Stack and Queue

Content

- Stack using arrays
- Stack using linked lists
- Queue using arrays
- Queue using linked lists
- Applications

Stack

- It is a linear data structure that follows a particular order in which the operations are performed.

- LIFO (Last In First Out)

This strategy states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first.

Basic Operations

- push() to insert an element into the stack
- pop() to remove an element from the stack
- top() Returns the top element of the stack.
- isEmpty() returns true is stack is empty else false
- size() returns the size of stack

Array Code

```
#include <stdio.h>
int stack[100],n,top=-1;
```

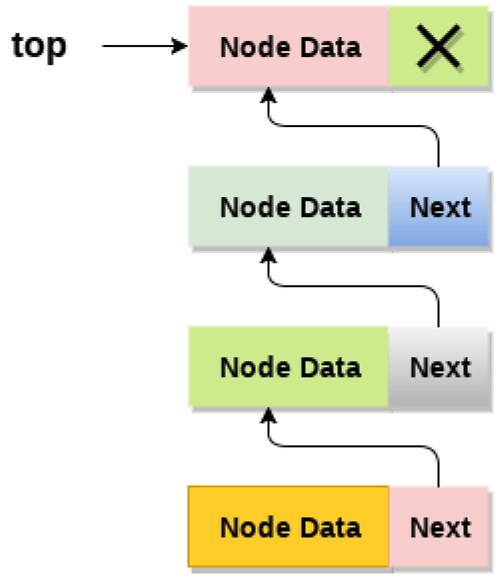
```
void push ()
{
   int val;
   if (top == n )
   printf("\n Overflow");
   else
   {
      printf("Enter the value?");
      scanf("%d",&val);
      top = top +1;
      stack[top] = val;
   }
}
```

```
void pop ()
{
   if(top == -1)
   printf("Underflow");
   else
   top = top -1;
}
```

Contd.

```
void show()
{
    for (i=top;i>=0;i--)
    {
        printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
        printf("Stack is empty");
    }
}
```

Linked List Implementation of Stack

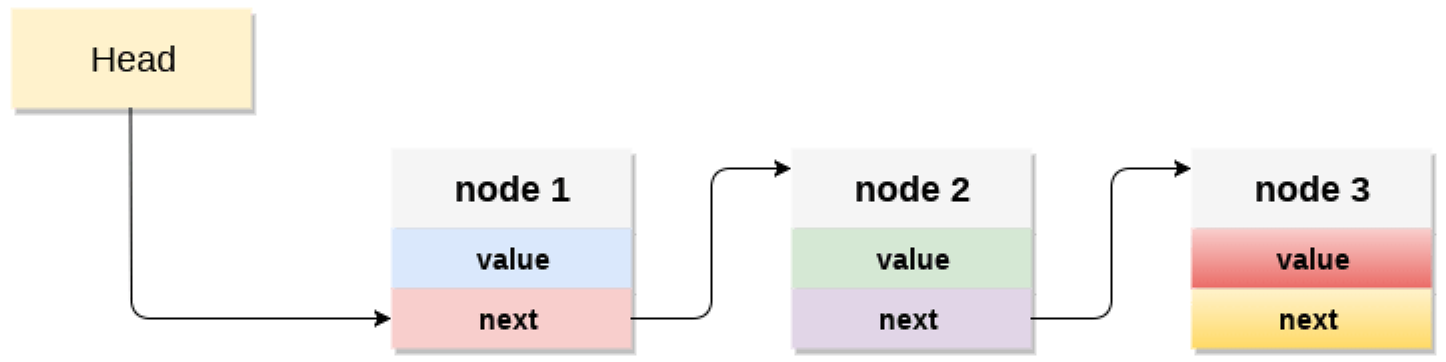


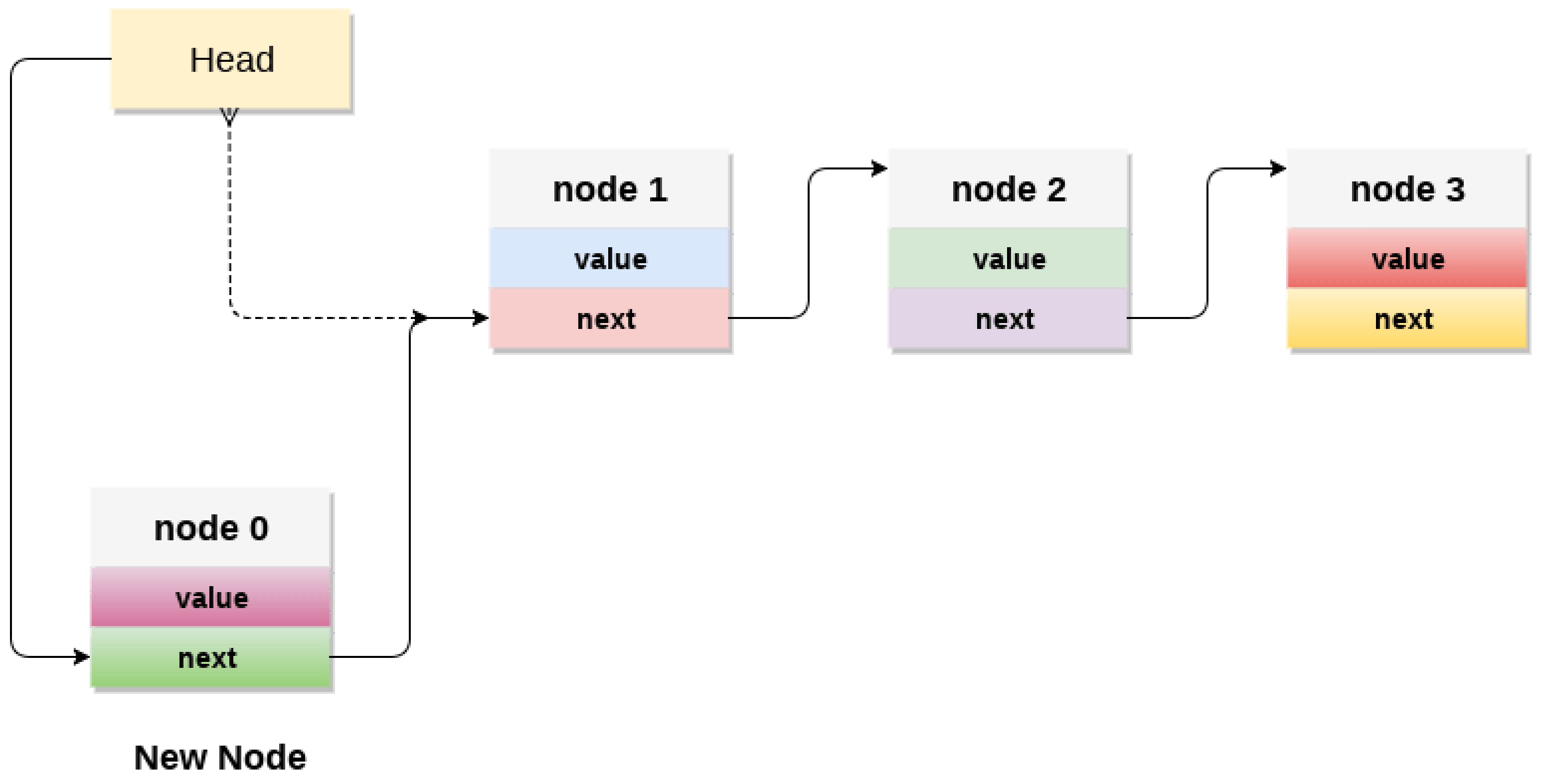
Stack

Push()

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Contd.


Head
node 1
value
next
node 2
value
next
node 3
value
next


Head
node 1
value
next
node 2
value
next
node 3
value
next
node 0
value
next
New Node

Code

```
void push ()
{
   int val;
   struct node *ptr =(struct node*)malloc(sizeof(struct
node));
   if(ptr == NULL)
   {
      printf("not able to push the element");
   }
   else
   {
      printf("Enter the value");
      scanf("%d",&val);
      if(head==NULL)
      {
         ptr->val = val;
         ptr -> next = NULL;
         head=ptr;
      }
      else
      {
         ptr->val = val;
         ptr->next = head;
         head=ptr;

      }
      printf("Item pushed");

   }
}
```

Pop()

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Code

```
void pop()
{
    int item;
    struct node *ptr;
    if (head == NULL)
    {
        printf("Underflow");
    }
else
    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");

    }
}
```

Display()

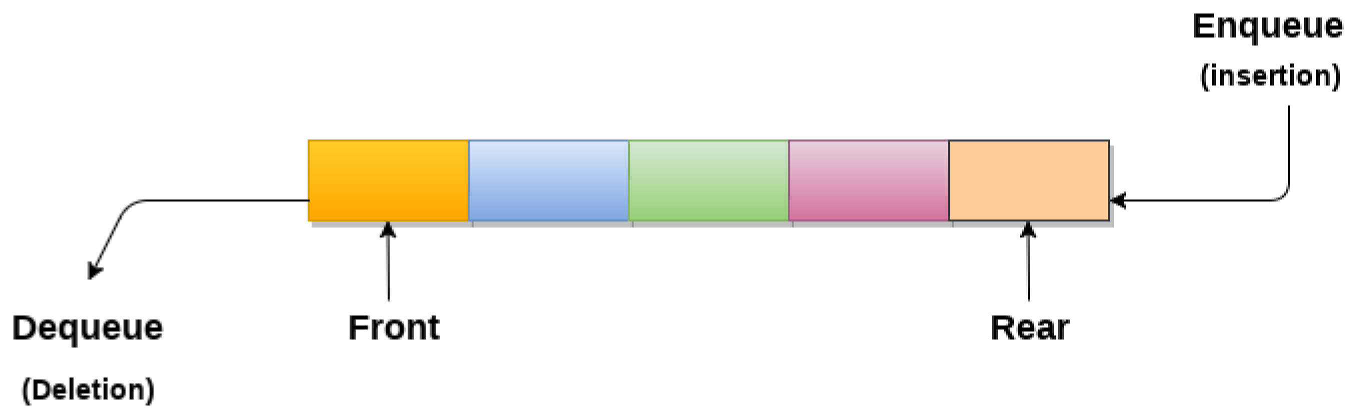
1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Code

```
void display()
{
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)
    {
        printf("Stack is empty\n");
    }
else
    {
        printf("Printing Stack elements 
\n");
        while(ptr!=NULL)
        {
            printf("%d\n",ptr->val);
            ptr = ptr->next;
        }
    }
}
```

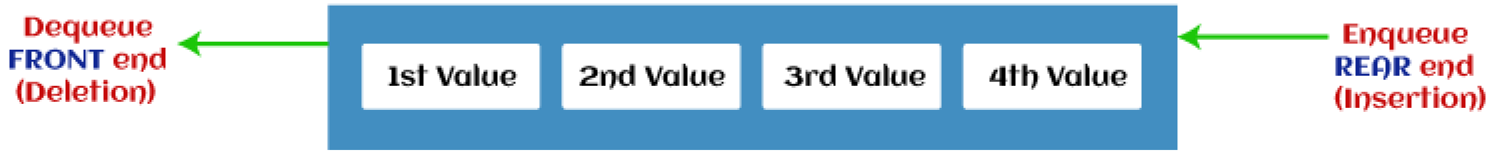
Queue

- A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
- Queue is referred to be as First In First Out list.
- For example, people waiting in line for a rail ticket form a queue.

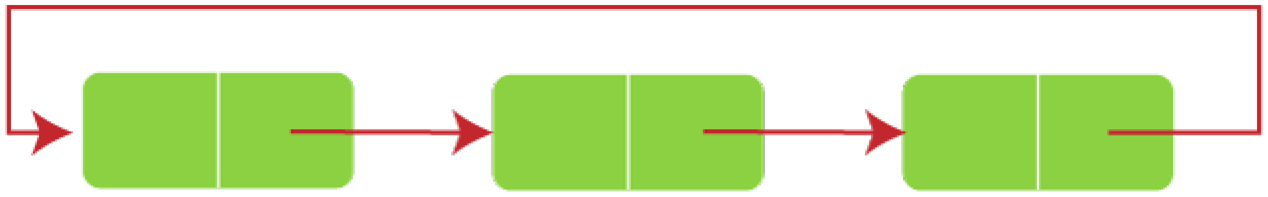


Types

- Simple Queue or Linear Queue

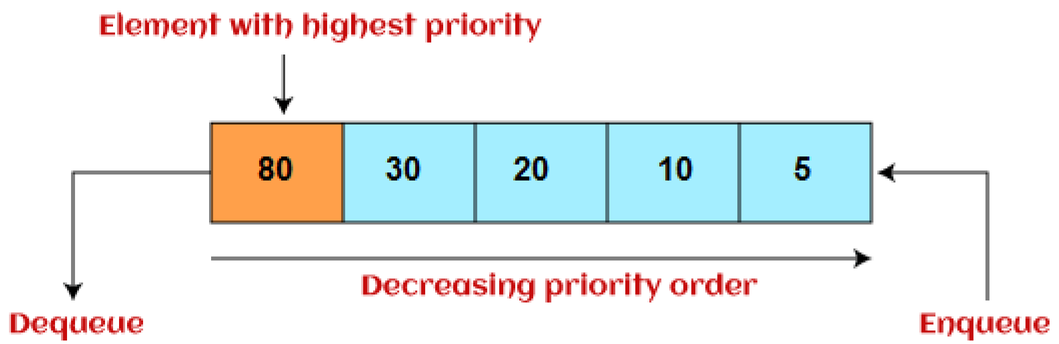


- Circular Queue

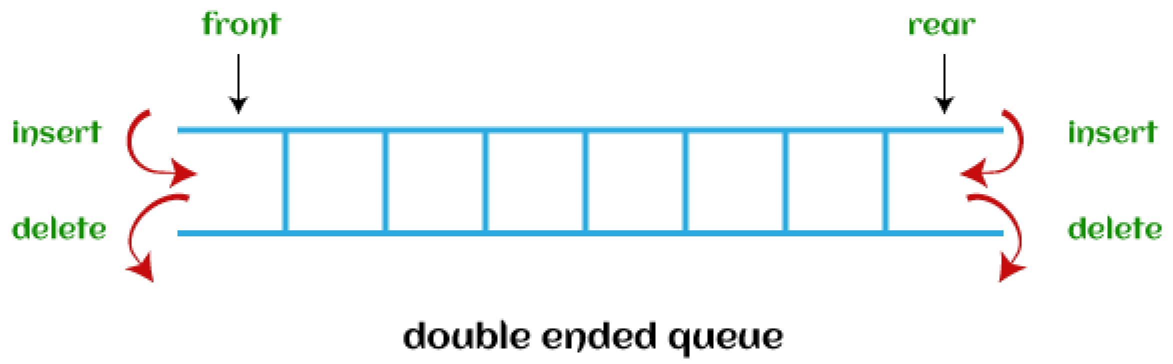

Circular Queue

Contd.

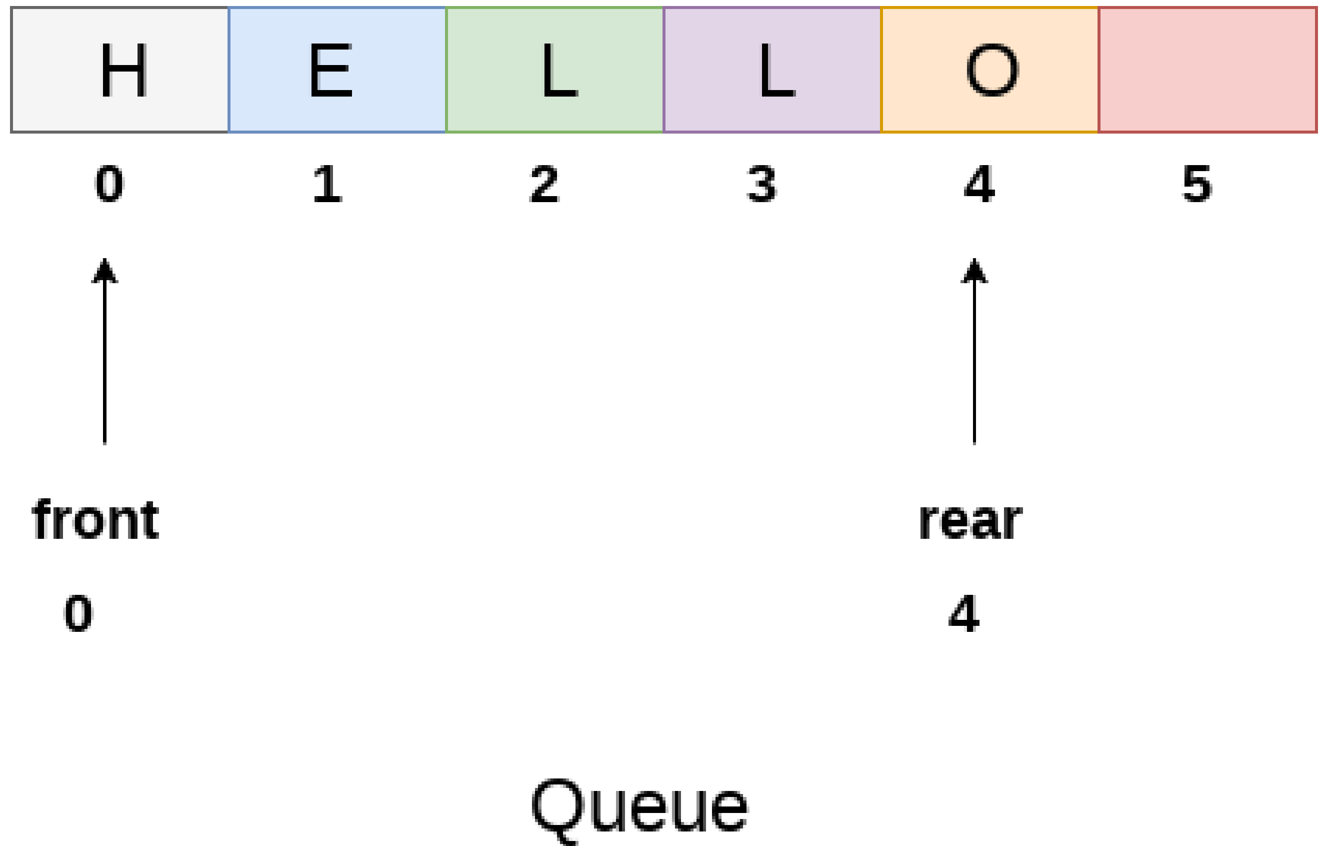
- Priority Queue



- Double Ended Queue (or Deque)



Queue using Arrays



Enqueue Code

```
void insert (int queue[], int max, int
front, int rear, int item)
{
    if (rear + 1 == max)
    {
        printf("overflow");
    }
    else
    {
        if(front == -1 && rear == -1)
        {
            front = 0;
             rear = 0;
        }
        else
        {
             rear = rear + 1;
        }
        queue[rear]=item;
    }
}
```

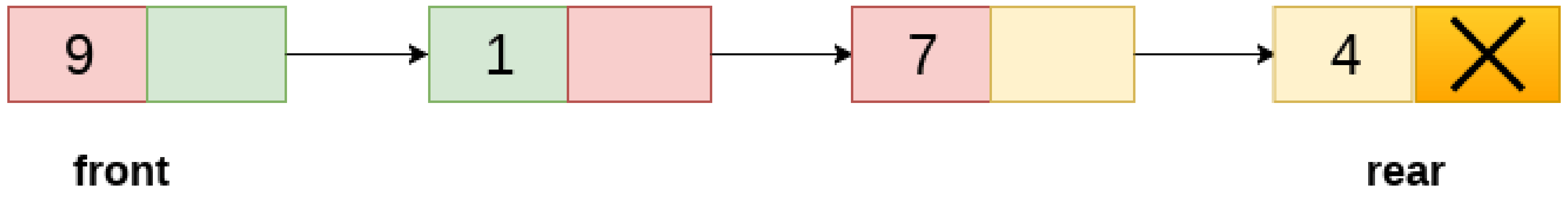
Dequeue Code

```
int delete (int queue[], int max, int
front, int rear)
{
   int y;
   if (front == -1 || front > rear)

   {
      printf("underflow");
   }
   else
   {
y = queue[front];
      if(front == rear)
      {
         front = rear = -1;
         else
         front = front + 1;

      }
      return y;
   }
}
```

Queue using Linked List



Linked Queue

Enqueue Code

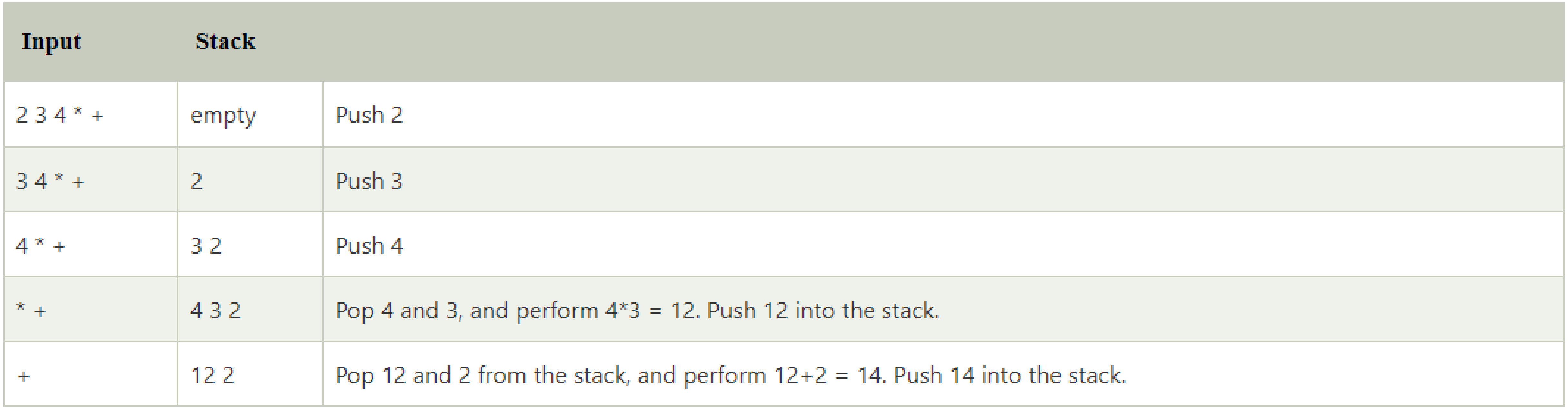
```
void insert(struct node *ptr, int item; )
{
   ptr = (struct node *) malloc (sizeof(struct
node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW\n");
      return;
   }
   else
   {
      ptr -> data = item;
      if(front == NULL)
      {
         front = ptr;
         rear = ptr;
         front -> next = NULL;
         rear -> next = NULL;
      }
      else
      {
         rear -> next = ptr;
         rear = ptr;
         rear->next = NULL;
      }
   }
}
```

Dequeue Code

```
void delete (struct node *ptr)
{
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
else
    {
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
```

Applications

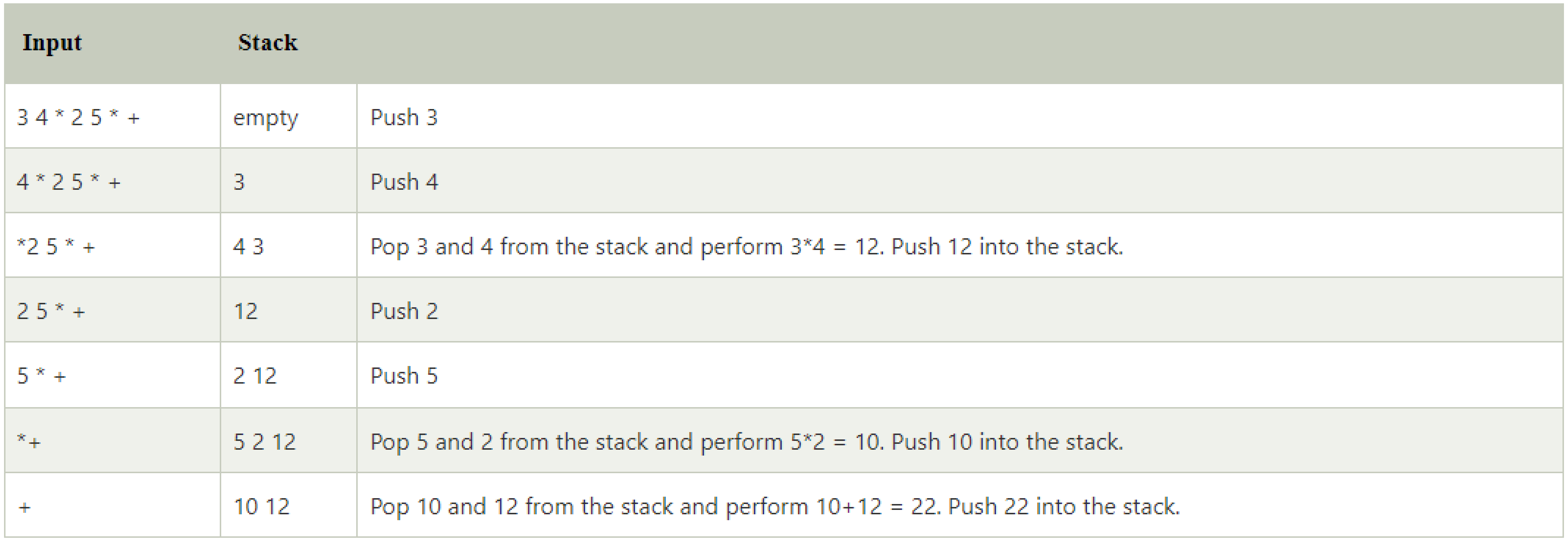
- postfix expression evaluation 2 3 4 * +



Input	Stack	
2 3 4 * +	empty	Push 2
3 4 * +	2	Push 3
4 * +	3 2	Push 4
* +	4 3 2	Pop 4 and 3, and perform 4*3 = 12. Push 12 into the stack.
+	12 2	Pop 12 and 2 from the stack, and perform 12+2 = 14. Push 14 into the stack.

Contd.

- 3 4 * 2 5 * +



Input	Stack	
3 4 * 2 5 * +	empty	Push 3
4 * 2 5 * +	3	Push 4
*2 5 * +	4 3	Pop 3 and 4 from the stack and perform 3*4 = 12. Push 12 into the stack.
2 5 * +	12	Push 2
5 * +	2 12	Push 5
*+	5 2 12	Pop 5 and 2 from the stack and perform 5*2 = 10. Push 10 into the stack.
+	10 12	Pop 10 and 12 from the stack and perform 10+12 = 22. Push 22 into the stack.

Conversion

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

Example

- Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q

Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	K L + M N* K L + M N* -
(	+ (	K L + M N *-
O	+ (	K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^
*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W
/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W*U
/	+ /	K L + M N* - O P ^W*U/
V	+ /	KL + MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/
T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

Evaluation of prefix expression

- EVALUATE_PREFIX(STRING)
- Step 1: Put a pointer P at the end of the end
- Step 2: If character at P is an operand push it to Stack
- Step 3: If the character at P is an operator pop two
- elements from the Stack. Operate on these elements
- according to the operator, and push the result
- back to the Stack
- Step 4: Decrement P by 1 and go to Step 2 as long as there
- are characters left to be scanned in the expression.
- Step 5: The Result is stored at the top of the Stack,
- return it
- Step 6: End

Example: +9*26

Character Scanned	Stack	Explanation
6	6	6 is an operand, push to Stack
2	6 2	2 is an operand, push to Stack
*	12 (6*2)	* is an operator, pop 6 and 2, multiply them and push result to Stack
9	12 9	9 is an operand, push to Stack
+	21 (12+9)	+ is an operator, pop 12 and 9 add them and push result to Stack

Conversion to prefix

- Example: K + L - M * N + (O^P) * W/U/V * T + Q

Step 1: If we are converting the expression from infix to prefix, we need first to reverse the expression. The Reverse expression would be:

Q + T * V/U/W *) P^O(+ N*M - L + K

Input expression	Stack	Prefix expression
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWPO
(	+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N
*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
-	++-	QTVUWPO^*//*NM*
L	++-	QTVUWPO^*//*NM*L
+	++-+	QTVUWPO^*//*NM*L
K	++-+	QTVUWPO^*//*NM*LK
		QTVUWPO^*//*NM*LK+-++