Array, Stacks, and Queues

Stacks: Stack, operations, implementation using arrays, recursion, ToH, linked list and Stack Applications: Infix to postfix expression conversion, Evaluation of Postfix expressions, balancing the symbols.

Stack

Infix to Postfix

Infix	Postfix
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +

A + B * C → (A + (B * C)) → (A + (B C *)) → A B C * +

A + B * C + D → ((A + (B * C)) + D) → ((A + (B C*))+ D) → ((A B C *+) + D) → A B C * + D +

Infix to postfix conversion

- Use a stack for processing operators (push and pop operations).
- Scan the sequence of operators and operands from left to right and perform one of the following:
  - output the operand,
  - push an operator of higher precedence,
  - pop an operator and output, till the stack top contains operator of a lower precedence and push the present operator.

The algorithm steps

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the **incoming symbol has higher precedence than the top of the stack, push it on the stack**.
6. If the incoming symbol **has equal precedence with the top of the stack, use association**. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. **If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator**. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Infix to Postfix Rules

```
stack s
char  ch, element

while(tokens are available)   {
    ch = read(token);
    if(ch is operand)   {
              print ch ;
    } else  {
              while(priority(ch) <= priority(top most stack))  {
                        element = pop(s);
                        print(element);
              }
              push(s,ch);
     }
}
while(!empty(s))  {
              element = pop(s);
              print(element);
}
```

Infix to Postfix Conversion

Requires operator precedence information

Operands:

Add to postfix expression.

Close parenthesis:

pop stack symbols until an open parenthesis appears.

Operators:

Pop all stack symbols until a symbol of lower precedence appears. Then push the operator.

End of input:

Pop all remaining stack symbols and add to the expression.

Infix to Postfix Rules

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

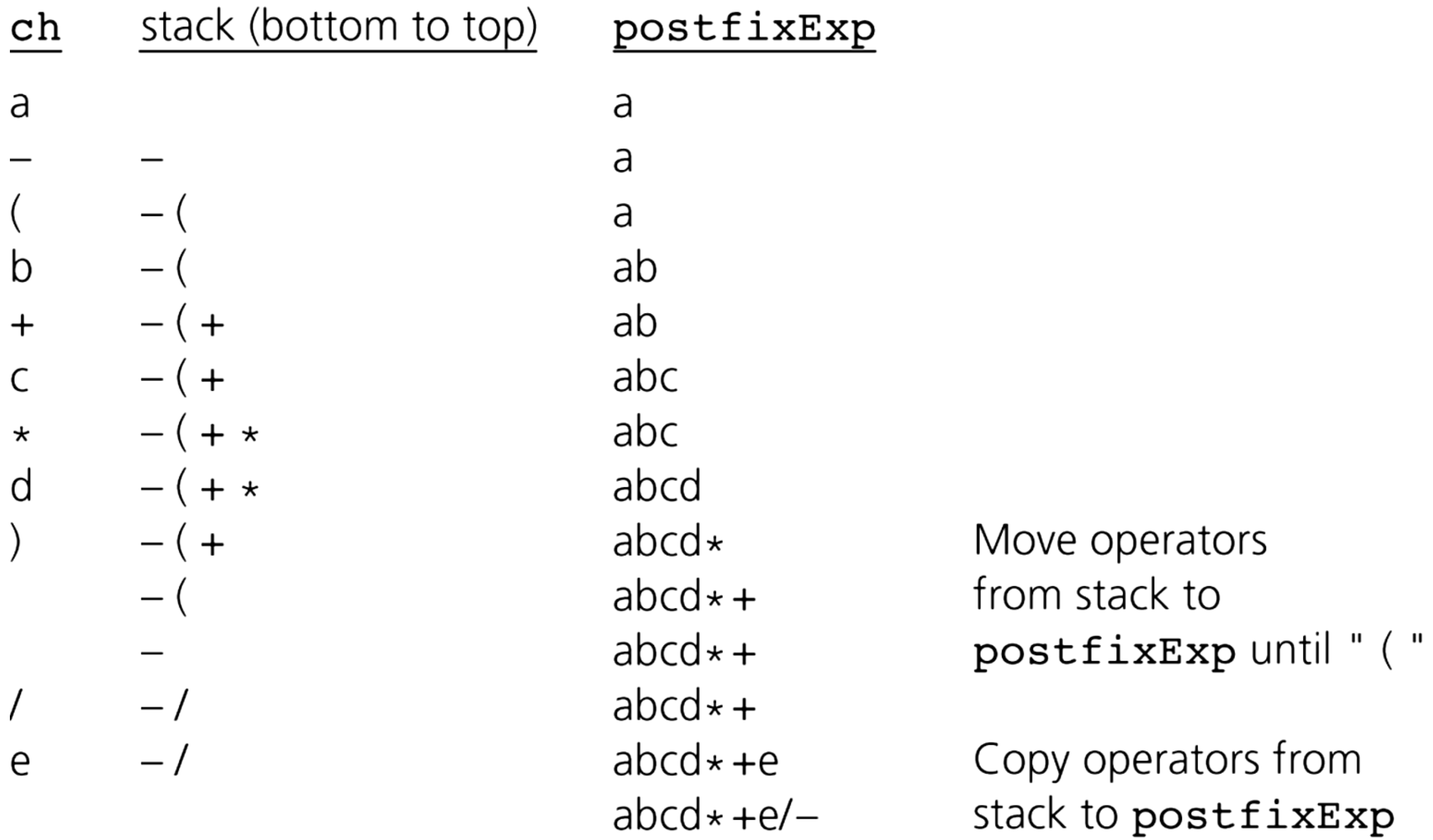
	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Convert the infix into its equivalent postfix expression

```
Algorithm to convert an Infix notation into postfix notation
Step 1: Add ‘)” to the end of the infix expression
Step 2: Push “(“ on to the stack
Step 3: Repeat until each character in the infix notation is scanned
          IF a “(“ is encountered, push it on the stack
          IF an operand (whether a digit or an alphabet) is encountered,
        add it to the postfix expression.
          IF a “)” is encountered, then;
          a. Repeatedly pop from stack and add it to the postfix expression until a “(” is
          encountered.
      b. Discard the “(“. That is, remove the “(“ from stack and do not
      add it to the postfix expression
        IF an operator X is encountered, then;
          a Repeatedly pop from stack and add each operator (popped from the stack) to the
          postfix expression which has the same precedence or a higher precedence than X
      b. Push the operator X to the stack
Step 4: Repeatedly pop from the stack and add it to the postfix expression
     until the stack is empty
Step 5: EXIT
```

Converting Infix to Equivalent Postfix Expressions

A trace of the algorithm that converts the infix expression *a* - (*b* + *c* * *d*)/*e* to postfix form



`ch`	stack (bottom to top)	`postfixExp`	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	`postfixExp` until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to `postfixExp`

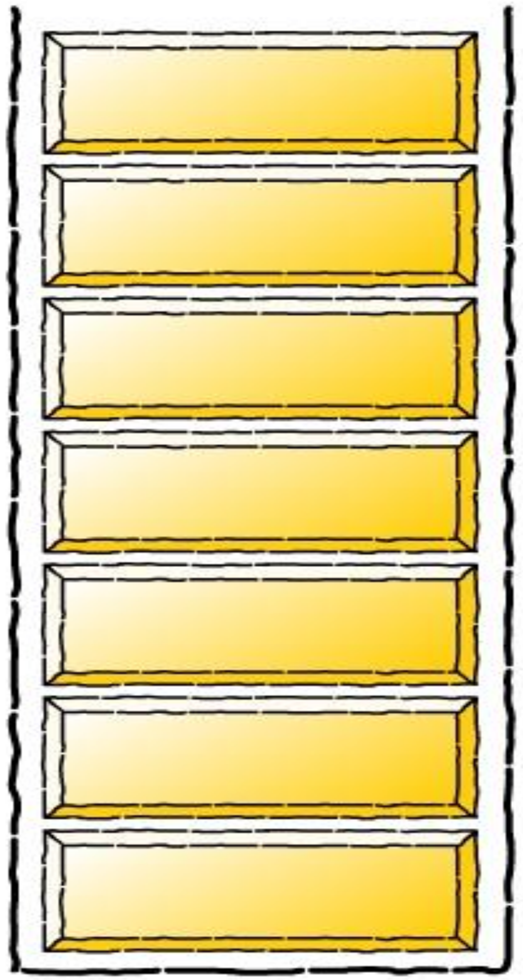
Another example

(A+(B+C*D)*E)

1.	(	$(	Nothing
2.	A	$(	A
3.	+	$(+	A
4.	(	$(+(	A
5.	B	$(+(	AB
6.	+	$(+(+	AB
7.	C	$(+(+	ABC
8.	*	$(+(+*	ABC
9.	D	$(+(+*	ABCD
10.	)	$(+	ABCD*+
11.	*	$(+*	ABCD*+
12.	E	$(+*	ABCD*+E
13.	)	$	ABCD*+E*+

One more Infix to postfix using Stack

Infix to postfix conversion

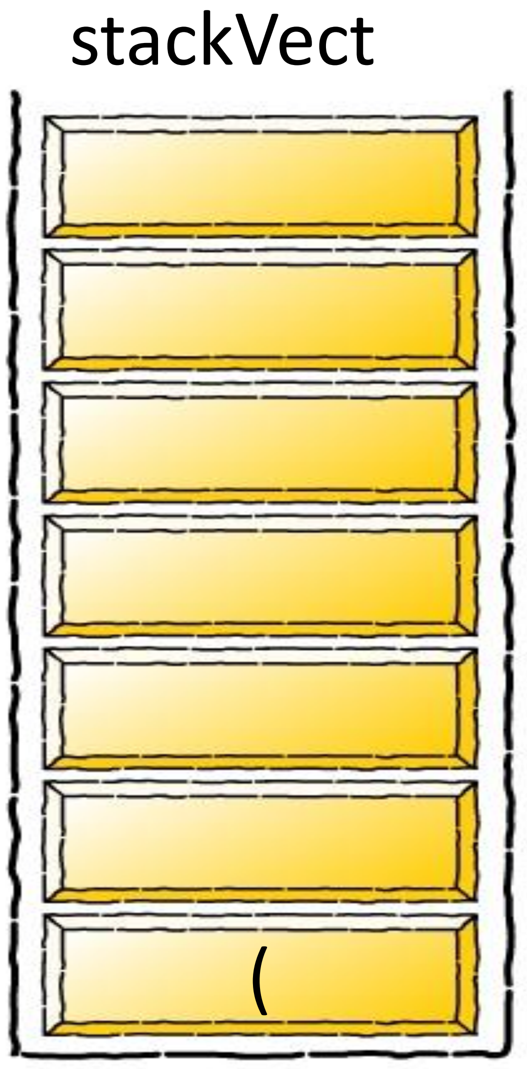


infixVect

(a + b - c) * d – (e + f)

postfixVect

Infix to postfix conversion



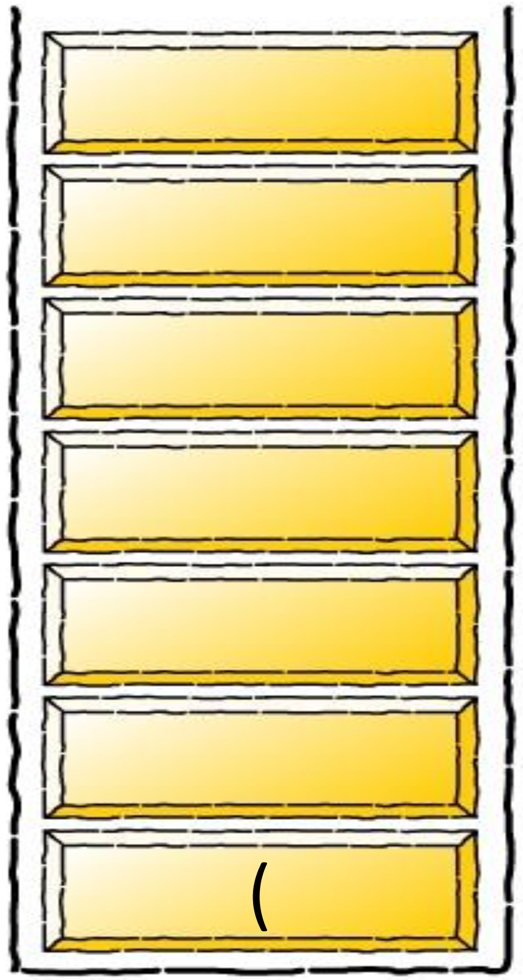
infixVect

a + b - c) * d – (e + f)

postfixVect

Infix to postfix conversion

stackVect



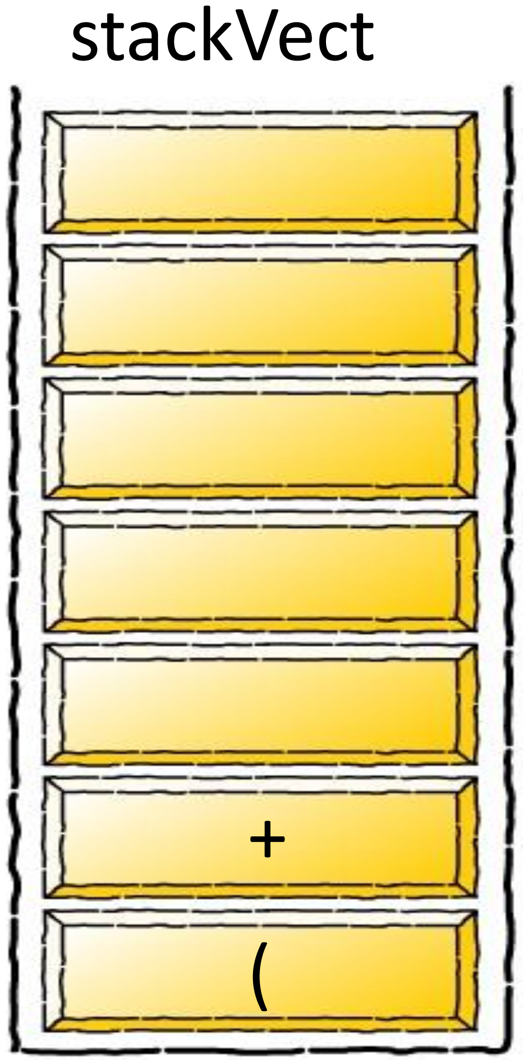
infixVect

+ b - c) * d – (e + f)

postfixVect

a

Infix to postfix conversion



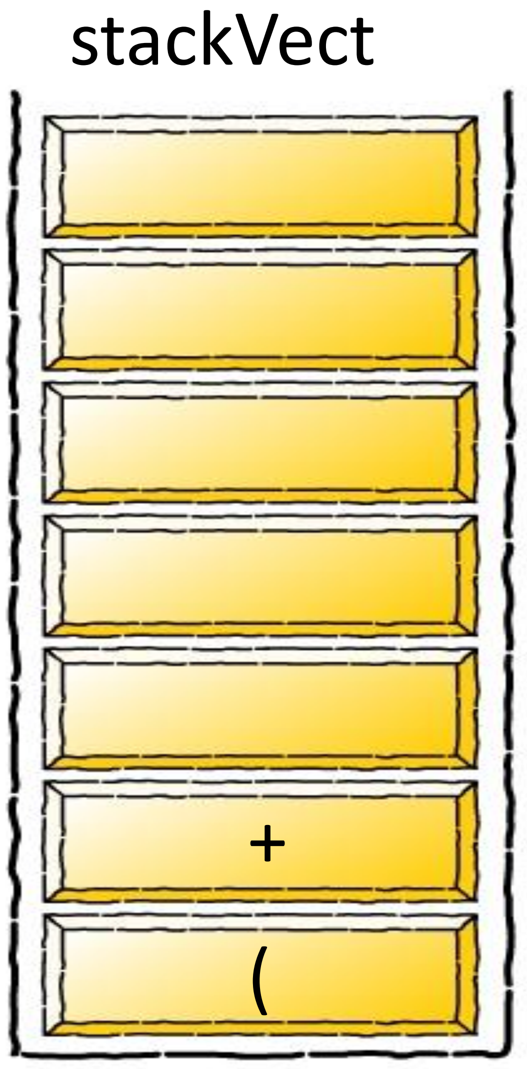
infixVect

b - c) * d – (e + f)

postfixVect

a

Infix to postfix conversion



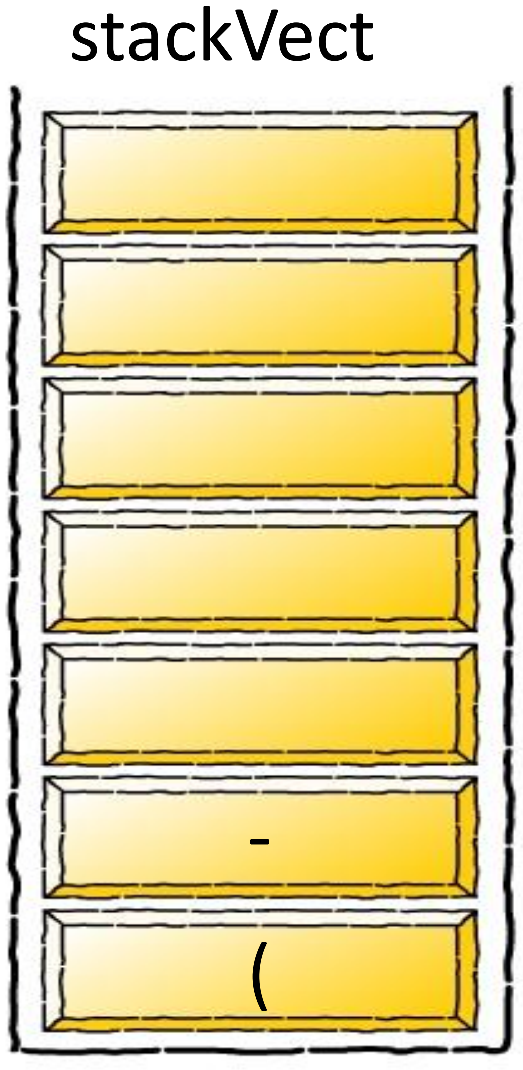
infixVect

- c) * d – (e + f)

postfixVect

a b

Infix to postfix conversion



infixVect

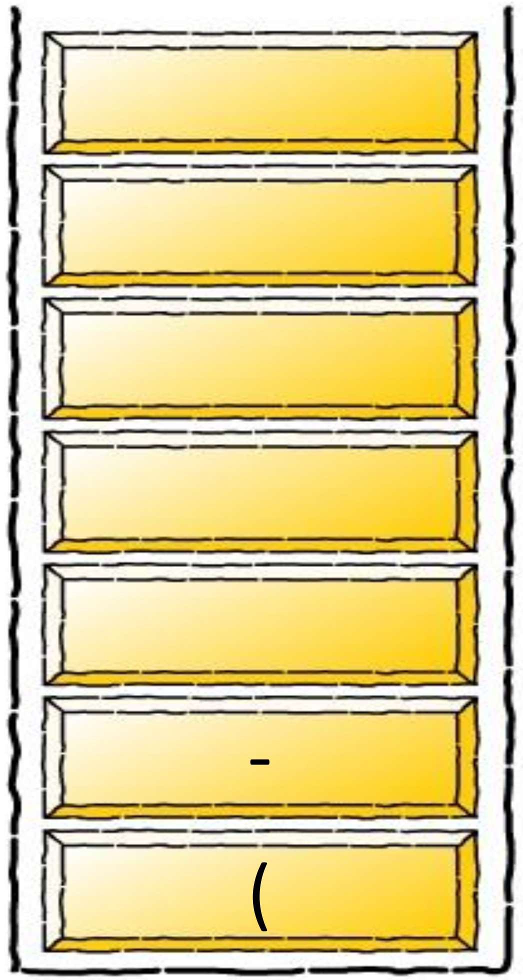
c) * d – (e + f)

postfixVect

a b +

Infix to postfix conversion

stackVect



infixVect

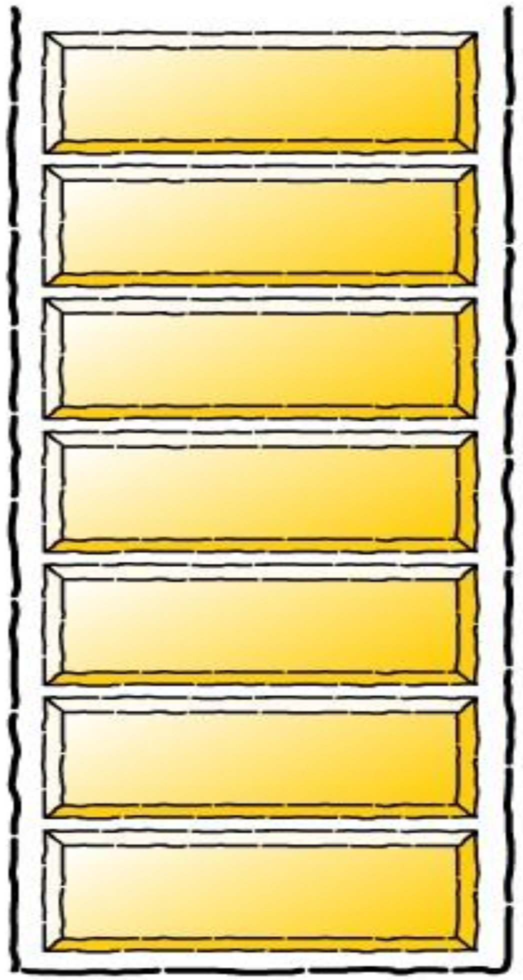
) * d – (e + f)

postfixVect

a b + c

Infix to postfix conversion

stackVect



infixVect

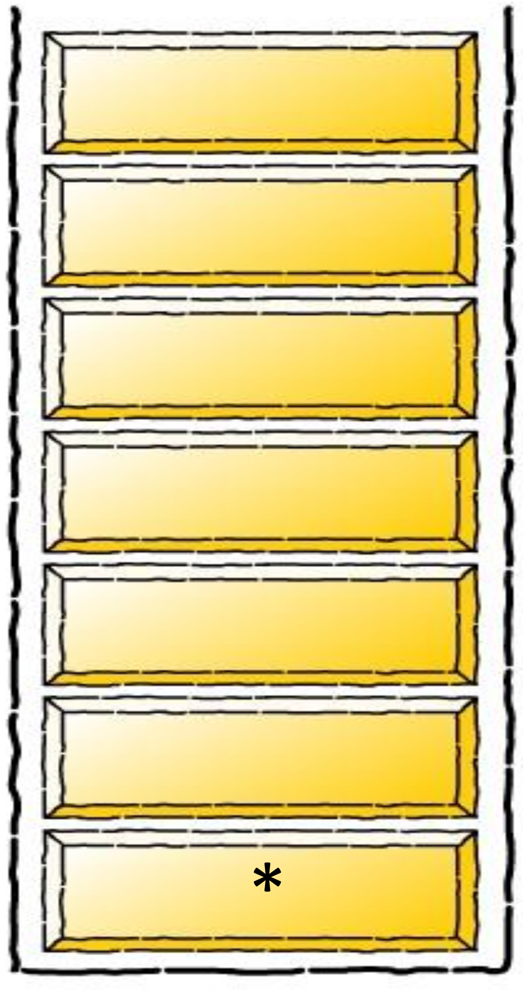
* d – (e + f)

postfixVect

a b + c -

Infix to postfix conversion

stackVect



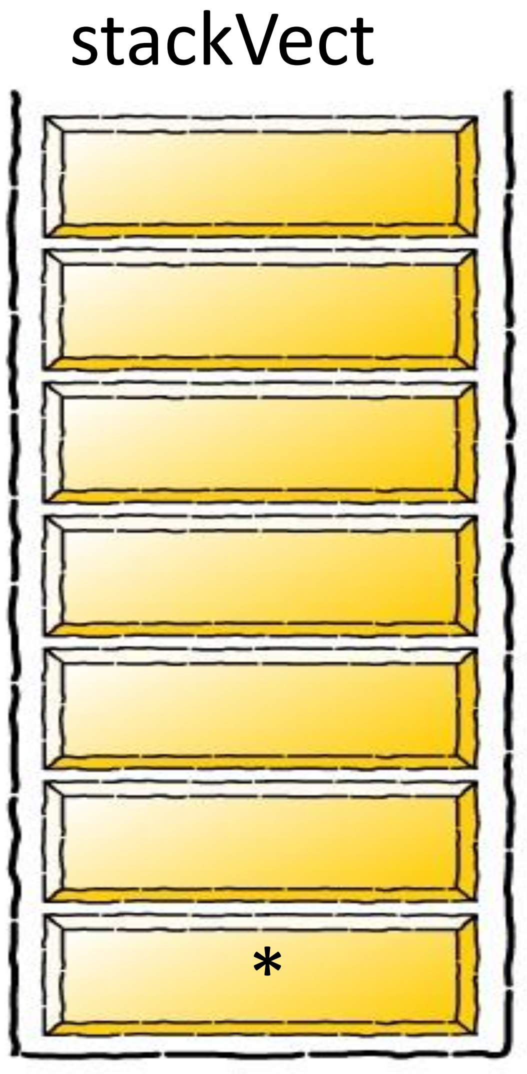
infixVect

d – (e + f)

postfixVect

a b + c -

Infix to postfix conversion



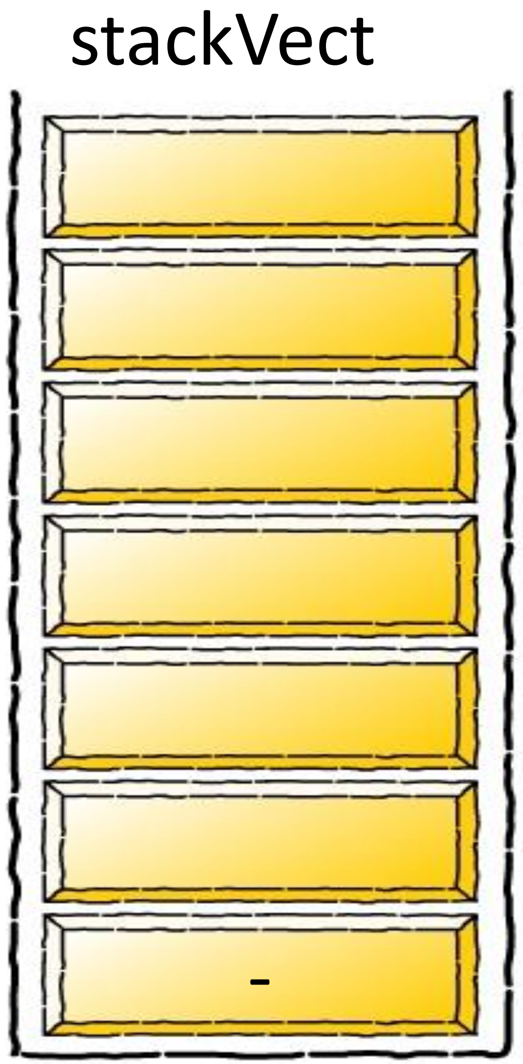
infixVect

– (e + f)

postfixVect

a b + c - d

Infix to postfix conversion



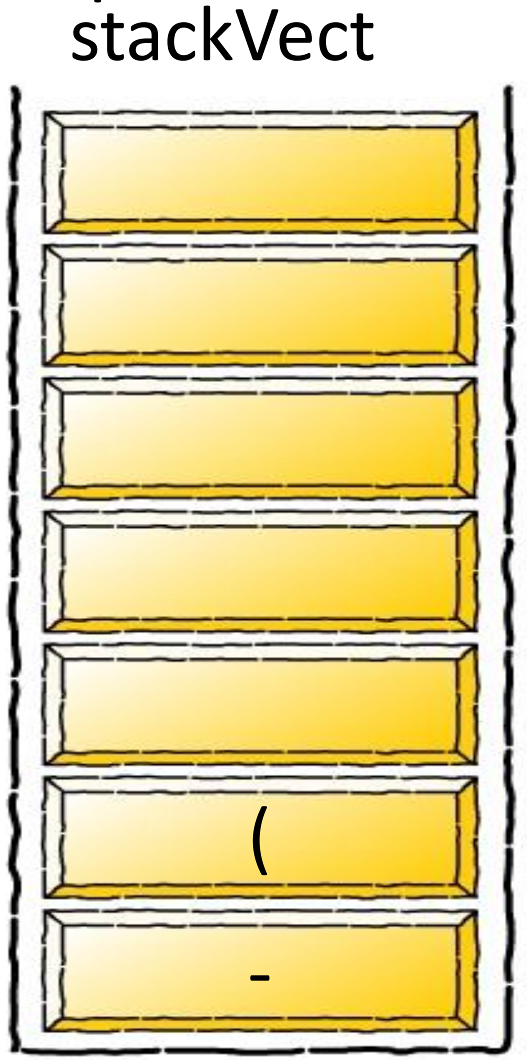
infixVect

(e + f)

postfixVect

a b + c – d *

Infix to postfix conversion



infixVect

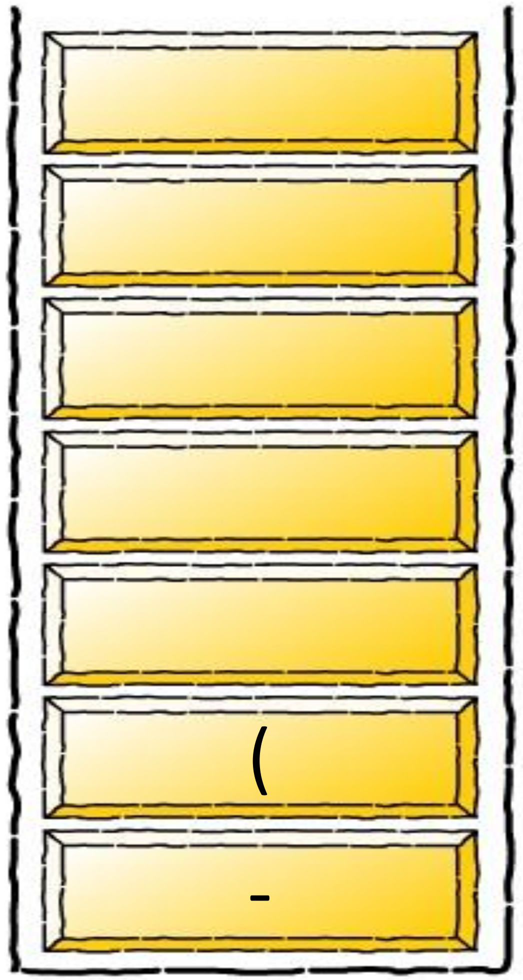
e + f)

postfixVect

a b + c – d *

Infix to postfix conversion

stackVect



infixVect

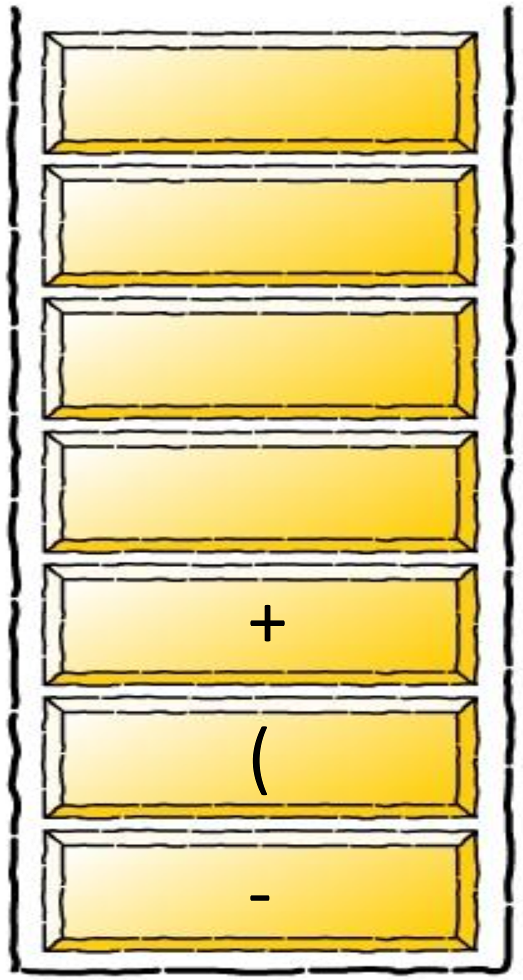
+ f)

postfixVect

a b + c – d * e

Infix to postfix conversion

stackVect



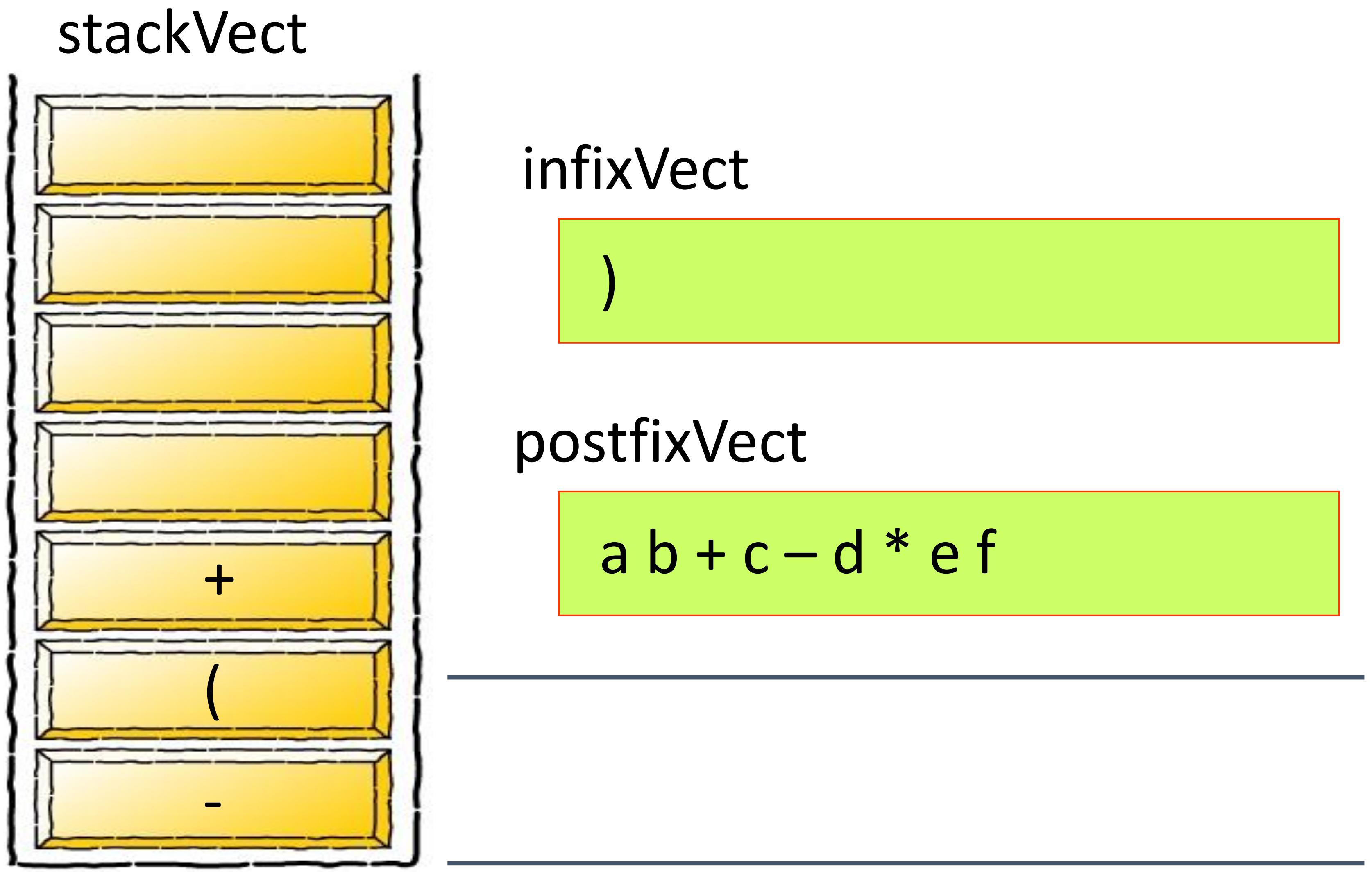
infixVect

f)

postfixVect

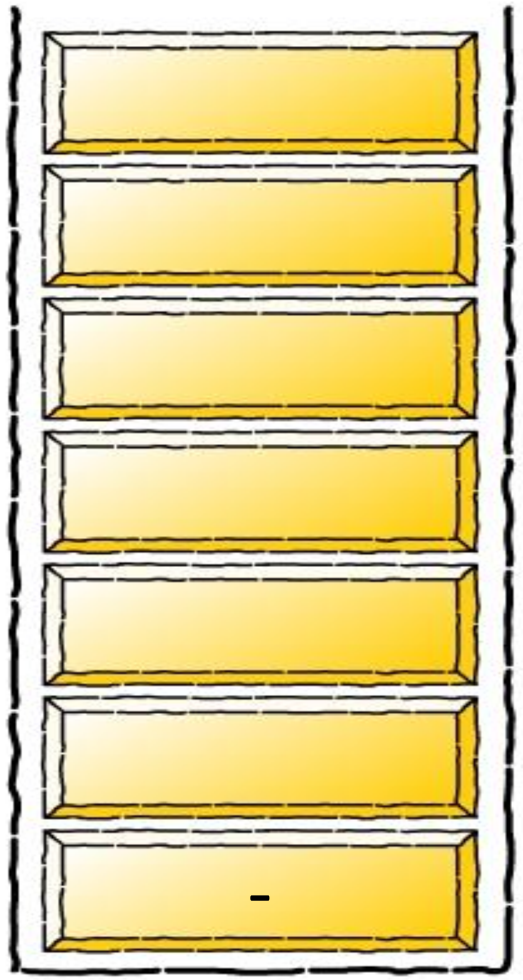
a b + c – d * e

Infix to postfix conversion



Infix to postfix conversion

stackVect



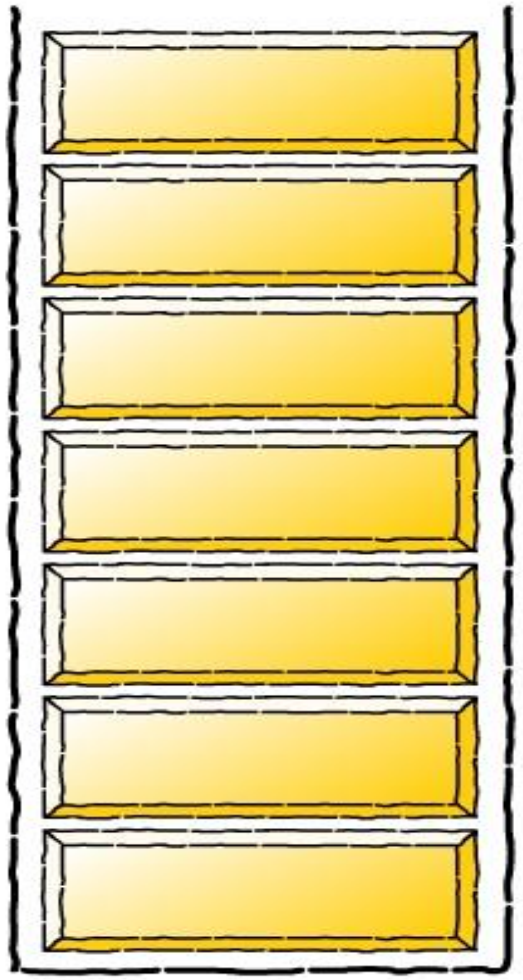
infixVect

postfixVect

a b + c – d * e f +

Infix to postfix conversion

stackVect



infixVect

postfixVect

a b + c – d * e f + -

C Program to Convert Infix to Postfix using Stack

```
#include<stdio.h>
#include<ctype.h>

char stack[100];
int top = -1;

void push(char x)
{
   stack[++top] = x;
}

char pop()
{
   if(top == -1)
      return -1;
   else
      return stack[top--];
}

int priority(char x)
{
   if(x == '(')
      return 0;
   if(x == '+' || x == '-')
      return 1;
   if(x == '*' || x == '/')
      return 2;
   return 0;
}
```

```
int main()
{
   char exp[100];
   char *e, x;
   printf("Enter the expression : ");
   scanf("%s",exp);
   printf("\n");
   e = exp;

   while(*e != '\0')
   {
      if(isalnum(*e))
         printf("%c ",*e);
      else if(*e == '(')
         push(*e);
      else if(*e == ')')
      {
         while((x = pop()) != '(')
            printf("%c ", x);
      }
      else
      {
         while(priority(stack[top]) >= priority(*e))
            printf("%c ",pop());
         push(*e);
      }
      e++;
   }

   while(top != -1)
   {
      printf("%c ",pop());
   }return 0;
}
```

Python to convert Infix to Postfix using stack

```python
# Class to convert the expression
class Conversion:

     # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []
          # Precedence setting
        self.output = []
        self.precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}

     # Check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False

     # Return the value of the top of the stack
    def peek(self):
        return self.array[-1]

     # Pop the element from the stack
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

     # Push the element to the stack
    def push(self, op):
        self.top += 1
        self.array.append(op)

     # A utility function to check is the given character
    # is operand
    def isOperand(self, ch):
        return ch.isalpha()

     # Check if the precedence of operator is strictly
    # less than top of stack or not
    def notGreater(self, i):
        try:
            a = self.precedence[i]
```

```python
# The main function that
    # converts given infix expression
    # to postfix expression
    def infixToPostfix(self, exp):

          # Iterate over the expression for conversion
        for i in exp:
             # If the character is an operand,
            # add it to output
            if self.isOperand(i):
                self.output.append(i)

             # If the character is an '(', push it to stack
            elif i == '(':
                self.push(i)

             # If the scanned character is an ')', pop and
            # output from the stack until and '(' is found
            elif i == ')':
                while( (not self.isEmpty()) and
                      self.peek() != '('):
                    a = self.pop()
                    self.output.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()

             # An operator is encountered
            else:
                while(not self.isEmpty() and self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)

          # Pop all the operator from the stack
        while not self.isEmpty():
            self.output.append(self.pop())

         for ch in self.output:
            print(ch, end="")

 # Driver code
if __name__ == '__main__':
```

Evaluate the postfix expression

$\updownarrow$

Evaluation of an Infix Expression

Evaluating Postfix Expression

1) Create a stack to store operands (or values).

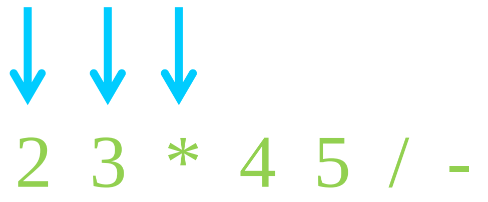
2) Scan the given expression and do following for every scanned element.
    a) If the element is an operand, **push** it into the stack
    b) If the element is an operator, **pop** operands for the operator from stack. Evaluate the operator and push the result back to the stack.

3) When the expression is ended, the number in the stack is the final answer

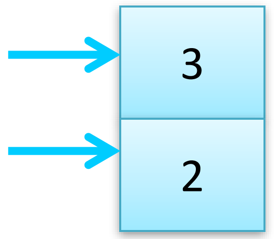
Evaluating Postfix Expression

Infix Expression: 2 * 3 – 4 / 5

Postfix Expression: 2 3 * 4 5 / -



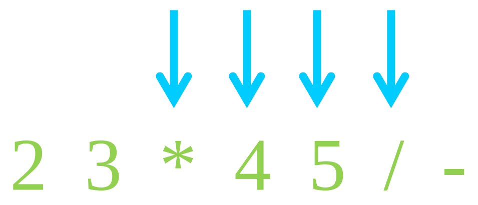
Evaluate Expression



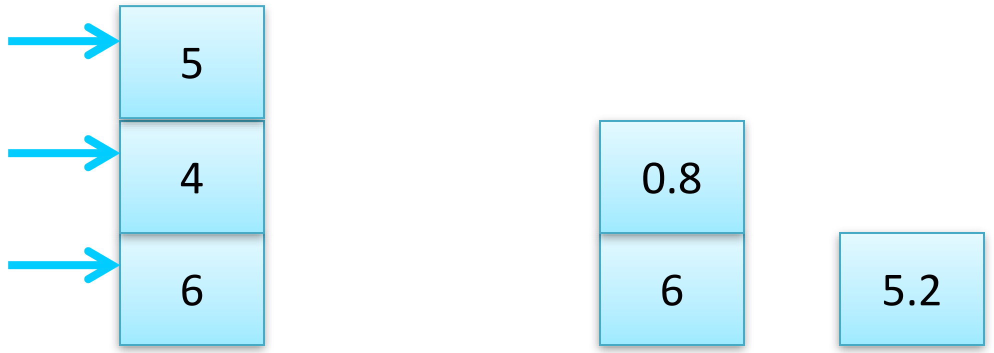
Evaluating Postfix Expression

Infix Expression: 2 * 3 – 4 / 5

Postfix Expression: 2 3 * 4 5 / -



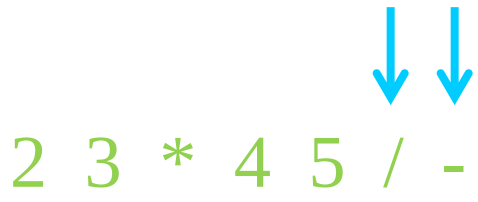
Evaluate Expression



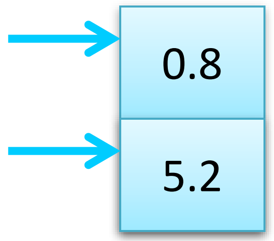
Evaluating Postfix Expression

Infix Expression: 2 * 3 – 4 / 5

Postfix Expression: 2 3 * 4 5 / -



Evaluate Expression



Evaluated Expression
(Stack top element) = 5.2

Evaluate the postfix expression

```
Algorithm to evaluate a postfix expression

Step 1: Add a ")" at the end of the postfix expression
Step 2: Scan every character of the postfix expression and
repeat
        steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered, push it on the stack
        IF an operator X is encountered, then
         a. pop the top two elements from the stack as A and B
           b. Evaluate B X A, where A was the topmost element
and B was
         the element below A.
           c. Push the result of evaluation on the stack
    [END OF IF]
Step 4: SET RESULT equal to the topmost element of the stack
Step 5: EXIT
```

Evaluation of an Infix Expression

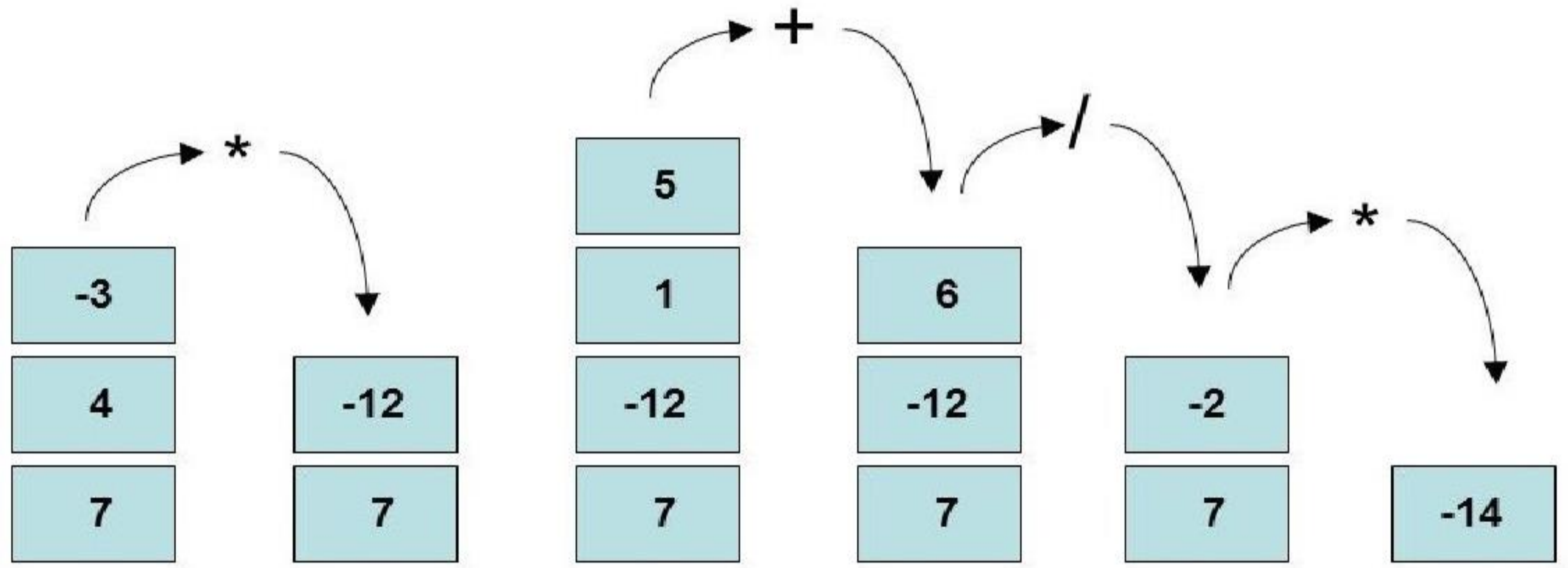
- Example: evaluate "9 - ((3 * 4) + 8) / 4".
- Step 1 infix "(9 - ((3 * 4) + 8) / 4)" => postfix "9 3 4 * 8 + 4 / -"
- Step 2 evaluate "9 3 4 * 8 + 4 / -"

infix	Stack	postfix
(	(	
9	(	9
–	(-	9
(	(-(	9
(	(-((	9
3	(-((	9 3
*	(-((*	9 3
4	(-((*	9 3 4
)	(-(	9 3 4 *
+	(-(+	9 3 4 *
8	(-(+	9 3 4 * 8
)	(-	9 3 4 * 8 +
/	(-/	9 3 4 * 8 +
4	(-/	9 3 4 * 8 + 4
)		9 3 4 * 8 + 4 / -

Character scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
–	4

Evaluating Postfix Expressions

- Expression = 7 4 -3 * 1 5 + / *



```python
# Python program to evaluate value of a postfix expression
# Class to convert the expression
class Evaluate:

    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []

    # check if the stack is empty
    def isEmpty(self):
        return True if self.top == -1 else False

    # Return the value of the top of the stack
    def peek(self):
        return self.array[-1]

    # Pop the element from the stack
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

    # Push the element to the stack
    def push(self, op):
        self.top += 1
        self.array.append(op)
```

```python
 # The main function that converts given infix expression
  # to postfix expression
  def evaluatePostfix(self, exp):

      # Iterate over the expression for conversion
      for i in exp:

          # If the scanned character is an operand
          # (number here) push it to the stack
          if i.isdigit():
              self.push(i)


          # If the scanned character is an operator,
          # pop two elements from stack and apply it.
          else:
              val1 = self.pop()
              val2 = self.pop()
              self.push(str(eval(val2 + i + val1)))


      return int(self.pop())

# Driver program to test above function
exp = "231*+9-"
obj = Evaluate(len(exp))
print ("postfix evaluation:
%d"%(obj.evaluatePostfix(exp)))
```

Complexity

- **Postfix -- Time Complexity:** O(N)
- Push/Pop/Peek – O(1)
- **ToH--Time Complexity**: $O(2^n)$

Recursion- Stack

Recursion

- ***A recursive function is a function that calls itself*** to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Every recursive solution has two major cases:
  - The ***base case*** in which the problem is simple enough to be solved directly without making any further calls to the same function.
  - ***Recursive case***, in which first the problem at hand is divided into simpler subparts. Second the function calls itself but with subparts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

```
int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; // composition
  else // base case
    fact = 1;

  return fact;
}
```

factorial Method

```
int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; // composition
  else // base case
    fact = 1;

  return fact;
}
```

```
# Python 3 program to find
# factorial of given number
def factorial(n):

    # Checking the number
    # is 1 or 0 then
    # return 1
    # other wise return
    # factorial
    if (n==1 or n==0):

        return 1

    else:

        return (n * factorial(n - 1))


# Driver Code
num = 5;
print("number : ",num)
print("Factorial : ",factorial(num))
```

```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

Example

```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

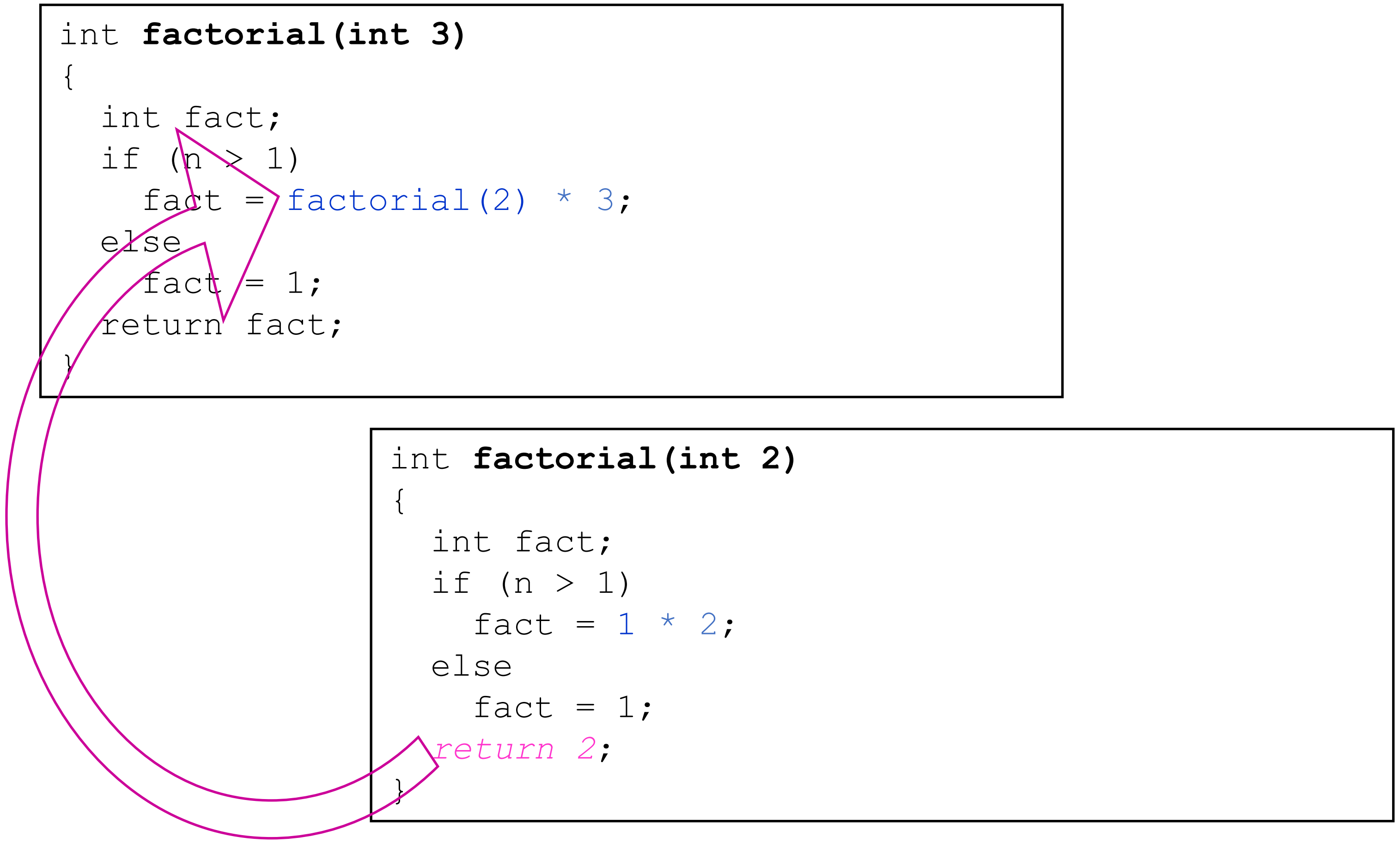
```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = factorial(1) * 2;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return 1;
}
```

```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

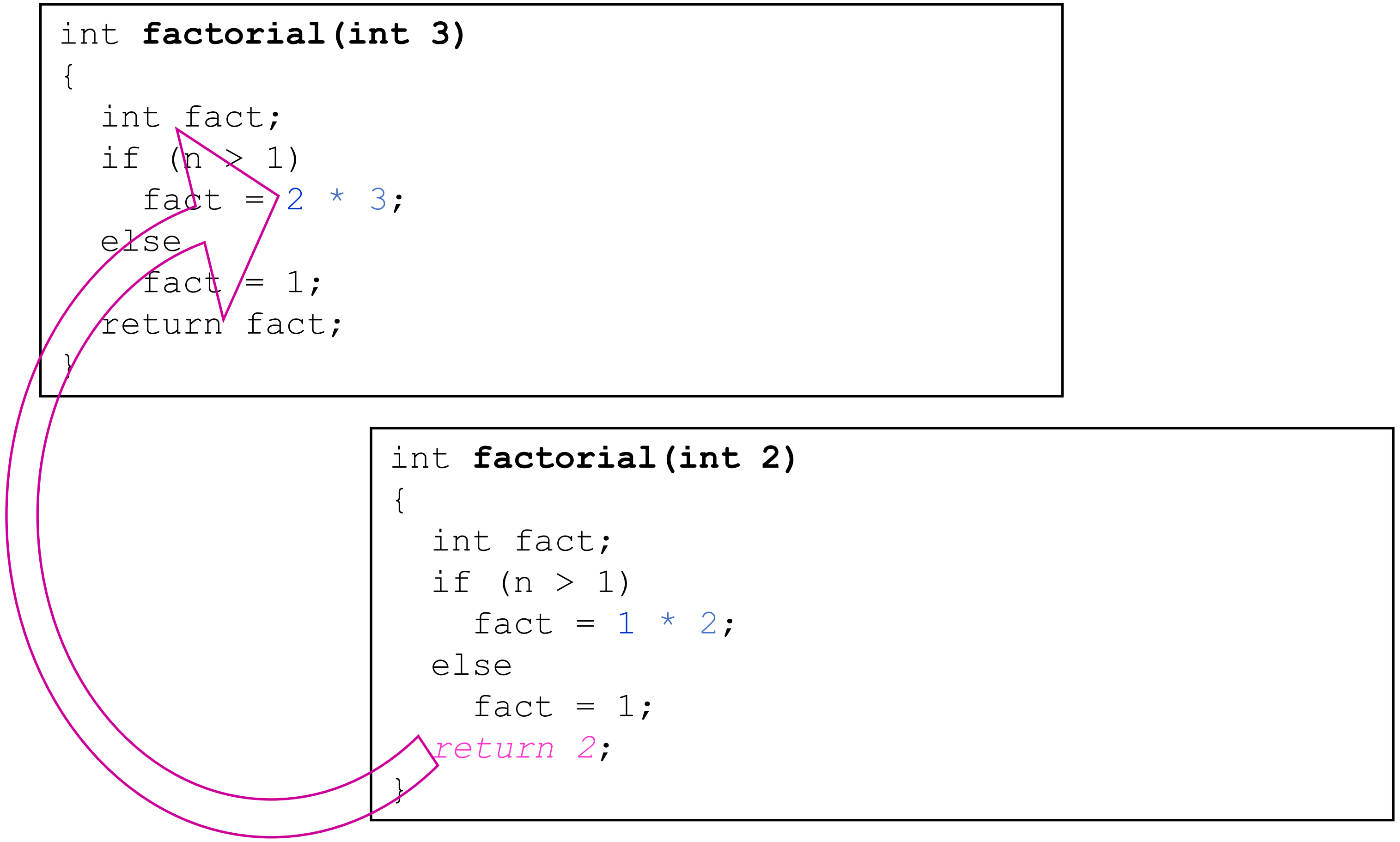
```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 1)
{
  int fact;
  if (n > 1)
    fact = factorial(n - 1) * n;
  else
    fact = 1;
  return 1;
}
```



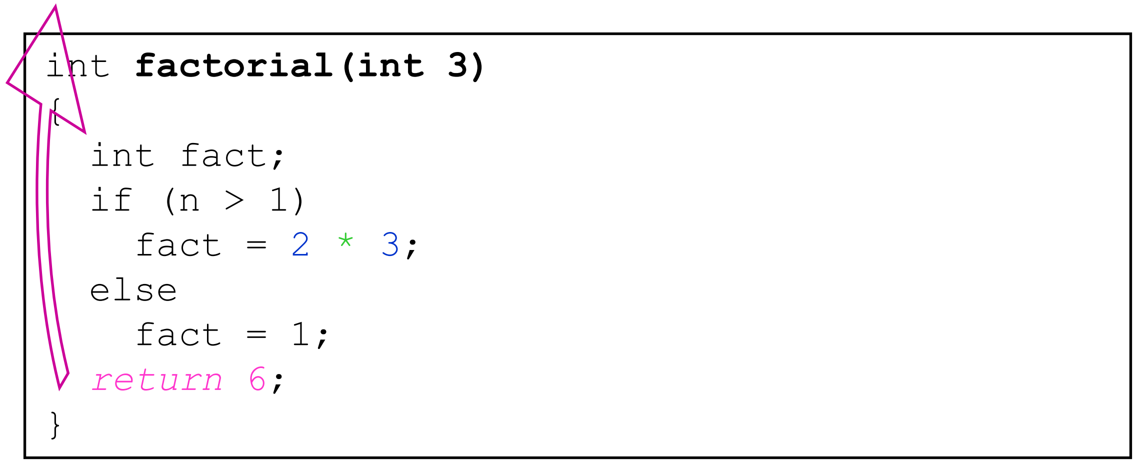
```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = factorial(2) * 3;
  else
    fact = 1;
  return fact;
}
```

```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return 2;
}
```



```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = 2 * 3;
  else
    fact = 1;
  return fact;
}
```

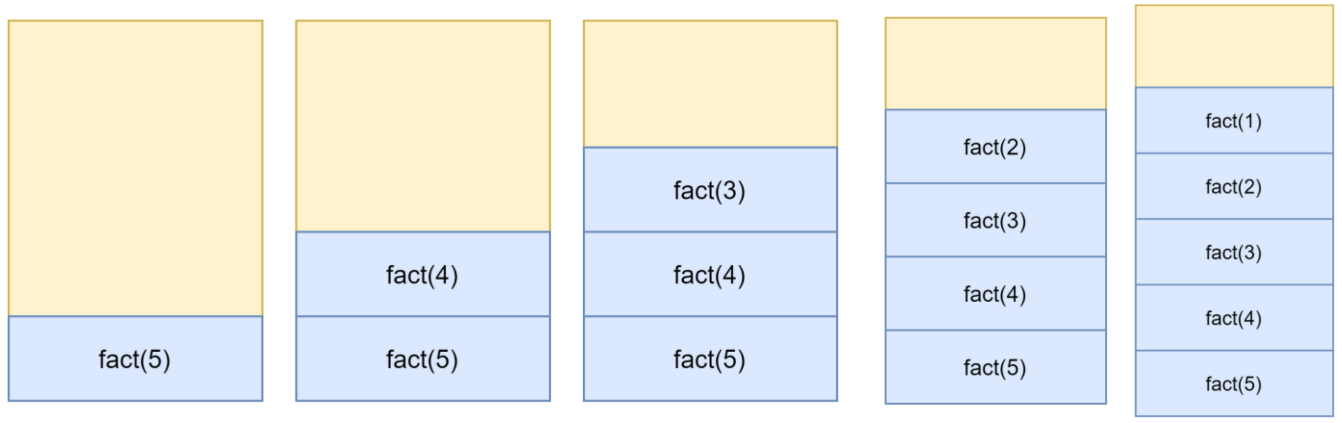
```
int factorial(int 2)
{
  int fact;
  if (n > 1)
    fact = 1 * 2;
  else
    fact = 1;
  return 2;
}
```



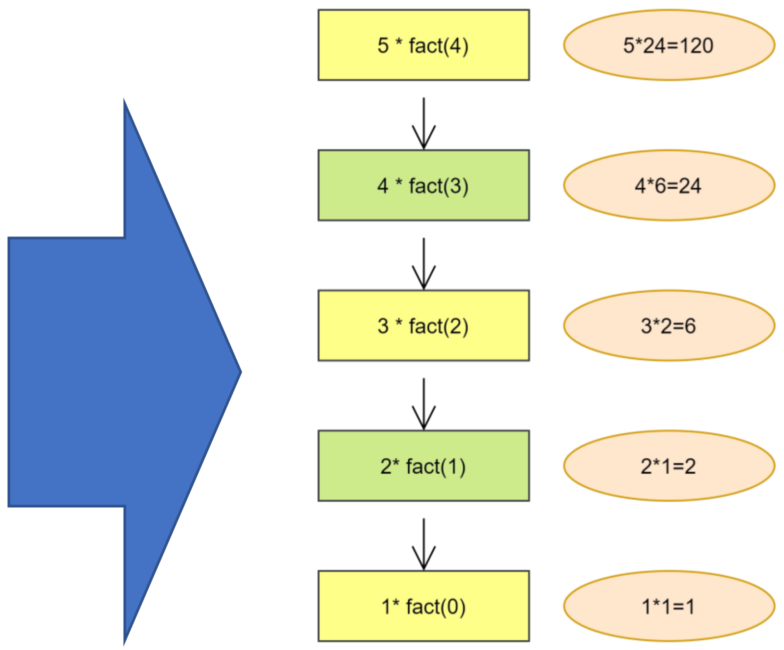
```
int factorial(int 3)
{
  int fact;
  if (n > 1)
    fact = 2 * 3;
  else
    fact = 1;
  return 6;
}
```

factorial(5)

In stack



Pictorial representation of the recursive function



Execution Trace (decomposition)

```
int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

$$\begin{tikzcd}
 & \texttt{factorial(4)} \arrow[dl] \arrow[dr] & \\
\texttt{factorial(3)} & & 4
\end{tikzcd}$$

Execution Trace (decomposition)

```
int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

$$\begin{tikzcd}
 & & \texttt{factorial(4)} \arrow[dl] \arrow[dr] & \\
 & \texttt{factorial(3)} \arrow[dl] \arrow[dr] & & 4 \\
\texttt{factorial(2)} & & 3 &
\end{tikzcd}$$

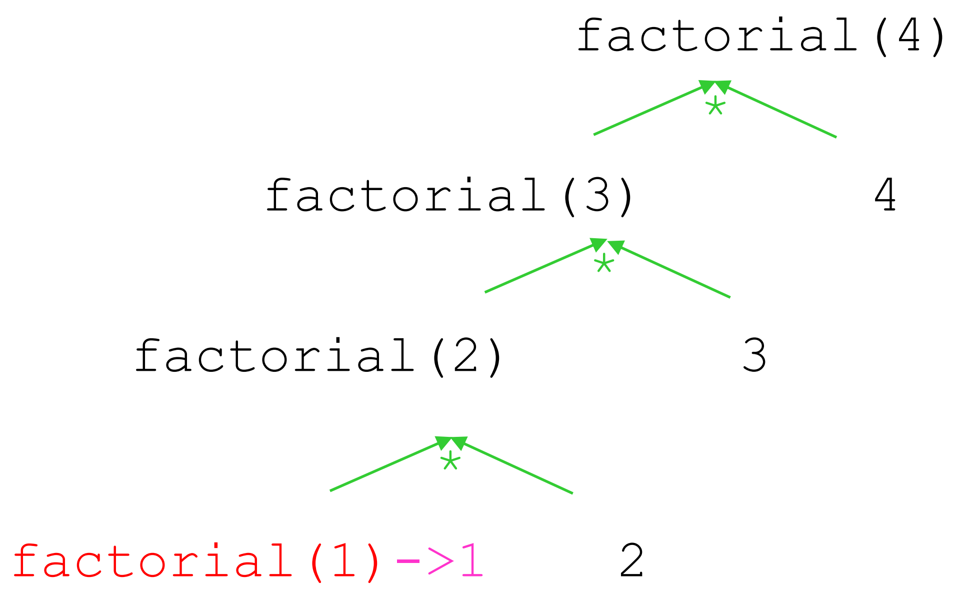
Execution Trace (decomposition)

```
int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

$$\begin{tikzcd}
 & & & \texttt{factorial(4)} \arrow[dl] \arrow[dr] & \\
 & & \texttt{factorial(3)} \arrow[dl] \arrow[dr] & & 4 \\
 & \texttt{factorial(2)} \arrow[dl] \arrow[dr] & & 3 & \\
\texttt{factorial(1)} & & 2 & &
\end{tikzcd}$$

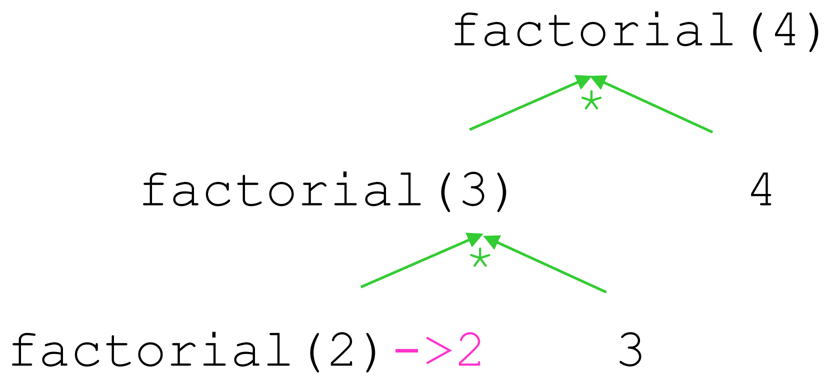
Execution Trace (composition)

```
int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



Execution Trace (composition)

```
int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```



Execution Trace (composition)

```
int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n – 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

$$\begin{tikzcd}
 & \texttt{factorial(4)} & \\
\texttt{factorial(3)->6} \arrow[ur, "*"] & & 4 \arrow[ul]
\end{tikzcd}$$

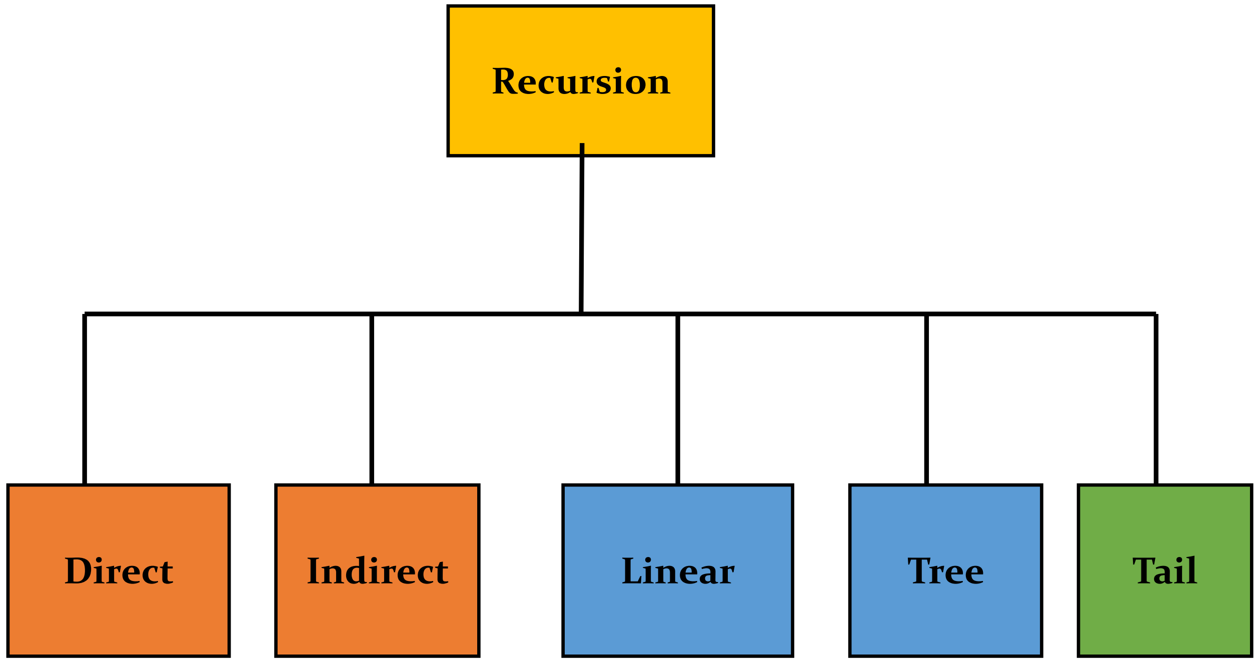
Execution Trace (composition)

```
int factorial(int n)
{
  int fact;
  if (n > 1) // recursive case (decomposition)
    fact = factorial(n - 1) * n; (composition)
  else // base case
    fact = 1;
  return fact;
}
```

```
factorial(4)->24
```

Types of Recursion

- Any recursive function can be characterized based on:
  - whether the function calls itself directly or indirectly (**direct or indirect recursion**).
  - whether any operation is pending at each recursive call (**tail-recursive or not**).
  - the structure of the calling pattern (**linear or tree-recursive**).



This following section should cover
by
Intro to programming

Direct Recursion

- A function is said to be *directly* recursive if it explicitly calls itself.
- For example, consider the function given below.

```
int Func( int n)
{
        if  ( n == 0 )
                retrun n;
        return Func(n-1);
}
```

```
def factorial(n):
    if n < 0 or n == 1:
    # The function terminates here
    return 1
   else:
     value = n*factorial(n-1)
   return value
```

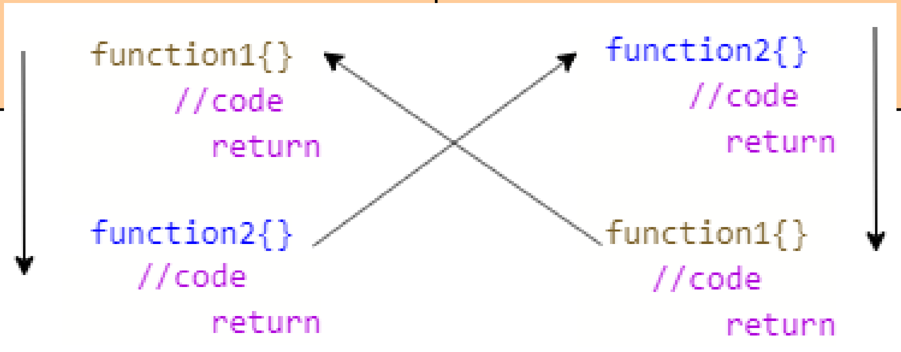
Indirect Recursion

- A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it.
- Look at the functions given below. These two functions are indirectly recursive as they both call each other.

```
int Func1(int n)
{
        if(n==0)
            return n;
        return Func2(n);
}
```

```
int Func2(int x)
{
      return Func1(x-1);
}
```

```
def odd(n):
 if n <= 10:
 return n+1
 n = n+1
 even(n)
 return

def even(n):
 if n <= 10:
 return n-1
 n = n+1
 odd(n)
 return
```



Linear Recursion

- Recursive functions can also be characterized depending on the way in which the recursion grows: in a linear fashion or forming a tree structure.
- In simple words, a recursive function **is said to be *linearly* recursive when no pending operation involves another recursive call to the function**.
- For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to *fact()* function.

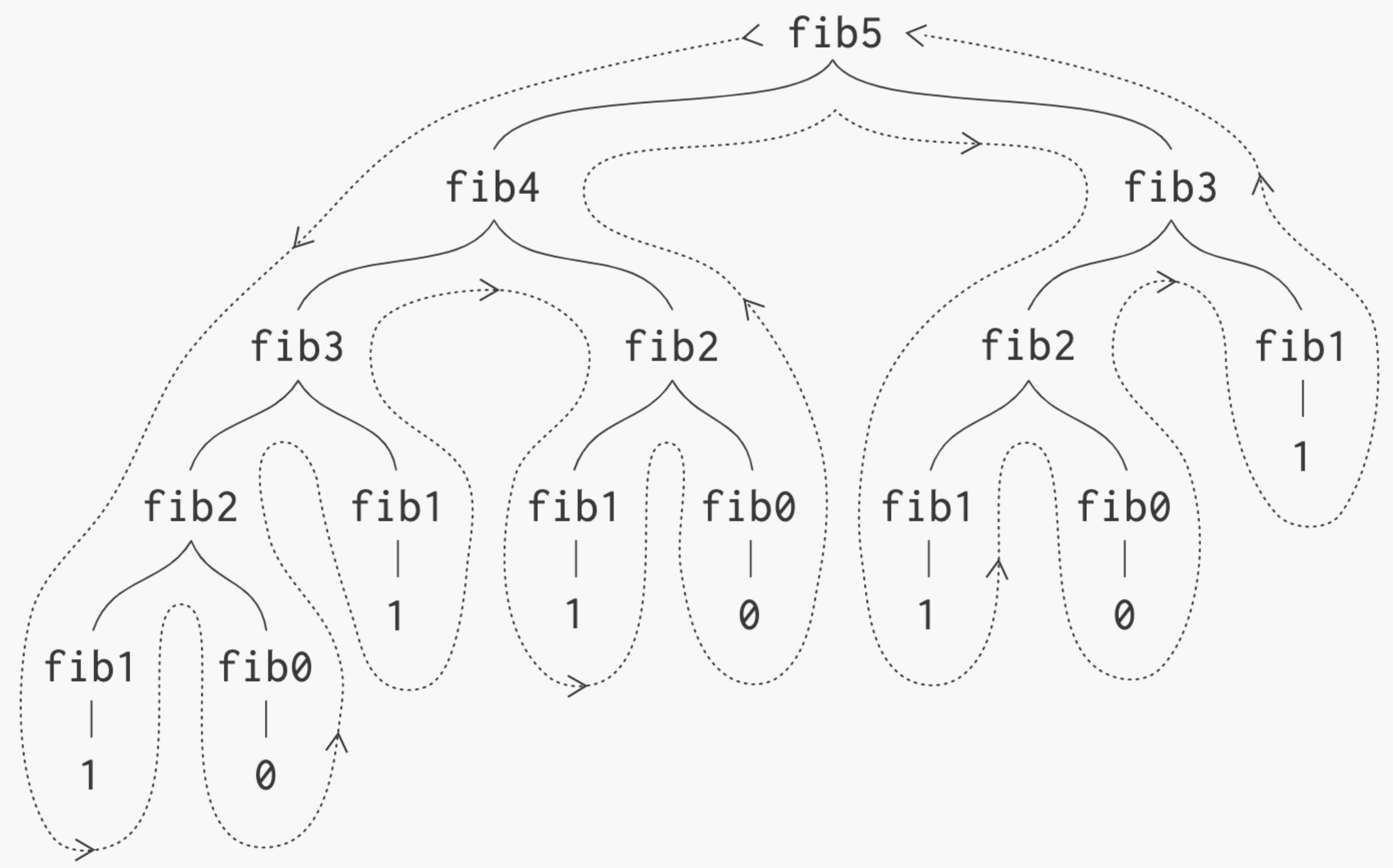
```
unsigned int factorial(unsigned int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

Tree Recursion

- A recursive function is said to be *tree* recursive (or *non-linearly* recursive) if the pending operation makes another recursive call to the function.
- For example, the Fibonacci function Fib in which the pending operations recursively calls the Fib function.

```
int Fibonacci(int num)
{
        if(num <= 2)
            return 1;
        return ( Fibonacci (num - 1) + Fibonacci(num – 2));
}
```

```
# Python program to display the Fibonacci
sequence

def recur_fibo(n):
   if n <= 1:
       return n
   else:
       return(recur_fibo(n-1) + recur_fibo(n-2))
```

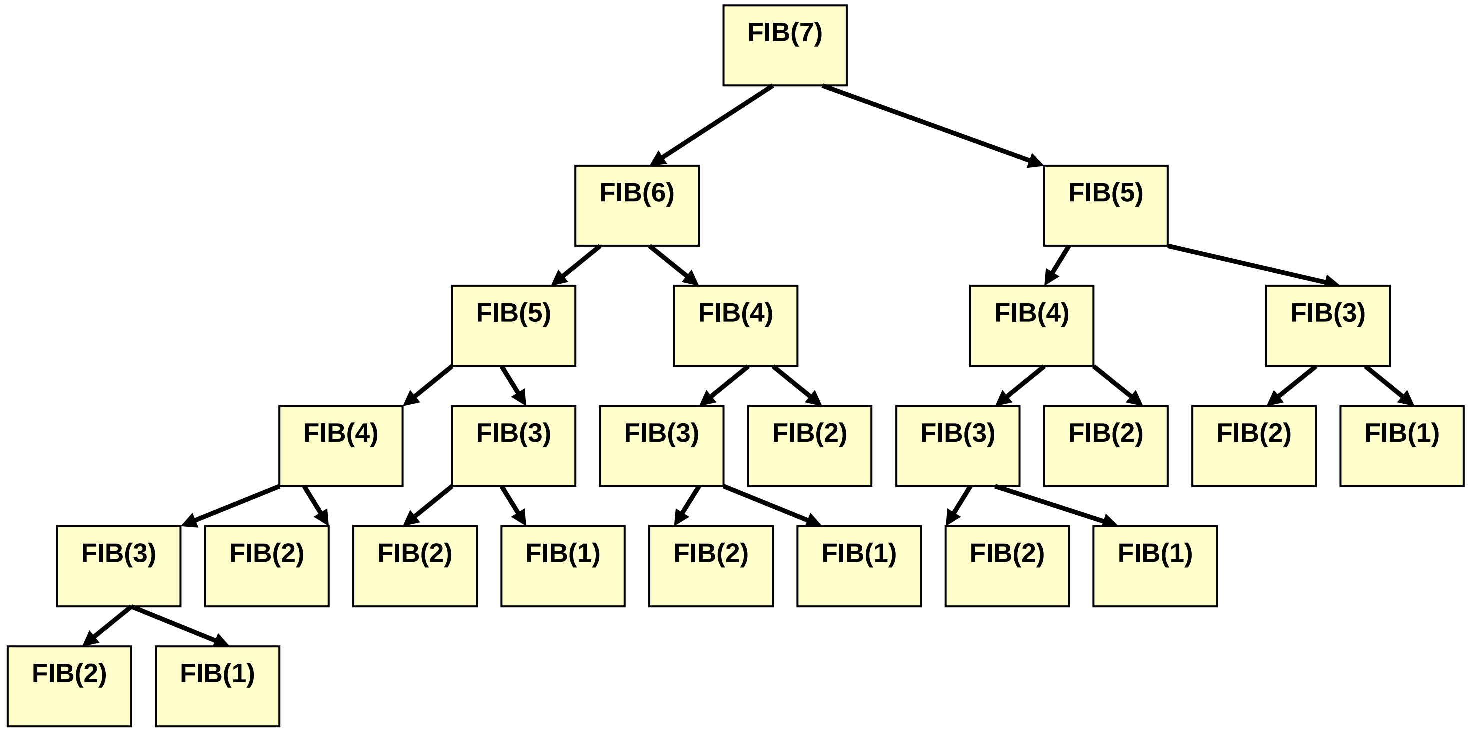

fib5
fib4
fib3
fib3
fib2
fib2
fib1
1
fib2
fib1
fib1
fib0
fib1
fib0
1
1
0
1
0
fib1
fib0
1
0

Fibonacci Series

The Fibonacci series can be given as:

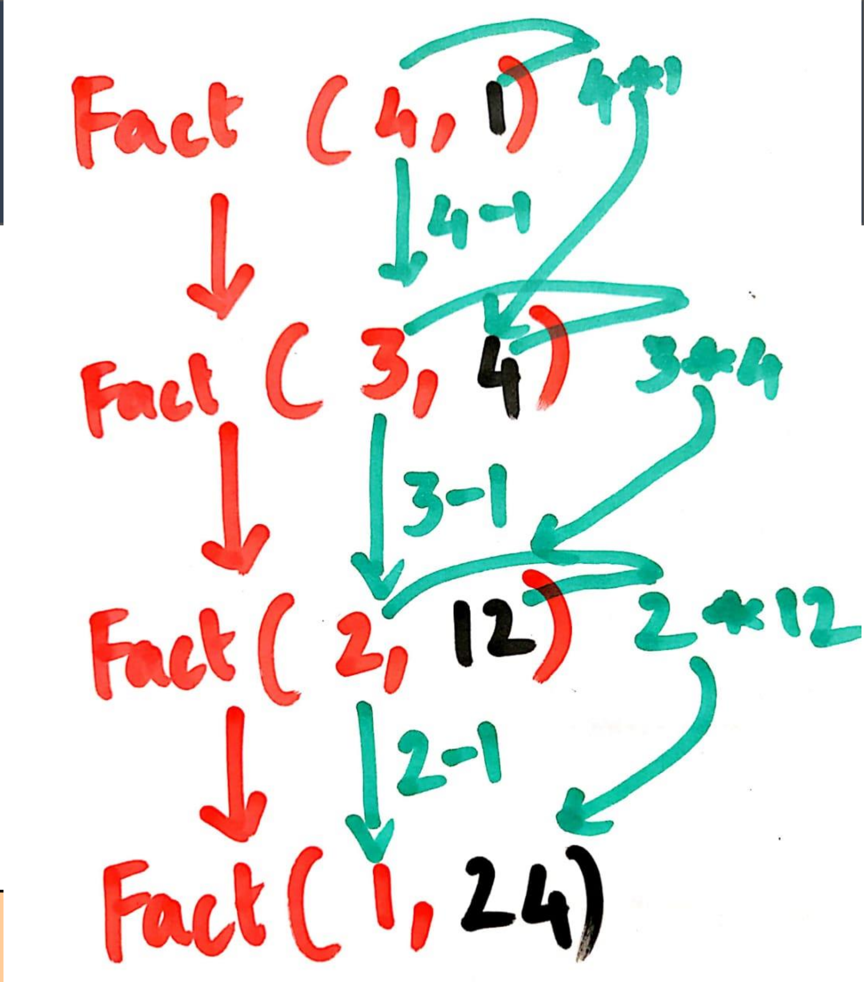
1 1 2 3 5 8 13 21 34 55……

```
int Fibonacci(int num) {
    if(num <= 2)  return 1;
    return ( Fibonacci (num - 1) + Fibonacci(num – 2));
}
```

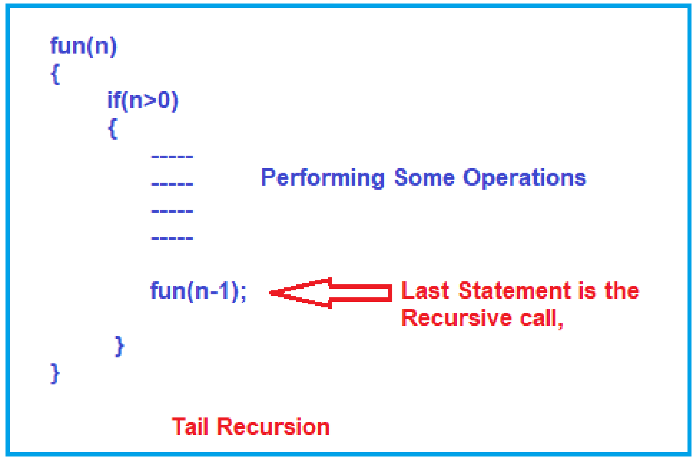


Tail Recursion

- A recursive function is said to be *tail recursive* ***<u>if no operations are pending to be performed when the recursive function returns to its caller</u>***.
- That is, when the called function returns, the returned value is immediately returned from the calling function.
- Tail recursive functions are highly desirable because they are much more efficient to use as in their case, the amount of information that has to be stored on the system stack is independent of the number of recursive calls.



`int Fact(n)`
`{`
`  return Fact1(n, 1);`
`}`
	`int Fact1(int n, int res)`
	`{`
	`      if (n==1)`
	`          return res;`
	`      return Fact1(n-1, n*res);`
	`}`



```
fun(n)
{
    if(n>0)
    {
        -----
        -----        Performing Some Operations
        -----
        -----

        fun(n-1);  <==  Last Statement is the
                        Recursive call,
    }
}
```

Tail Recursion

```python
# Code Showing Tail Recursion

# Recursion function
def fun(n):
    if (n > 0):
        print(n, end=" ")
        # Last statement in the function
        fun(n - 1)

# Driver Code
x = 3
fun(x)
```

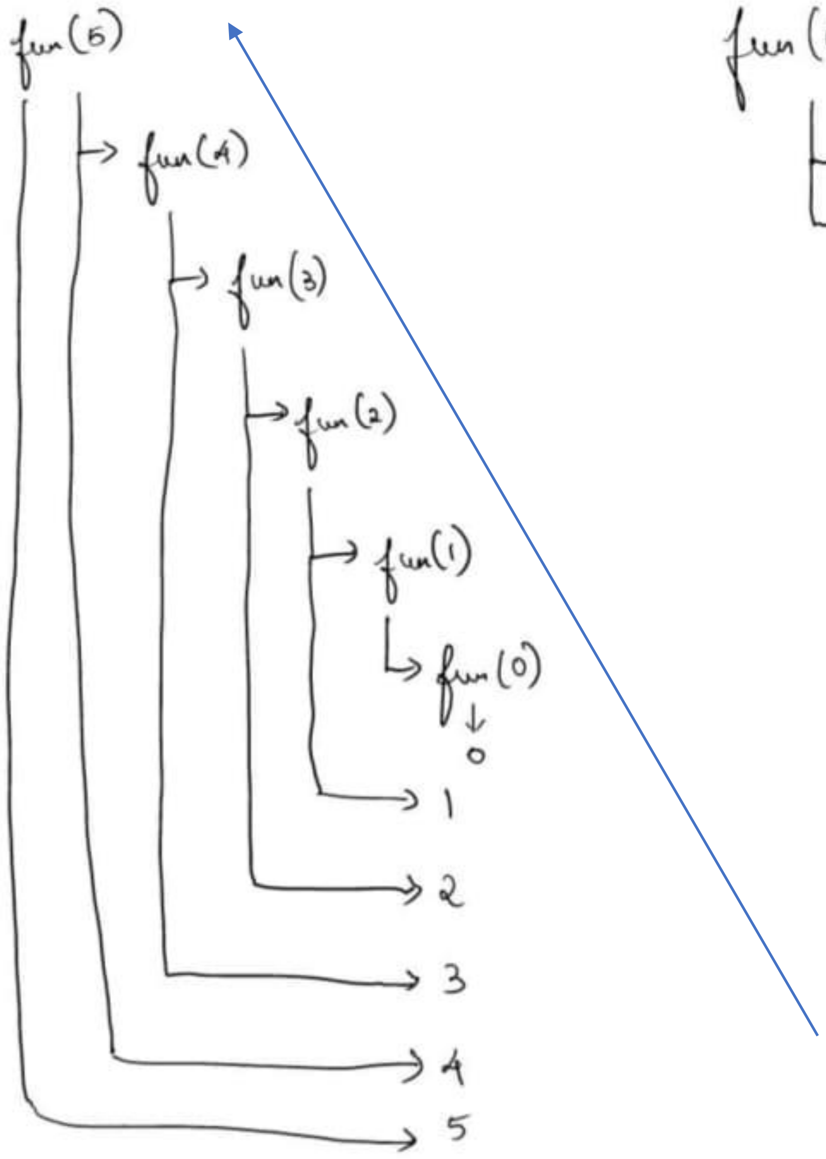
Time Complexity For Tail Recursion : O(n)
Space Complexity For Tail Recursion : O(n)

Now converting Tail Recursion into Loop and compare each other in terms of Time & Space Complexity

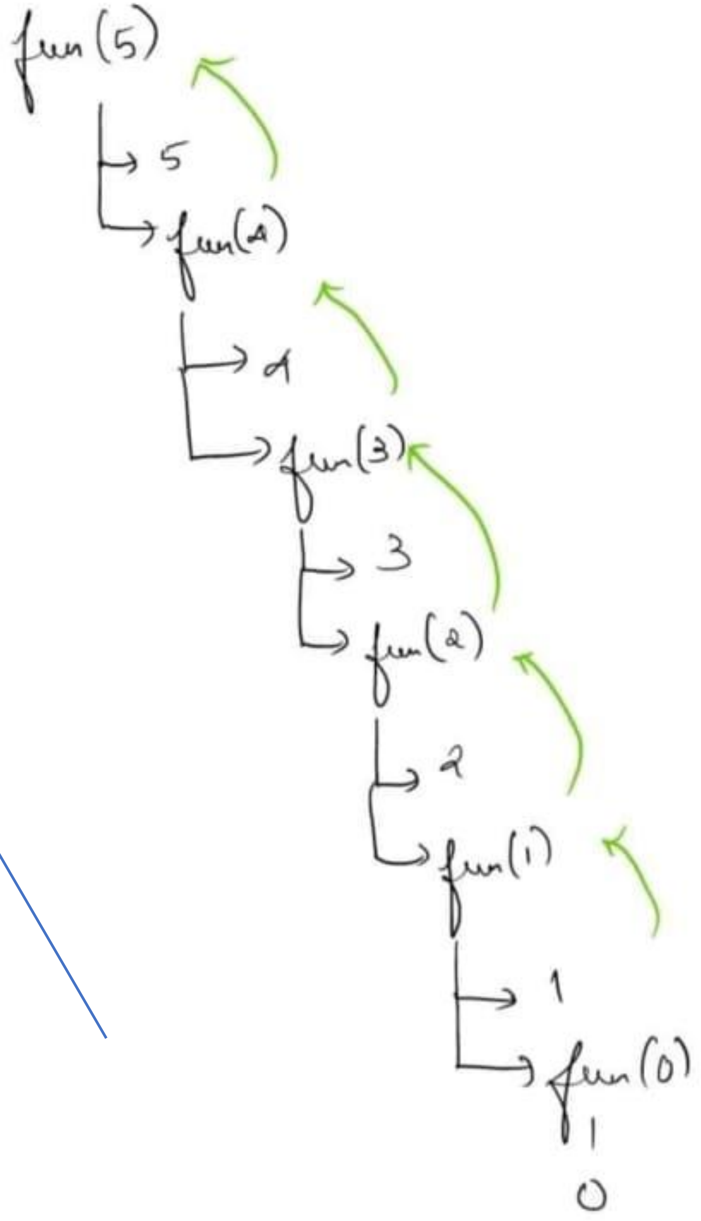
```python
# Converting Tail Recursion into Loop
def fun(y):

        while (y > 0):
                print(y , end = " ")
                y -= 1

# Driver code
x = 3
fun(x)
```

Time Complexity: O(n)
Space Complexity: O(1)

Head Recursion



Tail Recursion



```
void fun(int n) {
    if(n==0)
        return 0;

    fun(n-1);
    printf("%d", n); // Post recursive operation
}
```

```
void fun(int n) {
    if(n==0)
        return 0;

    printf("%d", n);
    fun(n-1);
}
```

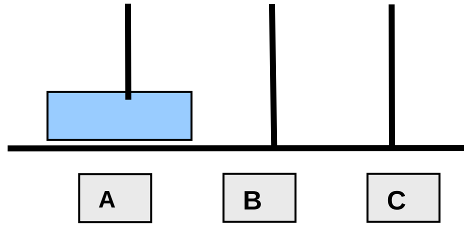
Tower of Hanoi

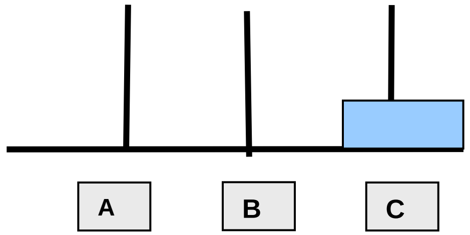
Problem statements

- Tower of Hanoi is a mathematical puzzle where **we have three rods and n disks**. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
  - 1) Only one disk can be moved at a time.
  - 2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
  - 3) **No disk may be placed on top of a smaller disk**.

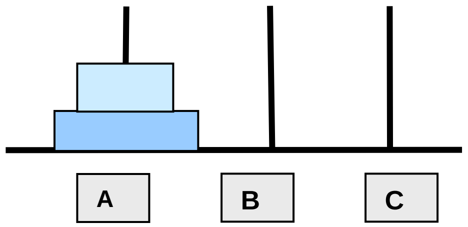
Tower of Hanoi

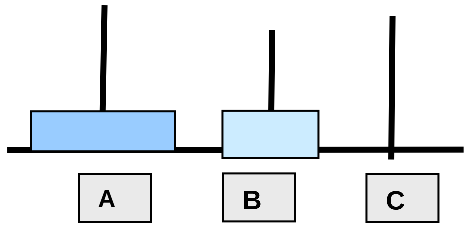
Tower of Hanoi is one of the main applications of a recursion. It says, "if you can solve *n-1* cases, then you can easily solve the *nth* case"

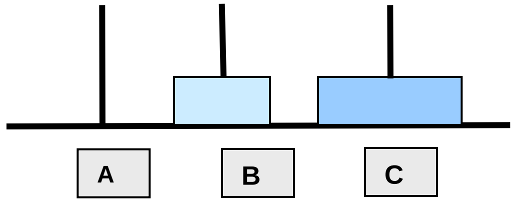


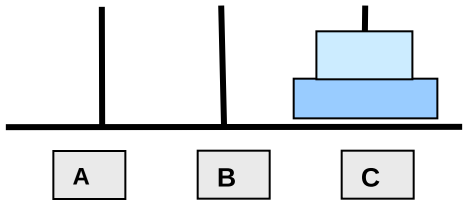


If there is only one ring, then simply move the ring from source to the destination





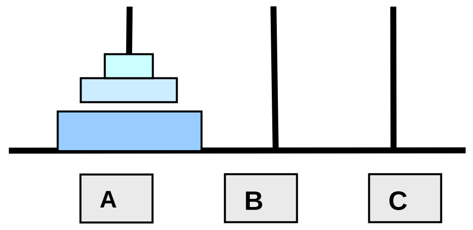


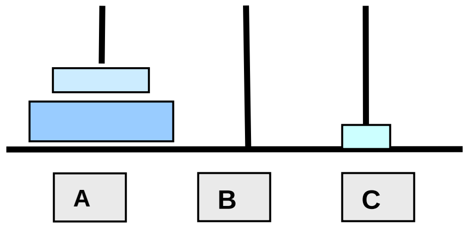


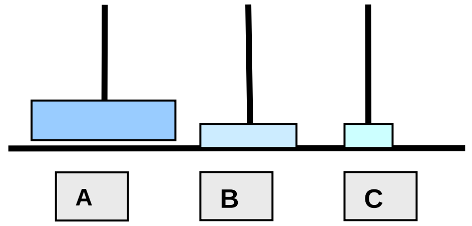
If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from the source to the destination

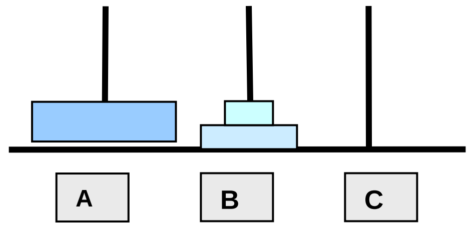
Tower of Hanoi

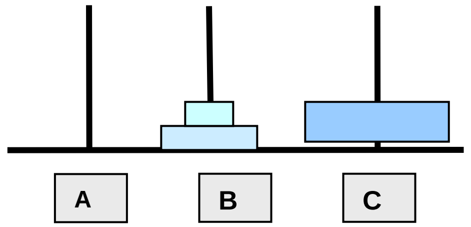
Consider the working with three rings.

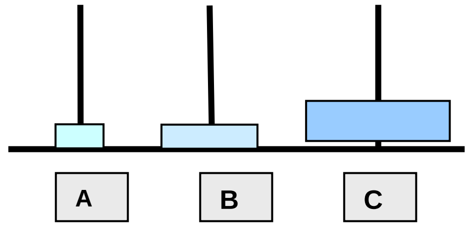


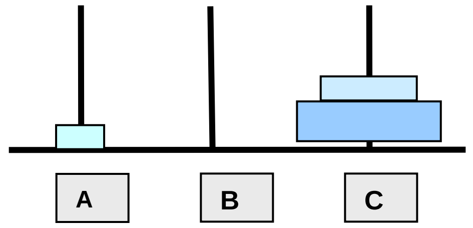


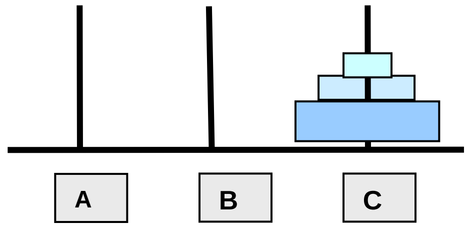








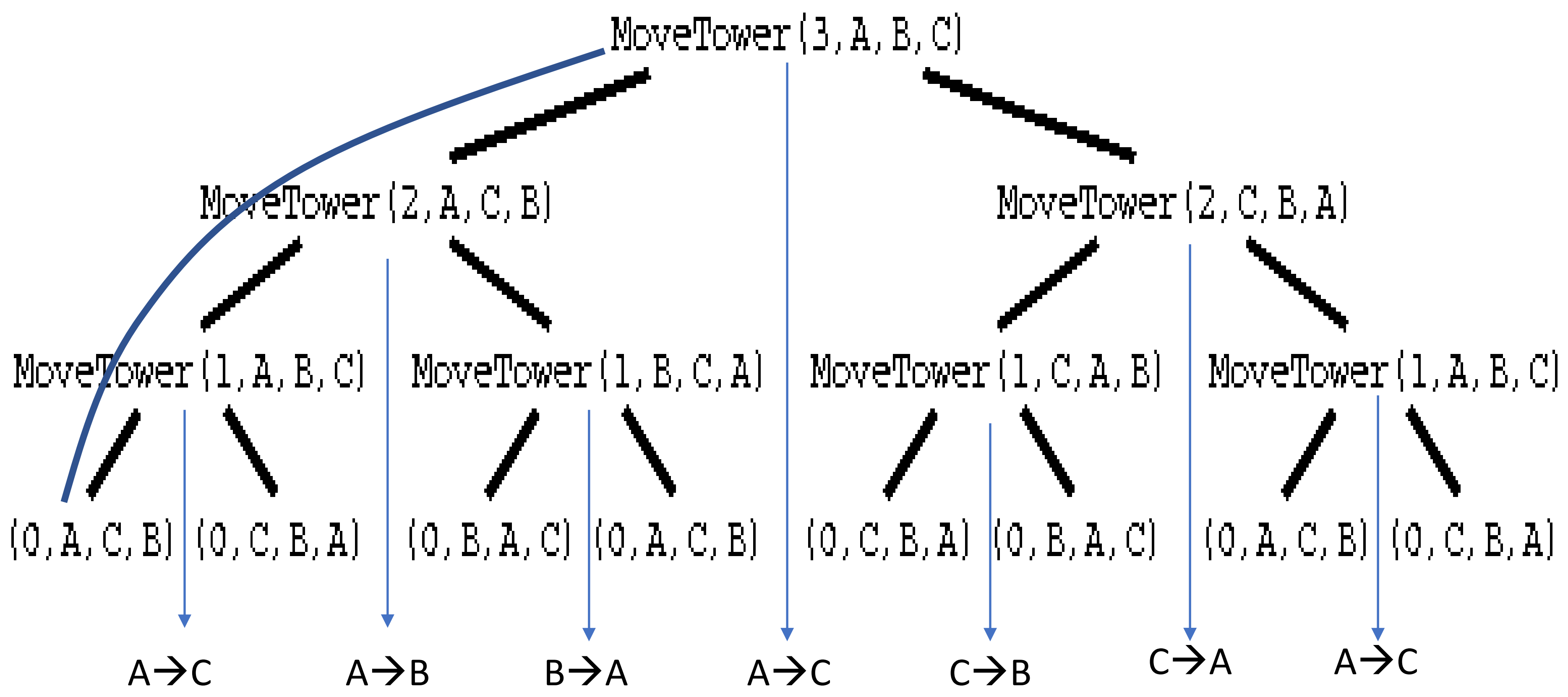




A→ C
A→ B
C→B
A→C
B→C
B→C
A→C

ToH using recursive function

```
#include <stdio.h>
// C recursive function to solve tower of hanoi puzzle

void towerOfHanoi(int n, char from_rod, char aux_rod, char to_rod)
{
        if (n == 1)
        {
                printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
                return;
        }
        towerOfHanoi(n-1, from_rod, to_rod, aux_rod);
        printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
        towerOfHanoi(n-1, aux_rod, from_rod, to_rod);
}

int main()
{
        int n = 3; // Number of disks
        towerOfHanoi(n, 'A', ‘B', ‘C'); // A, B and C are names of rods
        return 0;
}
```



A→ C
A→ B
C→B
A→C
B→C
B→C
A→C

ToH using recursive function

```
# Recursive Python function to solve the tower of hanoi

def TowerOfHanoi(n , source, destination, auxiliary):
        if n==1:
                print ("Move disk 1 from source",source,"to destination",destination)
                return
        TowerOfHanoi(n-1, source, auxiliary, destination)
        print ("Move disk", n, "from source",source,"to destination",destination)
        TowerOfHanoi(n-1, auxiliary, destination, source)

# Driver code
n = 4
TowerOfHanoi(n,'A','B','C')
# A, C, B are the name of rods
```

ToH using recursive function (Alternate solun)

```
#include <stdio.h>
// C recursive function to solve tower of hanoi puzzle

void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
	if (n == 1)
	{
		printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
		return;
	}
	towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
	printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
	towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
	int n = 4; // Number of disks
	towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
	return 0;
}
```

Using Iteration

```
# Python3 program for iterative Tower of Hanoi
importsys


# A structure to represent a stack
classStack:
    # Constructor to set the data of
    # the newly created tree node
    def__init__(self, capacity):
        self.capacity =capacity
        self.top =-1
        self.array =[0]*capacity


# function to create a stack of given capacity.
defcreateStack(capacity):
    stack =Stack(capacity)
    returnstack

# Stack is full when top is equal to the last index
defisFull(stack):
    return(stack.top ==(stack.capacity -1))

# Stack is empty when top is equal to -1
defisEmpty(stack):
    return(stack.top ==-1)

# Function to add an item to stack.
# It increases top by 1
defpush(stack, item):
    if(isFull(stack)):
        return
    stack.top+=1
    stack.array[stack.top] =item

# Function to remove an item from stack.
# It decreases top by 1
defPop(stack):
    if(isEmpty(stack)):
        return-sys.maxsize
    Top =stack.top
    stack.top-=1
    returnstack.array[Top]

# Function to implement legal
# movement between two poles
defmoveDisksBetweenTwoPoles(src, dest, s, d):
    pole1TopDisk =Pop(src)
    pole2TopDisk =Pop(dest)


    # When pole 1 is empty
    if(pole1TopDisk ==-sys.maxsize):
        push(src, pole2TopDisk)
        moveDisk(d, s, pole2TopDisk)

    # When pole2 pole is empty
    elseif(pole2TopDisk ==-sys.maxsize):
        push(dest, pole1TopDisk)
        moveDisk(s, d, pole1TopDisk)

    # When top disk of pole1 > top disk of pole2
    elseif(pole1TopDisk > pole2TopDisk):
        push(src, pole1TopDisk)
        push(src, pole2TopDisk)
        moveDisk(d, s, pole2TopDisk)

    # When top disk of pole1 < top disk of pole2
    else:
        push(dest, pole2TopDisk)
        push(dest, pole1TopDisk)
        moveDisk(s, d, pole1TopDisk)

# Function to show the movement of disks
defmoveDisk(fromPeg, toPeg, disk):
    print("Move the disk", disk, "from '", fromPeg, "' to '", toPeg, "'")

# Function to implement TOH puzzle
deftohIterative(num_of_disks, src, aux, dest):
    s, d, a ='S', 'D', 'A'

    # If number of disks is even, then interchange
    # destination pole and auxiliary pole
    if(num_of_disks %2==0):
        temp =d
        d =a
        a =temp
    total_num_of_moves =int(pow(2, num_of_disks) -1)

    # Larger disks will be pushed first
    fori inrange(num_of_disks, 0, -1):
        push(src, i)

    fori inrange(1, total_num_of_moves +1):
        if(i %3==1):
            moveDisksBetweenTwoPoles(src, dest, s, d)

        elseif(i %3==2):
            moveDisksBetweenTwoPoles(src, aux, s, a)

        elseif(i %3==0):
            moveDisksBetweenTwoPoles(aux, dest, a, d)


# Input: number of disks
num_of_disks =3


# Create three stacks of size 'num_of_disks'
# to hold the disks
src =createStack(num_of_disks)
dest =createStack(num_of_disks)
aux =createStack(num_of_disks)


tohIterative(num_of_disks, src, aux, dest)


# This code is contributed by divyeshrabadiya07.
```

Complexity

$$T(n) = 2T(n-1) + 1$$

$$T(n) = 2^k T(n-k) + 2^{(k-1)} + 2^{(k-2)} + \ldots\ldots\ldots\ldots + 2^2 + 2^1 + 1$$

Base condition T(1) =1
n – k = 1
k = n-1
put, k = n-1

$$T(n) = 2^{(n-1)} T(1) + +2^{(n-2)} + \ldots\ldots\ldots\ldots + 2^2 + 2^1 + 1$$

It is a GP series, and the sum is : $2^n - 1 = O(2^n)$

Pros and Cons of Recursion

Pros

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

Cons

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive process in midstream is slow.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counterpart.
- It is difficult to find bugs, particularly when using global variables.

Any question ?