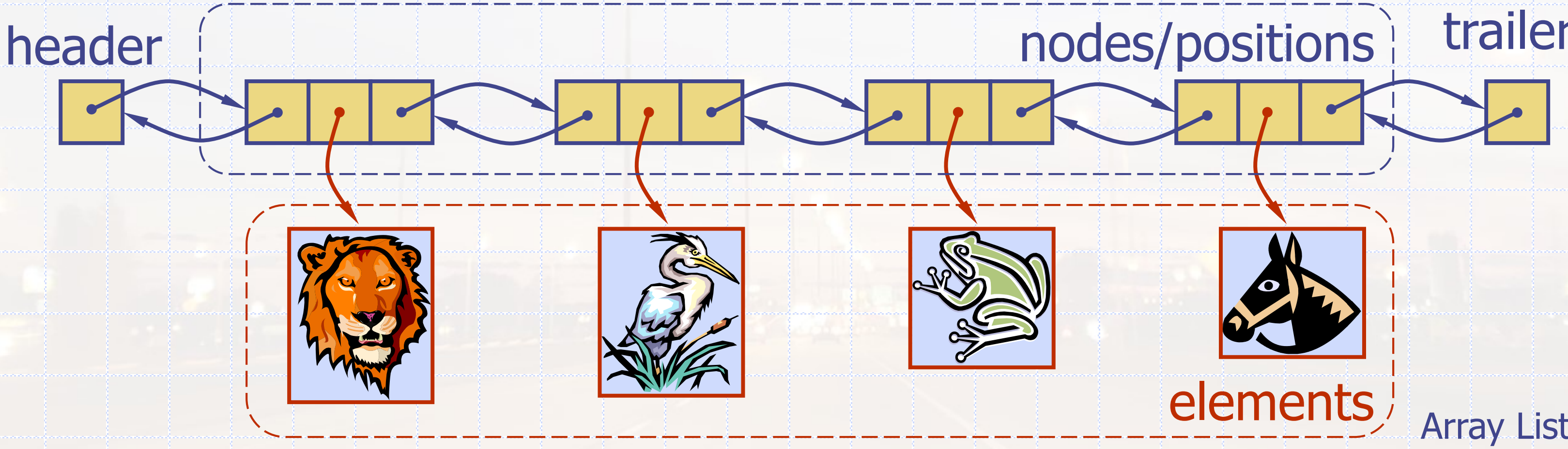
- NodeList-based methods:
  - first(), last(), prev(p), next(p), set(p, e), addBefore(p, e), addAfter(p, e), addFirst(e), addLast(e), remove(p)
- Additionally, the ADT provide two “bridging” methods, which provide connection between indices and positions:
  - atIndex(i): Returns the position of the element with index *i*; an error if $i < 0$ or $i > size - 1$
  - indexOf(p): Returns the index of the element at position *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, or list (since all needed methods are supported)
  - small database (e.g., address book)
- Indirect applications:
  - Building block of more complex data structures

Linked List Implementation

- A doubly linked list provides a reasonable implementation of the Sequence ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes
- Position-based methods run in constant time
- Index-based methods require searching from header or trailer while keeping track of indices; hence, run in linear time



Linked List Implementation

- <u>Efficiency of linked list implementation of Sequence ADT:</u>
  - All position-based operations (such as first(), last(), prev(*p*), next(*p*), set(*p, e*), addFirst(*e*), addLast(*e*), addBefore(*p, e*), addAfter(*p, e*), remove(*p*)), run in **O(1).**
  - All D.Q. operations also run in **O(1)**, since they only involve the two ends of the list.

Linked List Implementation

- Efficiency of linked list implementation of Sequence ADT:

  - However, methods of the ArrayList (such as get(i), set(i, e), add(i, e), remove(i)) would run in **O(n)** with such implementation since these operations would require hopping from one end of the list towards the other until locating index *i*.

  - Can this be optimized?

Linked List Implementation

- Efficiency of linked list implementation of Sequence ADT:
  - A slight optimization for methods such as get(i), set(i, e) can be achieved by starting from the closer end to the index. Locating index *i* would result in a running time of:
    - **O(min(i+1, n - i))**, where *n* is the number of elements in the list.
    - Worst case occurs when *i = floor(n/2)* ➔ Still **O(n)**

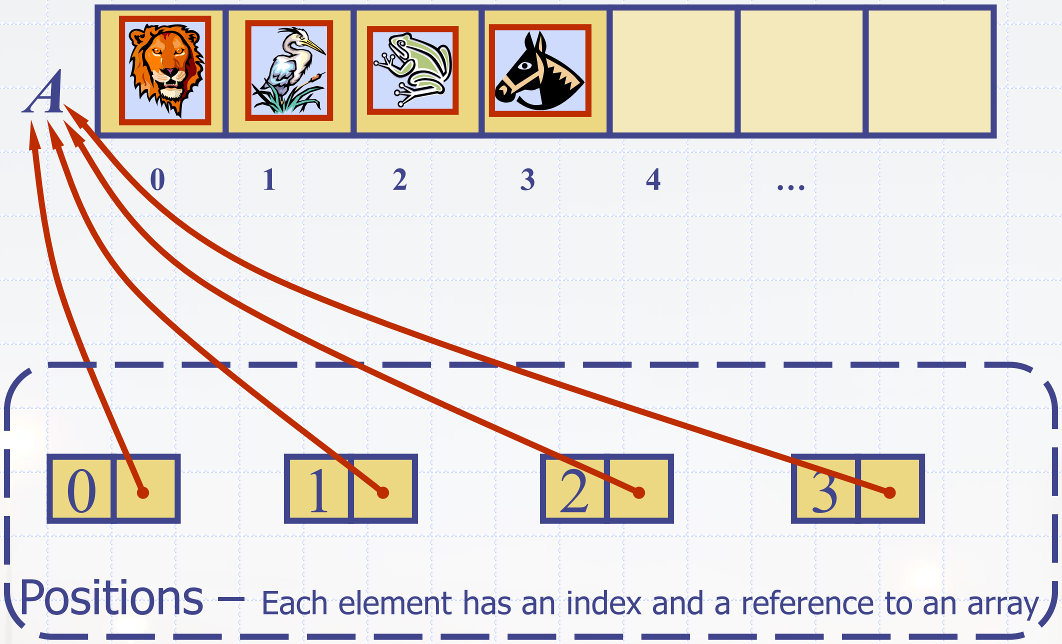
Linked List Implementation

- <u>Efficiency of linked list implementation of Sequence ADT:</u>
  - Similarly, running time of add(i, e) and remove(i) would be:

    O(min(i+1, n – i+ 1)), which still **O(n).**
    - One advantage of this approach however is that:
      - If $i = 0$ or $i = n - 1$, as is the case of the adaption of AttayList ADT to the D.Q ADT, then add and remove would run in O(1).
  - Nonetheless, as a general conclusion, linked list implementation for Sequence ADT is inefficient for ArrayList methods.

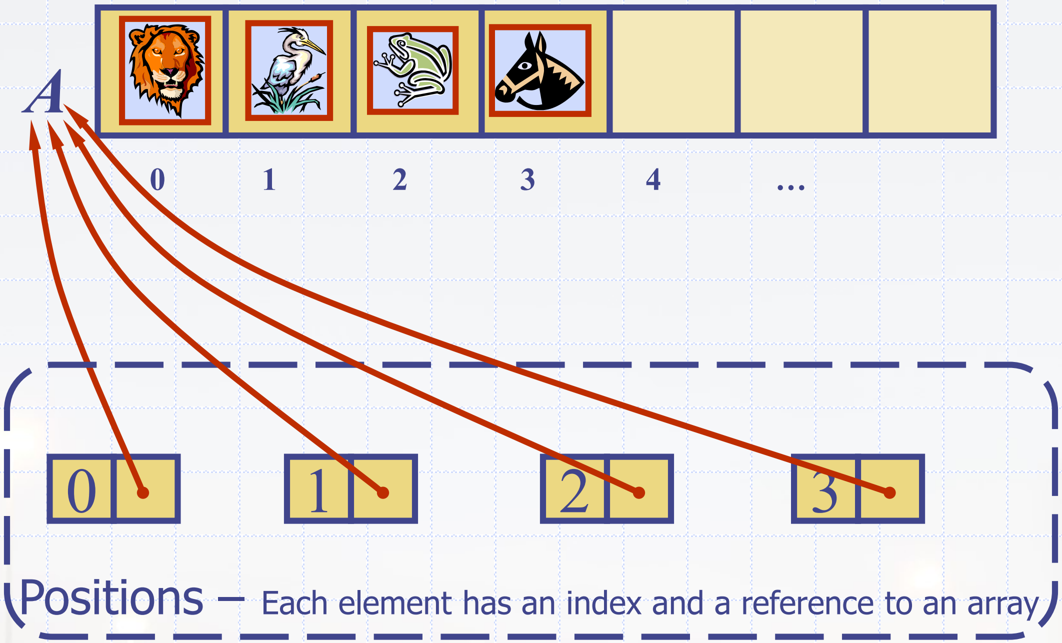
Array-based Implementation

- We can use an array storing each elements in a cell *A[i]*.
- A position object can the be defined to hold:
  - An index
  - A reference to the array



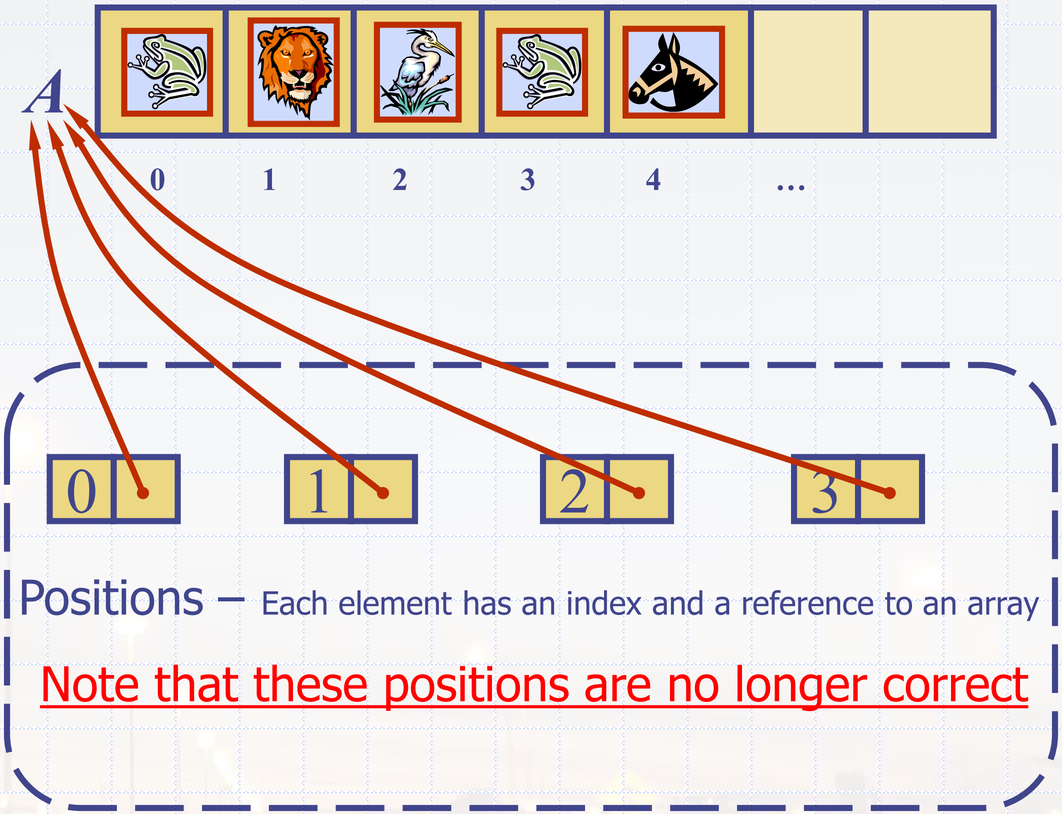
Array-based Implementation

- We then implement method *element(p)*, which returns *A[i]*.
- This approach however has a major drawback.
- The cells in the array have no way to reference their corresponding positions.



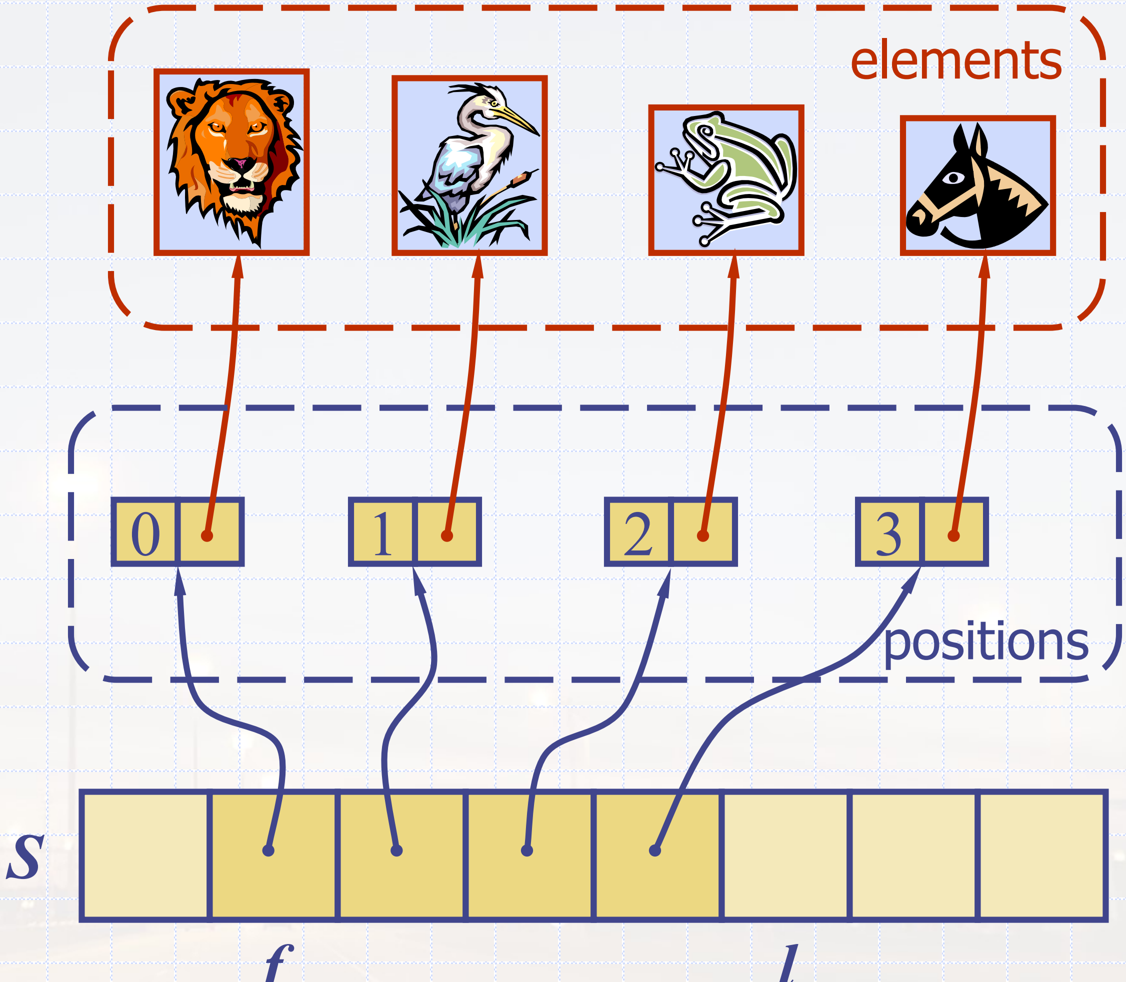
Array-based Implementation

- For instance, after performing *add(i, e)*, there is no way to inform the positions of S that their indices went up.
- Recall that positions are defined relatively to their neighbor positions and not to their indices.



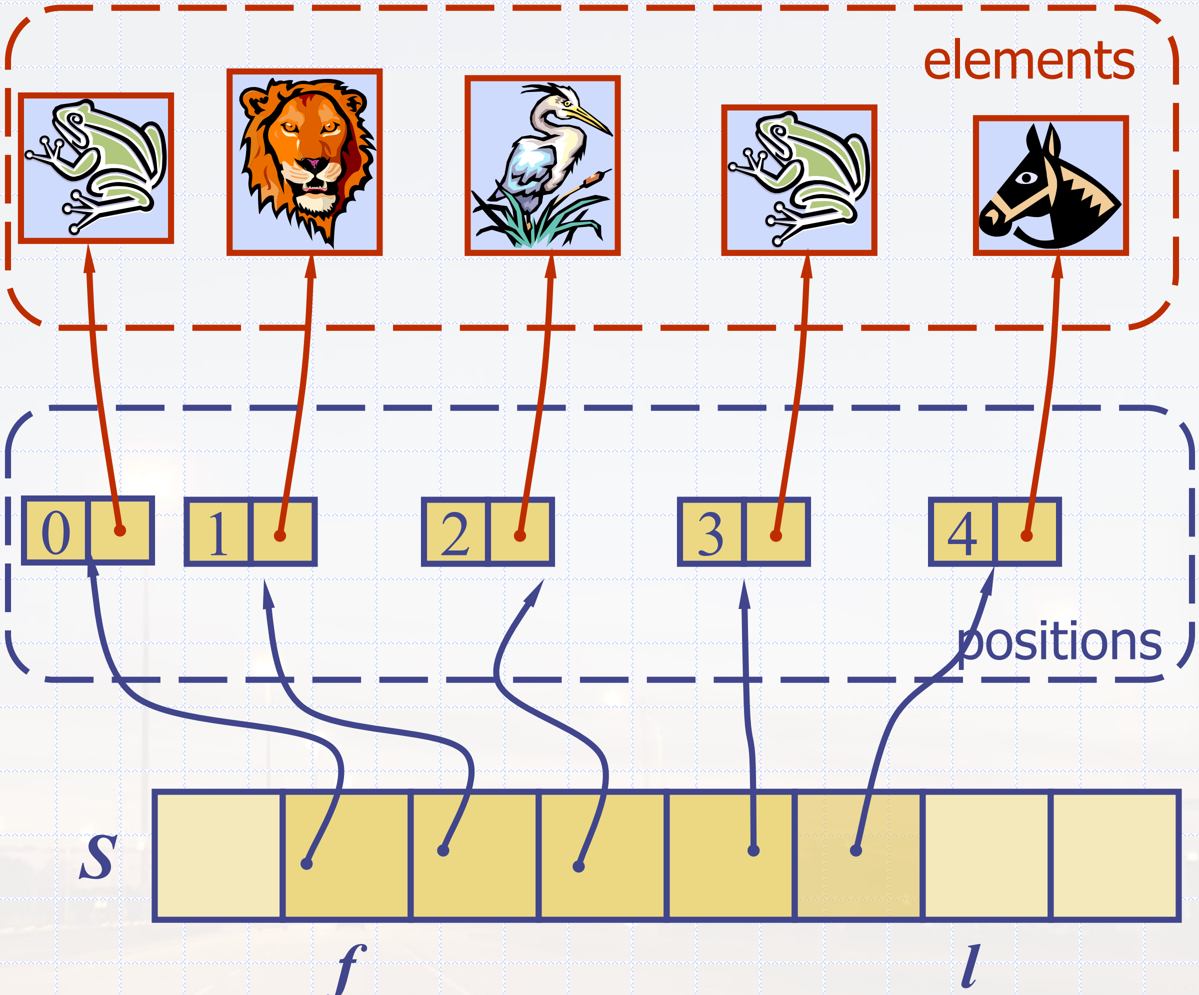
Array-based Implementation

- Alternatively, we can use a circular array storing positions
- A new position object is defined to store:
  - Index
  - Element associated with the position



Array-based Implementation

- With this data structure, we can easily scan through the array to update the index variable for each position whose index changed because of an insertion or deletion.



Array-based Implementation

- <u>Efficiency of array-based implementation of Sequence ADT:</u>
  - add(i), addFirst(*e*), addBefore(*p, e*), addAfter(*p, e*), remove(*p*) , remove(i), run in **O(n)** since we need to shift position objects to make room for insertion or adjust positions after removal.
  - All other methods run in **O(1)**.

Comparing 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(i), remove(i)	n	n
addFirst	n	1
addLast	1	1
addAfter, addBefore	n	1
remove(p)	n	1

Favorites List ADT

- The Favorites List ADT models a collection of elements while keeping track of the number of times each element is accessed.
- The access counts allow us to know which elements are most frequently accessed.
- Additional Methods:
  - access(e): accessed the element e while incrementing its access count.
  - remove(e): removes the element e from list.
  - top(k): returns list of k most accessed elements.

Applications of Favorites List

- Keeping track of most popular Web addresses for a Web browser.
- For a GUI interface: keeping track of most popular buttons for a pull-down menu.