CHAPTER - 20

STACKS

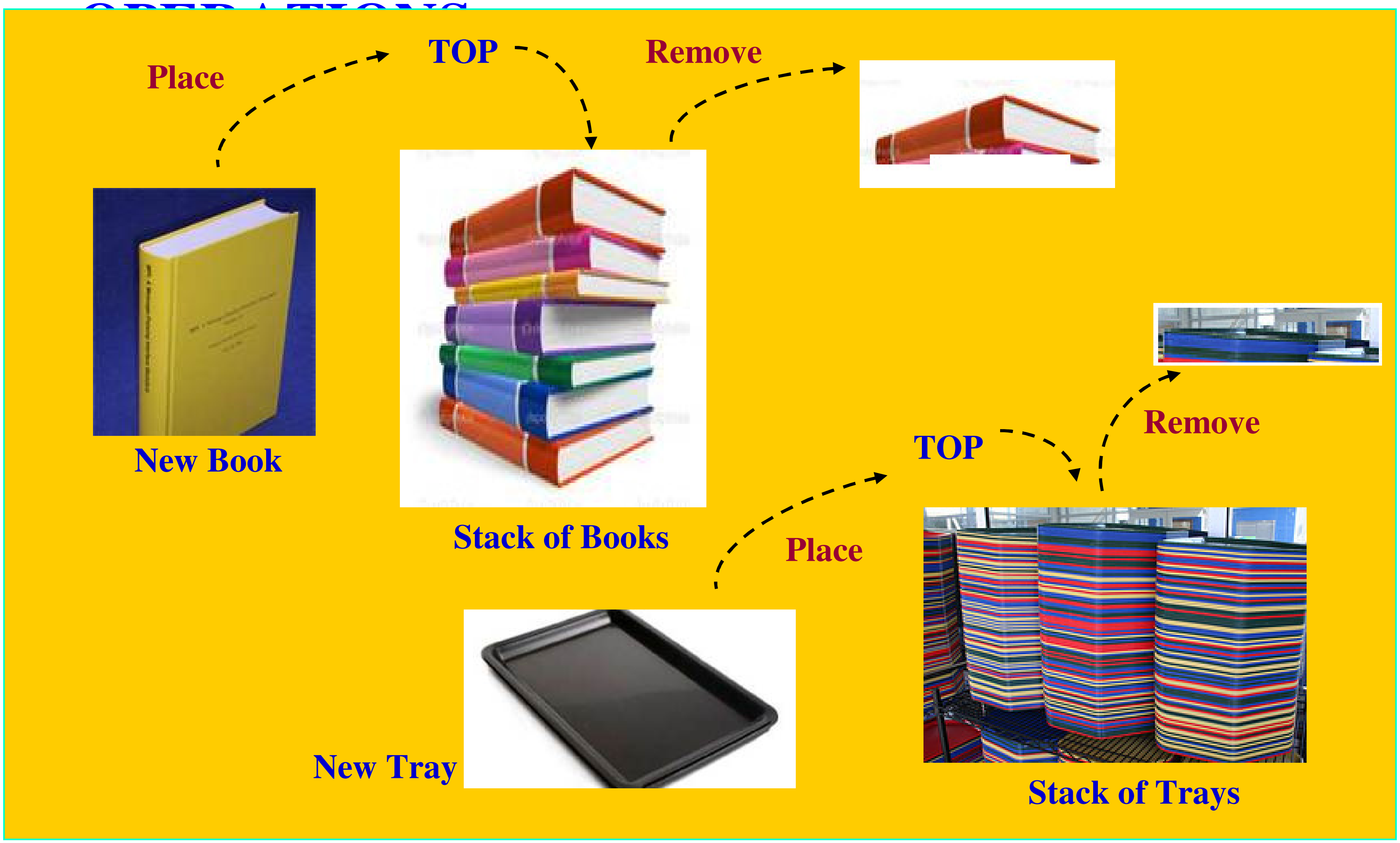
CHAPTER 20

STACKS

INTRODUCTION **(Reading)**
STACK IMPLEMENTATION **(Reading)**
STACK OPERATIONS
PRELIMINARY OPERATIONS ON A STACK
THE STACK AS ABSTRACT DATA TYPE
A STACK IMPLEMENTATION IN C
STACK PROCESS
ALGORITHM TO VALIDATE SCOPES

STACKS

STACK OPERATIONS



Place

TOP

Remove

New Book

Stack of Books

Remove

TOP

Place

New Tray

Stack of Trays

STACKS

PRELIMINARY OPERATIONS ON A STACK

empty condition:

- The starting condition or when all the elements on the stack are removed, the stack will be empty.
- No items can be removed from an empty stack unless more items are added.
- Before removing an element from the stack, it is customary to check whether the stack is empty.
- Empty condition can be checked on a stack with a function *isempty ()*.
- For an array implementation of the stack, the items can be placed from array subscript 0 to a subscript value of MAX – 1. An index will always point to a subscript from where the elements can be removed from the stack.

returns true or false *isempty* ()

overflow condition:

- Because of the array implementation, the array size is fixed at compile time.
- No additional elements can be added to the array more than the size of the array.
- With this implementation the maximum number of elements can stay on the stack are MAX which is defined at compile time.
- Before adding any element to the stack we need to check the overflow condition.
- For removing any element from the stack overflow condition need not be checked.
- Overflow condition is reached while the elements are added to the stack and sufficient number elements are not removed. This condition can be tested on the stack with a function *isoverflow*().

returns true or false *isoverflow* ()

STACKS

THE STACK AS ABSTRACT DATA TYPE

eltype is used to denote the type of the stack element and parameterize the stack type with eltype.

```
    /* value definition */
abstract typedef <<eltype>>  STACK (eltype);

    /* operator definition */
abstract empty (s)
    STACK (eltype) s;
          postcondition  empty == (len (s) == 0);
abstract  eltype pop (s)
    STACK (eltype) s;
    precondition      empty (s) == FALSE;
    postcondition     pop == first (s’);
          s     == sub (s’, 1, len(s’) – 1);
abstract push (s, elt);
    STACK (eltype) s;
    eltype  elt;
    postcondition     s     == <elt> + s’;
```

Primitive operations:

empty

push

pop

Value definition:

Stack (element type)

STACKS

PRELIMINARY OPERATIONS ON A STACK

push operation:

- One of the basic operation performed on a stack is to add elements to the stack.
- Each time the new element will be placed on the top of the element which is already on the stack or this will be first item on the stack.
- When a new item is added to the stack the top is incremented to a new value where the item is placed.
- Before adding any item on to the stack, the overflow condition needs to be tested whether there is place for the new item.
- ***push*** receives two arguments, the element type to be added onto the stack, the value of top of the array.
- ***push*** will increment the value of top after placing the new element onto the stack.

void *push* (stack element type, stack type);

pop operation:

- When elements are removed from the stack, there must be a receiver of the element type.
- We can implement the operation to remove an element from a non-empty stack with a function ***pop*** (stack, top).
- ***pop*** returns the element which is removed from the stack.
- Before removing an element from stack, ***pop*** verifies that there is no empty condition or underflow condition.
- ***pop*** does not cause an overflow condition, there is no need to test. If there was an overflow condition, ***pop*** actually removes the overflow condition.

stack element type se = *pop* (stack, top);

The variable receiving the item from *pop* will be of the type which is returned by *pop*.

STACKS

A STACK IMPLEMENTATION IN C

```
/* preprocessor directives  */
#include<stdio.h>
#include<stdlib.h>
#define MAXSTACK 26
#define EMPTYSTACK   -1
#define TRUE 1
#define FALSE 0

/* function prototypes (signatures)  */
int empty (struct stack *);
char pop (struct stack *);
void push (struct stack *, char);

/* global declarations  */
typedef struct stack {
      int top;
      char stackelement [MAXSTACK];
} CHSTACK;
```

```
/* start of main program logic  */
int main()
{
      CHSTACK  alphstack, *asp;
      char seq, ch, keystroke;
      alphstack.top = EMPTYSTACK;
      asp = &alphstack;

      /* fill up the stack with input from the user */
      for (int i = 0;  i < MAXSTACK;  i++)
      {
               printf ("\n Please enter the next char in seq");
               scanf ("%c%c", &seq, &keystroke);
                push (asp, seq);
      }   /* end of first for loop  */
```

STACKS

A STACK IMPLEMENTATION IN C

```
    /* empty the stack and print values  */
    for (i = 0;  i < MAXSTACK;  i++)
    {
        ch = pop (asp);
        printf ("\n next char in seq is:  %c", ch);
    }
    printf ("\n\n End of character seq **\n");
    return 0;
}   /* end of main routine   */
```

```
/* function verifies whether stack is empty */
 int empty (CHSTACK *ps)
{
        if (ps->top == -1)
           return (TRUE);
        else
           return (FALSE);
}  /* end of empty function */

char pop (CHSTACK *ps)
{
        if (empty (ps) )
        {
            printf (“%s”,  “stack underflow”);
            exit (1);
        }  /* end of empty if */
        return (ps-> stackelement[ps->top--] );
}  /* end of pop function */
```

STACKS

A STACK IMPLEMENTATION IN C

```
void push (CHSTACK *ps, char x)
{
      if (ps->top == MAXSTACK - 1 )
      {
          printf (“%s”,  “stack overflow”);
          exit (1);
      }  /* end of if */
      else
           ps->stackelement[++(ps->top) ] = x;
      return;
}  /* end of push function */
```

```
•     Stacking character sequence ********

•     Please enter the next character in sequence: A
•     Please enter the next character in sequence: B
•     Please enter the next character in sequence: C
•     Please enter the next character in sequence: D
•     Please enter the next character in sequence: E
•     Please enter the next character in sequence: F
•     Please enter the next character in sequence: G
•     Please enter the next character in sequence: H
•     Please enter the next character in sequence: I
•     Please enter the next character in sequence: J
•     Please enter the next character in sequence: K
•     Please enter the next character in sequence: L
•     Please enter the next character in sequence: M
•     Please enter the next character in sequence: N
•     Please enter the next character in sequence: O
•     Please enter the next character in sequence: P
•     Please enter the next character in sequence: Q
•     Please enter the next character in sequence: R
```

STACKS

A STACK IMPLEMENTATION IN C

- Please enter the next character in sequence: S
- Please enter the next character in sequence: T
- Please enter the next character in sequence: U
- Please enter the next character in sequence: V
- Please enter the next character in sequence: W
- Please enter the next character in sequence: X
- Please enter the next character in sequence: Y
- Please enter the next character in sequence: Z

- Popping of character sequence ********

- next character in sequence is: Z
- next character in sequence is: Y
- next character in sequence is: X
- next character in sequence is: W
- next character in sequence is: V
- next character in sequence is: U
- next character in sequence is: T
- next character in sequence is: S
- next character in sequence is: R
- next character in sequence is: Q
- next character in sequence is: P
- next character in sequence is: O
- next character in sequence is: N
- next character in sequence is: M
- next character in sequence is: L
- next character in sequence is: K
- next character in sequence is: J
- next character in sequence is: I
- next character in sequence is: H
- next character in sequence is: G
- next character in sequence is: F
- next character in sequence is: E
- next character in sequence is: D
- next character in sequence is: C
- next character in sequence is: B
- next character in sequence is: A

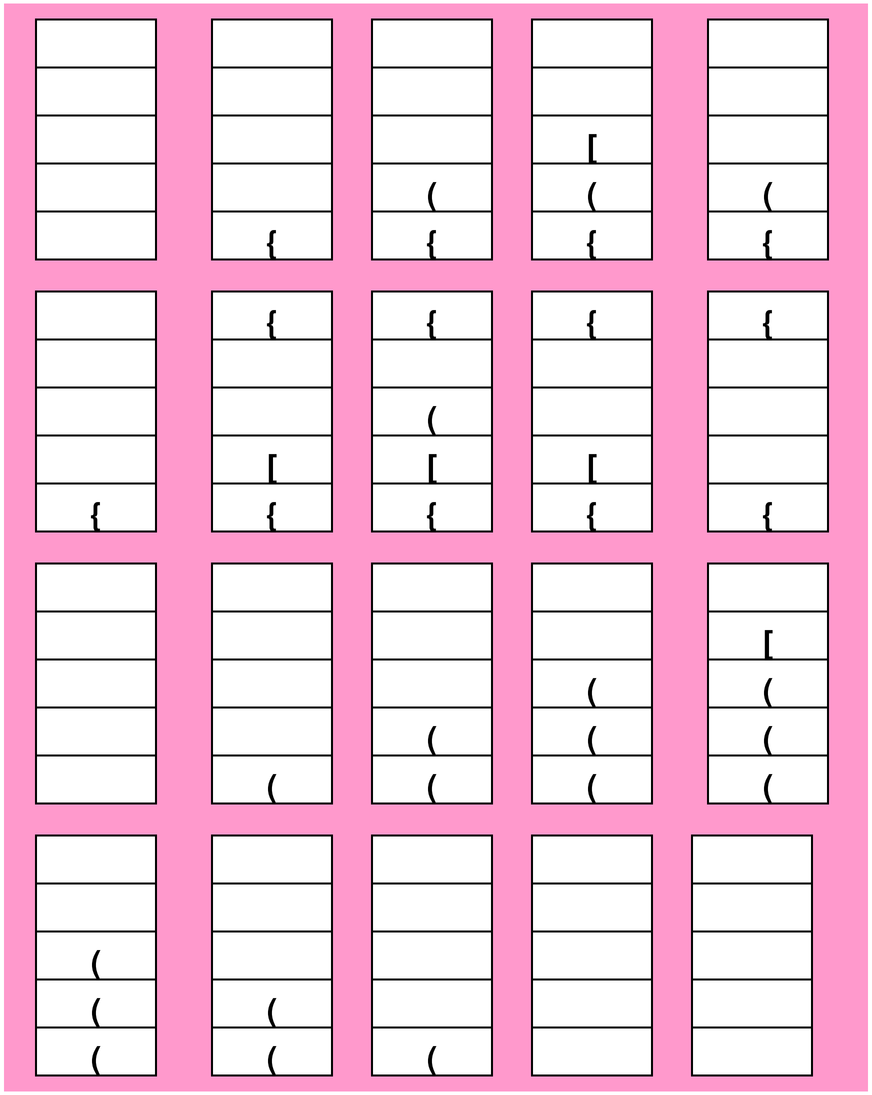
- End of character sequence ********

STACKS

STACK PROCESS

- Stack process can be applied to verify whether the scoping is correctly done in an expression.
- There are three kinds of scoping to evaluate the expression. None of the scopes have any precedence for the operation.
- Normal parenthesis, ‘(‘ and’)’, ‘{‘ and ‘}’, ‘[‘ and’]’ are used in the scoping example.
- Each begin scope should match with the corresponding end scope.
- Finally when all the scope enders are tallied with their corresponding scope beginners, the expression is valid.
- If any of the scope beginner or scope enders are left then the expression becomes invalid.
- Whenever a scope beginner is found in the expression, the scope is pushed on to the stack.
- When a scope ender is found, the stack is popped and the scope beginner is matched with the scope ender.

{x + (y – [a + b]) * c – [(d + e)] } / (h – (j – (k – [l – n])))



STACKS

STACK PROCESS

algorithm to validate to validate scopes:

```
set expression is valid
s = the empty stack;  /* to start with */

Loop as entire string is not read
{
      read the next symbol (symb) of the string;
      Print message about symbol and action
      ignore the operators and operands
      if (symbol is  one of the opening scope )
            place symbol on to the stack);
      if (symbol is  one of the closing scope )
        if (the stack is empty )
            set the expression is not valid
          else
          {
            remove the symbol from the stack;
              if ( the popped symbol is not the
                        matching end scope)
                set the expression is not valid
           }  /* end of not empty else  */
} /* end of loop  */

if (the stack not empty )
   set the expression is not valid;
print whether the expression is valid or not
```