DOUBLE-ENDED QUEUES (6.3)

EECS 2011

Double-Ended Queue ADT

- Deque (pronounced “deck”)
- Allows insertion and deletion at both the front and the rear of the queue
- Deque ADT: operations

addFirst(*e*): insert *e* at the beginning of the deque

addLast(e): insert *e* at the end of the deque

removeFirst(): remove and return the first element

removeLast(): remove and return the last element

first(): return the first element

last(): return the last element

isEmpty(): return true if deque is empty; false otherwise

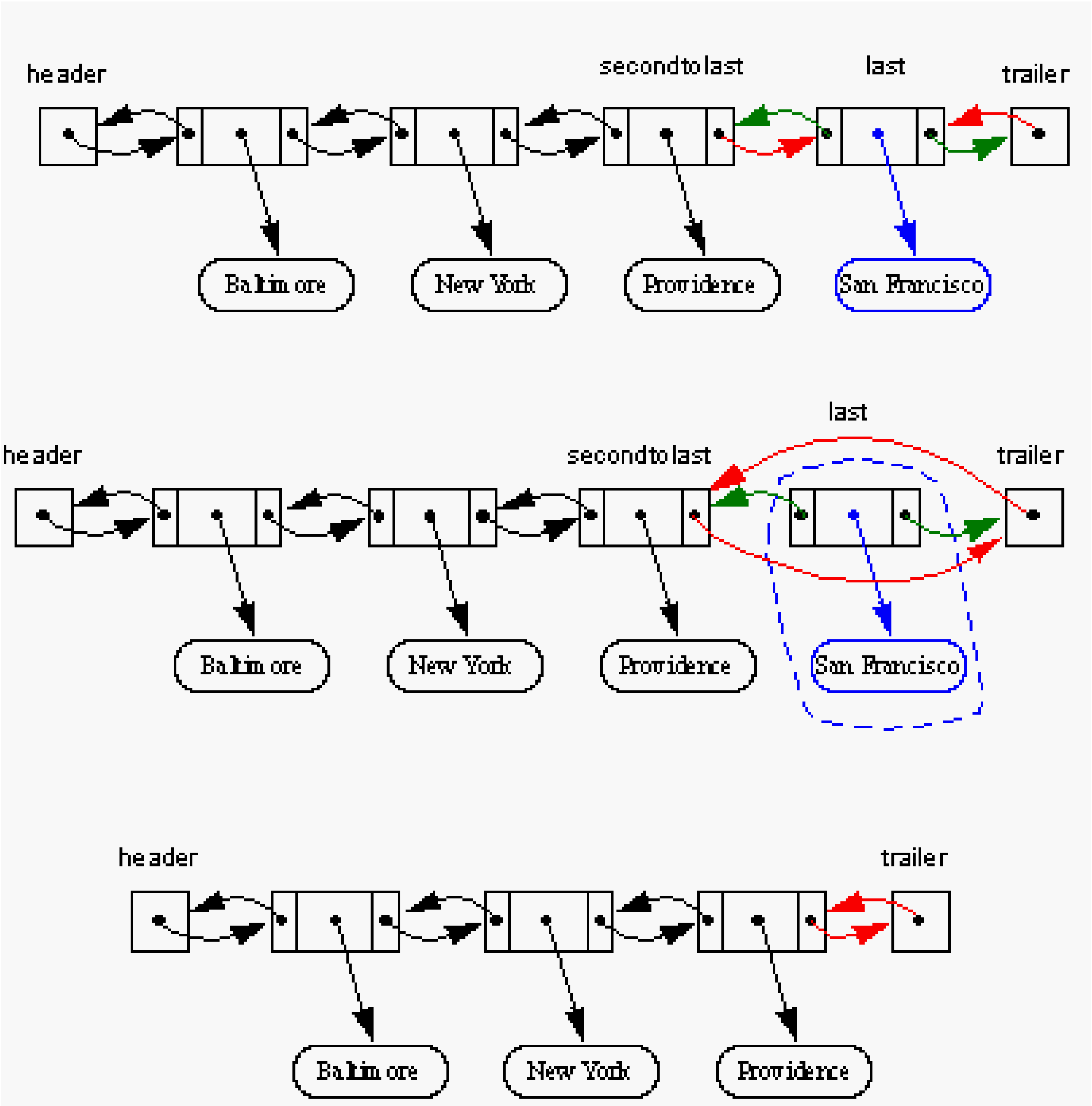
size(): return the number of objects in the deque

Implementation Choices

- Arrays
  - Similar to queue implementation
  - Incrementing an index circularly: f = (f + 1) % N
  - Decrementing an index circularly:
  - f = (f - 1) % N ➔ problem?
  - Solution: f = (f – 1 + N) % N

- Linked lists: singly or doubly linked?
  - Removing at the tail of a single linked list costs $\theta(n)$

removeLast() and *addLast*()



Implementing Stacks and Queues with Deques

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()

EXTENDABLE ARRAYS (7.2.3)

EECS 2011

Extendable Array in Java

- java.util.ArrayList and java.util.Vector use extendable arrays.
- *capacityIncrement* determines how the array grows:

  capacityIncrement = 0: array size doubles

  capacityIncrement = *c* > 0: array adds *c* new cells

- Vector object = new vector(int *initialcapacity*, *capacityIncrement*)

  Vector vec = new Vector(4, 6);
  - initial capacity = 4
  - to insert 5th element, increase vector capacity to 10.

Extendable Array Implementation

When *add*() is called and an overflow occurs ($n = N$):

- Allocate a new array T of capacity $2N$
- Copy contents of the original array V into the first half of the new array T
- Set $V = T$
- Perform the insertion using new array V

- Note: when the number of elements in the list goes below a threshold (e.g., $N/4$), shrink the array by half the current size N of the array.

Time Analysis

- "*add*": inserting an element to be the last element of a list (or top of a stack)
- *add*(*e*) {

  if (full stack) then extend the array;

  "push" *e* to new array*;*

  }
- Proposition 1:

  Let *S* be a list implemented by means of an extendable array *V* as described before. The total time to perform a series of *n* "add" operations in *S*, starting from *S* being empty and *V* having size $N = 1$, is $O(n)$.

Pseudo-code

```
int [ ] V = new int[1]; N = 1;  top = –1;
input element e;
for( i = 0; i < n; i++ ) {
  if( stack is full ) {
    allocate a new array T of capacity 2N;
    copy V[i] to T[i] for i = 0, 1, …, N–1;       // a for loop
    set V = T;
    N = N * 2;
  }
  top = top + 1;
  V[top] = e;
  input next element e;
}
```

Time Analysis (2)

1. All array extensions: O(?)

- Allocate a new array *T* of capacity 2*N*
- Copy *V*[*i*] to *T*[*i*] for i = 0, 1, …, *N*–1
- Set *V* = *T*

2. All "push" operations take O(*n*) (each "push" takes O(1))

Running time of all array extensions:

- If the array is extended *k* times, then $n = 2^k$
- The total number of copies is:

  $1 + 2 + 4 + 8 + \ldots + 2^{k-1} = 2^k - 1 = n - 1 = O(n)$

Total = $O(n) + O(n) = O(n)$

Increment Strategies

- java.util.ArrayList and java.util.Vector use extendable arrays.
- *capacityIncrement* determines how the array grows:

capacityIncrement = 0:	array size doubles
capacityIncrement = $c > 0$:	array adds c new cells

- Proposition 2:

  If we create an initially empty java.util.Vector object with a fixed positive *capacityIncrement* value, then performing a series of n "add" operations on this vector takes $\Omega(n^2)$ time.
- $\Omega(n^2)$: takes at least time n^2

Increment Strategies (2)

1. Array extensions: O(?)

- Let a be the initial size of array V
- Let $capacityIncrement = c$
- If the array is extended k times then $n = a + ck$
- The total number of copies is:
  $(a) + (a+c) + (a+2c) + \ldots + (a+(k-1)c) =$
  $ak + c(1+2+\ldots+(k-1)) = ak + ck(k-1)/2 = \theta(k^2) = \theta(n^2)$
- We infer $\Omega(n^2)$ from $\theta(n^2)$

2. All “push” operations take $O(n)$ (each “push” takes $O(1)$)

Which is the better increment strategy?

Next lecture ...

- Trees (chapter 8)