Stack

Unit-2

Presented By: Dr. Aparna Kumari

Array representation of stacks

- In computer's memory stacks can be represented as a linear array.
- Every stack has a variable TOP associated with it.
- TOP is used to store the address of the topmost element of the stack. It is this position from where the element will be added or deleted.
- There is another variable MAX which will be used to store the maximum number of elements that the stack can hold.
- If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX -1, then the stack is full.

Push Operation

- Procedure PUSH(S, TOP, X): This procedure inserts an element X on the top of a stack which is represented by a vector S containing N elements with a pointer TOP denoting the top element in the stack.

```
Algorithm to PUSH an element in a stack

Step 1: IF TOP = N-1, then
              PRINT "OVERFLOW"
              Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET S[TOP] = X
Step 4: END
```

Pop Operation

- Procedure POP(S, TOP): This procedure removes the element from the stack which is represented by a vector S and returns the element.

```
Algorithm to POP an element from a stack

Step 1: IF TOP = -1, then
           PRINT "UNDERFLOW"
           Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP - 1
Step 3: Return S(TOP+1)
Step 4: END
```

Applications of Stack

- Expression conversion
- Expression evaluation
- Recursion

Expressions

- Expressions is a string of operands and operators. Operands are some numeric values and operators are of two types: Unary and binary operators. Unary operators are '+' and '-' and binary operators are '+', '-', '*', '/' and exponential. In general, there are three types of expressions:
  - Infix expression : operand1 operator operand2
  - Postfix expression : operand1 operand2 operator
  - Prefix expression : operator operand1 operand2

Infix	Postfix	Prefix
(a + b)	ab +	+ ab
(a + b) * (c - d)	ab + cd - *	* + ab - cd
(a + b / e) * (d + f)	abe /+ df + *	* + a/be + df

Polish notation

- It is **a notation form for expressing arithmetic, logic and algebraic equations**.
- Here, operators are placed on the left of their operands.
- If the operator has a defined fixed number of operands, the syntax does not require brackets or parenthesis to lessen ambiguity.
- For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Reversed Polish Notation

- This notation style is known as **Reversed Polish Notation**.
- In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands.
- For example, **ab+**. This is equivalent to its infix notation **a + b**.

Expression Evaluation

- Evaluation of postfix expression
- Evaluation of prefix expression
- Evaluation of infix expression

Evaluation of postfix expression

- Step 1: If char read from postfix expression is an operand, push operand to stack.
- Step 2: If char read from postfix expression is an operator, pop the first 2 operand in stack and implement the expression using the following operations:
  - **pop(opr1) then pop(opr2)**
  - **result = opr2 operator opr1**
- Step 3: Push the result of the evaluation to stack.
- Step 4: Repeat steps 1 to steps 3 until end of postfix expression
- Finally, At the end of the operation, only one value left in the stack. The value is the result of postfix evaluation.

Evaluation of prefix expression

- Step 1: Reverse the prefix expression.
- Step 2: Read this reversed prefix expression from left to right one character at a time.
- Step 3: If char read from reversed prefix expression is an operand, push operand to stack.
- Step 4: If char read from reversed prefix expression is an operator, pop the first 2 operand in stack and implement the expression using the following operations:
  - **pop(opr1) then pop(opr2)**
  - **result = opr1 operator opr2**
- Step 5: Push the result of the evaluation to stack.
- Step 6: Repeat steps 3 to steps 5 until end of reversed prefix expression
- Finally, At the end of the operation, only one value left in the stack. The value is the result of prefix evaluation.

Recursion

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {
   recursion(); /* function calls itself */
}

int main() {
   recursion();
}
```

Recursion

Factorial_I
Input: An integer number *N*.
Output: The factoral value of *N*, that is *N*!.
Remark: Code using the iterative definition of factorial.

Steps:

```
1. fact =1
2. For (i = 1 to N) do
3.     fact = i * fact
4. EndFor
5. Return(fact)                                  // Return the result
6. Stop
```

Here, Step 2 defines the iterative definition for the calculation of a factorial. Now, let us see the recursive definition of the same.

Factorial_R //Code using the recursive definition of factorial
Input: An integer number *N*.
Output: The factoral value of *N*, that is *N*!.
Remark: Code using the recursive definition of factorial.

Steps:

```
1. If (N = 0) then                  // Termination condition of repetition
2.     fact = 1
3. Else
4.     fact = N * Factorial_R(N − 1)
5. EndIf
6. Return(fact)                              // Return the result
7. Stop
```

Here, Step 4 recursively defines the factorial of an integer *N*. This is actually a direct translation of $n! = n * (n - 1)!$ in the form of $Factorial\_R\ (n) = n * Factorial\_R\ (n - 1)$. This definition implies that $n!$ will be calculated if $(n - 1)!$ is known, which in turn if $(n - 2)!$ is known and so on until $n = 0$ when it returns 1 (Step 1).

Recursion

Steps:

```
1.  5! = 5*4!
2.          4! = 4*3!
3.                  3! = 3*2!
4.                          2! = 2*1!
5.                                  1! = 1*0!
6.                                          0! = 1
7.                                  1! = 1
8.                          2! = 2
9.                  3! = 6
10.         4! = 24
11. 5! = 120
```

- Here, it is required to push the intermediate calculations till the terminal condition is reached. In the above calculation for 5!, Steps 1 to 6 are the push operations. The subsequent pop operations will evaluate the value of intermediate calculations till the stack is exhausted.
- Stack for parameter(s): To store the parameter with which the recursion is defined.
- Stack for local variable(s): To hold the local variable that are used within the definition.
- Stack to store the return address.

Factorial with recursion using stack

```c
#include<stdio.h>
int stk[100]; // stack
int size = 100;    // size of stack
int Top = -1; // store the index of top element of the stack
// push x to stack
void push(int x){
    if(Top==size-1){
        printf("OverFlow \n");
    }
    else{
        ++Top;
        stk[Top] = x;
    }
}
// return top element of the stack
int top(){
    if(Top==-1){
        printf("UnderFlow \n");
        return -1;
    }
    else{
        return stk[Top];
    }
}
// remove top element from the stack
void pop(){

    if(Top==-1){
        printf("UnderFlow \n");
    }
    else{
        --Top;
    }
}
```

Factorial with recursion using stack

```
25  // remove top element from the stack
26  void pop(){
27
28      if(Top==-1){
29          printf("UnderFlow \n");
30      }
31      else{
32          --Top;
33      }
34  }
35  // check if stack is empty or not
36  int isempty(){
37      if(Top==-1)
38          return 1;
39      else
40          return 0;
41  }
42  int main() {
43      int i, n;
44
45      printf("Enter a number: ");
46      scanf("%d", &n);
47
48      push(1);
49      for(i=2;i<=n;++i){
50          push(top() * i);
51      }
52      printf("Factorial: %d", top());
53      return 0;
54  }
```

Tower of Hanoi Problem

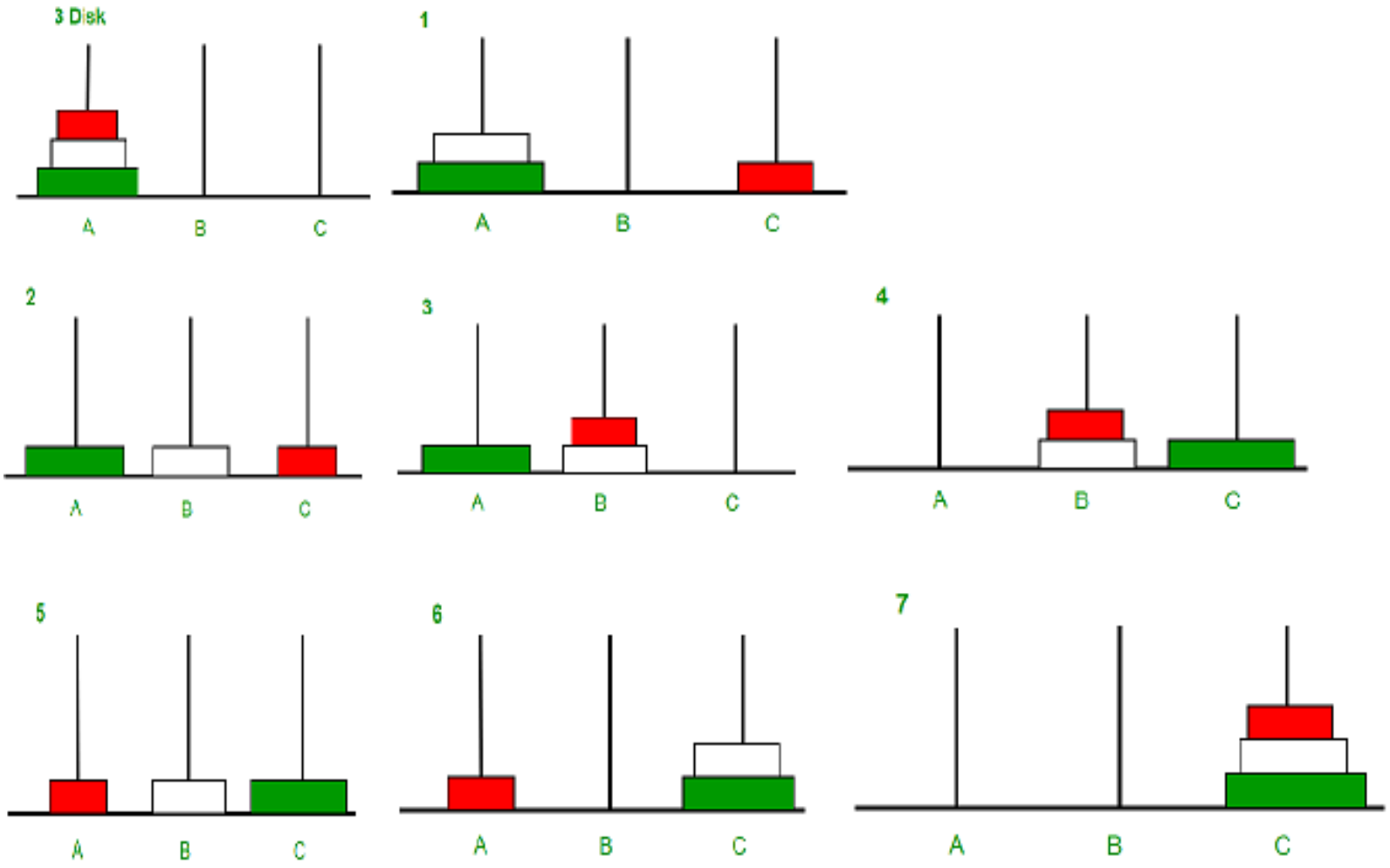
- The Tower of Hanoi puzzle is solved by moving all the disks to another tower by not violating the sequence of the arrangements.

The rules to be followed by the Tower of Hanoi are -

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

- **Algorithm**
- Step 1 − Move n-1 disks from source to aux
- Step 2 − Move nth disk from source to dest
- Step 3 − Move n-1 disks from aux to dest

Example



Example

```
void Tower_of_Hanoi(int n,char A,char B,char C)
{
   if(n>0) {
      TOH(n-1,A,C,B);
       printf(A to B);
      TOH(n-1,C,B,A);
   }
}
```

Continue…

- Input: n=3
- Output:

A to B

A to C

B to C

A to B

C to A

C to B

A to B

THANK YOU!!

ANY QUESTIONS??