CS2x1:Data Structures and Algorithms

Koteswararao Kondepu

k.kondepu@iitdh.ac.in

Recap

- Data type
- Data Structures
- Classifications of Data Structures
  - Primitive data structures → basic data structures
  - Non-Primitive data structures → complicated data structures
    - Linear data structures
    - Non-Linear data structures
- Abstract Data Structures
  - Stack (LIFO)
    - push/add/insert
    - pop/remove/delete
    - top/peek
    - overflow
    - underflow

Outline

- Exercises on Stack
- Implementation of Stack
- Applications of Stack

Exercise#1: Data Structure

Select the following correct options which define “Data Structure”!

a. Data structure is a special format for organizing and storing data

b. Data structure is used to denote a particular way of organizing data for particular type of operations

c. Data structure is a data organization, management, and storage format that enables efficient access and modification

d. None of the above

Exercise#2: Stack

- What is the output of the program for the following input?

  4 3 2 * 6 * +

Rules: (i) Read the element from Left to Right

(ii) If it is an operand push it into stack

(iii) if it is an operator →

- pop top 2 elements
- apply the operators on the popped elements
- push the results on to the stack

(iv) Pop → Results

Exercise#3: Stack

- What is the output of the program for the following input?

  4 3 2 * 6 * -

Rules: (i) Read the element from Left to Right

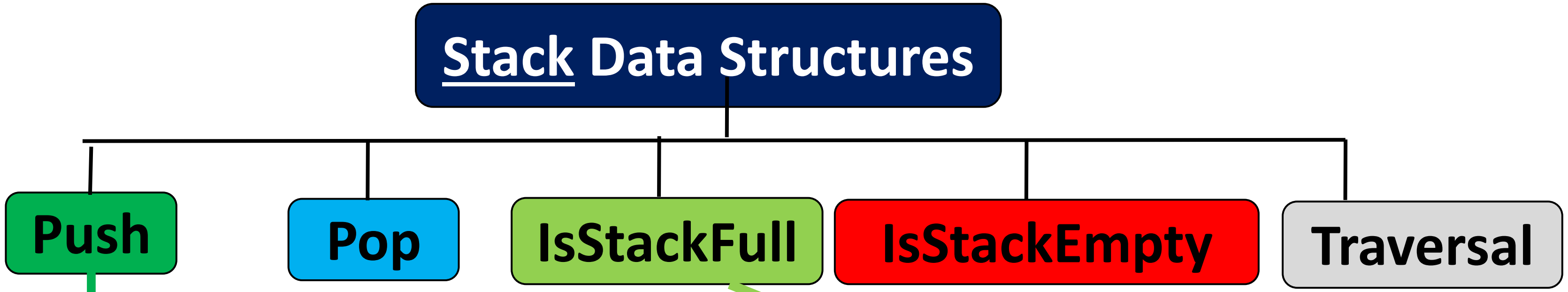
(ii) If it is an operand push it into stack

(iii) if it is an operator →

- pop top 2 elements
- apply the operators on the popped elements
- push the results on to the stack

(iv) Pop → Results

Implementation: Push

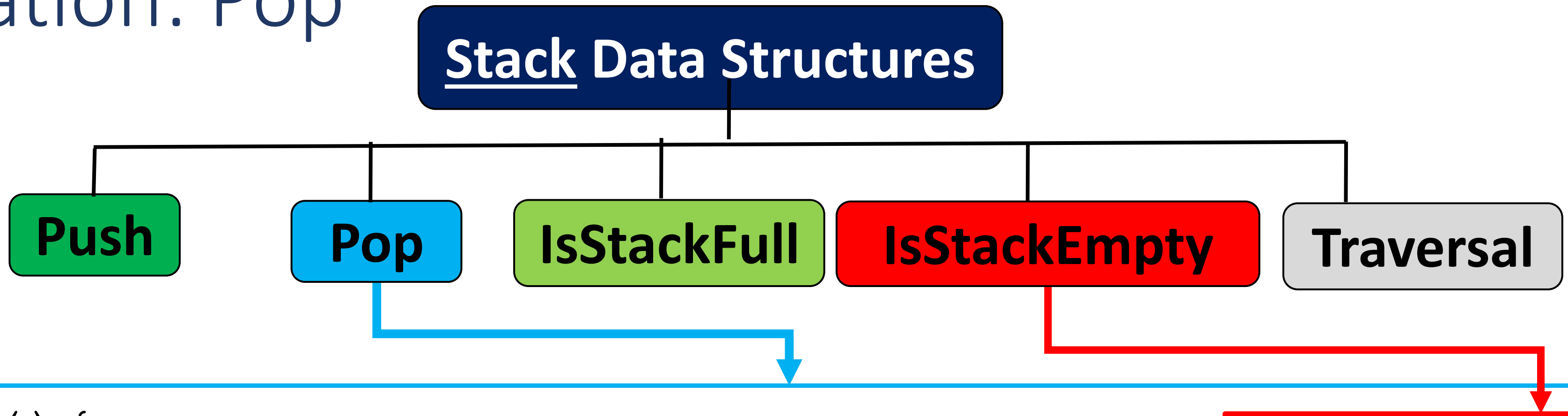


```
  void Push(){
    int Element;
1   if(!IsStackFull()){

2      printf("Enter element\n");
       scanf("%d", &Element);
3      Top++;
4      Stack[Top] = Element;
  }
    else
5   printf("Element cannot be pushed as stack is already
  full \n");}
```

```
int IsStackFull(){
  if(Top == StackSize-1){
    return 1;
  }
  else
    return 0;
}
```

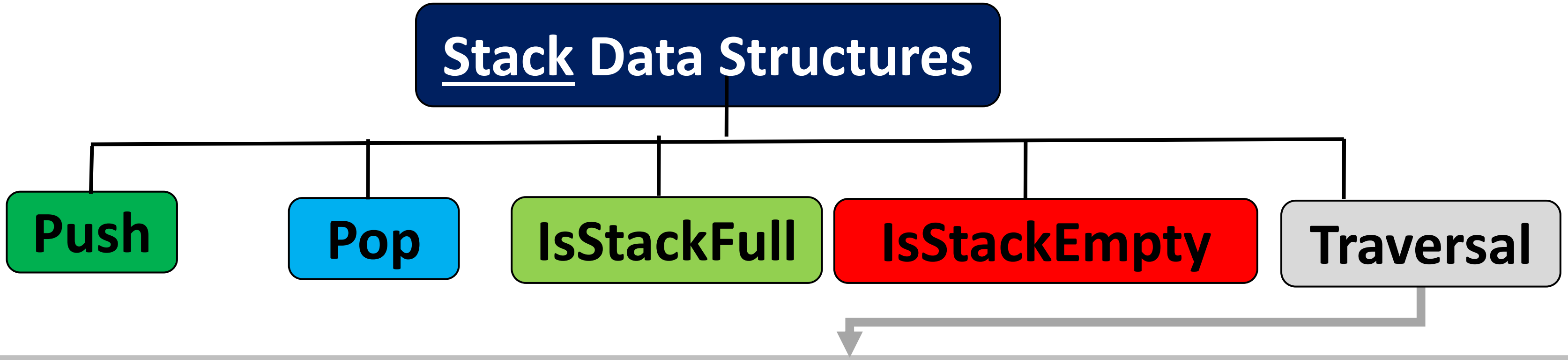
Implementation: Pop



```
void Pop(){
  if(!IsStackEmpty()){
    printf("Popped out element is: %d \n",
    Stack[Top]);
    //Stack[Top] = -1; ******
    Top--;
  }
else
printf("Stack is already empty, we cannot pop the element
from it\n");
}
```

```
int IsStackEmpty
(){
  if(Top == -1){
    return 1;
  }
  else
    return 0;}
```

Implementation: Traversal



```
void PrintStack(){
  int i=0;

  for(i=0; i<StackSize; i++){
    printf("Stack[%d] = %d \n", i, Stack[i]);
  }
   }
```

//Top ********

Stack Application: Recursion

- *Recursion:* (i) Any function which calls itself is called *recursive.*
  (ii) Recursion *terminates* → *we need to make sure*
  (iii) The small-small recursive functions should be convergence
  (iv) The code is shorter

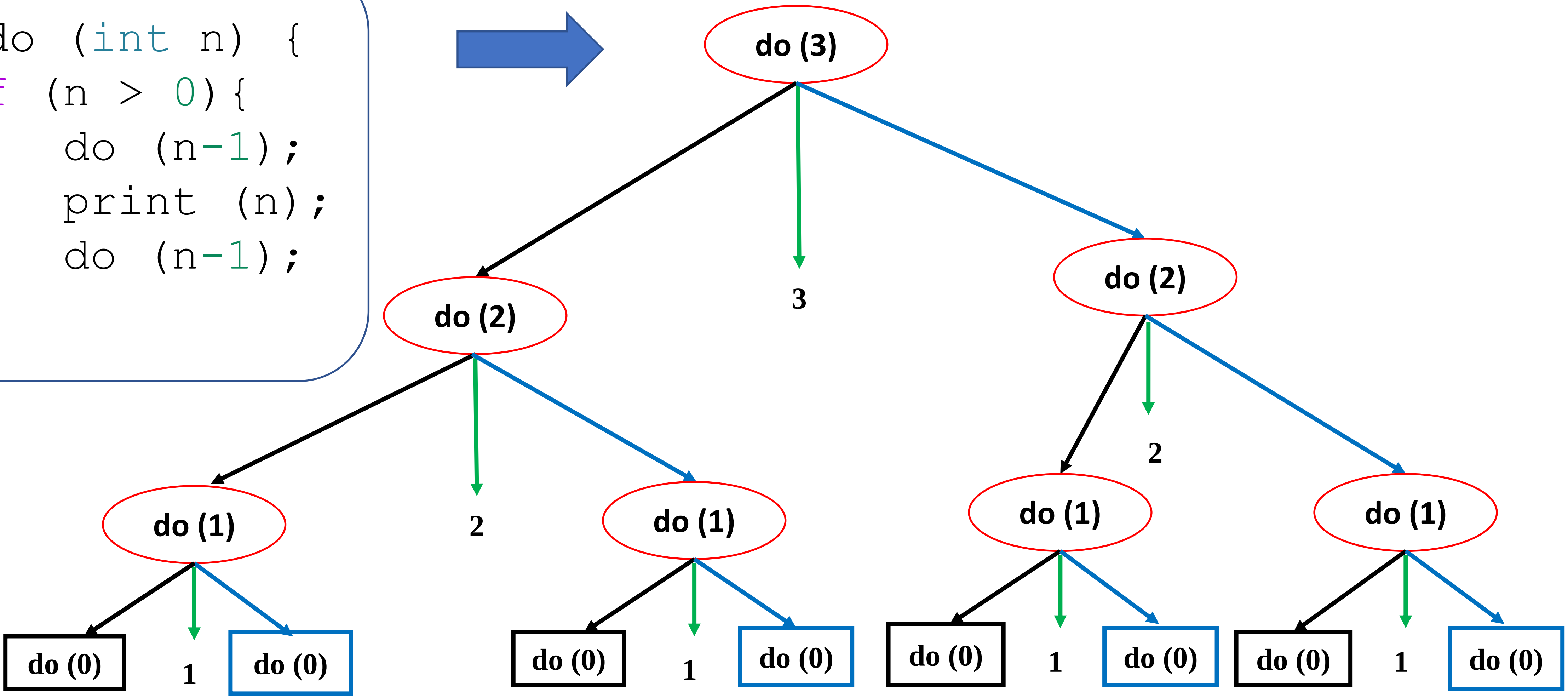
- Base case
- Sub task
- Recursive case

```
//Calculate the factorial of a positive integer
int Fact(int n){
    if (n == 1) // base case: fact of 0 or 1
       return 1;
    else if (n == 0)
       return 1;
    else  //recursive case: multiply n by (n-
1) factorial
      return n*Fact(n-1);
}
```

3	**1*Fact(1-1)**
2	2*Fact(2-1)
1	3*Fact(3-1)
0	4*Fact(4-1)

Stack Application: Recursion (1)

```
void do (int n) {
      if (n > 0){
            do (n-1);
            print (n);
            do (n-1);
      }
}
```



Stack Applications: Arithmetic expression evaluation

- Arithmetic expression → Consists of operands and operators.
- Arithmetic notation → Arrangement of operators and operands to write the arithmetic expression
  - Prefix expression (or Polish expression) → The operators in the expression are placed before the operands on which the operator works. e.g: a+b*c → +a*bc
  - Infix expression → The operators in the expression are placed in between the operands on which the operator works. e.g: a+b*c
  - <u>Postfix</u> <u>expression</u> → The operators in the expression are placed after the operands on which the operator works. e.g: a+b*c → abc*+

Stack Applications: Arithmetic expression evaluation

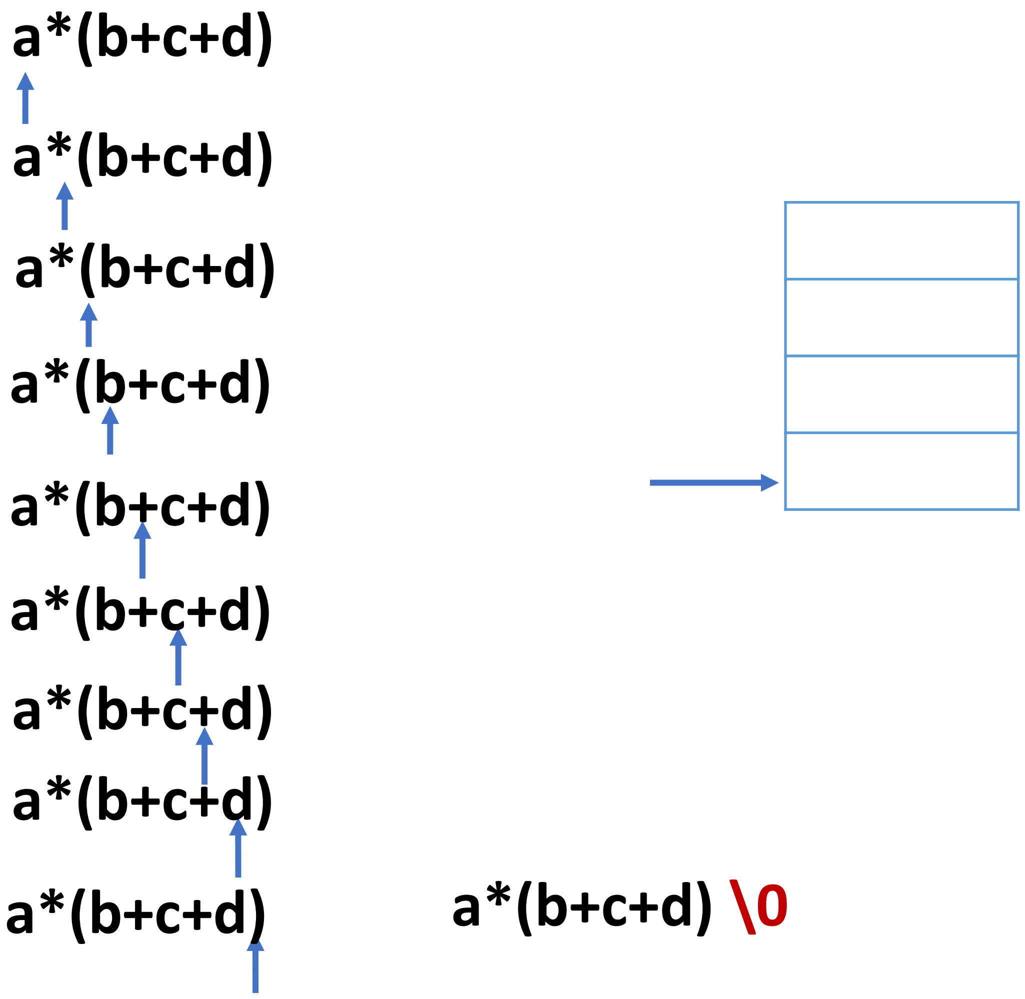
Precedence	***Operators***	***Associativity***
1	() [] -> . ++ --	Left to Right
2	+ – ! ~ ++ — (type)* & sizeof()	Right to Left
3	* / %	Left to Right
4	+ –	Left to Right
5	<<, >>	Left to Right
6	< <= > >=	Left to Right
7	== !=	Left to Right
8	&	Left to Right
9	^	Left to Right
10	|	Left to Right
11	&&	Left to Right
12	| |	Left to Right
13	?:	Right to Left
14	= += -+ *= /= %= >>= <<= &= ^= |=	Right to Left

Stack Applications: Arithmetic expression evaluation (1)

- Convert the following infix expression into the postfix expression

Infix expression: a*(b+c+d)

Infix **Stack** **Postfix**



a*(b+c+d)

a*(b+c+d)

a*(b+c+d)

a*(b+c+d)

a*(b+c+d)

a*(b+c+d)

a*(b+c+d)

a*(b+c+d)

a*(b+c+d)

a*(b+c+d) \0

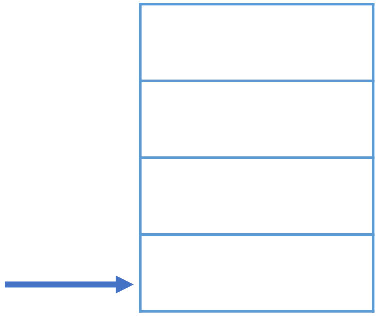
Stack Applications: Arithmetic expression evaluation (2)

- Convert the following infix expression into the postfix expression

Infix expression: x+y-z+(s^t)*u/v

Infix **Stack** **Postfix**

x+y-z+(s^t)*u/v



thank you!

email:
k.kondepu@iitdh.ac.in

NEXT Class: 24/04/2023