UNIT-3

Stacks and Queue

STACKS

A stack is a restricted linear list in which all additions and deletions are made at one end, the top. If we insert a series of data items into a stack and then remove them, the order of the data is reversed. This reversing attribute is why stacks are known as last in, first out (LIFO) data structures.

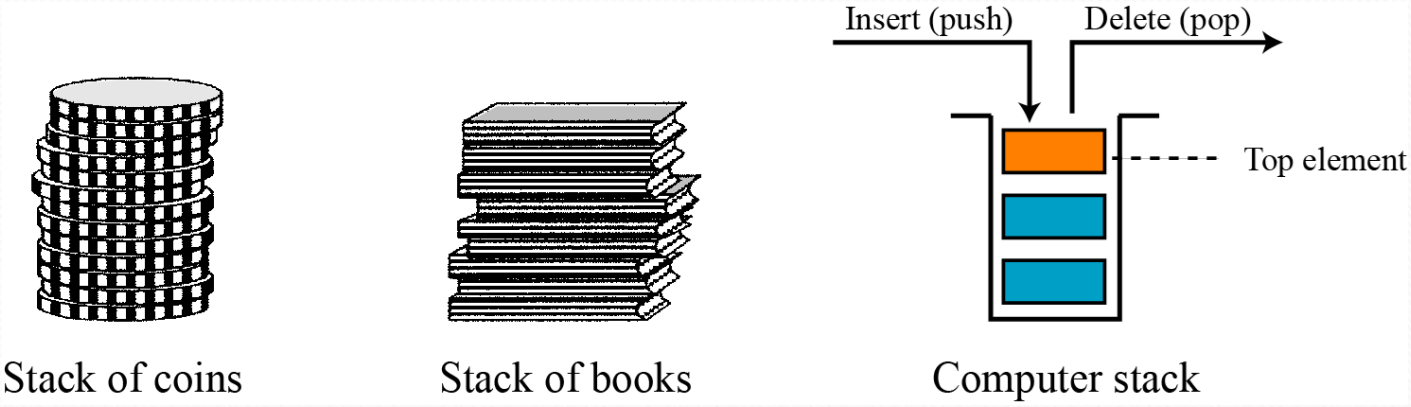


Figure Three representations of stacks

Stack ADT

We define a stack as an ADT as shown below:

Stack ADT

Definition	A list of data items that can only be accessed at one end, called the *top* of the stack.
Operations	**stack:** Creates an empty stack.
	push: Inserts an element at the top.
	pop: Deletes the top element.
	empty: Checks the status of the stack.

Peek - This returns the top data value of the stack.

Operations on stacks

There are four basic operations, *stack*, *push*, *pop* and *empty*, that we define in this chapter.

The *stack* operation

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

stack (stackName)

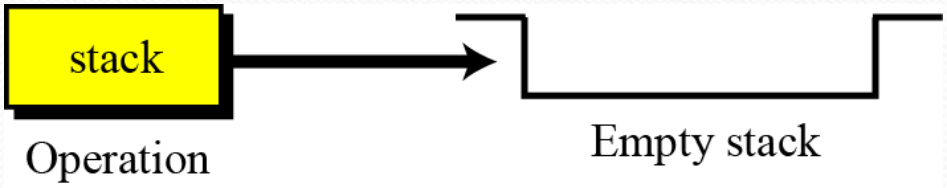


Figure **Stack operation**

The *push* operation

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

push (stackName, dataItem)

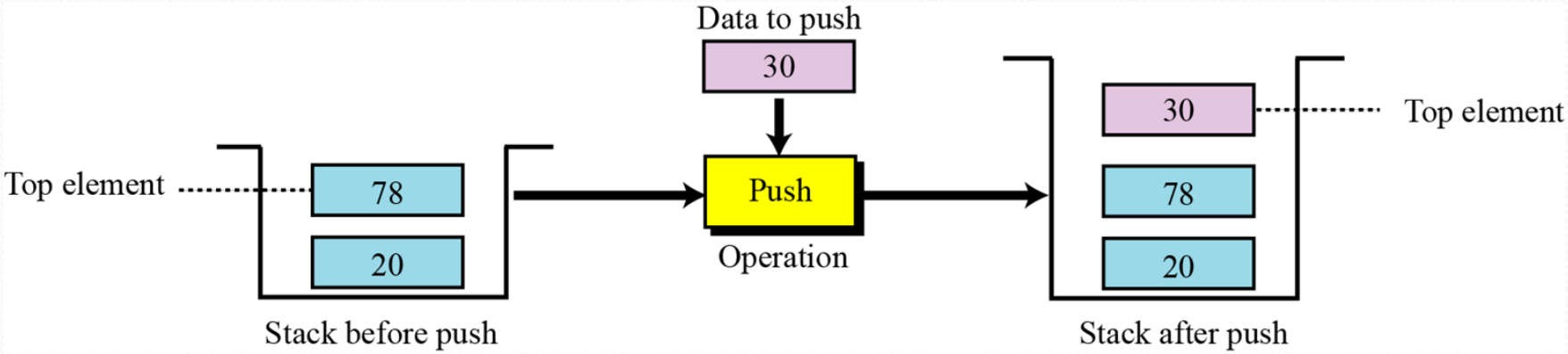


Figure Push operation

The *pop* operation

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

pop (stackName, dataItem)

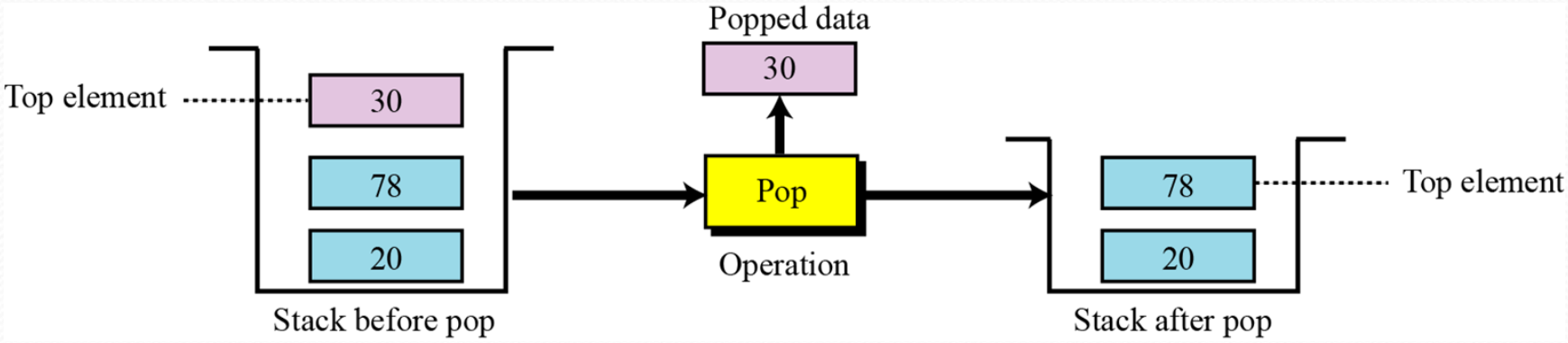


Figure Pop operation

The *empty* operation

The *empty* operation checks the status of the stack. The following shows the format.

```
empty (stackName)
```

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

Stack implementation

At the ADT level, we use the stack and its four operations; at the implementation level, we need to choose a data structure to implement it. Stack ADTs can be implemented using either an array or a linked list. Figure 12.7 shows an example of a stack ADT with five items. The figure also shows how we can implement the stack.

In our array implementation, we have a record that has two fields. The first field can be used to store information about the array. The linked list implementation is similar: we have an extra node that has the name of the stack. This node also has two fields: a counter and a pointer that points to the top element.

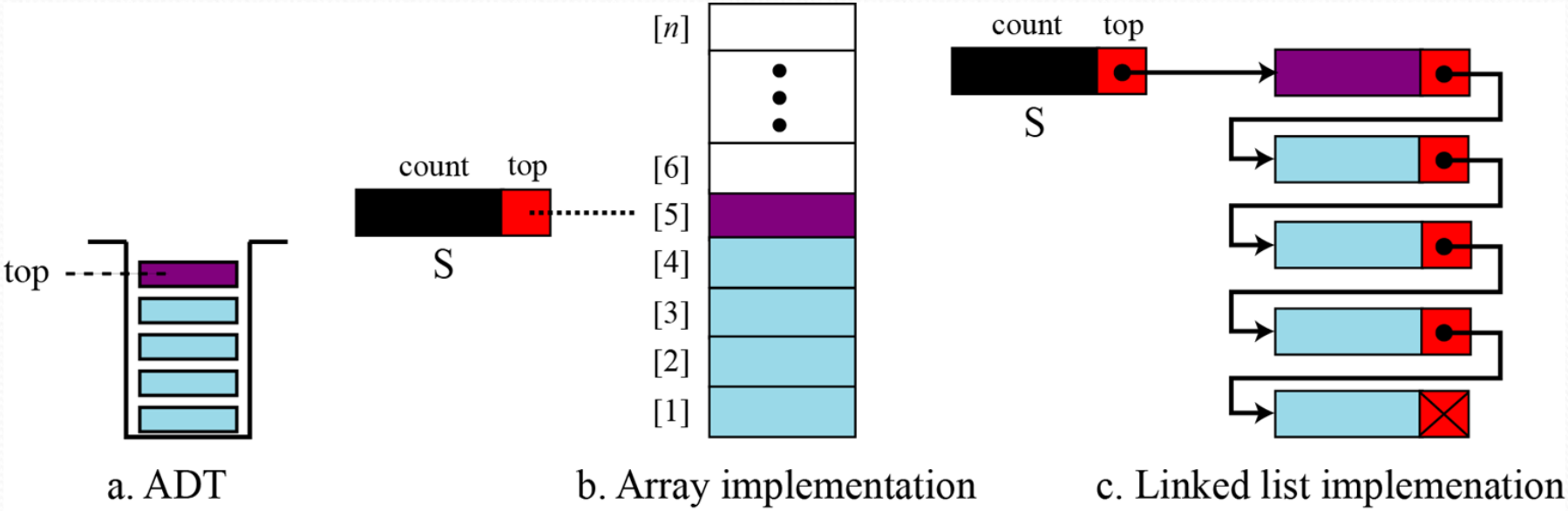


Figure **Stack implementations**

Code

```
struct Node {
int data;
struct Node *next;
};
```

```
1. void push(int val) {
2.    struct Node* newnode =
      (struct Node*)
      malloc(sizeof(struct Node));
3.    newnode->data = val;
4.    newnode->next = top;
5.    top = newnode;
6. }
```

The push() function takes argument val i.e. value to be pushed into the stack. Then a new node is created and val is inserted into the data part. This node is added to the front of the linked list and top points to it

Code

- The pop() function pops the topmost value of the stack, if there is any value. If the stack is empty then underflow is printed

```
1. void pop() {
2.    if(top==NULL)
3.    cout<<"Stack Underflow"<<endl;
4.    else {
5.       cout<<"The popped element is "<< top->data <<endl;
6.       top = top->next;
7.    }
8. }
```

Code

- The display() function displays all the elements in the stack. This is done by using ptr that initially points to top but goes till the end of the stack. All the data values corresponding ti ptr are printed.

```
1. void display() {
2.    struct Node* ptr;
3.    if(top==NULL)
4.    cout<<"stack is empty";
5.    else {
6.       ptr = top;
7.       cout<<"Stack elements are: ";
8.       while (ptr != NULL) {
9.          cout<< ptr->data <<" ";
10.         ptr = ptr->next;
11.      }
12.   }
      cout<<endl;
   }
```

Code

```
int main() {
   int ch, val;
   cout<<"1) Push in stack"<<endl;
   cout<<"2) Pop from stack"<<endl;
   cout<<"3) Display stack"<<endl;
   cout<<"4) Exit"<<endl;
   do {
      cout<<"Enter choice: "<<endl;
      cin>>ch;
      switch(ch) {
         case 1: {
            cout<<"Enter value to be
pushed:"<<endl;
            cin>>val;
            push(val);
            break;
         }
         case 2: {
            pop();
            break;
         }
         case 3: {
            display();
            break;
         }
         case 4: {
            cout<<"Exit"<<endl;
            break;
         }
         default: {
            cout<<"Invalid Choice"<<endl;
         }
      }
   }while(ch!=4);
   return 0;
}
```

Stack applications

Stack applications can be classified into four broad categories: ***reversing data***, ***pairing data***, ***postponing data usage*** and ***backtracking steps***. We discuss the first two in the sections that follow.

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

We often need to pair some characters in an expression. For example, when we write a mathematical expression in a computer language, we often need to use parentheses to change the precedence of operators. The following two expressions are evaluated differently because of the parentheses in the second expression:

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

When we type an expression with a lot of parentheses, we often forget to pair the parentheses. One of the duties of a compiler is to do the checking for us. The compiler uses a stack to check that all opening parentheses are paired with a closing parentheses.

Example

Algorithm shows how we can 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
}
```

Arithmetic expression conversion and evaluation

- Expression Parsing
- The way to write arithmetic expression is known as a **notation**.
- These notations are –
  1. Infix Notation
  2. Prefix (Polish) Notation
  3. Postfix (Reverse-Polish) Notation

1. **Infix Notation**

a - b + c,

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.

2. **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**.

3. **Postfix Notation**

This notation style is known as **Reversed Polish Notation**. 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

- 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 –

Associativity

It 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

Postfix Evaluation Algorithm

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

Eg. AB+C-BA+C^- A=1,B=2, C=3

Infix to Postfix Conversion

- To convert any Infix expression into Postfix or Prefix expression we can use the following procedure...

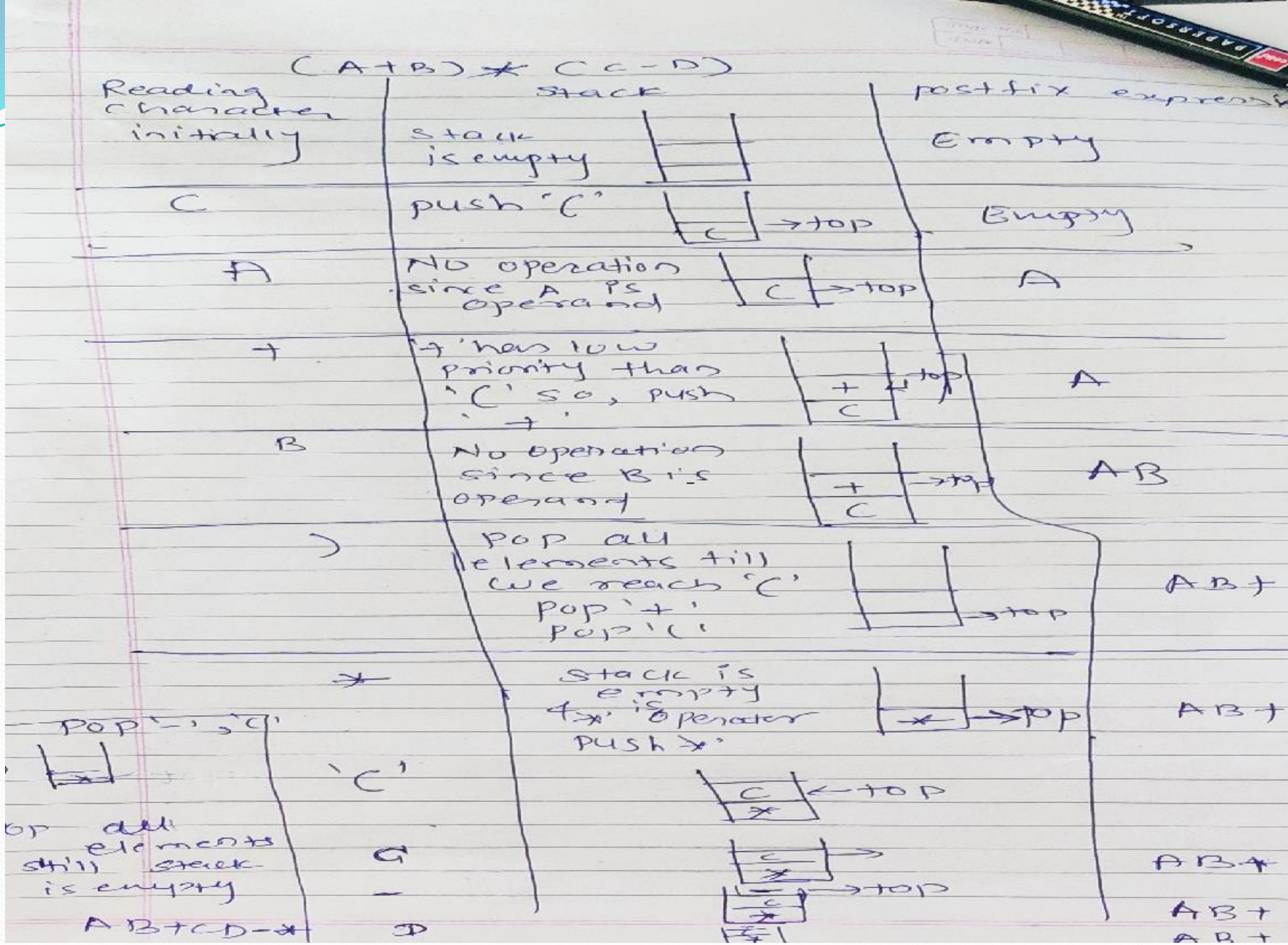
1. Find all the operators in the given Infix Expression.
2. Find the order of operators evaluated according to their Operator precedence.
3. Convert each operator into required type of expression (Postfix or Prefix) in the same order.

Eg. **D = A + B * C**

- **Step 1** - The Operators in the given Infix Expression : = , + , *
- **Step 2** - The Order of Operators according to their preference : * , + , =
- **Step 3** - Now, convert the first operator * ----- **D = A + B C** *
- **Step 4** - Convert the next operator + ----- **D = A BC*** +
- **Step 5** - Convert the next operator = ----- **D ABC*+ =**

Infix to Postfix Conversion using Stack Data Structure

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
5. If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.


(A+B) * (C-D)
Reading character initially
Stack
postfix expression
Stack is empty
Empty
C
push 'C'
top
Empty
A
No operation since A is operand
top
A
+
'+' has low priority than '(' so, push '+'
top
A
B
No operation since B is operand
top
AB
)
Pop all elements till we reach '('
Pop '+'
Pop '('
top
AB+
*
Stack is empty
'*' is operator
push '*'
top
AB+
Pop '-', '('
'C'
AB+CD-*
D
AB+
AB+

Convert Infix To Prefix Notation

Step 1: Reverse the infix expression i.e A+B*C will become C*B+A. Note while reversing each '(' will become ')' and each ')' becomes '('.

Step 2: Obtain the postfix expression of the modified expression i.e CB*A+.

Step 3: Reverse the postfix expression

Eg.

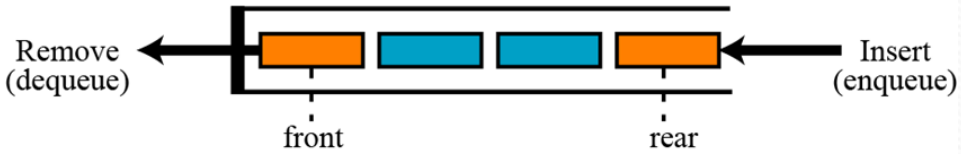
Input : A * B + C / D Output : + * A B/ C D

Input : (A - B/C) * (A/K-L) Output : *-A/BC-/AKL

QUEUES

A **queue** is a linear list or abstract data type in which data can only be inserted at one end, **called** the ***rear***, and deleted from the other end, called the ***front***. These restrictions ensure that the data is processed through the queue in the order in which it is received. In other words, a queue is a **first in, first out (FIFO)** structure.

A queue of people



A computer queue

Figure **Two representation of queues**

Queue ADT

We define a queue as an ADT as shown below:

Queue ADT

Definition	A list of data items in which an item can be deleted from one end, called the *front* of the queue and an item can be inserted at the other end, called the *rear* of the queue.
Operations	**queue:** Creates an empty queue.
	enqueue: Inserts an element at the rear.
	dequeue: Deletes an element from the front.
	empty: Checks the status of the queue.

Operations on queues

Although we can define many operations for a queue, four are basic: queue, enqueue, dequeue and empty, as defined below.

The *queue* operation

The queue operation creates an empty queue. The following shows the format.

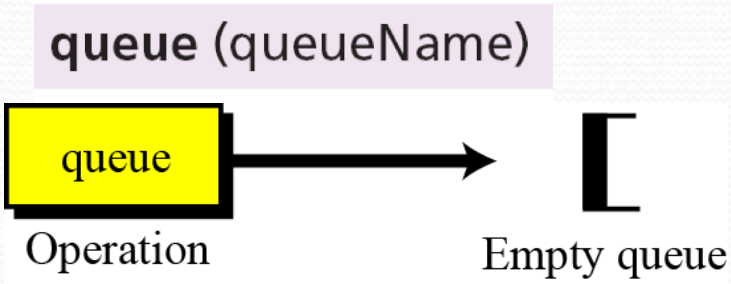


Figure The queue operation

The *enqueue* operation

The enqueue operation inserts an item at the rear of the queue. The following shows the format.

enqueue (queueName, dataItem)

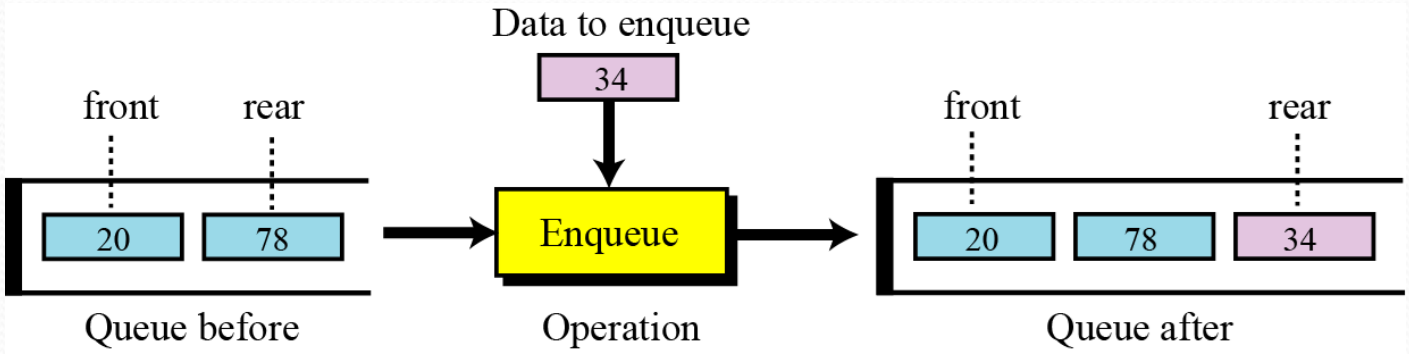


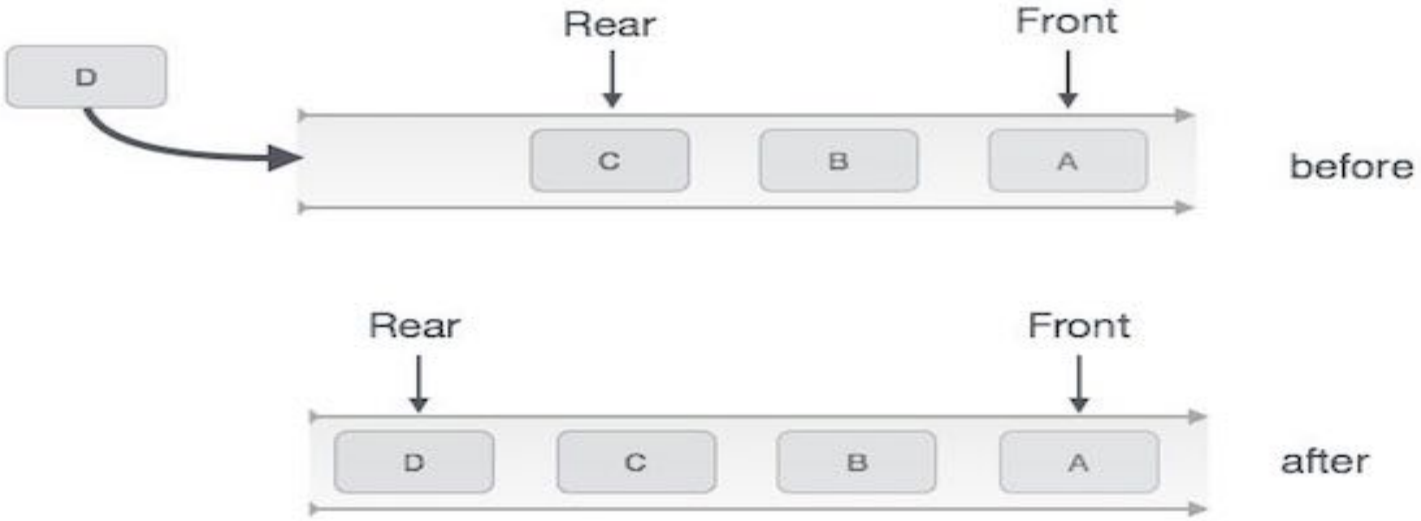
Figure The enqueue operation

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.



The *dequeue* operation

The dequeue operation deletes the item at the front of the queue. The following shows the format.

dequeue (queueName, dataItem)

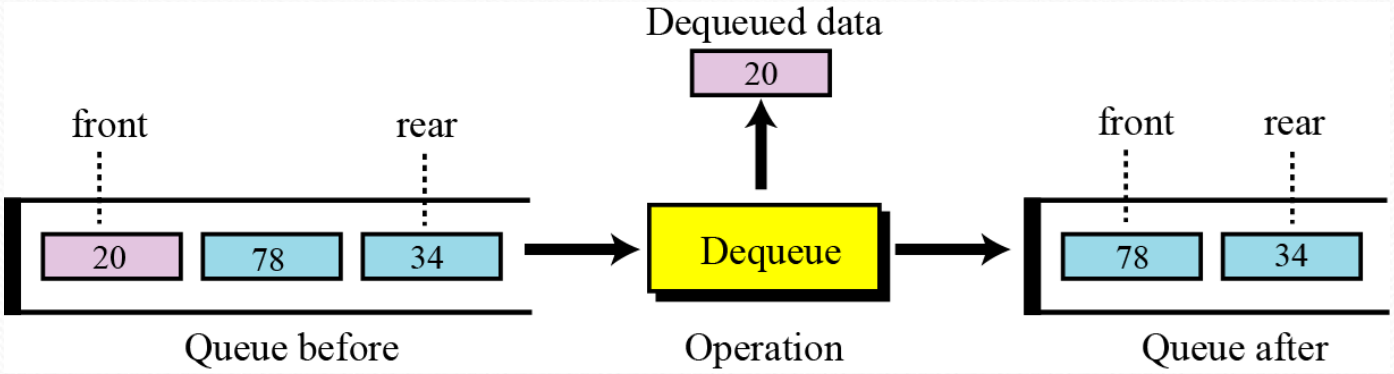
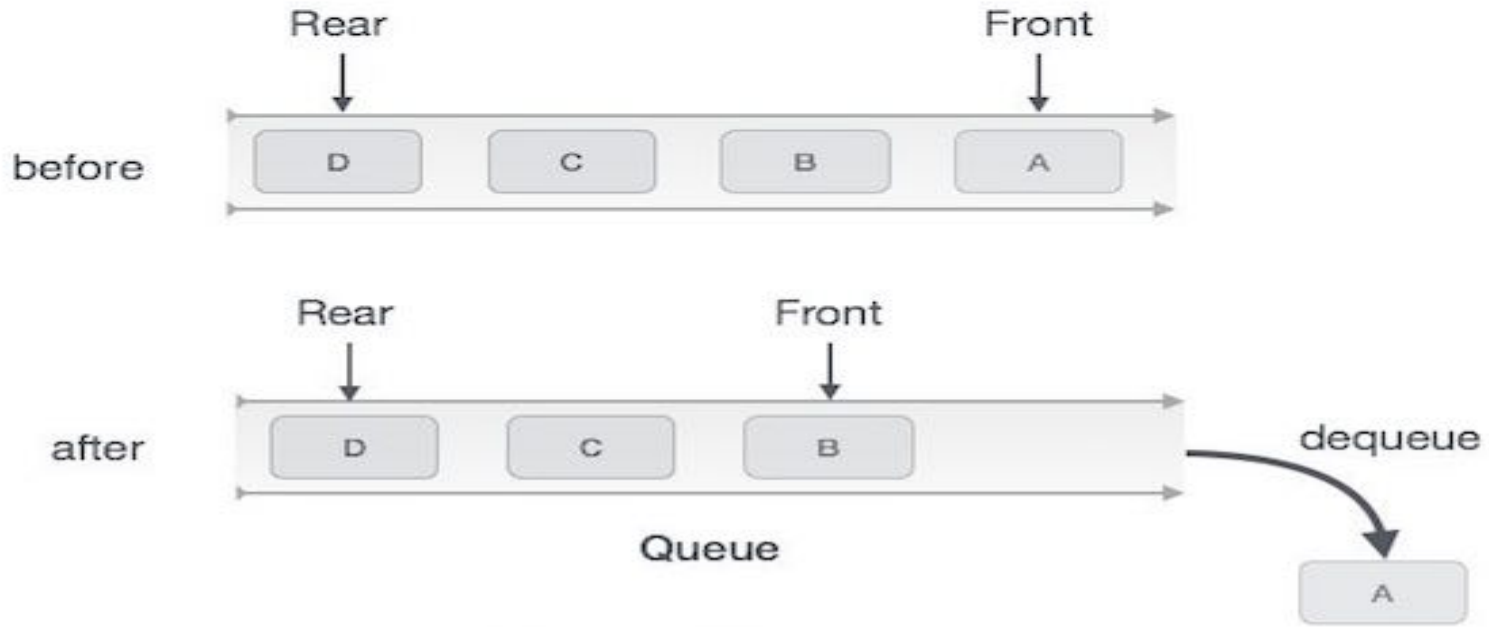


Figure The dequeue operation

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.



Queue Dequeue

The *empty* operation

The empty operation checks the status of the queue. The following shows the format.

empty (queueName)

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

Queue applications

Queues are one of the most common of all data processing structures. They are found in virtually every operating system and network and in countless other areas. For example, queues are used in online business applications such as processing customer requests, jobs and orders. In a computer system, a queue is needed to process jobs and for system services such as print spools.

Job Scheduling

Queue Simulation

- In general, queues are often used as "waiting lines". Here are a few examples of where queues would be used:
- In operating systems, for controlling access to shared system resources such as printers, files, communication lines, disks and tapes. A specific example of print queues follows:

-In the situation where there are multiple users or a networked computer system, you probably share a printer with other users. When you request to print a file, your request is added to the print queue. When your request reaches the front of the print queue, your file is printed. This ensures that only one person at a time has access to the printer and that this access is given on a first-come, first-served basis.

-For simulation of real-world situations. For instance, a new bank may want to know how many tellers to install. The goal is to service each customer within a "reasonable" wait time, but not have too many tellers for the number of customers. To find out a good number of tellers, they can run a computer simulation of typical customer transactions using queues to represent the waiting customers.

-When placed on hold for telephone operators. For example, when you phone the toll-free number for your bank, you may get a recording that says, "Thank you for calling A-1 Bank. Your call will be answered by the next available operator. Please wait." This is a queuing system.

Example

Another common application of a queue is to adjust and create a balance between a fast producer of data and a slow consumer of data. For example, assume that a CPU is connected to a printer. The speed of a printer is not comparable with the speed of a CPU. If the CPU waits for the printer to print some data created by the CPU, the CPU would be idle for a long time. The solution is a queue. The CPU creates as many chunks of data as the queue can hold and sends them to the queue. The CPU is now free to do other jobs. The chunks are dequeued slowly and printed by the printer. The queue used for this purpose is normally referred to as a spool queue.

Queue implementation

At the ADT level, we use the queue and its four operations at the implementation level. We need to choose a data structure to implement it. A queue ADT can be implemented using either an array or a linked list. Figure shows an example of a queue ADT with five items. The figure also shows how we can implement it. In the array implementation we have a record with three fields. The first field can be used to store information about the queue.

The linked list implementation is similar: we have an extra node that has the name of the queue. This node also has three fields: a count, a pointer that points to the front element and a pointer that points to the rear element.

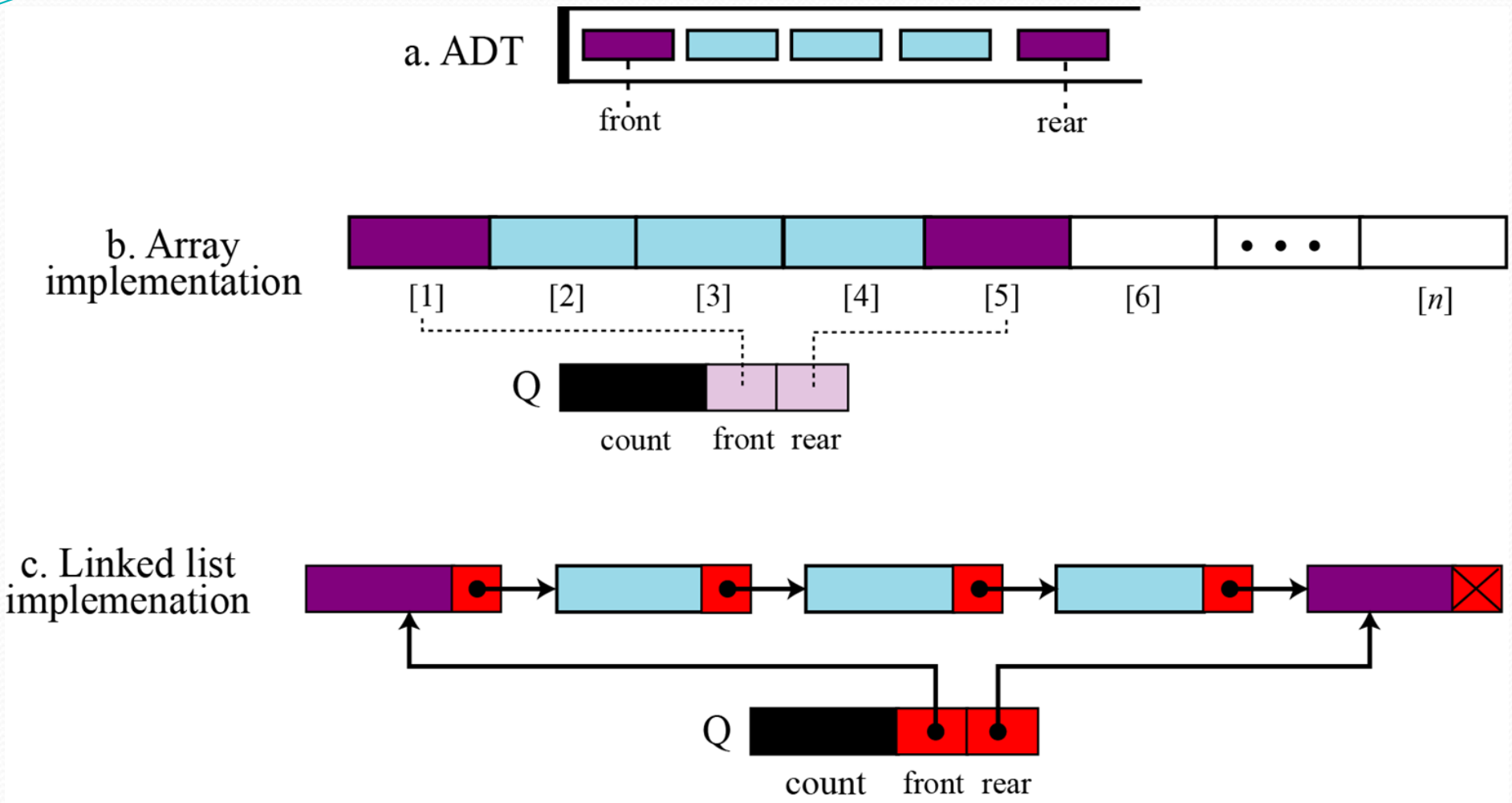


Figure Queue implementation

Code

```
#include <iostream>
using namespace std;
struct node {
int data;    struct node
*next; };
struct node* front =
NULL;
struct node* rear = NULL;
struct node* temp;
```

- insert() inserts an element into the queue. If rear is NULL,then the queue is empty and a single element is inserted. Otherwise, a node is inserted after rear with the required element and then that node is set to rear.

```
void Insert()
{    int val;    cout<<"Insert the element in queue : "<<endl;
 cin>>val;
if (rear == NULL) {
rear = (struct node *)malloc(sizeof(struct node));
rear->next = NULL;
rear->data = val;
 front = rear;    }
else {
   temp=(struct node *)malloc(sizeof(struct node));
rear->next = temp;
 temp->data = val;       temp->next = NULL;
   rear = temp;    } }
```

- In the function Delete(), if there are no elements in queue then it is underflow condition. If there is only one element in the queue that is deleted and front and rear are set to NULL. Otherwise, the element at front is deleted and front points to next element

```
void Delete() {
 temp = front;
if (front == NULL) {
    cout<<"Underflow"<<endl;
  return;    }
 else    if (temp->next != NULL)
{       temp = temp->next;
cout<<"Element deleted from queue is : "<<front->data<<endl;
   free(front);
  front = temp;
  } else
 {
cout<<"Element deleted from queue is : "<<front->data<<endl;
free(front);
  front = NULL;       rear = NULL;    } }
```

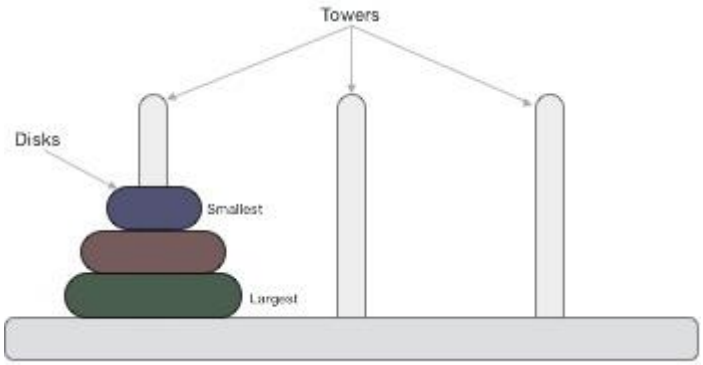
In the function display(), if front and rear are NULL then queue is empty. Otherwise, all the queue elements are displayed using a while loop with the help of temp variable.

```
void Display()
 {    temp = front;
  if ((front == NULL) && (rear == NULL))
 {       cout<<"Queue is empty"<<endl;
  return;
}
cout<<"Queue elements are: ";
while (temp != NULL)
 {       cout<<temp->data<<" ";
   temp = temp->next;
}
 cout<<endl;
 }
```

Tower of Hanoi

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

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



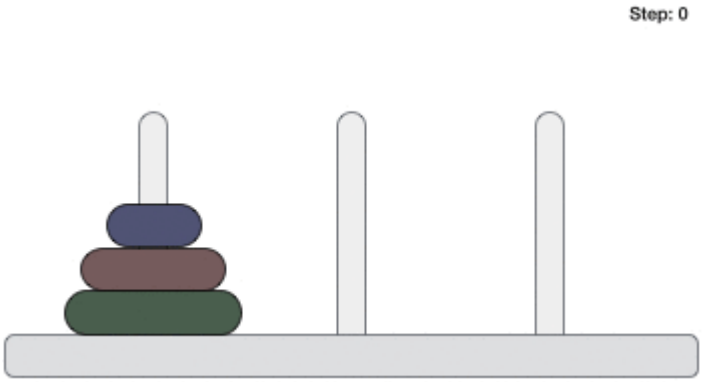
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

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 $\mathbf{2^n-1}$ steps. This presentation shows that a puzzle with 3 disks has taken $\mathbf{2^3 - 1 = 7}$ steps.

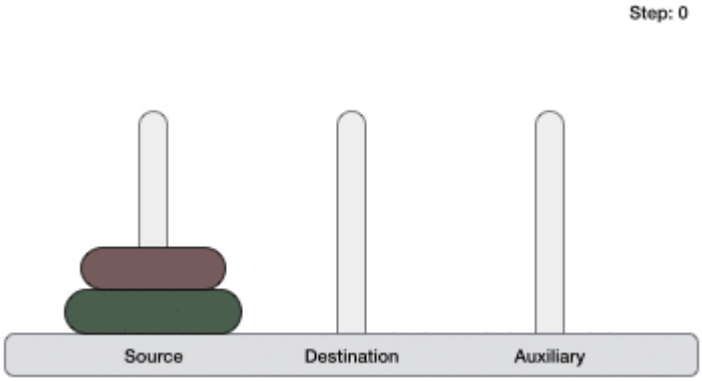
Algorithm

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 (n^{th} 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 n^{th} 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
```

Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function **α** either calls itself directly or calls a function **β** that in turn calls the original function **α**. The function **α** is called recursive function.

Example − a function calling itself.

```
int function(int value) {
   if(value < 1)
      return;
   function(value - 1);

   printf("%d ",value);
}
```

Example − a function that calls another function which in turn calls it again.

```
int function1(int value1) {
   if(value1 < 1)
      return;
   function2(value1 - 1);
   printf("%d ",value1);
}
int function2(int value2) {
   function1(value2);
}
```

Properties

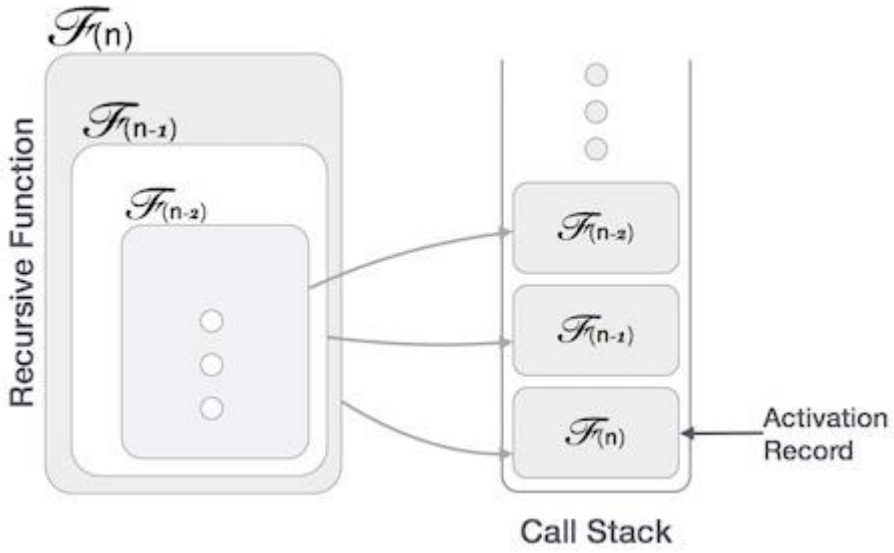
A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

- **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Implementation

Many programming languages implement recursion by means of **stacks**. Generally, whenever a function (**caller**) calls another function (**callee**) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.

This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.



This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.

Analysis of Recursion

One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

Time Complexity

In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is O(1), hence the (n) number of times a recursive call is made makes the recursive function O(n).

Space Complexity

Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.