1. **Stack**: “last in, first out” abstract data type.
   1. Functions
      1. With modifications
         1. push(): puts item on top.
         2. pop(): removes top item.
      2. Without modifications
         1. top(): returns value stored at the top of the stack without popping it off.
         2. empty(): Boolean function that returns true if empty.
         3. full(): Boolean function that returns true if full.
            1. In theory, no stack should ever become full.
            2. Any implementation limits the number of elements that can be stored.
   2. Code
      1. typedef struct node {
         1. struct node\* front;
         2. int size;
      2. }stack;
   3. Components
      1. size
      2. data
      3. pointer to top
   4. Implementations
      1. Linked List
         1. Functions
            1. push() = addToFront()

int push(stack\* myStack, int item) {

stack\* tmp = malloc(sizeof(node));

//Empty case

if (tmp == NULL) return 0;

//Create node; link to old front

tmp->data = item;

tmp->next = myStack->top;

//Reset front and size

myStack->top = tmp;

myStack->size++;

return 1;

}

* + - * 1. pop() = delete(first front)

node\* pop(stack\* myStack) {

//Empty case

If (myStack->size == 0) return NULL;

//Store return node

Node\* retval = myStack->top;

//Update new top

myStack->top = myStack->top->next;

//Update size

myStack->size--;

//To be safe, NULL this out

retval->next = NULL;

return retval;

}

* + - * 1. top()

int top(stack\* myStack) {

if (mystack->size == 0) return -1;

return mystack->top->data;

}

* + - * 1. size()

int size(stack\* myStack) {

return myStack->size;

}

* + - * 1. void init(stack\* myStack) {

top = NULL;

size = 0;

* + - * 1. }
      1. Types
         1. Call stack: stack for function calls.
         2. Operator stack
         3. Operand stack
    1. Array
       1. Functions
          1. push()

Check for full stack if dynamically-allocated to avoid crashes.

* 1. Algorithms
     1. Evaluating postfix expression (operand stack)
        1. Infix: how mathematical expressions are generally written.
        2. Postfix: unique unambiguous way to parenthesize an expression by writing operands last.
           1. Syntax: number number operator

Like a calculator

PEMDAS

* + - * 1. Example: 3 6 2 \* + 5 – 1 1 + /

\* operates on 6 and 2

+ operates on 3 and (2 \* 6)

– operates on 5 and (3 + (2 \* 6))

+ operates on 1 and 1

/ operates on (1+1) and (5 – (3 + (2 \* 6))

* + - 1. In stacks
         1. Rules

Read an operand 🡪 push onto stack

Read an operator 🡪 pop last 2 items off the stack op2, followed by op1. Calculate op1 op op2 and push this value onto the stack.

If you ever try to pop an empty stack, the expression is invalid postfix expression.

If after running operations you end up with a stack size > 1, it’s also invalid.

Should only end up with 1 answer

* + - * 1. Example: 3 6 2 \* + 5 – 1 1 + /

Stack

Step 1: add 3, 6, and 2 to the stack

2

6

3

Step 2: evaluate 6 \* 2; pop 2 and 6 off stack.

12

3

Step 3: evaluate 12 + 3; pop 3 and 12 off stack; add 5 to the stack.

5

15

Step 4: evaluate 15 – 5; pop 15 and 5 off stack; add 1 and 1 to stack

10

1

1

Step 4: evaluate 1 + 1; pop 1 and 1 off stack; add 2 to the stack

10

2

Step 5: evaluate 10 / 2; pop 2 and 10 off stack.

5

Step 6: return 5

* + 1. Converting infix to postfix (operator stack)
       1. Algorithm
          1. Rules

Open parenthesis 🡪 push onto stack

Operand 🡪 place into expression in order.

Close parenthesis 🡪 pop items off stack, placing each in the expression until we hit the first open parenthesis.

Operator 🡪 pop off the stack each operator of equal or higher precedence, placing each into the expression.

Stop popping when you reach…

an operator of lower precedence.

a parenthesis.

the end of the stack.

Push this operator onto the stack.

End 🡪 pop off remaining operators and place in the expression.