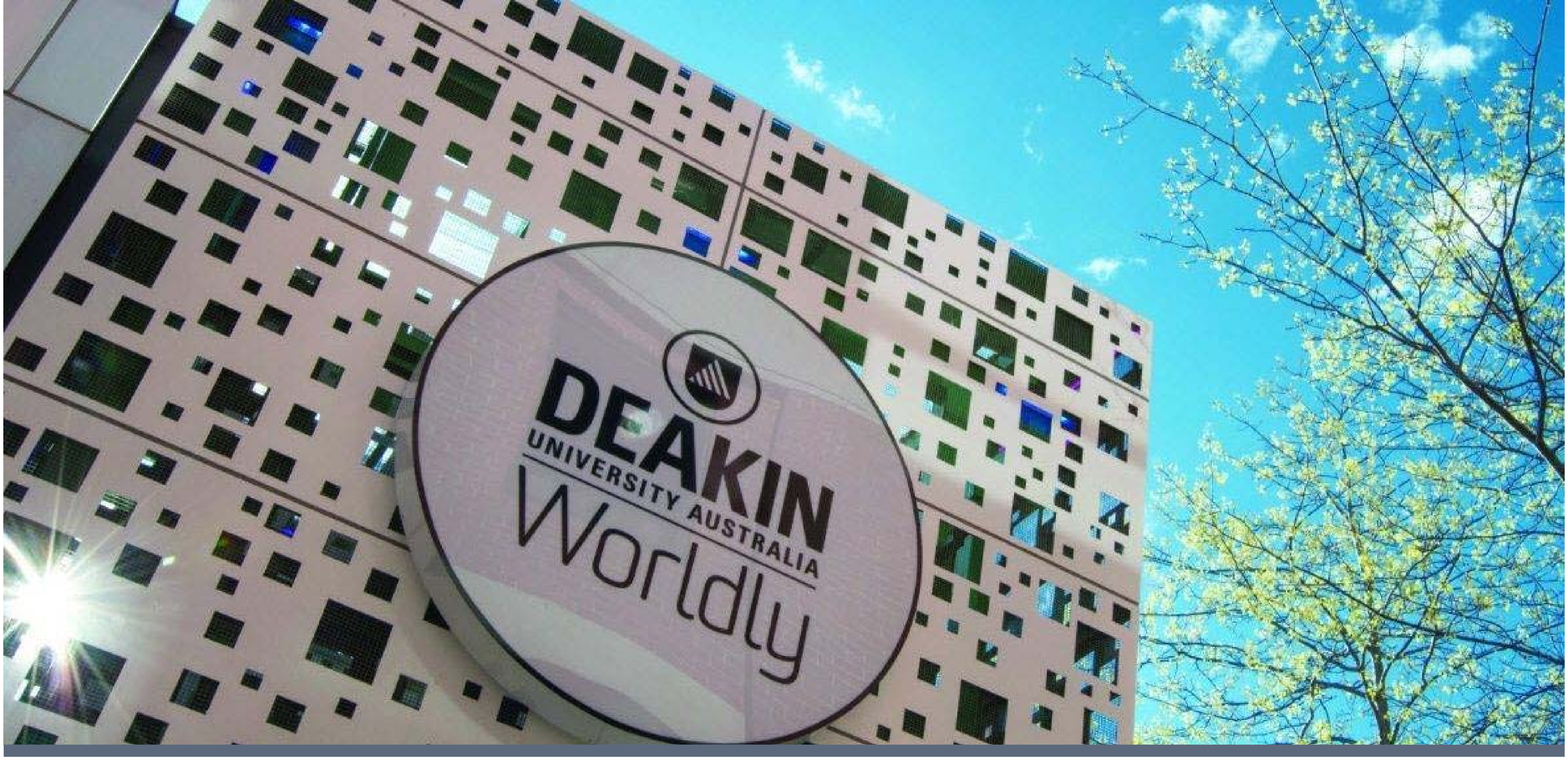


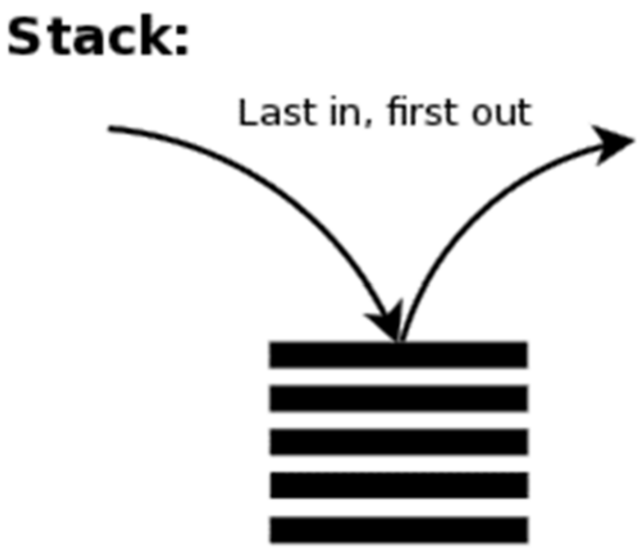
Lecture 5. Stacks and Queues

SIT221 Data Structures and Algorithms

Stacks: Definition

A stack is a data structure that retrieves data in the opposite order to which it was stored.

- Insertions and deletions follow the Last-In/First-Out (LIFO) scheme.
- You only ever have access to the top of the stack.
  Remember a stack of books? You can only grab the top one!



Stacks: Operations

The operations associated with a stack are:

- **push** inserts an element on top of the stack
- **pop** removes the top element off the stack and returns it

Auxiliary stack operations:

- **peek** returns the last inserted element without removing it
- **size** returns the number of elements stored in the stack
- **clear** empties the stack

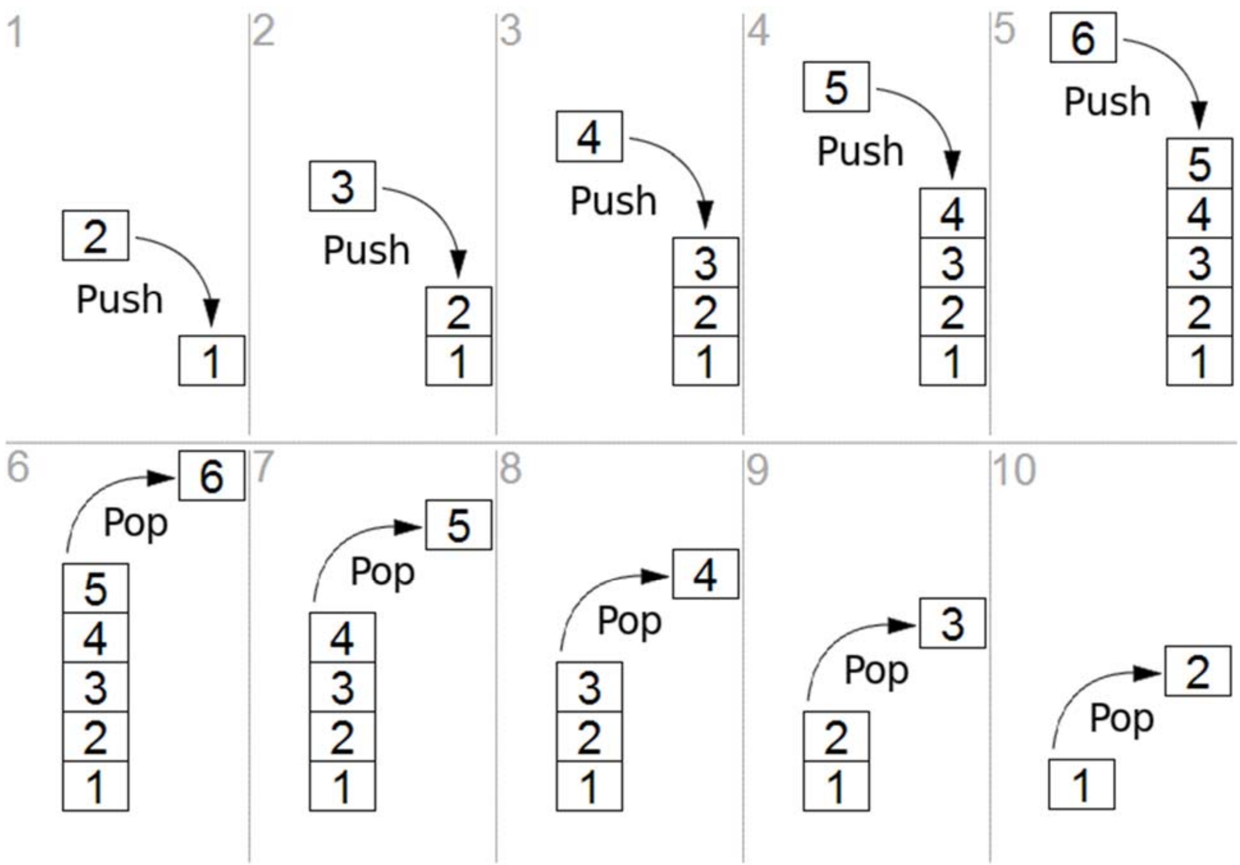
Attempting the execution of pop or peek on an empty stack throws an “EmptyStackException”-like exception.

Stacks: Operations

The operations associated with a stack are:

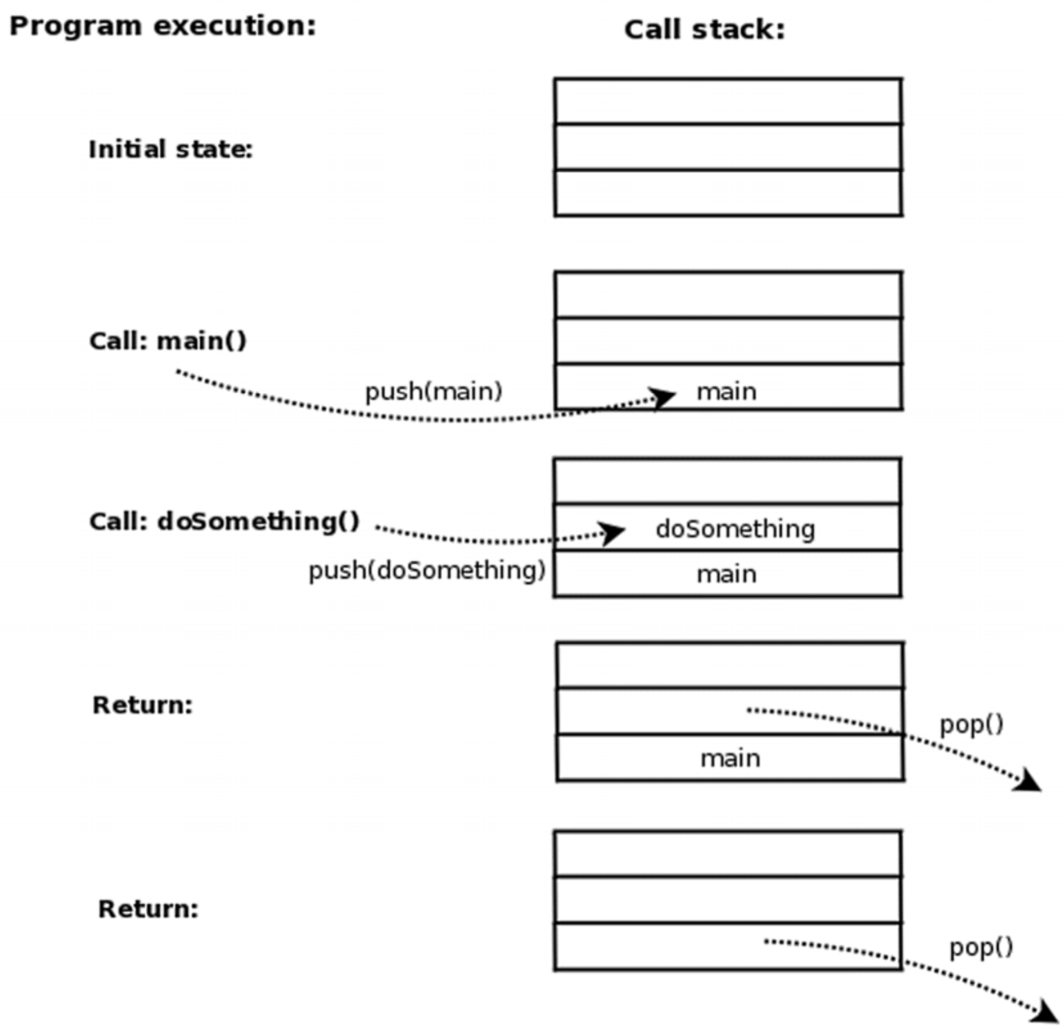
– **push** inserts an element on top of the stack

– **pop** removes the top element off the stack and returns it



Stacks: Examples

Program stack




```
procedure main ()
doSomething()

procedure doSomething ()
print "whoo!"
```

Stacks: Examples

Binary-to-Decimal Conversion

Convert 155_{10} to binary…

		Stack: top
• 155 / 2 = 77	Remainder 1	1
• 77 / 2 = 38	Remainder 1	0
• 38 / 2 = 19	Remainder 0	0
• 19 / 2 = 9	Remainder 1	1
• 9 / 2 = 4	Remainder 1	1
• 4 / 2 = 2	Remainder 0	0
• 2 / 2 = 1	Remainder 0	1
• 1 / 2 = 0	Remainder 1	1

Read the stack
for the answer

Answer = 10011011

Stacks: Applications

Direct applications:

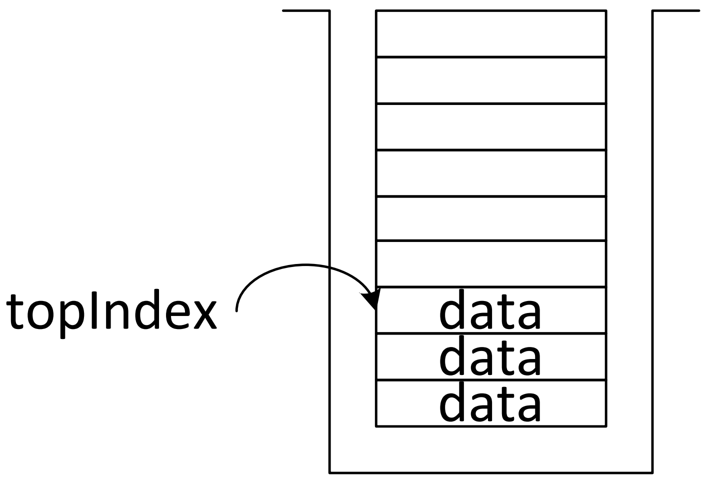
- Page-visited history in a Web browser
- Undo sequence in a text editor
- Chain of method calls in the Java Virtual Machine

Indirect applications:

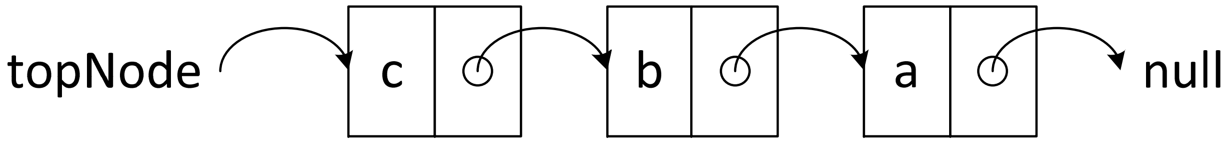
- Auxiliary data structure for algorithms
- Component of other data structures, e.g. Backtrackings & Graph Traversal

Stacks: Implementation

- Using Dynamic Arrays



- Using Linked Lists



Stacks: Array-based Stack

– A simple way of implementing the stack uses an array

– We add elements from left to right

– A variable keeps track of the index of the top element

Method **pop()**

if (S.Count == 0) **then** throw *EmptyStackException*

else

S.Count = S.Count – 1;

return S[S.Count];

Stacks: Array-based Stack

- The array storing the stack elements may become full.
  A push operation will then either increase the array's capacity or throw a "FullStackException"-like exception.
- Limitation of the array-based implementation. It's not intrinsic to the Stack.

Method **push**(new_element)

    if (S.Capacity == S.Count) **then** increase_capacity();

    else

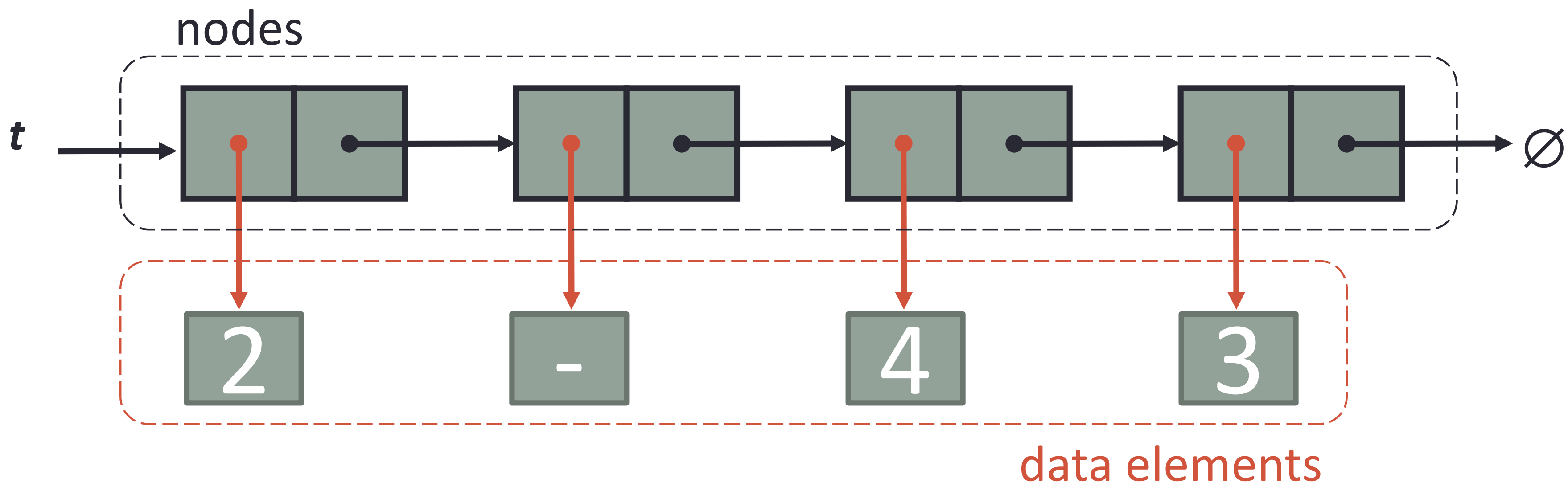
        S.Count = S.Count + 1;

        S[S.Count-1] = new_element;

Stacks: Singly Linked List-based Stack

Stacks are very easy to implement in linked lists.

- The top element is stored at the first node of the list.
- Push adds a node to the front of the list.
- Pop removes the node at the front, returns the value and destroys the old node, updating top to point to the new top.
- If top points to NULL, the stack is empty.
- The space used is $O(n)$ and each operation of the stack takes $O(1)$ time.



Stacks: Practical Examples

Parentheses Matching

Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “]”.

For example:

- correct: ( )(( )){([( )])}
- correct: ((( )(( )){([( )])}
- incorrect:)(( )){([( )])}
- incorrect: ({[ ])}
- incorrect: (

Stacks: Practical Examples

Parentheses Matching

Algorithm ParenthesesMatch(*X*, *n*):

Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

Output: **true** if and only if all the grouping symbols in *X* match

```
Let S be an empty stack
for ( i = 0 to n-1 ) do
        if ( X[i] is an opening grouping symbol ) then
                S.push(X[i]);
        else if ( X[i] is a closing grouping symbol ) then
                if ( S.Count==0 ) then
                        return false;                          // nothing to match with
                if ( S.pop() does not match the type of X[i] ) then
                        return false;                          // wrong type
if ( S.Count==0 ) then
        return true;                                           // every symbol matched
else
        return false;                                          // some symbols were never matched
```

Stacks: Practical Examples

Parentheses Matching: HTML Tag Matching

For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

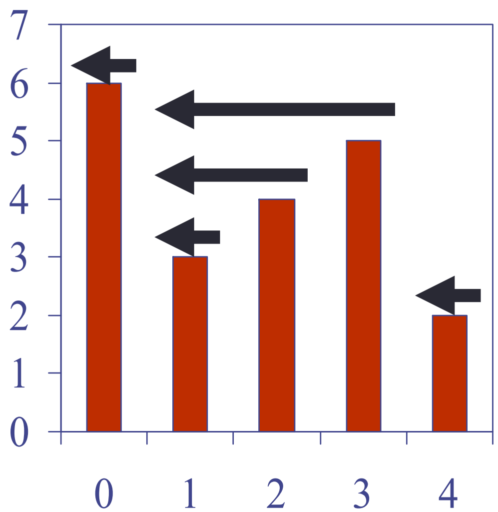
1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Stacks: Practical Examples

Computing Spans

We show how to use a stack as an auxiliary data structure in an algorithm

- Given an array X, the span S[i] of X[i] is the maximum number of consecutive elements X[j] immediately preceding X[i] and such that X[j] ≤ X[i]
- Spans have applications to financial analysis, e.g. stock at 52-week high



X	6	3	4	5	2
S	1	1	2	3	1

Stacks: Practical Examples

Computing Spans: $O(n^2)$-time algorithm

Algorithm **spans1**(X, n)

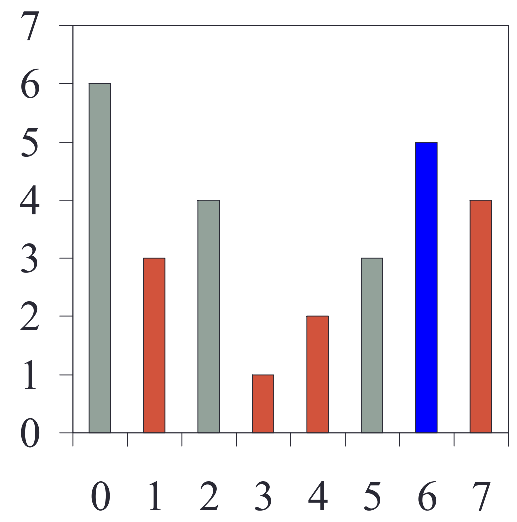
Input:	array X of n integers	
Output:	array S of spans of X	# steps

S = new array of n integers;	n
for (i = 0 to n – 1) **do**	n
s = 1;	n
while (s ≤ i **and** X[i - s] ≤ X[i]) **do**	n · (1 + 2 + …+ (n – 1))
s = s + 1;	n · (1 + 2 + …+ (n – 1))
S[i] = s;	n
return S;	1

Stacks: Practical Examples

Computing Spans with a Stack

- We keep in a stack the indices of the elements visible when "looking back"
- We scan the array from left to right
    - Let i be the current index
    - We pop indices from the stack until we find index j such that $X[i] < X[j]$
    - We set $S[i] \leftarrow i - j$
    - We push x onto the stack



Stacks: Practical Examples

Computing Spans: Linear Algorithm

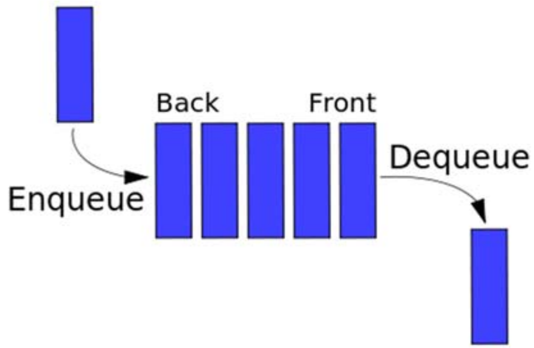
```
Algorithm spans2 ( X, n )                                        # runs
    S = new array of n integers;                                 n
    A = new empty stack;                                         1
        for ( i = 0 to n – 1 ) do                                n
            while ( A.Count ≠ 0 and X[A.top()] ≤ X[i] ) do       n
                A.pop();                                         n
            if ( A.Count == 0 ) then                             n
                S[i] = i + 1;                                    n
            else
                S[i] = i - A.top();                              n
            A.push(i);                                           n
    return S;                                                    1
```

- Each index of the array
  - Is pushed into the stack exactly once
  - Is popped from the stack at most once
- The statements in the while-loop are executed at most n times

Queues: Formulation

A queue is a data structure that retrieves data in the same order in which it was stored.

- You have access to the front of the queue, to remove things, and the back of the queue, to add things.
- This is called First-In/First-Out (FIFO).
- Think about the lineup at McDonalds. You queue from the back and are served from the front.



Queues: Operations

The operations associated with a stack are:

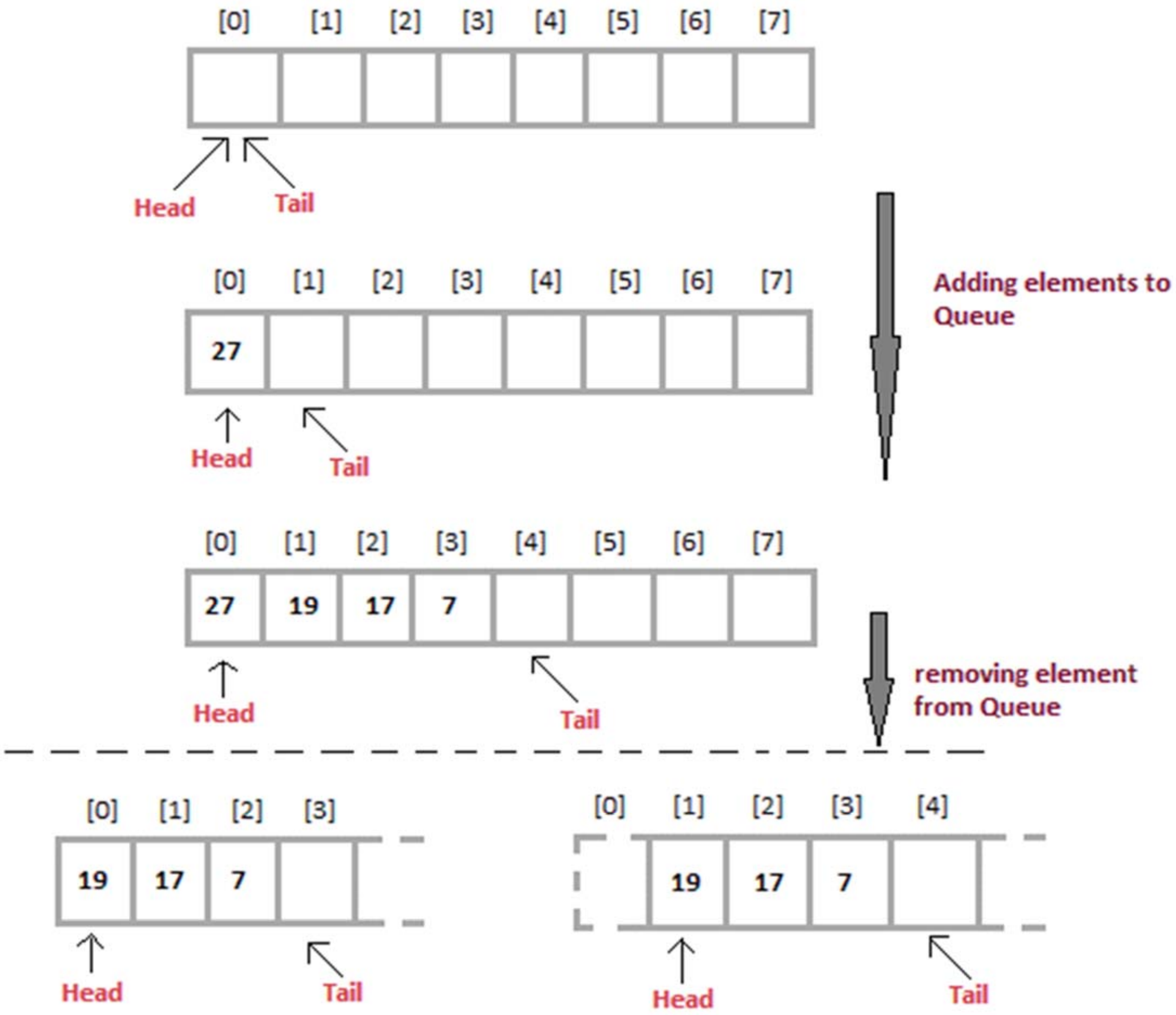
- **enqueue** inserts an element at the end of the queue
- **dequeue** removes and returns the element at the front of the queue

Auxiliary stack operations:

- **peek** returns the element at the front without removing it
- **size** returns the number of elements stored
- **clear** empties the stack

Attempting the execution of dequeue or peek on an empty queue throws an "EmptyQueueException"-like exception.

Queues: Operations



Start with an empty queue

Enque 27

Enque 19, 17, 7

Dequeue 27

Queues: Applications

Direct applications

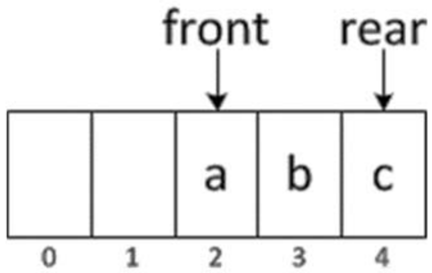
- Waiting lists, bureaucracy
- Access to shared resources (e.g., printer)
- Multiprogramming

Indirect applications

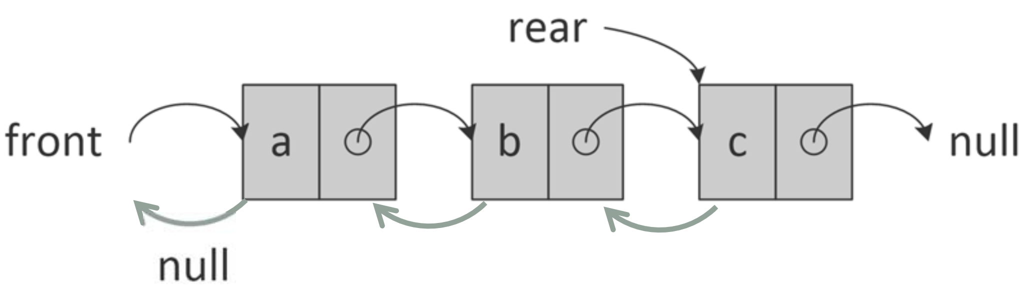
- Auxiliary data structure for algorithms
- Component of other data structures

Stacks: Implementation

- Using Dynamic Arrays

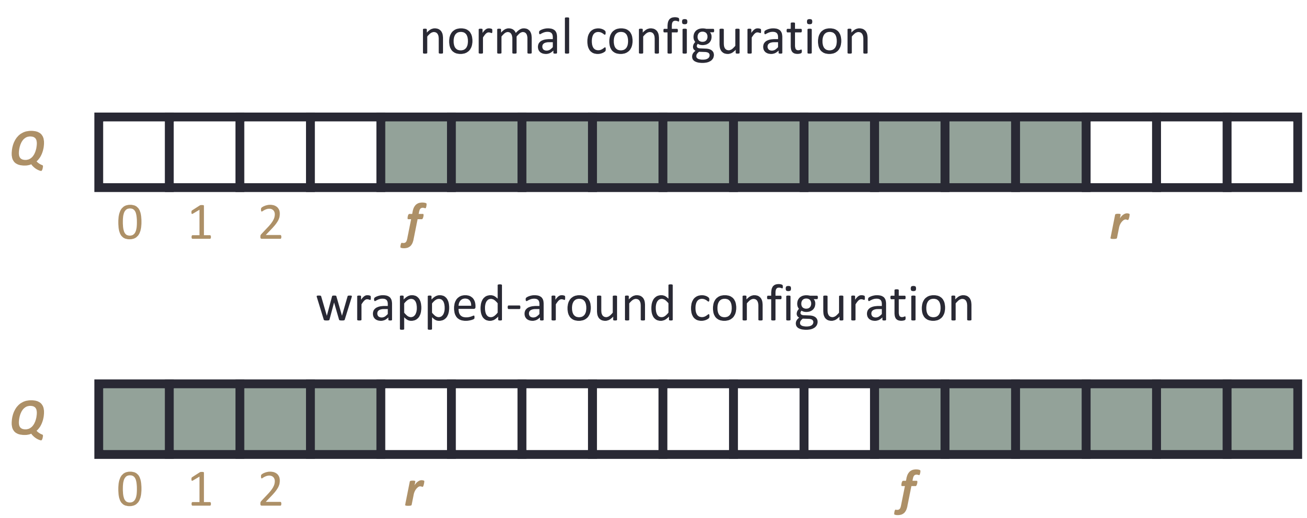


- Using Double Linked Lists



Queues: Array-based Queue

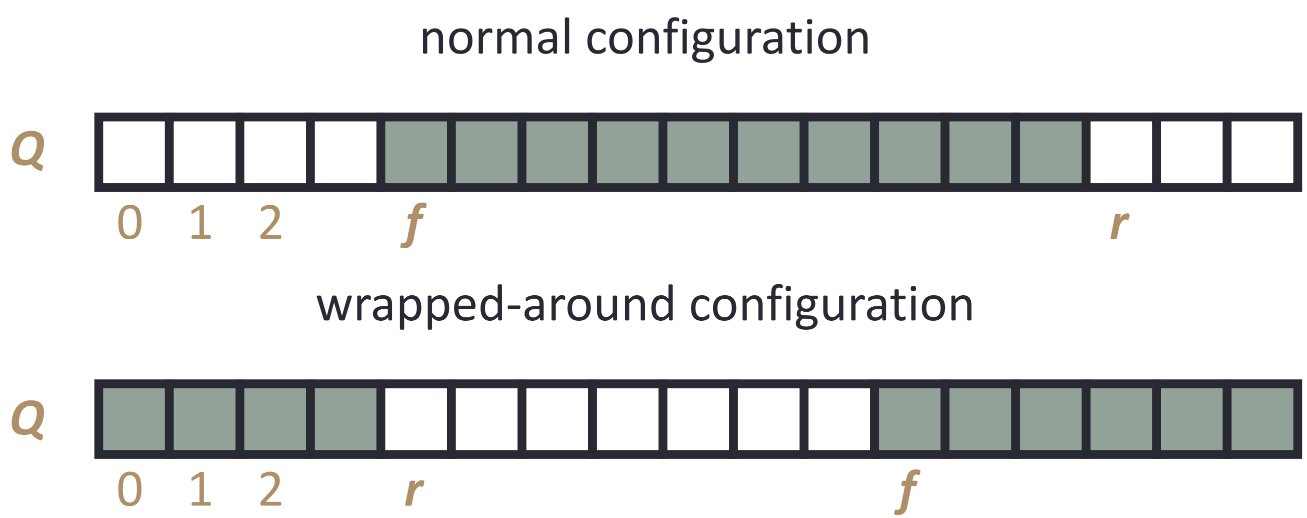
- Use an array of size *N* in a circular fashion
- Two variables keep track of the front and rear
  - *f* index of the front element
  - *r* index immediately past the rear element
- Array location r is kept empty



Queues: Array-based Queue

- We use the modulo operator (remainder of division)
- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

Method **enqueue**(new_element)
    if (Q.Count = N – 1) **then** throw “FullQueueException”;
    else
        Q[r] = new_element;
        r = (r + 1) mod N;



Queues: Array-based Queue

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

Algorithm dequeue()
    if (Q.Count == 0) **then** throw "EmptyQueueException";
    else
        element = Q[f];
        f = (f + 1) mod N;
        return element;

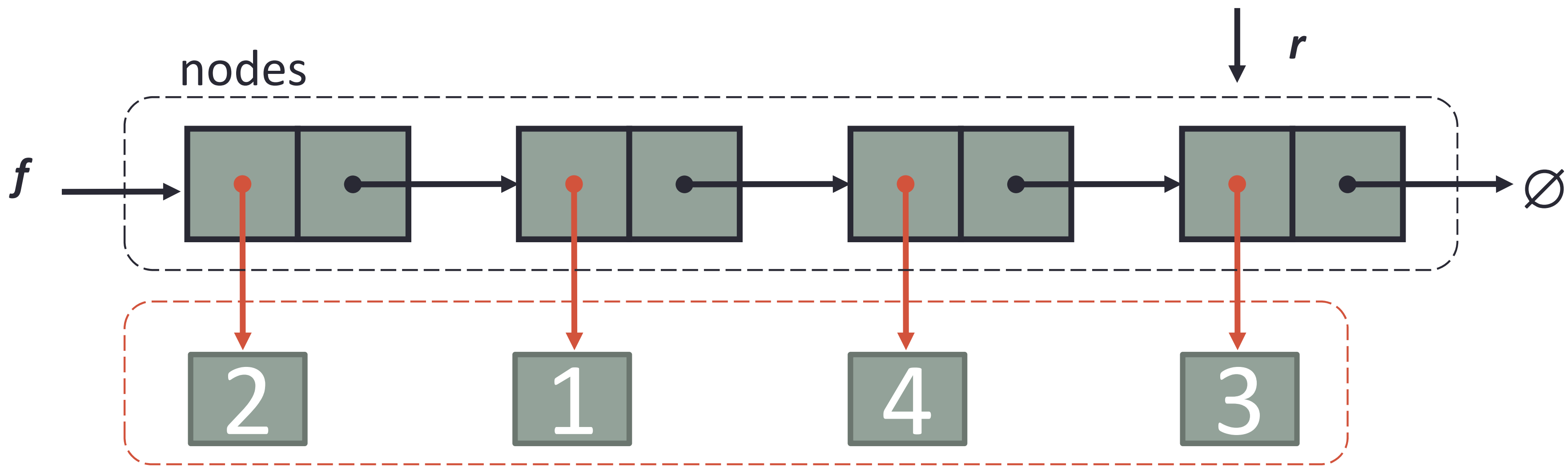
normal configuration

wrapped-around configuration

Queues: Doubly-Linked List-based Queue

We can implement a queue with a doubly linked list

- The front element is stored at the first node
- The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue takes $O(1)$ time

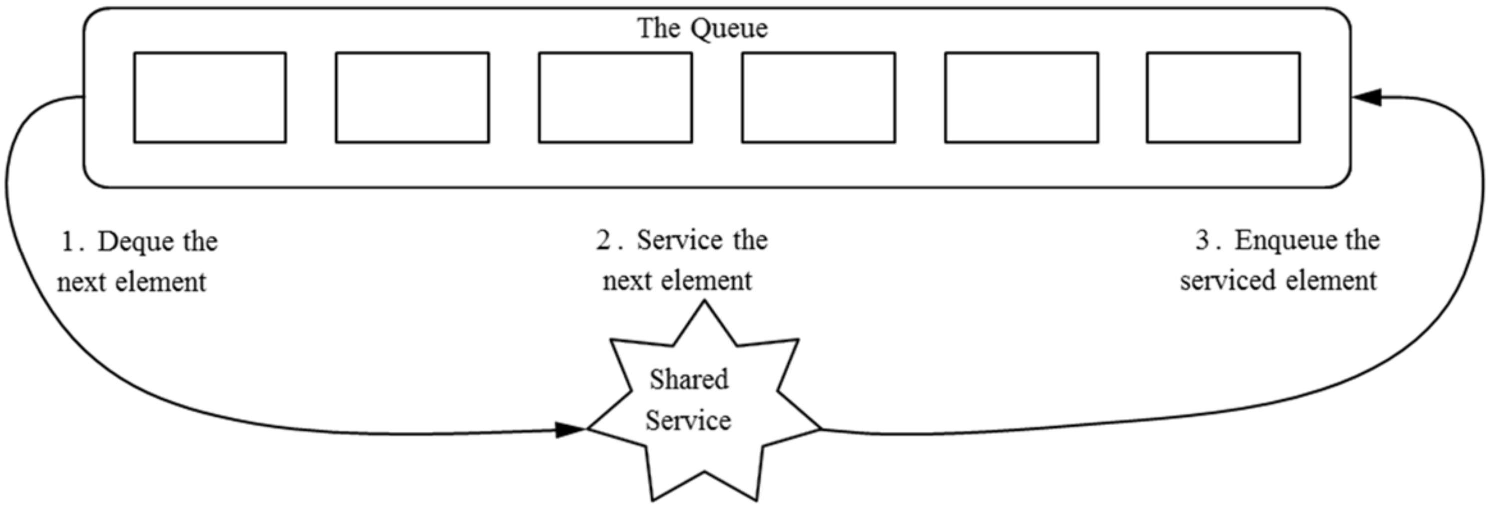


data elements

Queues: Practical Examples

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*)



Other references and things to do

- Read chapters 6.1 and 6.2 in Data Structures and Algorithms in Java. Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser, 2014.