Stacks

- Stack: An ordered collection of data items from which only the most recently inserted item can be deleted. New items are inserted with respect to the most recently inserted item.
- No limits on number of items.
- No limits on nature of items.
- The implementation will place limits.

1

Stacks

- LIFO - Last In, First Out.
- A stack is a LIFO structure.
- Standard stack operations
  - Push (insert)
  - Pop (delete)
  - Is_Empty.
- Stack of plates analogy

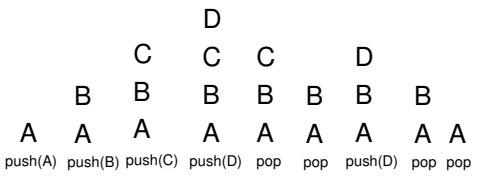
2

Stack Uses

- Operating systems use stacks
  - To support recursion
  - To facilitate program execution.
- Use stacks
  - When processing order is not important
  - When a simple structure is desired.
  - When application itself reflects LIFO order

3

Stack Example:



4

Stack ADT

- Standard stack operations
  - Push (insert)
  - Pop (delete)
  - Is_Empty
  - constructor
- Data: pointer to top of stack.
- Optional: Copy, Display, Peek, etc.

5

A PEZ® dispenser – an implementation of a stack

PEZ® is a registered trademark of PEZ Candy, Inc.

6

ADT Stack

Data
A list of items with a reference for the top of the stack, initialized to empty

Methods
isEmpty
Input: None
Preconditions: None
Process: Check whether the stack is empty.
Postcondition: Return True if stack is empty and False otherwise.
Output: None

Pop
Input: None
Precondition: Stack is not empty.
Process: Remove the item from the top of the stack.
Postcondition: Element at the top of the stack is removed.
Output: Return the element from the top of the stack.

7

ADT Stack (continued)

Methods (continued)
Push
Input: An item for the stack.
Precondition: None
Process: Store the item on the top of the stack.
Postcondition: The stack has a new element at the top
Output: None

Peek *//an optional method*
Input: None
Precondition: Stack is not empty.
Process: Retrieve the value o f t he item on the top of the stack.
Postcondition: The stack is unchanged.
Output: Return the value of the item from the top of the stack.

ClearStack *//an optional method*
Input: None
Precondition: None
Process: Deletes all the items in the stack and resets the top of the stack.
Postcondition: None
Output: None

Are there other desirable methods?

8

Stack ADT: Interface

```
public interface Stack {
    public int size();
    public boolean isEmpty();
    public Object peekb();
    public void push (object element);
    public Object pop()
}

public class ArrayStack implements Stack {…}
public class LinkedStack implements Stack {…}
```

9

Implementation of Interface Example

```
public class LinkedStack implements Stack
private Node top;
private int size;
public Linked Stack() {
        top = null;
        size =0;
}
public boolean isEmpty() {
        if (top == null) return true;
        return false;
}
public void push (Object obj) {
        node n = new Node();
        n.setElement(obj);
        n.setNext(top);
        top = n;
        size++;
}                                          …….etc
```

10

Stack Application Examples

- Parsing for delimiters.
- Convert an expression to postfix or prefix
- Evaluate a converted expression
- Convert infix to postfix or prefix

11

Parsing Delimiters: Approach 1

[(A + B)-C *{(2-3)C *}+5]$2

- count left delimiters ((A + B) + C)
- count right delimiters)A + B(
- compare: if equal then okay

12

Parsing Ddelimiters: Approach 2

[(A + B)-C *{(2-3)C *}+5]$2

- Count as you go along
- [(A+B)-C*{(2-3)*4}+5]$2
- 1 2 1 2 3 2 1 0
- Nonzero results indicate error
- Negative result indicates error

13

Parsing Delimiters

)A+B(((A+B) (A+B))

-1 12 1 +1 0(-1)

(A+B) {A+B)

1 0 1 0

- Won't work with mixed delimiters
- Need separate counter for each type
- Limited number of types can handle

14

Parsing Delimiters: Stacks

- Stacks can correctly parse
- Mixed delimeters
- Any number of delimiters

15

Parsing Delimiters: Stacks

Method

While input, read expression from left
If not a delimiter ignore
else if left delimiter, push
else //right delimiter
pop and compare
if equal continue,
else error
//no more input.
if stack not empty then error

16

Parsing Delimiters:Example

Item	Action	Stack	Comment
[	push	[	
(	push	[(	
A			
+			
B			
)	Pop (	[	ok so continue
-			
C			
*			
{	push	[{	
(	push	[{(	
2			
-			
3			
)	pop (	[{	okay, so continue
*			
4			
}	pop {	[	okay, so continue
+			
5			
]	pop [		okay, so continue
$			
2			$ means Expontiation

17

Convert Expression

- infix (3+4)*5 3+4*5
- prefix *+345 +3*45
- postfix 34+5* 345*+

18

Convert Expression:

Precedence Rules

unary operations
$
* /
\+ -
relational operations (=,<, >,etc.)

Use parentheses to change evaluation order

19

Convert Expression:

Left to Right Rule

- If two operators have the same precedence then we break the tie by giving precedence to the operator on the left.
- Consistent with ordinary arithmetic
- Use if same precedence
- Evaluate from left to right
- Consistent, repeatable results

20

Convert Expression:

Left to Right Rule

\+ and * are communitive & associative
(3 + 4) + 5 = 3 + (4 + 5) = 3 + (5 + 4)

\- & / are not communitive & associative
(3 - 4) - 5 <> 3 - (4 - 5) <> 3 - (5 - 4)
3/5 <> 5/3

Exception to left to right evaluation
2$3$2 = (2$3)$2 or 2$(3$2)
2^{3^2} $(2^3)^2$ $2^{(3^2)}$

21

Convert Expression

- we will defer conversion
- a compiler might do this

22

Postfix Evaluation

Method

while input, read expression // from left

if operand, push

else // operation

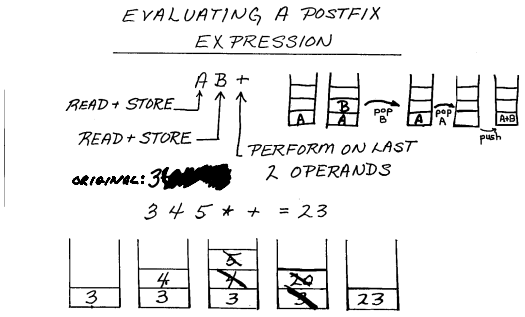
pop A

pop B

perform operation //B op A

push result

23



EVALUATING POSTFIX

6 2 3 + - 3 8 2 / + * 2 $ 3 +

OPERAND STACK	OPERATION
6	
6 2	
6 2 3	+
6 5	-
/	
/ 3	
/ 3 8	
/ 3 8 2	/
/ 3 4	+
/ 7	*
7	
7 2	$
49	
49 3	+
52	

((6 - (2 + 3)) * (3 + (8 / 2))) $ 2 + 3

25

Postfix Evaluation

To prove this we should evaluate both the infix expression and the postfix expression and compare the results.

To evaluate the infix expression:

A * B $ C - D + E / F / (G + H)

if values 5 2 3 4 6 1 0 2 are assigned

then

5 * 2 $ 3 - 4 + 6 / 1 / 2
5 * 8 - 4 + 6 / 1 / 2
40 - 4 + 6 / 1 / 2
40 - 4 + 6 / 2
40 - 4 + 3
36 + 3 = 39

26

To evaluate the postfix expression:

A B C $ * D - E F / G H + / +

if values 5 2 3 4 6 1 0 2 are assigned

then

operand stack	operation
5	
5 2	
5 2 3	$
5 8	*
40	
40 4	-
36	
36 6	
36 6 1	/
36 6	
36 6 0	
36 6 0 2	+
36 6 2	/
36 3	+
39	

27

Postfix Evaluation

- Older items on stack are further to left in the expression
- Exactly one item (the answer) is left on stack when done

28

Prefix Evaluation

Method

- *same as before except*
- read from right
- A op B **NOT** B op A
- Older items on stack are further to right in the expression

29

Prefix Evaluation

To prove this we should evaluate both the infix expression and the prefix expression and compare the results.

To evaluate the infix expression:

A * B $ C - D + E / F / (G + H)

if values 5 2 3 4 6 1 0 2 are assigned

then

```
5 * 2 $ 3 - 4 + 6  / 1 / 2
5 *   8   - 4 + 6  / 1 / 2
  40      - 4 + 6  / 1 / 2
  40      - 4 +    6  / 2
  40      - 4 +       3
      36      +       3                    = 39
```

30

To evaluate the prefix expression:

+ - * A $ B C D / / E F + G H

if values 5 2 3 4 6 1 0 2 are assigned
then

	operand stack	operation
	2	
	0 2	+
Top	2	
	6 1 2	/
	6 2	/
	3	
	2 3 4 3	$
	8 4 3	
	5 8 4 3	*
	40 4 3	-
	36 3	+
	39	

31

Convert Expression:
Convert to Postfix

INFIX TO POSTFIX RULES

1) A + B * C
   ↑ CONVERT FIRST

2) A + BC*
   ↑ SINGLE OPERAND

3) A + B - C } PROCESS LEFT
   AB + C - } TO RIGHT

4) A $_1$ B $_2$ C } PROCESS RIGHT
   A B C $_2$ $_1$ } TO LEFT

INFIX TO POSTFIX

A $ B * C - D + E / F / (G + H)

A $ B * C - D + E / F / GH +

AB$ * C - D + E / F / GH +

AB$C* - D + E / F / GH +

AB$C* - D + EF / / GH +

AB$C* - D + EF / GH + /

AB$C*D- + EF / GH + /

AB$C*D- EF / GH + / +

33

Convert Expression:
Convert to Postfix

Method

while input, read expression //from left
if operand, send to output
else //operation
while stacktop has precedence pop
push operator
//no more input
pop stack until empty

34

Convert Expression:
Convert to Postfix

- Older items in stack are further to left in the expression
- Parentheses require special handling. Still within precedence rules
- Treat conceptually as separate problem

35

Convert Expression:
Convert to Postfix

A + B * C

Symbol	Stack	Postfix String
A		A
+	+	A
B	+	AB
*	+ *	AB
C	+ *	ABC
	+	ABC*
		ABC*+

36

Convert Expression:
Convert to Postfix

(A + B) * C

Symbol	Stack	Postfix String
(	(	
A	(	A
+	(+	A
B	(+	AB
)		AB+
*	*	
C	*	AB+C
		AB+C*

37

Convert Expression: Convert to Postfix

A * B $ C - D + E / F / (G + H)

Symbol	Operator Stack	Postfix String
A		A
*	*	A
B	*	AB
$	*$	AB
C	*$	ABC
-	-	ABC$*

note: both items on the stack had precedence over the -

D	-	ABC$*D
+	+	ABC$*D-

note, - has precedence over + when - appears on the left

E	+	ABC$*D-E
$/_1$	$+/_1$	ABC$*D-E
F	$+/_1$	ABC$*D-EF
$/_2$	$+/_2$	ABC$*D-EF$/_1$

38

note: / has precedence over / when / appears on the left

(	$+/_2($	ABC$*D-EF$/_1$

note: (has precedence over every operator except)

G	$+/_2($	ABC$*D-EF$/_1$G
+	$+/_2(+$	ABC$*D-EF$/_1$G

note: any operator has precedence over (when (appears on the left

H	$+/_2(+$	ABC$*D-EF$/_1$GH
)	$+/_2$	ABC$*D-EF$/_1$GH+
	+	ABC$*D-EF$/_1$GH+$/_2$
		ABCS*D-EF$/_1$GH+$/_2$+

39

Convert Expression:
Convert to Prefix

- *same as before except:*
- Read from right
- Older items in stack are further to right

40

Convert Expression:
Convert to Prefix

A + B * C **Infix**
A + * BC **Convert ***
+ A * B C **Convert +**

Conversion - Operators precede Operands
Result treadted as a single operand

41

INFIX TO PREFIX

A $ B * C - D + E / F / (G + H)
A $ B * C - D + E / F / + G H
$ A B * C - D + E / F / + G H
* $ A B C - D + E / F / + G H
* $ A B C - D + / E F / + G H
* $ A B C - D + / / E F + G H
- * $ A B C D + / / E F + G H
+ - * $ A B C D / / E F + G H

42

Convert Expression:
Convert to Prefix

A * B $ C - D + E / F / (G + H)

Symbol	Operator Stack	Prefix String
)	)	
H	)	H
+	)+	H
G	)+	GH
(		+GH
/	/	+GH
F	/	F+GH
/	//	F+GH

Note: positional precedence is satisfied because we are processing the infix string from the right

43

E	//	EF+GH
+	/	/EF+GH
	+	//EF+GH

Note: both items on the stack have precedence over +

D	+	D//EF+GH
-	+-	D//EF+GH

Note: again positional precedence is satisfied because we are processing the infix string from the right

C	+-	CD//EF+GH
$	+-$	CD//EF+GH
B	+-$	BCD//EF+GH
*	+-*	$BCD//EF+GH
A	+-*	A$BCD//EF+GH
	+-	*A$BCD//EF+GH
	+	-*A$BCD//EF+GH
		+-*A$BCD//EF+GH

44

Stack Implementations

- Array Implementation
- Linked Implementation
- Hybrid Implementation

- Overflow vs. Underflow

45

Stack Implementations: Array

☹ Limits size (static)

😐 Requires stack to be homogeneous

☺ Random Access - Exploit!

Data: Need Top Pointer & Array for Data

46

Stack Implementations: Linked Implementation

☺ No size limits (dynamic allocation)

😐 Requires stack to be homogeneous

☹ Sequential Access

Need Top Pointer

47

Stack Code

A simple Linked implementation

```
class SNode {
   Datatype Data;    //any appropriate type
   SNode   Next;
}

public class LStack {
private SNode Top;
   private int size;   //optional size param.
```

48

```
public void LStack() {          //constructor
   Top = null;
   size = 0;                          //if used
}

public boolean Empty() {
   if (Top == null) return true
   else return false;
}
```

49

50

```
public void push ( Datatype Item) {

     SNode Temp = new SNode;
                          //allocate space
     Temp.Data = Item;        //stuff in info
     Temp.Next = Top;
                    //attach to top of stack
     Top = Temp;        //reset Top pointer
  }
```

51

52

```
public Datatype pop () {
   Datatype Item;
   SNode Temp = Top;
   if Empty() "error"       //do appro. error handling
   else {
      Item = Top.Data;                  //Extract info
      Top = Top.Next;              //Reset stack top
      Temp.Next = null;          //disconnect node
      Temp.Data = "a string of blanks";
      return Item; //Deleted value always returned
   }
}
```

53

```
public void StackCopy (SNode S) {
 //a copy constructor...takes existing stack
 //S and makes a copy of it to initiate a
 //new stack.
 //may require modifying interface or ADT
    ...exercise for the student…}
} //end LStack
```

54