PES UNIVERSITY

Data Structures

Dilip Maripuri
Associate Professor
Department of Computer Applications

Data Structures

Session Outline

PES UNIVERSITY

Stacks

Applications

PES UNIVERSITY

- **Reversing Strings**
  - **How It Works**
    - To reverse a string using a stack, each character of the string is pushed onto the stack.
    - When popping the characters off the stack, they come out in reverse order.
  - **Illustration**
    - For the string "HELLO", pushing each character onto the stack and then popping them out will result in "OLLEH".

Stacks – Applications

- **Parsing Expressions**
  - **Usage**
    - Stacks are crucial in evaluating and parsing expressions, especially in implementing infix, prefix, and postfix expression evaluators.
  - **Example**
    - In infix to postfix conversion, operators are pushed to a stack and the precedence is managed to format the expression correctly.

Stacks – Real World Applications

- **Undo Mechanism in Software**
  - **How It's Used**
    - Actions performed by a user are pushed onto a stack.
    - When the undo function is activated, actions are popped from the stack, effectively reversing the most recent actions.
  - **Example**
    - In text editing software, each edit (like insert, delete, format) can be stored in a stack, allowing users to undo them in the reverse order.

Stacks – Real World Applications

- **Backtracking in Browser History**
  - **Implementation**
    - Web browsers use stacks to manage the history of visited pages.
    - Each new page visited is pushed onto the stack, and the back button pops from this stack.

Stacks – Designing Complex Data Structures

PES UNIVERSITY

- **Double Stacks**
  - **Concept**
    - Double stacks use a single array to implement two stacks.
    - They start from opposite ends of the array and grow towards each other.
  - **Illustration**
    - Imagine an array from index 0 to 9.
    - Stack one starts at index 0, and stack two starts at index 9.
    - They grow towards each other as elements are added.

Stacks – Designing Complex Data Structures


```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 10

typedef struct {
    int arr[MAX_SIZE];
    int top1;
    int top2;
} *DoubleStack;

// Function to create a new double stack
DoubleStack createDoubleStack() {
    DoubleStack ds = (DoubleStack)malloc(sizeof(*ds));
    ds->top1 = -1;
    ds->top2 = MAX_SIZE;
    return ds;
}
```

Stacks – Designing Complex Data Structures


```
// Generalized push function
void push(DoubleStack ds, int stackNumber, int value) {
    if (ds->top1 < ds->top2 - 1){
        if (stackNumber == 1)
            ds->arr[++ds->top1] = value;
        else if (stackNumber == 2)
            ds->arr[--ds->top2] = value;  }
        else
            printf("Stack Overflow\n");
}
```

Stacks – Designing Complex Data Structures

PES UNIVERSITY

```
// Generalized pop function
int pop(DoubleStack ds, int stackNumber) {
    if (stackNumber == 1 && ds->top1 >= 0)
        return ds->arr[ds->top1--];
    else if (stackNumber == 2 && ds->top2 < MAX_SIZE)
        return ds->arr[ds->top2++];
    else
    {
        printf("Stack Underflow\n");
        return -1;
    }
}
```

Stacks – Designing Complex Data Structures

PES UNIVERSITY

```c
int main() {
    DoubleStack ds = createDoubleStack();

    // Example usage
    push(ds, 1, 1);
    push(ds, 2, 10);
    printf("Popped from stack1: %d\n", pop(ds, 1));
    printf("Popped from stack2: %d\n", pop(ds, 2));

    // Free memory
    free(ds);
    return 0;
}
```

Stacks – Designing Complex Data Structures

PES UNIVERSITY

- **Min/Max Stacks**
  - **Usage**
    - These stacks are designed to keep track of the minimum or maximum element in the stack in constant time.
  - **Implementation**
    - Along with the regular stack operations, a secondary stack is maintained that keeps track of the current minimum or maximum.
  - **Illustration**
    - If the stack elements are [3, 2, 5, 1], the min stack would be [3, 2, 2, 1], always holding the minimum element at the top.

Stacks – Designing Complex Data Structures


```c
typedef struct {
    int top;
    int data[MAX_SIZE];
} Stack;

void initialize(Stack *s)
{       s->top = -1;    }

int isFull(Stack *s)
{       return (s->top == MAX_SIZE - 1);        }

int isEmpty(Stack *s)
{    return s->top == -1;       }
```

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
```

Stacks – Designing Complex Data Structures


```
void push(Stack *s, int value) {
    if (!isFull(s))
        s->data[++s->top] = value;
    else
        printf("Stack Overflow\n");
}

int pop(Stack *s) {
    if (!isEmpty(s))
        return s->data[s->top--];
    else {
        printf("Stack Underflow\n");
        return -1;
    }
}
```

Stacks – Designing Complex Data Structures

PES UNIVERSITY

```
int peek(Stack *s) {
    if (!isEmpty(s))
        return s->data[s->top];
    else {
        printf("Stack is empty\n");
        return -1;
    }
}
```

```
typedef struct
{       Stack mainStack, Stack minStack; } MinStack;
```

Stacks – Designing Complex Data Structures


```
void initializeMinStack(MinStack *ms)
{   initialize(&ms->mainStack);
    initialize(&ms->minStack);          }

void pushMinStack(MinStack *ms, int value) {
    push(&ms->mainStack, value);
    if (isEmpty(&ms->minStack) || value <= peek(&ms->minStack))
        push(&ms->minStack, value);
}
```

Stacks – Designing Complex Data Structures

PES UNIVERSITY

```
int popMinStack(MinStack *ms) {
    int value = pop(&ms->mainStack);
    if (value == peek(&ms->minStack))
        pop(&ms->minStack);
    return value;
}

int getMin(MinStack *ms) {
    return peek(&ms->minStack);
}
```

Stacks – Designing Complex Data Structures

PES UNIVERSITY

```c
int main() {
    MinStack ms;
    initializeMinStack(&ms);

    // Example usage
    pushMinStack(&ms, 3);
    pushMinStack(&ms, 2);
    pushMinStack(&ms, 5);
    pushMinStack(&ms, 1);
    printf("Minimum value: %d\n", getMin(&ms));

    popMinStack(&ms);
    printf("Minimum value after popping: %d\n", getMin(&ms));

    return 0;
}
```

Call Stack Mechanics in Recursion

PES UNIVERSITY

- **Stack Frame in Function Calls:**
  - When a function is called, the computer must remember the place it needs to return to once the function is complete. For this, it uses a call stack.
  - A stack frame, also known as an activation record, is pushed onto the call stack each time a function is called.
    - This frame contains all necessary information about the function: parameters, local variables, and the return address.

Call Stack Mechanics in Recursion

PES UNIVERSITY

- **Stack Frames in Recursive Calls:**
  - In a recursive function, each call to itself creates a new stack frame and pushes it onto the call stack.
  - This process continues with each recursive call, adding a new frame to the stack, until a base condition is reached.

Call Stack Mechanics in Recursion

PES UNIVERSITY

- **Unwinding the Stack**
  - Once the base condition is met, the function starts returning its results, and the stack frames begin to pop off from the call stack.
  - Each frame's return feeds into the previous frame until the first call returns the final result.

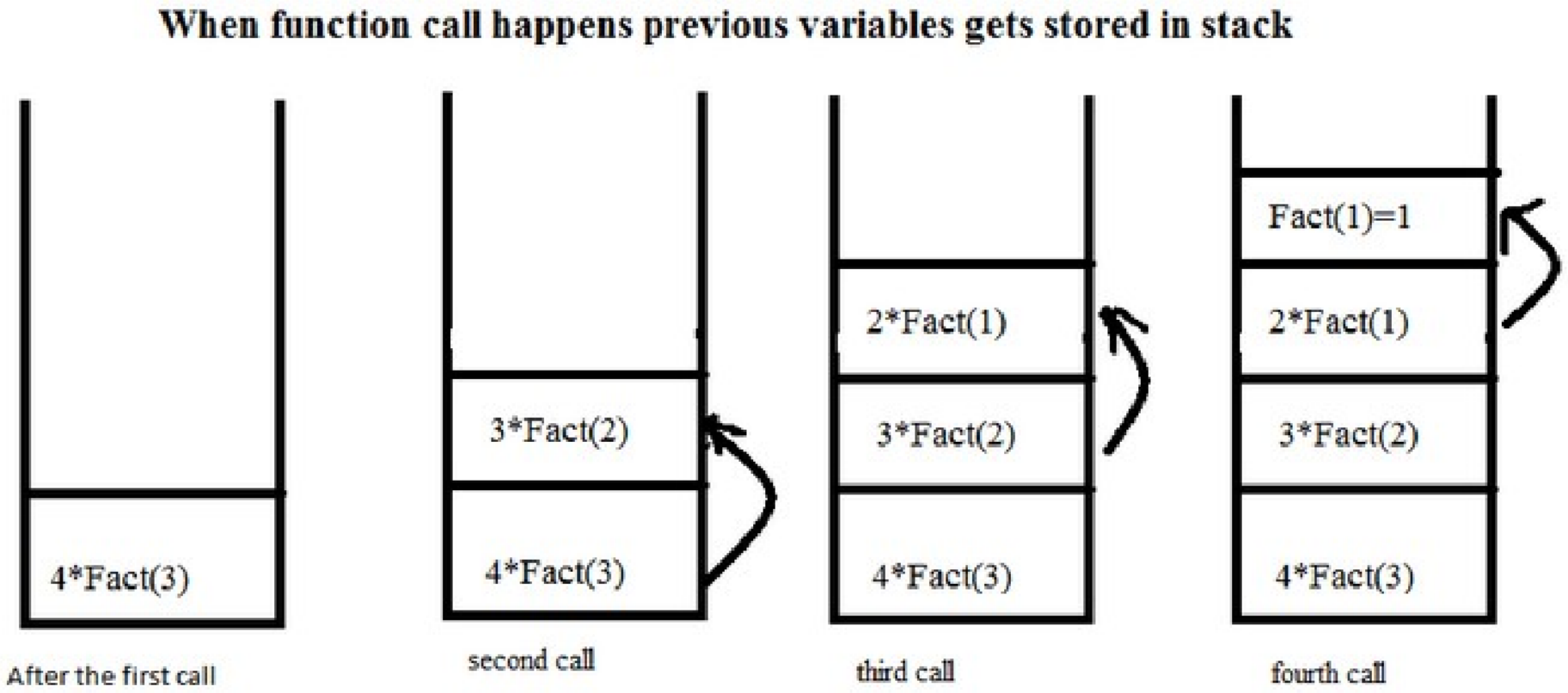
Call Stack Mechanics in Recursion

PES UNIVERSITY

- **Call Stack Progression - factorial(5)**
  - factorial(5) is called.
    - A frame for factorial(5) is pushed onto the stack.
  - This leads to factorial(4), and a new frame is added to the stack.
  - This continues down to factorial(1).
  - At factorial(1), the base case is met (as factorial(1) is 1), and the function begins to return.
  - The stack unwinds, each frame resolving and returning a value to the frame before it, eventually leading to the final result at factorial(5).

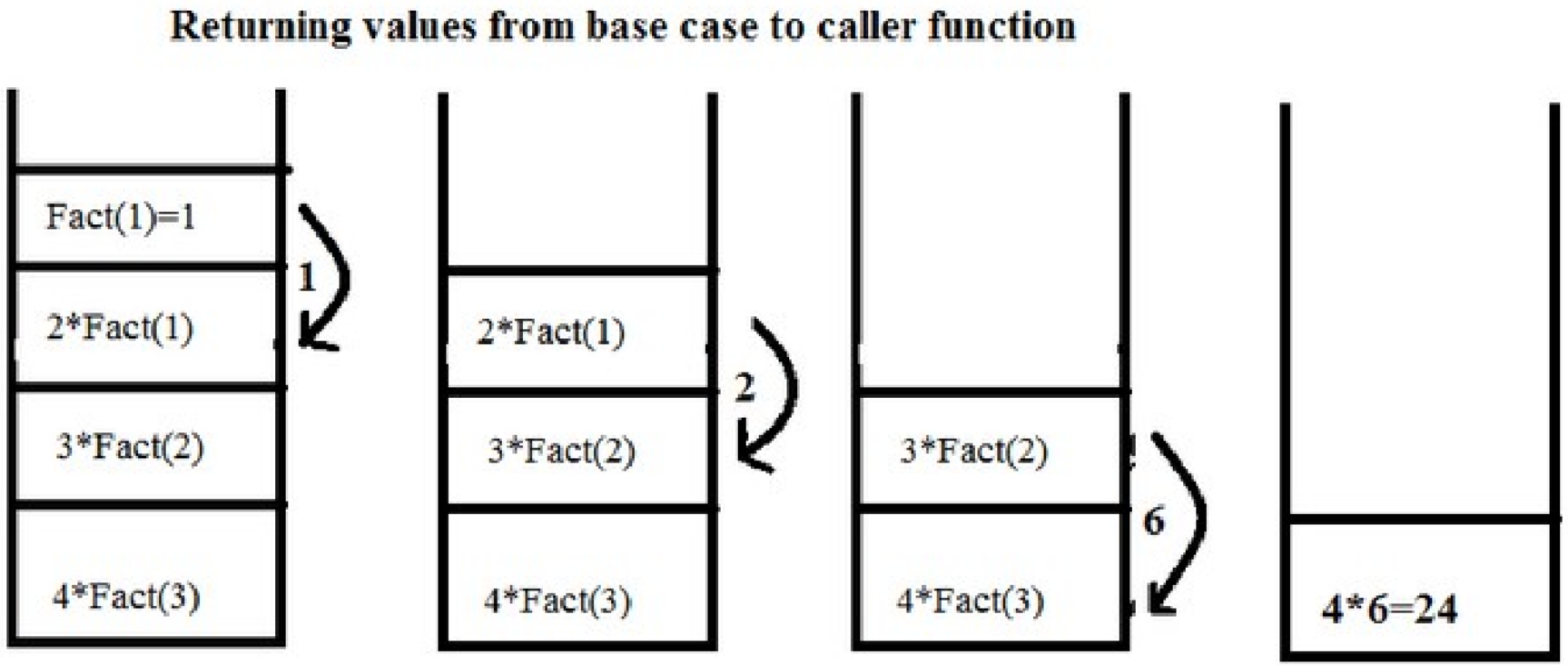
Call Stack Mechanics in Recursion

- **Call Stack Progression - factorial(4)**



Call Stack Mechanics in Recursion

- **Call Stack Progression - factorial(4)**



Role of Data Structures in Managing Expressions

- Expressions in the context of data structures are a fundamental concept in computer science and programming.
- An expression is a combination of values, constants, variables, operators, and functions that are constructed according to the syntax of a programming language to produce a new value.
- When it comes to data structures, expressions are particularly important in the evaluation, processing, and storage of data.

Role of Data Structures in Managing Expressions

PES UNIVERSITY

- **Parsing Expressions**
  - Complex expressions, especially those involving nested operations, require an organized method to be parsed and evaluated correctly.
  - Data structures like stacks and trees are crucial for parsing and evaluating expressions.

Role of Data Structures in Managing Expressions

PES UNIVERSITY

- **Infix, Prefix, and Postfix Expressions**
  - Different forms of expressions (infix, prefix, postfix) are better suited for specific data structures.
  - For instance, postfix expressions (Reverse Polish Notation) are efficiently evaluated using stacks, whereas prefix expressions (Polish Notation) are often processed using recursion or stacks.

Role of Data Structures in Managing Expressions

- **Stacks for Expression Evaluation**
  - Stacks are widely used for evaluating arithmetic expressions, particularly for converting infix expressions to postfix expressions.
  - The stack ensures that operators are applied in the correct order of precedence and associativity.

Role of Data Structures in Managing Expressions

PES UNIVERSITY

- **Application Examples**
  - **Compiler Design**
    - Data structures like stacks are used to parse expressions in the source code, perform syntax and semantic analysis, and generate the corresponding machine code or intermediate code.
  - **Calculators**
    - Software calculators often use stacks to manage arithmetic operations, especially for on-the-fly calculation as the user inputs an expression.

Role of Data Structures in Managing Expressions

PES UNIVERSITY

Feature	Prefix (Polish Notation)	Infix	Postfix (Reverse Polish Notation)
Notation	Operators precede their operands.	Operators are between operands.	Operators follow their operands.
Example	+ * 3 4 5 (represents (3 * 4) + 5)	3 * 4 + 5	3 4 * 5 + (represents (3 * 4) + 5)
Parentheses Required	No, due to unambiguous nature.	Yes, to override precedence.	No, due to unambiguous nature.
Evaluation Order	Read right to left.	Determined by precedence rules.	Read left to right.
Operator Precedence	Not relevant (implicit in the order).	Must be considered during parsing.	Not relevant (implicit in the order).

Role of Data Structures in Managing Expressions

PES UNIVERSITY

Feature	Prefix (Polish Notation)	Infix	Postfix (Reverse Polish Notation)
Ease of Computation	High for computers, low for humans.	Moderate for computers, high for humans.	High for computers, low for humans.
Stack Usage (for Evaluation)	Use a stack, pushing operands and popping them as operators are read (evaluate when operators are read).	Often requires conversion to prefix or postfix for machine evaluation.	Use a stack, pushing operands and popping them as operators are read (evaluate when operators are read).

THANK YOU

Dilip Maripuri
Associate Professor
Department of Computer Applications
dilip.maripuri@pes.edu