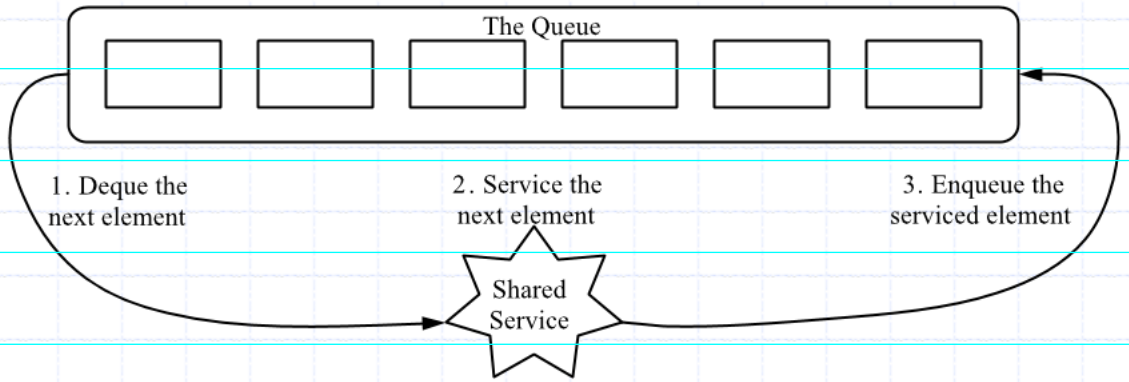
1] Suppose you were asked to implement a stack using one or more queues. What is the best implementation possible (in terms of efficiency of the push and pop operations)?

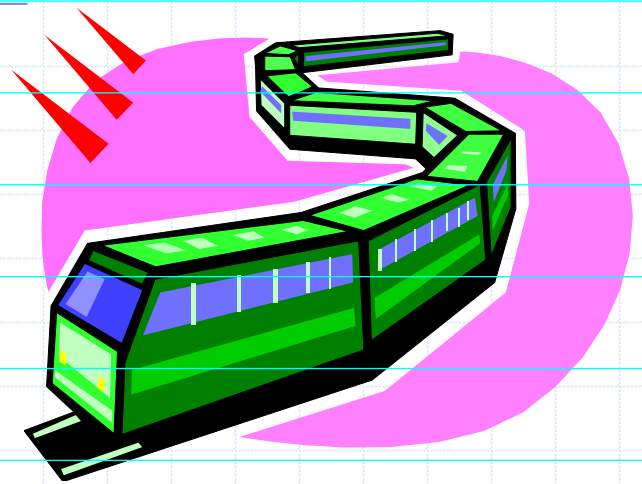
2] Now suppose you were asked to implement a queue using one or more stacks. Again, what is the best implementation possible (in terms of efficiency of the enqueue and dequeue operations)?

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue, *Q*, by repeatedly performing the following steps:
  1. *e* = *Q*.dequeue()
  2. Service element *e*
  3. *Q*.enqueue(*e*)



Lists



Position ADT (§ 5.2.2)

- 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

List ADT (§ 5.2.3)

- 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:

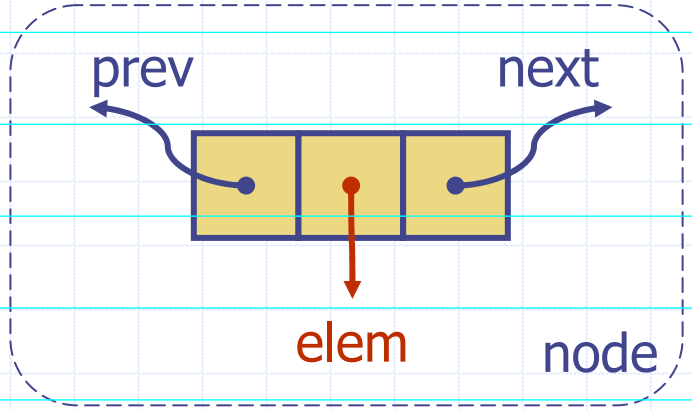
- first(), last()
- prev(p), next(p)

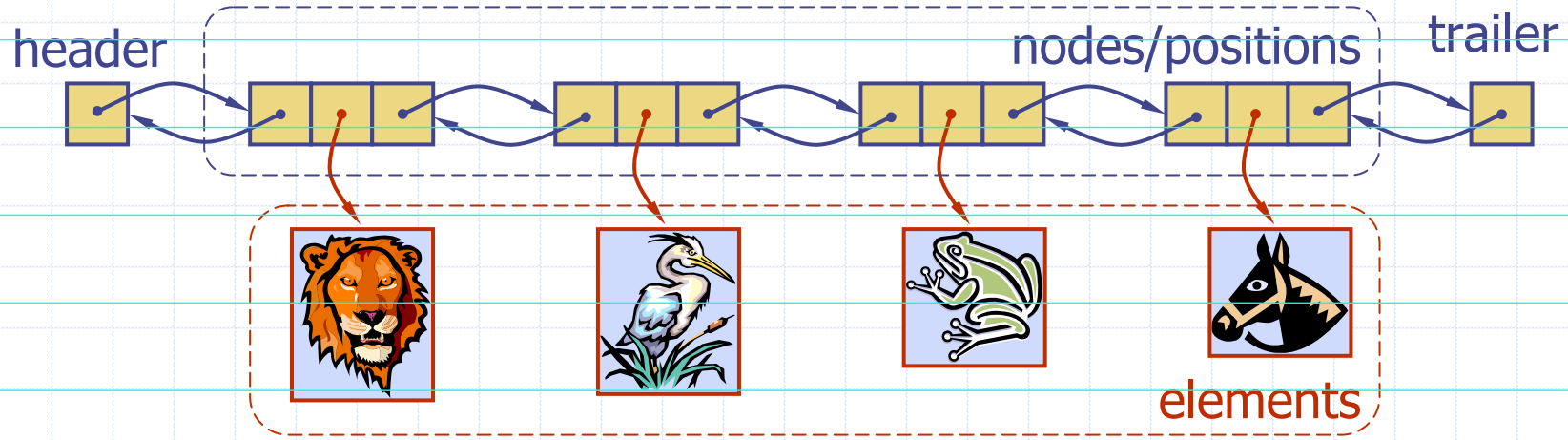
Update methods:

- replace(p, e)
- insertBefore(p, e), insertAfter(p, e),
- insertFirst(e), insertLast(e)
- remove(p)

Doubly Linked List

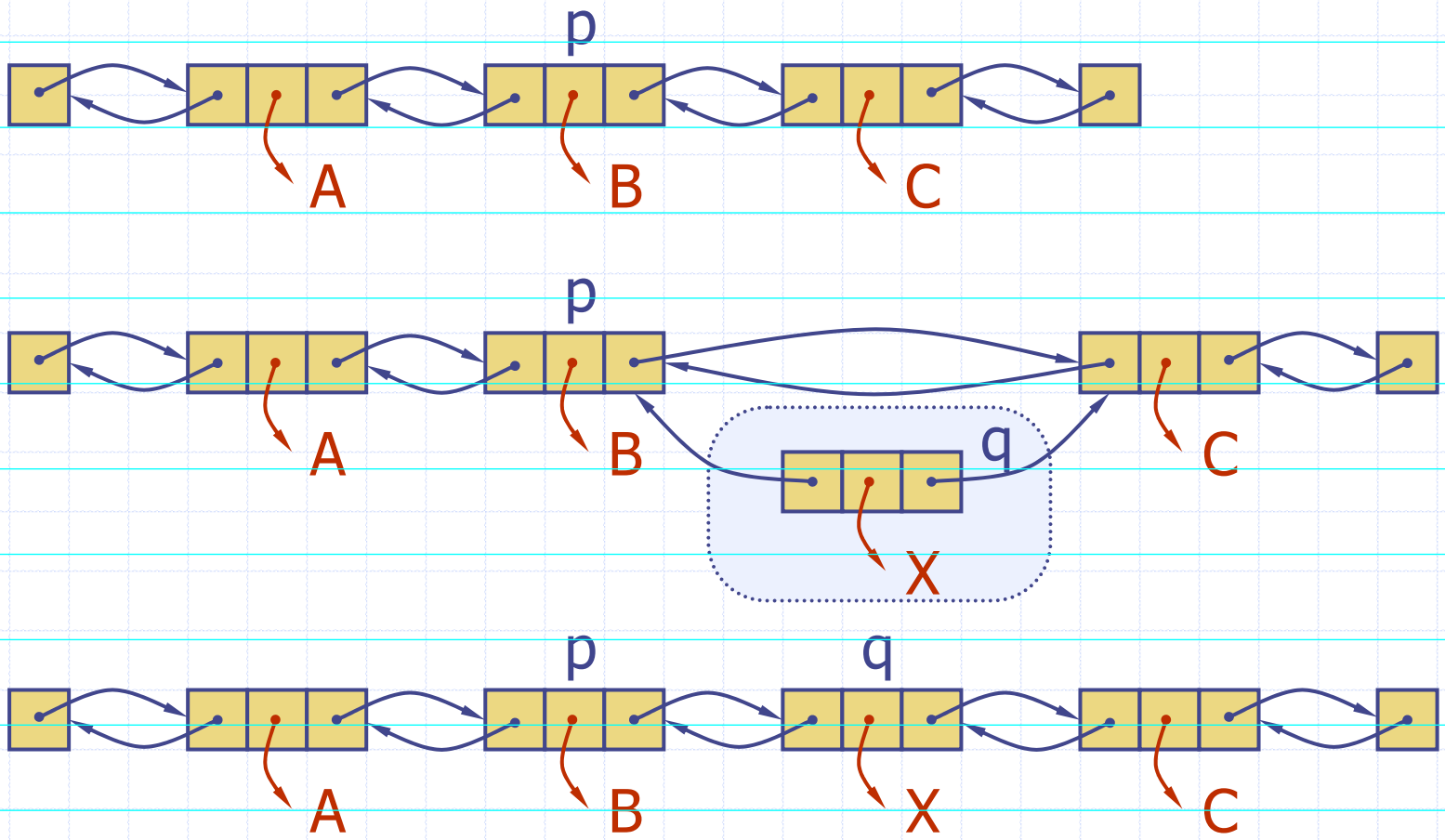
- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Insertion

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

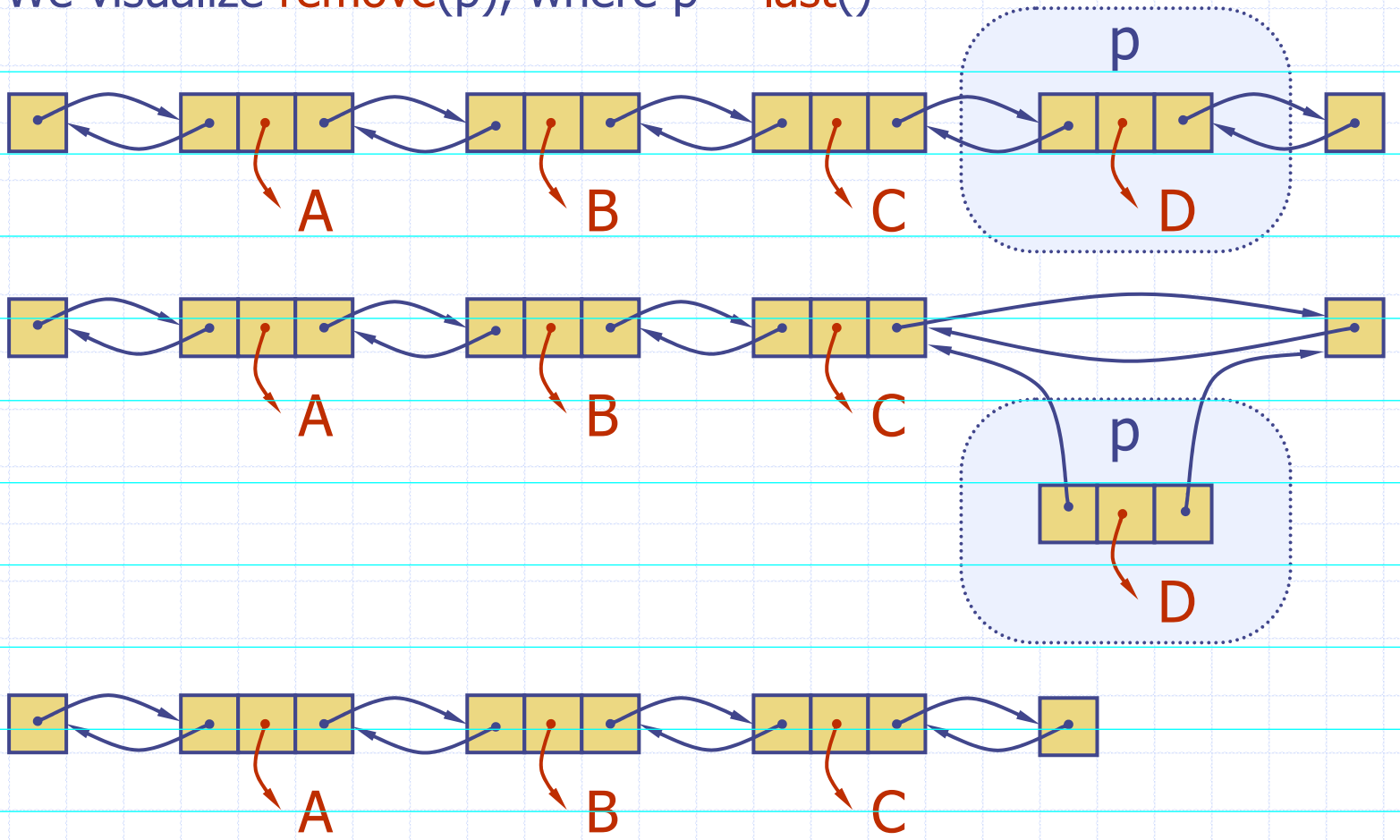


Insertion Algorithm

Algorithm insertAfter(*p*,*e*):
Create a new node *v*
v.setElement(*e*)
v.setPrev(*p*) {link *v* to its predecessor}
v.setNext(*p*.getNext()) {link *v* to its successor}
(*p*.getNext()).setPrev(*v*) {link *p*'s old successor to *v*}
p.setNext(*v*) {link *p* to its new successor, *v*}
return *v* {the position for the element *e*}

Deletion

- We visualize remove(p), where p = last()



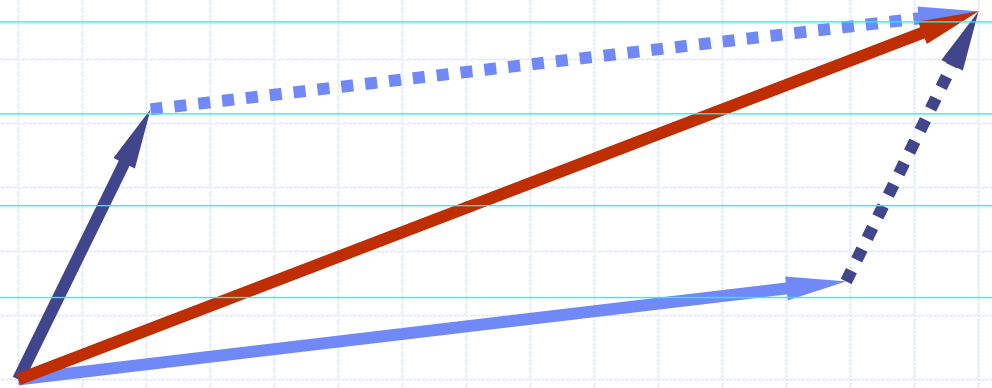
Deletion Algorithm

Algorithm remove(*p*):
    t = *p*.element {a temporary variable to hold the return value}
    (*p*.getPrev()).setNext(*p*.getNext()) {linking out *p*}
    (*p*.getNext()).setPrev(*p*.getPrev())
    p.setPrev(**null**) {invalidating the position *p*}
    p.setNext(**null**)
    return *t*

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

Vectors and Array Lists



The Vector ADT (§5.1)

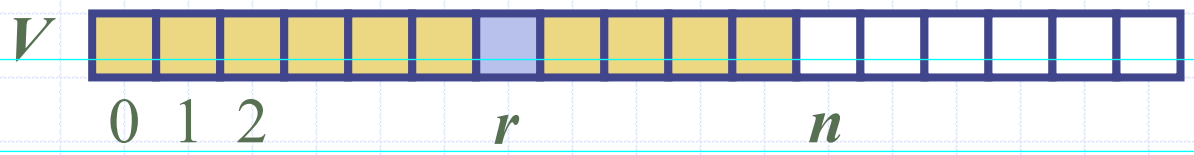
- The Vector ADT extends the notion of array by storing a sequence of arbitrary objects
- An element can be accessed, inserted or removed by specifying its rank (number of elements preceding it)
- An exception is thrown if an incorrect rank is specified (e.g., a negative rank)
- Main vector operations:
  - object elemAtRank(integer r): returns the element at rank r without removing it
  - object replaceAtRank(integer r, object o): replace the element at rank with o and return the old element
  - insertAtRank(integer r, object o): insert a new element o to have rank r
  - object removeAtRank(integer r): removes and returns the element at rank r
- Additional operations size() and isEmpty()

Applications of Vectors

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

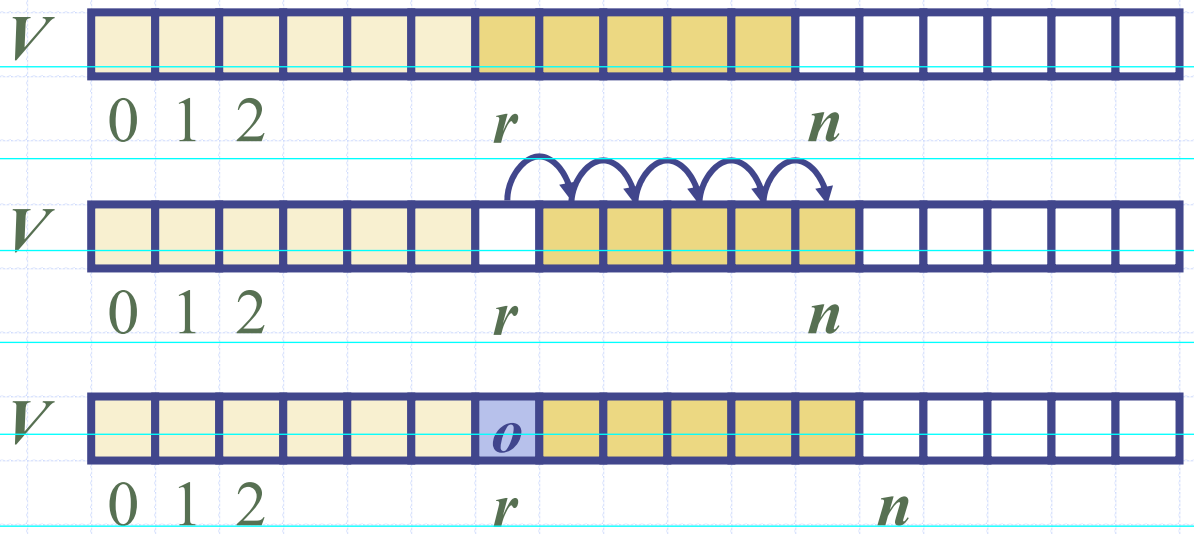
Array-based Vector

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



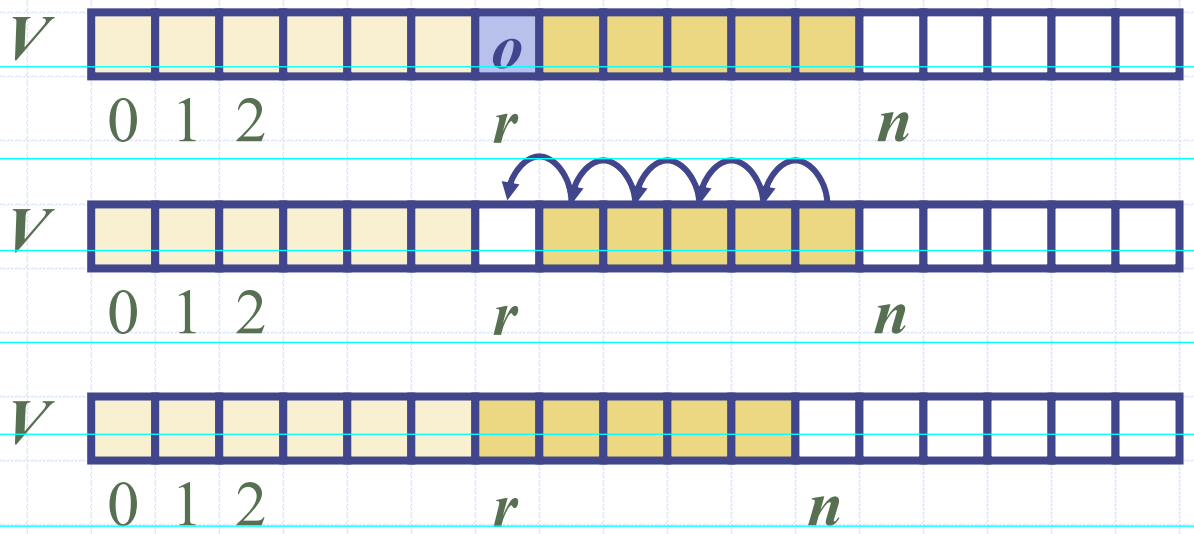
Insertion

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



Performance

- In the array based implementation of a Vector
  - 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
- If we use the array in a circular fashion, *insertAtRank*(0) and *removeAtRank*(0) run in $O(1)$ time
- In an *insertAtRank* operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Growable Array-based Vector

- In a push operation, when the array is full, instead of throwing an exception, we can 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 *push*(o)
  if $t = S.length - 1$ **then**
    $A \leftarrow$ new array of size …
    for $i \leftarrow 0$ **to** t **do**
      $A[i] \leftarrow S[i]$
    $S \leftarrow A$
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$

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 push operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of a push operation the average time taken by a push 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 push 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 a push operation is $O(n)$

Doubling Strategy Analysis

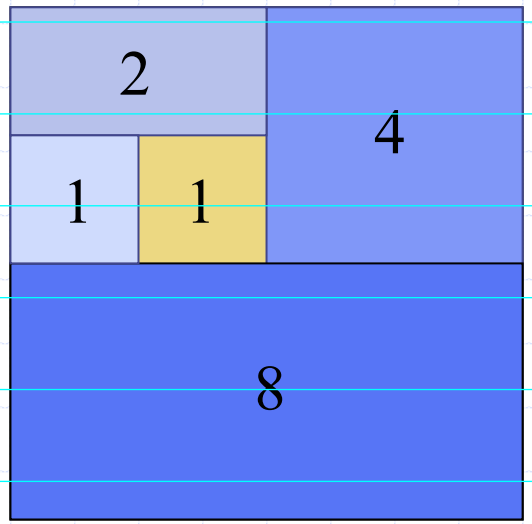
- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k =$$

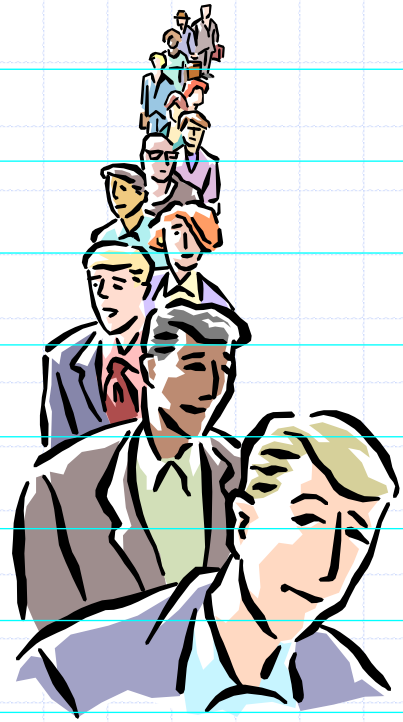
$$n + 2^{k+1} - 1 = 2n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$

geometric series



Sequences and Iterators



Sequence ADT (§ 5.3)

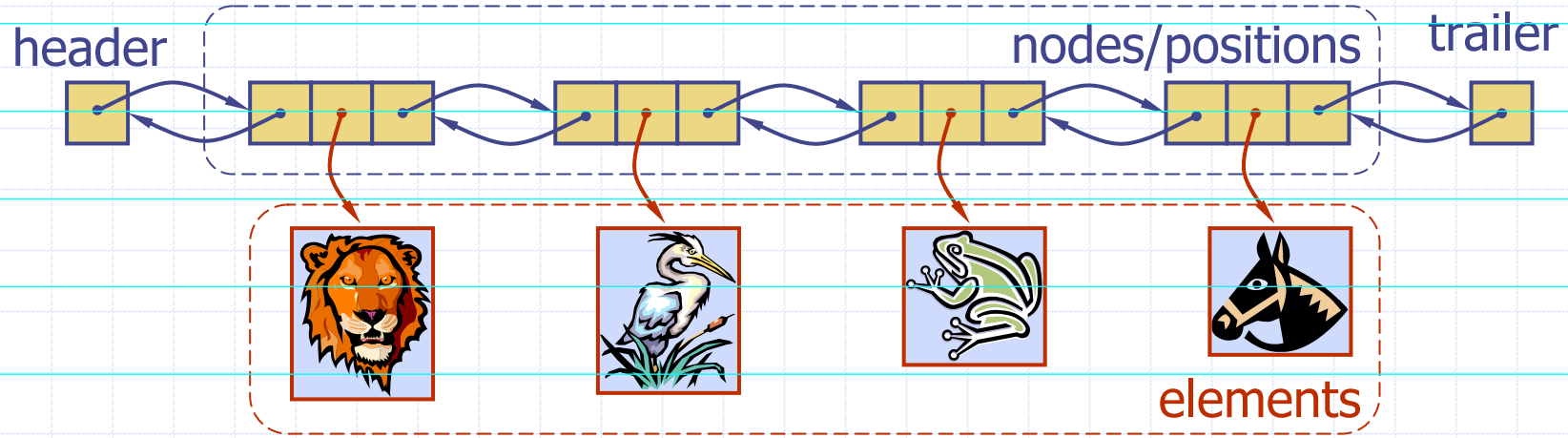
- The Sequence ADT is the union of the Vector and List ADTs
- Elements accessed by
  - Rank, or
  - Position
- Generic methods:
  - size(), isEmpty()
- Vector-based methods:
  - elemAtRank(r), replaceAtRank(r, o), insertAtRank(r, o), removeAtRank(r)
- List-based methods:
  - first(), last(), prev(p), next(p), replace(p, o), 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

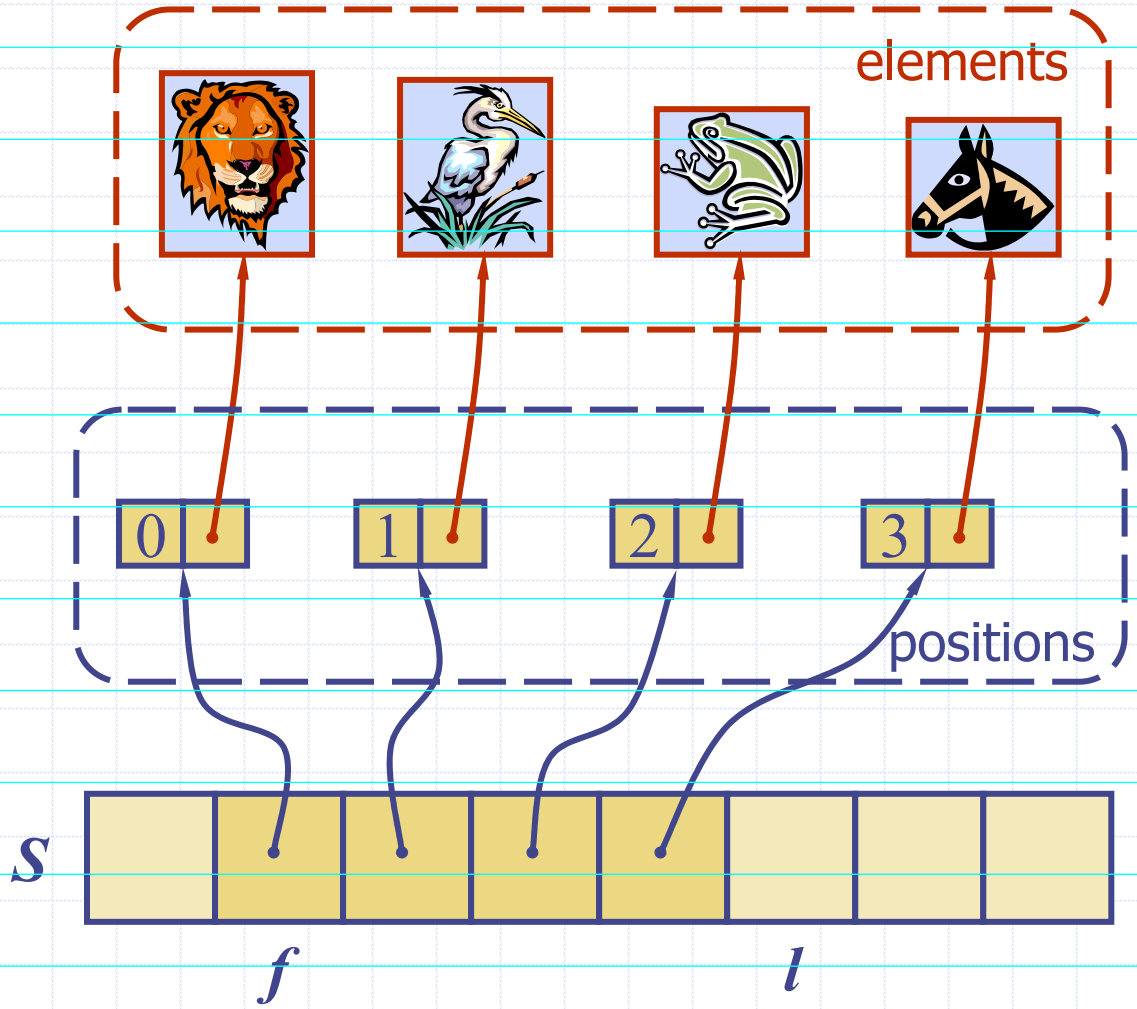
Linked List Implementation

- 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
- Rank-based methods require searching from header or trailer while keeping track of ranks; hence, run in linear time



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

Operation	Array	List
size, isEmpty	1	1
atRank, rankOf, elemAtRank	1	n
first, last, prev, next	1	1
replace	1	1
replaceAtRank	1	n
insertAtRank, removeAtRank	n	n
insertFirst, insertLast	1	1
insertAfter, insertBefore	n	1
remove	n	1

Iterators (§ 5.4)

- 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 an 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

Trees