Name : Sagar Bhatiya

Name - Sagar Bhatiya
CRN: 93
Sec: C

Assignment 2

Part 1 : Queues

1. Theory Questions :

Q.a) Explain the concept of Queue. What are its main characteristics ?

Ans. A Queue is a linear data structure that follows the First In, First Out (FIFO) principle, meaning the first element added to the queue will be the first one to be removed. It is used in scenarios where the order of processing is important, such as task scheduling or handling requests in servers.

Main Characteristics:

1. FIFO Structure : The first element inserted is the first to be removed.

2. Two main operations :
   - Enqueue : Adding an element to the end (rear) of the queue.
   - Dequeue : Removing an element from the front of the queue.

3. Dynamic Size : Depending on the implementation, the queue can grow or shrink dynamically (linked list) or have a fixed size (array-based).

Name : Sagar Bhatiya
Sec : C
CRN: 93

Qs b) What is a two-way header list? Explain its structure.

Ans. A two-way header list (also known as a doubly linked list with a header) is a type of doubly linked list that uses a special header node to simplify certain operations, such as list initialization and boundry conditions(like insertion or deletion at the head or tail).

Its Structure :

1. Node : Each node contain 3 fields:
   - Data : Stores value or data.
   - Next Pointer : Points to next node in the list.
   - Previous Pointer : Points to previous node in the list.

2. Header Node :
   - The header Node does not contain actual data but serves as a sentinel node.
   - Its next pointer points to the first node, & its previous pointer points to the last node making the list circular in nature.
   - If the list is empty, both the next & previous pointers of the header node point to itself.

Name: Sagar Bhatiya
CRN: 93
Sec: C

Qs b) Differentiate between Array representation and linked representation of queues.?

Ans. Array Representation of Queue:

1. Fixed Size: The size of the queue is fixed and defined at the time of creation.
2. Contiguous Memory: Elements are stored in contiguous memory location.
3. Efficient Access: Direct access to elements via index, making operations fast.
4. Overflow: The queue can suffer from overflow if the array becomes full.
5. Memory Wastage: After several enqueue and dequeue operations, unused space may accumulate (if not handled as a circular queue).

Linked Representation of Queues:

1. Dynamic Size: The queue can grow or shrink dynamically based on the number of elements.
2. Non-contiguous Memory: Elements are stored in nodes, with each node pointing to next.

Name: SagarBhatiya
CRN: 93
Sec: C

3. Efficient memory use: No predefined size, so memory is allocated only when needed.

4. No overflow: Unless system memory is exhausted, there's no limit on queue size.

5. Pointer overhead: Each node requires extra memory for pointers, leading to some overhead.

Hence, array provide fast access but are limited in size, while linked lists offer dynamic sizing at the cost of extra memory for pointers.

Name : Sagar Bhatiya
CRN: 93
Sec : C

Qs c) What is a circular Queue? How does it overcome the limitation of simple queue?

Ans. A circular Queue is a type of queue where the last position is connected back to first position, forming a circle. This structure allows more efficient use of memory compared to a simple queue.

Key characteristics:

1. Circular Structure: The rear wraps around to the front when it reaches the end of the array, allowing reuse of previously freed spaces.

2. Efficient Memory usage : unlike a simple linear queue, where memory can be wasted after several enqueue & dequeue operations, a circular queue utilizes all available space.

How it overcomes Simple Queue limitations

- No memory wastage : In a simple queue, after repeated dequeue operations the front pointer moves forward, potentially

Name: Sagar Bhatiya
CRN: 93
Sec: C

leaving unused space at the beggining. A circular queue eliminates the issue by reusing those free spaces.

- Efficient Wrap around:

When the rear reaches the end of the queue in a circular queue, it wraps around to the beginning if there is space, making it more efficient for continuous insertions & deletions.

This solves the problem of "queue overflow" in fixed size arrays when there are empty slots in the queue but rear pointer is at the end.

Name: Sagar Bhaliya
CRN: 93
Sec: C

Qs d) Explain the concept of a Dequeue. How is it different from a normal queue?

Ans. A Dequeue (short for Double-Ended Queue) is a linear data structure that allows insertion and deletion of elements from both ends - the front & rear.

Difference from a Normal Queue:

1. Normal Queue (FIFO): only allows insertion at the rear and deletion from front (First in, First out).

2. Dequeue: Allows insertion and deletion from both the front and rear, making it more flexible compared to a normal queue.

Dequeue can function as both a Stack (LIFO) and a queue (FIFO), depending on how operat-ions are used.

Name: Sagar Bhatija
CRN: 93
Sec: C

Qse) Describe the working of a priority queue & its applications.

Ans. A priority queue is a type of data structure where each element is associated with a priority, and elements are dequeued based on their priority rather than their insertion order.

Working:

1. Elements Insertion: Elements are inserted with a specific priority.
2. Dequeue operations: The element with the highest priority is removed first. If two elements have same priority, they are dequeued based on their order of arrival (FIFO for equal priorites).
3. Implementation: Priority queues can be implem-ented using:

- Heaps (binary heaps for efficient access).
- Arrays or Linked Lists (though less efficient)

Applications:

1) Task Scheduling: Operating systems use priority queues to manage

Name: Sagar Bhatiya
Sec: C
CRN: 93

task by prioritizing them based on their deadlines or importance.

2) Process scheduling: Tasks are assigned priorities based on their importance or urgency.

3) graph algorithms: Dijkstra's algorithm for shortest path finding uses a priority queue to explore vertices in order of their distance from the source.

4) Data compression: Huffman coding uses a priority queue to determine the frequency of characters & assign codes according.

Name: Sagar Bhatiya
CRN: 93
Sec: C

Qs f) What is the significance of converting an infix expression to postfix? Why is postfix notation preferred in some applications (eg, for stack-based evaluation)?

Ans. Converting an infix expression to postfix (also known as Reverse Polish Notation RPN) is significant because it simplifies the evaluation process, especially in computational applications that use stacks.

Here's why postfix notation is preferred:

1. No Need for Parentheses: In infix notation, parentheses are used to enforce precedence, which makes parsing and evaluating complex expressions harder. Postfix eliminates the need for parentheses, as the order of operations is inherent in the notation.

2. Efficient Stack-Based Evaluation: Postfix notation is ideal for stack-based systems because operators come after their operands. This allows the expression to be evaluated in a single left to right pass using a stack, pushing operands onto the stack and applying

Name: Sagar Bhatiya

Name: Sagar Bhatiya
CRN: 93
Sec: C

operators when encountered. This makes evaluation faster and simpler.

3. Avoiding Repeated Scanning: Infix notation requires multiple scans & lookups to resolve operator precedence and associativity, whereas postfix can be evaluated in one pass without backtracking or reprocessing.

Postfix notation enables easier and more efficient evaluation, especially in systems like calculators & compilers that rely on stack-based algorithms.

Name: Sagar Bhatiya
Sec: C
CRN: 93

Part 2 Linked list

Theory Questions:

Qs a) Explain the singly linked list. How does it differ from an array in terms of memory and performace?

Ans. A singly linked list is a dynamic data structure consisting of a sequence of nodes. Each node consist of:

1. Data : The value stored in node.
2. Next : A pointer to the next node in the sequence.

The first node is called the head. The last node's ~~head~~ next pointer points to NULL, indicating the end of the list. Singly linked lists allow for dynamic memory allocation and efficient insertion and deletion operations.

- Array uses contiguous memory while linked list uses non-contiguous memory.
- Linked list offer better flexibility with dynamic memory management and efficient insertion/deletion but have slower access times and slightly higher memory usage due to pointers. Arrays on other hand provide fast access but have fixed size and higher cost for insertion/deletion

Name: Sagar Bratiya
CRN: 93
Sec: C

Qsc What is the difference between traversal & searching in a linked list?

Ans. Traversal: Involves visiting each node in the linked list, usually to perform an operation (eg, printing values) on each node. The entire list is processed from the head to the end (or vice versa in doubly linked lists) regardless of the data.

searching: Specifically involves looking for a node with certain value. The search stops once the desired node is found or the list ends, unlike traversal which goes through the entire list.

Qs d) Explain the concept of overflow & underflow in linked list.

Ans. Overflow: occurs when there is no available memory to allocate a new node in the linked list. This happens in dynamic memory allocation environment if the system runs out of heap memory.

underflow: Happens when trying to perform an operation on an empty linked list. Since there is no element to remove or process, this leads to an error condition.

Name - Sagar Bhatiya
sec - C
CRN: 93

Qs e) Describe the Doubly Linked list & its advantage over a single linked list.

Ans. A Doubly Linked List is a type of linked list where each node has two pointers:

1. Next pointer : Points to the next node in the list.
2. Previous pointer: Points to the previous pointer in the list.

Advantages over a Singly Linked List:

- Bidirectional traversal : you can traverse the list in both forward & backward direction.
- easier deletion : Nodes can be deleted more efficiently because you have direct access to previous node.
- More flexible insertion : Inserting a node before or after a given node is easier since both next & previous nodes are accessible.

Sec : C Sagar Bhatiya

Name : Sagar Bhatiya
Sec : C
CRN: 93

Qs 5) Explain the concept of Polynomial representation using linked lists.

Ans. Polynomial representation using linked lists is a way to store and manipulate polynomials efficiently. A polynomial is typically expressed as:

$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$

Here $a_n, a_{n-1}, \cdots, a_1, a_0$ are the coefficients and $n, n-1, \cdots, 1, 0$ are the exponents.

1) Nodes Structure : Each term of the polynomial is stored in a node of a linked list. A node typically has three fields:

- Coefficient: The coefficient of the term
- Exponent: The exponent corresponding to the variable.
- Pointer : A link to next node.

2) Linked Representation : Instead of storing terms in an array (which may waste memory or require resizing), the linked list dynamically allocates memory for each term. Terms are stored in order of their exponents (either increasing or decreasing).

Name: Sagar Bhatiya
Sec : C
CRN: 93

Qs 8) Implement a Dequeue to check if a string is a palindrome by adding & removing characters from both ends.

Ans.

```
typedef struct Deque{
        char arr [MAX];
        int front;
        int rear;
    } Dq;

void initDeque ( Dq * dq){
    dq -> front = -1;
    dq -> rear = -1;
  }

bool isEmpty ( Dq * dq){
    return dq -> front == -1;
    }

bool isFull (Dq * dq) {
    return (dq -> front == 0 && dq -> rear == MAX -1)
        || (dq -> front == dq -> rear + 1);
    }

void addRear ( Dq * dq , char item) {
  if( isFull( dq)) {
      printf(" Deque is Full");
      return;
    }
  if (dq -> front == -1){
    dq -> front = 0;
    dq -> rear = 0;
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

```
}
else if( dq->rear == MAX-1){
        dq->front=0;
        dq->rear=0;
    }
 else{
    dq->rear++;
    }
  dq->arr[dq->rear]=item;
}

char removeFront(Dq *dq){
     if(isEmpty(dq)){
          printf(" Empty");
          return '\0';
      }
     char item = dq->arr[dq->front];
     if(dq->front == dq->rear){
        dq->front = -1;
        dq->rear = -1;
      }
    else if(dq->front==MAX-1){
          dq->front=0;
       }
     else{
        dq->front++;
       }
```

Name: Sagar Bhatiya
CRN: 93
Sec: C

Qs h) Compare & contrast Singly Linked Lists, Doubly Linked Lists and circular Linked lists.

Ans. 1) Structure

- Singly linked list
  -) Each node contains data & a pointer to the next node.
  -) Nodes are linked in a singee direction (from head to tail).
- Doubly Linked list
  -) Each node contains data, a pointer to next node, and a pointer to the previous node.
  -) Nodes can be traversed in both directions (forward & backward).
- Circular Linked lists:
  -) can be either singly or doubly linked.
  -) The last node points back to the first node, forming a circle.
  -) Traversal can continue indefinitely without reaching a null reference.

2) Memory Usage

- Singly LL
  -) Requires less memory per node (one pointer).
-) Doubly LL
  Requires more memory per node (two pointer).
-) Circular LL
  ... other it is single or doubly.

Name: Sagar Bhatiya
Sec: C
CRN: 93

```
char removeRear(Dq *dq){
    if (isEmpty(dq)) {
        printf("Empty");
        return '\0';
    }
char item = dq->arr[dq->rear];
if (dq->front == dq->rear){
    dq->front = -1;
    dq->rear = -1;
}
else if (dq->rear == 0) dq->rear = MAX-1;
else dq->rear--;
return item;
}

bool isPalindrome (char *str){
    Dq dq;
    initReque(&dq);
    int len = strlen(str);
    for(int i=0; i<len; i++){
        addRear(&dq, str[i]);
    }
while(dq.front != dq.rear && dq.front != (dq.rear+1)%
                                        MAX){
    char frontChar = removeFront(&dq);
    char rearChar = removeRear(&dq);
    if(front != rearChar){
        return false;
    }
}
return true;
```

Name: Sagar Bhatiya
CRN: 93
Sec: C

Qs 4) Compare & contrast Singly Linked lists, Doubly Linked lists and circular linked lists.

Ans. 1) Structure

- Singly linked list
  -) Each node contains data & a pointer to the next node.
  -) Nodes are linked in a singee direction (from head to tail).
- Doubly Linked list
  -) Each node contains data, a pointer to next node, and a pointer to the previous node.
  -) Nodes can be traversed in both directions (forward & backward).
- Circular Linked lists:
  -) can be either singly or doubly linked.
  -) The last node points back to the first node, forming a circle.
  -) Traversal can continue indefinitely without reaching a null reference.

2) Memory Usage

- Singly LL
  -) Requires less memory per node (one pointer).
-) Doubly LL
  Requires more memory per node (two pointer).
-) Circular LL
  Memory usage depends on whether it is single or doubly.

Name - Sagar Bhatiya

Sec: C

CRN: 93

```
int search (Node * head, int Key) {
    Node* temp = head;
    int idx = 0;
    while (temp != NULL){
        if (temp -> data == Key ) {
            return idx;
        }
        temp = temp->next;
        idx++;
    }
    return -1;
}
```

6.5.b) Implement a Two way header list and demonstrate insertion & deletion of nodes.

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

Node* createNode(int data){
    Node* nn = (Node*) malloc(sizeof(Node));
    nn -> data = data;
    nn -> prev = NULL;
    nn -> next = NULL;
    return nn;
}
```

Name : Sagar Bhatiya
Sec : C
CRN: 93

```
Node* initializeHeader(){
  Node* header = createNode(0);
   header->prev = header;
   header->next = header;
     return header;
  }
void insertEnd(Node* header, int data){
    Node* nn = createNode(data);
    Node* last = header->prev;
    last->next = nn;
    nn->prev = last;
    nn->next = header;
     header->prev = nn;
  }
void deleteNode(Node* header, int data){
   Node* temp = header->next;
   while(temp != header){
       if(temp->data == data){
          Node* prevNode = temp->prev;
          Node* nextNode = temp->next;
          prevNode->next = nextNode;
          nextNode->prev = prevNode;
          free(temp);
          return;
       }
      temp= temp->next;
   }
}
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

```
else{
    while (fast != NULL && fast->next != NULL){
        prev = slow;
        slow = slow->next;
        fast = fast->next->next;
    }
    nn->next = prev->next;
    prev->next = nn;
    return head;
  }
}
Node* insertAtEnd(Node* head, int val){
 Node*nn = createNode(val);
Node* temp= head;
if (head==NULL){
      return nn;
  }
else{
   while (temp->next != NULL){
          temp = temp->next;
       }
      nn->next = NULL;
      temp->next = nn;
  }
 return head;
 }
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

Qs c) Write algorithms to handle overflow & Underflow in linked list.

Ans Overflow

1. Allocate memory for the new node using malloc() or new (in C/C++).
2. If malloc() returns NULL, print an overflow error message & exit the operation.
3. If memory is successfully allocated:
   Proceed with inserting the node into the linked list at the desired position.

Underflow:

1) Check if list is empty:
   if the head pointer is NULL, print an "underflo
   error message & exit the operation.
2) If the list is not empty:
   Proceed to delete the node.

Qs d) Implement a Doubly Linked List

i) Traversal

ii) Insertion & deletion of nodes at various position.

Name: Sagar Bhatiya
Sec: C
CRN: 93

```
void traverseforward (Node*head){
    Node* temp = head;
    while (temp != NULL){
        printf("%d", temp->data);
        temp = temp->data;
    }
}
void traverseBackward (Node* tail){
    Node * temp = tail;
    while (temp != NULL){
        printf("%d", temp->data);
        temp = temp->prev;
    }
}
void insertAtBeggining (Node**head, int data){
    Node* nn = createNode(data);
    if (*head == NULL){
        *head = nn;
    }
    else{
        nn->next = *head;
        (*head)->prev = nn;
        *head = nn;
    }
}
void deleteFirstNode (Node**head){
    if(*head == NULL) return;
    Node* temp = *head;
    *head = (*head)->next;
    if(*head != NULL){
        (*head)->prev = NULL;
    }
    printf("%d", temp->data);
    printf(
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

25 e) Write a C program to represent & add two Polynomials using linked list.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int coeff;
    int power;
    struct Node *next;
} Node;
void insert (Node **poly, int coeff, int power){
Node *temp -> coeff = coeff;
 temp ->power = power;
 temp-> next =NULL;
 if (*poly ==NULL){
      *poly = temp;
       return;
    }
 Node*current =*poly ;
while (current ->next! =NULL){
     current = current -> next;
  }
  current -> next = temp;
}
Node * add (Node*poly1, Node*poly2){
   Node* result =NULL;
while (poly1!= NULL && poly2 != NULL ){
  if(poly1->power ==poly2->power) {
    insert(&result, poly1->coeff+ poly2->coeff, poly1->power);
    poly1 = poly1 -> next;
    poly2 = poly2->next;
}
```

Name → Sagar Bhatiya
Sec : C
CRN: 93

```
else if(poly1->power > poly2->power){
        insert(&result, poly1->coeff, poly1->power);
        poly1 = poly1->next;
    }
  else{
    insert(&result, poly2->coeff, poly2->power);
    poly2 = poly2->next;
  }
}
while(poly1 != NULL){
        insert(&result, poly1->coeff, poly1->exp);
        poly1 = poly1->next;
    }
  while(poly2 != NULL){
      insert(&result, poly2->coeff, poly2->exp);
      poly2 = poly2->next;
    }
    return result;
}
return result;
}

int main(){
 Node* poly1 = NULL;
 insert(&poly1, 5, 4);
 insert(&poly1, 3, 2);
 Node* poly2 = NULL;
 insert(&poly2, 4, 4);
 insert(&poly2, 2, 2);
 Node* result = add(poly1, poly2);
 printf("Result:");
 printf(result);
```

Qs 1 Write a C function to flatten a linked list where every node in the linked list contains two pointers - Next pointer to the next node in the list & child pointer to a linked list where the current node is the head.

```
typedef struct Node{
    int data;
    struct Node* next;
    struct Node* child;
} Node;
Node* createNode(int data){
    Node* nn = (Node*)malloc(sizeof(Node));
    nn->data = data;
    nn->next = NULL;
    nn->child = NULL;
    return nn;
}
void flattenList(Node* head){
 if(!head) return;
 Node* current = head;
 while(current){
    if(current->child){
      Node* childTail = current->child;
      while(childTail->next){
            childTail = childTail->next;
          }
      childTail->next = current->next;
      current->next = current->child;
      current->child = NULL;
    }
    current = current->next;
  }
}
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

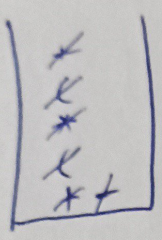
2) Practicle Excercises:

Ans a) Implement a Singly Linked list.

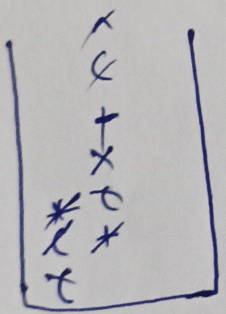
```
typedef struct node {
        int data;
        struct node* next;
        }Node;
Node* createNode ( int val){
    Node* nn = (Node*) malloc (sizeof (Node));
    nn->data = val;
    nn->next = NULL;
     return nn;
      }
Node* insertBeg (Node* head ,int val){
   Node* nn = createNode (val);
    if (head == NULL){
          return nn;
          }
    else{
       nn->next = head;
        head = nn;
       }
     return head;
NODE* insertAtMid(Node* head, int val){
   Node* slow = head;
   Node* fast = head;
    Node* prev = NULL;
     Node* nn = createNode (val);
      if (head == NULL){
            return nn;
            }
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

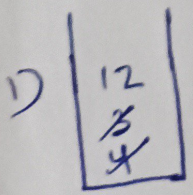
Qs g) Convert the following expression to postfix -
A+(B*(C+D))-E

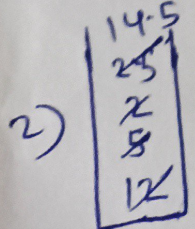
Ans.  ⇒ A B C D + * + E -

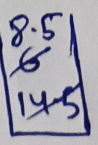
Qs h) Convert the following expression to postfix ((4*3)+(5/2)-6)+(8^2) and then evaluate using stack.

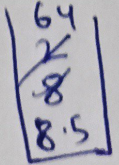
Ans.  ⇒ 4 3 * 5 2 / + 6 - 8 2 ^ +

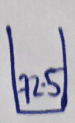
Evaluation:

1)  4*3 = 12

2)  5/2 = 2.5
12+2.5=14.5

3)  14.5 - 6 = 8.5

4)  8^2 = 64

5)  = 64+8.5 = 72.5

Name: Sagar Bhatiya

Sec: C

CRN: 93

Practical Exercises:

Q2a) Implement a queue using an array. Perform the following operations:

i) Create a Queue.
ii) Enque (iii) Dequeue (iv) Isfull or Empty.

```
# include <stdio.h>
# define SIZE 50
int queue[SIZE], front=-1, Rear=-1;

void enqueue(int item){
    if(Rear==SIZE-1){
        printf("FULL");
    }
    else{
        if(front==-1){
            front=0;
        }
        rear=rear+1;
        queue[rear]=item;
    }
}

void dequeue(){
    if(front==-1){
        printf("Empty");
    }
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

```
    else{
        printf("%d", queue[front]);
        front = front+1;

        if (front > Rear) {
            front = -1;
            Rear = -1;
        }
    }
}

void printQueue(){
    if (Rear == -1){
        printf("Empty");
    else{
        for(int i = front; i <= Rear; i++){
            printf("%d", queue[i]);
    }
}

int main(){
    enqueue(2);
    enqueue(4);
    enqueue(6);
    enqueue(8);

    dequeue();
    dequeue();
    printQueue();

    return 0;
```

Name: Sagar Bhatiya
Sec: C
ERN: 93

Qs b) Implement a Linked Queue using a linked list. Ensure that you implement the same operations (Create, Add, Delete, Full & Empty).

```
#include <stdio.h>
#include <stdlib.h>

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

typedef struct LinkedQueue{
        Node* front;
        Node* Rear;
    } LQ;

LQ* createQueue(){
    LQ* queue = (LQ*) malloc(sizeof(LQ));
    queue->front = queue->Rear = NULL;
    return queue;
    }

int isEmpty(LQ* queue) {
    return (queue->front == NULL);
    }
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

```
int isFull(){
  Node *temp = (Node*)malloc(sizeof(Node));
    if(temp==NULL){
      return 1;
    }
   free(temp);
   return 0;
  }
void enqueue(LQ* queue, int val){
  Node* temp = (Node*)malloc(sizeof(Node));
  if(temp==NULL){
      printf("FULL");
      return;
    }
 temp->data = val;
 temp->next = NULL;
 if(queue->Rear==NULL){
    queue->front = queue->Rear = temp;
    return;
    }
 queue->rear->next=temp;
 queue->rear = temp;
}
```

Name: Sagar Bhatiya
CRN: 93
Sec: C

```
void dequeue( LQ* queue){
    if(isEmpty(queue)) {
        printf("Empty");
        return;
    }
Node *temp = queue->front;
queue->foont = queue->front->next;
if(queue->front==NULL){
    queue->Rear=NULL;
}
free(temp);
}
void display(LQ* queue){
    if(isEmpty(queue)){
        printf(" Empty");
    }
    else{
        printf("%d", queue->front->data);
    }
}
int main(){
LQ* queue=createQueue();
enqueue(queue,10);
enqueue(queue,20);
enqueue(queue,30);
display(queue);
dequeue(queue);
display(queue);
return 0;
}
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

Q5C. Write a C Program to implement a circular Queue using an array or linked list

```
# include < stdio.h>
# define max 20
int queue [max];
int front = -1;
int rear = -1;

void enqueue ( int val){
 if (front == -1 && rear == -1){
      front = 0;
      rear = 0;
      queue [rear] = val;
  }
 else if ( (rear + 1) % max == front ) {
       printf ("Overflow");
     }
 else {
    rear = ( rear + 1) % max ;
    queue [rear] = val;
  }
}

int dequeue (){
   if ((front == -1) && ( rear == -1)) {
        printf ( " Underflow");
    }
```

Name : Sagar Bratiya
Sec: C
CRN: 93

Qs d) Implement a Dequeue (Double Ended Queue), where you can insert and delete elements from both ends.

Ans.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5

typedef struct Dequeue {
        int items[SIZE];
        int front;
        int rear;
    }dq;

dq* createDequeue(){
    dq*nn = (dq*)malloc(sizeof(dq));
    nn->front = -1;
    nn->rear = -1;
     return nn;
  }

int isFull(dq*nn){
if((nn->front==0 && nn->rear == SIZE-1)||(nn->front
    == nn->rear + 1)){

        return 1;
      }
    return 0;
  }
```

Name → Sagar Bhatiya
CRN : 93
Sec : C

```
int isEmpty(dq *nn){
    return(nn->front == -1);
}

void insertFront(dq *nn, int val){
    if(isFull(nn)){
        printf("FULL");
        return;
    }
    if(nn->front == -1){
        nn->front = nn->rear = 0;
    }
    else if(nn->front == 0){
        nn->front = SIZE-1;
    }
    else{
        nn->front--;
    }
    nn->items[nn->front] = val;
    printf("Inserted %d at front\n", val);
}

void insertRear(dq *nn, int val){
    if(isFull(nn)){
        printf("FULL");
        return;
    }
```

Name → Sagar Bhatiya
Sec: C
CRN: 93

```
if (qq->rear == -1){
    qq->front = qq->rear = 0;
}
else if (qq->rear == SIZE-1){
    qq->rear = 0;
}
else{
    qq->rear++;
}
qq->items[qq->rear] = val;
}

void deleteFront(dq *qq){
    if(isEmpty(qq)){
        printf("Empty");
        return;
    }
    printf("%d", qq->items[qq->front]);
    if(qq->front == qq->rear){
        qq->front = qq->rear = -1;
    }
    else if(qq->front == SIZE-1){
        qq->front = 0;
    }
    else{
        qq->front++;
    }
```

Name → Sagar Bhatiya
Sec : C
CRN: 93

```
void deleteRear(dq *nn){
    if(isEmpty(dq)){
        printf("Empty");
        return;
    }
    printf("%d", nn->items[nn->rear]);
    if(nn->front == nn->rear){
        nn->front = nn->rear = -1;
    }
    else if(nn->rear == 0){
        nn->rear = SIZE-1;
    }
    else{
        nn->rear--;
    }
}
void display(dq *nn){
    if(isEmpty(nn)){
        printf("Empty");
        return;
    }
    if(nn->rear >= nn->front){
        for(int i = nn->front; i<=nn->rear; i++){
            printf("%d", dq->items[i]);
        }
    }
}
```

Name → Sagar Bhatiya
Sec : C
CRN : 93

Q5c) Create a Priority Queue and demonstrate how elements are inserted and deleted based on their priority.

```
#include<stdio.h>
#include<stdlib.h>
#define SIZE 5
typedef struct Element{
        int data;
        int priority;
    }elem;
typedef struct PriorityQueue{
        elem items[SIZE];
        int front;
        int rear;
    }PQ;
PQ* createQueue(){
    PQ* pq = (PQ*)malloc(sizeof(PQ));
    pq->front = -1;
    pq->rear = -1;
    return pq;
}
```

Name : Sagar Bhatiya
Sec : C
CRN: 93

```
int isFull(PQ *pq){
    return(pq->rear == SIZE-1);
}
int isEmpty(PQ *pq){
    return(pq->front == -1);
}
void insert(PQ *pq, int data, int priority){
    if(isFull(pq)){
        printf("FULL");
        return 0;
    }
}
    elem newElement;
    newElement.data = data;
    newElement.priority = priority;
    if(isEmpty(pq)){
      pq->front = pq->rear = 0;
      pq->items[pq->rear] = newElement;
    }
  else{
  for(int i = pq->rear; i >= pq->front; i--){
        if(priority < pq->item[i].priority){
            pq->items[i+1] = pq->items[i];
        }
```

Name : Sagar Bhatiya
Sec : C
CRN : 93

```
    else{
        break;
    }
}
pq->items[i+1] = newElement;
pq->rear++;
}
}
void delete(PQ *pq){
    if(isEmpty(pq)){
        printf("Empty");
        return;
    }
printf("%d", pq->items[pq->front].data);
if(pq->front == pq->rear){
    pq->front = pq->rear = -1;
}
else{
    pq->front++;
}
}
void display(PQ *pq){
    if(isEmpty(pq)){
        printf("Empty");
        return;
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

```
for(int i = pq->front; i<= pq->rear; i++){
    printf("%d", pq->items[i].data);
  }
}

int main(){
   PQ *pq = createQueue();
   insert(pq,10,2);
   insert(pq,20,1);
   insert(pq,30,3);
   insert(pq,40,0);
   insert(pq,50,5);
   display(pq);
   delete(pq);
   delete(pq);
   display(pq);
   return 0;
}
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

Qs 8) Write a C function to check whether an expression has balanced parenthesis.

```
# include <stdio.h>
# include <stdlib.h>
# define MAX 100

typedef struct Stack{
        int top;
        char items[MAX];
} Stack;

Stack * createStack(){
   Stack * stack = (Stack *)malloc(sizeof(Stack));

   stack -> top = -1;
    return stack;
   }
int isEmpty(Stack * stack){
      return stack -> top == -1;
    }
void push(Stack * stack, char c){
      if(stack -> top == MAX-1){
         printf("Overflow");
          return;
       }
     stack -> items[++stack -> top] = c;
}
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

```
char pop(Stack* stack){
    if(isEmpty(stack)){
        printf("Underflow");
        return '\0';
    }
    return stack->items[stack->top--];
}
int isMatchingPair(char open, char close){
    return (open == '(' && close == ')') ||
           (open == '{' && close == '}') ||
           (open == '[' && close == ']');
}
int areParenthesesBalanced(char* exp){
    Stack* stack = createStack();
    for(int i = 0; i < strlen(exp); i++){
        char currentChar = exp[i];
        if(currentChar == '(' || currentChar == '{' ||
           currentChar == '['){
            push(stack, currentChar);
        }
        else if(currentChar == ')' || currentChar == '}'
                || currentChar == ']'){
            if(isEmpty(stack) || !isMatchingPair(pop(stack),
               currentChar)){
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

```
    }
  }
  return isEmpty(stack);
}

int main(){
    char exp[MAX];
    printf("Enter an expression: ");
    fgets(exp, MAX, stdin);
    exp[strcspn(exp, "\n")] = '\0';
    if(areParenthesesBalanced(exp)) {
            printf("Balanced Parentheses");
    }
    else{
        printf("Not Balanced Parentheses");
    }
    return 0;
}
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

2s g) Write a c program to find Next greater element using stack in an array.

```
#include <stdio.h>
#include< stdlib.h>
# define MAX_SIZE 100
int stack [MAX_SIZE];
int top = -1;
void push (int data) {
    if (top = = MAX_SIZE -1){
        printf ("Overflow");
        return;
    }
    top++;
    stack [top] = data ;
}
int pop() {
    if (top = = -1){
        printf ("Empty ");
        return -1;
    }
    int data = stack [top];
    top--;
    return data;
}
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

```
void print_nextgreater_element (int arr[], int n){
    int i, next, element;
    push(arr[0]);
    for(i=1; i<n; i++){
        next = arr[i];
        if (top != -1){
            element = pop();
            while (element < next){
                printf("%d --> %d", element, next);
                if (top == -1){
                    break;
                }
                element = pop();
            }
            if (element > next){
                push (element);
            }
        }
        push (next);
    }
while (top != -1){
        element = pop();
        next = -1;
        printf("%d --> %d", element, next);
    }
}
```

Name: Sagar Bhatiya
Sec: C
CRN: 93

```
int main(){
  int n = 6;
  int i;
  int arr[n] = {1,2,3,4,5,6};
 printf(" Array elements \n");
 for(i=0; i<6; i++){
     printf("%d", arr[i]);
   }
  printf(" The next larger elements are : \n");
  print_next_greater_element(arr,n);
  return 0;
}
```