COMP20010

Data Structures and Algorithms I
09 - Stacks

Dr. Aonghus Lawlor
aonghus.lawlor@ucd.ie

Stacks

Abstract Data Type (ADT)

ADT is a set of objects with a set of operations

mathematical abstractions only, doesn’t specify how they are implemented

lists, vectors, sets, and graphs, along with their operations, can be viewed as ADTs

In Java a class allows for the implementation of ADTs, hiding the implementation details.

If the implementation details change, it should be easy to do so, and should be completely transparent to the rest of the program.

Which operations are supported for each ADT is a design decision

List ADT

- The List interface has the following methods:

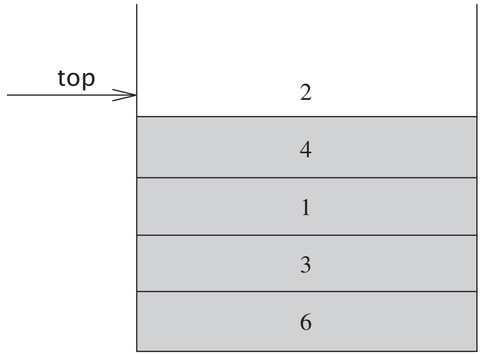
size()	Returns the number of elements in the list
isEmpty()	Returns a boolean indicating whether the list is empty
get(i)	Returns the element of the list with index I, error condition occurs if i is outside the range [0, size()-1]
set(i, e)	Replaces the element at index i with e, and returns the old element that was replaced; an error occurs if i is not in range [0, size()-1]
add(i, e)	Inserts a new element e into the list at index i, moving all subsequent elements one index later in the list; an error occurs if i is not in range [0, size()-1]
remove(i)	Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error occurs if i is not in range [0, size()-1]

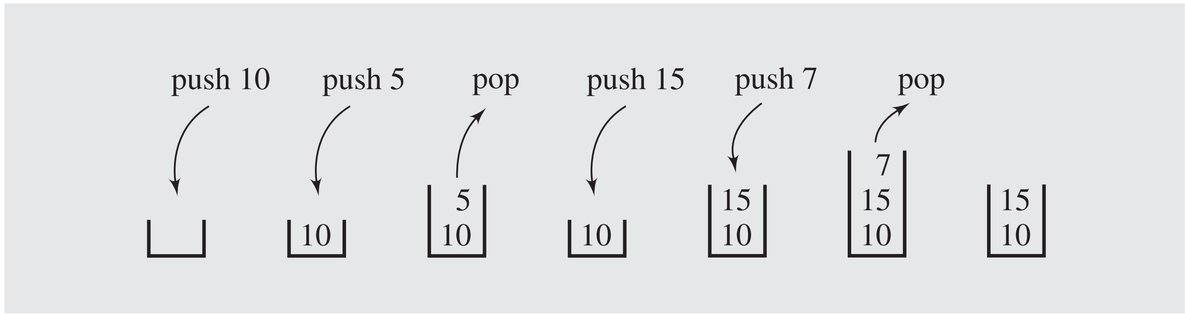
Stacks

- A stack is a container of objects / values.
- Insertion and removal based on the last-in-first-out (LIFO) principle.
  - Objects can be inserted at any time, but only the last (the most-recently inserted) object can be removed.
- Terminology:
  - Items are “Pushed” onto the stack (insertion).
  - Items can be “Popped” off the stack (removal).
  - The “Top” of the stack is the last item pushed

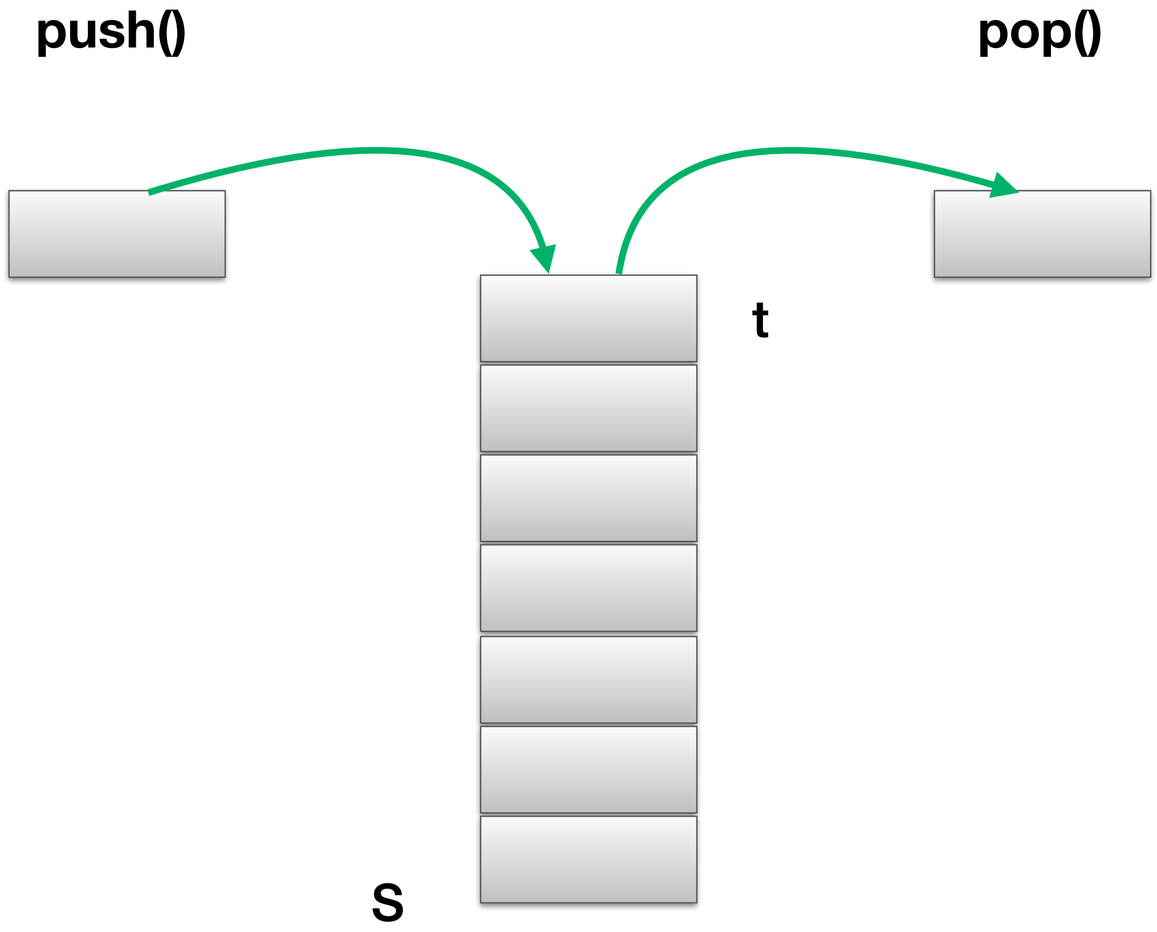
Stack

- a stack is defined in terms of operations which change its status and check its status
- generally a stack is useful when data has to be stored and retrieved in reverse order





Stacks



Everything happens at the ‘top’

Abstract Data Type

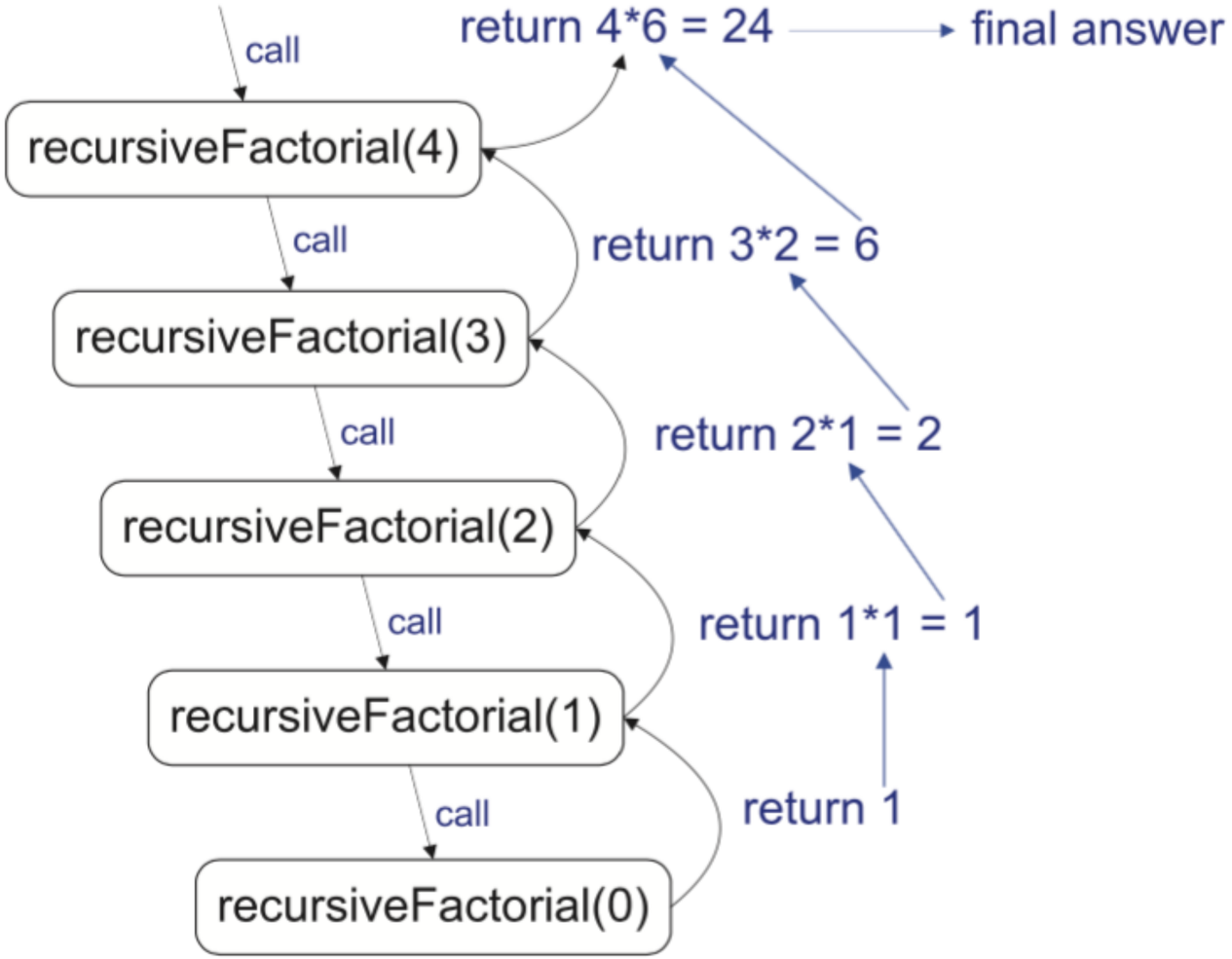
Stack ADT

push(e)	insert element e at top of stack
pop()	remove the top element from the stack
top()	return a reference to the top element of the stack without removing it
size()	number of elements in the stack
empty()	true if the stack is empty, false otherwise

Stack Applications

- Direct Applications
  - web page history in a web browser
  - undo sequence in a text editor
  - chain of method calls in VM
- Indirect Applications
- auxiliary data structure for algorithms
- component of other data structures

Java call stack



Stack - Implementation

Lists

- objects are stored in special nodes
- nodes maintain ordering information
- nodes link to next object in stack
- infinite capacity

Arrays

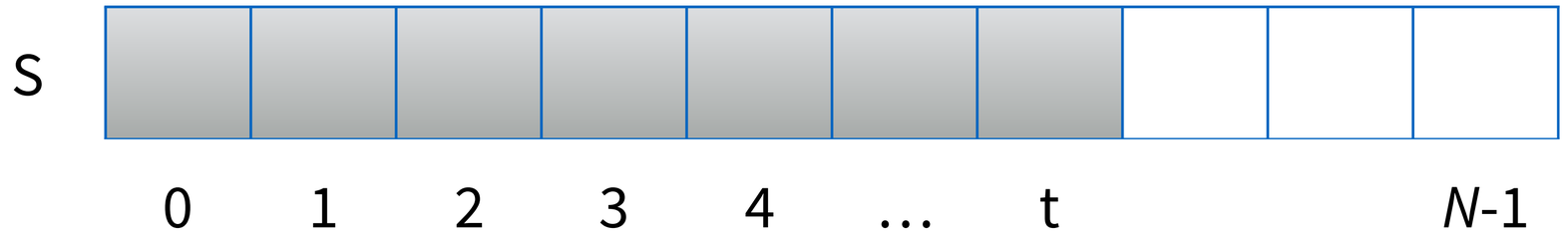
- array holds the objects pushed onto the stack
- insertion starts at 0
- auxiliary value required to keep track of top of stack
- finite capacity

Stacks

(Array Implementation)

Array based Stack

- Create a stack using an array by specifying the maximum size of our stack (eg. N=100)
- The stack consists of:
- N-element array *S*
- an integer variable *t*, the index of the top element in array *S*



Array based Stack

- Simple implementation of stack ADT uses array
- add elements from left to right
- use variable to keep track of top of array

```
Algorithm size():
    return t + 1
```

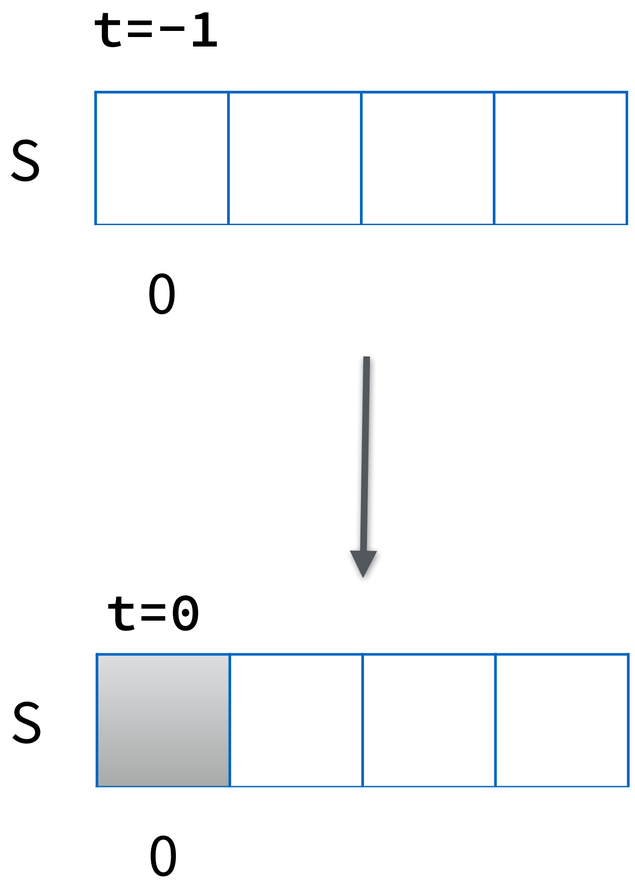
```
Algorithm empty():
    if t < 0:
        return true
    return false
```

```
Algorithm top():
    if empty():
        return StackEmptyError
    return S[t]
```

Array based Stack

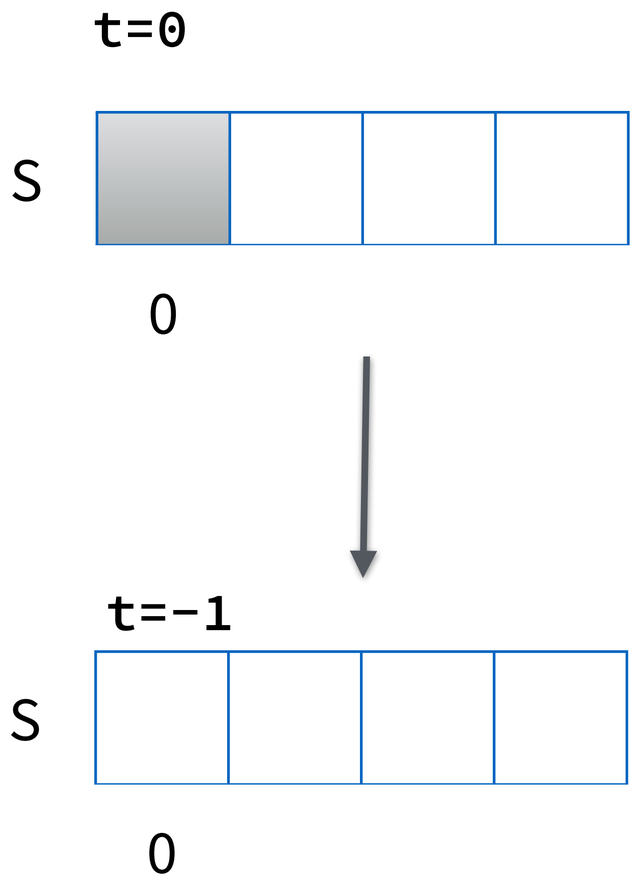
- the underlying array *S* may become full
- a push operation may cause an error (throw exception)
- limitation of the array based implementation
- not intrinsic to Stack ADT (we could avoid by using *ArrayList*)

Array based Stack




```
Algorithm push(e):
    if size() == N:
        return StackFullError

    t ← t + 1          {increment the top counter}
    S[t] ← e           {assign the data value}
```

Array based Stack




```
Algorithm pop():
    if empty() is True:
        return null
    else:
        r = S[t]
        t ← t - 1
        return r
```

Array based Stack

- view operations as atomic
- show the state of the array *S* and the top element index *t* after each operation

operation	t	0	1	2	3	4	5
push(H)	0	H					
push(A)	1	H	A				
push(A)	2	H	A	A			
pop()	1	H	A				
push(P)	2	H	A	P			
push(P)	3	H	A	P	P		
push(Y)	4	H	A	P	P	Y	

Performance

- Performance and Limitations
- n is the number of elements in the stack
- the space used is O(n)
- each operation runs in time O(1)

Operation	Time
size	O(1)
empty	O(1)
top	O(1)
push	O(1)
pop	O(1)

Limitations

The maximum size of the stack must be defined in advance (and cannot be changed unless we use a resizable array)

trying to push a new element into a full stack will cause an error (unless we use a resizable array)

Java Implementation

Start with the interface which represents the ADT

```
public interface Stack<E> {

  /**
   * Returns the number of elements in the
stack.
   */
  int size();

  /**
   * Tests whether the stack is empty.
   */
  boolean isEmpty();

  /*
   * Inserts element at the top of the stack.
   */
  void push(E e);

  /**
   * Returns, but does not remove, the element
at the top of the stack.
   */
  E top();

  /**
   * Removes and returns the top element from
the stack.
   */
  E pop();
}
```

Java Implementation

```
public class ArrayStack<E> implements Stack<E> {
  /** Default array capacity. */
  public static final int CAPACITY=1000;   // default array capacity

  /** Generic array used for storage of stack elements. */
  private E[] data;                        // generic array used for storage

  /** Index of the top element of the stack in the array. */
  private int t = -1;                      // index of the top element in stack

  /** Constructs an empty stack using the default array capacity. */
  public ArrayStack() { this(CAPACITY); }  // constructs stack with default
capacity

  /**
   * Constructs and empty stack with the given array capacity.
   * @param capacity length of the underlying array
   */
  @SuppressWarnings({"unchecked"})
  public ArrayStack(int capacity) {        // constructs stack with given capacity
    data = (E[]) new Object[capacity];     // safe cast; compiler may give warning
  }
}
```

Java Implementation

```
  /**
   * Returns the number of elements in
the stack.
   */
  @Override
  public int size() {
    return (t + 1);
  }
```

```
  /**
   * Returns, but does not remove, the
element at the top of the stack.
   */
  @Override
  public E top() {
    if (isEmpty())
      return null;
    return data[t];
  }
```

```
  /**
   * Tests whether the stack is
empty.
   *
   * @return true if the stack is
empty, false otherwise
   */
  @Override
  public boolean isEmpty() {
    return (t == -1);
  }
```

Java Implementation

```
  /**
   * Inserts an element at the top of the
stack.
   */
  @Override
  public void push(E e)
   throws IllegalStateException {
    if (size() == data.length)
      throw new IllegalStateException("Stack is
full”);
    // increment t before storing new item
    data[++t] = e;
  }
```

```
  /**
   * Removes and returns the top
element from the stack.
   */
  @Override
  public E pop() {
    if (isEmpty())
      return null;
    E answer = data[t];
    // dereference to help garbage
    collection
    data[t] = null;
    t--;
    return answer;
  }
```

Stacks

(List Implementation)

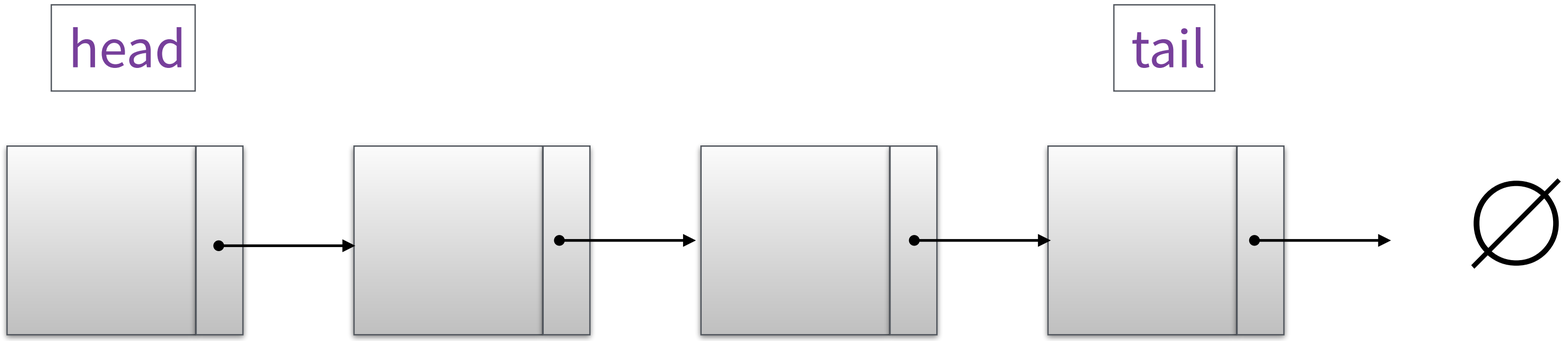
Array-based Stacked

- Array-based implementations => Finite Capacity.
  - Memory (Mis-)management issues.
  - Application Logic issues.
- Possible Solution: Use an “extendable array”
  - Creating a new larger array and copying the existing values into it.
  - Running time – O(n) => Pop / Push running times become O(n)
- Are there any other implementation strategies?

- YES! We can use the **Linked List** data structure.

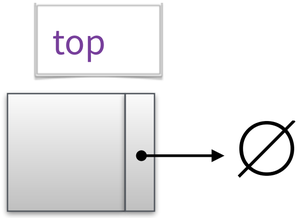
Singly linked list

- In a Linked List, the objects are stored in **nodes**.
  - Each node also maintains a reference to the next node in the list (this is the **link**).
  - The link of the last node in the list is set to null.

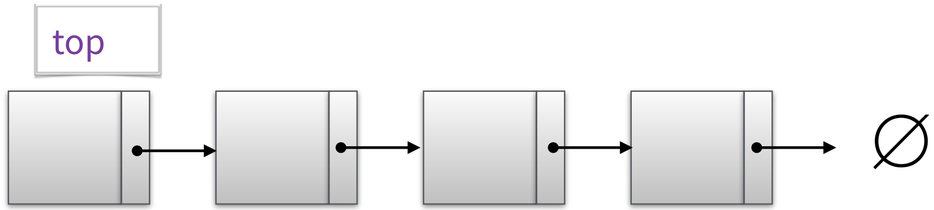


Link-based Stack

- Auxiliary data structure: Node
- a reference to the object being stored in the stack
- a link to the next node in the stack
- "top" node of the stack
- the link of the "bottom" node is set to point to null



empty stack



Link-based Stack

```
Algorithm size():
    Input: none
    Output: number of objects in stack
    return size
```

```
Algorithm empty():
    return size == 0
```

```
Algorithm top():
    Input: none
    output: the top object
    return top.element
```

Link-based Stack

```
Algorithm push(o):
    Input: object o
    Output: none
    node ← new Node(o)
    node.next ← top
    top ← node
    size ← size + 1
```

```
Algorithm pop():
    Input: none
    Output: the top object
    e ← top.element
    top.element ← null
    top ← top.next
    size ← size - 1
    return e
```

```
Algorithm push(o):
    Input: object o
    Output: none
    list.addFirst(o)
    size ← size + 1
```

```
Algorithm pop():
    Input: none
    Output: the top object
    e ← list.first()
    size ← size - 1
    return e
```

Java Implementation - Link-based Stack

same Stack interface as before

```
public class LinkedStack<E> implements
Stack<E> {
  /** The primary storage for elements of
the stack */
  private SinglyLinkedList<E> list = new
SinglyLinkedList<>(); // an empty list

  /** Constructs an initially empty stack.
*/
  public LinkedStack() {
  } // new stack relies on the initially
empty list

  /**
   * Returns the number of elements in the
stack.
   */
  public int size() {
    return list.size();
  }

  /**
   * Returns, but does not remove, the
element at the top of the stack.
   */
  public E top() {
    return list.first();
  }

  /**
   * Tests whether the stack is empty.
   */
  public boolean isEmpty() {
    return list.isEmpty();
  }
}
```

Java Implementation - Link-based Stack

```java
/**
 * Inserts an element at the top of the stack.
 */
public void push(E element) {
  list.addFirst(element);
}
```

```java
/**
   * Removes and returns the
    * top element from the stack.
   */
  public E pop() {
    return list.removeFirst();
  }
```

Link based Stacks

- operation running times
- Which implementation is better- list or array?

do you know how many items will be in the stack?

what about memory requirements?

do you need fast iteration through the stack?

Operation	Time
size	O(1)
empty	O(1)
top	O(1)
push	O(1)
pop	O(1)

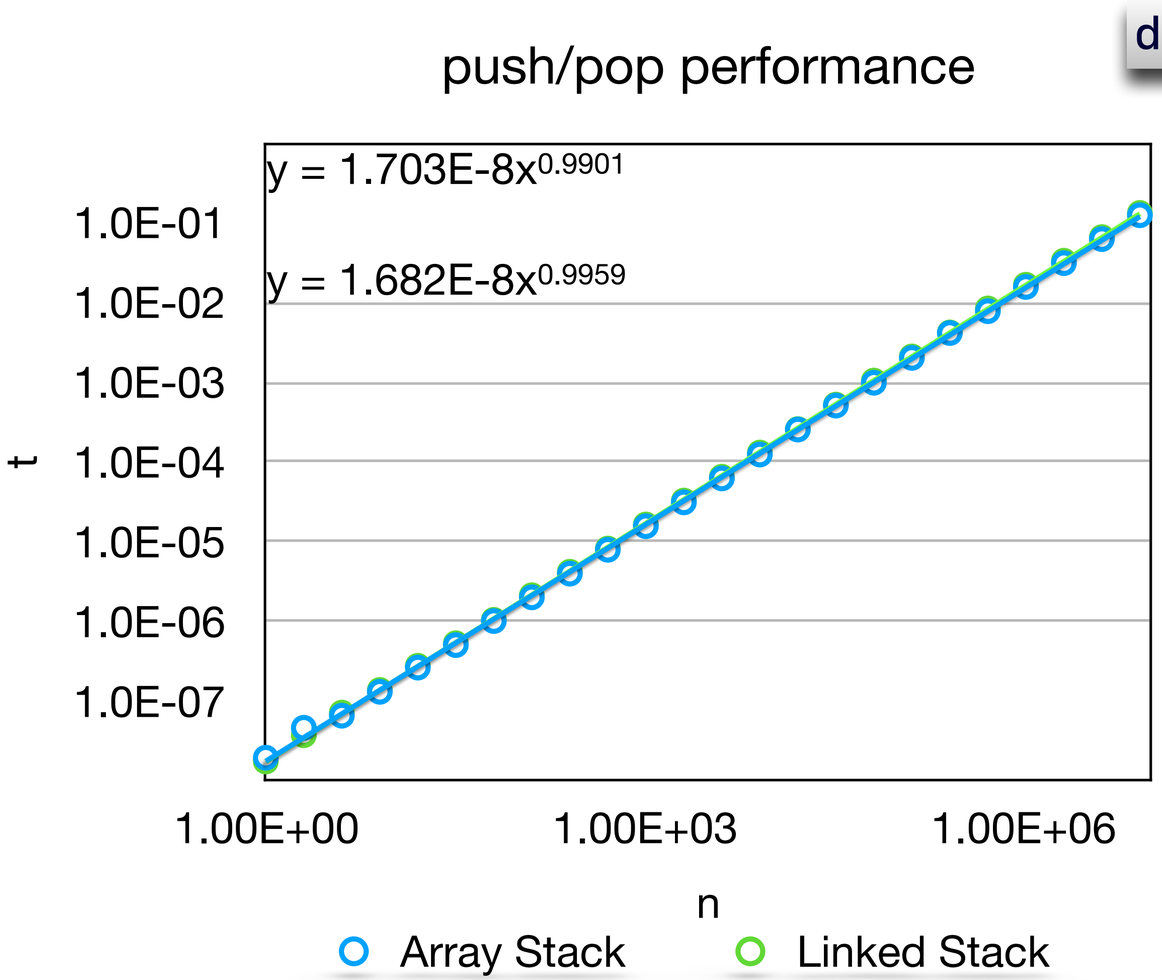
Testing Array/List Stack Performance

```
  public static void
stack_ops(LinkedList<Integer> s, int n,
Random rnd) {
    for(int i = 0; i < n; ++i) {
      if(rnd.nextFloat() < 0.3) {
        s.push(1);
      } else {
        if(!s.isEmpty()) {
          s.pop();
        }
      }
    }
  }
```

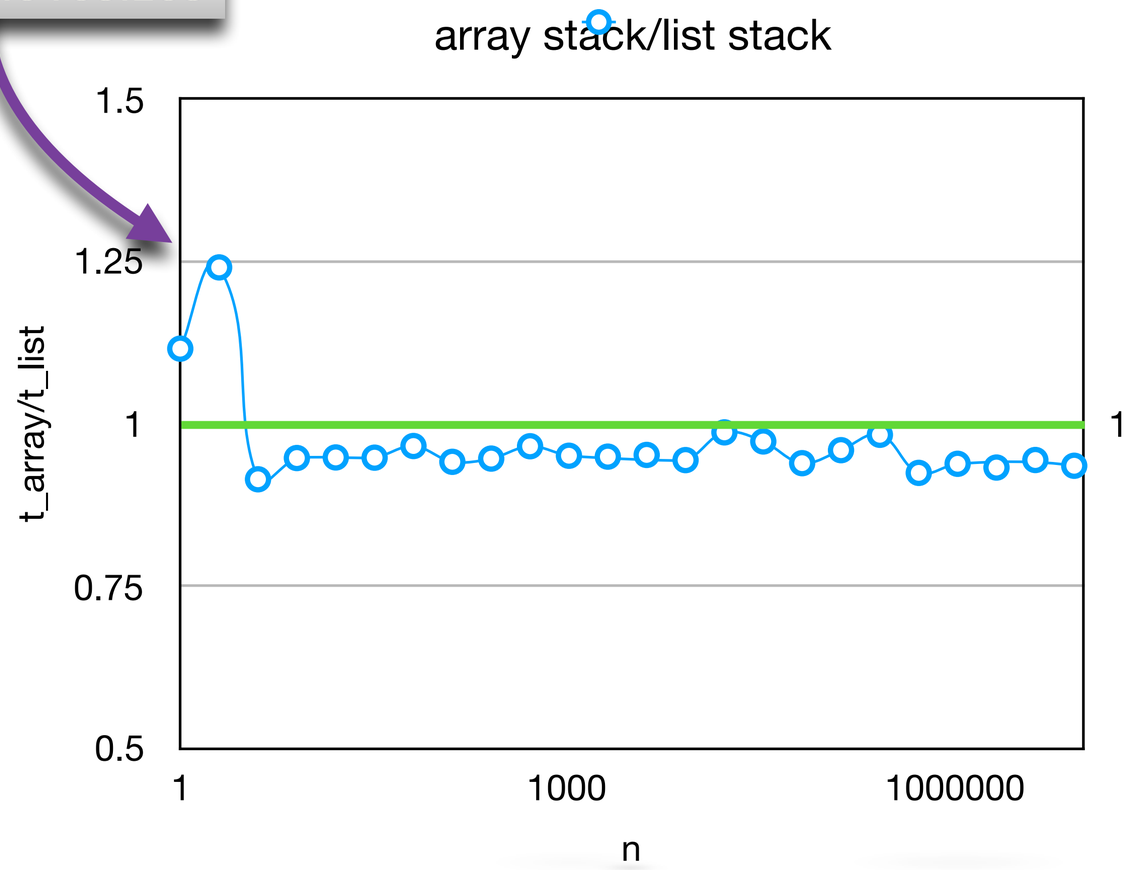
testing performance of n push/pop operations

Uses a Functional approach to specify the function to time

```
public static void test_stack() {
    Random rnd = new Random(20010);

    Function<Integer, Void> foo = (n) -> {
      Stack<Integer> s = new
Stack<Integer>();
      //LinkedList<Integer> s = new
LinkedList<Integer>();
      Timeit.stack_ops(s, n, rnd);
      return null;
    };

    for (int n = 1; n < 10000000; n *=2) {
      double t =
Timeit.timeit(foo).apply(n);
      System.out.println(n + "\t" + t);
    }
  }
```



dynamic resizes



timing of both Array and List based Stacks are O(1) (n operations take O(n) time). Arrays are slower for small values of n, but then slightly faster

Stack Applications

Stack Example

- addition

$$\begin{array}{r} 592 \\ +\ 3784 \\ \hline 4376 \end{array}$$

addition()

read the numerals of the first number and store the numbers corresponding to them on one stack;
read the numerals of the second number and store the numbers corresponding to them on another stack;

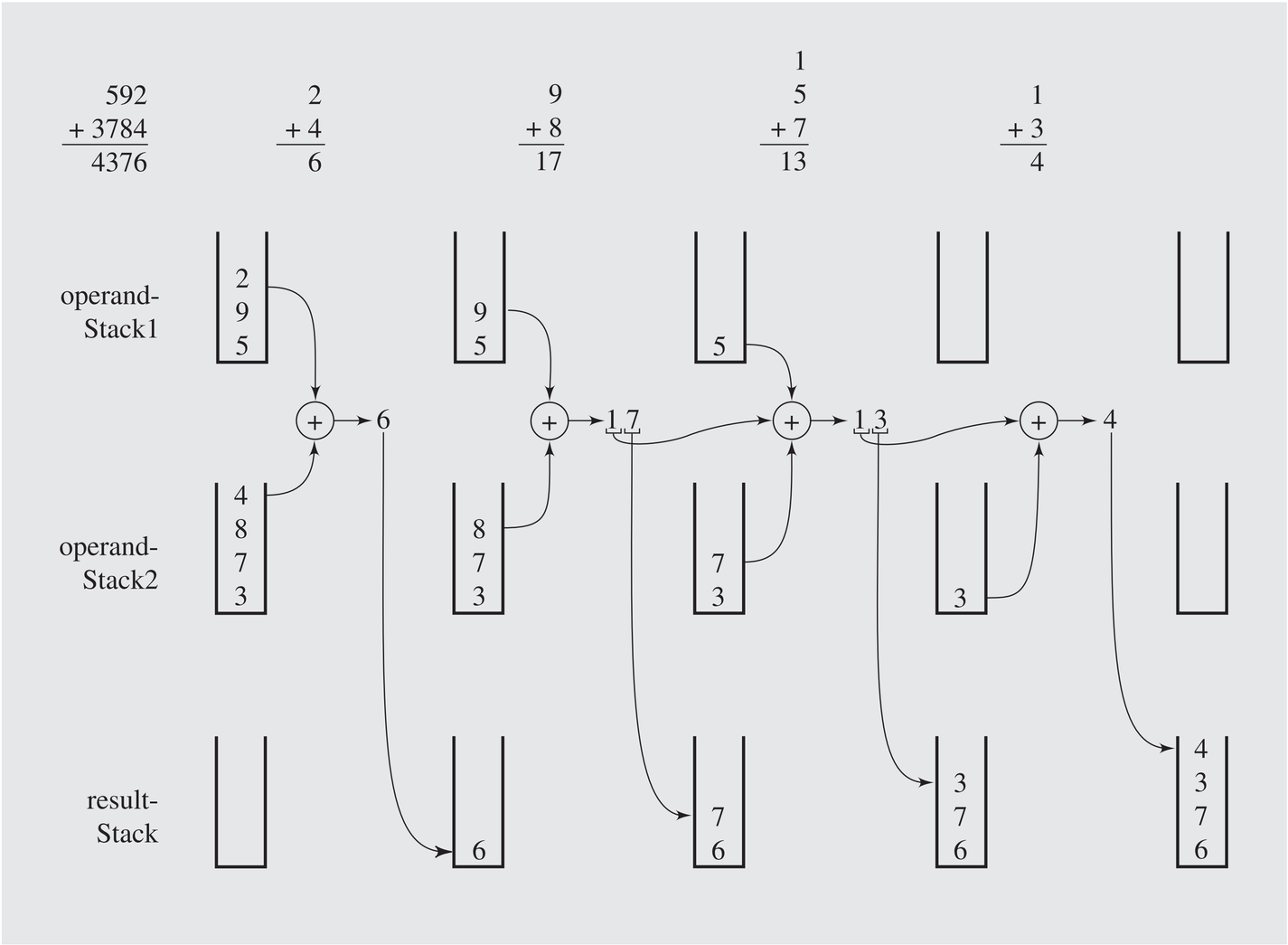
carry = 0;
while at least one stack is not empty

pop a number from each nonempty stack and add them to carry; push the unit part on the result stack;
store carry in carry;

push carry on the result stack if it is not zero;
pop numbers from the result stack and display them;

Stack Example

- addition



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

S ← Stack()

for i ← 0 to n-1 do:
    if X[i] is an opening group symbol then {one of ["(", "{", "["]}
        S.push(X[i])
    else if X[i] is a closing group symbol then
        if S.empty() 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.empty():
    return true {every symbol has matched}
else:
    return false {some symbols were not matched}
```

(	)
{	}
[	]

✅ ()(()){([()])}

✅ (([])((])){[]}

❌ ({[])}

❌ {

HTML tags

markup languages provide a system for inserting annotations into a document.

The most important feature of a markup language is that the *tags* it uses to indicate annotations should be easy to distinguish from the document *content*.

examples are LaTeX for document processing HTML or "Hypertext Markup Language"

In HTML, tags appear in "angle brackets" such as in "<html>".

When you load a Web page in your browser, you don't see the tags themselves: the browser interprets the tags as instructions on how to format the text for display.

HTML tags

Most tags in HTML are used in pairs to indicate where an effect starts and ends. For example:

 represents the start of a paragraph, and indicates where that paragraph ends.

 and are used to place the enclosed text in **bold** font, and and indicate that the enclosed text is *italic*.

Note that "end" tags look just like the "start" tags, except for the addition of a backslash after the < symbol.

Sets of tags are often nested inside other sets of tags. For example, an *ordered list* is a list of numbered bullets. You specify the start of an ordered list with the tag <ol>, and the end with . Within the ordered list, you identify items to be numbered with the tags <li> (for "list item") and .

HTML tags

For example, the following specification:

```
<ol>
  <li>First item</li>
  <li>Second item</li>
  <li>Third item</li>
</ol>
```

would result in the following:

1. First item
2. Second item
3. Third item

HTML tags

the pattern of using matching tags strongly resembles the pattern of matching parentheses

when you use parentheses, brackets, and braces, they have to match in reverse order, such as "{[()]}".

A pattern such as "[(])" would be incorrect since the right bracket does not match the left parenthesis.

a HTML pattern such as would be incorrect since the closing tags are in the wrong order.

Evaluating Arithmetic Expressions

$$14 - 3 * 2 + 7 = (14 - (3 * 2)) + 7$$

- Operator precedence

  * has precedence over +/-

- Associativity:

  operators of the same precedence group are evaluated from left to right

  eg: "*(x-y) + z*" instead of "*x - (y + z)*"

- Idea: push each operator on the stack, but first pop and perform higher and equal precedence operations

Evaluating Arithmetic Expressions

Use two stacks

op_stk holds the operators

value_stk holds the values

we use *$* as a signal for "*end of input*" and is the token with lowest precedence

```
Algorithm do_op():
    x ← value_stk.pop()
    y ← value_stk.pop()
    op ← op_stk.pop()
    value_stack.push(y op x)
```

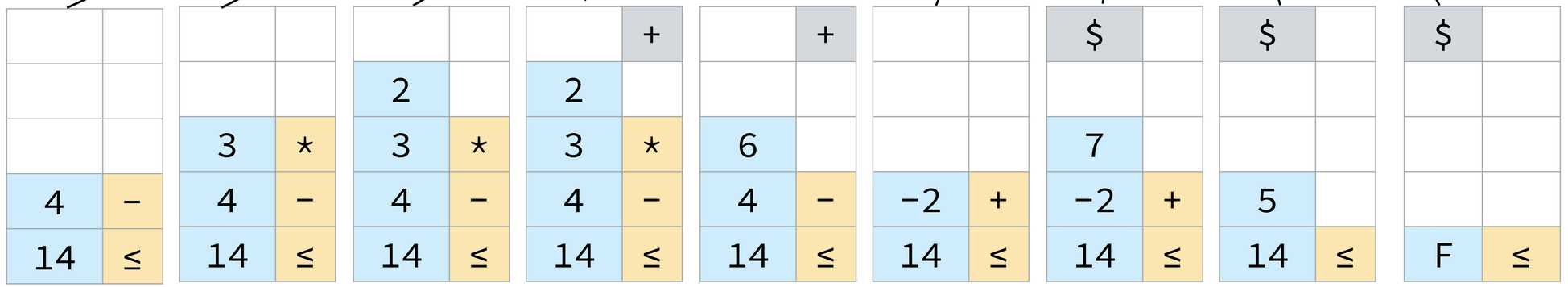
```
Algorithm evaluate_exp(stream):
    Input: a stream of tokens representing
    an expression
    Output: the value of the expression

    for token z in stream:
        if is_number(z) then:
            value_stk.push(z)
        else:
            repeat_ops(z)
            op_stk.push(z)

    repeat_ops($)
    return value_stk.top()
```

```
Algorithm repeat_ops(ref_op):
    while (value_stk.size() > 1 and
            prec(ref_op) <= prec(op_stk.top())):
        do_op()
```

Evaluating Arithmetic Expressions

$$14 \leq 4 - 3 * 2 + 7$$

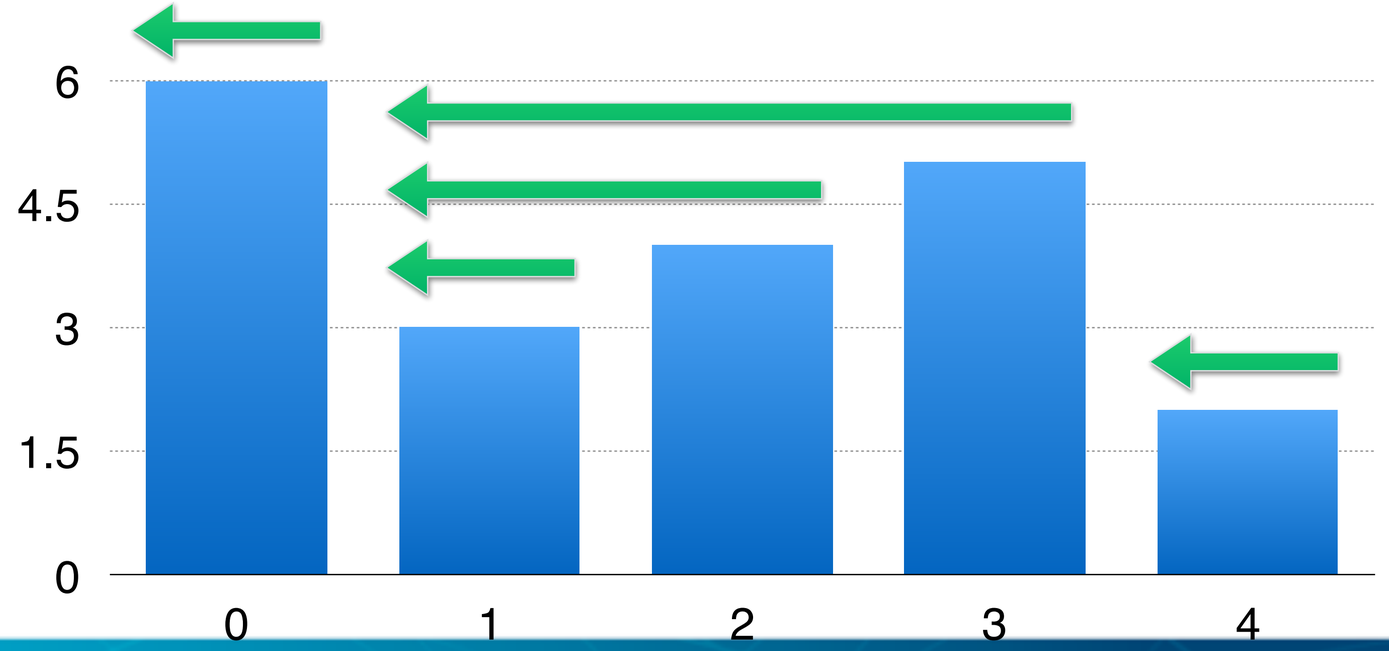
operator ≤ has lower precedence than +/-



Computing Spans

- 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] such that X[j] ≤ X[i]
- Spans have applications in financial analysis (eg. stock at 4-week high)

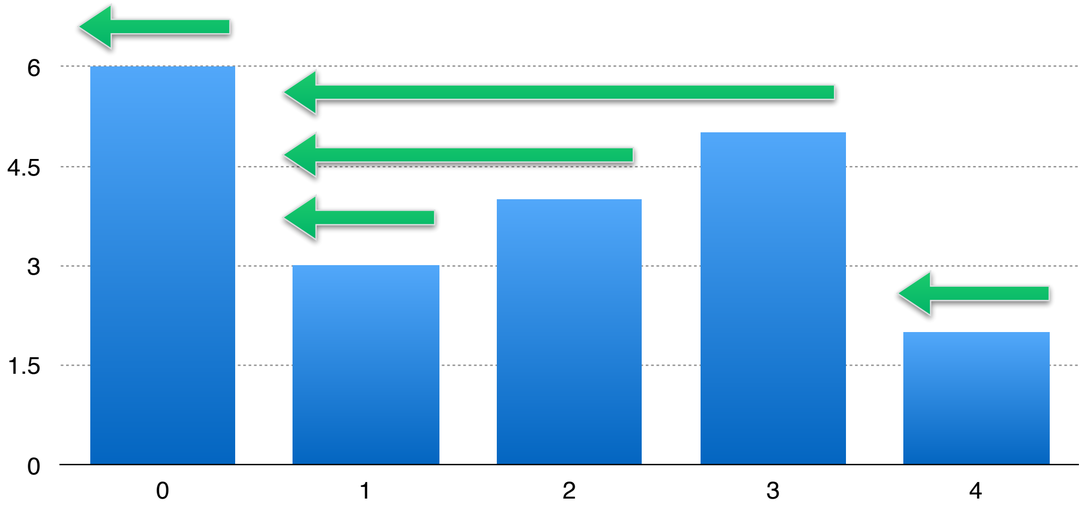
X	S
6	1
3	1
4	2
5	3
2	1



Computing Spans

```
Algorithm spans1(X, n):
    Input: array X of n integers
    Output: array S of spans of X

    S ← new array of n integers
    for i ← 0 to n − 1 do:
        s ← 1
        while s ≤ i and X[i−s] ≤ X[i]        1+2+...+(n−1)
            s ← s + 1
        S[i] ← s
    return S
```

$O(n^2)$



Computing Spans

- we keep in a stack the indices of the visible elements when ‘looking back’

- we scan the array from left to right
  - let i be the current index
  - pop indices from the stack until we find index j such that X[i] < X[j]
  - set S[i] = i -j
  - push x onto stack

Computing Spans

each index of the array is:

pushed onto the stack once

popped from the stack at most one

what is the growth factor of this algorithm?

```
Algorithm spans2(X, n)
    Input: array X of n integers
    Output: array S of spans of X

    S ← new array of n integers
    A ← new empty stack

    for i ← 0 to n − 1 do:
        while A is not empty and X[A.top()] ≤ X[i] do:
            A.pop()
        if A is empty then:
            S[i] ← i + 1
        else:
            S[i] ← i − A.top()
        A.push(i)
    return S
```

Stacks Java

java.util.Stack

java.util
Class Stack<E>

- java.lang.Object
  - java.util.AbstractCollection<E>
    - java.util.AbstractList<E>
      - java.util.Vector<E>
        - java.util.Stack<E>

- All Implemented Interfaces:
  Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

java.util.Stack

```
public class Stack<E>
extends Vector<E>
```

The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.
When a stack is first created, it contains no items.

java.util.Stack

Constructor Summary

- **Stack**() Creates an empty Stack.

Method Summary

- boolean **empty**() Tests if this stack is empty.
- E **peek**() Looks at the object at the top of this stack without removing it from the stack.
- E **pop**() Removes the object at the top of this stack and returns that object as the value of this function.
- E **push**(E item) Pushes an item onto the top of this stack.

java.util.Stack (jdk1.8)

```
    /**
     * Pushes an item onto the top of this stack. This has exactly
    * the same effect as:
    * <blockquote><pre>
    * addElement(item)</pre></blockquote>
    *
    * @param   item   the item to be pushed onto this stack.
    * @return  the <code>item</code> argument.
    * @see     java.util.Vector#addElement
     */
    public E push(E item) {
        addElement(item);

        return item;
    }
```

```
    /**
    * Removes the object at the top of this stack and returns that
    * object as the value of this function.
    *
    * @return  The object at the top of this stack (the last item
    *          of the <tt>Vector</tt> object).
    * @throws  EmptyStackException  if this stack is empty.
     */
    public synchronized E pop() {
        E       obj;
        int     len = size();
        obj = peek();
        removeElementAt(len - 1);
        return obj;
    }
```

- In the tutorial:
  - Write your own versions of the Stack ADT
  - implement the delimiter matching algorithm using Stack

Exercises

(a) Write your own implementation of the Stack interface using an array as the basic data structure [20]

(b) Write your own implementation of the Stack interface using your own `SinglyLinkedList` as the basic data structure [20]

(c) Write your own implementation of the Stack interface using `java.util.Vector` as the basic data structure [20]

Q2: _____ **(20points)**

Write a `BoundedStack` implementation in Java. A BoundedStack has a parameter `maxSize` which sets the maximum capacity on creation. On `push()`'ing an object to the BoundedStack, an exception is thrown if the stack is full.

Q3: _____ **(10points)**

Write the pseudocode algorithm which reverses the elements on a Stack using two additional Stack's (no other data structures are allowed).

Exercises

Q4: _____ **(50points)**

We want to write an algorithm to check if the parentheses in an expression is balanced:

$$(w * (x + y)/z–(p/(r–q)))$$

It may have several different types of

```
braces {}
brackets[]
parentheses()
```

Each opening on the left delimiter must be matched by a closing (right) delimiter.

Left delimiters that occur later should be closed before those occurring earlier.

Some examples are:

```
c[d]          // correct
a{b[c]d}e     // correct
a{b(c]d}e   // not correct; the "]" after the c doesn't match the "(" after the b
a[b{c}d]e}    // not correct; nothing matches the final }
a{b(c)        // not correct; nothing matches the opening }
```

Exercises

You should write a Java implementation of the following delimiter matching algorithm using your Stack implementation (either array or list based):

1. Read the characters from the string.
2. Whenever you see a left (opening) delimiter, push it to the stack.
3. Whenever you see a right (closing) delimiter pop the (hopefully matching) opening delimiter from the stack, and
4. If they don't match, report a matching error
5. If you can't pop the stack because it is empty, report a missing left delimiter error
6. If the stack is non-empty after all the characters of the expression have been processed, report a missing right delimiter error.

Check your solution with the following inputs:

1. `"{[()]}"`
2. `"{[(])}"`
3. `"{{[[(())]]}}"`