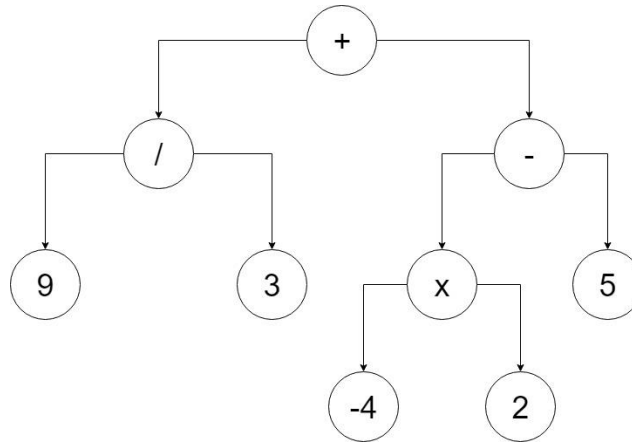


Recitation 3 – Stacks & Queues

A note on Prefix/Infix/Postfix notations

Given the expression tree as below, write down the actual expression:



a) In infix notation: $9 / 3 + -4 * 2 - 5$

b) In postfix notation: $9\ 3\ /\ -4\ 2\ *\ 5\ -\ +$

c) In prefix notation: $+ / 9\ 3 - * -4\ 2\ 5$

[Postfix notation exercise] Evaluate $5\ 2\ 4\ ^\wedge\ 3\ *\ +\ 9\ -$ where $^\wedge$ operator denotes exponentiation.

$5\ 16\ 3\ *\ +\ 9\ -$

$5\ 48\ +\ 9\ -$

$53\ 9\ -$

44

Note: Conversions between all these notations can be implemented using stacks. However, more intuition will possibly develop when traversals of binary trees are investigated.

3.23 a. Write a program to convert an infix expression that includes (,), +, -, *, and / to postfix. b. Add the exponentiation operator to your repertoire.

Infix to Postfix - Algorithm

```

stack ← ∅;
while (! eos(expr)){
    read(char);
    if (isOperand(char))
        output(char);
    else if (char == "(")
        while ((!isEmpty(stack)) && (sc=topAndPop(stack)) != "("))
            output(sc);
    else
        while ( (!isEmpty(stack)) &&
            (inStackPriority(top(stack)) >= (outStackPriority(char))))
            output(topAndPop(stack));
        push(char, stack);
}
while (!isEmpty(stack))
    output(topAndpop(stack));

```

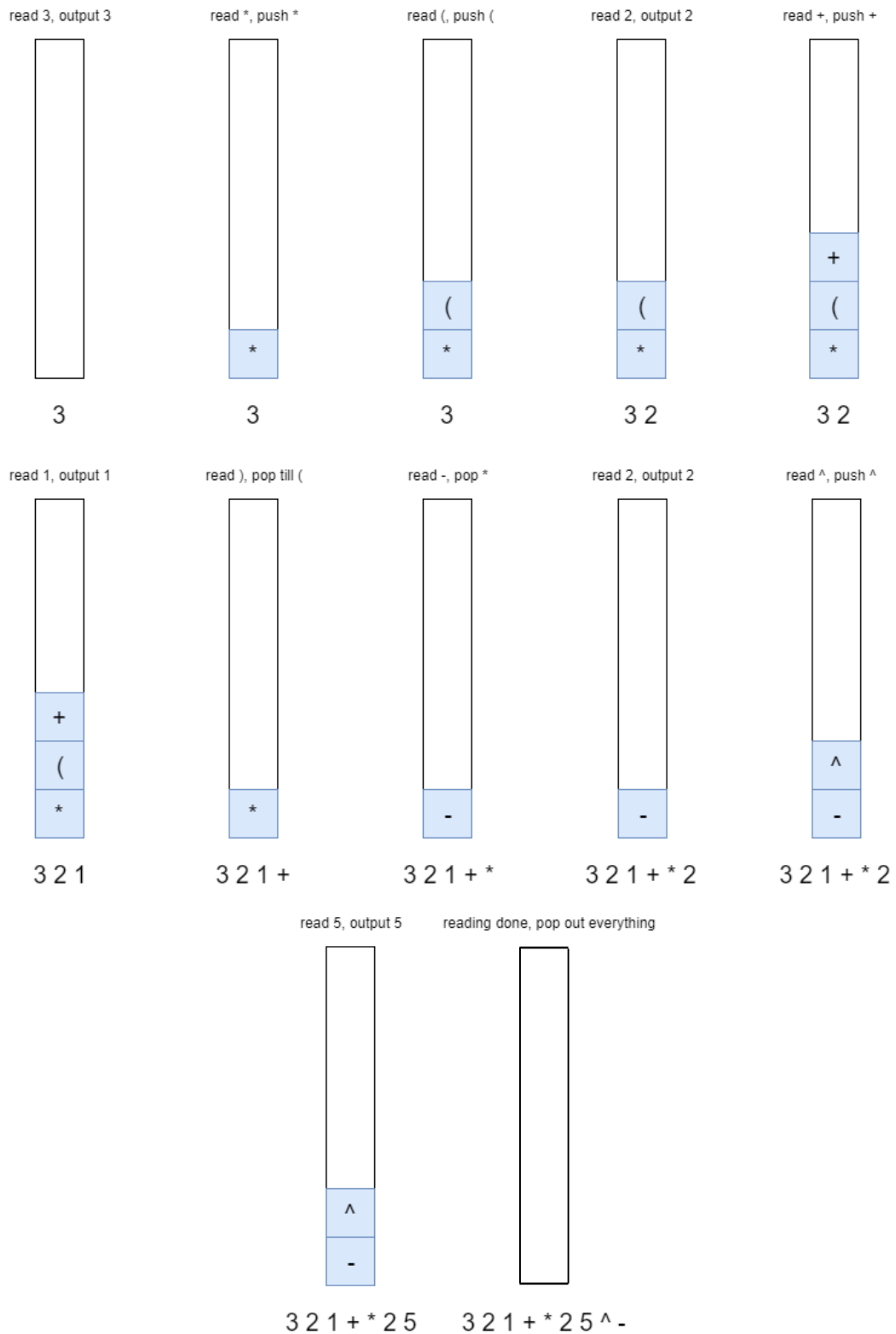
	inStackPriority	outStackPriority
** (RL)	4	5
*, / (LR)	3	3
+, - (LR)	2	2
(	1	6

inStackPriority("(")=very low

outStackPriority("(")=very high

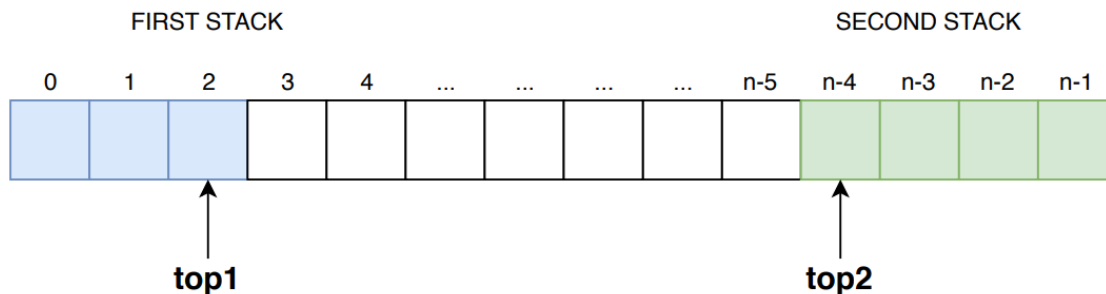
(Explain in words, do not write down the code on board)

Example: Convert $3 * (2 + 1) - 2 ^ 5$ to postfix notation using the algorithm given.



3.24 Write routines to implement two stacks using only one array. Your stack routines should not declare an overflow unless every slot in the array is used.

```
public TwoStacks(int n)
{
    this.two_stacks = new Integer[n];
    this.size = n;
    this.top1 = -1;
    this.top2 = n;
}
```



```
is_empty:
return check top1 == -1;

push: (check overflow first!)
if(top1+1 != top2)
two_stacks[++top1] = value;

pop: (check emptiness first!)
if(!is_empty)
int ret = two_stacks[top1];
two_stacks[top1--] = null;
return ret;
```

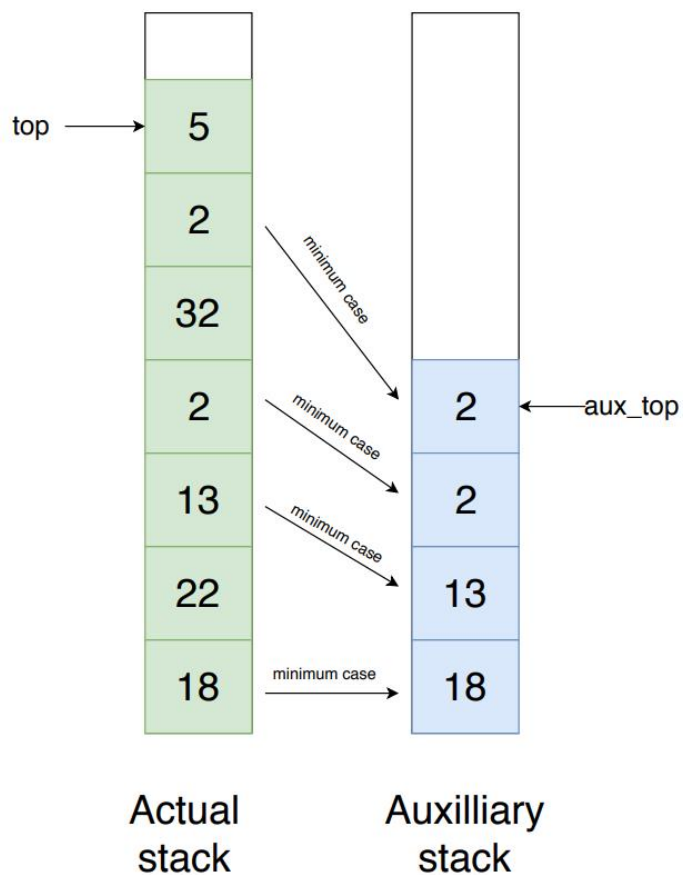
```
is_empty:
return top2 == size;

push: (check overflow first!)
if(top1+1 != top2)
two_stacks[--top2] = value;

pop: (check emptiness first!)
if(!is_empty)
int ret = two_stacks[top2];
two_stacks[top2++] = null;
return ret;
```

3.25 a. Propose a data structure that supports the stack push and pop operations and a third operation `findMin`, which returns the smallest element in the data structure, all in $O(1)$ worst-case time.

A special stack that has two stacks as members will be used, namely **stack** and **aux**



```
push:
if(value <= aux.peek())
    stack.push(value);
    aux.push(value);
else
    stack.push(value);
```

Minimum case

```
pop:
p = stack.pop();
if(p == aux.peek())
    aux.pop();
```

```
min:
return aux.peek();
```

Note that you also need to handle emptiness in full implementation.

Combining it with the last exercise we can do all of this within a single array of size $2n$.

This data structure can handle operations in $O(1)$ time, but it needs extra $O(n)$ space. Can you do it in $O(1)$ extra space? (as an exercise, you can get help from the Internet)