DATA STRUCTURE ALGORITHMS AND APPLICATIONS (CT- 159)

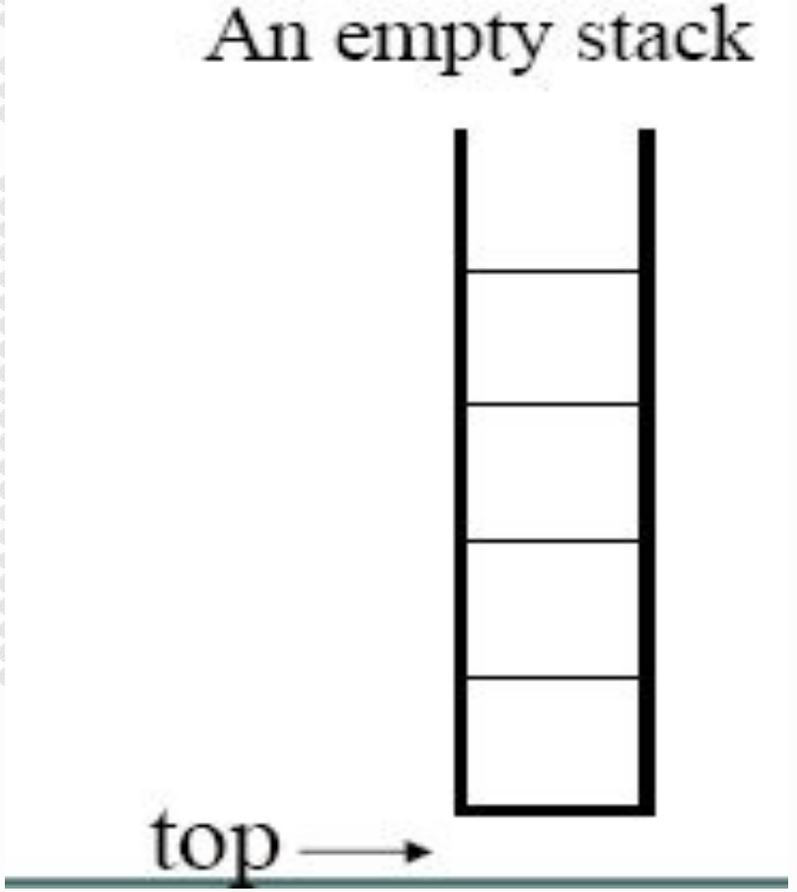
Lecture – 2
Stack and Recursion

Instructor: Engr. Nasr Kamal

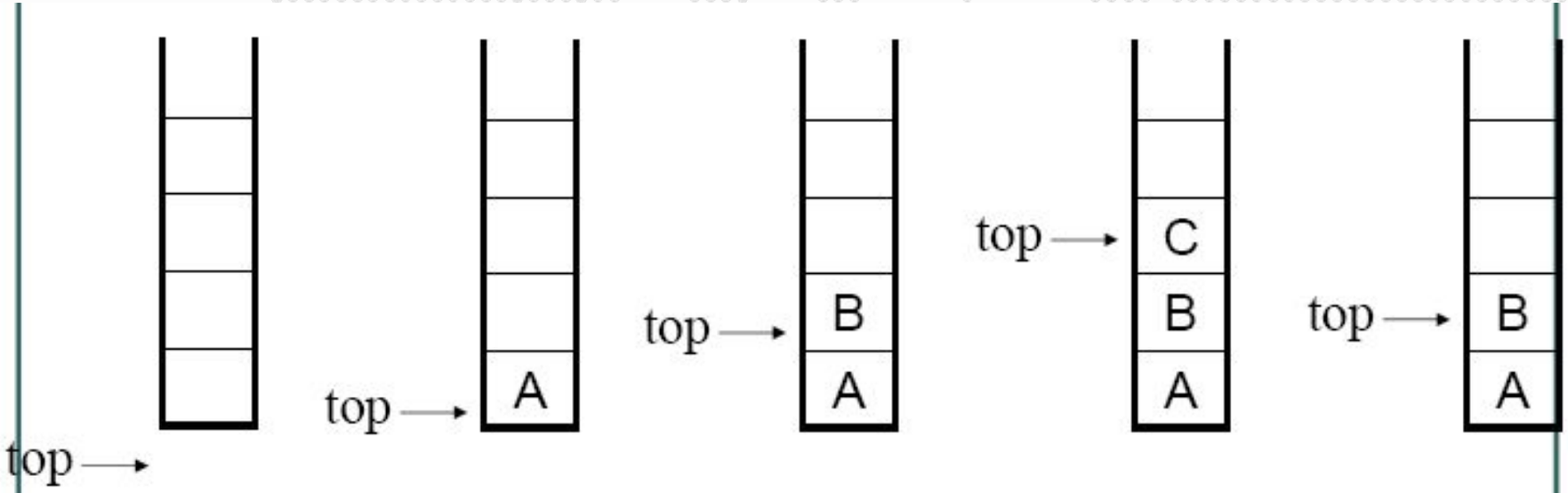
Department of Computer Science and Information Technology

Stack

- A stack is also known as a Last-In-First-Out (LIFO) list.
- A stack is a linear structure in which items are added or removed only at one end the top.
- The depth of stack is the number of elements it contains.
- An empty stack has depth zero.



Stack Example

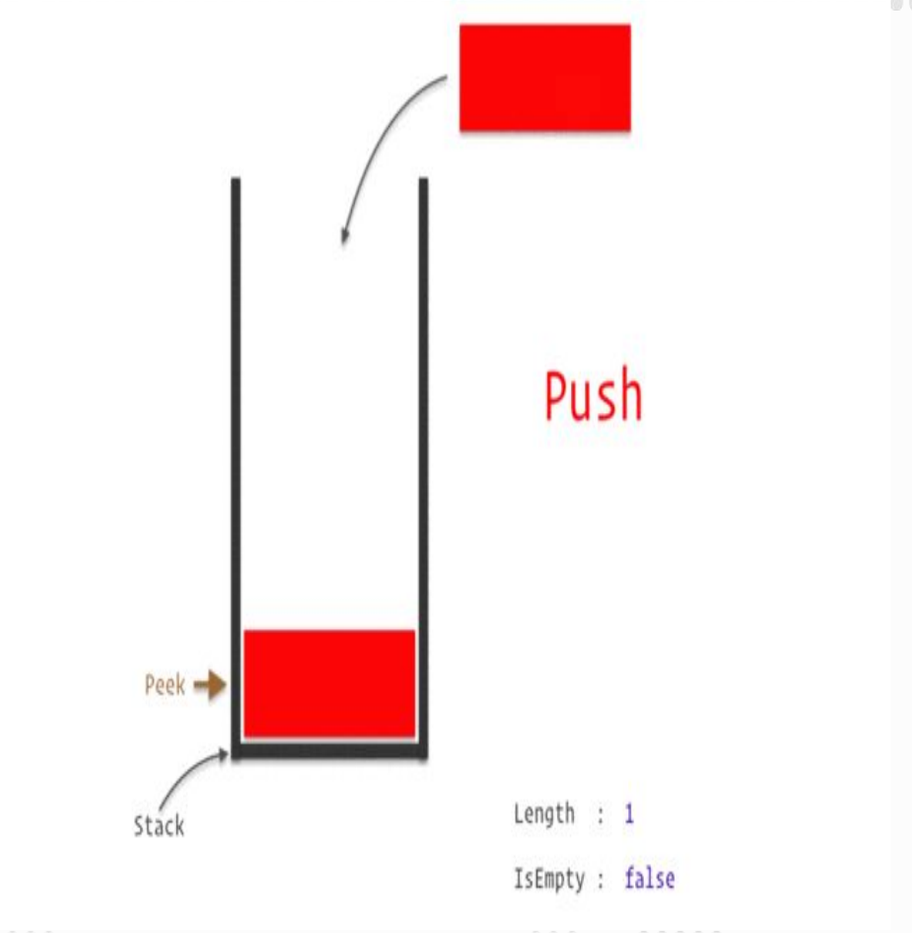


Add the elements A, B, and C to the stack, in that order, then C is the first element we delete from the stack.

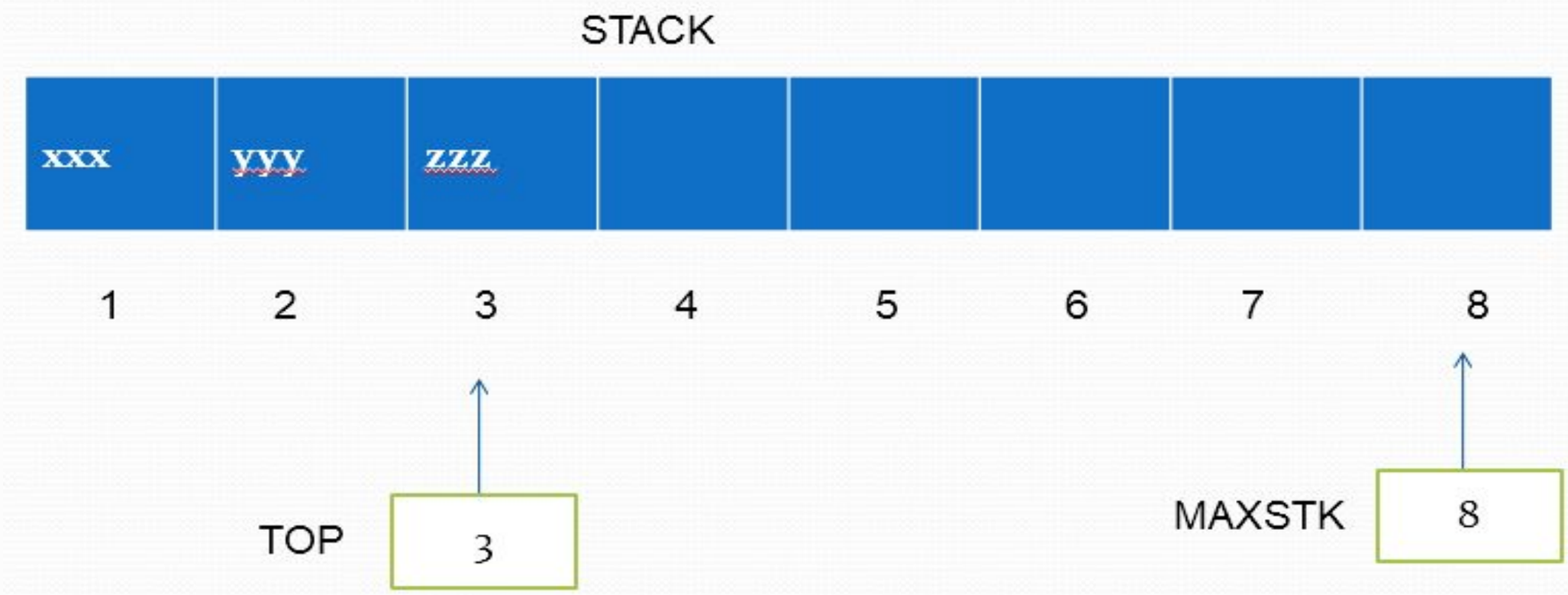
Stack Operations

A stack is an ADT that supports four main methods:

- New():ADT–Creates a new stack
- Push(S:ADT,O:Element):ADT–Inserts object O onto top of stack S.
- Pop(S:ADT):ADT–Removes the top object of stack S; if the stack is empty an error occur.
- Top/peek (S:ADT):element – Returns the top object of the stack, without removing it; if the stack is empty an error occurs.
- The following support methods could also be defined:
- Size(S:ADT):integer–Returns the number of objects in stack S
- IsEmpty(S:ADT):boolean–Indicates f stack S is empty(returns false)



Implementing Stack using Array



PUSH()

This procedure pushes an ITEM onto a stack

PUSH(STACK,TOP,MAXSTK,ITEM)
IF TOP = MAXSTK, then
Print : Overflow, and Return
Set TOP = TOP + 1 [Increase TOP by 1]
Set STACK[TOP]= ITEM [Insert ITEM in new TOP position]
Return

POP()

This procedure deletes the top element of stack and assigns it to the variable ITEM

POP(STACK,TOP,ITEM)
IF TOP = 0, then
Print : Underflow, and Return
Set ITEM = STACK[TOP] [Assigns TOP element to ITEM]
Set TOP= TOP – 1 [Decreases TOP by 1]
Return

Limitations

- The maximum size of the stack must be defined prior and cannot be changed
- Trying to push a new element into a full stack causes an implementation-specific exception

Question 1

- Consider the following stack of characters, where STACK is allocated N=8 memory cells.
- STACK=A,C,D,F,K,_,_,_
- Describe the stack as the following operations take place.
- POP(STACK,ITEM)
- POP(STACK,ITEM)
- PUSH(STACK,L)
- PUSH(STACK,P)
- POP(STACK,ITEM)
- PUSH(STACK,R)
- PUSH(STACK,S)
- POP(STACK,ITEM)

Question 2

Suppose STACK is allocated N=6 memory cells and initially STACK is empty. Find the output of the following procedure.

```
Set A=2 and B=5
 Call PUSH(STACK,A)
      Call PUSH(STACK,4)
     Call PUSH(STACK,B+2)
      Call PUSH(STACK,9)
      Call PUSH(STACK,A+B)
Repeat while Top!= 0
     Call POP(STACK,ITEM)
End of loop
```

Recursion

A function is said to be recursively defined, if a function containing either a Call statement to itself or a Call statement to a second function that may eventually result in a Call statement back to the original function.

Key Properties of a Recursive Function:

- Base Case (Stopping Condition):There must be a specific condition under which the function does not call itself anymore. This is called the base case. The base case acts as a stopping point for the recursion to prevent it from going on forever.
- Progress Towards the Base Case: Every time the function calls itself (either directly or indirectly through another function), it must do so with an argument that brings it closer to reaching the base case. This ensures that eventually, the base case will be met, and the recursion will stop.

RECURSION

Recursion – A Simple Example

Imagine you have a function to count down numbers from a given number to 1.

```
void countdown(int n)
{
if (n <= 0)
{
 // Base Case: Stop when n is 0 or less return;
}
 cout << n << " "; countdown(n - 1); // Recursive call with n-1, getting closer to the base case
}
```

Base Case: The function stops when n becomes 0 or less.
Progress Towards Base Case: Each time countdown calls itself, it reduces n by 1, getting closer to the base case.

Recursion

- In some problems, it may be natural to define the problem in terms of the problem itself.
- Recursion is useful for problems that can be represented by a simpler version of the same problem.

Example: the factorial function

6! = 6 * 5 * 4 * 3 * 2 * 1

We could write:

6! = 6 * 5!

Example 1: Factorial Function

In general, we can express the factorial function as follows:

n! = n * (n–1)!

The factorial function is only defined for positive integers. So we should be a bit more precise:

if n<=1, then n! = 1
if n>1, then n! = n * (n–1)!

The C++ equivalent of this definition:

```
int fac(int numb){
  if(numb<=1)
    return 1;
  else
    return numb * fac(numb–1);
}
```

recursion **means that a function calls itself**

Factorial Function

Assume the number typed is 3, that is, numb=3.

fac(3) :

3 <= 1 ? No.
fac(3) = 3 * fac(2)

fac(2) :
2 <= 1 ? No.
fac(2) = 2 * fac(1)

fac(1) :
1 <= 1 ? Yes.
return 1

fac(2) = 2 * 1 = 2
return fac(2)

fac(3) = 3 * 2 = 6
return fac(3)

fac(3) has the value 6

```
int fac(int numb){
   if(numb<=1)
     return 1;
   else
     return numb * fac(numb-1);
}
```

Factorial Function

For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. Compare the recursive solution with the iterative solution:

Recursive Solution	Iterative Solution

int fac(int numb){
  if(numb<=1)
  return 1;
  else
  return numb*fac(numb-1);
}
	int fac(int numb){
	  int product=1;
	  while(numb>1){
	  product *=numb;
	  numb--;
	  }
	  return product;
	}

Recursion

To trace recursion, recall that function calls operate as a stack – the new function is put on top of the caller

We have to pay a price for recursion:

- calling a function consumes more time and memory than adjusting a loop counter.
- high performance applications (graphic action games, simulations of nuclear explosions) hardly ever use recursion.

In less demanding applications recursion is an attractive alternative for iteration (for the right problems!)

Recursion

If we use iteration, we must be careful not to create an infinite loop by accident:

```
for(int incr=1; incr!=10;incr+=2)
  ...
```

```
int result = 1;
 while(result >0){
  ...
  result++;
 }
```

Recursion

Similarly, if we use recursion we must be careful not to create an infinite chain of function calls:

```
int fac(int numb){
  return numb * fac(numb-1);
}
```

Or:

```
int fac(int numb){
  if (numb<=1)
    return 1;
  else
    return numb * fac(numb+1);
}
```

No termination condition

Recursion

We must always make sure that the recursion bottoms out:

- A recursive function must contain at least one non-recursive branch.
- The recursive calls must eventually lead to a non-recursive branch.

Recursion

- Recursion is one way to decompose a task into smaller subtasks. At least one of the subtasks is a smaller example of the same task.
- The smallest example of the same task has a non-recursive solution.

  Example: The factorial function

  n! = n * (n-1)! and 1! = 1

Example 2: Binary Search

Search for an element in an array

- Sequential search
- Binary search

Binary search

- Compare the search element with the middle element of the array
- If not equal, then apply binary search to half of the array (if not empty) where the search element would be.

Binary Search with Recursion

```
// Searches an ordered array of integers using recursion
int bsearchr(const int data[], // input: array
             int first,       // input: lower bound
             int last,        // input: upper bound
             int value         // input: value to find
        )// output: index if found,  otherwise return –1

{  int middle = (first + last) / 2;
   if (data[middle] == value)
      return middle;
   else if (first >= last)
      return –1;
   else if (value < data[middle])
      return bsearchr(data, first, middle–1, value);
   else
      return bsearchr(data, middle+1, last, value);
}
```

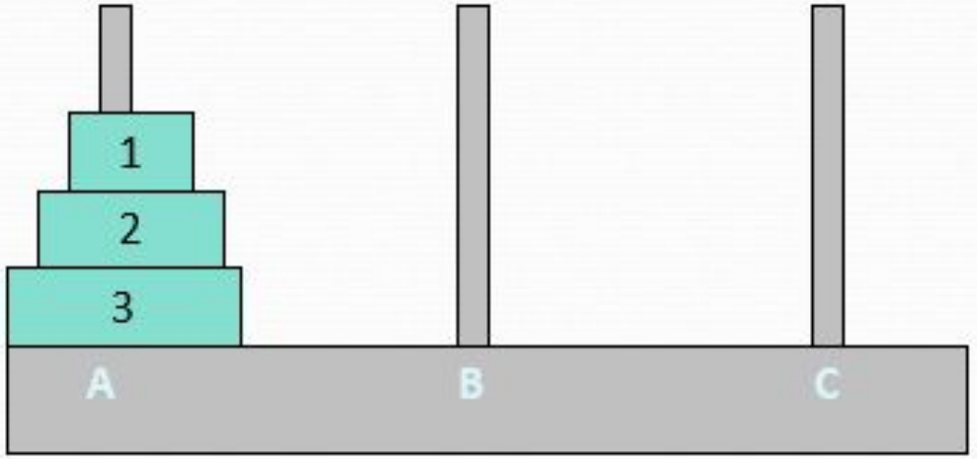
Binary Search with Recursion

```
int main() {
    const int array_size = 8;
    int list[array_size]={1, 2, 3, 5, 7, 10, 14, 17};
    int search_value;

    cout << "Enter search value: ";
    cin >> search_value;
    cout << bsearchr(list,0,array_size-1,search_value)
         << endl;

    return 0;
}
```

Example 3: Tower of Hanoi



- Only one disc could be moved at a time
- A larger disc must never be stacked above a smaller one
- One and only one extra needle could be used for intermediate storage of discs

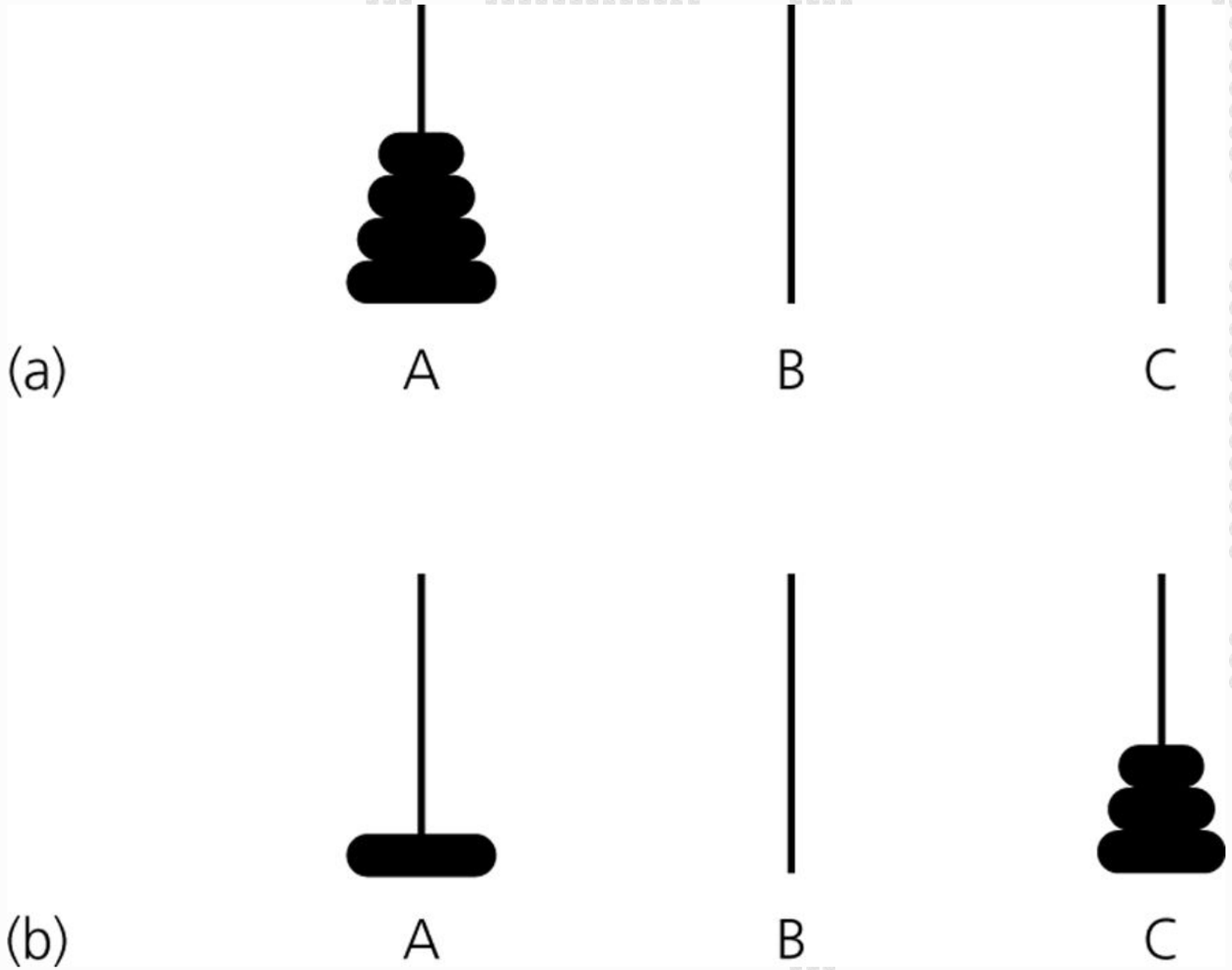
Tower of Hanoi

- From the moves necessary to transfer one, two, and three disks, we can find a recursive pattern – a pattern that uses information from one step to find the next step.

- If we want to know how many moves it will take to transfer 64 disks from post A to post C, we will first have to find the moves it takes to transfer 63 disks, 62 disks, and so on.

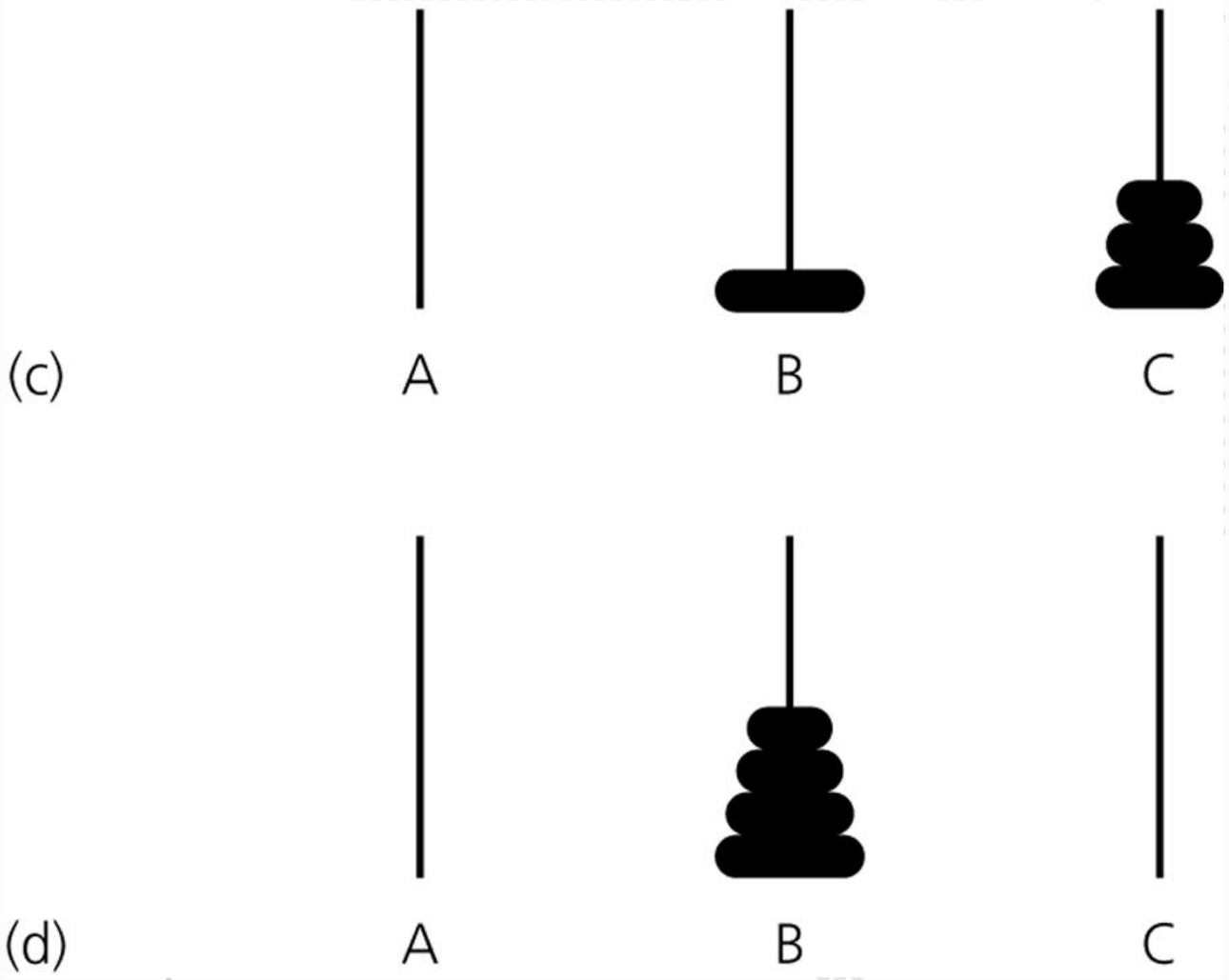
Tower of Hanoi

a) The initial state; b) move $n - 1$ disks from A to C



Tower of Hanoi

c) move one disk from *A* to *B*; d) move *n* – 1 disks from *C* to *B*



Tower of Hanoi

- The recursive pattern *can* help us generate more numbers to find an *explicit* (non-recursive) pattern. Here's how to find the number of moves needed to transfer larger numbers of disks from post A to post C, when M = the number of moves needed to transfer n-1 disks from post A to post C:
- for **1 disk** it takes 1 move to transfer 1 disk from post A to post C;
- for **2 disks**, it will take 3 moves: 2M + 1 = 2(**1**) + 1 = **3**
- for **3 disks**, it will take 7 moves: 2M + 1 = 2(**3**) + 1 = **7**
- for **4 disks**, it will take 15 moves: 2M + 1 = 2(**7**) + 1 = **15**
- for **5 disks**, it will take 31 moves: 2M + 1 = 2(**15**) + 1 = **31**
- for **6 disks**... ?

Tower of Hanoi

Number of Disks (n)	Number of Moves
1	$2^1 - 1 = 2 - 1 = 1$
2	$2^2 - 1 = 4 - 1 = 3$
3	$2^3 - 1 = 8 - 1 = 7$
4	$2^4 - 1 = 16 - 1 = 15$
5	$2^5 - 1 = 32 - 1 = 31$
6	$2^6 - 1 = 64 - 1 = 63$

- So the formula for finding the number of steps it takes to transfer n disks from post A to post C is:

$$2^n - 1$$