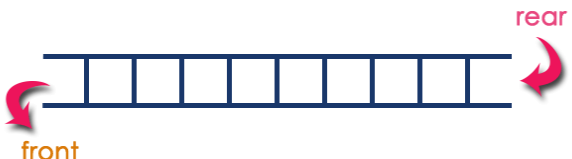


St. Peter's Engineering College (Autonomous) Dullapally (P), Medchal, Hyderabad – 500100. QUESTION BANK				Dept.	:	CSE, CSM, CSD, CSC
				Academic Year 2023-24		
Subject Code	:	AS22-05ES07	Subject	:	Data Structures	
Class/Section	:	B.Tech.	Year	:	I	Semester : II

BLOOMS LEVEL					
Remember	L1	Understand	L2	Apply	L3
Analyze	L4	Evaluate	L5	Create	L6

Q. No	Question (s)	Marks	BL	CO
UNIT – II				
1	a) Define Stack with an example.	1M	L1	C123.2
	A stack is an abstract data type that holds an ordered, linear sequence of items. In contrast to a <u>queue</u> , a stack is a last in, first out (LIFO) structure. A real-life example is a stack of plates: you can only take a plate from the top of the stack, and you can only add a plate to the top of the stack.			
	b) Define Polish Notation and Write any prefix Expression.	1M	L3	C123.2
	Polish Notation is a general form of expressing mathematical, logical and algebraic equations. * + A B + C D			
	c) Define Queue with suitable diagram.	1M	L4	C123.2
	Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing elements are performed at two different positions. <div style="text-align: center;">  </div>			

	d) Define Circular Queue.	1M	L2	C123.3
	A circular queue is a linear data structure in which the operations are performed based on FIFO principle and the last position is connected back to the first position to make a circle.			
	e) What is heap?	1M	L1	C123.4
	A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap.			
2	a) Explain in detail about Stack ,Write the Operations that allows to perform different actions on stack.	3M	L3	C123.2
	<p>A stack is an abstract data type that holds an ordered, linear sequence of items. In contrast to a <u>queue</u>, a stack is a last in, first out (LIFO) structure. A real-life example is a stack of plates: you can only take a plate from the top of the stack, and you can only add a plate to the top of the stack. If you want to reach a plate that is not on the top of the stack, you need to remove all of the plates that are above that one.</p> <p>In the same way, in a stack data structure, you can only access the element on the top of the stack. The element that was added last will be the one to be removed first. Therefore, to implement a stack, you need to maintain a pointer to the top of the stack (the last element to be added).</p> <div data-bbox="384 984 1448 1537" data-label="Diagram"> <h3 style="text-align: center;">Stack in Data structure</h3> <p>The diagram shows five stages of a stack's state:</p> <ul style="list-style-type: none"> empty stack: Top = -1. The stack is empty. push: Top = 0, stack[0]=1. An arrow points to the stack with the number 1 inside. push: Top = 1, stack[1]=2. An arrow points to the stack with elements 1 and 2 inside. push: Top = 2, stack[2]=3. An arrow points to the stack with elements 1, 2, and 3 inside. pop: Top = 1, return stack[2]. An arrow points to the stack with elements 1 and 2 inside, and the number 3 is shown being removed from the top. </div> <p>There are some basic operations that allow us to perform different actions on a stack.</p> <p>Push: Add an element to the top of a stack</p> <p>Pop: Remove an element from the top of a stack</p> <p>Peek: Get the value of the top element without removing it.</p> <p>IsEmpty: Check if the stack is empty.</p> <p>IsFull: Check if the stack is full.</p>			
	b) Write a algorithm for Tower of Hanoi.	3M	L2	C123.2

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 \rightarrow 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.

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

	c) Write a steps to display the elements of Circular Queue.	3M	L4	C123.2
	<p>display() - Displays the elements of a Circular Queue</p> <p>We can use the following steps to display the elements of a circular queue...</p> <ul style="list-style-type: none"> • Step 1 - Check whether queue is EMPTY. (front == -1) • Step 2 - If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function. • Step 3 - If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'. • Step 4 - Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i <= rear' becomes FALSE. • Step 5 - If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i <= SIZE - 1' becomes FALSE. • Step 6 - Set i to 0. • Step 7 - Again display 'cQueue[i]' value and increment i value by one (i++). Repeat the same until 'i <= rear' becomes FALSE. 			
	d) What is the difference between direct and indirect recursion?	3M	L2	C123.3
	<p>A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun_new and fun_new calls fun directly or indirectly. The difference between direct and indirect recursion has been illustrated in Table 1.</p> <p>// An example of direct recursion</p> <pre>void directRecFun() { // Some code.... directRecFun(); // Some code... }</pre> <p>// An example of indirect recursion</p> <pre>void indirectRecFun1() { // Some code... indirectRecFun2(); // Some code... } void indirectRecFun2() { // Some code... indirectRecFun1(); // Some code... }</pre>			

	e) Write the steps for implementing the circular Queue using Array.	3M	L2	C123.4
	<p>To implement a circular queue data structure using an array, we first perform the following steps before we implement actual operations.</p> <ul style="list-style-type: none"> • Step 1 - Include all the header files which are used in the program and define a constant 'SIZE' with specific value. • Step 2 - Declare all user defined functions used in circular queue implementation. • Step 3 - Create a one dimensional array with above defined SIZE (int cQueue[SIZE]) • Step 4 - Define two integer variables 'front' and 'rear' and initialize both with '-1'. <p>(int front = -1, rear = -1)</p> <ul style="list-style-type: none"> • Step 5 - Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue. 			

3	a) Explain about Polish Notations with their types.	5M	L3	C123.2

Polish Notation:

Polish Notation is a general form of expressing mathematical, logical and algebraic equations. The compiler uses this notation in order to evaluate mathematical expressions depending on the order of operations. There are in general three types of Notations used while parsing Mathematical expressions:

- Infix Notation
- Prefix Notation
- Postfix Notation

Infix Expression	Prefix Expression	Postfix Expression
$A + B * C + D$	$++ A * B C D$	$A B C * + D +$
$(A + B) * (C + D)$		$A B + C D + *$
$A * B + C * D$	$+ * A B * C D$	$A B * C D * +$
$A + B + C + D$	$+++ A B C D$	$A B + C + D +$

Infix Notation

Infix Notation or Expression is where the operators are written in between every pair of operands. It is the usual way to write an expression generally written with parentheses. For Ex: An expression like $X+Y$ is an Infix Expression, where $+$ is an Operator and X, Y are Operands.
Example

Now, let us look at an example on how to evaluate a Polish Notation or Prefix Expression to get the result.

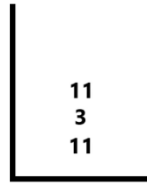
Consider this Expression: $/ * + 5 6 3 11$. While evaluating the expression we take decision for two cases: When the Character is an Operand or When the Character is an Operator. Let us look at the steps.

Step 1:

We will use a Stack for this evaluation. We scan the Expression from right to left, if the current character is an Operand we push it into the stack. So from 11 to 5 we push the elements into the stack. The stack looks:

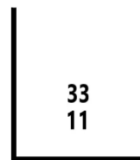
**Step 2:**

As soon as we get an operator we multiply its previous two elements, so continuing traversing from right to left we first get '+' operator so we pop two elements from stack (5 & 6) compute their result with the operator i.e. $5+6 = 11$, and push the result back into the stack for future evaluation. The Stack now is:



Step 3:

The next Operator is '*' Operator (Multiply), so we again pop the two elements from stack and repeating the process of Step 2. So we compute the result from their operation ($11 * 3 = 33$) and push it back to the stack again. The stacks now look like:



Step 4:

Finally, we have the '/' operator so we pop 33 and 11 compute the result push it back to the stack. Now we have reached the leftmost or start index of the expression so at this point our stack will contains only one value which will be our Resultant Evaluated Prefix Expression.



Postfix Evaluation:

Postfix Expression is also known as **Reverse Polish Notation**. These are the expression where the Operands precede the Operators i.e. the Operands are written before the Operators. For Example: The Infix $X+Y$ will be represented in Postfix or Reverse Polish as $XY+$

Example

Consider the following Infix Expression to be converted into Postfix Expression...

$$D = A + B * C$$

- 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 + ----- $D = A B C * +$
- Step 5 - Convert the next operator = ----- $D A B C * + =$

	b) Write a program in C for stack Reversing using Recursive function.	5M	L3	C123.2
	<pre> #include <stdio.h> #include <stdlib.h> struct node { int a; struct node *next; }; void generate(struct node **); void display(struct node *); void stack_reverse(struct node **, struct node **); void delete(struct node **); int main() { struct node *head = NULL; generate(&head); printf("\nThe sequence of contents in stack\n"); display(head); printf("\nInversing the contents of the stack\n"); if (head != NULL) { stack_reverse(&head, &(head->next)); } printf("\nThe contents in stack after reversal\n"); display(head); delete(&head); return 0; } void stack_reverse(struct node **head, struct node **head_next) { struct node *temp; if (*head_next != NULL) { temp = (*head_next)->next; (*head_next)->next = (*head); *head = *head_next; *head_next = temp; stack_reverse(head, head_next); } } void display(struct node *head) </pre>			


```

{
    if (head != NULL)
    {
        printf("%d ", head->a);
        display(head->next);
    }
}

void generate(struct node **head)
{
    int num, i;
    struct node *temp;
    printf("Enter length of list: ");
    scanf("%d", &num);
    for (i = num; i > 0; i--)
    {
        temp = (struct node *)malloc(sizeof(struct node));
        temp->a = i;
        if (*head == NULL)
        {
            *head = temp;
            (*head)->next = NULL;
        }
        else
        {
            temp->next = *head;
            *head = temp;
        }
    }
}

void delete(struct node **head)
{
    struct node *temp;
    while (*head != NULL)
    {
        temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}

```

c) Explain the Process of Tower of Hanoi with example program.

5M

L2

C123.3

Tower of Hanoi:

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.

Tower of Hanoi puzzle with n disks can be solved in minimum $2^n - 1$ steps. This presentation shows that a puzzle with 3 disks has taken $2^3 - 1 = 7$ 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 $\rightarrow 1$ 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.

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

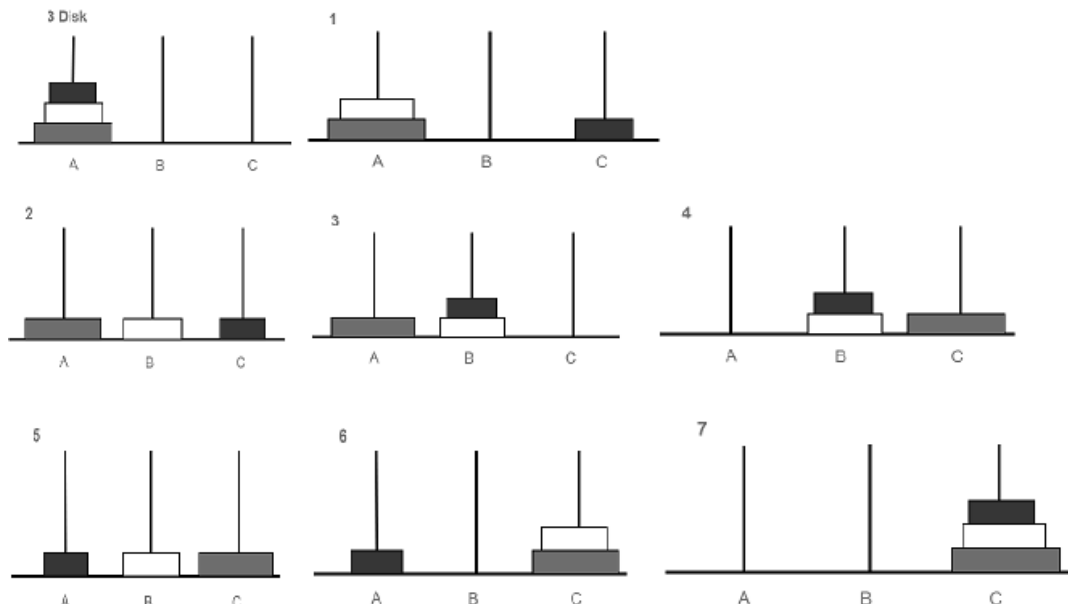
move disk from source to dest // Step 2

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

END IF

END Procedure

STOP



Sample Program

```
#include <stdio.h>
void toH(int n, char rodA, char rodC, char rodB)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c",rodA ,rodC );
        return;
    }
    toH(n-1, rodA, rodB, rodC);
    printf("\n Move disk %d from rod %c to rod %c", n, rodA, rodC);
    toH(n-1, rodB, rodC,rodA);
}

int main()
{
    int no_of_disks ;
    printf("Enter number of disks: ");
    scanf("%d", &no_of_disks);
    toH(no_of_disks, 'A','C','B');
    return 0;
}
```

Output:

```
Enter number of disks: 3
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
```

	d) Write an Operations on a Queue using Array.	5M	L3	C123.3
	<p>Queue Operations using Array</p> <p>Queue data structure using array can be implemented as follows...</p> <p>Before we implement actual operations, first follow the below steps to create an empty queue.</p> <ul style="list-style-type: none"> • Step 1 - Include all the header files which are used in the program and define a constant 'SIZE' with specific value. • Step 2 - Declare all the user defined functions which are used in queue implementation. • Step 3 - Create a one dimensional array with above defined SIZE (int queue[SIZE]) • Step 4 - Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1) • Step 5 - Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue. <p>enQueue(value) - Inserting value into the queue</p> <p>In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...</p> <ul style="list-style-type: none"> • Step 1 - Check whether queue is FULL. (rear == SIZE-1) • Step 2 - If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function. • Step 3 - If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value. <p>deQueue() - Deleting a value from the Queue</p> <p>In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from front position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...</p> <ul style="list-style-type: none"> • Step 1 - Check whether queue is EMPTY. (front == rear) • Step 2 - If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function. • Step 3 - If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1). <p>display() - Displays the elements of a Queue</p> <p>We can use the following steps to display the elements of a queue...</p> <ul style="list-style-type: none"> • Step 1 - Check whether queue is EMPTY. (front == rear) • Step 2 - If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function. • Step 3 - If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'. • Step 4 - Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value reaches to rear (i <= rear) 			

--	--

	<p>e) Convert the following infix expression to postfix expression.</p> <p>K + L - M*N + (O^P) * W/U/V * T +Q</p> <p>Ans:</p> <p>Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q</p> <table><tr><th>Input Expression</th><th>Stack</th><th>Postfix Expression</th></tr><tr><td>k</td><td></td><td>K</td></tr><tr><td>+</td><td>+</td><td></td></tr><tr><td>L</td><td>+</td><td>KL</td></tr><tr><td>-</td><td>-</td><td>KL+</td></tr><tr><td>M</td><td>-</td><td>KL+M</td></tr><tr><td>*</td><td>-*</td><td>KL+M</td></tr><tr><td>N</td><td>-*</td><td>KL+MN</td></tr><tr><td>+</td><td>+</td><td>KL+MN* KL+MN*-</td></tr><tr><td>(</td><td>+(</td><td>KL+MN*-</td></tr><tr><td>O</td><td>+(</td><td>K L + M N * - O</td></tr><tr><td>^</td><td>+(^</td><td>K L + M N * - O</td></tr><tr><td>P</td><td>+(^</td><td>K L + M N * - O P</td></tr><tr><td>)</td><td>+</td><td>K L + M N * - O P ^</td></tr><tr><td>*</td><td>+</td><td>K L + M N * - O P ^</td></tr><tr><td>W</td><td>+</td><td>K L + M N * - O P ^ W</td></tr><tr><td>/</td><td>+/</td><td>K L + M N * - O P ^ W *</td></tr><tr><td>U</td><td>+/</td><td>K L + M N * - O P ^ W * U</td></tr><tr><td>/</td><td>+/</td><td>K L + M N * - O P ^ W * U /</td></tr><tr><td>V</td><td>+/</td><td>KL + MN*-OP^W*U/V</td></tr><tr><td>*</td><td>+</td><td>KL+MN*-OP^W*U/V/</td></tr><tr><td>T</td><td>+</td><td>KL+MN*-OP^W*U/V/T</td></tr><tr><td>+</td><td>+</td><td>KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+</td></tr><tr><td>Q</td><td>+</td><td>KL+MN*-OP^W*U/V/T*Q</td></tr><tr><td></td><td></td><td>KL+MN*-OP^W*U/V/T*+Q+</td></tr><tr><td></td><td></td><td></td></tr></table>	Input Expression	Stack	Postfix Expression	k		K	+	+		L	+	KL	-	-	KL+	M	-	KL+M	*	-*	KL+M	N	-*	KL+MN	+	+	KL+MN* KL+MN*-	(+(KL+MN*-	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+				5M	L2	C123.3
Input Expression	Stack	Postfix Expression																																																																																
k		K																																																																																
+	+																																																																																	
L	+	KL																																																																																
-	-	KL+																																																																																
M	-	KL+M																																																																																
*	-*	KL+M																																																																																
N	-*	KL+MN																																																																																
+	+	KL+MN* KL+MN*-																																																																																
(+(KL+MN*-																																																																																
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+																																																																																
4	a) Explain in detail about Priority Queue.	10M	L2	C123.3																																																																														

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

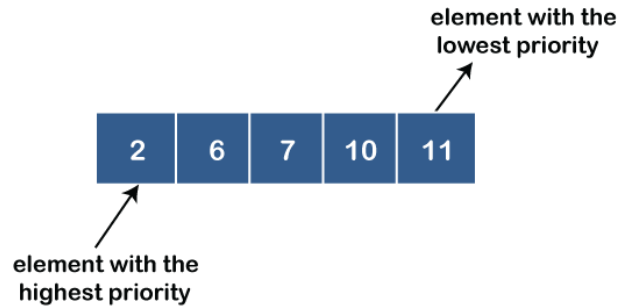
All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **poll()**: This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2)**: This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll()**: It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5)**: It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

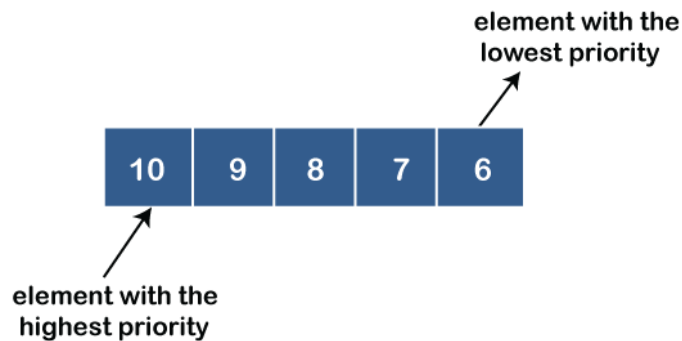
Types of Priority Queue:

There are two types of priority queue:

- **Ascending order priority queue**: In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Representation of priority queue

The priority queue by using the list given below in which **INFO** list contains the data elements, **PNR** list contains the priority numbers of each data element available in the **INFO** list, and **LINK** basically contains the address of the next node.

	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

create the priority queue step by step.

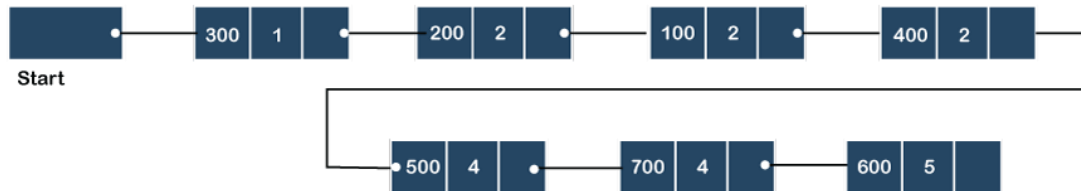
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 300, priority number 2 is having a higher priority, and data values associated with this priority are 200 and 100. So, this data will be inserted based on the FIFO principle; therefore 200 will be added first and then 100.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 400, 500, 700. In this case, elements would be inserted based on the FIFO principle; therefore, 400 will be added first, then 500, and then 700.

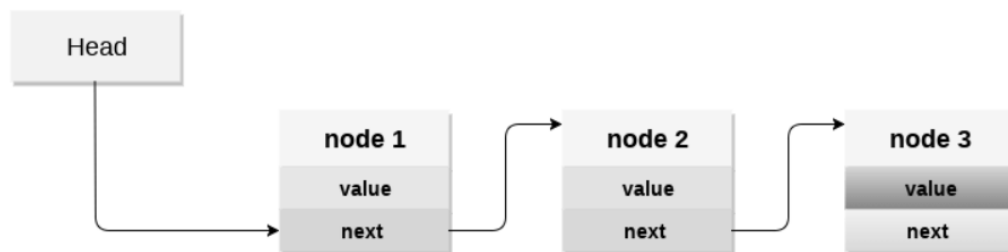
Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 600, so it will be inserted at the end of the queue.



b) Write Linked List Implementation of Stack. (10 M)

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1** - Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2** - Define a '**Node**' structure with two members **data** and **next**.
- **Step 3** - Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4** - Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.



push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1** - Create a **newNode** with given value.
- **Step 2** - Check whether stack is **Empty** (**top == NULL**)
- **Step 3** - If it is **Empty**, then set **newNode** → **next** = **NULL**.
- **Step 4** - If it is **Not Empty**, then set **newNode** → **next** = **top**.
- **Step 5** - Finally, set **top** = **newNode**.

```
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() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1** - Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2** - If it is **Empty**, then display "**Stack is Empty!!! Deletion is not possible!!!**" and terminate the function
- **Step 3** - If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4** - Then set '**top = top → next**'.
- **Step 5** - Finally, delete '**temp**'. (**free(temp)**).

```

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() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1** - Check whether stack is **Empty** (**top == NULL**).
- **Step 2** - If it is **Empty**, then display '**Stack is Empty!!!**' and terminate the function.
- **Step 3** - If it is **Not Empty**, then define a Node pointer '**temp**' and initialize with **top**.
- **Step 4** - Display '**temp → data --->**' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next != NULL**).
- **Step 5** - Finally! Display '**temp → data ---> NULL**'.

```

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;
        }
    }
}

```

--	--