Data Structures*

- Example: Sorting elements
  - Input: a set $S$ of numbers
  - Output: elements of $S$ in increasing order

  - Algorithm:
    1. Locate smallest item in $S$
    2. Output smallest item
    3. Delete smallest item from $S$
    4. GOTO 1, while $S \neq \emptyset$

  Key to a good solution: data structure for $S$

Topics for next 5 lectures

- Elementary data structures
  - Stacks and Queues
  - Linked lists
  - Graphs
  - Trees

- Today:
  - Stacks and Queues
  - FIFO vs. LIFO
  - Implementations using arrays
  - Expression Conversion

Stacks and Queues

- Dynamic sets in which the element removed from the set by the Delete operation is prespecified.

- STACK
  - Element deleted is: most recently inserted element

- QUEUE
  - Element deleted is: element that has been in the set the longest
Stack

- **Stack**: A list with *insertion* and *deletion* both take place at one end: the top
  - Main operations
    - **Push**
      - New item added on top of stack, size of stack increases by one
    - **Pop**
      - Top-item removed from stack, decreasing stack size by one
  - Other operations
    - Initialize, Empty, Size, Top, Stack_Top, Display, ...

- Implements a **LIFO** policy
  - New *addition* makes older items inaccessible

---

**Implementing Stack using Array**

<table>
<thead>
<tr>
<th>S</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>3</td>
<td>12</td>
<td>0</td>
<td>42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Top[S]=6

<table>
<thead>
<tr>
<th>S</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>3</td>
<td>12</td>
<td>0</td>
<td>42</td>
<td>17</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Top[S]=8

<table>
<thead>
<tr>
<th>S</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>3</td>
<td>12</td>
<td>0</td>
<td>42</td>
<td>17</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Top[S]=7

---

**Empty(S)**

```plaintext
if top[S]=1 then
  return true
else
  return false
```
Push \((S, x)\)

- if \(\text{STACK-FULL}(S)\) then
  - error “overflow”
- else
  - \(S[\text{top}[S]] := x\)
  - \(\text{top}[S] := \text{top}[S] + 1\)

Pop \((S)\)

- if \(\text{STACK-EMPTY}(S)\) then
  - error “underflow”
- else
  - \(\text{top}[S] := \text{top}[S] - 1\)
  - return \(S[\text{top}[S]]\)

Performance and Limitations of My_Stack

- **Performance**
  - Let \(N\) be the number of elements in the stack
  - The space used is \(O(N)\)
  - Each operation used is \(O(1)\)

- **Limitations**
  - The maximum size of the stack must be defined a priori and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Queue

- A list of elements with
  - Insertion: at end of list, tail
  - Deletion: at start of list, head

- Implements a **FIFO** policy
  - Queues are fair when someone has to wait

- **Examples:**
Implementing a Queue using an Array Q[1..12]

ENQUEUE (Q, x)

```
Q[tail[Q]] := x
if tail[Q] = length[Q] then
tail[Q] := 1
else
tail[Q] := tail[Q]+1
```

DEQUEUE (Q)

```
x := Q[head[Q]]
if head[Q] = length[Q] then
    head[Q] := 1
else
    head[Q] := head[Q]+1
return x
```

Operations on Queues

- Create, Enqueue, Dequeue, Size, Is_Empty, Is_Full, Display

Exercise: Update my_queue to make it a circular queue
Examples Using Stacks

- Infix vs. postfix
  - How to evaluate postfix
  - How to evaluate infix

- Convert Infix to Postfix

Infix vs. Postfix

<table>
<thead>
<tr>
<th>Infix Expressions</th>
<th>Corresponding Postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 + 3 + 4 + 1</td>
<td>5 3 + 4 + 1 +</td>
</tr>
<tr>
<td>(5 + 3) * 10</td>
<td>5 3 + 10 *</td>
</tr>
<tr>
<td>(5 + 3) * (10 - 4)</td>
<td>5 3 10 4 - *</td>
</tr>
<tr>
<td>5 * 3 / (7 - 8)</td>
<td>5 3 * 7 8 - /</td>
</tr>
<tr>
<td>(b * b - 4 * a * c) / (2 * a)</td>
<td>b b * 4 a * c * - 2 a * /</td>
</tr>
</tbody>
</table>

How to Evaluate Postfix

A program can evaluate postfix expressions by reading the expression from left to right

```
for I in 1 .. length loop
  If Is_Number(expr(I)) = true then
    push expr(I)
  end if
  If Is_Operator(expr(I)) then
    pop two numbers from the stack
    perform operation
    push result onto stack
  end if
end loop
-- result is on top of stack
```

Evaluating Infix Expressions

- Need two stacks, one for numbers and one for operators

```
for I in 1 .. length loop
  If Is_Number(expr(I)) = true then
    push expr(I) onto operand_stack
  end if
  If Is_Operator(expr(I)) then
    push expr(I) onto operator_stack
  end if
  If expr(I) = ')' then
    pop 2 numbers from operand_stack
    pop an operator from the operator_stack
    perform operation
    push the result onto the operand_stack
end loop
-- The top of stack contains the result.
```
Infix to Postfix: Example

- Infix Expression
  \[3 + 5 \times 6 - 7 \times (8 + 5)\]

- Postfix Expression
  \[3 \ 5 \ 6 \ + \ 7 \ 8 \ 5 \ + \ * \ -\]

Infix to Postfix

\[
\text{post\_fix} := ""
\]
\[
\text{Create}(\text{Op\_Stack})
\]
\[
\text{for} \ I \ \text{in} \ 1 \ .. \ \text{Length} \ \text{loop}
\]
\[
\quad \text{If} \ \text{Is\_Operand}(\text{expr}(I)) = \text{true} \ \text{then}
\quad \quad \text{Append}(\text{post\_fix}, \ \text{expr}(I))
\]
\[
\quad \text{If} \ \text{Is\_Operator}(\text{expr}(I)) = \text{true} \ \text{then}
\quad \quad \text{Process\_Next\_Operator}(\text{expr}(I))
\]
\[
\quad \text{end loop}
\]
\[
\text{Process\_Next\_Operator}
\]
\[
\text{Done} := \text{False}
\]
\[
\text{loop}
\]
\[
\quad \text{If} \ \text{Is\_Empty}(\text{Op\_Stack}) \ \text{or} \ \text{next\_op} \ \text{is} \ `'('\,
\quad \quad \text{push} \ \text{next\_op} \ \text{onto} \ \text{Op\_Stack}
\quad \quad \text{set} \ \text{Done} \ \text{to} \ \text{True}
\]
\[
\quad \text{Elsif} \ \text{precedence}(\text{next\_op}) > \ \text{precedence}(\text{top\_operator})
\quad \quad \text{Push} \ \text{next\_op} \ \text{onto} \ \text{Op\_Stack}
\quad \quad \text{\quad \quad \text{-- ensures higher precedence operators evaluated first}}
\quad \quad \text{Set} \ \text{Done} \ \text{to} \ \text{True}
\]
\[
\quad \text{Else}
\quad \quad \text{Pop} \ \text{the} \ \text{operator\_stack}
\quad \quad \text{If} \ \text{operator\_popped} \ \text{is} \ `'('\,
\quad \quad \quad \text{set} \ \text{Done} \ \text{to} \ \text{True}
\quad \quad \quad \text{Else}
\quad \quad \quad \quad \text{append} \ \text{operator\_popped} \ \text{to} \ \text{post\_fix} \ \text{string}
\quad \quad \quad \quad \text{\quad \quad \text{exit} when} \ \text{Done} = \ \text{True}
\]
\[
\quad \text{end loop}
\]