Data Structures

sushmakadge@somaiya.edu
swatimali@somaiya.edu

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?

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 or NotFull(Stack)= True

Postcondition: stack= stack + Element at the top

Or Stack= original stack with new Element at the top

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.

mpty(Stack)= True

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

–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: 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

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"

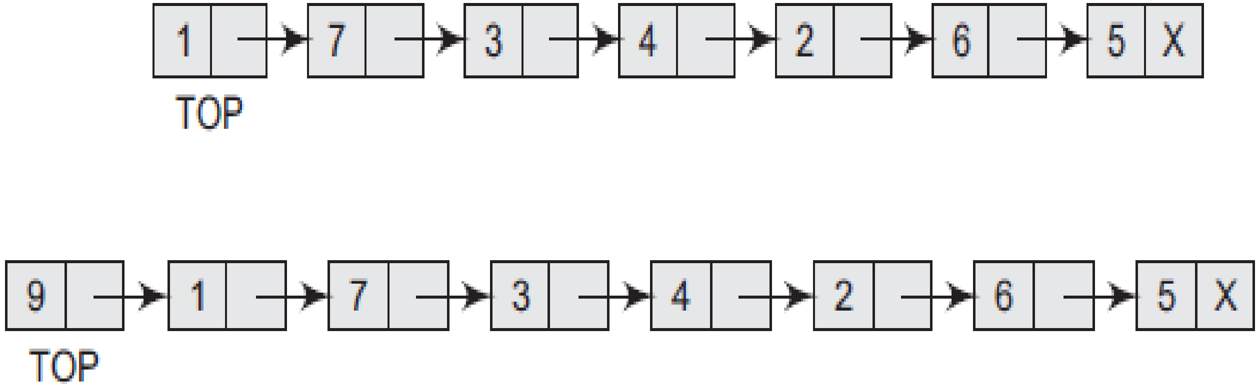
}

LINKED REPRESENTATION OF STACKs

- Technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size.
- If the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.
- The storage requirement of linked representation of the stack with n elements is O(n).

OPERATIONS ON A LINKED STACK

- A linked stack supports all the three stack operations, that is, push, pop, and peek.
- **Push Operation**



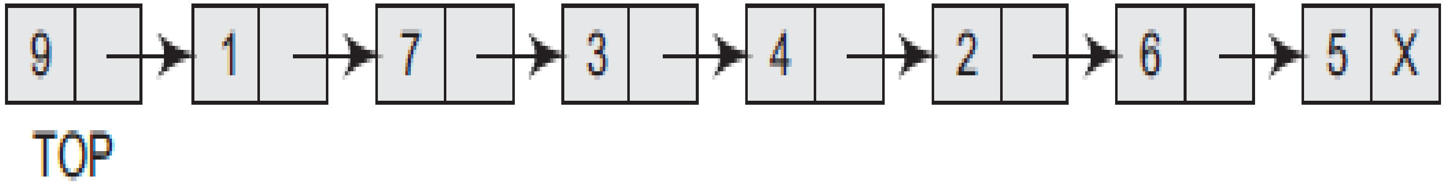
OPERATIONS ON A LINKED STACK

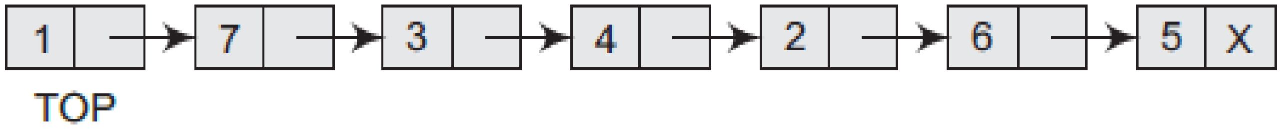
- Algorithm to insert an element in a linked stack

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

OPERATIONS ON A LINKED STACK

- Pop Operation





OPERATIONS ON A LINKED STACK

- Algorithm to delete an element from a linked stack

```
Step 1: IF TOP = NULL
              PRINT "UNDERFLOW"
              Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

APPLICATIONS OF STACKS

- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Reversing a list
- Parentheses checker
- Tower of Hanoi

APPLICATIONS OF STACKS

Evaluation of Arithmetic Expressions

- Polish Notations

Infix, postfix, and prefix notations are three different but equivalent notations of writing algebraic expressions.

INFIX :

The operator is placed in between the operands.

For ex, A+B;

Plus operator is placed between the two operands A and B.

Evaluation of Arithmetic Expressions- Polish Notations

Postfix notation :

- The operator is placed after the For ex, AB+
- The order of evaluation of a postfix expression is always from left to right
- Even brackets cannot alter the order of evaluation.
- The expression (A + B) * C can be written as:
- [AB+]*C
- AB+C* in the postfix notation
- Evaluation, addition will be performed prior to multiplication

Evaluation of Arithmetic Expressions- Polish Notations

Prefix notation :

The operator is placed before the operands. For example, +AB. While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.

Conversion of an Infix Expression into a Postfix Expression

The precedence of these operators can be given as follows:
Higher priority *, /, %
Lower priority +, –

Conversion of an Infix Expression into a Postfix Expression

Example 1 Convert the following infix expressions into postfix expressions.

(a) (A–B) * (C+D)

(b) (A + B) / (C + D) – (D * E)

Evaluation of Arithmetic Expressions- Polish Notations

(a) (A–B) * (C+D)

[AB–] * [CD+]

AB–CD+*

(b) (A + B) / (C + D) – (D * E)

[AB+] / [CD+] – [DE*]

[AB+CD+/] – [DE*]

AB+CD+/DE*–

Conversion of an Infix Expression into a Postfix Expression

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

(((A + B) * C) – ((D - E) * (F + G)))

Conversion of an Infix Expression into a Postfix Expression

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

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

Evaluation of Arithmetic Expressions- Polish Notations

Convert the following infix expressions into **prefix expressions.**

(a) (A + B) * C

(b) (A–B) * (C+D)

(c) (A + B) / (C + D) – (D * E)

Evaluation of Arithmetic Expressions- Polish Notations

Convert the following infix expressions into prefix expressions.

(a) (A + B) * C

(+AB)*C

*+ABC

(b) (A–B) * (C+D)

[–AB] * [+CD]

*–AB+CD

(c) (A + B) / (C + D) – (D * E)

[+AB] / [+CD] – [*DE]

[/+AB+CD] – [*DE]

–/+AB+CD*DE

Algorithm to convert an infix notation to 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 a character) 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 O 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 O
          b. Push the operator O to the stack
        [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

Convert the following infix expression into postfix expression

A – (B / C + (D % E * F) / G)* H

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (% *	A B C / D E
F	(– (+ (% *	A B C / D E F
)	(– (+	A B C / D E F * %
/	(– (+ /	A B C / D E F * %
G	(– (+ /	A B C / D E F * % G
)	(–	A B C / D E F * % G / +
*	(– *	A B C / D E F * % G / +
H	(– *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * –

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

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*-
NULL		

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+

Evaluation of a Postfix Expression

- The ease of evaluation acts as the driving force for computers to translate an infix notation into a postfix notation.
- The computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.
- Both these tasks—converting and evaluating make extensive use of stacks as the primary tool
- Using stacks, any postfix expression can be evaluated very easily.
- Every character of the postfix expression is scanned from left to right.

Evaluation of a Postfix Expression

- If the character encountered is an operand, it is pushed
- on to the stack.
- However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values.
- The result is then pushed on to the stack.
- Let us look at the algorithm to evaluate a 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 O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is 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
```

Algorithm to evaluate a postfix expression

Consider the infix expression given as 9 – ((3 * 4) + 8) / 4. Evaluate the expression.

The infix expression 9 – ((3 * 4) + 8) / 4 can be written as 9 3 4 * 8 + 4 / – using postfix notation. Look at Table, which shows the procedure.

Algorithm to evaluate a postfix expression

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

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

Input char	Opstack	Output
A		
+		
B		
*		
C		
-		
D		
-		
E		
*		
F		
+		
G		
NULL		

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+

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	
3	
+	
1	
*	
4	
5	
-	
7	
8	
+	
*	
-	

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

Convert the following infix expression into prefix expression

Ex: Given an infix expression (A – B/ C) * (A / K – L)

Step1: Reverse the infix string. Interchange Left & R parentheses.

(L – K / A) * (C / B – A)

Step 2: Obtain the corresponding postfix expression of the infix expression obtained as a result of Step 1.

Therefore, [L – (K A /)] * [(C B /) – A]

= [LKA/–] * [CB/A–]

= L K A / – C B / A – *

Step 3: Reverse the postfix expression to get the prefix expression

Therefore, the prefix expression is *** – A / B C – /A K L**

Convert the following infix expression into prefix expression

The algorithm is given

Ex : Given an infix expression
(A – B/ C) * (A / K – L)

Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.
Step 2: Obtain the postfix expression of the infix expression obtained in Step 1.
Step 3: Reverse the postfix expression to get the prefix expression

Algorithm to evaluate a prefix expression

```
Step 1: Accept the prefix expression
Step 2: Repeat until all the characters
        in the prefix expression have
        been scanned
         (a) Scan the prefix expression
             from right, one character at a
             time.
         (b) If the scanned character is an
             operand, push it on the
             operand stack.
         (c) If the scanned character is an
             operator, then
               (i) Pop two values from the
                   operand stack
              (ii) Apply the operator on
                   the popped operands
             (iii) Push the result on the
                   operand stack
Step 3: END
```

Evaluation of a Prefix Expression

There are a number of techniques for evaluating a prefix expression.

The simplest way of evaluation of a prefix expression is given in Fig.

For example, consider the prefix expression + – 2 7 * 8 / 4 12. Let us now apply the algorithm to evaluate this expression.

Algorithm to evaluate a prefix expression

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
-	24, 5
+	29

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.

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

Recursion

Definition:

- A recursive function is defined as 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 calling the same function again directly or indirectly.
- Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.

Recursion

- Every recursive solution has two major cases. They are
- Base case, in which the problem is simple enough to be solved directly without making any further calls to the same function. Ex factorial function
  when n = 1, because if n = 1, the result will be 1 as 1! = 1.
- Recursive case, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler subparts. factorial(n) = n × factorial (n–1)

Recursion 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.

Parentheses Matching Algorithm

- Stacks can be used to check the validity of parentheses in any algebraic expression.
- For example, an algebraic expression is valid if for every open bracket there is a corresponding closing bracket.
- For example, the expression (A+B} is invalid but an expression {A + (B – C)} is valid.

Parentheses Matching Algorithm

i/p string= {()()}

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
	(		(		
{	{	{	{	{	
step 1	Step 2	step 3	Step 4	step 5	step 6

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

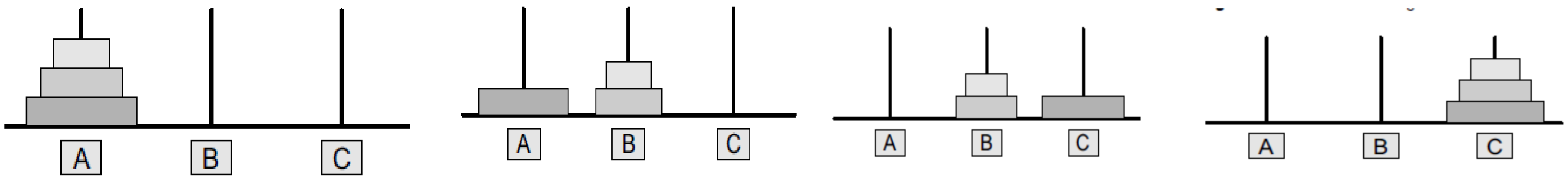
After step 8, stack is nonempty but there are more characters in input string. So given string of parenthesis is not balanced

Tower of Hanoi

The tower of Hanoi is one of the main applications of recursion.
It says, ‘if you can solve n–1 cases, then you can easily solve the nth case’.

Fig. which shows three rings mounted on pole A.
The problem is to move all these rings from pole A to pole C while maintaining the same order. The main issue is that the smaller disk must always come above the larger disk.

Tower of Hanoi



Tower of Hanoi

To summarize, the solution to our problem of moving n rings
from A to C using B as spare can be given as:
Base case: if n=1
Move the ring from A to C using B as spare
Recursive case:
Move n – 1 rings from A to B using C as spare
Move the one ring left on A to C using B as spare
Move n – 1 rings from B to C using A as spare

Queries???

Thank you!!