**UNIT-3**

**Stack**

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

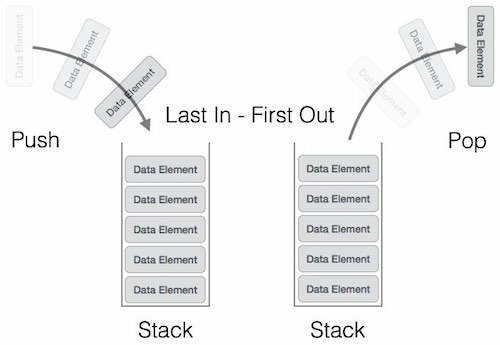


A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

**Stack Representation**

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

**Basic Operations**

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

* **push()** − Pushing (storing) an element on the stack.
* **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

* **peek()** − get the top data element of the stack, without removing it.
* **isFull()** − check if stack is full.
* **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

**peek()**

Algorithm of peek() function −

begin procedure peek

return stack[top]

end procedure

Implementation of peek() function in C programming language −

**Example**

int peek() {

return stack[top];

}

**isfull()**

Algorithm of isfull() function −

begin procedure isfull

if top equals to MAXSIZE

return true

else

return false

endif

end procedure

Implementation of isfull() function in C programming language −

**Example**

bool isfull() {

if(top == MAXSIZE)

return true;

else

return false;

}

**isempty()**

Algorithm of isempty() function −

begin procedure isempty

if top less than 1

return true

else

return false

endif

end procedure

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

**Example**

bool isempty() {

if(top == -1)

return true;

else

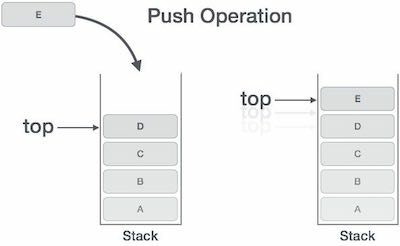
return false;

}

**Push Operation**

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

* **Step 1** − Checks if the stack is full.
* **Step 2** − If the stack is full, produces an error and exit.
* **Step 3** − If the stack is not full, increments **top** to point next empty space.
* **Step 4** − Adds data element to the stack location, where top is pointing.
* **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

**Algorithm for PUSH Operation**

A simple algorithm for Push operation can be derived as follows −

begin procedure push: stack, data

if stack is full

return null

endif

top ← top + 1

stack[top] ← data

end procedure

Implementation of this algorithm in C, is very easy. See the following code −

**Example**

void push(int data) {

if(!isFull()) {

top = top + 1;

stack[top] = data;

} else {

printf("Could not insert data, Stack is full.\n");

}

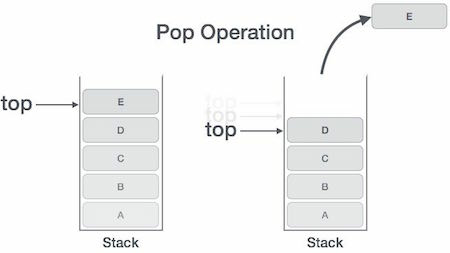
}

**Pop Operation**

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

* **Step 1** − Checks if the stack is empty.
* **Step 2** − If the stack is empty, produces an error and exit.
* **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
* **Step 4** − Decreases the value of top by 1.
* **Step 5** − Returns success.



**Algorithm for Pop Operation**

A simple algorithm for Pop operation can be derived as follows −

begin procedure pop: stack

if stack is empty

return null

endif

data ← stack[top]

top ← top - 1

return data

end procedure

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

* Infix Notation
* Prefix (Polish) Notation
* Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

**Infix Notation**

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

**Prefix Notation**

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

**Postfix Notation**

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations −

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.No.** | **Infix Notation** | **Prefix Notation** | **Postfix Notation** |
| 1 | a + b | + a b | a b + |
| 2 | (a + b) ∗ c | ∗ + a b c | a b + c ∗ |
| 3 | a ∗ (b + c) | ∗ a + b c | a b c + ∗ |
| 4 | a / b + c / d | + / a b / c d | a b / c d / + |
| 5 | (a + b) ∗ (c + d) | ∗ + a b + c d | a b + c d + ∗ |
| 6 | ((a + b) ∗ c) - d | - ∗ + a b c d | a b + c ∗ d - |

**Parsing Expressions**

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

**Precedence**

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

Operator Precendence

As multiplication operation has precedence over addition, b \* c will be evaluated first. A table of operator precedence is provided later.

**Associativity**

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr.No.** | **Operator** | **Precedence** | **Associativity** |
| 1 | Exponentiation ^ | Highest | Right Associative |
| 2 | Multiplication ( ∗ ) & Division ( / ) | Second Highest | Left Associative |
| 3 | Addition ( + ) & Subtraction ( − ) | Lowest | Left Associative |

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a + b\*c**, the expression part **b**\***c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)\*c**.

**Postfix Evaluation Algorithm**

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 − scan the expression from left to right

Step 2 − if it is an operand push it to stack

Step 3 − if it is an operator pull operand from stack and perform operation

Step 4 − store the output of step 3, back to stack

Step 5 − scan the expression until all operands are consumed

Step 6 − pop the stack and perform operation

**Queue**

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

**Queue Representation**

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

**Basic Operations**

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

* **enqueue()** − add (store) an item to the queue.
* **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

* **peek()** − Gets the element at the front of the queue without removing it.
* **isfull()** − Checks if the queue is full.
* **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

**peek()**

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

**Algorithm**

begin procedure peek

return queue[front]

end procedure

Implementation of peek() function in C programming language −

**Example**

int peek() {

return queue[front];

}

**isfull()**

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

**Algorithm**

begin procedure isfull

if rear equals to MAXSIZE

return true

else

return false

endif

end procedure

Implementation of isfull() function in C programming language −

**Example**

bool isfull() {

if(rear == MAXSIZE - 1)

return true;

else

return false;

}

**isempty()**

Algorithm of isempty() function −

**Algorithm**

begin procedure isempty

if front is less than MIN OR front is greater than rear

return true

else

return false

endif

end procedure

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

**Example**

bool isempty() {

if(front < 0 || front > rear)

return true;

else

return false;

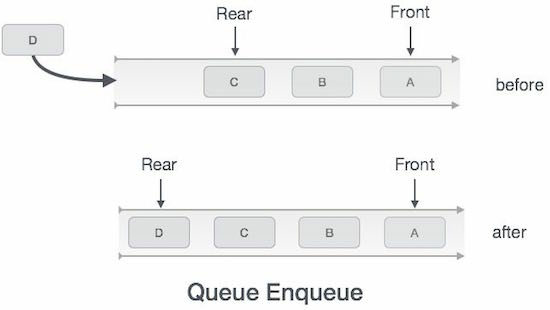
}

**Enqueue Operation**

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

* **Step 1** − Check if the queue is full.
* **Step 2** − If the queue is full, produce overflow error and exit.
* **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
* **Step 4** − Add data element to the queue location, where the rear is pointing.
* **Step 5** − return success.



Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

**Algorithm for enqueue operation**

procedure enqueue(data)

if queue is full

return overflow

endif

rear ← rear + 1

queue[rear] ← data

return true

end procedure

Implementation of enqueue() in C programming language −

**Example**

int enqueue(int data)

if(isfull())

return 0;

rear = rear + 1;

queue[rear] = data;

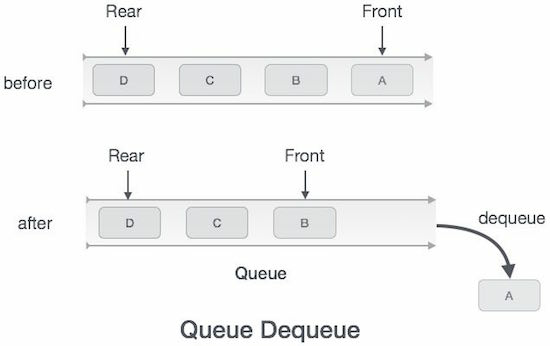
return 1;

end procedure

**Dequeue Operation**

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

* **Step 1** − Check if the queue is empty.
* **Step 2** − If the queue is empty, produce underflow error and exit.
* **Step 3** − If the queue is not empty, access the data where **front** is pointing.
* **Step 4** − Increment **front** pointer to point to the next available data element.
* **Step 5** − Return success.



**Algorithm for dequeue operation**

procedure dequeue

if queue is empty

return underflow

end if

data = queue[front]

front ← front + 1

return true

end procedure

Implementation of dequeue() in C programming language −

**Example**

int dequeue() {

if(isempty())

return 0;

int data = queue[front];

front = front + 1;

return data;

}

# Circular Queue

Circular queue avoids the wastage of space in a [regular queue implementation using arrays](https://www.programiz.com/data-structures/queue).

As you can see in the above image, after a bit of enqueueing and dequeueing, the size of the queue has been reduced.

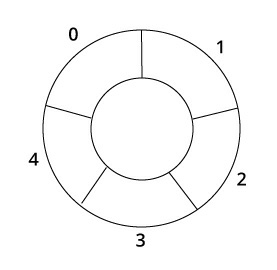
The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

## How Circular Queue Works

Circular Queue works by the process of circular increment i.e. when we try to increment any variable and we reach the end of queue, we start from the beginning of queue by modulo division with the queue size.

i.e.

if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)



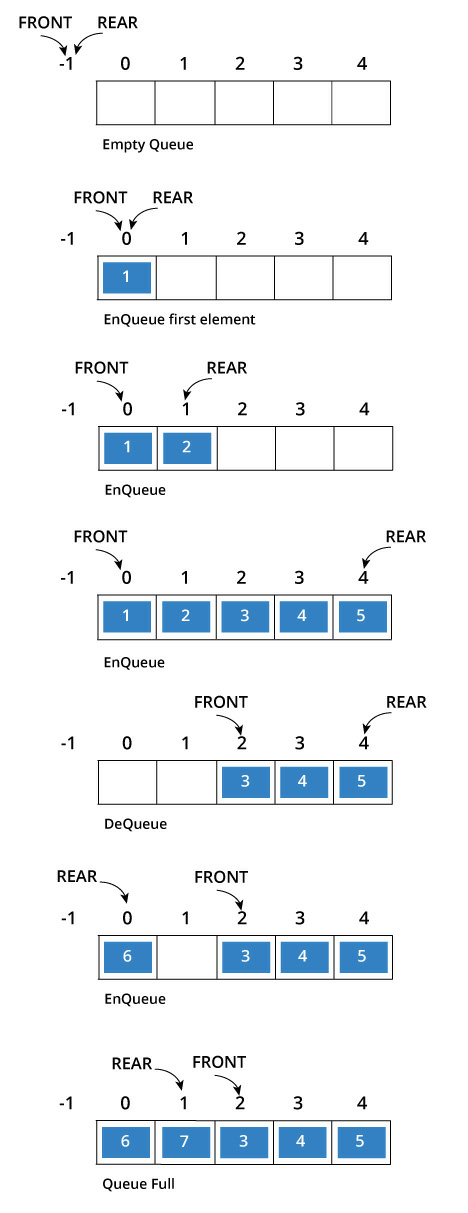
Queue operations work as follows:

* Two pointers called *FRONT* and *REAR* are used to keep track of the first and last elements in the queue.
* When initializing the queue, we set the value of *FRONT* and *REAR* to -1.
* On enqueing an element, we circularly increase the value of *REAR* index and place the new element in the position pointed to by *REAR*.
* On dequeueing an element, we return the value pointed to by *FRONT* and circularly increase the *FRONT* index.
* Before enqueing, we check if queue is already full.
* Before dequeuing, we check if queue is already empty.
* When enqueing the first element, we set the value of *FRONT* to 0.
* When dequeing the last element, we reset the values of *FRONT* and *REAR* to -1.

However, the check for full queue has a new additional case:

* Case 1: *FRONT* = 0 && REAR == SIZE - 1
* Case 2: FRONT = REAR + 1

The second case happens when *REAR* starts from 0 due to circular increment and when its value is just 1 less than *FRONT*, the queue is full.



## Circular Queue Implementation in programming language

The most common queue implementation is using arrays, but it can also be implemented using lists.

### Implementation using C programming

1. #include <stdio.h>
2. #define SIZE 5
3. int items[SIZE];
4. int front = -1, rear =-1;
5. int isFull()
6. {
7. if( (front == rear + 1) || (front == 0 && rear == SIZE-1)) return 1;
8. return 0;
9. }
10. int isEmpty()
11. {
12. if(front == -1) return 1;
13. return 0;
14. }
15. void enQueue(int element)
16. {
17. if(isFull()) printf("\n Queue is full!! \n");
18. else
19. {
20. if(front == -1) front = 0;
21. rear = (rear + 1) % SIZE;
22. items[rear] = element;
23. printf("\n Inserted -> %d", element);
24. }
25. }
26. int deQueue()
27. {
28. int element;
29. if(isEmpty()) {
30. printf("\n Queue is empty !! \n");
31. return(-1);
32. } else {
33. element = items[front];
34. if (front == rear){
35. front = -1;
36. rear = -1;
37. } /\* Q has only one element, so we reset the queue after dequeing it. ? \*/
38. else {
39. front = (front + 1) % SIZE;
41. }
42. printf("\n Deleted element -> %d \n", element);
43. return(element);
44. }
45. }
46. void display()
47. {
48. int i;
49. if(isEmpty()) printf(" \n Empty Queue\n");
50. else
51. {
52. printf("\n Front -> %d ",front);
53. printf("\n Items -> ");
54. for( i = front; i!=rear; i=(i+1)%SIZE) {
55. printf("%d ",items[i]);
56. }
57. printf("%d ",items[i]);
58. printf("\n Rear -> %d \n",rear);
59. }
60. }
61. int main()
62. {
63. // Fails because front = -1
64. deQueue();
66. enQueue(1);
67. enQueue(2);
68. enQueue(3);
69. enQueue(4);
70. enQueue(5);
72. // Fails to enqueue because front == 0 && rear == SIZE - 1
73. enQueue(6);
75. display();
76. deQueue();
78. display();
80. enQueue(7);
81. display();
83. // Fails to enqueue because front == rear + 1
84. enQueue(8);
86. return 0;
87. }

When you run this program, the output will be

Inserted -> 1

Inserted -> 2

Inserted -> 3

Inserted -> 4

Inserted -> 5

Queue is full!!

Front -> 0

Items -> 1 2 3 4 5

Rear -> 4

Deleted element -> 1

Front -> 1

Items -> 2 3 4 5

Rear -> 4

Inserted -> 7

Front -> 1

Items -> 2 3 4 5 7

Rear -> 0

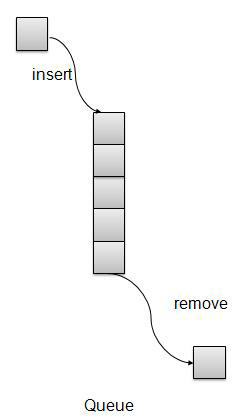
Queue is full!!

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

**Basic Operations**

* **insert / enqueue** − add an item to the rear of the queue.
* **remove / dequeue** − remove an item from the front of the queue.

**Priority Queue Representation**

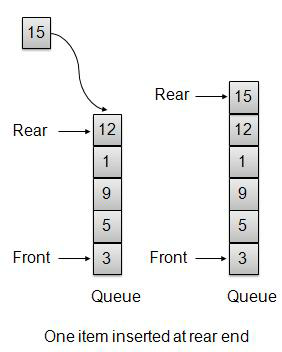


We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

* **Peek** − get the element at front of the queue.
* **isFull** − check if queue is full.
* **isEmpty** − check if queue is empty.

**Insert / Enqueue Operation**

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.



void insert(int data){

int i = 0;

if(!isFull()){

// if queue is empty, insert the data

if(itemCount == 0){

intArray[itemCount++] = data;

}else{

// start from the right end of the queue

for(i = itemCount - 1; i >= 0; i-- ){

// if data is larger, shift existing item to right end

if(data > intArray[i]){

intArray[i+1] = intArray[i];

}else{

break;

}

}

// insert the data

intArray[i+1] = data;

itemCount++;

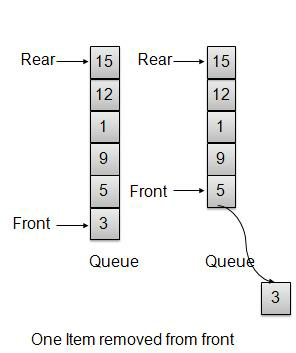
}

}

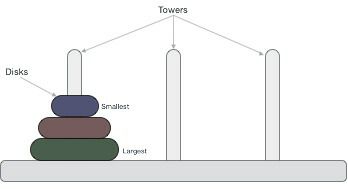
}

**Remove / Dequeue Operation**

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.



Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −



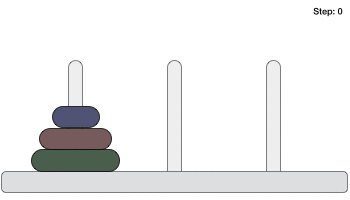
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

**Rules**

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

* Only one disk can be moved among the towers at any given time.
* Only the "top" disk can be removed.
* No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



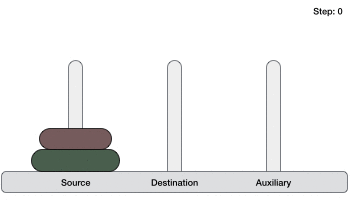
Tower of Hanoi puzzle with n disks can be solved in minimum **2n−1** steps. This presentation shows that a puzzle with 3 disks has taken **23 - 1 = 7** steps.

**Algorithm**

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

* First, we move the smaller (top) disk to aux peg.
* Then, we move the larger (bottom) disk to destination peg.
* And finally, we move the smaller disk from aux to destination peg.



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are −

**Step 1** − Move n-1 disks from **source** to **aux**

**Step 2** − Move nth disk from **source** to **dest**

**Step 3** − Move n-1 disks from **aux** to **dest**

A recursive algorithm for Tower of Hanoi can be driven as follows −

START

Procedure Hanoi(disk, source, dest, aux)

IF disk == 1, THEN

move disk from source to dest

ELSE

Hanoi(disk - 1, source, aux, dest) // Step 1

move disk from source to dest // Step 2

Hanoi(disk - 1, aux, dest, source) // Step 3

END IF

END Procedure

STOP