Stacks

Introduction

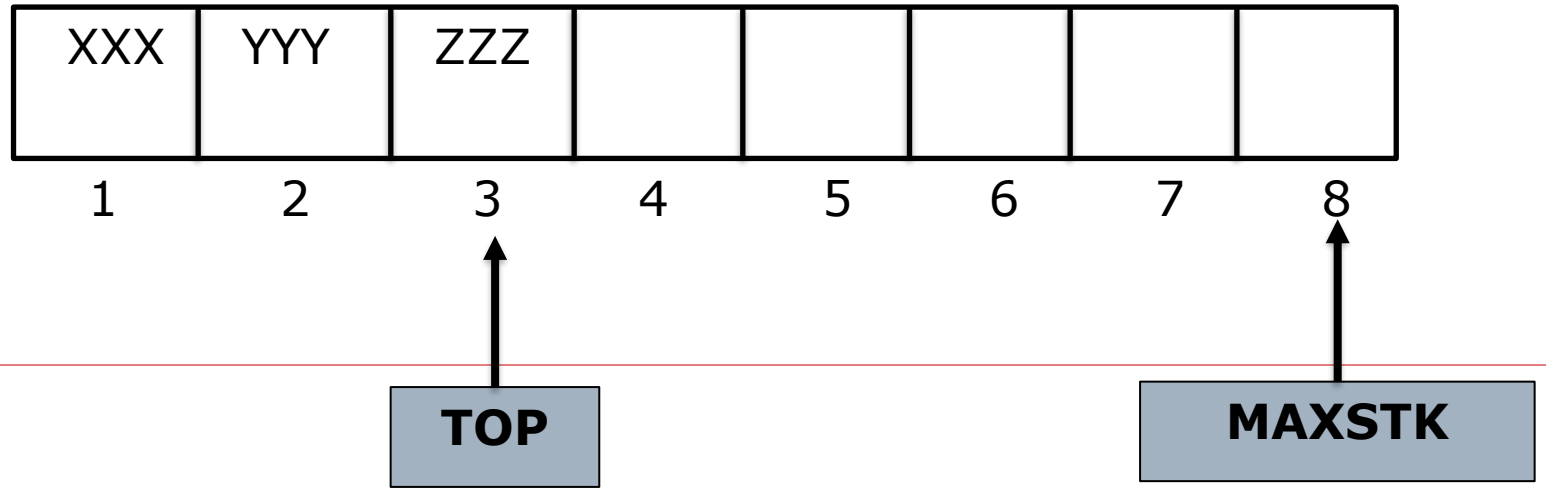
- A stack is a list of elements in which an element may be inserted or deleted only at one end called the top of the stack also called LIFO lists.
- Basic operations performed on stack

1. “Push” is the term used to insert element into a stack.
2. “Pop” is the term used to delete an element from a stack.

NOTE: Stacks are frequently used to indicate the order of processing of data when certain steps of the processing must be postponed until other conditions are fulfilled.

Array Representation of Stacks

- Each of our stack will be maintained by a linear array STACK.
- A pointer variable TOP which contains the location of the top element.
- A variable MAXSTK which gives the maximum no. of elements that can be held by the stack.
- The condition TOP=0 or TOP=NULL indicates that stack is empty.



Stack

Procedure 1: PUSH(STACK, TOP, MAXSTK, ITEM)

This procedure pushes an ITEM onto a stack.

Step 1: [Stack already filled? OVERFLOW?]

if TOP=MAXSTK, then

Print “OVERFLOW”

Return

Step 2: Set TOP:=TOP+1

Step 3: [Inserts ITEM in new TOP position]

STACK[TOP]:=ITEM

Step 4: Return

Stack

Procedure 2: POP(STACK, TOP, ITEM)

This procedure deletes the top element of STACK and assigns it to the variable ITEM

Step 1: [Stack has an item to delete? UNDERFLOW?]

if TOP=0, then

Print “UNDERFLOW”

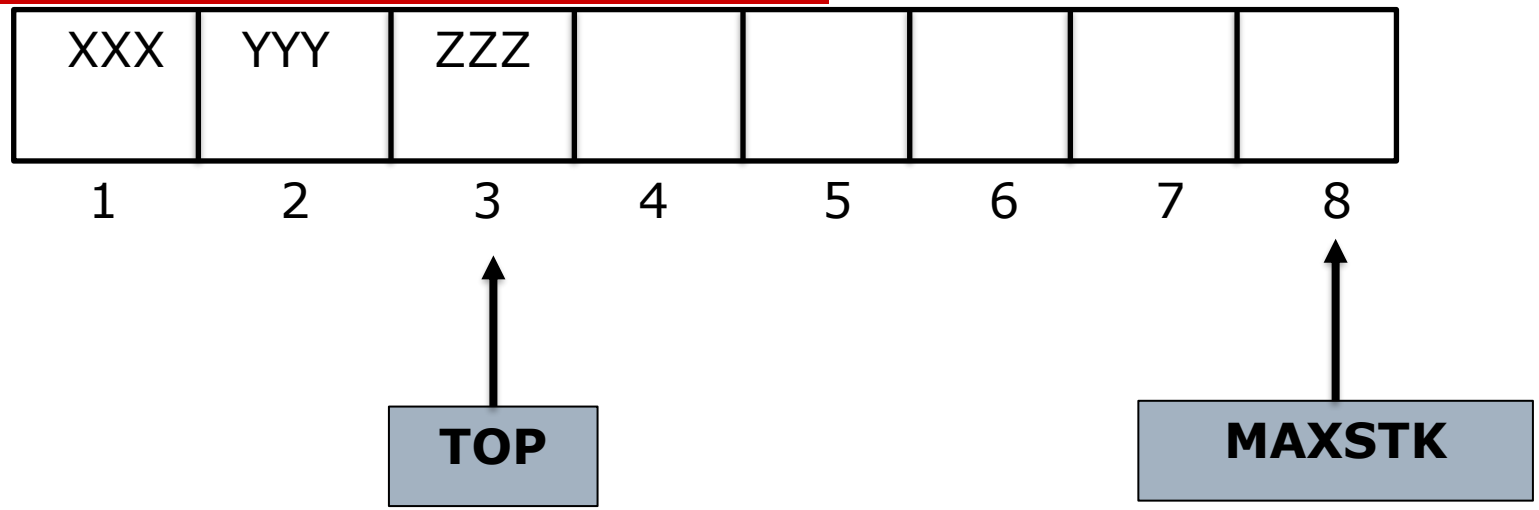
Return

Step 2: Set ITEM:=STACK[TOP]

Step 3: Set TOP:=TOP-1

Step 4: Return

Stack Example

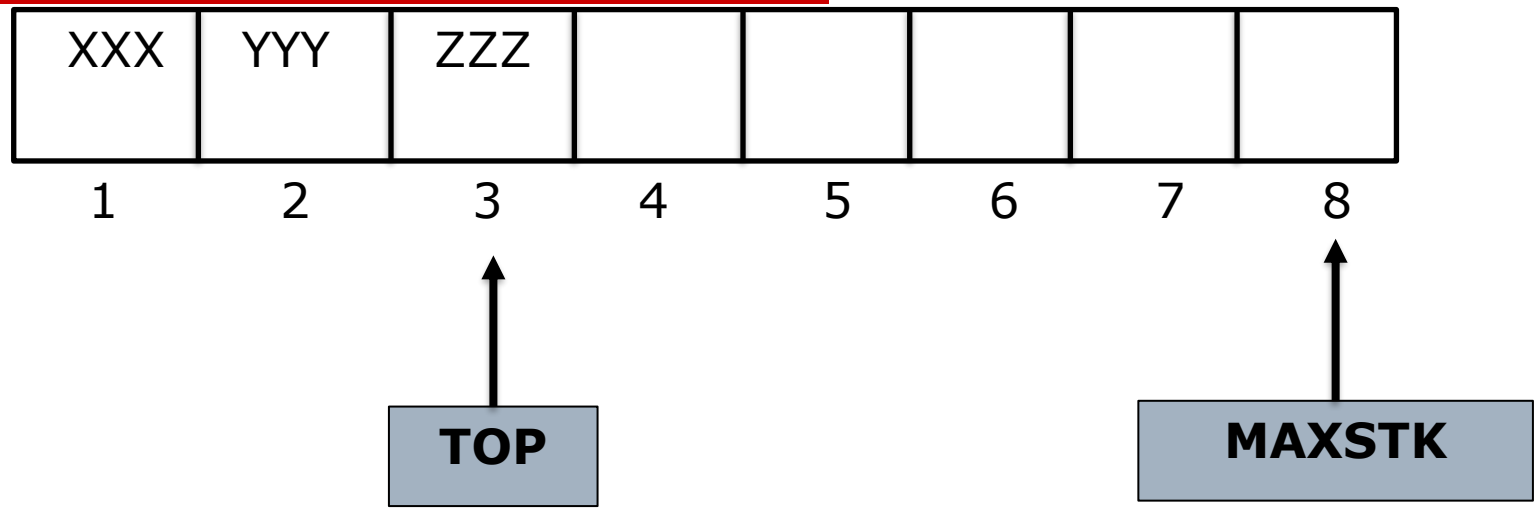


CASE 1: Push ITEM:=WWW

On to the stack.

1. **Since TOP=3 Control is transferred to Step 2**
2. **Top=3+1=4**
3. **STACK[TOP]=STACK[4]=WWW**
4. **Return**

Stack Example

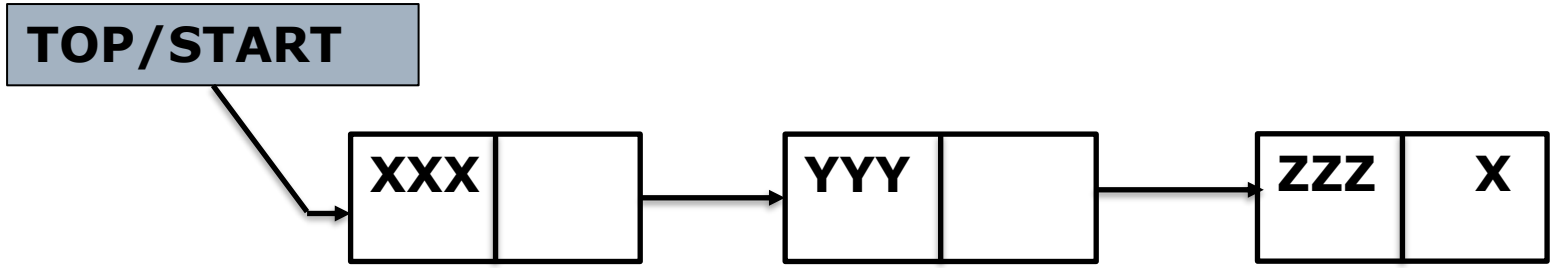


CASE 2: Pop an ITEM from the stack.

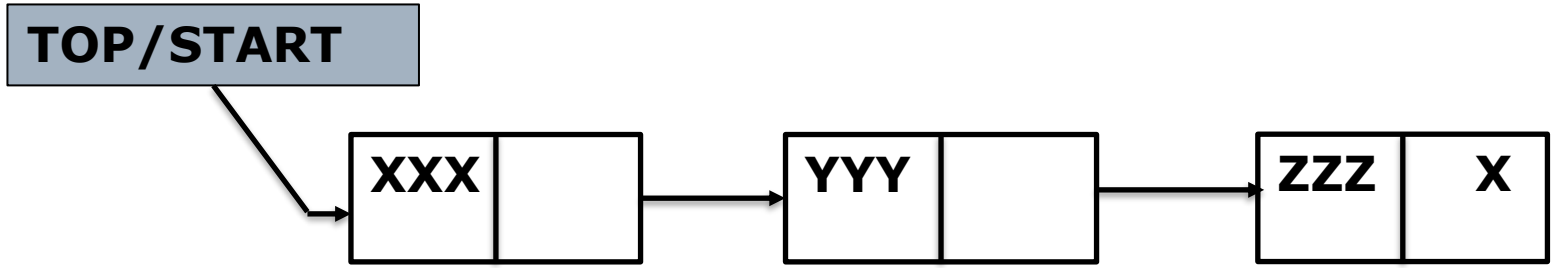
1. **Since TOP=3 Control is transferred to Step 2**
2. **ITEM=ZZZ**
3. **TOP=TOP-1=2**
4. **Return**

Linked Representation of Stacks

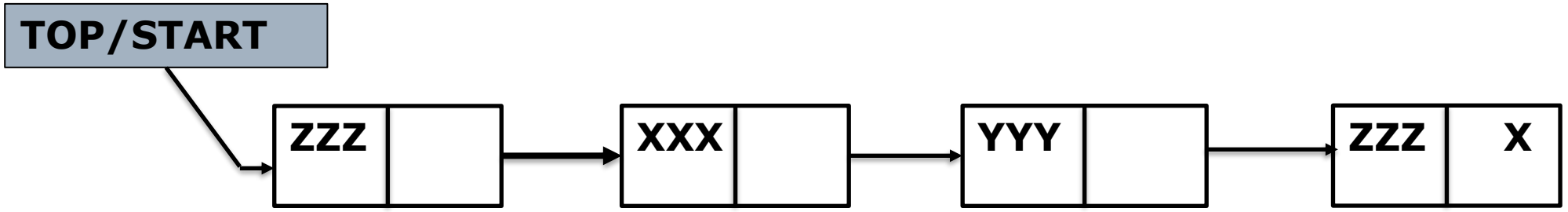
- The linked representation of a stack, commonly termed as linked stack i.e. a stack that is implemented using a singly linked list.
- The START pointer of the linked list behaves as the TOP pointer variable of the stack.
- The NULL pointer of the last node in the list signals the bottom of the stack.



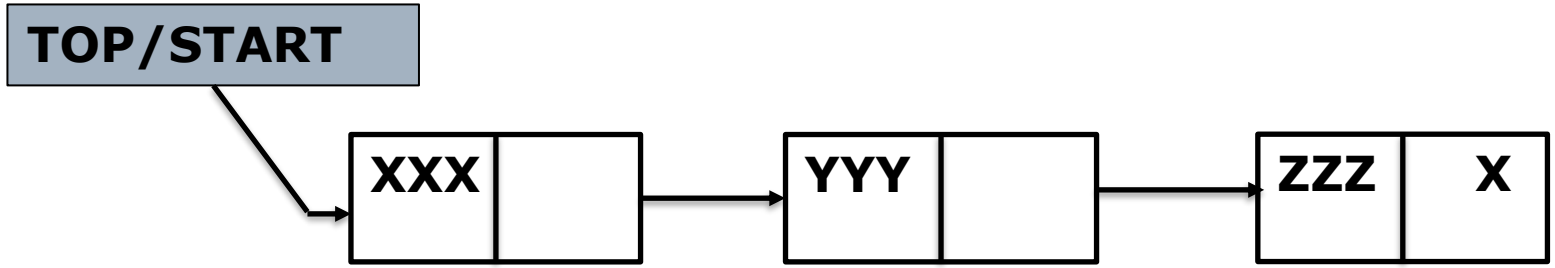
Linked Representation of Stacks



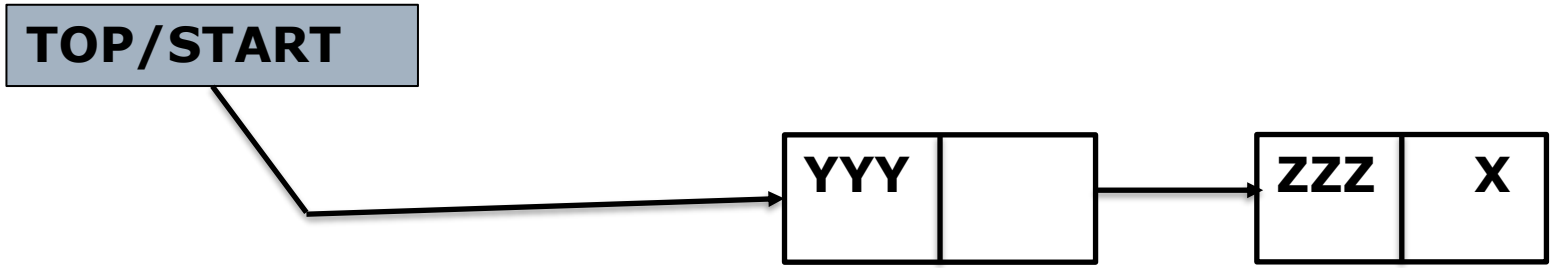
- PUSH Operation

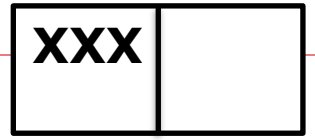


Linked Representation of Stacks



- POP Operation





Linked Representation of Stacks

- **Procedure 3: PUSH_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)**

This procedure pushes an ITEM into a Linked stack.

```
Step 1: [Available Space?]
        if AVAIL=NULL then
                Write "Overflow"
                Return
Step 2: [Remove first node from AVAIL list]
        Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
Step 3: Set INFO[NEW]:=ITEM
Step 4: Set LINK[NEW]:=TOP
Step 5: Set TOP:=NEW
Step 6: Return
```

Linked Representation of Stacks

- **Procedure 4: POP_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)**

  This procedure deletes the top element of the linked stack and assigns it to the variable ITEM

Step 1: [Stack has an item to be removed?]

```
if TOP=NULL then
        Write “Underflow”
        Return
```

Step 2: Set ITEM:=INFO[TOP]

Step 3: Set TEMP:=TOP and TOP:=LINK[TOP]

Step 4: Set LINK[TEMP]:=AVAIL and AVAIL:=TEMP

Step 5: Return

Applications of Stacks

- **Reversing a list**

This can be accomplished by pushing each character onto the stack as it is read.

When the line is finished , characters are popped off the stack-they come off in reverse order.

- **Polish Notation**

The process of writing the operators of an expression either before their operands or after them is called the polish notation.

The notation refers to these arithmetic expression in three forms:

1. **Prefix Notation**
2. **Postfix Notation**
3. **Infix Notation**

Applications of Stacks

- **Polish Notation**

1. Prefix Notation

If the operator symbols are placed before its operands, then expression is in prefix notation.

For E.g.: A+B → +AB

2. Postfix Notation

If the operator symbols are placed after its operands, then expression is in postfix notation.

For E.g.: A+B → AB+

3. Infix Notation

If the operator symbols are placed between its operands, then expression is in infix notation

Examples

Infix	Prefix	Postfix
A+B	+AB	AB+
(A-C)*B	* - ACB	AC-B*
A+(B*C)	+A*BC	ABC*+
(A+B)/(C-D)	/+AB-CD	AB+CD-/
(A+(B*C))/(C-(D*B))	/+A*BC-C*DB	ABC*+CDB*-/

Conversion of Infix to Postfix Expression

- Evaluation order in which the operations are executed
  - ✓ Brackets or Parenthesis
  - ✓ Exponentiation
  - ✓ Multiplication or Division
  - ✓ Addition or Subtraction

  NOTE: The operators with the same priority are evaluated from left to right.

Algorithm to convert Infix to Postfix Expression

The algorithm uses a stack to temporarily hold operators and parenthesis

Algorithm 5: POLISH(Q,P)

- Q → Infix expression
- P → Equivalent postfix expression

Step 1: Push “(“ onto STACK and add “)” to end of Q

Step 2: Scan Q from left to right and repeat Step 3 to 6 for each element of Q until the stack is empty.

Step 3: If an operand is encountered, add it to P.

Step 4: If a left parenthesis is encountered, put it on the STACK.

Algorithm to convert Infix to Postfix Expression

Step 5: If an operator $\otimes$ is encountered, then :

(a) Repeatedly pop from STACK and add to P each operator which has the same precedence as or higher precedence than $\otimes$

(b) Add $\otimes$ to STACK

[End of if structure]

Step 6: If the right parenthesis is encountered then:

(a) Repeatedly pop from STACK and add to P each operator until a left parenthesis is encountered.

(b) Remove the left parenthesis

[End of if structure]

[End of Step 2 Loop]

Step 7: Exit

Example to convert Infix to Postfix Expression

Transform Q into its equivalent postfix expression P.
Q: A + (B * C – (D / E ↑ F) * G) * H

Solution:

A	+	(	B	*	C	–	(	D	/	E	↑	F	)	*	G	)	*	H	)
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Example to convert Infix to Postfix Expression

A + (B * C – (D / E ↑ F) * G) * H)

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Symbol Scanned	Stack	Expression P
1. A	(	A
2. +	(+	A
3. (	(+(	A
4. B	(+(	AB
5. *	(+(*	AB
6. C	(+(*	ABC
7. -	(+(-	ABC*
8. (	(+(-(	ABC*
9. D	(+(-(	ABC*D
10./	(+(-(/	ABC*D
11.E	(+(-(/	ABC*DE

Example to convert Infix to Postfix Expression

A	+	(	B	*	C	–	(	D	/	E	↑	F	)	*	G	)	*	H	)
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Symbol Scanned	Stack	Expression P
12. ↑	(+(-(/ ↑	ABC*DE
13. F	(+(-(/ ↑	ABC*DEF
14.)	(+(-	ABC*DEF ↑ /
15. *	(+(-*	ABC*DEF ↑ /
16. G	(+(-*	ABC*DEF ↑ /G
17.)	(+	ABC*DEF ↑ /G*-
18. *	(+*	ABC*DEF ↑ /G*-
19. H	(+*	ABC*DEF ↑ /G*-H
20.)	-	ABC*DEF ↑ /G*-H*+

Evaluation of a Postfix Expression

Algorithm 5: This algorithm finds the VALUE of an arithmetic expression P written in postfix notation

1. **Add a right parenthesis “)” at the end of P**
2. **Scan P from left to right and repeat step 3 and 4 for each element of P until the “)” encountered.**
3. **If an operand is encountered, put it on STACK.**
4. **If an operator $\otimes$ is encountered, then**
   - **(a) Remove the top two elements of STACK where A is the top element and B is the next-to-top element.**
   - **(b) Evaluate B $\otimes$ A**
   - **(c) Place the result of (b) back on STACK**
   - **[End of If structure]**
   - **[End of Step 2 Loop]**
5. **Set VALUE equal to top element on STACK**
6. **Exit**

Evaluation of a Postfix Expression: Example

P: 5 6 2 + * 12 4 / -)

1 2 3 4 5 6 7 8 9 10

Symbol Scanned		STACK
1.	5	5
2.	6	5 , 6
3.	2	5 , 6, 2
4.	+	5 , 8
5.	*	40
6.	12	40 , 12
7.	4	40 , 12 , 4
8.	/	40 , 3
9.	-	37
10.	)	

VALUE=37

Recursion

Recursive Procedures

- If a procedure contain either a call statement to itself or a call statement to a second procedure that may eventually result in a call statement back to the original procedure. Then such a procedure is called a recursive procedure.
- For e.g.: The problem of factorial can be solved using a recursive procedure
- **Properties of recursive procedure:**

(i) There must be certain criteria called the base criteria for which the procedure does not call itself.

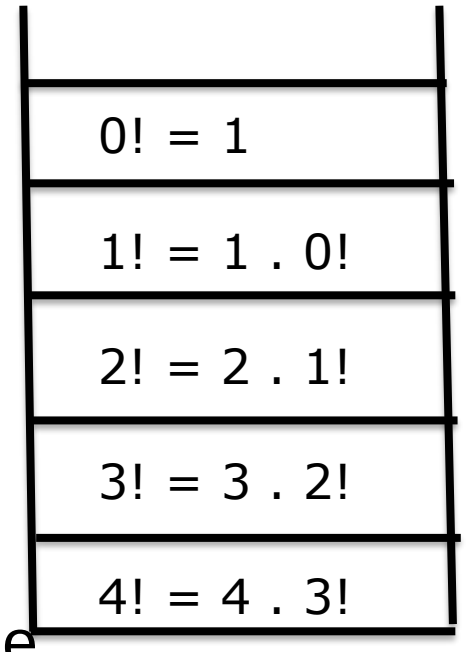
(ii) Each time the procedure does call itself it must be closer to the base criteria

Recursion: Example

- Find 4! Using recursion

(i) 4! = 4 . 3!
(ii) 3! = 3 . 2!
(iii) 2! = 2 . 1!
(iv) 1! = 1 0!
(v) 0! = 1
(vi) 1! = 1 . 1 = 1
(vii) 2! = 2 . 1 = 2
(viii) 3! = 3 . 2 = 6
(ix) 4! = 4 . 6 = 24

From Step (vi) to (ix) we back track
This type of postponed processing lends itself to the use
Of stacks.



Recursion: Example

Let a and b denote +ve integers. Suppose a function Q is defined recursively as follows:

$$Q(a,b) = \begin{cases} 0 & \text{if } a < b \\ Q(a-b, b) + 1 & \text{if } b \le a \end{cases}$$

(a) Find the value of Q(2, 3) and Q(14, 3)
(b) What does this function do ? Find Q (5861, 7)

Solution:
(a) Q(2,3) = 0 since 2<3
Q(14, 3) = Q(11, 3) + 1
= [Q (8, 3) +1] +1 = Q(8, 3) + 2
= [Q(5, 3) +1] + 2 = Q(5, 3) + 3
= [Q(2, 3) + 1] + 3 = Q(2, 3) + 4
= 0 + 4 = 4

Recursion: Example

Let a and b denote +ve integers. Suppose a function Q is defined recursively as follows:

$$Q(a,b) = \begin{cases} 0 & \text{if } a < b \\ Q(a-b, b) + 1 & \text{if } b \le a \end{cases}$$

(a) Find the value of Q(2, 3) and Q(14, 3)
(b) What does this function do ? Find Q (5861, 7)

Solution:
(b) Each time b is subtracted from a, the values of Q is increased by 1. Hence Q(a,b) finds the quotient when a is divisible by b. Thus, Q(5861, 7) = 837