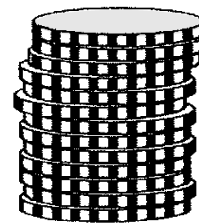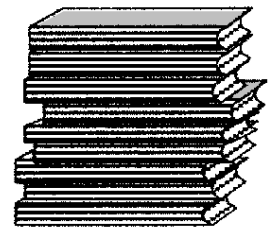# Stacks

Mohammad Asad Abbasi

Lecture 4

Stack

What is a Stack?

- A method of organizing data
- Defined structure and operations

➤ A stack is a data structure of ordered items such that items can be inserted and removed only at one end.

➤ Stacks typically used for temporary storage of data.

➤ A stack is a LIFO (Last-In/First-Out) data structure.

➤ A stack is sometimes also called a pushdown store.
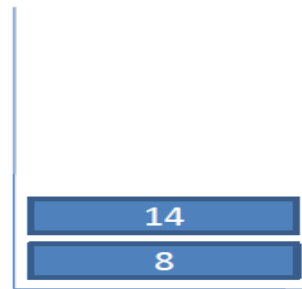
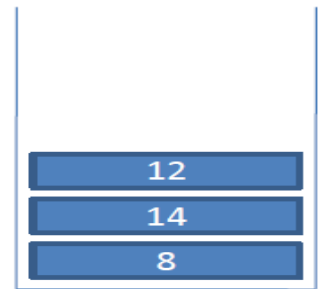Stack of coins   Stack of books
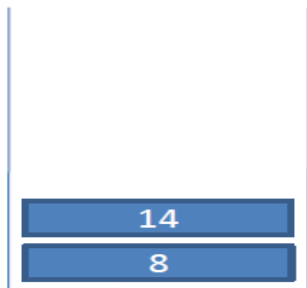
Stack Example

empty stack
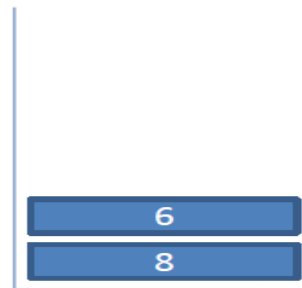
push 8

push 14

push 12

pop 12

pop 14

push 6

pop 6
pop 8

Stack

➢ **Problem:**

- What happens if we try to pop an item off the stack when the stack is empty?

  - This is called a stack underflow. The pop method needs some way of telling us that this has happened.

Stack Operations

➢ Main stack operations:
- **push(object):** inserts an element.
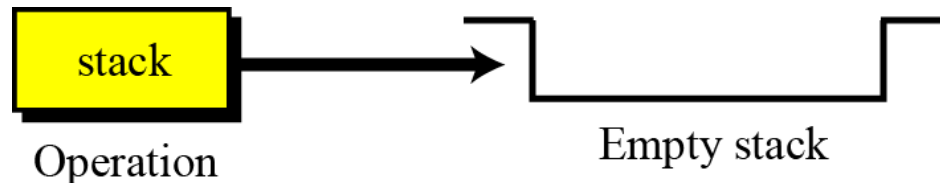- **object pop():** removes and returns the last inserted element.

➢ Auxiliary stack operations:
- **object top():** returns the last inserted element without removing it.
- **integer size():** returns the number of elements stored.
- **boolean isEmpty():** indicates whether no elements are stored.

The stack operation

➢ The stack operation creates an empty stack. The following shows the format.

The push operation

➢ The push operation inserts an item at the top of the stack. The following shows the format.

push (stackName, dataItem)

Data to push

| 30 |

| Push | → Operation

Top element ⟶ | 78 |
| 20 |

Stack before push

| 30 | ⟶ Top element
| 78 |
| 20 |

Stack after push
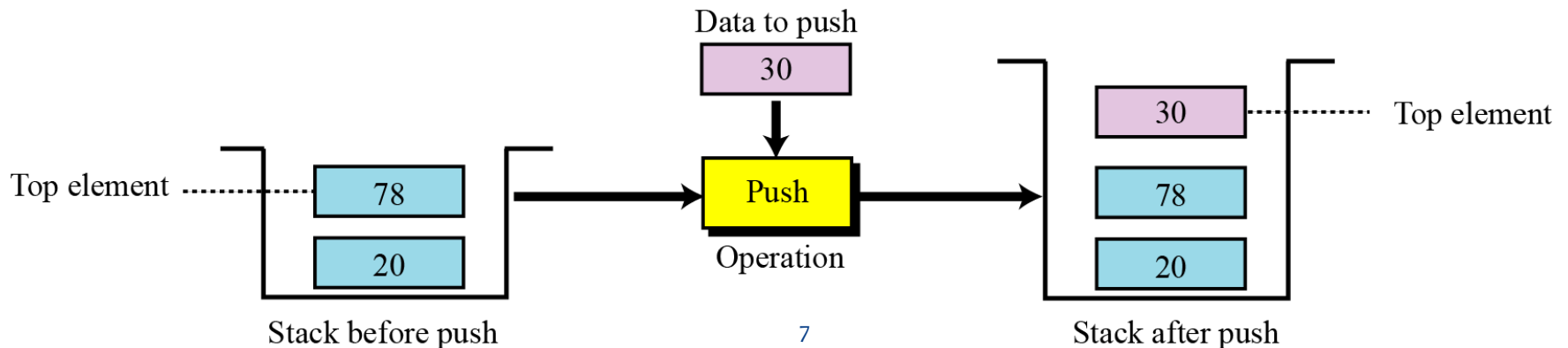
The pop operation

➢ The pop operation deletes the item at the top of the stack. The following shows the format.

pop (stackName, dataItem)

Popped data

30

Top element ---------- 30

78   Pop   78 ---------- Top element

20   Operation   20

Stack before pop   Stack after pop
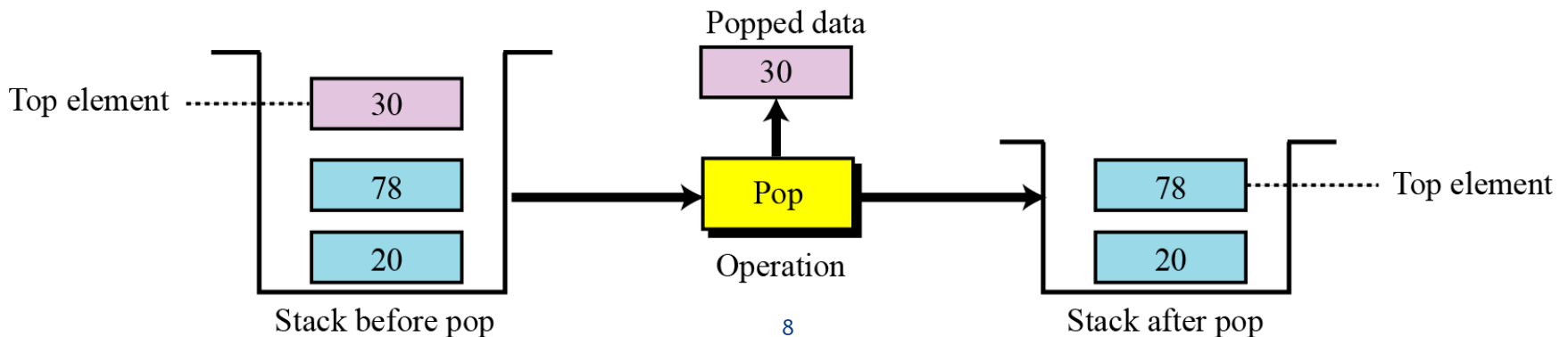
The empty operation

➢ The empty operation checks the status of the stack.

➢ This operation returns true if the stack is empty and false if the stack is not empty.

empty (stackName)

Example

> A segment of an algorithm that applies the previously defined operations on a stack S.
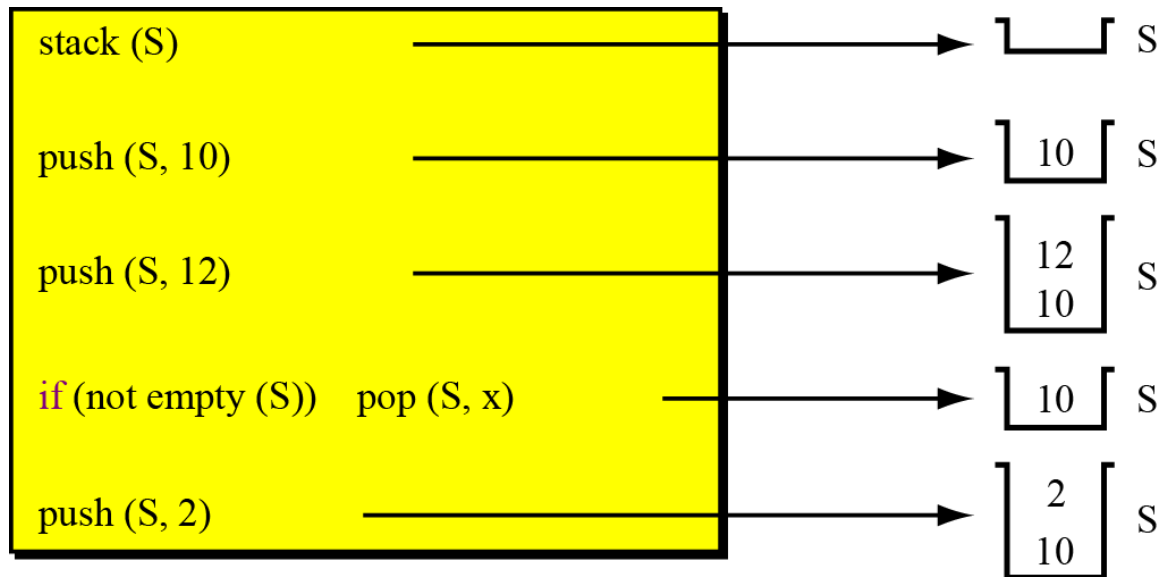
An algorithm segment

Selecting storage structures

➢ Two choices

- Select position 0 as top of the stack
- Select position 0 as bottom of the stack
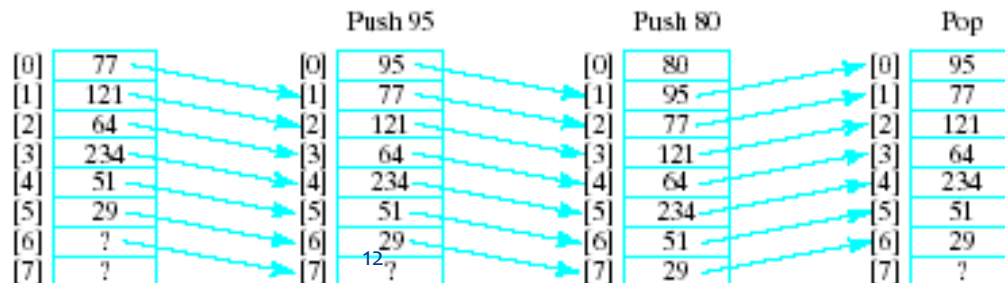
Select position 0 as top of the stack

➤ Model with an array

- Let position 0 be top of stack

[0]	77
[1]	121
[2]	64
[3]	234
[4]	51
[5]	29
[6]	?
[7]	?

➤ Problem … consider pushing and popping

- Requires much shifting

|			Push 95			Push 80			Pop	
[0]	77	[0]	95	[0]	80	[0]	95			
[1]	121	[1]	77	[1]	95	[1]	77			
[2]	64	[2]	121	[2]	77	[2]	121			
[3]	234	[3]	64	[3]	121	[3]	64			
[4]	51	[4]	234	[4]	64	[4]	234			
[5]	29	[5]	51	[5]	234	[5]	51			
[6]	?	[6]	29	[6]	51	[6]	29			
[7]	?	[7]	?	[7]	29	[7]	?			

12

Select position 0 as bottom of the stack

➤ A better approach is to let position 0 be the <u>bottom</u> of the stack
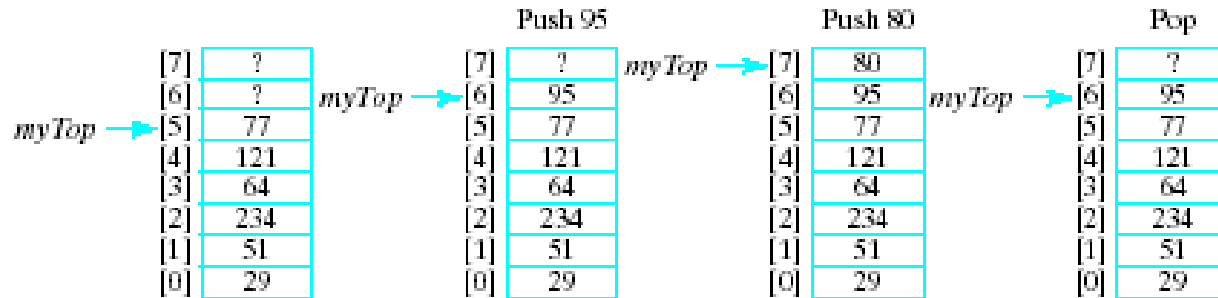
➤ Thus our design will include
- An array to hold the stack elements
- An integer to indicate the top of the stack

Implementing a Stack

➤ There are two ways we can implement a stack:

- Static implementation (Using an array)
- Dynamic implementation (Using a linked list)

a. ADT

b. Array implementation

c. Linked list implemenation
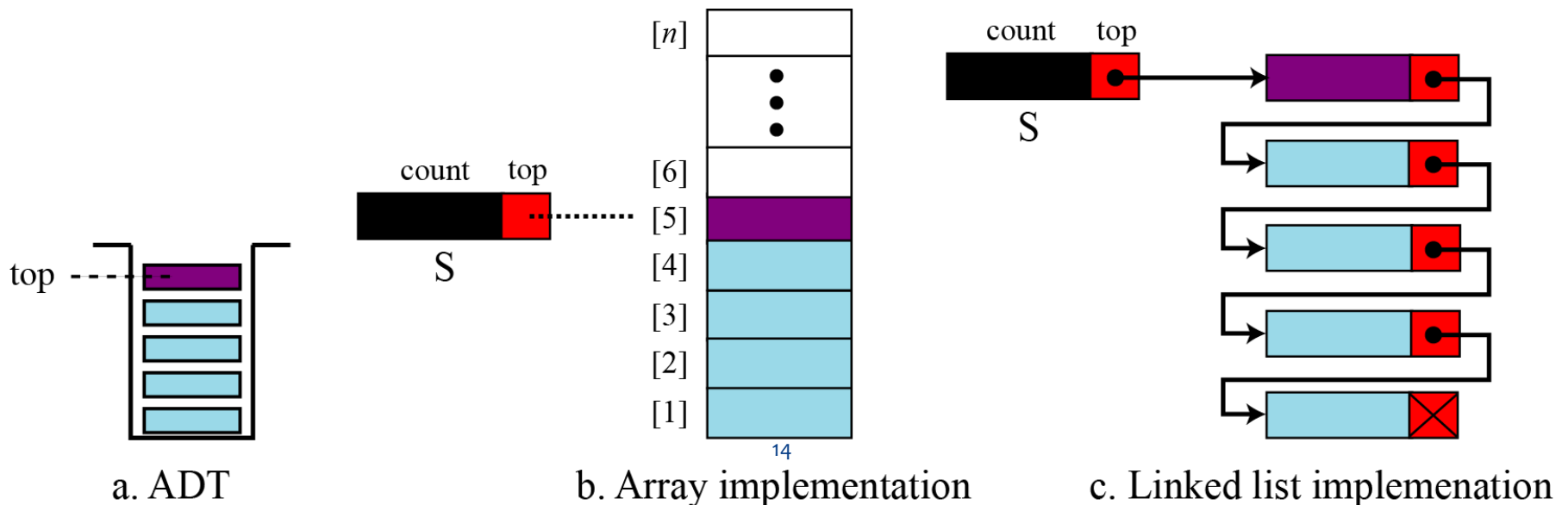
Stack (Array Implementation)

➢ **Algorithm for push:**

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

1. If TOP = SIZE − 1, then:
    (a) Display "The stack is in overflow condition"
    (b) Exit
2. TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit

Stack (Array Implementation)

```
1.    //This function will add/insert an element to Top of the stack
2.    void push()
3.    {
4.    int item;
5.    if (Top == MAXSIZE–1)        //if the top pointer already reached the maximum allowed size then
6.    {                           //we can say that the stack is full or overflow

7.    printf("\nThe Stack Is Full");
8.    getch();
9.    }                            //Otherwise an element can be added or inserted by incrementing the
                                   //stack pointer Top as follows

10.   else
11.   {
12.   printf("\nEnter The Element To Be Inserted = ");
13.   scanf("%d",&item);
14.   top++;
15.    stack[top] = item;
            }
            }
```
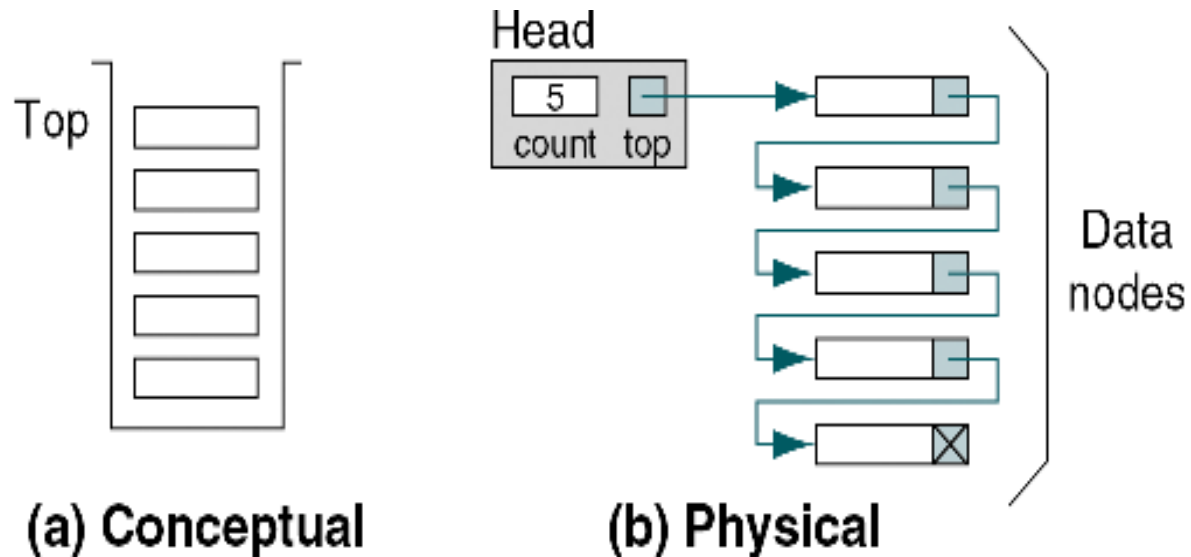
Stack (Array Implementation)

➢ **Algorithm for pop:**

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

1. If TOP < 0, then
       (a) Display "The Stack is empty"
       (b) Exit
2. Else remove the Top most element
3. DATA = STACK[TOP]
4. TOP = TOP – 1
5. Exit

# Stack (Array Implementation)

1. //This function will delete an element from the Top of the stack
2. void pop()
3. {
4. int item;   //If the Top pointer points to NULL, then the stack is empty
5. if (Top == -1)   // that is NO element is there to delete or pop
6. printf("\nThe Stack Is Empty");
7. //Otherwise the top most element in the stack is popped or deleted by decrementing the Top pointer
8. else
9. {
10. item= stack[top];
11. Top=top-1;
12. printf("\nThe Deleted Element Is = %d",item);
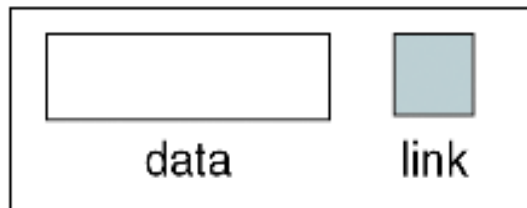13. }
14. }

Stack (Linked List Implementation)

(a) Conceptual   (b) Physical

Conceptual and Physical Stack Implementations

Stack (Linked List Implementation)
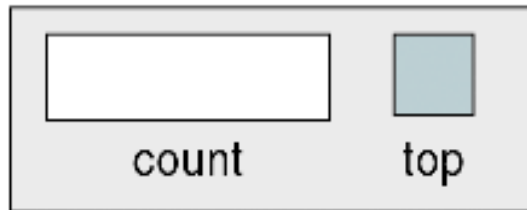
count   top

Stack head structure

data   link

Stack node structure

```
stack
  count
  top
end stack


node
  data
  link
end node
```

Stack Data Structure

ALGORITHM 3-1 Create Stack

```
Algorithm createStack
Creates and initializes metadata structure.
   Pre      Nothing
   Post     Structure created and initialized
   Return stack head
1 allocate memory for stack head
2 set count to 0
3 set top to null
4 return stack head
end createStack
```
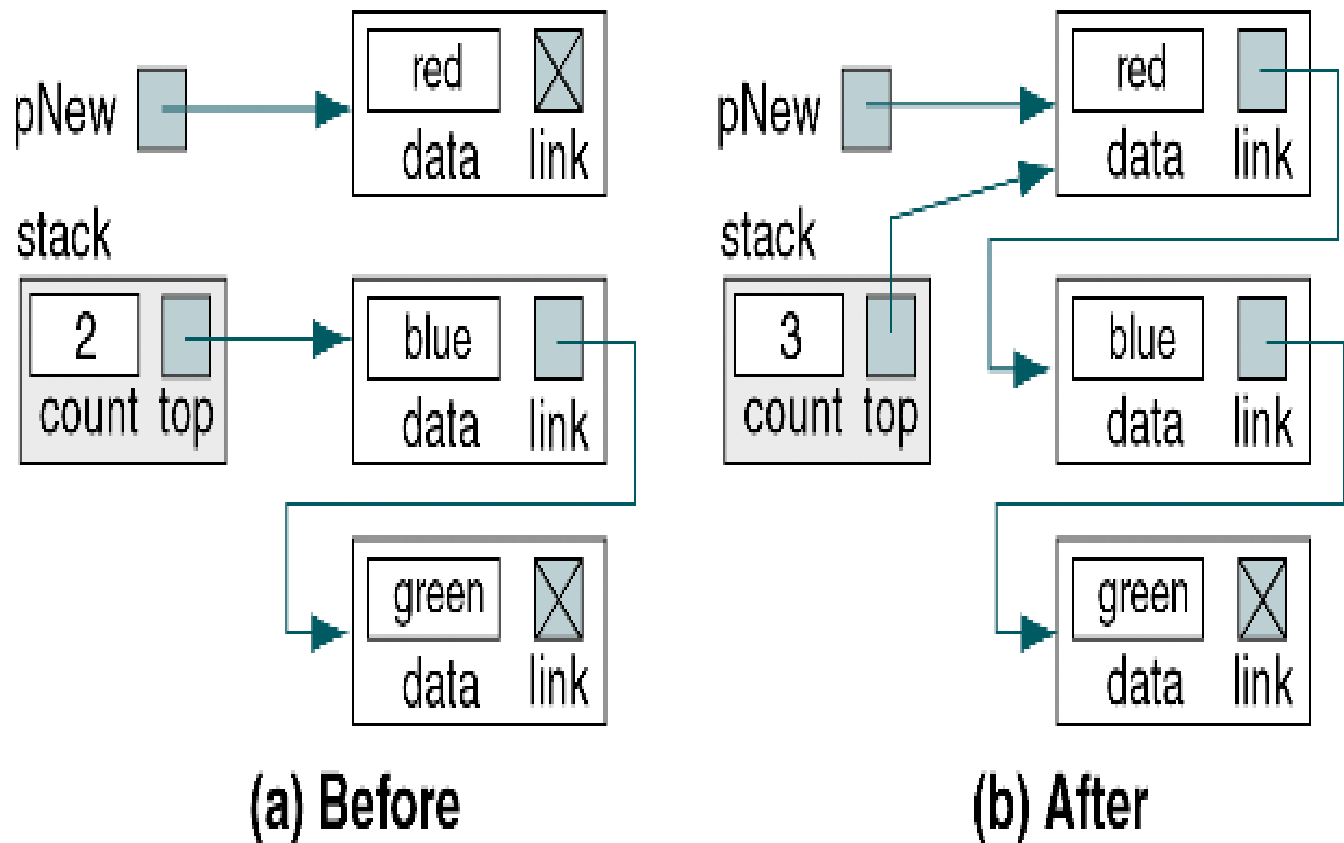
FIGURE 3-9 Push Stack Example

ALGORITHM 3-2   Push Stack Design

```
Algorithm pushStack (stack, data)
Insert (push) one item into the stack.
   Pre  stack passed by reference
        data contain data to be pushed into stack
   Post data have been pushed in stack
1 allocate new node
2 store data in new node
3 make current top node the second node
4 make new node the top
5 increment stack count
end pushStack
```

Push()

```
1.    void push()
2.    {
3.    struct node *ptr;
4.    printf("\n\nEnter ITEM: ");
5.    scanf("%d", &item);
6.    if (top == NULL)
7.    {
8.    top = (struct node *)malloc(sizeof(struct node));
9.    top->info = item;
10.   top->link = NULL;
11.   }
12.   else
13.   {
14.   ptr = top;
15.   top = (struct node *)malloc(sizeof(struct node));
16.   top->info = item;
17.   top->link = ptr;
18.   }
19.   printf("\nItem inserted: %d\n", item);
20.   }
```
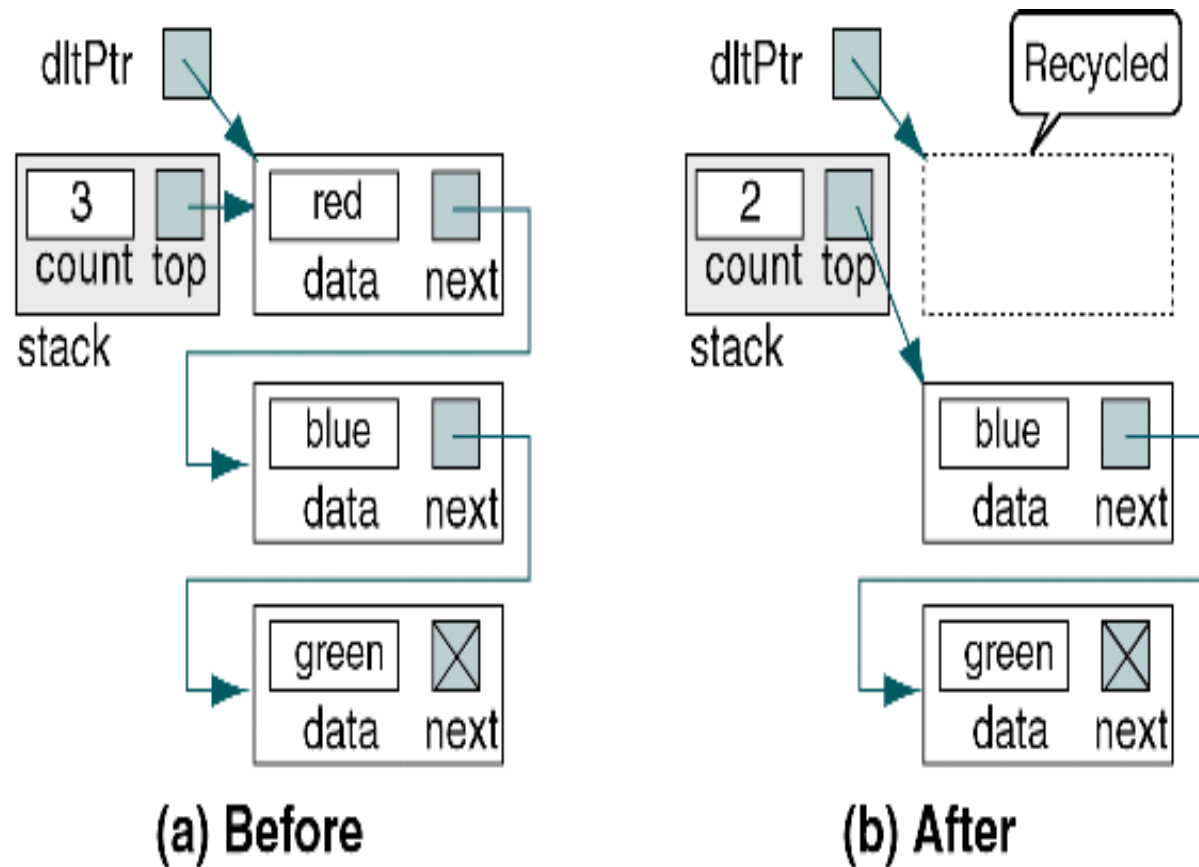
FIGURE 3-10 Pop Stack Example

ALGORITHM 3-3 Pop Stack

```
Algorithm popStack (stack, dataOut)
This algorithm pops the item on the top of the stack and
returns it to the user.
   Pre     stack passed by reference
           dataOut is reference variable to receive data
   Post    Data have been returned to calling algorithm
   Return true if successful; false if underflow
1  if (stack empty)
   1   set success to false
2  else
   1   set dataOut to data in top node
   2   make second node the top node
   3   decrement stack count
   4   set success to true
3  end if
4  return success
end popStack
```

Pop()

1. void pop()
2. {
3. struct node *ptr;
4. if (top == NULL)
5. printf("\n\nStack is empty\n");
6. else
7. {
8. ptr = top;
9. item = top->info;
10. top = top->link;
11. free(ptr);
12. printf("\n\nItem deleted: %d", item);
13. }
14. }

Traversing

```
1.  void display()
2.  {
3.  struct node *ptr;
4.  if (top == NULL)
5.  printf("\n\nStack is empty\n");
6.  else
7.  {
8.  ptr = top;
9.  while(ptr != NULL)
10. {
11. printf("\n\n%d",  ptr->info);
12. ptr = ptr->link;
13. }
```

Applications of Stacks

➢ Stack applications can be classified into four broad categories: *reversing data*, *pairing data*, *postponing data usage* and *backtracking steps*.

➢ Direct applications

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Chain of method calls in the Java Virtual Machine or C++ runtime environment
- Program execution
- Parsing
- Evaluating postfix expressions

➢ Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Reversing data items

➢ Reversing data items requires that a given set of data items be reordered so that the first and last items are exchanged, with all of the positions between the first and last also being relatively exchanged. For example, the list (2, 4, 7, 1, 6, 8) becomes (8, 6, 1, 7, 4, 2).

Pairing data items

➤ The following two expressions are evaluated differently because of the parentheses in the second expression:

$$3 \times 6 + 2 = 20 \qquad\qquad 3 \times (6 + 2) = 24$$

➤ The compiler uses a stack to check that all opening parentheses are paired with a closing parentheses.

Example

➢ Algorithm to check if all opening parentheses are paired with a closing parenthesis.

Algorithm: **CheckingParentheses** (expression)

Purpose: Check the pairing of parentheses in an expression

Pre: Given the expression to be checked

Post: Error messages if unpaired parentheses are found

Return: None
{
    stack (S)
    while (more character in the expression)
    {
        Char ← next character
        if (Char = '(')   push (S, Char)
        else
        {

Example (Continued)

```
                    if (Char = ')')
                    {
                            if (empty (S))      print (unmatched opening parenthesis)
                            else                pop (S, x)

                    }
            }
        }
    if (not empty (S))          print (a closing parenthesis not matched)
    return
}
```

Applications of Stacks

➢ **RECURSION**

▪ Recursion occurs when a function is called by itself repeatedly; the function is called recursive function. The general algorithm model for any recursive function contains the following steps:

1. *Prologue:* Save the parameters, local variables, and return address.

2. *Body:* If the base criterion has been reached, then perform the final computation and go to step 3; otherwise, perform the partial computation and go to step 1 (initiate a recursive call).

3. *Epilogue:* Restore the most recently saved parameters, local variables, and return address.
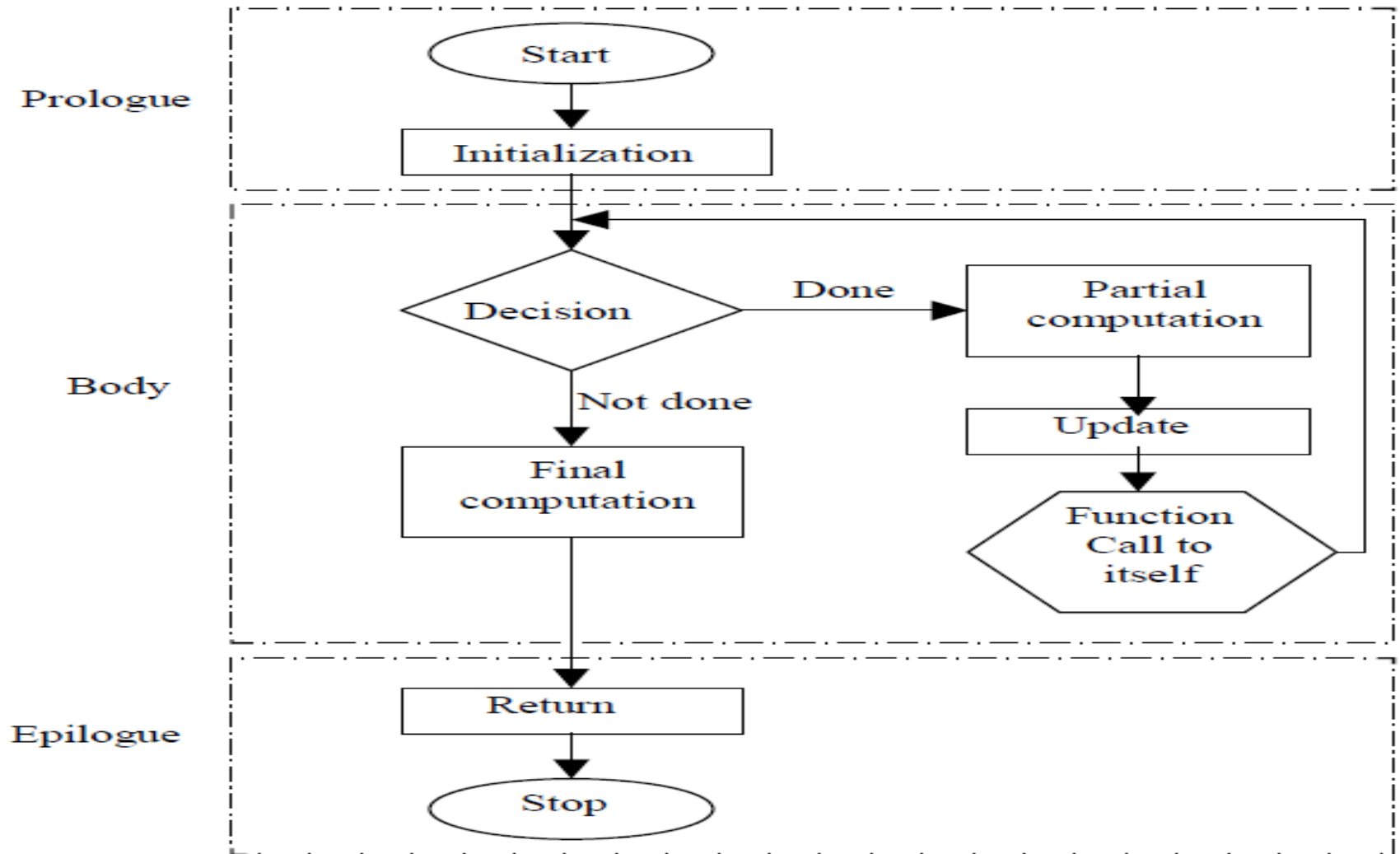
RECURSION

Fig. 3.2. Flowchart model for a recursive algorithm
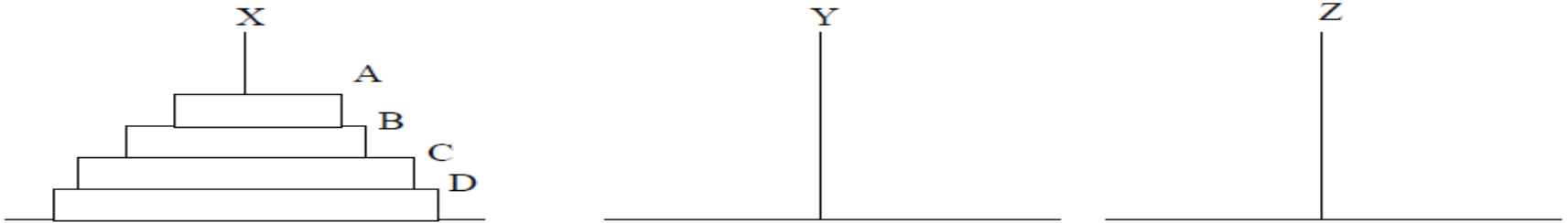
TOWER OF HANOI

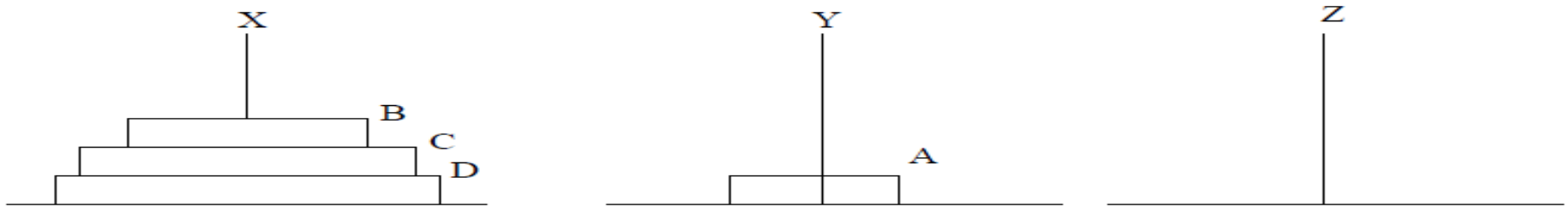Fig. 3.4. Initial position of the Tower of Hanoi

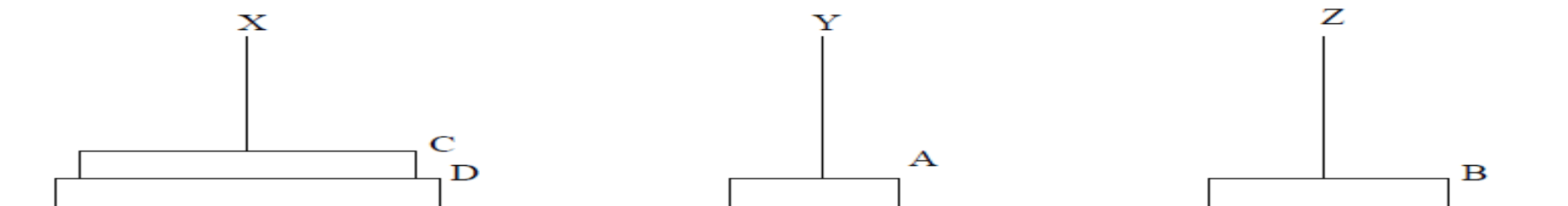Fig. 3.5. Move disk A from the peg X to peg Y

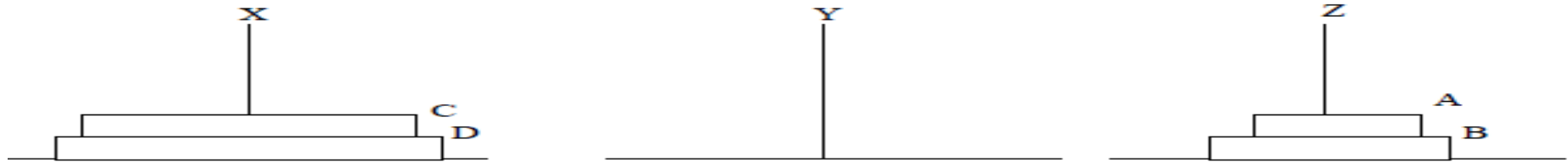Fig. 3.6. Move disk B from the peg X to peg Z

TOWER OF HANOI

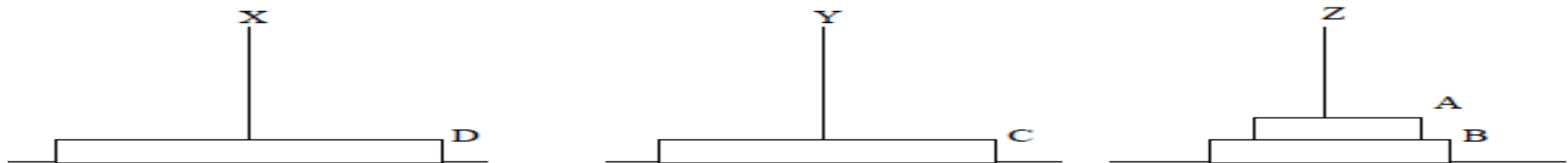Fig. 3.7. Move disk A from the peg Y to peg Z

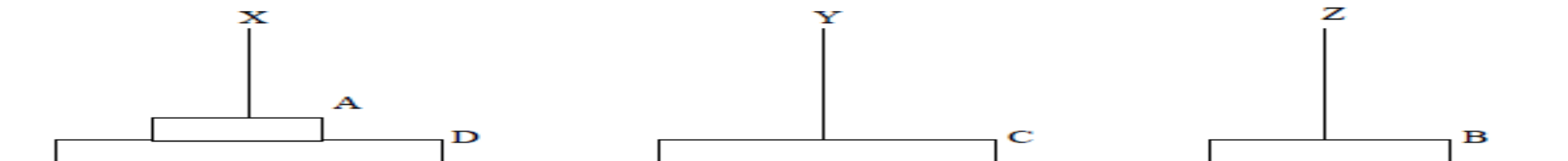Fig. 3.8. Move disk C from the peg X to peg Y

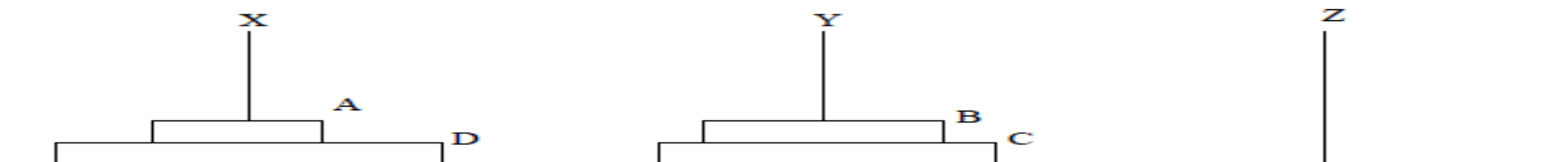Fig. 3.9. Move disk A from the peg Z to peg X

Fig. 3.10. Move disk B from the peg Z to peg Y

TOWER OF HANOI

Fig. 3.11. Move disk A from the peg X to peg Y

Fig. 3.12. Move disk D from the peg X to peg Z

Fig. 3.13. Move disk A from the peg Y to peg Z

Fig. 3.14. Move disk B from the peg Y to peg X

TOWER OF HANOI
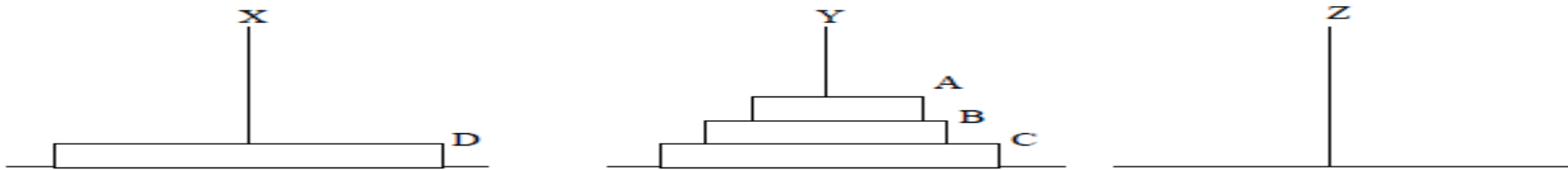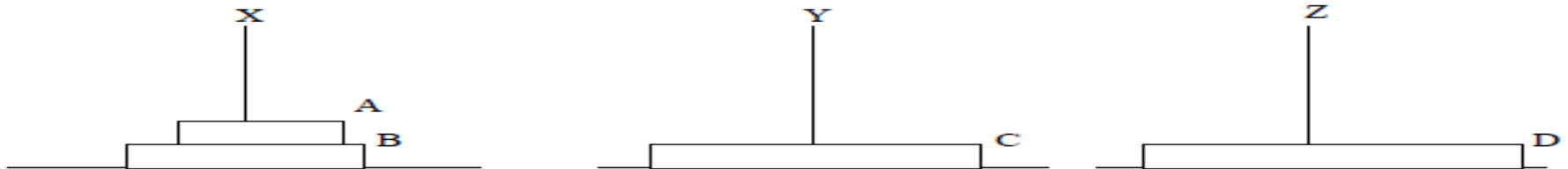

Fig. 3.15. Move disk A from the peg Z to peg X
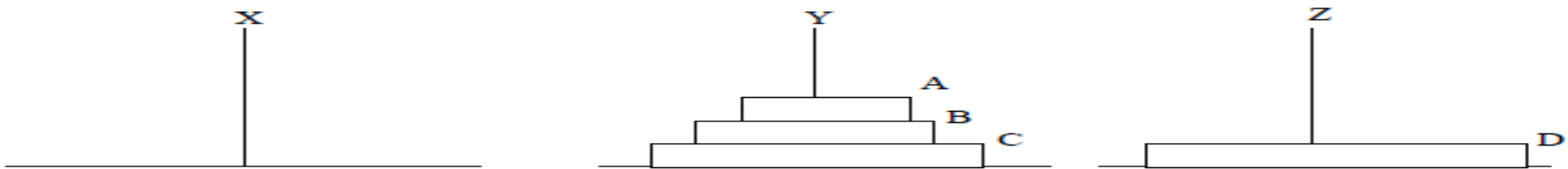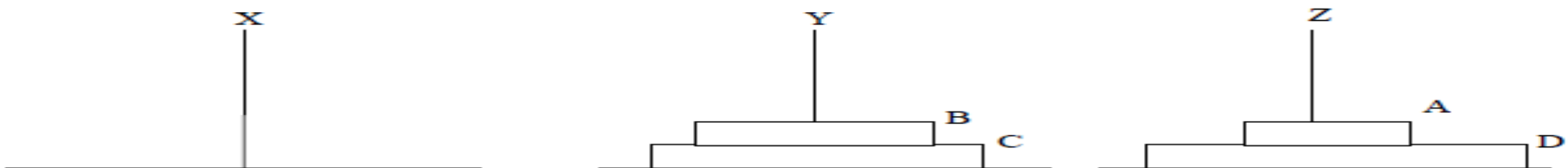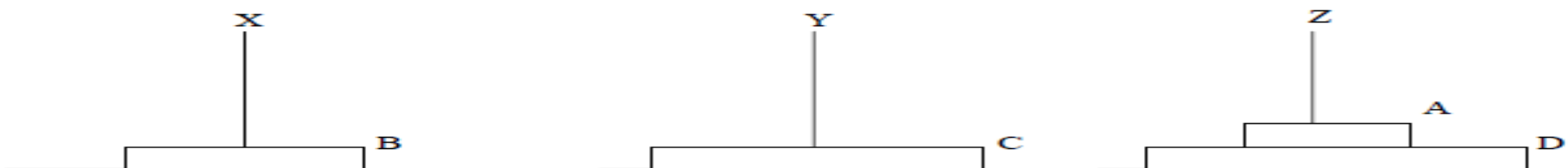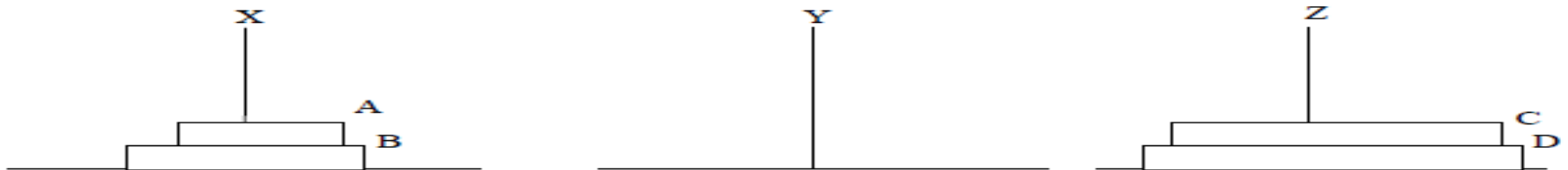

Fig. 3.16. Move disk C from the peg Y to peg Z
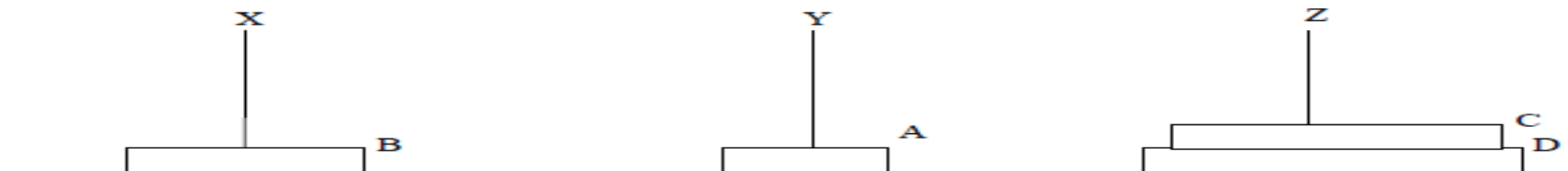
Fig. 3.17. Move disk A from the peg X to peg Y
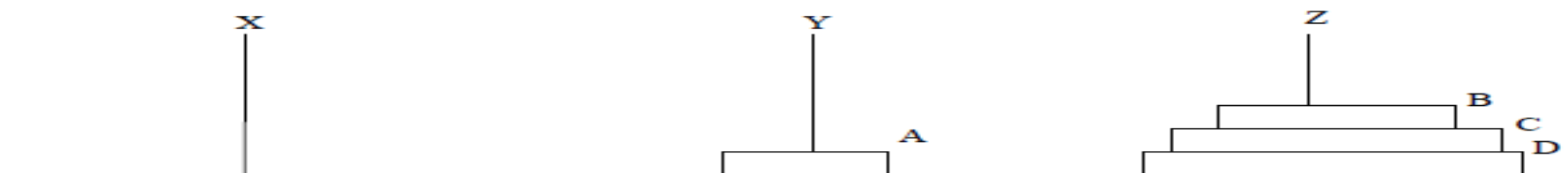
Fig. 3.18. Move disk B from the peg X to peg Z
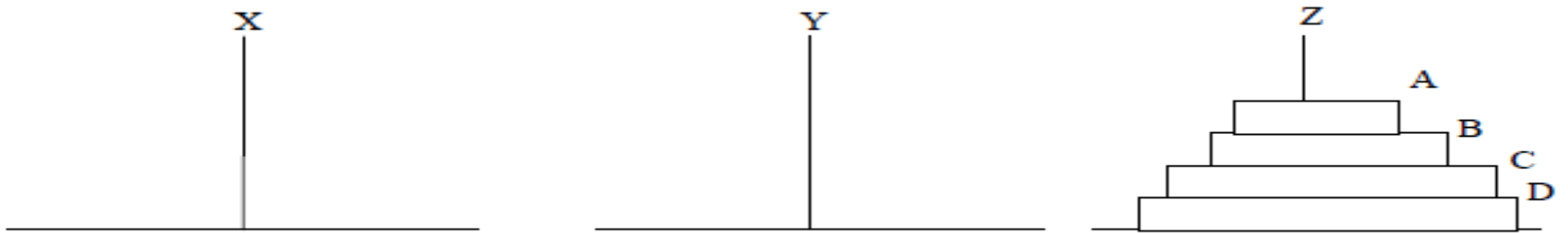
TOWER OF HANOI

Fig. 3.19. Move disk A from the tower Y to tower Z

TOWER OF HANOI

➤ Recursive Solution for the Tower of Hanoi with algorithm

▪ Let's call the three pegs Src(Source), Aux(Auxiliary) and st(Destination).

1) Move the top N – 1 disks from the Source to Auxiliary tower
2) Move the Nth disk from Source to Destination tower
3) Move the N – 1 disks from Auxiliary tower to Destination tower. Transferring the top N – 1 disks from Source to Auxiliary tower can again be thought of as a fresh problem and can be solved in the same manner. So once you master solving Tower of Hanoi with three disks, you can solve it with any number of disks with the above algorithm.

TOWER OF HANOI

➤ **Explicit Pattern**

▪ Number of Disks   Number of Moves

Number of Disks	Number of Moves
1	1
2	3
3	7
4	15
5	31

▪ *Powers of two help reveal the pattern:*

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	$2^5 - 1 = 32 - 1 = 31$

Evaluation of Expressions

- The representation and evaluation of expressions is of great interest to computer scientists.

  - *x = a/b - c+d*e - a*c*

- If we examine expression, we notice that they contains:

  - operators: ==, +, -, ||, &&, !

  - operands: a,b,c…

  - parentheses: ( )

- How to generate the machine instructions corresponding to a given expression?

  precedence rule + associative rule

Precedence hierarchy and associativity

Token	Operator	Precedence[1]	Associativity
() [] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! ~ - + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right-to-left
,	comma	1	left-to-right

1. The precedence column is taken from Harbison and Steele.
2. Postfix form
3. Prefix form

Figure 3.12: Precedence hierarchy for C

EXPRESSION

- ➢ **1. Infix notation**

- ▪ The *infix notation* is what we come across in our general mathematics, where the operator is written in-between the operands. For example : The expression to add two numbers A and B is written in infix notation as:

$$A + B$$

2. **Prefix notation**

- ▪ The *prefix notation* is a notation in which the operator(s) is written before the operands, it is also called polish notation in the honor of the polish mathematician Jan Lukasiewicz who developed this notation. The same expression when written in prefix notation looks like:

$$+ A B$$

EXPRESSION

➢ 3. **Postfix notation**

▪ In the *postfix notation* the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as *suffix notation* or *reverse polish notation*. The above expression if written in postfix expression looks like:

A B +

EXPRESSION

➤ The method of converting infix expression A + B * C to postfix form is:

A + B * C   Infix Form

A + (B * C)   Parenthesized expression

A + (B C *)   Convert the multiplication

A (B C *) +   Convert the addition

A B C * +   Postfix form

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.

2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression   B * C is parenthesized first before A + B.

3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.   47

4. Once the expression is converted to postfix form, remove the parenthesis.

Example 1

Give postfix form for A + [(B + C) + (D + E) * F] / G

Solution. Evaluation order is

A + { [(BC +) + (DE +) * F] / G}

A + { [(BC +) + (DE + F *] / G}

A + { [(BC + (DE + F * +] / G} .

A + [BC + DE + F *+ G /]

ABC + DE + F * + G / +   Postfix Form

Example 2

Give postfix form for (A + B) * C / D + E ^ A / B

Solution. Evaluation order is

[(AB +) * C / D] + [(EA ^) / B]

[(AB +) * C / D] + [(EA ^) B /]

[(AB +) C * D /] + [(EA ^) B /]

(AB +) C * D / (EA ^) B / +

AB + C * D / EA ^ B / +

Postfix Form

Algorithm

1. Push "(" onto stack, and add")" to the end of P.

2. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.

3. If an operand is encountered, add it to Q.

4. If a left parenthesis is encountered, push it onto stack.

5. If an operator $\otimes$ is encountered, then:

   (*a*) Repeatedly pop from stack and add P each operator (on the top of stack),

   which has the same precedence as, or higher precedence than $\otimes$.

   (*b*) Add $\otimes$ to stack.

6. If a right parenthesis is encountered, then:

   (*a*) Repeatedly pop from stack and add to P (on the top of stack until a left

   parenthesis is encountered.

   (*b*) Remove the left parenthesis. [Do not add the left parenthesis to P.]

7. Exit.

Example 3

➢ **Infix String : a+b*c-d**

▪ Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. Being an operator, it is pushed to the stack.
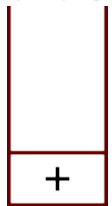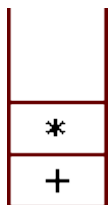
```
┌───┐
│   │
│   │
│   │
│ + │
└───┘
 Stack
```

```
┌─────────┐
│ a       │
└─────────┘
```
Postfix String

▪ Next character scanned is 'b' which will be placed in the Postfix string. Next character is '*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '*', so '*' will be pushed to the stack.

```
┌───┐
│   │
│ * │
│ + │
└───┘
 Stack
```

```
┌─────────┐
│ ab      │
└─────────┘
```
Postfix String

Example 3

- The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '*' which has a higher precedence than '-'. Thus '*' will be popped out from the stack and added to the Postfix string. Even now the stack is not empty. Now the topmost element of the stack is '+' which has equal priority to '-'. So pop the '+' from the stack and add it to the Postfix string. The '-' will be pushed to the stack.

| - |
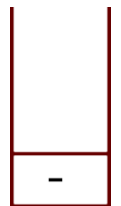
Stack

| abc*+ |

Postfix String

- Next character is 'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be :

Stack

| abc*+d- |

Postfix String

Applications of Stacks

➤ Consider the following arithmetic infix expression P

P = A + (B / C - (D * E ^ F) + G) * H

Character scanned	Stack	Postfix Expression (Q)
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 * +