SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Stack

swatimali@somaiya.edu

Somaiya TRUST

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Outline

- Stack – concept
- Stack ADT
- Stack operations
- Stack implementations
- Stack applications
- Summary
- Queries?

Somaiya TRUST

Stack

- Last In First Out
- Elements can be added or removed only from one end
- Gives access only to element at the top of data structure

What is this good for ?

- To store history in a Web browser
- Undo sequence in a any application software or text editor
- Saving local variables during function calls
- Recursions
- Watchlists?

A Stack

- Definition:
  - An ordered collection of homogenous data items
  - Can be accessed at only one end (the top)
- Operations:
  - Create an empty stack
  - check if it is empty
  - Push: add an element to the top
  - Pop: remove the top element
  - Peek: retrieve the top element(Not the deletion)
  - Destroy : remove all the elements one by one and destroy the data structure

The Stack ADT: Value definition

Abstract typedef StackType(ElementType ele)

Condition: none

Stack ADT: Operator definition

1. Abstract StackType CreateEmptyStack()

Precondition: none

Postcondition: CreateEmptyStack is created

2. Abstract StackType PushStack(StackType Stack, ElementType Element)

Precondition: Stack not full <u>or</u> NotFull(Stack)= True

Postcondition: stack= stack + Element at the top

<u>Or</u> Stack= original stack with new Element at the top

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Stack ADT: Operator definition

3. Abstract ElementType PopStack(StackType stack)

Precondition: Stack not empty or NotEmpty(Stack)= True

Postcondition: PopStack= element at the top,

Stack = stack - Element at the top

Or Stack= original stack without top Element

4. Abstract DestroyStack(StackType Stack)

Precondition: Stack not empty or NotEmpty(Stack)= True

Postcondition: Element from the stack are removed one by one starting from top to bottom.

Empty(Stack)= True

Somaiya TRUST

Stack ADT: Operator definition

5. Abstract Boolean NotFull(StackType stack)

Precondition: none

Postcondition: NotFull(Stack)= true if Stack is not full

NotFull(Stack)= False if Stack is full.

6. Abstract Boolean NotEmpty(StackType stack)

Precondition: none

Postcondition: NotEmpty(Stack)= true if Stack is not empty

~Empty(Stack)= False if Stack is empty.

Stack ADT: Operator definition

7. Abstract ElementType Peep(StackType stack)

Precondition: Stack not empty or NotEmpty(Stack)= True

Postcondition: PeepStack= element at the top,

Stack = original stack

Exercise: Stacks

Operation				
–Push(8)	8	Top		
–Push(3)	8	3	Top	
–Pop()	8	Top		
–Push(2)	8	2	Top	
–Push(5)	8	2	5	Top
–Pop()	8	2	Top	
–Pop()	8	Top		
–Push(9)	8	9	Top	
–Push(1)	8	9	1	Top

Implementing a Stack

- At least three different ways to implement a stack
  - array
  - vector
  - linked list
- Which method to use depends on the application
  - what advantages and disadvantages does each implementation have?

Implementing Stacks: Array

- Advantages -best performance
- Disadvantage - fixed size
- Basic implementation
  - initially empty array
  - field to record where the next data gets placed into
  - if array is full, push() returns false
    - otherwise adds it into the correct spot
  - if array is empty, pop() returns null
    - otherwise removes the next item in the stack

Implementing a Stack: Vector

- Advantages
  - grows to accommodate any amount of data
  - second fastest implementation when data size is less than vector size
- Disadvantage
  - slowest method if data size exceeds current vector size
    - have to copy everything over and then add data
  - wasted space if anomalous growth
    - vectors only grow in size – they don't shrink
  - can grow to an unlimited size
    - I thought this was an advantage?
- Basic implementation
  - virtually identical to array based version

Implementing a Stack: Linked List

- Advantages:
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantages
  - the common case is the slowest of all the implementations
- Basic implementation
  - list is initially empty
  - *push()* method adds a new item to the head of the list
  - *pop()* method removes the head of the list

Writing an algorithm

- Specify algorithm name, list of inputs, data types of the inputs and return data types clearly
- Specify purpose of the algorithm
- Algo should produce at least one Output
- Definiteness: Each step must be clear and unambiguous.
- Should react correctly to all valid and invalid inputs
- Finiteness: If we trace the steps of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps.
- Effectiveness:
- Comment Session: Comment is additional info of program for easily modification. In algorithm comment would be appear between two square bracket []. For example: [this is a comment of an algorithm].

Stack ADT: Array Implementation

1. Algorithm StackType CreateStack()

//This Algorithm returns an empty stack- stack

```
{ integer StackTop =-1;
Return stack;
}
```

2. Algorithm StackType PushStack(StackType Stack, ElementType Element)

// This algorithm accepts a StackType stack and ElementType Element as input and adds ‘Element’ at the top of ‘stack’. StackTop is an integer index that holds current value of StackTop position.

```
{
        if NotFull(Stack)= True
        stack[++StackTop]= Element
        Else “Error Message”
}
```

Stack ADT: Array Implementation

3. Algorithm ElementType PopStack(StackType stack)

// This algorithm accepts a stack as input and returns 'Element' at the top of 'stack'.

```
{  if NotEmpty(Stack)= True
Return Stack[StackTop--]
Else  print "Error Message"
}
```

4. Abstract DestroyStack(StackType Stack)

//This algorithm returns all the elements from Stack in LIFO order and destroys the data structure

```
{ if NotEmpty(Stack) = true
    while(NotEmpty(Stack))
        print PopStack(Stack)
   else print "Error Message"
}
```

Stack ADT: Array Implementation

5. Abstract Boolean NotFull(StackType stack)

```
// This algorithm returns true if the stack is not full, false otherwise.
{ if NotFull(Stack)
          retrun True
  else
          return False
}
```

6. Abstract Boolean NotEmpty(StackType stack)

```
// This algorithm returns true if the stack is not empty, false otherwise.
{ if NotEmpty(Stack)
          retrun True
  else
          return False
}
```

Stack ADT: Array Implementation

7. Abstract ElementType Peek(StackType stack)

//// This algorithm accepts a stack as input and returns 'Element' at the top of 'stack'.

{ if NotEmpty(Stack)= True

Return Stack[StackTop]

Else print "Error Message"

}

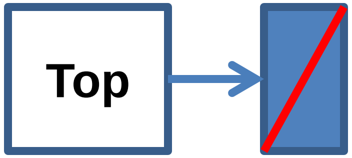
Implementing Stacks: Linked List

```
Struct NodeType{
            ElementType Element;
            NodeType Next;
      }
```

1. Algorithm StackType CreateStack()

//This Algorithm creates and returns an empty stack- pointed by a pointer-Top

```
{ createNode(Top);
Top =NULL;
}
```



Implementing Stacks: Linked List

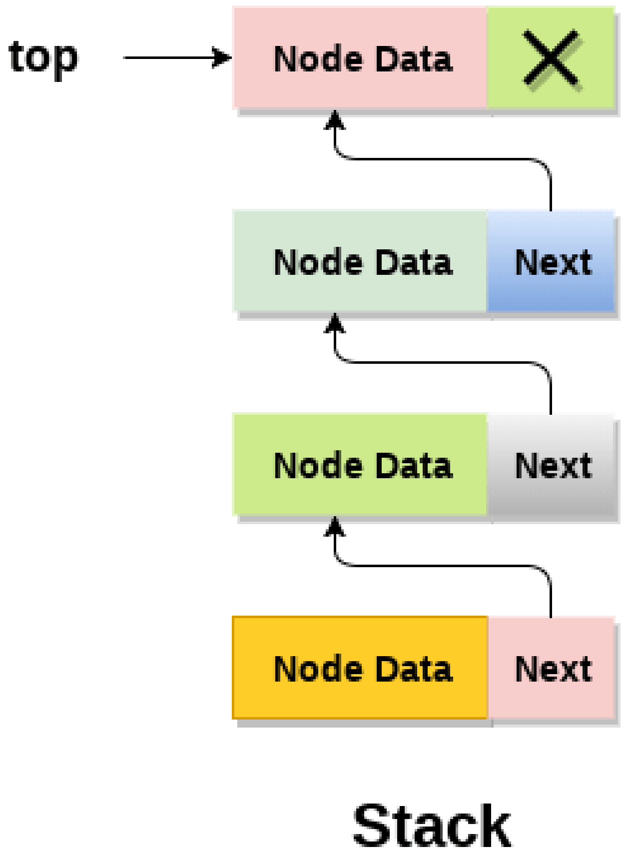


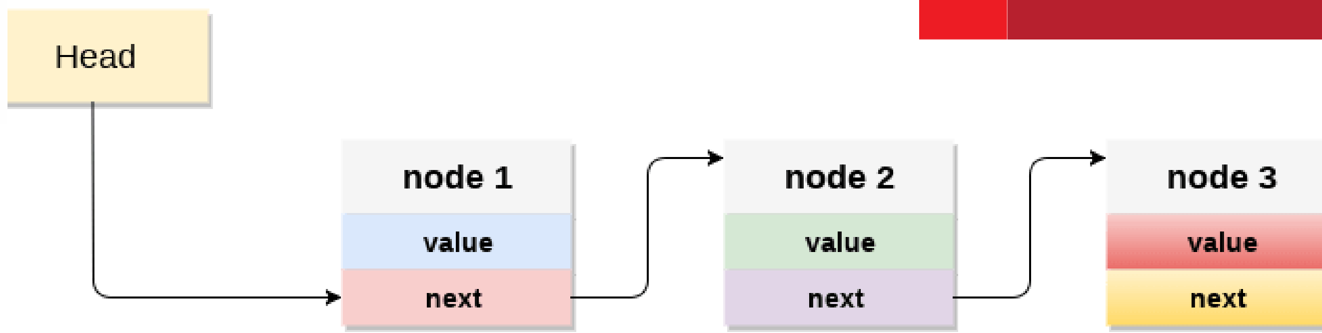
Image Courtesy: https://www.javatpoint.com/ds-linked-list-implementation-of-stack

Implementing Stacks: Linked List

2. StackType PushStack(StackType Stack, NodeType NewNode)

// This Algorithm adds a NewNode at the top of 'stack'. Top is an pointer that points to the topmost Stack node.

```
{ if Top ==NULL // first element in stack
        NewNode->Next = NULL;
        Top=NewNode;
 Else NewNode->Next=Top;// General case
        Top=NewNode;
}
```



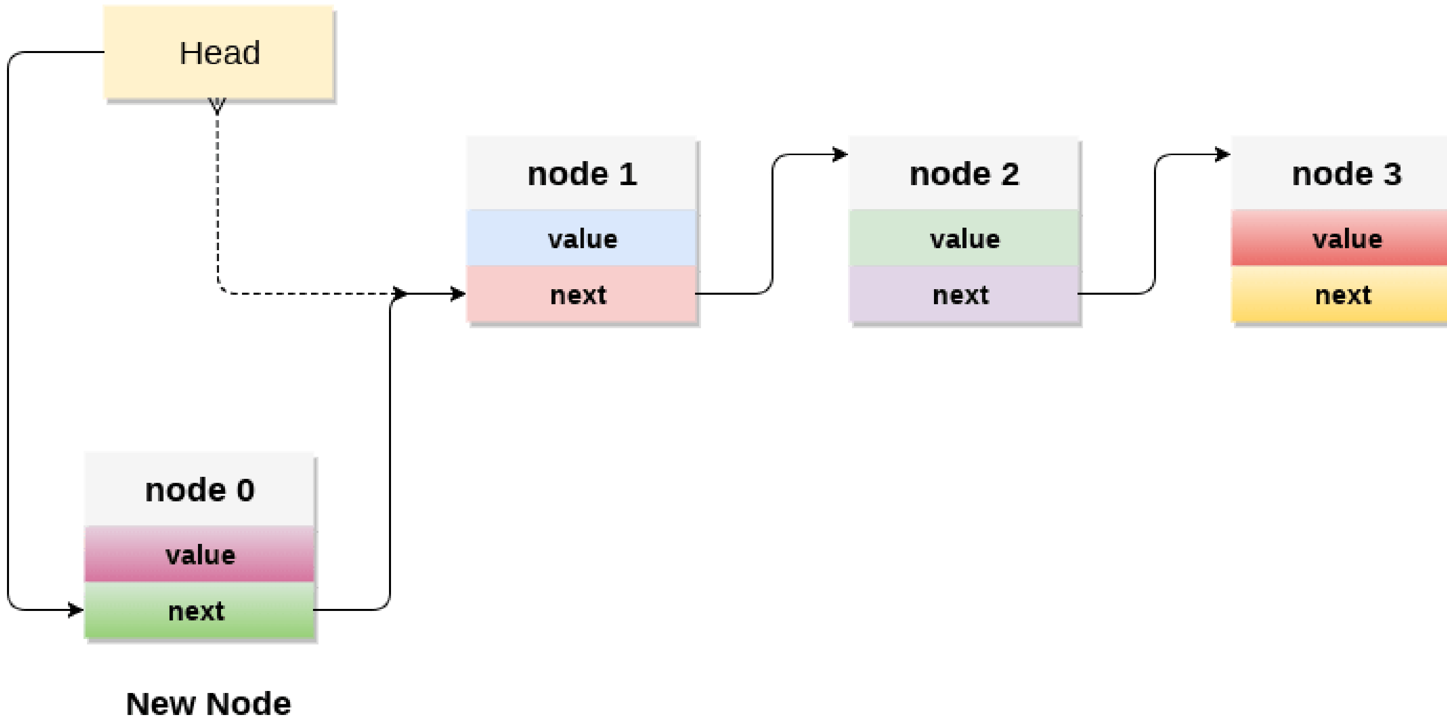


Image Courtesy: https://www.javatpoint.com/ds-linked-list-implementation-of-stack

Implementing Stacks: Linked List

3. Algorithm ElementType PopStack(StackType stack)

//This algorithm returns value of ElementType stored in topmost node of stack. Temp is a temproary node used in pop process.

```
{ if Top==NULL
          Print “Underflow”
 Else
          {
          createNode(Temp);
          Temp=Top;
          Top= Top->next;
          Return(temp->Data);
          }
}
```

Implementing Stacks: Linked List

4. Abstract DestroyStack(StackType Stack)

//This algorithm returns values stored in data structure and free the memory used in data structure implementation.

```
{{ if Top==NULL
           Print "Underflow"
 Else      createNode(Temp);
           while(NotEmpty(stack))
           {
           Temp=Top;
           Top= Top->next;
           Return(temp->Data);
           }
}
```

Implementing Stacks: Linked List

5. Abstract ElementType Peep(StackType stack)

//This algorithm returns value of ElementType stored in topmost node of stack.

```
{ if Top==NULL
        Print "Error Message"
 Else
        Return(Top->Data);
}
```

6. Abstract DisplayStack(StackType stack)

//This algorithm Prints all the Elements stored in stack. Temp purpose?

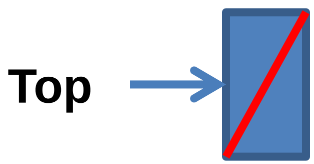
```
{ if Top==NULL
          Print "Error Message"
 Else {createNode(Temp)
          Temp=Top;
          While(Temp!=Null)
                    Print(Temp->Data);
                    Temp= Temp->next;
          }
}
```

Implementing Stacks: Linked List

- Push(8)
- Push(3)
- Pop()
- Push(2)
- Push(5)
- Pop()
- Peek()
- Peek()
- Pop()
- Push(9)
- Push(1)

Implementing Stacks: Linked List

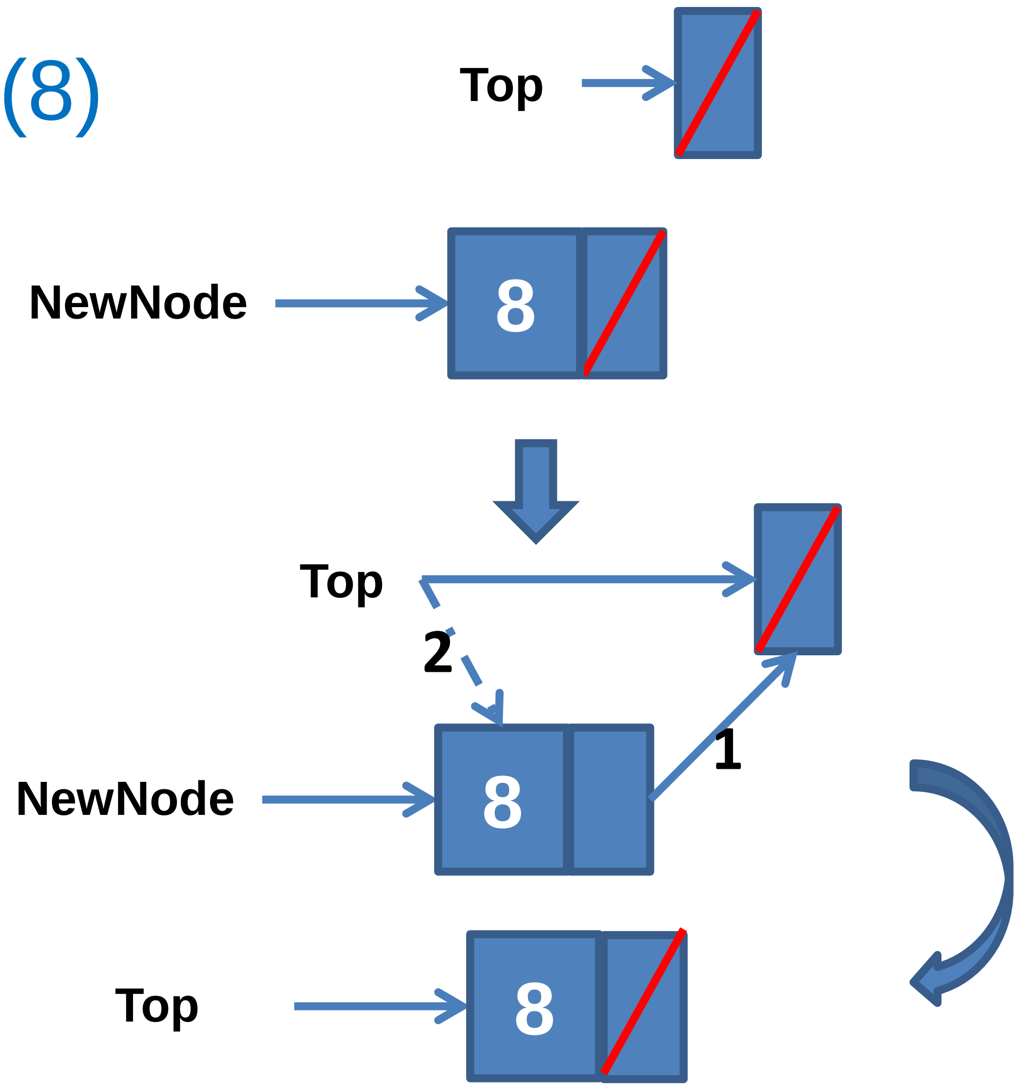
- Create empty stack



Push(8)

Implementing Stacks: Linked List

Push(8)



Stack Applications

- Stacks are a very common data structure
  - Compilers(parsing data between delimiters/ brackets)
  - operating systems (program stack)
  - virtual machines
    - manipulating numbers
      - pop 2 numbers off stack, do work (such as add)
      - push result back on stack and repeat
  - Algorithms
    - backtracking
  - artificial intelligence
    - finding a path

Stack applications

1. Parentheses Matching Algorithm

Algorithm Boolean ParenMatch(*X*,*n*):

Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

Output: **true** if and only if all the grouping symbols in *X* match

Let *S* be an empty stack

for *i*=0 to *n*-1 **do**

    if *X*[*i*] is an opening grouping symbol **then**

        S.push(*X*[*i*])

    else if *X*[*i*] is a closing grouping symbol **then**

        if *S*.isEmpty() **then**

            return false *{*nothing to match with*}*

        if *S*.pop() does not match the type of *X*[*i*] **then**

            return false *{*wrong type*}*

if *S*.isEmpty() **then**

    return true *{*every symbol matched*}*

else

    return false *{*some symbols were never matched*}*

Parentheses Matching Algorithm

i/p string= {()()}

step 1	Step 2	step 3	Step 4	step 5	step 6
i/p= {, push	i/p= (, push	i/p=), pop; ToS=(, match= true	i/p= (, push	i/p=), pop; ToS=(, match= true	i/p= }, pop; ToS= {, match= true
	(		(		
{	{	{	{	{	

After step 6, stack is empty. So given string of parenthesis is balanced

Parentheses Matching Algorithm

i/p string= {(){()}

i/p= {, push	i/p= (, push	i/p=), pop; ToS=(, match= true	i/p= {, push	i/p= (, push	i/p=), pop; ToS=(, match= true	i/p= }, pop; ToS=}, match= true	i/p= }, pop; underflow; Error
				(			
	(		{	{	{		
{	{	{	{	{	{	{	
step 1	Step 2	step 3	Step 4	step 5	step 6	step 7	step 8

After step 8, stack is nonempty but there are more characters in input string. So given string of parenthesis is not balanced

2. Infix to postfix

- Infix: operand operator operand
  - E.g a+b
- Postfix: operand operand operator
  - E.g. a b +
- Operator Precedence
  - ^ - exponential operator
  - *, /
  - +, -
- Infix to postfix expression with parenthesis

2. Infix to postfix

- Infix to postfix expression with parenthesis

(((a+b)*(c/d))^e)	
((ab+ * cd/) ^ e)	
((ab+cd/ *) ^ e)	
ab+cd/*e^	

2. Infix to postfix

- Infix to postfix expression with parenthesis

(((A + B) * C) – ((D - E) * (F + G)))	
((A B + * C) – (D E - * F G +))	
(A B + C *) – (DE – FG + *)	
A B + C * DE – FG + * -	

Infix to postfix

- Infix to postfix expression without parenthesis

A + B * C - D - E * F + G
A + BC* - D - EF* + G
ABC*+ - D - EF* + G
ABC*+D- - EF* + G
ABC*+D-EF*- + G
ABC*+D-EF*-G+

Infix to postfix process without parenthesis

- Create an empty stack called opstack for keeping operators. Create an empty list for output.
- Scan the input string from left to right.
  - If the input is an operand, append it to the end of the output list.
  - If the token is an operator, *, /, +, or -, push it on the opstack. However, first remove any operators already on the opstack that have higher or equal precedence and append them to the output list.
- When the input expression has been completely processed, check the opstack. Any operators still on the stack can be removed and appended to the end of the output list.

A + B * C - D - E * F + G

Input char	Opstack	Output
A		A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
-	-	ABC*+
D	-	ABC*+D
-	-	ABC*+D-
E	-	ABC*+D-E
*	-*	ABC*+D-E
F	-*	ABC*+D-EF
+	+	ABC*+D-EF*-
G	+	ABC*+D-EF*-G
NULL	EMPTYSTACK	ABC*+D-EF*-G+

- SOLVE : M*N+T^Q/F*A+B

MN*TQ^F/A*+B+

Infix to postfix process with parenthesis

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Scan X from left to right and repeat Step 2 to 5 for each element of X until the Stack is empty.
2. If an operand is encountered, add it to Y
3. If a left parenthesis is encountered, push it onto Stack.
4. If an operator is encountered ,then:
   - Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator until an opening parenthesis is encountered.
   - Add operator to Stack.
5. If a right parenthesis is encountered ,then:
   - Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
   - Remove the left Parenthesis.
6. END.

Input: input expression:(((A + B) * C) - ((D - E) * (F + G)))

Input char	stack	Output
(	(	
(	((	
(	(((	
A	(((	A
+	(((+	A
B	(((+	AB
)	((	AB+
*	((*	AB+
C	((*	AB+C
)	(	AB+C*
-	(-	AB+C*
(	(-(	AB+C*
(	(-((	AB+C*
D	(-((	AB+C*D
-	(-((-	AB+C*D
E	(-((-	AB+C*DE
)	(-(	AB+C*DE-
*	(-(*	AB+C*DE-
(	(-(*(	AB+C*DE-
F	(-(*(	AB+C*DE-F
+	(-(*(+	AB+C*DE-F
G	(-(*(+	AB+C*DE-FG
)	(-(*	AB+C*DE-FG+
)	(-	AB+C*DE-FG+*
)	EMPTY	AB+C*DE-FG+*-

3. Evaluation of postfix expression

- Create a stack for storing operands
- Scan the input expression from left to right
  - If the element is operand, push it onto the stack
  - If the element is operator, pop two operands, evaluate and push the result onto the stack
- If the expression is over, the stack contains the final answer

Input: input expression: AB+C*DE-FG+*-

e.g. A=2, B=3,C=1,D=4,E=5, F=7, G=8

Input char	stack
2	2
3	2, 3
+	(2+3)=5
1	5, 1
*	(5*1)= 5
4	5,4
5	5,4,5
-	5,-1
7	5,-1,7
8	5,-1,7,8
+	5,-1,15
*	5,-15
-	20

4. Reverse a string using Stack

1) Create an empty stack.
2) One by one push all characters of string to stack.
3) One by one pop all characters from stack and put them back to string.

5. Check if a string is palindrome

1) Push the input string onto the stack
2) POP characters ONE by one from stack and compare with string characters from left to right
3) If all comparisons are true, the string is palindrome

6. Recursion

- Definition: calling the same function again directly or indirectly
- Concept: represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion.
- The maximal number of nested calls (including the first one) is called *recursion depth*.

4. Recursion

Self study:

- Recursive Vs iterative implementation

Recursive function call

- The current function is paused.
- The execution context associated with it is remembered in a special data structure called *execution context stack*.
- The nested call executes.
- After it ends, the old execution context is retrieved from the stack, and the outer function is resumed from where it stopped.

Recursive function call

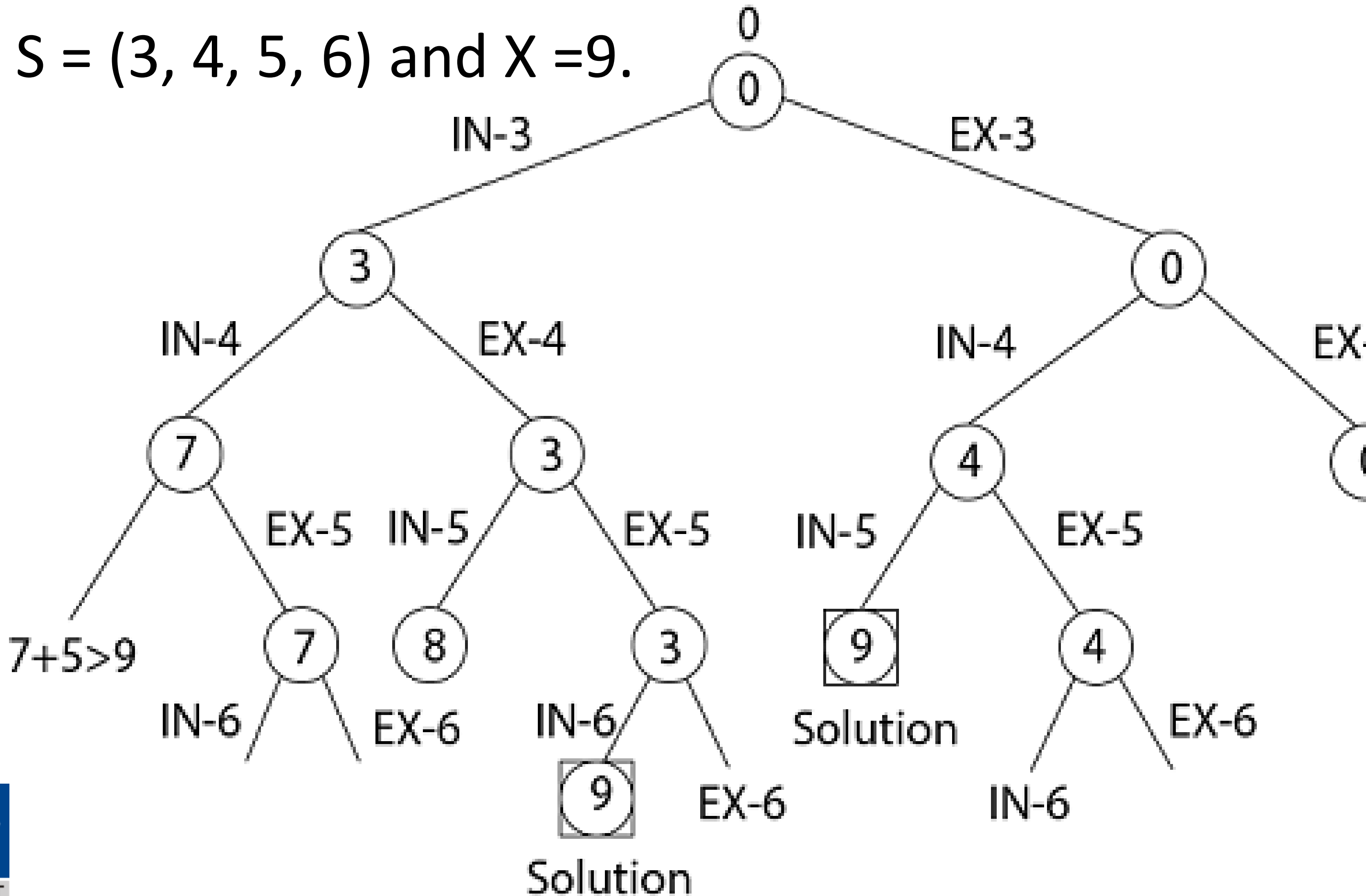
- In each recursive call, there is need to save the
  - current values of parameters,
  - local variables and
  - the return address (the address where the control has to return from the call).
- Also, as a function calls to another function, first its arguments, then the return address and finally space for local variables is pushed onto the stack.

Backtracking

- Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.
- Uses stack for storing solution path

Sum of subsets Backtracking

- S = (3, 4, 5, 6) and X =9.



Queries?

Thank you!