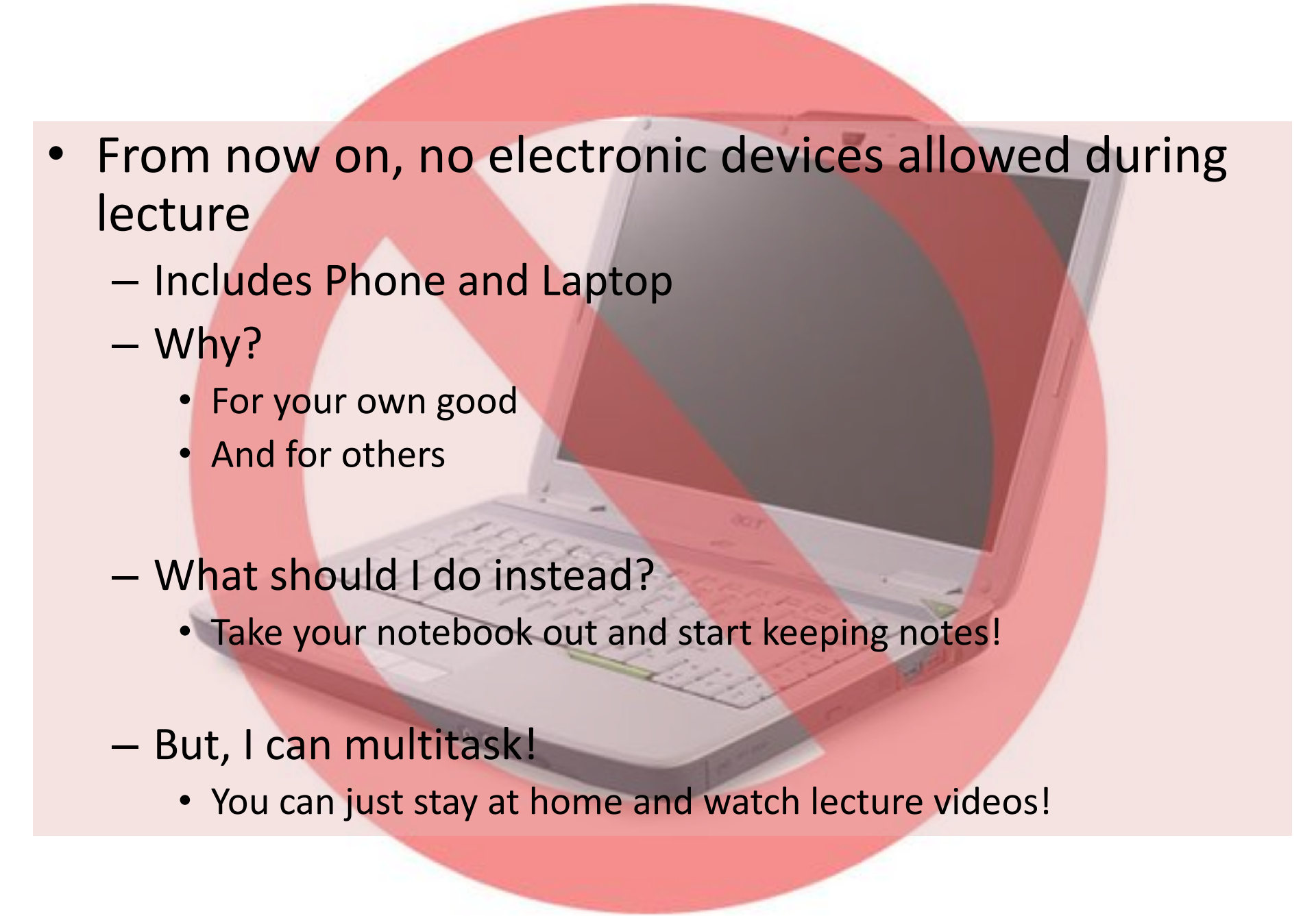
CSC 172– Data Structures and Algorithms

Lecture #10

Spring 2018

Please put away all electronic devices



- From now on, no electronic devices allowed during lecture
  - Includes Phone and Laptop
  - Why?
    - For your own good
    - And for others
  - What should I do instead?
    - Take your notebook out and start keeping notes!
  - But, I can multitask!
    - You can just stay at home and watch lecture videos!

Announcements

- Lab 5 is out (List ADT)

- Project 2 is out
  - Please finish the project before the spring recess
  - I really do not want you work on the project during the recess
  - The project looks really simple but probably is the hardest of all the projects.

PROJECT 2
(UR CALCULATOR)

Link

- http://www.cs.rochester.edu/courses/172/spring2018/projects/proj2.pdf

```
> a=b=c=d=50+(2*3 )/2
> a+b                                   Displays output
> VarE = a + { 50 + [30.52] }
> f = +++++a----b
> g = ++a + (b + c+ (+d - -VarE))
> a = a+b+c -[{(4+2)*32.9}/9]
> clear b
> show all                              Displays output
> clear all
> exit
```

Quiz on Project 2

- What will be the output if I enter

5+3

a) 8

b) 5+3

c) Nothing

Quiz on Project 2

- What will be the output if I enter

5

a) 5

b) 5.0

c) Nothing

Result of evaluation should be in double.

Quiz on Project 2

- What will be the output if I enter

```
a = 5+3
```

a) 8

b) 5+3

c) Nothing

Assignments won't produce any output

Quiz on Project 2

- What will be the output if I enter

+++5+-+-3

a) 8

b) 2

c) Error

+ + = +

- - = +

+ - = -

- + = -

Agenda

- We will talk about two new Data Structures
  - Map
  - Stack
  - (Both useful for Project 2)
- We will see you how can you use these in Java.
- Later, we will learn how to implement them.
  - Stack: This week
  - Map: After MidTerm

Maps

- A Map is an object that maps keys to values.
- A map cannot contain duplicate keys
- Each key can map to at most one value.

Map stores (key,value)

Students HashMap

Key	Value
Alice	15
Bob	12
Carol	108
Dave	105

Students.get(”Bob”) will give 12

```
HashMap< String,Integer> students =
          new HashMap< String,Integer>();
students.put("Alice", new Integer(15));
students.put("Bob", new Integer(12));
students.put("Carol", new Integer(108));
students.put("Dave", new Integer(105));

System.out.println(students.get("Alice"));
System.out.println(students.get("David"));
System.out.println(students.get("Dave"));
```

```
15
null
105
```

Agenda

- Stacks
- Well-ballanced expressions
- Infix and postfix expressions

- Well-formed expressions
- `stacks`
- Infix, postfix

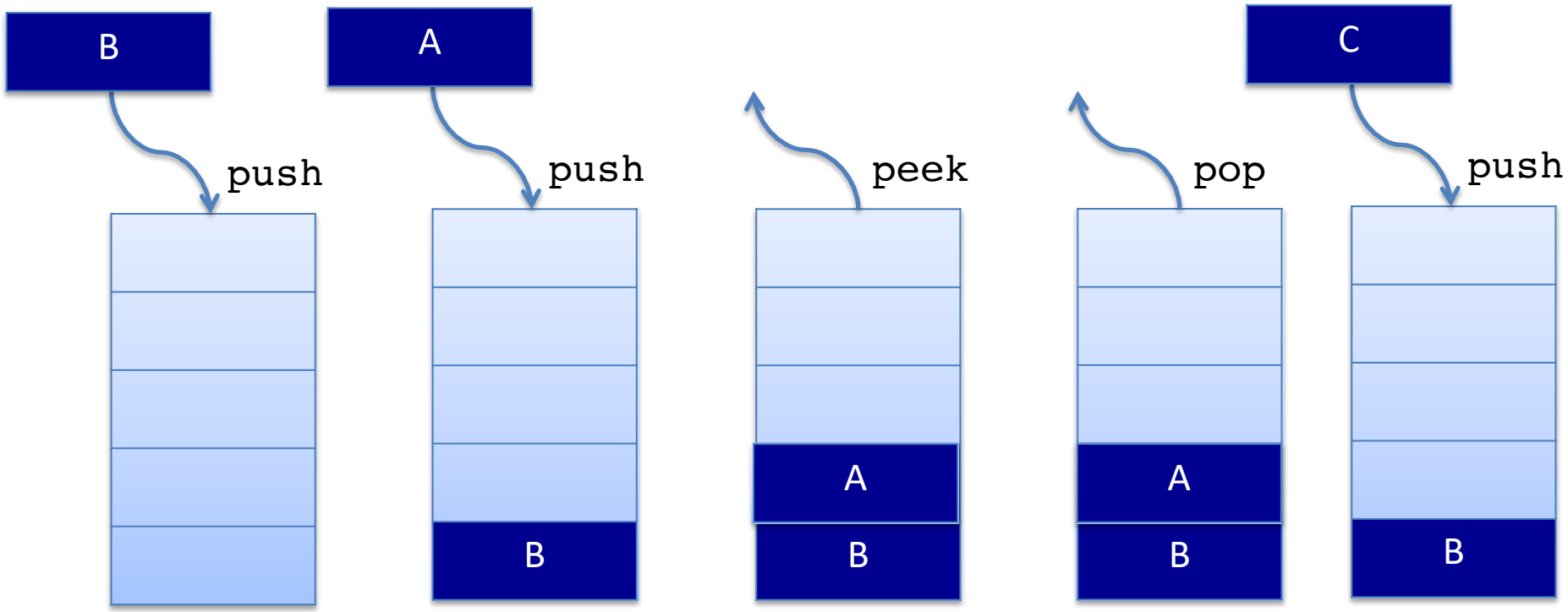
STACKS AND APPLICATIONS

STACK IN JAVA

Java Stack

Modifier and Type	Method and Description
`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.
`int`	**`search(Object o)`** Returns the 1-based position where an object is on this stack.

Stack: push(obj), pop(), and peek()



Practice problem

```
public static void main (String[] args)
    {
        Stack<Integer> stack = new Stack<Integer>();

        for(int i = 0; i <= 10; i=i+2)
        {
            stack.push(i);
        }

        System.out.println("Top of stack = "+ stack.peek());
        System.out.println("Popping element  = "+ stack.pop());
        System.out.println("Top of stack = "+ stack.peek());
        System.out.println("Popping element  = "+ stack.pop());
        System.out.println("Popping element  = "+ stack.pop());
        System.out.println("Top of stack = "+ stack.peek());
        stack.push(12);
        System.out.println("Top of stack = "+ stack.peek());

    }
```

```
Top of stack = 10
Popping element = 10
Top of stack = 8
Popping element = 8
Popping element = 6
Top of stack = 4
Top of stack = 12
```

Application: Parsing

Most parsing uses stacks

Examples includes:

- Matching tags in HTML
- In Java or Well-formed expression, matching
  - `( ... )`
  - `[ ... ]`
  - `{ ... }`

Parsing HTML

The first example will demonstrate parsing HTML

We will show how stacks may be used to parse an HTML document

HTML File

```
<div id="navigation">
 <div class="inner">
  <div id="searcher">
   <form method="get" action="http://www.gnu.org/cgi-bin/estseek.cgi">
    <div><label class="netscape4" for="phrase">Search:</label>
    <input name="phrase" id="phrase" type="text" size="18" accesskey="s"
           value="Why GNU/Linux?" onfocus="this.value=''" />
    <input type="submit" value="Search" /></div><!-- unnamed label -->
   </form>
  </div><!-- /searcher -->
  <ul>
   <li id="tabPhilosophy"><a href=
                          "/philosophy/philosophy.html">Philosophy</a></li>
   <li id="tabLicenses"><a href="/licenses/licenses.html">Licenses</a></li>
   <li id="tabEducation"><a href="/education/education.html">Education</a></li>
   <li id="tabSoftware"><a href="/software/software.html">Downloads</a></li>
   <li id="tabDoc"><a href="/doc/doc.html">Documentation</a></li>
   <li id="tabHelp"><a href="/help/help.html">Help GNU</a></li>
   <li id="joinfsftab"><a
href="https://www.fsf.org/associate/support_freedom?referrer=4052">Join the 
FSF!</a></li>
  </ul>

 </div><!-- /inner -->
</div><!--  /navigation -->
```

Parsing HTML

HTML is made of nested

- *opening tags*, e.g., `<some_identifier>`, and
- matching *closing tags*, e.g., `</some_identifier>`

```
<html>
  <head>
      <title>Hello</title>
  </head>
  <body><p>This appears in the
   <i>browser</i>.
  </p></body>
</html>
```

Parsing HTML

Nesting indicates that any closing tag must match the most recent opening tag

Strategy for parsing HTML:

- read though the HTML linearly
- place the opening tags in a stack
- when a closing tag is encountered, check that it matches what is on top of the stack
  - If yes, take it out. If no, error…
  - [Most Browser knows how to recover]

Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>			

Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>		

Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>	<title>	

Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>	<title>	

Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>		

Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>		

Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	

Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	<i>

Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	<i>

Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	

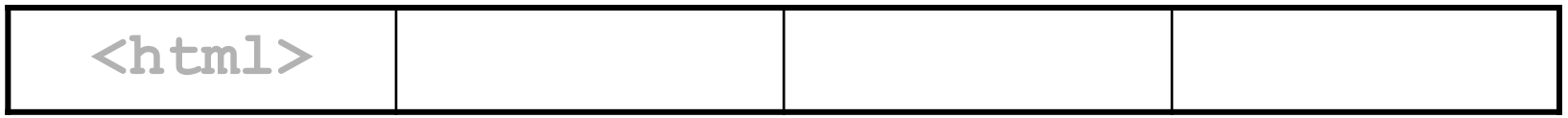
Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>		

Parsing HTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```



Parsing HTML

3.2.5.1

We are finished parsing, and the stack is empty

Possible errors:

- a closing tag which does not match the opening tag on top of the stack
- a closing tag when the stack is empty
- the stack is not empty at the end of the document

Well-Formed Expressions

Or “balanced expressions”:

- `([this is] { a number } 12345) # well-formed`
- `([this is] { a number ) 12345} # not wf`
- `{[(34+4)/5] + 7}/4 # wf`
- `{[(34+4)/5} + 7]/4 # not wf`

(Recursive) Definition of WFE

- The empty sequence is well-formed.
- If A and B are well-formed, then the concatenation AB is well-formed
- If A is well-formed, then [A], {A}, and (A) are well-formed.
- How to we check if an expression is WF?
  - Use a `stack`!

Algorithm for Recognizing WFE

- Read the next delimiter token.
- If it is an open delimeter (i.e. [({)
  - push it into the stack.
- If it is a close delimiter (i.e.]) })
  - match it with a corresponding open delimiter in the stack ([with] and so on).
  - If there is no match → not WF
  - If there is a match, stack.pop() and discard both
- When there is no more token left
  - If the stack is empty → WF
  - If the stack is not empty → not WF

Infix vs Postfix Expressions

- Infix: `5 + 4*5/2 – 3`
- Postfix: `5 4 5 * 2 / + 3 –`

- Infix: `(5+4)*5/2 – 3`
- Postfix: `5 4 + 5 * 2 / 3 –`

Postfix Expression Evaluation Algorithm

- Initialize an empty stack
- While (there is still a token to read)
  - read the token t
  - if t is an operand, push it onto the stack
  - if t is an operator,
    - pop two operands from the stack, compute the result (using t)
      // if there is division by zero, scream
    - push the result back onto the stack
      // if there is less than two operands, scream
- In the end, if there is one number in the stack, output it.
  - // If there is more than one number in the stack, scream.

Example We will see

- Infix:

  $1 - (2 + 3 + (4 - 5 \times 6) \times 7) + 8 \times 9$

- Postfix (Reverse-Polish Notation):

  $1\ \ 2\ \ 3\ +\ 4\ \ 5\ \ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ \ 9\ \times\ +$

Reverse-Polish Notation

Evaluate the following reverse-Polish(postfix) expression using a stack:

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

Reverse-Polish Notation

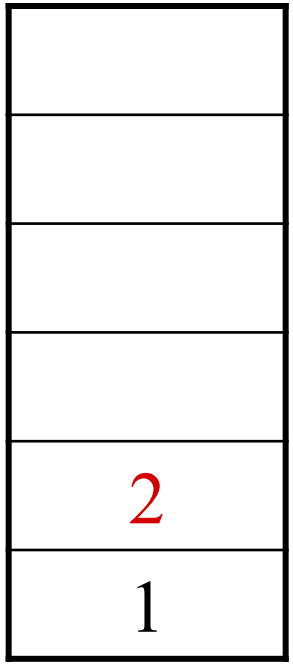
Push 1 onto the stack

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

Reverse-Polish Notation

Push 1 onto the stack

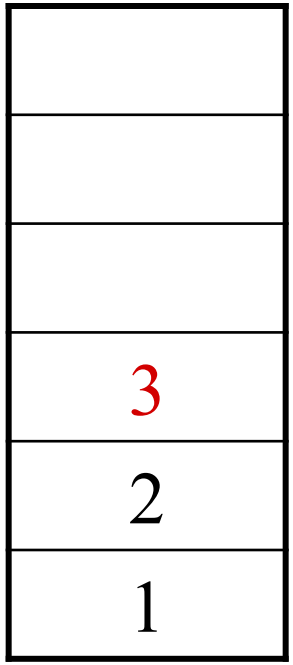
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 3 onto the stack

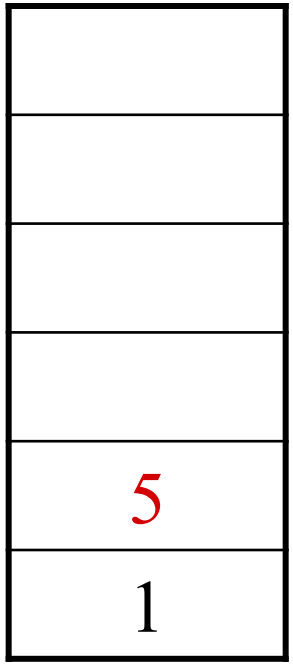
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Pop 3 and 2 and push $2 + 3 = 5$

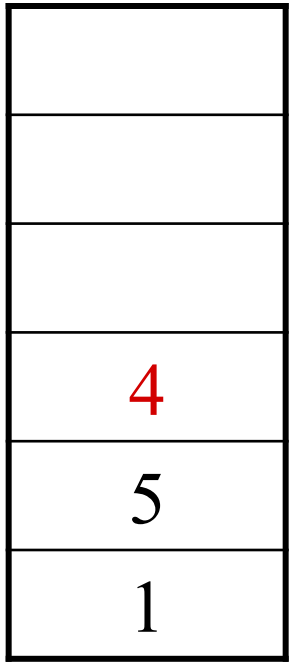
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 4 onto the stack

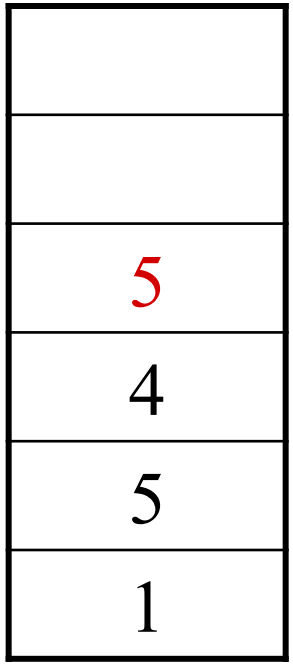
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 5 onto the stack

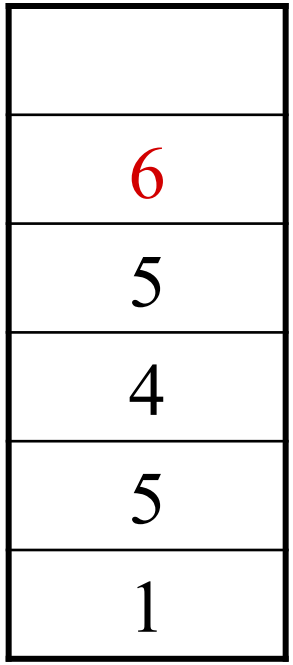
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 6 onto the stack

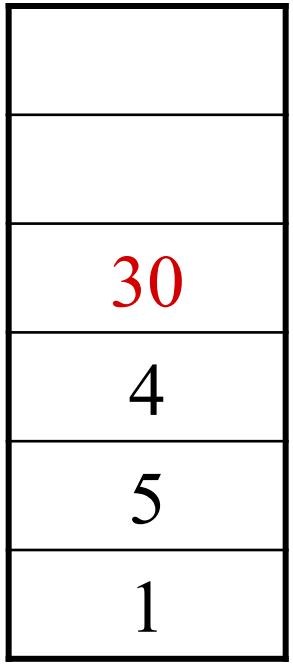
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Pop 6 and 5 and push $5 \times 6 = 30$

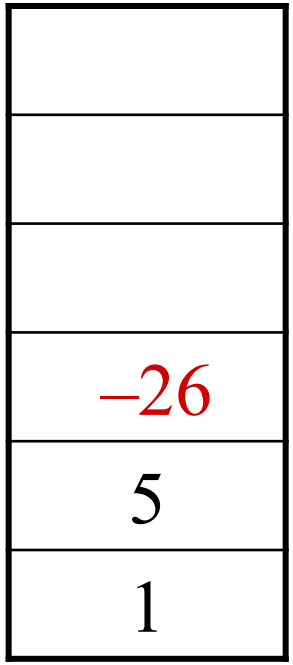
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Pop 30 and 4 and push 4 – 30 = –26

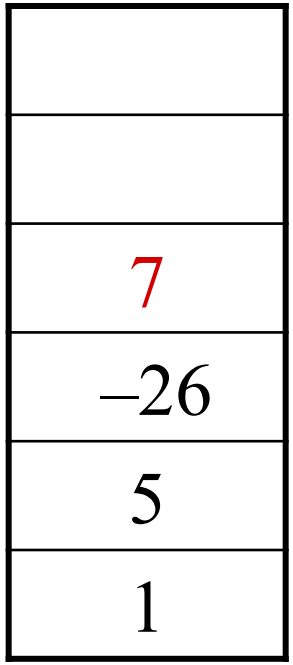
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 7 onto the stack

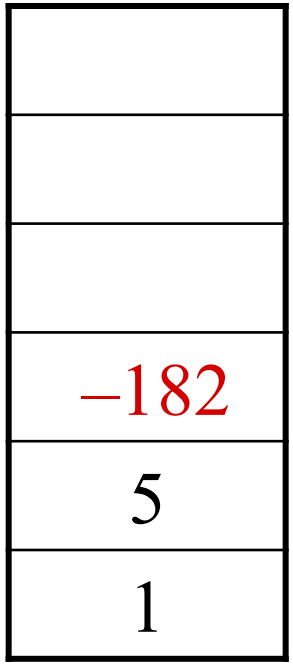
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Pop 7 and –26 and push $-26 \times 7 = -182$

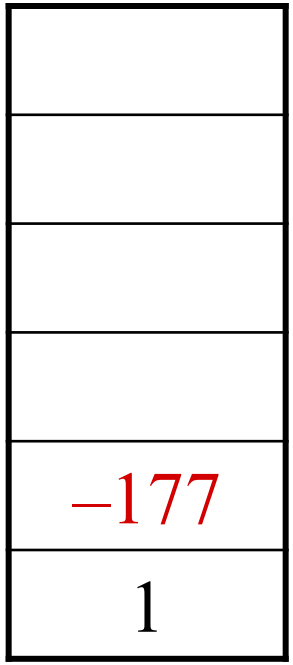
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Pop –182 and 5 and push –182 + 5 = –177

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

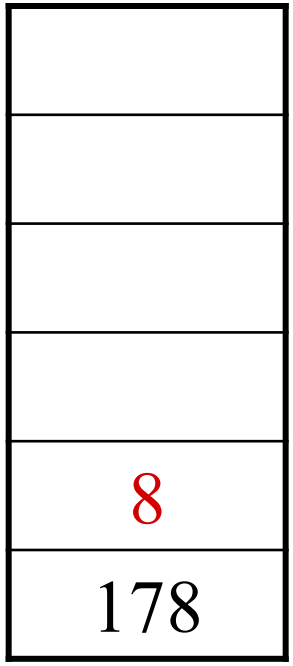
Pop -177 and 1 and push $1 - (-177) = 178$

$1\ \ 2\ \ 3\ +\ 4\ \ 5\ \ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ \ 9\ \times\ +$

Reverse-Polish Notation

Push 8 onto the stack

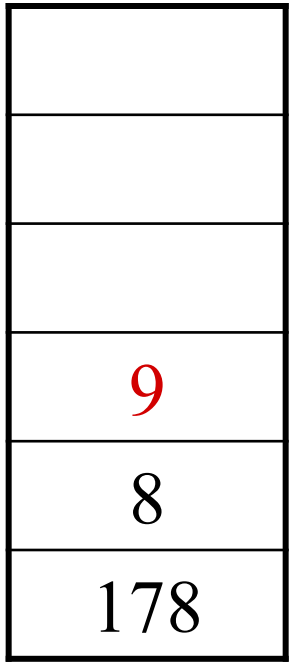
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Push 1 onto the stack

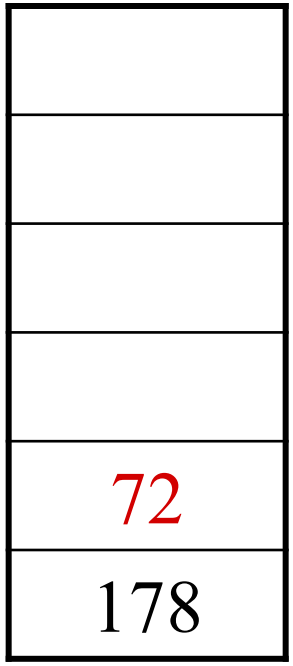
1 2 3 + 4 5 6 × – 7 × + – 8 9 × +



Reverse-Polish Notation

Pop 9 and 8 and push $8 \times 9 = 72$

$1\ \ 2\ \ 3\ +\ 4\ \ 5\ \ 6\ \times\ -\ 7\ \times\ +\ -\ 8\ \ 9\ \times\ +$



Reverse-Polish Notation

Pop 72 and 178 and push $178 + 72 = 250$

1 2 3 + 4 5 6 × – 7 × + – 8 9 × +

How about Infix Expression?

- Shunting yard algorithm
- Convert infix to postfix
- Or, evaluate infix expressions directly

Rough Idea (Shunting Yard Algorithm)

- Use 2 stacks: an operand stack, an operator stack
- If tok is an operand, push it on operand stack
- Else if tok is one of + - * /
  - while (precedence(tok) ≤ precedence(stack.peek())
    - Evaluate stack.peek()
  - Push tok on top of operator stack
- Else if tok is one of ([{
  - Push tok on top of operator stack
- Else if tok is one of)] }
  - Evaluate operators on top until ([{ seen, match up

Acknowledgement

- Douglas Wilhelm Harder.
  - Thanks for making an excellent set of slides for ECE 250 *Algorithms and Data Structures* course

- Prof. Hung Q. Ngo:
  - Thanks for those beautiful slides created for CSC 250 (Data Structures) course at UB.

- Many of these slides are taken from these two sources.