COSC 2011 Section N

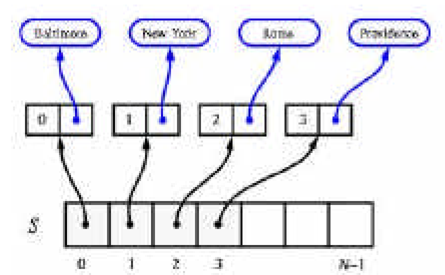
Tuesday, March 13 2001

Overview

- Assignment 1 Notes
- Array Based Implementations
  - ◆“Wrap-around” Queue, Vector, Sequence
- Linked Lists
  - ◆“Singly” & “Doubly” Linked
  - ◆Stack, Queue, Dequeue, “List”

Sequence: Array (1)

- Represent a *position* p with a new object:
  - ◆Each array element stores one object.
  - ◆Each object holds an element and an index i.



Array Based Implementations: Queue

- ◆Each time we increment f or r, compute the increment as:

  f = (f + 1) *mod* N

  r = (r + 1) *mod* N

- ◆Now have O(1) time enqueue and dequeue
- ◆Insists that Q can never hold more than N-1 elements.
- ◆Computing Size of the Queue:

  (N – f + r) *mod* N

Assignment 1 Notes: (1)

- Class Invariant:
  - ◆How the variables represent the data structure
  - ◆Assertion which should hold after initialization
  - ◆ Should be preserved by the operations on the data structure.
  - ◆E.g. Stack has top variable:
    - ⋆Initialized to –1.
    - ⋆S[top] always refers to the top element.

Assignment 1 Notes: (2)

- Assertions:
  - A logical condition that is assumed to hold at some point of program execution.
  - Precondition:
    - Some condition before the start of execution.
    - May not have any!
  - Postconditions:
    - If precondition is true and code is executed, these conditions will now hold:

Assignment 1 Notes: (3)

- Notation:

$$\{P\} \rightarrow S \rightarrow \{Q\}$$

If ***precondition*** P is true before execution of statements S, then the ***postcondition*** Q will be true after execution.

Sequence: Array (2)

- When inserting:
  - Need to shift position objects to make room for new position
- When deleting:
  - Shift position objects to fill hole created by removal of old position.
- *O(N)* Time when inserting or deleting!

Sequence: Comparison (1)

- Comparison of Sequence Implementations:
  - Array vs. Linked List

Operations	Array	List
size, isEmpty	$O(1)$	$O(1)$
atRank, rankOf, elemAtRank	$O(1)$	$O(n)$
first, last	$O(1)$	$O(1)$
before, after	$O(1)$	$O(1)$
replaceElement, swapElements	$O(1)$	$O(1)$
replaceAtRank	$O(1)$	$O(n)$
insertAtRank, removeAtRank	$O(n)$	$O(n)$
insertFirst, insertLast	$O(1)$	$O(1)$
insertAfter, insertBefore	$O(n)$	$O(1)$
remove	$O(n)$	$O(1)$

Sequence: Comparison (2)

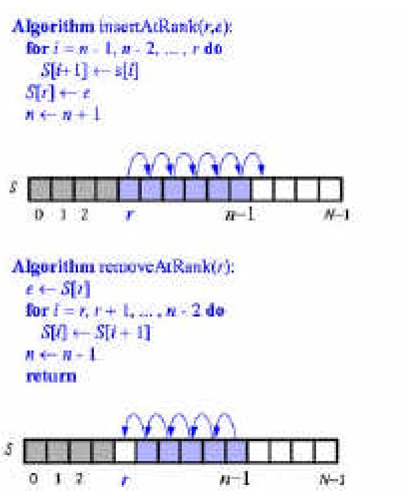
- Array imp. is superior for ranked based operations
  - ◆atRank, rankOf, elemAtRank
  - ◆Equal to linked list with other access methods.
- Linked list is superior for position based operations
  - ◆insertAfter, insertBefore, remove.
- Linked list is superior for space usage!
  - ◆O(n) vs. O(N)

Vector: Array (1)

- Obvious Choice:
  - ◆Use array A where A[i] stores reference to element with rank i.
  - ◆Choose array size N large!
  - ◆Maintain number of elements n < N in array.
  - ◆Need to shift elements in the array “up or down” to keep array cells contiguous.

Vector: Array (2)

- Method Implementations:



Vector: Array (3)

- ◆How about *elemAtRank*(r)?
- ◆When inserting or deleting:
  - ⋆Shift elements up or down to keep array cells contiguous!
  - ⋆Needed to maintain rule of always storing an element of rank I at index i in A.

Vector: Array (4)

- Running Time of Several Methods:

Method	Time
size	$O(1)$
isEmpty	$O(1)$
elemAtRank	$O(1)$
replaceAtRank	$O(1)$
insertAtRank	$O(n)$
removeAtRank	$O(n)$

  - insertAtRank(r, e): Worst case running time when r = 0.
  - removeAtRank(r, e): Worst case running time when r = 0.

Vector:Extendable Array (1)

- Regular Array Implementation Assumes Fixed Capacity!
  - Defeats the purpose of vector.
  - Waste of space or lack of space!
- Easy, Why not *Grow* the Array!.
  - Array capacity in Java is fixed – **Can't grow Array.**

Vector:Extendable Array (2)

- When ***Overflow*** occurs:
  1. Allocate new array of size 2N.
  2. Copy A[i] to B[i], for all between 0 and N - 1
  3. Let A = B. Now B supports the Vector.
  - This is an Extandable Array.
    - Extend the original array to make room for more elements.
    - Can grow by any size, not necessarily N, 2N etc.

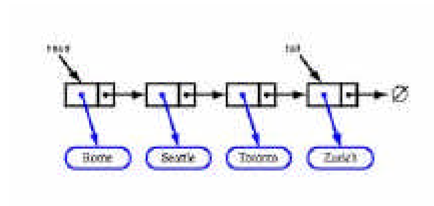
Vector:Extendable Array (3)

- Increasing the Array Size:
  - May seem slow since.
  - After array size is increased by N, space for an additional N elements.
  - Running time on a series of operations on an initially empty vector is actually quite efficient!

Singly Linked Lists (1)

- Definition:
  - Collection of ***nodes*** forming a linear ordering.
  - Ordering is determined as "Follow the Leader"
  - Each node stores a reference to an element and a reference, called ***next***, to another node.

Singly Linked Lists (2)



- ***null*** object denoted by ∅ and is pointed to by last node indicating end of list.
- "***head***" reference is a single instance variable.

Singly Linked Lists (3)

- ***next*** Reference:
  - ***Link*** or ***pointer*** to another node.
- ***Link Hopping***:
  - Moving from node to node, using next pointer.
- ***Head***: First node in list.
- ***Tail***: Last node in list and has a null next pointer.

Singly Linked Lists (4)

- As an array, keeps the elements in a certain linear order
- No fixed size
  - Uses space proportional to the number of its elements.
  - Space usage: O(n)
- Insert/Delete element at the head of the List in O(1) time.

Singly Linked Lists (5)

- Inserting at head of list:
  - Create a new node.
  - Set its ***next*** pointer to refer to same object as current head of list.
  - Set the head of list to point to new node.
- Inserting at tail of list:
  - Create a new node.
  - Set its ***next*** pointer to null.
  - Set next pointer of tail to point to new node.
  - Assign tail reference to new node.

Singly Linked Lists (6)

- Deleting a Node:
  - Can’t delete the tail node in O(1) time.
    - ⋆ Need to access node before the tail!
    - ⋆ Only way to do this, is to start from head of list and traverse entire list.

Linked List: Stack (1)

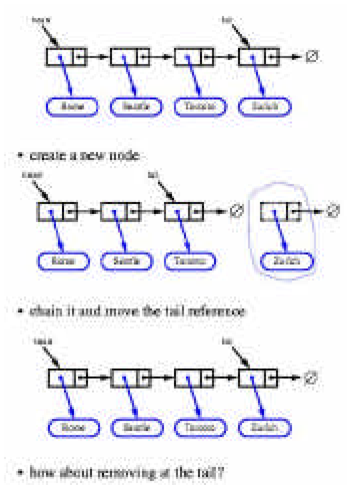
- Implementation:
  - Top of stack can be head or tail of list
    - ⋆ Since we can insert & delete at the head of list, make head the top .
  - Space Requirement: O(n)
  - No space limitations (bounded only by memory).

Linked List: Queue (1)

- Implementation:
  - Front of queue where we delete only is head of list.
  - Rear of queue where we insert elements is the tail.
  - Efficient! O(1) time.
  - Why not insert at the at the head and remove at the tail?
  - Need references to head and tail of the list.

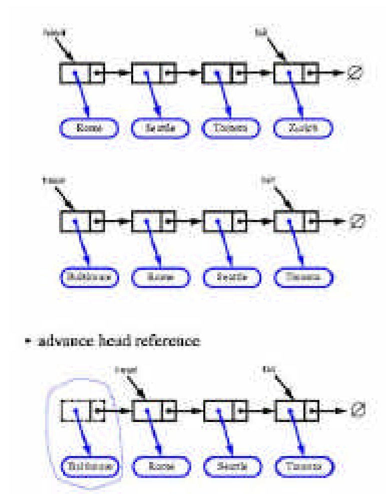
Linked List: Queue (2)

- Inserting at tail of list:



Linked List: Queue (3)

- Removing at head of list:



Doubly Linked Lists (1)

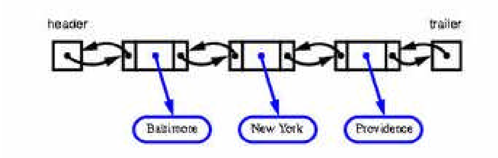
- Allows for a great variety of operations:
  - Insertion & deletion at both ends of list in O(1).
- Node stores 2 references:
  - ***next*** link – points to the next node in the list
  - ***prev*** link – points to the previous node in the list.

Doubly Linked Lists (2)

- Add special nodes at both ends of list which do not store any element:
  - ***header*** node just before the head of the list.
    - Valid ***next*** reference.
    - null ***prev*** reference.
  - ***trailer*** node just after the last node of the list.
    - Valid ***prev*** reference.
    - null ***next*** reference.

Doubly Linked Lists (3)

- Example:



Doubly Linked Lists (4)

- Inserting & removing at either end in O(1) time:
  - ***prev*** pointer eliminates need to traverse list to find node just before tail.
- Inserting & deleting anywhere else in list:
  - Simply a matter of changing the pointers!

Doubly Linked Lists (4)

- List ADT with doubly linked list:
  - Make nodes implement position interface
    - define method element() which returns element stored at node.
  - Nodes act as positions.
    - Viewed internally by list as nodes but on the outside as positions.

Doubly Linked Lists (5)

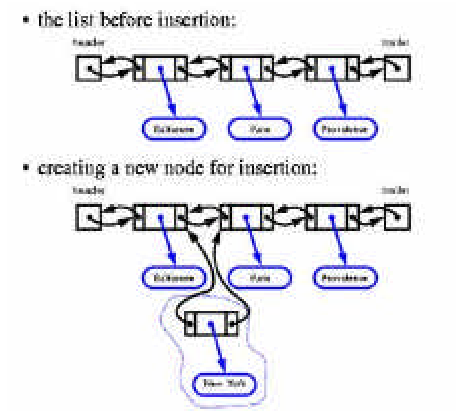
- In the internal node view:
  - Each node has instance variables ***prev*** and ***next***
    - Refer to predecessor and successor nodes respectively.
  - Given position p we cast it to a node.
    - Once we have node, can implement for example, before(p) by returning the prev reference.

Doubly Linked Lists (6)

- Supports object oriented approach additional time & space overhead:
  - Hides implementation details from user.
  - Aids in code reuse since user doesn't know that the position objects are actually treated as nodes!
  - See text book for pseudo code and Java Code!

Doubly Linked Lists (7)

- Inserting element into list:



Doubly Linked Lists (8)

- Removing element from list: