ADTs and Stacks

Lecture 6

1107186 – Estruturas de Dados

Prof. Christian Azambuja Pagot
CI / UFPB

Abstract Data Type

- ADT is a **mathematical (theoretical) model**.
- When we implement a ADT, we try to **mimic** is **semantics**.
- We can **implement** an **ADT** using:
  - Computer **data types**;
  - Computer **data structures**.

Examples of ADTs

- Lists
- Stacks
- Deques
- Etc.

What is a Stack?

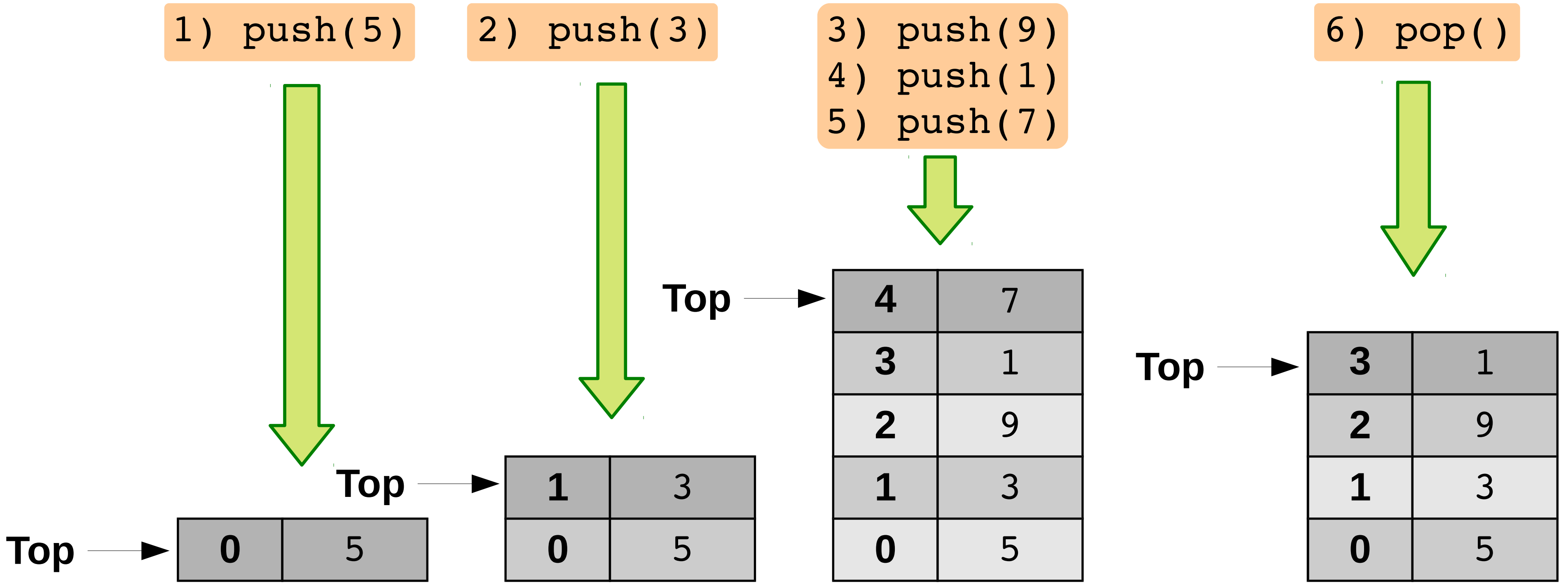
Gene Daniels, 1972.

What is a Stack?

- A **stack** is an **abstract data type** where elements can be **inserted** (*pushed*) and **removed** (*popped*) according to the following policies:
  - **Push** inserts a new item into the **top** of the stack.
  - **Pop** removes the item at the **top** of the stack (except when the stack is empty).
  - The above insert/remove policy is also called **LIFO** (**Last In-First Out**).

What is a Stack?

- **Example:**



An Useful Operation

- Often we want **just to check** the value of the topmost item on the stack. One **possible** approach to that would be:

```
x = pop()
push(x)
...
```

- An **easier** way would be to have a function that returns the value at the top **without actually popping** the value:

```
x = top()
...
```

Stacks in Practice

- **Two example applications:**
  - Parenthesis matching.
  - Reverse Polish Notation (RPN)

Parenthesis Matching

- An example arithmetic expression:

$$((a+b)\times(c+d)\times e)-(f+g)$$

- We must **check** if the parenthesis are **correctly grouped**:
  - # of left parenthesis = # of right parenthesis.
  - All right parenthesis are preceded by a corresponding left parenthesis.

Parenthesis Matching

- If each left parenthesis counts **1**, and each right parenthesis counts **-1**, the **overall sum** must be **0** (zero).
- At any position in the expression, the partial sum **must not be negative**!

How to keep track of the appropriate **counting** and **ordering**?

Guess what!

Parenthesis Matching

- Each time a **left** parenthesis is found, it is **pushed** into the **stack**.
- Each time a **right** parenthesis is found, one item is **popped** from the **stack**.
- At the end, if the expression is **correct**, the stack must be **empty**.

Parenthesis Matching

- **Example**:

$$((a+b)\times(c+d)\times e)-(f+g)$$

Push('(')

0	(

Parenthesis Matching

- **Example**:

$$((a+b)\times(c+d)\times e)-(f+g)$$

Push('(')

1	(
0	(

Parenthesis Matching

- **Example**:

$$((a+b)\times(c+d)\times e)-(f+g)$$

Pop()

0	(

Parenthesis Matching

- **Example**:

$$\big(\big(a+b\big)\times\big(c+d\big)\times e\big)-\big(f+g\big)$$

Push('(')

1	(
0	(

Parenthesis Matching

- **Example**:

$$((a+b)\times(c+d\boxed{)}\times e)-(f+g)$$

Pop()

0	(

Parenthesis Matching

- **Example**:

$$\big(\big(a+b\big)\times\big(c+d\big)\times e\big)-\big(f+g\big)$$

Pop()

Parenthesis Matching

- **Example**:

$$\left(\left(a+b\right)\times\left(c+d\right)\times e\right)-\left(f+g\right)$$

Push('(')

0	(

Parenthesis Matching

- **Example**:

$$\big(\big(a+b\big)\times\big(c+d\big)\times e\big)-\big(f+g\boxed{\big)}$$

Pop()

Results:

- Stack is **empty**.
- **No underflows** during process.

Conclusion:
Valid expression!

Mathematical Notation

- One aspect of the mathematical notation is related to the position of the **operator** relative to its **operands** in an expression.
- There are three possibilities:
  - **Infix** (most common): 5 + 8
  - **Prefix**: + 5 8
  - **Postfix**: 5 8 +

Reverse Polish Notation (RPN)

- RPN is a **postfix**, **parenthesis free**, mathematical notation where the operator **follows** their **operands**.
- **Example**:
  - Usual arith. expression: $(a+b)\times(c+d)$
  - RPN version: $ab+cd+\times$

Evaluating an Expression in RPN

- **OBS:** Each operator **refers** to the two **previous operands**.
- **Algorithm:**

```
for each symbol in expression
  if (symbol == operand)
    stack.push(symbol);
  else
    if (symbol == operator)
      op1 = stack.pop();
      op2 = stack.pop();
      result = eval(op1 operator op2);
      stack.push(result);
    end if
  end if
end for
```

At the end, the **stack** will **contain** the final **result**!

Evaluating an Expression in RPN

- **Example:**

```
for each symbol in expression
  if (symbol == operand)
    stack.push(symbol);
  else
    if (symbol == operator)
      op1 = stack.pop();
      op2 = stack.pop();
      result = op1 operator op2;
      stack.push(result);
    end if
  end if
end for
```

Usual not.: $(5+3)\times(5+8)$

RPN not.: $5\;3+5\;8+\times$

0	5

curr. step

Evaluating an Expression in RPN

- **Example:**

Usual not.: $(5+3)\times(5+8)$

RPN not.: $5\;3+5\;8+\times$

```
for each symbol in expression
  if (symbol == operand)
    stack.push(symbol);
  else
    if (symbol == operator)
      op1 = stack.pop();
      op2 = stack.pop();
      result = op1 operator op2;
      stack.push(result);
    end if
  end if
end for
```

0	5

prev. step

1	3
0	5

curr. step

Evaluating an Expression in RPN

- **Example:**

Usual not.: $(5+3)\times(5+8)$

RPN not.: $5\;3\;+\;5\;8\;+\;\times$

```
for each symbol in expression
   if (symbol == operand)
      stack.push(symbol);
   else
      if (symbol == operator)
         op1 = stack.pop();
         op2 = stack.pop();
         result = op1 operator op2;
         stack.push(result);
      end if
   end if
end for
```

1	3
0	5

prev. step

0	8

curr. step

Evaluating an Expression in RPN

- **Example:**

```
for each symbol in expression
  if (symbol == operand)
    stack.push(symbol);
  else
    if (symbol == operator)
      op1 = stack.pop();
      op2 = stack.pop();
      result = op1 operator op2;
      stack.push(result);
    end if
  end if
end for
```

Usual not.: $(5+3)\times(5+8)$

RPN not.: $5\,3+5\,8+\times$

0	8

prev. step

1	5
0	8

curr. step

Evaluating an Expression in RPN

- **Example:**

Usual not.: $(5+3)\times(5+8)$

RPN not.: $5\,3+5\,8+\times$

```
for each symbol in expression
  if (symbol == operand)
    stack.push(symbol);
  else
    if (symbol == operator)
      op1 = stack.pop();
      op2 = stack.pop();
      result = op1 operator op2;
      stack.push(result);
    end if
  end if
end for
```

1	5
0	8

prev. step

2	8
1	5
0	8

curr. step

Evaluating an Expression in RPN

- **Example:**

Usual not.: $(5+3)\times(5+8)$

RPN not.: $5\,3+5\,8+\times$

```
for each symbol in expression
   if (symbol == operand)
      stack.push(symbol);
   else
      if (symbol == operator)
         op1 = stack.pop();
         op2 = stack.pop();
         result = op1 operator op2;
         stack.push(result);
      end if
   end if
end for
```

2	8
1	5
0	8

prev. step

1	13
0	8

curr. step

Evaluating an Expression in RPN

- **Example:**

Usual not.: $(5+3)\times(5+8)$

RPN not.: $5\ 3 + 5\ 8 + \times$

```
for each symbol in expression
   if (symbol == operand)
      stack.push(symbol);
   else
      if (symbol == operator)
         op1 = stack.pop();
         op2 = stack.pop();
         result = op1 operator op2;
         stack.push(result);
      end if
   end if
end for
```

1	13
0	8

prev. step

0	104

curr. step

Stack Implementation in C

- A stack is a **ordered** list of **values**.
- This approach implements the stack as an **array**.
  - There is no need to allocate space for each new item **:)**
  - We don't have to worry about deallocating space **:)**
  - Space that is not used is wasted **:(**
  - The stack will be limited in size **:(**

Stack Implementation in C

- A **struct** will hold the **array** and the **top indicator**:

C code excerpt:

```c
#define STACK_MAX_SIZE 100

struct Stack{
    char v[STACK_MAX_SIZE];
    int top;
};

...

struct Stack s;
```

Stack Implementation in C

- **Push operation**:

C code excerpt:

```
void Push(struct Stack* s, char c)
{
    if (s->top < (STACK_MAX_SIZE-1))
    {
        s->top++;
        s->v[s->top] = c;
    }
    else
    {
        printf("Stack overflow!\n");
        exit(1);
    }
};
```

Stack Implementation in C

- **Pop operation**:

C code excerpt:

```
char Pop(struct Stack* s)
{
    if (s->top >= 0)
    {
        char tmp = s->v[s->top];
        s->top--;
        return tmp;
    }
    else
    {
        printf("Stack underflow!\n");
        exit(1);
    }
};
```

Stack Implementation in C

If not **properly initialized**, the stack may **not work**!

- **Use example**:

C code excerpt:

```
int main(void)
{
    struct Stack s = {.top = -1}; // or s.top = -1

    Push(&s, 'U');
    Push(&s, 'F');
    Push(&s, 'P');
    Push(&s, 'B');

    Pop(&s);
    Pop(&s);

    return 0;
}
```

Stack creation and initialization.

Pointers as arguments: we need to change the Stack state!

Think about it!

- **Considering the presented C-based stack implementation**:
  - Initialization is completely manual and prone to errors.
    - **How to automate it**?
  - Theoretically, stacks have no upper limits (one should be able to keep 'pushing' infinitely!).
    - **How we could improve our implementation with respect to the maximum size limit**?
  - We may want to 'stack' anything (numbers, chars, etc.).
    - **Do we have to implement one different stack (with push(), pop()) for each data type?**