MODULE 1-4 [CPE11201]

STACK

สแตก

Assoc. Prof. Dr. Natasha Dejdumrong

OUTLINES

- Stack
- Array Representation of Stack
- Operations on a Stack
- Linked List Representation of Stack
- Operations on a Linked Stack
- Applications of Stack

Stack

- Stack is an important data structure which stores its elements in an ordered manner.
- We will explain the concept of stacks using an analogy. You must have seen a pile of plates where one plate is placed on top of another as shown in Figure.
- Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position.

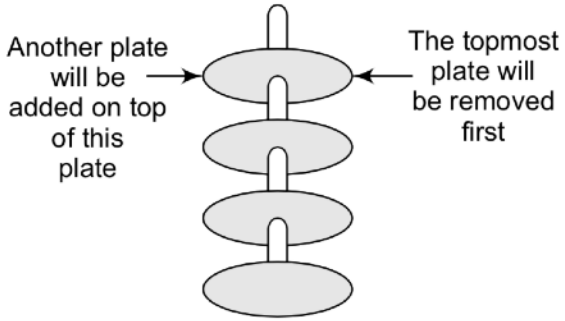


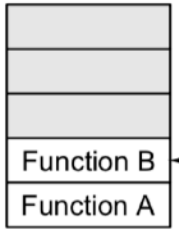
Figure 7.1 Stack of plates

Stack

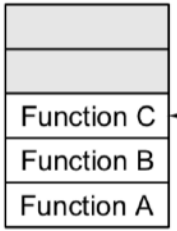
★ Now the question is where do we need stacks in computer science?

The answer is in function calls.

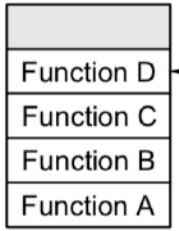
When A calls B, A is pushed on top of the system stack. When the execution of B is complete, the system control will remove A from the stack and continue with its execution.



When B calls C, B is pushed on top of the system stack. When the execution of C is complete, the system control will remove B from the stack and continue with its execution.



When C calls D, C is pushed on top of the system stack. When the execution of D is complete, the system control will remove C from the stack and continue with its execution.



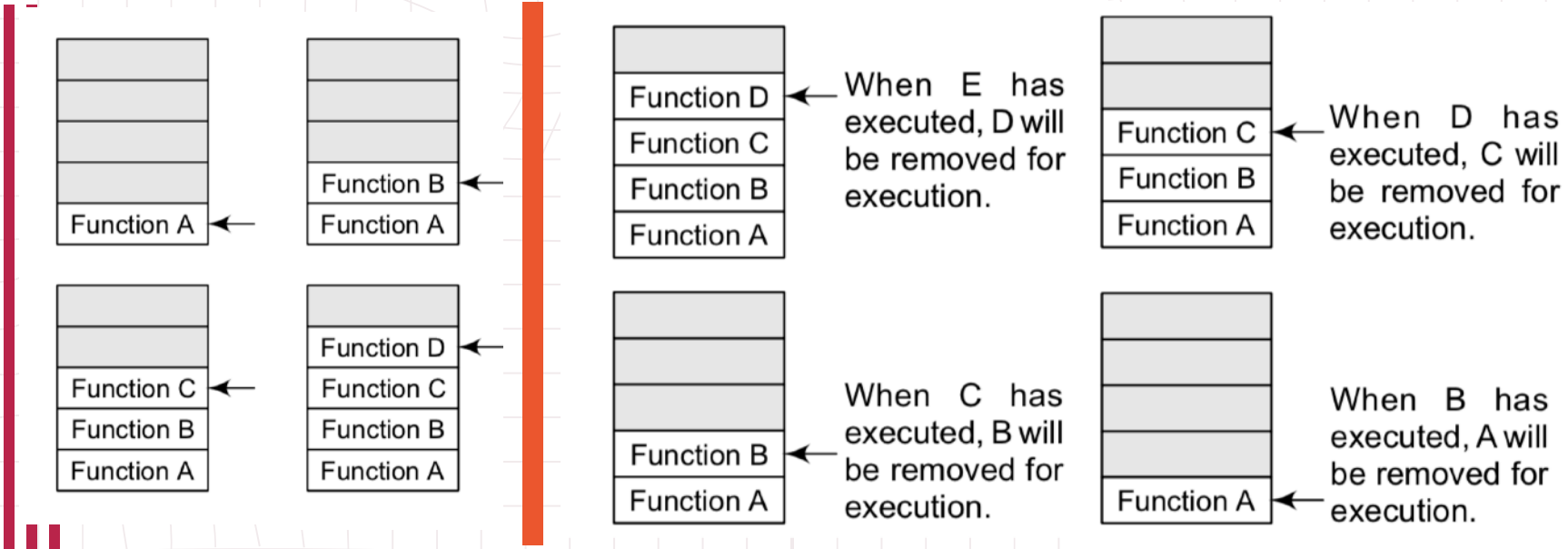
When D calls E, D is pushed on top of the system stack. When the execution of E is complete, the system control will remove D from the stack and continue with its execution.

Figure 7.2 System stack in the case of function calls

Stack

★ Now the question is where do we need stacks in computer science?

The answer is in function calls.



Array Representation of Stack

★ In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from.

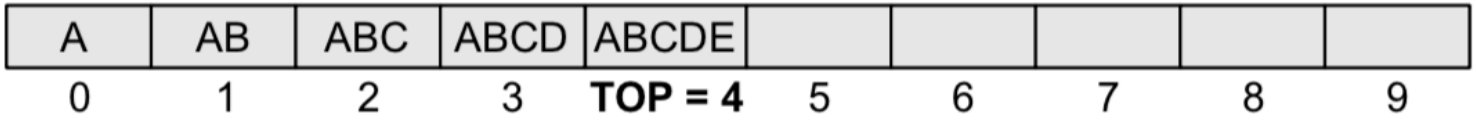


Figure 7.4 Stack

Operations on a Stack

- A stack supports three basic operations: push, pop, and peek.
- The push operation adds an element to the top of the stack.
- The pop operation removes the element from the top of the stack.
- The peek operation returns the value of the topmost element of the stack.

PUSH Operation

★ The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.

★ However, before inserting the value, we must first check if TOP = MAX–1, because if that is the case, then the stack is full and no more insertions can be done.

★ If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.

PUSH Operation

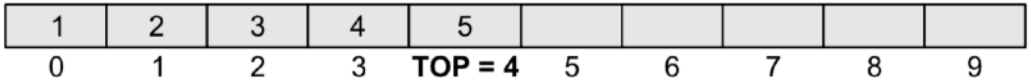


Figure 7.5 Stack

To insert an element with value 6, we first check if `TOP=MAX-1`. If the condition is false, then we increment the value of `TOP` and store the new element at the position given by `stack[TOP]`. Thus, the updated stack becomes as shown in Fig. 7.6.

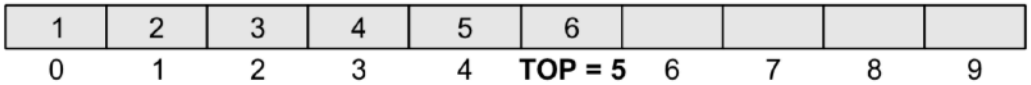


Figure 7.6 Stack after insertion

PUSH Operation

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Figure 7.7 Algorithm to insert an element in a stack

```
void push(int st[], int val)
{
        if(top == MAX-1)
        {
                printf("\n STACK OVERFLOW");
        }
        else
        {
                top++;
                st[top] = val;
        }
}
```

POP Operation

★ The pop operation is used to delete the topmost element from the stack.

★ However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done.

★ If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

POP Operation

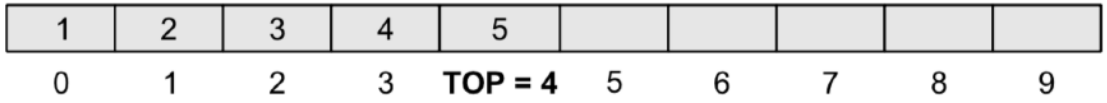


Figure 7.8 Stack

To delete the topmost element, we first check if TOP=NULL. If the condition is false, then we decrement the value pointed by TOP. Thus, the updated stack becomes as shown in Fig. 7.9.

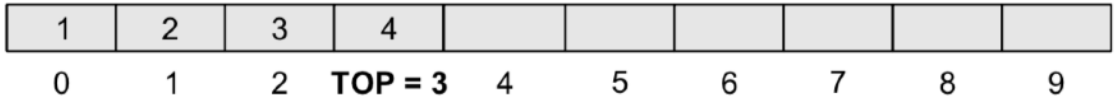


Figure 7.9 Stack after deletion

POP Operation

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Figure 7.10 Algorithm to delete an element from a stack

```
int pop(int st[])
{
        int val;
        if(top == -1)
        {
                printf("\n STACK UNDERFLOW");
                return -1;
        }
        else
        {
                val = st[top];
                top--;
                return val;
        }
}
```

PEEK Operation

★ Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.

★ However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.

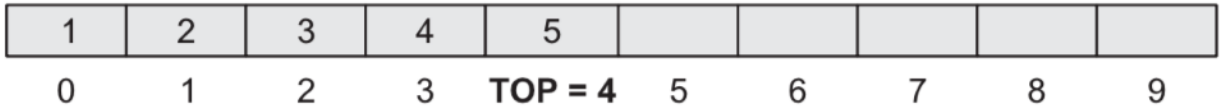


Figure 7.12 Stack

PEEK Operation

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Figure 7.11 Algorithm for Peek operation

```
int peek(int st[])
{
        if(top == -1)
        {
                printf("\n STACK IS EMPTY");
                return -1;
        }
        else
        return (st[top]);
}
```

PEEK Operation

```
int pop(int st[])
{
        int val;
        if(top == -1)
        {
                printf("\n STACK UNDERFLOW");
                return -1;
        }
        else
        {
                val = st[top];
                top--;
                return val;
        }
}
```

```
int peek(int st[])
{
        if(top == -1)
        {
                printf("\n STACK IS EMPTY");
                return -1;
        }
        else
        return (st[top]);
}
```

Linked List Representation of Stack

★ The technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size.

★ In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation.

★ But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.

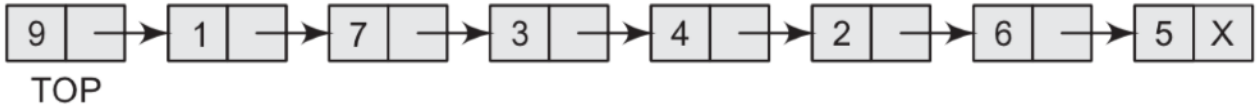


Figure 7.13 Linked stack

Linked List Representation of Stack

★ The storage requirement of linked representation of the stack with n elements is O(n), and the typical time requirement for the operations is O(1).

★ In a linked stack, every node has two parts–one that stores data and another that stores the address of the next node.

★ The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.

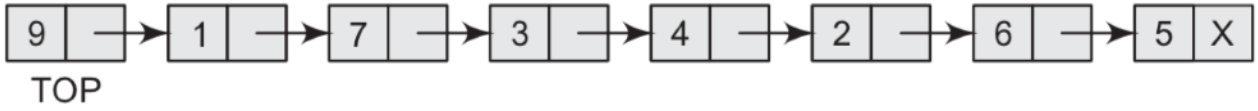


Figure 7.13 Linked stack

Operations on a Linked Stack

- A linked stack supports three basic operations:

  push, pop, and peek.

- The push operation adds an element to the top of the stack.
- The pop operation removes the element from the top of the stack.
- The peek operation returns the value of the topmost element of the stack.

PUSH Operation

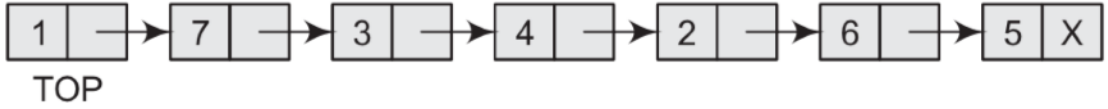


Figure 7.14 Linked stack

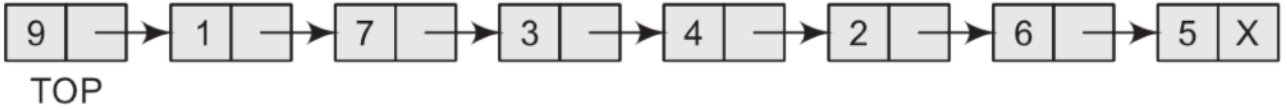


Figure 7.15 Linked stack after inserting a new node

PUSH Operation

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

Figure 7.16 Algorithm to insert an element in a linked stack

```
struct stack *push(struct stack *top, int val)
{
        struct stack *ptr;
        ptr = (struct stack*)malloc(sizeof(struct stack));
        ptr -> data = val;
        if(top == NULL)
        {
                ptr -> next = NULL;
                top = ptr;
        }
        else
        {
                ptr -> next = top;
                top = ptr;
        }
        return top;
}
```

POP Operation

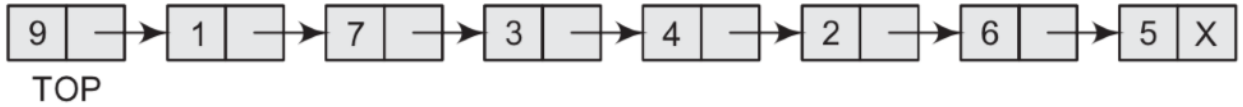


Figure 7.17 Linked stack

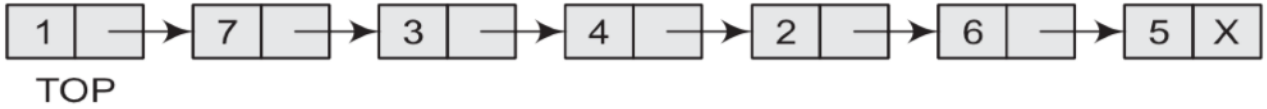


Figure 7.18 Linked stack after deletion of the topmost element

POP Operation

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

Figure 7.19 Algorithm to delete an element from a linked stack

```
struct stack *pop(struct stack *top)
{
        struct stack *ptr;
        ptr = top;
        if(top == NULL)
        printf("\n STACK UNDERFLOW");
        else
        {
                top = top -> next;
                printf("\n The value being deleted is: %d", ptr -> data);
                free(ptr);
        }
        return top;
}
```

Applications of Stack

- We will discuss typical problems where stacks can be easily applied for a simple and efficient solution. The topics that will be discussed in this section include the following:
  - Reversing a list
  - Parentheses checker
  - Conversion of an infix expression into a postfix expression
  - Evaluation of a postfix expression
  - Conversion of an infix expression into a prefix expression
  - Evaluation of a prefix expression
  - Recursion
  - Tower of Hanoi

Reversing a list

- A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack.
- Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

Reversing a list

```
#include <conio.h>
int stk[10];
int top=-1;
int pop();
void push(int);
int main()
{
        int val, n, i,
        arr[10];
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements of the array : ");
        for(i=0;i<n;i++)
                scanf("%d", &arr[i]);
        for(i=0;i<n;i++)
                push(arr[i]);
        for(i=0;i<n;i++)
        {
                val = pop();
                arr[i] = val;
        }
        printf("\n The reversed array is : ");
        for(i=0;i<n;i++)
                printf("\n %d", arr[i]);
        getche"();
        return 0;
}
```

```
void push(int val)
{
        stk[++top] = val;
}
int pop()
{
        return(stk[top--]);
}
```

Output

```
Enter the number of elements in the array : 5
Enter the elements of the array : 1 2 3 4 5
The reversed array is : 5 4 3 2 1
```

Parentheses checker

- Stacks can be used to check the validity of parentheses in any algebraic expression.

For example, an algebraic expression is valid if for every open bracket there is a corresponding closing bracket.

- For example, the expression (A+B} is invalid but an expression {A + (B – C)} is valid.
- Look at the program below which traverses an algebraic expression to check for its validity.

Conversion of an infix into a postfix expression

The expression `(A + B) * C` can be written as:

`[AB+]*C`

`AB+C*` in the postfix notation

Example 7.1 Convert the following infix expressions into postfix expressions.

Solution

(a) (A–B) * (C+D)

```
[AB–] * [CD+]
AB–CD+*
```

(b) (A + B) / (C + D) – (D * E)

```
[AB+] / [CD+] – [DE*]
[AB+CD+/] – [DE*]
AB+CD+/DE*–
```

Conversion of an infix into a postfix expression

Conversion of an Infix Expression into a Postfix Expression

Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only +, -, *, /, % operators. The precedence of these operators can be given as follows:

Higher priority *, /, %
Lower priority +, -

Example 7.2 Convert the following infix expressions into prefix expressions.

Solution

(a) (A + B) * C
(+AB)*C
*+ABC

(b) (A-B) * (C+D)
[-AB] * [+CD]
*-AB+CD

(c) (A + B) / (C + D) - (D * E)
[+AB] / [+CD] - [*DE]
[/+AB+CD] - [*DE]
-/+AB+CD*DE

Conversion of an infix into a postfix expression

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
       IF a "(" is encountered, push it on the stack
       IF an operand (whether a digit or a character) is encountered, add it to the
       postfix expression.
       IF a ")" is encountered, then
         a. Repeatedly pop from stack and add it to the postfix expression until a
            "(" is encountered.
         b. Discard the "(". That is, remove the "(" from stack and do not
            add it to the postfix expression
       IF an operator O is encountered, then
         a. Repeatedly pop from stack and add each operator (popped from the stack) to the
            postfix expression which has the same precedence or a higher precedence than O
         b. Push the operator O to the stack
       [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

Figure 7.22 Algorithm to convert an infix notation to postfix notation

Conversion of an infix into a postfix expression

Example 7.3 Convert the following infix expression into postfix expression using the algorithm given in Fig. 7.22.

(a) A - (B / C + (D % E * F) / G)* H

(b) A - (B / C + (D % E * F) / G)* H)

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
-	(-	A
(	(- (	A
B	(- (	A B
/	(- (/	A B
C	(- (/	A B C
+	(- (+	A B C /
(	(- (+ (	A B C /
D	(- (+ (	A B C / D
%	(- (+ (%	A B C / D
E	(- (+ (%	A B C / D E
*	(- (+ (% *	A B C / D E
F	(- (+ (% *	A B C / D E F
)	(- (+	A B C / D E F * %
/	(- (+ /	A B C / D E F * %
G	(- (+ /	A B C / D E F * % G
)	(-	A B C / D E F * % G / +
*	(- *	A B C / D E F * % G / +
H	(- *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * -

Recursion

★ A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases. They are

- ★ Base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- ★ Recursive case, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Recursion: Factorial

PROBLEM	SOLUTION
$5!$	$5 \times 4 \times 3 \times 2 \times 1!$
$= 5 \times 4!$	$= 5 \times 4 \times 3 \times 2 \times 1$
$= 5 \times 4 \times 3!$	$= 5 \times 4 \times 3 \times 2$
$= 5 \times 4 \times 3 \times 2!$	$= 5 \times 4 \times 6$
$= 5 \times 4 \times 3 \times 2 \times 1!$	$= 5 \times 24$
	$= 120$

Figure 7.27 Recursive factorial function

Recursion: Factorial

PROBLEM	SOLUTION
$5!$	$5 \times 4 \times 3 \times 2 \times 1!$
$= 5 \times 4!$	$= 5 \times 4 \times 3 \times 2 \times 1$
$= 5 \times 4 \times 3!$	$= 5 \times 4 \times 3 \times 2$
$= 5 \times 4 \times 3 \times 2!$	$= 5 \times 4 \times 6$
$= 5 \times 4 \times 3 \times 2 \times 1!$	$= 5 \times 24$
	$= 120$

Figure 7.27 Recursive factorial function

- **Base case** is when `n = 1`, because if `n = 1`, the result will be `1` as `1! = 1`.
- **Recursive case** of the factorial function will call itself but with a smaller value of `n`, this case can be given as

```
factorial(n) = n × factorial (n-1)
```

Recursion: Factorial

PROGRAMMING EXAMPLE

10. Write a program to calculate the factorial of a given number.

```
#include <stdio.h>
int Fact(int);    // FUNCTION DECLARATION
int main()
{
        int num, val;
        printf("\n Enter the number: ");
        scanf("%d", &num);
        val = Fact(num);
        printf("\n Factorial of %d = %d", num, val);
        return 0;
}
int Fact(int n)
{
        if(n==1)
                return 1;
        else
            return (n * Fact(n-1));
}
```

Output

```
Enter the number : 5
Factorial of 5 = 120
```

Recursion: Greatest Common Divisor

The greatest common divisor of two numbers (integers) is the largest integer that divides both the numbers. We can find the GCD of two numbers recursively by using the *Euclid's algorithm* that states

$$\text{GCD}(a, b) = \begin{cases} b, & \text{if } b \text{ divides } a \\ \text{GCD}(b, a \bmod b), & \text{otherwise} \end{cases}$$

GCD can be implemented as a recursive function because if `b` does not divide `a`, then we call the same function (GCD) with another set of parameters that are smaller than the original ones.

Recursion: Greatest Common Divisor

Working

Assume `a = 62` and `b = 8`

```
GCD(62, 8)
   rem = 62 % 8 = 6
   GCD(8, 6)
         rem = 8 % 6 = 2
         GCD(6, 2)
               rem = 6 % 2 = 0
         Return 2
   Return 2
Return 2
```

Recursion: Greatest Common Divisor

PROGRAMMING EXAMPLE

11. Write a program to calculate the GCD of two numbers using recursive functions.

```
#include <stdio.h>
int GCD(int, int);
int main()
{
        int num1, num2, res;
        printf("\n Enter the two numbers: ");
        scanf("%d %d", &num1, &num2);
        res = GCD(num1, num2);
        printf("\n GCD of %d and %d = %d", num1, num2, res);
        return 0;
}

int GCD(int x, int y)
{
        int rem;
        rem = x%y;
        if(rem==0)
                return y;
        else
                return (GCD(y, rem));
}
```

Output

```
Enter the two numbers : 8 12
GCD of 8 and 12 = 4
```

Recursion: Finding Exponent

We can also find exponent of a number using recursion. To find x^y, the base case would be when $y=0$, as we know that any number raised to the power 0 is 1. Therefore, the general formula to find x^y can be given as

$$EXP(x, y) = \begin{cases} 1, & \text{if } y == 0 \\ x \times EXP(x^{y-1}), & \text{otherwise} \end{cases}$$

Working

```
exp_rec(2, 4) = 2 × exp_rec(2, 3)
        exp_rec(2, 3) = 2 × exp_rec(2, 2)
              exp_rec(2, 2) 2 × exp_rec(2, 1)
                    exp_rec(2, 1) = 2 × exp_rec(2, 0)
                          exp_rec(2, 0) = 1
                    exp_rec(2, 1) = 2 × 1 = 2
              exp_rec(2, 2) = 2 × 2 = 4
```

Recursion: Finding Exponent

PROGRAMMING EXAMPLE

12. Write a program to calculate `exp(x,y)` using recursive functions.

```
#include <stdio.h>
int exp_rec(int, int);
int main()
{
        int num1, num2, res;
        printf("\n Enter the two numbers: ");
        scanf("%d %d", &num1, &num2);
        res = exp_rec(num1, num2);
        printf ("\n RESULT = %d", res);
        return 0;
}
int exp_rec(int x, int y)
{
        if(y==0)
                return 1;
        else
                return (x * exp_rec(x, y-1));
}
```

Output

```
Enter the two numbers : 3 4
RESULT = 81
```

Recursion: The Fibonacci Series

The Fibonacci series can be given as

0 1 1 2 3 5 8 13 21 34 55 .....

That is, the third term of the series is the sum of the first and second terms. Similarly, fourth term is the sum of second and third terms, and so on. Now we will design a recursive solution to find the nth term of the Fibonacci series. The general formula to do so can be given as

As per the formula, FIB(0) =0 and FIB(1) = 1. So we have two base cases. This is necessary because every problem is divided into two smaller problems.

$$\text{FIB}(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ \text{FIB}(n-1) + \text{FIB}(n-2), & \text{otherwise} \end{cases}$$

Recursion: The Fibonacci Series

```
int Fibonacci(int n)
{
        if ( n == 0 )
                return 0;
        else if ( n == 1 )
                return 1;
        else
                return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

Output

```
Enter the number of terms
Fibonacci series
   0     1        1        2        3
```

Recursion: Tower of Hanoi

★ The tower of Hanoi is one of the main applications of recursion. It says,

if you can solve n–1 cases,
then you can easily solve the nth case.

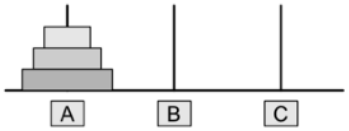


Figure 7.33 Tower of Hanoi

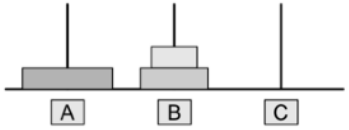


Figure 7.34 Move rings from A to B

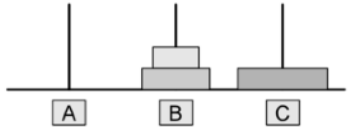


Figure 7.35 Move ring from A to C

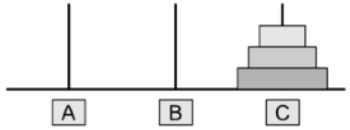


Figure 7.36 Move ring from B to C

Recursion: Tower of Hanoi

To summarize, the solution to our problem of moving n rings from A to C using B as spare can be given as:

Base case: `if n=1`

- Move the ring from A to C using B as spare

Recursive case:

- Move `n – 1` rings from A to B using C as spare
- Move the one ring left on A to C using B as spare
- Move `n – 1` rings from B to C using A as spare

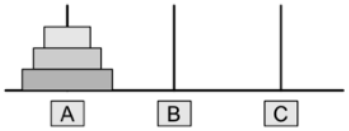


Figure 7.33 Tower of Hanoi

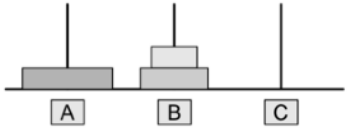


Figure 7.34 Move rings from A to B

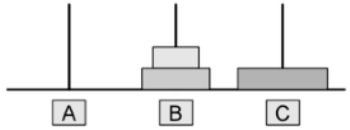


Figure 7.35 Move ring from A to C

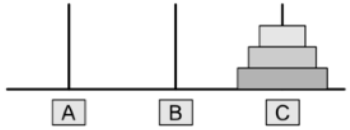
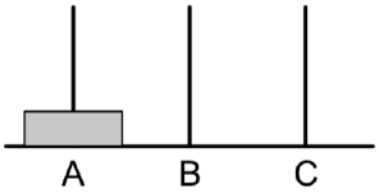
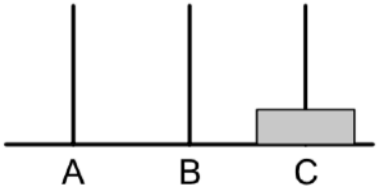


Figure 7.36 Move ring from B to C

Recursion: Tower of Hanoi

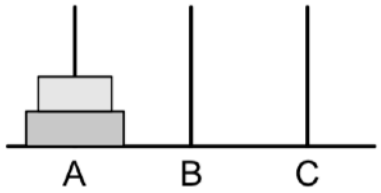


(Step 1)

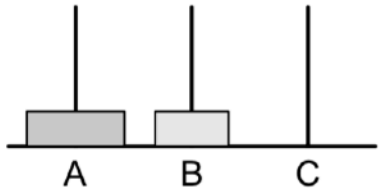


(Step 2)

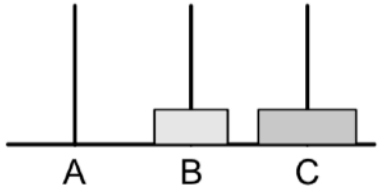
(If there is only one ring, then simply move the ring from source to the destination.)



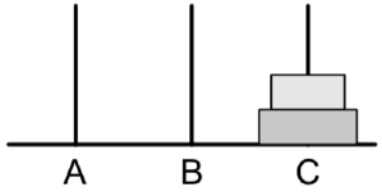
(Step 1)



(Step 2)



(Step 3)



(Step 4)

(If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from spare to the destination.)

Recursion: Tower of Hanoi

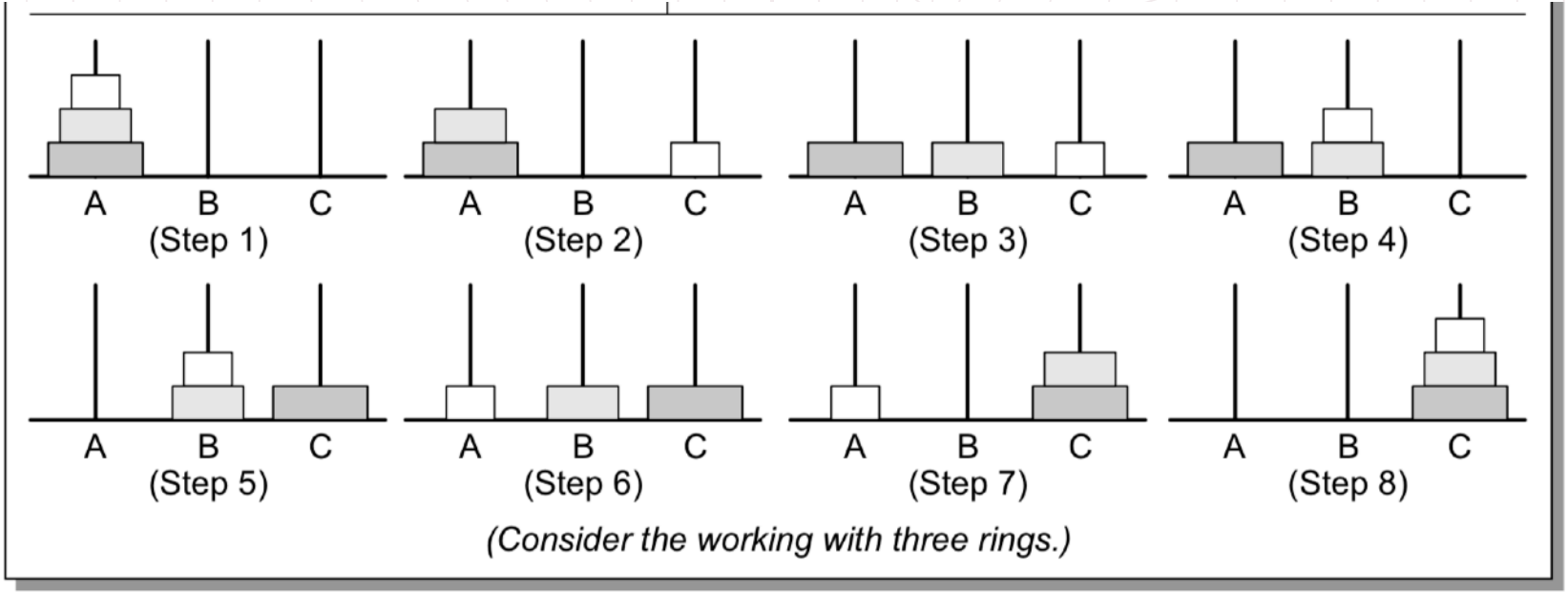


Figure 7.37 Working of Tower of Hanoi with one, two, and three rings

Recursion: Tower of Hanoi

```
#include <stdio.h>
int main()
{
	int n;
	printf("\n Enter the number of rings: ");
	scanf("%d", &n);
	move(n,'A', 'C', 'B');
	return 0;
}
void move(int n, char source, char dest, char spare)
{
	if (n==1)
		printf("\n Move from %c to %c",source,dest);
	else
	{
		move(n-1,source,spare,dest);
		move(1,source,dest,spare);
		move(n-1,spare,dest,source);
	}
}
```

Questions and Answers