INDEX

NAME: TK STD.: 3 SEC.: F ROLL NO.: CS310 SUB.: Data structures

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1	1/01/2024	Stack implementation using Arrays		
2.	01/01/2024	Infix to Postfix Expression conversion		
3.	08/01/2024	Linear Queue using Arrays		
4.	8/01/2024	Circular Queue using stacks		
5.	22/01/2024	Singly Linked List		
6.	29/01/2024	Stack implementation using singly Linked List		
7.	29/01/2024	Queue implementation using singly linked List		
8.	29/01/2024	sorting reverse and concateration of SLL		
9.	5/02/2024	Doubly linked List Implementation		
10.	05/02/2024	Score of parentheses		
11.	19/02/2024	Binary Search Tree		
12.	19/02/2024	Delete the middle node of a linked list		

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
13.	19/2/24	Odd Even Linked List		
14.	26/2/24	Delete Node in BST		
15.	26/2/24	Find bottom left tree value		
16.	26/2/24	Breadth First search Implementation		
17.	26/2/24	Depth First search Implementation		
18.	26/2/24	Hashing Techniques		

Week - 1

1) Program 1:

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

# define SIZE 10

void push (int);
void pop ();
void display ();

int stack [SIZE], top = -1;

void main ()
{
  int value, choice;
  clrscr ();
  while (1) {
  Printf ("\n\n ** Menu ** \n");
  Printf ("1. Push \n 2. Pop \n 3. Display \n 4. Exit");
  Printf ("\n Enter your choice: ");
  Scanf ("%d, & choice);
  switch (choice) {
   Case 1: Printf (" Enter the value to be insert: ");
           Scanf ("%d", & value);
           Push (value);
           break;
   Case 2: Pop ();
           break;
   Case 3: display ();
           break;
   Case 4: exit (0);
```

```
default: Printf("\n Wrong selection!! Try again!");
    }
  }
}

void push (int value){
  if (top == SIZE - 1)
    printf("\n stack is Full!! Insertion is
              not Possible !!!");
  else {
    top ++;
    stack [top] = value;
    Printf("\n Insertion success !!");
  }
}

Void POP(){
  if (top == -1)
    Printf("\n Stack is Empty!! Deletion is not possible");
  else {
    Printf("\n Deleted : %d", stack [top]);
  }
}

Void display (){
  if (top == -1)
    Printf("\n Stack is Empty!");
  else {
    int i;
    Printf("\n Stack elements are: \n");
    for (i=top; i>=0; i--)
      Printf("%d \n", stack[i]);
  }
}
```

Output:

→ Enter 1: push , 2: POP , 3: display , 4: Exit
2
Stack is Empty overflow condition
Enter 1: push , 2: POP , 3: display, 4: Exit
1
Enter a value : 10
Insertion successful
Enter 1: push , 2. pop , 3. display , 4. Exit
1
Enter a value : 10
Insertion successful
Enter 1: push, 2. pop , 3. display , 4. Exit
1
Enter a value 20
Insertion successful
Enter 1: push, 2. pop , 3. display , 4. Exit
2
20 removed successfully
Enter 1: push, 2: pop, 3. display 4. Exit
3

The elements of stack are 10
Enter 1: push, 2. pop , 3. display 4. Exit
4

Program 2: (Infix to postfix conversion).

```
# include <stdio.h>
# include <ctype.h>
# define SIZE 50

char stack [SIZE];
int top = -1;

void Push (char element)
{
    Stack [++top] = element;
}
char Pop()
{
    return (stack [top--]);
}
int Pr (char Symbol)
{
    if (symbol == '^')
    {
        return (3);
    }
    else if (symbol == '*' || symbol == '/')
    {
        return (2);
    }
    else if (symbol == '+' || symbol == '-')
    {
        return (1);
    }
    else {
        return (0);
    }
}
```

```
~~void~~ main()
{
    char infix[50], Postfix[50], ch, element;
    int i=0, K=0;
    Printf("Enter Infix Expression :");
    Scanf("%s", infix);
    Push('#');
    while ((ch = infix[i++]) != '\0')
    {
        if(ch == '(') push(ch);
        else
            if (isalnum(ch)) postfix[K++] = ch;
            else
                if (ch == ')')
                {
                    while (stack[top] != '(')
                        Postfix[K++] = pop(); top--;
                    element = pop();
                }
            else
                {   while (Pr(Stack[top]) >= Pr(ch))
                        Postfix[K++] = pop();
                    Push(ch);
                }
    }
    while (stack[top] != '#')
        Postfix[K++ = POP();
    Postfix[K] = '\0';
    Printf("\n Postfix Expression = %s\n", Postfix);
    return 0;
}
```

Output

Enter the infix expression : A*B+C*D-E.
Infix Expression : A*B+C*D-E.
Post fix Expression : AB*CD*+E-.

// Linear Queue using Arrays

```
# include <stdio.h>
# include <stdlib.h>
# define SIZE 3

int front = -1;
int rear = -1;
int queue [SIZE];

void insert ();
void deleteq ();
void display ();

int main ()
{ int choice;
  while (1)
  {
    Printf ("1. Insert \n 2. Delete \n 3. Display \n");
    Printf (" Enter a choice for user operation : ");
    Scanf ("%d", &choice);
    Switch (choice)
    {
     Case 1:
        Insert ();
        break ;
     Case 2:
        deleteq ();
        break;
     Case 3:
        display ();
        break;
     Case 4:
        exit (1);
```

```
    default :
        Printf (" invalid input !\n");
    }
  }
}

void insert ()
{
  int add;
  if (rear == SIZE-1)
  {
   Printf (" The Queue is in Overflow condition !\n");
  }
  else
  {
    if (front == -1)
    {
      front = 0;
    }
    Printf (" Enter a value to input : ");
    Scanf (" %d", & add);
    rear += 1;
    queue [rear] = add;
  }
}

void deleteq ()
{
  if ( front == rear || front > rear)
  {
    Printf (" The Queue is in underflow condition \n");
    return;
  }
  else
```

```
{
  Printf(" The deleted item will be %d ",
        queue [front]);
  front = front + 1;
 }
}

Void display () {
 int i;
 if (front == -1)
 {
  Printf(" The Queue is Empty \n"); }
 else
 {
  Printf(" The Queue is : \n");
  for( i = front; i <= rear; i++)
   { printf ("%d", queue [i]);
     Printf ("\n");
   }
 }
}
```

=> output:-

Enter your choice
1. Insert
2. Delete
3. Display
4. Exit

Enter your choice : Enter the number to be inserted

5

7

5

=>

=> Overflow condition

=> 2, the deleted element is 5.

=> 1, 5, Overflow condition

Circular Queue using stacks

```
# include <stdio.h>
# define SIZE 3
int items [SIZE];
int front = -1, rear = -1;
int isFull(){
if ((front == rear + 1) || (front == 0 && rear == SIZE-1))
return -1;
return 0;
}

int isEmpty()
{
    if (front == -1) return 1; return 0;
}
void enQueue (int element)
    { if (isFull())
      Print ("\n Queue is full!! \n");
      else {
      if (front == -1) front = 0;
        rear = (rear + 1) % SIZE;
        items [rear] = element;
        Printf ("\n Inserted -> %d", element);
    }
}
int deQueue () {
  int element;
  if (isEmpty()) {
    Printf("\n Queue is empty!! \n");
    return(-1);
    }
    else
    {
```

```
element = items [front];
if (front == rear) {
        front = -1;
        rear = -1;
        }
    else
        {
        front = (front+1) % SIZE; }
        Printf ("\n Deleted element -> %d \n", element);
        }
    }

void display ()
        {
        int i;
        if (is Empty ())
        Printf ("\n Empty Queue \n");
    else {
    Printf ("\n Front -> %d ", front);
    Printf ("\n Items -> ");
    for (i = front; i != rear; i = (i+1)%
    Printf ("%d", items [i]);                SIZE)
    }
    Printf ("%d", items [i]);
    Printf ("\n Rear > %d \n", rear);
    }

void main ()
{ int choice, element;
  while (1) {
  Printf ("\n 1. Insert \n 2. delete \n 3. display \n
                4. exit ");
  Printf ("%d", & choice);
```

```
switch (choice) {
    case 1: Printf ("\n Enter the element to insert");
            scanf ("%d", element);
            enqueue (element);
            break;
    case 2: element = dequeue ();
            if (element != -1)
            Printf ("%d is deleted item", element);
            break;
    case 3: display ();
            break
    case 4: exit(0);
    default: Printf ("Invalid choice");
}
}
```

⇒ Output:

Enter the operation:
1. enqueue
2. dequeue
3. display
4. exit

Enter the ~~Number~~ operation	Enter the operation
1	5
1	7
1	6

Enter the operation: 1, 4
⇒ Queue Overflow

Singly Linked List

Insertion and Deletion of a node at first position, at end of list.

```
# include <stdio.h>
# include <stdlib.h>
# include <conio.h>
struct node {
        int data;
        struct node* next; };

struct node * start = Null;
struct node * insert_beg (struct node*);
struct node * insert_end (struct node*);
struct node * delete_beg (struct node*);
struct node * delete_end (struct node*);
struct node * delete_at_pos (struct node*);
struct node * insert_at_pos (struct node*);
struct node * display (struct node*);

int main ()
{
    int option;
    do
    {
        Printf ("\n *** Menu ***");
        Printf ("\n 1: Add a node at the beginning");
        Printf ("\n 2: Add a node at the end");
        Printf ("\n 3: Add a node at a specific position")
        Printf ("\n 4: Delete a node from the beginning");
        Printf ("\n 5: Delete a node from the end");
        Printf ("\n 6: Delete a node from a specific position");
        Printf ("\n 7: Display the list");
```

```
printf ("\n 8: Exit");
Printf ("\n Enter your option :");
Scanf ("%d", &option);
Switch (option)
{
 Case 1:
      start = insert_beg (start);
      break;
 Case 2:
      start = insert_end (start);
      break;
 Case 3:
      Start = insert_at_Pos (start);
      break;
 Case 4:
      start = delete_beg (start);
      break;
 Case 5:
      start = delete_end (start);
      break;
 Case 6:
      start = delete_at_Pos (start);
      break;
 Case 7:
      start = display (start);
      break;
 }
} while (option != 8);

struct node * temp;
While (start != Null)
{
     temp = start;
     start = start -> next;
```

```
    free (temp);
  }
  return 0;
}

struct node * insert_beg (struct node * start)
{
    struct node * new_node;
    int num;
    Printf ("Enter the data :");
    scanf ("%d", &num);
    new_node = (struct node *) malloc (size of (
                struct node));
    new_node -> data = num;
    new_node -> next = start;
    start = new_node;
    Printf ("Inserted at the beginning. \n");
    return start;

struct node * insert_end (struct node * start)
{
    struct node * ptr, * new_node;
    int num;
    Printf ("Enter the data : ");
    scanf ("%d", &num);
    new_node = (struct node *) malloc (size of (struct node));
    new_node => data = num;
    new_node => Next = NULL;

    if (start == NULL)
    {
        start = new_node;
    }
```

```
else
{
  Ptr = start;
  while (Ptr -> next != NULL)
  {
     Ptr = Ptr -> next;
  }
  Ptr -> next = new_node;
}
Printf (" Inserted at the end \n");
return start;

Struct node * insert_at_Pos (struct node *start)
{
  struct node *, new_node, * Ptr, * Preptr;
  int Pos, num;
  Printf ("Enter the position to insert at: ");
  Scanf ("%d", &Pos);
  Printf ("Enter the data:");
  Scanf ("%d", &num);
  new_node = (struct node *) malloc (size of (
    struct node));
  new_node -> data = num;
  new_node -> next = NULL;
  if (Pos == 1)
  {
     new_node -> next = start;
     start = new_node;
     Printf (" Inserted at position %d \n", Pos);
     return start;
  }
  else
  {
```

```
    ptr = start;
    for (int i = 1; i < pos && ptr != NULL; i++)
    {
        Preptr = ptr;
        ptr = ptr -> next;
    }
    if (ptr == NULL && pos > i)
    {
        Printf(" Invalid position. Node can't be
                inserted. \n");
        return start;
    }
    Preptr -> next = new_node;
    new_node -> next = ptr;
    Printf(" Inserted at position %d. \n", Pos);
    return start;
  }
}

Struct node * delete_beg (struct node *start)
{
    struct node * ptr;
    ptr = start;
    if (ptr == NULL)
    {
        printf(" Empty list. Can't be deleted. \n");
        return start;
    }
    else
    {
        start = start -> next;
        free (ptr);
        printf(" Deleted at the beginning. \n");
        return start;
    }
}
```

```
struct node * delete_end (struct node * start)
{
    struct node * ptr, * ptr1;
    ptr = start;
    if (ptr == NULL)
    {
        printf (" Empty list. Can't be deleted. \n");
        return start;
    }
    else if (Ptr -> next == NULL)
    {
        free (ptr);
        start = NULL;
        printf (" Deleted at the end. \n");
        return start;
    }
    else
    {
        while (ptr -> next != NULL)
        {
            Ptr1 = Ptr;
            Ptr = Ptr -> next;
        }
        Ptr1 -> next = NULL;
        free (ptr);
        Printf (" Deleted at the end. \n");
        return start;
    }
}

Struct node * delete_at_pos (struct node * start)
{
    struct node * ptr, * preptr;
    int pos;
```

```
printf ("Enter the position to delete : ");
scanf (" %d", & pos);
if (start == NULL)
{
    printf (" Empty list . Can't be deleted. \n");
    return start;
}
Ptr = start;
if (Pos == 1)
{
    start = start -> next;
    free (ptr);
    printf ("Deleted at position %d. \n", Pos);
    return start;
}
else
{
    for (int i = 1; i < pos && ptr != NULL; i++)
    {
        preptr = ptr;
        ptr = ptr -> next;
    }
    if (Ptr == NULL)
    {
        printf (" Invalid position. Node can't be deleted. \n");
        return start;
    }
    preptr -> next = ptr -> next;
    free (ptr);
    printf (" Deleted at position %d. \n", pos);
    return start;
}
}
```

```
Struct node * display ( struct node * start)
{
    struct node * ptr;
    ptr = start;
    if (ptr == NULL)
    {
        printf (" Empty list . \n");
        return start;
    }
    else
    {
        printf (" Linked list elements : ");
        while (ptr != NULL)
        {
            printf ("%d \t" ptr -> data);
            ptr = ptr -> next;
        }
        printf ("\n");
        return start;
    }
}
```

Week - 05

Stack implementation using single linked list

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int date;
    struct Node* next;
};
Struct Node * top = NULL;
void push (int data){
    struct Node * New node = (struct Node*)
        malloc (size of (struct node));
    if (new node == NULL){
        printf ("Overflow \n");
        return;
    new code -> data = date;
    new code -> next = top;
    top = newcode;
}

int pop() {
    if (top == NULL){
        printf ("underflow \n");
        return -1;
    }
    struct Node* temp = top;
    int temp1 = temp -> data;
    top = temp -> next;
    free (temp);
    return temp1;
}
```

```
void display () {
  if (top == NULL) {
      printf (" Empty \n");
      return;
  }
Struct Node * current = top;
printf (" stack : ");
while (current != NULL) {
    printf (" %d", current -> data);
    current = current -> next;
  }
  printf ("\n");
}

int main () {
  int choice ;data ;
  while (1) {
    printf (" stack operation Menu : \n");
    printf (" 1. Push \n 2. Pop \n 3. Display \n 4. exit ");
    printf (" Enter your choice : ");
    scanf (" %d", & choice);
    switch (choice) {
      Case 1 : printf (" Enter data to push : ");
               scanf (" %d", & data);
               push (data);
               break;
      Case2 : if (top == NULL) {
                printf (" Underflow \n"); }
              else {
                printf (" popped element : %d \n", Pop());
              }
              break;
      case 3 : display ();
               break;
```

```
        Case 4: Printf ("Existing ...\n");
                exit (0);
        default : Printf ("Invalid choice, \n");
    }
}
return 0;
}
```

Output :-

stack Operations:
1. Push 2. POP 3. Display 4. Exit

=> Enter your choice : 1
Enter the element to push : 2
2 pushed to stack

=> Enter your choice : 1
Enter the element to push : 4
4 pushed to stack

=> Enter your choice : 2
popped element : 4

=> Enter your choice : 3
stack elements : 2

Queue implementation using single linked list

```
# include <stdio.h>
# include <stdlib.h>
Struct Node {
    int data;
    struct Node * next;
};
Struct Queue {
    struct Node* front;
    struct Node* rear;
};
Void initialise (struct Queue* queue) {
    queue -> front = NULL;
    queue -> rear = NULL;
}
int isEmpty (struct Queue* queue) {
    return (queue -> front == NULL);
}
Void enqueue (struct Queue* queue, int data) {
  struct Node* new node = (struct Node*) Malloc
  (size of (struct node));
  if (new node == NULL) {
    print ("Overflow \n");
    return;
  }
  new node -> data = data;
  newnode -> next = NULL;
if (is Empty (queue)) {
  queue -> front = newnode;
  queue -> rear = newnode;
}
  printf ("Enqueued! \n");
}
```

```
int dequeue (struct Queue* queue){
  if c is empty (queue)){
    printf ("underflow \n");
     return -1;
  }

Struct Node* temp = queue -> front;
int date = temp -> data;
if (queue -> front == queue -> rear){
    queue -> front = NULL;
    queue -> rear = NULL;
  }
else {
    queue -> front = queue -> front -> next;
  }
free (temp);
printf (" Dequeued %d \n", data);
  return Data;
}
void display (struct Queue* queue){
  if ( is Empty (queue)) {
    printf (" Empty \n");
  return ;
}
struct Node* current = queue -> front;
printf ("Queue : ");
  while (current != NULL){
    printf ("%d", current -> date);
    current = current -> next;
}
Printf ("\n");
}
```

```
int main ()
{
  struct Queue q;
  initialise (&q);
  int choice, data;
  while (1) {
  Printf (" Queue operation Menu: \n");
  printf (" 1. Enqueue \t 2. Dequeue \t
          3. Display \t  4. Exit \n");
  Printf (" Enter your choice:");
  scanf ("%d", &choice);
  switch (choice) {
  Case 1: printf (" Enter data to enqueue: ");
          scanf ("%d", &data);
          enqueue (&q, data);
          break;
  Case 2: dequeue (q);
          break;
  Case 3: display (q);
          break;
  Case 4: printf (" Exiting....");
          exit (0);
  default: printf (" Invalid choice \n");
  }
}
return 0;
}
```

Output:-

Queue operations:

1. Enqueue 2. Dequeue 3. Display Queue 0. Exit

=> Enter your choice : 1
Enter the element to enqueue : 4
4 enqueued to the queue.

=> Enter your choice : 1
Enter the element to enqueue : 6
6 enqueued to the queue

=> Enter your choice : 1
Enter the element to enqueue : 7
7 enqueued to the queue

=> Enter your choice : 2
4 dequeued from the queue.

=> Enter your choice : 3
Queue : 6 7

Sorting, reverse and concatenation of a SLL

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

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

Struct Node* insertEnd (struct Node* head,
 int value) {
 struct Node* new node = (struct Node*) malloc
   (size of (struct Node));
   newNode -> data = value;
   new Node -> next = NULL;

  if (head == NULL) {
      head = newNode;
  } else {
      struct Node* temp = head;
      while (temp -> next != NULL) {
          temp = temp -> next;
      }
      temp -> next = newNode;
  }
  return head;
}

Void displayList (Struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf ("%d ->", temp -> data);
        temp = temp -> next;
}
```

```
        printf("NULL\n");
}

void sortList (struct Node* head) {
    int swapped, temp;
    struct Node* current;
    struct Node* last = NULL;

    if (head == NULL || head -> next == NULL) {
    }

    do {
        swapped = 0;
        current = head;

while (current -> next != last) {
    if (current -> data > current -> next -> data) {
        temp = current -> data;
        current -> data = current -> next -> data;
        current -> next -> data = temp;
        swapped = 1;
    }
    current = current -> next;
    }
    last = current;
    } while (swapped);
}

Struct Node* reverseList (struct Node* head) {
    struct Node* prev = NULL;
    struct Node* current = head;
    struct Node* next = NULL;
```

```
while (current != NULL) {
    next = current -> next;
    current -> next = prev;
    prev = current;
    current = next;
  }
  return prev;
}

Struct Node* concaterate Lists (struct Node*
  list 1, struct Node* list 2) {
  if (list 1 == NULL) {
    return list 2;
  }

  Struct Node* temp = list 1;
  while (temp -> next != NULL) {
    temp = temp -> next;
  }

  temp -> next = list 2;
  return list 1;
}

int main () {
  Struct Node* list = NULL;
  int choice, value;

  do {
    printf ("\n1. Insert \n 2. Sort \n 3. Reverse
        \n 4. concaterate \n 5. Display \n
        0: Exit \n");
    printf ("Enter your choice: ");
    scanf ("%d", & choice);
```

```
Switch (choice) {
  Case 1:
      Printf (" Enter the value to insert : ");
      scanf ("%d", &value);
      list = insert End (list, value);
      break;
  Case 2:
      shortList (List);
      printf (" List sorted successfully. \n");
      break;
  Case 3:
      list = reverse List (list);
      printf (" List reversed successfully. \n");
      break;
  Case 4:
    {
      Struct Node* list 2 = NULL;
      printf (" Enter values for the second list
            (enter -1 to stop) : \n");
    while (1) {
      scanf ("%d", &value);
      if (value == -1) {
          break;
      }
      list 2 = insert End (list 2, value);
    }
    list = concatenate lists (List, list 2);
    printf (" lists concatenated successfully. \n");
  }
  break;
  Case 5:
      display List (list);
      break;
```

```
    Case 0:
        printf ("Exiting the program. \n");
        break;
    default:
        printf ("Invalid choice. please enter a
                Valid option. \n");
    }
} while (choice != 0);
    return 0;
}
```

Output:-

1. Insert
2. Sort
3. Reverse
4. Concatenate
5. Display
0. Exit

Enter your choice : 1
Enter ~~your~~ the Value to insert : 7

Enter your choice : 1
Enter the Value to insert : 5

Enter your choice : 1
Enter the Value to insert : 3

Enter your choice : 5
7 → 5 → 3 → NULL

Enter your choice : 2
List sorted successfully.

=> Enter your choice : 5
3 -> 5 -> 7 -> NULL

=> Enter your choice : 3
List reversed successfully

=> Enter your choice : 4
Enter the values for the second list (~~Enter~~ Enter -1 to stop):

7
8
-1
Lists concaterated successfully

=> Enter your choice :
5
7 -> 5 -> 3 -> 7 -> 8 -> NULL

SLL Operations

*** Menu ***

1. Add a node at the beginning
2. Add a node at the end
3. Add a node at a specific position
4. Delete a node from the beginning
5. Delete a node from the end
6. Delete a node from a specific position
7. Display the list
8. Exit

=> Enter your option : 1
Enter the data : 2
Inserted at the beginning

=> Enter your option : 2
Enter the data : 3
Inserted at the end

=> ~~Enter your option : 3~~
~~Enter the data : 4~~
~~Inserted at the position~~

=> Enter your option : 3
Enter the position to insert at : 1
Enter the data : 4
Inserted at the position : 1

=> Enter your option : 7
Linked list elements : 2 4 3

Doubley linked list Implementation

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

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

struct Node* create Node (int value)
{
    struct Node* new Node = (struct Node*) malloc (
                                size of (struct Node));
    new Node -> data = value;
    new Node -> prev = NULL;
    new Node -> next = NULL;
    return new Node;
}

void insert at Beginning (struct Node** head, int value)
{ struct Node* new Node = create Node (value);
  if (*head == NULL) {
    *head = new Node; }
  else {
    new Node -> next = *head;
    (*head) -> prev = new Node;
    *head = new Node;
  }
}
```

```
void insertToLeft(struct Node** head, int value,
                  int target value){
    struct Node* newNode = creatoNode(value);
    struct Node* current = *head;

    while (current != NULL && current->data !=
                                target value){
        current = current->next; }

    if (current == NULL) {
        printf("Target node with value %d not found.\n",
                                target value);
        free(newNode);
        return;
    }

    newNode->next = current;
    newNode->prev = current->prev;
    if (current->prev != NULL){
        current->prev->next = newNode;
    } else
    { *head = newNode; }
    current->prev = newNode;
    }

void deleteNode(struct Node** head, int target value)
{
    struct Node* current = *head;

    while (current != NULL && current->data !=
                                target value)
{   current = current->next; }
    if (current == NULL){
        printf("Node with value %d not found.\n",
```

```
target value);
  return; }

if (current -> prev != NULL) {
  current -> prev -> next = current -> next;
  } else
  {
    * head = current -> next;
    }
  if (current -> next != NULL) {
    current -> next -> prev = current -> prev;
    }

  free (current);
  printf ("Node with value %d deleted
          successfully. \n", target value);
    }

void displayList (struct Node* head) {
  printf ("Doubly Linked list :");
  while (head != NULL) {
    printf ("%d", head -> data);
    head = head -> next;
    }
  printf ("\n");
}

int main () {
  struct Node* head = NULL;
  int choice, value, target value;

  do {
    printf ("\n 1. Insert at the beginning \n");
    printf (" 2. Insert to the left of a node \n");
```

```
printf("3. Delete a node \n");
printf("4. Display the List \n");
printf("5. Exit \n");
printf("Enter your choice");
scanf("%d, &choice);

switch (choice) {
 Case 1:
   printf("Enter the value to insert: ");
   scanf("%d", &value);
   Insert At Beginning (&head, value);
   break;
 Case 2:
   printf("Enter the value to insert: ");
   scanf("%d", &value);
   printf("Enter the value of the target node:");
   scanf("%d", &target Value);
   Insert to left (&head, value, target value);
   break;                                          (A)
 Case 3:                                           (B)
   printf("Enter the value of the node to delete:");
   scanf("%d", &target value);                     A=>
   delete Node (&head, target Value);              B=>
   break;
 Case 4:                                           A=>
   display List (head);                            B=>
   break;
 Case 5:                                           A=>
   printf("Exiting the program. \n");              B=>
   break;
 default:
   printf("Invalid Choice. please enter            A=>
        a valid option. \n"); }
}
```

```
while (choice != 5);
  struct Node* temp;
  while (head != NULL) {
    temp = head;
    head = head -> next;
    free (temp);
  }
  return 0;
}
```

Output :-

1. Insert at the Begining
2. Insert to the left of a node
3. Delete a Node
4. Display the List
5. Exit

(A) Enter your choice : 1
(B) Enter the Value to Insert : 8

=> 1
=> 6

=> 1
=> 4

=> 2
> 3
Enter the Value of the target Node : 6

4
Doubly Linked List : 4 3 6 8

Enter your choice: 3
Enter the value of the node to delete : 6
Node with value 6 deleted successfully.

Enter your choice : 4
Doubly Linked List : 4 3 8

Enter your choice : 5
Exiting the program

executed

stop

Score of parentheses (Leet code program)

```
#include <stdio.h>

int score of parentheses (char* s){
  int score = 0;
  int depth = 0;

for(int i = 0; s[i] != '\0'; i++){
  if(s[i] == '(') {
     depth++;
  } else {
     depth--;
     if ( s[i-1] == '(') {
        score += 1 << depth;
     }
  }
}

  return score;
}

int main () {
  char s[51];
  printf ("Enter the balanced parentheses string :");
  scanf("%s", s);
  printf (~~"%s", s~~, score : %d\n", score of parentheses(s));
  return 0;
}
```

Binary Search Tree

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

Struct TreeNode {
    int val;
    struct TreeNode* left;
    struct TreeNode* right;
};

Struct TreeNode* newNode (int data) {
    struct TreeNode* Node = (struct Tree node*)
                malloc (sizeof (struct
                tree node));

    node -> val = data;
    node -> left = NULL;
    node -> right = NULL;
    return node;
}

Struct TreeNode* insert (struct TreeNode* node,
                int Key) {
    if (node == NULL)
    return newNode (Key);

    if (Key < node -> val)
        node -> left = insert (node -> left, Key);

    else if (Key > node -> val)
        node -> right = insert(node -> right, Key);

    return node;
}
```

```
Void inorder (struct Tree node* node) {
    if (node != NULL) {
        inorder (node -> left);
        printf ("%d", node -> val);
        inorder (node -> right);
    }
}

Void postorder (struct Tree Node* node) {
    if (node != NULL) {
        postorder (node -> left);
        postorder (node -> right);
        printf ("%d", node -> val);
    }
}

Void preorder (struct Tree Node* node) {
    if (node != NULL) {
        printf ("%d", node -> val);
        preorder (node -> left);
        preorder (node -> right);
    }
}

Void display (struct Tree Node* root) {
    printf ("Inorder traversal: ");
    inorder (root);
    printf ("\n");

    printf ("Postorder traversal: ");
    postorder (root);
    printf ("\n");
```

```
printf ("preorder traversal : ");
 preorder (root);
 printf ("\n");
}

int main () {

    struct TreeNode * root = NULL;
    int num-nodes , key;

    printf ("Enter the number of nodes in the
                                    BST : ");
    scanf ("%d", & num-nodes);

    printf ("Enter the elements of the BST : \n");
    for (int i = 0; i < num_nodes ; i++) {
        scanf ("%d", & key);
        root = insert (root, key);
    }

    printf ("Binary search Tree elements : \n");
     display (root);

    return 0;
}
```

Output:

Enter the number of nodes in the BST: 5
Enter the elements of the BST:
4
2 3
6
1
8
Binary Search Tree elements:
Inorder traversal: 1 4 6 8 2 3
Post order traversal: 1 8 6 2 3 4
Pre order traversal: 4 1 2 3 6 8

Delete the Middle Node of a Linked List

```
Struct ListNode* deleteMiddle (struct ListNode* head)
{
    if (head == NULL)
        return NULL;

    Struct ListNode* prev = (struct ListNode*)
                malloc (size of (struct ListNode));
    prev -> val = 0;
    prev -> next = head;

    struct ListNode* slow = prev;
    struct ListNode* fast = head;

    while (fast != NULL && fast -> next != NULL){
        slow = slow -> next;
        fast = fast -> next -> next;
    }

    Struct ListNode* temp = slow -> next;
    slow -> next = slow -> next -> next;
    free (temp);

    Struct ListNode* newHead = prev -> next;
    free (prev);
    return newHead;
}
```

Odd Even Linked List

```
struct ListNode* oddEvenList (struct ListNode*
                                head) {
    if (!head || !head -> next)
        return head;

struct ListNode* odd_head =
            (struct ListNode*) malloc (sizeof(struct
                                ListNode));
Struct ListNode* odd = odd_head;

struct ListNode* even_head = (struct ListNode*)
            malloc (sizeof (struct ListNode));
Struct ListNode* even = even_head;

int is_odd = 1;
while (head) {
    if (is_odd) {
        odd -> next = head;
        odd = odd -> next;
    } else {
        even -> next = head;
        even = even -> next;
    }
    is_odd = ! is_odd;
    head = head -> next;
}

even -> next = NULL;
odd -> next = even_head -> next;
free (even_head);
```

```
struct ListNode* result = odd->head->next;
free(odd->head);
return result;

}
```

Delete Node in a BST (Leet Code)

```
struct TreeNode* deleteNode (struct TreeNode*
            root, int Key) {
    if (root == NULL)
        return root;

    Struct TreeNode* minValueNode (struct TreeNode*
                node) {
        Struct TreeNode* current = node;
        while (current && current -> left != NULL)
            current = current -> Left;
        return current;
    }

    if (Key < root -> val)
        root -> left = deleteNode (root -> left, Key)
    else if (Key > root -> val)
        root -> right = deleteNode (root -> right, Key);
    else {
        if (root -> left == NULL) {
            Struct TreeNode* temp = root -> right;
            free (root);
            return temp;
        } else if (root -> right == NULL) {
            struct TreeNode* temp = root -> left;
            free (root);
            return temp; }

        Struct TreeNode* temp = minValueNode (root -> right);
        root -> val = temp -> val;
        root -> right = deleteNode (root -> right, temp -> val);
    }

    return root; }
```

26/2/24.

Find Bottom Left Tree value (Leet code)

```
int findbottomLeftvalue (struct TreeNode* root) {
    if (root == NULL)
        return -1;

struct TreeNode** queue = (struct TreeNode**)
        malloc(sizeof(struct TreeNode*) * 10000);
int front = 0, rear = 0, next level count = 0,
    current level count = 1;
int left most value = root -> val;

queue [rear++] = root;

while (front < rear) {
    struct TreeNode* current = queue [front++];
    current Level count --;

if (current -> left != NULL) {
    queue [rear ++] = current -> left;
    next level Count ++; }

if (current -> right != NULL) {
    queue [rear++] = current -> right;
    next Level count ++; }

if (current Level Count == 0) {
    if (next level Count > 0)
        left most value = queue [front] -> val;
    current level count = next level count;
    next level count = 0; }
    }

    free (queue);
    return left most value;
}
```

26/2/2024

Breadth First Search Implementation

```
# include <stdio.h>

void bfs (int a[10][10], int visited[], int n,
                                int source) {
    int queue [10];
    int front = 0, rear = -1;
    queue [++rear] = source;
    visited [source] = 1;

    printf ("Nodes visited from %d = ", source);

    while (front <= rear) {
        int current = queue [front++];
        printf (" %d", current);

        for (int i = 0; i <n; i++) {
            if (a [current][i] && !visited [i]) {
                queue [++rear] = i;
                visited [i] = 1;
            }
        }
    }
    Printf ("\n");
}

int main () {
    int n, a[10][10], visited [10] = {0};
    printf ("Enter the number of nodes : ");
    scanf ("%d", &n);

    printf ("Enter the adjacency matrix : \n");
```

```
    for (int i = 0; i<n; i++){
        for (int j = 0; j<n; j++){
            scanf("%d", &a[i][j]);
        }
    }
    for (int source = 0; source <n; source++){
        if (!visited[source]){
            bfs(a, visited, n, source);
        }
    }

    return 0;
}
```

Output:-

```
Enter number of nodes: 3
Enter the adjacency matrix:
0
1
0
1
1
0
1
1
0
1
1
Nodes visited from 0: 0 1 2
```

NA 26/2/2024

Depth first Search Implementation

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

int a[20][20], s[20], n;

void dfs (int v) {
    int i;
    printf ("%d", v);
    s[v] = 1;
    for (i = 0; i < n; i++){
        if (a[v][i] && !s[i]) {
            dfs (i);
        }
    }
}

int main () {
    int i, j, count = 0;
    printf ("\n Enter number of vertices : ");
    scanf ("%d", &n);
    for (i=0; i<n; i++) {
        s[i] = 0;
        for (j= 0; j <n; j++)
            a[i][j] = 0;
    }
    printf (" Enter the adjacency matrix : \n");
    for (i=0; i<n; i++)
        for (j= 0; j<n; j++)
            scanf ("%d", &a[i][j]);

    printf (" DFS traversal starting from vertex 0: ");
    dfs (0);
```

```
printf("\n");

for (i=0; i<n; i++){
    if (s[i])
        count++;
}

if (count == n)
    printf(" Graph is connected \n");
else
    printf(" Graph is not connected \n");

return 0;
}
```

26/2/24

Output:

Enter number of vertices: 4
Enter the adjacency matrix:
1 1 0 0
0 1 1 0
0 0 1 1
1 0 0 1

DFS traversal starting from vertex 0: 0123
Graph is connected.

Hashing Technique

```
#include <stdio.h>
#include <stdlib.h>
# define SIZE 10

typedef struct {
    int Key;
        } Employee;

typedef struct {
    int Key;
    Employee employee;
        } HashEntry;

typedef struct {
    Hash Entry *table;
    int size;
        } HashTable;

int Hashfunction (int Key, int size) {
    return Key % size;
    }

Void initialiseHashTable (HashTable *ht, int size) {
    ht -> size = size;
    ht -> table = (Hash Entry*) malloc (size * sizeof(
                                HashEntry));
    for (int i = 0; i < SIZE; i++) {
        ht -> table [i]. Key = -1;
    }
}
```

```
void insert (Hash Table *ht, int Key, Employee employee)
{
    int index = hash Function (Key, ht -> size);

    while (ht -> table [index]. Key != -1)
    {
        index = (index + 1) % ht -> size;
    }

    ht -> table [index] . Key = Key;
    ht -> table [index] . employee = employee;
    }

int search (HashTable *ht, int Key) {
    int index = hash Function (Key, ht -> size);
    int original Index = index;
    while (ht -> table [index] . Key != Key &&
            ht -> table [index]. Key != -1)
    {
        index = (index + 1) % ht -> size;
        if (index == original Index)
            return -1;
        }

    if (ht -> table [index] . Key == Key) {
        return index; }
    else {
        return -1;
    }
}
```

```
int main () {
    HashTable ht;
    initialize HashTable(&ht, HT_SIZE);

    int numEmployees;
    printf(" Enter the number of employees : ");
    scanf("%d", &numEmployees);

    for( int i = 0 ; i < numEmployees ; i++) {
        Employee emp;
        printf("Enter Key for employee %d : ", i+1);
        scanf("%d", &emp.Key);

        insert (&ht, emp.Key, emp);
    }


    int searchKey;
    printf(" Enter Key to search : ");
    scanf("%d", &searchKey);
    int resultIndex = search (&ht, searchKey);
    if (resultIndex != -1) {
        printf(" Employee with Key %d found at index
                %d.\n", searchKey, resultIndex); }
    else {
        printf(" Employee with Key %d not found.\n",
                                        searchKey); }

        return 0;
    }
```

Output:

Enter the number of employees: 3
Enter Key for employee 1: 1278
Enter Key for employee 2: 1378
Enter Key for employee 3: 1279
Enter Key to search: 1378
Employee with Key 1378 found at index 9.