Data Structures: Stacks and Queues

Abstract Data Types (ADTs)

- An ADT is the realization of a data type as a software component
- The **interface** of the ADT is defined in terms of:
  - A type
  - A set of operations on that type
- Types of operations
  - Constructors
  - Access functions
  - Manipulation functions
- The behavior of each operation is determined by its inputs and outputs
- An ADT does not specify **how** the data type is implemented

Dynamic Sets

- We will deal with ADTs, instances of which are sets of some types of elements
  - Operations are provided that change the set
- We call such class of ADTs **dynamic sets**

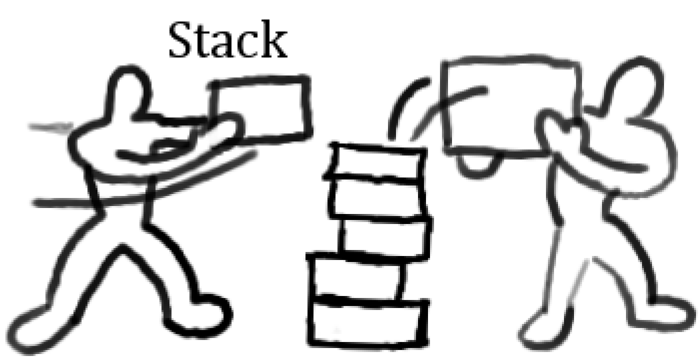
Dynamic Sets (cont.)

- An example of dynamic set ADT
  - Methods:
    - New():ADT
    - Insert(S:ADT, v:element):ADT
    - Delete(S:ADT, v:element):ADT
    - IsIn(S:ADT, v:element):boolean
  - Insert and Delete – “manipulation function” methods
  - IsIn – “access function” method
  - Some examples:
    - IsIn(New(), v) = false
    - IsIn(Insert(S, v), v) = true
    - IsIn(insert(S, u), v) = IsIn(S, v), if $v \neq u$

Stacks

- A **stack** is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle
- Objects can be inserted at anytime, but only the last (the most-recently inserted) object can be removed

- Inserting an item is known as "**pushing**" onto the stack
- "**Popping**" off the stack is synonymous with removing an item.



Stacks (cont.)

- A stack is an ADT that supports four main methods:
  - New():ADT – creates a new stack
  - Push(S:ADT, v:element):ADT – inserts object v onto the top of stack S
  - Pop(S:ADT):ADT – removes the top object of stack S; if the stack is empty an error occurs
  - Top(S:ADT):element – returns the top object of the stack without removing it; if the stack is empty an error occurs

Java Implementation

- Given the stack ADT, we need to code the ADT in order to use it in programs ⇒ we need **interfaces**
- An **interface** is a way to declare what a class is to do, but it does not mention how to do it
  - For an interface, we just write down the method names and the parameters. When specifying parameters what really matters is their types.
  - Later, when we write a class for that interface, we actually code the content of the methods
  - Separating interface and implementation is a useful programming technique

Java Implementation (cont.)

- The stack data structure is a "built-in" class of Java's java.util package, but we define our own stack interface:

```
public interface Stack{
    // access methods
    public int size();
    public boolean isEmpty();
    public Object top() throws StackEmptyException;

    // manipulation methods
    public void push(Object element);
    public Object pop() throws StackEmptyException;
}
```

Array-Based Stack in Java

- Create a stack using an array by specifying a maximum size N for our stack
- The stack consists of an N-element array S and an integer variable t, the index of the top element in array S.

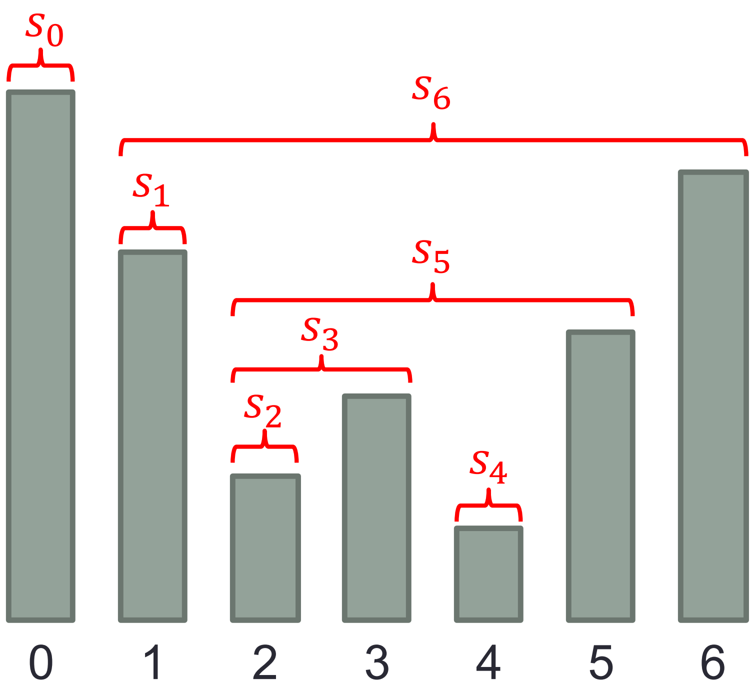
- Array indices start at 0, so we initialize t to -1
- See “ArrayStack.java” for the implementation.

Array-Based Stack in Java (cont.)

- The array implementation is simple and efficient (method performed in $O(1)$)
- There is an upper bound, N, on the size of the stack. The arbitrary value N may be too small for a given application, or a waste of memory.
- `StackEmptyException` is required by the interface
- `StackFullException` is particular to this implementation

Application: Time Series

- The **span** $\boldsymbol{s_i}$ of a stock's price on a certain day i is the maximum number of consecutive days (up to the current day) the price of the stock has been less than or equal to its price on day i



An Inefficient Algorithm

- Algorithm **computeSpans1(P)**:
  - Input: an n-element array P of numbers such that P[i] is the price of the stock on day i
  - Output: an n-element array S of numbers such that S[i] is the span of the stock on day i

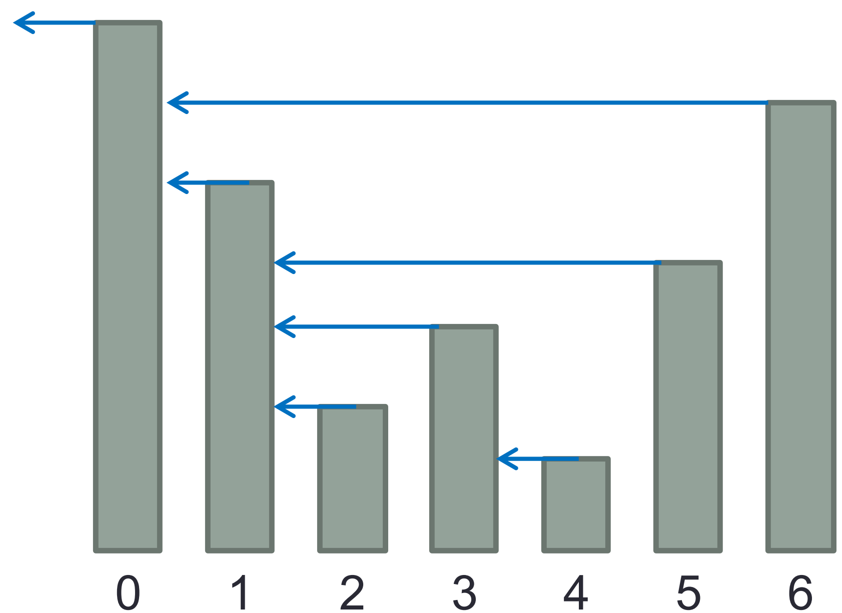
```
for i ← 0 to n-1 do
    k ← 0
    done ← false
    repeat
        if P[i-k]≤P[i] then k ← k+1
        else done ← true
    until (k>i) or done
    S[i] ← k
return S
```

- The running time of this algorithm is $O(n^2)$. Why?

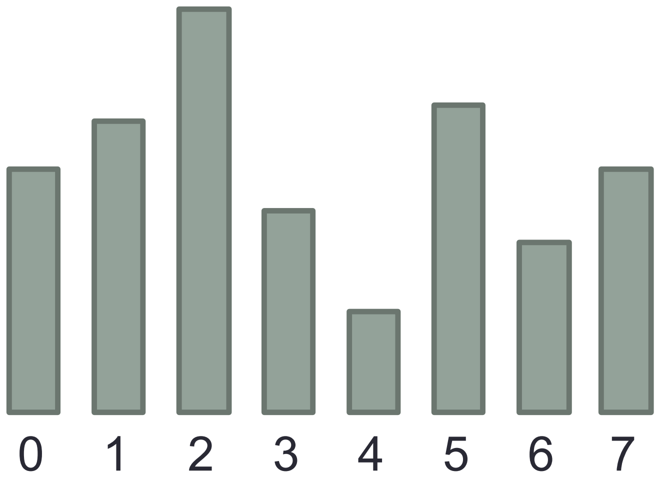
A Stack Can Help

- s_i can be easily computed if we know the closest day preceding i, on which the price is greater than the price on day i. If such a day exists, let's call it $h(i)$, otherwise, we conventionally define $h(i) = -1$

- In the figure, $h(3) = 1$, $h(4) = 3$, and $h(6) = 0$
- The span is now computed as $s_i = i - h(i)$



What to do with the Stack?



- What are the possible value of $h(7)$? Can it be 0 or 1 or 3 or 4?
- No, $h(7)$ can only be 2 or 5 or 6.

- We store indices 2, 5, 6 in the stack
- To determine $h(7)$ we compare the price on day 7 with the price on day 6, day 5, day 2 in that order
- The first price, larger than the price on day 7, gives $h(7)$
- The stack should be updated to reflect the price of day 7
- It should now contains 2, 5, 7

An Efficient Algorithm

- Algorithm **computeSpans2(P)**:
  - Let D be an empty stack

```
for i ← 0 to n-1 do
    k ← 0
    done ← false
    while not(IsEmpty(D) or done) do
        if P[i]≥P[Top(D)] then Pop(D)
        else done ← true
    if IsEmpty(D) then h ← -1
    else h ← Top(D)
    S[i] ← i-h
    Push(D, i)
return S
```

Growable Array-Based Stack

- Instead of giving up with a StackFullException, we can replace the array S with a larger one and continue processing push operations
  - Algorithm push(S, v)

```
if Size(S)=N then A ← new array length f(N)
for i ← 0 to N-1
    A[i] ← S[i]
S ← A
t ← t+1
S[t] ← v
```

- How large should the new array be?
  - tight strategy (add a constant): $f(N) = N + c$
  - growth strategy (double up): $f(N) = 2N$

Tight vs Growth Strategies

- To compare the two strategies, we use the following cost model:
  - A regular push operation: adds one element (costs one unit)
  - A special push operation: creates array of size $f(N)$, copies N elements, and adds one element (costs $f(N) + N + 1$ units)

Performance of the Tight Strategy

- In phase i the array has size $c \times i$
- Total cost of phase i is
  - $c \times i$ is the cost of creating the array
  - $c \times (i - 1)$ is the cost of copying elements into new array
  - c is the cost of the c pushes
- Hence, cost of phase i is $2 \times c \times i$
- In each phase we do c pushes. Hence for n pushes we need n/c phases. Total cost of these n/c phases:

$$2c\left(1 + 2 + 3 + \cdots + \frac{n}{c}\right) \text{ is in } O(n^2)$$

Performance of the Growth Strategy

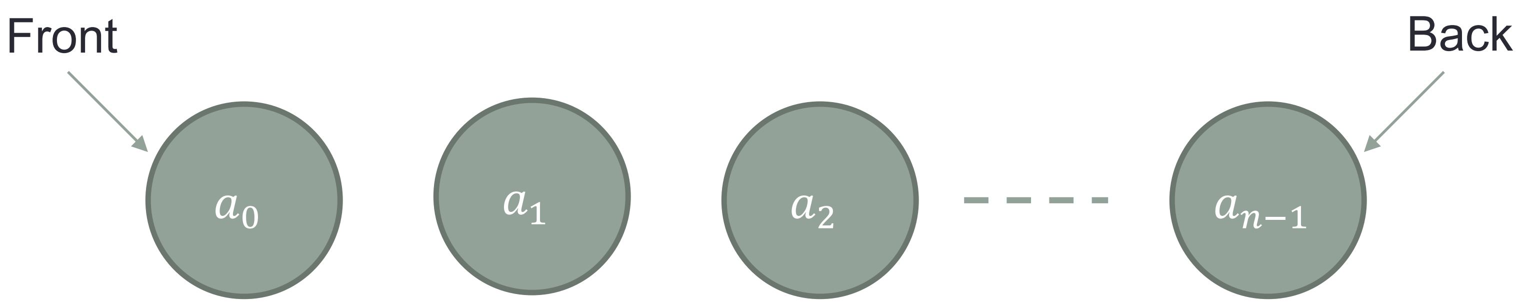
- In phase i the array has size 2^i
- Total cost of phase i is
  - 2^i is the cost of creating the array
  - 2^{i-1} is the cost of copying elements into new array
  - 2^{i-1} is the cost of the 2^{i-1} pushes done in this phase
- Hence, cost of phase i is 2^{i+1}
- If we do n pushes, we will have $\log_2 n$ phases
- Total cost of n pushes:

$$2 + 4 + 8 + \cdots + 2^{\log_2 n + 1} \text{ is in } O(n)$$

The growth strategy wins!

Queues

- A queue differs from a stack in that its insertion and removal routines follows the first-in-first-out (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the back (enqueued) and removed from the front (dequeued)



Queue ADT

- The queue supports these fundamental methods:
  - New():ADT – Creates and empty queue
  - Enqueue(S:ADT, v:element):ADT – Inserts object v at the back of the queue
  - Dequeue(S:ADT):ADT – Removes the object from the front of the queue; an error occurs if the queue is empty
  - Front(S:ATD):element – Returns, but does not remove the front element; an error occurs if the queue is empty

Queue ADT (cont.)

- These support methods should also be defined:
  - Size(S:ADT):integer
  - IsEmpty(S:ADT):Boolean
- Some examples:
  - Front(Enqueue(New(), v)) = v
  - Dequeue(Enqueue(New(), v)) = New()
  - Front(Enqueue(Euqueue(Q, w), v)) = Front(Enqueue(Q, w))
  - Dequeue(Enqueue(Enqueue(Q, w), v))

    = Enqueue(Dequeue(Enqueue(Q, w)), v)

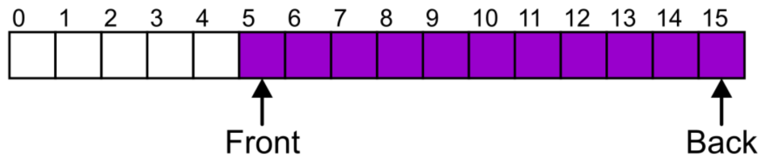
An Array Implementation

- A maximum size N is specified
- The queue consists of an N-element array Q and two integer variables:
    - f – index of the front element (head for dequeue)
    - b – index of the element after the back one (tail for enqueue)

- What does f = b means?

Circular Arrays

- Suppose that:
  - The array capacity is 16
  - We have performed 16 enqueues
  - We have performed 5 dequeues
    - The queue size is now 11



  - We perform one further enqueue

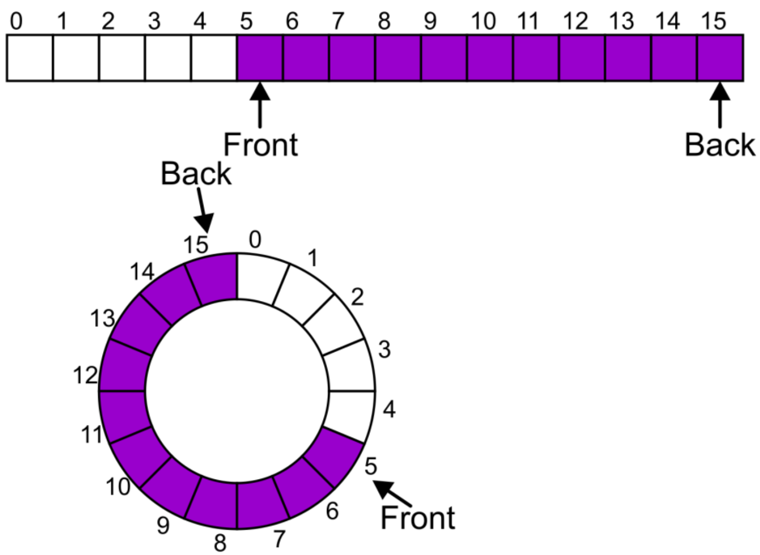
- In this case, the array is not full and yet we cannot place any more objects in to the array

Circular Arrays (cont.)

- Instead of viewing the array on the range 0, …, 15, consider the indices being cyclic:

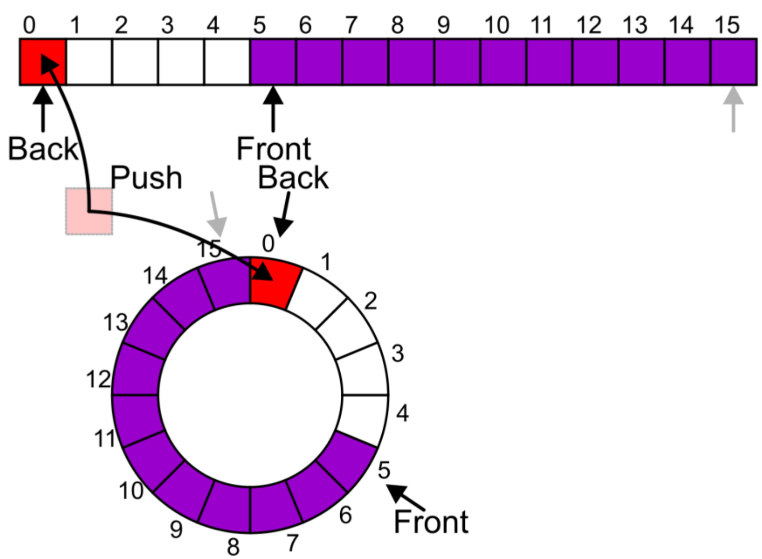
$$\ldots, 15, 0, 1, \ldots, 15, 0, 1, \ldots, 15, 0, 1, \ldots$$

- This is referred to as a *circular array*



Circular Arrays (cont.)

- Now the next enqueue may be performed in the next available location of the circular array



Pseudo Code

```
Algorithm Size()
return (N-f+b) mod N
```

```
Algorithm IsEmpty()
return (f=b)
```

```
Algorithm Front()
if IsEmpty() then
    return QueueEmptyException
return Q[f]
```

```
Algorithm Dequeue()
if IsEmpty() then
    return QueueEmptyException
Q[f] ← null
f ← (f+1) mod N
```

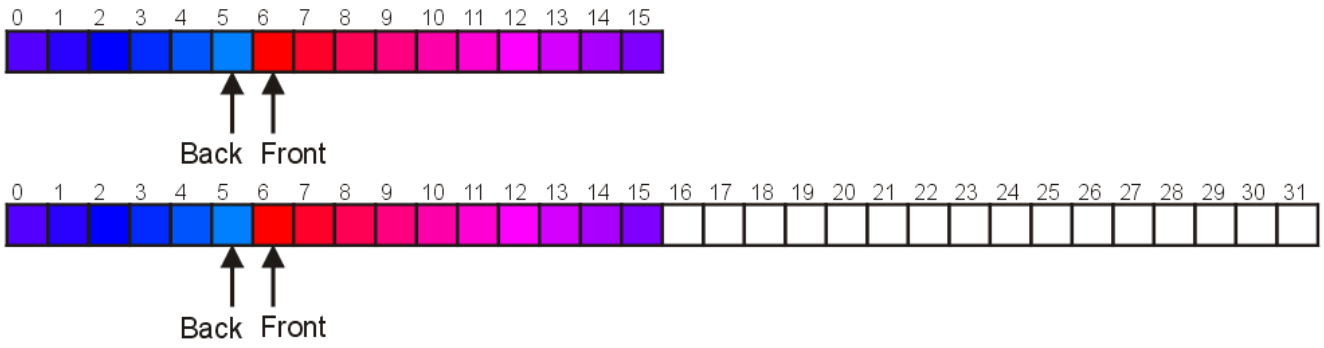
```
Algorithm Enqueue(v)
if Size() = N-1 then
    return QueueFullException
Q[b] ← v
b ← (b+1) mod N
```

Full Exceptions

- As with a stack, there are a number of options which can be used if the array is filled
- If the array is filled, we have the following options:
  - Throw an exception
  - Ignore the element being enqueued
  - Increase the size of the array

Increasing Capacity

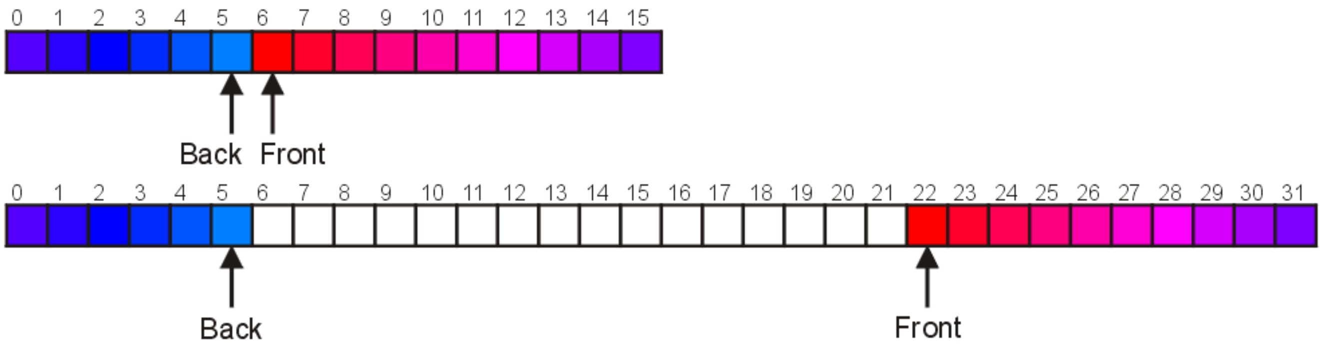
- Unfortunately, if we choose to increase the capacity (by growth strategy), this becomes slightly more complex
  - A direct copy does not work:



- There are two solutions

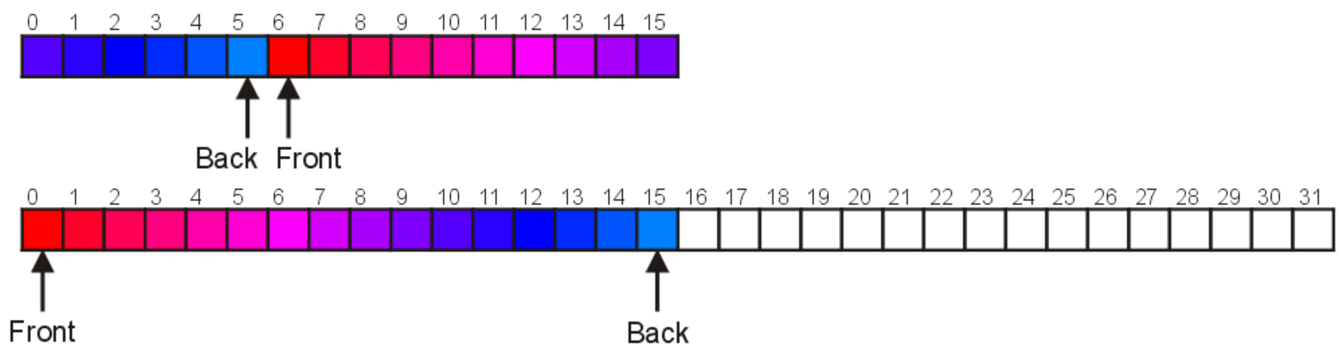
Increasing Capacity (cont.)

- The first solution:
  - Move those beyond the front to the end of the array
  - The next enqueue would then occur in position 6



Increasing Capacity (cont.)

- An alternate solution is normalization:
  - Map the front back at position 0
  - The next enqueue would then occur in position 16



References

- Clifford A. Shaffer. (2012). *Data Structures and Algorithm Analysis.* Department of Computer Science. Virginia Tech.
- Kurt Mehlhorn and Peter Sanders. (2007). *Algorithms and Data Structures.* Springer.
- Naveen Garg. (2008). *Data Structures and Algorithms*. Department of Computer Science and Engineering. Indian Institute of Technology, Delhi.
- Robert Lafore. (2003). *Data Structures & Algorithms in Java*. Sams Publishing.