stacks

Well-formed expression

Infix and postfix expressions

Stacks and Applications

Quiz

- Typo:

- Problem d) Third line
- node_ptr = node_ptr . next

- Answer format:
- Tom → Dick → Harry → Sam

Agenda

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

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

STACKS AND APPLICATIONS

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 -->
```

Well-Formed Expressions

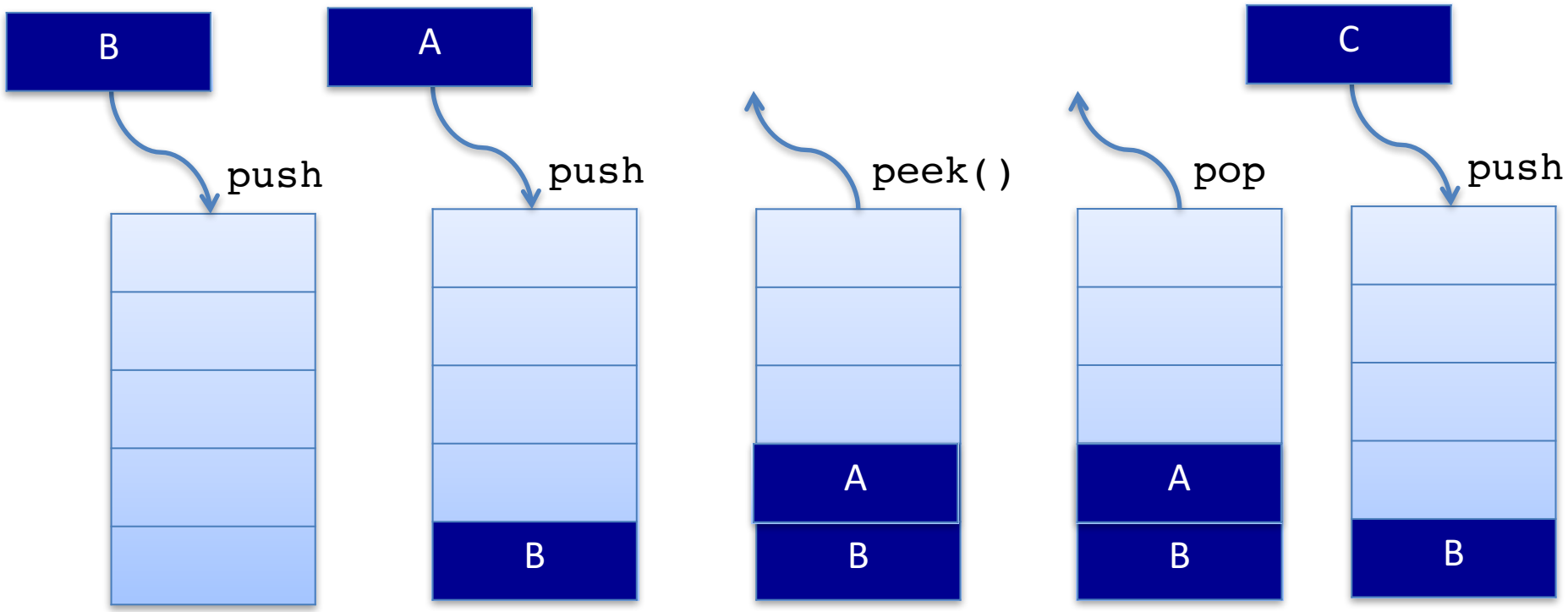
Or "balanced expressions":

- `([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`!

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



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 foul
    - push the result back onto the stack
      // if there is less than two operands, scream foul
- In the end, if there is one number in the stack, output it.
  - // If there is more than one number in the stack, scream foul.

How about Infix Expression?

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

Rough Idea

- 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