Lecture 3: Sequences

Pure Knowledge Has
Infinite Organizing Power

1

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.

2

Review: Key Idea in Implementing a List

- Elements are accessed by Position
- Position is an ADT that models a particular place or location in a data structure

3

Position ADT

- The Position ADT models the notion of a place within a data structure where a single object/element 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 or tree
- Just one method:
  - element(): returns the element stored at the position

4

List ADT: Complexity?

- 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, e), swapElements(p, q)
  - insertBefore(p, e), insertAfter(p, e),
  - insertFirst(e), insertLast(e)
  - remove(p)

5

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

6

Sequence ADT

7

Outline

- Sequence ADT
- Implementation of the Sequence ADT
- Iterators

8

The Sequence ADT

- A Sequence stores a sequence of elements
- Element access is based on the concept of Rank
  - 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)

9

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

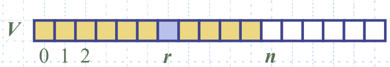
insertAtRank(*r*, *e*):
- insert a new element *e* to have rank *r*

removeAtRank(*r*):
- removes and returns the element at rank *r*

10

The Sequence is Array-based

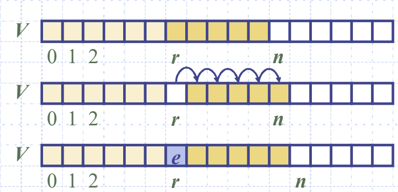
- Use an array V of size N
- A variable n keeps track of the size of the vector (number of elements stored)
- Operation ***elemAtRank***(r) is implemented in $O(1)$ time by returning $V[r]$



11

Insertion

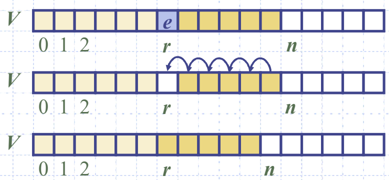
- In operation ***insertAtRank***(r, e), we need to make room for the new element by shifting forward the $n - r$ elements $V[r], \ldots, V[n-1]$
- In the worst case ($r = 0$), this takes $O(n)$ time



12

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



13

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

14

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:
  - 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)

15

How can we improve Performance?

- In the array-based implementation of a Sequence
  - ***insertAtRank*** and ***removeAtRank*** run in $O(n)$ time

16

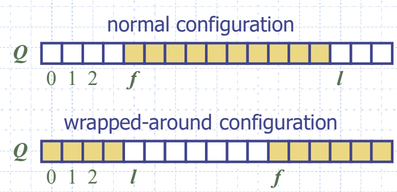
Performance Improvement

- If we use the array in a circular fashion, ***insertAtRank***(0) and ***removeAtRank***(0) run in $O(1)$ time
- Also, *insertFirst*(e) and *insertLast*(e) will run in $O(1)$ time
- Similarly S.*remove*(S.*first*()) and S.*remove*(S.*last*()) will 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

17

Circular Array-based Sequence

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear
  - f index of the first element
  - l index immediately past the last element
- Array location l is kept empty



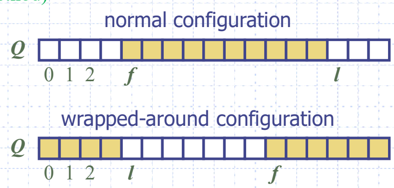
18

Circular Array-based Sequence

- Use an array V of size N
- Operation *elemAtRank*(r) is implemented in $O(1)$ time by returning $V[i]$ by calculating i using _*rank2index*(r) (private method)

Algorithm *#rank2index*(r)
  return $(f + r) \bmod N$

Algorithm *elemAtRank*(r)
  $i \leftarrow$ *#rank2index*(r)
  return $V[i]$



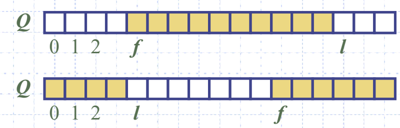
19

Circular Array-based Sequence Operations

- We use the modulo operator

Algorithm *size*()
  return $(N - f + l) \bmod N$

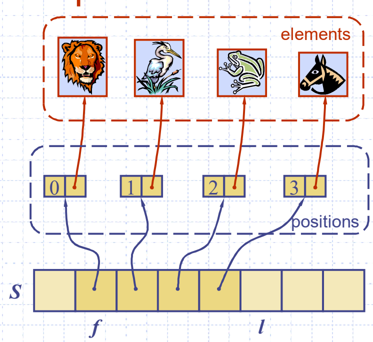
Algorithm *isEmpty*()
  return $(f = l)$



20

Array-based Implementation with Position Operations

- We use a circular array storing positions
- A position object stores:
  - Element
  - Rank (actually the index!)
- Indices f and l keep track of first and last positions



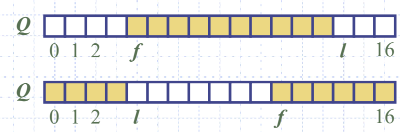
21

(#)Helper Functions

- We use the modulo operator (similar to remainder of division of natural numbers)
- Used to convert between rank and index in O(1) time

Algorithm *#index2rank*(i)
  return $(N - f + i) \bmod N$

Algorithm *#rank2index*(r)
  return $(f + r) \bmod N$



22

JavaScript Position as used in Sequences

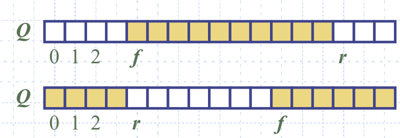
```
class APos {
  #elem;
  #index;
  constructor(elem, index) {
     this.#elem = elem;
     this.#index = index;
  }
  element() {
     return this.#elem;
  }
}
class Sequence {
…
  #index2rank(i) {
     return (this.#arr.length - this.#first + i) % this.#arr.length;
  }
  #rank2index(r) {
     return  (this.#first + r) % this.#arr.length;
  }
```


23

Circular Array-based Sequence Operations (cont.)

- Operation insertFirst throws an exception if the sequence is full
- This exception is specified in the queue ADT

Algorithm *insertFirst*(e)
  if *size*() $= N - 1$ **then**
    throw *FullSeqException*
  else
    $f \leftarrow (N + f - 1) \bmod N$
    $Q[f] \leftarrow (e, f)$ {insert Position}

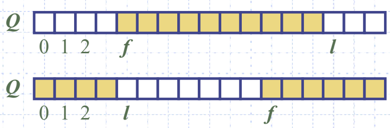


24

Sequence Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

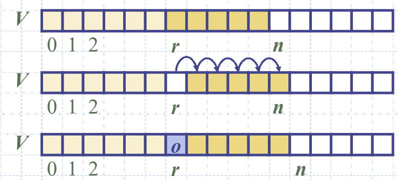
Algorithm *insertLast(e)*
  if *size*() = $N - 1$ **then**
    throw *FullSeqException*
  else
    $Q[l] \leftarrow (e, l)$ {insert Position}
    $l \leftarrow (l + 1) \bmod N$



25

Insertion

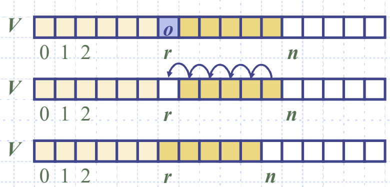
- In operation *insertAtRank*(r, o), we need to make room for the new element by shifting forward the $n - r$ elements $V[r], \ldots, V[n-1]$ if $n/2 \le r$
- In the worst case ($r = n/2$), this takes $O(n/2)$ time since we shift left if r < n/2 and shift right if $n/2 \le r$



26

Deletion

- In operation *removeAtRank*(r), we need to fill the hole left by the removed element by shifting backward the $n - r - 1$ elements $V[r + 1], \ldots, V[n - 1]$ $n/2 < r$
- In the worst case ($r = 0$), this takes $O(n)$ time



27

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.

28

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:
  - 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)

29

Random Access in a List ADT

- A List stores a sequence of elements
- Random element access is also based on the concept of Rank
  - Rank is the number of elements that precede an element in the list
- We want to be able to access, insert, or remove an element by specifying the rank (the number of elements that precede that element)
- An exception is thrown if an incorrect rank is specified (e.g., a negative rank)

30

List ADT (with Rank-based methods)

- The List ADT has the same methods as the Sequence ADT
- 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 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)

31

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
  - small database (e.g., address book)
- Indirect applications:
  - Building block of more complex data structures

32

List vs. Sequence Complexity

Rank Operations	Seq	List
size, isEmpty		
atRank(r), elemAtRank(r)		
replaceAtRank(r, o)		
insertAtRank(r, o), removeAtRank(r, o)		

33

List vs. Sequence Complexity

Rank Operations	Seq	List
size, isEmpty	1	1
atRank(r), elemAtRank(r)	1	r
replaceAtRank(r, o)	1	r
insertAtRank(r, o), removeAtRank(r)	$n/2$	r

34

List vs. Sequence Complexity

Position Operations	Seq	List
first(), last()		
before(p), after(p)		
replaceElement(p, o), swapElements(p, q)		
insertFirst(o), insertLast(o)		
insertAfter(p, o), insertBefore(p, o)		
remove(p)		
rankOf(p)		

35

List vs. Sequence Complexity

Position Operations	Seq	List
first(), last()	1	1
before(p), after(p)	1	1
replaceElement(p, o), swapElements(p, q)	1	1
insertFirst(o), insertLast(o)	1	1
insertAfter(p, o), insertBefore(p, o)	$n/2$	1
remove(p)	$n/2$	1
rankOf(p)	1	n

36

Exercise on Sequence 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:

- Write a method to calculate the average of the integers in a list of integers
  - Assume that an integer is stored in each node

Algorithm average(L)

37

Exercise on Sequence 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)
  - element elemAtRank(r)
  - integer rankOf(p)
  - position atRank(r)
- Update methods:
  - swapElements(p, q)
  - element replaceElement(p, o)
  - insertFirst(o)
  - insertLast(o)
  - insertBefore(p, o)
  - insertAfter(p , o)
  - element remove(p)

Exercise:

- Given a Sequence 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, return element e such that there is one more element that occurs after e than before
  - 1. Return the element e that is in middle; implement this without using a counter of any kind or any of the Random Access operations
  - 2. Do again using the rank-based methods
  - Analyze the complexity of your solution

Algorithm removeMiddleElem(L)

38

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

39

Main Point

2. The *List* and *Sequence* ADTs are abstractions of the same conceptual idea, a mathematical sequence. Thus the operations have the same specifications; however, their running times differ because they are based on different concrete implementations. The correct choice of which ADT to use depends on knowledge of the implementation strategies employed by both data structures.

   Science of Consciousness: Similarly, pure awareness is an abstraction of individual awareness; spontaneous correct action depends on knowledge and experience of unbounded pure awareness. Through regular practice of the TM technique, the full realization of the qualities of pure consciousness becomes more and more of a reality resulting in thoughts and actions that are correct in the sense that they are supported by and are in accord with natural law for maximum benefit to individual and society.

40

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.

41

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.

42