DATA STRUCTURES AND APPLICATIONS

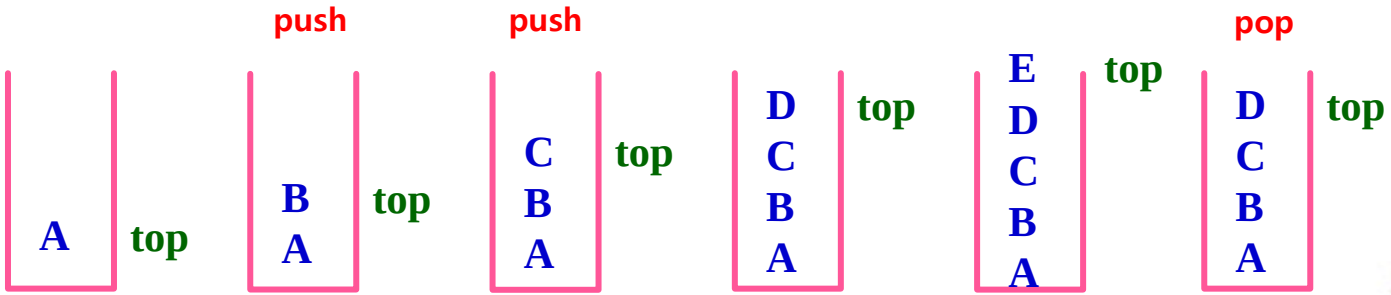
Module 2_1

Stacks and Queues

Dr. Pushpalatha K
Associate Professor
Sahyadri College of Engineering & Management

STACKS

- A stack is an ordered list in which insertions (pushes) and deletions (pops) are made at one end called the top
- Stack is a *LIFO* (last in, first out) data structure
  - New elements are added or pushed onto the top of the stack
  - The first element to be removed or popped is taken from the top

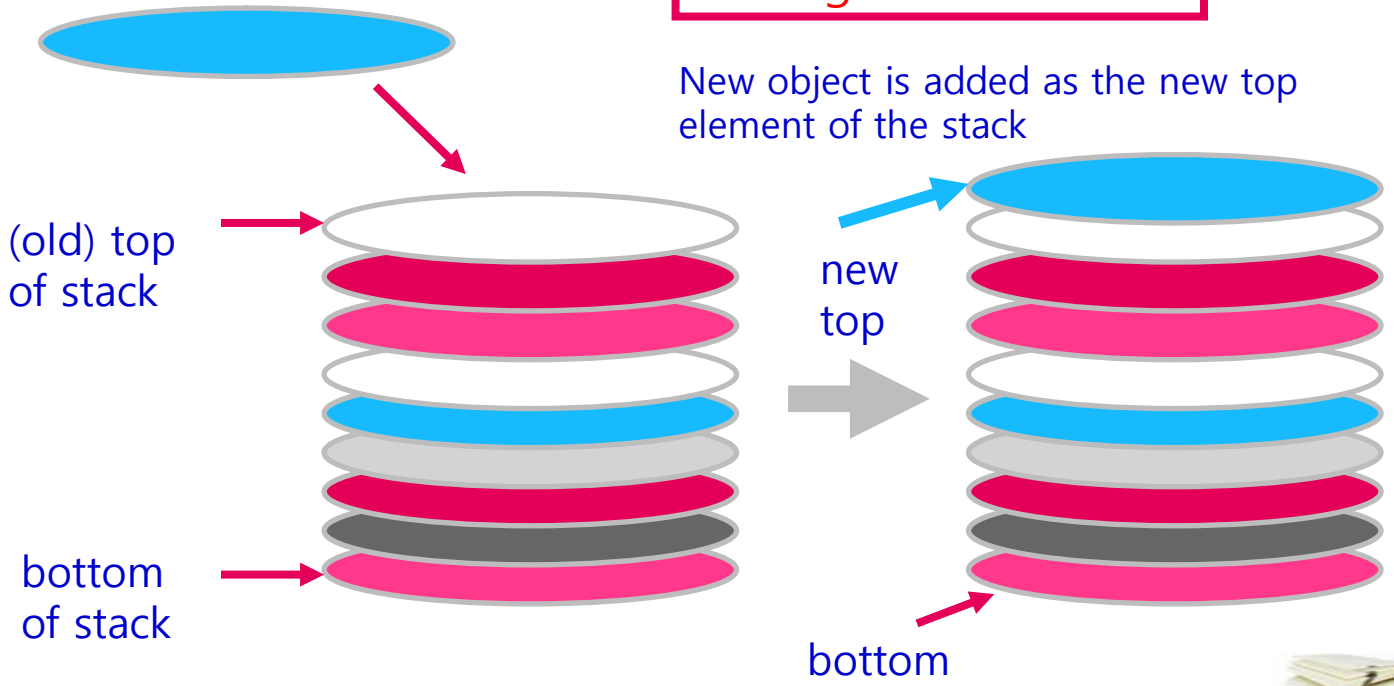


Inserting and deleting elements in a stack

Conceptual View of a Stack

Adding an element

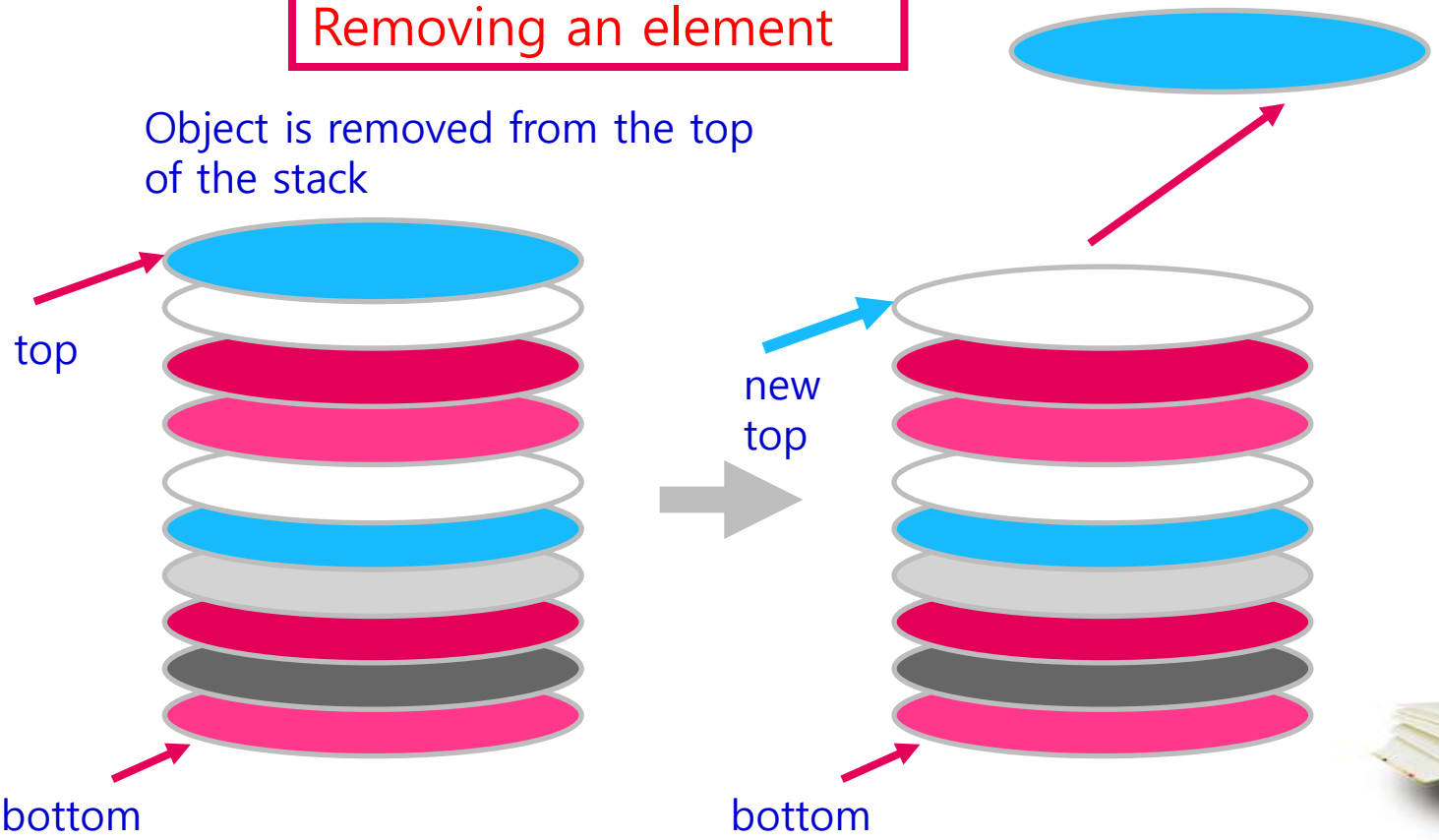
New object is added as the new top element of the stack



Conceptual View of a Stack

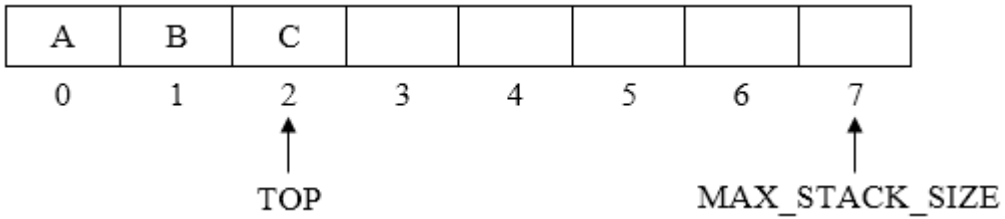
Removing an element

Object is removed from the top of the stack



ARRAY REPRESENTATION OF STACKS

- Stacks may be represented in the computer in various ways such as one-way linked list (Singly linked list) or linear array.
- Stacks are maintained by the two variables such as TOP and MAX_STACK_SIZE.
  - TOP which contains the location of the top element in the stack. If TOP= -1, then it indicates stack is empty.
  - MAX_STACK_SIZE which gives maximum number of elements that can be stored in stack.



Abstract Data Type for Stack

ADT Stack is

objects: a finite ordered list with zero or more elements.

functions:

for all stack ∈ Stack, item ∈ element, max_stack_size ∈ positive integer

Stack CreateS(max_stack_size) ::= create an empty stack whose maximum size is max_stack_size

Boolean IsFull(stack, max_stack_size) ::= if (number of elements in stack == max_stack_size) return TRUE else return FALSE

Stack Push(stack, item) ::= if (IsFull(stack)) stackFull else insert item into top of stack and return

Boolean IsEmpty(stack) ::= if(stack == CreateS(max_stack_size)) return TRUE else return FALSE

Element Pop(stack) ::= if(IsEmpty(stack)) return else remove and return the item on the top of the stack.

Stacks Using Dynamic Arrays

- **Stack using static array**

```
Stack CreateS() ::=
#define MAX_STACK_SIZE 100
typedef struct {
        int key;
        /* other fields */
} element;
element stack[MAX_STACK_SIZE];
int top = -1;
```

- **The element which is used to insert or delete is specified as a structure that consists of only a key field.**

```
Boolean IsEmpty(Stack) ::= top < 0;
Boolean IsFull(Stack) ::= top >= MAX_STACK_SIZE -1;
```

Push operation

- Function push checks whether stack is full.
  - If it is, it calls stackFull( ), which prints an error message and terminates execution.
  - When the stack is not full, increment top and assign item to stack [top].

```
void push(element item)
{/* add an item to the global stack */
   if (top >= MAX_STACK_SIZE - 1)
       StackFull();
  /* add at stack top */
   stack[++top] = item;
}
```

```
void StackFull()
{
fprintf(stderr, "Stack is full,
cannot add element.");
exit(EXIT_FAILURE);
}
```

Pop

- Deleting an element from the stack is called pop operation. The element is deleted only from the top of the stack and only one element is deleted at a time.

```
element pop()
{
  if (top == -1)
     return StackEmpty();
  return stack[top--];
}
```

Stack Using Dynamic Array

- The array is used to implement stack, but the bound (MAX_STACK_SIZE) should be known during compile time.
- The size of bound is impossible to alter during compilation
- This can be overcome by using dynamically allocated array for the elements and then increasing the size of array as needed.

```
Stack CreateS() ::=
typedef struct {
int key;
/* other fields */
} element;
```

```
element *stack;
MALLOC(stack, sizeof(*stack));
int capacity = 1;
int top = -1;
Boolean IsEmpty(Stack) ::= top< 0;
Boolean IsFull(Stack) ::= top >= capacity-1;
```

- **push()**
  - Here the MAX_STACK_SIZE is replaced with capacity

```
void push(element item)
    { /* add an item to the global stack */
    if (top >= capacity-1)
    stackFull();
    stack[++top] = item;
}
```

- **pop( )**
  - In this function, no changes are made.

```
element pop ( )
    { /* delete and return the top element from the stack */
    if (top == -1)
    return stackEmpty(); /* returns an error key */
    return stack[top--];
}
```

stackFull( )

- Increase the capacity of the array stack so that new element can be added into the stack.
- In array doubling, when STACK is full the array capacity is doubled

```
void stackFull()
    {
    REALLOC (stack, 2*capacity*sizeof(*stack));
    capacity *= 2;
    }
```

- In the worst case, the realloc function needs to allocate 2*capacity*sizeof (*stack) bytes of memory and copy capacity *sizeof (*stack)) bytes of memory from the old array into the new one.
- Under the assumptions that memory may be allocated in O(1) time and that a stack element can be copied in O(1) time, the time required by array doubling is O(capacity).
  - Initially, capacity is 1.
- the total run time of push over all n pushes is O(n).

STACK APPLICATIONS: POLISH NOTATION

- **Expressions:**
  - It is sequence of operators and operands that reduces to a single value after evaluation is called an expression.
  - Eg: X = a / b – c + d * e – a * c
    - Above expression contains operators (+, –, /, *) operands (a,b,c,d,e).
- **Expression can be represented in in different format such as**
  - Prefix Expression or Polish notation
  - Infix Expression
  - Postfix Expression or Reverse Polish notation

- **Infix Expression:**
  - In this expression, the binary operator is placed in-between the operand.
  - The expression can be parenthesized or un-parenthesized.
  - Example: A + B - > Here, A & B are operands and + is operand
- **Prefix or Polish Expression:**
  - In this expression, the operator appears before its operand.
  - Ex: + A B -> Here, A & B are operands and + is operand
- **Postfix or Reverse Polish Expression:**
  - In this expression, the operator appears after its operand.
  - Ex: A B + -> Here, A & B are operands and + is operand

Precedence of the operators

- In any programming language, a precedence hierarchy determines the order in which we evaluate operators.
- Operators with highest precedence are evaluated first.
- With right associative operators of the same precedence, we evaluate the operator furthest to the right first.
  - Ex: the multiplicative operators have left-to-right associativity. This means that the expression a * b / c % d / e is equivalent to (((a * b) / c) % d) / e)
- Expressions are always evaluated from the innermost parenthesized expression first.

Token	Operator	Precedence	Associativity
( ) [ ] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement	16	left-to-right
-- ++	decrement, increment	15	right-to-left
!	logical not		
-	one's complement		
- +	unary minus or plus		
& *	address or indirection		
sizeof	size (in bytes)		
(type)	type cast	14	right-to-left
* / %	mutiplicative	13	Left-to-right

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right -to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right -to-left
,	comma	1	left-to-right

INFIX TO POSTFIX CONVERSION

- **An algorithm to convert infix to a postfix expression**
  1. Fully parenthesize the expression.
  2. Move all binary operators so that they replace their corresponding right parentheses.
  3. Delete all parentheses.
- **Ex: Infix expression: a/b -c +d*e -a*c**
  - Fully parenthesized : ((((a/b)-c) + (d*e))-a*c))
  - a b / c – d e * + a c *-

INFIX TO POSTFIX CONVERSION USING STACK

- **Rules**
  - Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.
  - (has low in-stack precedence (isp), and high incoming precedence (icp).
- **Precedence-based postfix()**
  - The left parenthesis is placed in the stack whenever it is found in the expression, but it is unstacked only when its matching right parenthesis is found.
  - An operator can be removed from the stack only if its isp is greater than or equal to the icp of the new operator.

Ex:

a + b * c

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

a * (b +c) * d

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*	match ")"		0	abc+
*	*			0	abc+*
d	*			0	abc+*d
eos	*			0	abc+*d*

Infix and Postfix Notation

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b- c+d))*(e-a)*c	abc-d+/ ea- *c*
a/b- c+d*e- a*c	ab/c- de*+ac*-

Evaluating Postfix Expressions

- **Evaluation process**
  - Make a single left-to-right scan of the expression.
  - Place the operands on a stack until an operator is found.
  - Remove, from the stack, the correct numbers of operands for the operator, perform the operation, and place the result back on the stack.
- **Example: 6 2 / 3 – 4 2 * +**

Token	Stack [0]	[1]	[2]	Top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0