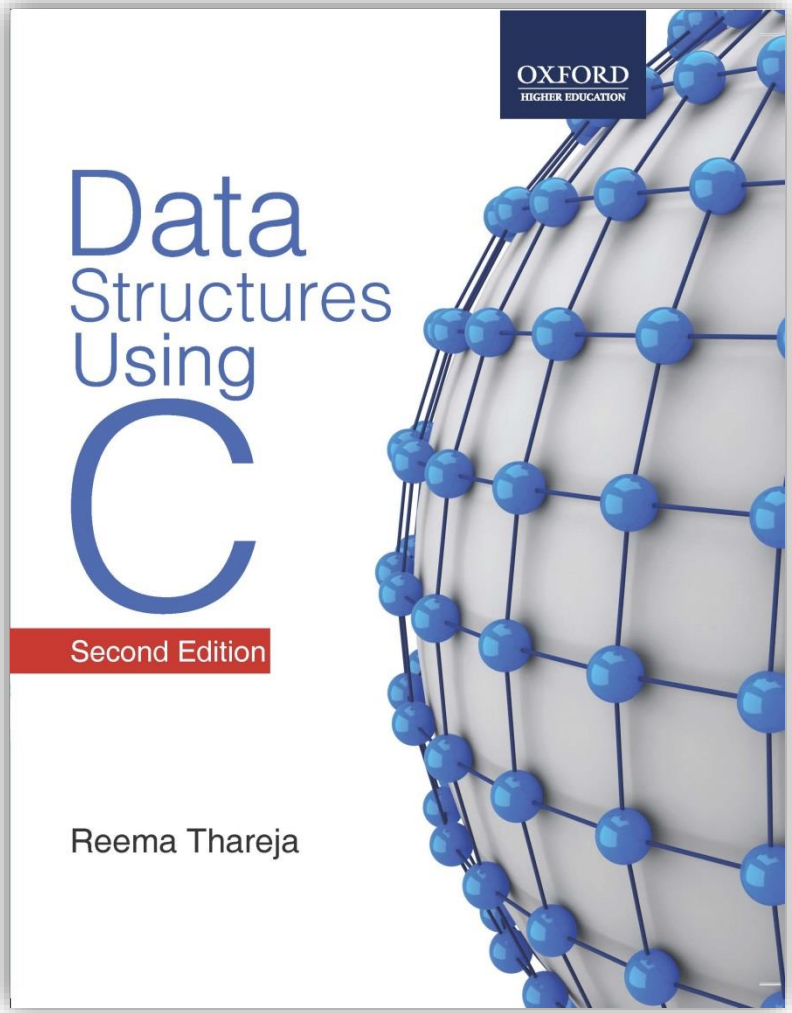


OXFORD
HIGHER EDUCATION

Data Structures Using C, 2e

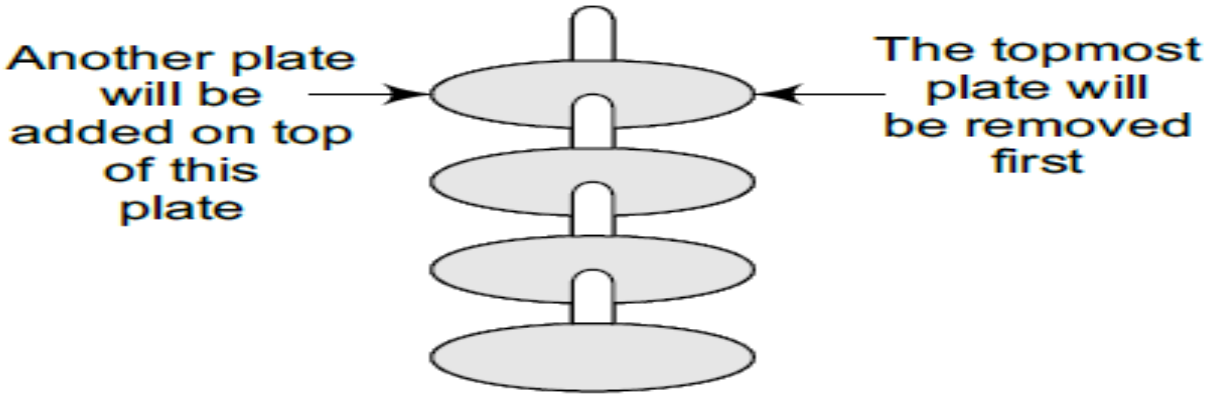
Reema Thareja

Chapter 7

Stacks

Introduction

- Stack is an important data structure which stores its elements in an ordered manner.
- Take an analogy of a pile of plates where one plate is placed on top of the other. A plate can be removed only from the topmost position. Hence, you can add and remove the plate only at/from one position, that is, the topmost position.



Stacks

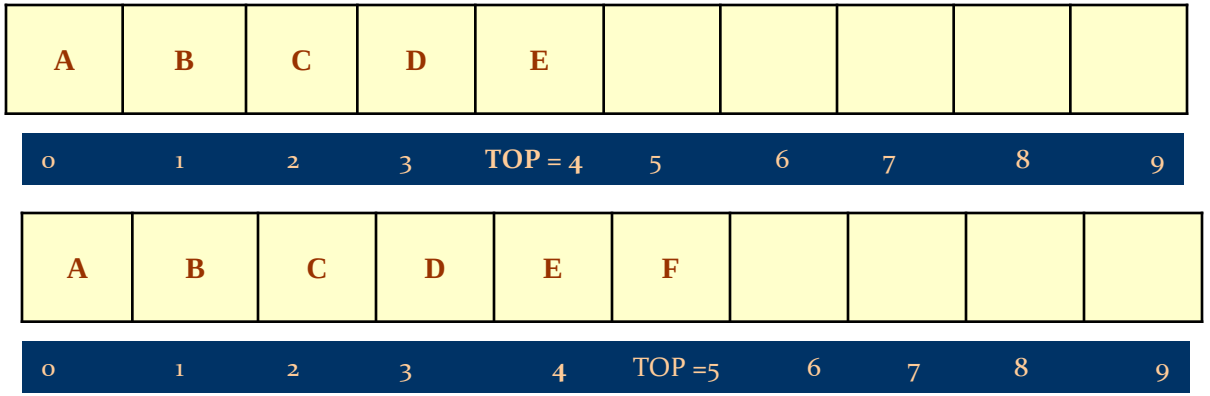
- A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the *top*.
- Hence, a stack is called a LIFO (Last-In, First-Out) data structure as the element that is inserted last is the first one to be taken out.
- Stacks can be implemented either using an array or a linked list.

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.

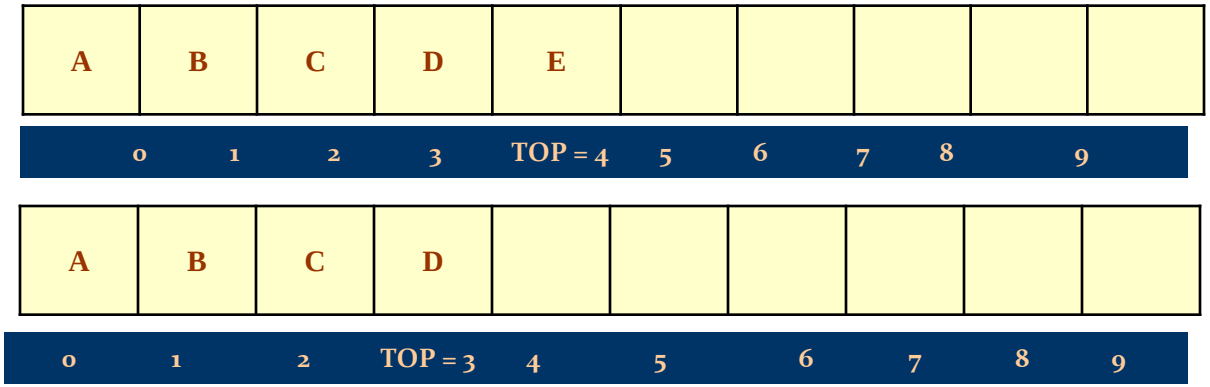
Push Operation

- The push operation is used to insert an element in to 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 this is the case then it means the stack is full and no more insertions can further be done.
- If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.



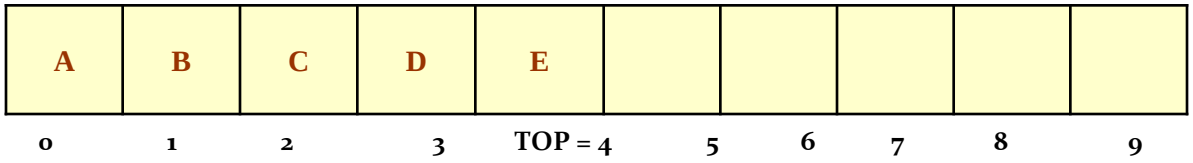
Pop Operation

- 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 this is the case then it means the stack is empty so no more deletions can further be done.
- If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.



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 peep operation first checks if the stack is empty or contains some elements.
- If TOP = NULL, then an appropriate message is printed else the value is returned.



Here Peep operation will return E, as it is the value of the topmost element of the stack.

Algorithm for Push Operations

```
Algorithm to PUSH an element in a
stack

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

Algorithm for Pop Operations

```
Algorithm to POP an element from
a stack

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

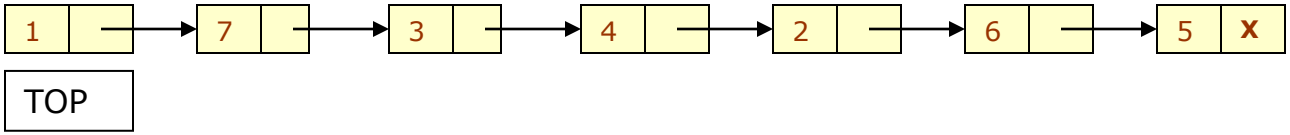
Algorithm for Peek Operation

```
Algorithm for Peek Operation

Step 1: IF TOP =NULL, then
     PRINT "STACK IS EMPTY"
     Go TO Step 3
          [END OF IF]
Step 2: RETURN STACK[TOP]
Step 3: END
```

Linear Representation of Stacks

- 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.
- If TOP is NULL then it indicates that the stack is empty.



Push Operation on a Linked Stack

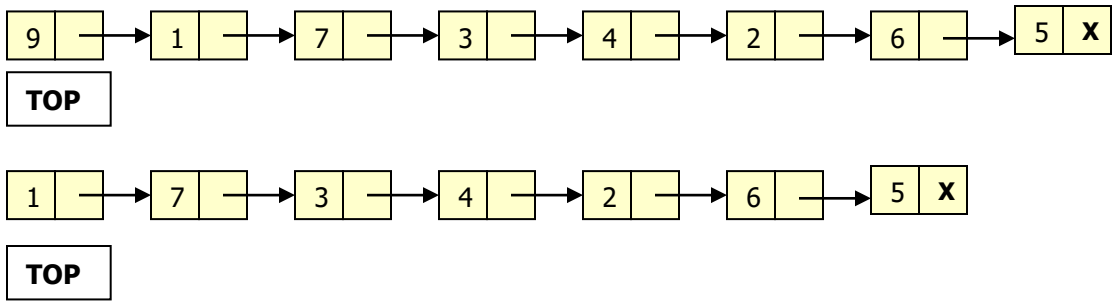
```
Algorithm to PUSH an element in a linked
stack

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, then
            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
```

Pop Operation on a Linked Stack

```
Algorithm to POP an element from a
stack

Step 1: IF TOP = NULL, then
              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
```



Applications of Stacks

- Parenthesis (Bracket) checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Recursion
  - Factorial
  - Exponentiation
  - GCD
  - Fibonacci number
  - Tower of Hanoi

Applications of Stacks

Parenthesis (Bracket) balance checker

- Algorithm

  1. Create a character stack, initially empty.

  2. Traverse the string exp one character at a time from start to end.

  2.1 If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then **push** it to stack.

  2.2 If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from the stack and if the popped character is the matching starting bracket, then continue to check next character(step 2.1). Else brackets are **Not Balanced**. Goto step 4.

  2.3 For any other character in the expression, continue to next character. Goto step 2.1.

  3. After complete traversal, if some starting brackets are left in the stack, then the expression is **Not balanced**, else **Balanced**.

  4. Stop

Polish Notations

- Algebraic expression can be represented in 3 notations:
- Infix notation E.g. (a+b) with use parenthesis.
- Polish notations are parenthesis free.
- Polish notations are devised by a Polish scientist named **Jan Łukasiewicz**.
- Prefix notation (Polish Notation) E.g. +ab
- Postfix Notation (Reverse Polish Notation) E.g: ab+

Infix Notation

- Infix, Postfix and Prefix notations are three different but equivalent notations of writing algebraic expressions.
- While writing an arithmetic expression using infix notation, the operator is placed between the operands. For example, *A+B;* here, plus operator is placed between the two operands A and B.
- Although it is easy to write expressions using infix notation, computers find it difficult to parse as they need a lot of information to evaluate the expression.
- Information is needed about operator precedence, associativity rules, and brackets which overrides these rules.
- So, computers work more efficiently with expressions written using prefix and postfix notations.

Postfix Notation

- Postfix notation was given by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher. His aim was to develop a **parenthesis-free prefix notation** (also known as Polish notation) and a postfix notation which is better known as **Reverse Polish Notation** or RPN.
- In postfix notation, the operator is placed after the operands. For example, if an expression is written as *A+B* in infix notation, the same expression can be written as *AB+* in postfix notation.
- The order of evaluation of a postfix expression is always from left to right.

Prefix Notation

- In a prefix notation, the operator is placed before the operands.
- For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.
- While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator.
- Prefix expressions also do not follow the rules of operator precedence, associativity, and even brackets cannot alter the order of evaluation.
- The expression (A + B) * C is written as:
  *+ABC in the prefix notation

Postfix Notation

- The expression (A + B) * C is written as:

  AB+C* in the postfix notation.

- A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands.

- For example, given a postfix notation AB+C*. While evaluation, addition will be performed prior to multiplication.

Conversion of Infix to Postfix Expression

Step 1: Create a character stack S
Step 2: Accept the input string in infix notation.
Step 3: Repeat until each character in the infix notation is scanned
3.1 If a "(" is encountered, push it on the stack
3.2 If an operand (whether a digit or an alphabet) is encountered, add it to the postfix expression.
3.3 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.
3.4 If any operator X is encountered, then;

a) Repeatedly pop operators with precedence >= that of X from stack and add them the postfix expression
b) Push the operator X to the stack
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT

Evaluation of an Infix Expression

Algorithm to evaluate a postfix expression

Step 1: Input the postfix expression(of the form e.g **56*7+**)
Step 2: Scan every character of the postfix expression and
repeat steps 3 until end of the expression.
Step 3: If an operand is encountered, push it on the stack
If an operator X is encountered, then
a. pop the top two elements from the stack as A and B
b. Evaluate B X A, where A was the topmost element and
B was the element below A.
c. Push the result of evaluation on the stack
[END OF IF]
Step 4: SET RESULT equal to the topmost element of the stack
Step 5: EXIT

Recursion

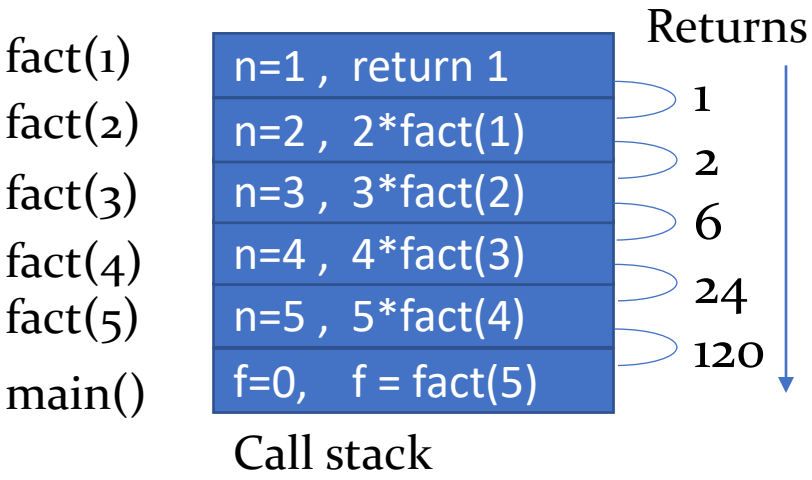
- Recursion is an implicit application of STACK ADT.
- ***A recursive function is a function that calls itself*** *to* ***solve a smaller version of its task*** *until* ***a final call is made which does not require a call to itself.***
- Every recursive solution has two major cases: the ***base case*** 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 subparts. Second the function calls itself but with subparts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Recursion examples: Factorial

$$Fact(n) = \begin{cases} 1, \text{ if } n = 1 \\ n * fact(n-1), \text{ if } n>1 \end{cases}$$

```
// recursive solution
int fact( int n)
{
   if( n==1 )
      return 1;
   return (n*fact(n-1));
}

int main( )
{
int  f;
f = fact(5);
printf("Factorial = %d", f);
}
```




```
// Iterative solution
int factorial(int n)
{
int fact=1, i ;
for(i=n; i>=1; i--)
   fact = fact * i;
return fact;
}
```

Recursion examples: exponent

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

```
14  int exp_it(int x, int y)
15  {
16      int i,res=1;
17      if (y==0)
18          return 1;
19
20      for(i=y; i>=1; i--)
21          res = res * x;
22      return res;
23  }
```

Iterative solution

```
int main()
{
    int p=2, q = 4, res;
    res = exp(p, q);
}
```

```
int exp(int x, int y)
{
if(y==0)
    return 1;
else
    return (x  *  exp(x, y-1));
}
```

Recursive solution

Recursion examples: GCD

GCD(x , y) =
- y, if y divides x.
- GCD(y, x mod y), otherwise.

```
int gcd(int x, int y)
{
     int rem;
     rem = x%y;
     while (rem != 0)
     {
          x=y;
          y = rem;
          rem = x%y;
     }
     return y;
}
```

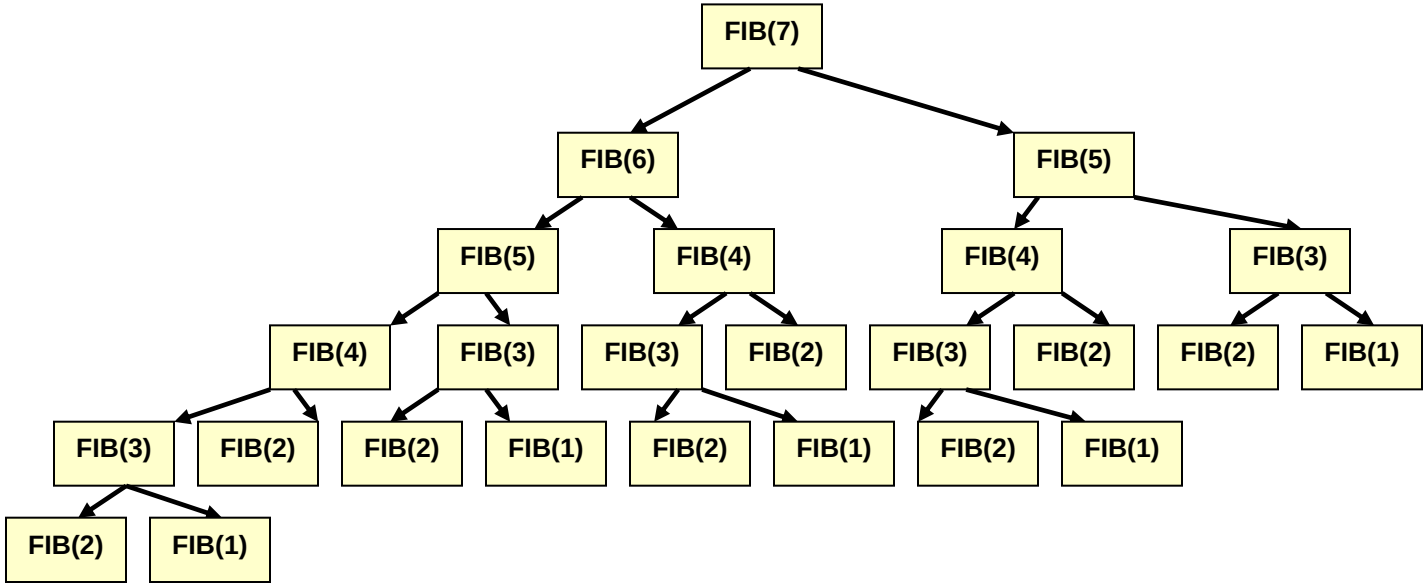
```
int main()
{
   int p=120, q=70, res;
   res = GCD(p, q);
}
```

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

Fibonacci Series

- The Fibonacci series can be given as:0 1 1 2 3 5 8 13…..
- 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, so on and so forth.
- A recursive solution to find the nth term of the Fibonacci series can be given as:

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



Fibonacci Series

n^{th} term of Fibonacci series

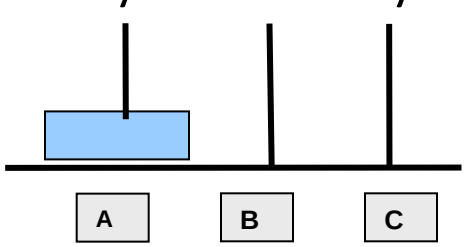
$$FIB\ (n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ FIB\ (n-1) + FIB(n-2), & \text{otherwise} \end{cases}$$

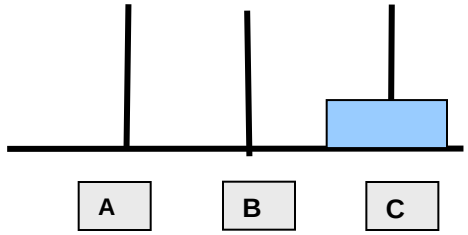
```
int main()
{
    int res;
    res = fib(4);
return 0;
}
```

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

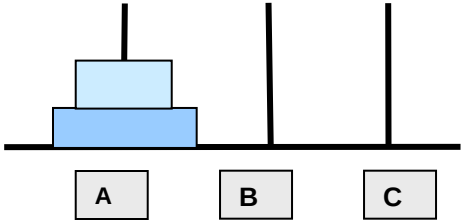
Tower of Hanoi

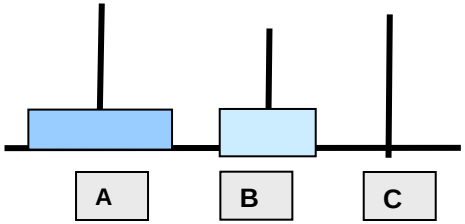
Tower of Hanoi is one of the main applications of a recursion. It says, "if you can solve *n-1* cases, then you can easily solve the *nth* case"

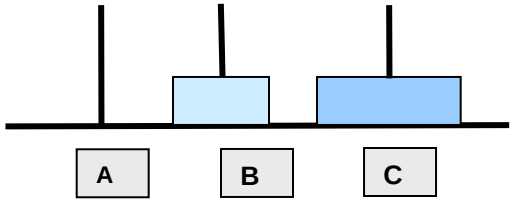


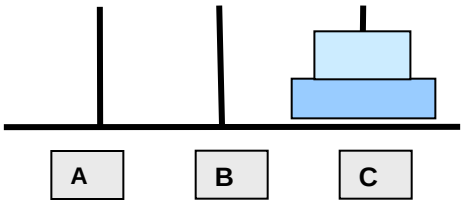


If there is only one ring, then simply move the ring from source to the destination





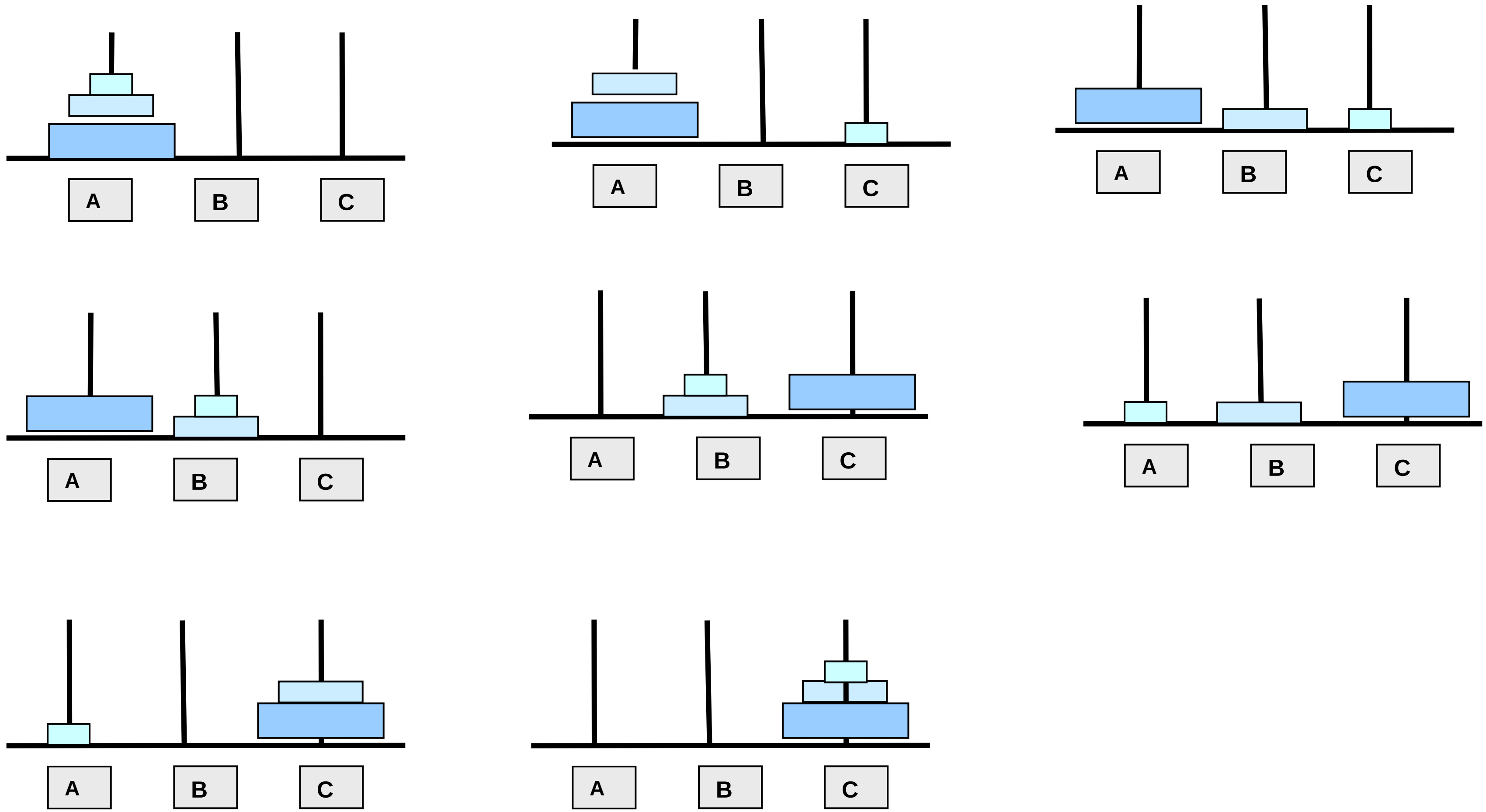




If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from the source to the destination

Tower of Hanoi

Consider the working with three rings.



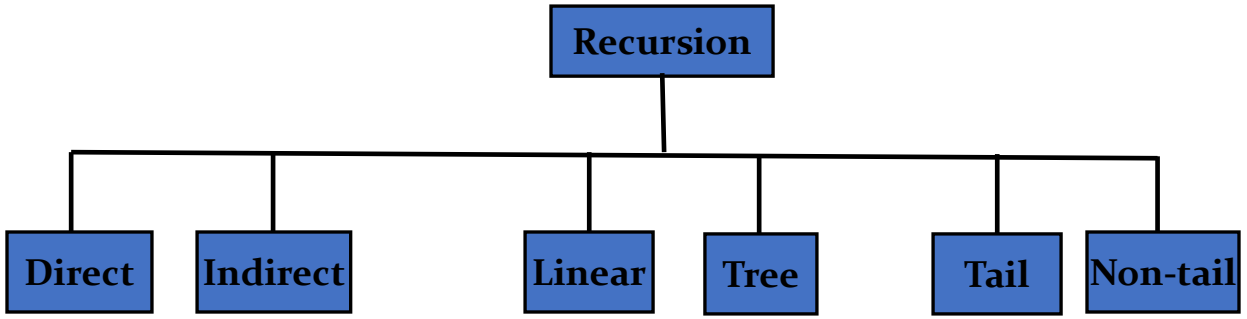
Tower of Hanoi

```
move(n,'A', 'B', 'C');
```

```
11  void move(int n, char A,  char B, char C)
12  {
13  if (n==1)
14     printf("\n Move %d from %c to %c",n, A,C);
15  else
16  {
17     move(n-1,A,C,B);
18     move(1,A,B,C);
19     move(n-1,B,A, C);
20  }
21  }
```

Types of Recursion

- Any recursive function can be characterized based on:
- whether the **function calls itself directly or indirectly** (direct or indirect recursion).
- whether any operation is pending at each recursive call (tail-recursive or not).
- the structure of the calling pattern (linear or tree-recursive).



Direct Recursion

- A function is said to be ***directly*** **recursive** if it explicitly calls itself.
- For example, consider the function given below.

```
int Func( int n)
{
        if(n==0)
                retrun n;
        return (Func(n-1));
}
```

Indirect Recursion

- A function is said to be ***indirectly*** **recursive** if it contains a call to another function which ultimately calls it.
- Look at the functions given below. These two functions are indirectly recursive as they both call each other.

```
int Func1(int n)
{
        if(n==0)
            return n;
        else
            return Func2(n);
}
```

```
int Func2(int x)
{
        return Func1(x-1);
}
```

Linear Recursion

- Recursive functions can also be characterized depending on the way in which the recursion grows: in a **linear fashion** OR forming a **tree structure**.
- In simple words, a recursive function is said to be *linearly* recursive when no pending operation involves another recursive call to the same function.
- For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to *fact()* function.

```
int fact( int n)
{
        if( n==1 )
                return 1;
        return (n*fact(n-1));
}
```

Tree Recursion

- A recursive function is said to be ***tree*** **recursive** (or *non-linearly* recursive) if the pending operation makes another recursive call to the function.
- For example, the Fibonacci function Fib in which the pending operations recursively calls the Fib function.

```
int Fibonacci(int num)
{
        if(num <= 1)
                return n;
        return ( Fibonacci (num - 1) + Fibonacci(num – 2));
}
```

Tail & Non-Tail Recursion

- A recursive function is said to be ***tail recursive*** if no operations are pending to be performed when the recursive function returns to its caller.
- That is, when the called function returns, the returned value is immediately returned from the calling function.

```
int Fact(int n)
{
    if (n == 1)
        return 1;
    else
        return (n * Fact(n-1));
}
```

Figure 7.30 Non-tail recursion

```
int Fact(n)
{
    return Fact1(n, 1);
}
int Fact1(int n, int res)
{
    if (n == 1)
        return res;
    else
      return Fact1(n-1, n*res);
}
```

Figure 7.31 Tail recursion

Pros and Cons of Recursion

Pros

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Follows a divide and conquer technique to solve problems.

Cons

- In general, Recursion is computationally less efficient than iteration due to many function calls & resultant stack operations.
- For some programmers and readers, recursion is a difficult concept.
- **Recursion is implemented using system stack.** If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive process in midstream is slow.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counterpart.
- It is difficult to find bugs, particularly when using global variables.