Stack & Queue

Operations	Stack	Queue		
		Queue	Circular Queue	Double Ended Queue (DeQue)
Definition	A stack is a linear data structure in which insertions and deletions are allowed only at the end, called the top of the stack.	Queue is a linear data structure in which the insertion is performed at the end called rear of the queue and deletion is performed at the beginning called front of the queue.		
		Linear or simple queue defines the simple operation of queue in which insertion occurs at the rear of the list and deletion occurs at the front of the list.	A circular queue is one in which the insertion of a new element is done at very first location of the queue if the last location of the queue is full.	Double ended queue (or Deque) is a special type of queue in which the insertion and deletion can be performed at either end of the queue.
ADT	When we define a stack as an ADT (Abstract Data Type), then we are only interested in knowing the stack operations from the user point of view. Means we are not interested in knowing the implementation details at this moment. We are only interested in knowing what type of operations we can perform on stack.	When defining a queue as an Abstract Data Type (ADT), the focus is on the operations from a user's perspective, abstracting away implementation details.		
Principle	It follows LIFO (Last in First Out) principle.	It follows FIFO (First in First Out) principle.		
Applications	**Application of Stack Data Structure:** • **Function calls and recursion:** When a function is called, the current state of the program is pushed onto the stack. When the function returns, the state is popped from the stack to resume the previous function’s execution.	**Application of Queue Data Structure:** • CPU scheduling, Disk Scheduling. • When data is transferred asynchronously between two processes. The queue is used for synchronization. For example: IO Buffers, pipes, file IO, etc. • Handling of interrupts in real-time systems.		

	• **Undo/Redo operations:** The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To undo the action, the top element of the stack is popped, and the reverse operation is performed. • **Expression evaluation:** Stack data structure is used to evaluate expressions in infix, postfix, and prefix notations. Operators and operands are pushed onto the stack, and operations are performed based on the stack's top elements. • **Browser history:** Web browsers use stacks to keep track of the web pages you visit. Each time you visit a new page, the URL is pushed onto the stack, and when you hit the back button, the previous URL is popped from the stack. • **Balanced Parentheses:** Stack data structure is used to check if parentheses are balanced or not. An opening parenthesis is pushed onto the stack, and a closing parenthesis is popped from the stack. If the stack is empty at the end of the expression, the parentheses are balanced. • **Backtracking Algorithms:** The backtracking algorithm uses stacks to keep track of the states of the problem-solving process. The current state is pushed onto the stack, and when the algorithm backtracks, the previous state is popped from the stack.	• Call Center phone systems use Queues to hold people calling them in order.

<table>
<tr><td rowspan="8">Time Complexity</td><td>push/enqueue</td><td>$O(1)$</td><td colspan="3">$O(1)$</td></tr>
<tr><td>pop/dequeue</td><td>$O(1)$</td><td colspan="3">$O(1)$</td></tr>
<tr><td>isFull</td><td>$O(1)$</td><td colspan="3">$O(1)$</td></tr>
<tr><td>isEmpty</td><td>$O(1)$</td><td colspan="3">$O(1)$</td></tr>
<tr><td>peek</td><td>$O(1)$</td><td colspan="3">$O(1)$</td></tr>
<tr><td>size</td><td>$O(1)$</td><td colspan="3">$O(1)$</td></tr>
<tr><td>Incremental Strategy: push [Amortized Time]</td><td>$O(n)$</td><td colspan="3">$O(1)$</td></tr>
<tr><td>Doubling Strategy: push [Amortized Time]</td><td>$O(1)$</td><td colspan="3">$O(1)$</td></tr>
<tr><td>Space Complexity</td><td>n push/enqueue</td><td>$O(n)$</td><td colspan="3">$O(n)$</td></tr>
<tr><td rowspan="2">Array Implementation Basic Operations Code</td><td>push/enqueue</td><td>

```
void push(int data)
{
  if (isFull())
  {
    printf("Stack Overflow\n");
  }
  else
  {
    top = top + 1;
    stack_arr[top] = data;
    printf("Pushed %d into the stack.\n",
data);
  }
}
```

</td><td>

```
void enqueue(int data)
{
  if (isFull())
  {
    printf("Queue
Overflow\n");
    exit(1);
  }
  if (front == -1)
  {
    front = 0;
  }
  rear = rear + 1;
  queue[rear] = data;
}
```

</td><td>

```
void enqueue(int data)
{
  if (isFull())
  {
    printf("Queue Overflow\n");
    exit(1);
  }
  if (front == -1)
  {
    front = 0;
  }
  if (rear == MAX - 1)
  {
    rear = 0;
  }
  else
  {
    rear = rear + 1;
  }
  circular_queue[rear] = data;
}
```

</td><td>

```
void enqueueFront(int data)
{
  if (isFull())
  {
    printf("Queue Overflow\n");
    exit(1);
  }
  if (front == -1)
  {
    front = 0;
    rear = 0;
  }
  else if (front == 0)
  {
    front = MAX - 1;
  }
  else
  {
    front = front - 1;
  }
  deque[front] = data;
}
```

</td></tr>
<tr><td colspan="3">

```
void enqueueRear(int data)
{
  if (isFull())
  {
    printf("Queue Overflow\n");
    exit(1);
  }
  if (front == -1)
  {
    front = 0;
    rear = 0;
```

</td></tr>
</table>

					}
					else if (rear == MAX - 1)
					{
					rear = 0;
					}
					else
					{
					rear = rear + 1;
					}
					deque[rear] = *data*;
					}

	pop/dequeue	*int* pop()
		{
		if (isEmpty())
		{
		printf("Stack Underflow\n");
		exit(1);
		}
		else
		{
		int value = stack_arr[top];
		top = top - 1;
		return value;
		}
		}
			int dequeue()
			{
			int data;
			if (isEmpty())
			{
			printf("Queue Underflow\n");
			exit(1);
			}
			data = queue[front];
			{
			front++;
			}
			return data;
			}
				int dequeue()
				{
				int data;
				if (isEmpty())
				{
				printf("Queue Underflow\n");
				exit(1);
				}
				data = circular_queue[front];
				if (front == rear)
				{
				front = -1;
				rear = -1;
				}
				else if (front == MAX - 1)
				{
				front = 0;
				}
				else
				front = front + 1;
				return data;
				}
					int dequeueFront()
					{
					int data;
					if (isEmpty())
					{
					printf("Queue Underflow\n");
					exit(1);
					}
					data = deque[front];
					if (front == rear)
					{
					front = -1;
					rear = -1;
					}
					else if (front == MAX - 1)
					{
					front = 0;
					}
					else
					front = front + 1;
					return data;
					}

					int dequeueRear()
					{
					int data;
					if (isEmpty())
					{
					printf("Queue Underflow\n");
					exit(1);
					}
					data = deque[rear];
					if (front == rear)
					{
					front = -1;
					rear = -1;
					}

					else if (rear == 0)
					{
					rear = MAX - 1;
					}
					else
					rear = rear - 1;
					return data;
					}

	isFull	int isFull()
		{
		if (top == MAX - 1)
		{
		return 1;
		}
		else
		{
		return 0;
		}
		}
			int isFull()
			{
			if (rear == MAX - 1)
			{
			return 1;
			}
			else
			{
			return 0;
			}
			}
				int isFull()
				{
				if ((front == 0 && rear == MAX - 1) || (front == rear + 1))
				{
				return 1;
				}
				else
				{
				return 0;
				}
				}	

	isEmpty	int isEmpty()
		{
		if (top == -1)
		{
		return 1;
		}
		else
		{
		return 0;
		}
		}
			int isEmpty()
			{
			if (front == -1 ||
			front == rear + 1)
			{
			return 1;
			}
			else
			{
			return 0;
			}
			}
				int isEmpty()
				{
				if (front == -1)
				{
				return 1;
				}
				else
				{
				return 0;
				}
				}	

	peek	`int peek()`
		`{`
		`  if (isEmpty())`
		`  {`
		`  printf("Stack Underflow\n");`
		`  exit(1);`
		`  }`
		`  else`
		`  {`
		`  return stack_arr[top];`
		`  }`
		`}`
			`int peek()`
			`{`
			`  if (isEmpty())`
			`  {`
			`  printf("Queue Underflow\n");`
			`  exit(1);`
			`  }`
			`  return queue[front];`
			`}`
				`int peek()`
				`{`
				`  if (isEmpty())`
				`  {`
				`  printf("Queue Underflow\n");`
				`  exit(1);`
				`  }`
				`  return circular_queue[front];`
				`}`
					`int peek()`
					`{`
					`  if (isEmpty())`
					`  {`
					`  printf("Queue Underflow\n");`
					`  exit(1);`
					`  }`
					`  return deque[front];`
					`}`

	print	`void print()`
		`{`
		`  if (isEmpty())`
		`  {`
		`  printf("Stack is empty\n");`
		`  return;`
		`  }`
		`  else`
		`  {`
		`  printf("Stack elements :\n\n");`
		`  for (int i = top; i >= 0; i--)`
		`  printf("%d\n", stack_arr[i]);`
		`  printf("\n");`
		`  }`
		`}`
			`void print()`
			`{`
			`  int i;`
			`  if (isEmpty())`
			`  {`
			`  printf("Queue Underflow\n");`
			`  exit(1);`
			`  }`
			`  printf("Queue: ");`
			`  for (i = front; i <= rear; i++)`
			`  {`
			`  printf("%d ", queue[i]);`
			`  }`
			`  printf("\n");`
			`}`
				`void print()`
				`{`
				`  int temp;`
				`  if (isEmpty())`
				`  {`
				`  printf("Queue Underflow\n");`
				`  exit(1);`
				`  }`
				`  temp = front;`
				`  if (front <= rear)`
				`  {`
				`  while (temp <= rear)`
				`  {`
				`  printf("%d ", circular_queue[temp]);`
				`  temp++;`
				`  }`
				`  }`
				`  else`
				`  {`
				`  while (temp <= MAX - 1)`
				`  {`
				`  printf("%d ", circular_queue[temp]);`
					`void print()`
					`{`
					`  int temp;`
					`  if (isEmpty())`
					`  {`
					`  printf("Queue Underflow\n");`
					`  exit(1);`
					`  }`
					`  temp = front;`
					`  if (front <= rear)`
					`  {`
					`  while (temp <= rear)`
					`  {`
					`  printf("%d ", deque[temp]);`
					`  temp++;`
					`  }`
					`  }`
					`  else`
					`  {`
					`  while (temp <= MAX - 1)`
					`  {`
					`  printf("%d ", deque[temp]);`
					`  temp++;`
					`  }`
					`  temp = 0;`

				temp++;
				}
				temp = 0;
				while (temp <= rear)
				{
				printf("%d ", circular_queue[temp]);
				temp++;
				}
				}
				printf("\n");
				}
					while (temp <= rear)
					{
					printf("%d ", deque[temp]);
					temp++;
					}
					}
					printf("\n");
					}