Stacks

D.E ZEGOUR
École Supérieure d'Informatique
ESI

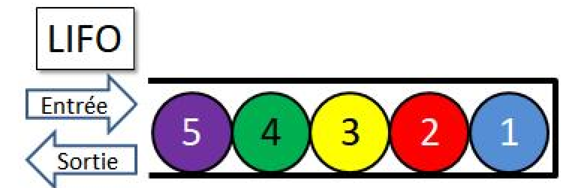
Stacks

Definition, Principle, Application Domains

One of the most widely used concepts in computer science

Collection of elements in which
- any new element is inserted at the end, and
- any element can only be removed from the end.

Principle : LIFO
Last In First Out // last entered, first served



Used in the field of compilation: object scope resolution, recursion, expression evaluation, and more.

Also used for traversing trees and solving a wide range of problems.

Stacks

Abstract Machine

CREATESTACK (S) :

Create an empty stack.

EMPTY_STACK (S) :

Test if stack S is empty.

PUSH (S, Val) :

Push (add to the top) value Val to stack S.

POP (S, Val) :

Pop (remove from the top) a value from stack S and put it in Val.

Stacks

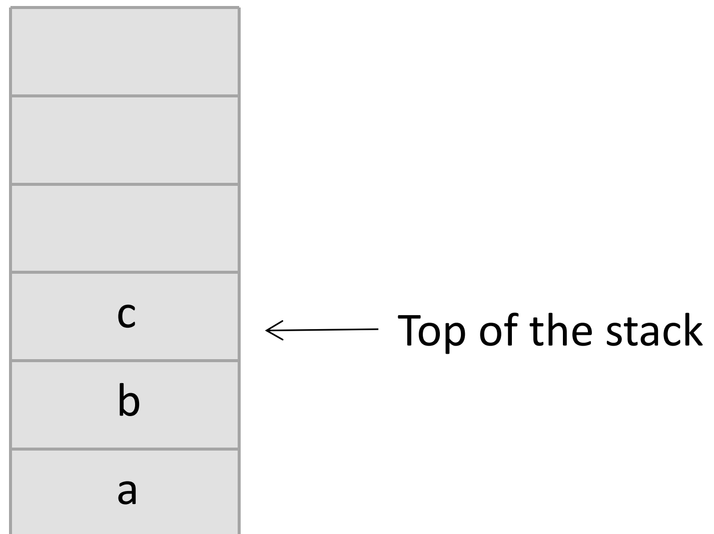
Implementation

Static : using arrays

Dynamic : using linked lists

Stacks

Static Implementation



A stack = (Array+ Integer)

Integer denotes the index of the top

Push operation :
- increment the top
- add the item at the top of the stack

Pop operation :
- remove the item from the top of the stack

Stacks

C Static Implementation

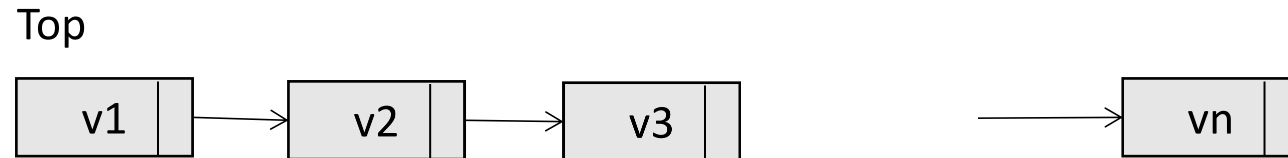
```
#define Max 100
#define True  1
#define False 0
typedef int Bool;
typedef int Anytype;

struct Stack
  {
    int Top ;
    Anytype  Tab[Max];
  };
void Createstack( struct Stack *S)
  {
   (*S).Top = 0 ;
  }
Bool Empty_stack (struct Stack S)
   { return ( S.Top == Max-1 ); }

Bool Full_stack (struct Stack S)
   { return ( S.Top == 0 ); }
```

```
void Push ( struct Stack *S, struct Node *Val)
  {
    if (  !Full_stack(*S)  )
      {
        (*S).Top++;
        (*S).Tab[(*S).Top] = Val ;
     }
    else
      fprintf(" %s", "Overflow \n");
  }
void Pop ( struct Stack *S, struct Node **Val)
  {
    if (  !Empty_stack(*S)  )
      {
       *Val =(*S) .Tab[(*S).Top]  ;
       (*S).Top--;
           }
    else
      fprintf(" %s", "Underflow \n");
  }
```

```
int main(int argc, char *argv[])
 {
   system("PAUSE");
   return 0;
 }
```

Stacks

Dynamic Implementation



As a linked list

The stack is defined by its top : Head of the list

Pushing: add an item at the beginning of the linked list
Popping: remove an item from the beginning of the linked list

Stacks

C Dynamic Implementation

```
#include <stdio.h>
#include <stdlib.h>

typedef int bool ;

#define True 1
#define False 0

/** -Implementation- **\: STACK OF INTEGERS**/
/** Stacks**/

typedef int Typeelem_Si ;
typedef struct Cell_Si * Pointer_Si ;
typedef   Pointer_Si  Typestack_Si  ;

struct Cell_Si
  {
    Typeelem_Si  Val ;
    Pointer_Si Next ;
  } ;
```

```
 void Createstack_Si( Pointer_Si *S )
   { *S = NULL ; }
 bool Empty_stack_Si ( Pointer_Si S )
   { return  (S == NULL ); }

void Push_Si ( Pointer_Si *S,  Typeelem_Si Val )
   {
     Pointer_Si Q;
     Q = (struct Cell_Si *) malloc( sizeof( struct
Cell_Si))   ;
     Q->Val = Val ;
     Q->Next = *S;
     *S = Q;
   }
```

```
 void Pop_Si ( Pointer_Si *S,  Typeelem_Si
*Val )
   {
     Pointer_Si Save;

     if (! Empty_stack_Si (*S) )
       {
         *Val = (*S)->Val ;
         Save = *S;
         *S = (*S)->Next;
         free(Save);
       }
     else printf ("%s \n", "Stack is empty");
}
 /** Variables of main program **/
 Pointer_Si S=NULL;

 int main(int argc, char *argv[])
   {
     system("PAUSE");
     return 0;
}
```

Stacks

Application : Arithmetic Expression Evaluation

We want to evalute the following expressions :
(a +b) * (c-d)
(a+b+c) / (a+b-c)
(-(a+b) * (c-d)) + (a*b)

Problems:
- How many temporary variables should we use?
- How to manage them?

Algorithmic solution : difficult

Turn to a compilation technique that proceeds in two steps:
- Transform the expression into a Postfix notation (in the semantic phase of the compilation process)
- Use a stack to evaluate it

(a +b) * (c-d) ➡ ab+cd-*

(a+b+c) / (a+b-c) ➡ ab+c+ab+c-/

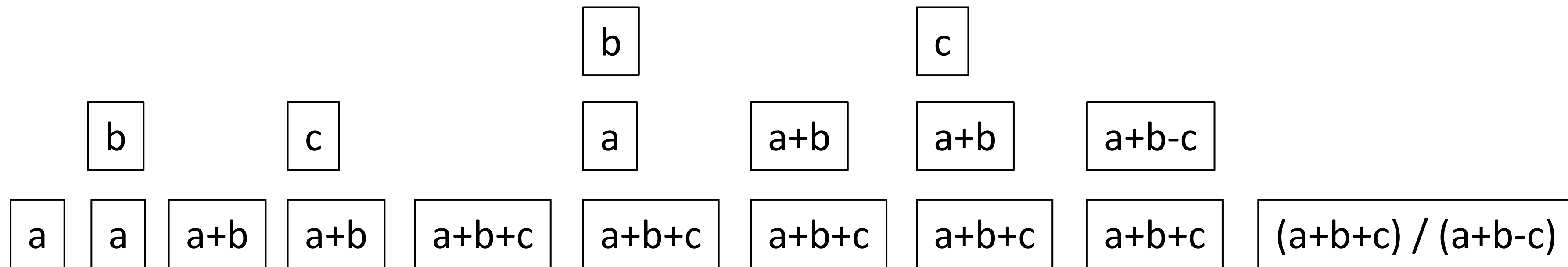
(-(a+b) * (c-d)) + (a*b) ➡ ab+!cd-*ab*+

Postfix Polish Notation

Stacks

Application : Arithmetic Expression Evaluation

Evaluate a b + c + a b + c - /



Evolution of the stack

Stacks

Application : Arithmetic Expression Evaluation

The expression is a string terminated by '#'

The string is read character by character

A character is either an operand (variable name) or an operator

Val (C) : value corresponding to the variable C

Operande(C) : boolean function returning True if C is an operand, False otherwise

Binary (C) : boolean function returning True if C is an operator, false otherwise

Oper(C, V1, V2) : integer function returning the result of the operation C
on variables V et V2

Oper2(C, V) : integer function returning the result of the operation C on variable V

```
I := 1 ;
  C := CHARACT ( Expression , I ) ;
  CREATESTACK ( P ) ;
  WH C <> '#'
     IF Operand ( C )
        PUSH ( P , Val ( C ) )
     ELSE
        IF Binary ( C )
           POP ( P , V ) ;
           POP ( P , V2 ) ;
           PUSH ( P , Oper ( C , V , V2 ) )
        ELSE
           POP ( P , V ) ;
           PUSH ( P , Oper2 ( C , V ) )
        ENDIF
     ENDIF ;
     I := I + 1 ;
     C := CHARACT ( Expression , I ) ;
  EWH ;
  POP ( P , V ) ;
  Eval := V
```