DS

INDEX

Data Structure
1BM22CS224

NAME: Rishabh Nomay STD: SEC: ROLL NO:

S. No.	Date	Title	Marks	Teacher's Sign / Remarks
1	21/12/23	Stack Implementation : Pop, Push, Display		
2	28/12/23	Infix to Postfix and Postfix Evaluation		
3	4/1/24	Queue and Circular Queue Implementation		
4	11/1/24	Singly Linked : Insert, Delete, Display		
5	18/1/24	Singly Linked List : Operations at any of the position		
6	1/2/24	Sort, reverse, Concatenate Stack & Queue implementation.		
7	1/2/24	Doubly Linked List		
8	15/2/24	Binary Search Tree		
9	22/2/24	Implementing Graphs and BFS.		
10	29/2/24	Implementing Hash Table		

Lab - 01

21-12-23

(Q) Program to swap numbers using pointers.

```
#include <stdio.h>

void swap (int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main (){
    int a,b;
    printf("Enter a and b: ");
    scanf("%d %d", &a, &b);
    swap (&a, &b);
    printf("After swapping a=%d and b=%d",
           a, b);
}
```

Output

```
Enter a and b: 5 8
After swapping a=8 and b=5
```

(Q) Program to demonstrate Dynamic Memory Allocation

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

int main(){
    int *ptr;
```

```
int size = 5;
ptr1 = (int *) malloc (size * sizeof (int));
if (ptr1 == NULL){
    printf ("Malloc failed");
    return 1;
}
printf ("Malloc Successful");

int *ptr2;
ptr2 = (int *) calloc (size, sizeof (int));
if (ptr2 == NULL)
{
    printf ("Calloc failed");
    return 1;
}
else
printf ("Calloc Successful");

int size2 = 10;
ptr1 = (int *) realloc (ptr1, size2 * sizeof (int));
if (ptr1 == NULL){
    printf ("Realloc Failed");
    return 1;
}
printf ("Realloc Successful");
free (ptr1);
free (ptr2);
}
```

Output :

Malloc Successful
Calloc Successful
Realloc Successful

(c) Program to implement Stack

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

#define MAX 10

struct stack {
    int items[MAX];
    int top = -1;
};

int isFull (struct stack *stack){
    return (stack -> top == MAX-1);
}

int isEmpty (struct stack *stack){
    return (stack -> top == -1);
}

void push (struct stack *stack, int value){
    if (isFull (stack)){
        printf ("Overflow");
    }
    else{
        stack -> top++;
        stack -> items[stack -> top] = value;
```

21/12/2023

```
        printf("Pushed");
    }
}

void pop(struct Stack *stack){
    if(isEmpty(stack)){
        printf("Underflow");
    }
    else{
        stack->top--;
        printf("Poped");
    }
}

void display(struct Stack *stack){
    if(isEmpty(stack)){
        printf("It is empty\n");
    }
    else{
        printf("Element of stack");
        for(int i=stack->top; i>=0; i++){
            printf("%d\n", stack->items[i]);
        }
    }
}
```

```
int main() {

    struct Stack stack;

    push (&stack, 10);
    push (&stack, 20);
    push (&stack, 30);

    display (&stack);

    pop (&stack);
    display (&stack);

    return 0;

}
```

Output:

```
Pushed 10
Pushed 20
Pushed 30

Elements of Stack

30
20
10
Poped the element

20
10
```

DS Lab - 2

28/12/23

(Q) Demonstrate Infix to postfix conversion

```
#include <stdio.h>
#include <ctype.h>
#define MAX_SIZE 100

int precedence(char op) {
    if (op == '+' || op == '-') {
        return 1;
    else if (op == '*' || op == '/')
        return 2;

    return 0;
}

void infixToPostfix(char* exp) {
    char postfix[MAX_SIZE];
    int i, j;
    int stack[MAX_SIZE];
    int top = -1;

    for (i=0, j=0; exp[i] != '\0'; i++){
        if (isalnum(exp[i]))
            postfix[j++] = exp[i];
        else if (exp[i] == '(')
            stack[++top] = exp[i];
        else if (exp[i] == ')') {
```

```
while (top != -1 && stack[top] != '(')
    postfix[j++] = stack[top--];
if (top == -1) {
    printf("Invalid Expression");
    return;
}
top--;
}
else {
    while (top != -1 && precedence(stack[top]
    >= precedence(exp[i]))
    postfix[j++] = stack[top--];
    stack[++top] = exp[i];
}
}
while (top != -1)
if (stack[top] == '(') {
    printf("Invalid");
    return;
    }
    postfix[j++] = stack[top--];
}
postfix[j] = '\0';
printf("Postfix: %s\n", postfix);
}
```

18/1/24

```
int main() {
    char exp[MAX_SIZE];
    printf("Enter Infix: ");
    fgets(exp, sizeof(exp), stdin);
    if (exp[strlen(exp)-1] == '\n')
        exp[strlen(exp)-1] == '\0';
    infixToPostfix(exp);
    return 0;
}
```

Output :-

Enter the infix expression : a+b*c-(d/e+f)

Postfix : abc*+de/f+-

(9) Postfix Evaluation using Stacks

```
#include <stdio.h>
#define MAX 5726

char stack[MAX-5726];
int top = -1;

void push(char ch) {
    stack[++top] = ch;
}

void pop() {
    return stack[top--];
}

int isOperand(char ch) {
    return (ch >= '0' && ch <= '9');
}

int evaluatePostfix(char *postfix) {
    int i, operand1, operand2;

    for(i=0; postfix[i]; i++) {
        if(postfix[i] == ' ' || postfix[i] == '\t')
            continue;
    }
```

NL 18/11/24

```
else if (isoperand (postfix[i])) {
    push (postfix[i] - '0');
}

else {
    operand2 = pop()
    operand1 = pop();

    switch (postfix[i]) {
        case '+': push (operand1 + operand2);
                  break;
        case '-': push (operand1 - operand2);
                  break;
        case '*': push (operand1 * operand2);
                  break;
        case '/': push (operand1 / operand2);
                  break;
    }
}
}
return pop();
}

int main() {
    char postfix[MAX_SIZE];
```

```
printf("Enter postfix:");
gets(postfix);
int res = evaluate_postfix(postfix);
printf("%d", result);
return 0;
}
```

Output:-

Enter postfix : 53*82/+

Result : 16

Test D
R
18/1/24

DS Lab - 3

4/01/24

① Program to Implement Circular queue.

```
#include <stdio.h>
#define SIZE 1000

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()) {
        printf("Can't enqueue, full\n");
        return;
    }
    if (front == -1) front = 0;
    rear = (rear+1) % SIZE;
```

```
    items[rear] = element;
    printf("pushed %d\n", element);
}

int dequeue() {
    if (isEmpty()) {
        printf("can't pop, is empty\n");
        return;
    }
    int element = items[front];
    if (front == rear) {
        front = -1;
        rear = -1;
    }
    else {
        front = (front + 1) % SIZE;
    }
    printf("popped %d\n", element);
    return (element);
}

int main() {
    enqueue(1);
    enqueue(2);
    enqueue(3);

    dequeue();
    dequeue();

    return 0;
}
```

Output :-

```
pushed 1
pushed 2
pushed 3
poped 1
poped 2
```

DS Lab - 4

11/01/24

(Q) Program to implement Singly linked list.

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

struct node {
    int value;
    struct node *next;
};

void displayLinkedList (struct node *p) {
    printf (" Printing ");
    while (p != NULL) {
        printf ("%d\n", p->value);
        p = p->next;
    }
}
```

int main()

```
{
    struct node *head;
    struct node *one = NULL;
    struct node *two = NULL;
    struct node *three = NULL;

    one = malloc (sizeof (struct node));
    two = malloc (sizeof (struct node));
    three = malloc (sizeof (struct node));

    one → value = 1;
    two → value = 2;
    three → value = 3;

    one → next = two;
    two → next = three;
    three → next = NULL;

    head = one;
    displayLinkedList (head);
}
```

Output

Printing Linked List

1
2
3.

9.1
11/1/24

① 1. Create Account
2. Withdraw
3. Deposit
4. Check Balance
5. Exit

Enter your choice: 1
Enter initial deposit amount: 1000
Account credited Successfully.

② Enter the string:
apple
grape
banana

Sorted String
apple
banana
grape

③ Enter the element to search: 5
5 is present.

④ Enter String: Hello World.
Substring: World
World found in String

⑤

(5) Enter the no: : 3
Index: 4

(6) Enter the number : 6
Index: 8

(7) Enter the no: 9
Index: 3

(8) Minimum element : 1
Maximum element : 5

DS - Lab - 5

18/01/24

(a) Singly linked list delete and display implementation.

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

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

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

void deleteNode(struct Node **head, int key) {
    struct Node *current = *head;
    *prev = NULL;
```

```
if (current != NULL && current -> data == key) {
    *head = current -> next;
    free (current);
    return;
}

while (current != NULL && current -> data != key)
{
    prev = current;
    current = current -> next;
}
if (current == NULL) {
    printf ("Key not found \n");
    return;
}
prev -> next = current -> next;
free (current);
}

int main () {
    struct Node * head = createNode (1);
    head -> next = createNode (2);
    head -> next -> next = createNode (3);
    head -> next -> next -> next = createNode (4);

    printf ("Original Linked List: ");
    displayList (head);
```

Executed NULL is passed

```
deleteNode(&head, 3);
printf("Linked list after deleting node
        with value 3: ");
displayList(head);
return 0;
}
```

DS Lab-7

01-02-23

(1) Execute double Linked List.

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

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

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

void InsertToLeftOfNode (struct Node ** head,
    struct Node * target, int data) {
    if (target == NULL) {
        printf ("NULL\n"); return;
    }
    struct Node * newNode = createNode (data);
    newNode -> prev = target -> prev;
    newNode -> next = target;
```

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

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

void deleteNode (struct Node** head, int data) {
    struct Node* current = *head;
    while (current != NULL) {
        if (current -> data == data) {
            if (current -> prev != NULL) {
                current -> prev -> next = current -> next;
            }
            else {
                *head = current -> next;
            }
```

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

int main() {
struct Node* head = NULL;
insertAtBeginning(&head, 3);
display(head);
insertToEndOfNode(&head, targetNode, 4);
deleteNode(&head, 3);
return 0;
}
```

(8) LeetCode - 725 (Split Linked List in Parts).

```
struct ListNode** splitListToParts(struct ListNode*
head, int k, int* returnSize) {
    struct ListNode** ans = (struct ListNode**)
calloc(1, sizeof(struct ListNode*) * k);
```

```
struct ListNode * prev;
int base, len = 0, part = 0;
for (struct ListNode *tmp = head; tmp; tmp = tmp->next) {
    len ++
}
base = len/k;
for (int i = len%k; i > 0; i--) {
    ans[part] = head;
    part ++;
}
for (int i=0; i < (base+1); i++) {
    prev = head;
    head = head->next;
}
prev->next = NULL;
if (base) {
    for (int i = part; i<k; i++) {
        ans[part] = head;
        part++;
        for (int j = 0; j < base; j++) {
            prev = head;
            head = head->next;
        }
    }
}
```

1/2/24

DS Lab-6

01-02-24

1) Implement concatenate, sort and reverse of single linked list

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

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

struct LinkedList {
    struct Node *head;
};

void SortList(struct LinkedList *List) {
    int swapped, temp;
    struct Node *current;
    struct Node *last = NULL;

    do {
        swapped = 0;
        current = list->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;
```

Ended 1/2/2024

```
        }
        current = current -> next;
    }
    tail = current;
}
}while(swapped);
}

void reverseList(struct LinkedList * list){
    struct Node *prev = NULL, * current =
    list -> head, * next Node;
    while (current != NULL){
        next Node = current -> next;
        prev = current;
        current = next Node;
    }
}

void Concatenate list (struct LinkedList * list1,
                       struct LinkedList * list2) {
    if (list1 -> head == NULL) {
        list1 -> head = list2 -> head;
    }
    else {
        struct Node * current = list1 -> head;
        while (current -> next != NULL) {
            current = current -> next;
        }
    }
```

DS - Lab 8

15-02-24

(1) To construct binary search tree, traverse and display the elements

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

struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};

struct Node *createNode(int data) {
    struct Node *newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

struct Node *insert(struct Node *root, int data) {
    if (root == NULL) {
        root = createNode(data);
    }
    else if (data == root->data) {
        root->left = insert(root->left, data);
    }
}
```

NLP 5/2/24

```
else {
    root->right = insert(root->right, data);
}
return root;
}

void inOrder(struct Node *root) {
    if (root != NULL) {
        inOrder(root->left);
        printf("%d", root->data);
        inOrder(root->right);
    }
}

void preOrder(struct Node *root) {
    if (root != NULL) {
        printf("%d", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

void postOrder(struct Node* root) {
    if (root != NULL) {
        postOrder(root->left);
        postOrder(root->right);
        printf("%d", root->data);
    }
}
```

```
void display (struct Node * root){
    printf ("In-order");
    inOrder (root);
    printf ("Pre-order");
    preOrder (root);
    printf ("Post-order");
    postOrder (root);
}

int main(){
    struct Node* root = NULL;
    root = insert (root, 10);
    root = insert (root, 5);
    root = insert (root, 15);
    root = insert (root, 7);
    root = insert (root, 3);
    root = insert (root, 12);

    display (root);

    return 0;
}
```

Output:

Inserted:	10	5	15	7	3	12
In-Order:	3	5	7	10	12	15
Pre-Order:	10	5	3	7	15	12
Post-Order:	3	7	5	12	15	10

(c) Leet Code (61), (Rotate List)

```
int GetLength (Struct ListNode * head)
{ if (head == NULL) {
    return 0;
  }
return 1 + GetLength (head->next);
}
Struct ListNode * rotateRight ( Struct ListNode * head,
                                int k ){
  if (head == NULL || k == 0){
      return head;
  int length = GetLength (head);
  if (length == 1)
      return head;
  for (int i=0 ; i < k%length ; i++) {
  Struct ListNode *p = head;
  while (p->next->next != NULL)
  {
    p = p->next;
```

```
struct ListNode *a = (struct ListNode *)
            malloc (sizeof (struct ListNode));
    a->val = b->next->val;
    a->next = head;
     head = a;
     b->next = NULL
  }
 return head;
}
```

DS - Lab 9

22/02/24

(9) Implementing graph, and bfs & dfs traversal.

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

int adjacencyMatrix[MAX][MAX];

void addEdge(int start, int end){
    adjacencyMatrix[start][end] = 1;
    adjacencyMatrix[end][start] = 1;
}

void bfsTraversal(int numVertices, int startVertex){
    int queue[MAX];
    int front = 0, rear = 0;
    int visited[MAX] = {0};

    visited[startVertex] = 1;
    queue[rear++] = startVertex;

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

        for(int i = 0; i < numVertices; i++){
            if(adjacencyMatrix[currentVertex][i] && !visited[i])
```

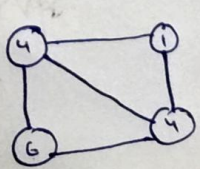
```
        visited[i] = 1;
        queue[rear++] = i;
      }
    }
  }
}

int isConnected(int numVertices) {
  int visited[MAX] = {0};

  void dfsUtil(int vertex) {
    visited[vertex] = 1;

    for (int i = 0; i < numVertices; i++) {
      if (adjacencyMatrix[vertex][i] && !visited[i]) {
        dfsUtil(i);
      }
    }
  }

int main() {
  int numVertices, startVertex;
  printf("Enter the no of vertices: ");
  scanf("%d", &numVertices);
```

```
printf("Enter the adjacency matrix:\n");
for(int i=0; i<numVertices; i++){
    for(int j=0; j<numVertices; j++)
        scanf("%d", &adjacencyMatrix[i][j]);
}
printf("\nEnter the starting vertex for BFS traversal: ");
scanf("%d", &startVertex);
printf("BFS Traversal starting from vertex %d:", startVertex);
bfsTraversal(numVertices, startVertex);
if(isConnected(numVertices))
    printf("\nThe graph is connected.\n");
else
    printf("\nThe graph is not connected.\n");
return 0;
}
```

Output :-

```
Enter the no. of vertices: 2
Adjacency Matrix: 4 1 6 4
Enter the starting vertex: 2
BFS Traversal from vertex 2: 2
The graph is connected.
```

$\begin{bmatrix} 4 & 1 \\ 6 & 4 \end{bmatrix}$



(10) HackerRank — Swap nodes (Algos)

```
void swap_nodes_at_level(struct node *root, int inc,
int level, int height) {

    struct node *tnode;
    if (!root) {
        return;
    }
    if (level > height) {
        return;
    }
    if (!(level % inc)) {
        tnode = root->left;
        root->left = root->right;
        root->right = tnode;
    }
    swap_nodes_at_level(root->left, inc, level+1, height);
    swap_nodes_at_level(root->right, inc, level+1, height);
}
```

Sol:
22/2/24

DS lab-10

29/02/24

(1) Implementation of Hash Table.

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

int hash_table[Table_size];

void initialize EmployeTable() {    // Initialising Employee Table
    for (int i=0; i<Table_size; i++) {
        hash_table[i] = -1;
    }
}

void insert Employee (int key) {
    int hkey = hash(key);
    int index = hkey;
    int i=0;

    while (hash_table[index] != -1) {
        i++;
        index = (hkey+i) % Table_size;
        if (i == Table_size) {
            printf("Full, Cant Insert");
            return;
        }
    }
    hash_table[index] = key;
}
```

```
void search(int key) {
    int hkey = hash(key);
    int index = hkey;
    int i = 0;
    while (hash_table[index] != key) {
        i++;
        index = (hkey + i) % Table_Size;
        if (hash_table[index] == -1 || i == Table_Size) {
            printf("Key %d not found", key);
            return;
        }
    }
    printf("Key %d found\n", key);
}

void displayTable() {
    printf("Hash Table :\n");
    for (int i=0; i< Table_Size; i++) {
        printf("%d : ", i);
        if (hash_table[i] != -1) {
            printf("%d", hash_table[i]);
        }
```

```
printf("\n");

}

int main() {

    initializeHashTable();

    insert(12);
    insert(25);
    insert(35);
    insert(26);
    insert(4);

    displayHashTable();

    search(35);
    search(26);
    search(80);

    return 0;

}
```

Output:

Hash Table:

Emp 0:
Emp 1: 4
Emp 2: 12
Emp 3:
Emp 4:
Emp 5: 25
Emp 6: 35
Emp 7: 26
Emp 8:
Emp 9:

Key 35 found at index 6

Key 80 not found.