Stack

- Last in First Out
- Elements can be added or removed only only from one end
- Gives access only be element at the top od data structure.

Definition

→ An ordered collection od homogenous data items

→ Can be accessed ~~at~~ 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 & destroy the data structure.

The Stack ADT

Value definition:

Abstract typedef StackType (ElementType ele)

Condition : none

Operator definition:

1) Abstract Stack Type CreateStack()
→ Precondition: None
→ Postcondition: Empty Stack is created

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

3) Abstract Stack Type PopStack(Stack Type 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 Destroy Stack (Stake Type Stack)
→ Precondition: Stack not empty or Not Empty (stack) = True.
→ Postcondition: Elements from the stack are removed one by one starting from top to bottom.
Empty(stack) = True

5) Abstract Boolean NotFull(StackType Stack)
→ Precondition : none
→ Postcondition : Not Full (Stack) = True if stack is not full
Not Full(Stack) = False if stack is full

6) Abstract Boolean Not Empty(StackType Stack)
→ Precondition : None
→ Postcondition : Not Empty (Stack) = True if stack is not empty
Not Empty (Stack) = False if stack is empty

7) Abstract ElementType Peek (StackType Stack)
→ Precondition: Stack not empty or NotEmpty(Stack) = True.
→ Postcondition: Peek stack = element at the top
Stack = Orginal stack.

Implementation of Stack

→ Three ways

→ Array

→ Vector

→ linked List

1] ~~Disc~~ Array

Advantage

→ Best performance

Disadvantage

→ Fixed size

Algo

~~1) Algorithm~~

1) Algorithm Stack Type CreateStack ( )

{

integer Stack Top = -1;

Return stack;

}

2) Algorithm Stack Type PushStack (Stack Type,
Element Type Element)

{

if (Not Full (stack)) = True

Stack [++ Stack Top] = Element;

Else "Error";

}

3) Algorithm Element Type Pop stack (Stack Type
Stack)

{ if (NotEmpty (stack)) = True

Return stack [Stack Top --];
Else "Error";
}

4) Abstract DestroyStack (Stack Type Stack)
{
if (NotEmpty(stack)) = True
while (NotEmpty(stack))
print PopStack(Stack);
Else "Error";
}

5) Abstract Boolean NotFull (Stack Type Stack)
{ if NotFull(stack)
return True;
else
retun False;
}

6) Abstract Boolean Not Empty (StackType St
{
if Not Empty (Stack)
return True;
else
return False;
}

7) Abstract Element Type Peek (Stack Type Sta
{ if (Not Empty (Stack) = True
Return Stack [Stack Top];
Else "Error";
}

2] Vector

Advantage

any amount of data

→ Grows to accommodate ~~new data~~

→ Second fastest when data size < vector size

Disadvantage

→ Slowest when data size > vector size

→ wasted space

→ can grow to unlimited size

→ Algo

Same as array

3) Linked List

→ Advantage

→ Always constant time to push or pop

→ Can grow to an ∞ size

Disadvantage

→ Slowest method

Algo

→ Struct Node Type {
Element Type Element;
Node Type Next;

}

1) Algorithm Stack Type CreateStack ( )

{

Create Node (TOP);

TOP = NULL;

}

2) Stack Type PushStack (Stack Type Stack
,Node Type NewNode)
{
if TOP == NULL {
 NewNode → Next = NULL;
 TOP = NewNode; }
Else {
NewNode → Next = TOP;
 TOP = NewNode; }
}

3) Algorithm Element Type PopStack (Stack
Type Stack)
{
if TOP == NULL
~~"Error"~~ Print "Underflow";
Else
{
Create Node (Temp);
Temp = TOP;
 TOP = TOP → Next;
Return (Temp → data);
}

4) Abstract Destroy Stack (Stack Type Stack)
{
if TOP == NULL
 Print "Underflow";
Else
{

```
CreateNode (Temp);
While (Not Empty (Stack))
{
Temp = Top;
Top = Top -> Next;
Return (Temp -> Data);
}
```

5) Abstract ElementType Peep (StackType Stack)

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

6) Abstract Display Stack (StackType Stack)

```
{
if Top == NULL
Print "Error"
Else
{
CreateNode ~~Type~~ (Temp);
Temp = Top;
While (Temp != NULL) {
Print (Temp -> data);
Temp -> Next;
}
}
}
```

Application

1) Parentheses Matching

Eg i/p = String = {()()}

Input	Stack
{	{
(	{(
)	{
(	{(
)	{
}	Empty

Basically Once we shut the parentheses that stack is Empty in the output

2) Infix to Postfix

→ Precedence

i) ^

ii) *, /

iii) +, -

Eg M*N+T^Q/F*A+B

	Input	Stack	Output
1)	M		M
2)	*	*	M
3)	N	*	MN
4)	+	+	MN*
5)	T	+	MN*T
6)	^	+^	MN*T
7)	Q	+^	MN*TQ
8)	/	+/	MN*TQ^

classmate

Date
Page 23

9) F	+/	MN*TQ^F
10) *	+*	MN*TQ^F/
11) A	+*	MN*TQ^F/A
12) +	+	MN*TQ^F/A*+
13) B	Empty	MN*TQ^F/A*+B+

→ If inside the stack is higher precendent sign in step 2

→ There is a lower precendent sign in the input it pops the Higher precendent sign in the Output in step 9

→ When there is a lower precendent sign in the stack & higher in input the higher is pushed int the stack without popping the lower precendent sign in step 6

→ If equal precendent pop the one in stack and push the one in input in step 10

3) Infix to postfix with parenthesis

Eg (((A+B)*C) - ((D-E)*(F+G)))

Input	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+*-

→ So basically once you shut the parenthesis whatever was in the the parenthesis it gets popped

4) Postfix evaluation
→ Create a stack for storing operands
→ Scan the input expression from Left to right.

5) Reverse a string using Stack
→ Create a stack (empty)
→ One by one push ~~the~~ all the characters
→ One by one pop all the characters from stack & put them back to string.

6) Check if a string is Palidrome
→ Do 5
→ Check if equal then true else false

7) Recursion
→ Calling the same function directly or indirectly
→ Represent a problem in terms of one or more smaller Problems, and add one or more base conditions that stop the recursion
→ The maximal no. of nested calls is called recursion depth

Recursive function Call

→ Current function is paused
→ The execution context associated with it is remembered in a special ~~data~~ DS called execution context stack.
→ The nested call executes
→ After it ends, the old excution context is retrieved from the stack, & the outer function is resumed from where it stopped
→ Each recursive call, needs to save
- Current value
- Local variable
- Return Address

→ Also, as a function calls to another function first its arguments, then the return address & finally local variable is pushed.

Backtracking

→ It 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

→